Hochschule
Ravensburg-Weingarten
University of Applied Sciences

Diplomarbeit

Datenbankgestütztes Verwaltungssystem von Mathematikaufgaben mit Mathematica und Hibernate

Autor: Sorg, Michael
Matrikelnummer: 14003
Studiengang: Angewandte Informatik/Informationsnetze

1. Prüfer: Prof. Dr.rer.nat. Martin Hulin
2. Prüfer: Prof. Dr.rer.nat. Wolfgang Ertel

Zeitraum: 01. Oktober 2005 - 31. Januar 2006

Erklärung

Hiermit erkläre ich, dass ich diese Diplomarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel verwendet, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Ravensburg, den 30. Januar 2006,

(Michael Sorg)

Inhaltsverzeichnis

1 Einführung **1**
1.1 Vorwort . . . 1
1.2 Aufgabenstellung . . . 2
1.3 Vorüberlegungen . . . 4

2 Verwendete Technologie **6**
2.1 Java . . . 6
2.1.1 Sicherheitskonzept . . . 7
2.2 Hibernate . . . 8
2.2.1 ORM-Formen . . . 9
2.2.2 Features . . . 10
2.2.3 Grundlagen . . . 11
2.2.4 Objektzustände . . . 14
2.2.5 Objektidentität . . . 18
2.3 Datenbank-Management-System . . . 21
2.4 Mathematica . . . 23
2.4.1 Das Mathematica-Notebook-Format (*.nb) . . . 25
2.5 Webmathematica . . . 26
2.6 Eclipse . . . 27
2.7 JGoodies . . . 28
2.8 Betriebssystem . . . 29
2.9 Tools . . . 30
2.9.1 Apache Ant . . . 30
2.9.2 XDoclet . . . 30
2.9.3 hbm2ddl . . . 31
2.9.4 JUnit . . . 31
2.9.5 CVS . . . 32
2.9.6 Log4J . . . 33

3 Programmarchitektur **34**
3.1 Schichtenmodell . . . 34
3.2 Persistenzschicht . . . 36
3.3 Design-Patterns . . . 41
3.3.1 MVC-Pattern . . . 41
3.3.2 Singleton-Pattern . . . 43
3.4 Das Komponentenmodell . . . 44
3.5 Die Komponenten . . . 46
3.5.1 Model . . . 48
3.5.2 View . . . 50

3.5.3 Controller 51
3.5.4 Aufgabe erstellen 52
3.5.5 Aufgabe bearbeiten 55
3.5.6 Aufgabenblatt erstellen 58
3.5.7 Aufgabenblatt bearbeiten 60
3.6 Die Tool-Klassen 62
3.6.1 Die Klasse HibernateUtil 63
3.6.2 Die Klasse NotebookParser.java 64
3.7 Konfigurationsdateien 68
3.7.1 Die Datei configuration.properties 70
3.7.2 Die Datei user.properties 71
3.7.3 Konfiguration von Hibernate 72
3.7.4 Konfiguration von Log4J 73
3.8 Algorithmen 74
3.8.1 Traversieren von Bäumen 74
3.8.2 Suchen und ersetzen von Zeichenketten 76
3.9 GUI-Design 77
3.10 Sicherheit 79

4 Erweiterungen und Verbesserungen 80
4.1 Bekannte Bugs und Unzulänglichkeiten 80
4.1.1 Probleme bei Integration in MS-Windows 81
4.2 Erweiterungsvorschläge 82
4.2.1 Administrationstool 82
4.2.2 Automatisches Einfügen von Querverweisen 83

5 Programminstallation 85
5.1 Systemvoraussetzungen 85
5.2 Inhalt der CD 85
5.3 Importieren des Datenbankschemas 86
5.4 Installation der Anwender-Version 87

6 Informationen für Entwickler 89
6.1 Integration in Eclipse 89

7 Schlussbemerkungen 91

Begriffe 92

Abbildungsverzeichnis 93

Listings 94

Tabellenverzeichnis **95**

Literaturverzeichnis **100**

1 Einführung

1.1 Vorwort

Als ich begann, mich um ein Thema für meine Diplomarbeit zu bemühen, stellte sich mir zuerst die Frage, für welche Institution ich die Diplomarbeit erstellen möchte. Einerseits bestand die Möglichkeit, diese für eine Firma zu schreiben, was insbesondere aus finanziellen Gesichtspunkten interessant gewesen wäre, andererseits werden von Professoren der Fachhochschule eine Fülle unterschiedlichster Themen für Diplomarbeiten ausgeschrieben. Ich habe mich frühzeitig dafür entschieden, meine Diplomarbeit an der Fachhochschule zu schreiben, unter der Voraussetzung, dass ich ein Thema finden würde, bei dem ich meine persönlichen Studienschwerpunkte umsetzen kann. Für diese Entscheidung gab es zwei Gründe:

- Einige Kommilitonen, die Ihre Diplomarbeiten in einem Unternehmen erstellt hatten, schilderten mir, dass sie sich des Öfteren im Tagesgeschäft der Firma wiedergefunden hätten und sich teilweise über einen Zeitraum von mehreren Wochen nicht mit ihrer Diplomarbeit beschäftigen konnten. Aus Sicht eines Unternehmens ist dies sicherlich verständlich, aus der eines Diplomanden jedoch nicht erstrebenswert.

- Ich versprach mir von einer Diplomarbeit an der Hochschule eine experimentellere Vorgehensweise. Dies bezieht sich vor allem auf den Einsatz von Technologien, auf die ich durch Bücher, Zeitschriften, das Internet, sowie Gesprächen mit Kommilitonen aufmerksam wurde, die ich jedoch bislang noch nicht verwendet hatte.

Die Suche nach einem geeigneten Thema orientierte sich an mehreren Gesichtspunkten. Es sollte vorwiegend Softwareentwicklung beinhalten, bevorzugt in *Java*, außerdem sollte es sich bei der zu entwickelnden Applikation um ein Programm mit Datenbankanbindung handeln. Neben meiner persönlichen Vorliebe für *Java* und Datenbanken, sind dies Kerninhalte, die mich auch an meiner zukünftigen Arbeitsstelle, der Firma *innoSysTec* in Salem, bei der ich ab 1. Februar dieses Jahres als Softwareentwickler eingestellt bin, erwarten.
Aufgrund des Datenbankeinsatzes informierte ich mich zuerst über Themen, die von Herrn Professor Dr. Hulin ausgeschrieben waren und wurde auch sehr schnell fündig. Es handelte sich um ein Programm zur Verwaltung von Übungsaufgaben mit Integration von *Mathematica*. Da mir die zu verwendende Technologie weitgehend freigestellt war, habe ich dieses Thema ausgewählt.

1.2 Aufgabenstellung

Es ist das Ziel der vorliegenden Diplomarbeit, ein Programm zu entwickeln, welches die Dozenten der Fachhochschule bei der Verwaltung ihrer Übungsaufgaben und Aufgabenblätter unterstützt. Bislang müssen die dafür notwendigen Arbeitsschritte, weitgehend manuell erledigt werden.

Ein Dozent hat seine Aufgaben und Aufgabenblätter, beispielsweise nach Vorlesung, Fachgebiet und Semester sortiert, auf seiner Festplatte gespeichert. Möchte er nun etwa eine Aufgabe eines Blattes austauschen, weil sich eventuell im aktuellen Semester die Inhalte der Vorlesung verändert haben, muss er folgende Schritte erledigen:

- Das zu ändernde Aufgabenblatt auf seiner Festplatte suchen.
- Die neue Aufgabe erstellen (oder im Dateisystem suchen, falls sie bereits existiert).
- Sein Textverarbeitungsprogramm starten und die alte Aufgabe entfernen.
- Die neue Aufgabe mittels *copy-paste* in das Aufgabenblatt einfügen.
- Eventuell (ebenfalls manuell) die Aufgabenreihenfolge im Blatt ändern, falls er dies wünscht.
- Darauf achten, dass er beim Speichern das ursprüngliche Blatt nicht überschreibt (denn es sollte ja archiviert bleiben).
- Darauf achten, dass er das neue Blatt korrekt benennt und am richtigen Ort abspeichert.

Da dieser Ablauf einerseits sehr zeitaufwändig, andererseits aber auch fehleranfällig ist, kam Herr Professor Dr. Hulin auf die Idee, die Entwicklung einer Aufgabenverwaltung als Thema für eine Diplomarbeit auszuschreiben. Das Programm sollte über folgende Features verfügen:

- Übungsaufgaben, zugehörige Musterlösungen und Metainformationen (Aufgabentitel, Schlüsselworte, Themengebiet) sind nicht mehr im Dateisystem, sondern in einer Datenbank gespeichert.

- Das Programm kann das *Mathematica-Notebook-Format (*.nb)* verarbeiten.
- Der Dozent erstellt eine Aufgabe mit *Mathematica*, das Programm parst die *Notebook*-Datei (extrahieren des eigentlichen Textes aus dem Dokument und entfernen von Kommentaren) und legt die Aufgabe in der Datenbank ab.
- Er kann anhand von Schlüsselworten nach einer Aufgabe suchen und diese bearbeiten, beispielsweise Metadaten oder Aufgaben– beziehungsweise Lösungstext ändern. Er kann die Aufgabe in geänderter Form oder auch als neue Aufgabe abspeichern.
- Er kann ein neues Aufgabenblatt erstellen, dieses ebenfalls mit Metadaten, wie Studiengang, Vorlesung und Semester versehen und in der Datenbank ablegen.
- Dem Aufgabenblatt können Aufgaben hinzugefügt werden, die anhand der definierten Schlüsselworte gesucht werden können.
- Der Dozent kann mit Volltextsuche nach Aufgabenblättern suchen, diese bearbeiten (Metadaten ändern, Aufgaben hinzufügen oder entfernen, Aufgabenreihenfolge verändern,. . .) und sowohl das Blatt in geänderter Form, als auch als neues Aufgabenblatt speichern.
- Aufgaben und Aufgabenblätter können direkt aus der Anwendung mit *Mathematica* geöffnet werden.
- Der Dozent kann wählen, ob bei Aufgabenblättern nur der Aufgabentext angezeigt werden soll, oder auch die Musterlösungen.
- Die Aufgaben sollen mit dem *Mathematica-Notebook-Frontend* (oder einem Frontend, das die gleiche Funktionalität besitzt) erstellt werden können.
- Das Programm sollte mittels der von *Mathematica* zur Verfügung gestellten (*C*- oder *Java*-) API mit *Mathematica* kommunizieren.

In Abbildung 1 sind die erforderlichen Features als Use-Case-Diagramm dargestellt.

Keine Vorgaben erhielt ich bezüglich Art der Applikation (Web- oder Standalone-Anwendung) und dem zu verwendenden Datenbank-Management-System.

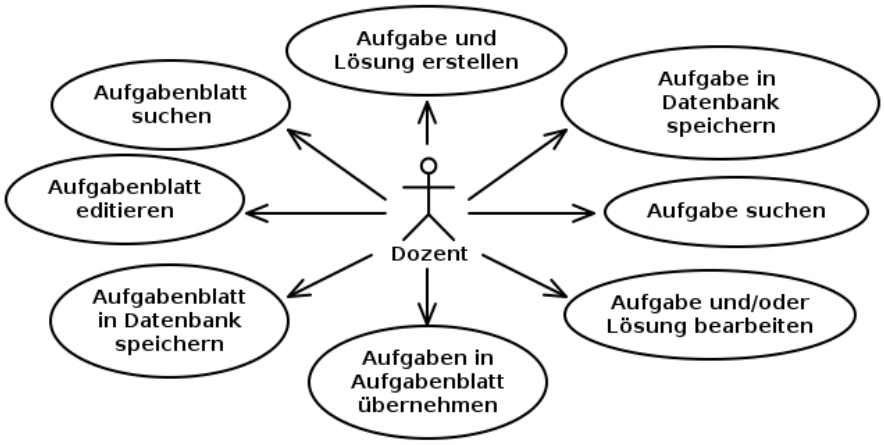

Abbildung 1: Use-Cases für die Aufgabenverwaltung

1.3 Vorüberlegungen

Da ich mich schon von vornherein dazu entschlossen hatte, die Anwendung in *Java* zu programmieren, bestand mein erster Schritt darin, die *Java*-API von *Mathematica* zu erkunden, da alle weiteren Überlegungen (Art der Applikation, Architektur der Applikation) von den Möglichkeiten dieser API abhingen. Es stellte sich bei diesen Recherchen heraus, dass es wohl nicht möglich sein würde, alle geforderten Features in die Software zu integrieren. Zwei Anforderungen schlossen sich gegenseitig aus:
Eine Kommunikation mit dem *Mathematica-Kernel* über die API ist zwar möglich, jedoch fand ich keinen Weg, mit dem *Notebook*-Frontend zu kommunizieren. Ich werde unter Kapitel 2.4 noch genauer auf diese Problematik eingehen.
Um die Funktionalität des Programms bestmöglich zu erhalten, überlegte ich mir zwei Lösungsmöglichkeiten:

- Eine Aufgabe wird mit dem *Notebook*-Frontend erstellt und auf dem lokalen Dateisystem abgespeichert. Die Aufgabenverwaltung bietet einen Dateiauswahl-Dialog an, mit dem die erstellte Aufgabe importiert werden kann. *Mathematica* kann hierzu direkt aus der Anwendung gestartet werden, auch das Anzeigen von bereits erstellten Aufgaben und Aufgabenblättern ist auf diese Weise möglich.

- Die *Java*-API wird wie geplant verwendet, jedoch nicht mit der Funktionalität des *Notebook*-Frontends. In der API ist ein Frontend enthalten, welches sich ähnlich bedienen lässt wie die *Mathemati-*

ca-Konsolenanwendung. Auf die erweiterten Features des *Notebook*-Frontends müsste dann allerdings verzichtet werden.

Eine dritte Möglichkeit habe ich gar nicht erst in Betracht gezogen, weil sie im Rahmen einer Diplomarbeit doch sehr unrealistisch gewesen wäre:
Diese hätte darin bestanden, die Funktionalität des *Notebook*-Frontends als *Java*-Programm „nachzubauen“ und damit auf den *Mathematica-Kernel* zuzugreifen. Der Zeitrahmen von 4 Monaten wäre dann aber sicherlich gesprengt worden.

Eine letzte Variante wäre gewesen, einen umgekehrten Ansatz zu wählen, also *Mathematica* als Basis zu verwenden und mittels der API den Funktionsumfang von *Mathematica* um eine Aufgabenverwaltung zu erweitern. Für diesen Weg konnte ich mich allerdings wenig begeistern, denn da die *Java*-API *(JLink)* ein direktes Abbild der *C*-API *(MathLink)* ist, die gleichen Funktionen und auch den gleichen Aufbau besitzt, ist sie in puncto Design nicht sehr objektorientiert gestaltet. Ich möchte dies jedoch nicht negativ verstanden wissen, denn wie ich in Kapitel 2.4 noch näher beschreiben werde, macht es durchaus Sinn, die API eher prozedural zu programmieren. Wäre diese Vorgehensweise allerdings Grundvoraussetzung gewesen, hätte ich mich auch sicherlich nicht für dieses Thema entschieden, da es letztendlich mein Ziel war, durch diese Diplomarbeit meine Kentnisse in objektorientierter Programmierung zu erweitern. Als ich die Problematik mit Herrn Professor Dr. Hulin erörterte, bat ich ihn deshalb, mich von dieser Möglichkeit zu entbinden.

Er entschied sich dafür, dass ich die erste Variante wählen solle, da er den zusätzlichen Aufwand, der durch das Zwischenspeichern der Aufgaben entsteht, für akzeptabel hielt.

Vor dem eigentlichen Entwicklungsprozess stand nun noch die Frage, ob ich das Programm als Webandwendung oder als Standalone-Applikation realisiere. Aus Gründen, die ich unter 2.1.1 näher beschreiben werde, entschied ich mich für eine Standalone-*Java*-Applikation.

2 Verwendete Technologie

2.1 Java

Java wurde Anfang der neunziger Jahre im Auftrag des US-amerikanischen Computerherstellers *Sun Microsystems* entwickelt. Ziel war einerseits, eine Programmiersprache auf Basis von C^{++} zu entwickeln, die das objektorientierte Paradigma, wie unter [RBP+90] beschrieben, konsequent umsetzt, sowie andererseits eine Laufzeitumgebung zu schaffen, die die Ausführung eines *Java*-Programms auf unterschiedlichsten Plattformen ermöglicht. Weiterführende Informationen zur Geschichte von *Java* sind unter [Dum] zu finden.

Trotz der sprachlichen Verwandtschaft mit C^{++}, gibt es einige konzeptionelle Unterschiede. So bietet *Java* beispielsweise nicht die Möglichkeit, eine Klasse von mehreren Klassen abzuleiten. Diese Tatsache hatte Auswirkungen bei der Entwicklung der Aufgabenverwaltung, auf die ich in Kapitel 3.5 noch näher eingehen werde.

Obwohl die Umsetzung eines Programms in Maschinencode bei *Java* zur Laufzeit erfolgt, kann *Java* nur bedingt als interpretierte Sprache bezeichnet werden, da vom *Java*-Compiler *javac* aus dem Quellcode plattformunabhängiger, sogenannter *Byte-Code* generiert wird, welcher bei Ausführung vom *JustInTimeCompiler (JIT)* in Maschinencode gewandelt wird. Dieser Vorgang , der in [Ull05] umfassend beschrieben wird, bewirkt eine deutliche Performance-Steigerung im Vergleich zu reinen Interpretersprachen, wie beispielsweise *Perl*.

Dennoch ist es verständlich, dass ein natives Maschinenprogramm letztendlich immer schneller laufen wird (vorausgesetzt, es ist adäquat programmiert) da der Zwischenschritt der Laufzeit-Kompilierung entfällt. Allerdings setzt mittlerweile auch *Microsoft* mit der *.NET*-Plattform auf ein ähnliches Konzept (siehe [Cha02]) und es bleibt abzuwarten, welchen Marktanteil native Anwendungsprogramme in der Zukunft einnehmen werden, da sich, aufgrund der stetigen Weiterentwicklung im Bereich der Computer-Hardware, der Geschwindigkeitsverlust weniger bemerkbar macht. Wie in [Shu05] zu sehen ist, ist der Performance-Unterschied zwischen *Java* und *.NET* nicht sehr groß.

War das Einsatzgebiet von *Java* bis Ende der neunziger Jahre noch hauptsächlich die Entwicklung von *Applets* für das Internet, hat sich *Java*

in den letzten Jahren aus diesen Gründen zunehmend auch als Plattform für Server- und Anwendungsprogramme etabliert. In meiner zukünftigen Firma (bei der ich auch mein zweites Praxissemester abgeleistet habe) kommt sowohl server- als auch clientseitig vorwiegend *Java*-Technologie zum Einsatz. Wie ich unter 2.7 beschreiben werde, ist auch der Aspekt der „gewöhnungsbedürftigen“ Optik von *Java*-GUI-Programmen so nicht mehr gültig.

Java ist mittlerweile in der Version 5.0 verfügbar, welche ich auch verwende. Da diese Version jedoch nicht abwärtskompatibel ist, ist es notwendig, die Version 5.0 zu installieren.

2.1.1 Sicherheitskonzept

Das Sicherheitskonzept von *Java* ist geprägt von der ursprünglichen Hauptanwendung, der Programmierung von *Applets* für das Internet. Es sollte verhindert werden, das geladener Code, der von einem unbekannten Server stammt, Schaden auf dem lokalen Rechner verursachen kann. Dies wird dadurch realisiert, dass diese Programme in einem abgeschlossenen Bereich, der sogenannten *Sandbox* ablaufen, die den Zugriff auf das eigentliche System verhindert. Da, aufgrund der Plattformunabhängigkeit, nur minimale Betriebssystemunterstützung vorausgesetzt werden kann und Betriebssysteme üblicherweise den Speicher nur vor Zugriffen unterschiedlicher Prozesse schützen, ist dieser Speicherschutz auf Sprachebene implementiert. Dies ist beispielsweise ein Grund, warum *Java* auf Zeigerarithmetik verzichtet (siehe [Pil]).

Im Serverbereich gilt das gleiche Prinzip. Es soll nicht möglich sein, dass auf Ressourcen des Servers zugegriffen werden kann, die sich außerhalb des geschützten Bereichs befinden. Dies gilt auch für den Zugriff auf das Dateisystem des Servers. So ist ein Datei-Upload nur in dafür spezifizierte Verzeichnisse möglich, die explizit deklariert werden müssen und es wird auch verhindert, dass Dateien aus diesen Verzeichnissen ausgeführt werden können (sollte jemand bösartigen Script-Code hochladen).

Aus diesen Gründen wäre eine Realisierung des Programms als Webandwendung problematisch gewesen, da es zwar möglich ist, *Mathematica* auf einer entfernten Maschine zu starten, aber nicht aus der Sandbox eines Application-Servers. Es gäbe zwar Vorgehensweisen, dies dennoch zu ermöglichen, was jedoch bedeuten würde, das Sicherheitskonzept von *Java* zu umgehen und potentiellen Angreifern die Gelegenheit zu geben,

den Server zu kompromittieren. In [Kum03] ist das Sicherheitskonzept, speziell in Bezug auf Serversysteme ausführlich beschrieben und es wird auch ausdrücklich davor gewarnt, dieses Konzept zu verletzen.

Auf einer lokalen Maschine ist der Zugriff auf Systemressourcen und Dateisystem selbstverständlich nicht beschränkt. Deshalb ist es möglich, *Mathematica* aus meinem Programm heraus zu starten.

2.2 Hibernate

Hibernate ist ein Framework, das objektorientierten Zugriff auf die Daten einer Datenbank bietet. Dieser Vorgang wird auch als *Object-Relational-Mapping (ORM)* bezeichnet.

> *„object-relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database“* [BK05]

Das *Hibernate*-Projekt wurde 2001 von Gavin King gegründet und bis zum Jahre 2003 als nicht-kommerzielles Open-Source-Projekt geführt (siehe [BK05]). Seit diesem Zeitpunkt ist das *Hibernate*-Team, unter kommerziellem Aspekt, Teil der *JBoss-Corporation*. *Hibernate* ist jedoch weiterhin kostenlos erhältlich und kann auch jederzeit kostenlos in eigenen (auch komerziellen) Projekten verwendet werden. *Hibernate* steht unter der *Lesser GNU Public License – LGPL* ([gnub]).

Hibernate wurde ursprünglich ausschließlich für die *Java*-Plattform entwickelt, mittlerweile existiert jedoch auch eine Portierung für *.NET* unter dem Namen *NHibernate*. *NHibernate* befindet sich allerdings noch in einem frühen Entwicklungsstadium und ist derzeit in der Version 1.0.2 erhältlich, *Hibernate* bereits in Version 3.1 [HTb]

Während meines zweiten Praxissemesters wurde in der Firma die Entscheidung getroffen, bei neuen Projekten als Persistenzschicht *Hibernate* einzusetzen. Zuvor wurde hierfür *BC4J* von *Oracle* verwendet, jedoch beschloss die Geschäftsführung aus strategischen Gründen, künftig vermehrt auf offene Standards und Open-Source-Lösungen zu setzen. Die Einführung verlief jedoch alles andere als reibungslos. Zwar ging die Portierung der einzelnen Komponenten von *BC4J* nach *Hibernate* relativ zügig, es stellte sich auch heraus, dass sich der Umfang des Quellcodes bei manchen Klassen drastisch verringerte, jedoch traten bei den anschließenden Tests spontan

scheinbar unerklärliche Fehler und Inkonsistenzen in den Datenbankeinträgen auf.

Da sich dieser Zustand bis zum Ende meines Praktikums nur unwesentlich änderte, beschloss ich, mich intensiver mit *Hibernate* zu befassen und nach den Ursachen dieser Fehlfunktionen zu suchen. Mittlerweile habe ich erkannt, dass ein genaues Verständnis der Funktionsweise von *Hibernate* notwendig ist, um es effektiv einsetzen zu können. Unter 2.2.4 und 2.2.5 werde ich auf einige wichtige Aspekte der Funktionsweise von *Hibernate* eingehen, die letztendlich zur Lösung meiner anfänglichen Probleme beitrugen.

Da ich mich in den letzten Monaten intensiv mit *Hibernate* beschäftigt hatte und auch bereits mein Informatikseminar über dieses Thema hielt, war es naheliegend, *Hibernate* auch in meiner Diplomarbeit zu verwenden. Der Einwand, dass ein Projekt diesen Umfangs, mit geringer Anzahl an Datenbanktabellen und Abfragen, den Einsatz eines solchen Frameworks nicht notwendig gemacht hätte, ist sicherlich berechtigt. Letztendlich hat die Integration von *Hibernate* und vor allem die anfänglichen Probleme, mehrere Wochen Zeit in Anspruch genommen, die ich mir bei einer rein *SQL/JDBC*-basierten Lösung hätte ersparen können. Ich sah es jedoch als Chance für mich, im Rahmen eines überschaubaren Projekts, eine Technologie zu erlernen, von der ich bereits zu Beginn meiner Diplomarbeit wusste, dass ich sie in meiner zukünftigen beruflichen Tätigkeit beherrschen muss.

2.2.1 ORM-Formen

Es gibt verschiedenste Produkte, von ORM-Lösungen, die sich in puncto Design und Funktionsumfang erheblich unterscheiden.
Auf [Fus] sind verschiedene Ausprägungen von objekt-relationalem Datenbankmapping beschrieben:

- *pure relational*
  Es wird direkt auf die *JDBC*-Schicht zugegriffen. Abfragen erfolgen mit *SQL*-Statements und es wird auch Geschäftslogik in die Datenbank verlagert (Stored Procedures). Dieses Vorgehen ist sehr performant, da die einzelnen *SQL*-Abfragen optimiert werden können. Insbesondere bei größeren Programmen ist jedoch die Übersicht und Wartbarkeit gefährdet.

- *light object mapping*
  Die einzelnen Entitäten sind, unter Verwendung bekannter Design-Patterns (z.B. *DAO* [Fed00]), als Klassen repräsentiert. (Selbstgeschriebene) *SQL*-Statements sind in Klassen, beziehungsweise Methoden gekapselt. Teilweise werden auch Stored Procedures verwendet.

- *medium object mapping*
  Die Anwendung verwendet ein Objektmodell. *SQL* wird beim Kompilieren von einem Code-Generation-Tool oder zur Laufzeit von Framework-Code generiert. Abfragen werden mit Hilfe einer objektorientierten Ausdruckssprache durchgeführt. Die einzelnen Objekte werden von einer Persistenzschicht zwischengespeichert, Stored Procedures werden eher selten verwendet. Bekannteste Vertreter dieser Variante sind *ADO.NET* von *Microsoft* und *BC4J* von *Oracle*.

- *full object mapping*
  Das Framework unterstützt alle Aspekte der Objektorientierung (Vererbung, Polymorphie). Es verfügt über effiziente Strategien zur Datenbereitstellung, die transparent implementiert sind (beispielsweise *lazy-* oder *eager-fetching* [HTa]). Objekte der Persistenzklasse können *persistent* gemacht werden, können aber darüber hinaus wie alle anderen Objekte verwendet werden (es ist kein spezieller Container notwendig, wie z.B. bei EJB). Transaktionen können transparent gesteuert werden. Zu dieser Kategorie zählt *Hibernate*.

2.2.2 Features

Hibernate verfügt über alle Features einer *Full-Object-Mapping*-Lösung (2.2.1). Weiterhin ist erwähnenswert:

- *Hibernate* verwendet konsequent offene Standards. So sind beispielsweise alle Deskriptor- und Konfigurationsdateien *XML*-basiert.

- Die Persistenzklassen sind normale *Java*-Klassen, sogenannte *POJO's (Plain Ordinary Java Objects)*, also *Java*-Klassen, die neben einem parameterlosen Konstruktor, *get()*- und *set()*-Methoden für jede Property besitzen müssen.

- *Hibernate* kann sowohl für Standalone-, als auch für Webanwendungen verwendet werden.

- *Hibernate* wird sehr aktiv weiterentwickelt.

- *Hibernate* verfügt über zusätzliche Tools, um bestimmte Aufgaben zu erleichtern, beispielsweise das Generieren des Datenbankschemas anhand der Deskriptor-Dateien.

