Modellbasierte Generierung von Benutzungsoberflächen

Dissertation

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

vorgelegt von

Andreas Wolff, geboren am 27. Oktober 1978 in Berlin-Köpenick
aus Rostock

Rostock, 5. Juli 2011

urn:nbn:de:gbv:28-diss2011-0110-1

Gutachter:
Prof. Dr. Peter Forbrig, Universität Rostock
Prof. Dr. Heinrich Hußmann, Ludwig-Maximilians-Universität München
Prof. Dr. Roland Petrasch, BEUTH Hochschule für Technik Berlin

öffentliche Verteidigung:
am 23. Mai 2011 in Rostock

Inhaltsverzeichnis

Abstract **vii**

1 Einleitung **1**

1.1 Einordnung der Arbeit 1

1.1.1 Einführung in die Problemstellung 2

1.1.2 Forschungsfragen 4

1.1.3 Ergebnisüberblick 5

1.2 Einführung in Modelle und Benutzungsoberflächen 6

1.2.1 Modellierung in der Softwaretechnik 6

1.2.2 Modelle für die Nutzerinteraktion 8

1.2.3 Historische Entwicklung von UI Modellierung 8

1.3 Interaktionsdefinition mit XML-Derivaten 13

1.3.1 XML-basierte UI-Beschreibungssprachen 14

1.3.2 Schlußfolgerung 22

2 Modellgetriebene Generierung von Benutzungsoberflächen **23**

2.1 Beschreibung des Rostocker Prozessmodells 23

2.1.1 Quellmodelle des modellgetriebenen Ansatzes 26

2.1.2 Dialogmodell 31

2.1.3 Abstrakte Oberfläche 37

2.1.4 Modelle für konkrete Oberflächen 42

2.2 Transformationstechniken 49

2.2.1 Model-To-Text 49

2.2.2 Model-To-Model 51

3 Patterns und Komponenten **53**

3.1 Pattern-Languages 53

3.2 Pattern Language Meta Language 56

3.2.1 Sprachstandard . . . 56
3.2.2 Erweiterungsvorschläge . . . 58
3.3 Textuelle domänenspezifische Sprache für PLML . . . 59
3.4 Grafische Darstellung von Pattern-Katalogen . . . 63
3.5 Pattern als Komponenten . . . 67
3.5.1 Klassifikation der Komponentisierbarkeit von Patterns . . . 67
3.5.2 Aufbau eines Katalogs von Patternkomponenten . . . 71
3.5.3 Deklarationsmodell für Patternkomponenten . . . 77
3.5.4 Einordnungen von Einzelpatterns . . . 81

4 Reengineering Interaktiver Systeme 87
4.1 Reverse Engineering und modellgetriebene Softwareentwicklung . . . 88
4.1.1 Reverse Engineering von Objektmodellen . . . 89
4.1.2 Kriterien für die Ableitung von MOF-Modellen aus Java-Quellcode . . . 91
4.1.3 Durchführung der Java⇒Ecore Transformation . . . 95
4.1.4 Reduzierung der Modelle . . . 98
4.2 Reverse Engineering des Oberflächenmodells . . . 100
4.2.1 Erzeugen eines Swing-Modells aus einer existierenden Oberfläche . . . 100
4.2.2 Modellinstanzen für XUL . . . 104
4.2.3 Ableitung des abstrakten Oberflächenmodells . . . 110
4.3 Forward Engineering . . . 110
4.4 Herausforderungen und Grenzen des vorgestellten Reengineering-Ansatz . . . 111

5 Zusammenfassung 113
5.1 Überblick . . . 113
5.2 Vorschläge für weiterführende Arbeiten . . . 116
5.2.1 Multimodale Nutzerschnittstellen . . . 116
5.2.2 Vorschläge für weiterführende Forschungsansätze . . . 116

A System- und Sprachbeschreibungen 119
A.1 Allzweckmodellierung . . . 119
A.1.1 UIDE . . . 119
A.1.2 DON . . . 120
A.1.3 HUMANOID . . . 122
A.2 Modellierung für Spezialbereiche . . . 124
A.2.1 DRIVE . . . 124
A.2.2 TEALLACH . . . 125

A.3 Systeme mit Aufgabenmodellierungshintergrund . . . 127
A.3.1 ADEPT . . . 127
A.3.2 MASTERMIND . . . 128
A.3.3 FUSE . . . 129
A.3.4 TADEUS . . . 130
A.3.5 TRIDENT . . . 131
A.3.6 L-CID . . . 132
A.3.7 MECANO, MOBI-D . . . 132
A.4 XML Derivate . . . 133
A.4.1 TERESA . . . 133
A.4.2 UIML . . . 135
A.5 Sonstiges . . . 136
A.5.1 CSS . . . 136
A.5.2 JavaScript . . . 137
A.5.3 Einbettung CSS, Javascript und XUL . . . 137

B Quelltexte **139**
B.1 PLML . . . 139
B.1.1 Grammatik für die Sprachversion 1.1 . . . 139
B.1.2 Grammatik der Sprachversion 1.5 . . . 141
B.1.3 xText-Grammatik für PLML 1.5 . . . 143
B.1.4 Beispielkatalog . . . 145
B.2 XUL . . . 147

C Technische Beschreibungen / Implementierungsdetails **149**
C.1 Korrektur des Ergebnis des XUL-XSLT-EMF Importers . . . 149
C.2 Erläuterungen zum XUL-Metamodell . . . 150
C.3 Pattern Language Meta Language . . . 151
C.3.1 Erläuterte Änderungen gegenüber PLML 1.1 . . . 151
C.3.2 Erläuterungen zur xText-Grammatik . . . 153
C.3.3 PLML Metamodell für den grafischen PLML-Editor . . . 155
C.4 Technischer Ablauf der Swing-Abbildung auf Ecore . . . 155

D Patternkomponenten **159**
D.1 Tabellarische Einordnung . . . 159

Abkürzungsverzeichnis **171**

Tabellenverzeichnis **173**

Abbildungsverzeichnis **176**

Listings **177**

Algorithmenverzeichnis **179**

Literaturverzeichnis **191**

Abstract

Diese Dissertation ist eine Arbeit im Bereich der modellbasierten Erstellung von Benutzungsoberflächen. Durch die Erarbeitung von geeigneten Metamodellen und Transformationen dieser Modelle werden die Schnittstellen der Mensch-Maschine-Kommunikation spezifiziert. Die erstellten Spezifikationen werden benutzt, um verwendungsfähigen Quellcode zu generieren. Zum Einsatz gelangen Basistechniken des MDA-Paradigmas: Model-to-Model- und Model-to-Text-Transformationen. Besonderen Fokus legt die Arbeit auf die Integration von Best-Practices in die User Interface (UI) Generierung. HCI-Pattern werden auf Formalisierbarkeit untersucht und sofern möglich, als anwendbare Komponenten in den UI-Erzeugungsprozess eingebracht. Als dritter wichtiger Aspekt wird eine Möglichkeit zur Durchführung von Reengineering untersucht. Existierende Softwaresysteme können analysiert und mit Instanzen der zuvor erarbeiteten Metamodelle beschrieben werden. Auf diese Art kann das MDA-Paradigma auch auf bestehende Alt-Software angewendet werden. Die Pattern-Verwendung und das Reverse Engineering werden an Beispielen demonstriert.

Kapitel 1

Einleitung

1.1 Einordnung der Arbeit

Kaum ein Forschungsthema steht für sich allein und kann vollständig und erschöpfend in einer einzelnen Publikation dargestellt und bearbeitet werden. Auch diese Dissertationsschrift führt die Ideen anderer Forscher weiter, greift bekannte Konzepte auf und verwendet bereitgestellte Frameworks und Werkzeuge. Die detaillierten Zusammenhänge zu anderen Arbeiten werden in den folgenden Kapiteln hergestellt und erläutert. Im Rahmen dieser Einführung soll zunächst nur eine allgemeine Einordnung vorgenommen werden.

Als geeignetes Mittel für eine solche Darstellung hat sich das Forschungsframework für Informationssys-

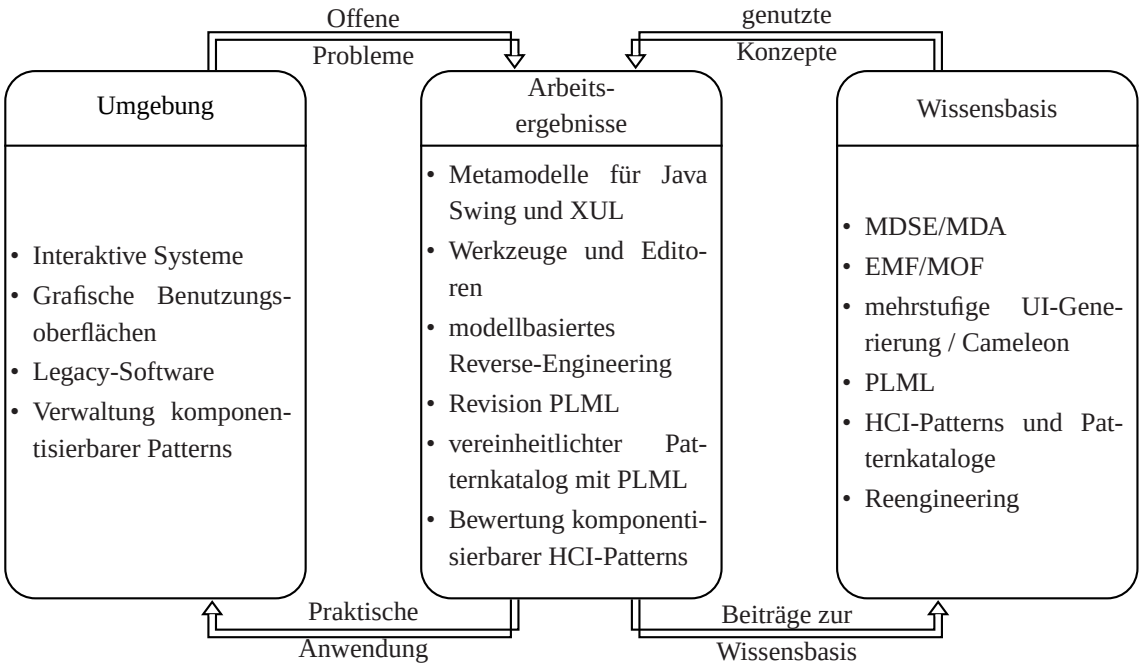

Abbildung 1.1: Inhaltsübersicht, als Instanz des Forschungsframeworks nach Hevner [HMPR04]

teme nach Hevner [HMPR04] erwiesen. Dabei handelt es sich um einen Strukturierungsformalismus für Forschungsarbeiten. In der Abbildung 1.1, wird angelehnt an Hevner, der Inhalt dieser Arbeit zusammengefasst.

Der Grafik ist zu entnehmen, dass die hier betrachteten Forschungsfragen die Problematik der Mensch-Maschine-Kommunikation mit modellbasierten Methoden berühren. Herausgehoben behandelt wird der Bereich der grafischen Benutzungsoberflächen.

Die dem allgemeinen Stand der Forschung entnommenen, als Wissensbasis bezeichneten, Konzepte und Techniken bilden den Ausgangspunkt und Rahmen der Arbeiten. Der Inhalt dieser Dissertationsschrift liefert einen nützlichen Beitrag zur Erweiterung dieser Wissensbasis.

1.1.1 Einführung in die Problemstellung

Seit der Entstehung mechanischer, elektrischer und elektronischer Rechenanlagen ab den 30er Jahren des 20. Jahrhunderts besteht die Herausforderung, diese Maschinen adäquat zu steuern. Diese Steuerung besteht auf unterster Ebene aus Folgen von Instruktionen, die das Rechenwerk eines Computers ausführt. In der Regel beschreiben diese Instruktionen Operationen auf Daten. Eine Abfolge solcher Instruktionen wird als Programm bezeichnet. Der Begriff Software wird oft als allgemeiner Sammelbegriff für Programme [1] benutzt. In seiner heutigen Verwendung umfasst die Software dabei, neben den reinen Ausführungsinstruktionen, auch Daten und Dokumentationen, die für den Ablauf eines Programms notwendig sind.

Die Herangehensweise an die Programmierung von Rechenanlangen hat sich im Laufe der Zeit mit den sich entwickelnden technischen Möglichkeiten gewandelt. Während die ersten Rechner mittels Lochstreifen[2] oder Lochkarten gesteuert wurden, ermöglichten Verbesserungen bei der Datenspeicherung die Hinterlegung von Programmen und Daten auf Massenspeichermedien. Insgesamt hatte Software im Laufe der Zeit die Tendenz größer und komplexer zu werden. Um mit diesem Problem umzugehen, wurde bei der Erstellung der Programme immer weiter vom eigentlichen Rechenwerk abstrahiert.

Die konkrete und damit unterste Ebene einer solchen Abstraktion sind die Instruktionsfolgen für das Rechenwerk, welche in einer Maschinensprache programmiert werden. Assemblersprachen, die nächsthöhere Abstraktionsebene, entstanden als eine menschenlesbare Abstraktion der Maschinensprachen. Assembler und Maschinensprachen sind immer an die zugrundeliegende Rechnerarchitektur gebunden, höhere Programmiersprachen wiederum abstrahieren von Assembler-Quellcode. Solche Programmiersprachen haben eine eigene wohldefinierte Syntax und ein Typsystem. Beides ist im Wesentlichen[3] von der zugrundeliegenden Architektur unabhängig und wird erst von Hilfsprogrammen, den Compilern, in die entsprechende Zielmaschinensprache übersetzt.

Seit dem Beginn der Entwicklung dieser Sprachen zum Anfang der 1950er Jahre entstand eine ungezählte Vielzahl solcher Sprachen. Es kam dabei zur Diversifizierung in Spezialsprachen für Teilbereiche der Informatik und zur Entwicklung von Allzwecksprachen. Diese Sprachen lassen sich nach verschiedenen Charakteristika klassifizieren und verfolgen in aller Regel hauptsächlich ein bestimmtes Programmierparadigma.

Die Erstellung einer Software verläuft mit diesen Programmiersprachen immer sehr ähnlich. Die reine Entwicklung besteht aus dem Bearbeiten des Quellcodes, in der Syntax der Sprache, dem Übersetzen in Maschinensprache, dem Testen und der anschließenden Fehlerbehebung oder sonstigen Weiterbearbeitung des Quellcodes. Dieser Zyklus bildet den Kern der Programmiertätigkeit und ist damit auch integraler

[1] Die Erstverwendung des Begriffes in diesem Sinne wird J. W. Tukey 1958 zugeschrieben
[2] bereits bei Zuses Z-Serie und Colossus
[3] durchaus abhängig bei Datentypenspeicherbreite, Reihenfolge bei Bitwertigkeit

Bestandteil von Softwareentwicklung.

Da Quellcode von Menschen geschrieben wird, gibt es eine Reihe von Problemen damit. Das Schreiben ist erstens ein sehr langwieriger Prozess; für einen durchschnittlichen Entwickler in einer modernen Hochsprache kann mit 200 Zeilen Quellcode pro Tag kalkuliert werden. Diese Zahl beinhaltet dann bereits Layout, Kommentare und Ähnliches, was nur als Lese- und Interpretationshilfe für den Quellcode dient. Bei der Berechnung über den gesamten Verlauf eines Softwareprojekts, wobei die eigentliche Programmierung dann meist nur einen kleineren Teil des Gesamtaufwandes umfasst, ergeben sich sogar Werte von durchschnittlich nur noch 10-20 Quellcodezeilen pro Tag [Voa98]. Obwohl die Metrik Quellcodezeilenzahl, englisch Lines of Code, sehr problematisch[4] ist, mag sie zur Verdeutlichung des Problems dienen, wenn man als Vergleichsgröße die immer wieder berichteten 40 Millionen Zeilen Quellcode für Windows XP heranzieht. Für eine kleinere Bürosoftware[5] sind 100.000 Zeilen Quellcode ebenfalls erreichbar.

Die Wartung, Weiterentwicklung und Fehlerbehebung derartig großer Software ist ein Problem. Moderne Entwicklungswerkzeuge, insbesondere hochentwickelte Programmierumgebungen, und Vorgehensweisen wie Kapselung und Wiederverwendung machen das Problem beherrschbar, lösen es jedoch nicht. Mit jeder weiteren Generation einer Software wird diese komplexer, die Abhängigkeiten größer und gleichzeitig weniger offensichtlich. Die Einarbeitung und das Hineindenken in existierende Software ist eines der größten Probleme im Alltag von Programmierern.

Eine weitere Problemdimension entsteht durch die Art und Weise wie Softwareentwicklung durchgeführt wird. Normalerweise arbeiten mehrere Personen an einem Programm, es wird also verteilt entwickelt. Die Synchronisation der Arbeitsergebnisse, also des Quellcodes, birgt oft Komplikationen.

Als Ausweg aus dieser Problematik bietet sich eine abermalige Abstraktion an. Statt Programme durch das manuelle Eingeben von Quellcode zu entwickeln, sollten Entwickler Modelle spezifizieren. Diese Spezifikation erfolgt in Modellierungssprachen mit eigener Syntax und Semantik.

Aus den spezifizierten Modellinstanzen wird die auszuführende Sequenz von Maschinensprachenbefehlen generiert. Diese Erzeugung läuft meist als mehrstufiger Prozess ab. Ein Code-Generator erzeugt Quellcode welcher anschließend von einem Compiler in Maschinensprache übersetzt wird. Heutige Compiler sind im Allgemeinen recht ausgereift, bei den Code-Generatoren wird an der Erreichung eines vergleichbaren Reifegrades gearbeitet. Ebenfalls denkbar wären Generatoren welche direkt Maschinensprache erzeugen, denn das Zwischenprodukt Quellcode könnte als redundant betrachtet werden und dementsprechend entfallen. Quellcode einer Hochsprache zu erzeugen bietet jedoch den Vorteil, die gesamte vorhandene und durchaus nützliche Palette von Hilfsprogrammen der Softwareentwicklung weiter zu nutzen, wie zum Beispiel Debugger, Deployment[6]-Tools und hochentwickelte integrierte Entwicklungsumgebungen. Darüberhinaus bietet diese Vorgehensweise die Möglichkeit, Alt-Quellcode in die Neuentwicklungen zu übernehmen. Dies kann aus diversen Gründen nützlich sein und ist prinzipiell der Möglichkeit ähnlich, Assembler-Instruktionen in höheren Programmiersprachen einzubetten.

Gemäß den Erfahrungen mit der Einführung der letzten beiden Abstraktionsebenen, wird die Einführung modellbasierter Techniken zur Erhöhung der Produktivität bei der Softwareerstellung und zur Reduktion der Komplexität bei der Bearbeitung der Software führen.

Neben einer verbesserten Handhabbarkeit von großen Softwareprojekten muss der Ansatz des Verwendens von Modellen weitere Vorteile bieten um breite Akzeptanz zu finden. Zwei wesentliche Dinge sind hier hervorzuheben.

[4] Zählweise, Abstraktionsebene, Sprachabhängigkeit
[5] DocFactory [AG]: 80.000 Zeilen Java + 35.000 Zeilen Stored Procedures
[6] Software-Verteilung

Modelle als Abstraktion über Quellcode bieten die Möglichkeit, aus den gleichen Modellen Quellcode für diverse variierende Nutzungsumgebungen zu erstellen. Dies ist vergleichbar mit der ursprünglichen Motivation bei der Einführung der höheren Programmiersprachen. Aus dem Quellcode einer höheren Programmiersprache kann Maschinencode für verschiedene Architekturen erzeugt werden, ebenso kann aus Modellen der Quellcode für verschiedene Hochsprachen erzeugt werden. Die dafür nötigen Hilfsprogramme werden als Model-To-Text Transformatoren bezeichnet.

Ein weiterer Vorteil bei der Anwendung von Modellen und Modellierungskonzepten liegt in der einfachen Integrierbarkeit von bekannt guten Lösungen und einer generell verbesserten Wiederverwendbarkeit von Spezifikationen. Bei der Planung einer Softwareentwicklung ist man in der Regel bestrebt, das bestmögliche Ergebnis zu erreichen. Die Kenntnisse und Fähigkeiten der jeweils an einer Entwicklung beteiligten Personen sind oft unterschiedlich verteilt. Indem man jedem Entwickler Zugriff auf, von überdurchschnittlich guten Entwicklern, erstellte Muster-Lösungen bietet und deren Verwendung abfordert, läßt sich möglicherweise die Gesamtqualität der entstehenden Software heben. Solcherart hervorgehobene (Muster-) Lösungen werden als Best-Practices, dies meint erwiesenermaßen bestmögliche Vorgehensweisen, bzw. Patterns bezeichnet. Patterns finden sich in vielen Bereichen der Informatik und werden in diversen Sammlungen publiziert. Prominentes Beispiel einer solchen Sammlung sind die objektorientierten Patterns der Gruppe um Erich Gamma [GHJV02], andere Patternsammlungen, für HCI-Patterns, werden in Kapitel 3 vorgestellt.

Die Verwendung von Modellierungssprachen führt zu dem Paradigma der modellgetriebenen Softwareentwicklung. Natürlich rechtfertigt die bloße Verwendung von Modellen in verschiedenen Phasen der Softwareentwicklung es noch nicht von einem eigenen Paradigma zu sprechen. Die modellgetriebene Softwareentwicklung stellt Modelle in das Zentrum der Entwicklung, sie treiben die Entwicklung gewissermaßen voran. Das Programmieren mit diesem Paradigma besteht daher aus dem Instanziieren von Modellen und der Festlegung von Transformationen zwischen Modellinstanzen. In diesen Kontext ordnet sich diese Dissertation ein.

1.1.2 Forschungsfragen

Das grundlegende Ziel dieser Arbeit ist die Erstellung von Benutzungsoberflächen unter Verwendung der modellgetriebener Softwareentwicklung. Dazu ist es erforderlich, über geeignete Metamodelle für alle vorzusehenden Abstraktionsebenen zu verfügen. Der Stand der Forschung bietet bereits eine Vielzahl ausgearbeiteter Metamodelle für abstrakte Beschreibungen von Nutzerschnittstellen. Deshalb sind Metamodelle für die Deklaration konkreter grafischer Benutzungsoberflächen von besonderem Interesse.

Somit ergibt sich als erste Forschungsfrage:

> 1. Lassen sich umfassende Metamodelle moderner grafischer Benutzungsoberflächen erstellen?

Aus einer angenommenen positiven Antwort auf diese Frage und auf Grund der Vielzahl von Darstellungsmöglichkeiten und Steuerparametern aktueller Grafiksysteme, erwachsen daraus die Anschlussfragen:

> 2. Wie kann die inhärente Komplexität solcher Metamodelle handhabbar gemacht werden?
> 3. Inwieweit kann nützliche Werkzeugunterstüzung gestellt werden?

Die Konzeption der Mensch-Maschine-Kommunikation ist ein ausführlich erforschter Bereich. Es wäre wünschenswert die dort gewonnenen Erkenntnisse im Rahmen der MDA nutzbar zu haben, daher:

4. Welche Möglichkeiten zur Integration von HCI-Patterns sind denkbar?
5. Welche Patterns sind soweit formalisierbar, dass sie direkt in einem Gesamtprozess anwendbar sind?

Die modellgetriebene Softwareentwicklung steht nicht allein, eine nicht abschätzbare Menge Software wurde ohne Beachtung dieses Paradigmas entwickelt:

6. Welche Möglichkeiten bestehen zur Überführung existierender Softwaresysteme in den modellgetriebenen Ansatz?
7. Lässt sich die Nutzerschnittstelle solcher Altsysteme automatisiert als Modellinstanz abbilden?

Die letzte, über allem schwebende Frage muss der Effizienz des Vorgehens nach diesem Paradigma gelten:

8. Welche Vorteile und Verbesserungen lassen sich für die modellgetriebene Entwicklung von Benutzungsoberflächen zeigen?

1.1.3 Ergebnisüberblick

Es soll bereits an dieser Stelle überblicksartig eine Vorschau auf die erarbeiteten Ergebnisse gegeben werden.

Die Antwort auf die Frage 1 fällt in der Tat positiv aus: es ist möglich, ausführliche Metamodelle für aktuelle grafische Benutzungsschnittstellen nach der Windows Icon Menu Pointer (WIMP) Idee anzugeben. Für die – von einer öffentlichen Community getragenen UI-Sprache – XUL wird in Punkt 2.1.4.2 gezeigt, wie aus deren frei verfügbaren Modellfragmenten ein umfassendes Ecore-Metamodell erstellt werden kann. Gänzlich anders gelagert sind die Probleme bei der Erstellung eines Metamodells des Swing-Frameworks der Sprache Java. Ein Verfahren zu dessen Erstellung wird in Unterabschnitt 4.2.1 vorgestellt. Sowohl bei XUL als auch bei Swing handelt es sich um moderne, komplexe, ausgereifte und weitverbreitete User Interface (UI)-Systeme.

Zur Beantwortung der Fragen 2 und 3 für XUL; das Metamodellvon XUL wurde unter anderem verwendet um damit Teile eines grafischen Editor zu generieren, die Herangehensweise und das Ergebnis sind in Punkt 2.1.4.3 kurz beschrieben. Im Fall des sehr großen Swing-Metamodells wurde ein, in Unterabschnitt 4.1.4 beschriebenes, Verfahren zur regelbasierten Modellreduzierung (Model-Pruning nach [SMBJ09]) erfolgreich genutzt. Dadurch konnte das Swing-Metamodell erheblich verkleinert und damit handhabbarer gemacht werden. Dieses Swing-Modell wird, wie in Kapitel 4 demonstriert, unter anderem in den Reengineering-Werkzeugen benutzt, womit ebenfalls die Fragen 2 und 3 für Swing beantwortet sind.

Eine schwache Form der Integration von Patterns in einen MDA-Gesamtprozess ist es, wenigstens die Patterns als Instanzen eines geeigneten Metamodells zu beschreiben. In Abschnitt 3.2 wird ein Metamodell für diesen Verwendungszweck erarbeitet und anschließend, in Unterabschnitt 3.5.2, verwendet, um aus frei verfügbaren Pattern-Deklarationen einen einheitlichen Patternkatalog zu erzeugen. Zur weitergehenden Integration von Patterns in einen modellgetriebenen Prozess muss sich die Lösung eines Patterns als Transformationsalgorithmus auf eine Modellinstanz anwenden lassen. Soweit ermittelt werden konnte, ist dies nur für eine Minderheit der in der Literatur beschriebenen HCI-Patterns sinnvoll möglich. Details zur Vorgehensweise werden in Abschnitt 3.5 präsentiert. Das Ergebnis dieser Überlegungen, die Patternkomponenten, werden unter anderem durch einen eigens entwickelten XUL-Editor benutzt.

Die Fragen zur Überführung von Alt- oder Legacy-Software in einen modellgetriebenen Prozess werden insbesondere im Kapitel 4 behandelt. Unter bestimmten Umständen ist es möglich äquivalente Modellinstanzen, geeignet zur Weiterverarbeitung, aus nicht modellbasiert entwickelten Systemen zu generie-

ren. Beispielsweise konnte in Punkt 4.2.1.2 und Unterabschnitt 4.2.2 gezeigt werden, dass es prinzipiell möglich ist, aus den Nutzerschnittstellen derartiger Systeme Modellinstanzen für den modellgetriebenen UI-Entwicklungsprozess (MD-UID Prozess) zu gewinnen.

Zur Beantwortung der Frage 8 sei an dieser Stelle auf die Zusammenfassung im Kapitel 5 verwiesen.

Abschließend noch ein genereller Überblick über die Organisation der Arbeit. In dieser Einleitung werden im Folgenden noch Technologien von allgemeiner Bedeutung erläutert, sowie artverwandte Ansätze kurz vorgestellt. Im daran anschließenden Kapitel 2 wird ein umfassender Gesamtprozess für die modellgetriebene Entwicklung von Benutzeroberflächen mit allen notwendigen Teilmodellen beschrieben. Das Kapitel 3 erklärt wie HCI-Patterns formalisiert, katalogisiert und im modellgetriebenen Ansatz benutzt werden können. Den Abschluß bildet Kapitel 4, worin eine Methodik zur Überführung vorhandener System in den modellgetriebenen Ansatz vorgestellt wird.

1.2 Einführung in Modelle und Benutzungsoberflächen

1.2.1 Modellierung in der Softwaretechnik

Mit dem Ziel der Erstellung eines allgemeinen Modellbegriffes definierte Stachowiak 1973 [Sta73] drei Kernmerkmale für Modelle: Abbildung, Pragmatik und Verkürzung. Demnach ist ein Modell immer ein Abbild oder Vorbild einer beliebigen Entität; eine Darstellung künstlicher, natürlicher, realer oder virtueller Originale. Die Pragmatik ist der Einsatzzweck in welchem ein Modell das Original ersetzt oder repräsentiert. Charakteristisch für jedes Modell ist außerdem die Abstraktion, es werden nicht alle Eigenschaften des Originals in das Modell übernommen.

Modellierung und Modelle werden in der Softwaretechnik schon seit langer Zeit eingesetzt. Beispiele dafür sind unter anderem Entity-Relationship-Modelle [Che76] für die Datenmodellierung, Zustandsautomaten für die Verhaltensmodellierung oder auch Architekturmodelle wie etwa das Seeheim-Modell [Tou07] oder PAC[7] als Überlegungen zur Gestaltung von Softwaresystemen. Die derzeit bekanntesten Modelle sind wohl diejenigen der UML[8], bei denen es sich um Modelle zur objektorientierten Softwareentwicklung handelt.

Oft wird Modellierung „nur“ für die Anforderungsanalyse, das Systemdesign und Dokumentationszwecke verwendet.

> In der **modellbasierten Softwareentwicklung** geht es demgegenüber darum, Modelle während des gesamten Softwarelebenszyklus einzusetzen. Je nachdem wieweit die Modelle den Kern eines solchen Lebenszyklusmodells bilden, spricht man entweder von modellbasierter oder von modellgetriebener Softwareentwicklung.
> Unter Modellbasierung, als der schwächeren Integrationsform, wird dabei die regelmäßige Verwendung von dedizierten Modellen verstanden. Eine **modellgetriebene Entwicklung**, oft als Model-Driven Software-Development (MDSD) bezeichnet, geht darüber hinaus. Es ist eine Softwareentwicklungsmethodik die (halb)automatisch durch Modelle vorangetrieben wird.

Transformationen zwischen Modelleninstanzen bzw. die Erzeugung von Text aus Modellinstanzen sind wesentliche Arbeitsschritte in diesem Paradigma. Das Abbilden einer Modellinstanz auf eine andere Instanz wird Model-To-Model (M2M) Transformation genannt. Die Erzeugung von Text aus einem Modell, oftmals Quellcode, wird im Allgemeinen als Model-To-Text (M2T) Transformation bezeichnet.

[7] Presentation, Abstraction und Controller nach [Tou07]
[8] Unified Modelling Language [UML06]

Eine wichtige Ausprägung der MDSD ist die durch die Object Managegement Group (OMG) propagierte Model-Driven-Architecture (MDA). MDA ist ein Vorschlag zur Lösung des Problems der plattformunabhängigen Spezifikation von Anwendungslogik. Diese Lösung besteht in der Anwendung diverser, ebenfalls durch die OMG definierter, objekt-orientierter Techniken. Zum Einsatz gelangen UML und MOF [MOF02] für die Modellierung und etwa QVT[9] für M2M-Transformationen.

Die Meta Object Facility (MOF), frei übersetzt etwa Objektbeschreibungssprache, definiert einen Beschreibungsmechanismus für Modelle. Dieser besteht aus einer Hierarchie von Meta-Modellen, also Modellen die Modelle beschreiben, und einer Definitionssprache. Ursprünglich [MOF02] eine 4-Ebenen Architektur definierend, besteht in den aktuellen Ausprägungen des Standards *Complete* und *Essential* MOF (EMOF) nach MOF 2.0 [MOF06] nur noch die Forderung nach mindestens zwei Ebenen:

> „The minimum number of layers is two so we can represent and navigate from a class to its instance and vice versa."

Der Großteil der Modelle in dieser Arbeit fußt auf Ecore [emf], dem Metamodell des Eclipse Modeling Framework. Zwischen Ecore und *Essential MOF* besteht ein enger Zusammenhang, nach [SBPM09]:

> „... with a focus on tool integration, rather than metadata repository management, Ecore avoids some of MOF's complexities, resulting in a widely applicable, optimized implementation"... „Essential Meta-Object Facility (EMOF) is the new lightweight core of the metamodel that quite closely resembles Ecore. Because the two models are so similar, EMF is able to support EMOF directly as an alternative XMI serialization of Ecore."

Die Abbildung 1.2 versucht eine Visualisierung des Metamodellzusammenhangs. Da der Standard MOF 2.0 keine vordefinierten Modellebenen mehr vorgibt sind die in der Grafik verwendeten M_x Nummerierungen keine offiziellen Benennungen. Insbesondere sind sie nicht als äquivalent zu den in der UML Infrastructure Specification [UML06] benutzten Ebenennummerierungen zu verstehen. Die dortige 4-Ebenen

[9] Query View Transformation [QVT08]

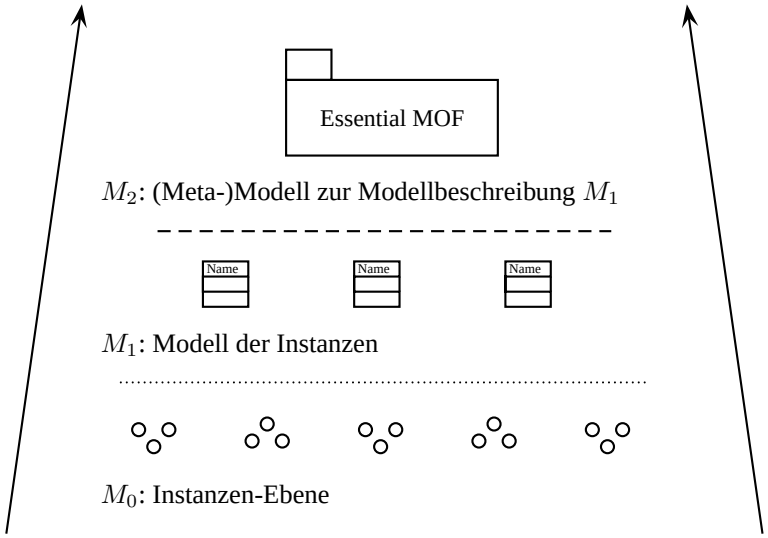

Abbildung 1.2: Instanzen und Modelle im EMOF-Zusammenhang

Architektur fußt auf einer Unterscheidung zwischen Runtime M_0 und Designtime M_1, eine Betrachtungsweise, die für die Modelle in dieser Arbeit nicht ohne Weiteres zutreffend und von Belang ist.

In Abbildung 1.2 ist die unterste Schicht M_0 die Ebene der gespeicherten Informationen bzw. der instanziierten Modellobjekte. In der Hierarchie darüber steht die Modellebene M_1 welche die Struktur der Einträge auf der M_0-Ebene festlegt. Auf der Ebene M_2, der ersten Metamodellebene, wird die Struktur des Modells der M_1-Ebene definiert.

Vereinfachend kann gesagt werden, dass die Struktur der Elemente einer M_{n-1}-Schicht durch die darüberliegende M_n-Schicht definiert wird. Der EMOF-Standard sieht prinzipiell eine beliebig tiefe Metamodellebenenstruktur vor.

1.2.2 Modelle für die Nutzerinteraktion

Die Schnittstellen in der Mensch-Maschine Kommunikation sind seit den Tagen der ersten automatischen Rechner ein Standardproblem der Programmierung. Je nach System und Umgebung ist die jeweilige Schnittstelle anders ausgeprägt. Es existieren insbesondere textuelle, grafische, sprachbasierte sowie berührungs- und bewegungsgesteuerte Interfaces. Ebenso wurden Interfaces entwickelt die mehrere dieser Interaktionsmöglichkeiten in einer Schnittstelle, in einem sogenannten multi-modalen Interface, kombinieren.

Die Programmierung der Nutzungsschnittstellen ist ein aufwändiges Problem und seit langer Zeit Gegenstand der Forschung. Seit etwa dem Ende der 1980er Jahre wird in der modellbasierten UI-Erstellung eine Lösung dieses Problems gesucht. Die Erforschung begann mit Systemen für textuelle User-Interfaces [10]. Nachfolgende Frameworks, Systeme und Ansätze zogen dann auch grafische UIs sowie Weboberflächen in Betracht bzw. wurden gleich auf multi-modale Interfaces ausgelegt.

Im Jahr 2005 haben Gomaa et al. eine Übersicht [GSR05] der bis dato eingeführten oder vorgestellten modellbasierten UI-Generierungsysteme zusammengestellt. Sie identifizierten 27 Ansätze, wovon allerdings drei als XML-UI-Beschreibungssprachen[11] für sich genommen keine eigenen Ansätze darstellen, sondern in anderen verwendet werden. Die verbleibenden 24 Umgebungen oder Systeme enthalten ebenfalls bei weitem nicht alle bekannten Verfahren. Nicht enthalten sind beispielsweise die in Frankreich am INRIA[12] entwickelten Verfahren zu selbstanpassenden User Interfaces [TC99] oder AMF [ST04] - Agent Multi-Facets - ebenfalls ein Framework für die oberen Abstraktionsebenen einer Multiple-UI Entwicklung. Ebenfalls nicht in [GSR05] aufgeführte System sind die lokal an der Universität Rostock entwickelten Ansätze USGP und die Weiterentwicklung des TADEUS-Projektes. Die Abbildung 1.3 stellt die Veröffentlichungszeitpunkte diverser Projekte, der modellbasierten UI-Entwicklung, vergleichend in einem Zeitstrahl dar. Die angegebenen Zeitpunkte entstammen [GSR05] und eigenen Recherchen.

1.2.3 Historische Entwicklung von UI Modellierung

Im Folgenden werden diverse Ansätze der Modellierung von Benutzeroberflächen vorgestellt, die in der Vergangenheit verfolgt und untersucht wurden. Natürlich ist es, in diesem Rahmen, nicht sinnvoll alle der im Zeitstrahl von Abbildung 1.3 eingetragenen Systeme vorzustellen. Daher wurde eine Kategorisierung vorgenommen. Es stellte sich heraus, dass sich die Systeme grob in drei Gruppen einteilen lassen. Jede dieser drei Gruppen wird kurz erläutert und es werden einzelne typische Vertreter vorgestellt.

[10] UIDE [FKKM91], beispielsweise stammt aus dem Jahr 1988
[11] XUL, XIML, UIML
[12] Institut national de recherche en informatique et en automatique

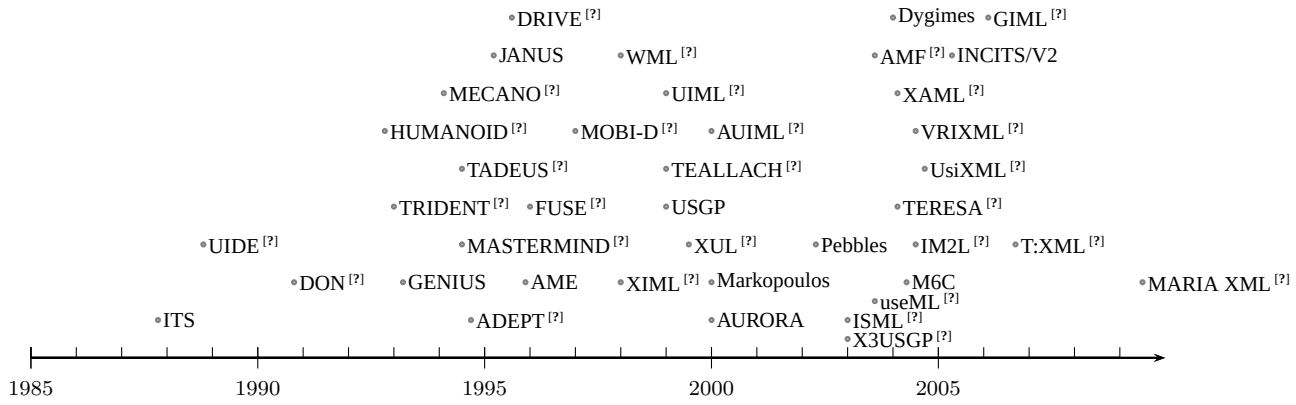

Abbildung 1.3: Zeitliche Einordnung diverser MB-UID Systeme und Umgebungen

Die erste Gruppe, im gewissen Sinne die allgemeinste Gruppe, bilden diejenigen Systeme, die ohne Beschränkung der Einsatzdomäne arbeiten und in ihren Prozessen nicht die Verwendung von Aufgabenmodelle vorsehen. Vertreter dieser Gruppe werden im Abschnitt A.1 vorgestellt.

Schränkt man den Einsatzbereich bzw. die Domäne der zu erstellenden Software genügend ein wird es möglich, diverse Hilfsmodelle domänenspezifisch auszulegen. Auf diese Art wird es wesentlich leichter, bzw. überhaupt erst sinnvoll möglich, die generierten Benutzungsoberflächen mit realer Geschäftslogik zu verknüpfen. Zwei Systeme aus dieser zweiten Gruppe werden im Punkt 1.2.3.2 vorgestellt.

Die dritte Gruppe von Systemen bilden die auf Aufgabenmodellen basierenden Ansätze. Ursprünglich stammt die Idee der Aufgabenmodelle aus der Verhaltensforschung und dient zur Beschreibung der Ziele, die menschlichen Verhaltensweisen zugrundeliegen. In der Softwaretechnik werden Aufgabenmodelle typischerweise im Rahmen der Anforderungsanalyse eingesetzt. Die in diese Gruppe eingeordneten Ansätze gehen darüber hinaus und nutzen die erstellten Aufgabenmodelle zur Strukturierung, Hierarchisierung und Modellierung der Funktionalitäten einer Anwendung. Aufbauend auf den in den Aufgabenmodellen identifizierten Funktionalitäten und Zusammenhängen werden schließlich Benutzungsoberflächen generiert. Die generelle Idee der Aufgabenmodelle wird in verschiedenen Ausprägungen umgesetzt. Als etablierte Vorgehensweisen seien hier GOMS[13], TKS[14] oder CTT[15] genannt. Die konkreten Unterschiede dieser Ansätze sollen hier nicht diskutiert werden. Dennoch werden bei den Erläuterungen in Punkt 1.2.3.3 und Abschnitt A.3 nicht nur die Systeme vorgestellt die Aufgabenmodelle verwenden, sondern jeweils auch das im Ansatz benutzte Konzept für Aufgabenmodelle kurz angerissen.

Im Anhang der Arbeit wird eine Auswahl an Systemen, Ansätzen und Frameworks vorgestellt, bei denen es sich um wesentliche bzw. repräsentative Kandidaten der drei vorstehend charakterisierten Gruppen dar.

1.2.3.1 Ansätze ohne Beschränkung des Einsatzbereichs

Idealerweise sollte ein modellbasiertes System funktionsfähige Benutzungsoberflächen für beliebige Anwendungsbereiche liefern. UIDE, DON und HUMANOID gehören zu den zeitlich frühesten Ansätzen die versuchten diesen Anspruch mittels deklarativer UI-Beschreibungen zu erfüllen. Die UI-Deklarationen erstrecken sich daher nicht nur auf Layout- und Designaspekte, sondern definieren auch die Anbindung an

[13] Goals, Objects, Methods, Selection Rules [CNM00]
[14] Task Knowledge Structures [JJ91]
[15] Concurrent Task Trees [Pat99]

die Systemumgebung. In allen drei Ansätzen werden die UI-Deklarationen zur Laufzeit interpretiert und einem Benutzer als funktionsfähige Anwendung dargestellt. Unterschiede bestehen in der Zielmodalität, textuell oder grafisch, den benutzten Hilfsmodellen und der Notation der Deklarationen. Letztlich konnte sich zwar keines dieser Systeme durchsetzen, die in Abschnitt A.1 dargelegten Überlegungen zu den Grundprinzipien sind aber auch heute noch relevant.

1.2.3.2 Modellierung für Spezialbereiche

Beschränkt man den Einsatzbereich, also die Domäne, für welche Anwendung und Benutzeroberfläche erzeugt werden sollen, können genau auf den Problembereich zugeschnittene Modelle verwendet werden. Das ist eine erhebliche Erleichterung gegenüber den breiter angelegten Ansätzen zur allgemeinen unbeschränkten Modellierung, insbesondere gestaltet sich die Anbindung von Verarbeitungslogik an die modellierte und generierte Oberfläche einfacher. Die Einschränkung des Einsatzbereich führt oft auch zur Reduzierung der in Frage kommenden UI-Elemente und generell der möglichen Nutzerinteraktionen. Diese Reduzierung der Problemkomplexität wirkt sich so gravierend aus, dass die im Abschnitt A.2 vorgestellten Ansätze DRIVE und TEALLACH in der Lage sind, funktionsfähige Software zu erzeugen.

1.2.3.3 Systeme mit Aufgabenmodellierungshintergrund

Die Aufgabenmodellierung, ist eine Grundlagentechnik der Anforderungsanalyse. Eine kurze Charakterisierung dieses Ansatzes findet sich bei Markopoulos [MPWJ92]:

> „A task model is a symbolic representation of that which the modeller considers significant about a task. Task models may be used descriptively, predictively, and prescriptively, though in practice few existing techniques have the power to do all.“

Es existieren diverse Umsetzungen der Aufgabenmodell-Idee. Allen gemeinsam ist die Vorstellung, Anwendungen vom Standpunkt der späteren Nutzer zu beschreiben. Von diesen Nutzern wird angenommen, mit dem Einsatz der Software für ihre Zwecke, ein Ziel erreichen zu wollen. Zur Erreichung dieses Zieles sind bestimmte Aufgaben zu absolvieren und dabei Artefakte zu bearbeiten. Normalerweise werden die identifizierten Aufgaben dazu in einer Hierarchie angeordnet. Zur Festlegung der Abarbeitungsreihenfolge kommen temporale Operatoren zur Anwendung. Diese Operatoren definieren die zeitliche Relation zwischen Aufgaben, etwa Parallelität oder Nacheinanderausführung. Die hinter diesen Operatoren stehende Logik hat ihre Wurzeln in der Prozessalgebra.

Aufgabenmodelle sind im Prinzip für alle denkbaren Domänen einsetzbar. Daher existiert theoretisch auch keine Beschränkung für aus Aufgabenmodellen erzeugte Anwendungen.

Das Erzeugen von Benutzungsoberflächen ausgehend von Aufgabenmodellen wurde in der Forschung ausführlich untersucht. Die in Abschnitt A.3 vorgestellten Systeme nutzen die in den Aufgabenhierarchien enthaltenen Informationen um mittels Heuristiken prototypische Nutzungsoberflächen zu kreieren. Allerdings beschreiben die von den Ansätzen jeweils eingesetzten Aufgabenmodelle die späteren Anwendungen auf einer hohen Abstraktionsebene. Verwendet man sie als Datenquelle in einem modellbasierten UI-Entwicklungsprozess liefern sie im Wesentlichen die Namen der vorgesehenen Aktionen und welche dieser benannten Aktionen zu einem gegebenen Zeitpunkt ausführbar sind. Allein auf Basis dieser Informationen ist es kaum denkbar, gebrauchstaugliche Nutzungsschnittstellen zu generieren. Die vorgestellten Systeme benutzen dementsprechend weitere Modellarten als Primärquellen in ihrem jeweiligen Ansatz.

1.2.3.4 Fazit

Insgesamt haben die in der Vergangenheit entwickelten Systeme bisher noch nicht den Reifegrad erreicht, der es ermöglicht, produktive Anwendungen mit ihnen zu entwickeln. Wieder und wieder findet sich auch diesbezügliche Kritik in Veröffentlichungen, beispielhaft seien Puertas MECANO-Paper [Pue96] und Szekely Ausführungen zu Mastermind [SSC+96] genannt. Die Kritik bei [SSC+96] fällt eher generell aus, während Puerta bereits 1996 zwei Forderungen [Pue96] an erfolgreiche modellbasierte UI-Systeme aufstellt:

> „The two central ingredients for success in model-based systems are: (1) a declarative, complete, and versatile interface model that can express a wide variety of interface designs, and (2) a sufficiently ample supply of interface primitives, elements such as push-buttons, windows, or dialogue boxes that a model-based system can treat as black boxes."

Diese Hauptzutaten sah er in den zum damaligen Zeitpunkt bekannten Systemen nicht enthalten. Leider erfüllen auch Puertas eigene Veröffentlichungen rund um MECANO und das bis heute nicht öffentlich freigegebene XIML diesen selbstgestellten Anspruch nicht voll.

Die Schwächen der im Bereich der MB-UID vorgestellten Systeme liegen eher nicht im Bereich der Aufgabenmodellierung oder der Ableitung abstrakter Oberflächen. Ganz im Gegenteil, trotzdem im Einzelfall diverse Operatoren in einem Ansatz möglicherweise nicht vorhanden sind, und falls benötigt über andere Konstrukte nachgebildet werden müssen, scheint die Ausdruckskraft der Aufgabenmodelle im Zusammenspiel mit den Operatoren der Communicating Sequencing Processes zur Beschreibung der Nutzerziele ausreichend zu sein. Selbst aktuellere Arbeiten, wie die ConcurTaskTrees [Pat99] oder HOPS [DF09], nutzen im Kern immer wieder diese bereits durch Hoare 1978 [Hoa78] bekannt gemachte Technik.

1.2.3.5 Einheitliche Klassifikation

Die Abfolge der Abstraktionsebenen und die benutzten Ausgangs- und Hilfsmodelle ähneln sich in den diversen Ansätzen aller drei Gruppen. Das ist kein Zufall sondern ergibt sich aus der verfolgten Verfeinerungsstrategie. Es existiert eine vereinheitlichte Klassifizierung - das CAMELEON Referenzframework - dessen zugrundeliegenes Ebenenmodell in Abbildung 1.4 dargestellt ist.

CAMELEON wurde primär als Referenzframework zur Klassifizierung von MB-UID Ansätzen vorgesehen, es gibt keine eigene Softwareimplementierung oder ähnliches. Im Abstract von [CCT+03] wird das CAMELEON Referenzframework beschrieben als ein

> „framework that serves as a reference for classifying user interfaces supporting multiple tar-

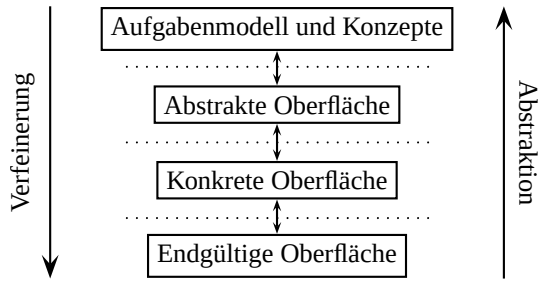

Abbildung 1.4: Die vier Abstraktionsebenen des CAMELEON-Referenzframeworks nach [CCT+03]

gets, or multiple contexts of use in the field of context-aware computing.“.

Die Klassifizierung nach CAMELEON besteht darin die in einem Ansatz auftretenden Modellarten in die Referenz-Layer einzuordnen. Ebenso werden möglichen Übergänge zwischen den Layern, sowie die daran jeweils beteiligten Teilmodelle gekennzeichnet und klassifiziert. Da eine Reihe von Systemen, wie z.Bsp. MECANO über eine eigene Laufzeit-Umgebung, mit eventuell eigenen Modellarten, verfügen, ist bei CAMELEON auch eine Unterscheidung in Design- und Runtime vorgesehen.

Die (Teil-)Modelle eines zu klassifizierenden Verfahrens werden in vier Referenzebenen eingeordnet. In der Reihenfolge des absteigenden Abstraktionsgrads sind dies: *Task Model and Concepts*, *Abstract Interface*, *Concrete Interface* und *Final Interface*. Diese wurden in der Grafik von Abbildung 1.4 ins Deutsche übertragen. Eine erwähnenswerte Abweichung in der Verwendung von Begrifflichkeiten, gegenüber den Termini in dieser Arbeit, liegt in der Unterscheidung zwischen konkretem und endgültigem Userinterface. Nach CAMELEON ist das konkrete Oberflächenmodell ein Teilmodell welches konkrete Interaktionsobjekte verwendet, das jedoch nach [CCT+03] **nur** in der Modellierungsumgebung benutzt wird:

> „[. . .] a Concrete UI makes explicit the final look and feel of the Final User Interface, it is still a mockup than runs only within the [. . .] development environment.“

Erst auf der Modellebene der endgültigen Oberfläche (Final UI = FUI) liegt demnach eine werkzeugunabhängige Form der Oberfläche vor, dabei wird es sich typischerweise um generierten Quellcode handeln. Deklarative UI-Beschreibungen welche von unabhängigen Renderern interpretiert werden zählen ebenfalls zur FUI. CAMELEONs Unterscheidungsebenen tragen zwar zur besseren Klassifizierbarkeit historischer MB-UI Ansätze bei, wirken jedoch insbesondere bei Verwendung XML-basierter UI-Beschreibungssprachen künstlich; eine nützliche Unterscheidung zwischen CUI und FUI ist zumeist nicht gegeben.

Die Abbildung 1.5 zeigt eine, durch die CAMELEON-Entwickler selbst vorgenommene, Beispielklassifikation. Zur Erklärung einzelner Syntaxelemente sei auf [CCT+03] verwiesen. Im linken Bereich der Darstellung sind werden die von dem jeweiligen Ansatz benutzten Metamodelle aufgeführt, bei CAMELEON als ontologische Modelle bezeichnet. Rechts davon sind zwei Konfigurationen dargestellt. Als Konfiguration bezeichnet man ein Ensemble von Modellen und die Notation von deren Zusammenhängen.

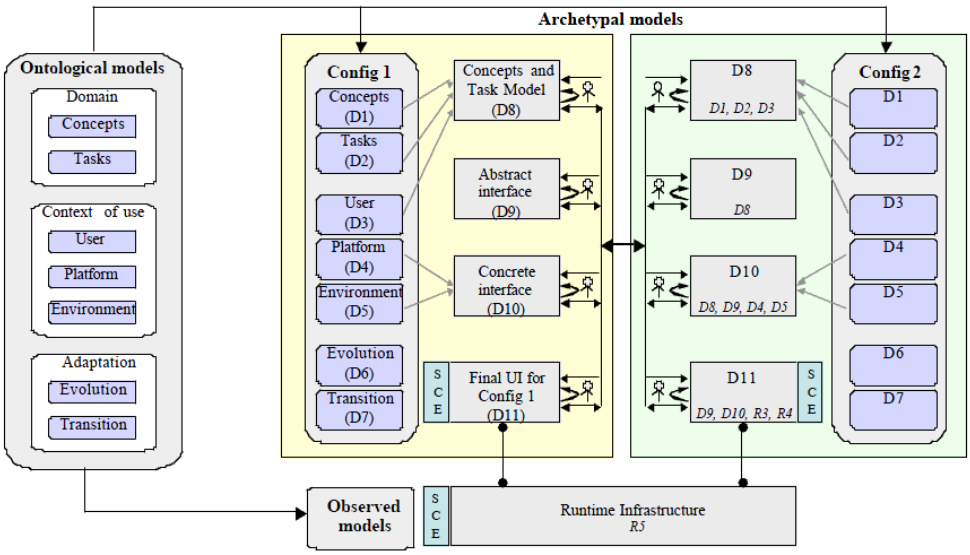

Abbildung 1.5: Beispielhafte CAMELEON-Klassifikation, nach [CCT+03]

Unter dem Namen Konfiguration 1 werden die zur Designtime verwendeten Modelle und deren Abhängigkeiten dargestellt. Aus dieser Darstellung des Beispielansatzes ist ersichtlich, dass während der Designtime keine Modelle zur Beschreibung von abstrakter oder endgültiger Oberfläche verwendet werden.

Erst im Zusammenhang mit Konfiguration 2, hier die Laufzeit-Modellzusammenhänge visualisierend, wird ersichtlich, dass das Modell der abstrakten Oberfläche aus den D8 (Concepts and Task Models) zugeordneten Teilmodellen abgeleitet wird. Die Erzeugung der endgültigen Oberfläche findet ebenso zur Laufzeit statt und benötigt dazu das Modell der abstrakten und der konkreten Oberfläche, sowie die Laufzeit-Instanzen von Nutzer- und Plattform-Modell. Die Laufzeitumgebung selbst wird derweil in einer Instanz des Umgebungsmodells beschrieben.

Durch die gewählte Abstraktionsebene und die inhärente Flexibilität des CAMELEON-Referenzframeworks lassen sich die allermeisten modellbasierten- und modellgetriebenen Oberflächenerstellungssysteme klassifizieren. Selbstverständlich auch Systeme die auf etablierten XML-UI Beschreibungssprachen aufsetzen, trotz der bereits angedeuteten fraglichen Nützlichkeit der FUI-Ebene für deren Fall.

1.2.3.6 Motivation für diese Arbeit

Nach Betrachtung vieler Veröffentlichungen zu den MB-UI Systemen entstand für mich der Eindruck, dass das Ziel der tatsächlichen Erzeugung funktionsfähiger und vollständig gestalteter Interfaces nicht im Vordergrund des Interesses steht. Meist werden einfache Beispiele mit wenigen Labels und Push-Buttons gezeigt deren Layout und Design durch eine nicht weiter erklärte Laufzeitumgebung bereitgestellt wird. Dieser letzte Schritt der tatsächlichen Oberflächenerzeugung bleibt in den Systembeschreibungen oft zu vage, oder es werden anscheinend vorhandene Widgetbibliotheken und Metamodelle nicht weiter beschrieben.

Bei oberflächlicher Betrachtung fallen keine besonderen oder gravierenden Probleme auf, die die Entwicklung ausführlicher Metamodelle für CUI oder FUI behindern. Klar scheint das Metamodelle zur Beschreibung realer Oberflächen wesentlich größer sein werden als die verhältnismäßig übersichtlichen Modelle der höheren Abstraktionsebenen. Dies ist nicht gleichbedeutend damit das sie komplexer oder schwieriger sind. Sie sind nur erheblich größer.

Grundsätzlich ist es nicht ausreichend einfach ein neues Metamodell zu entwerfen, die damit modellierten Instanzen müssen auch interpretiert oder gerendert werden. Da kein überzeugender Grund besteht nur für die MB-UID ein neues eigenes UI-Toolkit zu entwickeln, sollte man sich für diesen Zweck an die existierenden Toolkits anlehnen. Das bedeutet Transformationen zu erstellen, seien es Generatoren, Compiler oder Interpreter, die die Instanzen des eigenen Metamodells in eine dem Toolkit gerechte Form überführen.

Letztlich handelt es sich dabei um eine langwierige Ingenieursarbeit, die allerdings aus meiner Sicht unbedingt durchgeführt werden muss, um die modellgetriebene UI-Entwicklung voranzubringen. Die in Punkt-2.1.4.2 dargelegte Ableitung eines XUL-Metamodells und die in Unterabschnitt 4.2.1 beschriebene Erstellung des Metamodells von Java Swing sollen genau diese Lücke füllen.

1.3 Interaktionsdefinition mit XML-Derivaten

XML [XML] wurde ursprünglich für den Datenaustausch im Internet, konkreter im HTML-dominierten World Wide Web, konzipiert. Das Standardisierungsgremium legte daher Wert darauf das die Daten in einer für Menschen lesbaren und erfassbaren („legible and reasonably clear“) Form enkodiert würden. Natürlich ist, unabhängig vom Übertragungsmedium, für einen Datenaustausch immer auch eine Standar-

disierung des benutzten Datenformates nötig. Designziel für XML war es, das Format der Daten schnell definierbar, so kurz wie möglich aber formal korrekt zu beschreiben. Zur Umsetzung dieser Ziele wurden, in der Notation und Semantik, an EBNF angelehnte Grammatiken verwendet. Es war ausdrücklich kein Ziel von XML, kurze, kleine bzw. knapp gehaltene Dateninstanzen zu enthalten.

Diese Designziele können wohl als erreicht bezeichnet werden, denn die XML hat seit ihrer Entstehung 1996 weite Verbreitung gefunden. Die bei XML-Dokumenten anzutreffende Kombination aus, zur automatisierten Weiterverarbeitung ausreichender, Formalisierung bei gleichzeitiger Lesbarkeit für Menschen ist offenbar für viele Anwendungsgebiete hinreichend und nützlich. Negativ an XML-Dokumenten kann der inhärente Zwang zur Hierarchiebildung in der Datenstruktur sein, ebenso wird oft die teils ausufernde Größe von XML-Dateien kritisiert.

XML findet auch in der modellbasierten Softwareentwicklung häufig Anwendung. In den letzten Jahren ist es hier zum dominierenden Speicherformat für Modelle und Modellinstanzen aufgestiegen. Einen wesentlichen Anteil daran hat aus meiner Sicht das Eclipse Modeling Project [EMP]. Daneben setzen auch eine Vielzahl der außerhalb von EMP bestehenden Lösungen, wie TERESA [BP05] und CTTE [CTT] oder Puertas UI-FIN [PH09] für ihre Datenhaltung auf XML.

Selbstverständlich sind nicht alle aktuellen System XML-basiert. Ein Beispielansatz welcher weder auf XML noch auf das EMP aufsetzt ist MOOSE [Assc]. MOOSE nutzt als Basismodell den FAMIX-Core [Assb], ein Metamodell zur Beschreibung der statischen Struktur objektorientierter Systeme; Parallelen zu Ecore lassen sich finden. Als Speicher- und Austauschformat wird MSE [Assd] genutzt, dieses ist wie XML textbasiert und generisch. MSE verwendet eine LISP-ähnliche Syntax und bietet zusätzlich einen integrierten Referenzierungsmechanismus.

Der große Vorteil von XML ist eine reichhaltig vorhandene Technologie-Infrastruktur. Dies meint die Vielzahl vorhandener Software zur Bearbeitung, Darstellung und Transformation von XML-Dokumenten. Eventuelle Vorteile von beispielsweise MOOSE scheinen nicht so gravierend, diesen XML-Vorteil aufzuwiegen.

1.3.1 XML-basierte UI-Beschreibungssprachen

Strebt man die Modellierung von Benutzungsoberflächen auf Basis von XML an, ist es wünschenswert, diese Oberflächen ebenfalls mit XML zu beschreiben. In den letzten Jahren entstand daher eine schwer überschaubare Anzahl XML-basierter Oberflächenbeschreibungssprachen, so listet ein ITEA-Projektbericht zu UsiXML [Usic] siebenundzwanzig UIDLs auf. Teilweise handelt es sich um reine Markupsprachen zur Deklaration von CUI beziehungsweise FUI Instanzen, andere wiederum sind hybrid konzipiert und versuchen zusätzlich noch andere Abstraktionsebenen des CAMELEON-Frameworks in ein und derselben Sprache zu integrieren.

In Tabelle 1.1 sind eine Reihe XML-basierter Sprachen und MB-UID Systeme aufgeführt. Soweit diese Sprachen der Deklaration von CUIs dienen werden sie gewöhnlich als UIDL - User Interface Description/-Declaration Language - bezeichnet. Neben reinen UIDLs enthält die Tabelle auch vollständige Ansätze, die wesentlich auf XML-Technologie aufsetzen.

Wie in der Informatik, und besonders auch im XML-Umfeld, üblich, ist jede Sprache unter einem Kürzel bekannt; dieses wird in Tabelle 1.1 in der Spalte Kurzname angegeben, den eigentlichen Namen zeigt die Spalte Bezeichnung. In der mit CAMELEON überschriebenen Spalte findet sich eine Grobzuordnung der durch die jeweilige Sprache abgedeckten Abstraktionsebenen des Referenzframeworks. Die Abkürzung „TC“ steht für die oberste CAMELEON-Ebene Task and Concepts.

Kurzname	CAMELEON	Bezeichnung
AUIML	CUI	Abstract User Interface Markup Language
AMF	TC, AUI	Agents Multi-Facets
GIML	CUI	Generalized Interface Markup Language
ISML	TC, CUI	Interface Specification Meta Langugae
IM^{2}L	AUI	Interaction Multimodal Markup Language
MARIA	TC,AUI,CUI	Model Based Language For Interactive Applications
TERESA	TC, AUI	Transformation Environment for Interactive Systems Representations
T:XML	CUI	—
UIML	CUI	User Interface Markup Language
useML	AUI	Useware Markup Language
UsiXML	AUI,CUI	User Interface Extensible Markup Language
VRIXML	CUI	Virtual Reality Interaction XML
WML	CUI	Wireless Markup Language
X3USGP	AUI, CUI	XML-based User Interface Specification and Generation Process
XAML	CUI	Extensible Application Markup Language
XIML	CUI	Extensible Interface Markup Language
XUL	CUI	XML User Interface Language

Tabelle 1.1: Aufzählung bekannterer XML-basierter Sprachen zur UI-Definition

In den nachfolgenden Unterabschnitten wird versucht, Zweck und Inhalt einiger der in Tabelle 1.1 aufgezählten Sprachen kurz zu umreißen. Eine erschöpfende Übersicht oder ein Vergleich wurde nicht angestrebt.

1.3.1.1 Modellbasierte UI-Entwicklungsansätze auf XML-Basis

Die XML-Sprachen dieses Abschnittes dienen nicht primär der Deklaration konkreter Benutzungsschnittstellen. Vielmehr können sie als Fortführung der Ideen der in Unterabschnitt 1.2.3 dargelegten Ansätze aufgefaßt werden. Die Grundideen werden kontinuierlich weiterverfolgt, lediglich die Technologie gewechselt. Nachfolgend werden die drei zur Zeit wichtigsten Vorgehensweisen auf diesem Gebiet etwas ausführlicher vorgestellt: MARIA, als Fortführung von CTT und TERESA, UsiXML, als aktuelles EU-gefördertes Forschungsprojekt und XIML, ein kommerziell erfolgreicher Ansatz.

MARIA

MARIA führt die Ideen von TERESA fort, es ist in vielerlei Hinsicht deren Nachfolger. Im Kern verfolgt MARIA den selben modularen Aufbau und nutzt die gleichen Abstraktionsebenen.

Nach Aussage der Entwickler [PSS09] konzentriert sich das MARIA-Projekt auf die Modellierung von Benutzungsoberflächen für Web-Service basierte Anwendungen. Das Projekt ruht auf zwei Säulen, dem MARIA-Tool und der Sprache MARIA-XML. Die Sprache selbst dient lediglich der Deklaration abstrakter Oberflächen, die Modellierung der eigentlichen konkreten User Interfaces wird, wie bei TERESA, externen XML-Sprachen übertragen. MARIA-Tool ist sowohl Modelleditor als auch Transformations- und Laufzeitumgebung.

Erfahrungen aus dem TERESA-Projekt haben Paternò und seinen Kollegen gezeigt das die ausschließliche Fokussierung auf das Aufgabenmodell den Möglichkeiten und Anforderungen an moderne User Interfaces

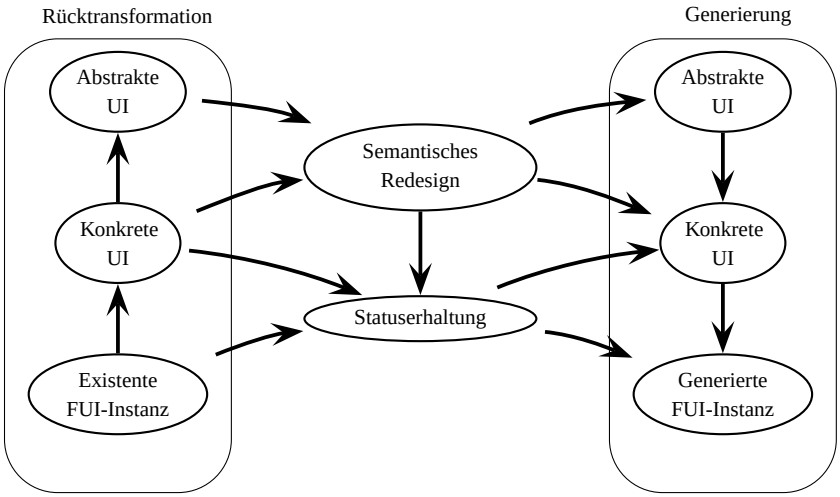

Abbildung 1.6: Modellzusammenhänge im MARIA-Migrationsprozess, nach [PSS09]

nicht gerecht wird. MARIA-XML führt daher gegenüber seinem TERESA-Pendant eine Reihe zusätzlicher Konzepte ein.

Mit MARIA-XML gibt es nun ein explizites Datenmodell für die von der UI dargestellten Daten. Im Grunde entspricht dies der Einführung eines Domänenmodells, wie es ja auch in den klassischen Verfahren (siehe Unterabschnitt 1.2.3) vielfach benutzt wird. Mit der Einführung eines Event-Konzeptes, auf der Ebene der Interface-Deklaration, wird der UI-Kontrollfluss in die Sprache integriert. In den Grundzügen ist dieses Konzept mit dem Ereignis-System von Humanoid, siehe Seite 122, vergleichbar.

Dazu kommen weitere Zugeständnisse an real existierende Oberflächensysteme. So ist es in MARIA-XML Deklarationen möglich, bestimmte Interaktionselemente als kontinuierlich-aktualisierend zu markieren. Als besonders nützlich wird diese Annotation in Zusammenhang mit AJAX[16] bezeichnet; seine Einführung wirkt daher zunächst wie eine Konzession an den Web 2.0-Hype hat aber dennoch das Potential, in späteren Ausbaustufen für vielerlei Zwecke nützlich zu sein.

Teile der mit MARIA-XML beschriebenen Oberflächen können als dynamisch austauschbar deklariert werden. Die alternativen Darstellungen werden dazu als Komponenten vordeklariert und aufgrund definierter Einsatzbedingungen oder durch explizite Auswahl eingesetzt.

Die bereits mit TERESA begonnenen „Migratory Interfaces“ , siehe Unterabschnitt A.4.1, werden auch bei MARIA weiter betrachtet, allerdings aus einem leicht anderen Blickwinkel. Da die UI-Sprache MARIA-XML bereits Datenmodell und Kontrollfluss enthält, bietet ein übergeordnetes CTT-Aufgabenmodell kaum zusätzlichen Informationsgewinn für die Übertragung von Oberflächenzuständen zwischen verschiedenen Geräten. Die Migration setzt daher nur noch auf die unteren drei CAMELEON-Ebenen auf. Den Zusammenhang der Modelle in diesem Prozess zeigt Abbildung 1.6. Darin visualisieren Pfeile, ähnlich wie bei den Darstellungen des CAMELEON-Referenzframeworks, die Richtung des Informationsflusses, es handelt sich nicht notwendigerweise um Modelltransformationen.

Für MARIA-XML wird mittelfristig die Standardisierung durch ECMA oder W3C angestrebt. Ein diese Richtung vorbereitender Workshop[17] fand im Mai 2010 in Rom statt.

[16] Asynchrones JavaScript und XML
[17] `http://www.w3.org/2010/02/mbui/cfp.html`

UsiXML

UsiXML zählt ebenfalls zu den einflussreicheren XML UIDLs. Veröffentlichungen über und zu UsiXML finden sich zahlreich seit 2004, z.B. [VLM+04]. Die Sprache wurde im Jahr 2006 dem W3C zur Standardisierung vorgeschlagen.

UsiXML ist nicht nur eine Sprachdefinition, es ist ein vollständiges Framework zur Durchführung modellgetriebener Entwicklung von Benutzungsoberflächen. Dementsprechend gehören zum UsiXML-Projekt diverse Editoren, Werkzeuge und Laufzeitumgebungen.

Ein bemerkenswerter Unterschied zwischen UsiXML und anderen, auf dem MDA-Konzept der OMG beruhenden Systemen ist, dass UsiXML auf Graphtransformationen setzt. Statt Sequenzen von Model-To-Model Transformationen abzuarbeiten, wird bei UsiXML immer der Baum des XML-Dokumentes transformiert. In der Konsequenz bleibt man also immer in derselben Modellinstanz.

Dennoch werden in einem UsiXML-Dokument eine Reihe von Teilmodellen unterschieden, in Abbildung 1.7 stellt alle Vorgesehenen dar. Selbstverständlich ist es nicht erforderlich, dass für eine einzelne Oberflächenbeschreibung alle Teilmodelle instanziiert werden. Für ein Minimalmodell genügt die Definition einer einzelnen CUI-Instanz.

Die Teilmodelle decken alle oberen Ebenen des CAMELEON-Frameworks ab. Im UsiXML-Ansatz gibt es jedoch keine explizite FUI-Ebene. Die Laufzeitumgebungen interpretieren direkt die Instanzen der CUI.

Das UsiXML-Projekt ist von Anfang an sehr ambitioniert angelegt. Leider haben aus meiner Sicht wichtige Teile des Frameworks nicht die notwendige Reife entwickelt. Im Rahmen einer studentischen Arbeit [Woi09] wurde UsiXML als CUI-Modell für den Rostocker USGP [Mül03]-Ansatz genutzt. Es erwies sich jedoch als schwierig die generierten, und nachprüfbar standardkonformen, Oberflächenbeschreibungen in den verfügbaren UsiXML-Renderern darstellen zu lassen.

Einzelne Teilmodelle von UsiXML lassen sich kaum sinnvoll in das CAMELEON-Framework einordnen. Das Ressourcen-Modell etwa, ein Datencontainer für die Kontextanpassung von UI-Resourcen. Bei wechselnden Kontexten, die naturgemäß im Kontext-Modell beschrieben sind, werden die Interaktionselemente gemäß den Informationen aus dem Ressourcen-Modell adaptiert. Die bis dato einzig funktionierende Kontext-Resourcenkombination ist die Internationalisierung von Teilen der Oberfläche. Beschriftungen

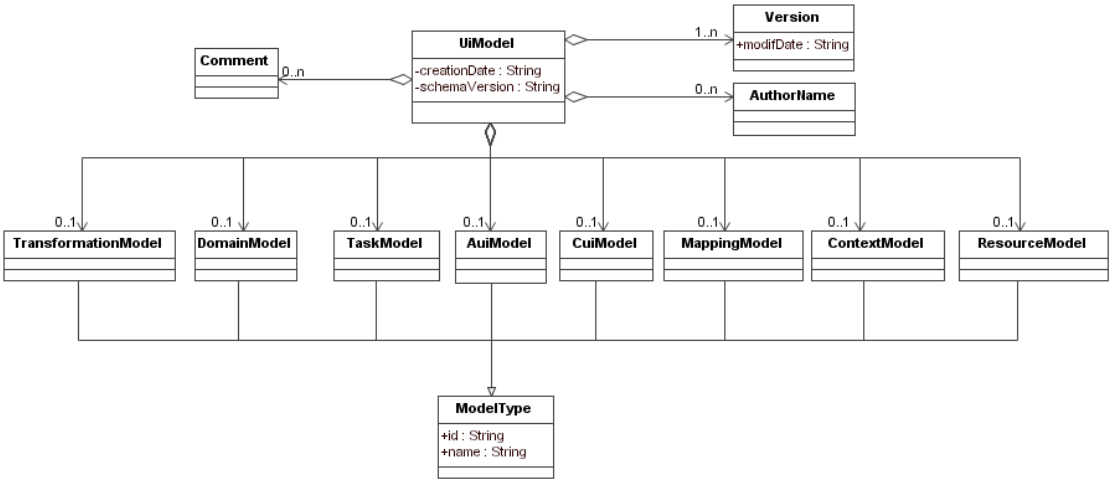

Abbildung 1.7: Klassendiagrammdarstellung des UsiXML-Schema, nach [VC]

und andere Texte können bei Sprachwechseln entsprechend übersetzt angezeigt werden, verantwortlich dafür ist allein die Laufzeitumgebung, dieser Mechanismus wird ohne Graphtransformationen umgesetzt.

Von den übrigen Teilmodellen werden Mapping- und Transformationsmodell dazu genutzt, die Graphentransformationen in UsiXML selbst zu beschreiben. Sie enthalten daher Knotenselektoren für das Matching, Ergebniskonfigurationen sowie Anwendbarkeitsbedingungen.

An der Weiterentwicklung von UsiXML wird aktuell gearbeitet. Angefangen im April 2009, läuft ein gesamteuropäisches Projekt mit dreieinhalbjähriger Laufzeit und 28 Partnern aus Industrie und Wissenschaft mit dem Ziel ein neues UsiXML zu spezifizieren und mit Werkzeugunterstützung zu versehen. Dieses ITEA[18]-Projekt definiert sich selbst folgendermaßen [Usib]:

> „The UsiXML project develops an innovative model driven language to improve the UI design, for the benefit of both industrial end-users actors in term of productivity & reusability; usability & accessibility by supporting the „μ7“ concept: multi-device, multi-user multiculturality/linguality, multi-organisation, multicontext, multi-modality and multi-platform.“

Über die endgültige Ausgestaltung des überarbeiteten UsiXML kann zum gegenwärtigen Zeitpunkt noch keine Aussage getroffen werden. Wenn das Projektziel erreicht und ein stabiles Fundament für MB-UID geschaffen wird, könnte UsiXML zu einem gewichtigen Standard der nächsten Jahre oder Jahrzehnte werden.

XIML

XIML ist die Fortführung des bereits vorgestellten MECANO-Ansatzes. Begonnen hat XIML 1998 als XML-Variante von MIMIC, der ursprünglichen MECANO-Modellierungssprache. Es zählt damit zu den Vorreitern der XML-Verwendung bei der MB-UID.

Gemäß der MECANO-Idee definiert XIML nur die Schnittstellen der Mensch-Maschine-Kommunikation. Zusammenfassend gesagt: Welche Aktionen und Informationen einem Nutzer bereitgestellt werden, welche Domänenobjekte dabei betroffen sind und welche Interaktionskomponenten prinzipiell zur Verfügung stehen. XIML ist jedoch keine UIDL. Dies hat insbesondere zur Folge das damit keine CUI oder FUI-Instanzen beschrieben werden können. Modellinstanzen benötigen immer eine Laufzeitumgebung die den Modellzusammenhang interpretiert und eine geeignetes Interface erstellt.

Soweit bekannt wird an XIML weiterhin gearbeitet und eine Standardisierung angestrebt. Aufgrund der kommerziellen Interessen der, hinter XIML stehenden, Firma „RedWhale Inc“, ist die Verwendung von XIML bisher nur eingeschränkt möglich. Nach Aussage von Angel Puerta, im Rahmen seiner Präsentation von [PH09], sollte im Oktober 2009 die Veröffentlichung von XIML 2.0 erfolgen. Dies ist nicht erfolgt. Zumindest war es bei dieser Gelegenheit möglich einen Blick auf die aktuelle Laufzeitumgebung namens „UI-Fin“zu werfen. UI-FIN wird demnach zur Zeit genutzt, um Software im Umfeld von Krankenhäusern zu erstellen.

Puertas bereits zitierte Anforderung, siehe Punkt 1.2.3.4, dass MB-UID zwingend ein gut ausgearbeitetes und vielseitiges Metamodell für CUI bzw. FUI benötigt, gilt natürlich auch für seine eigenen Arbeiten. Hier kam er letztlich zu dem Schluss, dass die Entwicklung, Fortführung und Wartung eines solchen Metamodells und besonders auch der zugehörigen Renderer die Kapazitäten einzelner Forschungsgruppen oder mittelständischer Firmen überfordern.

Seine Lösung besteht darin, die Laufzeitumgebung nicht mehr als Renderer und Applikationsumgebung zu verwenden, sondern als Generator für Interfacedeklarationen etablierter XML-UIDLs. RedWhale Inc. setzt

[18] Information Technology for European Advancement

zur Zeit auf Microsoft, daher erstellt UI-FIN aus XIML-Instanzen XAML-Benutzungsoberflächen. Diese lassen sich in .Net-Anwendungen integrieren und ansteuern.

XAML und eine Auswahl anderer XML-basierter UIDLs werden im nachfolgenden Punkt 1.3.1.2 erklärt. Prinzipiell ließe sich wohl jede dieser UIDLs in der aktuellen Ausprägung des XIML-Ansatzes als CUI-Modell verwenden.

1.3.1.2 User Interface Definition Languages

In diesem Unterkapitel werden XML-Sprachen vorgestellt die primär auf die Beschreibung von realen Nutzerschnittstellen ausgerichtet sind. Es existiert eine Vielzahl solcher Sprachen. Die schon erwähnte Übersicht [Usic] listet neun veröffentlichte UIDLs[19] auf, für die nachgewiesenermaßen auch Renderer verfügbar sind. Darüberhinaus existieren weitere XML-UIDLs die in [Usic] nicht erwähnt werden. Beispielhaft sei SwiXML [Pau] genannt, eine Möglichkeit Java Swing UIs per XML-Dialekt zu deklarieren.

Urahn der Forschungssprachen zur deklarativen Oberflächenbeschreibung mit XML ist das in Unterabschnitt A.4.2 erläuterte UIML, diese Sprache konnte sich letztlich jedoch nicht durchsetzen.

Von überragender Bedeutung für diese Arbeit ist XUL – die XML User Interface Language von der Mozilla Corporation. Mit dieser Sprache werden unter anderem die Benutzungsoberflächen der sogenannten Web-Suite SeaMonkey und des derzeit populären Browsers Firefox beschrieben. XUL und die damit gesammelten Erfahrungen werden, im Rahmen der Vorstellung der Forschungsergebnisse, in Punkt 2.1.4.1 detaillierter vorgestellt.

