Entwurf eines Objektmodells für semistrukturierte Daten im Kontext von XML Content Management Systemen

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Patrick Lay

aus

Bad Neuenahr-Ahrweiler / Rheinland-Pfalz

Bonn, 2006

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn. Diese Dissertation ist auf dem Hochschulschriftenserver der ULB Bonn `http://hss.ulb.uni-bonn.de/diss_online` elektronisch publiziert.

1. Referent: Prof. Dr. Armin B. Cremers
2. Referent: Prof. Dr. Rainer Manthey

Tag der Promotion: 12. Januar 2007

Erscheinungsjahr: 2007

Kurzfassung

Seit einigen Jahren werden zur Erstellung und Pflege umfangreicher Websites vornehmlich Content Management Systeme (CMS) eingesetzt. Die überwiegende Entwicklungsarbeit wurde bei diesen Systemen in immer mächtigere Templatesprachen und aufwändigere Benutzeroberflächen investiert. Aspekte, die das zugrunde liegende Datenmodell betreffen, wurden dabei allerdings vernachlässigt: zumeist wird mehr oder weniger direkt auf eine Datenbank zugegriffen.

Ziel der vorliegenden Arbeit ist der Entwurf einer auf die CMS-Aufgabenstellung ausgerichteten Architektur, deren Datenmodell zusammen mit einer darauf abgestimmten Templatesprache die Entwicklung neuer Webangebote effizienter und einfacher machen kann.

Zunächst werden die Anforderungen an ein Datenmodell für CMS erarbeitet. Darauf basierend wird ein objektorientiertes Modell entwickelt und mittels einer formalen Semantikbeschreibung definiert. Das Modell unterstützt inhärent die typischen hierarchischen Strukturen von Websites. Vererbung ist durch Verwendung einer speziellen Form der Familienpolymorphie sowohl zwischen einzelnen Klassen als auch zwischen kompletten Webanwendungen möglich. Ein Entwickler kann große Teile von vorhandenem Code auf dem Wege der Vererbung wiederverwenden. Eine Zerlegung von Objekten in feingranulare Bestandteile ermöglicht eine weitere Reduzierung von Redundanzen bei der Modellierung.

Als Bindeglied zwischen Daten und publizierten Dokumenten haben sich Templates bewährt. Eine Analyse der Anforderungen und vorhandenen Ansätze führt zu der Entscheidung für eine funktionale Templatesprache. Diese Templatesprache und die Beschreibung des objektorientierten Datenmodells werden zu einer einheitlichen Sprache integriert, die zudem durch die die Einbindung von XML die Ausgabe in unterschiedlichen Dokumentenformaten unterstützt.

Abstract

For several years, mostly Content Management Systems (CMS) are used for the creation and maintenance of large web sites. Most of the development effort of these systems has been put in more sophisticated template languages und complex user interfaces. Aspects regarding the underlying data model have been neglected though: the database is accessed more or less directly in most cases.

The goal of the present thesis is the design of an architecture which is attuned to the tasks of CMS, the data model of which together with a template language can make the development of new web sites more efficient and easier.

At first, the requirements on a data model for CMS are compiled. Thereupon an object-oriented model is developed and defined by means of a formal semantic description. This model inherently supports the typical hierarchical structures of websites. Inheritance is possible between single classes as well as between entire web applications by utilizing a special kind of family polymorphism. A developer can re-use large parts of existing code through inheritance. Moreover, a decomposition of objects into fine-grained parts allows the reduction of redundancies in modelling.

Templates have proved as the crucial link between data and published pages. An analysis of requirements and existing approaches leads to the decision for a functional template language. This template language and the description of the objekt-oriented data model are integrated into a uniform language which furthermore supports the output in different document formats by integrating XML.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

In den Anfängen des World Wide Web (WWW) zu Beginn der 1990er Jahre bestanden →[1]Websites zumeist aus einzelnen, statischen HTML-Dateien, die untereinander verlinkt waren – ein Ansatz, der bei kleinen Datenmengen bzw. Websites sinnvoll ist und dort auch heute noch verwendet wird. Andere, z. B. binäre Dokumente wie PDF-Dateien können ebenfalls eingebunden werden. Bei diesem Ansatz besteht in der Regel eine 1:1-Abbildung von →Webseiten auf Dateien, d. h. eine im Browser sichtbare Webseite entspricht einer Datei auf dem Webserver. Bei einer Anfrage liefert der Webserver die entsprechenden Dateien an den anfragenden →Client (zumeist ein Webbrowser) aus, wie Abbildung 1.1 illustriert.

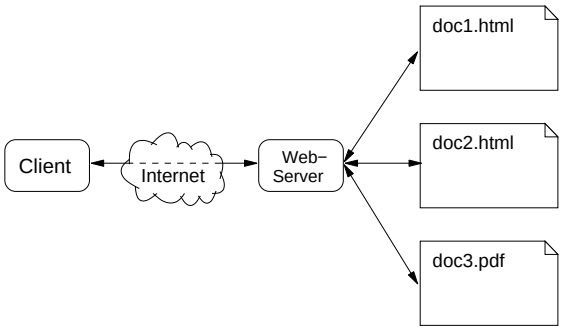


Abbildung 1.1.: WWW-Struktur (1): Statische HTML-Seiten

Mit zunehmender Größe sowohl des gesamten WWW als auch der einzelnen Websites an sich wurde dieser Ansatz jedoch zusehends unflexibel und schwer pflegbar: Beispielsweise müssen auf mehreren Seiten vorkommende Inhalte (z. B. Logos oder Kopfzeilen) in allen betreffenden Dateien wiederholt werden; zudem müssen Verknüpfungen (*Links*) zwischen den Seiten manuell konsistent gehalten werden, was erfahrungsgemäß schwierig bzw. fehleranfällig ist.

[1] Im Kontext der Arbeit wichtige Begriffe sind mit einem →gekennzeichnet und werden im Glossar (Seite 159 ff.) erklärt.

Eine Weiterentwicklung dieses Ansatzes, so genannte *CGI-Skripte* (Common Gateway Interface[2]), sind Programme, die auf dem Webserver ausgeführt werden und dort HTML-Seiten dynamisch, z. B. aus Datenbanken, generieren können – statische Seiten können weiterhin ausgeliefert werden. (s. Abbildung 1.2). Bei entsprechend umfangreichen Skripten können so einige der im statischen Ansatz auftretenden Probleme umgangen werden, beispielsweise können Links überprüft oder häufig benutzte Inhalte mehrfach verwendet werden. CGI-Skripte sind jedoch in der Regel Perl- oder Unix-Shellskripte, die bei größe-

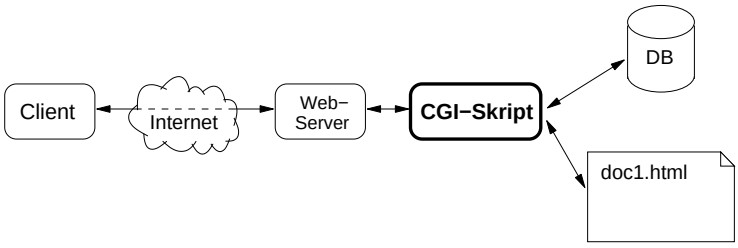


Abbildung 1.2.: WWW-Struktur (2): CGI-Skripte

ren Webangeboten hinsichtlich der Flexibilität und Effizienz zu wünschen übrig lassen, da derartige Scriptsprachen ursprünglich für andere Anwendungen konzipiert wurden und sie zudem in aller Regel interpretiert werden, was sich in der Regel in einer schlechten Performanz niederschlägt.

Daher wurden komplexere Anwendungen – Content Management Systeme (CMS, s. Abbildung 1.3) – entwickelt, die Daten flexibler und effizienter verarbeiten können und dem Benutzer die Arbeit mit der Website einfacher gestalten, indem sie umfangreiche Möglichkeiten der Contentverwaltung- und aufbereitung sowie verschiedene andere Komponenten wie beispielsweise ein Rechte- und Benutzermanagement zur Verfügung stellen. CMS werden in Kapitel 3 detailliert beschrieben.

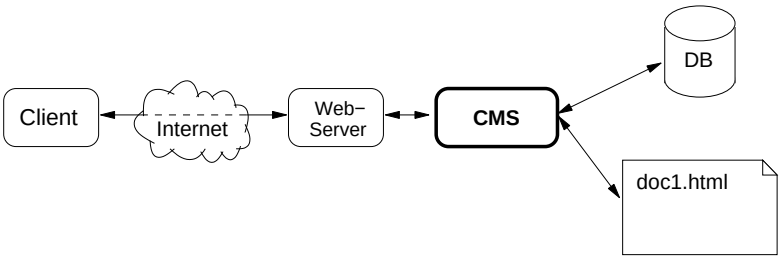


Abbildung 1.3.: WWW-Struktur (3): CMS

[2] `http://hoohoo.ncsa.uiuc.edu/cgi/`

Das Wob 1-System, welches im Vorfeld der vorliegenden Arbeit entwickelt wurde, war eines der ersten Systeme dieser Art. Erste Überlegungen bezüglich eines geeigneten Datenmodells, die im Laufe der Arbeit vorgestellt werden, wurden dort bereits prototypisch umgesetzt. Die mit Wob 1 gesammelten Erfahrungen zeigten, dass einige der dort umgesetzten Konzepte wie beispielsweise die Kombination von funktionalen und objekt-orientierten Aspekten im Kontext von CMS gut geeignet sind, aber in mancherlei Hinsicht noch Forschungs- und Verbesserungsbedarf besteht. Eine Bestandsaufnahme existierender Systeme und verwandter Arbeiten ergab, dass keiner der Ansätze alle Wünsche an ein geeignetes Datenmodell erfüllen kann.

So erschwert u. a. die Tatsache, dass die Systeme zumeist eine Dokumentensicht auf die Daten haben (d. h. einzelne Webseiten werden als einzelne Dokumente verwaltet)[3] die Möglichkeit, bestehende Daten bzw. Code wiederverwenden zu können. Daraus entstand die Idee, Webseiten in kleinere Bausteine – Objekte – zu unterteilen und so objektorientierte Konzepte mit CMS-Datenmodellen zu verknüpfen. Insbesondere das Konzept der Vererbung ermöglicht einen hohen Grad von Wiederverwendbarkeit und Redundanzvermeidung, da bereits bestehende Daten bzw. Code übernommen und erweitert bzw. spezialisiert werden können. Allerdings zeigte sich in Wob 1, dass es mit herkömmlicher Vererbung problematisch bzw. nicht möglich ist, ganze Klassen*hierarchien* abzuleiten, so dass eine erweiterte Form der Vererbung hier geeigneter schien, um u. a. neue Websites aus bestehenden zu erstellen, ohne die ursprünglichen verändern zu müssen. Diese Idee spiegelt sich auch im so genannten „open-closed-principle“([Mey88], [Mar00]) wieder:

> „Software entities (Classes, modules, functions, etc.) should be open for extension, but closed for modification.“

Das Datenmodell soll das Erstellen einer Website schnell, einfach, zuverlässig und wartbar für den Entwickler machen, wobei grundlegende Konzepte wie beispielsweise die Ausgabe in verschiedenen Formaten ebenfalls unterstützt werden sollen. Ziel der Arbeit war es also, ein Datenmodell zu entwerfen, mit dem die herausgearbeiteten Anforderungen, die in Kapitel 3.4 genauer spezifiziert werden, erfüllt werden können.

1.2. Beiträge und Einordnung der Arbeit

Im Rahmen der vorliegenden Arbeit wurden moderne, komplexe Webangebote in Hinblick auf deren Strukturen untersucht. Daraus wird abgeleitet, welche Eigenschaften das Datenmodell eines CMS, mit dem diese Daten verwaltet werden sollen, idealerweise haben sollte, um dem Entwickler (derjenige, der neue Websites erstellt) bzw. Redakteur (derjenige, der die Inhalte einer Website pflegt) die Arbeit mit dem System weitestgehend zu vereinfachen. Eine Analyse existierender CMS zeigt, dass keines der Datenmo-

[3] Diese Dokumentensicht liegt vermutlich in den oben beschriebenen Ursprüngen der statischen HTML-Seiten begründet.

delle flexibel und komplex genug ist, um alle Anforderungen abzudecken, so dass ein neues Datenmodell konzipiert wird. Die Hauptbeiträge der Arbeit sind:

1 – Das Objektmodell W2OM: Das W2OM ist ein hierarchisches, objektorientiertes Datenmodell, in dem nicht nur Klassen, sondern ganze Klassen*hierarchien* (Familien genannt) abgeleitet und spezialisiert werden können. Dieses Familienkonzept ermöglicht eine weitgehend redundanzfreie Erstellung von Webangeboten aus bestehenden heraus und vereinfacht die Entwicklung von Websites. Weiterhin wird durch das Konzept der *Daten-* und *Feld*klassen, bei dem Zustand und Verhalten der Objekte voneinander getrennt werden, eine besondere und im Kontext von CMS gut geeignete Art der Datenkapselung und -abstraktion ermöglicht. Webseiten können so feingranular (also nicht dokumentenzentriert) modelliert werden.

2 – Die funktionale Templatesprache W2L: Zur Generierung von Webseiten aus den Daten wird die W2L-Templatesprache entworfen, die auf Grund ihres funktionalen Charakters gut geeignet für die Verarbeitung hierarchischer Strukturen ist und, dank referenzieller Transparenz, effizient implementiert werden kann. Die Seiteneffektfreiheit garantiert zudem die Sicherheit, dass kein Programm ungewollte Änderungen am Zustand vornehmen kann. Ein einfaches, aber dennoch mächtiges statisches Typsystem, das u. a. einen speziellen Datentyp für XML zur Verfügung stellt, erleichtert die effiziente Generierung von Text bzw. XML zur Erzeugung der Webseiten. In Verbindung mit dem Konzept der Daten- und Feldklassen ist so eine geeignete Kapselung möglich, da die einzelnen Content-Objekte sich selbst darstellen können.

3 – Ein Konzept zur automatischen GUI-Generierung aus CMS-Daten: Ein – zwar nicht direkt mit dem Kern des Datenmodells verbundener, aber dennoch im Kontext wichtiger – Aspekt des in dieser Arbeit vorgestellten Konzepts ist das der *GUI-Generierung*, welche es ermöglicht, aus hierarchischen (W2OM-) Schemadaten automatisch Eingabemasken zur Pflege der Daten des CMS generieren zu können. Dadurch wird dem Entwickler die Aufgabe abgenommen, eine solche Eingabemaske zu implementieren.

Zudem wird eine *medienneutrale Zwischensprache* konzipiert, die die Ausgabe der Daten in verschiedenen Formaten erleichtert, indem die Erzeugung der unterschiedlichen Formate von der Anwendungslogik in den Templates entkoppelt wird. Einige Aspekte des theoretischen Gesamtkonzepts wurden prototypisch (als „proof of concept“) implementiert.

Neu ist die Kombination von Objektorientierung, funktionaler Programmierung sowie eines speziellen Vererbungskonzepts im Rahmen von CMS und semistrukturierten Daten. Dazu werden bewährte Ansätze aus anderen Kontexten angewandt sowie adaptiert

und mit neuen Ideen kombiniert, wodurch ein Gesamtkonzept entsteht, das als mehr als die Summe seiner Teile gesehen werden kann. Die Aspekte des Konzepts unterstützen eine schnelle und einfache Entwicklung von CMS-Anwendungen und erlauben es, sowohl neue Websites schnell von bestehenden ableiten als auch XML bzw. Texte effizient generieren zu können. Die hier entwickelten Konzepte stellen eine solide Grundlage für Content Management Systeme dar und ermöglichen eine elegante Datenmodellierung. Dabei stellt das Datenmodell eine einheitliche Datenabstraktionsschicht zur Verfügung und ist eine integrierte Lösung und kein „Patchwork“ verschiedener Technologien, wie es bei existierenden Systemen oftmals der Fall ist.

Die Arbeit versteht sich als übergreifender Ansatz, der thematisch nicht eindeutig einem Gebiet (einer Community) zuzuordnen ist, vielmehr versucht sie, unterschiedliche Aspekte aus verschiedenen Bereichen auf nichttriviale Weise miteinander zu verknüpfen, zu erweitern und deren Konzepte in einen neuen, bislang kaum untersuchten Kontext zu setzen. Abbildung 1.4 illustriert, in welcher Schnittmenge verschiedener Themengebiete die Arbeit angesiedelt ist.

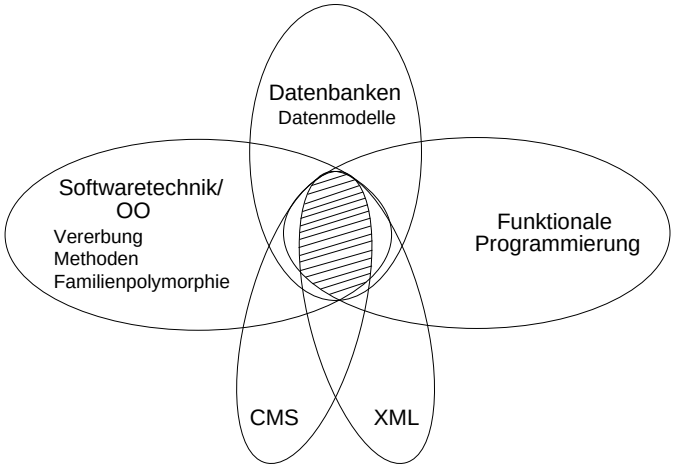


Abbildung 1.4.: Einordnung der Arbeit

Anzumerken ist hierbei, dass es sich bei der Thematik der vorliegenden Arbeit nicht um ein einfaches Software Engineering-Problem handelt, bei dem, ausgehend von einer Anforderungsanalyse, *ein* Modell und *eine* Implementierung entwickelt wird. Stattdessen stellt das hier hergeleitete Modell einen allgemeinen Rahmen (*Framework*) dar, der als Grundlage für konkrete CMS dienen kann. Das Modell bzw. System ist dabei flexibel erweiterbar und kann jederzeit auf eine konkrete Anwendung angepasst werden. Des Weiteren sei darauf hingewiesen, dass im Fokus der Arbeit das Datenmodell steht und

nicht die Persistenz- bzw. Datenbankschicht, die hier nur am Rande betrachtet wird.

1.3. Struktur der Arbeit

Die Arbeit gliedert sich wie folgt: nach einem Überblick über die benötigten Grundlagen in Kapitel 2 beschreibt Kapitel 3, was Content Management Systeme sind und welche Daten mit diesen verwaltet werden – hieraus werden die Anforderungen an die Datenmodelle der CMS abgeleitet und anschließend untersucht, ob bzw. inwiefern existierende Systeme diese Anforderungen erfüllen. Kapitel 4 beschreibt das im Rahmen der vorliegenden Arbeit entwickelte Konzept und begründet die bei der Konzeption getroffenen Entscheidungen. Die beiden daran anschließenden Kapitel spezifizieren die verschiedenen Aspekte des Modells sowie der konzipierten Sprache. In Kapitel 7 werden alternative Lösungansätze bzw. verwandte Arbeiten untersucht und diskutiert, aus welchen Gründen diese im Kontext der Arbeit eher ungeeignet sind. Kapitel 8 zeigt, wie das konzipierte Modell prototypisch im Wob 2-System umgesetzt wird und wie dessen Architektur aufgebaut ist, zudem wird der entwickelten Ansatz, aus dem Datenmodell automatisch Datenpflegemasken generieren zu können, beschrieben. Abschließend folgen die Zusammenfassung sowie ein Ausblick auf mögliche Erweiterungen bzw. anschließende Forschungsarbeiten.
Anmerkungen:

- Sämtliche in dieser Arbeit vorkommenden URLs wurden am 10. Januar 2007 auf Gültigkeit geprüft. Da diese sich z. T. leider schnell ändern können, mag es sein, dass zum Zeitpunkt des Lesens eine URL nicht mehr korrekt ist.
- Für Klassendiagramme wurde in dieser Arbeit die UML-Notation (Unified Modelling Language) benutzt (siehe u. a. [PJ99]).
- Häufig verwendete Akronyme werden im Akronymverzeichnis (Seite 162 ff.) aufgelistet.

2. Grundlagen

In diesem Kapitel werden die Grundlagen vorgestellt, auf denen die vorliegende Arbeit aufbaut. Im einzelnen sind dies: semistrukturierte Daten und XML-Technologien, objektorientierte Konzepte, Datenmodelle sowie Programmierparadigmen.

2.1. Semistrukturierte Daten

2.1.1. Kategorisierung von Daten

Daten können in drei verschiedene Kategorien eingeteilt werden: unstrukturierte, strukturierte und semistrukturierte Daten. In der Literatur finden sich unterschiedliche Definitionen (siehe u. a. [ABS99], [Abi97], [Car01]), deren gemeinsamer Nenner wie folgt zusammengefasst werden kann:

1. **Unstrukturierte Daten** zeichnen sich dadurch aus, dass sie ungetypt (vgl. Abschnitt 2.3.3) sind, keine Reihenfolge aufweisen und in Bezug auf deren Struktur bzw. Art nicht „vorhersehbar“ sind. Sie sind nicht durch ein Schema, mit dem die logische bzw. physische Struktur spezifiziert werden kann, beschreibbar. Typische Beispiele sind binäre Daten (bei Datenbanken: BLOBs, *binary large objects*) wie Musik, Bilder oder (unstrukturierte) Texte.

2. **Strukturierte Daten** sind in semantischen Einheiten (Entitäten) organisiert, wobei ähnliche Entitäten in Gruppen (z. B. Relationen oder Klassen) gruppiert werden. Entitäten der gleichen Gruppe besitzen die gleichen Attribute[1], welche das gleiche Format haben und alle vorhanden sind. Die Daten sind getypt und durch ein striktes Schema beschreibbar, in der Regel ist eine Eingabe von Daten erst nach der Erstellung eines Schemas möglich. Das klassische Beispiel hierfür sind relationale Datenbanken, bei denen die Daten einem festen, strikten Schema genügen (müssen).

3. **Semistrukturierte Daten** sind ebenfalls in semantischen Einheiten organisiert und in ähnliche Entitäten gruppiert. Der Unterschied zu strukturierten Daten liegt jedoch darin, dass die Entitäten derselben Gruppe unterschiedliche Attribute haben und die Attribute in unterschiedlicher Reihenfolge auftreten können. Ebenso

[1] Attribute definieren und beschreiben Merkmale einer Entität.

sind nicht alle Attribute erforderlich und der Typ der Attribute kann unterschiedlich sein. Man spricht auch von einem „lockeren“ Schema oder „softly typed data“.

Diese Eigenschaften fasst Abiteboul in [Abi97] wie folgt zusammen:

> „Semi-structured data is data that is neither raw data, nor very strictly typed, as in conventional database systems.“

Eine häufig anzutreffende Beschreibung definiert semistrukturierte Daten als „schemalos“ bzw. „selbstbeschreibend“ (z. B. [ABS99], [Bun97]), d. h. ohne separate Beschreibung der Typen bzw. Strukturen der Daten – die Beschreibung der Strukturen und Typen ist in Metadaten bzw. in den Elementnamen enthalten.

Die wesentlichen Vorteile semistrukturierter Daten sind:

1. Durch die Semistrukturiertheit entsteht eine größere Flexibilität beim Erfassen von Daten - so kann man z. B. einen Namen hierarchisch in Vornamen und Nachnamen aufteilen („Name besteht aus Vorname und Nachname“) oder als kompletten Namen eingeben, was z. B. bei einer relationalen Datenbank nicht ohne größeren Aufwand möglich wäre. Generell können hierarchische Daten gut modelliert werden.
2. Die Datenstruktur ist sowohl von Menschen als auch von Maschinen lesbar.
3. Die Datentypen müssen nicht explizit festgelegt werden, da die Typstruktur von semistrukturierten Daten selbstbeschreibend ist und keiner Zusatzinformation bedarf, wie es beispielsweise in Datenbanksystemen oder Programmiersprachen oft erforderlich ist.

Wie u. a. in Abschnitt 3.2 gezeigt wird, lassen sich Websites und Webseiten, bedingt durch ihre (Dokument-)Struktur, in der Regel hervorragend durch semistrukturierte Daten beschreiben. Weitere Details zu semistrukturierten Daten sind u. a. in [Abi97] und [Bun97] zu finden.

2.1.2. Darstellung von semistrukturierten Daten

Geeignete und daher oft verwendete Darstellungen semistrukturierter Daten sind ein Graph bzw. Baum oder eine serialisierte Liste.

Graph-basierte Darstellung

Zunächst einige Grundbegriffe aus der Graphentheorie (siehe u. a. [ABS99], [Tur96]):

Ein *gerichteter Graph* $G = (V, E)$ ist ein Tupel bestehend aus einer Menge von Knoten V (dargestellt durch Kreise) und einer Menge von Kanten $E \subseteq V \times V$ (dargestellt durch Pfeile zwischen den Knoten). Die Kanten in E seien geordnet.

Ein *Pfad* ist eine Folge von hintereinander liegenden Kanten durch einen Graphen, also eine endliche Folge $v_1, ..., v_n$ von Knoten, so dass $\forall_{1 \leq k < n} (v_k, v_{k+1}) \in E$. In diesem Fall *verbindet* der Pfad v_1 und v_n. Jedes Paar ist *Element* des Pfades. Unter einem *Zyklus* versteht man einen Pfad von einem Knoten zu sich selbst, d. h. $v_n = v_1$, $n \geq 1$. Enthält ein Graph einen Zyklus, so nennt man ihn *zyklisch*, enthält er keinen Zyklus, nennt man ihn *azyklisch*.

Sei $G = (V, E)$ und $v, v' \in N$. Ist $(v, v') \in E$, dann heißt v *direkter Vorgänger* oder *Vater* von v' und v' *direkter Nachfolger* oder *Kind* von v. (auch als $v \rightarrow v'$ notiert.) Falls ein Pfad von v nach v' führt, so heißt v *Vorgänger* bzw. *Vorfahre* von v' und v' *Nachfolger* bzw. *Nachfahre* von v (auch als $v \rightarrow^* v'$ notiert.)

Ein *gerichteter azyklischer Graph* (directed acyclic graph, DAG) ist ein gerichteter Graph $G = (V, E)$, der keine Zyklen enthält. Ein Knoten $r \in V$ eines DAG heißt *Wurzel*, falls es keine auf ihn gerichteten Kanten gibt. Hat ein DAG nur eine Wurzel, so heißt er *Wurzelgraph*. Man kann in diesem Fall von *seiner* Wurzel sprechen.

Ein *Baum* ist ein zusammenhängender Wurzelgraph, in dem zu jedem Knoten genau ein Pfad von der Wurzel aus führt. *Zusammenhängend* besagt, dass es keinen Knoten geben darf, der ohne Verbindung zu den anderen Knoten im Baum existieren darf. Ein *endlicher* Baum ist ein Baum, der keinen unendlichen Pfad besitzt.

Weiterhin gibt es knoten- und kantenbeschriftete Bäume. Seien Σ_V und Σ_E endliche Alphabete von Knoten- bzw. Kantenbeschriftungen und $l_v : V \rightarrow \Sigma_V$ bzw. $l_e : E \rightarrow \Sigma_E$ totale (Beschriftungs-)Funktionen, so heißt $G = (V, E, l_v)$ *knotenbeschrifteter* und $G = (V, E, l_e)$ *kantenbeschrifteter* Baum.

Grundsätzlich lassen sich knotenbeschriftete und kantenbeschriftete Bäume äquivalent verwenden und können ineinander überführt werden: Um beispielsweise von einer knoten- zu einer kantenbeschrifteten Darstellungsform zu gelangen, kann man die Beschriftungen aus den Knoten in der darüber liegende Kante ziehen - wobei man bei der Wurzel eine zusätzliche eingehende Kante erstellen muss.

Hängt die Zahl der Kinder nicht von der Beschriftung des Vaterknotens ab, spricht man von einem *unbeschränkten* Baum.

Semistrukturierte Daten lassen sich in kanonischer Weise als endliche, unbeschränkte sowie beschriftete Bäume auffassen, bei denen die Daten in Blättern und inneren Knoten gespeichert sind[2].

Serialisierte Darstellung

Eine serialisierte Darstellung stellt eine konkrete Syntax von semistrukturierten Daten dar und wird u. a. zur Übertragung/Datenaustausch eingesetzt; in den meisten Fällen wird eine textuelle Listenform verwendet. Eine *Markup-* oder auch *Auszeichnungs*sprache

[2] In der Regel werden dabei in den inneren Knoten strukturelle Daten wie beispielsweise `section`, `title` o. ä. und Inhalte in den Blättern gespeichert.

dient dazu, zum eigentlichen textuellen Inhalt eines Dokuments strukturelle Informationen hinzuzufügen (siehe z. B. [WR99]); demnach eignen sich Markupsprachen gut, um semistrukturierte Daten zu beschreiben. Markup-Sprachen lassen sich grob in zwei Gruppen unterteilen:

Prozedurale Markup-Sprachen beschreiben das Verfahren zur *Darstellung* der Daten. Beispiele hierfür sind TEX, PDF oder Postscript.

Deskriptive Markup-Sprachen beschreiben die *Syntax* von Daten. Beispiele hierfür sind LATEX[3] sowie SGML (Standard Generalized Markup Language) und XML und darauf basierende Sprachen.

Die Idee zu Markup-Sprachen entstand bereits Ende der 1960er Jahre, als William Tunnicliffe und Stanley Rice den Ansatz des sog. *generic coding* vorschlugen, d. h. die Auszeichnung von Texten mit beschreibenden *Tags* (engl. für Symbol, Zeichen, Markierung) sowie die Trennung von Inhalt und Formatierung, auch bekannt als *Trennung der Belange* („Separation of Concerns“, s. Abschnitt 3.1.1).

Mittlerweile wird der Begriff „Markup-Sprache“ zumeist synonym zu SGML/XML-basierten Sprachen benutzt. Die am häufigsten anzutreffende serialisierte Darstellung von semistrukturierten Daten ist die XML-Notation, bei der interne Knoten des zugehörigen Baums als *Elemente* bezeichnet und von einem Start- und einem (dazu passenden) End-Tag umschlossen werden. Elemente können weiterhin *Attribute*[4] als Paar von Name und Wert besitzen und weitere Elemente und/oder Text beinhalten. Listing 2.1 zeigt einige Beispiele: In den Zeilen 1-3 wird ein Element `a` (mit Start-tag `<a>` und End-tag `</a>` und Attribut `attr` mit Wert `wert`) definiert, in Zeile 4 ein leeres Element `b` und in den Zeilen 5-7 zwei ineinander verschachtelte Elemente `c` und `d`. Der Inhalt des Elements `c` wird auch als *mixed content* bezeichnet, da er sowohl Text als auch ein Element enthält.

Abbildung 2.1 zeigt die graphbasierte und die serialisierte Darstellung (in XML-Syntax) an einem einfachen Beispiel.

2.2. XML-Technologien

Es existieren viele verschiedene Technologien im Umfeld von XML, von denen ein Großteil vom World Wide Web Consortium (W3C[5]) entworfen wurde bzw. weiterentwickelt wird. In diesem Abschnitt werden die für die vorliegende Arbeit benötigten Technologien und Sprachen vorgestellt.

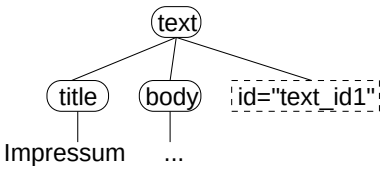
[3] In TEX muss ein Großteil der Inhalte genau gesetzt werden, d. h. die Darstellung der Daten wird vom Benutzer beschrieben; durch den Einsatz der Makropakets LATEX muss der Benutzer hauptsächlich die Inhalte beschreiben und das System regelt den genauen Textsatz in weiten Teilen.

[4] XML-Attribute entsprechen *properties* in anderen Datenmodellen (vgl. hierzu auch [ABS99])

[5] `http://www.w3c.org`

```
<a attr="wert">
  test
</a>
<b/>
<c>
  noch ein <d>test</d>
</c>
```

Listing 2.1: XML-Syntax




```
<text id="text_id1">
  <title>Impressum</title>
  <body>...</body>
</text>
```

Abbildung 2.1.: Darstellung semistrukturierter Daten
a) graphbasiert und b) serialisiert

2.2.1. SGML

1969 erweiterte ein Team um Charles Goldfarb den Ansatz des generic coding und entwickelte die *Generalized Markup Language* (GML, [GMP70]). Im Laufe des darauffolgenden Jahrzehnts wurde GML wiederum erweitert, um schließlich 1980 als sog. working draft und im Jahre 1986 als ISO-Standard 8879 [ISO86] veröffentlicht zu werden.

[BM00] beschreiben SGML als

> „komplexe Metasprache zur Generierung von Auszeichnungssprachen, die seit 1986 standardisiert ist. Ziel war es unter anderem, die Unabhängigkeit der Auszeichnung von der Ausgabe zu gewährleisten."

Mit dem Boom des WWW Anfang der 1990er Jahre konnte sich HTML als bekannteste SGML-basierte Sprache durchsetzen.

2.2.2. XML

Da SGML übermäßig komplex ist, wurde XML als Teilmenge von SGML 1996 erstmals vorgestellt [BSM96] und 1998 erstmals als Recommendation des W3C verabschiedet. Die Spezifikation liegt inzwischen in der dritten Überarbeitung[6] vor.

XHTML ist die Reformulierung von HTML in XML; Abbildung 2.2 zeigt die Beziehungen der vorgestellten Sprachen untereinander. Sowohl SGML als auch XML sind sog.

[6] http://www.w3c.org/TR/2004/REC-xml-20040204

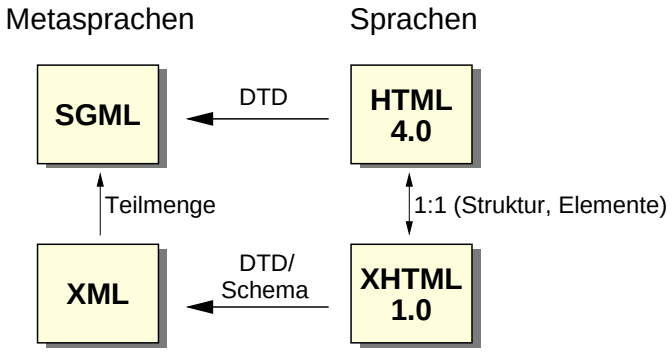


Abbildung 2.2.: SGML, (X)HTML und XML

Metasprachen, d. h. Sprachen, mit denen man andere Sprachen definieren kann, während HTML und XHTML konkrete (Dokumenten-)Beschreibungssprachen sind, welche durch eine DTD (Document Type Definition) bzw. ein Schema definiert werden (s. hierzu die Abschnitte 2.2.3 und 7.2). Es existieren hunderte von konkreten Beschreibungssprachen, auch XML-*Anwendungen* genannt, für die unterschiedlichsten Bereiche, z. B.:

- Scalable Vector Graphics (SVG): 2D-Grafiken
- Wireless Markup Language (WML): (Hyper-)Textdokumente für Mobile Endgeräte
- XML Metadata Interchange (XMI): Austausch von Metadaten für modelling tools (z. B. UML)
- Chemical Markup Language (CML): chemische Formeln
- Electronic Business XML Initiative (ebXML): B2B-Daten
- Mathematical Markup Language (MathML): mathematische Formeln
- User Interface Markup Language (UIML): Benutzeroberflächen

Prinzipiell bietet XML eine lineare Syntax für unbeschränkte, geordnete und beschriftete Bäume. Für XML-Dokumente wird in der Regel die knotenbeschriftete Form (vgl. Abschnitt 2.1.2), z. B. ein SSD-Graph oder das Object Exchange Model (OEM), vgl. [ABS99], verwendet. Syntaktisch sind XML-Dokumente wohlgeformte Klammerausdrücke; XML-Dokumente, die korrekt verschachtelt sind, nennt man auch *wohlgeformt*.

Ein XML-Dokument, auch XML-*Instanz* genannt, beginnt mit einer XML-Deklaration und einer (optionalen) DOCTYPE- bzw. Schema-Definition und besitzt genau ein Wurzelelement. Außerhalb dieses Wurzelementes dürfen weder Text noch andere Elemente auftreten. Als XML-*Fragment* bezeichnet werden ein oder mehrere XML-Elemente, die

Teil einer XML-Instanz sind und das somit nicht zwingend nur ein Wurzelelement haben muss. Ein XML-Fragment ist dann wohlgeformt, wenn durch die Ergänzung eines umschließenden (Wurzel-)Elements ein wohlgeformtes XML-Dokument entstehen würde. Man unterscheidet zwischen dokumentenzentrierter und datenzentrierter XML-Nutzung:

dokumentenzentriert: Dokumenten- oder auch präsentationszentrierte Nutzung zeichnet sich durch folgende Aspekte aus:

- Heterogene Verwendung von XML-Elementen, d. h. viele unterschiedliche Subelemente sowie eine komplexe Schachtelung von Elementen.
- Grobgranular, d. h. größere unmarkierte, nicht durch Elemente/Tags ausgezeichnete Dokumentteile.
- *Mixed content*: Elemente können sowohl textuelle Inhalte als auch Subelemente enthalten.
- Die Ordnung der Elemente ist signifikant (z. B. die Reihenfolge der Kapitel in einem Buch).

datenzentriert: Bei datenzentrierter Nutzung hingegen sind die folgenden Merkmale erkennbar:

- Hohe Regularität der Daten.
- Homogene Verwendung von XML-Elementen: Elemente enthalten eine Kollektion von gleichartigen Subelementen.
- Die Komplexität der „Datensätze“ ist meistens beschränkt:
  - Geringe Schachtelungstiefe.
  - Schachtelungen gleichnamiger Elemente werden vermieden.
- Wenige oder keine optionale Substrukturen.
- Textdaten nur innerhalb von Blattelementen.
- Dokumentstruktur ähnelt Datensätzen oder Kollektionen von Datensätzen einer konventionellen Datenbank.
- Im Gegensatz zu mixed content nur bzw. hauptsächlich *element content*, d. h. die Kinder eines Elements bestehen entweder nur aus Elementen oder nur aus Textknoten.
- Feingranular, d. h. die Daten sind in kleiner Granularität durch Tags ausgezeichnet.
- Die Daten sind getypt.
- Die Elementordnung ist nicht signifikant.

Bezug nehmend auf Abschnitt 2.1 entspricht die dokumentenzentrierte Sichtweise am ehesten den semi- und unstrukturierten Daten, und die datenzentrierte Sichtweise den strukturierten Daten. Während bei SGML bzw. XML ursprünglich die Dokumentenspeicherung im WWW (HTML-Seiten) im Vordergrund stand, gewinnt zunehmend die Speicherung von reinen Daten an Bedeutung, was sich nicht zuletzt auch in der weiten Verbreitung von reinen XML-Datenbanken wie beispielsweise eXist[7] [Mei02] wiederspiegelt.

2.2.3. Schemasprachen für XML

Ein *Schema* ist ein formales Modell der Struktur von Daten und wird in der Regel als Konstruktionsvorlage, zur Dokumentation oder zum Datenbankdesign benutzt. Ein Schema beschreibt eine Sprache sowie ein zugehöriges Typsystem. Der erste Ansatz, eine Schemasprache für semistrukturierte Daten und speziell XML zu spezifizieren, war die DTD (Document Type Definition) (vgl. dazu Abbildung 2.2 auf Seite 12), welche aber nach und nach von XML Schema abgelöst wird.

Wohlgeformte (XML-)Dokumente, die eine DTD (bzw. ein Schema) besitzen und den dort spezifizierten Einschränkungen genügen, heißen *gültig* (valid). Ein gültiges Dokument nennt man auch *Instanzdokument* des zugehörigen Schemas. Schemasprachen definieren eine Grammatik zur Beschreibung von Klassen von XML-Dokumenten; für ein Schema S ist die *Sprache* $\mathfrak{L}(S)$ definiert als die Menge aller XML-Dokumente, die gültig sind bzgl. S.

2.3. Objektorientierte Konzepte

Objektorientiertes Design erhebt oft den Anspruch, bestehende Systeme nicht durch Manipulation existierenden Codes, sondern allein, oder zumindest überwiegend, durch Hinzufügen neuer (Sub-) Klassen an neue oder veränderte Anforderungen anpassen zu können (vgl. hierzu das „open-closed-principle“ auf Seite 3). Hierzu versucht man, einen hohen Grad von Redundanzfreiheit und Re-Use zu erreichen. Im Folgenden werden die essentiellen Aspekte von Objektorientierung vorgestellt.

2.3.1. Objekte und Objektorientierung

Für den Begriff „Objektorientierung (OO)“ bzw. „Objektorientierte Programmierung (OOP)“ gibt es keine einheitliche Definition, vielmehr existieren verschiedene OO-Konzepte, die als mehr oder weniger essentiell für OO bzw. OOP erachtet werden und welche im folgenden Abschnitt näher beschrieben werden.

Dabei sind die in den meisten Definitionen auftauchenden Konzepte (siehe u. a. [DD96] oder [Pie02]) (Daten-)Kapselung und Vererbung. Zum Teil wird differenziert zwischen

[7] `http://www.exist-db.org`

strukturell-objektorientierten Systemen, die nur Gruppierung und Identität unterstützen sowie *verhaltens*-objektorientierten Systemen, die zusätzlich noch Kapselung und Prozeduren zur Verfügung stellen.

Ein Objekt lässt sich beschreiben als eine Datenstruktur, die einen Zustand (d. h. Daten bzw. Variablen) und Verhalten (in Form von Funktionen, Methoden bzw. Operationen) darauf kapselt:

> „An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.“ [Boo94]

> „[...] an *object* is just a data structure encapsulating some internal *state* and offering access to this state to clients via a collection of *methods*. The internal state is typically organized as a number of mutable *instance variables* (or *fields*) that are shared among the methods and inaccessible to the rest of the program.“ [Pie02]

Nach den obigen Definitionen verfügt ein Objekt über:

Identität: Jedes Objekt ist eindeutig von anderen Objekten zu unterscheiden. Diese Unterscheidung ist wertunabhängig, d. h. zwei verschiedene Objekte mit gleichem Zustand sind wohlunterscheidbar. Im Gegensatz dazu stehen beispielsweise wertebasierte Systeme wie relationale Datenbanken, die keine inhärenten IDs haben und die Identität über künstliche, nicht fachliche Primärschlüssel „simulieren“ müssen. Objektorientierte Systeme sind dagegen identitätsbasiert, die Objektidentität wird durch einen vom System vergebenen internen, eindeutigen und unveränderbaren Schlüssel gebildet. (Vgl. hierzu auch [Cat91], S. 85 u. S. 151)

Zustand: Der Zustand eines Objektes umfasst seine Struktur sowie seine aktuelle Wertbelegung. Die Struktur besteht aus einer Menge von Attributen (auch Properties genannt). Ein solches Attribut hat einen Datentyp, der insbesondere auch objektwertig sein kann. Damit kann ein Objekt andere Objekte referenzieren.

Verhalten: Das Verhalten eines Objektes ist die Gesamtheit seiner Funktionalität, welche mittels Methoden definiert wird. Eine Methode besteht aus ihrer Signatur sowie aus ihrer Implementierung. Eine *Signatur* definiert neben dem Namen der Methode deren Aufrufparameter, ein *Deskriptor* zusätzlich noch den Rückgabetyp. Die Implementierung einer Methode erfolgt mit den üblichen, in der Regel imperativen Programmiersprachkonstrukten. Man kann Methoden in statische Klassenmethoden, die mit Klassen assoziiert sind und dynamische Instanz- oder Objektmethoden, die mit Objektinstanzen assoziiert sind, differenzieren.

2.3.2. Essentielle objektorientierte Konzepte

Die Definitionen der objektorientierten Konzepte, die in diesem Abschnitt beschrieben werden, stammen u. a. aus [Weg90], [PJ99] sowie [Boo91]. Die im Folgenden vorgestellten Konzepte *Klasse, Kapselung, Vererbung* und *Polymorphie* sind essentiell für OO/OOP.

Klasse: Ähnlich wie bei dem Begriff der Objektorientierung gibt es auch keinen einheitlich definierten Klassenbegriff.

Eine Klasse lässt sich definieren als eine Beschreibung einer Menge gleichartiger Objekte mit

- gleichen Methoden (Verhalten),
- gleicher Implementierung,
- gleichen Attributen sowie
- (potenziell) unterschiedlichen Attribut-Werten (Zustand).

Während eine Klasse eine abstrakte Darstellung eines Objekts der realen Welt[8] ist, ist eine Instanz einer Klasse dessen konkrete Darstellung. Im Bereich der OOP wird der Begriff Instanz zumeist synonym zum Begriff Objekt benutzt.

Instanzvariablen definieren die Eigenschaften eines Objekts. Die Klasse definiert die Art der Eigenschaft, und jede Instanz speichert einen eigenen Wert für die jeweilige Eigenschaft.

Klassen werden zum einen instanzgenerierend (d. h. zur Erzeugung von Objekten nach einer „Schablone“) und zum anderen aus Gründen der Wiederverwendbarkeit (Re-Use), z. B. in Verbindung mit Vererbung, eingesetzt.

Kapselung (encapsulation) und Information-Hiding: Mit Kapselung wird das Gruppieren zusammengehöriger Daten und Funktionen in einer Einheit (Klasse) bezeichnet.

Information-Hiding ist eine Erweiterung der Kapselung insofern, als zwischen einer öffentlichen und einer privaten Objektschnittstelle unterschieden wird; von außen kann nur auf die als öffentlich deklarierten Attribute und Methoden des Objekts zugegriffen werden. Die privaten Anteile sind nach außen unsichtbar; man spricht daher auch von „Sichtbarkeit“ von Methoden und Attributen. Auf diese Weise kann die konkrete Implementierung des Objekts versteckt werden, wodurch ein hohes Maß an Implementierungsunabhängigkeit erreicht werden kann.

Das Abstrahieren von Implementierungsdetails wird in der OOP auch als *Abstraktion* bezeichnet.

[8] bzw. der Charakteristik einer Menge von Objekten

Vererbung (Inheritance) und Polymorphie (Polymorphism): sind im Kontext der vorliegenden Arbeit von besonderem Interesse und werden daher in den Abschnitten 2.3.4 sowie 2.3.5 ausführlich untersucht.

2.3.3. Typisierung

Ein *Typ* in einer Programmiersprache beschreibt (mögliche) Wertemengen von Objekten (Beispiele: `int`, `Object`, `LinkedList`). *Typisierung* (typing) meint die Annotation von programmiersprachlichen Elementen mit dem Typ des Ausdrucks (Beispiele: `int i`, `String getName()`), durch die Typannotationen werden also Bedingungen an die Variablen, Objekte bzw. Methoden geknüpft. In Bezug auf Methoden bedeutet dies, dass, wenn die Parameter den korrekten Typ haben, das Ergebnis der Methode den deklarierten Ergebnistyp hat. Eine Variable enthält nur Werte ihres Typs (bzw. eines substituierbaren Typs). Ein Programm heißt *typkorrekt*, wenn alle derartigen Typbedingungen erfüllt sind. Ein *Typfehler* tritt auf, wenn einer Variable ein Wert eines falschen Typs zugewiesen oder eine Methode mit nicht typkonformen Parametern aufgerufen wird. Eine Sprache wird *typsicher* genannt, wenn zur Laufzeit eines typkorrekten Programms keine Typfehler auftreten können.

Anzumerken ist, dass ein Typsystem keinesfalls zwingend für eine Programmiersprache ist, jedoch können hierdurch (Laufzeit-)Fehler vermieden bzw. erkannt werden. Beispiele ungetypter Sprachen sind Lisp, Prolog oder die Wob 1-Sprache, während Java oder C/C++ getypt sind.

Typinferenz

Unter Typinferenz versteht man das Folgern eines bestimmten Typs für einen gegebenen Ausdruck, obwohl der Typ nicht explizit angegeben wurde. Stattdessen wird unter Zuhilfenahme eines Algorithmus' auf den richtigen Typen geschlossen (zur Übersetzungszeit). Beispiel:

```
int x=2;
float y=0.7;
z1 = x + 4;
z2 = y * x;
```

Eine Sprache, die Typinferenz unterstützt, würde beim Übersetzen den Typ `int` für `z1` und den Typ `float` für `z2` inferieren.

Statische und dynamische Typisierung

Statisch typisierte Sprachen überprüfen Typ-Konformität zur Übersetzungszeit, während dynamisch typisierte Sprachen dies erst zur Laufzeit tun. Mit sorgfältig gewählten

Typen kann man ein Programm lesbarer machen und gibt insbesondere dem Compiler die Möglichkeit, Unstimmigkeiten aufzudecken. Statisch typisierte Sprachen unterscheiden sich allerdings in der Vielfalt der Typen. Je feiner unterschieden wird, desto mehr Sicherheit ist möglich und um so mehr Sorgfalt verlangt die Typenwahl. Bei dynamischer Typisierung sind dagegen durch weniger Typinformationen die Schnittstellen einfacher, was u. a. die Anwendung und Refactoring erleichtert.

(Sub)Typ vs. (Sub)Klasse

Auf Grund der sich ähnelnden Definitionen von Typ und Klasse stellt sich die Frage, worin die Unterschiede zwischen diesen beiden Begriffen liegen:

Ein Typ definiert nur die Schnittstelle, d. h. eine Menge von Namen und Typen der Operationen, die auf Objekten diesen Typs definiert sind, während eine Klasse auch die eigentlichen Implementierungen dieser Methoden bereitstellt. Einfache Datentypen (wie beispielsweise `int` in Java) besitzen keine Methoden, sind also auch nur Typ und keine Klasse (Im Unterschied zur `Integer`-Klasse). Typen haben einen „Typüberprüfungscharakter“, während Klassen einen instanzgenerierenden Charakter haben.

Ähnlich dem Vererbungskonzept bei Klassen (Super- und Subklasse, vgl. Abschnitt 2.3.4) gibt es auch das Konzept des Subtyps. In den meisten Sprachen erzeugen Subklassen auch Subtypen; in so genannten Prototyp-Sprachen (z. B. Sather) ist dies nicht der Fall, Vererbung ist dann nur Wiederverwendung von Programmcode.

2.3.4. Vererbungskonzepte

Ein weiteres Mittel zur Abstraktion und Erhöhung der Wiederverwendbarkeit in der objektorientierten Programmierung ist die Vererbung, bei der zwischen Einfach- und Mehrfachvererbung (single bzw. multiple inheritance) sowie erweiterten Vererbungskonzepten unterschieden wird. Der Grundgedanke ist dabei, dass eine Klasse Verhalten und Datenstrukturen einer oder mehrerer anderer Klassen erben und weiter verfeinern bzw. erweitern kann. Die Klasse, von der geerbt wird, nennt man *Basis-*, *Ober-* oder *Super*klasse, die erbende Klasse *abgeleitete*, *Unter-* oder *Sub*klasse. Im Folgenden wird die Notation `A <: B` (bzw. `B :> A`) verwendet, wenn A eine Subklasse von B ist.

Wichtig im Kontext der Vererbung ist das Konzept der Selbstreferenz (`self` oder `this`), die in den Methoden der Klasse verwendet werden kann und die auf die Objektinstanz des gerade aktuellen Objekts verweist. Des Weiteren kann (üblicherweise über das Schlüsselwort `super`) auf die Superklasse der aktuellen Klasse zugegriffen werden.

Einfachvererbung Im Falle der Einfachvererbung hat die Subklasse genau eine Superklasse – die Subklasse ist eine *Spezialisierung* der Superklasse und die Superklasse eine *Generalisierung* der Subklasse. Die beiden Klassen stehen in einer so genannten `ist_ein`

(`is_a`)-Beziehung. Beispiel: eine Katze `ist_ein` Tier, Heapsort `ist_ein` Sortieralgorithmus.

Dabei gilt das Liskov'sche Substitutionsprinzip [LW93]: Objekte von Subklassen können jederzeit anstelle von Objekten ihrer Superklasse(n) eingesetzt werden: sei S ein Subtyp von T, dann können Objekte vom Typ T in einem Programm durch Objekte vom Typ S ersetzt werden, ohne dass sich dadurch die Eigenschaften des Programms (wie beispielsweise die Korrektheit) ändern.

Wenn eine Methode einer Klasse in einer ihrer Subklassen neu definiert wird, bezeichnet man dies als Überschreiben (overriding, nicht zu verwechseln mit dem Überladen im Kontext von Polymorphie, s. u.)

Mehrfachvererbung Mehrfachvererbung liegt vor, wenn eine Klasse Attribute und Methoden von mehr als einer Klasse erbt. Beispiel: Ein Amphibienfahrzeug ist sowohl ein Land- als auch ein Wasserfahrzeug. Im Falle der Mehrfachvererbung kann das so genannte Diamantenproblem (siehe z. B. [Sny87]) auftreten, wenn dasselbe Attribut (bzw. dieselbe Methode) aus mehr als einer Klasse geerbt wird, mehr dazu im Folgenden.

