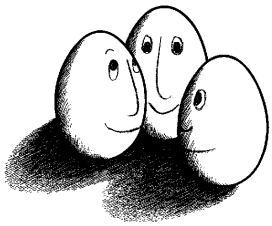
Endbericht

Projektgruppe 461

Kollaboratives Strukturieren von Multimediadaten für Peer-to-Peer Netze



Teilnehmer der Projektgruppe:

Metin Aksoy
Dominique Marc Burgard
Said Chihani
Oliver Flasch
Andreas Kaspari
Matthias Lüttgens
Maxim Martens
Bülent Möller
Umut Öztürk
Philip Thome

Betreuer:

Prof. Dr. Katharina Morik
Dipl.-Inform. Michael Wurst

29. September 2005

Inhaltsverzeichnis

1. Einleitung

Der Endbericht der Projektgruppe (PG) 461 „Kollaboratives Strukturieren von Multimediadaten für Peer-to-Peer Netze “ legt den Verlauf der Projektgruppe vor und stellt die Resultate der beiden Semester dar.

Die PG wurde vom Prof. Dr. Katharina Morik und Dipl.-Inform. Michael Wurst am Lehrstuhl für Künstliche Intelligenz im Fachbereich Informatik der Universität Dortmund betreut.

1.1. PG Aufgabe

Im Rahmen der Projektgruppe soll ein System zur dezentralen Verwaltung von multimedialen Daten auf Grundlage von Peer-to-Peer (P2P) Netzen entwickelt werden.

Die Grundidee besteht darin, dass Nutzer ihre Daten lokal und unabhängig von anderen Nutzern speichern und nach ihrer individuellen Sicht strukturieren und klassifizieren können (Audiodaten z.B. nach Musikgenres). Dabei sollen sie von intelligenten Agenten unterstützt werden, welche wiederum auf die Daten und persönlichen Strukturen anderer Nutzer zugreifen. Auf diese Weise können die Benutzer kollaborativ und verteilt Daten verwalten.

Als Basis soll dabei zum einen das Paradigma der P2P-Netzwerke dienen, welches, berühmt durch Anwendungen wie GNUTELLA [1] oder FREENET [2], mittlerweile auch in viele Groupwareszenarien Verwendung findet. Zum anderen sollen Methoden des Maschinellen Lernens verwendet werden, die es ermöglichen, große Mengen von multimedialen Datenobjekten effizient zu verarbeiten, d.h. zu klassifizieren oder zu gruppieren.

Die Studenten können zur Realisierung des Systems zum einen auf bestehende P2P Lösungen, wie z.B. JXTA[3], einer von Sun entwickelten, sprach- und plattformunabhängigen P2P-Plattform aufbauen, zum anderen auf Methoden und Algorithmen, welche im Rahmen des YALE[4]-Systems am Lehrstuhl für Künstliche Intelligenz entwickelt wurden.

Mindestziel der PG ist es, ein System zu entwickeln, in dem Benutzer Klassifikationsschemata erstellen und austauschen können. Außerdem soll es möglich sein, automatisch Daten in ein Schema einzuordnen.

Viele aktuelle Systeme zur Verwaltung von Informationen basieren auf einer zentralen Architektur. Dies betrifft zum einen das Bereitstellen der Daten auf einem zentralen Server zum anderen das Strukturieren der Daten durch ein gemeinsames globales Klassifikationsschema. Ein Beispiel für ein solches Klassifikationsschema ist z.B. der AMAZON[5] Produktkatalog.

Solche globalen Lösungen sind nicht immer zufriedenstellend. Zum einen haben verschiedene Benutzer oder Benutzergruppen häufig verschiedene Vorstellungen davon, wie Daten richtig strukturiert werden sollen (z.B. wie Musikgenres in Beziehung stehen und welches Musikstück zu welchem Genre gehört). Zum anderen ändern sich sowohl die Daten als auch die Klassifikationsschemata in vielen Szenarien eher lokal als global. Indem beispielsweise einzelne Nutzer lokal neue Datenobjekte hinzufügen, wird auch eine lokale Verfeinerung der Klassifikationsschemata nötig, beispielsweise durch das Hinzufügen von Unterkategorien.

[1] http://rfc-gnutella.sourceforge.net/
[2] http://freenetproject.org/
[3] Juxtapose http://www.jxta.org
[4] http://yale.cs.uni-dortmund.de
[5] http://www.amazon.com

Diese Dynamik ist durch zentralisierte Systeme nur schwierig einzufangen. Beide Probleme legen es nahe, die Datenverwaltung dezentral zu organisieren. Während verschiedene P2P-*Filesharing*- Systeme (wie GNUTELLA oder FREENET) zu einer völlig neuartigen und höchst populären Form des dezentralen Anbietens und Suchens von Daten geführt haben, soll an dieser Stelle ein Schritt weiter gegangen werden. Aufbauend auf existierenden P2P-Systemen soll eine dezentrale kollaborative Strukturierung von Daten ermöglicht werden.

Die Idee dabei ist folgende. An jedem Knoten des Netzwerks können Datenobjekte gespeichert und in einer individuellen Taxonomie (d.h. einem hierarchischen Klassifikationsschema) strukturiert werden. Jeder Benutzer kann somit eine völlig individuelle Sicht auf die Daten haben, unabhängig von allen anderen Benutzern. Um den Benutzern die Aufgabe der Datenverwaltung zu erleichtern, werden sie bei der Erstellung und Verwaltung ihrer persönlichen Taxonomie von intelligenten Agenten unterstützt. Basis für diese Unterstützung bilden wiederum persönliche Taxonomien anderer Nutzer. Ein typisches Beispiel wäre, dass ein Nutzer die Datenobjekte eines anderen Nutzers mit seiner eigenen Taxonomie strukturieren möchte, um z.B. festzustellen, ob dort für ihn interessante Daten liegen.

Es wäre aber auch möglich, dass ein Benutzer eine große Anzahl von bislang unstrukturierten Daten (z.B. Musikstücke, welche in einem Internetradio mitgeschnitten wurden) strukturieren möchte und dazu die Taxonomie eines anderen Benutzers anwendet. Aber auch klassische Aufgaben wie das Auffinden ähnliche Datenobjekte oder das Charakterisieren von Objekten kann auf diese Weise durchgeführt werden. So können beispielsweise Objekte die häufig gemeinsam in persönlichen Taxonomien vorkommen als ähnlich angesehen werden. Entsprechend können die Bezeichnungen von Kategorien in einer persönlichen Taxonomie als Basis für das Gewinnen von Stichwörtern genutzt werden.

Bezogen auf Musik könnten z.B. drei Nutzer ein Musikstück unter Jazz subsumieren und ein Nutzer unter Rock. Da das System die Nutzer nicht zwingt sich auf eine Bezeichnung zu einigen, können diese Meinungen koexistieren und werden nur durch statistische Verfahren vermittelt. Jazz würde z.B. in diesem Fall als relevantes Stichwort für das Musikstück ausgegeben, Rock hingegen als weniger relevantes.

1.2. Problembeschreibung

P2P-Netzwerke ermöglichen ein dezentrales Anbieten und Suchen von Daten. Um nach Daten suchen zu können, muss man dabei allerdings genau wissen, was man sucht. Dies ist häufig nicht der Fall. Im Internet haben sich als alternative Suchstrategie Webkataloge, wie Yahoo [6] oder Open Directory[7] etabliert, in denen die Daten (in diesem Fall Webseiten) hierarchisch in Kategorien strukturiert sind.

Diese Idee soll nun auf P2P-Netze übertragen werden. Konsequenterweise, kann in diesem Fall jeder Benutzer sein eigenes, lokales Klassifikationschema haben. Dieses reflektiert die individuelle Sicht dieses Nutzers auf die an seinem Knoten gespeicherten und angebotenen Daten.

Solche lokalen Klassifikationsschemata dienen einem Nutzer zunächst selbst dazu, seine Daten zu verwalten. Sie können aber natürlich auch für andere Nutzer nützlich sein. Im einfachsten Fall können sie anderen dazu dienen, sich durch die Datenkollektion dieses Nutzers zu browsen. Durch den Einsatz von Techniken des Maschinellen Lernens ergeben sich allerdings noch ganz andere Möglichkeiten. Beispielsweise könnte ein Benutzer A die Daten von einem Benutzer B in sein eigenes Klassifikationsschema einsortieren lassen, um so die Daten von B durch seine eigene Brille zu sehen.

Um eine solche Art von Verarbeitung zu ermöglichen, müssen die Daten zunächst geeignet kodiert werden. Dazu werden Methoden der Merkmalsextraktion angewendet. Für Musikstücke heißt das beispielsweise, dass Eigenschaften wie Tempo, Rhythmus, etc. aus den entsprechenden Mp3 Dateien extrahiert werden müssen, um diese zu beschreiben. Ein Lernalgorithmus kann dann diese Eigenschaften nutzen, um ähnliche Musikstücke zusammen zu gruppieren. Das YALE-System, welches hier am Lehrstuhl

[6] http://www.yahoo.com
[7] http://dmoz.org

erstellt wurde, stellt entsprechende Methoden zur Merkmalsextraktion zur Verfügung.

Aufgabe der PG ist es auf Basis des YALE-Systems die oben beschriebenen Funktionen auf ein bestehendes P2P-System aufzusetzen.

1.3. Zielsetzung

Das Ziel der Projektgruppe ist entsprechend folgendes: Basierend auf einer existierenden P2P-Plattform, welche Basisdienste wie Dateitransfer und Suche bereitstellt, sollen ein Protokoll und prototypische Agenten erstellt werden, welche kollaborativ und dezentral helfen, Daten in hierarchischen Strukturen zu verwalten, sowie neue relevante Datenobjekte zu finden und Beziehung zwischen Datenobjekten aufzudecken.

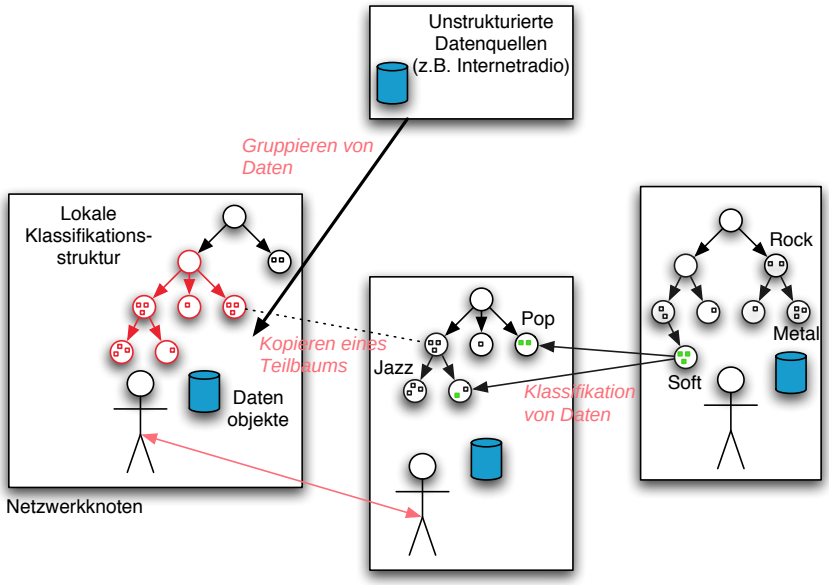


Abbildung 1.1.: Modellstruktur und Entwicklung

Abbildung 1.1 stellt die typische Struktur eines Modells dar.

Im einzelnen sind folgende Funktionen denkbar:

- Klassifikation von neuen Datenobjekten in eine bestehende persönliche Taxonomie
- Automatisches Erstellen einer personalisierten Taxonomie aus bislang unstrukturierten Daten
- Charakterisierung von Datenmengen durch Begriffe (Stichwörter)
- Auffinden interessanter Nutzer, bzw. Knoten im Netzwerk
- Empfehlen von Datenobjekten und Begriffen sowie das Auffinden von Beziehungen zwischen Objekten und zwischen Begriffen

Das System soll generisch sein und mit verschiedenen Datentypen umgehen können, allerdings wird der Schwerpunkt auf Multimediadaten liegen. Zur Lösung dieser Aufgabe können die Studenten auf das YALE-System zurückgreifen, welches am Lehrstuhl für Künstliche Intelligenz entwickelt wurde. YALE

ist eine Umgebung zur Durchführung von Experimenten für Maschinelles Lernen. Neben zahlreichen Klassifikations- und Clusteringverfahren beinhaltet YALE auch Operatoren zur Gewinnung von Merkmalen aus Text und Audiodaten, sowie verschiedene Verfahren zur Auswahl und zur Gewichtung von Merkmalen. Das System ist vollständig Java und XML [8] basiert und kann ohne großen Aufwand in neue Anwendungen eingebunden werden. Desweiteren soll auf ein bestehendes P2P-Netzwerk aufgebaut werden, welches bereits Basisfunktionalität wie Datenaustausch und Suche zur Verfügung stellt. Dieses System soll um die oben genannten Dienste erweitert werden.

Im einzelnen ist folgendes Vorgehen geplant:

- Es muss ein P2P-*Filesharing*-System ausgewählt werden, welches sich im obigen Sinne erweitern läßt. Besonders bietet sich die Java basierte Plattform JXTA für diese Aufgabe an, da diese bereits eine komplette Opensource Implementierung für das Austauschen und Suchen von Daten, sowie für das dynamische Verwalten von Benutzergruppen bereitstellt.
- Aufbauend auf dem gewählten System sollen XML-basierte Datenstrukturen und Kommunikationsprotokolle entwickelt werden, welche den Agenten den Austausch von relevanten Informationen ermöglichen.
- Um Verfahren des Maschinellen Lernens anwenden zu können, müssen die Datenobjekte zunächst geeignet kodiert werden. Dazu sind Verfahren der Merkmalsextraktion, welche im Rahmen des YALE- Systems am Lehrstuhl für Künstliche Intelligenz entwickelt wurden, anzupassen und einzubinden.
- Lernverfahren aus dem YALE System sollen angepasst werden um (eine Teilmenge der) Assistenzaufgaben zu lösen. Diese Verfahren sollen dann in intelligenten Agenten gekapselt werden, welche an jedem Knoten des Netzwerks ausgeführt werden.
- Um alle Systemkomponenten zusammenzuführen, soll eine prototypische Anwendungen implementiert werden.
- Das fertige System soll intensiv evaluiert werden, vor allem hinsichtlich der Frage, ob ein kollaborativer Ansatz (also das Verwenden von benutzererstellten Informationen) beim Strukturieren von Daten einem rein datenbasierten Herangehen überlegen ist.
- Sowohl die Gesamtarchitektur, als auch die einzelnen Komponenten sollen nach Standards der Softwareentwicklung dokumentiert und einer Qualitätssicherung durch die Studenten unterzogen werden. Auf diese Weise wird ein Einblick in professionelle Entwicklung und Verifikation von komplexen verteilten Anwendungen möglich.

1.4. Minimalziel

Das Minimalziel ist die Erstellung eines Systems, welches das lokale Speichern und Verwalten von Daten in Form von persönlichen Taxonomien erlaubt, sowie den Austausch von Daten und persönlichen Taxonomien zwischen Nutzern. Außerdem soll mindestens die automatische Klassifikation und das Gruppieren von Datenobjekten möglich sein.

1.5. Dank

Wir bedanken uns herzlich bei Prof. Dr. Katharina Morik und Dipl. Informatiker Michael Wurst für die Betreuung und Unterstützung der Projektgruppe 461. Desweiteren geht auch ein grosser Dank an Dipl.

[8] eXtensible Markup Language

Informatiker Ingo Mierswa, der sich immer Zeit für unsere Fragen bezüglich Yale genommen hat, und Fabian Mörchen für das zur Verfügung Stellen seiner Audiofeautures.

2. Ausgangslage

2.1. Persönliche Agenten und Wissenmanagement

2.1.1. Kollaborative Informationsagenten

Wikipedia definiert Agenten folgendermaßen: „Ein Agent (lat. Handelnder) handelt im Auftrag eines Anderen für dessen Interessen.“ In der Informatik unterteilt man grob in zwei Klassen von Agenten: Roboter (Hardware-Agenten) und Software-Agenten. Beispiele für Roboter sind z.B. Tribots oder Fabrikroboter. Diese Ausarbeitung beschäftigt sich allerdings hauptsächlich mit Software-Agenten, deren Vertreter nicht immer klar als solche zu erkennen sind, reicht das Spektrum doch von Spam-Filtern bis hin zu jeglicher Art von Assistenten.

Grundsätzlich sollten Agenten effektiv und robust sein. Ersteres meint, dass der Agent sein Ziel irgendwann erreicht, letzteres, dass er beständig sinnig agiert und nicht eine unerwartete Situation z.B. zum Totalabsturz führt. Als spezielle Eigenschaften sind noch zu nennen: Adaptivität (solche Agenten passen sich an ihre Umgebung an und verbessern sich mit der Zeit) und Kollaboration (Agenten, die gezielt gemeinsam arbeiten).

Mensch-Maschine-Interaktion

Im Fall, dass Agenten direkt für einen Benutzer arbeiten, stehen zwei weitere Eigenschaften im Mittelpunkt: Kompetenz und Vertrauen. Kompetenz bedeutet wie schon Effektivität, dass der Agent sein Ziel irgendwann erreichen sollte, aber auch, dass er nicht mehr tun sollte, als ihm aufgetragen wurde (er darf seine Kompetenzen nicht überschreiten). Dabei ist natürlich auch zu bedenken, dass der Agent nichts tun sollte, was sein Benutzer schneller und besser erledigen könnte. Desweiteren darf der Agent nicht zwischen Nutzer und Anwendung stehen. Wenn der Nutzer etwas selbst machen will, dann muß es ihm möglich sein.

Da der Agent für die Interessen des Benutzer handelt, muß der Benutzer diesem Vertrauen können. Dazu ist es wichtig, dass er immer das Gefühl hat, die Kontrolle zu besitzen. Der Agent muß die Sicherheit und die Privatssphäre des Benutzers schützen. Am günstigsten ist es, wenn der Benutzer ein Verständnis dafür entwickeln kann, wie der Agent seine Ziele erreicht.

Man unterscheidet hier zwischen expliziter und implizierter *Responsiveness*. Bei ersterer werden Befehle so ausgeführt, wie sie gegeben wurden: statisch, deterministisch, ohne jegliche Interpretation. Agenten mit impliziter *Responsiveness* hingegen interpretieren ihre Anordnungen je nach Benutzer und Situation.

Persönliche Agenten

Persönliche Agenten unterscheiden zwischen verschiedenen Benutzern und ihren *subjektive* Zielen. Sie beziehen in ihre Handlungen nicht nur das Wissen über die Umgebung mit ein, sondern auch das Wissen über ihren Benutzer.

Wissen kann hier in drei verschiedenen Formen vorliegen:

hardcoded: Fest implementiertes Wissen ist (zwangsläufig) statisch und objektiv in dem Sinne, dass es für jeden Benutzer gleich ist.

regel-basiert: Regelbasiertes Wissen führt zu halb-autonomem Verhalten. Der Benutzer legt die Regeln fest, dafür muß er aber ersteinmal erkennen, dass er dazu die Möglichkeit hat, er muß wissen, wie es funktioniert und: er muß selber die Mühe aufwenden, die Regeln zu definieren.

adaptiv: Wenn der Agent selber aus seinen Erfahrungen lernt, führt dies zu effizienterem Verhalten. Der Agent ist vollständig autonom und hoch dynamisch.

Adaptive und kollaborative Agenten

Hauptparadigma der adaptiven Agenten ist, dass sie sich mit der Zeit verbessern sollen. Dazu lernen sie auf unterschiedliche Weisen: Die (für den Benutzer) einfachste ist, dass der Agent ihm erstmal bei der Arbeit *über die Schulter schaut* und erst selbständig aktiv wird, wenn er genügend Informationen über den Benutzer gesammelt hat, um zuverlässig Entscheidungen treffen zu können. Eine andere Variante ist, dem Agenten explizite Beispiele zu geben, wie er sich verhalten soll. Und zuletzt kann der Agent eigenständig experimentieren.

In Multiagenten-Systemen können Agenten miteinander kooperieren. Dazu kommunizieren sie miteinander und tauschen z.B. Benutzerempfehlungen oder eigene Erfahrungen aus.

Vorteile davon sind eine schnellere Lernkurve und die Möglichkeit, Entscheidungen auf Basis von Empfehlungen anderer Benutzer zu treffen. Nachteil des Ganzen ist, dass Sicherheit und Privatssphäre eventuell nicht mehr geschützt sind.

2.1.2. Wissensmanagement und künstliche Intelligenz

Einleitung

Eine der wertvollsten Ressourcen eines Unternehmens ist in der heutigen Zeit Wissen. Das Wissen über effiziente Produktionsmethoden oder das Kaufverhalten von Kunden, um nur einige Beispiele zu nennen, ist ein entscheidender Wettbewerbsvorteil.

Um Wissen innerhalb eines Unternehmens effizient zu nutzen, ist ein ungehinderter Austausch zwischen den Mitarbeitern und ein einfacher Zugang zu Informationen unabdingbar. Allerdings sind Unternehmen in der heutigen Zeit oft geographisch weit verteilt, was diesen Bedingungen im Wege steht. Dieses und viele andere Probleme bei Erwerb, Archivierung und Verwendung von Wissen sollen durch den Einsatz sogenannter Wissensmanagement-System (KM-Systeme) gelöst werden. Unternehmen auf der ganzen Welt machen große Investitionen für die Entwicklung und Wartung eigener KM-Systeme.

Die Funktion eines KM-Systems ist es Wissen zu sammeln, zu archivieren und schließlich einer unternehmensweiten Nutzung und Wiederverwendung zugänglich zu machen. Dieses Wissen besteht beispielsweise in Know-How und Erfahrung einzelner Mitarbeiter bei der Lösung von Problemen, oder in unternehmenskritischen Fakten und Zahlen.

Bei der Einsatz eines KM-Systems im Unternehmen gilt es zu beachten, daß es sich einfach in bestehende Prozessen integrieren lassen sollte. Desweiteren muss die Nutzung und der Beitrag von Wissen durch ein leicht zugängliches, einfach zu bedienendes System attraktiv gemacht werden.

Ziel des Einsatzes eines KM-Systems ist die Steigerung der Effizienz (z.B. bei der Lösung von Problemen als auch bei Entscheidungen) als auch eine Steigerung der Produktivität und eine Verbesserung des Service. Langfristig soll das Personal motivierter sein und schnelleres Lernen möglich werden.

Anwendungen der KI

Anwendungen der Methoden der künstlichen Intelligenz gibt es im Wissensmanagement viele. So kann das Wissen darüber, wie Wissensbasen aufgebaut werden und in welcher Weise Wissensaquise,

Wissensrepresentation und Inferenz geschehen dazu eingesetzt werden, um die technische Basis eines Wissensmanagement-Systems zu legen.

Darstellung einer Wissensbasis Es gibt formale und weniger formale Möglichkeiten das Wissen eines Anwendungsbereichs darzustellen. Eine Möglichkeit bietet die *Description Logic*. Sie besitzt eine formale, logik-basierte Semantik (im Gegensatz zu vielen anderen Darstellungssprachen). Die Schlüsselelemente der Syntax dieser Logik sind die TBox und die ABox.

Die TBox enthält das „Vokabular" des Anwendungsbereichs in Form von atomaren Konzepten, welche eine Menge von Individuen darstellen, und atomaren Rollen, die diese Individuen (binär) in Relation zueinander setzen.

Die ABox dient der Einführung einzelner Individuen und dem Treffen von Annahmen über deren Eigenschaften.

Austausch von Wissen Mittels einer Darstellungsform von Wissen, wie dem im letzten Abschnitt vorgestellten Formalismus, lassen sich Ontologien repräsentieren. Eine Ontologie ist ein formales, vereinfachtes Modell eines Anwendungsbereichs. Sie definiert eine Menge von Begriffen (Vokabular), deren Bedeutung und geltenden Zusammenhänge.

Ontologien dienen einer Vereinfachung des Austauschs von Wissen. Dies soll an einem Beispiel verdeutlicht werden:

Zwei Einheiten tauschen Zeichenketten aus. A macht eine Aussage über „JAGUAR" und meint damit ein Tier. B denkt bei „JAGUAR" an das Auto.

A und B legen unterschiedliche Korrespondenzen zwischen „Begriff" und „Gegenstand" zugrunde. Die Aufgabe einer Ontologie ist nun die Reduzierung der möglichen Korrespondenzen auf genau eine. A und B einigen sich auf diese Ontologie (Commitment). B versteht nun, was A mit „JAGUAR" meint. Dies ermöglicht eine effektivere Kommunikation.

Wissensmanagement in der Praxis

Früher befand sich das Wissen eines Unternehmens auf Papier und in den Köpfen der Mitarbeiter. Die Probleme waren begrenzte Zugänglichkeit und die Schwierigkeit das Wissen auf dem aktuellem Stand zu halten. Eine Wiederbenutzbarkeit war kaum gegeben.

Erste Systeme in Richtung eines KM-Systems bildeten die sogenannten *data warehouses*, welche die Transaktionsdaten eines Unternehmens über viele Jahre archivierten und sie so einer Analyse zugänglich machten (Stichwort: *knowledge discovery*).

Ein Beispiel für den Einsatz von *knowledge discovery* ist ODIE (On-Demand Information Extractor). Dieser analysiert Zeitungen, findet Informationen über wichtige Ereignisse in der Wirtschaft und stellt diese Informationen zusammen.

Beispiele für Wissensbasen im Unternehmen sind Lessons-Learned-Datenbanken. Diese enthalten detaillierte Informationen zu durchgeführten Projekten (egal ob erfolgreich oder nicht). Weitere Beispiele sind Best-Practices-Datenbanken, Expertise-Verzeichnisse und Projektarchive. Dabei enthalten Projektarchive alles Wissenswerte zu einem Projekt, d.h. Kundendaten, Road Maps, Berichte, Präsentationen, Daten, Erfolge/Fehlschläge.

2.2. Verteilte Anwendungen

2.2.1. Vorstellung und Vergleich von Agentensystemen und P2P Computing

Die Agenten

Wenn man über „Agenten“ spricht, versteht jeder Mensch etwas anderes. Es gibt viele verschiedene Definitionen und Deutungen. Ganz allgemein ist ein Agent jemand, der das Recht hat für einen Anderen zu handeln. In der Technik ist ein Agent etwas, das seinem „Erschaffer“ die Arbeit abnimmt. Dazu muss er in der Lage sein, seine Umgebung wahrzunehmen und entsprechend darauf zu reagieren.

Die technische Definition von Agenten ist: Ein Agent ist ein Computersystem, das seine Umgebung mittels Sensoren wahrnimmt und darauf autonom und rational mittels Effektoren reagiert.[RN02]

Die Agentensysteme werden in viele verschiedene Gruppen unterteilt. Die einzige Gemeinsamkeit ist die Autonomie. Obwohl auch Roboter und Automaten (Ampelanlage) zu den Agenten gehören werden hier nur die Softwareagenten behandelt.

Ein Softwareagent ist ein Stuck Software, dass mehr oder weniger folgende Eigenschaften hat: Nach Übermittlung der Aufgabe handelt der Agent autonom, zur Lösung der Aufgabe kommuniziert er mit anderen Agenten und gegebenenfalls anderen Systemen (Sensoren) der Umwelt, er ist mobil, kann sich an verändertes Umfeld und Fragestellung anpassen, lernt im Laufe seines Lebenszyklus dazu. Sein Handeln muss auf einem Beobachter „intelligent wirken“. Die Softwareagenten lassen sich in viele verschiedene Klassen unterteilen. Bei kollaborierenden oder konkurrierenden Agenten ist jeder Agent autonom. Die einzelnen Agenten müssen sich in einem Gespräch/Verhandlung untereinander abstimmen und arbeiten gemeinsam auf eine Sache hin. Sie müssen selbständig auf Änderungen reagieren und ohne einen Einfluss von außen ihr Verhalten andern können. Als Beispiel kann jede verteilte Anwendung genommen werden. Bei den mobilen Agenten steht der Ort der Ausführung noch nicht fest. Der Programmcode liegt auf einem Server und wird vor der Ausführung auf den Zielrechner übertragen. Als Beispiel kann man eine Postscript-Datei nehmen, die an einen Drucker geschickt und erst dort ausgeführt wird. Die Interface-Agenten sollen einem User die Interaktion mit einer Anwendung erleichtern. Sie können ihn bei der Eingabe unterstützen oder Hilfe bei den Einstellungen bieten (Wizards). Der Agent kann auch das Nutzerverhalten analysieren und sich so seinen Gewohnheiten anpassen. Der Spamfilter ist ein gutes Beispiel für den Interface-Agenten, da er dem Nutzer die Sortierung der Mails abnimmt und das Filterverhalten an die Bedürfnisse des Nutzers anpasst. Reaktive Agenten sind modular aufgebaut. Es gibt keinen festen Weg um eine Aufgabe zu losen. Zur Ausführungszeit wird die Umgebung analysiert und entsprechend drauf reagiert. Es werden dann die nötigen Module ausgeführt, um für die aktuelle Lage das beste Ergebnis zu erzielen.

Mehrere Agenten bilden zusammen ein Agentensystem. Als Modell kann man sich eine objektorientierte Sprache vorstellen, wo jedes Objekt ein Agent ist, und erst das Zusammenspiel aller Agenten ein fertiges Programm ergibt. Es gibt wieder eine Unterscheidung zwischen statischen und mobilen Agentensystemen. Bei statischen Systemen können die Agenten zwar auf mehreren Rechnern ausgeführt werden, der Programmcode befindet sich aber immer auf dem selben Rechner. Bei mobilen Systemen kann der Programmcode von Rechner zu Rechner kopiert werden und so zur Ausführungszeit verändert werden.

Um effizient zu arbeiten, müssen die meisten Agenten miteinander kommunizieren. Um diese Aufgabe zu erleichtern wurde gemeinsame Standards und Protokolle fur die Kommunikation festgelegt. Beispiele dafür sind die Knowledge Query and Manipulation Language (KQML) und Knowledge Interchange Format (KIF). Beide werden verwendet, um in einer verteilten Umgebung Wissen zwischen den einzelnen Agenten auszutauschen.

Um den Informationsaustausch zu koordinieren gibt es viele Ansätze wie

- Blackboard

- Advertise oder Subscribe
- Contract Net
- Brokering und Matchmaking

Jedes dieser Ansätze unterteilt alle Agenten im Netz in mehrere Gruppen. Die Anbieter stellen Ressourcen zur Verfügung, und die Nachfrager fragen diese an. Dabei ist die Rolle nicht fest. Sie kann sich je nach Situation jederzeit ändern. Eine dritte Gruppe von Agenten übernimmt die Koordination und sorgt dafür, dass die Ressourcen effizient verteilt werden.

P2P-Systeme

Wenn man eine Anwendung ausführen will, kann das lokal auf den Rechner geschehen, oder man verbindet sich zu einem Netz, um zusätzliche Informationen oder Rechenleistung zu erhalten.

Die herkömmliche Netzstruktur beruht auf dem Client/Server Prinzip. Ein Server ist mehreren Clients verbunden. Die Clients können ihre Anfragen an den Server schicken, und kriegen eine Antwort zurück. Das System kann flach oder hierarchisch aufgebaut werden. Bei einem hierarchischem System kann ein Server die Anfrage an weitere Server weiterleiten. (DNS oder Proxy) Bei einer flachen Struktur kriegt der Client die Antwort sofort. (FTP) Das Client/Server System kann sehr leicht erstellt und gewartet werden. Der Nachteil ist allerdings der Single-Point-of-Failure. Fallt der Server aus, ist das ganze Netz lahmgelegt.

Um dieses Problem zu vermeiden kann man ein hybrides P2P System nehmen. Hier können die einzelnen Clients auch untereinander kommunizieren. Für den Zugang zum Netz wird aber immer noch ein Server benötigt, der die Infos über die einzelnen Rechner speichert. Fällt er aus, kann zwar kein neuer Rechner sich an dem Netz anmelden, aber die im Netz vorhandenen Rechner können ohne Probleme weiter funktionieren. Der nächste Ansatz geht noch weiter, er entfernt den Server komplett und erstellt ein pures Peer-to-Peer System, das ohne Server funktioniert. Peer-to-Peer ist ein System zum direkten Austausch von Daten und Ressourcen zwischen vernetzten Rechner. Der Name kommt aus dem englischen (peer = Gleicher) und bedeutet soviel wie „von gleich zu gleich“. Das wichtigste Merkmal von einem P2P Netz ist die direkte Verbindung der einzelner Rechner untereinander. Jeder Rechner fungiert gleichzeitig als Anbieter und als Nachfrager von Ressourcen. Diese können Dateien, aber auch nicht genutzte Rechenzeit oder Speicher sein.

Inzwischen sind P2P Systeme weit verbreitet. Eine der bekanntesten Anwendungen ist Bittorrent, die eine schnelle Dateiverteilung bei einer niedrigen Auslastung des Servers ermöglicht. Damit die einzelnen Clients im Netz kommunizieren können, müssen sie nicht unbedingt gleich sein. Es reicht, wenn die einzelnen Rechner das gleiche Protokoll für die Nachrichtenübertragung verwenden. Ein Ansatz dafür wurde mit JXTA gemacht. JXTA bietet eine Plattform, über die einzelne Rechner kommunizieren können. Dafür werden Protokolle definiert, die jeder Teilnehmer beherrschen muss. Zwar gibt es eine Java-Implementierung, so dass die Plattformunabhängigkeit erreicht wird. Leider ist die Benutzung alles andere als trivial und erfordert eine lange Einarbeitungszeit.

Der Vergleich

Obwohl beide Systemtypen aus völlig unterschiedlichen Richtungen kommen, sind ihre Gemeinsamkeiten sehr stark. Die Multiagentensysteme stammen aus dem KI-Bereich und setzten sich mit der Frage auseinander, wie man die Kommunikation zwischen einzelnen Agenten möglichst effizient gestalten kann. Die P2P-Systeme kommen von den verteilten Anwendungen, wo versucht wurde ein ausfallsicheres Netz zu konstruieren. Da ein Agent autonom handeln können muss, kommt jede Lösung, die auf dem Client/Server Prinzip basiert nicht in Frage, weil dadurch eine starke Abhängigkeit von den anderen Agenten entsteht. Der P2P-Ansatz bietet sich an, da er für Gleichberechtigung von einzelnen Agenten sorgt und die Kosten für die Kommunikation gleichmäßig verteilt. In den Netzwerksystemen

setzt sich die P2P Lösung immer mehr durch, weil dadurch Kosten gespart werden können. Mehrere kleine Rechner sind günstiger als ein Großer. Insgesamt nähern sich die beiden Bereiche immer mehr an. Die Agenten kommen nicht mehr ohne das Netzwerk aus, und die Peer-to-Peer Systeme werden immer unabhängiger und intelligenter.

2.2.2. Überblick : Konkrete, erweiterbare, Java basierte P2P/Agentensysteme

Im folgendem Abschnitt soll ein kurzer Überblick über verschiedene frei verfügbare P2P- bzw. Agentensystem gegeben werden. Was leisten solche Systeme, wie gut sind sie entwickelt und wie gut lassen sie sich erweitern? Ein besonderer Schwerpunkt liegt auf JXTA.[1]

Agentensysteme

agentTool agentTool[2] ist ein grafisches System zur formalen Spezifikation eines Agentensystems. Ein Systemdesigner spezifiziert die benötigten Strukturen und das erwünschte Verhalten eines Multiagentensystems und aus dieser grafischen Spezifikation wird dann halb-automatisch ein Java-Code-Gerüst für dieses System generiert. agentTool wurde im AFIT Artificial Intelligence Laboratory entwickelt. Es basiert auf Java 1.3 und wird noch weiter entwickelt.

JADE Das Java Agent DEvelopment Framework[3](JADE), ist eine Open-Source Entwicklungsumgebung für Multi-Agentensysteme. JADE ist vollständig in Java implementiert und entspricht den Spezifikationen der Foundation for Intelligent Physical Agents (FIPA). Die FIPA ist eine 1996 gegründete *non-profit* Organisation, die das Ziel verfolgt, die Entwicklung und Spezifikation von Agententechnologie zu fördern. Auch JADE wird stetig weiter entwickelt.

Java Agent Template Lite Java Agent Template Lite[4] (JATLite) ist ein in Java geschriebenes Programmpaket, das einen Menge von Java Templates und eine Java Agenten Infrastruktur enthält. Mit Hilfe dieser Templates, die dem Benutzer eine Vielzahl von vordefinierten Klassen zur Verfügung stellen, sollen Softwareagenten konstruiert werden, die über das Internet kommunizieren und als Applets auf den unterschiedlichsten Plattformen laufen können.

Durch einen modularen Aufbau und die Aufteilung in Layer ist das Paket gut erweiterbar. Entwickelt wurde JATLite am Center for Design Research an der Stanford Universität.

Aglets Aglets[5] sind mobile Agenten. Sie werden von einem Computer zu einem Remotecomputer geschickt werden, um dort Programmcode auszuführen. Auf dem Remotecomputer angekommen erhalten sie, je nachdem welche Zugriffsberechtigung sie haben, Zugriff auf die lokalen Dienste und Daten. Der Remotecomputer kann auch als Vermittler dienen und verschiedene mobile Agenten zusammen bringen.

Aglets ist eine Entwicklung von IBM, die inzwischen Open-Source ist und unter SourceForge zu finden ist. Auch sie basieren vollständig auf Java.

[1] JXTA ist ein sprach- und plattformunabhängiges Protokoll für P2P-Netzwerke und wurde von Sun entwickelt wurde. Weitere Informationen finden sich unter `http://www.jxta.org`

[2] `http://macr.cis.ksu.edu/projects/agentTool/agentool.htm`

[3] `http://jade.tilab.com/`

[4] `http://java.stanford.edu/`

[5] `http://aglets.sf.net`

P2P Netzwerke

Napster Napster war eines der ersten P2P-Netzwerke. Es ist aber kein reines P2P-Netz, sondern ein sogenanntes hybrides System. Das heißt, dass das Client-Server-Prinzip noch erkennbar ist. Der Client durchsucht den lokalen Computer nach Musikdateien und schickt diese Liste an einen Server im Internet. Stellt ein Client eine Anfrage, erhält er vom Server eine Liste mit den IP-Adressen, die die Clients identifizieren, die ein passendes Ergebnis haben. Die Clients können sich dann direkt miteinander verbinden und die Daten austauschen. Insgesamt besaß Napster ein zentrales Servercluster, das aus ca. 160 Servern bestand. Inzwischen ist Napster in einen kostenpflichtigen Dienst umgewandelt worden. Trotzdem wird das Protokoll auch heute noch genutzt.

Freenet Freenet[6] ist dagegen ein reines P2P-Netzwerk. Es ist vollkommen dezentral und alle Peers haben die gleichen Funktionen. Ziel von Freenet ist eine Zensur zu verhindern und den anonymen Austausch von Informationen zu ermöglichen. Dabei werden die Daten nicht auf dem lokalen Peer gehalten, sondern mehrfach auf anderen Peers gespeichert. Dies erhöht die Verfügbarkeit und zugleich die Anonymisierung, denn der Halter der Daten ist nicht der Bereitsteller. Dateien haben eindeutig bestimmte IDs, sogenannte Content Hash Keys. Auch der Datenaustausch findet anonym statt. Die Verbindung läuft über mehrere Knoten und jede P2P-Verbindung ist verschlüsselt. An keinem Zwischenpunkt sind die Endpunkte bekannt. Daher ist Freenet sehr sicher, aber auch sehr langsam.

Gnutella Auch Gnutella[7] ist ein reines P2P-Netzwerk. Gnutella ist eine Weiterentwicklung des Napster-Protokolls. Peers bilden Gruppen durch eine Ping/Pong-Methode. Diese Gruppen überlappen sich und bilden so zusammen das Gnutella-Netzwerk. Der Einstieg in ein solches Netzwerk erfolgt über sogenannte well-known Peers. Sucht ein Client eine Datei im Gnutella-Netzwerk, dann stellt er eine Anfrage zunächst an die unmittelbaren Nachbarn. Werden diese nicht fündig, leiten sie ihrerseits die Anfrage weiter. Dadurch wird ein erheblicher Teil der Netzwerkkapazitäten für Suchanfragen verbraucht. Das ursprüngliche Protokoll wurde inzwischen um einige Features erweitert. So ist zum Beispiel die P2P-Verbindung komprimiert und dem Problem mit den überflutenden Suchanfragen wird mit Ultrapeers begegnet.

JTella JTella[8] ist ein Gnutella-Client, der in Java programmiert worden ist. Früher wurde der JTella-Kern von dem bekannten Client Morpheus genutzt. JTella bietet eine Java API für Gnutella-Netzwerke und kann daher für einfache Entwicklungen von javabasierten Gnutella Tools eingesetzt werden.

KaZaA KaZaA[9] befindet sich zwischen Napster und Gnutella. Es ist ein gemischtes P2P-Netz. Sogenannte Super-Nodes vereinfachen die Kommunikation im Netzwerk und ersetzen den zentralen Server, der bei Napster benötigt wurde. Wie einige andere Clients auch basiert KaZaA auf dem FastTrack-Protokoll. Das FastTrack-Protokoll ist eine Erweiterung des Gnutella-Protokolls. Eine Verbindung geht immer über eine feste IP-Adresse von bekannten Super-Nodes.

JXTA

JXTA ist ein Projekt zur Standardisierung von P2P-Anwendungen durch frei zugängliche Protokolle. Als Protokoll ist JXTA unabhängig von Programmiersprache, Betriebssystem und darunterliegendem Transportprotokoll, wie beispielsweise TCP/IP oder Bluetooth. JXTA bietet die Möglichkeit, weitere

[6] `http://freenetproject.org`
[7] `http://www.gnutella.com`
[8] `http://jtella.sf.net`
[9] `http://www.kazaa.com`

Teilnehmern zu entdecken und Firewalls zu überwinden. Der Name JXTA leitet sich aus dem englischen 'juxtapose' ab. juxtapose bedeutet soviel wie Nebeneinanderstellung.

Peer Ein Peer ist ein Knoten im System. Ein Knoten kann alles sein, was mindestens eines der JXTA-Protokolle implementiert, wie beispielsweise Computer, PDAs, Handys oder Drucker. Spezielle Peers, wie der Rendezvous Peer, dienen zum Entdecken neuer Teilnehmer oder zum Überwinden von Firewalls, wie der Relay Peer. Jeder Peer erhält eine weltweit eindeutige Id.

Peer Groups Mehrere Peers können zu einer Peer Group zusammengefasst werden, um gemeinsam zu arbeiten. Ein einzelner Peer kann Mitglied in beliebig vielen Peer Groups sein. Jeder Peer muss die Protokolle seiner Gruppe implementieren und ist automatisch Mitglied der *NetPeerGroup*. Auch Peer Groups erhalten eine eindeutige ID.

Pipes Pipes sind virtuelle, unidirektionale Kanäle zwischen zwei oder mehreren Peers. Ihre Ein- und Ausgänge können dynamisch an verschiedene Endpunkte gebunden werden.

Messages Durch die Pipes werden Messages verschickt. Sie sind Objekte, über die Peers kommunizieren. Als Format wird meistens XML benutzt, aber auch binäre Daten können ausgetauscht werden.

Advertisements Ein Advertisement ist ein XML strukturiertes Dokument, das Ressourcen benennt, beschreibt und deren Existenz bekannt gibt. Ressourcen sind dabei Peers, Peer Groups, Pipes und Services.

Zentrale Protokolle

- Das Peer Discovery Protocol wird verwendet, um publizierte Advertisements innerhalb einer Peer Group aufzufinden.
- Das Peer Information Protocol bietet die Möglichkeit, Informationen über andere Peers abzufragen.
- Das Peer Resolver Protocol erlaubt es einem Peer, generische Anfragen zu verschicken und zu empfangen.
- Das Pipe Binding Protocol hat die Funktion, einen virtuellen Kommunikationskanal (Pipe) an einen Kanalendpunkt (Endpoint) zu binden.
- Das Endpoint Routing Protocol definiert eine Reihe von Nachrichten, die es möglich machen, eine Route von einem Quell-Peer zu einem Ziel-Peer aufzubauen.
- Das Rendezvous Protocol wird verwendet um, Messages innerhalb einer Peer Group zu versenden.

2.2.3. Algorithmen zur Verteilten Suche und zum Verteilten Indexieren

Das Problem der Verteilten Suche stellt sich, wenn die Daten, die ein Rechner benötigt, nicht lokal gehalten werden, sondern in einem Netzwerk über mehrere Rechner verteilt sind. Um effizient an die Daten heranzukommen, muss man wissen, wonach man suchen will, und wo genau sich diese Daten befinden. Dafür werden bestimmte Merkmale der Daten in einem Index gespeichert. Mögliche Einträge im Index sind Dateiname und Größe, ein Hashwert der Dateien (CRC32, MD5, SHA-1,...) und die Dateibeschreibung. Die Dateibeschreibung hängt sehr stark von den zu indexierenden Daten ab. Neben

den Angaben zum Inhalt können bei Audiodaten zum Beispiel die Bitrate, die Länge und der verwendete Codec gespeichert werden.

Um den Datenindex zu verwalten, gibt es mehrere Ansätze. Ein möglicher Ansatz ist der zentrale Indexserver. Dieser verwaltet eine Liste aller Rechner im Netz und den auf ihnen gespeicherten Daten. Jeder Rechner, der sich zu dem Netz verbinden will, muss zuerst lokal seine vorhanden Dateien indexieren und diese Information an den Server senden. Danach können auch Suchanfragen an den Server geschickt werden, um die Daten zwischen den einzelnen Rechnern auszutauschen. Der Vorteil dieses Ansatzes ist vor allem der niedrige Overhead an Daten, die die einzelnen Rechner austauschen müssen. Die Suchanfragen werden von dem Zentralrechner schnell bearbeitet und man erhält auf jeden Fall alle Suchergebnisse zurück. Der große Nachteil ist der Single-Point-of-Failure. Fällt der Zentralserver aus, bricht sofort das ganze Netz zusammen. Außerdem können sehr leicht Nutzerprofile erstellt werden, da sich jeder Nutzer immer an den gleichen Rechner anmelden muss.

Um die Stabilität des Netzes zu erhöhen, kann der zentrale Server durch mehrere kleinere Server ersetzt werden. Jeder Client verwaltet jetzt eine Liste mit aktuellen Servern und verbindet sich zu einem von ihnen. Fällt ein Server aus, wird einfach zu einem anderen Server verbunden, ohne dass die Stabilität des Netzes beeinträchtigt wird. Der Nachteil ist, dass ein Server nur einen Teil des Indexes speichern kann. Man bekommt also nicht alle Suchergebnisse, wenn man nicht alle Server hintereinander abfragt.

Einen ganz anderen Ansatz verfolgt man mit einem komplett serverlosen Netzwerk. Jeder Rechner im Netz fungiert als ein Indexserver für seine eigenen Daten. Jeder Rechner ist mit mehreren anderen Rechnern verbunden und sendet seine Suchanfragen an alle ihm bekannten Ziele, die diese Anfrage wiederum weiterleiten. Dabei entsteht ein großer Overhead an Daten. Um ihn in Grenzen zu halten, hat jede Suchanfrage eine TTL (Time to Live). Wird nach einer gewissen Zeit kein passender Eintrag im Index gefunden, wird die Anfrage einfach verworfen. Das führt aber zu einer Inselbildung der einzelnen Clients, so dass jeder Nutzer nur Antworten aus einem Teil des Netzes erhält. Um die Reichweite der Suchanfragen zu erhöhen und den Overhead für die einzelnen Clients zu verringern, kann man die einzelnen Rechner in zwei Kategorien einteilen. Der Leaf ist dann mit einem Supernode verbunden und sendet seine Suchanfragen nur an diesen weiter. Die Supernodes sind untereinander verbunden und leiten Suchanfragen an andere Supernodes und an die verbundenen Leafs weiter.

Eine gute Lösung ist also die, die ohne einen zentralen Server auskommt und den Bandbreiten-Overhead minimiert. Diese Vorgaben wurde versucht mit Content-Addressable Networks [RFH+01] zu erreichen. Man erstellt ein Koordinatensystem und unterteilt dieses in Zonen. Jeder Rechner im Netz bekommt eine Zone zugewiesen und verwaltet diese. Dabei ist jeder Knoten mit den Knoten der Nachbarzonen verbunden. Wenn ein Rechner sich zum Netz verbinden will, wird eine bestehende Zone geteilt und der neue Knoten erhält einen Teil der alten Zone und alle damit verbunden Indexdaten. Meldet sich ein Rechner ab, wird seine Zone mit einer Nachbarzone verbunden und seine Indexdaten an den zuständigen Nachbarknoten übertragen.