XAML

Microsofts XAML [Mic04] ist eine Möglichkeit innerhalb des .Net-Frameworks Benutzungsoberflächen der Windows Presentation Foundation (WPF) XML-basiert zu deklarieren. Das .Net-Framework ist in der Lage zur Laufzeit einer Anwendung XAML-Deklarationen nachzuladen und darzustellen und darüberhinaus ein Data Binding zu aktuellen Instanzen des Domänenmodells herzustellen. Mit XAML lassen sich daher lauffähige funktionale WIMP-Oberflächen beschreiben wobei die Anwendungslogik über eine beliebige .Net-Sprache implementiert werden kann. Mit XAML und der .Net-Integration verfolgt Microsoft ein sehr interessantes Konzept. Es dürfte lohnenswert sein, bei künftigen MD-UID Entwicklungen XAML zu berücksichtigen.

MXML

Adobes MXML [Ado] vervollständigt das Trio der industriell gepflegten und mit einem verbreiteten Renderer ausgestatteten XML UIDLs. Veröffentlicht wurde es 2004 von der damals noch eigenständigen Firma Macromedia. Normalerweise werden MXML-Oberflächen in das Binärformat SWF[20] kompiliert und dann durch Flash-Player gerendert.

In MXML-Dokumente können komplette Applikationen deklariert werden. Die Anwendungslogik wird dabei durch eingebettetes ActionScript bereitgestellt. Ansonsten lassen sich die wichtigen Aspekte einer CUI festlegen; die verwendeten Widgets, deren Aussehen und Anordnung, sowie das Data-Binding an Objekte des Domänenmodells als Datenquellen.

Die aus meiner Sicht gravierendste Einschränkung von MXML ist die Tatsache, dass scheinbar kein interpretierender Renderer zur Verfügung steht und somit immer eine Kompilierung in das Flash-Format

[19] VRML,X3D,MXML,XAML,XUL,VOICEXML,GIML,SUNML,UIML
[20] Shockwave Flash

erforderlich ist. Dies erschwert die Einbettung in andere Applikationen, was im Rahmen der MD-UID wünschenswert wäre, erheblich.

Sonstige WIMP-UIDLs

Eine weitere XML-UIDL ist WML - die Wireless Markup Language [W3C01]. WML entstammt einer Zeit in der die Fähigkeiten mobiler Endgeräte noch nicht weit genug entwickelt waren um übliche HTML-Webseiten darzustellen. Für diese Endgeräte wurde daher vom W3C eine eigene Sprache mit reduziertem Funktionsumfang spezifiziert, spezielle Rücksicht wurde auf kleine Displays, geringen Speicherausbau und schmalbandige Anbindung genommen. Dabei ist WML weder eine Untermenge von HTML noch von XHTML.

Zentrales Konzept WMLs ist die Kartenmetapher, eine Webseite wird jeweils komplett als Aggregation einzelner Karten, auch als Kartendeck bezeichnet, übertragen. Auf dem Endgerät wird immer genau eine Karte angezeigt; jede Karte kann Text, Steuerelemente oder auch multimediale Elemente enthalten. Das Navigieren auf einer Webseite entspricht dann dem Austauschen der jeweils angezeigten Karte.

Ein anderes Beispiel ist die Forschungssprache GIML [Kos06], die im Rahmen einer Promotion an der TU Dresden entstand. Konzeptioneller Rahmen ist die Erstellung eines generischen Interface Toolkits, GITK genannt, für WIMP-Oberflächen. GITK wird als Meta-Framework bezeichnet, als ein Wrapper um andere, dadurch austauschbar werdende, Toolkits wie etwa GTK oder QT.

GIML übernimmt im GITK-Framework die Deklaration der Oberflächenkomponenten. Die Sprache wurde, nach Betrachtung existierender XML UIDLs, von Grund auf neu entworfen. Zur Begründung der Notwendigkeit einer neuen Sprache führt der Autor Stefan Kost bei [Kos06] an, dass vorhandene UIDLs zu viele Sprachelemente beinhalten, welche inkonsistent benannt sind und ein unausgewogenes Verhältnis von Element-Tags und Attributen aufweisen.

Diese Kritik teile ich nur bedingt. Sie scheint ungerechtfertigt, insbesondere da er selbst das Sprachdesign von GIML als „demand driven“ bezeichnet; dies meint, dass nach und nach genau solche UI-Elemente in GIML eingeführt werden welche er selbst benutzt. Diese Vorgehensweise führt geradezu mit Sicherheit zu Inkonsistenzen und läuft aus meiner Sicht auch dem Anspruch einer generischen Sprache zuwider. Die Entwicklung von GITK und GIML im Besonderen scheint mit Abschluss der Dissertation eingestellt.

SwiXML [Pau], ToolkitModel [Foub] und XWT [Asse] sind drei gleichartige XML-UIDLs die ich an dieser Stelle noch erwähnen möchte. Bei XML-Dokumenten in diesen Sprachen handelt es sich eher um XML-Serialisierungen von Oberflächendeklarationen mit bekannten objektorientierten Frameworks, denn um eigenständige UI-Modellierungssprachen. Die Nutzung dieser Sprache erfolgt prinzipiell ähnlich dem bereits beschriebenen XAML.

Statt Java-Quellcode für Swing-Oberflächen zu implementieren, kann mit SwiXML die Oberfläche als XML-Dokument deklariert werden und dieses zur Laufzeit in Swing-Objekte und damit einer Benutzungsoberfläche umgesetzt werden. Ähnliches gilt für XWT, dem XML Windowing Toolkit, jedoch wird hier SWT genutzt. Das sogenannte ToolkitModel ist im Moment Bestandteil von Eclipse E4, es abstrahiert vom konkreten Toolkit und unterstützt dadurch die Generierung sowohl von Swing als auch SWT-Applikationen; das ToolkitModel ist gewissermaßen eine Fortführung der Ideen aus dem eingestellten AUIML [AUI]-Projekt von IBM.

Nicht-WIMP UIDLs

Selbstverständlich existieren XML-Deklarationssprachen auch speziell für User Interfaces die andere Interaktionsmetaphern bzw. Modalitäten verfolgen. Da sie sich damit zumeist außerhalb des WIMP-Bereichs bewegen haben diese Sprachen mit dem Themenbereich dieser Arbeit nur am Rande zu tun.

SVG - Scalable Vector Graphics - ist ein Beispiel für eine solche Sprache. Standardisiert [SVG] durch das W3C, wird SVG genutzt um zweidimensionale Vektorgrafiken und Animationen solcher Grafiken zu beschreiben. Prinzipiell handelt es sich um eine unidirektionale Visualisierungsschnittstelle. Nutzerinteraktionen sind seitens SVG nicht definierbar. Allerdings kann SVG wegen seiner XML-Abstammung leicht in andere XML-UIDLs integriert werden. Beispielsweise haben wir SVG in XUL-UI-Deklarationen [WFR05] eingebettet. Durch diese Herangehensweise lassen sich Nutzerinteraktionen auf SVG-Oberflächenbeschreibungen steuern.

Eine gänzlich andere Modalität unterstützt VoiceXML [Voi] - die Voice Extensible Markup Language - ebenfalls durch das W3C standardisiert. Gemäß des Abstract in [Voi] ist VoiceXML eine modulare Sprache zur Erstellung interaktiver Dialoge welche Sprachsynthese, Sprach- und Tonwahlerkennung oder sonstige digitale Audio- und Videodaten verwenden. Eine regelmäßige Anwendung von VoiceXML ist, unter dem Namen „X+V“, die Einbettung von VoiceXML-Konstrukten in XHTML. Paternò und andere an TERESA Beteiligte haben sich unter anderem in [BP05] mit der Benutzung von X+V im modellgetriebenen Kontext beschäftigt.

Als letztes Beispiel soll hier noch VRIXML [CRC04] - Virtual Reality Interaction XML - genannt werden. Deren Einsatzkontext sind 3-dimensionale virtuelle Umgebungen. Soweit aus den Veröffentlichungen ersichtlich kommen bei VRIXML letztlich typische WIMP-Widgets zum Einsatz, ergänzt um Positionierungs- und Ausrichtungsangaben in einer 3D-Welt. Das Prinzip ist jedoch offen für Erweiterungen, wie etwa spezialisierte 3-dimensionale Widgets.

Andere Aspekte von XML-Technologien bei UIDLs

Eine Reihe bekannterer Ansätze benutzt XML-Technologien um aus eigenen Deklarationen Oberflächenbeschreibungen mit den beschriebenen XML-UIDLs zu erzeugen. useML [Reu03] aus dem Useware-Ansatz zählt in diese Kategorie. In der ursprünglichen Fassung [Reu03] von 2003 handelt es sich um ein Beschreibungsmittel für betriebliche Abläufe. Es wird der Zusammenhang zwischen Arbeitsmitteln und Werkzeugen im Produktionsprozess XML-basiert beschrieben. Wofür Konzepte wie Funktionen, Nutzer, Vorgabewerte und -belegungen und ähnliches genutzt werden. Mittels XSL-Transformationen, siehe Punkt 2.2.1.1, werden diese useML-Deklarationen später in diverse FUIs überführt. Das bei [use] veröffentlichte Beispiel enthält Transformationen für verschiedene webbasierte Endgeräte; WML und verschiedene HTML-Ausprägungen, diese differieren je nach Fähigkeiten des Ziel-Endgerätes. Erwähnenswert an dieser Version von useML ist, dass die Sprache deutschsprachige XML-Tags benutzt, überwiegend werden englischsprachige Bezeichner gewählt. An useML, und generell an dem Useware-Ansatz für die intelligente Fabrik, wird weiterhin gearbeitet. In neueren Sprachversionen, ersichtlich in Veröffentlichungen wie [BMZ05], erfolgte eine Umstellung auf einen mehrstufigen, in CAMELEON einordbaren, Prozess.

Die spanische Universität La Mancha stellte 2009 [Rea09] einen eigenen Ansatz zum besseren Umgang mit UsiXML, und letztlich allen anderen XML-UIDLs, vor. Sie identifizierten in der bisherigen Verwendung dieser Sprache einen Mangel an Anpassbarkeit der engültigen Oberfläche. Ihr Ansatz ist, dass sich Benutzungsoberflächen im Laufe der Zeit schrittweise verändern, eine parallele Weiterentwicklung und -benutzung des ursprünglichen UsiXML-Dokumentes jedoch so nicht vorgesehen ist.

Diese Lücke versucht T:XML zu füllen, es handelt sich um ein Werkzeug, das mit einem Metamodell für

Modelltransformationen arbeitet. Genau wie die Transformationsregeln innerhalb UsiXML, werden die schrittweisen Anpassungen über Graphentransformationen definiert und dann in Instanzen des T:XML-Metamodells abgelegt. Aus diesen Instanzen wird XSLT, oder später womöglich QVT, generiert; mit diesen Techniken wird dann die eigentliche Transformation des UsiXML-Ausgangsmodells vorgenommen. Bei der Vorgehensweise nach T:XML handelt es sich um eine einigermaßen spezielle Lösung des Problems der Modellversionierung. Der Ansatz ist für Spezialprobleme der Anpassung an diverse Nutzungskontexte sicherlich nützlich und die Entwicklung eines vereinheitlichten Metamodells für Modelltransformationen wäre auch sehr wertvoll, diese Arbeiten stehen aber zur Zeit noch am Anfang.

1.3.2 Schlußfolgerung

Die Modellierung und Deklaration von Benutzungsschnittstellen ist eine seit mehr als 12 Jahren erprobte Technik. Aus den Anfängen XIML, UIML und XUL hat sich eine Vielzahl, teils erfolgreicher und oft genutzter, Sprachen mit interessanten und weitgespannten Einsatzbereichen entwickelt. Darüberhinaus hat XML seine Eignung auch für User Interfaces bewiesen die nicht das WIMP-Paradigma einsetzen.

Wegen ihres deklarativen Charakters eigenen sich die XML-UIDLs besonders zur Verwendung in einem modellbasierten oder -getriebenen Entwicklungsprozess. Einige vorgestellte Frameworks und Systeme die XML-Technologien auch für höhere Abstraktionsebenen anwenden unterstreichen dies. Im nachfolgenden Kapitel wird der an der Universität Rostock verfolgte Prozess detailliert vorgestellt. Darin spielt die XML-UIDL XUL eine wichtige Rolle, eingesetzt als Haupt-Deklarationssprache für konkrete Oberflächen.

Kapitel 2

Modellgetriebene Generierung von Benutzungsoberflächen

In diesem Kapitel werden Metamodelle und deren Einsatz in einem konkreten Vorgehens- oder Prozessmodell zur modellgetriebenen Erzeugung von User Interfaces erläutert. Dies ist Gegenstand des nachfolgenden Abschnittes. In den daran anschließenden Abschnitten werden die Grenzen des Ansatzes diskutiert sowie die eingesetzten Techniken kurz dargestellt.

2.1 Beschreibung des Rostocker Prozessmodells

In diesem Abschnitt werden die grundsätzlichen Möglichkeiten der Generierung von Nutzeroberflächen auf der Grundlage von Modellen diskutiert. Den Schwerpunkt dieser Betrachtungen werden Modelle bilden, die zur Deklaration typischer WIMP-Oberflächen dienen. Inwieweit diese Beschränkung möglicherweise die allgemeine Verwendbarkeit des präsentierten Gesamtprozesses herabsetzt, wird in Unterabschnitt 5.2.1 erörtert.

Automatisch oder Semi-Automatisch

Ein immer wieder hervortretender Problemkreis bei modellbasierten und modellgetriebenen Konzepten ist die Frage nach dem Grad der Automatisierung des Entwicklungsprozesses. Es gibt, zumindest im Bereich der UI-Modelle, keine allgemein akzeptierte Fixierung von Automatisierungsgraden. Dennoch lassen sich drei Schwerpunkte identifizieren:

manuell: Sämtliche Transformationen und Entscheidungen werden durch menschliche Designer vorgenommen. Diese Situation wird nicht weiter betrachtet.

semi-automatisch: Injektive und surjektive Transformationen werden automatisiert vorgenommen. Entscheidungen bei Mehrdeutigkeiten sind Prozessbestandteil und werden durch menschliche Designer vorgenommen.

automatisch: Wie semi-automatisch, jedoch werden notwendige Auswahlentscheidungen durch Heuristiken, Constraints oder sonstige Entscheidungsmodelle getroffen.

Neben der Betrachtung der reinen Transformationen könnte, zur Unterscheidung zwischen automatisiert und semi-automatisiert, auch noch die Frage der Durchführung einer Endaufbereitung herangezogen werden. Damit ist gemeint, inwieweit die Resultate der Transformationen vor ihrer Verwendung, einer Nachbearbeitung durch menschliche Designer unterzogen werden sollen. Dazu ist jedoch zu bemerken, dass bei allen bekannteren Techniken eine wie auch immer geartete nachträgliche Anpassung des erzielten Transformationsergebnisses vornehmbar ist. Dies degradiert das Kriterium der Nachbearbeitbarkeit zu einem Aspekt mit nurmehr sehr schwacher Unterscheidungswirkung. Aus diesem Grund wurde es bei der obigen Charakterisierung der drei Schwerpunkte ausgelassen.

Der überwiegende Teil der Forschungen zu MDD-UI geht davon aus, dass semi-automatische Transformationsprozesse die beste Alternative darstellen. Bereits in einer Veröffentlichung [SLN92] von 1992 schreiben Luo, et. al:

> „Most difficult design decisions are best left to the human designer, as they require knowledge which is more easily and quickly modelled by humans than by a system. E.g. knowledge about end-users and application domain."

Demnach sollen die schwierigsten Entscheidungen im Design einer Anwendung menschlichen Designern überlassen werden. Begründet wird dies damit, dass Menschen über Wissen über den Anwendungsbereich und die späteren Endnutzer verfügen, welches in einem Modellsystem nur schwierig und mit hohem Zeitaufwand beschreibbar ist.

Ansätze zur Vollautomatisierung finden sich beispielsweise bei TEALLACH (siehe Unterabschnitt A.2.2), aber auch bei den Enabling Task Sets (ETS) im Rahmen der CTTE. David Paquettes Veröffentlichung [PS06] stellt Techniken rund um die automatisierte Generierung von Oberflächen aus Aufgabenmodellen dar, ETS spielen in diesem Kontext die wichtige Rolle des Navigationsmodells. Leider ist die Qualität der Ergebnisse vollautomatischer Transformationen derzeit selbst auf niedrigem Niveau als bescheiden zu bezeichnen.

Prozessübersicht

Meine Arbeit ordnet sich in den Kontext des an der Universität Rostock verfolgten und in Abbildung 2.1 gezeigten Ansatz zur Erstellung von Benutzungsoberflächen ein. Es handelt sich hierbei um einen hybriden Ansatz, welcher auf den Arbeiten zum TADEUS-Projekt aufbaut, siehe Unterabschnitt A.3.4. Neben der Betrachtung der zu erzeugenden Benutzungsschnittstelle werden ebenfalls Transformationen von Modellen zur Generierung und Erstellung der Anwendungslogik im Prozess integriert. Klassifiziert werden kann dieses Vorgehensmodell als semi-automatisches modellgetriebenes Prozessmodell. Die Entscheidungen, welche menschliche Designer im Rahmen dieses Modells zu treffen haben sind vielfältig. Bei den Erläuterungen zu den einzelnen Sub-Modellen wird auf den Einfluss des menschlichen Faktors eingegangen.

Im Rostocker-Ansatz werden vier Quellmodelle betrachtet. Im Zentrum steht das Aufgabenmodell. Hier erfolgt eine Zerlegung des Anwendungsziels in atomare Teilaufgaben. Über die Definition temporaler Abhängigkeiten zwischen diesen Teilaufgaben, werden auf abstrakter Ebene die möglichen Abläufe in der späteren Anwendung modelliert. Relevante Charakterisierungen der späteren Endbenutzer werden im Nutzermodel vorgenommen. Der Kontext in welchem die Anwendung später eingesetzt ist, wird über das Domänenmodell spezifiziert. Alle Artefakte die durch die Anwendung benutzt oder modifiziert werden definiert das Objektmodell der Geschäftslogik. Den inneren Zusammenhang zwischen diesen vier Quellmodellen stellt das Aufgabenmodell, über Referenzen, bereit.

Transformationen der Quellmodelle führen über Zwischenmodelle zu zwei Ergebnismodellen. Dies sind

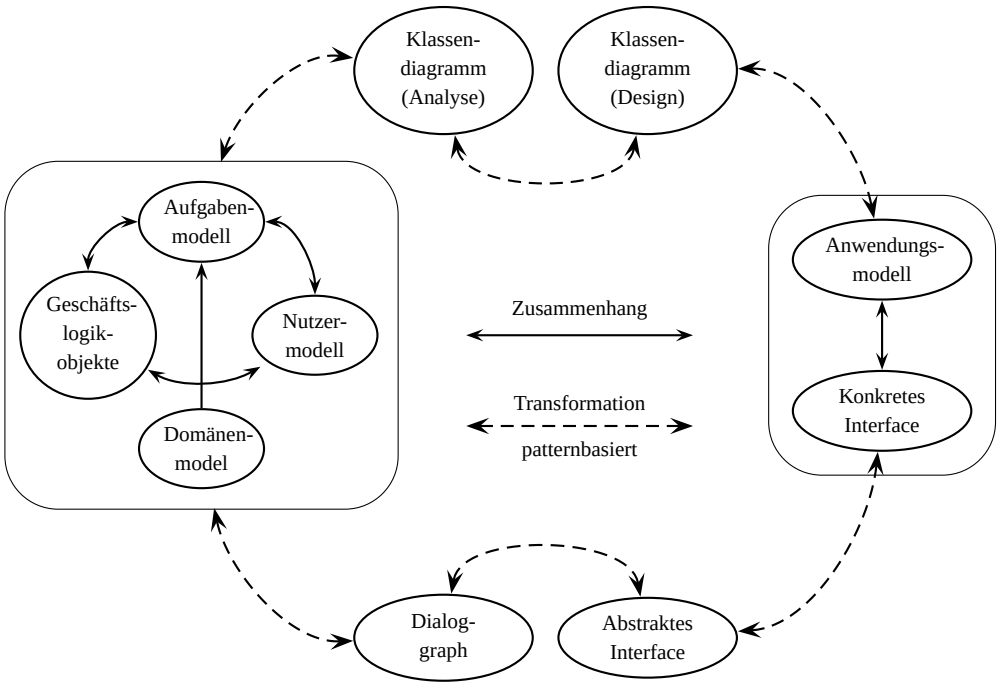

Abbildung 2.1: Transformationsorientierter MDD-Entwicklungsprozess

das Anwendungsmodell und das Modell des konkreten Interfaces. Durch Kombination beider Modelle soll schließlich mittels geeigneter Model-To-Text Transformationen ausführbarer oder interpretierter Quellcode erzeugt werden.

Nun besteht zwischen den Quellmodellen und lauffähigem Quellcode ein erhebliches Abstraktionsgefälle. Sämtliche auf den höheren Abstraktionsebenen nicht betrachteten Informationen müssen daher im Verlauf des Transformationsprozesses hinzugefügt werden. Diesem Zweck dienen die Zwischenmodelle.

Die Modellierung des Anwendungsmodells soll dabei ausgehend von den Quellmodellen über die Schritte Analysemodell und Designmodell erfolgen. Bei dem Analysemodell handelt es sich um ein Klassenmodell, welches die Erkenntnisse aus Nutzer-, Geschäftsbjekt- und Domänenmodell zu einem Klassenmodell kondensiert. Das Designmodell ist im gewissen Sinne die Konsolidierung des Analysemodells in Hinblick auf Umsetzbarkeit. Durch Transformation des Designmodells wird das Anwendungsmodell erzeugt. Hinter diesem Transformationspfad stehen also Standardvorgehensweisen der Softwaretechnik zur Ableitung von Anwendungsmodellen aus den Ergebnissen des Requirements Engineering. Einzelheiten dazu werden im Rahmen dieser Arbeit nicht betrachtet.

Stattdessen soll der untere Transformationspfad ausgearbeitet werden. Die Modell-Sequenz lautet hier: Quellmodelle →Dialoggraph→Abstrakte Oberfläche→Konkrete Oberfläche. Die möglichen Hindernisse auf dem Weg vom Aufgabenmodell zu lauffähigem Quellcode sind manigfaltig. Das zu überwindende Problem kann als Versuch zusammengefasst werden, eine Zeichenkette schrittweise zu einem semantisch passenden ausgestalteten Oberflächenelement zu transformieren.

Diese Ausgangszeichenkette ist der Name der jeweiligen Teilaufgabe, der eine grafische Darstellung zuzuordnen ist. Beispielsweise sollte eine Aufgabenbeschreibung wie „Email lesen“ dazu führen, dass letztlich eine Email-Auswahl erfolgen kann und deren Textteil formatiert angezeigt wird. Die hierfür erforderlichen Konzepte, dazu Dinge wie Layout und Design, müssen im Laufe der einzelnen Transformationsschritte

nach und nach der initialen Aussage hinzugefügt werden. Dies soll über die Zwischenmodelle, erweiternde Transformationen und insbesondere über die Entscheidungen und Gestaltungen der menschlichen Designer umgesetzt werden.

Daraus ergibt sich die Frage, welche Informationen konkret in den Zwischenschritten hinzugefügt werden müssen. Eine WIMP-Benutzungsoberfläche wird durch mehrere Merkmale charakterisiert:

1. der Auswahl der zur Verfügung stehenden **Interaktionselemente**
2. deren grafische Gestaltung, also das **Aussehen**
3. ihre räumliche Anordnung, innerhalb des W$_{\text{indow}}$-Containers, im Allgemeinen als **Layout** bezeichnet
4. sowie eventuell bestehende dynamische Abhängigkeiten, die eine **Navigation** begründen

Keines der Quellmodelle liefert substantielle Informationen zu auch nur einem Aspekt oder deckt ihn anderweitig adäquat ab. Es sind daher weitere Modellspezifikationen notwendig. Das in der Bearbeitungssequenz von Abbildung 2.1 erste Folgemodell, der Dialoggraph, dient der Definition der Grundstruktur des späteren Interfaces. Durch Festlegung von Sichten, im Endeffekt der Zuordnung von Aufgaben zu Fenstern, und den Übergängen zwischen Sichten wird die Navigation (4) vorgezeichnet und in gewisser Weise eine Vorauswahl zu den Interaktionselementen (1) vorgenommen.

Auf dieser Grundlage wird ein Modell für die abstrakte Oberfläche (AUI) gewonnen. Innerhalb dieses Modells findet eine Fixierung der Visualisierung jeder Aufgabe auf ein Element einer Interaktionskategorie (1) statt, beispielsweise wird die Entscheidung für eine Schaltfläche oder eine Tabelle getroffen. Je nach verwendetem Metamodell kann bereits auf der Ebene der abstrakten Oberfläche eine Gruppierung von Elementen vorgenommen werden und es können ggf. Richtlinien zur Elementanordnung innerhalb solcher Gruppen vorgegeben werden. In diesem Fall wird bereits auf der Ebene des AUI ein Layout (3) vorgezeichnet.

Die nächste Verfeinerungsebene ist das Modell der konkreten Oberfläche. Hier erfolgt das Mapping der abstrakten Interaktionsobjekte auf konkrete Widgets eines Zielmodells. Darüberhinaus wird das Design (2) jedes Widgets in diesem Modell beschrieben sowie das finale Layout (3) für die Benutzungsoberfläche der Anwendung festgelegt. Das Modell der konkreten Oberfläche stellt dann, in Kombination mit dem Anwendungsmodell, das Quellmodell für eine Model-To-Text Transformation zur Erzeugung des Quellcodes der Anwendung dar.

Das Vorgehensmodell nach Abbildung 2.1 sieht für die Modellabfolge bidirektionale Beziehungen vor. Modell-Transformationen sind demnach in beide Richtungen möglich, sowohl abstrahierend als auch konkretisierend. Ziel des Ansatzes ist es für alle Transformationsschritte Pattern zu identifizieren und diese dann mit Hilfe einer geeigneten Pattern Language, siehe dazu auch Abschnitt 3.5, Benutzern der Methodik zur semi-automatischen Anwendung bereitzustellen.

2.1.1 Quellmodelle des modellgetriebenen Ansatzes

Die in der Einführung angerissenen Quellmodelle werden im Folgenden im Detail vorgestellt. Hierzu werden jeweils die verwendeten Metamodelle vorgestellt und erklärt.

Diese Metamodelle sind in einem Systemkontext eingebunden welcher in den abgebildeten Klassendiagrammen auf zwei verschiedene Arten angedeutet wird. Die Beziehungen zum Kontext werden durch die Klassen Model und ModelElement hergestellt. Die Klasse Model dient dabei als Wurzelklasse aller Model-

le. ModelElement markiert diejenigen Typen, die die Struktur der Metamodellinstanzen aufbauen. Anders formuliert, diejenigen die im System direkt durch Editoren bearbeitet werden können.

2.1.1.1 Metamodell für Aufgabenmodelle

Die Aufgabenmodellierung ist eine etablierte Technik der Anforderungsanalyse. Bereits in Punkt 1.2.3.3 wurden diverse Systeme vorgestellt, welche Aufgabenmodelle als Ausgangspunkt einer modellbasierten Oberflächengenerierung verwenden. Der Punkt 1.3.1.1 gibt einen Überblick über weiter verbreitete Ansätze zur Aufgabenmodellierung.

Abbildung 2.2 zeigt das Metamodell für das Aufgabenmodell in unserem modellgetriebenen Ansatz. Wir verstehen es als eine Evolution des Metamodells der ConCurTaskTrees.

Eine Aufgabe wird als Instanz der Klasse Task angelegt. Für jeden Task können beliebig viele Vorbedingungen als Precondition definiert werden. Ebenso ist die Anzahl der Effects nicht beschränkt, hier sollen die Auswirkungen der Aufgabenabarbeitung notiert werden. Auswirkungen und Vorbedingungen werden durch Ausdrücke in einer nicht im Metamodell spezifizierten Sprache angegeben. Der referenzierte Typ Expression fungiert als Containertyp für Assoziationen zu beliebigen konkreten Sprachen.

Die Aufzählung TaskCategory überträgt die vier Aufgabenkategorien des CTT als benannte einfache Werte in das Metamodell. Verwendet wird dies im category-Attribut einer jeden Aufgabe. TemporalOperator ist eine Aufzählungsklasse und definiert Literale für die von CTT definierten temporalen Operatoren. Der Wert notspecified wurde zusätzlich aufgenommen, um als Standardwert für den Operationstyp von Instanzen der TemporalOperation-Klasse zu dienen. Dabei handelt es sich um ein Zugeständnis an die Usability. Der Wert notspecified wirkt als Marker für eine ungültige Modellinstanz und zwingt Designer explizit zum Setzen eines gültigen temporalen Operators.

Jedes TaskModel, welches die Verwaltungsklasse für das jeweilige Aufgabenmodell ist, enthält die Referenzen zu den anderen Quellmodellen, außerdem noch genau eine Referenz zu dem Wurzelknoten des Aufgabenbaums.

Im Gegensatz zu CTT werden in unserem Ansatz die Operatoren als Knoten in die Baumstruktur integriert. Operatorknoten dürfen wiederum Operatorknoten als Kinder haben, das Verschachteln temporaler Operatoren ist somit zugelassen. Daher kann die Knotenabfolge als abstrakter Syntaxbaum eines, darüber definierten, Prozessalgebra-Terms angesehen werden. Das Beispiel in Abbildung 2.3 illustriert diese Struktur.

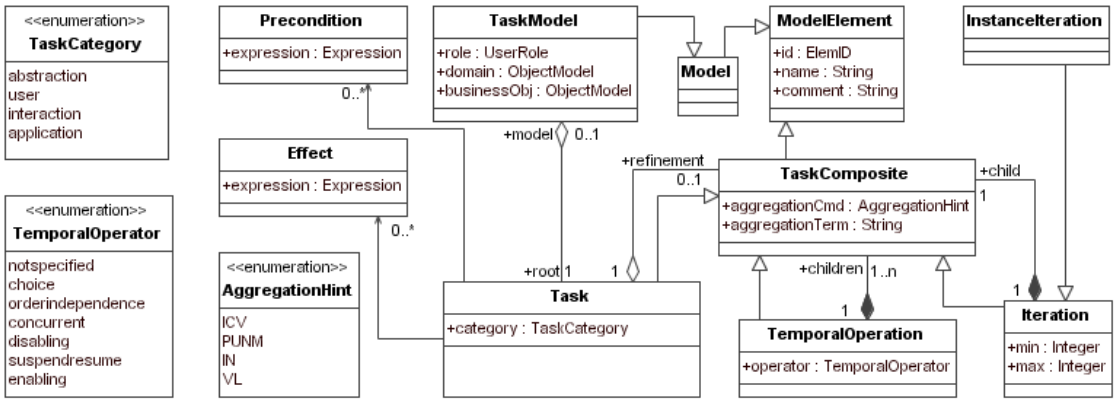

Abbildung 2.2: Metamodell für das Aufgabenmodell nach [WF09]

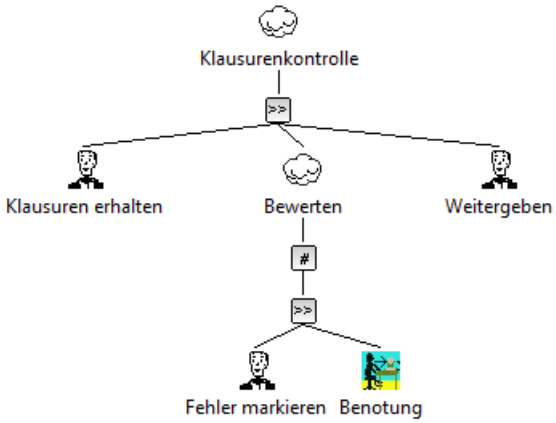

Abbildung 2.3: Aufgabenmodell für die Klausurenkontrolle in der Rostocker Notation

Die Beziehungen zwischen TemporalOperation, Iteration und TaskComposite sind Kompositionen. Dadurch wird erzwungen, dass die Blattknoten des Baumes immer Instanzen des Typs Task, also Aufgabenknoten, sind. Zwar nicht durch Kardinalitäten unterbunden, jedoch per Syntaxregel verboten sind:

- Iterationsknoten als Kind eines Iterationsknoten
- Syntaxverletzungen der CTT-Notation, insbesondere die Anforderungen an die Knotenkategorisierung

Warum existiert ein spezieller Knotentyp für Iterationen? Aus zwei Gründen, bei der Iteration handelt es sich weder um einen Operator noch um eine Aufgabe. Es handelt sich um die Aussage, dass der Teilbaum unter der iterierten Aufgabe beliebig oft wiederholt werden kann. Ein Iteratorknoten unterscheidet sich zusätzlich von einem Operatorknoten darin, dass er genau einen Kindknoten hat. Nämlich die Wurzel des zu wiederholenden Teilbaumes. Da zusätzlich die Angabe eines Intervalls für die Anzahl der Wiederholungen ermöglicht werden sollte, schien ein eigener Typ gerechtfertigt.

Die Originalbedeutung der Iteration im CTT-Ansatz ist, dass eine iterierter Aufgabe nur nach deren Beendigung neu gestartet werden kann. Es ist bei CTT nicht möglich, verschränkte Iterationen zu spezifizieren. Das kann für Aufgaben nützlich sein, die wiederholt gestartet werden sollen, obwohl die vorherige Iteration noch nicht beendet wurde. In der Rostocker Methodik wird der Begriff `Instanziteration` dafür verwendet. Im Metamodell wird dieser Spezialfall eines Iterationsknoten als Unterklasse InstanceIteration der allgemeinen Iteration reflektiert.

Ein Beispielszenario für Instanziteration bietet die Kontrolle von Klausuraufgaben. Die oft eingesetzte Strategie ist, diese in Eingangsreihenfolge abzuarbeiten. Schwierige Fälle jedoch, etwa Abweichungen von der Standardvorgehensweise oder Bewertungen bei denen es auf einzelne Punkte ankommt, werden gesondert behandelt. Deren endgültige Benotung wird herausgezögert und stattdessen die Benotung andere Klausuren vorgezogen. Die angefangene Kontrolle der Problemarbeiten wird zu einem späteren Zeitpunkt, aber noch innerhalb der gleichen Aufgabe, fortgesetzt und beendet. Durch Verwendung von Instanziterationen kann dieses Szenario leicht modelliert werden.

Die Abbildung 2.3 zeigt ein entsprechendes Aufgabenmodell. Der Instanzoperatorknoten findet sich als Kindknoten der abstrakten Aufgabe *Bewerten*, markiert mit dem Operatorsymbol #. Alle weiteren grafischen Elemente sind in ihrer Semantik äquivalent denen der CTT-Notation: Die Aufgabe Klausurenkontrolle wird in vier atomare Aufgaben zerlegt. Am Anfang steht die Entgegennahme der Klausuren, modelliert als die Nutzeraufgabe *Klausuren erhalten*. Nachdem diese Aufgabe abgeschlossen ist, wird der

Teilaufgabenbaum unterhalb der abstrakten Aufgabe *Bewerten* aktiviert. Die Aufgaben *Fehler markieren* und *Benotung* sind sequentiell abzuarbeiten. Diese Sequenz kann bereits vor deren Beendigung im Rahmen einer neuen Instanz neu gestartet werden. Die nachfolgende Aufgabe *Weitergeben* kann nur gestartet werden, nachdem alle Iterationsaufgaben abgeschlossen sind. Das Starten der Folgeaufgabe deaktiviert in jedem Fall die Iterationen.

Der in den bisherigen Erläuterungen nicht behandelte Typ AggregationHint ist insbesondere bei der Erstellung des Dialoggraphen relevant. Er wird daher im Unterabschnitt 2.1.2 erklärt.

2.1.1.2 Metamodell für Nutzermodelle

In diesem Modell werden die Nutzer des späteren Systems beschrieben. Diese Beschreibung sollte nur die Merkmale umfassen, die für die Interaktion der Nutzer mit der Anwendung von Bedeutung sind.

Das Metamodell für Nutzer, dargestellt in der Abbildung 2.4, reflektiert diesen Minimalansatz: Nutzer werden rein über Rollen charakterisiert. Die Rollen selbst werden durch den Typ UserRole umgesetzt. Für diesen Typ sind die von der Klasse ModelElement geerbten Attribute ausreichend.

Das bedeutet, dass Nutzerrollen in diesem Ansatz nicht mehr als bloße Namen sind. Eine Hierarchie oder Rollenzusammenhänge sind auf dieser Modellebene nicht vorgesehen. Falls es im Einzelfall gewünscht sein sollte, eine Rolle mit weiteren Informationen auszustatten, kann hierfür die Aggregation properties genutzt werden. Die Metaklasse Entity entstammt dem allgemeinen Objektmodell der Abbildung 2.5 und wird im Folgenden erläutert.

2.1.1.3 Domänenmodell und Geschäftslogik

Die in Abbildung 2.5 dargestellte Metamodell-Variante ist eine Umsetzung der Basiskonzepte der Objektorientierung. Es kann sowohl zur Beschreibung von Geschäftsobjekten als auch des Nutzungskontexts verwendet werden. Das an dieser Stelle weder das Ecore- noch das UML-Metamodell zum Einsatz kommen hat im Wesentlichen historische und softwarearchitektonische Gründe. In ihrer Ausdrucksfähigkeit und Interoperabilität wären diese genannten Alternativen mächtiger.

Zentrale Metaklasse dieses Modells ist die Entitätsmenge, jedes Objektmodell (ObjectModel) besteht aus Instanzen des Typs Entity. Eine Entitätsmenge lässt sich mit den Attributen des Typs ModelElement (Name, ID und Kommentar) hinreichend genau beschreiben, auf weitere Attribute in der Metaklasse wurde verzichtet. Zur Spezifikation einer Entitätsmenge gehören im OO-Paradigma immer die Eigenschaften der Instanzen, sowie Methoden welche die Instanzen ausführen können.

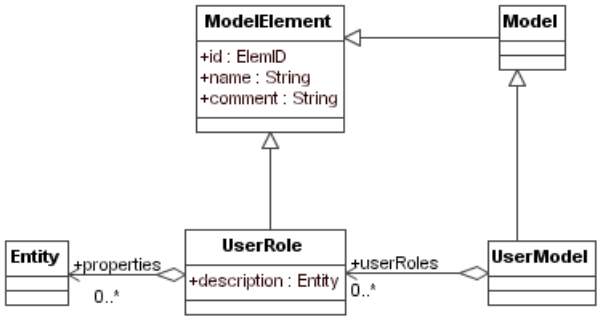

Abbildung 2.4: Metamodell für das Nutzermodell

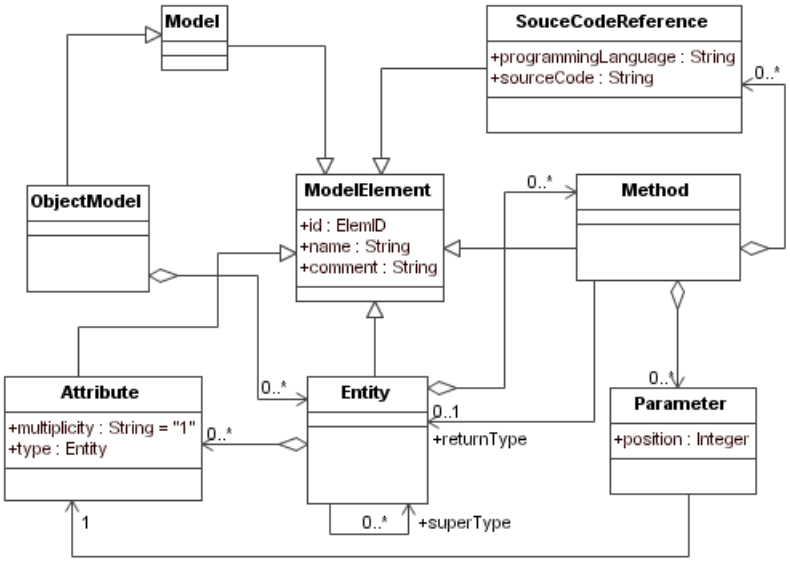

Abbildung 2.5: Allgemeines Objekt-Metamodell

Die Attribute der Instanzen einer bestimmten Entitätsmenge beschreibt die Aggregationsbeziehung zu Attribute. Die Signatur eines Attributes besteht aus seinem Namen und seinem Typ. Der Typ wird bei Anwendung des Metamodells von Abbildung 2.5 als Instanz von Entity spezifiziert. Zur Namensangabe dient das von ModelElement geerbte Attribut. Durch Angabe einer expliziten multiplicity können Vektoren und Optionalität ausgedrückt werden. Die Wertsemantik folgt hier derjenigen der UML, der Standardwert „1“ entspricht einem einfachen nicht-optionalem Attribut.

Methoden werden, über die Klasse Method, grundsätzlich auf die gleiche Weise definiert. Methodensignaturen enthalten typischerweise noch Eingabeparameter. Da für diese Parameter nicht nur der Typ, sondern auch die Reihenfolge bedeutsam ist, wurde die Assoziationsklasse Parameter eingeführt. Deren einziges Attribut position definiert die Indexposition in der Parameter-Folge. Jedem Parameter ist eine Instanz der Klasse Attribute zugeordnet, dies dient der Definiton von Name und Typ des jeweilige Parameters. Optional kann die Angabe des Rückgabetyps einer Methode, mittels der Assoziation zu returnType:Entity, erfolgen.

Das finale Ziel des hier vorgestellten Ansatzes ist es, ausgehend von den Modellen lauffähige Anwendungen zu generieren. Dafür kann es sinnvoll sein, insbesondere für die Methoden der Objekte der Geschäftslogik, Quellcode für die Implementierung vorzugeben. Die Metaklasse SourceCodeReference dient der Deklaration entsprechender Quellcodebestandteile. Für jede Methode können beliebig viele Quellcode-Ausschnitte definiert werden.

Die Abbildung 2.5 sieht Mehrfachvererbung für die Entitätsmengen vor. Die Assoziationsende mit dem Rollennamen superType kann zur Deklaration beliebig vieler Oberklassen genutzt werden. Es sind keine Ansätze zur Lösung oder Verhinderung der typischen Probleme bei Mehrfachvererbung vorgesehen. Das Vermeiden von Namenskonflikten, die Herstellung von Typsicherheit und das Verhindern von zyklischen Vererbungen sind damit dem Designer überlassen.

Zu Illustration findet sich in Abbildung 2.6 eine Darstellung der Anwendung des Metamodells und sowie eines damit gleichbedeutenden UML-Klassendiagramms. In der linken Hälfte findet sich die Darstellung im Standard-Ecore-Editor, rechts das gleichbedeutende UML-Modell. Da es sich um keinen spezialisierten Ecore-Editor handelt, sind Einzelheiten insbesondere Typ und Multiplizität, nur über die Eigenschaftenansicht der einzelnen Typen einseh- und änderbar.

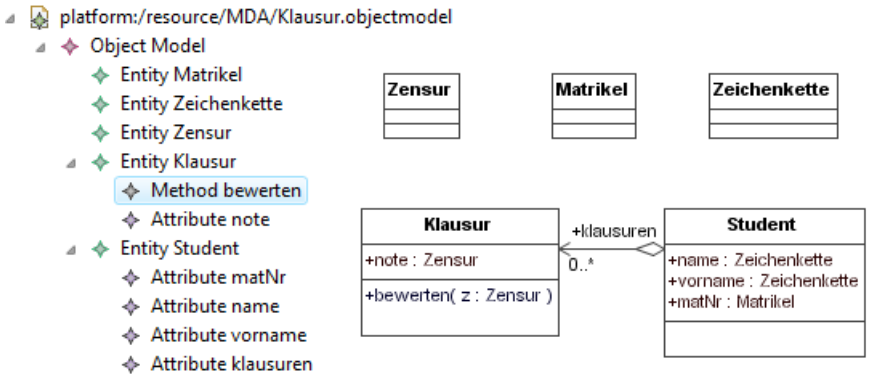

Abbildung 2.6: Gegenüberstellung Objektmodelleditor und UML-Modell

Im Beispiel wurden die Domänen-Objekte einer fiktiven Anwendung zur Unterstützung der Klausurenkontrolle modelliert. Die Entitätsmengen Matrikel, Zeichenkette und Zensur sind auf dieser Abstraktionsebene als reine Datentypen modelliert, es wurden keine Eigenschaften oder Methoden festgelegt. Diese Datentypen werden benötigt, um den Wertebereich der Attribute der beiden Entitätsmengen Klausur und Zensur festzulegen.

Gerätemodell als Spezialfall des Domänenmodells

Ein wichtiger Grund für die Verwendung modellbasierter Techniken ist die vereinfachte Anpassbarkeit an sich ändernde Nutzungsbedingungen. Im Rahmen von Benutzungsoberflächen sind dabei besonders zwei Änderungsdimensionen zu betrachten.

Zum einen die Geräteplattform und zum anderen die Modalität der Nutzerinteraktion. Wird beispielsweise der Interaktionsmodus von einer WIMP-Methodik auf akustisch-taktile Interfaces umgestellt, dürfte im Regelfall ein komplettes Re-Design der Benutzerschnittstelle notwendig sein. Der in dieser Arbeit vorgestellte Ansatz kann in Teilen auch für derlei Interfaces eingesetzt werden. Insbesondere sind die Quellmodelle unabhängig von der eingesetzten Interaktionsmodalität. Die im weiteren Verlauf vorgestellten Modelle für konkrete Oberflächen sind mit Schwerpunkt auf WIMP-Oberflächen entwickelt. Diese sind nur unzureichend für die Beschreibung multimodaler Nutzerinterfaces einsetzbar, weil sie in keiner Weise auf deren Besonderheiten eingehen.

Nutzungskontextänderungen die aus einem Wechsel der Geräteplattform bestehen, können mit den Modellen gut behandelt werden. Hardware-Plattformen werden typischerweise in einem Gerätemodell beschrieben. Es wäre möglich, ein spezielles Metamodell zu deren Beschreibung vorzugeben. Allerdings würde dieses strukturell sehr ähnlich dem des Objektmodells sein, siehe dazu Punkt 2.1.1.2. Statt ein weiteres Metamodell einzuführen, sollte das Gerätemodell unter Benutzung des Objekt-Metamodell aus Abbildung 2.5 spezifiziert werden. Dies ermöglicht flexible, aber dennoch formale, Geräte-Beschreibungen, die in den nachfolgenden Transformationen verwendet werden können.

2.1.2 Dialogmodell

Im Prozessmodell zur modellgetriebenen Oberflächenerzeugung, siehe Abbildung 2.1, folgt auf die Instanziierung der Quellmodelle die Prozessstufe Dialogmodell. Im Dialogmodell wird die Navigationsstruktur der Nutzungsoberfläche beschrieben.

Da wir eine WIMP-Anwendung mit diesem Mittel beschreiben, ist die Basisstruktur des Dialogmodells das Fenster. Die Abstraktionsebene des Dialogmodells sieht jedoch noch keine deratig konkreten Objekte vor, deshalb wird stattdessen der Begriff der Sicht, bzw. View, verwendet. Eine Sicht ist ein Gruppierungscontainer.

Jede Aufgabe, d.h. entweder Blattaufgabe oder eine mit Gruppierungsinformationen angereicherte Nicht-Blattaufgabe, aus dem Taskmodell wird einer Sicht zugeordnet. Durch das Erledigen oder Beenden einer Aufgabe wird typischerweise eine nachfolgende Aufgabe aktiviert. Wurde diese Aufgabe einer anderen als der aktuellen Sicht zugeordnet, ist es nötig einen Übergang zu dieser Sicht durchzuführen.

Ein solcher Übergang kann sequentiell oder nebenläufig sein. In beiden Fällen wird die neue Sicht, die Zielsicht, angezeigt. Im Falle einer sequentiellen Transition wird die Quellsicht von der Anzeige entfernt, bei nebenläufigen Transitionen würden simultan Quell- und Zielsicht dargestellt.

Das Dialogmodell wird durch eine Graphenstruktur beschrieben, dem sogenannten Dialoggraphen. Die Syntax und die zulässigen Elemente von Dialoggraphen, d.h. die Kanten und Knotentypen, werden in [SE96] erläutert und definiert. In der zusammenfassenden Darstellung von Abbildung 2.7 findet sich ein Überblick.

Bei der Erstellung des Graphen sollte jeder Sicht ein Name zugewiesen werden. Darüberhinaus gilt, dass die Kanten, welche die Übergänge zwischen den Sichten definieren, immer gerichtet sind. Nur Endknoten haben keine ausgehenden Kanten.

Eine exemplarische Darstellung eines Dialoggraphen ist Abbildung 2.8. Hier wurde eine Anwendung zur Anlage und Erweiterung von Börsenportfolio beschrieben. Nicht alle möglichen Syntaxkonstrukte wurden verwendet, so fehlt etwa eine explizite Endsicht. Die grafische Darstellung ist [Rie06] entnommen, an gleichem Ort findet sich auch das zugehörige Aufgabenmodell.

Die Anfangssicht ergibt sich implizit aus den gerichteten Transitionen, in diesem Fall „Anmelden“ . In dieser Sicht entscheidet der Nutzer ob er sich mit einem vorhandenen Account anmelden möchte, oder eine Neuregistrierung vornimmt. Entsprechend dieser Auswahl wird der obere oder der untere Pfad verfolgt.

Beide Pfade führen zur Sicht „Portfolio“. Bis hierher waren alle Transitionen sequentiell. Entscheidet sich der Nutzer an diesem Punkt dafür, die Aufgabe „Aktien hinzufügen“ auszuführen, erfolgt der Übergang in die Sicht „Aktiensuche“ nebenläufig. In der konkreten Anwendung blieben also beide Fenster geöffnet und es wäre sogar möglich, durch wiederholtes Ausführen von „Aktien hinzufügen“ beliebig viele weitere Fenster zu öffnen. Die von der Sicht „Aktiensuche“ weiterführenden Transitionen sind sequentiell. Wenn der durch „Aktien hinzufügen“ gestartete Transitionszyklus wieder die Sicht „Portfolio“ erreicht, wird das zusätzliche Fenster geschlossen.

Das Metamodell des Dialoggraph-basierenden Dialogmodells zeigt Abbildung 2.9. Es stellt weitere Funktionalität für Dialogmodelle bereit, welche nicht durch [SE96] gefordert wird, aber auch schon im Beispiel

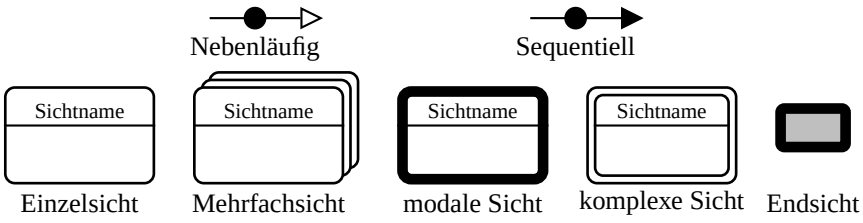

Abbildung 2.7: Syntaxelemente eines Dialoggraphen

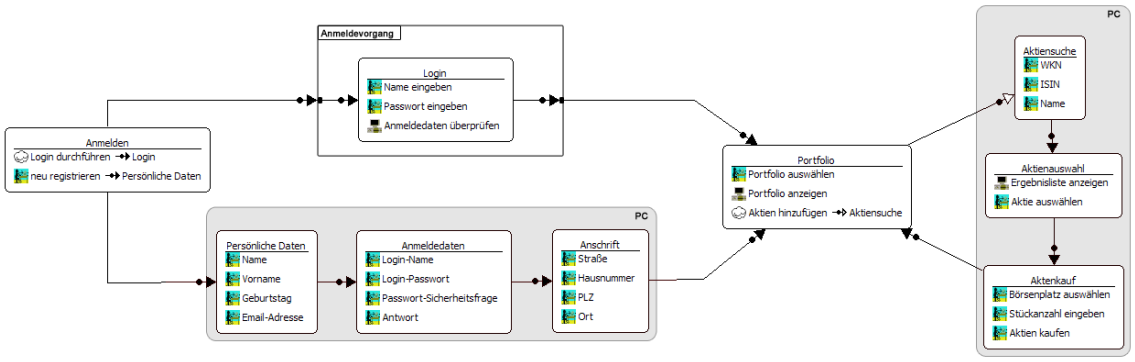

Abbildung 2.8: Beispiel eines Dialoggraphen, entnommen aus [Rie06]

von Abbildung 2.8 verwendet wurde. Die wesentlichen Erweiterungen gegenüber [SE96] sind die Einführung eines Port-Konzeptes, die Möglichkeit der Gruppierung von Sichten und die Hierarchisierung von Dialoggraphen.

Bei einem Port handelt es sich um eine vereinheitlichte Ein-/Ausgangsschnittstelle in eine Sicht oder auch in einen vollständigen Dialoggraphen. Transitionen, insbesondere auch sichtinterne Übergänge, lassen sich mittels des Portkonzeptes feingranularer steuern. Die Hierarchisierung entspricht in ihrer Natur derjenigen bei den UML-Statecharts. Eine Sicht kann, an Stelle einer Folge von Aufgaben, genau einen vollständigen Dialoggraphen enthalten. Innerhalb dieses können wiederum Sichten Dialoggraphen enthalten. Eine Limitierung der Verschachtelungstiefe ist nicht vorgesehen.

Sichten können in Gruppen zusammengefasst werden. In Abbildung 2.8 wurden zwei Gruppen definiert. Das eine Tripel enthält die Sichten („Persönliche Daten“ „„Anmeldedaten“ „„Anschrift“), das andere wird durch („Aktiensuche“ „„Aktienauswahl“ „„Aktienkauf“) gebildet. Zur Kenntlichmachung sind beide Tripel in der Abbildung farblich hinterlegt. Die Gruppenkennzeichnung „PC“ indiziert die Anwendung dieser Sichten-Gruppe/-Abfolge nur für die Plattform Desktop-Rechner.

Ableiten des Dialoggraphen aus dem Aufgabenmodell

Das Erstellen von Dialoggraphen kann durch eine Reihe von Softwarewerkzeugen unterstützt werden. So zeigt das Beispiel in Abbildung 2.8 einen Ausschnitt aus einem Screenshot von DiaTask, in der Version als GEF-Plugin. Auch mit Werkzeugunterstützung bleibt die Dialogmodellierung zunächst eine rein manuelle Tätigkeit. Die laut der Prozessübersicht, siehe Abbildung 2.1, anzustrebende patternbasierte Transformation ist noch zu erforschen. Ansätze dazu existieren bereits, der in [Die08] erarbeitete und in [WF09] in den weiteren Prozesskontext eingebettete Vorschlag, soll im Folgenden kurz erläutert werden.

Diebow schlägt in [Die08] vor, dass eine automatische Ableitung des Dialogmodells aus dem Aufgabenmodell durch bereits im Aufgabenmodell annotierte Zusatzinformationen erfolgen kann. Die Annotierung erfolgt durch menschliche Designer während der normalen Erstellung des Taskmodells. Dabei wird für ausgewählte Knoten, also Tasks, ein Algorithmus ausgewählt, nach dessen Regeln der Knoten oder dessen Kindern auf Sichten verteilt werden sollen. Diebow identifizierte vier Strategien, welche durch ihre englischsprachigen Abkürzungen bezeichnet werden:

ICV - Integrate Children into View: Alle Kindaufgaben des jeweiligen Aufgabenknotens werden der selben Sicht zugeordnet.

IN - Ignore Node: Der jeweilige Task, und der gesamte darunterliegende Teilbaum, werden ignoriert. Das

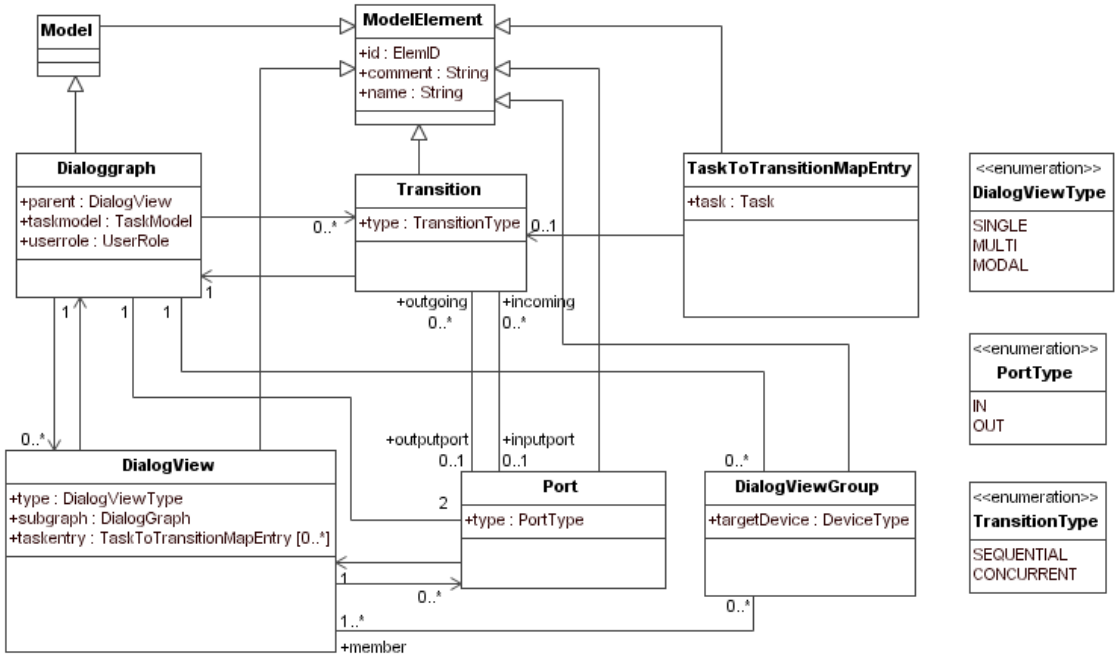

Abbildung 2.9: Metamodell Dialoggraph nach [FRW08]

bedeutet, dass keiner der darin enthaltenen Aufgabenknoten im Dialogmodell verwendet wird.

PUNM - Pick Up for Navigation Model: Derartig annotierte Aufgabenknoten werden in jedem Fall im Dialogmodell verwendet.

VL - View List: Hier erfolgt eine explizite Sichtzuweisung mittels einfacher Terme. Die EBNF-Grammatik für einen solchen Term lautet:

```
Kind ::= 1{"0" .. "9"}.
Term ::= "{" Kind {"," Kind } "}" {"," "{" Kind {"," Kind } "}" }.
```

Listing 2.1: EBNF Grammatik für die Sichtzuordnung von Aufgaben

Die Kindknoten werden nummeriert, ein Paar der geschweiften Klammern im Term entspricht einer Sicht im Dialogmodell. Sichten können nicht verschachtelt werden. Eine kommagetrennte Folge von Kindknotennummern definiert die View-Zuordnung der jeweiligen Aufgabe.

Die Strategien IN und PUNM sind nur im Zusammenhang mit Rollen oder Gerätemodellen sinnvoll einsetzbar. Die Abbildung 2.10 zeigt die Vorgehensweise an einem abstrakten Beispiel. Die Darstellung illustriert auch, dass jeder Knoten des Aufgabenmodells annotiert werden kann, neben Aufgaben auch die Operatorknoten. Zur Vereinfachung wurden die Strategien ICV, IN und PUNM hier nicht angewendet. Für den Enabling-Operator wurde die Konversionsreihenfolge mit einem Term über der Grammatik aus Listing 2.1 definiert. Der Term $\{0\}\{1,2\}\{3\}$ besagt, dass drei Sichten anzulegen sind. Die Erste und die Dritte sind mit nur einer Aufgabe zu belegen. In die zweite Sicht sind die beiden Tasks, die an den Kindpositionen 1 und 2 liegen, zuzuordnen. Würde anstelle der VL-Strategie am gleichen Knoten die ICV-Strategie gewählt, ergäbe sich nur eine einzige Sicht welche alle vier Kindknoten enthielte.

Bei der Erklärung des Metamodells des Aufgabenmodells wurde die Aufzählungsklasse AggregationHint

ausgelassen und stattdessen hierher verwiesen. Sie dient über die aggregation∗-Attribute der Task-Metaklasse zur Annotation der zu verfolgenden Dialogmodell-Erzeugungsstrategie.

Mockup-Prototypen aus dem Dialoggraph

Sobald das Dialogmodell spezifiziert wurde ist es möglich, einen Prototypen der Anwendungsoberfläche zu generieren. Die dabei entstehende Oberfläche kann, zu einem sehr frühen Zeitpunkt im Softwareentwicklungsprozess, genutzt werden um gemeinsam mit technisch weniger versierten Stakeholdern die Navigationsstruktur zu testen.

Auf dieser Abstraktionsebene liegen kaum verwendbare Informationen darüber vor wie die jeweiligen Aufgaben in der Oberfläche reflektiert werden sollen. Es scheint daher notwendig, Standardabbildungen vorzudefinieren. Die nötige Interaktivität der Oberfläche muss mit den gewählten Standard-Mappings umsetzbar sein.

Ein simples aber funktionales Mapping verwendet eine Kombination der Techniken XUL und HTML. Eine HTML-Seite soll als Startcontainer für die Simulation dienen, die eigentlichen Views werden per XUL dargestellt. Aus jeder View wird ein XUL-**window** generiert, jede Aufgabe als XUL-**button** umgesetzt und die Navigation durch Klicken auf die erstellten Buttons ausgelöst und über Javascript umgesetzt.

Die Startmaske für den Einsprung in die zu erstellende Anwendung wird mit einer Model-To-Text Transformation im Kontext des Wurzel-Dialoggraphen erstellt. Listing 2.2 zeigt den Quellcode für ein xPand-Template zur Generierung der HTML-Startmaske. Über OCL-Ausdrücke wird die Referenz auf den Start-Port ermittelt und für die darin referenzierte Zielansicht die Transformation des Gesamtgraphen gestartet.

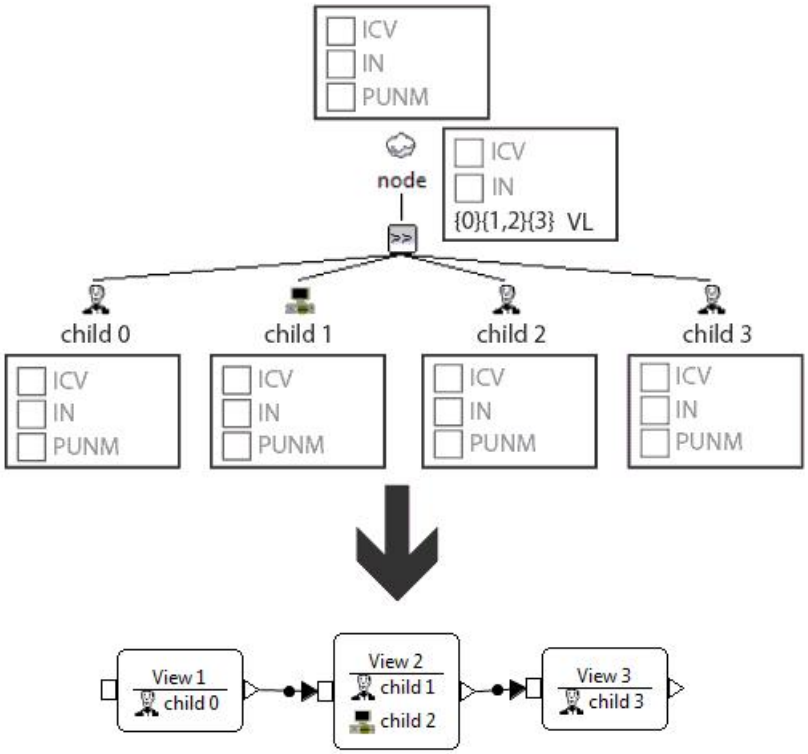

Abbildung 2.10: Annotiertes Aufgabenmodell und abgeleiteter Dialoggraph, nach [WF09]

Zur Interpretation der Selektorausdrücke kann Abbildung 2.9 hilfreich sein.

Nach Durchführung der Transformation nach Listing 2.2, wurde eine HTML-Datei mit dem Namen des Dialoggraphen erstellt. Diese Datei definiert nicht mehr als einen Link zu einer XUL-Datei.

```
«DEFINE root FOR Dialoggraph»
  «FILE name+".html"»
    <html><head><title>Dialoggraph−Simulation</title></head><body>
      «EXPAND Dialoggraph (this.Port.selectFirst(port|port.type.toString()=='in'))»
    </body></html>
  «ENDFILE»
«ENDDEFINE»

«DEFINE Dialoggraph (Port startPort) FOR Dialoggraph»
  «LET startPort.outgoing.at(1).inputport.DialogView AS startView»
  <a href="«name»/«startView.name».xul" target="_new">Simulation von «name» starten</a>
  «ENDLET»
«ENDDEFINE»
```

Listing 2.2: Ausschnitt aus einem xPand-Template

Die XUL-Dateien entstehen als Transformationergebnis der einzelnen DialogViews. Der Dateiname der XUL-Datei ergibt sich aus dem Namen der Sicht, dieses Wissen wurde für die Generierung des Linkziels im Template Dialoggraph (Port startPort) bereits verwendet um die Start-Sicht zu verlinken. Jede Transition wird in eine JavaScript-Navigationsanweisung übersetzt. Da Transitionen nach Aufgabenbeendigung durchgeführt werden, werden diese Navigationsinstruktionen in einem Attribut des jeweiligen Aufgaben-**button** als Event-Handler abgelegt. Auf die Abbildung der dazu notwendigen xPand-Templates soll an dieser Stelle verzichtet werden.

Das visuelle Ergebnis der Mockup-Generierung ist vom ästhetischen Standpunkt her sicher unbefriedigend. Einen Eindruck davon gibt Abbildung 2.11, darin wird das abstrakte Beispiel aus Abbildung 2.10 zu einem Navigationsprototypen fortgeführt. Die Oberflächen wurden durch Seamonkey gerendert. Links befindet sich die HTML-Einsprungsmaske, daran schließen sich in der Reihenfolge aus dem Dialogmodell die weiteren Sichten an.

Das der generierte Prototyp optisch roh und unfertig wirkt, hat durchaus auch positive Seiten. Van Buskirk et al. [VM03] bemerken, dass unfertig wirkende GUIs stimulierend auf die Diskussion mit künftigen Benutzern und anderen Stakeholdern wirken. Wohingegen bei aufwendig gestalteten Oberflächen dieselben

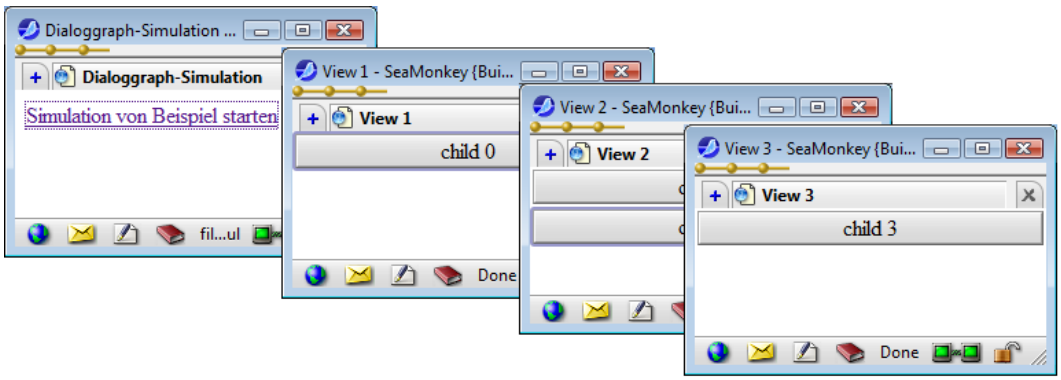

Abbildung 2.11: Screenshot-Sequenz eines generierten Navigationsprototypen

Nutzer Änderungsvorschläge nur eher zögerlich vorbringen. Van Buskirk leitete aus dieser Beobachtung das „Keep it ugly“-Prinzip für die Diskussion von Gestaltungsalternativen ab.

2.1.3 Abstrakte Oberfläche

Die abstrakte Oberfläche ist die Modellebene zur plattformunabhängigen Beschreibung der Benutzerschnittstelle. Den Plattform-Begriff definiert die OMG in [OMG] folgendermaßen:

> „**Platform:** ... set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented“

Diese Definition entstammt den Begriffserklärungen zu dem von der OMG vertretenen Paradigma der Model-Driven Architecture. Eine Plattform ist demnach eine Menge von Untersystemen bzw. Technologien, welche eine *zusammenhängende Funktionalität* zur Verfügung stellt. Da der Begriff in engem Zusammenhang zu den OMG-MDA Modellarten steht, lohnt möglicherweise noch die Betrachtung der Definition des Plattform-Modells:

> „**Platform Model:** A set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform.“

Leider führt dies auch nicht zu mehr Erkenntsgewinn als jede andere Formalisierung eines offensichtlichen Sachverhaltes. Es bleibt festzuhalten, dass eine noch generische Festlegung des Plattform-Begriffes kaum erfolgen kann.

Um die angemessene Abstraktionsebene für das Modell des Abstract User Interface (AUI) zu identifizieren ist es daher zwingend notwendig, sich für eine Plattform-Bedeutung zu entscheiden. Beschränkt man sich auf den Bereich der Softwaretechnik, werden in unterschiedlichen Kontexten mindestens diese Plattformbedeutungen unterschieden:

1. Oberflächenframeworks
2. UI-Beschreibungssprachen
3. Betriebssysteme und deren verschiedene Versionen
4. Geräteklasse, wie etwa Mobiltelefon oder Notebook
5. andere formalisierte Befähigungsklassifizierungen, wie beispielsweise über Java Specification Requests (JSR)

Zusätzlich können auch Faktorkombinationen aus mehreren dieser Gruppen eine Plattform charakterisieren. Plakativer ausgedrückt: einerseits bildet ein Symbian-Smartphone eine andere Plattform als ein Mobiltelefon auf der Basis von Windows Mobile. Andererseits sind beide zumeist in der Lage Applikationen der Java Micro Edition ablaufen zu lassen, aus dieser Sicht gibt es zunächst keine Plattformunterschiede. Solche Unterschiede finden sich jedoch mit hoher Wahrscheinlichkeit bei der Menge der umgesetzten Spezifikationen aus den JSRs.

JSRs sind Vorschläge für APIs für neue Funktionalität oder Änderungen an bestehenden Java-APIs. Sie werden im „Java Community Process“ [Jav] öffentlich diskutiert, formalisiert und verabschiedet. Oftmals werden die verfügbaren Features mobiler Geräte über die implementierten JSRs angegeben.

Im Rahmen dieser Arbeit werden besonders 2-dimensionale menü- und mausbasierte Benutzungsoberflächen untersucht. Die zu Grunde liegende Hardware soll hier nicht betrachtet werden. Ebenfalls nicht im Fokus stehen die anderen Aspekte der tatsächlichen technischen Umsetzung einer Oberfläche; die Geräteklasse und die speziellen Fähigkeiten einer Umgebung sind zunächst irrelevant.

Die Plattform für den hier betrachteten modellgetriebenen Prozess bilden Oberflächenframeworks und UI-Beschreibungssprachen für WIMP-Oberflächen. Die Anforderung an ein plattformunabhängiges AUI Modell ist daher, diese Art von Oberflächen allgemein zu definieren. Ein solches Metamodell enthielte theoretisch die abstrakte Beschreibung der Schnittmenge aller in Frage kommenden Plattformen.

In Abschnitt 1.3 wurden bereits einige Ansätze zur Erstellung derartiger Metamodelle diskutiert. Zur Verwendung im Rostock MD-UID Prozess kommen zwei Metamodelle. Bei einer Variante handelt es sich um eine Weiterentwicklung des in [Mül03] entwickelten Modells für abstrakte Oberflächen aus dem X3USGP [Mül03] Prozessmodell. Alternativ wird auch oft eine Untermenge von XUL als Ersatz-AUI-Modell, für Mockup-Prototypen, verwendet. Diese Untermenge wurde nie formal fixiert und die Auswahl der enthaltenen Elemente ist willkürlich. Wegen dieser Schwächen sei diese Alternative hier nur erwähnt, aber nicht weiter ausgearbeitet.

Das AUI-Metamodell des X3USGP-Prozesses nennt sich X-AIM, es handelt sich um ein XML-Derivat, welches als DTD[1] definiert vorliegt. Abbildung 2.12 und Abbildung 2.13 stellen das Ergebnis der Übertragung dieser Grammatik in die MDA-Methodik dar. Beide Grafiken wurden mit dem EMF-eigenen Diagrammeditor erstellt.

Das EMF-Metamodell für X-AIM entstand als das Ergebnis manueller Bearbeitung. Theoretisch ist es möglich aus der X-AIM DTD direkt ein zugehöriges EMF-Metamodell zu generieren. Dazu müsste zunächst das der DTD äquivalente XML-Schema generiert und dieses anschließend in ein Ecore-Metamodell übertragen werden. Eine Reihe manueller Eingriffe wären allerdings auch in diesem Prozess notwendig. So entsprechen etwa die in der DTD vorgesehenen Element-Namen nicht den Syntaxanforderungen für Ecore-Metaklassen, weiterhin sind geeignete EMF-Typen für die unterspezifizierten DTD-Attribute festzulegen. Die komplett manuelle Modellerstellung schien daher in diesem Fall daher die angemessenste Variante zu sein.

Die Darstellung des Metamodells wurde in zwei Segmente aufgeteilt. Die Abbildung 2.12 zeigt die Grundstruktur des Modells. Die in dieser Abbildung referenzierten Metaklassen Table, ListN und Tree werden in Abbildung 2.13 definiert.

Instanzen der Klasse UI dienen der Beschreibung jeweils eines Fensters. Sie enthalten mindestens eine Instanz eines InteractionElement. Interaktionselemente dienen entweder zur Ausgabe an oder der Eingabe von Informationen durch den Benutzer. Instanzen der Klasse UIO sind unspezifizierte UI-Elemente, sie enthalten ihrerseits wieder beliebig viele InteractionElement-Instanzen als Kindobjekte. Über diesen Mechanismus können Gruppierungen vorgenommen werden. Das X-AIM Metamodell enthält sonst keine Metaklassen welche zur Angabe von Layoutinformationen verwendet werden können. Es ist lediglich indirekt möglich, aus der Elementverschachtelung und -reihenfolge per Heuristik Layouthinweise abzuleiten.

Vorgesehene Eingabewidgets sind Eingabefelder (InputString), Tabellen(InputTable), Bäume(InputTree), Auswahllisten (Input1N und InputMN) und Schaltflächen (InputTrigger). Mit Ausnahme der Schaltflächen sind die gleichen Widgets für die Ausgabe von Informationen verfügbar. Die Angabe 1N bzw. MN der Auswahllisten, ist als Einfachselektion (1 aus N) bzw. Mehrfachauswahl (M aus N) zu interpretieren. Abweichend von der X-AIM Spezifikation des Typs InputString, und seinem Ausgabeäquivalent, wurde der Stringwert als Attribut der Metaklasse modelliert. Bzw. wurde an Stelle des in X-AIM definierten string_value-Typen

[1] Doctype Definition

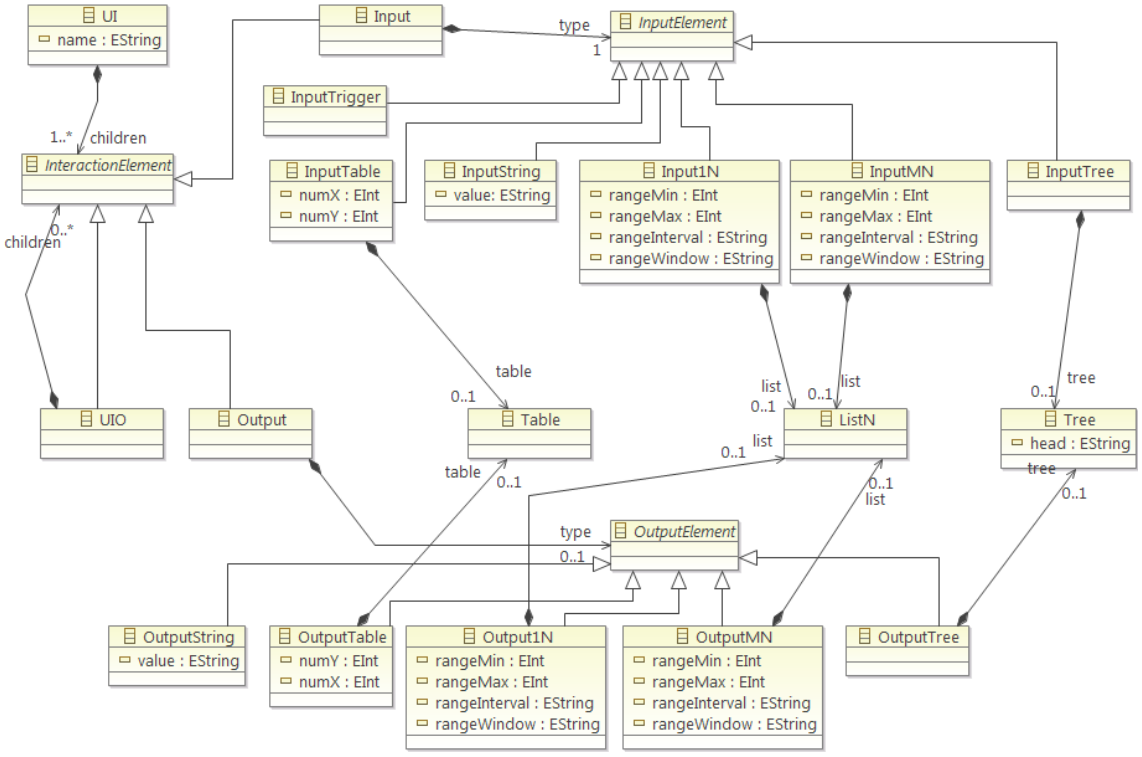

Abbildung 2.12: Metamodell für AUI im X-AIM Prozess, Grundstruktur nach [Mül03]

der Ecore-Standardtyp EString verwandt.

Die komplexeren Ein-/Ausgabewidgets nutzen weitere Metaklassen um genauer beschrieben zu werden. Eine Tabelle besteht gemäß dem Metamodell in Abbildung 2.13 aus Tabellenkopf oder Tabellendaten. Im Tabellenkopf werden die Überschriften der Spalten bzw. Zeilen, das ist interpretationsabhängig, angegeben. Der Tabellenkörper definiert die Informationen in den einzelnen Tabellenzellen.

Bäume haben ein eigenes Label, das Attribut head. Dieses wird zur Darstellung des Wurzelknotens verwendet. Neben diesem Label gibt es eine Sequenz von Kindknoten, dies sind die Knoten mit der Tiefe 1 im Baum. Die Baumknoten selbst werden über die Metaklasse TreeItem spezifiziert. Jedem Knoten ist wieder-

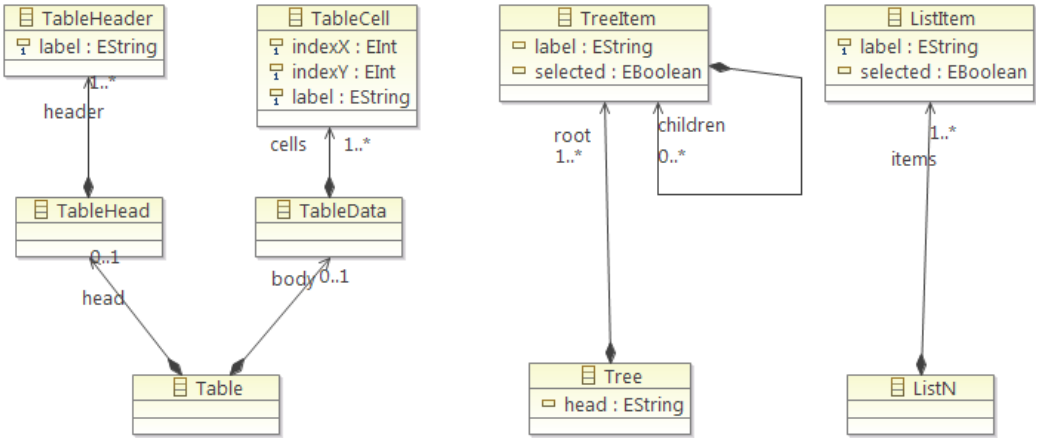

Abbildung 2.13: Metamodell für AUI im X-AIM Prozess, Widgets nach [Mül03]

um ein Label zugeordnet, sowie eine Auswahlmarkierung. Ein Knoten kann Vaterknoten einer beliebigen Anzahl Kindknoten sein. Bei der vorliegenden Definition des Metamodells, kann die Baumeigenschaft von Tree-Instanzen nur über Constraint-Prüfungen sichergestellt werden. Aus der ursprünglichen DTD-Grammatik folgt in jedem Fall, dass die Baumwidgets des X-AIM Bäume im Sinne der Graphentheorie sind. Auf der Klassenmodell-Ebene ist diese Forderung ohne Hinzufügung weiterer Attribute nicht umzusetzen.

Listen sind lediglich eine Sequenz von ListItems. Jedes Listenelement kann selektiert sein und verfügt über ein beschreibendes Label. Es sind keine Gruppierungen oder Hierarchisierungen vorgesehen. Die Unterscheidung zwischen Einfach- und Mehrfachselektion hat keinen Einfluss auf die Objektstruktur einer Liste.

Adaption als AUI-Modell im MDA-Prozess

Die Erstellung eines die DTD reflektierenden Ecore-Modells ist nur ein erster Schritt zur Integration des X-AIM Metamodells in den MDA-Ansatz. Es war beispielsweise nötig, den Modell-Klassen die ModelElement und Model-Eigenschaft zuzuordnen. Weiterhin entstand X-AIM als XML-Format und seine Konzeption ist dementsprechend nicht auf die Umsetzung mit Klassenmodellen als Metamodellen ausgelegt. Es war daher zu prüfen inwieweit die gleichen Konzepte eventuell mit weniger oder anders strukturierten Metaklassen umgesetzt werden können.