Vererbungsgraph Unter einem *Vererbungsgraph* versteht man einen gerichteten Graphen $G = (V, E)$ mit Knoten V und Kanten E, $E \subseteq V \times V$ und $(v_1, v_2) \in E$ gdw. $v_1 <: v_2$. Bei Sprachen mit Einfachvererbung ist der Vererbungsgraph ein Baum (Abbildung 2.3a), im Falle von Mehrfachvererbung ein (potenziell zyklischer) Graph, es können also mehrere Pfade zwischen zwei Klassen im Vererbungsgraphen existieren (Abbildung 2.3b).

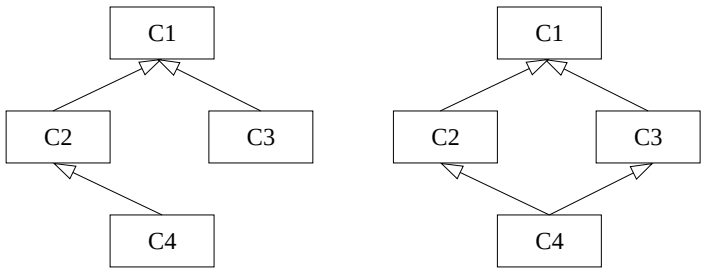


Abbildung 2.3.: Vererbungsgraph für
a) Einfach- und b) Mehrfachvererbung

Namensauflösung und Diamantenproblem Unter *Namensauflösung* (Name-Lookup) versteht man das Auffinden der für das aktuelle Objekt gültigen Methodenimplementierung bzw. Attribute. Im Falle der Einfachvererbung muss hierzu lediglich im Vererbungsgraph die jeweils nächsthöhere Klasse besucht werden, im Falle der Mehrfachvererbung kann diese Auflösung ggf. nicht eindeutig sein; wenn in Abbildung 2.3b sowohl in Klasse C2 als auch in C3 eine Methode gleichen Namens überschrieben würde, wäre

in Klasse C4 nicht eindeutig bestimmbar, welche Implementierung zu verwenden ist. Diese Problematik bezeichnet man auf Grund der Form des Vererbungsgraphen auch als „Diamantenproblem“. Mögliche Lösungen des Diamantenproblems sind:

- Ein generelles Verbot von Konflikten, d. h. von Mehrdeutigkeiten.
- Einschränkungen der Mehrfachvererbung: In Java ist Mehrfachvererbung beispielsweise nur von Interfaces erlaubt, in welchen keine Methoden definiert werden können, wodurch auch keine Konflikte bei der Namensauflösung entstehen können (da es immer nur eine eindeutig passende Methode geben kann).
- Der gewünschte Vererbungspfad muss explizit spezifiziert werden (z. B. in C++) bzw. die Reihenfolge der Spezifikation der Superklassen spiegelt eine implizite Vererbungspriorität wieder (z. B. in CLOS oder Perl).
- Die Vaterklassen können gewichtet werden, beispielsweise durch die Angabe einer Vererbungsreihenfolge („suche zuerst in Vaterklasse C2 und danach in Vaterklasse C3“) (z. B. in Perl).
- Der Vererbungsgraph kann linearisiert werden (z. B. in CLOS oder Dylan), ein Ansatz, der allerdings als umstritten gilt, u. a. weil er unbeabsichtigte Nebenwirkungen haben kann.

Erweiterte Vererbungskonzepte Über die zuvor genannten Vererbungsansätze hinaus existieren verschiedene erweiterte Vererbungskonzepte, die im Kontext der vorliegenden Arbeit von besonderem Interesse sind. Sie werden im Kapitel über verwandte Arbeiten (Abschnitt 7.4) detailliert beschrieben und diskutiert.

Aggregation und Komposition *Aggregation* beschreibt eine `ist-Teil-von` Beziehung zwischen zwei Objekten derart, dass ein Objekt Teil des anderen, umgebenden Objekts ist. Der Unterschied zur Vererbung/Spezialisierung liegt darin, dass hier eine Beziehung zwischen *Objekten* und bei der Spezialisierung zwischen *Klassen* bzw. *Typen* besteht. Demzufolge gilt bei Objekten, zwischen denen eine `ist_Teil_von`-Beziehung besteht, auch nicht das Liskov'sche Substitutionsprinzip, d. h. sie sind nicht austauschbar, da sie disjunkte Attribute bzw. Methoden haben (können).

Beispiel: ein Reifen `ist-Teil-von` einem Auto.

Komposition ist eine strengere Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind, sprich: die Teile können nur in Verbindung mit dem umgebenden Objekt existieren/instanziiert werden.

Delegation Der Begriff *Delegation* wurde erstmals von Liebermann in [Lie86] genannt (s. auch [Kni98]) und bezeichnet dynamische, objektbasierte Vererbung (im Gegensatz zu traditioneller, klassenbasierter). Grundidee hierbei ist, dass ein Objekt Nachrichten, die es nicht „versteht“, an Vaterobjekte delegieren, d. h. weiterleiten kann, wobei die Selbstreferenz an den jeweiligen Vorfahren gebunden wird. Hierbei ist es möglich, bei der Instanziierung eines Objekts explizit ein Vaterobjekt zu spezifizieren. Beispiel:

```
1  class A {
2    void foo() { print("A"); }
3  }
4  class B extends A {
5    void foo() { print("B"); super.foo();}
6  }
7  class C extends A {
8    void foo() { print("C"); super.foo();}
9  }
10 ...
11 A a = new C();
12 a.foo();          // Ausgabe: "CA"
13 A a = new C<new B()>();
14 a.foo();          // Ausgabe: "CBA"
```

In diesem Beispiel wird in Zeile 11 eine „normale“ Instanz der Klasse C erzeugt; in Zeile 13 wird zusätzlich explizit eine Instanz der Klasse B als Vaterobjekt angegeben.

2.3.5. Polymorphie

In einem monomorphen Typsystem (z. B. Pascal, Modula-2) können Funktionen und Prozeduren nur auf einen Typ angewendet werden. Mit Polymorphie bezeichnet man die Fähigkeit, einer Methode verschiedene Bedeutungen (d. h. Definitionen) zuzuordnen, wobei die „passende“ Bedeutung erst zur Laufzeit, also zum Zeitpunkt des Aufrufs der jeweiligen Methode, bestimmt wird. Diese Eigenschaft bezeichnet man auch als *dynamisches* oder *spätes Binden* (dynamic bzw. late binding), im Gegensatz zu *statischem* oder *frühem Binden* (static bzw. early binding), bei dem die Zuordnung bereits zur Übersetzungszeit feststeht. Strachey unterscheidet in [Str67] zwischen parametrischer und Ad-hoc Polymorphie, Cardelli und Wegner verfeinern diese Klassifikation in [CW85] noch weiter:

1. universelle Polymorphie
   a) Parametrische Polymorphie
   b) Inklusionspolymorphie

2. ad-hoc Polymorphie

    a) Überladen
    b) Coercion

Universelle Polymorphie unterscheidet sich von Ad-hoc-Polymorphie in mehreren Aspekten. Bei Ad-hoc-Polymorphie kann ein Name oder ein Wert nur endlich viele verschiedene Typen besitzen. Diese sind zudem zur Übersetzungszeit bekannt. Universelle Polymorphie dagegen erlaubt es, unendlich viele Typen zuzuordnen. Ein weiterer Unterschied liegt darin, dass die Implementierung einer universell polymorphen Funktion generell gleichen Code unabhängig von den Typen ihrer Argumente ausführt, während ad-hoc-polymorphe Funktionen abhängig von den Typen ihrer Argumente unterschiedlich implementiert sein können.

Universelle Polymorphie: *Parametrische Polymorphie* repräsentiert (implizit allquantifizierte) Typen, deren Definitionen Typvariablen enthalten. Dabei können Funktionen in der Regel einen oder mehrere Typparameter haben und die zulässigen Typen besitzen in der Regel eine gemeinsame Struktur. Die polymorphen Funktionen haben implizite oder explizite Typparameter, die den Typ des Funktionsergebnisses für jede Anwendung der Funktion festlegen. Beispiele hierfür sind Templates in C++ oder Generics in Java 5. Des Weiteren existieren Erweiterungen wie „Constrained Parametric Polymorphism“ [Pie91] oder „F-Bounded Polymorphism“ [CCH+89], die Einschränkungen für die möglichen bzw. erlaubten Typen bedeuten.

Inklusionspolymorphie bezeichnet die Eigenschaft, jede Methode statt auf einem Subtyp auch auf einem Basistypen ausführen zu können. Subtyping ist demnach eine Form der Inklusionspolymorphie.

Ad-hoc Polymorphie: Methoden sind *überladen*, wenn unterschiedliche Implementierungen mit demselben Namen verbunden sind. Beispielsweise können in vielen Sprachen mit dem Operator `+` sowohl ganze Zahlen als auch Fließkommazahlen addiert oder sogar Zeichenketten konkateniert werden. Anders ausgedrückt: Eine Methode $m()$ ist überladen, wenn mehrere Methodendefinitionen von m mit unterschiedlichen Deskriptoren existieren.

Coercion ist eine Art implizite Typumwandlung, um z. B. Argumente einer Funktion in die von der Funktion erwarteten Typen umzuwandeln. Coercion ist mit dem Überladen eng verknüpft. Ein Beispiel ist eine Addition von ganzen und Fließkommazahlen: `0.7+1`, bei der viele Sprachen die ganze Zahl 1 automatisch in eine Fließkommazahl umwandeln. Die explizite Typumwandlung wird als *Casting* bezeichnet.

2.3.6. Invarianz, Kovarianz und Kontravarianz

Eng verknüpft mit Typisierung und Vererbung sind die Begriffe von In-, Ko- und Kontravarianz im Kontext von Ergebnis- und Parametertypen bei Methoden.

Kovarianz bedeutet, dass die Typhierarchie der Parameter- bzw. Ergebnistypen mit der Vererbungshierarchie der zu betrachtenden Klassen dieselbe „Richtung“ hat. Wenn man also eine geerbte Methode anpassen will, so ist die Anpassung kovariant, wenn der Parametertyp eines Methodenparameters in der Superklasse ein Supertyp des Parametertyps dieser Methode in der Subklasse ist. Wenn die Typhierarchie der Parametertypen entgegengesetzt der Vererbungshierarchie der zu betrachtenden Klassen läuft, so spricht man von Kontravarianz. Wenn die Typen in der Super- und Subklasse nicht geändert werden, spricht man von Invarianz. Die statische Typsicherheit bleibt bei Kovarianz der Ergebnistypen und bei Kontravarianz der Methodenparametertypen gewährleistet (siehe [Car84]).

Listing 2.2 zeigt ein Beispiel. Die Klasse Y verfeinert dabei die Methoden der Klasse X. Es gilt:

- für `m1`: `Y <: X` und `B <: A` (Kovarianz)
- für `m2`: `Y <: X` und `B :> C` (Kontravarianz)
- für `m3`: Der Parametertyp bleibt unverändert (Invarianz)

```
1  class A {}
2  class B extends A {}
3  class C extends B {}
4
5  class X {
6    B m1 (B);
7    B m2 (B);
8    B m3 (B);
9  }
10
11 class Y extends X {
12   // Kovariante Redefinition, nicht statisch ueberpruefbar
13   C m1 (C);
14   // Kontravariante Redefinition, statisch typsicher
15   C m2 (A);
16   // Invariante Redefinition, statisch typsicher
17   C m3 (B);
18 }
```

Listing 2.2: In-, Ko- und Kontravarianz (bei Parametertypen)

2.4. Datenmodelle und Datenbankschemata

Ein Datenmodell ist ein System von Konzepten zur Strukturierung und Modellierung von Datenbeständen auf theoretischer Ebene, d. h. es dient zur abstrakten Beschreibung von Aspekten der realen Welt. Die am weitesten verbreitete Variante eines Datenmodells ist das Entity-Relationship-Modell [Che76], dessen Kern das ER-Diagramm ist. Ebenfalls gängig ist das konzeptionelle Relationenmodell in Form von Relationen.

Ein *Objektmodell* ist ein Datenmodell mit objektorientierten Konzepten (Objektidentität, Klassenbegriff, Kapselung, Vererbung); insbesondere kann im Objektmodell auch Verhalten modelliert werden, während in traditionellen Datenmodellen ausschließlich Daten modelliert werden. Analog zu ER-Diagrammen im ER-Modell wird bei Objektmodellen die UML-Darstellung verwendet.

Nach Codd [Cod80] definiert sich ein Datenmodell aus drei Eigenschaften:

1. Einer Sammlung von Datenstrukturen.
2. Einer Menge von Operatoren, die auf jede der Datenstrukturen unter 1. angewandt werden kann, um Daten abzufragen oder abzuleiten.
3. Einer Menge von (impliziten oder expliziten) Integritätsregeln, welche sicherstellen, dass fehlerfreie und nicht widersprüchliche Daten vorliegen.

Beeri ([Bee89], [Bee90]) differenziert dabei in einen Strukturteil, einen Operationenteil und höhere Konzepte wie Methoden und Vererbung.

Ein *Datenschema* ist eine Beschreibung einer Modellwelt in einem Datenmodell (d. h. mittels der Konzepte des Datenmodells), also auf praktischer bzw. technischer Ebene. Datenschemata können in drei Kategorien bzw. Ebenen eingeteilt werden: Ein *konzeptuelles* (oder *konzeptionelles*) Datenschema beschreibt das Modell systemunabhängig auf konzeptueller Ebene. Ein *logisches* (oder *formales*) Datenschema beschreibt die Daten in der *Data Definition Language* (DDL) eines bestimmten Datenbanksystems[9]. Ein *physisches* Datenschema beschreibt die tatsächliche Implementierung des Datenmodells.

Vossen schreibt hierzu in [Vos87]:

> „Auf der konzeptionellen Ebene wird die logische Gesamtsicht aller Daten in der Datenbank [...] im konzeptionellen Schema [...] repräsentiert. Dazu bedient man sich eines so genannten Datenmodells, welches insbesondere von der internen Sicht abstrahiert. Zur Gewährleistung logischer Datenunabhängigkeit ist dabei wesentlich, dass dieses Schema von Datenstruktur- oder Zugriffsaspekten frei ist; das konzeptionelle Schema beinhaltet ausschließlich eine Definition des Informationsgehaltes der Datenbank insgesamt.“

[9] Ein Großteil der Datenbank-Sprachen besteht aus zwei Teilen: zum einen der DDL und zum anderen der *Data Manipulation Language* (DML) zur Manipulation der Daten bzw. zum Formulieren von Anfragen.

Außer dem bereits erwähnten ER-Modell sind die folgenden Datenmodelle am bekanntesten bzw. am weitesten verbreitet: hierarchisches Modell, Netzwerkmodell, relationales Modell, objekt(-orientiertes) und objekt-relationales Modell. Diese werden im Folgenden kurz vorgestellt, weitere Einzelheiten zu den Modellen sind u. a. in [Vos87] und [Ull82] zu finden.

2.4.1. Hierarchisches Modell

Das hierarchische Datenmodell wurde in den 1960er Jahren entwickelt und ist eines der ältesten in Datenbanken verwendeten Datenmodelle. Es entstand aus Dateien, die eine variable Anzahl von geschachtelten Wiederholungsgruppen ermöglichen. Das bekannteste System, das mit dem hierarchischen Modell arbeitet, ist IMS (Information Management System) von IBM.

Das hierarchische Datenmodell kennt Records (Sätze) und physisch realisierte Vater-Sohn-Beziehungen (1:n). m:n-Beziehungen können (ohne Redundanz) mit so genannten *virtuellen* Records modelliert werden; ein solcher virtueller Record stellt einen Verweis auf den wirklichen Record dar. Durch Referenzen verknüpfte Records bilden eine hierarchische Baumstruktur. Eine hierarchische Datenbank besteht aus einer Menge von solchen Bäumen.

2.4.2. Netzwerkmodell

Ein verbreiteter Standard des Netzwerkmodells, der Anfang der 1970er Jahre vorgestellt wurde, ist der der *Data Base Task Group* der *Conference on Data System Languages* (CODASYL DBTG). Das Netzwerkmodell kennt Record- und Set-Typen. Die Set-Typen beschreiben 1:n-Beziehungen zwischen Owner-Record-Typ und Member-Record-Typ. Die 1:n Beziehungen müssen hier jedoch keine Hierarchie bilden (wie im hierarchischen Modell). Durch sie darf ein beliebiges Netzwerk aufgebaut werden. Das Netzwerkmodell besitzt eine größere physische Datenunabhängigkeit als das hierarchische Modell, da hier auf der konzeptuellen Ebene nicht festgelegt werden muß, welche 1:n-Beziehungen als „virtuelle Beziehungen“ realisiert werden sollen.

Sowohl das hierarchische Modell als auch das Netzwerkmodell sind implementierungsnah. In beiden Modellen ist ein Navigieren in oft komplexen Datenstrukturen erforderlich. Vossen schreibt in [Vos87]:

> „Vereinfacht ausgedrückt handelt es sich bei diesem Modell um ein ER-Modell, in welchem sämtliche Beziehungen auf zweistellige, many-one-Relationships beschränkt sind“

Bekannte Systeme, welche das Netzwerkmodell einsetzen, sind IDS I & II.

2.4.3. Relationales Modell

Das relationale Datenmodell, welches ebenfalls in den 1970er Jahren enstand, kennt nur Tabellen, auch Relationen genannt. Auf der Schema-Ebene werden die Formate der Tabellen beschrieben. Die Tabellen haben eindeutige Namen und jede Tabelle hat eine feste Anzahl eindeutig benannter Spalten. Für jede dieser Spalten muß ein Wertebereich angegeben werden, die in den Ausprägungen der Tabellen stehenden Zeilen (Tupel) müssen stets der Schemadefinition entsprechen. Weitere Konzepte des relationalen Modells sind:

- Erste Normalform (1NF) - in den Spalten dürfen nur atomare Werte stehen, Wiederholungsgruppen sind nicht erlaubt. Des Weiteren existieren noch weitere Normalformen (2NF, 3NF, 4NF, 5NF, BCNF, ...), die aber im Kontext der vorliegenden Arbeit nicht von zentralem Interesse sind.

- Zeilen können nur durch ihren Inhalt identifiziert werden, d. h. eine Tabelle ist eine Menge von Tupeln (duplikatfrei). Es gibt keine vom Modell vorgeschriebenen Tupel-IDs (vgl. Abschnitt 2.3.1 zu wertebasiert vs. identitätsbasiert).

- Tabellen können durch Mengenoperationen manipuliert werden, z. B. durch Selektion, Projektion, Join, Vereinigung, Differenz, ...

Im Sinne der Typisierung kann man Tabellen als Typ ansehen, d. h. die Attribut- (bzw. Spalten-)Definitionen sind die Typdeklaration und die Zeilen einer Tabelle sind die Instanzen dieses Typs.

Die meisten heutzutage verfügbaren Datenbanken verwenden das relationale Modell. Bekannte Vertreter relationaler Datenbankmanagementsysteme (RDBMS) sind Oracle, Ingres, Informix, DB2 oder auch MySQL.

2.4.4. Objektorientierte Datenmodelle

Mit dem objektorientierten Datenmodell, welches Ende der 1980er Jahre entwickelt wurde, wurden verschiedene Konzepte der objektorientierten Programmierung auf Datenbanken übertragen.

Da es, wie bereits in Abschnitt 2.3 beschrieben, unterschiedliche Definitionen des Begriffes „Objektorientierung“ gibt, wurde im „Object-Oriented Database System Manifesto“ [ABD+89] definiert, welche Konzepte als essentiell und welche als optional für eine OODB gelten. Als essentiell gelten demnach Klassen, Kapselung und Vererbung sowie die Unterstützung komplexer Objekte mit Identität. Zudem sollten OODBs die Qualitätsmerkmale traditioneller Datenbanksysteme übernehmen: Persistenz, Sekundärspeicher-Verwaltung, Mehrbenutzerbetrieb, Transaktionen, Recovery und Möglichkeiten zu Ad-Hoc Anfragen. Verschiedene andere Aspekte wie beispielsweise Mehrfachvererbung, Typ-Inferenz oder Unterstützung von Versionen wurden als optional deklariert.

Die ODMG (Object Data Management Group[10]) hat mit dem ODMG 3.0-Standard eine einheitliche Sicht auf objektorientierte Datenbanken definiert. Dies umfasst u. a. eine allgemeine Objektmodell-Definition, Beschreibungssprachen für Objekte sowie einheitliche Schnittstellen zu diversen Sprachen (in erster Linie Java, C++ und Smalltalk) und Architekturen wie beispielsweise Corba. Bekannte Vertreter sind u. a.: O2, ObjectStore, POET oder GemStome.

Objekt-relationales Modell

Mit dem objekt-relationalen Modell wurde versucht, das relationale Modell um verschiedene OO-Konzepte zu erweitern, insbesondere sind hier mengenwertige Attribute, geschachtelte Relationen und Vererbung zu nennen. Tabelleneinträge in einer relationalen Tabelle können also komplexe Struktur haben, welche einen abstrakten Datentyp (ADT) definieren. Das erste Objekt-relationale DBS war der *Universal Server* von Informix, inzwischen sind viele traditionell relationalen Datenbanksysteme wie beispielsweise Oracle objekt-relational.

2.4.5. XML-Datenmodell

Wie bereits in Abschnitt 2.1.2 beschrieben, besteht das Datenmodell von XML aus Bäumen, d. h. hierarchischen, Strukturen. Genauer beschrieben wird die Semantik durch das *XML Information Set*[11]: ein wohlgeformtes XML-Dokument, das die Namespace-Constraints erfüllt, hat ein Information Set (kurz Infoset). Das Dokument braucht dazu nicht gültig zu sein. Ein XML Information Set besteht aus einer Menge von Information Items (kurz InfoItem); ein Information Item besitzt eine Menge von Eigenschaften (Name-Wert-Paare). Ein Information Set entspricht einem Baum, ein Information Item einem Knoten. Während also eine DTD bzw. Schema das Vokabular eines XML-Dokuments definiert, wird durch das Infoset die Struktur spezifiziert und keine Aussage über den syntaktischen Aufbau getroffen.

Da semistrukturierte Daten geschachtelt sein und/oder Wiederholungen enthalten können, ist eine Einhaltung der ersten Normalform des relationalen Modells z. T. nur über einige Umwege wie beispielsweise die Zuhilfenahme von Hilfsrelationen möglich. Daher spricht man hier auch von der „non-first normal form“ (NF^2).

2.4.6. Klassifikation nach Stonebraker

Stonebraker klassifiziert in [SM96] Datenbankmodelle nach der Komplexität der Datenstrukturen bzw. der Anfragen. Demnach haben das hierarchische, Netzwerk- und

[10] `http://www.odmg.org`
[11] `http://www.w3c.org/TR/xml-infoset`

relationale Modell einfache Datenstrukturen und einfache bis komplexe Anfragen. Objektmodelle und das XML-Modell hingegen weisen eher komplexe Datenstrukturen und auch komplexe Anfragemöglichkeiten auf.

2.5. Programmierparadigmen

Unter einem Programmierparadigma versteht man das einer Programmiersprache oder Programmiertechnik zugrundeliegende Prinzip. Dabei sind als grundlegende Paradigmen das imperative und das deskriptive (auch: deklarative) zu sehen. Alle weiteren (verteilt, aspekt-orientiert, intentional, generativ, constraint-basiert, literate programming, objekt-orientiert, nebenläufig, ...) können als Verfeinerungen dieser Ansätze angesehen werden. Darüber hinaus sind viele Sprachen nicht eindeutig einem Paradigma zuzuordnen, sondern vereinen vielmehr mehrere Ansätze in sich und werden auch als „Hybridsprachen“ bezeichnet.

Imperative Programme beschreiben den Weg, wie ein Problem zu lösen ist, mit Hilfe einer Sequenz von Anweisungen, die den Zustand des Programmes bzw. des Speichers verändern können. Charakteristisch für imperative Programme sind auch bedingte Anweisungen und Schleifen. Beispiele sind die Sprachen Pascal, Java, C++ und Basic.

Im Gegensatz zu imperativen Programmen beschreiben *deskriptive* nicht den Lösungsweg, sondern lediglich das zu lösende Ziel, wobei die konkrete Umsetzung des Lösungswegs dem System überlassen bleibt. Statt Schleifen in imperativen Sprachen wird hier Rekursion verwendet. Beispiele deskriptiver Sprachen sind XSLT oder auch SQL. Auch in HTML[12] beschriebene Webseiten haben deskriptiven Charakter, da das Aussehen der Seiten nur beschrieben wird, die konkrete Darstellung aber dem Browser überlassen wird. Deskriptive Sprachen lassen sich weiter in logische und funktionale untergliedern.

Ein *logisches* Programm definiert eine oder mehrere Relationen mit Hilfe von Formeln bzw. Regeln. Die Auswertung der Formeln erfolgt dabei nach dem Resolutionsprinzip, bei dem als Ergebnis entweder eine neue Formel oder ein Wahrheitswert herauskommt. Prolog ist der bekannteste Vertreter der logischen Sprachen.

Funktionale Programme sind Funktionen, die durch Gleichungen definiert werden, sie stellen somit eine Abbildung von Eingabedaten auf Ausgabedaten dar. Sie werden durch Reduktion von Ausdrücken ausgeführt. Insbesondere existieren keinerlei An- oder Zuweisungen. Da im Rahmen der vorliegenden Arbeit eine funktionale Sprache konzipiert wurde (was in Abschnitt 4.4.1 begründet wird), wird an dieser Stelle näher auf die Besonderheiten eingegangen.

Rein funktionale Programmiersprachen fassen Programme als mathematische Funktionen auf: Ein Ausdruck hat dort während der Programmausführung immer den gleichen Wert[13]. Es gibt keine Zustandsvariablen, die während einer Berechnung geändert werden.

[12] Dabei ist HTML genaugenommen keine Programmiersprache, da z. B. Kontrollstrukturen fehlen.

[13] Aus diesem Grund bezeichnet man die funktionale Programmierung auch als werte-orientierte Pro-

Man bezeichnet diese Eigenschaft als referenzielle Transparenz (*referential transparency*). Dies hat zur Folge, dass ein Name jeweils durch den von ihm definierten Wert ersetzt werden darf, was ggf. Möglichkeiten für Optimierungen durch unfolding eröffnet. Darüber hinaus hängt der Wert eines Ausdrucks nur von den Werten seiner Teilausdrücke ab, es können somit keine Seiteneffekte[14] durch Variablen-Updates entstehen („Seiteneffektfreiheit"). Anders ausgedrückt: Die Ausgabe eines Programms hängt ausschließlich von den Werten der Argumente ab, nicht vom Aufrufkontext. Eine Folge der Seiteneffektfreiheit ist, dass die Reihenfolge der Auswertung der Teilausdrücke irrelevant ist, insbesondere ist eine parallele Auswertung möglich.

Ein essentielles Merkmal funktionaler Sprachen ist, dass Funktionen *first class citizens* sind, d. h. sie sind gleichberechtigt zu anderen Daten (Zahlen, boolesche Werte, Listen). Insbesondere können sie als Argumente und/oder Resultate anderer Funktionen auftreten (auch als *higher-order Funktionen* oder *Funktionale* bezeichnet).

Ein typisches Beispiel ist die `map`-Funktion, die eine beliebige, übergebene Funktion f auf jedes Element einer Liste anwendet:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

In der ersten Zeile wird das Ergebnis für eine leere Liste `[]` zurückgegeben; die zweite Zeile wendet die Funktion `f` auf das erste Listenelement `x` an und führt dann einen Rekursionsschritt für die restliche Liste `xs` durch.

Bei strikter Auswertung (eager bzw. strict evaluation) werden die Argumente von Funktionen zuerst ausgewertet. Dagegen werden bei der nicht-strikten Auswertung (auch: Bedarfsauswertung, lazy evaluation) zunächst die Ausdrücke als ganzes übergeben und erst ausgewertet, wenn deren Wert in einer Berechnung benötigt wird. Dadurch lassen sich z. B. potenziell unendlich große Datenstrukturen (die Liste aller natürlichen Zahlen, die Liste aller Primzahlen, etc.) definieren.

Die mathematische Grundlage der funktionalen Programmierung ist der λ-Kalkül, das auf Church (u. a. [Chu36]) zurückgeht. Ein funktionales Programm ist ein Ausdruck des λ-Kalküls, die Ausführung eines solchen Programms entspricht der Auswertung des Ausdrucks durch Reduktion. Im λ-Kalkül unterscheidet man drei grundlegende Bausteine:

1. Variablen: x.
2. Eine Funktionsabstraktion $\lambda x.A$ definiert eine Funktion, die als Argument die Variable x hat, und einen Ausdruck A als Funktionskörper hat, in dem in der Regel x vorkommt, aber nicht vorkommen muß. Die Funktionen haben dabei keinen Namen, sind also anonym.

grammierung.

[14] Als *Seiteneffekt* bezeichnet man eine bleibende Veränderung (und somit nachträglich beobachtbare Auswirkung) des Zustands eines Programms – auch eine Ausgabe ist ein Seitenffekt.

3. Eine Funktionsapplikation FA bedeutet, daß die Funktion F auf den Ausdruck A angewandt wird.

Beispiele:

- Die Identität: $\lambda x.x$.
- Eine Funktion, die jedes Argument auf die Identitätsfunktion abbildet: $\lambda y.(\lambda x.x)$.
- Die Identität, angewandt auf sich selbst: $(\lambda x.x)(\lambda y.y)$.

Anzumerken ist, dass mittels Abstraktion und Applikation auch Zahlen, Datentypen, logische Funktionen, etc. konstruierbar sind. Zu Einzelheiten sei an dieser Stelle auf die Literatur (z. B. [Bar84]) verwiesen. Ursprünglich war der λ-Kalkül ungetypt, später folgten aber auch getypte Varianten, welche die Grundlage für moderne funktionale Sprachen wie beispielsweise Haskell bilden.

3. Analyse von Content Management Systemen und deren Datenmodellen

Nachdem im vorangegangenen Kapitel die für das weitere Verständnis der Arbeit benötigten Grundlagen vorgestellt wurden, beschäftigt sich dieses Kapitel nun mit dem eigentlichen Gegenstand der Arbeit: Content Management Systeme und Datenmodelle für Websites. Zunächst wird dabei beschrieben, wodurch sich CMS auszeichnen und welche Funktionalitäten sie haben. Daran schließt eine Analyse der Aspekte der Datenmodellierung von CMS und die daraus resultierenden Anforderungen an ein Datenmodell, das diese Strukturen geeignet abbilden bzw. repräsentieren kann, an. Danach werden existierende Systeme und deren Limitierungen in Hinblick auf diese Anforderungen diskutiert und damit motiviert, warum die Konzeption eines neuen Datenmodells für CMS nötig ist.

3.1. Content Management Systeme: Grundlagen

Es existiert eine Vielzahl mehr oder weniger genauer Definitionen von Content Management Systemen, die im Folgenden zusammengefasst werden. Ein Content Management System ist ein System zur Administration von Websites, wobei der Erstellungsprozess basierend auf der Trennung von Inhalten und Struktur unterstützt wird; es verwaltet die Inhalte (*Contents* oder auch →Assets) einer →Website und liefert diese je nach Anfrage in unterschiedlichen Formaten und Formen aus. Ein *Web* Content Management System (WCMS) zeichnet sich zusätzlich noch dadurch aus, dass es komplett (oder zumindest in großen Teilen) über einen Webbrowser gepflegt werden kann und somit auf den Clients keine zusätzliche Software installiert werden muss. Ein Erstellen bzw. Pflegen komplexer Webauftritte ist ohne den Einsatz eines CMS kaum bzw. gar nicht möglich.

Ein Spezialfall eines CMS ist ein *Dokumentenmanagementsystem* (DMS), das zur Verwaltung elektronischer Dokumente eingesetzt wird. Im Unterschied zu (W)CMS sind diese Dokumente dabei in der Regel unveränderliche, in sich geschlossene Dokumente eines proprietären Formats, während sich →Webseiten aus kleineren Bausteinen zusammensetzen (können) und auch in verschiedenen Formaten ausgegeben werden können und sollen. Der Fokus der vorliegenden Arbeit liegt dabei auf WCMS und damit modellierten „klassischen“[1] Websites bzw. →Portalen.

[1] Hiermit seien solche Webangebote gemeint, die Informationen für menschliche Leser bereitstellen,

Die Grundfunktionen eines CMS umfassen:

Authoring und Pflege: Neue redaktionelle Inhalte lassen sich einfach ohne Programmier- bzw. HTML-Kenntnisse erstellen, bestehende Informationen können editiert werden.

Benutzer- und Rechteverwaltung: Die Inhalte können von verschiedenen Benutzern, die zudem verschiedene Rollen (Redakteur, Administrator, ...) einnehmen können, bearbeitet werden. Dabei lassen sich in CMS Rechte zumeist feingranular, d. h. beispielsweise für einzelne Assets oder Textteile, vergeben.

Qualitätssicherung und Freigabe: Mit →Workflows können Freigabeprozesse entwickelt werden, die eine ständige Kontrolle der neu erstellten Inhalte von Websites ermöglichen.

Verwaltung: Release- und Verfallsdatenüberwachung inkl. Versionierung und evtl. Archivierung ermöglichen es unter anderem, auf alte Versionen der Daten zuzugreifen oder bestimmte Inhalte erst ab (bzw. bis zu) einem bestimmten Termin öffentlich sichtbar zu machen.

3.1.1. Grundlegende CMS-Konzepte

Separation of Concerns: Ein essentieller Aspekt eines CMS ist die Trennung von Inhalten, Verarbeitungslogik und Layout, die, dank eines einheitlichen Datenmodells, dezentral und bei WCMS web-orientiert gepflegt werden können. Diese Trennung wird als „Trennung der Belange“ bzw. „Separation of Concerns“ [Maz01] bezeichnet, ein Paradigma, dass auch in anderen Bereichen wie z. B. im Software Engineering[2] oder beim Aspect-Oriented Programming (s. [HL95] oder [Ern03b]) einen großen Stellenwert einnimmt.

Der Vorteil dieses Konzepts liegt zum einen in einer besseren Abgrenzung der Zuständigkeiten, d. h. ein Redakteur kann sich auf die redaktionelle Arbeit konzentrieren, ohne sich mit Webdesign bzw. Programmierung auskennen zu müssen, während ein Designer nicht programmieren (können) oder Inhalte pflegen muss. Zum anderen kann Redundanz vermieden werden: beispielsweise müssen Änderungen am Design nur an einer zentralen Stelle durchgeführt werden und wirken sich auf den gesamten Webauftritt aus, wie z. B. ein Logo oder eine Fußzeile, die auf allen Seiten auftreten.

im Gegensatz zu reinen „Datenbank“-Seiten, die beispielsweise unformatierte, im Sinne von CMS unstrukturierte, Daten darstellen. Ein Großteil existierender Websites entspricht diesem Kriterium.

[2] Wie z. B. das MVC-Entwurfsmuster (s. [GHJV95]).

Templates: Ein zentrales Konzept von CMS, das insbesondere das eben vorgestellte Separation of Concerns-Paradigma unterstützt, sind Templates. Hierbei werden textuelle Schablonen mit Platzhaltern erstellt, welche zur Laufzeit mit Inhalt gefüllt werden. In derartigen Templates kann zumeist auch Programmcode einer *Templatesprache* eingebettet werden. Prinzipiell gibt es dabei zwei Varianten: zum einen Text- bzw. Datenschablonen, in denen Programmcode eingebettet ist und zum anderen Programme, in denen die Platzhalter mit Daten gefüllt werden. Der „Wechsel“ vom Text- zum Programmteil wird durch so genannte Quote- und Antiquote-Zeichen vorgenommen[3].

Templatesprachen eignen sich insbesondere zur schnellen und einfachen Generierung von Text, da auf einfache Weise textuelle Ausgaben mit dynamischen Informationen und Programmlogik verknüpft werden können. Daher sind sie für CMS im Allgemeinen und für die vorliegende Arbeit im Speziellen von besonderem Interesse.

Single source, multiple target: Das Prinzip, aus ein und denselben Daten mehrere unterschiedliche Ausgabeformate zu erzeugen, („Single source, multiple target“) ist ebenfalls essentiell für CMS. Auch hierbei wird versucht, Redundanzen zu vermeiden, indem die eigentlichen Daten nur einmal eingegeben werden müssen, aber in verschiedenen Formaten ausgegeben werden können – außer (X)HTML sind dies häufig PDF, RTF oder auch LaTeX, aber auch Grafiken (z. B. SVG). Im Kontext von XML lässt sich dies durch das Anwenden verschiedener XSLT-Stylesheets auf ein XML-Dokument umsetzen, wie in Abbildung 3.1 schematisch gezeigt. Wichtig

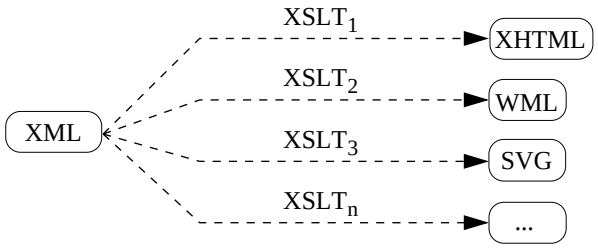


Abbildung 3.1.: Single source, multiple target

in diesem Kontext ist es aber auch, die Daten für unterschiedliche Endgeräte (PC, PDA, Handy) in unterschiedlich große Teilbereiche aufteilen und somit die Granularität der Ansicht verändern zu können – während ein PC-Benutzer einen langen Text mit Grafiken sieht, soll (bzw. will) ein PDA-Benutzer eine möglichst kompakte Darstellung der Informationen erhalten.

[3]Beispiele hierfür sind {{ und }} in Wob 1, `<%` und `%>` in JSP sowie `<?` und `?>` in PHP.

Mögliche Anwendungsbeispiele sind: Im WWW wird eine Webseite im Browser der besseren Lesbarkeit wegen in mehrere Seiten aufgeteilt, kann aber zum Ausdruck als Ganzes im PDF-Format ausgeliefert werden.

Internationalisierung und Personalisierung Gerade bei großen Webangeboten mit internationalen Adressaten ist es in der Regel wünschenswert, die Seiten in mehreren Sprachen ansehen zu können, wobei das Aussehen (d. h. Layout, Grafiken, usw.) möglichst unverändert bleiben soll. Zudem soll es Benutzern im Falle von Personalisierung möglich sein, das Erscheinungsbild bzw. die sichtbaren Inhalte des Webangebots nach ihren Wünschen zu variieren.

Barrierefreiheit Websites, die von allen unabhängig von ihren körperlichen und/oder technischen Möglichkeiten uneingeschränkt genutzt werden können, werden als barrierefrei bezeichnet. Das Ziel ist hier, dass Webseiten sowohl von (seh)behinderten Menschen als auch technisch eingeschränkten Systemen (z. B. Textbrowser) benutzt werden können. Nach dem Behindertengleichstellungsgesetz[4] sind öffentliche Einrichtungen verpflichtet, ihre Webseiten barrierefrei zu gestalten.

Auf der Implementierungs- bzw. Designseite umfasst dies verschiedene Aspekte wie beispielsweise Vermeidung von rein grafischen Elementen oder Tabellen zur Formatierung. Dieses Vorgehen wird durch die Trennung von Inhalten und Design nach dem Separation of Concerns-Paradigma erleichtert, z. B. durch den Einsatz von CSS (Cascading Style Sheets, [LB96]).

3.1.2. Charakterisierung von Änderungen

Änderungen im Kontext von CMS lassen sich grob in drei Kategorien klassifizieren, die in der Regel von unterschiedlichen Benutzern (bzw. einem Benutzer in unterschiedlichen Rollen) durchgeführt werden:

1. Strukturelle bzw. inhaltliche Änderungen: hierbei werden neue Inhalte eingepflegt bzw. bestehende geändert. Beispiele sind das Hinzufügen eines neuen Lehrveranstaltungseintrags oder das Ändern einer Überschrift. Diese werden von „normalen“ Benutzern/Redakteuren durchgeführt.

2. Designänderungen: hier werden Aspekte der Daten*präsentation* verändert – beispielsweise das Hervorheben von Textpassagen oder das Layout der Webseite. Diese werden in der Regel von Webdesignern durchgeführt.

3. Funktionale Änderungen: sind Änderungen, die die Funktionalität der Website betreffen, d. h. wie „reagiert“ das CMS auf Anfragen, wie werden Webseiten aus

[4] `http://www.bmgs.bund.de/download/gesetze/behinderung/bitv_ver.htm`

den Daten generiert, etc. Derartige Änderungen werden von Entwicklern durchgeführt, d. h. von denjenigen, die neue Websites/Anwendungen mit einem CMS implementieren.

Ein CMS-Datenmodell sollte dabei alle Arten unterstützen, insbesondere aber sollte es Entwicklern ermöglichen, funktionale Änderungen einfach, schnell und redundanzfrei spezifizieren zu können.

3.2. Aspekte der CMS-Datenmodellierung

Nachdem nun die grundlegenden Konzepte von CMS beschrieben wurden, werden im Folgenden die Daten und deren Strukturen, die mit CMS verwaltet werden, analysiert, um daraus dann die Anforderungen an ein geeignetes Datenmodell, das diese Strukturen abbilden kann, abzuleiten.

3.2.1. Hierarchien

Die im Vorfeld der Arbeit durchgeführte Analyse bestehender Websites bzw. Konzeption neuer Webauftritte zeigte, dass fast alle Websites eine ausgeprägte hierarchische Struktur besitzen, d. h. Seiten haben Vater-, Kinder- sowie Geschwisterseiten.

Als Beispiel seien hier die Lehre-Seiten des Instituts für Informatik der Universität Bonn angeführt (Abbildung 3.2, Abbildung 3.3 verdeutlicht die dahinterliegende Content-Struktur – aus Platzgründen weggelassene Knoten werden hierbei durch gepunktete Linien angedeutet). Die einzelnen Lehrveranstaltungen hängen dabei unter einem „Semester“-Knoten, der wiederum unter einem „Lehrveranstaltungs“-Knoten hängt, usw.

Diese Baumstruktur induziert eine natürliche Navigationsstruktur, bei der entlang der verschiedenen Achsen (Vorfahren, Geschwister, Nachkommen) navigiert werden kann, im Beispiel aus Abbildung 3.2 u. a. an den mit Pfeilen versehenen Links sichtbar. Ohne eine derartige Navigationsstruktur wären komplexe Websites kaum komfortabel bedienbar.

Ein Datenmodell, welches eine hierarchische Primärstruktur besitzt, würde also die Entwicklung bzw. Modellierung von Websites deutlich vereinfachen, da ein Entwickler auf diese inhärenten (Baum-) Strukturen zugreifen kann[5] und unter anderem eine Navigation innerhalb dieser Strukturen teilweise oder vollständig automatisch generiert werden kann. Die Erfahrungen mit dem im Wob 1-System verwendeten hierarchischen Modell bestätigen diese Behauptung.

Von der Primärstruktur abweichende Sichten auf die Daten wie Selektion (beispielsweise nur Lehrveranstaltungen einer bestimmten Abteilung) oder unterschiedliche Sortierungen, sollten ebenfalls über spezielle Anfragen bzw. das API von Modell unterstützt werden.

[5] Zum Beispiel über eine geeignete Programmierschnittstelle (API, Application Programming Interface).

Abbildung 3.2.: Beispiel Informatik-Lehrangebot

Ein weiterer Vorteil der hierarchischen Modellierung ist, dass hier auch tiefere Strukturen bzw. Substrukturen modelliert werden können. Beispielsweise können unter einen Lehrveranstaltungsknoten weitere Textknoten gehängt werden (so dass aus einem Blattknoten ein innerer Knoten wird). Abbildung 3.4 zeigt ein anderes Beispiel: hier sind innerhalb eines Veranstaltungskalenders verschiedene Vortragsknoten, aber auch ein Tagungsknoten enthalten, der wiederum einen weiteren Veranstaltungskalender mit Vorträgen umfasst. In einem flachen Modell ist dies meist nicht vorgesehen, weil es dort kompliziert zu modellieren bzw. zu bedienen ist (in einem RDBMS beispielsweise müssen dazu mehrere verknüpfte Tabellen verwendet werden), ein hierarchisches Modell erlaubt aber eine homogene und somit adäquatere Modellierung.

3.2.2. Klassen und Vererbung

Eine weitere Beobachtung ist, dass sich die Knoten in der Website-Hierarchie, also die Bausteine, aus denen Websites bestehen, in der Regel in Gruppen gleichartiger Entitäten

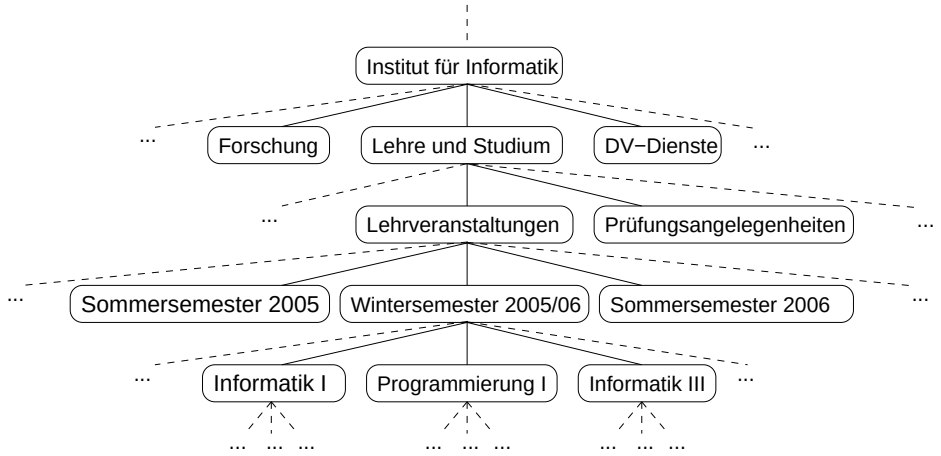


Abbildung 3.3.: Beispiel Informatik-Lehrangebot: Baumstruktur (Ausschnitt)

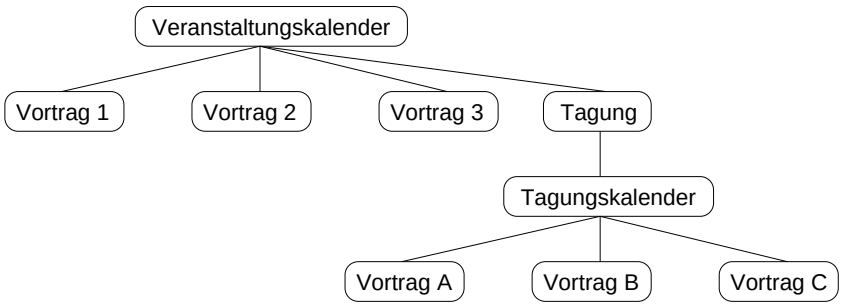


Abbildung 3.4.: Substrukturen

einteilen lassen, welche die gleichen Eigenschaften haben. Daher ist es also nahe liegend und vorteilhaft, die Strukturen über Attribute von Klassen bzw. deren Instanzen zu beschreiben.

Die Entitäten des obigen Lehre-Beispiels (Abbildungen 3.2 und 3.3) können beispielsweise mit den Klassen in Abbildung 3.5 modelliert werden. Dabei sind die Knoten „Institut für Informatik“, „Lehre und Studium“ und „Lehrveranstaltungen“ Instanzen der Klasse `Freier Text`, die Semesterknoten Instanzen der Klasse `Semester` und die Veranstaltungen Instanzen der Klasse `Veranstaltung`. Jede Klasse hat dabei entsprechende Attribute, Instanzen der jeweiligen Klassen haben (potenziell) unterschiedliche Attributwerte.

Darüber hinaus lässt sich beobachten, dass einige Klassen Spezialisierungen bzw. Verallgemeinerungen anderen Klassen sind; so ist im Lehre-Beispiel die Klasse `Freier Text`

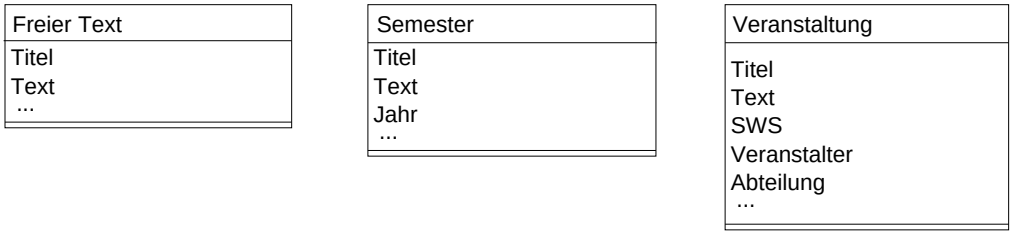


Abbildung 3.5.: Lehre-Beispiel: mögliche Klassenmodellierung (in UML)

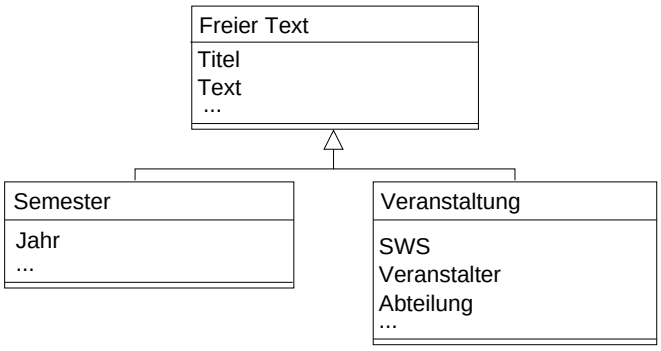


Abbildung 3.6.: Lehre-Beispiel: Klassen mit Vererbung

eine allgemeine Klasse mit den Attributen `Titel` und `Text`, welche die (spezielleren) Klassen `Semester` und `Veranstaltung` ebenfalls besitzen. Somit bietet sich hier zwecks Redundanzvermeidung und inkrementeller Spezifikation eine Vererbungshierarchie an, bei der `Semester` und `Veranstaltung` Subklassen der Klasse `Freier Text` sind, wie in Abbildung 3.6 illustriert.

3.2.3. Substrukturen und Aggregation

Die Untersuchung der Struktur einzelner Webseiten zeigte, dass diese in vielen Fällen ganz analog zur gesamten Website strukturiert sind, d. h. auch einzelne Seiten bestehen aus Substrukturen, im objektorientierten Sinne kann dies auch als Aggregation aufgefasst werden.

Als Beispiel sei diesmal eine einfache Homepage gegeben, wie in Abbildung 3.7a zu sehen. Diese Seite wird aus verschiedenen Content-Einheiten bzw. Substrukturen zusammengesetzt, die unterschiedliche Informationen darstellen. Abbildung 3.7b zeigt eine mögliche Aufteilung in derartige Substrukturen: Eine umfassende Entität definiert die Überschrift und ggf. weitere für diese Seite globalen Attribute; darin enthalten – aggre-

giert – sind ein Textblock („Willkommen“), ein Terminblock, der wiederum 3 Termine enthält (aggregiert), und ein Textblock am Ende („Impressum“). Abbildung 3.8 zeigt eine mögliche Modellierung als Baum von Instanzen verschiedener Klassen (`Text`, `Calendar`, `Appointment`), welche jeweils Attribute (`title`, `body`, `date`) besitzen.

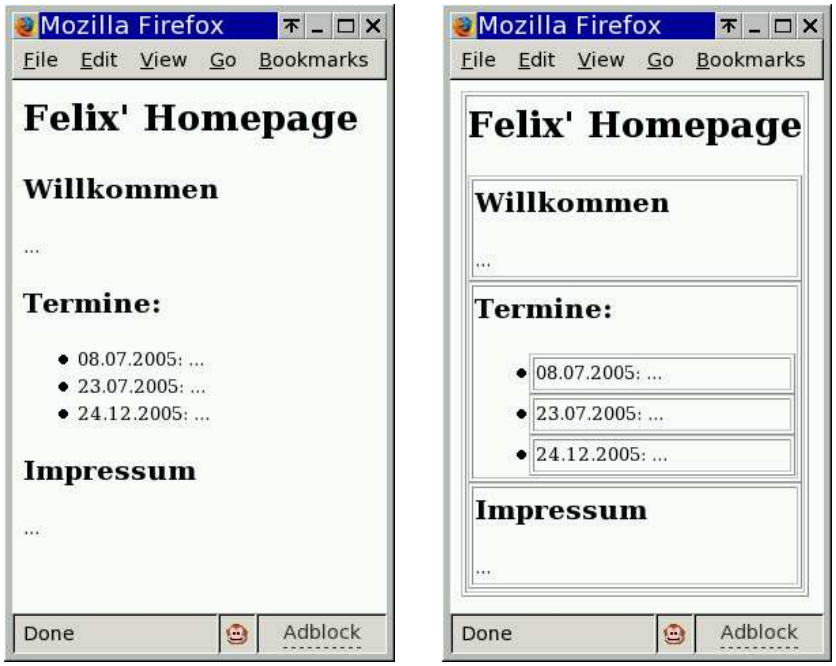


Abbildung 3.7.: Beispiel-Webseite:
a) normale und b) Substruktur-Ansicht

In Hinblick auf die gesamte Website lässt sich eine einzelne Seite also als (Teil-) Baum von Entitäten auffassen. Eine homogene Modellierung sowohl der gesamten Website als auch einzelner Webseiten ermöglicht zum einen einen hohen Grad an Flexibilität, falls man nachträglich die „Seitenschranken“ verschieben, d. h. die Granularität der Sicht auf die Daten verändern will, wenn beispielsweise zu viele Informationen auf einer Webseite sichtbar sind und diese auf mehrere Seiten aufgeteilt werden sollen. Weiterhin können so leichter andere Ansichten für unterschiedliche Endgeräte (z. B. PDAs) erzeugt werden, da dann z. B. immer nur kleinere Teile der Seite angezeigt werden können. Im Beispiel: statt der ganzen Seite könnte nur *ein* `Text`- bzw. `Calendar`-Block mit Navigationslinks zu den anderen Substrukturen angezeigt werden.