- *Hibernate* unterstützt eine Vielzahl von *SQL*-Dialekten, die die Anbindung nahezu jeder gängigen Datenbank ermöglichen. Es müssen lediglich die Angaben zu *JDBC*-Treiber und *SQL*-Dialekt in der Konfigurationsdatei geändert werden. Anschließendes Neukompilieren ist nicht erforderlich.

Details zu den Features von *Hibernate* finden sich unter [BK05] und [HTa].

2.2.3 Grundlagen

Bei *Hibernate* werden Datenbanktabellen auf *Java*-Klassen abgebildet. Jede *Java*-Klasse verfügt über eine korrospondierende Mapping-Datei, welche vom *Hibernate*-Framework zur Laufzeit ausgewertet wird. In Listing 1 ist eine sehr rudimentäre Mapping-Datei dargestellt:

```
1 <hibernate−mapping>
2     <class name="MyClass" table="MYTABLE">
3         <id name="id" column="ID">
4             <generator class="native"/>
5         </id>
6         <property name="name" type="string" column="NAME"/>
7         <property name="age" type="integer"column="AGE" />
8     </class>
```

Listing 1: Einfaches Mapping

In Zeile 2 wird festgelegt, dass die Datenbank-Tabelle *MYTABLE* auf die *Java*-Klasse *MyClass* abgebildet wird. In den Zeilen 3 und 4 wird der Primary-Key spezifiziert, welcher verpflichtend ist. Das Attribut *generator class ="native"*, sagt aus, dass der Primary-Key von der Datenbank generiert wird.
In den Zeilen 6 und 7 wird schließlich zuerst eine Property namens *name* mit Datentyp *String*, anschließend eine Property *age* mit Datentyp *Integer*, die auf die Tabellenspalten *NAME* und *AGE* gemappt werden, deklariert.

Die entsprechende *Java*-Klasse (Listing 2), muss, wie in 2.2.2 bereits erwähnt, als *POJO*-Klasse implementiert werden, benötigt also für jede Property *get()*- und *set()*-Methoden, auch der geforderte parameterlose Konstruktor ist vorhanden.

Selbstverständlich sind alle Arten von Relationen zu anderen Datenbanktabellen möglich. Im Falle einer *1:n*-Beziehung zwischen Tabelle *a* und Tabelle *b* wird in der Persistenz-Klasse *a* eine *Java*-Collection mit Objekten der Klasse *b* deklariert. Bei *n:m*-Relationen hält jede der beteiligten Klassen eine Collection der jeweils anderen. In der Datenbank werden die Relationen nach altbekanntem Muster angelegt, bei einer *n:m* Relation ist also weiterhin eine Verknüpfungstabelle notwendig, die jedoch keine korrospondierende *Java*-Klasse (und auch keine Mapping-Datei) besitzt.

```
1  public class MyClass{
2      private Long id;
3      private String name;
4      private Integer age;
5      //Konstruktor
6      public MyClass(){}
7      //Get−Methoden
8      public Long getId() {
9          return this.id;
10     }
11     public String getName() {
12         return this.name;
13     }
14     public Integer getAge() {
15         return this.id;
16     }
17     //Set−Methoden
18     public void setId(long id) {
19         this.id = id;
20     }
21     public void setName(String name) {
22         this.name = name;
23     }
24     public void setAge(int age) {
25         this.age = new Integer(age.intValue());
26     }
27 }
```

Listing 2: Einfache Persistenzklasse

Die Abbildung von Datenbanktabelle auf *Java*-Klasse muss nicht *1:1* erfolgen. *Hibernate* unterstützt unterschiedliche Granularitäten. Es kann also sowohl eine Klasse auf mehrere Tabellen, als auch umgekehrt, gemappt werden. Es ist also auf beiden Seiten ein optimales Design möglich, allzu kompromissbehaftete Lösungen können vermieden werden.

In den Mapping-Dateien können noch weitere Parameter gesetzt werden. So ist es möglich, Assoziationen zu kaskadieren. Wird nun ein Objekt gespeichert, welches kaskadierte Objekte hält, werden diese automatisch ebenfalls gespeichert. Mit dieser Funktionalität sollte man jedoch vorsichtig umgehen. Wenn in allen Persistenzklassen *cascade=delete* eingestellt wird und der Objektgraph sehr komplex ist, kann der Aufruf von *delete()* sehr schnell einen Großteil der Datenbank löschen.
Auch Strategien zum Laden der assoziierten Collections können spezifiziert werden. Der Parameter *lazy=true* bewirkt, dass die Daten der entsprechenden Collection erst beim ersten Zugriff geladen werden. Dies wäre sinnvoll, wenn sich in der Collection große Datenmengen befinden, die jedoch nur selten benötigt werden. Eine vollständige Dokumentation aller möglichen Optionen finden sich in der Referenz-Dokumentation ([HTa]).

Es ist jedoch keineswegs notwendig, das Erstellen der drei beteiligten Komponenten, *Java*-Klasse, Mapping-Datei und Datenbanktabelle manuell zu erledigen. Durch Einsatz entsprechender Tools, muss nur eine dieser Komponenten erstellt werden. In [BK05] unterscheidet Gavin King folgende Vorgehensweisen:

- Top-Down
  Es gibt einerseits die Möglichkeit, sich aus der Deskriptor-Datei die *Java*-Klasse erstellen zu lassen (mittels *hbm2java*, ein Tool, welches zum *Hibernate-Toolset* gehört), andererseits kann auch aus der *Java*-Klasse das Mapping-File generiert werden, wozu beispielsweise *XDoclet* verwendet werden kann, welches ich unter 2.9.2 noch näher beschreiben werde, da ich in der Diplomarbeit diesen Weg gewählt habe. Anschließend wird mit *hbm2ddl*, ebenfalls aus dem *Hibernate-Toolset* (siehe 2.9.3), das Datenbankschema erstellt.

- Bottom-Up
  Grundlage ist das Datenbank-Layout, mit Hilfe von *Middlegen*, einem dritten Werkzeug aus dem *Hibernate-Toolset*, werden die Mapping-Dateien generiert, anschließend, wie bereits bei Variante eins beschrieben, mit *hbm2java* die entsprechenden *Java*-Klassen. Diese Vorgehensweise wurde in meinem Praktikumsbetrieb angewandt.

- Meet-In-The-Middle
  Es sind einerseits Teile des Datenbankschemas vorhanden, andererseits aber auch Mapping-Dateien/*Java*-Klassen. Gavin King schreibt in [BK05] über diesen Weg:

> *„The most difficult scenario combines existing Java classes and an existing relational schema. In this case, there is little that the Hibernate toolset can do to help. This is an incredibly painful scescenario that is, fortunately, exceedingly rare. “*

Dieses Vorgehen sollte also nur in Ausnahmefällen gewählt werden.

Abfragen können in *Hibernate* mittels drei verschiedener Techniken durchgeführt werden:

- *SQL-Statements*
  Die Verwendung „normaler“ *SQL*-Statements bringt allerdings einen großen Nachteile mit sich: Man verliert die Datenbankunabhängigkeit.

- *HQL – Hibernate Query Language*
  Eine objektorientierte Ausdruckssprache, deren Synthax an *SQL* angelehnt ist. *HQL* garantiert Datenbankunabhängigkeit.

- *HCA – Hibernate-Criteria-API*
  Eine objektorientierte API, deren Klassen Methoden für unterschiedlichste Abfragen kapselt. Vorteil der *Criteria-API* ist die Typsicherheit. Fehler, die sonst erst zur Laufzeit erkannt werden, führen hier sofort zu einem Compiler-Fehler, was bei umfangreicheren Anwendungen wertvolle Zeit sparen kann. Aus diesem Grund wurde die API auch in meinem Praxisbetrieb verwendet. Bei der Aufgabenverwaltung habe ich sie ebenfalls eingesetzt. Die *Hibernate-Criteria-API* garantiert ebenfalls Datenbankunabhänigkeit.

2.2.4 Objektzustände

Hibernate ist nicht das Kernthema dieser Diplomarbeit. Da jedoch die Einarbeitung in das Framework nicht ganz einfach und vor allem zeitaufwändig war, möchte ich an dieser Stelle etwas detaillierter auf die Funktionsweise von *Hibernate* eingehen. Das Verständnis der Problematiken, die in diesem Kapitel (sowie in 2.2.5) beschrieben sind, haben sich für mich als Grundvoraussetzungen für ein erfolgreiches Arbeiten mit *Hibernate* erwiesen.

Die entsprechenden Informationen habe ich aus [BK05], [HTa] und [Ell04] entnommen, einige Details jedoch erst durch Studium des Quellcodes erkannt.

Wie in Abbildung 2 dargestellt ist, kann ein Objekt einer Persistenzklasse bei *Hibernate* drei Zustände einnehmen, die Gavin King in [BK05] als

transient, *persistent* und *detached* bezeichnet.
Wird in einem Programm eine Instanz einer Persistenzklasse erzeugt, besitzt es den Zustand *transient* und ist solange ein ganz normales *Java-Objekt*, bis es durch Aufruf der Methoden *save()* oder *saveOrUpdate()* in den Zustand *persistent* übergeht, oder mit einem bereits *persistenten* Objekt assoziiert wird und dann ebenfalls *persistent* wird.

Ab diesem Zeitpunkt steht es unter Persistenz-Verwaltung der aktuellen Session, das heißt es wird auch dann eine Referenz auf das Objekt gehalten, wenn der Programmierer im Programm seine Referenz auf *null* setzt. Die *Java-Garbage-Collection* wird in diesem Fall nicht aktiv, da ja noch eine Referenz vorhanden ist.

Das Objekt wird solange verwaltet, bis die Session geschlossen wird (durch eine Anweisung im Programm, beziehungsweise der eingestellte Time-Out-Zeitraum ist abgelaufen) oder das Objekt mittels *delete()* wieder *transient* wird. Im Falle der geschlossenen Session geht das Objekt in einen dritten Zustand, *detached*, über.

Sowohl *transient*-, als auch *detached*-Objekte sind wieder für die Garbage-Collection verfügbar und der Speicher wird freigegeben, wenn der Programmierer die Referenz auf das Objekt löscht. Die Besonderheit an *detached* ist, dass man ein solches Objekt jederzeit mit *update()* wieder *persistent* machen kann, da *Hibernate* von *detached*-Objekten Kopien hält und somit weiss, dass ein Objekt ehemals *persistent* war.

Da die Persistenz-Verwaltung zyklisch den Inhalt *persistenter* Objekte mit der Datenbank synchronisiert, wäre es theoretisch, im Falle einer ausreichend langen Session-Zeit, möglich, ein Objekt ein einziges Mal *persistent* zu machen, es beliebig oft zu verändern, ohne auch nur einen Speicherbefehl an *Hibernate* abzusetzen, da dies die Persistenz-Verwaltung erledigen würde. Man könnte nun argumentieren, dass dies ein hervorragendes Feature wäre. Der Entwickler müsste sich nicht mehr darum kümmern, Daten in die Datenbank zu schreiben. Er würde jedes erzeugte Objekt einmalig *persistent* machen, die *Hibernate-Session* die Daten laufend synchronisieren und der Entwickler kaum mehr wahrnehmen, dass er überhaupt ein Programm mit Datenbankanbindung entwickelt.

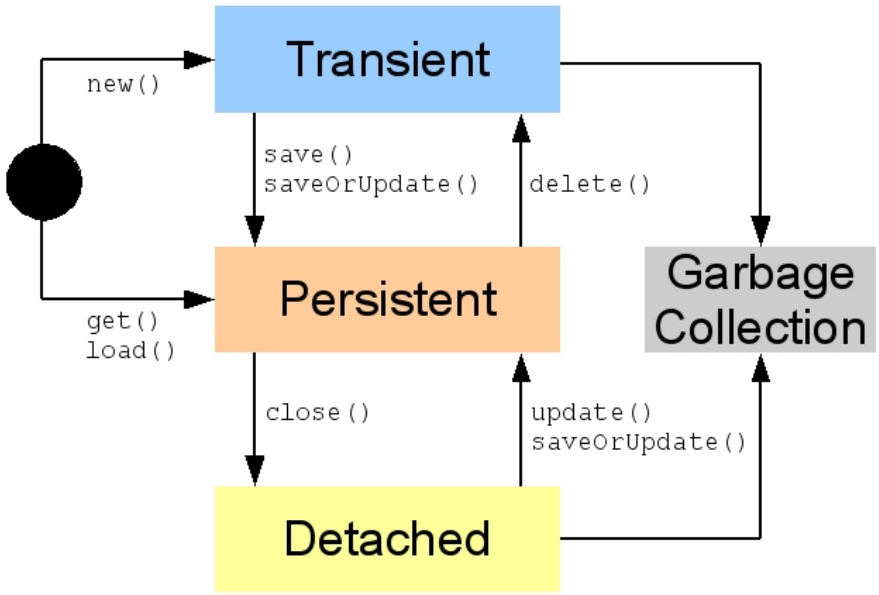

Abbildung 2: Objekzustände bei Hibernate

Dieser Wunschgedanke würde sich jedoch in der Praxis als fatal erweisen:

- Der Ansatz setzt voraus, dass man sich stets in der gleichen Session befindet. Man müsste also einen Time-Out von „unendlich“ einstellen, was jedoch unnötige Ressourcen binden würde, sollte der Anwender beispielsweise das Programm mehrere Tage laufen lassen. Die *Hibernate-Session* müsste in diesem Fall ständig eine Datenbankverbindung offen halten (beziehungsweise neu öffnen, wenn der Time-Out des Servers abläuft), um die gehaltenen Objekte mit der Datenbank abzugleichen, da es ja möglich wäre, dass von einem anderen Client Datenbankeinträge geändert werden.

- Ist der Time-Out begrenzt, wüsste der Entwickler nicht mehr, in welchem Status sich seine Objekte befinden. Es wäre beispielsweise möglich, dass eine Property eines Objekts, welche in der Datenbank *not null* deklariert ist, während des Programmablaufs für einige Zeit *null* ist, weil die Property erst später im Programmablauf neu gesetzt wird. Ist das Objekt *detached* funktioniert alles reibungslos. Ist es allerdings *persistent*, wird eine Exception ausgelöst. Der Entwickler könnte also seine Objekte nicht mehr wie normale *Java*-Objekte verwenden, da er ständig prüfen müsste, ob der aktuelle Zustand bestimmte Manipulationen erlaubt oder nicht.

- Daten würden außerhalb von Transaktionen in die Datenbank geschrieben. Das Prinzip der Transaktion, als Vorgang auf atomarer Ebene, mit der Möglichkeit eines anschließenden Rollbacks, würde ad absurdum geführt.

Außerdem ist die Annahme (von der auch ich anfangs ausging), dass das Schließen einer Session mit einem Trennen der *JDBC*-Verbindung gleichzusetzen ist, falsch:
Die *JDBC*-Verbindung wird von einem *Connection-Pool* verwaltet und bei Bedarf einer Session zugewiesen (siehe [BK05]). Sollte man also glauben, einen Performance-Gewinn zu erzielen, wenn man die Session offen hält, da dadurch Reconnects zur Datenbank entfallen, ist dies ein Irrtum. Das Gegenteil ist der Fall: Die Performance verschlechtert sich, da *Hibernate* bei geöffneter Session ständig mit der Datenbank kommuniziert.

Für Webandwendungen wird beispielsweise in der *Hibernate*-Referenz-Dokumentation ([HTa]) empfohlen, im Connection-Pool ständig eine gewisse Anzahl von *JDBC*-Verbindungen offen zu halten, die dann von den jeweiligen Sessions benützt werden. Bei einer Standalone-Anwendung kann man die Anzahl der Connections auf „1“ setzen, es sei denn, das Programm besteht aus mehreren Threads. Unter 3.6.1 werde ich jedoch eine Klasse beschreiben, die thread-safe eine *Hibernate-Session* verwaltet, es also ermöglicht, dass mehrere Threads die gleiche Session nützen können.

Aus diesen Sachverhalten ergibt sich die Konsequenz, dass man eine Session nur dann öffnen sollte, wenn man tatsächlich auf die Datenbank zugreift (lesend oder schreibend). Danach sollte die Session geschlossen werden.

Dementsprechend ist in Listing 3 zu sehen, wie sich Gavin King (in [BK05]) den Ablauf einer Transaktion vorstellt.

Übrigens erklärt das Wissen um Sessions und Objektzustände auch einige der spontan aufgetretenen und scheinbar nicht nachvollziehbaren Probleme, die sowohl in meinem Praktikumsbetrieb, als auch zu Beginn meiner Diplomarbeit aufgetreten sind.

```
public void save() {
    try {
        Session session = getSessionFactory().openSession();
        Transaction tx = session.beginTransaction();
        Message message = new Message("Hello World");
        // Objekt wird persistent gemacht
        session.save(message);
        tx.commit();
        // Transaktion erfolgreich, Session wird geschlossen
        session.close();
    } catch (HibernateExeption ex){
        if (tx != null) {
            // Transaktion nicht erfolgreich, Rollback
            // durchführen, anschließend Session schließen
            tx.rollback();
            ex.printStackTrace();
            session.close();
        }
    }
}
```

Listing 3: Transaktion

2.2.5 Objektidentität

Das Problem der Objektidentität ist der zweite große Stolperstein bei *Hibernate*. In der (relationalen) Datenbankwelt spricht man von *identischen* Datensätzen, wenn die Primärschlüssel den gleichen Wert haben. Bei *Java* hingegen, wird zwischen *Objektidentität* und *Objektgleichheit* unterschieden:

- Objektidentität
  Zwei Objekte a und b sind *identisch*, wenn gilt: $a == b$, es sich also bei a und b um die gleiche Instanz der gleichen Klasse handelt.

- Objektgleichheit
  Zwei Objekte a und b sind gleich, wenn gilt: $a.equals(b) == true$, was bei der Standardimplementierung der Klasse *Object* (welches ja die Superklasse aller *Java*-Klassen ist) bedeutet, dass a und b Objekte der gleichen Klasse sind und ihre Instanzvariablen die gleichen Werte enthalten.

Keine dieser beiden Definitionen stimmt mit der Definition der Datenbankidentität überein. Anpassungsprobleme dieser Art hat Gavin King (in [BK05]) unter dem Oberbegriff „*Object-Relational-Mismatch*“ zusammengefasst.

Hibernate verwaltet intern eigene Proxy-Objekte der Persistenzklassen (bei Objekten mit Status *persistent* und *detached*), die die entsprechenden Datensätze der Datenbank enthalten. Diese Objekte werden zur Laufzeit mittels *Reflection* erzeugt – also nicht durch Aufruf von *new(Klasse)*, sondern *Class.forName(Klassenname)*, da *Hibernate* die Namen der Klassen ja erst beim Systemstart, durch auslesen der Konfigurationsdatei bekannt sind.

Wenn man die Default-Implementierung von *equals()* in den Persistenzklassen *nicht* ändert, wird (über kurz oder lang) folgendes Szenario entstehen:

- Es wird ein Objekt *a* erzeugt und mittels *save() persistent* gemacht. *Hibernate* erzeugt ebenfalls intern ein Objekt *b*, das den gleichen Inhalt hat wie *b*, also wäre *a.equals(b) == true* und *Hibernate* weiss, dass es sich um das „gleiche“ Objekt handelt. Dies wird für die Dauer der aktuellen Session funktionieren, da ja alle Änderungen an *a* von *Hibernate* auf *b* übertragen werden. Erlangt *a* jedoch irgendwann den Status *detached* (weil die Session beendet wurde), wird anschließend verändert und mittels *update()* erneut persistiert, ist *a.equals(b) == false* und *Hibernate* löst eine Exception aus, da *Hibernate* das Objekt *a* nicht mehr kennt.

Eine Lösungsmöglichkeit wäre, die *equals()*-Methode an den Primärschlüssel zu binden (wie in Listing 4), dann würde jedoch folgendes passieren:

- Da es sinnvoll ist, den Primärschlüssel von der Datenbank setzen zu lassen (siehe [BK05] und [HTa]), hat ein neu erzeugtes Persistenzobjekt noch keinen Primärschlüssel. Wird das Objekt nun *persistent* gemacht, wird sich *Hibernate* zwar nicht beklagen, dass der Primärschlüssel *null* ist, da dies ja stets bei erstmalig persistierten Objekten der Fall ist. Vielmehr würde *Hibernate* sein Objekt *b* erzeugen und anschließend *a.equals(b) aufrufen*, was gleichbedeutend mit *null.equals(null)* ist, was zu einer *NullPointerException* führen würde.

```
public class PersistentClass{
    private Long primaryKey;
    public PersistentClass{}
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj!= null && obj instanceof PersistentClass) {
            PersistentClass pClass = (PersistentClass)obj;
            if (this.primaryKey == pClass.getPrimaryKey()) {
                return true;
            }
        }
        return false;
    }
}
```

Listing 4: equals() an Primärschlüssel gebunden

Um dieses Problem zu lösen, stieß ich auf zwei Lösungsansätze, die auch bei [HTa] beschrieben sind:

- Die *equals()*-Methode wird an bestimmte *not-null*-Properties gebunden, die sich nicht ändern dürfen und jeden Datensatz eindeutig kennzeichnen, beispielsweise den (eindeutigen) Benutzernamen. Diese Methode finde ich jedoch nicht sehr elegant. Was passiert, wenn der Benutzername doch irgendwann geändert werden soll oder eine Tabelle über keine eindeutigen Attribute (vom Primärschlüssel abgesehen) verfügt?

- Die *Hibernate*-Referenz-Dokumentation [HTa] empfiehlt das Hinzufügen eines weiteren Attributs in jede Persistenzklasse, welches als *Business-Key* bezeichnet wird. Bei der Erzeugung muss sichergestellt werden, dass dieses Attribut eindeutig ist.

Ich habe mich bei meiner Diplomarbeit für den Einsatz eines solchen *Business-Keys* entschieden.
Dieser wird in meinem Programm mit Hilfe der Klasse *Random* aus dem Paket *java.util* erstellt. Bei jeder Instantiierung eines Persistenzobjekts, wird gleichzeitig ein *Random*-Objekt erzeugt, welches als Seed die aktuelle Systemzeit in Millisekunden erhält. Die Instanz-Methode *nextRandom()* liefert dann ein 64 Bit Long-Object, das beim Persistenzobjekt als Business-Key gesetzt wird.
Die Methode *equals()* habe ich dann ähnlich wie in Listing 4 implementiert

und lediglich den *Primary-Key* durch den *Business-Key* ersetzt. Wichtig ist in diesem Zusammenhang, dass ebenfalls die Methode *hashCode()* überschrieben wird, da diese von *equals()* aufgerufen wird. Sie muss ebenfalls (wie in Listing 5), an den Business-Key gebunden werden:

```
1 public int hashCode() {
2         return new HashCodeBuilder(17, 37)
3             .append(this.businessKey)
4             .toHashCode();
5     }
```

Listing 5: hashCode() an Business-Key gebunden

2.3 Datenbank-Management-System

Wie bereits unter 2.2.2 beschrieben, ist *Hibernate* datenbankunabhängig und arbeitet mit einem Großteil der verfügbaren (relationalen) Datenbank-Management-Systemen zusammen. *Hibernate* realisiert dies mit verschiedenen *SQL*-Dialekten. In Tabelle 1 sind die aktuell unterstützten Datenbanken dargestellt (Quelle: [HTa]).

Da mir Herr Professor Dr. Hulin keine Vorgaben bezüglich des zu verwendenden DMBS gemacht hatte, konnte ich frei wählen. Da mir *Oracle* und *MySQL* bereits bekannt waren, entschied ich mich zuerst für *PostgreSQL*, da mir der Geschäftsführer meiner Praktikumsstelle erzählt hatte, dass – seiner Meinung nach – *PostgreSQL* künftig zum Konkurrenten für *Oracle* erwachsen könnte.

Leider stellte sich ein sehr triviales Problem ein:
Da ich nur über eine Einzelplatzlizenz für *Mathematica* verfüge, habe ich *Mathematica* auf meinem Notebook installiert, um auch mobil entwickeln zu können. Da mein Notebook nur über 512 MB RAM verfügt und bereits meine Entwicklungsumgebung sehr viel Arbeitsspeicher benötigt (siehe 2.6), hatte ich erhebliche Performance-Probleme, die ich bei Verwendung von *MySQL* nicht in diesem Ausmaß kannte. Nach einer Recherche im Internet erfuhr ich auf [Jan03], dass bei extrem kleinen Datenbanken, *MySQL* weniger Ressourcen benötigt als *PostgreSQL*, jedoch *PostgreSQL* besser skaliert, das heißt, dass sich bei steigender Datenbankgröße der Unterschied in puncto Ressourcenverbrauch ausgeglichen, beziehungsweise ins Gegenteil verkehrt hätte. Nachdem ich dann *MySQL* verwendete, wurde auch die Gesamtperformance besser.

Ich habe allerdings dann auf meinem Desktop-Rechner *PostgreSQL* installiert und diesen bei meinen abschließenden Tests (über Netzwerk) benutzt. Nachdem ich Treiber und Dialekt geändert hatte, lief das Programm problemlos.

Auf die Konfiguration des Datenbankservers werde ich unter 3.7 noch genauer eingehen.

Datenbank	Dialekt-Klasse
DB2	org.hibernate.dialect.DB2Dialect
DB2 AS/400	org.hibernate.dialect.DB2400Dialect
DB2 OS390	org.hibernate.dialect.DB2390Dialect
MySQL	org.hibernate.dialect.MySQLDialect
MySQL mit InnoDB	org.hibernate.dialect.MySQLInnoDBDialect
MySQL mit MyISAM	org.hibernate.dialect.MySQLMyISAMDialect
Oracle (alle)	org.hibernate.dialect.OracleDialect
Oracle 9i/10g	org.hibernate.dialect.Oracle9Dialect
Sybase	org.hibernate.dialect.SybaseDialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect
Microsoft SQL Server	org.hibernate.dialect.SQLServerDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
Informix	org.hibernate.dialect.InformixDialect
HypersonicSQL	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
Ingres	org.hibernate.dialect.IngresDialect
Progress	org.hibernate.dialect.ProgressDialect
Mckoi SQL	org.hibernate.dialect.MckoiDialect
Interbase	org.hibernate.dialect.InterbaseDialect
Pointbase	org.hibernate.dialect.PointbaseDialect
Firebird	org.hibernate.dialect.FirebirdDialect

Tabelle 1: Unterstützte Datenbanken

2.4 Mathematica

> *„Mathematica ist ein kommerzielles Softwarepaket der Firma Wolfram Research. Es ist eines der meistbenutzten mathematisch-naturwissenschaftlichen Programmpakete. Enthalten sind:*
>
> - *ein Computer-Algebra-System zur symbolischen Verarbeitung von Gleichungen*
> - *eine Numerik-Software zum numerischen Lösen oder Auswerten von Gleichungen*
> - *ein Visualisierungs-Tool zum Darstellen von Graphen*
> - *eine Programmiersprache, die Elemente des prozeduralen, objektorientierten und regelbasierten Programmierens in sich vereint“*
>
> (Quelle: wikipedia.de [de.])

Mathematica verfügt jedoch außerdem über ein sehr interessantes Feature, das eigentlich bei der Diplomarbeit zum Einsatz kommen sollte:
Wie bereits unter 1.3 kurz erwähnt, bietet es Schnittstellen an, die sowohl die Anbindung externer Programme an *Mathematica*, als auch die Nutzung des *Mathematica*-Kernels von außen ermöglichen. Diese Schnittstellen sind sowohl in *C* (*MathLink*), als auch *Java* (*JLink*) implementiert und liegen der *Mathematica*-Distribution bei. Die Verwendung dieser Bibliotheken ist für den nicht kommerziellen Einsatz kostenlos, für kommerzielle Anwendungen ist eine Lizenzgebühr zu entrichten (siehe [WRc]).

Wie in [WRa] nachzulesen ist, stellt *JLink* lediglich eine *Java*-API für den Zugriff auf *MathLink* dar. Die eigentliche Kommunikation erfolgt also über *MathLink*. Deshalb ist es nicht verwunderlich, dass *MathLink* und *JLink* über nahezu identische Methoden verfügen und *JLink* im „*C*-Stil“– also weitgehend prozedural – programmiert ist. In Abbildung 3 sind die verschiedenen Möglichkeiten, mit dem Kernel zu kommunizieren, dargestellt.