Will man jetzt eine Datei in das System einfügen, muss zuerst ein Hashwert berechnet und dieser dann auf das Koordinatensystem gemappt werden. Danach hat die Datei einen festen Platz im Netz. Die Indexdaten werden also an den Rechner übertagen, der diese Koordinatenzone verwaltet. Die Suche gestaltet sich einfach. Man muss zu den zu suchenden Daten den Hashwert berechnen. Danach steht die Zone mit den Indexdaten fest und die Suchanfrage kann direkt an den richtigen Knoten weitergeleitet werden. Das System kann noch weiter verbessert werden. Man kann die Dimensionsanzahl in dem Koordinatensystem erhöhen und so die Suchpfade verkürzen. Dadurch steigt aber auch die Anzahl der Nachbarknoten, mit denen man verbunden ist. Man kann parallel mehrere Koordinatensysteme verwenden, so dass jeder Knoten mehrere Koordinaten hat. Dadurch verwalten mehrere Knoten eine Zone und das Netz wird durch den Ausfall eines Knotens nicht beeinträchtigt. Man kann auch die Zonengröße nach der Bandbreite der Knoten wählen, sodass langsame Rechner für kleine und schnelle Rechner für große Zonen zuständig sind. Populäre Anfragen können in mehreren Knoten gespeichert werden, um die Last von einzelnen Knoten zu nehmen und die Anfragen schneller beantworten zu können. Leider hat dieses System auch einen gravierenden Nachteil. Da jeder Datei ein Hashwert zugeordnet wird, ist die Suche nicht ganz einfach. Man muss schon das genaue Kriterium kennen, das exakt zu dieser Datei

führt. Suchanfragen auf gut Glück („Gib mir alle AVIs mit mehr als tausend Quellen“) funktionieren bei diesem System nicht.

Als letzter Ansatz soll hier noch Freenet [CSWH00] vorgestellt werden. Dieses System ermöglicht es, anonym Dateien im Netz anzubieten. Die Anonymität wird durch eine komplizierte Schlüsselvergabe realisiert. Für jeden Knoten im Netz wird ein Schlüssel generiert. Der Knoten verbindet sich dann zu anderen Knoten mit ähnlichen Schlüsseln. Auch für jede Datei wird ein Schlüssel generiert und die Datei mit diesem Schlüssel verschlüsselt. Danach wird ein Hashwert erstellt und die Datei zu einem Knoten mit dem ähnlichsten Schlüssel hochgeladen. Die Knoten speichern also nicht den Index, sondern die Datei selber. Jeder Rechner muss also einen Teil seines Festplattenspeichers für fremde Daten bereitstellen, kann aber mit den Daten nichts anfangen, da ihm der Schlüssel fehlt, um diese Daten zu entschlüsseln. Da die Daten nicht mehr lokal gespeichert werden, hat der Anbieter keine Kontrolle mehr über die Verbreitung. Es kann also passieren, dass selten nachgefragte Daten aus dem Netz verschwinden. Um die Quelle einer Datei zu verschleiern, werden mehrere andere Knoten als Proxy benutzt, bevor die Datei an ihrem Zielort ankommt. Leider erhöht dieses Verfahren den Bandbreitenverbrauch um ein Vielfaches. Außerdem kann man im Netzwerk nicht nach Dateien suchen, da man sie nicht einfach entschlüsseln kann. Man ist also auf externe Quellen für die Schlüssel und die dazu gehörenden Hashwerte für die Dateien angewiesen. Dafür bietet das System eine Möglichkeit, um Daten wirklich anonym anzubieten.

2.2.4. WLANs (Wireless Local Area Networks)

Netzwerke kann man aus der Computerwelt nicht mehr wegdenken. Neben drahtgebunden Netzwerken gibt es seit einigen Jahren auch drahtlose Netzwerke, welche in diesem Abschnitts genau beschrieben werden sollen. Sie besitzen eine Reihe von Vorteilen, da das Netzwerkkabel fehlt[Rot02].

- Die Netzwerkverkabelung ist ein Kostenfaktor.
- Die Netzwerkverkabelung muss schon bei der Planung konzipiert werden.
- Manchmal ist das Verlegen von Kabeln aufwendig oder sogar unmöglich.
- Die Reichweite von Netzwerkkabeln ist auf ein Gebäude beschränkt.

Neben den vielen Vorteilen haben drahtlose Netze aber auch Nachteile.

- Drahtlose Netze haben im Gegensatz zur drahtgebundenen Variante eine geringe Bandbreite und eine signifikant höhere Fehlerrate.
- Da Funksignale über das lokale Netz hinaus abgestrahlt werden, können Übertragungen leicht abgehört werden.
- Bei mobilen Geräten mit eigener Stromversorgung erfordert die Funkübertragung ein erhebliches Maß an Batteriestrom.
- Die Abstrahlung von Funksignalen kann u.U. andere Systeme stören.

Wenn man die beiden Varianten miteinander vergleicht, kann man sagen, dass der drahtlose Datenaustausch in der Umsetzung technisch aufwendiger ist. Die drahtlosen Datenübertragung ist wesentlich störanfälliger, da jede elektromagnetische Störung das übertragene Signal beeinflussen kann. Durch die fehlende Abgrenzung kann theoretisch jeder, der sich in die Reichweite eines drahtlosen Datenübertragungssystems befindet, die Daten abhören oder manipulieren[Rec04].

Spezifikation des WLAN-Standards:

Für das Wort drahtlose LANs hat sich der englische Begriff Wireless LANs (WLANs) etabiliert, was auch im Rest des Textes verwendet wird. Dieser Unterabschnitt definiert die Spezifikation des WLAN-Standards nach IEEE 802.11. Die Spezifikation startete im Jahr 1997 und sah Datenraten bis 2 MBit/s vor. Dann wurden im Jahr 1999 die Erweiterungen 802.11a(max. 54 MBit/s) und 802.11b(max. 11 MBit/s) vorgestellt. Zur Erweiterungen wurden Arbeitsgruppen gebildet, die weitere Standards spezifizierten. Die zur Zeit gängigsten Standards sind 802.11b (2 bis 11 MBit/s; 2,4 GHz) und 802.11g (11 bis 54 MBit/s; 2,4 GHz).

Ein WLAN nach 802.11 kann in zwei verschiedenen Modi betrieben werden.

Infrastruktur-Modus Die Anbindung der Stationen erfolgt über feste Access Points. Dies sind Rechner, die sowohl über eine drahtlose Anbindung, als auch eine drahtgebundene Anbindung verfügen.

Ad-hoc-Modus In diesem Modus werden die Rechner nur untereinander verbunden, es wird keine Anbindung an ein festes Netz vorgenommen.

Der WLAN Standard 802.11 ist einer von vielen IEEE Netzwerkstandards. Die bekannte Einteilung in die Schichten Logical Link Control (LLC), Media Access Control (MAC) und Physical Layer (PHY) wurde in den Standard 802.11b übernommen. Beim Standard 802.11b wird der PHY in zwei Teilschichten unterteilt, das Physical Layer Convergence Protocol (PLCP), welches einen einheitlichen Zugriff auf die niedrigen Schichten bietet, und das Physical Medium Dependent (PDM), welches verschiedene Verfahren zur Bitübertragung definiert[Rot02].

Die LLC-Schicht wird vom IEEE definiert und ist für alle IEEE-Standards verbindlich. Somit kann die dritte Schicht des OSI-Modells, die Vermittlungsschicht, immer dieselbe Schnittstelle benutzen. Es macht dann keinen Unterschied, ob in den unteren Schichten WLAN, Ethernet etc. verwendet wird. Im Folgenden soll auf den Aufbau der Bitübertragungsschicht (PHY) und des MAC-Teilschichtprotokolls näher eingegangen werden.

Bitübertragungsschicht

Die größten Unterschiede zu den drahtgebundenen Netzwerken liegen in der Bitübertragungsschicht. Da die Übertragungen durch die Luftschnittstelle erfolgt, können Fehler entstehen. Fehler können z.B. durch

- Rauschen und Interferenzen
- Kollidieren der Funksignalen verschiedener Stationen
- Funksignale von Netzwerken mit gleiche Frequenz wie WLAN (z.B. Bluetooth,. . .)

entstehen.

Bei der drahtlosen Übertragung werden die Daten vom Sendergerät durch eine Modulation in einen Frequenzbereich verschoben und abgestrahlt. Das Empfängergerät kann dann durch eine Demodulation die Daten wieder in den ursprünglichen Bereich verschieben. Dieses geschieht über die PDM-Teilschicht. Für Funkübertragungen stehen drei Verfahren zur Verfügung. Frequency Hopping Spread Spectrum (FHSS), Direct Sequence Spread Sprectrum (DSSS) und Orthogonal Frequency Division Multiplexing (OFDM). Es steht noch eine optische Lösung zur Verfügung, die IEEE-Infrarot-Technologie, welche sich allerdings nicht durchsetzen konnte.

MAC-Teilschichtprotokoll:

Die Hauptaufgabe des MAC-Teilschichtprotokolls ist es, den Zugriff auf das Funkmedium zu regeln. Für den Zugriff definiert IEEE 802.11 drei verschiedene Verfahren:

- einfaches Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA)
- CSMA/CA mit Read To Send/Clear To Send (RTS/CTS)
- Point Coordination Function (PCF)

Nur CSMA/CA ist für eine konkrete WLAN-Realisierung verbindlich, die anderen sind optional. Die ersten beiden Verfahren werden unter dem Begriff Distributed Coordination Function (DCF) zusammengefasst. Dieses Verfahren kommt ohne zentrale Koordination aus. Das PCF-Verfahren ist nur im Infrastruktur-Modus verfügbar und erfordert die Koordination durch einen Access-Point.

Da zwei oder mehrere Stationen gleichzeitig Daten senden können, kann es zu Kollisionen kommen. Prinzipiell gibt es zwei Möglichkeiten mit Kollisionen umzugehen:

- Man versucht Kollisionen zu verhindern.
- Man integriert Mechanismen, um Kollisionen zu entdecken und später zu behandeln.

Bei dem CSMA/CA-Protokoll hört die sendewillige Station den Kanal ab. Ist dieser im Leerlauf, beginnt sie mit der Übertragung. Sonst wird das Ende der Übertragung abgewartet. Im Falle einer Kollisionen warten die Stationen und es wird später ein erneuerter Sendeversuch unternommen.

Es kann sein, dass alle Stationen nicht im gegenseitigen Funkbereich liegen. In diesem Fall können Übertragungen an anderen Orten nicht empfangen werden (Hidden Station Problem). Es kann aber auch sein, dass eine sendewillige Station eine Übertragung entdeckt und deshalb das Ende der Übertragung abwartet, obwohl die Zielstation empfangsbereit ist (Exposed Station Problem).

Um die beiden Probleme zu umgehen, kann man das Protokoll CSMA/CA mit RTS/CTS verwenden. Angenommen Station A möchte Daten an Station B senden. Dafür sendet A einen RTS-Rahmen an B. Antwortet B nun mit einem CTS-Rahmen, sendet A den Datenrahmen und startet einen Acknowledgement (ACK)-Timer. Nach dem Empfang des Datenrahmens antwortet B mit einem ACK-Rahmen. Wenn der ACK-Timer von A abläuft, bevor der ACK-Rahmen empfangen wird, muss das gesamte Protokoll erneut ausgeführt werden. Dabei wird während der Übertragung der Kanal durch Network Allocation Vector (NAV)-Signale für die anderen Stationen als belegt markiert.

Bei der PCF fragt der Point Coordinator (PC) die anderen Stationen, ob sie Daten senden möchten. Die Funktion des PC wird üblicherweise von einem Access Point übernommen. Da die Übertragungsreihenfolge vollständig durch den Access Point kontrolliert wird, treten hier keine Kollisionen auf.

Neben dieser Hauptfunktion bietet die MAC-Schicht noch weitere Funktionen wie Uhrensynchronisation, Power Management, Roaming und Sicherheit an.

Dienste

Der Standard 802.11 besagt, dass ein WLAN neun Dienste bereitstellen muss[Tan03]. Diese werden in zwei Kategorien unterteilt, fünf Verteilungsdienste und vier Stationsdienste. Die Verteilungsdienste sind Assoziation, Trennung, Erneute Verbindung, Verteilung, Integration und werden von der Basisstation bereitgestellt. Die Stationsdienste wurden als Authentifizierung, Aufhebung der Authentifizierung, Datenschutz und Datenzustellung festgelegt und sind in einer Zelle aktiv.

Mit “Assoziation” bauen Stationen eine Verbindung zur Basisstation auf und mit “Trennung” wird diese Verbindung wieder getrennt. Wandert eine Station zur einer anderen Zelle, sorgt der Dienst “erneute Verbindung” dafür, daß die Verbindung erhalten bleibt. Wie die Rahmen zur Basisstation weitergeleitet

werden, legt der Dienst “Verteilung” fest. Der “Integrationsdienst” ist für die Umwandlung des Rahmenformats zuständig.

Da bei drahtloser Kommunikation Daten von nicht berechtigten Stationen gesendet oder empfangen werden können, muss jede Station sich authentifizieren. Dies ist Aufgabe des Dienstes “Authentifizierung”. Wenn eine authentifizierte Station das Netz verlassen möchte, wird die Authentifizierung wieder aufgehoben. Damit die über ein WLAN gesendete Daten vertraulich bleiben, müssen sie verschlüsselt sein. Dieses wird durch den Dienst “Datenschutz” gewährleistet. Letztendlich geht es ja um die Datenübertragung, so dass 802.11 auch eine Möglichkeit bereitstellt, Daten zu übertragen und zu empfangen.

Sicherheitskonzepte von IEEE 802.11:

WLAN nach IEEE 802.11 bietet drei Sicherheitskonzepte an[Rot02].

- Man hinterlegt Zugriffslisten bei einem Access Point. Dieser akzeptiert daraufhin nur Pakete von Sendern, die in der Zugangsliste eingetragen sind.
- Man hinterlegt ein gemeinsames Kennwort bei allen Access Points und Stationen. Eine Station kann danach jede Station authentifizieren.
- Die Stationen und Access Points verschlüsseln alle Pakete mit Hilfe des gemeinsamen Kennworts.

Das Sicherheitskonzept des WLAN-Standards ist das Wired Equivalent Privacy(WEP). Wie der Name schon sagt, soll durch den Einsatz von WEP ein ähnliches Sicherheitsniveau wie bei den drahtgebundenen Netzen bereitgestellt werden.

Wegen der kurzen Schlüssellänge (40 Bit bzw. 104 Bit) des Algorithmus wurde WEP in der Vergangenheit mehrfach kritisiert. Daraufhin hat die IEEE-802.11-Arbeitsgruppe ein neues Sicherheitsverfahren, genannt WEP2, entwickelt. Auch die Wi-Fi-Alliance hat ein neues Sicherheitsverfahren unter der Bezeichnung Wi-Fi Protected Access(WPA) angekündigt. Mit WPA wird eine verbesserte Datenverschlüsselung realisiert und es ist abwärtskompatibel zum WEP-Verfahren. WPA ist aber nur dann sinnvoll einsetzbar, wenn alle WLAN-Komponenten auch tatsächlich WPA unterstützen. Sobald sich eine WLAN-Komponente in einer WPA-Funkzelle befindet, die nur das WEP-Verfahren unterstüzt, müssen alle WLAN-Komponenten wieder mit dem WEP-Verfahren arbeiten.

Immer mehr an Bedeutung gewinnen Virtual-Private-Network(VPN)-Lösungen im WLAN-Bereich. VPN wird benutzt, um sichere Ende-zu-Ende-Verbindungen bereitzustellen. Als VPN-Lösungen kommen das Layer-2-Tunneling-Protocol (L2TP), das Point-to-Point-Tunneling-Protocol(PPTP) oder das IP-Security-Protocol (IPSec) in Frage. Mittlerweile sind L2TP und PPTP als unsicher verrufen. Bei IPSec ist das Verschlüsselungsverfahren entscheidend. Für eine sichere und effiziente Datenverschlüsselung innerhalb des WLANs sollte z.B. das Verfahren „Advanced Encryption Standard“ (AES) eingesetzt werden.

2.3. Maschinelles Lernen

2.3.1. Hierarchische Klassifikation

Bei der Ausarbeitung wurde erst Klassifikationsarten und unterschiedliche Klassifikationsverfahren erklärt, die bei Nemoz verwendet werden. Dann wurde erklärt, wie genau hierarchische Klassifikation definiert ist und Integration von Hierarchische Klassifikation an Yale System.

Klassifikation ist das Einordnen von Objekten in vorgegebene Klassen und es ist wichtig zu finden, in welcher Klasse ein gegebenes Objekt aufgrund seiner individuellen Merkmalskombination am besten passt.

Klassifikationsarten

Klassifikationsarten sind als eindimensionale und mehrdimensionale Klassifikation unterteilt. Eindimensionale Klassifikation ergibt sich aus einem einzigen Kriterium. Sie lässt sich auf mehrere Weisen darstellen, z.B. als Baum wie in folgendem Schema, wo die Elemente a, b, c klassifiziert werden.

Mehrdimensionale Klassifikation ergibt sich aus mehr als eine Kriterium und besteht aus kreuz Klassifikation und hierarchische Klassifikation. Kreuz Klassifikation ist durch eine Menge von Elementen gegeben, z.B. von Bauklötzen verschiedener Formen und Farben. Man kann sie nach unterschiedlichen Kriterien klassifizieren. Eine Klassifikation ergibt sich, wenn man die Form als Kriterium wählt, eine andere, wenn die Farbe das Kriterium ist. Jede der beiden Klassifikationen ist vollständig und eindimensional. Kombiniert man die beiden zu einer einzigen Klassifikation, wird es eine (vollständige) mehrdimensionale, und zwar eine Kreuzklassifikation. Bei einer Kreuzklassifikation sind alle Klassifikationskriterien auf alle Objekte anwendbar. Sie sind nicht voneinander abhängig. Daher ist die Reihenfolge der Anwendung auch gleichgültig. Wenn die Bedingungen für eine Kreuzklassifikation nicht erfüllt sind, ergibt sich eine hierarchische Klassifikation. Das ist vor allem dann der Fall, wenn die Klassifikationskriterien verschieden allgemein sind. Jedes Klassifikationskriterium ist dann nur innerhalb eine Klasse anwendbar, die durch ein allgemeineres Kriterium vorab gebildet ist. Die Frage ist, wie man hierarchische Klassifikation an Yale integriern kann, damit Yale hierarchische Ebene auch klassifizieren kann.Die Idee ist, in jedem Knoten eine flache Klassifikation zu erzeugen und so zu erweitern wie Baumstruktur.

Klassifikationsverfahren

Es wird jetzt 4 wichtige Klassifikationsverfahren erklärt:

- **Entscheidungsbaum**
  Bei diesem Klassifikationsverfahren erfolgt die Klassifikation eines Objekts mit einem Entscheidungsbaum, indem man von der Wurzel ausgehend die sich an den Knoten befindlichen Attribute prüft und je nach vorliegender Ausprägung den entsprechenden Verzweigungen folgt.Das Klassifikationsergebnis steht fest, sobald man an einem Blattknoten angelangt ist: die Beschriftung des Blattknotens gibt dann die Klasse an, in die das Objekt einzuordnen ist. Die Konstruktion von Entscheidungsbäumen erfolgt in einem rekursiven Divide- and Conquer Verfahren anhand der Trainingsdaten: In jedem Knoten wird mit einer informationstheoretischen Kennzahl entschieden, anhand welches Attributes die nächste Verzweigung geschehen soll. Für jede vorkommende Ausprägung dieses Attributs wird eine Verzweigung gebildet und der Algorithmus mit denjenigen Trainingsobjekten rekursiv weitergeführt, die diese Ausprägung besitzen [Gra03]. Wenn schon alle Attribute zum Test verwendet wurden, geschieht die Beschriftung mit der in der Teilmenge häufigsten Klasse. Entscheidungsbäume können sehr einfach in leicht interpretierbare Wenn- Dann-Regeln konvertiert werden. Aber bei den meisten Verfahren müssen die Trainingsdaten komplett im Hauptspeicher gehalten werden.

- **Bayes Klassifikation**
  Bei der Bayes-Klassifikation wird ein Objekt derjenigen Klasse zuordnet, die für seine inviduelle Merkmalskombination am wahrscheinlichsten ist. Bayes Klassifikation erzielt bei Anwendung auf großen Datenmengen eine hohe Genauigkeit und eine vergleichbare Geschwindigkeit wie Entscheidungbaum. Aber wenn die Annahmen über Verteilungen und die Unabhängigkeit der Attribute ungerechtfertigt sind, werden die Ergebnisse ungenau

- **Support Vector Machine**
  Die Support Vector Machine ist ein mathematisches Modell, welche eine Menge von Daten, mit Hilfe einer Hyperebene separieren kann.Die Maschine soll die Daten in 2 Klassen einteilen. Die

Hyperebene wird definiert durch einen orthogonalen Vektor w und eine Verschiebung von Ursprung b.Dieser Verfahren kommen auch mit hohen Dimensionen klar und Klassifikator basiert auf wenigen Support Vektoren. Dagegen hat diese Klassifikationsverfahren Probleme mit fehlende Datenwerten [Mit97].

- **K nächste Nachbarn Verfahren**
  Die Idee dieses Verfahrens besteht darin, ein Objekt in die gleiche Klasse einzuordnen wie ähnliche Objekte aus der Trainingsmenge. Dazu werden diejenigen k Trainingsobjekte ermittelt, welche die größte Ähnlichkeit mit dem zu klassifizierenden Objekt besitzen. Gemessen wird dies mit einem festzulegenden Ähnlichkeitsmaß [HTF01]. Die Klasse, die unter diesen k Objekten am häufigsten auftritt, wird als Klassifikationsergebnis ausgegeben. Das Verfahren ist grundsätzlich sowohl für metrische als auch für kategorielle Merkmale anwendbar. Das Ähnlichkeits- bzw. Distanzmaß muss nur entsprechend sinnvoll definiert werden. Die Lernphase entfällt praktisch: alle Trainingsdaten werden nur zwischengespeichert und erst ausgewertet, wenn neue Objekte zu klassifizieren sind. Dagegen ist die Klassifikationsphase sehr aufwendig. Für jeden einzelnen Klassifikationsvorgang muss die gesamte Trainingsmenge zur Verfügung stehen und nach ähnlichen Objekten durchgearbeitet werden. Die Anzahl der zu berücksichtigenden Nachbarn k muss von außen festgelegt werden. Für größere Werte von k nimmt der Aufwand noch zu.

- **Multilayer Feed-Forward Neural Network**
  Ein Neuronales Netz besteht aus eine Menge von Knoten welche mit gewichteten Kanten miteinnander verbunden sind. Während der Lernphase werden die Kantengewichtung so abgestimmt das der Lerner sinnvolle Klassifizierung zu der Eingabemenge liefert.

2.3.2. Meta Learning

Einführung

Überwachtes Lernen (engl. *supervised learning*) bezeichnet maschinelles Lernen mit im Vorraus bekannten Zielwerten für bestimmte Eingabewerte. Bezogen auf das Lernen einer Funktion $f^* : I \rightarrow O \in O^I$ ist diese bis auf eine Menge $X = < i, f^*(i) > \subset I \times O$ von Eingabe-Ausgabe-Paaren (Beispielen) unbekannt.

Ein Lernalgorithmus (Lerner) generiert nun auf Basis von X eine Hypothese $h : I \rightarrow O$ für die unbekannte Funktion. Dabei soll er gut generalisieren, d.h. die Differenz (der Fehler) zwischen h und f^* soll auch auf zukünftigen (während des Lernvorgangs unbekannten) Beispielen möglichst gering sein.

$$\text{Minimiere } E = \sum_{i \in I} Prob(i)||f^*(i) - h(i)||$$

Die generierte Hypothese h stammt aus dem Hypothesenraum $H = O^I$. H ist ein unendlich großer Raum von Funktionen. Nur durch vorherige Einschränkung dieses Raums, man bezeichnet diese Einschränkung auch als den *bias* des Lerners, ist es möglich eine sinnvolle Hypothese zu generieren. Das Wissen um den relevanten Teil des Hypothesenraums stammt oft aus *a-priori* Wissen über die Problemdomäne.

Idee

Lernen zu lernen ist das Ziel des Meta-Learning. Ein Lernvorgang wird nicht als singulärer, sondern vielmehr als ein in den Kontext von schon bewältigten Lernaufgaben eingebetteter Vorgang betrachtet (*lifelong learning*).

Vorher erworbene Fähigkeiten und Wissen haben maßgeblichen Einfluß auf den Lernprozeß. So kann beispielsweise ein Mensch, der das Autofahren erlernt, auf Fähigkeiten, wie Sprache und Motorik, und Wissen, durch bereits im Straßenverkehr gemachte Erfahrungen, zurückgreifen.

Ein Lerner bearbeitet einen Strom von *base-level* Lernaufgaben. Sind die bearbeiteten Lernaufgaben verwandt, so kann Wissen über die Problemdomäne gesammelt und zum Lernen immer besserer Hypothesenräume $H_i \in P(O^I)$ (*domain-specific-bias*) genutzt werden. Man bezeichnet dies auch als *meta-level* Lernaufgabe. Die Bearbeitung dieser Lernaufgabe führt zur Generierung besserer Hypothesen als dies bei isolierter Abarbeitung möglich wäre.

Beispiel

Zur Verdeutlichung der Ideen des Meta-Learning soll nun das Verfahren des Task-Clusterings herangezogen werden. Die Idee des Task-Clusterings ist es, daß Transfer von Wissen aus anderen Aufgaben nur dann positive Auswirkungen hat, wenn diese Aufgaben verwandt sind. Der Verwandtschaftsgrad sei definiert als die Steigerung der Performance einer Aufgabe nach Wissentransfer aus einer anderen. Für alle Paare n, m von Aufgaben berechnet man den Performance-Gewinn durch Transfer von Wissen von m nach n. Die resultierende Matrix dient als Eingabe für ein Clusteringverfahren, welches die Aufgaben bezüglich ihres Verwandtschaftsgrads gruppiert. Wissen wird nun nur zwischen Mitgliedern eines Clusters ausgetauscht. Kommt eine neue Aufgabe hinzu, wird das verwandteste Cluster bestimmt und das dort vorhandene Wissen zur Lösung benutzt. Die Verwandtschaft zwischen Lernaufgaben hängt vom verwendeten Lernalgorithmus ab. Die in diesem Beispiel vorgestellte Implementierung benutzt KNN.

Sei f die durch KNN auf Basis der Trainingsdaten zu approximierende Funktion. Dann ist

$$f(x) = \frac{1}{K} \sum_{k=1}^{K} f(y_k)$$

wobei die K nächsten Nachbarn gemäß einer Distanzmetrik $dist(\cdot, \cdot)$ bestimmt werden. Wie KNN generalisiert, hängt stark von der verwendeten Distanzmetrik ab. Wir verwenden hier eine gewichtete Version der euklidischen Distanz:

$$dist_d(x, y) = \sqrt{\sum_i d^{(i)}(x^{(i)} - y^{(i)})^2}$$

d ist ein Vektor mit Gewichten $d^{(i)} \geq 0$ für jede Dimension. Damit lassen sich Streck- und Stauchoperationen durchführen. Für jede Menge von Trainingsdaten gibt es eine optimale Distanz-Metrik $dist_{d^*}$.

Ein optimale Distanz-Metrik minimiert den Fehler auf zukünftigen Daten. Da sich dieser Fehler nicht exakt berechnen lässt, minimieren wir statt dessen den Abstand zwischen Beispielen in gleichen Klassen und maximieren den Abstand zwischen Beispielen verschiedener Klassen.

$$\text{Minimiere } E(d) = \sum_{x,y} \delta_{xy} dist_d(x, y)$$

wobei $\delta_{xy} = 1$, wenn $f(x) = f(y)$ und -1 sonst. Da $E(d)$ monoton in d, kann das Gradientenverfahren zur Bestimmung von $d^* = argmin_d E(d)$ benutzt werden.

Für eine Menge von Aufgaben $A \subset \{1, 2, ...N\}$ ist die optimale Distanz-Metrik entsprechend

$$d_A^* = argmin_d \sum_{n \in A} E_n(d)$$

Man berechnet nun für jede Aufgabe die optimale Distanz-Metrik, d.h. den optimalen Gewichtsvektor. Berechne dazu eine Matrix $C = (c_{n,m})$, wobei $c_{n,m}$ die erwartete *generalization accuracy* für Aufgabe

n ist, wenn m's optimale Distanz-Metrik benutzt wird. Berechne $c_{n,m}$ durch *cross-validation*. Teile dazu die Menge der Beispieldaten X wiederholt in Trainingsmenge und Testmenge und berechne den Durchschnitt über alle Aufteilungen.

Sei $A_1 ... A_T$ eine disjunkte Zerlegung aller Aufgaben in T Cluster.

$$J = \frac{1}{N} \sum_{k=1}^{T} \sum_{n \in A_t} \frac{1}{|A_t|} \sum_{m \in A_t} c_{n,m}$$

J gibt die durchschnittliche, erwartete *generalization accuracy*, die erreicht wird, wenn Aufgabe $n \in A_t$ die optimalen Metriken aus allen anderen Aufgaben in seinem Cluster benutzt.

Die Optimierung von J auf Basis der gegebenen Matrix, ist ein gut erforschtes kombinatorisches Clustering-Problem, für das verschiedene Algorithmen existieren. Für jedes so erstellte Aufgaben-Cluster, berechne eine optimale Distanz-Metrik (wie vorhin erwähnt).

Wenn eine neue Lernaufgabe auftritt, gibt es zwei mögliche Vorgehensweisen. Entweder werden (*nicht iterativ*) alle Aufgaben neu geclustert, was rechenintensiv ist, oder (*iterativ*) die neue Lernaufgabe wird zum verwandtesten Cluster hinzugefügt und man benutzt dann die optimale Metrik dieses Clusters für die neue Aufgabe. Die zweite Methode ist nicht so rechenintensiv, liefert dafür aber auch keine optimale Lösung.

Zum Bestimmen des verwandtesten Clusters zur neuen Lernaufgabe kann *cross-validation*, wie oben beschrieben, nacheinander mit den Gewichtsvektoren aller Cluster und Auswahl des Vektors mit der größten Performance-Steigerung verwendet werden. Alternativ wähle das Cluster, daß das Verhältnis zwischen der Summe der Distanzen von Elementen verschiedene Klassen und der Summe der Distanzen von Elementen gleicher Klassen maximiert.

$$r(t) = (\sum_{x,y:\delta_{x,y}=-1} dist^*_{A_t}(x,y)) \cdot (\sum_{x,y:\delta_{x,y}=1} dist^*_{A_t}(x,y))^{-1}$$

Anwendung

Ein Großteil der Lernaufgaben in NEMOZ sind Klassifikationsaufgaben. Gegeben sei eine Menge von Klassifikationsschemata (Taxonomien) für Musik, die von verschiedenen Benutzern des Systems angelegt und verwaltet werden. Eine unendliche Menge von Features, die aus Musikstücken extrahiert werden können, dient als Basis für Lernverfahren, mit dem Ziel einer möglichst akkuraten automatischen Klassifikation neuer Musik. Eine Standard-Menge von Features reicht zum Lernen dieser verschiedenster Klassifikationsaufgaben nicht aus. Daher müssen zusätzliche Features (speziell für die einzelne Lernaufgabe zugeschnitten) konstruiert werden. Dafür sollte man allerdings **Geduld** mitbringen, doch gerade hier kann die verteilte Architektur von NEMOZ von Nutzen sein.

Alle Aufgaben haben eine Menge von Basis-Features gemeinsam. Jeder Lerner gewichtet diese Basis-Features nach der Relevanz für seine Aufgabe. Hat ein Lerner für seine Aufgabe schon eine Feature-Konstruktion durchgeführt, so wird sein Gewichtsvektor zusammen mit den zusätzlichen Features in einer *case base* gespeichert. Nun wird angenommen, daß man aus der Ähnlichkeit der Basisgewichte schließen kann, daß ähnliche zusätzliche Features relevant sind. Auf dieser Grundlage erhalten Lerner eine Empfehlung in Form einer Menge zusätzlicher Features, wenn sie ihren eigenen Gewichtsvektor an die *case base* schicken.

Ein Lerner, der eine Aufgabe t_i bearbeitet, gewichtet zunächst seine Basis-Features. Dann sucht er die k verwandtesten Aufgaben auf Basis einer Distanz-Metrik d. Man bildet die Vereinigung aller Features, die für diese k Aufgaben in der *case base* enthalten sind und verwendet sie für die Aufgabe t_i. Ist der Performanzgewinn durch die Verwendung der neuen Features hoch genug, so speichert man den Gewichtsvektor von t_i zusammen mit allen zusätzlichen Features in der *case base*.

Sonst werden die neuen Features als Initialisierung für eine klassische Feature-Konstruktion benutzt. Führt dies dann schließlich zu einem ausreichendem Performanzgewinn, so speichert man ebenfalls,

allerdings nur die lokal generierten Features. In dem hier vorgestellten Ansatz werden Feature-Gewichte durch eine *SVM* berechnet. Diese Gewichte und die Manhattan-Distanz-Funktion ergeben ein geeignetes Distanzmaß $d : T \times T \rightarrow \mathbb{R}^+$

Die Distanz der Feature-Gewichte alleine ist kein ausreichendes Maß für Ähnlichkeit von Aufgaben. Daher erweitert man das Distanzmaß, sodass es die Ähnlichkeit der konstruierten Features (Distanz der Methodenbäume) verschiedener Aufgaben berücksichtigt. Die Distanz von Methodenbäumen ist durch einen speziellen Ansatz in quadratischer Zeit (in der Anzahl der Features) berechenbar. Die Distanz kann in zwei Phasen bestimmt werden. Zunächst wähle k Aufgaben mit den ähnlichsten Feature-Gewichten aus. Schränke diese Menge dann durch den Vergleich von Methodenbäumen weiter ein.

2.3.3. Distributed Data Mining

Einleitung

Dieser Abschnitt befasst sich mit Distributed Data Mining [FPSS96]. Ich erläutere zunächst die Herausforderung, die verteiltes Data Mining stellt. Ich werde im Folgenden kurz auf das generelle Verfahren des Distributed Data Mining eingehen und dann einige Algorithmen vorstellen, die sich mit Clustering und Association Rule Mining befassen.

Distributed Clustering

Das generelle Problem des Distributed Clustering [JKP04a, JKP04b] besteht darin, dass die Daten verteilt sind und die Kosten alle Daten an einen zentralen Ort zu bringen, zu hoch sind. Man braucht also Verfahren, die die Daten dort bearbeiten, wo sie sich befinden. Außerdem sollten diese Verfahren natürlich so wenig Kommunikationskosten erzeugen, wie möglich. Einen abstrakten Ansatz für ein Distributed Clustering zeigt die Abbildung 2.1. Die Daten liegen in dieser Skizze an verschiedenen Orten (Site 1, ..., Site n). Es werden nun auf lokaler Ebene Repräsentanten für die einzelnen Cluster bestimmt. Im nächsten Schritt werden diese lokalen Repräsentanten zusammen gebracht und man bestimmt daraus globale Repräsentanten.

Dieses Vorgehen ist leider nicht ganz fehlerfrei. Die Abbildung 2.2 zeigt ein häufiges Problem des verteilten Clusterns. Während lokal die Datenpunkten in der oberen linken Ecke kein eigenes Cluster erhalten, zeigt sich bei einer globalen Betrachtung, dass diese durchaus ein eigenes Cluster bilden und nicht einfach nur, wie lokal angenommen, Ausreißer sind.

Im Folgenden stelle ich einen Algorithmus aus [EMH03] vor, der ein verteiltes Clustern von Dokumenten ermöglicht. Das Szenario ist dabei das Folgende: Man betrachtet ein P2P Netzwerk mit verschiedenen Rechnern. Auf diesen Rechnern sind unterschiedliche Dokumente abgespeichert. Die Kosten alle Daten an einen zentralen Rechner zu übertragen, welcher das Clustering übernimmt, sind zu hoch. Es wird also ein effizienter „Distributed Clustering Algorithm“ benötigt, der die verteilten Dokumente clustert.

Der Algorithmus benötigt zwei weitere Algorithmen, die er beide kombiniert. Zum einen wird das nichthierarchische Clusteringverfahren k-means benötigt:

Dieser Algorithmus erhält als Eingabe: eine Anzahl der Cluster k, eine n dimensionale Menge von Datenobjekten D und eine initiale n-dimensionale Menge von Cluster-Centroids C.

Der Ablauf des Algorithmus ist dann der Folgende:

1. In jeder Iteration wird die Distanz von jedem $d_i \in D$ zu allen $c_j \in C$ berechnet.
2. Jedes d_i wird dem nächsten c_j zugeordnet.
3. Jeder Centroid c_j wird nun, als der neue Centroid von allen Datenobjekten d_i, die ihm zugeordnet wurden, neu berechnet.

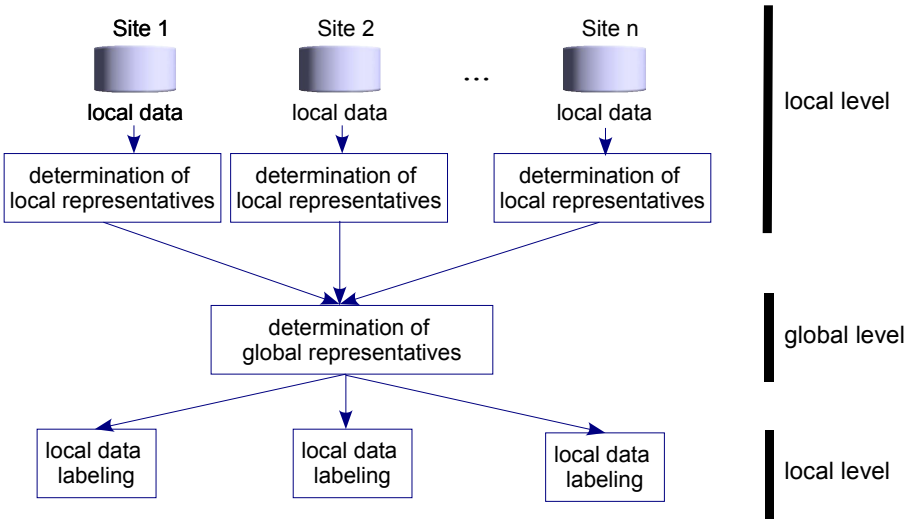


Abbildung 2.1.: Schema: Distributed Clustering aus [JKP04b]

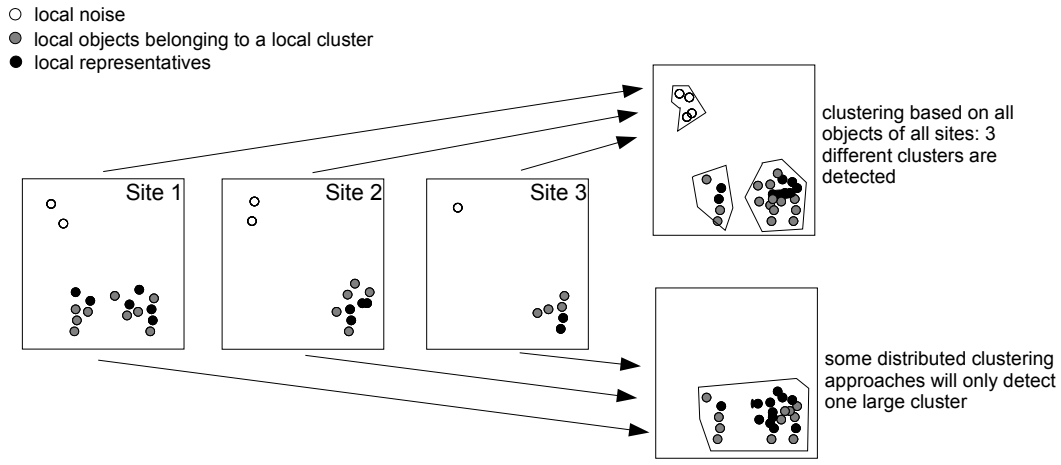


Abbildung 2.2.: Ein Problem des Distributed Clustering aus [JKP04b]

Wenn die Summe von allen Standardabweichungen (MSE) jedes Clusters nicht mehr sinkt, so terminiert der Algorithmus. Die einzelnen Knoten müssen also nur die Cluster-Centroids austauschen. Die einzelnen lokalen Verfeinerungen der Centroids können später bei dem Initialknoten verfeint werden. Somit wird das Ziel der geringen Kosten erreicht.

Außerdem benötigt der Algorithmus den Probe/Echo Mechanismus. Dies ist eine Algorithmus, der einen impliziten Spannbaum eines gegebenen Graphens berechnet. Der Ablauf sieht dabei folgendermaßen aus:

- Der Initiator markiert sich als `engaged` und `initiator`. Dann sendet er eine `PROBE` Nachricht an alle seine direkten Nachbarn.
- Wenn ein Knoten das erste Mal eine `PROBE` Nachricht empfängt, markiert er sich selbst als `engaged` und sendet selbst `PROBE` Nachrichten an alle angrenzenden Knoten.
- Wenn ein Knoten von allen Nachbarn `PROBE` oder `ECHO` Nachrichten empfangen hat, markiert er sich selbst als `not engaged` und sendet ein `ECHO` an den Knoten, von dem er die erste `PROBE` empfangen hat.
- Empfängt der Initiator von allen Nachbarn ein `ECHO`, terminiert der Algorithmus.

Eine nützliche Eigenschaft dieses Algorithmus ist, dass jeder Knoten eine Nachricht nur genau einmal empfängt. Genau diese Eigenschaft wird bei der Kombination der beiden Algorithmen wichtig.

Nun werden also die beiden oben vorgestellten Algorithmen kombiniert, was zu folgendem Ablauf führt:

1. Der Initiator schätzt eine initiale Menge von Cluster-Centroids und sendet dies mit einer `PROBE` an alle Nachbarknoten.
2. Jeder Nachbar, der eine `PROBE` erhält, sendet diese weiter zu seinen Nachbarn.
3. Nun macht er ein k-means Clustering auf den lokal verfügbaren Daten.
4. Wenn er ein `ECHO` empfängt, vereinigt er das Clusterergebnis des Nachbarn mit seinem eigenen Ergebnis.
5. Wenn der Knoten von jedem seiner Nachbarn eine `PROBE` oder ein `ECHO` erhalten hat, sendet er ein `ECHO` mit den vereinigten Cluster-Centroids und der Gewichtung an den Knoten, von dem er die erste `PROBE` erhalten hat.
6. Wenn der Initiator von allen angrenzenden Knoten ein `ECHO` erhalten hat, hat er alle Informationen über die derzeitige Iteration des k-means Algorithmus. Diese Iteration ist damit beendet.
7. Nun entscheidet der Initiator, ob eine weitere Iteration notwendig ist. Wenn dem so ist, sendet er die gerade erstellten Cluster-Centroids als neue Centroids an seine Nachbarn.

In der Regel benötigt man mehrere Iterationen, um ein gutes Ergebnis zu erhalten.

In einigen Experimenten mit synthetischen und realen Daten zeigte sich, dass dieses Vorgehen das Clustering, im Vergleich zu zentralen Clustering, beschleunigte. Man muss allerdings darauf achten, dass man die Verbindungsmöglichkeiten der einzelnen Peers beschränkt. Ab einem bestimmten Schwellwert ist der Overhead der Kommunikation zu groß, wodurch das Clustering langsamer wird. Zusammenfassend kann man sagen, dass dieser relativ einfache Algorithmus gute Ergebnisse liefert und Zeit spart.

Association Rule Mining

Das Association Rule Mining versucht automatische Regelmengen zu generieren, die auf bestimmten Abhängigkeiten der Attribute basieren. Der große Nachteil bei diesem Verfahren ist die große Vielfalt der Regeln. Aus dieser großen Menge gerade die Regeln zu bestimmen, die wirklich „interessant“ sind, ist schwierig. Die Regelmenge muss also noch analysiert werden, um überflüssige Regeln auszufiltern und die wirklich interessanten zu finden.

Auch für dieses Verfahren wurden verteilte Algorithmen entworfen, von denen ich drei kurz vorstellen werde.

Count Distribution Dies Verfahren baut auf dem Apriori-Algorithmus auf. Man tauscht dabei nicht die Datentupel selbst aus, sondern nur die Anzahlen. Der Ablauf sieht dabei wie folgt aus: In einem ersten Scan ermittelt jeder Prozessor lokale Kandidaten, abhängig von den in seiner Partition vertretenen Itemsets. Dann wird auf Basis der lokalen Daten, die jeder Prozessor ermittelt hat, eine globale Berechnung durchgeführt. Abbildung 2.3 zeigt den schematischen Aufbau des Verfahrens.

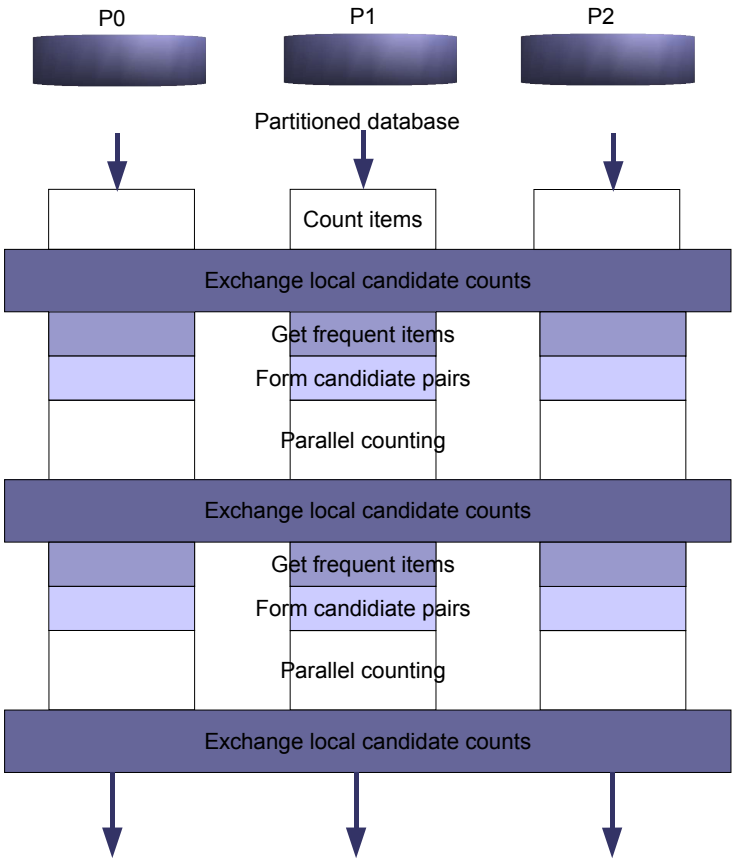


Abbildung 2.3.: Schema: Count Distribution aus [Zak99]

Fast Distributed Mining Dieses Verfahren findet ebenfalls Regeln in, auf mehrere Teilnehmer verteilten, Datenmengen. FDM sucht dazu die lokalen Supports und schließt alle seltenen, lokalen Items aus. Nach dieser lokalen Analyse werden Broadcast Nachrichten mit allen verbleibenden Itemset verschickt, um dort ihren Support zu erfragen. Daraus werden dann globale Itemsets bestimmt.

ODAM: An Optimized Distributed Association Rule Mining Algorithmen Ein weiteres Verfahren ist das ODAM. Es wurde in [ATS04] vorgestellt. Es werden hierbei zunächst die „1 itemsets“ wie im sequentiellen Apriori-Verfahren bestimmt. Danach sendet man diese Itemsets an einen anderen Rechner. Dieser berechnet nun die globalen „1 itemsets“. In der nächsten Iteration werden nun die „2 itemsets“ bestimmt und es wird wiederum deren Support berechnet. Alle global selten auftretenden „1 itemsets“ werden entfernt und die häufig auftretenden „1 itemsets“ werden in den Speicher geschrieben. Nun werden alle globalen „2 itemsets“ berechnet. So iteriert man im weiteren Verlauf durch den Speicher und generiert Support für die Kandidaten entsprechender Länge.

Dies Verfahren hat sich in Tests als sehr effizient erwiesen, wie man in [ATS04] nachlesen kann.

2.3.4. Ordnen von Suchergebnissen

Bei einer normalen Suche sendet der Client die Suchanfrage an eine Datenbank, die dann ein Suchergebnis zurückliefert. Die Qualität und die Anzahl der Suchergebnisse ist dabei sehr stark von der Qualität der Datenbank abhängig. Um die Suchergebnisse zu verbessern, kann man die Anfrage aber auch an mehrere Datenbanken schicken. Dabei wird ein Agent zwischengeschaltet, der die Useranfrage annimmt. Diese Anfrage wird dann in ein Format gewandelt, das die jeweilige Suchmaschine versteht. Der Agent sendet die Anfragen an einzelne Datenbanken und wartet dann auf die Antworten, fügt sie zusammen, und präsentiert dann das Ergebnis dem User.