Natürlich ist auch die Querreferenzierung zu den anderen Modellen aus dem Gesamtansatz von Abbildung-2.1 zu integrieren. Wichtigstes Element davon ist es, die UI-Widgets einer bestimmten Aufgabe zuordnen zu können. Andere Änderungen umfassen das Ändern von Eigenschaftennamen, um sie dem aktuellen Sprachgebrauch anzupassen, und das Überarbeiten der Relationen allgemein.

Das Ergebnis dieser Überlegungen stellen Abbildung 2.14 und Abbildung 2.15 dar.

Die Grundstruktur einer abstrakten Oberfläche nach dem adaptierten X-AIM Metamodell wird in Abbildung 2.14 gezeigt. Die Klasse UI ist weiterhin der Modellcontainer, welcher Sequenzen beliebig vieler InteractionElement-Instanzen enthält. Ebenfalls beibehalten wurde die Unterteilung in Input, Output und UIO, als allgemeinem Container. Im Gegensatz zu X-AIM wird nun jedem Interaktionselement ein Task, vgl. Abbildung 2.2, zugewiesen.

Instanzen der Input und Output Klassen enthalten jeweils genau ein Widget des Typs IOElement. Die ver-

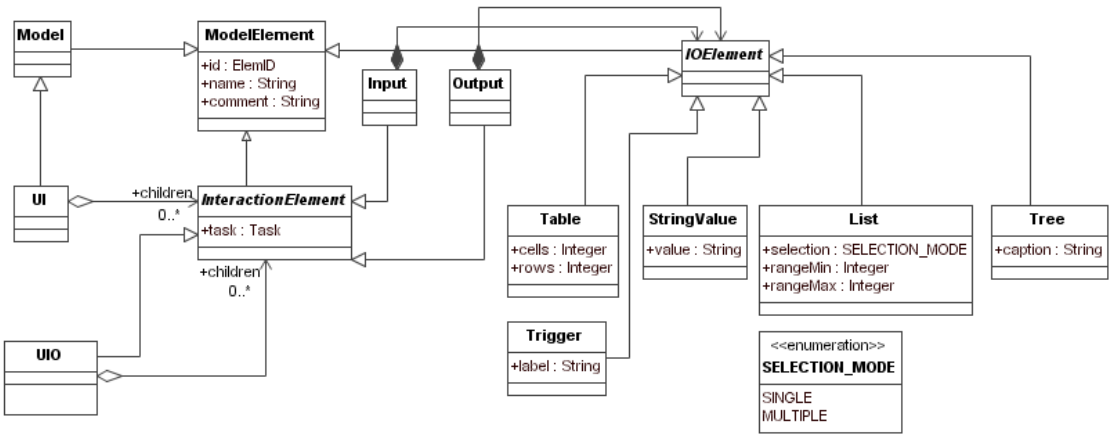

Abbildung 2.14: Adaptiertes AUI-Metamodell, Grundstruktur

fügbaren Widgets werden nun nicht mehr in zwei separaten Hierarchieebenen dupliziert. Als Widgettypen stehen weiterhin Tabellen, Auflistungen, Textfelder, Bäume und Schaltflächen Trigger zur Verfügung. Schaltflächen können nun auch als Ausgabe-Widgets verwendet werden. Elemente konkreter Oberflächen, welche als Schaltflächen zur Informationsausgabe genutzt werden, sind zum Beispiel Umschalter, auch als Togglebuttons bekannt. Schaltflächen können nun ebenfalls auch mit einem Label versehen werden.

Der Widget-Typ List steht für beliebige Auswahllisten. Der genaue Selektionstyp einer Liste wird über das Attribut selection, mit den Werten aus der Aufzählung SELECTION_MODE, gesteuert. Die Attribute rangeInterval und rangeWindow aus dem X-AIM Metamodell wurden nicht übernommen.

Tabellen sind weiterhin NxM-Gitter. Die Angabe der Tabellengröße erfolgt prinzipiell genau wie im X-AIM Metamodell, es wurden lediglich die Attributnamen in die derzeit allgemein üblichen geändert. Gleiches gilt für das caption-Attribut eines Baumes, hier wurde ebenso lediglich der Eigenschaftenname angepasst. Keine Änderungen ergaben sich für die Metaklasse StringValue.

Außerdem wurden die in Abbildung 2.12 dargestellten Kompositionsbeziehungen, soweit zutreffend, zu Aggregationen reduziert. Die Zuordnung der ModelElement-Eigenschaft ist trivial, auf dieser Ebene sollen alle Instanzen der Metamodellklassen im Editor bearbeitbar sein. Dementsprechend wurden diese als Unterklassen von ModelElement modelliert.

Die Modifikationen an den Metaklassen der komplexeren Widgets zeigt Abbildung 2.15. Alle hier dargestellten Klassen sind als Unterklassen von ModelElement bearbeitbar.

Von links nach rechts: Für Bäume wurde sichergestellt das die Instanzen die Baumeigenschaft erfüllen. Die children-Assoziation ist nun bidirektional mit den Kardinalitäten $0..1 \leftrightarrow 0..*$. Damit wird erreicht das jeder Knoten nur genau einen Vaterknoten haben kann. Tree ist in diesem Metamodell nicht mehr gleichzeitig die Wurzel des Baumes. Daher enthält die Aggregation root der Klasse Tree keine Sequenz von Kindknoten mehr, sondern nur die Referenz auf die Baumwurzel. Dies entspricht der typischen Modellierung von Bäumen als Klassendiagramm. Damit verbunden ist eine Semantikänderung der caption-Eigenschaft von Tree-Instanzen. Es handelt sich nicht mehr um das Label der Wurzel, sondern eher um einen beschreibenden Text im Sinne eines Titels oder einer Bildunterschrift.

Für die Elemente von Auflistungen ergaben sich keine Änderungen gegenüber dem X-AIM Modell. Da die

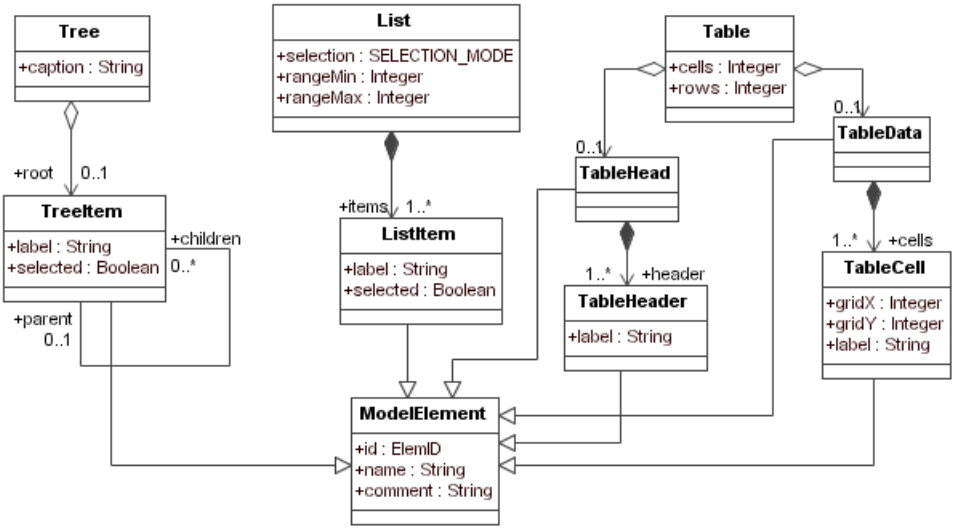

Abbildung 2.15: Adaptiertes AUI-Metamodell, Widgets

Kardinalitäten eine $1..*$ Beziehung vorsehen, konnte auch die Komposition beibehalten werden. Ähnliches gilt für die Tabelle. Mit Ausnahme der Attributsumbenennung von indexX und indexY nach gridX sowie gridY ergaben sich keine Änderungen.

2.1.4 Modelle für konkrete Oberflächen

Das konkrete Oberflächenmodell (CUI) ist das plattformspezifische Modell im modellgetriebenen Oberflächenentwicklungsprozess. In dieser letzten Prozessstufe wird die Anwendungsoberfläche mit den UI-Widgets der ausgewählten Plattform gestaltet.

Im Allgemeinen entsteht eine CUI-Instanz durch Model-To-Model Transformation aus einer Instanz eines abstrakten Oberflächemodells. Alternativ kann aber auch der aus dem Dialoggraph abgeleitete Mockup-Prototyp, siehe Unterabschnitt 2.1.2, als AUI angesehen werden und zu einer CUI-Instanz weiterverarbeitet werden.

Die in Unterabschnitt 2.1.3 angeführten Betrachtungen zum Begriff der Plattform treffen für die Positionierung des konkreten Oberflächenmodells ebenso zu. Dort wurde begründet, dass im Rahmen dieser Arbeit Frameworks für WIMP-Oberflächen als Plattform angesehen werden. Auch mit dieser Einschränkung ist die Menge der in Frage kommenden Metamodelle noch sehr groß. Zwei verschiedene Arten von Metamodellen sollen hier betrachtet werden.

1. die XML-abgeleiteten Oberflächenbeschreibungssprachen, siehe auch Abschnitt 1.3
2. Metamodell von Java-Swing, siehe Unterabschnitt 4.2.1

XML-abgeleitete Sprachen

In Tabelle 2.1 wird die Verfügbarkeit von formalen Beschreibungen der in Unterabschnitt 1.3.1 vorgestellten XML-UI Beschreibungssprachen gezeigt. Für drei der Modellierungssprachen scheint kein öffentlich verfügbares Modell (mehr) zu existieren. Ebenfalls drei Sprachen stellen nur eine DTD bereit.

DTD-Grammatiken sind ausreichend zur Prüfung von Syntax und Struktur der jeweiligen XML-Strukturen. Wie bereits in Unterabschnitt 2.1.3 angedeutet, ist es auch möglich, ausgehend von einer DTD zu einem Metamodell als Klassendiagramm zu gelangen. Weil jedoch eine DTD, ihrer Natur nach, eine Reihe von

Name	verfügbare Grammatik/Taxonomie/Metamodell
GIML	Grammatik liegt als DTD [gim] vor
VRIXML	keine öffentliche Grammatik, in Grenzen aus [CRC04] ableitbar
ISML	keine öffentliche Grammatik
XUL	XML Schema verfügbar, Details in Punkt 2.1.4.2
X3USGP	DTDs liegen in [Mül03] vor
useML	XML Schema verfügbar [use]
UsiXML	XML Schema, Rational Rose Modelle verfügbar [USIa]
XAML	XML Schema verfügbar
XIML	DTD für XIML1 für Forschungszwecke freigegeben, XIML2 weiter nicht veröffentlicht
UIML	XML Schema verfügbar [UIMa]
IM2L	keine Spezifikation verfügbar

Tabelle 2.1: Verfügbarkeit eines Metamodells für UIDLs aus Tabelle 1.1

Informationen über Datentypen und Wertebereiche nicht enthält, dürfte diese Transformation sinnvoll nur manuell durchführbar sein. Die fehlenden Puzzlestücke der Semantik müssten durch einen menschlichen Designer, in Kenntnis der Semantik der einzelnen Sprachelemente, ergänzt werden. Mangels verfügbarer (X3USGP) oder nicht mehr weiterentwickelter (GIML) Renderer scheint es nicht sinnvoll diesen Aufwand für GIML und X3USGP zu betreiben. Im Falle von XIML ist davon auszugehen, dass Puerta und seine Firma RedWhale Inc. längst über aussagekräftige Modelle verfügen, diese jedoch aus lizenz- oder firmenpolitischen Gründen nicht veröffentlichen.

Für fünf der UIDLs sind Spezifikationen erhältlich, welche als Quelle zur Erstellung eines Klassenmodells im MD-UI Prozess tauglich sind. Für UsiXML wird ein solches Modell im Rational Rose Format öffentlich [USIa] bereitgestellt.

Woitzel untersuchte im Rahmen einer studentischen Arbeit [Woi09] die Verwendung von UsiXML als CUI-Metamodell. Dabei wurde festgestellt, dass die Erstellung einer konkreten Oberfläche aus dem AUI-Modell, nach Unterabschnitt 2.1.3, ohne wesentliche Schwierigkeiten machbar ist. Allerdings stellte er auch fest, dass die Gebrauchstauglichkeit und vor allem die Standardkonformität der veröffentlichten UsiXML-Renderer unzureichend war. Es scheint daher zur Zeit nützlicher andere UIDLs als UsiXML zu verwenden.

Das im Rahmen des Useware-Ansatzes spezifizierte useML, kurz umrissen dargestellt in 1.3.1.2, wäre von der Papierform ein Kandidat hierfür. Das öffentlich verfügbares Schema der Sprachversion useML 1.0, dient allerdings nur zur Beschreibung von Abläufen. Erweiterungen in Richtung CUI-Modellierung sind derzeit in Arbeit, jedoch lässt sich im Moment das useML-Metamodell nicht als CUI-Metamodell verwenden.

Für UIML, Kurzvorstellung im Anhang auf Seite 135, sind laut der Projektwebseite [UIMb] bis zu elf verschiedene Renderer verfügbar. Problematisch ist, dass bei dieser Aufstellung nicht nach den verschiedenen Sprachversionen unterschieden wird. Dies hat jedoch letztlich keine Relevanz, da aktuell keiner der elf Renderer tatsächlich öffentlich verfügbar ist. Aus der Kurzbeschreibung der Renderer ist jedoch zu ersehen, dass UIML zur Anzeige zumeist in HTML, WML oder VoiceXML[2] übersetzt wurde. Leider scheint die Entwicklung an UIML in den letzten Jahren stillzustehen. Laut Jan van den Bergh wird an UIML weitergearbeitet, dennoch ist bis auf weiteres die vorläufige Spezifikation 3.1 vom März 2004, verfügbar bei [UIMa], die aktuellste Veröffentlichung. Die Nichtverfügbarkeit von Renderern und das auch sonst augenscheinlich verwaiste Projekt, lassen davon abraten, UIML als CUI-Metamodell einzusetzen.

Damit bleiben von den UIDLs aus Tabelle 2.1 noch zwei Sprachen zur Auswahl: XUL und XAML. Beide werden aktiv weiterentwickelt, stellen aktuelle Renderer bereit und es sind XML-Schemata als Sprachspezifikation erhältlich. Gleichzeitig sind beide sehr umfangreich, daher war es erforderlich, sich für eine Sprache zu entscheiden. Die Wahl fiel auf XUL, weil im Rahmen der Arbeitsgruppe bereits eine Reihe von Werkzeugen für diese Sprache entstanden war. Dies ist jedoch nicht als Wertung gegen XAML zu verstehen.

2.1.4.1 XUL

XUL - das Akronym steht für XML User Interface Language - ist die XML-UIDL für konkrete Benutzungsoberflächen im Prozessmodell nach Abbildung 2.1. Die Entwicklung und Spezifikation dieser Sprache ist eng mit der Gecko-Rendering Engine [Gec] verbunden. Gecko stellt die Implementierung von XUL und spezifiziert damit den de facto Standard. Eine formale Spezifikation existiert nicht. Aus den öffentlich verfügbaren Ressourcen kann dennoch ein Metamodell abgeleitet werden, die Vorgehensweise wird in

[2] für Sprachinterfaces

Punkt 2.1.4.2 erläutert.

Hauptsächlicher Bestandteil der XUL sind Widgets zur Deklaration von WIMP-Oberflächen. Das Sprachdesign sieht eine Trennung von Layout, Design und Dynamik vor. Dazu wird XUL eng verzahnt mit zwei weiteren Sprachen eingesetzt. Das ist zum einen die Auszeichnungssprache CSS - Cascading Style Sheets, erläutert in Unterabschnitt A.5.1 - und andererseits die Programmiersprache JavaScript, bzw. seit deren Standardisierung auch als ECMAScript bekannt. Eine kurze Ausführung zu JavaScript findet sich im Unterabschnitt A.5.2.

Der XUL-Teil einer Oberflächenbeschreibung deklariert nur die eingesetzten Widgets und deren Layout. Das Design wird mit CSS definiert, JavaScript kommt zur Definition der Nutzerinteraktionen zum Einsatz.

Verwendung von JavaScript und CSS innerhalb XUL

Die Elemente der XUL dienen der Beschreibung der Struktur der Oberfläche und definieren die anzuzeigenden Daten. Details der Darstellung, auch grundlegende Dinge wie Farben und Schriftarten, werden nicht mit den Sprachmitteln der XUL ausgedrückt. Jeder XUL-Renderer setzt zur Darstellung hierfür eigene Standardwerte ein.

Die Anpassung der Visualisierung erfolgt über CSS. Dazu verfügt jedes XUL-Widget über ein style-Attribut in welchem die relevanten CSS-Schlüssel, und die dafür gesetzten Werte, für dieses Element hinterlegt werden. Der in Unterabschnitt A.5.1 dargestellte Selektor-Mechanismus ist für die meisten Renderer zwar möglich, bevorzugt eingesetzt werden soll jedoch das style-Attribut.

JavaScript wird im Rahmen der XUL für den gleichen Zweck verwendet wie in HTML-Webseiten. Es dient zur Implementierung wesentlicher Teile der Dynamik der Oberfläche. Im Gegensatz zu der Integration von CSS erfolgt der JavaScript-Einsatz nicht über ein einziges spezielles Attribut[3]. Stattdessen existiert eine Reihe Attribute, die im Sinne von Eventhandlern die Reaktion des Widgets auf ein Oberflächen-Ereignis festlegen. Der Sprachstandard von XUL definiert über 30 solcher Events, z.B. onfocus wenn ein Widget den Focus erhält oder onkeypress für Tastendrücke. Einen Eindruck über das Zusammenspiel aller drei Sprachen gibt das Beispiel in Unterabschnitt A.5.3.

2.1.4.2 XUL Metamodell

Wie bereits dargelegt, existiert keine formale Spezifikation für XUL, und betrachtet man dessen Entwicklungsprozess, wird es auch in absehbarer Zeit keine solche geben. Einer Spezifikation am nächsten kommen die in unregelmäßigen Abständen, halb-offiziell und etwas indirekt, veröffentlichten XML-Schema von XUL.

Indirekt meint in diesem Fall, dass das Schema selbst nicht bereitgestellt wird. Wie man zu einem Schema gelangen kann, beschreibt [XUL]. Es läuft darauf hinaus, aus zwei Quelldateien im XML-Format über ebenfalls bereitgestellte XSLT-Transformationen das Schema zu generieren. Die Quellen sind die sogenannte XUL-Elementreferenz und die XUL-Strukturreferenz. Beide liegen in einem proprietären Format vor und enthalten sowohl die Grammatik als auch die notwendigen Typinformationen.

Zu den Standardmechanismen von EMF gehört ein sogenannter Model-Importer für XML-Schemata. Dessen Zweck ist es, für ein Schema das korrespondierende Ecore-Modell zu erzeugen. Dieser Importalgorithmus wurde für das XUL-Schema benutzt.

Wie bei vielen Automatismen war das Ergebnis dieser Model-To-Model Transformation nicht sofort zu-

[3] Ausnahme ist das **<widget />**-Tag mit seinem Attribut **script**

friedenstellend. Als Problem zu nennen sind hier die möglicherweise etwas unübliche Benennung der Metaklassen, die Aufteilung der Eigenschaften jedes Widgets auf zwei Klassen, eine große Anzahl nummerierter und anonymer Typen sowie generell eine unnötig komplizierte Typstruktur. Möglicherweise ergaben sich diese Probleme als Konsequenz daraus, dass es sich bei der Ecore-Erzeugung um die zweite komplett automatische Transformation der Ausgangsdaten (XUL Element- und Strukturreferenz) in Folge handelte.

Die Behebung dieser Unzulänglichkeiten ist im wesentlichen ein rein manueller Prozess. Er kann ergebnisneutral sowohl im XML-Schema als auch im erzeugten Ecore-Modell durchgeführt werden. Ich habe mich dafür entschieden die notwendigen Modifikationen im Ecore-Modell vorzunehmen. Einerseits waren dort ohnehin die zuvor bereits angerissenen Änderungen vorzunehmen, andererseits war die mir zur Verfügung stehende Werkzeugunterstützung besser zur Ecore- als zur Schema-Bearbeitung geeignet. Ausführlichere technische Details finden sich im Anhang, siehe Abschnitt C.1.

In Abbildung 2.16 wird ein Ausschnitt des resultierenden XUL-Metamodells dargestellt. Wegen der Dimensionen dieses Modells scheint eine vollständige Angabe an diesem Ort nicht sinnvoll, es wird stattdessen unter anderem hier [Mod] bereitgestellt.

Insgesamt enthält das XUL-Metamodell in der nachbearbeiteten Version 151 Klassen und Interfaces mit insgesamt 443 Attributen, dazu kommen weitere 31 Datentypen. Dabei handelt es sich ausschließlich um Enumerations. Selbstverständlich werden nicht alle der 131 Klassen genutzt um UI-Widgets zu beschreiben. Insbesondere enthält XUL eine eigene Templatesprache, Metaklassen die Elemente dieser Sprache definieren sind für ein CUI-Modell nicht unbedingt notwendig. Es besteht jedoch ebenfalls kein zwingender Grund diese zu entfernen, daher wurden sie beibehalten. Für eine ausführliche Erläuterung des XUL-Metamodells sei auf den Anhang, Abschnitt C.2, verwiesen.

Zusammenfassung zum modellgetriebenen Prozess

In den vorstehenden Abschnitten wurden Metamodelle, Konzepte und Transformationsansätze vorgestellt, mit denen der transformationsorientierte modellgetriebene UI-Entwicklungsprozess gemäß Abbildung-2.1 umgesetzt werden kann. Das XUL-Metamodell für die konkrete Oberfläche bildet gewissermaßen den Abschluss des Prozesses.

Im Rahmen dieser Arbeit wurde noch ein weiteres Metamodell für konkrete Benutzungsoberflächen entwickelt. Das in Unterabschnitt 4.2.1 erstellte Oberflächenmodell für das Swing-Framework kann nach

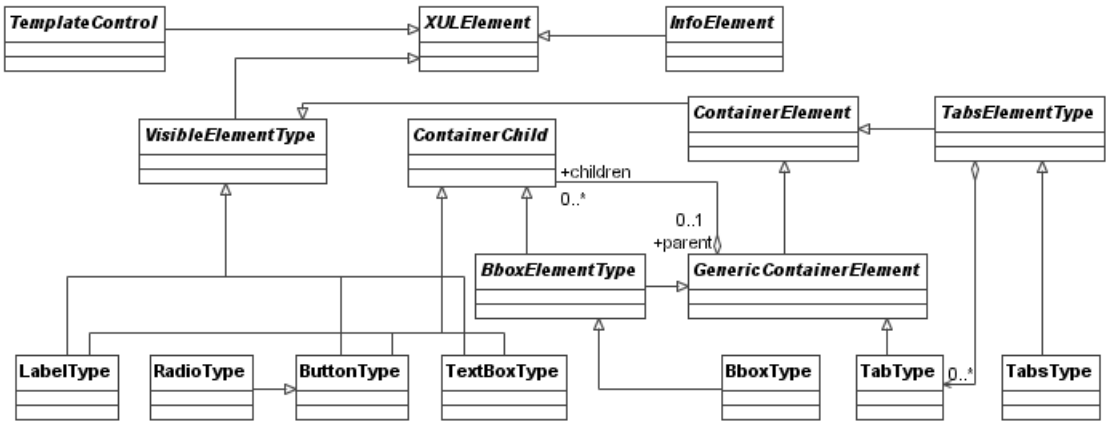

Abbildung 2.16: Ausschnitt des XUL-Metamodells

Einführung von Task-Referenzen ebenfalls im hier beschriebenen modellgetriebenen Prozess eingesetzt werden.

2.1.4.3 Grafischer XUL-Editor

Bei der Entwicklung von grafischen Benutzungsoberflächen kann permanentes visuelles Feedback eine sehr nützliche Hilfe sein. Viele Werkzeuge zur Unterstützung und Durchführung des Rostocker modellgetriebenen Prozesses sind daher in der Lage die Auswirkungen der getroffenen Designentscheidungen grafisch darzustellen. Auch für die Modellebene der konkreten Oberfläche existiert ein solches Tool, der grafische XUL-Editor. Die Abbildung 2.17 zeigt einen Screenshot dieser Anwendung.

Die Wurzeln dieses Editors liegen im Jahr 2003, ursprünglich entwickelt als grafischer WYSIWYG[4] GUI-Builder für Java Swing namens V4ALL [Assa]. Er verfügt über die gewöhnlichen Basisfeatures solcher Editoren, bietet beispielsweise Drag&Drop für Elementerzeugung und Platzierung. Die ursprüngliche Version [Assa] des Editors wurde außerhalb und komplett unabhängig von der Universität Rostock entwickelt. Da V4ALL aber als quelloffene Software zur Verfügung gestellt wurde, konnten studentische Arbeiten, sowie eigene Änderungen, darauf aufbauen und den aktuellen Entwicklungsstand schaffen.

Hier in Rostock wurde V4ALL im Laufe der Zeit zur Test- und Integrationsumgebung für diverse Ideen und Konzepte im Rahmen der modellgetriebenen UI-Entwicklung. Zu den durchgeführten Änderungen zählt, dass V4ALL um Funktionalität ergänzt wurde deklarative Oberflächen mit XUL zu designen. Anfangs noch vorgesehene Mechanismen zur Generierung von Java-Quellcode sind im Zuge mehrfacher Umgestaltungen der Software entfernt worden. V4ALL wurde damit zu einem reinen XUL-Editor. In seiner aktuellen Inkarnation wird der Editor daher schlicht als XULE, für XUL-Editor, bezeichnet.

XULE ist ein Eclipse-Plugin und seine technische Grundlage das Graphical Editing Framework (GEF) [Fouc] des Eclipse-Projekts. Das GEF forciert einen Model-View-Controller Ansatz für die damit erstellten Editoren, dementsprechend folgt auch XULE diesem Pattern. Ursprünglich wurden alle dafür notwendigen

[4] What You See Is What You Get

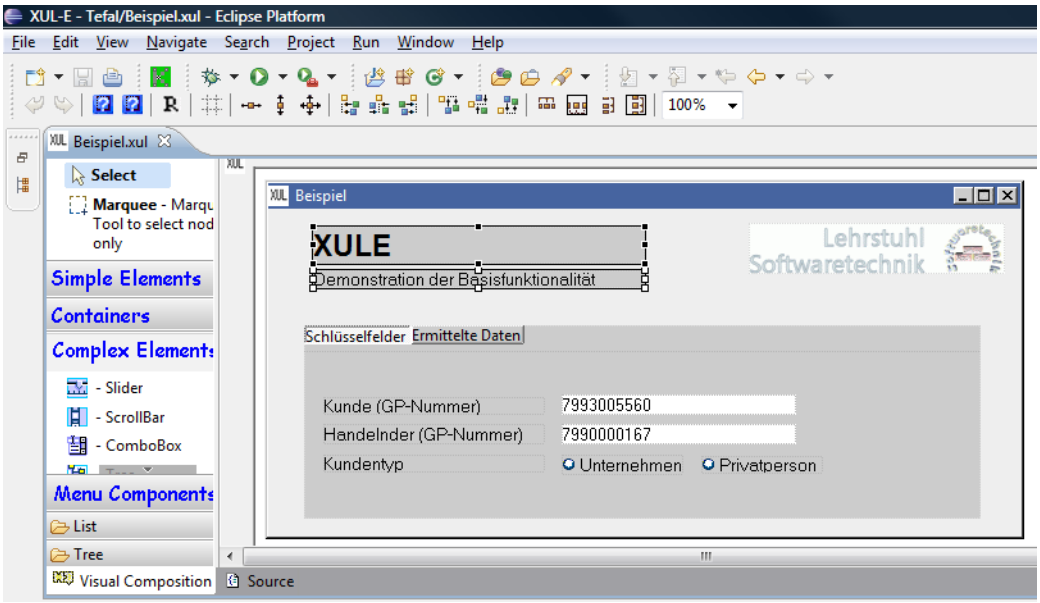

Abbildung 2.17: XULE - XUL-UI im Bearbeitungsmodus geöffnet

Klassen einmal manuell erzeugt. Schon bei der erstmaligen Erstellung eine beeindruckende Fleißarbeit, erwuchsen bei der späteren Wartung und Weiterentwicklung eine Reihe von Schwierigkeiten. Zu nennen wären hier Doppel- und Mehrfachimplementierungen, regelmäßige Inkonsistenzen im Lade- und Speicherzyklus, fehlende Eigenschafteneditoren und Weiteres. Besonders unpraktisch war auch, dass es nie gelang, den durch XULE generierten XUL-Quellcode vollständig standardkonform[5] zu erhalten.

Seit dem Vorliegen des XUL-Metamodells konnte darüber nachgedacht werden, dieses Metamodell in geeigneter Form für das interne Modell von XULE nutzbar zu machen. Denn vermutlich ließen sich dadurch einige der zuvor erwähnten Schwierigkeiten mit den manuell erstellten Modellen beheben. Als Hauptvorteil für Nutzer würde sich eine hundertprozentige Standardkonformität der bearbeitbaren Eigenschaften und des erzeugten Modells ergeben. Aus softwaretechnischer Sicht gäbe es ebenfalls eine Reihe signifikanter Vereinfachungen. Die internen Modellklassen des MVC-Paradigmas würden entfallen und Standardaufgaben wie die Modell(de-)serialisierung könnten durch erprobte EMF [emf]-Mechanismen erfolgen. Prinzipiell sollte, bei konsequenter Umsetzung des MVC-Patterns, das Austauschen des Modells einfach möglich sein und lediglich Anpassungen in der Controller-Komponente notwendig werden.

Leider wichen auch im Falle des XULE Theorie und Praxis von einander ab. Es zeigte sich, dass das via EMF generierte XUL-Modell das bisherige interne Modell nicht vollständig ersetzen konnte; beispielsweise sei an dieser Stelle an die Nichtabbildbarkeit von Farben und Schriftarten in XUL erinnert, obwohl dieses nicht das gravierendste Problem darstellte. Es gab zwei Alternativen: entweder müsste das XUL-Metamodell um XUL-fremde Eigenschaften erweitert werden oder aber die bisherige interne Modellstruktur würde beibehalten werden, jedoch so modifiziert, dass das die XUL-Eigenschaften in Instanzen des XUL-Metamodells gespeichert würden. Trotzdem die zweite Variante bereits im Vorhinein als umständlicher erkannt wurde, wurde entschieden diesen Weg zu gehen. Diese Variante ermöglichte es nämlich, die Umwandlung des internen Modells schrittweise vorzunehmen und zwar indem nach und nach Teil-

[5] d.h. Interpretationsidentisch zur Gecko-Engine

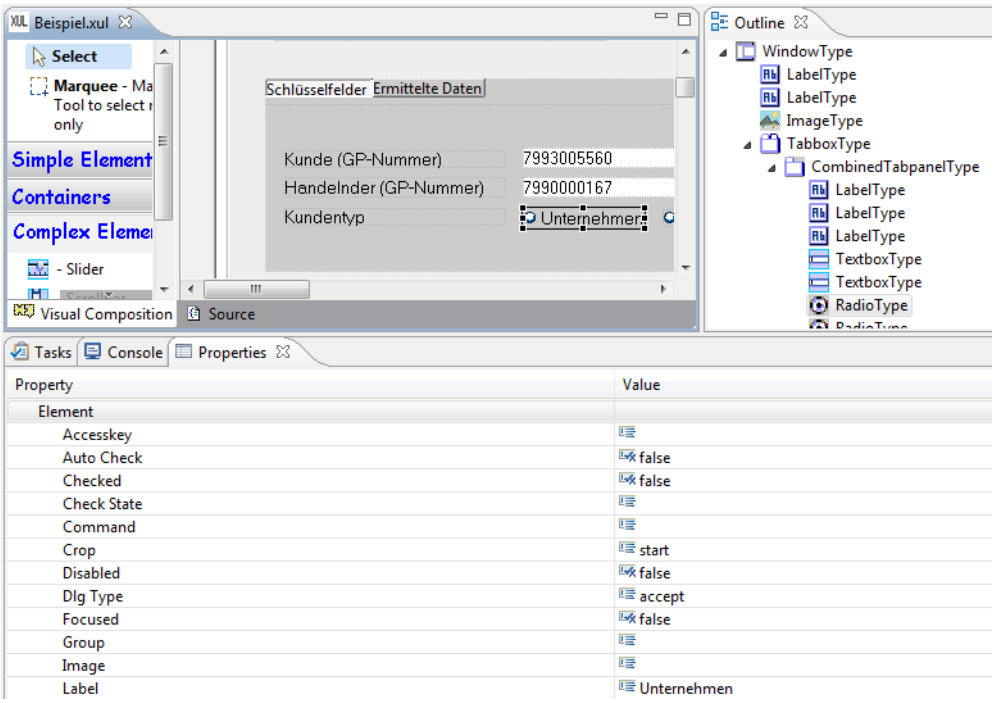

Abbildung 2.18: XULE - Strukturübersicht und Eigenschaftenbearbeitung

funktionalitäten des ehemaligen internen Modells an die XUL-Instanz ausgegliedert wurden. Zusätzliche Schwierigkeiten ergaben sich auch daraus, dass die im MVC-Paradigma vorgesehene strikte Trennung aller drei Teile, nach Jahren ständiger Umbauarbeiten, nicht mehr bestand. Letztlich führte die Ersetzung bzw. Ergänzung des internen Modells durch das generierte XUL-Modell zur einem umfangreichen mehrmonatigen Redesign des XULE Quellcodes.

Die wesentlichen Ergebnisse dieser Re-Implementierung des XULE sind in der Abbildung 2.18 zu sehen. Im unteren Bereich, d.h. bei den „Properties“, können nun alle im Metamodell definierten Eigenschaften eines Elementes bearbeitet werden. Diese Ansicht wird nahezu vollständig mittels des automatisch generierten XUL Ecore-Editors dargestellt. Minimale Änderungen waren lediglich erforderlich um Änderungsbenachrichtigungen und damit die Undo/Redo-Funktionalität des XULE sicherzustellen.

Die ebenfalls erkennbare „Outline“-Ansicht stellt die effektive Hierarchie der XUL-Modellelemente dar. Genutzt wird dieser Mechanismus insbesondere zur Bearbeitung der implizit visualisierten Sprachelemente. Dazu zählen beispielsweise die Spaltenüberschriften in Bäumen oder Listen sowie deren zugehörige Kontrollstrukturen. Diese Sprachelemente werden zwar nicht direkt als Widget an der Oberfläche dargestellt, es muss aber dennoch möglich sein, deren Eigenschaften zu bearbeiten.

Oberflächenwidgets sind bei der XUL nicht immer $1:1$ als ein Sprachelement umgesetzt. Diese Fälle bedürfen besonderer Beachtung bei der Implementierung eines grafischen modellbasierten Editors. Bei Bäumen, Listen und Tabellenstrukturen ist der Zweck dieses Sprachdesigns offensichtlich, bei anderen konkret „GroupBox“en und „Tabs“ weniger. Die Überschriften der beiden letztgenannten Widgets werden separat, d.h. mit jeweils eigenen Metaklassen, definiert. Beide dieser Metaklassen definieren nur wenige Attribute welche problemlos, ohne Informationsverlust, in der eigentlichen Widget-Hauptklasse hätten definiert werden können.

Im Ecore-Modell könnte diese Herangehensweise genutzt werden, um eine Überschrift einmalig zu definieren, aber von mehreren Widgets benutzen zu lassen. Mit den in XUL benutzten XML-Mechanismen kann dieser Vorteil aus technischen Gründen nicht genutzt werden. Da scheinbar ohnehin eine Sonderbehandlung notwendig werden würde, konnte bei der Umsetzung in XULE ausprobiert werden, inwieweit es nützlich wäre, jeweils die Klasse des Überschriftenelementes und die „Haupt“-Widgetklasse, per Mehrfachvererbung, zu einem einzigen Typ zusammenzuführen.

Für die „Tabs“ wurde die Klasse CombinedTabpanelType im XUL-Metamodell eingeführt, als Unterklasse von TabpanelType und TabType. Damit wurden alle Tab-bezogenen Eigenschaften, wie geplant und erwartet, in einem Typ vereint und bearbeitbar. Leider wird dieser neue Elementtyp von keinem XUL-Renderer, außer dem XULE, interpretiert. Es musste eine aufwändige Einzelfallbehandlung beim Laden und Speichern der Modellinstanzen implementiert werden, die zur Serialisierung wiederum auf die Standardsprachelemente zurückgreift. Im Vergleich mit den „GroupBox“en, bei denen als Vergleichselement zunächst kein spezieller verschmolzener Untertyp erstellt wurde, ergaben sich auch keine signifikanten Erleichterungen bei der Benutzung von XULE. Diese Vereinfachungsidee erwies sich insgesamt als unpraktisch und wurde daher nicht weiterverfolgt.

Mit XULE steht ein standardkonform arbeitender grafischer XUL-Editor zur Verfügung. Wesentliche Teile dieses Werkzeugs wurden modellbasiert entwickelt und sind vergleichsweise einfach an Weiterentwicklungen der XUL anpassbar.

2.2 Transformationstechniken

Modelltransformationen sind der Kern einer modellgetriebenen Entwicklung. Normalerweise werden die denkbaren Transformationen in zwei Klassen eingeteilt. Die texterzeugenden Model-To-Text (M2T) Transformationen erstellen aus Modellinstanzen beliebigen Text. Transformationen zwischen Modellen werden dagegen als Model-To-Model (M2M) Transformationen bezeichnet. Je nach Betrachtungsweise, insbesondere dessen was man als Modell ansieht, kann man M2M-Transformationen als Spezialfall der M2T auffassen.

Im Folgenden werden aktuell verbreitete Modelltransformationstechniken skizziert und deren jeweilige Verwendung im Rahmen dieser Arbeit erklärt. Diese Auflistung erhebt keinen Anspruch auf Vollständigkeit, denn neben den erwähnten Techniken existiert eine schwer überblickbare Anzahl weiterer Techniken.

2.2.1 Model-To-Text

Die Texterzeugung ist eine häufig wiederkehrende Aufgabe in Anwendungssystemen. Oftmals folgen diese Texte einem generellen Muster und müssen für den jeweiligen Adressaten nur an wenigen Stellen angepasst werden. Für diesen Einsatzzweck werden sogenannte *Template Engine*s genutzt. Zwar ist Quantität kein zulässiger Indikator für Qualität, dennoch illustriert die Tatsache, dass in dem Wikipedia-Artikel zum Thema „Template Engine“ [Wik] über 40, der Community öffentlich bekannte, Template Engines erwähnt sind, dass es sich hier um ein typisches wiederkehrendes Problem handelt.

In der modellgetriebenen Softwareentwicklung werden M2T-Transformationen oft genutzt um Quellcode zu erzeugen. Im Fall der MD-UID etwa den Quellcode der endgültigen Benutzungsoberfläche (FUI).

2.2.1.1 XSLT

XSLT, die Extensible Stylesheet Language Transformations, sei hier als erster Vertreter genannt. Dabei handelt es sich um eine deklarative Programmiersprache zur Umwandlung von XML-Dokumenten. Der besondere Vorteil dieser Sprache liegt in ihrer fast flächendeckenden Verbreitung, da nahezu alle modernen Browser XSLT Transformatoren enthalten. XSLT wird durch das World Wide Web Consortium (W3C) standardisiert [Con], aktuell liegt die Version 2.0 vor.

Primärer Einsatzzweck, und eine Reihe von Sprachkonstrukten ist nur hierfür sinnvoll, von XSLT ist die Erzeugung von XML-Dokumenten aus XML-Dokumenten. Während die Eingabedaten stets XML-Dokumente sein müssen, kann XSLT ebenfalls eingesetzt werden um beliebigen Text zu generieren.

XSLT ist für diese Arbeit nur am Rande ein Thema. In einer der Vorstufen des XUL-Metamodells wird diese Transformationstechnik benutzt, um aus einem XML-Dokument eine XML-Schemadeklaration zu generieren. Neben dieser Anwendung, wurde ebenfalls versucht XSLT-Deklarationen zu definieren welche aus JavaDoc Ecore-Klassendeklarationen extrahieren bzw. generieren. Nachdem die prinzipielle Machbarkeit dieses Ansatzes gezeigt war, d.h. aus dem JavaDoc trivialer Klassen konnten äquivalente Ecore-Klassen erzeugt werden, wurde diese Idee nicht weiterverfolgt. Das Vorgehen erwies sich als zu unpraktisch und unflexibel, außerdem sind diverse nützliche Typ-Informationen bereits nicht mehr in den JavaDoc-HTML Dateien enthalten. Statt daher umständlich JavaDoc zu parsen, wurde für diesen Anwendungsfall das in Punkt 4.1.3.1 dargestellte Prinzip des Eingriffs in die JavaDoc-Erzeugung entwickelt.

2.2.1.2 xPand

Die Template-Sprache xPand aus dem openArchitectureWare-Projekt [EFH+] wurde inzwischen ein Kernbestandteil des Eclipse Modeling Project (EMP). Ein typischer Anwendungsfall für xPand ist die Erzeugung von Quellcode aus Modellinstanzen. Im EMP wird xPand zur Erzeugung beliebiger Texte aus Ecore-Instanzen genutzt. Neben der reinen Texterzeugung bietet die Sprache eine Reihe zusätzlicher Mechanismen wie Modellvalidierung, Aspektorientierung und polymorphe Template-Aufrufe.

Auch im Rostocker MD-UID Prozeß wird xPand angewendet. Beispielsweise bei der in Unterabschnitt 2.1.2 gezeigten Erstellung der Mockup-Prototypen aus Dialoggraphen. Das dort angeführte Listing 2.2 vermittelt einen Eindruck der xPand-Syntax.

Für die im Abschnitt 3.3 weiterentwickelte Pattern-Beschreibungssprache PLML wird ebenfalls xPand genutzt. Die xPand-Templates dienen hier dazu, aus den Patterndefinitionen in einer selbst definierten Deklarationssprache, Patternbeschreibungen in einer standardisierten Sprache (PLML) zu generieren. Es existieren verschiedene Templates, die jeweils an die Ziel-Sprachversion von PLML angepasst worden.

Eine weitere Verwendung für xPand ergab sich bei einer gemeinsamen Arbeit mit Brüning [BW09], es ging um den Versuch MD-UID mit deklarativer Workflow-Modellierung zu verbinden. Ein Ergebnis dieser Kombination waren objektorientierte Prozessmodelle, deren Visualisierung über das XUL-Metamodell definiert wurde. Die Generierung der tatsächlichen Benutzungsoberflächen übernahmen bei diesem Ansatz xPand-Templates. Leider traten im Laufe dieser Zusammenarbeit einige Schwierigkeiten im Zusammenhang mit dem Polymorphie-Konzept von xPand zu Tage, unter bestimmten Umständen waren die Template-Aufrufe nicht deterministisch, sondern scheinbar bei jedem Durchlauf zufällig gewählt. Unter anderem auf Grund dieses Problems erfolgte für weiteren Arbeiten der Wechsel zu Acceleo als Standard-M2T Sprache.

2.2.1.3 Acceleo

Auch Acceleo [Obe] gehört inzwischen zu den Kernprojekten des Eclipse Modeling Projects, entwickelt wurde die Sprache ursprünglich von der französischen Firma Obeo. Das Ziel von Acceleo ist die pragmatische Umsetzung des OMG Model-To-Text Standards [Gro] im Sinne einer Referenzimplementierung. Eine vollständige Umsetzung des Standards wird wegen dessen diverser Errata nicht angestrebt.

Grundsätzlich dient Acceleo zur Transformation von Ecore-Instanzen in beliebigen Text. Es ist eng an die OCL [OCL10] gekoppelt, OCL-Terme werden z.B. für Selektions- und Iterationsausdrücke eingesetzt.

Zur Unterstützung des MD-UID Prozess wurden eine Reihe von Acceleo-Transformationen entwickelt. In einer typischen M2T-Anwendung wird die Sprache z.B. genutzt, um aus Instanzen des Swing-Ecoremodells, siehe Unterabschnitt 4.2.1, lauffähigen Java-Quellcode zu erstellen. Auch wurde eine Reihe Grafiken und Tabellen dieser Dissertation mittels Acceleo aus Ecore-Modellinstanzen erzeugt. Eine wichtige Rolle spielt Acceleo auch bei der Erstellung der Patternkataloge im Unterabschnitt 3.5.2. In diesem Prozess werden M2T-Transformationen eingesetzt, um die dynamisch instantiierten PLML-Kataloge korrekt in ihrer tDSL-Form zu serialisieren.

2.2.1.4 Java Emitter Templates

Die Java Emitter Templates [Cora] (JET) sind eine Basistechnologie des EMP. Denn die Generierung des Java-Quellcodes für die Ecore-Modelle, für den Standardeditor und das Testgerüst erfolgen über JET. Die JET sind eine Entwicklung der IBM, die aktuelle Version ist die JET2-Template Engine.

Das Grundprinzip ist ähnlich dem der Java Server Pages: so enthält ein JET-Template statischen Inhalt und

JET-Direktiven. Bei den ursprünglichen JET sind diese Direktiven, bis auf einige wenige Spezialkommandos, normaler Java-Quellcode. Die JET-Engine verwebt die Direktiven mit dem statischen Inhalt, dabei entsteht Quellcode einer Java-Klasse, dieser wird kompiliert und dann für die Eingabedaten ggf. mehrfach ausgeführt. Mit JET2 wurde die Anbindung verwandter Technologien wesentlich vereinfacht, beispielsweise kann nun XPath verwendet werden, außerdem wurden Sprachkonstrukte für regelmäßig auftretende Tätigkeiten wie Iterationen oder Auswahlen eingeführt.

Außer selbstverständlich zur Generierung des Java-Modellquellcodes für die diversen Ecore-Modelle wird JET im Rostocker MD-UID Prozess nicht genutzt. Dies ist insbesondere dem nur schwer realisierbaren Tracing und dem damit sehr umständlichen Debugging geschuldet, d.h. aus rein praktischen Gründen. Die Erfahrungen mit JET entstammen einem Versuch, die internen Klassen des XULE mittels JET-Transformationen über dem XUL-Metamodell zu erzeugen. Dies sollte zunächst durch Anpassen der Standard-Ecore-Templates realisiert werden. Im Erfolgsfall wären die Vorteile gewesen, nicht alles neu implementieren zu müssen und den ausgereiften Build- und Konfigurationsmechanismus des EMP mitbenutzen zu können. Die Komplexität und Wartungsresistenz der Ecore-JET-Templates, in Kombination mit den langwierigen Debugging-Prozessen, führten jedoch zur Einstellung dieses Versuchs. Alternativ wäre es möglich gewesen anstelle der Anpassung der Ecore-JET-Templates mit vollständig selbst entwickelten Templates zu arbeiten. Da aber die Vorteile JETs, insbesondere die Ausführungsgeschwindigkeit bei großen Modellen dank der Compilierung, die genannten Nachteile aus meiner Sicht nicht egalisieren können, kam JET in der Folge nicht weiter zum Einsatz.

2.2.2 Model-To-Model

Wie bereits in der Einführung dargelegt, sind Transformation zwischen Modellen das Wesen einer modellgetriebenen Softwareentwicklung. Dennoch existieren vergleichsweise wenige Transformationssprachen für diesen Zweck. Die vermeintlich bekanntesten Techniken im Umfeld des Eclipse Modeling Projects sind ATL, QVT und Kermeta. Diese werden nachfolgend kurz vorgestellt.

Wenn, wie im EMP üblich, die Serialisierungen der Modelle textuelle XML-Strukturen sind, dann lassen sich Model-To-Text Techniken ebenfalls bis zu einem gewissen Grad als Model-To-Model Transformationen verwenden. Außerhalb des EMP werden oft auch Graphentransformationen für diesen Zweck genutzt. Das in Punkt 1.3.1.1 vorgestellte UsiXML setzt z.B. auf diese Technologie.

2.2.2.1 QVT operational

Der OMG-Standard zu QVT [QVT08] - Query View Transformation - definiert eine hybride, d.h. eine deklarative und imperative, Sprache für Modelltransformationen. QVT als OMG Standard integriert OCL [OCL10] sehr eng. Die Unterebenen QVT Relations (QVTr) und QVT Core (QVTc) bilden den deklarativen Teil von QVT. Während mit QVT Operational (QVTo) im Standard eine imperative Modelltransformationssprache definiert wird. Es existieren Abhängigkeiten zwischen den drei QVT-Sprachteilen, diese sind hier allerdings nicht von Bedeutung.

Zum Einsatz im Rahmen dieser Arbeit und im vorgestellten MD-UID Prozess gelangt ausschließlich QVT Operational.

Hauptsprachbestandteile sind *Mappings*, die eigentlichen Transformationsbeschreibungen, und Queries, OCL-Aufrufe mit Wertrückgaben. Beide werden jeweils wie Funktionen normaler Programmiersprachen deklariert, d.h. mit Funktionsnamen, Eingabe und Ausgabeparametern. Eine QVTo-Transformation beginnt immer mit dem parametrisierten Aufruf eines *Mappings*. Dabei werden die in dem Mapping festgeleg-

ten Objekterzeugungen oder Funktionsaufrufe sequentiell abgearbeitet und das Verarbeitungsergebnis dem Aufrufer zur Verfügung gestellt. Patternmatching und andere deklarative Konzepte sind QVTr vorbehalten und werden von QVTo nicht unterstützt.

Im Rostocker MD-UID Prozess wird QVTo für mehrere Zwecke genutzt. Bei der Erstellung, Wartung und Pflege der Patternkataloge wird das Verweben der Instanzen von PLML mit den Ergänzungen aus den PLMLComponent-Instanzen, siehe Unterabschnitt 3.5.3, mittels QVTo durchgeführt. Eine andere Verwendung ist das Mapping von Swing-Ecore-Instanzen auf XUL-Ecore-Instanzen, eine CUI-To-CUI Transformation. Darüber hinaus können die Implementierungen einiger Patternkomponenten mit QVTo beschrieben werden.

2.2.2.2 Atlas Transformation Language

Die Atlas Transformation Language [JAB+06] (ATL) ist eine französische Entwicklung und entstand als Bestandteil der sogenannten ATLAS Model Management Architecture (AMMA). Auch ATL gehört inzwischen zu den Bestandteilen des Eclipse Modeling Projects. Gemäß der Selbstdarstellung der ATL-Entwickler Jouault et al. in [JAB+06] ist ATL eine QVT-ähnliche Transformationssprache. Beide Sprachen unterscheiden sich zwar in ihrer Syntax, es finden sich jedoch ein Reihe ähnlicher Konzepte.

Besonders erwähnenswert ist ATL auch wegen seines sogenannten „Modell-Zoos“ [Foua]. Einer öffentlich bereitgestellten Sammlung von ATL Transformationen die für eigene Projekte eingesetzt werden können.

Dennoch wird ATL im Rostocker MD-UID Prozess an keiner Stelle für Transformationen benutzt. Das liegt daran, dass eine Auswahlentscheidung zwischen QVT und ATL getroffen werden musste. Weil ATL über keine einzigartigen Funktionalitäten verfügt, die für diese Arbeit nützlich wären, fiel die Wahl auf die offiziell standardisierte Sprache QVT.

2.2.2.3 Kermeta

Das Projekt Kermeta [Ker] stellt neben der Sprache auch noch eine komplette Entwicklungsumgebung bereit. Auch dabei handelt es sich um Eclipse-Plugins, die jedoch ohne offizielle Anbindung an das Eclipse Modeling Project entwickelt werden. Dennoch kann Kermeta Ecore-Modelle verarbeiten.

Kermeta ist zunächst eine textuelle domänenspezifische Sprache zur Deklaration von Metamodellen, deren Syntax diverse Parallelen zu der der Sprache USE [GBR07] hat, USE selbst wurde unter anderem in dem bereits erwähnten Versuch [BW09] zur Verbindung von Workflow und MD-UID Entwicklung genutzt. Die mit Kermeta deklarierten Modelle, konkret die implementierten Operationen der Metaklassen, lassen sich in der Projekt- Umgebung ausführen.

In Unterabschnitt 4.1.4 wird diese Fähigkeit Kermetas benutzt, um mit einer Model-To-Model Transformation das Metamodell von Java Swing zu reduzieren. Ansonsten erfolgte bisher kein weiterer Einsatz von Kermeta im hier vorgestellten MD-UID Prozess.

Zusammenfassung

Das Kapitel 2 stellte alle wesentlichen Modelle, Verfahren und Techniken vor, die im Rostocker Forschungsansatz, eines modellgetriebenen Oberflächenentwicklungsprozesses, relevant sind. Dieser Prozess stellt den konzeptuellen Rahmen der nachfolgend vorgestellten weiteren Arbeiten und ist daher von grundlegender Bedeutung.

Kapitel 3

Patterns und Komponenten

Allgemein versteht man unter Patterns anerkannte, nachgewiesenermaßen erfolgreiche und daher empfohlene Lösungen für regelmäßig auftretende Designprobleme. Der Begriff geht zurück auf Alexander [AIS77], er führte die Idee mit einer Veröffentlichung über Pattern in der Architektur ein. Auch in vielen Bereichen der Softwaretechnik, bzw. der Informatik allgemein, konnten seitdem eine Reihe von Pattern identifiziert werden. Die vermutlich bekannteste Patternsammlung ist die von Gamma et al. [GHJV02] über Patterns in der Objektorientierung, im Folgenden als GoF (Gang of Four) abgekürzt.

Für den Bereich der Human Computer Interaction wurden ebenfalls Patterns vorgeschlagen und gesammelt. Die in dieser Domäne vorgeschlagenen Patterns beschreiben unter anderem „best-practices“ für die Navigation innerhalb von Anwendungen, für das Layout von Oberflächen oder deren Eingabemodalitäten. Manche Autoren sehen auch die Konzeption ganzer Webseiten[1] als so typisch an, dass sie deren Arrangement als Pattern vorschlagen.

3.1 Pattern-Languages

Prinzipiell kann ein Autor jedes wiederholbare Muster und jede nutzbare Vorlage als Pattern bezeichnen. Es gibt keine Normierungskörperschaft für Patterns, einen gewissen legislativen Charakter hat lediglich die Veröffentlichung in einem allgemein anerkannten Pattern-Katalog. Solche Sammlungen von Patterns existieren seit Alexander [AIS77], der diese Idee publik gemacht hat.

Im englischen Sprachraum werden Patternsammlungen oftmals als „pattern language“- Patternsprache - bezeichnet, andere Autoren benutzen den Begriff des Patternkatalogs. Den Unterschied zwischen diesen beiden Sammlungsbezeichnungen arbeiteten Finlay und Dearden in [DF06] heraus. Patternsprachen sind demnach Sammlungen stark vernetzter Pattern, so ergeben sich bei Alexander aus der Wahl eines Patterns sofort Verzweigungen zu anderen Patterns. Patternkataloge erreichen nicht diesen Vernetzungsgrad, Querbeziehungen zwischen Patterns existieren zwar, genügen jedoch nicht den Anforderungen an eine Sprache im Sinne Alexanders. Prominentes Beispiel eines Patternkataloges sind die bereits erwähnten GoF-Patterns.

Es gibt mehrere solcher Kataloge innerhalb der HCI Community. Die beiden bekannten Kataloge sind insbesondere der von Jennifer Tidwell [Tid] und Martijn van Welie’s [vW]. Daneben ist die Existenz eines IBM-internen Katalogs, unter anderem erwähnt in einem Workshop bei der Interact 2005, bekannt. Die Concordia-Universität Montreal führte ebenfalls für eine geraume Zeit einen kleinen Katalog, dieser

[1] vgl. das Pattern „Museum Site“ bei [vW]

umfasste etwas mehr als ein Dutzend Einträge. Insgesamt existieren zum gegenwärtigen Zeitpunkt mindestens fünf öffentliche Patternkataloge für HCI-Patterns. Neben Tidwells und van Welies sind das die Yahoo!-Sammlung [Corb], Anders Texboes [Tox] und der auf Patterns in Spiele-UIs konzentrierte Katalog von Eelke Folmer [Fol]. Im Rahmen dieser Arbeit werden im Weiteren nur die Patternlanguages von Welie und Tidwell betrachtet. Aus zwei Gründen: der Welie-Katalog ist der umfangreichste im World Wide Web; derjenige von Tidwell wurde in Buchform [Tid06] veröffentlicht und die enthaltenen Patterns sind aus meiner Sicht am Besten ausgearbeitet, begründet und präsentiert.

Beide Kataloge sind zum gegenwärtigem Zeitpunkt öffentlich im World Wide Web verfügbar. Wobei der Umfang des Webkatalog der Tidwellschen Patterns jedoch gegenüber der Buchform erheblich verkleinert ist. Im Buch [Tid06] werden 94 Patterns erklärt, der Webkatalog enthält deren 44. Der Unterschied entsteht vor allem dadurch, dass der Fokus des Webkatalogs auf WIMP-Patterns liegt. Wohingegen sich das erste Kapitel „What Users Do“ des Buches sich allgemeinen Mustern der Mensch-Maschine-Kommunikation widmet und dazu zwölf Patterns vorstellt. Beispielsweise das Pattern „Instant Gratification“- die Forderung nach sofortiger Belohnung in Form von Erfolgserlebnissen für den Nutzer. Oder das Pattern „Keyboard Only“ welches eine, zumindest alternative, Bedienbarkeit aller Funktionen einer Anwendung nur mit der Tastatur propagiert.

Alle weiteren Kapitel des Buches entsprechen den Kategorien des Webkatalogs. Dabei werden jeder Kategorie im Buch mindestens drei zusätzliche Patterns zugeordnet. Eine namentliche Auflistung der hinzugefügten Patterns wurde in Tabelle 3.1 vorgenommen. Die Kapitelreihenfolge darin folgt von oben nach unten derjenigen des Tidwell-Kataloges, zu beachten ist dass das Kapitel „Doing Things“ im Webkatalog als „Commands and Actions“ benannt wird und dass das Pattern „Corner Treatments“ zwar im Webkatalog namentlich eingeordnet aber nicht vorgestellt wird.

Der Unterschied im Umfang der beiden Instanzen des Tidwell-Katalogs ist möglicherweise dem kommerziellen Interesse der Vermarktung des Buches geschuldet. Für die in beiden Katalogversionen enthaltenen Patterns ergaben Stichproben eine direkte textuelle Übereinstimmung der Beschreibungstexte und Beispielillustrationen und somit der Patterndeklarationen. Ähnliche Vergleiche mussten für den Welie-Katalog nicht angestellt werden, da er nur in einer Veröffentlichungsform vorliegt.

Untersucht man die beiden maßgeblichen Kataloge vergleichend, fallen eine Reihe Inkonsistenzen und Diskrepanzen auf. So gibt es keine konsistente Benennung oder ein allgemein verbreitetes Namensschema für die enthaltenen Patterns. Das bedeutet unter anderem, dass eine erhöhte Wahrscheinlichkeit für Mehrfacheinträge besteht und - gravierender - dass gleiche Patterns nicht gleich benannt werden.

Kapitel	Patterns
Organizing the Content	Canvas plus Palette, Alternative Views, Multi-Level Help
Getting Around	Hub and Spoke, Pyramid, Modal Panel, Sequence Map, Breadcrumbs, Annotated Scrollbar, Escape Hatch
Organizing the Page	Right/Left Alignment, Diagonal Balance, Property Sheet
Showing Complex Data	Datatips, Dynamic Queries, Data Brushing, Local Zooming, Multi-Y Graph, Small Multiples, Treemap
Getting Data from Users	Structured Format, Autocompletion, List Builder, Same-Page Error Messages
Builders and Editors	Spring-Loaded Mode, Magnetism, Guides, Paste Variations
Making It Look Good	Corner Treatments, Borders That Echo Fonts, Hairlines, Contrasting Font Weights, Skins

Tabelle 3.1: Nicht im Webkatalog [Tid] enthaltene Patterns des Tidwell-Katalog [Tid06]

Die Beziehungen zwischen Patterns werden fast nur katalogintern angegeben und es existiert überhaupt keine Referenz zum jeweils anderen Katalog. Überhaupt unterscheiden sich beide Kataloge signifikant in der Häufigkeit der Angabe von Patternbeziehungen. Details dazu werden in Unterabschnitt 3.5.2 vorgestellt.

Leider unterscheiden sich die Kataloge auch in den spezifizierten Aspekten der jeweiligen Patterns. So unterläßt Welie [vW] eine explizite Problembeschreibung, während Tidwell [Tid] die Musterlösung des Patterns uneinheitlich verteilt über eine „Example"-Sektion und einen „How"-Block vorstellt.

Weiterhin variiert das Abstraktionsniveau der Patterns erheblich. So schlägt Welie, für eine Webseite, das Pattern „Fun" vor, mit der Problembeschreibung: „Users want to have fun or feel entertained"[2]. Ein anderes Pattern des gleichen Kataloges ist „Action Button". Darin geht es darum, dass Schaltflächen die wichtige Aktionen auslösen, in ihrer Beschriftung die auszulösende Aktion in ihrer Verbform enthalten sollen. Beides kann unter Umständen als Pattern angesehen werden, die jeweilige Lösung muss jedoch auf völlig verschiedenen Ebenen der Softwareentwicklung umgesetzt werden. Hier handelt es sich um einen bemerkenswerten Unterschied zu den GoF-Pattern, deren Anwendungsebene immer Klassendiagramme sind.

Die Patternlanguages von Tidwell [Tid06] und Welie verfolgen hauptsächlich das Ziel, Entwicklern und Designern die präsentierten HCI-Patterns zur Kommunikation zu vermittlen. Ziel ist es, dass alle Stakeholder im Entwicklungsprozess unter einem Patternnamen das Gleiche verstehen. Falls möglich, sollen die eigentlichen Designideen durch Kombination von Patterns ausgedrückt werden. In allen Fällen handelt es sich um textuelle Beschreibungen, zu Demonstrationszwecken meist unterstützt durch illustrierende grafische Darstellungen. Diagramme in standardisierten Notationen werden nicht verwendet. In der Konsequenz liegt die vorgeschlagene Patternlösung immer nur informal definiert vor. Auch hierin liegt ein erheblicher Unterschied zum Katalog der GoF-Pattern von [GHJV02], dort wird für jedes Pattern ein Klassendiagrammen und Beispielquellcode angegeben.

Selbstverständlich sind diese Beobachtungen zu den Patternkatalogen keine neuen Erkenntnisse. Es existieren mindestens zwei Ansätze zur Reduzierung der Inkonsistenzen durch Formalisierung. Ein wesentlicher Formalisierungsschritt ist die Anwendung einer standardisierten Pattern-Language für Patternkataloge.

Genau dafür wurde im Rahmen eines CHI-Workshops das XML-Derivat Pattern Language Meta Language PLML [Fin03] geschaffen. Darüberhinaus wurde um 2005 das TROST [Ham]-Pattern Description Template veröffentlicht. Es entstand aus vergleichenden Betrachtungen der Alexander, GoF und weiteren Patternsammlungen und ist letztlich eine Sammlung von Stichworten oder Aspekten die ein Pattern beschreiben. Die Unterschiede zu PLML sind nicht gravierend, bemerkenswert ist, dass auch Themen wie Usability, Testbarkeit, Support, Sicherheit und weiteres in den TROST-Beschreibungen berücksichtigt werden.

Für meine eigene Arbeit setzte ich auf PLML auf, da hiervon auch Berichte über den tatsächlichen Einsatz vorliegen.

Die Entwicklung von PLML begann mit dem Ziel eine gemeinsame Basis zur Spezifikation von Patterns zu schaffen. Darüberhinaus sollte es als Austauschformat zur Übertragung von Pattern zwischen verschiedenen, möglicherweise spezialisierten, Katalogen dienen.

Die Sprache PLML wurde darauf ausgelegt, Patterns für jeden Anwendungsbereich zu beschreiben, ihre Spezifikation sieht keine Sprachelemente speziell für HCI-Patterns vor. Der GoF-Patternkatalog könnte daher mit PLML beschrieben werden, mit einer entscheidenden Abwandlung; Quellcode und UML-Diagramme sind in PLML nicht formalisiert. Ohne eine Beschränkung auf eine bestimmte Anwendungsdomäne war es naturgemäß nicht möglich, für den Wertebereich der Sprachelemente Vorgaben zu machen.

[2] d.h. Pattern „Spaß"←Besucher wollen Spaß haben oder unterhalten werden

Die Konsequenz daraus ist, dass mit PLML das Problem der unzureichenden Formalisierung der Patternlösung nicht behoben werden kann.

Es wäre unzutreffend zu behaupten, dass PLML keine Verbreitung fand, immerhin stellen fünfzig Prozent der öffentlichen Patternkataloge ihren Inhalt auch über PLML bereit. Von einem regen Gebrauch kann allerdings ebenfalls nicht die Rede sein, zumal die Diskrepanzen zwischen dem Webkatalog- und dem PLML-Eintrag beim Welie-Pattern „Send-a-Friend Link“[3] darauf hindeuten, dass Welie nicht PLML als Backend für seinen Patternkatalog verwendet.

PLML wird im Rahmen dieser Arbeit verwendet, um den hiesigen Pattern-Katalog zu verwalten. Der Abschnitt 3.2 gibt einen Überblick über die Sprache PLML so wie sie zunächst spezifiziert wurde. Dabei werden diverse Unzulänglichkeiten des Sprachstandards aufgezeigt. Deren Behebung und eigene Erweiterungen werden vorgeschlagen und die daraus resultierende neue Sprachversion präsentiert. Mit dieser neuen Version von PLML wurde ein vereinheitlichter Patternkatalog, als Kombination von Welie und Tidwell angereichert um eigene Überlegungen, aufgebaut.

Der Katalog selbst wird mit einer textuellen domänenspezifischen Sprache gepflegt. In Abschnitt 3.3 wird deren Aufbau und Anwendung gezeigt. Der vereinheitlichte Patternkatalog ist kein Selbstzweck. Die darin enthaltenen Patterns wurden daraufhin untersucht, inwieweit es möglich ist, für die jeweiligen Patternlösungen Algorithmen als Model-To-Model oder Model-To-Text Transformation anzugeben. Der Abschnitt 3.5 stellt die Vorgehensweise und die dabei gewonnenen Erkenntnisse, also das Ergebnis dieser Untersuchungen, dar.

3.2 Pattern Language Meta Language

3.2.1 Sprachstandard

Die Pattern Language Meta Language PLML [Fin03] wurde 2003 durch Fincher, Tidwell, Welie et al. im Rahmen eines Workshops auf der CHI'2003 entwickelt. Ziel war die Formalisierung von Patternbeschreibungen, um auf diesem Weg alle damals existierenden HCI-Patternkataloge zu einem verteilten Katalog zu verschmelzen oder zumindest ein universelles Datenaustauschformat für diese Kataloge zu schaffen. Das Sprachdesign folgt der Vorgabe, mit PLML Patterns auf jeder Abstraktionsebene und aus jeder Domäne beschreiben zu können.

Neben Angaben zur Spezifikation der Patterns ist es in einem Katalog wünschenswert, auch diverse Metadaten zu den Pattern-Einträgen zu notieren. Beispiele dafür sind die Namen der Autoren und Versionierungsinformationen.

Da PLML nicht offiziell standardisiert ist, muss der Workshop-Report [Fin03] als die wesentliche verbindliche Sprachdefinition dienen. Geeignet dafür ist dieser Bericht auch deswegen, weil er neben der reinen PLML-Definition auch Erläuterungen zu den Sprachelementen liefert.

```
<!ELEMENT pattern (
    name?, confidence?, alias*, synopsis?, illustration?,
    context?, problem?, forces?, evidence?, solution?, diagram?,
    implementation?, related-patterns?, pattern-link*,
    literature?, management?)>
<!ELEMENT management (
    author?, revision-number?, creation-date?, last-modified?,
```

[3] Patternidee: Link anbieten, damit Nutzer einen Freund per Email direkt über etwas informieren können.

```
    change−log?, credits?)>
<!ATTLIST pattern
  patternID CDATA #REQUIRED
  collection CDATA #REQUIRED
>
<!ATTLIST context mylabel CDATA #IMPLIED>
<!ATTLIST pattern−link
  type CDATA #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #REQUIRED
  label CDATA #REQUIRED
>
```

Listing 3.1: DTD des PLML-Standard [PLM]

Listing 3.1 zeigt den Kern des Sprachstandards als Document Type Definition, in seiner offiziellen Version 1.1.2. Für sämtliche Elemente, d.h. XML-Tags, die in Listing 3.1 referenziert werden, aber nicht weiter spezifiziert sind, gilt als Typdefinition: #PCDATA oder ANY. Die Tabelle 3.2 gibt eine Erläuterung zu den Sprachbestandteilen, zu jedem Element ist eine Kurzbeschreibung angegeben. Die im Element *management* gekapselten Metadaten sollen wegen ihres selbsterklärenden Charakters hier nicht genauer erläutert werden.

Element	**Description**
patternID	Katalogweit eindeutige Id eines Pattern
name	Name, so kurz wie möglich
alias	Bekannte Namensalternativen
illustration	Grafische Darstellung einer besonders überzeugenden Pattern-Instance
problem	Designproblem welches das Pattern löst
context	Bedingungen unter denen die Patternanwendung am nützlichsten wird.
forces	Zwänge die durch die Anwendung des Patterns gelöst werden.
solution	Anweisungen deren Befolgung die Pattern-Idee umsetzen.
synopsis	Zusammenfassung des Patterns
diagram	Schematische Visualisierung des Pattern, skizzenhaft oder formal
evidence	Nachweis das es sich tatsächlich um ein Pattern handelt durch entweder:
example	- Bekannte Verwendungen
rationale	- Prinzipielle Überlegungen, Axiome, Gesunder Menschenverstand o.ä.
confidence	Sternwertung (0 bis 2 Sterne), ob der Eintrag tatsächlich als Pattern besteht
literature	Referenzen auf vergleichbare Arbeiten oder Pattern
implementation	Quellcode, oder Fragmente dessen, oder sonstige technische Dokumentation
related-pattern	Container-Element für Verbindungsdefinitionen zu anderen Pattern
pattern-link	Assoziation zu einem anderen Pattern, im Detail:
type	- Art der Verbindung, entweder *is-a, is-contained-by, contains*
patternID	- Verbindungsendpunkt
collectionID	- ggf. in anderem Katalog
label	- Beschreibendes Label

Tabelle 3.2: Kurzübersicht der Semantik der PLML-Sprachelemente [Fin03]

Bei genauerer Betrachtung der PLML-DTD fällt auf, dass es keine Vorgaben für die zugelassenen Inhal-

te der patternbeschreibenden Elemente gibt. Dieser dadurch erreichte hohe Freiheitsgrad für die Patterneinträge, entspringt der Absicht eine breitestmögliche Anwendbarkeit zu sichern. Der standardsetzende Workshop-Bericht [Fin03], erwähnt allerdings eine Reihe von Einschränkungen, z.B. für den Wertebereich des Confidence-Ratings, die jedoch nicht in die DTD des Standards eingeflossen sind. Im anschließenden Unterabschnitt 3.2.2 werden unter anderem diese zusätzlichen Erläuterungen in die Sprache eingeführt, sowie andere Erweiterungen die sich aus den Erfahrungen im praktischen Einsatz ergaben, integriert.

3.2.2 Erweiterungsvorschläge

Der Versuch des Erstellens eines Pattern-Katalogs unter Verwendung der PLML zeigt schnell ein erstes Problem auf. Der Fokus der PLML liegt auf dem einzelnen Pattern, an eine umschließende Verwaltungsentität wurde nicht gedacht. Mit anderen Worten, es ist kein Wurzelelement vorgesehen, welches als Container für die Einzelelemente dient. Per Definition können daher auch keine Metadaten über den aufzubauenden Katalog hinterlegt werden, weder Autoren, Versionsinformationen, Änderungszeitpunkte noch nicht einmal eine eindeutige Katalog-ID kann notiert werden. Dies ist in gewisser Weise erstaunlich, da für Pattern-Querreferenzierungen explizit die Angabe des Zielkatalogs, in welchem das Pattern definiert ist, über dessen ID gefordert ist.

Die erste DTD-Erweiterung aus Listing 3.2 führt diese Basis-Funktionalität in PLML ein. Das neue Element catalog dient künftig als Katalog-Wurzelelement. Das dringend notwendige Attribut zur Fixierung einer eindeutigen Katalog-ID wurde bei dieser Gelegenheit ebenfalls eingeführt.

```
<!−− Katalogelement als Minimalinvasive Änderung von PLML v1.1.2 −−>
<!ELEMENT catalog (pattern∗)>
<!ATTLIST catalog
  id CDATA #REQUIRED
>

<!−− Einsatzversion des Katalogelementes in PLML v1.5 −−>
<!ELEMENT catalog (name?, management, category∗, pattern+)>
<!ATTLIST catalog
  id CDATA #REQUIRED
>
```

Listing 3.2: Einführung eines Elementes für Katalog-Metadaten

Im Listing 3.2 wurden zwei verschiedene Definitionen für catalog angegeben. Die Erste, jene für v1.1.2, ist minimal gehalten, um weiterhin so nah wie möglich am Standard zu bleiben. Die mit dem Terminus Einsatzversion gekennzeichnete untere Definition ist etwas ausführlicher gehalten. Sie ermöglicht es, zusätzlich zur Katalog-Id eine Katalogbezeichnung als name anzugeben und Katalogmetadaten zu hinterlegen. Darüberhinaus lassen sich Kategorien zur kataloginternen Klassifikation von Einzelpatterns definieren.

Eine kleine Verschärfung oder Präzisierung der originalen Sprachdefinition liegt in der Bewertung der confidence. Also dem Vertrauen darin, dass es sich bei dem jeweiligen Eintrag im Patternkatalog tatsächlich um ein allgemein akzeptiertes Pattern handel. Laut Workshop-Report soll diese Bewertung mit einer Sternwertung vorgenommen werden.

Dieses Rating ist der allgemeinen Patternbeschreibung von Alexander [AIS77] entlehnt. Ebenfalls von Alexander stammt die Festlegung des Wertebereichs für die confidence: $**$ $\rightarrow$ sicher, $*$ $\rightarrow$ vieleicht, sowie 0 $\rightarrow$ Patternstatus unklar. Es ist mit wenig Aufwand möglich, diese Wertausprägungen im Standard zu definieren. Details dazu hält Unterabschnitt C.3.1 bereit.

In den Erläuterungen zur PLML wird oft Bezug auf Abbildungen genommen, welche als Beschreibungmittel diverser Aspekte von Pattern verwendet werden sollen. Die Sprachdefinition selbst wurde jedoch mit unzureichend wenig Unterstützung zur Einbettung von Grafiken oder Bildreferenzen vorgenommen. Änderungsvorschläge zu diesem Aspekt führt Punkt C.3.1.1 auf.

Für die Einträge des Patternkatalogs sollten, wiederum laut PLML-Standard, Literaturreferenzen angegeben werden. Unter anderem um jeweils die Existenz und Validität eines Patterns zu untermauern. Ich fand es daher geboten, auch für Literaturreferenzen einen minimalen formalen Rahmen vorzugeben, definiert und erläutert in Punkt C.3.1.2.

Gemäß dem Originalstandard ist die Notation von pattern−links im Content beinahe jedes XML-Elementes zulässig. Welie [vW] macht auch regelmäßigen Gebrauch davon, in dessen Katalogs finden sich pattern−links unter anderem bei den Beispielen und auch den Illustrationen. Das entspricht meiner Meinung nach nicht unbedingt dem Sprachstandard und wichtiger auch nicht der Idee des pattern−link wie in Tabelle 3.2 erläutert. Meiner Meinung nach geht es darum, explizit semantische Zusammenhänge zwischen Pattern zu deklarieren. Die Verwendung der pattern−links bei Welie zur Querverweisung von Beispielen, ist etwas anderes. Es wurde daher entschieden, pattern−links nur noch als Unterelemente von related−pattern zuzulassen. Auch die Implementierungsdetails dieser Änderung finden sich im Anhang bei Punkt C.3.1.3.

Neben den drei bereits in Tabelle 3.2 vorgestellten Relationsarten werden an dieser Stelle drei weitere Verbindungstypen in die PLML eingeführt. Das sind uses, comparable und contradicts. Die Semantik von uses ist die Aussage, dass Patterninstanzen des referenzierten Zielpatterns von der Lösung des aktuellen Patterns verwendet werden. Patternbeziehungen vom Typ contradicts identifizieren Patterns deren Lösung konträr zu der Lösung des referenzierten Patterns ist. Ein Beispiel ist das Pattern „Input Prompt“[4] welches gewissermassen das Gegenteil des Patterns „Good Defaults“[5] fordert. Die Benutzung des Markers comparable verdeutlicht, dass die Lösung(-sstrategie) des einen Patterns vergleichbar dem des jeweils anderen ist.

Je mehr Einträge ein Katalog hat, desto sinnvoller wird es, die enthaltenden Patterns Kategorien zuzuweisen. Sowohl Tidwell als auch Welie ordnen ihre Patterns in Kategorien. Welie verwendet eine 2-stufige Kategorisierung, Tidwell beläßt es bei einer Ebene. Das mag daran liegen, dass der Welie-Katalog etwa drei mal soviele Einträge enthält wie derjenige von Tidwell.

Der Sprachstandard von PLML sieht keine Kategorisierung von Patterns vor. Auch dies kann möglicherweise der Pattern-zentrierten Denkweise zugeschrieben werden, auf Grund jener bereits auf ein umschließendes Katalog-Element verzichtet wurde. Warum auch immer diese Entscheidung getroffen wurde, für den Einsatz von PLML für real existierende Kataloge ist die Einordnung und Gruppierung von Patterns in einer geeigneten Klassifizierungshierarchie sehr sinnvoll. Die notwendigen Anpassungen der PLML-DTD erläutert Punkt C.3.1.4.

3.3 Textuelle domänenspezifische Sprache für PLML

Patternkataloge, die mit PLML zusammengestellt werden, liegen in XML-Form vor. Für ein Speicher- und Datenaustauschformat ist das eine geeignete Form. Die Definition, Erweiterung und Wartung eines PLML-Patternkatalogs verlangt daher ebenfalls das Operieren auf XML-Baumstrukturen. Dies ist bei größeren XML-Strukturen allerdings unpraktisch und fehlerträchtig. Zur Bearbeitung von PLML-Katalogen durch menschliche Designer sind bessere Möglichkeiten denkbar.

[4] Erklärende Texte als Vorgabewerte in Textboxen oder Dropdown-Feldern die den Nutzer darauf hinweisen was er eingeben oder auswählen soll

[5] Sinnvolle Standardwerte vorauswählen

Es wäre selbstverständlich in jedem Fall möglich, eine eigene Applikation zur Katalogwartung zu implementieren. Davon abgesehen kann mit bereits existierenden Werkzeugen viel erreicht werden. Die einfachste Möglichkeit wäre es sicherlich, spezialisierte XML-Editoren zu verwenden. Im Zusammenspiel mit der DTD sind derartige Tools durchaus in der Lage, einigen Komfort zu bieten. Dazu gehören Syntaxvervollständigung, Templates oder auch Drop-Down Menüs zur Wertauswahl bei den Attributen deren Wertebereich durch Entitäten eingeschränkt ist. Leider können diese Werkzeuge keine Unterstützung bei Querreferenzierung bieten, also etwa bei pattern−links die Gültigkeit der Ziel-ID testen oder Ziel-IDs automatisch vervollständigen. Dies ist technologisch bedingt, die dazu notwendigen semantischen Informationen können in einer DTD-Grammatik schlicht nicht definiert werden.

Anders sieht dies bei XML-Schemata aus, diese XML-Modellbeschreibungen würden die notwendigen Informationen enthalten. Allerdings müsste ein entsprechendes Schema definiert werden. Auch dies wäre eine manuelle Aufgabe, denn ein XML-Schema kann nicht aus der DTD generiert werden. Beziehungsweise kann man nur ein Schema-Gerüst daraus generieren, die fehlenden semantischen Informationen müssten ja weiterhin aus einer anderen Quelle kommen.

Eine solche Quelle könnte eine andere Grammatik, ein Ecore-Metamodell oder eine Kombination aus beidem sein. Dies und mehr bietet das xText-Framework [EV].

xText, eine Entwicklung von Itemis [Ite], ist inzwischen ein Bestandteil des Eclipse Modeling Project, dort ist es das wesentliche Textual Modeling Framework. Vom xText-Framework werden, ausgehend von einer Grammatik, Metamodelle und Editoren generiert. Die Grammatik beschreibt dabei die Syntax einer textuellen domänenspezifischen Sprache (tDSL). Die Sprachgrammatik beschreibt eine Programmiersprache für einen dedizierten Einsatzzweck. Das xText-Framework generiert eine komplette Einsatzumgebung, als Eclipse-Plugins, für die neue Sprache.

Die Einsatzumgebung umfasst insbesondere das Ecore-Modell der Sprache und einen Texteditor mit einer Reihe Komfortfunktionen. Es ist nicht notwendig den erstellten Texteditor zur Bearbeitung von Dateien der tDSL zu benutzen. Jeder beliebige Dateieditor kann genutzt werden.

Der Zweck einer eigenen domänenspezifischen Sprache ist es Sachverhalte präziser und prägnanter als mit generischen Formalismen möglich zu fixieren. Es geht im allgemeinen aber nicht nur darum eine eigene Notation zu haben, sondern die damit beschrieben Sachverhalte auch weiterverarbeiten zu können. An dieser Stelle ist das aus der Grammatik erzeugte Metamodell von Bedeutung. Bestandteile des xText-Frameworks parsen die Textdateien der tDSL und stellen den Dateiinhalt als Instanzen des Metamodells der tDSL dar. Diese Metamodelleninstanzen lassen sich mit den üblichen Model-To-Model und Model-To-Text Technologien weiterbearbeiten.