Da ich in der Vorlesung *Mathematik für Informatiker* bei Herrn Professor Dr. Ertel, erfahren habe, dass beispielsweise die Langzahl- und Gleitpunkt-Arithmetik von *Mathematica* den Implementierungen gängiger Programmiersprachen bei weitem überlegen ist, müsste ein Entwickler, der für sein Programm erweiterte arithmetische Funktionalität benötigt, diese nicht selbst implementieren, sondern könnte mittels *MathLink* oder *JLink* auf den *Mathematica-Kernel* zugreifen und die Berechnungen von diesem

durchführen lassen. *Mathematica* muss hierzu nicht einmal auf der lokalen Maschine installiert sein. Der Zugriff kann auch über ein Netzwerk erfolgen.

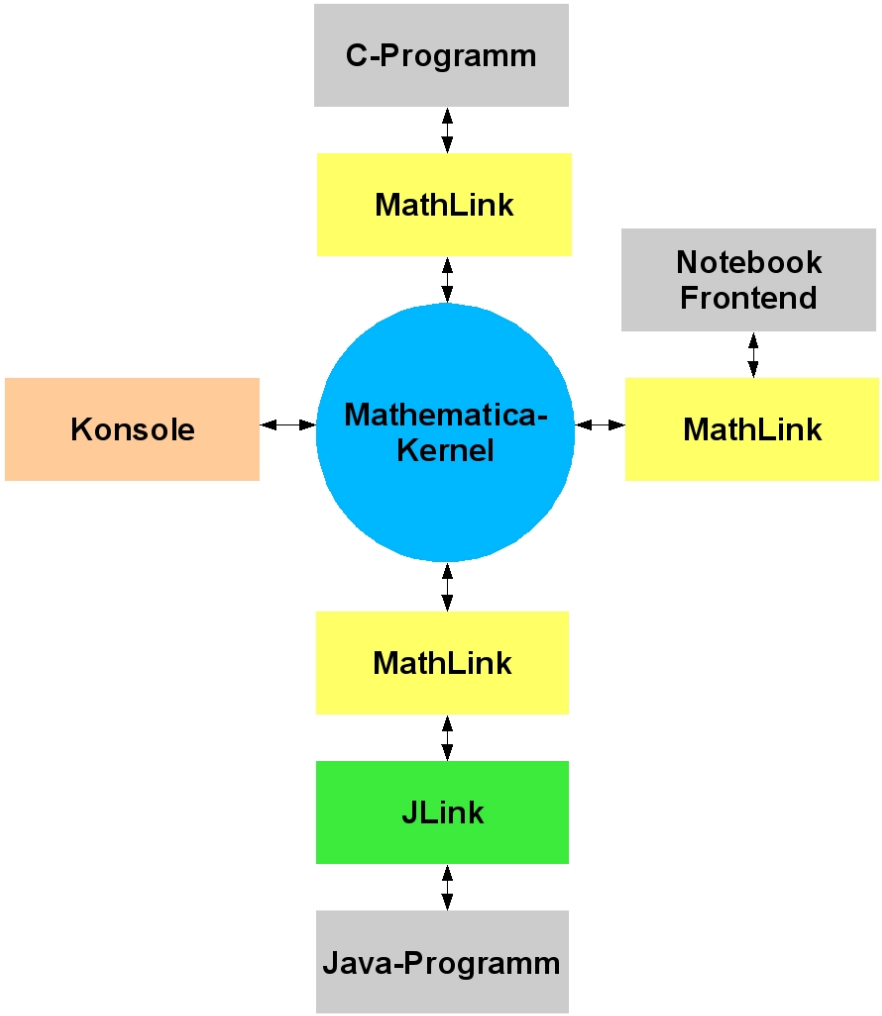

Abbildung 3: Kommunikation mit dem Kernel

Wie Stephen Wolfram in [Wol03] erläutert, dient die *MathLink*-Technologie nicht nur zur Anbindung externer Programme, sondern sie wird auch von *Mathematica* selbst verwendet. Das *Notebook*-Frontend kommuniziert ebenfalls über diese Schnittstelle mit dem *Mathematica-Kernel*. Es handelt sich also eigentlich bei Kernel und Frontend um zwei voneinander unabhängige Komponenten, die auch *standalone* verwendet werden können.

Diese Unabhängigkeit zwischen Kernel und Frontend dürfte wohl der Grund dafür sein, dass ich trotz tagelanger Recherche und Studium des Quellcodes keine Möglichkeit fand, über *JLink* auf das *Notebook*-Frontend zuzugreifen. Es stellte keine Schwierigkeit dar, mittels *JLink* den Kernel anzusprechen, aber das Frontend ist eben nicht Bestandteil des Kernels. Sollte dennoch eine Möglichkeit bestehen, würde es sich um ein „hidden-Feature“ handeln, welches so nicht dokumentiert ist. Zumindest stieß ich bei meinen Recherchen auf keine derartige Dokumentation und im Quellcode fand ich dazu ebenfalls keinen Anhaltspunkt.

2.4.1 Das Mathematica-Notebook-Format (*.nb)

Das Format **.nb* ist das gängige Dateiformat von *Mathematica*-Dokumenten. Es handelt sich im Grunde um eine *ASCII*-Textdatei, in der die Inhalte der *Mathematica*-Session mit Tags versehen, abgelegt sind. Diese Tags sind *Mathematica*-spezifisch und entsprechen in ihrer Synthax keinem gängigen Standard. Es handelt sich also um ein proprietäres Format. Es wird auch nicht von Tools zur Extrahierung von Inhalten (wie beispielsweise *DocBook* [DP]) unterstützt.

Glücklicherweise ist das Format jedoch sehr gut lesbar und es war nicht allzu schwierig, die Dateien entsprechend zu bearbeiten. Vereinfacht dargestellt, waren dabei folgende Schritte notwendig (die Details sind in Kapitel 3.6.2 beschrieben):

- Einlesen der Datei.
- Entfernen von Header und Footer.
- Entfernen von Kommentaren.
- Speichern des Strings in der Datenbank.
- Anzeigen einer Aufgabe:
  Header einfügen, Aufgabentitel, Aufgabentext, (falls vorhanden) Lösungstext und abschließend Footer anhängen.
- Anzeigen eines Aufgabenblattes:
  Header einfügen, Titel des Blattes, dann für jede Aufgabe, Nummer, Titel, Aufgabentext, gegebenenfalls Lösung und zuletzt den Footer anhängen.

2.5 Webmathematica

Bei meinen Vorüberlegungen (1.3) stellte sich bekanntlich die Frage, ob ich das Programm eventuell als Web-Anwendung entwickeln soll. Hierzu hätte sich der Einsatz von *Webmathematica* angeboten. *Webmathematica* basiert auf *Java-Servlet-Technologie* und läuft auf jedem gängigen *Servlet-Container*, beispielsweise *Apache Tomcat* [AFc]. Es kann kostenlos auf Webseiten eingesetzt werden (*Mathematica*-Lizenz vorausgesetzt). Die *Amateur-License* beinhaltet jedoch einige Einschränkungen:

- Keine kommerzielle Nutzung.
- Links zu jeder einzelnen Seite, die *Webmathematica*-Funktionalität beinhaltet, müssen *Wolfram-Research* mitgeteilt werden.
- Ein von *Wolfram-Research* bereitgestellter Link muss auf jeder Seite platziert werden.
- Es dürfen lediglich einzelne Berechnungen durchgeführt werden. Dem Nutzer darf über die Webseite nicht *Mathematica* selbst zur Verfügung gestellt werden.
- Es darf nicht möglich sein, dass der Benutzer selbst *Mathematica*-Befehle eingeben kann.

Diese Lizenz bietet nicht sehr viel Spielraum. Ein weiterer Punkt dieser Lizenz hätte den Einsatz von *Webmathematica* wohl unmöglich, beziehungsweise den Erwerb der kostenpflichtigen Version notwendig gemacht:

> *„You may not replicate, in whole or in part, the „look and feel“ of Mathematica with your Interface. In other words, your Interface cannot be composed of a combination of visual, design, or functional elements which could reasonably be expected to cause confusion between Mathematica and your Interface among Users of your Site. The criteria for assessing compliance with this restriction will be set and applied at our sole discretion. We agree to exercise that discretion reasonably.“*

Quelle: [WRe]

Mit anderen Worten, ich hätte die *Notebook*-Oberfläche nicht nachbauen dürfen, dies hätte wäre aber unumgänglich gewesen, denn in den *FAQ* zu *Webmathematica* ([WRd]) steht:

> *„No, webMathematica does not offer notebook simulation through a browser. However, noninteractive front-end functionality is accessible. Technology will be available in the future to link a client-side notebook front end through the web to a webMathematica server.“*

Mittlerweile ist mir noch ein weiterer Grund eingefallen, der gegen die Verwendung von *Webmathematica* gesprochen hätte. Unter 2.1.1 habe ich über das Sicherheitskonzept von *Java* gesprochen. Dieses Sicherheitskonzept wird von *Webmathematica* verletzt. *Webmathematica* bricht aus der Sandbox des Application-Servers aus und greift auf einen fremden Prozess (nämlich den *Mathematica-Kernel*) zu. Die *Webmathematica* beiliegende Installationsanleitung sieht auch entsprechende Konfigurationseinstellungen am Server vor.

2.6 Eclipse

Eclipse ist die für die Diplomarbeit verwendete Entwicklungsumgebung. *Eclipse* wurde ursprünglich von *IBM* entwickelt, Im Jahr 2002 entschloss sich *IBM* jedoch, das Projekt der Open-Source-Community zu übergeben. Zu diesem Zweck wurde die *Eclipse-Foundation* gegründet. Mittlerweile engagieren sich zahlreiche Unternehmen als strategische Partner für das Projekt. Neben *IBM* sind dies unter anderem *SAP*, *Borland* oder *Nokia*. Auf [EFb] findet sich eine Auflistung aller beteiligten Unternehmen.

Eclipse ist modular aufgebaut und kann mittels eines standartisierten Plugin-Systems (*OSGI*) erweitert werden. Während meines Praktikums gehörte es zu meinen Aufgaben, kleinere Plugins für *Eclipse* zu schreiben.

Zum Grundumfang gehört die Unterstützung der Sprache *Java*. Es sind jedoch Plugins für die unterschiedlichsten Programmiersprachen erhältlich. So wurde für die Diplomarbeit ein LaTeX-Plugin verwendet, mit dem dieses Dokument erstellt wurde.

Eclipse ist kostenlos bei [EF05] für verschiedenste Plattformen erhältlich und steht unter der *EPL (Eclipse Public License)* [EFa], die unter anderem auch erlaubt, modifizierte Versionen von *Eclipse* – auch kommerziell – zu vertreiben.

2.7 JGoodies

JGoodies ist ein von *Karsten Lentzsch* (siehe [Lena]) entwickeltes Framework, das die Entwicklung GUI-basierter Anwendungen unterstützt. Es bietet folgende Funktionalität:

- *Looks*
  Wie bereits unter 2.1 erwähnt, ist die Optik von *Java*-Anwendungen gewöhnungsbedürftig, da man sofort sieht, dass es sich um ein *Java*-Programm handelt. *JGoodies-Looks* bietet zusätzliche *Look-And-Feels*, unter anderem auch eines, welches der Optik von *Windows XP* sehr nahekommt.

- *Bindings*
  Ermöglicht das *binden* einer GUI-Komponente an ein Datenmodell, also beispielsweise eines Textfeldes an eine Persistenzklasse. Bei jeder Änderung an der GUI-Komponente wird automatisch das Modell geändert.

- *Forms*
  Erweitert *Java-Swing* um einen zusätzlichen Layoutmanager, das *Form-Layout*.

- *Validation*
  Stellt ein Framework zur Verfügung, dass die Validierung von Eingabedaten ermöglicht. Es kann überprüft werden, ob Pflichtfelder ausgefüllt sind, ob die Felder korrekt ausgefüllt sind (keine Buchstaben in einem Feld für Zahlen) oder ob bestimmte Menüeintrage ausgewählt sind. Dementsprechend können dann Aktionen (beispielsweise Fehlermeldungen) definiert werden.

In meinem Praktikumsbetrieb wurde der komplette Funktionsumfang von *JGoodies* verwendet. Die Aufgabenverwaltung beschränkte sich auf den Einsatz von *Looks* und *Forms*. Die *Bindings* hätten keine große Erleichterung gebracht, da die meisten Eingabeelemente Combo-Boxen sind, diese werden von den *Bindings* jedoch nicht unterstützt.
Die *Validation* erfordert einen hohen (einmaligen) Implementierungsaufwand, der für ein Projekt dieser Größe sicherlich unangemessen wäre.

Das Form-Layout ist ein sehr effizienter Layout-Manager für Formulardaten. Er arbeitet tabellenbasiert und verfügt über eine sehr gut lesbare Synthax, ähnlich wie bei Tabellen in LaTeX. Außerdem löst er ein Problem, das mit allen gängigen Layout-Managern auftritt:
Die Größe für GUI-Komponenten wird normalerweise in Pixeln angegeben. Da jedoch unklar ist, welche Systemschrift und Schriftgröße der Anwender auf seinem Rechner eingestellt hat, kann dies teilweise sehr unschöne Ergebnisse liefern. Form-Layout ermöglicht eine Größendefinition in sogenannten *Dialog-Units*. Diese sind von der Systemschrift abhängig. Die GUI-Komponente wird stets in optimaler Größe dargestellt.

Ich persönlich bin kein Freund von graphischen GUI-Buildern, da sie meist unübersichtlichen und oftmals auch überflüssigen Code erzeugen. Außerdem wird der Entwickler zu einem unsauberen Programmierstil eingeladen, da er stets geneigt ist, den ganzen Fensterinhalt in eine Klasse zu packen. Im Gegensatz dazu finde ich das arbeiten mit dem Form-Layout sehr angenehm. Es erfordert wenige Codezeilen, um auch komplexere Layouts zu entwerfen, die selbstverständlich auch beliebig verschachtelt werden können.

Als *Look-And-Feel* wird in der Aufgabenverwaltung unter *Windows* das *JGoodies-Windows-XP-* und unter *Linux* das *GTK-LAF* (aus dem *Java-SDK*) verwendet.

2.8 Betriebssystem

Das Programm wurde ausschließlich auf *Debian GNU Linux* entwickelt. Nach Abschluss des Entwicklungsprozesses habe ich das fertige Programm aber auch auf *Windows XP* getestet, in erster Linie um sicherzustellen, dass die optische Darstellung in Ordnung ist (Siehe 2.7). Leider musste ich feststellen, dass hierbei ein paar Probleme auftraten (siehe 4.1.1).

2.9 Tools

2.9.1 Apache Ant

Ant ist ein Buildtool der *Apache-Foundation*, das kostenlos auf deren Homepage heruntergeladen werden kann. Dort ist auch eine ausführliche Dokumentation vorhanden [AFa].

Ant verfügt über ähnliche Features wie das bekannte *GNU-make*, ist aber darüber hinaus mit spezieller Funktionalität für Programmierung in Java ausgestattet. *Ant* arbeitet mit sogenannten *Ant-Tasks*. Eine große Anzahl vordefinierter *Ant-Tasks* liegt bereits der Distribution von *Ant* bei. Diese ermöglichen unter anderem Dateioperationen, Kompilierung und Erstellen von *Jar-Archiven* und *JavaDoc*-Dokumentation. Es können aber auch eigene *Ant*-Tasks für spezielle Aufgaben erstellt werden.

Ant wird in der Diplomarbeit vielfach eingesetzt:

- Generieren der Mapping-Dateien aus den *Java*-Klassen (siehe 2.9.2).
- Erstellen und exportieren des Datenbankschemas (siehe 2.9.3).
- Anfertigen der Anwenderversion (siehe Kapitel 5.2).
- *Eclipse* selbst verwendet *Ant* für den Buildprozess.

Ant ist ebenfalls in *Java* geschrieben und somit plattformunabhängig.

2.9.2 XDoclet

> *„XDoclet ist eine OpenSource Engine für die Generierung von Code. Sie liest Java Quelldateien ein und wertet die darin enthaltenen Metainformationen aus. Die XDoclet Informationen werden in Form von JavaDoc-Tags direkt im Quellcode angegeben. Mit Hilfe von Code-Schablonen und den ermittelten Metainformationen erzeugt die Engine die gewünschten Ausgabedateien. Mögliche Zielformate sind unter anderem XML (z. B. Deployment Deskriptoren), Java Quelldateien oder Implementierungen der J2EE Patterns.“*

Quelle: Orientation In Objects [OIO]

Der Vorteil von *XDoclet* ist, dass man stets nur eine Datei bearbeitet. Bei *EJB (Enterprise Java Beans)* beispielsweise, erfolgt die Deklaration

mancher Komponenten teilweise in 6 voneinander unabhängigen Dateien. Dies ist einerseits unkomfortabel, andererseits fehleranfällig. Bei Einsatz von *XDoclet* wird *eine* Datei bearbeitet, mit Meta-Tags versehen, die restlichen werden automatisch generiert.

In meiner Diplomarbeit diente *XDoclet* zur Erstellung der *Hibernate*-Mapping-Dateien anhand der Metainformationen in den Persistenzklassen. Ich hatte *XDoclet* zuvor noch nie verwendet. In Anbetracht der geringen Anzahl an Datenbanktabellen wäre es vielleicht schneller gewesen, die Mapping-Dateien selbst zu erstellen, da durchaus eine gewisse Zeit für die Einarbeitung in *XDoclet* notwendig war. Sollte sich jedoch das Datenbank-Layout in der Zukunft ändern, etwa wegen zusätzlicher Funktionalität, ist der zusätzliche Aufwand durch die Verwendung von *XDoclet* geringer.

2.9.3 hbm2ddl

hbm2ddl ist Bestandteil des *Hibernate*-Toolsets. Es liest die Mapping-Dateien aus und erstellt (in Abhängigkeit von der verwendeten Datenbank und des verwendeten Dialekts) ein Datenbankschema und exportiert dieses Schema direkt in die Datenbank. Das Tool kann auch so konfiguriert werden, dass bei jedem Programmstart das Datenbankschema neu eingespielt wird, man also stets einen wohldefinierten Datenbankzustand hat.

2.9.4 JUnit

> *„JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units (meist Klassen oder Methoden) geeignet ist. Es basiert auf Konzepten, die ursprünglich unter dem Namen SUnit für Smalltalk entwickelt wurden.*
> *Ein JUnit-Test kennt nur zwei Ergebnisse: Entweder der Test gelingt (dann ist er „grün“) oder er misslingt (dann ist er „rot“).“*

Quelle: wikipedia.de [wika]

JUnit wurde ursprünglich von Erich Gamma und Kent Beck entwickelt, die auch den Begriff des *Test Driven Development* [MH04] mit geprägt haben.
Die Idee von *Test Driven Development* ist, dass Fehlerfreiheit dadurch garantiert wird, dass nichts implementiert wird, was nicht auch getestet wird. Werden Testfälle erst nach dem Code entwickelt, so besteht die Gefahr, einen Testfall zu übersehen. In der Praxis schreibt also ein Programmierer

zuerst einen automatisch wiederholbaren *(JUnit-)*Test und dann den zu testenden Code. Möchte ein anderer Programmierer den Code ändern, ruft er zuvor alle Tests auf, um sich zu vergewissern, dass der Code vor der Änderung fehlerfrei ist. Nun führt er seine Änderungen durch und ruft danach nochmals alle Tests auf. Wenn ein Test Fehler meldet, waren die Änderungen nicht korrekt und der Zyklus wiederholt sich so lange, bis alle Tests wieder fehlerfrei durchlaufen.

Mittlerweile gibt es derartige Test–Frameworks für viele Programmiersprachen. Selbst für Scriptsprachen wie beispielsweise *PHP* sind *JUnit*–basierte Frameworks verfügbar.

Nicht alle Klassen der Aufgabenverwaltung sind mit *JUnit*-Tests abgedeckt. Gerade bei GUI-Klassen ist das auch nicht ohne weiteres möglich, da dann ja Aktionen des Benutzers simuliert werden müssten, was erheblichen Implementierungsaufwand erfordern würde.

Ich habe mich deshalb darauf beschränkt, nur jene Methoden mit *JUnit* auf Korrektheit zu prüfen, die Berechnungen durchführen, Zeichenketten ersetzten oder Formate umwandeln. Ausreichende Testfälle vorausgesetzt, hat dies den Vorteil, dass eine Methode mit Sicherheit korrekte Ergebnisse liefert. Ein spätes Erkennen von Fehlern und daraus resultierende Fehlersuche, kann somit oftmals vermieden werden.

Anfangs habe ich mit *JUnit* auch Lese- und Schreibzugriffe auf die Datenbank durch *Hibernate* getestet, jedoch nur stichprobenhaft.

2.9.5 CVS

Das *Concurrent Version System* ist ein Open-Source-Programm für die Verwaltung unterschiedlicher Versionen von Dateien. Es ist kostenlos bei [GNUa] erhältlich. Die Verwendung einer Versionsverwaltung hat mehrere Vorteile:

- Man kann alte Versionen seiner Dateien mit der aktuellen Version vergleichen und gegebenenfalls wiederherstellen.
- Es besteht die Möglichkeit, alle aktuellen Dateien mit einem *Tag* zu versehen und damit als zusammengehörig zu kennzeichnen.
- Das ganze Projekt (das *Repository*) steht allen Programmierern im Netz zur Verfügung, die die nötige Zugriffsberechtigung haben.

Normalerweise wird *CVS* vorwiegend bei der Entwicklung im Team eingesetzt. Durch die Möglichkeit, alte Versionen wiederherzustellen, ist es aber auch sehr hilfreich, wenn man alleine arbeitet. Deshalb habe ich sowohl die Programmdateien, als auch dieses Dokument mit *CVS* verwaltet. Da das *CVS*-System auf einem anderen Rechner läuft, ist dies auch ein zusätzlicher Schutz vor Datenverlust.

2.9.6 Log4J

Log4J ist eine Bibliothek der Apache-Foundation [AFb], die dem Entwickler eine Logging-API zur Verfügung stellt. Der Vorteil zu einfachen Konsolenausgaben liegt darin, dass es verschiedene Logging-Level gibt (*DEBUG, INFO, WARN, ERROR*), der Programmierer also die Priorität der entsprechenden Ausgabe festlegen kann. In der Konfigurationsdatei von Log4J kann der Default-Level spezifiziert werden. Im Programmlauf werden nun alle Logging-Meldungen über dem Default-Level ausgegeben.

In der Praxis wird man während des Entwicklungsprozesses alle Logging-Ausgaben erhalten wollen und stellt folglich den Default-Level auf den niedrigsten Wert: *DEBUG*. Ist das Programm fertiggestellt und wird an den Anwender ausgeliefert, sollte man den Logging-Level erhöhen, da eine große Anzahl von Logging-Meldungen, zu Lasten der Performance geht. Wenn Fehler im Programm auftreten, kann er jederzeit zu Debug-Zwecken wieder heruntergesetzt werden. Es muss nicht neu kompiliert werden.

Ein weiterer Vorteil einer Logging-API ist die Tatsache, dass das Format der Ausgaben angepasst werden kann. Es ist zum Beispiel möglich, bei jeder Meldung Klassenname und Code-Zeile der entsprechenden Meldung mit auszugeben.

Mittlerweile besitzt das *Java-SDK* eine eigene Logging-API. Da jedoch *Hibernate* auch *Log4J* verwendet, schien es nicht sinnvoll noch eine zweite Logging-API zu verwenden.

3 Programmarchitektur

Grundsätzlich basiert das Programm unter dem Gesichtspunkt des Software-Designs auf zwei Modellen:

- Dem Schichtenmodell (siehe 3.1)
- Dem Komponentenmodell (siehe 3.4)

Wie in den entsprechenden Kapiteln ausführlich beschrieben ist, sind solche Modelle ein gutes Hilfsmittel, um Programme zu entwickeln, die durch saubere Strukturierung, gut zu verstehen, komfortabel weiterentzuentwickeln und einfach zu warten sind.

Bei *Java* werden die Klassen in *Packages* verwaltet. Diese Packages können beliebig tief verschachtelt werden. Ihre Benennung ist prinzipiell frei wählbar. Es hat sich jedoch eingebürgert, an den Anfang der Package-Hierarchie die Domain in umgedrehter Form anzuhängen, um ein Programm oder eine Bibliothek weltweit eindeutig zu kennzeichnen (also beispielsweise *com.jgoodies.forms*). Da dies jedoch keineswegs verpflichtend ist und ich mir nicht sicher war, ob ich die Domain *fh-weingarten.de* hierfür verwenden durfte, habe ich meinen Paketen das Präfix *math.manager* vorangestellt.

3.1 Schichtenmodell

In einem Schichtenmodell besteht ein Programm aus verschiedenen Schichten. Die Anzahl der Schichten ist hierbei abhängig von Art und Komplexität der Anwendung.
Heide Balzert schlägt in [Bal99] für kleinere und mittlere Anwendungen drei Schichten vor:

> *„Die GUI-Schicht realisiert die Benutzungsoberfläche einer Anwendung. Dazu gehören die Dialogführung und die Präsentation aller Daten in Fenstern, Berichten usw. Die Fachkonzeptschicht modelliert den funktionalen Kern der Anwendung. Außerdem enthält sie die Zugriffe auf die Datenhaltungsschicht, in der die jeweilige Form der Datenspeicherung realisiert wird, z.B. mit einem objektorientierten Datenbanksystem.“*

In Abbildung 7 sind die verschiedenen Schichten des Programms dargestellt. Das Datenbank-Management-System ist zwar nicht Bestandteil des Programms, wurde aber zum besseren Verständnis als *Datenhaltungsschicht* mit aufgenommen. Ebenso besteht die Datenzugriffsschicht streng genommen

aus zwei Schichten, da *Hibernate* auf *JDBC* aufsetzt.

In der Fachkonzept- und GUI-Schicht sind zusätzliche Unterteilungen vorhanden, da hier das *MVC*-Designpattern verwendet wird, das unter 3.3.1 näher beschrieben wird. In der Fachkonzeptschicht kommt zudem das *Singleton*-Design-Pattern zum Einsatz (3.3.2).

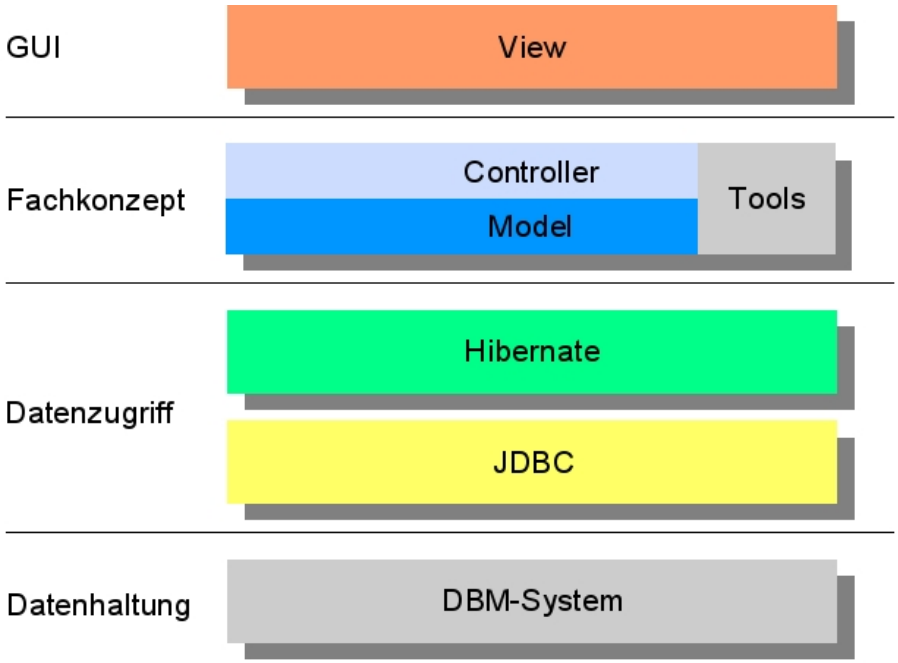

Abbildung 4: Schichten der Aufgabenverwaltung

Wie in [Bal99] nachzulesen ist, basiert ein solches Modell auf folgenden Prinzipien:

- Die Kommunikation erfolgt nur mit der jeweils nächsten Schicht.
- Die GUI-Schicht enthält keinerlei Programmlogik. Sie ist für die Darstellung zuständig und informiert die Fachlogikschicht über Zustandsänderungen (beispielsweise Benutzereingaben).
- Die Fachlogikschicht enthält die Fachklassen. Sie wird von der GUI-Schicht über Zustandsänderungen informiert und führt anhand dieser Änderungen Aktionen aus. Sie nützt die Dienste der Datenzugriffsschicht und initiiert Zustandsänderungen der GUI.