Das Problem, vor dem man steht, ist das “richtige” Ordnen der Suchergebnisse. Die Antworten der verschiedenen Datenbanken können sehr unterschiedlich sein. Manche schicken eine Bewertung der Ergebnisse mit. Andere nur eine Ordnung. Allgemein lässt sich das Problem so fassen: [CHT99]

Eine Dokumentordnung $R = (D, o)$ besteht aus einer Menge von Dokumenten D und einer Ordnung o auf diesen Dokumenten. Es sind N Ordnungen $R_1 ... R_n$ gegeben. Finde eine gemeinsame Ordnung $R_m = (D_m, o_m)$ so dass $D_m = D_1 \bigcup ... \bigcup D_n$ ist eine effektive Ordnung, die relevante Dokumente uber den nicht relevanten einordnet.

Auf welche Weise kann das geschehen? Es gibt zwei unterschiedliche Ansätze[ML99]. Der Eine setzt auf die Kommunikation mit dem Server (Integrierte Methode), der Andere arbeitet lokal auf den geschickten Daten (Isolierte Methode).

Bei der integrierten Methode kann man bei jeder Suchanfrage seine Präferenzen an den Suchserver schicken. Allerdings erfordert es einen größeren Overhead an Bandbreite. Dazu ist ein spezielles Protokoll für die Kommunikation mit dem Suchserver erforderlich. Das Ganze umgeht man, indem man seine Präferenzen auf dem Server speichert. Vor der Suche muss man sich zuerst identifizieren, z.B. durch Username/Passwort oder durch Cookies. Der Server speichert dann die Vorlieben des Benutzers und kann die Suchergebnisse entsprechend gestalten. Leider ist damit auch die Möglichkeit gegeben, sehr leicht ein Nutzerprofil zu erstellen. Außerdem hat man keine Kontrolle mehr über die persönlichen Daten, die auf dem Server gespeichert werden.

Wenn man seine Daten nicht preisgeben will, muss man also die Bewertung der Suchergebnisse für jede einzelne Suchmaschine lokal vornehmen. Damit erhöht sich zwar der Aufwand auf der Clientseite, aber man hat die komplette Kontrolle über die Ordnung der Ergebnisse.

Welche Daten stellt denn die Suchmaschine überhaupt zur Verfügung? Auf jeden Fall die Verweise auf die Suchergebnisse (und damit auch die Suchergebnisse selber) und die Rangordnung der Suchergebnisse. In manchen Fällen wird auch die Güte der einzelnen Ergebnisse mitgeliefert.

Die einfachste Methode die Suchergebnisse zu verbinden, ist die Sortierung nach dem lokalen Rang, den jede Suchmaschine diesen Ergebnissen gegeben hat. Dabei wird nach dem “Round Robin” Prinzip vorgegangen. Man nimmt sich also erst alle Suchergebnisse mit Rang 1, danach mit Rang 2 usw... So sind die Ergebnisse, die eine Suchmaschine für gut gefunden hat, auch weiter oben in der finalen Ergebnisliste.

Um diese Methode zu verbessern, kann man jeder Datenbank einen Vertrauenswert zuordnen, und die

Suchergebnisse entsprechend sortieren. Dabei wird die Rangnummer des Suchergebnisses mit dem Vertrauenswert multipliziert, sodass die Ergebnisse von der Datenbank mit dem größten Vertrauen weiter oben stehen. Hierbei stellt sich aber das Problem, dass man zuerst für jede Datenbank den Vertrauenswert erst mal haben muss. Und da sich die Güte einer Suchmaschine jederzeit ändern kann, muss dieser Wert entsprechend auch gepflegt werden.

Wenn man aber jeder Datenbank eine Wert zuordnet, so kann man auch die Gangordnung der Ergebnisse in eine Ergebnisgüte verwandeln. Dabei kann man sich der folgenden Formel bedienen:

$g = 1 - (r - 1) * F_i$ wobei:

$F_i = (r_{min})/(m * r_i)$ ist.

g ist dabei die Endpunktzahl für das Ergebnis. r ist die Rangnummer des Dokuments in der Datenbank. m ist die Anzahl der Ergebnisse, und r_i schließlich ist der Vertrauenswert der Datenbank i. Wie man sieht, nimmt die Güte der Ergebnisse mit steigendem F_i ab.

Manche Suchmaschinen erleichtern einem aber auch die Arbeit, indem sie eine Güte für die Suchergebnisse mitschicken. Somit kann man die Daten sofort sortieren. Man muss allerdings beachten, dass das Bewertungsformat nicht kompatibel sein muss. Manche liefern zum Beispiel eine Zahl von 0 bis 1. Andere wiederum einen Prozentwert. Deshalb muss man die Werte vor der Sortierung so weit wie möglich normalisieren.

Da man die Links auf die Ergebnisse hat, kann man die Ergebnisse auch selber herunterladen und noch einmal lokal bewerten. Somit kann man sofort die kaputten Links aussortieren, und kann eine auf den aktuellen Nutzer angepasste Sortierung vornehmen. Ein Ansatz dabei ist der Feature Ranking Distance Algorithmus. Dabei nimmt man an, dass ein Feature weniger wichtig ist, wenn es erst am Ende des Dokumentes vorkommt, nicht in der Nähe anderer Features ist oder sehr oft in einem Dokument vorhanden ist. Aus diesen Annahmen leitet man eine Formel ab, die dann den Gesamtwert des Dokuments berechnet.

Alle bisherigen vorgestellten Verfahren funktionieren statisch. Ihre Fähigkeiten Ergebnisse zu sortieren verbessern sich nicht mit Laufe der Zeit. Es gibt aber auch Verfahren, die sich dem User anpassen[CSY99]. Sie beobachten die Reaktionen des Users auf eine Sortierung und versuchen diese beim nächsten Mal zu verbessern.

Dazu sucht man eine Präferenzfunktion, die für jedes Ergebnispaar die Reihenfolge bestimmt. Durch paarweises Vergleichen von allen Daten, kann dann schließlich die komplette Ordnung hergestellt werden. Dazu braucht man erst elementare Vergleichsoperationen, die jeweils ein bestimmtes Merkmal der Daten vergleichen und den Wert 1 (a soll vor b eingeordnet werden) oder 0 (b soll vor a eingeordnet werden) annehmen können. Durch eine Linearkombination kann schließlich die Präferenzfunktion gebildet werden. Danach braucht man Feedback vom Benutzer, um die Koeffizienten der Gleichung optimal anzupassen. Dies kann durch explizite Anfragen geschehen, oder durch die Auswertung des Nutzerverhaltens (Welches Suchergebnis wurde zuerst angeschaut, oder wie lange hat man sich mit dem einzelnen Ergebnis beschäftigt). Das Feedback hat dann die Form „a sollte b vorgezogen werden“. Alle elementaren Vergleiche, die diesen Aussagen zustimmen, erhalten einen höheren Koeffizienten. Alle, die dagegen sind, erhalten einen niedrigeren. Danach wird die Präferenzfunktion mit den neuen Gewichten neu berechnet. Dabei müssen die Gewichte normalisiert werden, um insgesamt eine 1 zu ergeben.

Nachdem man die Präferenzfunktion hat, kann man die Ergebnisse sortieren. Leider ist dieses Problem NP-vollständig. Es gibt aber einen Greedy-Algorithmus, der eine gute Approximation liefert. Dabei betrachtet man das Ganze als einen gerichteten Graphen, wo jeder Knoten ein Suchergebnis ist und mit zwei Kanten zu jedem anderen Knoten verbunden ist. Jeder Kante ist ein Wert zwischen 0 und 1 zugewiesen. Je höher die Summe der Werte der ausgehenden Kanten, desto weiter oben wird dieses Ergebnis einsortiert. Um die Ergebnisse jetzt zu ordnen, sucht man den Knoten mit dem höchsten Wert und nimmt diesen aus dem Graphen (zusammen mit seinen Kanten) heraus. Danach werden die Gewichte der einzelnen Knoten neu berechnet, und wieder der mit dem höchsten Wert rausgenommen. Das macht man so weiter, bis keine Knoten im Graphen mehr vorhanden sind.

Um die Laufzeit zu verbessern, kann dieses Verfahren mit dem topologischen Sortieren verbunden werden. Dabei werden die Kanten von a nach b und b nach a durch eine Kante mit dem Betrag der Differenz der beiden früheren Kanten ersetzt. Zwischen gleichwertigen Knoten gibt es dann keine Kanten. Danach unterteilt man den Graphen in starke Zusammenhangskomponenten. Eine Zusammenhangskomponente mit ausgehenden Kanten wird dabei über der mit eingehenden Kantenverbindungen eingeordnet. Auf die einzelnen Komponenten wendet man dann den Greedy-Algorithmus an.

Zwar kann dieses Verfahren schnell sortieren, doch die Erstellung des Graphen ist sehr aufwendig. Da der Graph fast vollständig ist, sind es ca. 9900 Kanten bei 100 Suchergebnissen (Jeder Knoten ist mit 99 anderen Knoten verbunden). Bei der Gewichtung der Kanten müssen die elementare Vergleiche berechnet werden. Bei 50 elementaren Merkmalen sind es dann insgesamt ca. 500000 Operationen, die ausgeführt werden müssen, bevor man den Greedy-Algorithmus überhaupt anwenden kann.

2.3.5. Ontology Matching mit Hilfe des Maschinellen Lernens

In diesem Abschnitt betrachten wir die Suche nach einer semantischen Abbildung zwischen zwei gegebenen Ontologien. In der Fachsprache wird dieses Problem auch Ontology Matching genannt. Dieses Problem ist eines der Kernprobleme von vielen, Information verarbeitenden, Applikationen. Jede Applikation, die mehrere Ontologien betrachtet bzw. mit mehreren Ontologien arbeitet, muss eine semantische Abbildung zwischen diesen Ontologien herstellen.

Die soeben genannten Applikationen werden häufig in Bereichen, wie E-Commerce, Knowledge Management, Bioinformatik oder Tourismus angewandt.

Ontology Matching wird heutzutage immer noch per Hand durchgeführt, was sehr arbeitsintensiv und fehleranfällig ist.

Das von AnHai Doan und Pedro Domingos entworfene GLUE System[DMDH04], das eine semiautomatische semantische Abbildung zwischen Ontologien mit Hilfe des Maschinellen Lernens generiert, werden wir in den nächsten Unterabschnitten näher betrachten. Zuerst wollen wir mit einem motivierenden Beispiel beginnen.

Ein motivierendes Beispiel: Das Semantische Web

Die Anzahl der Seiten im WWW beläuft sich laut Google auf über 8 Milliarden. Die meisten Seiten sind nur in menschenlesbarer Form. Deswegen können Softwareagenten diese Information nicht verstehen bzw. verarbeiten. Dies führt dazu, dass ein großes Potential an Information im WWW unbetrachtet bleibt.

Um diese Information in maschinenlesbare Form zu bringen, wurde das semantische Netz entwickelt, in dem Daten eine Struktur haben und Ontologien die Semantik der Daten beschreiben. Das folgende Beispiel beschreibt die dahinterstehende Idee.

Stellt Euch vor, Ihr möchtet mehr über eine Person erfahren, die Ihr auf einer Konferenz kennengelernt habt. Folgende Information über diese Person sind bekannt.

- Diese Person heißt Cook,
- Er lehrt Informatik an einer Universität in den USA (Die genaue Universität ist unbekannt)
- Vorher hat er als Associate Prof. an einer australischen Universität gearbeitet.

Im Internet hättet ihr wohl sicherlich Schwierigkeiten diese Person zu finden. Die Information, die Ihr über diese Person gesammelt habt, steht sicherlich nicht auf einer einzigen Webseite. Eine Keyword-Suche erweisst sich als ineffektiv.

Ein Directory-Service würde die Arbeit eures Softbots, Informatik-Fakultäten in den USA zu finden, erheblich erleichtern. Solche Daten benutzen Ontologien, wie auf Abbildung 2.4 zu sehen.

Abbildung 2.4.: Ontologien des Fachbereich Informatik

Wie in Abbildung 2.4 zu sehen ist, werden die Daten in Taxonomien eingeordnet. Diese Taxonomien beinhalten Kurse, Professoren und sonstige Personen. Professoren haben Attribute, wie z.B. Name oder Abschluss. Solche markierte Datensätze vereinfachen die Suche nach einem Prof. Cook für euren Softbot erheblich.

Nachdem der Softbot Prof. Cook gefunden hat, findet er das Attribut *granting institution*, welche zu einer australischen Universität verlinkt. Der Softbot merkt sofort, daß die australische Universität einen ähnlich markierten Datensatz zur Verfügung stellt (siehe Abbildung 2.4). Das Problem ist aber, dass es mehrere Entitäten mit den Namen Cook an dieser Universität gibt. Wenn der Softbot aber in der Lage wäre zu wissen, daß *associate prof* und *senior lecturer* äquivalente Bedeutung haben, würde er in den richtigen Teilbaum verzweigen und die alte Homepage von Prof. Cook hervorheben.

Wie im eben betrachteten Beispiel zu sehen ist, beschreiben diese Ontologien ähnliche Domänen mit verschiedenen Terminologien. Um Daten von verschiedenen Ontologien zu verschmelzen, ist es notwendig die semantische Korrespondenz zu finden.

Ontology Matching

Eine Ontologie spezifiziert eine Konzeptualisierung einer Domain in Terme bzw. Konzeptterme, Attribute und Relationen. Die Konzepte werden typischerweise als Taxonomiebaum dargestellt, sodass jeder Knoten ein Konzept und ein Konzept eine Spezialisierung seines Elternknoten darstellt.

Jedes Konzept beinhaltet eine Menge von Instanzen und eine Menge von Attributen. Desweiterem definiert eine Ontologie auch eine Menge von Relationen zwischen den einzelnen Konzepten.

Das *Ontology Matching* Problem ist das Finden von semantischen Abbildungen zwischen gegebenen Ontologien. Wir werden uns grundsätzlich auf die *eins zu eins* Abbildung konzentrieren. Ein Beispiel zu einer *eins zu eins* Abbildung wäre das Mapping von Senior Lecturer auf Associate Prof. .

Die zentralen Komponenten einer Ontologie sind Taxonomien. Deswegen suchen wir zuerst für jeden Konzeptknoten in einer Taxonomie, den *Ähnlichsten* in der anderen Taxonomie.

Als Framework für unser Ähnlichkeitsmaß benutzen wir die gemeinsame Verteilung. Die gemeinsame Verteilung zwischen den Konzepten A und B ist gegeben durch:

P(A,B), P(A,$\overline{B}$), P($\overline{A}$,B), P($\overline{A}$,$\overline{B}$)

P(A,$\overline{B}$) ist die Wahrscheinlichkeit, dass eine auf dem betrachteten Universum zufällig gewählte Instanz zu A, aber nicht zu B gehört. Als Ähnlichkeitsmaß benutzen wir die Jaccard Similarity.

$$Jaccard - sim(A,B) = \frac{P(A,B)}{P(A,B) + P(A,\overline{B}) + P(\overline{A},B)}$$

Der Wert

$$P(A,B) = \frac{\text{Anzahl der Instanzen die zu A und B gehoeren}}{\text{Anzahl der Instanzen des betrachteten Universum}}$$

lässt sich nicht berechnen, denn das ganze Universum ist nicht bekannt. Daher müssen wir P(A,B) folgendermaßen abschätzen: U_i bezeichnet die Menge der Instanzen für die gegebene Taxonomie O_i. N(U_i) steht für die Anzahl der Instanzen in U_i und $N(U_i^{A,B})$ für die Anzahl der Instanzen in U_i, die zu A und B gehören.

Der Wert P(A,B) lässt sich dann wie folgt abschätzen:

$$P(A,B) = \frac{[N(U_1^{A,B}) + N(U_2^{A,B})]}{[N(U_1) + N(U_2)]}$$

Die Berechnung von P(A,B) reduziert sich dann auf die Berechnung von $N(U_1^{A,B})$ und $N(U_2^{A,B})$. Der Wert $N(U_2^{A,B})$ lässt sich berechnen, wenn wir für jede Instanz s in U_2 wissen, dass sie zu A und B gehört.

Die Entscheidung, ob s zu B gehört, ist trivial (wurde explizit spezifiziert). Die Entscheidung, ob s zu A gehört, greift auf Techniken des Maschinellen Lernens zurück.

Die Menge U_1 wird in 2 Mengen partitioniert, wobei die eine Menge die Instanzen, die zu A und die andere Menge die Instanzen, die nicht zu A gehören, enthalten. Der Klassifier bzw. Lerner lernt diese positiven und negativen Beispiele. Danach wird der Klassifier vorhersagen, ob s zu A gehört oder nicht.

Auf diese Art und Weise ist der Klassifier in der Lage, die Menge U_2^B ($U_2^{\overline{B}}$) in die Mengen $U_2^{A,B}$ ($U_2^{A,\overline{B}}$) und $U_2^{\overline{A},B}$ ($U_2^{\overline{A},\overline{B}}$) zu partitionieren. Mit einem einfachen Rollentausch der Taxonomien O_1 und O_2 lassen sich die restlichen Mengen $U_1^{A,B}$, $U_1^{A,\overline{B}}$, $U_1^{\overline{A},B}$ $U_1^{\overline{A},\overline{B}}$ berechnen. Mit den berechneten Mengen lässt sich dann

$$P(A,B) = \frac{[N(U_1^{A,B}) + N(U_2^{A,B})]}{[N(U_1) + N(U_2)]}$$

berechnen.

Die GLUE-Architektur

Wie in Abbildung 2 zu sehen ist, besteht die GLUE-Architektur aus drei Modulen, einem Schätzer für die Verteilung, einem Schätzer für die Ähnlichkeit und einem Relaxation-Labeler.

Der Verteilungschätzer berechnet für jedes Konzeptpaar die bedingte Wahrscheinlichkeit und gibt sie weiter zum Ähnlichkeitsschätzer, der anhand der *Jaccard similarity* eine Ähnlichkeitsmatrix berechnet.

Die GLUE-Architektur bevorzugt einen `multi strategy learning approach`, der sich aus zwei Base Lerner

- Content Lerner
- Name Lerner

und einen Meta Lerner zusammensetzt.

- Lineare Kombination der Base Lerner

Der Content Lerner benutzt den Naiven Bayes Ansatz zur Textklassifikation. Seine Vorhersagen beruhen auf die Häufigkeit der Wörter im Text.

Der Name Lerner ist ähnlich zum Content Lerner. Seine Vorhersagen beruhen auf den vollen Namen (der komplette Pfad der Instanz im Taxonomiebaum) einer Eingabeinstanz.

Der Meta Lerner kombiniert die Vorhersagen der Base Lerner, indem Vorhersagen mit entsprechenden Gewichten aufsummiert werden.

Relaxation Labeling ist eine effiziente Technik, um Knoten in einem Graphen zu beschriften. Die Grundidee ist, dass die Knotenbeschriftung von den Features der Nachbarknoten abhängt. Typische Features sind:

- Die Labels der Nachbarknoten
- Der Prozentsatz an Nachbarknoten, die bestimmte Kriterien erfüllen.
- Die Tatsache, dass eine bestimmte Bedingung erfüllt ist oder nicht.

Der Einfluss der Nachbarknoten auf die Knotenbeschriftung kann mit einer Wahrscheinlichkeitsformel als Funktion der Nachbarschaftsfeatures quantifiziert werden.

- Jedem Knoten wird eine Beschriftung anhand seiner Eigenschaften zugewiesen.
- Schätze anhand der Formel die Beschriftungen iterativ ab bis sich die Beschriftungen nicht mehr ändern, oder ein anderes Konvergenzkriterium erfüllt ist.

Zusammenfassung

Die GLUE-Architektur arbeitet automatisch an der Integration der Eingabe-Ontologien mit einer erstaunlich hohen Genauigkeit. Allerdings braucht das System eine gewisse Menge von Trainingsinstanzen, um optimal funktionieren zu können. Eine Benutzerinteraktion ist also doch immer noch notwendig, um die Präzision des Systems zu evaluieren.

2.3.6. Effiziente Ähnlichkeitsmaße für adaptive Featuremengen

Multimediadatenbanken haben in den letzten Jahren sehr an Bedeutung gewonnen. Ein wichtiges Merkmal dieser Datenbanken ist die Unterstützung inhaltsbasierter Suchanfragen (content based retrieval). Solche Suchanfragen erfordern eine effiziente Ähnlichkeitssuche (similarity search) als Basisfunktion der Datenbank. Natürlich basiert auch Nemoz auf einer einfachen, selbstentwickelten Multimediadatenbank. Der folgende Abschnitt bietet eine kurze Einführung in die Grundlagen von Distanzmetriken und befasst sich dann mit den Algorithmen und Datenstrukturen, die bei der effizienten Verarbeitung von Ähnlichkeitsanfragen Verwendung finden.

Grundlagen

Im Folgenden bezeichne $Obj = \{obj_1, ..., obj_N\}$ die Menge aller möglichen Objekte und $DB \subset Obj$ eine Multimediadatenbank. Die meisten Multimediadatenbanken lösen das Problem der Ähnlichkeitssuche durch die Extraktion eines (hochdimensionalen) Featurevektors für jedes Objekt in Obj mit Hilfe einer Featuretransformation $T : Obj \rightarrow \mathbb{R}^d$. Die Ähnlichkeit zweier Objekte läßt sich dann sehr einfach durch eine Vektorraum-Distanzmetrik δ bestimmen. Die Mehrheit aller Multimediadatenbanken verwendet derzeit Featurevektoren konstanter Dimension, nicht so jedoch Nemoz.

Featurevektoren in Nemoz In Nemoz werden Audiodaten als Featurevektoren variabler Dimension und Elementen aus $\mathbb{R}$) repräsentiert. Die Menge aller möglichen Features in Nemoz F_{Nemoz} ist abzählbar und geordnet, es gibt also eine Abbildung $\tau : \omega \rightarrow F_{Nemoz}$ (ω ist hier die Menge der natürlichen Zahlen). Daher lassen sich Featurevektoren in der folgenden Weise kompakt darstellen: $(i_1 : v_1, i_2 : v_2, ..., i_n : v_n)$, wobei $\tau(i_i) \in F_{Nemoz}$ und $v_i \in \mathbb{R}$. Diese Darstellungsform bedingt allerdings Featurevektoren variabler Dimension.

Metriken, Ähnlichkeitsmaße und Anfragetypen Eine Metrik ist eine totale Funktion $\delta : Obj \times Obj \rightarrow \mathbb{R}_0^+$, die den folgenden Bedingungen genügt:

1. [Symmetrie] $\delta(obj_1, obj_2) = \delta(obj_2, obj_1)$,
2. [Positivität] $\delta(obj_1, obj_2) > 0$ für $obj_1 \neq obj_2$ und $\delta(obj_1, obj_2) = 0$ für $obj_1 = obj_2$,
3. [Dreiecksungleichung] $\delta(obj_1, obj_2) \leq \delta(obj_1, obj_3) + \delta(obj_3, obj_2)$.

Ein metrischer Raum ist eine algebraische Struktur $\mathcal{M} = (Obj, \delta)$, bestehend aus einer Objektmenge Obj und einer Metrik $\delta : Obj \times Obj \rightarrow \mathbb{R}_0^+$. Ein (allgemeines) Ähnlichkeitsmaß ist eine Metrik der Form $\delta : Obj \times Obj \rightarrow \mathbb{R}_0^+$. Zwei Objekte obj_1 und obj_2 heißen **ε-ähnlich**, gdw.

$$\delta(obj_2, obj_1) < \varepsilon.$$

Für $\varepsilon = 0$ heißen die Objekte identisch. Zwei Objekte obj_1 und obj_2 heißen NN-ähnlich bezüglich einer Datenbank von Objekten DB, gdw.

$$(\forall obj \in DB, obj \neq obj_1)\ \delta(obj_2, obj_1) \leq \delta(obj_2, obj)$$

(wenn also kein Objekt in der Datenbank näher bei obj_2 liegt als obj_1). Diese Ähnlichkeitsbegriffe führen ganz natürlich zu drei Anfragetypen, die eine mögliche Indexstruktur effizient unterstützten sollte: Punktanfragen (*Identität*), Bereichsanfragen (*ε-Ähnlichkeit*) und Anfragen nach den nächsten Nachbarn (*NN-Ähnlichkeit*).

Distanzmetriken für Vektorräume Seien im Folgenden p, q durch eine Featuretransformation gewonnene Punkte im Datenraum. Die folgenden Distanzmetriken werden oft im Kontext von Vektorräumen verwendet:

- Euklidisch (L_2):
$$\delta_{Euklid}(p,q) := \sqrt{\sum_{i=0}^{d-1}(q_i - p_i)^2}$$
- Manhattan (L_1):
$$\delta_{Manhattan}(p,q) := \sum_{i=0}^{d-1}|(q_i - p_i)|$$
- Maximum (L_∞):
$$\delta_{Max}(p,q) := \max\{|(q_i - p_i)|\}$$
- Euklidisch (gewichtet):
$$\delta_{W.Euklid}(p,q) := \sqrt{\sum_{i=0}^{d-1} w_i \cdot (q_i - p_i)^2}$$
- Maximum (gewichtet):
$$\delta_{W.Max}(p,q) := \max\{w_i \cdot |(q_i - p_i)|\}$$
- Ellipsoid (für eine Ähnlichkeitsmatrix W):
$$\delta^2_{Ellipsoid}(p,q) := (p-q)^T \cdot W \cdot (p-q)$$

Der „Fluch der Dimension“ Erhöht man die Dimension des Datenraumes, lassen sich eine Vielzahl interessanter „mathematischer Phänomene“ beobachten, die man in ihrer Gesamtheit als den „Fluch der Dimension“ kennt. Einige dieser Phänomene sind nicht nur quantitativer, sondern auch qualitativer Natur. Aus diesem Grund lassen sich viele dieser (oft wenig intuitiven) Phänomene nicht im „zugänglichen“ $\mathbb{R}^3$ demonstrieren. So wächst das Volumen eines (Hyper-)Würfels bei konstanter Kantenlänge exponentiell mit der Dimension, wie auch das Volumen einer (Hyper-)Kugel bei konstantem Radius exponentiell mit der Dimension wächst. Der größte Teil des Volumens eines hochdimensionalen Würfels im R^d liegt nahe an seiner $(d-1)$-dimensionalen Oberfläche. Ein typisches Element einer Indexpartition wird die Mehrheit des hochdimensionalen Datenraums beinhalten und sich nur in wenigen Dimensionen aufteilen, daher sind typische Indexpartitionen in hochdimensionalen Räumen sehr grob.

Das folgende einfache Beispiel soll helfen, einige der nicht-intuitiven Aspekte des „Fluch der Dimension“ zu illustrieren. Wir betrachten einen kubischen, d-dimensionalen Datenraum der Ausdehnung $[0,1]^d$, dessen Mittelpunkt bei $c := (0.5, ..., 0.5)$ liegt. Die Annahme „Jede d-dimensionale Kugel, welche alle $(d-1)$-dimensionalen Grenzen des Datenraums tangiert oder schneidet enthält auch den Mittelpunkt c des Datenraums.“ ist offensichtlich wahr für $d = 2$ (und ebenso für $d = 3$). Für $d = 16$ ist diese Annahme jedoch sicher falsch, wie das folgende Gegenbeispiel zeigt: Wir definieren eine Kugel S um den Punkt $p := (0.3, ..., 0.3)$. Dieser Punkt p hat einen euklidischen Abstand von $\sqrt{16 \cdot (0.5 - 0.3)^2} = 0.8$ vom Mittelpunkt. Hat S einen Radius von 0.7, berührt oder schneidet sie alle

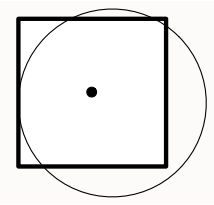


Abbildung 2.5.: Kugeln im hochdimensionalen Raum

15-dimensionalen Oberflächen des Datenraums, ohne jedoch den Mittelpunkt c zu enthalten. Phänomene dieser Art können einen starken negativen Einfluß auf die Performanz von Indexstrukturen ausüben, die ausschließlich in Hinblick auf niedrigdimensionale Daten entwickelt wurden.

Eine Taxonomie für „hochdimensionale Indexstrukturen“ Indexstrukturen für hochdimensionale Daten haben erst durch die weite Verbreitung von Multimediadatenbanken große praktische Relevanz erlangt. Es gibt daher noch keine etablierten Benchmarks für diese Indizes, was aussagekräftige Performanzvergleiche sehr erschwert. Die meisten Autoren vergleichen die Perfomanz ihrer Datenstrukturen und Algorithmen naturgemäß nur mit wenigen konkurrierenden Ansätzen. Dazu werden sehr häufig uniform zufällig verteilte Testdaten verwandt. Aus diesem Grund läßt sich nur eine „partielle Ordnung der Performanz“ aus der Gesamtheit der Veröffentlichungen ableiten.

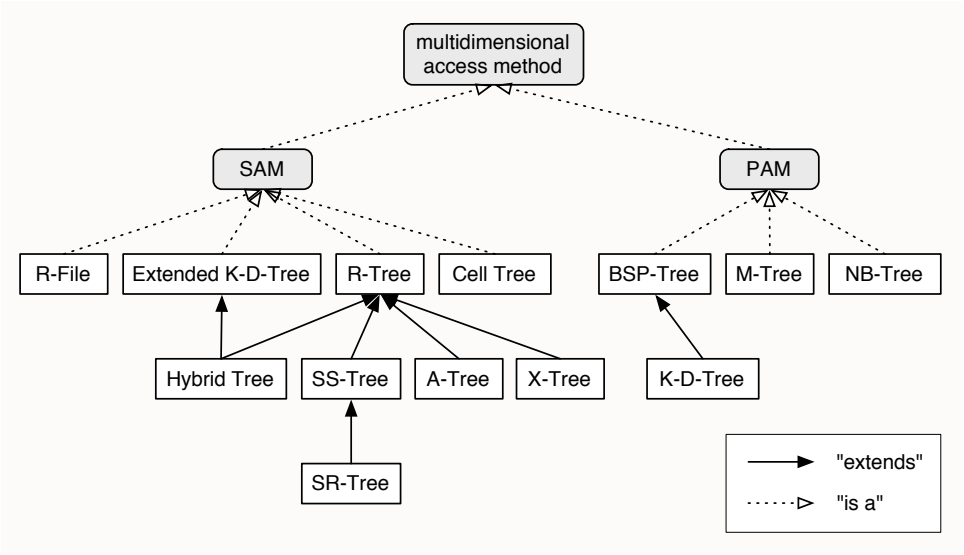


Abbildung 2.6.: Eine Taxonomie für „hochdimensionale Indexstrukturen“

Der R-Tree und seine Derivate

Der R-Tree [Gut84] ist eine am B-Tree orientierte Indexstruktur für räumliche Daten, die 1984 von Antonin GUTTMANN im Hinblick auf CAD- und GIS-Anwendungen entwickelt wurde. Im Gegensatz zu KD-Trees unterstützen R-Trees neben Punktdaten auch Intervalle (Hyperrechtecke). R-Trees zeigen eine sehr schlechte Performanz bei der Bearbeitung hochdimensionaler Daten, bieten aber die konzeptuelle Basis für eine Vielzahl komplizierterer Erweiterungen, die dieses Problem angehen.

Punktdaten variabler Dimension können von R-Trees nicht direkt verarbeitet werden; „Fehlende Dimensionen“ ergänzt man mit 0 oder einem speziellen Platzhalter.

Ein R-Tree ist ein dem B-Tree ähnlicher, „höhenbalancierter“ Suchbaum, dessen Blätter auf räumliche Datenobjekte verweisen. R-Trees sind vollständig dynamische Datenstrukturen: Such-, Einfüge- und Löschoperationen können beliebig gemischt werden und es ist keine periodische Reorganisation nötig. R-Trees mit N Einträgen und minimaler Knotenlänge m sind durch $O(|\log_m N|)$ in ihrer Höhe begrenzt.

M sei die maximale Anzahl von Einträgen je Knoten und $m \leq \frac{M}{2}$ deren minimale Anzahl. Ein R-Tree hat die folgenden Eigenschaften:

1. Jedes Blatt enthält zwischen m und M Einträge, sofern es sich nicht um die Wurzel handelt.
2. Jedes Blatt enthält das kleinste Hüll-Hyperrechteck, welches dessen räumliches Datenobjekt enthält.
3. Jeder innere Knoten enthält zwischen m und M Einträge, sofern es sich nicht um die Wurzel handelt.
4. Jeder innere Knoten enthält das kleinste Hüll-Hyperrechteck, welches die Hüll-Hyperrechtecke seiner Kinder enthält.
5. Die Wurzel hat mindestens zwei Kinder, sofern sie kein Blatt ist.
6. Alle Blätter liegen auf derselben Ebene.

Die Algorithmen für alle unterstützten Operationen findet man in [Gut84]. Exemplarisch geben wir hier den Algorithmus zur Suche in R-Trees an.

Algorithmus `Suche`: Finde in einem R-Tree mit Wurzel T alle Index-Einträge, deren Hyperrechtecke ein Suchhyperrechteck S überlappen:

1. `S1` [Unterbäume durchsuchen] Wenn T kein Blatt ist, prüfe für jeden Eintrag E, ob dieser S überdeckt. Falls ja, rufe `Suche` rekursiv für diesen Eintrag auf.
2. `S2` [Blatt durchsuchen] Wenn T ein Blatt ist, prüfe für jeden Eintrag E, ob dieser S überdeckt. Falls ja, füge diesen Eintrag zur Ausgabe hinzu.

Abbildung 2.7.: Algorithmus zur Suche in R-Trees

Node-Splitting in R-Trees Soll zu einem vollen Knoten ein Eintrag hinzugefügt werden, muß dieser geteilt werden. Dabei sollte die Wahrscheinlichkeit, daß bei späteren Anfragen beide der neuen Knoten durchsucht werden müssen, minimiert werden. Ein Knoten wird genau dann besucht, wenn sein Hüll-Hyperrechteck das Suchhyperrechteck vollständig überdeckt, daher sollte die Gesamtfläche beider Hüll-Hyperrechteckte nach einer Teilung minimiert werden.

Überlappen zwei Hüll-Hyperrechtecke R_1 und R_2 in einem R-Tree, müssen Suchanfragen, die innerhalb von $R_1 \cap R_2$ liegen, in beiden Rechtecken rekursiv fortgesetzt werden. Der „Fluch der Dimension“ bedingt jedoch, daß Überlappungen von Hüll-Hyperrechtecken mit steigender Dimension exponentiell zunehmen, weshalb bei Dimensionen von $d > 6$ oft schon beinahe alle Knoten des R-Tree durchsucht werden müssen. Bei hochdimensionalen Daten ist sogar lineare Suche im Durchschnitt schneller. Aus diesem Grund wurden in den letzten Jahren viele Erweiterungen des R-Trees für die Anwendung auf hochdimensionale Daten vorgeschlagen.

Der X-Tree Der X-Tree [BKK96] ist eine R-Tree-Erweiterung für „mäßig hochdimensionale“ Daten. Er verwendet eine Split History und das Konzept der Supernodes, um im Bedarfsfall überlappungsfreie Splits garantieren zu können. Blätter werden immer wie beim R-Tree geteilt, während innere Knoten nur „probeweise“ wie beim R-Tree geteilt werden. Resultiert eine solche Teilung in stark überlappenden Hüll-Hyperrechtecken, wird stattdessen ein überlappungsfreier Teilungsalgorithmus angewandt oder der Teilungsvorgang aufgeschoben und der betroffene Knoten zu einem Supernode erweitert. Bei niedrigdimensionalen Daten verhalten sich X-Trees sehr ähnlich zu R-Trees. Bei hochdimensionalen Daten sind sie zwar schneller als R-Trees, dennoch aber durchschnittlich langsamer als eine sequentielle Suche. Ihre beste Performanz erreichen X-Trees bei Daten von „mäßig hoher“ Dimension ($d \approx 15$).

Der M-Tree Die besprochenen „Distanzmetriken für Vektorräume“ gehören zur speziellen Klasse der L_p-Metriken. Beinahe alle Point Access Methods (PAMs) und Spatial Access Methods (SAMs) wurden im Hinblick auf das Indizieren von Vektorräumen entworfen und setzen die speziellen Eigenschaften der L_p-Metriken voraus. Eine Ausnahme bilden die „Metric Trees“, welche sich für das Indizieren beliebiger metrischer Räume eignen.

Der M-Tree [CPZ97] ist ein Metric Tree für die effiziente Bearbeitung von Ähnlichkeitsanfragen auf beliebigen metrischen Räumen. Im Gegensatz zu den meisten SAMs (und PAMs) minimieren M-Trees Distanzvergleiche und lassen sich daher sehr gut in Verbindung mit „teuren“ Metriken einsetzen.

Der NB-Tree als „einfache“ Alternative

Die NB-Tree-Struktur [FJ03] stellt eine sehr einfache und kompakte Methode zur Indexierung hochdimensionaler **Punkt**daten dar. Sie wurde von Manuel J. FONSECA und Joaquim JORGE auf der *„8th International Conference on Database Systems for Advanced Applications (DASFAA '03)“* vorgestellt. Im Gegensatz zu den betrachteten R-Tree-basierten Indexstrukturen unterstützt sie Punktdaten variabler Dimension auf natürliche Weise, allerdings werden keine Intervalldaten (Hyperrechtecke) unterstützt.

Im Folgenden sei d die Dimension des Datenraums und p ein d-dimensionaler Punkt. NB-Trees verwenden die euklidische Norm L_2 als Dimensionsreduktionsfunktion von $\mathbb{R}^d \rightarrow \mathbb{R}^1$:

$$\|p\| = \sqrt{p_0^2 + p_1^2 + \cdots + p_{d-1}^2}.$$

Die dimensionsreduzierten Datenpunkte, also die euklidischen Normen, werden dann mit Hilfe eines B^+-Trees indiziert, daher der Name **N**orm-**B-Tree**. B^+-Trees sind B-Trees, die Daten nur in ihren Blättern speichern. Die Blätter sind zusätzlich zu einer doppelt verketteten Liste verbunden. Die Performanz ist auch im Vergleich zu sehr viel aufwendigeren Indexstrukturen als gut zu bezeichnen, [FJ03] enthält detaillierte Benchmarks.

Zusammenfassung und Ausblick

Die Performanz fast aller der klassischen Indexstrukturen für räumliche Daten fällt mit steigender Dimension rapide ab. Erweiterungen solcher Indexstrukturen für hochdimensionale Daten sind oft sehr kompliziert und daher aufwendig und fehlerträchtig in der Implementierung. Schlimmer noch: Oft wiegt der Performanzgewinn diese Kompliziertheit nicht auf.

Bei der Entwicklung von Indexstrukturen für hochdimensionale Daten ist es extrem wichtig, die nichtintuitiven mathematischen Phänomene des „Fluchs der Dimension“ im Auge zu behalten. Möglicherweise bieten topologisch inspirierte Indexstrukturen wie z.B. Space Fitting Curves Ansatzpunkte für neue, bessere Lösungen. Sofern nur Punktdaten indiziert werden sollen, sind NB-Trees eine einfache und dennoch leistungsfähige Alternative für die Indizierung von Vektorräumen. M-Trees zeichnen sich vor allem durch ihre Eigenschaft aus, nicht auf Vektorräume beschränkt zu sein, sondern die Indizierung beliebiger metrischer Räume zu ermöglichen.

2.3.7. Clustering

Clustering ist das Aufsplitten einer Gesamtmenge in Teilmengen, sodass die Elemente einer einzelnen Teilmenge zueinander ähnlicher sind als die Elemente verschiedener Teilmengen. Clustering gehört zum unüberwachten Lernen, d.h. die Beispielmengen sind nicht vorklassifiziert. Mittlerweile wurden allerdings Methoden entwickelt, die diese Regelung aufweichen.

[Ber02] klassifiziert Clusteringalgorithmen folgendermaßen:

- Hierarchische Algorithmen
  - agglomerative Algorithmen
  - teilende Algorithmen
- Partitionierende Algorithmen
  - probabilistische Algorithmen
  - k-Medoids & k-Means
  - dichte-basierende Algorithmen
- u.a.

Dabei können sich die Algorithmen u.a. in folgenden, wichtigen Eigenschaften unterscheiden:

- Attributtypen
- Skalierbarkeit
- Anwendbarkeit auf hochdimensionale Daten
- Fähigkeit, Cluster von unregelmässiger Form zu finden
- Robustheit
- Zeitkomplexität
- Anhängigkeit von Ordnung der Daten
- Clusterzuweisung (hart vs. fuzzy)
- Abhängigkeit von a priori Wissen und benutzerdefinierten Parametern

Im Folgenden werden die Standardverfahren kurz vorgestellt. Von den Meisten gibt es mittlerweile viele Varianten. Wir beschränken uns auf die klassische Form.

Hierarchische Algorithmen

Hierarchische Clusteringalgorithmen zeichnen sich dadurch aus, daß sie ein so genanntes Dendrogramm als Ergebnis haben. Ein Dendrogramm ist ein binärer Baum, der die Abstände zwischen Clustern kennzeichnet. Hierarchische Algorithmen arbeiten entweder top-down (teilende) oder bottom-up (agglomerative). Erstere beginnen mit einem einzigen Cluster, der alle Beispiele enthält, und splitten diesen sukzessive in kleiner Cluster auf. Letztere betrachten zunächst alle Beispiele als einzelne Cluster und verschmelzen diese zu größeren Clustern. Teilende Algorithmen splitten dabei jeweils in die zwei am weitesten von einander entfernten Cluster auf, agglomerative verschmelzen die nächstliegenden.

Die Ähnlichkeit zwischen zwei Clustern kann dabei unterschiedlich interpretiert werden. Zum einen kann die Ähnlichkeit zwischen zwei Clustern als minimale Entfernung zwischen zwei Punkten aus den jeweiligen Clustern betrachtet werden, oder als Entfernung zwischen den Mittelpunkten der Cluster.

Die Vorteile dieser Algorithmen liegen darin, dass zum einen die Interpretation des Dendrogramms flexibel hinsichtlich der Granularität ist, und zum anderen, dass sich mit Leichtigkeit unterschiedliche Formen von Ähnlichkeit bzw. Abstand behandeln lassen, was die Anwendbarkeit auf beliebige Attributtypen zur Folge hat. Deutlicher Nachteil ist jedoch, dass einmal erstellte Cluster nie wieder aufgesplittet werden. Was einmal zusammen ist, bleibt es auch.

Partitionierende Algorithmen

Diese Art von Algorithmen erstellt eine direkte Einteilung der Daten statt einer Clusteringstruktur, wie z.B. das Dendrogramm der hierarchischen Algorithmen, und kann deshalb auf größere Datensätze angewandt werden. Es gibt verschiedene Ansätze, wie dieses Ziel erreicht wird. Allen liegt aber zunächst einmal iterative Optimierung und Neuverteilung von Daten zu Clustern zugrunde.

k-Medoids und k-Means sind die klassischen Clusteringalgorithmen. Cluster werden durch einen einzelnen Vektor repräsentiert. Bei k-Medoids ist das der zentralste Datenpunkt des Clusters, bei k-Means der Mittelpunkt der Elemente des Clusters. Die Vorteile von k-Medoids bestehen darin, auf beliebige Attributtypen angewandt werden zu können (bei k-Means ist dies schwierig) und eine gewisse Resistenz gegen Outlier. Die durch k-Means erstellten Cluster haben bei numerischen Attributen einfache geometrische und statistische Bedeutung. Der Nachteil besteht darin, dass die resultierenden Cluster sehr ungleichmässig in der Anzahl ihrer Elemente sein können.

Eine weitere Unterkategorie der partitionierenden Algorithmen sind probabilistische Clusterer. Diese nehmen an, dass die Daten unabhängig und gleichmäßig z.B. aus einem Mixture Model gezogen worden sind, und haben zum Ziel, die Parameter dieser Wahrscheinlichkeitsverteilung zu bestimmen. Als Mittel bedienen sie sich z.B. der Maximum Likelihood Methode oder dem Expectation Maximization (EM) Algorithmus.

Die dritte Klasse besteht aus den dichtebasierenden Clusteringalgorithmen, die davon ausgehen, dass Cluster dicht beieinanderliegende Komponenten haben, die es zu finden gilt. Die Vorteile sind, dass diese Algorithmen beliebig geformte Cluster erkennen können, sowie eine gute Skalierbarkeit. Das Clusteringprinzip sorgt zudem für einen natürlichen Schutz gegen Outlier. Allerdings lassen sich die Ergebnisse nur schwer interpretieren.

Für eine kurze Übersicht über Clustering-Algorithmen siehe [JMF99].

Non-Redundant Data Clustering

Die Idee dieses Ansatzes von Gondek und Hofmann [GH04] besteht darin, eine vorklassifizierte Menge von Daten so zu strukturieren, dass das resultierende Clustering quasi orthogonal zu der vorhandenen Klassifikation ist, und so neue Strukturen aufzeigt.

Zu diesem Zweck bedient es sich des gemeinsamen Informationsgehalt (Mutual Information), definiert als

$$I(A,B) \equiv \sum_a \sum_b P(a,b) log \frac{P(a,b)}{P(a)P(b)}$$

Ausgedrückt mit Hilfe des Begriffs der Entropie (mittlerer Nachrichtengehalt der Meldungen einer Informationsquelle) ergibt sich:

$$I(A,B) = H(A) - H(A|B) = H(B) - H(B|A)$$

Das Ziel ist das Finden einer stochastischen Abbildung $P_{C|X}$ von einer Menge von Beispielen X in eine Menge von Clustern C, die jedem Beispiel x die Wahrscheinlichkeit zuweißt, dass es zu Cluster c gehört.

Seien Y die relevanten Merkmale und Z gegebenes Hintergrundwissen. Eine Maßzahl dafür, wie viel C und Z also die neu gefundenen Cluster und das Hintergrundwissen, über die relevanten Features Y

im Vergleich zu Z allein aussagen, ergibt sich durch $I(C;Y|Z)$. Dies gilt es zu maximieren, da das Ziel ja darin besteht neue Informationen über X zu erhalten. Auf der anderen Seite möchte man Over-Confidence vermeiden. Deshalb beschränkt man den gemeinsamen Informationsgehalt von C und X: $I(C;X)$. Zusammen ergibt dies das *Conditional Information Bottleneck*:

$$P^*_{C|X} = argmax_{P(C|X)\in P} I(C;Y|Z) \quad mit \quad P \equiv \{P_{C|X} | I(C;X) \leq C_{Max}\}$$

In dieser Form weißt das Ergebnis allerdings noch einen Mangel an globaler Koordination auf, weshalb eine weitere Einschränkung eingeführt wird. Neben $I(C;Y|Z)$, dem gemeinsamen Informationsgehalt von C und Z, wollen wir zusätzlich wissen, wieviel relevante Information C allein erhält, also $I(C;Y)$. Dies führt zum *Coordinated Conditional Information Bottleneck*

$$P^*_{C|X} = argmax_{P(C|X)\in P} I(C;Y|Z) \quad mit \quad P \equiv \{P_{C|X} | I(C;X) \leq C_{Max}, I(C;Y) \geq I_{Min}\}$$

Experimente haben ergeben, dass $I_{Min} \geq 0$ zu globaler Consistence führt.

Multi-View Clustering

Der Ansatz von Steffen Bickel und Thomas Scheffer [BS04] splittet die Features der Daten in zwei unabhängige Untermengen, die konditionell unabhängig sein müssen, und überträgt Ideen des Multi-View Lernens aufs Clustering, was zumindest bei partitionierenden Algorithmen zu besseren Ergebnissen führt.

2.4. Multimediadaten

2.4.1. Überblick Multimediasuche/indexing

Information Retrieval könnte man als Informationswiedergewinnung übersetzen. Wenn man Information Retrieval (IR) definieren möchte, könnte man unter Information Retrieval - genauer wäre es, von Information Storage und Retrieval zu sprechen - alle Verfahren in der Informatik zusammengefassen, die der Aufbereitung, Speicherung und Wiedergewinnung von Wissen dienen.[10]

Das in einem IR-System gespeicherte Wissen ist im Prinzip nicht beschränkt. Dies können Texte, multimediale Dokumente, aber auch Fakten, Regeln oder semantische Netze sein. Das klassische Anwendungsgebiet des IR sind Literaturdatenbanken, die heute in Form digitaler Bibliotheken zunehmend an Bedeutung gewinnen. IR ist besonders populär geworden durch die Anwendung in Internet-Suchmaschinen; dadurch kommt jeder Internet-Nutzer mit IR-Methoden in Berührung[Fuh04].

Es gibt zwei Konzepte, die das IR prägen und es von der herkömmlichen Suche in Datenbanken abgrenzen: Vagheit und Unsicherheit.

Vagheit: Der Benutzer kann sein Informationsbedürfnis nicht präzise und formal ausdrücken und die Anfrage enthält daher vage Bedingungen.

Unsicherheit: Dem System fehlen Kenntnisse über den Inhalt der Dokumente. Dies führt zu fehlenden und fehlerhaften Antworten. Bei Texten bereiten z.B. Homonyme und Synonyme Probleme.

