Möglichkeiten der Gestaltung flexibler Softwarearchitekturen für Präsentationsschichten, dargestellt anhand episodenbasierter medizinischer Dokumentation unter Einbeziehung topologischer Befundung

Diplomarbeit zur Erlangung des akademischen Grades Diplominformatiker, vorgelegt der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau

von

Jens Bohl

Betreuer: Dipl.-Ing. Christian Heller

verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Ilka Philippow

Inventarisierungsnummer: 2003-01-02/006/IN97/2232

Ilmenau, den 30. Dezember 2002

Copyright©2002-2003. Jens Bohl.

Res Medicinae – Information in Medicine – <www.resmedicinae.org>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Danksagung

An dieser Stelle möchte ich es nicht versäumen, denen zu danken, die diese Arbeit erst ermöglicht haben.

Ein herzliches **Dankeschön** geht an

> ...meine Familie für die jahrelange moralische und finanzielle Unterstützung.
>
> ...meinen Betreuer Christian Heller für die Möglichkeit der Bearbeitung dieses äußerst interessanten Diplomthemas, sowie die vielen Hinweise und die Bereitstellung von entsprechender Fachliteratur.
>
> ...die Ärzte der *Res Medicinae*-Mailingliste für ihre hilfreichen fachlichen Ratschläge zur Umsetzung des Prototypen.
>
> ...den Verlag *Urban und Fischer* für das Einverständnis zur unentgeltlichen Verwendung von *Sobotta – Atlas der Anatomie* im Rahmen des Prototypen zur *Topologischen Befundung*.

Vorwort

Das vorliegende Dokument stellt die Diplomarbeit von Jens Bohl zur Erlangung des Diplomtitels der Studienrichtung Informatik der Technischen Universität Ilmenau dar.

Es ist das Ergebnis einer mehrmonatigen, intensiven, theoretischen wie auch praktischen Auseinandersetzung mit einem Thema, welches sich aufgrund seines interdisziplinären Charakters nicht nur einem konkreten wissenschaftlichen Forschungsbereich zuordnen lässt. Vielmehr umfasst der Inhalt dieser Arbeit mehrere unterschiedliche Aspekte der Informatik: Komponentenbasierte Softwareentwicklung, Entwurfsmuster und Softwarearchitekturen, Medizinische Informatik und Softwareergonomie. Dieses breite Spektrum an Betrachtungen, bestehend aus den verschiedensten Bereichen der Informatik, zeichnet den vielseitigen Charakter dieser Arbeit aus.

Die Wahl des Themas der Diplomarbeit liegt in den unterschiedlichen Interessen des Autors begründet. Zum Einen sind diese klar im Bereich der Informationstechnologie – genauer des Softwareentwurfes und -design – angesiedelt. Zum Anderen stand nach Erlangung der Allgemeinen Hochschulreife die Entscheidung zwischen einem Studium der Informatik – oder der Medizin. Trotz des Entschlusses, Informatik den Vorzug zu geben, ist das Interesse an medizinischen Sachverhalten seitdem nicht verloren gegangen und spiegelt sich nun im praktischen Abschnitt dieser Diplomarbeit wider.

Der Kern des theoretischen Teils manifestiert sich in neuesten Ideen und Forschungsergebnissen auf dem Gebiet der komponentenbasierten Softwareentwicklung, und hierbei im Speziellen im Entwurf eines ontologischen Software-Frameworks. Dieses Framework basiert auf der Idee, jedes Element eines Softwaresystems einer streng hierarchischen Struktur unterzuordnen. Es gleicht einem Schichtensystem, wobei die Granularität von Ebene zu Ebene

zunimmt. Jede objektorientierte Klasse einer Anwendung kann in eine dieser Schichten eingeordnet werden.
Ein solches Framework bezeichnet man als *Ontologie*.
Existentielle Objektbeziehungen werden dabei durch den erweiterten Lebenszyklus von Softwarekomponenten beschrieben. Ein solcher Kreislauf stützt sich auf die Festlegung, dass jede Instanz genau ein Elternobjekt besitzt, durch das es initialisiert wird und von dem es existenziell abhängt. Dabei wurden bereits bestehende Konzepte wie bekannte Architekturmuster oder der *Component Lifecycle* von Apache [ava] um eigene Ideen erweitert.
All diese theoretischen Betrachtungen vereinigen sich im praktischen Teil dieser Diplomarbeit. Vor dem Hintergrund der Entwicklung einer medizinischen Software fanden die beschriebenen theoretischen Konzepte eine anwendungsorientierte Umsetzung. Das entstandene Programm ist ein Prototyp, da es weder komplett in Funktionsumfang, noch voll einsetzbar ist. Dennoch läuft es keineswegs Gefahr, nach relativ kurzer Zeit in Vergessenheit zu geraten. Die Software ist Teil eines klinischen Informationssystems namens *Res Medicinae*[1], welches in der Zukunft einzigartig in Umfang und Umsetzung sein wird und längerfristig eine Alternative zu existierenden kommerziellen Produkten darstellen soll. Dabei ist nicht nur die konkrete Realisierung des Prototypen, sondern auch die in ihm vorhandene Umsetzung neuester medizinischer Modelle zur Diagnostik und Befundung von Interesse. Letzteres spielt in dieser technischen Diplomarbeit zwar eine eher untergeordnete Rolle, wird aber im Kontext des vorliegenden Dokumentes aus Gründen des besseren Verständnisses fachlicher Zusammenhänge nicht unerwähnt bleiben.

Die Struktur und Reihenfolge der folgenden Kapitel ist so angelegt, dass dem Leser ein unkomplizierter Zugang zum Thema geboten und er dann Schritt für Schritt über die softwaretheoretischen Grundlagen der Arbeit zu den enthaltenen eigenen Ideen und ihrer Umsetzung geführt wird.
Kapitel zwei bildet im Anschluss an die Einleitung mit Betrachtungen zur Gestaltung von Präsentationsschichten den Einstieg in diese Arbeit. Nach einer kurzen Vorstellung der unterschiedlichen Arten von Benutzerschnittstellen werden ergonomische Gesichtspunkte bei der Konzeption von grafischen Oberflächen dargelegt und technische Möglichkeiten der Pro-

[1] Res Medicinae (*lat.*): die Medizin betreffend, Sache der Medizin

grammierung solcher Schnittstellen gezeigt.

Im Anschluss an das zweite folgen drei Kapitel, die das Kernstück der theoretischen Arbeit darstellen: Muster, Frameworks und komponentenbasierte Softwareentwicklung werden vorgestellt, um diese "State-Of-The-Art"-Technologien als Ausgangspunkt der selbsterarbeiteten Konzepte zu betrachten. Diese eigenen Ideen werden auch an passender Stelle in den drei Kapiteln dargelegt – und das immer in Bezug auf den praktischen Teil dieser Diplomarbeit.

Diesem ist dann auch mit Kapitel sechs ein eigener Abschnitt gewidmet, dessen Inhalt zunächst noch einmal die Darlegung der Ziele dieser Arbeit ist. Daraufhin werden die zugrundeliegenden medizinischen Modelle präsentiert und dann die praktische Umsetzung im Rahmen des bereits erwähnten Prototypen besprochen.

Das letzte Kapitel enthält abschließende Bemerkungen und gibt einen Ausblick auf zukünftige Arbeiten zu diesem Thema.

Auszüge aus dem Programmcode werden in der Schriftart `Typewriter Typeface`, fremdsprachige Bezeichnungen sowie Eigennamen und Zitate *kursiv* dargestellt. Begriffe, die einer zusätzlichen Erläuterung bedürfen, sind lediglich als Fußnoten angeführt, da ein Glossar den Lesefluss der Arbeit beeinträchtigen würde. Wichtige Abkürzungen und Übersetzungen finden sich im Anhang wieder.

Ilmenau, den 30. Dezember 2002 — Jens Bohl

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Res Medicinae . . . 3

2 Präsentationsschichten in modernen Anwendungen **5**

2.1 Einordnung der Präsentationsschicht . . . 5

2.2 Arten von Benutzerschnittstellen . . . 7

2.2.1 Zeichenbasierte Schnittstellen . . . 7

2.2.2 Sprachbasierte Schnittstellen . . . 8

2.2.3 Grafische Nutzerschnittstellen . . . 8

2.3 Ergonomische Gesichtspunkte . . . 9

2.3.1 Ergonomie . . . 10

2.3.2 Kriterien der Oberflächengestaltung [Str01] . . . 10

2.3.3 Anforderungen an moderne Benutzerschnittstellen . . . 11

2.4 Realisierung grafischer Nutzerschnittstellen . . . 13

2.4.1 Standalone Client-Server Systeme . . . 13

2.4.2 Web-basierte Systeme . . . 15

3 Muster **17**

3.1 Idiome . . . 18

3.2 Entwurfsmuster . . . 18

3.3 Architekturmuster . . . 22

4 Komponentenbasierte Softwareentwicklung **27**

4.1 Der Komponenten-Lebenszyklus 27
4.2 Weiterführende Konzepte und Ausblick 31
4.2.1 Separation of Concerns 31
4.2.2 Aspektorientierte Programmierung 31
4.2.3 Ausblick 32

5 Frameworks 33
5.1 Einordnung 33
5.2 Vor- und Nachteile 34
5.3 ResMedLib Framework 35
5.3.1 Grundlegende Idee 35
5.3.2 Struktur 36

6 Record - Modul zur medizinischen Dokumentation 42
6.1 Ziele 42
6.2 Modelle der Befundung 44
6.2.1 Episodenbasierte Befundung 44
6.2.2 SOAP-Modell 46
6.2.3 Topologische Befundung 47
6.3 Realisierung 49
6.3.1 Domain-Modell 49
6.3.2 Treetable-Komponente 50
6.3.3 Prototyp zur Topologischen Befundung 51

7 Zusammenfassung und Ausblick 56
Anhang A Literatur I
Anhang B Abkürzungen III
Anhang C Übersetzungen wichtiger Begriffe IV
Anhang D Thesen V
Anhang E Eidesstattliche Erklärung VII
Anhang F GNU Free Documentation License VIII

Abbildungsverzeichnis

1.1 Module von Res Medicinae 3

2.1 3-Tier-Architektur 6
2.2 ABC-Modell 10

3.1 Klassendiagramm der Abstract Factory 20
3.2 Klassendiagramm des Composite 21
3.3 Klassendiagramm des Observer 21
3.4 MVC-Paradigma und darin enthaltene Entwurfsmuster 23
3.5 HMVC-Muster 25

4.1 Abstrakter Lebenszyklus von Komponenten 29

5.1 Klasse Item 36
5.2 Objektbeziehungen in Res Medicinae 38
5.3 System-Ontologie mit konkreten Unterklassen und Beispiel 39
5.4 Model-Ontologie und konkrete Unterklassen 41

6.1 Probleme und Episoden 45
6.2 Hierarchie von Darstellungen des menschlichen Skelettsystems 48
6.3 Domain-Modell unter Einordung in Model-Ontologie 50
6.4 Beispiel der XML-Darstellung eines Overlays 53
6.5 Grafische Darstellung eines Overlays des menschlichen Beines 55

Tabellenverzeichnis

4.1 Operationen des Komponentenlebenszyklus' 30

5.1 System-Ontologie mit Analogien 39

6.1 SOAP-Model mit Beispielen 47

Kapitel 1

Einleitung

Dieses kurze Kapitel soll dazu dienen, die Motivation der vorliegenden Diplomarbeit darzulegen und das Ziel der theoretischen Ausarbeitung und der damit verbundenen praktischen Umsetzung zu beschreiben.

1.1 Motivation

Nach DIN 9126 ist **Softwarequalität** "*...die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.*"
Welche sind aber solche Merkmale?
Im Bereich von Softwarearchitektur und -entwurf spricht man in diesem Zusammenhang von Kriterien wie *Korrektheit*, *Robustheit*, *Erweiterbarkeit* und *Wiederverwendbarkeit* – um nur die wesentlichsten zu nennen. Solche Eigenschaften sind vor allem für den Entwickler interessant und definieren die "innere" Qualität von Software. Diese wird also von programmiertechnischen Aspekten bestimmt und setzt eine stete Weiterentwicklung von Methodik und Technologie vorraus. Korrektheit und Robustheit sind dabei sicher die wichtigsten und gleichzeitig ältesten Anforderungen an die Qualität von Anwendungen.
Doch gerade von der Einführung der objektorientierten Programmierung versprach man sich die Gewährleistung der beiden anderen Merkmale: eine hohe Erweiterbarkeit und Wie-

derverwendbarkeit von Software. Dass diese ehrgeizigen Ziele durch die alleinige Einführung eines weiteren Programmierparadigmas nicht erreicht wurden, ist heute bekannt. Gegenstand aktueller wie auch zukünftiger Arbeiten wird also der Versuch sein, die Schwächen der objektorientierten Programmierung zu überwinden.

In diesem Zusammenhang sind Entwurfs- und Architekturmuster sowie Frameworks zu betrachten, die in gewisser Weise den heutigen Stand der theoretischen, aber auch technologischen Entwicklung in diesem Bereich repräsentieren.

Hierauf aufbauend stellt die Komponentenbasierte Softwareentwicklung einen weiteren Schritt in diese Richtung dar: Der Komponenten-Lebenszyklus, *Cybernetic Oriented Programming* [Hel02] und die darin vorhandenen Software-Ontologien sind domänenunspezifische, portierbare und intuitive Konzepte, deren Darlegung Aufgabe dieser Diplomarbeit ist.

Neben der Verwirklichung innerer Softwarequalität ist die optimale Erfüllung "äußerer" Qualitätsmerkmale eine zweite Herausforderung. In diesem Zusammenhang ist *Benutzbarkeit* von Software anzuführen. Benutzbarkeit bedeutet zum Einen, wie einfach das Programm zu bedienen ist, und zum Anderen, wie schnell seine Funktionalität erfasst und erlernt werden kann. Der Begriff der *Ergonomie* prägt dabei den Entwurf von Software und ist maßgebend für das Erreichen der äußeren Qualitätsmerkmale.

Diese können aber auch nur dann verwirklicht werden, wenn das zugrundeliegende Datenmodell, mit dem über eine grafische Oberfläche interagiert werden soll, eine für das jeweilige Problem optimale Struktur aufweist. Konventionelle Dokumentationsformen – bereitgestellt durch heutige Praxis-Software – weisen diesbezüglich Mängel auf. In dem zu dieser Diplomarbeit gehörenden Prototypen wurden neue medizinische Dokumentationsmodelle umgesetzt, die diese Schwachstellen beheben, und dadurch ebenfalls einen Beitrag zur Verbesserung der Qualität dieser Software leisten sollen.

Zunächst einige Worte zum Rahmenprojekt der praktischen Arbeit, das im Laufe des Dokuments immer wieder Erwähnung findet.

1.2 Res Medicinae

Der Kerngedanke bzw. die Intention des Projektes *Res Medicinae* besteht darin, ein stabiles, erweiterbares und plattformunabhängiges, klinisches Informationssystem auf der Basis von Open Source zu entwickeln. *Res Medicinae* ist also völlig frei erhältlich. Der implementierte Code unterliegt der *General Public License* (GPL) und darf somit unter Angabe des Autors weiterverwendet werden.

Res Medicinae hat sich zum Ziel gesetzt, der Kommerzialisierung heutiger Softwaresysteme sowie der damit verbundenen Abhängigkeit von einem bestimmten Hersteller entgegenzuwirken. Das Projekt-Team setzt sich bisher fast ausschließlich aus engagierten Studenten der Technischen Universität Ilmenau zusammen, ist aber offen für internationale Beteiligung. Unterstützt wird das Team dabei durch den kompetenten Rat zahlreicher Mediziner, die in den Mailinglisten Fragen beantworten und über fachliche Probleme bei der Entstehung der Software mitdiskutieren.

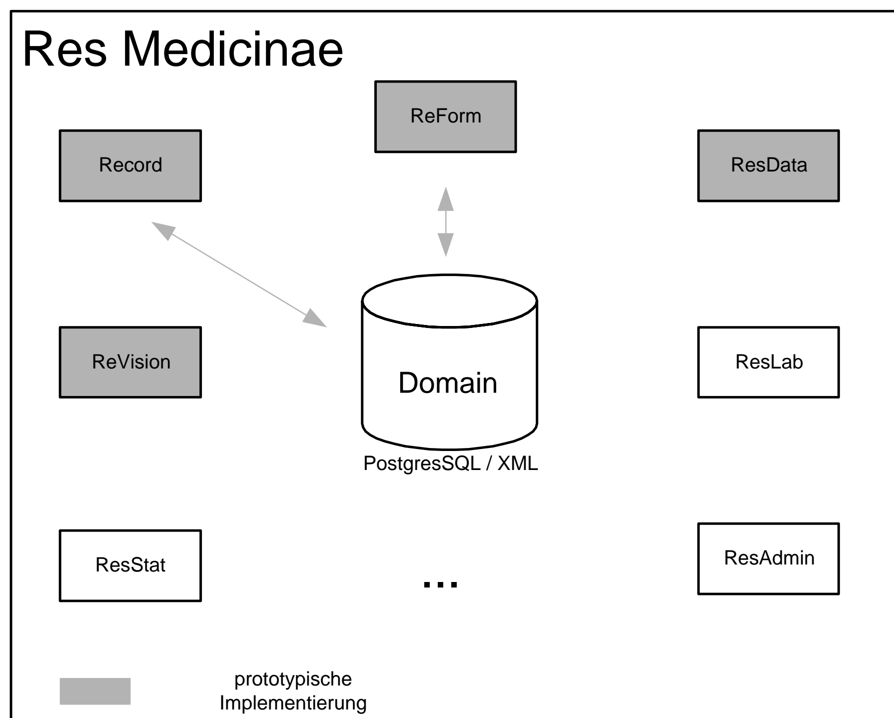

Abbildung 1.1: Module von Res Medicinae

Res Medicinae steht am Anfang seiner Entwicklung. Während der Implementation des Prototypen war die Struktur des zugrundeliegenden Frameworks ständigen Anpassungen unterworfen. Die in jener Zeit entwickelten Module[1] gehören zu den ersten Softwarekomponenten dieses Systems und stellen entsprechend den Eigenschaften einer prototypischen Implementierung auch nur rudimentäre Funktionalität bereit (siehe Abb. 1.1).

[1] Der Leser wird an dieser Stelle um etwas Geduld gebeten – eine genauere Erläuterung der Module (*Record*, *ReForm*) und der in ihnen implementierten Funktionalität folgt in Kapitel sechs.

Kapitel 2

Präsentationsschichten in modernen Anwendungen

Die Notwendigkeit, Präsentationsschichten informationsverarbeitender Systeme der jeweiligen Aufgabenstellung angemessen zu realisieren, stellte sich schon früh in der Geschichte der Softwaretechnik. Letztlich kann diese Eigenschaft die Akzeptanz von Software in gehörigem Maße beeinflussen. Genügte es in der Anfangszeit des digitalen Zeitalters, mit lediglich kommandozeilenbasierten Programmen zu arbeiten, stellt gerade der Softwarenutzer des 21. Jahrhunderts enorm hohe Ansprüche an Bedienbarkeit und Aussehen der Oberfläche einer Anwendung. Dabei gehen eine harmonische und ansprechende Gestaltung sowie eine nutzerfreundliche Steuerung Hand in Hand.