- Die Datenzugriffsschicht dient lediglich der Kommunikation mit der Datenbank. Sie enthält ebenfalls keine Programmlogik. Sie stellt der Fachlogikschicht Daten bereit und sorgt für die Ablage der von ihr erhaltenen Daten.

Wie zu erkennen ist, bestehen durchaus Gemeinsamkeiten zu anderen Schichtenmodellen, wie beispielsweise dem *OSI*-Modell. Im Unterschied zu diesem, hat hier die Fachlogikschicht Zugriff auf die darüber- und auch auf die darunterliegende Schicht. Im OSI-Modell kennt jede Schicht nur die jeweils darunterliegende.

Die Vorteile eines solchen Modells sind in [HR01] folgendermaßen beschrieben:

> - *„Höhere Ebenen (Systemkomponenten) werden einfacher, weil sie tiefere Ebenen (Systemkomponenten) benutzen können.“*
> - *„Änderungen auf höheren Ebenen sind ohne Einfluß auf tieferen Ebenen.“*
> - *„Höhere Ebenen lassen sich abtrennen, tiefere Ebenen bleiben trotzdem funktionsfähig.“*
> - *„Tiefere Ebenen können getestet werden, bevor die höheren Ebenen lauffähig sind.“*

In der Praxis hat sich der Einsatz dieses Modells als sehr hilfreich erwiesen. Durch die saubere Trennung der einzelnen Schichten blieb das Programm auch bei steigendem Umfang sehr übersichtlich. Ich habe *bottom-up* entwickelt, also mit der Datenzugriffsschicht begonnen, dann die Fachklassen und zuletzt die GUI entwickelt. Der Vorteil bestand für mich darin, dass es einfacher war, bereits vorhandene Funktionalität zu testen. Ich konnte mich bei der Entwicklung der nächst höheren Schicht weitgehend darauf verlassen, dass die tiefere Schicht funktioniert. Dieses Vorgehen setzt allerdings eine intensive Planung voraus, da sonst die Gefahr besteht, dass im nachhinein größere Änderungen an den tieferen Schichten notwendig werden. In diesem Fall hielten sich solche Änderungen in Grenzen.

3.2 Persistenzschicht

Wie im vorigen Kapitel erwähnt, war die Persistenzschicht die erste Schicht, die ich entwickelte. In Abbildung 5 ist das Entity-Relationship-Diagramm

und in Abbildung 6, die Tabellenstruktur der Datenbank dargestellt. Da im Programm, und auch bei den Datenbankeinträgen englische Bezeichnungen verwendet wurden, werden hier – um Unklarheiten zu vermeiden – die deutschen Übersetzungen genannt:

Lecturer	–	Dozent
Lecture	–	Vorlesung
Course	–	Studiengang
Excercise	–	Aufgabe
Sheet	–	Aufgabenblatt
Headword	–	Schlüsselwort
Subject	–	Fachgebiet

Bezüglich der Relationen und Kardinalitäten gab es folgende Vorüberlegungen:

- Jede Aufgabe wird von genau einem Dozenten erstellt.
- Jedes Aufgabenblatt wird von genau einem Dozenten erstellt und enthält ausschließlich Aufgaben dieses Dozenten.
- Jede Aufgabe gehört zu genau einem Fachgebiet, jedes Fachgebiet kann beliebig viele Untergebiete haben.
- Jede Aufgabe kann in beliebig vielen Aufgabenblättern vorhanden sein.
- Jedes Aufgabenblatt gehört zu genau einer Vorlesung.

Ich habe es auch ermöglicht, dass verschiedene Dozenten für die gleiche Vorlesung Aufgabenblätter erstellen können, da sich bei einigen Vorlesungen Dozenten semesterweise abwechseln. Es ist jedoch im Programm sichergestellt, dass ein Dozent nur die Aufgaben und Aufgabenblätter angezeigt bekommt, die er auch selbst erstellt hat, unabhängig davon, ob ein anderer Dozent Blätter für die gleiche Vorlesung besitzt.

Schwierig war die Frage, ob ein Blatt einer Vorlesung zugeordnet wird, da beispielsweise ein Aufgabenblatt in mehreren Vorlesungen verwendet werden könnte. Ich hielt dies jedoch eher für unwahrscheinlich.

Ein weiteres Problem stellte die Zugehörigkeit eines Dozenten zu einem Studiengang dar. Normalerweise gehört ein Dozent zu einem Studiengang. Da es jedoch auch Vorlesungen gibt, die von Studenten aus unterschiedlichen

Studiengängen gehört werden, habe ich dem Dozenten die Wahlmöglichkeit gelassen, unter welchem Studiengang er ein Blatt ablegt. So könnte Herr Professor Dr. Hulin ein Blatt zu *Mathematik 1* sowohl unter *AI*, als auch unter *TM*, ablegen. Man hätte hier sicher auch anders vorgehen und einen Dozenten fest mit einem Studiengang assoziieren können. Ich wollte die Dozenten aber in diesem Punkt nicht einschränken.

Wie bereits in 2.2.3 und 2.9.2 erwähnt, wurden zuerst die Persistenzklassen entwickelt, dann mittels *XDoclet* die Deskriptor-Dateien generiert und abschließend mit *hbm2ddl* das Datenbankschema erstellt. Die Klassen der Persistenzschicht befinden sich im Paket *math.manager.persistence*. Ich möchte an dieser Stelle darauf verzichten Quellcode abzubilden, unter 2.2.3 sind ja bereits exemplarisch Codebeispiele von Persistenzklasse und Mappingdatei dargestellt. Bei Bedarf kann der Quellcode jederzeit auf der beiliegenden CD eingesehen werden.

Zu den Persistenzklasse bleibt noch anzumerken, dass es keine korrospondierenden Klassen zu den Verknüpfungstabellen *EXCERCISE_HEADWORD* und *EXCERCISE_SHEET* gibt. Stattdessen enthalten die beteiligten Klassen Collecions der jeweils anderen (siehe 2.2.3). Die genaue Konfiguration von *Hibernate* ist in Kapitel 3.7 beschrieben.

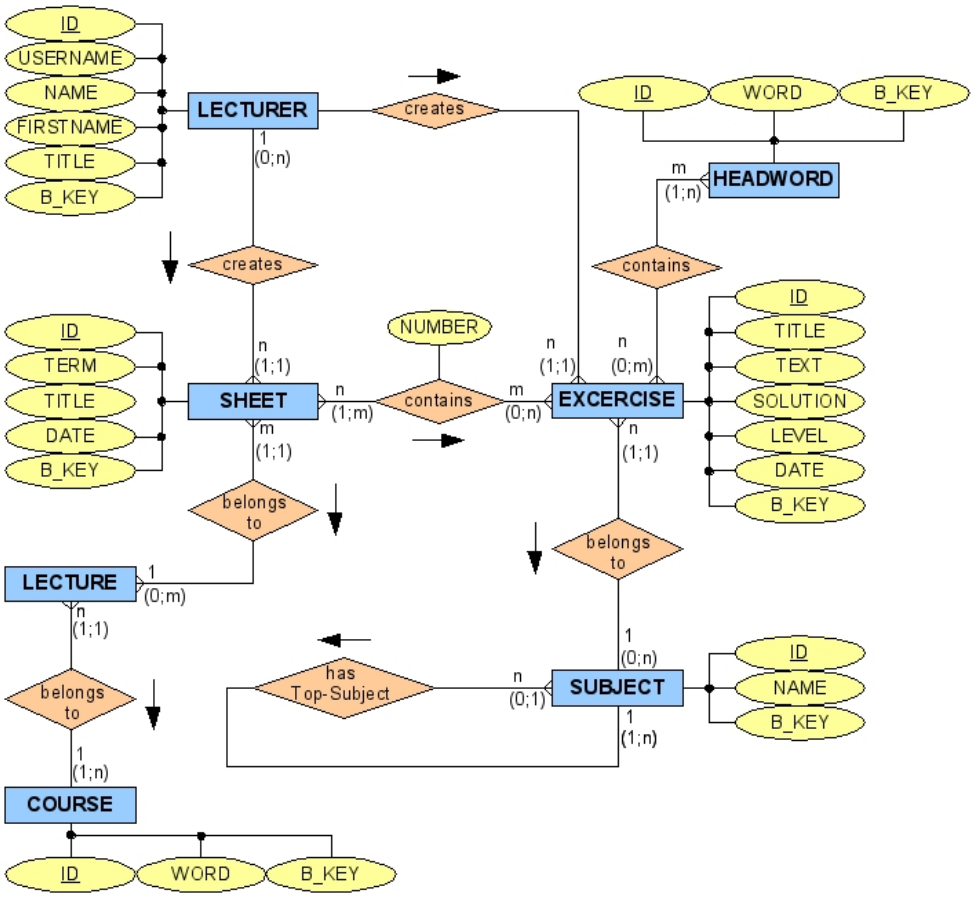

Abbildung 5: Entity-Relashionship-Diagram

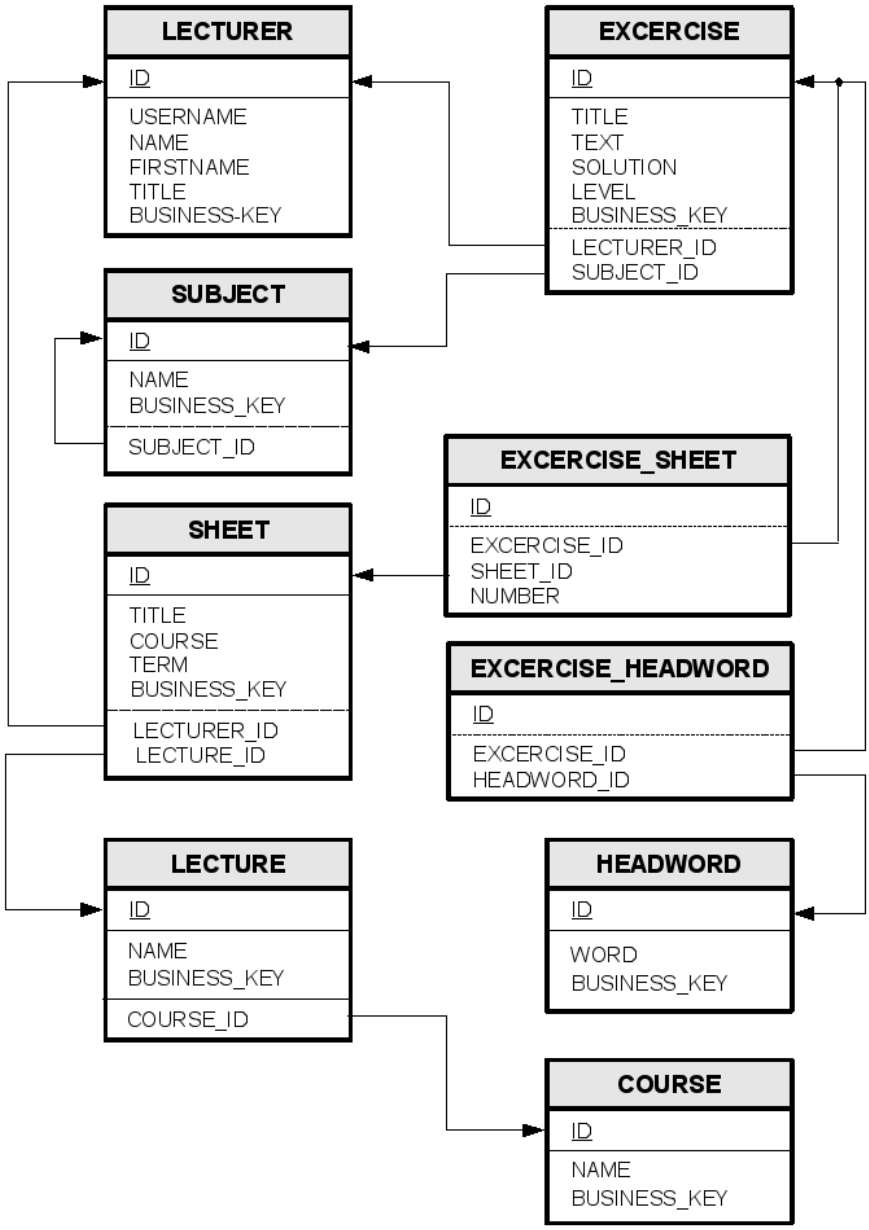

Abbildung 6: Tabellenstruktur

3.3 Design-Patterns

3.3.1 MVC-Pattern

MVC (Model-View-Controller) ist ein Design-Pattern, welches erstmals 1979 von *Trygve Reenskaug* vorgestellt wurde [Ree]. Es sieht vor, ein Programm in die drei Teile Datenmodell (*Model*), Darstellung (*View*) und Programmsteuerung (*Controller*) aufzuteilen (*Separation of Concerns*).

> *„MVC besteht aus den drei Klassen Model, View und Controller. Das Model-Objekt repräsentiert das Fachkonzeptobjekt. Oft gibt es mehrere Möglichkeiten, die fachlichen Daten zu präsentieren. Für jede Präsentation gibt es ein View-Objekt. Das Controller-Objekt bestimmt, wie die Benutzungsoberfläche auf Eingaben reagiert. Jedes View-Objekt besitzt ein zugehöriges Controller-Objekt, das diese Darstellung mit der Eingabe verbindet. Das impliziert, daß es zu jedem Model-Objekt eine beliebige Anzahl von Paaren (View, Controller) geben kann, jedoch mindestens eines.“*

Quelle: Heide Balzert, Lehrbuch der Objektmodellierung [Bal99]

Das *MVC*-Pattern steht in keinster Weise in Konkurrenz zu einem Schichtenmodell (siehe 3.1). Es dient vielmehr dazu, ein solches zu verfeinern. Ob das MFC-Pattern verwendet wird, ist weniger eine Frage des persönlichen Geschmacks, sondern vielmehr der verwendeten Programmiersprache und des Vorgehens bei der Programmentwicklung. Das von *Microsoft* bei den *Microsoft Foundation Classes (MFC)* verwendete *Document-View-Pattern* basiert beispielsweise auf dem *Observer-Pattern*. *View* und *Controller* sind hier nicht getrennt. Das *Document* entspricht dem *Model*, die *View* enthält zusätzlich den *Controller* [Sar]. *Visual Basic* verwendet ebenfalls dieses Prinzip.

Die Zusammenfassung von *View* und *Controller* drängt sich bei Entwicklung mit einem graphischen GUI-Builder förmlich auf: Es wird auf einen Button geklickt und automatisch erstellt der GUI-Builder die entsprechende Event-Handler-Methode in der gleichen Klasse. Es würde also Zusatzaufwand entstehen, die *Controller*-Methode in eine andere Klasse auszulagern.

Das Event-Handling von *Java-Swing* wurde anhand des *MVC*-Patterns designed [Sun]. Da ich außerdem (wie bereits in Kapitel 2.6 erwähnt) auch keinen GUI-Builder verwendet habe, war die Anwendung des *MVC*-Patterns obligatorisch.

In Abbildung 3.3.1 sind die Abläufe dargestellt:

- Werden Benutzereingaben gemacht (sich also die *View* ändert), erhält der *Controller* einen Event.
- In Abhängigkeit von der Art des Events ändert er das *Model* und/oder die *View*.
- Ändern sich Daten im *Model*, aktualisiert das *Model* die *View*.

In der Aufgabenverwaltung wird das *Model* von dem Package *math.manager.component*, die *View* von *math.manager.component.gui* und der *Controller* von *math.manager.action* repräsentiert.

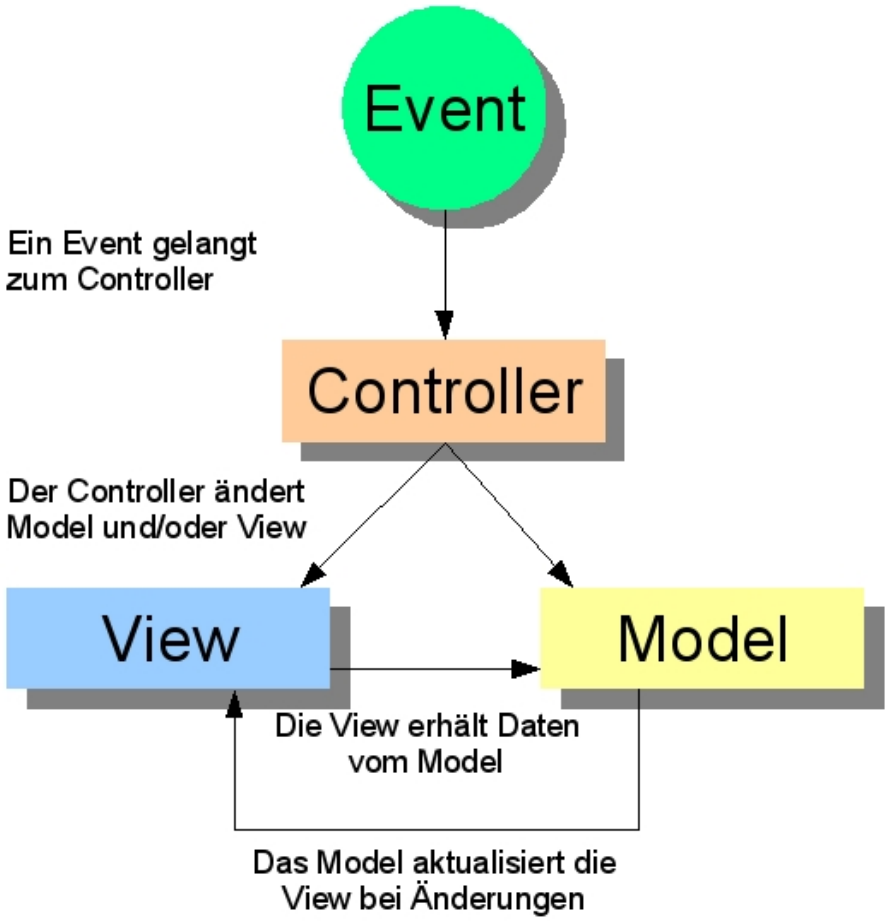

Abbildung 7: Das MVC-Pattern

3.3.2 Singleton-Pattern

In manchen Situationen ist es unnötig oder sogar ungünstig, dass von einer Klasse mehr als ein Objekt existiert. So hält beispielsweise die Klasse *HibernateUtil* (siehe 3.6.1) eine Referenz auf ein *Hibernate-Session*-Objekt. Da das Programm nicht über mehrere Threads verfügt, wäre es also vollkommen ausreichend, wenn im Programm stets das gleiche Session-Objekt verwendet würde und nur dann ein neues erzeugt wird, wenn der Time-Out abgelaufen ist. Hierfür gibt es das *Singleton*-Pattern.

Heide Balzer schreibt in [Bal99]:

> *„Verwenden Sie dieses Muster, wenn es genau ein Objekt einer Klasse geben und ein einfacher Zugriff darauf bestehen soll, ...“*

Alle Klassen aus dem Paket *math.manager.tools* (siehe Kapitel 3.6) verwenden das *Singleton*-Pattern. Es ist ja auch beispielsweise nicht nötig, dass von einer Klasse, die Formate konvertiert, mehr als eine Instanz existiert.

Es gibt mehrere Varianten, das *Singleton*-Pattern zu implementieren. Ich habe eine Variante verwendet, die Joshua Bloch in [Blo01] vorschlägt:

```
1  public Class Singleton {
2      // eine Instanz wird als statische Varaiable erzeugt
3      private static final Singleton instance = new Singleton();
4      // Konstruktor ist private, Es kann keine neue Instanz
5      // erzeugt werden
6      private Singleton(){}
7      // die Zugriffsmethode
8      public Singleton getInstance() {
9          return instance;
10     }
11 }
```

Listing 6: Singleton-Implementierung

Die Klasse *HibernateUtil* ist in dieser Hinsicht übrigens ein Spezialfall, da es hier ja nicht ausreicht, eine statische Instanz zu erstellen. Wenn der Time-Out einer Session abläuft, muss ein neues Session-Objekt erstellt werden. Dies werde ich aber in Kapitel 3.6.1 noch näher erläutern.

3.4 Das Komponentenmodell

Ein Komponentenmodell kapselt eine Funktionalität eines Softwareprogramms in einer Komponente. Es verfügt zudem über ein Framework, das mit den Komponenten kommuniziert, beispielsweise Komponenten lädt oder entlädt. Jede Komponente ist eigenständig, muss aber zwingend bestimmte Schnittstellen besitzen, um vom Framework angesprochen werden zu können.

Von Gruhn und Thiel stammt folgende, sehr detaillierte Definition [GT00]:

> *„Ein Komponentenmodell legt einen Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- bzw. Kompositionsmöglichkeiten (Komposition) sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt. Darüber hinaus wird durch ein Komponentenmodell eine Infrastruktur angeboten, die häufig benötigte Mechanismen eine konkreten [sic] wie Verteilung, Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementieren kann.“*

Bekannte Komponentenmodelle sind beispielsweise die *Microsoft COM*-Architektur oder *Enterprise Java Beans*. Bezüglich der Vorteile eines Komponentenmodells schreibt Volker Gruhn in [Gru02]:

> *„Ein Vorteil komponentenbasierter Softwarearchitekturen liegt in der mehrfachen Wiederverwendung von Bausteinen und in der damit verbundenen Steigerung der Produktivität.“*

Genau diese Erfahrung machte ich während meines Praktikums. In meinem Praktikumsbetrieb wird ebenfalls ein (selbstentwickeltes) Komponentenmodell verwendet. Nach einer gewissen Zeit der Einarbeitung ins Framework, lassen sich sehr schnell neue Komponenten erstellen. Gemeinsame Funktionalität ist (nur einmal) zentral, implementiert. Durch Verwendung generischer Methoden und Klassen, können diese in verschiedensten Situationen verwendet werden und müssen nicht laufend neu erstellt werden, was für bessere Wartbarkeit sorgt.

Ein solches Modell zwingt auch alle beteiligten Entwickler, sich an einen gemeinsamen Standard zu halten, da die Komponentenschnittstellen fest vorgegeben sind.

Andererseits weiß ich aus der Vorlesung *Enterprise Java Beans* bei Herrn Sproll, dass ein Komponentenmodell nicht nur Vorteile hat, insbesondere wenn es derart komplex wie *EJB* ist.

- Der Lernaufwand ist sehr hoch, da die Unterschiede zu herkömmlicher Softwareentwicklung sehr groß sind.
- Vieles wird (aus Entwicklersicht) transparent vom Framework erledigt. Es entsteht die Illusion, dass man sich um manche Aspekte gar nicht mehr kümmern müsse – zumindest so lange, bis Fehlersuche ansteht.
- Für kleine und mittlere Projekte ist der Einsatz solcher Technologie viel zu aufwändig. Laut Aussage von Herrn Sproll soll man *EJB* nur verwenden, wenn man Features wie *Clustering* tatsächlich benötigt.

Aufgrund meiner positiven Erfahrungen während des Praktikums entwickelte ich dennoch im Laufe meiner Diplomarbeit ein (ungleich simpleres) Komponentenmodell.

3.5 Die Komponenten

Die Aufgabenverwaltung verfügt über vier Komponenten:

- Aufgabe erstellen (3.5.4)
- Aufgabe bearbeiten (3.5.5)
- Aufgabenblatt erstellen (3.5.6)
- Aufgabenblatt bearbeiten (3.5.7)

Entsprechend dem *MVC*-Pattern (siehe 3.3.1) besteht jede dieser Komponenten aus drei Teilen: *Model*, *View* und *Controller*. In Abbildung 8 sind die Assoziationen zwischen diesen und den sonstigen am Ablauf beteiligten Klassen in einem Klassendiagramm dargestellt. Da das Programm aus insgesamt 52 Klassen besteht, wäre eine Abbildung des kompletten Klassendiagramms wohl zu umfangreich. Ich beschränke mich deshalb in diesem und den nächsten Kapiteln auf jene Teile, die aktuell von Bedeutung sind. Auch habe ich nur dann Variablen und Methoden dargestellt, wenn es mir zum Verständnis wichtig erschien. Beim Laden einer Komponente passiert folgendes:

- In der Klasse *MenuBuilder* befindet sich das Menü. Den einzelnen Menüpunkten ist die Klasse *ApplicationManager*, welche das *ActionListener*-Interface implementiert, als *ActionListener* zugewiesen.
- Der *ApplicationManager* erhält einen ActionEvent wenn der Benutzer einen Menüpunkt auswählt.
- Er erzeugt eine Instanz von *Model* (*AbstractComponent*), *Controller* (*ActionManager*) und *View* (*IGuiComponent*). Genau genommen erzeugt er natürlich Instanzen der konkreten Unter-Klassen (siehe Kapitel 3.5.1, 3.5.2 und 3.5.3). Weder *AbstractComponent*, noch *IGuiComponent* können instantiiert werden, da es sich um eine abstrakte Klasse, beziehungsweise ein Interface handelt. Er spricht sie jedoch über die Superklasse, beziehungsweise das Interface an und ruft ausschließlich deren Methoden auf. Ansonsten würde das Komponentenmodell nicht funktionieren, da es sich um einen generischen Ansatz handelt.
- Wie anhand der kursiven Schrift der Methoden von *AbstractComponent* zu erkennen ist, handelt es sich um abstrakte Methoden, die von den Sub-Klassen implementiert werden müssen.
- *ApplicationManager* ruft die Methode *setUp()* von *AbstractComponent* auf. Dies startet den Initialisierungsprozess des *Models*.

- Es werden nacheinander alle Methoden von *IGuiComponent* aufgerufen (bis auf *dispose()*). Nachdem *buildGui()* als letzte Methode abgearbeitet ist, wird die *View* sichtbar.
- Wird eine Komponente beendet, werden in *IGuiComponent* alle Listener entfernt und danach *dispose()* aufgerufen, wodurch der Frame geschlossen wird. *AbstractComponent* wird durch Aufruf der Methode *tieDown()* entladen. Die Objektreferenzen im *ApplicationManager* werden auf *null* gesetzt.

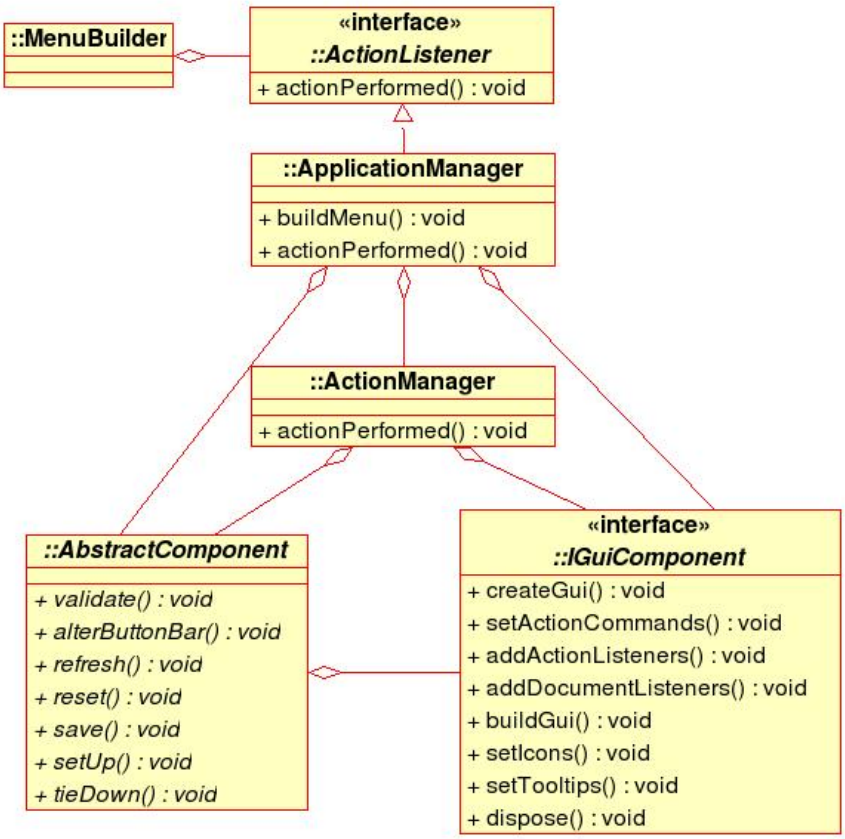

Abbildung 8: Aufbau einer Komponente

3.5.1 Model

Wie im vorigen Kapitel beschrieben, wird das Model von der Klasse *AbstractComponent* und ihren Subklassen repräsentiert. Im Klassendiagramm (Abbildung 9) ist die Vererbungshierarchie schematisch dargestellt.