Durch diese beiden Konzepte wird auch „Information Retrieval Systeme (IRS)“ definiert:

„IRS sind interaktive Informationssysteme für vage Anfragen und unsicheres Wissen“ [Fuh04].

Es gibt dabei Anforderungen an ein IRS:

Relevanz: Bezeichnet den Grad der Übereinstimmung der Suchfrage und die inhaltlichen Aussage eines Dokumentes.

[10] siehe: `http://is.uni-sb.de/studium/handbuch/exkurs.ir.html`

Angemessenheit: Beschreibt, inwieweit die Antwort dem Informationsbedarf entspricht.

Nützlichkeit: Information, die nicht im Kontext liegt, aber für eine andere Fragestellung nützlich ist, ist nicht relevant, aber nützlich. Diese Bewertungsart ist eher geeignet für ein Interessenprofil als ein Informationsbedarf.

Dynamik: Das Einfügen, Ändern und Löschen von Daten sollte beim IRS ohne großen Aufwand möglich sein.

Sortierung nach Relevanz: Beschreibt, wie die wichtigsten (relevantesten) Daten (z.B. bei Suchmaschinen die Seiten) geordnet werden müssen.

Indexierung und Informationretrieval

Eine der zentralen Operationen in einem IRS ist die Indexierung. Dabei werden Dokumente analysiert und daraus Deskriptoren abgeleitet, die den Inhalt eines Dokumentes repräsentieren. Es wird jedem Deskriptor ein Wert zugeordnet, der seine potentielle Bedeutsamkeit für das Retrieval widerspiegelt. Dabei erfüllt die Liste der Deskriptoren im wesentlichen zwei Funktionen. Zum einen repräsentieren sie ein Dokument über eine stichwortartige Kurzbeschreibung, zum anderen geben sie den Inhalt des Dokumentes wieder. Weiters werden im Indexierungsprozess die auf den Dokumentinhalt bezogenen Deskriptoren standardisiert, was der

- Suche nach Dokumenten, die für die Anfrage eines Benutzer relevant sind,
- Verknüpfung der Dokumente, die thematisch zusammengehören
- Relevanzbestimmung der einzelnen Dokumente

bezogen auf eine Suchanfrage dient.[11]

Die Güte des Indexierungsprozesses bestimmt letztendlich die Effektivität des Informationssystems bei der Recherche. Und diese Qualität kann mit Hilfe von zwei Parametern gemessen werden: Recall und Precision.

- **Recall** ist das Verhältnis der Anzahl, der vom System aufgrund der Suchanfrage gefundenen Dokumente, zu allen hinsichtlich des Suchbegriffes relevanten Dokumenten und liefert ein Maß für die Vollständigkeit des Suchergebnisses.
- **Precision** dagegen drückt die Genauigkeit der Suche als Anteil der relevanten Dokumente, die aufgrund der Suchanfrage gefunden wurden, aus.

Bei der Indexierung werden zwei Arten der Indexierung unterschieden: Die manuelle und die automatische Indexierung.

Die manuelle Indexierung wird von einer Person, dem Indexierer, durchgeführt, an den hohe Anforderungen bezüglich fachlicher Kompetenz gestellt werden. Zwar hat er Vokabularlisten und Terminologiebeschreibungen zur Verfügung, aber er muss genaue Kenntnisse über die Vokabulare haben, da auch die Pflege und Erweiterung der Vokabulare in seinem Aufgabenbereich liegen.

Bei der automatischen Indexierung ist das Ziel, aus den Dokumenten automatisch sinntragenden Wörter zu extrahieren. Das automatische Extrahieren besteht darin, den Wörtern Deskriptoren zuzuordnen. Dabei müssen die nicht aussagekräftige Stoppwörter eliminiert und vielmehr eine Auswahl sinntragender Wörter getroffen werden. Diese Auswahl kann entweder aufgrund der Worthäufigkeit in den Dokumenten oder aufgrund intellektuell erstellter Wortlisten erfolgen.

Als Beispiel für die Indexierung werden die invertierten Listen erläutert.

[11] siehe: `http://wwwai.wu-wien.ac.at/~koch/lehre/inf-sem-ws-00/weiss/indexierung.html`

Eine *invertierte Liste* für eine Dokumentmenge ist die sortierte Liste (der Index) der Terme der Dokumente, wobei jeder Term in der invertierten Liste Verweise auf genau die Dokumente enthält, in denen der Term auftritt [12]. Die Komponenten sind die Dokumente, invertierte Liste und Dictionary (Wörterbuch). Das Ziel dabei ist, dass Dokumente über die Wörter gefunden werden, die sie enthalten. Dazu werden zu jedem indexierten Term eine Liste der Dokumente gespeichert werden, welche den Term enthalten. Primäre Datenstrukturen der meisten IR-Systeme sind invertierte Listen. Oft benutzen auch Suchmaschinen invertierte Listen. Vorteile der invertierte Listen sind, dass sie einfach zu implementieren sind, sehr effizient sind und Synonyme einfach gehandhabt werden können. Ein Nachteil ist der große Overhead beim Speicherverbrauch. Auch die hohen Kosten für die Änderung und Reorganisation bei einem dynamischen Dokumentenbestand ist ein Nachteil der invertierten Listen.

Multimedia Information Retrieval

Den Begriff Multimedia kann man als Kombination verschiedener Medientypen definieren. Genauer gesagt ist es die Kombination diskreter und kontinuierlicher Medien. Die Informationen in diskreten Medien bestehen ausschließlich aus einer Folge einzelner Elemente. Beispiele sind Texte oder Grafiken. Bei kontinuierlichen (zeitabhängigen) Medien stecken die Informationen nicht nur in einem Wert, sondern auch im Zeitpunkt des Auftretens. Als Beispiele kann man Ton oder bewegte Bilder nennen.

Wie schon oben erwähnt ist das Wissen in einem IRS nicht beschränkt. In diesem Abschnitt geht es mehr um die multimediale Daten und deren Retrieval. Es gibt dabei verschiedene Bereiche des Multimedia Information Retrieval: Text-Retrieval, Bild-Retrieval, Audio-Retrieval und Video-Retrieval.

Text-Retrieval ist ein gut erforschtes Gebiet des Multimedia Information Retrieval. Aktuelle Forschung in diesem Bereich betrifft vor allem das Erschließen von Texten, d.h. linguistische Ansätze zum Verstehen von Sätzen und dem Aufbau von Relationen zwischen Termen (z.B. Überbegriff/Unterbegriff oder „Teil von“).

Im Bereich **Bild-Retrieval** gibt es seit Beginn der 90er Jahre einen Aufschwung. Zu den erforschten Themen gehören vor allem das Erfassen/Extrahieren von charakteristischen Merkmalen aus Bildern, die Definition von Ähnlichkeit zwischen zwei Bildern sowie Indexstrukturen zur effizienten Suche.

Beim **Audio-Retrieval** gibt es schon interessante Ideen und Ansätze. Gute Resultate hat man bei der Spracherkennung erzielt. Bei der Musiksuche ist aber noch kein Trend sichtbar, da man nicht weiß, welche Merkmale sich gut eignen, um Musik zu charakterisieren.

Im **Video-Retrieval** hat IBM mit QBIC (Query By Image Content) einen ersten Meilensteien gesetzt. Gute Ergebnisse liegen vor allem bei der „Shot Detection“ (Szenenerkennung) und der „Key Frame Extraction“ (Extraktion charakteristischer Bilder) vor.

Im Weiteren wird nun auf den Bereich Audio-Retrieval eingegangen.

Audio-Retrieval kann grob in vier Kategorien aufgeteilt werden[13]:

1. Suche auf Metadaten (*keyword-based search*)
2. Suche mit akustischen Merkmalen wie loudness, pitch, ... (*content-based similarity search*)
3. Suche mit Noten, respektive Tonintervallen (*search by humming*)
4. Suche in gesprochenem Text (*speech recognition*)

Suche auf Metadaten: Eine Audiodatei wird durch eine Menge von Attributen, wie z.B. Artist oder Album beschrieben. Je nach Applikation kann durch eines oder mehrere dieser Attribute nach Audiodaten gesucht werden. Beispiele sind z.B. Napster und Gnutella.

[12] siehe Abschnitt: Retrievalmodelle -> Boolsches Modell, Invertierte Listen: `http://www2.inf.fh-rhein-sieg.de/~pbecke2m/retrieval/`

[13] siehe Kapitel 6: Audio and Video Retrieval: `http://www.dbis.ethz.ch/education/ss2005/mmr_05/mmr_content`

Suche mit akustischen Merkmalen: Diese Methode wird später noch ausführlich erläutert, wird aber der Vollständigkeit halber hier kurz vorgestellt. Die Signalinformation wird mittels charakteristischer Merkmale beschrieben. Das Audiosignal wird aber typischerweise nicht durch einen einzelnen Vektor repräsentiert, sondern durch eine zeitabhängige Vektorfunktion. Bei der Suche wird nicht nur eine Audiodatei, sondern auch die Stelle gefunden, an der die gesuchten akustischen Merkmale auftreten. Der Vorteil dieser Methode ist es, daß sie auf beliebige Formate angewendet werden kann. Ihr Nachteil ist, dass die Suche rudimentär ist und nicht in allen Fällen weiter hilft.

Suche mit Noten, respektive Tonintervallen: Da bei Musikstücken akustische Merkmale wenig Sinn machen, sollte man die Ähnlichkeit über Takt, Tempo oder Noten definieren. Zur Zeit gibt es sehr vielversprechende Ansätze, die die Suche nach ähnlichen Melodien zulassen. Dabei gibt es jedoch zwei Probleme zu lösen:

- Erkennung der Noten aus dem Audiosignal
- Definition von Melodie

Bei diesem Ansatz müssen alle Musikstücke (gespeichert als Notenfolge, z.B. im MIDI-Format) analysiert werden. Für jede Note wird dann einer der drei folgenden Werte gespeichert (außer für die erste Note des Liedes):

- D (down) die vorherige Note war höher (Melodie geht „runter“)
- U (up) die vorherige Note war tiefer (Melodie geht „hoch“)
- S (same) die vorherige Note war dieselbe (Melodie bleibt)

Ein Musikstück kann dann als eine Zeichenfolge bestehend aus D, U und S dargestellt werden.

Man kann dann eine Anfrage auf zwei verschiedene Weisen formulieren:

- Summen der Melodie
- Direkte Eingabe der Zeichenfolge

Für eine Anfrage müssen dann diejenige Musikstücke gefunden werden, die diese Zeichenfolge enthalten.

Suche in gesprochenem Text: Bei diesem Verfahren „befreit“ man zuerst das Audiosignal von allen uninteressanten Merkmalen wie z.B. Tonhöhe, Tempo und Lautstärke, da diese keine Rolle bei der Erkennung von gesprochenem Text spielen. Das Resultat wird danach quantisiert. Letztlich wird das Audiosignal in einen Strom von Phonemen umgewandelt.

Das Retrieval kann nun entweder auf dem erkannten Phonemstrom erfolgen, oder aber man versucht die Phoneme zu Wörtern zusammenzusetzen.

- Retrieval im Phonemstrom: Eine Anfrage kann man auf zwei Arten eingeben:
  - Als Audiostrom, d.h. der Benutzer spricht die Anfrage in ein Mikrophon.
  - Als Textstrom, d.h. der Benutzer gibt die Anfrage über eine Tastatur ein.

  Da die Erkennung von Phonemen aus dem Audiostrom nicht perfekt ist, müssen die Retrievalverfahren fehlertolerant sein.
- Retrieval im Wörterstrom: Falls die Sprachkomponente Wörter liefert, so können auch Wörter als Suchterme eingesetzt werden. Man muss allerdings die Fehlerhaftigkeit der Sprachkomponente berücksichtigen. Es kann z.B. passieren, dass ein Wort nicht oder falsch erkannt wird. Um die Probleme zu umgehen, wird häufig ein Wörterbuch benutzt. Ebenso können Korrekturprogramme eingesetzt werden, um noch mehr Fehler zu eliminieren.

Man kann sagen, dass Audio-Retrival, abgesehen von „speech recognition“, noch in den Kinderschuhen steckt. In Zukunft wird noch einiges an Grundlagenforschung notwendig sein. Eine der konkreten und interessantesten Problemstellung ist die Suche nach ähnlicher Musik, was auch eine Funktionalität der entwickelten Software (Nemoz) seien wird.

2.4.2. Extraktion von Merkmalen aus Audiodaten

Maschinelles Lernen aus Merkmalsvektoren ist ein sehr gut erforschtes Feld der künstlichen Intelligenz. Der folgende Abschnitt soll einen kurzen Überblick über die Methoden zur Gewinnung solcher Merkmalsvektoren aus digitalen Audiodaten geben. Obwohl es sich bei der Merkmalsextraktion aus Audiodaten um ein noch recht junges Forschungsgebiet handelt, existiert bereits eine Fülle von Verfahren, weshalb aus Platzgründen nur eine kleine Auswahl vorgestellt werden kann. [Mie04] und [Tza02] bieten beide umfassende, aber dennoch verständliche Einführungen in das Thema.

Grundlagen

Merkmalsextraktion Da es sich bei digitalen Audiodaten um endliche, univariante Zeitreihen handelt, lassen sich diese direkt als Merkmalsvektoren auffassen und werden so dem maschinellen Lernen zugänglich gemacht. Dieser Ansatz bringt allerdings Probleme mit sich. So sind Audiodaten im allgemeinen sehr umfangreich und enthalten eine relevante zeitliche Ordnung die von Lernverfahren nicht direkt genutzt werden kann. Schwerer wiegt noch, daß sich verschiedene Instanzen sehr stark in ihrer Länge unterscheiden können. Fasst man Audiodaten also direkt als Merkmalsvektoren auf, hat man es mit im allgemeinen sehr langen, in der Länge stark variierenden Vektoren mit sehr geringer Dichte an für nachfolgende Lernverfahren relevanter Information zu tun.

Die Methoden der Merkmalsextraktion lösen diese Probleme. Durch Darstellung der Audiodaten als Zeitreihen werden diese allen bekannten Methoden der Wertereihenanalyse zugänglich gemacht. Diese Methoden liefern Merkmalsvektoren fester und handhabbar kleiner Länge. Des Weiteren wird die zeitliche Ordnung der Eingabedaten explizit dargestellt. Merkmalsextraktion liefert also Vektoren fester, handhabbar kleiner Dimension mit hoher Dichte an für die nachfolgenden Lernverfahren relevante Information. Einigen Merkmalen läßt sich eine intuitiv verständliche Semantik wie z.B. Tempo, Tonart, Klangfarbe, etc. zuordnen, wodurch ihre Extraktion selbst ohne die Verwendung nachgeschalteter Lernverfahren einen Wert an sich darstellt.

Audiodaten als Zeitreihen Wie bereits erwähnt, müssen Audiodaten als Zeitreihen dargestellt werden, um sie den Methoden der Wertereihenanalyse und damit der Merkmalsextraktion zugänglich zu machen. Unter einer allgemeinen Wertereihe versteht man eine Abbildung

$$x : \mathbb{N} \to \mathbb{R} \times \mathbb{C}^m.$$

Anstelle von $x(n)$ schreibt man auch x_n und für eine endliche Folge der Länge n auch $(x_i)_{i\in\{1,\dots,n\}}$. Die erste Komponente d_i eines Reihenelements $x_i = (d_i, \cdot)$ gibt die Position auf einer Zahlengeraden an und heißt Indexkomponente. Die Zahlengerade nennt man hier auch Indexdimension. Es gilt $i < j \Leftrightarrow d_i < d_j$ für alle $i, j \in \{1, ..., n\}$. Die zweite Komponente w_i eines Reihenelements $x_i = (\cdot, w_i)$ entspricht dem Wertevektor für jedes Element. Eine Zeitreihe ist eine Wertereihe, deren Indexdimension dem Zeitkontinuum entspricht. Man unterscheidet zwischen univarianten Zeitreihen, deren Werteraum eindimensional ist und multivarianten Zeitreihen, die einen höherdimensionalen Werteraum besitzen.

Audiodaten lassen sich als endliche, univariante und äquidistante Zeitreihen beschreiben, deren (einzige) Wertedimension der Elongation, d.h. der momentanen Schwingungsweite („Auslenkung einer Lautsprechermembran“), entspricht:

$$\mathrm{audiodata} : \mathbb{N} \to \mathbb{R} \times \mathbb{R}.$$

Äquidistant bedeutet hier, daß der Zeitabstand zwischen zwei Elongationsmessungen (Samples) immer gleich groß ist. Die Anzahl der Elongationsmessungen wird als Samplingrate bezeichnet und in Kilohertz (kHz) angegeben. Die Genauigkeit dieser Samples wird als Auflösung (Resolution) bezeichnet und in Bit angegeben. Das PCM-Verfahren der Audio-CD verwendet beispielsweise eine Samplingrate von 44.1 kHz bei einer Auflösung von 16 Bit.

Eine Systematik für Audiomerkmale

Nachdem die Grundlagen der Merkmalsextraktion aus Audiodaten dargelegt wurden, sollen nun einige der wichtigsten Verfahren, die auch in Nemoz Verwendung finden, exemplarisch umrissen werden.

Da die Wertereihenanalyse und damit die Merkmalsextraktion sehr allgemeine Konzepte darstellen, wurden in so unterschiedlichen Disziplinen wie Wirtschaftsmathematik, Informatik, oder auch Physik oft weitgehend unabhängig voneinander Methoden entwickelt, ohne daß sich eine gemeinsame Begrifflichkeit herausbilden konnte. Allein schon deshalb ist eine systematische Betrachtung wie [Mie04], auf der diese Einführung basiert, sinnvoll. Weiterhin hilft eine solche „Systematik für Audiomerkmale“ Redundanzen zu erkennen und ermöglicht die algebraische Kombination von Methoden zur Erzeugung neuer, synthetischer Methoden.

Eine Methode der Wertereihenanalyse ist eine Funktion, die auf Wertereihen definiert ist. In einem ersten Schritt hin zu einer Systematik gruppiert man die verschiedenen Methoden anhand ihres Ausgabetyps. Eine Methode, die als Ausgabe eine Wertereihe liefert, heißt Transformation, während eine Methode, die als Ausgabe einen Skalarwert (ein Merkmal) liefern, als Funktional bezeichnet wird. Transformationen sind also in gewisser Weise Vorverarbeitungsoperationen, während Funktionale die eigentlichen Merkmale liefern. Die Kombination von Transformationen und Funktionalen zur Erzeugung eines Merkmals ergibt den Methodenbaum dieses Merkmals, dessen Wurzel immer ein Funktional ist.

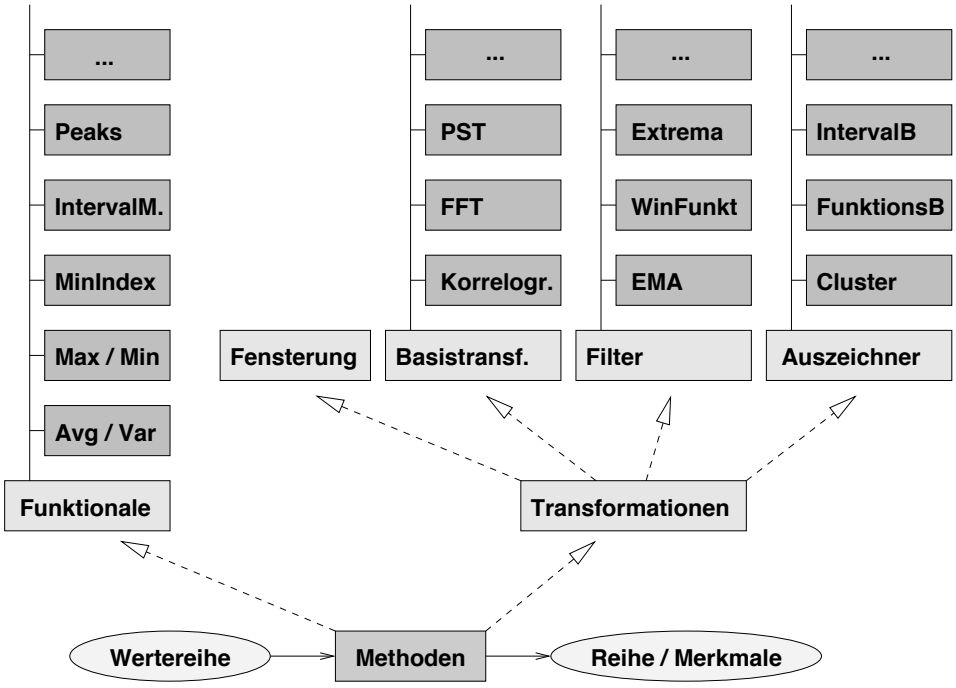
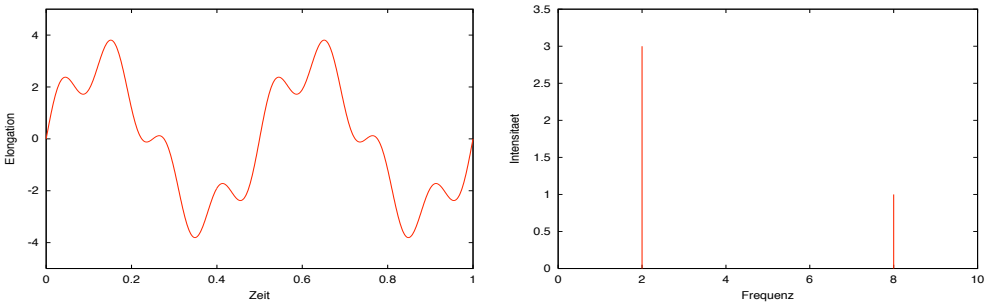


Abbildung 2.8.: Merkmalssystematik aus ([MKFR03])

Basistransformationen Eine wichtige Transformationsklasse bilden die Basistransformationen, welche Elemente eines Vektorraums in einen Vektorraum mit anderer Basis abbilden. Man unterscheidet bijektive und nicht-reversible Basistransformationen. Dabei muss eine Basis nicht endlich sein; Die Fourier-Transformation hat eine Zielbasis, die aus allen Sinusoiden besteht. Diese Basis erzeugt den Fourier-Raum, einen Funktionenraum. Dieser Raum wird oft auch als Frequenzraum bezeichnet, ein Graph in diesem Raum nennt man Frequenzspektrum. Grundlage der Fourier-Transformation ist der Satz von Fourier, welcher besagt, daß sich jede periodische Funktion als Summe von Sinusoiden darstellen läßt. Die Rückgewinnung einer Wertereihe aus ihrer Darstellung als Summe von Sinusoiden wird auch als Fourier-Synthese bezeichnet. Die Fourier-Transformation ist eine essentielle Grundlage



(a) "Zeit-Raum" (b) Frequenz-Raum

Abbildung 2.9.: Zeit- und Frequenzraumdarstellung einer Zeitreihe

für eine Vielzahl von spezielleren Audiomerkmalen. So lassen sich die durch Fourier-Transformation gewonnenen Frequenzspektren zur Analyse der Klangfarbe einzelner Instrumente eines Musikstücks oder zur Erkennung von Melodielinien und Harmonien nutzen. Aufgabe der Cochlea des Menschen ist unter anderem die Aufspaltung des Schalls in ein Frequenzspektrum, was die Bedeutung der Fourier-Transformation für die maschinelle Verarbeitung von Audiodaten unterstreicht.

Eine Verallgemeinerung der der Fourier-Transformation zugrundeliegenden Fourier-Analysis stellt die Wavelet-Theorie dar. Wavelets sind erst einmal Funktionen, die zur Repräsentation anderer Funktionen verwendet werden können. Sie spielen als Basisfunktionen eine ähnliche Rolle wie die Sinusoide der Fourier-Synthese. Im Gegensatz zu Sinusoiden sind Wavelets lokale Funktionen, also Funktionen mit endlichem Definitionsbereich. Wavelets eignen sich gut zur Approximation von unstetigen Funktionen mit scharfen Spitzen. Durch geeignete Wavelets können viele Funktionen extrem kompakt dargestellt werden.

Die Fourier-Transformation einer Wertereihe läßt sich in Zeit $O(n \log n)$ effizient berechnen, ihre Wavelet-Transformation sogar in Zeit $O(n)$, wobei n die Länge der Zeitreihe ist.

Eine dem Frequenzspektrum eng verwandte Darstellung ist das Korrelogramm einer Wertereihe. Dabei handelt es sich um einen Graphen, auf dem der Korrelationskoeffizient der Reihe gegen die Schrittweite (Lag) aufgetragen ist. Der Korrelationskoeffizient (aus dem Intervall $[-1, 1]$) ist ein Maß für die Autokorellation von Werten einer Wertereihe, die einen der Schrittweite entsprechenden Abstand zueinander haben. Ein Korellationskoeffizient von 0 bedeutet „keine Korrelation“, bei einem Koeffizienten von 1 ändern sich die Werte in gleichem Maße, bei einem Koeffizienten von -1 in umgekehrtem Maße. Eine einfache Anwendung von Korrelogrammen ist die Bestimmung des Tempos eines Musikstücks.

Dynamische Systeme Die bisher behandelten Transformationen betrachteten Musik in ihrer Rolle als Superposition harmonischer Schwingungen. Aus einer anderen Perspektive stellt sich Musik als komplexes dynamisches System mehrerer nichtlinearer Erzeuger dar. Das Verhalten eines solchen nichtlinearen Systems kann durch nichtlineare Differentialgleichungen beschrieben werden. Dabei ist die

Dimension eines solchen Systems als die Mindestanzahl der Größen, die dieses System vollständig beschreiben, definiert. Beispielsweise wird eine Masse, die unter dem Einfluss der Schwerkraft an einem Pendel schwingt, durch den Auslenkungswinkel Θ und die Geschwindigkeit $\dot{\Theta}$ vollständig beschrieben. Ein solches Pendel wird durch die nichtlineare Differentialgleichung

$$\ddot{\Theta}(t) = -mg \sin \Theta(t)$$

vollständig beschrieben. Der Zustandsraum eines dynamischen Systems besitzt als Basisvektoren dessen Zustandsvariablen. Die Elemente des Raums sind die Werte der Zustandsvariablen zu jedem beobachteten Zeitpunkt.

Der Zustandsraum eines Musikstücks läßt sich zur Gewinnung interessanter Merkmale nutzen, allerdings sind die Zustandsvariablen des Systems, welches das Musikstück erzeugt hat, unbekannt. Um dieses Problem zu umgehen, konstruiert man einen zum Zustandsraum topologisch identischen Raum, den Phasenraum: Sei $\mathbf{p}_i = (x_i, x_{i+d}, x_{i+2d}, ..., x_{i+(m-1)d})$. Dabei stellt d eine zeitliche Verzögerung (Delay) und m die Dimension des entstehenden Phasenraums dar. Die Menge

$$P_{d,m} = \{\mathbf{p}_i | i = 1, ..., n - (m-1)d\}$$

bildet die Transformierte der Reihe $(x_i)_{i \in \{1,...,n\}}$ im Phasenraum.

Filter Filter sind nichtreversible Transformationen, die Elemente im gleichen Raum liefern (es handelt sich um Vektorraumendomorphismen). Der Begriff ist durch die bekannten, aus der Elektrotechnik stammenden Beispiele des Hochpass- Bandpass- und Tiefpassfilters motiviert. Gängige Filter sind gleitende Durchschnitte, Extrema-Filter, Frequenzpässe und Funktionsfilter. Der Bark-Filter gewichtet das Frequenzspektrum entsprechend der menschlichen Hörfläche und kann zur Extraktion von Frequenzbändern, für die das menschliche Gehör besonders empfindlich ist, genutzt werden.

Auszeichnung von Audiodaten Eine Auszeichnung (Markup)

$$A : B \to E$$

weist einem Intervall E einer Dimension einer Wertereihe eine bestimmte (nominelle) Eigenschaft B zu. Bekannte Beispiele für Auszeichnungen in Texten sind die Tags der Auszeichnungssprachen `SGML` und natürlich `HTML`. Auszeichnung von Wertereihen ermöglicht es, bestimmte Bereiche einer Wertereihe getrennt zu analysieren. Beispielsweise folgen Popmusikstücke in der Indexdimension ihrer Zeitreihe oft dem Formschema Strophe-Refrain-Strophe-Refrain-Strophe-Übergang-Refrain. Allerdings können Intervalle jeder Dimension ausgezeichnet werden, nicht nur Intervalle der Indexdimension. So sind bedingt durch Tonart und instrumentelle Besetzung eines Musikstücks bestimmte Frequenzen stärker vertreten als andere. Diese Frequenzbänder können durch Auszeichnung markiert werden. Intervallauszeichnungen können beispielsweise durch K-Means-Clustering gefunden werden und bestehen aus Startwert, Endwert, Typ (nomineller Eigenschaft) und Dichte.

Fensterung Eine Transformation heißt Fensterung, wenn sie ein Fenster der Breite w mit Schrittweite s über ihre Eingabereihe bewegt und in jedem Schritt ein Fensterfunktional berechnet, um als Ergebnis die Wertereihe der Fensterfunktionalwerte y_j zu liefern:

$$y_j = F((x_i)_{i \in \{j \cdot s, ..., j \cdot s + w\}}).$$

Fensterungen sind also Kombinatoren für Methoden.

Statistische Kenngrößen Als „statistische Kenngrößen“ bezeichnet man die klassischen Funktionale der deskriptiven Statistik. Dabei handelt es sich um Mittelwerte wie arithmetisches, quadratisches, geometrisches und harmonisches Mittel, Median und absolutes Mittel. Im Weiteren zählen Varianz, Zentroid, Amplitude, Minima und Maxima sowie die Länge der Wertereihe zu den statistischen Kenngrößen.

Funktionscharakteristika Funktionale, die Extrema einer Wertereihe berechnen, nennt man auch Funktionscharakteristika. Zu dieser Funktionalklasse zählen das Peak-Funktional und die Werteextraktion nach einer Extrematransformation. Peaksuche ist vor allem in Spektren und Verteilungen sinnvoll. So liefert das k-Peaks-Funktional für eine Wertereihe die Stelle, die Höhe und die Breite der k höchsten Peaks. Angewandt auf ein Spektrum erzeugt dieses Funktional also $3k$ Merkmale, die für die Klangfarbe eines Musikstücks charakteristisch sind.

Das k-Extrema-Funktional liefert für eine Wertereihe die Stelle und Höhe der k Extrema mit dem höchsten Betrag. Im Gegensatz zum sehr ähnlichen k-Peaks-Funktional liefert dieses Funktional auch Minima. Die Differenzen zwischen Extrema können beispielsweise als alternatives Mittel zur Bestimmung des Tempos eines Musikstücks genutzt werden.

Audiomerkmale Als Audiomerkmale werden Funktionale bezeichnet, die aus der Psychoakustik und speziell aus der digitalen Sprachverarbeitung stammen. Das Spectral Flatness Measure Funktional (SFM) ist dasjenige Funktional, welches aus einem Spektrum das Verhältnis zwischen geometrischem und arithmetischem Mittel bestimmt. Dieses Funktional kann zur Unterscheidung von Rauschen von Musiksignalen genutzt werden. Das Funktional, welches aus einer Wertereihe das Verhältnis zwischen Maximum und arithmetischem Mittel bestimmt, heißt Spectral Crest Factor Funktional (SCF). Es zeigt an, wie klar einzelne Peaks in einem Spektrum „heraustreten“.

2.4.3. Audio Management Tools

In diesem Abschnitt wird Nemoz, das über zwei Semester entwickelt wurde, mit anderen über mehrere Jahre hinaus entwickelten Audio Management Tools verglichen. Diese anderen Programme wurden anhand von 17 Kriterien getestet. Nach jeweiligen Zusammenfassungen und Screenshots der Programme, werden die Kriterien in Tabellen zusammengefasst. Es wurden folgende Programme getestet:

- **amaroK**
- **Helium Music Manager 2005**
- **iTunes**
- **Musicmatch Jukebox**
- **Winamp**
- **Windows Media Player**

Es wurden folgende Kriterien zum Vergleich bzw. zum Testen ausgewählt:

1. **Lizenz:** Gibt an, unter welcher Lizenz man diese Software erwerben kann.
2. **Plattform:** Beschreibt unter welchen Betriebssystemen dieses Programm benutzt werden kann.
3. **Sprache:** Ist die Programmiersprache mit der das Tool geschrieben ist.
4. **Plugins:** Gibt an, ob die Software pluginfähig ist.
5. **Größe b. Inst.:** Größe bei Installation sagt aus, wieviel Speicherplatz das Programm auf der Festplatte einnimmt, wenn es installiert ist.
6. **Größe i. Spei.:** Größe im Speicher meint den Speicherverbrauch, der für das Abspielen eines Liedes gebraucht wird.

7. **Verbreitungsgrad:** Dieses Kriterium zeigt, wie populär das Tool ist. Dabei wird es als „sehr populär“ bewertet, wenn es von vielen Benutzern als Audio Management Tool verwendet wird. „Populär“ meint, dass es ein gutes Tool ist, aber dieses nur unter bestimmten Voraussetzung verwendet werden kann(z.B. nur unter bestimmte Betriebsysteme wie Linux o.ä. Kriterien). Und das Stichwort „einzelne Gruppe“ steht dafür, dass das Programm nur für bestimmte Benutzerkreise vorgesehen ist (wenn es z.B. mehr als wissenschaftliches Tool vorgesehen ist).

8. **Zuverlässigkeit:** Sagt aus, wie oft das Programm im durchschnittlichen Verbrauch abstürzt. Es wird als „sehr stabil“ bewertet, wenn es fast nie zum Absturz gekommen ist. Für ein bis drei Mal wird es „stabil“ und wenn es mehr als drei Mal zusammenbricht wird es als „nicht stabil“ bezeichnet.

9. **Dateiformate:** Dies sind die Dateiformate, die das Management Tool handhaben kann.

10. **Id3-Tag Editierung:** Gibt an, ob man mit dem Tool ID3-Tags editieren kann.

11. **Antwortzeit:** Es wird für drei Ereignisse die Antwortzeiten gemessen und als „klein“, „mittel“ und „groß“ bewertet. Dabei geht es um die Funktionalitäten „Suchanfragen“ (lokal und falls unterstützt im Netz), „Öffnen“(einer Datei) und „Importieren“. Klein bedeutet, dass der Vorgang nicht mehr als 5 Sekunden dauert. Für 5 bis 10 Sekunden wird es mittel und für mehr als 10 Sekunden wird es als groß bezeichnet.

12. **Strukturierung:** Hierbei geht es um die Strukturierung der Audiodateien. Es wird getestet, ob man mit dem Tool „Playlisten“, „intelligente Playlisten“ und „Taxonomien“ erstellen kann, und ob die „Verknüpfung der Taxonomien“ möglich ist.

13. **Metadatenerfassung:** Bei diesem Kriterium geht es darum, für eine Audiodatei Metadaten zu finden, z.B. den Titel, den Interpreten, aber auch andere Metadaten wie die Produktionsfirma, das Cover des Albums, Lyrics, etc. Dabei werden die Metadaten im „Internet“(z.B. bei Amazon) im „zentralen Server“(Musik Datenbanken im Internet wie z.B. CDDB oder FreeDB) gesucht. Oder aber der „Benutzer“ stellt Metadaten selber zur Verfügung.

14. **(Klassische) Suche:** Bei der Suche geht es darum, Audiodatein zu finden, welche einer Suchanfrage entsprechen. Dabei hat man die Möglichkeiten nach „Metadaten“, mit „Benutzertags“ und „automatische Tags“ in der eigenen Datenbank oder im Internet zu suchen.

15. **Intelligente Suche:** Bei der intelligenten Suche geht es darum, jegliche Suchanfragetypen, die bei der klassischen Suche möglich sind, zu kombinieren und Audiodaten, die diese Kriterien erfüllen, zu finden.

16. **Empfehlungen:** Tools, die diese Funktion haben, können ähnliche Musikstücke zu einem gewählten Musikstück finden. Die Empfehlung kann aber nicht nur „Musikstück bezogen“ sein, sondern auch „Benutzer bezogen“. Dies bedeutet, dass das Tool gemäß einer Menge von Musikstücken bei anderen Benutzern andere Musikstücke finden kann, die ihm gefallen könnten.

17. **Netzfunktionen:** Falls das Programm Netzfunktionen hat, wird mit diesem letzten Kriterium getestet, ob man in fremden Taxonomien „browsen“ kann, oder bei anderen Musikstücke „runterladen“ oder aber auch vorher „probehören“ kann.

amaroK

amaroK[14] ist ein Musikmanager für das K Desktop Environment[15] (KDE). Veröffentlicht unter der GPL, steht es für alle vom KDE-Projekt unterstützten Plattformen kostenlos zur Verfügung.

[14] `http://amarok.kde.org`
[15] `http://www.kde.org`

amaroK verwaltet Musik in einer sogenannten *collection*. Diese ist realisiert als eine Menge von Tabellen, die unter anderem die Metadaten der Musikstücke enthalten, welche in einer SQL-artigen Datenbank abgelegt sind. Die *collection* kann im Programm durch automatisch erstellte Taxonomien, wie z.B. *Genre* → *Artist* oder *Artist* → *Album*, oder aber als flache Liste, die aber durch mindestens ein Suchkriterium eingeschränkt sein muss, betrachtet werden. Letzteres vermeidet es geschickt jemals alle Einträge gleichzeitig anzeigen zu müssen. Für die Strukturierung der eigenen Musik nach persönlichen Vorgaben werden Playlisten angeboten. Dabei wird zwischen statischen Playlisten, die vollständig durch den Benutzer erstellt werden, und intelligenten Playlisten, die Musikstücke enthalten, die einem oder mehreren Suchkriterien genügen, unterschieden. Suchkriterien können sich ausschließlich auf die Metadaten der Musikstücke beziehen. Statische Playlisten enthalten eine feste Anzahl von Musikstücken, die zum Zeitpunkt ihrer Erstellung festgelegt wird, während die Größe intelligenter Playlisten je nach momentanem Inhalt der *collection* schwanken kann.

amaroK analysiert das Hörverhalten des Benutzers, wie z.B. welche Stücke wie oft und wann zum letzten Mal abgespielt wurden, und erstellt daraus eine Bewertung (auf einer Skala von 0 bis 100) für jedes dieser Musikstück. Diese wird in Kombination mit einem zusätzlichen GUI-Element, dem sogenannten Kontext-Browser genutzt. Hier werden abhängig vom Kontext des aktuell abgespielten Musikstücks Empfehlungen für andere Musikstücke eingeblendet. Diese Empfehlungen beschränken sich allerdings auf Musikstücke des gleiche Künstlers (nach der erwähnten Bewertung absteigend sortiert) sowie andere Alben des gleichen Künstlers. Inwieweit die in amaroK vorhandene *audioscrobbler*[16]-Anbindung zur Erweiterung dieser Empfehlungsliste genutzt werden kann, ist unklar, da sich diese beim Test des Programms nicht zur Mitarbeit überreden ließ. Statische Playlisten können automatisch durch Empfehlungen ergänzt werden.

amaroK kann verschiedene Mediaframeworks zur Wiedergabe von Musik benutzt. Zu den unterstützten Frameworks gehören u.a. Xine[17] und GStreamer[18]. Unterstützte Dateiformate sind MP3, AAC und andere.

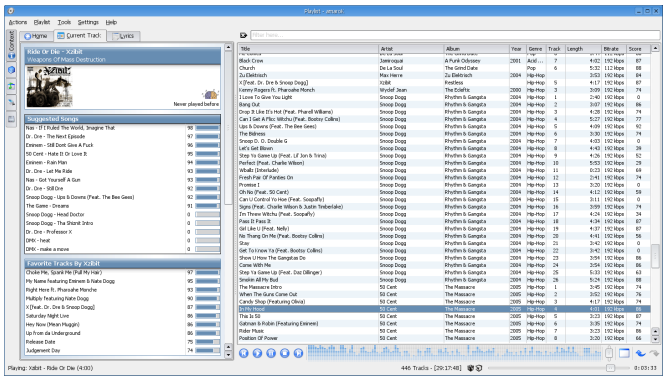


Abbildung 2.10.: Screenshot von amaroK 1.2

Helium Music Manager 2005

Der Helium Music Manager 2005[19] ist ein vielseitiges Programm, was die Verwaltung von Musikdaten angeht. Die 6,15 MB große Installationsdatei nimmt ca. 12 MB Speicherplatz auf der Festplatte ein und ist nur für Microsoft Windows 2000, XP und für Microsoft Windows Server 2003 erhältlich. Das englischesprachige Programm kann mit Hilfe eines Plugins auf Deutsch umgestellt werden. Eine

[16] http://www.last.fm
[17] http://xinehq.de
[18] http://gstreamer.freedesktop.org
[19] http://www.helium2.com/index.php

Lizenz kann nach dem Ende der 15. Benutzung des Programms(15-run evaluation Lizenz) für $35 USD erworben werden[20].

Ganz gewöhnungsbedürftig ist der Player von dem Manager platziert worden, und zwar ganz unten am Fensterrand. So wie es aussieht, ist der Player nicht das Herzstück des Programms. Auch der riesige Umfang an Verwaltungsfunktionen macht den Umgang mit dem Programm für einen Anfänger kompliziert.

Seine Stärken zeigt das Tool vor allem bei der Vielfalt der unterstützten Dateiformaten. Nach Angaben[21] soll der Music Manager MP3, Audio CD, WMA/ASF, AAc(M4A/M4P), AAC(MP4), OGG Vorbis, APE/MAC, MPC/MP+ und FLAC-Dateien unterstützen. Als Kernfunktionen sind unter anderem Database Management, Tag Editing und File Management zu erwähnen. Auch ganz interessant ist die Funktion, dass man mit Hilfe verschiedener Plugins nach Albumcovern, Lyrics, Biografien, Interpretbildern und Metadaten im Internet suchen kann. Als eine ganz nützliche Funktion hat sich der Id3-Tag Editor erwiesen, der Tags tauschen, entfernen, kopieren, importieren und exportieren kann. Man kann sogar mit dem Programm die Dateien auf der Festplatte umbennen oder aus dem Dateinamen die Id3-Tag-Informationen vervollständigen lassen.

Die Stabilität des Programms hat, nach Angaben der Benutzer[22], mit dieser Version zugenommen. Man kann zwar Playlisten mit dem Programm erstellen und diese abspielen, aber leider kann man keine Taxonomien erstellen. Auch Empfehlungen, sei es nach einem ähnlichen Musikstück oder einer Musikrichtung, werden nicht angeboten. Es werden auch keine P2P-Funktionen angeboten, sodass man diesen Manager fast als *standalone* Music Manager bezeichnen kann. Dagegen sind aber die Antwortzeiten bei Suchanfragen, beim Importieren und Öffnen einer Datei ganz klein.

Zusammengefasst ist der Helium Music Manager 2005 ein nützliches Tool mit vielen Vorteilen. Das zeigen auch die vielen Auszeichnungen[23] der verschiedenen Internetseiten und Zeitschriften. Was ihn aber von der Popularität aufhält ist sein Preis. Deswegen steigen viele Benutzer auf Freeware-Programme um.

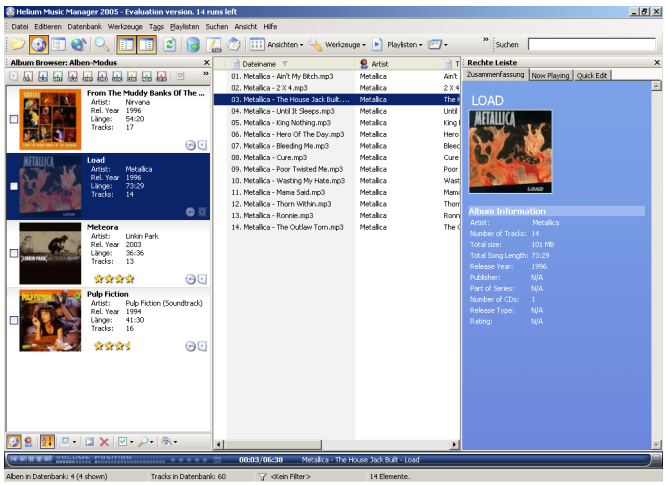


Abbildung 2.11.: Screenshot von Helium Music Manager 2005

[20] http://www.helium2.com/download.php
[21] http://www.helium2.com/index.php
[22] http://www.mpex.net/software/details/heliumae.html
[23] http://www.helium2.com/awards.php

iTunes

ITunes[24] ist der offizielle Musikmanager für Apple's „iPod" MP3-Player Serie und teilt deren beachtliche Popularität. Das Programm basiert ursprünglich auf SoundJam MP, einem relativ einfachen Musikmanager für das MacOs, welcher von Apple aufgekauft und weiterentwickelt wurde. Die vielleicht wichtigsten Merkmale von iTunes sind eine äußerst intuitive Benutzerführung, eine aufgeräumte und ästhetisch ansprechende GUI, sehr einfache Synchronisation von „iPod" MP3-Playern und die nahtlose Anbindung an den „iTunes-Musicstore", einer der größten Internet-Shops für Musik.

ITunes organisiert die Musikbibliothek des Benutzers in einer festen Taxonomie mit den Ebenen Musikrichtung, Interpret und Album. Zusätzlich ermöglicht es das Anlegen eigener (flacher) Wiedergabelisten sowie die Schnellsuche nach Interpreten, Alben, Komponisten und Titeln. Kompliziertere Suchanfragen werden durch „intelligente Wiedergabelisten" realisiert. Dies sind Datenbanksichten, die wie gewöhnliche Wiedergabelisten dargestellt werden, sich aber selbst aktualisieren.

ITunes verwendet von Haus aus das Datenformat AAC. Es basiert auf Apple's Multimediaframework Quicktime und kommt daher auch problemlos und ohne Konfigurationssarbeit mit MP3 und allen anderen von Quicktime unterstützten Dateiformaten zurecht. Für manche Anwender interessant ist die Unterstützung eines verlustfreien Codecs, der Musik auf die Hälfte ihrer Größe im PCM-Format der Audio-CD komprimieren kann. In der vorliegenden Version 4.9 werden auch Musikvideos im Quicktime-Format unterstützt. Weitere Dateiformate wie z.B. Ogg Vorbis können durch Plugins ergänzt werden.

Das Extrahieren von Musik aus Audio-CDs nach AAC oder MP3 funktioniert problemlos, da iTunes als kommerzielles Programm auch die Hintertüren kopiergeschützter CDs nutzen kann. Weiterhin lassen sich Wiedergabelisten direkt aus dem Programm heraus als MP3- oder Audio-CD brennen, sogar an eine Funktion zum Drucken einfacher Booklets und Labels wurde gedacht.

ITunes kann Peers im gleichen IP-Subnetz via IETF Zeroconf finden und zeigt diese dann wie Wiedergabelisten an. Der Benutzer kann die Musik eines entfernten Benutzern anhören, aber nicht herunterladen. Dabei wird die Musik beim anbietenden Benutzer dekodiert, verlustfrei rekodiert und verschlüsselt, bevor sie als Stream über das Netzwerk übertragen wird. Dieses aufwändige Verfahren kostet sehr viel Rechenzeit und ist wohl eher rechtlich als technisch motiviert. Inzwischen gibt es allerdings freie Serverprogramme, die sich für iTunes wie gewöhnliche Peers verhalten. Auf diese Weise kann man leicht große Musiksammlungen für iTunes-Benutzer im lokalen Netz zur Verfügung stellen.

Die Unterstützung für Metadaten ist recht vollständig, alle Tags der ID3-V.2-Spezifikation werden erkannt und können bearbeitet werden. Metadaten ungelabelter Musikstücke werden automatisch über den zentralen CDDB-Server ergänzt. Als Datenbank verwendet iTunes eine einfache, große XML-Datei, die bei jedem Start neu eingelesen wird, was zwar sehr schnell vonstatten geht, aber die effektive Größe einer Musiksammlung auf circa 150.000 Stücke begrenzt.

ITunes bietet viele weitere, zum Teil innovative Features wie z.B. Webradio, Podcasts, eine „Party-Jukebox" oder Visualisierungs-Plugins. Das Programm, welches es für Windows (2000 und aufwärts) und Mac OS X (10.3 und aufwärts), nicht aber für Linux gibt, wird aktiv weiterentwickelt und um neue Features ergänzt.

Musicmatch Jukebox

Musicmatch Jukebox wird vor allem mit MP3-Playern verkauft. Die gute Unterstützung von portablen Devices ist aufgrund der Marktausrichtung des Programms selbstverständlich. Auch dem iPod lag es früher bei, bevor es eine PC-Version von iTunes gab. Inzwischen gehört Musimatch zu Yahoo und bietet Zugriff auf Yahoo's Music Store. Musicmatch ist einmal als Freeware erhältlich, mit einem etwas abgespecktem Funktionsumfang. Volle Unterstützung (Pro-Version) erhält man für $19.99 USD. Als Dateiformat kann Musicmatch mit mp3, wav, m3u, cda, wma, mmo, und pls umgehen.

