Vesuf, eine modellbasierte User Interface Entwicklungsumgebung für das Ubiquitous Computing, vorgestellt anhand der Fallstudie PublicationPORTAL

Diplomarbeit

Lars Braubach und Alexander Pokahr

vorgelegt am 10. Dezember 2001 bei

Prof. Dr. Winfried Lamersdorf
Arbeitsgruppe Verteilte Systeme
und
Dr. Daniel Moldt
Arbeitsbereich Theoretische
Grundlagen der Informatik

am Fachbereich Informatik der
Universität Hamburg

Zusammenfassung

Das Zeitalter des „Ubiquitous Computing" – der uns allgegenwärtig umgebenden vernetzten Computer – bringt Veränderungen mit sich. Diese Veränderungen betreffen sowohl die Art und Weise wie wir in Zukunft mit Computern umgehen als auch die Methodik und die Systeme, die wir zur Erstellung von Applikationen einsetzen werden. Neben des notwendigen Aufbaus einer Infrastruktur für das Ubiquitous Computing bringt insbesondere auch die Konstruktion von Benutzungsschnittstellen für Anwendungen in diesem Umfeld eine Reihe neuer Anforderungen mit sich. Diese Arbeit legt den Fokus der Betrachtung auf die systematische Erstellung von User Interfaces für Applikationen im Kontext des Ubiquitous Computing.

Mit Hilfe der im Rahmen dieser Arbeit konzipierten und realisierten modellbasierten User Interface Entwicklungsumgebung (MB-UIDE) ist es möglich, Applikationen auf einfache deklarative Weise mit verschiedenen User Interface Modalitäten auszustatten. Die Beschreibung des User Interfaces in verschiedenen Teilmodellen erlaubt es, für jede Modalität eine optimale Schnittstelle zu konstruieren, ohne die Fachlogik modifizieren zu müssen. Die Autoren haben die praktische Einsatzfähigkeit des Vesuf Systems nachgewiesen, indem sie exemplarische heterogene Dienste in ein Internetportal integriert haben.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Ubiquitous Computing 1
1.2 Portale 4
1.3 Zielsetzung 5
1.4 Gliederung 5

2 Architekturen, Techniken und Werkzeuge **7**
2.1 Referenzarchitekturen 8
2.1.1 Das Seeheim Modell 8
2.1.2 Model-View-Controller (MVC) 11
2.1.3 Presentation-Abstraction-Control (PAC) 13
2.1.4 Arch / Slinky 14
2.1.5 PAC-Amodeus 18
2.1.6 Visual Proxy 20
2.1.7 Hierarchical MVC (HMVC) / Layered MVC 21
2.1.8 Zusammenfassung 23
2.2 Techniken 24
2.2.1 Klassifikation 24
2.2.2 Präsentationstechniken 27
2.2.3 Dialogtechniken 32
2.2.4 Domänentechniken 34
2.2.5 Zusammenfassung 39
2.3 Werkzeuge 39
2.3.1 Übersicht über User Interface Werkzeuge 40
2.3.2 Toolkits 41
2.3.3 Interface Builder 42
2.3.4 Frameworks 43
2.3.5 UIMS 44
2.3.6 Modellbasierte Werkzeuge (MB-UIDEs) 45
2.3.7 CASE Tools 49
2.3.8 Zusammenfassung 53

3 Untersuchte Systeme **55**
3.1 Nicht-deklarative Ansätze 56
3.1.1 MVC-Client 56
3.1.2 SanFrancisco 58
3.1.3 JWAM 60
3.1.4 Model-View-Presenter (MVP) 61

3.1.5 Vergleich der nicht-deklarativen Systeme 64
3.1.6 Weitere Forschungsansätze 64
3.2 Modellbasierte Ansätze 65
3.2.1 Die historische Evolution modellbasierter Systeme 65
3.2.2 Janus/Jade 67
3.2.3 Mobi-D 70
3.2.4 FUSE 73
3.2.5 TRIDENT 75
3.2.6 TADEUS 78
3.2.7 Teallach 80
3.2.8 MASTERMIND 83
3.2.9 Business Component Prototyper für SanFrancisco 85
3.2.10 Vergleich der modellbasierten Systeme 86
3.2.11 Weitere Forschungsansätze 90
3.3 Zusammenfassung 90

4 Vesuf Konzeption 93
4.1 Überblick 93
4.1.1 Modelle und Notationen 94
4.1.2 Werkzeuge der Vesuf Umgebung 96
4.1.3 Laufzeitarchitektur 97
4.2 Systemdetails 98
4.2.1 Beispielanwendung 99
4.2.2 Kopplung der Systemkomponenten 99
4.2.3 Domänenschicht 108
4.2.4 Dialogschicht 118
4.2.5 Präsentationsschicht 120
4.3 Konzeptuelle Probleme und Lösungsansätze 126
4.3.1 Das Abbildungsproblem 126
4.3.2 Das Assoziationsproblem 128
4.4 Ausblick und Erweiterungen 129

5 Fallstudie Global-Info 131
5.1 Global Info 131
5.1.1 PublicationPORTAL 132
5.1.2 Integration von Vesuf in das PublicationPORTAL 133
5.2 Methodologie 133
5.3 Online Dictionaries 134
5.3.1 Domänenmodell 135
5.3.2 Implementation 137
5.3.3 Präsentationsmodelle 141
5.3.4 Dialogsteuerung und Applikationsdeskriptor 149
5.4 Z39.50 Dienste 150
5.4.1 Domänenmodell 151
5.4.2 Implementation 154
5.4.3 Dialogsteuerung 160
5.4.4 Präsentationsmodelle 162
5.4.5 Applikationsdeskriptor 167

6 Zusammenfassung und Ausblick **169**
6.1 Ergebnisse der Forschung 169
6.2 Umgesetzte Zielvorgaben 170
6.3 Fallstudie 171
6.4 Ausblick 172

A VEPL Referenz **173**

Abbildungsverzeichnis **175**

Literaturverzeichnis **177**

Kapitel 1

Einleitung

1.1 Ubiquitous Computing

Die Historie der breiten Nutzung von Computern beginnt mit der Zeit der Grossrechenanlagen (Mainframes), die dadurch charakterisiert ist, dass viele Anwender gemeinsam an einem Gerät arbeiten müssen. Daran schließt sich die Personal Computing Ära an, in der für einen Benutzer jeweils ein (Desktop-) Computer zur Verfügung steht. Das nächste Zeitalter wird, bezugnehmend auf [Weiser 1995], Ubiquitous Computing[1] (UbiComp) genannt. UbiComp ist durch eine Vielzahl computerisierter Endgeräte (Devices) je Anwender geprägt, die in den verschiedensten Ausprägungen auftreten können. Die Technologie wird mehr und mehr in den Hintergrund treten und in unsere Umgebung eingebettet werden. Weiser bezeichnet die nahtlose und natürliche Integration des Computers in unsere Umgebung als ein wichtiges Ziel, das es uns erlaubt, ohne darüber nachzudenken und ohne Mühen computergestützte Dienste in Anspruch zu nehmen. Aus diesem Grund kann UbiComp als Gegenteil von Virtual Reality aufgefasst werden: Virtual Reality versucht, Menschen in eine künstliche Computerwelt zu versetzen, wohingegen UbiComp das Ziel hat, Computer in die reale alltägliche Welt des Menschen zu integrieren [Gallis et al. 2001].

Um diese Vision zu erreichen ist einerseits ein anderes Verständnis der Computer per se und sind andererseits Fortschritte in diversen wissenschaftlichen Disziplinen notwendig. [Banavar et al. 2000] betonen, dass UbiComp solange mehr Kunst als Wissenschaft bleiben wird, wie Menschen erstens mobile Computer als Mini-Desktops auffassen, zweitens Applikationen nur als Programme ansehen, die auf diesen Geräten ablaufen und drittens die Anwendungsumgebung als virtuellen Raum betrachten, der zur Ausführung einer Aufgabe betreten und nach Beendigung wieder verlassen wird. Sie fassen ihre Vorstellungen, wie diese drei Aspekte in Zukunft betrachtet werden sollten wie folgt zusammen:

1. Ein Gerät wird als Portal zu einem Applikations- und Datenraum verstanden, nicht als ein Behältnis benutzerverwalteter Software.

2. Eine Applikation zeichnet sich durch die Aufgaben oder Dienste aus, die sie einem Anwender zur Verfügung stellt. Sie sollte nicht als ein Stück Soft-

[1] Ubiquitous Computing kann mit der Allgegenwärtigkeit von miteinander vernetzten Computern übersetzt werden.

ware betrachtet werden, das die Möglichkeiten eines speziellen Endgeräts ausreizt.

3. Die Umgebung ist das um Informationsangebote erweiterte physikalische Umfeld eines Anwenders, nicht der virtuelle Raum zur Datenhaltung und Programmausführung.

Die technologischen Herausforderungen zur Verwirklichung eines solchen Szenarios liegen auf den Gebieten der Infrastruktur-, der Middleware- und der Applikationsentwicklung. Der Bereich der Infrastruktur muss im Besonderen dafür Sorge tragen, dass die Kommunikation zwischen verschiedenen Geräten bzw. Services reibungslos funktioniert. Gerade für mobile Geräte ist daher die drahtlose Kommunikation (siehe z. B. [Bluetooth SIG 1999]) von herausragender Bedeutung. Im Bereich der Middleware sind komplexe Aufgaben der Diensterkennung und der Dienstvermittlung zu lösen. Diese werden nachfolgend im Zusammenhang mit der Applikationsentwicklung betrachtet. In allen Gebieten muss darauf geachtet werden, Sicherheitsmechanismen zum Schutz persönlicher Informationen zu etablieren.

Die Entwicklung und Bereitstellung von Applikationen für eine UbiComp Umgebung unterscheidet sich grundsätzlich von der Entwicklung herkömmlicher Anwendungen, denn Applikationen existieren nicht mehr isoliert, sondern befinden sich in einem Netzwerk aus mobilen und stationären Endgeräten und können in diesem Netzwerk integrierte Dienste in Anspruch nehmen. Es ergibt sich ein riesiges ad-hoc verteiltes System mit diversen Endgeräten und Diensten, die beständig im System verfügbar werden und wieder verschwinden. [Banavar et al. 2000] präsentieren daher ein neuartiges Applikationsmodell, das diesen neuen Parametern gerecht wird. Sie unterteilen den Lebenszyklus einer Applikation in Design-time, Load-time und Runtime.

Da das Ziel ist, Applikationen zur Bearbeitung von Aufgaben zu entwickeln, muss der Entwickler dieser Tatsache schon während der Erstellung des Programms in vielerlei Hinsicht Rechnung tragen:

- Die Entwicklung sollte geräteunabhängig durchgeführt werden und die Domänenanalyse, bestehend aus Aufgaben- und Objektmodellierung in den Vordergrund rücken (device-independence).
- Des weiteren ist auf eine klare Separation zwischen Applikationslogik und der Benutzungsschnittstelle zu achten. Applikationen des UbiComp müssen in die Lage versetzt werden, auf unterschiedlichen Endgeräten ausgeführt zu werden und eine dem Gerät entsprechende Bedienung anzubieten (user interface / application separation).
- Die Anwendungen sollten derart konzipiert werden, dass sie die die durch die Umgebung angebotenen Dienste nutzen können. (service discovery and brokerage).

Um Applikationen für ein Endgerät bereitzustellen, ist es notwendig festzustellen, welche Anwendungen zur Zeit verfügbar sind und welche Anpassungsmaßnahmen durchzuführen sind, damit das Gerät die Applikation ausführen kann. Da eine Applikation sich durch die Aufgaben auszeichnet, die sie zu erledigen hilft und diese von der physischen Umgebung des Anwenders abhängen können, muss der Ladevorgang dynamisch gestaltet werden:

- Dienste und Applikationen befinden sich in der Umgebung des Endgeräts und müssen durch geeignete Mechanismen identifiziert und verfügbar gemacht werden (dynamic discovery).
- Dem Endgerät müssen sämtliche für die Ausführung einer Applikation notwendigen Dienste und Ressourcen zugänglich gemacht werden. Je nach Ausstattung des Endgeräts sind unterschiedliche Client / Server-Verteilungen angezeigt (requirements and capability negotiation, static apportionment).
- Das Endgerät muss in die Lage versetzt werden, die Applikation in geeigneter Form anzeigen zu können. Dazu muss das richtige User Interface, in Abhängigkeit von Größen- und Modalitätsfaktoren, vom System ausgewählt und evtl. sogar angepasst werden. Die dynamische Integration von kontextabhängigen Diensten in Applikationen macht es außerdem notwendig, Benutzungsschnittstellen von Diensten zu kombinieren. (presentation selection, adaptation and composition).

Zur Laufzeit einer Applikation unterliegt diese diversen Einflüssen der Umgebung. Veränderte Ressourcen, wie z. B. sich verschlechternde Übertragungsraten, neue Dienste oder neue Benutzer müssen reflektiert werden. Ziel ist es, eine hohe Transparenz der Einflüsse zu erreichen, die nicht direkt benutzerrelevant sind. Dazu muss die Anwendung mindestens folgende Merkmale aufweisen:

- Die Applikation muss durch Überwachung der Umgebung Änderungen feststellen können und in geeigneter Weise reagieren. So werden einerseits Migrationsmechanismen zur Reaktion auf sich verändernde Übertragungsleistungen benötigt (dynamic apportionment). Andererseits muss die Applikation auch dynamische Kompositionverfahren beinhalten, um mit transient verfügbaren Diensten umgehen zu können (dynamic composition).
- Die Trennung der Verbindung eines Endgeräts vom System muss von der Applikation angemessen berücksichtigt werden. Sie sollte geeignete Mechanismen einsetzen, um die negativen Folgen einer Verbindungsunterbrechung zu minimieren. (disconnection)
- Die Applikation sollte Strategien zur Fehlererkennung und Wiederherstellung alter Programmstände beinhalten (failure detection and revovery).

Für den Anwender entstehen durch UbiComp Applikationen eine Reihe angenehmer Effekte. Dem Grundgedanken des UbiComp folgend, kann ein Anwender zusätzlich zu den ohnehin auf seinem Endgerät befindlichen Applikationen, Dienste und Anwendungen der Umgebung nutzen. Der Anwender nimmt damit die Applikationen als Services wahr, die ihm abhängig von seinem Kontext angeboten werden. Zu diesem Kontext gehört nicht nur sein Aufenthaltsort, sondern viele weitere Faktoren, wie z. B. das aktuelle Wetter, Tageszeit, Wochentag, Art der gerade ausgeübten Tätigkeit. Die Applikationen werden mehr und mehr kontextsensitiv reagieren (context-awareness) und können sich dadurch individuell auf den jeweiligen Benutzer einstellen.

UbiComp wird auch dazu führen, dass Applikationen geräteunabhängig entwickelt werden müssen. Dadurch kann dieselbe Applikation von verschiedenen

Endgeräten aus genutzt werden und der sonst unvermeidliche Abgleich von Daten, die mit verschiedenen Applikationen des gleichen Typs bearbeitet wurden, entfällt. Neben dem Vorteil synchroner Daten wird auch der Programmzugriff vereinfacht, da er stets demselben Muster folgt. Natürlich sind die Benutzungsschnittstellen je nach Endgerät unterschiedlich, dennoch müssen die gleichen Aufgaben ausgeführt werden, um ein bestimmtes Ziel zu erreichen.

Die Realisierung von Aufgaben als dynamische Services der Umgebung und nicht als statische Software führt auch dazu, dass die Software stets in der aktuellen Version verfügbar ist und aufwändige Softwareupgrades entfallen.

1.2 Portale

Wie bereits eingangs angesprochen, werden Endgeräte zukünftig stärker als Portale zu Applikationen und Daten gesehen werden, die in der Umgebung des Endgeräts durch Server zur Verfügung gestellt werden. Bis Endgeräte aber in der Lage sind, die komplexen Portalbausteine selbständig zu verwalten, sind noch grosse Fortschritte in den oben aufgezeigten Disziplinen notwendig. Des weiteren verfügen viele auf dem Markt erhältliche mobile Geräte, wie z. B. PDAs noch nicht über die notwendigen Ressourcen, um Portalsoftware zu verwalten.

Heutzutage werden Portaltechnologien für Internetserver entwickelt, die dann (meist) über eine einfache Webschnittstelle vorher eingepflegte Applikationen und Dienste zur Verfügung stellen. Diese Internetportale zeichnen sich dadurch aus, dass sie dem Benutzer skalierbare Dienste zur Verfügung stellen und es ihm erlauben seinen Zugang zu personalisieren. Grundsätzlich wird zwischen vertikalen und horizontalen Portalen unterschieden. Vertikale Portale vertiefen ihr Angebot um einen Themenkreis, indem sie z. B. neben Artikeln zu einem Thema auch Foren und Neuigkeiten bereitstellen. Horizontale Portale bieten eine breite Angebotspalette, die viele Services und Themen abdeckt [Bartelt et al. 2001].

Global Info

Das Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (bmb+f) initiierte das Förderkonzept "Globale Elektronische und Multimediale Informationssysteme" (Global Info) mit dem Ziel der "Entwicklung integrierter wissenschaftlicher Informationssysteme, die alle Aspekte einer digitalen Bibliothek gleichermaßen umfassen: von der Erzeugung elektronischer wissenschaftlicher Informationen, über deren Verbreitung bis hin zu der Art und Weise wie nach ihnen gesucht wird und sie schließlich benutzt werden" [Global Info 2000]. Im Global Info Projekt "Infrastukturen für digitale Bibliotheken" beschäftigt sich das in der Arbeitsgruppe Verteilte Systeme (VSYS) des Informatikfachbereiches der Universität Hamburg durchgeführte Subprojekt "The Global Info Brokerage And Library Trading Architecture" (GIBRALTAR) [VSYS 2000] damit, wie über ein Netzwerk (z. B. das Internet) verstreute Dienste gefunden und in einem Portal zusammengefasst werden können, mit dem Ziel eine "Vermittlungskomponente für den entfernten Zugang zu globalen verteilten Informationsdiensten" zu entwickeln.

Die im Zuge des GIBRALTAR Projekts entwickelten Komponenten werden in das neu aufgesetzte PublicationPORTAL integriert. Dieses wurde als vertikales Portal ausgelegt, das für verschiedene Akteure als Zugangspunkt zu diversen

ePublishing Services dienen soll. Für die Dienstvermittlung und Komposition wurde eine Typemanager-Komponente [Häming 2000] realisiert und in das Portal eingebunden. Benötigt wurde eine Benutzungsschnittstellenkomponente, mit der heterogene Dienste mit einem einheitlichen Erscheinungsbild versehen und in das Portal aufgenommen werden können. Des weiteren soll diese Komponente die Technologie bereitstellen, mit der die Dienste des Portals auch über andere Endgeräte, z. B. ein WAP-Handy erreicht werden können.

1.3 Zielsetzung

Gegenstand dieser Arbeit ist die Konzeption und Realisierung eines Systems, mit dem die flexible geräteunabhängige Entwicklung von User Interfaces für Applikationen im Kontext des UbiComp möglich wird. Dieses System kann als Teil einer umfassenderen Portalstrategie verstanden werden, die weitere Komponenten für Personalisierungs- und Dienstbrokerageaufgaben bereitstellen muss, um UbiComp in grundlegender Form zu erlauben.

Die Autoren identifizieren im Folgenden Anforderungen für ein User Inferface Konstruktionssystem des UbiComp. Grundvoraussetzungen für jedes derartige System sind: (1) Der funktionale Applikationskern und das User Interface werden vollständig getrennt. Die Entwicklung der beiden Teile kann damit separat durchgeführt werden. Diese Separation impliziert, dass auch für die Verbindung der beiden Teile ein einfacher Mechanismus gefunden werden muss, der keinen grossen Aufwand mit sich bringt. (2) Das System vereinfacht die Entwicklung der fachlichen Logik und des User Interfaces.

Darüber hinaus gewinnen folgende Punkte speziell im Kontext des UbiComp stark an Bedeutung: (3) Das System ist erweiterbar. Es ist sowohl möglich, dem System neue User Interface Modalitäten zur Unterstützung weiterer Endgeräte hinzuzufügen als auch die fachliche Logik von Altsystemen, die in verschiedenen Implementationstechniken vorliegen kann, anzubinden. (4) Das System besitzt eine hohe Flexibilität. Für eine Applikation können beliebig viele verschiedene User Interfaces ohne Aufwand definiert und integriert werden. Ebenso kann ein Interface mit verschiedenen Implementationen einer Applikation verbunden werden. (5) Die Adaption des User Interfaces bezüglich verschiedener Aspekte sollte vom System berücksichtigt werden. Adaption im Hinblick auf das Endgerät sollte dessen spezielle Merkmale, wie z. B. die Bildschirmgröße für Anpassungsmaßnahmen heranziehen. Des weiteren kann Adaption situationsbedingt und auf Grundlage der Fähigkeiten des Benutzers durchgeführt werden. (6) Das System sollte die dynamische Komposition von User Interfaces möglich machen.

Die Tauglichkeit des Systems wird durch einen Praxiseinsatz untermauert. Die Autoren setzen das System im Kontext des Global Info Projekts ein, indem sie es in das Portal integrieren und exemplarische Dienste einbinden.

1.4 Gliederung

Im Anschluss an dieses Kapitel werden die Autoren zunächst untersuchen, welche Architekturen, Techniken und Werkzeuge es zur User Interface Entwicklung gibt und welche Ansätze besonders vielversprechend für die Realisierung unserer Zielsetzungen erscheinen. In Kapitel 3 werden diverse User Interface Frameworks

und modellbasierte Systeme vorgestellt und auf ihre Eignung in Bezug auf Anforderungen des UbiComp bewertet. Die Autoren präsentieren in Kapitel 4 das im Kontext dieser Arbeit neu entwickelte modellbasierte System „Vesuf." Anhand einer exemplarischen Dienstintegration heterogener Internetdienste in das PublicationPORTAL in Kapitel 5 wurde im Kontext des Global-Info Projektes die Praxistauglichkeit des Vesuf Systems nachgewiesen. In Kapitel 6 evaluieren die Autoren, inwieweit die Zielsetzungen erreicht wurden und beleuchten in welcher Richtung Erweiterungen vorstellbar sind.

Kapitel 2

Architekturen, Techniken und Werkzeuge

Benutzungsschnittstellen machen einen nicht unerheblichen Teil des Gesamtaufwands eines Softwareprojektes aus. In [Myers Rosson 1992] wurde statistisch ermittelt, dass durchschnittlich 48% des Programmcodes auf das User Interface entfallen. Außerdem ist zu erfahren, dass auch der Zeitaufwand erheblich ist, nämlich 45% innerhalb der Designphase, 50% in der Implementationsphase und immerhin noch 37% in der Wartungsphase.

Hinzu kommt, dass User Interfaces nicht nur aufwändig, sondern auch inhärent schwierig zu entwerfen und zu implementieren sind. Das ist unter anderem auf die nicht ausreichenden Designstrategien und auf den an sich schon problematischen iterativen Ansatz zurückzuführen. Ausführlich begründet wird die Problematik in [Myers 1993].

Die Entwicklung eines User Interfaces ist in der Folge für die Industrie ein gewichtiger Kostenfaktor des Softwareengineeringprozesses. Bei der Applikationsentwicklung im Kontext des Ubiquitous Computing kommt erschwerend hinzu, dass Benutzungsschnittstellen für verschiedene Interfacemodalitäten entwickelt werden müssen. Um die Kosten zu reduzieren und wissenschaftliche Grundlagen zu schaffen, wurden zahlreiche Projekte initiiert. Ergebnis dieser Aktivitäten sind diverse Werkzeuge, wiederverwendbare Software (Klassenbibliotheken, Frameworks und Toolkits), Techniken, Methodologien und Software-Architekturen [Kazman Bass 1996].

In diesem Kapitel werden zahlreiche Vertreter und Klassen dieser Artefakte untersucht, wobei besonderes Augenmerk darauf gelegt wird, inwieweit diese geeignet sind, die in Abschnitt 1.3 gesetzten Ziele zu unterstützen. Um die Einordnung dieser Ergebnisartefakte in Gruppen zu ermöglichen, schlagen die Autoren eine grobe Kategorisierung nach Abstraktionsniveau vor. Wir subsumieren unter der Kategorie Architekturen alle Artefakte, die auf einem sehr hohen Abstraktionsgrad wiederkehrende Designentscheidungen systematisieren. In die Gruppe der Techniken ordnen die Autoren Artefakte ein, die gezielt die Beschreibung bestimmter User Interface Aspekte formalisieren. Zur Kategorie der Werkzeuge rechnen die Autoren alle Artefakte, die dem Entwickler ein konkretes Stück Software in die Hand geben, um User Interfaces zu gestalten. Grundlage der meisten Werkzeuge sind Artefakte der Technikebene. Die Grenzen zwischen den

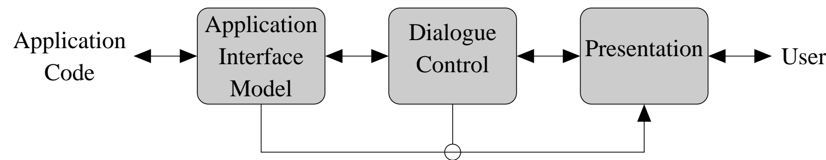


Abbildung 2.1: Seeheim Modell für UIMS-Architekturen (nach [Green 1985])

einzelnen Kategorien sind natürlicherweise fließend und die Zuordnung einzelner Elemente ist daher ist nicht immer eindeutig.

2.1 Referenzarchitekturen

Die Schwierigkeiten bei der Erstellung von Benutzungsschnittstellen haben, wie schon erwähnt, unter anderem die Entwicklung diverser Werkzeuge, wiederverwendbarer Softwarekomponenten und Software-Architekturen motiviert. In der Liste dieser Artefakte spielen die Architekturen eine zentrale Rolle. Werkzeuge und Softwarekomponenten reflektieren die Entscheidungen, die auf dem Architekturlevel gemacht werden. Aufgrund ihrer Bedeutung sind Architekturen für Benutzungsschnittstellen spätestens seit 1980 im Fokus von Forschung und Entwicklung.

In [Machiraju 1996] werden wesentliche Eigenschaften guter User Interfaces vorgestellt. Dies sind neben den in Abschnitt 1.3 als Kernpunkte eines UbiComp Systems identifizierten Eigenschaften Isolation und Adaptivität auch Funktionalität, Benutzbarkeit, Konsistenz und Standards. Diese Liste ist keineswegs vollständig, da zusätzlich Eigenschaften wie Performance und Skalierbarkeit für gute Benutzungsschnittstellen essentiell sind. In Softwarearchitekturen manifestieren sich Designentscheidungen, die auf höchstem Level in einem System gemacht werden, um eine Menge von erwünschten Qualitätszielen zu erreichen. Benutzbarkeit steht in keinem engen Zusammenhang zur Systemarchitektur und wird eher durch kontinuierliches Usability-Testing sichergestellt. Weitere nichtfunktionale Qualitätsziele wie Isolation und Skalierbarkeit sind hingegen auf der architektonischen Ebene von besonderer Bedeutung. Tatsächlich beeinflussen diese Ziele eine Systemarchitektur mehr als die Umsetzung einer speziellen Funktionalität [Kazman Bass 1996].

Architektonisches Modellieren findet in einer Reihe von Schritten statt: Zuerst wird die vom System erwartete Funktionalität identifiziert und die Struktur des Systems definiert. Danach wird die Funktionalität auf die definierten Komponenten verteilt. Zuletzt muss die Koordination zwischen den Komponenten, also das dynamische Verhalten des Systems, beschrieben werden. Referenzarchitekturen bieten ein Framework, um diese Schritte zu leiten [Calvary et al. 1997].

Im Folgenden betrachten die Autoren einige der verbreitesten und meistbeachtesten Referenzarchitekturen für interaktive Softwaresysteme.

2.1.1 Das Seeheim Modell

Das Seeheim Modell für UIMS-Architekturen (Abb. 2.1) wurde entwickelt auf einem UIMS-Workshop in Seeheim 1983 (siehe [Green 1985]). Es soll nicht dazu dienen, die Struktur oder Implementation eines User-Interfaces zu beschreiben,

sondern gliedert Komponenten nach ihren Aufgaben in einem UIMS (User Interface Management System, siehe Abschnitt 2.3.5). Diese Komponenten haben eine unterschiedliche Funktion, weshalb verschiedene Beschreibungs- und Implementationstechniken für diese Komponenten benötigt werden. Das Seeheim Modell identifiziert drei wesentliche Aufgaben für Komponenten einer Benutzungsschnittstelle:

Application Interface Die Anwendungsschnittstelle, auch semantisches Interface genannt, stellt die Funktionalität der Anwendung zur Verfügung, indem sie auf den Anwendungscode zurückgreift. Dabei stellt die Anwendungsschnittstelle das System aus Sicht des Benutzers dar und abstrahiert so von der internen Struktur der Implementation. Stattdessen werden Objekte des Systems so dargestellt, wie es den Vorstellungen entspricht, die der Benutzer von der Anwendungsdomäne hat. Auch Aufgaben, die der Benutzer durchführen will, sind im System oft als eine Reihe von bereitgestellten Operationen zusammengesetzt. Die Anwendungsschnittstelle fasst diese Operationen so zusammen, dass sie dem Benutzer als Einheit präsentiert werden können. Das Application Interface wird von den anderen beiden Komponenten benutzt. Bevor Eingaben an das System weitergegeben werden, können sie von der Application Interface Komponente validiert werden. Fehler werden so frühzeitig erkannt und dem Benutzer mitgeteilt. Auch eine Undo Funktionalität wird von Green in dieser Komponente angesiedelt.

Dialogue Control Die Dialogsteuerung ist als zentrale Komponente zuständig für die Ablaufsteuerung zwischen und innerhalb von Dialogen. Sie definiert die Struktur des Dialogs zwischen System und Benutzer, d. h. welche Eingaben der Benutzer zu welchem Zeitpunkt machen kann. Dabei verwaltet sie den Zustand des User-Interfaces und welche Ereignisse zu welchen Folgezuständen der Benutzungsschnittstelle führen. Ereignisse können dabei sowohl vom Benutzer über die Präsentationskomponente ausgelöst werden als auch von der Anwendung über die Anwendungsschnittstelle.

Presentation Die Präsentationskomponente ist zuständig für die physische Repräsentation (z. B. Position, Farbe und Typ von Widgets). Dazu gehört auch, dass Daten der Anwendung zwischen externen und internen Formaten übersetzt werden (z. B. muss ein Datum, dass der Benutzer als Text eingibt in ein applikationsinternes Datumsobjekt umgewandelt werden, und zur späteren Anzeige zurück in eine textuelle Repräsentation). Dazu muss die Präsentationskomponente nicht nur den Wert der Daten kennen (`"09.09.1999"`) sondern auch deren Typ (Datum). Auch Ereignisse müssen zwischen Benutzer und System übersetzt werden (z. B. das Klicken eines Buttons in einen Methodenaufruf).

Die drei Teile des Modells entsprechen den semantischen (application interface), syntaktischen (dialogue), und lexikalischen (presentation) Aspekten der Benutzerinteraktion, was dem User-Interface einen konversationalen Charakter verleiht.

In [Green 1985] werden verschiedene Beschreibungstechniken zur Darstellung der einzelnen Komponenten erläutert. Dabei wird insbesondere herausgestellt, dass aufgrund der unterschiedlichen Funktionen der Komponenten am

besten unterschiedliche Techniken zur Beschreibung der einzelnen Komponenten eingesetzt werden sollten. Die Autoren werden dieses Thema in Abschnitt 2.2 wieder aufgreifen, wo sie Techniken zur Entwicklung von Domänen-, Präsentations- und Dialogsteuerungskomponenten untersuchen.

Anmerkungen

Einer der Hauptkritikpunkte am Seeheim Modell ist, dass es nicht detailliert genug ist. Es eignet sich nur für eine sehr allgemeine Beschreibung einer interaktiven Anwendung. Des weiteren bietet es keine Unterstützung zur Definition von verteilten Anwendungen, Nebenläufigkeit, Kontextsensitivität und Performance. Auch semantische Unterstützung, also die Tatsache, dass Anwendungsinformationen im Interface verfügbar sein müssen, z. B. um dem Benutzer direktes Feedback bei Direct-Manipulation (s. u.) Schnittstellen zu geben, findet im Seeheim Modell keine Berücksichtigung. Für ein aussagekräftigeres Modell müssten die Abhängigkeiten zwischen Anwendung und Benutzungsschnittstelle stärker herausgearbeitet werden [ten Hagen 1991].

Das Seeheim Modell geht von einem konversationalen Interaktionsstil aus. Die Schnittstelle ermöglicht einen Dialog zwischen Anwendung und Benutzer: Der Benutzer spezifiziert Kommandos, das System führt daraufhin Aktionen aus und antwortet entsprechend. Der Benutzer kann dann die Antworten auswerten und neue Aktionen initiieren. Seit geraumer Zeit ist aber auch die Direct-Manipulation (DM) Metapher weit verbreitet. DM Schnittstellen geben dem Benutzer die Illusion, er würde direkt auf den Anwendungsobjekten arbeiten, ohne ein intermediäres System. In der konversationalen Metapher werden Objekte dagegen meist abstrakt z. B. per Name referenziert. Im Seeheim Modell ist die Syntax in der Dialogkomponente zentralisiert, in DM User-Interfaces ist die Syntax über die Domänen- und Präsentationsobjekte verteilt. Z. B. hat bei Drag-n-Drop das Loslassen (drop) eines mit der Maus gezogenen (drag) Objektes über einem anderen einen bestimmten Effekt, wie das Löschen einer Datei durch Ziehen in den Abfalleimer. Diese syntaktische Beziehung wird durch ein Hervorheben (highlighting) des Zielobjektes kenntlich gemacht, sobald das gezogene Objekt darüber geführt wird. Da diese syntaktische Beziehung dezentral zwischen Objekten besteht, ist das Seeheim Modell mit einer zentralen Dialogkomponente für diese Art der Interaktion ungeeignet [Hudson 1987].

Die konversationale Metapher erschwert auch eine Verschachtelung (interleaving) von Benutzereingaben und Systemfeedback [Calvary et al. 1997]. In [Baudel Lafon 1998] wird als Beispiel ein Progress-Indicator angeführt. Der Benutzer wird vom System kontinuierlich über den Fortschritt einer Aktivität informiert, er kann aber auch jederzeit die laufende Aktion unterbrechen. Das System besitzt damit keine festzulegende Abfolge von Ein- und Ausgaben.

Das Seeheim Modell konzentriert sich auf das Erreichen einer Isolation von Anwendungsfunktionalität und Benutzungsschnittstelle. Dies wird durch die Application Interface Komponente erreicht, die die Details der Implementation verbirgt. Die Isolation der Präsentations- von der Applikationsschicht durch eine Dialogschicht ist durch die konversationale Metapher motiviert und ist in dieser Interpretation für andere Interaktionsstile nicht geeignet.

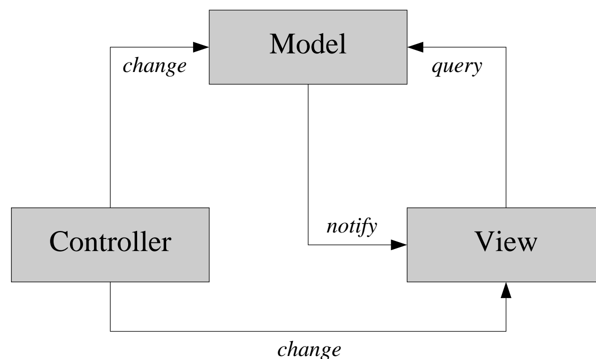


Abbildung 2.2: Beziehungen zwischen Model, View und Controller Objekten

2.1.2 Model-View-Controller (MVC)

Das Model-View-Controller Designpattern [Krasner Pope 1988] wurde von Xerox-Parc in den späten 70er Jahren entwickelt und für die Umsetzung der objektorientierten Programmierumgebung von Smalltalk 80 eingesetzt. Ziel dieses Patterns ist es, domänenspezifische Programmteile von ihrer grafischen Repräsentation und der Benutzerinteraktion zu trennen. Vor dem Einsatz von MVC wurden die verschiedenen Aspekte oftmals in einer Einheit vermengt. MVC entkoppelt sie und führt zu größerer Flexibilität und mehr Möglichkeiten, Code und Komponenten wiederzuverwenden. Weiterhin erlaubt es MVC, Domänendaten auf verschiedene Weisen zu betrachten.

Im MVC Paradigma werden Benutzungsschnittstellen aus mehreren Gruppen von jeweils drei Elementen (Triaden) aufgebaut (s. Abb. 2.2). Jedes dieser drei Elemente hat dabei spezielle Aufgaben. Im Folgenden werden nun kurz die Eigenschaften der drei Elemente Model, View und Controller beschrieben.

Model Ein Model repräsentiert ein Domänenobjekt, es kennt die darzustellenden Daten. Außerdem bietet es Möglichkeiten, diese Daten zu modifizieren und abzufragen. Es hat jedoch keinerlei Informationen über die Benutzungsschnittstelle. Weder kennt es die Art und Weise, in der die Daten dargestellt werden, noch kennt es die Interaktionen, die zur Datenmanipulation eingesetzt werden können.

View Eine Sicht bezieht sich auf ein Model. Sie hat die Aufgabe, Daten dieses Objektes grafisch darzustellen. Um an die aktuellen Informationen zu gelangen, benutzt sie die Abfragemethoden (query) des Models.

Controller Controllerobjekte haben die Aufgabe, Interaktionen eines Benutzers zu verarbeiten. Dazu propagieren sie Änderungen, die ein Nutzer vornimmt, an das Model oder direkt an einen View. Beispiel für das direkte Weiterleiten einer Nutzeraktion an einen View ist ein Sortierauftrag, der nicht die Daten selbst, sondern nur ihre Darstellung beeinflusst.

In Benutzungsschnittstellen arbeiten Views und Controller oft eng verzahnt. So ist ein Controller für das Aktualisieren eines Parameters im Model und der View für seine Darstellung verantwortlich. Diese Eins-zu-eins-Beziehung zwischen View und Controller führt manachmal auch dazu, dass die beiden Kompo-

nenten in ein Objekt zusammengefasst werden (Document-View Pattern, siehe Abschnitt 2.1.6). Ein Model steht in einer 1-n Beziehung zu View/Controller Paaren, die ausdrückt, dass ein Model auf verschiedene Weisen betrachtet werden kann.

Models sind dafür verantwortlich, Views über Änderungen ihres Status zu informieren. Ein View, der die Nachricht erhält, dass sich Daten seines Models verändert haben, muss sich neu darstellen. Um an die aktuellen Daten zu gelangen, benutzt er die Query-Methoden des Models. Interagiert ein Anwender mit dem Programm, so hat der Controller die Aufgabe, Modifikationen an das Model und evtl. auch an Views weiterzuleiten.

Anmerkungen

Das MVC-Pattern trennt die Eingabeverarbeitung (Controller) von der Ausgabe (View). Es unterstützt die methodologische Herangehensweise an User Interface Entwicklung und fördert nachvollziehbare saubere Designs. Die Entkoppelung von Interaktionslogik, Sichten und Domänenfunktionalität führt zu einer Reihe von Vorteilen. Die Trennung des Models von seinen Views führt zur Unabhängigkeit und Austauschbarkeit der Views genauso wie zur Unabhängigkeit der Implementation des Models. Dies gewährleistet Separation, Flexibilität und Erweiterbarkeit. Solange der Zugriff auf das Model nicht verändert wird, können interne Umstrukturierungen ohne Umstellungen im Rest des Programms vorgenommen werden. Die aus dem objektorientierten Paradigma stammende Dekomposition in unabhängige Model-Objekte wird durch eigenständige View- und Controllerobjekte in die Benutzungsschnittstelle übernommen. Dadurch wird dem Entwickler, im Gegensatz z. B. zum Seeheim Modell (Abschnitt 2.1.1), eine Methode an die Hand gegeben, die identifizierten Elemente durch unabhängige MVC-Triaden separat voneinander zu entwickeln.

MVC bietet jedoch keine Unterstützung, wie Abhängigkeiten zwischen verschiedenen Models, Views, oder Controllern zu implementieren sind. Daher ist es für menugesteuerte Anwendungen und komplizierte Dialogmodelle nicht unbedingt geeignet [Baudel Lafon 1998]. Darüber hinaus unterstützt es keine Abstraktionsniveaus, d. h. die Dekomposition erfolgt immer auf demselben Abstraktionslevel, eine hierarchische Dekomposition ist in der Grundform nicht vorgesehen.

Die allgemein gestiegene Komplexität durch die wechselseitigen Beziehungen der Elemente führt dazu, dass mehr Aufmerksamkeit für das Design aufgebracht werden muss. Zusätzlich wird es schwieriger, das Verhalten des Programms zur Laufzeit nachzuvollziehen, da die Komponenten automatisch Nachrichten untereinander versenden. Dieser Punkt hat auch Einfluss auf den resultierenden Code, der durch einen Overhead an Methodenaufrufen zwischen Versendern (publisher, observable) und Empfängern (subscriber, observer) bedingt durch das Observer-Pattern ineffizient werden kann (performance).

Eine explizite Trennung von Eingabe und Ausgabe erschwert oft die Entwicklung von Direct-Manipulation-Interfaces, da in DM Eingabe und Ausgabe eng gekoppelt sind. Mit der steigenden Verbreitung von DM Schnittstellen wurde es recht ruhig um MVC. In letzter Zeit erlebt MVC jedoch eine Renaissance, begründet durch den Aufstieg des World-Wide-Web und das Aufkommen von HTML-basierten Web-Anwendungen. In Web-Applikationen wird die Darstellung clientseitig von Browser vorgenommen, während die Verarbeitung der Ein-

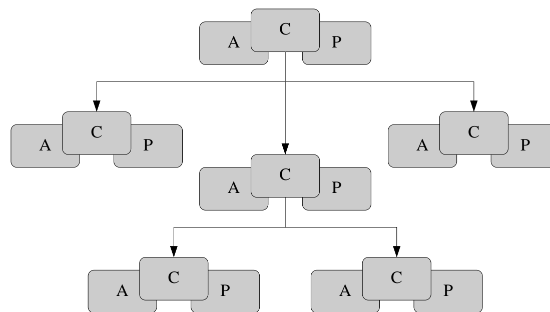


Abbildung 2.3: PAC-Architektur (aus [Kazman Bass 1996])

gaben (HTTP-Requests) serverseitig durchgeführt wird. Die explizite Trennung von Eingabe und Ausgabe in MVC bildet diese Struktur ab.

2.1.3 Presentation-Abstraction-Control (PAC)

Als Nachfolgeentwicklung zu MVC ist die Intention von PAC (Presentation-Abstraction-Control) [Coutaz 1987], zusätzlich zu Portabilität (durch Trennung von Anwendungsfunktionalität und Benutzungsschnittstelle) und Modifizierbarkeit (durch Dekomposition in gekapselte Objekte) eine bessere Skalierbarkeit zu erreichen [Kazman Bass 1996]. Dazu wird ein System aus einer Menge eigenständiger sog. Agenten (agents) aufgebaut, die untereinander Nachrichten austauschen. PAC wird daher auch als agentenbasierte (agent-based) Architektur bezeichnet. Die einzelnen PAC-Elemente entsprechen dabei Facetten, die zusammen einen Agenten ausmachen [Calvary et al. 1997].

Im Gegensatz zu MVC trennt PAC nicht Eingabe und Ausgabe (Controller und View), sondern fasst die Kommunikation zusammen. Die low-level Kommunikation mit dem Benutzer ist Aufgabe einer Präsentationsfacette (presentation), die damit die gesamte vom Benutzer wahrnehmbare Schnittstelle implementiert. Die Sicht auf die Applikationsfunktionalität wird in einer Abstraktionsfacette (abstraction) gekaspelt. Dazwischen dient die Control-Facette als Vermittler zwischen beiden Schichten. Diese Dekomposition entspricht der funktionalen Dekomposition des in Abschnitt 2.1.1 beschriebenen Seeheim Modells, wird allerdings bei PAC zur hierarchischen Dekomposition wiederholt auf verschiedenen Ebenen angewendet.

Dazu wird ein Agent in weitere PAC-Tripel verfeinert (s. Abb. 2.3), wobei die Abhängigkeiten zwischen den Sub-Agenten von der übergeordneten Control-Facette gesteuert werden.

Anmerkungen

Durch die Hierarchie der PAC-Agenten bietet PAC im Gegensatz zu MVC die Möglichkeit, Abhängigkeiten zwischen Interaktionskomponenten auszudrücken. Allerdings ist der MVC Ansatz oft ausreichend, wenn diese Beziehungen relativ einfach sind. Bei großen Systemen führt das PAC Modell jedoch zu einer verbesserten Wartbarkeit [Hussey Carrington 1995]. Dabei sollte man aber bedenken, dass im Kontext von UbiComp mehr als eine Interfacemodalität unterstützt werden muss, wozu mehrere PAC-Hierarchien erforderlich wären.

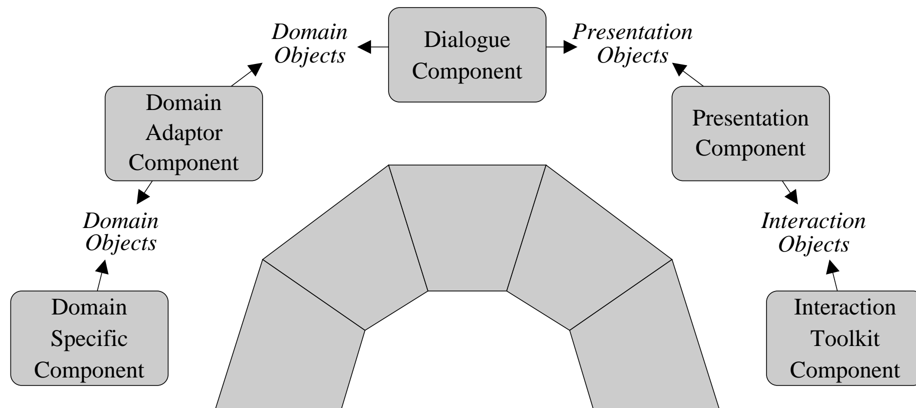


Abbildung 2.4: Arch Modell für UIMS-Architekturen (aus [Encarnação 1997])

In PAC sind die Abhängigkeiten zwischen den Facetten stärker als die der Model-Komponente von View und Controller in MVC. Dadurch wird erschwert, mehrere Repräsentationen einer Abstraktionsfacette darzustellen, was die Komplexität von Spezifikation und Implementation erhöht. Andererseits wird dadurch ermöglict, dem Benutzer direktes Feedback zu geben (siehe auch Progress-Indikator Beispiel in den Anmerkungen zum Seeheim Modell, Abschnitt 2.1.1).

2.1.4 Arch / Slinky

Das Arch / Slinky Metamodell [Bass et al. 1992] wurde nach einigen vorausgegangenen Meetings auf dem CHI'91 UIMS Tool Developer's Workshop entworfen. Das wesentliche Ziel bei der Entwicklung des Arch Modells war es, die zukünftigen Effekte sich verändernder Technologien zu minimieren, also z. B. den Rest des Systems von sich entwickelnden Toolkit-Bibliotheken (siehe Abschnitt 2.2.2.1) abzugrenzen.

Vor Arch gab es mit Seeheim (siehe Abschnitt 2.1.1), MVC (siehe Abschnitt 2.1.2), Seattle [Lantz et al. 1987], PAC (siehe Abschnitt 2.1.3) und Lisbon (Lissabon) [Duce et al. 1991] eine Reihe von Architekturen, die alle in der einen oder anderen Weise die Anwendungsfunktionalität von der Benutzungsschnittstelle zu entkoppeln suchten. Aufgrund der Vielfalt der unterschiedlichen vorhandenen Modelle war den Entwicklern von Arch klar, dass es nicht ein einziges Modell geben kann, das alle gegensätzlichen Design-Ziele (z. B. flexible Erweiterbarkeit vs. hohe Performance) erfüllt. Daher haben sie zusätzlich ein Metamodell entwickelt, mit dem an Hand vorgegebener Ziele speziell zugeschnittene Instanzen des Arch Modells abgeleitet werden können.

Das Arch Modell

Wie in Abbildung 2.4 zu sehen, bildet das Arch Modell einen Bogen, der auf zwei Sockeln ruht. Die Idee dahinter ist, dass sowohl die Domänenkomponente (links), als auch die Interaktionskomponente (rechts) zum großen Teil durch vorgegebene Technologien und Lösungen realisiert werden (z. B. DBMS auf der Domänenseite, Window-System auf der Interaktionsseite). Auf diesen beiden Grundsteinen aufbauend wird dann die restliche Anwendung entwickelt.

Die im Arch Modell beschriebenen Komponenten haben die folgenden Aufgaben:

Domain-Specific Component Die domänenspezifische Komponente kontrolliert, manipuliert und liefert Domänendaten und führt weitere Funktionen bezüglich der Domäne aus, d. h. sie implementiert die Funktionalität des Systems. Sie wird gelegentlich auch als funktionaler Kern (functional core) bezeichnet.

Domain-Adaptor Component Die Domänenadapterkomponente vermittelt zwischen der Domänen- und der Dialogkomponente. Diese Komponente implementiert die Operationen, die zur Benutzung des Systems notwendig, aber in der Domänenkomponente so nicht vorhanden sind (z. B. das Sortieren von Domänenentitäten zur Selektion durch den Benutzer). Außerdem überprüft sie Eingaben auf semantische Fehler und meldet diese und löst domänen-initiierte Dialoge aus. Darüber hinaus dient sie auch zur Abschirmung der Domänenkomponente, so dass die Eigenarten spezieller Implementationen (z. B. ob ein System auf einer relationalen oder objektorientierten Datenbank basiert) vor den übrigen Komponenten verborgen bleiben.

Dialogue Component Die Dialogkomponente ist verantwortlich für die Reihenfolgeplanung auf Ebene der Aufgaben für den Benutzer. Dazu gehört auch der benutzerabhängige Teil in der Applikationsdomäne. Sie gewährleistet die Konsistenz über mehrere Sichten und bildet Benutzungsschnittstellenformalismen auf domänenspezifische Formalismen ab und umgekehrt. Zusätzlich verwaltet sie den Anwendungskontext, also den aktuellen Zustand der Anwendung aus Sicht der Benutzungsschnittstelle (im Gegensatz zum internen, für den Benutzer nicht direkt sichtbaren, Systemzustand).

Presentation Component Die Präsentationskomponente ist ein Vermittler oder Puffer zwischen der Dialog- und der Toolkit-Komponente: Sie stellt der Dialogkomponente eine Menge toolkitunabhängiger Objekte zur Verfügung (z. B. ein Auswahl-Element, das als Menu oder als Gruppe von Radio-Buttons dargestellt werden kann). Die Entscheidung über die eigentliche Darstellung wird in der Präsentationskomponente getroffen.

Interaction-Toolkit Component Die Interaktions-Toolkit-Komponente implementiert die physische Interaktion mit dem Benutzer (in Hardware und Software). Als Toolkit-Komponente wird man meist eine vorhandene Toolkit Bibliothek einsetzen (z. B. MFC [Kruglinski 1996] oder Java Swing [Sun Microsystems 1998], siehe Abschnitt 2.2.2.1).

Im Gegensatz zum Seeheim Modell, das nur die funktionalen Teile einer Benutzungsschnittstelle betrachtet, liegt der Fokus beim Arch Modell auch auf der Natur der Daten, die zwischen den Interface- und Nicht-Interfacekomponenten ausgetauscht werden. Es werden verschiedene Arten von Objekten[1] identifiziert, die die zwischen den Komponenten ausgetauschten Informationen kapseln:

[1] Der Begriff „Objekt“ wird benutzt, um den Formalismus des Datentransfers zwischen den Komponenten zu beschreiben. Er bezieht sich nicht speziell auf formale, instantierte Objekte in einer objektorientierten Entwicklungsumgebung.

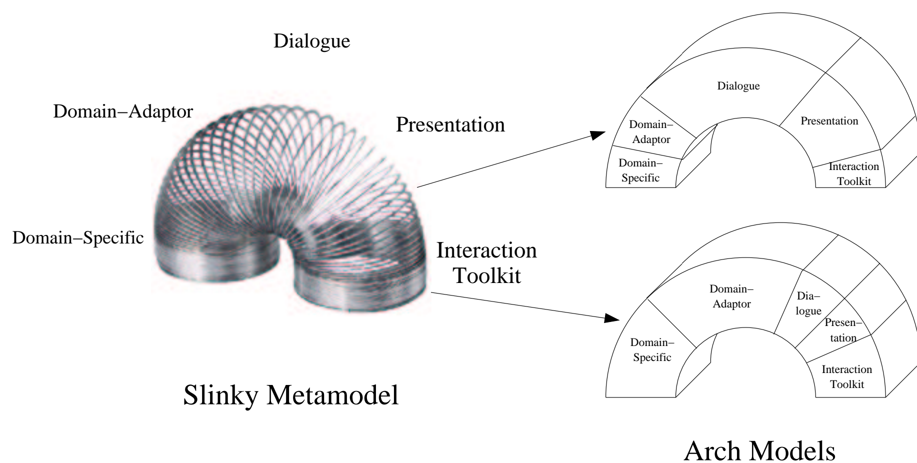


Abbildung 2.5: Ableitungen aus dem Slinky Metamodell (aus [Bass et al. 1992])

Domain Objects Domänenobjekte kapseln Daten und Operationen der Anwendungsdomäne und werden von der Domänenkomponente verwendet, um die nicht auf das User Interface bezogenen Funktionen (wie das Laden und Speichern von Daten in einer Datenbank) umzusetzen. Die Adapterkomponente verwendet sie, um die interfacerelevanten, nicht in der Domäne integrierten Operationen (s. o.) zu implementieren.

Presentation Objects Präsentationsobjekte sind virtuelle Interaktionsobjekte, die Benutzerinteraktionen kontrollieren. Sie sind abstrakte Interaktionseinheiten, die mit verschiedenen konkreten Toolkits dargestellt werden können.

Interaction Objects Bei konkreten Interaktionsobjekten handelt es sich um Objekte der verwendeten Toolkitbibliothek (widgets). Sie korrespondieren zu einem Präsentationsobjekt, dessen physische Repräsentation sie sind.

Das Slinky Metamodell

Das Arch Modell weist verschiedene Funktionalitäten separaten Komponenten zu, um bei Änderungen einer bestimmten Funktionalität die Auswirkungen auf die übrigen Komponenten zu beschränken. Das Slinky Metamodell dient dazu, spezielle Instanzen des Arch Modells abzuleiten, die ausgewählte Kriterien besser umsetzen. Die Bezeichnung „Arch“ steht für ein stabiles Modell, in dem Funktionalitäten bereits zugewiesen sind. Der Begriff „Slinky“ bezeichnet die Tatsache, dass im Metamodell noch keine Zuordnung von Funktionalitäten festgelegt wurde, Funktionalität also noch zwischen den Komponenten verlagert werden kann. Dieses Konzept ähnelt einer Spielzeug-Spirale, die unter dem Namen „Slinky“ vertrieben wird.[2]

Abbildung 2.5 zeigt wie das Slinky Metamodell angewendet wird. Je nach den vorgegebenen Kriterien erhält man Arch Modelle, in denen bestimmte Komponenten besonders stark ausgeprägt sind, während andere nur wenig Funktionalität enthalten.

[2] `http://www.slinkytoys.com`

Aufgrund der unterschiedlich ausgeprägten Funktionalitäten unterscheiden sich Arch Modelle je nach Art der Anwendung. Datenorientierte Anwendungen haben meist aufwändige Domänenkomponenten, aber nur sehr einfach gehaltene Dialog- und Interaktionsfunktionalität. Systeme zur Visualisierung von Daten haben hingegen erheblich aufwändigere Dialog- und vor allem Interaktionskomponenten, dafür aber kaum domänenspezifische Funktionalität. Dialogorientierte (z. B. formularbasierte) Systeme wiederum haben meist einfache Domänen- und Interaktionskomponenten und sind auf die Ablaufsteuerung und Verwaltung von Fenstern fokussiert.

Funktionalitäten können auch bedingt durch Fortschritte in der Softwareentwicklung anderen Komponenten zugewiesen werden. Ein Beispiel: Aktionen wie „Datei-öffnen“ wurden ursprünglich der Anwendungsdomäne zugeordnet, in den heutigen Toolkits ist aber meist ein File-Selection-Dialog enthalten, der dieselbe Funktionalität in der Interaktionskomponente implementiert. Die Funktionalität wurde also von einem Ende des Modells zum anderen verlagert.

Anmerkungen

In vielen Beschreibungen des Arch Modells wird die Unterscheidung zwischen Domänenkomponente und Adapterkomponente als Neuerung gegenüber dem Seeheim Modell angegeben (z. B. [Kazman Bass 1996]). Tatsächlich aber entspricht die Anwendungssemantik-Komponente des Seeheim Modells der Adapterkomponente des Arch Modells (z. B. Bündelung von Domänenoperationen zu komplexen Aufgaben). Da das Seeheim Modell nur die Benutzungsschnittstelle einer Anwendung, nicht aber den Anwendungskern modelliert, wird die semantische Komponente des Seeheim Modells gelegentlich fälschlicherweise als Domänenkomponente interpretiert.

Die vom Arch Modell eingeführte Beschreibung einer abstrakten Präsentationskomponente ist als weitergehende Abstraktion von virtuellen Toolkits zu verstehen. Die Unabhängigkeit von den konkreten Interaktionselementen hat sich in der Praxis als sehr nützlich erwiesen. Sie wird in vielen Systemen und Frameworks praktiziert (siehe z. B. JWAM, Abschnitt 3.1.3). Auch einige modellbasierte User-Interface Entwicklungswerkzeuge (siehe Abschnitte 2.3.6 und 3.2) übernehmen diese Trennung und führen abstrakte und konkrete Präsentationsmodelle ein.

Im Gegensatz zum Seeheim Modell macht das Arch Modell durch die Beschreibung der Kommunikationsobjekte die Natur der Kommunikation zwischen den Komponenten explizit. Es ist deutlich detaillierter als das Seeheim Model und bietet dadurch wesentlich mehr Anleitung zur Implementation von Systemen.

Anstelle der konversationalen Rolle der Dialogkomponente im Seeheim Modell wurde ihr im Arch Modell die Rolle der Reihenfolgeplanung auf Ebene der Aufgaben zugewiesen. Die low-level Syntax ist in wiederverwendbare Toolkit-Objekte verlagert. Somit wurde der Abstraktionsgrad der Dialog- und Präsentationskomponente erhöht. Es gibt explizite Ansatzpunkte, um Plattformunabhängigkeit und Modifizierbarkeit der Benutzungschnittstelle zu gewährleisten. Die Domänenadapterkomponente erlaubt Plattformunabhängigkeit durch austauschbare Domänenkomponenten. Die Präsentationskomponente, die die konkreten Toolkits vor der Dialogkomponente verbirgt, stellt die Modifizierbarkeit des User Interfaces sicher [Calvary et al. 1997].

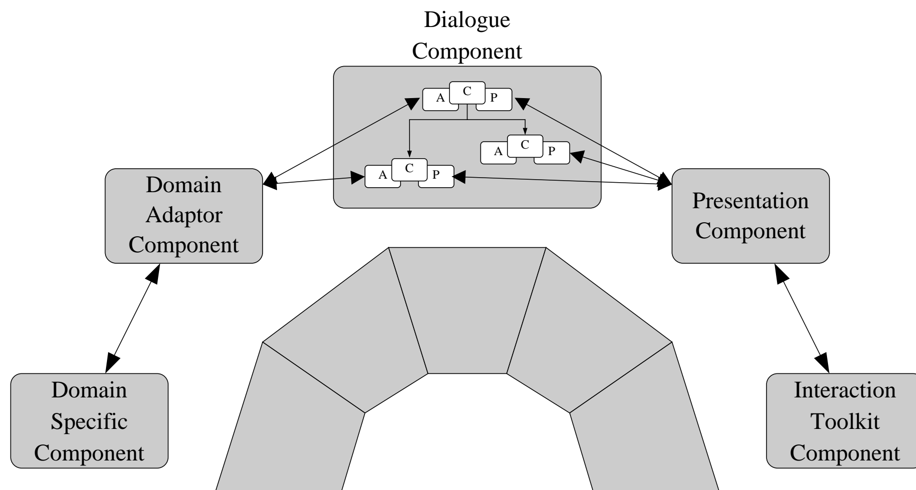


Abbildung 2.6: PAC-Amodeus (aus [Calvary et al. 1997])

Das Arch Modell erreicht durch die Adapterkomponenten vor allen Dingen Separation, Flexibilität und Erweiterbarkeit. Aspekte, wie Performance und Skalierbarkeit, wurden im Arch Modell explizit nicht berücksichtigt und werden vom Slinky Metamodel durch die flexible Verlagerung von Funktionalitäten abgedeckt. Dabei ist das Slinky Metamodell in seiner Flexibilität etwas beliebig und gibt keine klaren Richtlinien vor, wie bestimmte Ziele erreicht werden können. Eine konkrete Matrix, wie Funktionalitäten, bedingt durch vorgegebene Kriterien, zugewiesen werden können, sollte laut [Bass et al. 1992] Teil zukünftiger Forschung sein. Allerdings scheint das Interesse am Arch Modell nicht ausgereicht zu haben, um die Forschung auf diesem Gebiet zu intensivieren.

2.1.5 PAC-Amodeus

Die PAC-Amodeus-Architektur [Coutaz et al. 1995] versucht, die Vorteile des Arch Modells und der PAC-Architektur zu verbinden. Das Arch Modell (siehe Abschnitt 2.1.4) ist eine generelle Architektur, die zwecks Portabilität und Modifizierbarkeit Schichten einführt, welche die Anwendungsfunktionalität von der Benutzungsschnittstelle entkoppeln. Allerdings enthält Arch keine Anleitung, wie die einzelnen Komponenten zu implementieren sind. PAC (siehe Abschnitt 2.1.3) ist ein Modell zur hierarchischen Dekomposition einer Benutzungsschnittstelle, bietet aber keine direkte Unterstützung für Portabilität gegenüber Präsentations- und Anwendungskomponenten.

PAC-Amodeus benutzt die beiden Adapter-Komponenten des Arch Modells (Domain-Adapter und Presentation), um den Schlüsselbaustein, die Dialogkomponente, von der Anwendungs- und der toolkitspezifischen Komponente abzukoppeln. Die Dialogkomponente wird als PAC-Architektur mit Agenten realisiert (s. Abb. 2.6).

Die beiden Adapterkomponenten werden durch einzelne Entitäten realisiert. Dabei kann die Präsentationsfacette eines PAC-Agenten mit mehreren Entitäten aus der Präsentationskomponente in Verbindung stehen, so wie die Abstraktionsfacette mehrere Beziehungen zu Entitäten des Domänenadapters haben

kann. Die Kommunikation zwischen Presentation und Abstraction findet dann bevorzugt direkt über eine Control-Facette eines Agenten statt. Sie kann sich aber auch durch übergeordnete Control-Facetten über die ganze Hierarchie der PAC-Agenten erstrecken. Im allgemeinen haben nicht alle Agenten immer alle drei Facetten, einige Agenten, wie diejenigen, die die Konsistenz zwischen untergeordneten Agenten verwalten, besitzen lediglich eine Control-Facette. Es kommt vor, dass ein Agent keine Abstraktionsfacette besitzt und über übergeordnete Agenten mit einem Agenten kommuniziert, der keine Präsentationsfacette hat.

In [Coutaz et al. 1995] wird zusätzlich noch eine Reihe von Regeln vorgestellt, wie man die Dialog-Komponente in einem Bottom-Up-Ansatz auf Basis einer externen Spezifikation in Agenten realisieren kann. Z. B. wird für jedes Dialogfenster ein Agent modelliert. Muss die Konsistenz zwischen zwei Fenstern explizit gewahrt werden, so wird dafür ein übergeordneter Agent eingeführt. Allerdings ist auch ein Top-Down Ansatz auf Basis eines Taskmodells denkbar. Wenn externe Spezifikation und Taskmodell miteinander konsistent sind, so sollte laut [Coutaz et al. 1995] das Ergebnis des Entwurfs dieselbe Menge kooperierender Agenten sein.

Des weiteren wird das Konzept der Domain-Knowledge-Delegation angesprochen: Informationen, die eigentlich zur Domäne gehören, werden für den Zweck einer Benutzer-Interaktion in der Benutzungsschnittstelle verwaltet. Z. B. sollte das Ziehen einer Linie in einem Zeichenprogramm für den Benutzer ohne merkliche Verzögerung durchführbar sein. Aus Performancegründen kann es daher sinnvoll sein, erst nach Beendigung der Eingabe die Linie (die komplexe Eigenschaften wie Farbe, Dicke, antialiasing Optionen, usw. haben kann) als Domänenobjekt anzulegen. Während der Interaktion wird statt der eigentlichen Linie ein Platzhalter dargestellt, der zunächst keine Entsprechung in der Domäne hat. Ein geeigneter Ort, um solcher Art domänenspezifische Informationen in der Benutzungsschnittstelle zu verwalten, ist die Abstraction-Facette eines PAC-Agenten.

Eine Methode zur Verfeinerung der übrigen Komponenten des Arch Modells ist in PAC-Amodeus nicht vorhanden. Die Annahme ist hier, dass für diese Komponenten oft vorgefertigter, wiederverwendbarer Code zur Verfügung steht (etwa Toolkits für die Low-Level-Interaktionskomponente) und sich die Implementation daher nach den Gegebenheiten richten muss und keiner strengen Architektur folgen wird. Über die beiden Adapterkomponenten sollte die Dialogkomponente aber hinreichend gegenüber solchen Zwängen abgeschirmt sein.

Anmerkungen

Wie schon in PAC haben die Control-Facetten zwei Aufgaben. Einerseits sind sie für die Verwaltung der Subagenten zuständig, andererseits kontrollieren sie die Kommunikation zwischen Presentation- und Abstraction-Facette. Eine Trennung dieser beiden Aufgaben in eigene Facetten wäre vielleicht von Vorteil, auch wenn es scheint, als würde eine Control-Facette entweder Subagenten eines Agenten verwalten, der dann aber keine eigenen Presentation- oder Abstraction-Facette hat, oder Presentation- oder Abstraction-Facette eines Agenten verbinden, der dann nicht weiter verfeinert ist. Eine Unterscheidung zwischen atomaren und Container-Agenten böte sich an.

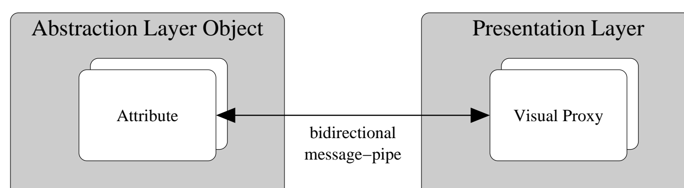


Abbildung 2.7: Visual Proxy Architektur (aus [Holub 1999])

2.1.6 Visual Proxy

Eine Vereinfachung des MVC Modells (siehe Abschnitt 2.1.2) ist das Document-View Pattern [Kruglinski 1996]. Das Prinzip entspricht einer Zusammenfassung der View und Controller Komponenten des MVC Modells. Dabei entspricht das Document dem Model aus MVC, und der View vereint mit den Funktionalitäten von Controller und (MVC-) View Input und Output. Im Swing Toolkit für Java wird diese Architektur auch als UI-Delegate bezeichnet [Sun Microsystems 1998].

Eine Architektur unter Verwendung des Document-View Patterns ist unter dem Namen Visual-Proxy ausführlich in [Holub 1999] beschrieben. Um Visual-Proxy anzuwenden, definiert Holub atomare Darstellungseinheiten, d. h. charakterisierende Eigenschaften eines Objektes, die eigenständig darstellbar sein sollen. Charakterisierende Eigenschaften sind Operation und Attribute, die von Holub unter der unglücklichen Bezeichnung „Attributes“ zusammengefasst werden. Dabei spielt keine Rolle wie ein Attribut letztendlich in einem Objekt implementiert ist, sondern lediglich, dass es dargestellt werden kann. Z. B. könnte ein Attribut „Name“ als Objekt-Variable oder als Zugriffsmethode, die eine Datenbankabfrage durchführt, implementiert sein. Die Implementation hat keinen Einfluss auf die Darstellung.

Holub sieht Visual-Proxy als eine Abwandlung der PAC-Architektur (siehe Abschnitt 2.1.3). Im Gegensatz zu PAC sind Presentation- (visual proxy) und Abstractionfacette (attribute) direkt verbunden. Die Control-Komponente wird nur zur hierarchischen Dekomposition benutzt, d. h. die Control-Komponente ist dafür zuständing, geeignete Proxies zu erzeugen[3] und in einem Dialog zusammenzufassen. Das Control-Objekt kann dabei wieder als Abstraction-Layer-Objekt aufgefasst werden, das als Proxy einen kompletten Dialog hat. Das Proxy eines Control Objektes enthält dann die einzelnen Proxies der vom Control zusammengefassten Attribute. An der weiteren Kommunikation zwischen Presentation und Abstraction nimmt die Control-Komponente in der Visual-Proxy-Architektur nicht teil (s. Abb. 2.7).

Proxies kommunizieren nur mit ihrem Attribut, nicht untereinander. Abhängigkeiten zwischen Proxies bestehen daher nicht direkt, sondern nur, wenn zwei Proxies dasselbe Attribut oder zwei voneinander abhängige Attribute repräsentieren.

Anmerkungen

Das Document-View Pattern ist von allen vorgestellten Architekturen am einfachsten. Eine gewisse Design-Freiheit besteht in der Art und Weise, wie die

[3] Bei Holub werden Proxies von den Objekten selbst bereitgestellt, und können über eine `proxyFor(attribute)` Methode angefordert werden.

Schnittstelle zwischen UI-Delegate und Domänenelement festgelegt wird. Holub implementiert diese als Teil des Applicationlayer Objektes, indem er den Innerclass-Mechanismus von Java verwendet. Im Gegensatz zum Observer Pattern [Gamma et al. 1995] ist damit die Schnittstelle des Applicationlayer Objektes nicht öffentlich zugänglich, worauf Holub besonderen Wert legt. In Swing am Beispiel von `javax.swing.text.Document` und `javax.swing.JTextField`, findet man das Observer Pattern implementiert, dass wohl die gängigste Umsetzung des Document-View Patterns darstellt. Das `JTextField` als Visual Proxy kennt die umfangreiche öffentliche Schnittstelle des `Document` Objektes, während das `Document` das `JTextField` nur über ein einfaches `DocumentListener` Interface kennt.

In beiden Implementationen sind Proxy und zu Grunde liegendes Objekt direkt verbunden. Keine zusätzlichen Eventhandler o. ä. sind nötig, um Benutzereingaben wirksam zu machen. Dennoch ist die Anwendungsfunktionalität von der Benutzungsschnittstelle entkoppelt, lediglich ein generischer Benachrichtigungsmechanismus muss implementiert werden, wenn Änderungen vom System aus an das User Interface propagiert werden sollen. In rein reaktiven Systemen, wo alle Änderungen direkt durch Benutzereingaben ausgelöst werden, kann dieser Mechanismus sogar entfallen.

Als einzige hier vorgestellte Architektur legt Visual Proxy die Granularität der Darstellungseinheiten fest. Dabei ist die Granularität wesentlich feiner als dies z. B. bei MVC gängige Praxis ist (ein MVC Tripel pro Dialog). Das Ziel dahinter ist, dass die einzelnen atomaren Eigenschaften (attributes) und die atomaren Darstellungseinheiten (visual proxies) so einfach gehalten werden können, das sie in verschiedenen Kontexten einsetzbar sind. Die Wiederverwertbarkeit eines MVC Tripels ist dagegen üblicherweise gering. Im Zusammenhang mit UbiComp bringt dies Vorteile, wenn z. B. abhängig von der Kapazität eines Endgerätes alle oder nur wenige essentielle Eigenschaften eines Objektes dargestellt werden können. Auch wird die Portierung auf neue Interaktionsmodalitäten erleichtert, wenn nur einfache Visual Proxies erstellt werden müssen, die dann in verschiedenen Anwendungen einsetzbar sind.

2.1.7 Hierarchical MVC (HMVC) / Layered MVC

In [Cai et al. 2000] wird ein Entwurfsmuster für die Clientschicht einer Mehrschicht-Webarchi-tektur beschrieben, die von MVC-Client und MVP (siehe Abschnitte 3.1.1 und 3.1.4) inspiriert ist. Es unterscheidet dabei zwischen einer Präsentationsschicht (auch Clientschicht genannt), und einer GUI-Schicht, wobei die GUI-Schicht eine Untermenge der gesamten Präsentationsschicht darstellt, zuständig für die Widgets und die unmittelbaren Effekte von Benutzeraktionen. Die Präsentationsschicht behandelt den Applikationsfluss und die Server-Interaktion.

Das HMVC-Muster strukturiert die Architektur eines Clients in mehrere MVC-Schichten (s. Abb. 2.8). Die Kommunikation zwischen den Schichten ist im Controller-Objekt gekapselt. Das soll zur besseren Wiederverwertbarkeit, Erweiterbarkeit und zu einfacherer Pflege führen, indem es die Abhängigkeiten zwischen getrennten Programmteilen verringert.

Cai, Kapila und Pal benennen drei Aspekte der Clientschicht Entwicklung: GUI-Layout-Code bestimmt die Anordnung von Widgets und das 'Look-and-Feel' der Dialoge. GUI-Feature-Code validiert Benutzereingaben und erfasst vom

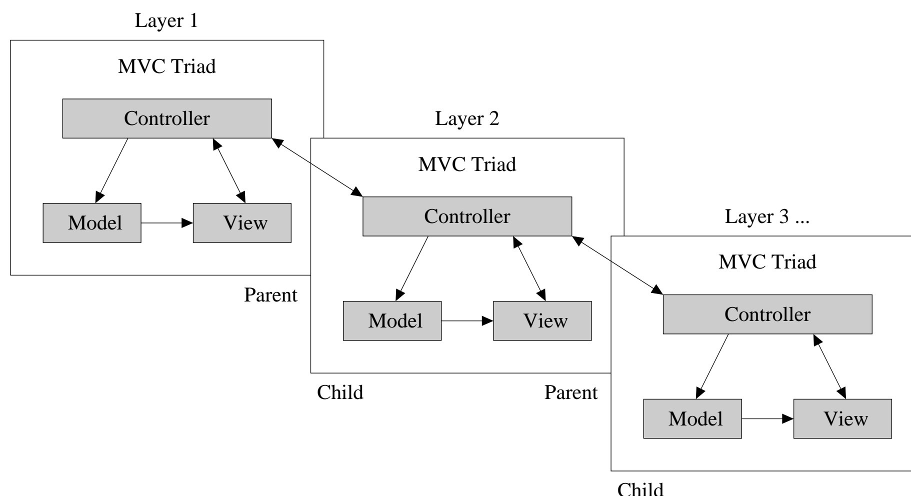


Abbildung 2.8: MVC layers (aus [Cai et al. 2000])

User ausgelöste Ereignisse. Die Applikationslogik steuert Applikationsfluss, Navigation und Server-Interaktion. Diese Aspekte werden vom MVC-Muster (siehe Abschnitt 2.1.2) abgebildet. Views bestimmen das GUI-Layout, Controller sind die GUI-Features, und die Applikationslogik wird vom Model implementiert.

In HMVC wird dieses Muster in verschiedenen Skalen eingesetzt: Ein top-level Container stellt die grobe Struktur der gesamten Applikation dar. Darin enthalten sind Dialoge, die in einzelne Bereiche zerlegt werden. Diese Bereiche enthalten letztendlich die Widgets eines Toolkits (z. B. Swing). Diese verschiedenen Level von Views haben jeweils ein zugeordnetes Model und einen Controller. Die Abhängigkeit von einem bestimmten Toolkit beschränkt sich dabei auf die letzte Ebene von Controllern, da nur diese Controller mit den im Bereich enthaltenen Widgets kommunizieren, die Controller unterhalten sich mit ihren über- bzw. untergeordneten Controllern über toolkitunabhängige Ereignisse (app events). Ein Ereignis, das von dem empfangenden Controller nicht verarbeitet werden kann, wird an den übergeordneten Controller weitergeleitet.

Anders als in MVC hat das View-Objekt keinen direkten Zugriff auf das Model-Objekt. Möchte stattdessen ein View die Daten aus dem Model anzeigen, sendet er eine Datenanfrage an seinen Controller, der diese an das Model weiterleitet. Nicht auf jeder Ebene der Anwendung muss ein Model implementiert werden. So gibt es für den obersten View (GUIContainer) üblicherweise kein zugehöriges Model, wenn für diesen keine Zustandsinformationen gespeichert werden. Wenn das Model eine Datenanfrage vom Controller empfängt, führt es z. B. eine SQL-Anweisung aus, um die Daten zu beschaffen, und informiert den View über die Verfügbarkeit neuer Daten.

Anmerkungen

HMVC vereint Prinzipien von MVC und PAC (siehe Abschnitte 2.1.2 und 2.1.3). Von PAC wird die hierarchische, über Control-Objekte aufgebaute Struktur übernommen, von MVC die Trennung von Eingabe und Ausgabe. Wie auch PAC weist HMVC gegenüber MVC eine wesentlich höhere Komplexität auf,

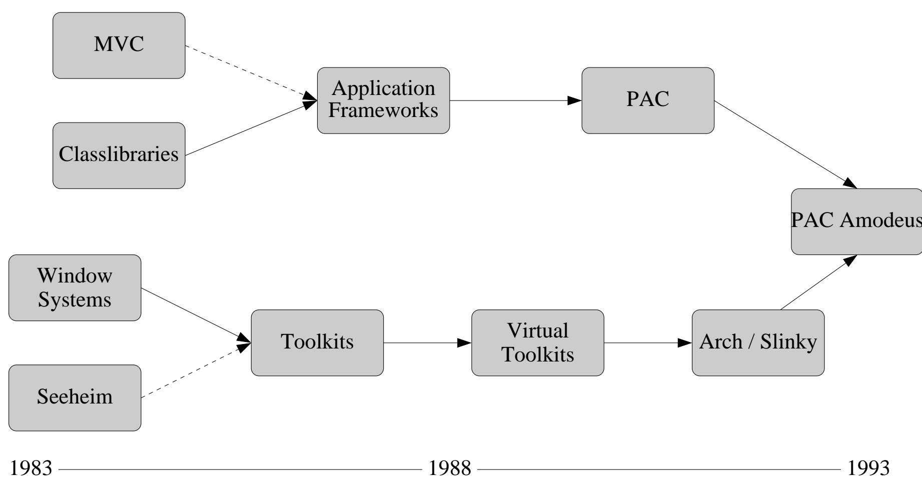


Abbildung 2.9: Entwicklung von User Interface Artefakten [Kazman Bass 1996]

die natürlich nur dann gerechtfertigt ist, wenn sie dem zu implementierenden System angemessen ist. Für einfache Systeme ist MVC die bessere Wahl.