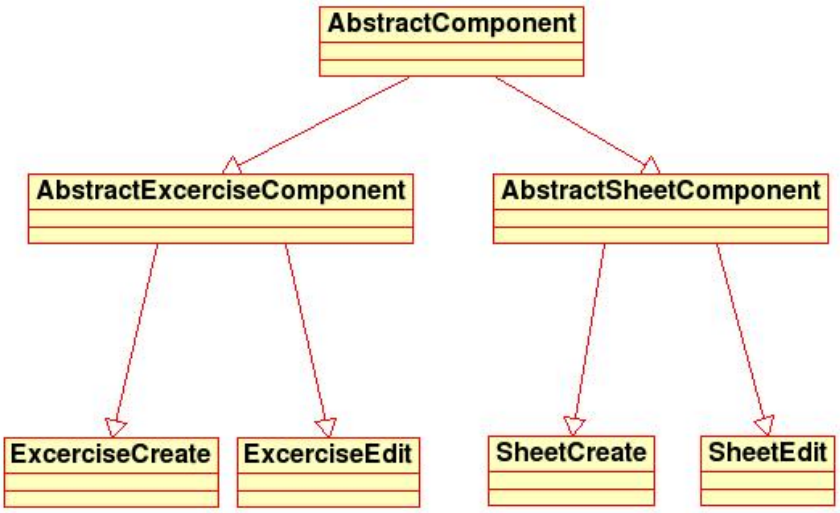

Abbildung 9: Vererbungshierarchie des Models

Von der Superklasse *AbstractComponent* sind zwei weitere (abstrakte) Klassen abgeleitet, *AbstractExcerciseComponent* und *AbstractSheetComponent*. Obwohl diese beiden Klassen keine abstrakten Methoden enthalten, sind sie dennoch so deklariert, weil ich sonst gezwungen gewesen wäre, in beiden Klassen alle abstrakten Methoden von *AbstractComponent* zu implementieren und sei es als leeren Funktionsrumpf. Abstrakte Methoden, die nicht in *AbstractExcerciseComponent* – beziehungsweise *AbstractSheetComponent* – implementiert sind, sind dies in den abgeleiteten, konkreten Klassen.

Die Vererbungshierarchie beruht darauf, gemeinsam genutzte Funktionalität nur ein einziges Mal zu implementieren und dadurch Zeit und Quellcode einzusparen. Es besteht allerdings immer die Gefahr, eine Methode um jeden Preis generisch erstellen zu wollen und am Ende vielleicht das Gegenteil zu erreichen. Dieses Prinzip wurde so auch von Herrn Professor Dr. Zeller in der Vorlesung Software-Engineering vermittelt.

Ich muss an dieser Stelle jedoch anmerken, dass ich im *Model* mit der Vererbungshierarchie nicht vollständig zufrieden bin. Die *Models* für Aufgaben (*ExcerciseCreate* und *ExcerciseEdit*) teilen sehr viel Funktionalität. Es war also sicher sinnvoll, sie von einer Klasse (*AbstractExcerciseComponent*) abzuleiten. Ebenso verhält es sich mit den *Models* für Aufgabenblätter. Allerdings gibt es auch zwischen *ExcerciseEdit* und *SheetCreate* ein paar Gemeinsamkeiten. Es wäre jedoch unmöglich gewesen, eine gemeinsame Superklasse für *ExcerciseEdit, SheetCreate* und *ExcerciseCreate* zu bilden, da die beiden Letzten keinerlei Gemeinsamkeiten haben. Das gleiche gilt für die andere Variante, *ExcerciseEdit* von *AbstractSheetComponent* abzuleiten. Die meiner Meinung nach ideale Lösung ist in Abbildung 10 dargestellt. Die gemeinsame Funktionalität von *ExcerciseEdit* und *SheetCreate* wäre hier in einer zusätzlichen, gemeinsamen Superklasse implementiert. *ExcerciseEdit* und *SheetCreate* würden also jeweils von zwei Klassen erben. In einer anderen Sprache wäre das so möglich, allerdings nicht in *Java*, da *Java* keine Mehrfachvererbung unterstützt.

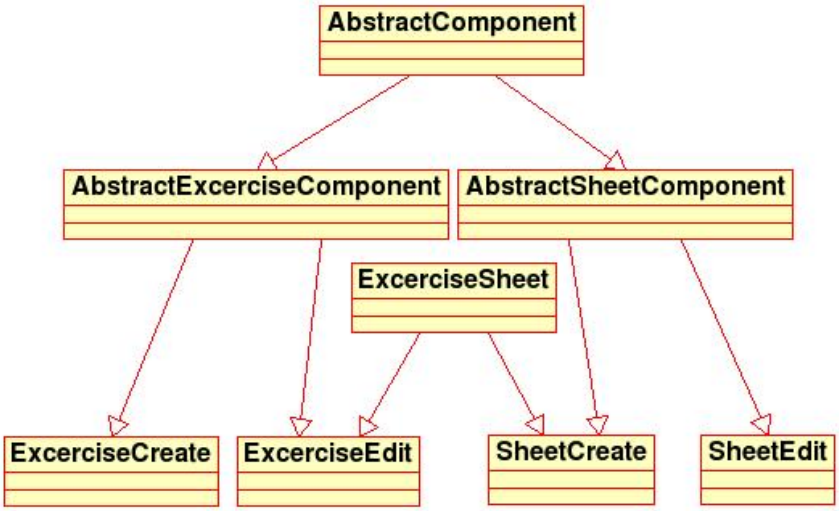

Abbildung 10: Model mit Mehrfachvererbung

3.5.2 View

Bei der *View* war eine etwas andere Vorgehensweise nötig. Zwar sieht die im Klassendiagramm (Abbildung 11) dargestellte Vererbungshierachie auf den ersten Blick gleich aus, wie jene in Abbildung 9 (aus Kapitel 3.5.1), jedoch wird hier die Implementierung verschiedener Methoden durch Verwendung eines Interfaces erzwungen.

Dies war aufgrund der fehlenden Mehrfachvererbung notwendig. Zwar sind die Klassen *AbstractExcerciseFrame* und *AbstractSheetFrame* keine abgeleiteten Klassen, jedoch beinhaltet das Paket *math.manager.component.gui* noch 13 weitere Klassen, die von den vier konkreten Subklassen der *View* verwendet werden. Es handelt sich dabei um verschiedene Panels und Frames, die in eigene Klassen ausgelagert sind, um die *View*-Klassen nicht zu groß werden zu lassen. Diese zusätzlichen Klassen wiederum, erben von *JPanel*, beziehungsweise von *JFrame*. Aufgrund des in Kapitel 3.4 beschriebenen Prinzips sollten diese Klassen jedoch ebenfalls über die vorgeschriebenen Methoden der *View* verfügen, da sie selbst Bestandteil der *View* sind. Also konnte die Implementierungsverpflichtung nur über ein Interface realisiert werden.

Im Programmablauf funktioniert dies folgendermaßen:
Wenn ein Objekt einer konkreten *View*-Klasse (beispielsweise *SheetEditFrame*) erzeugt wird und (wie in Kapitel 3.5 beschrieben) nacheinander die vorgeschriebenen Methoden aufgerufen werden, ruft die *View*-Klasse wiederum die gleichen Methoden in ihren jeweiligen Panels auf. Damit ist der Initialisierungsprozess vereinheitlicht. Ausgenommen sind hier natürlich Fenster, die an eine bestimmte Aktion gebunden sind und erst erzeugt werden, wenn ein Event ausgelöst wird, wie in etwa das Fenster, welches geöffnet wird, wenn sich der Anwender Informationen über eine Aufgabe anschauen möchte.

Dieses Vorgehen erwies sich für mich als sehr hilfreich, da man unabhängig von der aktuell bearbeiteten Klasse stets weiß, in welcher Methode welche Aktion ausgeführt wird. Außerdem ist es nahezu unmöglich, bestimmte Initialisierungsschritte – wie das hinzufügen eines Listeners – zu vergessen, da die entsprechende Methode zwingend implementiert werden muss. Letztendlich muss so weniger Zeit für die Suche nach trivialen Fehlern aufgewendet werden.

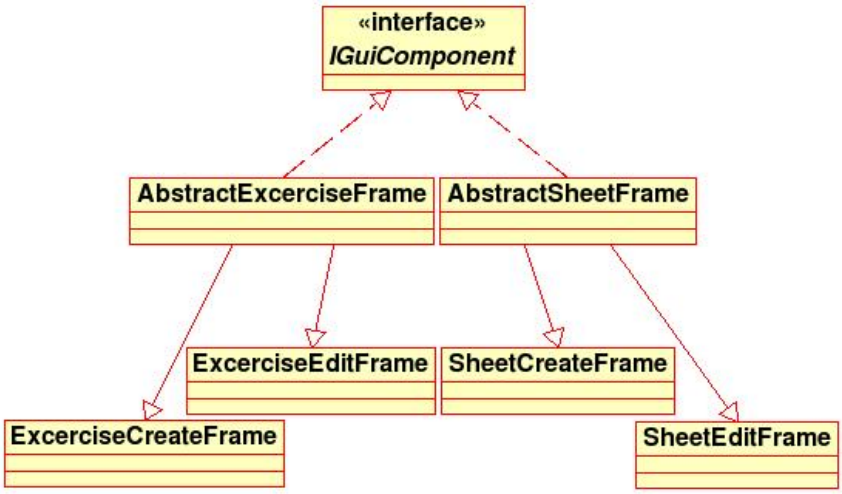

Abbildung 11: Vererbungshierarchie der View

3.5.3 Controller

Der *Controller* ist ähnlich aufgebaut wie das Model, besitzt jedoch ausschließlich konkrete Klassen. Da der *Controller* nur dazu dient, auf Action-Events zu reagieren und das *ActionListener*-Interface implementiert, ist in jeder Klasse nur eine Methode vorhanden: Die Methode *actionPerformed()*, die aufgerufen wird, wenn ein *ActionEvent* ausgelöst wird.

In *actionPerformed()* wird innerhalb einer Kontrollstruktur das *ActionCommand* des Events abgefragt und in Abhängigkeit davon die entsprechende Aktion ausgeführt. Die Subklassen rufen jeweils zuerst die Methode ihrer Superklasse auf, da auch hier das Prinzip gilt, nichts doppelt zu implementieren.

Die oberste Klasse (*ActionManager*) behandelt nur einige wenige Events, beispielsweise das Schließen der Komponente durch drücken des *Abbrechen*-Buttons.

Auch beim *Controller* wäre eine Variante mit Mehrfachvererbung (wie in Abbildung 10 aus Kapitel 3.5.1 dargestellt) vielleicht eine noch elegantere Lösung gewesen.

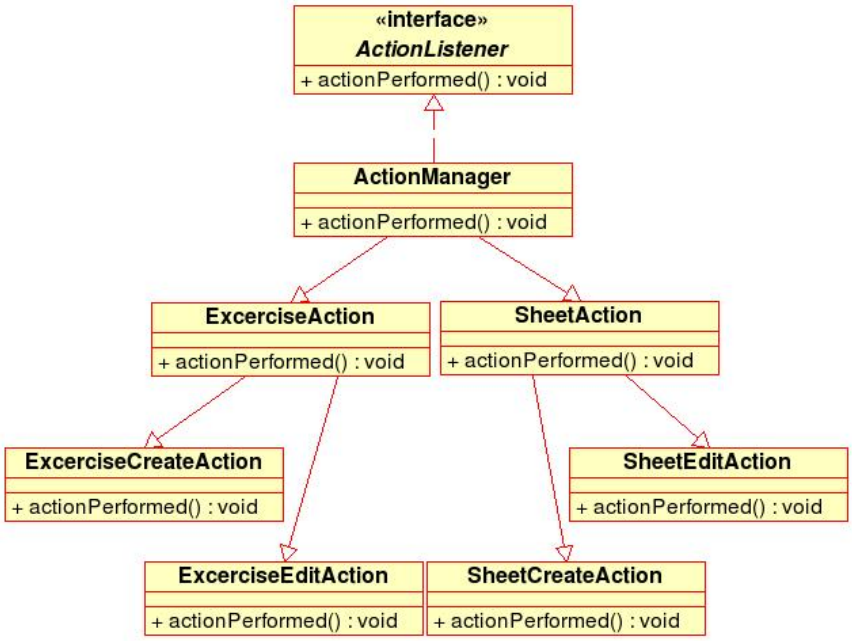

Abbildung 12: Vererbungshierarchie des Controllers

3.5.4 Aufgabe erstellen

Die Komponente zum Erstellen von Aufgaben (*ExcerciseCreate*) war die erste Komponente, die ich entwickelte. Obwohl sie diejenige ist, die die geringste Funktionalität besitzt, war die Fertigstellung am zeitaufwändigsten. Dies lag in erster Linie daran, dass ich währende der Entwicklung ständig Schwachstellen in der Architektur verbesserte und komponentenübergreifende Funktionalität hinzufügte.

Dieser anfängliche Aufwand hat sich jedoch – im nachhinein betrachtet – gelohnt. Die Entwicklung der restlichen Komponenten verlief relativ zügig, da bereits sehr viele generische Klassen und Methoden vorhanden waren. die sofort verwendet werden konnten. Am Programm selbst mußten keine Änderungen, sondern lediglich noch Ergänzungen vorgenommen werden.

Abbildung 13 zeigt einen Screenshot der GUI der Komponente unter *Windows XP*. Zu beachten ist das sehr realistische Erscheinungsbild des *JGoodies-XP-Look-And-Feels* (siehe Kapitel 2.7).

Im Inititialisierungsprozess werden aus der Datenbank alle Schlüsselwörter und Fachgebiete geladen, da diese sofort zur Verfügung stehen müssen. Der Dozent kann Schlüsselwörter und Fachgebiete in einer Combo-Box selektieren und sie mittels der entsprechenden Buttons zu seiner Auswahl hinzufügen, beziehungsweise aus dieser entfernen. Hinzugefügte Schlüsselworte, beziehungsweise Fachgebiete, werden untereinander in einer Text-Area dargestellt.

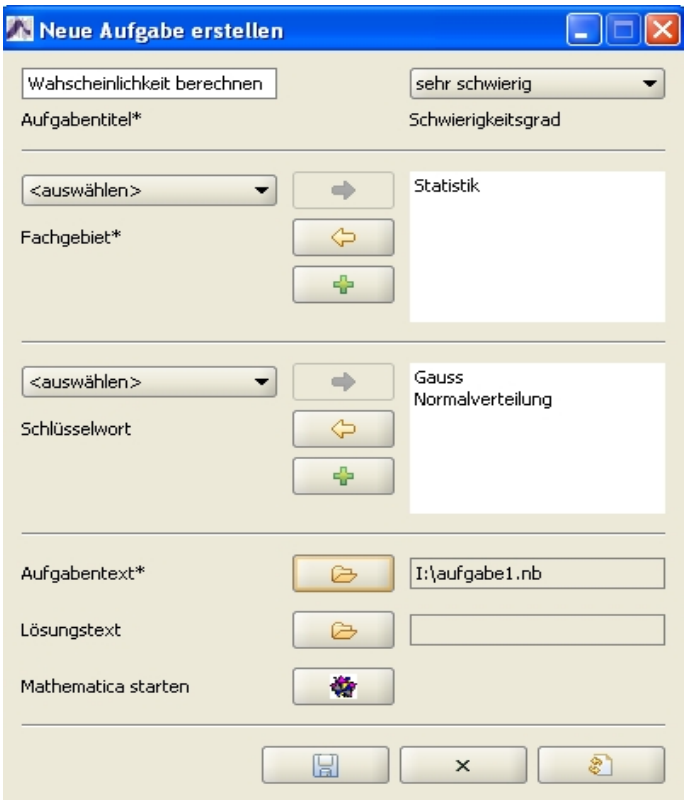

Abbildung 13: Screenshot der GUI von *Aufgabe erstellen*

Wenn ein Fachgebiet hinzugefügt wurde, können in der Combo-Box nur noch die Untergebiete dieses Fachgebiets selektiert werden.

Durch Drücken der Buttons *Neues Schlüsselwort* oder *Neues Fachgebiet* kann ein Dozent neue Fachgebiete, beziehungsweise Schlüsselworte erstellen. Ein neues Fachgebiet wird jeweils als Untergebiet des aktuellen Fachgebiets angelegt. Die Anzahl der Hierarchieebenen ist im Prinzip unbegrenzt.

Wenn ein Schlüsselwort hinzugefügt wurde, erscheint es nicht mehr in

der Combo-Box. Ein Schlüsselwort kann also jeder Aufgabe nur ein Mal zugewiesen werden. Wird ein Wort aus der Auswahl entfernt, steht es in der Combo-Box wieder zur Verfügung.

Sowohl Schlüsselworte als auch Fachgebiete können nur einmal vorhanden sein. Existieren sie bereits, wird eine Fehlermeldung ausgeben. Zwischen Groß- und Kleinschreibung wird nicht unterschieden. Zwei Fachgebiete sind allerdings nur dann gleich, wenn sie das gleiche Obergebiet haben. Ein Fachgebiet namens *Differenzialgleichungen* kann also durchaus mehrfach existieren, jedoch nur einmal als Untergebiet von *Analysis*.

Schlüsselwörter und Fachgebiete stehen jedem Dozenten zur Verfügung. Es hat also nicht jeder Dozent seine eigenen Schlüsselwörter und Fachgebiete. Sollte dieses Programm von vielen Dozenten, über einen längeren Zeitraum verwendet werden, wäre es möglich, dass die Liste der Schlüsselwörter sehr lang und unübersichtlich wird. Es wäre zu überlegen, ob man in diesem Fall die Combo-Boxen mit automatischer Vervollständigung (ähnlich der Adressleiste gängiger Webbrowser) versieht. Diese müssten allerdings selbst entwickelt werden, da *Java-Swing* über keine derartigen Combo-Boxen verfügt.

Der Titel der Aufgabe wird in ein Textfeld eingegeben. Aufgabentitel werden ebenfalls auf Existenz überprüft. Es ist möglich, dass Titel gleich sind, selbst wenn die Aufgaben zum gleichen Fachgebiet gehören, aber nur wenn die entsprechenden Aufgaben von verschiedenen Dozenten erstellt wurden. Aufgaben eines Dozenten dürfen nur dann den selben Titel besitzen, wenn sie unterschiedlichen Fachgebieten zugeordnet sind.

Der Schwierigkeitsgrad kann aus einer Combo-Box ausgewählt werden. Es gibt 4 Schwierigkeitsgrade: *leicht, mittel, schwer* und *sehr schwer*.

Möchte der Dozent eine Aufgabe erstellen, kann er aus der Anwendung heraus *Mathematica* starten. Es muss allerdings (wie in Kapitel 3.7 beschrieben) der Dateipfad zu *Mathematica* gesetzt sein. Wenn er die Aufgabe erstellt hat, speichert er sie ab und startet anschließend durch Drücken des entsprechenden Buttons einen Dateiauswahldialog. Er wählt die zuvor erstellte Datei aus und importiert sie in die Aufgabenverwaltung. Der Pfad wird in einem Textfeld dargestellt. Für die Aufgabenlösung gilt die gleiche Vorgehensweise.

In der Button-Bar befinden sich drei Buttons: *Abbrechen, Speichern* und *Zurücksetzen*. Der Button *Abbrechen* steht immer zur Verfügung, die beiden anderen werden aktiv, wenn sinnvolle Aktionen möglich sind.

- Der Button *Zurücksetzen* löscht alle Eingabefelder und bringt die Combo-Boxen in den Ausgangszustand. Er wird aktiv, sobald sich der Ausgangszustand ändert, also irgendwelche Eingaben gemacht wurden.
- Der Button *Speichern* speichert die Aufgabe in der Datenbank. Er wird aktiv, wenn die Pflichtfelder ausgefüllt sind. Pflichtfelder sind mit einem Stern gekennzeichnet.

Pflichtfelder sind:

- Aufgabentitel
- Fachgebiet
- Aufgabentext

Alle anderen Felder sind Wahlfelder und können auch zu einem späteren Zeitpunkt (mit der Komponente *Aufgabe bearbeiten*) ausgefüllt werden.

Wenn *Speichern* gedrückt wurde, wird automatisch die (von *Abstract-Component* abgeleitete und überschriebene) Methode *validate()* aufgerufen. Sie prüft, ob die Eingaben vollständig und zulässig sind. Sollte ein Dozent beispielsweise versehentlich ein *Word-Dokument* anstatt einer *Notebook*-Datei importiert haben, wird eine Fehlermeldung ausgegeben.

Wenn die Validierung erfolgreich war, werden die Daten innerhalb einer Transaktion in die Datenbank geschrieben und die Komponente beendet sich. Sollte die Transaktion fehlschlagen, führt *Hibernate* einen Rollback durch und es wird eine Fehlermeldung ausgegeben.

3.5.5 Aufgabe bearbeiten

Die Komponente *Aufgabe bearbeiten* (*ExcerciseEdit*) ermöglicht dem Dozenten das bearbeiten bereits vorhandener Aufgaben. Der Funktionsumfang ist ungleich größer, als jener der Komponente *Aufgabe bearbeiten*. Die GUI ist in Abbildung 14 zu sehen.

Im Rahmen der Initialisierung werden alle Schlüsselwörter, Fachgebiete und Aufgaben des Dozenten geladen. Es ist sichergestellt, dass ein Dozent nur seine eigenen Aufgaben bearbeiten kann (und auch angezeigt bekommt), selbst wenn ein Kollege im gleichen Fachgebiet Aufgaben abgelegt hat.

In der linken Hälfte des Frames werden dem Dozenten, in einer hierarchisch gegliederten Baumstruktur, Fachbereiche und Aufgaben dargestellt. Anfangs ist der Baum geschlossen. Der Dozent hat nun einerseits die Möglichkeit, den Baum selbst zu öffnen und manuell Aufgaben auszuwählen. Er kann allerdings jedoch auch mittels Suchfunktion nach Aufgaben suchen.

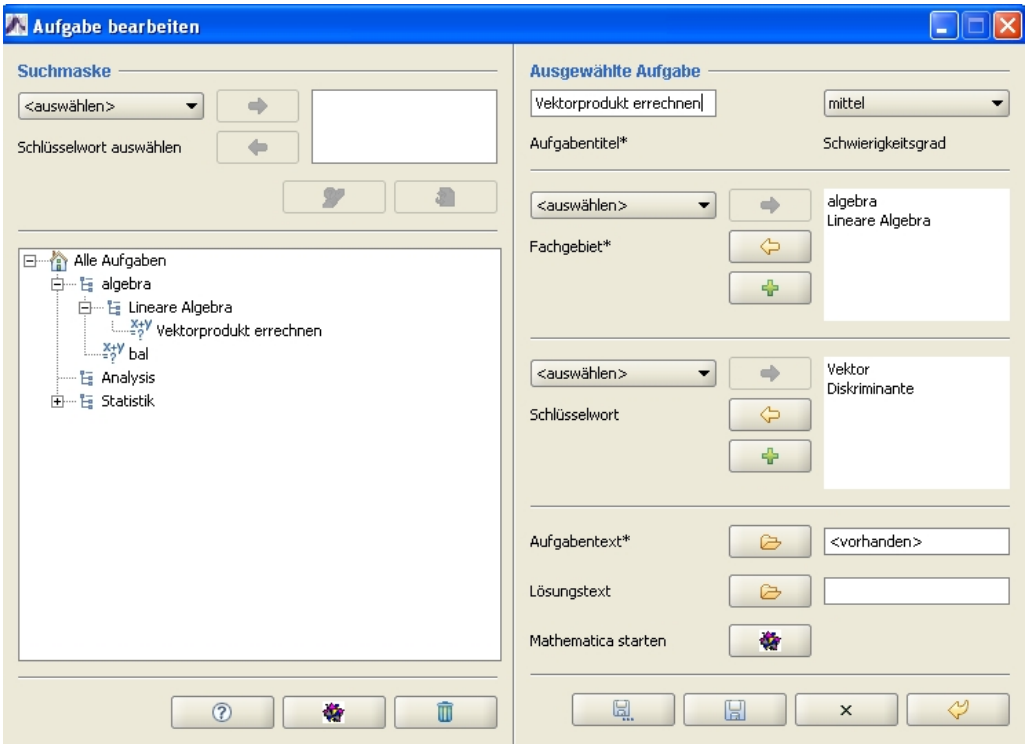

Abbildung 14: Screenshot der GUI von *Aufgabe bearbeiten*

In der Suchmaske, die sich oberhalb des Baums befindet, kann er aus einer Combo-Box Schlüsselworte selektieren. Das Hinzufügen und Entfernen von Schlüsselwörtern funktioniert gleich wie in der Komponente *Aufgabe erstellen* (3.5.4). Deshalb werde ich nicht näher darauf eingehen. Hat er ein oder mehrere Schlüsselwörter ausgewählt, kann er durch Drücken des Buttons *Suchen* die Suche starten. Handelt es sich um mehrere Schlüsselwörter, werden diese *und*-verknüpft.

Im Baum werden jetzt die Suchergebnisse dargestellt. Der Baum ist hierbei aufgeklappt, so dass alle Suchergebnisse für den Dozenten sichtbar sind. Es werden auch nur die Fachgebiete angezeigt, die tatsächlich Sucher-

gebnisse enthalten. Wenn der Dozent mit der Maus eine Aufgabe selektiert, ist die Aufgabe im rechten Panel dargestellt. Er kann nun mehrere Aktionen ausführen:

- Seine ursprünglichen Eingaben ändern oder ergänzen, beispielsweise einen Lösungstext anhängen.
- Sich durch Drücken des Buttons *Information* zusätzliche Informationen zur Aufgabe (wie Erstellungsdatum) in einem Popup-Frame anschauen.
- Die selektierte Aufgabe mit *Mathematica* betrachten (*Mathematica*-Button).
- Die ausgewählte Aufgabe löschen (*Löschen*-Button).

Wenn der Button *Suche-zurücksetzen* gedrückt wird, werden die Einträge in der Suchmaske entfernt. Der Baum ist im ursprünglichen Zustand und enthält wieder alle Aufgaben. Die Button-Bar verfügt im Gegensatz zu *Aufgabe bearbeiten* über einen zusätzlichen Button: *Speichern-unter*. Außerdem wurde der Button *Zurücksetzen* durch den Button *Rückgängig* ersetzt.

- *Speichern* speichert die geänderte Aufgabe in der Datenbank.
- *Speichern-unter* ermöglicht es, die Aufgabe unter einem neuen Namen abzuspeichern. Die ursprüngliche Aufgabe wird also nicht überschrieben. *Speichern* und *Speichern-unter* werden aktiv, sobald etwas an der ursprünglichen Aufgabe geändert wurde und die Pflichtfelder ausgefüllt sind.
- *Rückgängig* stellt die ursprüngliche Aufgabe wieder her. *Rückgängig* wird aktiv, wenn Änderungen vorgenommen wurden.

Sowohl bei *Speichern*, als auch bei *Speichern-unter*, wird die Aufgabe mit einem neuen Zeitstempel versehen.

Nach Drücken von *Speichern*, läuft die gleiche Prozedur ab, wie bei der Komponente *Aufgabe erstellen* (3.5.4), mit dem Unterschied, dass die Methode *validate()* anders implementiert ist. Bei *Speichern-unter* wird die Methode *validateAs()* aufgerufen, die nicht in der Superklasse vorhanden ist, da sie nicht in allen Komponenten benötigt wird. Nach erfolgreichem Speichervorgang beendet sich die Komponente. Andernfalls wird eine Fehlermeldung ausgegeben.

