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

von: Torsten Kunze

Betreuer: Dipl.-Ing. Christian Heller

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

Inventarisierungsnummer: 2003-01-02 / 005 / IN97 / 2232

eingereicht am: 02. Januar 2003

Danksagung

An dieser Stelle möchte ich mich ganz herzlich bei meinem Mentor Christian Heller bedanken, der mir stets mit guten Vorschlägen, beratenden Gesprächen und der Organisation von Literatur und Referenzen tatkräftige Unterstützung bot.
Einen wesentlichen Anteil tragen auch meine freundlichen Kommilitonen. Durch kritische Anmerkungen aber auch durch Aufforderung zu diversen kreativen Pausen und Diskussionen, aus denen ich neue Ideen schöpfen konnte, hatten sie großen Einfluss auf das Gelingen meiner Diplomarbeit.
Bedanken möchte ich mich auch bei meinen Eltern, meiner Schwester und meiner Freundin. Sie haben es mir ermöglicht, diese Ausbildung zu erhalten, mir immer die notwendige Unterstützung gegeben und sich für mich eingesetzt, wann immer es notwendig war.

Kurzfassung

Die vorliegende Diplomarbeit entstand im Zusammenhang mit einem informationstechnischen Projekt für den Einsatz in der Medizin. Es trägt den Namen *Res Medicinae*. Hierbei soll eine komponentenbasierte Anwendung realisiert werden, die es ermöglicht, Patientendaten zu verwalten und unter anderem auch graphisch auszuwerten.
Die einzelnen Komponenten sollen autonom als eigenständige Applikationen arbeiten, aber auch miteinander interagieren können.

Res Medicinae existierte zu Beginn der Diplomarbeit nur in Grundzügen. Unter anderem fehlte die Möglichkeit, Daten persistent zu sichern. Daher war ein Schwerpunkt der Diplomarbeit die Modellierung einer Persistenzschicht zu einem anzupassenden Domain-Modell, welches sämtliche Geschäftsdaten enthält. Eine zusätzliche, dem Anwender transparente Schicht zwischen Domain-Modell und Persistenzschicht verhindert die direkte Abhängigkeit der Persistenzlogik von den Domain-Daten und bildet je nach Funktion (Speichern oder Laden) jeweils den einen auf den anderen Teil ab. Diese Schicht wurde durch Umsetzung und Anpassung des Musters *Data Mapper* modelliert. In Voraussicht einer Verlagerung dieser Mapping-Schicht in einen separaten Prozess auf einen möglicherweise entfernten Rechner wurden die Untersuchungen für die Integration verschiedener Kommunikationsparadigmen wie Java RMI, JMS, CORBA in die Mapping-Schicht erweitert. Als Schwerpunkt und wesentliche Neuerung gegenüber früheren Ansätzen erhielt sie eine eigene Bezeichnung: *Layer PerCom* und kennzeichnet damit die Verwaltung bzw. Abbildung von Persistenz und Kommunikation in einer einzigen Schicht. Für die Persistenzmechanismen besteht bereits eine Implementation als Umsetzung in *Res Medicinae*. Die Interprozesskommunikation ist lediglich durch die abstrakte Definition der Klassen des zugehörigen Modells im Programmcode

enthalten. Damit ist die Basis für eine spätere praktische Realisierung geschaffen.
Zum Testen der Persistenzmechanismen und der Funktionsfähigkeit von *Layer PerCom* wurde der Prototyp eines Moduls für *Res Medicinae* entwickelt, der gespeicherte Informationen lädt bzw. neue Daten ablegen und modifizieren lässt und diese für den Druck von medizinischen Formularen aufbereitet.

Die Komplexität und der Umfang der Architektur des gesamten Softwaresystems *Res Medicinae* haben sich durch Hinzufügen der neuen Elemente um etwa ein Drittel vergrößert. Mit Hilfe der entworfenen Modelle und der darauf basierenden, teilweise praktischen Umsetzung ist es nun möglich, Daten lokal in XML-Dateien oder in einer entfernten relationalen Datenbank abzuspeichern und somit verteilt auf den selben Daten zu arbeiten. Die erneut vom jeweiligen Persistenzmedium geladen Informationen können in vorgefertigte medizinische Formulare ausgedruckt werden. Andere Module nutzen ebenfalls bereits *Layer PerCom* und die Persistenzschicht. In anknüpfenden Arbeiten, nach Implementation der modellierten Klassen zu den Kommunikationsparadigmen, kann die Funktionalität für eine gegenseitige Interaktion der einzelnen Module erweitert werden. Weiterhin ist abzusehen, dass eine Vielzahl zusätzlicher Informationen in die Datenbank und in das XML-Dateisystem aufgenommen werden muss, so dass die Erweiterung um Tabellen bzw. Tags ein Bestandteil weiterführender Untersuchungen sein könnte.

Ilmenau, den 02. Januar 2003 Torsten Kunze

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung und Motivation

Für medizinische Sachverhalte gibt es bereits eine Vielzahl an kommerziellen Programmen, die beim Lösen von Problemen oder Verwalten von Daten helfen sollen. Diese Diplomarbeit leistet neben den anschließend beschriebenen theoretischen Betrachtungen einen Teil zur Entwicklung einer *Open Source* Applikation für die Arbeit mit Patientendaten beliebiger Art.
Die Problematik bei den meisten Programmen besteht in der eingeschränkten Flexibilität bezüglich ihrer Erweiterbarkeit und in der schlechten Robustheit gegenüber Änderungen. Daher sollen Untersuchungen angestellt werden, ob die Verwendung von Software-Mustern diese Probleme minimiert.
Die Patientendaten werden in sogenannten *Electronic Health Records* (EHR) abgelegt und zur Verarbeitung im Programm gehalten. Es handelt sich deshalb bei einem EHR um einen eigens kreierten Datentyp, der aus mehreren unterschiedlichen, zusammengesetzten Datentypen besteht und patientenspezifische, medizinische Informationen enthält.
Werden die Daten nur auf eine Art gespeichert, so schränkt das die Flexibilität der Software sehr ein. Bei einer lokalen Speicherung bedarf es bei Anforderung von elektronischen Informationen einer direkten Kopie auf ein Wechselmedium und dessen physischen Transport zum anderen Rechner. Neben einer nicht vernachlässigbaren Zeitdauer und einem zusätzlichen personellen Arbeitsaufwand für diesen Vorgang treten häufig durch unsachgemäße Behandlung des Wechselmediums (z.B. heruntergefallene Diskette oder zerkratzte CD) Fehler auf, die bis zur völligen Unbrauchbarkeit der Daten führen können. Das etwaige Verlieren des Datenträgers kommt als Sicherheitsrisiko hinzu. Noch häufiger werden die Daten ausgedruckt,

per Post versendet und dann von einem anderen Arzt erneut in sein System übertragen. Auch hier handelt es sich um einen sehr zeitaufwendigen und unsicheren Prozess. Eine zentrale Datenbank innerhalb eines Netzwerkes schließt derartige Nachteile zwar aus, verhindert allerdings die Möglichkeit weiterzuarbeiten, falls das Netzwerk ausfällt. Daher wurde untersucht, inwieweit sich mehrere unterschiedliche Persistenzmechanismen (Dateisystem, Datenbank) in einer Anwendung vereinen lassen, ohne die Benutzerfreundlichkeit und Bedienbarkeit einzuschränken oder spezielles Fachwissen vom Anwender voraussetzen zu müssen.
Zu den theoretischen Betrachtungen dieser Diplomarbeit zählt noch ein weiterer Abschnitt, der sich damit befasst, wie Daten zwischen Prozessen ausgetauscht werden können. Da, wie bereits erwähnt, bei der Entwicklung der Anwendung Flexibilität und Erweiterbarkeit im Vordergrund stehen, darf die Einschränkung und Spezialisierung auf einige wenige Kommunikationsparadigmen, wie RMI, CORBA und JMS nicht erfolgen. Vielmehr soll eine Möglichkeit gefunden werden, den Anwender zur Laufzeit entscheiden zu lassen, welches Kommunikationsparadigma er verwenden möchte, wiederum ohne dass er über fachliche Kompetenz verfügen muss.
Eine Interprozesskommunikation ist für *Res Medicinae* notwendig, da dieses Softwaresystem aus einzelnen Modulen bestehen soll, von denen jedes als separater Prozess agiert, aber über die Möglichkeit verfügen muss, auf den selben Daten zu arbeiten, wie ein anderes Modul. Die Patientendaten zwischen zwei lokalen Prozessen auszutauschen ist der eine Aspekt. Ein Weiterer ist der Datentransfer zwischen mehreren Rechnern, denn damit können die lokal abgespeicherten Daten auf einem Rechner an einen anderen zur parallelen Weiterverabeitung gesendet werden.
In absehbarer Zeit wird das Projekt dahingehend erweitert werden, dass die Persistenzmechanismen auf einem entfernten Rechner installiert werden. Das macht die Entwicklung eines Konzepts zur Anbindung verschiedener Kommunikationsparadigmen ebenfalls zwingend erforderlich. Die Untersuchungen auf die Einbindung einer Art der Interprozesskommunikation in das System zu beschränken hätte nicht ausgereicht. Es soll für eine etwaige Notwendigkeit spezialisierter Protokolle die Möglichkeit der zukünftigen Anbindung gewährleistet werden. Der medizinische Formulardruck dient als prototypische Beispiel- und Testanwendung, um persistent abgelegte Daten wieder zu laden und sinnvoll zu verarbeiten.

1.1 Kapitelübersicht

Im Anschluss an diese Einleitung wird im zweiten Kapitel mit den Grundlagen für die in der Diplomarbeit verwendeten Software-Technologien begonnen. Es gibt eine Einteilung zu Mustern im Allgemeinen, bevor auf einige spezielle Beispiele eingegangen wird. Kapitel 3 beschäftigt sich mit der Einführung in die oben bereits angesprochenen Persistenzmechanismen und belegt deren Notwendigkeit. Während das zweite und dritte Kapitel den *State of the Art* beschreiben, diskutieren die anschließenden Abschnitte den Eigenanteil an der Diplomarbeit. So erläutert das vierte Kapitel die Umsetzung der in den vorangegangenen Abschnitten beschriebenen Technologien. Der fünfte Abschnitt stellt den zum Testen und Anwenden der Modelle entwickelten Prototyp vor. Ein letzter Punkt zieht noch einmal Rückschlüsse aus der Umsetzung und gibt Empfehlungen für weiterführende Realisierungsmöglichkeiten.

1.2 Besondere stilistische Kennzeichnungen

Im laufenden Text sind einige Begriffe durch *kursive Schrift* hervorgehoben. Hierbei handelt es sich fast ausschließlich um fremdsprachige, meist englische Bezeichnungen, die entweder im laufenden Text oder im Glossar des Anhanges Erläuterung finden.
Der Stil `Typewriter Typeface` wurde verwendet, um Java-Programmzeilen hervorzuheben. An den entsprechenden Stellen wird zusätzlich gesondert darauf hingewiesen.

1.3 Res Medicinae

Ein Wort noch zu dem Softwaresystem *Res Medicinae* im Ganzen.
Unterstützung findet das Projekt durch die weltweit größte *Open Source* Software Entwicklungswebseite im Internet. Es handelt sich hierbei um SourceForge.net [OSDN02]. Hier werden Dienste für *Open Source* Entwickler frei angeboten. So findet man ausführliche Dokumentationen und Mailinglisten zu sämtlichen unterstützten Projekten, die im umfangreichsten momentan existierenden *Open Source* Repository des Internets hinterlegt sind.

Der Name *Res Medicinae* stammt, wie es für die Medizin typisch ist, aus dem lateinischen Sprachgebrauch und bedeutet übersetzt *Sache der Medizin*. Damit ist das Einsatzgebiet der Anwendung bereits abgesteckt. Sie dient der Verwaltung und Darstellung medizinischer Daten, die während der Bearbeitung in dem bereits beschriebenen, eigens spezifizierten Datenmodell *Electronic Health Record* gehalten werden.

Viele in den Arzt-Praxen und Krankenhäusern verwendete Softwaresysteme zur Verwaltung von Patientendaten sind veraltet und neue sehr teuer. Res Medicinae steht als *Open Source* Projekt unter GNU GPL/FDL Lizenzvereinbarungen [FSF99] und ist damit kostenfrei erhältlich. Den große Vorteil gegenüber kommerzieller Software stellt der frei verfügbare Quellcode dar. Jeder Entwickler kann Anpassungen und Erweiterungen an den Applikationen vornehmen, sofern er die resultierende Software nicht zu kommerziellem Vertrieb weiterverwendet und sie ebenfalls offenlegt. Gleiches gilt auch für die Dokumentation des Projektes. Die uneingeschränkte Zugänglichkeit und Erweiterbarkeit des Quellcodes soll ein rasches Wachstum und eine möglichst weite Verbreitung gewährleisten.

Für das Gesamtsystem wurde sich ebenfalls bemüht, ausschließlich *Open Source* Programme, beispielsweise das Datenbankmanagementsystem PostgreSQL zu verwenden, um somit keine Kosten für den späteren Anwender entstehen zu lassen.

Die Programmierung erfolgte ausschließlich in Java. Damit wird die Plattformunabhängigkeit sichergestellt.

Es gibt verschiedene Möglichkeiten, Daten persistent abzulegen. Entweder lokal in Dateien oder aber auch auf einem entfernten Rechner in einer zentralen Datenbank für Zugriffe von mehreren Workstations aus. Für die lokale Speicherung existiert eine Vielzahl weiterer Möglichkeiten. Jedoch wurde hier ein XML-Format gewählt, weil es sich dabei um einen anerkannten Standard [OAS99] handelt, der im Zuge der Entwicklung des Internets permanent an Bedeutung gewinnt und bereits in den verbreitetsten Programmiersprachen verwendbar ist.

Die meisten Softwareprojekte basieren auf derartigen Standards. Doch oftmals wird zu spät erkannt, dass zwar komplexe Features implementiert wurden, aber eine Erweiterung um andere Standards nur schwer realisierbar ist. Mit Hilfe von Softwaremustern kann man dieses Problem minimieren. Allerdings bedeutet es oftmals einen nicht geringen Mehraufwand an

Entwurfs- und Implementierungsarbeit.

Der im Rahmen dieser Diplomarbeit entwickelte Ansatz ermöglicht eine flexible Expandierung des Projektes, was den erweiterten Realisierungsaufwand rechtfertigt.

Kapitel 2

Muster in der Softwaretechnik

Schon in frühen Zeiten hat der Mensch erkannt, dass in unterschiedlichen Vorgängen und Gebilden gewisse Äquivalenzen in Strukturen und Vorgehensweisen existieren. Beispielsweise wird ein Haus immer nach dem selben Schema gebaut: Es beginnt mit einer Drainage, darüber ein solides Fundament auf dem die Wände fußen, welche wiederum das Dach tragen. Somit existiert nicht nur ein Vorgehensmodell, sondern auch ein grobes Schema, welche Teile auf welche Weise zusammengehören. Der Vorteil dieses Musters für den Hausbau ist, dass das Haus immer ein Haus wird und beliebig erweitert bzw. modifiziert werden kann und daher trotz eines einheitlichen Grundkonzepts sehr individueller Gestalt sein kann.

Allgemein gültig formuliert ist ein Muster die Abstraktion einer konkreten Form, welche sich in spezifischen, unwillkürlichen Kontexten wiederholt. Es bietet meist eine Lösungsidee aber keine Fertiglösung.

Die resultierenden Vorteile zu nutzen, ist auch in der Softwaretechnik erwünscht. Es ist prinzipiell simpler und günstiger, auf ein bewährtes Modell aufzubauen und dieses seinen Bedürfnissen anzupassen, als zu einem bestehenden Problem eine vollständig neue Entwicklung zu versuchen.
"Muster werden nicht erfunden, Muster werden gefunden" [Har02]. Die Muster basieren auf jahrelangen Erfahrungen vieler professioneller Softwareentwickler. Täglich stehen neue Anforderungen und werden neue Erfahrungen gemacht, so dass man nicht davon ausgehen darf, dass bereits für jedes Problem das perfekte Muster existiert. Ganz im Gegenteil. Man geht

davon aus, dass bisher erst ein sehr kleiner Teil möglicher Muster erkannt wurde. Es ist nicht sinnvoll die Lösung eines Problems in ein oder mehrere Muster zu pressen, wenn sich der damit verbundene Aufwand ungerechtfertigt erhöht. Vielmehr sollen Muster genau das Gegenteil bewirken, die Wiederverwendbarkeit, Portabilität, Flexibilität, Verständlichkeit, Modularität erhöhen und Software damit robuster gegenüber Änderungen und Erweiterungen machen [Gam96].
Oftmals sind einzelne Muster in der Softwaretechnik nur wenig sinnvoll, aber im Verbund mit anderen Ausgewählten, erfüllen sie die eben angeführten Ziele.

2.1 Untergliederung von Mustern

Inzwischen gibt es eine Vielzahl an unterschiedlichen Mustern in den verschiedensten Bereichen der Softwaretechnik. Daher hat man gewisse Strukturierungen eingeführt, um sie ihrem Einsatzgebiet entsprechend zu sortieren und einen besseren Überblick zu erhalten.

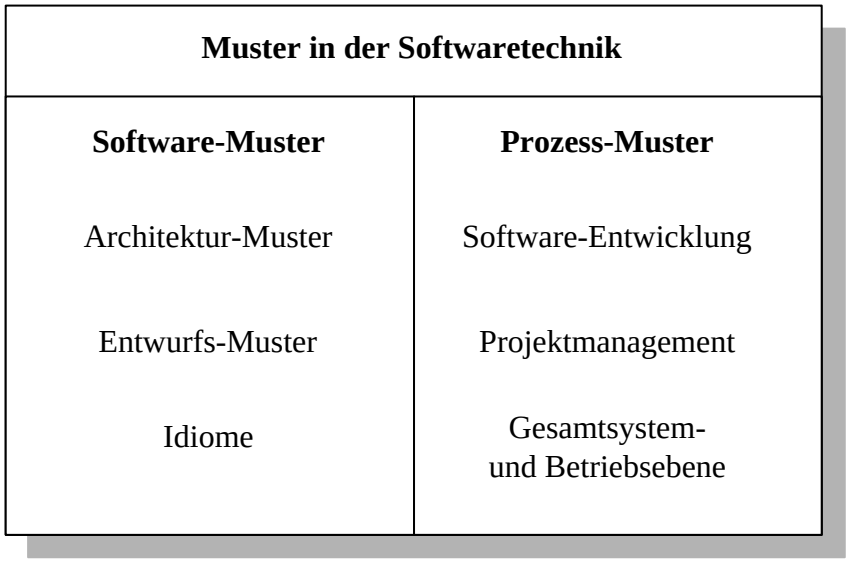

Abbildung 2.1: Einteilung der Muster

Wie zu erkennen ist, gibt es in der Softwaretechnik eine grobe Untergliederung in Muster, die zum einen die praktische Realisierung von Softwaresystemen und zum anderen gesamte

Prozesse auf jeweils unterschiedlichen Abstraktionsebenen betreffen. Da der Schwerpunkt der Diplomarbeit auf der ersten Gruppe liegt, wird im Folgenden hauptsächlich auf die Kategorien der Software-Muster [FB98] und nur kurz auf Prozess-Muster [Har02] eingegangen.

2.1.1 Prozess-Muster

Ein Prozess-Muster befasst sich mit den Aktivitäten der Softwareentwicklung und deren Reihenfolge. Es beinhaltet Wissen über Organisation und Abläufe, die für die Softwareentwicklung wesentlich sind.
Durch die Schnelllebigkeit von Technologien und Systemen ist die Interoperabilität zwischen Plattformen und Systemen ein wesentlicher Punkt, um überhaupt funktionsfähige, erneuerbare Systeme zu erhalten. Daher hat die Object Management Group (OMG) einen Standard definiert, auf dem Systeme und Produkte aufbauen können, um somit eine Basis für eine funktionierende Interoperation zu legen. Dieser Standard nennt sich *technology adoption process* [OMG02].

2.1.2 Software-Muster

Architekturmuster

Hierbei handelt es sich um Muster, die das gesamte Softwaresystem betreffen. Es werden Möglichkeiten beschrieben, die eine grundsätzliche Strukturierung umfassen, d.h. die Unterteilung in Subsysteme und das Festlegen von Schnittstellen zwischen diesen. Ein Beispiel für ein Architekturmuster ist *Layers*. Damit lassen sich in Teilaufgaben zerlegbare Anwendungen auf verschiedene Ebenen strukturieren. Die Teilaufgaben werden zu Gruppen der selben Ebene zusammengefasst, z.B. das ISO-OSI-Schichtenmodell oder die in Abschnitt 3.1.7 beschriebenen N-Tier-Schichtenmodelle.

Entwurfsmuster (Design Pattern)

Auf der nächst niedrigeren Abstraktionsebene greifen die Entwurfsmuster. Mit ihrer Hilfe können die Subsysteme, einzelne Komponenten oder Beziehungen zwischen diesen Komponenten strukturiert werden. Es existieren drei unterschiedliche Arten von Design Pattern [Gam96]:

- **Strukturmuster**

  Diese Kategorie der Entwurfsmuster beschreibt mögliche Kompositionen von Objekten und Klassen, um größere Strukturen zu bilden, die dennoch die zu Anfang des Kapitels angeführten Ziele von Mustern, wie Wiederverwendbarkeit und Erweiterbarkeit, gewährleisten.

- **Erzeugungsmuster**

  Sie dienen der Generierung von Objekten und zum Verstehen des Erzeugungsprozesses. Damit sollen komplette Softwaresysteme gebildet werden, die möglichst unabhängig von der Erzeugung, Zusammensetzung und Repräsentation ihrer Objekte sind.

- **Verhaltensmuster**

  Eine letzte Gruppe der Entwurfsmuster befasst sich mit Algorithmen und der Zuweisung von Zuständigkeiten zu Objekten. Sie beschreiben nicht nur Muster von Objekten und Klassen, sondern auch Interaktionen zwischen diesen und damit komplexe Kontrollflüsse, die zur Laufzeit nur schwer nachvollziehbar sind. Das heißt also, Verhaltensmuster beschreiben, wie Objekte untereinander interagieren.

Später werden in diesem Kapitel einige konkrete Beispiele zu diesen Mustertypen erläutert, die bei der Lösung der gestellten Aufgaben für diese Diplomarbeit zum Einsatz kommen.

Idiome

Sowohl Architekturmuster als auch Entwurfsmuster sind unabhängig von der verwendeten Programmiersprache und dem zugrunde liegenden Programmierparadigma. Das ist bei Idio-

men nicht der Fall. Sie stellen die unterste Ebene der Abstraktion dar und beziehen sich auf Problemlösungen für konkrete Programmiersprachen in Bezug auf Entwurf und Implementierung. Beispielsweise ist die Verwendung von Templates in C++ ein gern benutztes Verfahren, in Java jedoch nicht möglich.