Die Definition einer tDSL für PLML mit xText hat eine Reihe von nützlichen Konsequenzen. Die vermutlich Praktischste ist der Texteditor. Dank der, in der xText-Grammatik, vorhandenen Semantik der ID-Referenzen, kann dieser Texteditor die Gültigkeit der Referenzen testen, bietet Autovervollständigung, Syntax-Hervorhebung und eine anklickbare Baumübersicht des Kataloges. Alternativ wäre es möglich, EMF aus dem PLML-Ecore-Metamodell ein XML-Schema generieren lassen und dieses in externen XML-Editoren zu verwenden. Je nach Reifegrad des eingesetzten XML-Werkzeuges ließe sich dabei eine vergleichbare Funktionalität, dann auf XML-Basis, erhalten.

Wie erwähnt, lassen sich über dem Metamodell beliebige Transformationen durchführen. Das xText-Framework kann dabei unterstützend tätig werden, indem es ein eigenes Generatorprojekt anlegt. Transformationen mit diesem Generator werden über die Modeling Workflow Engine (MWE) gesteuert. Als Transformationstechnik ist standardmäßig xPand, und damit zunächst eine Model-To-Text Transformation, vorgesehen. Natürlich ist dies konfigurier- und änderbar. Sowohl MWE als auch xPand sind Itemis [Ite]-

Entwicklungen.

Zur Erstellung von xPand-Templates, die dann in einem Generatorprojekt genutzt werden können, gibt es von xText in der aktuellen Version keine weitere Unterstützung, abgesehen von den üblichen Bearbeitungshilfen durch den xPand-Editor. Zwei wesentliche Templates zur PLML-Verarbeitung wurden entwickelt. Diese generieren aus den mit der tDSL beschriebenen PLML-Katalogen, eine der Sprachdefinition entsprechende XML-Version des Kataloges. Es existiert jeweils ein xPand-Template zur Erstellung des Patternkataloges nach dem Standard der offiziellen PLML-Version und ein Template für das in der Arbeit erweiterte PLML v1.5.

Transformationen außerhalb des Generator-Projektes sind selbstverständlich auch möglich. So entstanden Abbildung 3.7 und Abbildung 3.8 aus Unterabschnitt 3.5.2 als Model-To-Text Transformationen mit Acceleo, siehe Punkt 2.2.1.3. Hier wurde aus der jeweiligen PLML-Kataloginstanz der LaTeX-Quellcode für die Grafiken generiert.

Listing B.4 auf Seite 145 zeigt einen mit der PLML-tDSL spezifizierten Beispielkatalog. Das abgebildete Listing ist ein Ausschnitt aus einem grösseren Katalog. Das Original ist die nach PLML umgewandelte Form des Webkatalogs von Martijn van Welie, von dem in diesem Beispiel nur zwei Einträge gezeigt werden. Vollständig abgebildet sind die Katalog-Metadaten und die Kategorie-Definitionen. Der erste Eintrag dieses Beispielkatalogs ist semantisch fehlerhaft, so fehlen z.B. die Problembeschreibung und einiges mehr. Dennoch illustriert das Listing B.4 die Referenzierungsmechanismen und die Einbettung von HTML für Bildverlinkung und Textformatierung.

Die vollständige PLML-tDSL Grammatik von xText ist im Anhang auf Seite 143 angegeben. Darüber hinaus finden sich in Unterabschnitt C.3.2 Detailerklärungen zu einem Ausschnitt dieser Grammatik. Das daraus vom xText-Framework erzeugte Metamodell wird in Abbildung 3.1 und Abbildung 3.2 dargestellt. Die Aufteilung in die Teilgrafiken wurde nach inhaltlichen Gesichtspunkten vorgenommen, alle Metaklassen sind Bestandteil eines Modells; insbesondere ist die in beiden Abbildungen enthaltene Klasse Pattern dieselbe. Weil es sich dabei um ein Ecore-Modell handelt, werden die Basistypen EString und EBoolean genutzt, und nicht deren UML::$*$-Entsprechungen.

In Abbildung 3.1 werden die Klassen des PLML-Metamodells dargestellt, die der Strukturierung sowie der Notation von Stammdaten des Katalogs dienen. Instanzen von Model bilden die Wurzel eines instanziierten Kataloges. Im Typ MasterData werden die Referenzen auf die typischen Stammdateneinträge vorgehalten. Entsprechend der Definition in der Grammatik sind einige davon, via Kardinalitäten, als Pflicht- oder Optionalangaben definiert. Über die Assoziation subCategory:CategoryDef läßt sich eine beliebig tiefe Kategoriehierarchie erstellen.

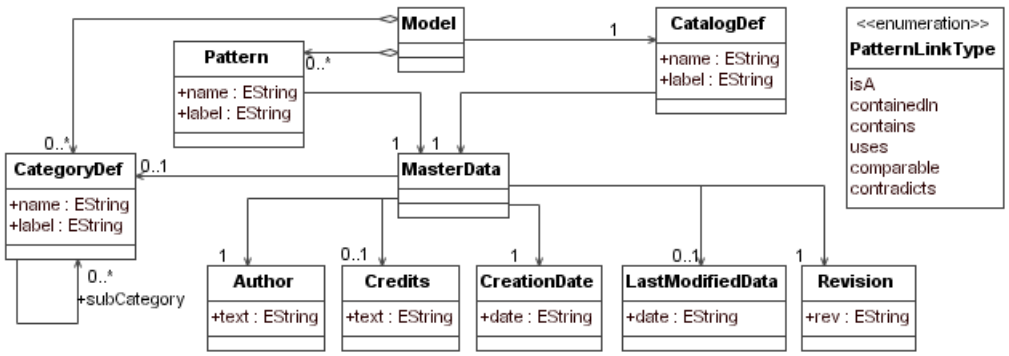

Abbildung 3.1: Hilfsklassen im PLML-Metamodell

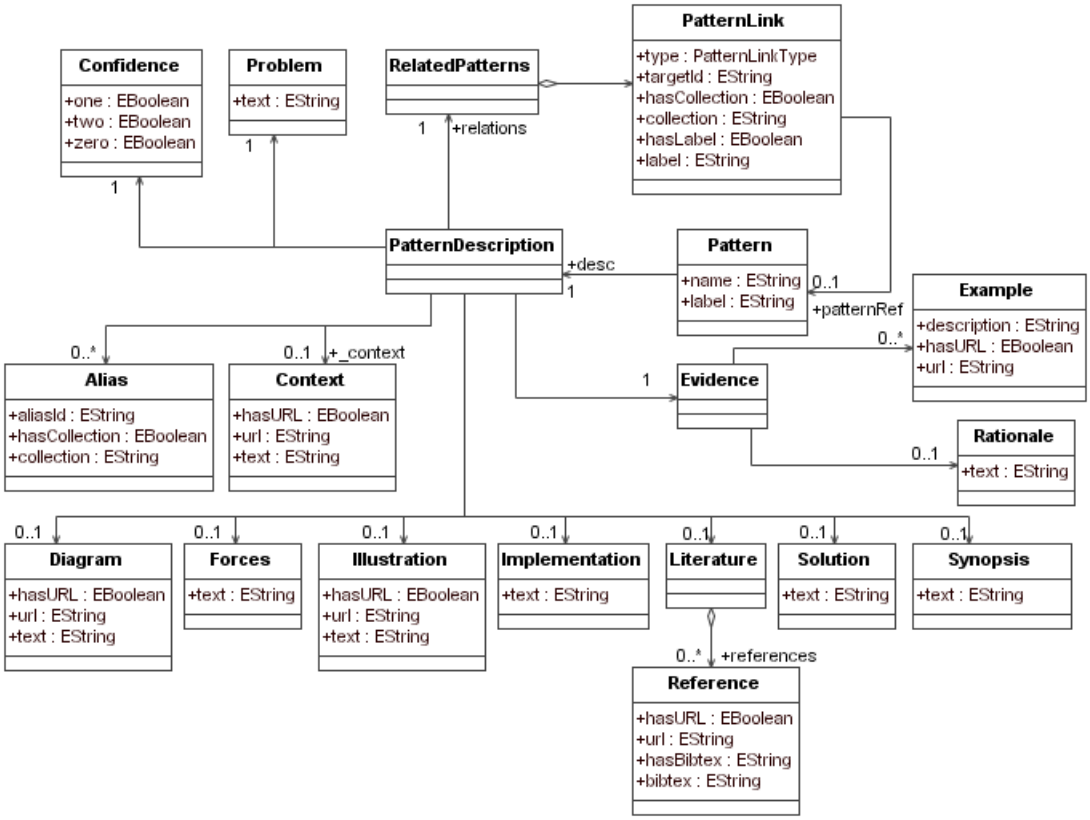

Abbildung 3.2: Kern des PLML-Metamodells, Deklaration eines Patterneintrags

Die für die Beschreibung der einzelnen Patterns genutzten Klassen sind in Abbildung 3.2 dargestellt. Jeder Aspekt eines Patterns wird in einer eigenen Metaklasse beschrieben. Der Name der Klasse leitet sich letztlich vom ursprünglichen PLML-Elementnamen ab und folgt nicht hundertprozentig den UML-Namensgebungsrichtlinien. Referenziert wird dies alles von PatternDescription-Objekten aus, jedes Pattern hat genau eines davon. Im oberen rechten Teil von Abbildung 3.2 sind die Metaklassen, die bereits in der Diskussion zu Listing C.7 erläutert wurden, abgebildet.

Dieses Metamodell für PLML weist eine Reihe Schwächen auf. Als menschlicher Designer würde man ein solches Modell eher nur in einer frühen Phase der objektorientierten Analyse modellieren. Optimierungspotential bietet zum Beispiel PatternDescription, die hier definierten Assoziationen könnten ohne Schwierigkeiten innerhalb Pattern definiert werden. Ebenso scheint es nur mäßig sinnvoll, sechs Klassen die jeweils nur aus dem Attribut data:EString bestehen zu definieren. Auch diese Information ließe sich ohne Semantikverlust in Pattern integrieren. Darüberhinaus sind sämtliche has$*$-EBoolean Attribute überflüssig, und die Modellierung der Confidence-Klasse scheint umständlich. Für die Stammdatensektion lassen sich ähnliche Kritikpunkte aufzählen.

Es handelt sich also weniger um ein geeignetes Metamodell für beliebige Patternkataloge, sondern *nur* um ein praktikables Metamodell für PLML. Trotz dieser teilweise unglücklichen Modellierung wurde an diesem Metamodell festgehalten, um nicht in die Automatismen von xText eingreifen zu müssen. Da die vorgeschlagenen alternativen Modellierungen keine Semantikänderung bewirken würden, sind die möglichen negativen Konsequenzen der Beibehaltung des suboptimalen Modells vernachlässigbar gering und schienen weniger gravierend als auf die Vorteile der Anwendung der xText-Automatismen zu verzichten.

3.4 Grafische Darstellung von Pattern-Katalogen

Bei der Benutzung der tDSL für PLML zeigte sich schnell, dass diese Art der Katalogbearbeitung dem direkten Editieren von XML-Strukturen überlegen ist. Da es nun eine textuelle Notation gibt, kann vergleichend untersucht werden, welche Vorteile eine grafische Notation erbringen könnte.

Offenbar zieht eine erhebliche Anzahl Menschen grafische Modellierungsmöglichkeiten gleichwertigen Textuellen vor. Andererseits argumentierten bereits Grönninger et al. [GKR+07] gegen die These, dass grafische Editoren und Notationen „besser sind weil sie grafisch sind“. Sie identifizierten als Vorteile der Benutzung von textuellen Notationen für DSLs: Skalierbarkeit, einfache Benutzung sowie die vergleichsweise einfache Wiederverwendbarkeit der damit erstellten Artefakte für andere Zwecke.

Grafische Bearbeitungswerkzeuge sind naturgemäß aufwändiger zu entwickeln als textuelle. Es schien daher nützlich, die Idee der Machbarkeit eines grafischen PLML-Editors im Vorhinein geeignet zu prüfen, bevor Ressourcen durch eine entsprechende eigene Entwicklung gebunden würden.

Wünschenswert ist darüber hinaus eine auf dem PLML-Metamodell aufsetzende Editor-Entwicklung, um den Aufwand für Pflege und Wartung bei Modelländerungen minimal zu halten. Eine technologisch sinnvolle Möglichkeit, für die Entwicklung eines prototypischen grafischen PLML-Editors auf Basis des Metamodells, ist die Verwendung des Graphical Modeling Framework (GMF) ins Spiel. GMF ist Bestandteil des Eclipse Modeling Project, es handelt sich um ein Framework zur modellgetriebenen Erzeugung grafischer 2D-Editoren für beliebige Ecore-Metamodelle. Diese generierten Werkzeugen werden als Diagramm-Editoren bezeichnet. Das heisst insbesondere, dass eine graphenbasierte Darstellungsmethapher zum Einsatz kommt. Damit sind die Grundelemente dieser Editoren Knoten und Kanten.

Bei dem GM-Framework selbst handelt es sich um eine Reihe von Metamodellen und Generatoren. Die einzige Verwendungsanforderung für das GMF ist, über ein Ecore-Modell der Zieldomäne zu verfügen. Da dies für PLML zutrifft, schließlich existiert dank xText ein PLML-Metamodell als Ecore-Modell, konnte die Idee der grafischen PLML-Bearbeitung mit GMF umgesetzt und getestet werden.

In einem Ecore-Metamodell liegt typischerweise keine Information darüber vor, wie dessen Instanzen grafisch dargestellt werden sollen. Diese Information muss daher dem GMF zusätzlich zur Verfügung gestellt werden.

Darüber hinaus ist zu beachten, dass es sich bei dem generierten Editor um eine eigenständige RCP[6]-Anwendung handelt. Demzufolge müssen, neben der reinen grafischen Anzeige, auch noch Menüs und Bearbeitungsaktionen bereitgestellt werden. Auch deren Ausgestaltung muss GMF aus anderen Quellen als dem darzustellenden Metamodell bekannt gemacht werden.

Die benötigten zusätzlichen Informationen werden dem GMF über eine Reihe verschiedener Modelle bereitgestellt. Die Metamodelle dieser Hilfsmodelle sind als Ecore-Modelle spezifiziert, ihre Instanzen lassen sich daher mit den Standardmechanismen des EMP bearbeiten. Dies ist jedoch nur in Grenzen empfehlenswert. Die Instanzen dieser Hilfsmodelle übersteigen regelmäßig die Größe des ursprünglichen darzustellenden Metamodells, [Sch08] nennt Faktor 10-20 als Größenordnung. Daher stellt das GMF selbst eine Reihe Wizards bereit, die zumindest bei der Erstellung der Grundstruktur der Hilfsmodelle behilflich sind.

Einen Übersicht über die (Teil)-Modelle im GMF-Prozess gibt Abbildung 3.3. Durchgezogene Pfeile kennzeichnen diejenigen Modelle die von dem Modell an der Pfeilspitze benutzt werden. Gestrichelte Pfeile kennzeichnen Transformationen, in der Prozessdarstellung sind zwei Transformationen enthalten. Erstens die Model-To-Model Transformation zwischen `Mapping` und `Editor Generation`-Modell

[6] Rich Client Platform

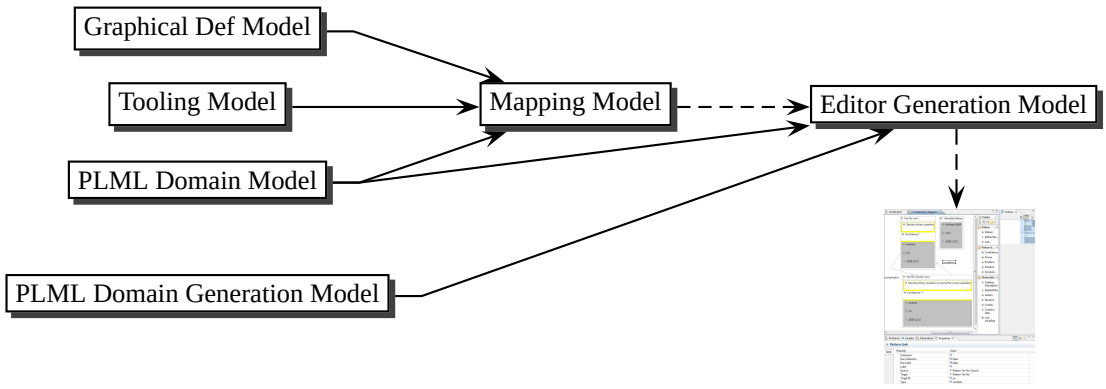

Abbildung 3.3: Zusammenhang der Modelle des GMF

und als Zweite die Model-To-Text Transformation welche den fertigen Editor als Eclipse-Plugin erstellt. Die erwähnten Wizards zur Generierung von `Graphical Definition`- und `Tooling`-Modell, welche ebenfalls als Model-To-Model Transformationen betrachtet werden können, sind in der Darstellung nicht enthalten.

Zuerst soll das *Graphical Definition*-Modell, im folgenden GMFGraph genannt, kurz erläutert werden. Dieses Teilmodell definiert Form und Gestaltung der später verwendeten grafischen Primitive. Im Metamodell von GMFGraph sind Basis-Primitive als Metaklassen vorgegeben, wie alle Metaklassen haben diese Attribute und Referenzen. Über die Referenz-Beziehungen können die Basis-Primitive kombiniert werden, und so komplexe Figuren beschrieben werden. Die so im GMFGraph-Modell festgelegten Kombinationen bilden später die Grundbausteine des generierten Editors. In der aktuelle Version von GMF sind die folgenden Basis-Primitive definiert: Label, Rechteck, abgerundetes Rechteck, Ellipse, sowie offene und geschlossene Polygonenzüge. Darüber hinaus können SVG-Grafiken [SVG] eingebettet werden, und sogar eigene extern implementierte Basis-Primitive referenziert werden. Oft werden allerdings nur die bereits eingebauten Basis-Primitive verwendet.

Die Basis-Primitive können nicht nur aneinandergereiht sondern auch über Containerbeziehungen verschachtelt werden. Die Darstellung einer UML-Klasse ließe sich etwa über ein Rechteck mit drei darin angeordneten Labels und zwei offenen Polygonenzügen für die Trennlinien realisieren. Für die Anordnung innerhalb eines Containers, in diesem Fall innerhalb des Rechteckes, werden Layoutmanager eingesetzt. Auch diese, genau wie deren mögliche Parameter, sind als Metaklassen im GMF definiert.

Soweit zur prinzipiellen Darstellung, im Detail ist das Erstellen des Graph-Modells ein langwieriger Prozess. Als besonders beschwerlich und zeitaufwändig erweist sich das Debugging und Tracing, da Veränderungen im Graph-Modell nur im generierten Editor geprüft werden können und dieser erst als Ergebnis von zwei nachgelagerten Transformationsschritten entsteht.

Im zweiten GMF-Basismodell, dem `Tooling`-Modell (GMFTool), werden die später möglichen interaktiven Bearbeitungsaktionen vorbereitet. Insbesondere sind dies das neu Anlegen von Objekten im Editor und das Herstellen von grafischen Verbindungen zwischen Editor-Objekten. Es sind prinzipiell beliebige weitere Aktionen denkbar, für die vorgenannten existiert jedoch bereits explizite Unterstützung durch GMF. Letztlich definiert das GMFTool-Modell im Wesentlichen die Menüstruktur und die Verteilung der Aktionen auf die verschiedenen Menüarten, wie Palette, Popup- oder Hauptmenü.

Das `Mapping`-Modell (GMFMap) fügt nun die Einzelteile zusammen. Unter anderem wird hier die Verbindung zwischen den einzelnen Metamodell-Elementen mit den in GMFGraph beschriebenen Figuren

und den Bearbeitungsaktionen hergestellt. Wie erwähnt verfolgt GMF die Graph-Metapher, dies zeigt sich im Mapping-Modell sehr deutlich. Hier entsteht aus der Zusammenführung von GMFGraph-, Tooling- und PLML-Domänenmodell zu den Diagramm-Knoten. Ein solcher Knoten führt eine Referenz auf die darstellende Figur und das darzustellende Metamodell-Element, er bildet den Controller-Teil des MVC-Pattern.

Darüberhinaus wird im GMFMap-Modell festgelegt, zwischen welchen Knoten überhaupt Verbindungen, also Kanten, zugelassen werden. Jede Kante muss detailliert, d.h. auf der Ebene der sie speichernden Metamodell-Attribute, definiert werden. Dasselbe gilt für Verschachtelungen bei Container-Figuren, auch für diese muss die sie speichernde Ecore-Containment-Beziehung festgelegt werden.

Eine verhältnismäßig gravierende Folge der Art und Weise wie diese Knotenmetapher in GMFMap umgesetzt ist, ist dass es nicht möglich ist beliebige Verschachtelungstiefen vorzusehen. Außerdem können nur konkrete Visualisierungsfiguren, verbunden mit Elementen konkreter Metamodell-Klassen, in Containern verschachtelt werden. Man kann zwar festlegen, dass ein Container A beliebige Unterelemente vom Typ B enthält. Es ist aber mit GMF nicht möglich in der gleichen Relation ein Unterelement des Typs C zu speichern, auch wenn C eine Unterklasse von B ist. Stattdessen muss zwangsweise eine explizite Container-Beziehung zwischen A und C spezifiziert werden.

Wahrscheinlich entstammt diese Einschränkung der Art und Weise, wie später das GEF[7] für den Editor benutzt wird. In jedem Fall macht es die Modellierung umständlich und läßt die damit erstellten Editoren gleichartig erscheinen.

Als Quellmodell für die Generierung des eigentlichen grafischen Editors dient das `Editor Generation`-Modell (GMFGen). Neben dem Metamodell der Zieldomäne wird vom GMFGen zusätzlich noch das *Generierung*smodell für das Metamodell der Zieldomäne benötigt. In diesem Generierungsmodell sind die Steuerungsparameter für die Java-Codegenerierung der Zieldomäne hinterlegt. Da der grafische Editor Instanzen des Zieldomänenmodells erstellen und manipulieren soll, wird dessen Modellimplementierung und De-/Serialisierungsmechanismus benötigt. Um deren korrekte Benutzung durch den Editor zu ermöglichen ist das Zieldomänen-Generierungsmodell nötig.

Im GMFGen lassen sich darüberhinaus noch künftige Klassennamen, diverse Labels und andere implementierungsnahe Eigenschaften festlegen. Der letzte Schritt im GMF-Prozess ist dann die Transformation des GMFGen-Modells in den Editor.

Mappings für den PLML-Editor

Bevor ein Editor spezifiziert werden kann, muss man sich natürlich Gedanken darüber machen, welche Teile der Sprache sinnvoll grafisch bearbeitet werden können. So auch für PLML. Offensichtlicher Kandidat für die Visualisierung sind die Patternbeziehungen. Wie in Tabelle 3.2 dargelegt, handelt es sich bei den meisten weiteren Sprachelemente um Texte, abgesehen von Abbildungen die ihrer Natur nach grafisch sind.

Im Ergebnis der Überlegungen sollte jedes Pattern als eigener Diagrammknoten und Patternbeziehungen als Kanten dargestellt werden. Um einen Knoten identifizieren zu können, schien es plausibel die Pattern-Id und den Namen im Knoten anzuzeigen. Beim Vertrauen (confidence) in den Patterncharakter eines Eintrags, ist ebenfalls eine hinreichend kleine Information um sie im Knoten unterzubringen. Für die textuelle Einträge wie etwa das, durch das Pattern gelöste, Problem oder die Patternlösung können auch im Knoten dargestellt werden. Da längere Texte den Umfang eines Knotens aufblähen, sollten diese jedoch in einklappbaren Teilcontainern untergebracht werden.

[7] Graphical Editing Framework

Damit ergeben sich für das GMFMap des PLML-gDSL[8] Editors folgende Zuordnungen (vgl. Abbildung 3.2 und Abbildung 3.1 für die Metaklassen). Die Typen Pattern, PatternDescription und Management werden durch grafische Container-Knoten im Editor dargestellt. Ein Pattern-Diagrammknoten bildet dabei jeweils den Wurzelcontainer einer einzelnen Patternbeschreibung, diese umfasst auch deren Stammdaten. PatternLinks werden als Verbindungen im Editor dargestellt und Labels an diesen Verbindungen stellen den Typ der Patternbeziehung dar. Für die textuellen Eigenschaften werden deren String-Attribute ebenfalls als editierbare Labels bereitgestellt und sind daher direkt im Editor bearbeitbar.

Zu den nicht im grafischen Teil des Editors bearbeitbaren Eigenschaften gehören die Boolean-Attribute der Confidence und der PatternLinkType. Das bedeutet nicht, dass diese Merkmale nicht änderbar sind. Denn, im generierten Standardeditor wird für jedes Modellelement auch noch ein sogenannter PropertySheet-Editor bereitgestellt. Dabei handelt es sich um eine tabellarische Schlüssel/Wert-Auflistung aller Attribute der Metaklasse. Nach Auswahl einer konkreten Instanz, im Diagramm-Editor, werde an dieser Stelle die Wertbelegungen der Attributwerte der aktuellen Instanz angezeigt. Die Instanzwerte können im PropertySheet-Editor nicht nur angezeigt, sondern auch bearbeitet werden. Das gilt selbstverständlich auch für die Eingangs erwähnten Merkmale.

Nach Festlegung aller Mappings für PLML generiert GMF einen grafischen PLML-Editor als Eclipse Plugin, eine Eindruck von diesem Werkzeug gibt Abbildung 3.4. In diesem speziellen Editor wurden auch die Stammdaten des Katalogs als eigener Diagramm-Knoten grafisch dargestellt. Der Katalog enthält zwei Patterns und es wurde eine bidirektionale Beziehung zwischen beiden definiert, es handelt sich um die Visualisierung einer Variante des Kataloges von Listing B.4. Zur Demonstration des PropertySheet-Editors, wurde in Abbildung 3.4 die Pattern-Beziehung „contains“ zwischen Yes-No-Cancel und Yes-No selektiert. Das Property-Sheet, also die Eigenschaftentabelle, im unteren rechten Drittel zeigt dementsprechend alle Eigenschaften dieser Relation; alle Attribute die im Metamodell für den Typ PatternLink existieren. Änderungen an diesen Eigenschaftswerten im PropertySheet haben direkte Auswirkung sowohl auf die Anzeige

[8] graphische DSL

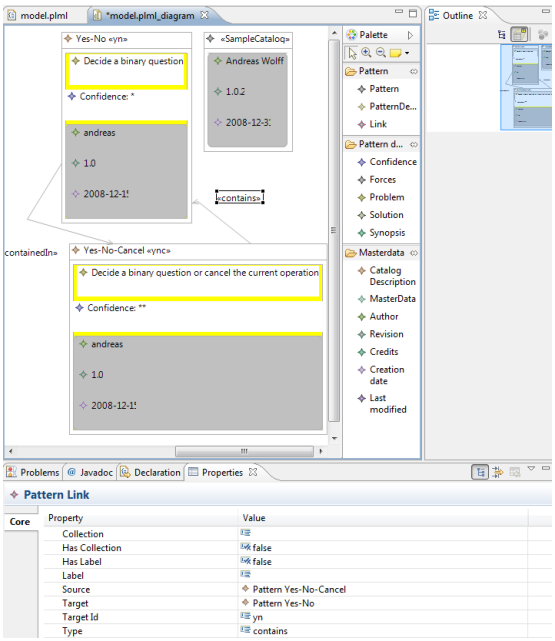

Abbildung 3.4: Screenshot eines exemplarischen graphischen PLML-Editors als Eclipse-Plugin

grafischen Editor als auch auf die Werte der Modellinstanz.

Der in Abbildung 3.4 präsentierte grafische Editor wurde ausgehend von einer früheren Version des Metamodells von PLML gebaut. In der Benutzung als Bearbeitungswerkzeug zeigte sich jedoch schnell, dass ein grafischer Editor nicht unbedingt die Optimallösung zur Pflege von PLML-Katalogen darstellt, das Projekt wurde demnach nicht weiter aktuell gehalten. Neben den Abweichungen die sich aus der Verwendung einer älteren PLML-Variante ergeben, erwiesen sich weitere Anpassungen am PLML-Metamodell als notwendig. Weitere Details zu diesem Thema finden sich in Unterabschnitt C.3.3

Sicherlich lassen sich mit dem GMF grafisch ansprechendere Diagramm-Editoren als jener aus Abbildung 3.4 bauen. Generell kann durch das Verwenden anderer Mappings eine nahezu beliebige Anzahl verschiedener grafischer Editoren für das PLML-Metamodell erzeugt werden. Limitierend in der Gestaltungsfreiheit dieser GMF-Editoren wirken aber immer: die bereitgestellten Grundfiguren, die Graph-Metapher und die Umsetzung des Containerkonzeptes.

Mögliche Änderungen am PLML-Editor müssen sich dabei nicht nur auf eine veränderte grafische Darstellung beschränken, theoretisch sind gänzlich andere Arten der Visualisierung möglich. Pattern-Beziehungen könnten beipielsweise durch grafische Container statt durch Verbindungspfeile dargestellt werden. Und sicherlich ist die Anzeige der Stammdaten auch nicht unbedingt ständig erwünscht.

Als vorläufiges Fazit der Untersuchung bleibt festzuhalten, dass ein grafischer Editor leider keine nennenswerte Verbesserung oder Vereinfachung bei der Pflege eines PLML-Kataloges bietet. Das Editieren von Texten in den Eingabefeldern des Diagramm-Editors erwies sich als umständlich, die Einbettung der Pattern-Illustrationen in die Knoten verbesserte nicht deren Erklärungspotential und generell litt die Übersichtlichkeit schon bei wenigen Pattern-Einträgen in einem Katalog.

Die Eignung speziell des PLML-Metamodell als generelles Metamodell für HCI-Patterns wurde bereits am Ende von Abschnitt 3.3 kritisch diskutiert. Daher könnten einige der Unzulänglichkeiten der automatisch erzeugten Editoren eventuell auch dem nicht-optimalen Metamodell zuzuschreiben sein. Jedoch war das PLML-Metamodell zu keiner Zeit der Grund dafür, dass eine bestimmte Visualisierung nicht durchgeführt werden konnte, im Gegensatz zu den restriktiven Vorgaben durch das GMF.

Aufgrund meiner Untersuchungen denke ich, dass sich wohl ansprechende grafische Darbietungsformen für einen Patternkatalog finden lassen. Bis auf weiteres überwiegen jedoch die Vorteile einer textuellen Katalogbearbeitung.

3.5 Pattern als Komponenten

3.5.1 Klassifikation der Komponentisierbarkeit von Patterns

Ein Kernaspekt des im Abschnitt 2.1 vorgestellten MDD-UI Prozessmodells ist es, in und zwischen allen Stufen Patterns zu integrieren. Dazu wird im Rahmen dieses Kapitels untersucht, welche HCI-Patterns sich für die Umsetzung in automatische oder halb-automatische Transformationen eignen können. Kann für ein Pattern ein solche Transfomation gefunden werden, dann ist es ein komponentisierbares Pattern. Die nötigen Entscheidungskriterien, eine Klassifikation sowie die Erkenntnisse zur möglichen Umsetzung eines Patterns als Komponente werden im folgenden vorgestellt.

Grundsätzlich sind mit Begriff HCI-Patterns Best-Practices aus allen denkbaren Bereichen der Mensch-Maschine-Kommunikation abgedeckt. Der Versuch allgemeingültige Kriterien für dieses weite Feld zu definieren scheint aussichtslos. Deshalb beschränke ich mich wiederum auf Patterns für WIMP-Oberflächen,

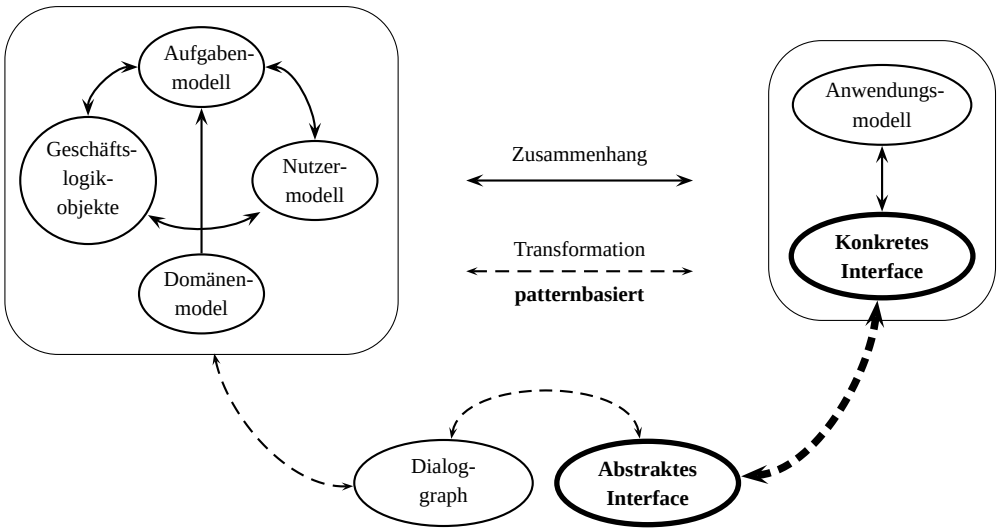

Abbildung 3.5: Verwendung von Pattern im MDD-UI Prozess

die dabei gewonnenen Erkenntnisse lassen sich jedoch in Grenzen generalisieren.

In Abbildung 3.5 ist der UI-relevante Ausschnitt des zugrundeliegenden MDD-UI-Prozess dargestellt. Die Modelle der abstakten und konkreten Oberfläche, sowie die Transformation dazwischen wurden besonders hervorgehoben, da es sich dabei um den Einsatzbereich der hier betrachteten Patterns handelt.

Komponentisierbarkeit, Algorithmenbegriff und Komponenten

Der Begriff *Komponente* hat eine sehr allgemeine Bedeutung. Dementsprechend kann auch *Komponentisierbarkeit* weit ausgelegt werden. In dieser Arbeit wird unter dem Konzept einer Patternkomponente die Idee verstanden, Patterns algorithmisch zu erfassen und umzusetzen. Komponentisierbarkeit meint in diesem Kontext die Möglichkeit der Erstellung einer Patternkomponente für ein gegebenes Pattern.

Zur Durchführung dieses Ansatzes ist zunächst der Begriff des Algorithmus zu klären. In [SW00] findet sich die folgende Definition:

> „Der Begriff Algorithmus (algorithm) bezeichnet eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist.“

Viele intuitive Begriffsklärungen folgen dieser Sichtweise. Eine formalere Herleitung, über Turing-Maschinen, verfolgt [Kir] und kommt zu dieser Definition:

> „Eine Berechnungsvorschrift zur Lösung eines Problems heißt Algorithmus genau dann, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt. “

Wichtiger als der konkrete Wortlaut dieser Definition sind die sich daraus ableitenden formalen Kriterien, diese sind nachstehend aufgelistet.

1. Operative Finitheit – endlich viele verschiedene Operatoren im Programm
2. Dynamische Finitheit – das Programm darf nur endlich viel Speicherplatz benötigen

3. Finitheit – der Programmtext muss endlich sein
4. Terminierung – der Programmablauf muss in endlicher Zeit terminieren
5. Vollständigkeit – zu jedem Zeitpunkt muss das Programm die nächste Operation eindeutig festlegen

Die zu identifizierenden Patternkomponenten für den modellgetriebenen Ansatz werden letztlich mit den in Unterabschnitt 2.2.2 beschriebenen Model-To-Model Transformationstechniken, unter Benutzung der Oberflächenmetamodelle, deklariert. Diese Festlegung limitiert die einsetzbaren Operatoren, somit erfüllen alle erstellbaren Patternkomponenten das Kriterium der operativen Finitheit (1). Das Kondensat der Kriterien (2) bis (5) ist die Forderung nach einem endlichen terminierenden deterministischen Programm, beziehungsweise in diesem Fall: Transformationsbeschreibung. Transformationsdeklarationen, als interpretierte Spezifikationen, sind immer endlich (3) und, zumindest gemäß der jeweiligen Sprachdefinitionen, deterministisch (5). Terminierung (4) und Speicherplatzbedarf (2) lassen sich nicht im Vorhinein entscheiden, sondern bedürfen einer Untersuchung der jeweiligen Transformation.

Komponentisierbarkeit der GoF-Patterns

In einer ähnlichen Untersuchung kategorisierte Arnout [Arn04] die bekannten objektorientierten Patterns von Gamma [GHJV02] und der GoF. In ihrer Dissertation ermittelte sie, welche der GoF-Patterns sich in der Programmiersprache Eiffel als vordefinierte Komponenten umsetzen lassen. Allerdings ist ihr Ansatz zur Komponentisierbarkeit ein anderer. Angepasst an die Einsatzdomäne der GoF-Patterns, handelt es sich bei Arnouts Komponenten um Klassen, vordefiniert und zur Verfügung gestellt in einer Bibliothek. Zur Nutzung dieser Patternkomponenten erstellt man eigene Klassen die von den bereitgestellten Komponentenklassen erben. Dabei sind zumeist abstrakte Methoden zu implementieren oder die Typen der generischen Klassen festzulegen.

Zu den Erkenntnissen Arnouts zählt, dass nicht für alle GoF-Patterns sinnvolle Eiffel-Komponenten angegeben werden können. Zur genaueren Klassifikation entwickelte sie in [Arn04] eine Hierarchie mit 14 Kategorien. Von den ursprünglichen 23 Patterns der GoF [GHJV02] lassen sich 15 in Komponenten darstellen. Von diesen 15 Patterns sind wiederum nur 9 Komponenten als nützliche Patternkomponenten klassifiziert. Die sechs weiteren Patterns sind entweder bereits als Basisfeature von Eiffel vorhanden (Prototype), nicht vollständig umsetzbar (Builder, Proxy, State), nur abweichend von der Patternidee komponentisierbar (Strategy) oder aber eine Komponente ist zwar möglich, aber letztlich nutzlos (Memento). Die quantitativen Ergebnisse von Arnouts GoF-Pattern Klassifikation sind in Tabelle 3.3 zusammengefasst.

Klassifikationskriterien

Zur Klassifikation der einzelnen Patternkomponenten stellt Arnout in [Arn04] die nachfolgenden Kriterien zur Bewertung von Patternkomponenten auf:

Vollständigkeit – Sind alle in der Patternbeschreibung enthaltenen Fälle enthalten?

Nützlichkeit – Ist die Verwendung der Patternkomponente sinnvoll im Vergleich zu einer jeweils eigenen Implementierung der Patternidee?

Performanz – Ist die Verwendung der Patternkomponente genauso effizient wie eine separate Implementierung der Patternlösung?

Ideentreuheit – Entspricht die Umsetzung des Patterns der vorgegebenen Patternlösung?

Referenz	Kategorieinhalt	# zugeordnete GoF-Pattern
1.	Komponentisierbare Patterns	15, davon
1.1	Sprachinhärent	1
1.2	Unterstützt durch Sprachbibliothek	-
1.3.	Entwickelte Komponenten	14, davon
1.3.1	vollständig Komponentisierbar	9
1.3.2	unvollständig Komponentisierbar	3
1.3.3	Komponentisierbar, abweichend von Patternidee	1
1.3.4	Komponentisierung möglich, Komponente jedoch nutzlos	1
1.4	Kandidat für Komponentisierung	-
2.	Nicht komponentisierbare Patterns	8, davon
2.1	Komponentengerüst erstellbar	4, davon
2.1.1	Komponentengerüst mit Methoden	2
2.1.2	Komponentengerüst nur ohne Methoden	2
2.2.	Komponentengerüst grundsätzlich denkbar	1
2.3.	gewisse Unterstützung durch Sprachbibliothek	1
2.4.	keine Unterstützung möglich	2

Tabelle 3.3: Klassifikationskategorien zur Komponentisierbarkeit von Pattern, nach [Arn04]

Typsicherheit – Ist die Komponente typsicher?

Erweiterte Anwendbarkeit – Deckt die Patternkomponente womöglich mehr Fälle als die ursprüngliche Pattern-Idee ab?

Diese Punkte sind ein geeigneter Startpunkt zur Identifikation eigener Kriterien. Eine $1:1$-Umsetzung und Anwendung für WIMP-Patterns ist aus mehreren Gründen kaum sinnvoll.

Wesentlich ist dies in der unterschiedlichen Einsatzdomäne begründet, außerdem verhindert der hohe Abstraktionsgrad, bei gleichzeitig mangelnder Formalisierung der WIMP-Patterns, eine direkte Kriterienübernahme: Die GoF-Patterns wurden in den Programmstrukturen objektorientierter Software identifiziert und sind als Klassendiagramme, in einer prä-UML Notation, formal beschrieben. Wie in der Einleitung von Kapitel 3 dargelegt, trifft dies auf die WIMP-Patterns so nicht zu. Deren Komponentisierung baut zwar auf Klassenstrukturen auf, da die Implementierung auf Ecore-Metamodellen aufsetzt, letztlich handelt bei den Komponenten aber um textuell deklarierte Transformationen.

In der Konsequenz bedeutet dies zunächst, dass das Kriterium der „Typsicherheit“ für WIMP-Patterns nicht zielführend sein kann. Ebenfalls kritisch zu hinterfragen ist die „Performanz“. Im Kontext einer Eiffel-Implementierung ist diese Frage sehr angebracht, unter modellgetriebenen Gesichtspunkten jedoch nicht wirklich von Bedeutung. In der MD-UID ist es das Ziel überhaupt geeignete Transformationen zu finden, die Performanz hat nur für zur Laufzeit interpretierte Modelle eine herausgehobene Bedeutung. Dies trifft für die Anwendung der hier für die Komponentisierbarkeit betrachteten Metamodelle nicht zu.

Von genereller Bedeutung ist hier die ähnlich gelagerte Frage, inwiefern sich Patternkomponenten in den üblichen Transformationstechniken eingebetten lassen. Dies ist jedoch kein Problem einzelner Patterns sondern eines des gesamten Ansatzes. Dazu ist festzustellen, dass die eingesetzten Techniken jeweils das Aufrufen externer Transformationen, quasi API-/Bibliotheksaufrufe, jeweils als Sprachbestandteil zulassen. Mit anderen Worten, QVTo-Transformationen können andernorts deklarierte QVTo-Transformationen benutzen. Der Ansatz an sich ist daher plausibel und durchführbar.

Eine Erweiterung gegenüber Arnouts Kriterien ergibt sich aus der Tatsache, dass ich die Komponentisierbarkeitsuntersuchungen für zwei CUI Metamodelle, Swing und XUL, vergleichend durchgeführt habe. Dabei stellte sich heraus, dass die Komponentisierbarkeit eines Patterns auch vom zugrundeliegenden Metamodell der konkreten Oberfläche abhängig ist.

Nach diesen einführenden Betrachtungen zum Gegenstand und Aspekten der Komponentisierung von Patterns geht es im Folgenden darum, eine einheitliche Basis von Patternbeschreibungen zu finden. Die Patterns in dem dabei vorgestellten Patternkatalog werden anschließend in Unterabschnitt 3.5.4 auf ihre Eignung als Komponenten untersucht und schließlich die Ergebnisse summarisch präsentiert.

3.5.2 Aufbau eines Katalogs von Patternkomponenten

Das Ziel dieser Arbeiten ist die Erstellung von direkt nutzbaren Patternkomponenten. Diese sollen möglichst in einem Katalog unter Verwendung der in Abschnitt 3.3 entwickelten textuellen domänenspezifische Sprache gesammelt werden.

Der zu erstellende Patternkatalog soll keine vollständig neue Eigenentwicklung werden, sondern das in den vorhandenen Patternlanguages gesammelte Wissen aufnehmen. Das erste Ziel ist daher einen vereinheitlichten Pattern-Katalog aus den vorliegenden Patternlanguages zusammenzustellen. Das ist mit PLML möglich, auch wenn zur Umsetzung der Idee mehrere Probleme zu lösen waren.

Im Grunde ist ein vereinheitlichter monolithischer Katalog nicht unbedingt notwendig, stellt PLML doch Sprachfeatures bereit, die den Zusammenhang zwischen diversen Katalogen abbilden können. Bei der praktischen Anwendung erweist es sich jedoch schnell als unvorteilhaft gleichartige Patterns lokal in verschiedene PLML-Kataloge aufzuteilen. Insbesondere geraten die Transformationen und Abfragen komplexer, und damit auch fehleranfälliger, ohne das dieser Nachteil durch entscheidende Vorteile eines verteilten Kataloges aufgewogen werden kann.

Die prinzipielle Vorgehensweise zur Erstellung eines vereinheitlichten Kataloges ist wie folgt:

1. Übernahme existierender Patternlanguages in den eigenen Katalog, dabei:
   (a) Formatumwandlung Input→PLML
   (b) Verschmelzung, inklusive der Entfernung von Doppeleinträgen
   (c) Identifikation von Querbeziehungen zwischen Patterns
2. Komponentenerstellung:
   (a) Bewertung der Komponentisierbarkeit jedes Patterns
   (b) i. ggf. Entwicklung eines Algorithmus für die Patternlösung
   ii. ggf. Entwicklung eines Templates für die Verwendung der Patternlösung in der Transformationsengine des XUL-Editors
3. Bereitstellung des Kataloges zur Nutzung in einem MD-UID Prozess

Ausgangspunkt der Arbeiten waren in unserem Fall die Web-Inkarnationen des Welie und Tidwell-Katalogs. Es wurde entschieden, Tidwells Webkatalog zu benutzen, anstelle der Buchfassung, weil für den Web-Katalog die Möglichkeit einer automatisierten Datenübernahme bestand; die Benutzung der Buchfassung hätte einen erheblichen manuellen Aufwand erfordert. Ein Mehraufwand der für eine Proof-Of-Concept Implementierung unangemessen erschien.

In den Erörterungen zu PLML wurde mehrfach auf die Beibehaltung einer Abwärtskompatibilität hingewiesen. Dies geschah nicht grundlos, denn der Webkatalog von Welie stellt seine Patternbeschreibungen als PLML zur Verfügung. Welies PLML-Patternbeschreibungen in die eigene PLML-Variante zu konvertieren wäre demnach verhältnismäßig einfach. Allerdings ist die durch Welie verwendete Fassung von PLML ebenfalls nicht mehr identisch zu dem in [Fin03] definierten. Leider ist das XML-Schema seiner Version nicht mehr verfügbar, so dass ein Vergleich der Sprachversionen manuell erfolgen musste.

Im Wesentlichen hat Welie ein Element hinzugefügt, zwei umbenannt und setzt für die eigentliche Patternbeschreibung auf die Sprache XHTML. Welie verwendet zur Deklaration von Aliasnamen das Element „aka“ statt „alias“ und hinterlegt die Zusammenfassung der Patternidee in einem Element „pattlet“ anstelle des PLML-Sprachelementes „synopsis“. Der Begriff Pattlet scheint im Umfeld von Microsoft geprägt zu sein und steht, z.B. nach [Mic], für die Kurzbeschreibung einer Musterlösung zu einem wiederkehrenden Problem - meint somit ebenfalls eine Patternzusammenfassung. Das neu hinzugekommene Element „code-link“ dient zur Verlinkung von Beispielquellcode über HTML-Mechanismen.

Importframework für externe Patternlanguages

Die Schrittfolge zur Übernahme aus anderen Katalogen ist jeweils sehr ähnlich, die folgenden Punkte sind abzuarbeiten:

1. Beschaffung einer XML-Serialisierung
2. Instanziieren eines semantisch gleichwertigen PLML-Ecoremodells
3. Generieren der tDSL-Beschreibung aus dem PLML-Ecoremodell

Es wurde ein Framework implementiert, das diese Aufgaben weitestgehend übernimmt. Die Anpassung an die variable Struktur der Eingaben, also der wechselnden Katalog-XML-Serialisierungen, folgt der Lösung des GoF-Patterns „Erbauer“. Der Implementierungsaufwand zum Parsen eines weiteren Webkatalog besteht daher in der Implementierung weniger Interfaces.

Das Framework benutzt für die Erstellung des korrespondieren PLML-Ecoremodells die durch EMF bereitgestellte Technik der dynamischen Modellinstanziierung. Nach abgeschlossenem Importvorgang wird dieses Modell schließlich mit den EMF-Mechanismen serialisiert. Diese Serialisierung, ein XML-Dokument, wird abschließend über eine Model-To-Text Transformation mit Acceleo in die tDSL-Serialisierung von PLML überführt.

Die Abbildung 3.6 illustriert wie dieser Prozess für den Fall der Tidwell-Patterns durchgeführt wurde. Das Problem mit diesem Webkatalog liegt darin, dass die Patterns in HTML-Form präsentiert werden. HTML als Seitenauszeichnungssprache ist zwar zur Präsentation von Daten geeignet, jedoch nicht unbedingt zu deren Weiterverarbeitung prädestiniert. Die unzähligen üblichen Abweichungen vom Sprachstandard erschweren das automatisierte Parsen von HTML. Vor einer Weiterverarbeitung bietet sich daher eine Transformation der HTML-Dateien in XML-Dokumente an.

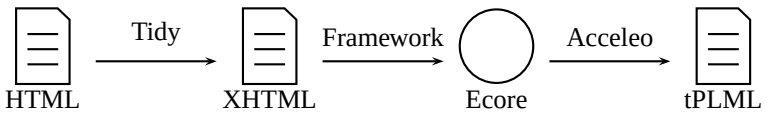

Abbildung 3.6: Verarbeitungskette zur Erstellung des PLML-Kataloges aus dem Tidwell-Webkatalog

Natürlich ist es kaum sinnvoll, einen eigenen Format-Umwandler zu entwickeln, daher wurde ein bekanntes Tool für diesen Zweck eingesetzt, das vom W3C initiierte Programm „HTML Tidy“ [Pro]. HTML Tidy behebt nicht nur einfache Syntaxfehler im HTML-Code sondern verfügt auch über Heuristiken um fehlerhafte Elementverschachtelungen und fehlende Element-Tags zu ermitteln. Das Ergebnis der Anwendung von HTML Tidy auf die HTML-Seiten der Patternbeschreibungen sind valide XHTML- und damit XML-Dokumente. Diese Dokumente werden vom Framework interpretiert und dabei die Nutzinformationen in einem Laufzeit-PLML-Ecoremodell aggregiert. Für die Einträge des Welie-Kataloges ist natürlich die Normalisierung mit HTML-Tidy entbehrlich.

Zu den dargestellten Erweiterungen, siehe Unterabschnitt 3.2.2, der PLML gehört das Kategorienkonzept. Beide hier untersuchte Kataloge machen Gebrauch davon. Welie nutzt sogar Unterkategorien, Tidwell beschränkt sich auf eine Kategorisierungsebene. Im Rahmen des Katalogimports werden diese Katalogstrukturen nachgebildet und die dementsprechenden Modellzuordnungen vorgenommen.

Patternbeziehungen innerhalb der Kataloge

Nach Durchlaufen des Importframeworks liegen die Patternkataloge als Textdateien mit der Syntax der PLML-tDSL vor. Die in den Webkatalogen enthaltenen grafischen Abbildungen wurden dabei als externe Links mit ihrer URL übernommen.

Patterns stehen nicht als isolierte Aufzählungen in den Katalogen. Oftmals kann für die Umsetzung eines Patterns die Idee anderer Patterns benutzt werden. Die typisierten kataloginternen Referenzen von PLML dienen dazu derartige Beziehungen auszudrücken.

Die Kataloge von Tidwell und Welie unterscheiden sich erheblich in Bezug auf die angegebenen Beziehungen. Die beiden Darstellungen von Abbildung 3.7 und Abbildung 3.8 sind ein Versuch diesen Sachverhalt zu visualisieren.

Beide Abbildungen wurden durch eine Acceleo-Transformation der jeweiligen PLML-tDSL Kataloge in LaTeX-PSTricks Quellcode erstellt. Alle Patterns eines Kataloges sind als einzelner Kreisknoten abgebildet. Über die Färbung des Kreises wird der Vernetzungsgrad des Patterns nochmals hervorgehoben. Weiß ausgefüllte Kreise stehen für Patterns die mindestens ein weiteres Pattern benutzen. Graue Füllungen weisen Patterns aus welche von mindestens einem anderen Pattern referenziert werden. Rot sind schließlich diejenigen Patterns hervorgehoben die andere Patterns weder durch andere Patterns referenziert werden noch ein anderes Pattern verwenden.

Im Tidwell-Katalog befinden sich demnach sechzehn Patterns die überhaupt keine Verbindung zu anderen Patterns haben. Zehn weitere Patterns werden zumindestens referenziert. Somit bleiben insgesamt nur achtzehn Patterns (40,9 Prozent) die zur Erklärung oder Lösung auf andere Patterns zurückgreifen. Die höchstintegrierten Patterns dieses Kataloges sind, mit jeweils fünf Referenzierungen, „Wizard“, „Two-Panel Selector“und „Titled Sections“.

Anders stellt sich das für den Welie-Katalog dar. Nicht nur durch die dichtere Positionierung aufgrund der höheren Anzahl von Patterns finden sich in Abbildung 3.8 optisch mehr Verbindungen. Nur neun der 131 Patterns der Welie-Sammlung sind mit keinem andern Pattern-Eintrag verbunden. Zwanzig weitere Einträge treten als Referenzziele in Erscheinung, referenzieren selbst aber keine anderen Patterns. Somit werden in immerhin 102 (77,9 Prozent) Patternbeschreibungen andere Patterns mitbenutzt. In diesem Katalog benutzt das Pattern „Museum“ 15 andere Patterns, „Search Results“ verwendet 12, an dritter Stelle folgen „Application “ und „Information-Experience“ mit jeweils 11 Referenzierungen. Vergleichend fällt beispielsweise auf, dass das Pattern „Wizard“ bei Welie im Gegensatz zu Tidwell nur ein einziges anderes Pattern referenziert.

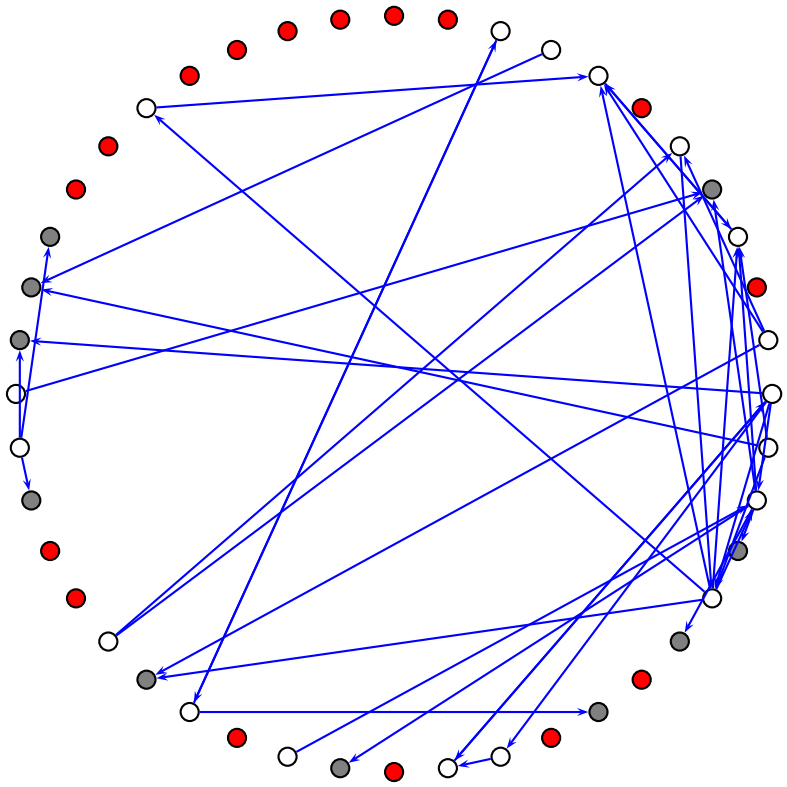

Abbildung 3.7: Intrakatalogreferenzen beim Tidwell-Katalog [Tid]

Einheitlicher Patternkatalog

Nachdem die Kataloge als PLML-Modellinstanzen bereitstehen, ist gemäß der in 3.5.2 vorgestellten Vorgehensweise, der nächste Schritt zur Erstellung eines vereinheitlichten Kataloges die Verschmelzung (1b). Dieser Prozessabschnitt erfordert eine Reihe menschlicher Entscheidungen und wird daher eine manuelle Tatägkeit bleiben.

Im Einzelnen ist für jedes Pattern zu entscheiden, ob es sich um einen Doppeleintrag oder Aliasnamen handelt, die referenzierten Patterns anzupassen und über die Angemessenheit des `Confidence`-Wert zu befinden. Bei der Übertragung externer Patternkataloge wird zunächst davon ausgegangen, dass jeder Eintrag ein nachgewiesenes Pattern darstellt. Dies wird, wie in Unterabschnitt 3.2.2 erläutert, durch den `Confidence`-Wert: ∗∗ ausgedrückt.

Nötig kann eine solche `Confidence`-Abwertung z.B. bei unvollständigen Patterndeklarationen sein. Offensichtlich könnten derlei Patterns auch sofort aus dem Katalog entfernt werden, allerdings könnten dabei Konsistenzprobleme auftreten, falls diese Einträge anderweitig referenziert sind, außerdem nimmt man sich die Möglichkeit den Patterneintrag später, eventuell aus anderen Quellen, zu vervollständigen. Die Abwertung der `Confidence` hat keine deratigen Konsequenzen und ist als Problemmarkierung ausreichend.

In Tabelle 3.4 sind die unterspezifizierten Patterns der beiden Webkataloge aufgeführt. Es werden die Patterns aufgeführt für die, nach der Datenübernahme, keine Angaben zur Problembeschreibung (`Problem`), dem Kontext (`Context`), der Patternlösung `Solution` oder zum Nachweis des Patterncharakters (`Evidence`) vorliegen. Bei einer Ausnahme: die bei Welie angegebene Patternlösung für „Tutorial“ besteht aus dem Verweis auf einen Buchtitel, nicht aus einer qualifizierten Beschreibung. Der Eintrag wurde daher

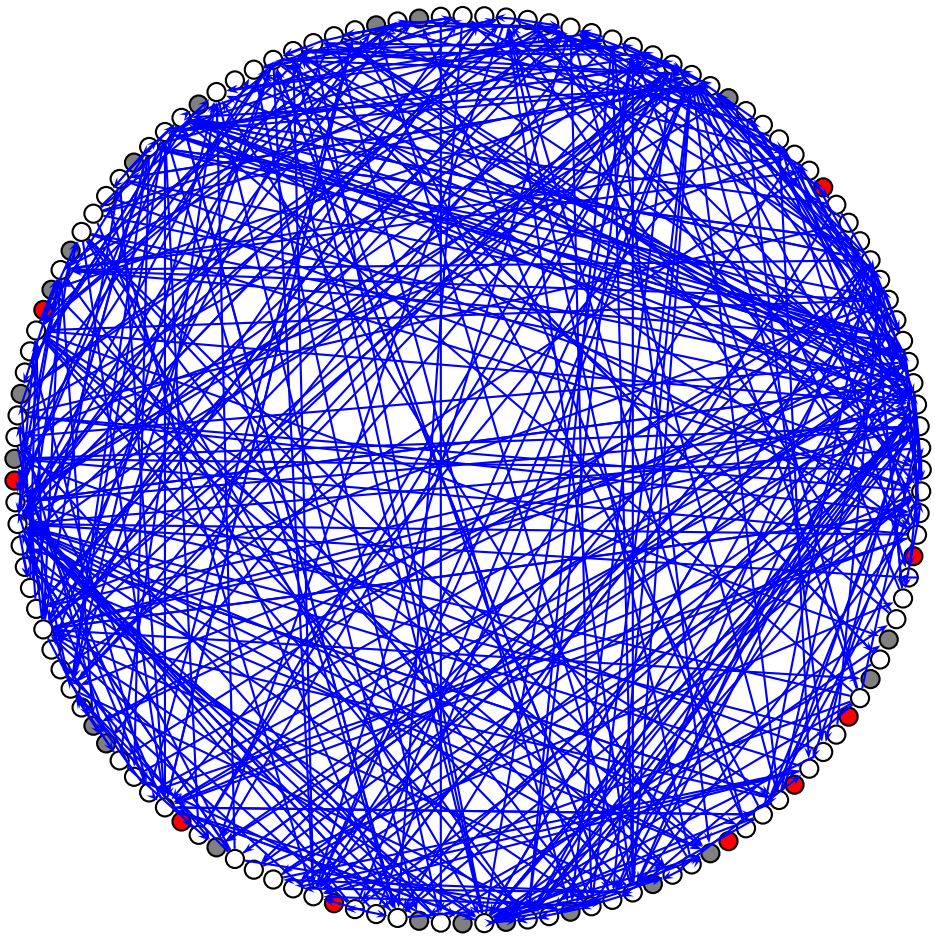

Abbildung 3.8: Intrakatalogreferenzen beim Welie-Katalog [vW]

in die Problem-Auflistung übernommen.

Nicht alle in Tabelle 3.4 aufgelisteten Patterns müssen im `Confidence`-Wert herabgestuft werden. So lassen sich ohne Schwierigkeiten Beispiele für das „One-Off Mode“[9]-Pattern finden und in den Katalog einpflegen. Die fehlende Lösung des „Send-To-Friend“-Patterns erklärt sich durch Diskrepanzen zwischen HTML-Version der Welie-Webseite und dem ausgelieferten PLML. Insgesamt betrachtet sind sechs bzw. vier problematische Patterns von 175 Datensätzen ein handhabbares Problem, das spricht für die Qualität der beiden Kataloge.

[9] Modusumschaltung nur für eine Aktion, danach Rückkehr in vorherigen Modus

fehlender Aspekt	Katalog Welie	Katalog Tidwell
Problembeschreibung	Details-on-Demand, Storytelling, Tutorial	—
Kontext	Details-on-Demand, Storytelling, Travel Site, Tutorial	—
Patternlösung	Send-To-Friend, (Tutorial)	—
Existenznachweis	—	One-Off Mode

Tabelle 3.4: Problematische Patterneinträge in den Webkatalogen

Durchführung der Verschmelzung

Nach Revision der Problempatterns erfolgt die tatsächliche Verschmelzung, d.h. die Punkte 1b und 1c in der Prozessbeschreibung nach 3.5.2. Beides läßt sich semi-automatisch durchführen; zum Einsatz gelangt wiederum selbstentwickelte Java-Software und QVT, welche auf Ecore und OCL aufsetzen.

Prinzipiell ist die Katalogverschmelzung eine Model-To-Model Transformation und hätte mit den hierfür vorgesehenen Transformationsmechanismen, siehe Unterabschnitt 2.2.2, implementiert werden können. Da jedoch das Import-Framework eine Java-Implementierung und die Katalogverschmelzung ein dem nahestehender Verarbeitungsprozess ist, wurde an dieser Stelle noch kein Technologiewechsel vorgenommen.

Vor der Darlegung des Verschmelzungsergebnis sollen noch einige kurze Erläuterungen zu den spezifischen Problemen der Vermischung erfolgen. Die Implementierung ist darauf ausgerichtet, immer genau zwei Kataloge (K_1,K_2) zusammenzuführen, dies hat keine Einschränkung der allgemeinen Anwendbarkeit zur Folge da mehrere Kataloge sequentiell paarweise vermischt werden können.

Für die Transformation gilt dann, dass alle Patterns P_i mit $P_i \in K_m \wedge P_i \notin K_n$ wobei $m, n \in \{1, 2\} \wedge m \neq n$, also alle nur in einem Katalog enthaltenen Patterns, direkt in den neuen Ergebniskatalog K_T übertragen werden. Die Entscheidung auf Enthaltensein basiert ausschließlich auf dem Vergleich der Pattern-ID, erfordert daher zwangsweise eine manuelle Nachbearbeitung. Da die Vergabe, Zusammensetzung oder Syntax einer Pattern-ID nicht festgelegt sind und daher von den Autoren der Patternkataloge frei gewählt werden können. Im Rahmen der Arbeiten zur Katalogerstellung wurde dennoch eine Normalisierung der IDs vorgenommen: alle Zeichen die nicht im lateinischen Alphabet enthalten sind werden durch den Unterstrich „_“ und alle Großbuchstaben durch ihren jeweiligen Kleinbuchstaben ersetzt.

Der Ergebniskatalog K_T wird neu angelegt. Bei der Übertragung der Patterns müssen daher alle Modell-Verlinkungen, dass sind neben den Patternbeziehungen auch die jeweilige Kategoriezuordnung, neu aufgebaut werden. Der Kategoriebaum von K_T ergibt sich aus der Vereinigung der Kategoriebäume von K_1 und K_2. Weil für Kategorien keine weiteren Abhängigkeiten bestehen, ist dessen Erstellung der erste Schritt bei der Katalog-Verschmelzung.

Das Übertragen und Auflösen von Intrakatalog-Patternreferenzen kann nicht in einem Durchlauf erfolgen, da oft der Fall auftreten wird, dass ein neuer Eintrag für K_T einen Patterneintrag aus seinem Quellkatalog referenziert, welcher jedoch noch nicht in K_T angelegt ist. Zur Überwindung dieses Hindernisses wird während der Übernahme-Phase der alternative Referenzierungsmechanismus über die Pattern-IDs genutzt. Erst nach Eintragung aller Patterns in K_T werden diese Pattern-IDs, mittels OCL-Queries, in die korrekten Objektassoziationen umgerechnet. Außerdem erfolgt noch eine aufsteigende Sortierung der Katalogeinträge nach der Pattern-ID, dies erwies sich als praktisch für die nachgelagerte manuelle Duplikatsuche. Natürlich wird im Rahmen der Referenzierungskorrektur ebenfalls der Fall gehandhabt, dass eine *inter*katalog Patternreferenz auf einen Eintrag im Verschmelzungspartnerkatalog zeigt; die Konsequenz dieses Falles ist schlicht, dass aus der *inter*katalog eine lokale *intra*katalog Referenz wird.

Etwas aufwändiger gestaltet sich die Vermischung falls ein Pattern, beziehungsweise dessen ID, in beiden Katalogen auftritt. Die Grundannahme für die automatische Transformation ist, dass ID-Äquivalenz auch semantische Äquivalenz beinhaltet. Selbstverständlich erfordert jeder einzelne Fall eine gründliche Überprüfung dieser Annahme.

Für die Vermischung der Welie und Tidwell-Kataloge trifft die Annahme zu. Insgesamt existieren lediglich drei gleichbenannte Patterns, im Einzelnen: „Wizard“, „Liquid Layout“und „Center Stage“. Wie bereits dargelegt, unterscheidet sich die Qualität der Beschreibungen in den Katalogen. Das Mischen zweier Einträge verfolgt daher das Konzept, einen Eintrag als Primärquelle auszuwählen und diesen durch die Informationen aus der Sekundärquelle aufzufüllen oder zu ergänzen. Für die Welie-Tidwell-Transformation

habe ich mich für die Patterneinträge von Tidwell als Primärquelle entschieden, weil diese subjektiv besser ausgearbeitet und durchdacht erscheinen.

Einen wirklichen Unterschied ergibt sich durch diese priorisierte Quellenwahl nur für die URL-Einträge zu Context, Diagram und Illustration. Ist in der Primärquelle bereits ein Wert für diese Eigenschaft gesetzt, wird derjenige der Sekundärquelle verworfen und nicht in K_T übernommen.

Viele Aspekte eines Patterns werden auch in PLML durch reinen Text beschrieben. Beim Zusammenführen dieser textuellen Informationen werden die Ausgangstexte jeweils in eigene HTML- Blöcke verpackt. Dies ist einerseits ein deutlicher Indikator dafür, dass an dieser Stelle menschliche Nachbearbeitung notwendig ist, ermöglicht aber gleichzeitig einem PLML-Renderer die beiden Textblöcke geeignet zu trennen oder auch nur einen der Texte darzustellen.

Bei Eigenschaftswerten mit der Multiplizität $1 : *$ werden die Einträge aus der Sekundärquelle, nach bestehen eines einfachen Duplikat-Checks, den Einträgen der Primärquelle hinzuaddiert. Das betrifft zum Beispiel Literature, Evidence und RelatedPatterns.

Ich habe mich entschieden bei dem Wert für Confidence von dem Primär-/Sekundärschema abzuweichen. Aus meiner Sicht ist die Abqualifizierung eines Patterneintrags zu einem Patternkandidat oder der Entscheidung auf „Nicht-Pattern“ ein gravierender Schritt der auf sorgfältigen Erörterungen beruht. Dem angemessen wird immer der minimale Confidence-Wert der beiden Patterneinträge als Confidence-Wert des K_T-Eintrages eingesetzt. Alle Patterns sowohl des Tidwell- als auch des Welie-Katalog sind mit dem Confidence-Wert „**“ ausgezeichnet, die vorstehenden Betrachtungen sind für deren Mischung, mit Ausnahme der in Tabelle 3.4 erwähnten Abstufungen, nicht relevant.

Keine Änderungen im Vermischungsfall ergibt sich für die Vorgehensweise der Auflösung der Patternverlinkungen, diese werden genauso im Post-Verschmelzungdurchlauf in Objektreferenzen umgesetzt. Die Kategoriezuordnung des verschmolzenen Patterneintrags erfolgt wiederum entsprechend der Kategorie der Primärquelle.

Im Ergebnis der Welie-Tidwell-Katalogverschmelzung entsteht ein Katalog mit 26 Kategorien und 172 Pattern-Einträgen. Die Analyse des Transformationsergebnisses ergibt, dass das Duplikat des Patterns „Color Coded Sections“ nicht erkannt wurde und somit zweifach im Ergebniskatalog vorliegt. Grund war, dass Tidwell das Pattern „. . . Sections“ nennt, während Welie auf das Plural-S verzichtet. Diese Patternverschmelzung war also noch manuell anzustoßen, wodurch sich letztlich der vereinheitlichte Katalog auf 171 Patterneinträge reduziert.

3.5.3 Deklarationsmodell für Patternkomponenten

Für die Beziehungen zwischen den Pattern im verschmolzenen Katalog sind selbstverständlich mehr Aspekte als die reine Namensgleichheit zu betrachten. Dazu zählt insbesondere die genauere Spezifikation der Pattern-Querbeziehungen. Andere Angaben die, im Rahmen der menschlichen Nachbearbeitung, nachträglich korrigiert oder hinzugefügt werden sind die Confidence-Klassifizierung, die Kategorisierung des Patterns aber auch so grundlegende Informationen wie der offizielle bzw. vereinheitlichte Name des Patterns.

Grundsätzlich könnten diese Informationen direkt durch manuelle Anpassung der PLML-Beschreibung erfolgen. Bei der Betrachtung des Gesamt-Workflows wäre eine solche Vorgehensweise jedoch mindestens als hinderlich zu bezeichnen. Denn, ausgehend von den Web-Inkarnationen der Kataloge konnte bis hierhin der verschmolzene Katalog ausschließlich über automatisierte Transformationen generiert werden. Im Falle von Aktualisierungen der Webkataloge lassen sich diese Informationen somit sehr schnell neuer-

lich auslesen. Editierte man nun das PLML-Transformationsergebnis manuell, dann stellte sich, bei einer anschließenden Neugenerierung ausgehend von den Webkatalogen, das Problem, wie diese Nachbearbeitungsvorgänge geeignet in den neugenerierten Katalog übertragen werden könnten.

Eine verwendbare Lösung dieses Problem bietet PLMLComponent, das PLML Komponentenmodell. Modellinstanzen dieses Metamodells dienen dazu Komponenten zu beschreiben welche Patterns umsetzen. Der Aufbau dieses Hilfs- oder Ergänzungsmodells ermöglicht es, PLMLComponent gleichzeitig als Reparaturmodell für PLML-Kataloge einzusetzen.

PLML bietet, abgesehen von den Implementation-Objekten, keine Möglichkeit der Definition von Patternkomponenten. Anpassungen von PLML in dieser Richtung sind natürlich denkbar. Dabei ist jedoch zu beachten, dass PLML auf Grund seiner Sprachauslegung nicht auf eine bestimmte Art von Patterns beschränkt ist. Daraus läßt sich die Forderung ableiten, die Patternkomponenten ebenfalls domänenunabhängig zu definieren. Dies hätte wiederum ein sehr allgemeines, schwach typisiertes und zur automatischen Weiterverarbeitung der Komponentendeklarationen wenig geeignetes Modell zur Folge. Für die Deklaration eines separaten, und PLML nur ergänzenden, Komponentendeklarationsmodells mussten diese PLML-Beschränkungen nicht beachtet werden.

Das Ergebnis der Überlegungen zu PLMLComponent ist in Abbildung 3.9 als Klassendiagramm angegeben. Um die Modellinstanzen von PLML und PLMLComponent möglichst unabhängig voneinander definieren zu können, habe ich darauf verzichtet aus dem PLMLComponent-Modell heraus direkt Pattern -Instanzen in PLML-Katalogen zu referenzieren, stattdessen wird der Fallback-Mechanismus, über den

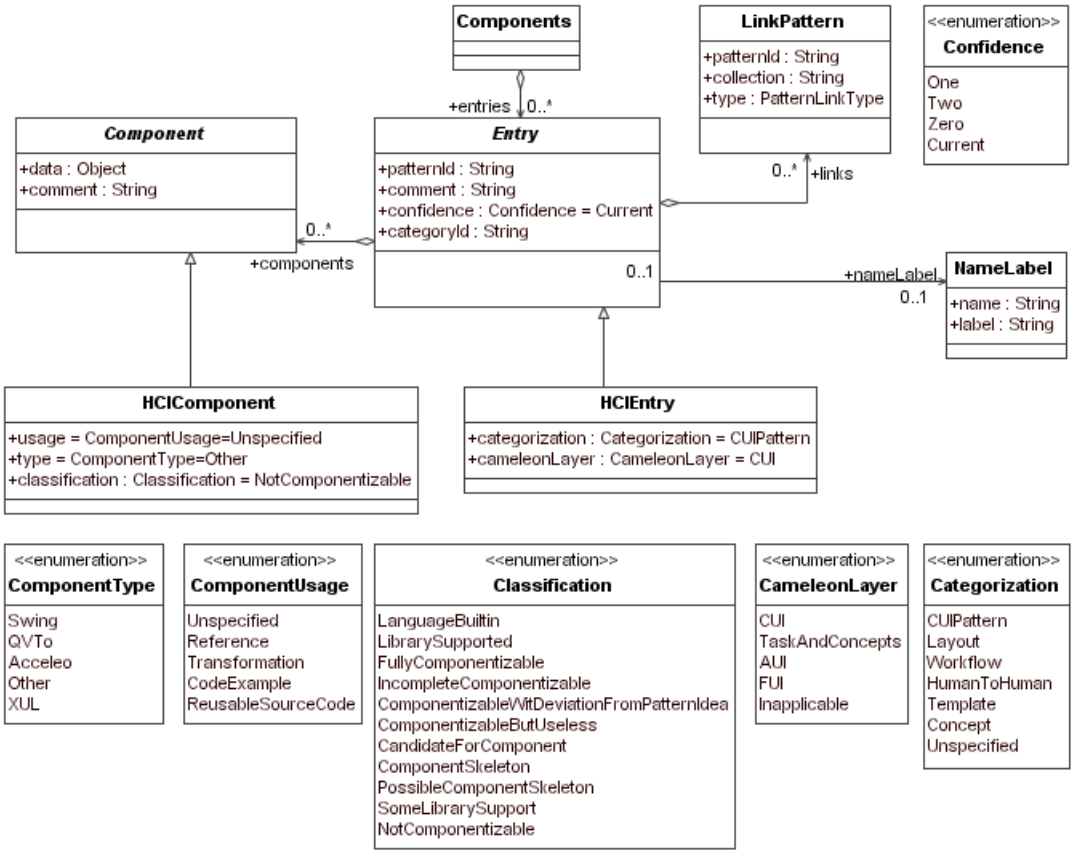

Abbildung 3.9: Metamodell für PLMLComponent

Primärschlüssel patternId:String, eingesetzt. Die einzige direkte Verwendung von Bestandteilen des PLML-Metamodells ist die Enumeration PatternLinkType zur Deklaration der Art des Links.

Die Anordnung der Klassen von Abbildung 3.9 ist nicht beliebig. Im oberen Teil dieser Abbildung sind die domänenunabhängigen Bestandteile des PLMLComponent-Modells angeordnet. Wohingegen die untere Hälfte die notwendigen Spezialisierungen für die HCI-Domäne deklariert. Würde man PLMLComponent erweitern, um damit Komponenten für andere Patternarten zu deklarieren, sollten wiederum konkrete Unterklassen von Entry und Component gebildet werden welche die spezifischen Adaptionen an diese neue Domäne beinhalten.

Die Klasse Components dient als Modellwurzel und Container für Katalogeinträge (Entry). Deren Instanzen definieren jeweils ein Pattern und dessen zugeordnete Komponenten. Die Klassen LinkPattern, für das Hinzufügen von Patternbeziehungen, NameLabel zur Korrektur von ID oder Bezeichnung des Patterns und Confidence setzen den PLML-Wartungszweck von PLMLComponent um, d.h. sie deklarieren ergänzende oder überschreibende Informationen über ein Pattern.

Die getroffene Auswahl der wartbaren Patterninformationen deckt lediglich die Basisbedürfnisse der Katalogwartung, konkret sind das die bei der Katalogverschmelzung aufgetretenden Probleme, ab. Prinzipiell ist es denkbar, auch für die hier nicht betrachteten Pattern-Aspekte eigene Informationsergänzungsklassen für PLMLComponent zu spezifizieren oder wahlweise direkt die PLML-Metaklassen geeignet zu referenzieren und weiterzuverwenden. Das entscheidende Problem ist weniger die Auswahl der Aspekte von PLML die via PLMLComponent editiert werden können, als wie die Instanzen von PLMLComponent genutzt werden können um einen PLML-Katalog zu überarbeiten.

Prinzipiell soll PLMLComponent auch zur Deklaration der Algorithmen der Patterntransformationen eingesetzt werden. Die Einbindung der jeweiligen Transformation kann über das data-Attribut der Klasse Entry erfolgen. Leider scheint es nicht sinnvoll den Datentyp dieser Assoziation genauer als als EObject zu definieren. Dies ist dadurch beding, dass an dieser Stelle alle theoretisch denkbaren Transformationstechniken angebunden werden können sollen. Weil kein übergeordnetes allgemeines Metamodell für Transformationen existiert, musste der kleinste gemeinsamen Nenner genutzt werden.

Einsatz des Komponentenmodells zur Katalogbereinigung

Das Zusammenführen zweier Instanzen von PLMLComponent und PLML bezeichne ich als Verweben, angelehnt an die Bedeutung dieses Wortes in der aspektorientierten Softwareentwicklung. Das Verweben ist eine ordinäre Model-To-Model Transformation. Als Transformationsergebnis entsteht ein neuer Katalog, d.h. eine neue PLML-Instanz, welcher die korrigierten und gefilterten Patterneinträge enthält.

Die Abbildung 3.10 illustriert das Zusammenspiel von PLMLComponent, PLML und QVT-operational (QVTo) als Transformationssprache. Selbstverständlich wird vor dem Verweben eine Reihe von Plausi-

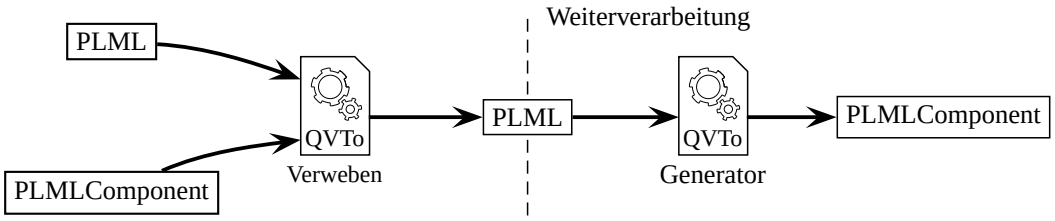

Abbildung 3.10: Zusammenhänge und Transformationen von PLML und PLMLComponent

bilitätschecks durchgeführt. Durch die Nutzung von QVTo ließen sich diese verhältnismäßig einfach als OCL-Abfragen implementieren.

Bei dem nachfolgenden Listing 3.3 handelt es sich um einen dieser Checks. Die dort deklarierte QVT-Query liefert eine Liste aller Entry-Objekte einer PLMLComponent-Instanz für die die Patternquerbeziehungen unzureichend spezifiziert sind. Diese Query wird in vier Tests benutzt, die im Einzelnen sicherstellen dass die `contradicts` und `comparable`-Relationen bidirektional spezifiziert sind, sowie die `contains` und `is-contained-by` jeweils paarweise in beide Richtungen deklariert werden.