In MVC hat der Controller nur die Aufgabe der Eingabeverarbeitung. Die Controlfacette in PAC ist für die Steuerung der Kommunikation zwischen Presentation- und Abstractionfacette und die Koordination mit anderen Controlfacetten verantwortlich. In HMVC müssen Controller-Objekte alle diese Aufgaben übernehmen, was die Modifizierbarkeit stark einschränkt.

2.1.8 Zusammenfassung

Alle vorgestellten Architekturen schlagen in der einen oder anderen Weise eine Trennung in Präsentations-, Dialog- und Applikationskomponente vor, eine Herangehensweise, die laut [Kazman Bass 1996] unstrittig ist. Ein wesentliches Unterscheidungsmerkmal ist, ob diese Trennung auf das System als Ganzes angewendet wird oder einzelne Teile (z. B. Sichten auf einzelne Objekte) nochmals in Präsentations-, Dialog- und Applikationskomponente aufgespalten werden. Daher lassen sich die beschriebenen Architekturen in zwei Typen einteilen: Die Schichtenmodelle wie das Seeheim und das Arch Modell dekomponieren das System als Ganzes und jede Schicht kapselt anwendungsweit einen Aspekt der Benutzungsschnittstelle. Die objektorientierten oder agentenbasierten Ansätze wie MVC und PAC unterteilen eine Anwendung in getrennte Komponenten, die eine Teilfunktionalität der Anwendung implementieren und dazu alle Benutzungsschnittstellenaspekte dieser Komponente berücksichtigen müssen.

[Kazman Bass 1996] beschreiben die historische Entwicklung dieser zwei Architekturtypen und wie sie untereinander und mit einigen anderen Artefakten der Benutzungsschnittstellenentwicklung zusammenhängen (s. Abb. 2.9). Es ist zu sehen, dass Windowsysteme etwa zur gleichen Zeit entstanden wie das Seeheim Modell und von diesem beeinflusst zur Entwicklung von Toolkits führten. Klassenbibliotheken entstanden etwa zur gleichen Zeit wie MVC und wurden später zu Frameworks erweitert, wobei die MVC Architektur oft eine zentrale Rolle spielte. Die PAC-Amodeus Architektur vereint schließlich die agentenba-

sierte und die schichtenorientierte Herangehensweise.

Der besondere Fokus bei den Schichtenmodellen liegt auf der Austauschbarkeit der einzelnen Ebenen (z. B. um eine Anwendung an ein neues Toolkit anzupassen), während das Ziel bei den objektorientierten Ansätzen ist, die einzelnen Teile der Anwendung gegeneinander abzuschirmen, damit Änderungen möglichst nur lokale Effekte haben. Bei den objektorientierten Ansätzen fördert die Dekomposition auf lokaler Ebene die Skalierbarkeit und die Modifizierbarkeit, da sie es ermöglicht, Funktionseinheiten einzeln zu entfernen und hinzuzufügen. Die Schichtenmodelle haben eine bessere Isolation, da sie die unterschiedlichen Aspekte einer Benutzungsschnittstelle stärker von einander abschirmen.

Die Schichtenmodelle erscheinen im Kontext des UbiComp besonders interessant. Die einzelnen Schichten eignen sich zur gezielten Portierung auf verschiedene Klassen von Endgeräten. Der Portierungsaufwand für eine Anwendung läßt sich so minimieren und auf einzelne Schichten beschränken. Aufgrund der Heterogenität der Endgeräte sind dabei verschiedene Aufteilungen der Schichten zwischen Endgerät und Server notwendig. So kann z. B. die Dialogsteuerung bedarfsweise auf das Endgerät verlagert werden oder auf dem Server verbleiben. Eine Architektur dieser Art auf Basis des Arch Modells beschreibt [Hejda 2000].

Schichtenmodelle adressieren jedoch nicht die Komplexität interaktiver Anwendungen. Um dieser Komplexität Herr zu werden bedarf es weiterer Methoden der Dekomposition innerhalb der Schichten, eine Notwendigkeit, die z. B. von PAC-Amodeus identifiziert wird. Besondere Bedeutung erlangt diese Struktur im Hinblick auf UbiComp. Um universellen Zugriff auf Anwendungen zu ermöglichen, sollten insbesondere die präsentationsrelevanten, d. h. dem Endgerät zuzuordnenden Schichten zur besseren Portierbarkeit in wiederverwendbare Bausteine zerlegt werden.

2.2 Techniken

In diesem Abschnitt betrachten die Autoren Techniken zur Beschreibung von Benutzungsschnittstellen. Darunter verstehen die Autoren etablierte Standards und Methoden, die in dem Sinne abstrakt sind, dass sie losgelöst von einer konkreten Implementation in einem Werkzeug bestehen. Sie sind daher unabhängig von der jeweiligen Entwicklungsumgebung und können in diversen Kontexten eingesetzt werden. Dennoch sind Techniken, anders als Architekturen, in einem anderen Sinne konkret, da mit ihnen konkrete Aspekte eines User Interfaces beschrieben werden. Die Beschreibungen können von entsprechenden Werkzeugen in eine Implementation umgesetzt werden. In Abschnitt 2.3 werden die Autoren sich Werkzeugen widmen, welche die im Folgenden vorgestellten Techniken anwenden.

2.2.1 Klassifikation

[Myers 1989] klassifiziert Spezifikationstechniken für User-Interfaces nach der Art und Weise, wie ein Entwickler damit Oberflächen beschreibt. Er unterscheidet sprachenbasierte und grafische Spezifikation. Des weiteren betrachtet er Werkzeuge, die Benutzungsschnittstellen automatisch aus Spezifikationen der Anwendungssemantik erzeugen können. [Fähnrich 1995] adaptiert dieses Schema. Als Abstraktionsgrade bei der Spezifikation unterscheidet Fähnrich zwei

Arten der sprachenbasierten Spezifikation: Einerseits die direkte Implementation in einer allgemeinen Programmiersprache, andererseits die deklarative Spezifikation in einer speziellen User-Interface Beschreibungssprache.

Am konkretesten ist die direkte Implementation einer Benutzungsschnittstelle in einer allgemeinen Programmiersprache. Auf bestimmte Aspekte der Benutzungsschnittstelle spezialisierte textbasierte Beschreibungssprachen abstrahieren von der konkreten Implementation. Eine noch größere Abstraktion erreichen grafische Notationen, die dazu dienen sollen, komplexe Zusammenhänge übersichtlicher darzustellen. Sowohl textuelle als auch grafische Beschreibungssprachen müssen von Code-Generatoren oder Interpretern weiterverarbeitet werden, um eine ausführbare Version zu erhalten. Den höchsten Abstraktionsgrad spricht Fähnrich den automatischen Generatoren zu.

Diesem Schema fügt Fähnrich als weitere Klassifizierungsdimension eine Unterscheidung nach Funktionalitäten hinzu. Eine funktionale Partitionierung spezifiziert die unterschiedlichen Funktionen oder Dienste, die ein individuelles System bereitstellen muss. Funktionale Partitionierung hilft zu verstehen, welche Funktionen allen Anwendungen einer Domäne gemein sind. Fachleute auf dem Gebiet der Mensch-Maschine Interaktion sind sich weitgehend einig, dass alle interaktiven Systeme über Präsentations-, Dialog- und Applikationsfunktionalität verfügen müssen, unabhängig davon, ob diese getrennten Funktionalitäten in verschiedenen Komponenten oder zusammen in einer einzigen Komponente realisiert werden [Kazman Bass 1996].

Eine Präsentationskomponente ist für die Darstellungsschicht einer Anwendung verantwortlich. Dialogkomponenten übernehmen die Ablaufsteuerung, sowohl auf Intra- als auch auf Interdialogebene und kapseln den Zustand der Benutzungsschnittstelle. Die Anbindung an die Implementation der Anwendungsfunktionalität wird über eine Applikationsschnittstelle durchgeführt. Das Seeheim Modell (siehe Abschnitt 2.1.1) gibt eine detaillierte Beschreibung der Aufgaben dieser Komponenten.

Übersicht über User Interface Beschreibungstechniken

Um User-Interface Beschreibungstechniken zu klassifizieren, unterscheiden die Autoren bei der Applikationssemantik zusätzlich zwischen Aufgaben (Tasks) und Entitäten. Bei Entitäten handelt es sich um Gegenstände der Anwendungsdomäne. Objektorientierte Techniken modellieren dabei nicht nur die Eigenschaften (=Daten) der Entitäten, sondern auch eine evtl. enthaltene Funktionalität. Aufgaben sind geeignet, die Inhalte und Abläufe einer Anwendung zu beschreiben. Sie leisten einen wichtigen Beitrag zur benutzerzentrierten Interfaceentwicklung, da sie die angestrebte Arbeitweise des Benutzers widerspiegeln.

Die Übersicht über die Techniken ist in Abbildung 2.10 gegeben. Die einzelnen Techniken konzentrieren sich jeweils auf einen funktionalen Aspekt der Benutzungsschnittstelle. Es gibt kaum eine Technik, die mehr als einen Aspekt abdeckt. Die einzige Ausnahme bilden hier Statecharts und Petrinetze, die sowohl zur Modellierung des Dialogverhaltens als auch des Verhaltens der Entitäten eingesetzt werden können oder zur Steuerung eines Aufgabenablaufs (Workflow). Die Autoren betrachten diese Techniken jedoch nur im Kontext der Dialogmodellierung, weil es nicht Aufgabe der Benutzungsschnittstelle ist, dafür zu sorgen, dass die Implementation der Anwendungsfunktionalität spezifiziertem Verhalten folgt.

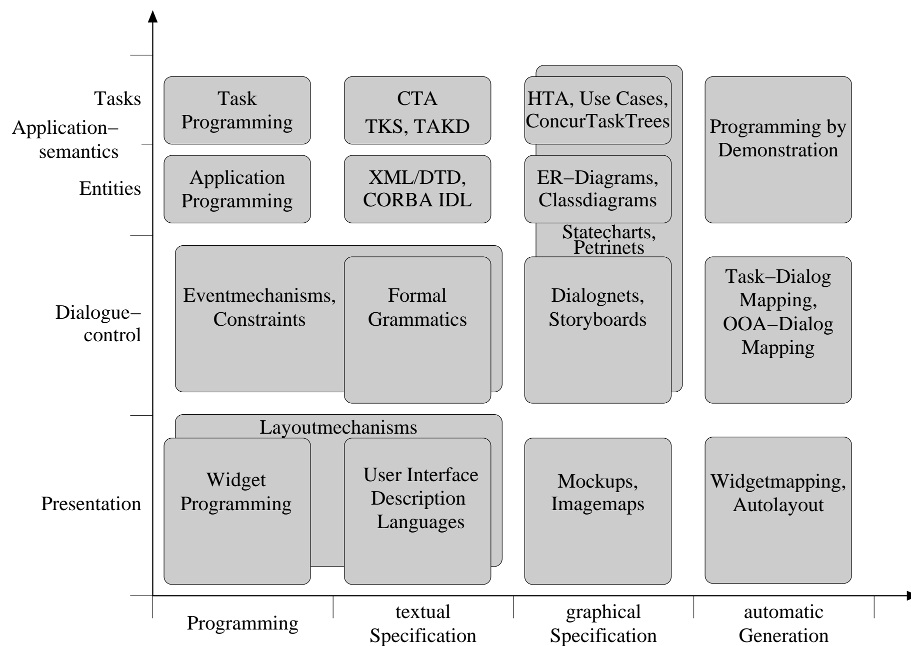


Abbildung 2.10: Klassifikation von Techniken (nach [Fähnrich 1995])

Nicht weiter eingehen werden die Autoren im Folgenden auf Mockups und Storyboards. Diese Techniken dienen nur zur Kommunikation mit den künftigen Benutzern in frühen Phasen der Entwicklung, wenn noch keine ausführbaren Prototypen existieren. Da sie keine formale Semantik besitzen, dienen sie lediglich der Analyse und sind nicht im Rahmen eines konkreten User Interface Entwicklungssystems einsetzbar. Bei Mockups handelt es sich um gezeichnete Entwürfe von Dialogen. Storyboards dienen der Visualisierung des Kontrollflusses zwischen Bildschirmen. Beide Techniken werden zur Kommunikation mit den Benutzern eingesetzt. Die Benutzer können dadurch schon früh Einfluss auf die Entwicklung nehmen und so sicherstellen, dass die entstehende Anwendung ihren Bedürfnissen entspricht.

Abgesehen von Event- und Constraintmechanismen sind die Techniken jeweils auf ein Abstraktionsniveau begrenzt. Event- und Constraintmechanismen können sowohl direkt auf programmiersprachlicher Ebene implementiert als auch mit Hilfe deklarativer Sprachen beschrieben werden.

Nicht alle Techniken dienen der Beschreibung bestimmter Benutzungsschnittstellenaspekte. Es haben sich auch einige Methoden etabliert, die Teile der Benutzungsschnittstelle aus anderen Aspekten automatisch erschließen. Weit verbreitet sind z. B. die sogenannten Widgetmapping Techniken. Dabei wird eine Reihe von Regeln vorgegeben, die zu einer geeigneten Auswahl bestimmter Widgets genutzt werden können. Auswahlkriterien sind z. B. der Typ der dargestellten Entitäten, oder eine Beschränkung der zulässigen Werte. Mehrere der in Abschnitt 3.2 vorgestellten Werkzeuge setzen diese Technik ein. Es gibt auch Ansätze zur Automatisierung des Dialoglayouts. Dabei handelt es sich jedoch eher um Forschungsprojekte denn um praktisch einsetzbare Methoden.

Zur Automatisierung der Dialogsteuerung gibt es Ansätze, die aktuell durch-

führbare Benutzeraktionen aus einem Taskmodell ableiten. Systeme, die Programming by Demonstration (PBD) einsetzen, analysieren fortwährend die Benutzereingaben und versuchen, daraus eine allgemeine Benutzerintention zu extrahieren. CAD-Systeme nutzen diese Technik, um dem Benutzer einmal eingegebene Konstrukte für nachfolgende Aktionen als primitive Objekte zur Verfügung zu stellen. Beispiel eines solchen Systems ist GIPSE [Patry Girard 1999], das PBD einsetzt, um das Taskmodell zur Laufzeit zu erweitern. Es implementiert zu diesem Zweck eine taskbasierte Dialogsteuerungsstrategie, die neue Tasks automatisch in die Benutzungsschnittstelle integriert.

2.2.2 Präsentationstechniken

Präsentationstechniken dienen dazu, die statischen Eigenschaften einer Benutzungsschnittstelle zu beschreiben. Dazu gehören sowohl strukturelle als auch stilistische Informationen. Durch Angaben über Hierarchie, Art und Position der Interaktionselemente wird ihre Struktur definiert. Mit Hilfe von Stileigenschaften wie z. B. Farben oder Fonts wird das Erscheinungsbild der Elemente beeinflusst.

Um eine Schnittstelle zu beschreiben, gibt es grundsätzlich verschiedene Möglichkeiten. Eine direkte Vorhensweise ist die Programmierung des User Interfaces aus Toolkit-Elementen. Diese „codezentrierte“ Beschreibung ist sehr einfach umsetzbar, da nur Wissen in der Zielsprache und über das verwendete Toolkit erforderlich ist, um eine interaktive Anwendung erstellen zu können. Allerdings ist diese Art des Vorgehens nur für sehr kleine Applikationen empfehlenswert, da größere Anwendungen bei Vermengung von Anwendungs- und User Interface Beschreibungscode schwer zu warten sind.

Um das Problem zu beheben, wurden Ressourcenansätze entwickelt, die eine strikte Trennung von Anwendungsquellcode und der Beschreibung der Benutzungsoberfläche ermöglichen. Die Ressourcen sind dabei unabhängig von der Programmiersprache und dem Compilertyp. Die Benutzungsoberfläche wird in einer oder in mehrern Ressourcedateien beschrieben, die von einem speziellen Ressource-Compiler übersetzt und zum restlichen Programm gebunden werden.

Neben dem Ressourcenkonzept gibt es auch verschiedene reine User Interface Beschreibungssprachen. Durch eine eigene Syntax und Semantik bieten sie die Möglichkeit, Beschreibungen einer Benutzungsschnittstelle vollständig aus dem Codecontext herauszuheben. Aber gerade in dieser Unabhängigkeit liegt auch die Schwierigkeit des Einsatzes begründet, denn die Verbindung zwischen beiden Welten muß durch einen geeigneten Mechanismus hergestellt werden.

2.2.2.1 Programmiersprachliche Beschreibung am Beispiel Java

Die plattformunabhängige Programmiersprache Java [Gosling et al. 1996] realisiert ein neuartiges Konzept, um Benutzungsschnittstellen zu beschreiben. Um eine Oberfläche aufzubauen, stehen die AWT- (abstract window toolkit) und die Swing-Klassenbibliothek zur Verfügung [Sun Microsystems 1998]. Grundlage ist ein Container-Prinzip, das es erlaubt, Komponenten in Containern zu plazieren und dadurch hierarchische Strukturen zu realisieren. Absolute Positionen für Elemente werden nicht verwendet, um mit den Grundeinstellungen verschiedener Plattformen zurechtzukommen. Stattdessen werden Layoutmanager für die Positionierung einzelner Elemente eingesetzt. Sie zerlegen den verfügbaren

Bereich in einzelne Zellen und ordnen diese den Oberflächenelementen zu. Die Regeln, nach denen eine Zergliederung in Teilbereiche stattfindet, machen den Typ des Layoutmanagers aus. Durch kombinierten und verschachtelten Einsatz des Container- und Layoutmanagerprinzips sind auch sehr komplizierte Aufteilungen realisierbar.

2.2.2.2 Ressourcen-Konzept am Beispiel Microsoft Windows(tm)

Die Beschreibung der Oberfläche besteht im Ressourcen-Ansatz aus der hierarchischen Auflistung der User Interface Elemente. Ausgehend von einem toplevel Fensterelement z. B. einem Dialog werden Gruppierungselemente wie Gruppen oder einfache Interaktionselemente definiert. Gruppen können neben einfachen Elemente auch weitere Gruppen beinhalten. Jedem Element wird eine feste Größe und Position zugeordnet. Die Maßeinheit dieser Angaben ist Dialogeinheiten [Microsoft 1995] und nicht Pixel, um eine Unabhängigkeit von den Systemvoraussetzungen wie Bildschirmauflösung und Fontgrößen zu erreichen. Die Positionsangaben für Elemente sind dabei immer relativ zum toplevel Element. In anderen Ressourcen-Ausprägungen (z. B. XMT, Open Interface) ist ein Bezug zum direkt übergeordnetem Element üblich.

2.2.2.3 User Interface Beschreibungssprachen

XUL Die XML-based User Interface Language (XUL) [Mozilla 1999a] wurde von der Mozilla Organisation zur einfacheren und effizienteren Weiterentwicklung des Mozilla-Webbrowsers entworfen. Mit Hilfe eines Renderers, der Gecko-Layout-Engine, werden die User Interface Beschreibungen aus den XUL-Dokumenten zur Laufzeit interpretiert und dargestellt (siehe Abb. 2.11).

Um die verschiedenen Aspekte eines Interfaces definieren zu können, bedient sich XUL verschiedener Techniken und kann somit selbst als Amalgram dieser Standards verstanden werden. Für die Beschreibung der Widgets stellt XUL Sprachelemente bereit, die nach und nach erweitert werden. Zusätzlich ist es erlaubt, beliebigen HTML Code in das Dokument einzufügen. Styleinformationen werden bei XUL in seperaten Cascading-Style-Sheet (CSS) Files [Bos et al. 1998] definiert und legen damit den sogenannten Skin der Applikation fest. Die Anbindung und das Verhalten der Interaktionselemente kann direkt im XUL-Dokument durch die Verwendung von Java-Script Code festgelegt werden. Auch die Anbindung an externen Applikationscode ist vorgesehen. Die Gesamtheit eines XUL-Interfaces wird als „Chrome“ einer Anwendung bezeichnet.

Ein XUL-Interface ist nur eine lose Sammlung nicht verbundener Widgets, bis durch eine Verhaltensbeschreibung eine Verbindung untereinander und zur Applikationslogik hergestellt wird. Da XUL beliebigen HTML Inhalt enthalten kann, ist eine direkte Verwendung von JavaScript Code [Netscape 1999] möglich. Einzelheiten findet man in [Mozilla 1999b]. Die Verbindung von XUL zum Applikationskern wird durch die Nutzung von XP-Technologie[4] erreicht. Eine einheitliche Sicht auf die Services der Applikation wird durch die Spezifikation der Schnittstellen in XPIDL (Cross Platform Interface Definition Language) erreicht. So wird sowohl von der Implementationssprache als auch von der

[4] XP steht für Cross Platform.

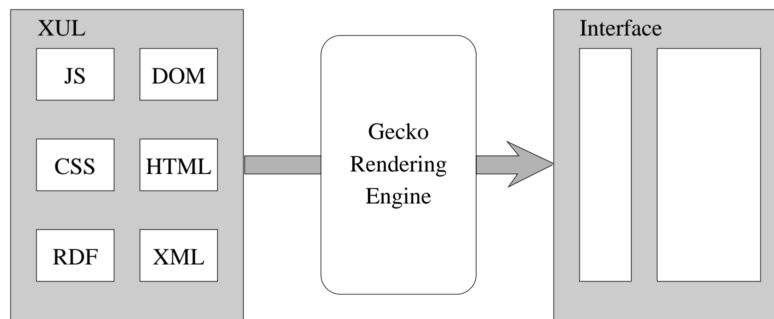


Abbildung 2.11: XUL-Komponenten (aus [Oeschger 2000a])

Plattform abstrahiert. Durch XPConnect Technologie wird die Verbindung mit JavaScript Code erzielt, wie ausführlich in [Oeschger 2000b] beschrieben.

Anmerkungen XUL wird im Kontext von Mozilla anwendungsnah entwickelt und besitzt daher ein ausgewogenes Verhältnis zwischen Komplexität und Funktionalität. Aus der Praxisnähe resultieren aber auch einige Beschränkungen des XUL-Ansatzes. Durch die Integration von Schlüsselworten für einzelne Widgets und direktem HTML-Inlinecode wird der Anwendungskontext auf grafische im Gegensatz zu nicht optischen Oberflächen limitiert. Die Definition von Schnittstellen, welche auf anderen Toolkits außer dem XPToolkit beruhen, wird weder konzeptionell angedacht noch durch die verwandten Techniken ansatzweise unterstützt.

UIML Die User Interface Markup Language (UIML) wurde an der Virginia Polytechnic Institute and State University entwickelt. Die Initiatoren gründeten 1997 die Unternehmung Harmonia [UIML 2000], welche UIML-Software wie z. B. Renderer für verschiedene User-Interface Toolkits vertreibt. Die Spezifikation UIML2 wurde als Vorschlag für einen offenen Standard beim W3C eingereicht. Die XML-konforme Beschreibungssprache wurde mit dem Ziel entworfen, Benutzungsschnittstellen geräteunabhängig zu definieren. So wird es möglich, mit geringem Modifikationsaufwand und Beibehaltung der Business-Logik eine Applikation auf verschiedenen Plattformen wie z. B. PCs, Handys, Handhelds auszuführen. Einzige Voraussetzung ist die Unterstützung der Zielplattform mit einem geeigneten Renderer. Für AWT, Swing und Voice-XML (VXML) sind bereits erste Testversionen verfügbar. Neben der Geräteunabhängigkeit war auch die Möglichkeit einer einfachen Internationalisierung der Oberfläche ein wichtiges Spezifikationskriterium. Dadurch können ohne Mehraufwand (abgesehen von der Übersetzung) Versionen eines User Interfaces in verschiedenen Sprachen realisiert werden. Die Architektur von UIML besteht aus diesen Bereichen:

- Die Beschreibung der User Interface Elemente (structure)
- Die Darstellungseigenschaften der Elemente (style)
- Das dynamische Elementverhalten (behavior)
- Die Inhaltsbeschreibung (content)

```
<?xml version="1.0"?>
<!DOCTYPE uiml PUBLIC "-//UIT//DTD UIML 2.0a Draft//EN"
    "http://uiml.org.dtds/UIML2_0a.dtd">
<uiml>
    <head>
        Metainformationen über das Dokument.
    </head>
    <interface>
        <structure>
            Hier werden Elemente (parts) deklariert.
        </structure>
        <style>
            Dieser Bereich enthält Darstellungseigenschaften.
        </style>
        <behavior>
            Verhaltensspezifikation der Elemente.
        </behavior>
        <content>
            Enthält Renderinginformationen zum Inhalt.
        </content>
    </interface>
    <peers>
        Mapping auf spezifische Toolkits.
    </peers>
    <template>
        Templates erlauben die Wiederverwendung von Code.
    </template>
</uiml>
```

Abbildung 2.12: UIML-Dokumentstruktur

Die klare Trennung der verschiedenen Aspekte führt zu einer sauberen Schnittstellenbeschreibung, die auch nachträgliche Modifikationen erleichtert, da sofort ersichtlich ist, in welchen Teil sie vorgenommen werden müssen. Auf Einzelheiten der Spezifikation eines UIML-Dokuments wie es in Abbildung 2.12 skizziert ist wird im Folgenden näher eingegangen.

Der zentrale Bestandteil eines UIML-Dokuments ist der Interface Abschnitt. Hier werden Struktur, Stil, Inhalt und das Verhalten von Oberflächenelementen angegeben. Innerhalb der Strukturbeschreibung (structure) werden die User Interface Bestandteile (parts) definiert, die später vom Renderer durch Widgets abgebildet werden. Weitergehende Renderinginformationen wie z. B. Schriftarten, Farben und Layouts werden im Darstellungsbereich (style) untergebracht. Um bestimmte Eigenschaften der Bausteine (parts, events, calls) festzulegen, werden Einträge verwendet (properties), die einzelne oder mehrere Elemente referenzieren. Die Property Namen sind nicht in UIML definiert, sondern können frei gewählt werden. So bietet es sich an, für ein Screendevice einen Propertynamen „Farbe“, für ein Audiodevice aber „Lautstärke“ zu definieren. Normalerweise werden diese Namen aber nicht vom Designer selbst definiert, sondern für ein Toolkit festgelegt und durch ein spezielles Element (peers) eingebunden.

Durch Einführung von Content-Abschnitten kann der Inhalt von der Struktur eines Widgets getrennt werden. In einem Contentbereich können Konstanten (constants) definiert werden, die dann aus anderen Teilen referenziert werden. So ist es möglich, durch die Bereitstellung verschiedener exklusiver Contentbereiche eine Internationalisierung des User Interfaces zu erreichen.

Das Verhalten wird in UIML durch Regeln (rules) festgesetzt. Eine Regel setzt sich zusammen aus Bedingungen (conditions) und Handlungen (actions). Interaktionen des Benutzers mit der Oberfläche führen zu Events, welche Einfluss auf den Status der Konditionen haben. Ist eine Bedingung erfüllt, werden die Anweisungen des zugeordneten Aktionsabschnitts ausgeführt. UIML erlaubt zwei konzeptionell verschiedene Arten, Aktionen zu definieren. Es ist sowohl möglich, direkte Skriptanweisungen wie z. B. JavaScript als auch externe Methodenaufrufe zu verwenden.

Anmerkungen UIML trennt die Applikationskomponenten von der User Interface Spezifikation. Dennoch können funktionelle Aspekte im Verhaltensabschnitt in beliebigem Umfang definiert werden, da nur so eine Verknüpfung der Oberflächenelemente mit der Applikationslogik herzustellen ist. Es ist nicht unproblematisch, die Beschreibung von dynamischem Verhalten nicht auf reine Aufrufe in die Applikationsebene zu beschränken und z. B. Korrektheitsprüfungen direkt mit Skripten in die Schnittstellenbeschreibung einzubetten. Dadurch würde die klare Trennung von User Interface und Logik aus Performancegründen aufgeweicht. So ist denn auch die Geschwindigkeit von Programmen mit UIML-definierter Schnittstelle zur Zeit noch ein Schwachpunkt. Die mangelnde Performance resultiert vor allem daraus, dass die Auswertung der UIML-Spezifikation zur Laufzeit stattfindet und dazu ein XML-Parser (DOM oder SAX) benötigt wird.

Nachteil von UIML-definierten Schnittstellen ist ferner die Komplexität der Dokumente, die schnell zu unübersichtlichen und schwer handhabbaren Beschreibungen führt. Um diesen Nachteil zu reduzieren, hat Harmonia neben dem UIML-Standard auch eine abkürzende Schreibweise (UIML2 Shorthand, siehe [Abrams 2000]) vorgeschlagen. Dokumente, die in der Kurzform erstellt werden, können dann mit Hilfe eines Konverters in UIML-dtd konforme Dateien übersetzt werden. Noch weiter verbessern ließe sich die Erstellung und Wartbarkeit von UIML-Dokumenten durch Entwicklung von User Interface Editoren, die als Resultat UIML-Beschreibungen produzieren. So könnte aus Designsicht weitestgehend von UIML-Code abstrahiert werden.

Andere Beschreibungssprachen Weder XUL noch UIML erlauben die abstrakte Schnittstellenspezifikation unabhängig von der Zielsprache. UIML erlaubt zwar die Integration beliebiger Toolkits, dennoch ist es notwendig, für jede Darstellungsform, wie z. B. HTML oder AWT, eigene Dokumente zu verfassen und diese synchron zu halten. Gerade bei sehr ähnlichen Toolkits, wie z. B. AWT und Swing, wäre ein automatisiertes Ableiten der konkreten Beschreibung aus einer abstrakteren Form wünschenswert. Genaueres findet man z. B. bei [Mueller et al. 2000], der diesen Ansatz verfolgt und eine eigene Beschreibungssprache sowie den halbautomatischen Generierungsprozess vorstellt.

Konkrete Beschreibungssprachen wie HTML, WML und VXML sind für Schnittstellenspezifikationen von Anwendungen nicht direkt geeignet, da sie al-

lesamt statisch sind. In die Benutzungsschnittstelle müssen jedoch während der Ausführung einer Anwendung dynamisch aktuelle Daten eingetragen werden. Mit Hilfe von Templatemechanismen läßt sich dieses Problem umgehen. Mit diesen Mechanismen lassen sich für verschiedene Beschreibungen wiederverwendbare Vorlagendokumente (templates) erstellen. Dabei können Templates direkt für konkrete Beschreibungssprachen erstellt werden, wobei das Auswerten der Templates die fertige Benutzungsschnittstelle z. B. in HTML liefert. Die Templates können aber auch auf der Ebene einer abstrakteren Beschreibungssprache wie UIML angewandt werden, wobei dann erst nach der Auswertung des Templates die UIML-Spezifikation vorliegt, die dann von einem UIML-Renderer verarbeitet werden kann.

2.2.3 Dialogtechniken

Dialogtechniken beschreiben die dynamische Sichtensteuerung des Systems. Man unterteilt die Dialogsteuerung in grobe und feine Dialogabläufe. Zu den groben Abläufen gehört die Abfolgensteuerung für Fenster und der Aufruf von Anwendungsfunktionen in Abhängigkeit von Benutzereingaben. Feine Dialogabläufe beschreiben die Zustandswechsel von Oberflächenelementen, die sowohl von anderen Interaktionselementen als auch von Systemeigenschaften abhängen können. In [Fähnrich 1995] findet sich eine ausführliche Darstellung und Bewertung bekannter Dialogbeschreibungstechniken. Neben den hier vorgestellten Ansätzen der Petrinetze, Zustandsübergangsdiagramme und Ereignismodelle finden sich dort außerdem Erläuterungen zu formalen Grammatiken und Constraints.

2.2.3.1 Ereignismodell

Der Benutzer löst durch Interaktionen mit dem System Ereignisse (Events) aus, die von der Anwendung verarbeitet werden. Sogenannte Ereignis-Behandler (Event-Handler) definieren Regeln, die Aktionen an ein Ereignis koppeln. Die Syntax der Regel-Spezifikationssprache orientiert sich meist an gängigen Programmiersprachen und entsprechend ähnlich ist das Aufstellen von Ereignisregeln zum Programmieren. Event-Handler spezifizieren für jedes Ereignis auf das sie reagieren die Ereignissignatur (z. B. die Daten die im Ereignisobjekt erwartet werden) und die Folge von Aktionen, die beim Auftreten des Ereignisses ausgelöst werden soll.

Das Ereignismodell wird in vielen Fenstersystemen eingesetzt und hat daher auch im Bereich der kommerziellen User Interface Management Systeme Verbreitung gefunden. Der Mechanismus ist sehr einfach und erlaubt dennoch die Beschreibung paralleler Dialoge durch separate Regelspezifikation. Zur Laufzeit können beliebig viele Regeln gleichzeitig aktiv sein, die dann die zu diesem Zeitpunkt möglichen Benutzeraktionen (=Ereignisse) bestimmen. Die separate Spezifikation der Regeln führt allerdings auch zu Schwierigkeiten, da das Verhalten des gesamten Systems aus den einzelnen Regeln heraus schwierig zu überblicken und nachzuvollziehen ist.

2.2.3.2 Petrinetze / Dialognetze

Ein Petrinetz wird durch einen Graph visualisiert, der aus Stellen und Transitionen besteht, die durch Kanten verbunden sind. Der Zustand eines Petrinetzes

ergibt sich aus seiner aktuellen Markierung, d. h. der Verteilung von Marken in den Stellen. Transitionen repräsentieren Aktionen, die unter geeigneten Bedingungen ausgeführt werden können und den Zustand des Netzes verändern. Stellen sind Behälter, in denen sich je nach Art des Petrinetzes[5] verschiedene Mengen und Typen von Marken befinden können. Eine ausführliche Einführung findet sich z. B. in [Jessen Valk 1987].

Petrinetze sind gut geeignet, nebenläufige Systeme zu beschreiben und beruhen auf einer ausgereiften Theorie, die auch formale Prüfungen wie z. B. eine Deadlockkontrolle zuläßt. Im Bereich der Dialogsteuerung werden sie seit Ende der 80er Jahre eingesetzt [Oberquelle 1987]. [Fähnrich 1995] stellt zu diesem Zweck die auf B/E-Netzen beruhenden Dialognetze vor. Sie wurden mit dem Ziel entwickelt, eine angemessene Beschreibungstechnik für Dialogabläufe auf Fensterebene bereitzustellen. Der Grundgedanke der Dialognetze besteht darin, dass die Stellen des Netzes Dialoge repräsentieren, die genau dann angezeigt werden, wenn die Stelle markiert ist. Entfernt eine Transition die Marke, wird das Fenster geschlossen. Zusätzlich werden einige wichtige semantische Erweiterungen eingeführt. So sind optionale Flüsse Elemente der Flußrelation, die zwar ein Token aus einer markierten Stelle entfernen, aber die Transition nicht blockieren, sollte die Stelle leer sein. Optionale Flüsse erlauben eine vereinfachte Darstellung der Netze. Weiterhin wurden modale Stellen zur Unterstützung der Modellierung modaler Dialoge eingeführt. Ist eine modale Stelle des Netzes markiert, können im nächsten Schritt nur Transitionen schalten, die diese als Eingangsstelle besitzen. Um die Handhabung großer Netze zu vereinfachen, gibt es Möglichkeiten, Netze hierarchisch zu strukturieren und durch Makros wiederkehrende Aufgaben vereinfacht aufzubauen. Voll spezifizierte Netze lassen es sogar zu, Bedingungen für den Aktivierungszustand einzelner Oberflächenelemente abzuleiten.

2.2.3.3 Zustandsübergangsdiagramme

Zustandsübergangsdiagramme basieren auf den von David Harel eingeführten Verallgemeinerungen der Konzepte von endlichen Automaten nach Moore und Mealy [Harel 1987]. Ein Zustandsdiagramm wird durch einen Graph repräsentiert, dessen Knoten den Zuständen entsprechen und dessen gerichtete Kanten die möglichen Zustandsübergänge (Transitionen) angeben. Ausgelöst werden Transitionen meist durch externe Stimuli, die als Ereignisse bezeichnet werden. Transitionen, dem Zustandsein- und austritt sowie dem Zustand selbst können Aktionen (Actions) zugeordnet werden. Ausführlich wird die Semantik der Statecharts z. B. in [OMG 2000a] und [Hitz Kappel 1999] beschrieben.

Die Verwendung von endlichen Automaten zur Beschreibung von Benutzungsschnittstellen geht zurück auf ein System, bekannt als Newman's Reaction Handler [Newman 1968]. Besondere Verbreitung haben Zustandsdiagramme für die Modellierung von Masken- und Menüsystemen gefunden [Denert 1991]. Daneben gibt es einige Ansätze, die Präsentationsschicht für HTML-basierte Webapplikationen mit Statecharts zu beschreiben [Anderson 1999]. Interessanterweise besitzt diese Kategorie Anwendungen Eigenschaften, die sie für die Modellierung durch Statecharts besonders geeignet erscheinen läßt. So kann z. B.

[5]Es existieren eine Vielzahl verschiedener Netzarten, z.B. S/T-, B/E-, gefärbte und Referenznetze.

die Zurück-Funktion (Back) sehr einfach durch einen History-Zustand ausgedrückt werden. Weiterhin besitzen die HTML-Anwendungen einen sequentiellen Kontrollfluß, der die Beschreibung mit Hilfe von Zuständen zusätzlich motiviert. Auch allgemeine Webnavigation mit Berücksichtigung nebenläufiger Zustände (Frames oder Windows) wurde untersucht und mit Statecharts realisiert [Leung et al. 1999].

Fähnrich kritisiert an Zustandsübergangsdiagrmmen zur Beschreibung von Dialogen ihre vornehmlich sequentielle Semantik [Fähnrich 1995]. Nach Anderson sind Statecharts noch nicht ausreichend für die Beschreibung von Transaktionen und Exceptions. Ausserdem würden die Diagramme schnell zu unübersichtlich. Anderson schlägt zur Lösung der angesprochenen Probleme einfache Erweiterungen der Statechartsemantik und -syntax vor [Anderson 2000a].

2.2.4 Domänentechniken

Unter dem Begriff Domänentechniken fassen die Autoren zwei verwandte Arten von Techniken zusammen. In Abschnitt 2.2.4.1 werden zunächst Techniken zur Beschreibung von Benutzeraufgaben beschrieben. Anschliessend wird in Abschnitt 2.2.4.2 auf einige Techniken zur Beschreibung von Einheiten der Anwendungsdomäne näher eingegangen.

2.2.4.1 Aufgabenbeschreibungstechniken

Taskbeschreibungstechniken sollen es ermöglichen, Aufgaben aus Sicht eines Systemanwenders zu formulieren und sollen damit helfen, die relevanten Abläufe eines Systems zu isolieren. Diese Techniken stehen in engem Zusammenhang mit Techniken für die Dialogmodellierung, obwohl ihr Aufgabenfeld prinzipiell auf die Modellierung von Systemaufgaben fokussiert sein sollte, gibt es einige Ansätze, welche die Beschreibung von Abläufen innerhalb der Benutzungsschnittstellen in den Vordergrund rücken. So dient z. B. die bekannte GOMS-Technik [Card et al. 1983] nahezu ausschliesslich der Messung von Performance-Eigenschaften von User Interfaces.

In den folgenden Abschnitten werden die Autoren einige ausgewählte allgemeine Techniken zur Modellierung von Systemaufgaben vorstellen.

Hierarchical Task Analysis (HTA) Bei der Hierarchical Task Analysis [Annett Duncan 1967] handelt es sich um eine der ältesten allgemeinen Aufgabenbeschreibungstechniken, die schon viele der Konzepte enthält, welche in späteren Arbeiten wieder auftauchen.

HTA beschreibt Aufgaben durch Operationen und Pläne. Operationen werden dabei als Aktivitäten angesehen, deren Ausführung dem Erreichen eines bestimmtes Ziels dient. Die Pläne enthalten Informationen über die Abfolge von Operationsausführungen und zusätzlich Bedingungen, welche festlegen, wann eine Operation ausgeführt werden soll. Die Operationen können hierarchisch in feinere Teiloperationen zerlegt werden. Faktisch entstehen dadurch Teilaufgaben, für die auch die Spezifikation von Plänen notwendig ist.

Diese Technik betrachtet damit Aufgaben als Aktivitäten, die zur Erreichung eines Zieles ausgeführt werden müssen. Ein Ziel wird als Systemzustand definiert, den es zu erreichen gilt. In der Notation werden die Ziele der Auf-

gaben jedoch nicht mit aufgeführt. Sie berücksichtigt lediglich die hierarchisch dekomponierten Operationen inklusive der Pläne [Welie 2001].

ConcurTaskTrees (CTT) Die Spezifikationstechnik ConcurTaskTrees wurden mit dem Ziel konzipiert, flexible und ausdrucksstarke Aufgabenstrukturen modellieren zu können, die auch von Menschen ohne formales Hintergrundwissen interpretiert werden können [Paternò 1999]. Aus diesem Grund besitzt die Technik eine visuelle Notation, die auch bereits mit einem frei verfügbaren Softwarewerkzeug bearbeitet werden kann. Die Entwicklung der ConcurTaskTrees wurde massgeblich von zwei Strömungen beeinflusst: dem modellbasierten User Interface Design und formalen Methoden der Mensch-Computer Interaktion.

Mit Hilfe der ConcurTaskTrees ist es möglich, Aufgaben hierarchisch in Baumstrukturen zu dekomponieren und mittels spezieller Operatoren in temporalen Bezug zu setzen. Diese Operatoren setzen auf der LOTOS [ISO 1988] Notation zur Beschreibung von Beziehungen nebenläufiger Aktivitäten auf und ergänzen sie um weitere Ausdrucksmittel. Die zeitlichen Bezüge können zwischen Aufgaben einer Ebene hergestellt werden und wirken sich in machen Fällen auch auf die nachgeordneten Aufgaben aus. So wurden Konzepte für die Vererbung von temporalen Beziehungen, die Rekursion und die Markierung von optionalen Aufgaben integriert. Zusätzlich sind Aufgaben typbehaftet und werden einer der Kategorien Benutzer, Computer, Interaktion oder Abstrakt zugeordnet. Dadurch wird verdeutlicht, welche Art von Entität eine bestimmte Aufgabe ausführen wird. Da der Typ einer Aufgabe von den Typen der nachgeordneten Aufgaben abhängt, kann in manchen Fällen keine eindeutige Zuordnung stattfinden und die Aufgabe wird mit dem Prädikat Abstrakt versehen. Die ConcurTaskTrees erlauben weiterhin die Spezifikation des internen Datenflusses, indem zu einer Aufgabe Objekte definiert werden können und Operatoren zur Weitergabe von Daten an eine andere Aufgabe existieren.

Use Cases UML Use Cases [OMG 2000a] beschreiben die Funktionalität eines Softwaresystems in Form von Anwendungsfällen aus der Sicht eines Benutzers. Außerhalb der Systemgrenzen befinden sich sogenannte Akteure (actors), die bestimmte Anwendungsfälle des Systems initiieren können. Akteure betrachten dabei das System als Black-Box und repräsentieren selbst keine Individuen, sondern Rollen, die Individuen oder Objekte einnehmen können. Innerhalb des Systems werden Anwendungsfälle definiert, die ein bestimmtes vom System erwartetes Verhalten beschreiben. Die Gesamtheit aller festgelegten Anwendungsfälle macht die Systemfunktionalität des Gesamtsystems aus.

Der Use Case Ansatz ähnelt in vielen Punkten den aufgabenbasierten Techniken. So werden in beiden Fällen Systemanforderungen aus der Sicht eines Anwenders zusammengetragen. Des weiteren repräsentieren die identifizierten Anwendungsfälle in etwa globale Aufgaben eines Aufgabenmodells. Im Unterschied zu vielen Techniken der Aufgabenbeschreibung fehlt den Anwendungsfällen aber die Möglichkeit zur klaren hierarchischen Strukturierung. Ausserdem wurden keine Mechanismen vorgesehen, um die temporale Ordnung zwischen verschiedenen Anwendungsfällen festzulegen. Es kommt hinzu, dass die Semantik der Use Cases sehr unterschiedlich ausgelegt wird und so leicht Kommunikationsschwierigkeiten und Missverständnisse aufkommen können.

2.2.4.2 Objektbeschreibungstechniken

Die hier vorgestellten Techniken erlauben die Spezifikation der Struktur domänenrelevanter Entitäten, die für das User Interface von Bedeutung sind (siehe Application Interface des Seeheim Modells, Abschnitt 2.1.1), d. h. es ist also in den meisten Fällen eine partielle Beschreibung des Gesamtsystems ausreichend. Aus dem Bereich des Softwareengineering resultieren viele verschiedene Techniken zur Modellierung eines Softwaresystems, wobei zwischen Ansätzen unterschieden werden kann, die sich auf reine Datenmodellierung konzentrieren (ER-Diagramme, XML) und denen, die zusätzlich Verhalten einbinden (Klassendiagramme, IDL).

Datenbeschreibungstechniken erschweren die Verwendung objektorientierter Analyse- und Designmethoden und eignen sich daher nur bedingt als Beschreibungsgrundlage für Domänenentitäten eines modernen User Interface Systems. In der Praxis existieren allerdings Systeme, die z. B. auf Entity-Relationship Diagrammen beruhen (wie TRIDENT, Abschnitt 3.2.5).

Entity-Relationship Diagramme Der Entity-Relationship Ansatz wurde von P. Chen [Chen 1976] eingeführt und bietet die Möglichkeit, das Datenmodell eines Systems auf abstrakte Weise zu spezifizieren.

Zur Modellierung werden genau abgegrenzte Entitäten (entities) und Relationen (relations) zwischen diesen Entitäten eingesetzt. Entitäten repräsentieren Objekte des Systems, denen bestimmte Eigenschaften (attributes) zugeordnet werden können. Die Attribute lassen sich in zwei Kategorien einteilen: Identifikatoren (identifiers) und Deskriptoren (descriptors). Ein Identifikator dient zur eindeutigen Bestimmung einer Entitätsinstanz, wohingegen deskriptive Attribute zur Spezifikation nichtidentifizierender Charakteristika verwendet werden. Relationen verkörpern Assoziationen zwischen einer oder mehreren Entitäten. Sie werden durch die Merkmale Kardinalität (degree), Konnektivität (connectivity) und Existenz (existence) charakterisiert. Die Kardinalität einer Relation beschreibt die Anzahl Enden einer Relation (binary, ternary, n-nary). Mit Hilfe der Konnektivität einer Relation läßt sich die Anzahl Entitätsinstanzen einer Relationsinstanz festlegen (one, many). Bei binären Relationen wird daher von 1:1, 1:n oder n:m Konnektivitäten gesprochen. Durch die Existenzbedingung wird bestimmt, ob eine Entitätsinstanz an ein Relationsinstanzende gebunden sein muss (mandantory, optional). Zum Verständnis einer modellierten Verbindung kann die Rolle (role) einer Entität in Beziehung zur Relation benannt werden, z. B. „nimmt Teil an“ oder „bearbeitet“.

Haupteinsatzgebiet der ER-Diagramme ist im Bereich der Modellierung von Datenbankapplikationen zu sehen, da ihre Struktur eine problemlose Übertragung in ein relationales Datenbankschema erlaubt. Das ursprügliche ER Modell wurde in vielerlei Hinsicht erweitert und stellt heutzutage Konzepte bereit, die sich auch in objektorientierten Techniken finden. So können z. B. Verallgemeinerungs- (generalization) und Aggregationsbeziehungen beschrieben werden. In [Teorey 1999] findet sich ein ausführlicher Vergleich zwischen der Modellierung mit ER- und UML Klassendiagrammen. Teorey kommt zu dem Schluss, dass sich die beiden Techniken bis auf den schon erwähnten Verhaltensaspekt sehr ähnlich sind.

XML Die eXtensible Markup Language (XML) [Bray et al. 2000] ist eine Empfehlung des W3C Consortiums zur Darstellung von strukturierten Daten in Textform. XML wurde als Teilmenge der Standard Generalized Markup Language (SGML, ISO 8879) definiert, um den Austausch strukturierter Dokumente über das Internet zu vereinfachen. Eine Markup-Sprache ist ein Mechanismus, mit dem Strukturen in ein Dokument eingefügt werden können. XML-Dokumente enthalten damit sowohl inhaltliche als auch strukturrelevante Informationen. Die XML Spezifikation definiert eine Standardmethode wie Markup Konstrukte in einen Text eingebettet werden können [Walsh 1998].

XML basiert auf dem Konzept, dass sich Dokumente aus einer Reihe von Entitäten zusammensetzen. Jede Entität kann hierarchisch weiter in logische Elemente dekomponiert werden und über Eigenschaften (attributes) zur näheren Qualifizierung verfügen. Attribute können durch einen Typ in semantisch verschiedene Gruppen eingeteilt werden. Dazu gehören u. a. auch Typen zur Festlegung (ID) und zur Referenzierung (IDREF) von Identitäten. Die Infrastruktur für Relationen zwischen Elementen ist damit bereits vorhanden.

Die erlaubte Struktur eines XML Dokumentes kann durch eine Document Type Definition (DTD) festgelegt werden. Technisch gesehen ähnelt dieser Formalismus formalen Grammatiken, die zur Spezifikation von Strukturen Produktionsregeln einsetzen. Durch die Beschreibung einer DTD wird die Menge an verfügbaren Elementen, Attributen und ihre Zuordnungen untereinander spezifiziert. Es wird faktisch eine neue Markup-Sprache mit domänenspezifischen Schlüsselwörtern geschaffen.

Die Einsatzgebiete von XML haben sich seit der Einführung stark erweitert. Zunächst wurde XML hauptsächlich für die Modellierung von Texten eingesetzt. Später erhielt XML auch steigende Beachtung als Formalismus für Daten- und Wissensmodellierung. Z. B. werden in [Buck 2000] Konzepte zur Erweiterung von XML zur Repräsentation relationaler Daten vorgestellt.

Klassendiagramme In diesem Abschnitt wird exemplarisch die Semantik des UML [OMG 2000a] Klassendiagramms erklärt. In vielen objektorientierten Analysemethoden finden sich ähnliche Konzepte, z. B. in OML [Firesmith et al. 1998] semantische Netze, OOA/OOD [Coad Yourdon 1991a, Coad Yourdon 1991b] das OOA Modell, in OMT [Rumbaugh et al. 1991] das Objektmodell und in OOSD [de Champeaux 1993] das statische Modell.

Das Klassendiagramm (class diagram) beschreibt die statische Struktur eines Systems mit Hilfe von Packages, Klassen, Interfaces und Beziehungen der Elemente untereinander. Eine Klasse repräsentiert eine Gruppe von Objekten mit ähnlichen Eigenschaften, d. h. sie bildet das zugrundeliegende Schema für konkrete Ausprägungen. Eine Klasse kann sowohl Eigenschaften, die durch Attribute beschrieben werden als auch Verhalten in Form von Operationen enthalten. Interfaces dienen dazu, Verhaltenstypen durch die Angabe von Operationssignaturen festzulegen. Klassen können zu Interfaces in einer Realisierungsbeziehung stehen, die ausdrückt, dass eine Klasse die Operationen beliebig vieler Interfaces zur Verfügung stellt und damit deren Verhaltenstypen adaptiert. Packages werden dazu benutzt, zusammengehörige Klassen und Interfaces in Einheiten zusammenzufassen.

Packages, Klassen und Interfaces können untereinander in verschiedenen Beziehungen stehen. Zwischen Interfaces und Klassen untereinander ist es möglich,

Verallgemeinerungsbeziehungen (generalizations) auszudrücken. Unter einer Generalisierung wird dabei eine taxonomische Beziehung zwischen einem allgemeineren und einem spezielleren Element verstanden. Das speziellere Element ist dadurch, dass es sowohl das Verhalten als auch alle Eigenschaften und Beziehungen des allgemeineren Elements besitzt vollkommen konsistent mit diesem und kann optional zusätzliche Informationen enthalten. Ferner gibt es in UML die Möglichkeit, Abhängigkeiten (dependencies) zwischen den Entitäten zu definieren, deren genaue Semantik durch Stereotypen (stereotypes) festgelegt werden kann. Damit können auch neue Arten von Abhängigkeiten durch entsprechende Stereotypen leicht eingeführt werden.

Durch die Definition einer Assoziation (association) kann eine Verbindung zwischen Klassen hergestellt werden. Eine Assoziation wird insbesondere durch folgende Eigenschaften charakterisiert: Kardinalität, Multiplizität (multiplicity), Navigierbarkeit (navigation) und Aggregation. Die Kardinalität einer Assoziation dient dazu, die Anzahl der Assoziationsenden zu erfassen. Man spricht in diesem Zusammenhang wie bei den Relationsbeziehungen in ER-Diagrammen von binären, ternären oder n-nären Assoziationen. Durch die Angabe der Multiplizität eines Assoziationsendes wird näher bestimmt, wieviele Instanzen dieses Endes der anderen Assoziationsseite zugeordnet sind. Erlaubt ist eine beliebige Mengenspezifikation, wie z. B. 0..1, 1..*. Die Navigierbarkeit einer Assoziation legt fest, ob ein Assoziationsende von einem anderen erreichbar ist. Mit der Aggregationseigenschaft kann spezifiziert werden, in welchem Verhältnis die Assoziationsseiten stehen (keine Aggregation, Aggregation, Komposition). Wird ein Ende als Aggregat (aggregate) gekennzeichnet, so ist das andere Ende ein Teil in einer Teil-Ganzes-Beziehung. Die Beschriftung eines Endes als Komposition (composite) führt dazu, dass das andere Ende ein nicht ohne das Ganze existenzfähiges und abhängiges Teil repräsentiert (starke Aggregation).

Eine detailierte Einführung in die Modellierung mit UML Klassendiagrammen anhand anschaulicher Beispiele findet sich beispeilsweise in [Wahl 1998].

CORBA IDL Die CORBA Interface Definition Language (IDL) ist Teil der umfangreichen Common Object Request Broker Architecture (CORBA) Spezifikation [OMG 2000b] und dient zur Beschreibung von Domänenentitäten unabhängig von der Implementationssprache. CORBA verwendet diese Spezifikationssprache zur einfachen Integration heterogener verteilter Anwendungen. IDL ist jedoch sehr allgemein gehalten und kann auch eingesetzt werden, um Domänenmodellierung losgelöst von der übrigen CORBA Infrastruktur zu betreiben.

IDL bietet umfangreiche Möglichkeiten, Entitäten aus der Anwendungsdomäne näher zu spezifizieren. Um eine Entität zu beschreiben, wird ein sogenanntes Interface definiert. In diesem können die Eigenschaften (attributes), versehen mit genauen Angaben des Typs und optionalen Einstellungen wie z. B. eine Beschränkung auf lesenden Zugriff (read-only), festgehalten werden. Die Funktionalität einer Entität kann durch die Spezifikation von Operationen (operations) modelliert werden. Es handelt sich dabei genauer gesagt um eine Festlegung der Nachrichtenschnittstelle, die die vom Element akzeptierten Aufrufe beschreibt, nicht aber um eine Verhaltensbeschreibung, die eine weitergehende Protokollspezifikation beinhalten müsste. Um auf diese Art und Weise definierte Elemente zu gruppieren und in zusammengehörige Einheiten aufzuteilen, hat IDL das Konzept der Module (moduls) eingeführt. IDL besitzt keine semantischen

Konzepte zur Modellierung von Relationen zwischen Entitäten außer den in der Objektorientierung üblichen Referenzbeziehungen.

Für IDL wurden von der OMG formale Abbildungen auf die gängigsten in der Praxis verwendeten Programmiersprachen wie z. B. Java, C, C++ entworfen. Um dieses Mapping in einfacher Form möglich zu machen, befindet sich IDL auf demselben Abstraktionsniveau wie die Zielsprachen und enthält keine höheren Konstrukte (z. B. zur Protokoll- oder zur Relationsdefinition).

2.2.5 Zusammenfassung

Ein wichtiges Kriterium zur Beurteilung von Techniken ist ihre Erlernbarkeit. Dabei haben die abstrakteren Techniken meist die geringere Lernschwelle. Die grafischen Techniken sind unter diesem Gesichtspunkt den textuellen und den Programmierungstechniken vorzuziehen. Automationstechniken müssen nicht erlernt werden, da sie dem Entwickler die Arbeit bestenfalls vollständig abnehmen. Ein weiterer Punkt betrifft die Standardisierung der Techniken. Unter den Präsentationstechniken gibt es lediglich bei den User Interface Beschreibungssprachen Ansätze zur Standardisierung, während die meisten Spezifikationstechniken für die Anwendungsebene auf formalen Definitionen beruhen und als Standard veröffentlicht sind.

Im Hinblick auf UbiComp spielt die Flexibilität einer Technik eine essentielle Rolle. Techniken für UbiComp sollten alle denkbaren Interfacemodalitäten in einer Weise unterstützen, dass Benutzungsschnittstellen für weitere Endgeräte im Verlauf der Anwendungsentwicklung hinzugefügt werden können. Dieses Kriterium wird nicht von den Programmiertechniken erfüllt, da diese nur geeignet sind konkrete Benutzungsschnittstellen zu entwerfen. Automationstechniken sind inhärent komplex und nicht immer praktisch ausgereift, stellen aber als Ergänzung zu den übrigen Techniken eine attraktive Option dar.

Damit sind unter den Techniken hauptsächlich die spezifikationsbasierten für UbiComp geeignet. Während die Programmiertechniken in heterogenen Umgebungen nicht flexibel genug sind, decken die Automationstechniken nicht die gewünschte Bandbreite an möglichen Anwendungen ab. Alle Spezifikationstechniken haben Vor- und Nachteile bezüglich Erlernbarkeit, Standardisierung, Flexibilität und Anwendungsbreite. Da keine einzelne Technik allen anderen vorzuziehen ist, und auch persönliche Präferenzen der Anwendungsentwickler eine Rolle spielen, sollte ein UbiComp System möglichst offen gegenüber den verschiedenen Techniken sein, um bei Bedarf (z. B. in unerwarteten Umgebungen) Alternativen anbieten zu können.

2.3 Werkzeuge

Neben den in Abschnitt 2.2 vorgestellten Techniken zur User Interface Gestaltung existieren auch Softwarewerkzeuge, die dem Entwickler während des User Interface Designprozesses assistieren. Diese Werkzeuge setzen die besprochenen Techniken programmatisch um und bieten dem Entwickler häufig eine einfach zu bedienende Schnittstelle. Gemeinsames Ziel aller Werkzeuge sollte es sein, diejenigen Menschen an der Gestaltung der Oberfläche teilhaben zu lassen, die wirklich etwas davon verstehen – die Designer. Leider ist das keine Selbstverständlichkeit wie z. B. an den Werkzeugklassen Toolkits und Frameworks zu

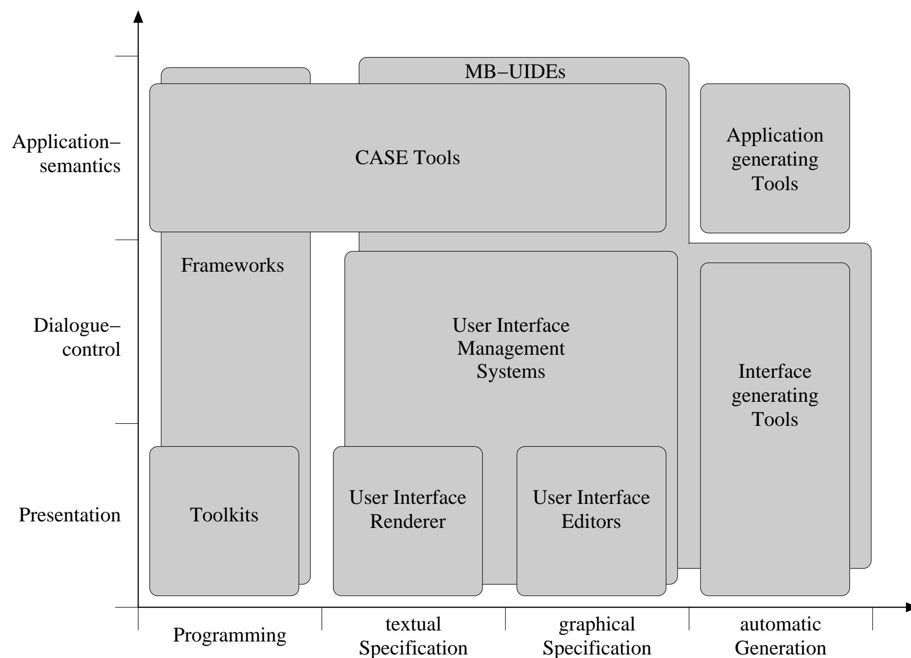


Abbildung 2.13: Klassifikation von Werkzeugen (aus [Fähnrich 1995])

sehen ist.

2.3.1 Übersicht über User Interface Werkzeuge

Die Klassifizierung der User Interface Werkzeuge erfolgt nach dem selben Schema (siehe Abb. 2.13), das die Autoren schon zur Einordnung der Techniken verwendet haben. Im Unterschied zu den Techniken fällt sofort auf, dass viele Werkzeugklassen sich über mehrere Felder erstrecken. Der Grund dafür ist, dass viele Tools eine möglichst umfassende Unterstützung anstreben.

Bevor die Autoren auf die dargestellten Klassen näher eingehen, möchten sie an dieser Stelle noch auf eine (triviale) Werkzeuggattung hinweisen, die aus Übersichtsgründen nicht mehr in das Diagramm aufgenommen wurde: „Papier und Stift“ (paper and pencil). In unser Diagramm fügt sich diese Klasse in die neun Felder, die zu den Spalten der Programmierung, der textuellen und der grafischen Spezifikation gehören. Zu dieser Kategorie rechnen die Autoren auch alle elektronischen Versionen dieser Werkzeuge, wie z. B. Malprogramme oder Texteditoren.

Trotz oder gerade wegen ihrer Einfachheit vereinen sie viele Stärken in sich. Anschauliche Zeichnungen der User Interfaces können mit wenig Aufwand erstellt werden und als Diskussionsgrundlage auch die Teamarbeit positiv beeinflussen. Der Hauptschwachpunkt dieser Werkzeugklasse besteht in der Unfähigkeit, Verhalten angemessen darzustellen. Außerdem ist es nur möglich, das Aussehen (look) nicht aber das mit der Bedienung verbundene Gefühl (feel) des User Interfaces zu testen, da keine ausführbare Version der Oberfläche zur Verfügung steht. „Papier und Stift“ sind trotz aller Schwächen die wahrscheinlich

populärsten Mittel, um User Interface Entwürfe anderen Personen zu präsentieren [Szekely 1994].

Aus Abb. 2.13 ersieht man, dass für die Programmierung der Präsentation eines User Interfaces Toolkits eingesetzt werden. Um Oberflächen abstrakter zu spezifizieren, wurden Werkzeuge entwickelt, die textuelle Beschreibungen interpretieren können (Renderer). Die Autoren werden in diesem Kapitel nicht explizit auf Renderer eingehen, da sie eng mit den verwendeten Beschreibungstechniken zusammenhängen (z. B. Gecko für XUL, UIML2Java Renderer, ...), die in Abschnitt 2.2.2 besprochen wurden. Weiterhin existieren User Interface Editoren, die eine grafische Erstellung einer Benutzungsschnittstelle zulassen und auf dieser Basis Teile des notwendigen Programmcodes in einer Zielsprache generieren. Frameworks setzen in den meisten Fällen auf Toolkits auf, bieten aber weitergehende Unterstützung zur Programmierung der Dialog- und Applikationssemantikkomponente.

Die Kategorien der UIMS und MB-UIDEs repräsentieren konzeptuell abstraktere Ansätze. UIMS erlauben die textuelle und grafische Spezifikation einer Oberfläche sowohl auf Präsentations- als auch auf die Dialogebene. Modellbasierte Systeme sind ursprünglich vom Automatisierungsgedanken geprägt und versuchen, möglichst viele Aspekte aus Modellinformationen abzuleiten. Moderne Systeme (in Kapitel 3 ist eine Auswahl zu finden) geben dem Entwickler die volle Kontrolle über alle User Interface Einstellungen und erlauben die textuelle oder grafische Spezifikation der Präsentations-, Dialog-, und Applikationssemantikschicht. Ebenfalls vom Automatisierungsgedanken geprägt sind die generatorbasierten Tools ohne Modellwissen. So gibt es z. B. Ansätze, nur aus Programmcode oder den Datenstrukturen einer Datenbank eine Benutzungsschnittstelle herzuleiten. Andere Werkzeuge gehen den umgekehrten Weg und versuchen, die Applikationssemantik aus Oberflächeninformationen zu erschliessen. Gerade für Designer ohne fundierte Entwicklungskenntnisse kann ein solches Vorgehen sinnvoll sein. Die Autoren werden in diesem Kapitel nicht auf Einzelheiten dieser Werkzeugklasse eingehen.

Eine ganz andere Art der Werkzeuge repräsentieren die CASE Tools. Sie stammen aus dem Softwareengineering und werden sowohl zur Systemanalyse als auch zum Systemdesign eingesetzt. Moderne Werkzeuge bieten dazu grafische Editiermöglichkeiten, die um textuelle Anmerkungen ergänzt werden können. Sie produzieren Applikationscode, der meist schon innerhalb des Werkeuges eingesehen und angepasst werden kann.

2.3.2 Toolkits

Ein User Interface Toolkit ist eine Sammlung von Interaktionstechniken, wobei eine Interaktionstechnik als Zusammenspiel aus der benutzerinitiierten Eingabe oder Manipulation eines Wertes und dem meist optischen Feedback des Systems definiert wird [Myers 1989]. Als Beispiele einiger Interaktionstechniken finden sich in grafischen Toolkits Menus, Scrollbars und Buttons. Natürlich kann ein Toolkit auch Elemente bereitstellen, die nur zur Repräsentation und nicht zur Manipulation eines Wertes dienen.

Die Verwendung von Toolkits stellt dem Entwickler eine Bibliothek mit Oberflächenelementen zur Verfügung und entbindet ihn von der Aufgabe, diese Elemente in Eigenregie zu erstellen. Das Erscheinungsbild der Applikationen wird dadurch standardisiert – wenn auch in Bezug auf ein spezielles Toolkit.

Außerdem bieten Toolkits neben den eigentlichen Interaktionselementen auch eine einheitliche Anbindung dieser Elemente an den fachlichen Applikationsteil an, z. B. durch einen Eventmechanismus.

Die Einführung von Toolkits schafft eine Basis für die User Interface Erstellung, löst aber bei weitem nicht alle assoziierten Probleme. Um User Interfaces allein mit Toolkits zu entwickeln, müssen die Designer Programmierkenntnisse besitzen. Zusätzlich sind die Elemente und Schnittstellen der Toolkits in vielen Fällen inkompatibel, d. h. eine Applikation, die unter Rückgriff auf ein bestimmtes Toolkit realisiert wurde, kann nicht ohne Anpassungsmaßnahmen auf ein anderes Toolkit aufgesetzt werden. Um Abhilfe zu schaffen wurden virtuelle Toolkits definiert, die eine allgemeine Schnittstelle bereitstellen [Rochkind 1992]. Des weiteren wird durch den Einsatz von Toolkits in keiner Form das Design einer Oberfläche oder die Dialogsteuerung unterstützt. Sie tragen auch nicht dazu bei, eine saubere Trennung zwischen Interface- und Applikationscode einzurichten. Stattdessen führen sie in der Regel zu einer großen Anzahl Callback-Methoden, die den Code leicht unübersichtlich werden lassen.

2.3.3 Interface Builder

Als User Interface Editoren oder GUI bzw. Interface Builder werden Softwarewerkzeuge bezeichnet, die es ermöglichen, ein User Interface aus elementaren Interface Bausteinen auf einfache Weise zu konstruieren. Dem Benutzer wird zu diesem Zweck eine WYSIWYG (what you see is what you get) Oberfläche geboten, in die er Elemente einfügen kann und von der er eine sofortige optische Rückkopplung seiner Bemühungen erhält. Das Gros dieser Werkzeuge produziert als Ergebnis der optischen Spezifikation Code, der direkt in eine Applikation gelinkt werden kann. In vielen Fällen muss daher darauf geachtet werden, dass die Zielsprache von Applikation und Interface Builder übereinstimmen. Um es mit den Worten von [Myers et al. 1999, S. 7] zusammenzufassen: “In these systems, simple things can be done in simple ways”.

Ein wichtiger Grund für den Erfolg der Interface Builder ist, dass die Werkzeuge grafische Mittel einsetzen, um grafische Konzepte auszudrücken. Die daraus resultierende Verlagerung einiger Aspekte der User Interface Implementation weg von herkömmlichem Code zu einem interaktiven Spezifikationssystem führte dazu, dass diese Aspekte der User Interface Implementation auch Nicht-Programmierern zugänglich gemacht wurden. Auch die Bedienung der Programme war durch eine intuitive Oberfläche einfach zu erlernen. Somit waren Domänenexperten in der Lage, speziell auf ihre Aufgaben zugeschnittene Oberflächen zu erstellen und Designer konnten einfacher und intensiver in die Oberflächengestaltung mit einbezogen werden.

Weiterer Vorteil dieser Werkzeuge ist, dass sie schnelle turnaround Zeiten im Entwicklungsprozess fördern. Zum einen, da sie eine interaktive Umgebung bereitstellen, in der alle relevanten Eigenschaften der Oberfläche justierbar sind und zum anderen, weil sie die Möglichkeit bieten, schnell zwischen Konstruktions- und Anwendungssicht umzuschalten.

Für die industrielle Verbreitung der Interface Builder hat letztendlich auch die Tatsache gesorgt, dass diese Werkzeuge direkt dafür eingesetzt werden können, auslieferbare Applikationen zu fertigen, d.h. das Ergebnis sind nicht prototypische Zwischenprodukte, die mit grossem Aufwand umgestellt werden müssen, sondern verwertbare Stufen in Richtung auf ein Endprodukt.

Der Einsatz von Interface Buildern bringt allerdings auch Probleme mit sich. In [Szekely 1994] werden drei Schwachstellen identifiziert:

Unter dem „main window“ Problem versteht Szekely die Einschränkung der Interface Builder, nur die statischen Teile einer Oberfläche spezifizieren zu können. Sie können nicht verwendet werden, um das „main window“ einer Applikation zu beschreiben, wenn dort direkt manipulierbare applikations-spezifische Informationen grafisch dargestellt werden sollen, z. B. die Formen in einem Malprogramm oder die Noten in einem Musikeditor. Interface Builder erlauben lediglich die Reservierung des Platzes, in dem die Informationen dargestellt werden sollen. Um diesen Raum mit Inhalt zu füllen, muss wiederum programmiert werden. Damit reduziert sich die allgemeine Anwendbarkeit der Interface Builder, und Designänderungen am Hauptfenster sind aufwändig zu integrieren.

Eine weitere Schwachstelle der Interface Builder liegt in der Schwierigkeit begründet, die Oberfläche vom fachlich relevanten Teil der Anwendung zu trennen. Interface Builder zwingen den Entwickler, eine grosse Anzahl Callback-Methoden auszufüllen, die bei Interaktion des Endnutzers mit der Anwendung aufgerufen werden. Änderungen an der Oberfläche führen dazu, dass diese Methoden überarbeitet werden müssen, um ein ausführbares Interface zu behalten. Dadurch verschlechtern sich die turnaround Zeiten und die einfache Benutzbarkeit wird relativiert. Zusätzlich bieten die Interface Builder keine Unterstützung, um noch nicht implementierte Applikationsteile zu simulieren. Die simulierten Informationen müssen betroffenen Interaktionselementen über selbstgeschriebene Callback-Methoden zugänglich gemacht werden.

Problematisch an Interface Buildern ist ferner, dass sie den Entwickler nötigen, die Oberfläche mit konkreten Interfaceelementen zu gestalten, bevor die Entscheidung über die Optik eines Interaktionselements angebracht ist. Damit sind Interface Builder nicht dafür geeignet, Informationen der Oberflächenfunktionalität separat von der Optik zu entwerfen.

2.3.4 Frameworks

Frameworks sind Softwarebibliotheken, die nicht nur Bausteine für das User Interface enthalten, sondern auch Klassen zur Entwicklung der eigentlichen Anwendung [Fähnrich 1995]. Sie geben Teile der Architektur und des Verhaltens für die zu entwerfenden User Interfaces vor. Dadurch unterscheiden sie sich von reinen Klassenbibliotheken wie Toolkits. Sie implementieren das „inversion of control“ Konzept, welches besagt, dass das Framework den Ablauf der Anwendung steuert und der Anwendungsprogrammierer sich nicht um das Anlegen und Verwalten von Objekten einer Klassenbibliothek zu kümmern braucht.

Grundsätzlich wird zwischen Black- und Whitebox Frameworks unterschieden. Blackbox Frameworks erlauben nur eine Schnittstellensicht auf die zur Verfügung stehende Funktionalität. Der Entwickler hat also die Möglichkeit, Frameworkobjekte zu erzeugen und durch meist separate Steuerungselemente zu konfigurieren. Whitebox Frameworks erlauben die Spezialisierung bestimmter frameworkeigener Klassen und erfordern daher Kenntnisse über Interna des Frameworkdesigns.

Diese Werkzeuge bieten eine umfassendere Unterstützung als Toolkits und verbergen in vielen Fällen das zugrundeliegende Fenstersystem vollständig. Applikationen werden dadurch portabler und durch die vorgegebene Architektur systematischer und besser wartbar. Da der wiederverwendbare Code des Frame-

works meist intensiv und in verschiedenen Kontexten getestet und eingesetzt wird, erhöhen Frameworks auch die Robustheit der Anwendungen. Weiterhin bieten Frameworks eine größere Flexibilität und Mächtigkeit als die meisten deklarativen Werkzeuge. Sie lassen sich beliebig erweitern und Anwendungsteile, die sie nicht unterstützen, lassen sich weiterhin direkt programmieren. Deshalb unterliegen Frameworks keinerlei Einschränkungen bezüglich der Systeme, die mit ihnen entwickelt werden können.

All diese Vorteile werden jedoch teuer erkauft, denn das User Interface und vor allem die Anbindung an den fachlichen Kern werden in derartigen Systemen weiterhin programmiert. Frameworks bieten insofern keine Abstraktion von der Komplexität der Schnittstellensprogrammierung. Um diese Schwäche etwas abzumildern, werden Frameworks oft im Zusammenspiel mit Interface Buildern eingesetzt. Ein Nachteil von Frameworks ist außerdem ihre komplexe, ohne umfangreiche Dokumentation schwer zu überblickende Struktur und eine daraus resultierende lange Einarbeitungszeit.

2.3.5 UIMS

User Interface Management Systems (UIMS), zu deutsch Systeme zur Verwaltung von Benutzungsschnittstellen, sind Softwarekomponenten, welche Benutzungsschnittstellen auf Basis abstrakter Spezifikationen ausführen. Der Begriff UIMS wird gelegentlich auch weiter aufgefasst, so dass er Systeme mit einschließt, die generell den Entwurf von Benutzungsschnittstellen unterstützen. In dieser Arbeit wollen die Autoren unter UIMS jedoch lediglich die Laufzeitumgebungen verstehen, die User Interface Beschreibungen interpretieren und daraus interaktive Programme ausführen. Entwicklungssysteme, die auf abstrakten Spezifikationen aufbauen, besprechen die Autoren in Abschnitt 2.3.6 (MB-UIDEs).

User Interface Management Systeme waren als Analogon zu Database-Management-Systemen (DBMS) gedacht. Im Gegensatz zu DBMS haben sich UIMS jedoch nie richtig durchgesetzt und seit den frühen achtziger Jahren ein Nischendasein gefristet. Ein Grund dafür war, dass UIMS eine Abstraktion von darunterliegenden Betriebssystemkonzepten (z. B. Fenstersysteme) anstrebten. Dies sollte der Abstraktion entsprechen, die Datenbanksysteme gegenüber Festplatten und Dateien bieten. Durch Standardisierung von User-Interface Elementen (Toolkits) in den späten Achtzigern und die damit einhergehende Vereinfachung der Interfaceentwicklung, wurde diese Abstraktion zunächst überflüssig [Myers et al. 1999].

UIMS erzwingen eine vollständige Separierung der Benutzungsschnittstelle von der Anwendungsfunktionalität. Der Vorteil ist, dass Benutzungsschnittstellen damit portabel werden. Wenn es für ein UIMS eine Portierung auf ein bestimmtes System gibt, können damit auch sämtliche zu diesem UIMS kompatible Spezifikationen von Benutzungsschnittstellen ausgeführt werden. Es bietet sich daher an, eine UIMS-Anwendung als Client / Server System zu realisieren, wobei die Schnittstelle zur Anwendung den Server stellt, und die Laufzeitumgebung der Benutzungsschnittstelle als Client eine Verbindung zum Anwendungsteil herstellen kann.

Heutzutage, im Zusammenhang mit Ubiquitous Computing, werden Ansätze wieder interessant, die Benutzungsschnittstellen abstrakt spezifizieren. Diese können so verschiedenste Endgeräte ansprechen, auf denen nach Voreinstellun-

gen des Benutzers hochspezialisierte Ausgaben generiert werden. Das Internet wird mittlerweile von einer breiten Masse benutzt, die über stark divergente Fähigkeiten und technische Ausrüstung verfügt. Es ist daher wünschenswert, für bestimmte Benutzergruppen (z. B. alte Menschen, Kinder, Menschen mit Behinderungen, ...) und spezielle Endgeräte (z. B. PCs, Handys, ...) maßgeschneiderte Schnittstellen anbieten zu können. Diese Fähigkeit, Anwendungen in Abhängigkeit des lokalen Kontextes ausführen zu können, sehen Myers et al. als Teil bzw. Erweiterung von Betriebssystemen der jeweiligen Endgeräte an. Sie schlussfolgern:

> „However, we believe that user interface design is poised for a radical change in the near future, primarily brought on by the rise of ubiquitous computing, recognition-based user interfaces, 3D and other technologies. [...] It will be important to have replaceable user interfaces for the same applications to provide different user interfaces on different devices for ubiquitous computing, and to support customization.“ [Myers et al. 1999, S. 25, 26]

2.3.6 Modellbasierte Werkzeuge (MB-UIDEs)

In die Kategorie Model Based User Interface Development Environments (MB-UIDEs) fallen alle Werkzeuge, die auf einer deklarativen Repräsentation eines User Interfaces aufsetzen. Im modellbasierten Paradigma spezifiziert der Entwickler ein deklaratives Modell des User Interfaces durch die Angabe von Informationen zu verschiedenen Bereichen. Dazu gehören in der Regel Angaben über die funktionalen Fähigkeiten des Systems, die von Benutzern durchzuführenden Aufgaben, Informationen zur Benutzungsschnittstelle, Anwendercharakteristika und die Ein- / Ausgabe-Techniken, die von der Zielplattform unterstützt werden. Durch Auswertung der Modellinformationen wird das Systemverhalten bestimmt [Szekely et al. 1995].