2.1 Einordnung der Präsentationsschicht

Innerhalb von Softwarearchitekturen sollte man zwischen mehreren logisch voneinander getrennten Schichten unterscheiden. Im Allgemeinen wird eine Separation zwischen Datenhaltung und der Repräsentation dieser Daten vorgenommen. So gibt es die sogenannte *3-Tier*[1]-Architektur, welche in der untersten Schicht die Datenverwaltung über eine Datenbank realisiert und mit der zweiten, die auch als *Business-Tier* bezeichnet wird, die eigentliche Logik

[1] Tier (*engl.*): Schicht

der Anwendung kapselt. Die oberste Schicht nimmt die visuelle Darstellung der Daten vor. Diese letzte wird daher als Präsentations- oder Darstellungsschicht bezeichnet.
Die folgende Abbildung stellt eine solche typische Trennung grafisch dar.

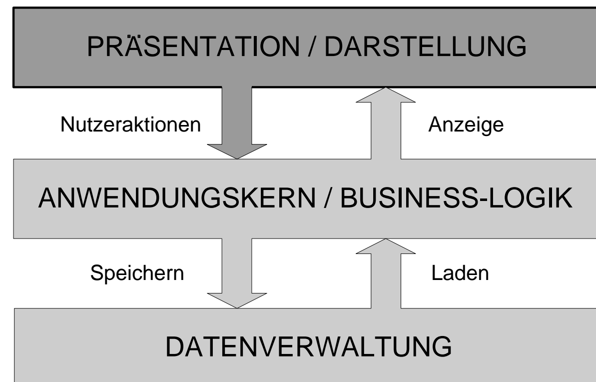

Abbildung 2.1: 3-Tier-Architektur

Die Datenverwaltung als sogenanntes *Back-End* des Systems übernimmt das Speichern und Laden, also die Bereitstellung von Informationen. Sie trennt das Datenbankmanagementsystem (DBMS) vom Rest der Anwendung. Über Programmierschnittstellen wie *Open Database Connectivity* (ODBC) oder *Java Database Connectivity* (JDBC) erfolgt der Zugriff auf eine Datenbank und dadurch auch die Transformation der unterschiedlichen Datenformate zwischen Persistenzmedium und Anwendung. Die physische Datenstruktur kann von der des Anwendungskernes stark abweichen. Auch diesen Punkt des Daten-Mappings übernimmt die Datenverwaltungsschicht.
Der Anwendungskern (*Middle-Tier*) als zweite Schicht realisiert die Fachlichkeit des Systems, d.h. die Funktionalität entsprechend der jeweiligen Spezifikation. Dieser Teil sollte streng von den anderen beiden getrennt sein, um nicht durch vermeidbare Abhängigkeiten die Konzepte der Wiederverwendbarkeit und Wartbarkeit zu untergraben.
Aufgabe der Präsentationsschicht ist die Visualisierung der durch die anderen Schichten aufbereiteten Informationen und die Bereitstellung von Möglichkeiten zur Manipulation der dargestellten Daten. Im Kontext dieser Diplomarbeit ist diese letzte Schicht als Benutzerschnittstelle oder *Front-End* einer Anwendung besonders interessant.

2.2 Arten von Benutzerschnittstellen

Benutzerschnittstellen dienen der Steuerung von Softwaresystemen und der Anzeige von Ergebnissen bzw. des aktuellen Systemzustands [Str01]. Zum Einen sollte über diese Schnittstelle eine umfassende unidirektionale Kontrolle des Programmes durch den Nutzer erfolgen. Zum Anderen stellt dieser Bereich einer Anwendung aber auch immer eine Form der Interaktion bereit: Es wird nicht nur erwartet, dass das System die geforderte Funktionalität abarbeitet, sondern auch Meldungen über den Status liefert und so den Nutzer über den aktuellen Stand der Arbeit informiert. Bei der Betrachtung von Nutzerschnittstellen unterscheidet man im wesentlichen zwischen folgenden drei Basistechnologien:

- zeichenbasierte Schnittstellen
- sprachbasierte Schnittstellen
- grafische Nutzerschnittstellen.

Auf diese drei Arten wird in den nächsten Abschnitten kurz eingegangen.

2.2.1 Zeichenbasierte Schnittstellen

Zeichenbasierte Schnittstellen sind sogenannte Kommandozeilenschnittstellen bzw. Textmasken, welche die Eingabe von Befehlen mit Parametern erlauben. Sie sind nach der Verwendung von Lochkarten die ältesten Schnittstellen zwischen Mensch und Computer. Diese Interaktionsform stellt in gewisser Weise das Pendant zur Eingabe über eine hierarchische Menüführung dar. Während der Nutzer über Menüs eine Auswahl der zum gegenwärtigen Zeitpunkt möglichen Funktionen präsentiert bekommt, wird bei kommandozeilenbasierten Werkzeugen vorrausgesetzt, dass die nötigen Befehle bekannt sind. Dieses Prinzip birgt Vor- und Nachteile. Zunächst muss der Nutzer ein bestimmtes Maß an Syntax und Semantik der Befehle beherrschen. Dementsprechend kann der Lernaufwand relativ hoch werden.
Auf der anderen Seite haben sich solche Systeme gerade im Kontext zeitkritischer Eingaben bewährt. Ein Funktionsaufruf über die Maus in möglicherweise verschachtelten Untermenüs

wird nie so schnell ausgeführt werden können, wie die Eingabe weniger Zeichen über die Tastatur. Zudem können häufig verwendete Funktionsabläufe in Skripten kaskadiert und individuell zusammengestellt werden. Kommandos sind parametrisierbar und erlauben auch auf diese Art und Weise ein hohes Maß an Flexibilität.
Zusammenfassend kann man sagen, dass diese Form einer Benutzeroberfläche trotz des Siegeszuges grafischer Oberflächen nie verschwinden wird, da der enorme Geschwindigkeitsvorteil bei der Bedienung besonders für geübte Anwender von Interesse ist.

2.2.2 Sprachbasierte Schnittstellen

Unter sprachbasierten Schnittstellen versteht man die automatische Erkennung von Sprache zur Bedienung eines technischen Systems. Diese recht junge Technologie wird zum Beispiel im Bereich automatischer Telefonauskünfte genutzt und findet in PC-basierten Softwarelösungen bisher kaum Einsatz.
IBM's *ViaVoice* als eines der wenigen PC-Programme zur Erkennung gesprochener Worte hinterlegt diese als Text in Editoren. Dies ist jedoch weniger eine Interaktionsform zwischen Mensch und Maschine, als eine alternative Art, dem Computer Informationen zu übermitteln. Sprachbasierte Schnittstellen stehen am Beginn ihrer Entwicklung, welche eng an den Fortschritt der Neuronalen Netze gebunden ist. Durch kontinuierliches Training ist es möglich, speziellen Neuronalen Netzen einen Wortschatz zuzuführen (Lernen des Netzes), auf den dann in der Abrufphase zurückgegriffen werden kann. So soll neben Erkennung von gesprochenen Worten (Spracherkennung) auch die Identifizierung des Sprechers (Sprechererkennung) möglich sein [Hel97].

2.2.3 Grafische Nutzerschnittstellen

Die wohl mittlerweile am weitesten verbreitete Art von Benutzerschnittstellen sind grafische Oberflächen, die sich besonders im Rahmen von PC-Anwendungen etabliert haben. Die Interaktion zwischen Mensch und Computer wird über vektorbasierte, grafische Komponenten vollzogen. Zu ihnen gehören zum Beispiel Schalter (Buttons), Formulare oder Menüs.

Damit diese Form der Interaktion eine für den Benutzer optimale Schnittstelle eines Informationssystems darstellt, müssen einige Kriterien beachtet werden, auf die im Folgenden eingegangen werden soll.

2.3 Ergonomische Gesichtspunkte

Die rasante Entwicklung auf dem Gebiet der elektronischen Datenverarbeitung führte lange Zeit dazu, viele Menschen nur als Teilnehmer der Innovationen zu betrachten, ohne wirklich auf die Bedürfnisse des Anwenders einzugehen. Es genügte in dieser Zeit, sich lediglich die Funktionalität einer Anwendung zur reinen Abarbeitung einer Aufgabe zu Nutze zumachen und diese weiter zu entwickeln. Im Laufe der Jahre wurde Entwicklern und Anwendern bewußt, dass diese recht einseitige Sicht der Dinge nicht allein schnelles und erfolgreiches Arbeiten mit dem Computer gewährleistet. Die Beachtung *menschlicher Bedürfnisse* gewann zunehmend an Bedeutung. Diese ergeben sich aus physiologischen sowie psychologischen Gesetzmäßigkeiten, denen jeder Mensch unterworfen ist. Dazu zählen Besonderheiten in der visuellen menschlichen Wahrnehmung ebenso wie die Lenkung der Aufmerksamkeit des Betrachters. Außerdem ist der heutige Nutzer eines Informationssystems längst nicht mehr solch ein Experte auf dem Gebiet der Informationstechnik wie es noch vor einigen Jahren der Fall war. Dennoch besitzt er unterschiedlichste Fähig- und Fertigkeiten im Umgang mit rechnergestützter Datenverarbeitung, die ebenfalls beim Software-Entwurf der Präsentationsschicht beachtet werden müssen.

Beginnend mit einer Erläuterung, was *Ergonomie* im eigentlichen Sinne beinhaltet, wird in diesem Abschnitt auf elementare Kriterien der Oberflächengestaltung eingegangen. Überlegungen, wie sie vor dem Entwurf einer jeden Benutzerschnittstelle angestellt werden sollten. Am Ende dieses Abschnittes werden gerade aktuelle Anforderungen an moderne grafische Schnittstellen besprochen. Diese Betrachtung erweitert die vorher dargelegten theoretischen Aspekte einer nutzerfreundlichen Gestaltung um erwünschte praktische Eigenschaften grafischer Benutzerschnittstellen im medizinischen Umfeld von *Res Medicinae*.

2.3.1 Ergonomie

Der Begriff *Ergonomie* leitet sich aus den beiden griechischen Wörtern *Ergos* für Arbeit und *Nomos* für Recht oder Gesetz ab. Laut Duden steht das Wort *Ergonomie* für die Anpassung der Arbeit an den Menschen. Die Ergonomie einer Benutzerschnittstelle wird danach bewertet, wie gut sie den Benutzer bei seiner Arbeit unterstützt. Softwarequalität misst man daher nicht mehr nur allein an Kriterien wie Korrektheit, Robustheit, Wartbarkeit oder Erweiterbarkeit, sondern auch an Bedienbarkeit.

2.3.2 Kriterien der Oberflächengestaltung [Str01]

Das *Aufgabe-Benutzer-Computer-Modell* (ABC-Modell) stellt die Problematik des Entwurfs von Benutzeroberflächen anschaulich dar.

ABC-Modell Das Dreiecksverhältnis zwischen Computer, Benutzer und der gestellten Aufgabe bestimmt eine Reihe von Regeln die aus diesen Beziehungen resultieren (vgl. Abb. 2.2).

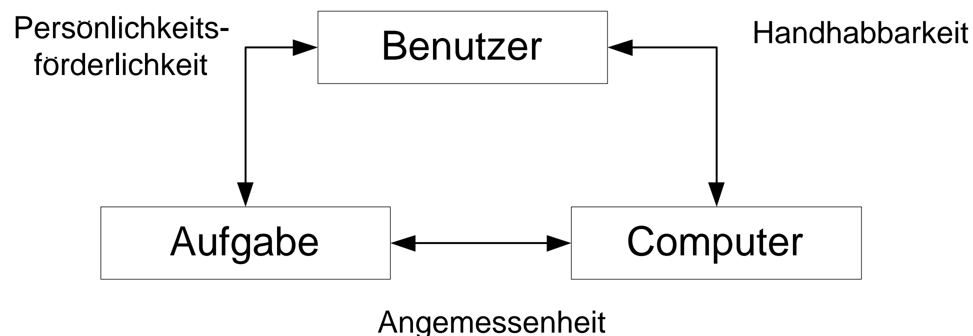

Abbildung 2.2: ABC-Modell

Der Benutzer sollte sich mit der Aufgabe identifizieren können, wobei der Begriff der *Persönlichkeitsförderlichkeit* eine Rolle spielt. Um die Aufgabe entsprechend abarbeiten zu können, ist sicherzustellen, dass der Anwender auch der richtige Bearbeiter hierfür ist. Dies ist jedoch ein arbeitswissenschaftlicher Aspekt, auf den nicht weiter eingegangen werden

soll. Im hier behandelten Kontext sind die Punkte *Handhabbarkeit* und *Angemessenheit* von größerer Bedeutung.
Angemessenheit bedeutet, inwiefern die Benutzerschnittstelle überhaupt in der Lage ist, die an sie gestellte Aufgabe erledigen zu können. Dies setzt vorraus, dass die Aufgabe in ihrer Komplexität, Strukturierung und dem dafür notwendigen Ablauf bekannt sein muss. Die Frage: "Was soll das System eigentlich leisten?" muss beantwortet sein.
Dem gegenüber steht die *Handhabbarkeit* des Systems. Es existieren unter Umständen die unterschiedlichsten Nutzergruppen bei – und Kontexte in denen die Anwendung eingesetzt werden soll. Auch spielt eine Rolle, wie oft das System für diese Aufgabe herangezogen wird und welche Vorkenntnisse deren Benutzer mit derartigen Schnittstellen besitzen.

Der nächste Abschnitt des Kapitels zeigt einige Beispiele für die Anpassung der grafischen Benutzerschnittstelle an konkrete Ansprüche des Anwenders. Auf Basis der zugrundeliegenden Anforderungen aus dem Analyse-Dokument (Pflichtenheft) des Projektes *Res Medicinae* werden erwünschte Eigenschaften der zukünftigen Oberfläche dargelegt.

2.3.3 Anforderungen an moderne Benutzerschnittstellen

Zu Beginn der Entwicklung von *Res Medicinae* wurde von den damaligen Projektmitgliedern ein Analyse-Dokument [CH02] verfaßt, welches Konzepte und Prinzipien der zu realisierenden Software beinhaltet. Das Projekt-Team setzt sich aus einer Reihe von praktizierenden Ärzten zusammen, die ihre Vorstellung von einem modernen klinischen Informationssystem darlegten. Vor allem der Gestaltung der grafischen Benutzeroberfläche sowie der zugehörigen Bedienung wurde große Bedeutung zugemessen.
Folgende Punkte traten besonders in den Vordergrund:

Programmsteuerung und "Intelligenz" des Systems Der Aspekt der **zeitlichen Effizienz** ist bei klinischen Anwendungen besonders zu berücksichtigen. Eine Oberfläche mit möglichst wenigen Aktionen schnell bedienen zu können, ist neben der Ausfallsicherheit eine der Hauptanforderungen an diese Systeme. In diesem Zusammenhang sollte die Steuerung

komplett über die Tastatur möglich sein. Die in einem der vorangegangenen Abschnitte besprochenen Vorteile einer kommandozeilenbasierten Oberfläche verschmelzen hier mit den Eigenschaften einer grafischen Schnittstelle.
Platzhalter in Form von Kurzbezeichnern sollen die Eingabe von immer wieder auftretenden Textpassagen umgehen. Ein Beispiel hierzu wäre die Eingabe der Zeichenfolge *SHT*, die dann durch das System in *Schädel-Hirn-Trauma* umgewandelt werden würde.
In ähnlicher Art und Weise existieren Ideen für das Laden von Medikamentendaten oder anderen praxisrelevanten Informationen: ein sogenanntes *Word Wheel* auf Textfeldern prüft nach jedem eingegebenen Zeichen, ob der bisher geschriebene Wortbestandteil zu einer in der Datenbank existierenden Bezeichnung paßt und schlägt diese dann vor. Durch Drücken von *Enter* kann ein Begriff dadurch schon vor seiner vollständigen Eingabe per Tastatur ausgewählt und so ebenfalls ein gewisser Anteil von Eingabearbeit vermieden werden.

Konfiguration Die Arbeit mit Textkürzeln bzw. Kurzbezeichnern impliziert auch ein hohes Maß an individueller Konfiguration. Abhängig von einem bestimmten Nutzer, von der Art der Arztpraxis sowie den gegebenen rechentechnischen Darstellungsmöglichkeiten sollte die Oberfläche in hohem Maße konfigurierbar gestaltet werden. Dies betrifft neben der Festlegung eigener *Shortkeys* besonders auch die Einstellung von Art und Umfang der permanent auf dem Bildschirm sichtbaren Informationen.

Strukturierung der Informationen Patientendaten sollen in unterschiedlichen Sichten darstellbar sein. Baumartige Strukturen sowie Diagramme bergen Vorteile gegenüber der konventionellen Darstellung von Patientendaten in Form einer Liste. In einer hierarchisch aufgebauten Struktur kann entschieden werden, bis zu welchem Detailgrad Daten präsentiert werden sollen – Baumknoten ohne Relevanz bleiben geschlossen.
In Balkendiagrammen können zeitliche Vorgänge, wie die Dauer einer Behandlung, optimal dargestellt werden. Auch dieser Fakt soll maßgeblich die Übersichtlichkeit der Daten erhöhen und so wiederum einer schnelleren Abarbeitung von Aufgaben entgegenkommen.

Optische Orientierungshilfen und permanent sichtbare Daten Wichtige Informationen schnell zu finden, ist ein weiterer Punkt, dem moderne Benutzerschnittstellen im medizinischen Umfeld Rechnung tragen müssen. Einem Penicillin-Allergiker gegenüber zu sitzen, sollte dem behandelnden Mediziner durch angemessene und deutliche Anzeige dieser Information stets bewußt gemacht werden. Die permanente Darstellung von sensiblen, diagnose- und therapieentscheidenden Daten hat also eine hohe Priorität.

2.4 Realisierung grafischer Nutzerschnittstellen

Um die Betrachtungen zu Präsentationsschichten abzurunden, soll zuletzt noch auf einige Technologien zur Realisierung dieser Schnittstellen eingegangen werden.
Genauso vielgestaltig wie das Aussehen, sind auch die Möglichkeiten der technischen Umsetzung grafischer Oberflächen, je nach Umfeld, Zweck und technologischer Infrastruktur. Betrachtet man allein PC-basierte Anwendungen, stößt man auf eine Reihe von Technologien, die bei der Entwicklung von modernen Softwaresystemen nicht mehr weg zu denken sind. An dieser Stelle werden nur kurz einige der populärsten Realisierungsmöglichkeiten besprochen. Eine Separation wird dabei in *Standalone Client-Server* und verteilte bzw. *web-basierte Systeme* vorgenommen. Dennoch ist diese Trennung in sich heterogen, da neben speziellen betriebssystemspezifischen Oberflächen, die einem de-facto-Standard genügen, auch auf Programmierschnittstellen zur Realisierung derselben eingegangen wird. Diese Betrachtung erhebt keinen Anspruch auf Vollständigkeit.