Abbildung 3.9 illustriert eine Verschiebung der Seitenschranken der Webseiten: Die Bäume repräsentieren dabei die gesamte Website, die gestrichelt umrandeten Knoten die jeweils angezeigte Webseite. Eine konkrete Webseite ist demnach als Fenster bzw.

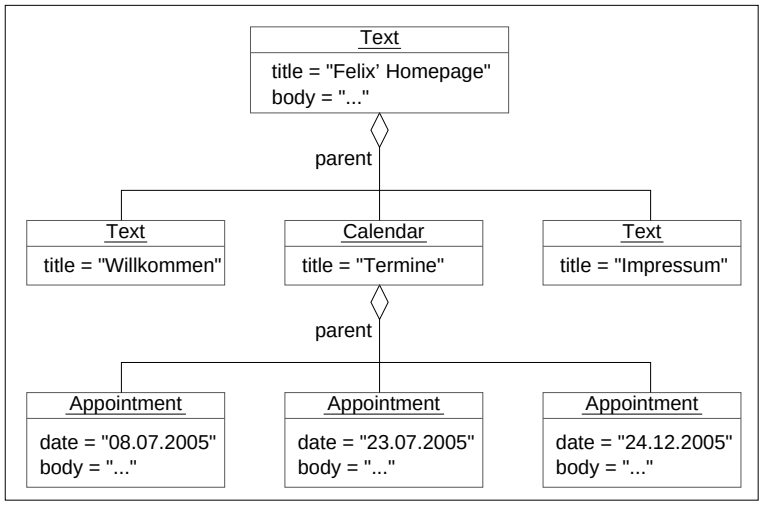


Abbildung 3.8.: Mögliche Modellierung der Beispiel-Webseite

Sicht auf einen Teilbaum zu verstehen.

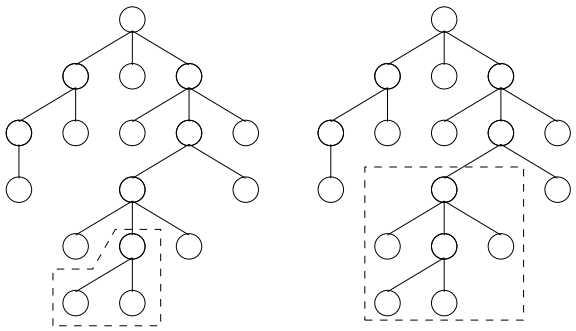


Abbildung 3.9.: Verschiebung der Seitenschranken von Webseiten

3.2.4. Inkrementelle Modellierung

Wie im vorigen Abschnitt erläutert, erlaubt ein Klassen-/Vererbungskonzept die inkrementelle und damit redundanzfreie Modellierung von Daten, da man in einer abgeleiteten Klasse alle Methoden der Superklasse(n) zur Verfügung hat und diese dann nur noch um gewünschte neue Methoden erweitern kann. Wie im Folgenden gezeigt werden wird, ist diese „normale“ Vererbung nicht unbedingt ausreichend und eine darüber hinausgehende Inkrementalität kann überaus hilfreich sein, wie das nachstehende Beispiel illustriert.

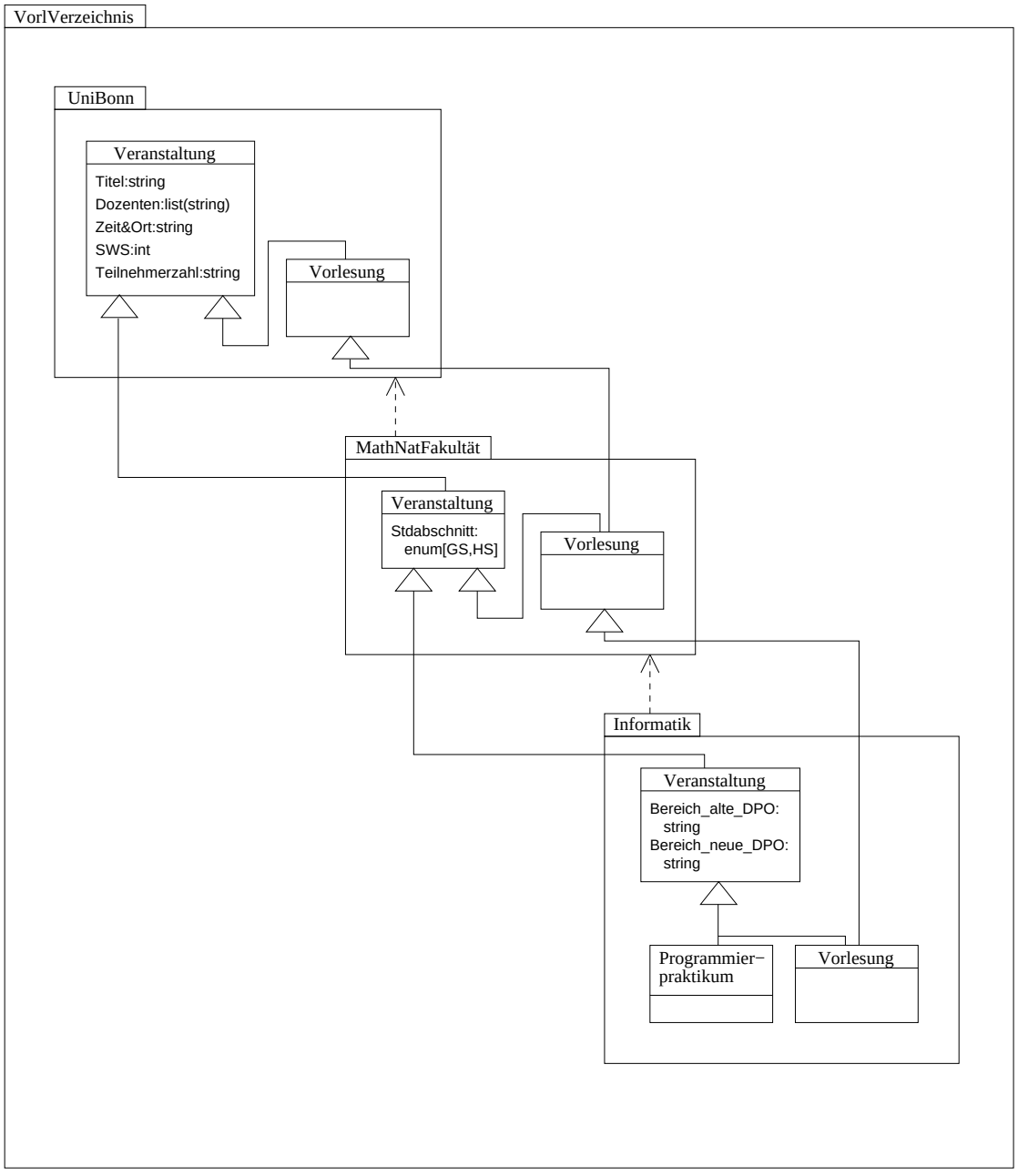


Abbildung 3.10.: Beispiel Vorlesungsverzeichnis

Abbildung 3.10 zeigt einen Ausschnitt einer möglichen Modellierung eines Vorlesungsverzeichnisses (der Universität Bonn). Auf der obersten Ebene der Universität werden allgemeine Veranstaltungsklassen modelliert, die auf Fakultäts- bzw. Institutsebene weiter verfeinert werden können und dabei neue Attribute (und ggf. Methoden) erhalten können. Dies ist insbesondere bei der Modellierung inhomogener Daten hilfreich bzw. nötig. Die Struktur der Veranstaltungen der Universität Bonn ist in diesem Sinne recht heterogen, da jede Fakultät bzw. jedes Institut unterschiedliche Veranstaltungstypen bzw. -attribute haben kann. Eine einfache Vererbung, wie sie bislang beschrieben wurde, reicht hierbei zur Modellierung nicht aus, da nicht einzelne Klassen, sondern komplette Klassenstrukturen bzw. -hierarchien vererbt werden müssen. Nachfolgend wird diese Problematik anhand eines (übersichtlicheren) Homepage-Beispiels näher beleuchtet.

Bereits bei der Konzeption von Webauftritten mit dem Wob 1-System zeigte sich die Notwendigkeit, aus einer bestehenden Anwendung schnell eine neue ableiten zu können. Mit Anwendung ist hier eine Klassenstruktur bzw. -hierarchie (mit enthaltenem Zustand und Verhalten) gemeint, wie sie z. B. in Abbildung 3.11 vereinfacht zu sehen ist. Eine solche Klassenhierarchie wird im Folgenden auch *Familie* genannt. Ziel war bzw. ist es, diese Klassenstruktur schnell und einfach „als Ganzes“ abzuleiten, so dass zum einen nachträgliche Änderungen an der Vateranwendung (wie z. B. Bugfixes oder Zusatzfunktionalität) an die Kindanwendung vererbt werden, aber zum anderen auch lokale Änderungen nur an der Kindanwendung möglich sind. Auf diese Art kann aus einer generischen Anwendung eine (bzw. mehrere) spezifische abgeleitet werden, so dass Methoden und Attributdeklarationen inkrementell spezifiziert werden können.

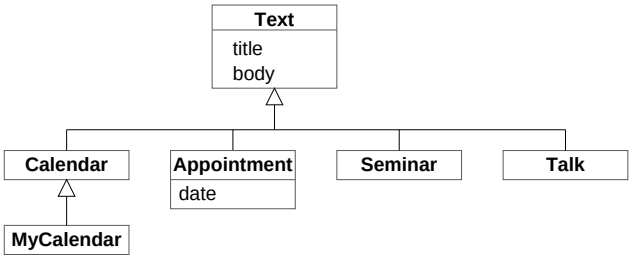


Abbildung 3.11.: Klassenstruktur homepages

Abbildung 3.12 zeigt eine derartig (von der `homepage`-Anwendung aus Abbildung 3.11) abgeleitete Anwendung, bei der die Klasse `Text` ein neues Attribut namens `subtitle` sowie eine neue Subklasse namens `Linklist` erhalten hat.

Der Versuch, diese Strukturen mit dem in Wob 1 verwendeten „normalen“ Vererbungsmodell umzusetzen, zeigte, dass dies nicht bzw. nur sehr umständlich realisierbar ist, wie im Folgenden gezeigt.

Ein erster, naiver, Ansatz wäre, jede Klasse einzeln abzuleiten, wie in Abbildung 3.13

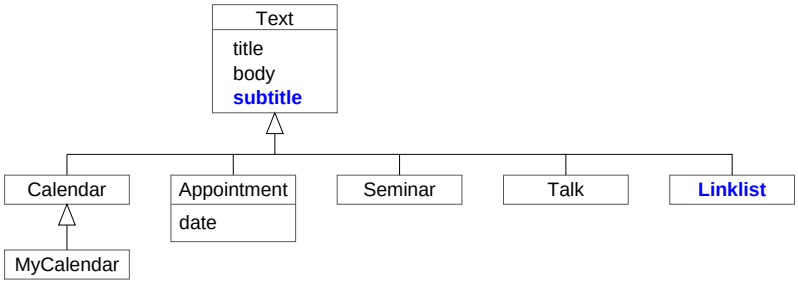


Abbildung 3.12.: Klassenstruktur mysite

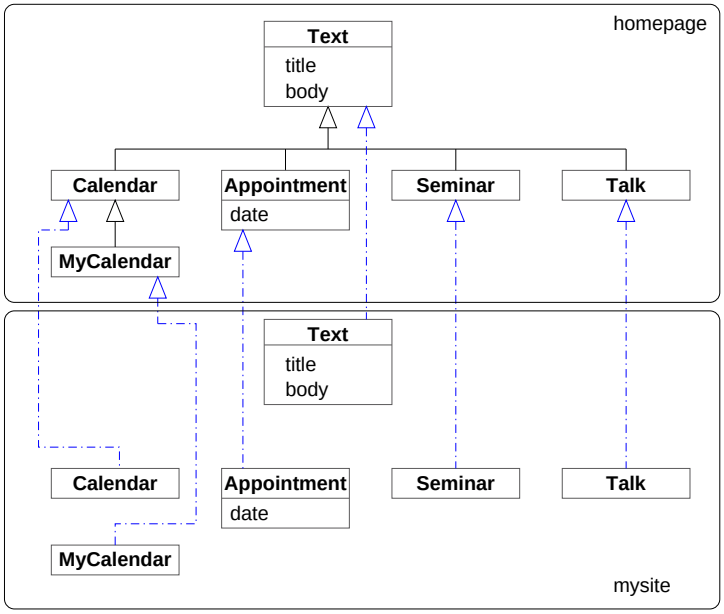


Abbildung 3.13.: Naiver Vererbungsansatz

zu sehen. Hier ginge dann allerdings die Struktur- bzw. Hierarchieinformation in der abgeleiteten Anwendung verloren, was u. a. zur Folge hätte, dass Änderungen in der abgeleiteten Klasse (`mysite.Text`) nicht mehr an die entsprechenden Subklassen propagiert werden würden (`mysite.Calendar`, `mysite.Appointment`, ...). Die Super/Subklassenbeziehungen in den jeweiligen Klassen müssten also manuell wieder eingefügt werden.

Eine weitere Möglichkeit wäre, die Klassenhierarchie 1:1 redundant zu kopieren. In diesem Falle ginge jedoch die Verbindung zur Superklasse verloren, so dass nachträgliche Änderungen in der Vateranwendung (Bugfixing, zusätzliche Funktionalität) nicht mehr in der Kindanwendung verfügbar wären.

Um derartige Anforderungen (aus objektorientierter Sicht) sauberer modellieren zu können, soll ein geeignetes Datenmodell also eine entsprechende Art der Vererbung unterstützen. Dann lassen sich Klassenhierarchien einfach inkrementell und redundanzfrei spezifizieren und zudem wird es möglich, Änderungen aus einer Vateranwendung an die Kindanwendung(en) zu vererben, aber auch lokale Änderungen an Kindanwendungen durchzuführen.

In Wob 1 wurde, da eine solche inkrementelle Spezifikation nicht vorgesehen war, eine improvisierte Lösung als „workaround“ entwickelt. Dabei wurden die verschiedenen Klassen (`Text`, `Calendar`, ...) als Attribut *subtype* von nur einer Master-Klasse modelliert, wie Abbildung 3.14 illustriert. In dieser Master-Klasse werden nun alle Attribute und Methoden zusammengefasst, die zu den jeweiligen Subklassen zugehörig sind. So ist für jeden subtype eine eigene `html_body`-Methode, die diesen subtype darstellt, nötig; die globale `html_body`-Methode entscheidet dann auf Grund des subtypes, welche Methode ausgewertet wird. Die gewünschte Hierarchie kann durch Instanziierung der Klasse mit unterschiedlichen subtypes erreicht werden (s. Abbildung 3.15).

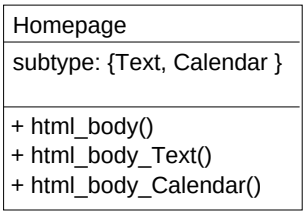


Abbildung 3.14.: Wob 1: Homepage-Klasse mit subtype

Eine Vererbung, welche die gewünschten inkrementellen Anforderungen erfüllt, ist in diesem Ansatz durch Ableiten der Masterklasse möglich; Änderungen können dann entweder in der abgeleiteten Klasse oder in der Masterklasse durchgeführt werden, wie in Abbildung 3.16 zu sehen.

Diese Lösung führt jedoch zu einer aufgeblähten Master-Klasse, die Zustand und Ver-

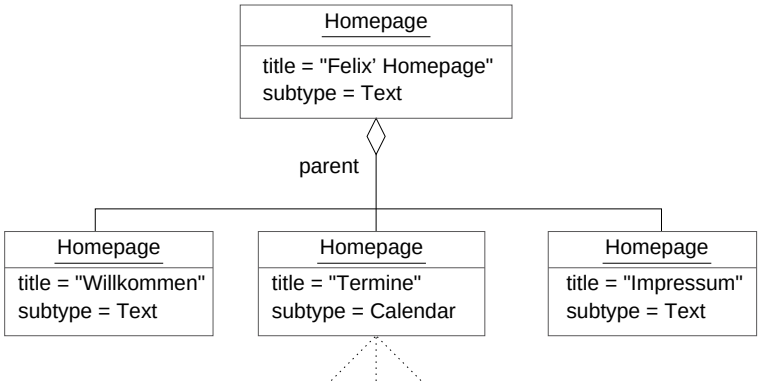


Abbildung 3.15.: Wob 1: Instanzenbaum mit subtype

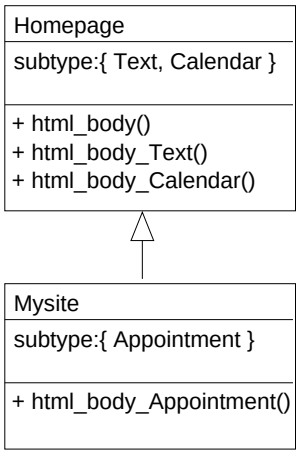


Abbildung 3.16.: Wob 1: Von `Homepage` abgeleitete Klasse `Mysite`

halten verschiedener, prinzipiell voneinander unabhängiger Entitäten vermischt und somit das OO-Grundprinzip der Kapselung verletzt. Insofern ist dieser Ansatz im Hinblick auf eine saubere objektorientierte Modellierung unbefriedigend, was u. a. darin begründet liegt, dass die gewünschte Funktionalität im Wob 1-Modell nur nachträglich aufgesetzt ist.

3.3. Existierende Content Management Systeme

In diesem Abschnitt werden existierende CMS im Hinblick auf die bisher aufgestellten Kriterien bezüglich der Daten untersucht und diskutiert.

Laut CMSmatrix[6] existieren zurzeit über 500 CMS auf dem Markt. Ein Großteil der CMS (ca. 50%) wurden in PHP implementiert, etwa 25% in Java (Servlets/JSP), die übrigen mit anderen Sprachen wie Python oder Perl. Im Bereich der Datenbank-Backends setzen die meisten (deutlich mehr als 50%) auf relationale Datenbanken auf, allen voran MySQL, während objektorientierte, objektrelationale und XML-Datenbanken einen eher geringen Anteil ausmachen. Das Verhältnis von kommerziellen zu freien (GPL- oder ähnliche Lizenz) Produkten beträgt in etwa 1:2.

Viele Systeme ähneln sich, gerade in den hier relevanten Aspekten, konzeptuell sehr, darüber hinaus sind viele der Systeme entweder noch im Prototypstatus oder auf Spezialfälle (z. B. Dokumentenmanagement oder Weblogging, „Blog“) ausgelegt und im allgemeinen CMS-Kontext eher nicht von Interesse. Zudem würde eine detaillierte Untersuchung der Eigenschaften auch nur eines Teils der Systeme den Rahmen der Arbeit sprengen, und oftmals sind bei kommerziellen Systemen keine bzw. wenig Informationen über die zugrunde liegenden Konzepte verfügbar. Aus diesen Gründen werden an dieser Stelle nur die wichtigsten, d. h. am weitesten verbreiteten, Systeme stellvertretend für die jeweils zugrunde liegenden Konzepte untersucht und diskutiert.

Allen existierenden Systemen ist gemein, dass sie die eigentlich hierarchisch strukturierten Daten „flach“ modellieren, d. h. ohne inhärente Baumstruktur. Dies liegt u. a. darin begründet, dass als Datenbank-Backend in den meisten Fällen ein relationales System verwendet wird und die vorhandenen Systeme diesen Ansatz auch auf die Content-Modellierungsebene übernehmen – in diesem Falle muss der Entwickler die Hierarchie manuell nachmodellieren.

Auch ist es in kaum einem System möglich, neue Content-Typen (Klassen) frei zu definieren, in den meisten Fällen steht nur eine eingeschränkte Menge vorgefertigter Typen zur Verfügung (vereinzelt auch als „Open vs. closed content“ bezeichnet und diskutiert[7]).

Des Weiteren beinhalten gängige Systeme, wenn überhaupt, nur rudimentäre objektorientierte Ansätze; spezialisierte Vererbungskonzepte, die eine inkrementelle Spezifikation von Klassenverbünden erlauben, hat keines der bekannten Systeme. Darüber hinaus sind die Systeme bzw. deren Datenmodelle in der Regel seitenorientiert aufgebaut, d. h. es werden immer komplette Webseiten bearbeitet. Die Granularität kann in den wenigsten Systemen verändert aufgebaut werden. Zudem sind in fast allen Fällen zur Erzeugung unterschiedlicher Ausgabeformate auch entsprechende Ausgabetemplates zu

[6] `http://www.cmsmatrix.org`
[7] `http://www.gadgetopia.com/post/259` und `http://www.gadgetopia.com/post/4670`

erstellen, d. h. bei n verschiedenen Ausgaben und m verschiedenen Zielformaten müssen $n * m$ Templates erstellt werden.

Typo3

Typo3[8] ist eines der am weitesten verbreiteten Opensource-CMS, das nach Angaben der Entwickler auf mehreren Tausend Websites eingesetzt wird. Geschrieben ist es in PHP und setzt auf ein relationales DBMS (MySQL, Oracle, ...) auf. Während das System für herkömmliche Publikationsaufgaben im Rahmen von Websites gut geeignet ist, stößt man im Kontext der oben aufgeführten Wünsche und Anforderungen schnell an Grenzen: das Datenmodell verwendet kaum objektorientierte Ansätze, es gibt nur rudimentäre Unterstützung für Klassen, und eine Vererbung darunter ist nicht möglich. Zudem ist durch die Trennung von Daten in RDBMS und Verhalten in PHP eine Kapselung nur schwer möglich. Darüber hinaus ist die Primärstruktur der Daten fest und es besteht keine Möglichkeit, verschiedene Ansichten (*Views*) auf die Daten zu erstellen. Out-of-the-box ist nur eine HTML-Ausgabe möglich, durch den Einsatz von zusätzlichen Plugins kann PDF generiert werden, andere Ausgabeformate werden nicht unterstützt.

XOOPS

XOOPS[9] (*eXtensible Object Oriented Portal System*) ist ebenfalls ein in PHP geschriebenes CMS, das auf MySQL aufsetzt. Auch wenn der Name ein objektorientiertes Konzept nahelegt, bezieht sich dieser jedoch nur auf die Implementierung des Systems selber und nicht auf das Datenmodell oder die dem Website-Ersteller zur Verfügung stehende Sprache. Tatsächlich sind hier fast keine OO-Aspekte zu finden – man kann lediglich neue Klassen von *einer festen* abstrakten Klasse ableiten. Methoden in Klassen, Kapselung oder Unterstützung für hierarchische Daten existieren nicht.

OpenCMS

OpenCMS[10] ist ein ebenfalls weit verbreitetes Opensource-CMS, welches auf Java-Technologie (Servlets und Java Server Pages) und MySQL aufbaut. Verhalten kann – getrennt von den Daten – in sog. JSP-Taglibs implementiert werden; Klassen und Vererbung im eigentlichen Sinne gibt es nicht, es können allerdings neue Seitentypen in XML Schema definiert werden, die dann mit den Einschränkungen von XML auch abgeleitet werden können.

[8] `http://www.typo3.org`
[9] `http://www.xoops.org/`
[10] `http://www.opencms.org/`

Zope und Plone

Plone[11] ist ein in Python implementiertes CMS, das auf dem Zope[12] Framework aufbaut. Im Unterschied zu einem Großteil der existierenden CMS wird dabei eine objektorientierte Datenbank, die so genannte ZODB (Zope Object Database) eingesetzt. Von Vorteil ist dabei, dass einige OO-Techniken wie Kapselung oder ein (einfaches) Klassenkonzept verwendet werden können. Allerdings ist Vererbung nur auf Instanzenebene möglich (Prototyping, hier *Acquisition* genannt), was, wie im Kontext von Wob 1 erläutert (s. u.), in mancher Hinsicht im CMS-Kontext auch sinnvoll sein kann. Jedoch erschwert die fehlende Vererbung auf Klassenebene eine inkrementelle Spezifikation von Anwendungen. Hinzu kommt, dass die Erstellung neuer Klassen (*Archetypes* genannt) umständlich und nur mit einem Zusatzmodul möglich ist.

Apache Cocoon und Apache Lenya

Ähnlich wie das Verhältnis von Plone und Zope ist Apache Lenya[13] ein CMS, das auf dem XML-basierten Web Publishing Framework Cocoon[14] aufbaut. Es wurde versucht, das Separation of Concerns-Paradigma sowie die Verwendung von XML möglichst durchgehend umzusetzen; zum einen sind sämtliche Eingabedokumente, Stylesheets und auch Konfigurationsdateien in XML-Syntax, zum anderen ist durch den Einsatz der XSP (eXtensible Server Pages)-Templatesprache in Verbindung mit so genannten Logicsheets eine größtmögliche Trennung von Logik, Formatierung und Daten zu erreichen.

Die wichtigsten Aspekte sind dabei:

Pipelines: intern arbeitet Cocoon mit SAX-Events (vgl. Abschnitt 7.3.1) in so genannten *Pipelines*, wie in Abbildung 3.17 dargestellt.

Eine Anfrage veranlasst dabei zu Beginn einen Generator, aus unterschiedlichen Quellen (XML-Datei, Datenbank, ...) SAX-Events zu generieren, die dann an einen *Transformer* geleitet werden. Dieser kann dann die Eingaben mit Hilfe eines (XSLT-)Stylesheets umwandeln und ggf. an weitere Transformer weiterleiten. Am Ende einer solchen Pipeline stehen Aggregator und Serializer, die die SAX-Events in eine serialisierte Darstellung im gewünschten Ausgabeformat umwandeln.

XSP: die in Cocoon eingesetzte Templatesprache XSP wird in Abschnitt 7.3.4 näher beschrieben.

Vorteilhaft in diesem Ansatz ist, dass relativ leicht und einfach verschiedene Ausgabeformate generiert werden können, was dem im Rahmen der vorliegenden Arbeit

[11] `http://www.plone.org`
[12] `http://www.zope.org/`
[13] `http://lenya.apache.org/`
[14] `http://cocoon.apache.org`

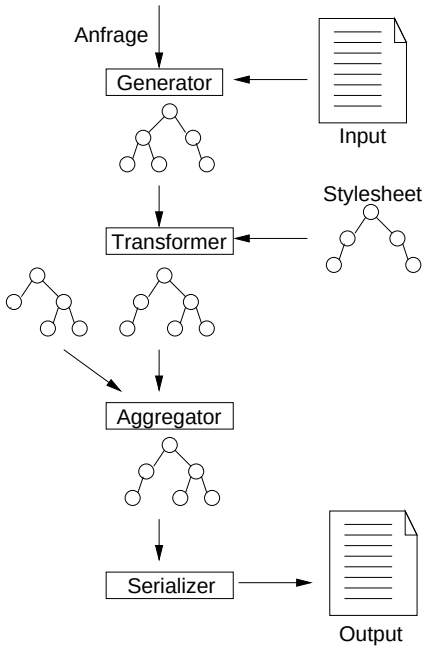


Abbildung 3.17.: Cocoon Pipelines

entwickelten Konzept der Zwischensprache (s. Abschnitt 8.4) ähnelt. Ein Nachteil der Pipeline-Architektur ist jedoch, dass zu Beginn der Pipeline bereits alle benötigten Daten vorhanden sein müssen, d. h. zur Erstellung einer Navigationsleiste o. ä. muss der komplette Datenbaum in die Pipeline „eingespeist“ werden, was einen erheblichen Overhead bedeutet[15]. Durch den Einsatz von XML als Datenmodell kann auch kein Verhalten der Daten modelliert werden und eine Vererbung ist ebenfalls nur in eingeschränkter Form möglich.

Das Wob 1-System

„The Wob - Objects in the Web“[16], das von S. Lüttringhaus-Kappel und P. Lay am Institut für Informatik III der Universität Bonn entwickelt wurde[17], stellt eine wichtige Vorarbeit dar, da die mit diesem System gesammelten Erfahrungen als Grundlage der vorliegenden Arbeit dienten.

Wob 1 ist ein objektorientiertes Web Content Management System, dessen Pflege und

[15] Eine mögliche Lösung hierzu wäre die Implementierung eines speziellen Generators, dadurch würden jedoch die Belange („Separation of Concerns“) vermischt werden.

[16] `http://wob.iai.uni-bonn.de`

[17] Eine Beschreibung des Systems sowie eine Erweiterung ist in [Lay99] zu finden.

Erweiterung rein browser-basiert erfolgt. Das System ist in Prolog geschrieben und setzt auf eine (eigenentwickelte) Prolog-Datenbank auf. Prolog wurde seinerzeit gewählt, da sich Syntax und Semantik von Sprachen bzw. Datenmodellen hiermit gut prototypisch realisieren und evaluieren lassen. Mit dem System wurden seit 1997 viele Webanwendungen inner- und außerhalb der Universität Bonn erfolgreich umgesetzt.

Im Wob 1-System wurde eine funktionale, ungetypte Templatesprache (als domain specific language, s. Abschnitt 7.3) implementiert. Sie bietet neben Kontrollstrukturen (`if`, `case`, `foreach`) knapp 100 eingebaute Funktionen und Operatoren zur Formatierung und Verarbeitung der Daten aus der Wob-Datenbank. Die Sprache ist eine interpretierte universelle Programmiersprache, welche sich durch eine einfache Syntax, bestehend aus Prolog-Termen und Operator-Definitionen, auszeichnet; higher-order Funktionen sind ebenfalls möglich.

Sowohl Daten als auch Methoden werden im Wob 1-Datenmodell in Attribut-Wert-Paaren abgelegt und somit analog zum λ-Kalkül (vgl. Abschnitt 2.5) gleich behandelt, d. h. Methoden sind Daten.

Objekte sind Instanzen einer Klasse und können ein Vaterobjekt sowie beliebig viele Kindobjekte besitzen (Einfachvererbung). Zusätzlich zu der in objektorientierten Systemen üblichen Klassenhierarchie können so auch Instanzenhierarchien aufgebaut werden wie Abbildung 3.18 veranschaulicht. Die so entstehenden Bäume sind demnach ein fester Bestandteil des Modells.

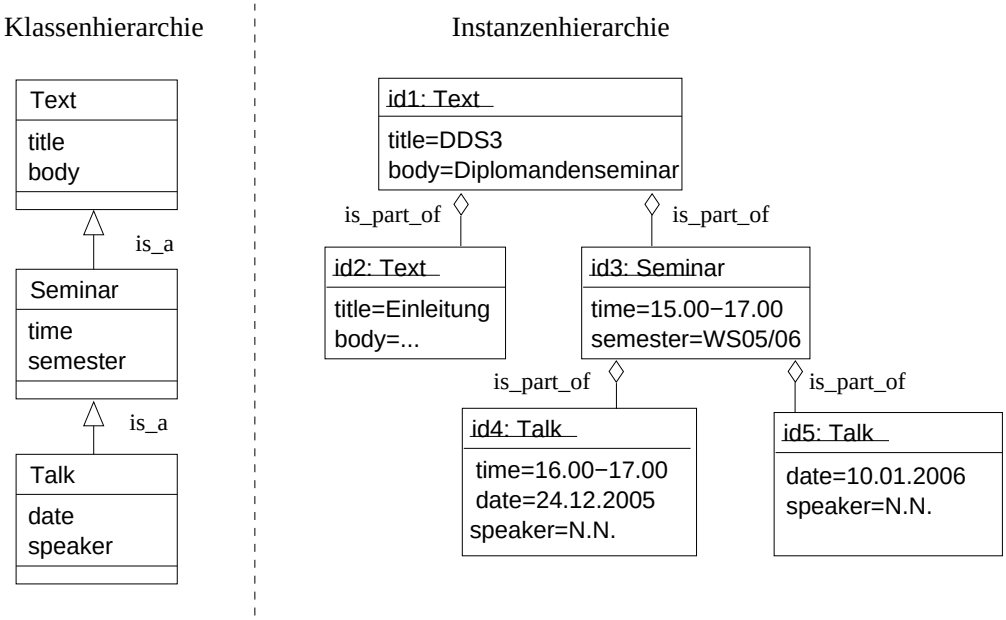


Abbildung 3.18.: Wob 1-Vererbungskonzept: Klassen- vs. Instanzenhierarchie

Wird auf ein Objekt-Attribut zugegriffen, welches im aktuellen Objekt nicht definiert ist, so sucht das System zunächst entlang der Instanzenhierarchie, also entlang der Vaterachse. Wird auch dort kein Wert gefunden, wird entlang der Klassenhierarchie gesucht.

Ein Vorteil der Vererbung entlang der Instanzenhierarchie ist, dass in einem Vaterobjekt ein Default-Wert gesetzt werden kann, welcher an die Kindobjekte vererbt wird und dort ggf. überschrieben werden kann. Im Beispiel (Abbildung 3.18) wird die Zeit von Seminarvorträgen (15.00-17.00) vom Seminar-Objekt id3 geerbt, kann aber in Ausnahmefällen von einzelnen Vorträgen überschrieben werden (z. B. von Objekt id4). Darüber hinaus sind in einem Kindobjekt alle Attribute sämtlicher Vorgängerobjekte verfügbar, d. h. als Anwendungsentwickler muss man nicht überlegen, in welchem (Vater-) Objekt ein bestimmtes Attribut definiert wird, um darauf zugreifen zu können.

Ein Nachteil bei der Vererbung entlang der Instanzenhierarchie ist, dass ohne zusätzlichen Aufwand keine statische Typüberprüfung möglich ist, denn durch die dynamische Hierarchie kann zur Übersetzungszeit nicht überprüft bzw. garantiert werden, dass ein „Vorfahre“ des aktuellen Objekts das entsprechende Attribut besitzt. Zudem kann, selbst wenn das Attribut in beiden Objekten vorhanden ist und denselben Typ hat, nicht garantiert werden, dass die Semantik dieselbe ist. Beispiel: in einem Objekt vom Typ `Vortrag` hat ein Attribut `Titel` vom Typ `String` eine andere Semantik (Vortragstitel) als in einem Objekt vom Typ `Person` („Dr.“, „Prof.“, ...).

Zusammenfassend lässt sich sagen, dass in Wob 1 einige Konzepte umgesetzt wurden, die eine geeignete Modellierung von CMS-Daten ermöglichen, dennoch fehlen einige Aspekte wie spezialisierte Vererbung, Kapselung oder eine einfache Möglichkeit der Ausgabe in verschiedenen Formaten.

3.4. Anforderungen an ein geeignetes Datenmodell für CMS

Wie in Abschnitt 2.4 erwähnt, sollte ein Datenmodell die zu modellierenden Daten möglichst strukturtreu abbilden, um unnötige bzw. komplizierte Mappings (d. h. Abbildungen zwischen Modell und Daten) zu vermeiden. Abschnitt 3.2 zusammenfassend, stellen sich also folgende Anforderungen an ein geeignetes CMS-Datenmodell:

A1 Hierarchien: Hierarchische Strukturen sollen modelliert werden können, wobei verschiedene Sichten auf die Primärstruktur möglich sein sollen. (s. Abschnitt 3.2.1)

A2 Content-Typen: Es soll die Möglichkeit geben, Typen (Klassen) von Entitäten definieren und instanziieren zu können. (s. Abschnitt 3.2.2)

A3 Vererbung: Klassen sollen zwecks inkrementeller Spezifikation vererbt werden können. (s. Abschnitt 3.2.2)

A4 Erweiterte Vererbung: Über die normale Vererbung hinaus soll die Möglichkeit bestehen, ganze Klassenstrukturen ableiten zu können. (s. Abschnitt 3.2.4)

A5 Substrukturen: Substrukturen sollen modelliert werden können, um die Granularität von Webseiten leicht verändern und verschiedene Ansichten auf die Daten generieren zu können. (s. Abschnitt 3.2.3)

Darüber hinaus soll ein geeignetes Datenmodell natürlich die bewährten, grundlegenden CMS-Konzepte unterstützen:

A6 Templates: Eine Templatesprache soll eine einfache Generierung von Text und XML ermöglichen.

A7 Single source, multiple target: Die Ausgabe in unterschiedlichen Zielformaten soll mit geringem Aufwand möglich sein.

Durch die Umsetzung dieser Anforderungen wird der Designprozess neuer Anwendungen und Webauftritte für den Entwickler effizienter und einfacher, da das Modell zum einen die Daten strukturtreu abbildet und zum anderen eine weitgehend redundanzfreie Spezifikation der Anwendungen ermöglicht. Wie im Verlauf der Arbeit gezeigt werden wird, ist darüber hinaus der Einsatz weiterer objektorientierter Konzepte wie Methoden, Kapselung und Information-Hiding im Kontext von CMS überaus sinnvoll.

3.4.1. Existierende Ansätze im Kontext der Anforderungen

Die Datenmodelle existierender CMS sind meist durch das dahinterliegende Datenbanksystem geprägt: so erlauben die bis auf wenige Ausnahmen verwendeten relationalen Datenbanken nur bedingt eine hierarchische Modellierung der Daten. Einige CMS setzen XML ein, jedoch nur als Bausteine der einzelnen Seiten, d. h. es wird nicht durchgängig ein hierarchisches Datenmodell verwendet bzw. XML wird fast ausschließlich dokumenten- und nicht datenzentriert eingesetzt. Weiterhin existieren keine bzw. nur geringe Re-Use-Möglichkeiten, da keine bzw. nur rudimentäre Klassen- bzw. Vererbungskonzepte existieren. Insbesondere sind auch keine *erweiterten* Vererbungskonzepte vorhanden, die es erlauben würden, Anwendungen aus anderen Anwendungen zu erzeugen bzw. abzuleiten. In existierenden CMS ist es daher nur bedingt möglich, Hierarchieeigenschaften in Bezug auf Wiederverwendbarkeit und Vermeidung von Redundanzen zu verwenden. Auch ist in der Regel die Primärstruktur insofern fest, als dass alternative Sichten bzw. Navigationsmöglichkeiten darauf kaum oder gar nicht erstellt werden können.

Darüber hinaus sind die meisten Systeme sehr auf die Publikation von Webseiten und damit unflexibel ausgerichtet, was sich u. a. darin zeigt, dass zumeist immer ganze Seiten bearbeitet werden können bzw. müssen – eine andere Granularität, also der Aufbau von Seiten aus kleinen Bausteinen, ist kaum bzw. nur umständlich möglich. Des Weiteren ist

die Ausgabe in verschiedenen Zielformaten größtenteils mit viel Redundanz verbunden, da in der Regel pro Sprache und Ausgabe ein separates Template implementiert werden muss.

Das Fazit ist somit, dass existierende Systeme nur sehr unflexible Datenmodelle haben, die in erster Linie auf dokumenten- und designorientierte Verarbeitung von Daten abzielen. Ein neues Datenmodell ist somit angebracht und wünschenswert. Nun stellt sich die Frage, ob zur Erfüllung der oben aufgestellten Anforderungen entweder *Evolution*, d. h. die Weiterentwicklung bestehender Ansätze oder *Revolution*, d. h. die Entwicklung eines neuen Datenmodells, der richtige Weg ist. Ausgangspunkt eines evolutionären Ansatzes wäre eines der in Abschnitt 2.4 vorgestellten Datenmodelle, die nachfolgend in Kontext der Arbeit diskutiert werden.

Obwohl im *hierarchischen Modell* die inhärente Struktur von Websites kanonisch abgebildet werden kann, eignet sich das Modell aus verschiedenen Gründen nicht für den Einsatz im Rahmen eines CMS:

- Die prozedurale, satzorientierte Anfragesprache stellt keine Iterationsmöglichkeit zur Verfügung, so dass diese manuell vom Benutzer bzw. Entwickler durchgeführt werden muss.
- Navigationsoperationen sind nur umständlich unter Verwendung von Aktualitätszeigern möglich, darüber hinaus ist die Semantik dieser Operationen oftmals nur schwer durchschaubar.
- Objektorientierte Konzepte wie Klassen/Instanzen oder Vererbung sind nicht vorhanden.

Im *Netzwerkmodell* ist, ähnlich wie im hierarchischen Modell, die Navigation umständlich, darüber hinaus sind auch hier keine objektorientierten Konzepte wie Klassen/Instanzen oder Vererbung vorhanden.

Der große Vorteil des *relationalen Ansatzes* ist die weite Verbreitung und die Effizienz. Im Kontext von Websites bzw. CMS ist jedoch von Nachteil, dass hierarchische Strukturen nicht direkt im Modell, sondern nur über Umwege wie beispielsweise Hilfsrelationen abgebildet werden können. Um die hierarchischen Strukturen in einem RDBMS speichern zu können, ist also eine Abbildung (*Mapping*) nötig. Mitunter wird auch der Ansatz verfolgt, Hierarchie- und Datenstrukturen voneinander getrennt zu verwalten, d. h. eine Baumstruktur repräsentiert die hierarchischen Informationen und die Blätter des Baumes verweisen auf die (in einem RDBMS) „flach verwalteten“ Datensätze, wie Abbildung 3.19 illustriert.

Aber auch diese Herangehensweise ist nicht optimal und stellt nur eine unsaubere Modellierung dar, denn bei der Verarbeitung der Daten und Erstellung der anzuzeigenden Webseiten müssen entweder jeweils zwei verschiedene Datenstrukturen berücksichtigt werden, was sich u. a. in uneinheitlichen Pflegeschnittstellen oder uneinheitlichen Ausgaben niederschlagen kann, oder es muss ein geeignetes API implementiert werden.

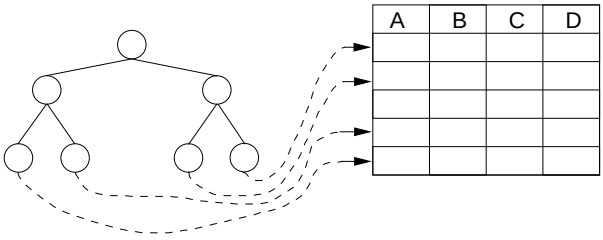


Abbildung 3.19.: Getrennte Verwaltung von Hierarchie- und Datenstrukturen

Der *objekt-relationale Ansatz* erfüllt zwar mit Methoden, Vererbung und geschachtelten Relationen in Ansätzen einige der Anforderungen, jedoch sind die grundlegenden Datenstrukturen immer noch Tabellen, so dass Hierarchien nachmodelliert werden müssten.

Ein häufig verwendetes Datenmodell im Kontext von CMS ist *XML*, mit dem die (semistrukturierten) Daten von Webseiten geeignet beschrieben werden können. Wie verschiedene Quellen (u. a. „XML is not object oriented“[18], „Desperately Seeking...Help for XML Schema“[19]) sowie eine Untersuchung im Rahmen der vorliegenden Arbeit (vgl. Abschnitt 7.2) zeigen, scheidet es jedoch im Hinblick auf die Anforderungen an ein geeignetes Datenmodell im CMS-Kontext aus verschiedenen Gründen aus:

- objektorientierte Konzepte wie Vererbung, Kapselung oder Familienkonzepte fehlen weitgehend
- die XML-Daten können kein Verhalten haben, d. h. der Zugriff auf die Daten muss über externe Schnittstellen bzw. Sprachen erfolgen – um in diesen Sprachen mit den Daten arbeiten zu können, müssen die Daten ggf. zunächst umständlich aus der Baumdarstellung in eine andere (Objekt-) Darstellung transformiert werden (Marshalling)
- der Datenzugriff in großen Datenbeständen wird schnell ineffizient, da die Daten in *einem* großen Baum verwaltet werden.

Mertz fasst dies im Artikel „XML and Data Models: Hierarchical, Relational and Object-Oriented“[20] zusammen:

> „XML is an extremely versatile data transport format; but despite some high hopes for it, XML is mediocre to poor as a data storage and access format.“

[18] `http://norman.walsh.name/2003/06/01/xmlnotoo.html`
[19] `http://xml.sys-con.com/read/40412.htm`
[20] `http://gnosis.cx/publish/programming/xml_matters_8.html`

Genau dieser Gedanke wird in der vorliegenden Arbeit umgesetzt: XML wird als Transport- bzw. Ex/Importformat verwendet, als eigentliches Datenmodell jedoch nicht.

Als Datenmodell kommt nach der obigen Diskussion einzig das objektorientierte für den Einsatz in Rahmen der vorliegenden Arbeit in Frage, kombiniert mit einigen ausgewählten Aspekten der anderen Modelle.

In Kapitel 7 werden verschiedene evolutionäre Ansätze diskutiert, bei denen bestehende Konzepte erweitert werden, um die Anforderungen umsetzen zu können. Der Nachteil dieser Herangehensweise ist jedoch, dass eben verschiedene, eigentlich aus anderen Kontexten erwachsene, Technologien „verbogen“ werden, um in diesen Kontext zu passen. Sowohl die klassischen Datenmodelle als auch programmiersprachliche Ansätze wie Java sind von den eigentlichen Anforderungen im Kontext der vorliegenden Arbeit aber konzeptionell zu weit entfernt, um sinnvoll zu sein. Weitaus eleganter und auch für den Benutzer verständlicher ist es daher, ein neues Modell zu konzipieren, wobei lediglich einige aus anderen Kontexten bekannte und bewährte *Ideen* verwendet und adaptiert werden.

4. Konzeption eines Objektmodells für Content Management Systeme

Nachdem im vorangegangenen Kapitel zum einen die Anforderungen an ein geeignetes CMS-Datenmodell aufgestellt wurden und zum anderen motiviert wurde, warum gängige Systeme bzw. Konzepte diese Anforderungen nur unzureichend erfüllen, folgt nun in diesem und in den anschließenden Kapiteln die Konzeption eines neuen Datenmodells, genannt *Wob 2 Object Model* (W2OM), und einer dazugehörigen Sprache, der *Wob 2 Language* (W2L). Dabei werden in diesem Kapitel die grundlegenden Ideen und deren Motivation präsentiert, die Details werden in den darauffolgenden Kapiteln spezifiziert. Der Entwurf findet auf konzeptueller bzw. logischer Ebene statt; eine Betrachtung der physischen Abbildung auf konkrete Datenbanken (*mapping*) folgt in Kapitel 8.2.

Anzumerken ist, dass das entwickelte Konzept die Einhaltung der Anforderungen *unterstützt*, aber nicht unbedingt *erzwingt* – ähnlich wie man beispielsweise mit Java auch rein imperativ und nicht-objektorientiert programmieren kann, kann man das hier spezifizierte Modell auch „falsch“ verwenden und so Redundanzen und unsaubere Datenmodellierungen produzieren. Das Modell *in Verbindung* mit der richtigen Methodik ist somit die Antwort auf die Anforderungen.

4.1. Content-Verhalten

Nachdem bislang die Content-*Strukturen* analysiert wurden, beschäftigt sich dieser Abschnitt mit der Präsentation der Daten, d. h. wie die Daten zur Erzeugung von Webseiten verwendet werden.

Die Analyse existierender CMS zeigte, dass diese die in Datenbanken oder Dateien persistent gespeicherten Daten (anhand von Templates) lesen und aufbereiten, d. h. die CMS sammeln bzw. ordnen die Daten und erzeugen daraus ein bestimmtes Ausgabeformat (HTML, PDF, LaTeX, ...). Das so generierte Dokument wird (via Internet) an den anfragenden Client geschickt, der diese Daten dann in geeigneter Form darstellt (rendert). Die Daten werden also rein lesend und „passiv“ genutzt. Dabei existiert eine Menge von Templates und eine Menge von (Roh-) Daten, wobei die Templates aus den Daten die Webseiten generieren und beliebig auf die Daten zugreifen können. Demnach ist die Intention, *was* mit den Daten geschehen soll, im System verstreut, so dass Inkonsistenzen auftreten können, wenn die Templates die Daten unterschiedlich interpretieren. Beispielsweise könnte ein Template eine „nackte“ Zahl einlesen und als Währung

interpretieren, ein anderes Template als Längenangabe.

Ein besserer Ansatz ist es, wenn der Zugriff auf die Daten in Klassen gekapselt werden kann und deren Instanzen selber Verhalten haben können, um sich z. B. selber darstellen zu können. Das CMS muss dann lediglich den Daten mitteilen, dass sie sich darstellen sollen, ohne über das „wie“ Bescheid zu wissen. Der Zugriff auf die Daten erfolgt dann immer nur kontrolliert über die Methode.

Da nach den Definitionen aus Abschnitt 2.3.1 ein Objekt eine Datenstruktur mit Zustand, Verhalten und Identität ist und diese Eigenschaften für die hier vorkommenden Strukturen als ausgesprochen sinnvoll erkannt wurden, wird im Folgenden weiter von *Content-Objekten* gesprochen. Details der im Rahmen der vorliegenden Arbeit konzipierten Sprache folgen in Abschnitt 4.4. Man kann verschiedene Arten des Verhaltens von Content-Objekten identifizieren:

- Da die Hauptaufgabe eines CMS ist, Inhalte in verschiedenen Formaten auszugeben, sind *Publikations-* (oder: Darstellungs- bzw. Visualisierungs-) Methoden die wichtigste Form von Verhalten.

- Unter *Kommunikationsverhalten* kann man zum einen die Kommunikation zwischen Content-Objekten verstehen, bei dem ein Objekt die Methode eines anderen aufruft. Beispielsweise könnte ein Objekt die Methode `getTitle()` auf allen seinen Kindobjekten aufrufen, um so aus den jeweiligen Überschriften eine Navigationsleiste o. ä. zu generieren. Eine andere Form der Kommunikation ist die Interaktion mit dem Benutzer, z. B. die Reaktion auf „Signale von der Benutzeroberfläche“ wie beispielsweise einen Klick auf einen Link. Anders als in gängigen CMS, bei denen eine zentrale Instanz auf Benutzeranfragen reagieren muss, kann also im hier entwickelten Konzept das Content-Objekt selbst darauf reagieren.

- *Zustandsänderungen*, bei denen der Zustand eines Content-Objekts verändert wird, können auch als eine Form von Verhalten angesehen werden.

4.2. Objekt- und Vererbungsmodell

Im Kern stellt das W2OM ein hierarchisches Klassen-/Objektmodell zur Verfügung. Dabei enthalten Klassen Zustand (in Form von Attributen) sowie Verhalten (in Form von Methoden) und definieren neue (Content-) Typen. Klassen können voneinander abgeleitet werden und (inkrementell) spezialisiert bzw. erweitert werden – Abbildung 3.11 auf Seite 42 zeigt ein Beispiel einer einfachen Klassenhierarchie. Somit werden die Anforderungen A1 bis A3 (s. Seite 51) nach hierarchischer Struktur, Content-Typen sowie Vererbung erfüllt.

Neu im W2OM ist die Möglichkeit, nicht nur einzelne Klassen, sondern ganze Klassenhierarchien (*Familien* genannt) voneinander ableiten zu können, wie in Abbildung 4.1

gezeigt. Zu sehen sind hier zwei Familien `homepage` und `mysite`, die verschiedene Klassen (`Text`, `Sitemap`, ...) mit Attributen (`title`, `body`, ...) haben; einige Klassen sind abgeleitet (z. B. `Calendar` von `Text` und `MyCalendar` von `Calendar`), der dicke Pfeil in der Mitte repräsentiert die Vererbung zwischen den beiden Familien. Das Besondere bei dieser Familienvererbung ist, dass eben die Struktur, d. h. die Beziehungen der Klassen untereinander, erhalten bleibt, aber trotzdem inkrementelle Änderungen (Hinzufügen von Attributen oder Methoden) möglich sind. Die hiermit verbundene Mehrfachvererbung wird in Abschnitt 4.5 diskutiert.