Die `is-a`-Beziehung wird im Zusammenhang des Verwebens als Äquivalenzmarker zweier Patterns betrachtet. Eine solche Patternbeziehung führt dazu, dass bei der Transformation das Pattern in welchem der `is-a` Eintrag vorliegt aus dem Katalog entfernt wird. Während der Entfernung eines Katalogeintrages werden sämtliche bisher auf dieses Pattern verweisenden kataloginternen Verlinkungen auf das jeweils per `is-a`-referenzierte Pattern umgeschrieben. Zur Konsistenzsicherung wird deshalb im Rahmen eines weiteren PLMLComponent-Checks sichergestellt, dass für neudefinierte `is-a` Patternbeziehungen keine zirkulären Abhängigkeiten bestehen.

```
//Liefert alle Entry−Objekte für die es einen Link vom Typ fromSelector gibt,
//ohne das es im damit referenzierten Entry einen Link zurück vom Typ reverseSelector gibt
query Components::checkCrossRelation
  (in fromSelector : PatternLinkType, in reverseSelector : PatternLinkType ) : OrderedSet(Entry) {
    return self.entry
      −>select(links.type−>includes(fromSelector))
      −>select(entlent.links−>select(type=fromSelector)
        −>select(link|self.entry−>select(link.patternId=patternID).links−>select(type=reverseSelector).patternId
          −>includes(ent.patternID))
        −> size()=0);
}
```

Listing 3.3: QVTo/OCL Testselektion zur Qualitätssicherung der PLMLComponent-Instanz

Die Angaben in Tabelle 3.5 vermitteln einen Eindruck über den Umfang der, während der Modellverwebung eingearbeiteten, manuellen Nachbearbeitungen. Bei den Namensanpassungen handelt es sich um achtzehn nachgetragene Patternbezeichnungen des Welie-Kataloges, welche in dessen originalen PLML-Daten nicht angegeben waren, sowie um eine Pattern-ID Korrektur, ebenfalls bei Welie, ein Tippfehler. Gleichfalls aufgrund von Tippfehlern im Welie-PLML mussten die beiden Kategoriezuordnungen nachträglich manuell vorgenommen werden, hier handelt es sich um reine Fehlerbehebungen.

Im Gegensatz dazu sind sämtliche neu definierten Patternbeziehungen manuell getroffene Klassifikations-

Modifizierte Patterns:	98
Anzahl Namensanpassungen:	19
Anzahl Kategoriezuordnungen:	2
Anzahl `is-contained-by`-Beziehungen:	7
Anzahl `contains`-Beziehungen:	7
Anzahl `is-a`-Beziehungen:	36
Anzahl `comparable`-Beziehungen:	18
Anzahl `contradicts`-Beziehungen:	6
Anzahl `Confidence`-Reratings:	22

Tabelle 3.5: Größenordnungen zur Nacheditierung des verschmolzenen Katalogs

entscheidungen. Aufgrund der bereits erwähnten PLMLComponent-Checks ist die Anzahl der `is-contained-by` und `contains` natürlich gleich, und die Anzahl der `comparable` und `contradicts` gradzahlig. Bei den 22 `Confidence`-Neubewertungen handelt es sich durchgängig um Abwertungen, dies ist die logische Konsequenz daraus, dass bei der ursprünglichen automatischen Datenübernahme alle Einträge der Webkataloge mit dem Alexander-`Confidence`-Rating $**$ aufgenommen wurden.

3.5.4 Einordnungen von Einzelpatterns

Im Anhang der Arbeit finden sich ausführliche Tabellen zur Klassifikation der einzelnen Patterns. Daher soll an dieser Stelle nur kurz ein Eindruck über die Art der Einträge gegeben werden.

Den Anfang macht die Tabelle 3.6. Hierin ist aufgelistet, welche Patterns als äquivalent, in Idee bzw. Lösung, einem anderen Pattern sind. In der linken Spalte sind die Patterns aufgelistet die die jeweilige Pattern-Idee umfassend darstellen, während die auf der rechten Seite aufgeführten Patterns nur Teilaspekte desselben Patterns enthalten oder anderweitige Kopien sind.

Einzelne Patterns wie „Wizard“ und „Accordion“ sind in dem verschmolzenen Katalog in sehr vielen ähnlichen Variationen anzutreffen. Andere Einträge, insbesondere einige $1:1$-Zuordnungen wie bei „Row Striping“ und „Sortable Table“ sind wiederum nur Patterndoppelungen die mit der einfachen Heuristik,

Primärpattern	Zuordnungen via `is-a`
Map Navigator	Store Locator
Flyout Menu	Dropdown Chooser, Pulldown button
Guided Tour	Virtual Product Display, Wizard
Wizard	Guided Tour, Help Wizard, Product Advisor, Product Configurator
Sortable Table	Table Sorter
Center Stage	Overview and Detail, View
Global Navigation	Footer bar, Main Navigation, Repeated Menu, Footer sitemap
Collector	Purchase Process, Shopping Cart
Input Hints	Search Tips
Slideshow	Scrolling Menu, Stepping
Article Page	Product Page
Progress Indicator	Processing Page
Simple Search	Search Area
Forms	Comment Box
Accordion	Closable Panels, Collapsible Panels, Inplace Replacement, Navigation Tree
Scrolling Menu	Carrousel, Illustrated Choices, Slideshow
Row Striping	Zebra Table
Fill-in-the-Blanks	Shortcut Box
Cascading Lists	Double tab, Two-Panel Selector
Card Stack	Tabbing
Extras On Demand	Details On Demand, Inplace Replacement
Tree-Table	Product Comparison
Directory Navigation	Faceted Navigation, Sitemap, Split Navigation
Illustrated Choices	Image Menu, Scrolling Menu
Overview Plus Detail	Panning Navigator

Tabelle 3.6: Tabelle der als äquivalent zugeordneten Patterns

über die ID-Gleichheiten, bei der ursprünglichen Welie/Tidwell-Verschmelzung nicht als gleich erkannt wurden.

Durch das Entfernen der 36 `is-a` Einträge aus dem verschmolzenen Katalog konnte dieser auf 135 Patterns reduziert werden. Für diese war nun zu untersuchen welche Patterns vermutlich komponentisierbar sind.

Technisch geschieht dies durch Aufbau und Pflege einer weiteren Instanz des PLML-Component Metamodells. Die darin vorgenommenen Eintragungen sind nunmehr keine Eintragskorrekturen mehr, sondern enthalten ergänzende Komponentisierbarkeitsinformationen zu dem jeweiligen Patterneintrag. In der Prozessdarstellung in Abbildung 3.10 wird diese Weiterverarbeitungsmöglichkeit des PLML-Kataloges auf der rechten Seite der gestrichelten Linie angedeutet.

Die dazu notwendige Datenerfassung und der Aufbau der notwendigen Modellinstanz ist eine manuelle Tätigkeit. Dennoch lässt sich z.B. mit einer Model-To-Model Transformation eine gewisse automatisierte Hilfestellung geben. Da die Struktur der PLMLComponent-Ecoreinstanz, in welcher die Komponentisierbarkeitsinformationen hinterlegt werden, bekannt ist, kann eine Eintragungsschablone per QVTo aus dem zu beschreibenden PLML-Katalog generiert werden. Die benötigte Transformation legt einfach ein neues PLML-Component Modell und darin für jedes Pattern des PLML-Katalogs eine HCIEntry-Instanz mit den vorgegebenen Standardwerten an. Die Standardwerte können dem in Abbildung 3.9 dargestellten Metamodell entnommen werden.

Im Rahmen der Arbeit wurde keine zusätzliche Werkzeugunterstützung zur Pflege des Komponentenmodells erstellt, die Möglichkeiten des im Eclipse Modeling Framework stets verfügbaren reflektiven Ecore-Editors zeigten sich ausreichend.

Vorgehensweise

Die Entscheidung zur Vermutung über die Komponentisierbarkeit eines Patterns erfolgte nach folgendem Schema:

1. Kategorisierung, d.h. Zuordnung des zutreffenden Literals der Enumeration Categorization
2. Einordnung in eine CAMELEON-Ebene, zur Zuordnung dient die Enumeration CameleonLayer
3. Entscheidung Komponentisierbarkeit, sofern grundsätzlich komponentisierbar, jeweils für XUL und Swing:
   (a) Komponentisierbarkeitsklassifikation, Zuweisung eines Literals von Classification
   (b) Einbettung der Komponente/Komponentenreferenz über das data:Object-Attribut
   (c) Komponententyp setzen, d.h. Zuordnung zu einem Literal der Enumeration ComponentType
   (d) Hinweismarker für die mögliche Komponentenverwendung setzen, d.h. Zuordnung zu einem Literal der Enumeration ComponentUsage

Kategorisierung der Einträge

Bereits in der Einführung wurde begründet, dass die Darstellbarkeit einer Patternlösung als Instanz eines Algorithmus eine Komponentiesierbarkeitsvoraussetzung sein dürfte. Da die Einträge des erstellten Pattern-Kataloges eine breite Streuung bezüglich Einsatzdomäne und Anwendungsebene aufweisen, finden sich in ihm eine Reihe Patterns für die prinzipbedingt kein Algorithmus angegeben werden kann. in

einer Grobkategorisierung wird versucht solche Patterneinträge zu identifizieren. Im PLMLComponent-Metamodell sind sieben Patternkategorien vordefiniert. Die nachfolgende Auflistung nennt die Einzelkategorien und gibt jeweils ein Beispiel eines zugeordneten Patterns.

- *CUIPattern* – Pattern für konkrete oder endgültige Userinterfaces (z.B.: „Date Selector“)
- *Layout* – Pattern schlägt eine bestimmte Anordnung, teilweise im Sinne eines Layoutmanagers vor (z.B.: „Grid-based Layout“)
- *Workflow* – Patterneintrag beschreibt im Wesentlichen einen Prozessablauf (z.B.: „Purchase Process“)
- *HumanToHuman* – Marker für Verfahren der zwischenmenschlichen Kommunikation (z.B.: „Storytelling“)
- *Template* – fertige Designvorlage/-vorschrift (z.B.: „Input Error“)
- *Concept* – bezeichnet Patterns die eine Konzeption oder einen Entwurf für eine Vorgehensweise beschreiben, ohne den Detailgrad von CUIPatterns oder Layout zu erreichen (z.B.: „Few Hues, many Values“)
- *Unspecified* – Platzhalterkategorie für nicht in die vorigen Kategorien einordbare Fälle

Die Einordnung in eine der Kategorien „HumanToHuman“, „Concept“ oder „Template“ bedeutet implizit, dass wohl keine Komponente für das jeweilige Pattern angebbar ist. Insbesondere für die „Template“-Kategorie kann diese Festlegung hinterfragt werden, da je nach Betrachtungsweise Templates sehr wohl als Instanzen von Patterns betrachtet werden könnten. Die Entscheidung wurde nach Untersuchung der Einträge des Kataloges so getroffen. Bei den dort als „Template“ kategorisierten Pattern handelt es sich oftmals um Vorlagenideen für Web-CMS[10], etwa zur Gestaltung künstlerischer Homepages („Artist Site“) oder für Webseiten sozialer Netzwerke („Community“). Für eine sinnvolle Umsetzung dieser Patterns im Rahmen des modellgetriebenen Prozess nach Kapitel 2 müssten auch die dem Interface vorgelagerten Modellebenen Aufgabenmodell und Dialoggraph einbezogen werden. Die dabei zu überwindenden Probleme sind derweil noch zahlreich und in absehbarer Zeit eher nicht in Algorithmen abbildbar.

Quantitative Angaben über die Zuordnungen aller Patterns zu diesen Kategorien enthält die Tabelle 3.7. Die Kategorie „Unspecified“ wurde im Fall des verschmolzenen Patternkatalogs nicht benötigt, alle Patterneinträge konnten kategorisiert werden. Zusätzlich findet sich in der dritten Spalte der Tabelle 3.7 eine Information darüber, für wieviele Patterneinträge der jeweiligen Kategorie die Deklaration von Komponenten bzw. Komponentenreferenzen möglich scheint.

[10] Content Management Systeme

Kategorie	Anzahl eingeordneter Patterns	Anzahl Komponentenangaben
CUIPattern	77	77
Concept	26	0
HumanToHuman	3	0
Layout	4	4
Template	21	0
Unspecified	0	–
Workflow	4	2

Tabelle 3.7: Kategorisierung der 135 Patterns des verschmolzenen Katalogs

Einordnung gemäß CAMELEON

Als eine weitere Klassifizierung der Vielzahl der Patterns scheint es nützlich zu sein, den primären Anwendungsbereich der Patternlösung im Rahmen des modellgetriebenen Gesamtprozesses einzuordnen. Mit dem bereits vorgestellten CAMELEON-Referenzframework steht dazu ein geeigneter allgemein akzeptierter Standard zur Verfügung. Nach der Einordnung der Einträge des erstellten Gesamtkatalogs ergaben sich die folgenden Zuordnungen für die einzelnen CAMELEON-Ebenen:

- *Task and Concepts* – 21 Patterns, davon lassen sich vermutlich für zwei Patterns („Login“, „Registration“) auf CUI-Ebene nutzbare Komponenten entwickeln
- *Abstract User Interface* – dieser Abstraktionsebene wurden 0 Patterns zugeordnet. Das ergab sich daraus, dass die Ausdruckskraft der Modelle dieser Ebene äußerst beschränkt ist, siehe auch Unterabschnitt 2.1.3, und deren Instanzen kaum dazu geeignet sind die Patterns von Welie oder Tidwell abzubilden.
- *Concrete User Interface* – gemäß dem Zweck der Ausgangskataloge ist dies die Hauptgruppe der Zuordnungen. 87 Pattereinträge werden der CUI zugeordnet, Angaben zur Komponentisierbarkeitsvermutung existieren für 81 davon.
- *Final User Interface* – 23 Patterns, insbesondere alle 21 als *Template* kategorisierten Patterns sowie die beiden, letztlich nur auf der FUI-Ebene umsetzbaren, Konzepte „Forgiving Format“ und„Enlarged Clickarea“.
- *Unzutreffend* – vier Patterns lassen sich nicht nach CAMELEON einordnen: die drei *HumanToHuman*-Patterns und das Konzept-Pattern „Fun “.

Sprachabhängigkeit

Mit den bisherigen Erörterungen und Abgrenzungen zur Komponentisierbarkeit ließ sich die Anzahl der in Frage kommenden Patterneinträge auf 83 reduzieren. Jedes einzelne davon wurde daraufhin untersucht, ob es unter Verwendung von XUL beziehungsweise Java Swing als Komponente umsetzbar wäre. Besonders interessant war die Frage, ob es zwischen den beiden UI-Systeme Unterschiede in Bezug auf die Komponentisierbarkeit geben würde. Die Eingangsvermutung, dass es solche Unterschiede existieren, bestätigte sich im Zuge der Untersuchung.

Arnouts in Tabelle 3.3 wiedergegebene Komponentisierbarkeits-Klassen werden als allgemeines Einordnungsschema für die Komponenten weiterbenutzt. Lediglich die beiden Punkte 2.1.1 „Komponentengerüst mit Methoden“ und 2.1.2 „Komponentengerüst ohne Methoden“ lassen sich in diesem Anwendungsfall nicht nutzbringend verwenden, stattdessen wird nur deren Obergruppe 2.1 „Komponentengerüst erstellbar“ zur Einordnung benutzt.

Die Klassendiagrammdarstellung (Abbildung 3.9) des PLMLComponent-Metamodells zeigt unter anderem die Enumeration Classification. Deren Wertliterale dienen der modellbasierten Umsetzung der Klassifikation. In der Tabelle 3.8 wird der semantische Zusammenhang der Literale zum Klassifikationsschema definiert. Die erste Spalte enthält die Ordnungsnummer der Arnout-Referenzklasse und die zweite Spalte führt das zugeordnete Literal der Aufzählung auf.

Größenordnung der Klassifikationsergebnisse

Die Komponentisierbarkeitsabschätzung der in Frage kommenden 83 Patterns ergab für die Benutzung von XUL als CUI-Sprache die in Tabelle 3.9 präsentierte Vermutung. Von den 83 Patterneinträgen lassen sich

Arnout-Referenz	PLMLComponent Classification-Literal
1.1	LanguageBuiltin
1.2	LibrarySupported
1.3.1	FullyComponentizable
1.3.2	IncompleteComponentizable
1.3.3	ComponentizableWithDeviationFromPatternIdea
1.3.4	ComponentizableButUseless
1.4	CandidateForComponent
2.1	ComponentSkeleton
2.2	PossibleComponentSkeleton
2.3	SomeLibrarySupport
2.4	NotComponentizable

Tabelle 3.8: Zuordnung der Classification-Literale zur Arnout'schen Einordnung

demnach wahrscheinlich 31 Patterns direkt als Komponenten umsetzen. Das umfasst alle Einträge welche entweder bereits in der Sprache XUL integriert, vermutlich als Komponentenalgorithmus dargestellt werden können oder von denen bekannt ist, dass sie in externen Bibliotheken verfügbar sind. Für zwanzig der 83 Patterneinträge lassen sich nach den Resultaten der Untersuchung wohl keine Komponenten angeben oder erstellen.

Die Tabelle 3.10 führt die numerische Aufstellung der Komponentisierbarkeitsvermutungen mit Java Swing als CUI-System auf. In diesem Framework sind 37 der 83 Einträge vermutlich als Komponente umsetzbar oder werden bereits vom System bereitgestellt. Bei Swing lassen sich für elf Patterns wahrscheinlich keine Komponenten angeben.

Vergleichende Betrachtung

Eine vergleichende Auswertung der Auswirkung der CUI-Sprache auf die Komponentisierbarkeit ist natürlich ebenfalls reizvoll. Da es sich bei der Klassifizierung um eine Modellinstanz von PLMLComponent handelt, kann wiederum QVTo für numerische oder statistische Auswertungen eingesetzt werden. Per OCL-Anfragen lassen sich diverse Aspekte untersuchen und Abhängigkeiten oder Zusammenhänge identifizieren.

Klassifikation	eingeordnete Patterns
LanguageBuiltin	7
LibrarySupported	6
FullyComponentizable	18
IncompleteComponentizable	6
ComponentizableWithDeviationFromPatternIdea	12
ComponentizableButUseless	10
CandidateForComponent	4
ComponentSkeleton	6
PossibleComponentSkeleton	2
SomeLibrarySupport	2
NotComponentizable	10

Tabelle 3.9: Komponentisierbarkeitsvermutung mit XUL als CUI-Sprache

Klassifikation	eingeordnete Patterns
LanguageBuiltin	9
LibrarySupported	9
FullyComponentizable	19
IncompleteComponentizable	4
ComponentizableWithDeviationFromPatternIdea	16
ComponentizableButUseless	12
CandidateForComponent	3
ComponentSkeleton	5
PossibleComponentSkeleton	0
SomeLibrarySupport	2
NotComponentizable	4

Tabelle 3.10: Komponentisierbarkeitsvermutung mit Java Swing als CUI-System

Eine Möglichkeit für eine entsprechende QVT-Abfrage zeigt Listing 3.4. Aufgerufen auf ein Components-Objekt liefert diese Query alle Entry-Instanzen deren Komponenten in einer ersten Sprache als classification eingeordnet wurden, deren Komponentenklassifikation in der zweiten Sprache jedoch davon abweicht.

Damit lassen sich beispielsweise diejenigen Patterns identifizieren, für die möglicherweise keine XUL-Komponenten erstellbar sind, aber Swing-Komponenten denkbar oder zumindest die Vermutung einer Komponentenunterstützung besteht. Das Parameter-Tripel zur Abfrage dieser konkreten Konstellation mit der angegebenen OCL-Query wäre: (ComponentType::Swing, ComponentType::XUL, Classification::NotComponentizable). Damit erhielte man als Vergleichsergebnis die Katalogeinträge „Thumbnail“ , „ Diagonal Balance“ , „Movable Panels“ , Constrained Resize“ , „Inplace Replacement“ und „Composite Selection“. Bei Umkehrung der Sprachreihenfolge ist die Ergebnismenge leer, d.h. es liegen keine Einträge vor, für die eine Komponentisierbarkeit mit Swing nicht wahrscheinlich ist, jedoch für XUL möglich erscheint.

```
// Vergleich der Komponentisierbarkeit nach Sprachen, liefert alle HCIEntry−Objekte deren Komponenten
// in lang1, jedoch nicht in lang2, in classification eingeordnet worden sind
query Components::compareLanguages(
  in lang1:ComponentType, in lang2:ComponentType, in classification:Classification) : Set(Entry) {
  return
   let entries = self.entry−>select(components−>size()>0) in
    entries−>select(components.oclAsType(HCIComponent)−>any(type=lang1).classification=classification)−>
   −(entries−>select(components.oclAsType(HCIComponent)−>any(type=lang2).classification=classification)); }
```

Listing 3.4: QVTo/OCL Sprachvergleichsabfrage über dem PLMLComponent-Metamodell

Zusammenfassung und Verwendung

Der mit den Metamodellen PLML und PLMLComponent aufgebaute Patternkatalog, ergänzt um die Komponenteneinträge, soll zukünftig die Transformationen für den patternbasierten modellgetriebenen Entwicklungsprozess nach Kapitel 2 bereitstellen. Zur Untersuchung der Machbarkeit wurde in einer ersten Phase ein weniger ambitioniertes Ziel verfolgt. Der Katalog wurde an den, in Punkt 2.1.4.3, vorgestellten XUL-Editor angebunden. Damit ist es mit diesem Werkzeug und der damit verbundenen Engine möglich, die XUL-Komponenten der Patterneinträge für die Gestaltung von Oberflächen einzusetzen. Zwar zeigt dies die prinzipielle Verwendbarkeit und Plausibilität des Ansatzes, überläßt gleichwohl eine Vielzahl noch notwendiger Schritte nachfolgenden Arbeiten.

Kapitel 4

Reengineering Interaktiver Systeme

Unter der Bezeichnung Reengineering werden Methoden zur Modifikation des Quellcodes von bereits bestehenden und im Einsatz befindlichen Anwendungen subsummiert. Typischerweise werden die Systeme auf welche Reengineering angewendet wird als Legacy-Systeme bezeichnet. Das englische Substantiv Legacy[1] impliziert, dass es sich zumeist um bereits langlaufende Systeme handelt. Die genaue Schwelle, ab welcher ein System als Legacy-Software eingeordnet wird, ist jedoch bestenfalls weich definiert und kontextabhängig. Als zeitlich gesehen unterer Extrempunkt ist folgende Definition anzusehen: „Sobald man die 2. Zeile Quellcodes beginnt handelt es sich bei der 1. Zeile um Legacy-Code“ . Diese Aussage, eines Kollegen bei einem Workshop [REM09], ist zutreffend aber wenig hilfreich. Eine anschauliche Klassifizierungshilfe stammt von Bruce Trask [TR09]: „Legacy-Software ist Software deren Quellcode-Basis sich Wartung und Erweiterungen widersetzt“ .

Unter Quellcode sind hierbei jegliche Artefakte der softwaretechnischen Basis zu sehen, welche zur Bereitstellung einer einsatzfähigen Version der Software nötig sind. Das sind unter anderem Build-Skripte, auch solche für Datenbank-Schemata, jegliche Art von Resourcen-Dateien wie Bilder oder Sounds, Zertifikate und Übersetzungsdateien. Ferner zählen API-Schnittstellen, etwa Header-Files, oder komplette Bibliotheken dazu. Diese Auflistung ist nicht abschließend und kann daher, je nach den Anforderungen an die konkrete Software, Weiteres umfassen.

Der Vorgang des Reengineering zerfällt grundsätzlich in die zwei Teilbereiche Reverse- und Forward-Engineering. Oftmals fehlen für Legacy-Systeme Dokumentationen über Schnittstellen, Build-Umgebung und Packaging oder auch Teile des eigentlichen Quellcodes. Das Ermitteln und Bereitstellen dieser notwendigen Informationen wird als Reverse-Engineering bezeichnet. Der RE-Begriff umfasst also alle Verfahren und Methoden, um solche fehlenden Quellcode-Artefakte aus dem Legacy-System selbst zu extrahieren. Das Verändern und die Neukombination der so zusammengesammelten Artefakte wird dann als Forward-Engineering bezeichnet, es unterscheidet sich im Allgemeinen nicht von der jeweils üblichen Vorgehensweise zur Softwareerstellung in der jeweiligen Domäne. Im Ergebnis eines erfolgreich durchgeführten Reengineerings entsteht eine einsetzbare modifizierte Variante des ursprünglichen Legacy-Systems.

Nachfolgend beschreibt Abschnitt 4.1 eine allgemeine Vorgehensweise zur Erzeugung von Modellen aus Quellcode, diese Methode wird benutzt um ein Ecore-Metamodell für Java Swing zu erzeugen. Im anschließenden Abschnitt 4.2 wird gezeigt, wie aus der Benutzungsoberfläche einer vorhandenen, nicht modellbasiert entwickelten, Java-Anwendung eine äquivalente deklarative Instanz des zuvor erzeugten Swing-Metamodells erstellt werden kann.

[1] Altlast, Hinterlassenschaft, o.ä.

4.1 Reverse Engineering und modellgetriebene Softwareentwicklung

Im Mittelpunkt dieser Arbeit steht das Paradigma der modellgetriebenen Softwareentwicklung. Die bei weitem überwiegende Mehrzahl der heutigen Software wurde jedoch nach anderen Paradigmen entwickelt. Um die Akzeptanz von MDSE weiter zu erhöhen ist eine Methodik wünschenswert, welche es ermöglicht, existierende Softwareprojekte soweit als möglich in die MDSE-Welt zu überführen. Gleichzeitig ist es anzustreben, die Vorteile von MDSE auch auf die Wartung und Weiterentwicklung von Legacy-Software anzuwenden. Schließlich ist es eines der Kernversprechen des Paradigmas in genau diesen Problemfeldern Vorteile gegenüber den klassischen Verfahren zu haben.

Die Möglichkeit zur Übertragung existierender Systeme in einen modellgetriebenen Ansatz wird durch mehrere Faktoren bestimmt.

Konformität die Konzeption der Software sollte sich mittels des Beschreibungsmittel der Meta Object Facility darstellen lassen.

Granularität für die Software sollte sich eine Modellhierarchie ergeben.

Assoziativität es sollten sich Relationsbeziehungen zwischen Modellelementen ergeben.

Der Aspekt der Konformität spielt auf den Fakt an das die Methodik der MOF-MDSE inhärent einen objekt-orientierten Ansatz verfolgt. Je weiter das zu beschreibende System von dieser Konzeption entfernt ist, desto weniger nutzbringend wird eine Modellierung sein. Dies liegt in der Tatsache begründet, dass die Hauptbeschreibungsmittel der Objektorientierung Klassen und Objekte sowie deren Beziehungen untereinander sind. Das Anwenden dieser Beschreibungsmittel auf funktionale oder logische Programme ist natürlich möglich, erscheint aber nicht übermäßig nützlich. So kann die Syntax eines Prolog-Term oder einer Funktionsdeklaration in Haskell ohne große Probleme als Instanz eines geeigneten Meta-Modells dargestellt werden. Dennoch, die eigentliche Mächtigkeit dieser Sprachen bzw. Paradigmen entstammt ja nicht ihrer Syntax sondern den damit beschriebenen Konzepten wie Unifikation, Backtracking oder Rekursion, um nur einige Beispiele zu nennen. Dies sind jeweils Denkmodelle die sich nur schwer vollständig in MOF-Modellen abbilden lassen.

Die Granularität einer Modellbeschreibung ist ein wesentlicher Qualitätsmaßstab der Modellierung. Idealerweise ist eine solche Beschreibung so konkret wie nötig und so allgemein wie möglich. Das Modell der Anwendung muss in der MDSE die Möglichkeit bieten, daraus Quellcode zu generieren. Gleichzeitig muss das Modell, um der Wartbarkeit etc. Genüge zu tun, einen guten und korrekten Überblick über das System liefern. Aus dieser Anforderung lässt sich somit der Zwang zu einer Hierarchiebildung ableiten. Die oberste Ebene einer solchen Hierarchie liefert dabei die grundlegende Einteilung, etwa im Sinne eines Kontextdiagramms, und ein entsprechender Drill-Down auf die unterste Ebene führt bis zu den Quellcodeeinheiten in welchen die eigentliche Datenhaltung und Verarbeitungslogik definiert wird.

Ein nachgelagertes Kriterium für die Modellgüte ist der Grad der Definition von Assoziationen. Die Beziehungen zwischen den Modellobjekten sollten so genau wie möglich beschrieben werden. Die hierfür zur Verfügung stehenden Beschreibungsmittel, das sind insbesondere die Typisierung sowie die Kardinalitätenangaben von Relationen und die Angabe von Aggregationsbeziehungen, sollten ausgiebig verwendet werden. Neben den direkten Beziehungen sind an dieser Stelle auch Präzisierungen mittels der OCL anzubringen. Je genauer das Modell beschrieben werden kann, desto besser bildet es das Problem ab und desto nutzbringender wird dessen Einsatz sein.

Sicherlich muss man sich an dieser Stelle Gedanken über eine etwaige Überspezifikation machen. Unter Überspezifikation versteht man dabei im Allgemeinen exzessives Definieren von Details, ohne das dem

ein adäquater Nutzwert auf der entsprechenden Modellebene entgegensteht. Beispielsweise wäre das Hereinziehen von Implementierungsdetails auf die Modellebene, wie etwa mit OCL möglich, ein Fall von Überspezifikation. Dazu ist jedoch grundsätzlich zu sagen, dass eine Überspezifikation zwar das allgemeine Handling des Modells erschwert, aber im Gegensatz zu unterspezifizierten Modellen keine zusätzlichen falschen Interpretationen des Modells oder falsche Aktionen zulässt. Die Wahrscheinlichkeit von Bugs, also Fehlern des Modells, ist in erster Linie von der Qualifikation der Modellierer abhängig und weniger von der Größe des Modells. Aus diesem Grund stellt Überspezifikation kein wesentliches Problem dar.

Lässt sich für das zu modellierende Legacy-System keine aussagekräftige Hierarchie definieren oder enthält das resultierende Modelle nur wenige Relationen zwischen den Modellobjekten so sind dies beides Kontraindikatoren für den Einsatz von MOF-MDSE Methodik und im speziellen Fall für dessen Anwendung im Rahmen von Reengineering. Insgesamt eignet sich also Software, welche nach dem objektorientierten Paradigma entwickelt wurde, am besten für ein MOF-MDSE Reverse Engineering.

4.1.1 Reverse Engineering von Objektmodellen

Das Objektmodell der Geschäftslogik ist eines der zentralen Modelle bei Anwendung von MDSE zur Softwareentwicklung. In diesem Abschnitt werden die Möglichkeiten zur Gewinnung von Objektmodellen von objektorientierten Legacy-Anwendungen diskutiert.

Die grundsätzliche Vorgehensweise ist dabei mehrstufig und wird mit dem Ziel betrieben ein MOF, bzw. EMF, konformes Klassenmodell zu erzeugen. Die Vorgehensweise ist dabei an die oben beschriebenen Anforderungen für erfolgreiches Reengineering angelehnt und besteht aus folgenden Schritten:

1. Bereitstellung oder Beschaffung des Quellcodes
2. Ermittlung der Package-Struktur
3. Aufbauen einer Klassenhierarchie über die gefundenen Klassen im Quellcode
4. Bestimmen der Attribut- und Assoziationstypen sowie der Kardinalitäten

Bereitstellung oder Beschaffung des Quellcodes

Natürlich ist das Vorhandensein auswertbaren Quellcodes die Grundvoraussetzung für die hier dargestellte Vorgehensweise. Für alle nicht interpretierten Sprachen ist dessen Beschaffung keine triviale Aufgabe. Aus eigener Erfahrung bei der Softwareentwicklung von produktiv eingesetzter Software kann ich sagen, dass es schon mehrfach unmöglich war den genauen Quellcode für eine ganz bestimmmte Einsatzversion zu rekonstruieren. Meine Erfahrungen entstammen dabei kommerzieller OpenSource-Software. Von einer Grundversion einer Software DF4.0 wurden dabei mehrfach zum Installationszeitpunkt bestimmte ausgewählte Bugfixes oder Funktionalitätserweiterungen in den Einsatzquellcode eingepflegt, jedoch aus nicht dokumentierten Gründen nie alle. Der dabei entstandene Quellcode-Snapshot wurde normal übersetzt und das Resultat, bei zufriedenstellender Funktionalität, so beibehalten. Unglücklicherweise wurde dieser Snapshot jedoch nicht ausreichend protokolliert und dokumentiert, sondern in letzter Konsequenz gelöscht. Die Situation in diesem Szenario war letztlich so, dass zwar im Grunde bekannt war welche Evolution der Software eingesetzt wurde und der dementsprechende Quellcode auch vorlag, dennoch traten diverse Programmfehler nicht bei allen Installationen oder aber nicht mit den gleichen Folgen auf. Sicherlich hätten diese Effekte mit besserer Organisation vermieden werden können, allerdings treten sie in der Praxis offenbar immer wieder auf und sollten daher betrachtet werden.

Wenn der Quellcode einer bestimmten Anwendung nicht mehr vorliegt, und auch nicht aus Versionskontrollsystemen oder ähnlichem beschafft werden kann, muss versucht werden, durch Dekompilierung äquivalenten Ersatz zu beschaffen. Je nach dem Grad des Schutzes, welchen die originalen Entwickler der Software gegen derartige Methoden vorgesehen haben, wird die Qualität des Dekompilates ausfallen.

Dekompiler übersetzen die einzelnen Maschinensprachbefehle, oder auch Befehlssequenzen wie etwa bei Schleifen, in Standardsyntaxkonstrukte. Schon bei dieser Rückübersetzung stehen eine Reihe von Informationen aus dem Originalquellcode nicht mehr zur Verfügung.

Ein Beispiel: Bekannterweise lassen sich alle drei Hauptarten von Schleifen (kopf- oder fussgesteuert, sowie Zählschleifen) ineinander überführen, das wissen auch die Entwickler von Compilern. Auf der Ebene der Maschinensprache gibt es daher oftmals nur einen Schleifentyp. So gibt es im Bytecode der Sprache Java nur kopfgesteuerte Schleifen, also **while**-Konstrukte.

Ein weiterer wichtiger Aspekt ist die Art der im Kompilat enthaltenen Debug-Informationen und generell der Symboltabellen. Bereits die Namensgebung von Methoden, Klassen und Variablen gibt oft einen guten Hinweis auf deren Verwendung. Im Normalfall wird, sowohl in der Lehre als auch in der bezahlten Softwareentwicklung, darauf gedrungen, dass eine sprechende Namensgebung durch den jeweiligen Entwickler angewendet wird, da dies eine nachgewiesenermaßen große Hilfe bei der Interpretation und dem Verstehen von Quellcode ist.

Eine beliebte Methode zur Verschleierung des eigenen Quellcodes ist es daher, diese Benennungskonvention zu hintertreiben. Das Ziel ist es, im Kompilat keine aussagekräftigen Namen mehr zu behalten. Weil man dennoch nicht darauf verzichten möchte das der eigene Quellcode lesbar bleibt, übernehmen sogenannte Code-Obfuskatoren die Umbenennung von Variablen als Präprozessoren vor dem eigentlichen Compiler. Nachteil dieser Vorgehensweise sind insbesondere unleserliche Fehlermeldungen und schwer interpretierbare Stack-Traces.

Weitere Werkzeuge zur Erschwerung der Dekompilierung sind Exe-Packer und sämtliche Ansätze mit selbstmodifizierendem Quellcode. Exe-Packer komprimieren die Compiler-Ergebnisse. Bei der Ausführung wird zunächst ein Bootstrap-Programm geladen und gestartet welches dann online, zur Laufzeit im Speicher, dekomprimiert und das eigentliche Programm startet. Gepackte Programmdateien lassen sich jedoch auch wieder entpacken und dann einer Dekompilierung zuführen. Selbstmodifizierender Code ändert, wie der Name impliziert, sich selbst durch absichtliches Überschreiben von Instruktionen im Speicher durch andere Anweisungen. Das Dekompilieren der statischen Dateien führt daher zu vollkommen anderen Erkenntnissen als eine Untersuchung der im Speicher vorliegenden Version.

Zusammenfassend kann festgestellt werden, dass wenn derjenige Entwickler, welcher die Einsatzversion einer objektorientierten Legacy-Software erstellt hat, diese mit Debug-Informationen und ohne Code-Verschleierung kompiliert hat, ein Dekompiler möglicherweise Quellcode liefern kann der qualitativ ausreichend zur Einsteuerung in den Modellbildungsprozeß ist.

Im einfachen Fall, und sicherlich auch gar nicht so selten, steht der originale Quellcode einer Legacy-Software zur weiteren Verarbeitung direkt zur Verfügung.

Ermittlung der Package-Struktur

Zur Gruppierung von Funktionalitäten und Zuständigkeiten sieht die Objektorientierung den Mechanismus der Packages vor. Darunter werden hierarchische Container-Strukturen verstanden. Die Klassen, aus welchen ein solches Softwareprojekt besteht, werden dem jeweils semantisch passenden Container zugeordnet. Neben dieser Kategorisierungsfunktion der Package-Container sind mit einer solchen Zuordnung zumeist auch noch Konsequenzen in Bezug auf die Sichtbarkeit der einzelnen Klassen verbunden. Was wiederum

Auswirkungen auf die Zulässigkeit von Assoziationen zwischen Klassen disjunkter Packages haben kann.

Aufbauen einer Klassenhierarchie über die gefundenen Klassen im Quellcode

Das Mapping zwischen den im Quellcode vorgefundenen Klassen sowie den im resultierenden MOF-Modell anzulegenden Klassenobjekten wird zunächst 1:1 sein. Für jede Klasse des Originalquellcodes, inklusive innerer Klassen oder generischer Typparameter, wird ein entsprechendes Klassenobjekt angelegt. Dabei ist besonders darauf zu achten die Vererbungsbeziehungen korrekt abzubilden oder sie, im Falle der Typparameter, so weit wie möglich einzuschränken. Jede Klasse wird dabei genau einem Package zugeordnet. In diesem Schritt ist auch die genaue Art der Klasse zu berücksichtigen. Klassenobjekte im EMF-Ecore können abstrakte oder konkrete Klassen, Interfaces, Enumerations oder unspezifizierte Datentypen sein. Unspezifizierten Datentypen kommen zum Einsatz wenn es nicht möglich oder erwünscht ist Details zur Beschreibung eines Typs anzugeben. Diese Datentypen enthalten maximal eine Referenz zu einer sie implementierenden, meist nicht modellierten, Klasse. Für jede im Quellcode vorgefundene und als Typ identifizierte Klasse ist derjenige Datentyp zu wählen welcher dem originalen Charakter am nächsten kommt. Dies ist kein triviales Problem und hat im Einzelfall weitreichende Konsequenzen. Der gewählte Datentyp ist dann in dem passenden Package dem Modell hinzuzufügen.

Bestimmen der Attribut- und Assoziationstypen sowie der Kardinalitäten

Das Vorhandensein einer innerhalb von Packages aufgebauten Typ- bzw. Klassenhierarchie ist die Grundlage für die Spezifikation von Attributen und Assoziationen. In einem letzten Schritt wird also der Legacy-Quellcode auf diese Eigenschaften hin untersucht. Hierbei ist es wiederum nötig auf die Spezifika der MOF-Modelle einzugehen, also Assoziationen mit ihren Kardinalitäten einzutragen und Spezialfälle wie Aggregationen zu erkennen und so genau wie möglich umzusetzen. Besondere Probleme stellen in diesem Schritt die Sichtbarkeitsbeschränkungen und konstante Werte dar.

4.1.2 Kriterien für die Ableitung von MOF-Modellen aus Java-Quellcode

Der in Unterabschnitt 4.1.1 skizzierte allgemeine Ablauf wurde für Quellcode der Programmiersprache Java umgesetzt. Dabei wird als MDSE-Zielmodell ein EMF-Ecore Modell erstellt. Zur Durchführung war es nötig einige Entscheidungen bezüglich der Übertragung von Java-Konzepten auf das Ecore-Metamodell zu treffen.

Sichtbarkeiten

Für jede Klasse, jedes Interface und Attribut, wird in Java eine Sichtbarkeit bzw. Zugriffseinschränkung festgelegt. Die Angabe kann explizit mittels Schlüsselwort erfolgen, zugelassen sind **private**, **public** und **protected**. Durch Weglassen eines Schlüsselwortes nutzt man die Standardsichtbarkeit. Die Zugriffsbeschränkung resultiert teilweise auch implizit aus dem Kontext. Spezifiziert man beispielsweise ein Attribut, inklusive Wertzuweisung, innerhalb eines Interfaces, wird im Endeffekt eine **public**-Konstante angelegt.

Im EMF-Metamodell (Ecore) sind keine solchen Zugriffsbeschränkungen vorgesehen. Jedes Modellelement, welches über die formatinhärenten Mechanismen, als „XPath-like“ im Standard beschrieben, adressiert werden kann, kann referenziert werden. Dies entspricht einer **public**-Sichtbarkeit für alle Modellobjekte.

Das Ecore-Modell erlaubt es, jedes Element mit beliebigen Schlüssel/Wert-Annotationen zu versehen. Über diesen Mechanismus wäre es in jedem Fall möglich, die Sichtbarkeitsinformation in das resultierende

Modell zu übernehmen. Dies wäre jedoch nicht standardisiert und die im Ecore-Umfeld allgemein verwendeten Werkzeuge schenkten einer solchen Annotation keinerlei Beachtung. Aus diesem Grund wurde entschieden nicht auf diese Art der Implementierung zu setzen.

Stattdessen wurden die möglichen Konsequenzen dieser omnipräsenten **public**-Sichtbarkeit evaluiert. Negative Folgen im Sinne eines ungültigen Modells können nur bei folgender Konstellation auftreten: Namensgleichheit aber Typwidersprüchlichkeit von Attributen in einer Oberklasse und einer in der Vererbungshierarchie darunter liegenden Unterklasse. Zusätzlich müssen diese betroffenen Features in der Vererbungskette versteckt worden sein. Dies ist zum Beispiel der Fall wenn ein Attribut in der Oberklasse **private** war oder wenn es sich um eine Vererbung von Attributen mit Standardsichtbarkeit über Packagegrenzen handelt. Listing 4.1 illustriert den ersten Fall. Im daraus resultierenden Modell hätte die Unterklasse zwei Attribute gleichen Namens, aber mit unterschiedlichem Typ. Da Attribute in EMF nur über den Namen, nicht über deren komplette Signatur, identifiziert werden, ist in diesem Fall das Modell mehrdeutig und somit ungültig.

```
class Oberklasse {
  private String lokalesAttribut;
}

class Unterklasse extends Oberklasse {
  private Character lokalesAttribut;
}
```

Listing 4.1: Sichtbarkeitsüberlagerung bei gleichbenannten Attributen

Listing 4.1 ist kein Beispiel für einen guten Programmierstil, dennoch treten solche Fällen in der Praxis auf. Im Ergebnis ist das resultierende Ecore-Modell ungültig. Tritt eine solche Situation auf, muss immer ein menschlicher Modellierer entscheiden wie weiter zu verfahren ist. Eine pragmatische Lösung scheint zunächst zu sein, dass in der Oberklasse auftretende Attribut aus dem Ecore-Modell zu löschen; mit der Begründung, dass das Verstecken des Attributes der Oberklasse offenbar die Intention der Entwickler war. Diese Strategie könnte jedoch zu kurz greifen, denn die Tatsache, dass in einem Ast des Vererbungsbaums eine Kollision auftrat, impliziert nicht dass die Eigenschaft in den Instanzen der Oberklasse nicht benötigt wird.

Varianz bei Vererbungen in Java

Um bei Vererbungsbeziehungen zwischen Java-Klassen eine Methode zu Überschreiben müssen deren Eingabeparameter invariant definiert werden und der Typ der Ausgabe invariant oder kovariant sein. Dies ist eine strengere Forderung als sie aus dem Liskovschen Substitutionsprinzip [WB09] folgt, erfüllt dieses jedoch und ist damit typsicher. Die vorstehend dargestellten Namenskollisionen können bei Methoden ebenfalls auftreten, konkret dann, wenn eine Methode mit invarianten Eingabeparametern weiter oben in der Typhierarchie definiert und über **private** o.ä. versteckt war.

In dem Fall aus Listing 4.2 würde sich kein gültiges Ecore-Modell ergeben. Eine Methodensignatur in EMF besteht aus dem Namen und der Sequenz der Eingabeparametertypen der Methode. Für jede dieser Signaturen wird in Ecore-Modellen Eindeutigkeit innerhalb eines Typs gefordert. Im Beispiel von Listing 4.2 resultieren im Typ Unterklasse zwei Methoden mit der Signatur methode(ParamObertyp p), dies ist nicht zulässig. Die Art der Varianz des Ausgabetyps bzw. generell dessen Typ spielt für diese Art von Problemen keine Rolle, da der Ausgabetyp nicht zur Methodensignatur innerhalb EMFs zählt.

```
class ParamObertyp { };
```

```
class ParamUntertyp extends ParamObertyp { }

class Oberklasse {
    private ParamObertyp methode(ParamObertyp p) {return new ParamObertyp();}
}
class Unterklasse extends Oberklasse {
    ParamUntertyp methode(ParamObertyp p) {return new ParamUntertyp();}
}
```

Listing 4.2: Potentielles Modellierungsproblem bei kontravariantem Ausgabeparametertyp

Basistypen

Aus Praktikabilitätsgründen existieren sowohl in Java als auch in Ecore EMF vordefinierte Basistypen. Es ist sinnvoll die Basistypen von Java, sowie deren objektorientierte Äquivalente die sogenannten Wrapperklassen, auf die entsprechenden Ecore-Typen abzubilden. Tabelle 4.1 stellt die vorgenommenen Abbildungen vor.

Die interessanteren Einträge in Tabelle 4.1 sind sicherlich diejenigen mit dem „-“ in der ersten Spalte. Die Klassen String und Date zählen zwar nicht zu den Basistypen, werden jedoch so häufig eingesetzt, dass ein Standardmapping sinnvoll erscheint. Mit java.lang.Object existiert außerdem noch eine Art „catchall“-Mapping, es dient zur Übertragung von unspezifizierten Referenzen.

Relationen

Auf der Basis der Signatur eines Attributes zu entscheiden ob dieses eine navigierbare Referenz oder einen Wertcontainer darstellt ist schwierig. Da für Ecore-Modelle eine solche Unterscheidung vorgesehen ist, und es nicht sinnvoll erscheint nur den einen oder den anderen Ansatz zu verfolgen, wurde eine Heuristik zur Unterscheidung entwickelt.

Jedes Attribut einer Klasse wird auch im EMF-Modell als Attribut angelegt falls es vom Typ, oder ein davon abgeleiteter Untertyp, eines der in Tabelle 4.1 vorgestellten Basistypen ist. Ist dies nicht der Fall, wird stattdessen eine Referenz angenommen und im Modell angelegt. Die Konsequenz ist also, dass die

Java (basis)	Java (Wrapper)	→Ecore
boolean	java.lang.Boolean	EBoolean
byte	java.lang.Byte	EByte
byte[]	java.lang.Byte[]	EByteArray
char	java.lang.Character	EChar
-	java.util.Date	EDate
double	java.lang.Double	EDouble
float	java.lang.Float	EFloat
int	java.lang.Integer	EInt
long	java.lang.Long	ELong
-	java.lang.Object	EJavaObject
short	java.lang.Short	EShort
-	java.lang.String	EString

Tabelle 4.1: Mapping von Java-Typen auf die Ecore-Äquivalente

automatische Transformation für Beziehungen zwischen den Klassen eines Modells stets Assoziationen erzeugt und diese nie als Attribute angelegt werden.

Aggregationen

Komposition und Aggregation, als spezielle Assoziationen, sind wichtige Konzepte der Objektorientierung. Jedoch finden sich weder in Java noch in EMF Syntaxkonstrukte um diese explizit umzusetzen. Ecore kennt die Spezialbeziehung Containment, welche in der Modellierung einer Aggregation entspricht.

Zur Erkennung von Aggregationen im Java-Quellcode wurde wiederum eine Heuristik entwickelt. Indikatoren für eine Containment-Beziehung finden sich in der Typsignatur. Entspricht diese einem 1-dimensionalem Array oder handelt es sich dabei um einen Untertyp des java.util.List oder java.util.Collection Interfaces wird das Vorliegen einer Aggregation angenommen und eine solche mittels der Ecore Entsprechung angelegt. Diese möglicherweise trivial erscheinende Herangehensweise erwies sich bis dato als durchaus nützlich.

Generische Klassen

Seit geraumer Zeit, konkret seit der Java Version 1.5, sind auch in dieser Sprache generische Klassen möglich. Damit einher gehen die Syntaxkonstrukte der Typparameter, dies sind normalerweise einfache Namenssymbole. Durch das Anlegen dieser Namenssymbole als eigenen Typ im Ecore-Modell kann das Konzept direkt abgebildet werden.

Es ist jedoch wünschenswert, diese Pseudotypen etwas genauer zu beschreiben. Oft ist es möglich den Typen des Parameters auf einen weniger allgemeinen Typen als java.lang.Object aufzulösen. Wenn dies zutrifft, wird statt des Namenssymbols dieser Typ an den Stellen des Typparameters eingesetzt.

Getter and Setter

Den Zugriff auf Attribute nur über spezialisierte Lese(get) und Schreibmethoden(set) zuzulassen ist ein Standardkonzept, nicht nur in Java-Programmen. Da das Ecore-Modell jedoch von der Implementierung abstrahiert, und diese Methoden auch nicht zur Präzisierung des Modells bzw. des Problems beitragen, ist es nicht zweckmäßig solche Methoden im Modell beizubehalten. Im Gegenteil, verwendet man das EMF Modell als Ausgangsbasis für die Codegenerierung so kommt es mit den Standardtemplates zu Namenskollisionen bei Beibehaltung der Get-/Set-Methoden, da diese ohnehin automatisch generiert werden.

Aufzählungen

Aufzählungen, bzw. Enumerationen, werden oft angewendet um diskrete Wertebereiche zu definieren. Aktuellerer Java-Quellcode, wiederum ab Sprachversion 1.5, kann sich dafür des Sprachkonstruktes **enum** bedienen. Enumerations wurden von den Sprachdesigner als Klassen umgesetzt. Diese können wie normale Klassen mit Attributen und Methoden versehen werden. EMF bietet hierfür den Spezialtyp EEnum, für diesen sind keine Attribute oder Methoden möglich. Die Abbildung von Java auf den Ecore-Typ kann demgemäß nicht verlustfrei erfolgen. Es ist nur möglich die Literale und ggf. den Ordinalwert im Ecore-Modell anzulegen, für die große Mehrzahl der Fälle scheint dies jedoch ausreichend.

Simple Enumerationen, also einfache Name→Wert-Beziehungen werden oft auch über simple Listen von Konstanten implementiert. Diese Art Aufzählung kann möglicherweise über Heuristiken identifiziert werden. Es wurde hier jedoch kein solcher Ansatz verfolgt.

Konstanten

Konstanten werden in Java als unveränderliche Klassenattribute, abgelegt in den Meta-Klassen, implementiert. Eine Unterscheidung zwischen Objektattribut und Klassenattribut existiert in EMF nicht, auch sonst kein Konstantenkonzept. Um dennoch nicht auf das in den Konstanten abgelegte Wissen zu verzichten, wird versucht über Steuerattribute die Konstanteneigenschaft bestmöglich abzubilden.

Jede Konstante einer Klasse wird dazu als normales Attribut der Klasse angelegt, jedoch als nicht änderbar und unveränderlich deklariert. Zusätzlich wird natürlich der eigentliche Wert im Modell abgelegt. Dieses Vorgehen ist nicht optimal, aber mit der EMF-Philosophie kaum anders vereinbar.

Sonstige technische Aspekte

Die Konzepte Interface, abstrakte Klasse und Vererbung existieren in EMF und werden beim Java⇒Ecore-Mapping umgesetzt. Die Tatsache, dass Java Mehrfachvererbung nur für Interfaces zulässt, für normale Klassen jedoch nicht, stellt bei der Generierung eines Ecore-Modells kein Problem dar, da EMF hier mächtiger ist.

Wie in Unterabschnitt 4.1.1 beschrieben ist die Package-Struktur ein wichtiger Punkt eines Modells. Da Java selbst dieses Konzept bietet ist hier wiederum nur eine einfache 1:1 Abbildung erforderlich; zwecks Inter- und Intramodellreferenzierung ist es dabei nötig, für jedes EMF-Package eine eindeutige URI sowie ein eigenes Präfix zu definieren. Beides kann einfach aus dem Packagenamen abgeleitet werden, schließlich sind diese zumindest in der vollklassifizierten Fassung eindeutig.

Ein weiteres Problem stellen diejenigen Typen dar, welche zwar referenziert werden, aber im untersuchten Quellcode-Abschnitt selbst nicht definiert sind. Hier kann es sich z.B. um Bestandteile der Java-API handeln oder um andere Klassen, die erst zur Laufzeit aus diversen anderen Quellen bereitgestellt werden. Diese externen, aus Sicht der Quellcode-Auswertung, Referenzen müssen dennoch definiert werden, um die Referenzierungsbeziehungen beizubehalten. Für diesen Zweck wird ein eigenes Package mit dem eindeutigen Namen nonlocal angelegt und die externen Referenzen als unspezifizierte Typen dorthin zugeordnet.

4.1.3 Durchführung der Java⇒Ecore Transformation

Unter Berücksichtigung der in Unterabschnitt 4.1.2 festgelegten und begründeten Kriterien wurde eine Anwendung entwickelt, welche in der Lage ist den Quellcode beliebiger Java-Anwendungen in EMF Modelle umzusetzen. Dazu bedient sie sich des bekannten JavaDoc-Frameworks.

Das JavaDoc-Framework wurde aus mehreren Gründen für diesen Einsatzzweck ausgewählt. Zum einen ist es frei erhältlich, zum anderen findet die Untersuchung des Java-Quellcodes mittels des gleichen Parser statt welcher auch vom Java-Compiler verwendet wird. Dadurch ist sichergestellt, dass alle Sprachfeatures korrekt, d.h. in diesem Fall genau so wie sie der Compiler interpretiert, aus dem Quellcode ausgelesen werden. Neben diesem Parser stellt das Framework ein Metamodell zur Verfügung, welches der Beschreibung der geparsten Informationen dient. Die eigentliche EMF-Erzeugung kann als eine Model-To-Text Transformation von diesem Metamodell in EMFs XMI-Serialisierung durchgeführt werden.

4.1.3.1 Java-Metamodell und M2T-Transformation

Die Abbildung 4.1 stellt die Typen und die Vererbungsbeziehungen in JavaDocs Metamodell dar. Dieses Modell wurde mittels der hier beschriebenen Anwendung erzeugt und ist das Ergebnis der Transformation

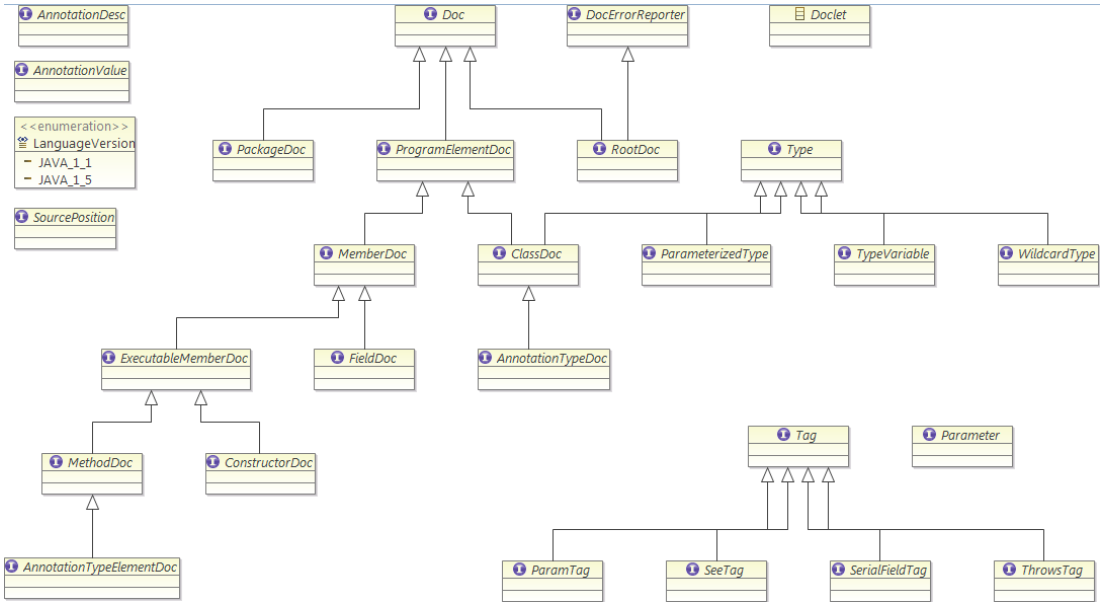

Abbildung 4.1: JavaDocs Java-Metamodell ohne Methoden

des Package com.sun.javadoc, des frei erhältlichen Java-Quellcodes. Die Erzeugung der graphischen Darstellung erfolgte mittels der Standard EMF-Mechanismen. Wie zu sehen ist, besteht dieses Metamodell im Wesentlichen aus Schnittstellenbeschreibungen mit der Ausnahme einer Enumeration. Die Verwendung der beide dort definierten Literale an geeigneter Stelle entscheidet darüber ob der Parser von JavaDoc die neueren Spracherweiterungen ab Java 1.5 berücksichtigt oder nicht. Die diversen Interfaces spezifizieren eine Reihe get-Methoden welche bei der Auswertung verwendet werden um die gewünschten Informationen des jeweiligen Typs auszulesen. In Abbildung 4.1 wurden diese Methoden aus Platzgründen nicht mit dargestellt. Die zugehörigen, also die Schnittstellen implementierenden, konkreten Klassen sind nicht von Belang.

Ablauf der Transformation

Das Generieren eines EMF-Modells aus dem Quellcode unterscheidet sich im Ablauf nicht von der Erzeugung der bekannten JavaDoc API-Übersicht. Zunächst erstellt der Parser die entsprechende Instanz des Metamodells, sämtliche Informationen darin sind ausgehend von einer Instanz des Interface com.sun.javadoc .RootDoc erreichbar. Dieses RootDoc-Objekt wird einer Template-Engine übergeben, diese ist für die weitere Transformation zuständig. In der JavaDoc-Terminologie wird die Template-Engine Doclet genannt. Jede Klasse welche eine Methode mit der Signatur **public static boolean** start(RootDoc rootDoc) bereitstellt, ist ein solches Doclet. Es bestehen prinzipiell keine weiteren Anforderungen. Das Standard-Doclet generiert die typische HTML API-Übersicht. Für die Erzeugung des Ecore-XMI musste also ein adäquates Doclet erstellt werden.

Die Tabelle 4.2 zeigt die wesentlichen Einstellparameter des EMF-Doclets und deren Voreinstellung für die Transformation. Die obersten fünf Stellgrößen, welche Elemente überhaupt im Ergebnismodell angelegt werden, scheinen nicht erklärungsbedürftig. Zu den weiteren Einstellungen: standardmäßig erfolgt die Erkennung von Gettern und Settern rein namensschemabasiert. Dennoch können Fälle auftreten, in denen diese vereinfachte Erkennung nicht ausreicht und *false positives* auftreten. Um dem entgegenzuwirken, kann optional zusätzlich zum Namensschema eine invariate Typübereinstimmung der Parameter zur

Eigenschaft	Standardwert
Methoden in Modell anlegen	ja
Attribute in Modell anlegen	ja
Konstanten in Modell anlegen	ja
Getter in Modell anlegen	nein
Setter in Modell anlegen	nein
Typinvarianz zur Accessor-Erkennung fordern	nein
Innere Klassen in Modell anlegen	ja
Eigene Packagehierarchie für innere Klassen	nein
Leere Packages verhindern	ja
vollqualifizierten Klassennamen als Attributwert setzen	ja
Serialisierungsidentifikatoren unterdrücken	ja
Sichtbarkeitslevel	**protected**
Verzeichnisse, Packageselektor-Muster, diverse JavaDoc-Steuerparameter	

Tabelle 4.2: Parameter zur Steuerung der Java⇒Ecore Transformation

Getter- und Setter-Ermittlung gefordert werden.

Einen eigenen Problemkreis stellen die inneren und anonymen Klassen dar. Anonyme Klassen entstehen, wenn Interfaces oder abstrakte Klassen direkt bei Zuweisungen implementiert werden, ohne zuvor explizite Klassendeklarationen durchzuführen. Anonyme Klassen sind also nur konkrete Instanzen anderer, nicht instanziierbarer, Modelltypen. Es ist daher nicht notwendig sie dem Modell hinzuzufügen.

Innere Klassen sind in anderen Klassen eingebettet. Praktisch können Instanzen innerer Klassen nur über Instanzen ihrer Host-Klassen erstellt werden. Es besteht also ein enger semantischer Zusammenhang zwischen diesen Typen, dieser kann jedoch in einem Ecore-Modell nicht mit den Mitteln des EMF nachmodelliert werden. Stattdessen werden die inneren Klassen ebenfalls als Klassentypen im Ecore-Modell angelegt.

Aus zwei Gründen ist jedoch im Java⇒Ecore-Generator eine Sonderbehandlung vorgesehen. Zum einen wird durch das Vergeben eines Namenspräfixes, nämlich dem Namen des Host-Typs, zumindest marginal der ursprüngliche Zusammenhang dargestellt. Wichtiger ist jedoch ein praktisches Problem. Innere Klassen können zahlreich verwendet worden sein und es kann dabei tatsächlich der Fall auftreten, dass zu viele Typen innerhalb eines Ecore-Packages definiert werden. Zu viele sind in diesem Fall mehrere hundert verschiedene Typen in einem Package. Theoretisch stellt dies kein Problem dar, praktisch entsteht ein unüberwindliches Problem mit dem EMF-Codegenerator für Java. Dieser generiert Methoden deren kompilierter Bytecode größer als 64kByte wird. So große Methoden sind nach dem Sprachstandard nicht zugelassen und im Bytecode-Format nicht darstellbar. Im Standard sind für lokale Sprünge 2 Byte große Zeiger vorgesehen, welche daher keinen größeren Adressraum adressieren können. Zur Umgehung dieser Problematik, oder vielmehr um den Schwellwert ab welchem diese auftritt nach oben zu verlegen, ist es möglich, die generierten Typen für die inneren Klassen in einer eigenen Packagehierarchie abzulegen.

Die folgenden beiden Parameter der Tabelle 4.2, vollqualifizierte Klassennamen und Identifikatattribute, dienen gleichfalls der Behandlung von Eigenheiten der Codegenerierungstemplates von EMF. Der Quellcode einer Java-Anwendung ist im Allgemeinen in einen mehrere Hierarchieebenen tiefen Packagebaum aufgeteilt. Oft enthalten die Knoten nahe der Wurzel keine Klassen, sondern dienen nur der Strukturierung und Namensgebung. Aus praktischen Gründen sollte diese Eigenschaft so nicht in das EMF-Modell übertragen werden. Leider handhaben die Code-Generierungstemplates typleere Packages fehlerhaft und generieren ungünstigen oder auch falschen Quellcode dafür. Es ist deswegen vorgesehen, dass das Doclet in Packages welche keine anderweitigen Typdeklarationen enthalten explizit einen eigenen leeren Typen

anlegt.

Aus einem ähnlichen Grund ist die Übernahme des vollqualifizierten Klassennamen in das dafür eigentlich vorgesehene Attribut instanceClassName abschaltbar. Denn, ist dieses EMF-Attribut für einen Typ mit einem Wert belegt, so wird für diesen Typ kein Quellcode generiert, sondern das Vorhandensein einer externen Implementierung angenommen und diese lediglich importiert. Dies gilt zumindest für die Verwendung der standardmäßigen Code-Erzeugungstemplates.

Serialisierungsidentifikatoren, das ist insbesondere das Attribut serialVersionUID, sind Klassenattribute zur entwicklergesteuerten feingranularen Angabe der Implementierungsversion einer Klasse. Für alle serialisierbaren Klassen, d.h. Implementierungen des Interfaces java.lang.Serializable, ist die Vergabe einer solchen ID vom Sprachstandard vorgesehen. Mit dem Übergang in einen MDSE-Prozess ist eine derartige Versionierung auf Klassenebene meist nicht mehr notwendig. Daher kann insbesondere dieses Attribut bei der Transformation explizit herausgelöscht werden.

Neben den EMF-Doclet spezifischen Stellgrößen stehen weiterhin alle Einstellparameter von JavaDoc selbst zur Verfügung. Von besonderem Interesse sind dabei die minimale Sichtbarkeit und die Parameter zur Auswahl der gewünschten Packages.

Nachbearbeitung

Im Ergebnis der Transformation entsteht eine Ausgabedatei in welcher das Modell als Ecore-XMI serialisiert ist. Dieses Modell ist nicht notwendigerweise valide, es kann also die Erfordernis zu manueller Nacharbeit bestehen. Das EMF stellt einen Validator bereit, mit welchem die Modelle getestet werden können.

Neben dem bereits erwähnten Themenkomplex der Sichtbarkeitsproblematiken bei Vererbungen treten typischerweise weitere Probleme auf. Beispielsweise bei der Namensgebung, Java unterscheidet Groß-/Kleinschreibung und EMF im Prinzip auch, allerdings wird bei Ecore dringend davon abgeraten. Bei der Codegenerierung werden diverse Variablen- und Konstantennamen aus den Namen der Attribute und Methoden des Typs abgeleitet. Diese Ableitung erfolgt für Features welche sich nur in der Schreibweise unterscheiden, nicht eindeutig. Daher wird der EMF-Modellvalidator in solchen Fällen eine Warnung ausgeben. Aus ähnlichen Gründen dürfen sich Namen nicht nur in der Anzahl der vorangestellten Unterstriche unterscheiden, es darf also beispielsweise nicht _x und __x als Namen innerhalb desselben Typs geben. Bei Auftreten solcher Fälle muss manuell einer der betroffenen Namen geändert werden oder die betroffene Eigenschaft ganz entfernt werden, eine automatische Behandlung erscheint nicht sinnvoll.

4.1.4 Reduzierung der Modelle

Die Modelle, die als Ergebnis der Transformationen in Punkt 4.1.3.1 entstehen, können sehr groß werden. Für den Quellcode eines Releases der Java-Version 1.5 ergaben sich die in Tabelle 4.3 gezeigten Größenordnungen. Selbst das kleinste Modell, welches beim Sichtbarkeitslevel **public** entsteht, ist zu groß um es

Sichtbarkeit	Typen	Methoden	Attribute	Externe Typen	leere Packages
public	12.689	79.993	891	80	99
protected	13.380	87.193	3.660	96	101
default	16.418	106.007	7.010	102	109
private	18.323	120.736	18.243	152	108

Tabelle 4.3: Modellgrößen für diverse Sichtbarkeitsstufen beim JDK-Parsing

sinnvoll mit den zur Zeit zur Verfügung stehenden Werkzeugen des EMF zu behandeln.

Beim Studium der Tabelle 4.3 fällt auf, dass die Anzahl der leeren Packages mit zunehmender minimaler Sichtbarkeit abnimmt, ein zunächst überraschender Effekt. Dieser erklärt sich aus der Art und Weise wie Packages identifiziert werden. In der vorliegenden Implementierung werden nur die Packages angelegt, in welchen Datentypen existieren und natürlich deren übergeordnete Packages. Mit der reduzierten Anzahl der erkannten Klassen sinkt somit im Regelfall auch die Anzahl der erkannten Packages, und damit auch die Anzahl der leeren Packages.

Bei diesen großen Modellen erweisen sich sowohl der Speicherbedarf als auch die Laufzeit von Transformationen als problematisch, beispielsweise wenn versucht wird, eine graphische Übersicht ähnlich derjenigen von Abbildung 4.1 zu generieren. Dies ist kein besonderer Nachteil des EMF, auch andere Modellierungswerkzeuge skalieren schlecht bei größer werdenden Modellen.

Um dennoch einen Nutzen aus derartig großen Modellen zu ziehen ist eine Modellverkleinerung anzustreben. Für eine Reihe von Einsatzzwecken in der MDSE wird nicht das gesamte Modell benötigt, sondern es sind Ausschnitte ausreichend. So werden Transformationen, mit Ausnahme kleiner Modelle, nur in seltenen Fällen alle Typen und alle deren Eigenschaften eines Modells ändern oder auslesen. Die EMF-Modelle die durch die Java⇒Ecore Transformation entstehen, enthalten neben den Domänen-Konzepten in aller Regel auch eine Reihe von implementierungsspezifischen Eigenschaften. Beispielhaft seien hier Methoden für Tracing-Ausgaben oder zur Behandlung von Bedienungs- oder Eingabefehlern genannt. Zur weiteren Bearbeitung im Rahmen der MDSE kann es wünschenswert sein, solche Modelleigenschaften zu entfernen oder diese zumindest auszublenden.

EMF selbst sieht zur Bildung von Modellausschnitten nur das bereits mehrfach angesprochene Konzept der Packages vor. Dieses führt, je nach Qualität der Umsetzung, zu einer gewissen lokalen Clusterung von Typen nach deren Verwendung.

Sen et al. [SMBJ09] verfolgen einen algorithmischen Ansatz. Der in Algorithmus 1 vorgestellte Prozess dient dazu, aus einem übergebenen Modell M_0 ein reduziertes Ergebnismodell E_0 zu generieren welches *konform* zu M_0 ist. Konformität ist definiert als Enthaltenseinsbeziehung auf der Ebene der Klasseneigenschaften:

> Ein Metamodell E_0 ist konform zu einem Metamodell M_0, wenn für jede Klasse C in E_0 genau eine Klasse C' in M_0 existiert, so dass alle Eigenschaften und Methoden von $E_0.C$ mit gleicher Signatur in $M_0.C'$ definiert sind. Eine solche Beziehung wird mit dem Operator „$< \#$“ , als $E_0 < \#M_0$ notiert.

Der Algorithmus nach Sen basiert auf der Idee, ausgehend von einer kleinen Menge initial bestimmter Typen und Eigenschaften „von Interesse“ ein minimales beschnittenes Modell zu erzeugen, welches dennoch konform zum Ausgangsmodell ist. Es werden verschiedene Strategien zur Durchführung der Modell-Beschneidung vorgeschlagen, die prinzipielle Strategie stellt Algorithmus 1 vor.

Die Referenz-Implementierung des Algorithmus erfolgte durch Sen innerhalb der Kermeta-Umgebung [Ker]. Dabei handelt es sich um spezialisiertes Framework zur Bearbeitung und Transformation von Metamodellen, vergleichbar dem Eclipse Modeling Project. Kermeta ist in der Lage Ecore-Modelle zu im- und exportieren, die Resultate der Java⇒Ecore Transformation lassen sich daher innerhalb Kermeta weiterbearbeiten. Insgesamt ist festzuhalten, dass die Qualität und der Umfang des resultierenden verkleinerten Ergebnismodell von der geeigneten Wahl der initial geforderten Eigenschaften abhängig ist.

Algorithmus 1 Modellreduzierungsalgorithmus nach [SMBJ09]

1. Wähle Typen und Eigenschaften aus M_0 die das beschnittene Modell enthalten **muss**
2. Initialisiere das Ergebnismodell E_0 mit der Auswahl von 1
3. Validiere E_0, wenn valide und konform zu $M_0 \Rightarrow 8$.
4. Entferne nicht referenzierte Attribute (unter Beachtung der Vererbungshierarchie) aus den E_0-Datentypen
5. Füge nicht enthaltene referenzierte Vererbungsobertypen aus M_0 in E_0 ein
6. Füge nicht enthaltene referenzierte Datentypen aus M_0 in E_0 ein
7. Wiederholen bei $\Rightarrow$3
8. Ergebnismodell liefern

Zusammenfassung

Im vorliegenden Kapitel wurde eine grundsätzliche Möglichkeit diskutiert, um aus Java-Quellcode Ecore-Modelle abzuleiten. Die praktische Umsetzbarkeit wurde gezeigt, aber auch die dabei auftretenden Probleme herausgestellt. Im nachfolgenden Abschnitt 4.2 wird die Methodik eingesetzt, um ein EMF-Modell für das Oberflächenframework Swing zu erstellen. Mittels dieses Swing-Metamodells, welches als CUI-Modell betrachtet wird, können die Oberflächen von Swing-Anwendungen in einem UI-MDSE Prozess beschrieben werden.

4.2 Reverse Engineering des Oberflächenmodells

In diesem Abschnitt wird dargestellt, wie unter Verwendung der in Abschnitt 4.1 vorgestellten Techniken aus existierenden Nutzeroberflächen von Legacy-Anwendungen ein konkretes Oberflächenmodell gewonnen wird. Die Grundidee dafür ist, zur Laufzeit die Objektstruktur der Oberfläche auszuwerten und diese in einer Instanz eines geeigneten Metamodells abzubilden. Durch Transformation einer solchen Modellinstanz ist es in einem weiteren Schritt möglich ein abstraktes Oberflächenmodell abzuleiten. Auf diesem Weg kann eine Einsteuerung in den normalen modellgetriebenen Oberflächenerzeugungsprozess, siehe Kapitel 2, erreicht werden.

Die Programmierung von Oberflächen ist eine komplexe Aufgabe. Zwar gibt es seit langer Zeit unterstützende Werkzeuge und verschiedene strukturierende Vorgehensmodelle, wie etwa das MVC-Pattern, viele Detailaspekte dynamischer Nutzerschnittstellen werden jedoch weiter manuell implementiert. Eine Folge davon scheint zu sein, dass oftmals UI-relevante Quellcode-Teile über den gesamten Quelltext verteilt werden und nicht ausschließlich in spezialisierten Packages zu finden sind. Daher ist es sehr schwierig, mit Hilfe der Methodik aus Abschnitt 4.1 oder auch über andere Mechanismen, durch rein statische Code-Analyse ein zutreffendes Modell der Oberfläche zu gewinnen. Eine gangbare Alternative ist es das zur Laufzeit instanziierte Objektmodell des Oberflächenframeworks zu untersuchen und, falls möglich, in eine CUI-Instanz zu transformieren.

4.2.1 Erzeugen eines Swing-Modells aus einer existierenden Oberfläche

Die Grundforderung für den erfolgreichen Einsatz dieser Vorgehensweise ist es, dass die verwendete Oberflächentechnologie überhaupt Objekthierarchien zur Beschreibung des Nutzerinterfaces einsetzt. Programme die das Anzeigegerät direkt pixelweise ansteuern erfüllen diese Forderung typischerweise nicht. Vie-

le Oberflächenframeworks der letzten Jahre verwalten jedoch intern den UI-Zustand über baumähnliche Objektstrukturen. Inwiefern deren Objektinstanzen zur Laufzeit der Introspektion offenstehen ist weiter abhängig vom Betriebssystem, allgemeinen Sicherheitsmechanismen und ob es sich um eine interpretierte oder kompilierte Sprache handelt. UI-Toolkits von interpretierten objektorientierten Sprachen sind ideale Kandidaten für diesen Ansatz. Für die Sprache Java stehen drei weitverbreitete Toolkits zur Verfügung: Swing, AWT und SWT.

Für eine Fallstudie zur Prüfung inwieweit der Laufzeit-Capturing Ansatz praktikabel ist, wurde entschieden die Oberfläche von Swing-Applikationen als Modellinstanzen eines CUI-Modells abzubilden. Zur Durchführung waren, neben einer entsprechenden Swing-Anwendung, mehrere technologische Voraussetzungen zu schaffen. Einerseits musste ein geeignetes Metamodell für Swing erstellt werden, andererseits musste ein Weg gefunden werden auf Objektinstanzen einer aktiven Java-Laufzeitumgebung zuzugreifen.

Das Swing-Metamodell wurde mit dem in Punkt 4.1.3.1 vorgestellten Ansatz erzeugt. Hierzu wurde der Quellcode unterhalb der Packages javax.swing und java.awt geparst. Wie oben beschrieben, ergänzt Swing AWT, viele Konzepte und vorhandene Implementierungen für Teilprobleme aus AWT werden von Swing daher weiter benutzt. Darunter sind Kernkonzepte wie die Implementierungen zu Schriftarten und Farben. Für ein sinnvolles, d.h. aussagekräftiges, Swing-Modell schien es daher angebracht, auch die AWT-Klassen in das Metamodell zu übernehmen.

Beim Studium des Quellcodes von Swing wurde festgestellt, dass in den Klassen viele Methoden implementiert wurden, die für ein rein die Oberfläche beschreibendes Metamodell irrelevant sind. Als weitere Einschränkung kam hinzu, dass ein Screenshot nach seiner Natur statisch ist; es besteht demgemäß keine Möglichkeit die Dynamik des Nutzerinterfaces abzubilden. Im Ergebnis dieser Überlegungen wurde zunächst entschieden, aus den vorgenannten Java-Packages nur die Attribute und Klassen mit der minimalen Sichtbarkeit **protected** zu parsen. Bei der Benutzung dieser Variante des Metamodells stellte sich jedoch heraus, dass viele Eigenschaften der UI-Widgets als **private** Attribute deklariert und nur über diverse **public** Accessormethoden zugänglich waren. Um diese Attribute dennoch in das Meta-Modell zu übernehmen, musste die minimale Sichtbarkeit doch auf **private** reduziert werden.

Im Ergebnis dieses Parsing-Durchlaufs entstand ein Meta-Modell aus 3.128 Typen mit 4.982 Attribute, dazu 147 externe Referenzen. Es handelte sich also wiederum um ein sehr großes Modell. Durch Anwendung des Model-Pruning Ansatzes von Unterabschnitt 4.1.4 konnte das Modell auf circa ein 20stel seiner Größe reduziert werden. Auf diese Weise ließ sich die Handhabung innerhalb des EMF, beziehungsweise generell innerhalb des Eclipse Modeling Project, erheblich erleichtern.

Die verhältnismäßig drastische Reduktion erklärt sich insbesondere dadurch, dass viele Klassen im Swing-Quellcode nur aus Implementierungsgründen notwendig sind. Für die hier angewendete Modellierung sind aber eigentlich nur die Schnittstellen interessant, diese definieren letztlich die überall verfügbare Funktionalität von Swing. Beispiele für entfernte Modellelemente sind Standardimplementierungen für allgemeine Aktionen wie Cut&Paste, Behandlung von Maus-Ereignissen oder auch die Datenmodelle der verschiedenen UI-Widgets.

4.2.1.1 Zugriff auf die Swing-Objektstruktur

Wie bereits dargelegt, verwaltet das Swing-Framework die dargestellte Oberfläche intern in einem Objekt-Graph. Es kann dabei nicht von einem Baum gesprochen werden, da navigierbare Querbeziehungen zwischen den Objekte zugelassen und vorgesehen sind. Der Einstiegspunkt in diesen Graphen ist das oberste Fensterobjekt: der javax.swing.JFrame. Ausgehend von diesem sind sämtliche Eigenschaften aller dargestellten UI-Widgets erreich- und auslesbar.

Die primäre Herausforderung bestand daher darin, Zugriff auf die durch eine Anwendung erzeugten JFrame -Instanzen zu erhalten. Selbstverständlich existiert innerhalb des Swing-Framework eine Liste aller erzeugten JFrames. Zum Abrufen dieser Verwaltungsinformation ist es allerdings erforderlich, einer Java-Applikation Zugriff auf den gleichen Speicherbereich wie die zu untersuchende Anwendung zu ermöglichen. Dies erscheint nur durchführbar wenn beide Programme in der gleichen Laufzeitumgebung existieren. Weil es sich bei der Java-Laufzeitumgebung um einen einzelnen Prozess handelt, kann eine parallele Existenz nur über Threading-Mechanismen implementiert werden. Daher werden sowohl der Swing-Parser als auch die Anwendung als eigener Thread gestartet. Das Starten und Konfigurieren dieser Threads übernimmt eine eigene Steuerungsapplikation. In Abbildung 4.2 sind die Architektur und der schematische Ablauf dargestellt. Die umgebende Steuerungsapplikation wird in Abbildung 4.2 als Container bezeichnet.

Hauptzweck dieses Containers, neben dem Starten der Threads, ist es, die Ladepfade des Klassenlademechanismus der Java-Laufzeitumgebung so zu konfigurieren, dass das Laden der Resourcen und des Bytecodes der Klassen in der von der Applikation ursprünglich vorgesehenen Art und Weise funktioniert. Generell hat sich diese Architektur als tauglich für den gedachten Einsatzzweck erwiesen. Als problematisch haben sich jedoch verteilte Anwendungen und Programme die einen eigenen Classloader einsetzen herausgestellt.

Im Gegensatz zur sonst üblichen Struktur von Swing gibt es für Änderungen an der Liste der JFrames in der Standard-API keinen Listener-Mechanismus. Der EMF-Generator kommt im Prinzip auch ohne derartige Meldungen über neu erstellte oder endgültig zerstörte JFrames aus. Es erwies sich jedoch als praktisch, über eine Tastenkombination direkt aus dem jeweiligen JFrame die EMF-Erzeugung zu starten. Dazu war es nötig, in jedem neu erzeugten JFrame diese Tastenkombination zu registrieren. Ein eigener Überwachungs-Thread wertet daher mehrmals pro Sekunde die Liste der aktiven Frames aus. Für neu erkannte JFrames wird dadurch sehr schnell der Tastatur-Event zur Ecore-Generierung registriert.

4.2.1.2 Erstellen der EMF-Instanz

Das Erzeugen einer Instanz des Swing-Metamodells stellt sich im Prinzip als $1:1$-Abbildung des Swing-Objektgraphen als Ecore-Modell dar. Der Graph wird grundsätzlich in einem preorder-Durchlauf über das component-Attribut der Swing-Componenten durchlaufen, beziehungsweise transformiert. Alle Untertypen von java.awt.Container, und das sind alle Java-Widgets, verfügen über dieses Attribut mit der Kardinalität

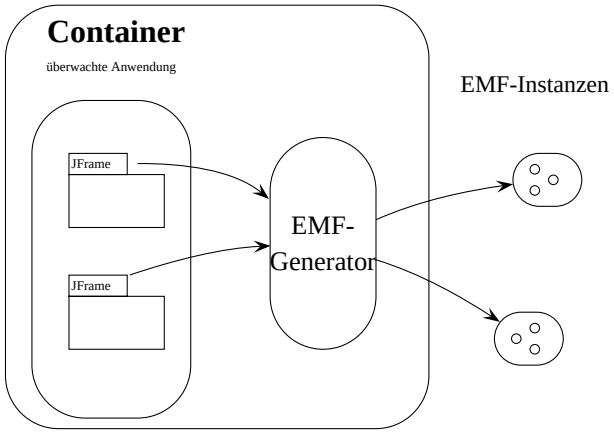

Abbildung 4.2: Zusammenhang zwischen Wrapper und EMF-Generator

$0..*$. Der Werttyp von component ist java.awt.Component, welcher wiederum die Wurzel der Typhierarchie aller AWT und Swing-Widgets ist.