Gleichgültig ob bei der Analyse, dem Entwurf oder einzelnen Prozessen, können je nach Erfahrung des Entwicklers in jedem Bereich gute und schlechte Lösungen produziert werden. Eine Möglichkeit, weniger gute Lösungen zu verbessern oder alte Systeme zu erweitern, bieten die *Anti Pattern*. Mit ihrer Hilfe lassen sich Muster in schlechten Lösungen finden. Diese geben Vorschläge und Methoden zur Verbesserung des Bestehenden. Man spricht dann auch von so genannten *Amelioration Pattern*, den Verbesserungsmustern [Gam02].

2.2 Einige ausgewählte Muster im Detail

Die hier im folgenden aufgeführten Muster kommen in den für die Diplomarbeit zu entwickelnden Modellen und Lösungen zum Einsatz. Mit Ausnahme des MVCs handelt es sich dabei um einige ausgewählte Muster für *Enterprise Application Architectures* [Fow02]. Wie in einem späteren Kapitel beschrieben, werden sie jedoch nicht in reiner Form übernommen, sondern den Anforderungen entsprechend angepasst.

2.2.1 Model View Controller - MVC

Das wohl bekannteste Architekturmuster ist Model View Controller. Es dient zur Strukturierung der Präsentation interaktiver Softwaresysteme.
Hierbei werden die unterschiedlichen Bereiche, bestehend aus Darstellung, Datenhaltung und Verwaltungsoperationen aus einem einzelnen Objekt entkoppelt und auf mehrere Objekte verteilt, um damit die Flexibilität und Wiederverwendbarkeit zu erhöhen. Es können durchaus mehrere View-Objekte existieren, die auf das selbe Model-Objekt zugreifen, aber unterschiedliche Darstellungsformen besitzen, beipielsweise Präsentation der Daten jeweils als Balken-, Kreis- oder Liniendiagramm. Allerdings müssen die View-Objekte sicherstel-

len, dass sie immer dem aktuellen Model-Zustand entsprechen. Um dies zu gewährleisten, kommt ein spezielles Entwurfsmuster zum Einsatz, das *Observer*- oder Beobachtermuster. Es bewirkt, dass bei einer Veränderung des Models sämtliche angemeldeten View-Objekte informiert werden, worauf sich diese selbständig aktualisieren.
Für die Umsetzung ergibt sich damit folgende Struktur:

- Das Model-Objekt: ist das Anwendungsobjekt. Es kapselt sämtliche Daten und enthält die Kernfunktionen.
- Das View-Objekt: stellt die Bildschirmpräsentation der Daten dar.
- Das Controller-Objekt: bestimmt die Möglichkeiten, wie Benutzungsschnittstellen auf Benutzereingaben reagieren.
- Das Beobachter-Objekt: ist ein zusätzlicher Bestandteil des MVC, der die Wahrung der Konsistenz zwischen Model und View gewährleistet. Seine Nutzung wird im klassischen MVC empfohlen.

Auch das *Composite Pattern*, ein weiteres Entwurfsmuster, kann beim MVC zum Einsatz kommen. Es beschreibt, wie Objekte zusammengefasst und als eine Einheit behandelt werden können. So kann ein View-Objekt aus Unter-Views bestehen, die einzeln besser zu verwalten sind. Die damit entstehende Hierarchie kann bis zu primitiven Objekten wie Buchstaben heruntergebrochen werden.
Ein vergleichbarer Ansatz dazu ist das HMVC [Fow02] (*Hierarchical Model View Controller*). Hierbei wird, wie die Bezeichnung bereits vermuten lässt, eine Hierarchie einzelner Komponenten erzeugt, die jeweils über Model, Controller und View verfügen. Nur der Controller der darüber liegenden Ebene kennt die Controller der nächst niedrigeren Stufe. Die Beziehungen auf einer Ebene sind identisch zum MVC.
Ein weiteres, jedoch nicht das letzte beim MVC anwendbare Muster soll kurz Erwähnung finden: das Strategy-Pattern. Die Art und Weise, wie der Controller auf bestimmte Nutzereingaben reagiert, soll nach Möglichkeit und entsprechend dem Einsatzgebiet austauschbar sein. Dafür werden seine Antwortalgorithmen in einem zusätzlichen, leicht ersetzbaren Objekt gekapselt, an welches der Controller die Anfragen delegiert. Die Delegation selber entspricht

wiederum einem Strukturmuster.

Das Model View Controller -Muster verdeutlicht noch einmal die in Abschnitt 2.1 angeführte Einteilung in Abstraktionsebenen. Während das MVC ein Architekturmuster ist, verwendet es für die einzelnen Software-Subsysteme diverse Entwurfsmuster.

2.2.2 Layer Supertype

Eines der wohl bekanntesten und am häufigsten verwendeten Entwurfsmuster ist *Layer Supertype*. Dabei wird für alle Klassen einer Ebene, die verwandten Typs sind, eine gemeinsame Superklasse entwickelt, die die gemeinsamen Attribute und Methoden in sich vereinigt. Damit wird das wiederholte Schreiben des selben Quellcodes in unterschiedlichen Klassen verhindert. Alle Subklassen erben damit die Eigenschaften des Supertyps. Ein weiter Vorteil

```
class A {
   void relax() { ... }
}
class B extends A {
   void relax() { ... }
}
class C extends B {
   void relax() { ... }
}
...
A a1 = new A();
A a2 = new B();
A a3 = new C();
...
a1.relax();          // relax() aus A
a2.rleax();          // relax() aus B
a3.relax();          // relax() aus C
...
a1 = a2;
a1.relax();          // relax() aus B
((C)a1).relax();     // relax() aus C
```

Abbildung 2.2: Java-Code-Beispiel zu dynamischem Binden

der sich daraus ergibt, ist das dynamische Binden. Im Gegensatz zum statischen Binden, bei dem der Typ einer Variablen bereits zur Übersetzungszeit feststeht, wird beim dynamischen oder auch späten Binden erst zur Laufzeit entschieden, welche konkrete Realisierung einer vom Supertyp geerbten Klasse als Datentyp zum Einsatz kommt. Ein Beispiel dazu zeigt Abbildung 2.2.

2.2.3 Domain Model

Es existiert bereits eine Vielzahl an Vorschlägen, wie man komplexe Geschäftsdaten mit Mustern strukturieren kann. Das hier beschriebene Domain Model [Fow02] ist einer von ihnen. Dabei wird ein Netz kommunizierender und unterschiedlich komplexer Objekte erzeugt. Sowohl Daten als auch Verarbeitungsalgorithmen bzw. ein eigenes Verhalten sind darin enthalten. Es gibt Objekte, die Daten enthalten und Objekte, die Logik implementieren. Weiterhin können auch Daten und Methoden zusammen in einem Objekt auftreten.
Man kann ein Domain Model als eigenständige Schicht in einem Softwaresystem betrachten, die ebenfalls weitere Muster verwenden darf. So kommen häufig *Layer Supertype* und das Strategiemuster zum Einsatz.
Eine der wichtigsten Regeln für das Domain-Modell ist seine prinzipielle Unabhängigkeit von den externen Schnittstellen, so dass eine Veränderung eines der beiden keine Anpassung des anderen erzwingt.
Liegt dem Softwaresystem eine Datenbank zugrunde und ist das Domain Model hinreichend komplex, kombiniert man es häufig mit einem *Data Mapper*.

2.2.4 Data Mapper

Sollen Datenbankschema und Objekt-Modell unabhängig voneinander entwickelt werden, so kommt der Data Mapper [Fow02] zum Einsatz. Er stellt eine, für den Anwender transparente Zwischenschicht dar, die sämtliche Daten von den Domain-Objekten in die Datenbank oder ein anderes Repository transferiert und auch wieder herausliest.
Zum Strukturieren der jeweiligen Daten verfügen Objekte und relationale Datenbanken

über völlig unterschiedliche Mechanismen und Datentypen. Daher benötigt man zusätzliche Operationen, die eine Möglichkeit des Datenaustausches bieten und somit als Vermittler zwischen diesen beiden Schichten arbeiten.
Der enorme Vorteil des Data Mappers ergibt sich daraus, dass die Datenbank beim Arbeiten mit dem Domain-Modell vernachlässigt werden kann. Allerdings ist die zusätzliche Schicht mit einem erheblichen Mehraufwand an Implementierungsarbeit verbunden.
Die Implementierung der SQL-Statements, das sind unter anderem die Finder-Methoden, sollte nicht in der Mapping-Schicht erfolgen, da sonst die Unabhängigkeit des Domain-Modells vom Mapper nicht mehr bestehen würde. Daher empfiehlt es sich, diese Methoden in einem separaten Package zu halten. Eine andere Möglichkeit ist die Definition von Finder-Schnittstellen im Domain-Modell, die von der Mappingschicht implementiert werden.

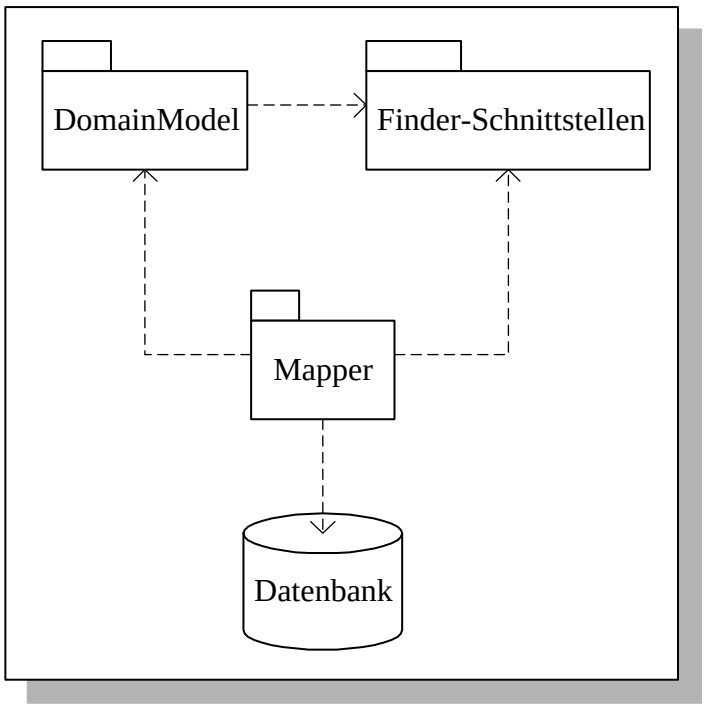

Abbildung 2.3: Abhängigkeiten der drei Schichten

Abbildung 2.3 zeigt die Abhängigkeiten der drei Schichten. Der Mapper ist abhängig von der Datenbank und vom Domain-Modell. In entgegengesetzte Richtung gilt das nicht. Die Domain-Objekte kennen ihre zugehörigen Finder-Schnittstellen. Damit ist es ihnen beispiels-

weise möglich, ein eigenständiges Update bzw. Laden von Informationen aus der Datenbank zu initiieren. Die Implementation der entsprechenden Schnittstelle gehört zur Mappingschicht. Von dort werden die aktuellen Daten an die Domain-Objekte übergeben.

2.2.5 Remote Facade

Während es sich anbietet beim objektorientierten Paradigma viele kleine feingranulare Objekte mit kurzen Methoden zu verwenden, führt dieses Modell bei Interprozesskommunikation zu einer weniger günstigen Performance. Bei lokalen Prozessen muss lediglich zwischen den Adressräumen gewechselt werden, was sich daher nicht ganz so stark in den Laufzeiten niederschlägt. Hingegen sind ferne Aufrufe erheblich langsamer. Daher empfiehlt es sich, zusammengehörige Daten gruppiert zu versenden, um die Anzahl der gegenseitigen Zugriffe von Client und Server zu verringern und damit die Prozessabarbeitung zu beschleunigen. Diese Aufgabe erfüllt die Remote Facade [Fow02]. Auf die feingranularen Objekte wird über eine grobgranulare Schnittstelle zugegriffen, deren Implementierung sich die notwendigen Daten zusammensucht. Weder verfügen die feingranularen Objekte über eine Remote-Schnittstelle, noch enthält die Remote Facade Teile der Domain-Logik. Damit besteht eine saubere Trennung zwischen Aufrufen von außerhalb des Prozesses und der internen Verarbeitung der Daten.
Unter Anwendung dieses Musters können beispielsweise Informationen wie Adresse, Kontodaten, Dimensionen eines Gegenstandes usw. die sich alle aus mehreren Attributen zusammensetzen, jeweils durch einen einzigen fernen Aufruf der entsprechenden Server-Methode - auch als *Remote Call* bezeichnet - übermittelt werden.
Nachteil dieses Verfahrens ist der größere Implementierungsaufwand.
Häufig werden die Daten in so genannten *Data Transfer Object*s zusammengefaßt. Dieses Muster beschreibt der folgende Abschnitt.

2.2.6 Data Transfer Object (DTO)

Oftmals muss beim Versenden von Daten über ein Netzwerk mit einem einzigen Transfer eine Vielzahl an Parametern übertragen werden. Bei Java dürfen Methoden aber maximal nur einen Rückgabewert haben. Daher wäre für jede Information ein eigener Aufruf notwendig, was sich wiederum zu Lasten der Performance auswirkt. Die Lösung bringt das Data Transfer Object [Fow02]. Es ist ein Objekt, das alle Informationen kapselt. Damit muss nur ein Parameter versendet werden.
Auch wenn der Client nicht immer alle Daten, die das Data Transfer Object enthält, benötigt, so werden sie dennoch gesendet. Wie bereits im vorigen Abschnitt erläutert wurde, ist es aus Sicht der Performance jedoch günstiger, mehr Daten in einem Aufruf zu übermitteln, als viele kleine Sendungen durchzuführen.
Nicht nur für den Transport von Daten über ein Netzwerk sind DTO's sinnvoll. Bei jeder Interprozesskommunikation können sie sich positiv auf den Datenverkehr und damit die Bearbeitungsgeschwindigkeit auswirken.
Eine Antwort auf die Frage, wieviele DTO's verwendet werden sollen, ist nicht leicht zu geben. Das hängt davon ab, wofür sie verwendet werden. Gewöhnlich orientiert sich die Struktur an den Bedürfnissen des Clients. Ein einzelnes *Data Transfer Object* ist leichter zu implementieren, während man bei spezialisierten DTO's einfacher erkennen kann, welche Daten zu welchem Aufruf versendet werden. Für sehr unterschiedliche Aufrufe sollten auch unterschiedliche DTO's verwendet werden, damit die Redundanz der gesendeten Daten hinreichend gering gehalten wird.
Mittels *Serialisierung* werden die Daten in einem binären Format zurück zum Client übertragen. Hierbei können allerdings Probleme bei der Synchronisation von Client- und Server-Objekt auftreten. Wird die DTO-Struktur auf Serverseite modifiziert, so kann es passieren, dass der Client aufgrund nun bestehender Typdifferenzen das Objekt nicht deserialisieren kann und somit alle gesendeten Informationen verloren gehen. Selbst eine recht einfache Änderung des DTO, wie das Hinzufügen eines optionalen Feldes, ruft diesen Effekt hervor. Daher kommt man zur Überlegung, ob nicht andere Möglichkeiten der Serialisierung existieren, welche die Klassen toleranter gegenüber derartigen Änderungen machen. Der

textbasierte Ansatz liefert eine Lösung. Mittels der *Extensible Markup Language* (oder kurz: XML) können Daten recht einfach in textueller Form hinterlegt werden. Die populärsten Programmiersprachen, unter anderen auch Java, bieten bereits ausgereifte Algorithmen zur Verarbeitung dieses international anerkannten Standards an. Ein wesentlicher Nachteil ist die höhere Bandbreite, die im Gegensatz zur binären Form benötigt wird.

Zusammensetzung eines DTO aus dem Domain-Modell

Die in dem *Data Transfer Object* gehaltenen Daten werden aus allen Server-Objekten zusammengesucht, von denen das Remote-Objekt und damit der Client, Informationen anfordert. *Data Transfer Object* und Domain-Objekte dürfen nicht unmittelbar voneinander abhängen. Die Änderung eines der beiden soll nicht unbedingt auch eine Modifikation des anderen erzwingen. Daher verwendet man zusätzlich ein Assembler-Objekt, dass zum einen aus den Domain-Objekten das DTO zusammensetzt und zum anderen am anderen Ende des Kabels das erhaltene Data Transfer Object wieder zerlegt und die entsprechenden Daten an die Domain-Objekte übergibt.
Die Attribute eines DTO können entweder einfache Objekte oder andere DTOs darstellen, so dass im Gegensatz zu der meist sehr aufwändigen Struktur des Domain-Modells eine recht simple hierarchische Struktur entsteht.

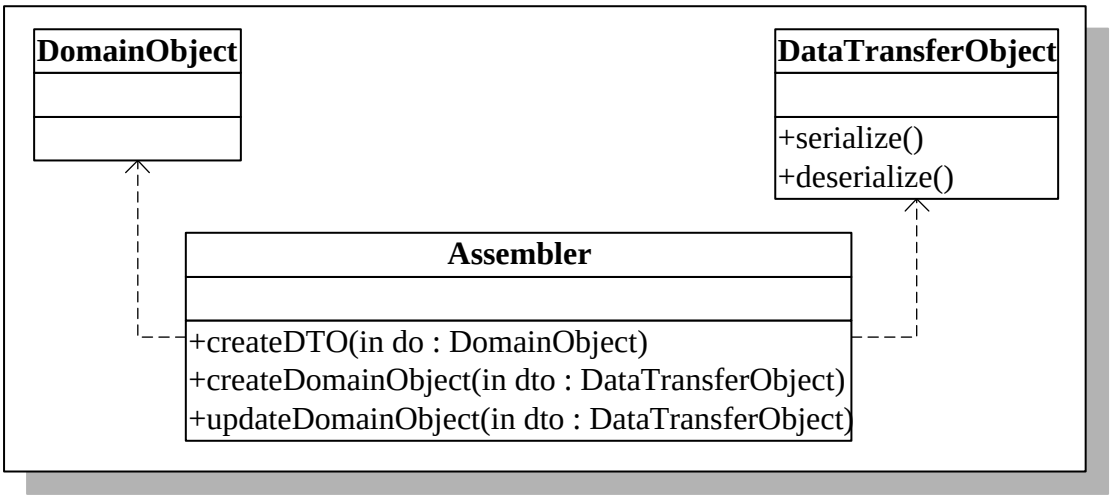

Abbildung 2.4: Beziehungen zwischen DTO, Assembler und Domain-Objekt

Kapitel 3

Persistenzmechanismen

Ein wesentlicher Bestandteil der Diplomarbeit war die Erzeugung einer Mappingschicht, die verwendete Mechanismen zum Speichern, Laden und Modifizieren von Datensätzen dem Anwender gegenüber transparent erscheinen lassen soll. In diesem Kapitel werden die Grundlagen zu den beiden verwendeten Mechanismen beschrieben. [Hel02]

3.1 Datenbanken und JDBC

Der große Vorteil beim Verwenden von zentralen Datenbanken auf einem entfernten Rechner ist die globale Zugänglichkeit sämtlicher Daten. Viele Anwender können simultan auf die gleichen Informationen zugreifen.

Von einer guten Datenbankanwendung erwartet man, dass der Anwender nicht erkennt, dass er auf eine Datenbank zugreift, sowie in welcher Art diese Aktionen ausgelöst und abgearbeitet werden. Da *Res Medicinae* eine reine Java-Applikation ist, beschäftigt sich dieses Kapitel hauptsächlich mit Datenbankanbindungen für diese Programmiersprache und Grundlagen zu JDBC, was für *Java Database Connectivity* steht.

Doch vorher ein paar Worte zu den notwendigen Bestandteilen einer lauffähigen Datenbankanwendung.

3.1.1 DBMS

Einer der wichtigsten Bestandteile ist das Datenbankmanagementsystem oder kurz DBMS. Wie der Name schon sagt, dient ein DBMS zur Verwaltung, d.h. Erstellen und Löschen von Datenbanken und Tabellen, sowie Einfügen, Abfragen und Aktualisieren von Datensätzen. Sollen Informationen aus einer Datenbank abgerufen werden, so erfolgt dass immer über das DBMS. Als erstes muss eine Verbindung zu diesem hergestellt werden. Es kennt die enthaltenen Datenbanken, kann somit die Anfragen entsprechend weiterleiten und die Ergebnisse an den Requestor zurücksenden.
Datenbankmanagementsysteme verfügen aber nicht nur über die Möglichkeit, Informationen abzuspeichern, sondern müssen auch zusätzliche Anforderungen erfüllen. So enthalten sie [Con02]:

- ein Sicherheitssystem, um die Daten vor nicht autorisierten Zugriffen zu schützen,
- ein Integritätssystem, um die Konsistenz der Daten sicherzustellen,
- ein Konkurenzkontrollsystem, zum Verwalten simultaner Zugriffe,
- ein Wiederherstellungssystem, welches im Falle eines Fehlers den letzten bekannten konsistenten Zustand der Datenbank wiederherstellt,
- einen Anwender-zugänglichen Katalog, der Beschreibungen der gespeicherten Daten enthält, sogenannte Metadaten.

3.1.2 SQL

Die Definition der Datenbanken erfolgt mittels *Data Definition Language* (DDL). Sie gestattet es dem Anwender Datentypen, Strukturen und Anweisungen zu spezifizieren. Eine *Data Manipulation Language* (DML) hingegen erlaubt es, Daten einzufügen, zu aktualisieren, zu löschen und abzufragen. Verfügt man über ein zentrales Repository, wie eine Datenbank, so bietet die DML eine allgemeine Möglichkeit, um Daten und Metadaten in Erfahrung zu bringen. Man spricht hier von *Query Languages* [Con02].