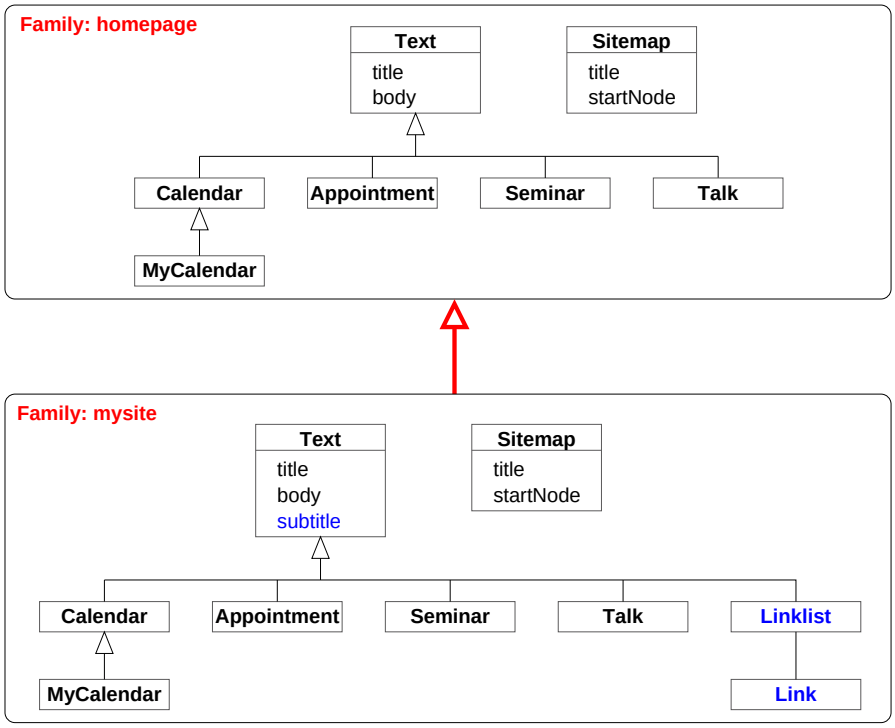


Abbildung 4.1.: Familienpolymorphie

Dabei wird im Kontext der vorliegenden Arbeit ein Webauftritt mit einer Familie gleichgesetzt, die jeweils in sich geschlossen ist in dem Sinne, dass – anders als in verwandten Vererbungskonzepten im OO-Bereich (vgl. Abschnitt 7.4) – keine Klassen unterschiedlicher Familien vermischt werden sollen und können. Eine initiale Familie stellt ein „Grundgerüst“ mit einigen Klassen zur Verfügung, die dann in abgeleiteten Familien erweitert werden können.

Auch bzw. gerade in Verbindung mit Verhalten ermöglicht ein solches Vererbungskonzept eine weitgehend redundanzfreie Spezifizierung: Es lassen sich in Klassen bzw. Fa-

milien gemeinsame Merkmale (Attribute *und* Methoden) von Objekten zusammenfassen und daraus entsprechende Objekte instanziieren, eine abgeleitete Klasse erbt Verhalten von ihrer Superklasse (bzw. Superfamilie) und kann zusätzlich noch weitere Methoden enthalten bzw. vererbte überschreiben.

Aus Gründen der Übersichtlichkeit und leichteren Typüberprüfbarkeit sei festgelegt, dass im W2OM sowohl Felder als auch Methoden in einer Familien-/Klassenhierarchie überall denselben Typ haben müssen, d. h. beispielsweise darf ein Feld `int title` in einer abgeleiteten Klasse bzw. Familie nicht als `String` deklariert werden. Ebenso darf eine Methode `XML view()` nicht von einer Methode `String view()` überschrieben werden.

4.2.1. Klassen- vs. Instanzenhierarchie

Bei der Instanziierung neuer Objekte kann optional die ID des Vaterobjektes spezifiziert werden, wodurch eine Instanzenhierarchie wie in Wob 1 (vgl. Abbildung 3.18 auf Seite 50) erzeugt werden kann. Aus den in Abschnitt 3.3 genannten Gründen der Typsicherheit wird im W2OM – im Unterschied zum Wob 1-Modell – eine ausschließlich „klassische“ Vererbung, d. h. nur entlang der Klassenhierarchie, eingesetzt.

Eine Erweiterung des Modells umfasst die Möglichkeit, *strukturelle Constraints* innerhalb der Instanzenhierarchie zu spezifizieren, um beispielsweise Instanzen der Klasse `Termin` nur unterhalb von Objekten der Klasse `Kalender` instanziieren zu können. Details folgen in Abschnitt 4.6.3.

4.2.2. Materialisierungssemantik

Neu im W2OM ist (im Vergleich zu Vererbungskonzepten im OO-Bereich), dass eine *Materialisierungssemantik* verwendet wird, bei der zur Übersetzungszeit die Klassen materialisiert werden, d. h. Attribute und Methoden werden sowohl aus der Klasse (desselben Namens) in der Superfamilie als auch aus der Superklasse der Familie materialisiert. Dieser Ansatz bietet die folgenden Vorteile:

- Effizienz: die (z. T. komplexen) Vererbungsregeln zwischen Klassen und Familien müssen nur einmal zur Übersetzungszeit und nicht zur Laufzeit angewendet werden.

- Kapselung: die Familien sind in sich „geschlossen“ in dem Sinne, dass sie alle Felder und Methoden enthalten, so dass sie auf Implementierungsebene eine „compilation unit“ bilden und nicht auf andere Familien zugreifen müssen. Dies ist auch deswegen sinnvoll, da sich bei der Analyse der Anwendungsszenarien im Rahmen eines CMS gezeigt hat, dass im W2OM-Kontext üblicherweise nur eine Familie pro Anwendung existiert.

4.2.3. Granularität

Durch die Verwendung von Objekten, die Teile einer Webseite darstellen, können Webseiten aus mehreren „Bausteinen“ zusammengesetzt werden. Dies ermöglicht es, Substrukturen von Websites zu modellieren und die Granularität von einzelnen Webseiten zu verändern. Statt also, wie ein Großteil existierender CMS, dokumentenzentriert ganze Webseiten als Entität zu modellieren, können in W2OM Seiten aus mehreren kleineren Objekten generiert werden.

Anzumerken ist, dass hier (wie auch bei normaler Vererbung) auf eine sinnvolle Granularität bei der Implementierung zu achten ist: Da *einzelne* Methoden nicht inkrementell spezifiziert werden können[1], sondern immer komplett überschrieben werden, würde der inkrementelle Ansatz bei einer einzigen Methode, in der sämtliche Funktionalität enthalten ist, kaum Vorteile bringen (da eben selbst für kleine Änderungen die gesamte Methode reimplementiert werden muss).

4.2.4. Objektidentität

Wie in Kapitel 2 beschrieben, ist ein essentieller Aspekt eines Objektmodells die Möglichkeit der eindeutigen Identifizierung von Objekten über die Objektidentität – Instanzen (d. h. Knoten in der Hierarchie) müssen zum einen referenzierbar und zum anderen wohlunterscheidbar sein.

Zusätzlich zu den in Abschnitt 2.3.1 vorgestellten Ansätzen (identitätsbasiert und wertebasiert) ist im Falle von hierarchischen Daten durch die gegebene Baumstruktur theoretisch noch eine *pfad*basierte Lösung denkbar. Jedes Element in einem solchen Baum ist eindeutig über einen Pfadausdruck[2] identifizierbar, so dass dieser Ausdruck als Identitätsfunktion eingesetzt werden könnte. Ändert sich jedoch die Baumstruktur durch Updates, ist eine solche Zuordnung nicht mehr unbedingt eindeutig, so dass dieser Ansatz im vorliegenden Kontext eher ungeeignet ist. Die (im Falle des W2OM gewählte) bessere Lösung ist also, eindeutige Objekt-IDs zu vergeben, die als unveränderliches Attribut im Objekt gespeichert werden.

4.3. Daten- und Feldklassen

Obwohl u. a. durch das Vererbungskonzept und die damit verbundene inkrementelle Spezifikation Redundanzen vermieden werden können, ist weiteres Potenzial zur Optimierung vorhanden, wie im Folgenden geschildert.

Abbildung 4.2 zeigt die Beispiel-`Text`-Klasse und eine mögliche Implementierung der `view()`-Methode (Details zur Sprache und deren Syntax folgen im weiteren Verlauf der

[1] Eine derartige inkrementelle Spezifikation von Methoden (auch *Submethoding* genannt) wäre als Erweiterung des Modells denkbar, Ansätze in dieser Richtung werden in [MMPN93] und [Red01] vorgestellt.
[2] Im Falle von XML z. B. XPath.

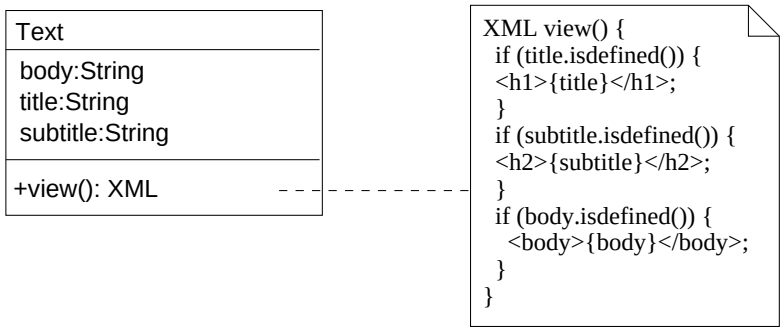


Abbildung 4.2.: UML-Klassendiagramm der Klasse `Text`, 1. Version

Arbeit). Zum einen ist zu beobachten, dass in dieser Methode viel redundanter Code (`if(...isdefined())`, Markup-Tags) vorhanden ist, zum anderen sind offensichtlich die Möglichkeiten für Re-Use eingeschränkt: Wird beispielsweise die Klasse `Text` abgeleitet und um ein weiteres Attribut erweitert, so muss die komplette `view()`-Methode neu geschrieben werden.

Im Rahmen dieser Arbeit wurde nun das bereits beschriebene Datenmodell erweitert bzw. verfeinert, um diesen Nachteil zu umgehen.

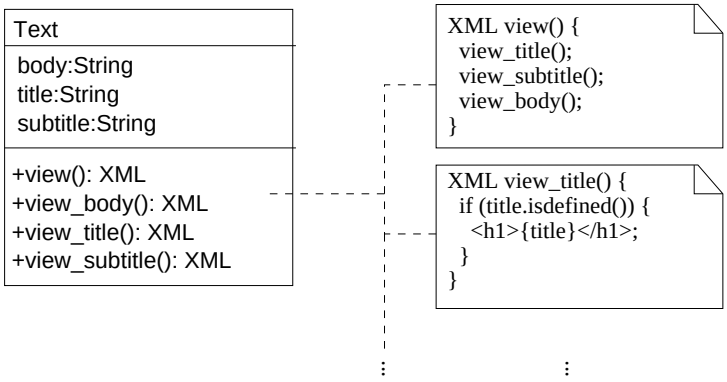


Abbildung 4.3.: UML-Klassendiagramm der Klasse `Text`, 2. Version

Abbildung 4.3 zeigt einen ersten Schritt in Richtung Modularisierung: hier wurde für jedes Attribut eine eigene `view()`-Methode implementiert, welche im übergeordneten `view()` aufgerufen wird. Dieser Ansatz ist geeigneter für Re-Use, da zum einen jede `view`-Methode in einer abgeleiteten Klasse separat überschrieben werden kann und zum anderen beim Hinzufügen eines neuen Attributs lediglich eine entsprechende `view()`-Methode für dieses Attribut implementiert sowie die übergeordnete `view()`-Methode

angepasst werden muss. Bei diesem Ansatz müssen allerdings für n Attribute $n + 1$ view-Methoden implementiert werden.

Eine konsequente Weiterverfolgung dieses Modularisierungsansatzes lieferte die Idee, aus jedem Attribut ein „Miniobjekt“ zu machen, welches sich selbst darstellen kann, wie in Abbildung 4.4 gezeigt – im oberen Teil ist der bisherige, „normale“ Ansatz zu sehen, im unteren der neue. Das Vaterobjekt, im Beispiel `Text`, nennen wir *Datenobjekt*, die „Miniobjekte“ *Feldobjekte*. Derartige Feldobjekte haben also nicht nur, wie bisher, einen Zustand, sondern können auch Verhalten in Form von Methoden haben.

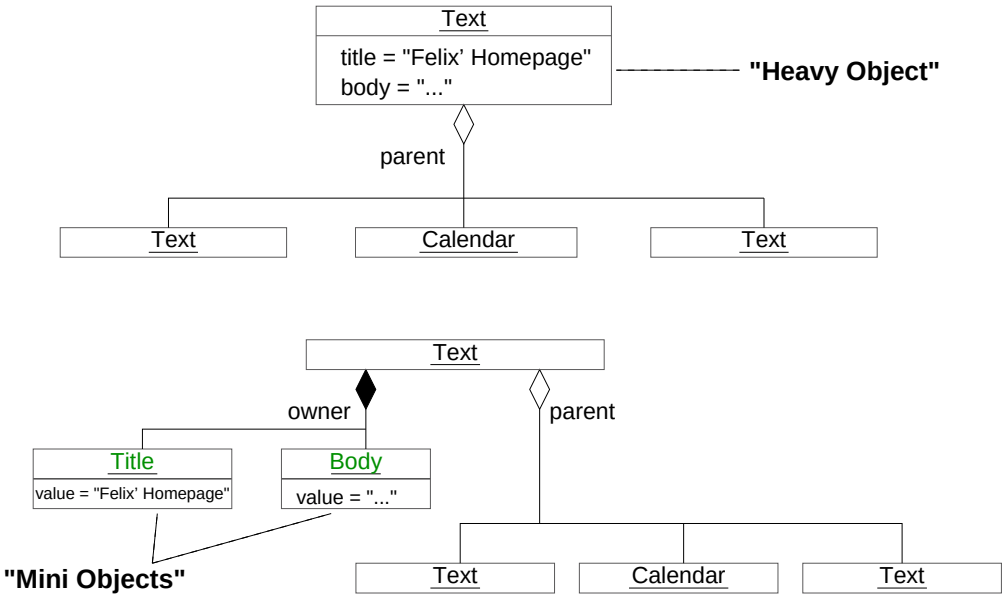


Abbildung 4.4.: Von „Heavy-“ zu „Mini-“ bzw. Daten- und Feldobjekten

Datenklassen definieren ausschließlich Verhalten, der Zustand von Objekten wird nur über die Feldobjekte eines Datenobjekts beschrieben. Darstellungsinformationen sind (überwiegend) in der jeweiligen Klasse der Feldobjekte (den *Feldklassen*) vorhanden; die Superklasse muss lediglich über alle Kinder iterieren und deren `view()`-Methode aufrufen. Abbildung 4.5 zeigt dies in UML-Darstellung. Wird eine neue Feldklasse bei einer abgeleiteten Klasse hinzugefügt, muss nur die `view()`-Methode dieser Feldklasse implementiert werden.

Bei genauerer Betrachtung fällt auf, dass dieser Ansatz eine spezielle Form des Composite-Entwurfsmusters (vgl. [GHJV95]) ist, bei dem die Datenobjekte den Composite-Objekten und die Feldobjekte den Leaf-Objekten entsprechen.

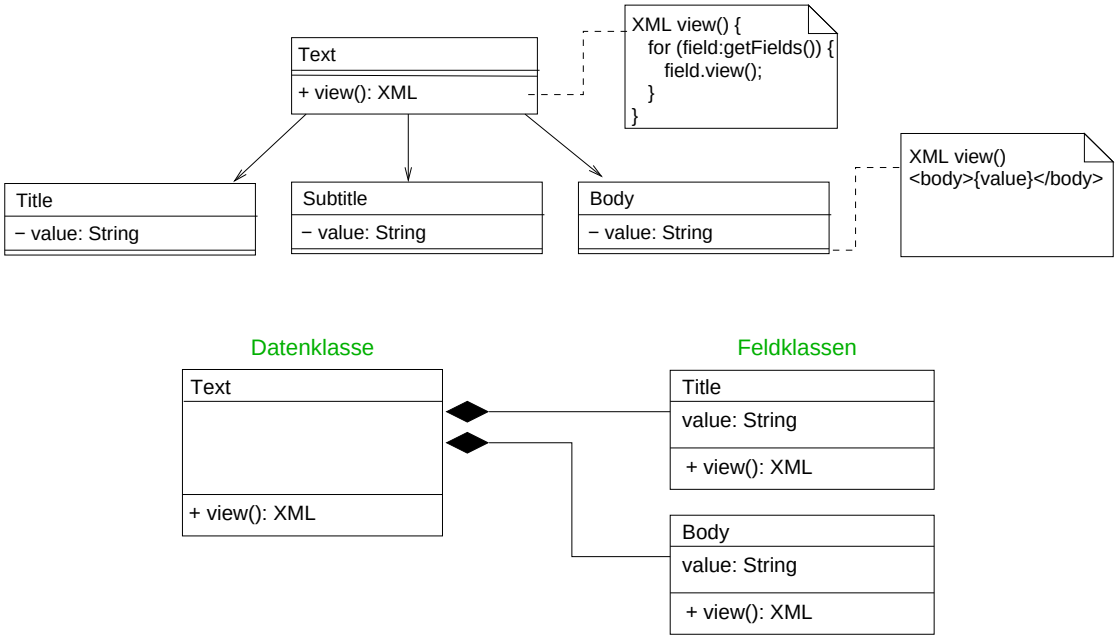


Abbildung 4.5.: Daten- und Feldklassen

4.3.1. Feldklassen und -objekte

Feldklassen sind im Prinzip Basisdatentypen, welche mit Methoden angereichert werden können. Der Zustand eines Feldobjekts ist in erster Normalform (1NF, vgl. Abschnitt 2.4.3), d. h. Feldobjekte enthalten nur einen Wert eines elementaren/atomaren Basisdatentyps. Die Motivation hierfür ist, dass mit einfach dekomponierten Objekten verschiedene Aufgaben wie beispielsweise die Eingabe- oder die (in einer zukünftigen Version des Systems geplante) Versionskomponente deutlich einfacher zu lösen sind. Die Folge hiervon ist, dass Kollektionen wie beispielsweise Hashmaps oder Listen zwar (temporär) in W2L verwendet werden können, aber nicht in den (persisten) Feldobjekten.

Um nicht für jeden Zustandwert eine eigene Feldklasse anlegen zu müssen, besteht die Möglichkeit, *anonyme* Feldklassen zu definieren, welche lediglich Methoden zu bestehenden (Feld-)Klassen hinzufügen. Anonyme Feldklassen können demnach auch nicht weiter abgeleitet werden.

Im Folgenden werden die Begriffe *Datenklasse* bzw. *Feldklasse* benutzt, wenn jeweils nur die entsprechende Struktur gemeint ist und der Begriff *Klasse* als Oberbegriff für Daten- und Feldklassen, wenn der Sachverhalt sich auf beide Konstrukte bezieht.

4.4. W2L

4.4.1. Überlegungen zur Wahl einer geeigneten Templatesprache

Nachdem im vorigen Abschnitt motiviert wurde, dass die objektorientierte Modellierung von Verhalten im Kontext eines CMS-Datenmodells von Vorteil ist, wird im Folgenden analysiert, welche Eigenschaften einer Sprache für CMS in Verbindung mit dem oben beschriebenen Datenmodell sinnvoll bzw. erforderlich sind und welche Sprache demnach zum Einsatz kommen soll und im Rahmen des vorliegenden Arbeit konzipiert wurde.

Wie bereits erläutert, ist die Hauptaufgabe eines CMS, verschiedenartige Inhalte in unterschiedlichen Formaten an Clients auszuliefern. Klar im Vordergrund steht also der Publikationsaspekt, d. h. eine Sprache für ein CMS muss in erster Linie lesend auf die Inhalte zugreifen und daraus eine Ausgabe generieren. Daher sollten Syntax und Befehlsumfang der Sprache eine einfache Erzeugung bzw. Aggregation von Inhalten ermöglichen. Wie schon in Abschnitt 3.1.1 erwähnt, haben sich Templates bzw. Templatesprachen bereits in vielen CMS als dazu gut geeignet erwiesen, so dass diese auch im W2OM eingesetzt werden.

Ein lesender (und demnach nicht zustandsverändernder) Zugriff auf die Daten, bei dem eine Abbildung von einer Eingabe[3] auf eine Ausgabe vorgenommen wird, legt die Verwendung einer funktionalen Sprache nahe. Auch sind funktionale Sprachen gut für die Verarbeitung von hierarchischen Baumstrukturen geeignet. Darüber hinaus lässt eine funktionale Sprache sich, dank Seiteneffektfreiheit und referenzieller Transparenz, effizient implementieren – beispielsweise können Templates und Teilausdrücke parallel ausgewertet bzw. zwischengespeichert (*Caching*) werden, was insbesondere in Hinblick auf Multithreading und die aktuellen Entwicklungen in Richtung Multi-Kern-CPUs auf Server-Rechnern interessant ist. Die Seiteneffektfreiheit garantiert zudem die Sicherheit, dass kein Programm ungewollte Änderungen am Zustand vornehmen kann, im Gegensatz zu beispielsweise PHP oder anderen populären Sprachen im CMS-Kontext, bei denen in der Regel auch bei der Ausgabe beliebige Schreibzugriffe auf die persistenten Daten möglich sind.

Wie oben gefordert, soll die Sprache aber auch objektorientierte Aspekte aufweisen, so dass also eine Hybridsprache, die funktionale und OO-Konzepte verknüpft, die beste Lösung ist, da sie sowohl die Vorteile der Objektorientierung als auch der eben genannten funktionalen Ansätze vereint. Objektorientierte Konzepte sind normalerweise jedoch zustandsverändernd und demnach auch seiteneffektbehaftet, so dass eine Kombination eines OO-Datenmodells mit einer funktionalen Sprache zunächst widersprüchlich bzw. einschränkend erscheint. Wie bereits oben erwähnt, sind die meisten CMS-Anwendungen jedoch sehr publikationslastig, so dass eine Änderung des Zustands in der Regel nicht nötig ist (da rein lesender Zugriff erfolgt) und die positiven Auswirkungen der Seiten-

[3] Die Eingabe besteht im Falle eines CMS aus einer Anfrage des Benutzers bzw. des Clients/Browsers an das CMS.

effektfreiheit wie Sicherheit oder Effizienz durch Parallelität überwiegen, auf der anderen Seite aber auch die Vorteile des objektorientierten Ansatzes wie Vererbung, Kapselung und Abstraktion ausgenutzt werden können. Die Ausgabe (die ja einen Seiteneffekt darstellt) erfolgt nicht im funktionalen Teil, sondern nur außerhalb der Modellkerns.

Allerdings existieren auch Fälle, in denen der Zustand verändert werden muss (z. B. bei Updates oder der Warenkorbfunktion in einem Webshop). Aber auch hier ist der funktionale Ansatz von Vorteil, da derartige Änderungen isoliert von den (lesenden) Anfragen durchgeführt werden können und so eine saubere Trennung vorliegt. Dies ist eine praktikable Standardtechnologie, die auch in anderen Ansätzen eingesetzt wird, zum Beispiel in SQL (`SELECT` für den seiteneffektfreien lesenden Zugriff, `UPDATE` für Änderungen), deduktiven Datenbanken (Update-Mengen) oder XUpdate[4] für XML. Bei diesen Ansätzen wird, anders als beispielsweise in Java, immer eine explizite, separate Menge von Änderungen spezifiziert. Im hier konzipierten Modell werden diese *Update-Sequenzen* genannt (s. Abschnitt 8.3).

Demnach ist die Kombination objektorientierter Konzepte wie Hierarchie, Klassen oder Vererbung und der Sicherheit und Effizienz des funktionalen Paradigmas besonders geeignet für ein CMS-Datenmodell. Dies wird im weiteren Verlauf der Arbeit an der konkreten Konzeption und Umsetzung weiter verdeutlicht werden.

XML

Wie in Abschnitt 3.4.1 ausgeführt, ist XML als *Datenmodell* im Kontext der vorliegenden Arbeit nicht geeignet; im Rahmen der hier konzipierten Templatesprache ist der Einsatz von XML an verschiedenen anderen Stellen jedoch nützlich:

Wie bereits angesprochen, ist eine Hauptaufgabe eines CMS, Inhalte in verschiedenen Ausgabeformaten ausliefern zu können („single source, multiple target“). Da ein Großteil der modernen, im Kontext von CMS auftauchenden Zielformate bzw. -sprachen entweder schon in XML-Syntax ist (XHTML, SVG[5], ...) bzw. aus XML generiert werden kann (PDF[6], LaTeX, ...), ist es sinnvoll, wenn die Ausgabe der Templates direkt in XML erfolgt. Darüber hinaus muss allerdings in den meisten existierenden CMS (so auch in Wob 1) für jede Zielsprache ein separates Template entworfen werden, d. h. bei m Zielsprachen und n Klassen müssen insgesamt $m * n$ Templates implementiert werden. Im Rahmen der vorliegenden Arbeit wurde hierfür mit der Einführung einer auf XML basierenden *Zwischensprache* eine bessere, redundanzfreiere Lösung gefunden, bei der nicht mehr $m * n$, sondern nur noch $m + n$ Templates nötig sind. Das Konzept der Zwischensprache wird in Abschnitt 8.4 näher spezifiziert. Des Weiteren werden die oben erwähnten Update-Sequenzen in XML ausgedrückt.

[4] `http://xmldb-org.sf.net/xupdate`

[5] Scalable Vector Graphics (zweidimensionale, skalierbare Vektorgrafiken)

[6] Zum Beispiel über den Einsatz von XSL:FO, den Formatierungsobjekten der XML Stylesheet Language.

Daher soll (wohlgeformtes) XML mit der Templatesprache einfach erzeugt werden können, was u. a. durch einen XML-Datentyp unterstützt wird, wie im Folgenden erläutert.

4.4.2. Typsystem

Wie in Abschnitt 2.3.3 geschildert, hat ein statisches Typsystem einige Vorteile gegenüber einem dynamischen: Zum einen sind entsprechende Programme bzw. deren Typinformationen besser lesbar, vor allem aber können viele Fehler(quellen) bereits zur Übersetzungszeit vom Compiler festgestellt und so frühzeitig vermieden werden. Zudem ist das Laufzeitverhalten dadurch effizienter. Aus diesem Grunde ist das W2L-Typsystem statisch.

Bei Verwendung eines statischen Typsystems kann auch die Wohlgeformtheit von XML-Fragmenten zur Übersetzungszeit zugesichert werden, da auf Grund der Typinformation XML-Fragmente von gewöhnlichen Strings unterschieden (und geparsed) werden können, wie das folgende Beispiel illustriert:

```
XML test1=<a>statisch typkorrekt</a>;
XML test2=<a>statischer Typfehler</b>;
```

In einem dynamischen oder ungetypten System (z. B. Wob 1) könnte ein derartiger Fehler (im Falle von `test2` wird `<a>` nicht mit `</a>`, sondern mit `</b>` beendet) nicht bzw. erst zur Laufzeit entdeckt werden – derartige Fehler können also bereits zur Uebersetzungszeit entdeckt werden, wenn ein String als XML geparsed wird.

4.5. Mehrfachvererbung: Probleme und Lösungen

Abbildung 4.6 zeigt nochmals die beiden Beispielfamilien aus Abbildung 4.1 (auf Seite 59), wobei hier zusätzlich die (impliziten) Vererbungspfade, die durch die Familienvererbung gebildet werden, hervorgehoben wurden. Es ist zu sehen, dass im Familienkonzept eine einfache Form der Mehrfachvererbung durch die impliziten Vererbungspfade vorhanden ist, z. B. existieren zwei Pfade von `homepage.Text` zu `mysite.Appointment`: einmal via `homepage.Appointment` und einmal via `mysite.Text`.

Abbildung 4.7 illustriert dies auf abstrakter Ebene. Die Familie $F1$ enthält eine Klasse $C1$ sowie eine davon abgeleitete Klasse $C2$. Eine von Familie $F1$ abgeleitete Familie $F2$ erbt auf Grund des Familienkonzepts diese Klassenstruktur. Die (in der Abbildung mit einem durchgezogenen Pfeil markierte) Vererbung zwischen den jeweiligen Klassen nennen wir *inner*familiär, die (in der Abbildung mit einem gestrichelten Pfeil gekennzeichnete) Vererbung zwischen $F1.C1$ und $F2.C1$ bzw. $F1.C2$ und $F2.C2$ *inter*familiär. Implizit entsteht so eine Rautenstruktur und es stellt sich die im Folgenden untersuchte Frage, welche Methode beim Aufruf von `F2.C2.m()` verwendet wird.

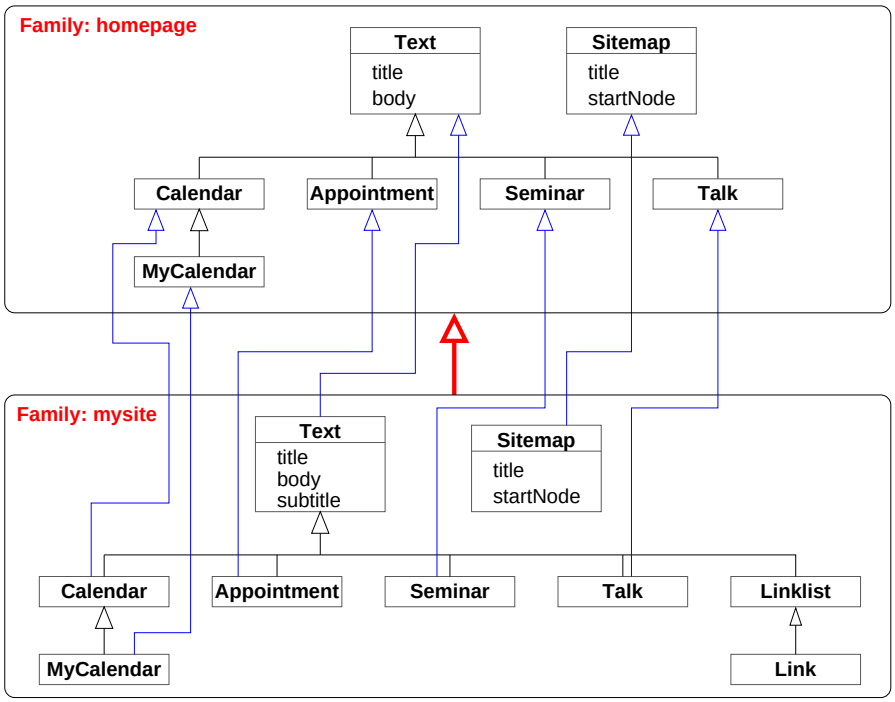


Abbildung 4.6.: Mehrfachvererbung bei Familienpolymorphie (1)

Auflösung von Mehrdeutigkeiten

Um eventuell auftretende Mehrdeutigkeiten aufzulösen, sind verschiedene Ansätze möglich (vgl. hierzu auch Abschnitt 2.3.4):

1. Mehrfachvererbung generell verbieten: eine einmal definierte Methode darf in Subklassen/-familien nicht mehr überschrieben werden. Hierdurch wird zwar statische Typsicherheit garantiert, allerdings gehen auch die Vorteile der (Mehrfach-)Vererbung verloren.

2. Priorisierung: Methoden können Vererbungsprioritäten zugewiesen werden. Diese können entweder manuell vergeben werden oder je nach Anwendung kann entweder die innerfamiliäre oder die interfamiliäre Vererbung priorisiert werden. Eine probeweise Modellierung verschiedener Anwendungen, u. a. im Wob 1-Kontext, ergab, dass sich innerfamiliäre und interfamiliäre Vererbung in vielen Fällen folgendermaßen charakterisieren lassen:

   - innerfamiliär: anwendungsspezifische Informationen (z. B. neue Attribute, Formatierung)

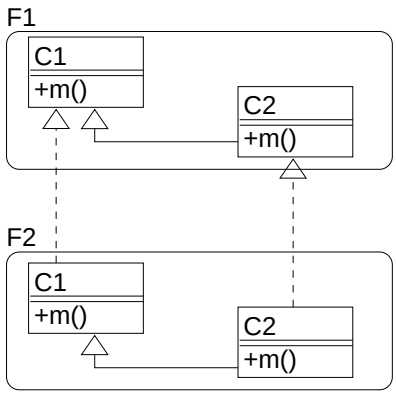


Abbildung 4.7.: Mehrfachvererbung bei Familienpolymorphie (2)

- interfamiliär: generische Informationen (z. B. Bugfixes in Methoden)

3. Entfernungsmaß: ein Ansatz wäre, die spezifischste Implementierung zu verwenden. Diese kann in vielen Fällen über ein Entfernungsmaß gefunden werden. Abbildung 4.8 illustriert dies. Hier wird eine Methode in $C1$ definiert und in $C2$ sowie $C3$ überschrieben. $C3'$ verwendet die Methode aus $C3$, da diese am „nächsten“ liegt. Allerdings sind hier ggf. unerwünschte Effekte denkbar; wenn beispielsweise

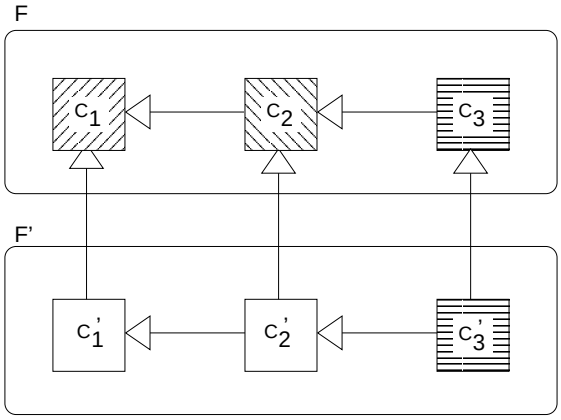


Abbildung 4.8.: Auflösung von Mehrdeutigkeiten über Entfernungsmaß

in Klasse $C1'$ eine Methode implementiert wird, die nicht in $C3'$ „ankommt“, da die Implementierung aus Klasse $C3$ „näher liegt“.

4. Temporaler Ansatz: denkbar ist auch eine Herangehensweise, bei der die (zeitlich)

aktuellste Implementierung einer Methode gewählt wird. Aber auch dieser Ansatz kann höchst unerwünschte Auswirkungen haben, da er auf einer anderen, nicht deklarativen, Ebene arbeitet, d. h. ein Benutzer des Systems muss zusätzlich zu den Deklarationen in Klassen und Familien noch die zeitliche Komponente „im Auge behalten“, um zu wissen, welche Methode nun ausgewählt wird.

Da die Familienvererbung und die damit verbundene Mehrfachvererbung im W2OM ein essentieller Aspekt ist, scheidet Punkt 1 der obigen Liste (Verbot von Mehrfachvererbung) aus, ebenso der temporale Ansatz auf Grund der unerwünschten bzw. unvorhersehbaren Auswirkungen. Eine Analyse verschiedener Anwendungen (im Wob 1-System) zeigte, dass eine automatische Auflösung nicht in allen Fällen zufrieden stellend ist, so dass zur Auflösung von Mehrdeutigkeiten eine Kombination aus (manueller) Priorisierung und Entfernungsmaß gewählt wurde – Details zur Lösung folgen in Abschnitt 5.2.1.

Anmerkung: die hier beschriebene Mehrfachvererbung ist *eingeschränkt* in dem Sinne, dass es immer genau eine explizite Superklasse (in derselben Familie) und eine implizite Superklasse (in der Superfamilie) gibt. Vollständige Mehrfachvererbung, die beispielsweise durch die Angabe mehrerer Superklassen (z. B. `extends [Linklist Calendar]`) spezifiziert werden könnte, ist momentan nicht vorgesehen und bringt im CMS-Kontext wohl auch mehr Probleme als Nutzen.

4.6. Besonderheiten des Objektmodells

Im Folgenden werden einige Aspekte sowie Erweiterungen vorgestellt, die insbesondere beim praktischen Einsatz für mehr Flexibilität und Bedienungskomfort sorgen.

4.6.1. Objektinstanzen und -hierarchien

Datenobjekte, d. h. Instanzen von Datenklassen, können prinzipiell auf zwei Arten instanziiert werden: Zum einen als Datenbankobjekte, d. h. Objekte, die persistent in der Datenbank gespeichert werden. Dies erfolgt über so genannte *Update-Sequenzen* (mehr dazu in Kapitel 8.3). Die zweite Möglichkeit sind Hilfsobjekte, die z. B. zu Berechnungen benötigt werden, und die nicht persistent in der Datenbank gespeichert werden. Auch Feldobjekte können als Hilfsobjekte instanziiert werden. Derartige Hilfsobjekte können mit Konstruktoren erzeugt werden, eine spätere Änderung des Zustandes ist jedoch auf Grund der Seiteneffektfreiheit nicht möglich[7]. Mehr zu Konstruktoren folgt in Kapitel 6.

Um außer einer Klassenhierarchie auch eine Instanzenhierarchie aufbauen zu können (vgl. Kapitel 1 und 4), kann bei der Generierung eines neuen Datenobjekts (als Instanz einer Datenklasse) ein Vaterobjekt spezifiziert werden, Details hierzu folgen in Kapitel 8.

[7] Im Sinne von Java sind also alle Felder `final`.

Die Instanzenhierarchie bietet – wie bereits motiviert – aus Gründen der Typsicherheit nicht die Möglichkeit der Vererbung, sondern dient nur dem Zweck, die (in der Regel hierarchischen) Inhalte der Website geeignet modellieren zu können. Eine derartige Instanzenhierarchie wird im Folgenden auch als *Datenbaum* (im Unterschied zum Klassenbaum) bezeichnet.

4.6.2. Reihenfolge von Objekten

Bei semistrukturierten Daten wie beispielsweise Texten ist die Reihenfolge der einzelnen Objekte in der Regel wichtig und sollte durch eine inhärente Reihenfolge repräsentiert werden (da z. B. eine explizite Angabe der Reihenfolge von Absätzen umständlich wäre). Durch die hierarchische Baumstruktur ist eine solche implizite Reihenfolge sowohl der Feldobjekte eines Datenobjekts als auch der Datenobjekte selber gegeben.

Da es in einigen Fällen sinnvoll sein kann, ist es möglich, diese inhärente Reihenfolge zu verändern, um z. B. die Ausgabereihenfolge anzupassen oder die Reihenfolge der zu den Objekten erzeugten Eingabekomponenten zu ändern. Bei Datenobjekten ist die Reihenfolge von Interesse, wenn auf die Liste aller Kinder eines Datenobjekts zugegriffen wird (z. B. über die `getChildren()`-Funktion, s. Abschnitt 6.3.1) – derartige Listen können dann über Sortierfunktionen manipuliert werden und so in einer anderen als in der durch die Baumstruktur vorgegebenen Reihenfolge auf die Datenobjekte zugegriffen werden. Im Beispiel des Vorlesungsverzeichnisses könnte so beispielsweise die Liste der Lehrveranstaltungen in verschiedenen Sortierungen ausgegeben werden. Bei der Ausgabe von Feldobjekten wird die Reihenfolge folgendermaßen bestimmt:

1. Die höchste Priorität hat eine explizite Spezifikation: in der Methode, die die Felder ausgibt (also beispielsweise eine `view()`-Methode) kann explizit eine Reihenfolge angegeben werden.
   Beispiel:

   ```
   class Text {
     String title;
     XML body;
     XML view() {
       <h1>{title}</h1>
       {body}
     }
   }
   ```

   Hier wird der `title` vor dem `body` ausgegeben.

2. Bei der Iteration über alle Felder mit der `getFields()`-Methode (ohne explizite Angabe einer Reihenfolge) wird die durch die Klassendeklaration implizit vorgege-

bene Ordnung benutzt.
Beispiel:

```
class Text {
  String title;
  XML body;
  XML view() {
    <fields>
    { for (Field myfield: getFields()) {
        myfield.view();
      }
    }
    </fields>
  }
}
```

Auch in diesem Beispiel wird der `title` vor dem `body` ausgegeben.

Dabei werden die in einer abgeleiteten Klasse neu definierten Felder an die der Superklasse angehängt. Um die neuen Felder abgeleiteter Klassen an anderer Stelle in die Liste einzufügen, können bei der Deklaration relative Angaben gemacht werden; `before` fügt das Feld vor und `after` nach einem anderen Feld ein. Weiterhin kann mittels des `order`-Schlüsselworts die Reihenfolge geerbter Felder geändert werden (ohne neue Felder zu definieren)Beispiel:

```
class Text {
  String title;
  XML body;
}

class ExtendedText extends Text {
  String subtitle after title;
}

class UpsideDownText extends Text {
  order title after body;
}
```

Die Reihenfolge der Ausgabe der Klasse `ExtendedText` ist `title`, `subtitle`, `body`. Bei Instanzen der Klasse `UpsideDownText` wird der `body` vor dem `title`-Feld ausgegeben.

Wie im folgenden Listing ersichtlich, kann durch diese Spezifikationsmöglichkeiten eine Teilordnung definiert werden:

```
class C1 {
  Field F1;
}
class C2 extends C1 {
  Field F2 after F1;
}
class C3 extends C2 {
  Field F3 after F1;
}
```

Hier gilt $F1 < F2$ sowie $F1 < F3$; $F2$ und $F3$ lassen sich nicht direkt vergleichen; auf Grund der Materialisierungssemantik kann die Reihenfolge jedoch bestimmt werden:

```
class C1 {
  Field F1;
}
class C2 {
  Field F1
  Field F2;
}
class C3 extends C2 {
  Field F3 after F1;
}
```

Durch die Materialisierung der Klasse C2 wird klar, dass $F3$ zwischen $F1$ und $F2$ eingefügt wird.

4.6.3. Strukturelle Restriktionen der Instanzenhierarchie

Wie sich bereits im Wob 1-System gezeigt hat, ist es in vielen Fällen sinnvoll, als Erweiterung des Konzepts der Instanzenhierarchie gewisse (semantisch motivierte) Einschränkungen in Bezug auf die Struktur des Datenbaumes spezifizieren zu können. Beispielsweise sollte ein einzelner Termin (Instanzen der Klasse `appointment`) nur unter einem Kalender-Objekt (der Klasse `calendar`) angelegt werden dürfen. Eine mögliche Realisierung dieser Restriktionen wird im folgenden skizziert.

Die Restriktionen können in den Datenklassen über verschiedene Eigenschaften mithilfe des `property`-Schlüsselworts zugewiesen werden:

`allowRoot (true|false):` gibt an, ob eine Instanz der entsprechenden Klasse als Wurzel eines Instanzenbaums auftreten darf. Fehlt diese Property, muss jede Instanz als Kindobjekt eines anderen Objektes auftreten. Grundeinstellung ist dabei `false`.

`allowedChildren:` spezifiziert eine Menge von Klassen, deren Instanzen als Kinder der aktuellen Klasse instanziiert werden dürfen. Sonderfälle sind die leere Menge (d. h. keine Unterobjekte möglich) oder das Mengenelement `*` (beliebige Objekte der aktuellen Familie sind erlaubt; dies ist der Default-Wert).

Um Redundanzen zu vermeiden und eine inkrementelle Spezifikation der erlaubten Klassen zu erlauben, werden in einer abgeleiteten Klasse die Mengen beider Klassen vereinigt. Somit gilt

$$\frac{\begin{array}{c} A \lhd B \\ \texttt{allowedChildren(A) = [}c_1\texttt{, ... }c_n\texttt{]} \\ \texttt{allowedChildren(B) = [}c_{n+1}\texttt{, ... }c_m\texttt{]} \end{array}}{\texttt{allowedChildren(B) = [}c_1\texttt{, ... }c_n\texttt{,}c_{n+1}\texttt{, ... }c_m\texttt{]}}$$

Da die Bildung der Vereinigungsmenge in gewissen Fällen unerwünscht sein kann, kann sie mit dem Schlüsselwort `only` unterdrückt werden.

- Beispiel 1: In diesem Beispiel dürfen (nur) Instanzen der Klassen C2 oder C3 als Kinder von C2 auftauchen:

```
class C1 {
  property allowedChildren [C2];
}
class C2 extends C1 {
  property allowedChildren [C3];
}
```

- Beispiel 2: Hier dürfen nur Instanzen der Klasse C3 als Kinder von C2 auftauchen, da die beiden `allowedChildren`-Mengen auf Grund von `only` nicht vereinigt werden:

```
class C1 {
  property allowedChildren [C2];
}
class C2 extends C1 {
  property allowedChildren [only C3];
}
```

`restrictToParent:` spezifiziert eine Menge von Klassen, deren Instanzen als Vater von Instanzen der aktuellen Klasse auftreten dürfen. Fehlt diese Property, dürfen Instanzen der aktuellen Klasse unter beliebigen Vaterobjekten stehen. `restrictToParent` hat dabei eine höhere Priorität als `allowedChildren`. Auch hier wird bei der Vererbung die Vereinigungsmenge der aktuellen Menge mit der der Superklasse gebildet, mit dem Eintrag `"-"` kann dies unterdrückt werden. Beispiel:

```
class C1 {  property allowedChildren [*];  }
class C2 {  property allowedChildren [C2, C3];  }
class C3 {  property restrictToParent [C1];  }
```

Den `allowedChildren`-Properties nach dürften Instanzen von C3 sowohl unter C1 als auch unter C2 stehen, durch die Verwendung der `restrictToParent`-Property wird dies jedoch auf C1 eingeschränkt.

Zusammenfassendes Beispiel

```
1  class Text {
2    property allowRoot true;
3  }
4
5  class Calendar extends Text {
6  }
7
8  class Appointment extends Text {
9    property restrictToParent [Calendar];
10 }
11
12 class Linklist extends Text {
13   property allowedChildren [Link];
14 }
15
16 class Link extends Text {
17 }
```

Nur Instanzen der Klasse `Text` dürfen auf der obersten Ebene stehen (Zeile 2). Instanzen der Klasse `Linklist` dürfen nur Objekte vom Typ `Link` enthalten, während die anderen Klassen beliebige Unterobjekte enthalten dürfen. Instanzen der Klasse Appointment dürfen nur unter Objekten des Typs Calendar auftauchen.

Erweiterungen dieses Ansatzes, beispielsweise die Spezifikation von Schachtelungen, Alternativen oder Kardinalitäten („es müssen mindestens 2, dürfen höchstens 5 Autoren zu diesem Buch existieren“) sind denkbar, bieten aber nur geringen praktischen Nutzen in den typischen Anwendungsszenarien im Kontext von CMS. Falls sich in Zukunft zeigen sollte, dass der hier gewählte Ansatz nicht ausreichend ist, könnten in einer Erweiterung des Modells separat spezifierbare Integritätsbedingungen konzipiert werden.

5. Spezifikation des W2OM-Objektmodells

Nachdem in Kapitel 4 die Kernkonzepte der vorliegenden Arbeit motiviert und eingeführt wurden, folgt in diesem und im folgenden Kapitel eine detaillierte, formale Spezifikation des Datenmodells und der Sprache inklusive Syntax und Semantik. Dieses Kapitel spezifiziert die strukturellen Anteile des Objektmodells, also Datendefinitions- und Vererbungsaspekte, während das folgende Kapitel sich der Datenverarbeitungssprache einschließlich der Kontrollstrukturen und des Typsystems widmet.

5.1. W2L-Syntax (strukturelle Aspekte)

In diesem Abschnitt werden zunächst die strukturellen (d. h. zur Familien- und Klassenspezifikation nötigen) Anteile der Syntax spezifiziert; die Besonderheiten der Ausdrücke und Details zum Typsystem folgen in Kapitel 6.

Die Syntax der W2L ist an die weithin bekannte Java-Syntax angelehnt, u. a. um den Um- bzw. Einstieg in die Sprache zu vereinfachen. Aus Gründen der Übersichtlichkeit wird eine erweiterte EBNF-Syntax verwendet, bei der $\overline{x}$ für die *Menge* $\{x_1, \ldots, x_n\}$, $\vec{x}$ für die *Liste* $x_1, \ldots, x_n$ und $\vec{T}\ \vec{x}$ für die Liste $T_1\ x_1, \ldots, T_n\ x_n$ stehen, mit $n \geq 0$ (s. auch [NCM04]). Terminale sind in normaler, Nichtterminale in kursiver Schrift gesetzt.

Programm	P	::=	$\overline{Family}$
Familie	$Family$	::=	family *Fam* extends *Fam′* { $\overline{Class_F}$ $\overline{Class_D}$ $\overline{Meth}$ }
Feldklasse	$Class_F$	::=	field class C_F extends $C_F{}'$ { $\overline{Meth}$ }
Datenklasse	$Class_D$	::=	class C_D extends $C_D{}'$ { $\overline{Field}$ $\overline{Meth}$ }
Feld	$Field$	::=	*Type F* ({ $\overline{Meth}$ })? ;
Methode	$Meth$	::=	static? *Type* $M(\overrightarrow{Type}\ \vec{X})$ M_{body}
Methodenrumpf	M_{body}	::=	(priority *Prio*; | { $\overrightarrow{Expr}$ })
Priorität	$Prio$	::=	(family | class)
Typ	*Type*		(s. Kapitel 6)
Ausdruck	*Expr*		(s. Kapitel 6)

Abbildung 5.1.: W2L-Syntax: strukturelle Anteile

Abbildung 5.1 zeigt die W2L-Syntax[1]. Ein Programm P besteht aus einer Menge

[1] Dabei sind *Fam* und *Fam′* Bezeichner von Familien, C_F und $C_F{}'$ von Feldklassen, C_D und $C_D{}'$ von Datenklassen, F von Feldern und M von Methoden.

von Familien. Eine Familie Fam enthält Mengen von Feldklassen $\overline{Class_F}$, Datenklassen $\overline{Class_D}$ sowie Methoden $\overline{Meth}$ (*Familien*methoden genannt). Feld- und Datenklassen sowie Methoden[2] werden von einer via `extends` spezifizierten Superfamilie geerbt[3]. Eine Feldklasse C_F erweitert eine Feldklasse $C_F{}'$ und enthält eine Menge von Methoden $\overline{Meth}$ (*Feldklassenmethoden*); eine Datenklasse C_D wiederum beinhaltet eine Menge von Feldern $\overline{Field}$ und eine Menge von Methoden $\overline{Meth}$ (*Datenklassenmethoden*). Ebenso wie Familien können auch Daten- und Feldklassen voneinander abgeleitet werden[4]. Ein Feld F hat einen Typ *Type* und optional eine Menge von Methoden $\overline{M}$ (*Anonyme Feldmethoden*). Eine (ggf. statische) Methode M hat einen Rückgabewert vom Typ *Type*, eine Liste von Parametern $X_1, \ldots, X_n$ vom Typ $Type_1, \ldots, Type_n$ und entweder eine Priorität $Prio$ (welche wiederum „family“ oder „class“ sein kann) oder einen Methodenrumpf als Liste von Ausdrücken $\overrightarrow{Expr}$.

Wie schon erwähnt, werden Typsystem und Ausdrücke in Kapitel 6 spezifiziert, der folgende Abschnitt beschäftigt sich mit der Semantik des W2OM.

5.2. Semantik des Objektmodells

5.2.1. Informelle Beschreibung der Semantik

Familien dienen dazu, zusammengehörige Klassen zu gruppieren. Eine Familie wird auch als Anwendung bezeichnet, da sie eine in sich geschlossene Einheit darstellt, die für die Repräsentation und Darstellung einer konkreten Website verwendet wird. Der Grundgedanke ist, dass derartige Familien voneinander abgeleitet werden können und so Strukturen und Methoden einer bestehenden Anwendung übernommen und um neue Funktionalität erweitert werden können.

Familien bestehen aus Familienmethoden, welche in der gesamten Anwendung verwendet werden können, sowie aus Feld- und Datenklassen. Feldklassen besitzen einen Zustand in Form genau *eines* Datenfelds (z. B. `String` oder `Number`) sowie Verhalten in Form von Feldklassenmethoden. Feldklassen erweitern somit ein Datenfeld um Verhalten und definieren darüber hinaus einen neuen Datentyp.

Demgegenüber können Datenklassen zwar ebenso Verhalten in Form von Methoden haben, ihr Zustand definiert sich allerdings ausschließlich über ihre Felder. Felder sind Feldobjekte als Instanzen einer Feldklasse oder einer *anonymen* Feldklasse – dies sind Feldklassen, die bestehende Datentypen mit Methoden erweitern, ohne einen (neuen)

[2] Da die W2L eine funktionale Sprache ist und somit Methoden auch immer funktionalen Charakter haben, werden die Begriffe Methode und Funktion im Folgenden mitunter synonym verwendet.

[3] Dabei existiert eine triviale Familie Fam^0 ohne Feld-/Datenklassen und Methoden, welche abgeleitet werden kann. Auf diese Weise lassen sich Syntax und Semantik übersichtlicher definieren. Im praktischen Einsatz ist es demgegenüber sinnvoller, die Ableitung via `extends` als optional zu deklarieren.

[4] Hier existieren analog zu Familien triviale Klassen C_D^0 und C_F^0.