Das Vorgehen beim Durchsuchen und Traversieren von Bäumen ist in Kapitel 3.8.1 näher beschrieben.

3.5.6 Aufgabenblatt erstellen

Mit der Komponente *Aufgabenblatt erstellen* kann ein Dozent aus verschiedenen Aufgaben, ein Aufgabenblatt erstellen. Der Initialisierungsprozess verläuft ähnlich wie bei den vorigen Komponenten, Schlüsselwörter, Fachgebiete und Aufgaben werden geladen, die Eingabefelder im rechten Panel bleiben leer.

Das linke Panel ist auch weitgehend identisch mit dem von *Aufgabe bearbeiten*. Wie in *Aufgabe bearbeiten* (Kapitel 3.5.5), kann ein Dozent entweder selbst den Baum nach einer Aufgabe durchsuchen oder die Suchfunktion benutzen. Der Button *Löschen* steht allerdings nicht zur Verfügung. Löschen von Aufgaben muss in *Aufgabe bearbeiten* durchgeführt werden.

Im linken Panel existiert nun zusätzlich ein Button, mit dem ein Aufgabenblatt mit *Mathematica* geöffnet werden kann. Dieser wird aktiv, wenn ein Blatt Aufgaben enthält und im rechten Panel der Wurzelknoten *Aufgabenblatt* ausgewählt ist.

Jedes Blatt kann mit Metadaten versehen werden. Der Titel des Aufgabenblattes wird in ein Textfeld eingegeben. Für die Gültigkeit eines Titels gelten die gleichen Kriterien, wie bei Aufgabentiteln. Es sind weiterhin Combo-Boxen vorhanden, aus denen Studiengang, Vorlesung und Semester ausgewählt werden können. Wie anhand der Sterne zu Erkennen ist (Abbildung 15), sind alle vier Angaben verpflichtend.

Im Ausgangszustand enthält die Combo-Box *Studiengang* alle Studiengänge, die Combo-Box *Vorlesung* alle Vorlesungen. Wird ein Studiengang ausgewählt, ändert sich der Inhalt der Combo-Box *Vorlesung* und es stehen nur noch die Vorlesungen zur Verfügung, die zum selektierten Studiengang gehören. Bei Auswahl einer Vorlesung im Ausgangszustand wird der Studiengang automatisch gesetzt.

In der Mitte des Frames sind untereinander die Buttons *Oben, Unten, Hinzufügen* und *Entfernen* angeordnet. Die letzten beiden verhalten sich äquivalent wie jene zum Hinzufügen und Entfernen von Schlüsselwörtern aus Kapitel 3.5.4 und 3.5.5. Ebenso führt die Aufnahme einer Aufgabe in ein Aufgabenlatt dazu, dass diese im linken Panel nicht mehr verfügbar ist. Wird eine Aufgabe entfernt, ist sie wieder im Baum vorhanden und steht zur Auswahl. Es ist also sichergestellt, dass eine Aufgabe nicht mehrmals in einem Blatt vorhanden sein kann. Mit den Buttons *Oben* und *Unten* kann

die Reihenfolge der Aufgaben verändert werden. Sie werden aktiv, wenn eine Aufgabe ausgewählt und ein Verschieben in diese Richtung möglich ist. Das heißt, sollte eine Aufgabe bereits an erster Stelle stehen, wird der Button *Oben* nicht aktiviert. Das gleiche gilt für den gegenteiligen Fall.

Die Buttons *Speichern*, *Abbrechen* und *Zurücksetzen* haben die gleiche Funktionalität, wie jene in *Aufgabe erstellen*. Ihrer Aktivierung liegt auch die gleiche Logik zugrunde.

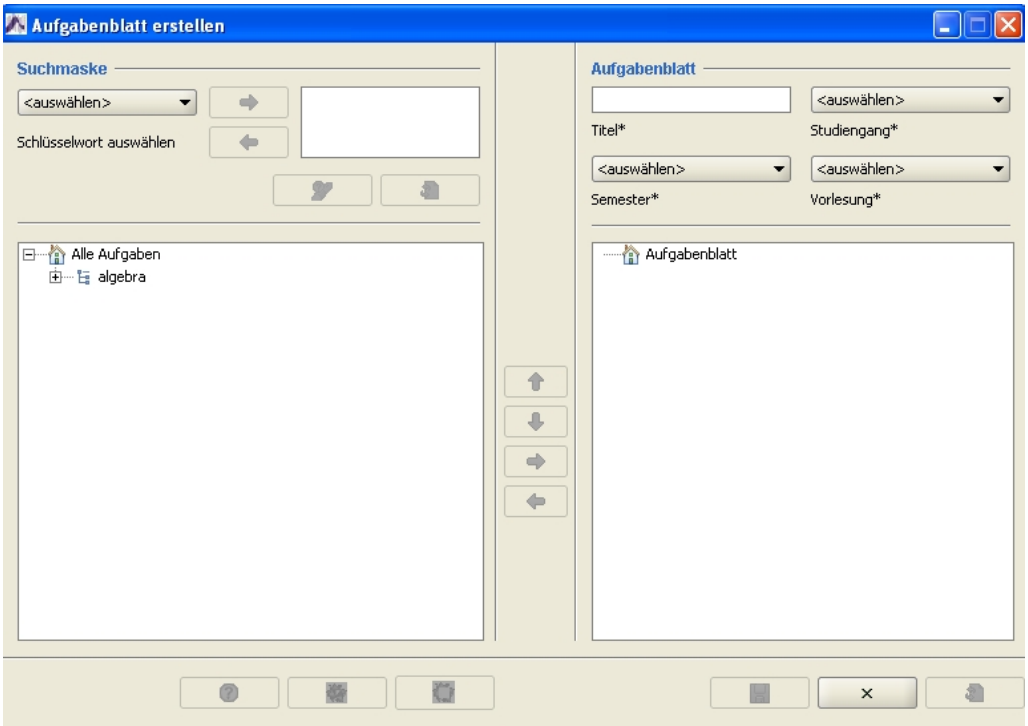

Abbildung 15: Screenshot der GUI von *Aufgabenblatt erstellen*

Da die Nummer einer Aufgabe einem Blatt zugeordnet werden muss, ist die Collection der Persistenzklasse *Sheet* eine *HashMap*, diese ermöglicht es, für jedes enthaltene Objekt einen Key anzulegen. *Hibernate* sorgt dafür, das dieser Key in der Verknüpfungstabelle *EXCERCISE_SHEET* in die Spalte *NUMBER* eingetragen wird.

3.5.7 Aufgabenblatt bearbeiten

Die Komponente *Aufgabenblatt bearbeiten* ist in Bezug auf Funktionalität die komplexeste der vier Komponenten. Das rechte Panel ist – bis auf geringe Veränderungen im Design (siehe Kapitel 3.9) und eine veränderte Button-Bar – gleich wie jenes von *Aufgabenblatt erstellen*. Im Ausgangszustand sind Textfeld und Combo-Boxen deaktiviert. Auch die vier Buttons zum Hinzufügen, Entfernen und Verschieben der Aufgaben entsprechen denen von Komponente *Aufgabenblatt erstellen*.

Da alle Aufgabenblätter nach dem Start der Komponente vorhanden sein müssen, werden diese zusätzlich bei der Initialisierung geladen.

Auf der linken Seite befinden sich zwei Panels, von denen jeweils nur eines sichtbar ist. Die Auswahl des Panels erfolgt über Karteireiter, die sich unterhalb der Titelleiste befinden. Im Ausgangszustand ist das Panel *Aufgabenblatt suchen* sichtbar. Das andere Panel, *Aufgabe suchen* ist derweil unsichtbar und inaktiv. Das heißt, der Reiter kann nicht selektiert werden.

Im dargestellten Baum befinden sich alle Aufgabenblätter des Dozenten, geordnet nach Vorlesung und Studiengang. Wenn der Dozent nicht selbst durch den Baum navigieren möchte, um ein Aufgabenblatt zu suchen, kann er in der Suchmaske Begriffe eingeben, nach denen die Aufgabenblätter mittels Volltextsuche durchsucht werden. Die Begriffe werden *und*-verknüpft.

Entsprechend der Suche nach Aufgaben, öffnet sich bei erfolgreicher Suche der Baum und die Suchergebnisse werden dargestellt. Wird ein Aufgabenblatt ausgewählt, werden Metadaten und Inhalt des Blattes im rechten Panel dargestellt. Bei Änderungen verhalten sich die Buttons der Button-Bar gleich wie diejenigen von Komponente *Aufgabe bearbeiten*.

Bei ausgewähltem Aufgabenblatt wird der zweite Karteireiter aktiviert. Wird dieser selektiert, steht dem Dozenten die bereits aus Kapitel 3.5.5 und 3.5.6 bekannte *Suche nach Aufgaben* zur Verfügung.

Das linke Panel besitzt einen zusätzlichen Button, mit dem ein Aufgabenblatt gelöscht werden kann. Dieser wird aktiv, sobald ein Aufgabenblatt ausgewählt ist.

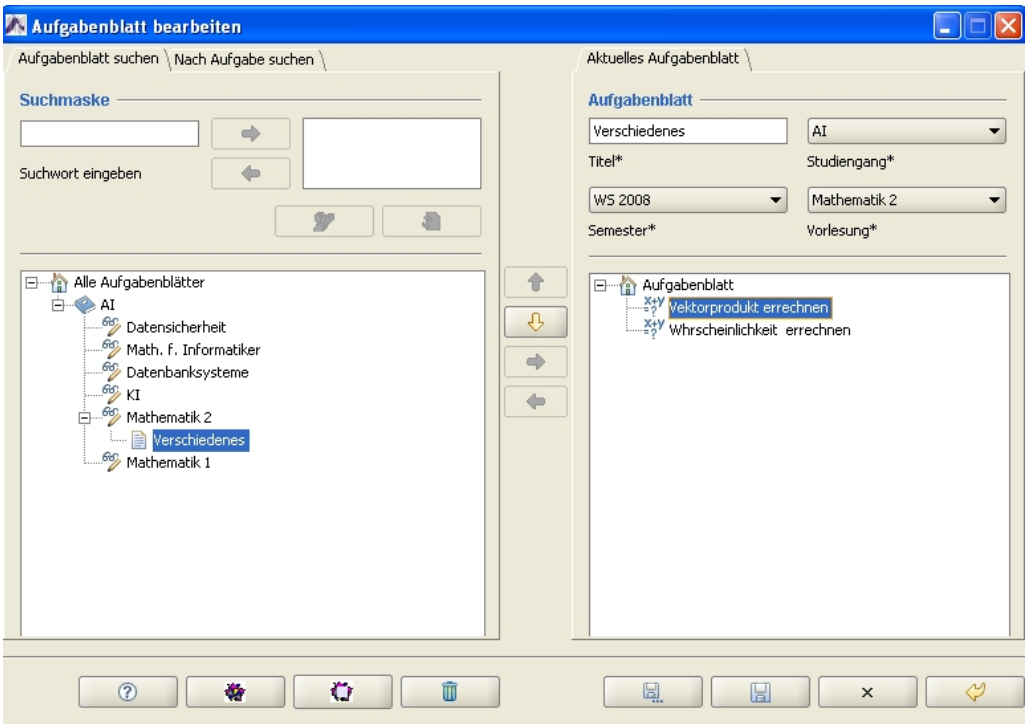

Abbildung 16: Screenshot der GUI von *Aufgabenblatt bearbeiten*

In dieser Komponente ist, genauso wie in *Aufgabenblatt erstellen*, Mehrfachauswahl möglich. Das heißt, wenn im linken Panel eine Aufgabe und im rechten ein Aufgabenblatt ausgewählt ist (oder umgekehrt), sind alle vier Buttons des linken Panels aktiv.

3.6 Die Tool-Klassen

Wie in Kapitel 3.3.2 bereits angedeutet, sind die Klassen aus dem Paket *math.manager.tools* nach dem *Singleton*-Pattern erstellt. Es existiert also immer nur eine Instanz. Wie in Abbildung 7 aus Kapitel 3.1 zu sehen ist, werden die Dienste der Toolklassen von der *Fachlogikschicht* (also *Model* und *Controller*) in Anspruch genommen. Da die Toolklassen selbst Programmlogik enthalten, sind sie ebenfalls dieser Schicht zuzuordnen. Die wichtigsten Klassen sind:

- Die Klasse *NotebookParser*
  Sie ist zuständig für alle Operationen, die das *Notebook*-Format betreffen. Da diese Klasse etwas umfangreicher ist, wird sie in Kapitel 3.6.2 näher beschrieben.

- Die Klasse *FormatTool*
  Sie stellt Methoden für Format-Konvertierungen zur Verfügung. So ist beispielsweise der Schwierigkeitsgrad einer Aufgabe in der Datenbank als Integer abgelegt. Zur Darstellung in einer Combo-Box ist jedoch eine String-Repräsentation nötig. Derartige Konvertierungen werden von FormatTool durchgeführt.

- Die Klasse *PropertyManager*
  Alle Zeichenketten, die im Programm verwendet werden (Menüeinträge, Beschriftung von GUI-Komponenten, Fenstertitel,. . .) sind in externen Property-Files ausgelagert. Die Methode *getProperty(String key)* liefert den entsprechenden Wert zum übergebenen Schlüssel (siehe 3.7).

- Die Klassen *ErrorDialog* und *InformationDialog*
  Sie funktionieren nach dem gleichen Prinzip wie *PropertyManager* und liefern Fehler- und Informationsmeldungen aus den entsprechenden Property-Files.

3.6.1 Die Klasse HibernateUtil

Die Klasse *HibernateUtil* wurde anhand der Empfehlungen der *Hibernate*-Referenz-Dokumentation [HTa] erstellt. Die Verwendung des *Singleton*-Patterns (wie in Listing 6, Kapitel 3.3.2 beschrieben) ist hier nicht möglich, da bei abgelaufener Session ein neues Objekt erzeugt werden muss. Dies wird dadurch erreicht, dass die Methode wie folgt implementiert wird:

```
1  public Class Singleton2 {
2      // diesmal nicht static
3      private Singleton2 instance = null;
4      // Privater Konstruktor um 'eigenmächtiges'
5      // Initialisieren zu verhindern
6      private Singleton2() {}
7      // wenn instance null, erzeuge neu,
8      // sonst liefere instance
9      public getInstance() {
10         if (instance == null) {
11             instance = new Singleton2();
12         }
13         return instance;
14     }
15 }
```

Listing 7: Singleton-Implementierung 2

Mit dieser Variante ist gesichert, dass nur dann ein neues Objekt erzeugt wird, wenn das aktuelle *null* ist. Ansonsten wird die bestehende Instanz zurückgegeben.

Dennoch wäre diese Klasse nicht brauchbar, wenn eine Applikation mehrere Threads verwendet, da eine *Hibernate-Session* nicht threadsafe, also nicht reentrant ist. Es gibt jedoch die Möglichkeit eine *ThreadLocal*-Variable zu benutzen. Eine ThreadLocal-Variable unterscheidet sich von normalen Variablen dadurch, dass jeder Thread über eine unabhängig initialisierte Kopie dieser Variablen verfügt. Außerdem kann eine *ThreadLocal*-Variable ein beliebiges Objekt aufnehmen, also beispielsweise eine *Hibernate-Session*.

Die Klasse *HibernateUtil* verzichtet auf eine *ThreadLocal*-Variable, da Threadsicherheit in der Aufgabenverwaltung nicht erforderlich ist.

3.6.2 Die Klasse NotebookParser.java

In *Mathematica* wird mit Zellen (*Cells*) gearbeitet. Es gibt verschiedene Arten von Zellen, beispielsweise *In*-Zellen, für die Eingabe von *Mathematica*-Befehlen, *Text*-Zellen, für „normalen“ Text, sowie auch Zellen für formatierten Text, wie er etwa für Überschriften verwendet wird (*Section*). Dies spiegelt sich auch im **.nb*-Format wieder.

In Listing 8 ist eine einfache *Notebook*-Datei dargestellt. Ich beschränke mich auf den eigentlichen Inhalt der Datei. *Mathematica* speichert noch mehr Information in der Datei ab, beispielsweise die *Mathematica*-Version, das verwendete Betriebssystem, Angaben zur aktuellen Fensterposition und einen Hashwert, aus dem *Mathematica* erkennen kann, ob die Datei außerhalb von *Mathematica* modifiziert wurde. Außerdem habe ich die Datei umformatiert, um durch Einrückungen die Verschachtelung der einzelnen Elemente hervorzuheben.

```
1  Notebook[{
2    Cell[
3      BoxData[
4        FormBox[
5          StyleBox[\(Das\ ist\ Text\),"Text"
6          ], TraditionalForm
7        ]
8      ], "Input"
9    ]
10 },
11 ]
```

Listing 8: Einfache Notebook-Datei

Es wird deutlich, dass abgesehen davon, dass andere Tags verwendet werden, die Struktur durchaus mit Formaten, wie beispielsweise *XML*, verglichen werden kann. Den äußeren Rahmen bildet das *Notebook*-Element. Die weiteren Elemente sind hierarchisch darunter angeordnet. In Zeile 5 ist erkennbar, dass es sich um den Typ *Text* handelt.

Listing 9 zeigt eine Datei mit zwei einfachen Berechnungen. Hier wurden keine Formatveränderungen vorgenommen. Es handelt sich um die Form, in der *Mathematica* die Aufgabe speichert. Die hierarchische Struktur wird dennoch deutlich. In den Zeilen 6 und 13 ist zu beachten, dass das Komma am Ende fehlt. Dadurch wird signalisiert, dass es das letzte Element im Parent-Element ist.

```
Notebook[{
    Cell[CellGroupData[{
        Cell[BoxData[
            \(5 + 4\)], "Input"],
        Cell[BoxData[
            \(9\)], "Output"]
    }, Open ]],
    Cell[CellGroupData[{
        Cell[BoxData[
            \(7 + 10\)], "Input"],
        Cell[BoxData[
            \(17\)], "Output"]
    }, Open ]]
},
]
```

Listing 9: Notebook-Datei mit Berechnungen

Die *Cell*-Elemente auf höchster Hierarchieebene aus Abbildung 8 und 9 könnten nun einfach hintereinander gehängt und mit einem *Notebook*-Element umschlossen werden. Es müsste sichergestellt sein, dass am Ende jedes Elements einer Hierarchieebene ein Komma steht – mit Ausnahme des letzten Elementes. Dies war im Prinzip meine Vorgehensweise.

Da jedoch der Titel von Aufgabe und Aufgabenblatt, sowie die Aufgabennummer ebenfalls automatisch eingefügt werden sollten war es notwendig, zusätzliche Elemente einzubauen. Ich habe hierzu das Element *Section* für Titel des Aufgabenblattes und *Subsection* für Aufgabentitel verwendet, da mir dies optisch am ansprechendsten schien.

Ich muss an dieser Stelle allerdings anmerken, dass die Darstellung auf einem *Windows*-Rechner eventuell ganz andere Ergebnisse liefert. Unter *Linux* verwendet *Mathematica* zur Grafikdarstellung die *Motif*-Grafikbibliothek. Diese wurde vor über 15 Jahren entwickelt, ist (optisch) seither nahezu unverändert geblieben und kann daher – was die Qualität der Darstellung anbelangt – kaum noch als zeitgemäße Grafikbibliothek bezeichnet werden. Sie wird in der Regel auch nur noch auf älteren Systemen benutzt (siehe [wikb]). Da ich, wie ich bereits in Kapitel 2.3 bemerkte, nur eine *Mathematica*-Einzelplatzlizenz für *Linux* habe, die *Mathematica*-Anbindung unter *Windows* also nicht testen konnte, kann ich daher nicht garantieren, dass meine Wahl der Schriftstile die richtige ist. Sollte das augenblickliche Erscheinungsbild nicht den Ansprüchen genügen, müssten

eventuell in der Konfigurationsdatei andere Tags deklariert werden. Näheres zu den Konfigurationsdateien folgt in Kapitel 3.7.

Wenn ein Dozent eine Aufgabe erstellt, in die Aufgabenverwaltung importiert und den *Speichern*-Button gedrückt hat, passiert folgendes:

- Die Datei wird geöffnet.
- Alle Kommentare und Metadaten werden entfernt.
- Das *Notebook*-Element wird entfernt (sowohl beginnender, als auch schließender Tag).
- Der String wird in die Datenbank geschrieben.

Das Anzeigen eines Aufgabenblattes ist ein etwas umfangreicherer Vorgang. Die Aufgabenblätter sind nicht als String in der Datenbank gespeichert, sondern werden jeweils zur Laufzeit erzeugt. Wie bereits in diesem Kapitel erwähnt, müssen für die Aufbereitung der einzelnen Aufgaben zusätzliche Elemente eingefügt werden. Das *Section*- und das *Subsection*-Element haben den gleichen Aufbau. Deshalb wird nur das *Section*-Element kurz beschrieben. Das beginnende Tag hat folgende Gestalt:

```
Cell[CellGroupData[{Cell["Der Titel","Section"],
```

Da der Titel von der Aufgabenverwaltung eingefügt wird, wird er in zwei Teile zerlegt. Außerdem muss beachtet werden, dass manche Sonderzeichen mittels eines *Backslashs* maskiert werden müssen. Es entstehen also folgende zwei Strings:

```
Cell\[CellGroupData\[\{Cell\[\"
```

und

```
\",\"Section\"\],
```

Das abschließende Tag lautet:

```
\} \, Open \]\]
```

Beim Erstellen eines Aufgabenblattes werden also nacheinander folgende Elemente an einen String angehängt:

- Der erste Teil des beginnenden Tags der Section
- Der Titel des Aufgabenblattes
- Der zweite Teil des beginnenden Tags der Section

Dann für jede Aufgabe des Aufgabenblattes:

- Der erste Teil des beginnenden Tags der Subsection
- Die Nummer der Aufgabe
- Der Titel der Aufgabe
- Der zweite Teil des beginnenden Tags der Subsection
- Der Aufgabentext
- Eine Leerzeile
- Wenn nicht letzte Aufgabe und keine Lösung vorhanden: Ein Komma
- Wenn Lösung vorhanden, der erste Teil des beginnenden Tags der Subsection
- Der String „Musterlösung zu Aufgabe“
- Die Nummer der Aufgabe
- Der zweite Teil des beginnenden Tags der Subsection
- Der Lösungstext
- Eine Leerzeile
- Wenn nicht Lösung von letzter Aufgabe: Ein Komma
- Das schließende Tag der Subsection

Am Ende wird das schließende Tag der Section angefügt, die Datei temporär abgespeichert und *Mathematica* übergeben.

Einen Fehler hatte ich beinahe übersehen. Ich entdeckte ihn nur zufällig:

Mathematica verwendet für deutsche Sonderzeichen keine *Unicode-Entities* oder ähnliches, sondern hat auch hier ein ganz eigenes Format. Der Buchstabe *Ö* wird beispielsweise repräsentiert durch:

```
\[CapitalODoubleDot]
```

Ich wurde darauf aufmerksam, als ich die Volltextsuche implementierte und bestimmte Worte nicht gefunden wurden. Es musste also eine entsprechende Konvertierung erfolgen, welche die Klasse *FormatTool* (siehe 3.6) vornimmt.

3.7 Konfigurationsdateien

Alle selbst erstellten Konfigurationsdateien sind als *Java-Property-Files* ausgeführt (siehe [SM]). Die Einträge sind nach folgendem Schema aufgebaut:

```
<Schlüssel> = <Wert>
```

Java bietet die Möglichkeit, mit Hilfe der Klasse *ResourceBundle*, auf komfortable Art und Weise auf die Datei und deren Einträge zuzugreifen. Die Datei muss hierbei nicht explizit geöffnet werden. Dies wird von API-Funktionen erledigt. Beispielsweise würde der Zugriff auf ein Property-File namens *my.properties*, das sich im Paket *xy* befindet, so erfolgen:

```
ResourceBundle rb = ResourceBundle.getBundle("xy.my");
```

Ebenso komfortabel ist der Zugriff auf die einzelnen Properties:

```
String wert = rb.getString(schlüssel);
```

Wie bereits im vorherigen Kapitel erwähnt, ist das Programm umfangreich konfigurierbar, ohne dass es neu kompiliert werden muss. Alle Menüeinträge, Tooltips, Beschriftungen von GUI-Komponenten, Fenstertitel und Icons sind veränderbar, ohne dass hierfür auch nur in einer einzigen Klasse Veränderungen vorgenommen werden müssen. Damit die Übersichtlichkeit erhalten bleibt, sind die zum „normalen“ Betrieb wichtigsten Konfigurationseinstellungen in einer einzigen Datei, der Datei *user.properties* (Kapitel 3.7.2), zusammengefasst. Diese Datei *muss* bearbeitet werden, damit das Programm lauffähig ist.

Die anderen Konfigurationsdateien können ebenfalls editiert werden, jedoch nicht notwendigerweise, wenn man mit den Default-Werten zufrieden ist. Sie sind nicht auf den ersten Blick sichtbar, da sie sich im Archiv *aufgabenverwaltung.jar* befinden. Sie liegen im Paket *math.manager.configuration*. Da es sich beim Format *jar* im Grunde genommen um *zip*-Dateien handelt, muss dieses Archiv nur entpackt, die Dateien bearbeitet und das Archiv neu erstellt werden.

Folgende Veränderungen können vorgenommen werden:

- *error.properties*:
  Texte der Fehlermeldungen, die ausgegeben werden
- *information.properties*:
  Informations- und Statusmeldungen
- *icon.properties*:
  Die vom Programm verwendeten Icons
- *tooltip.properties*:
  Die Tooltips zu den einzelnen GUI-Komponenten

Veränderungen an diesen Dateien beeinträchtigen die Lauffähigkeit des Programms prinzipiell nicht. Allerdings könnte Verwirrung entstehen, wenn man beispielsweise die Icons und Tooltips der Buttons *Speichern* und *Löschen* vertauscht. Allerdings ist dies eigentlich nur mutwillig möglich, da die Namen der Key-Value-Paare eindeutig sind.

Veränderungen an der Datei *configuration.properties* (siehe 3.7.1) können die Funktionalität jedoch sehr wohl gefährden.

Diese Vorgehensweise, alle Zeichenketten in externe Dateien auszulagern, habe ich während meines Praktikums kennengelernt. Ein großer Vorteil daran ist, dass sich so sehr einfach andere Sprachen in das Programm einbinden lassen (siehe [SM]).

Weiterhin gibt es Konfigurationen, mit denen nicht die Aufgabenverwaltung direkt, jedoch beteiligte Komponenten konfiguriert werden können. Dies betrifft *Hibernate* (3.7.3) und *Log4J* (3.7.4).

3.7.1 Die Datei configuration.properties

Die Datei *configuration.properties* enthält alle Zeichenketten, die das Programm im Ablauf benötigt. Dazu gehören:

- Pfadangaben:
  Alle Pfade, außer dem Pfad zu *Mathematica*, der vom Benutzer eingestellt werden muss (siehe 3.7.2). Dazu gehören unter anderem die relativen Pfade zu den Verzeichnissen *tmp, icons* und *bin*.

- Action-Commands:
  Die Kommandos, die den einzelnen GUI-Komponenten zugewiesen sind und anhand derer bestimmte Aktionen durchgeführt werden.

- Look-And-Feels:
  Die zur Darstellung der GUI unter *Windows*, beziehungsweise *Linux*, verwendeten *Look-And-Feels*.

- Properties der Klasse *NotebookParser*:
  Die Zeichenketten, die benötigt werden, um ein *Mathematica-Notebook-Dokument* zu parsen, beziehungsweise zu erstellen.

- Properties der Klasse *FormatTool*:
  Beispielsweise die String-Repräsentationen der einzelnen Schwierigkeitsgrade von Aufgaben (siehe 3.6).

Das Ändern dieser Werte sollte mit Bedacht vorgenommen werden, da es zur Folge haben kann, dass das Programm (zumindest teilweise) nicht mehr funktioniert. Kritisch ist hierbei vor allem das Ändern der Properties des *NotebookParsers*. Es kann jedoch notwendig sein, wenn die Darstellung der Aufgabenblätter nicht zufriedenstellend ist (siehe 3.6.2). Allerdings ist auch bei den anderen Properties Vorsicht geboten:

- Werden bei den Action-Commands mehreren Schlüsseln die gleichen Werte zugewiesen, oder gar Paare gelöscht, führt dies zu Fehlfunktionen. Gleiches gilt für die Properties der Klasse *FormatTool*.