Die Vertreter der ersten Generation von modellbasierten User Interface Werkzeugen (z. B. UIDE [Foley et al. 1991], MIKE [Olsen 1992]) erschienen als Weiterentwicklungen früherer UIMS. Sie besaßen nur einen geringen Abstraktionsgrad, da sie lediglich auf die Ausführung eines abstrakt spezifizierten User-Interfaces abzielten. So mussten konkrete Design-Entscheidungen schon in frühen Entwurfsphasen getroffen werden. In den neueren Generationen, können Entwickler Oberflächen auf abstrakterem Niveau spezifizieren und anschließend generieren und ausführen. Es werden CASE Tools und abstrakte Modellnotationen eingesetzt, und dem Entwickler stehen oft umfangreiche Design- und Laufzeitwerkzeuge zur Verfügung [da Silva 2000]. In Abschnitt 3.2 wird die Evolution modellbasierter Systeme an Hand der von ihnen verwendeten Modelle näher besprochen.

Laufzeit-Architekturen modellbasierter User Interface Werkzeuge

In Abb. 2.14 sind drei konzeptionell verschiedene modellbasierte Ansätze zur Erzeugung eines ausführbaren User Interfaces dargestellt [da Silva 2000]. Die Gruppe der Source Code Generatoren produziert Quellcode für die Benutzungsschnittstelle in einer Zielsprache. Manche Systeme erzeugen zusätzlich auch den Coderahmen für den fachlichen Kern. Der für das User Interface bestimmte

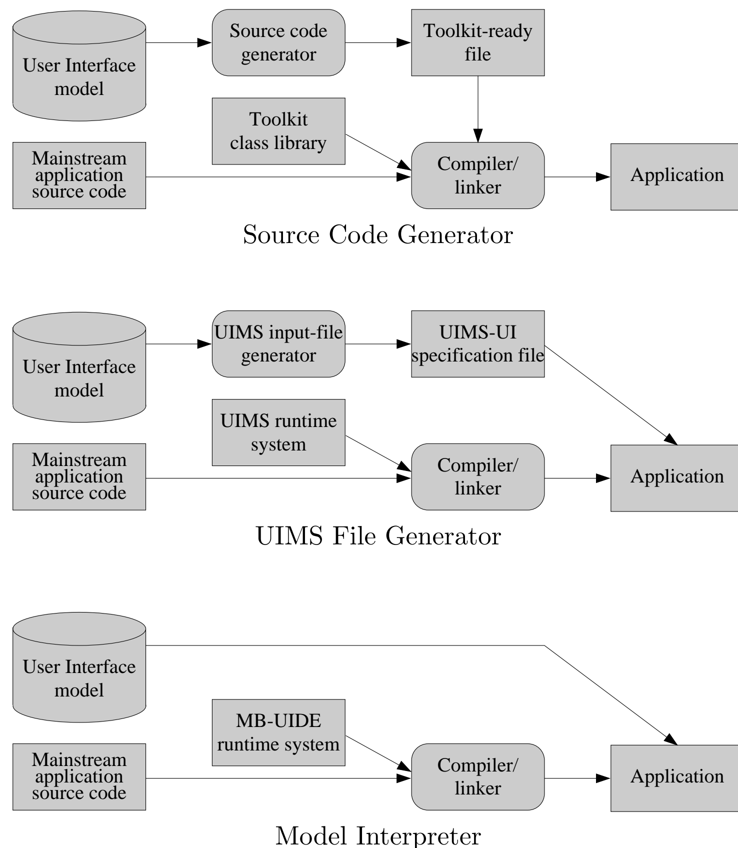


Abbildung 2.14: User Interface Laufzeitarchitekturen (aus [da Silva 2000])

Quellcode wird toolkit-spezifisch erzeugt und kann mit dem restlichen Anwendungscode durch herkömmliche Compiler/Linker in eine ausführbare Applikation übersetzt werden. In die Kategorie der UIMS-File Generatoren fallen alle modellbasierten Werkzeuge, die Modellinformationen unter Verwendung eines Generators in einem Zwischenschritt in eine UIMS Datei konvertieren. Ein zu der Applikation gelinktes UIMS Runtime System übernimmt die Ausführung der Oberfläche. Interpreterbasierte Ansätze extrahieren Modellinformationen zur Laufzeit, um daraus ein ablauffähiges User Interface zu erstellen. Die Auswertung der Modelldaten wird durch ein der Applikation hinzugelinktes MB-UIDE Runtime System vorgenommen.

Eine abschließende Beurteilung der drei Ansätze ist schwierig, da sich Pro- und Contraargumente zu jeder Kategorie finden lassen [da Silva et al. 2000]. Wählt man einen Source Code generierenden Ansatz, so hat die Applikation freien Zugriff auf alle durch die Zielsprache verfügbaren Möglichkeiten (z. B. Datenbank-, oder Remoteserverzugriff). Für die Konstruktion eines UIMS Systems spricht, dass die Funktionalitäten eines UIMS in vollem Umfang ausgenutzt werden können (z. B. kann das UIMS eine Laufzeitänderung der Interfacedarstellung anbieten). Interpreterbasierte Ansätze sind flexibel und erlauben,

Änderungen in den Modellen sofort in die laufende Applikation einzuspeisen. Ein sehr wichtiger Aspekt ist auch die Vermeidung des Postediting Problems,[6] unter dem alle generatorbasierten Systeme leiden. Demgegenüber steht jedoch ein nicht unwesentlicher Performancenachteil der interpreterbasierten Systeme, da die Auswertung von Modelldaten zur Laufzeit aufwändig und damit zeitintensiv ist. In der Praxis gibt es bereits einige Systeme, die einen hybriden Ansatz verfolgen. In der Modellierungsphase werden Interpreter eingesetzt, um eine schnelle iterative Entwicklung sicherzustellen. Ausgeliefert wird eine generierte Version, die auf Geschwindigkeit optimiert wurde (siehe MASTERMIND, Abschnitt 3.2.8).

Vorteile

[Szekely et al. 1995] betont folgende Vorteile des modellbasierten Paradigmas: Modellbasierte Systeme erlauben den Einsatz von mächtigen Werkzeugen. Das Modell stellt eine allgemeine Beschreibung dar, die von Werkzeugen gelesen bzw. geschrieben werden kann. So ist es möglich, Werkzeuge zur Unterstützung des Entwicklers während des Designs (design-time tools) und als Hilfestellung für den Anwender zur Laufzeit (runtime tools) bereitzustellen. Beispiele für design-time Werkzeuge sind Modelleditoren, automatisierende Werkzeuge, Design-Critics und Design-Advisors. Modelleditoren dienen dazu, Modelle visuell unterstützt zu editieren. Durch automatisierte Designwerkzeuge können Anteile des User Interfaces aus Modellinformationen deduziert werden. Design-Critics sind Programme, die das vom Entwickler spezifizierte Modell auf Schwachstellen untersuchen und diese anzeigen. Um den Entwickler nicht nur mit Designschwächen zu konfrontieren, sondern gleich geeignete Abhilfevorschläge anzubieten, können Design-Advisors eingesetzt werden. Im Bereich der runtime Tools finden sich kontextsensitive Hilfesysteme, Tools zur Auswertung der Benutzbarkeit (usability analyser) und Werkzeuge, mit denen Anwender das Programm an ihre Wünsche anpassen können (customization tools).

Die Konsistenz und Wiederverwendbarkeit der Komponenten eines Systems ist innerhalb und über Anwendungsgrenzen hinweg gewährleistet, da ihre Grundlage aus homogenem Modellwissen besteht. Durch eine semantische Dekomposition des User Interface Modells in einzelne funktional unterschiedliche Submodelle kann der Wiederverwendbarkeitsgrad noch erhöht werden. So ist es möglich spezielle Submodelle ohne grossen Umstellungsaufwand in anderen Kontexten erneut einzusetzen.

Das Verwenden von modellbasierten Entwicklungsumgebungen fördert auch ein frühes konzeptuelles Design, da die Verwendung deklarativer Modelle dazu führt, dass Designentscheidungen explizit gemacht werden müssen. Weiterhin bestärkt es die Entwickler, sich intensiver mit den Artefakten ihrer Arbeit auseinanderzusetzen.

MB-UIDEs tragen dazu bei, dass die Entwicklung einer Anwendung iterativ durchgeführt werden kann. Da die Modelle ausgeführt werden können bevor alle Details exakt spezifiziert sind, können Entwickler schon in frühen Entwicklungsphasen mit verschiedenen Entwürfen experimentieren. Dadurch wird es auch möglich, grundlegende Entwurfsfehler aufzudecken, bevor viel Aufwand in

[6] Dieser Begriff bezeichnet das Problem, dass generierte Quellen nicht von Hand editiert werden können, ohne den Verlust der Änderungen durch einen erneuten Generationsvorgang zu riskieren (siehe z. B. [Szekely 1996]).

die Implementation gesteckt worden ist. MB-UIDEs eignen sich daher gut für das Interface Prototyping [Szekely 1994].

In [da Silva 2000] werden weitere positive Aspekte modellbasierter Systeme aufgeführt: So ist es mit derartigen Systemen in der Regel möglich, User Interfaces in abstrakterer Form zu spezifizieren als dies bei herkömmlichen Werkzeugen möglich ist (man vergleiche dazu z. B. Nachteile der Interface Builder in Abschnitt 2.3.3).

Weiterhin begünstigt der Einsatz einer modellbasierten Entwicklungsumgebung die Einführung systematischer Entwicklungsmethoden oder sogar die Etablierung einer Methodologie für den Gesamtprozess. Nützlich dafür ist, dass User Interfaces in verschiedenen Abstraktionsgraden modelliert, Modelle inkrementell verfeinert und Interfacespezifikationen wiederverwendet werden können.

Die Autoren möchten dem noch hinzufügen, dass die Implementation des Systems in jeder iterativen Entwicklungsphase mit der durch das Modell gegebenen Spezifikation identisch ist. Diese projektinterne Konsistenz ist in herkömmlichen Entwicklungsprozessen oft ein Problem, da dort Modelle - sofern sie überhaupt eingesetzt werden - vor der Implementationsphase erstellt werden. Während der Implementationsphase werden sich die Entwickler jedoch oft Tatsachen gewahr, die in der Planung und damit in den Modellen nicht berücksichtigt wurden. Diese Änderungen fließen zumeist nicht zurück in die Modellbeschreibungen und verursachen Inkonsistenzen.

Fasst man die Modelle als Dokumentation eines Systems auf, so wird durch die inhärente Konsistenz der Modelle auch der Dokumentationsaufwand verringert, da die Modelle den aktuellen Systemstand festhalten. Weiterhin entsteht nicht das Problem, Dokumentation und Implementation synchron zu halten. Dennoch sollten auch die Modelle dort kommentiert sein, wo sie nicht selbsterklärend sind.

Probleme

Neben den angesprochenen Vorteilen bringt das modellbasierte Paradigma jedoch auch Probleme mit sich. Nachteile aktueller modellbasierter Werkzeuge sind nach [Szekely et al. 1995]: Die geringe Flexibilität der Systeme. Die Modellierungssprache existierender Werkzeuge ist nicht ausdrucksstark genug, um den Entwicklern die volle Kontrolle über alle Eigenschaften der Benutzungsschnittstelle zu geben. Gerade für die Erstellung praxistauglicher Anwendungen ist aber eine Steuerung beliebig feiner Details notwendig. Ausserdem sind die Werkzeuge dadurch in ihrer Anwendbarkeit eingeschränkt, z. B. auf Datenbankoberflächen oder auf formularbasierte Oberflächen.

Viele der verfügbaren Systeme bringen Geschwindigkeitsnachteile mit sich. Die schlechte Performance kann zum einem darauf zurückgeführt werden, dass viele Systeme sich noch im Experimentalstadium befinden und nicht auf Geschwindigkeitsaspekte optimiert wurden. Zum anderen hängt die Performance von der Art ab, wie und zu welchem Zeitpunkt die Modellinformationen ausgewertet werden. Systeme, die eine Laufzeitumgebung für die Interpretation einsetzen, können zwangsläufig nicht die Ausführungsgeschwindigkeit generatorbasierter Systeme erreichen.

Ein gravierender Nachteil der modellbasierten Systeme ist ihre hohe Lernschwelle (steep learning curve). In Anlehnung an das Zitat aus Abschnitt 2.3.3 kann man festhalten: “In these systems, complex things can be done but only in

complicated ways". Um das User Interface Modell zu spezifizieren, muss in den meisten Fällen zunächst eine komplexe Modellierungssprache erlernt werden. Für diese Modellierungssprachen existiert zur Zeit kein Standard, auf den die Systeme als gemeisame Basis zurückgreifen können. Gibt es keine Modelleditoren, so resultiert daraus eine weniger intuitive Art des Designs als beim Einsatz von grafischen Interface Buildern. Darüber hinaus wird Modellierung zu einer Form der Beschreibung, was nicht allen Interfacedesignern liegt.

In einer Untersuchung von 14 modellbasierten User Interface Werkzeugen identifiziert [da Silva 2000] die folgenden zur Zeit nicht gänzlich gelösten Probleme dieser Klasse von Entwicklungswerkzeugen: Es läßt sich schwer zeigen, dass Interfacemodelle alle relevanten Aspekte erfassen können, die nötig sind, um lauffähige Oberflächen zu generieren. Es gibt nur wenige Beispiele von Systemen, die in der Praxis erfolgreich eingesetzt wurden (z. B. ITS [Wiecha et al. 1990], mit dem Applikationen für die Expo '92 in Sevilla realisiert wurden). Das Problem, wie die Benutzungsschnittstelle mit der Applikationslogik verbunden werden kann, wird oft angesprochen, aber nicht vollständig gelöst. Es gibt keinen Konsens darüber, welche Menge von Modellen die bestgeeignetste ist, um Benutzungsschnittstellen zu beschreiben. Tatsächlich gibt es keine Übereinstimmung darüber, welche Teile einer Benutzungsschnittstelle überhaupt modelliert werden sollten.

Relativ unbestritten ist in der Literatur, dass eine vollautomatische Generierung des User Interfaces aus Modellinformationen nicht qualitativ hochwertig umsetzbar ist. In [Myers et al. 1999] wird die Brauchbarkeit automatisch generierender Systeme angezweifelt, da die Verbindung von Modell und konkretem Interface für den Entwickler schwer nachvollziehbar ist. Wenn für den Entwickler die Ableitung der Interfaces aus dem Modell nicht klar vorhersagbar ist, führt das zu einem trial-and-error statt zu einem methodischen Vorgehen (problem of unpredictability). In [Szekely 1996] wird eine Sequenz von fünf Schritten identifiziert, die ein automatisiertes Designwerkzeug im allgemeinen durchläuft: Bestimmung der Präsentationseinheiten, Bestimmung der Navigation zwischen den Präsentationseinheiten, Bestimmung der abstrakten Interaktionselemente für jede Präsentationseinheit, Mapping der abstrakten Interaktionselemente auf konkrete Interaktionselemente und die Bestimmung des Layouts der Präsentationseinheiten. Er kommt zu dem Schluß, dass keiner dieser Schritte vollständig automatisiert werden sollte. Stattdessen wäre es sinnvoll, dem Entwickler die Entscheidungen zu überlassen und ihn nur durch Vorschläge und alternative Wahlmöglichkeiten zu unterstützen. Szekely befürwortet daher eine Verlagerung von Automation zu Computer-Aided-Design.

2.3.7 CASE Tools

Computer-Aided Software Engineering (CASE) ist ein Oberbegriff für alle Arten von Werkzeugen, die die Entwicklung von Software unterstützen. Werkzeuge für die frühen Phasen des Software-Entwurfs (Anforderungsbestimmung, Analyse, Design) bezeichnet man als Upper-CASE Tools, Werkzeuge für die späteren Phasen (Implementation, Einsatz) werden als Lower-CASE Tools bezeichnet. Die Autoren beschränken sich hier auf die erste Kategorie von Werkzeugen, die dazu eingesetzt werden, Software-Systeme abstrakt zu repräsentieren, was gleichzeitig als Dokumentation und als Vorlage zur Implementierung dient.

CASE Tools helfen Softwareentwicklern, anerkannten Software-Engineering

Praktiken zu folgen. Sie assistieren in der Modellierung von Projektfunktionen, Informationsflüssen, Datenentitäten und anderen Informationen bezüglich angenommener Projektanforderungen. Auch Informationen über existierende Systeme können zu Dokumentationszwecken in diesen Werkzeugen gespeichert werden. Sie helfen, den Planungsprozess zu straffen und zu verbessern. Die in ein CASE Tool eingegebenen Informationen sind in einer allgemeinen Form vorhanden und können zur Unterstützung der Entscheidungsfindungsprozesse in einen breiten Bereich der Analyse einfließen.

Viele moderne Anwendungen werden basierend auf objektorientierten Technologien entwickelt und verwenden Konzepte wie Klassen, Methoden und Vererbung. Einer der Vorteile der Objektorientierung ist, dass ihre Konzepte sehr gut in grafischen Notationen zu repräsentieren sind. Objektmodellierungswerkzeuge bieten Unterstützung für objektorientierte Notationen und Methodologien und verwenden Generatoren, um einen Teil der Implementatierung zu automatisieren [Hebbel 1997].

Um die dazu notwendige Abstraktion zu erzielen, existiert bereits eine ganze Reihe an Notationen und Methodologien (siehe Abschnitt 2.2.4), aus denen sich mit UML [OMG 2000a] als Zusammenfassung mehrerer grafischer Beschreibungssprachen und Diagrammtypen ein verbreiteter Standard herauszukristallisieren scheint. Modellierungswerkzeuge unterstützten meist mehrere dieser Notationen. Sie sind in die Spalte der Werkzeuge zum strukturierten Editieren einzuordnen. Dabei konzentrieren sich viele CASE Tools vor allen Dingen auf die Modellierung der Anwendungssemantik. Einige Werkzeuge unterstützen auch die Beschreibung der Dialogkontrolle, wohingegen für den Entwurf der Präsentationsschicht wenige Hilfen angeboten werden. Außerdem dienen sie als Upper-CASE Tools eher der Dokumentation von Design-Entscheidungen denn der Erstellung eines lauffähigen Systems. Diese Aufgabe wird meist von Lower-CASE Tools wie Generatoren übernommen, die aus den Modellen Code-Gerüste erzeugen. Einige Tools (insbesondere im Bereich der Real-Time Systeme) unterstützen aber auch die (testweise) Ausführung des dynamischen Verhaltens direkt aus den Modellen.

Außer den unterstützten Methodologien gibt es noch weitere Unterscheidungsmerkmale dieser Werkzeuge. Dazu gehören die unterstützten Zielplattformen und -sprachen und inwieweit eine Integration mit Versionsmanagementsystemen möglich ist, um Revisionen der Modelle verwalten zu können.

Es ist interessant, dass CASE Tools oft mit dem Ziel angeschafft werden, den Softwareentwicklungsprozess zu beschleunigen und so Systeme schneller am Markt platzieren zu können. Tatsächlich aber ist zum produktiven Einsatz viel Erfahrung und Einarbeitung notwendig, so dass sich in den frühen Entwicklungsphasen durch den Werkzeugeinsatz kaum ein Zeitgewinn ergibt. Die Werkzeuge können aber die Softwarequalität verbessern, was einen geringeren Aufwand für spätere Test- und Wartungsphasen bedeutet [Netmation 2000].

Die Evolution der CASE Disziplin

Ende der 60er Jahre begann in der Softwareindustrie die sogenannte Softwarekrise. Die Software wurde immer komplexer, ohne dass es geeignete Methoden gab, diese Komplexität zu beherrschen. Damit wurde Software immer fehleranfälliger, und über 75% aller Softwareprojekte erreichten nicht die gesteckten Ziele. In dieser Zeit wurde schließlich der Begriff des Softwareengineering ge-

prägt. Es wird seitdem versucht, im Ingenieurbereich anerkannte Praktiken zur Qualitätssicherung auf die Softwareentwicklung zu übertragen [Gibbs 1994].

Die CASE Disziplin wurde in den 70ern formal definiert, motiviert durch ein wachsendes Bewusstsein, dass bei der Vermeidung und Entdeckung von potentiell kostspieligen Fehlern Design und Analyse eine wichtige Rolle spielen. Die erste Generation von Upper-CASE Tools in den späten 70er Jahren war noch textbasiert. Die Einführung von Personalcomputern und Workstations ermutigte die Softwarefirmen zur Entwicklung von grafischen Werkzeugen, die Analyse und Design unterstützten [Jorgensen 1994].

Die Entwicklung des computerunterstützten Softwareentwurfs vollzog sich in mehreren Phasen, in denen das Ziel kontinuierlich angepasst wurde. Die zweite Generation von CASE Werkzeugen wurde in den frühen 80ern entwickelt, um die Design und Analysephasen im Entwicklungsprozess besser zu unterstützen. Zuerst stellte CASE lediglich einen besseren Weg dar, um Designdiagramme zu zeichnen. Der Begriff stand für stand-alone Werkzeuge, die helfen sollten, Diagrammerstellung und Dokumentierung zu automatisieren [Netmation 2000].

Mitte der Achtziger Jahre wurden die Fähigkeiten von Analyse- und Designtools dahingehend erweitert, auch automatische Überprüfung von Entwürfen durchzuführen. Während dieser Zeit verbreitete sich zunehmend die Einsicht, Werkzeuge basierend auf einem zentralen Informationsrepository aufzubauen. CASE wurde nun eingesetzt, um den Fortschritt während der Entwicklung eines Systems in einem Repository zu speichern und um automatische Checks während der Design- und Analysephasen durchzuführen.

Als die Methoden für Analyse und Design ausgereifter wurden, wurden auch die Werkzeuge fehlertoleranter und waren besser in der Lage, die Einhaltung der Regeln der jeweiligen Methoden sicherzustellen. In den späten Achtzigern stabilisierte sich die zweite Generation von CASE Tools, und das objektorientierte Paradigma begann sich zunehmender Beliebtheit zu erfreuen. Mit dem Heranreifen der ersten objektorientierten Methoden kamen auch Werkzeuge auf den Markt, um dieses Paradigma in Analyse und Design zu unterstützen. Aus dem Repository wurde eine Brücke, die viele unterschiedliche Analyse- und Designwerkzeuge über eine zentrale Modellablage verband, und mit Generatoren konnten erstmals Teile der Implementierung automatisiert werden.

Anfang der 90er Jahre schließlich wurden CASE Tools dank verbesserter Benutzungsschnittstellen zur treibenden Kraft des Softwareentwicklungsprozesses. Die Modellinformationen eines Repositories werden als Möglichkeit gesehen, aus alten Bausteinen neue Systeme zu erstellen und so zu einer neuen Art der Softwareentwicklung zu führen.

Merkmale heutiger CASE Tools

Werkzeuge für die frühen Phasen eines Software-Lebenszyklus bieten meist Editoren für die Notationen, ein Repository, um Spezifikationen aus mehreren Projekten zu verwalten, Versionsmanagement, um mehrere Entwickler gleichzeitig an einem Projekt arbeiten zu lassen, Validierungs- und Verifizierungsfunktionen, um „korrekte“ Spezifikationen sicherzustellen, Reportgenerierung und eingeschränkte Codegenerierungswerkzeuge.

Objektorientierte CASE Tools erzeugen Diagramme, die ein Objektmodell mit den Notationselementen der entsprechenden Paradigmen repräsentieren. Die meisten, wenn nicht alle Werkzeuge, bieten die Möglichkeit an, aus diesen

Modellen Code-Rahmen zu generieren. Diese Code-Rahmen werden dann vom Entwickler mit Anwendungsfunktionalität in der jeweiligen Programmiersprache ausgefüllt. CASE Tools unterstützen aber nicht nur Objektmodellierung, sondern häufig auch logische Datenmodellierung und Geschäftsprozessmodellierung.

Sie bieten vielfältige Vorteile bei der Konstruktion von umfangreichen Systemen. Sie abstrahieren vom Quellcode auf ein Level, wo Architektur und Design augenscheinlich werden und damit leichter zu verstehen und zu modifizieren sind. Je größer ein Projekt ist, desto wichtiger ist es, CASE Technologien einzusetzen. Dies ermöglicht den Entwicklern, Teile eine Systems, die von anderen entworfen wurden, zu überblicken und zu verstehen, wie diese im System eingesetzt werden müssen. Das Management kann mit high-level Repräsentationen des Designs Projekte überblicken und den Fortschritt des jeweiligen Projektes insgesamt beurteilen. CASE Tools, zusammen mit Methodologien, bieten eine Möglichkeit Systeme darzustellen, die zu groß oder zu komplex sind, um sie allein aus dem Quellcode verstehen zu können [Hebbel 1997].

Als Folge des hohen Abstraktionsniveaus sind die mit CASE Tools erstellten Modelle auch für Personen zu verstehen, die keine Experten sind. Sie bilden daher eine ideale Basis, um während des Entwicklungsprozesses mit den Endnutzern zu kommunizieren. Um Modelle zur Dokumentation eines Systems einzusetzen, bieten sie die Möglichkeit, in die Modelle textuelle Anmerkungen einzufügen, die ebenfalls dem Revisionsmanagement unterliegen. Damit kann die Ausdruckskraft und Verständlichkeit der Modelle weiter verbessert werden.

Darüber hinaus lassen sich viele Werkzeuge in ihrer Funktionalität auch von Zusatzpaketen erweitern, die von Drittherstellern vertrieben werden oder die im Rahmen eines Projektes speziell entwickelt werden.

Ein weiterer wichtiger Aspekt an CASE Tools ist die Unterstützung von Entwicklungsteams. Einzelne Entwickler müssen unabhängig voneinander verschiedene Teile der Modelle bearbeiten können. Diese Modifikationen müssen zu gegebener Zeit wieder zusammengefügt werden können. Dazu verwenden einige Werkzeuge sogenannte Repositories, die unter der Aufsicht eines Versionsmanagementsystems stehen. Die Unterstützung mehrerer Entwickler ist allerdings nicht in allen Werkzeugen zur Zufriedenheit implementiert, insbesondere, wenn Modelle als eine grosse Datei abgelegt werden.

Um den Entwurfsprozess zu unterstützen, können Anforderungsdefinitionen durch alle Entwurfsphasen hindurch verfolgt werden (traceability). So ist jederzeit festzustellen, welche Konstrukte des Analyse-, Design- und Implementationsprozesses aus welchen Anforderungen hervorgehen.

Probleme

CASE Tools sind komplex. Um ein derartiges Werkzeug produktiv einzusetzen, ist eine lange Lernphase und viel Erfahrung erforderlich. Zusätzlich stellen CASE Tools hohe Anforderungen an die Hardware, was neben dem hohen Anschaffungspreis und notwendigen Schulungen zu weiteren Kosten führt.

Wird ein CASE Tools verwendet, um Anwendungscode zu generieren (forward engineering), führt dies zuerst dazu, dass das Modell und seine Implementation synchron sind, d. h., dass sie lediglich verschiedene Sichten auf dasselbe System darstellen. Wird in weiteren Iterationszyklen die Implementation immer weiter verfeinert und werden diese Verfeinerungen im Modell nicht reflektiert,

geht die Synchronität zwischen Modell und Implementation verloren. Das Modell verliert seine dokumentatorische Wirkung, da es nicht mehr das System beschreibt wie es zur Zeit implementiert ist [Hebbel 1997].

Um diesem Problem Herr zu werden, setzen viele Werkzeuge Reverse-Engineering-Mechanismen ein. Diese Mechanismen versuchen, Änderungen an der Implementation zurück ins Modell zu übertragen. Dazu werden im generierten Quellcode, meist in Form von Kommentaren, Markierungen platziert, die späteres Reverse-Engineering erleichtern sollen. Diese Markierungen machen den Quellcode allerdings auch unübersichtlicher und damit schwerer zu lesen.

Sowohl beim Forward- als auch beim Reverse-Engineering tritt das Postediting Problem auf. Werden Modell und Implementation unabhängig voneinander bearbeitet, werden beim erneuten Code-Generieren evtl. Änderungen an der Implementation überschrieben, beim Reverse-Engineering gehen vielleicht Modifikationen am Modell verloren. Es besteht die Hoffnung, dass in Zukunft integrierte Forward-Reverse-Engineering Techniken (round trip engineering), diese Probleme lösen können.

2.3.8 Zusammenfassung

Ein Problem bei der Verwendung von verschiedenen Werkzeugen ist die mangelnde Interoperabilität. Dies liegt u. a. in der generellen Schwierigkeit begründet, die Benutzungsschnittstelle an die Anwendungslogik zu koppeln, die durch den Einsatz unterschiedlicher Werzeuge zur Umsetzung dieser Komponenten noch verstärkt wird. Dies erklärt die geringe Verbreitung von UIMS, die als Komplettsystem zum Erstellen von User Interfaces unter dieser Problematik besonders leiden. Werkzeuge, die sich allein auf den Präsentationsaspekt konzentrieren, adressieren diese Schwierigkeit nicht, sondern überlassen der Dialogsteuerung die Anbindung an den funktionalen Kern.

Neben der Interoperatibilität ist auch die Anwendungsbreite, d. h. die Palette der konstruierbaren Anwendungen, entscheidend für den praktischen Nutzen eines Werkzeugs. User Interface Generatoren verfolgen einen interessanten Ansatz, sind aber in ihrer Anwendungsbreite eingeschränkt [Szekely 1996].

Da das Ziel des UbiComp die Entwicklung geräteunabhängiger Anwendungen ist, können CASE Tools unabhängig von Aspekten der Benutzungsschnittstelle bei der Implementation der Fachlogik eingesetzt werden. Um die Anbindungsproblematik zu berücksichtigen, sollten Werkzeuge zum Entwurf von Benutzungsschnittstellen jedoch alle drei Aspekte der Anwendungsentwicklung abdecken. Für ein vollständiges Benutzungsschnittstellensystem für das Ubiquitous Computing scheinen daher von den vorgestellten Werzeugklassen lediglich die Frameworks und MB-UIDES in ausreichendem Maße geeignet.

Kapitel 3

Untersuchte Systeme

In diesem Kapitel werden bekannte Werkzeuge zur systematischen User Interface Erstellung beschrieben. In Abschnitt 3.1 gehen die Autoren auf einige Vertreter der nicht-deklarativen Systeme (respektive Frameworks) ein und vergleichen diese miteinander. Anschliessend werden in Abschnitt 3.2 modellbasierte Systeme untersucht und einander gegenübergestellt. Am Ende wird in Abschnitt 3.3 kompakt beleuchtet, in wie weit sich die vorgestellten Werkzeuge der verschiedenen Kategorien als Konstruktionssysteme im Kontext des Ubiquitous Computing eignen.

Die Autoren unterscheiden die modellbasierten von den nicht-deklarativen Ansätzen. Nicht-deklarative Ansätze sind zumeist implementationsnah und bieten dem Entwickler praktische Design- und Umsetzungsunterstützung durch die Einführung neuer Abstraktionsebenen und vorgefertigter Softwarebausteine. Zu den nicht-deklarativen Ansätzen gehören MVC-Client (siehe Abschnitt 3.1.1), SanFrancisco (siehe Abschnitt 3.1.2), JWAM (siehe Abschnitt 3.1.3) und MVP (siehe Abschnitt 3.1.4).

Modellbasierte Systeme erlauben dem Entwickler, in einer frühen Entwicklungsstufe aufgestellte statische Modelle der Applikation für die Erzeugung des User-Interfaces und zum Teil auch anderer Aspekte wie z. B. Persistenz zu verwenden. Die modellbasierten Systeme unterscheiden sich untereinander sowohl stark im Umfang und in der Auswahl der Modellarten als auch in ihren zugrundeliegenden Realisierungskonzepten. Es kommt hinzu, daß es keinen Konsens in der Wahl eines geeigneten Metamodells und einer allgemeinen Modellierungsssprache zur Beschreibung aller möglichen Modelltypen gibt. Ein Ansatz in dieser Richtung ist das Mecano Projekt, dessen Vorschläge für ein ganzheitliches Metamodell MIM[1] und die einheitliche Beschreibungssprache MIMIC[2] im Mobi-D Projekt verwendet werden. Außer nach den verwendeten Modellen lassen sich die Systeme auch nach der Art der Verarbeitung der Modellinformationen klassifizieren (siehe Abschnitt 2.3.6). Man unterscheidet generatorbasierte, interpretative wie ITS [Wiecha et al. 1990] und hybride Ansätze wie Teallach und MASTERMIND (siehe Abschnitte 3.2.7, 3.2.8). Die generatorbasierten Ansätze lassen sich weiter in UIMS-File Generatoren wie FUSE, TRIDENT und TADEUS (siehe Abschnitte 3.2.4, 3.2.5, 3.2.6) und Quellcode Generatoren wie Janus und Mobi-D unterteilen (siehe Abschnitte 3.2.2 3.2.3). Eine genauere Er-

[1] MIM = Mecano interface model
[2] MIMIC = Mecano interface modelling language

örterung der Vor- und Nachteile der verschiedenen Konzepte wurde bereits in Abschnitt 2.3.6 gegeben.

3.1 Nicht-deklarative Ansätze

In diesem Abschnitt werden einige Ansätze zur unterstützten User Interface Entwicklung vorgestellt, die nicht auf deklarativen Modellbeschreibungen beruhen. Die Autoren beschränken sich auf die Betrachtung von Frameworks, da andere Werkzeugklassen wie z. B. GUI-Builder und Widgetbibliotheken nur Präsentationsaspekte eines Interfaces unterstützen und keine weitergehenden Konzepte für die Realisierung von interaktiven Anwendungen anbieten. Der fokale Punkt der Untersuchung liegt auf Applikations- und spezialisierten GUI-Frameworks. Es ist allerdings festzustellen, dass wenige reine GUI-Frameworks existieren. Viele Frameworks versuchen, Lösungen für möglichst viele softwaretechnische Problemstellungen anzusprechen (SanFrancisco, JWAM) und bieten sozusagen en passant auch eine User Interface Anbindung.

3.1.1 MVC-Client

Die MVC-Client Architektur [Sanderson 1999] wurde als Subsystem des Application-Frameworks der Firma Fourbit[3] entwickelt. Es handelt sich um eine Adaption des MVC-Designpatterns (siehe Abschnitt 2.1.2) auf zwei Ebenen, der Client- und der Attributebene (s. Abb. 3.1). Das Clientlevel behandelt Konzepte, die die Endnutzersicht auf die Applikation widerspiegeln, wohingegen sich die Attributebene mit einzelnen (atomaren) Eigenschaften der Clients auseinandersetzt.

Die Anwendung des MVC-Patterns auf Clientebene führt zu der Frage, wie Model, View und Controller repräsentiert werden und welche Aufgaben sie umsetzen. Das Model dieser Ebene wird als Client bezeichnet und enthält domänenspezifische Daten sowie Methoden, auf diese Daten zuzugreifen und sie zu manipulieren. Trotzdem ist der Client keine Implementation eines Businessobjektes, sondern vielmehr eine spezielle Sicht („Facade" Designpattern [Gamma et al. 1995]) auf Businessobjekte. Der Unterschied beider Objekte wird deutlich, wenn man sich das Ziel vor Augen führt, mit dem sie entworfen wurden. Clients sollen dazu dienen Benutzeraktionen auszudrücken, Businessobjekte hingegen enthalten geschäftsrelevante Daten und bilden Businessprozesse ab. Ein Client kann dazu die Eigenschaften mehrerer Businessobjekte oder auch nur eine Untermenge der Eigenschaften eines Businessobjektes verwenden. So kann der Umfang der in einem View eines Clients vorhandenen Informationen fein gesteuert werden. Unter einem View auf einen Client ist der clientrelevante Ausschnitt der GUI-Oberfläche zu verstehen. Der Controller fungiert als Bindeglied zwischen Model und View und reagiert auf Ereignisse der Views, die zu Änderungen im Model führen können.

Auf Attributebene wird das MVC-Paradigma in leicht abgewandelter Form eingesetzt. Statt View und Controller als getrennte Einheiten abzubilden, werden sogenannte Delegates (siehe Abschnitt 2.1.6) benutzt. Diese vereinen View- und Controller-Verhalten. Ergebnis der Verschmelzung sind Attribut-Widgets,

[3] http://www.fourbit.com

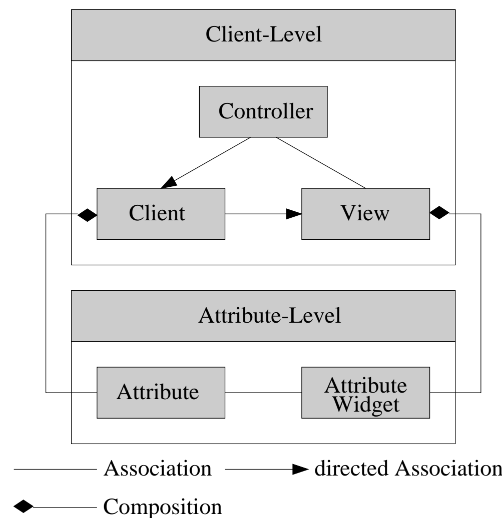


Abbildung 3.1: MVC-Client Architektur

die für die Darstellung und Interaktionsbehandlung von Attributen verantwortlich sind. Die Attribute bilden das Model dieser Schicht. Sie repräsentieren Entitäten des Objektmodells als atomare Dateneinheiten. Dabei ist der Begriff „atomar“ in der objekt-orientierten Entwicklung kontextsensitiv aufzufassen. Der Kontext entscheidet also darüber, ob ein Element atomar ist oder weiter verfeinert werden muß. Dadurch ist es möglich, daß ein Element im Kontext einer Anwendung als Attribut, im Kontext einer anderen als Client benutzt wird.

Die Verbindung der beiden Ebenen entsteht durch die Nutzung von Attributen durch die Clients. So faßt ein Client eine bestimmte Menge an Businessdaten zusammen. Um diese Daten zu konservieren, verwendet er aber keine primitiven Datentypen, wie z. B. Zeichenketten oder Ganzzahlen, sondern Attribute. Der View eines Clients verbindet die Attribute mit seinen entsprechenden Attribut-Widgets. Sämliche Widgeteigenschaften werden nun direkt durch die Attribute-Widgets in Abhängigkeit von Attributeigenschaften gesteuert. Dadurch enthält der View nur noch Layout- und Präsentationsspezifikationen, jedoch keine Verhaltensangaben mehr.

Business Rules, also Restriktionen für Businessobjekte, können im MVC-Client Framework sowohl in den Business-Objekten selbst als auch im Client untergebracht werden. Vorteil der Verlagerung in ein Clientobjekt ist, daß bei einer Client / Serververteilung der Applikation, Last vom Server genommen wird. Allerdings spiegelt ein Client die Sicht auf ein Business-Objekt wider und sollte daher auch nur Restriktionen einer Sicht beinhalten. Reine Businessrules sind eng mit dem Businessobjekt assoziert und werden daher zumeist im Businessobjekt selbst implementiert.

Anmerkungen

Die Verwendung des MVC-Frameworks führt zu einer strengen Trennung der Oberfläche von der Businesslogik. Die Clients realisieren Sichten auf Business-Objekte und können nach Belieben verändert oder ausgetauscht werden. Dies

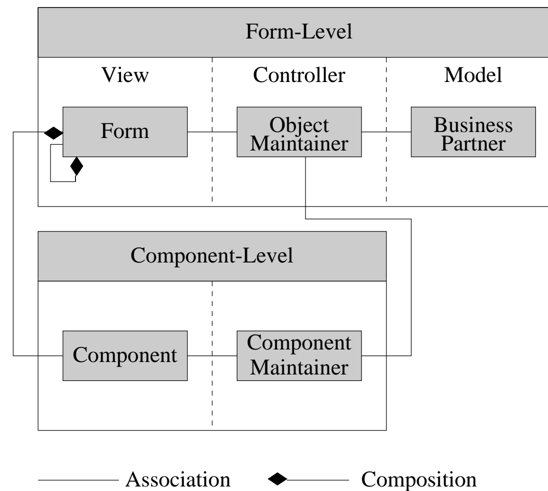


Abbildung 3.2: SanFrancisco Architektur

zieht keine Änderungen des Anwendungskerns nach sich. Die Einführung von Attribut-Metaelementen für Eigenschaften der Clients erlaubt Constraintspezifikationen und ein einheitliches Updateverfahren für die Attribut-Widgets. Damit stellen verschiedene Sichten auf das gleiche Modell immer konsistente Daten dar. Dieser Ansatz ist auch unter dem Namen Visual-Proxy bekannt und wird ebenfalls in [Holub 1999] eingesetzt (siehe Abschnitt 2.1.6).

Das MVC-Client Konzept unterstützt allerdings nur Teilaspekte der Entwicklungsarbeit. So sind die Clients lediglich eine Datensicht auf Businessobjekte und unterstützen nicht die Sicht auf ihr Verhalten. Außerdem bietet das Framework keine Möglichkeiten der Automatisierung bestimmter Aufgaben, so wäre es z. B. sinnvoll, einfache Clients direkt aus dem Businessmodell zu erzeugen. Ohne diese Möglichkeit wird der Entwickler gezwungen, Änderungen von Attributen der Businessobjekte in allen betroffenen Sichten anzupassen.

3.1.2 SanFrancisco

Das SanFrancisco GUI Framework [Tamminga et al. 1999] wurde von IBM entwickelt. Ziel des Systems ist es, die Umsetzung von User Interfaces auch für grosse Anwendungen auf eine solide Grundlage zu stellen.

Basiskonzept der SanFranscisco Architektur ist das MVC-Designpattern (siehe Abschnitt 2.1.2). Da leichte Modifikationen am Kommunikationsmechanismus zwischen den Komponenten vorgenommen wurden, werden die Begriffe Maintainer für Controller und Form für View eingeführt. Aus Abb. 3.2 ist zu ersehen, dass die MVC-Triade zu einer Schichtenarchitektur gemacht wurde, in der Form und Model vollständig entkoppelt sind. Um dies zu erreichen, wurde die direkte Kommunikation zwischen Model und Form durch eine indirekte Kommunikation über den Maintainer ersetzt. Bei genauer Betrachtung lässt sich eine große Ähnlichkeit mit der PAC-Architektur (siehe Abschnitt 2.1.3) feststellen.

Um einen Dialog (UIForm) zu spezifizieren, werden die Komponenten (UI-Components) und ihr Layout festgelegt. Verdrahtet wird der Dialog mit einem

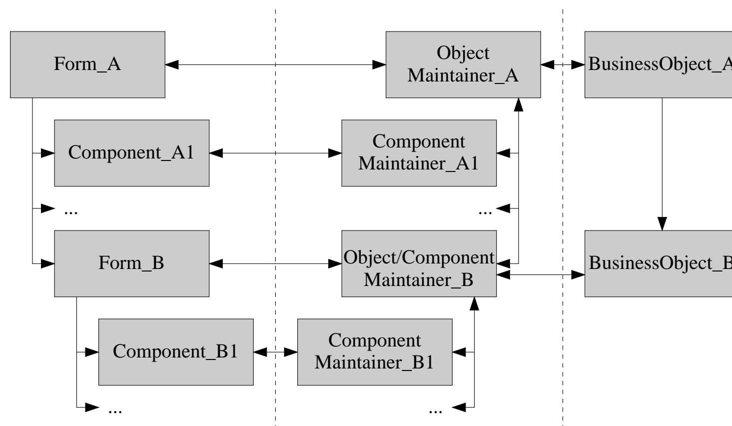


Abbildung 3.3: Beispielhafter Dialogaufbau nach SanFrancisco

Controller (ObjectMaintainer), der über generische Methoden Attribute des assoziierten Objektes (BusinessPartner) zugreifen kann. Den im Dialog enthaltenen Komponenten wird bei Bedarf ein eigener Controller (ComponentMaintainer) zugeordnet, der dafür zuständig ist, Daten vom Widget in Modelrichtung und umgekehrt zu transportieren. Um die Verbindung mit dem Model herzustellen werden diese Subcontroller nicht direkt mit dem Model verknüpft, sondern propagieren ihre Aufrufe an den übergeordneten Controller (ObjectMaintainer). In Abb. 3.3 ist gut zu erkennen, dass die Verbindungen der Controller (Maintainer) die Verbindungen der Dialogelemente nahezu identisch widerspiegeln. Durch die Möglichkeit, auch komplexe Dialoge (Forms) als Komponenten in andere Dialoge einzubinden, können beliebig komplizierte Sichten erstellt und bereits fertige Forms in anderen Kontexten wiederverwendet werden.

Anmerkungen

Die im SanFrancisco Ansatz gewählten Modifikationen des MVC-Patterns wirken sich positiv auf die Übersichtlichkeit der Architektur aus. Die übrigen Konzepte sind dem MVC-Client Ansatz vergleichbar. Da jedoch in SanFrancisco kein Model auf Widgetebene verwendet wird, muss der Zugriff auf das nicht vorhandene Modellelement dieser Ebene umgangen werden. Dies gelingt durch die eingeführte Schichtenarchtitektur, in der stets der Controller auf das Model bezogene Anfragen bearbeitet.

Beachtenswert ist die in SanFrancisco integrierte Internationalisierung des User Interfaces. So kann das Programm durch sprachspezifische Property Dateien ohne Programmieraufwand einfach in andere Sprachen übersetzt werden. Gerade für große Projekte gewinnt dieser Aspekt an Bedeutung. Ein ebenso wichtiger Punkt für die rationelle Realisierung komplexer Projektvorhaben ist die Möglichkeit, Software-Werkzeuge einsetzen zu können. Um das Interface für eine SanFrancisco Applikation zu entwerfen, können bekannte Interface Builder eingesetzt werden, da der daraus erzeugte Code leicht in die SanFrancisco Umgebung eingebettet werden kann. Alle in 10 angesprochenen Einschränkungen gelten auch für den SanFrancisco Ansatz.

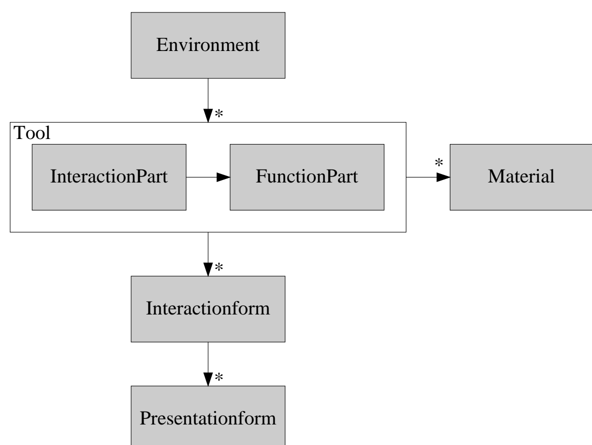


Abbildung 3.4: JWAM GUI Architektur

3.1.3 JWAM

Das JWAM-Framework[4] [Bleek et al. 1999a] wird seit 1998 am Arbeitsbereich Softwaretechnik der Universität Hamburg mit dem Ziel erstellt, die Entwicklung grosser interaktiver Anwendungssysteme zu vereinfachen.

Zur Konstruktion von Anwendungssystemen wird die WAM-Metapher (Werkzeuge, Automaten, Materialien) benutzt. „Werkzeuge, die fachlich die Arbeitsmittel im Anwendungsbereich repräsentieren, kapseln softwaretechnisch die Interaktion mit dem Benutzer“ [Bleek et al. 1999b, S. 1]. Werkzeuge setzen sich aus einer Funktions- und einer Interaktionskomponente zusammen, die konzeptionell verschiedene Verantwortlichkeiten ausdrücken. In der Funktionskomponente findet sich die fachliche Funktionalität, wohingegen die Interaktionskomponente Services der Funktionskomponente nutzt und die Verbindung zur Oberfläche herstellt. Die Arbeitsmittel, auf denen die Werkzeuge operieren, werden als Materialien bezeichnet. Eine genaue Beschreibung des WAM-Ansatzes findet sich in [Züllighoven 1998].

Um zu vermeiden, dass Interaktionskomponenten Widgets direkt benutzen und damit toolkitabhängig werden, wurden Interaktionsformen (IAF) als eine am Umgang orientierte Abstraktion von Toolkit-Elementen entwickelt. Der Abstraktionsansatz geht daher von den Anforderungen der Anwendungen und nicht von der Technologie des Toolkits aus. Kurz gesagt: „Interaktionsformen vergegenständlichen Benutzeraktionen auf einem fachlichen Niveau. Sie orientieren sich am Umgang des Anwenders mit der Benutzungsoberfläche des Programms“ [Bleek et al. 1999b]. Beispiele für Interaktionsformen sind die Auswahl eines Elements (1fromNSelection) und die Eingabe von Daten (FillIn).

Da eine Interaktionsform von verschiedenen Widgets realisiert werden kann, wird ein allgemeiner Kopplungsmechanismus mit den Oberflächenelementen benötigt. Interaktionsformen sind deshalb mit Präsentationsformen verbunden, die eine abstraktere Sicht auf die konkreten Interfaceelemente darstellen. Aus Abb. 3.4 lassen sich die direkten Verbindungen der Komponenten ersehen. Um Kommunikation auch in die andere Richtung zu erlauben, wird ein Benach-

[4] http://www.jwam.de

richtigungsmechanismus auf Basis von Events und Command Patterns (siehe [Gamma et al. 1995]) eingerichtet. Ausführlich beschrieben ist die Infrastruktur in [Lippert 1997].

Anmerkungen

Betrachtet man die WAM-Komponenten aus Abb. 3.4 im Vergleich zu anderen Techniken, lässt sich eine starke strukturelle Ähnlichkeit zu aufgabenorientierter Modellierung feststellen. Setzt man eine Aufgabe mit den Verantwortlichkeiten eines Werkzeuges gleich und wenden das MVC-Designpattern (siehe Abschnitt 2.1.2) auf dieser Detailstufe an, stellt die Funktionskomponente den Werkzeug-Controller und die Interaktionskomponente den View dar. Die Interaktionskomponente wird dabei wie in Arch (siehe Abschnitt 2.1.4) in abstrakte und konkrete Interaktionselemente unterteilt. Da für die Bearbeitung einer Aufgabe kein explizites Task-Objekt zur Verfügung steht, werden verschiedene Materialien als Model benutzt.

Die Einführung von Interaktionsformen führt dazu, dass die Anwendungen toolkitunabhängig entworfen werden können. Für das Screendesign können Standard InterfaceBuilder eingesetzt werden. Der dazu verwendete Mechanismus ist in [Lippert 1997] beschrieben. Auch für das JWAM Framework gelten die in 10 angesprochenen Einschränkungen.

3.1.4 Model-View-Presenter (MVP)

Das Model-View-Presenter (MVP) Programmiermodell [Potel 1996] wurde von der Firma Taligent entwickelt. Es handelt sich dabei um ein Tochterunternehmen von IBM, das unter anderem das Visual-Age[5] Projekt realisiert. MVP basiert auf dem MVC-Designpattern (siehe Abschnitt 2.1.2) und erweitert es um Konzepte zur Umsetzung komplexer Anwendungen.

Taligent führt eine Trennung zwischen Model und View-Controller (Presentation) ein, die eine Untergliederung eines Entwicklungsproblems in zwei fundamentale Konzepte widerspiegelt: Das Datenmanagement und das User-Interface (s. Abb. 3.5). Diese zwei Konzepte stellen grundlegende Designfragen dar, die jeder Entwickler addressieren muß. Das Datenmanagement betrifft nicht nur die zugrundegelegten Datenstrukturen des Models sondern auch ihren Zugriff, Persistenz, Verteilung und gemeinsam benutzte Informationen. User-Interface Aspekte betreffen die Visualisierung von Objekten (Sichten), Dialogmanagement und Feedback des Programms.

Um die vielfältigen Probleme des Daten-Managements besser lösen zu können, hat Taligent drei Kernfragen identifiziert:

1. Was sind meine Daten?
2. Wie wähle ich Teile meiner Daten?
3. Wie ändere ich meine Daten?

Die erste Frage betrifft die Repräsentationsform der Daten und spiegelt so das klassische Model des MVC-Designpatterns wider. Die zweite Frage beantwortet Taligent mit sogenannten Selections. Sie fassen Datenelemente zusammen

[5] Visual Age ist eine integrierte Programmierumgebung (IDE) für Java/C++.

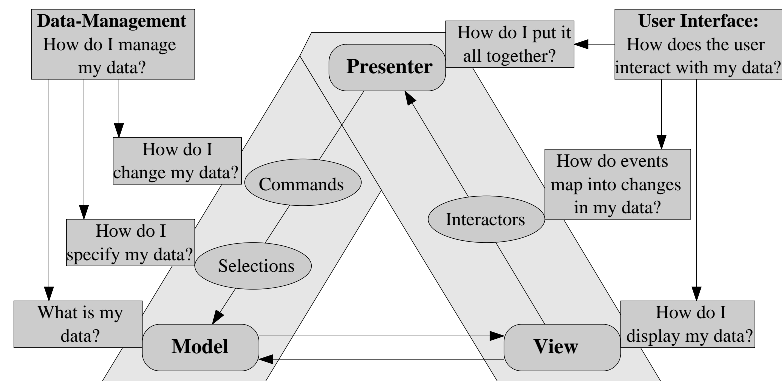


Abbildung 3.5: MVP-Architektur (aus [Potel 1996])

und ermöglichen so die Manipulation von beliebigen Teilmengen der Daten des Models. Datenänderungen (siehe dritte Frage) werden durch sogenannte Commands vorgenommen. Sie stellen Operationen dar, die auf Selections ausgeführt werden können.

In ähnlicher Art und Weise wurden Kernfragen für den User Interface Bereich formuliert:

1. Wie stelle ich meine Daten dar?
2. Wie werden Events in Datenänderungen abgebildet?
3. Wie setzt sich alles zusammen?

Die Darstellung der Daten wird durch die View Komponente des MVC-Patterns realisiert. Um ein allgemeines Konzept für die Ereignisbehandlung innerhalb der Benutzungsschnittstelle zu erreichen, werden sogenannte Interactors eingeführt. Interactors sind Observer (siehe [Gamma et al. 1995]) für Interactor-Events[6] und ermöglichen so eine einheitliche Behandlung der Ereignisse in der Anwendung. Die dritte Frage greift das Zusammenwirken der hier aufgeführten Komponenten auf. Eine als Presenter bezeichnete Komponente repräsentiert die traditionelle main-Methode oder die Event-Loop einer Anwendung. Aufgaben des Presenters bestehen sowohl in der Erstellung der Models, Selections, Commands, Views und Interactors als auch in der Bereitstellung von Businesslogik zur Steuerung. Der Presenter ist vergleichbar mit dem Controller des MVC-Patterns, wirkt aber auf Applikationsebene.

Um client / serververteilte Anwendungen mit dem MVP Modell zu realisieren, muß entschieden werden, welche Elemente der MVP-Abstraktionen sich in Teilen oder komplett auf dem Client bzw. Server befinden (s. Abb. 3.6). Eine herkömmliche Lösung würde den Presenter auf Client und Server verteilen. Das Model, die Selections und Commands stellen typische serverseitige Funktionalitäten dar. Der View und Interactor repräsentieren typische clientseitige Funktionalitäten. Der Presenter wird verteilt über Client und Server realisiert. Mit Hilfe entfernter Kommunikation wirkt er als eine konzeptuelle Einheit. Je

[6]Interactor-Events sind Ereignisse, die ein Benutzer durch Interaktion mit dem Programm auslöst.

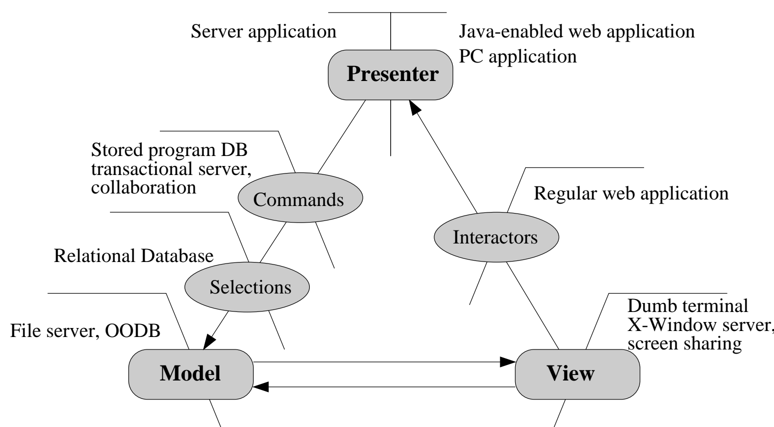


Abbildung 3.6: MVP-Verteilungsmöglichkeiten

nach Art des Protokolls[7] sind sowohl thin als auch fat Presenter-Clients vorstellbar.

Weitere Möglichkeiten der Verteilung von Elementen des MVP Modells korrespondieren in relativ künstlicher Art zu einer großen Anzahl populärer Client / Server Architekturen. Befindet sich nur das Model auf dem Server und alle anderen Komponenten auf dem Client, ähnelt das System einem File-Server oder einer OODB. Befinden sich nun zusätzlich Selections auf dem Server, ist das System mit einer relationalen Datenbank vergleichbar. Werden auch die Commands auf den Server verschoben, erhält man eine stored program DB oder auf anderer Ebene einen Transaction Server. Verfolgt man die Verteilung aus konträrer Richtung, verschiebt also alle Komponenten bis auf den View auf den Server, erhält man Terminal-System. Sofern man zusätzlich die Interactors auf dem Client unterbringt, erhält man eine heute übliche Web-Applikation. Der View ist die HTML-Seite, die in einem Browser dargestellt wird, und der Interactor steuert Mausklicks auf HTML-Links oder Form-basierte Interaktionselemente. Sämtliche Businesslogik und Daten befinden sich auf dem Webserver. Bei einem Java-Applet würden zusätzlich Teile des Presenters auf den Client verschoben.

Anmerkungen

Das MVP-Framework bereichert das MVC-Designpattern um zusätzliche applikationsrelevante Abstraktionsschichten, die dem Entwickler helfen, viele Probleme strukturiert zu lösen. Durch die klare Trennung der einzelnen Komponenten voneinander, wird ein hoher Grad an Flexibilität erreicht. Neben den bekannten Vorteilen, die durch die Adaption des MVC-Patterns erreicht werden (siehe Abschnitt 2.1.2), wird durch die Separation von Commands und Selections eine Wiederverwendbarkeit der Commands im Kontext anderer Anwendungen geschaffen. Durch die Trennung von Interactors und Views wird ein allgemeiner Eingabemechanismus etabliert. Ohne Änderungen der Applikationslogik oder der Sichten können verschiedene Menu-, Dialog-, und Tastaturäquivalente eingesetzt werden.

[7] Z.B. RPC (Remote Procedure Call), SQL (Structured Query Language), CORBA (Common Object Request Brokerage Architecture).

Die Unterteilung der Abstraktionen in präsentations- und datenmanagementrelevante Konzepte wirkt logisch und wurde auf Datenmanagementseite konsequent umgesetzt[8]. Die Unterteilung der User Interface Seite in Presenter, Interactor und View unterstützt den Entwickler nicht wesentlich. Die identifizierten Elemente wirken nicht natürlich und verlagern unangenehme Aufgaben in den Presenter, der eine Main-Methode objektorientiert kapselt.

3.1.5 Vergleich der nicht-deklarativen Systeme

Die Architekturen der vorgestellten Systeme basieren alle auf Entwurfsmustern, die in den meisten Fällen für eine Anwendung auch auf Applikationsebene erweitert wurden. So basieren MVC-Client, SanFrancisco und MVP auf dem MVC-Designpattern und JWAM verwendet den WAM-Ansatz als Grundlage. Darüber hinaus bieten MVP und JWAM Methodologien an, welche die Entwicklung mit dem jeweiligen System in geordnete Bahnen lenken soll. Keines der getesteten Frameworks bindet mehr als eine Interfacemodalität ein. Genauer gesagt beschränken sich die Ansätze darauf, User Interfaces im WIMP-Stil zu unterstützen. Orthogonal zu der Gruppe untersuchter WIMP-Systeme existieren auch einige Frameworks zur Entwicklung von Weboberflächen (HTML) wie z. B. Struts und Turbine (siehe Tablle 3.1). Ein Ansatz, der diese unterschiedlichen Strömungen in einem System zusammenzuführt, ist den Autoren nicht bekannt.

3.1.6 Weitere Forschungsansätze

Weitere Forschungsansätze im Bereich der nicht-deklarativen User Interface Werkzeuge, auf die im Kontext dieser Arbeit nicht näher eingegangen wird, sind in Tabelle 3.1 zusammengefasst.[9]

Name	Verfügbarkeit
FreeHEP	`http://java.freehep.org/`
Galaxy	Visix, eingestellt
GNOME	`http://www.gnome.org/`
JX	`http://www.newplanetsoftware.com/jx/`
StarView	StarDivision, eingestellt
Struts	`http://jakarta.apache.org/struts`
Turbine	`http://jakarta.apache.org/turbine`
Qcis	`http://www.cs.queensu.ca/~dalamb/qcis/`
Qt	`http://www.trolltech.com/`
wxWindows	`http://www.wxwindows.org/`
XVT	`http://www.xvt.com/`
zApp	Roguewave, eingestellt
ZINC	`http://www.windriver.com/zinc/`

Tabelle 3.1: Weitere nicht-deklarative Systeme

[8]Einzig etwas stutzig macht die Einführung einer “highlightSelection()” Methode in der Selection, die nicht zur Interfaceebene gehört.

[9]Gute Übersichten finden sich auf `http://www.cs.cmu.edu/afs/cs/user/bam/www/toolnames.html` und auf `http://www.newplanetsoftware.com/jx/compare.php`.

3.2 Modellbasierte Ansätze

In diesem Abschnitt wird eine Auswahl der modernen modellbasierten Forschungsansätze vorgestellt. Die Ansätze sind dabei nicht ohne weiteres miteinander vergleichbar, da sie zum Teil sehr unterschiedliche Aspekte der User Interface Entwicklung betonen. Sie unterscheiden sich insbesondere in der Modell- und Notationsauswahl, dem Grad der Automatisierung, der Methodologie, der industriellen Reife, der Nähe zur Softwaretechnik, dem Toolsupport und der Ausführungsart.

3.2.1 Die historische Evolution modellbasierter Systeme

Die Evolution modellbasierter User Interface Entwicklung lief parallel zur Entwicklung der User Interface Modelle selbst. Begonnen hat die modellbasierte Entwicklung mit der Verwendung von nur abstrakt vorhandenen Modellen und einfachen Applikationsdatenmodellen. Heutzutage werden zahlreiche Submodelle für die Beschreibung von Teilaspekten eines Systems eingesetzt. Während der gleichen Zeit konnte im Gebiet der User Interfaces eine Tendenz zu immer komplexeren Interfacegrundbausteinen beobachtet werden. Von elementaren Toolkitelementen wie z. B. Scrollbars über einfache textuelle oder grafische Editoren bis zu fertig vorbereiteten Elementen, wie z. B. ActiveX Controls, die dafür genutzt werden können, Interfaces portionsweise zusammenzusetzen. Die modellbasierte Entwicklung ist angewiesen auf derart vielfältige Interface Modelle, da sonst nicht zur Verfügung stehende Widgets durch Modellierung auf sehr niedriger Detailebene nachgebildet werden müssen. Das Zusammenwirken dieser beiden Entwicklungen gibt der modellbasierten Interfaceentwicklung eine solide Grundlage.

Frühe Interfacemodelle waren abstrakt und wurden eingesetzt, um die Oberflächenkomponenten und ihre Funktionen zu visualisieren. Von wenigen Ausnahmen abgesehen hatten diese Modelle keine computererfasste Entsprechung, sondern waren dafür gedacht, daß Oberflächendesign zu dokumentieren. Ein System, das auf einem abstrakten Interfacemodell basiert, ist L-CID [Puerta 1990].

Das erste explizite Modell, das für die modellbasierte Oberflächenerzeugung eingesetzt wurde, war das Datenmodell. Dieses Modell basiert direkt auf den in einer Applikation vorkommenden Datenstrukturen. Es hat sich als nützlich erwiesen, die Widgettypen anhand der zugrundeliegenden Datentypen zu bestimmen. Systeme, die diesen Ansatz verfolgen, sind z. B. UIDE [Foley et al. 1991] und Don [Kim Foley 1990]. Zusätzlich zum Widgettypemapping muß nur noch ein Layoutalgorithmus angewandt werden, um eine komplette statische Interfacebeschreibung zu erzeugen.

Fortschritte im Bereich der Softwaretechnik ließen andere Domänenmodelle in den Vordergrund rücken. Es wurden Systeme entworfen, die auf Entity-Relationshipdiagramme (Genius [Janssen et al. 1993]), erweiterte Datenmodelle (TRIDENT) oder objekt-orientierte Datenmodelle (FUSE, Janus, Mobi-D) zurückgriffen. Damit war es möglich, vollständige deklarative Domänenmodelle zu erstellen, die neben den Daten auch Verhalten und Beziehungen der Objekte untereinander auszudrücken vermochten. So war es möglich, zusätzlich zum statischen Layout, partielle Spezifikationen des dynamischen Verhaltens eines Interfaces zu erzeugen.

Um auch semantische Funktionen der Anwendung selbst beschreiben zu können, wurden verschiedene Formen von Applikationsmodellen eingeführt. Dadurch konnte nun auch Verhalten spezifiziert werden, das zwischen verschiedenen Domänenobjekten stattfindet und nicht dem natürlichen Verhalten eines der beteiligten Objekte zugeordnet werden kann. Systeme, die Applikationsmodelle einsetzen sind z. B. UIDE, TRIDENT und HUMANOID [Szekely 1992a].

Applikations- und Domänenmodelle werden heute als Teile eines umfassenderen Modellspektrums zur Beschreibung eines User-Interfaces gesehen, da sich mit ihnen nicht alle relevanten Aspekte des Interfacedesigns ausdrücken lassen. Im Laufe der Zeit entstanden weitere partielle Modelle, wie Benutzer-, Aufgaben-, Dialog- und Präsentationsmodelle. Besonders beachtenswert ist die Einbindung von Aufgabenmodellen in den Entwicklungszyklus, da mit ihnen eine benutzerzentrierte Designphilosophie in den Vordergrund rückt. Diese Modelle sind geeignet, Aufgaben zu beschreiben, die ein Benutzer des Systems ausführen will und für die entsprechende Interaktionskapazitäten geschaffen werden müssen. Die Verwendung von Taskmodellen führt zu einer aufgabenorientierten Designmethodologie, die es mit einfachen Mitteln erlaubt, alternative Designlösungen für eine Aufgabe zu erstellen und zu vergleichen. ADEPT [Markopoulos et al. 1992], FUSE, TADEUS und TRIDENT binden verschiedene Arten von Taskmodellen ein.

Ein Dialogmodell wird verwendet, um die Mensch-Maschine-Kommunikation zu steuern (siehe Abschnitt 2.2.3). Dazu gehört die feine Dialogsteuerung, welche die Ausführbarkeit und Anwählbarkeit bestimmter Funktionen regelt, und die grobe Dialogsteuerung, die den Ablauf von Dialogen bestimmt. Viele Dialogmodelle wurden erfolgreich auf verschiedene Weisen in Systeme integriert. Allerdings hat sich keine der Lösungen bisher als Standardverfahren durchgesetzt. Genius und TRIDENT verwenden Dialognetze, BOSS kontextfreie Grammatiken, HUMANOID Dialogtemplates und TRIDENT Zustandsübergangsdiagramme.

Das Präsentationsmodell bestimmt die Auswahl und Anordnung der Interaktionselemente innerhalb der Dialoge (siehe Abschnitt 2.2.2). Eine hierarchische Dekomposition der Sichten ermöglicht die Zuordnung der Widgets zu einzelnen Gruppen. Grundsätzlich sind Präsentations- und Dialogmodell eng verwandt, da sie nur gemeinsam das Aussehen und Verhalten der Anwendung ausreichend abbilden. Aus diesem Grund werden sie in modellbasierten Entwicklungsumgebungen häufig zusammen betrachtet. So bietet z. B. das ITS-System [Wiecha et al. 1990] eine Style-Bibliothek. Der ausgewählte Style beinhaltet Entscheidungen, die sowohl die Präsentation als auch die Dialogsteuerung betreffen und helfen, durch einheitliche Optik und Verhalten konsistente Applikationen zu entwickeln.

Andere in der Praxis selten verwendete partielle Modelle sind Plattform-, Benutzer-, und Arbeitsplatzmodelle. Sie werden dazu benutzt, Eigenschaften des jeweiligen Sachgegenstandes zu spezifizieren. Obwohl sie nicht unerheblichen Einfluss auf das User-Interface haben, wurde kaum Softwareunterstützung entwickelt. In der Praxis werden ihre Charakteristika im Präsentations- und Dialogmodell untergebracht. Um die Vielzahl verschiedener Submodelle zu vereinen, gibt es Ansätze, gesamtheitlich-integrierte Interfacemodelle zu definieren, z. B. Mobi-D, MASTERMIND. Neben der Ausdifferenzierung in verschiedene Teilmodelle unterstützen die neueren Ansätze zunehmend mehr Designphasen durch geeignete Softwaretools, z. B. MASTERMIND, TRIDENT, TADEUS, Mobi-D.

3.2.2 Janus/Jade

Ziel des Janus/Jade[10] Projektes [Balzert et al. 1995] der Ruhr Universität-Bochum ist eine effektive Entwicklung von betrieblichen Anwendungen mit grafischer Benutzungsoberfläche. Aus dem Analysemodell eines Standardverfahrens soll möglichst automatisiert[11] eine vollständige Applikationsumgebung generiert werden.

Der Janus Prozeß zur Anwendungsentwicklung (s. Abb. 3.7) sieht als ersten Schritt die Erstellung eines geeigneten Fachkonzepts für die Anwendungsdomäne vor. Das Fachkonzept wird anschließend durch Systemanalytiker in ein objektorientiertes Analysemodell umgesetzt, das als Grundlage sämtlicher Generierungsprozesse dient. Das System erzeugt neben der Benutzungsschnittstelle auch das Codegerüst für den Anwendungskern, die Datenhaltung, das Hilfesystem, die Client / Serververteilung und weitere optionale Dienstleistungen, wie z. B. Mehrbenutzerverwaltung und Mandantenverwaltung. Um eine ablauffähige Applikation zu erhalten, reicht eine Generierung der Einzelkomponenten nicht aus, sondern es ist notwendig, ihre Wechselwirkungen mit dem Systemkern in Betracht zu ziehen. Janus erzeugt daher auch die Anbindung der Komponenten an den Kern. Zur Zeit noch nicht generierbar ist die Implementierung der Semantik des Fachkonzeptes. CASE Tools (siehe Abschnitt 2.3.7) ermöglichen zwar, für Klassendiagramme Coderahmen zu erzeugen, bieten aber noch keine Möglichkeiten, spezifiziertes Verhalten in Code zu übertragen. Denkbar wäre es, daß UML-Tools in der Zukunft Diagrammtypen, wie z. B. Zustands- oder Aktivitätsdiagramme auch zur Modellierung von Methoden anbieten werden.

Basis für den Generierungsprozess des Janus-Systems bildet immer ein statisches OOA Modell (s. Abb. 3.8), d. h. ein Klassendiagramm. Dieses ist eine konkrete Ausprägung eines zugrunde gelegten Metamodells, welches fachliche, datenbankabhängige und oberflächenrelevante Eigenschaften einer Anwendung auszudrücken vermag. Es handelt sich dabei um eine Eigenentwicklung, die Konzepte aus den Objektmodellen der OMG [OMG 2000a] und der ODMG [Cattell 1997] integriert. Einzelheiten zum verwendeten Metamodell finden sich in [Balzert et al. 1995]. Abgelegt werden OOA Modelle in der Beschreibungssprache Janus-Definition-Language (JDL).

Die vom Janus-System erzeugte Software entspricht einer Drei-Schichten-Architektur mit strenger Trennung. Dazu gehört eine Schicht für die Benutzungsoberfläche, eine für den Anwendungskern und eine für die Datenhaltung. Durch die strikte Trennung der Schichten bleiben diese austauschbar und die Anwendung flexibel.

Der Generatorblock des Systems besteht aus zwei getrennten Einheiten, dem GUI- und dem App-Generator. Der App-Generator hat die Aufgaben, den Coderahmen für den Anwendungskern, die Ansteuerung der Datenbank und optional eine Client / Serververteilung zu erzeugen. Zusätzlich zu den üblichen Lese- und Manipulationsmethoden für Attribute werden auch Methoden zum typfreien Zugriff auf sämtliche Eigenschaften des Objektes generiert. Die Kommunikation von Oberflächenelementen mit dem Fachkern läuft ausschliesslich über diesen

[10] Da in der herangezogenen Literatur keine explizite Unterscheidung des Janus- und Jade-Ansatzes vorgenommen wird, werden fortan Janus und Jade synonym verwendet. Die Autoren mutmaßen, daß das Janussystem zunächst nur für die Generierung von Benutzungsoberflächen vorgesehen war, und der Jade-Ansatz die Generierung weiterer Aspekte eingeführt hat.

[11] Automatisiert wird im Sinne von automatisch mit Einflußmöglichkeiten benutzt.

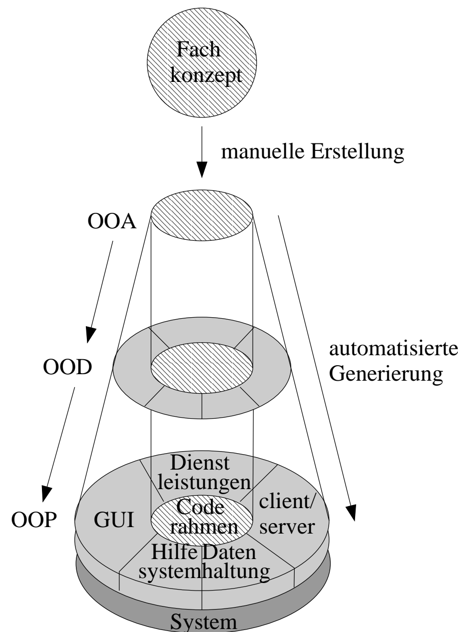


Abbildung 3.7: Das Janus Entwicklungskonzept (aus [Balzert et al. 1995])

Mechanismus. Im OOA Modell definierte Constraints, wie z. B. Wertebereiche und Abhängigkeiten, werden vor einem schreibenden Zugriff auf das Attribut überprüft.

Der GUI-Generator sorgt sowohl für die Gestaltung als auch für die Anbindung der Oberfläche an die Fachlogik. Die Gestaltung der Benutzungsoberfläche muß vor allem das Mapping-Problem (siehe auch Abschnitt 4.3.1) von Domänenentitäten auf konkrete Interaktionselemente lösen [Puerta Eisenstein 1999]. Der Generator bedient sich zu diesem Zweck einer GUI-Knowledgebase, die Regeln für die Abbildung festlegt. Einzelheiten der Transformationsregeln werden in [Balzert 1995] ausführlich beschrieben. Konkret werden Informationen des Metamodells über Attribute herangezogen, um seinen Widgettyp festzulegen. Eine Dialogidentifikation wird über Klassendefinitionen und Vererbungshierarchien realisiert. Da durch solch relativ einfache Gesetzmäßigkeiten nicht eine komplette Anwendung zufriedenstellend abgebildet werden kann, wurden manuelle Eingriffsmöglichkeiten sowohl auf globaler Ebene als auch zur Steuerung von Einzelfällen vorgesehen. Der GUI-Generator erzeugt als Ergebnis des Abbildungsprozesses Einheiten des Janus-Application-Frameworks (JAF), die in Windows-Ressource-Files (siehe 2.2.2.2) abgelegt werden. Diese sind als Erweiterungen simpler Interaktionselemente einer GUI-Klassenbibliothek realisiert. Wie aus Abbildung 3.9 ersichtlich, stellen die Elemente des JAFs eine direkte Verbindung zu bestimmten Elementen des Metamodells her. Neben einfachen Elementen zur Abbildung von Attributen (diverse UIControls) existieren auch Einheiten für Gruppierungszwecke (UIContainer) und zur Abbildung von Operationen (UIPushbutton). Genaue Erläuerungen zur Konzeption des JAF und seiner Elemente finden sich in [Balzert et al. 1995].

Neben der Auswahl von Interaktionselementen spielt auch ihre Positionie-

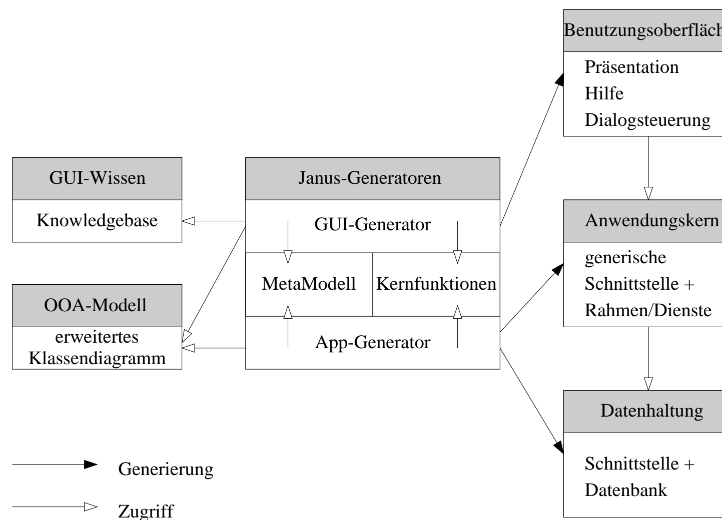


Abbildung 3.8: Generierungsprozeß im Janus-System (aus [Kruschinski 1999])

rung innerhalb der Fenster eine wichtige Rolle. Dazu wurde eine Layoutkomponente als unterstützendes System für die Generierung entwickelt. Ausführlich beschrieben wird sie in [Kruschinski 1999]. Weiterhin hat der GUI-Generator die Problematiken der Anbindung von GUI-Elementen an das Analysemodell zu adressieren. So kann sich die Repräsentation eines Wertes in einem Schnittstellenelement von der Repräsentation des Wertes in einem Attribut unterscheiden. Durch die Verwendung der typfreien Zugriffsmethoden auf Attribute wurde das Austauschformat auf Zeichenketten festgelegt. Daneben muß vor der Weitergabe von Daten aus dem GUI ihre Gültigkeit überprüft werden. Dies wird durch die Integration von Restriktionen aus dem OOA Modell erreicht. Außerdem muß es möglich sein, Verbindungen zwischen einzelnen Oberflächenelementen herzustellen, um Abhängigkeiten ausdrücken zu können. Janus löst das Problem durch Eventumleitungen. Genauere Ausführungen zu den Lösungsansätzen findet man in [Balzert et al. 1995].