Namen zu erhalten und die somit auch nicht weiter abgeleitet werden können. Wird in einer Feldklasse keine Methode definiert, so ist die Felddeklaration identisch zu einer normalen Variablendeklaration in Java, wie das folgende Beispiel zeigt:

```
String title;
```

Im nächsten Beispiel wird der Basisdatentyp `String` um eine Methode `view()` erweitert, deren Rückgabewert ein wohlgeformtes XML-Fragment ist, bei dem der Wert des (Feld-) Objekts in einem `h`-Element steht (zum XML-Datentyp mehr in Abschnitt 6.2.1).

```
String title {
  XML view() {
    <h>{this}</h>
  }
};
```

Sowohl Feld- als auch Datenklassen können voneinander abgeleitet werden, wobei hier nur Einfachvererbung möglich ist. Da auch Familien voneinander abgeleitet werden können, können Klassen sowohl von der (gleichnamigen) Klasse in der Superfamilie (interfamiliär) als auch der Superklasse in der (gleichnamigen) Familie (innerfamiliär) Methoden und Felder erben, wodurch eine einfache Form der Mehrfachvererbung, bei der maximal zwei Vaterklassen existieren können, induziert wird. Zur Auflösung eventuell dabei entstehender Konflikte kann pro Methode definiert werden, welche Art der Vererbung (interfamiliär, durch die Priorität „family“ gekennzeichnet vs. innerfamiliär, durch die Priorität „class“ bezeichnet) verwendet werden soll.

Normalerweise können in der Klasse definierte Methoden nur auf einer entsprechenden Instanz aufgerufen werden; ähnlich wie beispielsweise in Java können Methoden aber auch als statisch deklariert werden und werden so zu Methoden, die ohne Instanz direkt auf der Klasse aufgerufen werden können. Konstruktoren werden als ebensolche statische Methoden modelliert (mehr dazu in Abschnitt 6.1).

Die Vererbung der Werte von (statisch definierten) Feldern erfolgt analog zur Methodenvererbung; Details hierzu folgen in Abschnitt 6.2.2. Aus den in Abschnitt 4.2.1 beschriebenen Gründen der Typsicherheit erfolgt dabei keine (dynamische) Vererbung von Werten entlang des Objektbaumes.

Aus den bisherigen Spezifikationen folgt, dass außer dem Vererbungsgraph unter Familien innerhalb jeder Familie noch Vererbungsgraphen für Feld- und Datenklassen aufgebaut werden. Bedingt durch die partielle Ordnung ist der Vererbungsgraph für Familien ein Baum mit der Wurzel Fam^0 und innerhalb einer jeden Familie existiert ein Vererbungsgraph für Datenklassen mit C_D^0 als Wurzel sowie maximal n (mit $0 \leq$ *Anzahl Basisdatentypen*) Vererbungsgraphen für Feldklassen mit einem der Basisdatentypen als Wurzel.

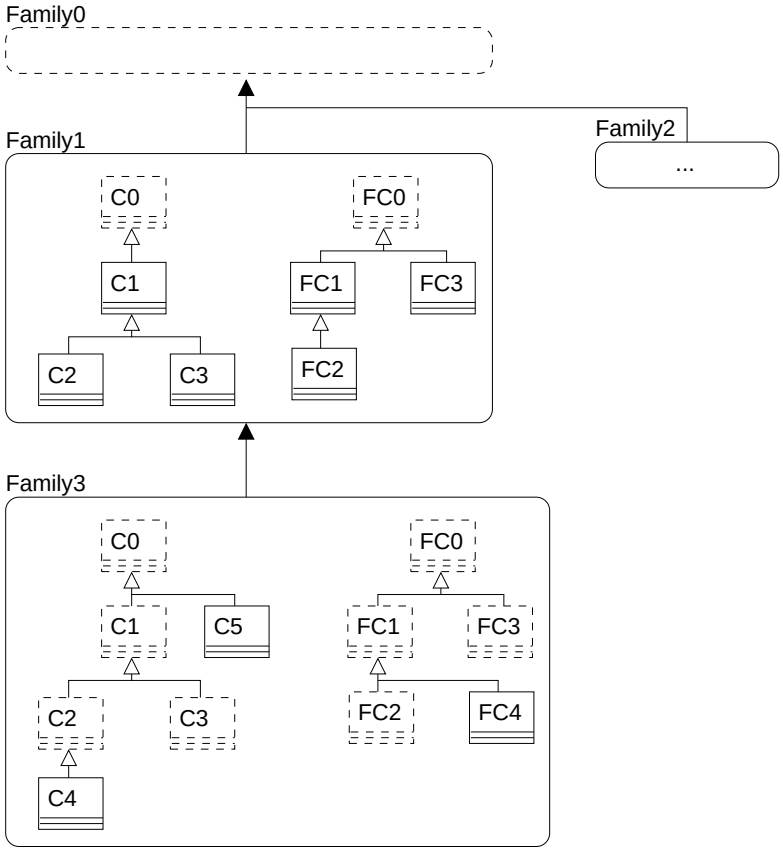
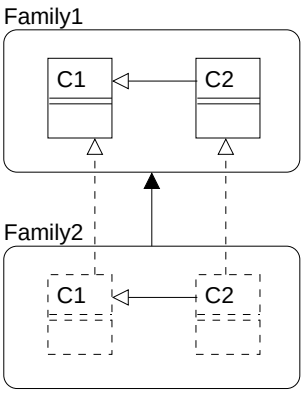


Abbildung 5.2.: Beispiel: Vererbungsgraphen von Familien, Daten- und Feldklassen




```
family Family1 {
  class c1 { }
  class c2 extends c1 { }
}
family Family2 extends Family1 {
}
```

Abbildung 5.3.: Materialisierungssemantik (1)

Abbildung 5.2 zeigt ein Beispiel. Dabei bedeuten durchgezogene Umrandungen eine explizite Familien- bzw. Klassendefinition und gestrichelte implizite, d. h. vererbte. `Family0`, `C0` bzw. `FC0` stellen die oben erwähnte triviale Familie und Daten- bzw. Feldklasse dar. Definiert sind die Familien `Family1` bis `Family3` und innerhalb `Family1` eine Datenklassenhierarchie und eine Feldklassenhierarchie; die Datenklassenhierarchie und die Feldklassenhierarchie werden in `Family3` erweitert. Die Mehrfachvererbung zeigt sich darin, dass beispielsweise die Klasse `Family3.C3` von den Klassen `Family3.C1` (innerfamiliär) und `Family1.C3` (interfamiliär) abgeleitet ist.

Im Folgenden werden die Details des Ansatzes anhand der etwas einfacheren Struktur aus Abbildung 5.3 (links in UML-Darstellung, rechts im W2L-Code) erläutert, wobei die gestrichelten senkrechten Pfeile die implizite Vererbung durch das Familienkonzept illustrieren. Explizit definiert sind in diesem Beispiel nur die Klassen `c1` und `c2` in der Familie `Family1`.

Wie in Abschnitt 4.2.2 erläutert, wird im W2OM eine Materialisierungssemantik eingesetzt, bei der abgeleitete Familien und Klassen zur Übersetzungszeit materialisiert werden, d. h. pro Familie und Klasse werden rekursiv alle vererbten Felder und Methoden aus der Superfamilie bzw. -klasse übernommen (im Beispiel in Abbildung 5.2 werden die Umrandungen der gestrichelten Klassen „durchgezogen“). Dabei wird entlang der Familienhierarchie, beginnend bei Fam^0, und innerhalb jeder Familie entlang der Klassenhierarchien, beginnend bei C_D^0 bzw. C_F^0, materialisiert (in den Beispielen also von oben nach unten und von links nach rechts). Auf diese Weise ist sichergestellt, dass alle Vorfahren (Superfamilie sowie Superklasse) bereits materialisiert sind. Im Beispiel aus Abbildung 5.3 wird also zunächst in `Family1` `C1` und dann `C2` materialisiert und dann in `Family2` `C1` und dann `C2`.

Eine Klasse C einer Familie Fam enthält die Vereinigungsmenge aller Felder und Methoden der Superfamilie Fam' sowie der Superklasse C'. Sofern die Herkunft eindeutig

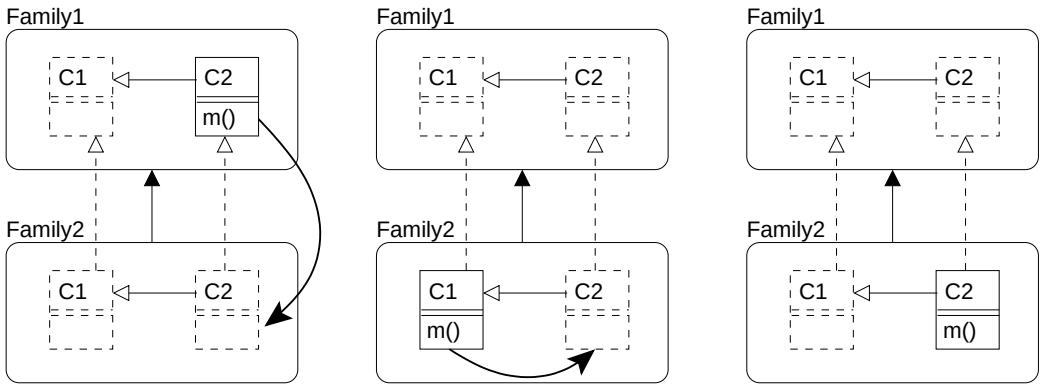


Abbildung 5.4.: Materialisierungssemantik (2)

ist, d. h. wenn ein Feld/eine Methode entweder nur in der Superfamilie (Abbildung 5.4a), nur in der Superklasse (Abbildung 5.4b) bzw. erstmals in der Klasse C (Abbildung 5.4c) definiert wird, wird das Feld/die Methode von dort übernommen.

Mehrfachvererbung und Priorität Wenn eine Methode auf zwei[5] verschiedenen Pfaden vererbt werden kann, so kann eine explizite Vererbungsrichtung priorisiert werden. Fehlt die Angabe einer Priorität, so wird die „nächstgelegene“ Definition materialisiert. Beispiel: in Abbildung 5.5a wird in `Family2.C1` die Methode `m(){1}` aus `Family1.C1` materialisiert – in `Family2.C2` wird jedoch die Methode `m(){2}` aus `Family1.C2` materialisiert, da die Distanz von `Family2.C2` zu der (expliziten) Definition von `m()` in `Family1.C2` kleiner ist.

Wenn eine Priorität angegeben ist, so wird:

- die Definition aus der Superfamilie gewählt, wenn die Methode die Priorität „family“ hat und
- die Definition aus der Superklasse gewählt, wenn die Methode die Priorität „class“ hat.

Ist die Priorität nicht explizit angegeben und keine eindeutig „nächstgelegene“ Definition vorhanden (z. B. `Family2.C2.m()` in Abbildung 5.5b), so ist dies ein Fehler. Ebenfalls ein Fehler ist es, wenn in der priorisierten Richtung keine Methodendefinition

[5] Wie weiter oben beschrieben, gibt es maximal zwei „Vorfahren“: die Klasse in der Superfamilie und die Superklasse, somit kann eine Methode auch nur auf höchstens zwei Pfaden im Vererbungsgraph vererbt werden.

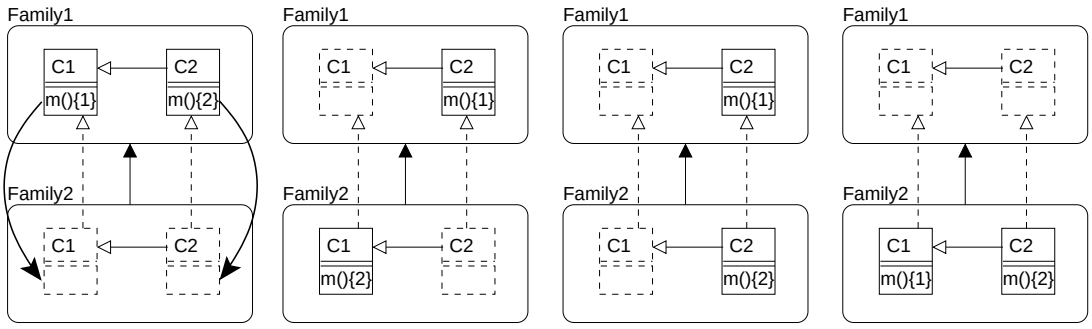


Abbildung 5.5.: Materialisierungssemantik (3)

zu finden ist (d. h. wenn beispielsweise in Abbildung 5.5a die Methode `m()` in `Family2.C1` die Priorität „class“ hätte).

Da der Ansatz Mehrdeutigkeiten lokal auflösen soll (d. h. an der Stelle, an der ein Konflikt entsteht), werden Prioritäten nicht vererbt, d. h. die Priorität muss an jeder möglichen Konfliktposition neu gesetzt werden.

Anonyme Feldmethoden Wie weiter oben definiert, können Feldklassen auch anonym definiert werden. Da diese Feldklassen nur innerhalb von Datenklassen auftreten können und keine weitere Vererbungssemantik beinhalten, ist die Semantik der Vererbung bei anonymen Feldmethoden analog zu der der Klassen zu sehen, d. h. auch bei anonymen Feldmethoden werden die jeweils gültigen Definitionen nach obigem Algorithmus ausgewählt.

Überschreiben von Methoden Wird eine in der Superfamilie (Abbildung 5.5c) oder Superklasse (Abbildung 5.5d) definierte Methode (mit demselben Deskriptor) überschrieben, so wird, wie beim Überschreiben üblich, nur die „neu“ definierte Methode übernommen, die vererbten Definitionen werden ignoriert. Ein Überladen von Methoden, d. h. Methoden mit gleichem Namen, aber unterschiedlichen Deskriptoren, wird nicht unterstützt.

Beispiel Materialisierungssemantik Abbildung 5.6 illustriert die Materialisierungssemantik an einem Beispiel. In der Ausgangssituation (5.6a), sind verschiedene explizit definierte Felder und Methoden zu sehen. In Schritt 1 (5.6b) wird `Family1.C2` materialisiert, indem die Felddefinition von `s` übernommen wird – `m()` bleibt dabei unverändert (überschreiben). In Schritt 2 (5.6c) werden in `Family2.C1` die Felddefinition von `s` sowie die Methodendefinition von `m()` aus `Family1.C1` übernommen. Im letzten Schritt

(5.6d) schließlich werden die Definitionen von Superklasse und Klasse in der Superfamilie nach `Family2.C2` materialisiert. Dabei wird (auf Grund der kürzeren Entfernung der expliziten Definition) für Methode `m()` der Methodenrumpf aus `Family1.C2` ausgewählt; der Rumpf aus der Superklasse hätte durch eine explizite Priorisierung via `Number m() priority class` in `Family2.C2` ausgewählt werden können.

Es wurde postuliert, dass die trivialen Klassen C_D^0 und C_F^0 keine Methoden- und Felddeklarationen enthalten. In der Praxis ist es hingegen durchaus sinnvoll, in dieser Klasse allgemeingültige Methoden[6] und Felder (z. B. Konstanten) zu definieren, die in allen Klassen der Familie zur Verfügung stehen sollen (ähnlich der `Object`-Klasse in Java). In abgeleiteten Familien können diese Klassen, *DataNode* bzw. *Field* genannt, wiederum um anwendungsspezifische Funktionalität erweitert werden. Die nachstehende Tabelle zeigt die verschiedenen Strukturen nochmals im Überblick.

	Zustand	Verhalten	Bemerkungen
Datenklassen (C_D)	Ja, in Feldern	Ja	komplexer Zustand (NF^2)
`DataNode`-Klasse (C_D^0)	Ja, in Feldern	Ja	Felder und Methoden, die in allen Datenklassen enthalten sein sollen.
Feldklassen (C_F)	Ja, in 1NF	Ja	
`Field`-Klasse (C_F^0)	Ja, in 1NF	Ja	Methoden, die in allen Feldklassen enthalten sein sollen.

5.2.2. Formale Semantik

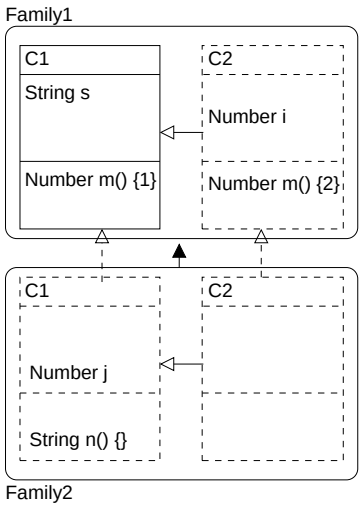
Nach der bisherigen informellen Spezifikation der Semantik folgt nun eine formale Definition. Wie auch in anderen objektorientierten Sprachen wird die Semantik anhand von Lookup-Funktionen spezifiziert ([CP89], [AC96]), mit denen die konkrete Definition der entsprechenden Familien, Klassen, Methoden und Felder gefunden werden kann[7].

Für die weitere Spezifikation sei $P = \overline{Fam}$, zudem werden die in Abbildung 5.7 gezeigten Notationen sowie Hilfsfunktionen benötigt. Es gilt: $Fam \lhd Fam'$, wenn Fam' Superfamilie von Fam ist; $Fam.C_D \lhd Fam.C_D{}'$, wenn in Familie Fam $C_D{}'$ Super(daten)klasse von C_D und $Fam.C_F \lhd Fam.C_F{}'$, wenn in Familie Fam $C_F{}'$ Super(feld)klasse von C_F ist. $\lhd$ ist reflexiv (C-Refl), transitiv (C-Trans) und antisymmetrisch.

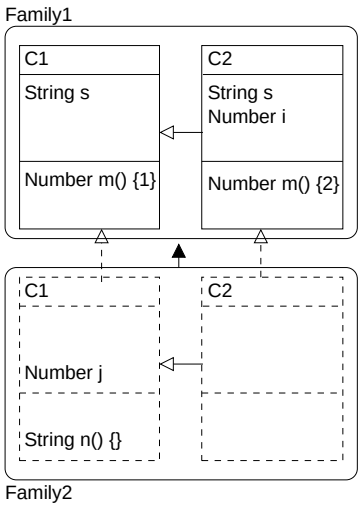
Die Funktion $Sig(M)$ liefert die Signatur einer Methode M, d. h. eine Liste der Typen der Parameter; die Kardinalität $|Sig(M)|$ gibt die Anzahl der Parameter an.

[6] Beispiele solcher Methoden sind z. B. `Boolean isdefined()` (`true`, wenn Feld definiert) oder `Boolean isempty()` (`true`, wenn definiert, aber leer).

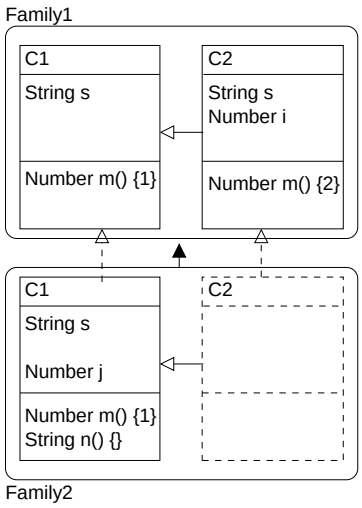
[7] Die verwendete Notation der Inferenzregeln wird u. a. in [Mit96] oder [Pie02] spezifiziert.



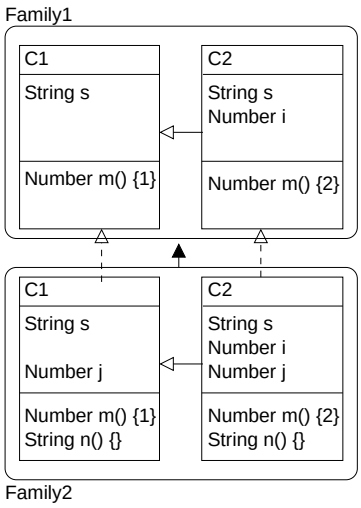
a) Ausgangssituation



b) Schritt 1



c) Schritt 2



d) Endsituation

Abbildung 5.6.: Materialisierungssemantik: Beispiel

$$\frac{\text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\ \} \in P}{Fam \lhd Fam'} \quad \text{(Fam-Sup)}$$

$$\frac{\begin{array}{c}\text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\ \} \in P \\ \text{class } C_D \text{ extends } C_D' \{\ \overline{Field}\ \overline{Meth_2}\ \} \in \overline{Class_D}\end{array}}{Fam.C_D \lhd Fam.C_D'} \quad (\text{C}_D\text{-Sup})$$

$$\frac{\begin{array}{c}\text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\ \} \in P \\ \text{field class } C_F \text{ extends } C_F' \{\overline{Meth_2}\} \in \overline{Class_F}\end{array}}{Fam.C_F \lhd Fam.C_F'} \quad (\text{C}_F\text{-Sup})$$

$$\frac{}{\overline{Fam.C \lhd Fam.C}} \quad \text{(C-Refl)}$$

$$\frac{Fam.C \lhd Fam.D \qquad Fam.D \lhd Fam.E}{Fam.C \lhd Fam.E} \quad \text{(C-Trans)}$$

$$\frac{\begin{array}{c}M \in \overline{Meth} \\ T\ M(T_1\ X_1,\ \ldots,\ T_n\ X_n), n \geq 0\end{array}}{Sig(M) = T_1\ , \ldots,\ T_n} \quad \text{(Sig-1)}$$

$$\frac{Sig(M) = T_1\ , \ldots,\ T_n}{|Sig(M)| = n} \quad \text{(Sig-2)}$$

Abbildung 5.7.: Notationen und Hilfsfunktionen: $\lhd$ und Sig

$$\frac{\text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\ \} \in P}{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\ \}} \quad \text{(Fam-1)}$$

$$\frac{\text{family } Fam \text{ extends } Fam' \{\ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\ \} \notin P}{\mathcal{F}am(Fam) = \bot} \quad \text{(Fam-2)}$$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam_1), \mathcal{F}am(Fam_2) \in P \\ \mathcal{F}am(Fam_1) \neq \mathcal{F}am(Fam_2)\end{array}}{Fam_1 \neq Fam_2} \quad \text{(Fam-unique)}$$

Abbildung 5.8.: Familien-Lookup

Abbildung 5.8 definiert die Familien-Lookup-Funktion $\mathcal{F}am$, welche die konkrete Definition einer Familie *Fam* ausgibt, sofern sie im Programm P definiert ist (Fam-1), und undefiniert($\perp$) sonst (Fam-2). Um Fehler zu vermeiden, wird festgelegt, dass Familiennamen in einem Programm eindeutig sein müssen, d. h. es darf nicht mehrere Familiendefinitionen mit demselben Namen geben, wie in (Fam-unique) formalisiert.

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \{ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1} \}\quad \text{class } C_D \text{ extends } C_D' \{ \overline{Field}\ \overline{Meth_2} \} \in \overline{Class_D}}{\mathcal{C}_{\mathcal{D}}(Fam, C_D) = \text{class } C_D \text{ extends } C_D' \{ \}} \quad (\text{C}_D\text{-1})$$

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \{ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1} \}\quad \text{class } C_D \text{ extends } C_D' \{ \overline{Field}\ \overline{Meth_2} \} \notin \overline{Class_D}}{\mathcal{C}_{\mathcal{D}}(Fam, C_D) = \mathcal{C}_{\mathcal{D}}(Fam', C_D)} \quad (\text{C}_D\text{-2})$$

$$\frac{}{\mathcal{C}_{\mathcal{D}}(Fam^0, C_D) = \perp} \quad (\text{C}_D\text{-3})$$

Abbildung 5.9.: Datenklassen-Lookup

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \{ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1} \}\quad \text{field class } C_F \text{ extends } C_F' \{ \overline{Meth_2} \} \in \overline{Class_F}}{\mathcal{C}_{\mathcal{F}}(Fam, C_F) = \text{field class } C_F \text{ extends } C_F' \{ \}} \quad (\text{C}_F\text{-1})$$

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \{ \overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1} \}\quad \text{field class } C_F \text{ extends } C_F' \{ \overline{Meth_2} \} \notin \overline{Class_F}}{\mathcal{C}_{\mathcal{F}}(Fam, C_F) = \mathcal{C}_{\mathcal{F}}(Fam', C_F)} \quad (\text{C}_F\text{-2})$$

$$\frac{}{\mathcal{C}_{\mathcal{F}}(Fam^0, C_F) = \perp} \quad (\text{C}_F\text{-3})$$

Abbildung 5.10.: Feldklassen-Lookup

In Abbildungen 5.9 und 5.10 sind die Lookup-Funktionen für Daten- und Feldklassen zu sehen. $\mathcal{C}_{\mathcal{D}}(Fam, C_D)$ liefert die Definition der Datenklasse C_D in der Familie *Fam* (analog $\mathcal{C}_{\mathcal{F}}(Fam, C_F)$ für Feldklassen). $\mathcal{C}_{\mathcal{D}}$ und $\mathcal{C}_{\mathcal{F}}$ sind also Abbildungen von Klassennamen auf deren Deklarationen. Ist die Klasse in der Familie *Fam* definiert, so wird diese direkt zurückgegeben (C_D-1 bzw. C_F-1), wenn nicht, wird in der Superfamilie gesucht (C_D-2 bzw. C_F-2), bis Fam^0 erreicht ist, in der per definitionem alle Klassen undefiniert

sind (C_D-3 bzw. C_F-3). Analog zu Familien müssen dabei die Klassennamen pro Familie eindeutig sein.

Des Weiteren darf der Vererbungsgraph bestehender Klassen innerhalb einer Familie in einer abgeleiteten Familie nicht verändert werden. Beispielsweise darf eine Deklaration `class` C `extends` C' in einer abgeleiteten Familie nicht von `class` C `extends` C'' (mit $C' \neq C''$) überschrieben werden. Darüber hinaus dürfen Familien-, Daten- und Feldklassenhierarchien nicht zyklisch sein, d. h. die jeweiligen Vererbungsgraphen dürfen keinen Zyklus enthalten.

Methoden werden ausschließlich über ihren Namen identifiziert, d. h. ein Überladen ist nicht möglich, sondern wird als Überschreiben interpretiert[8]. Existiert beispielsweise eine Methode $m(Number\ x)$ und in einer Subklasse eine Methode $m(String\ s)$, so ist in der Subklasse nur die Methode $m(String\ s)$ sichtbar.

Familienmethoden-Lookup Da bei der Familienvererbung nur Einfachvererbung möglich ist, verhält sich der Methodenlookup bei Familienmethoden analog zum Klassenlookup, wie in Abbildung 5.11 illustriert. (M-Family) definiert die Funktion $method(Fam, M)$, die die Methodendefinition liefert, sofern M in Familie Fam explizit definiert ist. Falls die gesuchte Methode dort nicht definiert ist, wird in der Superfamilie (M-Super) bis zur Family^0, in der per definitionem keine Methoden enthalten sind (M-Family^0-undef), gesucht.

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\} \quad \text{static? } Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \in \overline{Meth}}{method(Fam, M) = \text{static? } Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}} \text{ (M-Family)}$$

$$\frac{\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth}\} \quad \text{static? } Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \notin \overline{Meth}}{method(Fam, M) = method(Fam', M)} \text{ (M-Family-Super)}$$

$$\frac{}{method(Fam^0, M) = \bot} \text{ (M-Family}^0\text{-undef)}$$

Abbildung 5.11.: Familienmethoden-Lookup

[8] Diese Einschränkung dient dazu, die Semantik einfacher zu halten. In der Implementierung sollte ein Überladen aus Komfortgründen zugelassen werden. Eine mögliche Vorgehensweise ist dabei, intern die Signaturen mit in den Funktionsnamen zu integrieren. Beispielsweise können die zwei Methoden m(int) und m(long) intern durch zwei Methoden m-int(int) und m-long(long) „simuliert“ werden. Überladen ist im Zusammenhang mit higher-order Funktionen ebenfalls problematisch.

Klassenmethoden-Lookup Auf Grund der einfachen Form der Mehrfachvererbung von Klassen ist der Methodenlookup bei Klassen komplizierter als bei Familienmethoden, da Klassenmethoden sowohl aus der (gleichnamigen Klasse in der) Superfamilie als auch der Superklasse (in der gleichnamigen Familie) geerbt werden können. Dabei ist, ähnlich dem Klassenlookup, eine weitgehende Übereinstimmung bei Feld- und Datenklassen zu erkennen (abgesehen von der leicht unterschiedlichen Syntax).

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{class } C_D \text{ extends } C_D' \ \{\ \overline{Field}\ \overline{Meth_2}\ \} \in \overline{Class_D} \\ static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \in \overline{Meth_2} \\ M_{body} = \{\overrightarrow{Expr}\}\end{array}}{method_1(Fam, C_D, M) = static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}} \quad (\text{C}_D\text{-Method}_1)$$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{field class } C_F \text{ extends } C_F' \ \{\ \overline{Meth_2}\ \} \in \overline{Class_F} \\ static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \in \overline{Meth_2} \\ M_{body} = \{\overrightarrow{Expr}\}\end{array}}{method_1(Fam, C_D, M) = static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}} \quad (\text{C}_F\text{-Method}_1)$$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{class } C_D \text{ extends } C_D' \ \{\ \overline{Field}\ \overline{Meth_2}\ \} \in \overline{Class_D} \\ static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \in \overline{Meth_2} \\ M_{body} = \text{priority } Prio;\end{array}}{prio_1(Fam, C_D, M) = Prio} \quad (\text{C}_D\text{-Prio}_1)$$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{field class } C_F \text{ extends } C_F' \ \{\ \overline{Meth_2}\ \} \in \overline{Class_F} \\ static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body} \in \overline{Meth_2} \\ M_{body} = \text{priority } Prio;\end{array}}{prio_1(Fam, C_F, M) = Prio} \quad (\text{C}_F\text{-Prio}_1)$$

Abbildung 5.12.: Klassenmethoden-Lookup: Hilfsfunktionen

Abbildung 5.12 definiert zunächst einige Hilfsfunktionen. $method_1(Fam, C, M)$ mit $C \in \{C_D, C_F\}$ liefert die Definition einer Feld- oder Datenklassenmethode, sofern sie explizit (d. h. nicht vererbt) in der angegebenen Familie/Klasse definiert ist und undefiniert ($\perp$) sonst. Die Funktion $prio_1(Fam, C, M)$ liefert die Priorität einer Methode `Fam.C.M()`, falls (explizit) definiert, undefiniert sonst.

$$\frac{method_1(Fam, C, M) = static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}}{method(Fam, C, M) = static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}} \quad \text{(M-Class)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \\ Fam \lhd Fam' \\ s(Fam, C, M) = family \end{array}}{method(Fam, C, M) = method(Fam', C, M)} \quad \text{(M-Class-Superfamily)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \\ Fam.C \lhd Fam.C' \\ s(Fam, C, M) = class \end{array}}{method(Fam, C, M) = method(Fam, C', M)} \quad \text{(M-Class-Superclass)}$$

$$\frac{}{method(Fam^0, C, M) = \bot} \quad \text{(M-Class-Family0)}$$

$$\frac{}{method(Fam, C^0, M) = \bot} \quad \text{(M-Class-Class0)}$$

Abbildung 5.13.: Daten- und Feldklassenmethoden-Lookup ($C \in \{C_D, C_F\}$)

Die Funktionen für das Klassenmethoden-Lookup sind in Abbildung 5.13 zu sehen. Dabei liefert $method(Fam, C, M)$:

- die konkrete Definition von M, wenn M in $Fam.C$ definiert ist (M-Class).
- die Definition aus der Superfamilie, wenn die Selektionsfunktion s (s. u.) „family“ liefert (M-Class-Superfamily).
- die Definition aus der Superklasse, wenn die Selektionsfunktion s (s. u.) „class“ liefert (M-Class-Superclass).
- undefiniert($\bot$) bei Fam^0 bzw. C^0 (M-Class-Family0 bzw.M-Class-Class0).

Dabei setzt die Selektionsfunktion s die auf Seite 82 beschriebene Auflösung von Mehrdeutigkeiten um, wie in Abbildung 5.14 zu sehen. Sofern explizit Prioritäten gesetzt sind, liefern (s-1) und (s-2) dementsprechend `family` bzw. `class` zurück. Wenn die Priorität nicht gesetzt ist und die Methodendefinition in der Superfamilie (bzw. -klasse) „näher“ ist, so wird `family` (bzw. `class`) zurückgegeben. Ist die Distanz gleich (s-5), so ist dies ein Fehler.

Weiterhin ist in Abbildung 5.14 die Distanzfunktion zu sehen: Wenn eine Methode explizit definiert ist, so ist die Distanz $dist(Fam, C, M)$ gleich 0 (Dist-1). Ansonsten ist die Distanz das Minimum der Distanzen der Methode in der Superfamilie und in der Superklasse (Dist-2).

$$\frac{prio_1(Fam, C, M) = class}{s(Fam, C, M) = \text{class}} \quad \text{(s-1)}$$

$$\frac{prio_1(Fam, C, M) = family}{s(Fam, C, M) = \text{family}} \quad \text{(s-2)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \qquad prio_1(Fam, C, M) = \bot \\ Fam \lhd Fam' \qquad Fam.C \lhd Fam.C' \\ dist(Fam', C, M)) < dist(Fam, C', M) \end{array}}{s(Fam, C, M) = \text{family}} \quad \text{(s-3)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \qquad prio_1(Fam, C, M) = \bot \\ Fam \lhd Fam' \qquad Fam.C \lhd Fam.C' \\ dist(Fam', C, M)) > dist(Fam, C', M) \end{array}}{s(Fam, C, M) = \text{class}} \quad \text{(s-4)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \qquad prio_1(Fam, C, M) = \bot \\ Fam \lhd Fam' \qquad Fam.C \lhd Fam.C' \\ dist(Fam', C, M)) = dist(Fam, C', M) \end{array}}{s(Fam, C, M) = Error} \quad \text{(s-5)}$$

$$\frac{method_1(Fam, C, M) = static?\ Type\ M(\overrightarrow{Type}\ \vec{X})\ M_{body}}{dist(Fam, C, M) = 0} \quad \text{(Dist-1)}$$

$$\frac{\begin{array}{c} method_1(Fam, C, M) = \bot \\ Fam \lhd Fam' \qquad Fam.C \lhd Fam.C' \end{array}}{dist(Fam, C, M) = min(dist(Fam', C, M), dist(Fam, C', M)) + 1} \quad \text{(Dist-2)}$$

Abbildung 5.14.: Selektionsfunktion s und Distanzfunktion $dist$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{class } C_D \text{ extends } C_D' \ \{\ \overline{Field}\ \overline{Meth_2}\ \} \in \overline{Class_D} \\ Type\ F\ (\{\ \overline{Meth_3}\ \})? \in \overline{Field}\end{array}}{field_1(Fam, C_D, F) = Type\ F\ (\{\ \overline{Meth_3}\ \})?} \quad (\text{Field}_1\text{-1})$$

$$\frac{\begin{array}{c}\mathcal{F}am(Fam) = \text{family } Fam \text{ extends } Fam' \ \{\overline{Class_F}\ \overline{Class_D}\ \overline{Meth_1}\} \\ \text{class } C_D \text{ extends } C_D' \ \{\ \overline{Field}\ \overline{Meth_2}\ \} \in \overline{Class_D} \\ Type\ F\ (\{\ \overline{Meth_3}\ \})? \notin \overline{Field}\end{array}}{field_1(Fam, C_D, F) = \bot} \quad (\text{Field}_1\text{-2})$$

Abbildung 5.15.: Feld-Lookup (1)

Abbildung 5.15 zeigt die Feld-Lookup-Funktion $field_1(Fam, C_D, F)$, die analog zu der $method_1$-Funktion bei Methoden die konkrete Definition eines Feldes liefert, sofern dieses explizit in $Fam.C_D$ definiert ist, und undefiniert($\bot$) sonst.

Die Definition der Funktion $field(Fam, C_D, F)$ ist in Abbildung 5.16 gezeigt. Es gilt: Wenn das Feld explizit in $Fam.C_D$ definiert ist, wird dieses zurückgegeben (Field-1). Sofern es nur interfamiliär oder nur interfamiliär vererbt wird, wird die jeweilige Definition zurückgegeben (Field-2 bzw. Field-3). Ist das Feld in beiden Klassen definiert und die Definitionen (d. h. Typ und Methodendefinitionen) identisch, so wird diese zurückgegeben (Field-4), unterscheiden sie sich, so ist dies ein Fehler (Field-5). (Field-6) bis (Field-8) schließlich liefern undefiniert zurück, wenn das Feld weder in Superfamilie noch -klasse definiert ist bzw. für Fam^0 und ${C_D}^0$.

$$\frac{field_1(Fam, C_D, F) = Type\ F\ (\{\ \overline{Meth}\ \})?}{field(Fam, C_D, F) = Type\ F\ (\{\ \overline{Meth}\ \})?} \quad \text{(Field-1)}$$

$$\frac{\begin{array}{c} field_1(Fam, C_D, F) = \bot \\ Fam \lhd Fam' \qquad Fam.C_D \lhd Fam.C_D{}' \\ field(Fam', C_D, F) = Type\ F\ (\{\ \overline{Meth}\ \})? \\ field(Fam, C_D{}', F) = \bot \end{array}}{field(Fam, C_D, F) = field(Fam', C_D, F)} \quad \text{(Field-2)}$$

$$\frac{\begin{array}{c} field_1(Fam, C_D, F) = \bot \\ Fam \lhd Fam' \qquad Fam.C_D \lhd Fam.C_D{}' \\ field(Fam', C_D, F) = \bot \\ field(Fam, C_D{}', F) = Type\ F\ (\{\ \overline{Meth}\ \})? \end{array}}{field(Fam, C_D, F) = field(Fam, C_D{}', F)} \quad \text{(Field-3)}$$

$$\frac{\begin{array}{c} field_1(Fam, C_D, F) = \bot \\ Fam \lhd Fam' \qquad Fam.C_D \lhd Fam.C_D{}' \\ field(Fam', C_D, F) = Type_1\ F\ (\{\ \overline{Meth_1}\ \})? \\ field(Fam, C_D{}', F) = Type_2\ F\ (\{\ \overline{Meth_2}\ \})? \\ Type_1 = Type_2 \qquad \overline{Meth_1} = \overline{Meth_2} \end{array}}{field(Fam, C_D, F) = field(Fam', C_D, F)} \quad \text{(Field-4)}$$

$$\frac{\begin{array}{c} field_1(Fam, C_D, F) = \bot \\ Fam \lhd Fam' \qquad Fam.C_D \lhd Fam.C_D{}' \\ field(Fam', C_D, F) = Type_1\ F\ (\{\ \overline{Meth_1}\ \})? \\ field(Fam, C_D{}', F) = Type_2\ F\ (\{\ \overline{Meth_2}\ \})? \\ Type_1 \neq Type_2 \quad \vee \quad \overline{Meth_1} \neq \overline{Meth_2} \end{array}}{field(Fam, C_D, F) = Error} \quad \text{(Field-5)}$$

$$\frac{\begin{array}{c} field_1(Fam, C_D, F) = \bot \\ Fam \lhd Fam' \qquad Fam.C_D \lhd Fam.C_D{}' \\ field(Fam', C_D, F) = \bot \\ field(Fam, C_D{}', F) = \bot \end{array}}{field(Fam, C_D, F) = \bot} \quad \text{(Field-6)}$$

$$\frac{}{field(Fam^0, C_D, F) = \bot} \quad \text{(Field-7)}$$

$$\frac{}{field(Fam, C_D{}^0, F) = \bot} \quad \text{(Field-8)}$$

Abbildung 5.16.: Feld-Lookup (2)

6. Spezifikation der Sprache W2L

Dieses Kapitel beschäftigt sich mit Ausdrücken, Methoden und dem Typsystem der W2L-Sprache. Dabei stehen insbesondere die Aspekte des Modells im Vordergrund, die (in dieser Form) nicht in bisherigen Ansätzen bzw. Java vorhanden sind.[1] Anhang A illustriert verschiedene Aspekte von Modell und Sprache in einem „real world"-Beispiel.

6.1. Klassen und Objekte

Wie bereits in Abschnitt 5.2.1 erwähnt, können Objekte prinzipiell auf zwei verschiedene Arten instanziiert werden: Entweder als persistente Datenbankobjekte (s. Abschnitt 8.3) oder als nicht persistente Hilfsobjekte. Die Instanziierung erfolgt über (statische) Konstruktoren, ähnlich wie in anderen objektorientierten Sprachen. Jede Klasse stellt hierbei einen Default-Konstruktor zur Verfügung, der denselben Namen wie die Klasse hat und überschrieben werden kann.

Der Default-Konstruktor von *Feldklassen* erwartet als Argument einen Wert des Typs, den die Klasse erweitert. Da der Zustand von Feldobjekten in 1NF ist, ist die Default-Konstruktormethode immer genau einstellig; andere Konstruktoren können andere Stelligkeiten haben, wie im folgenden Beispiel zu sehen:

```
field class Euros extends Number {
 static Euros Euros(Number); // Default-Konstruktor
 static Euros eurosFromEurosCents(Number euros,Number cents) {
   Euros(euros + cents/100);
 }
 static Euros eurosFromCents(Number cents) {
   Euros(cents/100);
 }
}

class C {
  // Instanziierung
  Euros subsum=Euros(2, 30);
}
```

[1] Anmerkungen zur Notation in diesem Kapitel: `A` und `B` (bzw. `<A>` und `<B>`) bezeichnen Typvariablen, `C` und `D` (Daten)klassen, `f` und `g` Funktionen.

Die Klasse `Euros` definiert eine Feldklasse zur Arbeit mit Euro-Beträgen, die von `Number` abgeleitet ist. Der Default-Konstruktor erwartet einen Euro-Betrag als `Number` und gibt ein Euro-Objekt mit diesem Betrag zurück. Die beiden Konstruktoren `eurosFromEurosCents` und `eurosFromCents` sind selbstgeschriebene Konstruktoren, die ein Euro-Objekt aus unterschiedlichen Eingaben erzeugen. Abschließend ist zu sehen, wie ein solches Euro-Objekt instanziiert werden kann.

Die Stelligkeit der Default-Konstruktoren bei Datenklassen entspricht der Anzahl der Felder (da der Zustand von Datenobjekten nicht in 1NF sein muss). Dabei gibt die Reihenfolge der Felder die Reihenfolge der Argumente vor, wie im folgenden Beispiel illustriert. Ebenfalls zu sehen ist die Instanziierung eines solchen Hilfsdatenobjekts.

```
class Text {
   String title;
   XML body;

   static Text(String title, XML body); // Default-Konstruktor
}

class C {
   // Instanziierung eines Hilfsdatenobjektes
   Text helpText=Text("Mein Titel",<p>body</p>);
}
```

Ähnlich wie in anderen Sprachen gelten auch in der W2L Namenskonventionen. So dürfen Familien-, Klassen-, Methoden- und Feldnamen alphanumerische Zeichen in Groß- und Kleinschrift sowie _ enthalten. Dabei sollen Familien- und Klassennamen groß und Instanz- sowie Methodennamen klein geschrieben werden.

6.2. Typsystem

Aus den in Kapitel 4 geschilderten Gründen ist das Typsystem weitestgehend statisch modelliert, d. h. die Typ-Konformität wird, soweit möglich, zur Übersetzungszeit geprüft. Abschnitt 6.2.3 beschäftigt sich mit Schwierigkeiten hinsichtlich dynamischer Typen.

6.2.1. Datentypen

Ähnlich wie in anderen objektorientierten Programmiersprachen definieren (Daten- und Feld-) Klassen in W2L neue Datentypen, Subklassen sind auch Subtypen. Prinzipiell sind alle Datentypen Feld- oder Datenklassen, insbesondere sind auch die nachfolgend aufgeführten Basisdatentypen spezielle vordefinierte Feldklassen. Jedes Feldobjekt kann

entweder als Objekt (zur Darstellung) oder als einfacher Datentyp (zum „Rechnen“) angesprochen werden; somit entfällt der Umstand, die Typen umzuwandeln (sog. „Autoboxing“, u. a. in Java seit Version 5 vorhanden). Die W2L stellt folgende Basisdatentypen und Konstanten zur Verfügung:

- `Boolean`: Wahrheitswert (`true` oder `false`)
- `String`: Zeichenkette, umschlossen von `"`.
  Beispiel: `"Test"`
- `Number`: Zahlen, sowohl ganze als auch Fließkommazahlen[2].
  Beispiele: `-3`, `42`, `3.141`, `-0.007`.
- `Date`: Datum im ISO 8601-Format `jjjjmmtt`
- `Time`: Zeit im ISO 8601-Format `hh:mm:ss`

Eine Sonderstellung nimmt in der W2L der Datentyp `XML` für wohlgeformte XML-Fragmente (s. Definition auf Seite 12) ein. Felder dieses Typs können beliebige XML-Fragmente aufnehmen, welche statisch überprüft werden, so dass eine Wohlgeformtheit bereits zur Übersetzungszeit garantiert werden kann. Nachfolgend einige Beispiele:

```
1  XML test1=<a/>;
2  XML test2=<a>test</a>;
3  XML test3=<a>noch ein <b>test</b></a>;
4  XML test4=<a>test</a><b>test</b>;
5
6  XML test5=<a>statischer Typfehler</b>;
7
8  String test6 = "foo";
9  XML test7 = "foo";
10
11 XML test8="<a>test</b>"; // wird escaped zu:
12 XML test8="&lt;a&gt;test&lt;/b&gt;";
```

Zeilen 1 bis 4 definieren „normale“ wohlgeformte XML-Daten, wobei `test4` zwar ein XML-*Fragment*, aber kein XML-*Dokument* darstellt (da mehr als ein Wurzelelement existiert). `test5` würde vom Compiler zur Übersetzungszeit als Typfehler erkannt werden, da Start-Tag und End-Tag nicht übereinstimmen. Während `test6` vom Typ `String` ist, ist `test7` ein XML-Textknoten. Beachte: `test8` wird (bedingt durch die Anführungszeichen) ein String und damit ein Textknoten zugewiesen, so dass der Compiler dies zu der Zeichenkette in Zeile 12 „escaped“.

[2] Realisiert als double oder infinite precision Zahl.

Listen enthalten Werte desselben Typs[3]. Die einzelnen Listenelemente werden über Ausdrücke spezifiziert; eine Liste definiert wiederum selbst einen neuen Datentyp. Die Syntax der Deklaration von Listen lautet ($n \geq 0$):

```
list(<Listentyp>) <Listenname> = [ <Expr1>, ..., <Exprn> ];
```

Der Zugriff auf Listenelemente erfolgt über die Syntax `<Listenname>[<Index>]` mit `<Index>` ≥ 0. Beispiele:

```
list(XML) xmllist = [ <a/>, test, <b>...</b>, gettitle() ]

list(list(Number)) listlist = [[3+4], [], [-3, 99]];
```

Dabei ist `xmllist[0] = <a/>`, `listlist[2] = [-3, 99]` und `listlist[0][0] = 7`.

Folgende Builtin-Funktionen können auf Listen angewandt werden:

- `<A> nth(Number n, list(<A>) l)`: liefert das n-te Element der Liste `l` zurück.
- `Number length(list(<A>) l)`: liefert die Länge der Liste `l` zurück.
- `nil`: liefert eine leere Liste zurück.
- `list(<A>) cons(<A> head, list(<A>) tail)`: liefert ein neues Listenobjekt mit Kopfelement `head` und Restliste `tail` zurück.

6.2.2. Statische Variablenbindungen

Die W2L verwendet den in der funktionalen Programmierung üblichen Mechanismus der Variablenbindung, d. h. Variablen werden bei der Deklaration einmalig an Werte gebunden und können nicht mehr verändert werden (Analog zur Mathematik: „Sei $y = f(x)$“ oder „Sei $z = 3$“). Zuweisungen im Sinne von imperativen Sprachen existieren demnach nicht. Syntaktisch wurde allerdings die Form der Initializer in Java gewählt, d. h. `type var = expr;` – alle drei Angaben sind erforderlich. Beispiele: `Number x=3; XML y=<a/>; String z=maketitle("test");`

Derartig statisch in Klassen gebundene Variablen werden ebenfalls entlang der Familien- und/oder Klassenhierarchie vererbt; dabei wird derselbe Priorisierungsalgorithmus wie bei Methoden (s. Abschnitt 5.2.1) angewendet, Syntax-Beispiel:

```
class C { Number pi = 3.1415 priority family; }
```

[3] Abschnitt 6.6 spezifiziert eine darüber hinausgehende Erweiterung dieses Ansatzes.

6.2.3. Dynamische Aspekte des Typsystems und Casting

Wie bereits erläutert, ist das Typsystem weitgehend statisch, jedoch sind einige Aspekte nur dynamisch typisierbar. Ein Beispiel ist eine Schleife über alle Kindobjekte eines Objekts, die ja Instanzen unterschiedlicher (Daten-)Klassen sein können und deren Typ statisch nicht bestimmt werden kann. Aus eben diesen Gründen existiert auch die Möglichkeit, Objekte dynamisch zu konvertieren (Casting – s. Abschnitt 2.3.5). Die Syntax ist hierbei dieselbe wie in Java. Ohne einen Cast können nur Methoden der `DataNode`-Klasse verwendet werden. Beispiel:

`((Text) child).showheadline();` – Cast der Instanz `child` nach `Text`.

Wie in Java kann mit dem Operator `instanceof` die Klasse eines Objekts überprüft werden, dabei liefert

```
Boolean obj instanceof Class
```

`true`, wenn `obj` eine Instanz der Klasse `Class` ist und `false` sonst.

Beispiel:

```
Text t = new Text();
if(t instanceof Text) {
  // a
} else {
  // b
}
```

Da Objekt `t` eine Instanz der Klasse `Text` ist, wird Teil a des `if`-Konstrukts ausgewertet.

6.3. Methoden

Wie bereits erläutert, sind Methoden in W2L immer Funktionen. Darüber hinaus sind Funktionen auch funktionale Objekte, d. h. eine Methode `f` kann entweder über `f(x)` oder aber über `f.apply(x)` aufgerufen werden. Wie in Abschnitt 5.1 spezifiziert, können Methoden statisch oder dynamisch sein. Die Syntax von Methoden lautet (vgl. Abbildung 5.1 auf Seite 77):

Methode	*Meth*	::=	static? *Type* $M(\overrightarrow{Type}\ \vec{X})\ M_{body}$

Methoden sind n-stellige Funktionen ($n \geq 0$) mit einem Rückgabewert. Der Methodenrumpf besteht aus einer Liste von Ausdrücken (bzw. einer Prioritätsangabe, s. Kapitel 5). Da (dynamische) Methoden immer an ein Objekt gebunden sind (Auswertungskontext), kann man sie auch als n+1-stellige Funktion mit dem zugehörigen Objekt als Parameter auffassen.

Eine Besonderheit gegenüber Java ist, dass keine expliziten `return`-Anweisungen nötig sind; zurückgegeben wird immer der Wert der Berechnung. Existieren mehrere Ausdrücke, so werden diese im Falle von Strings und XML konkateniert zurückgegeben, bei anderen Datentypen ist dies ein Fehler.

Bedingt durch das funktionale Paradigma existieren keine Ausgabefunktionen (wie `print()`-Anweisungen), da diese einen Seiteneffekt bedeuten würden. Stattdessen wird der Rückgabewert der zuerst aufgerufenen Methode an die Ausgabekomponente (s. Abschnitt 8.4) übergeben und von dieser ausgegeben.

Methoden ohne Rückgabetyp, wie z. B. in Java durch `void` deklariert, machen in W2L keinen Sinn, weil in Methoden keine (direkte) Ausgabe erfolgen kann und Methoden den Zustand von Objekten nicht verändern können. Daher muss in W2L immer ein Rückgabetyp angegeben werden. Wie in Java bezeichnet das Schlüsselwort `this` das aktuelle Objekt. Beispiel:

```
1 Number fac1(Number n) {
2   if (n <= 1) {
3     1;
4   } else {
5     Number k = n - 1;
6     n * fac1(k);
7   }
8 }
```

Hier wird der Wert des `if`-Ausdrucks, also entweder in Zeile 3 der Wert `1` oder in Zeile 6 das Ergebnis von `n * fac1(k)` zurückgegeben.

6.3.1. Builtin-Funktionen und Bibliotheken

Auf Datenobjekten können folgende Builtin-Funktionen aufgerufen werden, um Objekte aus dem Instanzenbaum zu erhalten (in Anlehnung an verschiedene DOM-Funktionen):

- `list(DataNode) getChildren()`: liefert eine Liste aller Kindobjekte.
- `DataNode getParent()`: liefert das Vaterobjekt (falls vorhanden).
- `DataNode getFirstChild()`: liefert das erste Kindobjekt (falls vorhanden).
- `DataNode getNextSibling()`: liefert das nächste Geschwisterobjekt (falls vorhanden).
- `DataNode getPreviousSibling(DataNode node)`: liefert das vorhergehende Geschwisterobjekt (falls vorhanden).
- `DataNode getObject(Number ID)`: liefert das Objekt mit der ID `ID`

- `list(Field) getFields()`: liefert eine Liste aller Feldobjekte.