Das Programm liegt in der Version 10 vor. Ältere Versionen gibt es auch in verschiedenen Sprachen. Die

[24] `http://www.apple.com/de/itunes/`

Abbildung 2.12.: Screenshot von iTunes

aktuelle Version gibt es bis jetzt nur in Englisch. Lobenswert sind die Ansichten der Library. Sie läßt sich nach verschiedensten Kriterien ordnen. Songs werden nach Alben oder Künstlern zusammengefasst und strukturiert angezeigt. Taxonomien lassen sich nicht erstellen. Ein weiteres Feature ist der AutoDJ. Ihm gibt man einige Künstler vor und AutoDJ erstellt dann eine dazu passende Playlist aus den Songs der lokalen Library und wahlweise dem *on demand* Angebot. Diese Funktion liefert aber nicht immer überzeugende Ergebnisse. Erst recht nicht, wenn es keine bekannten Künstler sind.

Weitere Extras sind das integrierte CD-Brennprogramm und das Rip-Tool, die aber ihren vollen Funktionsumfang erst in der Pro-Version bereit stellen.

Zusätzlich gibt es noch die Premiumdienste. Mit ihnen erhält man Zugriff auf über 900.000 Songs *on demand*. Die Bedienung ist manchmal etwas eigenwillig und die Aufforderung die ProVersion zu kaufen ist nervig, aber wenn man sich daran gewöhnt hat, ist Musicmatch Jukebox 10 ein guter Allround-Manager für Musiksammlungen.

Abbildung 2.13.: Screenshot von Jukebox

Winamp

Winamp[25] ist ein weit verbreiteter Audio Player für Windows und steht in drei verschiedenen kostenlosen Versionen und einem kostenpflichtigen Version zum Download[26] bereit. Die kostenpflichtige Version kann im Vergleich zu den anderen auch noch CDs rippen und encoden. Die Lite Version enthält nur einen Basis Player, die Pro Version noch eine Medienbibliothek, die Fähigkeit Audio- und Videostreams zu empfangen, und noch einige Zusatzfunktionen. Winamp unterstützt in allen drei Versionen fast alle Codecs. Der Player kann unter anderem folgende Dateiformate abspielen: aacPlus, AAC, WMA, MP3, ... Eine vollständige Liste der unterstützen Formate findet man auf der Homepage[27] des Players.

Desweiteren gibt es mittlerweile unzählige von Plugins für Winamp, die kostenfrei im Internet verfügbar sind. Im Folgenden eine Reihe von Plugins, die im Vergleich mit Nemoz interessant sein könnten:

Abbildung 2.14.: Screenshot von Winamp

Moretunes
MoreTunes liefert Daten über den Song, der gerade gespielt wird und genau über:

- ALBUMS: Sucht nach Albums. Für jedes Album sieht man die Anzahl von Songs auf diesem Album mitsamt ihrem Titel. Gleichzeitig wird das Cover des Albums angezeigt.
- COVER: Wenn man auf Cover klickt, kann man das Cover auf der Festplatte speichern.
- REVIEWS: Wird ein Text geschrieben über die CD, den Interpreten oder das Genre.
- ID3 MANAGER: Damit kann man Id3-Tags (Title, Artist, Album, Year, Genre...) reparieren.

Wombat Share Mittels dieses Plugins kann man sich mit anderen Rechnern, auf denen das gleiche Plugin installiert ist, verbinden, und deren Musik anhören.

Virtual Playlist Manager Dieses Plugin unterstützt das komfortable Verwalten seiner Playlisten.

[25] http://www.winamp.com
[26] http://www.winamp.com/player/index.php
[27] http://www.winamp.com/player/free.php#filetype

	Lizenz	Plattform	Sprache	Größe b. Inst.(MB)	Größe i. Spei.(MB)
amaroK	Open Source	Linux,Unix, MacOSX	C++	–	–
Helium	Shareware	Windows	k.A.	ca. 12	ca. 48
iTunes	Freeware	Win.,Mac.	C++	ca. 14	ca. 35
Musicmatch	Freeware	Windows	k.A.	ca. 38	ca. 10
Nemoz	Open Source	unabhängig	Java	ca. 12	ca. 50
Winamp	Shareware	Windows	k.A.	ca. 12	ca. 20
WMP	Freeware	Windows	k.A.	–	ca. 22

Tabelle 2.1.: Allgemeiner Toolvergleich 1

Moodmixer Moodmixer ist mehr ein Programm als ein Plugin und behauptet von sich, eigenständig Playlisten erstellen zu können anhand von Kriterien, die der Benutzer auf verschiedenen Skalen einstellen kann, wie z.B. Rhythmus.

Windows Media Player

Der Windows Mediaplayer ist der Standardplayer, der mit einem Windows-Betriebssystem ausgeliefert wird. Inzwischen wird auch eine Windowsversion ohne den Player angeboten. Dieser kann aber nachträglich von Microsoft runtergeladen (ca. 13 MB) und nachinstalliert werden. Da dieser Player einer der wenigen ist, der Digital Rights Management unterstützt, ist er sehr verbreitet und wird von vielen Online Musikshops vorausgesetzt.

Standardmäßig werden von WMP nur die Audioformate MP3 und WMA unterstützt. Andere Formate können per Plugins hinzugefügt werden. Der Player kann in drei Modi betrieben werden. Im „Vollmodus“ verwaltet man seine Musikdateien und stellt die Playlisten zusammen. Im „Designmodus“ sind nur die Navigationssymbole und die Fortschrittsanzeige sichtbar. Es bleibt also nur die reine Playerfunktionalität übrig. In dem „Miniplayermodus“ sind schließlich nur die Bedienelemente in dem Systemtray sichtbar und können so aus jedem anderen Programm aufgerufen werden.

Um Dateien zu der Library hinzufügen, kann man den gesamten Rechner scannen lassen, oder man gibt einfach ein Verzeichnis an, dass importiert werden soll. Alternativ kann man auch Verzeichnisse überwachen lassen. Bei Änderung des Inhalts kann dieser sofort in der Library hinzugefügt werden. Das Hinzufügen selber ist recht schnell. Von jeder Datei werden die Tags ausgelesen, analysiert und diese Datei dann automatisch an die entsprechende Stelle in die Datenbank eingefügt. Bei dem langsamen Modus wird die komplette Datei durchsucht, um die optimale Lautstärke für die Wiedergabe zu finden. Die Anordnung der Daten in der Library ist vorgegeben. Es sind einige Taxonomien wie Album, Interpret, Genre und Bewertung vorgegeben. Eigene Taxonmien kann man leider nicht erstellen. Durch das Editieren des MP3 Tags kann man aber schon in einer gewissen Weise den Ort der Datei in der Datenbank festlegen. Zu jedem Song werden zusätzlich Statistiken und das Nutzerverhalten gespeichert. Aus diesen Statistiken werden dann automatische Playlisten erstellt, zum Beispiel mit Songs, die man gerne am Wochenende hört, oder die man noch nie gehört hat. Man kann die Playlisten aber auch manuell erstellen und so die Songreihenfolge selbst bestimmen.

Der Mediaplayer arbeitet mit vielen Onlineshops zusammen. Eine Internetverbindung vorausgesetzt, kann man zu den Songs zusätzliche Information oder das Albumcover suchen. Leider hängt die Qualität der Ergebnisse und die Suchgeschwindigkeit vom jeweiligen Shop ab. Ist die Suche erfolgreich, findet man meistens auch andere Alben des Interpreten und kann in diese reinhören oder sie online kaufen.

Insgesamt macht der Player einen ziemlich soliden Eindruck. Durch den integrierten Webbrowser ist der Speicherverbrauch zwar etwas hoch, dafür kann man aber bequem die Onlineshops durchsuchen. Das einzige große Manko des WMP sind die wenigen unterstützten Formate, so dass er für *Nicht-MP3*-Nutzer eine ziemlich schlechte Wahl ist.

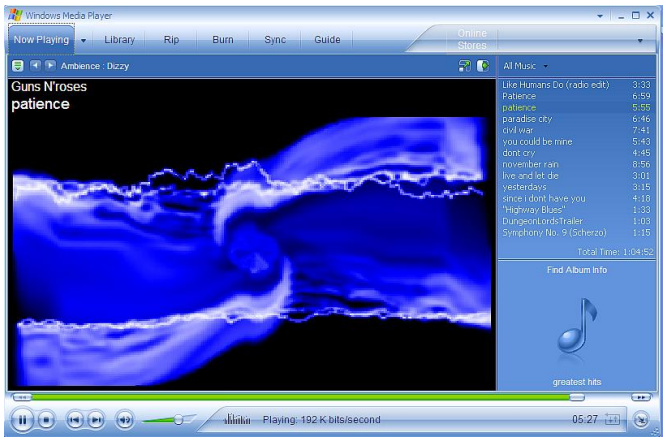


Abbildung 2.15.: Screenshot von Windows Media Player

	Plugins	Verbreitungsgrad	Zuverlässigkeit	Dateiformate
amaroK	ja	populär	sehr stabil	aac, mp3
Helium	ja	populär	sehr stabil	s. Beschreibung
iTunes	ja	sehr populär	sehr stabil	aac, mp3
Musicmatch	ja	populär	stabil	s. Beschreibung
Nemoz	ja	einzelne Gruppe	stabil	mp3
Winamp	ja	populär	sehr stabil	s. Beschreibung
WMP	ja	sehr populär	stabil	mp3, wma

Tabelle 2.2.: Allgemeiner Toolvergleich 2

	Id3-Editierung	Antwortzeiten			Strukturierung				Metadatenerfassung		
		Suchanfrage	Öffnen	Importieren	Playlisten	Intell. Playlisten	Taxonomien	Verkn. Taxonomien	Internet	Zentr. Server	Benutzer
amaroK	ja	klein/–	klein	klein	ja	ja	nein	nein	ja	ja	ja
Helium	ja	klein/–	klein	klein	ja	nein	nein	nein	ja	ja	ja
iTunes	ja	klein/klein	klein	klein	ja	ja	nein	nein	nein	ja	ja
Musikmatch	ja	klein/–	klein	klein	ja	ja	nein	nein	ja	ja	ja
Nemoz	nein	klein/klein	klein	mittel	ja	nein	ja	ja	nein	nein	nein
Winamp	ja	klein	klein	klein	ja	ja	nein	nein	ja	ja	nein
WMP	ja	klein/mittel	klein	klein	ja	ja	nein	nein	ja	nein	ja

Tabelle 2.3.: Funktionsvergleich einzelner Tools 1

	Suche			Intell. Suche	Empfehlungen		Netzfunktionen		
	Metadaten	Auto. Tags	Benutzertags		Musik bez.	Benutzer bez.	Browsen	Runterladen	Probehören
amaroK	ja	ja	ja	nein	ja	ja	nein	nein	nein
Helium	ja	ja	nein	ja	nein	nein	nein	nein	nein
iTunes	ja	ja	nein	ja	ja	nein	ja	nein	ja
Musikmatch	ja	nein	nein	nein	nein	nein	on demand	on demand	nein
Nemoz	ja	nein	nein	ja	ja	nein	ja	ja	ja
Winamp	ja	nein	nein	nein	ja	nein	nein	nein	nein
WMP	ja	nein	nein	nein	nein	nein	nein	ja	ja

Tabelle 2.4.: Funktionsvergleich einzelner Tools 2

Wie schon erwähnt, Nemoz wurde in einer Projektgruppe in einem akademischen Jahr entwickelt. Da die Zeit begrenzt war, konnten nicht alle für ein Audiomanagementtool wichtige Funktionen implementiert werden. Z.B. unterstüzt Nemoz nicht Id3-Tag-Editierung, was aber nicht heißen soll, dass es so bleiben wird. Da das Projekt ein Open Source Projekt war und Nemoz auch pluginfähig ist, kann Nemoz jeder Zeit um solche Funktionen erweitert werden. Was Nemoz von den anderen Tools unterscheidet, ist seine „intelligenten“ Funktionen. Auch um solche Funktionen wird Nemoz in der Zukunft erweitert. Mit seinen jetzigen Zustand ist er aber schon als Audiomanagementtool einsetzbar. In Kapitel 6 kann man einen Ausschnitt finden was für Nemoz in der Zukunft geplant ist.

2.5. Softwareentwicklung und Projektmanagement

2.5.1. Werkzeuge für die gemeinsame Softwareentwicklung

Die vorliegende Ausarbeitung liefert eine kurze Übersicht, welche Tools für ein professionelles und effizientes Arbeiten bei selbst kleinen Software-Projekten unverzichtbar sind. Insbesondere wird demonstriert, wie sich im Java-Umfeld eine vollständig auf Opensource-Technologien basierende Entwicklungsumgebung aufbauen und betreiben lässt.

Motivation

Zur Entwicklung guter und wartbarer Software ist ein gewisser Grundbestand an Hilfs- und Unterstützungswerkzeugen absolut unerläßlich. Die Erfahrung zeigt, daß winzige oder kurzlebige Projekte quasi nicht existieren. Selbst bei nur wenig beteiligten Programmierern und einer kurzen Entwicklungsdauer erwarten die Anwender über einen mittel- bis längerfristigen Zeitraum eine adäquate Wartung und Pflege. Dies läßt sich ohne geeignete Werkzeuge gar nicht oder nur mit unverhältnismäßig hohem Aufwand erreichen.

In der Praxis wird aus Kostengründen oft auf die Anschaffung und Nutzung der entsprechenden Entwicklungshilfen verzichtet. Somit gilt es, kostengünstige, oder sogar kostenfreie, Lösungen, etwa aus dem Opensource Bereich, zu finden, die auf Dauer im professionellen Entwicklungsumfeld eingesetzt werden können.

Entwicklungswerkzeuge

Sourcecode Management System Ein Sourcecode Management System ist das mit Abstand wichtigste Werkzeug bei der Softwareentwicklung und bietet die folgenden Möglichkeiten:

Zentrale Ablage von Quellcode Dies gewährleistet, daß alle Mitarbeiter mit den gleichen Quelldateien arbeiten. Eine zentrale Ablage vereinfacht darüberhinaus erheblich die Sicherung des Codebestandes auf Backup-Medien und erlaubt die Einrichtung einer Zugriffskontrolle. Zusätzlich sollte man in der Lage sein, zusammengehörige Dateien logisch zu gruppieren, etwa in Projekte und Subprojekte. Zur Bearbeitung von Code werden lokale Kopien auf den Entwicklungsrechnern erzeugt, die nach Änderung wieder in die zentrale Ablage übernommen werden.

Speichern der Änderungsgeschichte jeder Datei Jede an einer Datei vorgenommene Änderung wird unter Angabe des Zeitpunktes, des Nutzers und des Änderungsgrundes protokolliert. Außerdem muß es möglich sein, bei Bedarf jederzeit wieder eine vergangene Version einer Datei zu erhalten.

Versionsinformation verwalten Dieses Feature gestattet es, eine bestimmte Markierung an allen Dateien eines Projekts anzubringen, um zum Beispiel einen bestimmten Releasestand zu markieren.

Parallele Entwicklung ermöglichen Durch das Arbeiten mit lokalen Kopien kann es zu Synchronisationsproblemen beim Zurückspielen der Änderungen in den gemeinsamen Codebestand kommen. Die Sourcecode Verwaltung muß Änderungen, die sich überschneiden, erkennen und entsprechend darauf reagieren.

Open Source Tool: CVS Concurrent Version System Das Concurrent Version System (CVS) ist eines der populärsten Sourcecode Management Systeme überhaupt. Die hohe Verbreitung rührt neben seiner Zuverlässigkeit daher, daß es seit seiner Entstehung Mitte der 80er Jahre kostenfrei für nahezu jedes Betriebssystem verfügbar ist.

Neben der reinen Kommandozeilenversion existieren auch eine Vielzahl graphischer Ergänzungen zu CVS, die ein GUI-basiertes Arbeiten mit Sourcecode Archiven ermöglichen. Außerdem bietet so gut wie jede moderne integrierte Entwicklungsumgebung (IDE) mittlerweile eine eingebaute Unterstützung für CVS an.

Build Management Tool Ein weiterer Grundpfeiler in der Softwareentwicklung ist eine solide Buildumgebung, die das einfache und konsistente Übersetzen eines Projektes ermöglicht. Die nachfolgenden Features erweisen sich in diesem Zusammenhang als nützlich:

Inkrementelle und schnelle Builds Um das tägliche Arbeiten möglichst effizient gestalten zu können, sollte das Build Management Tool in der Lage sein, das Projekt schnell und inkrementell zu übersetzen. Bestimmte Arbeitsweisen, wie Extreme Programming (XP) oder Refactoring, setzen dies sogar als Grundbedingung voraus.

Automatisierbarkeit Neben dem eigentlichen Übersetzen des Quellcodes sollte es eine gute Buildumgebung ermöglichen, weitere Aufgaben automatisiert auszuführen. Hierzu gehören zum Beispiel der Abgleich mit dem gemeinsamen Sourcecode Archiv oder das Durchführen von Testläufen. Kommandozeilentools, wie 'make' aus dem C Umfeld, sind dabei häufig deutlich flexibler als graphische Umgebungen.

Möglichst portabel In der Praxis kann es sich als nützlich erweisen, sich nicht exklusiv an die Tools bestimmter Entwicklungsumgebungen zu binden oder sich zu früh auf bestimmte Betriebssysteme festzulegen.

Apache Ant Bei Apache Ant handelt es sich um ein Java basiertes Buildtool für Java Projekte. Es lässt sich entfernt mit Make vergleichen, ist jedoch vor allem durch seine XML Builddateien ungleich einfacher in der Handhabung.

Ant läßt sich in fast jede populäre Entwicklungsumgebung integrieren und ist bei Bedarf auch beliebig erweiterbar. Die Tatsache, daß mittlerweile Sun selbst für viele Java Beispiele Ant verwendet, zeigt deutlich, daß es zur Quasi-Standard Buildumgebung für Java geworden ist.

Testing Framework Testen von Software ist ein unverzichtbarer Bestandteil eines soliden Entwicklungsprozesses. Obwohl je nach Projekt die notwendigen Tests stark variieren können, sollten die folgenden Punkte bei allen Tests sichergestellt sein:

Paralleles Entwickeln und Testen Für qualitativ hochwertige Software ist es unverzichtbar, Fehler während der Entwicklung möglichst früh zu erkennen. Dadurch ergibt sich die Notwendigkeit bereits während der Entwicklung zu testen und nicht erst nachdem die Entwicklung (scheinbar) abgeschlossen ist. Bestimmte Entwicklungstechniken wie Extreme Programming (XP) und Refactoring setzten ein paralleles Entwickeln und Testen sogar als unabdingar voraus.

Möglichst automatisiertes Testen Manuelles Testen ist sehr arbeits- und zeitintensiv und zusätzlich auch sehr fehleranfällig. Folglich ist es von großem Vorteil, sowohl für die Testaufwände als auch für die Menge entdeckbarer Fehler, wenn der Großteil der Software automatisiert getestet werden kann.

Open Source Tool: JUnit Bei JUnit handelt es sich um ein Open Source Unit Testing Framework, welches das automatisierte Testen von Java Software ermöglicht. Mittlerweile existieren schon einige Ableger von JUnit, die sich schwerpunktmäßig auf bestimmte Umgebungen spezialisieren, etwa HttpUnit zum Testen von HTML basierten Oberflächen. Wie bereits bei dem zuvor erwähnten Ant, bieten die meisten Java Entwicklungsumgebungen eine graphische Integration von JUnit an.

Integrierte Entwicklungsumgebung (IDE) Für ein produktives Entwicklungsumfeld ist es notwendig, die bereits vorgestellten Tools effizient miteinander zu verbinden. Hierzu werden in der Regel integrierte Entwicklungsumgebungen, sogenannte IDEs, verwendet, die dem Entwickler alle notwendigen Werkzeuge gebündelt zur Verfügung stellen. Folgende Merkmale sollte eine modernen IDE besitzen:

Leistungsfähiger Editor Die meiste Zeit der Entwicklung besteht aus Erstellen und Editieren von Quellcodedateien. Entsprechend sollte die IDE diese Arbeit so komfortabel wie möglich gestalten. Hierzu gehören zum Beispiel Features wie automatische Vervollständigung, Formatierung und sprachbezogene Darstellung des Quellcodes.

Gute Projektübersicht Die IDE sollte eine möglichst gute Übersicht über die gesamte Projektstruktur bieten und umfangreiche Navigationsmöglichkeiten vorsehen. Dies beinhaltet etwa eine Klassenhierarchie bei objektorientierten Sprachen sowie kontextbezogene Suchmöglichkeiten.

Integration weiterer Tools Die IDE sollte eine einfache Schnittstelle bieten, um weitere Tools selbst erstellen, oder bereits bestehende anbinden zu können.

Open Source Tool: Eclipse Eclipse ist eine erweiterbare, auf Java basierende IDE, die das komfortable Entwicklen mit Java und anderen Sprachen ermöglicht. Sie zeichnet sich neben ihren eigenen Features, erwähnt sei hier zum Beispiel die hervorragende Refactoring Unterstützung, durch einen starken Integrationscharakter aus, der das Einbinden externer Tools ermöglicht. Insbesondere können alle zuvor erwähnten Open Source Tools integriert eingesetzt werden.

Sourceforge

SourceForge ist ein gemeinschaftliches Software-Entwicklungsmanagementsystem, das zur Entwicklung von Open Source-Programmen benutzt wird.

Fazit

Zusammenfassend lässt sich sagen, daß es problemlos möglich ist, eine vollständige Java Entwicklungslandschaft basierend auf Open Source Tools zu schaffen. Der Verbreitungsgrad der vorgestellten Werkzeuge spricht sowohl für ihre Qualität, als auch für ihre praktische Anwendbarkeit, was in zahlreichen kleinen, wie großen Projekten bestätigt werden konnte.

2.5.2. Softwareentwicklungsparadigmen und Vorgehensweisen

Einleitung

Dieser Abschnitt bietet einen kleinen Überblick über Softwareentwicklungsparadigmen und Vorgehensweisen. Im Einzelnen werden einige Vorgehensmodelle vorgestellt, darunter z.B. das klassische Wasserfallmodell, aber auch moderne Verfahren, wie z.B. Extrem Programming.

Anschließend erläutere ich wie eine Dokumentation aufgebaut werden sollte, was hinein gehört, wann sie geschrieben werden sollte und warum sie so nützlich ist.

Zum Abschluß befasse ich mich mit den Methoden des Softwaretest, einem nicht zu vernachlässigendem Punkt der Softwareentwicklung.

Vorgehensmodelle

Das Wasserfallmodell Das erste Vorgehensmodell wurde 1970 von Winston Royce entwickelt. Er orientierte sich dabei an anderen ingenieurwissenschaftlichen Modellen. Es wird oft auch als „software life cycle“ bezeichnet. Der Prozeß der Softwareentwicklung wird hierbei in mehrere Phasen eingeteilt. Das Ergebnis einer jeden Phase ist ein Bericht. Die folgende Phase sollte erst beginnen, wenn die vorhergehende beendet ist. In der Praxis überlappen sich die Phasen allerdings. Die Softwareentwicklung ist jedoch kein einfacher linearer Prozess, wie man bei der Betrachtung des Modells vielleicht annehmen könnte. Vielmehr tauchen in den Phasen immer wieder Probleme auf, sodass vorangegangene Ergebnisse revidiert werden müssen. Demnach ist also eine wiederholte Iteration der Entwicklungsschritte notwendig. Eben diese Wiederholungen sind natürlich teuer und deshalb fixiert man nach ein paar Wiederholungen die Ergebnisse und versucht neu auftauchende Probleme zu umgehen. Dies führt dazu, dass die Software nicht genau das leistet, was man von ihr erwartet, dass Designfehler auftreten und eine unsaubere Programmierung entsteht.

Abbildung 2.16 visualisiert den genauen Aufbau der einzelnen Phasen und zeigt die Verbindungen zwischen ihnen.

Vorteile: Das Wasserfallmodell ist übersichtlich und leicht zu verwalten.

Nachteile: Es ist relativ unflexibel. Das Projekt wird strikt in die bestimmten Stufen eingeteilt. Verpflichtungen müssen früh eingegangen werden. Daher ist es schwierig nachträglich Änderungen einzubauen.

Anwendung: Das Wasserfallmodell sollte daher nur genutzt werden, wenn die Anforderungen genau und detailliert erörtert wurden. Noch heute werden Modelle genutzt, die auf dem Wasserfallmodell beruhen, gerade dann, wenn es sich nur um einen Teil eines größeren Projektes handelt.

Spiral Modell Statt den Software Prozess als sequentielle Abfolge von Aktivitäten mit Rückwirkungen darzustellen, wird der Prozess hier als Spirale angesehen. Jede Schleife repräsentiert eine Phase des Software Prozesses.

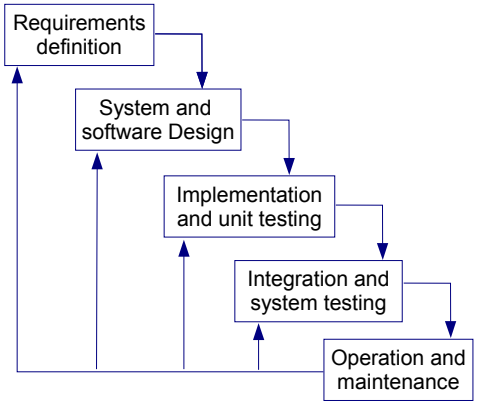


Abbildung 2.16.: Das Wasserfallmodell ([Som01] S. 45)

Jede Schleife der Spirale unterteilt sich dabei in vier Abschnitte:

1. Objective setting
2. Risk assessment and reduction
3. Development and validation
4. Planning

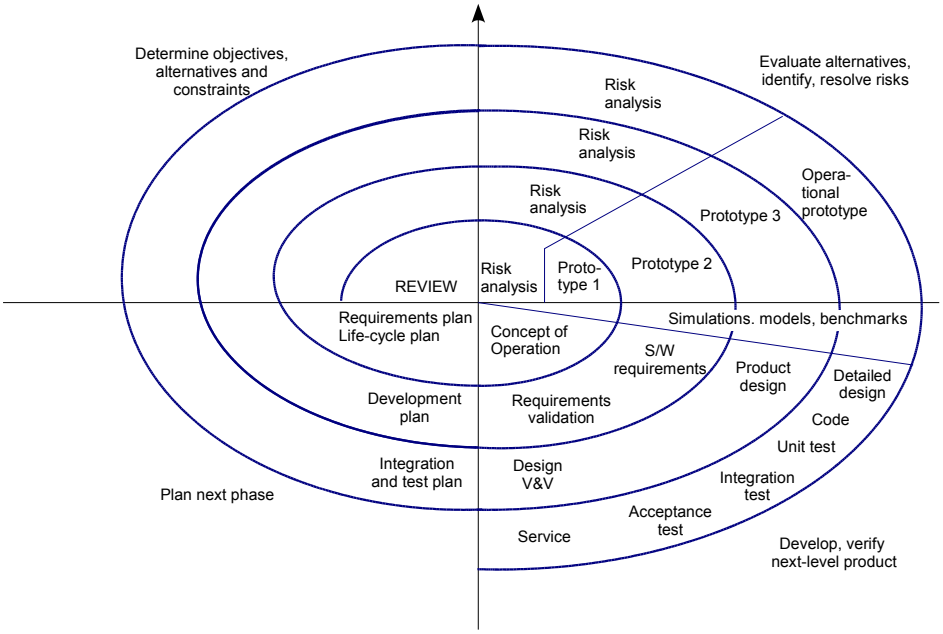


Abbildung 2.17.: Das Spiralmodell ([Som01] S. 54)

Die Abbildung 2.17 beschreibt den genauen Ablauf dieses Verfahrens und zeigt deutlich die Wiederholungen der einzelnen Phasen.

Vorteile: Die Risikoanalyse wird schon fest in die Entwicklung einbezogen. Es gibt keine fest eingeteilten Phasen, wie z.B. Spezifikation oder Design. Das Spiralmodell ist somit flexibler als das Wasserfallmodell. Außerdem kann das Spiralmodell andere Vorgehensmodelle umfassen. So kann z.B. ein Wasserfall Modell für einzelne Phasen benutzt werden.

Evolutionäres Modell Ausgangspunkt ist hierbei nicht wie bei anderen Modellen die komplette Systemanforderung, sondern nur eine Kern- oder Mussanforderung. Diese definiert den Produktkern, welcher anschließend entworfen und implementiert wird. Die erste Version des Systems, die sog. Nullversion wird dann an den Auftraggeber ausgeliefert. Der Auftraggeber sammelt nun Erfahrungen mit dieser Nullversion und entwickelt daraus weitere Produktanforderungen und Verbesserungen. Die Nullversion wird nun gemäß den Vorstellungen des Auftraggebers weiterentwickelt. Diese Weiterentwicklung ersetzt die Nullversion, woraufhin neue Anforderungen durch den Einsatz dieser neuen Version entwickelt werden und es beginnt wieder von vorn. Es handelt sich also um ein endlos iteratives Modell. Die Entwicklung erfolgt stufenweise, gesteuert durch die Erfahrung aus den Vorgängerversionen. Pflegeaktivitäten werden einfach als normaler Entwicklungsschritt betrachtet.

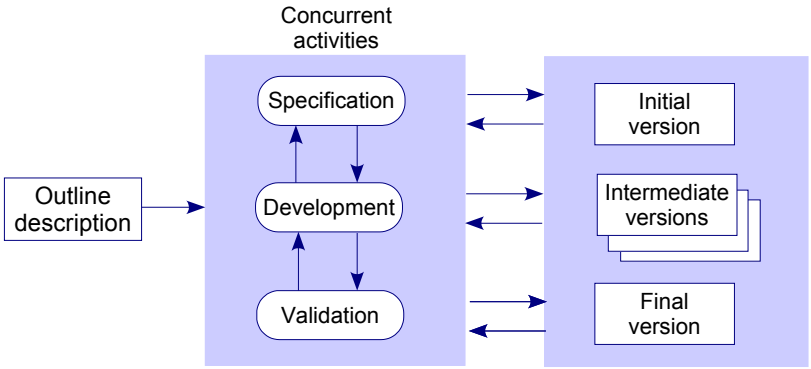


Abbildung 2.18.: Das Evolutionäre Modell ([Som01] S. 47)

Vorteile: In kurzen Zeitabständen entstehen lauffähige Produkte. Der frühzeitige Einsatz einer eingeschränkten Version gibt Einblick über die Auswirkungen des Produkteinsatzes auf die Arbeitsabläufe. Diese Erfahrung kann dann in der nächsten Version eingesetzt werden. Durch die kleinen Arbeitsschritte in überschaubarer Größe ist es möglich, die Richtung der Entwicklung Schritt für Schritt zu beeinflussen und zu korrigieren. Die Entwicklung ist nicht auf einen in weiter Ferne gesetzten Endabgabetermin ausgerichtet, sondern auf viele, nahe, einsatzfähige Zwischenergebnisse.

Nachteile: Wurden in der Nullversion Kernanforderungen nicht beachtet, muss möglicherweise in späteren Versionen die gesamte Systemarchitektur überarbeitet und geändert werden. Durch kontinuierliche Änderungen und Richtungswechsel sind so entwickelte Systeme oft nur schwach strukturiert. Es besteht die Gefahr, dass die Nullversion nicht flexibel genug ist, um sich an ungeplante Evolutionspfade anzupassen. Der Entwicklungsprozess kann nicht, wie z.B. beim Wasserfallmodell, durch Berichte nach jeder Phase verfolgt werden, da es zu teuer wäre, für jede Version wieder neue Berichte zu erstellen.

Anwendung: Das Modell ist gut geeignet, wenn der Auftraggeber seine Anforderungen noch nicht vollständig überblicken kann: „I can't tell you what I want, but I'll know it when I see it." Für kleine (weniger als 100.000 Codezeilen) und mittlere (bis zu 500.000 Codezeilen) Systeme mit einer relativ kurzen Lebensdauer ist dieses Modell sehr gut geeignet. In großen, lang eingesetzten Systemen ist die fehlende Struktur aber ein zu großes Problem, als dass es effektiv eingesetzt werden kann. Für solche Systeme empfiehlt Ian Sommerville[Som01] eine Kombination der Vorteile des Wasserfallmodells und der evolutionären Entwicklung.

Inkrementelles Modell Dieses Modell vermeidet die Nachteile des Evolutionären Modells. Die Anforderungen werden zu Beginn möglichst vollständig erfasst und modelliert. Weitere Schritte erfolgen analog zum evolutionären Modell. Der Auftraggeber vergibt Prioritäten für die einzelnen Funktionen des Systems und die Entwicklung orientiert sich an diesen Prioritäten. Es wird also zunächst wieder nur ein Teil des Systems implementiert und in Folgeschritten immer weitere Teile. Durch den anfänglichen Entwurf ist nun sichergestellt, dass neue Teile zusammenpassen, die Vorteile des evolutionären Modells bleiben aber erhalten. So erhält der Auftraggeber z.B. auch hier schnell ein einsatzfähiges Produkt, mit dem er Erfahrungen sammeln kann.

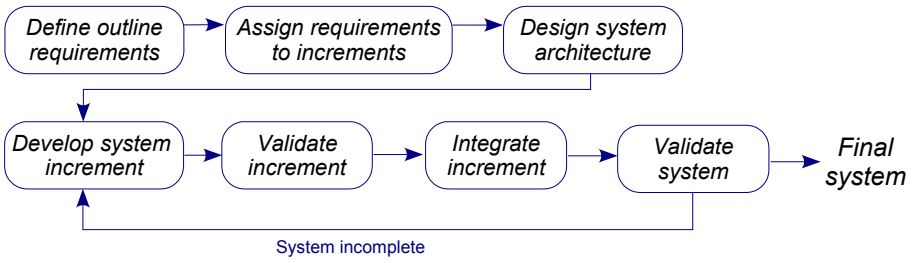


Abbildung 2.19.: Das Inkrementelle Modell ([Som01] S. 52)

Nachteile: Da die Anforderungen zunächst nur allgemein besprochen werden, kann es zu Schwierigkeiten kommen, wenn z.B. verschiedene Teile des Systems eine Funktion benutzen und diese nicht detailliert genug festgelegt war, sodass es zu Problemen mit den Schnittstellen kommen kann.

Anwendung: Für relativ kleine (bis 20.000 Codezeilen) Systeme geeignete.

Extreme Programming Extreme Programming [28] (XP) ist eine Weiterentwicklung des Inkrementellen Modells, die Kent Beck [BF01] entwickelte. XP verbessert ein Software Projekt durch vier Mittel: Kommunikation, Einfachheit, Feedback und Mut.

Die Grund *Prinzipien* sind dabei:

- Pair Programming

  Zwei Programmierer teilen sich eine Tastatur und einen Monitor, einer schreibt, der andere kontrolliert und denkt mit.

- Integration der einzelnen Komponenten zu einem lauffähigen Gesamtsystem in kurzen Zeitabständen

- Test First Development

  Es werden erst die Unit-Tests geschrieben, dann erst die eigentliche Funktionalität. Die Tests werden nach jedem Programmierschritt ausgeführt und liefern Rückmeldung über den Entwicklungsstand.

- Enge Einbeziehung des Kunden

  Der Kunde gibt das Iterationsziel vor und hat sofort die Möglichkeit das Programm zu testen und zu prüfen.

- 40 Stunden Woche

  Überstunden mindern die Freude an der Arbeit und somit auch die Qualität.

[28] Extremeprogramming `http://www.extremeprogramming.org/`
Xprogramming `http://www.xprogramming.com/`

- Keine exponentielle Steigerung, sondern gleich bleibende Änderungskosten

Anwendung: XP wurde entwickelt, um auf Änderungen der Anforderungen zu reagieren. Der Auftraggeber hat, ähnlich wie beim Evolutionären Modell, keine klare Idee, was das System leisten soll. Die Funktionalität des Systems muss alle paar Monate geändert werden. XP ist für kleine Gruppen zwischen 2 und 12 Programmierern geeignet.

Dokumentation

Eine gute Dokumentation sollte nach [Bal00] folgende Angaben enthalten:

- Kurzbeschreibung des Programms,
- Verwaltungsinformationen,
- Kommentierung des Quellcodes

Die ersten beiden Punkte können hierbei in einem Programmvorspann zusammengefasst werden:

- Programmname: Name, der das Programm genau beschreibt.
- Aufgabe: Beschreibung des Programms einschließlich der Angabe, ob es sich um ein GUI-, ein Fachkonzept- oder ein Datenhaltungsprogramm bzw. eine entsprechende Klasse (bei OOP) handelt.
- Zeit- und Speicherkomplexität des Programms
- Name des Programmautors, bzw. der Autoren
- Versionsnummer und Datum

Die Versionsnummer besteht dabei aus zwei Teilen: Der Release Nummer und der Level Nummer. Die Release Nummer steht, getrennt durch einen Punkt, vor der Level-Nummer (maximal zweistellig). Vor der Release-Nummer steht ein V.

Die Level-Nummer wird jeweils um eins erhöht, wenn eine kleine Änderung vorgenommen wurde. Bei größeren Änderungen oder Erweiterungen wird die Release-Nummer um eins erhöht, wobei gleichzeitig die Level-Nummer auf Null gesetzt wird. Ein erstmals fertiggestelltes Programm sollte die Versionsnummer 1.0 erhalten. Beginnt man mit der Entwicklung, dann sollte man mit der Zählung bei 0.1 beginnen.

Es gibt verschiedene Gründe warum die Dokumentation ein integraler Bestandteil der Softwareentwicklung sein sollte:

- Bei einer Nachdokumentation am Ende der Code-Erstellung sind wichtige Informationen, die während der Entwicklung angefallen sind, oft nicht mehr vorhanden.
- Entwicklungsentscheidungen müssen dokumentiert werden, um bei Modifikationen und Neuentwicklungen bereits gemachte Erfahrungen auswerten zu können.
- Der Aufwand für die Dokumentation wird reduziert, wenn zu dem Zeitpunkt, an dem die Information anfällt von demjenigen, der sie erzeugt oder verarbeitet, auch dokumentiert wird.

Vorteile der integrierten Dokumentation: Sie reduziert den Aufwand zur Dokumentenerstellung und stellt sicher, dass keine Informationen verloren gehen. Außerdem wird so die rechtzeitige Verfügbarkeit der Dokumentation garantiert.

Eine gute Dokumentation bildet die Voraussetzung für die leichte Einarbeitung in ein Produkt bei Personalwechsel oder durch neue Mitarbeiter, sowie gute Wartbarkeit des Produkts.

In [McC93] findet man auf Seite 456f eine übersichtliche Checkliste, die bei gewissenhafter Kontrolle einen gut strukturierten und nachvollziehbaren Programmcode ermöglicht. Zudem haben wir ein paar Beispiel-Klassen entworfen in denen die Programmierstandards vorgeführt wurden. Im wesentlichen orientierten wir uns dabei an den „Code Conventions for the Java Programming Language“[29].

Softwaretest

Im folgenden Abschnitt stelle ich kurz die gängigsten Softwaretestmethoden vor.

Genereller Test Prozess Man beginnt mit Tests von einzelnen Programmteilen, wie z.B. Funktionen oder Objekte. Ist dieser Test abgeschlossen, werden die einzelnen Teile in Subsysteme und Systeme integriert und die Interaktion der Teile wird in der Umgebung getestet. Testen ist i.d.R. ein intuitives Vorgehen, da man nicht die Zeit hat für jeden Teil eines Softwaresystems eine ausführliche Spezifikation zu schreiben. Testen basiert also auf dem intuitiven Verständnis davon wie sich der Programmteil verhalten sollte. Der Integrationstest sollte allerdings schon auf einer genauen Systemspezifikation basieren.

Defect testing Defect testing versucht verborgene Fehler in einem Softwaresystem zu finden, bevor dieses ausgeliefert wird. Ein erfolgreicher Defekt Test ist ein Test, der das System dazu bringt inkorrekt zu antworten und somit einen Defekt entdeckt. Man kann aber nicht alle möglichen Zustände eines Programmes testen, da der Aufwand viel zu hoch wäre. Vielmehr entwickelt man spezielle Testfälle, z.B. alle möglichen Programmbefehle, oder alle Befehle, die über ein Menü zu erreichen sind, und auch mögliche Kombinationen, z.B. bestimmte Formatierungen kombiniert mit bestimmten Menübefehlen usw.

Black-box testing Der Tester gibt dem System bestimmte Eingaben und erwartet vom System eine korrekte Antwort. Ist dies nicht der Fall, ist der Test erfolgreich und man hat einen Fehler gefunden. Das Hauptproblem ist hierbei natürlich Eingaben zu wählen, die mit hoher Wahrscheinlichkeit zur Menge der Fehler hervorrufenden Eingaben gehören.

Äquivalenzklassen Die Eingabedaten eines Programmes können normalerweise in eine bestimmte Anzahl von Klassen unterteilt werden. Für jedes Mitglied einer Klasse verhält sich das Programm gleich. Es ist also nur noch nötig jeweils ein Mitglied einer Klasse zu testen. Mögliche Klassen sind z.B. alle positiven Zahlen, alle Negativen, alle Strings ohne Leerzeichen etc.

White-Box testing Kann man Tests aufgrund des Wissens über die Softwarestruktur und Implementierung ableiten, so sind Strukturtests, auch White-Box testing genannt, angebracht. Diese Art des Testens ist für kleine Programme oder Subroutinen geeignet. Der Tester analysiert hierbei den Code und leitet mit Hilfe seines Wissens über die Struktur der Komponente Testdaten her.

Integrationstests Sind die einzelnen Komponenten getestet, müssen sie zusammengefügt werden und diese neue Komponente muss wieder getestet werden, da bei der Interaktion der einzelnen Komponenten untereinander Fehler auftreten können. Das Hauptproblem des Integrationstests ist die Fehlerstelle zu lokalisieren. Hat man ein großes System aus vielen Komponenten zusammen gesetzt, so ist diese Suche sehr zeitaufwändig. Daher sollte man beim Zusammensetzten und Testen des gesamten Systems inkrementell vorgehen. Man testet also zunächst einen kleinen Teil. Funktioniert alles, kombiniert man dies mit einer weiteren Komponente und testet diese wieder usw.

Top-Down testing Erst werden die High-Level Komponenten eines Systems integriert und getestet. Top-down Tests entdecken leichter Fehler in der Systemarchitektur.

[29] `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`

Bottom-Up testing Zunächst werden die Low-Level Komponenten integriert und getestet, dann die High-Level Komponenten. Bei Bottom-up werden High-Level Designfehler erst spät erkannt und somit sind sie weitaus teurer zu beheben.

Stress Testing Stress Testing geht an die Grenzen des Systems und darüber hinaus, um Fehler zu provozieren, die bei normaler Nutzung nicht auftreten würden. Ist ein System z.B. darauf ausgelegt 100 Transaktionen in der Sekunde durchzuführen, so geht Stress Testing darüber hinaus und testet, ab wann das System Fehler macht. Das System sollte selbst unter Stress keine Daten verlieren oder sonstige schwerwiegende Fehler machen. Stress Testing ist gerade für verteilte Systeme von besonderer Wichtigkeit.

2.5.3. Testen jenseits von JUnit

Um umfangreiche Anwendungen adäquat zu testen, reicht JUnit nicht aus. Deshalb werden im Folgenden verschiedene Erweiterungen zu JUnit und weitergehende Konzepte vorgestellt, mit denen ein umfassenderes Testen möglich ist. Ein weiteres mächtiges Werkzeug sind Profiler. Mit Profilern ist es möglich, ein Programm während der Laufzeit genau unter die Lupe zu nehmen. Am Ende dieses Abschnittes wird noch das *framework for integrated tests* (FIT) vorgestellt. Dieses Framework dient der Automatisierung von Akzeptanztests. Mit diesen Tests soll es dem Auftraggeber oder Kunden ermöglicht werden, ganz einfach eigene Tests zu formulieren.

Erweiterungen von JUnit

Multithreading - JMTUnit Beim Testen von Anwendungen, die in mehrere Threads aufgeteilt sind, reicht ein normaler JUnit-Test nicht aus. Das Hauptproblem beim Testen solcher Multithreading-Anwendungen liegt im Nichtdeterminismus. Jeder Programablauf ist anders und das Verhalten hängt von verschiedenen Faktoren ab:

- Scheduling
- Leistung des verwendeten Systems
- Momentane Systemlast
- Verwendeter Compiler

Ein normaler Test ist durch diese Faktoren nicht einfach wiederholbar. Sein Ablauf ist einzigartig. Daher können bei gleichbleibender Eingabe manchmal Fehler auftreten, obwohl der Test in 99% der Durchläufe fehlerfrei abgearbeitet wurde. Dadurch sind Fehler oft nicht reproduzierbar und so auch schlecht zu finden. Die Ergebnisse solcher Tests sind noch unzuverlässiger als normale JUnit-Tests ohnehin schon sind.

Abhilfe soll das Framework JMTUnit[30] schaffen. Mit einem ähnlichen Interface wie JUnit bietet JMTUnit Erweiterungen an, die es ermöglichen speziell Programme mit mehreren Threads zu testen. Es versucht Deadlocks aufzuspüren und bringt eine Reproduzierbarkeit ins Testen. Aber die normalen JUnit-Tests müssen für JMTUnit angepasst werden.

Threading Model Um diese Reproduzierbarkeit zu ermöglichen, besitzt JMTUnit ein Threading Model. In diesem Model existieren *Worker-Threads* und normalerweise eine deutlich höhere Anzahl an *User-Threads*. Alle *Worker-Threads* warten in einem Pool auf ihren Einsatz. Jeder User hat eine gegebene Anzahl an Simulationsschritten, die nacheinander abgearbeitet werden. Dadurch wird eine identische Testreihenfolge garantiert.

[30] `http://www.michaelmoser.org/jmtunit/`

NoUnit Ein weiteres Problem ist die Qualität von JUnit-Tests. Werden alle Klassen getestet? Wo ist noch Code, der nur indirekt getestet wird? Auf diese Fragen versucht NoUnit eine Antwort zu geben. Dazu wird der Bytecode der Anwendung und der Tests analysiert. Diese Analyse wird dann in From eines HTML-Reports ausgegeben. Dabei unterscheidet NoUnit drei verschiedene Zustände für jede einzelne Klasse:

- Fully Tested
- Partially Tested
- Not Tested

Durch diese Aufstellung wird deutlich, an welchen Stellen die Tests noch Lücken aufweisen und man kann anhand der Komplexität und der Relevanz einer Klasse entscheiden, ob man zusätzliche Tests einfügt.

JUnitPerf Eine weitere Ergänzung zu JUnit ist JUnitPerf[31]. Perf steht für Performance. Mit JUnitPerf ist es möglich Anwendungen auf Performance und auf Skalierbarkeit hin zu untersuchen. Dazu bietet JUnitPerf verschiedene Decorators an:

- TimedTest misst die verbrauchte Zeit. Der Konstruktor erwartet einen TestCase und eine Zeitangabe. Wird diese Zeit überschritten, dann schlägt der Test fehl. Mit einem zusätzlichen optionalen Parameter kann fest gelegt werden, ob der Test nach einer Zeitüberschreitung weiter laufen soll, oder direkt abgebrochen wird.
- LoadTest ermöglicht einen Test mit mehreren simulierten ‚gleichzeitigen Benutzern und mehreren Iterationen laufen zu lassen. Zusätzlich kann im Konstruktor ein Timer angegeben werden, der die Benutzer steuert. Dadurch kann man eine konstante Zeit zwischen den Anfragen der Benutzer festlegen, oder man verwendet den RandomTimer, der zufällige Zeitabstände zwischen den Benutzern wählt.

Profiler

Ein Profiler (z. B. Eclipse Profiler Plugin [32]) bietet umfangreiche Möglichkeiten, ein Programm zur Laufzeit zu überwachen und zu analysieren. Dazu fügt der Profiler selbstständig in alle zu überwachenden Klassen eine *enter* und eine *leave* Methode ein. Bei jedem Aufruf einer so präparierten Methode wird ein Timestamp erzeugt. Mit Hilfe dieser Timestamps kann dann der exakte Zeitverbrauch jeder einzelnen Methode ermittelt werden. Aber auch darüber hinaus gehende Analysen sind möglich. In einem Callgraph kann zum Beispiel nachvollzogen werden, welche Methode andere Methoden aufgerufen hat.

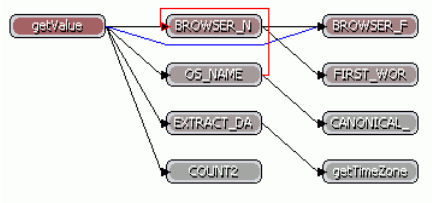


Abbildung 2.20.: Callgraph eines Profilers

[31] http://www.clarkware.com/software/JUnitPerf.html
[32] http://eclipsecolorer.sf.net/index_profiler.html

Unterschiedliche Farben und die Pfeildicke spiegeln Informationen über Zeitverbrauch und die Richtung der Aufrufe wieder. Weitere Ansichten bieten einen Überblick über Pakete und deren CPU-Zeit-Verbrauch. Mit Filtern kann man die Sicht auf die wesentlichen Teile begrenzen, da sonst diese Darstellungen schnell zu unübersichtlich werden. Weitere Möglichkeiten bieten Darstellungen von Thread-Trees und ein Graph der Heap-Auslastung.