Ein Generatorkern stellt Funktionen und Daten bereit, die von beiden Generatoren gemeinsam benutzt werden. Hauptsächlich handelt es sich dabei um Methoden zur Codegenerierung, die auf einer abstrakten Zwischensprache basieren. Dadurch geraten die Generatoren nicht in direkten Kontakt mit der Zielsprache und ihre Umstellung führt nur zu Anpassungsmaßnahmen an den Kernfunktionen.

Ergebnis aller Generierungsprozesse ist Quellcode einer spezifischen Programmiersprache (hier C++). Das Compilieren und Binden dieses Codes führt zu einer kompletten Anwendung. Den Programmierern obliegt es nun nur noch, die fachlichen Teile der Anwendung in den Coderahmen hineinzuimplementieren.

Anmerkungen

Der Janus-Ansatz rückt die Wandlung des Softwareerstellungsprozesses von Programmiertätigkeiten hin zur automatisierten Softwaregenerierung in den Mittel-

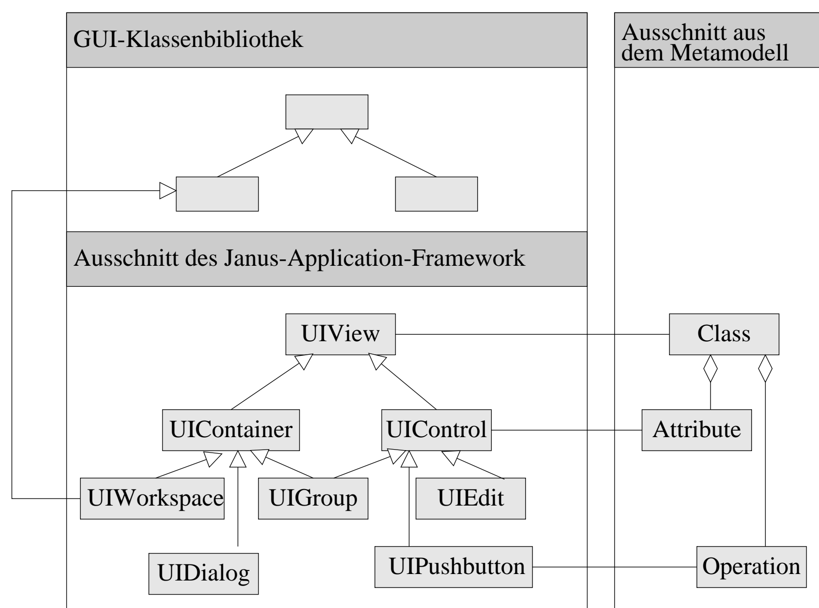


Abbildung 3.9: Das Janus-Application-Framework (aus [Balzert et al. 1995])

punkt des Interesses. Wie in Abbildung 3.7 gezeigt, bestünde die Hauptaufgabe der Anwendungsprogrammierer dann nur noch darin, den funktional relevanten Teil einer Anwendung umzusetzen. Die Vision einer in grossen Teilen automatisierten Anwendungsentwicklung lässt sich aber schon für den Interface Bereich ohne qualitative Einschränkungen nicht erreichen. In [Szekely 1996] wird detailliert dargestellt, warum die Informationen eines Domänenmodells nicht ausreichen, ein User Interface vollständig zu generieren.

Janus verwendet zur Interfacebeschreibung lediglich ein Domänenmodell. Der Verzicht auf andere Modelle führt zu weiteren Einschränkungen in der Anwendbarkeit des Systems. Weder Aufgaben- noch Dialogsequenzen können spezifiziert werden. Damit kann im Janus-Paradigma zwar objektorientiert, aber nicht mehr benutzerzentriert modelliert werden, da die Aufgabenabläufe der Benutzer nicht angemessen nachgebildet werden können. Das Fehlen eines abstrakten Präsentationsmodells führt dazu, dass manuelle Anpassungen am konkreten Modell (den Windows-Ressource-Files) vorgenommen werden müssen. Inwieweit diese Änderungen bei weiteren Generierungsdurchläufen berücksichtigt werden, wird in der Literatur nicht erwähnt (*Postediting Problem*, vergl. Abschnitt 2.3.6).

Vorbildlich wurde im Janus-System die Brücke zur Softwaretechnik geschlagen. Die Erweiterung bekannter CASE Tools ermöglicht nicht vorgebildeten Entwicklern einen problemlosen Einstieg und senkt die Hemmschwelle, sich mit dem System auseinanderzusetzen.

3.2.3 Mobi-D

Mobi-D (Model-based interface development) [Puerta 1997, Puerta 1998] ist ein Projekt der Stanford University, welches das Ziel verfolgt, ein stark interak-

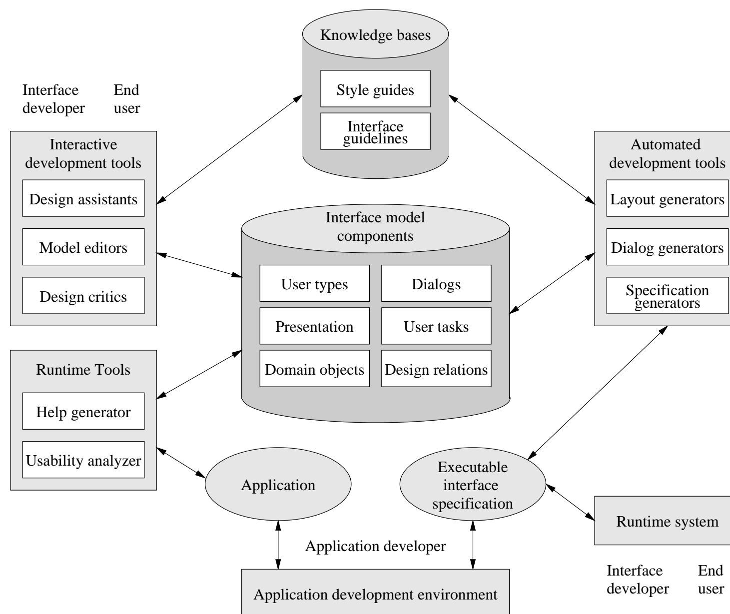


Abbildung 3.10: Mobi-D Architektur (aus [Puerta 1997])

tives, modellbasiertes und domänenunabhängiges Softwareentwicklungssystem bereitzustellen. Es ist Nachfolger des Mecano-Projektes [Puerta 1996], einem modellbasierten System, das automatisch formularbasierte Oberflächen aus Informationen eines Domänenmodells erzeugen kann. Das Mobi-D Projekt wird von der Defense Advanced Research Projects Agency (DARPA) unterstützt und wurde bereits zur Entwicklung von Applikationen im medizinischen und militärischen Sektor eingesetzt.

Die Elemente der Mobi-D Entwicklungsumgebung sind in Abb. 3.10 dargestellt. Das Interfacemodell ist die primäre Wissensdatenbank für das Interfacedesign. Es enthält eine deklarative Repräsentation aller für eine Oberfläche wichtigen Aspekte in Form von in Beziehung stehenden Submodellen. Zu diesen gehört das User Modell, welches die verschiedenen Rollen der Benutzer identifiziert und ihre relevanten Eigenschaften erfasst. Es ist nicht dazu gedacht, den mentalen Zustand eines Benutzers zu einem bestimmten Zeitpunkt während einer Interaktion widerzuspiegeln. Das User-Task Modell enthält alle Aufgaben, die ein Benutzer durch Interaktion mit dem System ausführen kann. Zu einem Task gehört ein Ziel, das durch seine Ausführung erreicht werden soll. Der Task kann sich aus mehreren Subtasks zusammensetzen, die in einer bestimmten Reihenfolge abgearbeitet werden. Zusätzlich können Bedingungen angegeben werden, die zur erfolgreichen Beendigung erfüllt sein müssen. Ein zentraler Bestandteil ist das Domänenmodell, das die Objekte, ihre Eigenschaften, Funktionen und Relationen untereinander enthält. Das Präsentationsmodell enthält die statischen Eigenschaften einer Oberfläche, wie z. B. das Layout, die Interaktionselemente, Fonts und Farben. Daneben gibt es das Dialogmodell, welches für die feine und grobe Dialogsteuerung (vergl. Abschnitt 2.2.3) verantwortlich ist. Das Designmodell enthält Relationen der Elemente anderer Modelle untereinander. Diese Sammlung von Abbildungen wird in Mobi-D Interfacedesign genannt

und ermöglicht dem Entwickler, verschiedene Ansichten nach unterschiedlichen Kriterien wie z. B. dem Usertyp festzulegen.

Entwickler greifen auf die Modelle durch spezielle Software zu. Es werden Designtime-, Runtime-Tools und Runtimesysteme unterschieden. Designtime-Tools arbeiten auf einem Interfacemodell, um ein Interfacedesign zu erzeugen. Zu dieser Klasse von Werkzeugen gehören sowohl interaktive Tools wie Modelleditoren und Designassistenten als auch automatisch arbeitende Tools, z. B. Layoutgeneratoren. Runtime-Tools benutzen ein Interfacemodell zur Laufzeit, um Endnutzeraktivitäten zu unterstützen, z. B. die automatische Erzeugung einer Hilfeoption oder die Analyse von Laufzeitdaten für ein Benutzbarkeitsprofil. Zum Teil greifen die Werkzeuge dazu neben dem Interfacemodell auch auf weitere Wissensdatenbanken zurück. Ein Runtime-System arbeitet auf einer ausführbaren Interfacespezifikation und erlaubt, ein Interfacemodell frühzeitig zu testen.

Der Entwicklungszyklus von Mobi-D ist in Abb. 3.11 skizziert. Alle Prozesse sind interaktiv und können den Endnutzer einbeziehen. Der Zyklus selbst ist iterativ aufgebaut: Er beginnt mit der Ausarbeitung der Usertasks, die eine enge Zusammenarbeit von Entwicklern und Domänenexperten erfordert. Ein Taskelicitation-Tool[12] fördert den Übergang von einer informalen Beschreibung der Aufgaben hin zu einem ausmodellierten Taskmodell. Dazu trägt der Anwender eine Beschreibung der Aufgaben in textueller Form ein und wählt dann aufgabenrelevante Aktionen und Objekte aus. In Zusammenarbeit mit dem Entwickler wird durch Strukturierung und Formalisierung dieser Daten der Aufgabenumfang festgelegt. Die Ausarbeitung der Tasks hat zwei Gründe. Sie vermindert die Gefahr von Mißverständnissen und nicht korrekten Anforderungen durch die unterstützte Kommunikation zwischen Anwender und Entwickler, und sie bietet dem Entwickler eine Grundlage für die Usertask- und Domänenmodellierung.

In der nächsten Phase hat der Entwickler die Aufgabe, aus der skizzenhaften Aufgabenbeschreibung ein Task- und Domänenmodell zu entwerfen. Er bedient sich dazu Modelleditoren, die es ihm erlauben, die einzelnen Modelle zu verfeinern und zu integrieren. Unter Integration wird hier die Zuordnung von Domänenentitäten zu einzelnen Aufgaben verstanden. Die so erarbeiteten Task- und Domänenmodelle bilden die Grundlage des User-Interface Designs.

Im nachfolgenden Schritt werden die Präsentation und das Dialogdesign der Applikation festgelegt. Sogenannte Decision-Support-Tools analysieren die erstellten Modelle und geleiten den Designer schrittweise durch jeden für einen Subtask erzeugten Dialog. Sie unterbreiten Vorschläge für die einzusetzenden Widgets, die der Designer auf Wunsch ablehnen und durch selbst ausgewählte Elemente ersetzen kann. Durch die klare Verbindung zwischen Oberflächendesign und User- bzw. Domänenmodell, kann während dieser Phase leichter die Meinung des Anwenders einbezogen werden. Der Anwender erhält nämlich ein besseres Verständnis für die Rolle jedes einzelnen Widgets durch den Bezug zum abgebildeten Task. Ist das Interfacedesign abgeschlossen, kann der Anwender die Oberfläche testen und Feedback geben.

[12] Elicitation ist im Sinne von Ausarbeitung gemeint.

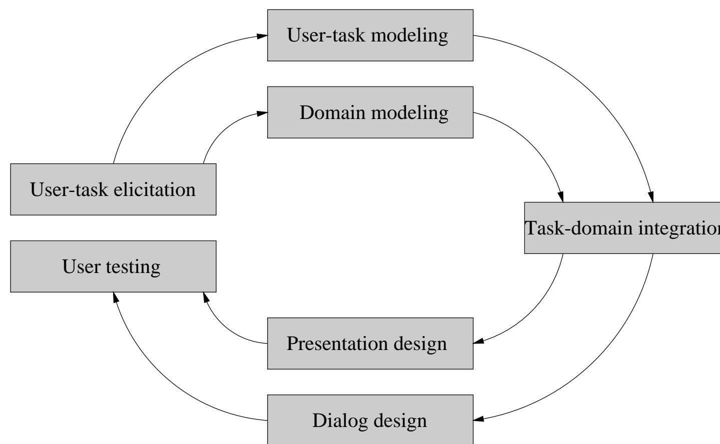


Abbildung 3.11: Mobi-D Entwicklungszyklus (aus [Puerta 1997])

Anmerkungen

Das Mobi-D System ist eines der neuesten und fortgeschrittensten modellbasierten Systeme. Es unterscheidet sich in einer Reihe von Punkten von den anderen Ansätzen. Im Mobi-D Paradigma können eine große Anzahl von deklarativen in Beziehung stehenden Modellen die formale Repräsentation eines User Interface Designs beeinflussen. Weiterhin führt Mobi-D zur Beschreibung der einzelnen Modelle erstmals eine einheitliche Modellierungssprache[13] und für die Relationen der Elemente verschiedener Submodelle ein eigenes Modell (das Designmodell) ein.

Mobi-D hat den Anspruch, alle Phasen des User-Interfaceentwicklungsprozesses durch Werkzeuge zu unterstützen. Diese Werkzeuge haben weniger automatisierenden als entscheidungsunterstützenden Charakter. Das System hat also nicht die Aufgabe, wie z. B. bei Janus, möglichst automatisch das Interface zu generieren, sondern die Auswahlmöglichkeiten auf sinnvolle Alternativen zu beschränken. Mobi-D propagiert damit eine leicht abgewandelte Designphilosophie, welche die Einschränkungen und Probleme vollständig automatischer Ansätze (siehe dazu [Puerta Eisenstein 1999]) vermeidet. Ein weiterer Eckpfeiler der Architektur ist die Fokussierung auf benutzerzentrierte User Interface Entwicklung. Dies wird sowohl im Design der Werkzeuge als auch in der extensiven Nutzung des Taskmodells reflektiert.

3.2.4 FUSE

Das FUSE System (Formal User interface Specification Environment) ist ein Projekt der Technischen Universität München (TUM) [Lonczewski et al. 1996]. Es hat das Ziel, eine Methodologie und einen Satz integrierte Werkzeuge für die automatische Generierung von grafischen Benutzungsoberflächen bereitzustellen. Des weiteren besteht der Anspruch, Tool-Support für alle Entwicklungsphasen des zugrunde gelegten Entwicklungspozesses anzubieten. Ferner soll es

[13] MIMIC (Mecano interface modeling language) aus dem Mecano Projekt resultierend.

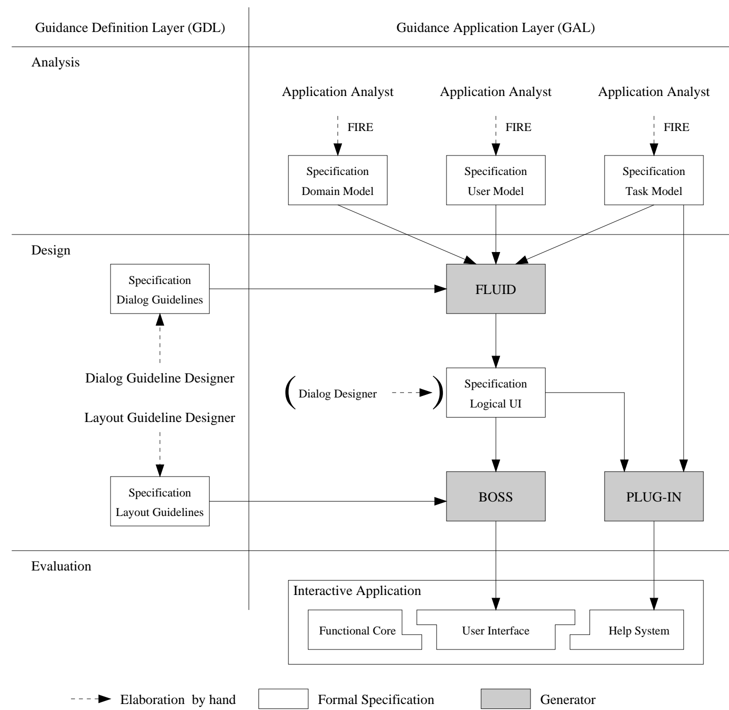


Abbildung 3.12: FUSE-Systemarchitektur

Rapid-Prototyping und die Generierung einer mächtigen Hilfekomponente unterstützen.

Die Architektur des FUSE-Systems ist in Abb. 3.12 dargestellt. Grundlage für die Erzeugung des User-Interfaces ist die Spezifikation von drei Modellen. Im Problem Domain Model werden die für die Benutzungsoberfläche relevanten Objekte und Funktionen des Anwendungskerns beschrieben. Das User-Model dient zur Festlegung der statischen und dynamischen Eigenschaften von individuellen Benutzern und Benutzergruppen. Unter statischen Eigenschaften werden z. B. der Wissensstand eines Users bezüglich der Anwendung und ihrer Aufgaben verstanden. Diese Eigenschaften werden für Benutzer bzw. Benutzergruppen einmalig festgelegt. Beispiel für eine dynamische Eigenschaft ist die Häufigkeit, mit der ein Benutzer bereits erfolgreich bestimmte Aufgaben ausgeführt hat. Dynamische Eigenschaften können für die Adaption der Hilfekomponente sehr hilfreich sein. Das dritte Modell ist das Taskmodell, in dem für das User Interface relevante Aufgaben angegeben werden. Sämtliche Modelle werden in einer neu definierten formalen algebraischen Spezifikation definiert. Genaue Ausführungen dazu finden sich in [Lonczewski et al. 1996].

Der Entwicklungsprozeß eines User-Interfaces im FUSE-System besteht aus der Anforderungsanalyse-, der Design- und der Evaluationsphase. Eine Implementationphase wird nicht benötigt, da das System ablauffähige User Interfaces aus design-level Spezifikationen erzeugt. In der Analysephase werden die drei Modelle definiert. Um diese Aufgabe zu vereinfachen, wurde ein Softwarewerkzeug zur Unterstützung entwickelt. Mit FIRE (Formal Interface Requirements Engineering) lassen sich alle Modelltypen grafisch editieren und ein früher In-

terface Prototyp erzeugen. In der Designphase verwendet das Werkzeug FLUID (FormaL User Interface Development) [Bauer 1996] die Informationen der drei Modelle und zusätzlich Dialogrichtlinien zur Erzeugung einer formalen Spezifikation des logischen User Interfaces. Mit Hilfe des BOSS (BedienOberflächen-SpezifikationsSystem) [Schreiber 1994] wird die abstrakte in eine konkrete User Interface Beschreibung transformiert. Dazu verwendet das Tool neben der abstrakten Beschreibung des User Interfaces auch Layoutrichtlinien. Im Anschluss kann das konkrete Modell mit dem BOSS Werkzeug manuell nachbearbeitet werden. Als Spezifikationstechnik verwendet BOSS Hierarchic Interaction Graph Templates (HIT), welche in [Schreiber 1994] beschrieben sind. Ein in die Applikation integriertes Hilfesystem kann durch die PLUG-IN (PLan-based User Guidance for Intelligent Navigation) Komponente generiert werden. PLUG-IN benötigt die abstrakte Interfacebeschreibung und Taskmodellinformationen als Eingabe.

Anmerkungen

Das FUSE-System bietet Tool-Support für alle Phasen des Interfaceerstellungsprozesses und erlaubt die Generierung funktionsfähiger Prototypen in frühen Entwicklungsphasen. Weiterhin wird die Standardisierung von User-Interfaces durch formale Spezifikation der Dialog- und Layoutrichtlinien unterstützt. Besonders herausragend im Gegensatz zu den anderen Systemen ist die ausgefeilte Hilfegenerierungskomponente PLUG-IN. Im Ergebnis steht dem Benutzer eine intelligente, taskbasierte Onlinehilfe zur Verfügung, die neben obligatorischen Hilfebeschreibungen den aktuellen Zustand des Programms nutzt, um Informationen über die Menge möglicher Aktionen, die zu jeder Aktion gehörenden User-Interaktionen und sogar Animationssequenzen zum erfolgreichen Beenden eines Tasks einblendet.

Kritisch gesehen werden muß allerdings die Verwendung neu definierter formaler Spezifikationsgrundlagen, da damit die Lern- und Hemmschwelle für die Nutzung des Systems stark ansteigt. So wird z. B. in [Schlungbaum Elwert 1995] die Wichtigkeit der Verwendung von bekannten Softwareengineering-Standards für die Spezifikation von Modellen betont, da dadurch Anforderungsbeschreibungen aus frühen Applikationsentwicklungsphasen für die User Interface Entwicklung weiterverwendet werden können.

3.2.5 TRIDENT

Das TRIDENT (Tools foR an Interactive Development EnvironmeNT) Projekt [Bodart et al. 1993, Bodart et al. 1994, Bodart et al. 1996] der belgischen Fakultät Notre-Dame de la Paix a Namur (FUNDP) wurde 1990 mit dem Ziel ins Leben gerufen, eine Designmethodologie für die Erstellung von Human-Machine-Interfaces (HMI) bei Konzentration auf stark-interaktive betriebswirtschaftliche Anwendungen zu entwickeln. Die TRIDENT Methodologie umfaßt neben einer in genaue Aktivitäten eingeteilten Vorgehensweise auch Modelle, auf denen die Prozesse gründen und interaktive Tools zur Unterstützung bei einigen Arbeitsschritten.

Wie in Abb. 3.13 gezeigt, beinhaltet das TRIDENT Architekturmodell einen semantischen Anwendungskern und ein Dialogmodell. Das Dialogmodell wird in einen Conversation- und einen Presentation-Teil untergliedert, wobei die Con-

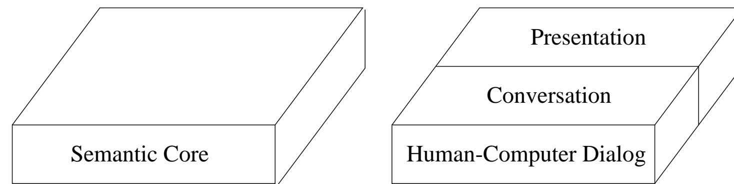


Abbildung 3.13: TRIDENT-Komponenten (aus [Bodart et al. 1993])

versation-Komponente für die Behandlung von high- und mid-level Interaktionen zwischen der Applikation und dem User und die Präsentationskomponente für Darstellung bzw. Eingabe von Applikationsdaten in geeigneter Weise verantwortlich ist. Diese Autonomie führt zur Trennung von dynamischem Dialogverhalten und statischem Dialogerscheinungsbild. Als Metamodell des semantischen Anwendungskerns wird ein erweitertes Attribute-Entity-Relationship Modell (ERA) benutzt, das zur Beschreibung die Dynamic Specification Language (DSL) einsetzt. Das statische Taskmodell wird informal festgehalten, und für den dynamischen Informationsfluß wird die auf Datenflußdiagrammen beruhende Activity-Chaining-Graph (ACG) Technik verwendet. Ruft man sich das in Abschnitt 2.1.4 vorgestellte Arch / Slinky Metamodell in Erinnerung, läßt sich die TRIDENT Architektur wie folgt in dieses Schema einbetten: Der semantische Anwendungskern entspricht der Domänenadapterkomponente. Sie hat vor allem die Aufgabe, Daten aus einer persistenten Datenhaltung zu holen und dort wieder abzulegen. Die Datenhaltung wird folglich als domänenspezifische Komponente gesehen. Die Conversation-Abstraktion der TRIDENT-Architektur materialisiert sich in der Dialogkomponente, die auch im Arch Modell denselben Name trägt. Ihr zugeordnet sind primär das Dialogmanagement und die Konsistenzwahrung von dargestellten und persistenten Daten. Die Präsentation findet sich in einer Präsentationskomponente realisiert. Sie hat die Aufgabe, die Kommunikation zwischen Anwender und Dialogkomponenten zu steuern. Zusätzlich ist sie verantwortlich für syntaktische Kontrollen der vom User eingegebenen Daten. Der toolkitspezifische Komponente des Arch Modells entspricht das physikalische Toolkit, mit dem schließlich das fertige User Interface gerendert wird. Infolge der Identifikation dieser Komponenten wurden sogenannte Application-, Dialog-, und Abstract-Interaction-Objekte definiert. Ihre Kommunikation folgt strengt den im Arch Modell aufgezeigten Wegen. Einzelheiten zur Architektur finden sich in [Bodart et al. 1993].

Die Methodologie enthält wegen der Teilung in Conversation und Presentation zwei modellbasierte Ansätze für die Dialogerstellung. Ausgangspunkt für beide Prozesse ist eine Context Analyse, die Taskanalyse und Beschreibungen von User-Stereotypen und des Arbeitsplatzes[14] beeinhaltet (s. Abb. 3.14).

Die Schritte des Ansatzes für die Erstellung der Conversation-Komponente sind wie folgt: Auf den Ergebnissen der Taskanalyse aufbauend wird eine hierarchische Architektur von Dialogobjekten definiert. Jedes Dialogobjekt ist genau für einen Teil der globalen Kommunikation zuständig. Das intradialog Objektverhalten wird mit einer regelbasierten Sprache beschrieben, die sich auch als Statecharts visualisieren läßt. Für das interdialog Objektverhalten oder einfa-

[14] Arbeitsplatzeigenschaften sind z.B. der Prozesstyp (kann der interaktive Task unterbrochen werden?) und die Prozesskapazität (sind nebenläufige Tasks erlaubt?).

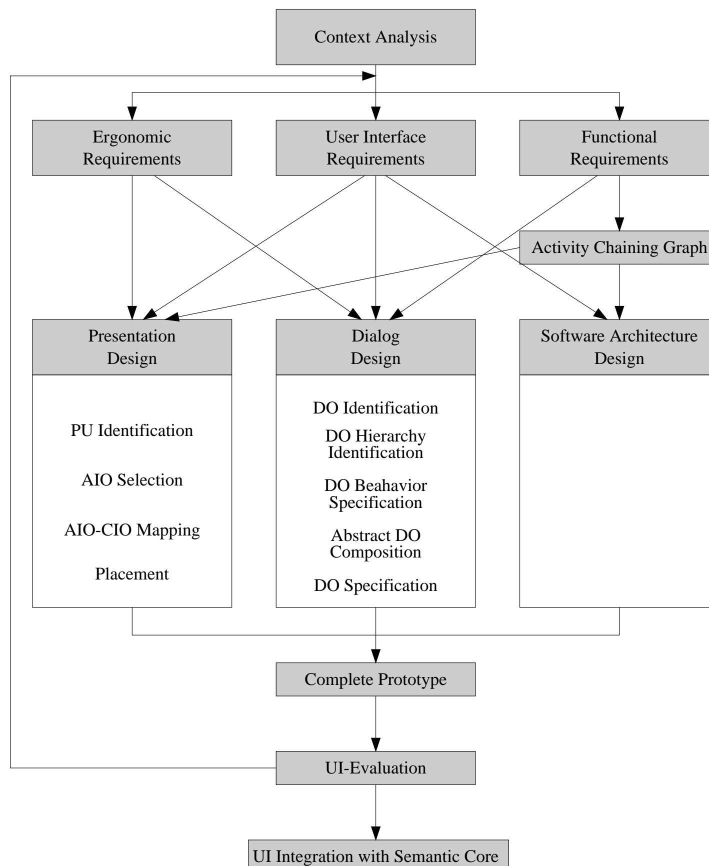


Abbildung 3.14: TRIDENT Methodologie (nach [Bodart et al. 1996])

cher gesagt die globale Kommunikation werden sogenannte Supervisory-Objekte eingeführt. Ausführlich sind die Schritte in [Bodart et al. 1993] beschrieben.

Um die Presentation-Komponente zu konstruieren, werden auf Basis der dynamischen Task-Beschreibung in Form von ACGs Presentation-Units (PUs) definiert. Eine PU wird für jeden Subtask eines interaktiven Task festgelegt und bildet damit einen sinnvollen Bearbeitungsschritt aus Usersicht ab. Im nächsten Schritt findet eine expertensystemgestützte Auswahl geeigneter Abstract-Interaction-Objects[15] für jede PU statt. Anschließend werden die abstrakten Interaktionsobjekte (AIOs) unter Benutzung einer toolkitabhängigen Abbildungstabelle auf Concrete-Interaction-Objects (CIOs) abgebildet. Im letzten Bearbeitungsschritt werden die Widgets mit Hilfe einer statischen oder dynamischen Plazierungsstrategie an ihre endgültigen Positionen innerhalb der Fenster gebracht. Eine genaue Beschreibung der Presentation-Konstruktion findet sich in [Bodart et al. 1994].

[15]AIOs sind abstrakte Elemente eines virtuellen Toolkits.

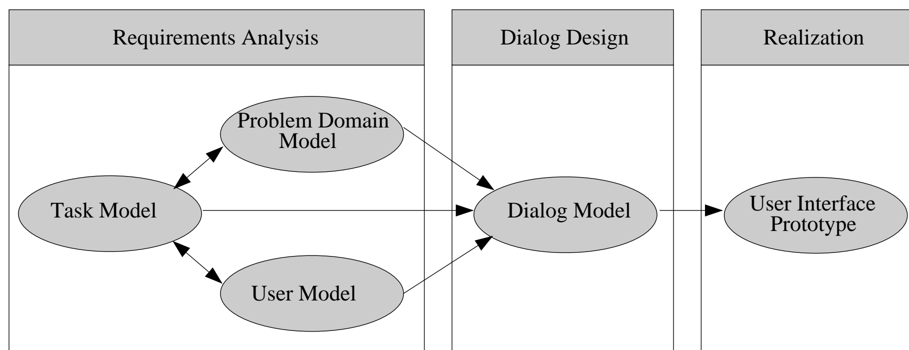


Abbildung 3.15: Der TADEUS-Ansatz (aus [Schlungbaum Elwert 1995])

Anmerkungen

Der TRIDENT Ansatz betont sehr stark die Methodologie und die Prozesse, die zur Erstellung einer interaktiven Applikation führen. Damit wird die Nutzung des TRIDENT Systems eng verknüpft mit der Akzeptanz der in schmalem Rahmen vorgegebenen Prozesse. In [Bodart et al. 1996] wird sogar ein abstraktes methodologisches Framework beschrieben, das die Schlüsselaktivitäten für die GUI-Anwendungsentwicklung zu identifizieren sucht.

Wie aus Abb. 3.14 zu erkennen ist, generiert das TRIDENT-System keinen Code zur Verbindung des UI mit dem Applikationskern. Damit muß die Integration von UI und der Anwendung nach wie vor programmiert werden, was gerade bei iterativem Applikationsdesign zu bedeutendem Aufwand führen kann.

3.2.6 TADEUS

Das TADEUS-System (TAsk-based DEvelopment of USer interface software) [Schlungbaum Elwert 1995] ist ein Projekt der Universität Rostock. Es wurde mit dem Ziel initiiert, eine aufgaben- und userzentrierte Entwicklung von Benutzungsoberflächen zu ermöglichen.

Die Methodologie von TADEUS umfasst die Prozesse, um eine interaktive Applikation zu erstellen, ein Vorgehensmodell (Action-Model), das die Schritte des Dialog-Designers während der Prozesse beschreibt und die Metamodelle, auf denen die Prozesse aufsetzen. Verwendung finden ein Task Modell, ein Problem-Domain Modell, ein User Modell und ein Dialog Modell. Zusätzlich gibt es interaktive Werkzeuge, die dem Dialogdesigner helfen, ein User Interface in einer automatisierten und computerunterstützten Form zu entwerfen. Das TADEUS Action Modell unterteilt den Entwicklungsprozess für eine Benutzungsschnittstelle in drei Phasen, die Anforderungsanalyse, das Dialogdesign und die Umsetzung (s. Abb. 3.15). Die Anforderungsanalyse bildet die Grundlage der TADEUS Methodologie. Das Taskmodell wird in einer hierarchischen Zielstruktur beschrieben. Ein Ziel wird mit der erfolgreichen Ausführung einer Aufgabe definiert und besteht aus drei Teilen, einer zugeordneten Aufgabe, einer oder mehreren Rollen und optionalen sekundären[16] Objekten. Das Domä-

[16]Sekundäre Objekte stellen Arbeitsmittel dar (vergl. WAM-Ansatz [Züllighoven 1998]), die von den primären Objekten (respektive Werkzeugen bzw. Aufgaben) zur Auführung verwendet

nenmodell wird in Form eines objektorientierten Analysemodells (OOA) nach Rumbaugh [Rumbaugh et al. 1991] beschrieben. Es enthält die Beschreibungen der primären und sekundären Objekte. Das Usermodell charakterisiert die Anwender des Systems durch Rollen. Die Aufgaben, Rollen und die Beziehungen zwischen Rollen und Aufgaben werden durch Attributwerte qualifiziert und werden zur Generierung eines prototypischen Dialogmodells eingesetzt.

Während der Dialogdesignphase bearbeitet der Dialogdesigner das generierte Modell toolunterstützt weiter, indem er Sichten des Task- und Domänenmodells identifiziert. Eine Sicht repräsentiert dabei einen sinnvollen Bearbeitungsausschnitt einer Aufgabe, der dem Benutzer gleichzeitig dargeboten wird. Innerhalb des Dialogmodells werden zwei Arten von Dialogen differenziert. Ein Navigation-Dialog dient zur Abfolgensteuerung und ein Processing-Dialog beschreibt die internen Interaktionen zwischen Sicht und dem Aufruf von Domänenfunktionalität. Zur Beschreibung des Dialogmodells wurde eine eigene Notation entwickelt, die Dialoggraphen verwendet. Dialoggraphen bestehen aus Knoten, die Dialogsichten oder Dialogobjekte darstellen und aus Transitionen, die die möglichen Interaktionen zwischen den Knoten beschreiben. Da Dialoggraphen auf gefärbten Netzen - einer speziellen Form der Petrinetze - beruhen, können sie durch ein geeignetes Mapping abgebildet werden und sind somit den bekannten Analysemöglichkeiten wie z. B. Deadlock- und Lebendigkeitsprüfungen zugänglich.

Die Umsetzungsphase generiert nun unter Verwendung einer Wissensdatenbank aus dem Dialogmodell und den anderen Modellen eine prototypische Beschreibung des User Interfaces für ein UIMS.[17] Je nachdem wie genau der Designer das Dialogmodell ausgestaltet hat, kann der Generierungsprozess zwischen vollautomatisch und stark interaktiv variieren. Fehlende Informationen werden in Form von zu beantwortenden Fragen ergänzt.

Anmerkungen

TADEUS unterstützt einen homogenen und kontinuierlichen Entwicklungsprozess von der Anforderungsanalyse einer interaktiven Applikation bis zum generierten Interfaceprototyp. Besonderer Wert wurde auf die Einführung eines neuartigen Dialogmetamodells gelegt. TADEUS trennt damit explizit die Taskmodellierung vom Dialogdesign. Der Übergang von den identifizierten Tasks zu den entsprechenden Sichten wurde als wichtiges Mappingproblem erkannt und soll durch die Bereitstellung von beratenden Tools vereinfacht werden (siehe dazu [Schlungbaum 1997]). Die Technik der Dialoggraphen ermöglicht eine genaue Beschreibung der groben Dialogsteuerung. Elwert [Elwert 1996] betont die Wichtigkeit eines Dialogmodells für die Beschreibung aller Stufen der Dialogmodellierung mit einer einheitlichen Notation. Daher werden in TADEUS auch für die feine Dialogsteuerung Dialoggraphen eingesetzt. Inwieweit auf dieser Ebene ein so komplexer Mechanismus benötigt wird, ist zu überprüfen.

Bemerkenswert ist der Anspruch der TADEUS Entwickler, die modellbasierte GUI-Entwicklung in Richtung der aktuellen Techniken des Softwareengineering voranzubringen. Dazu gehören neben der Konzentration auf ein taskbasiertes, userorientiertes Design auch besonders die Beschreibungssprachen der Modelle. In [Stoiber 1999] wird untersucht, inwieweit UML geeignet ist, die in

werden.

[17] Verwendung findet hier der ISA Dialog Manager.

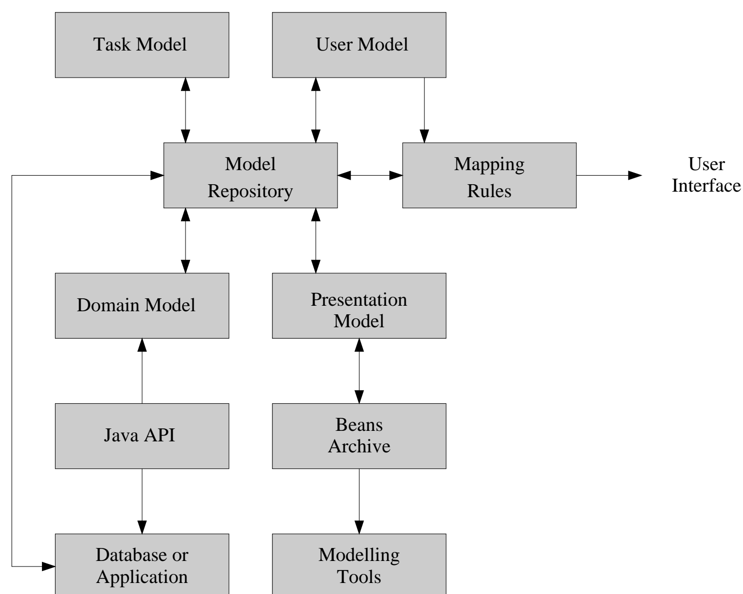


Abbildung 3.16: Teallach Architektur (aus [Griffiths et al. 1998b])

TADEUS verwendeten Modelle auszudrücken. In vielen Fällen wurde dabei eine starke strukturelle Ähnlichkeit oder sogar eine Äquivalenz der Semantik der Notationen festgestellt.

3.2.7 Teallach

Das Teallach-System[18] [Gray et al. 1998, Griffiths et al. 1998b] ist ein Projekt der Universität Manchester. Motivation für das Projekt ist die beobachtete Entwicklung von einfachen Internetseiten und -applikationen hin zu immer komplexeren Oberflächen. In vielen Fällen basieren diese Services auf Datenbankapplikationen, für die auch nicht webbasierte Interfaces verfügbar sein müssen. Um den Entwicklungsaufwand für derartige Applikationen zu vereinfachen, wurde das Teallach Projekt initiiert. Dem Projekt sind eingehende Studien über die Eignung existierender MB-UIDEs im Datenbankkontext vorausgegangen (siehe [Griffiths et al 1998a]).

Um ein User Interface zu definieren, werden vier Modelle verwendet, ein Domänenmodell (domain model), ein Aufgabenmodell (task model), ein Benutzermodell (user model) und ein Präsentationsmodell (presentation model). Gespeichert werden die Modelle in einem gemeinsamen Repository (shared model repository) (s. Abb. 3.16). Dadurch kann es in einigen Fällen erreicht werden, aus bereits vorhandenen Daten spezifizierter Modelle den Initialzustand eines anderen Modells abzuleiten, z. B. werden das Aufgabenmodell und das Benutzermodell eingesetzt, um ein erstes Präsentationsmodell zu generieren.

Das Domänenmodell wurde so konzipiert, dass mit ihm sowohl die Funktionalität einer ODMG-konformen Datenbank als auch einer nicht persistenten

[18] Der Begriff Teallach stammt aus dem Gaelischen und bedeutet Schmiede. In der Vergangenheit wurden auch Amboß oder Schmelzofen damit bezeichnet. In jedem Fall bringt man den Begriff mit einem Ort in Verbindung, an dem Werkzeuge hergestellt werden.

Applikation beschrieben werden kann. Als Metamodell wurde daher das ODMG-Datenmodell [Cattell 1997] gewählt. Da die primäre Motivation des Teallach Systems darin besteht, die Entwicklung von User Interfaces für Datenbankapplikationen zu vereinfachen, sieht das Domänenmodell die Datenbank durch ein ODMG-Java Binding als eine Menge von Java-Klassen, welche die Struktur und Funktionalität der Datenbank und ihrer Metadaten reflektiert. Sofern Teallach für nicht persistente Anwendungen eingesetzt werden soll, muß sichergestellt werden, dass die Applikation in Form von Java-Klassen zugänglich ist und zusätzlich eine Möglichkeit besteht, Meta-Informationen über diese Klassen abzurufen. Als visuelle Repräsentation des Teallach Domänenmodells wurde eine Teilmenge des UML-Klassendiagramms gewählt.

Das Aufgabenmodell ist dazu gedacht, Handlungen zu beschreiben, die Benutzer mit dem System durchführen wollen. Diese Aufgaben werden in einer geordneten hierarchischen Struktur definiert, die auch Konstrukte zur Spezifikation temporaler Abhängigkeiten beeinhaltet. Die Struktur und Semantik der initialen Aufgabenhierarchie ähnelt den von Adept [Markopoulos et al. 1992] und TADEUS (siehe Abschnitt 3.2.6) benutzten Strukturen. Es wurden sieben konzeptuell verschiedene temporale Spezifikationen herausgearbeitet: sequentiell (sequential), reihenfolgen-unabhängig (order-independent), wiederholbar (repeatable), nebenläufig (parallel), nebenläufig-synchronisiert (interleaved), auswählbar (choice) und wahlweise (optional). Genaue Erklärungen zu den verschiedenen Arten der zeitlichen Abfolgen findet man in [Griffiths et al. 1998b]. Auf der untersten Detailstufe werden Aufgabenmodellentitäten durch Ausdrücke einer Expressionlanguage direkt an Domänenmodellkonzepte gebunden. Zusätzlich zur Hierarchie wird der interne Datenfluß durch Zustandsinformationen beschrieben. Die Komplexität des Taskmodells wird in Teallach durch die Einführung von verschiedenen Sichten auf das Modell beherrschbar. So können jeweils für einen Designschritt unwichtige Aspekte der Modellierung bei Bedarf ausgeblendet werden.

Das Benutzermodell beinhaltet Informationen über den präferierten User Interface Style von einzelnen Benutzern und Benutzergruppen. Arbeitet das Telleach-System mit einer Datenbankapplikation wird das Modell auch eingesetzt, um Informationen zur Berechtigungsstufe und den damit einhergehenden Datenzugriffs- und Manipulationsmöglichkeiten abzulegen. Weiterhin kann es dazu verwendet werden, persönliche, die Oberfläche betreffende Einstellungen festzulegen.

Das Präsentationsmodell existiert auf abstrakter und konkreter Ebene. Die abstrakte Form (abstract presentation model) reflektiert genau die Struktur des Aufgabenmodells, da eine Abbildung von Aufgaben auf Präsentationskonzepte benutzt wird. Ausgehend von dem abstrakten Modell, können beliebig viele verschiedene konkrete Präsentationsmodelle definiert werden. Konkrete Modelle (concrete presentation model) beinhalten genug Informationen, um als User-Interface dargestellt zu werden. Sie können durch Anwendung von Abbildungsregeln und Informationen aus dem Benutzermodell automatisch generiert werden oder von einem Designer mit Hilfe eines design-time Tools neu erstellt bzw. verfeinert werden. Eine Besonderheit des Präsentationsmodells von Teallach ist die Verwendung von beliebigen Java Beans [Hamilton 1997] als Widgets. Um eine solch offene Architektur zu erreichen, muß das Modell Informationen zu den verfügbaren (registrierten) Beans verwalten. Einzelheiten zu den verwendeten Kriterien werden in [Gray et al. 1998] erläutert.

Um schliesslich ein ablauffähiges Interface zu erhalten, kann der Designer zwischen zwei verschiedenen Vorgehensweisen wählen. Er kann sich entweder für eine interpretierbare oder für eine compilierte Version (in Java) entscheiden (siehe dazu auch 3.2.8). Im Normalfall wird eine interpretierbare Version während der Entwicklungphase vorgezogen, da der Compilierungsprozess aufwändig ist und viele Zyklen durchlaufen werden müssen. Ist die Entwicklung weitgehend abgeschlossen, können durch compilierte Versionen Performancevorteile erzielt werden.

Anmerkungen

Eine Besonderheit des Teallach-Ansatzes ist seine offene Architektur (openness), die durch eine effiziente und transparente Kommunikation der MB-UIDE zur Applikation und zu den Interface Komponenten gekennzeichnet ist. Es wurden vier Kernaspekte identifiziert, die eine MB-UIDE Architektur als offen kennzeichnen:

Applikationswissen: Ein modellbasiertes System muss eine grundlegende Kontrollfunktionalität über interne Applikationsabläufe besitzen. So hat das System z. B. auf propagierte Ereignisse oder Ablaufunterbrechungen adäquat zu reagieren, indem es diese in geeigneter Form im User Interface repräsentiert.

Externer Datenfluß: Das User Interface ist für den Dialog zwischen Benutzer und Applikation verantwortlich. Eine offene MB-UIDE hat also die Aufgabe, Informationen von und in den Applikationskern zu transportieren.

Interner Datenfluß: Der interne Datenfluß bezieht sich auf den Workflow der Applikation. Betrachtet man die Applikation aus Sicht der Usertasks, ist es notwendig, Zustandsinformationen bei Zustandswechseln zu übertragen.

Widgetrepository: Ein modellbasiertes System sollte die einfache Einbindung weiterer Toolkits erlauben und so z. B. die Integration neuer Interfacemodalitäten motivieren.

Werden alle Punkte außer dem letzen erfüllt, integriert ein System Interface und Applikation, d. h. die Kommunikation zwischen Interface und der Applikation wird durch die MB-UIDE eingerichtet. Sofern das nicht der Fall ist, ist mit einem erheblichen programmatischen Mehraufwand zu rechnen.

Im folgenden wird noch einmal kurz zusammengefasst, wie das Teallach-System diese Anforderungen erfüllt. Applikationswissen fließt in Teallach mit in das Aufgabenmodell ein. Exceptions werden explizit modelliert und finden sich damit auch im Präsentationsmodell wieder. Das Widgetrepository von Teallach ist durch die Verwendung von Java Beans leicht erweiterbar und plattformunabhängig. Der Informationstransport zwischen Applikation und Interface wird durch Eventspezifikationen für einzelne Beans des Repositories genau festgelegt. Um Zustandsinformationen von einem Zustand in den Nachfolgezustand zu übertragen, wird das Aufgabenmodell mit Inschriften verfeinert. Eine Expressionlanguage bietet dafür alle notwendigen Konstrukte.

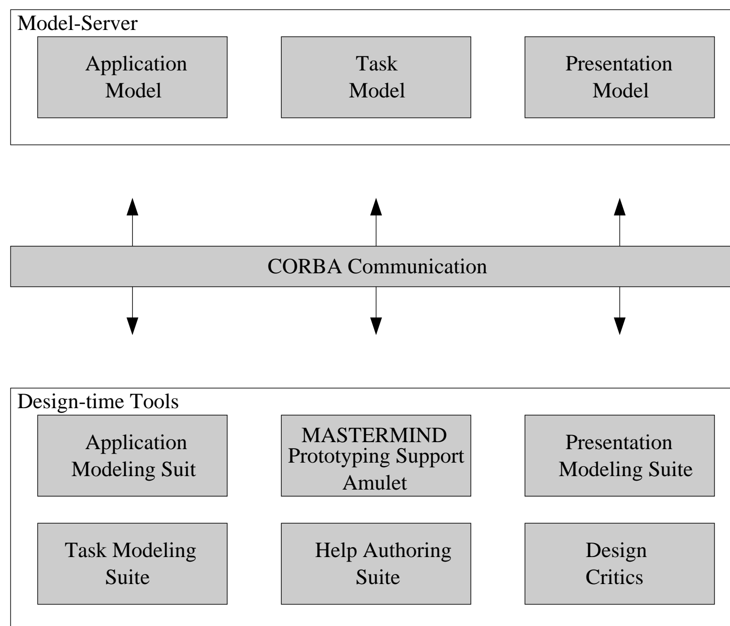


Abbildung 3.17: MASTERMIND Designumgebung (aus [Szekely et al. 1995])

3.2.8 MASTERMIND

Das MASTERMIND-System ist ein Gemeinschaftsprojekt vom Graphics, Visualization, and Usability Center (GVU), dem Georgia Institute of Technology (GT) und dem Information Sciences Institute (ISI) der University of Southern California [Szekely et al. 1995, Browne et al. 1997, Rugaber 1998]. Das Projekt vereint die Stärken der modellbasierten Ansätze UIDE [Foley et al. 1991] und HUMANOID [Szekely 1992a]. Herausragendes Ziel war es, ein MB-UIDE zu entwerfen, dass die Hauptschwachstellen (siehe Abschnitt 2.3.6) existierender Systeme vermeidet und es erlaubt industriell einsetzbare Applikationen zu generieren.

MASTERMIND besitzt daher verschiedene Architekturen für Anwendungen im Entwicklungs- und im Auslieferungsstadium. Die designtime Architektur (s. Abb. 3.17) unterstützt einen iterativen Entwicklungsprozeß und mächtige Designwerkzeuge. Sie verwendet dazu die Modelle in einer deklarativen Form. Die Architektur für fertiggestellte Anwendungen wurde auf Performanceaspekte abgestimmt. Hier finden kleine compilierte Modellrepräsentationen Anwendung, die eine schnelle User Interface Ausführung erlauben.

MASTERMIND verwendete zunächst drei unabhängige Modelle zur Spezifikation von User-Interfaces, ein Application-Model, ein Task-Model und ein Presentation-Model [Szekely et al. 1995], die alle als Notation auf CORBA IDL [OMG 2000b] zurückgreifen. Um die Verbindungen zwischen den einzelnen Modellen herzustellen, besitzt MASTERMIND eine Sprache zur Beschreibung von Ausdrücken (Expressionlanguage). Beispiele solcher Ausdrücke sind Aufrufe von Applikationsmethoden, Prädikate, arithmetische Ausdrücke, Fallunterscheidungen und Iterationen. MASTERMIND überwacht diese Ausdrücke und berechnet sie bei Bedarf automatisch neu. Das Application-Model ist eine Erweiterung

des CORBA [OMG 2000b] Objektmodells um zwei Konzepte. So wird es dem Entwickler durch die Einführung von Vorbedingungen (Preconditions) möglich gemacht festzulegen, wann es erlaubt ist, bestimmte Methoden aufzurufen. Die zweite Erweiterung betrifft Abhängigkeiten von Objekten untereinander. Durch die Festlegung von Berichten (Reports) kann genau angegeben werden, welche Arten von Änderungen ein Objekt bekannt gibt. Andere Objekte können sich an für sie interessanten Reports registrieren und werden fortan über Änderungen benachrichtigt (Observer-Pattern [Gamma et al. 1995]).

Das MASTERMIND Taskmodell dient zur Beschreibung von Aufgaben, die ein Anwender durch Kommunikation mit dem System ausführen kann. Für jeden Task müssen sein Ziel, die Bedingungen unter denen er ausführbar ist, seine Auswirkungen, die benötigten Informationen und die Dekomposition in Subtasks angegeben werden. Eine Aufgabe besteht dabei aus einer beliebigen Kombination von User-, Interface-, und Applicationtasks. Usertasks ermöglichen die Angabe von notwendigen Eingaben, die von einem Anwender erwartet werden. So sind auch einfache Interaktionstechniken, wie das Klicken eines Buttons in die Kategorie Usertasks einzuordnen. Interfacetasks legen fest, in welcher Form MASTERMIND die Oberfläche aktualisieren soll, und Applicationtasks bestimmen welche Anwendungsroutinen aufgerufen werden.

Mit Hilfe des Presentation-Models wird die visuelle Erscheinung des User-Interfaces definiert. Erreicht wird dies durch eine hierarchische Anordnung von Präsentationsobjekten, die eine Oberfläche als Ganzes beschreiben. Die wichtigsten Eigenschaften eines Präsentationsobjektes sind der Prototyp, der als Grundlage für das neu definierte Objekt dient, die Subparts, die ebenfalls Präsentationsobjekte sind und weitere Einstellungen, die u. a. für das Layout verwendet werden. Alle Eigenschaften können durch die Angabe von Bedingungen dynamisch verändert werden. So können alternative Präsentationen in Abhängigkeit der dargestellten Daten oder Displayeigenschaften ausgewählt werden. Für das Layout der Präsentationsobjekte wurde in MASTERMIND das Konzept der Gitter und Führungslinien (grid and guides) integriert. Es handelt sich dabei um eine einfache und mächtige Technik, die seit Jahren im Bereich des Buch- und Zeitungssatzes eingesetzt wird und Teil vieler Interface Richtlinien ist. Sie besitzt im Vergleich zu den üblichen Verfahren vieler Interfacebuilder den Vorteil, daß alle Elemente unabhängig von ihrer Tiefe innerhalb der Darstellungshierarchie an bestimmten Linien ausgerichtet werden können.

Im Zuge des Projektfortschritts wurden weitere Modelle in MASTERMIND integriert. Darunter ein auf kontextfreien Grammatiken beruhendes Dialogmodel, ein Interactionmodel zur Spezifikation möglicher low-level Interaktionen zwischen User und System, ein Contextmodel zur Deklaration von Beziehungen zwischen Präsentationsobjekten und sogenannte Application Wrappers. Angedacht wurde auch die Einbettung von User-, Toolkit- und Display-Device Modellen. Einzelheiten zu diesen Modellen finden sich in [Browne et al. 1997].

Um eine ausführbare Version des User-Interfaces zu erhalten, werden die Modelle durch Codegeneratoren in die Zielsprache (C++) übersetzt. MASTERMIND besitzt separate Generatoren für die einzelnen Modelle, die unabhängig voneinander arbeiten. Dadurch müssen in einem weiteren Schritt die Übersetzungen der Einzelmodelle zusammengesetzt werden. Grundlage dieser Modellkomposition bildet eine multi-agenten User Interface Architektur, die in [Rugaber 1998] erläutert wird.

Anmerkungen

Die kommerzielle Ausrichtung des MASTERMIND Projektes rückt einige wichtige Aspekte modellbasierter User Interface Entwicklungssysteme in den Vordergrund. So spielen in vielen rein wissenschaftlichen Systemen folgende Punkte nur eine untergeordnete Rolle:

- einfache Benutzbarkeit
- parallele Gruppenarbeit
- Performance
- Runtime-Tools

MASTERMIND ermöglicht das Arbeiten mit mehreren Teams an einem Projekt durch die Datenhaltung der Modelle in einem Model-Server, der für die Konsistenz der Daten sorgt und entfernte Zugriffe erlaubt (s. Abb. 3.17). Die Performance der von MASTERMIND erzeugten Applikationen wird durch die Trennung von Design- und Industrieversionen optimiert. Um die Benutzbarkeit des Systems zu verbessern, war es geplant, zahlreiche Designwerkzeuge bereitzustellen (s. Abb. 3.17). Bekannt ist, dass ein Taskeditor (Dukas) und ein modellbasierter Interfacebuilder zur Komposition von Benutzungsschnittstellen per drag-n-drop realisiert wurden. Das geplante Hilfesystem und die DesignCritics / Advisor Komponenten werden im Projektabschlussbericht [Rugaber 1998] nicht im Abschnitt der umgesetzen Tools erwähnt.

3.2.9 Business Component Prototyper für SanFrancisco

[van Emde Boas 2000] beschreibt ein Werkzeug zur Unterstützung eines prototypischen User Interface Entwurfes mit dem IBM SanFrancisco Framework (siehe 3.1.2). Ziel des Prototypers ist es, eine Umgebung anzubieten, die Modellierung und Programmierung in einer integrierten Art und Weise unterstützt.

BC-Prototyper besteht aus einem Modellierungswerkzeug, einer Programmierumgebung, einem Code-Generator, einem GUI-Builder und SanFrancisco Utilities. Verwendete Technologien beinhalten XML [Bray et al. 2000], JavaBeans [Hamilton 1997] und Introspection. Die Werkzeuge wurden mit dem BC-Prototyper selbst entwickelt, d. h. der Prototyper erzeugt sich aus einem Modell selbst und kann daher genutzt werden, um sich selbst anzupassen.

Modellinformationen werden über die JavaBean Introspection Technik zugegriffen. Über Javas Reflection-Mechanismus kann der Prototyper aus Klassen, die als `.class` Dateien vorliegen, ein Modell erzeugen. Vorlagen (Templates) für den Code-Generator werden in XML spezifiziert. Der Code-Generator erzeugt aus dem Modell den Java-Code für das User Interface. Um den Generator zu beeinflussen, können im Modell die Eigenschaften der darzustellenden Elemente näher spezifiziert werden (s. Abb. 3.18). So kann man für ein Attribut eines Objektes ein Label angeben und den Widgettyp festlegen.

BC-Prototyper verwendet festgelegte Abbildungsregeln, um aus dem Modell ein User Interface zu erzeugen: Z. B. werden Klassen auf Fenster (`Frame`), Attribute auf Widgets (`TextField`, `TextArea`, `Checkbox`, `Choice`, `Button`) und eine 1-zu-N-Beziehung auf eine Liste abgebildet. Attribute und Listen können dabei einer sog. TabGroup zugewiesen werden, die verwendet wird, um die Dialoge in Tab-Panels aufzuteilen.

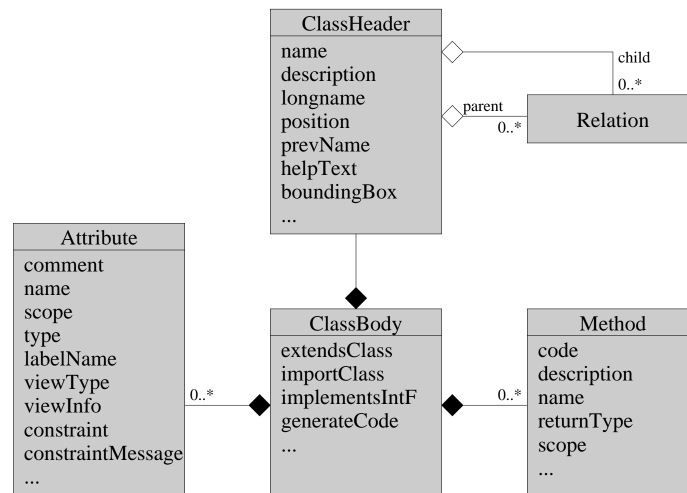


Abbildung 3.18: Metamodell des BC-Prototyper (aus [van Emde Boas 2000])

Anmerkungen

Wie Janus (siehe Abschnitt 3.2.2) verwendet der BC-Prototyper nur ein Modell, es sind also dieselben Einschränkungen vorhanden, die schon an Janus kritisiert wurden. [van Emde Boas 2000] behauptet, dass die vorgestellten Abbildungsregeln ausreichen, um benutzbare User Interfaces zu erzeugen. Diese Behauptung findet sich jedoch in der Literatur (z. B. [Szekely 1996]) widerlegt. Für prototypische User Interfaces mögen die Regeln jedoch ausreichen, um einen ersten Eindruck von der Funktionalität einer Anwendung zu erhalten. Die bei Bedarf automatische Auswahl geeigneter Widgets ist bei der Anwendungsentwicklung eine große Hilfe und wird auch von [Szekely 1996] generell nicht in Frage gestellt.

Um einigermaßen benutzbare Oberflächen erzeugen zu können, enthält das Metamodell, wie in Abbildung 3.18 zu sehen, viele Attribute, die eigentlich Teil eines Präsentationsmodells sein sollten (labelName, viewType, constraint-Message, boundingBox, ...), BC-Prototyper verwendet also ein kombiniertes Domänen-Präsentationsmodell.

Der Code-Generator ist dank eines Templatemechanismus sehr flexibel. Es ist leicht vorstellbar, dass für verschiedene Toolkits (AWT, Swing) verschiedene Templates verwendet werden können, die aus demselben Anwendungsmodell unterschiedliche Benutzungsschnittstellen generieren können. Wahrscheinlich wäre es sogar möglich, wenn man sich geeignete Templates schreibt, eine Web-Anwendung auf HTML-Basis zu generieren. Außerdem wird die Möglichkeit angesprochen, auch weitere Aspekte wie Dokumentation oder Konfigurationsdateien zusätzlich zur eigentlichen Anwendung zu generieren.

Der Java-Reflection Ansatz ermöglicht, auch User Interfaces für Objekte zu entwerfen, die zunächst nicht mit dem BC-Prototyper entwickelt wurden.

3.2.10 Vergleich der modellbasierten Systeme

In Abb. 3.19 sind die modellbasierten Systeme in Bezug auf einige wichtige Faktoren tabellarisch gegenübergestellt. In Anlehnung an [da Silva 2000] werden die

Ansätze anhand der verwendeten Modelle und ihrer Notationen, ihrer Architekturmerkmale und der zur Verfügung gestellten Werkzeugpalette verglichen.

Alle betrachteten Systeme verwenden ein Applikations- und ein Präsentationsmodell. Ruft man sich die Geschichte der MB-UIDEs in Erinnerung, ist dies nicht verwunderlich, da sie als Weiterentwicklung der UIMS konzipiert wurden und dort eine eindeutige Trennung zwischen User Interface und dem Anwendungskern zwingend notwendig war. Viele Applikationsmodelle basieren auf Softwareengineeringstandards (ERA, ODMG OM, CORBA OM, ...) und profitieren davon in mehrfacher Hinsicht. Neben der impliziten Robustheit sind in vielen Fällen Werkzeuge zur Spezifikation bereits vorhanden. Zusätzlich sinkt die Hemmschwelle für Erstanwender, da ein Teil der verwendeten Techniken bereits bekannt ist und nicht alles vollkommen neu erlernt werden muß. Für die Präsentationsseite sind kaum Standards verfügbar, so dass fast alle Systeme selbstdefinierte Modelle verwenden. Bemühungen, auch für die Präsentationsschicht Standards zu formulieren, werden z. B. in [da Silva Paton 2000a] verfolgt. Einige Systeme wie Janus und TADEUS weisen gar kein explizites Präsentationsmodell auf, da der Benutzer aufgrund der starken Automatiserung nicht mit dem Modell in direkten Kontakt gerät, sondern lediglich den Generierungsprozess durch Parametrisierung steuern kann.

Die meisten der aufgeführten Ansätze nutzen Taskmodelle, um die User-Zentrierung zu betonen und die grobe Dialogsteuerung zu vereinfachen. In vielen Systemen finden hierarchische Zielstrukturen zur Definition des Aufgabenmodells Anwendung, die meist Erweiterungen bekannter Standards wie z. B. HTA, GOMS (vergl. Abschnitt 2.2.4.1) sind. Bei Systemen, die kein Dialogmodell aufweisen, werden oftmals die Informationen über Tasks in einfacher Form (eins-zu-eins) auf Dialoge abgebildet. In welcher Art und Weise und in welchem Maße die Informationen aus dem Taskmodell für ein Dialogmodell relevant sind, ist nicht vollständig geklärt und wird u. a. durch TADEUS und Mobi-D untersucht.

Einige der Systeme verwenden weitere Modelle, wie User-, Integrations-, Kontext-, Interaktionsmodelle. Da das User Management in vielen Systemen eine wichtige Rolle spielt, ist die Einführung eines eigenen Modells sicher gerechtfertigt. Insbesondere bei Systemen, die auch Persistenzmechanismen bereitstellen wollen, wird eine separate Rechte- und Rollenverwaltung unverzichtbar (Janus, Teallach). Leider wird das Benutzermodell in der Literatur wenig diskutiert, so dass nicht eindeutig ist, welche Daten es enthalten sollte. Die Relevanz anderer Modelle bleibt unklar.

Im Architekturabschnitt werden die Einflüsse der Architektur auf die Modellierungsmöglichkeiten und das erzeugte User Interface dargestellt. Die identifizierten Kriterien erweitern die in [Griffiths et al. 1998b] vorgestellten Merkmale um den Aspekt der Offenheit (openness), der den Punkt der Interface Applikationsintegration mit einbezieht (siehe Abschnitt 25). Weiterhin wird das Vorhandensein bzw. die Art der Methodologie (methodology) dargestellt. Eine Methodologie schreibt in vielen Fällen die strikte Ordnung der auszuführenden Entwicklungsaktivitäten vor, führt also zu sequentiellen (sequential) Prozessabläufen. Wurde keine Methodologie vorgeschlagen, ist die Reihenfolge der Prozesse nur implizit vorgeschrieben (ad-hoc). Eine wichtige Eigenschaft einer Methodologie ist es, ob Abläufe automatisiert werden (automation) oder eine umfangreiche Spezifikation (specification) notwendig ist. Die Wichtigkeit einer Methodologie zur Interface Entwicklung wird in der Literatur kontrovers disku-

Model / Notation	Application	Task	Dialog	Presentation	Other Models
Janus	OOA	None	None	Implicit	Database Model
	JDL (Janus Definition Language)			Part of AM	In Application Model
Mobi-D	MIM (Mecano Interface Model)	MIM	MIM	MIM	Design Relations User Types
	MIMIC (Mecano Interface Modeling Language)	MIMIC	MIMIC	MIMIC	
FUSE	FUSE Object Model	FUSE Task Model	None	FUSE Presentation Model	User Model
	Algebraic specification	HTA		HIT (Hierarchic interaction graph template)	
TRIDENT	ERA	TRIDENT Task Model	Dialogmodel on basis of Taskmodel	Trident Presentation Model	User Stereotypes
	DSL (Dynamic Specification Language)	ACG (Activity Chaining Graph)	Dialog/Supervisory Objects	Custom (not labeled)	In Task Model
Teallach	ODMG Data Model	Teallach Task Model	None	Teallach Presentation Model	User Model
	ODMG ODL	Hierachical tree with state objects		Custom (not labeled)	
MASTERMIND	Extended CORBA OM	MM Task Model	MM Dialog Model	MM Presentation Model	Interaction Model Context Model Application Wrapper
	MDL (MASTERMIND Definition Language)	MDL	MDL	MDL	
TADEUS	OOA	TADEUS Task Model	TADEUS Dialog Model	Implicit	User Model
	OMT	Hierachical goal structure	Dialogue Graphs	UIMS Description Language	
BC-Prototyper	OOA	None	None	Implicit	None
	UML			Part of AM	

Architecture	Methodology	Openness	Multi-Platform Support	Supported Interfaces
Janus	Ad-Hoc (Automation)	Integration Widgetrestriction	None	WIMP
Mobi-D	Sequential (Decision)	Integration Widgetrestriction	None	WIMP
FUSE	Ad-Hoc (Automation)	None	None	N/A
TRIDENT	Sequential (Automation)	None	None	N/A
Teallach	Ad-Hoc (Automation)	Complete	Platform	WIMP
MASTERMIND	Ad-Hoc (Specification)	Integration Widgetrestriction	Platform, Environment	WIMP
TADEUS	Sequential (Automation)	None	Platform	WIMP
BC-Prototyper	Ad-Hoc (Automation)	Complete	Platform	WIMP

Tools	Modelling	Design	Implementation	Others
Janus	Together C++ Paradigm Plus	Automated Design	C++ Code Generator	None
Mobi-D	Task Elicitation/ Modelediting Tools	Decision Support Tools	C++ Code Generator	Help Generator Usability Analyser
FUSE	FIRE	FLUID, BOSS	BOSS (UIMS File)	PLUG-IN
TRIDENT	Direct Manipulation Editor	Selection of various Rules	UIMS File Generator	None
Teallach	Task-Editor, Wizards	Design Editor	Java Code Generator / Interpreter	None
MASTERMIND	Application Suites	Presentation Modelling Suite	Java Code Generator / Interpreter	Design Critics, Advisors
TADEUS	Dialogue Graph Editor	Editable Mapping Tables	UIMS File Generator	None
BC-Prototyper	UML-Editor / Reverse Engineering	Automated Design	Template Java Code Generator	None

Abbildung 3.19: Modellbasierte Ansätze - Überblick

tiert (man vergleiche TRIDENT vs. Teallach), denn einerseits führt die Etablierung einer Methodologie zu mehr Disziplin bei der Entwicklung, andererseits wird dadurch der Freiraum des Entwicklers eingeschränkt.

Weiterer Vergleichspunkt ist die Unterstützung von verschiedenen Plattformen (multi-platform support). Dabei werden zwei Varianten unterschieden. Erstens kann ein System auf verschiedenen Zielplattformen bzw. Betriebssystemen mit entsprechender Anpassung der Widgets an den jeweiligen Plattformstil lauffähig sein (platform). Zweitens kann ein MB-UIDE verschiedene Interfaces in Abhängigkeit der Anwendungumgebung (environment) erzeugen (z. B. Handheld, Laptop, Computer mit Großleinwand). Der letzte Punkt zeigt auf, welche Arten von Interfaces (supported interfaces) unterstützt werden (z. B. WIMP = Window Icon Menu Pointer, maskenbasiert, textuell, 3D, . . .).

Die Einteilung der Softwaretools entspricht in etwa der von [Szekely 1996]. Unter Modellierungs-Tools werden Werkzeuge zusammengefasst, die dem Anwender helfen, Modelle zu erstellen. Diese Werkzeuge dienen in erster Linie dazu, die Syntax der Modellierungssprache vor dem Entwickler zu verbergen und eine adäquate Darstellungsweise auch für große Informationsmengen bereitzustellen. In diesem Bereich wurde eine Vielzahl unterschiedlicher Tools entwickelt, die von einfachen Texteditoren (MASTERMIND) bis zu komplexen grafischen Editoren (FUSE, Mobi-D) reichen. Für einige Systeme wurden vorhandene CASE Tools erweitert (Janus).

Unter (automatisierten) Screendesign-Tools sollen Werkzeuge verstanden werden, die beim Design des abstrakten oder konkreten User-Interfaces eine Rolle spielen. Auf der einen Seite gibt es Systeme wie z. B. Janus, die sämtliche Schritte des Designprozesses automatisch ableiten und daher regelbasierte Hilfskomponenten bereitstellen. Dieser Ansatz weist eine Reihe schwerwiegender Probleme auf (siehe [Szekely 1996]). Auf der anderen Seite existieren Systeme wie z. B. MASTERMIND, die eine vollständige Spezifikation des Designs erfordern. Dafür bieten diese Systeme dem Entwickler wesentlich mehr Flexibilität und Kontrolle. Außerdem ist in den meisten Fällen eine Wiederverwendung einmal spezifizierter Komponenten vorgesehen. Zwischen diesen beiden Extrema befinden sich Systeme, die nicht den Automatisierungsgedanken völlig aufgegeben haben und dem Entwickler dennoch mehr Freiraum geben wollen. Zu dieser Art Systeme gehört z. B. das Mobi-D System, das Werkzeuge zur Begrenzung des Designraumes bietet und damit den Entwickler vor einfacher zu entscheidende Probleme stellt.

Die Gruppe der Implementationstools schließt diejenigen Tools ein, die zur Übersetzung der Modelle in eine ausführbare User-Interface Version verwendet werden. Drei Arten von Implementationstools lassen sich klassifizieren: Source-Code Generatoren (Janus, MASTERMIND, Mobi-D, BC-Prototyper) erzeugen Code einer Zielsprache, oft C++. UIMS Generatoren (FUSE, TADEUS) generieren Quellcode, der direkt von einem UIMS oder einem Interface Builder gelesen werden kann. Interpreter (MASTERMIND) werten die Modelle zur Laufzeit aus.

Neben der eigentlichen User Interface Erzeugung bieten einige Systeme auch die Erzeugung weiterer Komponenten an. Vielfach wurden Hilfesysteme (FUSE, Mobi-D) realisiert, die durch Auswertung von Modellinformationen kontextsensitive Hilfe anbieten können. Weitere Ansätze forschen in Richtung Design Critics (MASTERMIND), Design Advisors (TRIDENT) und Usability-Analyzer (Mobi-D).

3.2.11 Weitere Forschungsansätze

Weitere untersuchte Forschungsansätze im Bereich der MB-UIDEs, auf die im Kontext dieser Arbeit nicht näher eingegangen wird, sind in Tabelle 3.2 aufgeführt.

Name	Quelle
AME	[Märtin 1996]
Adept	[Markopoulos et al. 1992]
Don/UIDE	[Kim Foley 1990, Foley et al. 1991]
Drive	[Mitchell et al. 1995]
Genius	[Janssen et al. 1993]
GIPSE	[Patry Girard 1999]
Higgens	[Hudson King 1988]
HUMANOID	[Szekely 1990, Szekely 1992a]
ITS	[Wiecha et al. 1990]
MIKE	[Olsen 1992]
MIKEY	[Olsen 1992]
Modest	[Birnbaum et al. 1997]
TACTICS	[Kovacevic 1993]

Tabelle 3.2: Weitere modellbasierte Systeme

3.3 Zusammenfassung

In Abb. 3.20 sind die für das UbiComp relevanten Eigenschaften zur Erstellung von User Interfaces (siehe Abschnitt 1.3) für die beiden Systemgruppen Frameworks und modellbasierte Systeme zusammengefasst.

Alle vorgestellten nicht-deklarativen Systeme erreichen die Trennung von User Interface und Applikationskern durch die in Kapitel 2 vorgestellten MVC oder PAC Designpatterns. Diese agentenorientierten Architekturen eignen sich besonders gut für die Verwendung im Frameworkkontext, da das Framework zumeist selbst objektorientiert realisiert ist und so eine direkte Anwendbarkeit der Muster auf die einzelnen Komponenten gewährleistet ist. Positiv hervorzuheben ist auch die vereinfachte Erstellung von Applikationen mit der Hilfe von Frameworks. Dies ist zum einen sicherlich dadurch begründet, dass Entwickler sich durch die Nähe zur Implementation leicht in derartige Systeme einarbeiten können und keine Abstraktionshürden überwinden müssen. Zum anderen sind zumindest Whitebox Frameworks leicht erweiterbar und erlauben so, in einfacher Weise Anpassungen des Systems durchzuführen. Ebenfalls technisch gut gelöst ist die Anbindung des User Interfaces an die Fachlogik. Da die Implementationsebene nicht verlassen wird, können auch hier Entwurfsmuster, wie Observer- oder Command-Pattern [Gamma et al. 1995], eingesetzt werden. Die Erweiterbarkeit der Systeme in Bezug auf weitere User Interface Modalitäten wurde von den getesteten Frameworks völlig ausgeklammert, obwohl sie potentiell durchaus zu realisieren wäre. Ohne Ausnahme wird lediglich ein Benutzungsschnittstellentyp (WIMP) unterstützt und es sind keine Mechanismen vorgesehen, um andere Modalitäten mit wenig Aufwand in die Systeme einzufügen. Die Erweiterbarkeit in Hinsicht auf Anbindungstechniken zu verschiedenen Realisierungen der

	Separation Connection	Simplification	Extensibility UI / Connection	Flexibility UI / Impl.	Adaptation	Komposition
Frameworks	x/x	x	o/(x)	o/o	o	o
Modelbased Systems	x/(x)	o	(x)/(x)	(x)/(x)	(x)	o

x : Nearly all systems have the property
(x): Only some systems have the property
o: No tested system has the property

Abbildung 3.20: Subsumierte UbiComp-Eigenschaften der Systemgruppen

Fachlogik wird von einigen Systemen wie MVP oder SanFrancisco unterstützt. MVP realisiert sogar ein umfassendes Datenmanagementkonzept zur einfachen Integration von Fachlogik mit persistent zu haltenden Daten. Für alle bisher angesprochenen Aspekte war die Nähe zur Implementationsschicht ein grosser Vorteil. Anders verhält es sich bei der Betrachtung der Flexibilität. Frameworks sind nicht in der Lage Benutzungsschnittstellen mit einer Applikation so lose zu koppeln, dass problemlos Oberflächen ausgetauscht werden können oder mehrere Oberflächen für eine Applikation spezifiziert werden können. Selbst wenn es ein Framework mit diversen Interfacemodalitäten gäbe, muss programmiert werden, um die Verbindung zur Fachlogik herzustellen. Auch die vielleicht weniger bedeutende Flexibilität hinsichtlich des Austauschs von Implementationen für ein User Interface unterliegt denselben Einschränkungen. Die Aspekte der Interfaceadaption und Komposition werden von keinem Framework berücksichtigt.

Die modellbasierten Systeme erreichen die Trennung von User Interface und Fachlogik entweder durch die Einführung expliziter Modelle zur Beschreibung der Oberfläche oder durch die automatische Generierung der Benutzungsoberfläche aus Domänenmodellen. Die Verbindung zum Applikationskern wird von den meisten Systemen hergestellt, lediglich TRIDENT und TADEUS beschränken sich auf die Erzeugung einer Oberfläche, die durch Programmieraufwand noch mit der Fachlogik verknüpft werden muss. Die übrigen Systeme setzen unterschiedliche Kopplungsstrategien ein. Janus und BC-Prototyper generieren das User Interface durch Evaluierung von Modellinformationen. Damit benötigen diese Systeme keinen Mechanismus, um Modelle in Beziehung zu setzen. Das Teallach und das MASTERMIND System verwenden eine neu definierte Expressionlanguage zur Verbindung von Modellen, wohingegen das Mobi-D System ein eigenes Modell einführt, das Informationen über die Relationen der Modelle untereinander aufnimmt. Der Aspekt der vereinfachten User Interface Entwicklung wird derzeit von modellbasierten Systemen aufgrund fehlender Standards in den Bereichen der User Interface Metamodellkonzeption nicht erreicht. Die Erweiterbarkeit modellbasierter Systeme bezüglich der Einführung neuer Interfacemodalitäten ist bei vielen Systemen machbar, wird aber nur vom Teallach System explizit angesprochen. Der Einsatz verschiedener Anbindungstechniken zur Verknüpfung eines Modells mit verschiedenen Arten von Implementationsschichten wird ebenfalls von wenigen Systemen unterstützt. Lediglich Janus und Teallach enthalten Metamodellkonzepte zum Umgang mit Daten, die persistent gemacht werden müssen. Es lässt sich feststellen, dass modellbasierte Systeme für die flexible Austauschbarkeit von User Interfaces prinzipiell prädestiniert

sind, jedoch kein System Applikationen mit verschiedenen Benutzungsschnittstellen vorstellt. Die Autoren halten es für sehr wahrscheinlich, dass zumindest die Systeme mit expliziten User Interface Modellen in der Lage sind, Benutzungsoberflächen verschiedenen Typs für eine Applikation bereitzustellen. Des weiteren sollte es ebenfalls möglich sein, Implementationen eines Domänenmodells unter Berücksichtigung gewisser Randbedingungen, wie etwa der Typkompatibilität, auszutauschen. Die Adaption von User Interfaces wird ausschließlich vom MASTERMIND System berücksichtigt. Es ist in der Lage, User Interfaces je nach den Ausmaßen des vorhandenen Endgeräts zu verändern und Details bei geringen Auflösungen auszublenden (Interface-Tailoring, siehe [Szekely 1996]). Von keinem vorgestellten System werden die Problematiken der Komposition von User Interfaces adressiert.