Die Bekannteste von ihnen ist SQL, die Structured Query Language. Sie hat sich zu einem Standard entwickelt. Anweisungen, die diesem Standard genügen versteht das DBMS.
Da, wie bei anderen Standards auch, über die Zeit Weiterentwicklungen und Anpassungen vorgenommen wurden, haben sich bei SQL unterschiedliche Versionen etabliert. Das führt natürlich zu Problemen bei der Portierung von Applikationen auf unterschiedliche Datenbankmanagementsysteme, da sich die Hersteller nicht auf einen Standard einigen. Vielmehr verwendet jeder den für seine Anwendung passendsten. Selbst die Behauptung zweier unterschiedlicher Hersteller, jeweils SQL2 zu unterstützen, kann unterschiedliche Bedeutung haben. Während es für SQL2 bzw. SQL92 (ISO/IEC 9075:1992 Information technology - Database language) drei Abstufungen gibt [Klu98]:

1. Entry Level: Enthält die wichtigsten Features und damit auch die geringsten Anforderungen an die Implementierung eines DBMS

2. Intermediate: Baut auf Entry Level auf

3. Full SQL: Das ist die oberste Stufe und unterstützt somit den gesamten Sprachumfang,

enthält der aktuelle Standard (ISO/IEC 9075:1999(E)) Information technology - Database languages) auch bezeichnet als SQL3 oder SQL99 eine derartige Hierarchie nicht mehr. Er umschließt eine Vielzahl neuer Features [Han02].
Leider werden von vielen DBMS nicht alle Standards unterstützt. An Stelle dessen versuchen sie sich durch eigene Features von anderen Systemen abzuheben. Man spricht von sogenannten proprietären Schnittstellen, über die entsprechende Features genutzt werden können. Das macht es dem Entwickler nicht gerade einfach, eine portierbare Applikation zu entwickeln, muss er doch somit über eine genaue Kenntnis jedes verwendeten DBMS und seiner proprietären Eigenschaften verfügen.
Inzwischen haben sich bereits sehr viele unterschiedliche Features in den verschiedenen DBMS herausgestellt, so dass es nicht möglich ist, eine SQL-konforme Testanwendung zu entwickeln, die herstellerunabhängig ist. Das wirkt sich natürlich auch enorm auf die Softwarekosten aus. Einige Entwickler beginnen bereits zu zweifeln, ob SQL noch als einheitlicher Standard anerkannt werden sollte [Sei02].

3.1.3 Embedded SQL

Schnittstellenfunktionen können nicht immer in SQL ausgedrückt werden, weil die Sprache nicht *computational complete* ist, d.h. verschiedene Berechnungen und Operationen, wie beispielsweies Rekursionen, sind nicht in SQL ausführbar. Vielmehr ist ein zusätzlicher Befehlssatz notwendig, um fortgeschrittene Techniken für die Manipulation der Daten zu unterstützen. Die SQL-Anweisungen, man nennt sie auch SQL-Statements, müssen unmittelbar aus einer Applikation heraus gestartet und von dieser auch ausgewertet werden. Deshalb hat man es als sinnvoll erachtet, Programmiersprachen zu erweitern und SQL direkt in diese "einzubetten". Daher auch die Bezeichnung *Embedded SQL*. Besonders die Auswertung der von dem Datenbanksystem zurückgelieferten Informationen wird somit wesentlich vereinfacht, denn die Ergebnisse können direkt oder zumindest mit geringem Aufwand in die eigens vorgesehenen Variablen geschrieben werden.

3.1.4 Der Weg zur Java Database Connectivity (JDBC)

Einen weitverbreiteten Mechanismus für Datenbankzugriffe stellt ODBC (*Open Database Connectivity*), bestehend aus einem *Application Programming Interface* (API) für C/C++ und einem für die Datenbankzugriffe verantwortlichen Treiber, dar. Das API bietet eine Anzahl einheitlicher Schnittstellen an, über die entsprechende Dienste aufgerufen werden können. Der Vorteil von Schnittstellen ist die Unabhängigkeit von der jeweiligen Implementation der Funktionalität des DBMS. Da die unterschiedlichen DBMS verschiedene Features beinhalten und selbst für einheitliche Dienste herstellerspezifische Algorithmik und Methodik verwenden, übernimmt es ein Treiber, die Anweisungen auf das jeweilige DBMS abzubilden. Diese ODBC-Treiber werden vom DBMS-Hersteller mitgeliefert.
Nicht geregelt ist die Verwendung der SQL-Anweisungen, wie im vorangehenden Abschnitt 3.1.2 bereits diskutiert wurde. Bei diesen handelt es sich meist um einfache Zeichenketten (Strings), die von dem Treiber an das darunterliegende DBMS durchgereicht werden.
Aufgrund dessen, dass ODBC ein reines C/C++ -API ist, lässt es sich für Java-Applikationen nicht nutzen. Daher benötigte man einen neuen Standard, der gleichzeitig die wesentlichen

Nachteile unterbindet. So wurde JDBC entwickelt, das folgende Verbesserungen enthält [Klu98]:

- Plattformunabhängigkeit und damit verbundene Verringerung bzw. Entfallen von Portierungsaufwand auf unterschiedliche Betriebssysteme.
  Die einzige Notwendigkeit ist das Vorhandensein einer *Java Virtual Machine*.

- Unabhängigkeit von der Serverplattform: damit spielt es für den Client keine Rolle, an welchem Ort sich der Server in einem Netzwerk befindet. Die konkrete Lokalisiation wird in Abschnitt 4.5 genauer beschrieben.

- Möglichkeit der Nutzung von Datenbanken in Java-Applets

- Anwendung des objektorientierten Paradigmas,
  ODBC ist nicht objektorientiert, da es vollständig in C entwickelt wurde

- Verzicht auf Zeiger und Adressarithmetik vermindert die Fehlerträchtigkeit der Anwendungen

- Unabhängigkeit vom DBMS: Sofern sich die Anwendung an den allgemeinen Standards für Datenbankmanagementsysteme orientiert, sollten keine größeren Portierungsschwierigkeiten auf andere Datenbanksysteme auftreten. So wird von den meisten DBMS mindestens SQL92-Entrty Level unterstützt. Probleme könnten entstehen, wenn zusätzlich proprietäre Features genutzt werden.

JDBC arbeitet analog zu ODBC. Die Datenbankanwendung öffnet unter der Voraussetzung, dass ein entsprechender JDBC-Treiber vorhanden ist, eine Verbindung zu einem beliebigen DBMS. Der Treiber ist ebenfalls in Java geschrieben und schränkt somit die Protierbarkeit nicht ein. Er wird erst bei Bedarf zur Laufzeit geladen.

Damit ergeben sich folgende vier Schritte für den Ablauf einer jeden JDBC-Anwendung:

1. Laden eines JDBC-Treibers
2. Öffnen einer Verbindung zur Datenbank
3. Senden von SQL-Anweisungen an die Datenbank
4. Empfangen und Auswerten der Ergebnisse

3.1.5 JDBC-Treiber

Der JDBC-Treiber stellt die einheitliche Schnittstelle für den Client zur Verfügung. Wie auch beim ODBC-Treiber ist die interne Realisierung je nach DBMS und Schichtenmodell sehr unterschiedlich, daher muss ein entsprechender Treiber vom Hersteller des DBMS bereitgestellt werden. Das bedeutet wiederum, dass die Anbindung einer Applikation an unterschiedliche Datenbankmanagementsysteme auch erhöhten Verwaltungsaufwand im Programmcode erfordert und für jedes dieser DBMS ein eigener Treiber geladen werden muss.

Man unterscheidet vier Typen von JDBC-Treibern [Ham97]:

Typ 1: Auch bekannt als JDBC-ODBC-Bridge. Hierbei handelt es sich um eine einfache Lösung für beliebige ODBC-Datenbanken, die von JDBC genutzt werden sollen. Es wird allerdings empfohlen, sie nur als Übergang für solche Datenbanken zu betrachten, für die noch keine vollständigen Java-Treiber vom Typ 3 oder 4 vorhanden sind.
Der Treiber nutzt nach unten hin den vorhanden ODBC-Treiber und greift über dessen Funktionalität auf die im DBMS enthaltenen Tabellen zu.
Der große Nachteil bei diesem Verfahren ist die Plattformabhängikeit, da der ODBC-Treiber eine C-Implementierung verwendet.

Typ 2: Dieser Treibertyp ist ebenfalls plattformspezifisch, aber sehr einfach und schnell zu realisieren. Auf einem existierenden in C geschriebenen ODBC-Treiber wird ein JDBC-Aufsatz erzeugt.

Typ 3: Der erste der beiden völlig in Java implementierten Treiber verfügt über die größtmögliche Flexibilität. Er findet seinen Einsatz in Kapitel 3.1.7 erläutertem Dreischichtenmodell. Dabei schickt er seine Anfragen über das Netzwerk an eine sogenannte *Middleware*, welche sich um die weitere Kommunikation mit der Datenbank kümmert und nicht zwingend eine Java-Applikation sein muss. Damit ist der Treiber nicht vom darunterliegenden DBMS abhängig, sondern nur von der jeweiligen Middleware.

Typ 4: Der letzte und ebenfalls hundertprozentig in Java verfasste JDBC-Treibertyp wird im klassischen Zweischichtenmodell, dessen Struktur in Kapitel 3.1.7 beschrieben steht, verwendet. Hierbei wird auf eine Middlerware verzichtet, so dass der Treiber unmittelbar mit dem zugehörigen DBMS kommuniziert. Er ist unter anderem dazu gedacht, Übergangslösungen mit Typ-2-Treibern abzulösen.

Ein Typ-4-Treiber ist eine sehr komplexe Software. Da die meisten verbreiteten DBMS keine Java-Implementierung sind und der Treiber somit nicht unmittelbar auf Klassen des DBMS zugreifen kann, muss er intern über eine eigene herstellerspezifische, dem Anwender verborgene Middleware verfügen.

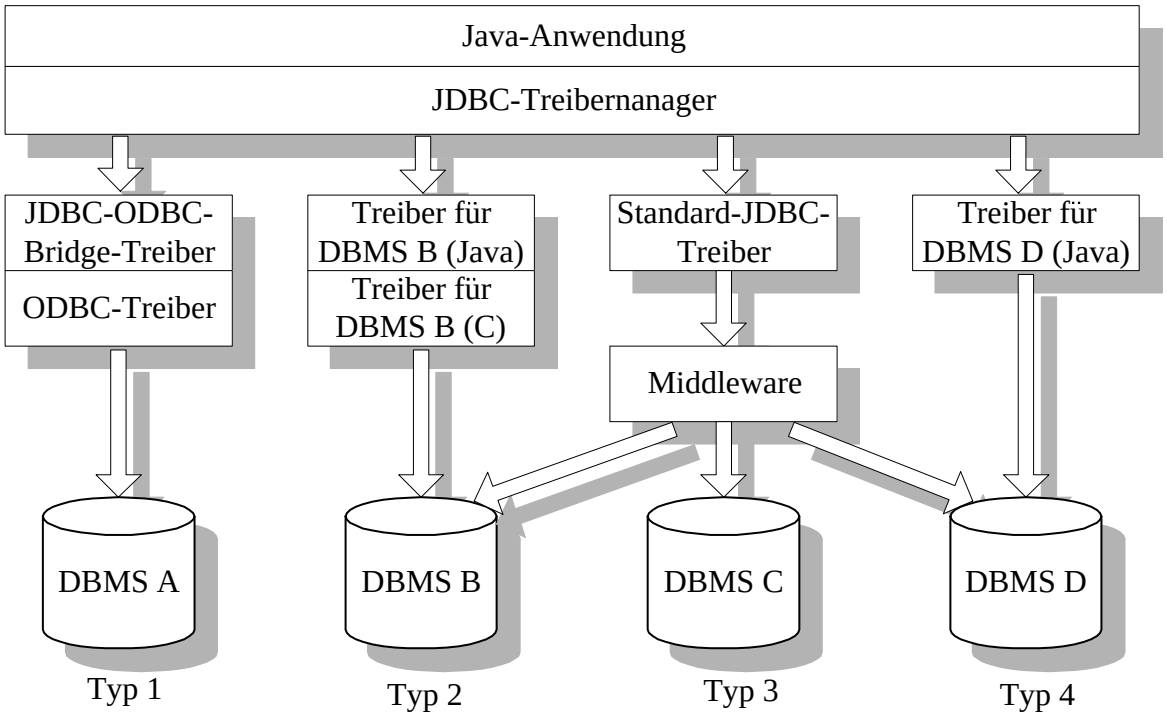

Abbildung 3.1: JDBC-Treibertypen

Zusätzlich gewährleistet ein JDBC-Treiber die Unabhängigkeit von der Serverplattform. Da JDBC grundsätzlich netzwerkfähig ist und der Treiber ebenfalls, spielt es keine Rolle auf welchem Rechner und in welchem Netzwerktyp auf einen Datenbankserver zugegriffen wird. Die Adressierung erfolgt über einen JDBC-URL, der im Aufbau an den Uniform Resource Locator (URL) des Internet-Protokolls angelehnt ist. Damit ist es ebenfalls möglich, Daten von unterschiedlichen Plattformen und verschiedenen Datenbanken zusammenzuführen und gemeinsam zu verarbeiten oder darzustellen. Derartige Informationen-verknüpfende Zugriffe findet man häufig unter der englischen Bezeichnung *Join* wieder [Ham97].

3.1.6 Transaktionen

Häufig kommt es vor, dass Daten nicht nur mit einer einzigen Anweisung in die Datenbank geschrieben werden können, sondern eine ganze Anzahl zusammengehörender Operationen notwendig ist, damit am Ende wieder ein konsistenter Zustand erreicht wird. [Klu98]
Um dies zu gewährleisten, werden derartige Anweisungen in einer Transaktion gruppiert. Man spricht hier von mehrstufigen Transaktionen, da mehrere Datenbankoperationen zusammengefasst sind.
Zuerst informiert man die Datenbank, dass Informationen modifiziert, gespeichert oder gelöscht werden sollen, indem man den Zugriff auf diese Daten beschränkt. Dafür gibt es mehrere Ebenen, die im Anschluss noch erläutert werden. Nach Initiierung dieser Zugriffsbeschränkung werden alle notwendigen Operationen - das können durchaus auch Leseanweisungen sein - ausgeführt. Wenn alle Anweisungen erfolgreich abgeschlossen wurden, müssen sie noch bestätigt, man sagt auch *committed*, werden. Erst jetzt erhalten sie ihre Gültigkeit in der Datenbank. Für den Fall, dass mindestens eine Operation nicht wie gewünscht ausgeführt wird, erfolgt auch keine Bestätigung der anderen. Vielmehr wird ein so genanntes *Rollback* ausgeführt, das alle Änderungen unwirksam macht. Nach dem erfolgreichen Commit-Vorgang wird die Zugriffsbeschränkung der Tabellen wieder aufgehoben. Standardmäßig führt JDBC nach jedem Datenbankzugriff ein Commit aus, so dass es Aufgabe des Entwicklers ist, zusammen gehörende Anweisungen in Transaktionen zu gruppieren.
Noch ein paar Anmerkungen zur zwingend erforderlichen Zugriffsbeschränkung. Bei Ver-

zicht besteht unter Umständen die Gefahr, dass inkonsistente Daten verwendet werden. So könnten während einer nicht vollständig ausgeführten Schreiboperation von einem anderen Anwender teilweise alte und neue Daten gelesen werden. Das wäre fatal.
Man unterscheidet mehrere Stufen bei der Zugriffsbeschränkung, die die englische Bezeichnung *transaction isolation level* tragen. Ihren Einsatz muss der Entwickler je nach Bedarf für seine Anwendung abwägen. Je stärker die Einschränkung, desto länger werden die Zugriffszeiten. Andererseits erhöht sich das Risiko für die Entstehung inkonsistenter Daten, je schwächer die Beschränkung gewählt wird.

Isolationsgrad	**Dirty Reads**	**Nonrepeatable Reads**	**Phantom Reads**
READ UNCOMMITTED	Ja	Ja	Ja
READ COMMITTED	Nein	Ja	Ja
REPEATABLE READ	Nein	Nein	Ja
SERIALIZABLE	Nein	Nein	Nein

Tabelle 3.1: Die Transaktionsisolierungsstufen

Konfliktsituationen bei Transaktionen

- **Inkonsistente Abfrageergebnisse (Dirty Reads)**
  Die Abfrage einer Transaktion liefert das Ergebnis einer anderen noch nicht abgeschlossenen Transaktion. Das kann nicht nur zu einem inkonsistenten Ergebnis führen, sondern auch dazu, dass die auslösende Transaktion zurückgesetzt wird und somit das ermittelte Resultat nicht mehr existiert bzw. aus logischer Sicht niemals existiert hätte.

- **Nicht wiederholbare Abfragen (Nonrepeatable Reads)**
  Bei Wiederholung der selben Abfrage innerhalb einer Transaktion werden abweichende Daten zurückgegeben. Eine andere, parallel ausgeführte und nun abgeschlossene Transaktion hat diese Daten modifiziert. Damit ist das erstmalige Ergebnis nicht mehr nachvollziehbar und nicht wieder herstellbar.

- **Phantomergebnisse (Phantom Reads)**

  Die wiederholte Ausführung einer Abfrage mit identischen Selektionsbedingungen innerhalb einer Transaktion gibt eine unterschiedliche Anzahl an Datensätzen als Ergebnis zurück, denn parallel hat eine andere, inzwischen abgeschlossene Transaktion zusätzliche, dieser Auswahlbedingung entsprechende Datensätze eingefügt. Die neuen Datensätze bezeichnet man als Phantomdatensätze, da sie im ersten Abfrageergebnis nicht enthalten waren.

3.1.7 Kunden, Dienstleister und Schichtenmodelle

Wie bei Netzwerkanwendungen üblich, basieren auch Datenbankapplikationen auf dem Client-/ Server-Prinzip, wobei ein Kunde (Client) eine Dienstleistung in Anspruch nehmen möchte und der Dienstleister (Server) diese erbringt. Ein Client umfasst Programmlogik sowie Benutzerschnittstelle und öffnet bei Bedarf einen Kommunikationskanal zum Datenbankserver um Daten einzutragen, abzurufen, zu modifizieren oder zu löschen.
Ein Datenbankmanagementsystem stellt den Datenbankserver dar, der die Anfragen des Clients bearbeitet und beantwortet.
Angewandt wird dieses Prinzip in den beiden nun folgenden Schichtenmodellen, die bereits vor der Entwicklung von JDBC eingesetzt wurden [Klu98].

Das Zweischichtenmodell

Das Zweischichtenmodell, auch bekannt als *Two Tier Model* stellt einen einfachen, wie der Bezeichnung zu entnehmen ist, zweischichtigen Client-/Server-Datenbankzugang bereit, der im wesentlichen auch nur aus dem Kunden (Schicht 1) und dem Diensterbringer (Schicht 2) besteht. Abbildung 3.2 soll dies verdeutlichen. Aufgrund der recht einfachen Architektur bringt es eine Reihe Nachteile mit sich. So muss der Client für alle DBMS, auf die er zugreifen will, einen Treiber beinhalten. Bei hersteller- und DBMS-unabhängigen Datenbankanwendungen wird dem Entwickler einiges an Know How abverlangt, um diese Anforderungen zu realisieren.

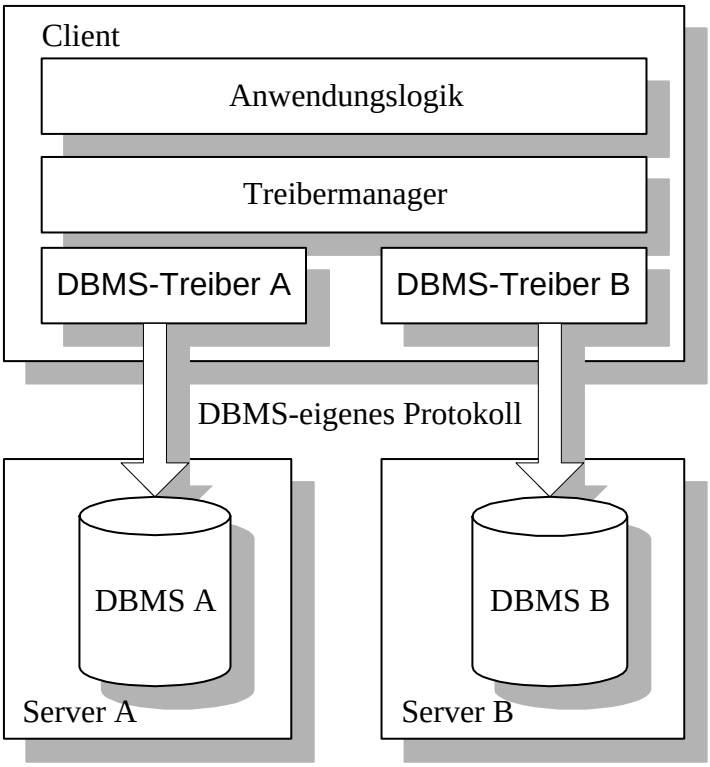

Abbildung 3.2: Schema einer Two-Tier-Applikation

Ein weiterer Nachteil sind sogenannte *Fat Clients*. Da die Zugriffslogik auf die Datenbank vollständig im Client steht, bläht es diesen enorm auf. Das macht sich besonders bemerkbar, wenn mehrere DBMS unterstützt werden sollen.
Einige DBMS verlangen, dass Client und Server auf dem selben Rechner laufen müssen oder benötigen eine zusätzliche Software, um eine Verteilung auf das Netz umzusetzen.
Alle diese Nachteile werden im Dreischichtenmodell vermieden.

Das Dreischichtenmodell

Bei einem Dreischichtenmodel, auf Englisch *Three Tier Model*, greift der Client nicht direkt auf das DBMS zu, sondern überlässt die Abbildung auf die einzelnen Systeme einer sogenannten Middleware, an die er auch seine Anfragen stellt. Die gewünschte Datenbank wird über den bereits angesprochenen JDBC-URL spezifiziert. Die Middleware lädt den benötigten DBMS-Treiber, der die Verbindung zur Datenbank herstellt. Nun kann der Client über die-

se Verbindung SQL-Anweisungen zum DBMS senden und die Ergebnistabellen in Empfang nehmen. Damit muss dem Client das zugrunde liegende Datenbanksystem nicht bekannt sein. Neben der Anwendungslogik enthält er lediglich ein Kommunikationsprotokoll für den Datenaustausch mit der Middleware.

Abgesehen von den Standardprotokollen wie JDBC und ODBC gibt es noch die Möglichkeit,

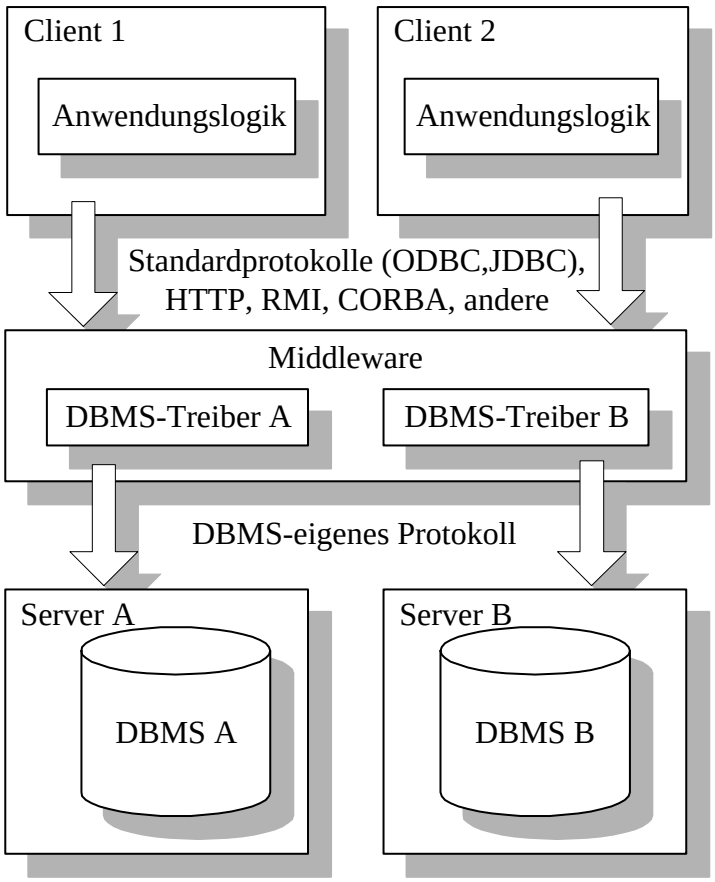

Abbildung 3.3: Schema einer Three-Tier-Applikation

andere Protokolle bzw. Paradigmen zu nutzen, beispielsweise RMI oder CORBA. Sie bieten eine breite Palette an Diensten, sind allerdings mit einem erheblichen Mehraufwand an Implementierung verbunden.