Darüber hinaus stehen in der konkreten Implementierung verschiedene Bibliotheken bereit, die die Arbeit mit dem System komfortabler machen; als Beispiel seien hier String- und Datumverarbeitungsfunktionen sowie arithmetische Funktionen und Sortierfunktionen auf Listen genannt.

6.3.2. Higher-order Funktionen

In einem ersten Ansatz sind die folgenden higher-order Funktionen in der W2L vordefiniert, eine in Abschnitt 6.6.3 vorgestellte Erweiterung ermöglicht es, eigene higher-order Funktionen zu definieren.

- `map :: (A -> B) -> [A] -> [B]`:
  wendet eine Funktion auf jedes Element einer Liste an.
  Syntax:

```
<B> f(<A> arg) = ...;

list(<B>) map(f, list(<A>));
```

  Beispiel:

```
1 Number f(Number i) {   // Funktion: f(i) := i+1
2   i+1;
3 }
4
5 list(Number) test = [1,2,3,4,5];    // erzeugen einer Liste
6
7 list(Number) test2 = map(f, test);  // f auf Liste anwenden
8
9 // Ergebnis: test2=[2,3,4,5,6]
```

  In diesem Beispiel wird zunächst eine Funktion $f(i)$ definiert, die $i+1$ zurückgibt. In Zeile 5 wird eine Liste von ganzen Zahlen erzeugt und in Zeile 7 wird die Funktion f auf jedes Element der Liste angewendet.

- `filter :: (A -> Boolean) -> [A] -> [A]`:
  selektiert die Elemente aus einer Liste, die einen gegebenen Ausdruck erfüllen.
  Syntax:

```
Boolean f(<A> arg) = ...;

list(<A>) filter(f, list(<A>));
```

  Beispiel:

```
Boolean f(Number i) {
  i>3;
}

list(Number) test = [1,2,3,4,5];    // erzeugen einer Liste

list(Number) test2 = filter(f, test);  // f auf Liste anwenden

// Ergebnis: test2=[4,5]
```

- `foldl :: (B -> A -> B) -> B -> [A] -> B` und
  `foldr :: (A -> B -> B) -> B -> [A] -> B`:
  Faltung – wendet eine Funktion `f` links- bzw. rechtsassoziativ auf aufeinanderfolgende Listenelemente an, beginnend mit einem Startelement.
  Syntax:

```
<B> f(<B> arg1, <A> arg2) = ...;

<B> foldl(f, <B> start, list(<A>));
```

  Dabei gilt:

```
foldl(f, x, []) = x

foldl(f, x, [l1, ..., ln]) = foldl(f, f(x,l1), [l2, ..., ln])
```

6.4. Kontrollstrukturen

Die W2L stellt die Kontrollstrukturen `if`, `switch` und `for` zur Verfügung, `while`-Schleifen können über Rekursion simuliert werden. Obwohl diese auf den ersten Blick imperativen (Anweisungs-)Charakter zu haben scheinen, können (und werden) sie jedoch als rein funktionale Zuordnungsvorschrift gesehen, wie im Folgenden gezeigt .

`if` stellt eine bedingte Auswertung dar und ist eine Funktion $if(Boolean, A, A) \rightarrow A$.
Die konkrete W2L-Syntax lautet:

```
if (<expression1>) {
  <expression2>
} else {
  <expression3>
}
```

Die Semantik ist dabei: Wenn `expression1` wahr ist, wird `expression2` ausgewertet, ansonsten `expression3`.

Beispiel:

```
if (field.isdefined()) {
  field.view()
} else {
  "undefiniert"
}
```

Wenn das Feld `field` definiert ist, wird dessen `view()`-Methode ausgewertet, ansonsten wird der Text `'undefiniert'` zurückgegeben.

Mit `switch` können fallweise Auswertungen realisiert werden; `switch` ist als Funktion $switch_n(Number, \underbrace{Number, A, ..., Number, A}_{n-mal}, A) \rightarrow A$ definiert.

Die konkrete W2L-Syntax lautet:

```
switch (<expression1>) {
  case <expression2a>: <expression2b>
  case <expression3a>: <expression3b>
  ...
  default: <expressionn>
}
```

Anmerkung: Im Unterschied zu Java existiert kein explizites `break`, d. h. in W2L wird immer genau ein Fall ausgewertet. Darüber hinaus ist die Angabe des `default`-Ausdrucks obligatorisch.

`for` iteriert über eine Liste und konkateniert die Ergebnisse (die Verwendung macht demnach nur bei den Rückgabetypen `string` und `xml` Sinn). Der Typausdruck ist $for(list(A)) \rightarrow String, XML$

Die konkrete W2L-Syntax lautet:

```
for (<var> : <list>) { }
```

`var` ist dabei die Schleifenvariable.

Beispiel:

```
for (i : [1,2,3]) {
  i+1
}
```

Ausgabe: 234

6.5. Verwendung in Templates

Die W2L kann auch im Sinne einer Templatesprache (vgl. Abschnitt 3.1.1) verwendet werden, wobei die geschweiften Klammern `{}` als Quote/Antiquote-Zeichen dienen. Beispiel:

```
1  ...
2  <head>{title}</head>
3
4  <body>{if (body.isdefined()) {
5      body.view()
6    } else {
7      "kein Body vorhanden!"
8    }}
9  </body>
10 ...
```

In Zeile 2 wird im `<head>`-Element der Inhalt des `title`-Felds ausgegeben. In den Zeilen 4 bis 8 steht ein längeres `if`-Konstrukt innerhalb des `body`-Elements.

6.6. Mögliche Erweiterungen des Klassen- und Typsystems

6.6.1. Parametrische Polymorphie und Typgenerizität

Ähnlich wie Generics in Java 1.5 oder Templates in C++ könnten in einer Erweiterung der W2L generische Klassen definiert werden, bei der, in Anlehnung an Java, die Typvariablen in spitzen Klammern geschrieben werden. Mithilfe dieser generischen Klassen können die Feldklassen/typen elegant definiert werden, wie das folgende Beispiel zeigt.

Beispiel:

```
1  field class Field<A> {
2     A value;
3     A get() { value; }
4  }
5
6  field class Number extends Field<Number> { ... }
```

Eine Feldklasse wird als generische Klasse mit formalem Typparameter A deklariert (Zeilen 1-4); eine konkrete Feldklasse wird von dieser Klasse „instanziiert“ (Zeile 6).

6.6.2. Algebraische Klassen

Bislang wurde unterschieden zwischen Feldklassen mit Zustand in erster Normalform, mit denen auch direkt arithmetische Operationen ausgeführt werden können, und Datenklassen, die komplexen Zustand haben können, mit denen aber nicht „gerechnet“ werden kann. Eine Erweiterung dieses Modells sind so genannte *algebraische* Klassen, die komplexen Zustand haben können und die algebraische Konstruktoren haben. (Sinnvolle) Beispiele für deren Verwendung sind ein Datentyp `Pair` oder Hashmaps; zudem können Listen eleganter als in Abschnitt 6.2.1 beschrieben implementiert werden.

6.6.3. Definition von higher-order Funktionen

Wie weiter oben bereits erwähnt, ist es in einer Erweiterung des Modells möglich, eigene higher-order Funktionen zu definieren. Die Syntax lehnt an Higher-Order Java[4] an. Beispiel:

```
<A,B> List<B> map(Fun<A,B> f, List<A> xs) {
      List<B>.cons(f.apply(xs.head()), map(f, xs.tail()))
}
```

Die higher-order Funktion `map` mit Typvariablen `A` und `B` erwartet als erstes Argument eine Funktion (genauer: ein funktionales Objekt) `f :: A -> B` und eine Liste `xs` vom Typ `A` und liefert eine Liste vom Typ `B` zurück. Im Methodenrumpf wird rekursiv die Ergebnisliste zusammengesetzt, indem `f` zunächst auf das Kopfelement von `xs` und map rekursiv auf die Restliste angewendet wird und beide Ergebnisse mit dem `Cons`-Konstruktor zu einer neuen Liste (vom Typ `B`) zusammengesetzt werden.

[4] `http://www.cs.chalmers.se/~bringert/hoj/`

7. Diskussion verwandter Arbeiten und Technologien

In diesem Kapitel werden Arbeiten, die Berührungspunkte bzw. Überlappungen mit den Themengebieten der vorliegenden Arbeit haben, vorgestellt und miteinander verglichen. Wie bereits in Kapitel 1 ausgeführt, ist die vorliegende Arbeit in der Schnittmenge mehrerer Forschungsgebiete angesiedelt, so dass bei den verwandten Arbeiten verschiedene Sichtweisen auf bzw. Herangehensweisen an die Problematik denkbar sind: Zunächst werden Datenbanksysteme mit eventuellen Erweiterungen (Abschnitt 7.1) sowie XML-basierten Ansätzen (Abschnitt 7.2) untersucht. Daran schließt eine Betrachtung von Programmiersprachen im Kontext semistrukturierter Daten (Abschnitt 7.3) sowie von erweiterten Vererbungskonzepten (Abschnitt 7.4) an. Die jeweiligen Ansätze werden dabei in Hinblick auf die in Kapitel 4 erarbeiteten Kernkonzepte eines geeigneten Datenmodells (wie Objektorientierung, funktionale Methoden oder Vererbungskonzepte) untersucht und diskutiert. Wie bereits in Kapitel 4 erläutert, sind einige dieser existierenden Konzepte theoretisch entsprechend erweiterbar, um die meisten Anforderungen bzw. Wünsche an ein Modell/System zu erfüllen, jedoch sind diese dann entweder umständlich in der Anwendung oder das eigentliche System ist als solches nicht mehr zu erkennen.

7.1. Datenbanken und Erweiterungen

Datenbanken sind als Backend, d. h. zur persistenten Speicherung der Daten eines CMS, gut geeignet (wobei hier in der Regel ein geeignetes Mapping des CMS-Datenmodells auf das Datenbank-Datenmodell gefunden werden muss; s. hierzu auch Abschnitt 8.2). Für den Einsatz im Rahmen eines CMS sind jedoch noch weitere Komponenten nötig, da Datenbanken eben primär zur Speicherung und nicht auf die Publikation/Aufbereitung von Daten ausgelegt sind. Auch ist der Ansatz, beispielsweise SQL innerhalb von Templates zur Publikation zu verwenden, nicht zufrieden stellend, da SQL zum einen keinen disziplinierten Zugriff bezüglich eines höheren Datenmodells (ER, OO) zulässt, zum anderen zu wenig Kontrollstrukturen bietet und nicht berechnungsuniversell ist. Zudem sind Vererbung oder die hierarchische Struktur von Websites in relationalen Datenbanken nur umständlich abbildbar (vgl. Kapitel 3.4.1). Aktuelle objektorientierte und objektrelationale Datenbanksysteme (z. B. Oracle) vereinen zwar verschiedene geeignete Ansätze wie Klassen und Vererbung, Hierarchie und Methoden (*Stored Procedures*), sind aber in Hinblick auf die Publikation der Daten und ein über normale Vererbung hinausge-

hendes Konzept (für sich alleine gesehen) unzureichend. Hinzu kommt, dass Vererbung und Hierarchien das Mapping in objektrelationalen Datenbanksystemen verkomplizieren, da zusätzliche Aspekte wie beispielsweise die Versionierung auch unterstützt werden müssen.

7.2. XML-basierte Ansätze

7.2.1. XML-Schemasprachen

Eine im Vorfeld der vorliegenden Arbeit durchgeführte Untersuchung gängiger XML-Schemasprachen lieferte die bereits in Kapitel 4 beschriebene Erkenntnis, dass XML alleine in verschiedenen Hinsichten unzureichend für die Datenmodellierung im CMS-Kontext ist. Im Folgenden werden die Ergebnisse dieser Voruntersuchung (im Hinblick auf die identifizierten Anforderungen) kurz vorgestellt.

Die *Document Type Definition (DTD)* wurde schon in den 1970er Jahren für die in erster Linie dokumentenzentriert eingesetzte SGML (s. Abschnitt 2.2.1) entwickelt und war bzw. ist die einzige Schemasprache für SGML. Für XML wurde eine vereinfachte Version der DTDs entwickelt, welche auch in der XML 1.0-Spezifikation[1] enthalten ist.

DTDs sind, bedingt durch ihre SGML-Herkunft, gut für dokumentenzentrierte Nutzung von Dokumenten geeignet und durch ihre Einfachheit sowie die Verwendung der BNF (Backus-Naur-Form)-Notation gut für menschliche Leser verständlich. Vor allem im Bereich der datenzentrierten Nutzung zeigen sich jedoch einige Schwächen, welche in anderen Schemasprachen, deren Entwicklung bereits frühzeitig begann, vermieden werden sollten. Diese Schwächen der DTD bestehen vor allem in den folgenden Aspekten:

- Sie besitzen kein Typsystem, sämtliche Daten sind Zeichenketten (Strings), mit Ausnahme von ID-Typen.
- Sie haben keine XML-Syntax, sowohl Benutzer als auch Programme müssen demnach eine von XML unterschiedliche Syntax verstehen können.
- Sie unterstützen keine XML-Namespaces.

In Bezug auf objektorientierte Konzepte lässt sich sagen, dass man Elemente komplexen Inhalts insofern als Klasse ansehen kann, als dass sie gleichartige Entitäten beschreiben, ein darüber hinaus gehendes Klassenkonzept existiert jedoch nicht. Da alle Elemente auf der oberste Ebene stehen, ist eine Kapselung im eigentlichen Sinne nicht vorhanden, ebenso kein Information-Hiding. Weitere objektorientierte Konzepte wie beispielsweise Vererbung, Methoden oder Polymorphie sind nicht vorhanden.

[1] `http://www.w3c.org/TR/2004/REC-xml-20040204`

Das W3C hat in den Jahren 1998 und 1999 verschiedene Entwürfe für XML-Schemasprachen vorgeschlagen (XML-Data[2], XML-Data reduced[3], Document Content Description (DCD[4]) , Schema for Object-oriented XML (SOX[5]) und Document Definition Markup Language (DDML[6])), deren Hauptkonzepte dann später in *XML Schema* mündeten. Hauptmotivation war es, eine flexible Schemasprache ohne die Schwächen der DTD zu entwerfen. Die wesentlichen Konzepte sind demnach ein umfangreiches Typsystem mit objektorientierten Konzepten, ein Modulkonzept sowie die Eigenschaft, dass XML Schemata selbst in XML-Syntax formuliert sind, also mit einem XML-Parser auf Wohlgeformtheit überprüft werden können.

Ein Klassensystem im objektorientierten Sinne existiert nicht, am nächsten kommen komplexe Typen/Elemente, die beispielsweise eine einfache Form von Vererbung unterstützen. Durch das „Schachteln“ der Definitionen können Elemente bzw. Attribute in andere Elemente gekapselt werden, ein „Verstecken“ im Sinne des Information-Hiding ist nicht möglich. Methoden oder Polymorphie sind nicht vorhanden. Eine detaillierte Analyse dieser Konzepte ist in [Mal02] zu finden.

Ausgangspunkt für *RELAX (REgular LAnguage description for XML)* [Rel00] war der grammatikbasierte Ansatz der DTDs. Die Hauptideen von RELAX sind zum einen eine solide mathematische Grundlage („hedge automaton theory“, [Mur00]) sowie die Vereinfachung der XML Schema Syntax: („Tired of complicated specifications? You just RELAX !“) RELAX definiert keine Datentypen, stattdessen werden die XML Schema Datentypen unterstützt.

TREX erweitert das in XDuce (s. Abschnitt 7.3.2) eingeführte Typsystem mit einigen zusätzlichen Merkmalen zu einer Schemasprache. Grundlage sind zum einen Bäume und zum anderen reguläre Ausdrücke; dabei wurde auch auf eine für menschliche Leser gut verständliche Syntax geachtet.

RELAX NG vereint die wichtigsten Konzepte von RELAX und TREX. Die Sprache hat sowohl eine XML- als auch eine (alternative) kompakte nicht-XML-Syntax. Darüber hinaus werden Elemente und Attribute im Wesentlichen gleich behandelt, zudem können XML Schema Datentypen verwenden werden

Da die Konzepte von DTDs nur im Hinblick auf Datentypen, Namespaces und syntaktische Aspekte erweitert wurden, sind die objektorientierten Konzepte in RELAX NG prinzipiell identisch zu denen der DTDs und somit für die im Rahmen der vorliegenden Arbeit identifizierten Anforderungen weniger geeignet.

Im Folgenden werden einige Ansätze vorgestellt, die versuchen, die Schwachstellen der

[2] `http://www.w3c.org/TR/1998/NOTE-XML-data-0105`
[3] `http://stuff.mit.edu/afs/sipb/user/cbf/xmldata-reduced.htm`
[4] `http://www.w3c.org/TR/1998/NOTE-dcd-19980731`
[5] `http://www.w3c.org/TR/NOTE-SOX`
[6] `http://www.w3c.org/TR/1999/NOTE-ddml-19990119`

eben beschriebenen Schemasprachen zumindest in Teilen zu überwinden.

7.2.2. XML++:

Jamil und Modica stellen in [JM02a] und [JM02b] einen Ansatz vor, objektorientierte Konzepte wie Vererbung mit einem klassenähnlichen Konzept in semistrukturierte Daten bzw. XML zu integrieren. Die „Klassen“ sind jedoch keine richtigen Klassen im Sinne der OO, sondern eigentlich Instanzen, die dann als Schablone bzw. Prototyp verwendet und spezialisiert werden können. Die Architektur von XML++ zielt darauf ab, aus verschiedenen XML++-Dokumenten zunächst *ein* XML-Dokument und daraus dann über ein XSLT-Stylesheet eine Webseite zu generieren. Hierzu wird lediglich das XML-Datenmodell um ein einfaches Vererbungs- bzw. Aggregationsmodell erweitert. XML++ ist ungetypt. Es folgt eine Analyse der objektorientierten Konzepte:

Klassenkonzept: Wie bereits beschrieben, wird nicht zwischen Klassen und Instanzen unterschieden, da Instanzen lediglich über die Default-Werte der Klassendefinitionen definiert sind.

(Mehrfach-)Vererbung: Vererbung (auch Mehrfach-) ist auf Ebene der Daten möglich, wobei auf Grund des unzureichenden Klassenkonzepts eher Delegation (d. h. Vererbung auf Instanzenebene, s. Abschnitt 2.3.4) vorliegt. Bei der Mehrfachvererbung eventuell auftretende Konflikte werden insofern aufgelöst, als dass die Reihenfolge, in der die Vaterklassen spezifiziert wurden, berücksichtigt wird.

Verhalten: Verhalten wird z. T. unterstützt. Methoden einer Hostsprache (momentan nur Java, andere Sprachen sind vorgesehen) können in einem XML-Dokument aufgerufen werden. Die Methoden können die XML++-Elemente bzw. Dokumente nicht verändern, sind in dieser Hinsicht also seiteneffektfrei. Lesend kann über XPath-Ausdrücke auf das XML++-Dokument zugegriffen werden. Methoden werden extern, d. h. außerhalb der eigentlich Klassen/Instanzen definiert, zudem gibt es keine Möglichkeit der Vererbung oder Polymorphie bei Methoden.

Kapselung und Information-Hiding: Daten können in Klassen gekapselt werden, wobei durch die getrennte Definition der Methoden kein Verhalten, sondern nur der Zustand von Klassen bzw. Objekten definiert werden kann. Information-Hiding ist nicht vorgesehen.

Polymorphie: Polymorphie sowie erweiterte Vererbungskonzepte wie Familienpolymorphie sind nicht vorhanden.

Zudem wird ein dokumentenzentrierter Ansatz verfolgt, da immer nur ein Dokument (Datei) „bearbeitet“ wird, aus der nicht auf andere Daten bzw. Dokumente zugegriffen

werden kann, was beispielsweise die Generierung einer Navigationsleiste o. ä. unmöglich macht.

Aus diesen Gründen ist XML++ für die Verwendung im in Kapitel 4 beschriebenen Kontext nicht sinnvoll, da viele Anforderungen nicht bzw. nur sehr umständlich umgesetzt werden könnten.

7.2.3. WebComposition Markup Language

Gellersen und Gaedke stellen in [GG99] und [GSG00] die WebComposition Markup Language (WCML) vor, die einige Parallelen zu XML++ aufweist. So wird auch hier versucht, in XML-Instanzen ein klassenähnliches Konzept mit Vererbung zu integrieren, wobei eben keine „richtige“ Vererbung, sondern nur Delegation/Prototyping unter Instanzen möglich ist. Erweiterte Vererbungskonzepte wie Familienpolymorphie fehlen ebenso wie Verhalten oder ein Typsystem. Auch hier liegt der Fokus auf (dokumentenzentrierter) Publikation, zudem ist auch dieser Ansatz nicht weiterverfolgt worden, so dass außer einigen wenigen Veröffentlichungen keine weiteren Informationen, z. B. über implementierte Prototypen, vorhanden sind.

7.2.4. Object Oriented XML (OOXML):

Das in [Sil98] vorgestellte Konzept von OOXML schneidet einige gute Ansätze in Richtung Objektorientierung an; allerdings wurde das Projekt offenbar nicht weiter verfolgt, so dass außer einem kurzen Konzeptpapier von 1998 nicht mehr Informationen vorhanden sind. Im Rahmen dieses Papiers wird erwähnt, dass XML-Elemente Klassen definieren, Methoden innerhalb von XML-Dokumenten in Lisp spezifiziert werden können und (Einfach-)Vererbung möglich ist.

7.2.5. XML-basierte Ansätze: Fazit

Zusammenfassend lässt sich sagen, dass kein XML-basierter Ansatz die Anforderungen für ein komplexes, objektorientiertes Datenmodell erfüllt. Zum Teil sind einige Ansätze erkennbar, die aber z. B. weitergehende Vererbungsansätze wie beispielsweise Familienpolymorphie nicht unterstützen. Erklärbar ist das z. B. damit, dass die meisten Schemasprachen mehr den dokumenten- und weniger den datenorientierten Ansatz verfolgen.

XML++ verfügt noch über die vielversprechendsten Ansätze in Richtung Objektorientierung, ist aber auf Grund der Vermischung von Klassen und Instanzen, der fehlenden (Familien-)Polymorphie sowie der Dokumentenzentrierung für die vorliegende Arbeit ebenfalls nicht geeignet.

7.3. Programmiersprachen und Erweiterungen

In diesem Abschnitt werden verschiedene Konzepte und Programmiersprachen vorgestellt, die (u. a.) zur Verarbeitung von semistrukturierten Daten entwickelt wurden. Da bereits ausgeführt wurde, dass funktionale Sprachen besonders gut für diesen Zweck geeignet sind, stehen eben diese Sprachen hier im Vordergrund. Dabei werden die Sprachen anhand der beiden folgenden Kategorien gruppiert (vgl. [vDKV00]):

General purpose language (GPL) bezeichnet eine Programmiersprache wie Java oder C++, die für viele Anwendungen einsetzbar ist. Für eine bestimmte Anwendung können dann entweder zusätzliche Bibliotheken (auch: APIs) oder aber *Data bindings*, d. h. Abbildungen der Anwendungsdaten auf Datenstrukturen der jeweiligen GPL, implementiert werden.

Domain specific language (DSL) ist eine speziell auf eine Anwendung bzw. einen Problembereich konzipierte Sprache. Man unterscheidet hier eigenständige (*standalone*), d. h. vollständig neu entworfene Sprachen, und eingebettete (*embedded*) DSLs. Embedded DSLs sind als „Kompromiss“ zwischen GPLs und DSLs anzusehen; hierbei werden spezielle domänenspezifische Datentypen und ggf. syntaktische Konstrukte in eine GPL „eingebettet“. Der Vorteil bei diesem Ansatz ist, dass auf eine bestehende GPL und deren Compiler zurückgegriffen werden kann. Auf der anderen Seite erlauben nicht alle Sprachen derartige Erweiterungen, zudem ist diese Lösung ggf. nicht optimal in Hinblick auf Performanz (da zwei verschiedene Sprachen ausgeführt werden müssen) bzw. die Bedienbarkeit durch Nutzer (da z. B. bei Fehlermeldungen nicht unbedingt direkt ersichtlich ist, ob diese von der eingebetteten oder der einbettenden Sprache kommen).

Der Vorteil von DSLs gegenüber GPLs ist, dass die vorhandenen Datenstrukturen sehr effizient und intuitiv in der Sprache abgebildet/eingebettet werden können, während bei GPLs dazu spezielle Konstrukte vonnöten sind. Im Falle von GPLs in Verbindung mit semistrukturierten Daten und XML kann auch die Typisierung der Daten problematisch sein: werden diese z. B. aus Strings zusammengesetzt, können leicht Dokumente erzeugt werden, die nicht wohlgeformt sind, wie das folgende Beispiel in Java zeigt:

```
String xmlresult = "<name>"+variable+"</test>";
```

Diese Problematik kann beim geschickten Design einer DSL umgangen werden. Des Weiteren können in DSLs ggf. Kontrollstrukturen speziell auf die Anwendung „zugeschnitten“ werden. Ein Vorteil von GPLs ist hingegen, dass bestehende und somit bekannte und bewährte Sprachen eingesetzt werden können.

In den folgenden Abschnitten werden existierende Sprachen, die im Kontext dieser Arbeit von Interesse sind, kurz beschrieben. Weitere Details sind u. a. in [GJS05], [MS05]

und [KMS04] zu finden. Anzumerken ist, dass der Übergang zwischen GPL und DSL z. T. fließend ist, mitunter haben sich einige Sprachen durch konstante Weiterentwicklungen von ursprünglichen DSLs zu GPLs entwickelt.

7.3.1. GPL – Bibliotheken und Data Bindings

Die beiden bekanntesten Vertreter von APIs/Bibliotheken für XML sind SAX (Simple API for XML[7]) und DOM (Document Object Model[8]), welche beide nur die Interfaces definieren, die dann in verschiedenen Sprachen implementiert werden können bzw. worden sind.

DOM arbeitet baumbasiert, d. h. ein (XML-)Dokument wird beim Parsen in eine Baumstruktur eingelesen, welche dann traversiert werden kann. Hierbei muss zunächst immer der gesamte Datenbaum eingelesen werden, was gerade bei großen Datenbeständen ineffizient ist. Im Gegensatz dazu arbeitet SAX ereignisorientiert, d. h. beim Parsen werden Ereignisse (events) produziert, die dann geeignet verarbeitet werden können. SAX ist in der Regel schneller, u. a. da nicht der gesamte Datenbaum eingelesen werden muss. Allerdings erlaubt SAX nur lesenden Zugriff, es können also keine Daten verändert oder neue Elemente generiert werden.

Java APIs und Data Bindings

dom4j[9], SAXDOMIX[10], JDOM[11] , JAXP[12] (Java API for XML processing) und XOM[13] (XML Object Model):

Diese Ansätze bauen auf DOM und SAX auf und stellen eine API zum Zugriff auf XML-Strukturen zur Verfügung. Z. T. werden hier die verschiedenen Ansätze von SAX und DOM kombiniert (z. B. in SAXDOMIX) oder vereinfacht (z. B. XOM). Alle Ansätze sind eher low-level orientiert, d. h. die XML-Strukturen müssen mehr oder weniger manuell traversiert werden.

Castor[14] und JAXB[15]: Beide Systeme ermöglichen es, zu einem gegebenen XML-Schema eine Java-Klassenhierarchie zu generieren („Binding“), welche die Struktur und Daten des Schemas beschreibt. Instanzen dieser Klassen repräsentieren dann die entsprechenden XML-Instanzen. Auf diese Weise kann auf einem höheren Abstraktionsniveau

[7] http://www.saxproject.org
[8] http://www.w3c.org/DOM
[9] http://www.dom4j.org
[10] http://www.devsphere.com/xml/saxdomix
[11] http://www.jdom.org
[12] http://java.sun.com/xml/jaxp
[13] http://www.xom.nu
[14] http://castor.exolab.org
[15] http://java.sun.com/xml/jaxb

als beispielsweise mit JDOM (s. o.) gearbeitet werden (vgl. auch [KL03a]). Die Umwandlung von XML in Objekte nennt man *Unmarshalling*, die umgekehrte Richtung *Marshalling*.

Relaxer[16] verfolgt einen ähnlichen Ansatz wie Castor und JAXB für RELAX Schemas (vgl. Abschnitt 7.2.1). Weitere Java-Systeme sind Zeus, JaxMe, XMLBeans, die prinzipiell ähnliche Ansätze wie die bislang genannten Systeme verfolgen.

Im Kontext der vorliegenden Arbeit scheiden DOM-basierte Ansätze aus, da sie durch Parsen und (Un)marshalling bei großen Datenbeständen ineffizient werden können. SAX käme theoretisch in Frage, jedoch ist ein Traversieren des Datenbaumes auf Grund der (linearen) Ereignisorientierung eher umständlich zu realisieren – um beispielsweise auf bereits besuchte Knoten zugreifen zu können, müsste der gesamte Baum zwischengespeichert werden, wodurch der Effizienzvorteil von SAX wieder zunichte gemacht würde. Auch die anderen genannten Ansätze sind eher ungeeignet, da ein erheblicher Overhead durch das Transformieren aus der Baum- in die Objektdarstellung (Marshalling) entsteht, der gerade bei großen Datenmengen zum Tragen kommt. Darüber hinaus sind die Vorteile funktionaler Sprachen (wie Seiteneffektfreiheit oder referentielle Transparenz) in Java nicht vorhanden, wodurch weitere Probleme entstehen (bzw. einige Vorzüge wegfallen).

APIs und Data Bindings für Haskell

HaXml [WR99] ist eine Sammlung von Haskell-Programmen zum Parsen, Filtern, Transformieren und Generieren von XML Dokumenten. Grundidee ist es, DTDs in das Typsystem (genauer: in eine Serie von Typdeklarationen) und Dokumente in Werte von Haskell zu übersetzen. Durch die Verwendung der DTD findet beim Validieren eines Dokuments automatisch eine Typüberprüfung statt. Diesen Ansatz nennt man auch „getyptes Data Binding“, im Gegensatz zum ungetypten, bei dem keine DTD bzw. XML Schema verwendet wird. Des Weiteren stellt HaXml eine „Combinator library“ mit higher-order Funktionen zur Verarbeitung von XML bereit, mit denen sich sehr kompakte Programme schreiben lassen.

HXML[17] und die Haskell XML Toolbox [Sch02] sind Weiterentwicklungen von HaXml und zeichnen sich vor allem durch die „lazy evaluation“ (Bedarfsauswertung) aus, d. h. es werden immer nur die Bereiche des Dokuments geparsed, die zur Berechnung benötigt werden. Beide Systeme unterstützen allerdings nur den ungetypten Ansatz, können dafür aber über einen XPath-Filter bestimmte Unterbäume der Dokumente adressieren. Die Haskell XML Toolbox verwendet darüber hinaus ein generisches Datenmodell, um die Dokumente zu repräsentieren.

[16] `http://www.relaxer.org`
[17] `http://www.flightlab.com/~joe/hxml`

UUXML [ACJ04] realisiert getypte Data Bindings in Verbindung mit XML Schema. Harp [BFS04] ist eine lightweight Implementierung, die regular expression patterns (wie XDuce und CDuce, s. Abschnitt 7.3.2) bereitstellt.

Während in diesen Data Bindings für Haskell die Vorteile des funktionalen Paradigmas vorhanden sind, fehlen die Vorzüge der Objektorientierung (wie Vererbung oder Kapselung). Zudem gelten auch hier die oben erwähnten Nachteile bzgl. der Transformation zwischen verschiedenen Datenrepräsentationen.

7.3.2. Standalone-DSLs

XSLT

XSLT (XSL Transformations[18]) bildet, zusammen mit XPath (XML Path Language[19], eine Sprache zur Adressierung/Referenzierung von Teilen von XML-Dokumenten) und XSL-FO (XSL Formatting Objects[20], zuständig für die Formatierung/Ausgabe von XML-Dokumenten) die *Extensible Stylesheet Language Family (XSL).*

XSLT ist eine funktionale Sprache und darüber hinaus template-basiert; ein Template entspricht in etwa einer Funktion in einer funktionalen Programmiersprache. Die Templates werden anhand von Mustern zur Berechnung ausgewählt. Higher-order Funktionen sind möglich, z. T. allerdings nur mit zusätzlichen Erweiterungen (*FXSL-XSLT Functional Programming Library*, [Nov03]).

Im Kontext der vorliegenden Arbeit sind diese Aspekte zwar von Vorteil, XSLT ist jedoch als Kernkomponente eher ungeeignet, da es nur funktional und nicht objektorientiert ist – so ist beispielsweise eine Kapselung nur schwer möglich, da Daten und zugehöriges Verhalten getrennt voneinander sind. Auch sind keine Vererbungskonzepte (bzw. nur rudimentär) vorhanden. Des Weiteren ist XSLT nur zur XML-*Verarbeitung* und nicht zur *Modellierung* von Daten geeignet.

XQuery

Ende der 1990er Jahre existierten viele Konzepte für XML-Anfragesprachen[21], welche dann in XQuery mündeten. Mit XQuery[22] wurde versucht, eine Anfragesprache für XML-Dokumente bzw. Daten zu konzipieren, welche das SQL-Modell auf das komplexere Datenmodell von semistrukturierten Daten generalisiert. Auch hier wurden funktionale Ansätze verarbeitet, zudem können die Datentypen von XML Schema verwendet werden.

[18] `http://www.w3c.org/TR/xslt20`
[19] `http://www.w3c.org/TR/xpath20`
[20] `http://www.w3c.org/TR/xsl11`
[21] u. a. XQL, XML-QL, OQL, Lorel, YATL ([SC99], [CS00]) und Quilt, für einen Überblick und Vergleich s. [Püt01]
[22] `http://www.w3c.org/TR/xquery`

Prinzipiell ist XQuery gut geeignet, Daten aus XML-Dokumenten zu „extrahieren“, jedoch ist es nicht darauf ausgelegt, komplexere Logik und Publikationsaufgaben auszuführen – vergleichbar mit SQL, das in der Regel auch nur zur Datenabfrage, aber nicht zur weiteren Verarbeitung eingesetzt wird. Zudem sind keine objektorientierten Konzepte wie beispielsweise Polymorphie oder erweiterte Vererbungskonzepte vorhanden.

XDuce, CDuce, XHaskell, Xtatic und XMλ

XDuce ([HP00], [HP03]) ist eine funktionale, statisch getypte Sprache, welche Einfluss auf weitere Sprachen sowie Schemaentwicklungen gehabt hat. Die Hauptmerkmale sind zum einen sog. „regular expression types“ als eine Generalisierung von DTDs mit entsprechenden pattern-matching Mechanismen sowie zum anderen lokale Typinferenz: bei Funktionsargumenten werden Typen spezifiziert, beim pattern matching werden sie inferiert. Demgegenüber können keine higher-order Funktionen verwendet werden und Element- und Attributnamen können nicht dynamisch berechnet werden. Die in einer ersten Version fehlende Unterstützung für Attribute und ungeordnete Daten wurde in [HM03] nachgereicht.

Ausgehend von XDuce wurde mit den Sprachen CDuce und XHaskell [LS04] versucht, allgemeinere, d. h. nicht so speziell auf XML zielende, Sprachen zu entwickeln. XHaskell integriert XML typsicher in Haskell. CDuce [BCF03] erweitert das Typsystem von XDuce um nicht-XML-spezifische Aspekte wie beispielsweise das Kreuzprodukt oder boolsche Operatoren und führt higher-order Funktionen sowie Iteratoren ein.

Mit Xtatic [GLPS04], einer embedded DSL, wurde versucht, den Ansatz von XDuce auf objektorientierte Sprachen am Beispiel von C# zu übertragen.

XMλ [MS99] ist ebenfalls eine funktionale und statisch getypte Sprache, die in erster Linie zum Generieren von XML-Dokumenten entworfen wurde. Ein XML/HTML-Quelltext ist für XMλ nicht nur ein Stringliteral, sondern ein Programm, bei dem jedes Element ein Programmkonstrukt mit zugehörigem Typ ist. Ein XML-Element entspricht demnach einem XMλ-Typ, eine DTD definiert also alle dieser DTD zugehörigen XMλ-Typen. Fragmente eines (XML-)Dokuments sind XMλ-Werte. Validiert wird ein Dokument beim Parsen; ein Dokument ist gültig, wenn das entsprechende XMλ-Programm typkorrekt ist.

Alle diese Konzepte bieten jeweils gute Teilansätze (wie das funktionale Paradigma oder das Typsystem) im Kontext der vorliegenden Arbeit, sind jedoch insgesamt jeweils nicht geeignet, da objektorientierte Konzepte wie Vererbung fehlen und XML als Datentyp sehr im Mittelpunkt steht.

7.3.3. Embedded-DSLs

XAct, XJ und XOBE

XAct [KMS04], XJ [HRS+05] und XOBE [KL03b] sind eingebettete DSLs für Java, in denen XML-Baumstrukturen eingelesen und als Java-Objekte repräsentiert werden, in denen dann u. a. mit XPath navigiert werden kann (ähnlich dem oben beschriebenen Marshalling)

Während diese Ansätze gut zur Verarbeitung von XML sind, sind sie doch im Kontext der vorliegenden Arbeit eher schlecht geeignet, da die Daten wiederum zwischen zwei „Welten“ (XML und Java) transformiert werden müssen, keine funktionalen Aspekte vorhanden sind und Vererbung zwar im Rahmen von Java, nicht aber im Rahmen der (XML-)Daten verwendet werden kann.

Featherweight Java

Featherweight Java (FJ, [IPW99]) ist eine Reduzierung von Java auf einen kompakten funktionalen Kalkül, ähnlich wie der λ-Kalkül für Haskell oder ML. Die Kernidee ist, den Sprachumfang von Java auf eine minimale Menge von Ausdrücken zu reduzieren, um so ein kompaktes Berechnungsmodell zu erhalten. U. a. durch Weglassen von Zuweisungen entstand dadurch eine funktionale Sprache. Igarashi erweitert FJ in [ISV05] um Familienpolymorphie. Somit sind zwei im Kontext der vorliegenden Arbeit wichtige Aspekte (funktionales Paradigma, Vererbung mit Familienkonzept) in diesem Ansatz enthalten. Was jedoch fehlt, ist die Möglichkeit, in hierarchischen Daten navigieren und die Daten in verschiedenen Formaten publizieren zu können.

7.3.4. Templatesprachen

Servlets und JSP: Servlets sind prinzipiell reguläre Java-Programme, die auf einem (Web-)Server ausgeführt werden und (über normale `println()`-Statements) HTML-Ausgabe erzeugen. Da Servlets jedoch gerade bei wenig Programmlogik und viel Ausgabe recht unübersichtlich werden können, hat man mit Java Server Pages (JSP) das Modell umgekehrt: hier werden in HTML-Seiten Java-Ausdrücke bzw. Programmteile eingebettet.

Servlet-Beispiel (Ausschnitt):

```
out.println("<title>");
if (title == null)
  out.println("(kein Titel)");
else
  out.println(title);
out.println("</title>");
```

JSP-Beispiel (Ausschnitt):

```
<title>
<%if (title == null) { %>
  (kein Titel)
<% } else { %>
  <%=title%>
<% } %>
</title>
```

Velocity[23]: Mit dieser ebenfalls Java-basierten Templatesprache wurde versucht, die in Servlets und JSP mitunter nicht konsequent durchgehaltene Trennung von Inhalt, Logik und Layout (Separation of Concerns, Abschnitt 3.1.1) mehr in den Mittelpunkt zu rücken. Davon abgesehen sind die Technologien ähnlich.

ASP: ASP (Active Server Pages)[24] ist eine von Microsoft entwickelte Templatesprache, die in Ihrem Aufbau sehr JSP ähnelt, statt Java aber VBScript verwendet.

PHP: PHP[25] ist eine vor allem im Bereich von CMS verbreitete Templatesprache (vgl. Abschnitt 3.1.1). Ausdrücke dieser (domänenspezifischen) Sprache werden in HTML-Dokumente eingebettet.

XSP: XSP (eXtensible Server Pages) wurde im Rahmen des Cocoon-Projekts (s. Abschnitt 3.3) entwickelt. XSP-Programme sind gültige XML-Dokumente, können also beispielsweise auch von einem XML-Parser überprüft werden. Eingebettet werden können verschiedene Sprachen wie z. B. Java oder PHP. XSP-Programme erzeugen im Gegensatz zu anderen Templatesprachen nicht direkt HTML, sondern SAX-Events, die dann von Transformern weiterverarbeitet oder via Serializer in das gewünschte Ausgabeformat umgewandelt werden können. Auf diese Weise kann XSP einfach in eine Cocoon-Pipeline integriert werden. Von Vorteil ist weiterhin, dass – genau wie im W2OM-Konzept – wohlgeformtes XML erzeugt wird.

7.3.5. Programmiersprachen: Fazit

Im Hinblick auf die in Kapitel 4 identifizierten Anforderungen – insbesondere die schnelle Texterzeugung – ist eine Templatesprache prädestiniert. Der Einsatz eines speziellen, inkrementellen Datenmodells hingegen spricht für eine DSL, da diese effizient und in einfacher Weise mit diesem Modell arbeiten kann. Besonders geeignet sind demnach templatebasierte DSL-Sprachen.

[24] http://msdn.microsoft.com/workshop/server/asp/aspfeat.asp
[25] http://www.php.net

In den vorherigen Abschnitten zeigte sich, dass existierende Sprachen (bzw. deren Datenmodelle) alleine nicht alle gewünschten Aspekte abdecken, so dass höchstens Kombinationen der Sprachen mit anderen Datenmodellen Sinn machen würden. Diese Kombination ist aber wegen Unterschieden in den (übrigen) Sprachkonzepten nicht nahtlos möglich, darüber hinaus wäre auch die Semantik eines solchen „Patchwork"-Systems nicht klar definiert. Der im Rahmen der vorliegenden Arbeit entwickelte Ansatz bietet sämtliche Aspekte in *einem* Konzept.

Eine Möglichkeit wäre es beispielsweise, Java als GPL einzusetzen. Da (reines) XML als Datenmodell aus den o.g. Gründen ausscheidet, müsste hierfür ein neues Daten- bzw. Objektmodell entwickelt werden. Erweiterte Vererbungskonzepte sind in Java allerdings auch nur über Zusätze (s. beispielsweise [Wit03]) möglich. Eine weitere Variante wäre der Einsatz des Document Object Models (DOM), das zwar eine elegante Modellierung semistrukturierter Baumstrukturen ermöglicht, jedoch zum einen für große Datenmengen ineffizient ist, zum anderen verschiedene Aspekte wie beispielsweise die Navigation im Baum zum Teil nur umständlich und wenig elegant durchführbar sind.

7.4. Erweiterte Vererbungskonzepte

In diesem Abschnitt werden erweiterte Vererbungskonzepte, die (alternativ zum gewählten Familienkonzept) für den Einsatz im Rahmen der vorliegenden Arbeit möglich erschienen, vorgestellt und im Hinblick auf die Anforderungen untersucht. Allen Ansätzen ist gemein, dass sie über die normalen Vererbungskonzepte (Einfach- und Mehrfachvererbung) hinausgehende Aspekte beinhalten und so eine weitergehende inkrementelle Spezifikation erlauben.

Für allgemeine vergleichende Untersuchungen dieser Ansätze sei auf die Arbeiten von Nystrom et al. [NCM04] sowie Jolly et al. ([JDAO04], [Jol04]) verwiesen.

Das folgende Beispiel motiviert die Notwendigkeit für erweiterte Konzepte im Kontext von Programmiersprachen: in Listing 7.1 ist eine einfache Klassenstruktur gezeigt, die einen Graphen mit Knoten (Node) und Kanten (Edge), die jeweils zwei Knoten verbinden, definiert. Die `Node`-Klasse hat eine Methode `touches()`, welche überprüft, ob eine Kante e den Knoten berührt. Diese Struktur soll nun insofern erweitert werden, als dass eine Kante „deaktiviert" werden kann. Dazu erweitern die Klassen `OnOffNode` und `OnOffEdge` die Klassen `Node` bzw. `Edge`.

```
class Node  {
  boolean touches(Edge e) { return(this==e.n1) ||(this==e.n2) }
}
class Edge  { Node n1, n2; }

class OnOffNode extends Node
  boolean touches(Edge e) {
```

```
    return((OnOffEdge)e).enabled ? super.touches(e) : false;
  }
}
class OnOffEdge extends Edge {
  boolean enabled;
  OnOffEdge() { this.enabled=false; }
}
```

Listing 7.1: Familienkonzept(1)

Nachfolgend ist eine mögliche Anwendung zu sehen:

```
1  public class Main {
2    static void build(Node n, Edge e, boolean b) {
3      e.n1=e.n2=n;
4      if (b == n.touches(e)) System.out.println("OK");
5    }
6
7    public static void main(String[] args) {
8      build(new Node(), new Edge(), true);             // OK
9      build(new OnOffNode(), new OnOffEdge(), false); // OK
10     build(new OnOffNode(), new Edge(), true);// ClassCastException!
11   }
12 }
```

Das Problem ist, dass das Mischen der Klassen aus verschiedenen „Familien“ in Zeile 10 zu einem Laufzeitfehler führt, der statisch nicht erkannt werden kann.

7.4.1. Virtuelle Typen und Klassen

Eingeführt wurde das Konzept der virtuellen Typen und Klassen (virtual types/classes, [MMP89]) zuerst in der Sprache Beta [MMPN93], welche von Simula[DN66], einer der ersten objektorientierten Sprachen, abstammt. (vgl. auch [EOC06])

Virtuelle Klassen sind klassenwertige Attribute von Objekten. Sie werden, ähnlich wie virtuelle Methoden oder innere Klassen, in einer Klasse spezifiziert und können demnach auch abgeleitet und überschrieben werden. Im Unterschied zu „normaler“ Vererbung können diese Klassen nicht nur einfach überschrieben, sondern erweitert werden.

Im Gegensatz zu inneren Klassen werden virtuelle Klassen dynamisch gebunden, d. h. der eigentliche Wert einer virtuellen Klasse kann erst zur Laufzeit festgelegt werden, wodurch ein hoher Grad an Flexibilität ermöglicht wird. Da Methodenparametertypen kovariant sein können, sind virtuelle Klassen nicht statisch typsicher (s. Abschnitt 2.3.6).

Virtuelle Klassen bilden die Grundlage für verschiedene weitere Arbeiten und Sprachen wie beispielsweise gbeta [Ern99], Caesar [Wit03] oder VC [AO03].

Im nachfolgenden Listing ist ein Beispiel virtueller Klassen zu sehen. Die (virtuellen) Klassen `Expr` (Zeile 2) und `Num` (Zeile 5) werden in der Klasse `EvalExpr` um eine

`eval()`-Methode erweitert (Zeilen 14 und 20), die `print()`-Methode der `Eval`-Klasse wird überschrieben (Zeile 15).

```
1  class ExprFamily {
2    virtual class Expr {
3      void print() {}
4    }
5    virtual class Num extends Expr {
6      final int value;
7      Num(int _value) { value = _value; }
8      void print() { printf(value); super.print(); }
9    }
10 }
11
12 class EvalExpr extends ExprFamily {
13   override class Expr {
14     int eval() { return 0; }
15     void print() {
16       printf("eval to"); printf(eval()); super.print();
17     }
18   }
19   override class Num  {
20     int eval() { return this.value; }
21   }
22 }
```

7.4.2. Mixins, Mixin Layers und Delegation Layers

Ein Mixin [BC90] ist eine abstrakte Klasse, die es erlaubt, neue Attribute oder Methoden in verschiedenen Klassen einzufügen. Mit anderen Worten: ein Mixin beschreibt nur die inkrementellen Änderungen, die an einer (bzw. mehreren) Klassen durchgeführt werden sollen. Da sie abstrakte Klassen sind, können sie nicht instanziiert werden.

Das Mixin-Konzept ist eher für kleine Änderungen ausgelegt, Mixins werden jeweils nur auf eine Klasse angewendet; Smaragdakis und Batory erweiterten daher in [SB02] das Konzept zu Mixin Layers. Damit ist es möglich, nicht nur einzelne Klassen, sondern mehrere Klassen gleichzeitig mit Mixins zu erweitern. Mixins können dabei andere Mixins enthalten. Mixins und Mixin Layers definieren ein statisches Vererbungskonzept.

Ostermann wiederum erweitert Mixin Layers in [Ost02] zu Delegation Layers, die im Gegensatz zu Mixin Layers ein Familienkonzept beinhalten. Grundgedanke ist hierbei, die Konzepte der Delegation und der virtuellen Klassen (s. o.) miteinander zu vereinen. Auf diese Weise können Methodenaufrufe auch über eine Kette von Instanzen (→ Delegation, s. Abschnitt 2.3.4) anstatt nur über die Klassenhierarchie abgewickelt werden. Der Ansatz ähnelt der Vererbung entlang der Instanzenhierarchie im Wob 1-Modell (s.

Abschnitt 3.3). Da Delegation Layers auf virtuellen Klassen und (dynamischer) Delegation basieren, sind sie nicht statisch typsicher.

7.4.3. Familienpolymorphie und Higher-Order Hierarchies

Die Kernidee von Familienpolymorphie [Ern01] ist es, eine generalisierte Form der Polymorphie einzuführen, mit der die Beziehungen zwischen Klassen durch Gruppierung von Klassen zu Familien polymorph deklariert werden können.

Ernst motiviert das Konzept mit diesem anschaulichen Beispiel:

> In einer Lobby eines Hotels warten mehrere Personen darauf, einem Zimmer zugewiesen zu werden. Der Hotelangestellte fragt einen Mann „Sind Sie ein Ehemann?“ und eine Frau „Sind Sie eine Ehefrau?“. Nachdem diese mit „Ja“ antworten, werden sie einem Zimmer zugewiesen, zusammen mit einem Mädchen, das angibt, sie sei eine Tochter.

Die Problematik in diesem Fall ist, dass Personen untereinander in Beziehung gesetzt werden, die zwar eine bestimmte/entsprechende Rolle innehaben, jedoch in verschiedenen Familien. Dieses Beispiel lässt sich auf programmiersprachliche/objektorientierte Konzepte und, wie im Rahmen der vorliegenden Arbeit, auch auf Datenmodellierungsaspekte übertragen.

Vereinfacht lässt sich sagen, dass im Falle der Familienpolymorphie die Familienzugehörigkeit über ein Attribut der Klasse bzw. der Objekte realisiert wird und so vermieden werden kann, dass Klassen verschiedener Familien untereinander „vermischt“ werden.

Das nachstehende Listing zeigt, wie das obige edge/node-Beispiel mit dem Familienkonzept von Ernst gelöst werden kann. Dabei werden die zueinander gehörigen Klassen `Node` und `Edge` in den Klassen `Graph` bzw. `OnOffGraph` gruppiert (Zeilen 1-6 bzw. 7-14).

```
1  class Graph {
2    class Node {
3      boolean touches(Edge e) { return(this==e.n1) ||(this==e.n2); }
4    }
5    class Edge  { Node n1, n2; }
6  }
7  class OnOffGraph extends Graph {
8    class Node {
9      boolean touches(Edge e) {
10       return(OnOffEdge)e).enabled ? super.touches(e) : false;
11     }
12   }
13   class Edge { boolean enabled; }
14 }
15
```

```
16 final g1=new Graph;
17 final g2=new OnOffGraph;
18
19 new g1.Edge(new g1.Node, new g1.Node); // OK
20 new g2.Edge(new g2.Node, new g2.Node); // OK
21
22 new g2.Edge(new g1.Node, new g2.Node); // (statischer) Typfehler
```

Zudem kann explizit spezifiziert werden, von welcher (Instanz einer) Familie eine Klasse instanziiert werden soll. Beispielsweise werden in Zeile 19 eine Kante und zwei Knoten der `Graph`-Familie instanziiert, in Zeile 20 der `OnOffGraph`-Familie. Werden Klassen zwischen Familien gemischt (Zeile 22), führt dies zu einem Compilierfehler. Zu beachten ist, dass im Ansatz von Ernst die Familien mit Instanzen und nicht mit Klassen assoziiert sind (im Beispiel: `new g1.Node` statt `new Graph.Node`). Demnach ist die Zahl der Familien (theoretisch) unbegrenzt.

Das Konzept der Familienpolymorphie wurde bislang in den folgenden Programmiersprachen umgesetzt: gbeta (eine Erweiterung der Sprache Beta), die im Rahmen der Dissertation von Ernst [Ern99] entwickelt wurde; Wittmann überträgt das Konzept in [Wit03] auf Java.