Aus diesem Vergleich lässt sich schlussfolgern, dass die nicht-deklarativen Ansätze aufgrund der konzeptionell bedingten mangelhaften Flexibilität bezüglich der Verwendung von verschiedenen User Interfaces für eine Applikation für das UbiComp weniger geeignet erscheinen. Modellbasierte Systeme überwinden dieses Problem und besitzen alle Eigenschaften, die sie für das UbiComp qualifizieren. Innerhalb der modellbasierten Systeme erweisen sich die interpreterbasierten Systeme als besonders interessant, da sie den vom UbiComp geforderten Anforderungen hinsichtlich sich dynamisch verändernder User Interfaces Rechnung tragen können. Generatorbasierte Systeme erscheinen vor diesem Hintergrund als zu unflexibel. Bevor modellbasierte Systeme jedoch im Kontext des UbiComp eingesetzt werden können, müssen noch einige Problemstellungen überwunden werden. Dazu gehört in erster Linie die Nichtexistenz von etablierten Modellierungsstandards für User Interfaces, die praktische Unterstützung von mehreren User Interfacemodalitäten und die Forschung im Bereich von Adaption und Komposition von Benutzungsschnittstellen.

Kapitel 4

Vesuf Konzeption

In diesem Kapitel werden die Autoren das neu konzipierte und realisierte User Interface Entwicklungssystem Vesuf vorstellen. Unter Berücksichtigung der Ergebnisse von Kapitel 2 und 3 haben sie ein modellbasiertes System entworfen, das geeignete Voraussetzungen für die Erreichung der in Abschnitt 1.3 formulierten Forderungen aufweist.

Die Autoren werden zunächst kurz die Konzeption des Vesuf Systems darlegen, und auch darauf eingehen, welche Fähigkeiten das Vesuf System besitzt und inwieweit sich die Verwendung des Systems vorteilhaft auswirkt (siehe Abschnitt 4.1). Anschliessend werden sie die einzelnen Schichten des Systems näher beleuchten und interessante Aspekte anhand eines anschaulichen Beispiels im Detail erklären (siehe Abschnitt 4.2). Es folgt eine Darstellung der konzeptuellen Probleme (siehe Abschnitt 4.3) und am Ende des Kapitels werden einige Erweiterungsmöglichkeiten für das System angesprochen, welche entweder die Anwendbarkeit des Systems erhöhen sollen oder noch nicht erreichte Grundvoraussetzungen für das UbiComp adressieren (siehe Abschnitt 4.4).

4.1 Überblick

Bei Vesuf handelt es sich um eine von den Autoren im Rahmen dieser Arbeit neu entwickelte modellbasierte User Interface Development Environment (MB-UIDE).[1] Sie ermöglicht die Erzeugung von ablauffähigen und vollständig integrierten User Interfaces durch einen Interpreter, der zur Laufzeit Modellinformationen evaluiert. Das System basiert konsequent auf Standards und verwendet zur Modellierung fast ausschließlich den de-facto Industriestandard zur Anwendungsspezifikation – UML [OMG 2000a].

Die Verwendung eines MB-UIDEs geht naturgemäß mit einem erhöhten Spezifikationsaufwand einher. Nachfolgend werden die Vorteile der Verwendung des Vesuf Systems vorgestellt (siehe dazu auch Abschnitt 1.3):

- User Interface und Anwendungskern sind klar getrennt. Die Entwicklung der beiden Teile kann separat durchgeführt werden. Die Fachlogik wird durch die Implementation und das Domänenmodell beschrieben und das

[1] Das Vesuf System ist ein Opensource Projekt und kann über folgende Adresse bezogen werden: `http://sourceforge.net/projects/vesuf`

User Interface wird vollständig durch Modelle spezifiziert. Über einen Pfadmechanismus werden die unterschiedlichen Teile ohne Implementationsaufwand in einfacher Weise verbunden.

- Da das Vesuf System auf UML-Semantik beruht, ist die Lernschwelle für Erstanwender niedrig und die Entwicklung von User Interfaces wird mit dem System nicht wesentlich komplizierter gemacht. Zusätzlich schafft das System durch die Automatisierung von Aufgaben und durch die vollständige Integration von User Interface und Fachlogik die Grundlage für schnelles Prototyping. Der Codeumfang dieser Prototypen wird erheblich reduziert, da viele Aspekte bereits durch Systemfunktionalität abgedeckt sind, z. B. die Behandlung von Benutzerinteraktionen und die Einbeziehung von Constraints.

- Das Vesuf System ist sowohl im Hinblick auf User Interface Modalitäten als auch bezüglich der Anbindung verschiedener Implementationstypen (respektive Altsysteme) erweiterbar. Für das Vesuf System wurden bereits zwei Interpretermodi (aktiv vs. passiv) entwickelt, die einserseits interaktive Modalitäten wie z. B. AWT und Swing und andererseits anfrageorientierte Modalitäten wie z. B. HTML, WML und VXML unterstützen. Die Ergänzungen, die zur Integration von neuen User Interface Modalitäten vorgenommen werden müssen, sind auf die Präsentationsschicht von Vesuf begrenzt. In ähnlicher Form sind auch die Erweiterungen zur Einbeziehung von neuen Anbindungstechniken lokal begrenzt.

- Das Vesuf System ermöglicht es, verschiedene Arten von User Interfaces auf einfache Weise für einen Anwendungskern bereitzustellen. Außerdem können auch verschiedene schnittstellenkompatible Anwendungskerne für eine Präsentation entwickelt werden.

- Die Philosophie des Vesuf Ansatzes ist es, die Grenze zwischen expliziter Interface Spezifikation und automatischer Extrapolation variabel zu halten. Die Autoren stimmen mit [Szekely 1996] darin überein, dass eine vollständig automatisierte Oberflächenerzeugung qualitativ nicht ausreichend sein kann. Vesuf unterstützt daher ausgehend von unvollständigen oder nicht vorhandenen Interfacemodellen, die iterative Ergänzung und Verfeinerung der Modellinformationen. Die Autoren nennen diese Art der Systementwicklung *slinky automation*.

4.1.1 Modelle und Notationen

In Abb. 4.1 sind Komponenten des Vesuf Systems dargestellt. Um eine Applikation mit dem Vesuf System zu realisieren, muss der fachliche Teil der Anwendung implementiert werden (application implementation). Zusätzlich müssen Modelle für das User Inteface spezifiziert werden (ui specification). Um die Spezifikation der einzelnen Modelle zu vereinfachen, können verschiedene Entwurfswerkzeuge (design-time tools) eingesetzt werden. Zur Laufzeit (runtime level) werden die Modellinformationen von einem Interpreter unter Zuhilfenahme von Automationswerkzeugen (automation tools) ausgewertet. Er erzeugt daraus unter Verwendung von Systemdiensten (system features) eine auführbare Benutzungsoberfläche.

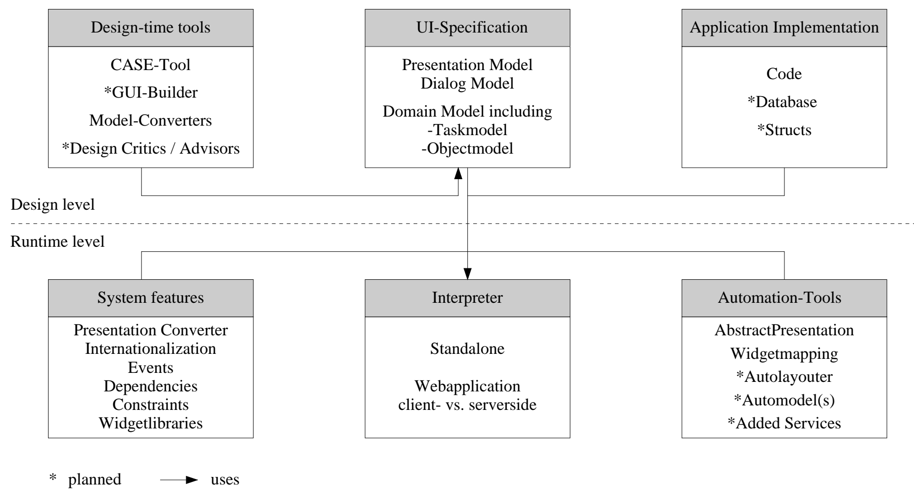


Abbildung 4.1: Vesuf Architektur

Vesuf verwendet zur Spezifikation eines User Interfaces zur Zeit vier Modelle: ein Objektmodell (object model), ein Aufgabenmodell (task model), ein Dialogmodell (dialog model) und ein Präsentationsmodell (presentation model) (siehe Abb. 4.1). Im Objektmodell werden die für die Oberfläche relevanten Entitäten der Problemdomäne definiert. Dazu gehören neben einer Beschreibung der Entitäten selbst durch ihre Eigenschaften und Methoden auch ihre Beziehungen zu anderen Elementen. Grundlage des Objektmodells und seiner Notation ist das UML Klassendiagramm.

Das Aufgabenmodell dient zur Beschreibung von Aufgaben, die ein Benutzer durch Interaktion mit dem System erledigen möchte. Zur Zeit wird im Aufgabenmodell lediglich die Struktur einzelner Aufgaben, jedoch nicht ihre temporale Ordnung beschrieben. Daher adaptiert das Aufgabenmodell in der aktuellen Version direkt die Semantik der UML Use Cases. Eine UML-konforme Erweiterung dieses Modells wird angestrebt. Aufgaben- und Objektmodell werden unter dem Begriff Domänenmodell zusammengefasst.

Mit Hilfe des Dialogmodells wird in Vesuf die grobe Dialogsteuerung (siehe Abschnitt 2.2.3) spezifiziert, d. h. dieses Modell ist verantwortlich für die Abfolgesteuerung der Sichten des Systems. Grundlage für die Beschreibung ist die UML Statechart Semantik, die um das Konzept der Stateobjects[2] erweitert wurde (siehe Abschnitt 4.2.4).

Das Präsentationsmodell dient zur Beschreibung einzelner Dialoge. Es wird dabei nicht explizit zwischen einem abstrakten und einem konkreten Modell unterschieden, da ein abstraktes Modell durch die Angabe zusätzlicher Informationen stets konkretisiert werden kann. UML stellt zur Zeit noch kein Metamodell für die Präsentation und keine User Interface Beschreibungssprache bereit. Die Autoren verwenden daher ein eigenes Metamodell (siehe Abschnitt 4.2.5), welches mit UIML (siehe Abschnitt 5) beschrieben wird. Um eine übersichtliche Darstellung des Präsentationsmodells zu erreichen, kann zur Notation des

[2]Unter Stateobjects werden Objekte verstanden, die direkt mit einem Zustand assoziiert sind. Um Objekte zwischen unterschiedlichen Zuständen zu verschieben, werden interne Datenflüsse benötigt.

Modells auch UIML-Shorthand benutzt werden.

In einem Applikationsdeskriptor fließen schließlich alle Informationen zusammen, die für eine ausführbare Oberfläche benötigt werden. Neben den verwendeten Submodellen werden an dieser Stelle Einzelheiten der Modellintegration festgelegt (siehe Abschnitt 4.2.2.1).

4.1.2 Werkzeuge der Vesuf Umgebung

Die Werkzeuge der Vesuf Umgebung lassen sich in zwei Kategorien einteilen: Design-time und Runtime-Tools. In die Gruppe der Design-time Tools fallen alle Werkzeuge, die den Entwickler bei der Spezifikation der Modelle unterstützen. Mit Hilfe jedes Standard UML CASE Tools lassen sich die Domänenmodelle und das Dialogmodell editieren. Einzige Bedingungen für die Verwendung eines bestimmten Werkzeuges sind ein standardkonformer XMI-Export und die Möglichkeit, die Standard Erweiterungsmechanismen von UML[3] zu nutzen. Die nach XMI exportieren Modelle werden dem Vesuf System durch Konverter zugänglich gemacht. Ein Konverter liest ein XMI-konformes Modell und erzeugt daraus ein kompatibles Vesufmodell. Ist kein UML-CASE Tool verfügbar oder die Spezifikation mit einem UML-Werkzeug nicht erwünscht, können Domänen- und Dialogmodell auch direkt als Java-Sourcefile spezifiziert werden. In einem Java-Sourcefile kann ein Modell unter Verwendung der Konstruktoren der Metamodellelemente definiert werden. Um ein Präsentationsmodell für das Vesuf System zu erstellen, kann entweder ein UIML-konformes File mit einem Texteditor beschrieben werden oder wiederum auf die Spezifikation mittels eines Java-Sourcefiles zurückgegriffen werden.

In die zweite Kategorie fallen Werkzeuge, die zur Laufzeit verwendet werden. Die Autoren fassen diese Werkzeuge unter dem Oberbegriff Automatisierungswerkzeuge (automation tools) zusammen, da sie alle einen bestimmten, nicht in den Modellen enthaltenen Aspekt der Applikation aus vorhandenem Wissen inferieren. Basis dieser Schlussfolgerungen sind einerseits in den Modellen vorhandene Informationen und andererseits vorgegebene Regeln und Wissensbasen. Geht man beispielsweise davon aus, dass kein Präsentationsmodell spezifiziert wurde, verwendet das System zunächst ein Werkzeug, welches die Inhalte einer darzustellenden Domänenentität bestimmt (abstract presentation tool). Im Anschluss daran können die Oberflächenbausteine und ihre Anordnung errechnet werden (widgetmapping, auto layout). Resultat ist eine ablauffähige Oberfläche.

Die Autoren schlagen vor, den Gedanken der automatischen Generierung nicht vollständig aufzugeben, sondern den Grad des automatischen Designs flexibel zu halten und damit eine iterative Entwicklung, ausgehend von einem Minimum an Spezifikation zu ermöglichen. Je weiter der Entwicklungsprozess voranschreitet, desto größer werden die Anteile manuell spezifizierter Dialoge.

Weitere Werkzeuge, die nicht Bestandteil des Vesuf Systems sind, wie z. B. Automodeltools und kontextabhängige Hilfesysteme, werden in Abschnitt 4.4 näher erläutert.

[3] Die Erweiterungsmechanismen von UML sind Stereotypes, Constraints und Taggedvalues.

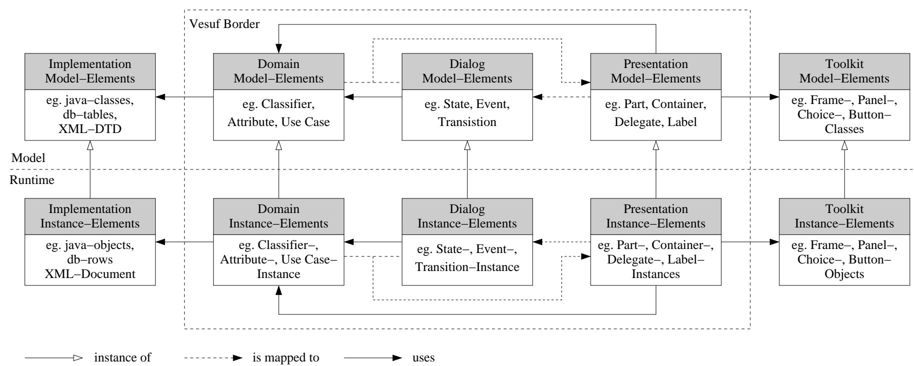


Abbildung 4.2: Laufzeitarchitektur

4.1.3 Laufzeitarchitektur

In Abb. 4.2 ist die Laufzeitarchitektur einer Vesuf Applikation dargestellt. Grundlegendes Konzept dieser Architektur ist die strenge Separierung zwischen Modell- und Laufzeitentitäten in allen Schichten. [Kent et al. 1999] schlagen vor, diese Trennung zwischen den beiden Ebenen als Basis des UML-Metamodells einzuführen.[4] Modellelemente (ModelElements) der Vesuf Umgebung werden eingesetzt, um die verschiedenen Modelle einer Anwendung zu beschreiben, wohingegen Laufzeitelemente (InstanceElements) konkrete Ausprägungen solcher Modellelemente darstellen. Um sich die Relation der Elemente einfach vorzustellen, kann man sich die Trennung innerhalb der Implementationsschicht vor Augen führen. Wird z. B. Java verwendet, gibt es Java-Klassen (die Modellelemente) und Objekte (die Laufzeitelemente). Bei der Nutzung von XML findet sich die Grammatik bzw. DTD (deklariert Modellelemente) und das XML-Dokument selbst (enthält Laufzeitelemente). Es lässt sich ebenfalls feststellen, dass das Vesuf System dieselben funktionalen Komponenten identifiziert wie das in Abschnitt 2.1.4 vorgestellte Arch Schichtenmodell. So entspricht die Vesuf Implementationsschicht der domänenspezifischen Komponente des Arch Modells genau wie die Toolkit-Schicht auf die Interaktionstoolkit Komponente abbildbar ist. Zwischen diesen Schichten befinden sich die Komponenten, die direkt dem Vesuf System zuzuordnen sind (durch den Rahmen in Abb. 4.2 angedeutet). Wie im Arch Modell stellen zwei Adapter die Unabhängigkeit des Systems von der Implementations- und Toolkitseite sicher. Vesuf führt dazu eine Domänenschicht als Arch Domänenadapterkomponente und eine Präsentationsschicht als Arch Präsentationskomponente ein. Des weiteren besitzt Vesuf eine Dialogkomponente zur Sichtensteuerung des Systems, welche sich unter dem Namen Dialogsteuerung im Arch Modell wiederfindet.

Mit der Modellierung des Domänenmodells werden Domänenmodellentitäten aus UML wie z. B. Classifier, Attribute, Use Case Elemente der Implementationsschicht eindeutig zugeordnet. Um die Abbildung zu definieren, werden den UML-Elementen TaggedValues mit der exakten Bezeichnung des Implementationselements hinzugefügt, z. B. erhält eine UML-Klasse, die die Java-Klasse Point repräsentieren soll, einen TaggedValue mit `tag=implclass` und `va-`

[4] Kent et al. bezeichnen die Komponenten der Modellebene als Deskriptorelemente und die Komponenten der Laufzeit als Instanzelemente.

`lue=java.awt.Point`. Zur Laufzeit des Programms werden Point-Objekte durch die Laufzeitumgebung (UML-Runtime) mit UML Klasseninstanzen der UML Klasse für Point assoziiert.

Hat ein Entwickler die Sichtenabfolge einer Anwendung in Form eines Zustandsdiagramms spezifiziert, muss zusätzlich noch festgelegt werden, welches Domänenelement in einem Zustand dargestellt werden soll. Im Normalfall werden einzelne Aufgaben des Aufgabenmodells in einem Dialog visualisiert, in manchen Fällen macht es aber auch Sinn, eine Entität des Objektmodells direkt darzustellen. Die Verbindung eines Zustandes mit einem Domänenmodellelement wird durch die Angabe eines Pfades hergestellt (siehe Abschnitt 4.2.2.2). Wird die Anwendung ausgeführt, können Pfadinstanzen ausgewertet werden, um die aktuelle Domäneninstanz zu erhalten.

Neben der Information, welche Entität dem Benutzer präsentiert werden soll, ist es auch notwendig festzulegen, wie dieses Element zu rendern ist. Diese Informationen sind im Präsentationsmodell abgelegt und müssen mit dem aktuellen Dialogmodellzustand verknüpft werden. Die Autoren verwenden dazu keine starre Verbindung, sondern ein flexibles Mapping auf Modellebene, das als Eingabe Dialogzustand und Domänenmodellelement erhält und als Ergebnis die am Besten passende Sicht liefert. Dieses Mapping realisiert dadurch eine Funktion, die ein Paar aus Dialogzustand und Domänenelement auf ein Präsentationselement abbildet. Es ist dabei nicht immer ausreichend, nur den Dialogzustand als Eingabe zu verwenden, da zur Laufzeit auch verfeinerte Domänenentitäten in einem Dialogzustand auftauchen können. Für diese Elemente können so andere Sichten zurückgeliefert werden.

Einem Dialog ist damit eine Domänenentität zugeordnet. Um eine Integration von User Interface und Applikation zu erreichen, muss auch eine Konnexion bestimmter Präsentationselemente des Dialogs mit Objektmodellelementen hergestellt werden. Dazu können Pfade als Eigenschaften der Oberflächenelemente spezifiziert werden. Ausgangspunkt dieser Pfade ist das Domänenobjekt des zugrundeliegenden Dialogs. Eine weitere Eigenschaft jedes konkreten Präsentationselements ist sein Widgettyp, der die Verbindung mit einem speziellen Toolkitelement arrangiert. Außer Interaktionselementen, die mit einer Entität aus dem Objektmodell assoziiert sind, gibt es auch reine Präsentationselemente, die ohne Bezug auskommen (z. B. Rahmen) und solche, die für die Navigation von Belang sind (z. B. next Button). Aus diesem Grund besteht auch eine Abbildungsverbindung zwischen Präsentations- und Dialogelementen.

4.2 Systemdetails

Nachfolgend werden die einzelnen Schichten des Vesuf Systems detailliert vorgestellt. In Abschnitt 4.2.3 wird die Domänenschicht, in Abschnitt 4.2.4 die Dialogschicht und in Abschnitt 4.2.5 die Präsentationsschicht beschrieben. Zuvor wird in Abschnitt 4.2.2 aufgezeigt, in welcher Form die Systemkomponenten zu einer vollständigen Applikation zusammengeführt werden. Spezielle Konzepte werden anhand der im nächsten Abschnitt vorgestellten Beispielanwendung erläutert.

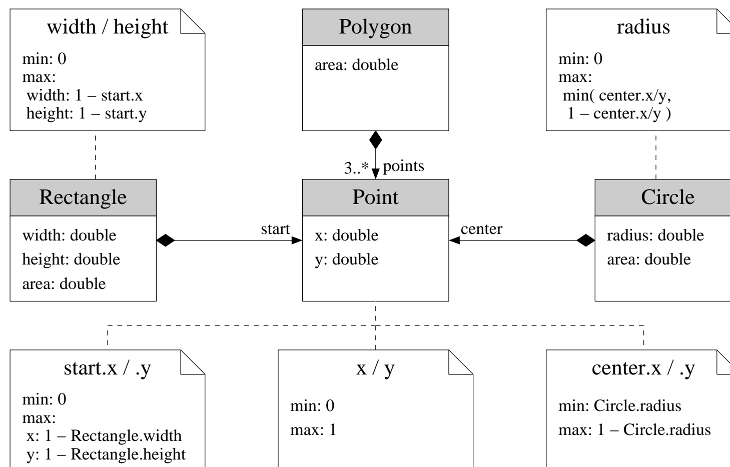


Abbildung 4.3: Beispielmodell (Shape)

4.2.1 Beispielanwendung

Abbildung 4.3 zeigt ein einfaches Objektmodell, auf das die Autoren in den folgenden Abschnitten zurückgreifen wollen, um die komplexen Zusammenhänge der Vesuf Internas anschaulich darzustellen. Das Beispielmodell enthält vier Klassen, die geometrische Objekte (shapes) repräsentieren. Ein Punkt (point) hat eine X- und eine Y-Koordinate. Ein Rechteck (rectangle) hat einen Startpunkt, sowie Breite (width) und Höhe (height). Ein Kreis (circle) wird durch den Mittelpunkt (center) und den Radius bestimmt. Ein Polygon besteht aus beliebig vielen, jedoch mindestens drei Punkten. Kreis, Rechteck und Polygon besitzen zusätzlich ein Attribut, dass den Flächeninhalt (area) enthält. Dieses kann nur abgefragt, aber nicht gesetzt werden, da es sich aus anderen Attributen (z. B. Radius) ergibt. In diesem Modell sollen sämtliche geometrischen Objekte im Bereich (0, 0) – (1, 1) liegen. Drei Klassen, sind daher mit Einschränkungen (Constraints) versehen. Diese gelten für die X- und Y-Koordinaten eines Punktes, für Breite und Höhe eines Rechteckes sowie für den Radius eines Kreises. Je nachdem ob ein Punkt alleine steht oder z. B. Mittelpunkt eines Kreises ist, gelten für ihn unterschiedliche Beschränkungen.

4.2.2 Kopplung der Systemkomponenten

Dieser Abschnitt beschreibt die Kopplungsmechanismen von Vesuf, die sowohl auf Modellebene als auch zur Laufzeit eine Rolle spielen. Eine Vesuf Anwendung besteht stets aus den in Abschnitt 4.1.3 vorgestellten Schichten. Es ist also notwendig, Spezifikationen des Domänen-, des Dialog und des Präsentationsmodells zu erstellen und die entsprechende Fachlogik zu entwickeln. Da es möglich ist, verschiedene Modelle für eine Applikation zu entwerfen (z. B. verschiedene Präsentationsmodelle für verschiedene Interfacemodalitäten), muss festgelegt werden, welche Modelle von einer speziellen Anwendung benutzt werden sollen. Dafür wurde in Vesuf der sogenannte Applikationsdeskriptor (siehe Abschnitt 4.2.2.1) vorgesehen. Um einzelne Modellelemente über Modellgrenzen hinweg zu referenzieren, wurde ein Pfadmechanismus eingeführt (siehe Abschnitt 4.2.2.2).

Während der Programmausführung können die statisch spezifizierten Pfade dynamisch aufgelöst werden. Zur Laufzeit einer Anwendung werden die Modelle von einem Interpreter ausgewertet (siehe Abschnitt 4.2.2.3) und erzeugen ein User Interface. Um Benutzerinteraktionen und sich ändernde Systemzustände im System zu reflektieren, wurde in Vesuf ein globaler Ereignismechanismus integriert (siehe Abschnitt 4.2.2.4).

4.2.2.1 Applikationsdeskriptor

Der Applikationsdeskriptor dient zur Bündelung aller für eine ausführbare Applikation notwendigen Einstellungen. In Abb. 4.4 wird anhand des Shape Deskriptors exemplarisch dargestellt, welche Informationen in einem Applikationsdeskriptor vorkommen.

Neben dem Namen des Applikationsdeskriptors (Zeile 4) müssen die zu verwendenden Modelle angegeben werden. Im Beispiel werden das Domänen- und das Präsentationsmodell auf die speziell für die Shape-Applikation erstellten Submodelle `shape.ShapeObjectModel` (Zeile 7) und `shape.ShapePresentationModel_awt` (Zeile 9) gesetzt. Da keine spezielle Dialognavigation benötigt wird, reicht die Festlegung des Dialogmodells auf das bereits in Vesuf spezifizierte `DefaultDialogModel` (Zeile 8) aus.

Um zu bestimmen, wie eine Applikation gestartet wird, muss definiert werden, welche Dialogmaschine mit der Ausführung beginnt und mit welchem initialen Objekt sie konfiguriert wird. Im Beispiel kommt die `DefaultDialogMachine` (Zeile 12) zum Einsatz. Sie wird mit einer SampleTask Instanz gestartet, die durch die Pfadangabe `<static>(SampleTask).<create>().<value>()` (Zeile 13) beschrieben ist.

Schliesslich ist es noch notwendig zu definieren, welche Präsentationskomponente die Abbildung von Domänen- auf Präsentationselemente vornimmt. In diesem Fall wird eine Standardkomponente verwendet (Zeile 16), welche bei der Abbildung die im `shape_mapping.properties` File definierten Abhängigkeiten umsetzt (Zeile 17).

Optional kann zusätzlich noch angegeben werden, welche Datei Informationen über die Abbildung von abstrakten zu konkreten Interaktionselementen enthält. Das Shape-Beispiel verwendet dazu die bereits in Vesuf konstituierten Abbildungsregeln für AWT (Zeile 18), so dass diese Angabe nur der Vollständigkeit halber im Shape Deskriptor aufgenommen wurde.

4.2.2.2 Pfade

Ein Pfad (path) beschreibt eine Route von einem Startpunkt bis zu einem bestimmten Zielelement. Sowohl der Startpunkt als auch das Ziel sind Modellelemente eines beliebigen Modells. Ein Pfad setzt sich aus einzelnen Pfadelementen (path elements) zusammen, welche jeweils die Navigation über eine Referenz des (erweiterten) UML-Metamodelles repräsentieren. Ein Pfad zwischen zwei Elementen ist damit konstruierbar, wenn das Zielelement vom Startpunkt aus über den in der VEsuf Path Language (VEPL) realisierten Ausschnitt des Metamodells erreichbar ist.

Zur Laufzeit werden Pfadinstanzen der Pfade mit einer Startentität erzeugt. Sie können dynamisch ausgehend von der Startentität ausgewertet werden und liefern so stets das aktuelle Zielelement eines Pfades. Verändern sich Teile des

```
01:  # Properties for the shape application in awt
02:
03:  # Name of the application descriptor:
04:  name                = shape.ShapeApplicationModel_awt
05:
06:  # Model specifications:
07:  domainmodel         = shape.ShapeObjectModel
08:  dialogmodel         = org.vesuf.model.navigation.DefaultDialogModel
09:  presentationmodel   = shape.ShapePresentationModel_awt
10:
11:  # Dialog properties:
12:  dialogmachine       = DefaultDialogMachine
13:  initobject          = <static>(SampleTask).<create>().<value>()
14:
15:  # Presentation properties:
16:  presentationmapper = org.vesuf.model.application.PropertyPresentationMapper
17:  mapper_properties   = /shape/shape_mapping.properties
18:  cio_mapping         = /org/vesuf/presentation/awt/mapping_awt.properties
```

Abbildung 4.4: ApplicationDescriptor des Shape-Beispiels

für eine Pfadinstanz relevanten Objektgraphens in der Implementation, wird dies bei der nächsten Auswertung reflektiert.

Um die Modellierung von Pfaden in Vesuf zu erlauben, wurde eine textuelle Beschreibungsform für Pfade und die notwendigen Metamodellelemente in das System integriert.

Die Einführung des Pfadkonzepts in die Modellierung führt zu einigen Vorteilen, da sie universell in verschiedenen Teilbereichen eingesetzt werden können, und weiterführende Konzepte Pfade als natürliche Grundlage aufgreifen. Durch Pfade bietet sich die Möglichkeit der exakten Spezifikation von Verbindungen zwischen beliebigen Modellelementen. Z. B. werden im Präsentationsmodell Pfade benutzt, um bestimmte Präsentationselemente mit Elementen der Domänenschicht zu verbinden. Im Gegensatz zur direkten Spezifikation des Modellelements kann so das Präsentationselement auf einfache Weise auch auf Änderungen von im Pfad enthaltenen Elementen reagieren. Des weiteren basiert das Konzept der Abhängigkeiten (dependencies) auf einer Pfadangabe zwischen einem abhängigen Element und der Quelle der Abhängigkeit. Diese natürliche Art der Spezifikation von Abhängigkeiten führt dazu, dass zur Laufzeit eines Systems die Abhängigkeit stets die richtigen Elemente betrifft, da Änderungen des Objektgraphens durch die Pfadinstanz auch in einer angepassten Abhängigkeitsbeziehung resultieren.

Die Spezifikation von Pfaden in VEPL Die textuelle Beschreibung von Pfaden im Vesuf System erfolgt in der VEsuf Path Language (VEPL), deren Grammatik auszugsweise in Abbildung 4.5 dargestellt ist. Die Autoren haben uns an dieser Stelle aus Gründen der Übersichtlichkeit darauf beschränkt, den Teil der EBNF-Spezifikation[5] darzustellen, der für das Verständnis der nach-

[5]Die Backus-Naur Form [ISO/IEC 1996] ermöglicht eine formale mathematische Beschreibung einer Sprache und wurde von John Backus und Peter Naur eingesetzt, um die Syntax der Algol 60 Programmiersprache zu definieren. Sie wird verwendet, um die Grammatik einer Sprache formal festzulegen und bestimmt dadurch eindeutig ob ein Ausdruck zur Sprache gehört oder nicht. Die BNF basiert auf mathematischen Grundlagen, die es sogar erlauben, automatisiert Parser für eine in BNF spezifizierte Sprache zu entwickeln. Die Extended Backus Naur Form erweitert die BNF um Operatoren, welche die Spezifikation vereinfachen.

```
01: # Initial path elements
02: path = classifier_ref | typedelement_ref | operation_ref | static | ...
03:
04: # References
05: classifier_ref   = attribute | operation | composite | create | static
06: typedelement_ref = attr_value | owner | constraint | static
07: operation_ref    = op_value | owner | constraint | static
08:
09: # Path elements
10: # type_name: A name of a model classifier (fully qualified).
11: # name: Simple name of a modelelement.
12: attribute  = "<attribute>" name ("." typedelement_ref)?
13: attr_value = "<value>" ("." classifier_ref)?
14: op_value   = "<value>(" values ")" ("." classifier_ref)?
15: static     = "<static>" type_name ("." classifier_ref)?
16: create     = "<create>(" parameters? ")" ("." operation_ref)?
17: ...
18: values     = value (", " value)*
19: value      = literal | path
20: parameters = name ":" type_name (", " parameters)?
21:
22: # Java literals
23: # classname: Name of a Java class.
24: # java_constant: Is a static field of a java class.
25: # java_literal: Is a string description, used as parameter
26: # for the constructor of the required class.
27: literal             = constructor_literal | constant_literal
28: constructor_literal = classname"(" ( lit_parameter(","lit_parameter)*)? ")"
29: constant_literal    = java_constant | java_literal | "null"
30: lit_parameter       = (classname":")? literal
```

Abbildung 4.5: Ausschnitt der VEPL Spezifikation in EBNF

folgenden Beispiele notwendig ist. Die vollständige Spezifikation findet sich in Anhang A, Abbildung A.1.

Die Autoren haben die Grammatik in vier unterschiedliche Bereiche eingeteilt. Ausgangspunkt für die Pfadbeschreibung ist das `path` Element (Zeile 2), das zwei grundsätzlich verschiedene Pfaddeklarationen zulässt. Es ist zulässig, einen Pfad entweder von einem Modellelement aus zu beschreiben (`*_ref`) oder ohne Berücksichtigung eines Startpunktes statisch zu beginnen (`static`).

In der zweiten Sektion (Zeilen 4-7) sind Elementreferenzen aufgeführt. Sie drücken aus, welche Elemente erreichbar sind, wenn der aktuelle Bezugspunkt das im Namen enthaltene Element ist. So ist z. B. in `cassifier_ref` enkodiert, welche Ziele von einem Classifier aus erreichbar sind.

Der dritte Bereich (Zeilen 9-20) enthält die Beschreibungen der Metamodellelemente selbst und einige Hilfskonstrukte (Zeilen 18-20). Sämtliche Metamodelldeskriptoren beginnen mit einem Schlüsselwort, das zu ihrer Identifikation vorgesehen ist, z. B. `<attribute>` und schließen mit der Option, den Pfad fortsetzen zu können, z. B. `("." typedelement_ref)?`. Wie hier zu erkennen ist, werden einzelne Pfadelemente durch einen Punkt voneinander abgesetzt.

Im vierten Abschnitt (Zeilen 22-30) ist definiert, wie in Vesuf Java-Literale eingesetzt werden können.[6] Die Autoren unterscheiden zwei Arten von Java-

[6]Die Autoren bezeichnen diese Art der Ausdrücke als Java-Literale, da für das Vesuf System lediglich die zu einem Wert evaluierten Ausdrücke von Belang sind. Um die in VEPL spezifierten Java-Literale auszuwerten, verwendet das Systeme eine VEPL-externe Ressource, die auch in weiteren Bereichen zur Konvertierung eingesetzt wird. Die Auswertung der Java-

Literalen: Literale, die durch den Aufruf eines Java-Konstruktors erstellt werden (`constructor_literal`) und Literale, die sich auf ein statisches Feld innerhalb einer Java-Klasse beziehen (`constant_literal`). Ein Konstruktorliteral setzt sich aus dem Klassennamen der zu erzeugenden Entität und einer optionalen Parameterliste zusammen. Ein Parameter ist wiederum ein Java-Literal, dem optional sein Typ vorangestellt werden kann. Diese Angabe ist dann unnötig, wenn die Anzahl der Parameter und übrigen Typangaben für eine eindeutige Konstruktorauswahl ausreichend sind. Ein Konstantenliteral dient zur Spezifikation eines konstanten Wertes, der entweder durch den Namen referenziert (z. B. `java.awt.Color.lightGray`)[7] oder direkt durch eine Zeichenkette beschrieben wird (z. B. Zahlenwerte wie `0.5`). Des weiteren kann die java-spezifische Null-Referenz definiert werden.

Einzelheiten werden nachfolgend anhand dieser zwei Beispiele erläutert:

(a) `<attribute>Startpoint.<value>.<attribute>X.<value>`

(b) `<static>(Point).<create>(x:double, y:double).<value>(0.5, 0.5)`

Im ersten Fall handelt es sich um einen Pfad, der eine Navigation von einem Rechteck zu der X-Koordinate seines Startpunktes realisiert. Da die Basis für den Pfad das Rechteck ist, beginnen man mit einer Classifier-Referenz (`classifier_ref`). Diese Referenz erlaubt es, ein Attribut des Classifiers auszuwählen (`<attribute>Startpoint`). Vom Attribut Startpunkt muss man nun zunächst zum eigentlichen Startpunkt navigieren (`typedelement_ref`), der den Wert des Attributes darstellt (`<value>`). Das Vorgehen um vom Startpunkt zum Wert der X-Koordinate zu gelangen, entspricht exakt den eben beschrieben Schritten vom Rechteck zum Startpunkt. Es ist daher notwendig, zunächst das Attribut der X-Koordinate (`<attribute>X`) des Startpunktes und schliesslich seinen Wert (`<value>`) über die bereits angeführten Produktionsregeln zu referenzieren. Um die Ableitung des Pfades in Einzelschritten zu verfolgen, haben die Autoren eine Navigationliste mit den Zeilennummern der verwendeten Produktionsregeln zusammengestellt (2, 5, 12, 6, 13, 5, 12, 6, 13).

Der zweite Fall repräsentiert einen Pfad, der unabhängig vom Ausgangspunkt zu einem im Zuge der Pfadevaluierung neu erstellten Punkt führt. Das erste Element dieses Pfades ist eine statische Referenzierung der Klasse Punkt (`<static>(Point)`), die durch die schon bekannte Classifier Referenz fortgesetzt werden kann. In diesem Fall navigiert man zu dem Konstruktor der Klasse Punkt, der als Parameter zwei Fliesskommazahlen erwartet (`<create>(x:double, y:double)`).[8] Vom Konstruktor kann der Pfad durch eine Operationsreferenz (`operation_ref`) zum Rückgabewert einer Operation fortgeführt werden, wobei der Rückgabewert eines Konstruktors die neu erzeugte Instanz ist (`<value>(0.5, 0.5)`). Die in den Klammern beigefügten Zahlen, sind die für den Konstruktoraufruf einzusetzenden Werte, für die hier Java-Literale benutzt werden (`0.5`).

Literale erfolgt, wenn die textuelle Repräsentation in Metamodellkonstrukte gewandelt wird, d. h. statisch zur Modellierungszeit und nicht dynamisch zur Laufzeit.

[7] Grundsätzlich können nicht nur Konstanten, sondern beliebige statische Java Felder referenziert werden. Da jedoch die Auswertung der Konstanten-Literale nur einmal erfolgt, werden Änderungen an den Feldern nicht reflektiert.

[8] Die Deklaration der Parameter mit double bedeutet nicht, dass an dieser Stelle direkt die entsprechende Java-Klasse eingesetzt wird. Vielmehr werden Classifier des Vesuf Systems referenziert, die wie ihre Java Pendants benannt sind.

Die Produktionsregeln sind wie folgt: (2, 15, 5, 16, 20, 7, 14, 18, 19, 27, 29, 19, 27, 29).

Die Spezifikation von Pfaden in VEPL ist unkompliziert aber durch die längliche Schreibweise etwas unkomfortabel, umständlich und anstrengend. Um dem entgegenzuwirken, schlagen die Autoren eine abkürzende Schreibweise für VEPL-Pfade vor. Die Regeln zur Vereinfachung sind folgende:

1. `<value>` kann weggelassen werden, wenn es für die Fortsetzung des Pfades unabdingbar ist.

2. `<attribute>` kann immer weggelassen werden.

3. `<operation>` kann immer weggelassen werden.

4. `<static>type.<create>(...)` kann durch `<create>type(...)` abgekürzt werden.

5. `<operation>op(para1:type1, para2:type2, ...).<value>(arg1, arg2, ...)` kann abgekürzt werden durch `<operation>op(para1:type1 = arg1, para2:type2 = arg2, ...)`.

6. Bei `para:type` kann wahlweise `para` oder `:type` weggelassen werden, solange die Operation eindeutig identifizierbar ist.

Die Beipiele lassen sich daher auch wie folgt kompakt beschreiben:

(a) `Startpoint.X.<value>` (nach Regeln 1, 2)

(b) `<create>Point(x = 0.5, y = 0.5)` (nach Regeln 4, 5, 6)

4.2.2.3 Interpreterbeschreibung

Um eine Vesuf Applikation auszuführen, wird der Interpreter mit einem Applikationsdeskriptor initialisiert und gestartet. Er lädt die im Applikationsdeskriptor festgelegten Modelle und erzeugt deren Laufzeitpendants (runtimes). Die Applikationsausführung wird dadurch angestossen, dass der Interpreter die initiale Dialogmaschine startet. Der Interpreter wird über Zustandsänderungen der Dialogmaschine informiert und kann geeignete Maßnahmen, z. B. zur Darstellung eines Dialogs, ergreifen. Um festzustellen, welches Präsentationselement für den eingenommenen Dialogzustand adäquat ist, wird der ebenfalls im Deskriptor spezifizierte Mapper herangezogen. Die Verbindung vom Interpreter hin zur eigentlichen Fachlogik ist über das Dialogmodell indirekt hergestellt. So kann durch geeignete Transitionsanschriften dafür gesorgt werden, dass Teile der Fachlogik ausgeführt werden.

Das Vesuf System unterstützt derzeit zwei verschiedene Interpretermodi: Aktive Interpreter wie der GUIRunner und das VesufApplet sind für die Ausführung von interaktiven Applikationen (z. B. mit AWT oder Swing Präsentation) vorgesehen und können selbständig Dialoge öffnen. Als passiven Interpreter stellt Vesuf das ApplicationServlet zur Verfügung, das Dialogdarstellungen als Antwort auf HTTP-Requests generiert.

Die aktiven Interpreter überwachen Zustandsänderungen in den Dialogmaschinen, die diese über Ereignisbenachrichtigungen publizieren. Wird ein neuer

Dialogzustand eingenommen, so erzeugt der Interpreter das zugehörige Präsentationselement, und sorgt für dessen Darstellung. Wird dagegen ein Dialogzustand verlassen, so löscht der Interpreter alle zu diesem Zustand gehörigen Präsentationselemente. Der Kommandozeileninterpreter (GUIRunner) beendet sich selbst, sobald die letzte aktive Dialogmaschine in einen Endzustand übergeht. Die vom interaktiven Interpreter erzeugten Präsentationselemente (z. B. AWT-Widgets) sind selbst für die Verarbeitung von Benutzereingaben zuständig (siehe Abschnitt 4.2.5.2).

Die vom ApplicationServlet auszuführende Applikation kann sowohl als Init-Parameter des Servlets in einer Web-Applikation als auch als Teil der URL eines HTTP-Requests angegeben werden. Innerhalb einer HTTP-Session wird eine Applikation jeweils nur einmal ausgeführt, so dass ein Benutzer beim erneuten Zugriff auf die gleiche URL, denselben Anwendungskontext präsentiert bekommt. Beim ersten Zugriff auf eine bestimmte Anwendung wird diese instantiiert und die initiale Dialogmaschine gestartet und bis zum ersten stabilen Zustand[9] ausgeführt. Die zu diesem Zustand gehörigen Präsentationselemente werden erzeugt und als Ergebnis der Anfrage zurückgesandt. Dazu stellen die Präsentationselemente für den passiven Interpreter (z. B. HTML-Elemente) eine Methode zur Verfügung, um ihre Darstellung auf den Ausgabestrom des HTTP-Requests zu schreiben. Der Content-Type der Präsentationselemente (z. B. text/html oder image/gif) kann als Eigenschaft im Präsentationsmodell angegeben werden. Benutzeraktionen werden dem Vesuf System als HTTP-POST Request zugänglich gemacht. Das ApplicationServlet extrahiert die Requestparameter, und leitet sie an die zuständigen Präsentationselemente weiter, die ihrerseits den Zugriff auf die Anwendungslogik durchführen. Nach Verarbeitung aller Benutzeraktionen, die auch Zustandsänderungen in den Dialogmaschinen auslösen können, wird der wiederum aktuelle Zustand der Anwendung bestimmt, und die zugehörigen Präsentationselemente generiert.

4.2.2.4 Eventmechanismus

Ereignisse können auf der Objekt- / Taskebene und auf der Dialogebene ausgelöst werden. Vesuf unterstützt zur Zeit keine Events auf der Präsentationsebene.[10] Die Entitäten der Domänenschicht publizieren Ereignissse, die sie selbst betreffen. Wird z. B. der Wert eines Attributes geändert wird eine Benachrichtigung, die diese Wertveränderung (value changed) zum Inhalt hat, generiert. Weitere mögliche Ereignisse in der Domänenschicht sind Änderungen in Constraints (constraints changed), Hinzufügen oder Entfernen von Elementen aus Mengen (values added / removed) oder die Initiierung, Beendigung oder der Abbruch von Operationen (operation started / finished / failed).

Ereignisse auf der Dialogebene werden von den Dialogzuständen (states) selbst und von der StateMachine, die die Zustände enthält, publiziert. Folgende Ereignisse können auftreten: Erreichen des Endzustands einer StateMachine (statemachine stopped), Betreten und Verlassen eines Dialogzustands (state entered / exited), Beendigung der mit einem Zustand verknüpften Aktivität (ac-

[9] Als stabilen Zustand bezeichnen wir einen Dialogzustand, der nicht automatisch, sondern erst nach einem externen Ereignis verlassen wird.

[10] Da die Präsentationsebene auf den anderen beiden Ebenen aufsetzt, ohne dass diese von der Präsentation abhängig sind, und Präsentationselemente nicht untereinander kommunizieren, gibt es zur Zeit keine möglichen Adressaten für Ereignisse der Präsentationsebene.

tivity completed). Des weiteren können in allen Schichten Ereignisse, die Änderungen innerhalb von Pfaden (path change) beschreiben, publiziert werden, sowie Ereignisse, die für Elemente publiziert werden, wenn sich deren Abhängigkeiten ändern (dependency change).

Nicht immer treten Ereignisse einzeln auf. Häufig fallen mehrere Ereignisse zusammen. Dies kommt zum einen vor, wenn mehrere Änderungen als Einheit vollzogen werden (z. B. wenn X und Y Koordinate eines Punktes gleichzeitig gesetzt werden sollen). Zum anderen können Ereignisse über Dependencies (siehe Abschnitt 4.2.3.3) Folgeereignisse auslösen. Aus Performancegründen und zur Gewährleistung von Konsistenz ist es wünschenswert, dass diese Ereignisse gemeinsam publiziert werden, so dass die Benutzungsschnittstelle nur einmal aktualisiert werden muss. Die Autoren führen dazu einen Ereigniscontainer ein (InstanceEvent), der die einzelnen Ereignisse (InstanceElementEvents) enthält. Alle Instanzelemente bieten die Methoden `addInstanceListener()` und `removeInstanceListener()` an, mit denen man sich für Ereignisse eines Laufzeitelements registrieren kann. Verwaltet werden diese Registrierungen nicht von den Laufzeitelementen selbst, sondern von einem EventDispatcher, der für jede Ebene (Objekt-, Dialog-) einer Laufzeitumgebung existiert.

Auftretende Ereignisse werden von den Ereignisquellen direkt an den zuständigen Dispatcher gemeldet, der diese in einem Container sammelt und dann an alle betroffenen registrierten Listener weiterleitet. Über die Methoden `collectEvents()` und `commitEvents()` kann der Dispatcher aufgefordert werden, Ereignisse nicht sofort weiterzuleiten: Alle Ereignisse, die nach einem `collectEvents()` Aufruf gemeldet werden, werden erst nach einem `commitEvents()` Aufruf gemeinsam publiziert. Natürlich dürfen die collect und commit Aufrufe auch wiederholt geschachtelt auftreten.[11]

Durch diesen Mechanismus kann es passieren, dass ein und dasselbe Laufzeitelement nacheinander zwei Ereignisse publiziert, die aber vom Dispatcher gemeinsam weitergeleitet werden sollen. In diesem Fall werden nicht einfach beide Ereignisse (InstanceElementEvents) dem Ereigniscontainer hinzugefügt. Wenn der aktuelle Ereigniscontainer schon einen Event für ein bestimmtes Laufzeitelement enthält, werden weitere Ereignisse für dieses Laufzeitelement mit dem bereits vorhandenen Event zusammengefügt. Dazu besitzt ein InstanceElementEvent eine Methode `merge()`, die von den speziellen Event Subklassen in geeigneter Weise überschrieben wird.

Implementation Damit Events vom Dispatcher publiziert werden können, müssen sie zuvor gemeldet werden. Für die Implementationsschicht bietet Vesuf zwei Möglichkeiten an, wie Ereignisse festgestellt und gemeldet werden: Zum einen können die ImplementationAccessors (siehe Abschnitt 4.2.3.4) so konfiguriert werden, dass sie nach erfolgten schreibenden Zugriffen die durchgeführten Änderungen melden. Damit werden jedoch nur Ereignisse gemeldet, die über den Vesuf Zugriffsmechanismus ausgelöst werden. Für in Java implementierte Objekte ist dies meistens nicht ausreichend, da Objektverhalten implizit Ereignisse auslösen kann. Deshalb bietet Vesuf für Java Implementationen eine

[11] Auch wenn die `commitEvents()` Methode dies suggerieren mag, so ist ein Rollback von Events nicht möglich. Der Collect/Commit-Mechanismus ist lediglich in der Lage die Publizierung bereits eingetretener Ereignisse zu verzögern, er kann jedoch keine Ereignisse Rückgängig machen.

```
01:  public void valueChanged(String name, Object value);
02:  public void valueChanged(String name, double value);
03:  ...
04:  public void addElement(String name, Object key);
05:  public void addElement(String name, double key);
06:  ...
07:  public void removeElement(String name, Object key);
08:  public void removeElement(String name, double key);
09:  ...
10:  public void collectEvents();
11:  public void commitEvents();
```

Abbildung 4.6: Methoden des EventDispatcherProxies

```
01:  /** Construct a new Rectangle. */
02:  public Rectangle(Point point, double width, double height)
03:  {
04:    this.point = point;
05:    this.width = width;
06:    this.height = height;
07:    this.dispatcher = new EventDispatcherProxy(this);
08:  }
09:
10:  /** Set the width. */
11:  public void setWidth(double width)
12:  {
13:    this.width = width;
14:    dispatcher.valueChanged("Width", width);
15:  }
```

Abbildung 4.7: Anwendung eines EventDispatcherProxies

weitere Möglichkeit an, wie Events aus der Implementationsebene bekanntgemacht werden können. Der Programmierer der Implementationsschicht kann dazu für seine Objekte von Vesuf zu diesem Zweck bereitgestellte EventDispatcherProxies anlegen. Das EventDispatcherProxy findet selbständig die aktuelle Applikationsumgebung[12] und damit den für die Implementationsschicht zuständigen EventDispatcher. Das EventDispatcherProxy bietet Methoden zur Eventmeldung (s. Abb. 4.6, Zeilen 1-9), die im Gegensatz zu ähnlichen Methoden des EventDispatchers keine Elemente aus der Vesuf Laufzeit (InstanceElements) als Parameter erwarten. Stattdessen können die betroffenen InstanceElements durch einen String benannt und neue Werte für Attribute, qualifizierende Attribute und Indizes als Java-Objekte oder direkt als Werte eines der acht Java-Basistypen[13] und nicht als Vesuf Instanz übergeben werden. Das EventDispatcherProxy ermöglicht also eine Anbindung an die Vesuf Laufzeitumgebung, ohne dass der Anwendungsprogrammierer mit den komplexen Laufzeitelementen hantieren muss.

Abbildung 4.7 zeigt beispielhaft, wie dieser Mechanismus in der Implementation des Shape Modells aussieht. In Zeile 7 wird ein EventDispatcherProxy

[12]Je nach Interpreter können mehrere Applikationsumgebungen gleichzeitig instantiiert sein (z. B. mehrere Applets in einer Webseite, oder mehrere Sessions in einem Servlet). Über einen ApplicationManager kann in der Implementation jederzeit die Applikationsumgebung bestimmt werden, der ein Objekt zuzuordnen ist.

[13]Die Java-Basistypen (primitives) sind [Gosling et al. 1996]: boolean, char, double, float, long, int, short und byte.

für das in der Erzeugung befindliche Rechteck angelegt. In Zeile 14 wird dieses Proxy, dass in einem Feld namens `dispatcher` abgelegt wurde, verwendet, um den schreibenden Zugriff auf das Width-Attribut zu melden. Dazu wird der Name des geänderten Attributs und der neue Wert übergeben. Das Proxy versucht nun von der Rectangle-Instanz, die es für das Rechteck, dem es in Zeile 7 zugewiesen wurde, erzeugt hat, die Instanz des Width-Attributs zu finden. Wäre im Objektmodell kein Width-Attribut definiert, würde an dieser Stelle eine RuntimeException geworfen. Als nächstes legt das Proxy für den neuen Wert (`width`) eine Vesuf Instanz an. Die beiden Laufzeitelemente (geändertes Laufzeitelement und neue Wertinstanz) werden dann als Parameter für einen `valueChanged()` Aufruf am eigentlichen EventDispatcher verwendet, der die weitere Eventbehandlung übernimmt.

4.2.3 Domänenschicht

Die Domänenschicht beinhaltet Aufgaben- und Objektmodell einer Vesuf Anwendung. Die Modellierung der Aufgabenschicht wird derzeit mit Use Cases durchgeführt. Da UML das Modellelement Use Case als eine Spezialisierung von Classifier einführt, können Use Cases mit Eigenschaften und Verhalten modelliert werden. Die temporalen Aspekte und die hierarchische Aufgabendekomposition fließen aus Gründen der in Abschnitt 2.2.4.1 beschriebenen Einschränkungen des Use Case Ansatzes nicht in die formale Modellspezifikation ein. Stattdessen bilden die identifizierten Aufgaben mit den Ergebnissen der Domänenanalyse ein gemeinsames Domänenmodell auf Basis eines Klassendiagramms.

Die Modellierung der Objektschicht erfolgt mit Hilfe der üblichen UML Klassendiagrammelemente, wie vor allem Packages, Klassen, Interfaces, Attribute, Operationen, Assoziationen und Constraints. Nachfolgend wird auf Entitäten näher eingegangen, die Besonderheiten aufweisen. Dies betrifft Elemente, deren UML-Semantik erweitert wurde (Constraints) und deren zugeordnete Laufzeitentitäten komplexe Zusammenhänge aufweisen (Elementmengen). Außerdem wird auf das Konzept der Abhängigkeiten (Dependencies) unter Modellelementen und seine Bedeutung für das User Interface eingegangen. Schließlich wird beleuchtet, auf welche Weise Vesuf die Kommunikation zwischen Domänen- und Implementationsschicht regelt. Diese Mechanismen greifen für alle Domänenentitäten (Objekte und Aufgaben) gleichermaßen.

4.2.3.1 Elementmengen

Mit dem Begriff Elementmengen (collection elements) bezeichnen die Autoren Elemente, die zur Laufzeit mit einer Quantität größer eins vorkommen können. Dabei wollen die Autoren Menge nicht im mathematischen Sinne, sondern als Oberbegriff für verschiedene Ausprägungen multiplizitärer Elemente verstanden wissen.

Im UML-Metamodell existieren drei grundsätzlich verschiedene Modellelemente, die in Mengenform auftreten können: Attribute, Parameter und Assoziationen. Attribute und Assoziationen besitzen zwei übereinstimmende Konstrukte, die zur sich Beschreibung von Mengen eignen. Mit Hilfe der Multiplizität kann modelliert werden, wieviele Attributinstanzen zur Laufzeit zu einer Objektinstanz gehören können. Die zweite Eigenschaft kann zur Beschreibung einer

die Menge charakterisierenden Ordnung (ordering) verwendet werden, z. B. ob die Menge ungeordnet, geordnet oder sortiert ist. Darüber hinaus gibt es die Möglichkeit, eine Liste qualifizierender Attribute (qualifier) für ein Assoziationsende festzulegen. Die Werte dieser qualifizierenden Attribute dienen dazu, einzelne Elemente oder Untermengen aus der Menge der Instanzen der Assoziationsenden auszuwählen.

Für Parameter wurden keine speziellen Eigenschaften vorgesehen, die eine Modellierung von Mengen unterstützen. UML bietet ferner keine Möglichkeit, den Container der Elemente zu spezifizieren, d. h. es kann zwar angegeben werden, dass es von einem Element zur Laufzeit mehrere Entitäten gibt, aber nicht, welches Element diese Menge verwaltet. Damit ist es dann natürlich ebenfalls nicht möglich, Constraints eines Containers zu beschreiben. Zusätzlich ist die Spezifikation von Mengen in Mengen mit Problemen behaftet. Eine Möglichkeit, verschachtelte Mengen zu modellieren, besteht darin, die Semantik der Multiplizität zu erweitern, z. B. könnten Notationen der Form *, * eingesetzt werden oder spezielle Arten der Ordnung definiert werden (z. B. 8x8 Ordnung).

Damit bleibt festzustellen, dass das UML-Metamodell derzeit noch keine ausreichenden Eigenschaften zur Modellierung von potentiell multiplizitären Elementen bereitstellt. Im Vergleich der drei Modellelemente besitzt das Assoziationsende die vielfältigsten Mittel zur Mengenmodellierung. Dennoch sind diese Konstrukte nach Meinung der Autoren nicht ausreichend, da der Zugriff auf Elemente einer Menge nicht nur über qualifizierende Attribute, sondern auch über einen Index möglich sein sollte. In der Praxis ist die Handhabung von Mengen der angesprochenen Elemente unverzichtbar. Die Autoren haben daher an dieser Stelle Ergänzungen an allen drei Elementen vorgenommen. Sie führen in Vesuf die Schnittstelle eines `TypedElement` als Basistyp für die drei Elemente ein, da die Konzepte zur Modellierung von Mengen in allen Fällen gleich sind. Attribute, Assoziationsenden und Parameter erhalten durch dieses Vorgehen eine Multiplizität, qualifizierende Attribute und eine Ordnungsdimension. Die Ordnungsdimension wird dazu verwendet die Schachtelungstiefe auszudrücken, in der nicht durch qualifizierende Attributwerte, sondern durch den Index auf einzelne Elemente der Menge zugegriffen wird. Die Summe aus Ordnungsdimension und der Anzahl qualifizierender Attribute stellt damit die gesamte Schachtelungstiefe des Elements dar.

Durch die geringfügigen Ergänzungen erreichen wir, dass beliebig tief verschachtelte Mengen in zwei unterschiedlichen Formen modellierbar sind. Mit Hilfe der qualifizierenden Attribute können Abbildungsmengen (maps) definiert werden. Einfache Listen (sequences) lassen sich durch die Angabe der Ordnungsdimension kreieren. Man erhält dadurch die Möglichkeit, beliebig tiefe Maps und darin beliebig tiefe Sequences zu modellieren. Da qualifizierende Attribute vor der Ordnungsdimension ausgewertet werden, lassen sich nur Listen in Abbildungsmengen, aber keine Abbildungsmengen in Listen modellieren. Dieses Problem läßt sich umgehen, indem man Listen auch durch qualifizierende (Integer-) Attribute modelliert. Dadurch geht jedoch die spezielle Semantik von Listen verloren: Während Indizes fortlaufend sind, können qualifizierende Attribute beliebige Werte annehmen.

Die außergewöhnliche Modellierung von Mengen in UML ohne die direkte Angabe der Containerart führt dazu, dass auch die Abbildung multiplizitärer Elemente in die Laufzeitebene mit Schwierigkeiten behaftet ist. Betrachtet man die Implementation von Mengen auf programmiersprachlicher Ebene, lässt sich

erkennen, dass eine explizite Auswahl und Behandlung der Container unumgänglich ist. Die Autoren reflektieren diese Tatache in der Laufzeitebene dadurch, dass sie Laufzeitelemente auch für die Behälter der Elemente zur Verfügung stellen. Die Evaluierung eines Modellelementes mit einer Mengenangabe führt zur Konstruktion von Laufzeitelementen für die Container und die enthaltenen Entitäten.

4.2.3.2 Constraints

In UML wird ein Constraint als eine semantische Bedingung oder Restriktion in textueller Form verstanden. Ein Constraint ist ein Boolscher Ausdruck,[14] der einem oder mehreren Modellelementen zugeordnet ist. Eine Constraint-Instanz kann zur Validitätsüberprüfung von Laufzeitelementen herangezogen werden. Zur Notation des Constraint kann natürliche Sprache oder eine spezielle Constraintlanguage mit definierter Syntax und Semantik verwendet werden. Eine Möglichkeit ist die ebenfalls in UML definierte Object Constraint Language (OCL, Kapitel 7 in [OMG 2000a]).

Constraints spielen auch im Zusammenhang mit dem User Interface eine wichtige Rolle. So können Eingaben eines Anwenders schon von der Präsentation aus auf ihre Gültigkeit hin überprüft werden (siehe dazu die Bedeutung der Application Interface Komponente im Seeheim Modell, Abschnitt 2.1.1). Interessant ist außerdem, Restriktionen auch für weitere Zwecke einzusetzen, z. B. können sie dazu benutzt werden, den Wertebereich eines Attributes festzulegen. Um einigen Präsentationselementen zu erlauben, solche weiterführenden Informationen auszulesen, reicht die Semantik eines einfachen Boolschen Ausdrucks nicht aus. Die Autoren erweitern daher das Konzept eines Constraints für die Vesuf Umgebung leicht. Der Status des Boolschen Ausdrucks kann in Vesuf durch die `isValid()` Methode einer Constraintinstanz abgefragt werden, wobei der zu überprüfende Zustand (z. B. der Wert eines Attributes) als Parameter übergeben wird. Zusätzlich stellen Constraintinstanzen eine `check()` Methode zur Verfügung, die im Gegensatz zur `isValid()` Methode im Falle eines ungültigen Zustandes eine Exception (`ConstraintsViolatedException`) wirft. Diese Exception enthält weitere Informationen, die von der Benutzungsschnittstelle ausgewertet werden können, um dem Benutzer mitzuteilen, warum seine Eingabe ungültig war. Um die oben beschriebene zusätzliche Semantik (z. B. Wertebereich) umzusetzen, können Constraintinstanzen Properties erhalten, die zur Laufzeit ausgewertet werden.

Jedem Constraint wird ein ConstraintHandler zugewiesen. Instanzen von ConstraintHandlern verwalten die Properties für eine ConstraintInstanz und implementieren die von den Properties abhängige Validitätsprüfung. Properties können als konstanter Wert (Literal) oder als Pfad (ausgehend von der Constraintinstanz) spezifiziert werden. Damit eine bestimmte Property auch in den Validierungsmethoden berücksichtigt wird, muss eine spezielle Subklasse einer ConstraintHandler-Instanz implementiert werden. Im Vesuf System sind zur Zeit fünf Properties vordefiniert, die die Auswertung der Constraints beeinflussen:

valid Der Validitätszustand kann direkt als Property, die zu einem Boolschen Wert evaluiert, ausgedrückt werden. Dazu wird man üblicherweise Pfade

[14]Der Boolschen Algebra liegt die Annahme zu Grunde, dass alle Boolschen Ausdrücke genau festgelegte Werte besitzen. Die Propositionen können wahr oder falsch sein.

verwenden, da ein fortwährend erfülltes oder nicht erfülltes Constraint im allgemeinen wenig sinnvoll erscheint.

check Statt einen Pfad anzugeben, der den Validitätszustand zurückliefert (valid), kann ein Pfad angegeben werden, dessen fehlerfreie Auswertung die Validität bedeutet. So kann z. B. eine Methode angegeben werden, die bei nicht erfüllten Constraints eine Exception wirft. Der Vorteil gegenüber einer valid Property ist, dass in einer Exception ein Grund für die Invalidität angegeben werden kann. Die Exception kann von der Benutzungsschnittstelle ausgewertet werden, um dem Benutzer ein Feedback zu liefern, warum seine Aktion gescheitert ist.

values Diese Property dient dazu, die erlaubten Werte eines getypten Elements (Attribut, Assoziationsende, Parameter) anzugeben. Das User Interface kann diese Werte z. B. benutzen, um eine Drop-down Liste darzustellen, aus der der Benutzer direkt einen Wert auswählen kann.

range Über diese Property kann der Wertebereich eines getypten Elements festgelegt werden. Damit überprüft werden kann, ob ein Wert innerhalb des Bereiches liegt, muss für den Datentyp ein DatatypeHandler definiert sein. Vesuf stellt für die Standarddatentypen, für die Wertebereiche definiert werden können (Integer, Real, Date), bereits DatatypeHander zur Verfügung. Wenn man für eigene Datentypen (z. B. Internetadresse) Wertebereiche definieren will, muss ein entsprechender DatatypeHandler implementiert werden. Über eine weitere Property kann die Präzision (precision) angegeben werden. Die Präzision wird bei der Validierung nicht überprüft, aber sie kann verwendet werden um die Bereichsangabe dynamisch in eine Liste der erlaubten Werte umzuwandeln. Damit ist es zum Beispiel möglich, dass zur Darstellung eines Datumsfeldes ein Scrollbar verwendet wird, der je nach Genauigkeit tageweise oder monateweise einstellbar ist.

active In UML können Attribute und Assoziationsenden als „frozen" deklariert werden. Damit wird festgelegt, dass ein Element grundsätzlich nicht manipulierbar ist. In Anwendungen müssen Elemente jedoch häufig abhängig vom Systemzustand temporär gesperrt werden. Dabei gibt es zwei Gründe für das Sperren. Einerseits kann die Sperre von der Benutzungsschnittstelle eingerichtet werden, z. B. in einem read-only Dialog, der verhindern soll, dass ein Benutzer aus Versehen Änderungen eingibt. Zum anderen kann die Sperre Teil des Systems sein, um bestimmte Teile der Anwendung zwischenzeitlich zu deaktivieren. Ersteres wird im Präsentationsmodell spezifiziert (s. u.) und ist nicht Teil des Domänenmodells. Letzteres kann durch ein Active-Constraint im Domänenmodell erreicht werden. So kann sichergestellt werden, dass je nach Systemzustand bestimmte Teile der Anwendung – unabhängig von der Benutzungsschnittstelle – nicht aktivierbar sind.

Für statische Constraint-Properties ist die Spezifikation als Literalkonstante im Modell ausreichend. Dynamische, sich während der Laufzeit ändernde Properties können durch Methoden der Implementationsschicht realisiert werden. Deren Rückgabewerte sind über modellierte Operationen oder Attribute in Pfaden zugreifbar. Dabei ist es für das Vesuf System unerheblich, ob die Constraintmethoden von Hand programmiert werden oder von Constraint-Codegeneratoren

automatisch erzeugt werden. Ein OCL-Parser könnte beispielsweise dazu eingesetzt werden, in OCL definierte Constraints des Modells in Java-Code für die Implementationsschicht umzusetzen (siehe [Hussmann et al. 2000]). Die modellierten Constraints werden dann durch Pfade automatisch mit der generierten Implementationen verbunden.

Unabhängig von der Art der Implementation existieren zur Laufzeit Constraintinstanzen, die dynamisch Pfade auswerten und zur Überprüfung von Bedingungen einsetzen. Die Constraintinstanzen beziehen sich dabei immer auf genau ein Laufzeitelement, was einen eindeutigen Kontext zur Auswertung der Propertypfade sicherstellt. Ein Laufzeitelement kann jedoch beliebig viele Constraints besitzen. Um das Abfragen der Constraints von außen (z. B. aus der Präsentationsschicht) zu vereinfachen, führen die Autoren das Konzept der Unified Constraints ein. Jedem Laufzeitelement (außer den Constraintinstanzen) ist eine Unified Constraintinstanz zugeordnet, die alle Properties der auf dem Element definierten Constraints zusammenfasst. Die Constrainthandler Instanzen bieten dazu eine Methode `unifyProperties()` an, mit der die einzelnen Properties zusammengefasst werden. Beispielsweise bildet die ValuesConstraintHandler Instanz die Schnittmenge aus allen definierten Values-Properties.

Eine weitere Besonderheit der Constraints im Vesuf System ist, dass sie vom Kontext abhängig sind und dadurch nicht zu jedem Zeitpunkt Gültigkeit haben müssen. Zwar ist es nicht möglich, zur Laufzeit neue Constraints hinzuzufügen oder alte zu entfernen, aber die im Modell definierten Constraints können abhängig vom Zustand des zugeordneten Laufzeitelements ausgewertet werden. Pfade, die ins Nichts führen (z. B. weil sie über eine Referenz navigieren, der im aktuellen Kontext kein Wert zugeordnet ist), verursachen bei ihrer Auswertung keinen Fehler, sondern liefern lediglich kein Laufzeitelement zurück.[15] Führt ein Propertypfad ins Nichts, so interpretiert Vesuf dies so, als ob die Property auf diesem Element nicht definiert wäre. Dieses Detail wird z. B. mit Hilfe des Composite Pfadelements zu einem mächtigen Werkzeug. Im Shape Beispiel machen die Autoren sich dies zunutze: Auf dem Punktobjekt werden drei verschiedene Constraints spezifiziert, die nicht alle zugleich gelten sollen, sondern von der Rolle des Punktes abhängen (vgl. Abb. 4.3). So soll für alle Punkte gelten, dass ihre Koordinaten im Bereich $[0, 1]$ liegen. Für Startpunkte eines Rechteckes und Mittelpunkte eines Kreises gelten aber weitere Bedingungen. Diese Bedingungen sind in Operationen der Klassen Circle und Rectangle implementiert. Über einen Pfad wie `<owner>.<owner>.<composite>Rectangle.checkPointX(<state>)`[16] können diese Operationen aufgerufen werden. Dies geschieht jedoch nur dann, wenn es sich beim Composite des Punktes tatsächlich um ein Objekt des im Composite-Pfadelement angegebenen Typs handelt.

Dieses Beispiel zeigt auch, wie es möglich ist, Constraints abhängig von Elementen (in diesem Fall Rechteck) zu definieren, zu denen die beschränkten Elemente (in diesem Fall Punkt), keine direkte Navigationsmöglichkeit haben. Das Vesuf System verwaltet automatisch, welche Instanz das Composite einer anderen Instanz ist. Das Composite einer Instanz ist dabei immer eindeutig, da laut Definition (siehe [OMG 2000a]) eine Instanz immer nur Teil einer einzigen

[15] Im Gegensatz dazu liefert ein Pfad, der als letzes Element den Wert einer Referenz auswertet für den Fall das diese null ist eine „getypte Nullinstanz“ zurück.

[16] Dieser Pfad geht aus von dem Constraint und verläuft über folgende Modellelemente: `<owner>`=Attribut X/Y, `<owner>`=Klasse Punkt, `<composite>Rectangle`=Klasse Rechteck, `checkPointX/Y()`=Eine Operation der Klasse Rechteck.

Kompositionsbeziehung sein kann, auch wenn das korrespondierende Modellelement als Teil mehrerer Kompositionsbeziehungen entworfen wurde. Wird diese Bedingung zur Laufzeit verletzt, so wird vom Vesuf System sofort eine Runtime-Exception geworfen. Die Composite Metabeziehung und das Composite Pfadelement ermöglichen, dass dieses Konzept von den anderen Vesuf Konstrukten wie Constraints genutzt werden kann. Ohne die Implementation oder die Modellierung des Elementes, das Teil der Kompositionsbeziehung ist, zu ändern, können aus Sicht des Composites weitere Constraints auf dem Teilelement definiert werden. Diese nichtinvasive Erweiterungsmöglichkeit ist besonders wichtig, um einmal definierte und implementierte Modelle in möglichst vielen verschiedenen Kontexten einsetzen zu können.

Durch die modellierten Constraints kann das Vesuf System unabhängig von der Benutzungsschnittstelle und der Implementation Falscheingaben erkennen und aussagekräftige Fehlermeldungen propagieren. Ein gewisses Maß an Benutzbarkeit ist damit bereits durch das Domänenmodell gewährleistet und nicht vom Anwendungsprogrammierer oder dem User Interface Designer abhängig.

4.2.3.3 Dependencies

In Abschnitt 4.2.2.4 haben die Autoren den Eventmechanismus von Vesuf beschrieben. Damit können Elemente sie selbst betreffende Änderungen publizieren. Häufig kommt es jedoch vor, dass nicht die Quelle einer Änderung direkt von Interesse ist, sondern von dieser Quelle wiederum abhängige Elemente. Besonders deutlich wird dieser Zusammenhang am Beispiel der Constraints.

Ein Constraint kann über einen PathConstraintHandler (siehe Abschnitt 4.2.3.2) z. B. zu einem Attribut in Beziehung gesetzt werden. Ändert sich der Wert des Attributs (genauer gesagt der Attributinstanz), dann hat sich damit auch das Constraint geändert. Dennoch ist nicht das Constraintobjekt direkt manipuliert worden, so dass es keine sich selbst betreffende Änderung publiziert. Daher wird ein Mechanismus gebraucht, der bei Änderung des Attributs auch die Änderung des Constraints publiziert.

Diese Abhängigkeiten treten aber nicht nur bei Constraints auf. Im Shape Beispiel hat ein Rechteck das Attribut Fläche, das aus Breite und Höhe berechnet wird. Ändern sich Breite oder Höhe ändert sich damit auch der Flächeninhalt. Der Benutzungsschnittstelle muss mitgeteilt werden, dass bei Änderungen an Breite und Höhe auch der Wert des Flächenattributs neu dargestellt werden muss. Um dies zu erreichen, greifen die Autoren in Vesuf das Konzept der Dependencies auf.

Eine Dependency in UML ist eine Beziehung zwischen abhängigen Elementen (clients) und Elementen, von denen die Clients abhängig sind (supplier). Dependencies sind also Modellelemente, die die Abhängigkeiten zwischen anderen Modellelementen beschreiben. In Vesuf wirken sich Dependencies auf die Generierung von Events aus. Ändern sich auf Grund eines Events Supplier einer Dependency, werden für die entsprechenden Clients ebenfalls Events erzeugt. Dependencies haben keine Auswirkungen auf die Objektschicht selbst (in der sie definiert sind) oder gar auf die Implementationsschicht. Sie dienen lediglich dazu, dass implizite Abhängigkeiten, die in der Implementation exisitieren, auch im User Interface Berücksichtigung finden. Die Benutzungsschnittstelle weiß z. B. nur, dass es ein Attribut für den Flächeninhalt gibt. Dass dies in einer `getArea()` Methode implementiert ist, weiß sie nicht, und dass die `getArea()` Methode die

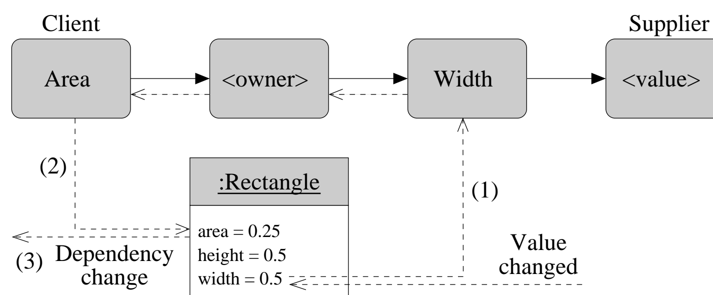


Abbildung 4.8: Eventgenerierung bei Dependencies

Werte Breite und Höhe benötigt, schon gar nicht. Eine Dependency kann diesen Sachverhalt beschreiben, ohne zu viele Implementationsdetails preiszugeben.

In Vesuf lassen die Autoren nur Dependencies mit jeweils genau einem Client und Supplier zu.[17] Das erlaubt uns als zusätzliche Eigenschaft einer Dependency den Dependency-Path einzuführen. Der Dependency-Path ist der Pfad (siehe Abschnitt 4.2.2.2) vom Client zum Supplier. Damit ist in Vesuf nicht nur festgelegt, welches Modellelement der Supplier einer Dependency ist. Mit Hilfe des Pfades kann das Vesuf System zur Laufzeit feststellen, welches Laufzeitelement der Supplier für eine Clientinstanz ist. Im Shape Beispiel hat das Area-Attribut eine Dependency mit dem Pfad `Area.<owner>.Width.<value>`, es ist also vom Wert des Width-Attributs seines Rechteckes abhängig.

Die Auswertung von Dependencies ist Aufgabe des EventDispatchers (siehe Abschnitt 4.2.2.4). Bevor ein Event verschickt wird, muss der Dispatcher analysieren, ob irgendein Laufzeitelement Client einer Dependency ist, die im Pfad die Quelle des Events enthält. Berücksichtigung bei der Analyse finden dabei nur Laufzeitelemente, auf denen ein Listener registriert wurde. Nur diese Elemente sind dem Dispatcher bekannt und nur für diese müssen abhängige Events erzeugt werden.

Auch auf benutzerdefinierten[18] Pfaden können Listener registriert werden. Damit bilden Pfade implizite Dependencies, denn wenn sich ein Laufzeitelement, das Teil eines Pfades ist, ändert, dann bezieht sich der Pfad im weiteren Verlauf möglicherweise auf andere Laufzeitelemente. Der EventDispatcher analysiert dies und fügt Events für die betroffenen Pfadelemente (PathElementInstances) ein, die anzeigen, dass sich der Pfad geändert hat (path change).

Der Analysemechanismus ist dabei derselbe. Der EventDispatcher überprüft alle ihm bekannten Instanzen von Pfadelementen daraufhin, ob sie zur Zeit das ereignisauslösende Element referenzieren. Handelt es sich bei dem betroffenen Pfad um einen Dependency-Pfad, so muss für das Element in der Wurzel des Pfades (das ist der Client der Dependency) ein Event generiert werden, der anzeigt, dass sich das Element aufgrund einer Abhängigkeit geändert hat (de-

[17] Dies ist bedeutet keine Einschränkung, da jedes Element in beliebig vielen Dependencies Client sein kann. Es lediglich nicht möglich, dass ein Element neben einem anderen Client in derselben Dependency ist. In diesem Fall muss man zwei Dependencies mit demselben Supplier definieren. Gleiches gilt auch für die Supplier-Seite. Es ist weiterhin möglich, dass ein Element zugleich Client und Supplier einer Dependency ist.

[18] Als benutzerdefinierte Pfade bezeichnen wir diejenigen Pfade, die nicht Teil einer Dependency sind, sondern z. B. in der Präsentation dargestellte Elemente referenzieren.