2.4.1 Standalone Client-Server Systeme

Motif: Dies ist eine grafische Benutzeroberfläche, die auf X-Windows[2] aufbaut und durch die Open Software Foundation festgelegt wurde. Kern ist der Motif Window Manager, zu dessen Aufgaben die Zuordnung des Eingabefokus, der Aufbau von Farbtabellen, Kontrolle

[2] X-Windows ist ein vom Massachusetts Institute of Technology (MIT) entwickeltes netzwerkfähiges Fenstersystem auf Client-Server-Basis.

der Position und Größe von Fenstern sowie die Bereitstellung eines Systemmenüs zählen. Motif findet unter Unix breite Anwendung.

Macintosh Finder: Die Computer der Macintosh-Familie werden ausschließlich von der Firma Apple Computer hergestellt. Das Desktop-System *Finder* war eines der ersten grafischen Oberflächen und führte die Fenstertechnik noch vor Windows ein.

Microsoft Foundation Classes (MFC): MFC bezeichnet eine Klassenbibliothek in C++, die heute von nahezu allen größeren Windows-Programmen genutzt wird. Sie stellt eine Programmierschnittstelle bereit, die neben der Implementation von Microsoft-Standard-Konzepten wie COM/DCOM auch die Umsetzung von grafischen Nutzeroberfächen im Windows *Look-And-Feel*[3] ermöglicht

Qt: Qt ist ein von der Firma TrollTec entwickeltes plattformübergreifendes Toolkit basierend auf C++. Es wurde als *Widget*[4]-Set für X-Windows Systeme entworfen. Mit seiner Entwicklung wollte man die Komplexität und Fehleranfälligkeit der Motif-Programmierung umgehen. Da auch die MFC eine sehr große Anzahl an vielfach unüberschaubarer Funktionalität bereitstellen, wurde Qt auch unter Windows zur Verfügung gestellt.

K-Desktop (KDE): Das auf Qt basierende KDE ist eine der am häufigsten verwendeten grafischen Desktop-Umgebung für Linux. Es ist frei erhältlich (Software und Quelltext), basiert auf X-Windows und arbeitet mit einem eigenen Window Manager.

Java-AWT: Das mit Java 1.0 entstandene *Abstract Window Toolkit* für die Programmierung grafischer Bedienelemente besitzt nur bedingte Möglichkeiten zur Umsetzung moderner Nutzeroberfächen. Zudem ist AWT nicht unabhängig von dem zugrundeliegenden Betriebssystem. Dieser Vorläufer von Swing findet heute hauptsächlich bei der Implementation von Applets Anwendung.

[3] *Look-And-Feel* bezeichnet die Gesamtheit aller Gestaltungsmittel (*Look*) entsprechend vorgegebener Richtlinien (*Style Guides*) sowie die zugehörigen Steuermechanismen (*Feel*) zur Realisierung einer grafischen Benutzeroberfläche.

[4] Widget (*engl.*): grafisches Interaktionsobjekt

Java-Swing: Mit Version 1.2 wurde Swing in den Standardumfang des Java Development Kit (JDK) aufgenommen. Unabhängigkeit vom jeweiligen Betriebssystem sowie die Anpassung an beliebige *Look-And-Feel*-Varianten heben sich als klare Vorteile dieser Bibliothek hervor. Neue grafische Komponenten wie Tabellen, Statusanzeigen sowie Trees zur Darstellung baumartiger Datenstrukturen ermöglichen die Erstellung moderner Oberflächen. Java Swing wird zur Programmierung von *Res Medicinae* verwendet.

2.4.2 Web-basierte Systeme

Statisches HTML: Die Hyper Text Markup Language bezeichnet ein Datenformat zur Darstellung von Informationen über das Internet. Statische HTML-Seiten verändern ihr Aussehen nicht. Sie werden durch einen Webserver mit Hilfe des HTTP-Protokolls an den Klienten gesandt und in dessen Bowser dargestellt. Mit reinem HTML, das lediglich zur Darstellung von Text entwickelt wurde, besitzt man nur bedingte Möglichkeiten, grafisch anspruchsvolle Internetseiten zu erstellen.

Applets: Diese kleinen Java-Programme können in Webseiten integriert werden. Java-Applets werden von jedem Web-Browser ausgeführt, der in der Lage ist, den Java-Bytecode[5] zu interpretieren und in Maschinensprache umzusetzen. Durch Applets erweitert man die Multimediafähigkeit von Internetseiten und die Möglichkeit der Interaktion mit ihnen.

Servlets: Während Applets clientseitig zum Einsatz kommen, befinden sich Servlets auf dem Web-Server. Sie bearbeiten Anfragen des Klienten und generieren als Antwort darauf dynamischen HTML-Code. Meist greifen Servlets auf eine Datenbank zu und können zudem die gesamte Breite der Java-Funktionalität nutzen.

Active Server Pages (ASP): Durch Einbinden von Skripten (VB-Script, J-Script) werden HTML-Seiten dynamisch erzeugt. Da die Skripte in ASP- Seiten (*.asp) durch den Server abgearbeitet werden, kann jeder Browser mit ASP arbeiten - unabhängig davon, welche Skript-Sprache er unterstützt.

[5] Java-Bytecode ist ein plattformunabhängiger Code, der vom Java Compiler erstellt und vom Java Interpreter (Laufzeitumgebung) ausgeführt wird.

Java Server Pages (JSP): Java Server Pages erzeugen auf Template-Basis ähnlich den Servlets dynamisches HTML und trennen dabei die Code-Abschnitte von HTML und Java. Sie ähneln in ihrer Struktur stark den ASP, benutzen statt einer Skript-Sprache jedoch Java.

XML/XSL: Die Extensible Markup Language bietet im Gegensatz zu reinem HTML die Möglichkeit, Informationen unabhängig von ihrer konkreten Präsentation zu betrachten. XML-Dateien enthalten lediglich die darzustellenden Daten; die entsprechende Formatierung dieser wird durch XSL (Extensible Style Language) vorgenommen. Durch simples Austauschen von XSL-Stildateien sind unterschiedlichste Präsentationen ein und desselben XML-Datensatzes in verschiedenen Ausgabeformaten, wie zum Beispiel HTML oder PDF, möglich.

PHP Hypertext Preprozessor (PHP): Diese Skript-Sprache für Webserver ermöglicht es relativ schnell und mit wenig Aufwand, dynamische Internetseiten für Multimedia oder E-Commerce-Anwendungen zu realisieren. PHP wird serverseitig in HTML eingebettet und erlaubt standardmäßig den Zugriff auf eine Reihe von SQL-Datenbanken. Die Syntax von PHP ist der von C sehr ähnlich.

Flash: Dieses von Macromedia eingeführte Format gilt in Fachkreisen längst als Standardformat für vektorbasierende Grafiken. Animationen, Sound und eine Vielzahl unterschiedlicher Interaktionsformen bieten dem Webdesigner ein breites Spektrum an Möglichkeiten der Gestaltung web-basierter Oberflächen. Für die Verwendung von Flash in einem Browser wird ein Plug-In benötigt.

Kapitel 3

Muster

Allgemein betrachtet, beschreiben Muster in Architektur, Wirtschaftswissenschaften und eben auch in der Softwaretechnik Lösungen für immer wieder auftretende Probleme, die denselben oder zumindest einen sehr ähnlichen Charakter aufweisen. Es werden keine völlig neuen Lösungen gesucht, sondern auf bereits bekanntes Wissen zurückgegriffen.
Christopher Alexander schreibt: "*Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen.*"[CA77]
Zwar steht diese Definition eines Musters für immer wiederkehrende Entwurfsprobleme an Gebäuden und Städten, jedoch kann sie ebenfalls ohne weiteres auf das objektorientierte Paradigma bezogen werden.
Bezüglich der Abstraktionsebene kann man Muster in drei Kategorien einteilen:

- Idiome
- Entwurfsmuster
- Architekturmuster

3.1 Idiome

Idiome sind programmiersprachenspezifische Muster auf einer niedrigen Abstraktionsebene [FB98]. Ein Idiom beschreibt, wie man bestimmte Aspekte von Komponenten oder Beziehungen zwischen ihnen mit den Mitteln einer bestimmten Programmiersprache implementieren kann. Sie können zur konkreten Umsetzung von Entwurfsmustern verwendet werden, d.h. direkt die Implementierung eines Entwurfsmusters beschreiben.
Ein Beispiel für ein Idiom ist der *Counted-Pointer* [Cop92]. Benutzt man beispielsweise C++ als objektorientierte Programmiersprache, kommt man an dem Thema der Speicherverwaltung nicht vorbei. Dieses Idiom dient der Verwaltung von dynamisch erzeugten, mehrfach referenzierten Objekten in C++. Grundgedanke hierbei ist die Implementierung eines Referenzzählers für eine Rumpfklasse, die von den Zugriffsobjekten aktualisiert wird. Durch dieses Zählen der Referenzen verhindert man zum Einen, dass ein Objekt durch einen Klienten gelöscht wird, obwohl es ein anderer noch referenziert. Zum Anderen wird vermieden, dass nicht mehr referenzierte Objekte als solche unerkannt bleiben, nicht gelöscht werden und somit unnötig Arbeitsspeicher belegen.
Da im Rahmen der vorliegenden Diplomarbeit Idiome eine untergeordnete Rolle spielen, soll die Betrachtung dieses Mustertyps hier abgeschlossen werden.

3.2 Entwurfsmuster

Entwurfsmuster sind Muster auf einer höheren Abstraktionsebene. Nach Gamma [EG96] stellen Entwurfsmuster Beschreibungen zusammenarbeitender Objekte und Klassen dar, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen. Grundsätzlich gliedert sich ein solches Muster strukturell in vier Teile: Zunächst erhält es einen ausdrucksstarken Namen, der das Muster verbal charakterisiert, eindeutig ist, und den Entwicklern zur Kommunikation untereinander dient. Der zweite Teil ist der sogenannte Problemabschnitt, welcher beschreibt, wann das Muster anzuwenden ist. Der dritte Teil eines Entwurfsmusters ist der Lösungsabschnitt, der die Elemente darlegt, aus denen der

Entwurf besteht, sowie die Beziehungen zwischen diesen beschreibt. Als letztes wird noch der Konsequenzenabschnitt angeführt, in dem Vor- und Nachteile einer Verwendung dieses Musters zu finden sind.
Entwurfsmuster sollen die Entwicklung, Wartung und Erweiterung von großen Softwaresystemen unterstützen und sind generell unabhängig von verwendeten Programmiersprachen. Sie sind Denkmodelle zur strukturierten und dementsprechend wiederverwendbaren Lösung ein und desselben Entwurfsproblems. Solche Muster stehen vielerorts in Beziehung zu einander - sie können untereinander verfeinert oder kombiniert werden um komplexere Aufgabenstellungen zu bewältigen.
Man unterscheidet im Wesentlichen zwischen drei Kategorien [EG96]:

- Erzeugungsmuster
- Strukturmuster
- Verhaltensmuster

Diese drei Arten werden im Folgenden kurz erläutert. Jedes der angeführten Beispiele findet auch in der einen oder anderen Form innerhalb des implementierten Prototypen zu *Res Medicinae* Anwendung.

Erzeugungsmuster Diese Entwurfsmuster dienen der Erzeugung von Objekten und sind besonders dann anzuwenden, wenn Systeme mehr von Objektkomposition als von Vererbung abhängen.
Ein Vertreter dieser Kategorie ist die *Abstract Factory* (Abstrakte Fabrik). Mit Hilfe dieses Musters können Objekte zur Laufzeit instanziiert werden, von denen die zugehörige Klasse und Implementierung nicht notwendigerweise bekannt sein muss. Um beispielsweise von unterschiedlichen Betriebssystemen und den damit verbundenen *Look-And-Feel*-Standards unabhängig zu sein, sollte sich eine Anwendung nicht auf spezifische grafische Bedienelemente festlegen. Vielmehr wird angestrebt, unter Verwendung einer *Abstract Factory* die für die jeweilige Plattform notwendigen Elemente zur Laufzeit zu erzeugen.
Die Schnittstelle einer solchen *Factory*-Klasse besitzt für jedes *Look-And-Feel* eine Unterklas-

se und diese enthalten dann jeweils für jedes grafische Bedienelement eine Methode, die das entsprechende Element zurückliefert. Klienten rufen also auf der Super-Klasse Operationen auf, ohne die konkreten Sub-Klassen zu kennen und bleiben so unabhängig vom aktuellen *Look-And-Feel*.

Abbildung 3.1 stellt die *Factory* in UML dar. Es existieren zwei konkrete Unterklassen der abstrakten *Factory*-Klasse: eine zur Realisierung von Windows- und eine weitere zur Umsetzung von OSF-Motif-*Look-And-Feel*.

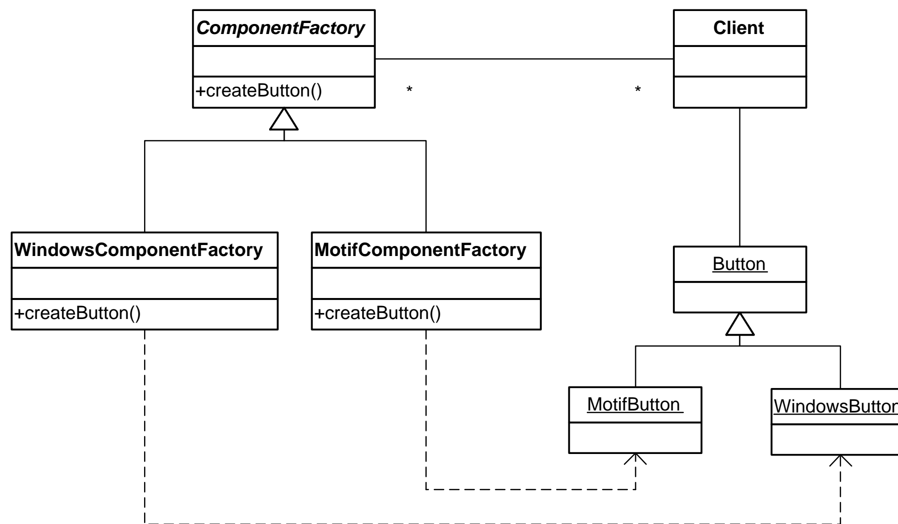

Abbildung 3.1: Klassendiagramm der Abstract Factory

Strukturmuster Wie aus dem Namen bereits hervor geht, dienen diese Muster der Strukturierung von Softwaresystemen. Sie fügen meist primitive Elemente zu komplexen Strukturen zusammen.

Ein Beispiel hierfür ist das *Composite* (Kompositum). Dieses Muster repräsentiert Objekte innerhalb einer Baumstruktur und realisiert auf diese Art und Weise "Teil-Ganzes"-Hierarchien. Ein solches Strukturmuster ist als wichtiges Grundkonzept im gesamten *Res Medicinae*-Framework wiederzufinden (vgl. Abb. 3.2).

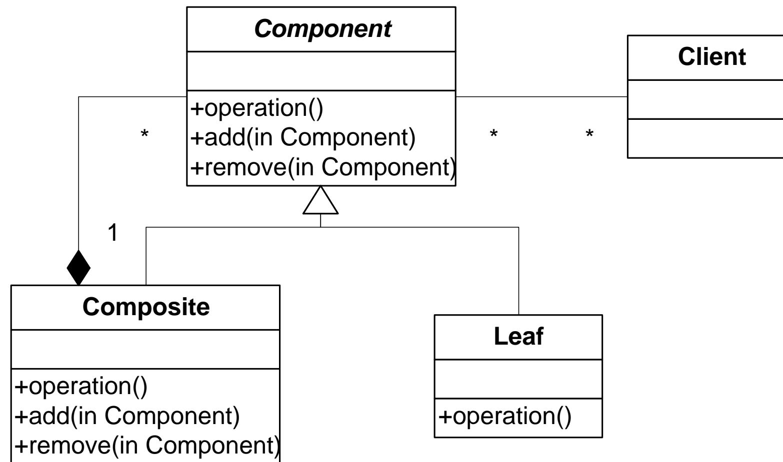

Abbildung 3.2: Klassendiagramm des Composite

Verhaltensmuster Verhaltensmuster können Funktionalität in Form von Algorithmen kapseln oder die Interaktion zwischen mehreren Objekten beschreiben. Bestes Beispiel hierfür ist der *Observer* (Beobachter) (vgl. Abb. 3.3). Er definiert und verwaltet die Abhängigkeiten zwischen Objekten. Eine Änderung des Zustandes eines Objektes bewirkt die Benachrichtigung anderer Objekte, der sogenannten *Observer*. Dieses Muster findet Anwendung bei der Entwicklung von *Model-View-Controller*-basierten Schnittstellen.

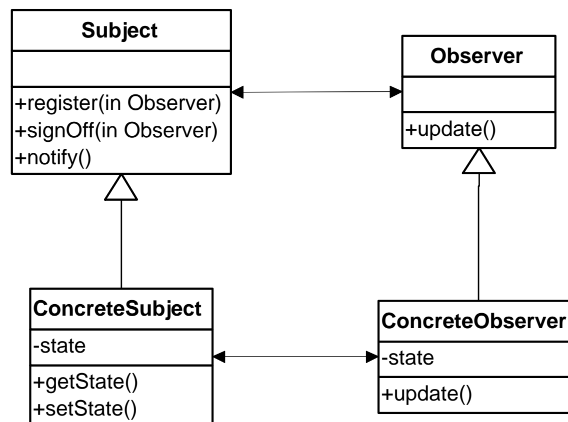

Abbildung 3.3: Klassendiagramm des Observer

3.3 Architekturmuster

Architekturmuster beschreiben die unter allen Mustern abstrakteste Form des Softwareentwurfes. Sie sind Schablonen für konkrete Softwarearchitekturen [FB98] und geben grundsätzliche Strukturprinzipien vor. Wie Entwurfsmuster, können auch Architekturmuster in verschiedene Arten kategorisiert werden. Man unterscheidet zwischen Mustern für:

- Strukturierung
- Verteilung
- Adaption
- Interaktion.

Strukturmuster dienen dazu, Softwarekomponenten logisch zu gliedern, um so eine gewisse Ordnung innerhalb des Systems zu erreichen. Das bekannteste Muster dieser Kategorie ist das *Layer*-Muster, zu dem auch die im ersten Kapitel besprochene *3-Tier*-Architektur gehört. Das Wesen dieses Musters liegt in der sinnvollen Zerlegung der Systemkomponenten in mehrere kooperierende Schichten mit unterschiedlichen Teilaufgaben. Dabei müssen nicht notwendigerweise nur drei Schichten existieren. Je nach Komplexität des Systems sind auch mehrere denkbar. Man spricht daher im Allgemeinen von einer *n-Tier*-Architektur.
Das *Broker*-Muster als *verteiltes Muster* realisiert eine umfangreiche Infrastruktur für verteilte Anwendungen und wurde von der *Object Management Group*[1] (OMG) standardisiert [OMG92].
Eine weitere Kategorie von Architekturmustern sind Muster zur Realisierung von *Adaptierbaren Systemen*. Das *Reflection*-Muster unterstützt zum Beispiel die Erweiterung von Anwendungen und ihre Anpassung an sich ständig ändernde funktionale Anforderungen.
Interaktive Systeme, wie die bereits besprochenen grafische Benutzeroberflächen, werden entweder nach dem *Presentation-Abstraction-Control* oder dem weit verbreiteten *Model-*

[1] Die OMG ist ein 1989 gegründeter Zusammenschluss von mehreren Firmen wie Hewlett-Packard, Philips Telecommunications N.V., Sun Microsystems oder Canon, um eine standardisierte Architektur für verteilte Anwendungen zu schaffen.