In [Ern03a] erweitert Ernst das Modell zu sog. *Higher-Order Hierarchies*, welche es erlauben, komplexere Vererbungshierarchien als Ganzes abzuleiten - der Zusammenhang ist hier ähnlich wie zwischen Mixins und Mixin/Delegation Layers (s. o.). Der Unterschied ist, dass Higher-Order Hierarchies sich auf die klassenbasierte Vererbung und die Delegation Layers sich auf die Vererbung auf Instanzenebene beziehen.

7.4.4. Nested inheritance

Nested Inheritance [NCM04] ist eine eingeschränkte, statisch typsichere Version des Familienkonzepts bzw. der Higher-Order Hierarchies in Kombination mit virtuellen Klassen. Im Unterschied zur Familienpolymorphie, wo jede Instanz eine Familie definiert, wird bei der nested inheritance eine Familie durch die (Container-)Klasse (im Beispiel `Graph` bzw. `OnOffGraph`) definiert, d. h. es kann nur eine begrenzte Anzahl an Familien geben. Wie in [NCM04] gezeigt, ermöglicht dies eine bessere Wiederverwendbarkeit (Re-Use) und eine bessere statische Typsicherheit.

Das nachfolgende Listing zeigt eine Lösung des Graph-Beispiels mit nested inheritance. Die Klassendeklaration ist identisch zum Ansatz von Ernst, allerdings werden bei der Instanziierung (Zeilen 16, 19 und 20) nicht die Instanzen, sondern die (Container-)Klassen spezifiziert (also `g2.class.Node` statt `g2.Node`).

```
1 class Graph {
2   class Node  { bool touches(Edge e) {
3       return(this==e.n1) ||(this==e.n2);}
4   }
```

```
5   class Edge  { Node n1, n2; }
6 }
7 class OnOffGraph extends Graph {
8   class Node { bool touches(Edge e) {
9       return(OnOffEdge)e).enabled ? super.touches(e) : false;
10    }
11  }
12  class Edge { bool enabled; }
13 }
14
15 final Graph g1 = new Graph;
16 new g1.class.Edge(new g1.class.Node,new g1.classNode); // OK
17
18 final Graph g2 = new OnOffGraph;
19 new g2.class.Edge(new g2.class.Node,new g2.class.Node);// OK
20 new g2.class.Edge(new g1.class.Node,new g2.class.Node);//Cmp.fehler
```

7.4.5. Perspektiven

Reddig bildet in [Red01] das Konzept der Sichten (views) in Datenbanken auf den objektorientierten programmiersprachlichen Kontext ab. Die Motivation ist hier, dass ein und dasselbe Objekt je nach Anwendung aus einer anderen *Perspektive* verwendet werden kann bzw. soll. Dies wird an Beispielen aus dem Bauingenieurswesen erläutert: ein `Bauteil` kann demnach, je nach Sichtweise, als `BauteilStatik` oder `BauteilTragwerk` angesehen werden. Um redundante Redefinitionen zu vermeiden, wird vorgeschlagen, durch Typbeschreibungen statisch zur Übersetzungszeit festzulegen, welche Objekte zu welcher Perspektive zugehörig sind. Somit verfolgt Reddig einen etwas anders gelagerten Ansatz als für die vorliegende Arbeit benötigt.

7.5. Weitere Konzepte

7.5.1. Materialisierungssemantik

Der im W2OM verwendete Ansatz der Materialisierungssemantik, d. h. die Idee, nicht materialisierte Daten aus Gründen der Effizienz zu materialisieren, ist auch in anderen Forschungsgebieten populär. Im Gebiet Datenbanken und Data Warehouse etwa finden sich viele Arbeiten, u. a. in [ZGMHW95] oder[BDD$^+$98]. Die Idee hier ist, Sichten (Views) zu materialisieren, um schnelleren Zugriff auf sie zu haben. Auch im Bereich der deduktiven Datenbanken (u. a. [CGH94]) ist Materialisierung ein fester Begriff.

7.5.2. Daten- und Feldobjekte

Wie bereits in Abschnitt 4.2.2 erläutert, kann das im Rahmen der vorliegenden Arbeit entworfene Konzept der Daten- und Feldobjekte als eine spezielle Form des Composite-Entwurfsmusters (vgl. [GHJV95]) gesehen werden. In beiden Varianten ist die Grundidee eine weitestgehende Abstraktion zur Vermeidung von Redundanzen.

7.6. Fazit

Die Abschnitte 7.1 bis 7.3 zeigten, dass weder reine Datenbanklösungen noch XML für sich alleine alle Anforderungen abdecken können. Als gut geeignet erwiesen sich jedoch verschiedene objektorientierte Konzepte, insbesondere spezielle Vererbungsmodelle.

Prinzipiell gehen alle in Abschnitt 7.4 vorgestellten Vererbungstechniken in die richtige, d. h. für die Anforderungen der vorliegenden Arbeit passende, Richtung. Wie allerdings in Kapitel 4 hergeleitet, sollte ein geeignetes Konzept zum einen klassen- und nicht instanzenbasiert und zum anderen statisch typsicher sein. Des Weiteren ist es nicht nötig bzw. gewünscht, dass Objekte unterschiedlicher Familien untereinander gemischt werden, d. h. die Familien sind in sich „geschlossen“.

Aus diesem Grunde wurde für das Konzept der vorliegenden Arbeit eine statisch typsichere Familienvererbung auf Klassenebene umgesetzt, die sich aus den soeben vorgestellten Ansätzen „zusammensetzt“: die klassenbasierte Herangehensweise der virtuellen Klassen zusammen mit dem Familiengedanken der Familienpolymorphie und der statischen Typsicherheit der Nested inheritance. Neu hinzugekommen ist der Ansatz, Familien in sich zu „kapseln“. Auf diese Weise ermöglicht es das Konzept, Daten weitestgehend redundanzfrei und statisch typsicher modellieren zu können.

8. Ausblick auf ein integriertes System

Wie schon ausgeführt, entstand bereits bei der Entwicklung des Wob 1-Systems die Idee, ein adäquates Datenmodell für CMS zu entwickeln, welches verschiedene Aspekte aus unterschiedlichen Forschungsrichtungen miteinander vereint. Einige Aspekte der im Rahmen der vorliegenden Arbeit entstandenen theoretischen Ideen wurden dabei prototypisch umgesetzt, um den konzeptuellen Ansatz des Modells zu überprüfen und zu untermauern („proof of concept“). Alle Ergebnisse der Arbeit werden in die Entwicklung eines konkreten Systems, dem Wob 2-System, einfließen. Wob 2 ist somit also ein konkretes CMS, welches die als Framework ausgelegten theoretischen Konzepte der vorliegenden Arbeit integriert und dem Entwickler somit eine redundanzfreie, flexible und mächtige Möglichkeit der Modellierung von CMS-Daten zur Verfügung stellt. Das System wird in Java geschrieben und setzt auf offene Standards und Technologien auf.

Wob 2 soll in Zukunft kontinuierlich erweitert und verbessert werden und das Wob 1-System auf mittelfristige Sicht ablösen, indem die dort laufenden Anwendungen nach und nach portiert werden.

In diesem Kapitel werden die wichtigsten bzw. aus wissenschaftlicher Sicht interessantesten Grundaspekte der Architektur sowie einiger implementierungsnaher Details wie der Zwischensprache oder der Updatekomponente genauer vorgestellt. Auf diese Weise wird der Aufbau des Systems und die Umsetzung des theoretischen Konzepts illustriert.

8.1. Architektur

Um ein System leicht erweitern und es so auf neue Anforderungen anpassen zu können, sollte es eine offene, flexible Architektur haben. Beispielsweise erlaubt ein Komponenten- oder Pipeline-Konzept (vgl. Abschnitt 3.3) das „Zusammenstecken“ von Komponenten, die später angepasst bzw. ausgetauscht werden können. Dies umfasst auch das Datenbank-Backend, das ebenfalls transparent für das System austauschbar sein soll, um nicht von einem Produkt bzw. einer Technologie abhängig zu sein; dadurch ist das System offen für zukünftige Entwicklungen und Erweiterungen.

Wob 2 wird aus einem vergleichsweise kleinen Systemkern in Verbindung mit einer initialen Familie bestehen . Erweiterungen und zusätzliche Funktionalitäten werden dann über abgeleitete Familien realisiert, ohne dass das Kernsystem verändert werden muss. Dies ermöglicht einen modularen Aufbau sowie ein „rapid prototyping“. Ein ähnliches Vorgehen hat sich bereits beim Wob 1-System bewährt.

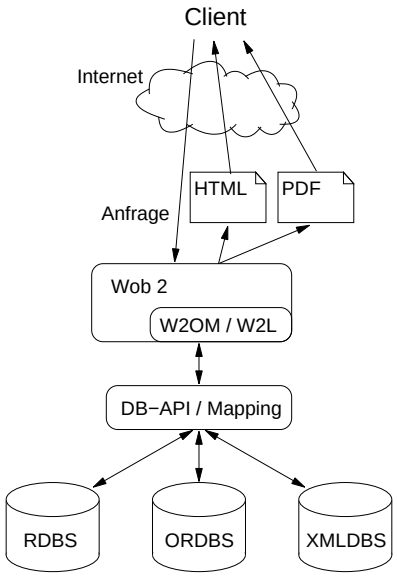


Abbildung 8.1.: Gesamtaufbau des Wob 2-Systems

Der Gesamtaufbau des Wob 2-Systems ist in Abbildung 8.1 dargestellt. Ein Client (Browser) stellt eine Anfrage an Wob 2. Auf Basis dieser Anfrage generiert das System aus den Daten der Datenbankschicht ein Dokument im gewünschten Ausgabeformat und liefert dieses an den Client zurück.

Abbildung 8.2 skizziert das Konzept der Materialisierungssemantik schematisch. Auf der obersten Ebene stehen die abstrakten Familiendefinitionen (in W2L). Aus diesen Definitionen werden dann zur Übersetzungszeit vom System die konkreten Familien materialisiert. Wie in Kapitel 5 spezifiziert, werden bei dieser Expansion jeweils die impliziten, in Superfamilien und -klassen definierten, Methoden und Felder materialisiert. In diesem Schritt werden also alle Vererbungsaspekte sowie die (Familien-)Polymorphie aufgelöst.

In einem weiteren Schritt werden diese Familiendefinitionen jeweils in einen Verhalten- und einen Zustand-Bereich aufgeteilt. Motivation hierbei ist, dass für bestimmte Aufgaben wie beispielsweise die GUI-Generierung nur die Spezifikation des Zustandes benötigt wird und aus den Methoden via Wob 2-Klassendateien Java-Classfiles erzeugt werden können. Das Familienkonzept wird also quasi als „Präprozessor“ verwendet, um einfach und redundanzfrei Vererbung spezifizieren zu können. Durch die Expansion ist jede Anwendung/Familie vollständig materialisiert, wodurch zur Laufzeit keine (Zeit kostenden) Materialisierungen mehr nötig sind.

Nachdem das Wob 2-Konzept bislang eher allgemein bzw. schematisch vorgestellt wurde, folgt nun eine konkrete Beschreibung der Anfragebearbeitung.

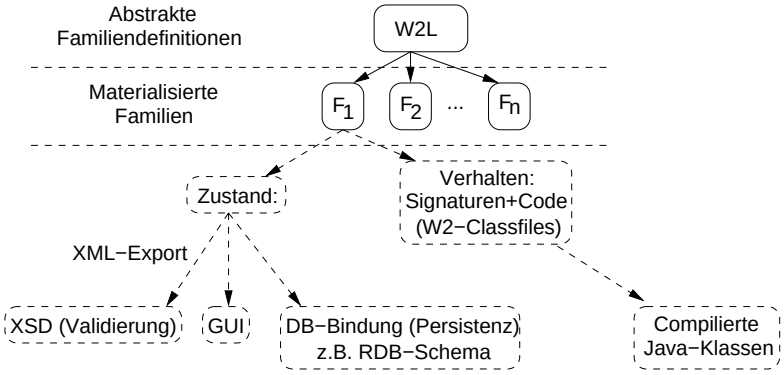


Abbildung 8.2.: Materialisierungssemantik, schematisch

Eine Anfrage, die über die URL gestellt wird, besteht prinzipiell aus: Familienname, Objekt-ID, Methodenname und gewünschtem Ausgabeformat sowie optionalen Parametern. Ein Beispiel sieht folgendermaßen aus:

`http://wob2.iai.uni-bonn.de/Demo/view/de/761235.html`

Dabei steht nach der Server-Adresse (`wob2.iai.uni-bonn.de`) zunächst die Familie, an die die Anfrage gestellt wird (`Demo`), gefolgt von der auszuwertenden Methode (`view`), der gewünschten Ausgabesprache (`de`), der (Daten-) Objekt-ID (`761235`)[1] und dem gewünschten Ausgabeformat (`html`). In diesem Falle würde also die Methode `view` des Objekts mit der ID `761235` der Familie `Demo` ausgewertet, das Ergebnis wird in HTML ausgegeben.

Die Grammatik der W2L-Sprache wurde mit Hilfe des Java Compiler Compiler (JavaCC[2]) beschrieben. Dieser Parsergenerator erzeugt aus einer spezifizierten Grammatik die nötigen Java-Dateien, die zum Parsen und Weiterverarbeiten von W2L-Dateien benötigt werden. Als Grundlage der W2L-Sprache wurde, wie bereits erwähnt, die Syntax der Sprache Java verwendet und dem Konzept entsprechend erweitert bzw. verfeinert. Wie in Abbildung 8.2 angedeutet, werden bei der Übersetzung aus dem W2L-Code (durch die Materialisierungssemantik in sich geschlossene) „compilation units“ erzeugt, aus denen dann compilierte Java-Klassen generiert werden.

Abbildung 8.3 illustriert den Ablauf einer Anfrage. Zentrale Bearbeitungskomponente ist der Dispatcher, der Anfragen annimmt und verarbeitet. Die einzelnen Schritte der

[1] Aus „kosmetischen“ Gründen können einzelne Objekt-IDs auch in Textform, z. B. `myhomepage` angegeben werden – diese werden dann vom System in die entsprechende Objekt-ID aufgelöst. Zudem können noch default-Familie und default-Methode definiert werden, so dass eine vereinfachte Anfrage-URL beispielweise `http://www.my-appl.de/de/761235.html` lautet, die dann vom System entsprechend aufgelöst wird.

[2] `http://javacc.dev.java.net/`

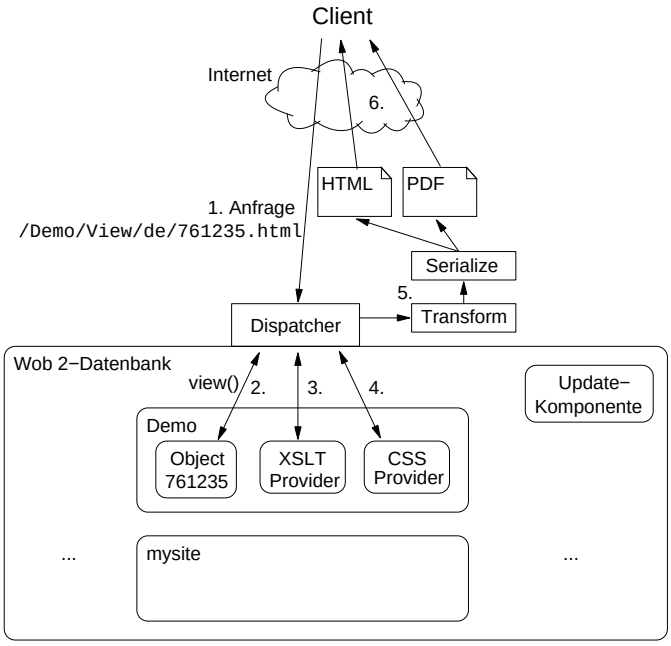


Abbildung 8.3.: Ablauf einer Anfrage

Anfragebearbeitung sind folgendermaßen:

1. Der Client stellt eine Anfrage an Wob 2 über die URL.
2. Der Dispatcher ruft die Methode `view()` des Objekts 761235 aus der Familie `Demo` auf und erhält von diesem das Ergebnis (in der Zwischensprache).
3. Der Dispatcher fordert von dem speziellen Objekt `XSLTProvider` der Familie `Demo` ein XSLT-Stylesheet an.
4. Der Dispatcher fordert von dem speziellen Objekt `CSSProvider` der Familie `Demo` ein CSS-Stylesheet an.
5. Der Dispatcher transformiert das Ergebnis von `761235.view()` mit dem XSLT-Stylesheet.
6. Der Dispatcher schickt dieses Ergebnis und das CSS-Stylesheet an den Client.

Dabei haben alle Komponenten (also Objekte und Provider) Zugriff auf die Übergabeparameter. Updates werden über eine getrennte Komponente durchgeführt, mehr dazu in Abschnitt 8.3.

8.2. Persistenz

Prinzipiell kann das W2OM in nahezu alle Datenbank-Datenmodelle abgebildet werden, wobei es jeweils ein geeignetes Mapping zu finden gilt. Im Folgenden werden hierzu einige Überlegungen präsentiert; da die physische Speicherung jedoch nicht im Fokus der vorliegenden Arbeit stand, sei hierzu auf die verschiedene Ansätze in der Literatur verwiesen, einen Überblick liefert z. B. die Diplomarbeit von Pützfeld [Püt01] oder die Arbeiten von Bourret (`http://www.rpbourret.com/xml/XMLDBLinks.htm`).

Wie in Abschnitt 6.3.1 spezifiziert, können Objekte im Datenbaum aus der W2L mit verschiedenen Funktionen (`getObject`, `getChildren`, `getParent`, ...) abgerufen werden. Bei einem Mapping müssen diese Funktionen auf das dahinterliegende Modell abgebildet werden (können).

Für das im Kontext der vorliegenden Arbeit entwickelte Objektmodell liegt natürlich eine Verwendung eines Datenbankmodells nahe, welches hierarchische Strukturen unterstützt (wie beispielsweise objektorientierte Datenbanken, native XML-Datenbanken oder hierarchische Datenbanken), zur Speicherung von semistrukturierten Daten eignen sich NF^2-Datenbankmodelle (non-first normal form). Allerdings können hierarchische Objektmodelle auch auf nicht-hierarchische Datenbankmodelle abgebildet werden, wobei dann die Hierarchieeigenschaften explizit nachgebildet werden müssen.

Datenobjekte mit ihren enthaltenen Feldobjekten können in kanonischer Weise in XML serialisiert werden und so beispielsweise in (XML-)Datenbanken persistent gespeichert werden. Dabei werden Feldobjekte als Unterelemente des dazugehörigen Datenobjektes gespeichert. Beispiel:

> Gegeben seien Klassen C1 und C2 mit Feldern F1, F2 und F3. Instanzen von Klasse C2 dürfen als Kindobjekte von Instanzen der Klasse C1 auftreten:

```
class C1 {
  property allowedChildren [C2];
  String F1;
  Number F2;
}
class C2 {
  String F3;
}
```

> Die Serialisierung eines Beispielobjekts, bei dem Unterobjekte in einer Baumhierarchie gespeichert werden, sieht folgendermaßen aus:

```
<C1 id="42">
  <F1>Feld 1</F1>
  <F2>4711</F2>
```

```
  <C2 id="55">
    <F3>Feld 3a</F3>
  </C2>
  <C2 id="56">
    <F3>Feld 3b</F3>
  </C2>
</C1>
```

Alternativ können aggregierte Objekte nicht geschachtelt, sondern separat gespeichert werden; in diesem Falle sind parent und/oder child-Verknüpfungen nötig:

```
<C1 id="42" children="55 56">
  <F1>Feld 1</F1>
  <F2>4711</F2>
</C1>
<C2 id="55" parent="42">
  <F3>Feld 3a</F3>
</C2>
<C2 id="56" parent="42">
  <F3>Feld 3b</F3>
</C2>
```

Diese alternative Variante ist zwar nicht ganz so gut „lesbar“ in dem Sinne, dass die parent/children-Verknüpfungen zunächst aufgelöst werden müssen, jedoch lässt sich eine solche Struktur einfacher beispielsweise in relationale Datenschemata abbilden (Im Beispiel könnte man jeweils eine Tabelle für `C1`-Objekte und für `C2`-Objekte verwenden).

Je nach eingesetztem Datenbank-Backend kann eine der beiden Varianten gewählt werden.

8.3. Eingabekomponente: Update-Sequenzen

Wie bereits in Abschnitt 4.4.1 diskutiert, ist eine Konsequenz der Seiteneffektfreiheit der W2L-Sprache, dass sie nicht verändernd auf den Datenbaum zugreifen oder gar neue Daten/Objekte erzeugen kann. Da natürlich auch Änderungen an den Daten vorgenommen werden sollen bzw. müssen, können Updates in einer separaten, von der Ausgabe getrennten, Anfrage durchgeführt werden (*Update-Sequenzen* genannt). Diese Update-Sequenzen werden in XML formuliert und können demnach einfach mit der W2L generiert werden. Prinzipiell ist dabei der Ablauf folgendermaßen: Die in der Eingabemaske vorgenommenen Eingaben werden an Wob 2 übermittelt, dort in Update-Sequenzen (in

XML) transformiert und damit ein Datenbank-Update ausgeführt. Im Folgenden wird erläutert, wie Änderungen an der Datenbank mit Update-Sequenzen ausgedrückt werden.

Einfügungen

Datenobjekte werden mit dem Element `insert` eingefügt, wobei die Attribute `class` und `parent` die Klasse resp. das Vaterobjekt spezifizieren. `class` muss dabei zwingend angegeben werden, `parent` ist optional – fehlt es, so wird ein neues Objekt dieser Klasse ohne Vaterobjekt eingefügt, ist `parent` angegeben, wird ein neues Objekt der spezifizierten Klasse unterhalb des Vaterobjekts eingefügt, wobei die Familie der des Vaters entspricht. Die Familie ist dabei fest (über den URL-Parameter der Anfrage) spezifiziert. Auf die Reihenfolge der Objekte (s. Abschnitt 4.6.2) kann Einfluss genommen werden, indem (optional) die Position angegeben wird: `position="2"` fügt das Objekt als zweites Kindobjekt ein, `position-after="99"` fügt das Objekt nach dem Objekt mit der ID 99 ein, sofern vorhanden (analog: `position-before`). Fehlen die `position`-Attribute, wird das Objekt am Ende eingefügt.

Felder des Datenobjekts werden über ein `field`-Element, das den Namen der Feldklasse als `name`-Attribut enthält, spezifiziert. Da Feldobjekte in 1NF sind, können sie kanonisch innerhalb eines XML-Elements eingesetzt werden. Wie in Kapitel 5 erläutert, können *Feld*objekte alleine (d. h. außerhalb eines Datenobjekts) zwar temporär instanziiert, aber nicht persistent gespeichert werden; somit entfällt eine separate Syntax für Einfügungen.

Das System führt bei Einfügungen eine (Typ-)Überprüfung auf Basis der Klassendeklaration durch; überprüft wird dabei, ob die entsprechenden Felder in der Klasse vorhanden sind, ob die übermittelten Daten mit dem entsprechenden Typ des Feldes übereinstimmen und ob Familie, Klasse und Vaterobjekt existieren. Des Weiteren wird ggf. überprüft, ob eventuell spezifizierte strukturelle Restriktionen (s. Abschnitt 4.6.3) verletzt werden.

Das nachstehende Listing zeigt ein Beispiel einer Einfügung: hier wird ein neues Objekt der Klasse `Text` unter das Vaterobjekt mit der ID 99 als erstes Kindobjekt eingefügt (vgl. hierzu auch Abschnitt 6.1 zum Thema Default-Konstruktoren)

```
<insert class="Text" parent="99" position="1">
  <field name="title">Ganz neuer Titel</field>
  <field name="body"><p>...mit neuem Rumpf</p></field>
</insert>
```

Löschungen

Das Element `delete` beschreibt Löschungen, wobei im erforderlichen Attribut `ids` eine (durch Leerzeichen getrennte) Liste von zu löschenden Datenobjekt-IDs steht. Das

folgende Listing illustriert, wie ein bzw. mehrere Datenobjekte gelöscht werden können.

```
<delete ids="22"/>
<delete ids="37 49"/>
```

Änderungen

Änderungen werden mittels des `update`-Elements spezifiziert, welches die ID des zu ändernden Datenobjekts als erforderliches Attribut haben muss. Auch hier werden dieselben (Typ-)Überprüfungen wie im Falle der Einfügungen vorgenommen. Nachfolgend ein Beispiel einer Änderung.

```
<update id="42">
  <field name="title">Ein anderer Titel</field>
  <field name="body">Dies ist
  der
  Rumpf.
  </field>
</update>
```

Kopieren und Verschieben

Obwohl Kopieren und Verschieben theoretisch mit den obigen Funktionen simuliert werden können (über Einfügen bzw. Löschen+Einfügen), sind sie u. a. aus Gründen der Benutzerfreundlichkeit („syntactic sugar“) zusätzlich implementiert worden. Die move-Operation ist darüber hinaus zur Bewahrung der ID wichtig, damit Referenzen auf das Objekt persistent bleiben. Eine „Verschiebung“ via Einfügen+Löschen würde eine neue ID generieren. Auch hier kann, analog zu Einfügungen, Einfluss auf die Position in der Hierarchie genommen werden. Die folgenden Beispiele kopieren bzw. verschieben das Objekt mit der ID `97` unter das Vaterobjekt mit der ID `99` und hinter das Objekt mit der ID `72`:

```
<copy id="97" target="99" position-after="72"/>
```

```
<move id="97" target="99" position-after="72"/>
```

Update-Konflikte

Die Update-Operationen werden (im Sinne des ACID-Paradigmas) atomar, d. h. simultan durchgeführt. Da Update-Operationen sequentiell durchgeführt werden, können theoretisch denkbare Konflikte (in einer Update-Sequenz wird beispielsweise ein Datenobjekt zweimal geändert) aufgelöst werden. Zudem kann eine Warnung ausgegeben werden.

8.4. Ausgabekomponente: Zwischensprache

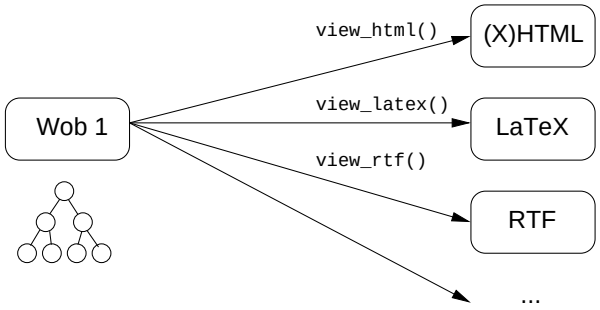


Abbildung 8.4.: Wob 1: Ausgabeformate und -templates

Ein wichtiges Kernkonzept von CMS ist es, Inhalte in verschiedenen Ausgabeformaten[3] ausgeben zu können (Single source, multiple target-Prinzip). In gängigen CMS (so auch in Wob 1) müssen hierzu jedoch immer speziell auf das gewünschte Format angepasste Templates geschrieben werden, was Abbildung 8.4 illustriert. Bei m Ausgabetemplates[4] und n Ausgabeformaten müssen also insgesamt $m * n$ Templates geschrieben werden. Abbildung 8.5 zeigt zwei (Wob 1-) Templates exemplarisch: hier zeigt sich, dass die Vermischung von Formatierung (`<h1>`, `\large`), Inhalten (`Anstehende Termine:`) und Programmlogik (`{{*termin}}`) die Templates unübersichtlich macht und Redundanzen beinhaltet. Diese Redundanz zieht einen erhöhten Aufwand bei der Eingabe, aber auch bei der späteren Pflege nach sich; für eine Änderung eines Textes (beispielsweise „Anstehende Termine“ in „Terminliste“) müssen sämtliche Templates geändert werden. Zudem wird (unnötigerweise) die Programmlogik dupliziert.

```
<h1>{{*title}}:</h1>
Anstehende Termine:
<ul>
 <li> Termin: {{*termin}}
</ul>
```

```
{\large {{*title}}:}
Anstehende Termine:
\begin{itemize}
  \item Termin: {{*termin}}
\end{itemize}
```

Abbildung 8.5.: Ausgabetemplates
a) für HTML b) für LaTeX

Im Rahmen der vorliegenden Arbeit wurde nun das Konzept einer *medienneutralen Zwischensprache* entwickelt; die Templates erzeugen hierbei eine Ausgabe, die *un-*

[3] Z. B.: XML, (X)HTML, LaTeX, RTF, PDF, ...
[4] Für verschiedene Ansichten, also z. B. `view` für die normale Ansicht oder `summary` für eine Kurzübersicht.

abhängig vom gewünschten Ausgabeformat ist. Diese wird dann in einem weiteren Schritt, z. B. über verschiedene XSLT-Stylesheets, in das entsprechende Zielformat transformiert. Durch den Einsatz einer solchen Zwischensprache müssen nun nicht mehr $m * n$, sondern nur noch $m + n$ Ausgaben implementiert werden, wie in Abbildung 8.6 gezeigt. Zudem werden Inhalte von Formatierung getrennt, d. h. das „Separation of Concerns"-Paradigma (vgl. Abschnitt 3.1.1) kommt zum Tragen.

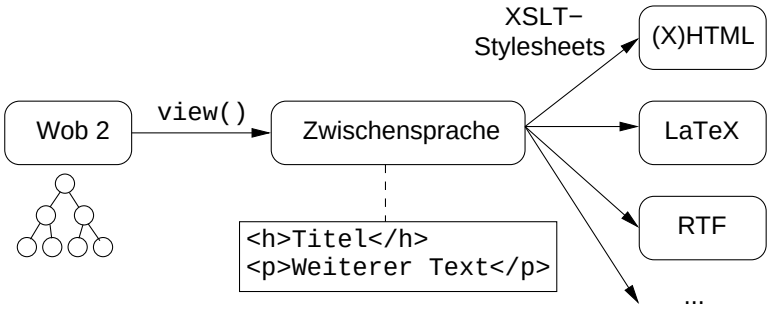


Abbildung 8.6.: Zwischensprache

Durch diesen Ansatz ist normalerweise pro Klasse immer nur ein einziges `view()`-Template erforderlich, deren Ausgabe in der Zwischensprache erfolgt. Pro Zielformat wird dann noch ein XSLT-Stylesheet benötigt, welches aber lediglich einmal erstellt werden muss. Für den Fall, dass neue Klassen erstellt werden, muss kein neues Stylesheet erstellt werden, da die Zwischensprache in diesem Fall nicht verändert wird (das „Vokabular" bleibt gleich). Dies entspricht dem „inkrementellen" Gedanken der Arbeit insofern, dass redundanzfrei immer nur Änderungen spezifiziert werden müssen. Umgekehrt gilt: Wenn das Design einer Anwendung/Familie geändert werden soll, muss lediglich das Stylesheet verändert werden (ggf. jeweils pro Zielsprache), die Klassen müssen jedoch nicht geändert werden. Das gleiche gilt für das Hinzufügen neuer Zielsprachen.

Die Zwischensprache ist insoweit flexibel, als sie dynamisch (pro Familie und/oder Klasse) erweitert werden kann, um etwa neue Elemente hinzuzufügen. Dann jedoch müssen sowohl die Klassen als auch die Stylesheets angepasst werden. Anders gesagt: Das Stylesheet definiert die Zwischensprache.

Als Zwischensprache wird ein (erweiterbares) Derivat aus XHTML2[5] und Docbook[6] eingesetzt werden, da dies besonders geeignet erscheint, um die im Rahmen der geplanten (publikationsorientierten) Anwendungen auftretenden Fälle beschreiben zu können.

[5] `http://www.w3.org/TR/xhtml2/`
[6] `http://www.oasis-open.org/docbook/specs`

8.5. Generierung von Eingabemasken

Eine weiterer Aspekt, der für ein CMS sinnvoll erscheint und im Rahmen dieser Arbeit untersucht wurde, sind erweiterte Funktionalitäten bei der Dateneingabe bzw. -pflege durch die automatische Generierung einer Datenpflegemaske (oder: graphische Benutzerschnittstelle, engl: graphical user interface, GUI) aus dem Datenschema. Dieser Abschnitt gibt im Wesentlichen die Ergebnisse von [LLK04] wieder.

8.5.1. Motivation

Üblicherweise müssen bei Datenbankanwendungen wie auch bei CMS zusätzlich zu der eigentlichen Applikation sowohl das Datenschema als auch eine Datenpflegemaske zum Einfügen, Bearbeiten bzw. Löschen von Daten entwickelt werden. Die Idee, die im Rahmen der vorliegenden Arbeit entstand, war, *automatisch* aus dem Datenschema Eingabemasken zu generieren, um so den Erstellungsaufwand von Applikationen zu verringern.

Dass dieser doppelte Aufwand nicht unerheblich ist, zeigt u. a. eine Untersuchung in [MR92]:

> „Statistically, an average of 48% of the written code and nearly half of the design, implementation and maintenance time during development is associated with GUI creation and maintenance.“

Für RDBMS existieren bereits verschiedene Systeme, die dies ermöglichen (z. B. in Microsoft Access oder phpMyAdmin[7] für MySQL), jedoch ist der Einsatz im Kontext semistrukturierter Daten ungleich komplexer: Durch die heterogene Daten- bzw. Schema-Struktur (vgl. Abschnitt 2.1) müssen die GUIs flexibel aufgebaut sein, zudem müssen wegen der hierarchischen Eigenschaften ggf. Unterbäume dynamisch zur Laufzeit generiert werden. Es ist also notwendig, (baumartige) Strukturen dynamisch erzeugen und ggf. auch wieder löschen zu können.

Web-basierte Formulare (XHTML, WML, ...), die zumeist im Kontext von CMS eingesetzt werden, lassen oftmals wünschenswerte Flexibilität vermissen, die nur mit anderen Technologien (z. B. Java Swing[8]) erreicht werden kann. Beispielsweise ist die Anzahl der Eingabekomponenten (*Widgets*) beschränkt und das Layout ist in der Regel statisch[9].

Kernpunkt des hier entwickelten Ansatzes ist es, direkt aus dem W2OM-Modell automatisch eine Datenpflegemaske generieren zu können. Die Eingabemaske erzeugt aus den Eingaben dann wiederum die entsprechenden Update-Sequenzen (s. Abschnitt 8.3), die zur Änderung des Daten(bank)zustandes nötig sind.

[7] `http://www.phpmyadmin.net`

[8] eine in Java enthaltene Klassenbibliothek zur Erstellung von GUIs

[9] Mehr Dynamik kann durch den Einsatz von Skriptsprachen wie JavaScript erreicht werden, jedoch ist dies aus verschiedenen Gründen wie z. B. Sicherheitsaspekten, oft nicht angebracht oder gewünscht.

Nach einer Beschreibung der benötigten Grundlagen wird der Ansatz dieser GUI-Generierung im Folgenden vorgestellt.

8.5.2. Grundlagen

Arten von Client/Server-GUIs

Man kann Clients in Client/Server-Systemen grob in *Fat-* und *Thin clients* kategorisieren:

Fat clients bestehen aus einer komplexen Anwendung, die einen gewissen Anteil an (Business-)Logik enthält, wie z. B. größere Java-Applikationen oder (native) Windows/Linux-Programme. Thin clients hingegen sind zumeist nur zur Datendarstellung konzipiert und beinhalten keine oder kaum Anwendungslogik, ein Beispiel eines thin clients im CMS-Kontext ist ein Webbrowser. Die Vorteile von Thin clients sind:

1. Größere Plattformunabhängigkeit.
2. Geringe Hard- und Softwareanforderungen.
3. Wenig Administrationsaufwand (da beispielsweise Browser in der Regel bereits installiert sind).

Dem gegenüber stehen die Nachteile: zum einen fehlende Flexibilität und (vergleichsweise) wenig Widgets, zum anderen schlechte Antwortzeiten, da beispielsweise Konsistenzchecks auf Grund fehlender Anwendungslogik erst an den Server übertragen und dort verarbeitet werden müssen.

Bei Fat clients verhält es sich entgegengesetzt, d. h. mehr Flexibilität und Performanz in Hinblick auf Antwortzeiten erkauft man sich mit eingeschränkter Plattformunabhängigkeit und höheren Anforderungen an Hard- und Software sowie Adminstrationsaufwand. Des Weiteren existieren noch Ansätze wie Rich thin clients oder Smart clients (z. B. Java WebStart), die versuchen, Vorteile aus beiden Ansätzen zu kombinieren.

Der im Rahmen der vorliegenden Arbeit entwickelte Ansatz setzt dabei das thin client-Prinzip um.

XML Persistence in Java

Der hier entwickelte Ansatz nutzt das so genannte „XML Persistence“-Konzept des Java Development Kits (JDK), welches es ermöglicht, beliebige Java Beans in XML zu serialisieren und später wieder zu deserialisieren[10] (siehe u. a. [Mil02]). Mit diesem Mechanismus können komplette Swing-Bäume, deren Klassen größtenteils Beans sind, serialisiert und später wieder deserialisiert werden.

[10] Die entsprechenden Klassen heißen `java.beans.XMLEncoder` bzw. `java.beans.XMLDecoder`.

Das nachfolgende Listing zeigt ein einfaches Beispiel, in dem (in Zeilen 1-3) zunächst eine Instanz des XMLEncoders erzeugt wird, mit welcher dann (in Zeile 4) ein neu erzeugter JButton[11] in die Datei `button.xml` serialisiert wird. Das Ergebnis ist in Listing 8.1 zu sehen.

```
1 XMLEncoder e = new XMLEncoder(
2   new BufferedOutputStream(
3     new FileOutputStream("button.xml")));
4 e.writeObject(new JButton("I'm a JButton"));
5 e.close();
```

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.5.0_03" class="java.beans.XMLDecoder">
 <object class="javax.swing.JButton">
  <string>I'm a JButton</string>
 </object>
</java>
```

Listing 8.1: Datei `button.xml`

Unter Verwendung des XMLDecoders kann diese XML-Repräsentation wieder in eine Objektstruktur deserialisiert werden. Abbildung 8.7 illustriert dies.

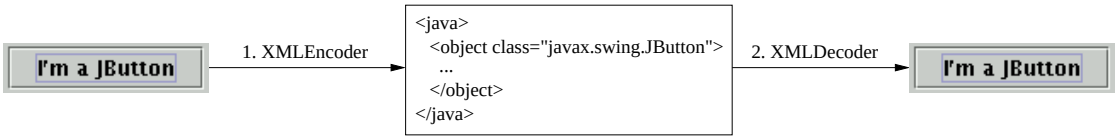


Abbildung 8.7.: Funktionsweise der XMLen/decoder-Klassen

8.5.3. Vom Datenschema zur Java Swing GUI

Wie in Abschnitt 8.1 beschrieben, können die expandierten Familien und der Zustandsanteil (Datenschema) in XML exportiert werden. Dadurch ist die Generierung von Eingabemasken in einem XML-Dialekt (XHTML, WML, ...) leicht mit einem XSLT-Stylesheet machbar, wie in Abbildung 8.8 illustriert. Wie in Abschnitt 8.5.1 erwähnt, lassen diese Sprachen jedoch in Hinblick auf Flexibilität und Komfort zu wünschen übrig.

Neu ist bei dem hier vorgestellten Ansatz, dass auch komplexe, dynamische Java-Swing-GUIs erzeugt werden können (der mit $XSLT_3$ beschriftete Pfeil in Abbildung 8.8).

[11] Eine Swing-Klasse für Button-Widgets.

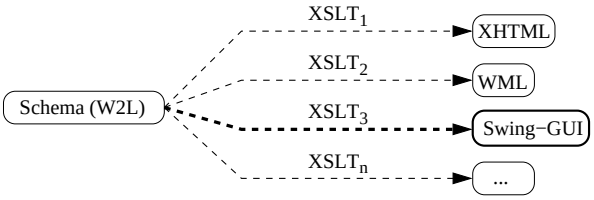


Abbildung 8.8.: GUI-Generierung

Die grundlegende Idee ist, die serialisierte Darstellung (vgl. Abschnitt 8.5.2) des GUI per XSLT-Stylesheet direkt aus dem Datenschema zu erzeugen, wie in Abbildung 8.9 am Beispiel eines Buttons gezeigt.

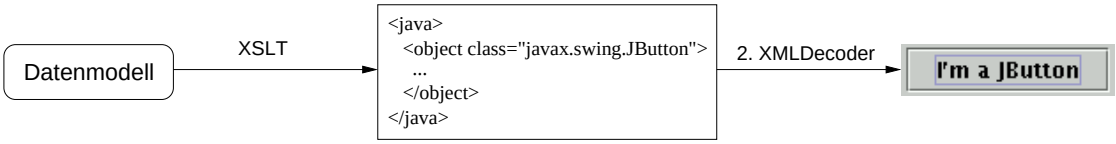


Abbildung 8.9.: Vom Datenschema zur Swing GUI

Von Vorteil ist es, dass nun nur noch diese serialisierte Darstellung zum Client übertragen und dort wieder deserialisiert werden muss, auf der Clientseite ist somit nur ein sehr schlankes Programm (also ein thin client) vonnöten, um das GUI anzeigen zu können.

GUI-Komponenten

Beim Design eines Java Swing GUIs ist es sinnvoll, das GUI in verschiedene miteinander kommunizierende Komponenten aufzuteilen. Beispielsweise können mehrere Swing-Elemente für ein Feldobjekt existieren (ein `JLabel` als Beschriftung, eine `JBorder` als Umrandung und das Feldobjekt in einem `JTextField`). Im hier entwickelten Ansatz werden drei Komponenten verwendet:

- der eigentliche Inhalt der Feldobjekte,
- die Swing GUI Elemente, in einem Swing-Baum gespeichert, und
- Controller-Objekte, ebenfalls in einem Baum organisiert, die die Strukturen verwalten und synchronisieren.

Dies entspricht einer Anwendung des Model-View-Controller-Entwurfsmusters (vgl. [GHJV95]), das somit auch eine Form des Separation of Concerns-Paradigmas ist, da Inhalt, Formatierung und Logik voneinander getrennt werden.

Die Datentypen der einzelnen Feldobjekte wie `integer` oder `string` können dabei einfach auf Eingabefelder wie `JTextField` übertragen werden. Ggf. können hier spezielle Eingabefelder verwendet werden, die z. B. nur numerische Eingaben zulassen. Des Weiteren kann beispielweise für ein Attribut vom Typ `date` ein Kalenderwidget generiert werden, auf dem komfortabel ein Tag ausgewählt werden kann. In Standard-HTML-Formularen wäre dies nicht möglich. Zudem ist es möglich, Daten- bzw. Feldobjekte mit (externen) Annotationen zu versehen, um in Einzelfällen spezielle Widgets zu generieren.

Abbildung 8.10 zeigt einen Screenshot eines GUI mit komplexen Widgets, die in dieser Form in HTML-Formularen o. ä. nicht umsetzbar sind.

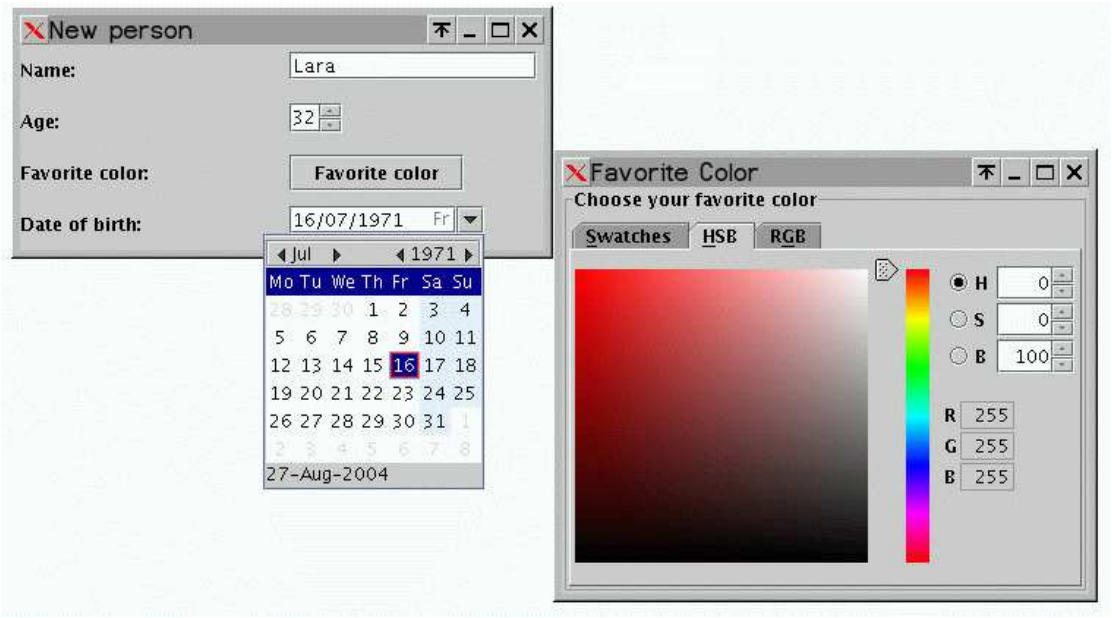


Abbildung 8.10.: GUI mit komplexen Widgets

Besonderheiten im Hinblick auf die Semistrukturiertheit

Für den Fall, dass nur *ein* Datenobjekt editiert (bzw. eine Instanz einer Datenklasse erstellt) werden soll, ist bei der Generierung lediglich auf die Reihenfolge der einzelnen Felder zu achten. Hierbei wird die in Abschnitt 4.6.2 beschriebene, durch die Klassendeklaration vorgegebene, implizite Reihenfolge verwendet.

Komplizierter wird es, wenn mehrere Datenobjekte gleichzeitig editiert werden sollen, beispielweise eine Textklasse mit mehreren Bildern. In diesem Fall werden die zu editierenden Datenobjekte in dem GUI einfach in der entsprechenden impliziten Reihenfolge hintereinander gehängt (s. ebenfalls Abschnitt 4.6.2).

Um an bestehende (Daten-) Objekte weitere Kindobjekte anhängen zu können, stellt das GUI hierzu geeignete Funktionalität in Form von „Hinzufügen“- und „Löschen“-Buttons bereit, wobei die in Abschnitt 4.6.3 beschriebenen Eigenschaften (`allowChildren` und `restrictToParent`) beachtet werden. Beispielweise würden im Falle einer Datenstruktur wie im zusammenfassenden Beispiel auf Seite 75 bei einem Objekt der Klassen `Text`, `Calendar`, `Appointment` und `Link` entsprechende „Hinzufügen“-Buttons für alle in der Familie enthaltenen Klassen außer `Appointment` (da Instanzen dieser Klasse auf `Calendar`-Vaterobjekte beschränkt sind) erscheinen, während bei Objekten der Klasse `Linklist` nur ein „Hinzufügen: Link“ generiert würde.

Schablonen

Da die Struktur des GUI sich zur Laufzeit ändern kann, können nicht alle Komponenten zur Generierungszeit generiert werden. Beispielsweise ist die Anzahl der Bilder, die auf einer Webseite eingebunden werden sollen, dem System a priori nicht bekannt. Um nicht alle Komponenten erst zur Laufzeit berechnen zu müssen, werden *Schablonen*, welche zur Generierungszeit berechnet werden, benutzt und zur Laufzeit je nach Bedarf (ggf. mehrfach) instanziiert. Eine Schablone repräsentiert ein Datenobjekt und besteht aus:

Modell: ist eine Referenz auf das Datenobjekt, das editiert wird,

Swing (Unter-)Baum: repräsentiert den View des MVC-Modells, z. B. ein `JTextField` für jedes enthaltene Feldobjekte oder komplexere Anordnungen von Swing Komponenten,

Controller: regelt die Geschäftslogik des Elements, d. h. der Controller hat Referenzen auf das Datenobjekt und die Swing-Komponenten. Zudem behandelt er Ereignisse wie Button-Klicks o. ä.

Zur Generierungszeit wird pro Datenklasse eine Schablone generiert und in einer Schablonenbibliothek abgelegt. Zur Laufzeit wird, je nach Bedarf, die entsprechende Schablone aus der Bibliothek instanziiert. Schablonen werden auch dann generiert, wenn der Benutzer auf einen „Hinzufügen“-Knopf klickt. In diesem Fall wird ein neues Datenobjekt mit seinen Feldobjekten erzeugt und angehängt. Wenn der „Löschen“-Button angeklickt wird, werden die entsprechenden Komponenten gelöscht.

8.5.4. Verwandte Arbeiten

Zu Beginn der 1990er Jahre wurde verstärkt im Bereich der modellbasierten GUI-Generierung geforscht (vgl. u. a. [Bal93], [Pue97], [EVP01]), was sich aber als zu unflexibel herausgestellt hat, wie beispielsweise in [Pue96] beschrieben. Der Unterschied zum hier gewählten Ansatz ist, dass bei der modellbasierten Generierung die einzelnen

Modelle explizit vom Entwickler spezifiziert werden müssen, während im Rahmen der hier vorgestellten Arbeit das Datenschema quasi als GUI-Modell, also implizit, verwendet wird und somit der zusätzliche Aufwand der Modellerstellung entfällt.

In [LC01] wird ein XML-basierter Ansatz zur GUI-Generierung vorgestellt, welcher aber ebenfalls eine explizite Beschreibung des gewünschten GUI benötigt.

Kohlhaas untersuchte in [Koh03], inwiefern der modellbasierte Ansatz verwendet werden kann, um Datenpflegemasken aus (manuell erstellten) Modellen unter Zuhilfenahme des Mozilla-Frameworks[12] generieren zu können.

Mancke befasst sich in [Man03] sowie [RMC04] mit der Fragestellung, wie sich Benutzerschnittstellen auf darstellungsunabhängige Weise definieren und an die Eigenschaften unterschiedlicher Endgeräte anpassen lassen. Der Fokus lag dabei auf der abstrakten Beschreibung von Benutzerschnittstellen unter der Zuhilfenahme von Ontologien und darauf, wie daraus GUIs auf verschiedenen Endgeräten effizient und benutzerfreundlich generiert werden können. Auch hier müssen jedoch die Ontologien und Modelle manuell erstellt werden.

Alternativ zu den XMLEn/Decoder-Klassen existieren die beiden Ansätze SwiXml[13] sowie SwingML[14] (Beispiele s. nachfolgende Listings), die im Prinzip nur eine kompaktere und „schönere“, d. h. für menschliche Leser lesbarere Syntax verwenden. Diese wären leicht in das hier entwickelte Konzept zu integrieren, würden aber keine neue Funktionalität liefern.