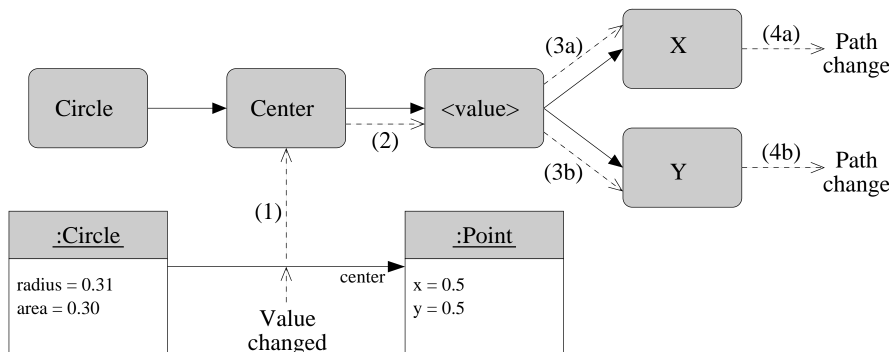


Abbildung 4.9: Eventgenerierung für benutzerdefinierte Pfade

pendency change). Handelt es sich um einen benutzerdefinierten Pfad, so werden für alle nachfolgenden Pfadelemente, bis hin zu den Blättern des Pfad-Baumes PathChange Events generiert, sofern auf den entsprechenden Pfad-Elementen Listener registriert sind. Für diese Vorgänge folgt nun jeweils ein Beispiel.

Abbildung 4.8 zeigt die Generierung eines DependencyChange Events für das oben angesprochene Flächenattribut eines Rechteckes. In der oberen Hälfte sind die vier Elemente der Dependency-Pfadinstanz zu sehen, unten die Instanz eines Rechteckes. Die gestrichelten Pfeile zeigen den Ablauf des Algorithmus, mit dem der EventDispatcher Dependencies auflöst. Eine Instanz des Width-Attributs meldet eine Wertveränderung (value changed). Der Dispatcher stellt fest, dass er eine Pfadelementinstanz kennt, die auf die entsprechende Attributinstanz verweist (1). Dazu verwendet er die `getElement()` Methode der Pfadelementinstanz, die den Pfad bis zu diesem Punkt traversiert. Da es sich um einen Dependency-Pfad handelt, bestimmt der Dispatcher über das Wurzelelement des Pfades das zugehörige Laufzeitelement, in diesem Fall eine Instanz des Area-Attributes (2). Für dieses Laufzeitelement wird ein DependencyChange Event erzeugt und dem aktuellen Event-Container hinzugefügt. Dieser Event kann nun von der Benutzungsschnittstelle empfangen werden, die daraufhin den dargestellten Wert des Flächeninhaltes aktualisiert (3).

Abbildung 4.9 zeigt eine Pfadkonstruktion, die ausgehend von einem Kreis zu den X und Y-Koordinaten eines Punktes navigiert. Dies ist ein typischer Anwendungsfall für Pfade in der Benutzungsschnittstelle. Ein Dialog, der die Eigenschaften eines Kreises anzeigt, enthält einen Bereich für den Mittelpunkt (center), und stellt von diesem die Koordinaten dar. Um die Darstellung immer aktuell zu halten, werden Pfade definiert, auf deren Instanzen Listener registriert werden. Wird nun dem Kreis ein neuer Mittelpunkt zugewiesen, meldet die Referenz eine Wertveränderung (value changed). Wie schon im obigen Beispiel, durchsucht der Dispatcher seine internen Listen nach Pfadelementinstanzen, die auf das Element des Ereignisses zeigen. In diesem Fall ist das Center Element mit der Referenz, die der Kreis auf den Punkt hat, assoziiert (1). Bei benutzerdefinierten Pfaden ist nicht das Element der Wurzel von Änderungen betroffen, sondern die nachfolgenden Pfadelementinstanzen. Der EventDispatcher geht rekursiv alle nachgeordneten Elemente der betroffenen Pfadelementinstanz durch (2, 3a, 3b) und überprüft, ob auf diesen Elementen Listener eingetragen sind. Für diejenigen Elemente, auf denen Listener registriert sind, werden Pfadände-

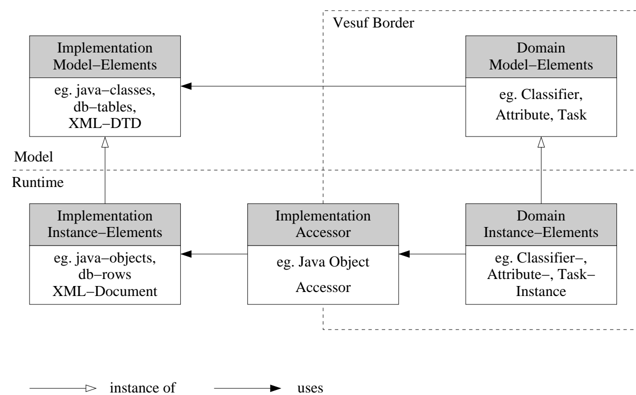


Abbildung 4.10: Das ImplementationAccessor Konzept

rungsevents erzeugt und in den aktuellen Event-Container eingefügt. In diesem Fall sind dies die Pfadelementinstanzen für die X und Y-Attribute des Punktes. Nachdem die Ereignisverarbeitung abgeschlossen ist, werden die Events des Event-Containers publiziert (4a, 4b).

Das UML-Metamodell [OMG 2000a] definiert vier Arten von Dependecies: Abstraction, Binding, Usage und Permission. Der in Benutzungsschnittstellen benötigten Abhängigkeit entspricht die Usage Dependency, denn die Clients benutzen die Supplier für ihre eigene Implementation (z. B. zur Berechnung der Fläche werden Breite und Höhe benutzt). Die Autoren führen für diese Klasse einen neuen Stereotyp *VesufDependency* ein. Der Dependency-Pfad wird als TaggedValue definiert. Über den requiredTag Mechanismus wird sichergestellt, dass jede VesufDependency einen Pfad hat.

4.2.3.4 ImplementationAccessors

Die Kommunikation zwischen Domänen- und Implementationsschicht wird zur Laufzeit durch eine zwischengeschaltete Komponente realisiert. In Abb. 4.10 ist zu erkennen, dass Laufzeitentitäten der Domänenschicht auf Implementation-Accessors zurückgreifen, um Aktionen in der Implementationsschicht auszulösen. Konzeptionell müssen ImplementationAccessors folgende Dienste bereitstellen:

1. Auslesen des aktuellen Wertes einer Attributinstanz.
2. Setzen des aktuellen Wertes einer Attributinstanz.
3. Aufrufen einer beliebig parametrisierten Operationsinstanz.

Durch die Einführung dieser Schnittstelle wird das Domänenmodell von der zugrunde gelegten Technik der Implementationsschicht entkoppelt. So ist es durch Änderung der vom Domänenmodell eingesetzen ImplementationAccessors mit wenig Aufwand möglich, die Technik der Implementationsschicht z. B. von Java auf JDBC umzustellen.

Im Vesuf System wurden ImplementationAccessors für den Zugriff auf Java-Entitäten realisiert. Nachfolgend wird auf einige Einzelheiten dieser Umsetzung

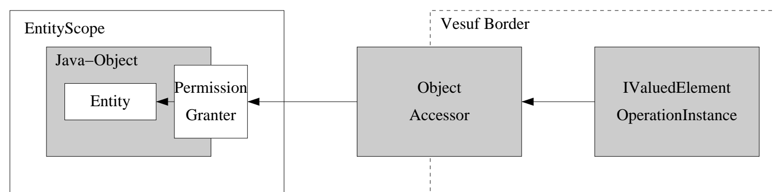


Abbildung 4.11: PermissionGranter-Konzept

näher eingegangen. In der Implementation kann ein Attribut durch eine Variable, durch Zugriffsmethoden oder durch eine Kombination aus beiden repräsentiert werden. Dabei sind abhängig vom Kontext verschiedene Lösungen angebracht. Im einfachsten Fall existiert nur eine Variable, auf die von außen zugegriffen werden kann. Normalerweise werden Zugriffs- und Modifikationsmethoden hinzugefügt, um die Sichtbarkeit der Variable beschränken zu können. Unter anderen Bedingungen werden nur Zugriffsmethoden eingesetzt und der Zustand wird nicht in einer Variablen gespeichert. Diese Vielzahl möglicher Realisierungen eines Attributes erfordert einen flexiblen und fein steuerbaren Mechanismus. Das Vesuf System erlaubt daher die Spezifikation von Eigenschaften, mit denen der Zugriff beeinflusst werden kann:

accesspolicy_get/set (für 1, 2) Duch die Zugriffsstrategie werden die Möglichkeiten des Zugriffs und die Reihenfolge der Zugriffsversuche festgelegt. Mit der Standardeinstellung `method, field` wird festgelegt, dass zunächst ein Methodenzugriff und bei Nichterfolg ein direkter Variablenzugriff probiert wird. Eine Einstellung von z. B. `field` führt dazu, dass der Accessor ausschließlich versucht, auf das Feld direkt zuzugreifen. Wird eine Zugriffsstrategie auf einem Modellelement definiert, das weitere Elemente beinhaltet, wie z. B. das Modell selbst, wirkt sie sich auf alle enthaltenen Entitäten aus (deep property). Elemente können diese geerbten Einstellungen überschreiben.

methodname_get/set (für 1, 2) Dient zur Spezifikation des Methodennamens der Zugriffsmethoden.

fieldname (für 1, 2) Erlaubt die Spezifikation des Variablennamens.

methodname (für 3) Für den Aufruf von Methoden kann der Methodenname festgelegt werden.

Die Spezifikation dieser Eigenschaften ist optional und abhängig davon, ob die Defaulteinstellungen des Systems ausreichend sind. So kann Vesuf den Namen von Implementationskonstrukten meist aus denen der modellierten Elemente erschließen (z. B. den Namen einer get-Methode aus dem eines Attributs). Leider ist oben beschriebener Mechanismus noch nicht für alle Fälle ausreichend, da die Sichtbarkeit der Variablen und Methoden bisher nicht betrachtet wurde. Die Implementation-Accessors befinden sich in einem Package innerhalb des Vesuf Systems. Daher ist es ihnen nur erlaubt, auf öffentliche Variablen und Methoden der Anwendungsklassen zuzugreifen. Um auch geschützte[19] Variablen und Methoden vom System aus zuzugreifen zu können, führen die Autoren

[19]Geschützt bedeutet protected, default und private access.

das Konzept des PermissionGranters ein. Das Interface eines PermissionGranters realisiert die bereits mit (1-3) beschriebenen Konzepte mittels typfreier[20] Methoden:

```
01:  public Object get(Field f) throws Throwable;
02:  public void set(Field f, Object o) throws Throwable;
03:  public Object invoke(Method m, Object[] args) throws Throwable;
```

In Abb. 4.11 wird eine konkrete Realisierung vorgestellt. Hier leitet der PermissionGranter Aufrufe an die geschützten Membermethoden weiter. Damit dieser Aufruf Erfolg haben kann, muss der PermissionGranter Code innerhalb des erforderlichen Zugriffbereiches liegen, d. h. bei protected Members reicht ein im Package befindlicher PermissionGranter. In einer anderen Umsetzung kann ein applikationsweiter PermissionGranter eingerichtet werden, der die Zugriffsrechte[21] geschützer Attribute und Methoden ändern kann. Dadurch können auch nur im Class-Format vorliegende Applikationsteile unterstützt werden. Spezifiziert wird der für ein Modellelement zu verwendende PermissionGranter ebenfalls durch einen TaggedValue.

4.2.4 Dialogschicht

Task- und Objektmodell bilden zusammen das Domänenmodell. Es beschreibt, aus Sicht des Benutzers, die Struktur und das Verhalten der in der Domäne vorkommenden Entitäten und wie diese zur Durchführung von Aufgaben eingesetzt werden können. Es beschreibt jedoch nicht, in welchem Zusammenhang der Benutzer welche Aufgaben durchführen kann, wie er die Eingangsdaten einer Aufgabe erhält und wie er Ergebnisse einer Aufgabendurchführung weiterverwenden kann. Dies wird im Dialogmodell beschrieben, das in Vesuf auf die Beschreibung der groben Dialogsteuerung beschränkt ist. Die Autoren orientieren sich dabei an [Anderson 2000a], der UML-Statecharts zur Modellierung der Dialogsteuerung vorschlägt. Zukünftige Versionen von Vesuf sollen ein allgemeineres Konzept zur abstrakten Dialogbeschreibung erhalten, das dann auf verschiedene Weisen implementiert werden kann. So sollen zukünftig auch weitere Dialogbeschreibungstechniken (siehe Abschnitt 2.2.3) unterstützt werden, z. B. Petrinetze.

4.2.4.1 Dialogmodell

Die wichtigsten Elemente in UML-Statecharts sind Zustände und Transitionen, welche die möglichen Übergänge zwischen einzelnen Zuständen repräsentieren. In der zur Zeit in Vesuf implementierten Statechartsemantik repräsentieren Zustände einen Ausschnitt der Benutzungsschnittstelle, der zu einem bestimmten Zeitpunkt für den Benutzer zugänglich ist. Die Transitionen zwischen diesen Dialogzuständen repräsentieren die Navigationsmöglichkeiten, die sich dem Benutzer bieten.

Um die Dialogzustände mit Entitäten aus dem Domänenmodell zu verbinden, ordnen die Autoren Zuständen entsprechende Elemente zu (stateobjects).

[20] Zur Realisierung der typfreien Methoden wurde auf Java-Reflection zurückgegriffen.

[21] Seit dem JDK-1.2 gibt es das Package javax.accessibility, mit dem Rechte in Abstimmung mit dem SecurityManager verändert werden können.

Üblicherweise wird man einen Zustand zur Durchführung einer Aufgabe modellieren, es können aber auch beliebige andere Elemente des Domänenmodells einem Zustand zugeordnet werden (z. B. Objekte oder Assoziationen). Damit wird der Typ der Domänenelemente festgelegt, die in einem Zustand bearbeitet werden können. Als Transitionsanschriften werden Pfade (siehe Abschnitt 4.2.2.2) spezifiziert, die einen Datenfluss zwischen den Zuständen modellieren. Dieses Konzept ähnelt den Task-Object-Charts [Fähnrich 1995], wobei diese nur als informales Analysewerkzeug eingesetzt werden, während die in Vesuf implementierte Statechartsemantik dank Transitionsanschriften in VEPL zur formalen Spezifikation beliebig komplexer Datenflüsse geeignet ist.

Zum bedingten Auslösen von Zustandsübergängen können zu Transitionen Guards und Trigger spezifiziert werden. Guards sind Einschränkungen, ähnlich wie Constraints, welche die Zulässigkeit eines Zustandsübergangs an eine Bedingung knüpfen. Trigger dienen dazu, Transitionen gezielt durch externe Ereignisse wie z. B. Benutzereingaben auszulösen.

4.2.4.2 Implementation und Laufzeitverhalten

Als Metamodell der statechartbasierten Dialogmodellierung dienen die Klassen des Packages StateMachines der BehavioralElements des UML-Metamodells. Die Vesuf spezifischen Erweiterungen (Stateobjects und Transitionsanschriften) werden durch TaggedValues repräsentiert und erfordern keine Änderungen am UML-Metamodell. Das Laufzeitverhalten orientiert sich an der in [OMG 2000a] beschriebenen Semantik.

Die Ausführung der Dialogmodelle zur Laufzeit erfolgt nicht durch eine einzige Interpreterkomponente, sondern wird als Kollaboration vieler einzelner Instanzelemente realisiert. Zu jeder Klasse des Metamodells implementieren die Autoren zusätzlich ein Laufzeitelement, welches das dynamische Verhalten des jeweiligen Elements beinhaltet (vgl. Abschnitt 4.1.3). So besitzt eine Transitionsinstanz eine `fire()` Methode, die alle notwendigen Schritte für einen Zustandsübergang durchführt. Diese Schritte beinhalten eine Überprüfung vorhandener Guards, das Verlassen des Ausgangszustands und den Eintritt in den Zielzustand. Innerhalb der `fire()` Methode muss auch der Datenfluss berücksichtigt werden der beim Zustandsübergang stattzufinden hat. Dazu wird der im Modell spezifizierte Transitionspfad anhand des aktuellen Kontexts ausgewertet, um das zugeordnete Instanzelement für den Folgezustand zu bestimmen. Zustandsinstanzen besitzen `enter()` und `exit()` Methoden, welche die im Modell spezifizierten Entry- und Exitaktionen initiieren. Zustandsübergänge können von Eventinstanzen angestoßen werden. Eine Eventinstanz sucht in ihrer `trigger()` Methode im Rahmen eines vorgegebenen Kontexts selbständig eine aktivierbare Transitionsinstanz. Dabei werden lokale Transitionen globalen vorgezogen. Ist eine geeignete Transitionsinstanz gefunden, löst die Eventinstanz einen Zustandsübergang durch Aufrufen der `fire()` Methode aus.

Die Verwaltung der einzelnen Instanzelemente zu einer Statemachine übernimmt eine Statemachineinstanz. Aktuell berücksichtigt Vesuf keine Nebenläufigkeit innerhalb einer Statemachineinstanz. Nebenläufigkeit in Benutzungsschnittstellen kann dennoch ausgedrückt werden, da in einer Vesuf Applikation zu jeder Zeit beliebig viele unabhängige Statemachineinstanzen existieren dürfen. Eine Koppelung der verschiedenen Statemachineinstanzen ist zur Zeit nicht möglich. Ein ähnliches Konzept zur Nebenläufigkeit in der Dialogsteuerung wur-

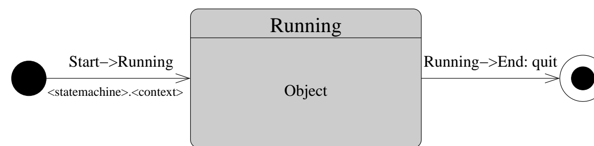


Abbildung 4.12: Standarddialogmodell

de bereits 1985 vorgestellt (siehe [Jacob 1985, Jacob 1986]). Die aktiven Statemachineinstanzen einer Vesuf Applikation werden als Control-Flows bezeichnet und von der sog. NavigationRuntime verwaltet. Die NavigationRuntime einer Vesuf Applikation stellt damit eine Instantiierung des Dialogmodells dar.

4.2.4.3 Standarddialogmodell

Für den häufig auftretenden Fall, dass eine Applikation lediglich einen einzigen Dialog besitzt, definiert das Vesuf System bereits ein Standarddialogmodell, das sogenannte DefaultDialogModel (s. Abb. 4.12). Es besitzt einen einfachen Zustand (Running), dem über die initiale Transition als Stateobject das Kontextobjekt der Dialogsteuerung zugewiesen wird. Das Kontextobjekt kann im Applikationsmodell (s. Abschnitt 4.2.2.1) über die Eigenschaft `initobject` angegeben werden. Zum Beenden der Applikation besitzt das DefaultDialogModel einen Endzustand, der über den Event `quit` erreicht werden kann.

4.2.5 Präsentationsschicht

Die Präsentationsschicht des Vesuf Systems ist für das konkrete Erscheinungsbild einer Benutzungsschnittstelle verantwortlich. Die Spezifikation eines Präsentationsmodells erfolgt auf Basis des nachfolgend präsentierten Metamodells. Um ein Präsentationsmodell zu beschreiben, kann UIML oder UIML-Shorthand (siehe Abschnitt 2.2.2.3) verwendet werden.

4.2.5.1 Das Präsentationsmetamodell

Das in Vesuf eingesetzte Präsentationsmetamodell ist in Abbildung 4.13 dargestellt. Um eine nahtlose Verbindung zum UML Metamodell zu ermöglichen, wurden alle Elemente als Verfeinerungen der Klasse ModelElement aus dem Package `uml.foundation.core` konzipiert.

Grundlage für sämtliche User Interface Elemente ist das Part Element. Da es von ModelElement abgeleitet ist, erbt es unter anderem den flexiblen Erweiterungsmechanismus mit Tagged Values, der intensiv für die Beschreibung von toolkitspezifischen Eigenschaften ausgenutzt wird. Die Autoren haben die weiteren Einheiten des Metamodells anhand einer Identifizierung verschiedener funktionaler Rollen der Elemente einer Benutzungsschnittstelle definiert.

Ein Delegate repräsentiert eine Einheit aus dem Präsentationsmodell zur Darstellung und Manipulation eines Elements aus einem anderen Submodell. In vielen Fällen werden Delegates daher durch Pfade mit Elementen aus dem Domänenmodell verknüpft. Außerdem werden Delegates auch mit Elementen des Dialogmodells (z. B. Events) assoziiert, um Benutzern Steuerungsmöglichkeiten anbieten zu können. Die Verbindung eines Delegates zu einem Element

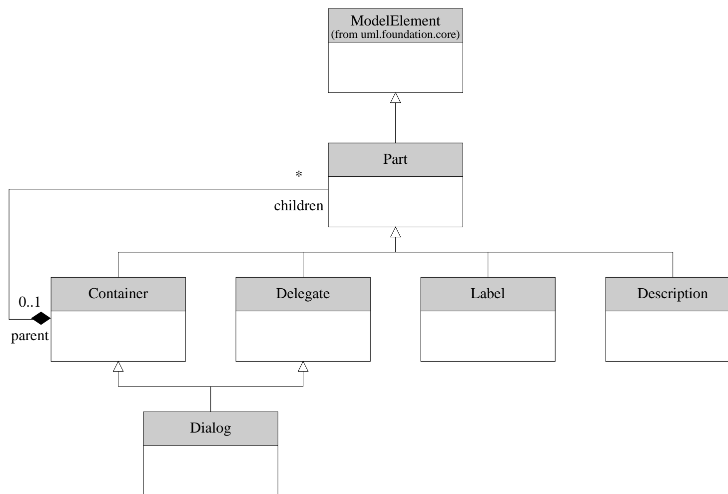


Abbildung 4.13: Präsentationsmetamodell als UML Klassendiagramm

impliziert eine Verantwortlichkeit für gewisse Mechanismen zur Synchronisierung zwischen den beiden Seiten zur Laufzeit, d. h. eine Delegateinstanz muss Daten zur Darstellung vom Element holen und zur Speicherung wieder zurückschreiben können.

Container sind Behälter für beliebig viele Subparts. Durch Container wird es möglich, eine Oberfläche in Form einer hierarchischen Struktur zu beschreiben. Werden zusätzlich auch hier Synchronisationsmechanismen zur Laufzeit benötigt, können Dialoge verwendet werden. Dialoge sind verantwortlich für alle enthaltenen Subparts und können diese auffordern, ihren Zustand zu erneuern oder zu speichern. Die Trennung dieser beiden Formen von Behältern ist sinnvoll, da je nach Kontext unterschiedliche Anforderungen bestehen. Beispielsweise ist der Einsatz von Dialogen in einem interaktiven User Interface wie AWT angebracht, da dort Änderungen lokal wirksam werden. In einem statischen Umfeld unter HTML ist jedoch eher die Anwendung von Containern angezeigt, weil Benutzerinteraktionen (requests) stets zu einer Auswertung aller Parts der dargestellten Oberfläche führen.

In vielen Fällen werden die funktionalen Komponenten eines User Interfaces (delegates) durch einen Text näher erläutert. Diese Elemente sind im Metamodell unter dem Begriff Labels aufgeführt. Eine ausführliche Beschreibung zu einem Element kann mit Hilfe von Descriptions angelegt werden. Descriptions können z. B. von einem Hilfesystem dazu verwendet werden, kontextsensitive Hilfefunktionalitäten bereitzustellen.

4.2.5.2 Präsentationselemente zur Laufzeit

Die Elemente des dynamischen Teils des Präsentationsmetamodells inklusive ihrer toolkitspezialisierten Ausprägungen sind in Abb. 4.14 dargestellt. Wie in Abschnitt 4.1.3 bereits verdeutlicht wurde, findet in den meisten Fällen eine exakte Spiegelung von Modellelementen auf Laufzeitelemente statt. Auch das Präsentationsmetamodell bildet dabei keine Ausnahme und wird in dieser Art und Weise in die Laufzeitebene abgebildet. Die Struktur der in Abb. 4.14 ge-

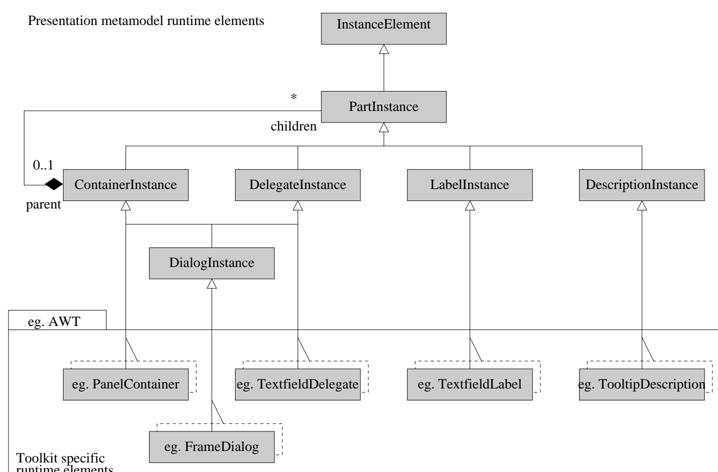


Abbildung 4.14: Struktur der Präsentationselemente zur Laufzeit

zeigten Laufzeitebene entspricht daher im oberen Bereich genau der Struktur des in Abb. 4.13 vorliegenden statischen Metamodells. Die Verbindung zwischen entsprechenden Elementen der unterschiedlichen Ebenen besteht in der bereits erläuterten Form.

Zusätzlich befinden sich im unteren Teil der Abbildung jeweils mehrere spezialisierte Elemente zu jeder generischen Metamodellklasse. Diese Elemente sind die konkreten Entitäten, mit denen die Benutzungsschnittstelle zur Laufzeit aufgebaut wird. Sie besitzen eine Verbindung zu einem spezifischen Toolkitelement (nicht in der Abb. gezeigt), das sie zur Darstellung einsetzen. Um ein spezielles Toolkit im Vesuf System verwenden zu können, müssen entsprechende Spezialisierungen der allgemeinen Laufzeitelemente entworfen werden.

Elemente, für die es notwendig ist, Daten mit dem Domänenmodell auszutauschen, benutzen zu diesem Zweck Pfadinstanzen. Dadurch werden die Darstellungselemente auch über Änderungen an übergeordneten Elementen benachrichtigt und können sich aktualisieren (für Einzelheiten siehe Abschnitt 4.2.3.3). Dieser Mechanismus ermöglicht, dass einem Dialog zugrunde liegende Objekte einfach ersetzt werden können. Die Präsentation reagiert darauf, indem alle betroffenen Präsentationseinheiten benachrichtigt werden und ihre Darstellungen anpassen. Damit Änderungen an Domänenobjekten überhaupt für das Vesuf System sichtbar werden und der dargestellte Mechanismus greifen kann, ist eine geringe Modifikation an der Applikationsimplementation notwendig. Jede Änderung einer im Modell erfassten Einheit, wie z. B. eines Attributes, muss durch einen Event bekanntgemacht werden. Einzelheiten zum Vesuf Eventhandling finden sich in Abschnitt 4.2.2.4.

Abhängigkeiten zwischen verschiedenen Modellelementen können durch Dependencies modelliert werden. Präsentationselemente der abhängigen Entitäten werden dadurch auch in Kenntnis gesetzt, wenn sich die Quelle der Abhängigkeit ändert. Indirekt können auf diese Weise Abhängigkeiten zwischen verschiedenen Präsentationselementen ausgedrückt werden.

Zur Laufzeit des Programms, werden Instanzen der Parts auf Anforderung des Interpreters erzeugt. Um die Partinstanzen zu erhalten, wird eine im Präsen-

```
public void handleAction(Object args)
{
  try
  {
    action(args);
    this.valid = true;
    this.error = null;
  }
  catch(Exception e)
  {
    this.valid = false;
    this.error = e;
    handleError(e);
  }
}
```

Abbildung 4.15: Sichere Methode

tationsmodell festgelegte Bausteinfabrik verwendet. Diese bestimmt durch eine Abbildung der zugrundeliegenden Metamodellklasse (Dialog, Delegate, ...) und des optional definierten Widgettyps ein angemessenes Laufzeitelement. Begreift man die Kombination der erwähnten Eigenschaften als abstrakten Widgettyp, so handelt es sich bei diesem Schritt um ein abstract-concrete Widgetmapping. Gesteuert wird diese Abbildung durch ein separates Property File. Dadurch sind Konfigurationsänderungen und Erweiterungen der Widgetbibliotheken unproblematisch. Wurde im Präsentationsmodell der Widgettyp eines Elements nicht angegeben, kann die Fabrik auf ein Automatisierungswerkzeug zurückgreifen. Dieses versucht durch eine Auswertung bestimmter Eigenschaften ein passendes Element auszuwählen.

Implementationsdetails der Laufzeitelemente Die Laufzeitelemente der Präsentationsschicht wurden gemäß der Visual Proxy Architektur (siehe Abschnitt 2.1.6) realisiert. Damit Präsentationselemente Applikationsdaten darstellen und zurückschreiben können, besitzen sie durch die Pfadinstanzen einen allgemeinen Zugriffsmechanismus. Die so erreichbaren Daten liegen allerdings in einem systeminternen Format vor, das zur Ausgabe nicht zwangsläufig geeignet sein muss. Z. B. kann intern mit Gleitkommazahlen operiert werden, wohingegen die Darstellung im Präsentationselement eine Zeichenkette verlangt. Damit ist eine Umwandlung der Datenformate in beide Richtungen notwendig. Die Autoren führen zu diesem Zweck Konverter ein, die genau diese Konversionsanforderungen erfüllen. Da die Präsentationselemente mit beliebigen Konvertern ausgestattet werden können, bleiben sie unabhängig von der internen Datenrepräsentation.

Zur Instantiierung der Präsentationselemente verwendet Vesuf Factorykomponenten. Diese erzeugen nach vorgegebenen Eigenschaften wie Klasse (z. B. Delegate, Label) oder Widgettyp die entsprechenden konkreten Interaktionselemente.

Neben Darstellungsaspekten müssen die Präsentationselemente auch mit beliebigen Benutzereingaben zurechtkommen. Die Autoren haben daher eine sichere Methode (sandbox) eingerichtet, die jedes Element zur Behandlung von

Benutzerinteraktionen aufruft. Die Implementation der sicheren Methode ist in Abb. 4.15 dargestellt. Die eigentliche Aktion, also z. B. der Aktualisierungsversuch eines Attributes, wird durch den Aufruf `action(args)` gestartet. Schlägt die Aktion fehl, wird der Fehler abgefangen und kann durch Delegation in der `handleError(e)` Methode weiterbehandelt werden. Die Methoden `action(args)` und `handleError(e)` werden durch die Präsentationselemente in geeigneter Weise überschrieben.

Es existieren drei grundsätzlich verschiedene Fehlertypen, die bei der Behandlung von Benutzereingaben auftreten können. Erstens können die eingegebenen Daten unzulässig sein, d. h. sie haben z. B. den falschen Typ oder liegen ausserhalb des erlaubtes Wertebereichs. Diese Art von Fehler wird erkannt, wenn versucht wird, den neuen Wert zuzuweisen (innerhalb der action Methode). Bevor ein neues Datum akzeptiert wird, werden die Constraints des betroffenen Laufzeitelements ausgewertet. Verursachen die Constraints einen Fehler greift der oben beschriebene Mechanismus zur Fehlerbehandlung.

Zweitens ist es möglich, dass innerhalb einer durch einen ImplementationAccessor aufgerufenen Zugriffsmethode ein Fehler auftritt, z. B. wenn eine angebundene Datenbank nicht erreichbar ist. Der ImplementationAccessor reagiert darauf, indem er einen Zugriffsfehler weitermeldet (AccessException).

Drittens kann es passieren, dass die ImplementationAccessors auf ein Laufzeitelement nicht zugreifen können, da z. B. die Zugriffsmethode in der Implementation nicht gefunden werden konnte oder der Sichtbarkeitsbereich für einen Zugriff nicht ausreichend war. Die ImplementationAccessors propagieren daraufhin wiederum einen Zugriffsfehler. Im Gegensatz zu den ersten beiden Fehlertypen deutet dieser Fall stark auf einen Implementations- oder Modellierungsfehler hin.

Beschreibung eines konkreten Präsentationmodells In Abb. 4.16 ist ein Ausschnitt aus dem konkreten Präsentationsmodell des Shape-Beispiels in UIML2-Shorthand Notation dargestellt. Das konkrete Präsentationsmodell enthält die Hauptdialoge des Systems als hierarchisch bis in sämtliche Einzelheiten spezifizierte Interaktionselemente.

Der Head-Bereich (Zeilen 3-8) wird verwendet, um einige globale Eigenschaften des Modells durch Metatag-Einträge festzuhalten. Dazu gehört der Name des Modells, der in diesem Fall auf `Shape_awt` gesetzt wird. Wie der Name schon andeutet, wird ein User Interface Modell für das Java-AWT Toolkit definiert. Des weiteren wird die Quelle des dem Präsentationsmodell zugeordneten Domänenmodells mit `shape.ShapeDomainModel` beschrieben. Diese Angabe ist optional, da diese Zuordnung durch Applikationsdeskriptoren (siehe Abschnitt 4.2.2.1) hergestellt wird. Wird aber wie im Beispiel das Domänenmodell spezifiziert, so kann bereits während der Konvertierung des Präsentationsmodells überprüft werden, ob die angegebenen Pfade grundsätzlich mit der Modellwelt in Einklang sind. Fehlerhafte Pfadangaben können auf diese Weise schon vor Ausführung der Anwendung entdeckt werden. Der letzte Metatag-Eintrag bestimmt, dass zur Erzeugung der Laufzeitelemente für die Präsentation die AWT-Factory verwendet werden soll.

Im Structure-Bereich (Zeilen 11-53) findet sich eine Beschreibung der einzelnen Dialoge des Systems. Da das Shape-Beispiel allerdings nur einen Hauptdialog besitzt, bleibt dieser Bereich recht überschaubar. Zusätzlich wurde aus Platz-

```
01: <uiml>
02:
03:   <head>
04:     <meta name="Name"        content = "Shape_awt"/>
05:     <meta name="DomainModel" content = "shape.ShapeDomainModel"/>
06:     <meta name="Factory"
07:         content = "org.vesuf.runtime.presentation.awt.Factory"/>
08:   </head>
09:
10:   <interface name="Dialog for Shape Objects">
11:     <structure>
12:
13:       <Dialog name = "SampleDialog"
14:               path = "SampleTask"
15:               widgettype = "Frame"
16:               layout = "GridBagLayout()"
17:               background = "Color.lightGray"
18:               width = "400"
19:               height = "100"
20:               closingevent = "quit" >
21:
22:         <Dialog name = "A Point"
23:                 path = "Point"
24:                 widgettype = "Panel"
25:                 layout = "GridLayout(0,2)"
26:                 layoutconstraints
27:                     = "GridBagConstraints(0,0,1,1,...)">
28:
29:           <Label    name = "X_label"
30:                     path = "X"
31:                     widgettype = "Label"/>
32:           <Delegate name = "X_delegate"
33:                     path = "X"
34:                     widgettype = "TextField"/>
35:
36:           <Label    name = "Y_label"
37:                     path = "Y"
38:                     widgettype = "Label"/>
39:           <Delegate name = "Y_delegate"
40:                     path = "Y"
41:                     widgettype = "TextField"/>
42:         </Dialog>
43:
44:         <Dialog name = "A Rectangle"
45:           ...
46:         </Dialog>
47:
48:         <Dialog name = "A Circle"
49:           ...
50:         </Dialog>
51:
52:       </Dialog>
53:     </structure>
54:   </interface>
55: </uiml>
```

Abbildung 4.16: Konkretes Präsentationsmodell des Shape-Beispiels

gründen darauf verzichtet, die dem Punktdialog sehr ähnlichen Beschreibungen der Dialoge anderer geometrischer Formen wie Rechteck oder Kreis detailliert aufzuführen (Zeilen 44-50).

Der Hauptdialog des Beispiels ist dazu gedacht, den `SampleTask` des Domänenmodells zu repräsentieren. Er besitzt daher den Namen `SampleDialog` (Zeile 13) und den Pfad `SampleTask`, der das SampleTask Objekt referenzieren wird. Damit dieser Dialog als ein Fenster dargestellt wird, wurde der Widgettyp auf `Frame` festgelegt (Zeile 15). Bis auf eine Ausnahme dienen alle weiteren Angaben des Dialogelements dazu, Eigenschaften des Fensters zu spezifizieren. Dazu gehören das Layout (Zeile 16), die Hintergrundfarbe (Zeile 17) und die Ausmaße (Zeilen 18, 19). In Zeile 20 wird der Schließmechanismus des Fensters mit einen Event aus dem Dialogmodell assoziiert. Versucht der Benutzer das Fenster zu schließen, wird in der Dialogsteuerung der `quit` Event ausgelöst.

Zusammengesetzt ist der Hauptdialog aus einer Anzahl Teilbereiche für je eine geometrische Form. Der Container für die Oberflächenelemente des Punkts erhält den Namen `A Point` (Zeile 22) und den Pfad `Point`. Die Pfadangabe ist dabei relativ zum übergeordneten Element und wird bei der Konvertierung zu `SampleTask.Point` expandiert. Die Eigenschaften des Punktes werden durch Delegates repräsentiert (Zeilen 32-34 und 39-41). Um die Verbindung mit der Domäne herzustellen, erhalten sie die relativen Pfadangaben auf die Attribute X und Y (Zeilen 33, 40). Beide Attribute werden als Textfelder (Zeilen 34, 41) abgebildet und erhalten weiterhin textuell vorangestellte Kurzbeschreibungen, die als Labels realisiert wurden (Zeilen 29-31 und 36-38).

4.3 Konzeptuelle Probleme und Lösungsansätze

Die Autoren beschreiben im Folgenden zwei grundsätzliche Probleme, die im Zusammenhang mit dem modellbasierten Ansatz und UML auftreten. Es ist für die Konstruktion eines vollständigen Systems notwendig, sich mit diesen Problemen auseinanderzusetzen.

4.3.1 Das Abbildungsproblem

Ziel jeder MB-UIDE ist es, für eine deklarative Spezifikation eines User Interfaces eine ausführbare Version zu erzeugen. Untersucht man die Elemente der Submodelle einer User Interface Spezifikation, lässt sich eine natürliche Einteilung in abstrakte und konkrete Elemente finden. Mit konkreten Elementen werden Einheiten bezeichnet, die direkter Teil der ausführbaren Benutzungsschnittstelle sind, z. B. Widgets und Dialoge. Abstrakte Elemente sind nicht direkt vom Anwender modifizierbar, sondern hinter der Oberfläche verborgen, z. B. Aufgaben und Objekte. Die Erzeugung eines ausführbaren User Interfaces kann unter diesem Aspekt als Aufgabe verstanden werden, die zu einer abstrakten Spezifikation eine konkrete Version sucht, z. B. für eine Aufgabe den richtigen Dialog bestimmt. Puerta und Eisenstein bezeichnen die Problematik der Verbindungen zwischen abstrakten und konkreten Elementen als das Abbildungsproblem (*mapping problem*):

> „We call the problem of linking abstract and concrete elements in an interface model the mapping problem. Solving the mapping problem

> in a general sense is essential for the construction of model-based systems of wide applicability in user interface design.“
> [Puerta Eisenstein 1999, S. 1]

Die mangelnde Flexibilität vieler modellbasierter Systeme (siehe dazu Abschnitt 7) rührt in den meisten Fällen davon her, dass die Abbildungen zwischen abstrakten und konkreten Elementen nicht allgemein genug gelöst wurden. Für ein modellbasiertes System ohne Einschränkungen der Anwendbarkeit ist es notwendig, wichtige Abbildungen einstellbar zu machen. Puerta beschreibt drei verschiedene Arten der Abbildung von abstrakten auf konkrete Einheiten. Im folgenden wird auf diese Varianten und die in Vesuf realisierten Lösungen eingegangen.

Task-Dialog Mapping

Ein Task- und ein Dialogmodell enthalten in der Regel Informationen, die einander in gewisser Weise verwandt sind. Identifiziert wurde eine starke strukturelle Ähnlichkeit zwischen der Ausführungsreihenfolge der Aufgaben und der Dialognavigation. Des weiteren wurde festgetellt, dass auch zwischen den Bedingungen für die Taskausführung und dem Zugriffsstatus der Interaktionselemente (enabled / disabled) ein Zusammenhang besteht.

Die Ähnlichkeiten zwischen den beiden Modellen können dazu genutzt werden, Teile eines Modells aus dem anderen abzuleiten, d. h. gewisse Strukturen automatisiert zu erstellen. Denkbar ist z. B. die Generierung der Dialognavigation aus der Ausführungsreihenfolge der Aufgaben. So kann aus dem Aufgabenmodell zumindest ein initiales Dialogmodell erstellt werden (siehe auch Abschnitt 3.2.6).

In Vesuf wird derzeit die Verknüpfung zwischen Aufgaben und Dialogzuständen durch die Definition des Dialogmodells hergestellt. Zu diesem Zweck werden Zustände des Dialogmodells mit Domänenelementen (Aufgaben und Objekte) verbunden. Da das Taskmodell zur Zeit keine prozeduralen Informationen enthält, kann es nicht zur automatischen Generierung der Dialogstruktur benutzt werden.

Task-Presentation Mapping

Vergleicht man das Aufgabenmodell mit dem Präsentationsmodell, lässt sich eine Ähnlichkeit in bezug auf ihre hierarchische Ordnung feststellen. Daraus lässt sich auf eine Verbindung zwischen der Task/Subtask und der Part/Subpart Hierarchie schließen. Diese Verwandtschaft kann die Umsetzung eines automatisierten Layoutalgorithmus durch die Abbildung der Task/Subtask Ordnung auf die Benutzungsschnittstellenstruktur ermöglichen.

Da Aufgaben in Vesuf, ebenso wie Klassen, als Elemente, die Instanzen klassifizieren, betrachtet werden (classifier), unterscheidet sich die Abbildung der Aufgabeninstanzen auf Präsentationselemente nicht von der Abbildung von Objekten auf Präsentationselemente.

Object-Presentation Mapping

Das Objektmodell enthält Entitäten, deren Eigenschaften und Verhalten durch Elemente des Präsentationsmodells in Form eines User Interfaces sichtbar ge-

macht werden. Diese Verbindung zwischen interner und externer Repräsentation kann ausgenutzt werden, um die Präsentation in Abhängigkeit von den Domänenentitäten zu erzeugen. Dazu müssen die darzustellenden Informationen und die Art der Darstellung dieser Informationen bestimmt werden. Um ein geeignetes Widget auszuwählen, können die zusätzlichen Informationen über die Attribute (z. B. Datentyp oder Constraints) ausgewertet werden, wie z. B. in [Eisenstein Puerta 2000] beschrieben.

Vesuf realisiert die Abbildung von einem Domänenelement auf ein Präsentationselement völlig flexibel. Dazu werden Abbildungskomponenten (presentation mapper) eingesetzt, die eine Funktion von einem Tupel, bestehend aus Dialogzustand und Domänenelement, auf ein Präsentationselement realisieren. Durch diese Form der Abbildung ist sichergestellt, dass verschiedene Präsentationselemente in verschiedenen Dialogzuständen ausgewählt werden können. Welche Abbildungskomponente für eine Applikation eingesetzt wird, kann im Applikationsmodell festgelegt werden (siehe Abschnitt 4.2.2.1). Damit sind beliebig komplexe Regeln leicht in das System integrierbar.

Eine weitere Abbildung wird automatisch vollzogen, wenn zu konkreten Präsentationselementen kein Widgettyp angegeben ist. Vom Interface Designer definierbare Regeln bestimmen aus Datentyp und Constraints ein geeignetes Widget.

4.3.2 Das Assoziationsproblem

Assoziationen sind ein zentrales Instrument von UML zur Beschreibung von Beziehungen in einem Klassendiagramm (siehe Abschnitt 2.2.4). Sie besitzen mindestens zwei Assoziationsenden, die jeweils eine Verbindung zu einem klassifizierenden Element (classifier), also z. B. zu einer Klasse herstellen. Die Assoziationen selbst sind nicht einem der angeschlossenen Elemente zugeordnet, sondern werden lösgelöst direkt im Modell verwaltet.

Die Autoren haben gemäß dem in Abschnitt 4.1.3 vorgestellten Muster der Modell- und Laufzeitebenentrennung dynamische Elemente für Assoziationen entwickelt und sind dabei auf mehrere technische und konzeptionelle Probleme gestoßen. So bleibt in der UML-Spezifikation [OMG 2000a] die Semantik für n-näre Assoziationseigenschaften wie Multiplizität, Aggregation, Navigierbarkeit, Veränderlichkeit und qualifizierende Attribute unklar. Dennoch ist es bei Vernachlässigung dieser Aspekte möglich, eine rudimentäre Laufzeitunterstützung aufzubauen. Die Autoren haben jedoch empirisch feststellen müssen, dass die Verwendung von Verbindungen als eigenständiges, vom Benutzer direkt manipulierbares Element in Benutzungsschnittstellen ungewöhnlich ist. Trotz einiger Überlegungen haben sie keine Anwendungsbeispiele für derartige Elemente identifizieren können. Der einzige Ansatzpunkt für die Verwendung von separaten Assoziationen scheint in der Möglichkeit begründet, globale Anfragen auf der Menge aller Verbindungen durchführen zu können. Um Verbindungen zu modifizieren, wird jedoch die Objektsicht präferiert, da die Manipulation von Verbindungen zu unbeabsichtigten Anomalien führen kann. Die MOF-Spezifikation [OMG 2001] greift diese Problematik auf und führt mit den sogenannten Referenzen ein Konzept ein, das Assoziationen aus Sicht der Objekte heraus zugreifbar macht.

4.4 Ausblick und Erweiterungen

Um das Vesuf System möglichst flexibel in verschiedenen Anwendungsdomänen einsetzen zu können, ist es essentiell, die aus der Aufgabenanalyse gewonnenen Informationen in einem vollständigen Aufgabenmodell ausdrücken zu können. In diesem Zusammenhang ist auch die geeignete Abbildung von Geschäftsprozessen (Workflows) in die Aufgabenebene zu berücksichtigen. Die Autoren sind daher stark daran interessiert, ein ausdrucksstarkes eigenständiges Aufgabenmodell in die Vesuf Modellfamilie zu integrieren. Allerdings ist dabei unbedingt die UML-Konformität weiterhin so weit wie möglich aufrechtzuerhalten. Eine geeignete Aufgabenbeschreibungstechnik stellen nach Meinung der Autoren die ConcurTaskTrees dar (siehe Abschnitt 2.2.4.1), da sie einerseits formal, ausdrucksstark und skalierbar sind und es andererseits Bemühungen seitens ihrer Entwickler gibt, sie in den UML-Standard einzubinden [Paternò 2000].

Die Autoren streben weiterhin an, ein abstrakteres Dialogmetamodell in Vesuf zu integrieren und nicht direkt auf Statecharts aufzusetzen. So wäre es ausreichend, Dialogzustände zugreifbar zu machen und die interne Repräsentation flexibel zu halten. Auf diese Weise wäre es möglich, verschiedene Implementationen der Dialogschicht wie z. B. Dialognetze oder auch Zustandsdiagramme einzusetzen.

Im Hinblick auf die Präsentation wäre es wünschenswert, einen Standard zu entwicklen und in UML zu integrieren. Einen interessanten Ansatz verfolgen [da Silva Paton 2000b] mit der Entwicklung von UML*i*. Sie erweiteren dazu UML um Notationen und Diagramme zur Beschreibung von User Interfaces. Es bleibt allerdings abzuwarten, inwieweit sich diese Vorschläge in den nächsten Versionen des UML-Standards wiederfinden.

Neben der Weiterentwicklung der einzelnen Modelle ist für die Praxistauglichkeit des Systems insbesondere auch die direkte Anbindung an verschiedene Implementationstechniken von großem Interesse. So wäre eine Integration von Zugriffsmechanismen auf Basis von z. B. JDBC, XML/DOM und WSDL/SOAP wünschenswert.

Um die einfache Anwendbarkeit und damit die Akzeptanz des Vesuf Systems zu verbessern, können ein Reihe von Erweiterungen im Bereich der Werkzeugunterstützung vorgenommen werden. In der Kategorie der design-time Tools ist es in erster Linie wichtig, ein Werkzeug für das visuelle Editieren von Benutzungsschnittstellen anzubieten. Dadurch ist gewährleistet, dass User Interface Designer mit den gewohnten Hilfsmitteln arbeiten können und keine unbekannte Schnittstellenbeschreibungssprache neu erlernen müssen. Interessant sind ferner Werkzeuge zur Unterstützung des Designprozesses. So können Komponenten entwickelt werden, die dem Entwickler helfen sollen, schlechtes Design frühzeitig zu erkennen (design critics). In einer weiteren Stufe würden derartige Tools nicht nur das existierende Design kritisieren, was unter Umständen leicht zum Verdruss der Betroffenen führt, sondern auch Ratschläge geben, wie eine bessere Lösung gefunden werden kann (design advisors). Außerdem können Werkzeuge entwickelt werden, die aus den Informationen bereits spezifizierter Modelle zumindest den Initialzustand anderer Modelle inferieren. So ist es z. B. möglich, ein initiales Dialogmodell aus den Daten des Aufgabenmodells zu erzeugen.

Vielfältige Erweiterungen sind auch in der Kategorie der runtime Tools denkbar. So ist es notwendig, die Palette der Automatisierungswerkzeuge zu komplettieren. Insbesondere fehlt für eine vollständige Ausnutzung des slinky au-

tomation Gedankens eine geeignete Komponente zur automatischen Layoutgenerierung. Des weiteren kann ein Werkzeug entwickelt werden, das automatisch ein Domänenmodell aus einer vorgegebenen Implementation ableitet. Auf diese Weise ist die einfache Anbindung von Fremdsystemen möglich. Besonders interessant ist auch die Erstellung eines kontextsensitiven Hilfesystems, das die zur Verfügung stehenden Modellinformationen und den aktuellen Anwendungszustand evaluiert, um einem Anwender bei der Benutzung eines Programms zur Seite zu stehen. Des weiteren besteht die Möglichkeit, Tools zu entwickeln, welche versuchen, die Benutzbarkeit einer Oberfläche qualitativ zu messen (usability analyser). Mit Hilfe dieser Daten kann der Entwickler Rückschlüsse auf das Anwendungsdesign ziehen und Schwachstellen erkennen.

Ein Aspekt verteilter Anwendungen, der gerade auch im Hinblick auf das UbiComp zunehmend an Bedeutung gewinnt, ist die Verteilung der Anwendungskomponenten zwischen Client und Server. Die steigende Heterogenität der Endgeräte führt zu unterschiedlichsten Systemkapazitäten. In Zusammenhang mit schwankenden Übertragungsleistungen spricht dies für die Integration einer dynamischen Verteilungskomponente, die auf die unterschiedlichen Parameter der Umgebung durch Umverteilung der Anwendungsbausteine reagieren kann (apportionment). Vesuf legt einen Grundstein für ein Verteilungskonzept, indem es eine klare Schichtenarchitektur umsetzt. Strategien zur dynamischen Verteilung sind Thema aktueller Forschung [Banavar et al. 2000].

Kapitel 5

Fallstudie Global-Info

In Kapitel 4 haben die Autoren das Vesuf System vorgestellt. Mit Hilfe dieses modellbasierten Systems haben sie unterschiedliche Dienste realisiert und in ein Internetportal integriert. In diesem Kapitel werden die Autoren die einzelnen Dienste präsentieren und ihre Implementation im Kontext des Global-Info Projekts erläutern. Des weiteren wird gezeigt, wie für einen Dienst verschiedene Interfacemodalitäten entwickelt werden können.

5.1 Global Info

Das vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (bmb+f) initiierte Förderkonzept "Globale Elektronische und Multimediale Informationssysteme" (Global Info) hat das Ziel, den grundsätzlichen Wandel in der wissenschaftlichen Informationsinfrastruktur zu unterstützen, indem Wissenschaftlern der effiziente Zugang zu den weltweit vorhandenen elektronischen und multimedialen Informationen, die in verteilten Informationssystemen digital gespeichert sind, von ihrem Arbeitsplatzrechner aus eröffnet werden soll. Ein besonderes Merkmal von Global Info ist, daß alle am Prozess der Bereitstellung von Information Beteiligten, d. h. Autoren, Verlage, Bibliotheken, Fachinformationszentren und Nutzer bei der Gestaltung der neuen Strukturen zusammenwirken und dadurch helfen, Fehlentwicklungen durch einen Ausgleich der Interessen im Vorfeld zu vermeiden. Das Global Info Förderkonzept unterstützt folgende unterschiedliche Teilprojekte: Content Analysis, Retrieval and MetaData: Effective Networking (CARMEN), Dateninteraktives Publizieren, E-Verlage, Werkzeuge für elektronisches Publizieren (WEP) und Infrastruktur [Global Info 2000].

Das Subprojekt „Infrastukturen für digitale Bibliotheken" ist in drei weitere Forschungsschwerpunkte unterteilt. Die Universität Berlin entwickelt einen Benachrichtigungsdienst (Hermes) [Faensen et al. 1999], der die aktive Rolle des Benutzers bei der Informationssuche umzukehren sucht. Der Dienst informiert den Nutzer automatisch über Publikationen, die zu seinem Wissensprofil passen. Des weiteren wird an der Universität Magdeburg an einem Föderationsdienst (Demetrios) [Schallehn et al. 2000] gearbeitet, der seinen Nutzern die Literatur- und Informationsrecherche in heterogenen Informationssystemen ermöglichen soll. Die Arbeitsgruppe Verteilte Systeme (VSYS) des Informatikfachbereiches der Universität Hamburg erarbeitet im Kontext von „The Global Info Brokerage

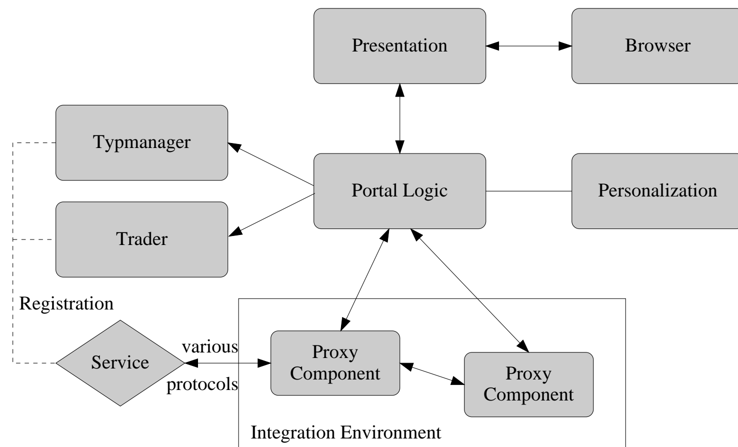


Abbildung 5.1: PublicationPORTAL Architektur (aus [Zirpins et al. 2001])

And Library Trading Architecture“ (GIBRALTAR) [VSYS 2000] Komponenten zur dynamischen Verwaltung und zur Vermittlung von Diensten. Diese Komponenten werden in Form eines Services in einem Internetportal integriert, der seine Nutzer bei der zielgerichteten Auswahl von Diensten unterstützt und eine automatische Vermittlung durchführen kann.

5.1.1 PublicationPORTAL

Das im Kontext von GIBRALTAR aufgebaute PublicationPORTAL beruht technologisch auf dem Jetspeed Portalframework der Apache Group [ASF 2001b] und ergänzt es um zusätzliche Funktionalitäten. In Abbildung 5.1 ist die Systemarchitektur der GIBRALTAR Implementation dargestellt. Wesentliche Bausteine der Architektur sind die Typmanager- und die Traderkomponenten, die in Kombination eine Dienstvermittlung realisieren. Der Typmanager basiert auf einem komponentenbasierten Typsystem [Häming 2000] und erlaubt die Administration von generischen Dienstspezifikationen. Der auf dem Typmanager aufgesetzte Trader verwaltet dynamisch die zur Verfügung stehenden Dienste und ist in der Lage, Anfragen nach bestimmten Services in optimierter Form zu bearbeiten. Dienste, die nur entfernt zugänglich sind, werden innerhalb einer Integrationsumgebung durch komponentenorientierte Serviceproxies repräsentiert, die auf einer erweiterbaren Menge von Adaptern für verschiedene Protokolltypen wie IIOP, SOAP oder HTTP basieren. Die eigentliche Portalfunktionalität ist in der Portallogik Komponente gekapselt. Sie greift auf die weiteren Komponenten, wie Typmanager, Trader und Integrationsumgebung zurück, um die erweiterten Möglichkeiten, wie z. B. die Dienstvermittlung zu realisieren. Zusätzliche Funktionalitäten wie beispielsweise einige Aspekte der Personalisierung werden dafür vorgesehen, von externen Dienstanbietern umgesetzt zu werden [Zirpins et al. 2001].

5.1.2 Integration von Vesuf in das PublicationPORTAL

Um die Praxistauglichkeit des Vesuf Systems nachzuweisen, setzen die Autoren es als Teil der Präsentationskomponente (vgl. Abb. 5.1) des PublicationPORTALs ein. Der Zugriff auf die zu integrierenden Dienste erfolgt dabei über in Java zu implementierende Proxy-Komponenten. Das Vesuf System wird verwendet, um Benutzungsschnittstellen für die einzelnen Dienste zu erzeugen, die in sogenannten Portlets als einzelne unabhängige Teilbereiche in die Darstellung des Gesamtportals eingebunden werden. Bei Portlets handelt es sich um Applikationen des Portals, die genutzt werden können, um verschiedene Dienste innerhalb einer Seite anzubieten. Aus Sicht des Benutzers ist ein Portlet ein spezialisierter Bereich, der einen Ausschnitt der Portalseite belegt. Portlets können unabhängig voneinander modifiziert und vom Benutzer auf einer Portalseite nach seinen Vorstellungen angeordnet werden.

Die Schnittstelle eines Portlets im Jetspeed Framework ähnelt der eines Servlets, die in der Java Servlet API [Davidson Coward 1999] beschrieben ist. Das in Abschnitt 4.2.2.3 vorgestellte VesufServlet generiert in der `doGet()` Methode eine Benutzungsschnittstelle auf Basis eines spezifizierten Applikationsmodells und verarbeitet Benutzereingaben in der `doPost()` Methode. Ein Portlet des PublicationPORTALs besitzt als Äquivalent zu den `doGet()`/`doPost()` Methoden eines Servlets die Methoden `getContent()` und `handleRequest()`. Ausgehend vom VesufServlet wurde ein VesufPortlet entwickelt, das in der `getContent()` Methode die Benutzungsschnittstelle anhand des Applikationsmodells eines Dienstes generiert. Es musste berücksichtigt werden, dass ein Servlet seinen Inhalt direkt auf einen Ausgabekanal schreibt, während ein Portlet in Jetspeed ein ECS-Element [ASF 2001a] erzeugen muss. Da Vesuf intern streambasiert arbeitet, erzeugt das VesufPortlet ein spezielles ECS-Element, das den in einen String umgewandelten Inhalt des Ausgabestroms enthält. Die `doPost()` Methode ließ sich ohne weitere Schwierigkeiten in die `handleRequest()` Methode überführen.

Berücksichtigung finden musste weiterhin, dass ein und derselbe Dienst auch mehrfach in einer Portalseite integrierbar sein sollte. Das VesufServlet geht jedoch davon aus, dass eine Applikation nur einmal pro HTTPSession ausgeführt wird. Daher muss das VesufPortlet, das wie auch das VesufServlet laufzeitspezifische Daten in der HTTPSession ablegt, diese Daten nach der eindeutigen ID der Portletinstanz verwalten, damit Portletinstanzen nicht interferieren und fälschlicherweise Daten einer anderen Portletinstanz zugreifen.

Wie das VesufServlet erhält auch das VesufPortlet als Parameter das Applikationsmodell der auszuführenden Anwendung. Somit lassen sich mit Vesuf realisierte Dienste mit den vorhandenen portalspezifischen Mechanismen in das PublicationPORTAL integrieren.

5.2 Methodologie

Um die Dienste mit dem Vesuf System zu konstruieren, sind die Autoren einer Vorgehensweise gefolgt, die sich in einer natürlichen Art und Weise aus den zu erledigenden Teilaufgaben ergeben hat. Diese Methodologie sollte keinesfalls als verbindliche Vorgabe zur Entwicklung von Anwendungen mit Vesuf gesehen werden, sondern vielmehr zur Vorstellung aller notwendigen Arbeitsschritte dienen.

Die Autoren führen zunächst stets eine elementare Taskanalyse durch, deren Ziel darin besteht, die relevanten Aufgaben zu isolieren, die dem Benutzer bei der Verwendung eines Dienstes zur Verfügung stehen sollen. Mit Hilfe dieser Informationen kann bereits ein erstes Taskmodell aufgebaut werden, das evtl. auch die temporalen Übergänge zwischen den verschiedenen Aufgaben enthält. Die Autoren erachten es als wichtig, Aufgaben auf Systemebene zu identifizieren. Aufgaben niedrigerer Ebenen sind wenig sinnvoll für die Modellierung mit dem Vesuf System und führen nur zu einer unnötig gesteigerten Komplexität der Modelle.

Nach der Taskanalyse erarbeiten die Autoren, welche weiteren Objekte für die Modellierung des vollständigen Domänenmodells notwendig sind. Sie führen dazu eine Domänenanalyse durch, deren Ausgangspunkt die bereits in der Taskanalyse festgestellten Aufgaben sind. Dazu ist zu untersuchen, welche Domänenelemente durch das User Interface widergespiegelt werden sollen und welche Verbindungen untereinander und mit den Aufgaben bestehen. Des weiteren sollte an dieser Stelle das integrierte Domänenmodell so weit verfeinert werden, dass auch Informationen zu den Eigenschaften und Operationen der Objekte im Modell reflektiert werden.

Im Anschluss an und zum Teil parallel zur Domänenmodellierung implementieren die Autoren den fachlichen Kern einer Anwendung. Der Implementationsprozess wirft im allgemeinen neue Erkenntnisse und zum Teil auch neue Problemstellungen auf, die zu Änderungen am Domänenmodell führen können. Die Autoren haben festgestellt, das ein iteratives Vorgehen für Domänenmodellierung und Domänenimplementation notwendig und angenehm ist, da schrittweise Änderungen einfacher in die bereits vorliegenden Teile integriert werden können.

Den nächsten Schritt des Entwicklungsprozesses sehen die Autoren im Entwurf der Dialogsteuerung. Als Basis dieses Arbeitsschrittes können die Ergebnisse der Taskanalyse herangezogen werden. Sie geben einen ersten Anhaltspunkt, welche verschiedenen Dialogtypen innerhalb der Anwendung gefordert sind. Je nach Interfacemodalität und Endgerätetyp sind verschiedene Abbildungen von Aufgaben auf Dialoge als sinnvoll zu erachten. Im einfachsten Fall kann eine Eins-zu-eins-Abbildung gewählt werden, indem für jeden Task ein Dialogzustand vorgesehen wird.

Weiterhin muss die Präsentation der im Dialogmodell definierten Sichten festgelegt werden. Die Autoren halten es für sinnvoll, diesen Schritt in zwei gedanklichen Stufen durchzuführen. Zunächst sollte überlegt werden, welche Elemente das abstrakte Präsentationsmodell ausmachen, d. h. welche Interaktionsmöglichkeiten dem Benutzer dargeboten werden. Danach kann entschieden werden, durch welche konkreten Interaktionselemente die bereits festgelegten Interaktionsmöglichkeiten präsentiert werden sollen.

Im letzten Schritt müssen die Applikationsdeskriptoren für jede Anwendung erstellt werden. Dabei muss für eine Applikation mit verschiedenen User Interface Modalitäten je ein Applikationsdeskriptor pro Modalität entworfen werden.

5.3 Online Dictionaries

Im World Wide Web gibt es eine Vielzahl von Online-Wörterbüchern, die zwischen verschiedenen Sprachen übersetzen können. Die online verfügbaren Wör-

terbuchdienste divergieren sehr stark bezüglich der Breite und Tiefe des von ihnen bereitgestellten Angebots. So gibt es Dienste wie TravLang's Translating Dictionaries,[1] die viele verschiedene Sprachen anbieten, aber einen geringen Wortschatz besitzen (große Breite, geringe Tiefe) und Dienste wie das LEO English/German Dictionary,[2] die nur wenige Sprachen anbieten, dafür aber einen umfangreichen Wortschatz besitzen.

Es können meist nicht nur einzelne Wörter übersetzt werden, sondern auch ganze Phrasen. Daher wird neben der Übersetzung auch der Zusammenhang (Kontext), in dem das gesuchte Wort gefunden wurde, angezeigt. Ein Beispiel aus dem Wörterbuch von LEO: Sucht man dort nach *„Beispiel“* findet LEO u. a. die Phrase *„ein Beispiel geben“* mit der Übersetzung *„to set an example.“*

Aufgrund der Vielfalt und unterschiedlichen Ausrichtung der verschiedenen Online Dictionaries ist keines für alle Arten von Abfragen geeignet. Zur Integration in das PublicationPORTAL wurde daher ein Meta-Dictionary Service entworfen, über den Anfragen gezielt an verschiedene Online Dictionaries gestellt werden können. Oberstes Ziel dieses Metadienstes war dabei, für alle im Internet verfügbaren Wörterbuchdienste eine einheitliche Benutzungsschnittstelle präsentieren zu können, so dass der Benutzer keine Schwierigkeiten dabei hat, schnell zwischen verschiedenen Wörterbuchdiensten hin und her zu schalten.

Gemein ist allen Online Dictionaries der Zugriff über das WorldWideWeb: Auf der Webseite des Dienstes kann ein Suchbegriff eingegeben werden, und die Ergebnisse werden als eine weitere Webseite präsentiert. Die Übertragung der Abfragen und Ergebnisse erfolgt über das HTTP Protokoll [Fielding et al. 1999]. Anfragen an Online Dictionaries werden üblicherweise in der URL eines HTTP-GET Requests kodiert. Als Ergebnis des Requests liefert das Online Dictionary die HTML-Seite mit den Abfrageergebnissen.

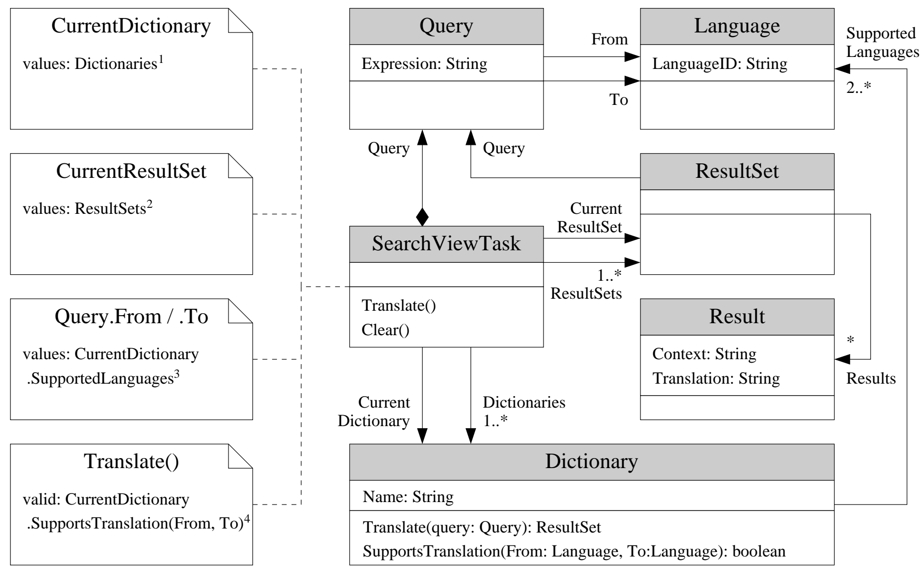
5.3.1 Domänenmodell

Eine elementare Taskanalyse des Metawörterbuchkonzeptes ergibt, dass ein Benutzer zur Bedienung des Dienstes drei Tasks durchführt, die jedoch so einfach gehalten sind, dass sie in einem übergeordneten Task zusammengefasst werden sollten. Die drei Teilaufgaben sind die Auswahl eines Online Dictionaries, die Eingabe der Wörterbuchanfrage und das Betrachten der zurückgelieferten Ergebnisse.

Abbildung 5.2 zeigt das entworfene Domänenmodell des Metawörterbuchdienstes. Zentrales Element des Modells ist die in der Taskanalyse festgestellte übergeordnete Aufgabe – der SearchViewTask. Dieser Task ermöglicht dem Benutzer, eine Anfrage zu stellen. Dazu leitet er eine vom Benutzer zu spezifizierende Anfrage an den ausgewählten Wörterbuchdienst weiter und verwaltet die von den Anfragen produzierten Ergebnisse.

Um eine Anfrage zu spezifizieren, werden drei Informationen benötigt: Ein Ausdruck (Expression) sowie Ausgangs- und Zielsprache (Language). Diese Informationen werden von einem Anfrageobjekt (Query) gekapselt. Zusätzlich wird der spezifische Wörterbuchdienst ausgewählt. Dazu hat der SearchViewTask eine Referenz auf den aktuell ausgewählten Dienst (CurrentDictionary), der aus einer Liste mit bekannten Diensten (Dictionaries) ausgewählt werden kann.

[1] `http://dictionaries.travlang.com`
[2] `http://dict.leo.org`



1) VEPL: <owner>.<owner>.Dictionaries.<value>
2) VEPL: <owner>.<owner>.ResultSets.<value>
3) VEPL: <owner>.<owner>.<composite>SearchViewTask.CurrentDictionary.SupportedLanguages.<value>
4) VEPL: <owner>.<owner>.CurrentDictionary.SupportsTranslation(From = <owner>.<owner>.Query.From, To = <owner>.<owner>.Query.To)

Abbildung 5.2: Domänenmodell des Wörterbuch Dienstes

Ein Dictionary-Objekt kapselt einen Wörterbuchdienst. Zur Unterscheidung verschiedener Online-Dienste, hat jedes Wörterbuchobjekt einen Namen. Es stellt eine Operation zum Übersetzen einer Anfrage bereit (Translate), die ein Anfrageobjekt als Eingabe erwartet und eine Ergebnismenge (ResultSet) zurückliefert. Um Informationen über die möglichen Anfragen an einen Dienst zu erhalten, kann man von einem Dictionary-Objekt die unterstützten Sprachen (SupportedLanguages) abfragen. Über eine weitere Operation läßt sich genau feststellen, ob ein Dienst die Übersetzung von einer spezifischen Ausgangs- in eine spezifische Zielsprache unterstützt (SupportsTranslation).

Einzelne Abfrageergebnisse werden in Result-Objekten gespeichert. Diese besitzen dazu Attribute für die Übersetzung (Translation) und für den Kontext, in dem die Anfrage gefunden wurde. Der Kontext wird von vielen Wörterbuchdiensten deshalb angeboten, weil die Dienste nicht nur einzelne Wörter übersetzen, sondern auch ganze Phrasen. Der Kontext stellt dann die Phrase dar, in der die Suchanfrage gefunden wurde, während die Übersetzung nicht nur die Anfrage, sondern die ganze Phrase übersetzt. Da meist mehr als ein Anfrageergebnis pro Anfrage zurückgeliefert wird, werden Anfrageergebnisse in Ergebnismengen zusammengefasst. Der Vollständigkeit halber erhält eine Ergebnismenge zusätzlich einen Verweis auf die Anfrage, durch die sie produziert wurde. Der SearchViewTask verwaltet die bisherigen Ergebnismengen (ResultSets), die durch den Benutzer zur Darstellung ausgewählt werden können. Die aktuell dargestellte Ergebnismenge ist dem SearchViewTask durch eine Referenz bekannt (CurrentResultSet). Um die bisherigen Abfragen zu löschen, stellt der SearchViewTask eine weitere Operation zur Verfügung (Clear).

5.3.1.1 Constraints

Zur besseren Benutzbarkeit sind für den SearchViewTask einige Constraints definiert, die die erlaubten Benutzereingaben der jeweiligen Einstellungen auf sinnvolle Werte beschränken. Zum einen sind dies die Constraints auf den Referenzen CurrentDictionary und CurrentResultSet, welche die erlaubten Werte (values) auf die dem SearchViewTask bekannten Dictionaries bzw. ResultSets beschränken.

Zum anderen wird auch die Auswahl der Ausgangs- und Zielsprache eines Query-Objektes jeweils durch Values-Constraints eingeschränkt. Gültige Eingaben sind hier nur die vom aktuellen Dictionary-Objekt des SearchViewTasks unterstützten Sprachen. Diese Constraints sind damit abhängig vom SearchViewTask, dem das Query-Objekt durch eine Kompositionsbeziehung zugeordnet ist.

Das letzte Constraint ist auf der Translate-Operation des SearchViewTasks definiert. Es legt fest, das die Operation nur dann aufgerufen werden sollte, wenn das im Query-Objekt ausgewählte Sprachenpaar vom aktuell eingestellten Dictionary-Objekt unterstützt wird. Dazu wird die SupportsTranslation-Operation als Pfad einer Valid-Property angegeben, wobei die im Query festgelegten Einstellungen für Ausgangs- und Zielsprache als Argument übergeben werden.

5.3.1.2 Dependencies

Das Constraint für die Translate-Operation des SearchViewTasks verwendet als Argumente im Pfad der Valid-Property die From- und To-Referenzen des Queries. Damit hängt das Constraint von diesen Referenzen ab. Die aktuelle Vesuf Implementation kann diese Art der Abhängigkeit (Pfade, die als Parameter eines Operationsaufrufes angegeben werden) nicht automatisch erkennen (siehe Abschnitt 4.2.3.3). Diese Abhängigkeiten müssen daher explizit spezifiziert werden. Der Dependency-Pfad ist identisch mit dem jeweiligen Argumentpfad. Zu beachten ist aber, das dem Dependency-Pfad nicht automatisch ein `<value>` Element angefügt wird. Für das From-Argument ist der korrekte Dependency-Pfad also (vgl. Abb. 5.2): `<owner>.<owner>.Query.From.<value>`.

5.3.2 Implementation

Die Klassen Query, Language, Result und ResultSet implementieren die Datenstrukturen, mit denen der Wörterbuchdienst operiert. Sie haben kein eigenes Verhalten und sind daher einfach umzusetzen. Lediglich das Query-Objekt ist modifizierbar. Die anderen drei Objekte können, nachdem sie initialisiert wurden, nicht mehr geändert werden. Deshalb erzeugt sich ein ResultSet, dass mit einem Query initialisiert wird, intern eine Kopie des Queries, die dann nicht mehr verändert werden kann.

Da das Query-Objekt modifizierbar ist, ist es notwendig, dass es auftretende Ereignisse publiziert. Dies betrifft Änderungen an der Expression und der gewählten Ausgangs- bzw. Zielsprache. Dazu verwendet es den in Abschnitt 4.2.2.4 beschriebenen Eventmechanismus von Vesuf. Für den schreibenden Zugriff auf die Attribute und Referenzen des Query-Objektes werden set-Methoden implementiert, die mit Hilfe eines EventDispatcherProxies einen ValueChanged Event auf dem entsprechenden Element generieren.

Der SearchViewTask implemetiert die vom Benutzer durchführbaren Aktionen in den Methoden `clear()` und `translate()`. Bei `clear()` wird die Liste der bisherigen Ergebnismengen gelöscht. Dieses Ereignis wird mittels der `elementRemoved()` Methode des EventDispatcherProxies publiziert, wodurch etwaige Darstellungen der Ergebnismengen automatisch aktualisiert werden. Die `translate()` Methode ruft das aktuell eingestellte Wörterbuch mit dem spezifizierten Query auf, fügt die Ergebnismenge der Ergebnismengenliste hinzu und setzt die aktuelle Ergebnismenge neu. Diese Ereignisse werden wieder publiziert (`elementAdded()` und `valueChanged()`). Die Einstellung des aktuellen Wörterbuches ist durch get/set Methoden implementiert, wobei auch hier in der set-Methode eine Werteänderung (value changed) publiziert wird. Einstellungen der Anfrage werden direkt auf dem Queryobjekt vorgenommen. Der SearchViewTask legt bei seiner Erzeugung einmalig ein Queryobjekt an, das er daraufhin für alle Anfragen wiederverwendet.

Zur Initialisierung der verfügbaren Wörterbücher lädt der SearchViewTask bei seiner Erzeugung ein Propertyfile, in dem durch Einträge der Form `Wörterbuchname=Wörterbuchklasse` alle Wörterbuchdienste spezifiziert sind. Um größtmögliche Flexibilität zu gewährleisten, sind Wörterbuchdienste durch ein Interface (IDictionary) repräsentiert. Die in Abbildung 5.2 dargestellten Attribute und Referenzen eines Dictionaries (Name und SupportedLanguages) werden durch get-Methoden schon im Interface deklariert. Der SearchViewTask instantiiert alle in den dictionaries.properties angegebenen Wörterbuchobjekte, auf die er anschließend über das IDictionary Interface zugreifen kann.

5.3.2.1 Struktur der einzelnen Dictionary-Dienste

Ziel des Meta-Dictionaries ist es, das Einbinden der unterschiedlichen Online-Wörterbücher möglichst einfach zu machen. Der generelle Abfragemechanismus, der in der `translate()` Methode eines Dictionaryobjektes implementiert ist, ist für diverse Onlinedienste identisch und besteht aus fünf Schritten: Erstens, bevor eine Abfrage durchgeführt wird, kann bereits überprüft werden, ob die Anfragespezifikation überhaupt sinnvoll ist. Dazu kann überprüft werden, ob die gewählten Sprachen vom Wörterbuch unterstützt werden, und ob die Übersetzungsrichtung zulässig ist.

Handelt es sich um eine gültige Anfrage, so muss zweitens die URL konstruiert werden, mit der der Onlinedienst abgefragt wird. Alle von uns betrachteten Dienste unterstützen einen HTTP-GET Request, in dem sämtliche Anfrageparameter in der URL kodiert sind und nicht wie beim POST Request zusätzlich zur URL gesendet werden müssen. In die URL fließen somit alle Eigenschaften der Anfrage wie Ausdruck, Ausgangs- und Zielsprache ein sowie Parameter die für den Onlinedienst spezifisch sind. Die Art und Weise, wie die Anfragen in der URL kodiert sein müssen, divergiert bei den einzelnen Diensten sehr stark.

Steht die URL fest, wird drittens das vom Onlinedienst produzierte Ergebnis heruntergeladen. Da Abfrageergebnisse von Onlinewörterbüchern meist sehr einfach gehalten sind, verzichten die Autoren auf eine streambasierte Lösung, in der bereits Teilergebnisse weiterverarbeitet werden können. Stattdessen wird ein Abfrageergebnis erst verarbeitet, wenn es komplett gesendet wurde.