Alle drei Schichten sind unabhängig voneinander und können auch jeweils auf unterschiedlichen Rechnern im Netzwerk laufen. Ist das DBMS nicht netzwerkfähig, müssen Middleware und Managementsystem auf der selben Maschine eingesetzt werden. Abbildung 3.3 zeigt die typische Architektur eines Dreischichtenmodells einschließlich der Kommunikationsprotokol-

le.

Sicherlich eröffnet sich nun die Frage, warum nicht in allen Datenbankprojekten ein solches Mehrschichtenmodell zum Einsatz kommt, wenn es doch so enorme Vorteile bietet. Die Antwort ist ebenso simpel wie einleuchtend. Die Middleware ist keine *Open Source* Software und damit teuer vom jeweiligen Softwareproduzenten zu erwerben.

Da Res Medicinae kostenfrei zur Verfügung gestellt werden soll, kann man von zukünftigen Anwendern keine finanzielle Beteiligung an einer Lizenz für die Middleware erwarten. Somit verwendet man hier ebenfalls das Zweischichtenmodell. Eine Anbindung von CORBA oder RMI kann Alternativen bieten und wurde beim Entwurf des Systems berücksichtigt, so dass eine spätere Implementierung möglich ist.

3.1.8 PostgreSQL

Als Datenbankmanagementsystem wurde für die Realisierung von *Res Medicinae* das *Open Source* -Projekt PostgreSQL [Pos02b] ausgewählt. Hierbei handelt es sich um ein objektrelationales DBMS, dessen Ursprung man bis ins Jahr 1977 zurückverfolgen kann. Damals erfolgte an der Universität Berkeley die Entwicklung des relationalen DBMS Ingres, das 1986 die Basis für das heutige PostgreSQL lieferte.

Die für diese Diplomarbeit zugrunde gelegte aktuelle Version 7.2.1 unterstützt nach Aussage des Herstellers die Standards SQL92 und SQL99. Das umfasst unter anderem die *referenzielle Integrität*, eine *Transaktionssteuerung* und eine *Cursorverarbeitung*.

Für den Mehrbenutzerbetrieb verwendet es das so genannte MVCC-Verfahren [Pos02a]. Die Abkürzung steht für *Multiversion Concurrency Control*. Die meisten Datenbanksysteme benutzen Schlösser (*locks*) für die Wahrung der Konsistenz bei konkurrenten Zugriffen. PostgreSQL verwendet dafür ein Multiversions-Modell, d.h. jede Transaktion sieht nur einen Auszug der Daten, wie sie kurze Zeit zuvor vorlagen, unabhängig vom aktuellen Zustand der unterliegenden Daten. Dieser Mechanismus schützt die Transaktionen davor, inkonsistente Daten zu sehen, die von anderen parallel ausgeführten Transaktionen eingefügt worden sind. Im Unterschied zur Schloß-Methode blockieren Lese- niemals Schreiboperationen und umgekehrt blockieren Schreib- auch keine Leseoperationen.

Von den Transaktions-Isolations-Stufen sind in PostgreSQL *Read Commited* und *Serializable* implementiert, wobei erstere als Voreinstellung verwendet wird.

Die OID

Unabhängig von den Primärschlüsseln der Tabellen erhält jeder Datensatz beim Speichern vom DBMS eine zusätzliche, systemweit eindeutige Nummer, eine Object Identity (OID). Bei der Initialisierung von PostgreSQL wird ein Zählmechanismus eingerichtet, der diese OIDs generiert.

3.2 Extensible Markup Language - XML

XML wurde von einer XML-Arbeitsgruppe entwickelt, die man 1996 unter der Schirmherrschaft des World Wide Web Consortium (W3C) gründete. Die erste Version wurde 1998 als Standard vom W3C [W3C02a] beschlossen.
XML bietet unter anderem die Möglichkeit, Daten zu speichern und im Internet zugänglich zu machen.

```
<users>
	<user>
		<name>Bohl</name>
		<firstName>Jens</firstName>
		<group>User</group>
	</user>
	<user>
		<name>Kunze</name>
		<firstName>Torsten</firstName>
		<group>User</group>
	</user>
</users>
```

Abbildung 3.4: Ein typisches XML-Dokument

Die Idee liegt in der Entwicklung einer Syntax, die es erlaubt, besser strukturierte Informationen im World Wide Web bereitzustellen. Zudem gestattet es den Austausch von Daten zwischen verschiedenen Datenbanken und Anwendungen. Genaugenommen wird der Austausch von Informationen zwischen Datenbanken einer der wesentlichen Einsatzbereiche von XML sein.

Wer heute HTML einsetzt, kümmert sich in erster Linie darum, wie die Informationen auf den Bildschirmen der Anwender erscheinen. XML ist dagegen stärker darauf ausgerichtet, Informationen so aufzubereiten, dass sie leicht weiterverarbeitet werden können. Es geht nicht darum, wie die Informationen auf dem Monitor dargestellt werden, sondern viel mehr, wie sie strukturiert sind. Die Vision von XML für die Zukunft ist, die Sprache als Basis für jede Art von Daten zu etablieren. Aus diesen Daten können dann fast beliebig, andere Dokumenttypen erzeugt werden. Die anfallende Doppelarbeit bei Erfassung und Konvertierung fällt weg. Deutlich wird dabei auch, dass nicht mehr einzig und allein die Verarbeitung von Dokumenten im Vordergrund steht, sondern selbst Datenbanken wie beispielsweise Artikelstammdaten mit Hilfe von XML verarbeitet werden können. Abbildung 3.4 zeigt ein typisches XML-Dokument. Kennzeichnend sind jeweils das einführende und abschließende Tag vor und hinter den Daten. Damit werden sie in einer baumartigen Hierarchie abgelegt. Diese Eigenschaft nutzen XML-Parser, um die Daten wieder aus den Dokumenten zu lesen. Zusätzlich überprüfen sie die syntaktische Korrektheit und somit die Wohlgeformtheit, beispielsweise ob für jedes öffnende Tag auch ein entsprechendes Ende-Tag existiert. Weiterhin können sie die Auszeichnung der Elemente von ihrem Inhalt trennen und Operationen auf diesen Elementen ausführen.

Es gibt zwei Hauptarten von Parsern. Zum einen baumbasierte und zum anderen ereignisbasierte. Ein baumbasierter Parser übersetzt ein XML-Dokument in eine interne Baumstruktur und erlaubt dann einer Applikation in diesem Baum zu navigieren. Die Document Object Model (DOM) -Arbeitsgruppe im World Wide Web Consortium (W3C) entwickelte ein Standard-Baum-basiertes API für XML- und HTML-Dokumente [W3C02b].

Auf der anderen Seite schickt ein ereignisbasiertes API Ereignisse, wie beispielsweise den Beginn und das Ende eines Elements, mittels *Callback-Methoden* direkt an eine Anwendung und erstellt für gewöhnlich intern keine Baumstruktur. Die Applikation implementiert einen

Document-Handler, um die verschiedenen Ereignisse zu verwalten, ähnlich dem Verwalten der Ereignisse eines *Graphical User Interfaces* (GUI). Im Speziellen reagiert ein solcher Parser auf das Auftreten bestimmter Typen von Tags. Das sind Dokumentbeginn, Dokumentende, Zeichenketten, Elementbeginn, Elementende. Das Simple API for XML (SAX) [Meg02] enthält einen ereignisbasierten Parser.
An dieser Stelle wird auf eine weitere Erläuterung von DOM und SAX verzichtet und auf die Studienjahresarbeit [Kun02] des Diplomanden verwiesen, in der diese Themen ausführlich behandelt werden.

3.3 Konkretisierte Aufgabenstellung

Unter Zusammmenfassung des in den letzten beiden Kapiteln beschriebenen *State of the Art* soll hier nun noch einmal die konkrete Aufgabenstellung für den eigenständig zu entwickelnden Anteil auf einen Blick dargestellt werden.
Basierend auf bestehenden Softwaremustern sollen Persistenz- und Kommunikationsparadigmen in einer gemeinsamen, transparenten Schicht der Applikation vereinigt werden. Bisherige Ansätze in diesen Mustern beschreiben lediglich die Realisierung für jeweils einen Persistenzmechanismus bzw. ein Kommunikationsparadigma. Es sind Untersuchungen anzustellen, wie die verschiedenen Persistenzmechanismen sinnvoll für die Anwendung umgesetzt und ineinander überführt werden können, sowie auf welche Art verschiedene Kommunikationsparadigmen parallel in einer Applikation zum Einsatz kommen können.
Die in Abschnitt 2.2 erörterten Softwaremuster sollen eingesetzt sowie entsprechend angepasst werden und helfen die Struktur möglichst flexibel, erweiterbar aber auch robust zu gestalten. Besonderes Augenmerk gilt dem Muster *Data Mapper*, da es einen guten Ansatz liefert, die direkte Abhängigkeit zwischen Domain-Modell und Persistenzschicht zu vermeiden. Mit *Data Transfer Object* und *Remote Facade* lassen sich Kommunikationsprozesse optimieren. Sie ermöglichen es, unnötig viele Datentransfers und Fernprozeduraufrufe zu unterbinden. Das Muster *Model View Controller* dient bei *Res Medicinae* als prinzipielles Architekturmuster zur Strukturierung der Präsentationsschicht sämtlicher Module und

kommt daher auch bei dem zu entwickelnden Prototyp *ReForm* zum Einsatz. *ReForm* dient dem Testen der zu entwerfenden Modelle, indem eine Anwendung generiert wird, die das Drucken medizinischer Formulare unterstützt. Mit möglichst geringem Aufwand müssen die Patientendaten aus dem jeweiligen Persistenzmedium geladen und in medizinische Formularvordrucke ausgegeben werden können.

Ein in Grundzügen bestehendes Domain-Modell muss an die permanent wachsenden Anforderungen angepasst und erweitert werden. Hierfür wird das gleichnamige Strukturmuster *Domain Model* verwendet. Die Speicherung der enthaltenen Geschäftsdaten soll zum einen per XML in einer zu modellierenden Dateistruktur auf der lokalen Festplatte jedes einzelnen Anwenders und zum anderen in einer zentralen, über ein Netzwerk angebundenen Datenbank erfolgen. Für diese ist eine Tabellenstruktur zu entwerfen, welche den aktuellen Anforderungen an eine moderne Datenbank gerecht wird. Als Datenbankmanagementsystem ist PostgreSQL zu verwenden, das für JDBC einen Typ-4-Treiber zur Verfügung stellt. Die Anfragen sind in Embedded SQL zu formulieren und damit unmittelbar in den Java-Quellcode einzubinden.

Kapitel 4

Die Gesamtarchitektur

Während in den beiden einführenden Kapiteln die verwendeten Technologien separat beschrieben wurden, soll nun die Anwendung und Verbindung zu einer Gesamtlösung für die transparente Mappingschicht und deren Anbindung und Interoperation mit den benachbarten Schichten diskutiert werden. Das Hauptaugenmerk liegt hierbei auf der Umsetzung verschiedener Muster, um eine möglichst flexible und zugleich robuste Struktur zu erhalten. Abbildung 7.1 im Anhang A zeigt das gesamte Klassenmodell auf einen Blick. Die einzelnen Teile werden im Anschluss genauer betrachtet.

Da jedes Anwendungsmodul von *Res Medicinae* als autonomer Prozess läuft, ist eine Interprozesskommunikation notwendig, um Daten zwischen mehreren Modulen auszutauschen. Beispielsweise muss *Record* seine Patientendaten an das *ReForm* senden, wenn ein Rezept ausgestellt und gedruckt werden soll.

Während die Hauptaufgabe des prototypisch umgesetzten Moduls *ReForm* die funktionale Steuerung von medizinischen Formularen ist, realisiert *Record* [Boh03] die Verwaltung und Darstellung der Patientendaten.

Für diese und ähnliche Aktivitäten wurde untersucht, auf welche Art sich ein Kommunikationsparadigma in die Applikationen einarbeiten lässt. Aufgrund der gewünschten Flexibilität und noch nicht vollständig abschätzbaren Komplexität der finalen Version des Programms, das als wesentliches Merkmal eine sehr gute Erweiterbarkeit aufweisen muss, wurde die Betrachtung auf eine verteilte Kommunikation, d.h. einen Datenaustausch zwischen mehreren Rechnern erweitert.

4.1 Das Framework

Im Rahmen des Projektes *Res Medicinae* entstand ein Framework, auf dem sämtliche Applikationen und Komponenten, die im Rahmen dieser Diplomarbeit entwickelt wurden, aufbauen. Es kapselt das objektorientierte Klassenmodell von Java und erweitert es zu einer hierarchischen Struktur. Ähnlich der Klasse *Object* des Java Development Kits von SUN Microsystems, existiert eine Basisklasse *Item*, von der alle anderen Klassen erben. Sie bildet die Grundlage für die Hierarchie. Jedes Attribut einer Klasse wird als Kindobjekt betrachtet, das ebenfalls über den Supertyp *Item* verfügt und deshalb wiederum eigene Kindobjekte enthalten kann. Diese Technik ist typisch für das hier angewendete Kompositum-Muster, wie es in Abschnitt 2.2.1 erläutert wurde.
Warum der Umstand einer Hierarchie und Kapselung aller bestehenden Klassen? Bei genauer Betrachtung wird ersichtlich, dass es sich hierbei ganz und gar nicht um einen Umstand handelt. Vielmehr ist die Unterteilung des Frameworks an das Vorbild "Natur" angelehnt. Abstrakt betrachtet bildet sie ein riesiges System, das bis zum Atom und kleiner verfeinert werden kann. Die Klassenstruktur des Frameworks gliedert sich ähnlich. Ausgehend von einem System, der Software-Applikation, unterteilt sich die Struktur in Block, Region, Komponente, Teil und Kette, wobei man Systeme noch zu Familien gruppieren kann. Die Klassen einer Ebene dürfen die Klassen einer niederen Schicht benutzen, nicht jedoch umgekehrt. Damit sind unidirektionale Abhängigkeiten und eine kreisfreie Vererbung sichergestellt.
Eine derartige Struktur und Darstellung komplexer Zusammenhänge in Hierarchieebenen wird auch als *Ontologie* bezeichnet [Boh03].

4.2 Die Applikation

Die zentrale Einheit eines jeden *Res Medicinae*-Moduls ist sein *Controller*, abgeleitet aus dem Model View Controller -Muster (Abschnitt 2.2.1), welches hier zum Einsatz kommt. Im Controller laufen alle Ereignisse zusammen, werden ausgewertet und entsprechende Operationen ausgeführt. Im Gegensatz zum klassischen Model View Controller wird bei *Res Medicinae*

auf eine Anwendung des Beobachter-Musters verzichtet. Die Aktualisierungsoperationen für die Views werden vom Controller gesteuert, wodurch der Implementierungsaufwand für eine zusätzliche Beobachter-Komponente eingespart werden konnte.
Das Framework sieht eine Klasse *Controller* vor, deren Funktionalität nach der Vererbung beliebig erweitert werden kann. *AdvancedBasicApplication* ist eine ihrer Subklassen. Von dieser werden sämtliche Basisoperationen, die in Verbindung mit der Mapping-Schicht stehen, initiiert und gesteuert. Im speziellen betrifft das Methoden für das Laden, Speichern, Anlegen, Löschen, Modifizieren und Importieren von Patienten-Karteien. Zusätzlich wird hier der gewünschte Persistenzmechanismus bestimmt, d.h. der Anwender kann zur Laufzeit entscheiden, ob er die Daten in einem XML-Format auf seiner lokalen Festplatte oder zentral in der Datenbank ablegen möchte.
Die Identifikation des Kommunikationsparadigma erfolgt ebenfalls durch den Anwender zur Laufzeit. Es kann zwischen RMI, CORBA und anderen gewählt werden, deren vollständige Implementierung jedoch nicht Teil der Diplomarbeit war und daher auch nur vorbereitet, das heißt beim Entwurf der Architektur berücksichtigt wurde.
Während der Initialisierung der *AdvancedBasicApplication* erfolgt unter anderem die Instanziierung einer Server- und einer Client-Klasse, standardmäßig RMI. Das Server-Objekt meldet seine verfügbaren Dienste am zugehörigen Name Service an, so dass Clients nun über diese Dienste Daten anfordern können. Eine ausführliche Beschreibung erfolgt im Abschnitt 4.6.
Jedes Modul hält jeweils eine Referenz auf ein Client- und auf ein Serverobjekt. Das Auslagern von Server und Client in separate Klassen macht durchaus Sinn. Zum einen wird die Methodik der Klassen aufgrund einer klaren Trennung weniger unübersichtlich und zum anderen sind die zusammengehörigen Algorithmen für einen Kommunikationsdienst in jeweils einer Klasse gruppiert. Daher ist es recht einfach, Module um weitere Dienste zu erweitern oder andere zu entfernen. Man findet diese Technik auch beim Model View Controller wieder. Sie trägt hier die Bezeichnung Strategie-Muster.
Mit der Realisierung der Klasse *AdvancedBasicApplication* verfügt jedes von ihr erbende Modul, wie beispielsweise *ReForm* oder *Record* über eine grundlegende Funktionalität, die bei Bedarf genutzt werden kann.

4.3 Die Domain

Die Domain-Architektur von *Res Medicinae* beinhaltet den wesentlichen Eigenanteil des entworfenen und umgesetzten Modells für die Diplomarbeit. Sie gliedert sich in drei Ebenen. An oberster Position befindet sich das Domain-Modell. Es verwaltet sämtliche Daten der Geschäftslogik. Die unterste Ebene, die Persistenzschicht, dient dem langzeitigen Speichern dieser Daten und sorgt im Falle einer Datenbank für die zentrale Zugänglichkeit. Um eine direkte Abhängigkeit zwischen Domain-Modell und Persistenzmechanismen zu vermeiden, wurde eine für den Anwender transparente Zwischenschicht eingefügt. Sie ist angelehnt an das in Abschnitt 2.2.4 beschriebene Muster Data Mapper. Durch eine Abbildung der Domain-Daten auf die entsprechende Speicherstruktur, muss im Falle einer Modifikation von Domain-Modell oder Persistenzschicht, die jeweils andere Ebene nicht angepasst werden. Eine Änderung ist lediglich in der Mappingschicht nötig.

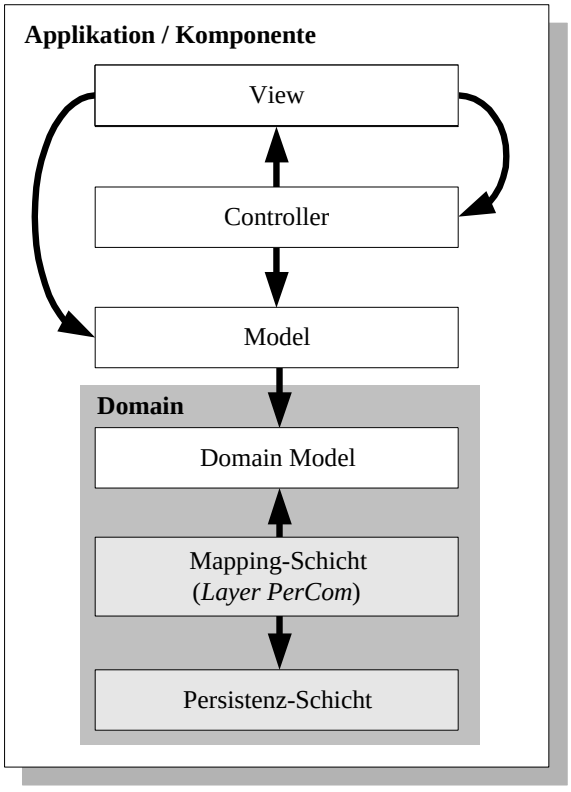

Abbildung 4.1: Die logische Architektur von Res Medicinae

Besonders in verteilten Anwendungen ist das sinnvoll, da Modifikationen der Datenbank die Notwendigkeit einer Anpassung sämtlicher Clients nach sich ziehen würde. Wohingegen die Änderung der Mappingschicht, die in diesem Fall von den Clients extrahiert wurde und sich auf einem anderen Rechner befindet, einen wesentlich kleineren Aufwand bedeutet. Eine derart verteilte Realisierung ist bis zum gegenwärtigen Zeitpunkt in *Res Medicinae* praktisch noch nicht umgesetzt, aber durch die Verwendung des Data Mapper -Musters in der aktuellen Architektur berücksichtigt worden. Da also die Mappingschicht als wesentliche Neuerung gegenüber anderen Ansätzen die Möglichkeit bietet, nicht nur einen, sondern mehrere Pesistenzmechanismen zu verwenden und zusätzlich ein Modell für die Anbindung unterschiedlicher Kommunikationsparadigmen vorschlägt, wurde projektintern für diese Schicht eine neue eigenständige Bezeichnung geprägt: *Layer PerCom*.
Abbildung 4.1 verdeutlicht, wie sich die Domain-Architektur in die Applikation eingliedert. Für die Umsetzung des Model View Controller -Musters werden die Daten des Domain-Modells in ein an die View angepasstes Model transferiert. Bei Änderung der Daten im Modell wird die View vom Controller aufgefordert, sich zu aktualisieren. Das ist immer dann der Fall, wenn von der Mappingschicht Daten aus der darunter liegenden Datenquelle gelesen und in das Domain-Modell geschrieben werden.
Zusätzlich muss die View Kenntnis von dem Controller besitzen, denn dieser fungiert auch als Listener, d.h. er "hört", ob in der View Ereignisse ausgelöst wurden und antwortet mit einem entsprechenden Methodenaufruf.
Die in der obigen Abbildung grau hinterlegte Domain-Architektur entspricht dem im Rahmen dieser Diplomarbeit entwickelten Modell zur persistenten Sicherung von Daten. Die beiden hellgrau gekennzeichneten Schichten *Layer PerCom* und Persistenzschicht wurden eigenständig entworfen und entwickelt. Das Domain-Modell unterlag mit wachsender Komplexität der Anforderungen auch für parallel entwickelte Module zum jeweiligen Zeitpunkt entsprechenden Erweiterungen.