View-Controller Muster entworfen. Letzteres wird im weiteren Verlauf dieses Kapitels noch ausführlich behandelt.

Model-View-Controller (MVC) Das mit der Programmiersprache Smalltalk 80 entstandene MVC-Muster ist heute die gebräuchlichste Art, grafische Benutzerschnittstellen umzusetzen. In diesem typischen Architekturmuster wird die logische und funktionale Aufteilung in drei Klassen vorgenommen: *Model*, *View* und *Controller* (vgl. Abb. 3.4). Das Model-Objekt repräsentiert die der Präsentationsschicht zu Grunde liegenden Daten, wel-

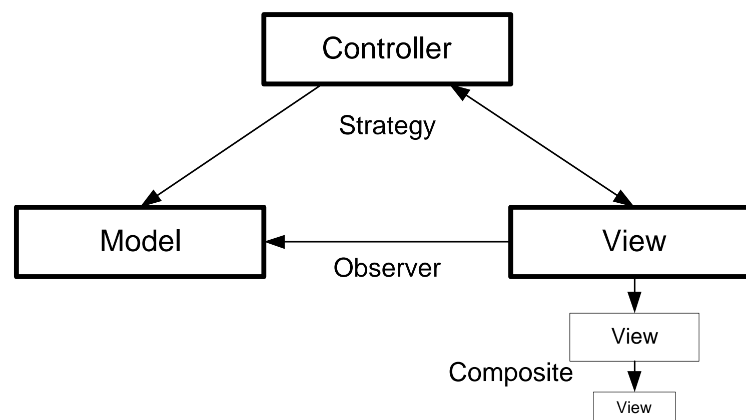

Abbildung 3.4: MVC-Paradigma und darin enthaltene Entwurfsmuster

che durch das View-Objekt dargestellt und über das Controller-Objekt manipuliert werden können. Bevor MVC Bedeutung erlangte, fasste man die drei genannten Bestandteile in einem Objekt zusammen.

Wie eigentlich die gesamte objektorientierte Programmierung sollte auch dieses MVC-Muster Flexibilität und Wiederverwendbarkeit von Software erhöhen. In vielen Veröffentlichungen wird von MVC als einem Entwurfsmuster gesprochen. Gamma [EG96] bezeichnet MVC sogar als ein Paradigma, in dem mehrere Entwurfsmuster Anwendung finden. So gleicht zum Beispiel die Beziehung des View-Objektes zum Model-Objekt der eines *Observers*. Änderungen des Models sollen dementsprechend auch zu Änderungen des Views führen, wobei notwendiger Weise nicht beide Objekte einander bekannt gemacht werden müssen.

Weiterhin kann ein View-Objekt mehrere andere View-Objekte geschachtelt referenzieren (*Composite*, vgl. Abb. 3.2). Primitive Objekte fügen sich in hierarchischer Form zu komple-

xeren Komponenten zusammen. Zum Beispiel könnte ein Anwendungsfenster als ein View-Objekt angesehen werden, das weitere in Form von grafischen Oberflächen-Komponenten wie zum Beispiel Buttons, Textfeldern, Menüs oder Schiebereglern enthält.
Das Controller-Objekt selbst ist ein typisches Beispiel für ein *Strategy*-Muster. Diese Art von Entwurfsmustern kapselt Funktionalität. *Strategy*-Objekte repräsentieren Algorithmen. Eine derartige Kapselung bringt den Vorteil mit sich, Funktionalität der mit Hilfe von MVC implementierten Software bei Bedarf ohne Veränderung der anderen Objekte austauschen zu können. Will man beispielsweise die Arbeitsweise eines Menüpunktes ändern, bedarf dies lediglich der Ersetzung des entsprechenden Controller-Objektes.
Diese drei Muster - *Observer*, *Composite* und *Strategy* - beschreiben die zentralen Beziehungen zwischen den Objekten des *Model-View-Controller*- Paradigmas.
Weiterhin können *Factorys* (vgl. Abb. 3.1) dazu verwendet werden, die Klasse der Controller- oder View-Instanzen zur Laufzeit festzulegen. *Decorators* (Dekorierer)[2] erweitern das View-Objekt um zusätzliche, jedoch von dem Objekt unabhängige, Funktionalität.
Das MVC-Paradigma, welches im Rahmen dieser Diplomarbeit eine übergeordnete Rolle spielt, kann also mit Hilfe einiger weniger Entwurfsmuster detailliert beschrieben werden. Jedoch ist in diesem Zusammenhang auch zu erwähnen, dass die konkrete Umsetzung noch stark abhängig von den zugrundeliegenden Programmiersprachen sowie eingesetzten Frameworks und keinesfalls als statisches und zwischen einzelnen Anwendungen frei portierbares Konstrukt anzusehen ist. Die Trennung von Model, View und Controller ist dabei die einzige Richtlinie. Genauere Designentscheidungen, wie Schnittstellen, Vererbung und Klassenanzahl müssen an das jeweilige Problem angepasst werden.

Hierarchisches Model-View-Controller (HMVC) Wie bereits angedeutet, steht die Weiterentwicklung von MVC keineswegs still. Zwar wird der grundlegende Gedanke immer derselbe bleiben, jedoch existieren bereits Überlegungen, das schon seit den achtziger Jahren des letzten Jahrhunderts existierende Architekturmuster an die komplexen Anforderungen heutiger Benutzeroberflächen anzupassen.

[2] Ein *Decorator* ist ein Entwurfsmuster, das ein Objekt dynamisch um Zuständigkeiten erweitert. Im Allgemeinen sind dies zusätzliche Objekte, die neue Methoden oder Zustände beinhalten.

Das *Hierarchische Model-View-Controller* Muster [JC01] unterteilt eine Benutzeroberfläche und die auf ihr enthaltenen Komponenten logisch in mehrere Schichten (*Layers*). Diese Trennung entspricht dem bereits besprochenen *n-Tier* Strukturmuster. Es existieren mehrere MVC-Triaden[3] innerhalb dieser Schichten, wobei die entsprechenden Controller-Objekte hierarchisch auf einander referenzieren (vgl. Abb. 3.5). Dieser Umstand impliziert den Aufbau von Eltern-Kind-Beziehungen zwischen den einzelnen Komponenten.

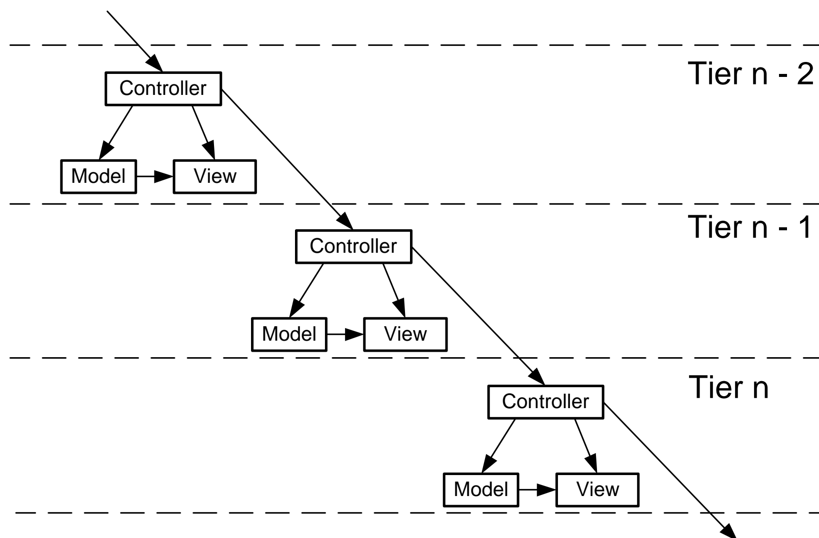

Abbildung 3.5: HMVC-Muster [JC]

An einem praktischen Beispiel erläutert, sähe eine solche Anordnung wie folgt aus: Auf oberster Ebene (*Tier*) befindet sich ein *Top-Level-Container* wie beispielsweise ein *Dialog*. Auf der darunterliegenden Schicht, d.h. in diesem Dialog selbst, existiert ein weiterer *Container*, wie z. B. ein *Panel*. In diesem wiederum ist eine Navigationsleiste plaziert. *Dialog*, *Panel* und Navigationsleiste entsprechen jeweils einer MVC-Triade bestehend aus Model-, View- und Controller-Objekt. Über den Controller erfolgt nun entweder die direkte Verarbeitung von Ereignissen oder die Weiterleitung an das Controller-Objekt der darunterliegenden Schicht. So entsteht eine *Chain of Responsibility* (Kette von Zuständigkeiten), in der Ereignisse bis an den Ort ihrer endgültigen Auswertung geleitet werden. Ist ein Controller nicht in der Lage, auf ein Ereignis zu reagieren, nimmt er die Weiterleitung vor.

[3] Eine Triade ist eine Gruppe bestehend aus drei Objekten, die in Relation zu einander stehen.

Neben der herkömmlichen Separierung der Komponenten in die drei Objekte des MVC, gelingt durch das *Hierarchische Model-View-Controller* Muster die Einführung einer weiteren Dimension in der Strukturierung von Oberflächenelementen und ihren Beziehungen zueinander. Die Vorteile des Architekturmusters *Layers* sowie des Entwurfsmusters *Chain of Responsibility* nutzend, verspricht man sich durch den Einsatz von HMVC vor allem eine Verringerung der Abhängigkeiten zwischen einzelnen, austauschbaren Programmelementen sowie die einfache Erweiter- und Wartbarkeit solcher Implementierungen.

Fazit Es wurde gezeigt, dass sowohl View-, als auch Controller-Objekte einer logischen Hierarchie unterworfen werden können. Auch die Model-Komponente kann ohne weiteres eine solche Struktur aufweisen: die dem MVC zugrundeliegenden Daten bestehen in den meisten Fällen aus kompositiv zusammengesetzten Objekten.
Diese Überlegungen implizieren folgende Idee: Die generalisierte Verwendung des *Composite*-Musters auf oberster Ebene der Klassenhierarchie garantiert, dass alle Objekte in ein Hierarchie von "Teil-Ganzes"-Beziehungen eingegliedert werden können, ohne dass jede Klasse diese Eigenschaften selbst bereitstellen und redundant implementieren muss. Eine Klasse namens *Item* verwirklicht dieses Muster als Wurzelklasse des Frameworks von *Res Medicinae*. Auf beide – Framework und *Item* – wird in Kapitel fünf eingegangen.

Kapitel 4

Komponentenbasierte Softwareentwicklung

Das Wort *Softwarekomponente* ist ein sehr abstrakter Begriff der Softwaretechnik. Softwarekomponenten können Anwendungen, Module oder einfach nur – im Sinne der objektorientierten Programmierung – Entitäten darstellen, die Daten sowie Funktionalität kapseln.
Einfache Instanzen des objektorientierten Paradigmas stehen stets in Beziehung zueinander. Sie können miteinander über Nachrichten kommunizieren und sich auf diese Weise gegenseitig beeinflussen. Ein großer Nachteil besteht aber in den Abhängigkeiten, die zwischen diesen Instanzen entstehen können. Dadurch sind sie nicht mehr einfach ersetzbar.
Komponentenbasierte Softwareentwicklung geht deshalb einen Schritt weiter. Eine einfache und umfassende Kontrolle der Objektreferenzen wird angestrebt. Komponenten sollen die Eigenschaft der einfachen Ersetzbarkeit besitzen und somit die objektorientierte Programmierung bzw. Modellierung ihren eigentlichen Zielen von Intuitivität, Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit näher bringen.

4.1 Der Komponenten-Lebenszyklus

Die Natur ist der beste Lehrer. Über Jahrmillionen entstandene, sich durch Selektion und Mutation stetig verbessernde Anpassungen des Lebens an sich ständig verändernde Umwelteinflüsse bieten hervorragende, wenn auch sehr komplexe Vorlagen für technische Systeme. Schon seit Jahrzehnten versucht man auf dem Gebiet der Neuroinformatik, Kybernetik und

der biomedizinischen Technik dynamische Vorgänge in der Natur mathematisch zu erfassen und die "Ideen" der Natur auf diese Art und Weise für den Menschen nutzbar zu machen. Neuronale Netze sind wohl die besten Beispiele für solche künstlichen Systeme. Aber auch die Automatisierungstechnik und Fuzzy-Logik nehmen die Natur zum Vorbild und versuchen durch selbstregelnde Prozesse solche Konzepte technisch zu verwirklichen.
Angelehnt an den natürlichen Lebenszyklus von organischen Zellen kann man auch Softwarekomponenten einem ähnlichen Kreislauf unterwerfen. Diese Vorgehensweise verfolgt das Ziel, die oft sehr komplexe Dynamik von Software besser beschreiben und definierte Abhängigkeiten zwischen Komponenten effizienter erkennen und nachbilden zu können.

Grundlegende Idee Organische Zellen enstehen in einem komplexen Teilungsprozess. Dabei übernehmen sie das komplette Erbmaterial der Elternzelle. Dieser Umstand impliziert, dass Zellen ihre meisten Eigenschaften über unzählige Generationen hinweg vererbt bekommen. Zudem werden auch neue Eigenschaften entsprechend der jeweilig vorhandenen Umgebung neu erworben. Der Zelle ist es somit möglich, sich anzupassen.
Während ihres Lebens wächst die Zelle, es bilden sich entsprechend der übergebenen Erbinformation alle notwendigen Organellen aus. Die Zelle ist ab diesem Zeitpunkt voll lebensfähig und - was gerade im Zellverband von grosser Bedeutung ist - sie ist funktionsfähig, kann die an sie gestellten Aufgaben erfüllen. Hat sie ein gewisses Alter erreicht bzw. ihren Nutzen im Zellverband eines Organismus' erfüllt, stirbt sie und der Kreis schliesst sich.

Modellierung Diese Idee ist nun Vorbild für den Lebenszyklus von Softwarekomponenten [ava].
Jede Komponente wird innerhalb der sie beinhaltenden Umgebung diesem Kreislauf unterworfen. Methoden, die auf den Komponenten aufgerufen werden, bestimmen den Ablauf des Zyklus und damit auch die Reihenfolge der Aktionen. So existieren Operationen für das Anlegen, die Initialisierung, Konfiguration und die Bedienung, aber auch für das Löschen bzw. Verwerfen von Komponenten.
Das folgende Bild (siehe Abb. 4.1) zeigt im Stile eines Zustandsübergangsgraphen den Le-

benszyklus einer solchen Software-Komponente.

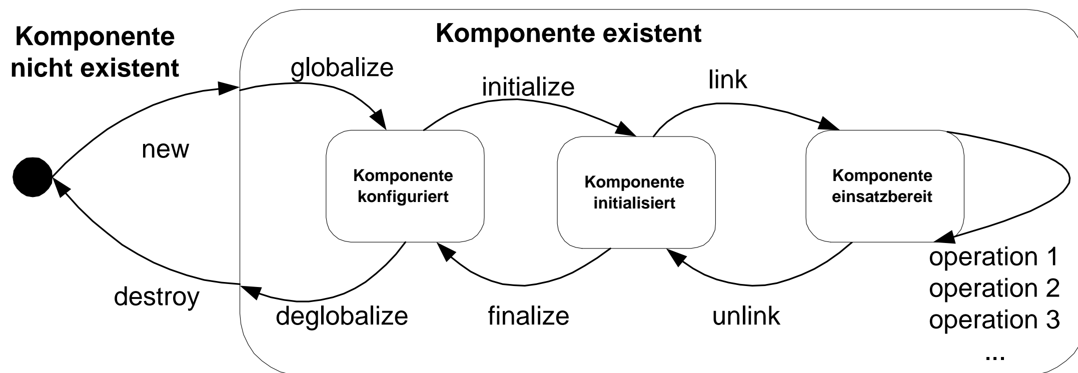

Abbildung 4.1: Abstrakter Lebenszyklus von Komponenten

Prinzipiell existieren zwei elementare Grundzustände, die eine Komponente annehmen kann: *nicht existent* und *existent*. Vom Zustand *Komponente ist nicht existent* wird die Komponente durch Konstruktoraufrufe in den Zustand *Komponente existent* gebracht. In diesem Status ist es möglich, ihr grundlegende Eigenschaften über Konfiguration, Initialisierung und Komposition zu zuweisen. Die dabei verwendeten Methoden werden `globalize`, `initialize` und `link` genannt und in dieser Reihenfolge ausgeführt. Durch diese Anwendung der Operationen gelangt die Komponente wiederum in Unter-Zustände, welche sie zunächst als konfiguriert, dann initialisiert und zuletzt voll einsatzfähig charakterisieren. Nun ist die Komponente bereit, genutzt zu werden, also ihren Zweck innerhalb der Umgebung zu erfüllen. Die sich jetzt anschließenden Operationen hängen von der spezifischen Aufgabe der Komponente innerhalb des Systems ab.

Besteht nach einer bestimmten Zeit keine Verwendung mehr für die Komponente, wird sie durch `unlink`, `finalize` und `deglobalize` dekonfiguriert und kann durch einen Destruktionsoperator wieder gelöscht werden. Dieser letzte Umstand macht zudem *Garbage Collection*[1] überflüssig, da das Beseitigen von Objekten explizit überwacht werden kann.

Tabelle 4.1 stellt diese Schritte noch einmal in chronologischer Reihenfolge dar, wobei in der letzten Spalte die zu jedem Methodenaufruf gehörenden natürlichen

[1] Als *Garbage Collection* bezeichnet man das Finden und Löschen von Objekten, die nicht mehr referenziert werden. Dieser Prozess ersetzt in Java die aktive Anwendung von Destruktoren durch den Programmierer.

Tabelle 4.1: Operationen des Komponentenlebenszyklus'

Schritt	Operation	Erläuterung	Natürlicher Vergleich
1	`new`	Konstruktoraufruf	Zellteilung (Mitose), Zellentstehen
2	`globalize`	Übergabe globaler Konfigurationsparameter	Weitergabe der DNS an Kindzellen
3	`initialize`	Erzeugung von abhängigen Unterobjekten	Ausbildung von Zellorganellen (Zellkern, Plasma usw.)
4	`link`	Assoziationen zwischen Unterobjekten	Bindung in Zellverband
5	`operation`	Komponentennutzung	Aufgabenerfüllung der Zelle im Verband, (z.B. Transport von Sauerstoff durch rote Blutkörperchen, Verarbeitung von Signalen in Nervenzelle)
6	`unlink`	Lösen der Assoziationen	Zellverfall, Sterben der Zelle
7	`finalize`	Beginn der Zerstörung von Unterobjekte	
8	`deglobalize`	Trennung aus Gesamtsystem	
9	(`delete`)	Destruktoraufruf, existiert nicht in allen OO-Programmiersprachen	Zelltod

Vergleiche dargelegt werden. Dies soll dem Verständnis des Lebenszyklus' einer Komponente dienen.