Der vierte Schritt besteht dann darin, die einzelnen Ergebniseinträge aus dem als HTML vorliegenden Ergebnis zu extrahieren. Hierin liegt der aufwändigste Teil der Wörterbuchimplementation, da die Onlinedienste ihre HTML-Seiten

nicht zur einfachen Weiterverarbeitung optimieren, sondern im Gegenteil nur darauf achten, das diese Seiten in Web-Browsern angemessen dargestellt werden. Auch unterscheiden sich die Darstellungstechniken stark. So ordnet z. B. der LEO-Dienst die einzelnen Treffer und Übersetzungen eins-zu-eins in einer Tabelle (`<table>`) an, während das Travlang-Dictionary präformatierten Text (`<pre>`) benutzt, und zu einem Treffer auch mehrere Übersetzungen nacheinander anbietet. Für die extrahierten Ergebniseinträge werden Resultobjekte erzeugt, die einem neuen ResultSet hinzugefügt werden. Als fünfter und letzter Schritt wird die erzeugte Ergebnismenge zurückgeliefert (z. B. an den SearchViewTask).

5.3.2.2 Parser

Drei dieser Schritte können dabei schon unabhängig vom Onlinedienst umgesetzt werden. Dazu haben die Autoren eine abstrakte Klasse Parser, die das Dictionary Interface implementiert. Für jeden spezifischen Onlinedienst muss eine Subklasse dieser Klasse entworfen werden, die die nicht allgemein implementierbaren Details realisiert. Die abstrakte Parserklasse verwaltet bereits die unterstützten Sprachen, und die zulässigen Übersetzungsrichtungen. Dazu müssen von einer speziellen Subklasse einmalig die unterstützten Sprachpaare, bestehend aus Ausgangs- und Zielsprache, initialisiert werden. In der `translate()` Methode der abstrakten Parserklasse kann somit bereits die Gültigkeit der Anfrage in bezug auf den spezifischen Wörterbuchdienst überprüft werden. Ebenfalls allgemein implementierbar ist das Herunterladen des Ergebnisses eines Onlinedienstes. Eine Besonderheit unserer Implementation ist, dass das Metawörterbuch wahlweise als reine HTML-Lösung oder als Java-Applet genutzt werden kann. Da ein Applet nur auf den Server, von dem es geladen wurde, zugreifen darf, kann das Metadictionary-Applet nicht direkt auf die diversen Onlinedienste zugreifen. Für den Fall, das das Metadictionary als Applet ausgeführt wird, sendet der Parser alle Anfragen an ein Proxy-Servlet, das die Anfragen an die jeweiligen Dienste weiterleitet. Die erzeugte Ergebnismenge wird am Ende der `translate()` Methode zurückgeliefert.

Die übrigen zwei Schritte, also die Konstruktion der URL und das Parsen des Ergebnisses, müssen jedoch speziell für jeden Dienst implementiert werden. Die abstrakte Parserklasse deklariert zu diesem Zweck zwei abstrakte Methoden, die von der `translate()` Methode aufgerufen werden. Die Methode `createQueryURL()` erzeugt für ein gegebenes Anfrageobjekt eine URL. Dabei gibt es gravierende Unterschiede zwischen den einzelnen Diensten. So enthält z. B. eine URL des TravLang Dienstes Verzeichniseinträge, die die Ausgangs- und Zielsprache angeben, während bei leo.org, wo nur die Sprachen Deutsch und Englisch unterstützt werden, lediglich ein Parameter für die Abfragerichtung (-1 für Englisch-Deutsch, 1 für Deutsch-Englisch) vorhanden ist.

Der aufwändigste Teil eines spezifischen Parsers ist jedoch die `parse()` Methode, in der eine geladene HTML-Seite verarbeitet wird, um die Ergebnisse zu extrahieren und das Ergebnismengenobjekt zu erzeugen. Um diese Aufgabe so einfach wie möglich zu machen, stellen die Autoren eine Hilfsklasse namens StringTaggenizer bereit, die ähnlich wie die Standard Javaklasse StringTokenizer eine Zeichenkette verarbeitet und in einzelne Teile zerlegt. Der StringTaggenizer wurde entwickelt um HTML-Markup in möglichst einfacher Weise zu verarbeiten. Er zergliedert eine HTML-Seite in Tags und Text. Über die Methoden `nextTag()` und `hasMoreTags()` kann man durch den HTML-Code navigieren,

```
01:  <TD VALIGN=TOP COLSPAN=4 WIDTH="80%">
02:    <STRONG>13 search results for 'dictionary'</STRONG>
03:    <TABLE COLUMNS=3 CELLPADDING=1 CELLSPACING=0 BORDER=0 WIDTH="100%">
04:      <TR>
05:        <TH>English:</TH>
06:        <TH>  </TH>
07:        <TH>German:</TH>
08:      </TR>
09:      <TR VALIGN="top" BGCOLOR="#FFFCDF">
10:        <TD><B>dictionary</B></TD>
11:        <TD>  </TD>
12:        <TD>das Wörterbuch</TD>
13:      </TR>
14:      <TR VALIGN="top">
15:        <TD><B>dictionary</B></TD>
16:        <TD>  </TD>
17:        <TD>das Verzeichnis</TD>
18:      </TR>
19:      ...
20:    </TABLE>
21:  </TD>
```

Abbildung 5.3: Von dict.leo.org generiertes HTML-Ergebnis

die Methode `currentText()` liefert den Klartext an der aktuellen Position bis zum Start des nächsten Tags. Im folgenden Abschnitt beschreiben die Autoren am Beispiel des LeoParsers, wie die `parse()` Methode eines Dictionary-Objektes implementiert werden kann.

5.3.2.3 HTML-Parsing

Abbildung 5.3 zeigt einen Ausschnitt aus einem vom LEO Online-Wörterbuch generierten HTML-Ergebnis (gesucht wurde nach „dictionary“). Dem gegenüber stellt Abbildung 5.4 die `parse()` Methode des LeoParsers dar. Zu Beginn der `parse()` Methode wird für die Anfrage ein Ergebnismengenobjekt erstellt (Zeile 3 in Abb. 5.4), das im Verlauf der Methode mit den extrahierten Ergebniseinträgen gefüllt wird. Danach wird in Zeile 4 für das HTML-Ergebnis (`input`) ein StringTaggenizer erzeugt. In Abbildung 5.3 ist zu sehen, dass bevor die einzelnen Ergebnisse in einer Tabelle dargestellt werden, eine Zeile mit der Anzahl der gefundenen Ergebnisse ausgegeben wird, die von einem `<strong>` Tag eingeschlossen ist (Zeile 2 in Abb 5.3). Den Umstand, dass dies das einzige `<strong>` Tag in den von Leo generierten HTML-Codes ist, macht sich der Parser zunutze. Die Schleife in Zeile 7 in Abbildung 5.4 überspringt alle Tags im Dokument, bis sie auf ein `<strong>` Tag stößt. Die einzelnen Ergebnisse sind im HTML in einer Tabelle dargestellt (Zeilen 3-20, Abb. 5.3). Die erste Zeile der Tabelle (Zeilen 4-8) enthält dabei eine Überschrift, und zählt somit nicht zu den Ergebnissen. Deshalb wird in der `parse()` Methode bis zum Ende der ersten Tabellenzeile gesprungen, das durch eine `</tr>` Tag gekennzeichnet ist (Zeile 8 in Abb. 5.3, Zeile 10 in Abb. 5.4). In der `parse()` Methode folgt eine Schleife (Zeilen 13-29), die die folgenden Zeilen der HTML-Tabelle bearbeitet und die enthaltenen Einträge extrahiert. Wie in Abbildung 5.3 zu sehen, besteht jede Zeile der Tabelle aus drei Tabellenzellen (markiert durch `<td>` Tags), wobei die mittlere Zelle lediglich ein Lückenfüller ist, der optischen Zwecken dient. Die Zeilen 15-18 der `parse()` Methode sammeln sämtlichen Text bis zum Ende der ersten Tabellenzelle in der

```
01:  protected ResultSet parse(String input, Query query)
02:  {
03:    ResultSet rs  = new ResultSet(query);
04:    StringTaggenizer  stag  = new StringTaggenizer(input);
05:
06:    // Move to start of result table.
07:    for(; stag.hasMoreTags() && !stag.nextTag().equals("<STRONG>"); );
08:
09:    // Skip first row (title).
10:    for(; stag.hasMoreTags() && !stag.nextTag().equals("</TR>"); );
11:
12:    // Process result rows until no more rows (end of table).
13:    while(stag.hasMoreTags() && stag.nextTag().startsWith("<TR"))
14:    {
15:      // Parse left side until end of table cell.
16:      String  left  = "";
17:      for(; !"</TD>".equals(stag.nextTag()); )
18:        left  += stag.currentText();
19:
20:      // Skip one table cell.
21:      for(; stag.hasMoreTags() && !stag.nextTag().equals("</TD>"); );
22:
23:      // Parse rigth side, until end of table row.
24:      String  right  = "";
25:      for(; !"</TR>".equals(stag.nextTag()); )
26:        right  += stag.currentText();
27:
28:      rs.addResult(new Result(left.trim(), right.trim()));
29:    }
30:
31:    return rs;
32:  }
```

Abbildung 5.4: `parse()` Methode des LeoParsers

Variable `left`, wobei alle anderen vorkommenden Tags, wie zum Beispiel die Boldface Einstellungen (`<b>`, s. Zeilen 10, 15, Abb. 5.3) ignoriert werden. Zeile 21 überspringt die Lückenfüllerzelle, und Zeilen 23-26 legen den textuellen Inhalt bis zum Ende der Tabellenzeile in der Variable `right` ab. Zeile 28 erzeugt einen neuen Ergebniseintrag aus den Variablen `left` und `right`, der zugleich der in Zeile 3 erzeugten Ergebnismenge hinzugefügt wird. Die Schleife wird solange durchlaufen, bis alle Tabellenzeilen verarbeitet sind. Zuletzt wird in Zeile 31 das vollständige Ergebnismengenobjekt zurückgeliefert.

5.3.3 Präsentationsmodelle

Um die Flexibilität des Vesuf Systems unter Beweis zu stellen, wurde der Metawörterbuchdienst in einer Reihe von unterschiedlichen Interfacemodalitäten verfügbar gemacht. Neben einer AWT-Version, die auch als Applet ausgeführt werden kann, wurde der Dienst servletbasiert als HTML, WML und VXML Anwendung realisiert. Die Details der einzelnen Präsentationsspezifikationen werden in den folgenden Abschnitten besprochen.

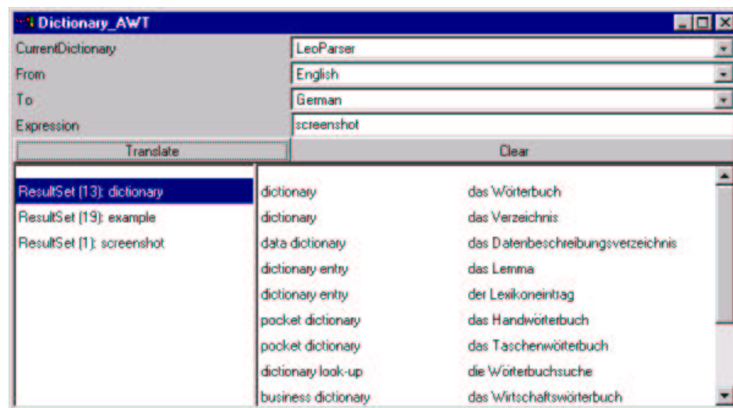


Abbildung 5.5: Screenshot des Meta-Dictionaries (Java-AWT)

5.3.3.1 AWT-Version

Abbildung 5.5 zeigt die Benutzungsschnittstelle des Meta-Dictionaries in der Java-AWT Version. Der obere Bereich enthält alle notwendigen Interaktionselemente zur Eingabe der Suchanfrage. Ein Textfeld wird zur Eingabe des gesuchten Ausdrucks (Expression) verwendet. Aktuelles Wörterbuch sowie Ausgangs- und Zielsprache können durch Auswahlelemente (ChoiceComponents) dargestellt werden, da über die auf dem Domänenmodell definierte Values-Constraints die auswählbaren Werte verfügbar sind. Die beiden Operationen (Translate und Clear) des Dienstes kann der Benutzer durch Buttons auslösen. Die Liste der bisherigen Anfragen und die aktuellen Ergebnisse werden darunter durch zwei Listenelemente dargestellt.

Die Spezifikation der AWT-Präsentation ähnelt dem in Abschnitt 4.2.5 detailliert besprochenen Beispielmodell. In einer UIML-Datei werden sowohl die abstrakten Präsentationselemente wie Dialoge, Labels und Delegates spezifiziert als auch ihre AWT-spezifischen konkreten Eigenschaften wie z. B. Layout. Die AWT-Präsentationselemente machen sich dabei den Eventmechanismus von Vesuf zunutze. Wird ein neues aktuelles Dictionary ausgewählt, werden durch Dependencies Änderungen der zulässigen Werte für die Ausgangs- und Zielsprache propagiert (siehe Abschnitt 5.3.1.2). Dadurch werden die Auswahlelemente automatisch mit den unterstützten Sprachen des jeweiligen Wörtebuchdienstes gefüllt.

Das User Interface stellt die bisherigen Anfragen und die aktuelle Ergebnismenge in zwei Listenelementen dar (links bzw. rechts unten). Auch hier ermöglicht der Eventmechanismus, dass bei einer Auswahl der Ergebnismenge (durch Klicken auf einen Eintrag der linken Liste) die gewählte Ergebnismenge (rechte Liste) sofort angezeigt wird. Das Listenelement stellt also nicht, wie für Delegates üblich, ein einziges Domänenelement dar, sondern gleich zwei: Einerseits wird die Liste der letzten Ergebnisse dargestellt (ResultSets), andererseits ist über die ausgewählte Listenzeile auch das anzuzeigende Ergebnis (CurrentResultSet) repräsentiert.

Zur Darstellung der Liste der aktuellen Ergebniseinträge wird ein ListDelegate verwendet, dessen UIML-Spezifikation in Abbildung 5.6 aufgeführt ist. Der Pfad des dargestellten Domänenelementes ist `CurrentResultSet.Results` (s.

```
01: <Delegate name              = "CurrentResultSet"
02:           path              = "CurrentResultSet.Results"
03:           widgettype        = "List"
04:           layoutconstraints = "GridBagConstraints(...)"
05:           background        = "white">
06:
07:   <Delegate name        = "Context"
08:             path        = "&lt;index&gt;.Context"
09:             widgettype  = "Label"/>
10:   <Delegate name        = "Translation"
11:             path        = "&lt;index&gt;.Translation"
12:             widgettype  = "Label"/>
13:
14: </Delegate>
```

Abbildung 5.6: Spezifizierung eines Listenelementes

Zeile 2). Damit visualisiert die Liste ein multiplizitäres Attribut (vgl. Abb. 5.2). Bei der Instantiierung des Listenelements repliziert das Vesuf System für jeden Eintrag des multiplizitären Attributes alle Unterelemente des Listenelements. In diesem Fall wird also für jedes Result des ResultSets jeweils ein Delegate für den Kontext (Zeilen 7-9) und ein Delegate für die Übersetzung (Zeilen 10-12) erzeugt und dargestellt. Das `<index>` Element[3] des Pfades (Zeilen 8, 11) dient dabei als Platzhalter für den Index des jeweiligen Eintrags, und wird während der Erzeugung durch den entsprechenden Wert ersetzt. Werden dem zugrundeliegenden Attribut zur Laufzeit neue Werte hinzugefügt oder bestehende Werte gelöscht, so werden die einzelnen Einträge des Listenelements automatisch entsprechend angepasst.

5.3.3.2 HTML-Version

In Abbildung 5.7 ist ein Browserfenster abgebildet, welches das Portlet des Dienstes im PublicationPORTAL darstellt. Der obere Bereich mit den Eingabemasken CurrentDictionary und Query entspricht von der Funktionalität der AWT-Schnittstelle. Da jedoch in der HTML-Version der Eventmechanismus nicht genutzt werden kann, gibt es einen zusätzlichen „Select"-Knopf zur Auswahl eines Wörterbuchdienstes, der dazu führt, dass die Seite neu geladen wird, woraufhin auch die Auswahlelemente für die Sprachen neu initialisiert werden. Die HTML-Version stellt die bisherigen Anfragen in einem Choice dar, und bietet, wie schon zur Auswahl des Wörterbuchdienstes, einen Button an („Show"), um die Darstellung an eine neue Selektion anzupassen. Die Darstellung der Ergebnismenge als HTML-Tabelle wird intern in einem Template generiert.

Die UIML-Spezifikation des Präsentationsmodells ist in Abbildung 5.8 wiedergegeben. Das allen anderen Präsentationselementen übergeordnete Element ist der DictionaryDialog (Zeilen 10-22). Dargestellt wird er durch ein Widget vom Typ Template (Zeile 12). Ein Template wird durch eine Velocity-Datei gerendert, deren Pfad über eine Property angegeben ist (Zeile 13). Alle innerhalb des DictionaryDialogs definierten Präsentationselemente können aus dem Templates heraus zugegriffen und dargestellt werden. Bemerkenswert an der Spezifikation des Präsentationsmodells ist weiterhin die Angabe des Contenttype

[3]Da XML das Verwenden von spitzen Klammern nicht zulässt, muss dieses Element in der UIML-Datei durch `<index>` kodiert werden.

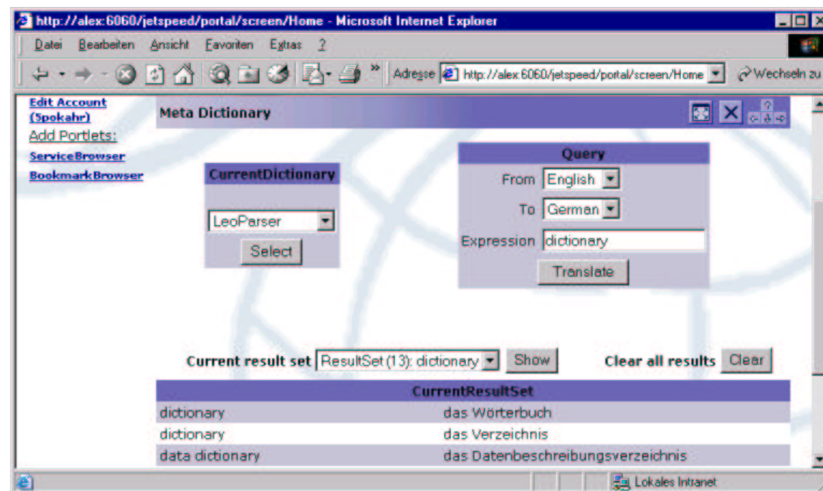


Abbildung 5.7: Screenshot des Meta-Dictionaries (HTML-Portlet)

(Zeile 5), die vom servletbasierten Interpreter (VesufServlet) benötigt wird.

Abbildung 5.9 zeigt einen Auschnitt aus dem Velocity-Template, das den DictionaryDialog rendert. Das Template enthält hauptsächlich HTML-Code. Es ist eine Eigenschaft von Velocity, dass man sich Platzhalter definieren kann, die später im Template referenziert werden können. Verschiedene Platzhalter werden vom Vesuf System definiert und dem Template zugänglich gemacht. So ist z. B. die Basisadresse des VesufServlets bzw. VesufPortlets über den Namen `$baseurl` verfügbar und kann als Ziel für Links und Forms verwendet werden (Zeile 2). Über die Referenz `$this` kann auf das zu rendernde Präsentationselement, in diesem Fall also auf den DictionaryDialog zugegriffen werden. In den Zeilen 6 und 12 in Abb. 5.9 ist zu sehen, wie man über diese Referenz die im Präsentationsmodell nachgeordneten Elemente in das Dokument einfügen kann.

Der untere Teil des Templates ist für die Generierung der Liste der Ergebniseinträge als Teil einer HTML Tabelle zuständig. In Zeile 22 wird das Präsentationselement für die Liste in der Referenz `$list` abgelegt. Das `#foreach` Konstrukt der Velocity Templatesprache (Zeilen 23-33) wird dazu benutzt, für jeden Eintrag in der aktuellen Ergebnismenge eine Tabellenzeile (`<TR>`, Zeilen 24-32) zu generieren. Diese Schleife geht alle Schlüsselelemente des Listenelements durch (Zeile 23). Bei den Schlüsselelementen eines multiplizitären Attributs handelt es sich um die einzelnen Indizes. Die Zeilen 26, 27 und 29, 30 erzeugen jeweils Delegates für Context und Translation des Ergebnismengeneintrages. Die Delegates werden im in UIML spezifizierten Präsentationsmodell (vgl. Abb. 5.6) mit Pfaden versehen, die ein `<index>` Element enthalten. Zur Instantiierung der Delegates wird das Schlüsselelement (`$key`) als Wert für dieses Pfadelement verwendet (Zeilen 26, 29). So kann auf die einzelenen Präsentationselemente für den Listeninhalt zugegriffen werden, die dann in den Zeilen 27 und 30 in das generierte Dokument eingefügt werden.

```
01: <head>
02:   <meta name="Name" content="Dictionary_HTML"/>
03:   <meta name="ObjectModel" content="dictionary.DictionaryObjectModel"/>
04:   <meta name="Factory" content="org.vesuf.runtime.presentation.html.Factory"/>
05:   <meta name="ContentType" content="text/html"/>
06: </head>
07:
08: <interface name="Dictionary">
09:   <structure>
10:     <Dialog name       = "DictionaryDialog"
11:             path       = "SearchViewTask"
12:             widgettype = "Template"
13:             template   = "dictionary/dictionary_html.vm">
14:
15:       <Label    name       = "CurrentDictionary_label"
16:                 path       = "CurrentDictionary"
17:                 widgettype = "Text"/>
18:       <Delegate name       = "CurrentDictionary_delegate"
19:                 path       = "CurrentDictionary"
20:                 widgettype = "Choice"/>
21:       ...
22:     </Dialog>
23:   </structure>
24: </interface>
```

Abbildung 5.8: Spezifikation des HTML Präsentationsmodells

```
01: ...
02: <FORM action="$baseurl" method="POST" name="Query">
03:   <TABLE bgcolor="$tablecol" cellspacing="0" cellpadding="3">
04:     <TR>
05:       <TH valign="top" bgcolor="$titlecol">
06:         $this.printSubpartInstance("CurrentDictionary_label")
07:       </TH>
08:     </TR><TR>
09:       <TD> </TD>
10:     </TR><TR>
11:       <TD valign="middle">
12:         $this.printSubpartInstance("CurrentDictionary_delegate")
13:       </TD>
14:     </TR><TR>
15:       <TD valign="bottom" align="center">
16:         <INPUT type="submit" name="Select" value="Select">
17:       </TD>
18:     </TR>
19:   </TABLE>
20: </FORM>
21: ...
22: #set($list  = $this.getSubpartInstance("CurrentResultSet_delegate"))
23: #foreach($key in $list.Keys)
24:   <TR>
25:     <TD bgcolor="$bgcol">
26:       #set($context  = $list.getSubpartInstance("Context", $key))
27:       $this.printSubpartInstance($context)
28:     </TD><TD bgcolor="$bgcol">
29:       #set($context  = $list.getSubpartInstance("Translation", $key))
30:       $this.printSubpartInstance($translation)
31:     </TD>
32:   </TR>
33: #end
34: ...
```

Abbildung 5.9: HTML Template (Ausschnitt)

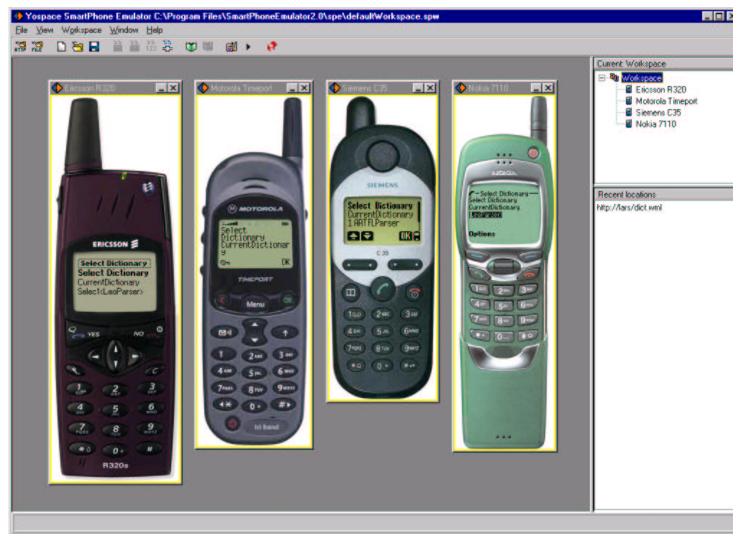


Abbildung 5.10: Screenshot des Meta-Dictionaries (WML)

5.3.3.3 WML-Version

Die Abbildung 5.10 zeigt die Meta-Dictionary Applikation, während sie von einem Handy-Simulator ausgeführt wird. Leider haben diverse von den Autoren getestete Simulatoren fehlerhafte Cacheimplementationen,[4] die eine fehlerfreie Programmausführung nicht in jedem Fall gewährleisten. Die Autoren haben deshalb zumeist auf die frei verfügbare Entwicklungsumgebung von Nokia zurückgegriffen, bei der keine Cacheproblematiken auffällig wurden.[5]

Um die Benutzungsschnittstelle der WML-Version zu entwerfen, verwenden die Autoren ein zur HTML-Version nahezu identisches Präsentationsmodell. Die einzigen Modifikationen bestehen in der Verwendung eines speziellen WML-Templates, das in Abbildung 5.11 auszugsweise dargestellt ist, und in der Änderung des Contenttypes auf `text/vnd.wap.wml`. Da die zur Verfügung stehende Bildfläche WML-fähiger Endgeräte sehr viel geringer ausfällt als bei herkömmlichen Desktopgeräten, bietet WML Elemente zur Gruppierung einzelner funktionaler Präsentationsblöcke mittels card-Tags an. Die Autoren spezifizieren daher Blöcke für die Auswahl des Wörterbuchs (Zeilen 1, 14), für die Eingabe einer Anfrage und für die Präsentation der Ergebnisse. Um zwischen den Karten zu navigieren, können sie mit Titel und Identifikation versehen werden. Die Karte zur Auswahl eines Wörterbuchs besteht aus einem Erklärungstext (Zeilen 3, 4), dem Delegate für die Wörterbuchselektion (Zeilen 5, 6) und der Aktion, die zu einer serverseitigen Auswertung der Eingaben führt (Zeilen 8-13). Die Autoren verwenden zu diesem Zweck eine Post-Methode mit der bereits erwähnten `$baseurl` Referenz, die um den Zusatz `#query` ergänzt wurde. Durch diesen Zusatz wird nach der Serverauswertung die Karte für die Eingabe einer

[4]Diverse Simulatoren gehen davon aus, dass sie eine Seite cachen können, sofern die URL bereits aufgerufen wurde. Gerade aber für serverseitig generierte Seiten, die einen Programmstatus miteinbeziehen, führt dies zu einer fehlerhaften Darstellung.

[5]Man erhält das Nokia Development Kit nach einer Registrierung unter: `http://www.forum.nokia.com/main/1,6668,1_1,00.html`

```
01:  <card id="dictionary" title="Select Dictionary">
02:    <p>
03:      <strong>Select Dictionary</strong><br/>
04:      $this.print("CurrentDictionary_label")
05:      #set($cdpid = $this.getSubpartInstance("CurrentDictionary_delegate"))
06:      $this.print($cdpid)
07:    </p>
08:    <do type="accept" label="OK">
09:      <go method="post" href="$baseurl#query">
10:        <postfield name ="$cdpid.Part.getName()"
11:                   value="\$($this.getWMLName($cdpid))"/>
12:      </go>
13:    </do>
14:  </card>
```

Abbildung 5.11: WML-Template (Ausschnitt)

Anfrage aktiviert. Zusätzlich zur URL wird dem Aufruf mittels eines Postfields der Parameter des Wörterbuchs beigefügt (Zeilen 10, 11). Leider muss aufgrund der WML-Spezifikation, die keine Punkte innerhalb von Variablennamen erlaubt, eine Konversionsmethode für den Parameternamen eingeführt werden (`getWMLName()`, Zeile 11).

Für die Queryspezifikation und die Präsentation der Ergebnisse wurden Karten in ähnlicher Form definiert. Um dem Benutzer die Bedienung zu erleichtern, wurden Querverweise zwischen den einzelnen Karten eingefügt. So ist es möglich, von der Karte zur Präsentation der Ergebnisse zurückzunavigieren, um eine neue Anfrage an das bereits ausgewählte Wörterbuch zu stellen. Sowohl von der Ergebniskarte als auch von der Karte zur Queryspezifikation kann überdies über eine Option zurück zur ersten Karte navigiert werden, um ein neues Wörterbuch auszuwählen. Die im Metawörterbuchdienst implementierte Fähigkeit, Anfrageergebnisse während einer Sitzung zu speichern und erneut darzustellen, schien uns aufgrund der damit verbundenen Komplexität der Benutzungsschnittstelle für einen WML-Dienst ungeeignet, und wurde daher nicht genutzt.

5.3.3.4 VXML-Version

In Abbildung 5.12 ist die Benutzungsschnittstelle der Voice XML Version des Metawörterbuchdienstes dargestellt [VXML Forum 2000]. Der abgebildete Browser für VXML Anwendungen ist als Paket von Motorola frei erhältlich.[6] Es enthält neben der Browseroberfläche auch die notwendigen Komponenten für die Spracherkennung von Microsoft und einen Agenten zur Sprachausgabe von Lernout&Hauspie.

Das Präsentationsmodell der VXMLVersion unterscheidet sich nicht wesentlich von der HTML-Version. Der Contentype wird auf `text/vxml` festgelegt und mit Ausnahme des Template-Parts werden dieselben Präsentationselemente (parts) definiert. In Abbildung 5.13 ist der Ausschnitt des VXML-Templates dargestellt, der für die Auswahl des Wörterbuchs (Zeilen 1-6) und zur Anfragespezifikation (Zeilen 8-24) benötigt wird. Um diese Eingabeeinheiten funktional zu trennen und einfach referenzieren zu können, werden sie in form-Tags eingefasst (Zeilen 1, 6, 8, 24) und mit einer Identifikation versehen. Um die Auswahl

[6] Das Motorola Mobile ADK for Voice ist nach einer Registrierung verfügbar unter: `http://developers.motorola.com/developers/wireless/`

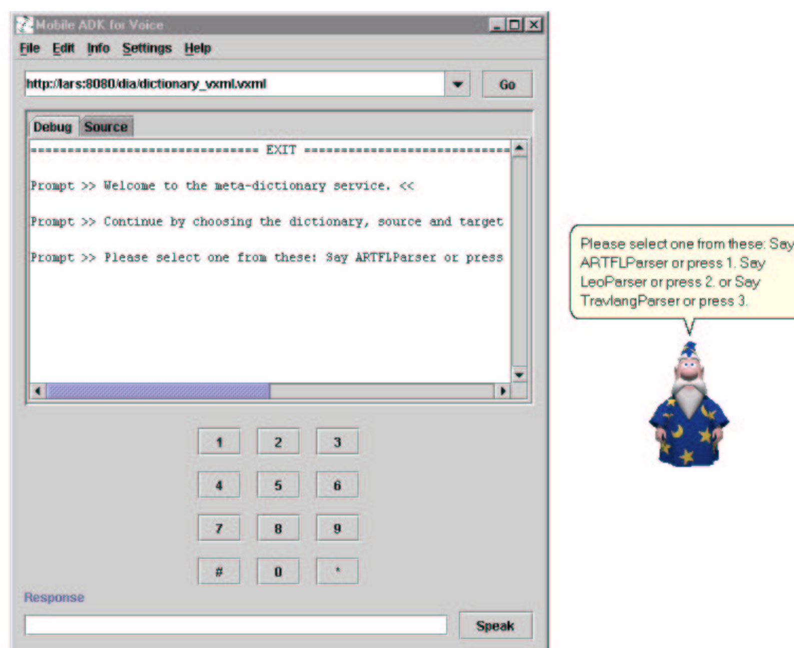


Abbildung 5.12: Screenshot des Meta-Dictionaries (VXML)

des Wörterbuches zu realisieren, muss lediglich die Ausgabe der Delegateinstanz für das aktuelle Wörterbuch veranlasst werden (Zeile 2). Es ist ferner notwendig, die serverseitige Auswertung der neuen Daten zu fordern (Zeile 3-5), da mit Auswahl des Wörterbuchs die zur Verfügung stehenden Sprachen festgelegt werden. In der URL dieses Requests ist zusätzlich enkodiert, das mit der Eingabeeinheit zur Anfragenspezifikation (`#query`) fortgefahren werden soll. Dazu werden zunächst die Eingabemöglichkeiten für die Quell- und Zielsprache und für den zu übersetzenden Ausdruck[7] ausgegeben (Zeilen 14-16) und anschließend wieder eine serverseitige Auswertung der Benutzereingaben angefordert (Zeilen 21-23). Es ist in diesem Fall notwendig, zusätzlich zu den Namen der Eingaben auch den Namen des Übersetzungsdelegates als Parameter anzugeben, da ansonsten die entsprechende Übersetzung nicht ausgelöst wird. Weiterhin muss wie schon bei WML eine Methode zur Konversion von Vesuf Namen eingeführt werden, da VXML auf ECMA-Skript basiert und damit ebenfalls keine Punkte innerhalb von Variablennamen zulässt.

In ähnlicher Art und Weise wurden auch Eingabeeinheiten für die Präsentation der Ergebnisse und für eine Menusteuerung realisiert. Die Menusteuerung ermöglicht dem Benutzer, nach der Durchführung einer Übersetzung eine Auswahl zu treffen, wie er fortfahren möchte.

[7]In VXML ist es in Version 1.0 nach Kenntnisstand der Autoren nicht möglich eine Grammatik zu definieren, die beliebige Worte als Eingaben akzeptiert. Um dennoch die Eingabe beliebiger Wörter als zu übersetzenden Ausdruck zu ermöglichen, haben die Autoren eine Grammatik spezifiziert, die es erlaubt eine beliebig lange Abfolge von Buchstaben zu definieren: +[a b c d e . . .]. Das entsprechende Eingabeelement interpretiert diese Buchstaben als zusammenhänges Wort.

```
01:  <form id="dictionary">
02:    $this.print("CurrentDictionary_delegate")
03:    <block>
04:      <submit method="post" next="$baseurl#query"/>
05:    </block>
06:  </form>
07:
08:  <form id="query">
09:    #set($frpid = $this.getSubpartInstance("From_delegate"))
10:    #set($topid = $this.getSubpartInstance("To_delegate"))
11:    #set($expid = $this.getSubpartInstance("Expression_delegate"))
12:    #set($dopid = $this.getSubpartInstance("Translate_delegate"))
13:
14:    $this.printSubpartInstance($frpid)
15:    $this.printSubpartInstance($topid)
16:    $this.printSubpartInstance($expid)
17:
18:    <block>
19:    <var name="$this.getVXMLName($dopid)" expr="'-'"/>
20:      <prompt>Please be patient while processing request.</prompt>
21:      <submit method   = "post" next="$baseurl#results"
22:              namelist = "$this.getVXMLName($frpid) $this.getVXMLName($topid)
23:                         $this.getVXMLName($expid) $this.getVXMLName($dopid)"/>
23:    </block>
24:  </form>
```

Abbildung 5.13: VXML-Template (Ausschnitt)

```
01: # Properties for the dictionary awt/html/wml/vxml application
02:
03: # Name of application model
04: name               = dictionary.dictionary_application_awt/html/wml/vxml
05:
06: # Model specifications
07: objectmodel        = dictionary.DictionaryObjectModel
08: navigationmodel    = org.vesuf.model.navigation.DefaultDialogModel
09: presentationmodel  = dictionary.dictionary_awt/html/wml/vxml
10:
11: # Navigation refinement and starting machine
12: dialogmachine      = DefaultDialogMachine
13: initobject         = <static>SearchViewTask.<create>().<value>()
14:
15: # Presentation mapper
16: presentationmapper = org.vesuf.model.application.PropertyPresentationMapper
17: mapper_properties  = /dictionary/dictionary_mapping.properties
```

Abbildung 5.14: Appliationsdeskriptoren des Metawörterbuchdienstes

5.3.4 Dialogsteuerung und Applikationsdeskriptor

Für jeden Interaktionsstil eines Dienstes muss für das Vesuf System ein Applikationsdeskriptor erstellt werden. Abbildung 5.14 zeigt die Applikationsdeskriptoren des Metawörterbuchdienstes, wobei die Zeilen, die sich zwischen den einzelnen Interaktionsstilen unterscheiden, kursiv hervorgehoben sind.

Die Applikationsdeskriptoren unterscheiden sich im Namen (Zeilen 1, 4) und im zu verwendenden Präsentationsmodell (Zeile 9). Alle vier Anwendungen basieren auf dem DictionaryObjectModel (Zeile 7). Da der Metawörterbuchdienst nur einen Dialog (SearchViewDialog) definiert, verwendet er als Dialogmodell

das DefaultDialogModel (Zeile 8), das die DefaultDialogMachine (Zeile 12) spezifiziert. Dieses Standard Dialogmodell ist bereits Teil des Vesuf Systems (s. Abschnitt 4.2.4.3) und muss daher nicht zusätzlich erstellt werden. Als `initobject` für das Dialogmodell wird eine Instanz des SearchViewTasks erzeugt (Zeile 13). Zur Abbildung von Domänenentitäten und Dialogzuständen auf Präsentationselemente wird ein PropertyPresentationMapper verwendet (Zeile 16). Die in Zeile 17 angegebene Mappingtabelle enthält lediglich eine Abbildung, die den einzigen Task des Dienstes auf den im Präsentationsmodell spezifizierten toplevel Dialog abbildet: `SearchViewTask = DictionaryDialog`.

5.4 Z39.50 Dienste

In die Kategorie der Z39.50 Dienste falle alle Services, die das Z39.50 Protokoll zur Kommunikation einsetzen. Daher wird nachfolgend zunächst auf die grundlegenden Eigenschaften dieses Protokolls eingegangen. Das Z39.50 Protokoll ist ein internationaler Standard ANSI/NISO Z39.50-1995 [ANSI/NISO 1995] (ISO 23950), der die Kommunikation zwischen Computersystemen vereinfachen soll. Er basiert auf dem Client / Server Modell und spezifiziert Aktivitäten zur Informationssuche und zum Informationserhalt. Der Standard definiert Z39.50 dabei als OSI-Anwendungsprotokoll, d. h. als Protokoll der Ebene 7 des ISO/OSI-Netzwerkmodells [ISO 1977].

Durch die Verwendung des Z39.50 Protokolls wird von den eingesetzten Datenbanken, der lokalen Querysyntax, dem jeweiligen Betriebssystem und der Hardware abstrahiert. Man kann sich das Z39.50 Protokoll als eine Art Datenbank-Esperanto vorstellen, das jedem Client einen Dialog mit jeder Datenbank ermöglicht.

Z39.50 Dienstleistungen werden durch Nachrichtenaustausch (request, response) zwischen Z-Client (origin) und Z-Server (target) erbracht. Der Client initiiert eine stehende Verbindung zu einem Server und authentifiziert sich, wenn dies notwendig ist. Der Server bearbeitet die Anfrage und liefert die Anzahl gefundener Treffer zurück. Im nächsten Schritt kann die Clientseite nun eine bestimmte Auswahl an Datensätzen anfordern. Beendet wird die Verbindung explizit durch den Client oder durch Zeitconstraints des Servers. Während einer Sitzung bleiben sämtliche Voreinstellungen des Clients erhalten.

Z39.50 ist in elf grundlegende strukturelle Einheiten (facilities) unterteilt[8], die sich wiederum aus einem oder mehreren Diensten (services) zusammensetzen. Dienste haben die Aufgabe, bestimmte Operationen zwischen Origin und Target zu vereinfachen und werden von einer Z39.50 Applikation eingesetzt, um ihre Funktionalität zu realisieren. Die Basisdienste, die in den meisten Z39.50 Applikationen zu finden sind, setzen sich aus Initialisierung (initialization), Suche (search) und Ergebnisabfrage (present) zusammen. Aufgabe der Initialisierungsphase ist es, die Verbindung eines Origins mit einem Target herzustellen und gewisse Randbedingungen wie z. B. die Art der Querybearbeitung festzulegen.

Mit Hilfe des Search-Dienstes wird es dem Origin möglich gemacht, Anfragen an das Target zu senden. Diese Anfragen können in ihrer Komplexität stark

[8]Die elf strukturellen Einheiten werden wie folgt bezeichnet: Initialization, Search, Retrieval, Result-set-delete. Browse, Sort, Access Control, Accounting/Resource Control, Explain, Extended Services, Termination.

variieren und aus Boolschen Teilausdrücken zu beliebigen Binärbäumen zusammengesetzt werden. Um die Anfragen unabhängig von speziellen Datenbanken festlegen zu können, werden bei einem Z39.50 Query spezielle Query-Attribute (access points) eingesetzt. Diese Query Attribute sind innerhalb einer standardisierten domänenspezifischen Attributmenge (attribut set) definiert, die bei der Konstruktion einer Anfrage angegeben werden muss. Zunächst speziell für die bibliografische Domäne vorgesehen und später durch eine Reihe von Ergänzungen im Anwendungskontext wesentlich flexibler ist der Bib-1 Attributsatz.

Der Ergebnisabfrage-Service wird dazu verwendet Treffer einer Anfrage in einem bestimmten standardisierten Format (record syntax) vom Target abzurufen. So ist es beispielsweise durchaus möglich die ersten Ergebnisse in UKMARC und alle weiteren Ergebnisse in USMARC abzurufen. Unterstützt das Target die geforderte Record Syntax nicht, kann es die Ergebnisse in einem anderen selbstbestimmten Format zurücksenden [Miller 1999].

Besondere Verbreitung hat Z39.50 in der Welt der Bibliotheksanwendungen gefunden. So gibt es weiter standardisierte domänenspezifische Query-Attributsätze, wie z. B. Bib-1, Exp-1, GILS und standardisierte Ergebniseinträge, wie z. B. MARC-Record mit diversen lokalen Derivaten UK-MARC, US-MARC, DAN-MARC. Nichtsdestoweniger ist der Z39.50 Standard nicht auf eine bestimmte Domäne beschränkt.

5.4.1 Domänenmodell

Die für einen Z39.50 Dienst durchgeführte Aufgabenanalyse führte zur Identifikation von drei Hauptaufgaben, die dem Anwender verfügbar gemacht werden sollten. Er muss in die Lage versetzt werden, eine Verbindung zu einem bestimmten Datenbestand seines Interesses aufzubauen (ConnectTask). Anschließend muss es ihm ermöglicht werden, sich aus diesem Datenbestand durch Anfragen interessante Teile heraussuchen (SearchingTask) und diese präsentieren zu lassen (PresentTask).

Das Ergebnis der Domänenanalyse ist das in Abbildung 5.15 in Form eines UML-Klassendiagramms dargestellte Domänenmodell. Es besteht im Wesentlichen aus den drei Aufgaben und den für das User Interface benötigten Domänenklassen. In den Notizen (UML-Notes) wurden in Kurzform Einschränkungen (Constraints, siehe Abschnitt 4.2.3.2) zu einigen Elementen definiert.

Der ConnectTask speichert Informationen, die zum Aufbau einer Verbindung benötigt werden. Um dem Benutzer ein einfaches Handling zu ermöglichen, führen die Autoren sogenannte Profile ein, die alle notwendigen Informationen zum Verbindungsaufbau exklusive evtl. notwendiger Authentifizierungseinstellungen bündeln. Die Klasse ConnectTask verwaltet die verschiedenen Profile durch einen Namen und besitzt daher das Attribut Pnames, über das alle abgelegten Profile eindeutig referenziert werden können. Durch das Attribut Profile wird kenntlich gemacht, welches Profil gerade aktiviert ist. Um dem Anwender auch die Möglichkeit zu geben, nicht in den Profilen gespeicherte Server anzusprechen, werden alle Verbindungseinstellungen auch separat abgelegt. Dazu gehören die Serveradresse (Address) mit dem Zugriffsport (Port) und die Authentifizierungsdaten, welche sich aus Benutzername (Username) und dem Passwort (Password) zusammensetzen. Um schliesslich eine Verbindung mit den gegebenen Parametern aufzubauen, wird die Operation Connect verwendet. Als

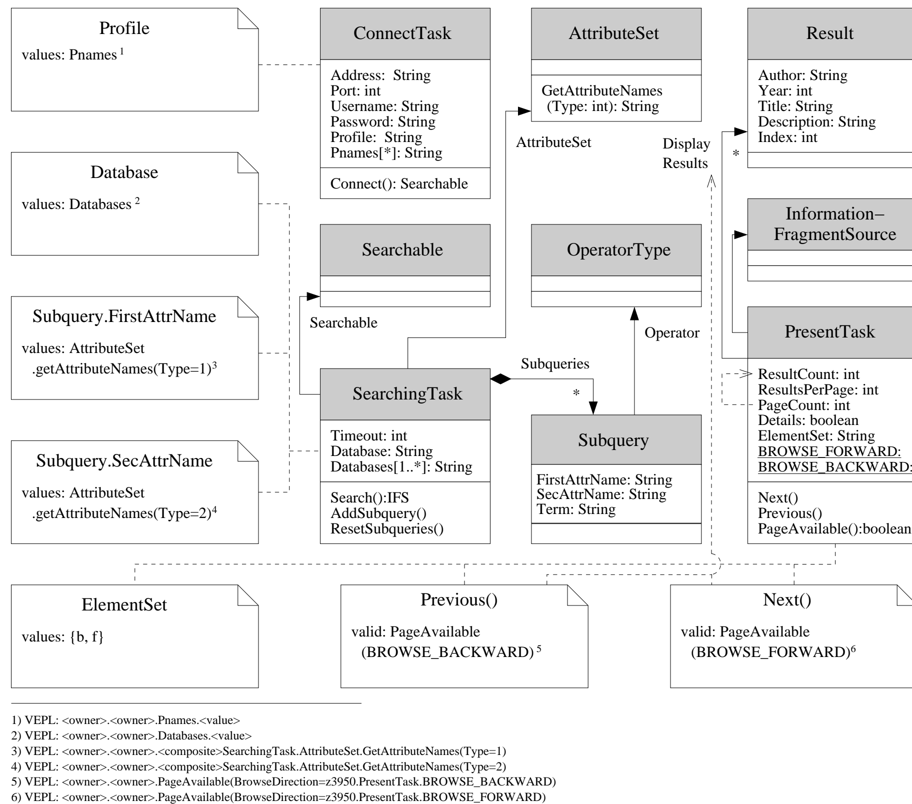


Abbildung 5.15: Domänenmodell des Z39.50 Dienstes

Rückgabewert liefert diese Operation ein in Vesuf modelliertes Searchable Objekt, das im JZKit Framework implementiert ist.

Der SearchingTask beinhaltet Informationen zur Suche nach bestimmten Datensätzen in einer Datenbank durch die Spezifikation von Anfragen. Der Name der zu durchsuchenden Datenbank wird im Attribut Database abgelegt. Diese Datenbank kann aus den durch den Server zur Verfügung gestellten Datenbanken ausgewählt werden. Zur Speicherung der Namen sämtlicher Datenbanken eines Servers wird das multiplizitäre Attribut Databases eingesetzt. Neben der Angabe einer Zieldatenbank für eine Anfrage muss spezifiziert werden, wonach gesucht werden soll. Dazu werden eine beliebige Anzahl verknüpfter Subqueries eingesetzt. Der SearchingTask besitzt aus diesem Grund eine gerichtete, multiplizitäre Kompositionsbeziehung in Richtung der Klasse Subquery.

Ein Subquery selbst repräsentiert eine atomare simplifizierte Abfrage, die aus zwei Einstellungen und einem Operator zur Verknüpfung mit dem nachfolgenden Subquery besteht. In den zwei Einstellungen können zwei Attribute des mit dem SearchingTask assoziierten AttributeSets festgelegt werden. Verwendet der SearchingTask z. B. die Bib1-AttributSet Spezifikation, kann die erste Einstellmöglichkeit (FirstAttrName) ein Attribut der Bib-1 Use Attributes und die zweite Einstellmöglichkeit (SecAttrName) ein Attribut der Bib-1 Relation Attributes qualifizieren. Der Operator zur Verknüpfung der Ausdrücke wurde als spezieller Aufzählungstyp (enumeration type) in einer eigenen Klasse (Ope-

ratorType) definiert.

Die Anzahl der für eine Abfrage zu kombinierenden Subqueries ist durch den Benutzer frei wählbar. Zu diesem Zweck führen die Autoren die Operationen AddSubquery und ResetSuqueries ein. Im Initialzustand besitzt ein SearchingTask genau ein Subquery. Durch den Aufruf der AddSubquery Operation wird dem SearchingTask ein Subquery hinzugefügt. Mittels ResetSubqueries wird der Ausgangszustand mit einem leeren Subquery wiederhergestellt.

Zusätzlich zur Anfrage selbst und dem Anfrageziel ist es möglich, eine Zeitschranke für den Request festzulegen. Innerhalb dieser im Attribut Timeout des SearchingTasks definierten Zeitspanne muss eine Antwort des Servers erfolgen, andernfalls wird die Suchanfrage mit einer Fehlermeldung abgebrochen. Zur Durchführung einer Abfrage wird die Search Operation des SearchingTasks verwendet. Als Ergebnis liefert sie ein InformationFragmentSource zurück, dessen Implementation ebenfalls aus dem JZKit Framework stammt.

Der PresentTask enthält Informationen, die zur Darstellung der Suchergebnisse benötigt werden. Er besitzt folglich eine gerichtete multiplizitäre Assoziation zu den aktuell präsentierten Ergebnissen (DisplayResults). Da die Quantität der Ergebnisse (ResultCount) sehr unterschiedlich sein kann und sehr große Ergebnismengen möglich sind, ist es sinnvoll, nicht alle Ergebnisse, sondern kleinere Teilmengen darzustellen. Die Anzahl der Ergebnisse, die maximal zu einer Teilmenge gehören, kann vom Benutzer konfiguriert werden und wird im Attribut ResultsPerPage gespeichert. Die Anzahl der Ergebnismengen, die durch die Aufteilung entsteht, wird im Attribut PageCount festgehalten. Mittels der Operationen Next und Previous kann die nächste oder vorhergehende Ergebnismenge ausgewählt werden. Abgelegt werden die neuen Ergebnisse in den DisplayResults.

Die Einstellungen Details und ElementSet dienen dazu, die Art der Ergebnisdarstellung zu verändern. Durch die Aktivierung der Details werden zusätzlich sämtliche Informationen zu einem Ergebniseintrag als fortlaufender Text dargeboten. Mit Hilfe des ElementSets kann zwischen einer ausführlichen oder knappen Darstellung ausgewählt werden.

Ein Ergebnis (Result) beinhaltet alle Eigenschaften, die zur Präsentation eines Eintrags notwendig sind. Dazu gehören der Autor (Author), der Titel (Title) und das Erscheinungsjahr (Year) einer Publikation. Des weiteren enthält das Attribut Description alle verfügbaren Informationen zu diesem Ergebnis in textueller Form. Weiterhin existiert ein Attribut Index, das die Nummer des Results in der Ergebnismenge beschreibt.

5.4.1.1 Constraints

Das Attribut Profile des ConnectTasks ist mit einem Constraint versehen, das die zulässigen Werte dieses Attributes einschränkt. Die Wertemenge ist als Pfad auf den Wert des Attributes Pnames definiert. Da Pnames als Attribut mit Multiplizität modelliert ist, können beliebig viele gültige Werte eingetragen werden. Auf dem SearchingTask sind ebenfalls Constraints definiert. So ist auch ein Values-Constraint für das Attribut Database durch die Angabe eines Pfades auf den Wert des Attributes Databases festgelegt.

Des weiteren werden zwei Constraints mit dem SearchingTask modelliert, die sich auf die assoziierten Subqueries beziehen. Dies ist möglich, da es sich um eine Kompositionsbeziehung handelt (siehe Abschnitt 4.2.3.2). Es werden

Values-Constraints für die Attribute FirstAttrName und SecAttrName des Subqueries eingeführt, die abhängig sind von dem AttributSet des SearchingTasks. Um dies zu erreichen wird ein Pfad definiert, der über das Kompositionselement (SearchingTask) die Operation getAttributeNames des assoziierten AttributeSets zugreift. Der Parameter Type dieser Operation wählt aus, welche Attributgruppe eines AttributeSets abgerufen wird.[9] Wird z. B. das AttributSet Bib-1 verwendet, referenzieren die als Konstanten eingetragene Werte (eins, zwei) die Bib-1 Use- und die Bib-1 Relation-Attributes.

Auch für den PresentTask wurden Constraints festgelegt. Der einfache Werte-Constraint für das Attribut ElementSet schreibt als einzig gültige die zwei konstanten Werte {b, f} vor. Komplexer sind die Einschränkungen für die Operationen Next und Previous des PresentTasks. Für jede Operation führen die Autoren je ein Valid-Constraint ein, das anzeigt, wann der Aufruf der betroffenen Operation zulässig ist. Für jedes Constraint wird ein Pfad definiert, der auf der PageAvailable Operation des PresentTasks endet. Der einzige Unterschied zwischen den beiden Constraints ist der Wert des Parameters, mit dem die PageAvailable Operation aufgerufen wird. Dieser Parameter legt fest, in welche Richtung von der aktuellen Position fortgesetzt werden soll. Daher können die Parameterwerte als Konstanten in den Pfad eingetragen werden.

5.4.1.2 Dependencies

Im Domänenmodell sind außer den Klassen auch die identifizierten Abhängigkeiten (Dependencies, siehe Abschnitt 4.2.3.3) zwischen den Elementen eingetragen. So existiert eine direkte Abhängigkeit zwischen den Attributen PageCount und ResultCount des PresentTasks. Diese Dependency bedeutet, dass Änderungen des Attributs ResultCount auch allen Observern des Attributs PageCount mitgeteilt werden. Des weiteren müssen die Constraints für die Operationen Next und Previous erneut ausgewertet werden, wenn sich die DisplayResults ändern. Folglich gibt es eine Abhängigkeit zwischen diesen Constraints und den DisplayResults.

5.4.2 Implementation

Die Implementation des Z39.50 Dienstes stützt sich auf das Opensource Framework JZKit [Ibbotson et al. 2001], das zum Aufbau von Information-Retrieval Systemen auf Basis des Z39.50 Protokolls entwickelt wurde. Im folgenden wird daher zunächst ein grober Überblick über das Framework gegeben. Anschließend werden die Autoren darauf eingehen, welche funktionalen Ergänzungen vorgenommen werden mußten und wie die Vesuf Implementation umgesetzt wurde.

5.4.2.1 JZKit

Das JZKit Framework besitzt eine Schichtenarchitektur mit drei Schichten auf unterschiedlichen Abstraktionsniveaus: die Kodierungsschicht (encoding layer), die Protokollendpunktschicht (protocol endpoint layer) und die Informationsabfrageschicht (information retrieval layer). Die Kodierungsschicht bietet eine

[9]Mit Attributgruppe bezeichnen die Autoren eine Menge zusammengehörige Attribute eines AttributSets. Z. B. gibt es im Bib-1 AttributSet die Attributgruppen Use-, Relation-, Position-, Structure-, Transaction- und Completness-Attributes.

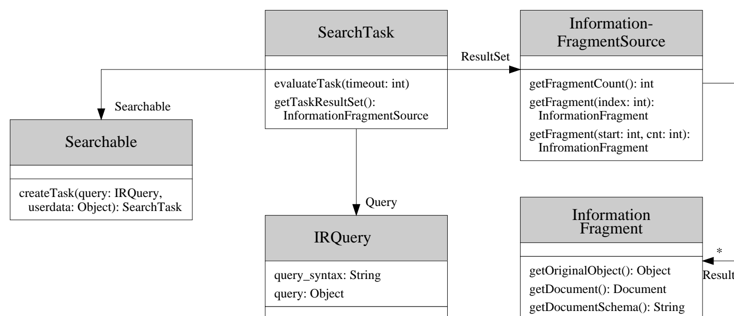


Abbildung 5.16: Information Retrieval Modell des JZKit Frameworks

Umsetzung des Z39.50 Protokolls mit den low-level Schnittstellen. Die Protokollendpunktschicht kann von reinen Z39.50 Applikationen verwendet werden und stellt bereits Komponenten für die einfache Anwendungskonstruktion bereit. Noch weiter abstrahiert die Informationsabfrageschicht, welche die Art des zugrundeliegenden Protokolls transparent macht [Ibbotson 2001].

In Abbildung 5.16 sind die am Information Retrieval beteiligten Komponenten des JZKit Frameworks gezeigt. Dabei haben die Autoren zugunsten einer übersichtlichen Darstellung darauf verzichtet, die Elemente vollständig spezifiziert aufzuführen. Alle Komponenten, die eine Suche ermöglichen, sind vom Typ Searchable. Jedes Searchable Objekt ist in der Lage, für eine Anfrage – bestehend aus einem IRQuery und optionalen Benutzerdaten – einen SearchTask zu generieren. Der SearchTask erlaubt, die Anfrage auf dem Searchable durchzuführen (mittels `evaluateTask()`) und die Ergebnismenge in Form einer InformationFragmentSource abzurufen (mittels `getTaskResultSet()`). Einzelne Ergebnisse oder Teilergebnismengen können vom InformationFragmentSource (mittels der `getFragment()` Methoden) angefordert werden. Des weiteren kann abgefragt werden, wieviele Treffer die Suche insgesamt ergeben hat (mittels `getFragmentCount()`). Ein einzelnes Ergebnis wird durch ein InformationFragment gekapselt. Es macht den genauen Typ des Originalergebnisrecords zugänglich (mittels `getDocumentSchema()`) und erlaubt, das Ergebnis sowohl in der Originalform (mittels `getOriginalObject()`) als auch (mittels `getDocument()`) in einer DOM-Repräsentation (siehe [le Hors et al. 2000]) weiterzuverarbeiten.

5.4.2.2 Z39.50 Tasks

Der ConnectTask realisiert die Verwaltung der Profileinstellungen von Z39.50-Servern und kann die Verbindung zu einem ausgewählten Ziel herstellen. Um die Z39.50-Serverprofile flexibel handhaben zu können und Erweiterungen bzw. Modifikationen unabhängig von der Anwendung durchführen zu können, werden die Einstellungen in einem Propertyfile separat abgelegt. Die Syntax dieser Spezifikation ist wie folgt:

```
# address: internetaddress
# port: accessport
# dbname: database name
profilename = address[:port][/dbname[,dbname]]
```

```
01:  /** Connect to the target. */
02:  public Searchable connect()
03:  {
04:    // Build init request.
05:    Properties initparms = new Properties();
06:    initparms.put("ServiceHost", this.address);
07:    initparms.put("ServicePort", ""+this.port);
08:    if(username.length()!=0 && password.length()==0)
09:    {
10:      initparms.put("service_auth_type", "2"); // Authentication with id.
11:      initparms.put("service_user_principal", username);
12:    }
13:    else if(username.length()!=0 && password.length()!=0)
14:    {
15:      initparms.put("service_auth_type", "3"); // Authent. with id and passwd.
16:      initparms.put("service_user_principal", username);
18:      initparms.put("service_user_credentials", password);
20:    }
21:    Searchable origin = new Z3950Origin(); // Create a Z39.50 origin.
22:    origin.init(initparms);
23:    // Test connection (as is not done in jzkits init).
24:    try
25:    {
26:      IRQuery query = new IRQuery();
27:      query.collections = new Vector();
28:      query.collections.addElement("default");
29:      query.query_syntax = "PREFIX";
30:      query.query = "@attrset bib-1 @attr 1=1 test";
31:      origin.createTask(query, null).evaluate(5000);
32:    }
33:    catch(SearchException e){}
34:    catch(Exception e)
35:    {
36:      throw new RuntimeException("Connection failed: "+e);
37:    }
38:    return origin;
39:  }
```

Abbildung 5.17: Connect-Methode des ConnectTasks

Um nach der Initialisierung der Profile entscheiden zu können, welcher Server defaultmäßig eingestellt werden soll, wird eine mit „initial“ betitelte Property herangezogen. Diese enthält den Namen des zuerst zu aktivierenden Profils. Ändert der Benutzer durch Interaktion das aktuelle Profil, führt dies zum Aufruf der `setProfile()` Methode. Diese Methode sorgt dafür, dass sowohl der Profilname als auch die dadurch betroffenen Einstellungen Serveradresse und Port angepasst werden. Es werden folglich am Methodenende Änderungen für alle drei Einstellungen über das EventDispatcherProxy propagiert.

Die eigentliche Aufgabe des ConnectTasks – nämlich eine Verbindung mit einem Server herzustellen – wurde innerhalb der `connect()` Methode realisiert (s. Abb. 5.17). Zu diesem Zweck wird zunächst ein Objekt vom Typ Searchable (Z3950Origin) mit den Verbindungsparametern erzeugt (Zeilen 21-22). Die Verbindungsparameter werden in einem Properties Objekt unter im JZKit Framework vordefinierten Bezeichnungen abgelegt (Zeilen 5-20). Dazu gehören neben den obligatorischen Einstellungen der Serveradresse (`ServiceHost`) und dem Port (`ServicePort`) auch die evtl. notwendigen Authentifizierungsinformationen. Abhängig davon, ob nur ein Benutzername (`service_user_principal`) oder ein Be-

```
01:  /** Build a query tree form subqueries. */
02:  public IRQuery buildQuery()
03:  {
04:    // Create and initialize a query.
05:    IRQuery query = new IRQuery();
06:    query.hints = new Hashtable();
07:    query.hints.put("record_syntax", recordsyntax);
08:    query.collections = new Vector();
09:    query.collections.addElement(database);
10:    query.query_syntax = "INTERNAL";
11:
12:    RootNode rn = new RootNode();
13:    rn.setAttrset(attributeset.getName());
14:    Subquery sq = (Subquery)subqueries.elementAt(0);
15:    AttrPlusTermNode ch = (AttrPlusTermNode)rn.getChild();
16:    int[] id = attributeset.getAttributeId(sq.firstattrname);
17:    ch.setAttr(null, new Integer(id[0]), new Integer(id[1]));
18:    if(isAvailable(sq.secattrname))
19:    {
20:      id = attributeset.getAttributeId(sq.secattrname);
21:      ch.setAttr(null, new Integer(id[0]), new Integer(id[1]));
22:    }
23:    ch.setTerm(sq.term);
24:
25:    for(int i=1; i<subqueries.size(); i++)
26:    {
27:      // When there is more than one subquery
28:      // add the operator from the former subquery.
29:      ComplexNode cn = rn.expandChild(rn.getChild());
30:      cn.setOp(sq.operator);
31:      sq = (Subquery)subqueries.elementAt(i);
32:      AttrPlusTermNode rhs = (AttrPlusTermNode)cn.getRHS();
33:      id = attributeset.getAttributeId(sq.attributenamegroup1);
34:      rhs.setAttr(null, new Integer(id[0]), new Integer(id[1]));
35:      if(isAvailable(sq.secattrname))
36:      {
37:        id = attributeset.getAttributeId(sq.secattrname);
38:        ch.setAttr(null, new Integer(id[0]), new Integer(id[1]));
39:      }
40:      rhs.setTerm(sq.term);
41:    }
42:    query.query = rn;
43:    return query;
44:  }
```

Abbildung 5.18: BuildQuery-Methode des SearchingTasks

nutzername mit einem Passwort (`service_user_credentials`) spezifiziert sind, wird der Authentifikationsmodus (`service_auth_type`) automatisch richtig eingestellt (Zeilen 8-20).

JZKit führt derzeit keinen Test der Verbindung zum Server während der Initialisierung durch.[10] Um sicherzustellen, dass das Ziel erreichbar ist, testet daher die `connect()` Methode durch ein Placebo-Query die Verbindung selbst (Zeilen 23-37). Die Verbindung ist fehlerhaft, wenn die Anfrage zu einer Exception führt, die nicht vom Typ SearchException ist.

[10] Die JZKit Klasse Z3950Origin besitzt bereits eine Methode zum Testen einer Verbindung (checkConnection), die aber nur von der evaluateTask Methode verwendet wird. Der Sichtbarkeitsbereich dieser Methode gestattet ihre Benutzung von aussen nicht.

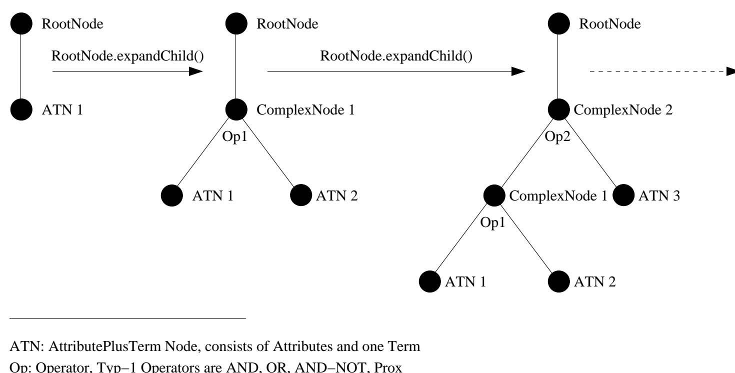


Abbildung 5.19: Typ-1 Query Konstruktion

Der SearchingTask dient dazu, eine Anfrage aus den Benutzereingaben zu konstruieren und an einen Server zu versenden. Er ist zur Zeit darauf ausgelegt, Typ-1 Queries aufzubauen. Mit geringen Modifikationen lässt sich aber auch die erweiterte Form der Typ-101 Queries realisieren. Gemäss der Z39.50 Spezifikation [ANSI/NISO 1995] sind Typ-1 Queries beliebige binäre RPN-Bäume.[11] Da jedoch die Möglichkeit zur Eingabe beliebiger RPN-Strukturen die Komplexität der Erfassung unangemessen heraufsetzen würde, beschränken sich die Autoren darauf, nur eine spezielle Art der Abfrage zu gestatten. Sie setzen ein Query aus einer Reihe sogenannter Subqueries zusammen, die durch Operatoren mit dem Vorgänger verknüpft sind. Die Anzahl der Subqueries beträgt initial eins. Um ein weiteres Suchkriterium hinzuzufügen, wird die Methode `addSubquery()` verwendet. Außerdem kann der Ausgangszustand durch Aufruf der `resetSubqueries()` Methode wiederhergestellt werden.

Die eigentliche Konstruktion eines JZKit Queries wird in der `buildQuery()` Methode durchgeführt (s. Abb. 5.18). Zunächst wird ein IRQuery Objekt des JZKit Frameworks erzeugt und mit einigen Parametern initialisiert (Zeilen 4-10). Die Recordsyntax legt fest, welche Form das Ergebnis des Queries aufweisen soll (Zeilen 6-7). Weiterhin wird festgehalten, in welcher Datenbank die Suche erfolgen soll (Zeilen 8-9). Mit Angabe der Querysyntax wird schließlich bestimmt, in welcher Form das Query spezifiziert wird. JZKit unterscheidet `INTERNAL` als Kennung für einen fertigen RPN-Query und `PREFIX` für eine Zeichenkette in Prefixnotation, die noch durch einen Parser in einen RPN-Baum umgewandelt werden muss. Da die `buildQuery()` Methode ein RPN-Query selbst erstellt, wird die Syntax auf `INTERNAL` festgeschrieben.

Basis der Query-Konstruktion ist eine RootNode (Zeile 12), welche solange um Subquery Blätter erweitert wird, bis der komplette Baum aufgebaut ist. Alle Blätter des Baumes sind vom JZKit-Typ AttrPlusTermNode. Diese Art von Knoten ist geeignet, alle Informationen eines einzelnen Suchausdrucks aufzunehmen. Dazu gehören die verwendeten Attribute und der Ausdruck (term), auf den sich die Attribute beziehen. Erster Schritt zum Aufbau des Queries ist daher, die Informationen vom ersten Suchausdruck in die Childnode einzutragen (Zeilen 13-23). Anschliessend wird über alle weiteren Suchausdrücke iteriert. In jeder Iteration wird der Baum wie in Abb. 5.19 dargestellt expandiert und die

[11] RPN = Reverse Polish Natation.

```
01  /**  Retrieve the results to be displayed. */
02:  protected void  retrieveResults()
03:  {
04:    int old = 0;
05:    if(displayresults!=null)
06:       old = displayresults.length;
07:
08:    // Update displayresults.
09:    int count = resultset.getFragmentCount();
10:    int start = resultsperpage * (currentpage-1) + 1;
11:    if(start>count)
12:       displayresults  = new Result[0]; // Shouldn't happen.
13:    else
14:    {
15:      InformationFragment[] ifs = resultset.getFragment(start,
16:        Math.min(count - start + 1, resultsperpage));
17:      this.displayresults = new Result[ifs.length];
18:      for(int i=0; i<ifs.length; i++)
19:        this.displayresults[i] = Result.create(ifs[i], start+i);
20:    }
21:
22:    // Inform EventDispatcherProxy about changes.
23:    dispatcher.collectEvents();
24:    if(old>displayresults.length)
25:    {
26:      for(int i=old-1; i>=displayresults.length; i--)
27:         dispatcher.removeElement("DisplayResults", i);
28:    }
29:    else if(old<displayresults.length)
30:    {
31:      for(int i=old; i<displayresults.length; i++)
32:         dispatcher.addElement("DisplayResults", i);
33:    }
34:    dispatcher.valueChanged("DisplayResults", displayresults);
35:    dispatcher.commitEvents();
36:  }
```

Abbildung 5.20: Retrieve-Methode des PresentTasks

Suchinformationenen werden in einem neuen Blatt untergebracht (Zeilen 25-41). Die Konstruktion der weiteren Blätter ist identisch zu der Vorgehensweise, die bereits am ersten Suchausdruck vorgestellt wurde. Die Verknüpfung zweier Blätter wird über den jeweilig übergeordneten Knoten hergestellt, der vom JZKit Typ ComplexNode ist. Eine ComlexNode besitzt zwei Child-Knoten und einen Operator, der die Art der Verknüpfung festlegt. In jeder Iterationsstufe wird daher der Operator des vorangehenden Subqueries in die übergeordnete ComplexNode eingetragen.

Der PresentTask setzt die Darstellung der Ergebnisse einer Anfrage um. Wie bereits erwähnt, werden Einzelergebnisse vom JZKit Framework als InformationFragments zurückgeliefert. Es sind allerdings noch keine Spezialisierungen dieser Klasse vorhanden, die den verschiedenen Ergebnistypen Rechnung tragen. Die Autoren erweitern daher die Ergebnisdarstellung um Klassen, die direkt wichtige Ergebnistypen repräsentieren und zugleich bibliographische Eckdaten (Autor, Titel, Erscheinungsjahr) auf eine einheitliche Weise zugänglich machen. Sie führen zu diesem Zweck die Klassen MarcResult, GRSResult, SutrsResult und OpacResult ein, die allesamt auf der abstrakten Result Klasse basieren. Diese Klasse stellt eine Factory-Methode `create()` bereit, die zur Erzeugung ei-

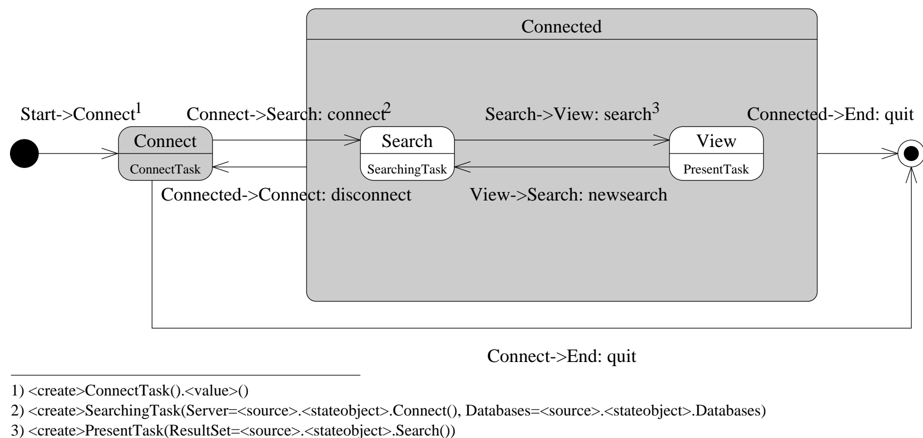


Abbildung 5.21: Dialogmodell des Z39.50 Dienstes

nes Result Objektes für ein beliebiges InformationFragment verwendet werden kann. Anhhand des Dokumentenschemas erzeugt diese Methode den richtigen konkreten Ergebnistyp.

Eine zentrale Aufgabe des PresentTasks ist es, die aktuelle Ergebnismenge vom Server zu holen. Zu diesem Zweck existiert im PresentTask die `retrieveResults()` Methode (s. Abb. 5.20), welche die aktuell dargestellte Ergebnismenge (displayresults) aktualisiert. Grundlage der Beschaffung von Einzelergebnissen ist das vorläufige Ergebnis der Suchanfrage, die InformationFragmentSource (resultset). Diese wird angewiesen, eine Anzahl Ergebnisse von einem Startindex ausgehend zu holen (Zeile 15-16). Der Startindex (Zeile 10) ergibt sich aus der Anzahl Ergebnisse auf einer Seite (resultsperpage) und der aktuellen Seitennummer (currentpage). Die Anzahl der abzufragenden Ergebniseinträge ist mit Ausnahme der letzten Seite stets durch die Variable resultsperpage festgelegt. Soll die letzte Seite dargestellt werden, kann es vorkommen, dass weniger Einträge vorhanden sind als die Seite eigentlich fassen kann (Zeile 16). Für die so erhaltenen InformationFragments werden über die bereits vorgestellte Factory Methode `create()` der Klasse Result typabhängige Ergebnisobjekte erzeugt (Zeilen 18-19).

Die Notifikation des EventDispatcherProxys über Änderungen muss zwei Gesichtspunkten Rechnung tragen. Da sich alle Einzelergebnisse geändert haben, wird die gesamte Elementmenge (displayresults) als modifiziert propagiert (Zeile 34). Zusätzlich muss der Grösse des multiplizitären Elements Rechnung getragen werden. Hat sich die Anzahl enthaltener Entitäten verkleinert (Zeile 24) oder vergrössert (Zeile 29), werden entsprechende Events generiert. Da diese Überprüfung zu einer Vielzahl an Benachrichtigungen führen kann, werden die Events gesammelt (Zeile 23) und im Bündel verschickt (Zeile 35).

5.4.3 Dialogsteuerung

Das Dialogmodell des Z39.50 Dienstes ist in Abbildung 5.21 dargestellt. Es setzt sich zusammen aus den drei einfachen Zuständen Connect, Search und View und dem komplexen Zustand Connected, der die einfachen Zustände Search

und View einfasst. Die Modellierung der groben Dialogsteuerung ist stark beeinflusst von den Ergebnissen der Aufgabenanalyse. Die Autoren haben sich in diesem Fall dazu entschieden, jeder in der Taskanalyse identifizierten Aufgabe einen eigenen Dialogzustand zuzuweisen, d. h. sie ordnen dem Connect-Task den Connect-Zustand, dem SearchingTask den Search-Zustand und dem PresentTask den View-Zustand zu. Den Zuständen werden daher Stateobjects zugewiesen, die diesem Mapping entsprechen. Eine Separation des Search- und View-Dialogs erscheint uns für diesen Dienst sinnvoll, da die Ergebisse nicht nur visualisiert, sondern vom Benutzer auch in der Art der Darstellung modifiziert werden können. Der Presenttask besitzt folglich genug Funktionalitäten, die einen eigenständigen Dialog rechtfertigen. Einzelheiten zu allgemeinen Aspekten des Task-Dialog Mappings finden sich in Abschnitt 4.3.1.

Ausgehend vom Startzustand wird der Zustand Connect automatisch eingenommen (Start->Connect). Die Transitionsanschift dieses Übergangs beschreibt den internen Datenfluss zwischen diesen Zuständen (siehe Fussnote 1 in Abbildung 5.21). Dem Zustand Connect wird daher eine neue Instanz eines Connect-Tasks zugewiesen. Im Connect-Zustand werden dem Benutzer die Interaktionsmöglichkeiten des ConnectTasks dargeboten. Er kann also alle Einstellungen vornehmen, die zu einem Verbindungsaufbau notwendig sind.

Der Aufbau einer Verbindung wird durch einen Connect-Event initiiert. In der Folge wird der Subzustand Search des Connected-Zustandes eingenommen (Connect->Search: connect). Auch dieser Übergang ist mit einer Transitionsanschrift versehen (siehe Fussnote 2 in Abbildung 5.21). Dem Search-Zustand wird ein neu instantiierter SearchingTask als Stateobject übergeben. Da der SearchingTask mit dem Server, der die Anfragen bearbeitet und den Namen der darauf vorhandenen Datenbanken erzeugt werden muss, beinhaltet diese Stateobjekt-Anschrift im Parameterteil weitere Pfadangaben. Die Navigation erfolgt ausgehend von der Transition zum Ursprungszustand (mittels `<source>` zum Connect-Zustand) und von dort zum Stateobjekt (mittels `<stateobject>` zum Connecttask). Das Stateobject des Connect-Zustandes ist ein Connecttask. Der entsprechende Server kann nun als Rückgabewert erhalten werden, indem die Connect Operation aufgerufen wird. Die Namen der verfügbaren Datenbanken sind direkt vom ConnectTask über ein Attribut referenzierbar (Databases).

Da mit dem Search-Zustand der Searchingtask assoziiert ist, kann der Anwender Anfragen definieren und abschicken. Soll eine neue Anfrage bearbeitet werden, wird durch einen Search-Event der aktuelle Zustand verlassen und der View-Zustand betreten (Search->View: search). Die Stateobject-Anschrift dieser Transition bewirkt, dass der Present-Zustand eine neue Instanz eines PresentTasks erhält (siehe Fussnote 3 in Abb. 5.21). Der PresentTask erwartet bei der Erzeugung eine Ergebnismenge (ResultSet) als Parameter. Aus diesem Grund wird als Parameter ein Pfad deklariert, der mit der Search Operation des SearchingTasks endet. Der Rückgabewert dieser Methode ist das entsprechende ResultSet.

Um es einem Anwender zu gestatten, weitere Anfragen aufzubauen, muss er erneut in den Dialog zur Requestspezifikation schalten können. Es ist deswegen möglich, vom View-Zustand zurück in den Search-Zustand zu gelangen, wenn ein Newsearch-Event eintrifft (View->Search: newsearch).