- Wenn Pfadangaben geändert werden, ohne die entsprechenden Verzeichnisse auch tatsächlich zu verschieben, wird das Programm entweder nicht mehr starten (*bin*-Verzeichnis), keine Icons mehr anzeigen (*icons*-Verzeichnis) oder es kann kein Aufgabenblatt mehr dargestellt werden (*tmp*-Verzeichnis).

- Nicht alle *Look-And-Feels* werden auf allen Plattformen unterstützt.

3.7.2 Die Datei user.properties

Die Datei *user.properties* ist speziell für den Anwender gedacht. Er soll alle zum Betrieb wichtigen Einstellungen in einer zentralen Datei vornehmen können. Ähnlich sieht die Lösung bei einigen Serverdiensten aus (zum Beispiel *MySQL*): Es gibt zwar mehrere Konfigurationsdateien, im normalen, nicht professionellen Betrieb muss man jedoch nur eine Datei anpassen. Da die Datei editiert werden muss, ist sie auch sichtbar im Hauptverzeichnis des Programms.

Folgende Einstellungen sind möglich:

- *MATH_MANAGER_PATH_TO_MATHEMATICA*:
  Pfad zu *Mathematica*. Muss als absoluter Pfad eingegeben werden. Es ist auch wichtig, dass das *Notebook*-Frontend benutzt wird, also *mathematica* (unter *Windows*: *mathematica.exe*) anstatt *math* (*math.exe*).

- *MATH_MANAGER_DB_DIALOG*:
  Wenn diese Property auf den Wert *true* gesetzt ist, wird beim Programmstart ein Passwortdialog angezeigt. Hat sie den Wert *false*, wird der Passwortdialog umgangen. Passwort und Benutzername werden dann aus den entsprechenden Properties in dieser Datei ausgelesen. Es liegt also im Ermessen des Anwenders, ob er sich für die eher komfortable oder die eher sichere Variante entscheidet.

- *MATH_MANAGER_DB_USER*:
  Der Benutzername für die Datenbankverbindung. Wenn *MATH_MANAGER_DB_DIALOG* den Wert *true* hat, wird diese Property nicht ausgelesen, ihr muss dann kein Wert zugewiesen sein.

- *MATH_MANAGER_DB_PW*:
  Das Passwort für die Datenbankverbindung. Hier gilt Gleiches wie beim Benutzernamen.

- *MATH_MANAGER_DB_URL*:
  Der Connection-String, mit dem die Datenbankverbindung aufgebaut wird.

- *MATH_MANAGER_DB_DRIVER*:
  Der zu verwendende *JDBC*-Treiber (Klassenname). Der Treiber muss sich im *lib*-Verzeichnis der Anwendung befinden.

- *MATH_MANAGER_SQL_DIALECT*:
  Der zu verwendende *SQL*-Dialekt. Er muss mit dem verwendeten Treiber übereinstimmen (siehe Tabelle 1).

Die Datei ist mit ausführlichen Kommentaren versehen, sodass die Konfiguration sehr einfach durchzuführen ist.

3.7.3 Konfiguration von Hibernate

Die globale Konfigurationsdatei von *Hibernate* ist die Datei *hibernate.cfg.xml*, welche sich im Archiv *aufgabenverwaltung.jar* befindet. Sie muss allerdings nur dann editiert werden, wenn sich das Datenbankschema oder die Persistenzklassen ändern. Es müssten dann die neu hinzugekommenen Mapping-Dateien deklariert oder nicht mehr vorhandene entfernt werden. Neben der Deklaration der Mapping-Dateien steht in dieser Datei nur noch ein einziger Eintrag, nämlich die Anzahl der verwalteten *JDBC*-Verbindungen. Da es sich um eine Standalone-Anwendung handelt, kann der voreingestellte Wert „1“ beibehalten werden (siehe 2.2.4).

Alle anderen konfigurierbaren Properties bezüglich Datenbankverbindung, wurden aus Gründen der Benutzerfreundlichkeit in die Datei *user.properties* ausgelagert (3.7.2).

Es können bei *Hibernate* aber auch Strategien für die Durchführung von Schreib- und Lesevorgängen festgelegt werden. Dies könnte einerseits durch Ändern der Mapping-Dateien erfolgen, die sich im Paket *math.manager.persistence* befinden. Es wäre so kein neuerliches Kompilieren notwendig, da die Dateien zur Laufzeit eingelesen werden. In diesem Fall halte ich das aber nicht für empfehlenswert. Der Umweg über *XDoclet* (2.9.2) wäre weit weniger fehleranfällig und würde das Programm konsistent halten. Es müsste dann allerdings neu kompiliert werden.

Sollen sich nicht nur Strategien, sondern auch Relationen ändern, hätte dies Auswirkungen auf das komplette Programm zur Folge.

Folgende Voreinstellungen habe ich hergestellt:

- Kaskadierung:
  Bei allen Relationen auf *save-update*. *Delete* mit in die Kaskadierung aufzunehmen ist riskant (siehe 2.2.3) und teilweise auch nicht sinnvoll: Wenn ein Aufgabenblatt gelöscht wird, sollten nicht automatisch alle enthaltenen Aufgaben gelöscht werden.

- Lazy-Loading:
  Bei allen Relationen auf *true*. Es werden beim Start einer Komponente stets nur die benötigten Relationen geladen. Selbst bei geringer Tabellenanzahl kann ein Objektgraph schnell sehr umfangreich werden, wenn die Tabellen viele Datensätze enthalten. Die *Hibernate*-Referenz-Dokumentation ([HTa]) empfiehlt ebenfalls diese Einstellung.

Für eine vollständige Auflistung aller möglichen Konfigurationsoptionen verweise ich auf die *Hibernate*-Referenz-Dokumentation [HTa].

3.7.4 Konfiguration von Log4J

Log4J wird über die Datei *log4j.properties* (Paket *math.manager*) konfiguriert. Es kann entweder ein Default-Log-Level für das ganze Programm gesetzt werden (siehe Kapitel 2.9.6) oder ein eigener Level für verschiedene Pakete und Klassen. Da das Programm jedoch nicht mehr im Entwicklungszustand ist, habe ich als Default-Level *Warn* eingestellt. Systemkritische Ausgaben werden weiterhin erfolgen, aber keine Mitteilungen, die vielleicht nur für Entwickler wichtig sind. Der Log-Level für *Hibernate* kann unabhängig gesetzt werden. Er hat momentan ebenfalls den Wert *Warn*.

Über ein Pattern ist es auch möglich, das Format der Logging-Ausgabe festzulegen. In [Sch] ist beschrieben, wie sich ein solches Pattern zusammensetzt und wie damit Inhalte ausgegeben werden können. Das von mir erstellte Pattern beinhaltet:

- Den Log-Level
- Klassenname
- Zeile im Quellcode
- Meldung
- Uhrzeit mit Genauigkeit in Millisekunden

3.8 Algorithmen

3.8.1 Traversieren von Bäumen

Während der Entwicklung des Programms wurde ich mehrmals mit dem Problem konfrontiert, durch Bäume navigieren zu müssen:

- Konstruktion des Baums, der Fachgebiete und Aufgaben darstellt (3.5.5).
- Konstruktion des Baums, der Studiengänge, Vorlesungen und Aufgabenblätter darstellt (3.5.7).
- Suchen nach einzelnen Knoten.
- Hinzufügen und Entfernen einzelner Knoten.
- Entfernen unbenötigter Knoten (wenn ein Fachgebiet keine Aufgaben besitzt, muss es nicht dargestellt werden).
- Setzen des darzustellenden Pfades (wenn eine Suche erfolgt ist, sollen die Suchergebnisse in einem aufgeklappten Baum dargestellt werden).

Der Baum, der Studiengänge, Vorlesungen und Aufgabenblätter anzeigt, verfügt über maximal drei Ebenen. Die beschriebenen Problemstellungen konnten also sehr einfach durch Anwendung iterativer Algorithmen gelöst werden.

Der andere Baum hat jedoch eine beliebige Tiefe, da die Hierarchieebenen bei Fachgebieten unbegrenzt sind.

In der Vorlesung *Künstliche Intelligenz* bei Herrn Professor Dr. Ertel habe ich die Verfahren *Tiefensuche* und *Breitensuche* kennengelernt. Bei Tiefensuche wird rekursiv der jeweils erste Nachfolgerknoten extrahiert, bis ein Blattknoten erreicht ist. Bei Breitensuche werden nacheinander alle Nachfolgerknoten der jeweiligen Ebene durchsucht. Gäbe es nur ein Fachgebiet, das keine Untergebiete, sondern nur Aufgaben enthält, hätten beide Verfahren die gleiche Effektivität, da sie die Knoten in gleicher Reihenfolge abarbeiten würden.

Bei einer Struktur wie in Abbildung 17 wäre die Tiefensuche – bei der Suche nach Aufgaben – im Vorteil (Fachgebiete sind blau, Aufgaben orange dargestellt). Wie zu Sehen ist, müssten bei Breitensuche 5 Knoten betrachtet werden, bevor die erste mögliche Lösung gefunden wird. Bei

Tiefensuche wären nur drei Knoten notwendig. Anders verhält es sich in Abbildung 18. Bei Breitensuche ist der dritte extrahierte Knoten die erste Möglichkeit, die gesuchte Aufgabe zu finden. Tiefensuche würde erst als vierten Knoten erstmals eine Aufgabe betrachten.

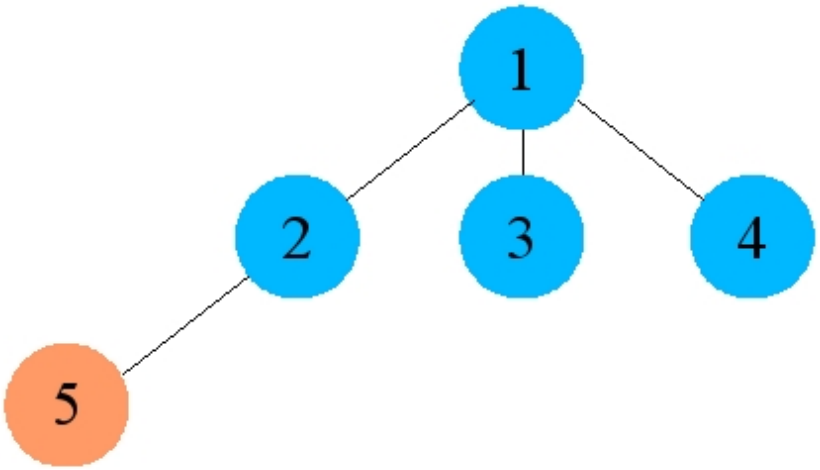

Abbildung 17: Breitensuche 1

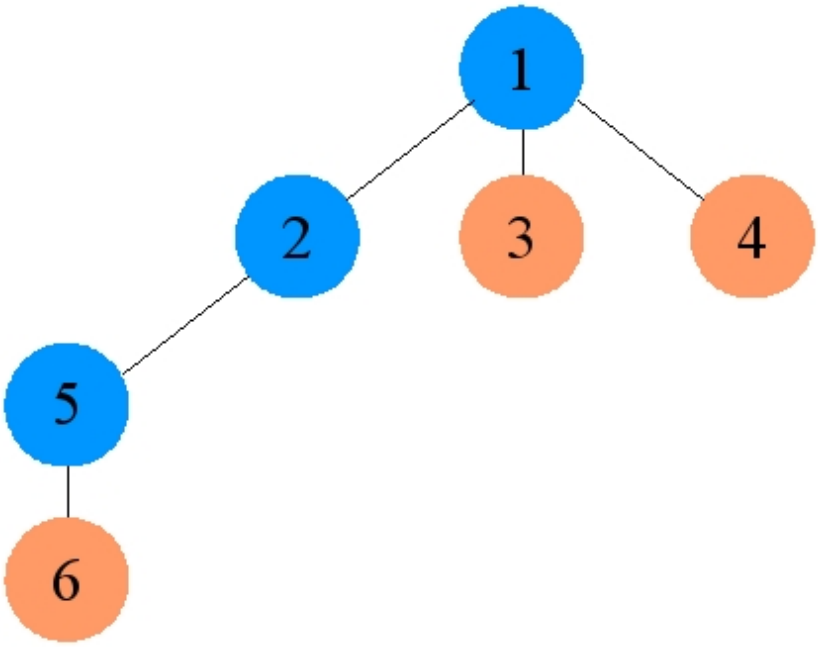

Abbildung 18: Breitensuche 2

Unter der Annahme, dass mehrere Hierachieebenen verwendet werden, habe ich mich für Tiefensuche entschieden. Außerdem besitzt Tiefensuche lineare Komplexität, bei Breitensuche müssen die zuvor betrachteten Knoten im Speicher gehalten werden.

3.8.2 Suchen und ersetzen von Zeichenketten

In der Vorlesung *Grundlagen der Informatik 2* von Herrn Professor Dr. Ertel wurde ich erstmals mit regulären Ausdrücken konfrontiert. Sie sind ein mächtiges Werkzeug, um mit wenigen Codezeilen umfangreiche Manipulationen an Zeichenketten durchzuführen. In *Java* können reguläre Ausdrücke verwendet werden (Paket *java.util.regex*).

Während meines Praktikums hatte ich die Aufgabe, ein Tool zu schreiben, welches in Textdokumenten nach bestimmten Zeichenketten sucht und diese ersetzt. Ich benützte dazu reguläre Ausdrücke. Mir wurde jedoch anschließend gesagt, ich solle aus Gründen der besseren Lesbarkeit künftig die entsprechenden API-Funktionen verwenden.

Aus diesem Grund habe ich für das Parsen der *Mathematica*-Documente die API-Funktionen eingesetzt. Da die Manipulationen einfacher Art waren – in erster Linie aus dem Entfernen zusammenhängender Zeichenketten, beziehungsweise deren Verkettung bestanden – war dies nicht aufwändig. Das Vorgehen wurde bereits ausführlich in Kapitel 3.6.2 beschrieben. Ein Blick in den Quellcode dieser Funktionen zeigt, dass sie ihrerseits mit regulären Ausdrücken arbeiten.

3.9 GUI-Design

Sun hat in den *Java Look and Feel Design Guidelines* [SM01] Regeln – bezüglich des Erscheinungsbildes einer *Java*-GUI-Anwendung – aufgestellt. Diese beinhalten unter anderem:

- Größenvorgaben für bestimmte GUI-Elemente (beispielweise für Buttons einer Button-Bar)
- Abstände zwischen GUI-Elementen
- Gestaltung von Icons
- Design von Panels
- Erscheinungsbild eines Menüs

Diese Regeln sind zwar nicht verpflichtend, deren Beachtung hat allerdings einige Vorteile:

- Bei der Entwicklung im Team ist das Design einheitlich, unabhängig vom jeweiligen Entwickler.
- Der Anwender hat bereits beim ersten Systemstart das Gefühl, das Programm bereits zu kennen, da ihm das Design vertraut erscheint. Er kann so das Programm intuitiv kennenlernen.
- Die Regeln sind bewährt und basieren – bezüglich Ergonomie – auf wissenschaftlichen Erkenntnissen.

In meinem Praktikumsbetrieb dienten die *Look and Feel Guidelines* ebenfalls als Grundlage für die GUI-Entwicklung.

Bei Verwendung des *Form-Layouts* (siehe 2.7) wird einem Entwickler sehr viel Arbeit abgenommen. Im Form-Layout sind einige Regeln bereits integriert. So entspricht beispielsweise die Breite des Fensterrahmens (*Dialog-Border*) oder die Größe der Buttons den Vorgaben der Look and Feel Guidelines.

Das Design der einzelnen Panels muss jedoch nach wie vor selbst entworfen werden.

Es gibt auch Empfehlungen bezüglich der Benutzereingaben. Die beste Methode zur Vermeidung von Fehleingaben besteht darin, diese erst gar

nicht zu ermöglichen. In der Aufgabenverwaltung gibt es deshalb nur dann Felder zur manuellen Texteingabe, wenn dies unbedingt erforderlich ist. Ansonsten lässt sich das Programm vollständig mit der Maus bedienen. Auch werden dem Benutzer nur jene Aktionen ermöglicht, die aktuell sinnvoll sind. Sind beispielsweise nicht alle Pflichtfelder ausgefüllt, ist der *Speichern*-Button inaktiv.

In den Kapiteln 3.5.4, 3.5.5, 3.5.6 und 3.5.7 sind die einzelnen Oberflächen der Komponenten dargestellt. Ich habe versucht, diese so übersichtlich wie möglich zu gestalten. Jedes Fenster sollte über einen klar definierten Funktionsumfang verfügen und ebenso klar definierte Aktionen ermöglichen, die der Anwender auch ohne Lesen einer Dokumentation sofort erfassen kann. Aus diesem Grund sind auch alle GUI-Elemente mit Tooltips unterlegt, anhand derer sich der Anwender über die Funktion informieren kann. Sollten die von mir vorgegebenen Tooltips nicht aussagekräftig sein, können sie jederzeit geändert werden (siehe Kapitel 3.7).

Damit bei den einzelnen Baumknoten sofort klar ist, ob es sich um eine Aufgabe, ein Aufgabenblatt, ein Fachgebiet, einen Studiengang oder eine Vorlesung handelt, habe ich einen eigenen *Tree-Cell-Renderer* geschrieben, der das Austauschen der Standard-Icons ermöglicht. Die Icons der Buttons sollten einerseits optisch ansprechend sein, andererseits dem Anwender auf den ersten Blick Aufschluss über ihre Funktion geben. Die meisten der verwendeten Icons sind von *Eclipse* entnommen, einzelne habe ich auch selbst erstellt.

Bei den Panels habe ich Wert auf Symmetrie und korrekte horizontale wie vertikale Ausrichtung der enthaltenen Elemente gelegt.

GUI-Anwendungen sind sicherlich Geschmackssache. Es gibt auch Anwender, die GUI-Programme generell ablehnen und lieber mit textbasierten Programmen arbeiten. Sie argumentieren, dass man mit Konsolenprogrammen schneller zum Ziel käme. Ich persönlich glaube, dass beide Arten ihre Berechtigung haben. Bei einem Programm wie *GNU-make* habe ich noch nie eine GUI vermisst und würde sie wohl auch nicht benutzen. Eine solche Aufgabenverwaltung kann ich mir allerdings ohne GUI kaum vorstellen. Ob die von mir entworfene GUI gefällt und ob ich die beschriebenen Ziele – bezüglich der Bedienergonomie – erreicht habe, kann letztendlich nur der Anwender entscheiden.

3.10 Sicherheit

Das Thema Sicherheit stand bei dieser Diplomarbeit nicht im Vordergrund. Vorgabe von Herrn Professor Dr. Hulin war, eine einfache Benutzerauthentifizierung zu implementieren. Die Anmeldung läuft nach folgender Prozedur ab: Wenn der Benutzer das Programm startet, wird ein Fenster angezeigt, in das er Benutzername und Passwort eingeben muss. Hat er durch Drücken von *OK* seine Eingaben bestätigt, wird von der Methode *hashPassword()*, der Klasse *FormatTool*, eine 128-Bit *MD5-Checksum* erzeugt. Dies ist nötig, weil das Passwort nicht im Klartext in der Datenbank gespeichert ist, um wenigstens rudimentäre Sicherheit zu gewährleisten.

Wenn dem Benutzer das Eingeben von Benutzername und Passwort zu umständlich erscheint, kann er den Passwortdialog jedoch auch deaktivieren. Die Verbindungsdaten werden dann (wie in Kapitel 3.7.2 beschrieben) aus der Konfigurationsdatei ausgelesen.

Anschließend wird versucht, eine Datenbankverbindung aufzubauen. sind die Verbindungsdaten ungültig (wenn also keine Verbindung zur Datenbank aufgebaut werden kann) wird eine Fehlermeldung ausgegeben. Sind die Daten korrekt, startet das Programm.

Da die Aufgabenverwaltung momentan kein Administrationstool besitzt, um beispielsweise einen neuen Benutzer anzulegen, müsste dieser zuerst eine MD5-Checksum von seinem Passwort erstellen, die dann in der Datenbank abgelegt werden kann. Um diesen Vorgang etwas komfortabler zu gestalten, habe ich ein kleines Werkzeug integriert, welches genau dies erledigt. Es ist zu Finden im Menü unter *Werkzeuge-Passwort hashen*

Anmerkung: Da ich die Vorlesung *Datensicherheit* bei Herrn Professor Löhmann besucht habe ist mir bewußt, dass diese Vorgehensweise nicht wirklich sicher ist. So wäre jederzeit eine *Brute-Force*-Attacke möglich. Ich glaube aber, dass sie den Ansprüchen einer einfachen Benutzerauthentifizierung genügt. Auch die bekannte Forensoftware *phpBB* verwendet dieses Prinzip ([Phi]).

4 Erweiterungen und Verbesserungen

4.1 Bekannte Bugs und Unzulänglichkeiten

Das Programm wurde von mir nach der Fertigstellung ausgiebig getestet. Bezüglich der Funktionalität der einzelnen Komponenten kann ich zum jetzigen Zeitpunkt keine Fehler mehr feststellen. Ebenso erfüllt das Programm das Anforderungsprofil. Allerdings konnte ich, aus den in Kaptitel 2.3 und 3.9 beschriebenen Gründen, den vollen Funktionsumfang lediglich auf einem *Linux*-Rechner testen. Bei der Integration in *Windows* traten ein paar Probleme auf, die in Kapitel 4.1.1 noch näher beschrieben sind.

Es gibt weiterhin Features, die zwar nicht gefordert waren, die ich jedoch gerne integriert hätte. Leider konnte ich dies aus zeitlichen Gründen nicht mehr realisieren:

- Das Programm sollte über ein Adminstrationstool verfügen (siehe 4.2.1).
- Die Fenstergröße der einzelnen Komponenten kann derzeit nicht verändert werden. Ich hätte dies zwar durch Einfügen einiger weniger Codezeilen ermöglichen können, jedoch hätte das kein optisch ansprechendes Ergebnis geliefert. Um den Anforderungen der *Java Look and Feel Design Guidelines* (siehe 3.9) zu genügen, müsste sichergestellt sein, dass die einzelnen GUI-Elemente bei Größenänderung unterschiedlich skalieren (auch Abstände zwischen GUI-Elementen). Die verschiedenen Gewichtungen sind von der Gestalt des jeweiligen Panels abhängig. Wie ich in meinem Praktikum festgestellt habe, ist eine korrekte Skalierung sehr zeitaufwändig zu realisieren, da die Anwendung häufig neu gestartet werden muss, um das Ergebnis zu überprüfen. Der Layout-Manager bietet hierfür keine Unterstützung. Ich bin bei der Entwicklung der GUI von einer Bildschirmauflösung von 1024x768 und einer Systemschriftgröße von 12 pt ausgegangen. Bei extrem großer Auflösung und kleiner Systemschrift, werden die Fenster eventuell sehr klein dargestellt.

Ein weiteres negatives Detail ist die Tatsache, dass der Initialisierungsprozess von *Hibernate* einige Zeit benötigt. Der Grund hierfür ist die Arbeitsweise von *Hibernate*. Wie in Kapitel 2.2.5 beschrieben, erzeugt *Hibernate* anhand der Mapping-Dateien Proxy-Objekte. Dies ist ein einmaliger Vorgang, der beim erstmaligen Öffnen einer Session erfolgt. Da die Instantiierung dieser Objekte mittels *Reflection* realisiert wird, ist sie recht zeitintensiv. Dies ist unter anderem dadurch bedingt, dass die Mapping-Dateien zuerst vom

XML-Parser auf Korrektheit geprüft werden (siehe 4.1.1).

Wenn eine Aufgabe oder ein Aufgabenblatt mit *Mathematica* geöffnet wird, gibt *Mathematica* eine Meldung aus, verschiedene Angaben seien in der Datei nicht vorhanden. Dies liegt vermutlich daran, dass die Meta-Attribute entfernt wurden. Auf die Funktionalität hat die Meldung jedoch keinen Einfluss – sie ist auch als *Message* und nicht als *Error* deklariert.

4.1.1 Probleme bei Integration in MS-Windows

Java ist plattformunabhängig. Umso erstaunter war ich, als ich die Aufgabenverwaltung erstmals unter *Windows* startete. Der *XML*-Parser lieferte die Fehlermeldung, dass er die *Hibernate*-Dateien nicht validieren könne, weil die Pfadangabe ungültig sei.

Ein *XML*-Dokument wird mit Hilfe der *Document Type Definition (DTD)* auf Korrektheit geprüft. In jedem *XML*-File ist der Pfad zur *DTD* im Header angegeben. Dieser Pfad kann sowohl auf eine Datei des lokalen Dateisystems, als auch auf eine Webressource verweisen. Ersteres ist selbstverständlich performanter, da kein Internet-Zugriff erfolgt.

Ich hatte deshalb bei der *Hibernate*-Konfigurationsdatei sowie den Mapping-Dateien, den Pfad zur *DTD* auf das Stammverzeichnis der Aufgabenverwaltung gesetzt und die *DTD*-Dateien dort abgelegt. Die Pfadangabe erfolgte relativ, wie in [Har02] beschrieben. Unter *Linux* funktionierte dies auch anstandslos, bei der Integration in *Windows* trat jedoch das oben beschriebene Problem auf. Mit geändertem Pfad (Verweis auf die Webressource) startete das Programm auch unter *Windows*, jedoch wurde durch den Webzugriff der Initialisierungsprozess von *Hibernate* zusätzlich verlangsamt (siehe 4.1).

Da ich trotz intensiver Recherche im Internet keine Information zu dieser Problemstellung gefunden habe, vermute ich, dass es sich nicht um ein generelles *Windows*-Problem handelt. Sicherheitshalber sind jedoch auf der beiliegenden CD (siehe Kapitel 5.2) zwei Versionen des Programms vorhanden. Diese unterscheiden sich bezüglich der Deklaration der *DTD*'s. Der Installationsvorgang ist in Kapitel 5.4 ausführlich beschrieben.

Es traten unter *Windows* außerdem Schwierigkeiten beim Laden der GUI auf. Der Aufbau der einzelnen Frames erfolgte deutlich träger als unter *Linux*. Teilweise wurde nach dem Start des Programms nur ein weißes

Fenster dargestellt. Nach Minimieren und anschließendem Wiederherstellen des Fensters war die Darstellung korrekt. Auch eine Änderung des *Look-And-Feels* brachte keine Verbesserung. Dieses Verhalten ist umso ungewöhnlicher, da der *Windows*-Rechner mit schnellerer Hardware als mein Notebook ausgerüstet ist.

4.2 Erweiterungsvorschläge

Wie in Kapitel 3.4 erwähnt, ist das Programm so konzipiert, dass es möglichst einfach erweitert werden kann. Dies betrifft sowohl die Integration einer neuen Komponente, als auch Änderungen an bestehenden Komponenten.

4.2.1 Administrationstool

Derzeit verfügt das Programm noch nicht über ein Werkzeug zur Konfiguration. Das heißt, das Anlegen neuer Vorlesungen und Studiengänge sowie das Hinzufügen von Benutzern muss direkt auf der Datenbank erfolgen. Diese Prozedur ist fehleranfällig. Wenn etwa das Anlegen einer Relation vergessen wird, ist die Funktion des Programms nicht mehr gewährleistet.

Ein Administrationstool würde dies erleichtern. Das Tool könnte sowohl als eigenständiges Programm oder auch als zusätzliche Komponente implementiert werden.

Sollte es als neue Komponente entwickelt werden, schlage ich folgende Vorgehensweise vor:

- Das Menü *Tools* erhält den zusätzlichen Menüeintrag, *Administration*.
- Wird dieser Eintrag ausgewählt, öffnet sich ein Fenster. Hier muss das Administratorpasswort eingegeben werden.
- Es wird eine zweite Datenbankverbindung, in einem separaten Thread, geöffnet. Hierzu müsste in *hibernate.cfg.xml* die Anzahl der *JDBC*-Verbindungen erhöht werden.
- Nach erfolgreicher Anmeldung können die Änderungen in einer grafischen Oberfläche durchgeführt werden. Die Programmlogik überwacht, dass alle Relationen korrekt und alle notwendigen Spalten ausgefüllt sind.