Bei diesem Durchlauf wird, über die Standardmechanismen der dynamischen Modellinstanziierung des EMF, im Speicher die Ecore-Instanz aufgebaut. Zur Weiterbearbeitung dieses Modells ausserhalb der Container-Anwendung wird dieses Modell in einer Datei abgelegt. Als finales Ergebnis der Transformation entsteht daher eine XML-Datei welche die XMI-Serialisierung des Ecore-Modells darstellt.

Die Modellserialisierung, d.h. die Umwandlung der Ecore-Instanz aus dem Speicher in die persistente XMI-Form, erfolgt über die durch die Code-Generatoren erzeugten Methoden zur Modellspeicherung. Diese Vorgehensweise war zwar zunächst praktisch, andererseits wegen der eingeschränkten Verfügbarkeit von Dokumentationen zum verwendeten XMI-„Dialekt“ zum Implementierungszeitpunkt auch kaum anders möglich. Als besonders hinderlich für eine eigene Implementierung erwies sich, dass keine genauen Angaben[2] zur Kreuzreferenzierung von Modellobjekten aufzufinden waren. Der gravierende Nachteil bei der Verwendung des EMF eigenen Serialisierungsmechanismus ist, dass es hierzu unbedingt notwendig ist vorher mit den EMF-Codegeneratoren aus dem Swing-Metamodell den Modell-Quellcode generiert zu haben.

Dieser Zwang stellt sich insbesondere nach Änderungen am Metamodell als hinderlich und zeitintensiv heraus, dies begründet sich folgendermaßen: Das Swing-Metamodell ist auch nach der Reduktion noch ein großes Modell. Dementsprechend entstehen bei der Code-Generierung viele Dateien. Ändert man nun das Modell und startet die Generierung erneut, wird logischerweise jede dieser Dateien erneut generiert. Die bisher vorliegenden Dateien werden jedoch nicht einfach überschrieben, sondern miteinander verschmolzen. Das ist nötig damit manuelle oder sonstige externe Änderungen am „Quellcode“ nicht verloren gehen.

Dieser Verschmelzungsprozess ist zeitaufwändig und noch nicht perfekt, oftmals sind manuelle Nacharbeiten notwendig. Als besonders problematisch, im Sinne von fehlerproduzierend, hat sich das Entfernen von Typen und teilweise auch Attributen aus dem Modell herausgestellt. Zusätzlicher Aufwand ist nach jeder Modelländerung zu betreiben, um nach Abschluss der Code-Erzeugung das Ergebnis wiederum neu zu packagen und es dem EMF-Generator zugänglich zu machen.

Erweiterung des Swing-Metamodells

Natürlich ergaben sich während des Arbeitens mit dem Swing-Modell diverse praktische Hindernisse. Zu nennen sind hier die Widrigkeiten mit dem durch EMF generierten Editor. Dieser erlaubt es zwar, den Baum der UI-Widgets aufzubauen, bot jedoch zunächst keine Unterstützung um diejenigen Eigenschaften welche in eigenen Typen definiert werden anzulegen und zu bearbeiten.

Ein konkretes Beispiel: Die maximale Größe der Swing-Widgets lässt sich über das Attribut maxSize: Dimension festlegen. Der Typ Dimension entstammt dabei den AWT-Packages, und definiert lediglich die beiden ganzzahligen Attribute height und width, deren Werte begrenzen also Breite und Höhe der jeweiligen Swing-Komponente. Die Eigenschaft maxSize:Dimension wurde, gemäß der Festlegungen aus Unterabschnitt 4.1.2, als Assoziation im Typ java.awt.Component angelegt. Der mit den Standardheuristiken erzeugte EMF-Editor ist in der Folge nicht in der Lage diese Eigenschaft anzulegen oder wenigstens zu bearbeiten. Das liegt daran, dass die Instanzen des Dimension-Typs nicht als eigene Knoten in der Baumhierarchie auftreten. Nur diejenigen Attribute des Metamodells die Containment-Beziehungen, d.h. Aggregationen, definieren werden standardmäßig im Editor als mögliche Kanten für den Objektbaum angelegt. Anders formuliert, nur Objekte die über Aggregationsbeziehungen mit dem Wurzelobjekt verbunden sind, werden als eigene Knoten im Editor dargestellt.

[2] im Standard ist die Rede von „XPath-like“ Referenzen

Der Code-Generierungsprozess wird im EMF über ein eigenes Modell gesteuert, das sogenannte Genmodel. Änderungen des Generierungsverhalten sollten zunächst über dieses Steuermodell veranlasst werden. Hier wäre es unter anderem möglich, auch für normale Assoziationen das Anlegen von Knoten im erzeugten Editor zu veranlassen. Dies müsste dann für jedes Attribut, das änderbar sein soll, durchgeführt werden.

Alternativ besteht die Möglichkeit, an entsprechender Stelle der Vererbungshierarchie ein neues Attribut einzuführen, welches seinerseits eine allgemeine Aggregation vom Typ EObject bereitstellt. Es wurde entschieden nach dieser Variante zu verfahren. Der Typ PropertyRegistry, bestehend aus einer $0..*$-Aggregation properties:EObject wurde dem Package javax.swing hinzugefügt. Dieser wird als Attribut propertiesRegistry der Klasse java.awt.Component im Swing-Metamodell integriert. Da alle UI-Widgets Unterklassen der Component-Klasse sind, steht somit für jedes Widget ein eigener Eigenschaftencontainer zur Verfügung. Der besondere Vorteil dieses Ansatzes ist, dass damit auch die Möglichkeit, besteht Instanzen von Eigenschaftenklassen anzulegen die nicht in einem UI-Widget referenziert werden. Somit können beispielsweise Designs vordefiniert abgelegt werden und bei deren Verwendung sind nur noch die entsprechenden Referenzen zu setzen. Durch die separate PropertyRegistry werden dem Modell keine neuen Ausdrucksmöglichkeiten oder ähnliches hinzugefügt, es handelt sich ausschließlich um eine praktikable Arbeitserleichterung.

Die Beschreibung weiterer Details des technischen Ablaufs sind im Anhang im Abschnitt C.4 dargestellt. Dessen letztliches Ergebnis zeigt Abbildung 4.3. Hierin wird eine beispielhafte Java-Applikation sowie deren resultierendes Ecore-Modell dargestellt. Es handelt sich um einen einfachen, vollfunktionsfähigen Taschenrechner welcher als Demonstrationsapplikation implementiert wurde. Dieser wurde gemäß der obigen Beschreibung über den Wrapper gestartet.

4.2.2 Modellinstanzen für XUL

Die Technik des Einhängens in den Swing-Objektgraphen ist nicht auf die Verwendung des Swing-Metamodells beschränkt. Zur Erzeugung anderer Modellinstanzen ist es lediglich notwendig den Transformationsalgorithmus auszutauschen. Ein weiterer Transformator wurde für das XUL-Metamodell implementiert. Die vorbereitenden Arbeiten zu diesem Einsatz, insbesondere die Injektion in die Legacy-Anwendung, erfolgen analog dem Vorgehen beim Swing-Generator. Genau genommen ist es möglich beide Generatoren gleichzeitig im Container aktiv zu haben. Die Abbildung 4.4 stellt die um die XUL-Generierung erweiterte Infrastruktur aus Abbildung 4.2 dar.

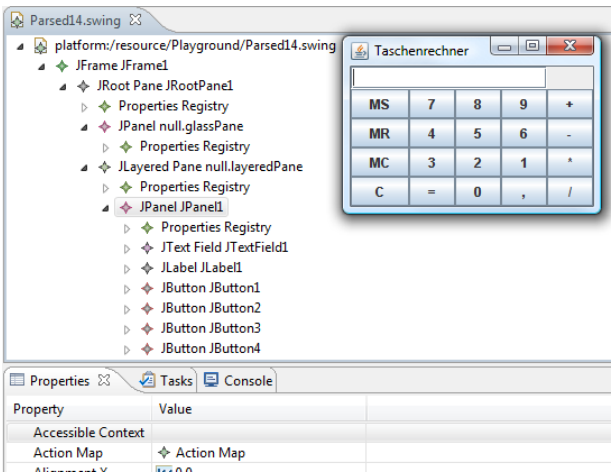

Abbildung 4.3: Taschenrechner-Anwendung und geparste Modellinstanz im Standardeditor

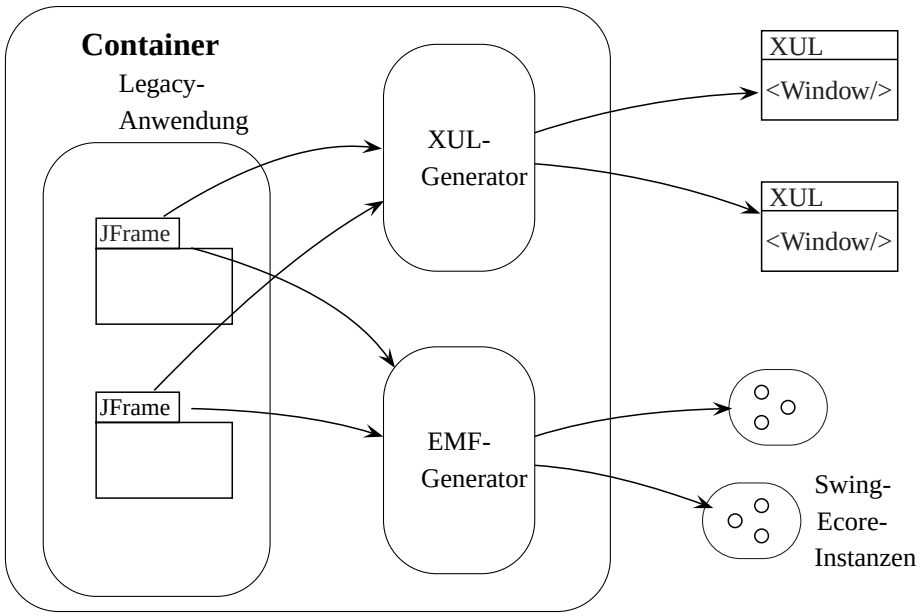

Abbildung 4.4: Container-Infrastruktur mit Swing-Ecore und XUL-Generator

4.2.2.1 Der XUL-Generator

Der prinzipielle Ablauf der XUL-Erzeugung unterscheidet sich nicht von der zuvor beschriebenen Swing-Transformation. Es handelt sich wiederum um einen preorder-Graphendurchlauf. Ebenfalls wird für jeden Knoten das Äquivalent aus dem XUL-Metamodell ermittelt, instanziiert und im Ergebnisbaum angelegt.

Selbstverständlich waren auch für die XUL-Erzeugung eine Reihe praktischer Probleme zu überwinden. Zunächst zu den Eigenschaften. Wie bereits in Punkt 2.1.4.1 ausgeführt, erfolgt die Festlegung der grafischen Eigenschaften der Oberflächenelemente für XUL mittels CSS. Es gibt also im Metamodell keine Attribute zur Festlegung von Farben oder der zu verwendenden Schriftart. Nur das Attribut **style** existiert, in welchem die Definition dieser Eigenschaften in der CSS-Syntax erfolgt.

Es war also ein CSS-Exporter zu schreiben, dessen Zweck darin besteht, die Eigenschaften eines Swing-Objektes in eine CSS-Beschreibung zu übersetzen. Die vorliegende Implementierung des Exporters ist nicht vollständig, sondern eher als minimal zu beschreiben. Es wurden lediglich CSS-Schlüssel für die oben bereits erwähnten Farben und Schriftarten umgesetzt. Die Begründung für diese Einschränkung ist, dass es im Rahmen dieser Arbeit nicht um eine vollständige Umsetzung des CSS-Standards geht, sondern darum, das Konzept zu evaluieren.

Eine gewisse Vereinfachung gegenüber dem Swing-Modell entstand dadurch, dass im XUL-Metamodell keine Referenztypen für die Eigenschaften verwendet werden. Ein eigener Objekt-Cache für die Transformation war demzufolge nicht von Nöten.

Die Hauptschwierigkeit bei der XUL-Generierung lag jedoch nicht in der Eigenschaftenübertragung oder dem Knotenmapping. Als aufwändig stellte sich die Übertragung des Layouts der Oberfläche heraus.

Layouterzeugung

XUL-Oberflächenbeschreibungen erfolgen normalerweise ohne Angabe fixer Größen und Koordinaten. Im Metamodell sind zwar Attribute für diese Merkmale vorgesehen, im Hinblick auf den Verwendungszweck einer XUL-Definition sollten diese jedoch nicht verwendet werden. Stattdessen wird dem jeweiligen Renderer die Platzierung und die Festlegung der effektiven Größe der UI-Widgets überlassen. Dem Renderer können jedoch über diverse Sprachkonstrukte Richtlinien für das Layout vorgegeben werden.

Zu diesen Sprachkonstrukten zählen Containertypen wie beispielsweise **vbox** und **hbox**. XUL-Widgets, die direkte Kindknoten unterhalb einer **vbox** sind, werden in Knotenreihenfolge vertikal angeordnet. Für eine **hbox** gilt sinngemäß das Gleiche, es handelt sich jedoch um eine waagerechte Anordnung. Container können verschachtelt werden, eine **vbox** kann also etwa eine **hbox** enthalten. Durch die Wahl einer geeignete Verschachtelung kann die Grobstruktur der Oberfläche festgelegt werden.

Die Anordnung und die Größenverteilung innerhalb eines Containers kann ebenfalls relativ fein definiert werden. Durch Zuweisung einer Gewichtung an die Kindelemente wird der verfügbare Oberflächen-Raum anteilig verteilt. Ein Beispiel: Eine **hbox** habe drei Kindelemente E_0, E_1 und E_2. Die Gewichtung $g(e)$ sein folgendermaßen belegt: $g(E_0) = 0.2$, $g(E_1) = 0.3$ und $g(E_2) = 0.05$. Der Renderer ermittelt nun, dass der Container, in horizontale Richtung, 200 Pixel breit sein wird.

Damit ergibt sich die Breite eines Elementes folgendermaßen: $breite(e) = 200px * \frac{g(e)}{\sum_{i=0}^{2} g(E_i)}$ Das Element E_2 wird in diesem Fall mit einer Breite von $200 * \frac{0.05}{0.2+0.3+0.05} = 18,\overline{18} \approx 18$ Pixeln dargestellt.

Durch Einfügen von gewichteten Zwischenräumen kann die Positionierung, bei fixer Gesamtgröße, nahezu pixelgenau erfolgen. Die Genauigkeit ist dabei einerseits abhängig von eventuellen Rundungsfehlern, aber hauptsächlich davon, wie der jeweilige Renderer den den Untercontainern zur Verfügung bereitgestellten Platz berechnet. Beispielsweise kommt es vor, dass einige Pixel für einen imaginären Rand reserviert werden. Eine tatsächlich pixelgenaue Positionierung über die Container und Gewichtung ist daher immer nur für einen bestimmten Renderer möglich; mit anderen Worten, es ist nicht sinnvoll das anzustreben.

Die Umkehrung des Prinzips der Anteilsverteilung liefert einen Algorithmus um Koordinatenangaben in Gewichte umzurechnen. Es müssen „nur“ die Container geeignet gefunden werden und von deren Inhalt die Position sowie die Größe in eine Gewichtung umgerechnet werden.

Die Identifikation der Container kann automatisch erfolgen. Die Grundidee ist, die dargestellten UI-Widgets nach Überlappung in genau einer Ebene, waagerecht oder senkrecht, zu gruppieren. Jede so entstehende Gruppe wird zu einem Container. Anschliessend werden die Container, die mehrere Widgets enthalten, rekursiv weiter in der jeweils alternierenden Dimension zerlegt. Die Rekursion endet, sobald sich jedes UI-Widget einzeln innerhalb eines eigenen Containers befindet.

Für die Gewichte biete sich eine triviale Herangehensweise an. Es besteht keinerlei Zwang das Gewicht als Prozentsatz oder sonstige rationale Zahl anzugeben. Die Gewichtsverhältnisse, die sich hier für die Oberflächen ergeben, können in jedem Fall durch natürliche Zahlen definiert werden. Die Ausgangsgröße für die Verteilung der Gewichte ist die Größe des untersuchten JFrame, also ein Wert $(hoehe, breite)$ wobei ebenfalls $hoehe, breite \in \mathbb{N}$. Die Idee welche sich aufdrängt ist daher, für den Gewichtswert jeweils die Ausdehnung in Pixeln zu setzen. Selbstverständlich als dimensionslose Zahl. Dadurch entfällt die Notwendigkeit zur Berechnung von Verhältniszahlen komplett.

Der Algorithmus 2 beschreibt die Durchführung des Layoutings unter Benutzung von Pseudocode.

Die Abbildung 4.5 stellt die Anwendung dieses Algorithmus für ein kleineres Beispiel dar. Erstellt und dargestellt wurde die Beispieloberfläche mit dem XUL-Editor.

Algorithmus 2 Layout-Bildung mit relativen Koordinatenangaben

1. Setze $Containerausdehnung$ = Gesamtframe, $Vaterknoten$ = **window**, $Durchlaufrichtung$ = vertikal
2. Bilde Cluster sich überlappender Elemente innerhalb $Containerausdehnung$ in Durchlaufrichtung
3. Für jeden Cluster, in Abhängigkeit von der $Durchlaufrichtung$

   vertikal: für Cluster einen **hbox**-Knoten anlegen, als dessen Gewicht wird die überdeckte `Höhe` in Pixeln gesetzt

   horizontal: für Cluster einen **vbox**-Knoten anlegen, als dessen Gewicht wird die überdeckte `Breite` in Pixeln gesetzt Pixeln gesetzt
4. Erzeugten Cluster-Knoten als Kind unterhalb des $Vaterknoten$ anlegen
5. Cluster-Zwischenräume als **spacer**-Knoten, wiederum mit der Ausdehnungsgewichtung in Pixeln, zwischen den Containern unterhalb des Vaterknoten anlegen
6. Für jeden Cluster-Knoten:
   (a) Falls mehr als 1 Widget im Cluster, oder das Einzel-Widget nimmt nicht den gesamten Clusterbereich ein: Alterniere $Durchlaufrichtung$, $Vaterknoten$=Cluster-Knoten, $Containerausdehnung$ = Koordinatenbereich des Clusters; weiter bei 2.
   (b) Sonst, Widgetübertragung von Swing→XUL
7. Ausgabe der Ergebnisstruktur

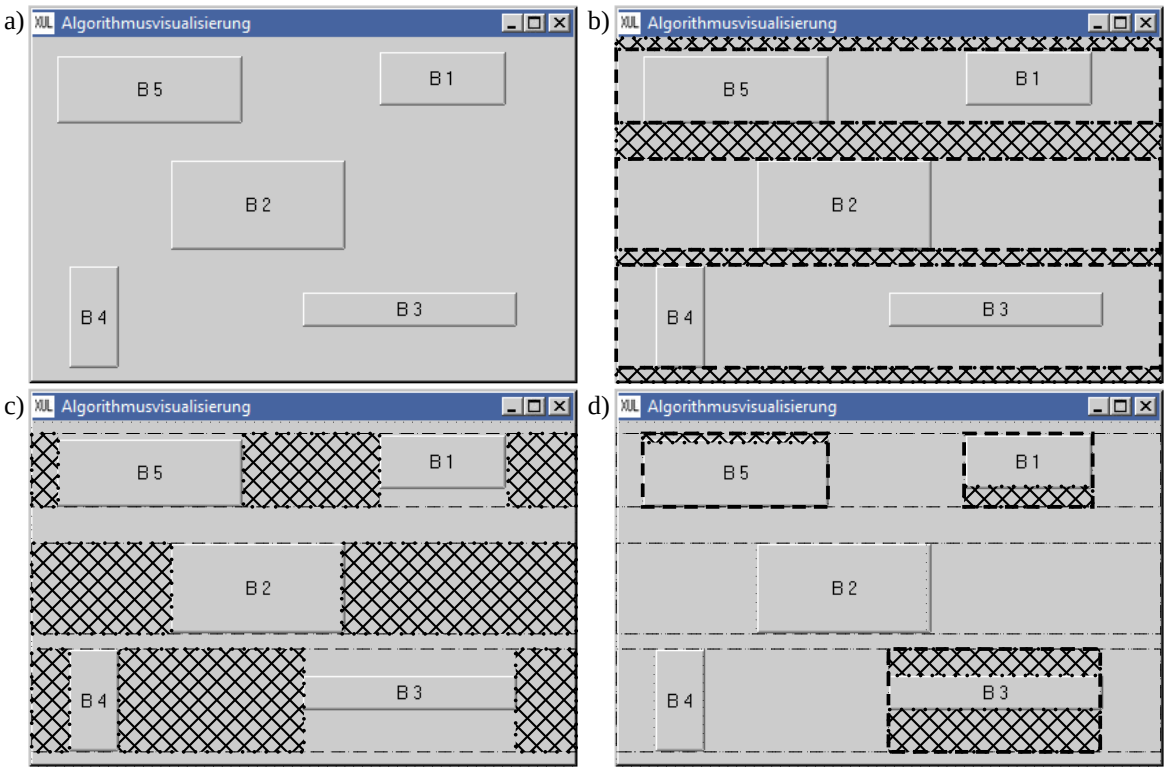

Abbildung 4.5: Visualisierung des Layoutalgorithmus für XUL

Alle vier Teilabbildungen in Abbildung 4.5 zeigen das gleiche Nutzerinterface, jedoch mit verschiedenen Overlays. In Abbildung 4.5a) ist als Ausgangssituation eine schematische Nutzeroberfläche mit fünf Widgets dargestellt. Die Schaltflächen B_1 bis B_5 seien in diesem Fall die Repräsentanten beliebiger Widgets.

In Abbildung 4.5b)-d) wird Schritt für Schritt die Zerlegung in Containern gezeigt. Schraffiert dargestellt wurden erkannte leere Zwischenräume. Neu erkannte Cluster und damit die anzulegenden Container wurden dick-gestrichelt dargestellt.

Die Ausgangszerlegung des Gesamtframes findet sich in Abbildung 4.5b). Durch vertikale Aufspaltung wurden drei Cluster erkannt.

1. die Vertikalausdehnungen der Widgets B_5 und B_1 überlappen sich
2. das Höhenintervall des Widgets B_3 liegt innerhalb dessen von B_4
3. die Y-Koordinaten von Widget B_2 überschneiden sich mit keinem anderen Widget, es handelt sich um einen 1-elementigen Cluster

Zudem wurden oberhalb, unterhalb sowie zwischen den Clustern insgesamt vier Zwischenräume erkannt. Diese werden einfach als **spacer** angelegt, der dadurch umschlossene Raum auf der Oberfläche wird nicht weiter betrachtet. Das Widget B_2 füllt seinen Container nicht vollständig aus, daher wird auch dieser im nächsten Schritt rekursiv weiter zerlegt. Die Aufspaltungsrichtung war vertikal, also wird für jeden Cluster eine **hbox** angelegt.

Die Zerlegung dieser drei **hbox**-Gruppen wird in Abbildung 4.5c) dargestellt.

- $\forall B_n$ ist bereits nach dieser Aufspaltung die Breite festgelegt
- für B_2 und B_4 sind deren Position und Grösse so genau wie mit diesem Ansatz möglich definiert $\Rightarrow$ Abbruch der Layout-Rekursion und Transformation des Widgets in das korrespondierende XUL-Element, hier **button**
- eine weitere Clustering-Rekursion muss für die Container um B_1, B_3 und B_5 durchgeführt werden

Die Abbildung 4.5d) zeigt diesen letzten Rekursionsschritt. Da in jeder Rekursionsstufe die Aufspaltungsrichtung alterniert, wird in diesen Containern wiederum vertikal gruppiert. Über die dabei ermittelten Zwischenräume sind die Lage und Größe aller Widgets relativ beschrieben. Der Quellcode-Ausschnitt in Listing B.5 zeigt das vorläufige Endergebnis der Layoutbemühungen.

Die Zerlegung wurde in diesem in Fall in vertikaler Richtung begonnen. Strukturell das selbe Ergebnis würde man erhalten, wenn die Clusterung in horizontaler Richtung begonnen hätte. In der Beispieloberfläche ist initial keine horizontale Gruppierung möglich, weil sich die Koordinatenbereiche in X-Richtung aller Widgets überlappen. Es ergäbe sich daher eine umschliessende hbox, innerhalb welcher dann mit der vertikalen Aufsplittung begonnen wird. Generell gilt, dass, solange die Widgets nicht übereinander liegen, immer entweder eine vertikale oder eine horizontale Clusterung möglich ist. Die Entscheidung ob in waagerechter oder senkrechter Richtung mit dem Algorithmus begonnen wird ist unerheblich für das erzeugte Endergebnis.

Zu den Grenzen dieses Algorithmus: Es ist in dieser einfachen Version nicht möglich Nutzeroberflächen nach XUL zu überführen, in denen Widgets verwendet werden, die sich überlagern. Diese Beschränkung ergibt sich aus der Art der Clusterung. Elemente welche übereinander liegen können weder in horizontaler noch in vertikaler Richtung auseinander gruppiert werden. Daher würde bei Anwendung des Verfahrens eine Endlosrekursion auftreten und kein Ergebnis erzeugt werden. In der Implementierung für den

Swing→XUL Konverter werden derartige Fälle zwar erkannt und abgefangen, eine Lösung wird jedoch nicht erzeugt.

Die Sprachmittel von XUL erlauben übereinander liegende Widgets. Man kann sich dafür unter anderem des integrierten Layer-Konzeptes bedienen, Widget-Stacks benutzen, oder aber die Positionierungsangaben über CSS-Angaben vornehmen. Gleichwohl sind gestapelte Widgets eher ein Sonderfall bei Nutzeroberflächen. Daher erscheint das Weglassen einer geeigneten Layouter-Implementierung für diese Situationen keine gravierende Einschränkung zu sein.

Die Abbildung 4.6 zeigt das Ergebnis der Transformation der Oberfläche der Taschenrechner-Beispielanwendung nach XUL. Auf der linken Seite findet sich ein Screenshot des Ecore-Standardeditors, rechts oben das Rendering-Ergebnis von SeaMonkey und in der unteren rechten Hälfte die gleiche XUL-Datei dargestellt im XUL-Editor. Im direkten Vergleich der beiden Renderer ist zu sehen, dass das obere Textfeld im SeaMonkey-Browser breiter dargestellt wird als im XUL-Editor und auch in der ursprünglichen Java-Anwendung. Das liegt daran, dass die Gecko-Engine das Label welches sich rechts vom Textfeld befindet nur dann ins Layout einrechnet wenn es eine nicht-leere Zeichenkette darstellt. Da im Moment des Snapshots in diesem Label nur ein Leerzeichen als Wert vorlag, wird es nicht dargestellt und sogar auch dessen Gewichtung ignoriert. Damit erhöht sich der relative Anteil an der Breite für das Gewicht des Textfeldes, und es wird etwas breiter. Unabhängig davon, welcher der Renderer in diesem Fall falsch rechnet, zeigt es einmal mehr die Abhängigkeit der konkreten Darstellung XULs vom jeweiligen Interpreter.

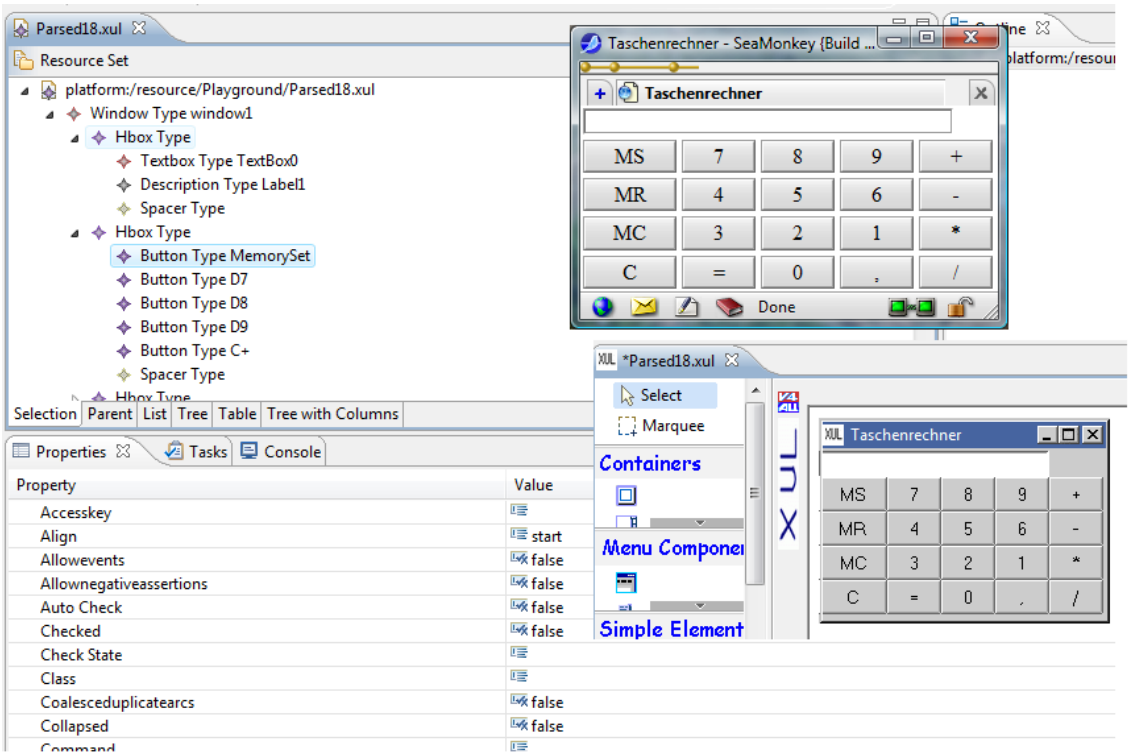

Abbildung 4.6: Drei Renderer des XUL-Snapshots der Taschenrechneranwendung

4.2.3 Ableitung des abstrakten Oberflächenmodells

Der Prozess des Reverse Engineering ist auf der Ebene des konkreten Oberflächenmodells noch nicht abgeschlossen. Wie bereits in Abschnitt 2.1 dargelegt, ist das CUI Bestandteil eines, idealerweise bidirektionalen, Gesamtprozesses. In diesem Prozess ist die dem CUI vorgelagerte Abstraktionsebene die des Abstrakte Oberflächenmodells (AUI).

Typischerweise entsteht die CUI-Instanz als Ergebnis einer Model-To-Model Transformation aus einer AUI-Instanz. Aber auch die Umkehrung dieser Transformation ist denkbar und wird angewendet. So ist beispielsweise in Paternòs MARIA-XML [PSS09] diese Transformationsrichtung Bestandteil der „Migratory Interface“, d.h. der Übergabe von Benutzungsoberflächen zwischen Geräten.

Bei dem MARIA-XML Use Case handelt es sich nicht um Reverse Engineering, das Modell der abstrakten Oberfläche ist vordefiniert und wird nicht aus der vorhanden Instanz des CUI generiert. Die prinzipielle Möglichkeit der Generierung einer AUI-Instanz aus einer CUI-Instanz untersuchte Rathsack [Rat06]. Die von ihm erarbeitete Transformation generierte aus XUL-Oberflächenbeschreibungen, ohne Verwendung des XUL-Metamodell aus Punkt 2.1.4.2, Instanzen des X-AIM AUI-Metamodells. Zwar erfolgte seine damalige Implementierung ohne Einsatz von MDSE-Techniken, es handelt sich um ein manuell erstelltes Java-Programm, die Machbarkeit konnte dennoch gezeigt werden. Die Erstellung ähnlich mächtiger Model-To-Model Transformationen ist zumindest für den Rostocker MD-UID Ansatz bisher noch nicht erfolgt.

Überdenkt man die Mechanismus der Ableitung der AUI aus einem CUI-Modell, so gelangt man schnell in das Themengebiet der Pattern-Erkennung, bzw. Pattern-Matchings. Die Identifikation der in der untersuchten Oberfläche eingesetzten Patterns bietet eine nützliche Zusatzinformation, insbesondere auch im Hinblick auf weiteren Bearbeitungsschritte. Möglicherweise finden sich Wege des maschinellen Abgleichs mit Patternkatalogen wie dem in Abschnitt 3.5 erstellten. Dabei handelt es sich jedoch um ein nicht triviales Problem, welches in weiteren Arbeiten, z.B. unter Einsatz von QVT Relations, weiter beleuchtet werden sollte.

4.3 Forward Engineering

Reengineering wird nicht als Selbstzweck betrieben. Das Ziel des hier vorgestellten Reengineering ist es, bereits existierende Software mit den Mitteln der modellgetriebenen Softwareentwicklung weiter zu Pflegen und zu Warten. Dieses Ziel ist weit gesteckt und seine Erreichung steht in der nächsten Zeit nicht unbedingt zu erwarten. Dennoch lohnt es sich, auf der Basis des heute Bekannten und Möglichen, Überlegungen zum Forward Engineering, d.h. der Rückrichtung des Reverse Engineering anzustellen.

Grundsätzlich unterscheidet sich das Forward Engineering eines durch Reverse Engineering gewonnenen Oberflächenmodells nicht von der Bearbeitung einer nativ modellgetrieben entwickelten Benutzungsoberfläche.

Mit den bisherigen Mitteln ist es möglich, aus einer Instanz des Swing-Ecore Modells wiederum Java Swing-Quellcode zu erstellen. Je ausgefeilter die an dieser Stelle eingesetzte Model-To-Text Transformation ausgestaltet wird, desto genauer wird das Ergebnis der Quelle entsprechen. Gleichfalls existieren M2M-Transformationen, bidirektional, zwischen XUL und Java Swing CUI-Modellinstanzen. Für diese Algorithmen gilt gleichfalls, dass sie wohl nie vollends fertiggestellt werden können, aber bei genügendem Aufwand ein hoher Reifegrad errreicht werden kann.

Eine Anwendungsmöglichkeit für die Überführung von Swing-Oberflächen in Instanzen des XUL-Meta-

modells ist es, diese UIs grafisch mit dem XUL Editor XULE zu bearbeiten, sie anschließend wieder nach Java Swing zu transformieren und dabei wiederum die Anbindung an die bisherige Anwendungslogik herzustellen. Dabei entsteht im Idealfall eine funktional identische Software, die sich lediglich durch ihre Benutzungsoberfläche von der ursprünglichen Anwendung unterscheidet. Zwar ist ein solches Vorgehen bei der Reife der aktuellen Werkzeuge und Transformationen nur bedingt vorteilhaft, gleichwohl gelang der Nachweis der prinzipiellen Machbarkeit dieses in [WF07] vorgestellten Ansatzes.

Die per Reverse Engineering gewonnenen Modellinstanzen weisen keine besonderen Charakteristika auf, die den Einsatz der bereits diskutierten Patternkomponenten beeinflussen. Für das Forward Engineering kann daher auf die im Patternkatalog definierten oder referenzierten Patternkomponenten zurückgegriffen werden. Somit können auf diesem Weg, im Idealfall, Legacy-UIs auf die Verwendung moderner HCI-Patterns umgestellt werden.

4.4 Herausforderungen und Grenzen des vorgestellten Reengineering-Ansatz

In diesem Kapitel wurden Methoden und Techniken vorgestellt die es ermöglichen können, die Vorteile der modellgetriebenen Softwareentwicklung auch für Software nutzbar zu machen die nicht mit diesem Paradigma entwickelt wurden. Dabei wurde der Aspekt der Benutzungsoberflächen in den Vordergrund gestellt und anhand dieses Problems gezeigt wie Modelle zur Laufzeit aus Legacy-Software gewonnen werden können.

In den Erläuterungen zu diesem Prozedere wurden die Bedingungen dargestellt, welche zu dessen Anwendbarkeit notwendig sind. Treffen diese Voraussetzungen nicht zu, ist dies nicht gleichbedeutend damit, dass prinzipiell kein Reverse Engineering möglich wäre. Es sollten dann jedoch andere Verfahren gewählt bzw. entwickelt werden.

Auch das präsentierte Vorgehen bietet noch eine Vielzahl Herausforderungen. Bisher wurden nur Betrachtungen zu den unteren CAMELEON-Ebenen angestellt, AUI und CUI. Allerdings sind im Rostocker MDUID Prozess ebenfalls noch die höheren Abstraktionsebenen Dialoggraph und Aufgabenmodell zu betrachten.

Die in Punkt 4.2.1.2 vorgestellte Snapshot-Technik erstellt zunächst ein statisches Abbild des aktuellen Zustands des Objektgraphen der Benutzungsoberfläche. Dynamische Aspekte, wie etwa die Navigation durch die Anwendung und Anpassungen von Teilen der Oberfläche als Konsequenz von Aktionen, bleiben im Prinzip unberücksichtigt. Implizite Informationen dazu liegen jedoch durchaus vor, da der Zugriff auf den Objektbaum auch den Zugriff auf alle per Anwendungslogik definierten Aktionen einschließt. Eine semantische Auswertung des zugrundeliegenden Quellcodes könnte, zumindest für einige einfache Sonderfälle, die gesuchten Navigationsinformationen liefern. Der dazu notwendige Aufwand ist ohne gründliche Überlegungen kaum abschätzbar. Dennoch scheint für ein tatsächlich praktisch anzuwendendes Werkzeug die zumindest annähernd korrekte Ermittlung der sichtinternen Navigation und damit auch des Kontrollfluss nahezu unverzichtbar.

Zu den mit dem jetzigen Stand der Werkzeuge schon denkbaren praktischen Einsatzzwecken zählt das automatische Testen von Benutzungsoberflächen. Momentane UI-Testverfahren automatisieren Tests durch Aufzeichnungen von Beispielabläufen. In einem Durchlauf werden die Interaktionsevents, wie Mausklicks an Stelle (x,y) oder Tastendrücke, aufgezeichnet. Bei der eigentlichen Testdurchführung wird der Musterablauf gestartet und Abweichungen gegenüber dem als richtig bekannten Ergebnis gemeldet. Diese Verfahren sind verhältnismäßig empfindlich gegenüber Layout- und Designänderungen der Benutzungsoberfläche. Es

ist z.B. ein reales Problem, dass sich nach Änderung an der Corporate Identity einer Firma, auf Grund von Schriftarten- und Logo-Änderungen, die Position von UI-Elementen ändert. Die Anwendung der oben skizzierten Testverfahren kann es daher durchaus notwendig werden, die Musterabläufe neu aufzuzeichnen.

Das in diesem Kapitel vorgestellte Verfahren der Nutzung der UI-Objektstruktur, kann stattdessen den Testgegenstand über Objektidentitäten identifizieren und ist nicht auf pixelgenaue Positionsangaben angewiesen. Er ist daher sehr robust gegenüber Layoutveränderungen auf Grund von Grössen-, Schriftarten- und sonstigen dekorativen Änderungen. Vielversprechende Möglichkeiten bieten sich auch bei der Definition der Testbedingungen, denn diese können auf der Basis des Metamodells z.B. mit OCL [OCL10] sehr präzise formuliert werden.

An seine Grenzen trifft der vorgestellte Ansatz bei verteilten Anwendungen, da mit dem Wrapper-Mechanismus immer nur der Teil der Software untersucht werden kann welcher auch lokal vorhanden ist. Ebenfalls scheint es kaum erfolgversprechend auf diese Weise UIs zu untersuchen die permanenten dynamischen Änderungen unterliegen. Beispielsweise grafisch anspruchsvolle Spiele oder andere in irgendeiner Weise gezeichnete Oberflächen.

Neben der bereits erwähnten Dialogmodell-Problematik, stellt das Aufgabenmodell eine kaum zu überwindende Grenze dar. Die Konkretisierung und Formalisierung von Nutzerzielen ist ein inhärent manueller Prozess. Im Fall der Bottom-Up Ableitung, d.h. abstrahierend beginnend auf der CUI-Ebene, ist zur Zeit kaum denkbar, wie aus einem Screen- bzw. Snapshot einer Benutzungsoberfläche auf das aktuelle oder globale Nutzerziel geschlossen werden kann. Ein solches Vorgehen bedarf mit hoher Wahrscheinlichkeit einer Vielzahl manueller Eingriffe. Im Augenblick laufen Arbeiten [GB10], die die Möglichkeiten zur Ableitung von Aufgaben- oder Prozessmodellen aus Nutzeraktionen untersuchen. Vielleicht lassen sich diese Ergebnisse in Zukunft einmal für das hier dargestellte Reverse Engineering anwenden.

Kapitel 5

Zusammenfassung

5.1 Überblick

Der Inhalt dieser Dissertationsschrift ist eine Untersuchung zu Möglichkeiten der Erstellung grafischer Benutzungsoberflächen mit Methoden der modellgetriebenen Softwareentwicklung. Als Ergebnis wurde herausgearbeitet, dass diese Technik für WIMP-Oberflächen anwendbar ist und in diesem Kontext nutzbringende Vorteile bieten kann.

Diese Vorteile ergeben sich insbesondere aus der modellinhärenten Abstraktion. Durch spezialisierte und strukturierte Abstraktionsebenen gelingt die Aufgliederung, auch komplexer, Benutzungsoberflächen in handhabbare Teilmodelle. Pflege, Wartung und Weiterentwicklung einer Anwendung erfolgen zentral in Modellinstanzen, langwierige Such- und Bearbeitungsprozesse direkt im Quellcode werden im Prinzip obsolet, bleiben aber dennoch möglich.

Weil die Oberflächenbeschreibungen in der MD-UID jeweils einer bekannten formalisierten Beschreibung entsprechen können standardisierte Automatismen zu deren Bearbeitung entwickelt werden. In der Arbeit wurde weiter untersucht inwieweit HCI-Patterns in diesem Kontext als Modelltransformationsautomatismen umgesetzt werden könnten. Unter anderem erwies sich dabei eine Klassifikation und modellbasierte Beschreibung von HCI-Patterns als notwendig. Ein Vorschlag hierfür wurde entwickelt und damit das Konzept eines Katalogs von Patterntransformationen begonnen. Dessen initialer Inhalt wurde, soweit möglich, modellgetrieben aus öffentlich verfügbaren HCI-Patternsammlungen generiert.

Ein weiteres Kapitel der Arbeit ist der Untersuchung von Techniken zum modellbasierten Reengineering gewidmet. Das Ziel war es, einen Weg zu finden, Modelle bzw. Modellinstanzen aus Software zu generieren oder abzuleiten welche ursprünglich nicht modellbasiert entwickelt wurde. Die Motivation zu den ausgearbeiteten Ansätzen ergab sich aus zwei praktischen Problemen. Erstens war es nicht anders möglich an ein vollständiges Klassenmodell des Java-Swing Frameworks zu gelangen. Zweitens ist die Deklaration einer komplexen Benutzungsoberfläche, auch bei Einsatz der MD-UID, eine durchaus zeitaufwändige Angelegenheit. Ein Verfahren das es ermöglicht, bereits ausprogrammierte Oberflächen als Quelle für Modellinstanzen zu nutzen, erhöht daher die Produktivität in bestimmten Einsatzfällen ganz erheblich.

Eng verbunden ist die Arbeit mit dem an der Universität Rostock verfolgten modellbasierten Gesamtprozess zur Softwareentwicklung. Aus diesem Grund wurde dieser Prozess, sowie eine Reihe weiterer in der Forschung diskutierter Ansätze, vorgestellt.

Wie bei den allermeisten Forschungsarbeiten, fand sich bei der Bearbeitung der jeweiligen Problemstel-

lungen eine Vielzahl von Anknüpfungspunkten für weiterführende Arbeiten. Einige wurde bereits in den vorstehenden Kapiteln skizziert, eine Zusammenfassung dieser Ideen und zusätzliche Gedanken werden in Abschnitt 5.2 diskutiert.

Im Einleitungskapitel wurden acht Fragen aufgestellt, die eine Einordnung von Motivation und Kontext dieser Arbeit bieten. Die vorstehenden Kapitel bearbeiteten diese Forschungsfragen im Detail. Hier soll nun deren summarische Zusammenfassung und Beantwortung erfolgen.

1. Lassen sich umfassende Metamodelle moderner grafischer Benutzungsoberflächen erstellen?

Die Antwort auf diese Frage fällt positiv aus, beispielsweise wurde für die Oberflächen-Deklarationssprache XUL ein Metamodell aufgestellt. XUL zählt durchaus zu den modernen Beschreibungssprachen für grafischen Benutzungsoberflächen. Für die Erzeugung dieses Metamodells konnten Techniken der modellgetriebenen Softwareentwicklung genutzt werden, es wurde über manuell nachbearbeitete Modelltransformationen erzeugt.

Auch das Java GUI-Framework Swing ist ein übliches modernes UI-System. Durch Kombination mehrerer Verfahren konnte auch für Swing ein Metamodell aufgestellt werden.

2. Wie kann die inhärente Komplexität solcher Metamodelle handhabbar gemacht werden?

Bei dieser Frage ist zu nächst zu validieren, ob die erhaltenen Metamodelle tatsächlich komplex sind. Tatsache ist das sie tendenziell eine hohe Anzahl an Metaklassen, mehrere Dutzend bis einige Hundert, aufweisen. Genauere Zahlen finden sich bei den Modellerläuterungen. Weiterhin existiert in den Modellen eine Vielzahl von Attributen, Assoziationen und Vererbungsbeziehungen.

Da an dieser Stelle kein Maß für Modellkomplexität herangezogen oder gar erarbeitet werden soll bleibt nur der Rückzug auf einen praktischen Standpunkt. Zum gegenwärtigen Zeitpunkt bringt die Größe des unreduzierten Swing-Modells die Werkzeuge des Eclipse Modeling Projects an den Rand der Praktikabilität.

Bei der Suche nach Wegen die Dimensionen des Metamodells zu verkleinern fand sich im Ansatz des Model-Prunings eine gangbare Lösung. Rein quantitativ betrachtet gelang es, mit diesem Verfahren das Swing-Modell auf etwa ein zwanzigstel seiner Größe zu reduzieren ohne die Ausdrucksmächtigkeit entscheidend zu verringern.

Eine alternative Vorgehensweise zur Komplexitätsreduktion ist die Einführung zusätzlicher Abstraktionsebenen. Wie bei der Diskussion der laufenden und historischen Forschung herausgearbeitet wurde, ist dies eine bei der MD-UID etablierte und häufig eingesetzte Technik. Auch in dieser Arbeit wurde davon Gebrauch gemacht.

3. Inwieweit kann nützliche Werkzeugunterstüzung gestellt werden?

Die Nützlichkeit eines Werkzeuges für die MD-UID ist kaum objektiv bewertbar. Zunächst ist die Frage zu stellen, ob überhaupt Tool-Support nötig und in zweiter Stufe, auch möglich ist. Prinzipiell kann die Bearbeitung der meisten Modellinstanzen mit jedem beliebigen Texteditor erfolgen. Die zweite Forschungsfrage impliziert jedoch das die Modelle im Allgemeinen sehr umfangreich sind. Übliche generische Texteditoren werden daher eher nicht das Mittel der Wahl für die alltägliche Modellierung sein. Nützlich scheinen eher integrierter Entwicklungsumgebungen, wie sie auch bei der klassischen Softwareentwicklung zum Einsatz kommen. Aktuelle Untersuchungen darüber, was geeignete Werkzeugunterstützung leisten können sollen, führt beispielsweise Opoka [CDSR09] mit der Zielsetzung der Gestaltung einer Entwicklungsumgebung für OCL.

Für den Rostocker modellgetriebenen Prozess wurde eine Vielzahl von, zumeist grafischen, Entwicklungswerkzeugen geschaffen. Diese unterstützen Modellierer und Designer bei der Bearbeitung der wichtigsten Teilmodelle. In dieser Arbeit wurde mit dem grafischen Editor XULE in Punkt 2.1.4.3 ein Werkzeug zur manuellen Bearbeitung konkreter Benutzungsoberflächen erstellt. XULE ermöglicht die Bearbeitung aller Modellparameter, liefert soweit möglich sofortiges visuelles Feedback und leitet und unterstützt den Benutzer indem nur gültige Modellierungen zugelassen werden. Für andere Teilmodelle und Transformationen des Gesamtprozesses ist die Werkzeugunterstützung weniger weit entwickelt.

4. Welche Möglichkeiten zur Integration von HCI-Patterns sind denkbar?

Es lassen sich alle diejenigen HCI-Patterns in den MD-UID Prozess integrieren, deren Pattern-Lösung als Modelltransformation oder Modellinstanz ausgedrückt werden können. Dies trifft aus mehreren Gründen auf viele HCI-Patterns nicht zu. In der Arbeit wurde ein existierendes Metamodell für Patternkataloge erweitert, sowie ein ergänzendes Metamodell ausgearbeitet, das zur formalisierten Speicherung möglicher Patternlösungen geeignet ist. Die Metamodelle für diesen Katalog wurde im Hinblick auf Integrierbarkeit in den vorgestellten Prozess konstruiert. Ein eigens erstellter Patternkatalog, als Instanz dieser Metamodelle, wurde aus öffentlich verfügbaren Quellen befüllt.

5. Welche Patterns sind soweit formalisierbar, dass sie direkt in einem Gesamtprozess anwendbar sind?

Wie vermutet lassen sich nur wenige HCI-Patterns soweit formalisieren das ihre Lösung direkt auf Modellinstanzen anwendbar ist. Eine zahlenmäßige Aufstellung einer manuellen Klassifikation bietet Unterabschnitt 3.5.4.

6. Welche Möglichkeiten bestehen zur Überführung existierender Softwaresysteme in den modellgetriebenen Ansatz?

Die in dieser Arbeit untersuchte Lösungsmöglichkeit ist das Reverse Engineering – andere Verfahren sind selbstverständlich denkbar. In Kapitel 4 konnte gezeigt werden das mehrere geeignete Möglichkeiten zur Erstellung von Modellinstanzen aus Software-Artefakten denkbar sind. Konkret demonstriert wurde die Generierung von Ecore-Klassenmodellen aus Java-Quellcode.

7. Lässt sich die Nutzerschnittstelle solcher Altsysteme automatisiert als Modellinstanz abbilden?

Diese Frage lässt sich ebenfalls positiv beantworten. Liegen geeignete Metamodelle der Oberflächensysteme dieser Altsysteme vor, so können deren Oberflächen prinzipiell automatisiert als Modellinstanzen beschrieben werden. Der Einsatz des in der Arbeit vorgestellten Proof-Of-Concepts - die Benutzungsoberfläche einer Java-Swing Anwendung wurde als Instanz des Swing-Metamodell abgebildet - ist allerdings auf bestimmte Rahmenbedingungen beschränkt.

8. Welche Vorteile und Verbesserungen lassen sich für die modellgetriebene Entwicklung von Benutzungsoberflächen zeigen?

Der Nachweis direkter Vorteile der modellgetriebenen Entwicklung ist für die Erzeugung von Benutzungsoberflächen schwer zu führen. Die mit den Mitteln und Ergebnissen dieser Arbeit erstellten GUIs halten einem Vergleich mit aktuellen, klassisch entwickelten, Oberflächen kaum Stand. Allerdings bieten die erarbeiteten umfangreichen Modelle nun, zumindest theoretisch, die Möglichkeit, GUIs gleicher Qualität zu erstellen.

Mag das Layout- und sonstige designerische Ergebnis derzeit noch nicht unbedingt höchsten Ansprüchen genügen, kommen doch die allgemeinen Vorteile eines MDSE zum Tragen. Die Entwicklung von

Software-Prototypen gelingt schnell. Die Separation-Of-Concerns funktioniert gut, durch Aufteilung einer UI-Deklaration auf verschiedene Modellebenen sind auch komplexe GUIs einfach wartbar. Gleichzeitig ist das Abstraktionsniveau so variabel das sogar die Einbettung von HCI-Patterns gelingt.

5.2 Vorschläge für weiterführende Arbeiten

5.2.1 Multimodale Nutzerschnittstellen

Die Arbeit ist primär mit Modellen für grafische Benutzungsoberflächen nach dem WIMP-Prinzip befasst. Daneben existiert jedoch noch eine Vielzahl weiterer Techniken für Benutzungsinterfaces, z.B. taktile Verfahren, Gestenerkennung, Sprachsteuerung oder 3-dimensionale Darstellungsverfahren. Eine Abschätzung inwiefern die präsentierten Modelle und Vorgehensweise auf andere UI-Modalitäten übertragbar sind ist daher interessant.

Grundsätzlich anwendbar ist der in Abschnitt 2.1 vorgestellte Gesamtprozess. Gerade der Fokus auf Plattform- und Geräteunabhängigkeit der Oberflächenbeschreibungen führte zu dessen Teilmodellen und Abstraktionsebenen. Dies beinhaltet ausdrücklich auch andere Interface-Modalitäten.

Ebenfalls problemlos geeignet für Nicht-WIMP UIs sind selbstverständlich die benutzten Transformationstechniken. Für die Anwendbarkeit von Model-To-Model oder Model-To-Text Transformationstechniken spielt keine Rolle, welche konkreten Wertausprägungen die Modellinstanzen haben, oder welchem Zweck die Modelle dienen. Entscheidend ist allein die Konformität zu den entsprechenden Metamodellen.

Die Metamodelle sind dagegen sehr wohl abhängig von der umzusetzenden oder angestrebten Modalität. Die vorgestellten und vorgeschlagenen Modelle für konkrete Oberflächen lassen sich kaum nutzbringend für andere Interface-Arten anwenden. Beispielsweise könnten die WIMP-Modelle auch in 3D-Umgebungen benutzt werden, sie wären nicht zur Beschreibung aller denkbaren und für diese Interface-Art markanten Features geeignet.

Je näher der Abstraktionsgrad der Zielplattform ist desto spezialisierter werden die beschreibenden Modelle sein. Wahrscheinlich ist es daher günstig für andere UI-Modalitäten spezialisierte Metamodelle zu definieren. Bei deren Konstruktion sollen und können natürlich Anleihen bei den WIMP-Modellen genommen werden, die Definition technischer oder semantischer Abhängigkeiten über Assoziationen oder Vererbungen usw. scheint jedoch zunächst wenig vielversprechend.

5.2.2 Vorschläge für weiterführende Forschungsansätze

Das Thema der modellgetriebenen Entwicklung von Benutzungsoberflächen hält noch viele interessante Fragestellungen bereit. Neben der bereits angesprochenen Anpassung des Prozesses für multi-modale Interfaces bietet auch der ordinäre WIMP-Erzeugungsprozess viele Ansatzpunkte für vertiefende Arbeiten.

Beispielsweise bei der Erzeugung eines Dialoggraphen aus Aufgabenmodellen, dem ersten Transformationsschritt im Rostocker Gesamtprozess. Derzeit handelt es sich dabei um einen, im Wesentlichen, manuellen Prozess. Zwar existiert ein Ansatz zur tatsächlichen Automatisierung der Transformation, dieser ist aber erst in groben Zügen ausgearbeitet und seine praktische Anwendbarkeit nicht ausreichend untermauert.

Bidirektionale Modelltransformationen sind ein integraler Bestandteil des vorgestellten MD-UID Prozess, denn dadurch werden sowohl Reverse Engineering als auch Migratory Interfaces wesentlich nützlicher. Obwohl die Bidirektionalität als Konzept gut verstanden und in ihrer Bedeutung anerkannt ist, wird der Erstellung von Rücktransformationen für den Prozess wenig Beachtung geschenkt. Für den Rostocker Prozess

sind derzeit verschiedene Transformationen zwischen konkreten Oberflächenmodellen sowie zur Ableitung einer abstrakten Oberfläche aus XUL-Instanzen vorhanden.

Das generelle Problem der Entwicklung geeigneter Rücktransformation besteht auch für andere Ansätze, z.B. bei MARIA, und wird von einer Reihe von Forschern untersucht. Eine Besonderheit im hiesigen Prozessmodell ist die Fragestellung nach der Ableitung von Dialoggraphen aus AUI-Deklarationen. An diesem Übergang liegt eine doppelte Schwierigkeit vor, die ersichtlich wird, wenn man sich die Kernelemente von Dialoggraphen vor Augen hält: Sichten, Aufgaben und Transitionen zwischen Sichten. Neben dem sich daraus ergebenden, bereits für sich schwierigen, Problem der Identifikation der Dynamik einer Oberfläche muss gleichfalls ein Aufgabenmodell aus den Quellmodellen generiert werden. Unter anderem um diesem Ziel näher zu kommen wird gegenwärtig die Synthetisierung nützlicher Aufgabenmodelle aus konkreten Rohdaten am Lehrstuhl untersucht. Ähnlich gelagerte Probleme werden weiterhin unter dem Stichwort „Process Mining“ erforscht.

Teilweise könnte es gelingen, in einem Swing-Oberflächensnapshot Merkmale zu identifizieren, die Rückschlüsse auf die vorgesehene Navigation beziehungsweise den Steuerfluss ermöglichen. Zielführend dürften solche Untersuchungen aber nur im Zusammenhang mit gleichzeitiger Analyse des zugrundeliegenden Java-Quellcodes sein. Womit sich die Frage stellt, ob hier überhaupt eine automatische Erkennungsheuristik anzustreben ist oder gleich untersucht werden sollte, wie ein menschlicher Modellierer bestmöglich unterstützt werden kann.

Solange für beide genannten Probleme, also Aufgabenmodell- und Dynamikerkennung, keine stabilen zufriedenstellenden Lösungen vorliegen, wird die Ableitung von Dialoggraphen aus den konkreteren Teilmodellen zur Oberflächendeklaration eine schwierige und offene Frage bleiben.

Weitere Ansätze für Forschungsarbeiten ergeben sich aus dem Patternkatalog. Einerseits bleibt das große Thema der Patternerkennung in vorhandenen Modellinstanzen relevant und andererseits hat sich herausgestellt, dass bestimmte Interface-Patterns geradezu bestimmte Strukturen in der Geschäftslogik oder den Datenstrukturen erfordern. Es sollten also die Zusammenhänge zwischen Patterns ausführlich erörtert werden, möglicherweise lassen sich dann einige Pattern-Ensembles finden, d.h. Pattern-Instanzen die oftmals gemeinsam auftreten.

Besonders erfolgversprechend scheint das Gebiet der Testautomatisierung zu sein. Einige der im Rahmen des Reverse Engineering diskutierten Methoden zur Instanzableitung von CUI-Modellen aus laufender Software, bieten interessante Ansatzpunkte zur Fremdsteuerung der untersuchten Software. Derzeit werden die Objekteigenschaften nur gelesen, es existiert jedoch kein prinzipielles Hindernis welches einen schreibenden Zugriff verhindern würde. Denkbar wäre es daher, die CUI-Instanzen zur Definition von Testskripten heranzuziehen, und die Testvalidierung ebenfalls mit typischen Methoden, etwa OCL, vorzunehmen.

Natürlich sind auch die Arbeiten an den CUI-Metamodellen nicht als abgeschlossen zu betrachten, interessant scheint beispielsweise auch das XUL-Metamodell dem Modellpruning zu unterziehen. Vermutlich ergibt sich hierfür keine Reduktion auf ein Zehntel oder Zwanzigstel wie bei dem Swing-Metamodell, aber es könnte gelingen, alle nicht darstellungsrelevanten Metaklassen aus dem Modell zu entfernen.

Eine abschließende Bemerkung gilt dem aktuell zur Verfügung stehenden Tool-Support. Generell sollten auch weiterhin Ressourcen in dessen Verbesserung gesteckt werden. Die Anwendung der meisten Werkzeuge des EMP wird Endanwendern nicht ohne Weiteres gelingen da die Tools wenig fehlertolerant, gleichzeitig aber schwer zu tracen und zu debuggen sind.

Konkrete, teils schwerwiegende, Probleme ergaben sich an mehreren Punkten. So war das unreduzierte Swing-Metamodell so umfangreich das die Generierung des Modellquellcodes, insbesondere mit Dif-

ferenzanalyse zu existierendem Quellcode, praktikable Zeit- und Speicherplatzgrenzen überschritt. Eine zweite gravierende Einschränkung trat im Bereich der Model-To-Text Transformationen mit Acceleo zutage; mit den verfügbaren Versionen war es nicht möglich mehr als eine Modellinstanz als Eingabe für Transformationen zu verwenden. Dies schränkt Acceleos Anwendungsmöglichkeiten derzeit noch empfindlich ein, beispielsweise ließ es sich auf Grund dieser Einschränkung nicht für Vergleiche von Modellinstanzen nutzen und konnte ebenfalls nicht direkt für die Wartung des Patternkatalogs eingesetzt werden.

Wünschenswert wären auch Arbeiten auf dem Gebiet der XML-Schema zu Ecore-Transformation. Wie bereits dargestellt, war das Ergebnis dieser Generierung nicht zufriedenstellend. Möglicherweise ließen sich die manuell durchgeführten Verfeinerungsschritte als Algorithmen definieren und zur Verbesserung dieses Prozesses einsetzen. Insbesondere sollte die Identifikation einer geeigneten Klassenhierarchie angestrebt werden.

Anhang A

System- und Sprachbeschreibungen

A.1 Allzweckmodellierung

A.1.1 UIDE

UIDE [FKKM91], von 1988, ist ein System zur Erzeugung textueller User Interfaces. Die Benutzeroberflächen werden dynamisch zur Laufzeit, durch Interpretation eines Modells, erzeugt und dargestellt. Dieses allem zugrundeliegende Modell ist die sogenannte Knowledge-Base.

Die Idee ist es, das gesamte Designwissen zu einer Anwendung in dieser Wissensdatenbank zu speichern. Zum Designwissen zählt in diesem Fall insbesondere auch das Modell der Anwendung, d.h. es werden die Domänenobjekte und deren Operationen in der Wissensdatenbank abgelegt. Diese Beschreibung ist formalisiert und definiert für die Operationen ordentliche Signaturen mit korrekten Datentypen für die Ein- und Ausgaben. Um dies zu erreichen ist die Wissensbasis als Objekt-Metamodell strukturiert. Aufgrund des frühen Entwicklungszeitpunktes liegt dieses Metamodell natürlich nicht als UML-Klassendiagramm vor. Die Definition des Metamodells erfolgt durch ART[1]-Schemata.

Modellinstanzen, also die Einträge in die Wissensbasis, werden über eine sogenannte Interface Definition Language (IDL) beschrieben. Diese IDL ist eine typische textuelle domänenspezifische Sprache.

Listing A.1 vermittelt einen Eindruck davon, zur Erklärung: Hier wird eine Aktion, d.h. Methode, create_shape in der Wissensbasis deklariert. Bei der Modellinitialisierung wird eine Variable shape vom Typ **number** deklariert und mit 0 als Startwert belegt. Es wird keine Vorbedingung zur Auslösung der Aktion verlangt, bzw. diese ist immer true und damit irrelevant. Die darauffolgende Zeile definiert die Eingabeparameter der Methode, es sind hier keine Rückgabetypen vorgesehen. Die Typen der Parameter müssen an anderer Stelle in der Wissensbasis vorliegen. Nach dem Methodenende wird als Nachbedingung dieser Aktion die Anzahl der shape hochgezählt.

```
initial: number(shape)=0
pre−condition: true
create_shape(p: position, c: color, a:angle, object_type: shape_class)
post−condition: number(shape)=number(shape)+1
```

Listing A.1: Ausschnitt einer UI-Spezifikation mit der UIDE-IDL

[1] Automated Reasoning Tool

Die Laufzeitkomponente, d.h. der Generator der als UI-Management System bezeichnet wird, soll aus der Wissensdatenbank ein voll-funktionales textuelles Nutzungsinterface erstellen. Dies scheint in Grundsätzen vergleichbar zu dem in dieser Arbeit verfolgten Ansatz, aus einem Klassenmodell die Benutzungsoberfläche abzuleiten. Auch die Entwickler von UIDE hatten das Problem, dass die Wissensbasis als solche bei weitem nicht genügend Informationen bereitstellt um nützliche Oberflächen daraus abzuleiten. Daher bilden sogenannte Constraints einen weiteren integralen Modellbestandteil von UIDE. Dabei handelt es sich um deklarative Regeln zur Steuerung des Generators. Letzlich sind die Constraints Abbildungs- oder Transformationsregeln. Sie definieren wie die einzelnen Parametertypen der Wissenbasis dargestellt werden sollen.

Der Generierungsprozess läuft folgendermaßen: Über die Auswertung der Vorbedingungen der Aktionen werden die zum jeweiligen Zeitpunkt ausführbaren Aktionen ermittelt. Die Methoden-Signaturen der Aktionen liefern dann die Informationen darüber, welche Daten vom Nutzer bereitzustellen sind. Und über die Constraints ist die UI-Darstellung für jeden abzufragenden Datentyp festgelegt. Damit ist es schlussendlich möglich, eine Eingabemaske darzustellen.

Selbstverständlich gibt es weitere syntaktische Konstrukte um Feinheiten der Oberflächenerzeugung zu steuern. Operationsparameter können als implizit, explizit oder optional definiert sein, mit den dementsprechende Auswirkungen auf die dem Nutzer zur Datenabfrage bereitgestellte UI. Ebensfalls wurden Vorkehrungen zur Ableitung von Menüstrukturen getroffen, es existiert ein sogenanntes Slot-Konzept. Diejenigen Aktionen die dem gleichen Slot zugewiesen werden bilden ein Menü. In Grenzen können Abhängigkeiten zwischen Menüeinträgen spezifiziert werden. Beispielsweise bewirkt die Definition als $\text{mutually}-\text{exclusive}$ action für Aktionspaare, dass immer nur eine der beiden Aktionen dem Nutzer bereitgestellt wird. Der Effekt ist, dass beide Aktionen ein und dieselbe Menüposition belegen. Weitere Eigenschaften, Syntaxelemente und deren Konsequenzen in der Constraint-Sprache werden in [GKF86] diskutiert.

Eine Reihe von grundsätzlichen Problemen bleibt im UIDE-Ansatz offen, bzw. werden durch die Beschränkung auf kleine Systeme mit textuellen Interfaces nicht relevant. So ist zum Beispiel die Anordnung und Visualisierung der zur Auswahl stehenden Aktionen nicht spezifiziert. Betrachtet man einen typischen Arbeitsprozess bleibt auch die Frage offen, wie die Ausgaben einer Operation sinnvoll als Eingaben für eine daran anschließende Operation benutzt werden können, ohne sie manuell einzugeben. Untersucht man das motivierende Beispiel zur Nutzung von UIDE in [FGK88] fällt zudem auf, dass bereits bei kleinen Problemen die Spezifikation widerspruchsfreier Vor-/Nachbedingungen für die Aktionen ein nicht-triviales Problem darstellt. Eine Machbarkeitsstudie für eine reale Software mit UIDE steht, soweit ich das beurteilen kann, weiter aus.

A.1.2 DON

Anfang der 1990er Jahre wurde UIDE für graphische Benutzungsoberflächen weiterentwickelt. Diese Weiterentwicklung erhielt den Namen DON, der Name entstammt dem englischen „to don = Sachen anziehen“. Die Entwickler bezeichnen DON als einen User Interface Presentation Design Assistant [KF90].

Aufgrund des GUI-Focus liegt ein Schwerpunkt von DON im Dialog-Design und der Menüerzeugung für WIMP-Oberflächen. Entsprechend seiner Abstammung verwendet DON ebenfalls eine globale Wissensbasis und Constraints. DON tritt mit dem Anspruch an, in den Constraints das Wissen und die Vorgehensweisen von menschlichen Designern in Algorithmen umzusetzen und diese zur Erzeugung besserer Benutzeroberflächen anzuwenden.

Die Interface Definition Language wurde gegenüber UIDE erweitert. Beispielsweise können nun Mengen- und Aufzählungstypen deklariert werden. Insgesamt steigt dabei die Ausdrucksmächtigkeit und der Detail-

grad der Spezifikationen.

```
style: set(1,4) of (Bold, Underline, Shadow, Italic) sequenzing ALPHABETICAL
```

Listing A.2: Regeldefinition für DON

Listing A.2 definiert den Mengentyp **style**. Parameter dieses Typs müssen genau einen der vier zur Auswahl gestellten Werte annehmen. Wenn ein Nutzer die Wertbelegung auswählt, sollen ihm die Optionen in alphabetischer Reihenfolge präsentiert werden. In der interpretierten Oberfläche könnte DON für **style** z.B. eine Dropdown-Box anbieten.

Das Erstellen von Benutzeroberflächen mit DON gleicht dem UIDE-Prozess. Die erste Schritt ist immer die Strukturierung der gewünschten Aktionen in der Wissensbasis, also die Festlegung auf ein Anwendungsmodell, mit Typen und Operationen. Unter Anwendung der Constraints des Regelsystems findet im Anschluß daran das Mapping von Aktionen auf passende Userinterface-Objekte (UIO) statt.

Im Unterschied zu UIDE wird von DON das Problem der Anordnung der Oberflächenelemente behandelt. Auch Layouting-Verfahren können in Constraints umgesetzt werden. Das hierfür innerhalb von DON aufgestellte und implementierte Regelsystem soll generelle Design-Prinzipien, die auf Erkenntnissen aus der Gestaltungspsychologie [Wer44] basieren, umsetzen.

Das bedeutet, dass die Wertheimerschen Kriterien Nähe (proximity), Gleichartigkeit (similarity), Ausrichtung (direction) und ggf. Enthaltensbeziehungen (closure) in Constraints umgesetzt wurden. Zusätzlich identifizierte Foley, ein Mitglied des DON-Entwicklunsteams, während seiner Arbeit an DON die Kriterien Balance (balance) als Objektverteilung auf der Oberfläche, Proportionen (proportion) und die Ausrichtung an Gittern (gridding). Da die Transformationsregeln erweiterbar sind, ist diese Kriterienliste prinzipiell nicht abgeschlossen, sondern eine Erweiterung um weitere Kriterien war immer möglich und angestrebt.

Im Gegensatz zu UIDE können für die Ausführbarkeitsentscheidung und die Konsequenzen einer Aktion mehrere Vor- und Nachbedingungen festgelegt werden. Die Definition von Aktionsabfolgen muss auch bei DON deklarativ, über die geeignete Festlegung von Vor- und Nachbedingungen, erfolgen. DON versucht, durch Auswertung der über die Bedingungen definierten Aktionszusammenhänge zu ermitteln, welche Aktionen zusammengruppierbar sind. Dies ist insoweit bedeutsam, da es sich dabei um die Aktionen handelt, die gleichzeitig auf der Oberfläche dargestellt werden müssen.

Die Durchführung dieser Gruppierung wird automatisiert durch zwei Hauptkomponenten, Manager genannt, vorgenommen. Diese Manager sind im Wesentlichen Regelsammlungen welche die Weinheimerschen und Foleys Layout-Prinzipien so weit als möglich umsetzen.

Die erste Komponente ist der *Organisation*-Manager, dieser gruppiert die im Anwendungsmodell hinterlegten Aktionen entsprechend ihrer Parameter und erstellt eine Struktur darüber. Diese Struktur bildet die Grundlage für das Menü der Anwendung. Der Organisation-Manager wertet dazu den Detailgrad, die gegenseitigen Abhängigkeiten und den Geltungsbereich der Spezifikationen der Operationen aus und ordnet die Aktionen zunächst in eine Prioritätsklasse ein. Im DON-Ansatz wurden 9 Klassen bzw. Gruppierungen identifiziert. Die Klassen 1-7 sind dabei direkt aus der Spezifikation ableitbar, beispielsweise ist die Bedingung für die Einordnung in die Klasse 6, dass Vor- und Nachbedingung der Operation identisch sind. Für die beiden Klassen 8 und 9 wird menschliches Wissen bzw. eine manuelle Designentscheidung benötigt. In Klasse 8 wird die „Erwartete Wichtigkeit“ und in Klasse 9 die „Erwartete Nutzungshäufigkeit“ bewertet. Vom Organisation-Manager wird ausserdem das Mapping von Aktionen auf User Interface Objekte nach den Auswahlregeln durchgeführt. Die Auswahl der gewünschten Mappings kann über Profile gesteuert werden, durch Austausch der verwendeten Profile kann eine Anpassung an unterschiedliche Nutzungsszenarien erfolgen.

Die zweite Hauptkomponente ist der *Presentation-Manager*, er ist für das eigentliche Layout von Menüs und Dialogboxen verantwortlich. Der Presentation-Manager nutzt die Ergebnisse der Mappings des Organisation-Managers und die Einträge in der Wissensbasis. Seine Hauptaufgabe ist die Zuordnung der Interface-Objekte zu Layoutcontainern.

Der Zweck dieses Managers ist, es Probleme wie Überfüllung oder Verletzung der Gestaltungskriterien der Oberfläche zu handhaben. Als Hauptproblem hat sich beim Einsatz von DON die Überfüllung herausgestellt. Im Rahmen des sogenannten Overflow-Handling entscheidet der *Presentation-Manager* welche Elemente der Oberfläche angezeigt, modifiziert oder ausgeblendet werden. Hierfür werden verschiedene Strategien verfolgt, diese sind in der Reihenfolge ihrer Auswirkungen auf die Gestaltung der Benutzeroberfläche priorisiert. Als Low-Impact Problembehebungsstrategie, und damit eine der bevorzugten Strategien, wird z.B. Größenanpassung einzelner Oberflächenelemente angesehen. Demgegenüber zählt das Austauschen der Visualisierung einer Aktion durch eine andere Darstellungsform zu den High-Impact Strategien. Beispielsweise könnte ein Slider-Control durch ein schmales Textfeld ersetzt werden, welches horizontal weniger Platz benötigt. Dieser Austausch ändert das Aussehen der Benutzeroberfläche erheblich, behält aber dennoch die gleichen Ein- und Ausgabemöglichkeiten bei. Das Ziel des *Presentation-Manager* ist es, innerhalb der durch den Organisation-Manager aufgebauten logischen Ordnung, eine passend layoutete Umgebung zu erstellen. Ein Durchbrechen dieser genannten Ordnung ist in den Strategien zwar vorgesehen, jedoch als High-Impact Strategie entsprechend niedrig priorisiert.

Organisations- und Presentation-Manager erzeugen zum Start der Applikation die gesamte Oberfläche. Im Unterschied zu UIDE findet bei DON keine Anpassung der Benutzungsoberfläche zur Laufzeit mehr statt. DON ist ein interessanter Ansatz für GUI-Oberflächen. Anscheinend treten jedoch die gleichen Schwierigkeiten wie bei UIDE auf. Alle möglichen Handlungsfolgen einer realen Anwendung rein deklarativ zu beschreiben ist sehr aufwändig. Darüberhinaus für jeden erreichbaren Anwendungszustand die Benutzungsoberfläche vorgeneriert zur Verfügung zu halten ist ebenfalls ein äußerst schwieriges Unterfangen. Es ist daher nicht wahrscheinlich, dass die Kombination dieser beiden Probleme die anzustrebende Lösung der modellgetriebenen Oberflächenentwicklung darstellt. Die Tatsache das in den letzten zwanzig Jahren keine DON-basierten Anwendungen entstanden unterstreicht dieses Urteil.

A.1.3 HUMANOID

HUMANOID [SLN92] ist prinzipiell ein Designwerkzeug für grafische Nutzerinterfaces, das Anfang der 1990er Jahre entstand. Gleichzeitig handelt es sich hierbei um die Proof-Of-Concept Implementierung einer speziellen Designherangehensweise. HUMANOID liegt die Idee zugrunde, dass die schwierigsten Designentscheidungen menschlichen Designern überlassen werden sollen. Die Designer sollen durch Automatisierung bei repetitiven Standardtätigkeiten und -entscheidungen entlastet werden. Das besondere im Ansatz von HUMANOID ist es, das High-Level Design eines User Interface als das zu erreichende Ziel zu definieren und sich diesem Ziel durch Dekomposition in Design-Teilziele anzunähern.

Ein High-Level Design-Ziel im Sinne von HUMANOID ist eine Beschreibung dessen was mit dem betrachteten Teil des User-Interface erreicht werden soll. Das Design-Ziel ist also explizit kein UI-Mockup oder Prototyp, sondern eine teilformalisierte textuelle Beschreibung einer Interface-Aktion.

Für HUMANOID wurde eine Ontologie von sogenannten Top-Level Interface-Zielen und diesen zugeordneten Unterzielen entwickelt. Beispielsweise ist das Top-Level Ziel „Made-Object-Draggable“ definiert, d.h. ein UI-Objekt ziehbar machen. Diesem werden die Unterziele „Dragging-Interactor (Behaviour)-Elaborated “, etwa Zieh-Auslöser ausgearbeitet, und „Feedback-Template-Elaborated“ , etwa als Rückmeldung ausgearbeitet übersetzbar, zugeordnet. Viele Unterziele in der HUMANOID-Ontologie enthalten

selbst weitere Unterziele. Diese Schlagworte erinnern Designer, welche Probleme noch bedacht und geklärt werden müssen.

Durch Zusammensetzung solcher Haupt- und Unterziele wird die Abarbeitung einer Nutzeraktion beschrieben. Wenn auf diese Weise die Beschreibung fertiggestellt ist, folgt der automatisierte Teil von HUMANOID, die Prototypen-Generierung. HUMANOID wurde als ein modellbasiertes System umgesetzt.

Im Ergebnis einer Modellierung mit HUMANOID entsteht eine, in der HUMANOID-Laufzeitumgebung, ausführbare UI. Es handelt sich dabei um einen Oberflächen-Prototyp. Außer der in HUMANOID selbst hinterbaren Modifikation von Anwendungsobjekten und bietet dieser Prototyp normalerweise keine Anbindung an extern programmierte Geschäftslogik.

Bei HUMANOID gibt es ein Modell der Anwendung, welches inhaltlich sehr ähnlich der Wissensbasis von DON/UIDE ist. Es enthält Informationen über die Objekte und Operationen der Anwendungslogik. Und ähnlich der DON-Idee definieren auch hier die Operationen implizit die gewünschten Funktionalitäten auf der Oberfläche. Dieses Modell wird nach [SLN92] in fünf „halb-unabhängige“Dimensionen unterschieden. Letztlich sind unter diesen Dimensionen Teilmodelle zu verstehen.

Die erste Dimension, *Application semantics* entspricht in etwa einem Domänenmodell, beschreibt also die Einsatzumgebung der Software. Hier werden die Anwendungsobjekte und ausführbare Befehle (Commands) definiert und dabei deren Datentypen festgelegt. Auch in HUMANOID existiert das aus UIDE bekannte Slot-Konzept, allerdings werden hier Commands zu Slots zugewiesen. Commands unterscheiden sich von den Aktionen bei UIDE dadurch, dass zusätzlich die Aspekte Dateneingabe und CallBack-Procedure[2] definiert werden können.

Das *Presentation*-Modell als zweite Dimension beschreibt die grafische Darstellung von Daten und Commands. Dieses Teilmodell benutzt hierarchisch organisierte Templates. Die Blattknoten in diesem Hierarchie-Baum sind direkte grafische Primitive, Widgets oder sonstige building-blocks des zugrundeliegenden Grafiksystems. Über ebenfalls Slots genannte Parameter werden den Templates die darzustellenden Informationen übergeben. Die HUMANOID-Umgebung stellt diverse Template-Bibliotheken zur direkten Weiterverwendung zur Verfügung.

Die dritte Dimension ist das *Manipulation*-Modell. Dieses Modell stellt die Verbindung zwischen Nutzer-Eingaben und den Objekten des *Presentation*-Modells her. Die betrachtete Grundeinheit sind Gestures, das Wort ist ein Versuch sind sämtliche Interaktionsmöglichkeiten in einen Begriff zu fassen. Unter Gestures ist daher die Gesamtheit von Tastendrücken, Mouseevents und anderen denkbaren Interaktionsmöglichkeiten zu verstehen. Dieses Teilmodell fügt letztlich eine Verhaltensspezifikation zu den Templates des *Presentation*-Modells hinzu. Es kann mit dem Dialog- bzw. dem Präsentationsmodell anderer Systeme verglichen werden. Der Inhalt dieses Modell sind Schlüssel-Wert Paare der Art: (“Cursortaste runter gedrückt“,“reduziere Wert um 4“).

Im vierten Teilmodell *Action side effects* werden regelmäßig benutzte Informationsaktionen definiert. Darunter werden in HUMANOID Tonausgaben oder Hinweisnachrichten verstanden, welche den Nutzer sofort über den Status einer Aktion informieren können. Die verwendeten Schlüsselwörter dazu sind selbsterklärend, z.B.: Beep−When−Correct oder Message−When−Correct. Es gibt eine Bibliothek mit derlei Standardaktionen, die allgemeine Wirkungsweise entspricht jener von Templates im Dialogmodell.

Die fünfte Dimension ist das *Sequencing*-Modell. Es ist ein Dialog-Modell und beschreibt die Abfolge der Eingabemasken der Anwendung sowie die Aktivitätenreihenfolge für einzelne in einer Maske enthaltene Elemente. Es werden keine expliziten Aktivierungsanweisungen durch die Designer definiert, sondern die Reihenfolge wiederum mittels geeigneter Constraints deklarativ spezifiziert. Die Constraint-Sprache zur

[2] Aufzurufende Methode nach Durchführung der Operation

Beschreibung der Sequenzierung unterscheidet zwischen Eingabe-, Gruppierungs- und Befehlselementen. Die Laufzeitumgebung von HUMANOID wertet die Reihenfolge während des Programmablaufes aus und schaltet zwischen den entsprechenden Masken und Aktivitäten um. Auch für diesen Teilaspekt existiert eine Bibliothek mit typischen Sequenzierungen.