Das Kernstück der Mappingschicht bilden die Assembler. Es wird zwischen zwei Haupttypen unterschieden. Zum einen existieren Assembler, die eine Konvertierung der Informationen des Domain-Modells für den jeweiligen Persistenz-Mechanismus vornehmen. Im Speziellen erstellt der *XMLAssembler* ein XML-Dokument, das als solches in einer Datei

gespeichert wird. Der *ERAssembler* hingegen verwaltet die Daten unter Verwendung des SQL-Packages in einer Datenbank. Der Prefix *ER* steht für *Entity Relationship*. Das eigens für *Res Medicinae* entwickelte SQL-Package umfasst sämtliche Statements, die für Operationen auf der Datenbank notwendig sind. Jedes Statement wird dabei jeweils in einer eigenen Klasse gekapselt, so dass die Applikation sehr simpel um neue SQL-Anweisungen erweitert werden kann.
Der andere Assembler-Typ ist für die entsprechende Abbildung der Domain-Daten auf das vom Anwender aktuell selektierte Kommunikationsparadigma verantwortlich. Es liegen noch keine implementierten Algorithmen für diese Aktionen vor. Berücksichtigt wurden die Architektur und Kommunikationseigenschaften. Weiterhin wurde versucht, die Realisierung so allgemein wie möglich zu halten, damit eine Erweiterung um andere Paradigmen ohne großen Aufwand vollzogen werden kann. Vorgesehen sind bisher Java-RMI und CORBA. JMS ist ebenfalls möglich.

4.4 Das Domain-Modell

Das Domain-Modell von *Res Medicinae* ist zum bisherigen Zeitpunkt noch recht übersichtlich. Die zentrale Klasse *HealthRecord* erbt die Eigenschaften der Frameworkklasse *DomainObject*. Ein *HealthRecord* enthält sämtliche allgemeinen Patientendaten, beispielsweise Name, Adresse, Telefonnummer und Versicherung. Zusammengesetzte Datentypen, wie Adresse, sind als eigene Klassen modelliert worden. Weiterhin kann ein *HealthRecord* beliebig viele Probleme enthalten. Ein Problem stellt eine Krankheit dar. Es setzt sich aus Episoden zusammen, welche wiederum aus einer Anzahl partieller Kontakte bestehen. Während ein partieller Kontakt genau einen Arzttermin zu einer bestimmten Krankheit darstellt, verkörpert eine Episode einen Teilabschnitt zusammengehörender partieller Kontakte. Dazu werden *Subjective*, *Objective*, *Assessment* und *Plan*, also Anamnese, Befund, Bewertung und Vorgehen festgehalten. Da eine Krankheit über einen längeren Zeitraum wiederholt auftreten kann, besteht die Möglichkeit, dass ein Problem eine Vielzahl an Episoden enthält, allerdings gibt es zu jedem partiellen Kontakt jeweils nur genau eine

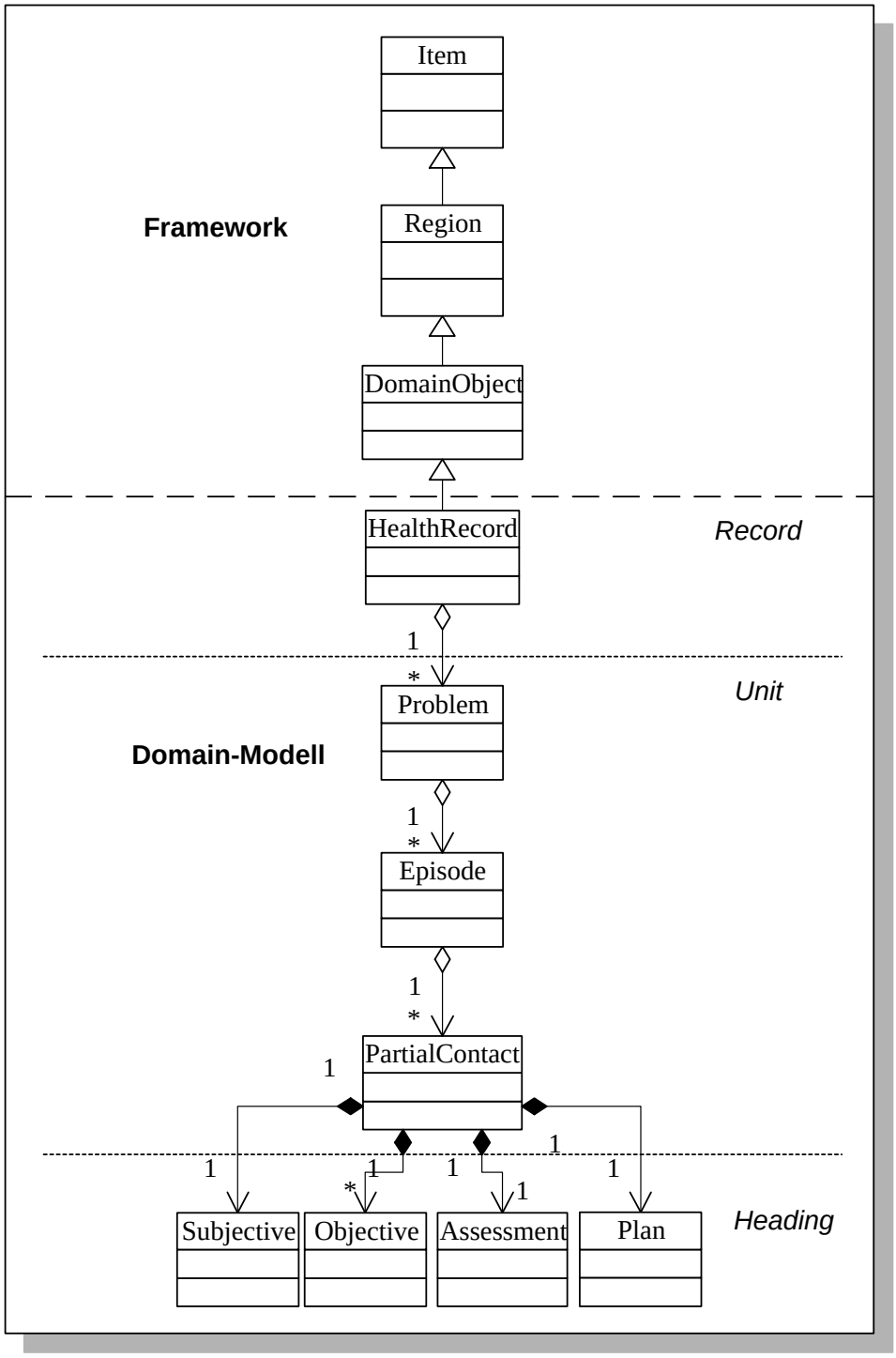

Abbildung 4.2: Das Domain-Modell von Res Medicinae

Anamnese, eine Bewertung und ein Vorgehen. Befunde können mehrere existieren.
Wie in Abbildung 4.2 ersichtlich ist, fügt sich das Domain-Modell in die *Ontologie* des Frameworks als *Region* ein, die wie alle anderen Klassen an oberster Stufe von *Item* erbt. Eine zusätzliche Untergliederung in *Record*, *Unit*, *Heading* und *Description* bildet für das Domain-Modell eine eigene *Ontologie*. *Description* kennzeichnet die Primitive der Domain-Struktur, beispielsweise Blutdruck-, Pulsmesswerte oder Name. Sie können in jede Klasse der übergeordneten Ebenen eingebunden werden. *Heading*s bilden die zweit niedrigste Stufe der Einteilung. Wenig komplexe Verknüpfungen von *Description*s und *Heading*s zu einer Einheit verschiedener Sachgebiete werden von *Unit*s gebildet. Letztendlich fasst ein *Record* alle Daten zu einem Ganzen zusammen. Aus Übersichtlichkeitsgründen wurde auf die Darstellung von Beispielklassen zu *Description* in der Abbildung verzichtet. Die Komplexität der Schichten nimmt von unten nach oben zu. Damit zeichnet sich diese *Ontologie* ebenfalls durch einen hierarchischen Aufbau aus.

4.5 Der Data Mapper als Persistenzsteuerung

Die Implementierung der grundlegenden Methodik zum Verwalten der Domain-Modell-Daten stellte einen der Schwerpunkte des praktischen Teils der Diplomarbeit dar.
Hierfür wurde vorerst eine XML-Realisierung verwendet, dessen Funktionalität die Klasse *XMLAssembler* beinhaltet. Alle Daten werden in einem XML-Dokument lokal auf der Festplatte gespeichert, wie es in Abschnitt 4.5.1 beschrieben wurde.
Der *ERAssembler* hingegen realisiert die Verwaltung sämtlicher Daten in einer Datenbank. Über die Anweisungen

```
Class.forName(''org.postgresql.Driver'');
Connection connection =
DriverManager.getConnection(''jdbc:postgresql://zone3/resmedicinae'',
''zonie'', '''');
```

verbindet sich der Assembler mit dem zugrunde liegenden DBMS PostgreSQL. Der Datenbanktreiber wird zur Laufzeit mit dem ersten der beiden Befehl geladen. Auffällig ist

hier, dass keine Instanziierung der Klasse *Driver* erfolgt. Es genügt eine Referenz im Arbeitsspeicher. Damit steht eine PostgreSQL-spezifische Implementierung der angebotenen Operationen zur Verfügung.
Der erste Parameter von *getConnection()* beinhaltet unter anderem einen Verweis auf den Datenbanknamen "resmedicinae". Die Addresse des Rechners "zone3", das verwendete Protokoll "jdbc" sowie Subprotokoll bzw. Identifikator des DBMS "postgresql" sind ebenfalls enthalten, so dass die Verbindung zur Datenbank eindeutig spezifiziert ist. Mit den anderen beiden Parametern werden Nutzername und Passwort übergeben. In diesem Fall ist kein Passwort erforderlich.

4.5.1 Das XML-Datenmodell

Wie eingangs erwähnt, handelt es sich bei *Res Medicinae* um ein relativ junges Projekt. So stand zu Beginn dieser Diplomarbeit noch keine lauffähige Datenbankanbindung zur Verfügung. Die Datensätze wurden aus diesem Grund vorerst ausschließlich in einem XML-Format hinterlegt. Es zeigte sich, dass das entwickelte Verfahren für diese lokale Speicherung der Daten auch im späteren Umgang als sinnvoll erachtet und somit ebenfalls Bestandteil der Mappingschicht wurde.
In einer Indexdatei *EHRIndex* (siehe Abbildung 4.4) werden zunächst nur die wichtigsten Informationen aller Patienten gehalten. Das sind Patientenidentifikationsnummer, Patientenname und -vorname. Erstere wird im Folgenden unter der Abküzung Patienten-ID häufige Verwendung finden. Diese Patienten-ID dient vor allem zur eindeutigen Kennzeichnung und somit Identifikation der einzelnen Patientendatensätze und bietet sich somit auch zur Verwendung als Dateiname für die einzelnen Datensätze an. Jeder Patient mit seinen sämtlichen verfügbaren Informationen wird in einer eigenen durch seine ID festgelegten Datei hinterlegt und kann aufgrund der Kennzeichnung in der Indexdatei jederzeit lokalisiert werden.
Die oben beschriebene Architektur bietet den Vorteil, dass nicht sämtliche vorhandene Datensätze beim Starten der Applikation eingeladen werden müssen, was mit wachsender Anzahl einen immensen Speicherbedarf mit sich brächte, sondern dass zu jedem Patienten die Daten erst bei Bedarf geladen und dann allerdings im Arbeitsspeicher bzw. in einer

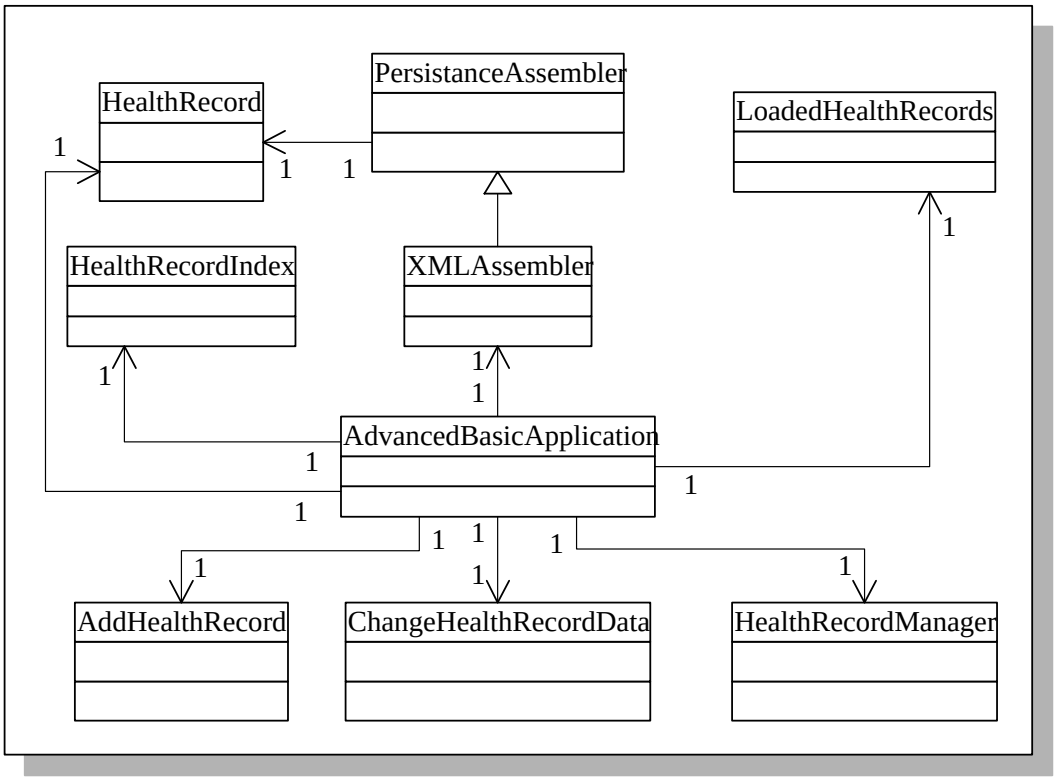

Abbildung 4.3: Die Klassenstruktur der XML-Komponente von Layer PerCom

Instanz der Klasse *LoadedHealthRecords* gehalten werden, um einen schnelleren Zugriff zu ermöglichen und wiederholtes Lesen von der Festplatte zu vermeiden. Dies bedeutet allerdings auch, dass die Daten vor Beendigung des Programmes durch ein vom Benutzer ausgelöstes Speichern gesichert werden müssen. Damit dieser Vorgang nicht versehentlich übergangen wird, öffnet sich im Falle einer ungespeicherten Modifikation der Daten ein separater Hinweisdialog.

Wie dem Klassendiagramm zu entnehmen ist, bildet die Klassse *AdvancedBasicApplication* die zentrale Instanz. Sie empfängt alle Ereignisse aus den Dialogen *HealthRecordManager*, *AddHealthRecord*, *ChangeHealthRecordData* und reagiert mit dem Aufruf entsprechender Methoden des *XMLAssemblers*. Dieser erzeugt aus den ihm übergebenen Daten ein XML-Dokument, dass er anschließend in einer Datei abspeichert, oder er lädt die Daten aus einer Datei. Dafür erzeugt er unter Zuhilfenahme des XML-Parsers XERCES [Kun02] ein XML-Dokument, von dem er die Informationen extrahiert und an die *AdvancedBasicApplication* als neuen *HealthRecord* zurückgibt. Die Klasse *HealthRecordIndex* dient zum Auffinden der

gesuchten Datei. Sie enthält sämtliche Daten der oben angesprochenen Indexdatei und damit auch die als Dateinamen dienenden Patienten-IDs.
Unter dem Eintrag *Cave* (lateinisch für Achtung) verbergen sich besondere Merkmale der Patienten, die bei Behandlungen zu berücksichtigen sind, wie beispielsweise Allergien.
Ein wesentlicher Vorteil der gewählten Architektur ist die ähnliche Strukturierung zu den Tabellen der Datenbankanbindung. So sind unter anderem die IDs der Datensätze bzw. die Dateinamen der XML-Files direkt als Primärschlüssel (zu Englisch: Primary Key) der Tabelle *HealthRecords* verwendbar. Mit der konkreten Generierung der Primary Keys beschäftigt sich ein Teil des nächsten Abschnittes.

4.5.2 Das Datenbankmodell

Die Anwendung basiert auf einem relationalen Datenbankentwurf. So wie die noch recht simple Gestalt des Domain-Modells eine recht einfache Speicherstruktur für XML-Dateien ermöglicht, spiegelt sie sich ebenfalls in Anzahl und Aufbau der Tabellen der zu generierenden Datenbankanbindung wider. Man kann sagen, dass die Klassenstruktur des Domain-Modells direkt in die Architektur der Datenbank übertragen wurde. Tatsächlich gibt es für die Abbildung von Klassen auf Tabellen drei empfohlene Ansätze:

1. Eine Tabelle für jede Hierarchie
2. Eine Tabelle für jede konkrete Klasse
3. Eine Tabelle für jede Klasse (auch abstrakte Klassen)

Für die umgesetzte Datenbankanbindung wurde die zweite Variante gewählt, d.h. für jede Klasse des Domain-Modells, die zur Datenhaltung verwendet wird, existiert eine Tabelle, abgesehen von Subjective, Objective, Assessment und Plan sowie den Descriptions, die lediglich als Spalten in der Tabelle *PartialContacts* benötigt werden. Sie bestehen jeweils nur aus einer Zeichenkette.
Während in Abbildung 4.4 die Primärschlüssel fett, unterstrichen und die Fremdschlüssel fett, ohne Unterstreichung dargestellt sind, werden im anschließenden Entity Relationship

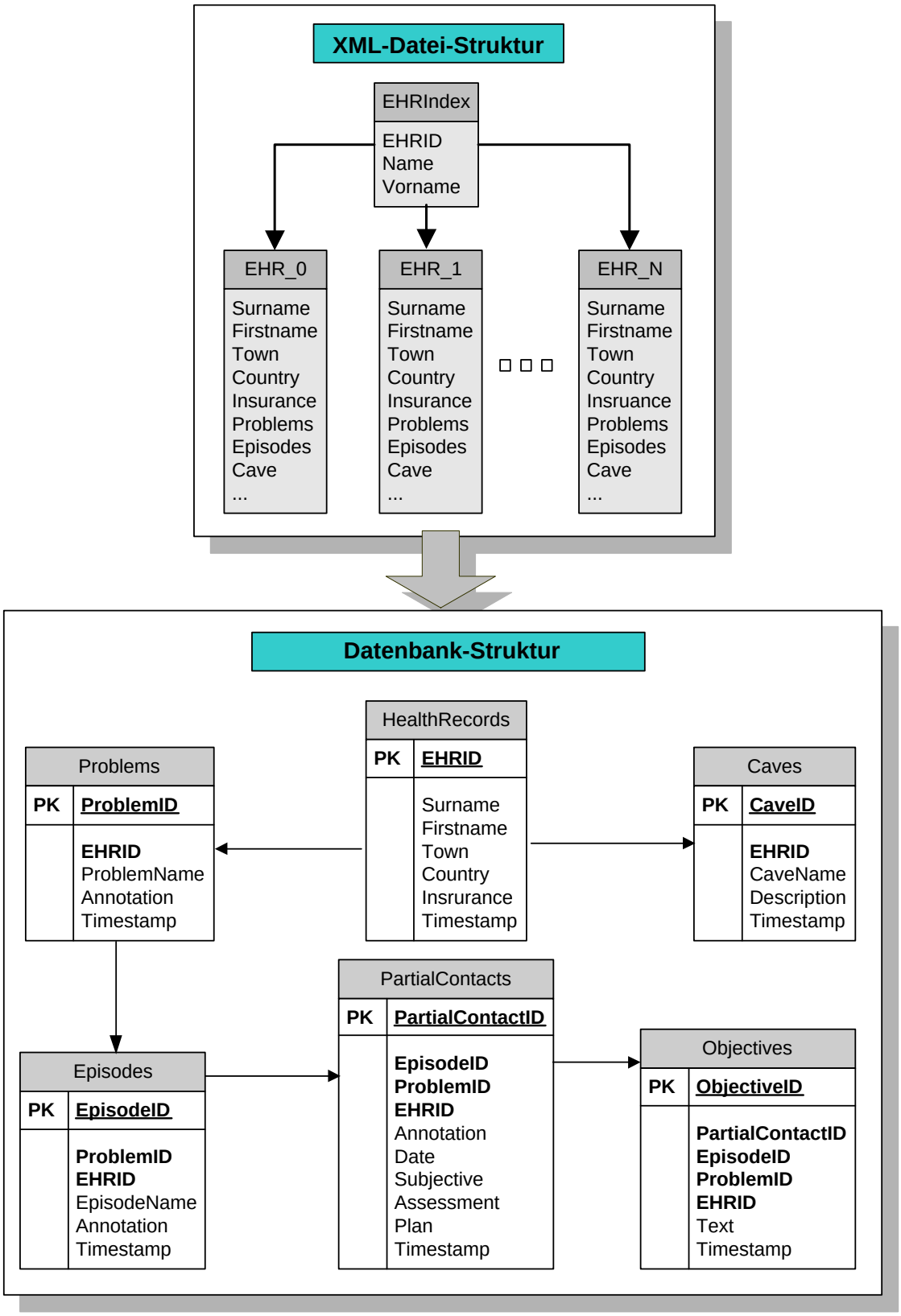

Abbildung 4.4: Überführung der XML-Datei-Struktur in eine DB-Struktur

Diagram die Primärschlüssel den ERD-Konventionen entsprechend unterstrichen und die Fremdschlüssel mit unterbrochener Unterstreichung hervorgehoben. Das ERD visualisiert noch einmal die Abhängigkeiten der Tabellen untereinander. Allerdings wurde zu Gunsten der Übersichtlichkeit nur ein Teil der Attribute eingezeichnet.