Unabhängig davon, in welchem Zustand sich die Dialogmaschine gerade befindet, soll sie mit einem Quit-Event beendet werden können. Um dies zu erreichen, werden Transitionen mit dem Quit-Event sowohl vom Connect-Zustand

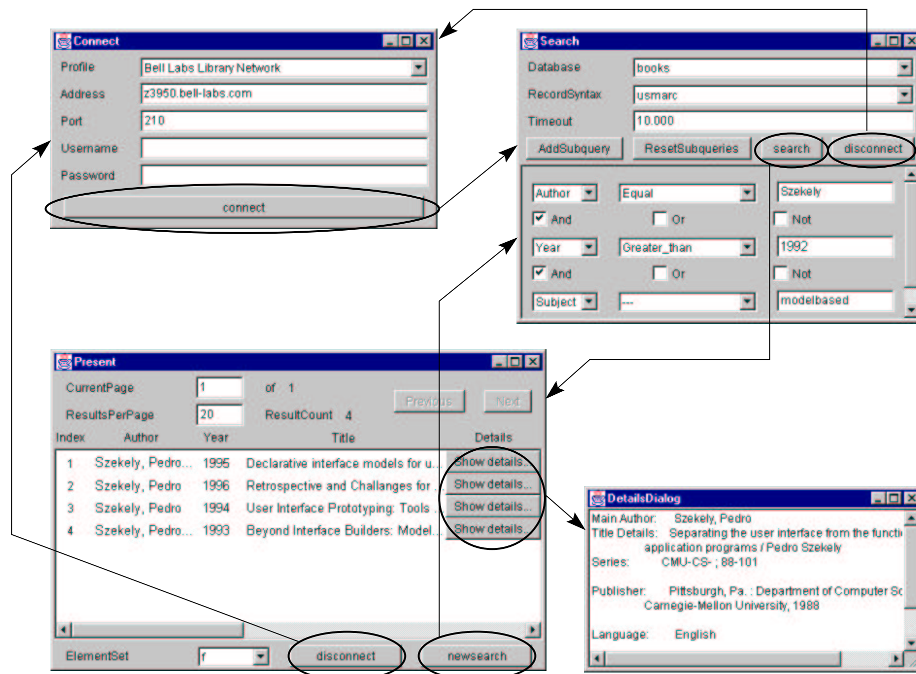


Abbildung 5.22: Snapshots der Z39.50 AWT Anwendung

(Connect->End: quit) als auch vom Connected-Zustand (Connected->End: quit) konstruiert. Die Deklaration der Quit-Transition, ausgehend vom Connected-Zustand, beinhaltet, dass auch die enthaltenen Subzustände (hier der Search- oder View-Zustand) mitverlassen werden.

5.4.4 Präsentationsmodelle

Das Präsentationsdesign für den Z39.50 Dienst wurde für eine AWT und eine HTML Benutzungsoberfläche durchgeführt, auf die nachfolgend näher eingegangen wird. Die Autoren vermeiden es jedoch, sämtliche Details eines Präsentationsmodells vorzustellen, da viele Definitionen sehr ähnlich sind und ihre Erklärung keine weiteren Erkenntnisse bringt. Stattdessen greifen sie neuralgische Punkte heraus, an denen sie neue Konzepte erläutern werden.

5.4.4.1 AWT-Version

Das Präsentationsmodell für die AWT-Version des Z39.50 Dienstes besteht aus vier Hauptdialogen (Connect-, Searching-, Present- und Details-Dialog), die jeweils als einzelne Fenster (Frames) realisiert wurden. Ein Snapshot dieser Fenster während der Programmausführung ist in Abbildung 5.22 gezeigt. Neben der visuellen Repräsentation der einzelnen Dialoge enthält die Abbildung auch die mögliche Navigation zwischen den Dialogen.

Der Connect-Dialog ist einfach aufgebaut und besteht im wesentlichen aus vier mit Führungstexten versehenen Textfeldern, die zur Eingabe der Zugangsdaten Serveradresse, Serverport, Benutzername und Passwort dienen. Des wei-

```
01:  <Delegate name              = "ToConnect(Navigation)_delegate"
02:            event             = "connect"
03:            layoutconstraints = "GridBagConstraints(0,5,2,1,1,1,CENTER,
04:                                 HORIZONTAL,Insets(10,10,0,10),0,0)"
05:            widgettype        = "ButtonEvent"/>
```

Abbildung 5.23: Navigationselement im Connect-Dialog

teren wurde ein Auswahlelement (Choice) für die Selektion eines Profils vorgesehen. Unterhalb der Eingabeelemente wurde ein Knopf für die Initialisierung einer Verbindung positioniert (Connect-Button). Dieser Knopf ist ein Navigationselement und kann einen Zustandswechsel der aktiven Dialogmaschine durch einen Event auslösen. Die Spezifikation eines Navigationselements erfordert daher stets die Angabe des Events, der bei Aktivierung gefeuert werden soll. In Abb. 5.23 ist der relevante Ausschnitt zur Deklaration des Connect-Knopfes dargestellt. Es handelt sich in diesem Fall um ein Delegate (Zeile 1) mit Widgettyp ButtonEvent (Zeile 5). Der zu propagierende Event ist durch seinen Namen eindeutig bestimmt und wird hier mit `connect` definiert (vgl. Abb. 5.21).

Der Searching-Dialog besteht aus drei Bereichen. Im oberen Teil befinden sich Interaktionselemente für die Einstellung allgemeiner Suchparameter mit den wiederum vorangestellten Führungstexten. Zur Bestimmung der Datenbank und des gewünschten Ergebnisformats wurden Auswahlelemente vorgesehen. Des weiteren kann über ein Textfeld eingestellt werden, wie groß die maximale Zeitspanne ist, nach dessen Überschreitung ein Verbindungsversuch als gescheitert angesehen wird (Timeout).

Im unteren Bereich wird die Anfrage (Query) spezifiziert. Sie besteht aus einer Liste, welche die Subqueries darstellt. Die Beschreibung der Liste ist in Abbildung 5.24 dargestellt. Es wird von einem homogenen Listeninhalt ausgegangen, der durch die prototypische Beschreibung eines Exemplars ausreichend definiert ist.[12] Jedes Subquery wiederum wird durch einen eigenen Dialog repräsentiert (Zeile 5-29), dessen Pfad auf ein durch den Index eindeutig bestimmtes Element zeigt (Zeile 7). Er besteht aus zwei Auswahlelementen für die Attributselektion (Zeilen 9-18), einem Textfeld für den Term (19-23) und einer Auswahlgruppe (CheckboxGroup) für den Operator (Zeilen 24-28).

Zwischen den beiden Einstellungsbereichen wurden im Mittelteil Buttons angeordnet. Die AddSubquery und ResetSubqueries Knöpfe sind direkt (über VEPL-Pathes) auf Operationen des Domänenmodells gemappt, welche die Anzahl der Subqueries um eins erhöhen bzw. auf den Initialzustand zurücksetzen. Der Knopf Disconnect ist mit einem Navigationsevent verbunden und führt dazu, dass der Dialog Connect angezeigt wird. Der Searchbutton repräsentiert ebenfalls ein Navigationselement und ist nicht, wie der Name suggerieren könnte, direkt mit der Search Operation verbunden. Stattdessen löst dieses Element den Dialogübergang zum Present-Fenster aus, in dessen Folge auch die Search Operation bei der Pfadevaluierung aufgerufen wird (siehe Z39.50 Dialogmodellbeschreibung, Abschnitt 5.4.3).

Das Herzstück des Present-Dialogs ist die Ergebnisliste, in der eine einstellbare Anzahl Einzelergebnisse auf einmal dargestellt werden. Sie wurde konzeptuell

[12] In den meisten Fällen stellt dies keine grosse Einschränkung dar. Gerade aber für die Darstellung dieses Dienstes ist es ungünstig, da dem Anwender jeweils genau Subqueries-1 Operatoren präsentiert werden sollten.

```
01:  <Delegate name              = "Subqueries"
02:            path              = "Subqueries"
03:            layoutconstraints = "Center"
04:            widgettype        = "List">
05:    <Dialog name                = "Subquery"
06:            widgettype          = "Panel"
07:            path                = "&lt;index&gt;"
08:            layout              = "GridBagLayout()">
09:      <Delegate name              = "FirstAttrName_delegate"
10:                path              = "FirstAttrName"
11:                layoutconstraints = "GridBagConstraints(0,0,1,1,0,1,WEST,
12:                                     NONE,Insets(5,10,0,10),0,0)"
13:                widgettype        = "Choice"/>
14:      <Delegate name              = "SecAttrName_delegate"
15:                path              = "SecAttrName"
16:                layoutconstraints = "GridBagConstraints(1,0,1,1,0,1,WEST,
17:                                     NONE,Insets(5,10,0,10),0,0)"
18:                widgettype        = "Choice"/>
19:      <Delegate name              = "Term_delegate"
20:                path              = "Term"
21:                layoutconstraints = "GridBagConstraints(2,0,1,1,1,1,WEST,
22:                                     HORIZONTAL,Insets(5,10,0,10),0,0)"
23:                widgettype        = "TextField"/>
24:      <Delegate name              = "Operator_delegate"
25:                path              = "Operator"
26:                layoutconstraints = "GridBagConstraints(0,2,3,1,1,1,WEST,
27:                                     HORIZONTAL,Insets(5,10,0,10),0,0)"
28:                widgettype        = "CheckboxGroup"/>
29:    </Dialog>
30:  </Delegate>
```

Abbildung 5.24: Subquery-Liste des Searching-Dialogs

```
01:  <Delegate name          = "Details_delegate"
02:            path          = "&lt;index&gt;.&lt;value&gt;"
03:            dialogmachine = "DefaultDialogMachine"
04:            label         = "Show details..."
05:            widgettype    = "ButtonReference"/>
```

Abbildung 5.25: Detail-Knopf eines Ergebnisses im Present-Dialog

in gleicher Weise spezifiziert, wie die Liste des Searching-Dialogs. Jedes Einzelergebnis wird textuell durch die bibliographischen Eckdaten Autor, Titel, Erscheinungsjahr, Index bezogen auf die Gesamtergebnismenge und einem Detail-Knopf visualisiert. Der Detail-Knopf hat die Aufgabe, alle verfügbaren Angaben, die zu einem Ergebnis existieren, in einem separaten Dialog darzustellen und den weiteren Zugriff auf die Ergebnisliste nicht zu verhindern (s. Abb. 5.25). Um dies zu erreichen, startet das zugrundeliegende Dialogelement einen neuen Kontrollfluss (siehe Abschnitt 4.2.4.2). In diesem Fall wird ein Delegate (Zeile 1) mit Widgettyp ButtonReference (Zeile 5) definiert. Dieses Element erfordert die Spezifikation der einzusetzenden Dialogmaschine (Zeile 3) und des initialen Stateobjektes (Zeile 2). Für das Detail-Fenster reicht es aus, die bereits in Vesuf vorspezifizierte DefaultDialogMachine einzusetzen und das initiale Objekt durch eine Pfadangabe auf den Wert eines Einzelergebnisses festzulegen.

Oberhalb der Ergebnisliste befinden sich mit Führungstexten versehene Einstellungen, mit denen sich der Umfang (resultsperpage) und die Auswahl der zu präsentierenden Ergebnisse (currentpage) festlegen lässt. Durch den Umfang

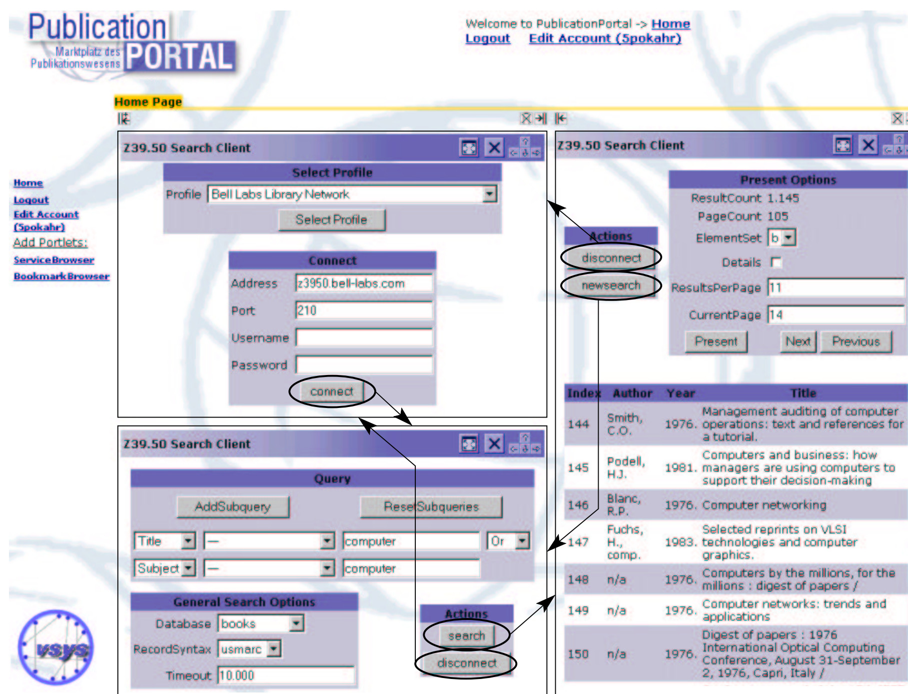


Abbildung 5.26: Snapshots der Z39.50 HTML Anwendung

lässt sich die Anzahl der Treffer regulieren, die gleichzeitig dargestellt wird. Die Auswahl bestimmt dann, welche Teilergebnismenge präsentiert werden soll. Für beide Parameter wurden Textfelder vorgesehen. Zusätzlich kann die Auswahl durch Knöpfe zum Vor- und Zurückblättern (Previous, Next) in komfortabler Form modifiziert werden.

Im unteren Bereich wurde die Navigation zu den Dialogen Search und Connect über Navigationsknöpfe realisiert. Des weiteren wurde dort ein Auswahlelement mit Führungstext plaziert, mit dem das ElementSet bestimmt werden kann.

5.4.4.2 HTML-Version

Die HTML-Version ist in ihrer Bedienung der AWT-Version sehr ähnlich. Da das Vesuf System zur Zeit keine Nebenläufigkeit in HTML-Benutzungsschnittstellen unterstützt, wurde auf einen Details-Dialog jedoch verzichtet. Stattdessen kann man im Present-Dialog zwischen einer kompakten und einer detaillierten Darstellungsweise wählen (s. Abb. 5.26).

Wie beim Metawörterbuchdienst werden die Präsentationselemente in einem UIML-Präsentationsmodell spezifiziert, während der eigentliche HTML-Code von Templates erzeugt wird. Für jeden der drei top-level Dialoge (Connect, Search und Present) wird ein eigenes Template definiert (s. Abb. 5.27). Der Interpreter des Vesuf Systems (z. B. VesufServlet oder VesufPortlet) bestimmt bei jedem Zugriff durch einen Browser den aktuellen Zustand der Dialogsteuerung und den zu diesem Zustand gehörigen Dialog (siehe Abschnitt 4.2.2.3). Über das Template des jeweiligen Dialogs wird dann die entsprechende HTML-Seite generiert. Es gibt Interaktionsmöglichkeiten, die zwar zu einem Neuladen der HTML-

```
01: <interface name="Z3950-Client">
02:   <structure>
03:
04:     <Dialog name       = "ConnectDialog"
05:             path       = "ConnectTask"
06:             widgettype = "Template"
07:             template   = "z3950/connect.vm">
08:       ...
09:     </Dialog>
10:
11:     <Dialog name       = "SearchingDialog"
12:             path       = "SearchingTask"
13:             widgettype = "Template"
14:             template   = "z3950/searching.vm">
15:       ...
16:     </Dialog>
17:
18:     <Dialog name       = "PresentDialog"
19:             path       = "PresentTask"
20:             widgettype = "Template"
21:             template   = "z3950/present.vm">
22:       ...
23:     </Dialog>
24:
25:   </structure>
26: </interface>
```

Abbildung 5.27: Z39.50 Präsentationsmodell (HTML-Version)

```
01: #if($element.getAttributeLink("Details").Value.Object)
02:   #set($desc  = $list.getSubpartInstance("Description_delegate", $key))
03:   <TD bgcolor="$bgcol">
04:     <PRE>$this.printSubpartInstance($desc)</PRE>
05:   </TD>
06: #end
```

Abbildung 5.28: HTML-Template des PresentDialogs (Ausschnitt)

Seite führen (z. B. die Next und Previous-Operationen des PresentTasks), aber die den aktuellen Dialogzustand nicht verändern. Andere Präsentationselemente, wie die Navigationselemente (connect, search, newsearch, disconnect) führen zu einer Änderung des Dialogzustandes, was zur Folge hat, dass die HTML-Seite des Folgedialogs generiert wird.

Abbildung 5.28 zeigt einen Auschnitt aus dem Template für den Present-Dialog. Das dynamische Einblenden der Details erfolgt mit Hilfe des Velocity `#if` Konstrukts (Zeilen 1-6). Über die vom Vesuf System im Templatekontext abgelegte `$element` Referenz hat das Template Zugriff auf die jeweiligen Domänenentitäten. Das Domänenelement des Presentdialoges ist der PresentTask. Das Template kann direkt den Wert des Details-Attribut des PresentTasks abfragen (Zeile 1, vgl. Abb. 5.15) und davon abhängig den dargestellten Inhalt variieren. Zeile 2 referenziert aus dem Präsentationsmodell das Description-Delegate zur Darstellung der Details eines Ergebnismengeneintrages. Dieses wird in Zeile 4, eingeschlossen in Schlüsselworte zur vorformatierten Ausgabe (`PRE` Tags), in das generierte Dokument eingefügt.

```
01:  # Properties for the z3950 awt/html application
02:
03:  # Name of application model
04:  name               = z3950.z3950_application_awt/html
05:
06:  # Model specifications
07:  objectmodel        = z3950.Z3950ObjectModel
08:  navigationmodel    = z3950.Z3950DialogModel
09:  presentationmodel  = z3950.z3950_awt/html
10:
11:  # Navigation init and starting machine
12:  dialogmachine      = Z3950DialogMachine
13:  #initobject is not needed.
14:
15:  # Presentation mapper
16:  presentationmapper = org.vesuf.model.application.PropertyPresentationMapper
17:  mapper_properties  = /z3950/z3950_mapping.properties
```

Abbildung 5.29: Applikationsdeskriptoren für AWT/HTML Version

```
01: Connect = ConnectDialog
02: Connected.Search = SearchingDialog
03: Connected.View = PresentDialog
04: Result = DetailsDialog
```

Abbildung 5.30: Mapping properties des Z39.50 Dienstes

5.4.5 Applikationsdeskriptor

Die vollständigen Applikationsdeskriptoren für die AWT und HTML Version des Z39.50 Dienstes sind in Abbildung 5.29 gezeigt. Es ist notwendig, für jeden Applikationstyp einen eigenen Applikationsdeskriptor zu verwenden. Da sich die beiden Deskriptoren aber nur marginal unterscheiden, stellen die Autoren beide in einer Abbildung dar und kennzeichnen die Teile, in denen Unterschiede zu Tage treten (Zeilen 1, 4 und 9).

Mit Hilfe einer Benennung (Zeile 4) wird das Applikationsmodell respektive der Applikationsdeskriptor eindeutig indentifizierbar. Die verwendeten Modelle (Zeile 7-9) unterscheiden sich nur beim Präsentationsmodell, das für beide Applikationstypen getrennt entworfen wurde (siehe Abschnitt 5.4.4). Da sie das gleiche Dialogmodell benutzen, ist die Festlegung der Dialogmaschine (Zeile 12) und des dazugehörigen Initparameters (Zeile 13) identisch. Die Deklaration des Initparameters kann entfallen, da die Dialogmaschine beim Starten selbst durch die Evaluierung eines Pfades eine Instanz des Connect-Tasks erzeugt (siehe Abschnitt 5.4.3). Die Abbildung der Zustände auf Dialoge erfolgt durch den PropertyPresentationMapper (Zeile 16). Dieser greift auf eine deklarierte Abbildungstabelle (Zeile 17) zu, die in Abbildung 5.30 dargestellt ist.

Kapitel 6

Zusammenfassung und Ausblick

Ubiquitous Computing wird die Anwendungsentwicklung in Zukunft stark beeinflussen. Laut einer in [Avedik 2001] beschriebenen Diebold-Studie wird bis zum Jahr 2005 über die Hälfte aller Internetzugriffe von mobilen Geräten aus erfolgen. Die Vielzahl der unterschiedlichen Geräte führt dazu, dass sich die heute bereits existierende Heterogenität an Zielplattformen für die Anwendungsentwicklung weiter verstärken wird [Huang et al. 1999]. Einige Autoren wie z. B. [Grimm et al. 2001] ziehen daraus den Schluss, dass herkömmliche Ansätze des Anwendungsentwurfs für UbiComp nicht direkt geeignet sind, da Anwendungen für bestimmte Klassen von Geräten oder Plattformen entworfen werden, was zu verschiedenen Versionen einer Anwendung für Handhelds und Desktops führt. Stattdessen benötigt man eine Plattform, auf der Anwendungen geräteunabhängig entwickelt werden können.

Ziel dieser Arbeit war ein System zur Entwicklung von Benutzungsschnittstellen für Applikationen des UbiComp. Es ermöglicht die geräteunabhängige Anwendungsentwicklung auf Basis von heterogenen Internet Diensten und stellt universelle Zugriffsmöglichkeiten auf die Anwendungen sicher. Als Nachweis der praktischen Einsatzfähigkeit wurden beispielhaft Dienste des Publikationswesens angepasst und im Rahmen des Global Info Projektes in das Publication-PORTAL integriert.

6.1 Ergebnisse der Forschung

Die Autoren haben im Bereich der User Interface Erstellung von Applikationen des UbiComp zunächst potentielle Systemgrundlagen ausgewertet. Es stellte sich heraus, dass Schichtenarchitekturen generell für UbiComp angemessen sind, da sie ein erhöhtes Maß an Portierbarkeit von Applikationen auf verschiedene Endgerätetypen durch die Austauschbarkeit von einzelnen Schichten ermöglichen. Als alleinige Grundlage für ein UbiComp System reichen sie aufgrund ihrer mangelhaften Adressierung der Komplexität interaktiver Anwendungen jedoch nicht aus.

Im Bereich der Techniken hat sich gezeigt, dass im besonderen Maße spezifikationsbasierte Entwurfstechniken gute Voraussetzungen für den Einsatz in Ubi-

Comp Systemen aufweisen. Sie unterstützen die essentiell wichtige Flexibilität dieser Systeme durch ihren deskriptiven Charakter maßgeblich. Automationstechniken sind zwar nicht immer praktisch ausgereift, stellen aber insbesondere zu den spezifikationsbasierten Techniken eine attraktive Ergänzung dar.

Bei der Evaluierung der verschiedenen Werkzeugklassen, haben die Autoren festgestellt, dass insbesondere diejenigen mit einer vollständigen Abdeckung aller Aspekte einer Benutzungsschnittstelle für die Konstruktion von UbiComp Applikationen wertvoll sind. Dies ist dadurch begründet, dass die Anbindung des User Interfaces an die Fachlogik ein schwieriger Arbeitsschritt ist, der durch die Verwendung von Systemen vereinfacht werden sollte. Des weiteren ist gerade im Hinblick auf UbiComp eine hohe Flexibilität und Austauschbarkeit von Benutzungsschnittstellen für eine Applikation notwendig. Die Autoren haben daher vielversprechende Werkzeuge der Kategorien Frameworks und modellbasierter Systeme genauer unter die Lupe genommen.

Es bleibt festzustellen, dass alle untersuchten Frameworks zu unflexibel sind, um UbiComp Anwendungen zu entwickeln. Keines der frameworkbasierten Systeme ist in der Lage, mehr als eine Interfacemodalität pro Applikation zu verwalten. Die modellbasierten Systeme (MB-UIDEs) sind zwar prinzipiell geeignet, aber in der Praxis mussten die Autoren feststellen, dass ebenfalls keine Unterstützung von mehreren Interfacemodalitäten geboten wird. Hinzu kommt in der Regel eine hohe Lernschwelle für User Interface Entwickler, da die Systeme u. a. nicht auf Standards zur User Interface Spezifikation zurückgreifen können.

6.2 Umgesetzte Zielvorgaben

Im folgenden greifen die Autoren die in Abschnitt 1.3 gesetzten Ziele auf, und erläutern wie das Vesuf System den Anwendungsentwickler diesen Zielen näherbringt.

(1) Separation und Verbindung Um die Benutzungsschnittstelle vom Anwendungskern zu separieren verwenden die Autoren eine Adapterkomponente. Diese Komponente repräsentiert als Domänenschicht unseres Systems die benutzerzentrierte Sicht auf eine Anwendung. Die Adapterkomponente besitzt eine allgemeine Schnittstelle auf Basis des UML-Metamodells, die für alle Arten von Anwendungen gleich ist. Die Benutzungsschnittstellenelemente sind über in die Domänenschicht generisch integrierte Konzepte wie Pfade, Constraints und Dependencies mit dem Anwendungskern verknüpft. Dies ermöglicht die flexible Anbindung von Benutzungsschnittstellen vollkommen losgelöst von konkreten Implementationstechnologien. Das UML-Metamodell bildet somit eine robuste und skalierbare Grundlage unserer Architektur.

(2) Vereinfachung Das Vesuf System führt zu einer Vereinfachung der Anwendungsentwicklung. In den einzelnen Schichten mit klar voneinander abgegrenzten Funktionalitäten können Teile von Anwendungen unabhängig voneinander entwickelt werden. Dem Entwickler bleibt dabei selbst überlassen, welche Schichten einer Anwendung geräteunabhängig entworfen und welche spezifischen Endgeräten angepasst werden. Automatisierungsansätze nehmen dem Entwickler weitere Arbeit ab und fördern ein Rapid-Prototyping von Anwendungen.

[Banavar et al. 2000] sagen „developing a device-independent application is inherently more complex than developing a device specific one.“ Mit Vesuf ist dies nicht mehr der Fall. Anwendungsteile, die natürlicherweise geräteunabhängig sind (z. B. der Funktionale Kern) werden geräteunabhängig entwickelt. Geräteabhängige Teile können gezielt für einzelne Geräte entwickelt oder von Automationswerkzeugen generiert werden. Zur Vereinfachung trägt auch bei, dass unser Ansatz konsequent auf etablierte Standards, insbesondere UML setzt. Ein sich daraus ergebender Vorteil ist die Kompatibilität zu verbreiteten Entwicklungswerkzeugen wie Rational Rose.

(3) Erweiterbarkeit Die einzelnen Schichten des Systems sind durch klare Schnittstellen voneinander abgegrenzt. Damit lassen sich verschiedene Implementationstechniken für einzelne Schichten integrieren, ohne Abhängigkeiten zu anderen Schichten zu erzeugen. Innerhalb der Präsentationsschicht wurde auf eine kompakte flexible Architektur (Visual Proxy) gesetzt, die es erlaubt, mit minimalem Aufwand neue Interfacemodalitäten zu unterstützen (siehe auch Abschnitt 6.3). Darüber hinaus ist die Architektur aufgrund der deklarativen Modelle offen für zusätzliche Design-, Laufzeit-, und Automationswerkzeuge.

(4) Flexibilität Das Vesuf System stellt eine universelle Plattform zur Verfügung, die die Ausführung beliebiger Anwendungen als Dienste ermöglicht. Für diese können auf einfache Weise verschiedene Arten von Benutzungsschnittstellen umgesetzt werden. Benutzungsschnittstellen sind direkt auf die Geräte (z. B. als Java-Applet oder JavaWebStart-Anwendung) übertragbar. Über einen Server ist auch der entfernte Zugriff durch diverse Endgeräte möglich (z. B. Web-Browser, WAP-Handy, Telefon). Damit können beliebige Geräte als Zugang (Portal) zu diesen Diensten genutzt werden.

(5) Adaption und (6) Komposition Auch wenn das Vesuf System in seiner aktuellen Form weder Adaption noch Komposition unterstützt, sind die Autoren der Überzeugung, das der modellbasierte Ansatz für diese Aspekte prädestiniert ist. Zur dynamischen Adaption von Benutzungsschnittstellen könnten z. B. Umgebungsmodelle und Modelle der Fähigkeiten der Benutzer und Endgeräte in Vesuf integriert werden. Diese Modelle würden die gezielte Anpassung laufender Anwendungen an aktuelle Gegebenheiten ermöglichen.
Die deklarative Natur aller Vesuf Modelle erleichert die Entwicklung von Strategien zur dynamischen Komposition von spezifizierten Anwendungen.

6.3 Fallstudie

Das PublicationPORTAL will verteilte heterogene Dienste unter einer einheitlichen Oberfläche zusammenfassen. Um dies zu erreichen, wurde das Vesuf System über den Portletmechanismus in die Präsentationskomponente des Portals integriert. Unser Ansatz ermöglicht die Homogenisierung der einzubindenen Dienste durch die abstrakten Modellierungsschichten. Damit steht im PublicationPORTAL nun ein allgemeiner Mechanismus zur Verfügung, um neue Dienste unter

Vereinheitlichung der Oberfläche zugänglich zu machen. Darüber hinaus unterstützt das Vesuf System bereits die Bereitstellung verschiedener User Interfaces zugeschnitten auf diverse Endgeräte, so dass einer Erweiterung des Portals, um mobilen und ubiquitären Zugang zu ermöglichen, nichts mehr im Wege steht.

Die Fallstudie stellt nicht nur die flexible Einsetzbarkeit des Vesuf Systems bezüglich der Dienstprotokolle (Z39.50, HTTP) unter Beweis und verdeutlicht, wie leicht es ist, verschiedene Benutzungsschnittstellen für einen Dienst zu spezifizieren. Die Tatsache, dass die Integration zweier neuer Interfacemodalitäten (WAP bzw. Voice) in das Vesuf System in wenigen Tagen erfolgte, belegt eindrucksvoll die einfache Erweiterbarkeit.

6.4 Ausblick

In Abschnitt 4.4 wurden bereits die nächsten möglichen Schritte zur Weiterentwicklung des Vesuf Systems z. B. um weitere Modelle oder Entwicklungswerkzeuge angesprochen. Zum Abschluss wird im folgenden noch ein Überblick über die Themen gegeben, die im Kontext dieser Arbeit nur gestreift werden konnten und die in zukünftigen Projekten addressiert werden sollten.

So konnten zwei der identifizierten Anforderungen für UbiComp Benutzungsschnittstellensysteme (Adaption und Komposition) nicht ausreichend betrachtet werden. Die Komposition von Benutzungsschnittstellen ist indes nicht allein im Kontext des UbiComp von Interesse, sondern stellt auch ein offenes Problem im Rahmen der generativen Programmierung (s. z. B. [Griffel et al. 2001]) dar. Das PublicationPORTAL verwendet das generative Paradigma zur dynamischen Komposition von Diensten aus kleineren Einheiten anhand der vom Benutzer spezifizierten Anforderungen. Es liegt also nahe, in weiteren Projekten die Kompositionsfunktionalität über den modellbasierten Ansatz auf die Benutzungsschnittstelle auszudehnen. In diesem Zusammenhang sind dann auch Strategien von Interesse, die dem Benutzer interaktive Konfigurationsmöglichkeiten bezüglich der spezifizierten Anwendungen geben (end user programming).

Weitere Aspekte im Kontext des UbiComp, die im Rahmen dieser Arbeit nicht berücksichtigt werden konnten, sind die dynamische Verteilung von Anwendungskomponenten (apportionment) zwischen Client und Server und multimodale, d. h. zur gleichen Zeit über verschiedene Modalitäten (z. B. Eingabe durch Sprache, Ausgabe über Display) interagierende Anwendungen.

Die Autoren sind der Ansicht, dass alle genannten Problemstellungen mit dem Vesuf Ansatz zu verwirklichen sind. Aufgrund seiner offenen auf Standards basierenden Architektur ist das Vesuf System in idealer Weise als Grundlage für weitere Forschungsprojekte verwendbar. Im Rahmen eines einzelnen Projektes ist das Potential eines solchen Systems bei weitem nicht auszuschöpfen. Um das System einer breiten Öffentlichkeit zugänglich zu machen, ist es daher unter `http://sourceforge.net/projects/vesuf` als Open Source Projekt verfügbar.

Anhang A

VEPL Referenz

Regeln der VEPL Shorthand Notation

1. `<value>` kann weggelassen werden, wenn es für die Fortsetzung des Pfades unabdingbar ist.

2. `<attribute>` kann immer weggelassen werden.

3. `<operation>` kann immer weggelassen werden.

4. `<static>type.<create>(...)` kann durch `<create>type(...)` abgekürzt werden.

5. `<operation>op(para1:type1, para2:type2, ...).<value>(arg1, arg2, ...)`
   kann abgekürzt werden durch
   `<operation>op(para1:type1 = arg1, para2:type2 = arg2, ...)`.

6. Bei `para:type` kann wahlweise `para` oder `:type` weggelassen werden, solange die Operation eindeutig identifizierbar ist.

```
# EBNF (Extended Backus Naur Form) Specification of VEPL (VEsuf Path Language)

# Initial path elements
path    = classifier_ref | typedelement_ref | operation_ref |
          constraint_ref | state_ref | statemachine_ref | static

# References
classifier_ref   = attribute | operation | composite | create | static
typedelement_ref = attr_value | owner | constraint |static
operation_ref    = op_value | owner | constraint |static
constraint_ref   = owner | state
state_ref        = statemachine | stateobject
statemachine_ref = context

# Path elements
# type_name: A name of a model classifier (fully qualified).
# name: Simple name of a modelelement.
# key: Is a string identifier for an operation instance.
attribute    = "<attribute>" name ("."typedelemenent_ref)?
operation    = "<operation>" name "(" parameters? ")"
               ("$"key)? ("."operation_ref)?
constraint   = "<constraint>"name ("."constraint_ref)?
parameter    = "<parameter>"name ("."typedelement_ref)?
composite    = "<composite>(" type_name ")" ("."classifier_ref)?
static       = "<static>" type_name ("." classifier_ref)?
source       = "<source>" state_ref
stateobject  = "<stateobject>" classifier_ref | typedelemenent_ref |
               operation_ref
index        = "<index>" ("(" value ")")? ("."typedelemenent_ref)?
qualifier    = "<qualifier>" ("(" value ")")? ("."typedelemenent_ref)?
create       = "<create>(" parameters? ")" ("."operation_ref)?
statemachine = "<statemachine>" statemachine_ref
context      = "<context>" classifier_ref | typedelemenent_ref |
               operation_ref
attr_value   = "<value>" ("." classifier_ref)?
op_value     = "<value>(" values ")" ("." classifier_ref)?
values       = value (","value)*
value        = literal | path
parameters   = name ":" type_name (", " parameters)?
state        = "<state>" ("." classifier_ref)?

# Java literals
# classname: Name of a Java class.
# java_constant: Is a static field of a java class.
# java_literal: Is a string description, used as parameter
# for the constructor of the required class.
literal             = constructor_literal | constant_literal
constructor_literal = classname"(" ( lit_parameter(","lit_parameter)*)? ")"
constant_literal    = java_constant | java_literal | "null"
lit_parameter       = (classname":")? literal
```

Abbildung A.1: VEPL-Spezifikation in EBNF

Abbildungsverzeichnis

2.1 Seeheim Modell für UIMS-Architekturen (nach [Green 1985]) . . 8
2.2 Beziehungen zwischen Model, View und Controller Objekten . . 11
2.3 PAC-Architektur (aus [Kazman Bass 1996]) 13
2.4 Arch Modell für UIMS-Architekturen (aus [Encarnação 1997]) . . 14
2.5 Ableitungen aus dem Slinky Metamodell (aus [Bass et al. 1992]) 16
2.6 PAC-Amodeus (aus [Calvary et al. 1997]) 18
2.7 Visual Proxy Architektur (aus [Holub 1999]) 20
2.8 MVC layers (aus [Cai et al. 2000]) 22
2.9 Entwicklung von User Interface Artefakten [Kazman Bass 1996] . 23
2.10 Klassifikation von Techniken (nach [Fähnrich 1995]) 26
2.11 XUL-Komponenten (aus [Oeschger 2000a]) 29
2.12 UIML-Dokumentstruktur 30
2.13 Klassifikation von Werkzeugen (aus [Fähnrich 1995]) 40
2.14 User Interface Laufzeitarchitekturen (aus [da Silva 2000]) 46

3.1 MVC-Client Architektur 57
3.2 SanFrancisco Architektur 58
3.3 Beispielhafter Dialogaufbau nach SanFrancisco 59
3.4 JWAM GUI Architektur 60
3.5 MVP-Architektur (aus [Potel 1996]) 62
3.6 MVP-Verteilungsmöglichkeiten 63
3.7 Das Janus Entwicklungskonzept (aus [Balzert et al. 1995]) 68
3.8 Generierungsprozeß im Janus-System (aus [Kruschinski 1999]) . . 69
3.9 Das Janus-Application-Framework (aus [Balzert et al. 1995]) . . 70
3.10 Mobi-D Architektur (aus [Puerta 1997]) 71
3.11 Mobi-D Entwicklungszyklus (aus [Puerta 1997]) 73
3.12 FUSE-Systemarchitektur 74
3.13 TRIDENT-Komponenten (aus [Bodart et al. 1993]) 76
3.14 TRIDENT Methodologie (nach [Bodart et al. 1996]) 77
3.15 Der TADEUS-Ansatz (aus [Schlungbaum Elwert 1995]) 78
3.16 Teallach Architektur (aus [Griffiths et al. 1998b]) 80
3.17 MASTERMIND Designumgebung (aus [Szekely et al. 1995]) . . . 83
3.18 Metamodell des BC-Prototyper (aus [van Emde Boas 2000]) . . . 86
3.19 Modellbasierte Ansätze - Überblick 88
3.20 Subsumierte UbiComp-Eigenschaften der Systemgruppen 91

4.1 Vesuf Architektur 95
4.2 Laufzeitarchitektur 97
4.3 Beispielmodell (Shape) 99

4.4 ApplicationDescriptor des Shape-Beispiels 101
4.5 Ausschnitt der VEPL Spezifikation in EBNF 102
4.6 Methoden des EventDispatcherProxies 107
4.7 Anwendung eines EventDispatcherProxies 107
4.8 Eventgenerierung bei Dependencies 114
4.9 Eventgenerierung für benutzerdefinierte Pfade 115
4.10 Das ImplementationAccessor Konzept 116
4.11 PermissionGranter-Konzept 117
4.12 Standarddialogmodell 120
4.13 Präsentationsmetamodell als UML Klassendiagramm 121
4.14 Struktur der Präsentationselemente zur Laufzeit 122
4.15 Sichere Methode 123
4.16 Konkretes Präsentationsmodell des Shape-Beispiels 125

5.1 PublicationPORTAL Architektur (aus [Zirpins et al. 2001]) . . . 132
5.2 Domänenmodell des Wörterbuch Dienstes 136
5.3 Von dict.leo.org generiertes HTML-Ergebnis 140
5.4 `parse()` Methode des LeoParsers 141
5.5 Screenshot des Meta-Dictionaries (Java-AWT) 142
5.6 Spezifizierung eines Listenelementes 143
5.7 Screenshot des Meta-Dictionaries (HTML-Portlet) 144
5.8 Spezifikation des HTML Präsentationsmodells 145
5.9 HTML Template (Ausschnitt) 145
5.10 Screenshot des Meta-Dictionaries (WML) 146
5.11 WML-Template (Ausschnitt) 147
5.12 Screenshot des Meta-Dictionaries (VXML) 148
5.13 VXML-Template (Ausschnitt) 149
5.14 Appliationsdeskriptoren des Metawörterbuchdienstes 149
5.15 Domänenmodell des Z39.50 Dienstes 152
5.16 Information Retrieval Modell des JZKit Frameworks 155
5.17 Connect-Methode des ConnectTasks 156
5.18 BuildQuery-Methode des SearchingTasks 157
5.19 Typ-1 Query Konstruktion 158
5.20 Retrieve-Methode des PresentTasks 159
5.21 Dialogmodell des Z39.50 Dienstes 160
5.22 Snapshots der Z39.50 AWT Anwendung 162
5.23 Navigationselement im Connect-Dialog 163
5.24 Subquery-Liste des Searching-Dialogs 164
5.25 Detail-Knopf eines Ergebnisses im Present-Dialog 164
5.26 Snapshots der Z39.50 HTML Anwendung 165
5.27 Z39.50 Präsentationsmodell (HTML-Version) 166
5.28 HTML-Template des PresentDialogs (Ausschnitt) 166
5.29 Applikationsdeskriptoren für AWT/HTML Version 167
5.30 Mapping properties des Z39.50 Dienstes 167

A.1 VEPL-Spezifikation in EBNF 174

Literaturverzeichnis

[Abrams 2000] Marc Abrams. *Proposal for UIML2 Shorthand.* UIML.org. 2000

[Anderson 1999] D. Anderson. *Server-side MVC Architecture, Part 1-3.* UIDesign.net Whitepaper, October 1999. `http://www.uidesign.net/1999/papers/webmvc_part1.html`

[Anderson 2000a] D. Anderson. *Extending UML for UI - A position paper for the TUPIS2000 workshop at UML2000.* UIDesign.net Whitepaper, September 2000. `http://www.uidesign.net/2000/papers/TUPISproposal.html`

[Anderson 2000b] D. Anderson. *TUPIS2000 - Notes on Interaction Spaces from UML2000.* UIDesign.net Conference Report, Dezember 2000. `http://www.uidesign.net/2000/conference/TUPISreport.html`

[Annett Duncan 1967] J. Annett, K. Duncan. Task analysis and training in design. Occupational Psychology 41, pp. 211-221.

[ANSI/NISO 1995] ANSI/NISO. *Information Retrieval (Z39.50): Application Service Definition and Protocol Specification.* 1995. `http://www.niso.org/z3950.html`

[ArgoUML.org 2000] *ArgoUML - Object-oriented design tools with cognitive support.* ArgoUML.org, 2000. `http://www.argouml.org`

[Artim 1997] J. M. Artim. *Integrating User Interface Design And Object-Oriented Development Through Task Analysis And Use Cases.* User Centered Design Group, OOCL Inc. 1997. `http://www.cutsys.com/CHI97/Artim.html`

[ASF 2001a] Apache Software Foundation. *ECS.* 2001. `http://jakarta.apache.org/jetspeed`

[ASF 2001b] Apache Software Foundation. *Jetspeed.* 2001. `http://jakarta.apache.org/jetspeed`

[ASF 2001c] Apache Software Foundation. *turbine.* 2001. `http://jakarta.apache.org/turbine`

[Avedik 2001] Thomas Avedik. *Mobile banking - a evolution race for ownership.* MRE: The Mobile Commerce Project, Montgomery Research Europe Ltd. 2001
http://www.mcommcentral.com/documents.asp?d_ID=404

[Balzert 1995] H. Balzert. *From OOA to GUI - The Janus System.* In Proceedings of the 5th Conference on Human-Computer Interaction (INTERACT'95), Chapman & Hall, 1995.
http://www.swt.ruhr-uni-bochum.de/forschung/veroeffent/index.

[Balzert et al. 1995] H. Balzert, F. Hofmann, C. Niemann. *Vom Programmieren zum Generieren. Auf dem Weg zur automatisierten Anwendungsentwicklung.* Paper der Ruhr-Universität Bochum. 1995.
http://www.swt.ruhr-uni-bochum.de/forschung/veroeffent/index.

[Banavar et al. 2000] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, D. Zukowski. *Challenges: An application Model for Pervasive Computing.* ACM Press New York, NY, USA. 2001.

[Bartelt et al. 2001] A. Bartelt, C. Zirpins, D. Fahrenholtz. *Geschäftsmodelle der Electronic Information: Modellbildung und Klassifikation.* In Informatik 2001 - Wirtschaft und Wissenschaft in der Network Economy. Österreichische Computergesellschaft (OCG), pp. 902-908, 9/2001.
http://vsys-www.informatik.uni-hamburg.de/publications/viewpu

[Bass et al. 1992] L. Bass, R. Franeuf, R. Little, N. Mayer, B. Pellegrino, S. Reed, R. Seacord, S. Sheppard and M. Szczur. *A Metamodel for the Runtime Architecture of an Interactive System.* The UIMS Tool Developers Workshop, SIGCHI Bulletin, 24(1), pp. 32-37, January 1992.

[Baudel Lafon 1998] Thomas Baudel, Michel Beaudouin-Lafon. *Outils et Méthodes de Construction d'Interfaces.* Tutoriel IHM'98, Groupe Interaction Homme-Machine, Laboratiore de Recherche en Informatique, Université Paris-Sud, 1998.
http://www-ihm.lri.fr/~thomas/Docs/Toolkits/CourArchi/cours.h

[Bauer 1996] B. Bauer. *Generating User Interfaces from Formal Specifications of the Application.* DSV-IS'96 Workshop. 1996.
http://www.isi.edu/isd/Mastermind/Papers/DSVIS96.ps

[Birnbaum et al. 1997] L. Birnbaum, R. Barreiss, T. Hinrichs, C. Johnson. *Model-Based Human-Computer Interaction.* Northwestern University, The Institute for the Learning Sciences, 1997.

[Bleek et al. 1999a] W.-G. Bleek, G. Gryczan, C. Lilienthal, M. Lippert, S. Roock, H. Wolf, H. Züllinghoven. *Frameworkbasierte Anwendungsentwicklung (Teil 2): Die Konstruktion interaktiver Anwendungen.* In OBJEKTspektrum 2/99, pp. 78-83.

[Bleek et al. 1999b] W.-G. Bleek, M. Lippert, S. Roock, W. Strunk, H. Züllinghoven. *Frameworkbasierte Anwendungsentwicklung (Teil 3): Die Anbindung von Benutzungsoberflächen und Entwicklungsumgebungen an Frameworks.* In OBJEKTspektrum 3/99, pp. 90-95.

[Bluetooth SIG 1999] Bluetooth SIG. Specification of the Bluetooth system, Dec. 1999. `http://www.bluetooth.com`

[Bodart et al. 1993] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Sacre, J. Vanderdonckt. *Architecture Elements for Highly-Interactive Business-Oriented Applications.* L. Bass, J. Gornostaev and C. Unger (Eds.), Lecture Notes in Computer Science, Vol. 753, Springer-Verlag, Berlin, 1993, pp. 83-104. `http://www.qant.ucl.ac.be/membres/jv/publi/Publications.html`

[Bodart et al. 1994] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, J. Vanderdonckt. *A Model-Based Approach to Presentation: A Continuum from Task Analysis to Prototype.* Proc. of the Eurographics Workshop "Design, Specification and Verification of Interactive Systems." 1994. `http://www.qant.ucl.ac.be/membres/jv/publi/Publications.html`

[Bodart et al. 1996] F. Bodart, A.-M. Hennebert, J.-M. Leheureux, I. Provot, J. Vanderdonckt, G. Zucchinetti. *Key Activities for a Development Methodology of Interactive Applications.* Chapter 4 in Critical Issues in User Interface System Engineering, D Benyon and P. Palanque (Eds.), SpringerVerlag, 1995. `http://www.qant.ucl.ac.be/membres/jv/publi/Publications.html`

[Bos et al. 1998] B. Bos, H. W. Lie, C. Lilley, I. Jacobs. Cascading Style Sheets, level 2, CSS2 Specification. W3C Recommendation 12.05.1998. `http://www.w3.org/TR/REC-CSS2/`

[Bray et al. 2000] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation. World Wide Web Consortium (W3C). 2000. `http://www.w3.org/TR/REC-xml`

[Browne et al. 1997] T. Browne, D. Davila, S. Rugaber, K. Stirewalt. *Using Declarative Descriptions to Model User*

Interfaces with MASTERMIND. Graphics, Visualization, and Usability Center, Georgia Institute of Technology. 1997.
http://citeseer.nj.nec.com/browne97using.html

[Buck 2000] L. Buck. *Modeling Relational Data in XML.* White Paper.
http://apps.xmlschema.com/white_papers/modeling.htm

[Burbeck 1992] S. Burbeck. *Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller(MVC).* 1992.
http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[Cai et al. 2000] J. Cai, R. Kapila und G. Pal. *HMVC: The layered pattern for developing strong cient tiers.* JavaWorld, July 2000.
http://www.javaworld.com/javaworld/jw-07-2000/jw-0721-hmvc.ht

[Calvary et al. 1997] G. Calvary, J. Coutaz und L. Nigay. *From Single-User Architectural Design to PAC*: a Generic Software Architecture Model for CSCW.* CHI 97 Papers, Atlanta 1997.
http://iihm.imag.fr/publs/1997/

[Card et al. 1983] S. Card, T. Moran, A. Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum Associates. 1983.

[Cattell 1997] R. Cattell. 1997. *The Object Database Standard: 2.0.* Morgan Kaufmann Publishers, Inc. 1997.

[de Champeaux 1993] D. de Champeaux 1993, D. Lea, P. Faure. *Object-Oriented System Development.* Addison-Wesley, 1993.

[Chen 1976] P. P. Chen. *The Entity-Relationship Model - Towards a Unified View of Data.* ACM Trans. on Database Systems, 1(1), pp. 9-36. 1976.

[Coad Yourdon 1991a] P. Coad, E. Yourdon. *Object Oriented Analysis.* Yourdon Press. Englewood Cliffs. New Jersey. 1991.

[Coad Yourdon 1991b] P. Coad, E. Yourdon. *Object Oriented Design.* Yourdon Press. Englewood Cliffs. New Jersey. 1991.

[Coutaz 1987] Joëlle Coutaz. *PAC, an Object Oriented Model for Dialog Design.* Human-Computer Interaction - INTERACT'87, H.-J. Bullinger and B. Shackel (Eds.), Elsevier Science Publishers B.V. (North-Holland), 1987. pp. 431-436.

[Coutaz et al. 1995] J. Coutaz, L. Nigay, D. Salber. *Agent-Based Architecture Modelling for Interactive Systems.* Critical Issues in User Interface Engineering. P. Palanque, D. Benyon (Eds.), Springer-Verlag: London. 1995. pp. 191-209. http://fermivista.math.jussieu.fr/ftp/ftp.imag.fr.html

[Davidson Coward 1999] James Duncan Davidson, Danny Coward. *Java(tm) Servlet Specification, v2.2.* Sun Microsystems. 1999. http://java.sun.com/products/servlet/download.html

[Deakin 2001] N. Deakin. *XUL Tutorial.* XUL Planet. 2001. http://www.xulplanet.com/

[Denert 1991] E. Denert, J. Siedersleben. Softwareengineering: methodische Projektabwicklung. Springer Verlag. 1991.

[Duce et al. 1991] D. A. Duce, M. R. Gomes, F. R. A. Hopgood and J. R. Lee (Eds.). *User Interface Management and Design.* Proceedings of the Workshop on UIMS, Lissabon, Portugal, 1991, Springer Verlag Berlin, 1991

[Eisenstein Puerta 2000] Jacob Eisenstein, Angel Puerta. *Adaptation in Automated User-Interface Design.* In H. Lieberman (Ed.), IUI2000: International Conference on Intelligent User Interfaces, ACM, New York, 2000. http://lieber.www.media.mit.edu/people/lieber/IUI/Eisenstein/Eisenstein

[Elwert 1996] T. Elwert. *Continuous and Explicit Dialogue Modelling.* SIGCHI'96 Proceedings. 1996. http://www.acm.org/sigchi/chi96/proceedings/shortpap.htm

[van Emde Boas 2000] H. van Emde Boas-Lubsen. *Business Component Prototyper for SanFrancisco: An experiment in architecture for application development tools.* In IBM Systems Journal, Vol 39, No 2, 2000. http://www.research.ibm.com/journal/sj/392/vanemdeboas.html

[Encarnação 1997] L. Miguel Encarnação. *Concept and realization of intelligent user support in interactive graphics applications.* Dissertation der Fakultät für Informatik der Eberhard-Karls-Universität zu Tübingen. 1997.

[Fähnrich 1995] K.-P. Fähnrich. *Methoden und Werkzeuge zur softwareergonomischen Entwicklung von Informationssystemen.Heimsheim: Jost-Jetter Verlag, 2000.* http://www.service-engineering.net/vorlesungen/software_ergonomie/theme

[Faensen et al. 1999] D. Faensen, H. Schweppe, L. Faulstich, A. Hinze A. Steidinger. *Project Hermes: Alerting Services for Digital Libraries.* Projektbeschreibung. http://www.inf.fu-berlin.de/~ag-db/projects/project_hermes.html

[Fielding et al. 1999] R.Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach , T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* The Internet Engineering Task Force (IETF), 1999.
http://www.ietf.org/rfc/rfc2616.txt

[Firesmith et al. 1998] D. Firesmith, B. Hendersson-Sellers, I. Graham. *OPEN Modeling Language (OML) reference manual.* Cambridge University Press, 1998.

[Foley et al. 1991] J. Foley, W.Kim, S. Kovacevic, K. Murray. *UIDE - An Intelligent User Interface Design Environment.* In Intelligent User Interfaces, pp. 339-384. Addison-Wesley, ACM Press, 1991.

[Gallis et al. 2001] H. Gallis, J. Petter, J. Herstad. *The multidevice paradigm in Knowmobile - Does one size fit all?* Department of Informatics, University of Oslo. 2001.
http://www.stud.ifi.uio.no/~jarlek/knowmobile/articles/multide

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gibbs 1994] W. Wayt Gibbs. *Software's Chronic Crisis.* Trends in Computing, Scientific American, September 1994.

[Global Info 2000] Global Info. *Globale Elektronische und Multimediale Informationssysteme für Naturwissenschaft und Technik des bmb+f.* Bundesministerium für Bildung und Forschung, 2000.
http://www.global-info.org/

[Gosling et al. 1996] J. Gosling, B. Joy, G. Steele. *The Java Language Specification.* Addison Wesley Developers Press, Sunsoft Java Series, 1996.
http://java.sun.com/docs/books/jls/

[Gray et al. 1998] P. Gray, R. Cooper, J. Kennedy, P. Barclay, T. Griffiths. *A Lightweight Presentation Model for Database User Interfaces.* University of Manchester, 1998.
http://ui4all.ics.forth.gr/UI4ALL-98/proceedings.html

[Green 1985] M. Green. *Report on Dialogue Specification Tools.* in Günther E. Pfaff (Ed.) *User Interface Management Systems.* Springer Verlag, 1985.

[Griffel et al. 2001] F. Griffel, C. Zirpins and S. Müller-Wilken. *Generative Softwarekonstruktion auf Basis typisierter Komponenten.* In U. Killat und W. Lamersdorf

(Eds.), Proceedings: Kommunikation in Verteilten Systemen (KiVS), Springer-Verlag, Berlin, pp. 325-338, 2001.
http://vsys-www.informatik.uni-hamburg.de/publications/

[Griffiths et al 1998a] T. Griffiths, J. McKirdy, G. Forrester, N. Paton, J. Kennedy, P. Barclay, R. Cooper, C. Goble, P. Gray. *Exploiting model-based techniques for user interface to databases.* University of Manchester. In Proceedings of VDB-4, Italy, May 1998.
http://img.cs.man.ac.uk/publications.htm

[Griffiths et al. 1998b] T. Griffiths, J. McKirdy, N. Paton, J. Kennedy, R. Cooper, B. Barclay, C. Goble, P. Gray, M. Smyth, A. West, A. Dinn. *An Open Model-Based Interface Development System: The Teallach Approach.* In Proc. Eurographics Workshop DSV-IS'98, pp. 32-49, 1998.
http://img.cs.man.ac.uk/publications.htm

[Grimm et al. 2001] Robert Grimm, Janet Davis, Ben Hendrickson, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, David Wetherall. *Systems Directions for Pervasive Computing.* University of Washington, 2001
http://www.cs.washington.edu/homes/gribble/pubs.html

[Gupta et al. 1998] Suchitra Gupta, Jeff Hartkopf, Suresh Ramaswamy. *Event Notifier, a Pattern for Event Notification.* Java Report, July 1998, Volume 3, Number 7, SIGS Publications.
http://www.users.qwest.net/~hartkopf/notifier

[Häming 2000] A. Häming. *Konzeption und Realisierung einer Typmanagement-Komponente zur Unterstützung generativer Softwarekonstruktion.* Universität Hamburg, Fachbereich Informatik, Arbeitsgruppe Verteilte Systeme, Diplomarbeit. 2000.

[ten Hagen 1991] P. J. W. ten Hagen. *Critique of the Seeheim Model.* In [Duce et al. 1991], pp. 3-6.

[Hamilton 1997] G. Hamilton. *The Java Beans 1.01 Specification.* Sun Microsystems Inc. 1997.
http://splash.javasoft.com/beans/docs/spec.html

[Harel 1987] D. Harel. *Statecharts: A visual Formalism for Complex Systems.* Scientific Computer Programs 8, pp. 231-274, 1987.

[Hebbel 1997] F. Hebbel. *Using Object Modeling CASE Tools.* DBMS Online, Volume 10 Number 8, July 1997.
http://www.dbmsmag.com/9707d00.html

[Hejda 2000] Petr Hejda. *Architectural Model for User Interfaces of Web-based Applications.* In P. L. Emiliani und C. Stephanidis (Eds.), Proceedings of the 6th ERCIM Workshop *User Interfaces for All*, Florenz, Italien, 2000.
`http://ui4all.ics.forth.gr/UI4ALL-2000/proceedings.html`

[Hitz Kappel 1999] M. Hitz, G. Kappel. *UML @ Work.* dpunkt.verlag, 1999.

[Holub 1999] A. Holub. *Building user interfaces for object-oriented systems, Part 2: The visual-proxy architecture.* Java Toolbox, JavaWorld September 1999.
`http://www.javaworld.com/javaworld/jw-09-1999/jw-09-toolbox.h`

[le Hors et al. 2000] Arnaud le Hors, Philippe le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, Steve Byrne. *Document Object Model (DOM) Level 2 Core Specification.* W3C Recommendation, World Wide Web Consortium (W3C). 2000.
`http://www.w3.org/TR/DOM-Level-2-Core/`

[Huang et al. 1999] Andrew C. Huang, Benjamin C. Ling, Shankar Ponnekanti, Armando Fox. *Pervasive Computing: What Is It Good For?* Stanford University, 1999.
`http://citeseer.nj.nec.com/huang99pervasive.html`

[Hudson 1987] Scott. E. Hudson. *UIMS Support for Direct Manipulation Interfaces.* ACM Computer Graphics, Volume 21 No. 2, pp. 120-124. April 1987.

[Hudson King 1988] S. E. Hudson, R. King. *Semantic feedback in the Higgens UIMS.* IEEE Transactions on Software Engineering, 14(8):1188-1206, August 1988.

[Hussey Carrington 1995] Andrew Hussey and David Carrington. *Comparing two user-interface architectures: MVC and PAC.* Technical Report No. 95-33, Software Verification Research Centre, Dept. of Computer Science, University of Queensland, Australia. December 1995.
`ftp://ftp.cs.uq.edu.au/pub/SVRC/techreports/`

[Hussmann et al. 2000] H. Hussmann, B. Demuth, F. Finger. *Modular Architecture for Toolset Supporting OCL.* Dresden University of Technology, Department of Computer Science. In Proceedings of UML 2000 Conference in York, UK.
`http://www.inf.tu-dresden.de/TU/Informatik/ST2/ST/papers`

[Ibbotson 2001] I. Ibbotsson. *JZKit: Project Goals.*
`http://www.k-int.com/products/jzkit/goals.php`

[Ibbotson et al. 2001] I. Ibbotson, M. Neale. *Opensource JZKit Z39.50 Framework.* `http://sourceforge.net/projects/jzjkit/`

[ISO 1977] ISO 7498. *ISO OSI Basic Reference Model.* 1977.

[ISO 1988] ISO. *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on Temporal Ordering of Observational Behaviour.* ISO/IS 8807.

[ISO/IEC 1996] ISO/IEC 14977: 1996(E). Extended Backus Naur Form specification. `http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf`

[Jacob 1985] Robert J. K. Jacob. *A State Transition Diagram Language for Visual Programming.* IEEE Computer 18(8). pp. 51-59, August 1985.

[Jacob 1986] Robert J. K. Jacob. *A Specification Language for Direct-Manipulation User Interfaces.* ACM Transactions on Graphics 5(4). pp. 283-317 Oktober 1986

[Jacobson et al. 1992] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley. 1992.

[Janssen et al. 1993] C. Janssen, A. Weisbecker, J. Ziegler. *Generating User Interfaces from Data Models and Dialogue Net Specifications.* In Proceedings of INTERCHI'93, pp. 418-423, ACM Press.

[Jessen Valk 1987] E. Jessen, R. Valk. *Rechensysteme: Grundlagen der Modellbildung.* Springer Verlag. 1987.

[Jorgensen 1994] Steven A. Jorgensen. *An Object-Oriented Approach to Tool Integration in an Integrated CASE Environment.* Master Thesis, Electrical and Computer Engineering, University of New Mexico, July 1994. `http://www.khoral.com/staff/steve/home.html`

[Kazman Bass 1996] Rick Kazman, Len Bass. *Software Architectures for Human-Computer Interaction: Analysis and Construction.* submitted to ACM Transactions on Human-Computer Interaction. 1996. `http://citeseer.nj.nec.com/36927.html`

[Kent et al. 1999] S. Kent, A. Evans, P. Rumpe. UML Semantics FAQ. ECOOP'99. ???

[Kim Foley 1990] W. Kim, J. Foley. *DON: User Interface Presentation Design Assistant.* In Proceedings of UIST'90, pp. 10-20, ACM Press, 1990.

[Kovacevic 1993] S. Kovacevic. *TACTICS for User Interface Design: Coupling the Compositional and Transformational Approach.* Techincal Report, US West Advanced Technologies, 1993.
`http://www.gvu.gatech.edu/gvu/reports/1993/`

[Krasner Pope 1988] G. E. Krasner und S. T. Pope. *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System.*
`http://citeseer.nj.nec.com/krasner88description.html`

[Kruglinski 1996] D. Kruglinski. *Inside Visual C++.* Microsoft Press, 1996.

[Kruschinski 1999] V. Kruschinski. *Layoutgestaltung grafischer Benutzungsoberflächen. Generierung aus OOA-Modellen.* Spektrum Akademischer Verlag, Berlin 1999.

[Lantz et al. 1987] K. A. Lantz, P. P. Tanner, C. Binding, K. T. Huang, A. Dwelly. *Reference models, window systems and concurrency.* Computer Graphics, Vol. 21, No. 2. 1987. pp. 87-97.

[Leung et al. 1999] K. Leung, L. Hui, S.M. Yiu, R. Tang. *Modelling Web Navigation by Statechart.* Dept. of Computing and Mathematics, Hong Kong Institute of Vocational Education, 1999.
`http://citeseer.nj.nec.com/291119.html`

[Lippert 1997] M. Lippert. *Konzeption und Realisierung eines GUI-Frameworks in Java nach der WAM-Metapher. Studienarbeit.* Arbeitsbereich Softwaretechnik. Fachbereich Informatik. Universität Hamburg. 1997. German.
`http://www.jwam.de/product/literature.html`

[Lonczewski et al. 1996] F. Lonczewski, S. Schreiber. *Generating User Interfaces with the FUSE-System.* Technische Universität München, 1996.
`http://wwwbib.informatik.tu-muenchen.de/infberichte/1996/`

[Luo et al. 1993] P. Luo, P. Szekely, R. Neches. *Management of Interface Design in HUMANOID.* In Proceedings of INTERCHI'93.
`http://www.isi.edu/isd/HUMANOID/humanoid-papers.html`

[Machiraju 1996] V. Machiraju. *A Survey on Research in Graphical User Interfaces.* 1996.
`http://citeseer.nj.nec.com/machiraju96survey.html`

[Märtin 1996] C. Märtin. *Software Life Cycle Automation for Interactive Applications: The AME Design Environment.* In CADUI'96 FUNDP Namur. Namur University Press.

[Markopoulos et al. 1992] P. Markopoulos, J. Pycock, S. Wilson, P. Johnson. *Adept - A task based design environment.* In Proceedings of the 25th Hawaii International Conference on System Sciences, pp. 587-596. IEEE Computer Society Press, 1992.

[Mitchell et al. 1995] K. Mitchell, J. Kennedy, P. Barclay. *Using a Conceptual Data Language to Describe a Database and its Interface.* In British National Conference on Databases 13, Manchester, England, pp. 101-119.

[Microsoft 1995] Microsoft. The Windows Interface Guidelines for Software Design. Microsoft Press, Redmond, 1995.

[Miller 1999] P. Miller. *Z39.50 for All.* Ariadne Issue 21. 9/2001. http://www.ariadne.ac.uk/issue21/z3950/

[Mozilla 1999a] Mozilla org. *XUL Language Spec.* Mozilla Paper. http://www.xulplanet.com

[Mozilla 1999b] Mozilla org. *XUL Introduction to a XUL Document.* Mozilla Paper. http://www.xulplanet.com

[Mueller et al. 2000] A. Mueller, T. Mundt, W. Lindner. *Using XML to Semi-Automatically Derive User Interfaces.* Dept. of Computer Science, University of Rostock. http://wwwtec.informatik.uni-rostock.de/IuK/publications/tagungen.html

[Myers 1989] B. A. Myers. *User-Interface Tools: Introduction and Survey.* IEEE Software, Volume 6 No. 1, January 1989, pp. 15-23. http://www-2.cs.cmu.edu/afs/cs/project/garnet/www/papers.html

[Myers Rosson 1992] B. A. Myers. *Survey on User Interface Programming.* In Proceedings SIGCHI'92. http://www-2.cs.cmu.edu/afs/cs/project/garnet/www/papers.html

[Myers 1993] B. A. Myers. *Why are Human-Computer Interfaces Difficult to Design and Implement?* Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-93-183. July 1993. http://www-2.cs.cmu.edu/afs/cs/project/garnet/www/papers.html

[Myers 1995] B. A. Myers. *User Interface Software Tools.* ACM Transactions on Computer-Human Interaction. vol. 2, no. 1, March, 1995. pp. 64-103. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/bam/www/toolnames.html

[Myers et al. 1999] B. A. Myers, S. E. Hudson und R. Pausch. *Past, Present and Future of User Interface Software Tools.* Human Computer Interaction Institute, Carnegie Mellon University, Pittsburgh. Draft of 09/16/1999, to appear in ACM TOCHI. http://www.cs.cmu.edu/afs/cs.cmu.edu/user/bam/www/resume.html

[Netmation 2000] Netmation. *CASE Tools.* Netmation Inc. 2001.
`http://netmation.com/docs/bb17.htm`

[Netscape 1999] Netscape. *Client-Side JavaScript Reference.* Netscape Corp. 1999.
`http://developer.netscape.com/docs/manuals/javascript.html`

[Newman 1968] W. Newman. *A System for Interactive Graphical Programming.* In AFIPS Conference Proceedings, Spring Joint Computer Conference, Washington, DC. Thompson Books, 1968. pp. 47-54.

[Oberquelle 1987] H. Oberquelle. Sprachkonzepte für benutzergerechte Systeme. Springer Verlag. 1987.

[Oeschger 2000a] I. Oeschger. *XUL Genealogy: What Does XUL Have To Do With XML?.* Mozilla Paper.
`http://www.mozilla.org/docs/xul/xulnotes/index.html`

[Oeschger 2000b] I. Oeschger. *XUL Notes: A XUL Bestiary.* Mozilla Paper.
`http://www.mozilla.org/docs/xul/xulnotes/index.html`

[Olsen 1992] D. Olsen. *User Interface Management Systems: Models And Algorithms.* Morgan Kaufmann Publishers, San Mateo, California. 1992.

[OMG 2000a] Object Management Group. Unified Modelling Language (UML), version 1.3.
`http://www.omg.org/technology/documents/formal/uml.htm`

[OMG 2000b] Object Management Group. *CORBA: Architecture and Specification.*
`http://www.omg.org/technology/documents/formal/corba_iiop.htm`

[OMG 2001] Object Management Group. *Meta Object Facility (MOF) Specification.*
`http://www.omg.org/technology/documents/formal/mof.htm`

[Palanque Bastide 1995] P. Palanque, R. Bastide. *Design, Specification and Verification of Interactive Systems '95.* Springer Verlag.

[Paternò 1999] F. Paternò. *Model-Based Design and Evaluation of Interactive Applications.* Springer Verlag. 1999.

[Paternò 2000] F. Paternò. *ConcurTaskTrees and UML: how to marry them?* Not published.
`http://giove.cnuce.cnr.it/Guitare/Document/ConcurTaskTrees_an`

[Patry Girard 1999] G. Patry, P. Girard. *GIPSE, a Model Based System for CAD Software.* Third Conference on Computer-Aided Design of User Interfaces (CADUI'99), Louvain-la-Neuve, Belgique, October 1999, pp. 61-72.

[Potel 1996] M. Potel. *MVP: Model-View-Presenter. The Taligent Programming Model for C++ and Java.* http://www-106.ibm.com/developerworks/library/mvp.html

[Puerta 1990] A. R. Puerta. *L-CID: A Blackboard Framework to Experiment with Self-Adaption in Intelligent Interfaces.* Dissertation, USCMI Report-Nummer 90-esl-6, Center for Machine Intelligence, University of South Carolina, Columbia, 1990.

[Puerta 1996] A. R. Puerta. *The Mecano Project: Comprehensive and Integrated Support for Model-Based Interface Development.* Computer-Aided Design of User Interfaces, ed. by J. Vanderdonckt. Presses Universitaires de Namur, Namur Belgium, 1996. http://smi-web.stanford.edu/projects/mecano/publicat.htm

[Puerta 1997] A. R. Puerta. *A Model-Based Interface Development Environment.* Stanford University. http://smi-web.stanford.edu/projects/mecano/publicat.htm

[Puerta 1998] A. R. Puerta. *Supporting User-Centered Design of Adaptive User Interfaces Via Interface Models.* First Annual Workshop On Real-Time Intelligent User Interfaces For Decision Support And Information Visualization, San Francisco, 1998. http://smi-web.stanford.edu/projects/mecano/publicat.htm

[Puerta Eisenstein 1999] A. R. Puerta, J. Eisenstein. *Towards a General Computional Framework for Model-Based Interface Development Systems.* Proceedings IUI'99, Los Angeles, CA, 1999. http://smi-web.stanford.edu/projects/mecano/publicat.htm

[Raggett et al. 1999] Dave Raggett, Arnaud Le Hors, Ian Jacobs. *HTML 4.01 Specification.* W3C Recommendation. World Wide Web Consortium (W3C). 1999. http://www.w3.org/TR/html4/

[Rational 2000] Rational Software Corporation. *Rational Unified Process: Best Practices for Software Development Teams.* Rational Software Corporation, 2000. http://www.rational.com/products/whitepapers/100420.jsp

[Rochkind 1992] M. J. Rochkind. *An Extensible Virtual Toolkit (XVT) for Portable GUI Applications.* pp. 485-494. Digest of Papers, COMPCON (Spring 1992). San Francisco, CA: Thirty-Seventh IEEE Computer Society International Conference, February 1992.

[Rugaber 1998] S. Rugaber. *MASTERMIND Project Final Report.* Graphics, Visualization, and Usability Center, Georgia Institute of Technology. http://www.cc.gatech.edu/gvu/user_interfaces/Mastermind/final.html

[Rumbaugh et al. 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Sanderson 1999] R. Sanderson. *MVC-Client: Putting Model-View-Controller to work.*
http://www.fourbit.com/products/fab/papers/fab_mvc_clients.htm

[Schallehn et al. 2000] E. Schallehn, M. Endig. *Föderierungsdienste für heterogene Dokumentenquellen.* Projektbeschreibung.
http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/globalinfo/

[Schlungbaum Elwert 1995] E. Schlungbaum, T. Elwert. *Modelling and Generating of Graphical User Interfaces in the TADEUS Approach.* Universität Rostock. In [Palanque Bastide 1995].

[Schlungbaum 1997] E. Schlungbaum. *(Knowledge-based) Support of Task-based User Interface Design in TADEUS.* Universität Rostock.

[Schreiber 1994] S. Schreiber. *Specification and Generation of User Interfaces with the BOSS-System.* Universität München. 1994.
http://www2.informatik.tu-muenchen.de/pub/papers/sis/

[da Silva 2000] P. P. da Silva. *User Interface Declarative Models and Development Environments: A Survey.* Department of Computer Science, University of Manchester. 2000.
http://img.cs.man.ac.uk/umli/publications.html

[da Silva et al. 2000] P. P. da Silva, T. Griffiths, N. W. Paton. *Generating User Interface Code in a Model Based User Interface Development Environment.* In Proceedings of the International Conference on Advanced Visual Interfaces (AVI2000), Palermo, Italy, 2000.
http://img.cs.man.ac.uk/umli/publications.html

[da Silva Paton 2000a] P. P. da Silva, W. Paton. *User Interface Modelling with UML.* In Information Modelling and Knowledge Bases XII (10th European-Japanese Conference on Information Modellind and Knowledge Representation, Saariselka, Finland, May 2000). Pages 203-217, IOS Press, Amsterdam, 2001.
http://img.cs.man.ac.uk/umli/publications.html

[da Silva Paton 2000b] P. P. da Silva, W. Paton. *UMLi: The Unified Modeling Language for Interactive Applications.* UML 2000 - The Unified Modeling Language.

Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings
http://img.cs.man.ac.uk/umli/publications.html

[Stoiber 1999] S. Stoiber. *UML-Support for Model- and Task-Based Development of Interactive Software Systems.* Universität Linz. 1999.
http://www.ce.uni-linz.ac.at

[Szekely 1990] P. Szekely. *Template - Based Mapping of Application Data to Interactive Displays.* In Proceedings UIST'90, pp.1-9.
http://www.isi.edu/isd/humanoid-papers.html

[Szekely 1992a] P. Szekely. *Interactive Specification of Centext-Sensitive Displays in Humanoid.* USC/Information Science Institute, California. Internal Memo, 1992.
http://www.isi.edu/isd/HUMANOID/humanoid-papers.html

[Szekely et al. 1992] P. Szekely, P. Luo, R. Neches. *Facilitating the Exploration of Interface Design Alternatives: The HUMANIOID Model of Interface Design.* In Proceedings SIGCHI'92, pp. 507-515.
http://www.isi.edu/isd/humanoid-papers.html

[Szekely 1994] P. Szekely. *User Interface Prototyping: Tools and Techniques.* Technical Report, Intelligent Systems Division, University of Southern California. 1994.
http://www.isi.edu/isd/humanoid-papers.html

[Szekely et al. 1993] P. Szekely, P. Luo, R. Neches. *Beyond Interface Builders: Model-Based Interface Tools.* In Proceedings INTERCHI'93.
http://www.isi.edu/isd/humanoid-papers.html

[Szekely et al. 1995] P. Szekely, P. Sukaviriya, P. Castells, J. Muthukumarasamy, E. Salcher. *Declarative interface models for user interface construction tools: the MASTERMIND approach.* Georgia Institute of Technology, Universidad Autonoma de Madrid, University of Technology, Graz.
http://www.isi.edu/isd/Mastermind/Papers/

[Szekely 1996] P. Szekely. *Retrospective and Challanges for Model-Based Interface Development.* Information Sciences Institute (ISI), University of Southern California.
http://www.isi.edu/isd/Mastermind/Papers/

[Sun Microsystems 1998] Sun Microsystems. *Java (TM) Foundation Classes (JFC).* Sun Microsystems Inc. 1998.
http://java.sun.com/products/jfc

[Tamminga et al. 1999] P. Tamminga , D. Faidherbe, L. Misciagna, F. Yuliani. *SanFrancisco GUI Framework: A Primer.* `http://www.ibm.com/Java/Sanfrancisco/`

[Teorey 1999] T. J. Teorey. Database Modeling & Design. Third Edition. Morgan Kaufman Publishers, Inc. San Francisco, California. 1999.

[UIML 2000] UIML. *User Interface Markup Language (UIML) Draft Specification.* Language Version 2.0a. 2000. `http://www.uiml.org/specs/index.htm`

[VXML Forum 2000] VoiceXML Forum. Voice eXtensible Markup Language VoiceXML. Version 1.0. 3/2000. `http://www.voicexml.org/spec.html`

[VSYS 2000] *The Global Info Brokerage And Library Trading Architecture.* Universität Hamburg, Arbeitsgruppe Verteilte Systeme. `http://vsys-www.informatik.uni-hamburg.de/projects/GlobalInfo`

[Wahl 1998] G. Wahl. *UML kompakt.* In OBJEKTspektrum 2/1998. `http://www.sigs.de/publications/docs/obsp/umlkompt/umlkompt.ht`

[Walsh 1998] N. Walsh. *A Technical Introduction to XML.* Arbor Text, Inc. 1997, 1998. `http://nwalsh.com/docs/articles/xml/`

[Weiser 1995] R. Want, B. N. Schilit, N. I. Adams, R. Gold, K. Petersen, D. Goldberg, J. R. Ellis, and M. Weiser. *An overview of the ParcTab ubiquitous computing experiment.* IEEE Personal Communications Magazine, 2(6): pp. 28-43, Dec. 1995. `http://www.fxpal.com/people/schilit/parctab-pcs-jan96.pdf`

[Welie 2001] M. van Welie. *Task-Based User Interface Design.* SIKS Dissertation Series No. 2001-6. `http://www.cs.vu.nl/~martijn/publications_all.html`

[Wheeler 1996] S. Wheeler. *Object-Oriented Programming with X-Designer, 4.1 The MVC-Architecture.* CERN - European Labatory for Particle Physics. 1996. `http://atddoc.cern.ch/Atlas/Notes/004/Note004-1.html`

[Wiecha et al. 1990] C. Wiecha, W. Bennett, S. Boises, J. Gould, S. Green. *ITS: A Tool for Rapidly Developing Interactive Applications.* ACM Transactions on Information Systems. July 1990.

[Zirpins et al. 2001] C. Zirpins, H. Weinreich, A. Bartelt, W. Lamersdorf. *Advanced Concepts for Next Generation Portals.* First International Workshop on Web Based Collaboration WBC'01, to appear. 9/2001.

[Züllighoven 1998] H. Züllighoven. *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz.* dpunkt-Verlag. 1998. German.

Arbeitsteilung

Die Autoren haben folgenden Anteil an der vorliegenden Arbeit geleistet:

	Lars Braubach (in Abschnitten)	Alexander Pokahr (in Abschnitten)
Kapitel 1 Einleitung	zusammen	zusammen
Kapitel 2 Architekturen, Techniken und Werkzeuge	2.3	2.1, 2.2
Kapitel 3 Untersuchte Systeme	3.2, 3.3	3.1
Kapitel 4 Vesuf Konzeption	zusammen	zusammen
Kapitel 5 Fallstudie Global Info	5.2, 5.4	5.1, 5.3
Kapitel 6 Zusammenfassung und Ausblick	zusammen	zusammen

Erklärung

Hiermit versichern wir, die vorstehende Arbeit selbständig und ohne fremde Hilfe unter ausschließlicher Nutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Hamburg, den 10.12.2001

(Alexander Pokahr) (Lars Braubach)