4.2 Weiterführende Konzepte und Ausblick

4.2.1 Separation of Concerns

Erste Versuche, den Lebenszyklus von Komponenten im Verlauf der Realisierung von *Res Medicinae* zu gestalten, gingen in die Richtung der bereits bekannten Trennung von Zuständigkeiten (*Separation of Concerns*). Sollten Komponenten befähigt werden, eine bestimmte Aufgabe abarbeiten oder Funktionalität im Komponentensystem anbieten zu können, war es notwendig, eine ensprechende Schnittstelle für diese Komponenten einzurichten und durch eine konkrete Klasse zu implementieren. Dementsprechend gab es für die unterschiedlichsten Belange (*Concerns*) verschiedenste Schnittstellen. So existierten Interfaces zur Initialisierung, Kopplung, Trennung sowie Zerstörung von Komponenten. Vorteil dieser Idee war es, Objekte zur Laufzeit nicht direkt über ihre Klassen, sondern die entsprechenden zuständigen Schnittstellen ansprechen zu können, um so eine gewisse Unabhängigkeit von der konkreten Implementierung zu erreichen.

4.2.2 Aspektorientierte Programmierung

Aspektorientierte Softwareentwicklung vermeidet, Verantwortlichkeiten von Klassen in ihnen selbst zu behandeln, sondern nimmt eine Modellierung über sogenannte *Aspekte* vor [Fix01]. Diese sind klassenähnliche Strukturen, die Zuständigkeiten an bestimmte Objekte delegieren. Objekte bzw. ihre Klassendefinitionen dienen in Form von Entitäten lediglich der Datenkapselung und Aspekte der Bereitstellung der zugehörigen Logik. Dieser sehr junge Ansatz, bislang nur in der aspektorientierten Erweiterung von Java (AspectJ) umgesetzt [asp02], versucht, die Modellierungsschwächen von objektorientierten Sprachen zu beheben. Diese Treten insbesondere dann auf, wenn es darum geht, Konzepte zu realisieren, die die

Klassenhierarchie durchschneiden, wie beispielsweise *Tracing*[2] oder *Logging*[3].

4.2.3 Ausblick

Beide Ideen, *Separation of Concern* und *Aspektorientierte Programmierung* wurden zwar während des Entwurfes von *Res Medicinae* untersucht, fanden jedoch keine praktische Umsetzung. Hierfür einige Gründe:
Die auf der Verwendung von *Separation of Concern* basierende Aspektorientierte Programmierung erschien für die Arbeit an diesem Projekt ungeeignet, da die Klassenhierarchie durchschneidende Konzepte zum Großteil durch die Einführung des Komponentenlebenszyklus verwirklicht werden können. Objekte werden durch diesen erzeugt, initialisiert und auf die Verwendung vorbereitet. Ebenso denkbar wäre also auch, *Tracing*, *Logging* oder andere *Concerns* über entsprechende Lebenszyklus-Methoden zu realisieren. Ein anderer Punkt ist der zusätzliche, recht gewöhnungsbedürftige "Aspekt"-Code im Quelltext: Zwar amortisiert sich der relativ hohe Lernaufwand bei mittleren und grossen Projekten mit umfangreichen Concerns, jedoch beschränkt die Anzahl möglicher *Join Points*[4] wiederum die relevanten Einsatzmöglichkeiten. Die alleinige Umsetzung dieser Idee in Java (AspectJ) verhindert weiterhin die Portierbarkeit zwischen Programmiersprachen.
Ein weiterer Grund gegen ein Verwendung von *Concerns* liegt in der speziellen Struktur des Frameworks von *Res Medicinae*: Sein hierarchischer Charakter basiert auf der Implementation von Klassen, die im Stile eines Schichtensystems an die Granularität der Anwendung angepasst sind. Der Vorteil von *Concerns* liegt in der Möglichkeit, Objekte zur Laufzeit statt über ihre konkrete Klassen, über Schnittstellen (für die jeweilige Aufgabe vorgesehene Signaturen) verwenden zu können. Diesen Umstand kann man aber auch durch die Wahl einer streng hierarchischen Klassenstruktur erzielen, innerhalb der die Instanzen über die Schnittstelle einer (Ober)-Klasse angesprochen werden.
Eine genauere Erläuterung dieser Idee folgt im nächsten Kapitel.

[2] Protokollierung technischer Vorfälle, wie Fehlermeldungen oder Ausnahmen
[3] Protokollierung aufgetretener fachlicher Ereignisse, wie Zugriffe und Transaktionen.
[4] Join Points sind wohldefinierte Punkte im Ablauf eines Programmes, wie z. B. Zuweisungen und Referenzierungen, Methodenaufrufe oder der Empfang von Nachrichten.

Kapitel 5

Frameworks

In diesem Kapitel wird das Konzept der Frameworks vorgestellt und deren typische Eigenschaften sowie die damit verbundenen Vor- und Nachteile dargelegt. Das Softwareprojekt *Res Medicinae* besitzt ebenfalls ein solches "Grundgerüst" namens *ResMedLib*, auf das im letzten Abschnitt des Kapitels eingegangen wird.

5.1 Einordnung

In der Softwaretechnik versteht man unter einem Framework eine Menge untereinander kooperierender Klassen, die Elemente eines wiederverwendbaren Entwurfes darstellen [EG96]. Sie sind unvollständige Softwaresysteme, die durch Instanziierung und Spezialisierung ihrer Klassen von einem abstrakten Status in den Zustand einer funktionsfähigen und ausführbaren Anwendung gehoben werden.
Ein Framework legt die Grundarchitektur für eine Familie von Anwendungen fest. So existieren beispielsweise Frameworks für grafische Darstellungsprogramme, die einmal für die Implementierung eines Bildbearbeitungsprogrammes und ein anderes Mal für die Realisierung einer CAD[1]-Anwendung eingesetzt werden können.
Nach Pree [Pre94] besteht ein Framework aus statischen und veränderbaren Bereichen.

[1] Unter CAD (*Computer Aided Design*) werden Konstruktionsanwendungen für Maschinenbau, Bauingenieurwesen usw. zusammengefasst.

Statische Elemente eines Frameworks bleiben in jeder Instanziierung gleich. Sie bilden eine Art Grundstruktur einer jeden von dem Framework abgeleiteten Anwendung. Dem gegenüber stehen die veränderlichen Teile eines Frameworks, die für das jeweilige Softwaresystem spezifisch sind. Diese werden entsprechend den Bedürfnissen und Anforderungen an das System spezialisiert. Die Flexibilität eines Frameworks wird dabei nicht nur durch die Anwendung von objektorientierten Prinzipien wie Vererbung oder Polymorphie erreicht - viele Frameworks implementieren *Factory*- oder Befehlsmuster, um eine Anpassung zu ermöglichen. Oft werden gerade Entwurfsmuster als elementare Bausteine von Frameworks verstanden.

Ein Framework besitzt einen "umgekehrten" Aufrufmechanismus. Verwendet man eine herkömmliche Bibliothek, die bestimmte Funktionalitäten bereitstellt, so nutzt man diese, indem man die betreffenden Klassen bzw. die von ihr bereitgestellten Methoden aufruft. Der Hauptteil der Anwendung wird also vom Programmierer entworfen und realisiert, spezielle Operationen nach dem Prinzip der Wiederverwendung genutzt. Frameworks hingegen stellen den eigentlichen Hauptteil eines Systems dar. Die vom Entwickler programmierten Teile werden durch das Framework aufgerufen und in den Dienst der Anwendung gestellt.

5.2 Vor- und Nachteile

Frameworks gewinnen zunehmend an Bedeutung. Objektorientierte Systeme erreichen durch ihren Einsatz einen sehr hohen Grad an Wiederverwendbarkeit. Sie bestimmen Entwurfsparameter, um die sich der Entwickler nicht mehr kümmern muss. Vielmehr kann sich sein Augenmerk auf die Umsetzung der spezifischen Details der zu entwickelnden Anwendung richten. Zwar muss sich der Programmierer an vorgegebene Operationsschnittstellen halten, jedoch befreit ihn dieser Umstand von möglicherweise langwierigen Entwurfsentscheidungen. Programme können so entschieden schneller entwickelt werden. Desweiteren besitzen Anwendung, die mit Hilfe ein und desselben Frameworks implementiert wurden, ähnliche Strukturen, was dem Aspekt der Konsistenz und Wartung entgegen kommt.

Nachteilig ist aber, dass Anwendungen der Evolution des zugrundeliegenden Frameworks unterworfen sind. Auch der durch sie verursachte Einarbeitungsaufwand ist nicht zu unterschätzen – jedoch kann er durch sinnvollen Einsatz von Entwurfsmustern verringert werden. Dieser Fakt impliziert wiederum ein hohes Maß an Dokumentation.
Frameworks sind weniger abstrakt als Entwurfsmuster. Sie können in den Code einer Programmiersprache überführt werden und somit im Gegensatz zu Entwurfsmustern nicht nur theoretisch untersucht, sondern auch praktisch angewandt und getestet werden.

Im Anschluss an diese theoretischen Ausführungen soll nun auf ein konkretes Beispiel eingegangen werden.

5.3 ResMedLib Framework

5.3.1 Grundlegende Idee

Eine sogenannte *Ontologie* als strenge Hierarchie von Typen bildet die Grundlage des *Res Medicinae*-Frameworks *ResMedLib*. Das aus dem Griechischen stammende Wort *Ontologie*[2] wird in vielen Veröffentlichungen unterschiedlich gebraucht und auf die verschiedensten Arten definiert. Im Kontext dieser Arbeit steht der Begriff für die **systematische** Beschreibung von komplexen fachlichen Zusammenhängen unter Einführung einer konkreten **Terminologie** und entsprechender Begriffsbestimmungen.
Auch hier wurde wieder einmal die Natur zu Rate gezogen. Ausgehend von der einfachen Idee, jedes denkbare Element des Universums in die Hierarchie eines "Großen Ganzen" einordnen zu können, ist bei der Konzeption dieses Frameworks eine Strukturierung vorgenommen worden, die zwischen den Komponenten eine strikte "Teil-Ganzes"-Beziehung vorsieht. Durch diese Aufteilung steigt die Granularität der Komponenten von Ebene zu Ebene.

---

[2] Der Begriff *Ontologie* setzt sich zusammen aus *ontos* (Sein) und *logos* (Wort) und diente in der Antike als Bezeichnung für die *Wissenschaft des Seienden* oder *Lehre von der Natur der Realität*.

5.3.2 Struktur

Die oberste Wurzelklasse *Item* spielt innerhalb des Frameworks eine besondere Rolle. Sie stellt elementare Eigenschaften und Operationen bereit, die durch Vererbung an jede andere Klasse des Frameworks weitergegeben werden. In der momentan aktuellen Version sind noch viele Klassen direkte Subtypen von *Object*[3]. *Item* bildet als zukünftig einzig erlaubte direkte Unterklasse von *Object* die Wurzel in einer hierarchischen Klassenstruktur (vgl. Abb. 5.1). Diese bislang nur teilweise umgesetzte Idee soll die Grundlage für eine spätere Anwendung des Frameworks auf alle Klassen des JDK schaffen.

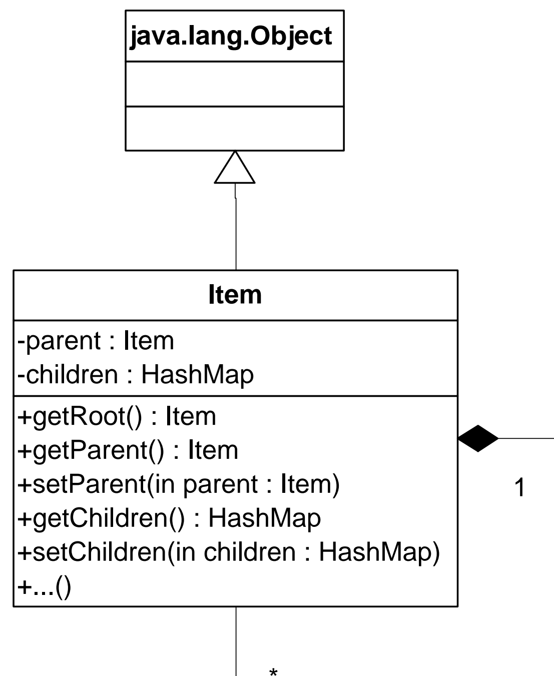

Abbildung 5.1: Klasse Item

Eine der wichtigsten Eigenschaften ist hierbei die Möglichkeit, jedes Objekt von *Item* zu einem Bestandteil eines Baumes bestehend aus einer beliebigen Anzahl von Instanzen der Klasse *Item* werden zu lassen. Dies entspricht der Realisierung des *Composite*-Musters, wie es in Abschnitt 2.2 besprochen wurde.

[3] *Object* ist die Superklasse jeder anderen Java-Klasse des Java Development Kits und aller weiteren Bibliotheken.

Alle Klassen innerhalb des Frameworks sind direkte oder indirekte Subtypen von *Item* und ihrerseits wiederum in hierarchische Schichten gegliedert. Diesem Modell entsprechend setzen sich Objekte von Klassen aus einer oberen Schicht immer aus Instanzen derselben oder darunterliegender Schichten zusammen.

Die Idee, ein Framework als geordnete Baumstruktur zu gestalten, impliziert eine einfache Kontrolle der existierenden Objektreferenzen. Diese Aussage stützt sich auf folgende Überlegungen:

Ein Baum ist in der theoretischen Informatik nichts anderes, als ein Graph mit speziellen Eigenschaften. Die wichtigste ist dabei die *Kreisfreiheit*[4]. Eine weiteres Merkmal ergibt sich aus der Kreisfreiheit selbst: Besitzt ein Baum *m* Knoten, so enthält er exakt *m-1* Kanten. Im Kontext dieses Dokumentes sind also die Knoten Objekte und Kanten die Referenzen bzw. Assoziationen zwischen diesen.

Entlang jedem Pfad dieses Baumes sind die Operationen des Lebenszyklus, wie sie in Abschnitt 3.1 beschrieben wurden, strikt anzuwenden. Die dazu benutzten Assoziationen stellen dementsprechend **existenzielle Eltern-Kind-Relationen** zwischen diesen Objekten dar. Äste[5] dieses Baumes entstehen durch Konstruktion neuer Objekte oder ganzer Komponenten. Ein Kindobjekt wird immer nur durch ein Elternobjekt erzeugt und nimmt so am Lebenszyklus der Komponenten teil. Nicht zuletzt basiert das Entfernen von Ästen dieses Baumes auf demselben Prinzip: Komponenten, die nicht mehr benötigt werden, treten aus dem Lebenszyklus aus, indem sie durch ihre Elternkomponenten zerstört werden.

In Abbildung 5.2 wird im linken Baum durch *Kind 2* ein neues Objekt erzeugt (Konstruktoraufruf `new`). Nach der Übergabe von Konfigurationsparametern durch `globalize` und der Initialisierung durch `initialize` werden mit `link` Assoziationen zu Objekten derselben Ebene erzeugt (siehe Assoziation 1, Abb. 5.2). Dabei ist zu beachten, dass Assoziation 1 **keine** existentielle Eltern-Kind-Relation darstellt, und daher nicht das Prinzip der Kreisfreiheit verletzt. Auf diese Art und Weise enstehen neue Äste des Baumes.

Der beschriebene Vorgang kann auch wieder rückgängig gemacht werden, indem die Assoziation zwischen dem Elternobjekt *Kind 2* und dem Kindobjekt *Kind 4* entfernt wird. Zuvor

[4] Ein Graph ist kreisfrei, wenn zu jedem Knoten genau ein Weg existiert. Dieser ist dann auch gleichzeitig der kürzeste.

[5] Ein Ast ist eine Aneinanderreihung von Objektreferenzen.

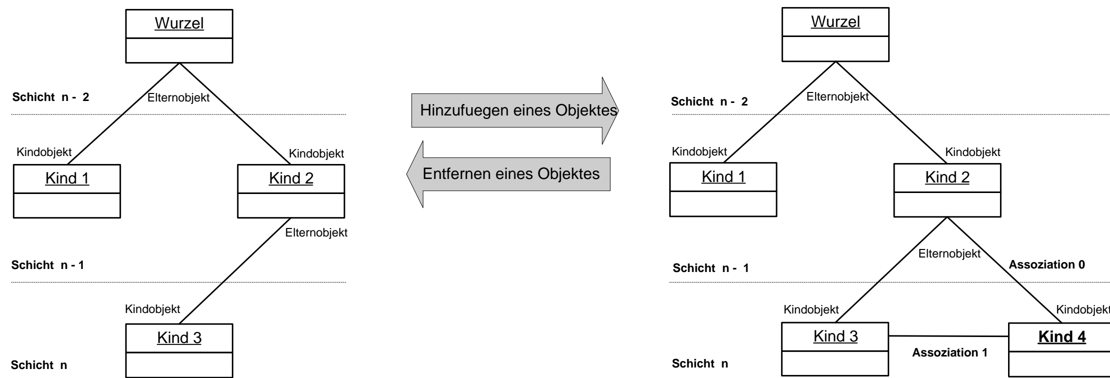

Abbildung 5.2: Objektbeziehungen in Res Medicinae

werden die Operationen zur Beseitigung der existierenden Objektbeziehungen aufgerufen (`unlink`, `finalize` und `deglobalize`). Dabei entfernt `unlink` im Speziellen die Referenz auf *Kind 3* (Assoziation 1). Durch das finale Beseitigen der Assoziation 0 existiert nun keine Referenz mehr auf *Kind 4*, was dazu führt, dass dieses Objekt aus dem Speicher entfernt wird (z. B. in Java durch *Garbage Collection*). Diese beiden Eigenschaften, Separation in eine Hierarchie von Schichten und Komposition durch strikte Einhaltung einer Baumstruktur sind die charakteristischen Merkmale einer Ontologie.
Im Rahmen des *Res Medicinae*-Frameworks unterscheidet man zwischen insgesamt drei solcher Ontologien [Hel02]:
Mit der *System-Ontologie* ist der Entwurf von Softwaresystemen und der in ihnen vorhandenen Funktionalität bzw. Fachlichkeit möglich. Eine *Model-Ontologie* dient der Modellierung der zugehörigen Domain-Informationen. Beiden liegt eine *Language-Ontologie* zugrunde, durch welche alle komplexen auf primitive Datentypen und somit letztlich 0 und 1 zur Verarbeitung im Rechner abstrahiert werden.
Die im Folgenden gezeigten Abbildungen sind in UML-Notation zu lesen, wobei die Darstellung von Klassen aus Platzgründen vereinfacht wurde.

System-Ontologie

Natürliches Vorbild eines solchen Denkmodelles ist die oben erwähnte Organisation allen natürlichen Lebens.