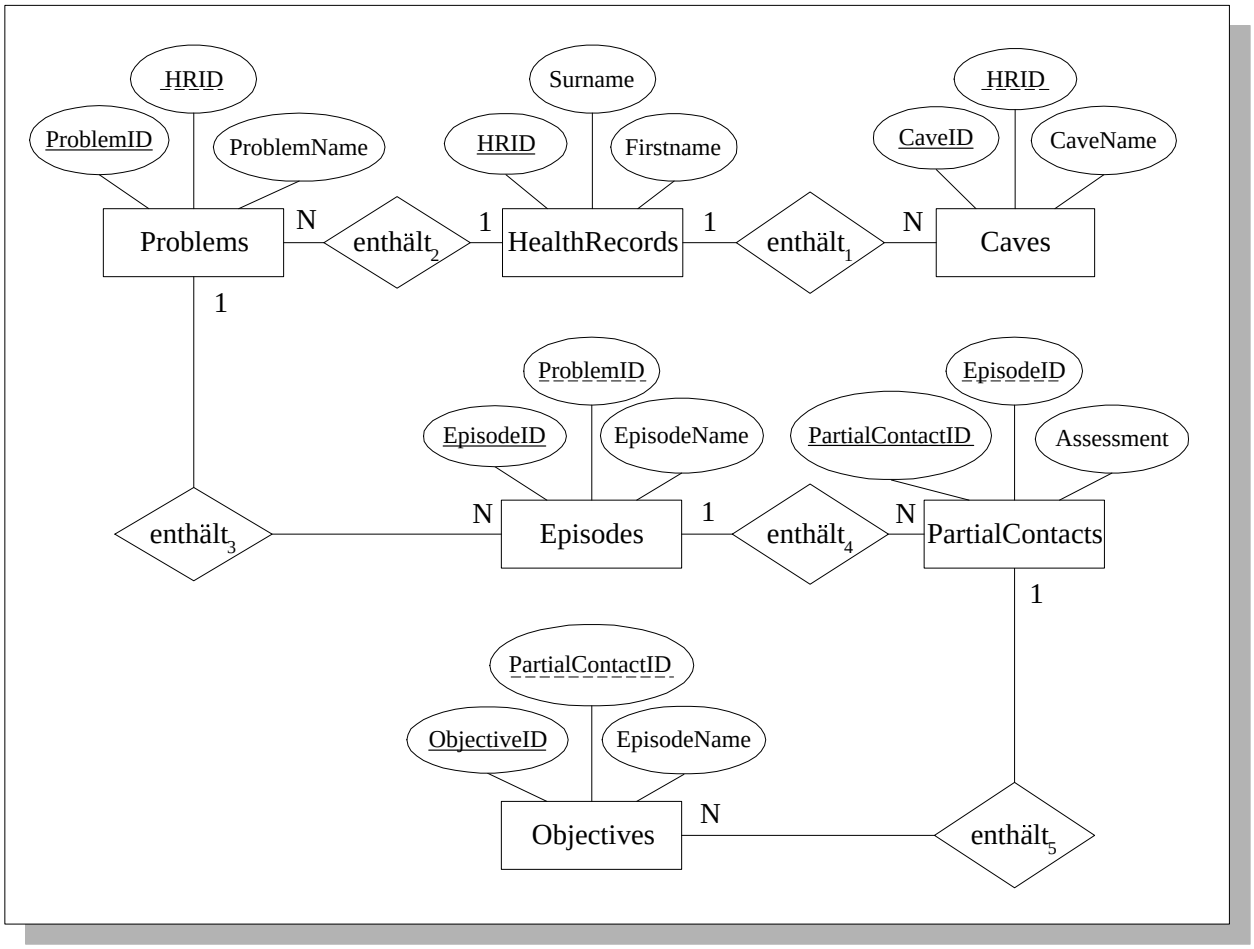

Abbildung 4.5: Das Entity Relationship Diagram der Datenbank

Sollen Daten in ein oder mehrere Tabellen geschrieben werden, findet zunächst eine Überprüfung der existierenden Einträge in der Datenbank statt. Sie testet auf das Vorhandensein des Primärschlüssels unter dem der Datensatz abgelegt werden soll. Ist dies nicht der Fall wird eine Insert-Anweisung des SQL-Befehlssatzes ausgeführt, anderenfalls erfolgt ein Update.

Jeder Schreibvorgang in eine Tabelle erhält zusätzlich zu den Nutzdaten einen *Timestamp*, der beim Laden mitgesendet wird. Er ermöglicht es, zu überprüfen, ob die augenblicklich

im Domain-Modell gehaltenen Daten die Aktuellsten sind, oder ob ein anderer Anwender bereits darauf geschrieben hat und deshalb ein Abgleich der Informationen notwendig ist.
Erfolgt bei Abarbeitung einer mehrstufigen Transaktionen (siehe Abschnitt 3.1.6) ein Fehler, wird ein Rollback initiiert und somit sämtliche bis zu diesem Zeitpunkt vollzogenen Modifikationen in der Datenbank rückgängig gemacht.
Die referenzielle Integrität wird durch die Vergabe von Fremdschlüsseln sichergestellt. Zugunsten der Performance wurde auf eine Normalisierung verzichtet und diese Fremdschlüssel als redundante Informationen in die entsprechenden Tabellen übertragen. Dabei wurde der hierarchische Aufbau des Domain-Modells berücksichtigt, d.h. die Tabellen, welche Klassen einer unteren Ebene umsetzen, enthalten alle Primärschlüssel ihrer übergeordneten Tabellen als Fremdschlüssel. Damit ist eine performantere Zuordnung von Teilinformationen zum Domain-Modell möglich und die Implementierung vereinfacht sich ebenfalls. So führt beispielsweise das Löschen eines Eintrages in der Tabelle *HealthRecords* dazu, das alle mit ihm verbundenen Einträge in den anderen Tabellen automatisch entfernt werden.

Die Primary Keys

Die Primärschlüssel einer Tabelle dienen der eindeutigen Identifikation eines Datensatzes. Dazu wurde in Abschnitt 3.1.8 eine Variante beschrieben - die Nutzung der vom DBMS PostgreSQL erzeugten OIDs. Bei *Res Medicinae* kommt aber nicht ausschließlich die Speicherung in eine Datenbank, sondern auch in lokale Dateien zum Einsatz. Die OIDs werden beim Schreiben eines neuen Datensatzes in eine Tabelle erzeugt. Würde der Datensatz beim Anlegen nicht zentral in die Datenbank geschrieben, sondern lokal abgelegt, so stünde hierfür noch keine eindeutige Identifikationsnummer zur Verfügung. Weiterhin ist es möglich, dass später hinzugefügte Datenbanken keine eigenen Routinen zum erzeugen von OIDs besitzen, so dass es vorteilhafter ist, von Anfang an eine eigene Methodik für die Primärschlüsselgenerierung zu entwickeln. Eine eigene Klasse *ObjectID* übernimmt diese Funktion. Sie bildet durch Konkatenation von Netzwerkadresse des Clients und der aktuellen Zeit eine eindeutige Identifikationsnummer. Der ermittelte Zeitwert wird auf die Nanosekunde genau verarbeitet.

Der folgende Java-Programmausschnitt erzeugt die OIDs.

```
1:  public String getNewOId() {
2:      String idString = InetAddress.getLocalHost().toString();
3:      idString = idString.substring(idString.indexOf(''/'') + 1);
4:      idString = idString.replace('.', '_');
5:      long time = System.currentTimeMillis();
6:      idString = idString + ''_'' + time + Math.round(getNanos(time)/100000);
7:      return idString;
8:  }
9:  public long getNanos(long time) {
10:     return new Timestamp(time).getNanos();
11: }
```

Daraus resultiert die OID-Struktur: <IP-Adresse>_<aktuelle Zeit>. Die aktuelle Zeit wird mit zwei Aufrufen zusammengesetzt. Zeile 5 liefert die Anzahl der Millisekunden seit Mitternacht des ersten Januars 1970. Zu dieser Zahl werden in Zeile 6 die Nanosekunden ermittelt und angehangen. Die Methode *InetAdress.getLocalHost()* liefert einen String, bestehend aus Hostname und Internetadresse (in dieser Reihenfolge), die durch einen Slash voneinander getrennt sind. Da nur die IP-Adresse notwendig ist, wird der Rest in Zeile 3 abgeschnitten. Die Punkte in der IP-Adresse werden in Zeile 4 durch Unterstriche ersetzt, um ein einheitliches Trennzeichen zwischen den einzelnen Zahlen zu definieren. Eine spätere Identifikation des Erstellers wird somit möglich.
Für jeden neuen Datensatz wird direkt nach dem Anlegen im Programm, d.h. noch vor dem ersten Abspeichern, mit *getNewOID()* eine OID angefordert. Die Mechanismen für die Speicherung von XML-Dateien nutzen diese eindeutige Identifikationsnummer als Dateinamen. In den Tabellen der Datenbank kommt sie als Primärschlüssel zum Einsatz. Damit lassen sich bei Ausfall des Netzwerkes neue Datensätze erzeugen und später in die Datenbank integrieren. Konflikte mit anderen Primärschlüsseln können nicht auftreten.

Das SQL-Package

Martin Fowler empfiehlt in seinen Ausführungen zu den Mustern *Domain Model* und *Data Mapper* [Fow02], die SQL-Statements in einem separaten Paket zu verwalten. Das verhindert zum einen die direkte Abhängigkeit des Domain-Modells von der Datenbank und zum anderen gestaltet es die Software-Architektur übersichtlicher. Die Umsetzung in *Res Medicinae* wurde so vorgenommen, dass zunächst eine oberste Klasse *SQLStatement* die wesentlichen Operationen definiert, analog dem Muster *Layer Supertype*. Von ihr erben die spezialisierteren Statement-Typen für Suchen, Einfügen, Modifizieren, Löschen der Daten und Tabellen. Sie implementieren jeweils eine eigene *execute()*-Methode, die entsprechende Operationen für das Arbeiten auf der Datenbank ausführt. Auf unterster Ebene ist jedes Statement in einer entsprechenden Klasse gekapselt. Für die Instanzen dieser Klassen liefert *execute()* eine Ergebnismenge als Array, bestehend aus Vektoren, an das aufrufende Objekt zurück. Der erste Vektor enthält dabei die Spaltennamen der ausgelesenen Tabellen, so dass eine eindeutige Einordnung der Daten in das Domain-Modell erfolgen kann. Allerdings liefern nicht alle Datenbankoperationen eine Ergebnismenge zurück. Vielmehr betrifft das ausschließlich die Suchmethoden. Fehlgeschlagene Operationen werden mittels Exception an das aufrufende Objekt gemeldet, so dass bei Ausführung einer mehrstufigen Transaktion, siehe Abschnitt 3.1.6, ein Rollback durchgeführt werden kann.

4.5.3 Resümee

Die Domäne bestehend aus Domain-Modell, Data Mapper und Persistenzmechanismen bietet eine flexible und zugleich robuste Struktur gegenüber Erweiterungen und Anpassungen. Die an eine moderne Persistenzschicht gestellten Anforderungen werden dabei erfüllt:

- Unterstützung verschiedener Persistenzmechanismen: Dateien in einem XML-Format und eine relationale Datenbank
- Vollständige Kapselung aller Persistenzmechanismen durch die Anwendung der Möglichkeiten des objektorientierten Porgrammierparadigmas

- Multi-Objekt Aktionen: Simultane Bearbeitung mehrerer Objekte. Mit einer Anfrage werden Daten für verschieden Objekte des Domain-Modells angefordert.
- Unterstützung von Transaktionen
- Erweiterbarkeit
- Benutzung von Objektidentifizierungsnummern (OIDs)
- Verwendung der Structured Query Language bzw. Embedded SQL
- Multiple Verbindungen, d.h. gleichzeitige Verbindungen zu weiteren Datenbanken, sind momentan noch nicht umgesetzt, können aber bei Bedarf durch die Flexibilität der Mappingschicht problemlos hinzugefügt werden.
- *Cursor* werden vom zugrunde liegenden Datenbankmanagementsystem unterstützt, sind aber ebenfalls noch nicht in der Klassenstruktur enthalten, da ihre Verwendung bisher noch nicht als sinnvoll und notwendig erachtet wurde.

Für die Abfrage des aktuellen Zustandes der Datenbank und Tabellen wurde ein Werkzeug namens SQuirreL verwendet, dass bei Bedarf in jedes Modul eingebunden und vom Anwender gestartet werden kann. Es handelt sich hierbei um ein weiteres, in Java implementiertes *Open Source* Projekt, das ebenfalls bei Sourceforge Unterstützung findet und im Internet erhältlich ist [Bel02].

4.6 Der Data Mapper als Kommunikationssteuerung

Sämtliche, im Anschluss beschriebenen Kommunikationsparadigmen nutzen die Operationen der Mapping-Schicht. Für jede dieser Realisierungen ist im Klassenmodell ein spezialisierter, bereits angesprochener Assembler vorgesehen und weiterhin zusätzliche Klassen für die Optimierung und Separierung der Kommunikation. Dafür kommen einige der in Abschnitt 2.2 erläuterten Muster zum Einsatz. Ihre Umsetzung wird im Folgenden am Beispiel Java-RMI erläutert.

Die Klasse *RMIDTO* entspricht dem Kernstück des *Data Transfer Object* -Musters für eine RMI-Middleware. Der *RMIAssembler* wird von *RMIServicesImpl* als Remote Facade benutzt und bekommt die ferne Anfragen eines Clients delegiert. Unter Benutzung verschiedener Methoden des *RMIAssembler*s werden die gewünschten Daten aus dem Domain-Modell zusammengetragen und in einem *RMIDTO* gekapselt. Letztendlich wird eine Kopie dieses DTOs an den Client als Ergebnis zurück gesendet. Dort erfolgt im *RMIAssembler* unter Verwendung spezialisierter Umkehroperationen eine Auswertung des Resultates, wonach die Daten in das eigene Domain-Modell übertragen werden. Das Sequenzdiagramm in Abbildung 4.6 soll diesen Vorgang veranschaulichen.
Für CORBA mit einer Versionsnummer kleiner als 3 sind Data Transfer Objects nicht anwendbar, da keine Objekte als Ganzes versendet werden können, sondern nur Datentypen, die der *Interface Definition Language* (IDL) zur Verfügung stehen. IDL ist eine programmiersprachenunabhängige Schnittstellenbeschreibungssprache. Sie realisiert, dass CORBA ebenfalls sprachenunabhängig ist. Daher besteht die Möglichkeit, die Clients beispielsweise in Java und den Server in C++ zu implementieren. Daraus ergibt sich allerdings der Nachteil, dass Objekte, die vom Client zum Server gesendet werden, von Letzterem nicht verarbeitbar sind. Deshalb werden Datentypen verwendet, die in möglichst vielen Programmiersprachen vorhanden oder zumindest leicht in sprachenspezifische Typen überführbar sind. Aus diesem Grund muss anstelle des bei RMI nutzbaren DTOs ein anderer Datentyp zur Übertragung der Domain-Daten verwendet werden, beispielsweise ein Struct oder ein Array aus Strings. Ab CORBA 3 ist das Versenden von Objekten möglich, denn IDL spezifiziert einen neuen Typ, genannt *valueType*, der diesen Vorgang möglich macht [Vin02].

4.6.1 Java-RMI

Die Remote Method Invocation oder kurz RMI [SUN95] zu verwenden bietet sich an, da hierfür im Java Development Kit von SUN Microsystems eigens ein Package implementiert wurde, d.h. der Applikation keine weitere Software zur Verfügung gestellt werden muss.
Eine jede verteilte Anwendung benötigt immer mindestens drei Teile für eine Client-/ Server-Kommunikation. Zum einen das Remote-Interface des Servers, welches die RMI-

Standardschnittstelle *java.rmi.Remote* erweitert. Es definiert Methoden, die später vom Client, dem zweiten Bestandteil, als Dienste genutzt werden können. Eine Klasse *RMIServicesImpl* implementiert diese Schnittstelle und erbt zusätzlich die Funktionalität von *javax.rmi.PortableRemoteObject*. Der Server und damit der dritte Teil einer verteilten Kommunikation meldet die Dienste an einem *Name Service* an, welcher die Aufgabe einer Registrierung übernimmt. Nun wartet der Server in einer sich ständig wiederholenden Schleife auf eingehende Anfragen. Die Registrierung der Anwendungsdienste erfolgt mit diesen Anweisungen:

```
RMIServicesImpl rmiServices = new RMIServicesImpl();
Context initialNamingContext = new InitialContext();
initialNamingContext.rebind(''RMIServicesResMedicinae'', rmiServices );
```

Über den Name Service lässt sich die Lokation sämtlicher angemeldeter Dienste eindeutig identifizieren. Adressiert wird dieser Dienst über einen RMI-URL, analog zum Uniform Resource Locator von TCP/IP. Der Name Service kann sich daher an beliebiger Stelle in einem Netzwerk befinden. Nachteilig ist allerdings, dass bei Änderung der Position dieses Name Services, alle Clients und Server an den damit verbundenen neuen URL angepasst werden müssen.
Der Client selber holt sich mittels Lookup des Name Services eine Referenz auf die Remote-Schnittstelle. Gleichzeitig wird die erlangte Objektreferenz auf den gewünschten Typ mittels Typecast abgebildet bzw. beschränkt:

```
RMIServices rmiS = (RMIServices)Naming.lookup(
    ''rmi://resmedicinaeserver/RMIServicesResMedicinae''
);
```

Dabei haben die einzelnen Teile des RMI-URL, der als String an lookup() übergeben wird, nachfolgende Bedeutung und Gestalt. Beim Weglassen des Name-Service-Ports wird der Standardport 1099 angenommen.

```
rmi://<host-name>[:<name-service-port>]/<service-name>
```

Die eigentliche Kommunikation zwischen Client und Server erfolgt über Surrogate-Objekte,

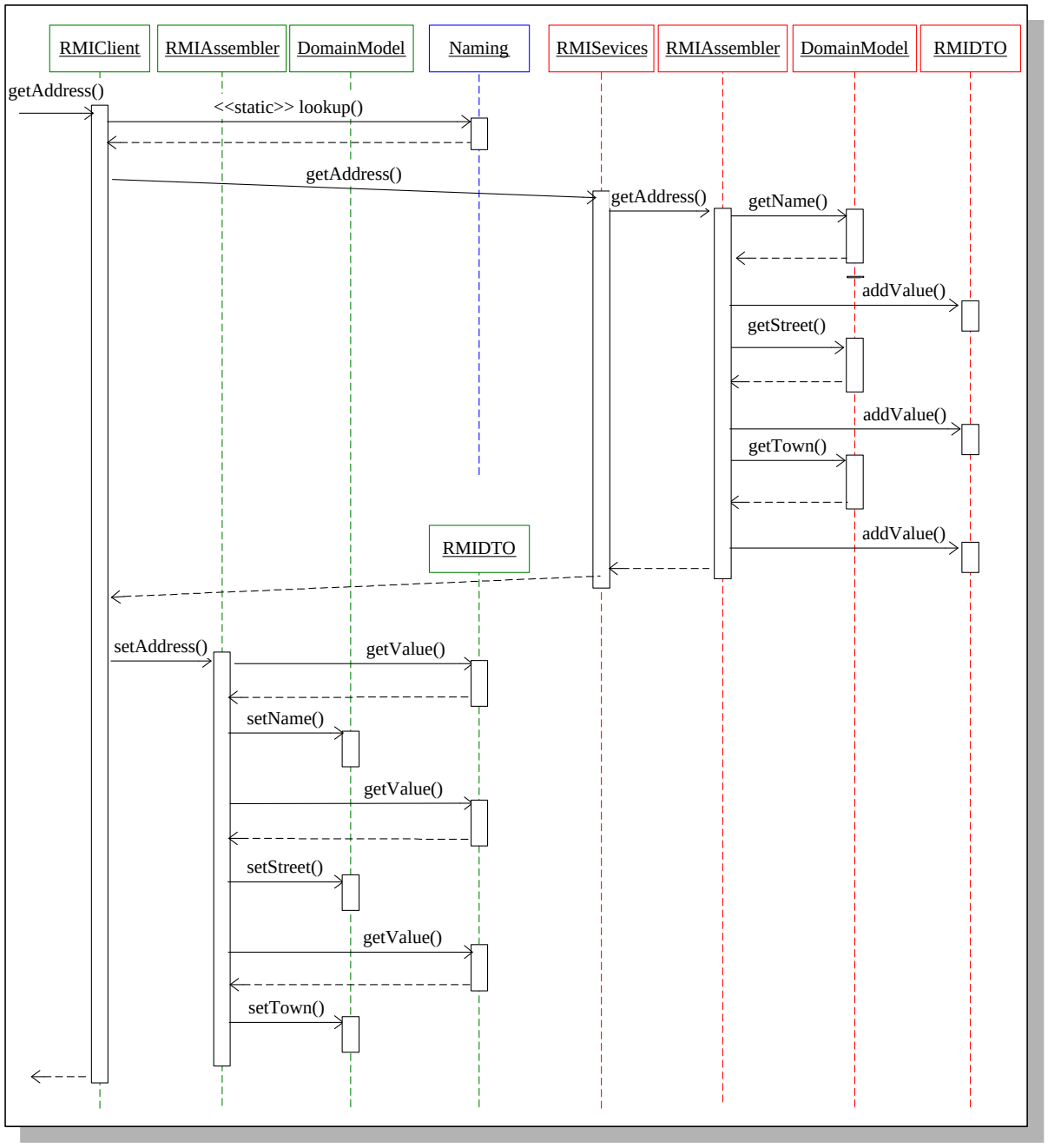

Abbildung 4.6: Sequenzdiagramm für eine Client-Server-Kommunikation

Client-Stub und Server-Skeleton. Sie sind für das Marshalling und Unmarshalling, also das Ver- und Entpacken, sowie für die Weiterleitung der Daten an das Client- bzw. Server-Objekt verantwortlich, wobei RMI für das Verpacken eine Objekt-*Serialisierung* verwendet [jGu00]. Das sich anschließende Sequenzdiagramm in Abbildung 4.6 beschreibt einen einfachen RMI-Kommunikationsprozess zum Übermitteln der Adresse eines Patienten. Für eine bessere Visualisierung sind die Objekte der unterschiedlichen Kommunikationspartner farblich differenziert. Clientobjekte sind grün, Serverobjekte rot und Name Service -Objekte blau dargestellt. Die Klasse Naming befindet sich ebenfalls auf dem Client-Rechner. Sie wurde hier lediglich farblich abgegrenzt, damit ersichtlich wird, dass die Methode *lookup()* auf einen separaten Prozess zugreift, der unter Umständen auf einem entfernten Computer arbeitet. Geneigte Pfeile dienen als Darstellungsform von Nachrichtenaufrufen in unterschiedlichen Prozessen bzw. auf verschiedenen Rechnern und implizieren eine längere Laufzeit als Aufrufe innerhalb ein und desselben Prozesses. Für gewöhnlich erfolgt der Aufruf von lookup() nur einmal bei der Initialisieung des Clients. Die Referenz auf den Name Service wird in einer Variablen gehalten.
Der Server muss zu Beginn einen Security Manager installieren. Er wird benötigt, damit geladene Klassen keine Operationen ausführen, die nicht erlaubt sind. Wenn kein Security Manager eingerichtet wurde, können weder Client noch Server eine RMI-Klasse laden.
Das Data Transfer Object *RMIDTO* ist als Hashtable modelliert. Über einen festgelegten Schlüssel werden die Daten mittels *addValue()* serverseitig hinzugefügt. Der Client greift auf die ihm zugesandte Kopie mit dem selben Schlüssel über die Methode *getValue()* zu und überträgt die so erhaltenen Daten anschließend unmittelbar in das Domain-Modell.