Akzeptanztest mit FIT

Akzeptanztests stellen sogenannte Benutzergeschichten dar. Eine gute Einführung gibt es bei [Wes04].Sie geben das eigentliche Ziel der Entwicklung vor. FIT steht wie oben schon erwähnt für *framework of integrated tests*. FIT-Tests sind datengetrieben. Ausgehend von Daten werden verschiedene Methoden getestet. Dabei werden die Testdaten in einer HTML-Tabelle angeordnet. Um diese Tabellen können beliebige Erläuterungen notiert werden, die von FIT ignoriert werden. FIT liest dies Dokumente ein, führt die Tests mit den Daten aus den Tabellen aus und erzeugt ein Ergebnisdokument. Dabei werden drei Basis-Fixtures unterschieden:

- ColumnFixture: Prüft Eingabedaten gegen Ausgabedaten. Besonders geeignet für Geschäftslogik etc.
- RowFixture: Stellt Ergebnismengen dar, wie sie z.B. in einer Collection erwartet werden. Sie sammelt Ergebnisse auf und vergleicht sie dann auch Vollständigkeit und Korrektheit.
- ActionFixture: Orientiert sich mehr an der Benutzerschnittstelle. Sie führt eine Reihe von Kommandos auf der GUI aus und ist daher für Oberflächentests besonders geeignet.

Mit diesen drei Basis-Fixtures sind die meisten Fälle abgedeckt. Aber es besteht die Möglichkeit eigene Fixtures zu implementieren und so FIT an die eigenen Bedürfnisse anzupassen.

Zusammenfassung

Dieser Abschnitt konnte nur einen sehr kleinen Teil von JUnit-Erweiterungen und Ergänzungen vorstellen. Die Möglichkeiten Programme Tests zu unterziehen mit den verschiedensten Hilfsmitteln sollten nicht davon ablenken, dass immer noch einiges an Energie in gute Tests gesteckt werden muss. Und lieber sollte man ein paar Tests mehr schreiben, als zuwenige.

3. Manual

3.1. Installation

3.1.1. System Requirements

The program is written in Java (Version 1.4.2) to ensure platform independence. It is therefore possible to run the program on every operating system that supports Java 1.4.2 or later versions. It has been tested under Windows XP, Linux and MAC on computers with more than 128 MB of RAM. Some functions are computationally expensive and may take a while depending on CPU speed.

3.1.2. Installation

To install Nemoz you need the JVM (Java Virtual Machine) 1.4.2 or higher. If you don't have the actual JVM you can download it at Java.com <`http://java.com`>.

Download Nemoz from nemoz.sourceforge.net <`http://nemoz.sf.net`>. Using WinXP you can start the installation by a double click on the jar-file. If there's another program connected with jar-files besides the JVM you have to make a right click on the file and select 'Open with...' Java. The installation wizard will guide you through the installation.

After you have accept the terms of GNU General Public License you can select the packs you want to install. If you are interested in the source code, select the pack "Source".

Then select a installation path and the installation begins.

To complete the installation select a program group and decide if you want a shortcut on the desktop.

3.1.3. Deinstallation

To deinstall Nemoz use the Uninstaller in the Nemoz program group.

3.2. Getting started

Nemoz supports you in organizing your mp3 collection. It offers a lot of functionality for ordering, grouping and searching your songs. And you can share your music and your structures (Taxonomies) with other Nemoz users. A quick overview will introduce you to the basic functionality of Nemoz.

For a closer look on the components of Nemoz see the next chapters

- The User Interface
- Functionality

3.2.1. Import of Songs

 The only way to add local files to the library of Nemoz is the import. Importing does not only mean adding a link to them, but also extracting features of the song, which are used for the intelligent functions Nemoz offers. As this is computatinally expensive it might take a while.

Click on the Import-Button in the toolbar. In the Import Dialog select the mp3-files or a folder with mp3-files, you want to import. An imported song is added to the library. The physical file remains at the position in your filesystem. Only a link is added to the library.

3.2.2. Creating a Taxonomy

 To structure your collection you can create taxonomies above your library. Every taxonomy can distinguish songs by different topics. You might have one taxonomy to classify your songs by genre, and/or one that only says if you like a song or not. You can create a new taxonomy by clicking the ”New Taxonomy” button in the source view and enter a name for the new taxonomy.

3.2.3. Creating a Folder

 To create a new folder you first have to choose the taxonomy by selecting it in the source view. Then you can create a folder via the context menu in the taxonomy view or use the ”New Folder” button of the taxonomy view toolbar. There’s no limit to the number of folders in a taxonomy. So you can classify your songs as precise as you like.

3.2.4. Sorting songs by hand

If you have finished building your taxonomy you can fill it with songs of the library. There are two ways to add songs to a taxonomy:

- You can just drag’n’drop selected songs from the library to your taxonomy or
- you use the copy and paste buttons of the toolbar.

Afterwards you can sort them into the right folders by drag’n’drop.

3.2.5. Playing

 Nemoz has an integrated mp3 player. To play music you can select a song and press the play button or you simple double-click on the song you want to listen to.

The order of playing depends on the order in the taxonomy. The next song is the song under the current song in the played Taxonomy.

3.2.6. Further Functionality

Nemoz has much more functions. For further informations read the sections in the chapter Functionality. Here is only a short summary of the options you have:

- Network

  Nemoz supports sharing songs in a local network with other Nemoz usern.

  For more information see Network.

- Privacy Concept

  To protect your privacy Nemoz has a sophisticated privacy concept for songs and taxonomies.

  If you are interested how to use it see privacy concept for more information.

- Intelligent Functions

  The real power of Nemoz are the intelligent functions. Nemoz supports you by sorting songs into taxonomies and splitting taxonomy nodes that contains to much songs. Nemoz also gives you suggestions for labeling folders and taxonomies.

  See Intelligent Functions for more.

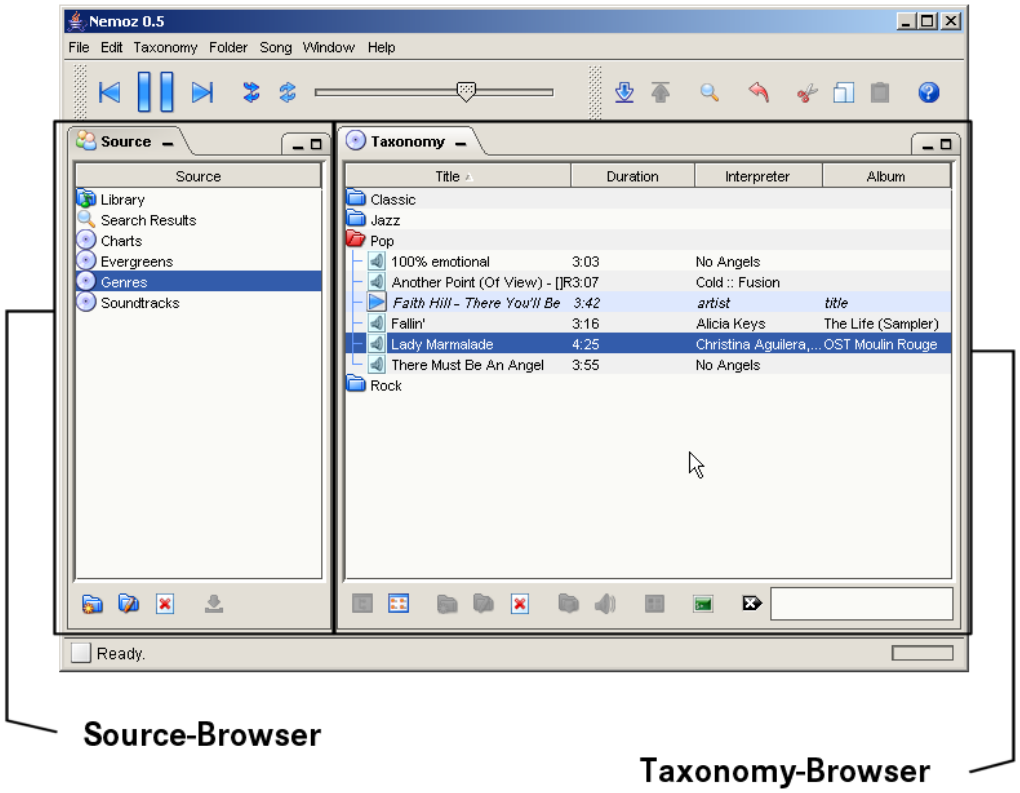
- Search

  One more highlight is the search Nemoz offers. Not only a realtime search in your library for songs by title, interpreter or album name, but also extended search functions for a search by for example similarity.

  Learn more about the search.

3.3. The User Interface

3.3.1. Main Window



The main window: Whenever you run Nemoz you will see this window. Every component will be explained in this chapter. So read on to have a closer look on it.

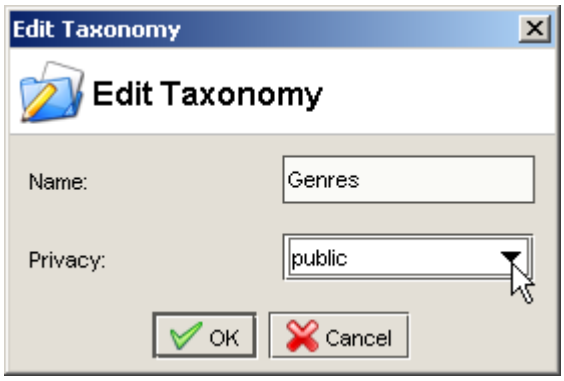
3.3.2. Source-Browser

On the left hand side you will see the sources window, every source of your music appears here. First of all there is your library, which contains all your songs. Then there is the ”search result” source: whenever you started a search with the search window, the result of this search will appear under this source taxonomy. The next things that appear in the source view are your own taxonomies. And the last items that are listed in the source window are the other Nemoz users.

On the bottom of the window you will recognize a toolbar. There you will find all actions concerning the sources.

 First of all there is the ”Create new Taxonomy” button, this will open a new dialog where you enter the name of the new taxonomy, and after pressing the ÖK” button the newly created taxonomy will appear in the sources window.

If you want to edit the name you've chosen for a taxonomy you will need the "Edit Taxonomy" button, same when you want to change the privacy of the selected taxonomy. Select a taxonomy and press the "Edit Taxonomy" button and you will see a new dialog, where you can choose a new name or change the privacy of the selected taxonomy.



If you want to delete the selected taxonomy, it's a good advice to press the "Delete Taxonomy" button. After you've confirmed that you really want to delete the taxonomy, it will be deleted.

The next two buttons are very similar, the first one will insert all descriptors that are in the clipboard (see Main Toolbar for details on clipboard) automatically in the selected taxonomy. Nemoz learns how you sorted the songs in the taxonomy and inserts the descriptors accordingly. The second button will do the same, but with all descriptors of your library. Remember that only copies will be inserted, your music will still be in all other taxonomies, especially in the library.

Of course you will find all of the described action also in the context menu, when you press the right mouse button, and in the main menu bar. Additionally you will find some more features which will be described later.

3.3.3. Taxonomy-Browser

Next to the sources window there is the taxonomy view window. This is the essential window for creating, looking at, playing and organizing your music structures. You choose an option out of the source window, which is described above. Then you will see the taxonomy you have selected in the sources view. Doesn't matter if this is a user, your library, a search result or a taxonomy. Everything appears in this window.

Let's have a closer look on editing a taxonomy you've created. First you will probably create some new folder.

 Therefore you choose the third button out of the toolbar on the bottom of this view, the ”Create a new Folder” button, and then you type a new name in the appearing dialog box and press ”OK”. The newly created folder will appear in the view. For subfolders it is necessary to select the parent folder and then press the button, the newly created folder will then appear in the selected folder.

 The next button is of course the ”Edit Folder” button, when you want to change the name of a folder use this one to do so.

 And whenever you want to delete a folder or a descriptor you should try the ”Delete Item” button, but remember, deleting a descriptor in a taxonomy will have no effect on the same descriptor in other taxonomies, especially your library. Only if you delete a descriptor in your library it will be really deleted everywhere, of course the file on your hard disk will still remain!

 Well, let’s get back to the toolbar, the next button allows you to download a descriptor, including the mp3 file, of another user, but only for the case that the privacy of this descriptor is set to public, otherwise this won’t work. You will find the file in your Nemoz directory.

 The ”Listen To Remote Descriptor” button will open a stream to the owner of this descriptor and you will be able to listen to the song.

 The ”Cluster” button will ask you to choose a number of clusters that should be created, i.e. the number of subfolders. Nemoz splits the folder you selected into this number of subfolders, putting silimar objects into the same folder.

 The next button will open a window where you can enter scheme commands. This is for really advanced users only, so the details are omitted here. Have a closer look on the source code in the folder ßcm” for details.

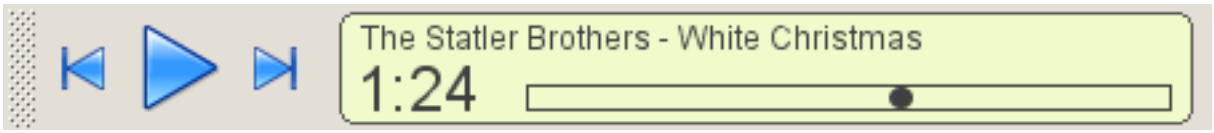
The last thing that is located in the taxonomy view toolbar is a really useful one. It’s the quick search, whenever you will search let’s say every music from ”Metallica” in the view you simply type ”Metallica”. When you type you will recognize that every character will make a change on the view. With this feature you will be really fast by finding your music.

 Well what about this two buttons? Don’t panic. These two will switch the view between treetable (the standard) and treemap. Just try, you will see the difference. A few word to the handling: In the treetable you will get a tree view on your taxonomy, where the details of the descriptors, like interpreter, title and so on, are listed as a table. By clicking twice on a folder you will open it. Drag’n’Drop is enabled here, so you can easily exchange descriptors and folders between your taxonomies or other users. But beware, when you take a descriptor from a other user, let’s say by drag’n’drop or by the clipboard you will only get the descriptor, not the mp3 file which is linked with it. To get the file have a closer look at the download button, which is described above. Well, the treemap is quite a cool feature; here

you can see your whole taxonomy at once. By clicking the title you will zoom into the folders, and by clicking the title again you will zoom out. So just try it, it's real fun.

Of course you will be able to choose all above mentioned actions in the main menu bar, too, and in the context menu. There will be a few more actions which will be described in this manual, too.

3.3.4. Player



In the upper left corner of the main window you will find the player toolbar. This will help you to listen to your music. First of all you will probably recognize the play button, it's a little bit larger ;-)

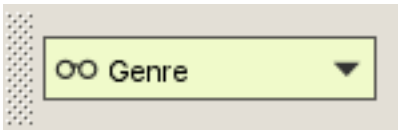
When you selected a descriptor in the taxonomy view window or a folder or even a whole taxonomy and then you press this button you will hear the selected descriptor(s). If more than one, a play list will be created and you will sequential hear the songs.

To step one song further or back use the buttons next to the play button. And that's all about the player. Well you will of course be able to jump in the played song by using the slider in the display. Here you can also see what's playing and how long it's already been playing.

3.3.5. Main Toolbar

-  The first button will open the import dialog, see that chapter for details.
-  The second button will export the selected structure to a XML file.
-  The third button will open the search dialog; see the next chapter for details.
-  The next button might be a really important one, if you damaged the Nemoz data structure. This will make a rollback and you get the last backup.
-  The next three buttons are cut, copy and paste, no need to explain this further.
-  The last button will open the help system of Nemoz, whenever you need it, just push this button.

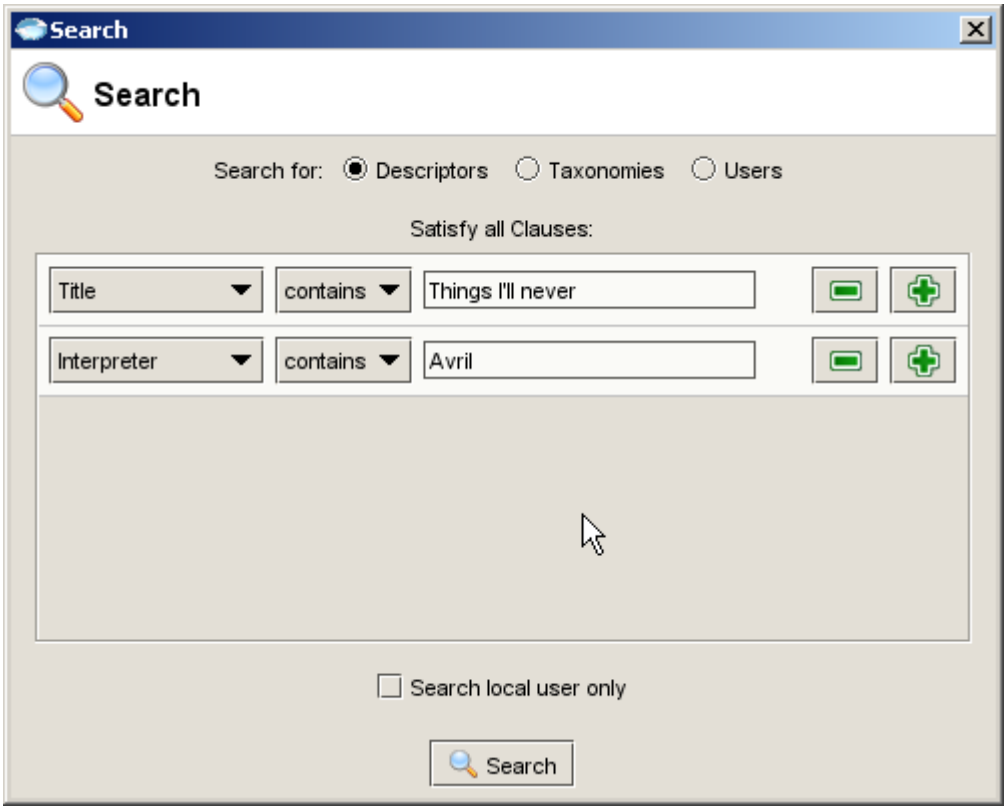
3.3.6. Goggle Toolbar



The Goggle Toolbar is another cool feature. The idea is to 'look through one taxonomy on another', i.e. to sort all the descriptors of one taxonomy into another to see where everything fits into.

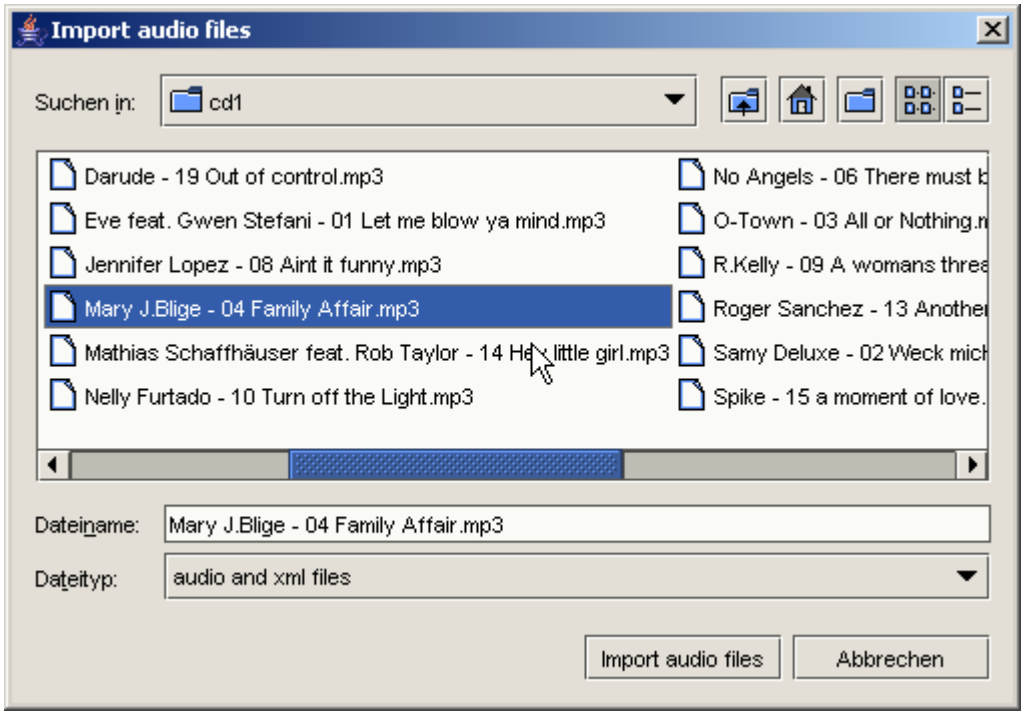
For more information see Goggle Operation.

3.3.7. Search Dialog



The search dialog is everything you need when you want to find something in the Nemoz Net. First you have to choose what kind of element you want to find: Descriptors, Taxonomies or Users. Then you have to choose the conditions which should be satisfied. If you need more conditions just push the green plus and you will get one additional row. Want less? Click on the green minus and a row will disappear. If you prefer to search only your local data you can check the box at the bottom of the dialog.

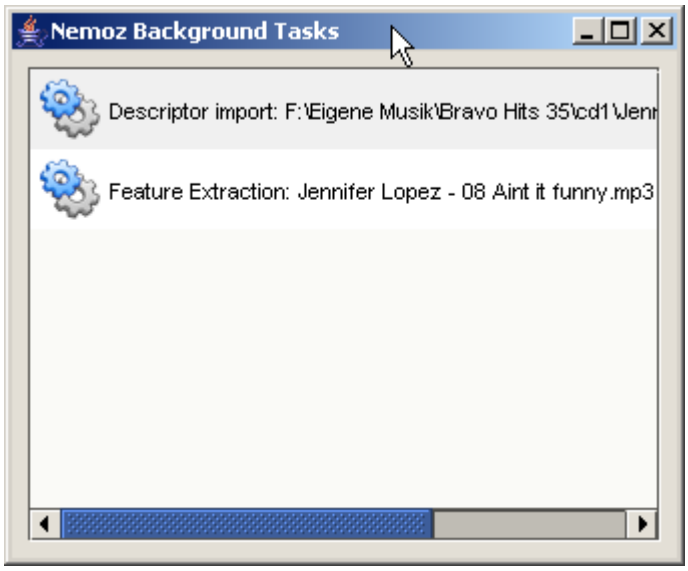
3.3.8. Import Dialog



In this dialog you can choose which mp3 files you want to import to Nemoz. You can either select one or more files or a directory. The directory will be browsed recursively and all music files will be added to your library.

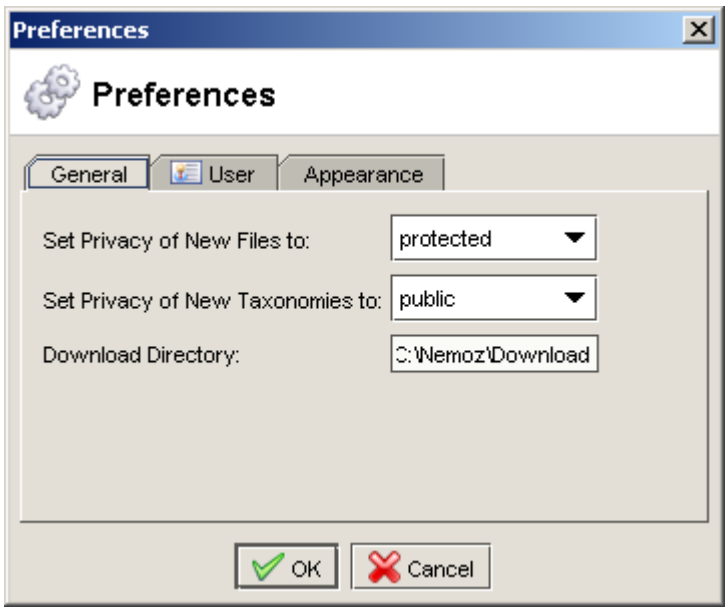
It's also possible to import an xml file, which you've previously exported.

3.3.9. Taskmanager



The task manager window will show you all tasks that are currently performed. At this moment it's not possible to kill or pause a task.

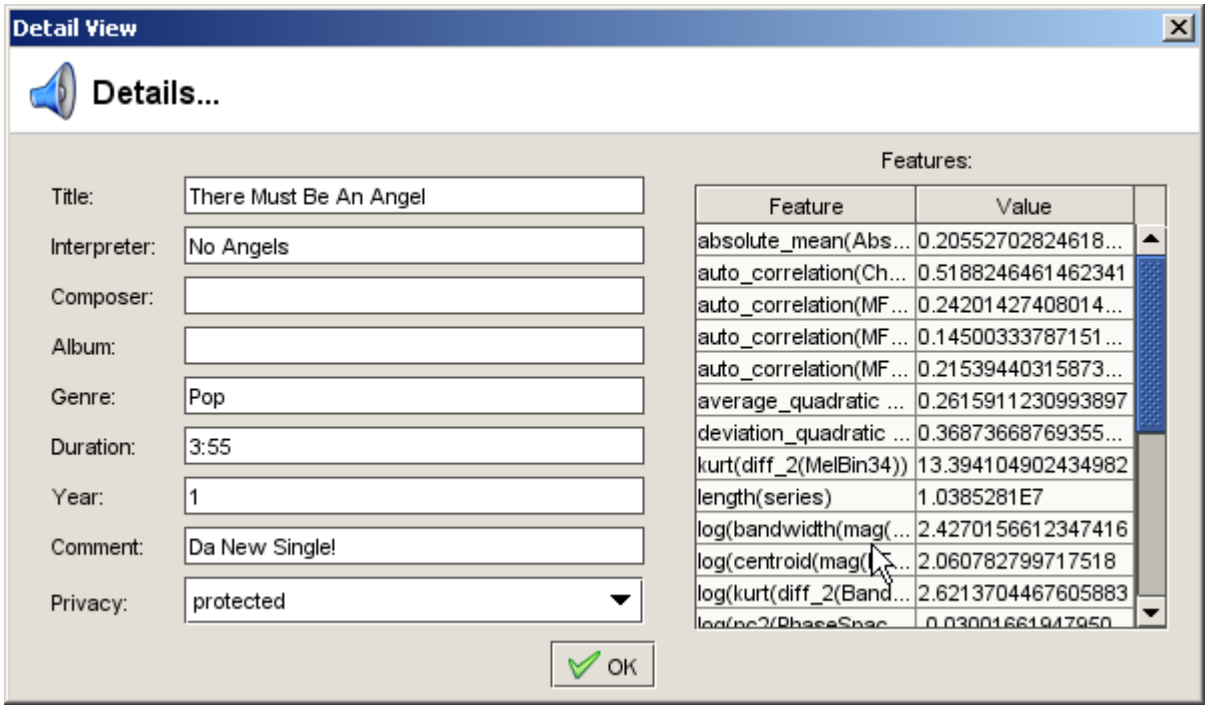
3.3.10. Preferences



In the General Tab you can set some general preferences like the privacy of new files etc. In the User Tab you can enter some details about your person. Most of it should be self-explanatory.

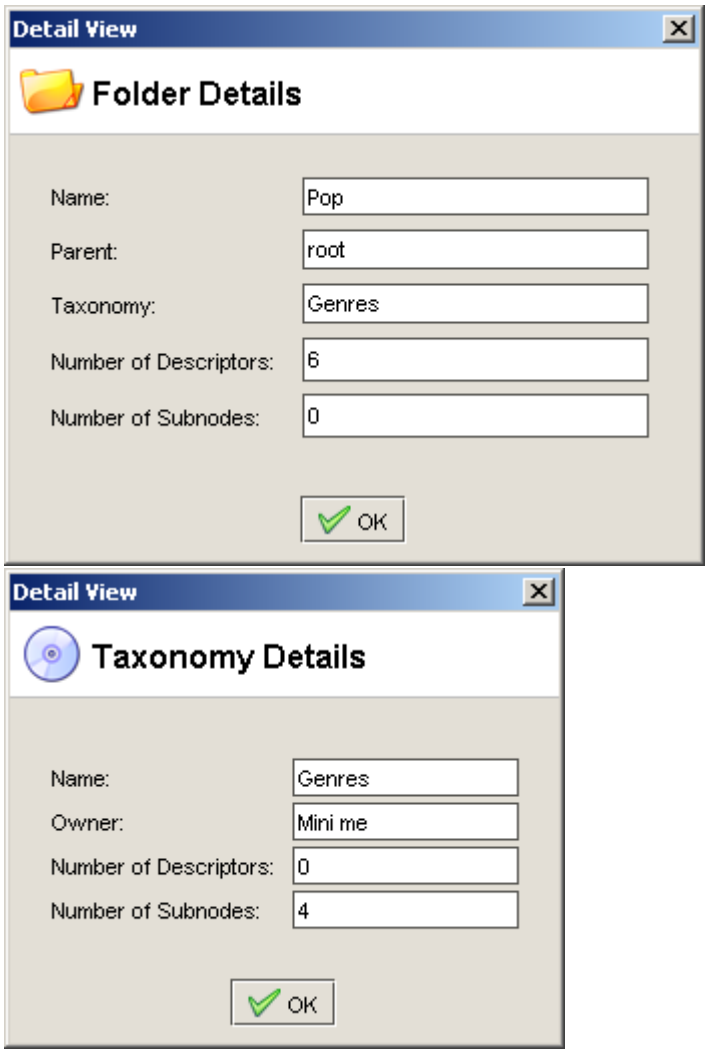
In the Appearance Tab you can choose between native and Nemoz, which is recommended.

3.3.11. Detail View of Songs



This dialog shows you some details of a selected song. You'll find it in the context menu of the song.

3.3.12. Folder & Taxonomy Details



These two dialogs show a few details of the selected folder or taxonomy.

3.3.13. User Details

This dialog shows you some details of a selected user. You'll find it in the context menu of the user.

3.4. Functionality

3.4.1. Songs

Songs are the core of Nemoz. In this section you will learn how to work with songs in Nemoz.

Import

First of all you have to import files to Nemoz. For this you can select the import button of the main toolbar or select it from the menu.

The songs are not touched. They stay where they are. Nemoz only analyzed them. While this analysis Nemoz finds charactaristic features for this song. This feature extraction will take a while, because Nemoz needs time to ”listen” to the song. Keep that in mind when you import folders or oodles of songs.

See also Import Dialog

Edit

You can change the privacy mode for every song. To do this select a song and you will find ”Details” in the context menu. In the Dialog you can change the privacy state of the selected song.

See also

Detail View of Songs.

- Detail View of Songs
- Privacy Concept

Download

If you are browsing the taxonomies of another Nemoz user, you might want to download a song. If this song is public you can start the download by the context menu or by the toolbar in the taxonomy view. The song will be saved in the download folder you can set on the Preferences & Settings Dialog.

See also

- Preferences
- Privacy Concept

3.4.2. Taxonomies

Taxonomies bring some structure in your song collection. In a taxonomy you can categorize songs under different topics. You can build unlimited taxonomies (so long as you have free memory). Taxonomies compries songs and folders. With a folder you can exacting your classification. A folder can contains subfolders and so on. A song can appear in every taxonomy exactly once.

Create

To create a new taxonomy click the NNew Taxonomy” button in the source view and enter a name for the new taxonomy.

To create a new folder you first have to chose the taxonomy by clicking it in the source view. Then you can create a folder via the context menu in the taxonomy view or use the ”New Folder” button of the taxonomy view toolbar.

See also

- Source View

- Taxonomy View

Edit

To change the name or to change the privacy state of a taxonomy use the edit button in the source view. Or simple use the context menu.

See also

- Folder & Taxonomy Details
- Privacy Concept

View

The default view in the taxonomy view shows you a tree view of your selected taxonomy. The treemap makes the features visible. There you can see your whole taxonomy at once. By clicking the title you will zoom into the folders, and by clicking the title again you will zoom out.

See also Taxonomy-Browser

3.4.3. Search

Another advanced feature is the powerfull search.

Quick searching by Title, Interpreter and Album

If you search for a specific song and you only know one word or a phrase of the titel, interpeter or the album the quick search is the function you need. Tip the phrase in the quick search box and all songs that match this phrase and are in or under the selected taxonomy will be itemized in the taxonomy view in real time.

Similar Songs

Because Nemoz analysed each song you can find songs that sounds similar to a given song. Therefore Nemoz use the charactaristic features which are calculated by importing.

To find songs that are similar to a selection of songs choose ”Find Similar Songs” in the context menu of the songs. In the ”Search Result” the result is shown.

Advanced Search

But the really advanced search you will reach over the loupe icon on the toolbar. In the search window you can define clauses which have to be satisfied. You can add more clauses by using the ”+” button. Every clause contains a subject that you are searching for, a predicate and a condition. When you have all your clauses defined click the ”Search” button and the result will be appear under ”Search Results” in the source view.

Songs You can search songs by

- Title
- Interpreter
- Album
- Composer
- Genre

The search goes over the ID3-Tags.

Taxonomies Taxonomies can be searched by name.

User Users can be searched by name and age.

3.4.4. Intelligent Functions

Nemoz has some advanced features. It uses techniques from machine learning to support you.

Automatically Insert

If you have filled your taxonomy with songs, you can try the function ”Automatically Insert”. Copy the songs you want to insert into the clipboard and use the ”Auto. Insert” button in the context menu of the taxonomy, in which the songs shoud be inserted.

The more songs you sort by hand in the taxonomy the better is the outcome.

Clustering of a Folder

If a folder in one of your taxonomies has too much songs so that it is too complex, then you can use the clustering function. Clustering splits the folder into a number of subfolders by grouping similar songs. To cluster a folder choose ”Cluster” from the context menu in the taxonomy view.

Goggle Operation

Sometimes it is interesting to look at some desriptors through a taxonomy. For this task you can use the goggle operator. Select a taxonomy of the goggle-combobox as the goggle-taxonomy. Now you can see all other taxonomies throw this goggle-taxonomy.

For example you can make a ”like/disliketaxonomy and use this as goggle-taxonomy. If you now browse through the taxonomies of another user Nemoz sorts the songs of this user into the ”like/dislike” structure of the goggle-taxonomy.

3.4.5. Network

You can share your songs and your taxonomies. To control access to your content you can use the privacy settings.

Browsing other Users Taxonomies

First of all you can browse taxonomies of other users. Users in your network are announced in the source view by an icon and their name. Select one user and in the taxonomy view there will appear all public taxonomies of the selected user. Now you can browse through this taxomonies similar to your own collection of taxonomies. When you select a node, only the songs and the subfolders are loaded. This reduces the traffic and makes Nemoz faster. But if you select a new node or subnode it will take a moment until the content is showed.

Download a song

If a song is public you can download it. Select the song you want to download and use the ”Download” button from the context menu. In the Preferences you can change the download directory. The download will start immediately.

Copy a Taxonomy

If you like a taxonomy of an other user you can copy the structure of this taxonomy in your own collection. Select a taxonomy and use the context menu.

3.4.6. Privacy Concept

Nemoz has a sophisticated privacy concept for songs and taxonomies.

Privacy of songs

Every song has one of the following privacy state:

-  private

  a private song is only visble for you. No one else can see or download it.

-  protected

  songs that has the state protected are visible for other users, but nobody is able to download it.

-  public

  public songs can be accessed and downloaded by all other Nemoz users.

To change the privacy state of a song select it and select ”Details” form the context menu. In the following popup window you can change the privacy.

To change the default privacy state for new songs you can use from the ”File” menu the ”Preferences” option. In the Combo-Box you can select one of the three privacy states.

See also

- Detail View of Songs
- Preferneces

Privacy of Taxonomies

Taxonomies only need two privacy states:

-  private
  private Taxonomies are not visible to other Nemoz users.
-  public
  in contrast public taxonomies are shared with all other Nemoz users.

To change the privacy state of a taxonomy select it and select ”Edit” form the context menu. In the following popup window you can change the privacy.
The default privacy setting for taxonomies can be set in the ”Preferences” dialog.

4. Umsetzung

Die softwaretechnische Umsetzung der komplexen Spezifikation von Nemoz stellt sicherlich eine Herausforderung dar, wie schon die mit ca. 60.000 Zeilen *Java* sehr umfangreiche Implementierung zeigt.

Dieses Kapitel beginnt mit einem Überblick über allgemeine, sowie für Nemoz spezifische Herausforderungen, der Softwareentwicklung in Projektgruppen und die daraus resultierende Designphilosophie. Es folgt eine Beschreibung der konkreten Softwarearchitektur von Nemoz, die sich an deren geschichtetem Aufbau orientiert. Diese Beschreibung beginnt mit der innersten Schicht, dem Domänenmodell, wendet sich nach einem kurzen Exkurs über Architektur- und Infrastrukturframeworks der Serviceschicht zu, um nach Behandlung der Integrationsschicht mit den äußersten, dem Benutzer zugewandten Schichten, der graphischen Benutzerschnittstelle und der *Script Engine*, zu schließen.

4.1. Überblick

4.1.1. Herausforderungen

Am Anfang der Entwicklung von Nemoz stand die Entscheidung, wer die Zielgruppe des Programms sein sollte, wobei rein akademische Nutzung als auch die Nutzung durch einen allgemeineren Personenkreis ("Endbenutzer") denkbar war. Durch die Entscheidung für ein "populäres" System stellten sich erhöhte Anforderungen nicht nur an die Benutzerfreundlichkeit, sondern auch an Antwortzeiten, Skalierbarkeit, Erweiterbarkeit und Stabilität.

Akzeptable Antwortzeiten lassen sich nur durch eine hohe Nebenläufigkeit erreichen, wobei Nemoz als verteilte Anwendung natürlich ohnehin Nebenläufigkeit voraussetzt. Die Verwendung einfacher adhoc Algorithmen und Datenstrukturen verbietet sich von selbst, wenn Skalierbarkeit eine Rolle spielt. All dies führt zu einer immensen Komplexität der Software, was zweifellos eine Herausforderung an sich darstellt. Die Bewältigung dieser Komplexität unter Wahrung eines hohen Qualitätsstandards erfordert ein flexibles, dabei aber ausreichend strenges Verfahrensmodell zur Softwareentwicklung im Team. Dieses muss auch unter den besonderen Bedingungen einer Projektgruppe effektiv anwendbar sein.

Benutzerfreundlichkeit

Benutzerfreundlichkeit und Ergonomie sind allgemeine Werte, die sich schwer messen und leider noch schwerer erreichen lassen. Eine schöne GUI alleine genügt nicht, alles muss in sich stimmig sein. Ein für die Benutzer intuitiv verständliches Modell des Anwendungsgebiets ist unerlässlich. Was gut für die Nutzer ist, kann für die Entwickler nicht schlecht sein und so sollte es nicht überraschen, wenn sich nicht nur die Benutzerschnittstelle, sondern auch der Entwurf und die Implementierung sehr gut auf einem solchen *Domänenmodell* aufbauen lassen. Dazu jedoch mehr in Abschnitt 4.1.2.

Die Benutzerfreundlichkeit moderner GUIs hat heute einen sehr hohen Standard erreicht. Leistungsmerkmale wie *Undo*, *Copy and Paste* oder *Drag and Drop* sind eine Selbstverständlichkeit, ihre Implementierung ist aber auch unter Zuhilfenahme moderner Frameworks wie *Java Swing* nicht trivial. Zu diesem Grundstock an zu implementierender Funktionalität kommen die besonderen Anforderungen an eine GUI zur kollaborativen Strukturierung von Audiodaten. Diese umfassen das Finden geeigneter graphischer Darstellungen für strukturierte Mengen von Musikstücken, die Visualisierung neuer automatischer Strukturierungshilfen oder auch die Darstellung der Aktivität entfernter Benutzer, um nur

einige Beispiele zu nennen.

Nebenläufigkeit und Verteiltheit

Während die Implementierungssprache für Nemoz, *Java* in der Version 1.4.2, Netzwerkprogrammierung sehr gut unterstützt, bietet sie leider nur rudimentäre Werkzeuge zur Entwicklung nebenläufiger Anwendungen. Thread-Synchronization wird nur durch einfache Monitor-Locks unterstützt, kompliziertere Primitive wie Read-Write-Locks oder Semaphoren, deren Verwendung in einer zuverlässigen nebenläufigen Anwendung unverzichtbar sind, müssen selbst entwickelt oder wenn möglich aus Bibliotheken bezogen werden. Der richtige Umgang mit diesen Methoden zur Thread-Synchronisation ist nicht eben trivial und setzt gute Kenntnisse und auch ein wenig Erfahrung in diesem Gebiet voraus. Allgemein ist das Testen von nebenläufigen Programmen mit automatischen Verfahren wie Unit-Tests schwierig, ähnliches gilt für das Beweisen von Eigenschaften nebenläufiger Programmme.

Die Netzwerkprogrammierung bietet eine Reihe technischer und konzeptueller Herausforderungen. Automatische Tests von Netzwerksoftware ist mit den heutigen Standardwerkzeugen schwierig bis unmöglich. Ein Netzwerksystem für Nemoz muss mit Verbindungen unterschiedlicher Geschwindigkeit und Zuverlässigkeit zurechtkommen und dabei Klippen wie zum Beispiel Firewalls umschiffen, während es gleichzeitig effizient die zur Verfügung stehende Bandbreite nutzt und mit steigender Benutzerzahl im Netzwerk gut skaliert.

Hinter vielen dieser technischen anmutenden Herausforderungen der Netzwerkschicht verbergen sich in Wirklichkeit konzeptuelle Probleme und Ambiguitäten, die sich durch nur durch Ergänzung und Vertiefung des der gesamten Anwendung zugrundeliegenden Domänenmodells lösen lassen. So gibt es zum Beispiel die Möglichkeit, die Sichten der Benutzer auf den Datenbestand ihrer Peers zu allen Zeiten synchron und konsistent zu halten (*Stateful World*), was zwar benutzerfreundlich, aber sehr schwierig in skalierbarer Weise umsetzbar ist. Die Alternative einer *Stateless World*, welche schließlich für Nemoz gewählt wurde, läßt es zu, dass die Sichten der Benutzer auf den Datenbestand ihrer Peers inkonsistent werden können und überträgt die Verantwortung der Sichtsynchronisation auf die Benutzer. Designentscheidungen wie diese müssen in Einklang mit dem Domänenmodell gebracht werden.

Skalierbarkeit

Die Benutzer aktueller Musikmanagementsoftware besitzen oft Musiksammlungen von einigen tausend Musikstücken[1], die sie effizient verwalten möchten. Während die Performanz vieler der in Nemoz einsetzbaren Lernverfahren von großen Beispielmengen profitiert, stellen die damit einhergehenden Datenmengen aber vor allem eine Herausforderung dar. So ist es ab einer bestimmten Anzahl nicht mehr möglich, alle Deskriptoren einer Musiksammlung komplett im Hauptspeicher zu halten, was die Verwendung vieler einfacher Algorithmen und Datenstruktuen zum Speichern und Suchen ausschließt.

Ein naheliegender Lösungsweg für dieses Problem, die Verwendung eines der üblichen relationalen Datenbankmanagementsysteme, ist wegen der Natur der in Nemoz vorkommenden Daten und Anfragetypen leider blockiert. Daher müssen eigene Infrastrukturframeworks zum Speichern und Suchen entwickelt werden, die auf Datenstrukturen und Algorithmen basieren, welche für die speziellen Anforderungen von Nemoz angepasst oder sogar entworfen sind.

Wartungsfreundlichkeit und Erweiterbarkeit

Das Ziel, Nemoz als nützliches Softwareprodukt zu etablieren, dessen Lebensspanne nicht auf die Dauer einer Projektgruppe beschränkt bleibt, sowie die Veröffentlichung von Nemoz unter einer Open Source Lizenz stellt hohe Anforderungen an Wartungsfreundlichkeit und Erweiterbarkeit. Die Möglichkeit von

[1] 5000 ist eine realistische Anzahl von Musikstücken, die Nemoz noch effizient verwalten sollte.

sich an das Projekt anschließenden Diplomarbeiten macht diese Anforderungen erfreulicherweise auch direkt zum persönlichen Anliegen eigentlich jedes Teilnehmers.

Ein sauberer Entwurf, wie er durch die Zugrundelegung eines durchdachten Domänenmodells möglich wird, hilft bereits, eine gewisse Wartungsfreundlichkeit und Erweiterbarkeit zu erreichen. Da sich aber nicht jeder, der gerne Erweiterungen für Nemoz entwickeln würde, in das gesamte System einarbeiten kann oder möchte, ist eine klar spezifizierte Programmierschnittstelle für Erweiterungen nützlich und auch nötig. Eine bessere Lösung wäre ein Pluginsystem, wie es zum Beispiel in *YALE* [RKF+01] oder *Eclipse* [BG04] realisiert ist. Dieses Pluginsystem würde es ermöglichen Features, die Benutzer von einer Musikmanagementsoftware erwarten, die aber aus Zeitgründen nicht im Rahmen der Projektgruppe entwickelt werden können, als Plugin nachzuliefern. Eine mögliche spätere Erweiterung zur Unterstützung von Videodaten würde durch ein umfassendes Pluginsystem ebenfalls sehr erleichtert.

Für einfache Erweiterungen, zur Automatisierung von Routineaufgaben, zur Nutzung einer Anwendung an deren GUI vorbei und zu deren passgenauer Anpassung an die Anforderungen eines fortgeschrittenen Benutzers werden oft Skriptsprachen eingesetzt. Eine flexible Skriptsprache ist also ein wünschenswertes Leistungsmerkmal für jede größere Anwendung. Hinzu kommt, dass die Entwicklung der Abstraktionen und Sprachkonstrukte einer speziell auf ein Anwendungsgebiet angepassten Skriptsprache sehr häufig zu einem vertieften Verständnis der Konzepte des Anwendungsgebiets selbst führt. Dieses neu erworbene Wissen kommt über ein verbessertes Domänenmodell der gesamten Anwendung zugute.

Komplexität der Problemstellung

Wie inzwischen sicher klar geworden ist, bedingt die Kombination von Verteiltheit, Nebenläufigkeit, den Anforderungen an Stabilität und Skalierbarkeit und nicht zuletzt einem umfangreichen Pflichtenheft die große Komplexität von Nemoz. Ein weiterer Faktor ist, dass die einzelnen Elemente des Domänenmodells konzeptuell stark voneinander abhängen, eine einfache Trennung in einzelne, unabhängige Module ist nur schwer möglich. Es führt kein Weg daran vorbei, dieser inhärenten Komplexität mit einer ausgefeilten Softwarearchitektur zu begegnen, auch wenn so nicht viel "Room for Lunch" übrig bleibt.[2].

Die Anbindung von *YALE* löst natürlich viele Probleme, bietet aber auch eigene Herausforderungen. So verwendet *YALE* ein eigenes Domänenmodell, dessen Elemente auf Elemente des Nemoz-Modells abgebildet werden müssen, ohne die Integrität eines der beiden beteiligten Modelle zu gefährden, was die Komplexität der Problemstellung weiter erhöht.

Die begrenzten Mittel einer Projektgruppe (eigentlich jeden Softwareprojekts) erzwingen Kompromisse bei einigen Detaillösungen. Die Kunst beim Eingehen von Kompromissen besteht darin, diejenigen zu vermeiden, die später oftmals zu überproportionaler Mehrarbeit führen. Sicherer ist es, wann immer möglich Kompromisse bei den Anforderungen zuzulassen, für jedes Leistungsmerkmal also verschiedene Ausbaustufen vorzusehen. Das Erfolgsgeheimnis liegt in einem vernünftigen Mittelweg zwischen *"The Right Thing"* und *"Worse Is Better"* [Gab91].

Softwareentwicklung in Projektgruppen

Softwareentwicklung im Team ist ein prinzipiell schwieriges Unterfangen. Auch wenn heute sogar schon die Rede ist vom Übergang des Software-Engineering in das goldene Zeitalter der "Software-Industrialisierung", eine Wahrheit bleibt bestehen: Darüber, wie man Softwareentwicklung richtig macht, weiß man auch heute nicht viel, weniger noch darüber, wie man Softwareentwicklung im Team richtig macht [FPB78],[NB02]. Wenn das Team dann eine Projektgruppe ist, eine Projektgruppe aus naturgemäß eher unerfahrenen Studenten - so motiviert sie auch sein mögen - ist das sicherlich keine große Hilfe. Einer Projektgruppe, die sich den oben skizzierten Herausforderungen stellen will, ist sicher nicht zu wünschen, dass ihr in den hoffnungsfrohen Tagen ihrer Jugend zwei Teilnehmer abhanden kommen.

[2] "Room for Lunch" war laut Ken Thompson eine der Designmaximen bei der Entwicklung von Unix. Ein Ausdruck des gesamten Quelltextes sollte in eine Aktentasche passen und noch genügend Platz für das Mittagessen lassen.