Tabelle 5.1: System-Ontologie mit Analogien

Schicht	Klasse	Vergleich
1	Family	Familie, Team
2	System	Mensch, Tier
3	Block	Organ: Gehirn, Muskel, Augen
4	Region	Gehirnregion, Muskelfaser, Netzhaut
5	Component	Neuron, Muskelzelle, Netzhautzelle
6	Part	Zellorganellen (Zellkern, Plasma usw.)
7	Chain	Chromosom, DNA

Basierend auf dieser grundsätzlichen Idee wurden bisher sieben Unterklassen von *Item* als Grundlage einer System-Ontologie festgelegt, welche zusammen mit ihrem "natürlichen" Pendant in Tabelle 5.1 aufgeführt sind. Die oberste Ebene bildet *Family*. Sie ist die Stufe

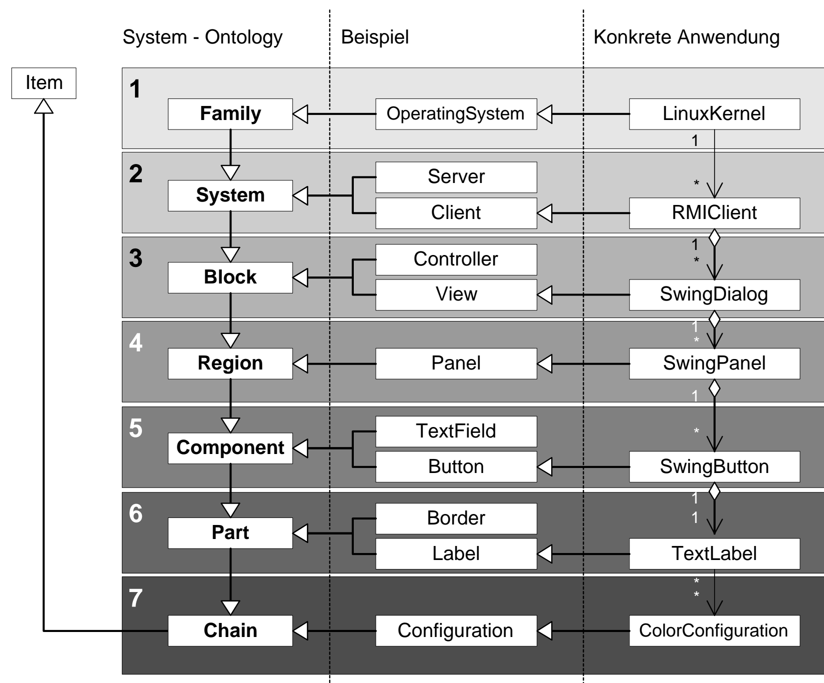

Abbildung 5.3: System-Ontologie mit konkreten Unterklassen und Beispiel

niedrigster Granularität. Darunter folgt *System* als Klasse direkter Unterobjekte. Diese Gliederung nimmt von Stufe zu Stufe einen höheren Detailgrad an, wobei *Chain* die momentan unterste Ebene höchster Granularität darstellt. Eine dieser Ordnung entgegengesetzt ver-

laufende Vererbungshierarchie besitzt *Chain* als einzige direkte Unterklasse von *Item* (vgl. Abb. 5.3). Ein Objekt einer in der Schichtenarchitektur untergeordneten Klasse darf nie ein Objekt der darüberliegenden Abstraktionsstufen referenzieren. Diese Regel entspricht der bereits besprochenen Kreisfreiheit eines Baumes. Die in der Tabelle angeführten Vergleiche sind lediglich ein Mittel, um die Analogien zwischen diesem Framework und der Organisation des Lebens in der Natur aufzuzeigen; sie stellen keine Vorschläge zur Modellierung natürlicher Prozesse dar.
Alle sieben Klassen sind ihrerseits wiederum Oberklassen für eine bestimmte Art von Objekten. Beispielsweise können Client oder Server als Spezialisierung von *System* und grafische Komponenten wie Buttons oder Textfelder als Kindklassen von *Component* abgeleitet werden.

Model-Ontologie

Genau wie die oben beschriebene Organisation von Systemen folgt auch die Model-Ontologie einer hierarchischen Schichten-Struktur. Die oberste Ebene bildet *Record*. In der medizinischen Datenverarbeitung könnte dies ein *Electronic Health Record* (EHR)[6] [ope] sein, im Bereich der Versicherungssoftware eine Versichertenakte. Die weitere Abstufung innerhalb dieser Ontologie vollzieht sich ähnlich der System-Ontologie: Objekte höherer Schichten referenzieren stets nur Objekte der gleichen oder niederer Ebenen. Ein EHR beispielsweise setzt sich neben Stammdaten[7] auch aus Dokumentationen der einzelnen Konsultationen zusammen, die wiederum Befunde (*Objective*) mit Werten für den Blutdruck (*BloodPressure*) etc. beinhalten können.
Die Struktur mit einer sehr einfachen Beispiel-Hierarchie ist Abbildung 5.4 zu entnehmen. Diese Ontologie ist jedoch nicht nur zur Modellierung medizinischer Aspekte geeignet. Sie ist unabhängig von der zugrundeliegenden Domäne und kann für den Entwurf beliebiger Daten-Modelle herangezogen werden.

[6] Als *Electronic Health Record* bezeichnet man die elektronische Patientenakte. Diese beinhaltet die Gesamtheit aller Informationen bezüglich eines Patienten. Andere Bezeichnungen sind *Electronic Medical Record* (EMR) oder *Electronic Patient Record* (EPR).
[7] Stammdaten sind allgemeine Patientendaten wie Name, Adresse usw.

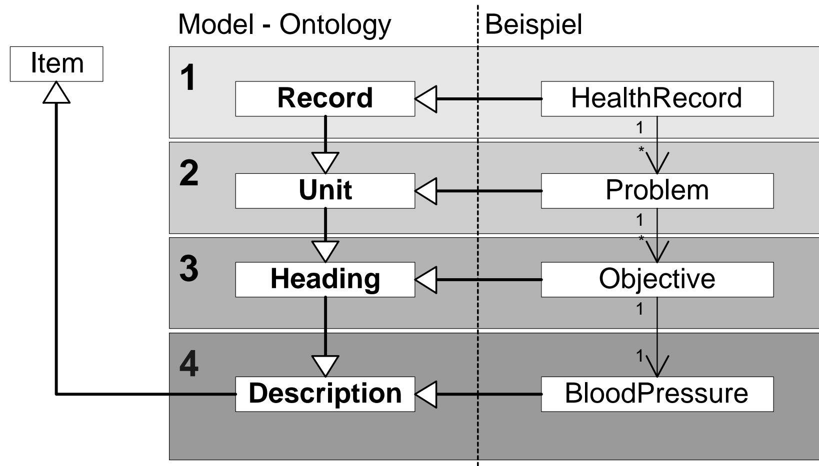

Abbildung 5.4: Model-Ontologie und konkrete Unterklassen

Language-Ontologie

Jeder Softwareentwickler, der mit einer Programmiersprache arbeitet, verwendet (häufig unbewusst) bereits eine Ontologie, nämlich die von der Sprache zur Verfügung gestellten Typen. Diese unterliegen natürlich ebenfalls einer Hierarchie. So besteht z.B. eine Zeichenkette (*String*) aus einzelnen Zeichen (*Character*).
Eine konkrete Modellierung von Klassen dieser Ontologie ist Teil zukünftiger Arbeiten.

Zusammenfassung Der grosse Vorteil dieses Frameworks liegt in seiner sehr einfachen und intuitiven Struktur. Objektrelationen werden dadurch transparent und leicht nachvollziehbar. Das liegt vor allem daran, dass man den gesamten Aufbau des Frameworks von *Res Medicinae* als Konglomerat verschiedenster Architektur- und Entwurfsmuster betrachten kann. Die aus Abbildung 5.2 nachvollziehbare *Chain of responsibility* mit der typischen Weiterleitung von Zuständigkeiten und die ebenfalls ersichtliche *Layer*-Struktur sowie die Realisierung eines *Composite* durch die Klasse *Item* sind dabei nur die drei wichtigsten verwendeten Muster.
Zum anderen besitzt das Framework keinen dogmatischen Charakter und lässt genug Freiheiten für die Implementation zusätzlicher Ideen entsprechend der gewünschten Anwendung.

Kapitel 6

Record - Modul zur medizinischen Dokumentation

Nachdem in den vorangegangenen Kapiteln die theoretischen Konzepte dieser Diplomarbeit beschrieben wurden, soll nun auf die praktische Umsetzung dieser Techniken eingegangen werden.
Der im Rahmen dieser Arbeit implementierte Prototyp *Record* zur medizinischen Befundung ist Teil eines größeren Projektes namens *Res Medicinae*, welches sich in mehrere voneinander unabhängige Softwaremodule gliedert. Diese Module decken eigenständige Aufgabenbereiche der zukünftigen Anwendung ab. So existiert innerhalb von *Res Medicinae* beispielsweise ein weiteres Modul *ReForm* für das Ausdrucken von Formularen [Kun03]. Weiterhin geplant sind Module für die Administration von Labordaten und die Archivierung und Bearbeitung von medizinischem Bildmaterial.

6.1 Ziele

Elektronische Datenverarbeitung ist im medizinischen Bereich nicht mehr wegzudenken. Neben Informationsgewinnung und -verarbeitung tut sich die Präsentation von Daten als ein Hauptaufgabengebiet der Informatik hervor. Krankenhausinformationssysteme als monolithische Anwendungen oder über Rechenzentren koordiniert, decken diese Aufgaben im klinischen Umfeld (mehr oder weniger zufriedenstellend) weitestgehend ab.

Ziel des *Record*-Moduls ist es, den Arzt bei der Erstellung von patientenbezogenen Diagnosen, d.h. bei der **medizinischen Dokumentation**, zu unterstützen. Mit Hilfe des folgenden Szenarios soll erläutert werden, was hierunter zu verstehen ist:

Ein Patient konsultiert seinen Hausarzt bezüglich eines bestimmten Problems. Er schildert ihm dieses, worauf hin der Mediziner die Anamnese[1], also eine Dokumentation der Aussagen eines Patienten über den Verlauf einer Krankheit, anfertigt.

Danach wird der Patient in den meisten Fällen problembezogen untersucht werden, womit die eigentliche Befundung beginnt. Eine solche Untersuchung kann von einem schlichten Messen des Blutdruckes über ein Abhören der Herzaktivität bis hin zur Erstellung eines Elektro-Kardiogrammes (EKG) reichen. In anderen Fällen wird dieser Teil der Konsultation auch auf weiterführende Diagnosemethoden ausgeweitet. So kann es der Arzt für notwendig erachten, eine radiologische Untersuchung (z. B. Röntgen) oder eine Gastroskopie[2] anzuordnen. All diese Untersuchungsmethoden haben ein Ergebnis: den Befund. Dieser wiederum kann die unterschiedlichsten Formen annehmen: Bei der Untersuchung des Blutdruckes werden die numerischen Werte für Systole[3] sowie Diastole[4] gemessen und notiert, bei einer Gastroskopie die Beschaffenheit der Magenschleimhaut bzw. ob an ihr krankhafte Veränderungen sichtbar sind. Sind keine pathologischen[5] Erscheinungen festzustellen, spricht man auch von *keinem Befund* (*ohne Befund* bzw. *o. B.*). Ausgehend von Anamnese und Befund kann die Diagnose erstellt werden. Inhalt dieser ist die Krankheitserkennung. Darauf aufbauend kann die Therapie eingeleitet werden. Therapeutisches Handeln umfasst Medikamentenverschreibungen, Überweisungen zu anderen Ärzten, Verordnung von Kuren, Massagen oder anderen physiotherapeutischen Maßnahmen, sowie Hinweise an den Patienten, wie seine Genesung durch ihn selbst vorangebracht werden kann.

Der dargestellte Vorgang stellt natürlich einen Idealfall der Diagnose und Therapie dar. In vielen Fällen kann nicht davon ausgegangen werden, bei der ersten Konsultation des Patienten sofort die optimalen therapeutischen Maßnahmen einleiten zu können.

Aufgabe des *Record*-Modules ist die bestmögliche Unterstützung dieser dargestellten Abläufe

[1] Anamnese *(lat.)*: Wiedererinnern, Krankheitsvorgeschichte
[2] Eine Gastroskopie (Magenspiegelung) ist eine endoskopische Untersuchung des Magens.
[3] Systole (griech.): Kontraktion des Herzmuskels, Druck des ausströmenden Blutes.
[4] Diastole (griech.): Erschlaffung des Herzmuskels, Druck des einströmenden Blutes.
[5] pathologisch (lat.): krankhaft

auf Dokumentationsbasis. In enger Zusammenarbeit mit den Ärzten der *Res Medicinae*-Analyse-Mailingliste wurde versucht, das Modul als effektives Werkzeug der medizinischen Dokumentation zu konzipieren.
Ein solches Vorgehen beinhaltet zunächst einmal den Entwurf einer grafischen Oberfläche nach ergonomischen Gesichtspunkten, wie sie im ersten Kapitel beschrieben wurden.
Die in den Kapiteln drei bis fünf enthaltenen theoretischen Konzepte zum erweiterten Lebenszyklus von Softwarekomponenten sowie das ontologische Framework finden hier natürlich ebenfalls ihre praxisnahe Umsetzung.
Ein weiterer Punkt ist die softwaretechnische Umsetzung neuester Modelle auf dem Gebiet der medizinischen Befundung. Diese Modelle zeigen neue Sichtweisen auf und stellen alternative Strukturen für medizinische Daten vor. Die einfache und sehr unübersichtliche Präsentation der gesamtem elektronischen Patientenakte in einer schlichten Tabelle oder Liste hat sich in den letzten Jahren als unvorteilhaft und zu wenig intuitiv herausgestellt. Vielmehr ist man an einer selektiven Darstellung von Patienteninformationen interessiert, die statische wie auch dynamische Gesichtspunkte einer Befundung repräsentiert.

6.2 Modelle der Befundung

Dieser Abschnitt soll die im Modul *Record* umgesetzten neuen Modelle der Befundung kurz vorstellen.

6.2.1 Episodenbasierte Befundung

Episodenbasierte Befundung stellt einen alternativen Weg dar, Konsultationen eines Arztes durch einen Patienten zeitlich sowie problembezogen zu dokumentieren. An dieser Stelle sind die Begriffe *Teilkontakt* und *Problem* zu erläutern. Ein Teilkontakt bezeichnet das Aufsuchen des Arztes durch einen Patienten aufgrund eines vorhandenen gesundheitlichen Problems. Klagt der Patient während einer Konsultation über mehrere Probleme, so existieren dementsprechend auch mehrere Teilkontakte.

Als Problem bezeichnet man in diesem Zusammenhang einen pathologischen Zustand, also eine Beeinträchtigung der Gesundheit über eine bestimmte Zeitspanne.
Hierauf aufbauend, definiert sich die *Episode* als eine zeitliche Aneinanderreihung von Teilkontakten bezogen auf das zugehörige Problem. Eine Episode kann aus einem oder mehreren solcher Kontakte bestehen (vgl. Abb. 6.1).

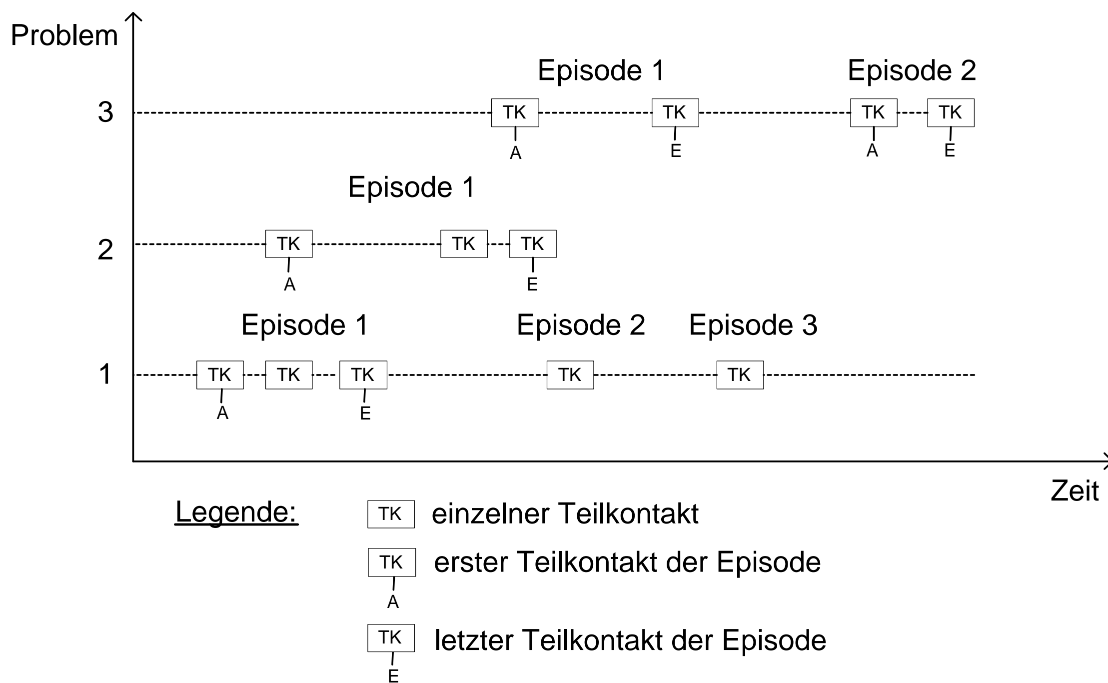

Abbildung 6.1: Probleme und Episoden

Der erste Teilkontakt markiert den Anfang, der letzte das Ende einer Episode.
In der obigen Abbildung existieren für das erste Problem insgesamt drei Episoden, wobei die erste aus drei Teilkontakten besteht und die letzten beiden jeweils nur aus einem. Nimmt man beispielsweise an, bei dem Problem handelt es sich um eine chronisch auftretende Speiseröhrenentzündung, so hätte der Patient den Arzt diesbezüglich fünf mal in der betrachteten Zeitspanne (z. B. 2 Jahre) konsultiert. Der Abschluss der einzelnen Episoden könnte mit einem Abklingen bzw. einer Beseitigung der Beschwerden in Verbindung gebracht werden. Durch Medikamentengabe wäre also eine Heilung bis zum dritten Teilkontakt eingetreten und die Episode beendet. Ein erneutes Auftreten der Symptome zu einem späteren Zeitpunkt würde die Episoden zwei und drei notwendig machen, welche aber aufgrund einer schnellen Abheilung aus lediglich einem Teilkontakt bestehen.