4.6.2 JMS

Eine aufwendige Middleware zu verwenden ist nur sinnvoll, wenn auch eine Anzahl von bereitgestellten Diensten und Features genutzt wird. Hinzu kommt dass die verteilte Kommunikation über ein Netzwerk mit nicht vernachlässigbaren Laufzeiten verbunden ist. Bei einer lokalen Interprozesskommunikation, d.h. für den Austausch von Daten zwischen zwei *Res Medicia-ne* -Modulen auf dem selben Rechner, genügen simplere Dienste. Der *Java Message Service*

(JMS) bietet sich hierfür an.
Eigentlich handelt es sich dabei, im Gegensatz zu den anderen beschriebenen Kommunikationsparadigmen um ein asynchrones Nachrichten-orientiertes Verfahren. Das heißt, auf die Antwort eines fernen Methodenaufrufes wird nicht unmittelbar gewartet bevor ein weiteres Arbeiten möglich ist. Teilweise sind Antworten auf eine Nachricht gar nicht erforderlich, deshalb verringert dieses Verfahren den notwendigen Datenverkehr innerhalb eines Netzwerkes. Neben dieser Primäridee für JMS existiert ebenfalls noch die Möglichkeit für einen synchronen Datenaustausch. Der Anwender kann dementsprechend wählen, welche der beiden Techniken für seine Anwendung günstiger ist.

4.6.3 Resümee

In Abbildung 4.7 wird einmal die gesamte Kommunikationsstruktur innerhalb der Domäne eines *Res Medicinae* -Moduls veranschaulicht. Man erkennt die Strukturierung als Zweischichtenmodell. Die beiden Applikationen, die sich nicht zwingend auf dem selben Rechner befinden müssen, tauschen ihre Daten über die in den vorangehenden Abschnitten beschriebenen Kommunikationsparadigmen aus, wobei auf die Berücksichtigung von einzelnen Details, wie beispielsweise den *Name Service* von RMI und CORBA, verzichtet wurde.
Der Zugriff über *DataTransferAssembler* auf *PersistanceAssembler* oder eine seiner Subklassen darf nicht realisiert werden, da dies ein Sicherheitsrisiko darstellt. Angenommen ein Client verfügt nur über einen eingeschränkten Zugang zur Datenbank. Dann bestünde die Möglichkeit, dass sich dieser Client über eine Remote-Operation Zugang zu den nicht autorisierten Daten verschafft, da der andere Anwender über diese Rechte verfügt. Ebenso verhält es sich mit den lokalen XML-Dateien des Remote-Rechners. Aus diesem Grund wird dem Client lediglich der Zugriff auf das aktuell im Speicher befindliche Domain-Modell über den *DataTransferAssembler* des entfernten Rechners gestattet.
Noch ein Wort zur Architektur visualisiert in Abbildung 7.1 Anhang A. Die Instanz der Klasse *AdvancedBasicApplication* enthält jeweils ein *DataTransferAssembler*-, ein *PersistanceAssembler*-, ein *Server*- und ein *Client*-Objekt. Die spezialisierten Server- und Client-Objekte enthalten eine Referenz auf ihre zugehörigen, spezialisierten Assembler. Da-

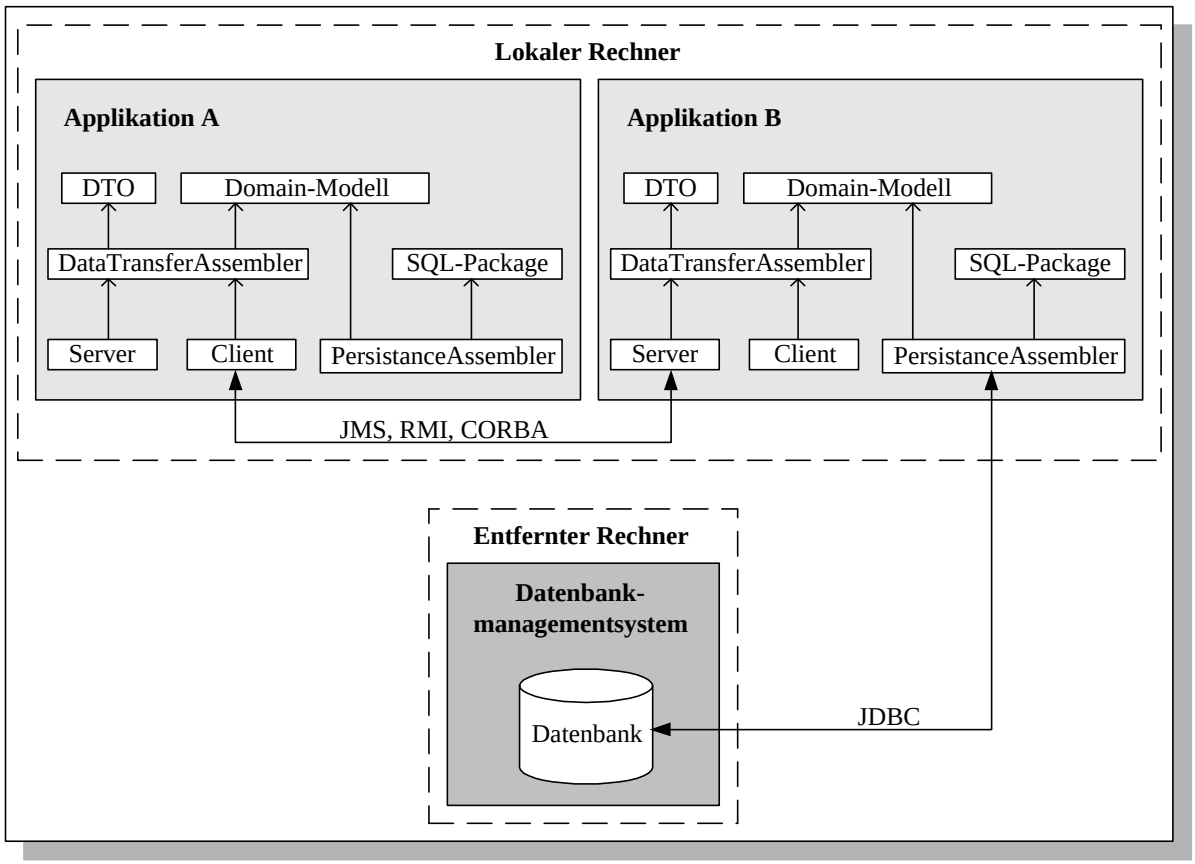

Abbildung 4.7: Res Medicinae als Zweischichtenmodell

mit ist der Applikation immer die aktuell ausgewählte Middleware für die verteilte Kommunikation bekannt. Es kann sogar in der selben Applikation, zur selben Zeit, für Client und Server jeweils unterschiedliche Middleware verwendet werden. Lediglich zusammengehörige Kommunikationspartner müssen sich auf ein Verfahren einigen.

Kapitel 5

Der Protoyp: Das Formularmodul - ReForm

Zusammengesetzt aus *Re* für Res Medicinae und *Form*, englisch für Formular, lässt die Bezeichnung dieser Komponente bereits ihren Aufgabenbereich erahnen. Sie ermöglicht es dem Anwender zu vorhandenen Patienten-Karteien Formulare über vorgefertigte Eingabemasken auszufüllen und in medizinische Formularvordrucke auf Papier auszugeben. Abbildung 5.1 zeigt ein eigens für Res Medicinae mit Java-Swing realisiertes Rezept-Formular. Es existieren von anderen Softwareproduzenten veröffentlichte Vorschläge [GNU02] für Eingabemasken von medizinischen Formularen, denen praktizierende Ärzte bereits ihre positive Zustimmung für den Gebrauch gegeben haben, so dass sich im weiteren Softwareentwicklungsprozess daran orientiert werden sollte.

Der Nutzer möchte möglichst wenige Eingaben wiederholt tätigen. Deshalb werden alle vorhandenen Daten bei Anwahl eines bestimmten Formulars unmittelbar aus dem Domain-Modell in die entsprechenden Felder übertragen. Um dies zu gewährleisten, wird wenigstens eine Umsetzung der im vorangegangen Kapitel beschriebenen Interprozesskommunikationsparadigmen benötigt. Mit ihrer Hilfe fordert das *ReForm* eine anderes Modul, beispielsweise *Record* auf, seine Domain-Daten zu transferieren. Da alle Komponenten die gleichen elementaren, von *AdvancedBasicApplication* geerbten Fähigkeiten besitzen, besteht ebenfalls die Möglichkeit, benötigte Daten unmittelbar aus der Persistenzschicht zu laden. Damit wird ersichtlich, dass durchaus differierende Patientenkarteien von den verschiedenen Modulen bearbeitet werden können. Allerdings muss deshalb bei der Interprozesskommunikation sichergestellt werden, dass die übermittelten Daten in den richtigen Datensatz geschrieben

werden. Die exakte Identifikation ist durch den zusätzlichen Transfer der OID der jeweiligen Patientenkartei und einem simplen Vergleich mit dem Identifikator des aktuellen Domain-Modells problemlos möglich.

Für die Formulare verwendet man ebenfalls das Design-Pattern *Modell View Controller* (Abschnitt 2.2.1). Dazu wurde jeweils eine Elternklasse mit grundlegenden Eigenschaften implementiert. Jede abgeleitete Klasse muss bestimmte Methoden überladen, wie beispielsweise ein neuer View, der unter anderem eine Methode zum Erzeugen einer Druckmaske zu überschreiben hat. Zur Notwendigkeit dieser Maßnahme wird auf den anschließenden Abschnitt verwiesen.

Ausgehend davon, dass allgemeine Patienteninformationen wie Name, Adresse, Versicherung auf allen medizinischen Formularen einzutragen sind, wurde ein Teil-View entworfen, der für die Präsentation dieser Informationen verantwortlich ist. Man kann sie später mit anderen Teil-Views für spezielle Formulareigenschaften kombinieren und erspart sich damit den Aufwand einer jeweiligen Neuimplementation.

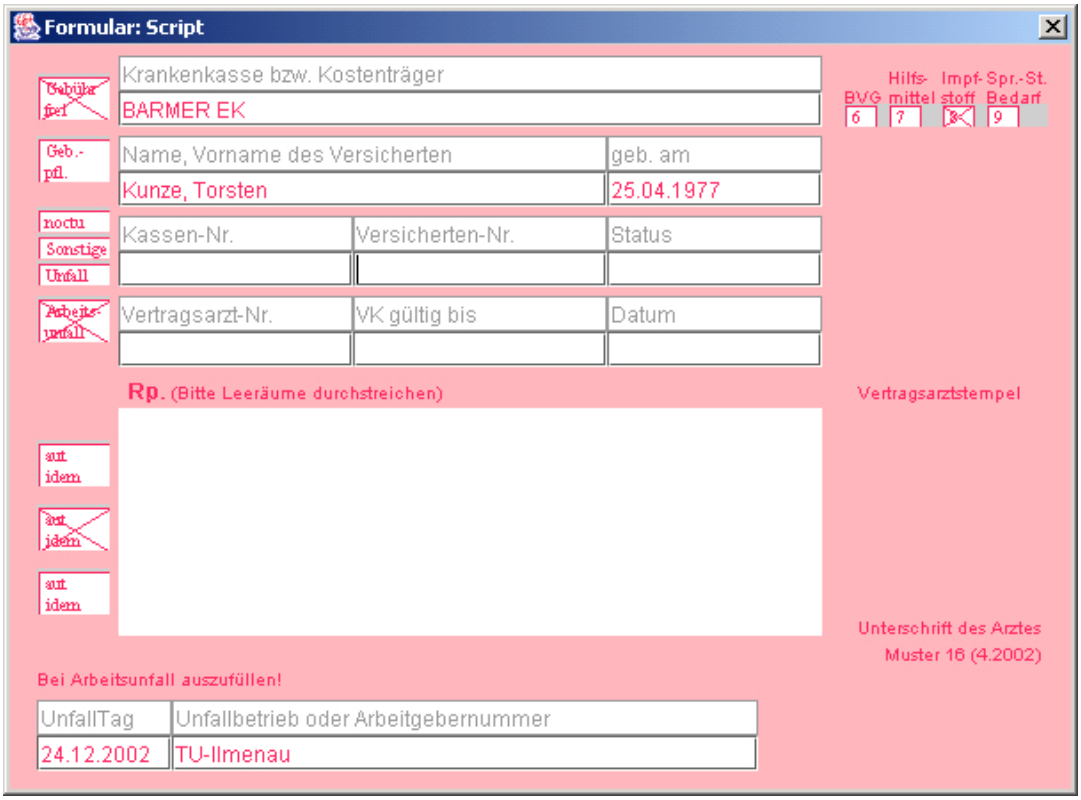

Abbildung 5.1: Das Rezept - Formular

5.1 Drucken der Formulare

Um das Drucken nicht nur auf die Formulare zu beschränken, sondern für beliebige Anwendungsbereiche zugänglich zu halten, wurde die Funktionalität von den Formularelternklassen separiert und eine spezialisierte Klasse erstellt, die diese Aufgabe bewerkstelligt.
Es bieten sich zwei Möglichkeiten des Druckens.
Zum einen kann eine Java-Swing-Komponente unmittelbar an diese Klasse übergeben werden. Bei Einhaltung des Design-Pattern Model View Controller handelt es sich hierbei um den aktuellen Zustand des jeweiligen View-Objektes. Dieses wird bei Bedarf, sofern es den bedruckbaren Bereich des Papierformates überschreitet, herunterskaliert. Dabei offenbart sich der Nachteil des Verfahrens, denn mit dem Skalieren werden Schrift und View-Komponenten meist stark verzerrt, da Java keine optimierten Bildbearbeitungsoperationen zur Verfügung stellt. Nach Möglichkeit sollten deshalb nur Views gedruckt werden, die vollständig auf das gewählte Papierformat passen.
Diese Variante erfüllt nicht die Anforderungen des *ReForm*-Moduls, daher wurde noch eine Weitere evaluiert. Sie setzt die Existenz einer eigens hierfür zuständigen Methode im View voraus, welche eine Maske zum Drucken von Text und anderen Swing-Komponenten erstellt. Damit wird nicht mehr der vollständige View ausgegeben, vielmehr kann man in vorgefertigte Standardformulare drucken und diese Maske entsprechend des Papierformates gestalten. Der Nachteil ist ein erweiterter Implementierungsaufwand für die Masken-Methode. Diese zweite Variante kommt bei *ReForm* zum Einsatz.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Mit dem Einsatz von Software-Mustern in Applikationen bieten sich dem Entwickler eine Vielzahl an Möglichkeiten, Programmcode besser zu strukturieren. Das äußert sich in einer zugleich flexiblen Erweiterbarkeit der Funktionalität aber auch in Robustheit gegenüber Änderungen. In Kapitel 2 wurden einige dieser Muster vorgestellt. Ihr Einsatz zur Modellierung einer unabhängigen Domain-Schicht beschrieb Abschnitt 4. Die Geschäftslogik kann auf die Domain-Daten zugreifen, sie laden, modifizieren und speichern, ohne über eine direkte Abhängigkeit zu den verwendeten Persistenzmechanismen zu verfügen.
Die Sicherung der Geschäftsdaten kann entweder lokal in XML-Dateien oder in einer Datenbank erfolgen. Erst die Kombination mehrerer dieser Möglichkeiten erfüllt das Maß an notwendiger Zuverlässigkeit für heutige Softwaresysteme. JDBC als Kommunikationsprotokoll für die Interaktion zwischen der objektorientierten Programmiersprache Java und einem Datenbankmanagementsystem bietet eine sichere und leicht implementierbare Variante für die Speicherung von Informationen in einer zentralen und entfernten Datenbank.
Wie für Persistenzmechanismen existieren für die Optimierung der Interprozesskommunikation bereits verschiedene Strukturmuster. Sie ermöglichen es die Anzahl der Interprozessaufrufe zu minimieren und somit Lauf- und Antwortzeiten einzusparen. Wird die Methodik zur Verwaltung der Kommunikation ebenfalls in eine transparente Zwischenschicht verlagert, ist eine leicht modifizierbare Struktur realisierbar, die eine andernfalls notwendige Anpassung

von Geschäfts- und Präsentationslogik überflüssig macht.
Diese Zwischenschicht bildet *Layer PerCom*. Sie enthält als wesentliche Verbesserung gegenüber bekannten Ansätzen, die Möglichkeit nicht nur einen, sondern mehrere Persistenzmechanismen und gleichzeitig mehrere Kommunikationsparadigmen in einer dem Anwender transparenten Schicht zu vereinigen. Auf vertikaler Ebene ist eine Trennung in Persistenz und Kommunikation durchaus sinnvoll und wird auch in dem entwickelten Modell vorgenommen. So werden zwei verschiedene, spezialisierte Assembler (*PersistanceAssembler*, *DataTransferAssembler*) verwendet.

6.2 Ausblick

In Abschnitt 3.1.7 wurde ausgeführt, dass es häufig aus Kostengründen nicht möglich ist, eine Middleware zur Verwaltung der Datenbankzugriffe zu erwerben. Jedoch wäre zu überlegen, ob es nicht sinnvoll ist, eine eigene Middleware zu entwerfen. Sie müsste zunächst nicht in allen Einzelheiten vollständig sein und auch nicht von Anfang an mehrere Datenbank-Treiber unterstützen. Vielmehr genügt es, wenn sie den Anforderungen entsprechend erweiterbar ist und vorerst die Grundoperationen realisiert. Da eine Kommunikation mit CORBA und RMI im Projekt vorgesehen ist, wäre nach deren Implementierung das Einfügen dieses neu angepassten Data Mappers ein gute Lösung, um die momentan sehr umfangreichen Fat-Clients in ihrer Komplexität etwas herunterzuskalieren, so dass hauptsächlich nur noch Domain- und Präsentationslogik enthalten wären (siehe Abbildung 6.1). Das würde dem Anwender auch die Konfiguration der Datenbankserveranbindung ersparen und mögliche Lokationswechsel und Portierungen der Datenbank vereinfachen.
Im Vergleich mit Abbildung 4.7 erkennt man, dass im speziellen der Teil der Mapping-Schicht ausgelagert wurde, welcher sich mit der Verwaltung der Daten in der Datenbank befasst. Deshalb können der *ERAssembler* und das SQL-Package aus der Komponente entfernt und angepasst als Bestandteile des Data Mappers auf einem entfernten Rechner eingesetzt werden. Während eine lokale Interprozesskommunikation aus Komplexitäts- und Aufwandsgründen über JMS erfolgt, basiert der Datenaustausch zwischen Komponenten und Data Mapper auf

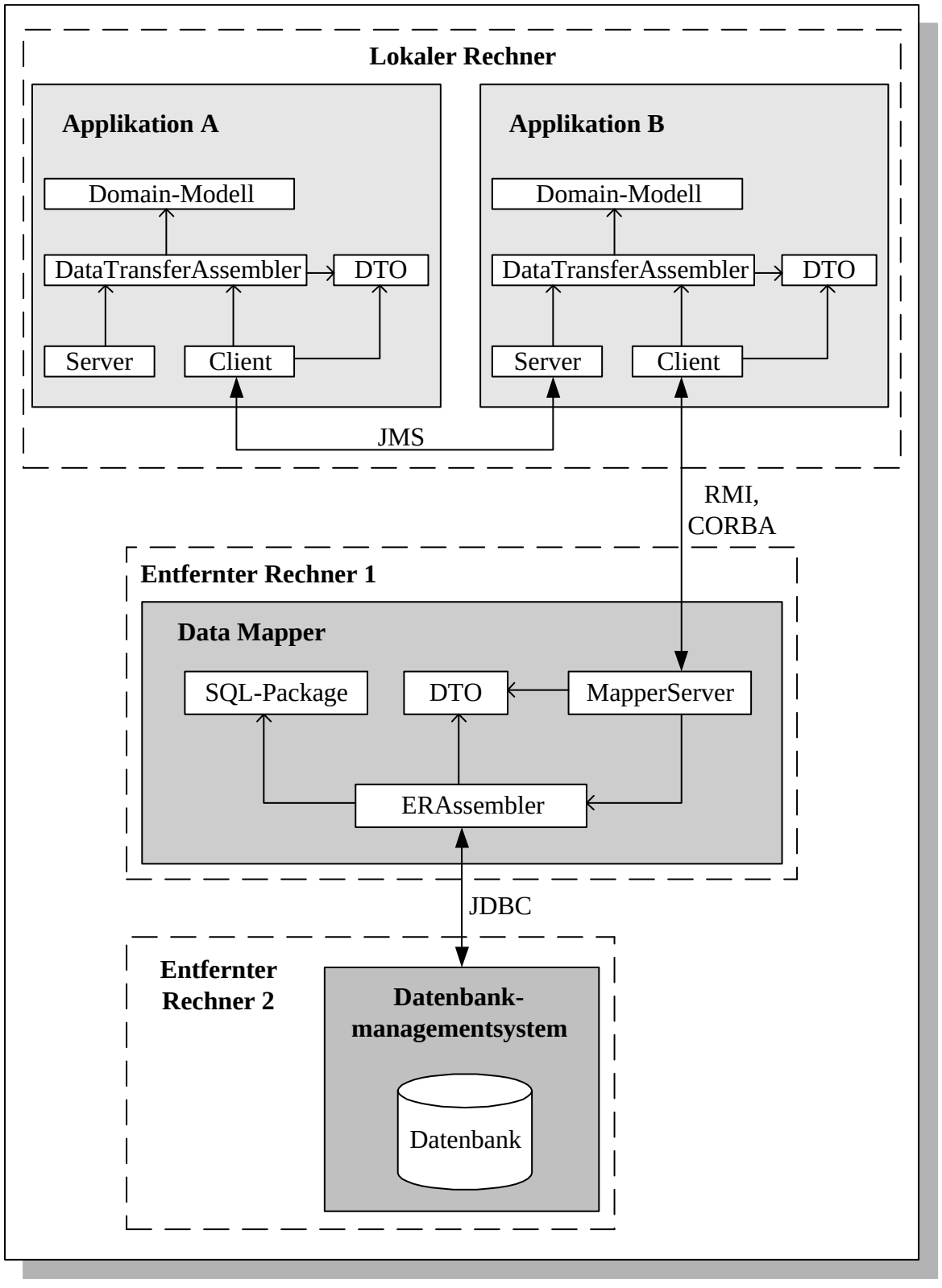

Abbildung 6.1: Res Medicinae als Dreischichtenmodell

RMI und CORBA, wobei die Daten in einem DTO verpackt und übertragen werden.
Data Mapper und DBMS können durchaus auch auf dem selben Remote-Rechner eingesetzt werden. Das reduziert allerdings die Flexibilität des Dreischichtenmodells.
Wie in der vorangegangenen Abbildung ersichtlich ist, wird durch diese Umsetzung eine reine Client-/ Server-Kommunikation möglich.
Dennoch kann nicht der gesamte Data Mapper auf einen entfernten Rechner verlagert werden. Wenigsten die lokale XML-Dateiverwaltung muss in den einzelnen Komponenten enthalten bleiben.