Projektgruppen sind typischerweise sehr heterogen, die Nemoz-Projektgruppe vielleicht sogar mehr als üblich. Glücklicherweise bot die Spezifikation eine Vielzahl von gänzlich unterschiedlichen Betätigungsfeldern, von wissenschaftlicher Arbeit im Bereich der künstlichen Intelligenz über Softwarearchitektur bis hin zu Dokumentation, Design und Musik. Diese Vielfalt der Aufgabengebiete und die Mischung von Theorie und Praxis bedingt Teilnehmer aus ganz unterschiedlichen „Subkulturen“ innerhalb und außerhalb der Informatik und stellt eine Herausforderung dar. Aber nicht zuletzt ist es genau diese Mischung, die Nemoz interessant macht.

4.1.2. Philosophie, Pattern und Paradigma

Nachdem die Herausforderungen, denen sich die Projektgruppe bei der Umsetzung von Nemoz zu stellen hatte, anschaulich gemacht wurden, beschäftigt sich dieses Kapitel mit den Pattern und Paradigmen, Konzepten und Kompromissen dieser Umsetzung - mit der Designphilosophie also.

Die „Nemoz-Philosophie“ ist ein postmodernes Patchwork von Einflüssen und Ideen, denen im Folgenden ein wenig Tribut gezollt werden soll. Diese Betrachtungen sind zwar hilfreich, aber sicher nicht essentiell für das Verständnis der Nemoz-„Makroarchitektur“, deren Beschreibung im folgenden Abschnitt 4.1.3 erfolgt.

Domain-Driven Design

Domain-Driven Design ist eine Denkweise und eine Menge von Prioritäten, die auf die Beschleunigung von Softwareprojekten in komplizierten Einsatzgebieten zielen[Eva04]. Domain-Driven Design ist die zugrundeliegende Philosophie einer großen Zahl von Herangehensweisen und Methoden, die erfahrene Entwickler ganz intuitiv zur Lösung komplizierter Probleme anwenden. Domain-Driven Design ist die der Nemoz-Softwarearchitektur im Kern zugrundeliegende Designphilosophie.

Domain-Driven Design legt dem gesamten Softwareentwicklungsprozess, von der Definition der Anforderungen bis hin zu Implementierung und Deployment, ein gemeinsames und umfassendes Modell des Anwendungsgebiets zugrunde. Dieses *Domänenmodell* (engl. *Domain-Model*) muss sich also ebenso gut für die Problemanalyse wie für die Implementierung eignen. Dies setzt einen „agilen“ Entwicklungsprozess voraus, der diese Phasen nicht strikt und bürokratisch trennt und den Entwicklern auch keine allzu starren Rollen auferlegt. Domain-Driven Design ist eine ambitionierte Designphilosophie für anspruchsvolle Anwendungen und verlangt eigentlich nach einem Team erfahrener Entwickler. In diesem Sinne ist sie die Antithese zu dem viel zu oft eingesetzten SMART-UI-(Anti-)Pattern und der nicht immer, aber oft bei Projektgruppen beobachteten BIG BALL OF MUD-(Anti-)Pattern Language [Foo97] (Abschnitt 4.1.2 gibt eine kurze Einführung in Design Patterns).

Da Analyse, Design und Implementierung auf dem selben Modell basieren, wird das eingesetzte Programmierparadigma einen großen Einfluss auf dieses Modell ausüben (Abbildung 4.1). Es sollte daher mit Bedacht gewählt werden. Objektorientierte Programmierung ist das derzeit vorherrschende Paradigma und eignet sich sehr gut für primär reaktive Systeme wie Nemoz. Allerdings beschäftigt sich Nemoz im Kern seines Domänenmodells intensiv mit Taxonomien und statistischen Lernverfahren, die sich oft viel klarer und direkter im funktionalen Paradigma modellieren ließen. Soweit es ging wurde an diesen Stellen auch in einem funktionalen Stil modelliert und implementiert, allerdings setzten die gewählten Tools diesem Unterfangen enge Grenzen bezüglich dem, was nicht nur elegant, sondern auch effizient möglich war. Es gibt also genügend Argumente, die für eine Implementierung dieser Teile in einer funktionalen Programmiersprache sprechen, da allerdings viele der eingesetzten Lernalgorithmen innerhalb von *YALE* bereits in einer objektorientierten Implementierung vorlagen und weil mehrere Paradigmen in einem Modell oft die Gefahr konzeptueller Inkonsistenz in sich bergen, wurde auf einen Multiparadigmen-Ansatz verzichtet.

Abbildung 4.2 zeigt einen Ausschnitt der Pattern Language des Domain-Driven Design, welche bei der Entwicklung eines flexiblen, klaren und tiefgründigen Domänenmodells helfen soll.

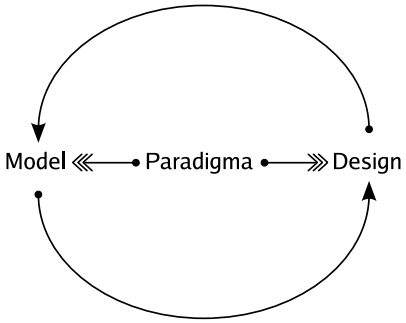


Abbildung 4.1.: Modell, Paradigma und Design

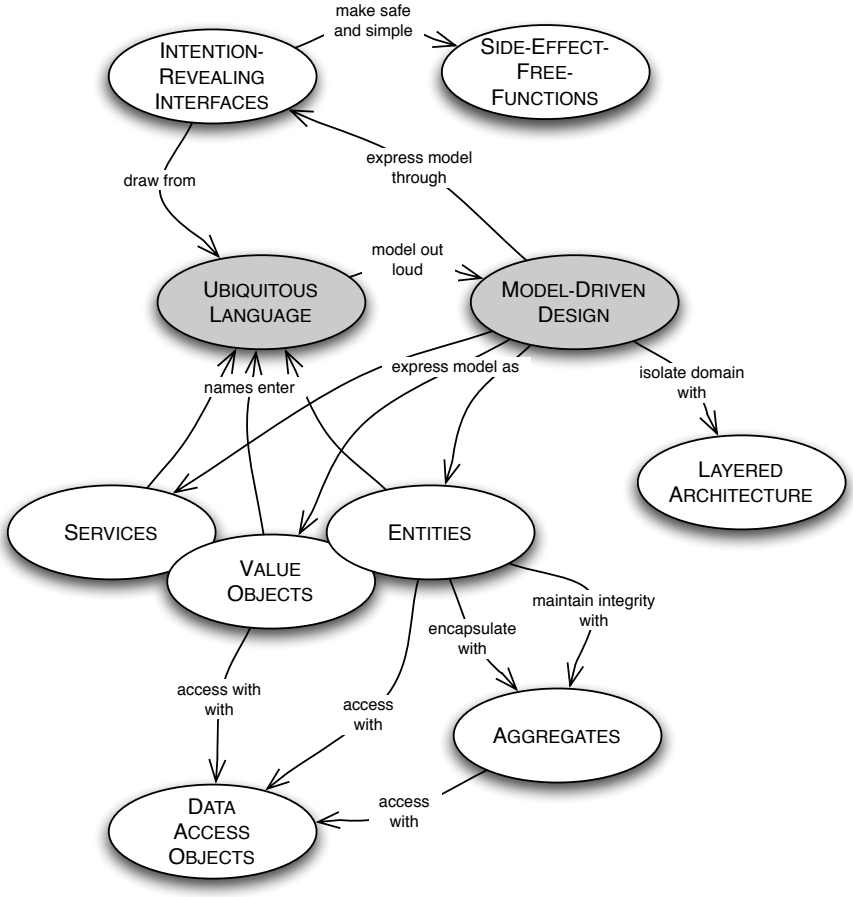


Abbildung 4.2.: Patterns des Domain-Driven Design, nach [Eva04]

Model-Driven Design Die Pattern MODEL-DRIVEN DESIGN und UBIQUITOUS LANGUAGE beschreiben Kerneigenschaften des Domain-Driven Design. Ein MODEL-DRIVEN DESIGN gibt ein Domänenmodell nahezu „wörtlich“ in Software wieder, die Abbildung zwischen Software und Modell sollte offensichtlich sein. ENTITIES, VALUE OBJECTS, AGGREGATES und SERVICES stellen die Elemente dieses Modells, welches in vielen Iterationen verfeinert und vertieft wird, um es einerseits natürlicher und effizienter in Software auszudrücken und andererseits, um die inhärenten Konzepte des Anwendungsgebiets besser herauszuarbeiten. Das Modell unterstützt eine UBIQUITOUS LANGUAGE auf effektive Weise. Namen von Modellelementen und deren Relationen fließen in diese UBIQUITOUS LANGUAGE ein. Das Ergebnis dieses Prozesses ist ein Modell, dass durch die Explizitmachung grundlegender Eigenschaften, Relationen und Elemente des Anwendungsgebiets die Entdeckung leistungsfähiger neuer Funktionen ermöglicht. Eine mathematische Theorie, die mit einigen wenigen einfachen Objekten und allgemeinen Sätzen eine Vielzahl von Anwendungen unterstützt, ist eine geeignete Metapher für ein exzellentes Domänenmodell.

Ubiquitous Language Eine Sprache, die dem Anwendungsgebiet zugrundeliegende Konzepte explizit benennt und von allen Teammitgliedern „gesprochen“ wird, ist eine der wichtigsten Voraussetzungen für den Erfolg eines Softwareprojekts. Diese gemeinsame Sprache ermöglicht erst eine effektive Kommunikation zwischen Teammitgliedern, die Missverständnisse vermeiden hilft und erarbeitetes Wissen vermittelbar macht. Ein spielerischer Umgang mit Sprache hilft dabei oft, neue Konzepte und Abstraktionen zu finden und dadurch das Domänenmodell zu vertiefen. Einige Beispiele sollen ein Gefühl für die Vorteile dieses Patterns vermitteln. Die UBIQUITOUS LANGUAGE in Nemoz ermöglicht es Sätze wie „Die `User`in «Jane» fügt durch eine `AddOperation` einen `Descriptor` zu ihrer `Library` hinzu.“ zu formulieren. Statt „Die GUI bekommt eine Suchanfrage als `String` und sucht in der `Map` der Datenbank nach passenden Einträgen.“ (technisch und vage) heißt es in Nemoz: „Der `SearchService` findet `Descriptor`en, welche die `Predicate`s eines `SearchQuery` erfüllen.“.

Entities Manche Objekte sind nicht primär durch ihre Attribute definiert, sondern repräsentieren Gegenstände mit kontinuierlicher Identität. Ein Objekt einer Klasse Person mit Attributen Name und Vorname muss von anderen Objekten dieser Klassen unterschieden werden können, selbst wenn Name und Vorname gleich sind (es gibt sicherlich viele Jane Does auf der Welt). Andererseits müssen einige Person-Instanzen identifiziert werden, selbst wenn sich ihre Attribute, z.B. der Nachname nach einer Hochzeit, nun unterscheiden. Objekte mit diesen Eigenschaften heißen ENTITIES. Sie bekommen meist eine eindeutige ID in Form eines Attributs zugeordnet. Wichtig ist, dass ENTITIES als konzeptuelle Einheiten oft eine längere Lebensdauer haben als die sie repräsentierenden Objekte. So kann eine ENTITY im Laufe ihrer Lebensspanne mehrmals in einer Datenbank gespeichert oder sogar auf andere Systeme übertragen werden, wo sie möglicherweise völlig anders dargestellt wird.

Value Objects Viele Objekte besitzen keine konzeptuelle Identität, sie beschreiben viel mehr Eigenschaften von Dingen. Farben sind ein gutes Beispiel für solche Objekte. Farben können sehr komplizierte Darstellungen besitzen, wie schon die große Anzahl verschiedener Farbräume wie RGB, CMYK oder HSV zeigt. Farben können durch Mischung anderer Farben entstehen und zwar auf die erstaunlichste Weise. Trotzdem ist es der Künstlerin einerlei, wie nun genau eine bestimme Instanz eines karminroten Buntstiftes entstanden ist, er kann schadlos durch jeden beliebigen anderen karminroten Buntstift ersetzt werden. Objekte mit dieser Eigenschaft heißen VALUE OBJECTS. Sie sind fast immer *immutable* und können daher beliebig kopiert und ausgetauscht, aber für Änderungen nur als Einheit ersetzt werden. Diese Eigenschaften sind der Schlüssel für einen funktionalen Stil, der nur VALUE OBJECTS erlaubt. Dieser Stil ermöglicht eine Reihe von Optimierungen und Vereinfachungen im Domänenmodell.

Services Manche Operationen lassen sich nicht in natürlicher Weise einer bestimmten Klasse zuordnen, gehören aber dennoch zum Domänenmodell. Diese Operationen heißen SERVICES. Sie sind nicht zu verwechseln mit den allgemeinen Diensten der Infrastrukturschicht. Beispielsweise gehört die ähnlichkeitsbasierte Suche nach Deskriptoren und Taxonomien zur Kernfunktionalität von Nemoz und damit zu dessen Domänenmodell. Allerdings läßt sich diese Funktionalität, die Deskriptoren wie Taxonomien gleichermaßen betrifft, keiner dieser beiden Klassen widerspruchsfrei zuordnen. Ein eigens eingerichteter SearchService löst das Problem. SERVICES sind zustandsfrei, eine `SearchQuery` an den `SearchService` hat keine Seiteneffekte und somit keine Auswirkungen auf die Ergebnisse nachfolgender `SearchQueries`.

Aggregates Ein AGGREGATE ist eine konzeptuell stark zusammenhängende Menge von Klassen, die eine Einheit bezüglich Änderungen bildet. Eine definierte Klasse, welche mit dem Stereotyp «Aggregate Root» markiert wird, bildet die Schnittstelle des AGGREGATEs zur Außenwelt. Alle Änderungen an Mitgliedern des AGGREGATEs dürfen nur durch Methoden des «Aggregate Root»s erfolgen. Klassen außerhalb des AGGREGATEs dürfen nur temporäre Referenzen auf Klassen innerhalb des AGGREGATEs halten, außer es handelt sich bei der referenzierten Klasse um das «Aggregate Root». Dieses Pattern ermöglicht es, Konsistenz in Form von Invarianten zwischen Klassen zu garantieren und zu erhalten. Außerdem vereinfacht es die effiziente Implementierung von Transaktionen und Persistenz in nicht unerheblicher Weise. Nicht zuletzt machen AGGREGATEs Aspekte eines Modells explizit, die sonst oft gar nicht spezifiziert werden. Zu diesen Aspekten gehören unter anderem der Wirkungsbereich (Scope) von Änderungen. Beispielsweise sollen sich Änderungen am Preis eines Produkts nicht auf schon ausgestellte Rechnungen auswirken. Ein anderer Aspekt betrifft die Eindeutigkeit der IDs von ENTITIES. Da die ENTITIES eines AGGREGATEs, ausgenommen vom «Aggregate Root», außerhalb dieses AGGREGATEs nur temporär referenziert werden dürfen, müssen deren IDs nur innerhalb des AGGREGATEs eindeutig sein.

Data Access Objects DATA ACCESS OBJECTS (DAOs) kapseln das Erzeugen, Laden und Speichern von ENTITIES, VALUE OBJECTS und AGGREGATES. Als Kombination des ABSTRACT FACTORY- [EGV95] und des REPOSITORY-Patterns [Eva04] ermöglichen sie es, vom konkreten Typ und der konkreten Darstellung eines Objekts zu abstrahieren. In Nemoz sind Deskriptoren, wie auch DAOs für Deskriptoren als abstrakte Klassen (Java-Interfaces) realisiert. Beispielsweise können Deskriptor-DAOs beliebig ausgetauscht werden und verschiedene DAOs können Instanzen verschiedener Implementierungen der abstrakten Klasse Deskriptor erzeugen. Dieses Pattern erlaubt eine große Flexibilität bei der Datenhaltung. Es ist zum Beispiel ohne weiteres möglich, die XML-basierte Persistenz der Nemoz-Library durch eine SQL-basierte Persistenz zu ersetzen.

Layered Architecture Die Aufteilung eines Systems in aufeinander aufbauende Schichten, die LAYERED ARCHITECTURE, gehört sicher zu den einfachsten und natürlichsten Strukturierungsweisen für Softwaresysteme, wird allerdings viel zu selten konsequent durchgehalten. Als Folge entstehen Systeme, die kritische Anwendungslogik mit GUI- oder Datenbankcode vermischen und damit extrem schwer zu warten und zu erweitern sind. Eine LAYERED ARCHITECTURE ermöglicht ein effektives und verständliches Domänenmodell durch dessen Trennung von technischem Infratruktur- und GUI-Code, der nichts zum Verständnis des eigentlichen Modells beiträgt. Die Schnittstellen einer jeden Schicht sind klar spezifiziert und „schlank“, die Schichten selbst koppeln lose aneinander. Klassen innerhalb einer Schicht zeichnen sich dagegen durch einen hohen konzeptuellen Zusammenhang aus. Nemoz unterscheidet streng zwischen der GUI-Schicht, der Integrationsschicht, der Service-Schicht und der Schicht des Datenmodells. Die beiden letzten Schichten bilden in Nemoz das Domänenmodell. Einen Sonderfall stellen Architektur- und Infrastruktur-Frameworks dar, welche von allen Schichten verwendet werden. Jede Schicht hängt nur von den Schichten ab, die „unter“ ihr liegen. Zusätzlich sind alle Schichten durch Interfaces spezifiziert und deren konkrete Implementierungen ohne Codeänderungen an anderen Schichten beliebig austauschbar.

Intention-Revealing-Interfaces Das Pattern der INTENTION-REVEALING-INTERFACES erklärt die Kunst, Schnittstellen zu entwerfen, die den Zweck der dahinter liegenden Komponenten und deren Benutzung klar beschreiben. Die Namen der Methoden dieser Schnittstellen können dann in die UBIQUITOUS LANGUAGE einfließen. SIDE-EFFECT-FREE-FUNCTIONS vereinfachen diese Schnittstellen zusätzlich. Ein Beispiel für INTENTION-REVEALING-INTERFACES ist der Nemoz-`Cloakroom-Service`. Er nutzt die Metapher einer Garderobe, um seinen Zweck und seine Benutzung intuitiv verständlich zu machen. Abschnitt 4.4.2 beschreibt diesen Service im Detail.

Side-Effect-Free-Functions Nemoz ist soweit wie mit den gewählten Tools effizient möglich in einem funktionalen/deklarativen Stil entworfen und implementiert. Es nutzt somit das Pattern der SIDE-EFFECT-FREE-FUNCTIONS, das eben diesen Stil empfiehlt, in extensiver Weise. Wie bereits erwähnt, bietet der funktionale Programmierstil einige Vorteile in Bezug auf Lesbarkeit, Korrektheit und Einfachheit des entstehenden Codes und findet breite Anwendung in der Implementierung des Nemoz-Domänenmodells. Aber auch die GUI nutzt deklarative Programmierung zum Beispiel zur Ermittlung der Operationen, die ein Benutzer in einem bestimmten GUI-Zustand ausführen darf.

Design Patterns

Design Patterns, oder *Entwurfsmuster*, stellen erprobte Lösungen immer wieder auftretender Designprobleme in kompakter, generischer und deshalb wiederverwendbarer Form dar. Eine Menge konzeptuell zusammenhängender Design Patterns bildet eine *Pattern Language*. In diesem Endbericht sind die Namen von Design Patterns, wie zum Beispiel MODEL-VIEW-CONTROLLER, in Kapitälchen gesetzt. Design Patterns entstammen ursprünglich der Architektur [AIS77] und wurden erstmals von E. Gamma et. al. [EGV95] in die Softwaretechnik eingeführt, wo sie sich seitdem auf einem recht beispiellosen Siegeszug befinden. Nemoz verwendet Design Patterns aus den verschiedensten Quellen und enthält ein eigenes Framework zur Unterstützung des OBSERVER-Patterns (siehe Abschnitt 4.3.1).

Aspektorientierte Programmierung

Auch wenn das Nemoz-Projekt bisher keine speziellen Tools für die aspektorientierte Softwareentwicklung [Lad03] nutzt, so stützt es sich doch auf einige derer zentralsten Ideen und versucht diese innerhalb der Möglichkeiten der verwendeten Tools umzusetzen. So wird eine *Seperation of Concerns* in Nemoz durch Ausgliederung der „peripheren“ Aspekte wie Persistenz und Transaktionssicherheit in eigene Serviceschichten erreicht. Der `Locator` in Verbindung mit der gesamten Servicearchitektur bietet darüber hinaus die Grundlage für *Inversion of Control* bei der Implementierung der Services.

Extreme-Programming (XP)

Als Softwareentwicklungsprozess kam für Nemoz aus den in den letzten Abschnitten dargelegten Gründen nur ein „agiler“ Prozess in Frage. Der schließlich angewandte Prozess bestand aus den wichtigsten Elementen des Extreme-Programming (XP). Abschnitt 2.5.2 gibt eine kurze Einführung diese Methodologie.

4.1.3. Die Nemoz-„Makroarchitektur“

Die Nemoz-„Makroarchitektur“ überträgt die Prinzipien der in den vorherigen Abschnitten beschriebenen Designphilosophie in die Praxis. Als LAYERED ARCHITECTURE ist sie in mehrere getrennte, aufeinander aufbauende Schichten geteilt, welche in Abbildung 4.3 als konzentrische Ringe dargestellt sind. Die „untersten“ Schichten liegen innen, die „obersten“ Schichten außen. Architektur- und

Infrastruktur-Frameworks, die in allen Schichten Verwendung finden, werden in dieser Abbildung nicht gezeigt, aber in Kapitel 4.3 in aller Ausführlichkeit beschrieben.

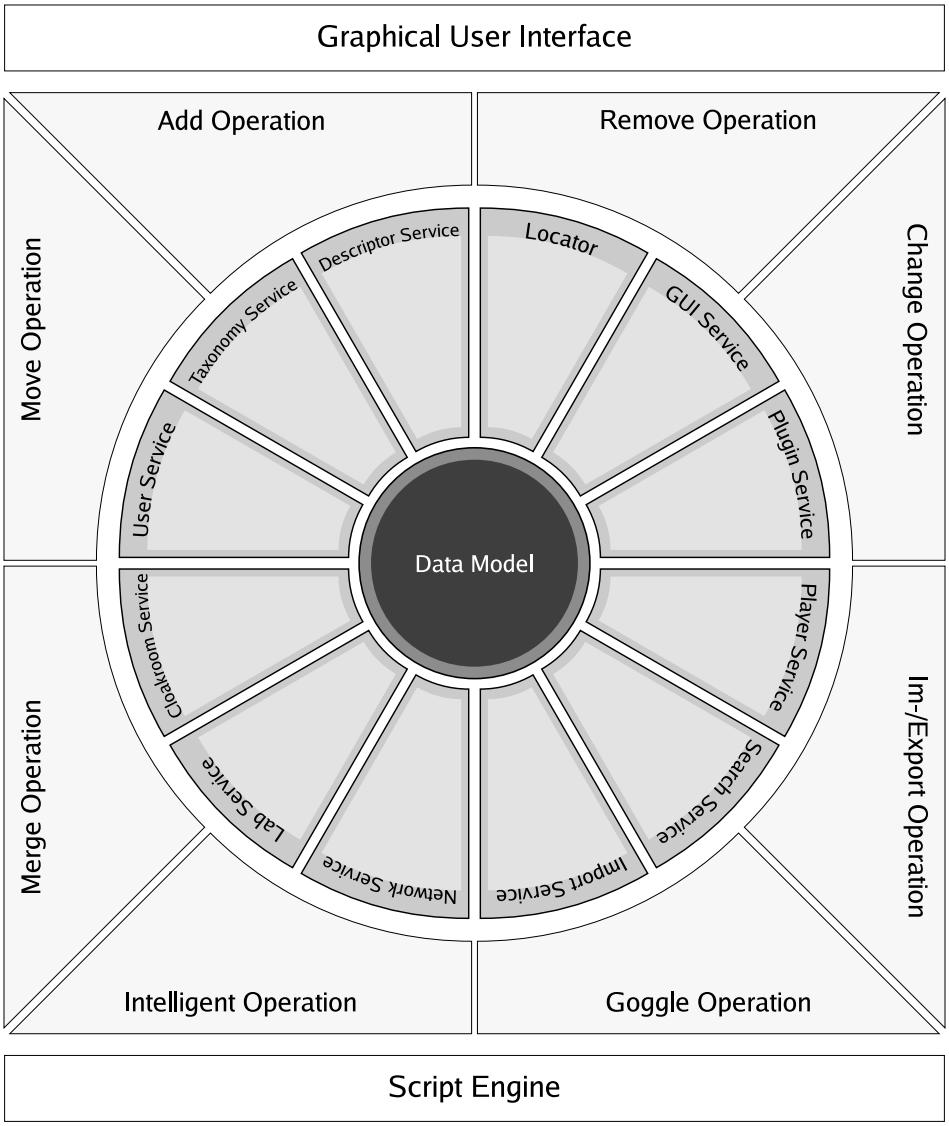


Abbildung 4.3.: Nemoz-„Makroarchitektur“

Das Datenmodell stellt die unterste oder innerste Schicht dar. Es setzt die grundlegenden Konzepte von Nemoz in Software um. So enthält es zum Beispiel ENTITIES, VALUE OBJECTS und AGGREGATES zur Repräsentation von Usern, Libraries, Taxonomien und Deskriptoren.

Die darüber liegende Serviceschicht implementiert Prozesse und Dienste, die sich am natürlichsten in Form von SERVICES darstellen lassen. Dazu gehören zum Beispiel die Suche (`SearchService`), die Netzwerkanbindung (`NetworkService`), die Anbindung an *YALE* (`LabService`) und der Player (`PlayerService`). Der `Locator` nimmt in dieser Schicht eine Sonderrolle ein. Er dient der Kapselung der einzelnen Services von ihren konkreten Implementierungen. Einige Dienste der Serviceschicht, zum Beispiel der `DescriptorService` oder der `TaxonomyService`, erfüllen eine Vermittlerfunktion zwischen dem Datenmodell und den allgemeiner gehaltenen Infrastruktur-Frameworks. Sie helfen bei der Implementierung von Aspekten wie Persistenz und Thread-Sicherheit, ohne dass diese

die Übersichtlichkeit des Datenmodells einschränken müssten. In Verbindung mit der Kapselung dieser Aspekte durch den `Locator` bietet dieser Ansatz einige der wichtigsten Vorteile aspektorienterter Programmierung.

Das Datenmodell bildet gemeinsam mit der Serviceschicht das Nemoz-Domänenmodell. Die Aufteilung der Domänenmodellschicht in zwei eigenständige Schichten ist wegen der inhärenten Komplexität des Anwendungsgebiets notwendig und sinnvoll.

Die Integrationsschicht, welche über der Serviceschicht angesiedelt ist, stellt die *high-level* Funktionalität von Nemoz zur Verfügung. Diese Schicht wird in vielen Architekturen auch als "Anwendungsschicht" bezeichnet. Sie integriert die unter ihr liegenden Schichten, ohne selbst Funktionalität zu implementieren. In Nemoz besteht diese Schicht aus acht generischen Operationen, die jeweils die Serviceschicht und das Datenmodell nutzen, um die an sie gestellten Anforderungen zu erfüllen. Zum Beispiel bietet die `AddOperation` eine Reihe von Methoden für das Hinzufügen von Elementen zum Datenmodell. So wird ein Deskriptor zu einer Taxonomie mit Hilfe einer Methode der `AddOperation` hinzugefügt. Die Integrationsschicht ist also eine API zur high-level Funktionalität von Nemoz über die die darüber liegenden, benutzerzugewandten Schichten auf diese Funktionalität zugreifen. Software, die Nemoz einbetten möchte, würde ebenfalls diese API nutzen.

Die oberste Schicht bildet die graphische Benutzeroberfläche (GUI) oder alternativ die Script Engine. Die GUI zeigt eine aktuelle Sicht auf das Datenmodell und ermöglicht es dem Nutzer, die Operationen der Integrationsschicht bequem auszuführen. Um diese Anforderungen zufriedenstellend erfüllen zu können, benötigt sie eine Reihe eigener Datenstrukturen und unterstützender Frameworks, wodurch sie von ihrem Umfang und ihrer Komplexität her neben der Serviceschicht sicher die aufwändigste Schicht in Nemoz darstellt.

Die Script Engine stellt eine relativ einfache, aber erweiterbare Schnittstelle der Integrationsschicht zur Skriptsprache *Scheme* dar. Sie ermöglicht die Erweiterung der Nemoz-Funktionalität durch Scheme-Skripte, die im laufenden Betrieb getestet und verändert werden können und dadurch „exploratives Programmieren" unterstützen. Als Interpreter findet die freie Implementierung *SISC* [3] Verwendung.

Abschließend soll am Beispiel des Entfernens eines Deskriptors aus einer Taxonomie das Zusammenwirken der einzelnen Schichten illustriert werden.

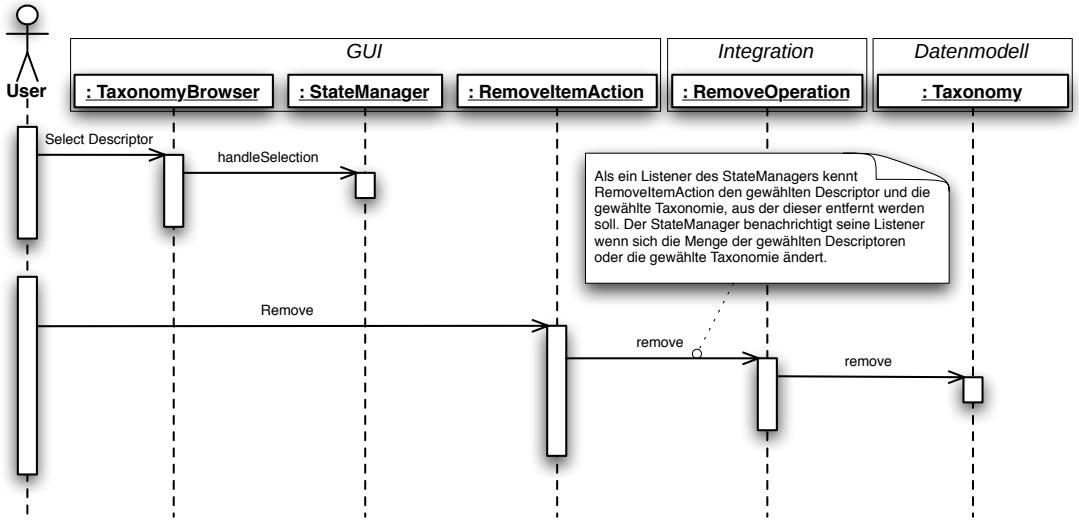


Abbildung 4.4.: Ablaufsequenz beim Löschen eines Deskriptors via GUI

[3] „Second Interpreter of Scheme Code, an extensible Java based interpreter of the algorithmic language Scheme." Siehe auch `http://sisc.sourceforge.net`.

Abbildung 4.4 zeigt die Ablaufsequenz beim Löschen eines Deskriptors mit Hilfe der GUI. Der Nutzer wählt den zu löschenden Deskriptor in der GUI aus, wodurch diese Auswahl im internen Zustand der GUI vermerkt wird. Im nächsten Schritt startet der Nutzer die Löschoperation entweder über einen Menüpunkt, über ein Kontextmenü, über eine Werkzeugleiste oder über eine Tastenkombination. In jedem Fall wird die `RemoveItemAction` aktiv, welche den gewählten Deskriptor und die gewählte Taxonomie kennt und so eine `RemoveOperation` mit den richtigen Parametern ausführen kann. Diese erledigt dann das eigentliche Löschen des Deskriptors aus einer Instanz der Datenmodell-Klasse `Taxonomy`. Services kommen in diesem einfachen Beispiel nicht vor, werden aber von vielen der komplizierteren Operationen der Integrationsschicht verwendet. Am eigentlichen Ablaufprinzip ändert sich dadurch allerdings nichts.

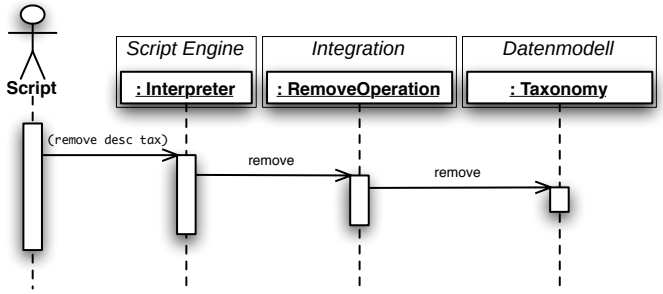


Abbildung 4.5.: Ablaufsequenz beim Löschen eines Deskriptors via Script Engine

Abbildung 4.5 zeigt den gleichen Vorgang bei Verwendung der Skript Engine. An die Stelle des Benutzers tritt ein Scheme-Skript, die Rolle der GUI wird durch die Skript Engine übernommen. Ansonsten ändert sich nichts am Ablauf der Operation.

4.2. Datenmodell (`nemoz.data`)

Wie bereits erwähnt, bilden Datenmodellschicht und Serviceschicht das Domänenmodell von Nemoz. Dieses Kapitel beschreibt die Datenmodellschicht im Detail, im nächsten Kapitel (4.4) folgt die Behandlung der Serviceschicht.

4.2.1. Überblick

Dieses Kapitel enthält eine detaillierte Beschreibung des Datenmodells von Nemoz. Angefangen mit `Descriptor`en und ihrer Verwaltung in der `Library`, führt diese über die `Taxonomy` und der in ihr enthaltenen `TaxonomyNode`s schließlich zum `User` und seiner Speicherung in der `UserList`. Abbildung 4.6 zeigt das Datenmodell im Überblick. Abschließend findet das auf dem DATA ACCESS OBJECT-Pattern basierende System zur Persistenz Erwähnung.

4.2.2. Descriptor

Ein `Descriptor` beschreibt ein beliebiges Datenobjekt. Diese Beschreibung besteht aus *Merkmalen* und *Metadaten*, die aus dem Datenobjekt gewonnen werden, sowie einer Referenz auf dessen Speicherort. Einem Datenobjekt ist genau ein `Descriptor` zugeordnet. Die Eindeutigkeit dieser Zuordnung wird über eine im `Descriptor` enthaltene, aus dem Datenobjekt erzeugte, Id (4.2.3) gewährleistet.

In *Nemoz* werden `Descriptor`en zur Beschreibung von Musikdateien verwendet. Diese spezielle Art von `Descriptor` wird `AudioDescriptor` genannt. `AudioDescriptor`en enthalten Audio-

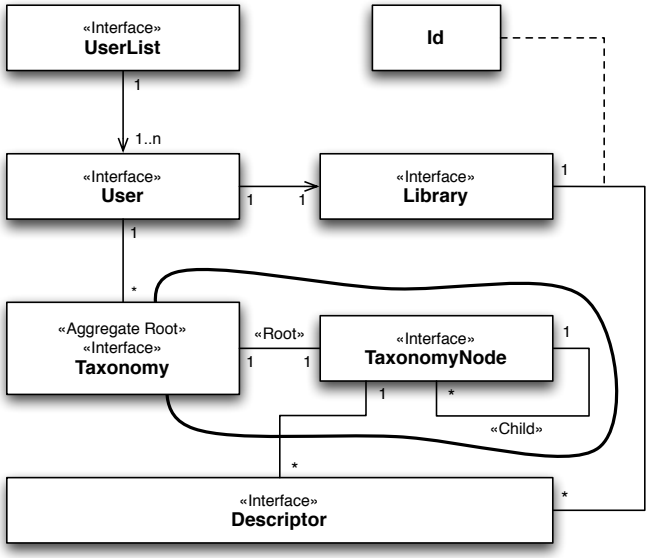


Abbildung 4.6.: Data Model (Überblick)

merkmale, die durch eine Merkmalsextraktion [MKFR03] aus der Wellenform der Musikdateien extrahiert werden, sowie Metainformationen, wie z.B. den Titel oder Interpret der enthaltenen Musikstücke. Die Erzeugung von `AudioDescriptor`en aus Musikdateien ist Aufgabe des `ImportService` und wird in 4.4.4 genauer beschrieben.

Wie bereits in 4.1.1 erwähnt, ist eine der Anforderungen an Nemoz die Verwaltung großer Musikbibliotheken. Um dieser Anforderung gerecht zu werden, gibt es neben dem `Descriptor`, der alle Daten direkt enthält, `FatDescriptor` genannt, den sogenannten `LazyDescriptor`, dessen einziger Inhalt die Id ist. Da `LazyDescriptor`en im Gegensatz zu `FatDescriptor`en sehr klein sind, können sie komplett im Speicher gehalten werden. Anfragen an `LazyDescriptor`en werden an den `DescriptorService` weitergeleitet und von diesem beantwortet. Der `DescriptorService`, dessen Aufgabe die Verwaltung aller `Descriptor`en ist, wird in 4.4.3 detailliert beschrieben.

`Descriptor`en sind modelliert als Value-Objects und daher *immutable*. `Descriptor`en, die hinzugefügt wurden, können nicht verändert, sondern nur komplett ersetzt werden.

Abbildung 4.7 zeigt die Modellierung der `Descriptor`en in Nemoz im Überblick.

4.2.3. Id

Eine `Id` ist eine eindeutige Bezeichnung, die der Identifikation eines Objekts dient.

In Nemoz ist dieses Objekt eine Folge von Bytes, gegeben als eine Datei oder eine Zeichenfolge. Zur Erzeugung der `Id` aus dieser Folge wird eine Hashfunktion verwendet. Die gewählte Hashfunktion sollte vor allem die Kriterien Eindeutigkeit und Effizienz gut erfüllen, um einerseits Fehler bei der Zuordnung zu vermeiden und andererseits zur Laufzeit des Systems ohne großen Ressourcenverbrauch anwendbar zu sein. Für die vorliegende Implementierung wurde die Hashfunktion MD5[4] gewählt.

Ein Einsatzgebiet von `Id`s ist es, eine eindeutige Zuordnung zwischen Musikdateien und `AudioDescriptor`en zu gewährleisten. Jeder `AudioDescriptor` enthält eine `Id`, die aus dem Inhalt der Musikdatei erzeugt wurde. Dies ermöglicht eine Zuordnung, selbst wenn die Datei verschoben wurde. Die Unabhängigkeit der `Id` vom Speicherort der Datei macht es zudem möglich über Systemgrenzen

[4] `http://www.ietf.org/rfc/rfc1321.txt`

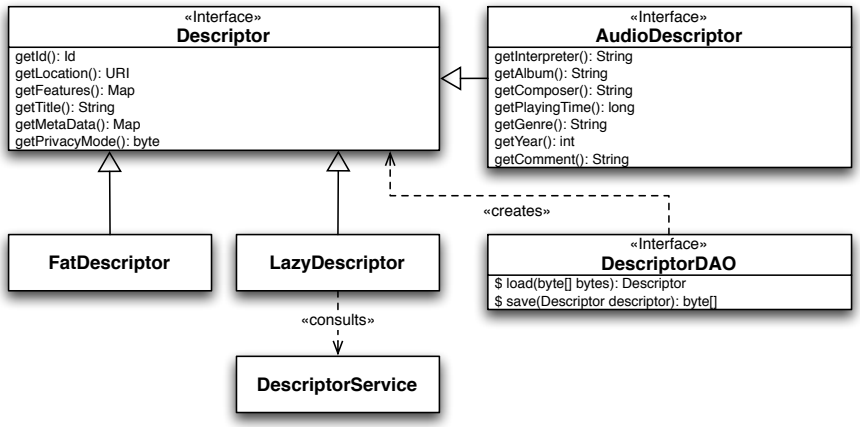


Abbildung 4.7.: Descriptor Model

hinaus, im Netzwerk, Musikdateien eindeutig zu identifizieren.

Weitere Einsatzgebiete für `Id`s sind die eindeutige Identifizierung von Benutzern (4.2.7) und die Zuordnung von Anfragenachrichten und Antwortnachrichten im Rahmen des NetworkService(4.4.8).

4.2.4. Library

Eine `Library` wird, wie die Übersetzung des Wortes als "Bibliothek" schon vermuten lässt, zur Archivierung eingesetzt mit dem Unterschied, dass der Gegenstand dieser Archivierung nicht Bücher, sondern `Descriptor`en sind. Sie verwaltet einen Katalog aller archivierten `Descriptor`en, der es ermöglicht diese unter Angabe ihrer `Id` zu finden.

Aufgrund der verteilten Architektur von Nemoz gibt es zwei Arten von Libraries. Die `LocalLibrary` enthält `Descriptor`en des lokalen Benutzers, während die `RemoteLibrary` als Cache für die `Descriptor`en eines entfernten Benutzers dient, also als Zwischenspeicher für die `Descriptor`en in dessen `LocalLibrary`.

Wird ein neuer `Descriptor` für ein Musikstück erstellt, so entsteht ein `FatDescriptor` und wird in die `LocalLibrary` eingefügt. Diese speichert ihn unter Verwendung des entsprechenden DAOs (siehe 4.2.10) und des Cloakrooms (siehe 4.3.5) auf der Festplatte. So belegen neue `Descriptor`en keinen Speicher und sind darüber hinaus beim nächsten Programmstart verfügbar. Alle Zugriffe auf den Inhalt dieses `Descriptor`s geschehen nun über `LazyDescriptor`en, die mit dessen `Id` initialisiert werden. Wird eines der Attribute eines `LazyDescriptor`s abgefragt, führt dies dazu, dass der entsprechende `FatDescriptor` *just in time* von der Festplatte geladen wird und solange im Speicher gehalten wird, bis die Anfrage beantwortet wurde.

Die Aufgabe der Persistenz entfällt bei der `RemoteLibrary`. Sie hält alle Deskriptoren (`FatDescriptor`en) direkt im Speicher und muss nicht persistent gemacht werden, da sie zum Programmende ihre Gültigkeit verliert.

4.2.5. TaxonomyNode

`TaxonomyNode`s sind die Knoten eines durch eine `Taxonomy` (siehe 4.2.6) repräsentierten Baums von `Descriptor`en. Sie sind ENTITIES, bilden mit einer `Taxonomy` ein AGGREGATE und werden komplett im Speicher gehalten. Eine `TaxonomyNode` hat ein Namen, der für alle Geschwisterknoten eindeutig ist und enthält eine Menge von `TaxonomyNode`s, die mit ihr in einer Elter-Kinder-Beziehung stehen. Eine `TaxonomyNode` darf sich nicht selbst enthalten und die resultieren-

de Struktur muss ein Baum sein. Kindknoten müssen also immer unterhalb ihres Elter angeordnet sein. Zusätzlich kann jede TaxonomyNode eine geordnete Menge von Descriptoren enthalten, wobei jeder Descriptor im Kontext der gesamten Taxonomy nur einmal vorkommen darf. Es besteht die Möglichkeit TaxonomyNodes durch beliebige Attribute auszuzeichnen. Meist sind dies vom LabService verwendete Datenstrukturen wie beispielsweise Klassifikationsmodelle. Diese sind über den YaleTaxonomyNodeDecorator in komfortabler Weise veränderbar.

Aus Gründen der Performanz wird in jeder TaxonomyNode zusätzlich die Menge aller Descriptoren gespeichert, die in dem Teilbaum enthalten sind, dessen Wurzelknoten sie ist: Das sogenannte DescriptorExtensionSet. Die Frage nach dieser Menge muss beispielsweise im Rahmen von Clustering-Experimenten schnell beantwortet werden können, weshalb der zusätzliche Speicherverbrauch in Kauf genommen wurde. Desweiteren enthält jede TaxonomyNode eine Referenz auf ihren Elterknoten und auf die sie enthaltene Taxonomy.

TaxonomyNodes enthalten Locks zur Gewährleistung der Konsistenz auch unter parallelem Zugriff, wobei zwischen lesendem und schreibendem Zugriff unterschieden wird.

Aus Performanzgründen werden Änderungen wie das Hinzufügen von Descriptoren oder das Anfügen neuer Kindknoten an Taxonomies direkt an den TaxonomyNodes durchgeführt, im Widerspruch zu den Regeln zur Implementierung von Aggregates, die unter 4.1.2 beschrieben sind.

Durch die verteilte Architektur von Nemoz wird auch hier zwischen zwei Arten von TaxonomyNodes unterschieden. LocalTaxonomyNodes werden innerhalb von LocalTaxonomies verwendet und werden für Taxonomien des LocalUsers benutzt. Sie unterscheiden sich von der zweiten Art von TaxonomyNode, der RemoteTaxonomyNode vor allem dadurch, dass sie LazyDescriptoren enthalten, die wie unter 4.2.2 erwähnt die *lightweight*-Variante eines Descriptors darstellen, während RemoteTaxonomyNodes FatDescriptoren enthalten. Zudem werden RemoteTaxonomyNodes erst bei Bedarf mit Inhalt, also ihren Descriptoren und Kindknoten, befüllt. Dies ist Aufgabe des NetworkService.

4.2.6. Taxonomy

Eine Taxonomy ist eine ENTITY, die mit TaxonomyNode ein AGGREGATE bildet. Jede Taxonomy enthält zumindest eine TaxonomyNode, der Wurzel des durch sie dargestellten Baums ist. Zudem hat jede Taxonomy einen Besitzer, gegeben durch eine User-Instanz, und einen Namen. Dieser Name ist eindeutig unter allen Taxonomien des Benutzers. Alle Funktionalität zum Verändern der Struktur einer Taxonomy wurde in die TaxonomyNodes ausgelagert, wie schon in 4.2.5 erwähnt wurde. Taxonomien werden zu Usern hinzugefügt bzw. aus diesen entfernt, wobei im Fall des LocalUser der TaxonomyService die Verwaltung übernimmt, die beispielsweise in einer Persistenzfunktion besteht.

Abbildung 4.8 zeigt die Modellierung von Taxonomien im Überblick.

4.2.7. User

Durch User Objekte werden in Nemoz sowohl lokale Benutzer einer Nemoz-Instanz als auch entfernte, durch das Netzwerk verbundene, Benutzer repräsentiert. Der lokale Benutzer wird durch ein LocalUser Objekt repräsentiert. Dieses enthält verschiedene persönliche Daten, eine netzwerkweit eindeutige Id, die Library mit allen Descriptoren des Benutzers und alle seine Taxonomien. Der TaxonomyService ist für diese Taxonomien das, was der DescriptorService für die Descriptoren ist. Er verwaltet alle lokalen Taxonomien und übernimmt dabei Aufgaben, wie beispielsweise Persistenz. Der TaxonomyService wird in 4.4.5 genauer beschrieben.

Entfernte Benutzer werden durch RemoteUser Objekte repräsentiert. Ein RemoteUser stellt eine Momentaufnahmes eines entfernten LocalUsers dar. Er enthält all seine persönlichen Daten, seine Id, eine RemoteLibrary und eine Menge von RemoteTaxonomies. Während durch Nachrich-

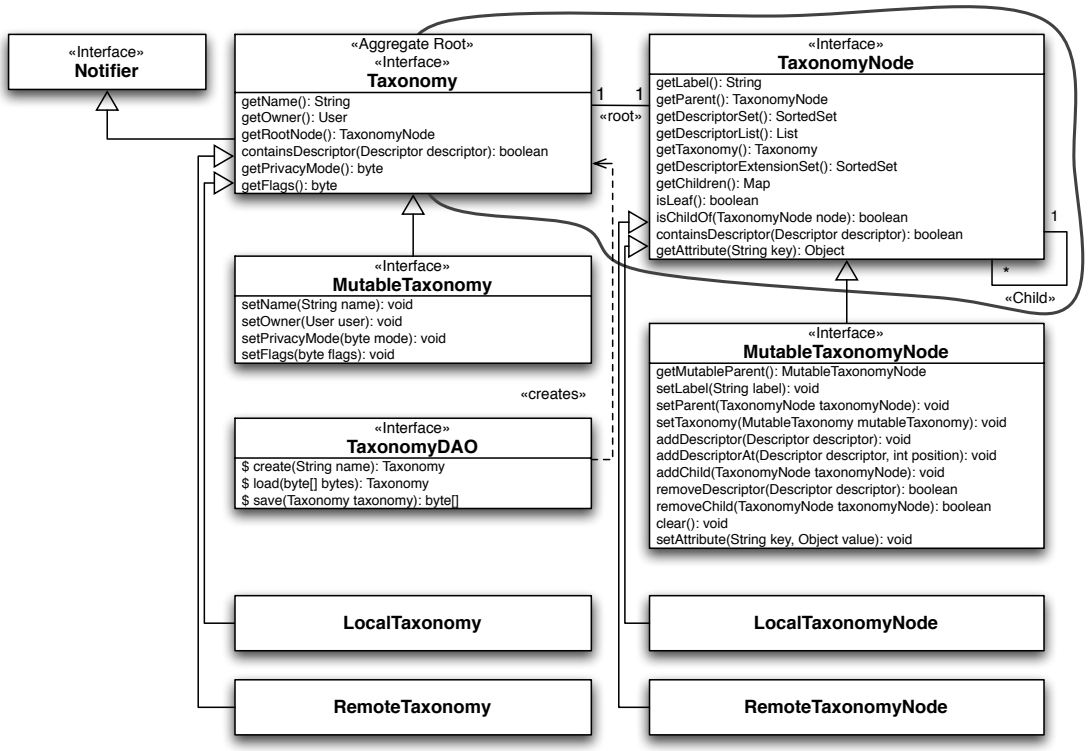


Abbildung 4.8.: Taxonomy Model

ten des `NetzworkService` `RemoteUser` und der zugehörige `LocalUser` bei Änderungen an den persönlichen Daten konsistent gehalten werden, können Änderungen an `LocalLibrary` und bei den `Taxonomien` zu Inkonsistenzen führen. Die `Library` enthält alle `Descriptor`en, die in den `Taxonomien` vorkommen, aber nicht ein komplettes Abbild der `LocalLibrary` des zugehörigen `LocalUsers`, was in einem deutlich reduzierter Netzwerktraffic begründet ist. Da in `RemoteTaxonomies` wie unter 4.2.6 und 4.4.8 beschrieben `TaxonomyNodes` und enthaltene `Descriptor`en nur bei Bedarf übertragen werden, geben diese nicht zwangsläufig die Struktur der zugehörigen `LocalTaxonomies` wieder. Eine genaue Beschreibung der Mechanismen zur Erstellung, Synchronisation und Verwaltung von `RemoteUser` Objekten findet sich im Kapitel über den `NetworkService` unter 4.4.8.