6.2.2 SOAP-Modell

Um von Anfang an Missverständnissen aus dem Weg zu gehen: Der Begriff *SOAP* bezeichnet in diesem Kontext **nicht** das gleichnamige XML-basierte Kommunikationsprotokoll, sondern steht für ein medizinisches Dokumentationsmodell.
Ein Teilkontakt (siehe vorheriger Abschnitt) gliedert sich gemäß des SOAP-Modells in die vier Abschnitte **S***ubjective* (Anamnese), **O***bjective* (Befund), **A***ssessment* (Bemerkung bzw. Einschätzung) und **P***lan* (Vorgehen). Im weiteren Verlauf dieses Dokumentes wird zwar noch vom SOAP-Modell gesprochen, jedoch werden die entsprechenden deutschen Begriffe für Subjective, Objective, Assessment und Plan verwendet. Tabelle 6.1 soll einen möglichen Teilkontakt darstellen, dessen Einteilung nach dem SOAP-Model erfolgt. Dabei werden die medizinischen Begriffe Anamnese, Befund, Bemerkung und Vorgehen durch Beispiele zusätzlich erläutert.

Tabelle 6.1: SOAP-Model mit Beispielen

Abk.	Begriff	Erläuterung	Beispiel
S	Anamnese	Schilderung des gesundheitlichen Problems durch den Patienten	Patient klagt über starke Schmerzen im Oberbauch, Völlegefühl, Sodbrennen, Durchfall seit 3 Tagen
O	Befund	Objektiver Eindruck des Mediziners, Feststellung auffälliger Merkmale	Harte Bauchdecke, Zungenbelag, Blutdruck: 150/110, Puls: 90
A	Bemerkung, Einschätzung	Allgemeine Angaben: Körperlicher Allgemeinzustand, soziale nicht-medizinische Einflußfaktoren, Sonstiges	Übergewicht, Stress im Beruf, Choleriker, Scheidung
P	Vorgehen	Therapeutische Maßnahmen, Medikamentenverordnung	Gastroduodenoskopie (Magen-Darm-Spiegelung), Omeprazol (Magen-Darm-Therapeutikum) 300 mg

6.2.3 Topologische Befundung

In vielen Fällen der Befundung von pathologischen Erscheinungen ist neben einer präzisen Darstellung des eigentlichen Sachverhaltes auch eine Lokalisation der krankhaften Veränderung notwendig. Der Mediziner kennt hierbei für jedes noch so kleine Körperteil und seine unterschiedlichen Ansichten eine zumeist lateinische Bezeichnung[6] und kann diese textuell notieren und zur Dokumentation anwenden.
Eine alternative Möglichkeit bietet die sogenannte *Topologische Befundung*. Mit ihr ist es

[6] *Ventral* und *dorsal* bezeichnen beispielsweise Ansichten eines Körperteils von der Vorder- und Rückseite, *lateral* die Seitenansicht.

möglich, den betreffenden Abschnitt des Körpers anhand eines grafischen Modells zu selektieren und an den gewünschten Stellen Befunde zu dokumentieren. Gleichzeitig soll durch das Setzen von Markierungen die exakte Positionierung von Befunden möglich sein.

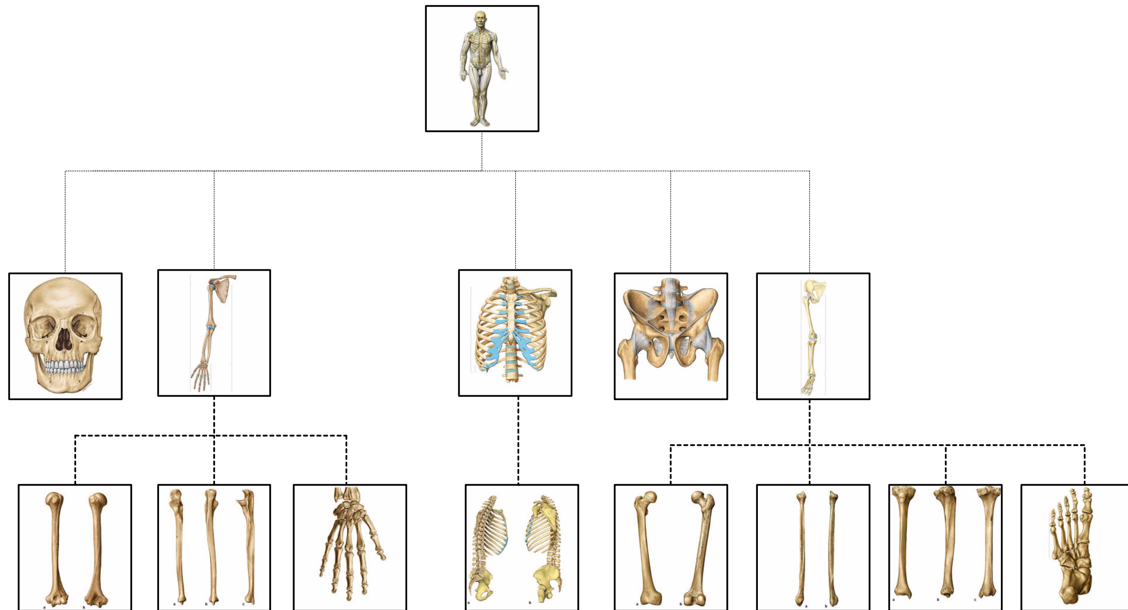

Abbildung 6.2: Hierarchie von Darstellungen des menschlichen Skelettsystems

Es existieren dabei anatomische Abbildungen des Menschen, die einen ausreichenden Detaillierungsgrad aufweisen. Ausgehend von einer Gesamtdarstellung kann über Maus und Tastatur in alle Regionen des menschlichen Organismus verzweigt werden.

Eine Hierarchie von Bildern des Skelettsystems (vgl. Abb. 6.2) macht dies deutlich: Initiale Darstellung ist die ganze Ansicht des menschlichen Skeletts. Hier sind Schädel, Brustkorb, Becken und die Extremitäten selektierbar, die dann auf der zweiten Ebene zu sehen sind. Von hier aus kann wiederum noch tiefer in die Skelett-Anatomie vorgedrungen werden. Abbildung 6.2 zeigt jedoch nicht die alternativen Ansichten jeder Region, denn zu fast allen Bildern existieren neben ventralen und dorsalen, auch laterale Abbildungen.

Die Idee der Umsetzung eines Prototypen zur Topologischen Befundung kam während der Formulierung dieser Diplomarbeit, auch angeregt durch das Analysedokument von *Res Medicinae* [CH02]. Wie sich kurz darauf herausstellte, ist diese Form der Befundungsunterstützung kein wirklich neues Konzept, jedoch hat es im Rahmen von klinischen Informationssystemen noch kaum Anwendung gefunden. Grund hierfür ist einmal mehr der zeitliche

Aspekt einer solchen Methode. Vielen Ärzten ist die umständliche Bedienung einer Anwendung zum Erstellen einer Befundung zu wider. Die meisten beschränken sich auf die Kurzform von Diagnosen in ICD[7] -Form. Diese Methode ist in ihrer Ausführung zeitlich sicher nicht zu unterbieten.

Dennoch stellt die Topologische Befundung kein redundantes Werkzeug dar. Sie sollte diagnoseunterstützend gerade bei der Lokalisation und Markierung räumlich abgegrenzter, pathologischer Befunde eingesetzt werden. Dies könnten beispielsweise Frakturen, Entzündungen, Verbrennungen, Unfallverletzungen oder Tumore sein. Neben dieser notizblockähnlichen Gedächtnisstütze für den behandelnden Mediziner sind diese Abbildungen auch zur Erläuterung medizinischer Sachverhalte während der Sprechstunde geeignet.

6.3 Realisierung

In den letzten Abschnitten wurden die für das Verständnis notwendigen medizinischen Dokumentationsmodelle beschrieben. Diese implizieren eine Klassenstruktur, die das Episoden- und das SOAP-Modell in sich vereint und dabei den Regeln des ontologischen Framework von *Res Medicinae* folgt.

Zunächst wird das entworfene Domain-Modell dargelegt, danach auf die Treetable-Komponente von *Record* eingegangen und schließlich die Realisierung der topologischen Befundung innerhalb des Prototypen vorgestellt.

6.3.1 Domain-Modell

Abbildung 6.3 zeigt dieses Domain-Modell. Die linke Seite entspricht der bereits vorgestellten Model-Ontologie (vgl. Abb. 5.4). In der obersten Ebene ist der *HealthRecord* zu finden, der die gesamte elektronische Patientenakte verkörpert. Problem, Episode und Teilkontakt folgen in dieser Reihenfolge.

[7] Die ICD (*International Classification of Diseases*) ist eine international standardisierte Klassifikation von Krankheiten und Todesursachen.

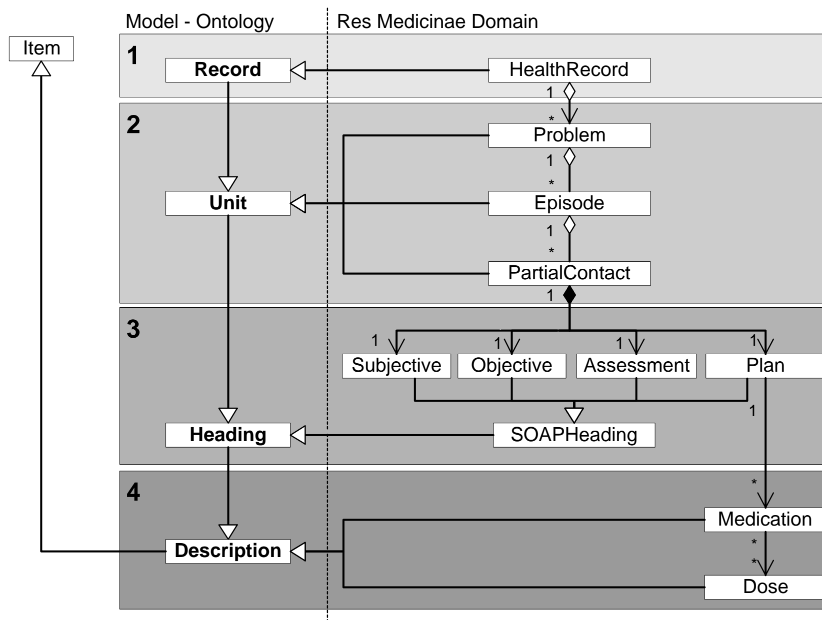

Abbildung 6.3: Domain-Modell unter Einordung in Model-Ontologie

Ein Problem (Klasse *Problem*) kann beliebig viele Episoden (Klasse *Episode*) und diese wiederum eine unbegrenzte Anzahl an Teilkontakten (Klasse *PartialContact*) beinhalten. Diese drei Klassen befinden sich in der ontologischen Ebene *Unit*.
Jeder Teilkontakt besteht aus je einem Objekt der Klassen *Subjective*, *Objective*, *Assessment* und *Plan*, welche sich in der Ebene *Heading* befinden und sich über eine Klasse namens *SOAPHeading* in die Klassenhierarchie einfügen. *SOAPHeading* bündelt als Hilfsklasse gemeinsame Eigenschaften.
Auf der untersten Ebene sind lediglich zwei Klassen angedeutet, die in dem Domain-Modell eine Rolle spielen: Das Vorgehen kann als therapeutische Maßnahmen Medikamentenverschreibungen (Klasse *Medication*) mit den entsprechenden Dosen (Klasse *Dose*) beinhalten.

6.3.2 Treetable-Komponente

Die beschriebenen medizinischen Modelle setzen eine optimale Präsentation der durch sie in Struktur gebrachten Informationen vorraus. Patientenbezogene Daten, inhaltlich gegliedert nach dem SOAP-Modell und durch die Zuordnung zu einer Episode in die korrekte zeitliche Reihenfolge gebracht, verlangen nach einer ebenso strukturierten Darstellung. Diese wird

durch eine baumartige Anzeige in einem *Treetable* gewährleistet.

Eine solche Komponente vereint die Vorteile von Bäumen mit denen von Tabellen. Prinzipiell sind die Informationen noch in Zeilen und Spalten organisiert, können aber zum Einen hierarchisch und zum Anderen platzoptimiert untergebracht werden. Platzoptimiert deshalb, weil es durch die Bedienung eines solchen Baumes möglich ist, bestimmte Teile der Tabelle auszublenden oder sichtbar zu machen, je nach dem, welcher Detail-Level gewünscht wird.

Eine solche Form der Präsentation von Daten ist nichts grundlegend Neues. Zur Verwaltung von großen Dateisystemen mit Ordnern und vielen verschiedenen Dateiformaten wird sie schon seit dem Aufkommen grafischer Benutzeroberflächen verwendet.

Weiterhin kann die Anzeige durch Filtereinstellungen zusätzlich angepasst werden. Möchte man beispielsweise nur die Befunde innerhalb einer bestimmten Episode bewerten, kann man diese mit Hilfe des Filters separiert von den anderen Informationen darstellen. Dasselbe gilt für alle anderen Elemente des SOAP-Modells sowie die vorhandenen Teilkontakte, Episoden und Probleme.

6.3.3 Prototyp zur Topologischen Befundung

Weiterer Bestandteil des *Record*-Moduls ist ein Prototyp zu der bereits besprochenen *Topologischen Befundung*. In ihm enthalten ist ein *Homunculus*[8], welcher das menschliche Skelett fast vollständig repräsentiert und in dem sowohl mit Hilfe der Maus, als auch über die Tastatur navigiert werden kann.

Ausgehend von der Komplettdarstellung des menschlichen Skelettsystems kann der Nutzer in die unterschiedlichen Bereiche des Knochengerüstes vordringen und an gegebener Stelle eine Befundung eingeben. Bei dieser Form der Darstellung können Frakturen oder andere pathologische Veränderungen der Knochen dokumentiert werden. Zusätzlich zur Erfassung von textuellen Notizen ist es auch möglich, Regionen im Bild zu markieren. Dies kann besonders bei der Lokalisation von Bruchstellen hilfreich sein und soll die Dokumentation von Befunden visuell unterstützen.

[8] Homunculus (*lat.*): künstlich erzeugter Mensch

Anatomisches Bildmaterial Einen erheblichen Zeitaufwand verursachte die Beschaffung von geeignetem und verwendbarem Bildmaterial. Dabei mussten die einzelnen Abbildungen von Teilen des menschlichen Körpers bestimmten Kriterien genügen. Zum Einen war es notwendig, dass diese Bilder zwar einen gewissen Detailgrad aufwiesen, jedoch in der absoluten Genauigkeit der Darstellung nicht jede Einzelheit hervorhebten. Der *Homunculus* sollte die Befundung unterstützen und nicht als detailgetreues, fotografisches Anschauungsmaterial für Medizinstudenten dienen. Auf der anderen Seite mussten Abbildungen gefunden werden, die in der Form der grafischen Umsetzung untereinander stimmig und konsistent waren. Das bedeutet, dass die Art der Zeichnung der Knochen zu der Darstellung von Weichteilen optisch passen musste.
In einem kommerziellen Atlas der Anatomie wurden genau solche Abbildungen gefunden. Auf Anfrage an den herausgebenden Verlag durften diese auch unter Angabe des Produkt- und Verlagnamens unentgeltlich verwendet werden.

Overlays Alle ein solches anatomisches Bild betreffenden Informationen, wurden in einem dafür vorgesehenen XML-Format gespeichert. XML stellte zum damaligen Zeitpunkt das einzige, im Rahmen von *Res Medicinae* existierende, Persistenzmedium dar. Die in einer solchen Datei enthaltenen Informationen umfassen neben trivialen Angaben wie dem Namen der Abbildung oder der zugehörigen medizinischen Beschreibung auch Daten über die sensiblen Regionen des Bildes und eine Referenz auf das zugrundeliegende Bild aus dem Atlas der Anatomie. Da dieses XML-Format einen fiktiven bzw. gedachten "Überzug" über ein real existierendes digitales JPEG-Bild darstellt, wurde ihm der Name *Overlay* gegeben. Abbildung 6.4 zeigt beispielhaft ein solches XML-Format für ein Overlay des menschlichen Beines in der Skelettdarstellung und Vorderansicht.

```xml
<?xml version="1.0" ?>
<Overlay>
	<Name>leg_front</Name>
	<Image>skeleton/extremities/leg/leg_front.jpg</Image>
	<Description>leg, front view</Description>
	<OverlayRegion>
		<OverlayRegionName>pelvis front</OverlayRegionName>
		<TargetOverlay>male_pelvis_front</TargetOverlay>
		<Number>1</Number>
		<Neighbours>
			<Top/>
			<Bottom>hip_joint_front</Bottom>
			<Left/>
			<Right/>
		</Neighbours>
		<Point>
			<X>171</X>
			<Y>21</Y>
		</Point>
		<Point>
			<X>166</X>
			<Y>26</Y>
		</Point>
		...
	</OverlayRegion>
</Overlay>
```

Abbildung 6.4: Beispiel der XML-Darstellung eines Overlays

Im Folgenden sind die dazu verwendeten XML-Elemente aufgeführt.

- <`Name`> Name des Overlays.
- <`Image`> Name des zugehörigen JPEG-Bildes mit Pfadangabe.
- <`Description`> Genauere medizinische Beschreibung des Overlays.
- <`OverlayRegion`> Sensibler Bereich des Overlays. Genauere Erläuterung im nächsten Abschnitt.
- <`OverlayRegionName`> Name des sensiblen Bereiches.
- <`TargetOverlay`> Name des Overlays auf das mit Hilfe der OverlayRegion verzweigt wird.
- <`Number`> Nummer der Overlay-Region innerhalb des Overlays.
- <`Neighbours`> Nachbarregionen dieser Overlay-Region. Werden zur Navigation auf dem Overlay mit Hilfe der Pfeiltasten benötigt.

- <Top> Oberer Nachbar.
- <Bottom> Unterer Nachbar.
- <Left> Linker Nachbar.
- <Right> Rechter Nachbar.
- <Point> X- und Y-Koordinaten der Overlay-Regionen.
- <X> X-Koordinate.
- <Y> Y-Koordinate.

Beim Start des *Record*-Moduls werden alle zur Verfügung stehenden Overlays mit Hilfe des XML-Parsers *Xerces* geparst und eingelesen. Dieser wandelt das Dateiformat XML in ein *Document Object Model* (DOM), das als charakteristische Objekt-Struktur allen Auszeichnungssprachen[9] zugrunde liegt. Das DOM stellt über eine Programmierschnittstelle (API) Operationen zur Bearbeitung der in ihm enthaltenen Objekte bereit und ermöglicht so eine Manipulation der Elemente. Bei der Transformation eines XML-Formates in ein Document Object Model bleibt der verschachtelte Charakter dieser Struktur erhalten. Die in dem entstandenen DOM vorhandenen Informationen des Overlays können jetzt über die Funktionen der API gelesen sowie dem Modell neue Elemente hinzugefügt werden.

Overlay-Regionen Overlay-Regionen bezeichnen die "sensiblen" Areale eines Overlays. Prinzipiell sind diese nichts anderes als beliebige Polygone, die in Form von Flächenabschnitten bestimmte Bereiche von Overlays abgrenzen. Punkte dieser Polygone werden durch die Tags namens <Point> beschrieben (vgl. Abb. 6.4).
Geht man zum Beispiel von der bereits erwähnten Darstellung des menschlichen Beines aus, existieren Overlay-Regionen für den Oberschenkel, das Knie, Teile des Beckens, das Schienbein und den Fuß (vgl. Abb. 6.5). Diese Regionen verfärben sich beim Überfahren mit dem

[9] In Auszeichnungssprachen sind verschiedene Befehle definiert, die der optischen und inhaltlichen Strukturierung von Informationen dienen. Zu diesen Sprachen gehören neben XML z. B. auch die Standard Generalized Markup Language (SGML) und die Hyper Text Markup Language (HTML).