In Abschnitt 4.5.2 wurde erwähnt, dass sich die augenblickliche Tabellenstruktur der Datenbank dem aktuelle Entwicklungsstand der Applikation entsprechend noch in ihrer Anfangsphase befindet. Zusätzliche Tabellen könnten für die Speicherung von Daten im Zusammenhang mit anderen Modulen, wie beispielsweise Record, notwendig werden.

Formulare zu programmieren und sie entsprechend den Wünschen der Ärzte anzupassen ist eine Aufgabe, die nach Möglichkeit automatisiert ablaufen sollte. Zumal von Zeit zu Zeit Aktualisierungen oder neue Formulare erscheinen. Deshalb wäre es günstig, über einen Formulardesigner zu verfügen, mit dessen Hilfe und unter Vorgabe spezifischer graphischer Komponenten, View-Objekte auf einfache Weise erstellt werden können.

Eine wesentliche Neuerung bei *Res Medicinae* ist, die Vorteile des Internets und verteilter Anwendungen auch für Patientenkarteien nutzbar zu machen. Patientendaten ein und der selben Person könnten an verschiedenen Stellen der Welt gleichzeitig abgerufen und bearbeitet werden. So entfielen lange Wartezeiten beim Überweisen eines Patienten an einen Facharzt, da das Versenden der Patientenakte per Post überflüssig wäre. Alle notwendigen Informationen würden in einer zentralen Datenbank abgespeichert. Die Transaktionen erfolgten unter Beachtung des Datenschutzgesetz (DSG 2000), wobei der Einsatz zuverlässiger Verschlüsselungs- und Zertifizierungsmechanismen notwendig werden würde. Die anzustellenden Untersuchungen könnten ebenfalls Bestandteile weiterführender Teilprojekte werden.

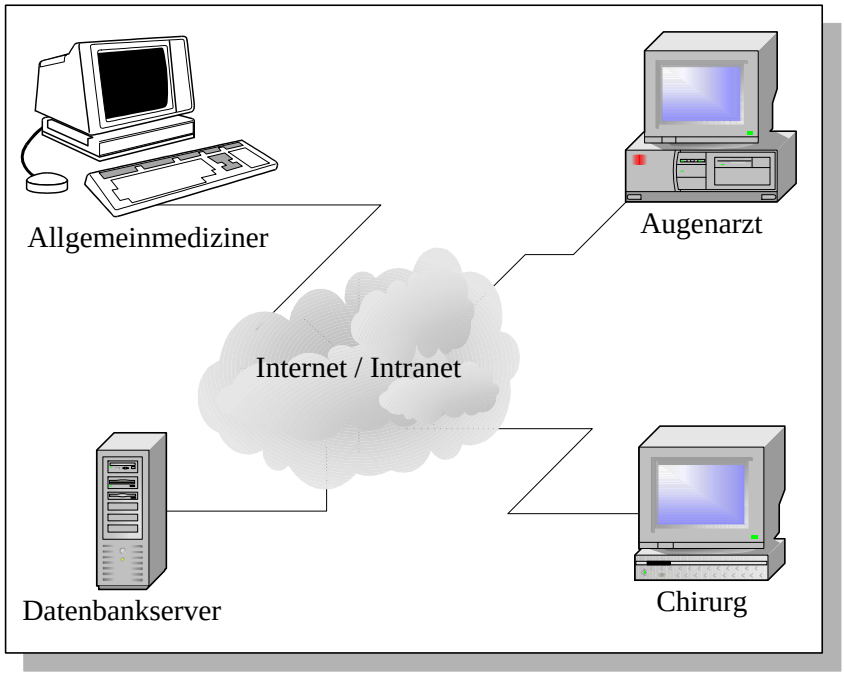

Abbildung 6.2: Einsatz von Res Medicinae im Internet

Eine andere Möglichkeit wäre, *Res Medicinae* einrichtungsintern, beschränkt zu nutzen, beispielsweise innerhalb eines Krankenhauses, um lediglich interne Daten zu verwalten. Diese Daten könnten aber ebenso wie im ersten Beispiel von jedem Arzt an einem Terminal abgerufen und modifiziert werden. Denkbar wäre auch, die Mediziner mit einer Art Handheld oder Tablet PC (einer tragbaren Rechnerversion) auszustatten, der über Funk mit der Datenbank des Krankenhauses verbunden ist. Dabei dürfte das Sende-/ Empfangssignal energetisch nur ein sehr niedriges Niveau erreichen, um die empfindlichen medizinischen Apparaturen nicht zu beeinflussen, wie es beispielsweise bei Handys der Fall sein kann. Es wären jedoch nur sehr kurze Distanzen innerhalb der Einrichtung zu überwinden und deshalb keine starken Signale notwendig.

Somit bestünde im Bereich des Funknetzes eine Ortsunabhängigkeit. Die Installation mit den Apparaturen am Krankenbett würden eine Funkverbindung gänzlich überflüssig machen. Dem Arzt bliebe es erspart, nach einzelnen Patientenbesuchen einen Terminal oder eine Workstation aufzusuchen. Er könnte seine elektronischen Akten gleich an Ort und Stelle verwalten.

Es gilt noch anzumerken, dass selbst innerhalb einer Einrichtung nicht alle Personen unein-

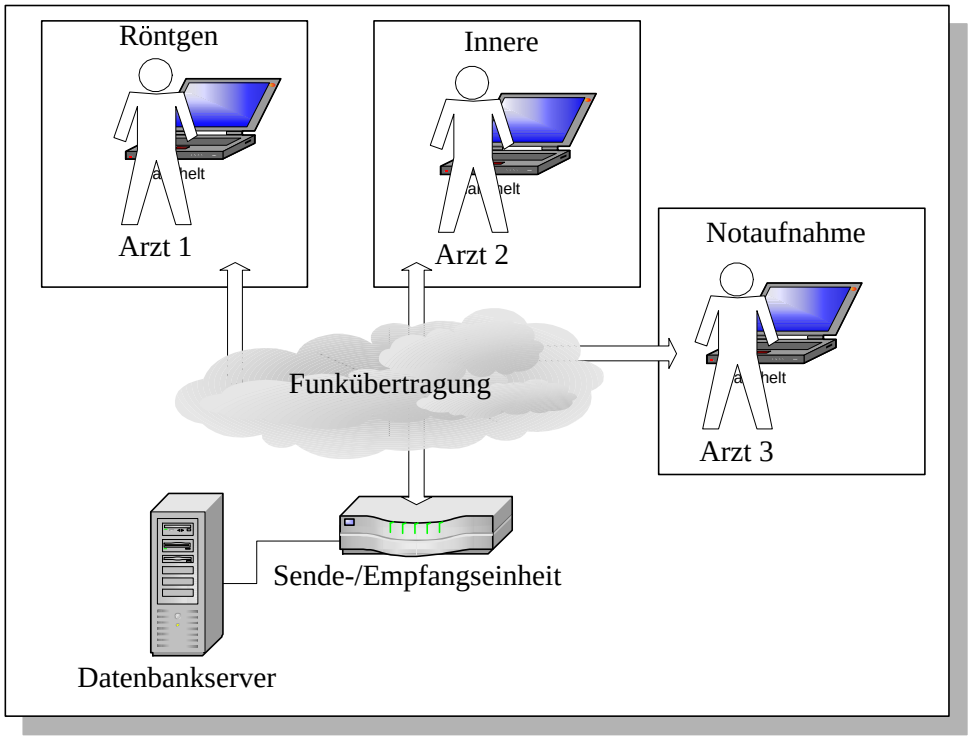

Abbildung 6.3: Einsatz von Res Medicinae einrichtungsintern

geschränkten Zugriff auf sämtliche Daten erhalten dürften. Über ein Login könnte man sich an der Datenbank anmelden und entsprechend seiner zugewiesenen Rolle im Rechte-System der Datenbank lediglich die frei gegebenen Informationen abfragen und modifizieren.

Anhang

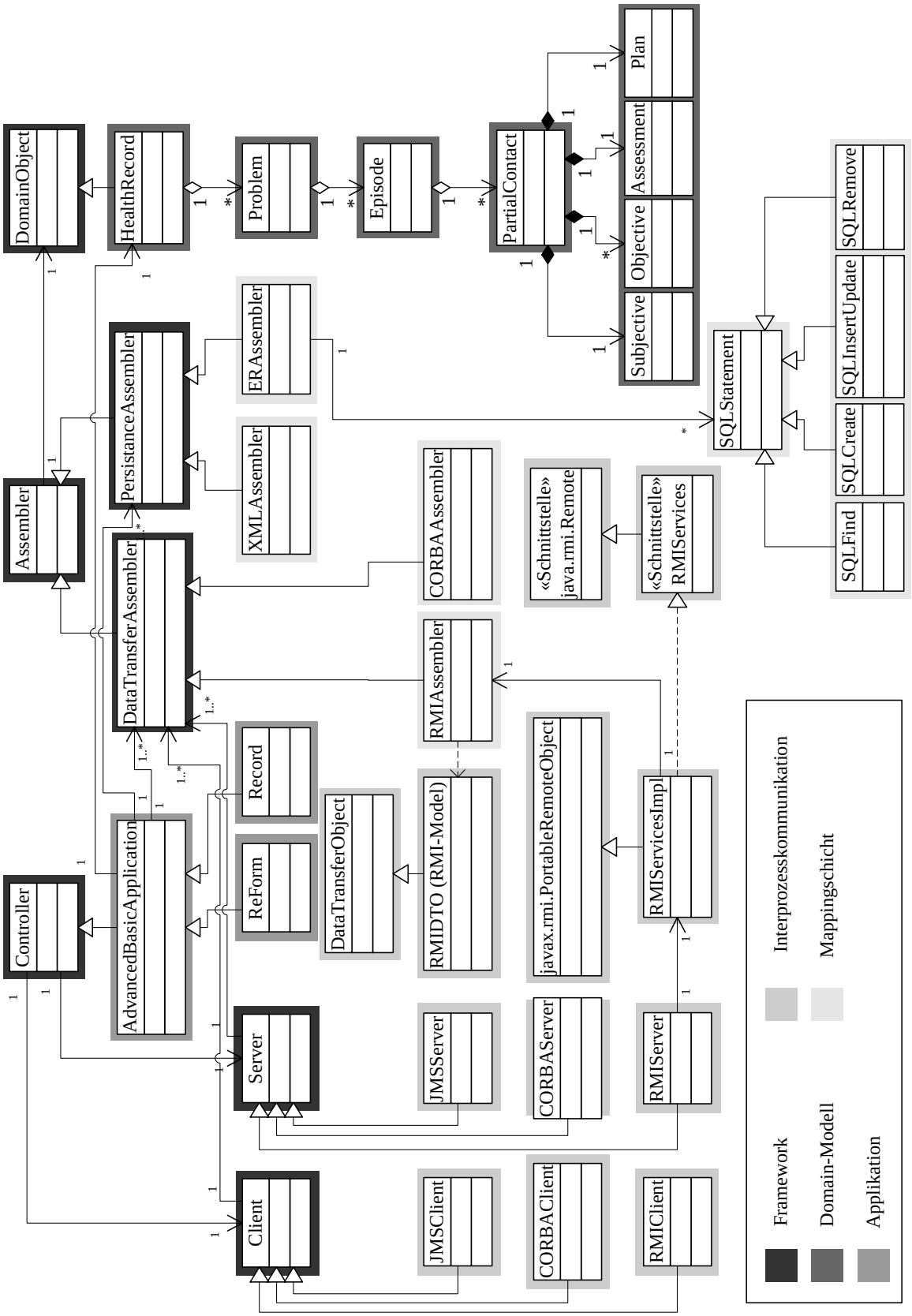

Abbildung 7.1: Klassendiagramm der in Kapitel 4 und 5 beschriebenen Architektur

Glossar

C

Callback-Methoden, S.32 Das sind Methoden, die an das Auftreten eines bestimmten Ereignisses (Events) gekoppelt sind, d.h. sie werden genau dann ausgeführt, wenn dieses ihnen zugewiesene Ereignis eintritt.

Cursorverarbeitung, S.30 Ein Cursor ist eine logische Verbindung zur Datenbank. Er stellt verschiedene Mechanismen bereit, die dem Anwender einen kontrollierten Abruf von Abfrageergebnissen ermöglichen. So kann zuerst ein Teil und anschließend ein weiterer Teil der Ergebnismenge verarbeitet werden, ohne eine erneute Anfrage an die Datenbank senden zu müssen.

O

Ontologie, S.36 Der Begriff stammt aus dem Griechischen. Zusammengesetzt aus "ontos", das Sein, und "logos", das Wort, beschreibt es die Lehre vom Sein, von den Ordnungs-, Begriffs- und Wesensbestimmungen des Seienden.

Open Source, S.1 Im Gegensatz zu kommerziellen Anwendungen werden Open Source Projekte nicht mit dem Ziel einer entgeltlichen Profitierung entwickelt. Der Quellcode ist frei zugänglich und erweiterbar. Man erhofft sich dadurch eine schnellere Verbreitung und eine große Zahl zusätzlicher, freiwilliger Entwickler, die ihren Ideen entsprechend, die Projekte weiterentwickeln. Zur Vermeidung des Missbrauches des frei zugänglichen Quellcodes unterliegen Open Source Applikationen meist gewissen Lizenzvereinbarungen (siehe Abschnitt 1.3).

R

referenzielle Integrität, S.30 Durch die Vergabe von Fremdschlüsseln werden Datensätze verschiedener Tabellen einer Datenbank als zusammengehörig markiert. Mittels die-

ser Referenzierung wirken sich Aktualisierungen einer Tabelle automatisch auch auf die anderen aus, wodurch sich "Datenleichen" vermeiden lassen. Ebenfalls wird das unkontrollierte Löschen eines Datensatzes oder einer Tabelle verhindert.

S

Serialisierung, S.16 Durch Serialisierung existiert eine Möglichkeit, ein Objekt, das sich im Hauptspeicher der Anwendung befindet, als Byte-Strom in eine Datei zu schreiben oder über eine Netzwerkverbindung zu transportieren. Das schließt natürlich auch den umgekehrten Weg mit ein, also das Rekonstruieren eines Objekts in das interne Format der laufenden Java-Maschine. Diesen umgekehrten Vorgang nennt man Deserialisierung.

State of the Art, S.3 Der aktuelle Zustand bestehender Technologien, wird in der Diplomarbeit in den Kapiteln 2 und 3 beschrieben, um eine Abgrenzung zur und die Notwendigkeit für die Neuerung der Entwicklung zu begründen.

T

Transaktionssteuerung, S.30 Unter Transaktionssteuerung versteht man den Einsatz von Transaktionen, um die Tabellen einer Datenbank von einem konsistenten Zustand vor Ausführung der Datenbank-Anweisungen in einen konsistenten Zustand nach Abarbeitung all dieser Anweisungen zu überführen. Transaktionen werden im Abschnitt 3.1.6 ausführlich besprochen.

Literaturverzeichnis

[Bel02] Bell, Colin, Greg Mackness: *SQuirreL*. http://squirrel-sql.sourceforge.net/, 2002.

[Boh03] Bohl, Jens: *Möglichkeiten der Gestaltung flexibler Software-Architekturen für Präsentationsschichten, dargestellt anhand episodenbasierter, medizinischer Dokumentation unter Einbeziehung topologischer Befundung*. Diplomarbeit, Technische Universität Ilmenau, Januar 2003.

[Con02] Connolly, Thomas, Carolyn Begg: *Database Systems – A practical approach to design, implementation and management*. Addison-Wesley, Harlow, England [u.a.], 3. Ausgabe Auflage, 2002.

[FB98] Frank Buschmann, et al.: *Patternorientierte Software-Architektur*. Addison-Wesley, Bonn [u.a.], 1998.

[Fow02] Fowler, Martin, et al.: *Pattern of Enterprise Application Architecure*. Addison-Wesley, 2002.

[FSF99] Free Software Foundation, Inc.: *GNU Licenses*. http://www.gnu.org/licenses/licenses.html, 1999.

[Gam96] Gamma, Erich, Richard Helm Ralph Johnson und John Vlissides: *Entwurfmuster – Elemente wiedeverwendbarer objektorientierter Software*. Addison-Wesley, München [u.a.], 1996.

[Gam02] Gamma, Erich: *Patterns For Beginners.* http://c2.com/cgi/wiki?PatternsForBeginners/, 2002.

[GNU02] GNU MED: *Formular Eingabemasken.* http://www.gnumed.org/screenshots.html, 2002.

[Ham97] Hamilton, Graham, Rick Cattell Maydene Fisher: *JDBC Database Access with Java.* Addison-Wesley, Reading, Mass. [u.a.], 1997.

[Han02] Han, Ki-Joon: *SQL3 Standardization.* Technischer Bericht, Kon-Kuk University, Department of Computer Engeneering, 2002.

[Har02] Harrer, Andreas: *Skript und Materialien zur Vorlesung: Muster in der Software-Technik.* http://wwwbruegge.in.tum.de/teaching/ss02/muster/, 2002.

[Hel02] Heller, Christian, Karsten Hilbert Roland Colberg und Peter Hahn: *Analysedokument zur Erstellung eines Informationssystems für den Einsatz in der Medizin.* 2002.

[jGu00] jGuru: *Fundamentals of RMI.* http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html, 2000.

[Klu98] Klute, Rainer: *JDBC in der Praxis.* Addison-Wesley-Longman, Bonn [u.a.], 1998.

[Kun02] Kunze, Torsten und Bohl, Jens: *Studienjahresarbeit: Redesign eines Formularservers unter Berücksichtigung der Performance der verwendeten Technologien.* Diplomarbeit, Technische Universität Ilmenau, Februar 2002.

[Meg02] Megginson, David, et al.: *About SAX.* http://www.saxproject.org/, 2002.

[OAS99] OASIS: *XML.ORG.* http://www.xml.org/, 1999.

[OMG02] OMG: *OMG - Technology adoption process.* http://www.omg.org, 2002.

[OSDN02] Open Source Development Network, Inc. (OSDN): *Sourceforge.net*. http://www.sourceforge.net, 2002.

[Pos02a] PostgreSQL: *Multiversion Concurrency Control*. http://www.postgresql.org/idocs/index.php?mvcc.html, 2002.

[Pos02b] PostgreSQL: *PostgreSQL Homepage*. http://www.postgresql.org, 2002.

[Sei02] Seiner, Robert S.: *Is SQL a real standard anymore?* http://www.tdan.com/i016hy01.htm, 2002.

[SUN95] SUN Microsystems: *RMI Tutorial*. http://java.sun.com/docs/books/tutorial/rmi/index.html, 1995.

[Vin02] Vinoski, Steve: *New features for CORBA 3.0*. Technischer Bericht, 2002.

[W3C02a] W3C: *World Wide Web Consortium*. http://www.w3c.org/, 2002.

[W3C02b] W3C, World Wide Web Consortium: *Document Object Model (DOM)*. http://www.w3.org/DOM/, 2002.

Abkürzungsverzeichnis

API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DBMS	Datenbankmanagementsystem
DOM	Document Object Model
DDL	Data Definition Language
DML	Data Manipulation Language
DTO	Data Transfer Object
EHR	Electronic Health Record
ERD	Entity Relationship Diagram
GNU	GNU is not Unix
GNU GPL	GNU General Public License
GNU FDL	GNU Free Documentation License
GUI	Graphical User Interface
HMVC	Hierarchical Model View Controller
HTML	Hypertext Markup Language
JDBC	Java Database Connectivity
JMS	Java Message Service
MVC	Model View Controller
ODBC	Open Database Connectivity
OID	Object Identifier
RMI	Remote Methode Invocation
SAX	Simple API for XML
SQL	Structured Query Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Thesen

- JDBC bietet eine gute Möglichkeit für den Zugriff auf Datenbanken innerhalb von Java-Programmen und gewinnt zunehmend an Bedeutung.

- SQL ist ein Standard, der in seinen Kernfunktionen eine gute Interoperabilität mit Datenbanken liefert, aber aufgrund häufig auftretender Verwendung proprietärer Operationen eine sehr komplexe Portierarbeit nach sich zieht.

- Die Extensible Markup Language (XML) als weltweit anerkannter Standard wird eine führende Rolle beim Speichern und Austauschen von Daten über Netzwerke einnehmen. Sie bietet eine robuste Möglichkeit, die Daten auszutauschen und auf Veränderungen in Server- oder Clientstruktur zu reagieren, ohne mit einen vollständigen Datenverlust rechnen zu müssen, im Gegensatz zu bekannten Objektserialisierungsverfahren.

- Der Datenaustausch in einem XML-Format - anstelle einer binären Serialisierung - bringt allerdings einen größeren Datenverkehr mit sich.

- Muster erlauben es, Software flexibler für Erweiterungen und robuster gegenüber Änderungen zu gestalten, beispielsweise bei Verwendung des Architekturmusters Model View Controller für die Entwicklung einer vernünftig strukturierten Präsentationsschicht.

- Das Anbieten eines einzelnen Persistenzmechanismus reicht für die heutigen Anforderungen an zuverlässige medizinische Software nicht aus.

- Die Verwendung des Data Mapper Musters hält Geschäftslogik und Persistenzmechanismen unabhängig voneinander und ermöglicht daher die Änderung eines der beiden, ohne die Notwendigkeit einer Modifikation des anderen zu verursachen.

- Mehrere verschiedene Persistenzmechanismen können durch eine Mappingschicht in einer Anwendung vereinigt werden und unabhängig voneinander operieren.

- Zusätzlich können in die selbe Schicht unterschiedliche Kommunikationsparadigmen eingebunden werden, um somit ebenfalls eine Transparenz und Flexibilität bei der Interprozesskommunikation zu gewährleisten.

- Die Anwendung der Strukturmuster Remote Facade und Data Transfer Object bei Interprozesskommunikation verbessert die Performance der Applikation.

- Drei- und N-Tier-Modelle erhöhen die Flexibilität von Softwaresystemen und vermeiden die Entstehung von Fat-Clients.

- Andererseits entlasten Fat-Clients den Server, da alle grundlegenden Operationen bereits im Client erfolgen. Dafür müssen aber bei Änderungen in der Struktur der Daten oder des Programmes auf jedem Client die selben Anpassungen vorgenommen werden.

Ilmenau, den 02. Januar 2003

Torsten Kunze

Eidesstattliche Erklärung

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

Ilmenau, den 02. Januar 2003

Torsten Kunze

GNU Free Documentation License

Version 1.2, November 2002

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.