Die Erstellung einer Oberfläche mit HUMANOID zerfällt somit in zwei sehr unterschiedliche Teile. Zunächst sind alle erwünschten Top-Level Ziele zu identifizieren. Durch Anwendung der Begrifflichkeiten aus der Ontologie ist das Erreichen dieser Hauptziele durch Einsetzen erforderlicher Unterziele zu beschreiben. Gleichzeitig sind die zuvor beschriebenen Teilmodelle zu erstellen.

Abschließend müssen die Verbindungen zwischen der Zielhierarchie und den Teilmodellen, und ggf. die Beziehungen zwischen den Teilmodellen, aufgebaut werden. So könnte z.B. einem Teilziel „Show-Object-Contents“ eine Textbox zur Darstellung und ein Domänenobjekt als Inhaltslieferant zugewiesen werden. Ebenfalls könnten die Auswirkungen von Cursorbewegungen in der Textbox festgelegt werden. Das Ergebnis dieser Mappings wird durch die HUMANOID-Umgebung interpretiert und dargestellt.

HUMANOID wurde nie verwendet um reale Anwendungen zu gestalten. Die verwendeten Teilmodelle scheinen jedoch durchaus detailliert zu sein. Die Idee der Zerlegung der Designziele hat sich nicht durchgesetzt, möglicherweise weil die Abstraktionsebene auf welcher operiert wurde bereits zu niedrig, zu implementierungsnah war.

A.2 Modellierung für Spezialbereiche

A.2.1 DRIVE

Als erstes System dieser Kategorie soll DRIVE [MK95] aus dem Jahr 1995 vorgestellt werden. Hierbei handelt es sich um eine experimentelle Softwareumgebung und ein Framework für die Definition und Erstellung von Benutzeroberflächen von Datenbank-Anwendungen.

Im Selbstverständnis der DRIVE-Entwickler sollte eine UI-Erprobungsumgebung („sketch-pad“) für neue Techniken bei Nutzungsoberflächen geschaffen werden. Besonderer Fokus wurde auf die Möglichkeit der Anbindung 3-dimensionaler Visualisierungstechniken gelegt. Um zur Evaluierung verschiedener Techniken eine Vergleichbarkeit zwischen den diversen Oberflächenprototypen herzustellen, sollte die Präsentation gleicher Daten und das Auslösen der gleichen Geschäftslogik ermöglicht werden. Zur Vermeidung zeitintensiver und fehlerträchtiger Mehrfachimplementierungen bei Daten und Programmlogik, wurden auf modellbasierte Techniken eingesetzt. Der Name DRIVE steht daher für **D**atabase **r**epresentation **i**ndependent **v**isual **e**nvironment, frei übersetzt etwa „Datenmodellunabhängige Visualisierungsumgebung“.

Das DRIVE-Framework besteht aus mehreren Teilen, deren Zusammenspiel in Abbildung A.1 angedeutet wird. Es handelt sich dabei um eine von mir vorgenommene Verallgemeinerung der Abbildung 4 aus [MK95]. Zur Interpretation ist die Darstellung von Rechts nach Links zu lesen, die Originalautoren entschieden sich für diese Leserichtung.

Das Fundament des DRIVE-Ansatzes bilden zwei Modelle, das Nutzer- und das Datenmodell. Wie üblich besteht das Datenmodell aus einem Klassenmodell der Anwendungsobjekte. DRIVE wurde speziell für objektorientierte Datenbanken entwickelt. Das Datenbankschema der Anwendung ist daher gleichzeitig das Datenmodell im DRIVE-Ansatz. Außerdem wird durch die DB-internen Methodenimplementierungen die Geschäftlogik bereitgestellt. Für das Nutzermodell von DRIVE wird ein eigenes Metamodell vorgeschlagen, dieses soll hier nicht weiter betrachtet werden.

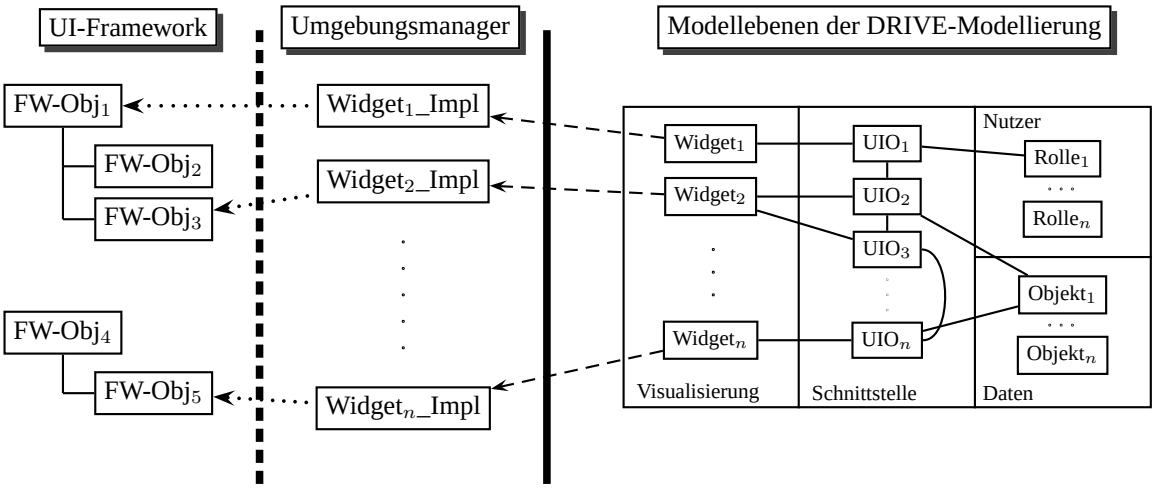

Abbildung A.1: DRIVE - Modellsystem, nach [MK95]

Ausgehend von Nutzer- und Datenmodell wird im DRIVE-Ansatz die Visualisierungsschnittstelle modelliert. Im ersten Schritt wird entschieden welche Interaktionsobjekte (UIO) die Nutzungsoberfläche zur Verfügung stellen soll. Dabei wird auch festgelegt, an welchen UIOs Instanzen von Daten- oder Nutzermodell beteiligt sind, sei es als Datenquellen/-senken oder Interaktionsauslöser. Die gefundenen Beziehungen werden als Assoziationen auf Klassenmodellebene, mit einer eigenen Modellierungssprache ausgedrückt, spezifiziert. Mit einiger Berechtigung kann die Visualisierungsschnittstelle als abstrakte Oberfläche angesehen werden.

Nachdem die Schnittstelle fixiert ist, kann die eigentliche Visualisierung festgelegt werden. Dies besteht im Wesentlichen aus einer Zuordnung der UIOs zu bestimmten vordefinierten Widgettypen. Diese Beziehungen zu den vordefinierten Widgets des DRIVE-Frameworks werden wiederum durch Assoziationen auf Klassenebene modelliert. Bemerkenswert ist hier, dass das Klassenmodell der Widgets selbst editiert werden soll; heutzutage würde vermutlich eher ein Plugin-Mechanismus oder Vererbung eingesetzt werden. Damit ist die DRIVE-Modellierung im Grunde abgeschlossen.

Für die vorgegebenen DRIVE-Widgetklassen existieren C/C++ Implementierungen, diese Bilden den Kern des sogenannten Umgebungsmanagers. Dieser Bestandteil der DRIVE-Laufzeitumgebung nutzt die Informationen aus dem Visualisierungsmodell um die notwendigen Instanzen der vorimplementierten Zwischenmodellklassen zu instanziieren. Die Laufzeitumgebung berücksichtigt dabei auch die spezifizierte Assoziationskette zu Daten- bzw. Nutzermodell. Damit ist an dieser Stelle der ursprüngliche Anspruch von DRIVE erfüllt; Daten, Geschäftslogik und ein Interface-Modell werden Datenbankunabhängig zur Visualisierung bereitgestellt. Natürlich kann, solange kein UI-Framework zur eigentlichen Darstellung angegeben wurde, noch keine Nutzungsoberfläche dargestellt werden. Das DRIVE-Framework sieht hierfür spezialisierte Editoren und die Benutzung von Drag&Drop-Techniken vor. In [MK95] wurde auf diesem Weg eine Beispielobefläche für einen Museumsführer entwickelt. DRIVE selbst fand kaum Verbreitung, Ideen dieses Ansatzes wurden aber in TEALLACH weitergeführt.

A.2.2 TEALLACH

Bei TEALLACH [GBM+99] handelt es sich ebenfalls um ein Framework zur modellbasierten Entwicklung von Nutzeroberflächen für Datenbankanwendungen. Es ist eine Entwicklung, die am Ende der 1990er

Jahre aus einer Zusammenarbeit der britischen Universitäten Glasgow, Manchester und Edinburgh entstand. Schwerpunkt des damaligen Forschungsverbundes war die Generierung von UIs für Applikationen welche auf objektorientierte Datenbankmanagementsysteme aufsetzen. Die Hauptentwicklungsplattform für TEALLACH war das DBMS POET[3].

TEALLACH nutzt drei Hauptmodelle; ein Domänenmodell, ein Aufgaben-Modell und ein Präsentationsmodell. Das Domänenmodell bildet die Ausgangsbasis der Modellierung. Für dieses Teilmodell sind keine Referenzen zu den anderen Modellen zugelassen. Die Instanzen von Aufgaben- und Präsentationsmodell dürfen durchaus auf Einträge untereinander verweisen, wie auch auf Objekte des Domänenmodells.

Die Motivation für diese Einschränkung ist möglicherweise auf technischer Seite zu finden. Ähnlich DRIVE ist das Domänenmodell ein objektorientiertes Klassenmodell. Und dieses entsteht durch Ableitung, soweit möglich, direkt aus dem Datenbankschema der zu erstellenden Anwendung. Referenzen zu einem Präsentations- oder Aufgabenmodell sind in solchen Datenbankschemata im allgemeinen nicht vorgesehen.

Die Schemadefinitionen des OO-DBMS POET erfolgten mit der ODL[4]. Wegen der zum damaligen Zeitpunkt eingeschränkten Akzeptanz von ODL wurden von Anfang an auch alternative Konfigurationsmöglichkeiten vorgesehen. Dazu wurde eine Möglichkeit geschaffen, auch nicht ODL-konforme Schemadefinitionen zu interpretieren und als Domänenmodell zu verwenden.

Das Aufgabenmodell TEALLACHs entspricht in seiner Grundstruktur den Aufgabenmodellen anderer MB-UID Systeme. Es handelt sich um eine zielorientierte Aufgabenhierarchie, in deren Blattknoten, auch primitive Aufgaben genannt, die eigentlichen Interaktionen und Systemaktionen spezifiziert werden. Zur Beschreibung der Ablaufbeziehungen werden zwischen den Geschwisterknoten des Aufgabenhierarchiebaums temporale Operatoren notiert. Bei TEALLACH-Modellen können die in Tabelle A.1 beschriebenen sieben Operatoren verwendet werden. Die primitiven Aufgaben referenzieren direkt die für ihre Ausführung notwendigen Datenobjekte und besonders auch die auszulösenden Methoden des Domänenmodells.

Im Präsentationsmodell wird die Nutzungsoberfläche definiert. Es unterscheidet zwei Abstraktionsebenen; das abstrakte (APM) und das konkrete Präsentationsmodell (CPM). Im APM wird die Oberfläche skizziert, hierbei wird für die diversen Interaktionsvorgänge eine sogenannte Darstellungskategorie gewählt. Da die Interaktionen als Blattknoten im Aufgabenmodell definiert sind, findet auf dieser Modellebene die Koppelung von Aufgaben- und Präsentationsmodell statt. Beispiele für die im TEALLACH-APM vorgesehenen Kategorien sind *Chooser* - Elementauswahl, *Editor*, *Display* - Datenanzeige und *ActionItem* - Aktionsauslösung. Zur genaueren Charakterisierung des jeweiligen abstrakten Interaktionselementes sind für die

[3] inzwischen als Versant FastObjects vermarktet
[4] Spezifikationssprache der ODMG

Operator	Bedeutung
Sequential	Sequentielle Ausführung
Order-Independent	Ausführungsreihenfolge egal, Ziel nach Beendigung aller Tasks erreicht
Concurrent	Parallele Ausführung, Ziel nach Beendigung aller Tasks erreicht
Repeatable	Iteration, Anzahl oder Bedingungsbeschränkt
Choice	Auswahl durch den Nutzer, impliziter Choice
Optional	0..n aller Subtasks eines Tasks auszuwählen durch den Nutzer
	parallele Ausführung, Beendigung aller Subtasks $\hat{=}$ Zielerreichung
Conditional	Bedingte Auswahl, expliziter Choice

Tabelle A.1: Temporale Operatoren zwischen Aufgaben in TEALLACH [GBM+99]

einzelnen Kategorien noch Steuerparameter vorgesehen.

Über die Wertbelegung dieser Parameter erfolgt die Feinjustierung der Mappings des konkreten Präsentationsmodells. In dem CPM wird die Implementierung der APM-Kategorie für die einzelnen Oberflächenbestandteile fixiert. Diese Implementierung wird durch die Widgets von realen UI-Frameworks gestellt, in [GBM+99] wurde von Java Swing genutzt.

Ein besonderes Merkmal von TEALLACH ist, dass Objekte des Domänenmodells direkt als Vorlage für ihre Repräsentation im konkreten Präsentationsmodell verwendet werden können. Mittels Introspektion wird dabei für alle Getter und Setter eine Standarddarstellung gemäß des benutzten Datentypes gewählt. Aus diesen Standarddarstellungen wird eine fertig konfigurierte Komponente für die Oberfläche generiert. Offen bleibt, wie die Ableitung des zugehörigen APM aus den so erstellten CPM-Instanzen, d.h. eine geeignete Kategorisierung dieser Komponenten für die Verwendung im abstrakten Präsentationsmodell, erfolgen kann. Gravierender scheint jedoch, das ohne APM eigentlich keine Verbindung zum Aufgabenmodell und damit zur Geschäftslogik erstellt werden kann.

Die für TEALLACH erstellte Werkzeugunterstützung unterstützt bereits das visuelle Editieren von Modellen und Assoziationen mittels Drag&Drop. Nachdem alle Teilmodelle spezifiziert sind, generiert das TEALLACH-Framework den Quellcode einer vollständigen Anwendung; normalerweise Java.

Die Entwicklung von TEALLACH wurde anscheinend Anfang der 2000er Jahre eingestellt. Der Ansatz, aus der in einer Datenbank gekapselten Anwendungslogik über ein, wie auch immer geartetes Zwischenmodell zur Oberflächenbeschreibung, eine voll-funktionsfähige Anwendung zu erzeugen, wird in der Praxis durchaus weiterverfolgt. So ist etwa die „Oracle Forms“-Technologie ein prominentes Beispiel dieser Herangehensweise.

A.3 Systeme mit Aufgabenmodellierungshintergrund

A.3.1 ADEPT

ADEPT [MPWJ92], der **A**dvanced **d**esign **e**nvironment for **p**rototyping with **t**asks, ist ein solches Framework für aufgabenmodellbasiertes User-Interface Design. Es wurde im Jahr 1992 vorgestellt.

Zur Aufgabenmodellierung nutzt ADEPT ein an die sogenannten Task-Knowledge-Structures – TKS [JJ91] angelehntes Aufgabenmodell. TKS sind Aufgabenmodelle deren Struktur speziell darauf zugeschnitten wurde, Interface-Designer bei der Gestaltung gebrauchstauglicher Benutzungsoberflächen zu unterstützen. In TKS werden Aufgaben in eine Taxonomie eingeordnet, zielorientiert strukturieriert und die zu ihrer Durchführung notwendigen Handlungen notiert.

Unter einer zielorientierten Struktur ist die für Aufgabenmodelle typische hierarchische Zerlegung des obersten Nutzerziels zu verstehen. Zur Notation dieser Dekomposition nutzt ADEPT zusammenhängende Graphen. Die Abfolge der Teilaufgaben wird durch Operatoren gesteuert, die in den ADEPT-Graphen direkt an die Kanten angezeichnet werden. Die bereitgestellten Operatoren sind eine Untermenge derjenigen von Hoare in [Hoa78] für Communicating Sequencing Processes (CSP) vorgestellten.

ADEPT nutzt neben dem Aufgabenmodell noch ein explizites *User* und ein *Interface*-Modell. Das User-Modell besteht aus (Schlüssel,Wert)-Tupeln, genannt Facts. Diese beschreiben die Nutzer genauer, d.h. deren Fähigkeiten und Fertigkeiten. Es enthält Informationen der Art: (’application experience’, ’high’) oder (’frequency of use’,’high’). Eine Ontologie für die möglichen Schlüssel dieses Modells ist nicht öffentlich verfügbar.

Benutzt werden die Facts durch die sogenannten Design Rules, dies sind 3-Tupel (Schlüssel k, Falls-Wert v, Dann-Regel r), mit der Bedeutung: falls ein Fact mit dem Schlüssel k und dem Wert v existiert, dann wende Regel r an. Diese Regeln beeinflussen die Ableitung des abstrakten Oberflächenmodells aus dem Aufgabenmodell.

Das 3-Tupel: $('typingskills','low','fill-in\ forms')$ beschreibt eine solche Regel. Sie besagt, dass langsam tippenden Nutzern ein Formular bereitgestellt werden sollte. Fortgeschrittenen Tastaturnutzern soll nach der Regel $('typingskills','high','free-form\ text')$ ein Freitextbereich für die Dateneingabe erzeugt werden.

In ADEPT wird zwischen abstrakten und konkreten Oberflächen unterschieden. Beide Abstraktionsebenen enthalten Interfacebeschreibungen als eine Hierarchie zusammengesetzter Objekte. Aussagen über den Charakter der darin referenzierbaren Interfaceobjekte sind kaum zu finden. Es liegt anscheinend kein öffentlich zugängliches Metamodell hierfür vor; es findet sich jedoch eine Absichtserklärung der ADEPT-Entwickler, welche die Interfaceobjekte bis auf die Ebene von Input-Ereignissen und Zustandsautomaten spezifizieren wollten.

Der Gesamtprozess des ADEPT-Ansatzes beginnt mit der Spezifikation von Aufgabenmodell und Nutzermodell. Eine Interface-Generator genannte Komponenten erzeugt aus Aufgaben- und Nutzermodell ein abstraktes UI-Modell. Dieses wird durch den UI-Generator in ein konkretes Oberflächenmodell transformiert. Im Ergebnis des ADEPT-Prozesses entsteht der UI-Quellcode für eine Programmiersprache. ADEPT selbst wurde mit Smalltalk implementiert und generiert standardmäßig Quellcode dieser Sprache.

A.3.2 MASTERMIND

MASTERMIND [SSC+96] ist eine Modellierungs- und Generierungsumgebung aus der Mitte der 1990er Jahre, Veröffentlichungen finden sich zwischen 1995 und 1998, der Name des Systems wird als „Models Allowing Shared Tools and Explicit Representations to Make Interfaces Natural to Develop“ erklärt. Zur Enstehungszeit von MASTERMIND lagen bereits einige Erfahrungen mit älteren MB-UID Werkzeugen und Ansätzen vor. Die Untersuchung von UIDE, ADEPT, HUMANOID und MECANO schienen einen kurz- oder mittelfristigen Erfolg des Paradigmas nicht erwarten zu lassen. Das Forschungsteam stellte sich daher die Frage, welche Schwächen denn diese Systeme hatten, die eine breite Anwendung verhinderten. Sie identifizierten in [SSC+96] drei Problemschwerpunkte: eine insgesamt mangelnde Flexibilität, sowie dürftige Qualität der Ergebnisse bei zumeist umständlicher Benutzung. Der Anspruch von MASTERMIND ist es, diese Probleme zu beheben oder zumindest im Ausmaß drastisch zu reduzieren.

Der Lösungsvorschlag von MASTERMIND besteht darin, zum einen die Abstraktionsebenen zu reduzieren und zum anderen erprobte und verbreitete Technologien, anstelle von Insellösungen, einzusetzen. Daher sind viele Aspekte der Modellierung von MASTERMIND an CORBA angelehnt, einer damals aufstrebenden und auch heute noch relevanten OMG-Technologie. Prinzipiell finden sich auch hier die üblichen Modellarten der MB-UID; Application-, Task- und Präsentationsmodell werden verwendet.

Die MASTERMIND-Umgebung strebt die Unterstützung verteilter Anwendungsentwicklung an. Dazu werden alle Modellinstanzen und Modellbeschreibungen zentral auf einem Modell-Server bereitgestellt. Alle Werkzeuge, Generatoren und Editoren greifen auf dieses einheitliche Repository zurück. Dank der einheitlichen CORBA-Schnittstelle zu den Modellen ist auch eine verhältnismäßig einfache Austauschbarkeit der Werkzeuge möglich.

Eine andere Besonderheit von MASTERMIND ist die vordergründige Unterscheidung in Runtime- und Design-Time Modelle. Die Runtime-Modelle werden als komprimierte und kompilierte Variante der Design-Time Modelle dargestellt. Hauptzweck dieser expliziten Runtime-Modelle ist es die erzeugte Anwendung,

durch Wegfall der sonst nötigen Modell-Interpretation, zu beschleunigen. Auf den Komprimierungsvorgang an sich wird nicht direkt eingegangen, so dass offen bleibt ob damit nicht ohnehin nur die kompilierten Versionen der CORBA-Objekte gemeint sind.

Zum Application-Modell ist wenig zu sagen, es handelt sich um das Domänenmodell der Anwendung welches mit einer CORBA-IDL ähnlichen Sprache definiert wird. Dabei entsteht naturgemäß ein OO-typisches Modell mit Klassen, Attributen und Methoden. Interessanter ist das Aufgabenmodell. MASTERMIND verwendet den Begriff des Ziels nicht, wie oft üblich, als das Wurzelelement der Aufgabenhierarchie. Jedem einzelnen, auch atomaren, Task kann ein Ziel zugeordnet sein. Entweder textuell ausgedrückt oder als formalisierter Term, MASTERMIND bietet hierfür eine eigene Constraint-Sprache. Sofern das Ziel als Text angegeben wird handelt es sich um wenig mehr als Dokumentation. Terme hingegen werden zur Ausführungszeit im Kontext der Aufgabe ausgewertet und bestimmen darüber ob der Task überhaupt auszuführen ist. Ebenfalls vom Standardkonzept für Tasks abweichend ist, dass Aufgaben immer von vordefinierten Aufgabenprototypen erben. Als Beispiel wird eine Aufgabe „Email ausdrucken“ genannt, welche eine Spezialisierung der generischen Aufgabe „Ausdrucken“ ist. Eine öffentliche verfügbare Bibliothek dieser Aufgabenprototypen ist jedoch anscheinend nicht erhältlich. Auch in MASTERMINDs Aufgabenmodell werden Tasks zu Kategorien zugeordnet, in Task-Hierarchien strukturiert und die Abarbeitungsabfolge mittels temporaler Operatoren gesteuert.

Wie das Aufgabenmodell wird auch MASTERMINDs Interface-Modell mit einer eigenen Sprache textuell spezifiziert. Im Interface-Modell werden die Interaktions-Objekte (UIO) der Anwendung beschrieben. Ähnlich der Situation bei den Tasks, muss jeder neudefinierte UIO-Typ von einem vordefinierten UIO-Prototypen erben. Die Beschreibungsebene der abstrakten Oberfläche entfällt in MASTERMIND, es werden nur konkrete UI-Widgets beschrieben. Layout und Aussehen der Widgets werden im Interfacemodell bereits pixelgenau beschrieben, auch dies ist eine bemerkenswerte Abweichung zu anderen MB-UID Ansätzen. Komplette Benutzungsoberflächen entstehen auch bei MASTERMIND durch Aggregationen von UIOs. Verhältnismäßig viel Wert legen die Autoren noch darauf, dass im Interface-Modell nicht mit Tabellen, oder anderweitig verschachtelten horizontalen und vertikalen Boxen, gearbeitet wird, sondern mit Gitterstrukturen und Ausrichtungshilfslinien. Für die präsentierten Beispiele scheint das jedoch nicht besonders relevant zu sein, d.h. diese liessen sich auch mit der Tabellentechnik umsezten.

Die Herstellung der Modellzusammenhänge wurde im ursprünglichen MASTERMIND-Toolkit an die den Modell-Server nutzenden Werkzeuge ausgelagert und nicht direkt betrachtet. In den späteren Veröffentlichungen z.B. in [SR98] aus dem MASTERMIND-Umfeld wurde dann das Aufgabenmodell nicht mehr betrachtet. Stattdessen wurde der Fokus wurde auf die Ablaufsteuerung der Benutzungsoberfläche gelegt. Hierzu wurden Dialogmodelle, konzeptionell vergleichbar den in Unterabschnitt 2.1.2 vorgestellten, in den Ansatz eingeführt. Über diese zusätzliche Modellart, und einer unter anderem dafür neugestalteten Runtime-Komponente, konnte dann auch der Zusammenhang zwischen Domänenmodell und UI-Beschreibung hergestellt werden.

A.3.3 FUSE

FUSE [LS96], die **F**ormal **U**ser Interface **S**pecification **E**nvironment, ist als Softwareentwicklungsumgebung konzipiert worden, in der eine Reihe von Werkzeugen zusammenspielt um Benutzungsoberflächen auf Basis formaler Spezifikationen zu generieren. Es ist eine Entwicklung der TU München aus der Mitte der 1990er-Jahre.

Auch FUSE benutzt Domänenmodell, Aufgabenmodell und Nutzermodell als Ausgangsmodelle der MB-UID. Die Kombination der Informationen aus diesen Teilmodellen ergibt jedoch zunächst kein abstraktes

Modell der Oberfläche. Eine Besonderheit des FUSE-Ansatzes ist es, zunächst Datenfluss und Abarbeitungsreihenfolge zu spezifizieren, bevor mit der Erzeugung der eigentlichen UI begonnen wird. Die dabei notwendigen Transformationen werden wesentlich durch sogenannte Design-Guidelines gesteuert. Das Transformationsergebnis kann und soll in jedem Schritt durch menschliche Nacharbeit angepasst werden. Sowohl die Modelle als auch die Guidelines werden formal, mit einer eigenen Sprache, textuell spezifiziert. FUSE generiert keinen Quellcode, sondern interpretiert seine CUI-Instanzen zur Laufzeit.

Das FUSE-Framework besteht aus vier Teilkomponenten mit unterschiedlichen Verantwortlichkeiten welche aufeinander aufbauen. Das FLUID-Subsytem, **F**orma**L** **U**ser **I**nterface **D**evelopment, spielt die Rolle eines automatischen Dialog-Designers. Durch Zusammenführen von Dialogdesign-Guidelines und den drei Basis-Modellen werden die statischen und dynamischen Eigenschaften einer Oberfläche ermittelt und als sogenannte logische Oberfläche formalisiert. Zur Formalisierung werden Hierarchic Interaction graph Templates - HIT - benutzt. HIT sind eine Kombination aus dynamischen attributierten Grammatiken und Datenflussdiagrammen, ergänzt um Ereignis- und Timing-Konzepte. Die Herangehensweise an die Modellierung ist ähnlich jener der strukturierten Analyse [DeM79]: Eine HIT-Spezifikation, und damit die logische Oberflächenspezifikation, ist ein Baum von Templates. Den Templates werden die im Aufgabenmodell identifizierten Funktionen zugeordnet. Ebenfalls hinterlegt werden die Datenflüsse zwischen den Funktionen, diese Information lässt sich aus dem Domänenmodell ableiten. Ein Template ist also zunächst einmal ein Datenflussdiagramm, allerdings bietet die HIT-Notation natürlich noch weitergehende Syntaxkonstrukte. Außerdem wird in dem Template-Baum definiert in welcher Relation die jeweiligen Untertemplates eines Knotens stehen. Letztlich bilder der Baum eine hierarchische Dekomposition der Zieloberfläche und beschreibt welche Templates wie zu einem User-Interface aggregiert werden sollen.

Die UI-Darstellung wird durch die Komponente BOSS, das **B**edien**O**berflächen-**S**pezifikations**S**ystem, durchgeführt. Layout-Guidelines nutzen die logische Oberflächenbeschreibung um für die einzelnen Funktionen ein passendes Interaktionsobjekt zu identifizieren. Dies geschieht durch Auswertung der Datenflüsse, wodurch sich Typ und Kardinalitäten der Ein- und Ausgabeparameter ergeben. Daraus kann, ähnlich der Idee bei UIDE, ein passendes Interaktionsobjekt ausgewählt werden. Durch Austauschen der relevanten Guidelines können verschiedene Oberflächen für die selbe logische Oberfläche erzeugt werden.

Die Komponente PLUG-IN,**PL**an-based **U**ser **G**uidance for **I**ntelligent **N**avigation, erstellt aus dem Aufgabenmodell und der logischen Oberfläche dynamisch Hilfeseiten, Zustandsautomaten der Anwendung und daraus wiederum animierte Durchläufe durch die Anwendung. Die vierte Komponente wird als FIRE, **F**ormal **I**nterface **R**equirements **E**ngineering, bezeichnet. Dabei handelt es sich um graphische Editoren die die Bearbeitung der sonst eigentlich textuellen Spezifikationen erleichtern.

A.3.4 TADEUS

Unter dem Namen TADEUS sind in den 1990er Jahren zwei Systeme, getrennt jedoch in Zusammenarbeit, entwickelt worden. Eines an der Universität Rostock und eines an der Johannes Kepler Univerität Linz.

Hier soll das Rostocker TADEUS [Sch93] erläutert werden, da die darin vertretenen Ideen seit langem Grundlagen der Arbeit am Lehrstuhl Softwaretechnik sind. TADEUS, **Ta**sk based **De**velopment of **U**ser interface **S**oftware, teilt den UI-Entwicklungsprozess in drei Phasen. In der Ersten Phase, der Bedarfsanalyse werden eine Zielhierarchie, Klassendiagramme und Nutzermodelle spezifiziert. Die Definition folgt dem üblichen Schema, ein Ziel wird dabei durch ein Tupel aus Teil-Aufgaben, betroffenen Rollen und zu bearbeitenden Domain-Objekten beschrieben. Es erfolgt eine hierarchische Dekomposition der Nutzerziele mit Definition expliziter temporaler Beziehungen zwischen den Zielen. Dieses Vorgehen entspricht dem der auch von ADEPT genutzten Task Knowledge Structures.

Das Domain-Modell wird in TADEUS objektorientiert per OMT[5] definiert. Die Beziehungen zwischen Tasks und Domain-Objekten werden dabei explizit modelliert.

Das User-Modell beschreibt potentielle oder existierende Nutzergruppen in den modellierten Domains. Es definiert die verfügbaren Rollen und stellt Beziehungen zwischen Rollen und Aufgaben her. Die Rollen können in Hierarchien eingeordnet sein. Beschrieben werden sie durch taskabhängige oder -unabhängigen Eigenschaften. So kann z.B. die Ausprägung der Nutzererfahrung mit interaktiven Systemen eine Rolle charakterisieren.

In der zweiten Phase des TADEUS-Ansatzes, dem Dialog-Design, wird die Zuordnung der einzelnen Tasks zu den jeweiligen Views vorgenommen. Diese Zuordnung erfolgt durch Annotationen im Aufgaben-Modell. Neben der Einteilung in eine bestimmte Sicht wird für jeden Task seine Dialogform bestimmt. Ein Task zur Dateneingabe enthält etwa die Dialogform 'data input' zugeordnet. TADEUS' Dialog-Modell verwendet die Notation der Dialog-Graphen [SE96].

Die eigentliche Interface-Generierung erfolgt in der dritten Phase. Zunächst erfolgt die Festlegung von Standardlayouteigenschaften für das ganze Projekt, z.B. Details wie Fensterhintergrundfarbe usw. Danach wird das Dialog-Modell abgearbeitet, dabei wird für jede im Dialog-Graph enthaltene View die Task-Dialogform in abstrakte Interaktionsobjekte (AIO) umgesetzt. Die diversen Dialogformen können Subtypen definiert haben, welche zur speziellen AIO-Umsetzung verwendet werden. Für die beispielhaft erwähnte Dialogform 'data input' existieren etwa die folgenden Untertypen-Mappings. Der Sub-Typ Free wird auf das AIO:input field abgebildet, der Subtyp 1:m als AIO:single selector (Einfachauswahl) und der Sub-Typ n:m als AIO:multiple selector (Mehrfachauswahl) in die abstrakte Oberflächenspezifikation übernommen.

Der letzte Schritt auf dem Weg zu einer Oberfläche ist die Umsetzung der AIO in konkrete Interaktionsobjekte. Die Umsetzung erfolgt unter zu Hilfenahme sogenannter Interaction Tables und kann neben 1:1 Abbildungen unter Beachtung von zusätzlichen Randbedingungen ablaufen. Randbedingungen dafür kann z.B. eine vorgegebene Maximalzahl von Elementen in einer GroupBox mit RadioButtons, bei deren Überschreitung dann auf die Darstellungsform ListBox ausgewichen wird. Der Mechanismus ist grob vergleichbar mit den Layout-Priosierungsalgorithmen von DON.

Das Ergebnis des TADEUS-Ansatzes ist eine Schnittstellenbeschreibung welche interpretiert werden muss, Anwendungslogik für eine ablauffähige Anwendung wird nicht integriert.

Das andere TADEUS wurde unter der Leitung von Stary in Linz zu TADEUS++ [Sta00] entwickelt. Außer dem Namen gibt es inzwischen kaum noch Gemeinsamkeiten der beiden Systeme.

A.3.5 TRIDENT

Das TRIDENT [Van95]-System, **T**ools fo**r** an **i**nteractive **d**evelopment **en**vironmen**t**, von 1995 entstand mit dem Fokus auf interaktive geschäftsorientierte Anwendungen. Dabei wurde darauf hingearbeitet im Designprozess eine größtmögliche Automatisierung zu erreichen. Dennoch beginnt auch bei TRIDENT die Entwicklung mit der Aufgabenanalyse und dem Aufstellen eines Domänenmodells. Das TRIDENT-Aufgabenmodell entspricht den Anforderungen der TKS-Methodik.

Besondere Beachtung erhalten bei TRIDENT diejenigen Aufgaben, die als Interaktion-Aufgaben kategorisiert werden. Solcherart werden die Aufgaben klassifiziert, die die Nutzer direkt durch Nutzung des zu erstellenden Systems ausführen. Sie werden mittels sogenannter Aktivitätsverkettungsgraphen, bzw. activity chaining graph (ACG), spezifiziert. Ein solcher ACG beschreibt dabei den Informationsfluß, also die Ein- und Ausgabedaten, zwischen den, im Domänenmodell definierten, Funktionen der Anwendung.

[5] Object Modeling Technique, inzwischen in UML aufgegangen

Die im ACG verketteten Funktionen sind die Operationen deren Ausführung, gemäß der Semantik des Aufgabenmodells, zur Zielerreichung nötig ist. Auch hier bietet sich wiederum der Vergleich zu den Datenflussdiagrammen, ähnlich dem Vorgehehn bei der FUSE-Methodik, an.

Zur Erzeugung einer Benutzungsoberfläche nutzt auch TRIDENT das Konzept der Interaktionsobjekte. Es werden konkrete und abstrakte Interaktionsobjekte (AIO/CIO[6]) unterschieden. Das CIO-Modell TRIDENTs definiert reale UI-Komponenten wie Labels, Buttons oder diverse Widgets.

Ausgehend von der Aktivitätsverkettung im ACG werden Darstellungseinheiten (PU)[7] identifiziert. Eine Darstellungseinheit ist eine Menge von gleichzeitig darzustellenden Aufgaben. Ein Expertensystem weist den Aufgaben geeignete abstrakten Interaktionsobjekte zu. Zur Laufzeit werden die AIOs durch CIOs der Ausführungsplattform dargestellt.

Das TRIDENT-System ist darüber hinaus auch ein dreigliederiges Architekturmodell. Dieses besteht aus Interaktions-, Steuerungs- und Anwendungsobjekten und somit aus den gleichen Teilen wie das bekannte MVC[8]-Pattern. Im Gegensatz zu diesem wird jedoch der Fokus auf die Hierarchie der Steuerungsobjekte und deren Verantwortlichkeiten gelegt, während die Interaktions- und Anwendungs(modell)objekte nur auf der Blattebene der CIOs eine Rolle spielen.

A.3.6 L-CID

Ein regelmäßig wiederkehrendes Problem der bis hierher vorgestellten Systeme ist es, durch Benutzung von Heuristiken, Algorithmen oder Expertenwissen einer Aufgabe oder Funktion eine passende Darstellungsart zuzuordnen. Ein Ansatz zur Lösung dieses Zuordnungsproblems ist es, Methoden der Künstlichen Intelligenz darauf anzuwenden. L-CID [Pue91] aus dem Jahre 1991 macht den Versuch die Technik des machinellen Lernens (machine learning) zur Ermittlung optimierter Lösungen einzusetzen. L-CID generiert seine Interface-Vorschläge aus drei Basismodellen; es nutzt Aufgaben-, Domänen- und Nutzermodell.

Verwendet wurde ein expertengestützter Lernansatz[9]. Sämtliche verfügbaren Präsentationstechniken wurden in einem Pool zur Auswahl gestellt und das System wählte mit Hilfe diverser Heuristiken eine passende Präsentationstechnik für die anstehende Aufgabe aus. Passte keine Heuristik, z.B. für eine bis dato unbekannte Aufgabe, war die Auswahl zufällig. Menschliche Experten bewerten diese getroffene Auswahl und L-CID nutzt diese Bewertung, um die Auswahl beim nächsten Auftreten dieses Tasks zu optimieren. Der Schwerpunkt bei L-CID ist insgesamt nicht auf einen Interface-Generator gelegt, sondern auf die Erstellung einer hinreichend großen Wissensbasis für die Zuordnung sinnvoller Visualisierungen bzw. Visualisierungstechniken zu Aufgaben.

A.3.7 MECANO, MOBI-D

Das MECANO-Projekt [Pue96] der Universität Stanford von 1996 ist ein weiterer Versuch, geeignete Modelle, Modellierungssprachen und eine Laufzeitumgebung für modellbasierte Oberflächenentwicklungen zu entwickeln. Es wurden die wesentlichen Ideen der bis dato bekannten Vorgänger zusammen gefasst und eine generische Sprache (MIMIC[10]) zur Spezifikation von Interface-Modellen entwickelt. MIMIC ist eine textuelle domain-spezifische Sprache (tDSL), deren Grammatik in der BNF-Notation veröffentlicht wurde.

[6] Concrete Interaction Object
[7] Presentation Units
[8] Model View Controller
[9] entsprechend einem Blackboard-System
[10] MECANO Interface Language

Das MECANO-Projekt stellt die zu erstellenden Interfaces in den Mittelpunkt. Jede dieser Schnittstellen der Mensch-Maschine-Kommunikation wird durch Instanzen von sechs Modellarten definiert. Dies sind Aufgaben-, Präsentation-, Dialog-, Design-, Nutzer- und Domänenmodell. Es ist jedoch nicht zwingend notwendig alle diese Teilmodelle für ein Interface zu instanziieren. Die Instanzen der Teilmodelle und deren Zusammenhänge werden mit MIMIC beschrieben. Eine Besonderheit der MECANO-Teilmodelle ist, dass in allen eine hierarchische Ordnung vorgenommen wird. Allgemein üblich ist dies nur für Aufgabenmodelle.

Neben den damals bereits üblichen Angaben in einem hierarchischen Aufgabenmodell wurde bei MECANO von Anfang an vorgesehen, explizite Vor- und Nachbedingungen für die Durchführung einer Aufgabe zu notieren. Das Domänen-Modell definiert nur Typ und Relationen der Objekte der Anwendungsdomäne, deren etwaiige Methoden werden nicht mitmodelliert. Mittels des Präsentationsmodells werden die zu Verfügung stehenden Interaktionskomponenten und im Nutzermodell die Systemnutzer definiert. Die Metamodelle beider Modellarten sind isomorph dem des Domänenmodells. Die Interface-Komponenten werden daher wie beliebige Objekte beschrieben, im Gegensatz zu anderen Ansätzen findet keine Festlegung auf eine bestimmte Interface-Modalität statt. Das Nutzermodell ist in vielen anderen Ansätzen ohnehin unterspezifiziert; der MECANO-Gedanke Nutzer als methodenlose Objekte zu behandeln scheint eine gangbare Formalisierung zu sein.

Die Interaktion wird bei MECANO durch Kommandos gesteuert. Diese werden im Dialog-Modell definiert. im Gegensatz zur typischen Verwendung dieses Begriffes, werden in diesem Modell jedoch nicht die Transitionen zwischen verschiedenen Dialogen einer Anwendung beschrieben. Kommandos sind Aktionen die ein bestimmtes Nutzerziel, aus dem Aufgabenmodell, erfüllen und deren Ausführung Konsequenzen für die Anwendungsobjekte hat. Diese Auswirkungen werden über Vor- und Nachbedingungen oder auch explizite Wertsetzungen modelliert. Kommandos können, da sie hierarchisch organisiert sind, verschachtelt werden.

Das Design-Modell stellt die Assoziationen zwischen den Teilmodellen. So wird hier z.B. der Zusammenhang zwischen Aufgaben und Domänenobjekten hergestellt. Ebenfalls wird entschieden, mit welchen Präsentationskomponenten die Kommandos bzw. Aufgaben visualisiert werden sollen.

Zur Interpretation und Darstellung der Modellierungsergebnisse kommt MOBI-D [11] [PM97] zum Einsatz. Es wurde im Rahmen desselben Projektes entwickelt. MOBI-D dient zur Interpretation der MIM-Schnittstellenbeschreibungen. Dazu war zwingend eine Festlegung von Interface-Modalität und Zielplattform notwendig. Die Entscheidung fiel, wenig überraschend auf Standard-WIMP Oberflächen. Zur Implementierung wurden C++ Toolkits benutzt. Die mit dem MIM/MOBI-D Ansatz erstellten Benutzungsoberflächen wurden praktisch eingesetzt; mit MOBI-D sind die diverse Anwendungen im Bereich der militärischen und medizinischen Logistik erstellt worden. Ein anderes weitreichendes Ergebnis des MECANO-Projektes ist die Schnittstellenbeschreibungssprache XIML, siehe Seite 18, sie ist wesentlich von den Erkenntnissen des Projektes beeinflusst bzw. daraus hervorgegangen.

A.4 XML Derivate

A.4.1 TERESA

Viel Wert legt wird bei TERESA auf die Multi-Plattform Idee gelegt. Der Grundtenor des TERESA-Ansatzes ist: „One Model, Many Interfaces“. Für die bis dato allgemein akzeptierten User Interface-

[11] Model based interface designer

Modellierungsebenen wurden XML-Äquivalente erstellt.

Dieses „eine Modell“, sozusagen das Kernmodell, ist ein Concurrent Task Tree (CTT), ein Aufgabenmodell. Das Metamodell für CTT wurde in [MPS04], dort Abbildung 3, veröffentlicht; aktuelle Versionen sind bei [TER] verfügbar. Das im hiesigen M6C-Ansatz benutzte Metamodell ist durch die CTT-Arbeiten deutlich inspiriert, in Abbildung 2.2 wird dessen aktueller Stand vorgestellt. Auch der CTTE genannte Editor für die TERESA-Aufgabenmodelle hat beachtliche Verbreitung gefunden und wird oftmals zur Visualisierung von Aufgabenmodell benutzt.

Die endgültige Umsetzung eines Interfaces ist nicht Kern von TERESA. So existiert keine eigene CUI Sprache, sondern es werden je nach Modalität passende Sprachen als Ausgabeformat gewählt. Etwa VoiceXML für Sprachinterfaces oder diverse XHTML-Varianten für WIMP-artige GUIs. Die Erzeugung von deren XML-Dokumenten erfolgt durch mehrstufige Modell-Transformationen, ausgehend von den CTT-XML-Dokumenten.

Das TERESA-Projekt untersucht besonders Möglichkeiten zur Verschiebung von Oberflächen zwischen Geräten, sogenannte „Migratory Interfaces“. Die Prozessdarstellung in Abbildung A.2 zeigt wie sich die TERESA-Autoren diese Migration vorstellen. In der Darstellung werden die Komponenten und der Informationsfluss bei einer Migration der Benutzungsoberfläche von einer Desktop-Applikation an eine stimmenbasierte PDA-Steuerung abgebildet.

Wegen der unterschiedlichen UI-Modalität, GUI gegenüber Stimme, sind die beiden Benutzungsschnittstellen nicht Instanzen desselben abstrakten Oberflächenmodells. Aber das Aufgabenmodell ist in beiden Geräten dasselbe, es muss daher die Rolle des Fixpunktes der Zuordnung von UI-Verantwortlichkeiten übernehmen. Zur eigentlichen Migration sind sowohl die beiden Aufgabenmodellinstanzen als auch die konkreten UI-Schnittstellen zu synchronisieren.

Synchronisation bedeutet hier, sie in einen semantisch äquivalenten Status zu versetzen. Eine Aufgabe die durch Heuristiken und Algorithmen zur Statuserkennung sowie automatischen Transformationen durchgeführt werden soll. Die vorläufigen Ergebnisse von TERESA und aktuelle Arbeiten etwa des DAI-Labors [DR10] lassen jedoch vermuten, dass in diesem Bereich noch viel Arbeit offen, und der aufgaben-

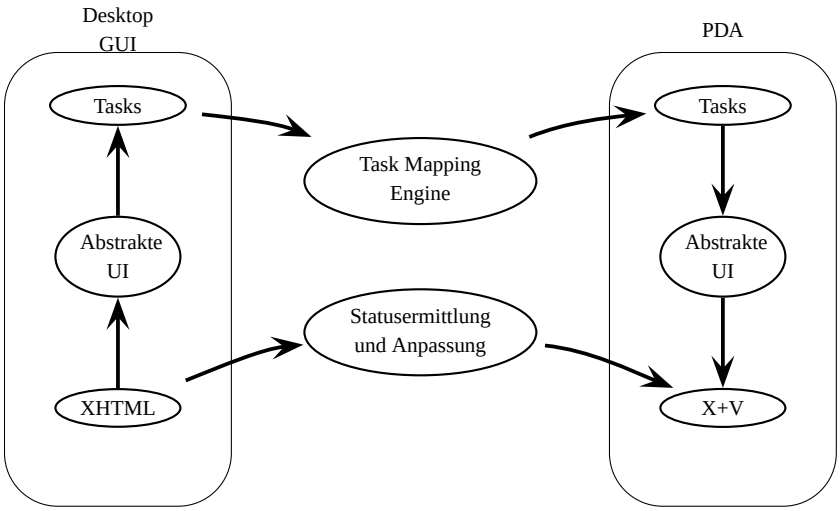

Abbildung A.2: Schrittfolge im TERESA-Migrationsprozess zur Übertragung der Benutzeroberfläche, nach [BP05]

modellzentrierte Ansatz möglicherweise nicht der fruchtbringendste ist.

A.4.2 UIML

UIML [APB+99], von 1999, ist der Urahn der Forschungssprachen zur deklarativen Oberflächenbeschreibung mit XML. Die Autoren sehen UIML als eine:

> [..] solution [..] to build interfaces with a single, universal language free of assumptions about appliances and interface technology.

In diesem Selbstverständnis ist UIML eine, an keine Interface-Modalität oder Geräteklasse gebundene, Universalsprache.

Der Fokus der UIML-Entwicklung lag stets auf Oberflächen für WWW-Anwendungen. So lag bereits seiner Entstehung die Beobachtung zugrunde, das Anzahl und Diversität der für Internet-Zugang zur Verfügung stehenden Geräte „explodierenden“ [APB+99]. Viele der um die Jahrtausendwende vermarkteten Gerätschaften und Technologien wurden über proprietäre Sprachen gesteuert. Aus der Sicht der UIML-Entwickler war das der „Turmbau zu Babel“ im Bereich der Benutzungsschnittstellen. Und ihre neuentwickelte Sprache die Lösung dieses Problems.

Die Kernidee von UIML ist es, Benutzungsschnittstellen generisch zu beschreiben und die Anpassung an den tatsächlichen Nutzungskontext in Style-Definitionen auszulagern. Auf diesem Weg kann die Ansteuerung der UI-Komponenten plattformunabhängig erfolgen und der weitgehend der gleiche Quellcode für Schnittstellendeklaration und deren Datenanbindung genutzt werden.

Eine deklarative Oberflächenbeschreibung mit UIML besteht aus sechs Teilen, Abbildung A.3 stellt diese dar. Der generische Kern der UIML-Beschreibung erfolgt in der Schnittstelle. Im Strukturmodell werden die Teile deklariert aus welchen sich das User Interface zusammensetzt. Dieser Teil entspricht in der Idee einer abstrakten Oberfläche im CAMELEON-Sinn. Das Gestaltungsteilmodell definiert Anzeigeeigenschaften für die im Strukturmodell definierten UI-Teile. Hier erfolgen gemäß [UIMa] Angaben zu Farben oder Schriftarten und anderen geräteunabhängig definierbaren Eigenschaften. In den meisten Fällen werden die Informationen in diesem Modell implizit die Ausgabemodalität fixieren; Festlegungen zur Schriftfarbe sind etwa bei Sprachinterfaces gegenstandslos. Den UI-Teilen kann bereits auf der Ebene der UIML-Deklaration ein darzustellender Inhalt zugewiesen sein. Ein Beispiel der Spezifikation [UIMa] nutzt diesen Mechanismus um Schaltflächenbeschriftungen zu internationalisieren. Prinzipiell können den UI-Teilen aber auch komplexere Inhalte, wie Bilder, Filme oder Sprachdateien, zugeordnet werden.

Das Verhalten einer Benutzungsschnittstelle kann vergleichsweise weitreichend innerhalb eines UIML-Dokumentes definiert werden. Über das integrierte Event-Konzept in Verbindung mit Schaltbedingungen,

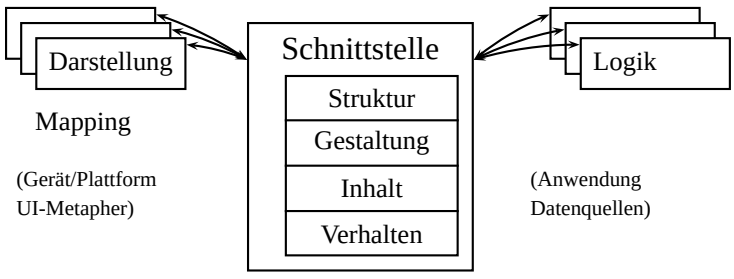

Abbildung A.3: UIML – Teilmodelle und Zusammenhänge nach [APB+99]

Regeln und Aktionsdefinitionen können einfache Algorithmen umgesetzt werden. Selbstverständlich handelt es sich bei UIML dennoch nicht um eine Programmiersprache. In diesen Algorithmendeklarationen werden Funktionen referenziert deren Implementierung im Logik-Teilmodell auf real existierende Anwendungslogik der Nutzungsumgebung abgebildet wird.

Das Mapping der deklarierten UI-Teile auf die Widgets einer Zielplattform wird schließlich im Präsentationsmodell beschrieben. Diese Art Abbildungen werden bei UIML als „Vocabulary“ , also etwa Wortschatz, bezeichnet. Einmal definierte Vocabularies lassen sich verhältnismäßig einfach als Standardmappings für die Zielplattform wiederverwenden. Für die Sprachversion 3.1 stehen bei [APB+99] elf Vocabularies zur Verwendung bereit. Sie ermöglichen die Verwendung von UIML zur Deklaration von Oberflächen mit folgenden FUIs: HTML mit CSS, WML, Java-AWT und Java-Swing, VoiceXML sowie einer GenericJH genannten Kombination aus Java und HTML.

UIML konnte sich letztlich nicht durchsetzen. Über die Gründe lässt sich nur spekulieren. Beigetragen dazu hat sicherlich der Fakt, dass die Qualität der vorhandenen UIML-Renderer der eigentlichen Mächtigkeit der Sprache nicht gerecht wurde. Die Art und Weise die Mappings zu definieren, als zwar austauschbare aber dennoch 1:1-Abbildungen, limitiert die Einsetzbarkeit weiter. Dedizierte Transformationen mit dafür geeigneten Model-To-Model und Model-To-Text Sprachen scheinen eine bessere Steuerbarkeit und im Endeffekt eine qualitativ bessere Benutzungsoberfläche zu ermöglichen.

A.5 Sonstiges

A.5.1 CSS

CSS, ist eine standardisierte [CSS] Sprache zur Festlegung von Designeigenschaften über Schlüssel-Wert Beziehungen. Im CSS-Standard werden die Schlüsselnamen und der mögliche Wertebereich definiert. Die Verfügbarkeit einzelner standardisierter Schlüssel ist abhängig von der eingesetzten Sprachversion.

Ausdrücke der CSS bestehen aus einem Selektor-Ausdruck, zur Identifikation und Auswahl der auszuzeichnenden Elemente siehe dazu Tabelle A.2, und einer Auflistung der Eigenschaften welche den so ausgewählten Elemente zugeordnet werden. Die Selektor-Ausdrücke erwarten implizit eine Baumstruktur der Gastsprache, mit der Möglichkeit an den einzelnen Knoten benannte Eigenschaften zu notieren. Mit anderen Worten eine XML-Struktur. Listing A.3 zeigt im oberen Teil die Grundstruktur eines CSS-Ausdruckes. Innerhalb eines CSS-Dokumentes können beliebig vieler solcher Blöcke definiert sein. Der Zweite CSS-Block in Listing A.3 definiert einen Selektor für alle Kindelemente eines Opel-Elementes dessen direkte Kinder ein motor-Attribut mit dem Wert "benzin" aufweisen. Im anschließenden XML-Beispiel würde damit der Knoten Omega selektiert werden, und für seine visuelle Darstellung die Eigenschaft Textfarbe:Rot gesetzt werden.

```
Selektormuster {
  Eigenschaft 1: Wert für Eigenschaft 1;
  Eigenschaft 2: Wert für Eigenschaft 2;
  ...
  Eigenschaft n: Wert für Eigenschaft n;
}

Opel *[motor="Benzin"] {
  color: red;
}
```

```
<Autos>
  <Daewoo><Lanos motor="Benzin">farblos</Lanos></Daewoo>
  <Opel>
    <Omega motor="Benzin">Dieser Text wird Rot</Omega>
    <Insignia motor="Diesel">ebenfalls keine explizite Farbzuweisung</Insignia>
  </Opel>
</Autos>
```

Listing A.3: Syntaxstruktur und beispielhafte Verwendung von CSS

A.5.2 JavaScript

Bei JavaScript, inzwischen als Derivat des als ECMA-Standard 262 spezifizierten ECMAScript [ECM] klassifiziert, handelte es sich ursprünglich um eine schwach getypte objekt-orientierte Skriptsprache. Der Einsatzzweck bei seiner Inkarnation um 1996, Interaktion auf sonst statischen HTML-Webseiten, ist auch heute noch die dominante Verwendung. Im Laufe seiner Evolution wurde JavaScript unter anderem um funktionale Aspekte erweitert, zusätzlich ist ebenfalls die Programmierung nach dem imparativen Paradigma möglich. Seine Syntax ist an diejenige von C angelehnt, es finden sich jedoch auch Einflüsse anderer Sprachen, beispielsweise aus Pascal die Schlüsselworte **function** zur Deklaration von Sub-Routinen oder **var** zur Deklaration und Initialisierung von Variablen.

A.5.3 Einbettung CSS, Javascript und XUL

Der Quellcode aus Listing A.4 zeigt die Verwendung von CSS und JavaScript in einem kleinen XUL-Beispiel. Ein Renderer erstellt aus dieser UI-Beschreibung ein Fenster mit einer Schaltfläche. Diese ist mit einem blauen Schriftzug „Hallo“ bezeichnet. Sobald diese Schaltfläche angetippt wird, entsteht ein Click-Event welcher im onclick-Eventhandler abgefangen wird. Die Abarbeitung des Eventhandlers lässt ein Hinweisfenster mit dem Text „Welt“ erscheinen. Bei Listing A.4 handelt es sich also um ein typisches „Hallo Welt“ Einführungsbeispiel.

```
<window xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">
```

Selektor-Pattern	Bedeutung
*	jedes Element
E	jedes Element vom Typ E
E F	jedes Element vom Tyo F unterhalb eines Element vom Typ E
E > F	jedes Element vom Typ F das ein Kindknoten eines Elementes vom Typ E ist
E:<Pseudoklasse>	jedes Element vom Typ E das die Bedingungen der Pseudoklasse erfüllt, z.B.:
E:first-child	jedes Element vom Typ E, dass das 1. Kind seines Vaterknoten ist
E:focus	jedes Element vom Typ E, dass während div. Nutzeraktionen fokussiert ist
E:lang(c)	jedes Element vom Typ E in der natürlichen Sprache c
F + E	jedes Element E dessen Vorgänger ein Element vom Typ F ist
E[foo]	jedes E dessen Attribut foo gesetzt ist
E[foo="bar"]	jedes Element E dessen Attribut "fooäuf den Wert "bar"gesetzt ist
E#foobar	jedes Element vom Typ E mit der ID "foobar", Syntaxabkürzung für [id="foobar"]

Tabelle A.2: Syntaxelemente für Selektorausdrücke in CSS, nach [CSS]

```
  <button label="Hallo" style="color:blue" onclick="javascript:alert('Welt')" />
</window>
```

Listing A.4: XUL-Minimalbeispiel mit CSS und JavaScript

Die Anwendung der JavaScript-Kommandos für die Navigationsinstruktionen bei der Generierung des Mockup-Prototypen aus dem Dialoggraph, siehe Unterabschnitt 2.1.2, ist ein weiteres Beispiel für die Einbettung externer Sprachen in XUL.

Anhang B

Quelltexte

B.1 PLML

B.1.1 Grammatik für die Sprachversion 1.1

```
<!ELEMENT catalog (pattern*)>

<!ELEMENT pattern (name?, alias*, illustration?, problem?, context?, forces?, solution?, synopsis?, diagram?,
      evidence?, confidence?, literature?, implementation?, related-patterns?, pattern-link*, management?)>
<!ATTLIST pattern
  patternID CDATA #REQUIRED
>

<!ELEMENT name (#PCDATA)>
<!ELEMENT alias (#PCDATA)>
<!ELEMENT illustration ANY>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT context ANY>
<!ELEMENT forces ANY>
<!ELEMENT solution ANY>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT diagram ANY>
<!ELEMENT evidence (example*, rationale?)>
<!ELEMENT example ANY>
<!ELEMENT rationale ANY>
<!ELEMENT confidence (#PCDATA)>
<!ELEMENT literature ANY>
<!ELEMENT implementation ANY>
<!ELEMENT related-patterns ANY>

<!ELEMENT pattern-link EMPTY>
<!ATTLIST pattern-link
  type CDATA #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #REQUIRED
```

```
  label CDATA #REQUIRED
>

<!ELEMENT management (author?, credits?, creation-date?, last-modified?, revision-number?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT creation-date (#PCDATA)>
<!ELEMENT last-modified (#PCDATA)>
<!ELEMENT revision-number (#PCDATA)>
```

Listing B.1: PLML DTD, Sprachversion 1.1

B.1.2 Grammatik der Sprachversion 1.5

```
<!ELEMENT catalog (name?, management, category*, pattern+)>
<!ATTLIST catalog
  id CDATA #REQUIRED
>

<!ELEMENT pattern (name?, alias*, illustration?, problem?, context?, forces?, solution?, synopsis?, diagram?,
      evidence?, confidence?, literature?, implementation?, related-patterns?, management?)>
<!ATTLIST pattern
  patternID CDATA #REQUIRED
>
<!ELEMENT alias EMPTY>
<!ATTLIST alias
  id CDATA #REQUIRED
  collection CDATA #IMPLIED
>
<!ELEMENT name (#PCDATA)>

<!ELEMENT category (name,category*)>
<!ATTLIST category
  id CDATA #REQUIRED
>
<!ELEMENT illustration ANY>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT context ANY>
<!ATTLIST context
  url CDATA #IMPLIED
>

<!ELEMENT forces ANY>
<!ELEMENT solution ANY>
<!ELEMENT synopsis (#PCDATA)>
<!ELEMENT diagram ANY>
<!ATTLIST diagram
  url CDATA #IMPLIED
>

<!ELEMENT evidence (example*, rationale?)>
<!ELEMENT example ANY>
<!ATTLIST example
  url CDATA #IMPLIED
>

<!ELEMENT rationale ANY>

<!ENTITY % confidenceLevels "(0 | * | **)">
<!ELEMENT confidence EMPTY>
<!ATTLIST confidence
  level %confidenceLevels; #REQUIRED
>
```

```
<!ENTITY % literatureReferenceTypes "(bibtex|url)">
<!ELEMENT literature (literatureEntry*)>
<!ELEMENT literatureEntry (#PCDATA)>
<!ATTLIST literatureEntry
  refType %literatureReferenceTypes; #REQUIRED
>

<!ELEMENT implementation ANY>
<!ELEMENT related-patterns (pattern-link+)>

<!ENTITY % linkType "(is-alis-contained-by|contains|uses|comparable|contradicts)">
<!ELEMENT pattern-link EMPTY>
<!ATTLIST pattern-link
  type %linkType; #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #IMPLIED
  label CDATA #IMPLIED
>
<!ELEMENT management (author, credits?, creation-date, last-modified?, revision-number)>
<!ATTLIST management
  categoryId CDATA #IMPLIED
>
<!ELEMENT author (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT creation-date EMPTY>
<!ATTLIST creation-date
  date CDATA #REQUIRED
>
<!ELEMENT label (#PCDATA)>
<!ELEMENT last-modified EMPTY>
<!ATTLIST last-modified
  date CDATA #REQUIRED
>
<!ELEMENT revision-number EMPTY>
<!ATTLIST revision-number
  rev CDATA #REQUIRED
>
```

Listing B.2: PLML DTD, Sprachversion 1.5

B.1.3 xText-Grammatik für PLML 1.5

```
grammar de.unirostock.informatik.swt.patterns.PLML with org.eclipse.xtext.common.Terminals

generate plml "http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/andreas/uri/plml/1/"

Model: catalog=CatalogDef (categories+=CategoryDef)* (pattern+=Pattern)*;

PatternDescription:
  confidence=Confidence
  evidence=Evidence
  problem=Problem
  relations=RelatedPatterns
  (alias+=Alias)*
  (_context=Context)? //Umbenannt, da sonst Namenskollision mit OCL-Syntax bei M2T mit Acceleo
  (diagram=Diagram)?
  (forces=Forces)?
  (illustration=Illustration)?
  (implementation=Implementation)?
  (literature=Literature)?
  (solution=Solution)?
  (synopsis=Synopsis)?;

MasterData: "MasterData" "{"
  author=Author
  ("Category" category=[CategoryDef|(PATTERN_ID)])?
  (credits=Credits)?
  creationDate=CreationDate
  (lastModified=LastModifiedDate)?
  revision=Revision
"}";

CategoryDef: "CategoryDef" "{"
  name=PATTERN_ID //Attribut heisst name, um keinen Extraaufwand für Referenzierungen zu haben
  (hasLabel?="label=" label=STRING)?
  (subCategory+=CategoryDef)*
"}";

Alias: "Alias" aliasId=STRING (hasCollection?="collection=" collection=PATTERN_ID)?;
Author: "Author" text=STRING;
CatalogDef: "Catalog" name=PATTERN_ID (label=STRING)? masterData=MasterData;
Confidence: "Confidence" (zero?="0"|one?="*"|two?="**");
Context: "Context" (hasURL?="url=" url=URL)? (text=STRING)?;
CreationDate: "CreationDate" date=DATE;
Credits: "Credits" text=STRING;
Diagram: "Diagram" (hasURL?="url=" url=URL)? (text=STRING)?;
Evidence: "Evidence" "{" (examples+=Example)* (rationale=Rationale)? "}";
Example: "Example" (description=STRING)? (hasURL?="url=" url=URL)?;
Forces: "Forces" text=STRING;
Illustration: "Illustration" (hasURL?="url=" url=URL)? text=STRING;
Implementation: "Implementation" text=STRING;
```

```
LastModifiedDate: "LastModified" date=DATE;
Literature: "Literature" "{" references+=Reference+ "}";
Pattern: "Pattern" name=PATTERN_ID (label=STRING)?
  "{" masterData=MasterData desc=PatternDescription "}"; //name statt id, wegen Referenzierung
PatternLink: type=PatternLinkType
  (("name=" targetId=PATTERN_ID (hasCollection?="collection=" collection=PATTERN_ID)?)
  |patternRef=[Pattern|(PATTERN_ID)]) (hasLabel?="label=" label=STRING)?;
Problem: "Problem" text=STRING;
Rationale: "Rationale" text=STRING;
Reference: "Reference" ((hasURL?="url=" url=URL) | (hasBibtex?="bibtex=" "{" bibtex=STRING "}"));
RelatedPatterns: "RelatedPattern" "{" (references+=PatternLink)* "}";
Revision: "Revision" rev=VERSION_TYPE;
Solution: "Solution" text=STRING;
Synopsis: "Synopsis" text=STRING;

//Typen:
enum PatternLinkType: uses="uses" | containedIn="is-contained-by" | contains="contains" | isA="is-a" |
      comparable="comparable" | contradicts="contradicts";

terminal VERSION_TYPE: ('0'..'9')+('.''0'..'9')*;
terminal DATE: (('2''0'..'9''0'..'9''0'..'9')-(('0''1'..'9')|('1''1'..'2'))-(((1'|'2')'0'..'9')|('0'('1'..'9'))
      |('3'('0'..'1'))));
terminal URL: (('http'|'file'|'ftp')'://'(('a'..'z'|'A'..'Z'|'0'..'9'|':'|'/'|'.'|'@'|'-'|'+'|'%'|'_'|'#')*));
terminal PATTERN_ID: ('a'..'z'|'A'..'Z'|'0'..'9'|':'|'.'|'-'|'_')+;
```

Listing B.3: PLML xText Grammatik, Sprachversion 1.5

B.1.4 Beispielkatalog

```
Catalog welie "Welie's Pattern"
  MasterData {
    Author "Andreas Wolff"
    Credits "Sammlung von www.welie.com/patterns -- automatisch transformiert"
    CreationDate 2009-06-15
    LastModified 2010-03-15
    Revision 0.9.0
  }
CategoryDef { UserNeeds label="User needs"
  CategoryDef { NavigatingAround label="Navigating around" }
  CategoryDef { BasicInteractions label="Basic interactions" }
  CategoryDef { Searching label="Searching" }
  CategoryDef { DealingWithData label="Dealing with data" }
  CategoryDef { Personalizing label="Personalizing" }
  CategoryDef { Shopping label="Shopping" }
  CategoryDef { MakingChoices label="Making choices" }
  CategoryDef { GivingInput label="Giving input" }
  CategoryDef { Miscelleaneous label="Miscelleaneous" }
}
CategoryDef { ApplicationNeeds label="Application needs"
  CategoryDef { DrawingAttention label="Drawing attention" }
  CategoryDef { Feedback label="Feedback" }
  CategoryDef { SimplifyingInteraction label="Simplifying interaction" }
}
CategoryDef { ContextOfDesign label="Context of design"
  CategoryDef { SiteTypes label="Site types" }
  CategoryDef { Experiences label="Experiences" }
  CategoryDef { PageTypes label="Page types" }
}

Pattern details-on-demand "" {
  MasterData {
    Author "Martijn van Welie"
    Category DealingWithData
    CreationDate 2010-03-15
    LastModified 2010-03-15
    Revision 1.0
  }
  Confidence **
  Evidence {
    Example url=http://www.welie.com/patterns/#
    Example "<img src=\"#\" ></img>"
    Rationale ""
  }
  Problem ""
  RelatedPattern {
    uses retractable-menu
  }
  Context ""
```

```
  Solution "Isn't even a fly-out menu an instance of this pattern? sort of a <div patternID=\"retractable-
      menu\" class=\"pattern-link\">retractable-menu</div>"
  Synopsis ""
}

 Pattern doormat "Doormat" {
  MasterData {
    Author "Martijn van Welie"
    Category NavigatingAround
    CreationDate 2010-03-15
    LastModified 2010-03-15
    Revision 1.0
  }
  Confidence **
  Evidence {
    Example url=http://www.welie.com/patterns/images/doormat-att-small.png
    Example "<img src=\"images/doormat-att-small.png\" border=\"1\" ></img>"
    Rationale "The doormat give users a quick and informative overview of the primary choices they need to
        make. Nothing important is hidden and all options in the doormat are accessible in one click."
  }
  Problem "Users need to be directed to the right section of the website"
  RelatedPattern {
    uses corporate
    uses homepage
    uses information-experience
  }
  Context "You are designing the<div patternID=\"homepage\" class=\"pattern-link\" >homepage</div>of
      an information rich website. Typically a<div patternID=\"corporate\" class=\"pattern-link\" >
      corporate</div>or an<div patternID=\"information-experience\" class=\"pattern-link\" >
      information-experience</div>."
  Solution "Divide your site into very few main sections. Three or four main sections is preferable because the
      clean and informative effect of the doormat is otherwise rapidly diminishing. Place the labels of each
      main section above a list of sub-items of that section. The sub-items must also not be too numerous,
      typically 4-8 items."
  Synopsis "List the main categories with the elements in the center of the home-page"
}
```

Listing B.4: Beispielkatalog mit der PLML-tDSL

B.2 XUL

```
<window orient="vertical" width="400" height="250">
  <spacer flex="9" />
  <hbox flex="52">
    <spacer flex="21" />
    <vbox flex="135"> <spacer flex="3" /> <button flex="49" label="B 5" /> </vbox>
    <spacer flex="100" />
    <vbox flex="92"> <button flex="39" label="B 1" /> <spacer flex="12" /> </vbox>
    <spacer flex="52" />
  </hbox>
  <spacer flex="27" />
  <hbox flex="65">
    <spacer flex="104" /> <button flex="127" label="B 2" /> <spacer flex="169" />
  </hbox>
  <spacer flex="12" />
  <hbox flex="74">
    <spacer flex="30" />
    <button flex="74" label="B 4" />
    <spacer flex="134" />
    <vbox flex="156">
      <spacer flex="19" /> <button flex="25" label="B 3" /> <spacer flex="29" />
    </vbox>
    <spacer flex="44" />
  </hbox>
  <spacer flex="11" />
</window>
```

Listing B.5: XUL mit relativer Positionierung für das Demonstrationsbeispiel

Anhang C

Technische Beschreibungen / Implementierungsdetails

C.1 Korrektur des Ergebnis des XUL-XSLT-EMF Importers

Zwei Klassen pro XUL-Widget

Nach dem Import besteht die Definition eines XUL-Widget immer aus einer abstrakten Klasse (Namenschema: „<Widgetname>ElementType“) und einer konkreten Unterklasse dieser abstrakten Klasse, benannt nach dem Schema „<Widgetname>Type“. In der abstrakten Klasse werden die Elementbeziehungen, d.h. die Assoziationen zu anderen Widgets, festgelegt. Alle Eigenschaften die keine Assoziationen sind werden als Attribute der konkreten Klasse definiert. Im Zuge der Umarbeitung des Metamodells nach dem Import wurde ein Großhteil dieser Typ-Doppelung entfernt, indem die Typen zu einem einzigen verschmolzen wurden. Da Ecore eine semantische Trennung zwischen Assoziationen (EReference) und Attributen (EAttribute) vorsieht, bleibt diese ursprüngliche Unterscheidung auch im verschmolzenen Typ weiter erhalten. Das vom Importer vorgelegte Namensschema wurde beibehalten.

Nummerierte und anonyme Typen

Die Vielzahl der nummerierten und anonymen Typen entstanden offentsichtlich als Folge nicht erkannter Typ-Äquivalenzen, d.h. es wurden homomorphe Typdefinitionen, mit unterschiedlichen Namen, mehrfach angelegt. Durch geeignete manuelle Nachbearbeitung des XML-Schemas ließ sich dieses Problem beheben. Andere Datentypen wurden im XML-Schema nicht genauer spezifiziert, als dass es sich um Unterklassen von String handelt. Ein Beispiel hierfür ist der Typ ColorHex zur Angabe von RGB-Farben als Zeichenkette. In diesen Fällen wurde die automatisch generierte Metaklasse komplett entfernt und stattdessen das jeweilige Attribut als EString typisiert.

Weiterhin wurde vom Modellimporter für jeden Aufzählungstyp, neben der Metaklasse, noch eine eigene Objekt-Metaklasse angelegt. Beispielsweise werden die möglichen Werte für die Eigenschaft cropping, d.h. das Kürzen eines Textlabels falls der Platz zur Darstellung nicht ausreichend ist, über die Aufzählung CropAttributeType definiert[1]. Die Ecore-Erzeugungstransformation legte darüberhinaus noch die Metaklasse CropAttributeTypeObject an, welche als Pointer auf einen EMF-internen Implementierungstyp für

[1] möglich sind „start“, „end“, „center“ und „none“

Aufzählungen dient. Diese *Object-Metaklassen sind für die Verwendung als CUI-Modell irreführend und unnötig, sie wurden daher entfernt.

Mangelnde Nutzung von Vererbung

Das Transformationsergebnis der XSLT-Transformation war aus objektorientierter Sicht unbefriedigend, insbesondere die begrenzte Verwendung von Vererbung erschwert die Weiterverarbeitung. Das erstellte Metamodell definierte die meisten Attribute eines jedes XUL-Elements bei dessen jeweiliger Deklaration. Übergeordnete Typen zur Vererbung von Standardattributen wurden nicht verwendet. Ein Beispiel mag das Problem verdeutlichen: Für nahezu jedes XUL-Element kann dessen relative Größe über das Attribut flex angegeben werden. Dementsprechend wurde dieses Attribut auch in den meisten Metaklassen-Deklarationen der XUL-Elemente aufgeführt. Das hat allerdings zur Folge, dass unter dem Gesichtspunkt der Typsicherheit bzw. -hierarchie, die Eigenschaft Button.flex nicht das Gleiche wie die Eigenschaft VBox.flex ist. Beide müssten separat ausgewertet werden, die Folge wären unnütze und fehlerträchtige Mehrfacharbeiten. Eine wesentliche Strukturänderung für die Attribute schien daher unumgänglich.

Im Rahmen der Nachbearbeitung mussten sinnvolle Ober-/Unterklassenbeziehungen identifiziert und eine Vererbungshierarchie im Modell aufgebaut werden. Das XUL-Schema liefert dazu wertvolle Hinweise, genauer betrachtet, liegt im Schema bereits eine sinnvolle Klassifizierung vor. Jedoch werden dort an Stelle der Vererbung von Attributen Referenzen auf Attributgruppen vererbt, dabei handelt es sich um ein besonderes Sprachelement von Schemadefinitionen. Der EMF-Schema-Importer setzt diese Gruppen leider in der oben beschriebenen - unpraktischen und eigentlich auch semantisch abweichenden - Weise um.

C.2 Erläuterungen zum XUL-Metamodell

Zur Erklärung der Abbildung 2.16 und des Metamodells. Die Klasse XULElement bildet die Wurzel der Typhierarchie. Alle Widgetklassen werden von dieser abgeleitet. Hierin werden eine ganze Reihe der allgemeingültigen Standardattribute, wie z.B. flex siehe oben, definiert. Die Werte dieser Attribute werden mit Standardwerten vorbelegt, dass betrifft im Wesentlichen die Eigenschaften die durch Enumerations deklariert werden, oder bleiben ungesetzt.

Alle Unterklassen von TemplateControl werden genutzt um die Template-Engine zu steuern. Die Klasse InfoElement ist die Oberklasse für alle Sprachelemente die keine Widgets sind und nicht unterhalb TemplateControl einzugruppieren sind. Dazu zählen die Festlegung von Tastaturkürzeln, Popupgruppen und ähnlichem.

Die Metaklasse VisibleElementType ist die Wurzelklasse der UI-Widgets. Diese werden in zwei Gruppen unterschieden, einerseits diejenigen die andere Widgets enthalten können und andererseits diejenige für die eine solche Verschachtelung nicht möglich ist. Das Metamodell nutzt Mehrfachvererbung um zwischen diesen beiden Gruppen zu unterscheiden.

Unterklassen der abstrakten Klasse ContainerElement sind diejenigen Widgets, in welchen andere Widgets verschachtelt werden können. Für eine Reihe von UI-Elementen ist es nicht sinnvoll beliebige Kindelemente zuzulassen. Deshalb existiert weiterhin eine Unterscheidung zwischen GenericContainerElement und ContainerElement. Erstere können beliebig viele Widgets, die Unterklassen von ContainerChild sind, enthalten. Bei UI-Widgets, die nur Unterklasse von ContainerElement sind, muss die Containment-Beziehung auf der Ebene des jeweiligen Types definiert werden.

Ein Beispiel dafür ist die Klasse TabsElementType, ebenfalls in Abbildung 2.16 enthalten. Als Unterklasse

von ContainerElement kann sie, bzw. die Instanzen von TabsType, Kindelemente aufnehmen. Allerdings nur solche vom Typ TabsType. TabsType seinerseits ist ein GenericContainerElement, es kann damit beliebige ContainerChilds enthalten. Im Listing C.1 ist ein Minimalbeispiel dazu angegeben, es zeigt die vorstehend erläuterte Verschachtelung der Elemente auf XUL/XML-Ebene.

```
<tabs>
  <tab><button label="Schaltfläche im ersten Tab" /></tab>
  <tab><label value="Label im zweiten Tab" /></tab>
</tabs>
```

Listing C.1: Minimal-XUL zur Illustration der Tab-Verschachtelung

Einfache, oder auch atomare, Widgets sind direkte konkrete Unterklassen von VisibleElementType. Die Widgets welche als Kindknoten innerhalb von GenericContainerElementen auftreten, erben zusätzlich von ContainerChild. Neben der Markierung als gültiger Kindknoten, erhält das Widget auf diesem Weg auch eine Assoziation parent zum Vaterknoten.

In Abbildung 2.16 werden sieben Widgets gezeigt. Es sind dies die konkreten Klassen im unteren Bereich der Darstellung. Bei LabelType handelt es sich um eine gewöhnliche Beschriftung, ButtonType sind Schaltflächen und Instanzen von TextBoxType beschreiben Textfelder. Instanzen von RadioType werden zu $1 : N$-Auswahlen verwendet und stellen dabei jeweils eine Auswahloption dar. Diese Auswahlmöglichkeiten werden in sogenannten Radiogroups zusammengestellt, im Metamodell über den Typ RadiogroupElementType abgebildet. Da RadioType eine Unterklasse von ButtonType ist, können deren Instanzen jedoch als Kindknoten beliebiger GenericContainerElement-Instanzen verwendet werden. Das Metamodell ist damit konsistent zur, den Standard definierenden, Gecko-Engine. Inwieweit die Verwendung von RadioType-Instanzen außerhalb von RadiogroupElementType sinnvoll sein kann, ist der Entscheidung des jeweiligen Verwenders überlassen. BboxType ist ein Spezialfall eines <hbox>-Containers, also einer Box mit horizontaler Aneinanderreihung ihrer Kindknoten. In <bbox>-Container werden die Kindknoten zusätzlich noch an einer gemeinsamen Linie, der sogenannten „baseline“ ausgerichtet, daraus leitet sich auch das erste B ab →„BaselineBox“.

Einbettung in den Gesamtprozeß

Um das entstandene XUL-Metamodell als CUI für den Rostock MD-UID Prozess verwenden zu können, ergaben sich noch zwei kleine Änderungen am Modell. Zum einen musste, um weiterhin den mit jedem Widget assoziierten Task notieren zu können, der Klasse XULElement eine Referenz task:Task hinzugefügt werden. Die zweite Änderung stellt sich eher als Arbeitserleichterung dar. Es erwies sich für diverse Transformationen als praktisch, mehr als ein Fenster innerhalb einer XUL-Modellinstanz haben zu können. Der dafür eingeführte Metatyp MultiRoot stellt die entsprechenden Referenzen bereit, andere Eingriffe in das Metamodell waren dazu nicht nötig.

C.3 Pattern Language Meta Language

C.3.1 Erläuterte Änderungen gegenüber PLML 1.1

```
<!ENTITY % confidenceLevels "(0|*|**)">
<!ELEMENT confidence EMPTY>
<!ATTLIST confidence level %confidenceLevels; #REQUIRED>
```

Listing C.2: Einschränkung des Wertebereichs für confidence-Angaben

Listing C.2 zeigt die dafür nötigen Erweiterungen. Darin wird die XML-Entität confidenceLevels, die genau einen der drei möglichen Werte annehmen kann, deklariert. Diese Entität wird als Wertebereich für das Attribut level festgelegt. Gleichzeitig wird confidence als inhaltsloses Element redefiniert, d.h. es darf nun keinen Inhalt irgendeiner Art enthalten. Mit dieser Anpassung wird die Notierung von Confidence-Werten formalisiert, es bedarf keiner Interpretation eines eventuellen Textinhaltes mehr um ein Pattern einer der drei Kategorien zuzuordnen.

C.3.1.1 Abbildungsreferenzierung

Prinzipiell ist es möglich, Abbildungen als Binärdaten in XML-Dokumente einzubetten, der Standard selbst untersagt dies auch nicht. Dies wäre jedoch eine zumindest unübliche Vorgehensweise, für die kaum Vorteile zu erwarten sind. Praktischer, format- und plattformunabhängiger, ist jedoch die Einführung von Referenzen nach dem etablierten URL-Standard [rfc].