4.2.2 Automatisches Einfügen von Querverweisen

Von Herrn Professor Dr. Hulin kam der Vorschlag, anhand von bestimmten *Tags*, Querverweise zu einer anderen Aufgabe zu spezifizieren.

Der Dozent gibt beispielsweise in seiner Aufgabe ein:

> *„Vergleichen sie diese Aufgabe mit #107“*

#107 soll dann ersetzt werden durch:

> *„Lineare Algebra - SS 2005 - Blatt 5 - Aufgabe 4“*

Ich habe mir zwei Vorgehensweisen überlegt, die sich darin unterscheiden, wie der Dozent nach der Aufgabe, auf die verwiesen wird, suchen möchte.

- Er sucht nach einer Aufgabe, lässt sich die Aufgabenblätter anzeigen, die diese Aufgabe enthalten und wählt ein Blatt aus.
- Er sucht nach einem Aufgabenblatt und wählt eine der Aufgaben aus.

Für die Realisierung der ersten Variante hätte ich folgenden Vorschlag:

- Die Komponente *Aufgabe erstellen* wird um die Funktion, *Aufgabe suchen*, erweitert.
- Das Fenster, welches sich beim Drücken des Buttons *Informationen zur Aufgabe* öffnet, wird verändert. Bislang werden nur die Titel der Aufgabenblätter angezeigt. Damit der Dozent weiß, dass er das richtige Blatt gewählt hat, wird die Möglichkeit geschaffen, diese mit *Mathematica* zu öffnen. Diese Änderung wird am besten in *AbstractExcerciseComponent* (sowie *AbstractExcerciseFrame*) durchgeführt, da sie auch bei *Aufgabe bearbeiten* notwendig ist.
- Es wird ein zusätzliches Textfeld eingefügt, das bei Auswahl eines Aufgabenblattes eine Zeichenkette (beispielsweise in der Form, *#<ID der Aufgabe>_<ID des Aufgabenblattes>*) darstellt.
- Der Dozent fügt diese Zeichenkette mittels *copy-paste* in seinen Aufgabentext ein.
- Es wird eine neue Methode implementiert, die aus Zeichenketten dieser Form die ID der Aufgabe und die ID des Aufgabenblattes regenerieren kann.

- Wenn ein Aufgabenblatt (oder eine Aufgabe) angezeigt werden soll, wird der Text zuvor nach Zeichenketten dieser Form durchsucht.
- Falls enthalten, wird die Zeichenkette der oben genannten Methode übergeben. Da die zurückgegebenen ID's eindeutig sind (Primary-Keys), müssen nur noch die benötigten Felder aus der Datenbank geholt und eine Zeichenkettenersetzung durchgeführt werden.

Bei der zweiten Variante müssten die Komponenten *Aufgabe erstellen* und *Aufgabe bearbeiten* um die Suchfunktion nach Aufgabenblättern erweitert werden. Um Mehrfachimplementierung zu vermeiden wäre es elegant, diese Funktion in die Superklasse *AbstractComponent* zu verlagern. Der Dozent kann nun nach einem Blatt suchen und sich nach erfolgreicher Suche die einzelnen Aufgaben anzeigen lassen. Wie in der ersten Variante wird ein neues Textfeld integriert, das bei Selektion einer Aufgabe die bereits beschriebene Zeichenkette anzeigt. Das weitere Vorgehen ist identisch zu Variante eins. Bezüglich der Bedienoberfläche würde sich ein ähnliches Design wie bei Komponente *Aufgabenblatt bearbeiten* – also Verwenden von Karteireitern – empfehlen. Außerdem sind diese Panels bereits vorhanden, müssten also nicht neu entwickelt werden.

5 Programminstallation

5.1 Systemvoraussetzungen

Folgende Anforderungen sind für den fehlerfreien Betrieb des Programms notwendig:

- *MS Windows 2000/XP* oder *Linux* mit Kernelversion >= 2.4 und X-Server mit KDE oder Gnome
- *Java Runtime Environment*, Version >= 5.0
- Zugriff auf ein unterstütztes Datenbank-Management-System (Siehe Tabelle: 1)
- Bildschirmauflösung >= 1024x768

5.2 Inhalt der CD

Auf der CD sind enthalten:

- Dieses Dokument als *pdf*-Datei (*diplomarbeit.pdf*).
- Die Entwicklerversion des Programms, als *Eclipse*-Projekt inklusive aller benötigten Bibliotheken für den Programmlauf und die verwendeten Entwicklungstools, sowie das Datenbankschema für *MySQL* (Verzeichnis *diplomarbeit_entwickler*). Die Entwicklerversion ist für Personen gedacht, die dieses Programm weiterentwickeln, verändern oder mögliche Fehler beseitigen möchten.
- Die Anwenderversion des Programms, inklusive aller benötigten Bibliotheken für den Programmlauf sowie das Datenbankschema für *MySQL* (Verzeichnis *diplomarbeit_anwender*). Die Anwenderversion enthält keine Quelldateien. Diese befinden sich nur in der Entwicklerversion. Die Anwenderversion ist für den Betrieb des Programms ausreichend.
- Eine zweite Anwenderversion, bei der im Gegensatz zur ersten nicht auf die lokalen *DTD*-Dateien verwiesen wird (siehe 4.1). Sie sollte nur dann verwendet werden, wenn bei der anderen Version Fehlermeldungen auftreten (Verzeichnis *diplomarbeit_anwender_extern_dtd*).
- *Eclipse* für *Linux* und *Windows* (Verzeichnis *eclipse_win*, beziehungsweise *eclipse_lin*).

5.3 Importieren des Datenbankschemas

Das Datenbankschema ist in zwei Ausführungen vorhanden:

- Die reine Tabellenstruktur, ohne Datensätze (*datenbankschema_leer.sql*).
- Die Tabellenstruktur inklusive einzelner Testdatensätze (*datenbankschema_inserts.sql*).

Folgende Testdatensätze werden bei Verwendung von *datenbankschema_inserts.sql* angelegt:

- Ein Datensatz für den Studiengang *AI*.
- Sechs Datensätze für die Vorlesungen: *Mathematik 1, Mathematik 2, Datenbanksysteme, Mathematik für Informatiker, Datensicherheit und Künstliche Intelligenz.*
- Zwei Datensätze für die beiden Prüfer dieser Diplomarbeit, Herrn Professor Dr. Hulin und Herrn Professor Dr. Ertel.

Die Datenbankschemen befinden sich auf der CD sowohl im Verzeichnis *diplomarbeit_anwenderversion*, als auch in *diplomarbeit_entwicklerversion*. Wenn ein anderer Datenbankserver als *MySQL* verwendet wird, müssen sie entsprechend angepasst werden. Das Schema kann allerdings auch datenbankunabhängig von *Hibernate* eingespielt werden. Dies funktioniert allerdings nur mit der Entwicklerversion, da hierfür *Eclipse* notwendig ist (6.1).

In einem weiteren *SQL*-Script, der Datei *benutzer.sql*, wird die eigentliche Datenbank erstellt (falls sie noch nicht existiert) und drei Datenbankbenutzer angelegt:

- Ein Benutzer für Herrn Professor Dr. Hulin:
  Benutzername: *hulin*
  Passwort: *hulin.*
- Ein Benutzer für Herrn Professor Dr. Ertel:
  Benutzername: *ertel*
  Passwort: *ertel.*
- Ein Benutzer *math* (Passwort: *math*), der erweiterte Rechte erhält. Dieser wird benötigt, wenn der *Ant*-Task *schema-export* ausgeführt wird (siehe Kapitel 6.1).

Der schnellste Weg, die Datenbank zu Testzwecken für die Aufgabenverwaltung einzurichten besteht also darin, zuerst das Skript *benutzer.sql* und anschließend *datenbankschema_inserts.sql* auszuführen.

Wie in Kapitel 3.10 beschrieben, werden die Passwörter nicht im Klartext sondern als *MD5*-Checksum abgelegt. Sollte also der Wunsch bestehen, die Benutzer manuell hinzuzufügen, sind hier zum schnellen Einstieg die Hashwerte der Passwörter der oben genannten Benutzer:

Das Passwort *hulin*:	3890f5cfc4e00c882b69c43c88cd3422
Das Passwort *ertel*:	3890f5cfc4e00c882b69c43c88cd3422
Das Passwort *math*:	7e676e9e663beb40fd133f5ee24487c2

Die Passwörter können später jederzeit mit Hilfe des *MD5*-Tools (siehe 3.10) nach Belieben neu erstellt werden.

5.4 Installation der Anwender-Version

Obwohl die Installation des Programms sehr simpel ist, sind ein paar grundlegende Schritte notwendig. Entsprechend der Anleitung im letzten Kapitel sollte ein (unterstützter) Datenbankserver, ein Passwort und ein Benutzername vorhanden sein. Das Datenbankschema muss importiert sein.

Im Auslieferungszustand, ist das Programm für den Betrieb mit einem *MySQL*-Datenbankserver vorbereitet. Sollte ein anderer Datenbankserver verwendet werden, muss der entsprechende *JDBC*-Treiber beschafft werden (Siehe Tabelle: 1).

- Das Verzeichnis *diplomarbeit_anwenderversion* muss (mit allen Unterverzeichnissen) auf die Festplatte kopiert werden.
- Das Zielverzeichnis ist beliebig. Es ist jedoch eine Schreibberechtigung für dieses Verzeichnis nötig.
- Die Datei *diplomarbeit_anwenderversion/user.properties* muss entsprechend der Ausführungen von Kapitel 3.7.2 angepasst werden.
- Wenn es sich beim Datenbankserver nicht um *MySQL* handelt, muss der *JDBC-Treiber* des verwendeten Datenbankservers nach *diplomarbeit_anwenderversion/lib* kopiert werden.
- In einer Shell wird ins Verzeichnis *diplomarbeit_anwenderversion* gewechselt.

- Durch folgendes Kommando wird das Programm gestartet:
  `java -jar aufgabenverwaltung.jar`
- Das Programm startet. Alternativ zur manuellen Eingabe des Kommandos, kann dieses auch in eine Batch-Datei (*Windows*) oder ein Shell-Script (Linux) geschrieben werden.
- Sollte unter *Windows* eine Fehlermeldung des *XML*-Parsers erscheinen, handelt es sich vermutlich um das gleiche Problem, dass ich bei der Integration in *Windows* feststellte. Es muß dann die Version *diplomarbeit_anwender_extern_dtd* verwendet werden.
- Das Verzeichnis *diplomarbeit_anwenderversion* kann beliebig umbenannt werden.

6 Informationen für Entwickler

6.1 Integration in Eclipse

Um die Entwicklerversion als Projekt in *Eclipse* zu importieren, ist zuerst sicherzustellen, dass es sich bei der verwendeten Version von *Eclipse* um die Versionsnummer *3.1* (welche derzeit aktuell ist) handelt. Diese Version ist auch Bestandteil der beiliegenden CD (siehe 5.2). Ob ein Import bei zukünftigen Versionen möglich ist, kann ich zum momentanen Zeitpunkt selbstverständlich nicht beurteilen.

Der Ordner *diplomarbeit_entwicklerversion* enthält alle benötigten Dateien und Verzeichnisse. Das Importieren des Projekts geschieht mit dem Menüpunkt *File-Import-Filesystem*. Sollte das Projekt im *Package Explorer* erscheinen, war der Import erfolgreich.

Es sind nun noch ein paar Schritte zu erledigen, um das Programm lauffähig zu machen:

- Da *Eclipse* in seiner *Classpath*-Datei die Klassenpfade absolut einträgt, findet er die verwendeten Bibliotheken noch nicht. Die Pfade können entweder durch Editieren der (versteckten) Datei *diplomarbeit_entwicklerversion/.classpath* manuell oder durch auswählen des Menüpunktes *Project-Properties-Java Build Path* menügesteuert geändert werden. Die benötigten Libraries liegen im Verzeichnis *diplomarbeit_entwicklerversion/lib*. Die Bibliotheken in den beiden Unterverzeichnissen *Hibernate* und *XDoclet* brauchen nicht eingebunden zu werden. Dies werden lediglich von *Ant* verwendet.
- Die Datei *user.properties* muss angepasst werden (siehe Kapitel 3.7.2).
- In der Datei *hibernate.properties* müssen die Parameter der Datenbankverbindung geändert werden. Diese Datei ist nicht redundant, da sie von *hbm2ddl* (siehe 2.9.3) verwendet wird. Der dort spezifizierte Benutzer benötigt Rechte zum Erstellen und Löschen von Tabellen.

Wenn entsprechend der Beschreibung in Kapitel 5.3 Benutzer eingerichtet wurden, kann jetzt das Datenbankschema eingespielt werden. Dies geschieht durch Ausführen des *Ant*-Targets *schema-export*. Jetzt startet das Programm.

Wenn Veränderungen an den *XDoclet*-Annotations in den Persistenklassen vorgenommen werden, sollte im direkten Anschluss das *Ant*-Target

generate_hibernate gestartet werden, da ansonsten die Änderungen nicht auf die Mapping-Dateien übertragen werden. Wenn in *hibernate.properties* die Property *MATH_MANAGER_DB_FLUSH_ON_START*, den Wert *false* hat, was dem Auslieferungszustand entspricht, muss anschließend auch noch das Target *schema-export* aufgerufen werden, um das Datenbankschema zu aktualisieren.

Wenn *MATH_MANAGER_DB_FLUSH_ON_START* auf *true* gesetzt wird, wird bei jedem Start des Programms das Datenbankschema neu eingespielt.

Wenn das Programm verändert wurde, kann mit Hilfe des Targets *create_bin* die Anwenderversion erstellt werden. Diese wird im Unterverzeichnis *deploy* angelegt.

7 Schlussbemerkungen

Obwohl ich ein paar Details zeitlichen Gründen nicht mehr realisieren konnte (siehe 4.1), bin ich mit dem erreichten Ergebnis zufrieden, zumal der Entwicklungsprozess nicht reibungslos verlief. Insbesondere der Umgang mit *Hibernate* bereitete mir über einen längeren Zeitraum Probleme, die ich jedoch letztendlich lösen konnte.

Die Integration von *Mathematica* konnte leider nicht wie ursprünglich geplant umgesetzt werden. Dies ist einerseits schade, andererseits glaube ich, dass die daraufhin gewählte Vorgehensweise akzeptabel ist.

Sehr zufrieden bin ich mit der Tatsache, dass sich die von mir vorab entworfene Architektur, im Entwicklungsprozess bewährt hat. Es lohnt sich also durchaus, zu Anfang ausreichend Zeit für die Planung eines Projekts aufzuwänden. Die investierte Zeit amortisiert sich auf längere Sicht.

Ebenso habe ich während der Diplomarbeit einige Kentnisse erworben, die auch in meiner künftigen beruflichen Tätigkeit gefordert sind. Dies betrifft sowohl den Einsatz bestimmter Technologien, als auch die Fähigkeit, auftretende Probleme effektiv zu lösen.

Abschließend möchte ich noch meinem Betreuer, Herrn Professor Dr. Hulin, für die Unterstützung während der letzten Monate danken.

Ant Build-Tool von der Apache Foundation

BC4J Objektrelationales Datenbankmapping von Oracle

CVS *Concurrent Versions System* – Versionsverwaltungssoftware

DAO *Data Access Objects* – Design-Pattern für objektorientierten Datenbankzugriff

DBMS Datenbank Management System

DTD *Document Type Definition* – Dateiformat, das die Validierung von *XML*-Dateien ermöglicht

Eclipse Für die Diplomarbeit verwendete Entwicklungsumgebung

hbm2dll *Hibernate*-Tool, das anhand der Deskriptor-Dateien ein Datenbankschema generiert

Hibernate Framework für objektrelationales Datenbankmapping

JGoodies Framework von Carsten Lentzsch, das die Entwicklung GUI-basierter Anwendungen unter *Java* erleichtert

JIT *JustInTimeCompiler* – Kompiliert zur Laufzeit *Java-Byte-Code* in Maschinencode

JRE *Java Runtime Environment*

JUnit *Java*-Framework, das Softwaretests auf Modulebene erleichtert

MVC Design-Pattern, dessen Ziel die Trennung von Programmlogik und Programmdarstellung in GUI-basierten Anwendungen ist

ORM Objekt-relationales Datenbankmapping

POJO *Plain Ordinary Java Objects* – *Java*-Klassen mit *get()* und *set()* Methoden für jede Property

Singleton Design-Pattern für die Erstellung von Factory-Klassen

XDoclet Tool, das die Erstellung von *XML*-Deskriptor-Dateien automatisiert

Abbildungsverzeichnis

1 Use-Cases für die Aufgabenverwaltung 4
2 Objekzustände bei Hibernate 16
3 Kommunikation mit dem Kernel 24
4 Schichten der Aufgabenverwaltung 35
5 Entity-Relashionship-Diagram 39
6 Tabellenstruktur 40
7 Das MVC-Pattern 42
8 Aufbau einer Komponente 47
9 Vererbungshierarchie des Models 48
10 Model mit Mehrfachvererbung 49
11 Vererbungshierarchie der View 51
12 Vererbungshierarchie des Controllers 52
13 Screenshot der GUI von *Aufgabe erstellen* 53
14 Screenshot der GUI von *Aufgabe bearbeiten* 56
15 Screenshot der GUI von *Aufgabenblatt erstellen* 59
16 Screenshot der GUI von *Aufgabenblatt bearbeiten* 61
17 Breitensuche 1 75
18 Breitensuche 2 75

Listings

1 Einfaches Mapping 11
2 Einfache Persistenzklasse 12
3 Transaktion 18
4 equals() an Primärschlüssel gebunden 20
5 hashCode() an Business-Key gebunden 21
6 Singleton-Implementierung 43
7 Singleton-Implementierung 2 63
8 Einfache Notebook-Datei 64
9 Notebook-Datei mit Berechnungen 65

Tabellenverzeichnis

1 Unterstützte Datenbanken 22

Literatur

[AFa] APACHE-FOUNDATION: *Ant Documentation*. `http://ant.apache.org/manual/index.html`. – Online–Ressource, Abruf: 14. Jan. 2006

[AFb] APACHE-FOUNDATION: *Apache-Foundation Homepage*. `http://www.apache.org`. – Online–Ressource, Abruf: 20. Jan. 2006

[AFc] APACHE-FOUNDATION: *Apache-Tomcat*. `http://tomcat.apache.org/`. – Online–Ressource, Abruf: 19. Jan. 2006

[Bal99] BALZERT, Heide: *Lehrbuch der Objektmodellierung*. Spektrum, 1999. – ISBN 3–8274–0285–9

[BK05] BAUER, Christian ; KING, Gavin: *Hibernate in Action*. Manning, 2005. – ISBN 1–932394–15–X

[Blo01] BLOCH, Josuah: *Effective Java*. Sun Microsystems, 2001. – ISBN 0–201–31005–8

[Cha02] CHAPELL, David: *.NET verstehen*. Addison-Wesley, 2002. – ISBN 3–8273–2018–6

[de.] DE.WIKIPEDIA.ORG: *Mathematica*. `http://de.wikipedia.org/wiki/Mathematica`. – Online–Ressource, Abruf: 18. Jan. 2006. – Letzte Änderung: 13.Dez.2005

[DP] DOKBOOK-PROJECT: *DocBook*. `http://docbook.sourceforge.net/`. – Online–Ressource, Abruf: 19. Jan. 2006

[Dum] DUMKE, R.: *Zur Geschichte von Java*. `http://ivs.cs.uni-magdeburg.de/~dumke/EAD/Skript02.html`. – Online–Ressource, Abruf: 16. Jan. 2006

[EFa] ECLIPSE-FOUNDATION: *Eclipse Public License*. `http://www.eclipse.org/legal/epl-v10.html`. – Online–Ressource, Abruf: 17. Jan. 2006

[EFb] ECLIPSE-FOUNDATION: *Strategic Members*. `http://www.eclipse.org/membership/members/strategic.php`. – Online–Ressource, Abruf: 17. Jan. 2006

[EF05] ECLIPSE-FOUNTADION: *Eclipse Homepage*. Version: 2005. `http://www.eclipse.org`. – Online–Ressource, Abruf: 20. Jan. 2006

[Ell04] Elliott, James: *Hibernate: A Developer's Notebook*. O'Reilly, 2004. – ISBN 0–596–00696–9

[Fed00] Feddema, Helen: *DAO Object Models*. O'Reilly, 2000. – ISBN 1–5659–2435–5

[Fus] Fussel, Mark L.: *Foundations of Object-Relational Mapping*. `http://www.chimu.com/publications/objectRelational/`. – Online–Ressource, Abruf: 17. Jan. 2006

[GHJV98] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Addison-Wesley, 1998. – ISBN 0–201–63361–2

[GNUa] GNU: *Concurrent Version System*. `http://www.nongnu.org/cvs/`. – Online–Ressource, Abruf: 20. Jan. 2006. – Letzte Änderung: 14.Nov.2005

[gnub] gnu.org: *GNU Lesser General Public License*. `http://www.gnu.org/copyleft/lesser.html`. – Online–Ressource, Abruf: 17. Jan. 2006. – Letzte Änderung: 05.Mai.2005

[Gru02] Gruhn, Volker: *EJB 2.0 Anwendungen*. Addison-Wesley, 2002. – ISBN 3–8273–2037–2

[GT00] Gruhn, Volker ; Thiel, Andreas: *Komponentenmodelle*. Addison-Wesley, 2000. – ISBN 3–8273–1724–X

[Har02] Harold, Elliotte R.: *Processing XML with Java*. Addison-Wesley, 2002. – ISBN 0–2017–7186–1

[HR01] Härder, Theo ; Rahm, Erhard: *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer, 2001. – ISBN 3–540–42133–5

[HTa] Hibernate-Team: *Hibernate Reference Documentation*. `http://www.hibernate.org/hib_docs/reference/en/html_single/`. – Online–Ressource, Abruf: 17. Jan. 2006

[HTb] Hibernate-Team: *NHibernate for .NET*. `http://hibernate.org/343.html`. – Online–Ressource, Abruf: 17. Jan. 2006

[Jan03] Jansen, Rudolf: Open Source-Datenbanken: Die aktuellen Versionen von MySQL und PostgreSQL. In: *Java-Magazin* (2003), Dezember, Nr. 12

[Knu91] KNUTH, D. E.: *Computers & Typesetting, Vol. A: The TEX Book*. Addison-Wesley, 1991. – ISBN 0–201–13438–1

[Kum03] KUMAR, Pankaj: *J2EE Security*. Hewlett Packard, 2003. – ISBN 0–1314–0264–1

[Lena] LENTZSCH, Karsten: *About JGoodies*. `http://www.jgoodies.com/about.html`. – Online–Ressource, Abruf: 20. Jan. 2006

[Lenb] LENTZSCH, Karsten: *JGoodies Form-Layout Whitepaper*. `https://forms.dev.java.net/files/documents/32/505/whitepaper.pdf`. – Online–Ressource, Abruf: 16. Jan. 2006. – Letzte Änderung: 21.Apr.2004

[MH04] MASSOL, Vincent ; HUSTED, Ted: *JUnit in Action*. Manning, 2004. – ISBN 1–930110–99–5

[OIO] ORIENTATION-IN-OBJECTS: *XDoclet - Model Driven Architecture*. `http://www.oio.de/xdoclet.htm`. – Online–Ressource, Abruf: 20. Jan. 2006

[Phi] PHILIPP, K.: *Verschlüsselung der Passwörter bei phpBB*. `http://www.phpbb.de/doku/kb/artikel.php?artikel=78`. – Online–Ressource, Abruf: 16. Jan. 2006. – Letzte Änderung: 10.Sep.2005

[Pil] PILZ, Markus: *Das Sicherheitskonzept von Java*. `www.ifi.unizh.ch/richter/people/pilz/pubb/Pilz97a.pdf`. – Online–Ressource, Abruf: 17. Jan. 2006

[Plö04] PLÖD, Michael: Objektrelationales Datenbankmapping mit Hibernate. In: *Java-Magazin* (2004), August, Nr. 8

[RBP+90] RUMBAUGH, James ; BLAHA, Michael ; PREMERLANI, William ; EDDY, Frederick ; LORENSEN, William: *Object-Oriented Modeling and Design*. Prentice Hall, 1990. – ISBN 3–486–24319–5

[Ree] REENSKAUG, Trygve: *Models, Views, Controllers*. `http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf`. – Online–Ressource, Abruf: 21. Jan. 2006

[RS05] RAINSBERGER, J. B. ; STIRLING, Scott: *JUnit Recipes*. Manning, 2005. – ISBN 1–932394–23–0

[Sar] SARMA, T. K.: *Applying Observer Pattern in C++ Applications.* http://www.codeproject.com/cpp/observer.asp. – Online–Ressource, Abruf: 21. Jan. 2006. – Letzte Änderung: 09.Jan.2001

[Sch] SCHNELLE, Dirk: *Einführung in log4J.* http://www.javacore.de/jumpToDownload.php?id=8. – Online–Ressource, Abruf: 30. Jan. 2006. – Letzte Änderung: 20.Jan.2006

[Shu05] SHUDO, Kazuyuki: *Performance Comparison of Java/.NET Runtimes.* Version: 2005. http://www.shudo.net/jit/perf/. – Online–Ressource, Abruf: 16. Jan. 2006. – Letzte Änderung: 20.Nov.2005

[SKPH] SCHMIDT, Walter ; KNAPPEN, Jörg ; PARTL, Hubert ; HYNA, Irene: *LaTeX 2 Kurzbeschreibung.* ftp://dante.ctan.org/tex-archive/info/lshort/german/. – Online–Ressource, Abruf: 17. Jan. 2006. – Letzte Änderung: 04.Okt.2003

[SM] SUN-MICROSYSTEMS: *Backing a ResourceBundle with Properties Files.* http://java.sun.com/docs/books/tutorial/i18n/resbundle/propfile.html. – Online–Ressource, Abruf: 24. Jan. 2006

[SM01] SUN-MICROSYSTEMS: *Java Look and Feel Design Guidelines: Advanced Topics.* Addison-Wesley, 2001. – ISBN 0–201–77582–4

[Sun] SUNDSTED, Todd: *MVC meets Swing.* http://www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html. – Online–Ressource, Abruf: 21. Jan. 2006. – Letzte Änderung: Apr.1998

[Ull05] ULLENBOOM, Christian: *Java ist auch eine Insel.* Galileo, 2005. – ISBN 3–89842–747–1

[wika] WIKIPEDIA.DE: *JUnit.* http://de.wikipedia.org/wiki/JUnit. – Online–Ressource, Abruf: 20. Jan. 2006. – Letzte Änderung: 06.Jan.2006

[wikb] WIKIPEDIA.DE: *Motif.* http://de.wikipedia.org/wiki/Motif. – Online–Ressource, Abruf: 23. Jan. 2006. – Letzte Änderung: 23.Jan.2006

[Wol03] WOLFRAM, Stephen: *Mathematica Book.* Wolfram Media, 2003. – ISBN 1–57955–022–3

[WRa] Wolfram-Research: *How JLink works*. http://www.wolfram.com/solutions/mathlink/jlink/qa.html#howitworks. – Online–Ressource, Abruf: 19. Jan. 2006

[WRb] Wolfram-Research: *Mathematica Tutorial*. http://documents.wolfram.com/mathematica/. – Online–Ressource, Abruf: 17. Jan. 2006

[WRc] Wolfram-Research: *MathLink License Agreement*. http://www.wolfram.com/solutions/mathlink/license.html. – Online–Ressource, Abruf: 18. Jan. 2006

[WRd] Wolfram-Research: *Q+A to Webmathematica*. http://www.wolfram.com/products/webmathematica/qa/technology.html. – Online–Ressource, Abruf: 17. Jan. 2006

[WRe] Wolfram-Research: *Webmathematica Amateur License*. http://www.wolfram.com/products/webmathematica/versions/amateurlicense.html. – Online–Ressource, Abruf: 19. Jan. 2006

[WRf] Wolfram-Research: *Webmathematica Tutorial*. http://documents.wolfram.com/webmathematica/. – Online–Ressource, Abruf: 17. Jan. 2006

[XT] XDoclet-Team: *XDoclet Documentation*. http://xdoclet.sourceforge.net/xdoclet/index.html. – Online–Ressource, Abruf: 16. Jan. 2006. – Letzte Änderung: 05.Mai.2005