```
<!-- SwiXml Beispiel -->
<frame size="640,480" title="Hello SWIXML World">
  <panel constraints="BorderLayout.CENTER">
    <label LabelFor="tf" Foreground="blue" text="Hello World!"/>
    <textfield id="tf" Columns="20" Text="Swixml"/>
    <button Text="Click Here" Action="submit"/>
  </panel>
</frame>
```

```
<!-- SwingML Beispiel -->
<panel orientation="center">
  <label name="helloLabel" text="Hello"/>
  <textfield
     name="worldField"
     text="Waiting for event" cols="20"/>
  <button name="worldButton" Text="Click Here">
    <listener event="ActionListener.actionPerformed">
      <action
```

[12] http://www.mozilla.org
[13] http://www.swixml.org
[14] http://swingml.sf.net

```
        component="worldField"
        method="setText"
        types="String" values="World"/>
    </listener>
  </button>
</panel
```

8.5.5. Fazit und Ausblick

In diesem Abschnitt wurde ein Ansatz vorgestellt, um aus W2OM-Schemata automatisiert Dateneingabe/Pflegemasken zu generieren. Die Vorteile liegen zum einen in der Arbeitsersparnis und zum anderen im thin client-Prinzip, d. h. es können mit wenig Aufwand und ohne viel Installationsarbeit (auch) komplexe Datenpflegemasken erstellt und verwendet werden.

Anzumerken ist, dass nur GUIs zur *Datenpflege* generiert werden können, da die Integration von beliebig komplexer Business-Logik einige der eben erwähnten Vorteile wieder zunichte machen würde. Beispielsweise müssten dann (anders als bei thin clients) erheblich mehr Daten in Form von Programmen übertragen werden, zudem sind nicht alle Aspekte automatisch generierbar, so dass manuelle Anpassungen nötig wären und so der Vorteil der Automatisierung wegfallen würde.

Darüber hinaus ist der momentane Stand der Implementierung als „proof-of-concept“ zu verstehen; die generierten GUIs können noch in Hinblick auf Ergonomie und Flexibilität optimiert werden.

9. Zusammenfassung und Ausblick

9.1. Zusammenfassung

In den Anfängen des WWW bestanden Websites zumeist aus statischen HTML-Dateien, die untereinander verlinkt waren. Mit zunehmender Größe sowohl des gesamten WWW als auch der einzelnen Websites wurde dieser Ansatz jedoch von Systemen abgelöst, die die ausgelieferten HTML-Webseiten dynamisch, z. B. aus einer Datenbank, generieren. Auf diese Weise entstanden z. T. komplexe Internet-Informationssysteme, Content Management Systeme genannt, die die eigentlichen Inhalte (Texte, Bilder, Videos, ...) verwalten und Entwicklern sowie Redakteuren die Erstellung und Pflege der Websites deutlich erleichtern. Das Wob 1-System (s. Abschnitt 3.3) war eines der ersten Systeme dieser Art. Die in aller Regel hierarchisch strukturierten Daten enthalten zumeist Substrukturen und sollen, je nach Anfrage, für verschiedene Endgeräte in unterschiedlichen Formaten und Granularitäten ausgeliefert werden können.

Dies stellt einige besondere Anforderungen an das System und insbesondere an das dahinterliegende Datenmodell, das diese Daten möglichst strukturtreu abbilden soll. Ein geeignetes Datenmodell ermöglicht es dem Entwickler, Informationsangebote einfach, redundanzfrei und unter Verwendung bestehender Informationen (Re-Use) erstellen zu können. Auch die oben erwähnte Ausgabe in unterschiedlichen Formaten und Granularitäten muss in adäquater Form unterstützt werden.

Eine Analyse bestehender CMS im Vorfeld der Arbeit zeigte jedoch, dass bestehende Systeme und deren Datenmodelle diesbezüglich unflexibel und, gerade im Hinblick auf Wiederverwendbarkeit und Redundanzfreiheit, unzureichend sind. So ist die Erstellung neuer Anwendungen aus bestehenden sowie die Ausgabe in unterschiedlichen Formaten mit viel Redundanz verbunden. Dies liegt u. a. darin begründet, dass bestehende Systeme sehr stark designorientiert sowie dokumentenzentriert und nicht darauf ausgelegt sind, die Daten in einer anderen als der durch die Primärstruktur vorgegebenen Form auszugeben.

Daher wurde im Rahmen der vorliegenden Arbeit ein objektorientiertes Datenmodell konzipiert, das es ermöglicht, die hierarchischen Strukturen mit enthaltenen Substrukturen feingranular zu modellieren sowie Anwendungen weitgehend redundanzfrei zu spezifizieren. Die Kernidee des Modells ist die Verschmelzung unterschiedlicher Aspekte aus verschiedenen Forschungsgebieten: Insbesondere die Verbindung objektorientierter Konzepte mit funktionalen Ansätzen im Kontext von CMS-Datenmodellen schafft dabei ein in dieser Form neues und sehr gut zur Modellierung semistrukturierter Daten geeignetes

Modell.

Die wichtigsten Aspekte dieses Modells sind:

- Ein Vererbungskonzept, mit dem statt einzelner Klassen ganze Klassenstrukturen (Familien) abgeleitet und spezialisiert werden können. Auf diese Weise können neue Anwendungen weitestgehend redundanzfrei spezifiziert werden, wobei auch Änderungen wie etwa Bugfixes an den ursprünglichen Klassen bis in die neuen Anwendungen vererbt werden.
- Eine Verfeinerung des Objektansatzes, bei dem die Attribute der Datenobjekte in eigenständige Feldobjekte „ausgelagert“ werden, die sich selbst darstellen können. Dies ermöglicht ebenfalls eine knappe, redundanzfreie Modellierung.
- Eine funktionale Templatesprache, mit der textuelle Ausgaben einfach und (durch die Seiteneffektfreiheit) sicher erzeugt werden können. Hiermit wird eine kompakte, aber dennoch berechnungsuniverselle Sprache zur Verfügung gestellt, mit der ein Großteil der Aufgaben der Datenausgabe im CMS-Bereich effizient gelöst werden können. Ein statisches Typsystem mit einem XML-Datentyp unterstützt diese Ausgabe ebenfalls, da durch den Einsatz einer medienneutralen Zwischensprache die Ausgabe in verschiedenen Formaten deutlich erleichtert wird (z. B. durch den Einsatz von XSLT).

Insgesamt wird durch das W2OM ein solides Datenmodell als einheitliche Datenabstraktionsschicht bereitgestellt, welches durch einen hohen Grad von Re-Use und Redundanzvermeidung das Erstellen komplexer CMS-Anwendungen ermöglicht; mit nur wenig Aufwand können derartige Anwendungen, auch aus bestehenden, erstellt werden. Einem Entwickler von CMS-Anwendungen wird die Arbeit so erleichtert, da das Modell eine adäquate Modellierung der Daten ermöglicht.

Die Schwierigkeit bei der Konzeption bestand darin, bestehende Ansätze aus anderen Communities aufzugreifen, zu erweitern und in nichttrivialer Weise in einem neuen Kontext zu kombinieren. Auf diese Weise können die identifizierten Anforderungen an ein CMS-Datenmodell erfüllt werden. Einige Aspekte des Konzepts wurden prototypisch im Rahmen des Wob 2-Systems umgesetzt.

9.2. Ausblick

Die vorliegende Arbeit stellt einen soliden theoretischen Rahmen für den Entwurf und die Entwicklung von Content Manegement Systemen dar. Aufbauend auf dem entwickelten Datenmodell lassen sich die zu verwaltenden Daten sehr gut modellieren sowie konkrete Systeme implementieren. Darüber hinaus bieten die eingeführten Konzepte in verschiedenen Aspekten Möglichkeiten für daran anschließende theoretische Forschungsarbeiten, aber auch für konkrete praktische Realisierungen. Einige dieser Gesichtspunkte werden im Folgenden kurz beschrieben.

9.2.1. Datenmodell

Wie im Verlauf der Arbeit bereits an mehreren Stellen erwähnt, sind einige Aspekte als mögliche Erweiterungen denkbar. Ob bzw. inwieweit diese umgesetzt werden, wird sich anhand erster Erfahrungen mit dem prototypischen System ergeben.

- Die in Abschnitt 4.6.3 beschriebenen strukturellen Constraints könnten erweitert werden, um flexiblere Spezifikationen wie beispielsweise Schachtelungen, Alternativen oder die Angabe von Kardinalitäten („mindestens 2 Autoren, maximal 5 Autoren“, „maximal 2 Bilder“) zu erlauben. Eine Möglichkeit wäre, dies in einer BNF-ähnlichen Form spezifizieren zu können, also z. B.
  ```
  property allowedChildren [Autor[2..5],(ISBN|URL)]
  ```
  (Ein Buch-Element muss zwischen 2 und 5 Autor-Unterelemente und entweder ein ISBN- oder ein URL-Unterelement haben). Ein alternativer Ansatz hierzu wäre die separate Spezifikation von Integritätsbedingungen.

- Typinferenz: Typen von Methoden oder Feldern könnten inferiert werden (vgl. Abschnitt 2.3.3). Dies würde das Erstellen neuer Anwendungen bzw. Methoden erleichtern, da Typangaben in vielen Fällen nicht explizit deklariert werden müssten, das Typsystem aber trotzdem typsicher bliebe.

- Wie bereits am Ende von Abschnitt 4.5 erwähnt, könnte eine richtige Mehrfachvererbung umgesetzt werden, bei der eine Datenklasse mehrere Vaterklassen haben kann (z. B. `class LinkCalendar extends [Linklist Calendar]`), wobei hierbei der Nutzen gegen die damit verbundenen Probleme abgewägt werden muss.

- Eine Vererbung entlang des Datenbaums (Delegation) könnte implementiert werden (s. Abschnitt 4.2.1), wobei dann allerdings durch geeignete Maßnahmen die Typsicherheit zu garantieren wäre.

Von größer praktische Relevanz im Kontext von CMS sind Mehrsprachigkeit und Versionierung. Sie sind orthogonal zur bisher behandelten Datenmodellierung in dem Sinne, dass sie eine andere „Dimension“ der Daten darstellen: ein Feld hat, je nach Sprachauswahl oder gewünschter Version, verschiedene Werte. Dies kann und soll auch im W2OM umgesetzt werden.

Mehrsprachigkeit: Verschiedene Felder sollen, je nach Einstellung (von System bzw. Benutzer) in unterschiedlichen Sprachen ein- und ausgegeben werden können. Beispielsweise soll ein Titel in Deutsch oder Englisch ausgegeben werden können. Andere Felder wiederum bleiben davon unberührt (z. B. ein eingebettetes Bild). Andere Aspekte von Mehrsprachigkeit wie beispielsweise die unterschiedliche Formatierung von Datumsangaben können über XSLT-Stylesheets[1] realisiert werden.

[1] XSLT unterstützt verschiedene Konzept zur Lokalisierung.

Versionen: Auch die für eine zukünftige Version des Systems geplante Versionierungskomponente, mit der es möglich sein wird, verschiedene Versionen der Objekte zu verwalten, um beispielsweise auf alte Zustände zugreifen zu können, ist eine solche orthogonale Komponente. Ein erster Ansatz in dieser Richtung wurde in [Bar05] gezeigt.

9.2.2. Praktische Umsetzung

Ein wichtiges Stichwort in Kontext des praktischen Einsatzes ist das des Cachings, d. h. es ist zu untersuchen, inwiefern bzw. an welche Stellen bei der Anfragebearbeitung Daten zwischengespeichert werden können. Auf Grund der referenziellen Transparenz der W2L ist zu erwarten, dass hier ein effizientes Caching möglich ist, da (Teil-)Ausdrücke auch parallel ausgewertet werden können.

Des Weiteren muss der Punkt der Datenbankanbindung noch evaluiert werden; hier müssen geeignete Abbildungen (Mappings, vgl. Abschnitt 8.2) auf eine konkrete Datenbank entwickelt werden. Geplant ist auch ein „virtuelles Datenbank-API“, mit dem verschiedene Datenbankbackends transparent an das Wob 2-System angebunden werden können.

Mit dem Wob 2-System werden in der Arbeitsgruppe Internet-Informationssysteme der Abteilung III des Instituts für Informatik der Universität Bonn zunächst exemplarisch die folgenden Anwendungen umgesetzt werden:

Homepages: Grundlage für viele Anwendungen, unter anderem auch im Wob 1-System, ist eine Homepage-Anwendung, welche häufig verwendete Basisklassen wie `Text`, `Linklist`, `Sitemap` oder `Calendar` zur Verfügung stellt. Auch Wob 2 soll eine solche Anwendung zur Verfügung stellen; hiermit sind beispielsweise persönliche Homepages erstellbar, ohne besondere Vorkenntnisse von HTML oder ähnlichen Technologien haben zu müssen.

Hier kommt der Vorteil des inkrementellen Konzepts zum Tragen: beispielsweise könnte für ein Institut eine generische Anwendung erstellt werden, die dann jede Abteilung des Instituts ableiten und erweitern kann; trotzdem könnten Bugfixes o. ä. an der ursprünglichen Anwendung eingepflegt und an die neue Anwendungen vererbt werden.

Lehre: In Erweiterung des Wob 1-Angebots zur Unterstützung der Lehre und Lehrplanung sollen folgende Funktionalitäten angeboten werden:

- ein kommentiertes Vorlesungsverzeichnis.
- ein personalisierter Stundenplan.
- ein Vergabesystem für Übungen, Praktika und Seminare.

- ein Anmeldesystem zu Seminaren und Praktika, z. B. auch mit der Möglichkeit, Fragebögen an die Studenten zu Vorkenntnissen o. ä. zu erstellen.

Bei dieser Anwendung kommen insbesondere zwei Aspekte des W2OM zum Tragen: zum einen die Möglichkeit, alternative Sichten auf die Daten zu haben (wie personalisierter Stundenplan und Vorlesungsverzeichnis), zum anderen die veränderliche Granularität (beispielsweise im Stundenplan nur der Titel einer Veranstaltung, in einer Detailansicht Detailinformationen).

Dokumenten-/Folienverwaltung: Es soll möglich sein, z. B. Vorlesungsfoliensätze nicht nur zum Download anzubieten, sondern auch zu verwalten bzw. in Wob 2 zu erstellen. Genauer: Dokumente und Folien können, ähnlich wie normale HTML-Seiten, aus Bausteinen zusammengesetzt werden, wobei →Assets wie Bilder oder auch ganze Folien dann mehrfach verwendet werden können. Aus den Daten können dann leicht verschiedene Ausgabeformate (HTML, PDF, LaTeX, ...) generiert werden.

Insbesondere diese Anwendung ist mit den unflexiblen und auf normale Publikation ausgelegten Datenmodellen gängiger Systeme kaum realisierbar, da bei diesen die Rohdaten in verschiedenen Formaten und unterschiedlichen Granularitäten aufbereitet werden müssen.

Nach den positiven Erfahrungen der ersten prototypischen Umsetzungen im Rahmen der Arbeit ist zu erwarten, dass Modellierung und Implementierung dieser Anwendungen auf Grund des Datenmodells bedeutend einfacher und besser zu machen ist als mit existierenden Modellen bzw. Systemen.

A. Beispiel: `homepage.w2l`

In diesem Anhang ist eine beispielhafte Familiendefinition aufgeführt, die die in den vorangegangenen Kapiteln spezifizierten Modell- und Sprachkonzepte anhand des Homepage-Beispiels illustriert. Aus Gründen der Übersichtlichkeit wird das Listing dabei abschnittsweise kommentiert. Abbildung A.1 zeigt das zugehörige UML-Klassendiagramm; dabei sind nur die explizit in den Klassen definierten Felder bzw. Methoden aufgeführt.

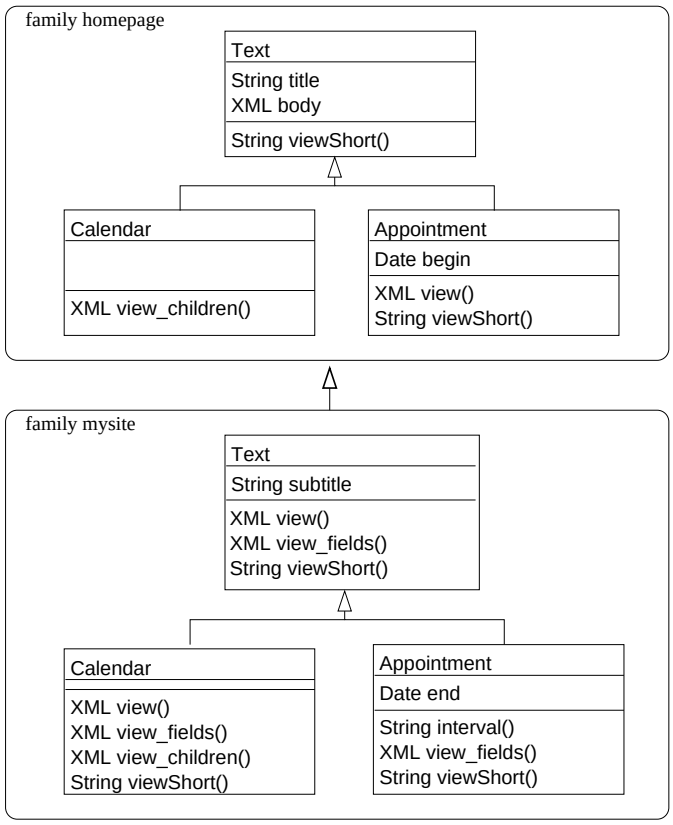


Abbildung A.1.: UML-Klassendiagramm `homepage.w2l`

In den Zeilen 1 bis 29 wird zunächst die default-Familie `Family` mit der default-Daten- bzw. Feldklasse definiert. Diese ist fest im System implementiert und muss nicht vom Benutzer spezifiziert werden. Diese implizite Definition ist hier nur aus Gründen der Dokumentation aufgeführt.

Die Methode `XML view()` der `Fieldclass` liefert über das Schlüsselwort `this` den Wert des Feldobjekts zurück. Die `view()`-Methode der `Dataclass` gibt innerhalb eines `<block>`-Elements das Ergebnis der Methoden `view_fields()` bzw. `view_children()` zurück. Hier werden in einer Schleife jeweils die `view()`-Methoden der Feldobjekte (Zeile 19) bzw. der Kindobjekte (Zeile 25) aufgerufen. Das Ergebnis ist also eine rekursive Ausgabe der Feldobjekte innerhalb einer Hierarchie von Objekten (Datenbaum). Dabei wird jeweils die inhärente Reihenfolge (s. Abschnitt 4.6.2) verwendet.

```
1  family Family {                    // default root family
2
3   class Fieldclass {                // default field class
4    XML view() {
5     this;
6    }
7   }
8
9   class Dataclass {                 // default data class
10   XML view() {
11    <block>
12     view_fields();
13     view_children();
14    </block>
15   }
16
17   XML view_fields() {
18    for (field : getFields()) {
19     if(field.isdefined()) field.view();
20    }
21   }
22
23   XML view_children() {
24    for (child : getChildren()) {
25     child.view();
26    }
27   }
28  }
29 }
```

family homepage

Die Zeilen 30 bis 67 definieren die (benutzerdefinierte) Familie `homepage` mit den Klassen `Text`, `Calendar` und `Appointment`.

Die Klasse `Text` (Zeilen 31 bis 42) ist eine einfache Textklasse, die einen Titel vom Typ `String`, einen Rumpf vom Typ `XML` und eine Methode `String viewShort()` hat. `Text.title` ist eine Feldklasse, die ihre eigene `view()`-Methode definiert, hier wird der Titel (Selbstreferenz `this`) innerhalb eines `<title>`-Elements zurückgegeben. `<title>` ist dabei ein Bestandteil der Zwischensprache (s. Abschnitt 8.4) und wird später, zusammen mit dem Rest der Ausgabe, über ein XSLT-Stylesheet in HTML transformiert. `viewShort` gibt einen kurzen String zurück, der z. B. als Linktext auf das entsprechende Objekt verwendet werden kann (in diesem Fall nur den Titel als String).

```
30 family homepage {     // implizit: extends Family
31  class Text {         // implizit: extends Dataclass
32   String title {
33    XML view() {
34     <title>{this}</title>
35    }
36   };
37   XML body;
38
39   String viewShort() {
40    title;
41   }
42  }
```

`Calendar` (Zeilen 43 bis 51) ist als Containerklasse bzw. -objekt für einzelne Termine gedacht, so dass nur Kindobjekte der Klasse `Appointment` erlaubt sind (Zeile 44, vgl. hierzu Abschnitt 4.6.3). Die `view_children()`-Methode wird derart überschrieben, dass die Kindobjekte nach dem Starttermin sortiert ausgegeben werden.

```
43  class Calendar extends Text {
44   property allowedChildren [Appointment];
45
46   XML view_children() {
47    for (child : sort_objects(getChildren(),["begin"])) {
48     child.view();
49    }
50   }
51  }
```

Die Klasse Appointment (Zeilen 52 bis 66) modelliert einen einzelnen Termin, dessen Datum im Feld `begin` steht. Die `view_fields()`-Methode ist derart überschrieben, dass zunächst Titel und Beginn des Termins, gefolgt vom `body` ausgegeben werden. Der Beginn wird dabei innerhalb eines `<title_extra>`-Elements[1] ausgegeben. Hier wird also nicht auf die automatische Ausgabe der Felder über eine Schleife (Zeile 18) zurückgegriffen. `viewShort()` liefert wiederum eine kompakte Ausgabe.

```
52  class Appointment extends Text { // einfacher Termin: nur Beginn
53   Date begin;
54
55   XML view_fields() {
56    {title.view()}
57    <title_extra>{begin.toString()}</title_extra>
58    {body.view()}
59   }
60
61   String viewShort() {
62    begin.toString();
63    ": ";
64    super.viewShort();
65   }
66  }
67 } // family homepage
```

[1] Diese Elemente sind wiederum Bestandteile der Zwischensprache, mit denen der eigentliche Titel bzw. eine Zusatzinformation dazu angegeben wird.

family mysite

In den Zeilen 68 bis 114 wird die von der Familie `homepage` abgeleitete Familie `mysite` definiert. Die Klasse `Text` enthält ein weiteres Feldobjekt, `subtitle`, das einen Untertitel spezifiziert, die Klasse `Appointment` einen Endtermin.

In der Klasse `Text` wird dabei in Zeile 69 definiert, dass das Feldobjekt `subtitle` zwischen `title` und `body` „einsortiert“ wird (vgl. Abschnitt 4.6.2). Dieses Feld wird dadurch in den `view()`- bzw. `view_fields()`-Methoden ausgegeben, also auch in abgeleiteten Klassen. Die Methode `view_fields()` fügt nach der Ausgabe der Felder noch eine Fußzeile hinzu, `view_short()` erzeugt wie bisher eine kompakte Ausgabe.

```
68 family mysite extends homepage {
69  class Text {                //implizit: extends homepage.Text
70   String subtitle after title {
71    XML view() {
72     <subtitle>{this}</subtitle>
73    }
74   };
75
76   XML view_fields() {
77    super.view_fields();
78    <p>Fußzeile</p>
79   }
80
81   String viewShort() {
82    title; " ("; subtitle; ")";
83   }
84  }
```

Die Klasse `Calendar` (Zeilen 85 bis 90) enthält lediglich zwei Methodendeklarationen, die jedoch keinen Programmcode enthalten, sondern nur eine Prioritätsangabe. Wie in Abschnitt 5.2.1 beschrieben, werden die entsprechenden Methodendefinitionen hier aus der Superfamilie (also `homepage.Calendar` und nicht `mysite.Text`) materialisiert. Würde die Prioritätsangabe fehlen, so wäre dies ein Fehler, da nicht eindeutig bestimmbar ist, welche Methode die „richtige“ ist, da beispielsweise `view()` sowohl in `homepage.Calendar` als auch in `mysite.Text` definiert ist.

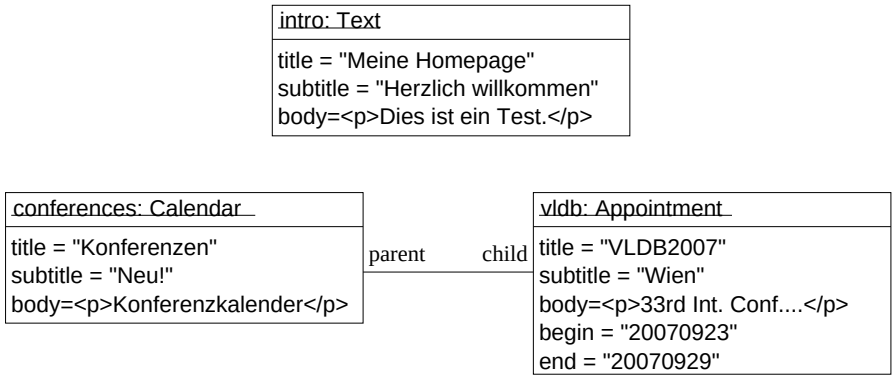
```
85  class Calendar extends Text {
86   XML view() priority family;
87   XML view_fields() priority family;
88   XML view_children() priority family;
89   String viewShort() priority family;
90  }
```

Die Klasse Appointment wird in den Zeilen 91 bis 113 um ein Feld `end` erweitert, das das Ende eines Termins angibt. Die Methode `interval` gibt dabei den Starttermin aus, und falls definiert, den Endtermin mit führendem Bindestrich. Die Methoden `view_fields()` und `viewShort()` sind entsprechend angepasst, so dass statt des Starttermins eben jenes Intervall angegeben wird.

```
91  class Appointment extends Text {
92   Date end;
93
94   String interval() {
95    if (begin.isdefined()) {
96     begin.toString();
97    }
98    if (end.isdefined()) {
99     " - "; end.toString();
100   }
101  }
102  XML view_fields() {
103   {title.view()}
104   <title_extra>{interval()}</title_extra>
105   {subtitle.view()}
106   {body.view()}
107  }
108  String viewShort() {
109   interval();
110   ": ";
111   super.viewShort();
112  }
113 }
114 }  // family mysite
```

Um das Beispiel abzurunden. werden im Folgenden einige Beispielobjekte sowie Beispielausgaben gezeigt. Die nachstehende Abbildung zeigt drei Beispielobjekte aus der `mysite`-Familie in UML-Darstellung[2].

[2] `intro`, `conferences` sowie `vldb` sind hier Objekt-IDs.



Der Aufruf von `object(intro).view()` würde folgendes liefern[3]:

```
<block>
  <title>Meine Homepage</title>
  <subtitle>Herzlich Willkommen</subtitle>
  <p>Dies ist ein Test</p>
  <p>Fußzeile</p>
</block>
```

Die Ausgabe erfolgt entsprechend der Methode `mysite.Text.view_fields()` (Zeilen 76-79 im obigen Beispiel).

`object(conferences).view()` würde nachstehendes ausgeben:

```
1  <block>
2    <title>Konferenzen</title>
3    <subtitle>Neu!</subtitle>
4    <p>Konferenzkalender</p>
5    <block>
6      <title>VLDB2007</title>
7      <title_extra>23.09.2007 - 29.09.2007</title_extra>
8      <subtitle>Wien</subtitle>
9      <p>3rd Int. Conf....</p>
10   </block>
11 </block>
```

Hier stammen die Zeilen 1-3 vom Objekt `conferences` (via `view_fields()` ausgegeben) und die Zeilen 4-7 vom Objekt `vldb` (rekursiv über `view_children()` ausgegeben).

Zu beachten ist bei diesem Beispiel, dass aus Gründen der Übersichtlichkeit Aspekte, die die Lokalisierung betreffen, ausgeklammert wurden. So müssen beispielsweise Texte

[3] Die `object`-Notation bezeichne hier die Dereferenzierung einer Objekt-ID zu einem Objekt.

in verschiedenen Sprachen und Zahlen sowie Datumsangaben sprachspezifisch ausgegeben werden (können). Dies ist jedoch kein zentraler Aspekt der Templatesprache und wird weitestgehend über XSLT-Stylesheets geregelt, was u. a. den Vorteil hat, dass der Entwickler sich nicht darum kümmern muss.

Glossar

Assets: bezeichnen die inhaltlichen Einheiten, aus denen eine →Webseite zusammengesetzt wird, wie z. B. Texte oder multimediale Komponenten (Bilder, Videos oder Sounds). Assets (beispielsweise ein Firmenlogo) werden auf einer →Website häufig mehrfach verwendet. Die Granularität von textuellen Einheiten hängt dabei von der jeweiligen Modellierung ab, so können beispielsweise eine gesamte Webseite, deren Absätze oder einzelne Sätze als Asset modelliert werden.

Assetmanagement: ist die Komponente eines CMS, welche die →Assets verwaltet.

Client/Server-System: Ein CMS besteht in der Regel aus einem via Internet erreichbaren Server, auf dem das eigentliche CMS läuft, und mehreren Clients, über die mit dem System gearbeitet werden kann. Dabei kann der Client ein einfacher Webbrowser sein oder eine komplexe Anwendung. Verschiedene Varianten von Clients werden in Abschnitt 8.5.2 vorgestellt; zu Grundlagen von Client/Server-Systemen siehe u. a. [Dad96].

Content: kann, je nach Kontext, ein →Asset oder aber den Inhalt einer →Webseite oder einer →Website bezeichnen.

Content Syndication: wird der Austausch bzw. Handel von Informationen genannt. Als Beispiel sei hier die Integration von Nachrichtenüberschriften („Headlines“) von einer anderen Webseite genannt. Eine oftmals verwendete Technologie hierbei ist RSS (Really Simple Syndication, siehe u. a. `http://www.rss3.org/`).

Portal: eine →Website, die versucht, verschiedene regelmäßig benötigte Dienste zu bündeln oder eine Übersicht für den Einstieg in einen Themenkomplex zu schaffen.

Webseite vs. Website: Mit Web*seite* wird zumeist eine im Browser sichtbare Seite bezeichnet. In einem statischen System ist dies z. B. eine HTML-Seite mit zugehörigen Bildern. Mit Web*site* (auch: Webpräsenz) ist im Allgemeinen das gesamte Webangebot gemeint, welches auf einem Server bzw. unter einer Domain erreichbar ist. Dies kann man auch als geschlossenes Informationssystem im WWW mit Webseiten auffassen, die über eine gemeinsame Homepage erreichbar sind.

Web Site Management: bezeichnet die administrativen Vorgänge bei der technischen bzw. redaktionellen Betreuung einer →Website.

Workflow: Workflows definieren Arbeitsabläufe, bei denen Dokumente, Informationen oder Aufgaben zwischen verschiedenen Beteiligten delegiert werden. Ein Beispiel eines Workflows im Kontext von CMS ist der Vorgang, dass ein Artikel, der von einem Redakteur geschrieben wird, zunächst nicht öffentlich sichtbar ist, und erst nach einer Korrekturlesung von einem Chefredakteur von diesem veröffentlicht wird.

Workflow-Management-Systeme: unterstützen die Spezifikation, Ausführung und Steuerung von Workflows. Oftmals haben CMS integrierte Workflow-Management-Module/Komponenten.

Akronymverzeichnis

1NF:	Erste Normalform
API:	Application Programming Interface
ASP:	Active Server Pages
BCNF:	Boyce-Codd Normalform
BLOB:	Binary Large Objects
BNF:	Backus-Naur Form
CGI-Skript:	Common Gateway Interface-Skript
CML:	Chemical Markup Language
CMS:	Content Management System
CSS:	Cascading Style Sheet
DAG:	Directed Acyclic Graph
DBMS:	Database Management System
DCD:	Document Content Description
DDL:	Data Definition Language
DML:	Data Manipulation Language
DMS:	Document Management System
DOM:	Document Object Model
DSL:	Domain Specific Language
DTD:	Document Type Definition
EBNF:	Erweiterte Backus-Naur Form
ebXML:	Electronic Business XML Initiative
ER-Modell:	Entity-Relationship Modell
FJ:	Featherweight Java
GML:	Generalized Markup Language
GPL:	General Purpose Language
GUI:	Graphical User Interface (dt.: Benutzer- oder Benutzungsschnittstelle)
HTML:	Hypertext Markup Language
JDK:	Java Developers Kit
JSP:	Java Server Pages
MathML:	Mathematical Markup Language
MVC:	Model-View-Controller
NF^2:	Non-First Normal Form
OO:	Objektorientierung
OODB:	Objektorientierte Datenbank
OOP:	Objektorientierte Programmierung
OOXML:	Object Oriented XML
PHP:	PHP Hypertext Preprocessor

PDA:	Personal Digital Assistant
PDF:	Portable Document Format
RDBMS:	Relationales Datenbankmanagementsystem
RELAX:	Regular Language description for XML
RELAX:	Regular Language description for XML-New Generation
RTF:	Rich Text Format
SAX:	Simple API for XML Processing
SGML:	Standard Generalized Markup Language
SQL:	Structured Query Language
SVG:	Scalable Vector Graphics
TREX:	Tree Regular Expressions for XML
UIML:	User Interface Markup Language
UML:	Unified Modelling Language
URL:	Uniform Resource Locator
W2L:	Wob 2 Language
W2OM:	Wob 2 Object Model
W3C:	World Wide Web Consortium
WCML:	WebComposition Markup Language
WCMS:	Web Content Management System
WML:	Wireless Markup Language
Wob:	Objects in the Web
WWW:	World Wide Web
XHTML:	eXtensible Hypertext Markup Language
XMI:	XML Metadata Interchange
XML:	eXtensible Markup Language
XPath:	XML Path Language
XSD:	XML Schema Definition
XSL:	eXtensible Stylesheet Language
XSLT:	eXtensible Stylesheet Language: Transformations
XSL-FO:	eXtensible Stylesheet Language: Formatting Objects
XSP:	eXtensible Server Pages

Literaturverzeichnis

[ABD+89] Atkinson, Malcolm, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier und Stanley Zdonik: *The Object-Oriented Database System Manifesto.* In: *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Seiten 223–240, Kyoto, Japan, 1989.

[Abi97] Abiteboul, Serge: *Querying Semi-Structured Data.* In: Afrati, Foto N. und Phokion Kolaitis (Herausgeber): *Database Theory—ICDT'97, 6th International Conference*, Band 1186 der Reihe *Lecture Notes in Computer Science*, Seiten 1–18, Delphi, Greece, 8–10 Januar 1997. Springer.

[ABS99] Abiteboul, Serge, Peter Buneman und Dan Suciu: *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[AC96] Abadi, M. und L. Cardelli: *A Theory of Objects.* Springer-Verlag, New York, 1996.

[ACJ04] Atanassow, Frank, Dave Clarke und Johan Jeuring: *UUXML: A Type-Preserving XML Schema–Haskell Data Binding.* In: Jayaraman, Bharat (Herausgeber): *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 2004, Proceedings*, Nummer 3057 in *LNCS*, Seiten 71–85, Berlin Heidelberg, June 2004. Springer–Verlag.

[AO03] Anderson, C. und K. Ostermann: *VC - foundations for virtual classes with dependent types.* `http://www.binarylord.com/work/vc.pdf`, 2003.

[Bal93] Balzert, Helmut: *Der JANUS-Dialogexperte: Vom Fachkonzept zur Dialogstruktur.* In: Doberkat, E.-E. (Herausgeber): *Proceedings der GI-Fachtagung Softwaretechnik 93*, Seiten 62–72, Dortmund, FRG, November 1993.

[Bar84] Barendregt, Henk P.: *The Lambda Calculus.* North Holland, Revised Auflage, 1984.

[Bar05] Bartholomäus, Michael: *Versionskontrolle von XML-Daten*. Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelms Universität Bonn, 2005.

[BC90] Bracha, Gilad und William R. Cook: *Mixin-based Inheritance*. In: *OOPSLA/ECOOP*, Seiten 303–311, 1990.

[BCF03] Benzaken, Véronique, Giuseppe Castagna und Alain Frisch: *CDuce: an XML-centric general-purpose language*. ACM SIGPLAN Notices, 38(9):51–63, September 2003.

[BDD+98] Bello, Randall G., Karl Dias, Alan Downing, Jr. James J. Feenan, James L. Finnerty, William D. Norcott, Harry Sun, Andrew Witkowski und Mohamed Ziauddin: *Materialized Views in Oracle*. In: *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, Seiten 659–664, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[Bee89] Beeri, C.: *Formal Models for Object Oriented Databases*. In: *First Int. Conf. on Deductive and Object-Oriented Databases , Kyoto*, Dezember 1989.

[Bee90] Beeri, C.: *A Formal Approach to Object-Oriented Databases*. Data & Knowledge Engineering, 5:353–382, 1990.

[BFS04] Broberg, Niklas, Andreas Farre und Josef Svenningsson: *Regular expression patterns*. SIGPLAN Not., 39(9):67–78, 2004.

[BM00] Behme, Henning und Stefan Mintert: *XML in der Praxis, zweite Auflage*. Addison Wesley Longman Verlag GmbH, Bonn, 2000.

[Boo91] Booch, Grady: *Object-Oriented Design with Applications*. Benjamin/ Cummings Series in Ada and Software Engineering. The Benjamin/ Cummings Publishing Company, Inc., 1991.

[Boo94] Booch, Grady: *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, second Auflage, 1994.

[BSM96] Bray, Tim und C. Michael Sperberg-McQueen: *Extensible Markup Language*. In: *SGML '96 Conference Proceedings*, Seiten 399–404, 1996.

[Bun97] Buneman, Peter: *Semistructured data*. In: ACM (Herausgeber): *PODS '97. Proceedings of the Sixteenth ACM SIG-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12–14, 1997, Tucson, Arizona*, Seiten 117–121, New York, NY 10036, USA, 1997. ACM Press.

[Car84] Cardelli, L.: *A Semantics of Multiple Inheritance.* In: *Proceedings of the International Symposium on Semantics of Data Types, Sophia-Antipolis (France)*, Seiten 51–68, Berlin, Juni 1984. Springer LNCS 173.

[Car01] Cardelli, Luca: *Describing semistructured data.* SIGMOD Record (ACM Special Interest Group on Management of Data), 30(4):80–85, Dezember 2001.

[Cat91] Cattell, R. G. G.: *Object Data Management: Object-Oriented and Extended Relational Database Systems.* Addison-Wesley, 1991.

[CCH+89] Canning, Peter, William Cook, Walter Hill, Walter Olthoff und John C. Mitchell: *F-bounded polymorphism for object-oriented programming.* In: *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, Seiten 273–280, New York, NY, USA, 1989. ACM Press.

[CGH94] Cremers, Armin B., Ulrike Griefahn und Ralf Hinze: *Deduktive Datenbanken: Eine Einführung aus der Sicht der logischen Programmierung.* Vieweg, 1994.

[Che76] Chen, Peter P.: *The Entity-Relationship Model - Toward a Unified View of Data.* ACM Trans. Database Syst., 1(1):9–36, 1976.

[Chu36] Church, Alonzo: *An unsolvable problem of elementary number theory.* American Journal of Mathematics, 58:345 – 363, 1936.

[Cod80] Codd, Edgar F.: *Data Models in Database Management.* In: *ACM SIGMOD Record, ACM*, Band 11, Juni 1980.

[CP89] Cook, William R. und Jens Palsberg: *A Denotational Semantics of Inheritance and its Correctness.* In: *OOPSLA*, Seiten 433–443, 1989.

[CS00] Cluet, S. und J. Simeon: *YATL: a functional and declarative language for XML.* Draft manuscript, 2000.

[CW85] Cardelli, Luca und Peter Wegner: *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 17(4):471–522, 1985.

[Dad96] Dadam, Peter: *Verteilte Datenbanken und Client/Server-Systeme.* Springer Verlag, Berlin, 1996.

[DD96] Doberkat, Ernst-Erich und Stefan Dissmann: *Einführung in die objektorientierte Programmierung mit BETA.* Addison-Wesley Publishing Co., Bonn, 1996.

[DN66] Dahl, Ole-Johan und Kristen Nygaard: *SIMULA: an ALGOL-based simulation language*. Commun. ACM, 9(9):671–678, 1966.

[EOC06] Ernst, Erik, Klaus Ostermann und William R. Cook: *A virtual class calculus*. ACM SIGPLAN Notices, 41(1):270–282, Januar 2006.

[Ern99] Ernst, Erik: *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Doktorarbeit, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern01] Ernst, Erik: *Family Polymorphism*. In: Knudsen, J. L. (Herausgeber): *ECOOP 2001*, Nummer 2072 in *LNCS*, Seiten 303–326. Springer Verlag, 2001.

[Ern03a] Ernst, Erik: *Higher-Order Hierarchies*. In: Cardelli, Luca (Herausgeber): *Proceedings ECOOP 2003*, LNCS 2743, Seiten 303–329, Heidelberg, Germany, Juli 2003. Springer-Verlag.

[Ern03b] Ernst, Erik: *Separation of Concerns*. In: *Proceedings of Software Engineering Properties of Languages for Aspect Technologies, SPLAT 2003, in association with AOSD 2003*, Seite 6, 2003.

[EVP01] Eisenstein, Jacob, Jean Vanderdonckt und Angel Puerta: *Applying Model-Based Techniques to the Development of UIs for Mobile Computers*. In: *Proceedings of the 2001 International Conference on Intelligent User Interfaces*, Seiten 69–76, 2001.

[GG99] Gellersen, Hans-W. und Martin Gaedke: *Object-Oriented Web Application Development*. IEEE Internet Computing, 3(1):60–68, 1999.

[GHJV95] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GJS05] Guerra, Rui, Johan Jeuring und Doaitse Swierstra: *Generic validation in an XPath-Haskell data binding*. In: *Proceedings of PLAN-X 2005*, 2005.

[GLPS04] Gapeyev, Vladimir, Michael Y. Levin, Pierce Pierce und Alan Schmitt: *The Xtatic Experience*. Technical Report MS-CIS-04-24, University of Pennsylvania, Oktober 2004.

[GMP70] Goldfarb, C. F., E. J. Mosher und T. I. Peterson: *An Online System for Integrated Text Processing*. Proceedings of the American Society for Information Science, 7:147–150, 1970.

[GSG00] Gaedke, Martin, Christian Segor und Hans-Werner Gellersen: *WCML: Paving the Way for Reuse in Object-Oriented Web Engineering.* In: *SAC (2)*, Seiten 748–755, 2000.

[HL95] Hürsch, Walter und Cristina Videira Lopes: *Separation of Concerns.* Technischer Bericht, College of Computer Science, Northeastern University, 1995.

[HM03] Hosoya und Murata: *Boolean Operations for Attribute-Element Constraints.* In: *CIAA: International Conference on Implementation and Application of Automata, LNCS*, 2003.

[HP00] Hosoya, Haruo und Benjamin C. Pierce: *XDuce: A Typed XML Processing Language (Preliminary Report).* In: Suciu, Dan und Gottfried Vossen (Herausgeber): *International Workshop on the Web and Databases (WebDB)*, Mai 2000. Reprinted in *The Web and Databases, Selected Papers*, Springer LNCS volume 1997, 2001.

[HP03] Hosoya, Haruo und Benjamin C. Pierce: *XDuce: A Statically Typed XML Processing Language.* ACM Transactions on Internet Technology, 3(2):117–148, Mai 2003.

[HRS+05] Harren, Matthew, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski und Vivek Sarkar: *XJ: facilitating XML processing in Java.* In: *WWW '05: Proceedings of the 14th international conference on World Wide Web*, Seiten 278–287, New York, NY, USA, 2005. ACM Press.

[IPW99] Igarashi, Atshushi, Benjamin Pierce und Philip Wadler: *Featherweight Java: A Minimal Core Calculus for Java and GJ.* In: Meissner, Loren (Herausgeber): *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, Band 34(10), Seiten 132–146, N. Y., 1999.

[ISO86] International Organization for Standardization, Genf, Schweiz: *ISO 8879:1986: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*, 1986.

[ISV05] Igarashi, Atsushi, Chieri Saito und Mirko Viroli: *Lightweight Family Polymorphism.* In: *APLAS*, Seiten 161–177, 2005.

[JDAO04] Jolly, Paul, Sophia Drossopoulou, Christopher Anderson und Klaus Ostermann: *Simple Dependent Types: Concord (FTfJP workshop version).* Juni 2004.

[JM02a] Jamil, Hasan M. und Giovanni A. Modica: *An object-oriented extension of XML for autonomous web applications.* In: Kalpakis, Konstantinos, Nazli Goharian und David Grossmann (Herausgeber): *Proceedings of the Eleventh International Conference on Information and Knowledge Management (CIKM-02)*, Seiten 161–168, New York, November 4–9 2002. ACM Press.

[JM02b] Jamil, Hasan M. und Giovanni A. Modica: *What hierarchies can do for XML.* Technischer Bericht, Department of Computer Science, Mississippi State University, Juni 2002.

[Jol04] Jolly, Paul: *Simple Dependent Types: Concord.* Masters thesis, Department of Computing, Imperial College London, Juni 2004.

[KL03a] Kempa, M. und V. Linnemann: *On XML Objects.* In: *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003.

[KL03b] Kempa, Martin und Volker Linnemann: *Type Checking in XOBE.* In: Weikum, Gerhard, Harald Schöning und Erhard Rahm (Herausgeber): *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung,*, Band P-26 der Reihe *Lecture Notes in Informatics*, Seiten 265–274. Gesellschaft für Informatik, 26.-28. Februar 2003.

[KMS04] Kirkegaard, C., A. Moller und M. I. Schwartzbach: *Static Analysis of XML Transformations in Java.* IEEE Transactions on Software Engineering, 30(3):181–192, 2004.

[Kni98] Kniesel, Günter: *Type-Safe Delegation for Dynamic Component Adaptation.* In: *ECOOP Workshops*, Seiten 136–137, 1998.

[Koh03] Kohlhaas, Christopher: *Pflegeoberflächen für Internet-Informationssysteme.* Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelms Universität Bonn, 2003.

[Lay99] Lay, Patrick: *Alternative/aktive Klienten für das Wob-System.* Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelms Universität Bonn, 1999.

[LB96] Lie, Håkon Wium und Bert Bos: *Cascading Style Sheets, level 1.* W3C Recommendation 17 Dec 1996, `http://www.w3.org/TR/CSS1`, 1996.

[LC01] Luyten, Kris und Karin Coninx: *An XML-Based Runtime User Interface Description Language for Mobile Computing Devices.* Lecture Notes

in Computer Science: Interactive Systems: Design, Specification, and Verification: 8th International Workshop, DSV-IS 2001. Glasgow, Scotland, UK, 2220:1–15, 2001.

[Lie86] Lieberman, Henry: *Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems.* In: Meyrowitz, Norman (Herausgeber): *OOPSLA'86 Conference Proceedings: Object-Oriented Programming: Systems, Languages, and Applications*, Seiten 214–223. ACM SIGPLAN, ACM Press, 1986.

[LLK04] Lay, Patrick und Stefan Lüttringhaus-Kappel: *Transforming XML Schemas into Java Swing GUIs.* In: Dadam, Peter und Manfred Reichert (Herausgeber): *GI Jahrestagung (1), INFORMATIK 2004 - Informatik verbindet, Band 1, Beiträge der 34. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 20. September - 24. September 2004 in Ulm*, Band P-50 der Reihe *LNI*, Seiten 271–276. GI, 2004.

[LS04] Lu, Kenny Zhuo Ming und Martin Sulzmann: *An Implementation of Subtyping Among Regular Expression Types.* In: *APLAS*, Seiten 57–73, 2004.

[LW93] Liskov, B. und J. Wing: *Family Values: A behavioral notion of subtyping.* Technischer Bericht MIT/LCS/TR-562b, 1993.

[Mal02] Malik, Ayesha: *Create flexible and extensible XML schemas-Building XML schemas in an object-oriented framework.* `http://www-106.ibm.com/developerworks/library/x-flexschema/`, 2002.

[Man03] Mancke, Sebastian: *Spezifikation und Adaption service-orientierter Benutzerschnittstellen.* Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelms Universität Bonn, 2003.

[Mar00] Martin, Robert C.: *The open-closed principle.* Seiten 97–112, 2000.

[Maz01] Mazzocchi, Stefano: *Reducing the Effects of Growth Saturation on Web Sites Production with the Adoption of a Publishing Framework Based on XML Technologies.* Diplomarbeit, Università degli Studi di Pavia, 2001.

[Mei02] Meier, Wolfgang: *eXist: An Open Source Native XML Database.* In: Chaudri, Akmal B., Mario Jeckle, Erhard Rahm und Rainer Unland (Herausgeber): *Web, Web-Services, and Database Systems*, NODe 2002 Web- and Database-Related Workshops. Springer LNCS Series, 2002.

[Mey88] Meyer, Bertrand: *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Mil02] Milne, Philip: *Long Term Persistence of JavaBeans Components: XMLEncoder*. Sun Microsystems Inc., `http://java.sun.com/products/jfc/tsc/articles/persistence4/`, 2002.

[Mit96] Mitchell, John C.: *Foundations for Programming Languages*. Cambridge, MA, 1996.

[MMP89] Madsen, Ole Lehrmann und Birger Møller-Pedersen: *Virtual Classes: A Powerful Mechanism in Object-Oriented Programming*. In: *Proc. of the OOPSLA-89: Conference on Object-Oriented Programming: Systems*, Seiten 397–406, Languages and Applications, New Orleans, LA, 1989.

[MMPN93] Madsen, Ole Lehrmann, Birger Møller-Pedersen und Kristen Nygaard: *Object-oriented programming in the BETA programming language*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[MR92] Myers, Brad A. und Mary Beth Rosson: *Survey on User Interface Programming*. In: *Proceedings of SIGCHI'92: Human Factors in Computing Systems*, May 1992.

[MS99] Meijer, Erik und Mark Shields: *XMλ: A Functional Language for Constructing and Manipulating XML Documents*. (Draft), 1999.

[MS05] Møller, Anders und Michael I. Schwartzbach: *The Design Space of Type Checkers for XML Transformation Languages*. In: *Proc. Tenth International Conference on Database Theory, ICDT '05*, Band 3363 der Reihe *LNCS*, Seiten 17–36. Springer-Verlag, Januar 2005.

[Mur00] Murata, M.: *Hedge Automata: a Formal Model for XML Schemata*. `http://www.geocities.com/ResearchTriangle/Lab/6259/hedge_nice.pdf`, 2000.

[NCM04] Nystrom, Nathaniel, Stephen Chong und Andrew C. Myers: *Scalable extensibility via nested inheritance*. In: *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, Seiten 99–115. ACM Press, 2004.

[Nov03] Novatchev, Dimitre: *Functional programming in XSLT using the FXSL library*. In: *Extreme Markup Languages 2003, Montreal, Canada*, August 2003.

[Ost02] Ostermann, Klaus: *Dynamically Composable Collaborations with Delegation Layers.* In: *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, 2002.

[Pie91] Pierce, Benjamin C.: *Programming with Intersection Types and Bounded Polymorphism.* Doktorarbeit, 1991.

[Pie02] Pierce, Benjamin C.: *Types and Programming Languages.* MIT Press, 2002.

[PJ99] Page-Jones, Meilir: *Fundamentals of object-oriented design in UML.* Addison-Wesley, 1999.

[Pue96] Puerta, Angel: *Issues in Automatic Generation of User Interfaces in Model-Based Systems.* In: Vanderdonckt, J. (Herausgeber): *Proceedings of the 2nd International Workshop on Computer-Aided Design of User Interfaces (CADUI'96) Namur, 5-7 June*, 1996.

[Pue97] Puerta, Angel: *A Model-Based Interface Development Environment.* IEEE Software, 14:41–47, 1997.

[Püt01] Pützfeld, Achim: *Effiziente Speicher- und Zugriffstrategien für XML-Dokumente.* Diplomarbeit, Institut für Informatik III, Rheinische Friedrich-Wilhelms Universität Bonn, 2001.

[Red01] Reddig, Wolfgang: *Perspektiven: Persistente Objekte mit anwendungsspezifischer Struktur und Funktionalität.* Doktorarbeit, Universität Bonn, Deutschland, 2001.

[Rel00] *Document Description and Processing Languages — Regular Language Description for XML (RELAX) — Part 1: RELAX Core.* Technischer Bericht DTR 22250-1, ISO/IEC, Oktober 2000.

[RMC04] Radetzki, Uwe, Sebatian Mancke und Armin B. Cremers: *IRIS: A Framework Supporting Composition and Device-Specific Access of Software Services.* In: *Proceedings of the 18th International Conference Informatics for Environmental Protection (EnviroInfo)*, Geneva, Switzerland, 2004.

[SB02] Smaragdakis, Yannis und Don S. Batory: *Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs.* ACM Trans. Softw. Eng. Methodol., 11(2):215–255, 2002.

[SC99] Simeon, J. und Sophie Cluet: *Using YAT to Build a Web Server.* In: *WebDB '98: Selected papers from the International Workshop on The World Wide Web and Databases*, Seiten 118–135. Springer-Verlag, 1999.

[Sch02] Schmidt, Martin: *Design and Implementation of a validating XML parser in Haskell.* Diplomarbeit, Fachhochschule Wedel, 2002.

[Sil98] Silberzahn, Nicoloas: *Dealing with the electronic patient record variability: Object Oriented XML.* Presented on the workshop „SGML/XML in Healthcare“, May 18th 1998 accompanying the GCA SGML/XML Europe 98 Conference Paris `http://www.digitalairways.com/NiS/ParisXML98/`, 1998.

[SM96] Stonebraker, Michael und Dorothy Moore: *Object-Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1996.

[Sny87] Snyder, Alan: *Inheritance and the Development of Encapsulated Software Components.* In: Shriver, Bruce und Peter Wegner (Herausgeber): *Research Directions in Object-Oriented Programming*, Series in Computer Systems, Seiten 165–188. The MIT Press, 1987.

[Str67] Strachey, Christopher: *Fundamental Concepts in Programming Languages.* Unveröffentlichtes Vorlesungsskript einer Vorlesung im Rahmen einer International Summer School in Computer Programming in Kopenhagen, August 1967.

[Tur96] Turau, Volker: *Algorithmische Graphentheorie.* Oldenbourg Verlag, 1996.

[Ull82] Ullman, J. D.: *Principles of Database Systems, 2nd ed.* Computer Science Press, 1982.

[vDKV00] Deursen, Arie van, Paul Klint und Joost Visser: *Domain-Specific Languages: An Annotated Bibliography.* SIGPLAN Notices, 35(6):26–36, 2000.

[Vos87] Vossen, Gottfried: *Datenmodelle, Datenbanksprachen und Datenbank-Management-Systeme.* Addison Wesley, Reading, Mass., 1987.

[Weg90] Wegner, Peter: *Concepts and paradigms of object-oriented programming.* SIGPLAN OOPS Mess., 1(1):7–87, 1990.

[Wit03] Wittmann, A.: *Towards Caesar: Family polymorphism for Java.* Diplomarbeit, Technische Universität Darmstadt, Fachbereich Informatik, 2003.

[WR99] Wallace, M. und C. Runciman: *Haskell and XML: Generic combinators or type-based translation?* In: *Proc. Int. Conf. on Functional Programming*, Seiten 148–159, September 1999.

[ZGMHW95] Zhuge, Yue, Hector Garcia-Molina, Joachim Hammer und Jennifer Widom: *View maintenance in a warehousing environment.* In: *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, Seiten 316–327, New York, NY, USA, 1995. ACM Press.