Illustrationen werden in Patternkatalogen vielfach eingesetzt; um Beispiele zu geben, den Kontext zu erläutern oder in Diagrammen die Patternlösung zu fixieren. Listing C.3 definiert optionale url Attribute für die drei entsprechenden PLML Elemente.

```
<!ATTLIST context url CDATA #IMPLIED>
<!ATTLIST diagram url CDATA #IMPLIED>
<!ATTLIST example url CDATA #IMPLIED>
```

Listing C.3: Annotation von Illustrations-URLs

C.3.1.2 Literaturreferenzen

```
<!ENTITY % literatureReferenceTypes "(bibtex|url)">
<!ELEMENT literature (literatureEntry*)>
<!ELEMENT literatureEntry (#PCDATA)>
<!ATTLIST literatureEntry
  refType %literatureReferenceTypes; #REQUIRED
>
```

Listing C.4: Notation von Literaturreferenzen

Literaturreferenzen werden gemäß der Grammatik aus Listing:PLMLLiterature notiert. Das bedeutet das zwei Notationsformen zugelassen werden, URLs und komplette Bibtex-Einträge. Beide sind jeweils als Textinhalt des literatureEntry anzugeben. Natürlich kann mit den Ausdrucksmitteln einer DTD keine Formatprüfung des Textes auf eine valide URL oder gar Bibtex erfolgen.

C.3.1.3 Patternquerverweise

Analog zum Vorgehen bei den Literaturreferenzen und der Confidence-Wertung, wurde wiederum eine XML-Entität benutzt um den gültigen Wertebereich zur Angabe des Typs der Patternbeziehungen zu fixieren.

```
<!ENTITY % linkType "(is-a|is-contained-by|contains|uses|comparable|contradicts)">
<!ELEMENT pattern-link EMPTY>
<!ATTLIST pattern-link
```

```
  type %linkType #REQUIRED
  patternID CDATA #REQUIRED
  collection CDATA #IMPLIED
  label CDATA #IMPLIED
>
```

Listing C.5: Notation von Patternverlinkungen

Das redefinierte Element pattern−link zeigt Listing C.5. Die Angabe des Zielkataloges (collection) und eines beschreibenden Labels sind im Gegensatz zum PLML-Standard nach Listing 3.1 optional. Die neue XML-Entität linkType enthält die drei, bereits in Tabelle 3.2, vorgestellten Relationsarten. Zusätzlich werden die Verbindungstypen uses, comparable und contradicts zugelassen.

C.3.1.4 Kategoriemechanismus

```
<!ELEMENT category (name,category*)>
<!ATTLIST category
  id CDATA #REQUIRED
>
```

Listing C.6: Notation von Kategorien

Es stellte sich die Frage wo die Kategorien sinnvoll zu hinterlegen sind. Ich habe mich dafür entschieden, sie in den jeweiligen Katalog einzubetten. In Listing C.6 wird das XML-Element category definiert. Jeder Kategorie muss eine eindeutige Identifikation zugewiesen werden. Das der Name der Kategorie in einem eigenen Element hinterlegt wird, ist möglicherweise unüblich; typischerweise würde man dies ebenfalls als Attribut modellieren. Da aber im PLML-Standard für Pattern-Einträge genauso vorgegangen wird, wurde die Benennung der Kategorien aus Konsistenzgründen gleich gestaltet. Jede Kategorie kann beliebig viele Unterkategorien haben, dies ist inspiriert von Welies 2-stufiger Hierarchie, und ermöglicht grundsätzlich eine beliebig tief verschachtelte Kategoriehierarchie. Die Zuordnung eines Patterns zu einer Kategorie erfolgt innerhalb dessen Stammdaten-Element (managagement), per Attribut wird die ID der (Sub-)Kategorie angegeben.

C.3.2 Erläuterungen zur xText-Grammatik

Eine xText-Grammatik ist eine kontextfreie Grammatik, deren Notation an EBNF angelehnt ist. In der Informatik wird unter einer Grammatik normalerweise das 4-Tupel (T, N, P, s_0) verstanden. T ist die Menge der Terminalsymbole, N die Menge der Nicht-Terminale, P die Produktionsregeln und s_0 das Startsymbol, meist gilt $s_0 \in N$.

In einer xText-Grammatik sind Nichtterminale Zeichenketten die mit Grossbuchstaben beginnen. Terminalsymbole sind dadurch gekennzeichnet, dass sie in Anführungszeichen stehen. Eine Produktionsregel wird durch den Namen eines Nichtterminals eingeleitet, gefolgt von einem Doppelpunkt hinter dem der Inhalt der Regel notiert wird.

```
RelatedPatterns:
  "RelatedPattern" "{" (references+=PatternLink)* "}";
PatternLink: type=PatternLinkType (
    ("name=" targetId=PATTERN_ID (hasCollection?="collection=" collection=PATTERN_ID)?)
    | patternRef=[Pattern|(PATTERN_ID)]
```

```
  )
  (hasLabel?="label=" label=STRING)?;
enum PatternLinkType:
  uses="uses" | containedIn="is-contained-by" | contains="contains" |
  isA="is-a" | comparable="comparable" | contradicts="contradicts";
```

Listing C.7: xText-Grammatikausschnitt zur Eintragung von Patternbeziehungen

Einen Ausschnitt der PLML-Grammatik zeigt Listing C.7. Um den Sinn des Listings zu erfassen, sollte man wissen, dass im aus der Grammatik generierten Metamodell alle Nichtterminale als Metaklassen angelegt werden. Diese Klassen verfügen über Attribute und Referenzen, jedoch nicht über eigene Methoden.

Der Name und der Typ der Attribute leitet sich aus der Grammatik ab. Zeichenketten die mit Kleinbuchstaben beginnen und keine Terminale sind, werden zu Attributen des Nichtterminals in dessen Produktionsregel sie notiert sind. Über den, sich an den Attributnamen, anschließenden Operator wird der Typ festgelegt: Bei $=$ entspricht der Attributtyp dem Typ der rechten Seite der Zuweisung, bei $?=$ handelt es sich einen Boolean-Wert und mit $+=$ werden Listen deklariert, genauer Listen von Elementen vom Typ der rechten Zuweisungsseite.

ID und STRING sind vordefinierte xText-Typen, für Zeichenketten und Identifikationsliterale, sie werden im Metamodell auf EString gemappt. Das xText-Schlüsselwort Enum auf der linken Seite einer Regel vor dem Nichtterminal, dient der Definition von Aufzählungstypen. Die Teilterme auf der rechten Regelseite deklarieren dann die möglichen Werte, für deren Syntax gilt: $Wertname = "Wertliteral"$. Die Bedeutung der Metasymbole $*$,? und $+$ ist äquivalent zu EBNF-Grammatiken.

Eckige Klammern [,] werden zur Deklaration von Referenzen genutzt, die Syntax ist $[Nichtterminal \mid (Typder Identifizierungseigenschaft)]$. Unglücklicherweise kann der Name der identifizierenden Eigenschaft nicht in der Grammatik angegeben werden. Standardmäßig muss dieses Attribut name heissen, alternative Namen sind grundsätzlich möglich, erfordern aber eigens implementierte Java-Klassen. Um auf diesen Aufwand zu verzichten, werden die Id-Attribute in der PLML-xText Grammatik durchgehend als name benannt. Das dies im Widerspruch zur originalen Sprachdefinition, und auch zur eigenen PLML-v1.5 steht, ist irrelvant, da die Benennung der Attribute des Metamodells keine Rolle für die Funktionalität des PLML-Editors und -Generators spielt.

Zur Erläuterung des Grammatikausschnitts aus Listing C.7. Hier wird zum einen die Struktur der Metaklasse RelatedPatterns definiert und auch die Deklaration ihrer Instanzen in der PLML-tDSL festgelegt. Demnach wird eine Instanz von RelatedPatterns im Textkörper der tDSL mit dem Schlüsselwort $RelatedPattern$ eingeleitet. Die Metaklasse enthält außerdem genau ein Attribut, eine Liste von beliebig vielen Referenzen auf Instanzen des Typs PatternLink. Diese werden zwischen öffnenden und schließenden geschweiften Klammern notiert.

Das Nichtterminal PatternLink ist komplexer. Dessen Metaklasse wird mit sieben Attributen angelegt. Die beiden Boolean-Attribute hasCollection und hasLabel dienen als Marker, ob ein Zielkatalog (collection) gesetzt oder eine Linkbeschriftung (label) gesetzt wurden. Im Attribut type wird die Art der Patternreferenz, über die Literalen des Aufzählungstyps PatternLinkType, deklariert. Eine Pattern-Referenz kann entweder durch Angabe einer beliebigen Id, ggf. in einem beliebigen Katalog, oder durch direkte Referenzierung eines Patterns im selben Katalog erfolgen.

C.3.3 PLML Metamodell für den grafischen PLML-Editor

Bei genauerer Betrachtung der, im Screenshot Abbildung 3.4 im unteren Bereich zu erkennenden, Eigenschaftentabelle fallen zwei Attribute auf, die nicht in der Metaklasse PatternLink nach Abbildung 3.2 enthalten sind, dafür fehlt das Attribut patternRef:Pattern. Die beiden hinzugefügten Attribute, source und target sind vom Typ Pattern und wurden als abgeleitete und transiente Attribute definiert. Das bedeutet, dass ihr konkreter Wert aus den Belegungen der anderen Attribute hergeleitet werden kann und der jeweilige Wert auch nicht in die Serialisierung des Modells übernommen wird. Mit anderen Worten: er wird nicht in den Katalog eingetragen. Der Wert des target-Attribute leitet sich dabei aus der Auflösung der targetID des Typs PatternLink in das konkrete referenzierte Pattern ab. Es entspricht also dem patternRef:Pattern aus dem aktuellen PLML-Metamodell.

Das Attribut source speichert das Pattern-Objekt das die Quelle der Pattern-Referenzbeziehung ist. Verwendet wird source im GMFMap-Modell. Wie erwähnt werden dort die möglichen Kanten des Graphen detailliert auf Metaklassen-Attribute abgebildet. Das source-Attribut dient dort der Speicherung des Ausgangsknotens einer Patternreferenz.

Einen weiteren erwähnenswerten Unterschied gibt es bei der Darstellung des Confidence Wertes. Im durch xText generierten Metamodell wird dieser durch drei Boolean-Attribute dargestellt, eine Konsequenz aus der Art wie diese Information in der Grammatik spezifiziert wurde. Diese Attribute sind gewissermaßen eine technische Notwendigkeit, aber im Handling unpraktisch. Um Nutzern die Sternwertung besser zu signalisieren, als durch die Boolean-Attribute, war die direkte Anzeige von deren Wertbelegung wünschenswert. Hierfür wurde wiederum ein transientes abgeleitetes Attribute label definiert, dessen Wert die Belegung der Boolean-Variablen Menschen-lesbar darstellt. Im Diagramm-Editor wird der Wert von Confidence.label daher als Beschriftung des Confidence-Wertes eingesetzt und grafisch angezeigt.

Selbstverständlich müssen die jeweiligen Werte der von target, source und label irgendwo berechnet werden. Dazu muss die Modellebene verlassen werden und eigener Quellcode implementiert werden. Nach geeigneten Anpassungen in dem Metamodell-Editierungsquellcode, der durch Nutzung des PLML-Generierungsmodells erzeugt wurde, modifiziert das Ändern der abgeleiteten Werte im Editor die zugrundeliegenden Attribute und wird der im Editor angezeigte Wert der Eigenschaften aus den zugrundeliegenden Attributen berechnet.

C.4 Technischer Ablauf der Swing-Abbildung auf Ecore

Weil es möglich ist, eine direkte Beziehung zwischen dem zukünftigen EMF-Objekt und dem effektiven Swing-Objekt herzustellen, kann der Implementierungsaufwand für den eigentlichen Mapping-Prozess reduziert werden. Durch Benutzung von Reflection bzw. Introspection kann vermieden werden, dass das Übertragen jeder einzelnen Eigenschaft ausprogrammiert werden muss. Dazu wird aus der Meta-Klasse für das jeweilige Swing-Widget die Liste von dessen potentiellen Eigenschaften ausgelesen. Diese sind im Namen identisch mit den Namen der Objekteigenschaften des Swing-Objektes. Für jeden Eigenschaften-Namen wird schließlich die Belegung des gleichbenannten Attributes per Introspection ausgelesen. Der erhaltene Wert muss dann nur noch als Eigenschaftwert in der Ecore-Instanz gesetzt werden. Theoretisch also ein Vorgang ohne wesentliche Schwierigkeiten.

Praktisch war es allerdings notwendig, einen Weg zu finden um Eigenschaftswerte an den Sichtbarkeitsbeschränkungen vorbei auszulesen. Wie bereits erwähnt, ist ein gewisser Teil der Eigenschaften als private -Attribut implementiert und nur über public-Accessor Methoden auslesbar und modifizierbar. Es wäre nun möglich gewesen, dass Swing-Metamodell so umzugestalten, dass die Zugriffsmethoden für diese Attri-

bute in das Modell übernommen werden; soweit dabei nicht die Getter/Setter-Problematik getriggert würde. Dann müsste auf Metamodellebene die Verbindung zwischen Attribut und Zugriffsmethode markiert werden, etwa über eine Annotation. Zu guter Letzt ließe sich diese Information dann vom Swing→EMF-Konverter auslesen und zum Auslesen des Wertes verwenden.

Insgesamt ein gangbarer Weg, obwohl hierbei die Sauberkeit des methodenlosen Swing-Modells reduziert würde. Ebenfalls müsste hierfür der Modellreduzierungsprozeß neu überdacht werden, denn sowohl dessen vorliegende Kermeta-Implementierung als auch die dahinterstehende Methodik berücksichtigt keine Methoden.

Java bietet im Rahmen der Reflection-Mechanismen allerdings auch die Möglichkeit den syntaktischen Schutz der Sichtbarkeiten zu durchbrechen. Voraussetzung hierzu ist das Setzen eines geeigneten Security-Managers für die Laufzeitumgebung. Diese Variante hat den Charme, dass keinerlei Sonderbehandlung für das Auslesen bestimmter Eigenschaften notwendig ist, zudem ist der Aufwand gegenüber der Alternative wesentlich geringer. Aus diesem Grund wurde daher dieser Weg verfolgt.

Ein Security-Manager ist in der Sprachumgebung Javas der Standardmechanismus um die Zulässigkeit diverser Aktionen einer Applikation zu entscheiden. Einen solchen Security-Manager zu implementieren ist recht einfach. Man muss eine eigene Unterklasse von java.lang.SecurityManager erstellen und diejenigen Methoden überschreiben, die für die gewünschten Rechte zuständig sind. Beispielsweise testet die Methode public void checkDelete(String file), ob der Java-Runtime erlaubt werden soll die Datei file zu löschen, bzw. dies überhaupt zu versuchen. Löst der Aufruf von checkDelete eine bestimmte Exception aus, dann verbietet die Java-Runtime dem Aufrufer diese Dateioperation. Endet die Methode durch einfachen Rücksprung, ist die Datei-Operation zugelassen.

Damit das Durchbrechen der Sichtbarkeitsbeschränkungen via Reflection möglich wird, muss public void checkMemberAccess(Class<?> clazz, int which) überschrieben werden. Der Parameter which bezeichnet dabei das angeforderte Sichtbarkeitslevel. Die Standardimplementierung dieser Methode setzt die bekannten Java-Sichtbarkeitsregeln um. Durch Überschreiben mit einer leeren Implementierung, werden diese Regeln ausser Kraft gesetzt. Der so entstandende eigene Security-Manager muss der Java-Runtime zum Start des Containers bekannt gegeben werden.

Zurück zum eigentlichen Ablauf, wie erwähnt, erfolgt die Transformation mit einem preorder-Durchlauf durch den Objektgraphen. Über den vollqualifizierten Namen der dabei angetroffenen Objekte wird die Meta-Klasse des anzulegenden Ecore-Objekts ermittelt. Deren Modellspezifikation wird im nächsten Schritt benutzt um die zu mappenden Eigenschaften zu identifizieren. Hierfür erfolgt eine Iteration über alle Features des Metamodell-Typs und via des oben angerissenen Reflektionsverfahrens wird versucht den Eigenschaftswert auszulesen. Erwähnenswert hierzu ist, dass auf der Suche nach einem bestimmten Eigenschaftswert auf der Javaseite, die komplette Typhierarchie bottom-up durchlaufen wird bis ein entsprechendes Attribut gefunden wird, bzw. mit dem Typ java.lang.Object die Wurzelklasse aller Typen erreicht ist. Die Notwendigkeit zu dieser Vorgehensweise ergibt sich wiederum aus den etwaigen Sichtbarkeitsbeschränkungen.

Auch im Swing-Metamodell wird zwischen Werten und Referenzen unterschieden. Werte, d.h. Instanzen der Java-Basis-typen, stellen kein Problem dar. Sie können direkt übertragen werden. Etwas aufwendiger gestaltet sich die Übernahme komplexer Eigenschaften, also solche die als Referenzen ins Modell eingetragen werden müssen. Für diese wird ebenfalls über deren vollqualizierten Namen der Abbildungstyp im Metamodell identifiziert und eine entsprechende Instanz dessen angelegt. Diese Instanz wird in einer geeigneten PropertyRegistry abgelegt. Geeignet hierfür sind Typen welche eine Kombination dieser beiden Kriterien darstellen: möglichst lokal an der Stelle der Verwendung und möglichst weit oben im Objektbaum. Querreferenzierungen sind im Objektgraphen häufig anzutreffen, beispielsweise wird die standardmäßige

Hintergrundfarbe nur einmal als ColorUIResource-Objekt angelegt, aber von allen Widgets referenziert. Um diese Referenzen im Ecore-Modell nachzuvollziehen werden während der Transformation alle erstellten Instanzen in einem Objekt-Cache eingetragen. Dieser Objekt-Cache wird immer konsultiert bevor eine neue Instanz eines Eigenschaftenobjekts angelegt wird. Liegt für einen Eigenschaftenwert ein Cachetreffer vor, wird die bereits existierende Instanz referenziert.

Im Prinzip könnte die Übernahme der Attributwerte aus den komplexen Eigenschaftswerten über das selbe Reflektionsverfahren durchgeführt werden, wie dies oben für die Ermittlung der Attribute der ursprünglichen Widgets beschrieben wurde. Dabei ist es jedoch problematisch, dass das Swing-Metamodell nicht deckungsgleich dem Java-Metamodell ist. Es kommt also zu Fällen, in denen die Attributtypen der komplexen Eigenschaftswerte nur als leere Klassen im nonlocal-Package angelegt sind. Diese lassen sich dann nicht mehr adäquat über die Metamodellinformationen auswerten. Ein anderes Problem stellen Arrays bzw. Vektoren dar, wie sie z.B. bei der RGB-Definition von Farben auftreten. Diese werden im Metamodell als EList umgesetzt, wofür wiederum eine Sonderfallbehandlung nötig ist. Daneben existieren auch noch Eigenschaften der UI-Widgets, die gar nicht als Attribute in deren Klassen hinterlegt sind. Dies gilt etwa für Tooltips, diese sind in einer Schlüssel/Wert-Tabelle abgelegt und nur über Accessormethoden auszulesen. Wegen der Vielzahl an Ausnahmebehandlungen, wurde entschieden die Wertübernahme aus komplexen Eigenschaftstypen jeweils in eigenen Java-Methoden auszuprogrammieren.

Anhang D

Patternkomponenten

D.1 Tabellarische Einordnung

Tabelle D.1: Patternkurzbeschreibungen und Komponentisierbarkeitsvermutung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
accordion	Navigating around	**
„Accordion “	1.3.1	1.3.1
Stack panels vertically or horizontally and open up one panel at the time while collapsing the other panels		
action_button	Basic interactions	**
„Action Button “	1.3.4	1.3.4
Use push-button with the action ’verb’ as part of the label.		
action_panel	Commands and Actions	**
„Action Panel “	1.3.4	1.3.4
Instead of using menus, present a large group of related actions on a UI panel that’s richly organized and always visible.		
advanced_search	Searching	**
„Advanced Search “	1.3.3	1.3.3
Offer a special advanced search function with extended term matching, scoping and output options.		
animated_transition	Getting Around	**
„Animated Transition “	1.2	1.2
Smooth out a startling or dislocating transition with an animation that makes it feel natural.		
application	Site types	**
„Web-based Application “	2.4	2.4
Structure the site around ’views’ and allow users to work inside views		
article_page	Page types	**
„Article Page “	2.4	2.4
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Present the article in a consistent and structured format, and place it in the center of the page.		
artist_sites	Site types	**
„Artist Site “	2.4	2.4
Create a site with background information, timely information and "digital merchandise".		
autocomplete	Searching	**
„Autocomplete “	1.4	1.2
Suggest possible label names as users are typing		
automotive	Site types	**
„Automotive Sites “	2.4	2.4
Offer heavily branded overview of cars and related information supporting purchasing		
blog	Page types	**
„Blog Page “	2.4	2.4
Create a page with daily news ’blobs’ and archived news blobs		
booking	Shopping	**
„Booking “	2.4	2.4
Allow users to search for the ’object’ flexibly, especially concerning date/time versus prize. Then allow that to make the actual booking.		
brand_promo_site	Site types	**
„Branded Promo Site “	2.4	2.4
Provide info and entertainment related to the brand		
campaign_site	Site types	**
„Campaign Site “	2.4	2.4
Create small thematic site that markets a product in a different way than simply listing its features		
captcha	Drawing attention	**
„Captcha “	1.2	1.2
Present users with a mangled image containing numbers and letters that humans can still ’decipher’ but is hard for machines to read.		
card_stack	Organizing the Page	**
„Card Stack “	1.1	1.1
Put sections of content onto separate panels or "cards,änd stack them up so only one is visible at a time; use tabs or other devices to give users access to them.		
cascading_lists	Showing Complex Data	**
„Cascading Lists “	1.3.2	1.3.2
Express a hierarchy by showing selectable lists of the items in each hierarchy level. Selection of any item shows that item’s children in the next list.		
case_study	Page types	**
„Case Study “	2.4	2.4
Describe a case by describing the problem, the solution and the value of that solution		
center_stage	Organizing the Page	**
„Center Stage “	2.4	2.4
Put the most important part of the UI into the largest subsection of the page or window; cluster secondary tools and content around it in smaller panels. — Create a large "center stage"that dominates on the page		
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
clear_entry_points	Getting Around	**
„Clear Entry Points “	2.4	2.4
Present only a few entry points into the interfaces; make them task-oriented and descriptive.		
color_coded_sections	Getting Around	**
„Color-Coded Sections “	2.4	2.4
Use color to identify which section of an application or site that a page belongs to. — Color each major section with it’s own color		
column_filter	Dealing with data	*
„Table Filter “	2.2	1.3.1
Allow the users to select a subset of the information items directly above the table		
combined_menu	Navigating around	**
„Header-less Menu “	1.3.3	1.3.3
Combine menus in a vertical menu using different visual clues instead of headers		
command_history	Commands and Actions	**
„Command History “	1.4	1.3.3
As the user performs actions, keep a visible record of what was done, to what, and when.		
commerce	Site types	**
„E-Commerce Site “	2.4	2.4
Create a ’virtual’ store where visitors can browse, choose and pay for all their selections in one go.		
community	Site types	**
„Community Site “	2.4	2.4
Create a simple site offering information about the topic and the group.		
community_building	Experiences	**
„Community Creation “	2.4	2.4
Create a site where users can collect, share, relate and donate		
composite_selection	Builders and Editors	**
„Composite Selection “	2.4	1.4
Use different gestures – or mouse clicks in different screen areas, such as the composite’s edges versus its insides – to determine whether you should select a composite itself or allow its contained objects to be selected.		
constrained_resize	Builders and Editors	**
„Constrained Resize “	2.4	1.4
Supply resize modes with different behavior, such as preserving aspect ratio, for use under special circumstances.		
contact_us	Page types	**
„Contact Page “	2.4	2.4
Provide a special page that tells users how to get in touch with the organization		
content_lock	Drawing attention	**
„Premium Content Lock “	1.3.1	1.3.1
Show previews of premium content and mark it visually		
corporate	Site types	**
„Corporate Site “	2.4	2.4
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Structure the site based on the primary questions of potential visitors.		
country_selector	Making choices	**
„Country Selector “	1.3.1	1.3.1
Place a language, and region, selector on the home-page.		
creating	Experiences	**
„Information Management “	2.4	2.4
Allow users to manage sets of objects using overviews and detail-views		
crumbs	Navigating around	**
„Breadcrumbs “	1.3.4	1.3.4
Show the hierarchical path from the top level to the current page and make each step clickable		
customization_window	Personalizing	*
„Customization Window “	2.4	2.4
Use "windows"with select items that users can adapt or click away.		
date_selector	Making choices	**
„Date Selector “	1.1	1.2
Use a combination of an edit box and a graphical click able calendar		
deep_background	Making It Look Good	**
„Deep Background “	1.3.3	1.3.3
Place an image or gradient into the page’s background that visually recedes behind the foreground elements.		
diagonal_balance	Organizing the Page	**
„Diagonal Balance “	2.4	1.3.4
Arrange page elements in an asymmetric fashion, but balance it by putting visual weight into both the upper-left and lower-right corners.		
directory	Navigating around	**
„Directory Navigation “	1.3.3	1.3.3
Sum up level 1 and 2		
doormat	Navigating around	**
„Doormat “	1.3.3	1.3.4
List the main categories with the elements in the center of the home-page		
edit_in_place	Builders and Editors	**
„Edit-in-Place “	2.4	2.4
Use a small, dynamic text editor to let the user change text ïn place": position the editor directly over the original text, rather than using a separate panel or dialog box.		
enlarged_clickarea	Simplifying interaction	*
„Enlarged Clickarea “	2.4	2.4
Enlarge the click area to include neighboring areas		
event_calendar	Page types	**
„Event Calendar “	2.4	2.4
Present a list of events starting from the current date and allow users to select/search for other dates		
extras_on_demand	Organizing the Content	**
„Extras On Demand “	1.3.1	1.3.1

Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Show the most important content up front, but hide the rest. Let the user reach it via a single, simple gesture.		
faq	Searching	**
„Frequently Asked Questions (FAQ) “	2.4	2.4
Create a page with Frequently Asked Questions (FAQ) and provide short answers		
$favourites$	Dealing with data	**
„Collector “	2.1	1.3.3
Allow users to build their list of items by selecting the items as they are viewing them. Place a link to the collected items list on every page in the site.		
$few_hues_many_values$	Making It Look Good	**
„Few Hues, Many Values “	2.4	2.4
Choose one, two, or at most three major color hues to use in the interface. Create a color palette by selecting assorted values (brightnesses) from within those few hues.		
$fill_in_the_blanks$	Getting Input From Users	**
„Fill-in-the-Blanks “	2.4	2.4
Arrange one or more fields in the form of a prose sentence or phrase, with the fields as "blanks"to be filled in by the user.		
fly_out_menu	Navigating around	**
„Flyout Menu “	1.2	1.2
Combine horizontal navigation with a sub-menu that flies-out when the users hovers over the main menu-item		
$font_enlarger$	Simplifying interaction	*
„Font Enlarger “	1.3.1	2.1
Allow users to increase/decrease the font size of the text using special controls in the page.		
$forgiving_format$	Getting Input From Users	**
„Forgiving Format “	2.4	2.4
Permit users to enter text in a variety of formats and syntaxes, and make the application interpret it intelligently.		
$format$	Giving input	**
„Constrained Input “	1.3.4	1.3.4
Only allow the user to enter data in the correct syntax.		
$forms$	Giving input	*
„Forms “	1.3.4	1.3.4
Offer users a form with the necessary elements		
$forum$	Page types	**
„Forum “	2.4	2.4
Create a list of topics and allow users to place comments on the topic		
fun	Experiences	0
„Fun “	2.4	2.4
Add challenging and surprising elements to your site, supported by additional visual fun-adding details, to create a highly interactive and visual experience.		
$global_navigation$	Getting Around	**
„Global Navigation “	1.3.3	1.3.3
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Using a small section of every page, show a consistent set of links or buttons that take the user to key sections of the site or application.		
good_defaults	Getting Input From Users	**
„Good Defaults “	2.4	2.4
Wherever appropriate, prefill form fields with your best guesses at the values the user wants.		
grid_based_layout	Drawing attention	**
„Grid-based layout “	1.1	1.1
Use a grid system for the placement and alignment of all visual objects on the web page		
guestbook	Page types	**
„Guestbook “	2.4	2.4
Show comments and give the users the opportunity to give comments.		
guided_tour	Basic interactions	**
„Guided Tour “	2.4	2.4
Show users how to do it in several interactive steps		
help_page	Page types	**
„Help Page “	2.4	2.4
Place a link on every page to the Help page where users find help with the most common problems		
home	Navigating around	**
„Home “	1.3.1	1.3.4
Use a fixed element, such as the site’s logo, as a link to the home page		
homepage	Page types	**
„Home-page “	2.4	2.4
Create a home-page that introduces the site to users and that helps them to get on their way on the site		
hotlist	Miscelleaneous	**
„Hotlist “	2.4	2.4
Show a hot-list of most popular items		
illustrated_choices	Getting Input From Users	**
„Illustrated Choices “	2.4	2.4
Use pictures instead of words (or in addition to them) to show available choices.		
information_experience	Experiences	*
„Information Seeking “	2.4	2.4
Primarily allow users to browse the information but combine it with more specific search tools that support other types of searching.		
inplace_replacement	Dealing with data	**
„Inplace Replacement “	2.4	2.1
When selecting an item, create more space for the item and display additional details		
input_error	Feedback	**
„Input Error Message “	2.4	2.4
Tell the users that there is a problem and how to solve the problem. Also tell the users where the problem occurred.		
input_hints	Getting Input From Users	**
„Input Hints “	1.3.3	1.3.3

Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Beside an empty text field, place a sentence or example that explains what is required.		
input_prompt	Getting Input From Users	**
„Input Prompt “	1.3.4	1.3.4
Prefill a text field or dropdown with a prompt that tells the user what to do or type.		
intriguing_branches	Organizing the Content	**
„Intriguing Branches “	2.3	2.3
Place links to interesting content in unexpected places, and label them in a way that attracts the curious user.		
jump_to_item	Showing Complex Data	**
„Jump to Item “	1.1	1.3.1
When the user types the name of an item, jump straight to that item and select it.		
language_selector	Making choices	**
„Language Selector “	1.3.1	1.3.1
Ask the language spelled in the specific language		
learning	Experiences	**
„Learning “	2.4	2.4
Construct a learning experience from the basic learning tasks that is in line with your audience and site		
liquid_layout	Organizing the Page	**
„Liquid Layout “	1.1	1.1
As the user resizes the window, resize the page contents along with it so the page is constantly "filled.– Allow certain parts of the page to scale		
list_builder	Dealing with data	**
„List Builder “	2.1	2.1
Present the total list and provide editing functionality next to it.		
list_entry_view	Dealing with data	*
„List entry “	2.1	2.1
Show a simple entry form directly above the total list of items		
login	Personalizing	**
„Login “	1.3.3	1.3.3
When needed, ask the users to login using a combination of an email-address and a password		
map_navigator	Navigating around	**
„Map Navigator “	1.2	1.2
Show a map with the points of interest and provide navigation links in all corners		
meta_navigation	Navigating around	**
„Meta Navigation “	1.3.1	1.3.1
Reserve an area on every page for communication and secondary navigation elements.		
minesweeping	Navigating around	**
„Minesweeping “	2.3	2.3
Show graphical elements that upon mouse-over reveal their meaning		
movable_panels	Organizing the Page	**
„Movable Panels “	2.4	1.1
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
Put different tools or sections of content onto separate panels, and let the user move them around to form a custom layout.		
multi_level_undo	Commands and Actions	**
„Multi-Level Undo “	1.4	1.3.3
Provide a way to easily reverse a series of actions performed by the user.		
multinational	Site types	**
„Multinational Site “	2.4	2.4
Create an proxy site that leads users to sub-sites.		
museum	Site types	**
„Museum Site “	2.4	2.4
Create an informative site focussing on the museum main collection, activities and visitor information. Complement the site by offering online ticket sales, memberships and online shop.		
my_site	Site types	**
„Personalized ’My’ Site “	2.4	2.4
Create a part of the site that belongs to a user and that is controlled by that user.		
news	Miscelleaneous	*
„What’s new? “	2.1	2.1
Add a news section to the home page that contains the most recent news headlines.		
news_site	Site types	**
„News Site “	2.4	2.4
Create a site with categorized articles that are accessible via headlines		
news_ticker	Miscelleaneous	*
„News Ticker “	1.3.1	1.3.1
Use a box with scrolling text to display the latest info and place it at the top part of the page.		
newsletter	Page types	**
„Newsletter “	2.4	2.4
Send users a newsletter regularly		
one_off_mode	Builders and Editors	**
„One-Off Mode “	2.4	2.4
When a mode is turned on, perform the operation once. Then switch back automatically into the default or previous mode.		
one_window_drilldown	Organizing the Content	**
„One-Window Drilldown “	1.3.1	1.3.1
Show each of the application’s pages within a single window. As a user drills down through a menu of options, or into an object’s details, replace the window contents completely with the new page.		
outgoing_links	Drawing attention	**
„Outgoing Links “	1.3.1	1.3.1
Mark links to external sites with an icon after the label		
overlay_menu	Navigating around	**
„Overlay Menu “	1.3.1	1.1
Present the menu at the mouse pointer location after the users click		
overview_plus_detail	Showing Complex Data	**

Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
„Overview Plus Detail “	2.1	1.1
Place an overview of the graphic next to a zoomed "detail view.Äs the user drags a viewport around the overview, show that part of the graphic in the detail view.		
paging	Basic interactions	**
„Paging “	1.3.3	1.3.3
Present the results grouped in pages with a fixed number of items and allow the users to move easily from one page of items to another		
parts_selector	Dealing with data	**
„Parts Selector “	2.1	1.3.1
Show all the parts and allow the user to add or remove a part from the selection list.		
poll	Making choices	**
„Poll “	1.3.1	1.3.1
List the statements as exclusive options and present the results directly after voting.		
portals	Site types	**
„Portal “	2.4	2.4
Create several sub-sites, one per topic, with one overall home-page		
printable_pages	Page types	**
„Printable Pages “	2.4	2.4
Place a link to a printer-friendly version of the page the users are viewing next to the page content		
progress_indicator	Commands and Actions	**
„Progress Indicator “	1.1	1.1
Show the user how much progress was made on a time-consuming operation.		
purchase_process	Shopping	**
„Purchase Process “	2.4	2.4
Present users with the purchase steps		
rating	Making choices	**
„Rating “	1.3.3	1.3.3
Present a rating next to the product and the option to rate it		
registration	Personalizing	**
„Registration “	1.3.3	1.3.3
Offer users to possibility to store their personal information for later use		
responsive_disclosure	Organizing the Page	**
„Responsive Disclosure “	1.3.2	1.3.2
responsive_enabling	Organizing the Page	**
„Responsive Enabling “	1.3.2	1.3.2
Starting with a UI that’s mostly disabled, guide a user through a series of steps by enabling more of the UI as each step is done.		
retractable_menu	Navigating around	**
„Retractable Menu “	1.2	1.2
Create a menu that can be put aside and easily retrieved again.		
row_striping	Showing Complex Data	**
		Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
„Row Striping “	1.3.4	1.3.3
Use two similar shades to alternately color the backgrounds of the table rows.		
scrolling_menu	Navigating around	**
„Scrolling Menu “	1.3.1	1.3.1
Show the items on a linear scrolling menu		
search	Searching	*
„Simple Search “	1.3.1	1.3.1
Offer a search		
search_results	Searching	**
„Search Results “	2.4	2.4
Present sorted results with a short description		
send_to_friend	Miscelleaneous	*
„Send to a friend “	1.3.1	1.3.4
Offer a the possibility to send a email with a link to the item		
service	Experiences	**
„Assistence Site “	2.4	2.4
Create a support section and back it up with additional searching facilities		
shopping	Experiences	**
„Shopping Experience “	2.4	2.4
Create an online shopping experience that matches off-line shopping experiences		
site_index	Searching	*
„Site Index “	1.3.4	1.3.3
Show all pages in an alphabetical index or by topic.		
slideshow	Basic interactions	**
„Slideshow “	1.2	1.2
Show each image for some seconds and provide controls to manually navigate back and forward, pause/resume and stop/return		
smart_menu_items	Commands and Actions	**
„Smart Menu Items “	1.3.4	1.3.4
Change menu labels dynamically to show precisely what they would do when invoked.		
smart_selection	Builders and Editors	**
„Smart Selection “	2.4	2.4
Make the software smart enough to automatically select a coherent group of items, rather than making the user do it.		
sortable_table	Showing Complex Data	**
„Sortable Table “	1.3.2	1.3.1
Show the data in a table, and let the user sort the table rows according to the column values.		
storytelling	Experiences	0
„Storytelling “	2.4	2.4
tag_cloud	Searching	**
„Tag Cloud “	2.2	1.2

Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
List the most common tags alphabetically and indicate their popularity by chaning the font size and weight		
teaser_menu	Navigating around	**
„Teaser Menu “	1.3.2	1.3.2
Show a partial menu with ëxpand"capabilities		
testimonials	Shopping	*
„Testimonials “	2.4	2.4
Allow users to give independent feedback on the quality product or services		
thumbnail	Dealing with data	*
„Thumbnail “	2.4	1.1
Display a smaller version of the movie, image or page.		
titled_sections	Organizing the Page	**
„Titled Sections “	1.1	1.1
Define separate sections of content by giving each one a visually strong title, and then laying them all out on the page together.		
top	Navigating around	**
„To the Top “	1.3.4	1.3.4
Provide a link to the top of the page at locations in the main content		
topic_pages	Searching	**
„Topic Pages “	2.4	2.4
Offer special topic pages with links to most relevant documents		
trail_menu	Navigating around	**
„Trail Menu “	1.3.3	1.3.3
Show the traversed path in the menu		
travel_site	Site types	**
„Travel Site “	2.4	2.4
Present a searchable database with booking possibilities		
tree_table	Showing Complex Data	**
„Tree-Table “	1.3.2	1.3.1
Put hierarchical data in columns, like a table, but use an indented outline structure in the first column to illustrate the tree structure.		
tutorial	Page types	**
„Tutorial “	2.4	2.4
two_panel_selector	Organizing the Content	**
„Two-Panel Selector “	1.3.1	1.3.1
Put two side-by-side panels on the interface. In the first, show a set of items that the user can select at will; in the other, show the content of the selected item.		
visual_framework	Organizing the Page	**
„Visual Framework “	1.4	1.4
Design each page to use the same basic layout, colors, and stylistic elements, but give the design enough flexibility to handle varying page content.		
wizard	Organizing the Content	**

Fortsetzung auf nächster Seite

Tabelle D.1 – Fortsetzung

Pattern-Id	**Kategorie**	**Confidence**
Bezeichnung	**Klassifikation XUL**	**Klassifikation Swing**
Kurzbeschreibung		
„Wizard “	1.3.1	1.3.1
Lead the user through the interface step by step, doing tasks in a prescribed order. — Take the user through the entire task one step at the time. Let the user step through the tasks and show which steps exist and which have been completed.		

Abkürzungsverzeichnis

AUI Abstract User Interface

AWT ist das Abstract Windowing Toolkit. Es ist das ursprüngliche UI-Toolkit der Sprache Java und entstand als die Schnittmenge der auf allen Plattformen darstellbaren UI-Elemente. Diesem Ziel der maximalen Plattformunabhängigkeit wurden erweiterte Fähigkeiten verschiedener Plattformen geopfert.

CUI Concrete User Interface

DSL Domain Specific Language

EMOF Essential MOF

EMP Eclipse Modeling Project

FUI Final User Interface

GMF Graphical Modeling Framework

HCI Human Computer Interaction / Mensch-Maschine-Kommunikation

MD-UID Model Driven User Interface Development

M2M Model To Model (Transformation)

M2T Model To Text (Transformation)

MDA Model Driven Architecture

MDSD Model Driven Software Development

MOF Meta Object Facility

OMG Object Management Group

PLML Patern Language Meta Language

Swing als der Nachfolger und Komplementär von AWT wurde entwickelt, um solche erweiterten UI-Fähigkeiten, wie z.B. Drag&Drop, Tooltips oder Multi-Document-Interfaces auch für Java-Programme plattformunabhängig bereitzustellen. Um dies zu erreichen werden die Bildschirmelemente durch Swing selbst gezeichnet, es erfolgt keine Koppelung mit plattform-nativen Widgets.

SWT ist das Standard Widget Toolkit, die Mächtigkeit von Swing und SWT ist vergleichbar. SWT koppelt jedoch die UI-Elemente mit nativen Widgets. Stellt die Plattform Widgets oder Funktionalitäten nicht zur Verfügung, werden diese durch SWT emuliert. SWT ist nicht auf alle Plattformen welche Java unterstützen portiert worden, dementsprechend sind Java-Programme die SWT nutzen nicht plattformunabhängig.

tDSL textuelle DSL

UI User Interface / Benutzungsschnittstelle

UIDL User Interface Definition/Declaration Language

WIMP Windows Icon Menu Pointer

WPF Windows Presentation Foundation

XHTML Extensible HyperText Markup Language, erweiterbares HTML, XML-konform

Tabellenverzeichnis

1.1 Aufzählung bekannterer XML-basierter Sprachen zur UI-Definition 15

2.1 Verfügbarkeit eines Metamodells für UIDLs aus Tabelle 1.1 42

3.1 Nicht im Webkatalog [Tid] enthaltene Patterns des Tidwell-Katalog [Tid06] 54
3.2 Kurzübersicht der Semantik der PLML-Sprachelemente [Fin03] 57
3.3 Klassifikationskategorien zur Komponentisierbarkeit von Pattern, nach [Arn04] 70
3.4 Problematische Patterneinträge in den Webkatalogen 75
3.5 Größenordnungen zur Nacheditierung des verschmolzenen Katalogs 80
3.6 Tabelle der als äquivalent zugeordneten Patterns 81
3.7 Kategorisierung der 135 Patterns des verschmolzenen Katalogs 83
3.8 Zuordnung der Classification-Literale zur Arnout'schen Einordnung 85
3.9 Komponentisierbarkeitsvermutung mit XUL als CUI-Sprache 85
3.10 Komponentisierbarkeitsvermutung mit Java Swing als CUI-System 86

4.1 Mapping von Java-Typen auf die Ecore-Äquivalente 93
4.2 Parameter zur Steuerung der Java⇒Ecore Transformation 97
4.3 Modellgrößen für diverse Sichtbarkeitsstufen beim JDK-Parsing 98

A.1 Temporale Operatoren zwischen Aufgaben in TEALLACH [GBM+99] 126
A.2 Syntaxelemente für Selektorausdrücke in CSS, nach [CSS] 137

D.1 Patternkurzbeschreibungen und Komponentisierbarkeitsvermutung 159

Abbildungsverzeichnis

1.1 Inhaltsübersicht, als Instanz des Forschungsframeworks nach Hevner [HMPR04] 1
1.2 Instanzen und Modelle im EMOF-Zusammenhang 7
1.3 Zeitliche Einordnung diverser MB-UID Systeme und Umgebungen 9
1.4 Die vier Abstraktionsebenen des CAMELEON-Referenzframeworks nach [CCT+03] . . . 11
1.5 Beispielhafte CAMELEON-Klassifikation, nach [CCT+03] 12
1.6 Modellzusammenhänge im MARIA-Migrationsprozess, nach [PSS09] 16
1.7 Klassendiagrammdarstellung des UsiXML-Schema, nach [VC] 17

2.1 Transformationsorientierter MDD-Entwicklungsprozess 25
2.2 Metamodell für das Aufgabenmodell nach [WF09] 27
2.3 Aufgabenmodell für die Klausurenkontrolle in der Rostocker Notation 28
2.4 Metamodell für das Nutzermodell 29
2.5 Allgemeines Objekt-Metamodell 30
2.6 Gegenüberstellung Objektmodelleditor und UML-Modell 31
2.7 Syntaxelemente eines Dialoggraphen 32
2.8 Beispiel eines Dialoggraphen, entnommen aus [Rie06] 33
2.9 Metamodell Dialoggraph nach [FRW08] 34
2.10 Annotiertes Aufgabenmodell und abgeleiteter Dialoggraph, nach [WF09] 35
2.11 Screenshot-Sequenz eines generierten Navigationsprototypen 36
2.12 Metamodell für AUI im X-AIM Prozess, Grundstruktur nach [Mül03] 39
2.13 Metamodell für AUI im X-AIM Prozess, Widgets nach [Mül03] 39
2.14 Adaptiertes AUI-Metamodell, Grundstruktur 40
2.15 Adaptiertes AUI-Metamodell, Widgets 41
2.16 Ausschnitt des XUL-Metamodells 45
2.17 XULE - XUL-UI im Bearbeitungsmodus geöffnet 46
2.18 XULE - Strukturübersicht und Eigenschaftenbearbeitung 47

3.1 Hilfsklassen im PLML-Metamodell 61

3.2 Kern des PLML-Metamodells, Deklaration eines Patterneintrags 62
3.3 Zusammenhang der Modelle des GMF 64
3.4 Screenshot eines exemplarischen graphischen PLML-Editors als Eclipse-Plugin 66
3.5 Verwendung von Pattern im MDD-UI Prozess 68
3.6 Verarbeitungskette zur Erstellung des PLML-Kataloges aus dem Tidwell-Webkatalog . . . 72
3.7 Intrakatalogreferenzen beim Tidwell-Katalog [Tid] 74
3.8 Intrakatalogreferenzen beim Welie-Katalog [vW] 75
3.9 Metamodell für PLMLComponent 78
3.10 Zusammenhänge und Transformationen von PLML und PLMLComponent 79

4.1 JavaDocs Java-Metamodell ohne Methoden 96
4.2 Zusammenhang zwischen Wrapper und EMF-Generator 102
4.3 Taschenrechner-Anwendung und geparste Modellinstanz im Standardeditor 104
4.4 Container-Infrastruktur mit Swing-Ecore und XUL-Generator 105
4.5 Visualisierung des Layoutalgorithmus für XUL 107
4.6 Drei Renderer des XUL-Snapshots der Taschenrechneranwendung 109

A.1 DRIVE - Modellsystem, nach [MK95] 125
A.2 Schrittfolge im TERESA-Migrationsprozess zur Übertragung der Benutzeroberfläche, nach [BP05] 134
A.3 UIML – Teilmodelle und Zusammenhänge nach [APB+99] 135

Listings

2.1 EBNF Grammatik für die Sichtzuordnung von Aufgaben 34
2.2 Ausschnitt aus einem xPand-Template 36
3.1 DTD des PLML-Standard [PLM] 56
3.2 Einführung eines Elementes für Katalog-Metadaten 58
3.3 QVTo/OCL Testselektion zur Qualitätssicherung der PLMLComponent-Instanz 80
3.4 QVTo/OCL Sprachvergleichsabfrage über dem PLMLComponent-Metamodell 86
4.1 Sichtbarkeitsüberlagerung bei gleichbenannten Attributen 92
4.2 Potentielles Modellierungsproblem bei kontravariantem Ausgabeparametertyp 92
A.1 Ausschnitt einer UI-Spezifikation mit der UIDE-IDL 119
A.2 Regeldefinition für DON 121
A.3 Syntaxstruktur und beispielhafte Verwendung von CSS 136
A.4 XUL-Minimalbeispiel mit CSS und JavaScript 137
B.1 PLML DTD, Sprachversion 1.1 139
B.2 PLML DTD, Sprachversion 1.5 141
B.3 PLML xText Grammatik, Sprachversion 1.5 143
B.4 Beispielkatalog mit der PLML-tDSL 145
B.5 XUL mit relativer Positionierung für das Demonstrationsbeispiel 147
C.1 Minimal-XUL zur Illustration der Tab-Verschachtelung 151
C.2 Einschränkung des Wertebereichs für confidence-Angaben 151
C.3 Annotation von Illustrations-URLs 152
C.4 Notation von Literaturreferenzen 152
C.5 Notation von Patternverlinkungen 152
C.6 Notation von Kategorien 153
C.7 xText-Grammatikausschnitt zur Eintragung von Patternbeziehungen 153

Algorithmenverzeichnis

1 Modellreduzierungsalgorithmus nach [SMBJ09] 100
2 Layout-Bildung mit relativen Koordinatenangaben 107

Index

ADEPT, 127

CAMELEON, 11
CSS, 136

DON, 120
DRIVE, 124

EclipseXML, 20

FUSE, 129

GIML, 20

HUMANOID, 122

JavaScript, 137

L-CID, 132

MASTERMIND, 128
MDA, 7
MDSD, 6
MECANO, 132
MOBI-D, 133

SwiXML, 20

T:XML, 21
TADEUS, 130
TEALLACH, 125
TRIDENT, 131

UIDE, 119
UIML, 135
useML, 21

WIMP, 23
WML, 20

XAML, 19
XIML, 18

Literaturverzeichnis

[Ado] Adobe. MXML 2009 - Functional and Design Specification. http://opensource.adobe.com/wiki/display/flexsdk/MXML+2009. Zuletzt abgerufen: 25.05.2010

[AG] HSH Nordbank AG. DocFactory. http://wwwswt.informatik.uni-rostock.de/deutsch/Mitarbeiter/andreas/docfactory.html. Zuletzt abgerufen: 06.06.2010

[AIS77] Christopher Alexander, Sara Ishikawa, Murray Silverstein. *A Pattern Language. Towns, Buildings, Construction.* Oxford University Press, 1977. ISBN: 0-19-501919-9

[Ant] ANTLR - Another Tool for Language Recognition. http://www.antlr.org. University of San Francisco. Zuletzt abgerufen: 06.06.2009

[APB+99] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, Jonathan E. Shuster. UIML: an appliance-independent XML user interface language. *Comput. Netw.* 31(11-16):1695–1708, 1999. DOI: http://dx.doi.org/10.1016/S1389-1286(99)00044-4

[Arn04] Karine Arnout. *From Pattern to Components*. Dissertation, ETH Zurich, 2004.

[Assa] Ramin Assisi. Visual Builder for Eclipse. http://v4all.sourceforge.net. Zuletzt abgerufen: 06.06.2009

[Assb] Moose Association. FAMIX Core 3.0. http://www.moosetechnology.org/docs/famix/3.0. Zuletzt abgerufen: 25.05.2010

[Assc] Moose Association. Moose - A platform for software and data analysis. http://www.moosetechnology.org/. Zuletzt abgerufen: 25.05.2010

[Assd] Moose Association. MSE - ein generisches Datenformat. http://scg.unibe.ch/wiki/projects/fame/mse. Zuletzt abgerufen: 25.05.2010

[Asse] Moose Association. XWT. http://www.moosetechnology.org/docs/famix/3.0. Zuletzt abgerufen: 25.05.2010

[AUI] AUIML - Abstract User Interface Markup Language Toolkit. IBM Alphaworks, Projekt eingestellt. Zuletzt abgerufen: 24.05.2010

[Beu05] Christian Beutenmüller. Deklarative XML-Sprachen für Benutzerschnittstellen. http://ebus.informatik.uni-leipzig.de/www/media/lehre/uiseminar05/ausarbeitung-beutenmueller.pdf, 2005. Seminararbeit. Zuletzt abgerufen: 25.05.2010

[BMZ05] Alexander Boedcher, Kizito Mukasa, Detlef Zuehlke. Capturing Common and Variable Design Aspects for Ubiquitous Computing with MB-UID. In *MDDoAUI 2005, Proceedings*. 2005 Montego Bay, Jamaika 2005. Zuletzt abgerufen: 15.02.2010

[BP05] Silvia Berti, Fabio Paternò. Migratory MultiModal interfaces in MultiDevice environments. In *ICMI '05: Proceedings of the 7th international conference on Multimodal interfaces*. Pp. 92–99. 2005 Torento, Italy ACM, New York, NY, USA, 2005. DOI: http://doi.acm.org/10.1145/1088463.1088481 ISBN: 1-59593-028-0

[BW09] Jens Brüning, Andreas Wolff. Declarative Models for Business Processes and UI Generation using OCL. In *The Pragmatics of OCL and Other Textual Specification Languages, Workshop at MODELS 2009*. 2009 Denver, Colorado, USA 2009. Zuletzt abgerufen: 30.09.2010

[CCT+03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, Jean Vanderdonckt. A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers* 15(3):289 – 308, 2003. Computer-Aided Design of User Interface. DOI: DOI: 10.1016/S0953-5438(03)00010-9 `http://www.sciencedirect.com/science/article/B6V0D-48KMN1X-1/2/231bbbf7f18c62246b343aab118ca2b7`

[CDSR09] Joanna Chimiak–Opoka, Birgit Demuth, Darius Silingas, Nicolas F. Rouquette. Requirements Analysis for an Integrated OCL Development Environment. In *The Pragmatics of OCL and Other Textual Specification Languages, Workshop at MODELS 2009*. 2009 Denver, Colorado, USA 2009. Zuletzt abgerufen: 30.09.2010

[CH03] Simon Crowle, Linda Hole. ISML: An Interface Specification Meta-language. In *Interactive Systems. Design, Specification, and Verification*. Pp. 255–268. 2003 2003.

[Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.* 1(1):9–36, 1976. DOI: http://doi.acm.org/10.1145/320434.320440

[CNM00] Stuart K. Card, Allen Newell, Thomas P. Moran. *The Psychology of Human-Computer Interaction*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 2000. ISBN: 0898598591

[Con] World Wide Web Consortium. XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/xslt20/`. W3C Recommendation 23 January 2007. Zuletzt abgerufen: 06.06.2010

[Cora] IBM Corporation. Generate code with Eclipse's Java Emitter Templates. `http://www.ibm.com/developerworks/library/os-ecemf2/`. Online Tutorial. Zuletzt abgerufen: 06.06.2010

[Corb] Yahoo! Corporation. Yahoo! Design Pattern Library. `http://developer.yahoo.com/ypatterns/`. Zuletzt abgerufen: 06.06.2010

[CRC04] Erwin Cuppens, Chris Raymaekers, Karin Coninx. VRIXML: A User Interface Description Language for Virtual Environments. In *Developing User Interfaces with XML: Advances on User Interface Description Languages*. Pp. 111–117. 2004 Gallipoli, Italy Mai 2004. Zuletzt abgerufen: 04.03.2010

[CSS] Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. `http://www.w3.org/TR/CSS2/`. Zuletzt abgerufen: 05.02.2010

[CTT] ConCurTaskTrees Environment. `http://giove.isti.cnr.it/tools/ctte/index.html`. Zuletzt abgerufen: 15.02.2010

[DeM79] T. DeMarco. *Structured analysis and system specification*. Yourdon Press, Upper Saddle River, NJ, USA, 1979. ISBN: 0-917072-14-6

[DF06] Andy Dearden, Janet Finlay. Pattern Languages in HCI: A critical review. *Human-Computer Interaction 21(1)*, Januar 2006.

[DF09] Anke Dittmar, Peter Forbrig. Task-based design revisited. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*. Pp. 111–116. 2009 Pittsburgh, PA, USA ACM, New York, NY, USA, 2009. DOI: http://doi.acm.org/10.1145/1570433.1570455 ISBN: 978-1-60558-600-7

[Die08] Christian Diebow. Entwicklung eines Konzeptes zur interaktiven Transformation von Aufgabenmodellen in Navigationsmodelle. Diplomarbeit, Universität Rostock, 2008.

[DR10] Sahin Albayrak Dirk Roscher, Marco Blumendorf. Multimodal User Interface Model For Runtime Distribution. In *MDDoAUI 2010, Proceedings*. Pp. 5–8. 2010 Atlanta, Georgia, United States April 2010.

[ECM] ECMAScript Language Specification, ECMA-262. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf`. Zuletzt abgerufen: 05.02.2010

[EFH+] Sven Efftinge, Peter Friese, Arno Haase, Dennis Hübner, Clemens Kadura, Bernd Kolb, Jan Köhnlein, Dieter Moroff, Karsten Thoms, Markus Völter, Patrick Schönbach, Moritz Eysholdt, Dennis Hübner, Steven Reinisch. openArchitectureWare User Guide, Version 4.3.1. `http://www.openarchitectureware.org/pub/documentation/4.3.1/openArchitectureWare-4.3.1-Reference.pdf`. Zuletzt abgerufen: 30.09.2010

[emf] Eclipse Modeling Framework Project. `http://www.eclipse.org/modeling/emf/?project=emf`. Eclipse Foundation. Zuletzt abgerufen: 06.06.2009

[EMP] Eclipse Modeling Project. `http://www.eclipse.org/modeling/`. Eclipse Foundation. Zuletzt abgerufen: 25.05.2010

[EV] Sven Efftinge, Markus Voelter. oAW xText: A framework for textual DSLs. `http://eclipsesummit.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium12_xTextFramework.pdf`. Zuletzt abgerufen: 06.06.2009

[FFMM04] D. Fogli, G. Fresta, A. Marcante, P. Mussio. IM2L: A User Interface Description Language Supporting Electronic Annotation. In *Proc. Workshop on Developing User Interface with XML: Advances on User Interface Description Languages*. Pp. 135–142. May 2004 May 2004.

[FGK88] J. Foley, C. Gibbs, S. Kovacevic. A knowledge-based user interface management system. In *CHI '88: Proceedings of the SIGCHI conference on Human factors in computing systems*. Pp. 67–72. 1988 Washington, D.C., United States ACM, New York, NY, USA, 1988. DOI: http://doi.acm.org/10.1145/57167.57178 ISBN: 0-201-14237-6

[Fin03] Sally Fincher. Perspectives on HCI patterns: concepts and tools (introducing PLML). In *Workshop at CHI 2003*. 2003 Sept. 2003. www.cs.kent.ac.uk/~saf/patterns/CHI2003WorkshopReport.doc

[FKKM91] James Foley, Won Chul Kim, Srdjan Kovacevic, Kevin Murray. UIDE—an intelligent user interface design environment. In *Intelligent user interfaces*. Pp. 339–384. 1991 ACM, New York, NY, USA, 1991. DOI: http://doi.acm.org/10.1145/107215.128716 ISBN: 0-201-50305-0

[Fol] Eelke Folmer. Interaction Design Patterns for Games Library. http://www.helpyouplay.com/wiki/index.php5?title=Main_Page. Zuletzt abgerufen: 06.06.2010

[Foua] Eclipse Foundation. ATL Modell-Zoo. http://www.eclipse.org/m2m/atl/atlTransformations/, 2010-06-06.

[Foub] Eclipse Foundation. Eclipse Toolkit Model. http://wiki.eclipse.org/E4/UI/Toolkit_Model. Zuletzt abgerufen: 25.05.2010

[Fouc] Eclipse Foundation. Graphical Editing Framework. http://www.eclipse.org/gef/. Zuletzt abgerufen: 25.05.2010

[FRW08] Peter Forbrig, Daniel Reichard, Andreas Wolff. User Interfaces from Task Models. In *MDSE'08*. 2008 Berlin, Germany 2008.

[GBM+99] Tony Griffiths, Peter Barclay, Jo Mckirdy, Norman Paton, Philip Gray, Jessie Kennedy, Richard Cooper, Carole Goble, Adrian West, Michael Smyth. Teallach: A Model-Based User Interface Development Environment for Object Databases. In *In Proceedings of UIDIS'99*. Pp. 86–96. 1999 IEEE Press, 1999.

[GBR07] Martin Gogolla, Fabian Büttner, Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, pp. 69:27–34, 2007.

[Gec] Gecko Rendering Engine für XUL. https://developer.mozilla.org/de/Gecko. Mozilla Corporation. Zuletzt abgerufen: 05.03.2010

[GHJV02] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 24th edition, 2002.

[gim] DTD der GIML. http://gitk.sourceforge.net/giml.dtd. GITK Sourceforge Projekt. Zuletzt abgerufen: 04.03.2010

[GKF86] C. Gibbs, Won Chul Kim, James Foley. Case Studies in the Use of IDL: Interface Definition Language. Report GWU-IIST-86-30, Dept. of EE&CS, George Washington University, Washington DC, 1986.

[GKR+07] Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, Steven Völkel. Text-based Modeling. In *4th International Workshop on Language Engineering (ATEM 2007)*. 2007 2007. Zuletzt abgerufen: 19.03.2010

[gmf] Eclipse Graphical Modeling Framework Project. http://www.eclipse.org/modeling/gmf/. Eclipse Foundation. Zuletzt abgerufen: 06.06.2009

[GB10] Peter Forbrig Gregor Buchholz. Analyse von Handlungsprotokollen zur Modellbildung. *Mensch & Computer 2010 München: Oldenbourg Verlag*, pp. 361–370, 2010.

[Gro] Object Management Group. MOF Model to Text Transformation Language. `http://www.omg.org/spec/MOFM2T/1.0/`. Sprachstandard Version 1.0. Zuletzt abgerufen: 06.06.2010

[GSR05] Mohammed Gomaa, Akram Salah, Syed Rahman. Towards A Better Model Based User Interface Development Environment: A Comprehensive Survey. `http://www.micsymposium.org/apache2-default/mics_2005/papers/paper72.pdf`, 2005. Midwest Instruction and Computing Symposium. Zuletzt abgerufen: 02.07.2009

[Ham] Dennis E. Hamilton. TROST Pattern Format. `http://trosting.org/info/2005/08/i050803c.htm`. Zuletzt abgerufen: 27.05.2010

[HMPR04] Alan R. Hevner, Salvatore T. March, Jinsoo Park, Sudha Ram. Design Science in Information Systems Research. *MIS Quarterly* 28(1):75–105, 2004. `http://www.jstor.org/stable/25148625`

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM* 21(8):666–677, 1978. DOI: http://doi.acm.org/10.1145/359576.359585

[Ite] Open Architecure Ware - Projekthauptseite. `http://oaw.itemis.de/`. Itemis AG. Zuletzt abgerufen: 18.03.2010

[JAB+06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, Patrick Valduriez. ATL: a QVT-like transformation language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. Pp. 719–720. 2006 Portland, Oregon, USA ACM, New York, NY, USA, 2006. DOI: http://doi.acm.org/10.1145/1176617.1176691 ISBN: 1-59593-491-X

[Jav] Java Community Process, Fragen und Antworten zu. `http://www.jcp.org/en/introduction/faq#general`. Zuletzt abgerufen: 05.03.2010

[JJ91] Hilary Johnson, Peter Johnson. Task knowledge structures: Psychological basis and integration into system design. *Acta Psychologica* 78(1-3):3 – 26, 1991. DOI: DOI: 10.1016/0001-6918(91)90003-I `http://www.sciencedirect.com/science/article/B6V5T-45WYWWC-7F/2/f3ec4ae7650971c6d88746471de30b69`

[Ker] Kermeta meta programming environment. `http://www.kermeta.org`. Zuletzt abgerufen: 28.01.2010

[KF90] Won Chul Kim, James D. Foley. DON: user interface presentation design assistant. In *Proceedings of the 3rd annual ACM SIGGRAPH symposium on User interface software and technology*. UIST '90, pp. 10–20. 1990 Snowbird, Utah, United States ACM, New York, NY, USA, 1990. DOI: http://doi.acm.org/10.1145/97924.97926 `http://doi.acm.org/10.1145/97924.97926` ISBN: 0-89791-410-4

[Kir] Thomas Kirste. Lehrmaterialien Programmierungstechnik I, WS 2006.

[Kos06] Stefan Kost. *Dynamically generated multi-modal application interfaces*. Dissertation, Technische Universität Dresden, 2006.

[Kro99] Eric Krock. Welcome to the Navigator 5 XUL newsgroup! http://groups.google.com/group/netscape.public.dev.xul/browse_thread/thread/d75112c638f8f72e, 1999. Usenet-Posting durch verantwortlichen Netscape-Mitarbeiter. Zuletzt abgerufen: 25.05.2010

[LPV01] Quentin Limbourg, Costin Pribeanu, Jean Vanderdonckt. Towards Uniformed Task Models in a Model-Based Approach. In *Lecture Notes in Computer Science Volume 2220/2001*. Pp. 164–182. 2001 2001.

[LS96] Frank Lonczewski, Siegfried Schreiber. The FUSE-System: an Integrated User Interface Design Environment. In *CADUI'96*. Pp. 37–56. 1996 1996.

[mda] MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01. Object Management Group. Zuletzt abgerufen: 06.07.2009

[MFD+05] Pierre-Alain Muller, Franck Fleurey, Zoé Drey, Damien Pollet, Frédéric Fondement. On Executable Meta-Languages applied to Model Transformations. In *in Model Transformations In Practice Workshop, Montego*. 2005 2005.

[Mic] Microsoft. Pattlet Definition. http://guidanceexplorer.codeplex.com/wikipage?title=GuidanceExplorerOverview. Zuletzt abgerufen: 06.06.2010

[Mic04] Microsoft. Extensible Application Markup Language. http://msdn.microsoft.com/en-us/library/ms752059.aspx, 2004. XAML. Zuletzt abgerufen: 25.05.2010

[MK95] Kenneth Mitchell, Jessie B Kennedy. An Environment for the Organised Construction of User-Interfaces to Databases. http://www.soc.napier.ac.uk/publication/op/getpublication/publicationid/4151, 1995. Technischer Bericht, freigegeben zur Veröffentlichung. Zuletzt abgerufen: 01.07.2009

[Mül03] Andreas Müller. *Spezifikation geräteunabhängiger Benutzerschnittstellen durch Markup-Konzepte*. Dissertation, Universität Rostock, 2003.

[Mod] Metamodelle im MDD-Prozess. http://www.ptbv.de/metamodels. Eigene Homepage. Zuletzt abgerufen: 01.12.2010

[MOF02] Meta Object Facility (MOF), Version 1.4. http://www.omg.org/cgi-bin/doc?formal/2002-04-03, 2002. öffentliche Spezifikation. Zuletzt abgerufen: 01.07.2009

[MOF06] Meta Object Facility (MOF), Version 2.0. http://www.omg.org/cgi-bin/doc?formal/2006-01-01, 2006. öffentliche Spezifikation. Zuletzt abgerufen: 23.07.2010

[MPS04] Giulio Mori, Fabio Paternò, Carmen Santoro. Design and Development of Multi-Device User Interfaces through Multiple Logical Descriptions. *IEEE Transactions on Software Engineering* 30(8):507–520, Aug 2004.

[MPWJ92] P. Markopoulos, J. Pycock, S. Wilson, P. Johnson. Adept-a task based design environment. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*. Volume ii, pp. 587 –596 vol.2. 1992 jan 1992. DOI: 10.1109/HICSS.1992.183310

[Obe] Obeo. Acceleo : MDA generator. http://www.acceleo.org/pages/home/en. Zuletzt abgerufen: 06.06.2010

[OCL10] Object Constraint Language, Version 2.2. http://www.omg.org/spec/OCL/2.2/, 2010. öffentliche Spezifikation. Zuletzt abgerufen: 01.10.2010

[OMG] MDA Guide Version 1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01. Zuletzt abgerufen: 15.02.2010

[Pat99] Fabio Paternò. *Model-Based Design and Evaluation of interactive applications*. Springer, 1999. ISBN: 1-852-33155-0

[Pau] Wolf Paulus. SwiXML. http://www.swixml.org. Zuletzt abgerufen: 03.06.2010

[PE02] Angel Puerta, Jacob Eisenstein. XIML: A Universal Language for User Interfaces. http://www.ximl.org/documents/XimlWhitePaper.pdf, 2002. Zuletzt abgerufen: 25.05.2010

[Pet] Roland Petrasch. Model Based User Interface Design: Model Driven Architecture und HCI Patterns. http://pi.informatik.uni-siegen.de/stt/27_3/03_Technische_Beitraege/MDA_HCI_Patterns_Petrasch_Short.pdf. Zuletzt abgerufen: 06.06.2009

[PH09] Angel Puerta, Martin Hu. UI Fin: A Process-Oriented Interface Design Tool. In *IUI 2009 - Sanibel Island, Florida, USA*. Pp. 345–354. 2009 2009.

[PLM] DTD of PLML. http://www.hcipatterns.org/tiki-download_file.php?fileId=7. Zuletzt abgerufen: 06.06.2009

[PM97] Angel R. Puerta, David Maulsby. MOBI-D: a model-based development environment for user-centered design. In *CHI '97: CHI '97 extended abstracts on Human factors in computing systems*. Pp. 4–5. 1997 Atlanta, Georgia ACM, New York, NY, USA, 1997. DOI: http://doi.acm.org/10.1145/1120212.1120215 ISBN: 0-89791-926-2

[Pro] Sourceforge Projekt. HTML Tidy Library Project. http://tidy.sf.net. Zuletzt abgerufen: 03.03.2010

[PS06] David Paquette, Kevin A. Schneider. Task Model Simulation Using Interaction Templates. *Interactive Systems*, pp. 78–89, 2006. DOI: 10.1007/11752707_7

[PSS09] Fabio Paterno', Carmen Santoro, Lucio Davide Spano. MARIA: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact.* 16(4):1–30, 2009. DOI: http://doi.acm.org/10.1145/1614390.1614394

[Pue91] Angel R. Puerta. Rapid Prototyping of Self-Adaptive Interfaces with the L-CID Model. In *In Proceedings of the Intelligent Multimedia Interfaces Workshop, AAAI-91*. Pp. 37–46. 1991 1991.

[Pue96] Angel Puerta. The MECANO Project: Comprehensive and Integrated Support for Model-Based Interface Development. In *In Computer-Aided Design of User Interfaces*. Pp. 5–7. 1996 Namur University Press, 1996.

[QVT08] Query View Transformation. http://www.omg.org/spec/QVT/1.0/, 2008. öffentliche Spezifikation. Zuletzt abgerufen: 11.08.2010

[iRAVI05] Workshop im Rahmen Advanced Visual Interfaces 2004. Linking GUI Elements to Tasks – Supporting an Evolutionary Design Process. In *Developing User Interfaces with XML: Advances on User Interface Description Languages, Gallipoli, Italy*. 2005 2005.

[Rat06] Robert Rathsack. Generierung von Geraetespezifikationen aus abstrakten Spezifikationen unter Beachtung von HCI Pattern. Diplomarbeit, Universität Rostock, 2006.

[Rea09] Lopez-Jaquero; Montero; Real. Designing User Interface Adaptation Rules with T:XML. In *IUI 2009 - Sanibel Island, Florida, USA*. 2009 2009.

[REM09] *R.E.M. 2009, Workshop at WCRE 2009*. 2009 Lille, France Okt. 2009.

[Reu03] Achim Reuther. *Systematische Entwicklung von Maschinenbediensystemen mit XML*. Dissertation, Technische Universität Kaiserslautern, 2003.

[rfc] Uniform Resource Locators. `http://www.ietf.org/rfc/rfc1738.txt`. IETF - RFC 1738. Zuletzt abgerufen: 19.06.2009

[Rie06] Sven Rieckerhoff. Modularisierung von Navigationsspezifikationen unter Berücksichtigung von HCI-Patterns. Diplomarbeit, Universität Rostock, 2006.

[RWF06] Robert Rathsack, Andreas Wolff, Peter Forbrig. Using HCI-Patterns with Model-based Generation of Advanced User-Interfaces. In Pleuss et al. (eds.), *Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces*. 2006 Genua, Italien, Okt. 2006. `http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-214/paper11.pdf` Zuletzt abgerufen: 22.06.2011

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks. *EMF Eclipse Modeling Framework, Second Edition*. Addison-Wesley, 2009.

[Sch93] Thomas Schlungbaum, Egbert; Elwert. Dialogue Graphs - A Formal and Visual Specification Technique For Dialogue Modelling. In *Procedings of the BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*. 1993 1993.

[Sch08] Enrico Schnepel. GenGMF: Efficient editor development for large meta models using the Graphical Modelling Framework. In *SIG-MDSE 2008, Transformations and Tools*. 2008 2008. Zuletzt abgerufen: 23.03.2010

[SE96] Edgar Schlungbaum, Thomas Elwert. Dialogue Graphs - A Formal and Visual Specification Technique for Dialogue Modelling. In *BCS-FACS Workshop on Formal Aspects of the Human Computer Interface*. 1996 Sheffield, UK BCS, British Computer Society, 1996.

[SLN92] Pedro Szekely, Ping Luo, Robert Neches. Facilitating the exploration of interface design alternatives: the HUMANOID model of interface design. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*. Pp. 507–515. 1992 Monterey, California, United States ACM, New York, NY, USA, 1992. DOI: http://doi.acm.org/10.1145/142750.142912 ISBN: 0-89791-513-5

[SMBJ09] Sagar Sen, Naouel Maha, Benoit Baudry, Jean-Marc Jezequel. Meta-model pruning. In *Models 2009, Proceedings*. Pp. 32–46. 2009 Denver, Colorado, United States 2009.

[SR98] K. Stirewalt, S. Rugaber. Automating UI Generation by Model Composition. *Automated Software Engineering, International Conference on* 0:177, 1998. DOI: http://doi.ieeecomputersociety.org/10.1109/ASE.1998.732624 ISBN: 0-8186-8750-9

[SSC+96] Pedro A. Szekely, Piyawadee Noi Sukaviriya, Pablo Castells, Jeyakumar Muthukumarasamy, Ewald Salcher. Declarative interface models for user interface construction tools: the MASTERMIND approach. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*. Pp. 120–150. 1996 Chapman & Hall, Ltd., London, UK, UK, 1996. ISBN: 0-412-72180-5 Zuletzt abgerufen: 30.11.2010

[ST04] Kinan Samaan, Franck Tarpin-Bernard. Task models and interaction models in a multiple user interfaces generation process. In *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*. Pp. 137–144. 2004 Prague, Czech Republic ACM, New York, NY, USA, 2004. DOI: http://doi.acm.org/10.1145/1045446.1045471 ISBN: 1-59593-000-0

[Sta73] Herbert Stachowiak (ed.). *Allgemeine Modelltheorie*. Springer, Wien [u.a.], 1973. `https://aleph.ub.uni-kl.de/F/?func=full-set-set&set_number=002380&set_entry=000012&format=999` ISBN: 3-211-81106-0

[Sta00] C. Stary. TADEUS: seamless development of task-based and user-oriented interfaces. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on* 30(5):509–525, sep 2000. DOI: 10.1109/3468.867859

[Sti97] R. E. Kurt Stirewalt. The Design and Implementation of the MASTERMIND Toolkit. Technical report, Georgia Institute of Technology, 1997. Zuletzt abgerufen: 30.11.2010

[SVG] Scalable Vector Graphics. `http://www.w3.org/TR/SVG11/`. W3C-Standard, V1.1. Zuletzt abgerufen: 15.02.2010

[SW00] Uwe Schneider, Dieter Werner. *Taschenbuch der Informatik*. Fachbuchverlag Leipzig, 2000. ISBN: 3-446-21331-7

[TC99] David Thevenin, Joelle Coutaz. Plasticity of User Interfaces: Framework and Research Agenda. In *Human-computer interaction INTERACT'99*. Pp. 110–116. 1999 Edinburg, United Kingdom IOS Press, 1999.

[TER] TERESA XML Standards. `http://giove.isti.cnr.it/tools/TERESA/teresa_xml.html`. Zuletzt abgerufen: 15.02.2010

[Tid] Jennifer Tidwell. Patterns for Effective Interaction Design. `http://www.designinginterfaces.com`. Zuletzt abgerufen: 06.06.2011

[Tid06] Jennifer Tidwell. *Designing Interfaces*. O'Reilly Media, Inc., 1005 Gravenstein Highway North Sebastopol, CA 95472 USA, 2006. ISBN: 978-0-596-00803-1

[Tou07] Frederic Toussaint. *Grafische Benutzungsunterstützung auf Befehlsebene für die Entwicklung massivparalleler Programme*. Dissertation, Universität Karlsruhe, 2007.

[Tox] Anders Toxboe. User Interface Design Patterns. `http://ui-patterns.com`. Zuletzt abgerufen: 06.06.2010

[TR09] Bruce Trask, Angel Roman. Model Driven Engineering. In *R.E.M. 2009, Workshop at WCRE 2009*. 2009 Lille, France Sept. 2009. Zuletzt abgerufen: 18.01.2010

[UIMa] Spezifikation von UIML. `http://www.oasis-open.org/committees/download.php/5937/uiml-core-3.1-draft-01-20040311.pdf`. UIML Projekt. Zuletzt abgerufen: 04.03.2010

[UIMb] Auflistung/Verlinkung UIML-Renderer. http://www.uiml.org/tools/index.htm. UIML Projekt. Zuletzt abgerufen: 24.09.2010

[UML06] UML Infrastrucutre Specifikation, Version 2.3. http://www.omg.org/spec/UML/2.3/Infrastructure/PDF/, 2006. öffentliche Spezifikation. Zuletzt abgerufen: 23.07.2010

[use] Schema und Werkzeuge für useML. http://www.uni-kl.de/pak/useML/. UsiXML Projekt. Zuletzt abgerufen: 04.03.2010

[USIa] Schema und Modelle für UsiXML. http://www.usixml.org/index.php?mod=pages&id=5. UsiXML Projekt. Zuletzt abgerufen: 04.03.2010

[Usib] UsiXML ITEA Projekt. http://itea.defimedia.be/about-the-project. Zuletzt abgerufen: 25.05.2010

[Usic] State of the Art of User Interface Description Languages. http://itea.defimedia.be/sites/default/files/UsiXML_D1.1_Final.doc. Zum Abrufzeitpunkt als Rohversion vorliegend. Zuletzt abgerufen: 02.06.2010

[Van95] François Bodart; Anne-Marie Hennebert; Jean-Marie Leheureux; Isabelle Provot;Benoît Sacré; Jean Vanderdonckt. Towards a Systematic Building of Software Architectures: the TRIDENT Methodological Guide. In *Eurographics Workshop on Design, Specification, Verification of Interactive Eurographics Workshop on Design, Specification, Verification of Interactive Systems DSV-IS'95*. 1995 Château de Bonas, France 1995.

[VC] Jean Vanderdonckt, Juan Manuel Gonzalez Calleros. UsiXML, a User Interface Model and Language Engineering approach. http://www.w3.org/2008/10/mbui/UsiXML-MBUI-W3C2008.pdf. Präsentation auf W3C-Workshop. Zuletzt abgerufen: 25.05.2010

[VLM+04] Jean Vanderdonckt, Quentin Limbourg, Benjamin Michotte, Laurent Bouillon, Daniela Trevisan, Murielle Florins. USIXML: a User Interface Description Language for Specifying Multimodal User Interfaces. In *W3C Workshop on Multimodal Interaction*. Juli 2004 Juli 2004.

[VM03] R. Van Buskirk, B. W. Moroney. Extending prototyping. *IBM Syst. J.* 42:613–623, October 2003. DOI: http://dx.doi.org/10.1147/sj.424.0613 http://dx.doi.org/10.1147/sj.424.0613

[Voa98] J.M. Voas. Certifying off-the-shelf software components. *Computer* 31(6):53–59, Jun 1998. DOI: 10.1109/2.683008

[Voi] Voice Extensible Markup Language. http://www.w3.org/TR/voicexml30/. W3C-Standard, V3.0. Zuletzt abgerufen: 15.02.2010

[W3C01] W3C. Wireless Markup Language, Version 2, Spezifikation. http://www1.wapforum.org/tech/terms.asp?doc=WAP-238-WML-20010911-a.pdf, 2001. Zuletzt abgerufen: 25.05.2010

[vW] Martijn van Welie. Patterns in interaction design. http://www.welie.com/patterns/index.php. Zuletzt abgerufen: 23.06.2011

[Wer44] Max Wertheimer. *Gestalt Theory*. Hayes Barton Press, 1944.

[WF07] Andreas Wolff, Peter Forbrig. Model-based Reengineering of User Interfaces. In *MDDoAUI 2007, Proceedings*. 2007 Nashville, Tennessee, USA 2007. Zuletzt abgerufen: 15.02.2010

[WF09] Andreas Wolff, Peter Forbrig. Deriving User Interfaces from Task Models. In *MDDoAUI 2009, Proceedings*. 2009 Sanibel Island, Florida, United States 2009. Zuletzt abgerufen: 22.06.2011

[WFDR05a] Andreas Wolff, Peter Forbrig, Anke Dittmar, Daniel Reichart. Development of Interactive Systems Based on Patterns. In *Workshop on Mapping User Needs into Interaction Design Solutions at Interact*. 2005 Rome, Italy, Sept. 2005.

[WFDR05b] Andreas Wolff, Peter Forbrig, Anke Dittmar, Daniel Reichart. Linking GUI Elements to Tasks – Supporting an Evolutionary Design Process. In *Tamodia 2005 - Danzig, Polen*. 2005 2005.

[WFR05] Andreas Wolff, Peter Forbrig, Daniel Reichart. Tool Support for model-based Generation of Advanced User-Interfaces. In *MDDoAUI 2005, Proceedings*. 2005 Montego Bay, Jamaika 2005. Zuletzt abgerufen: 15.02.2010

[Wik] Wikipedia.de. Template Engine. `http://de.wikipedia.org/w/index.php?title=Template_Engine&oldid=78943646`. Zuletzt abgerufen: 30.09.2010

[WB09] R.J. Wirfs-Brock. Principles in Practice. *Software, IEEE* 26(4):11 –12, Juli 2009. DOI: 10.1109/MS.2009.99

[Woi09] Felix Woitzel. UsiXML im USG-Prozess. Lehrstuhl Softwaretechnik, Universität Rostock, Mai 2009. Studienarbeit.

[Wol04] Andreas Wolff. Ein Konzept zur Integration von Aufgabenmodellen in das GUI-Design. Diplomarbeit, Universität Rostock, 2004.

[WV03] Martijn van Welie, Gerrit C. van der Veer. Pattern Languages in Interaction Design: Structure and Organization. In al. (ed.), *Proceedings of Interact 2003*. Pp. 527–534. 2003 2003.

[XML] XML Standard. `http://www.w3.org/TR/2008/REC-xml-20081126/`. W3C Recommendation 26 November 2008. Zuletzt abgerufen: 24.05.2010

[XUL] Gecko Rendering Engine für XUL. `http://mozilla.doslash.org/xulschema/generating.html`. Mozilla Corporation. Zuletzt abgerufen: 05.03.2010