Mauszeiger bzw. der Anwahl über die Pfeiltasten und signalisieren so den Status der Selektion. Durch Anklicken mit der Maus oder Drücken von *Enter* erscheint dann das mit dieser Region assoziierte Overlay. Bei einer Selektion des Schienbeines würde dieses also als nächstes Overlay geladen werden. *Escape* bewirkt den Rücksprung auf das vorherige Overlay. Auf diese Art und Weise kann sich der Nutzer ausgehend von der Gesamtdarstellung des menschlichen Skeletts in jeden möglichen Abschnitt bewegen.

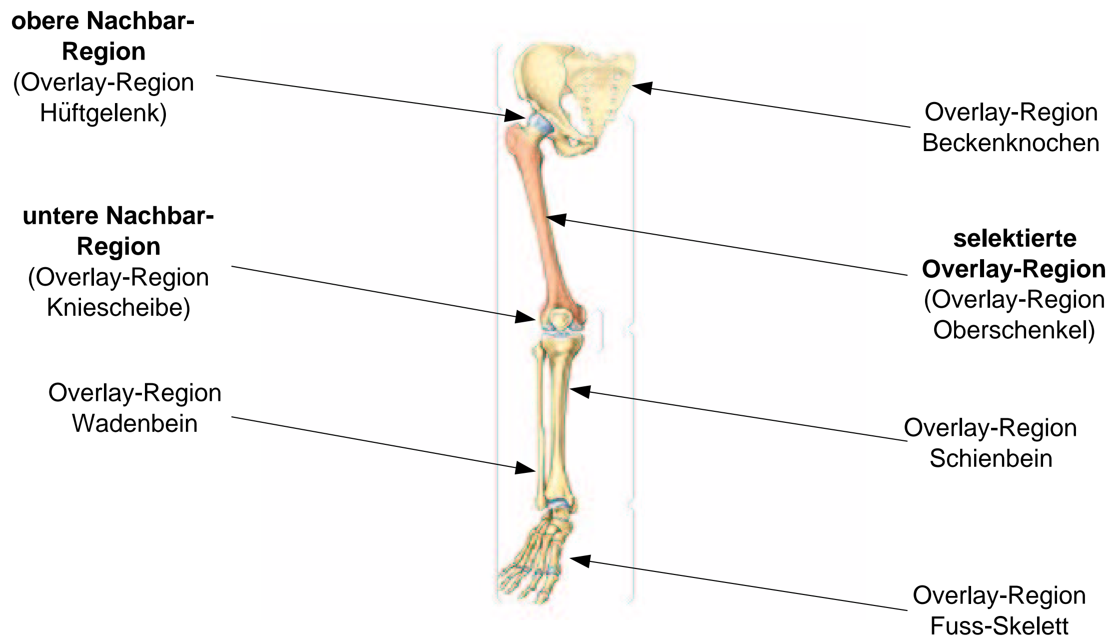

Abbildung 6.5: Grafische Darstellung eines Overlays des menschlichen Beines

Overlay-Editor Um dem System neue Overlays hinzufügen und alte anpassen zu können, existiert ein Editor als integrativer Bestandteil des *Record*-Moduls. Dieser ermöglicht es, Bearbeitungsoperationen auf den Overlays ausführen zu können. So können damit z. B. alte Overlays gelöscht und neue hinzugefügt werden. Es ist durch ihn möglich, Overlay-Regionen zu erstellen und zu entfernen sowie die Reihenfolge zur Navigation zwischen den einzelnen Overlays festzulegen.

Dieses zusätzliche Werkzeug soll auch nach Beendigung der Arbeit an diesem Prototypen eine zukünftige Anpassung des *Record*-Modules an neue anatomische Bilder und die damit verbundenen Overlays ermöglichen.

Kapitel 7

Zusammenfassung und Ausblick

Unter steter Bezugnahme auf die zugrundeliegende praktische Umsetzung im Projekt *Res Medicinae*, sind in dieser Arbeit seit Jahren existierende Konzepte der Softwaretechnik miteinander kombiniert und um eigene Ideen erweitert worden.

Mit den ersten beiden Kapiteln wurde versucht, dem Leser die Motivation dieser Diplomarbeit darzulegen und ihm einen Zugang zu diesem Thema zu ermöglichen. Angefangen bei sehr allgemeinen Betrachtungen zu Präsentationsschichten, konkretisierten sich die Gedanken bei der Behandlung von grafischen Oberflächen. In diesem Zusammenhang wurde es als notwendig erachtet, neben den technischen Gesichtspunkten des Entwurfes dieser Nutzerschnittstellen auch die ergonomischen zu berücksichtigen. Die Präsentationsschicht bzw. grafische Oberfläche des Softwaresystems *Res Medicinae* stellte den praktischen Bezug dazu her. Das zweite Kapitel fällt damit inhaltlich etwas aus der Reihe, wurde aber als Einstieg gewählt, um dem Leser den Ort der Realisierung der in den darauffolgenden Kapiteln beschriebenen Konzepte vorzustellen: die Präsentations- bzw. Darstellungsschicht.

Muster, ihre Kategorisierung und Anwendung bildeten den Inhalt des dritten Kapitels. Auch hier richtete sich der Fokus auf die Umsetzung von Entwurfs- und Architekturmustern, speziell in Präsentationsschichten. Letztere Musterart fand in der Vorstellung des *Model-View-Controller*-Paradigmas sowie der hierfür existierenden Erweiterung in Form des HMVC besondere Aufmerksamkeit.

Ideen der komponentenbasierten Softwareentwicklung sind – wie Muster – nicht nur für die Konzeption und Umsetzung grafischer Oberflächen geeignet. Das vierte Kapitel behandelte

daher den Begriff der *Softwarekomponente* und den damit verbundenen Lebenszyklus als eine von der grafischen Oberfläche unabhängig entwickelte Idee. Natürliche Lebensvorgänge und Organisationsstrukturen dienten den Komponenten ebenso wie dem *Res Medicinae*-Framework *ResMedLib* als Vorbild. Dieses wurde in Kapitel fünf vorgestellt und auf Besonderheiten und Vorteile einer solchen ontologischen Hierarchie eingegangen.
Den Abschluss der Diplomarbeit bildete mit Kapitel sechs die Präsentation des Prototypen, in dem alle in den vorhergehenden Kapiteln beschriebenen Denk-Modelle eine praktische Realisierung fanden.

Der erweiterte Komponentenlebenszyklus und die Ontologien erfuhren mit der Entwicklung dieses Prototypen eine der ersten praktischen Anwendungen. Dementsprechend ist die entstandene Software also auch als Mittel zur Evaluation dieser theoretischen Ideen anzusehen. Aber der entwickelte Prototyp bedarf einer stetigen Verbesserung und Weiterentwicklung – hierfür einige abschließende Anregungen:

Fehlerbehandlung Die Einhaltung der Reihenfolge des Aufrufs der einzelnen Operationen des Komponentenlebenszyklus' (`globalize`, `initialize`, `link` usw.) ist für die Vermeidung von Laufzeitfehlern von besonderer Wichtigkeit. Ein Objekt darf erst benutzt werden, wenn es ordnungsgemäß initialisiert und konfiguriert worden ist. Deshalb wäre die Entwicklung geeigneter Fehlerklassen hilfreich, die sich entweder in die System-Ontologie einfügen, oder ihrerseits eine eigene Ontologie bilden (*Exception*[1]*-Ontologie*).
Das Gleiche gilt für die Verletzung der angesprochenen Kreisfreiheit der Objektbeziehungen innerhalb einer Ontologie. Dieser Verstoß gegen die Regeln des Frameworks sollte ebenfalls durch eine geeignete Fehlerbehandlung – nach Möglichkeit schon zu Compile-Zeit – angezeigt werden.

Erweiterung des Komponentenlebenszyklus Inwieweit die bisher gefundenen Methoden zur Verwirklichung des Lebenszyklus von Softwarekomponenten praktikabel sind, ist nur durch praktische Anwendung zu ermitteln. Daher kann eine Erweiterung des Zyklus notwen-

[1] Exception (*engl.*): Ausnahme (im Sinne eines aufgetretenen Fehlers)

dig werden, die jetzt noch nicht absehbar ist. Ob die Reihenfolge der Aufrufe `globalize`, `initialize`, `link` usw. genügend Granularität aufweist und auch immer auf alle Komponenten gleich angewandt werden muss, wird also ebenfalls Inhalt zukünftiger Arbeiten sein.

Erhöhung der Granularität von ontologischen Hierarchien Dasselbe gilt auch für das ontologische Framework. Sollte sich beispielsweise die Trennung der Ontologie für Systeme (*Family*, *Block*, *Region* usw.) als zu abstrakt erweisen, kann man durch die Einführung weiterer Schichten die Granularität der Hierarchie an die Erfordernisse der konkret zu implementierenden Anwendung anpassen.

Kopplung ontologischen Hierarchien Das Ziel weiterer Untersuchen könnte darin bestehen, die drei Ontologien (*System*, *Model* und *Language*) in geeigneter Weise zu kombinieren und Beziehungen zwischen diesen aufzubauen.

Darstellung der Episoden auf Zeitachse Um die Episoden, ähnlich wie in Abbildung 6.1, auf einer Zeitachse darzustellen, wäre die Entwicklung einer weiteren Grafik in Form eines Balkendiagrammes denkbar. Dieses würde die zeitlichen Zusammenhänge zwischen Problemen bzw. Episoden verdeutlichen.

Web-basierte Steuerung Das XML-Datenformat der Overlays eignet sich nicht nur für einen Einsatz unter Java Swing. Die darin vorhandenen Informationen können durch XSL/XSLT in alternative Formate wie HTML umgewandelt werden, wodurch eine Publikation über das Internet möglich wird (siehe Abschnitt 2.4.2: XML/XSL). Durch die Wahl von XML als Datenformat für die Repräsentation des *Homunculus* ist also implizit der Grundstein für eine web-basierte Steuerung gelegt worden.

Versendung von Overlays per Email Auch ist eine Versendung solcher Overlays via Email und anschließende Weiterverarbeitung und -verwendung des XML-Formats in anderen Programmen denkbar.

Erweiterung des Homunculus Im Rahmen dieser Diplomarbeit wurde lediglich ein Skelett-*Homunculus* umgesetzt. Aufgabe zukünftiger Arbeiten könnte es also sein, weitere Ansichten des Körpers wie Weichteile, Blutgefäß- oder Nervensystem einfließen zu lassen. Der Overlay-Editor bietet dabei die Möglichkeit einer einfachen Erweiterung des Bildbestandes und die Pflege der Overlays. Dadurch ist die Verwaltung einer großen Menge anatomischer Bilder möglich und kann auch bei Bedarf von den Medizinern selbst vorgenommen werden.

Konfiguration der Steuerung Neben einer Steuerung des Prototypen per Maus ist es ebenfalls möglich, alle Funktionen über die Tastatur abzurufen. Jedoch ist es noch nicht vorgesehen, den Nutzer selbst über die Wahl der Tastenbelegung entscheiden zu lassen. Auch dieser Punkt wäre als verbesserungswürdig zu betrachten.

Literaturverzeichnis

[asp02] *AspectJ.* http://aspectj.org, 2002.

[ava] *Apache Jakarta Avalon Framework.* http://jakarta.apache.org/avalon/index.html.

[CA77] Christopher Alexander, Sara Ishikawa, Muray Silverstein et al.: *A Pattern Language.* New York: Oxford University Press, 1977.

[CH02] Christian Heller, Karsten Hilbert, Roland Colberg et al.: *Analysedokument zur Erstellung eines Informationssystems für den Einsatz in der Medizin.* 2002.

[Cop92] Coplien, J. O.: *Advanced C++ –Programming Styles and Idioms.* Bonn: Addison-Wesley, 1992.

[EG96] Erich Gamma, Richard Helm, Ralph Johnson et al.: *Entwurfsmuster– Elemente wiederverwendbarer objektorientierter Software.* Bonn: Addison-Wesley, 1996.

[FB98] Frank Buschmann, Regine Meunier, Hans Rohnert et al.: *Patternorientierte Softwarearchitektur.* Bonn: Addison-Wesley, 1998.

[Fix01] Fix, Bernd: *Aspekt-orientierte Programmierung – Eine Einführung.* http://www.brainon.ch/area51/visions/AspectJ.pdf, 2001.

[Hel97] Heller, Christian: *Entwicklung eines Praktikumsversuches zu Neuronalen Netzen auf der Basis von Borland Delphi.* Diplomarbeit, Technische Universität Il-

menau, 1997.

[Hel02] HELLER, CHRISTIAN: *Cybernetics Oriented Programming.* http://resmedicinae.sourceforge.net/model/design/cybop/index.html, 2002.

[JC01] JASON CAI, RANJIT KAPILA, GAURAV PAL: *HMVC: The layered pattern for developing strong client tiers.* http://www.javaworld.com, 2001.

[Kun03] KUNZE, TORSTEN: *Untersuchung zur Realisierbarkeit einer technologieneutralen Mapping-Schicht für den Datenaustausch am Beispiel einer Anwendung zum medizinischen Formulardruck als integrativer Bestandteil eines Electronic Health Record (EHR).* Diplomarbeit, Technische Universität Ilmenau, 2003.

[OMG92] OMG: *The Common Object Request Broker: Architecture and Specification.* 1992.

[ope] *Open EHR.* http://www.openehr.org.

[Pre94] PREE, W.: *Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design.* In: *Proceedings of ECOOP '94*, Seiten 150–162, 1994.

[Str01] STRAUSS, FRIEDRICH: *Ergonomische Gestaltung von Benutzerschnittstellen.* Vortrag, 2001.

Abkürzungen

Abb.	Abbildung
DOM	Document Object Model
EHR	Electronic Health Record
engl.	englisch
griech.	griechisch
HMVC	Hierarchischer Model-View-Controller
HTML	Hypertext Markup Language
ICD	International Classification of Diseases
JDK	Java Development Kit
lat.	lateinisch
MVC	Model-View-Controller
SOAP	Subjective-Objective-Assessment-Plan
UML	Unified Modelling Language
vgl.	vergleiche
XML	Extensible Markup Language
XSL	Extensible Style Language

Übersetzungen wichtiger Begriffe

Abstract Factory	Abstrakte Fabrik
Assessment	Bemerkung, Einschätzung
Chain of Responsibility	Kette der Zuständigkeiten
Composite	Kompositum
Decorator	Dekorierer
Description	Beschreibung
Disease	Krankheit
Domain	Domäne
Electronic Health Record	Elektronische Patientenakte
Garbage Collection	Speicherbereinigung
Homunculus	künstlich erzeugter Mensch
Layer	Schicht
Objective	Befund
Observer	Beobachter
Overlay	Überzug
pathologisch	krankhaft
Plan	Vorgehen
Res Medicinae	Sache der Medizin
Separation of Concern	Trennung der Angelegenheiten
Strategy	Strategie
Subjective	Anamnese
TargetOverlay	Ziel-Overlay
Tier	Schicht, Ebene
Word Wheel	Wort-Rad

Thesen

- Die Einhaltung softwareergonomischer Prinzipien gewährleistet Akzeptanz einer Software durch den Benutzer und bedingt gleichzeitig ihren effektiven Einsatz.

- Benutzerfreundlichkeit einer Oberfläche wird durch den Einsatz eingabeunterstützender Komponenten, die ihrerseits ein Maß an "Intelligenz" besitzen, erhöht (Word Wheel).

- Konzepte der Natur (Lebenszyklus, Ontologien) sind bis zu einem gewissen Grad auch auf Probleme der Softwaretechnik übertragbar.

- Der Lebenszyklus von Softwarekomponenten ist eines dieser Prinzipien, die die Natur zum Vorbild nimmt. Durch ihn werden Objekte einem geordneten Kreislauf unterworfen, der die Kontrolle von Status und Abhängigkeiten dieser Entitäten ermöglicht.

- Die Entwicklung über Frameworks beschleunigt die Realisierung grosser Systeme. Gleichzeitig ist aber die genaue Analyse und Planung von Frameworks notwendig, da nachträgliche Modifikationen auch zu Änderungen der Systeme führen, die auf Frameworks basieren. Dieses vorausschauende Denken ist äußerst schwierig.

- Kaskadierung und Aggregation von Entwurfsmustern kann die Vorteile dieser zusammenführen und Wartbarkeit und Erweiterbarkeit von Software erhöhen. Ontologien sind ein Beispiel für ein solches Vorgehen.

- Ontologien, angelehnt an natürliche Hierarchien, gewährleisten die Kontrolle von Objektreferenzen durch eine eindeutig definierte Richtung der Assoziationen. Dies kann aber nur bei strikter Einhaltung der Regeln einer Ontologie der Fall sein.

- Ontologien machen den Gebrauch von Schnittstellen (Interfaces) sowie abstrakter Klassen aufgrund ihres streng hierarchischen Charakters überflüssig. Aber hier liegt auch die Kritik dieses Denkmodells: Die Einhaltung dieser Hierarchie ist zwingend notwendig und stellt damit (wie jedes Framework) auch ein Einschränkung der eigenen Kreativität des Entwicklers dar.

- Open Source verbindet das Wissen vieler unterschiedlicher Entwickler und macht Software auf Dauer erschwinglich. Open Source stellt zukünftig – aber auch schon teilweise heute – eine Alternative zu den kommerziellen Produkten der aktuellen Marktführer dar.
- Die triviale und sequentielle Auflistung von Patientendaten in Listen oder einfachen Tabellen offenbart Schwächen in der Übersichtlichkeit der Informationen und lässt zeitliche Zusammenhänge zwischen Problemen ausser Acht. Die episodenbasierte Befundung präsentiert einen alternativen Weg, medizinische Informationen zu strukturieren.
- Topologische Befundung unterstützt den Mediziner visuell bei der Anfertigung von Befunden und ist ein hilfreiches Werkzeug zur Lokalisation örtlich begrenzter pathologischer Erscheinungen.

Ilmenau, den 30. Dezember 2002

Jens Bohl

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel geschrieben zu haben.

Ilmenau, den 30. Dezember 2002

Jens Bohl

GNU Free Documentation License

Version 1.2, November 2002

Copyright©2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual

or work. Any member of the public is a licensee, and is addressed as ”you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A ”Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A ”Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The ”Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ”Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ”Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format

whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque

copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. C. State on the Title page the name of the publisher of the Modified Version, as the publisher. D. Preserve all the copyright notices of the Document. E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. H. Include an unaltered copy of this License. I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there

is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. O. Preserve any Warranty Disclaimers. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity

you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled ”History” in the various original documents, forming one section Entitled ”History”; likewise combine any sections Entitled ”Acknowledgements”, and any sections Entitled ”Dedications”. You must delete all sections Entitled ”Endorsements.”

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually

under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warrany Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.