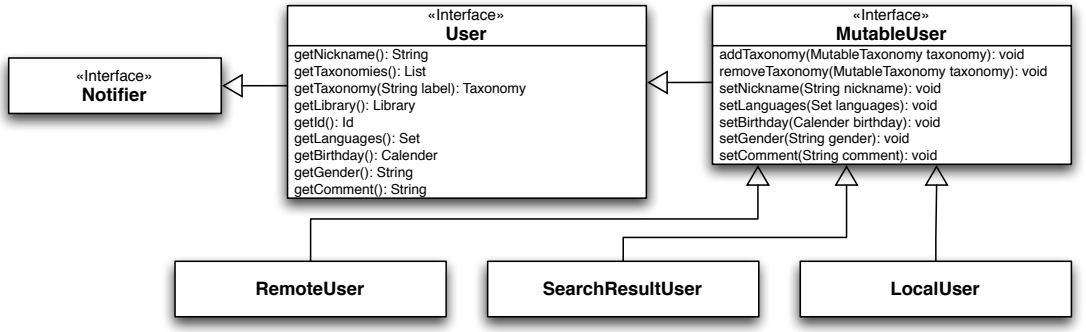


Abbildung 4.9.: User Model

4.2.8. UserList

Die `UserList` ist die *low level* Datenstruktur, die vom `UserService` , der in 4.4.6 beschrieben wird, benutzt wird, um die der lokalen Nemoz-Instanz bekannten `User` zu speichern. Dies sind neben einem `LocalUser` alle durch den `NetworkService` im Netzwerk gefundenen `RemoteUser`. `User` können hinzugefügt und bereits enthaltene entfernt werden, wobei all diese Änderungen über den `UserService` laufen. Vor allem `RemoteUser` werden häufig hinzugefügt oder entfernt, da sie sich über das Netzwerk jederzeit verbinden oder abmelden können.

4.2.9. Path

Während `TaxonomyNode`s und `Descriptor`en der lokalen Nemoz-Instanz direkt als Objekte referenziert werden können, ist dies an den Grenzen, die einerseits zu YALE und andererseits zu per Netzwerk verbundenen anderen Nemoz-Instanzen bestehen, nicht möglich. `Path`s (Pfade) bieten die Möglichkeit der eindeutigen Referenzierung über diese Grenzen hinaus.

Pfade gliedern sich in zwei Arten. Die Struktur eines absoluten Pfads zur Referenzierung eines `Descriptor`s ist `nemoz://UserId:TaxonomyName/NodeLabel/.../DescriptorId`, während durch `nemoz://UserId:TaxonomyName/.../NodeLabel/` eine `TaxonomyNode` referenziert wird. `UserId` ist die unter allen Nemoz-Instanzen im Netzwerk eindeutige `Id` des Besitzers der `Taxonomy`, die das referenzierte Objekt enthält. Sie wird gefolgt vom (pro Benutzer eindeutigen) `TaxonomyName`, einem oder mehreren `NodeLabel`s (Namen von `TaxonomyNode`s) dieser Taxonomie und schließlich (optional) der `DescriptorId`, die für jedes Musikstück eindeutig ist.

Relative Pfade werden relativ zu einer `TaxonomyNode` aufgelöst, werden von Nemoz aber nur intern benutzt. Sie sind von der Struktur `.../DescriptorId` bzw. `.../NodeLabel/`.

Der `PathFinder` dient der Umwandlung von Objektreferenzen auf lokale `TaxonomyNode`s in `Path` Instanzen sowie von `Path` Instanzen, die sich auf die lokale Nemoz-Instanz beziehen, in entsprechende Objektreferenzen und nimmt so eine zentrale Rolle in der Kommunikation mit der „Außenwelt“ ein.

4.2.10. Data Access Objects

DATA ACCESS OBJECTS (DAOS) werden wie schon in 4.1.2 angesprochen dazu verwendet, um die Details der Erzeugung, des Ladens und des Speicherns vom Datenmodell zu trennen. DAOs können ohne Änderungen am Datenmodell ausgetauscht werden und ermöglichen so eine äußerst flexible Anpassung des Persistenzsystems an neue Anforderungen. In Nemoz existieren für alle Elemente des Datenmodells DAOs, wobei die vorliegende Implementierung gezipptes XML für ihre Repräsentation benutzt. DAOs der gleichen Art können untereinander “zusammengesteckt” werden. So kann ein DAO, der Elemente des Typs A verarbeitet, einen DAO für Elemente des Typs B benutzen, sollte er diese innerhalb der von ihm prozessierten Datenstruktur antreffen. Auf diese Art ist eine Exportfunktion für Taxonomien realisiert, die einen `TaxonomyDAO` verwendet, welcher intern für die enthaltene Deskriptoren die Funktionalität eines `DescriptorDAO`s in Anspruch nimmt, um nur ein Beispiel zu nennen.

Die FACTORY `DAOFactory` stellt eine zentrale Anlaufstelle für alle Komponenten von Nemoz dar, die einen geeigneten DAO für ihre Zwecke (in der Regel sind dies Persistenz oder Netzwerkübertragung) benötigen. Aufgrund einer solch zentralen Bereitstellung aller DAOs wirkt sich die Einführung einer neuen DAO-*Familie*, sollte sich dies einmal als notwendig erweisen, sofort auf das ganze System aus.

Ein denkbarer Grund für einen zukünftigen Austausch der aktuellen XML-basierten DAO-*Familie* ist die Verwendung einer JDO [5]-Implementierung anstelle des `Cloakrooms`.

[5] JDO steht für Java Data Objects. Weitere Informationen finden sich unter `http://java.sun.com/products/jdo`

4.3. Architektur- und Infrastrukturframeworks (`nemoz.commons`)

Dieser Abschnitt enthält eine Übersicht über die im Rahmen der Entwicklung von Nemoz entstandenen Architektur- und Infrastrukturframeworks. Architekturframeworks helfen bei der Strukturierung eines Softwaredesigns, Infrastrukturframeworks implementieren „Produktivdienste“. In diesem Sinne sind `Observing`, `Tasks`, `Functions` und `Concurrency` Architekturframeworks, während es sich bei `Indexing` und `Persistence` um Infrastrukturframeworks handelt. Die hier vorgestellten Frameworks gehören keiner speziellen Schicht an (siehe 4.1.3) und werden von beinahe allen Komponenten in Nemoz verwendet.

4.3.1. Observing (`nemoz.commons.observing`)

Das Observing-Framework bietet elegante Unterstützung für das in Nemoz intensiv genutzte OBSERVER-Pattern [EGV95]. Dieses Pattern erlaubt es, Nachrichten über Zustandsänderungen an einem Objekt an andere, von diesem Objekt abhängende Objekte zu übertragen, ohne alle beteiligten Klassen eng aneinander zu koppeln. Nemoz nutzt dieses Pattern unter anderem zur Erhaltung der Konsistenz voneinander abhängender Objekte im Datenmodell, wie auch zur automatischen Aktualisierung seiner GUI-Browser. MODEL-VIEW-CONTROLLER, eine Spezialisierung dieses Patterns, die erstmals im GUI-Framework von *Smalltalk-80* [GR89] eingesetzt wurde, gehört sicherlich zu den bekanntesten Pattern überhaupt. Abbildung 4.10 gibt einen Überblick über die Klassen des Observing-Frameworks.

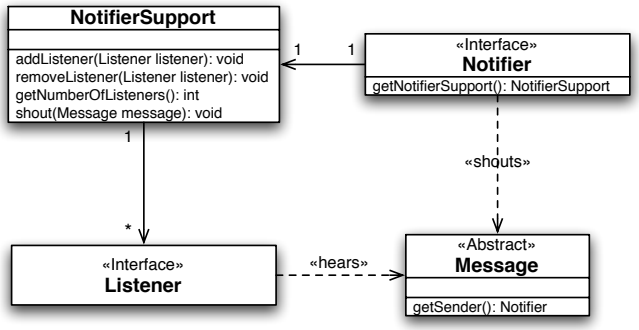


Abbildung 4.10.: Klassen des Observing-Frameworks

Klassen, die andere Klassen über Zustandsänderungen benachrichtigen müssen, implementieren das Interface `Notifier`, welches nur die Methode `NotifierSupport getNotifierSupport()` enthält. Dazu genügt es in einem Attribut eine (finale) Instanz der Klasse `NotifierSupport` vorzuhalten, die dann von `NotifierSupport getNotifierSupport()` zurückgegeben wird. Eigentlich hätte man dieses einfache Verhalten auch in einer abstrakten Klasse `Notifier` kapseln können. Da Java aber nur einfache Vererbung bei Klassen erlaubt, könnten `Notifier` von keiner anderen Klasse mehr erben. Dieses Problem wird durch die Verwendung eines Interfaces umgangen.

Die Klasse `NotifierSupport` verwaltet die Menge der an Nachrichten über Zustandsänderungen interessierten Objekte, der `Listener`. Sie enthält Methoden zum Hinzufügen und Entfernen von `Listener`n, sowie zum Senden („`shout`ing“) von Nachrichten. Diese werden im Observer-Framework als Spezialisierungen der abstrakten Klasse `Message` dargestellt. Diese Spezialisierungen können beliebige Inhalte transportieren.

Das Interface `Listener` selbst enthält keine Methoden, es handelt sich um ein *Marker-Interface*. Implementierende Klassen müssen jedoch für jede Nachrichtenklasse, deren Nachrichten sie empfangen möchten, jeweils eine Methode der Signatur `public void hear(`*`Nachrichtenklasse`* `msg)`

enthalten. Sendet ein `Observer` über seine `shout`-Methode eine Nachricht eines bestimmten Typs (einer bestimmten Klasse), so ruft das Framework alle `hear`-Methoden (registrierter `Listener`) mit einem Parameter dieses oder allgemeineren Typs automatisch auf.

Dieses Verhalten erlaubt es den `Listener`n die für sie interessanten `Messages` in sehr eleganter Weise zu wählen. So enthält Nemoz beispielsweise eine abstrakte `Message`-Klasse `PlayerMessage`, die durch einige konkrete Spezialisierungen wie `PlayerMessage.Playing`, `PlayerMessage.Paused` oder `PlayerMessage.Stopped` ergänzt wird. Diese Spezialisierungen sind der Übersichtlichkeit wegen als innere Klassen in `PlayerMessage` enthalten. Ein hypothetischer `Listener`, der an allen Nachrichten des Players interessiert ist, erhält eine Methode der Signatur `public void hear(PlayerMessage msg)`. Soll ein `Listener` hingegen nur `Paused`- und `Stopped`-Nachrichten empfangen, erhält er zwei Methoden mit den Signaturen `public void hear(PlayerMessage.Paused msg)` und `public void hear(PlayerMessage.Stopped msg)`.

Das Observing-Framework findet „passende“ `hear`-Methoden durch Java-*Reflection*. Es ist dabei aber dennoch hinreichend effizient, da diese Methodensuche nur ein einziges mal durchgeführt werden muss, nämlich beim Hinzufügen eines neuen `Listener`s zu einer `NotifierSupport`-Instanz.

4.3.2. Tasks (`nemoz.commons.task`)

Viele prinzipiell zeitaufwendige Operationen, wie zum Beispiel die Extraktion von Audiomerkmalen, die Wiedergabe von Musikstücken oder das Lernen von Klassifikationsmodellen, werden in Nemoz als Hintergrundthreads ausgeführt. Das Task-Framework bietet die nötige Unterstützung bei der Verwaltung dieser Hintergrundthreads. Abbildung 4.11 gibt einen Überblick über dessen Klassen.

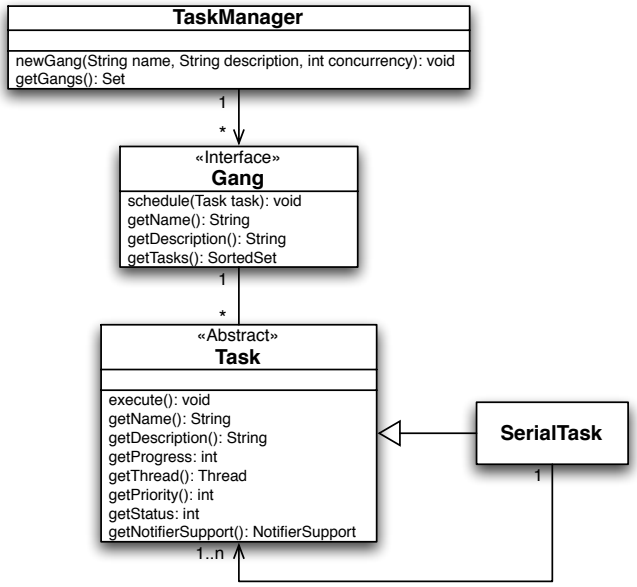


Abbildung 4.11.: Klassen des Task-Frameworks

Hintergrundthreads werden in diesem Framework als Spezialisierungen der abstrakten Klasse `Task` gekapselt. Diese Spezialisierungen müssen die abstrakte Methode `abstract void execute()` implementieren, innerhalb der sie ihre eigentliche Aufgabe durchführen können. `Task`s besitzen einen Namen, eine Beschreibung, einen Status und eine Priorität, wobei ein höherer Integerwert eine höhere Priorität bedeutet. Mit Ausnahme des Status können alle diese Attribute von Spezialisierungen der Klasse `Task` über set-Methoden mit *protected*-Sichtbarkeit selbst gesetzt werden. Der Status, `READY`,

RUNNING oder DONE, wird vom Scheduler bestimmt und kann nur gelesen werden. Tasks nutzen das Observing-Framework (siehe 4.3.1) um interessierte Listener über Änderungen an ihrem Status informieren zu können. Auf diese Weise wird zum Beispiel die GUI benachrichtigt, wenn der Nemoz-Player die Wiedergabe eines Musikstücks beendet hat.

Konzeptuell zusammenhörige Tasks werden in sogenannten Gangs („Arbeitskolonnen“) zusammengefasst, die das Scheduling der ihnen zugeordneten Tasks übernehmen. Ähnlich den Tasks haben auch Gangs einen Namen und eine Beschreibung. Gangs werden mit Hilfe der FACTORY METHOD [EGV95] `Gang newGang(String name, String description, int concurrency)` des TaskManager erzeugt, dabei gibt die *Concurrency* an, wieviele Tasks maximal parallel in dieser Gang arbeiten dürfen. In Nemoz gehören beispielsweise alle Merkmalsextraktionstasks der selben Gang an.

Die Klasse SerialTask ermöglicht es, eine Menge von Tasks bequem in einer gegebenen Reihenfolge hintereinander auszuführen. Dazu wird ihrem Konstruktor ein Array von Tasks übergeben und die enstandene SerialTask-Instanz zu einer Gang hinzugefügt. Wenn deren Scheduler diese SerialTask-Instanz schließlich zur Ausführung bringt, werden die darin enthaltenen Tasks in der durch das Array gegebenen Reihenfolge seriell ausgeführt.

4.3.3. Functions (`nemoz.commons.functions`)

Das Nemoz-Functions-Framework (NFF) implementiert *First-Class Functions* und *Lazy Evaluation* in relativ naiver Weise durch Java-Objekte. Eine einzige abstrakte Klasse, Function stellt die Basis des NFF dar. Spezialisierungen dieser Klasse müssen vier abstrakte Methoden implementieren:

1. `Object eval(final Object[] arguments)` Führt die eigentliche Berechnung des Funktionswerts durch.
2. `String getSymbol()` Gibt den „Namen“, bzw. das Funktionssymbol der Funktion zurück.
3. `Class[] getArgumentSorts()` Gibt die Argumentsorten der Funktion, dargestellt als Array von Classes, zurück.
4. `Class getResultSort()` Gibt die Ergebnissorte der Funktion, dargestellt als Class, zurück.

Eine auf diese Weise erzeugte Function f kann dann mit Hilfe ihrer Methode `Object apply(final Object[] arguments)` auf ein „passendes“ Array von Argumenten angewandt werden. Sind alle Objekte dieses arguments-Arrays definiert und von korrekter Sorte (von korrektem Typ), gibt diese Methode einen Funktionswert der durch die Implementierung der Methode `Class getResultSort()` definierten Sorte zurück. Hat hingegen mindestens eines der Objekte des arguments-Arrays den Wert null, liefert apply eine neue Function f' zurück, die das Ergebnis der partiellen Anwendung von f auf dieses arguments-Array darstellt.

Diese Function f' ist dabei wie folgt definiert: Sei $p_1 \dots p_n$ der Inhalt des arguments-Arrays und $p_{j_1} = \texttt{null} \dots p_{j_m} = \texttt{null}$. Dann gelte:

$$fp_1 \dots p_n := \lambda x_1 \dots x_m . fp'_1 \dots p'_n, \text{wobei}$$

$$p'_i := \begin{cases} x_k & \text{falls } i = j_k \\ p_i & \text{sonst.} \end{cases}$$

Nemoz verwendet das NFF primär bei der Darstellung und Berechnung von Distanzmetriken für die Ähnlichkeitssuche.

4.3.4. Concurrency (`nemoz.commons.concurrent`)

Aufgrund der inhärenten Nebenläufigkeit von Nemoz und aufgrund der dafür in Java 1.4.2 sehr beschränkt vorhandenen Unterstützung ergab sich schnell der Bedarf nach einer Erweiterung in diesem Bereich. Während Java 1.5 diese Erweiterungen von Haus aus mitbringt, wurde für 1.4.2 ein Backport [6] dieser Erweiterungen verwendet. Relevante Funktionalität wie beispielsweise `Locks` und `Semaphoren` wurde direkt in Nemoz integriert.

`Locks` werden verwendet um dem Zugriff mehrerer Threads auf eine Ressource zu kontrollieren. Der Zugriff auf eine Ressource ist nur nach Erlangen des Locks möglich und dies ist nur für einen Thread gleichzeitig möglich, während alle weiteren Threads warten müssen. Es wird zwischen fairen und unfairen Locks unterschieden. Bei fairen Locks bekommt derjenige Thread als nächster das Lock, der als erster danach gefragt hat, während die unfaire Implementierung die Auswahl des Nächsten dem Zufall überlässt. Manchmal wird zwischen lesendem und schreibendem Zugriff unterschieden. So ist lesend gleichzeitiger Zugriff mehrerer Threads möglich, während schreibender Zugriff wiederum nur einem Thread erlaubt ist.

In Nemoz finden Locks beispielsweise im `DescriptorService` und in `TaxonomyNodes` Verwendung. Diese Notwendigkeit entstand daraus, dass häufig gleichzeitig auf diese Datenstrukturen zugegriffen wird. So ist es beispielsweise möglich, dass gerade die in einer `TaxonomyNode` enthaltenen `Descriptoren` gelesen werden, um daraus ein Experiment im `LabService` zu erstellen, aber gleichzeitig über eine `AddOperation` ein `Descriptor` hinzugefügt werden soll. Durch Locks kann ein solcher Zugriff in geregelter Weise erfolgen.

`Semaphoren`, genauer zählende Semaphoren (engl. counting semaphors), ermöglichen die Kontrolle der Anzahl von Threads, die gleichzeitig auf eine Ressource Zugriff nehmen dürfen. Eine Semaphore, die mit 3 initialisiert wurde, erlaubt demzufolge maximal 3 Threads den Zugriff. Alle weiteren Threads müssen warten bis wieder ein Platz frei wird.

Semaphoren bieten grundsätzlich keine zusätzliche Funktionalität im Vergleich zu Locks. Das Gegenteil ist der Fall. Für einen speziellen Anwendungsfall stellten sie aber gerade deshalb eine willkommene Ergänzung dar. Dies soll am Beispiel des ImportService verdeutlicht werden, der in 4.4.4 noch eingehend erläutert werden wird. Bevor ein `Descriptor` hier fertig *zusammengebaut* werden kann, muss erst das Ergebnis eines Experimentes des `LabService` abgewartet werden. Eine Semaphore wird hier dazu verwendet die Ausführung der aktuellen Methode solange anzuhalten, bis das Experimentergebnis verfügbar ist. Dies war mit Locks nicht möglich, da diese bei einem `unlock` den momentanen Besitzer des Locks überprüfen.

4.3.5. Persistence (`nemoz.commons.cloakroom`)

Das Cloakroom-Framework implementiert die Persistenz von Objekten in Nemoz auf der untersten Ebene. Es nutzt die Metapher einer Garderobe (engl. cloakroom), um seinen Zweck und seine Handhabung intuitiv klar zu machen. Wie an einer Garderobe gibt ein „Kunde“ am `Cloakroom` Objekte ab, die er für eine gewisse Zeit nicht benötigt und sicher verwahrt wissen möchte. Beim `Cloakroom` handelt es sich bei diesen Objekten natürlich nicht um Kleidungsstücke, sondern um Arrays von Bytes, die zum Beispiel Taxonomien oder Deskriptoren darstellen. Ähnlich wie an einer Garderobe erhält der Benutzer darauf hin vom `Cloakroom` eine „Garderobenkarte“ in Form eines `long`-Wertes, die er später „vorzeigen“ muss, um sein „abgelegtes Objekt“ zurückzuerhalten. Die Analogie geht dabei sogar soweit, dass einem „Kunden“ mit verlorener „Garderobenkarte“ nun leider wirklich überhaupt nicht weitergeholfen werden kann und trotz ausgiebigen Tests der `Cloakroom`-Implementierung keine Haftung für die darin abgelegten Objekte übernommen wird. Abbildung 4.12 zeigt die Klassen des Cloakroom-Frameworks.

[6] Weitere Informationen finden sich unter `http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html`.

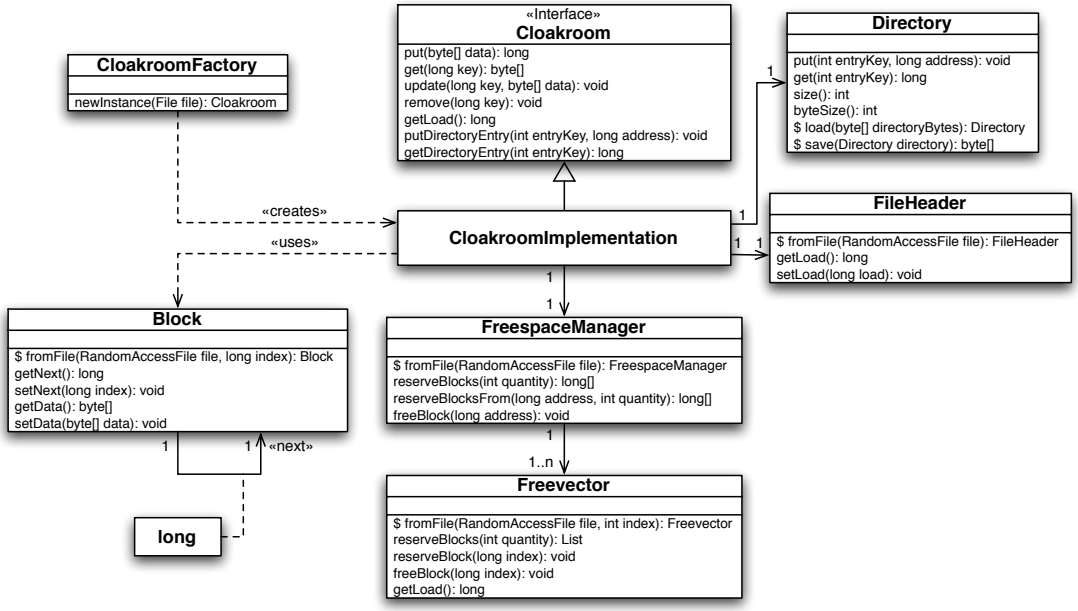


Abbildung 4.12.: Klassen des Cloakroom-Frameworks

`Cloakroom`, `CloakroomImplementation` und `CloakroomFactory` bilden gemeinsam die nötigen Elemente des ABSTRACT FACTORY-Patterns [EGV95], welches den sorgenfreien Austausch der Cloakroom-Implementierung ermöglicht. Denkbar wäre beispielsweise eine Implementierung, die auf einem relationalen Datenbankmanagementsystem basiert.

Das Interface `Cloakroom` bietet Methoden zum Hinzufügen, Entfernen, Holen und Aktualisieren von Einträgen, also abgelegten Objekten. Beim Hinzufügen eines neues Objekts wird eine Adresse in Form eines `long`-Wertes zurückgegeben, die dieses Objekt dann referenziert. Alle anderen der genannten Methoden verlangen eine solche Adresse als Parameter. Zusätzlich wird ein einfaches, flaches, persistentes Verzeichnis unterstützt, welches „wohlbekannte“ Integer-Konstanten auf beliebige Adressen abbildet. So nutzt Nemoz zum Beispiel die Konstante `LIBRARY_INDEX`, um mit Hilfe des Verzeichnisses die Adresse der gespeicherten `Library` des lokalen Benutzers zu finden. Dieses Verzeichnis wird durch die Klasse `Directory` implementiert.

Die derzeit in Nemoz eingesetzte `CloakroomImplementation` verwendet eine Datei mit wahlfreiem Zugriff wie sie durch die Java-Standardbibliothek in Form der Klasse `RandomAccessFile` angeboten wird. Da diese Datei nur endlich groß werden kann und das `Cloakroom`-Interface das Löschen von Einträgen erlaubt, war es nötig einen `FreespaceManager` zu implementieren. Zu diesem Zweck wird der gesamte Freispeicher [7] in *Block*s fester Größe eingeteilt. Der `FreespaceManager` verwaltet nun eine Liste von *Bitvektoren* [SGG00], in denen freie und belegte Blocks in Form gelöschter oder gesetzter Bits verzeichnet sind. Diese Bitvektoren werden durch die Klasse `Freevector` implementiert. Da sie nur endlich groß werden und damit auch nur eine endliche Menge von `Blocks` verwalten können, können sie überlaufen. In diesem Fall wird vom `FreespaceManager` einfach ein weiterer `Freevector` angelegt.

`CloakroomImplementation` speichert ein Array von Bytes, indem sie zuerst dessen Bedarf an Blocks berechnet. Dann sucht sie im ersten nicht völlig belegten `Freevector` nach dem ersten freien Block und ermittelt dessen Adresse. Danach erzeugt sie einen neuen `Block`, setzt in dessen *Header* die Größe des zu speichernden Arrays, füllt diesen mit Nutzdaten, speichert ihn an der ermittelten Adresse

[7] Ein großes Schwimmbecken auf der Spitze des Elfenbeinturms, in welchem früher einmal Elfen zu baden pflegten, das nun aber meist hoffnungslos überfüllt ist mit Trollen und Kobolden, vor allem im Sommer.

ab und markiert diese Adresse als belegt. Sind noch nicht alle Bytes des Arrays gespeichert, wird die Adresse des nächsten freien Blocks auf gleiche Weise ermittelt, diese Adresse im Header des vorherigen `Block`s gespeichert, ein neuer `Block` angelegt, mit Nutzdaten gefüllt, gespeichert und die jeweilige Adresse als belegt markiert. Dieser Prozess wird solange wiederholt, bis das gesamte Array gespeichert ist. Danach wird der letzte `Block` der Kette als „letzter Block“ markiert. Das Aktualisieren von Byte-Arrays verläuft analog zum Speichern.

Ein Array von Bytes an gegebener Adresse wird von `CloakroomImplementation` gelesen, indem der `Block` an dieser Adresse geholt und ein Array der in dessen Header gespeicherten Größe angelegt wird. Danach wird das Array gefüllt, indem die Nutzdaten aller `Block`s der Kette nacheinander hineinkopiert werden, bis der letzte Block der Kette erreicht ist. Darauf hin wird das nun fertige Array zurückgegeben. Das Löschen von Byte-Arrays funktioniert ganz ähnlich.

Die Klasse `FileHeader` kapselt den Header einer Cloakroom-Datendatei. Dieser enthält eine „Magic Number“ anhand derer Cloakroom-Datendateien von anderen Dateien unterschieden werden, sowie die Anzahl der belegten Blocks. Diese Anzahl wird von der `long getLoad()`-Methode des `Cloakroom`-Interfaces benötigt, welche die Größe der gerade belegten Plattenspeichers in Bytes zurückgibt.

Alles in allem zeichnet sich das Persistenz-Framework durch eine recht einfach gehaltene Implementierung aus, die dennoch Zeit- und Speicherplatzeffiziente Unterstützung für Datenmengen im Bereich mehrerer Gigabytes bietet.[8]

4.3.6. Indexing (`nemoz.commons.index`)

Das Indexing-Framework besteht aus Java-Interfaces für String-Indizes und metrische Indizes und einigen Implementierungen dieser Interfaces. Ebenfalls enthalten sind einige DECORATORs [EGV95] für String-Indizes, die zu indizierende Strings auf verschiedene Weisen vorverarbeiten. Abbildung 4.13 gibt einen Überblick über die Klassen des Indexing-Frameworks.

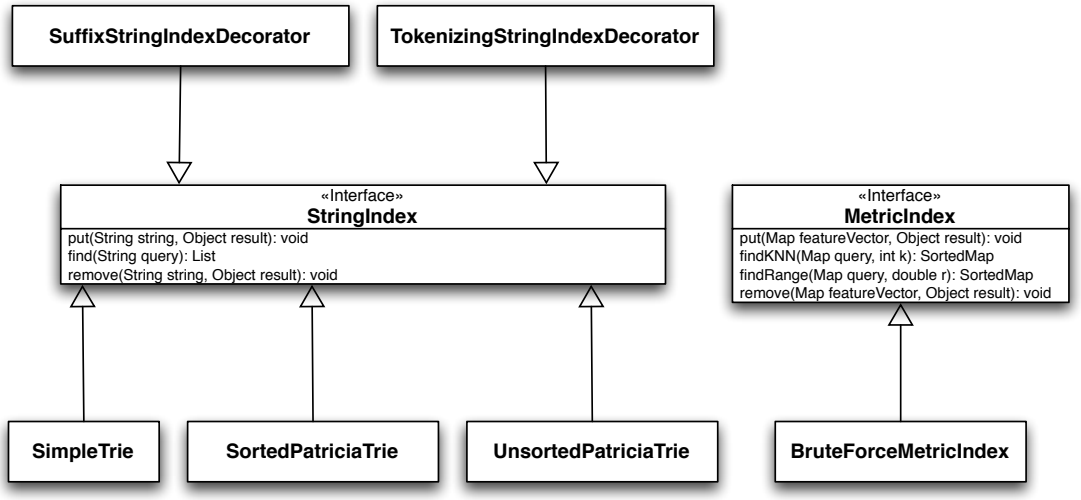


Abbildung 4.13.: Klassen des Indexing-Frameworks

`StringIndex` stellt eine abstrakte Abbildung von `String` nach `Object` dar und bietet Methoden zum Ergänzen dieser Abbildung durch neue Assoziationen, zum Löschen von Assoziationen sowie zum Finden aller mit einem `String` assoziierten `Objects`. `SimpleTrie` ist eine Implementierung dieses

[8] Ein typischer Nemoz-Cloakroom ist allerdings selten größer als einige Dutzend Megabytes.

Interfaces, die eine einfache *Trie*-Struktur nutzt. `UnsortedPatriciaTrie` und `SortedPatriciaTrie` sind ebenfalls Implementierungen dieses Interfaces, die jedoch auf *PATRICIA*s [Knu98], einer erweiterten Trie-Struktur, aufbauen. `SortedPatriciaTrie` gibt die Menge der mit einem `String` assoziierten `Object`s gemäß deren „natürlicher“ (durch Java's `compareTo`-Methoden induzierter) Ordnung sortiert zurück, während diese Rückgabereihenfolge bei `UnsortedPatriciaTrie` nicht definiert ist. Alle beschriebenen String-Indizes werden derzeit vollständig im Hauptspeicher gehalten, eine Erweiterung des Frameworks um *externe* Datenstrukturen ist aber ohne großen Aufwand möglich.

Die DECORATORS `SuffixStringIndexDecorator` und `TokenizingStringIndexDecorator` ermöglichen es, beliebe String-Indizes mit spezieller *Preprocessing*-Funktionalität auszustatten. Ein mit einem `SuffixStringIndexDecorator` versehender `StringIndex` indiziert nicht nur den vom Benutzer gegebenen `String` mit einem gegebenen `Object`, sondern auch dessen Suffixe. Diese Funktion ist nützlich bei der Implementierung einer Teilstringsuche mit Tries. Ein mit einem `TokenizingStringIndexDecorator` „dekorierter“ `StringIndex` indiziert neben dem gegebenen `String` auch alle `Token`, also alle durch Leerzeichen getrennte Worte innerhalb dieses `String`s.

Das Interface `MetricIndex` stellt die Basis aller metrischen Indizes [CPZ97] für die Ähnlichkeitssuche in Nemoz dar. Es handelt sich um die Abstraktion einer Abbildung von Merkmalsvektoren (dargestellt als `Map`s) auf Datenobjekte (dargestellt als `Object`s) und bietet Methoden zum Ergänzen dieser Abbildung um neue Assoziationen, zum Entfernen von Assoziationen sowie zur Bearbeitung von *KNN*- und *Range-Queries*. Derzeit ist dieses Interface nur durch die Klasse `BruteForceMetricIndex`, einer Art „Un-Index“, der jede Anfrage mit Hilfe erschöpfender Suche beantwortet, implementiert. Diese Lösung ist alles andere als optimal und sollte baldmöglichst durch einen *M-Tree* [CPZ97] ersetzt werden. Bei den in Nemoz typischerweise vorkommenden Datenmengen liefert dieser Kompromiss aber Ergebnisse in vertretbarer Zeit.

4.4. Services (`nemoz.services`)

Wie bereits erwähnt besteht das Nemoz-Domänenmodell aus Datenmodellschicht und Serviceschicht. Services sind Abstraktionen von Diensten und Prozessen, die sich schlecht nur einer einzigen Datenmodellklasse zuordnen lassen. Dieser Abschnitt enthält eine detaillierte Beschreibung dieser Services.

4.4.1. Locator

Der `Locator` stellt die zentrale Anlaufstelle für den Zugriff auf alle Services dar. Die Services, deren Schnittstellen über Java-Interfaces definiert sind, werden hier mit ihren Implementierungen *verkabelt*. Dies schafft ein hohes Maß an Flexibilität des Gesamtsystems, da Implementierungen der Services jederzeit ohne Änderungen am Rest des Systems ausgetauscht werden können.

Der `Locator` sorgt beim Start von Nemoz dafür, dass alle Services gestartet werden, wobei hier die Reihenfolge von Bedeutung ist, da Services von anderen Services abhängen können. So benötigen beispielsweise `DescriptorService`, `TaxonomyService` und `UserService` einen fertig initialisierten `CloakroomService`, um auf ihren persistenten Datenbestand zuzugreifen. Für jeden Service existiert eine *Accessor*-Methode. So liefert beispielsweise `getCloakroomService()` den `CloakroomService` zurück.

4.4.2. Cloakroom Service

Der `CloakroomService` ermöglicht gegenseitigen Ausschluss beim gleichzeitigen Zugriff mehrerer Threads auf den eigentlich nicht „threadsicheren“ Nemoz-Cloakroom. Sein Interface gleicht dem des

Cloakrooms und wurde bereits in Abschnitt 4.3.5 ausführlich erläutert.

4.4.3. Descriptor Service

Der DescriptorService, auf der LocalLibrary aufbauend, verwaltet alle Descriptoren des LocalUsers. Diese Funktionalität ist durch die LocalLibrary [9] bereits gegeben.
Der DescriptorService erweitert diese um Locks, um den gegenseitigen Ausschluss bei Modifikationen zu gewährleiten und einen Cache von FatDescriptoren. FatDescriptoren werden zu diesem Cache hinzugefügt, sobald sie einmal aus dem Cloakroom geladen wurden und sind so beim zweiten Zugriff sofort verfügbar. Der Cache passt sich automatisch der aktuellen Speicherauslastung an und verkleinert sich gegebenenfalls.

4.4.4. Import Service

Der ImportService wird zum Importieren von Musikdateien in die durch Nemoz verwaltete Musiksammlung des Benutzers verwendet. Für jede Musikdatei wird dabei ein AudioDescriptor erstellt, der diese in der unter 4.2.2 angegebenen Weise beschreibt. Anschließend wird dieser automatisch hinzugefügt. Bei den Musikdateien muss es sich um Dateien in einem unterstützten Format handeln. In der vorliegenden Implementierung steht vor allem das Mp3-Format im Vordergrund. Nähere Informationen zum gesamten Nemoz-Audiosystem finden sich in der Beschreibung des PlayerService unter 4.4.9. Der ImportService verwendet das Task-Framework, um für jede zu importierende Musikdatei einen sogenannten ImportTask zu starten, der alle zur Konstruktion des AudioDescriptors notwendigen Schritte durchführt. Diese Tasks laufen im Hintergrund ab und währenddessen kann die Arbeit mit Nemoz ungehindert fortgesetzt werden.

Die einzelnen Schritte der Konstruktion eines AudioDescriptors sollen im Folgenden erklärt werden. Die Elemente eines AudioDescriptors sind Audiomerkmale, Metadaten und eine Id. Zunächst wird eine Id aus der Musikdatei erzeugt und durch den DescriptorService überprüft, ob schon ein Descriptor mit dieser Id existiert. Ist dies der Fall, wird der Importvorgang abgebrochen. Um nun die Audiomerkmale zu erhalten, wird mit dem LabService ein Experiment zur Merkmalsextraktion gestartet. Gleichzeitig wird das Nemoz-Audiosystem dazu verwendet, um die Metadaten aus der Musikdatei zu lesen. Sobald dann die Experimentergebnisse der Merkmalsextraktion vorliegen, wird der neue AudioDescriptor erstellt und über den DescriptorService eingefügt.

4.4.5. Taxonomy Service

Der TaxonomyService verwaltet alle Taxonomien des LocalUsers. Neue Taxonomien werden hier hinzugefügt und bestehende können entfernt werden. Zusätzlich sorgt dieser Service für Persistenz. Taxonomien werden bei Initialisierung unter Verwendung des CloakroomService in den Speicher geladen und bei Programmende oder bei Aufforderung durch den Benutzer gespeichert.

4.4.6. User Service

Der UserService fasst alle Aspekte, die Nemoz-User betreffen, unter dem Dach eines Dienstes zusammen. Er erlaubt es RemoteUser, also Repräsentationen entfernter Benutzer über deren ID zu holen. Dazu bedient er sich des Network Services (siehe Abschnitt 4.4.8). Zusätzlich verwaltet er die Liste aller bekannten Benutzer (die UserList) und den lokalen Benutzer. Alle Nemoz-Komponenten, die Zugriff auf lokale oder entfernte Benutzer, deren Taxonomien oder Libraries benötigen, wenden sich also zuerst an diesen Service. Die Persistenz der UserList und des lokalen Users wird ebenfalls

[9] Für eine ausführliche Beschreibung siehe Kapitel 4.2.4.

vom `UserService` ermöglicht, welche dieser mit Hilfe des `CloakroomServices` (siehe Abschnitt 4.4.2) realisiert.

4.4.7. Lab Service

Der `LabService` erweitert die Serviceschicht um Methoden des maschinellen Lernens. Es werden Funktionen zur Verfügung gestellt, die für die *intelligenten* Aufgaben von Nemoz von zentraler Bedeutung sind, wobei theoretisch die gesamte Bandbreite an Funktionalität des im Hintergrund verwendeten YALE-Systems zur Verfügung steht, was Nemoz eine solide Grundlage für zukünftige Erweiterungen bietet. Die momentan von Nemoz verwendeten Funktionen sind die Extraktion von Audiomerkmalen, flaches Clustering sowie das Lernen und Anwenden eines flachen Klassifikationsmodells, wobei dies im Rahmen einer in Nemoz implementierten, hierarchischen Klassifikation, Verwendung findet.

Für jede Art von Experiment definiert der `LabService` das von YALE durchzuführende Experiment. So ist es möglich durch eine minimale Änderung an dieser Stelle beispielsweise die Klassifikation in Nemoz von J48 auf KNN umzustellen.

Eine der Designideen des `LabService` war einem *Auftraggeber* möglichst keine Kenntnis des YALE-Datenmodells abzuverlangen. Dies ist inspiriert von der *Funktionsweise* eines richtigen Labors, wo jemand für ein Blutbild lediglich eine Blutprobe abgeben muss und nichts über die Verfahren wissen muss, die notwendig sind, um dieses zu erstellen. Die Umsetzung dieser Idee erleichtert zum einen die Benutzung der zur Verfügung gestellten Funktionen und sorgt zudem dafür, dass die Umwandlung zwischen diesem und dem Nemoz-Datenmodell an zentraler Stelle geschieht. Soll beispielsweise ein Klassifikationsmodell gelernt werden, so wird dazu der betreffende `TaxonomyNode` übergeben. Die Umwandlung in das für YALE benötigte `ExampleSet` übernimmt der `LabService`. Die Analogie des Labors gilt auch für die Behandlung von Experimentergebnissen, die der *Auftraggeber* nach Zustellung selbst interpretieren muss.

Der `LabService` startet zum Zeitpunkt seiner Initialisierung eine Instanz von YALE. Experimente werden unter Benutzung einer Kombination des Task- und des Observing-Frameworks auf dieser Instanz zur Ausführung gebracht. Dazu wurde die `ExperimentTask` entwickelt. Der Aufruf einer der Methoden des `LabService` führt zur Erstellung einer neuen `ExperimentTask`, die das von YALE auszuführende Experiment enthält. YALE führt alle diese Tasks sequentiell aus. Sobald eine `ExperimentTask` beendet wurde, erhält ihr *Auftraggeber* das Ergebnis des Experiments. Dieses wird unter Verwendung des Observing-Frameworks in einer `Message` zugestellt.

4.4.8. Network Service

Überblick

Der `NetworkService` übernimmt alle Aufgaben, die etwas mit der Kommunikation zu den anderen Clients zu tun haben. Das beinhaltet das Finden anderer Nutzer, das Versenden und Empfangen von Nachrichten und den Dateitransfer. Bei dem Entwurf des `Networkservice` gingen wir zuerst von einer Nutzung im Internet aus. Dazu haben wir zwei verschiedene P2P Systeme betrachtet.

JXTA JXTA sollte ein Framework für die Kommunikation zwischen den einzelnen Rechnern bereitstellen. Mit diesem System sollte man bequem Nachrichten zwischen den einzelnen Benutzern verschicken können, ohne sich um den Netzwerklayer zu kümmern. Leider erwies sich die Programmierung mit diesem System als ziemlich kompliziert. Die Dokumentation (falls vorhanden) war veraltet, mit dem Javadoc konnte man nicht viel anfangen und die Tutorials waren auch nicht sehr hilfreich. Auch die Konfiguration von JXTA war alles andere als trivial. Nach zwei Wochen Testphase war klar, dass dieses System für Nemoz ungeeignet ist.

Jtella JTELLA sollte eine fertige Implementierung des GNUTELLA-Protokolls sein. Leider wurde das Projekt lange vor der Fertigstellung aufgegeben. Das Original (0.7) war veraltet und konnte sich nicht mal zu dem Netz verbinden. Es gab eine Weiterentwicklung, die sich zu dem Netz verbinden konnte und sogar Dateidownload unterstützt hat. Bei näherer Betrachtung stellte sich das GNUTELLA-Protokoll auch als ungeeignet für das Projekt heraus, da die Nachrichtengröße auf 256 Byte Anfrage und 4KB Antwort beschränkt war. Zwar kam die Idee auf, das GNUTELLA-Netz für die Suche nach anderen Benutzern zu verwenden, aber dies wurde schnell verworfen, da man für den Nachrichtenaustausch trotzdem ein eigenes Protokoll verwenden müsste.

JavaSockets Der dritte Anlauf war, die ganze Kommunikation selbst zu schreiben. Java stellt dafür die `Socket` Klassen (und einige sehr schöne Beispiele) zur Verfügung. Zu diesem Zeitpunkt war auch klar, dass sich Nemoz wegen seinen Bandbreitenanforderungen nur in LANs zum Einsatz kommen wird, man also kein Routing braucht und alle Vorteile des UDP-Multicast ausnutzen kann.

Umsetzung

UDP/TCP Für die Kommunikation zwischen den einzelnen Rechnern wird UDP und TCP verwendet. Über UDP werden per Multicast Nachrichten an alle Rechner im Netz verschickt. TCP wird für Punkt-zu-Punkt Verbindungen zwischen den einzelnen Rechnern benutzt. Dafür werden zwei Threads gestartet, die auf eingehende Daten auf Port 10000(UDP) und 20000(TCP) warten. Um die Ports nicht zu blockieren wird dann für jede ankommende Verbindung ein eigener Thread gestartet.

Die Nachrichten Jede Nachricht, die über das Netz verschickt wird, besteht aus einem Header und dem Payload. Im Header stehen Information über den Absender, den Empfänger und die Art der Daten im Payload. Außerdem hat jede Nachricht eine zufällig generierte ID, um sie von den anderen Nachrichten zu unterscheiden und doppelt empfangende Nachrichten zu löschen. Der Header wird von der `MessageProcessor` Klasse des Netzwerkservices analysiert und die Daten im Payload entsprechend weitergeleitet. Manche Nachrichten (z.B. Pings) können sofort bearbeitet werden, andere (z.B. die Suche) werden an andere Services weitergereicht.

Suche nach Usern Wenn sich ein Client zu dem Netz verbindet, muss er sich zuerst bekannt machen. Dazu sendet er eine LOGON-Nachricht mit den Userinformationen (Nickname usw) per Multicast an das gesamte Netz. Jeder andere Client antwortet mit einem Ping, so dass eine vollständige Liste der User im Netz aufgebaut werden kann. Bei der Trennung vom Netz wird auch eine entsprechende Nachricht an alle anderen verschickt, damit sie diesen Client aus der Liste der aktiven User streichen können.

Der Ping Alle 10 Minuten versendet jeder Client eine Ping-Message an alle anderen um mitzuteilen, dass man immer noch online ist. Jeder Client besitzt eine Userliste, in der die Zeit seit dem letzten Ping gespeichert ist. Vergehen 12 Minuten ohne einen Ping zu erhalten wird der User aus der Userliste gelöscht. So kann der Status der User, die sich nicht abmelden konnten, aktuell gehalten werden.

TCP-Verbindung zu anderen Usern Manchmal ist es notwendig eine Nachricht nicht an alle, sondern nur an einen einzelnen Peer zu verschicken. Dazu baut man zuerst eine TCP-Verbindung auf. Ist dies erfolgreich, meldet sich der Sender als „NEMOZ CLIENT“. Der Empfänger antwortet mit dem gleichem String und man kann annehmen, dass zwei Nemoz Clients miteinander kommunizieren wollen. Jetzt gibt es zwei Möglichkeiten. Der Sender kann eine Nachricht verschicken oder einen Dateidownload anfordern.

Browsing Bevor man in den Taxonomien Anderer rumstöbern kann, müssen diese zuerst angefordert werden. Klickt man auf einen User wird die entsprechende Nachricht verschickt. Als Antwort kommt eine Liste der Taxonomien und ihrer Rootknoten. Da die Taxonomien sehr viele Daten erhalten können, werden diese knotenweise übertragen. Erst wenn der User auf einen Knoten klickt, wird dessen Inhalt und die Infos über die Kinderknoten übertragen um so den Datentransfer zu minimieren. Bei der Übertragung wird auch der Privacy-Status beachtet. Taxonomien und Deskriptoren mit dem Status „privat“ werden automatisch gefiltert und gelangen nicht auf den fremden Rechner.

Dateitransfer Wird eine interessante Datei gefunden, kann man versuchen diese herunterzuladen. Dazu schickt man eine entsprechende Anfrage an den Besitzer. Ist der Status der Datei auf „public“, wird diese dann auch übertragen. Dabei hat man zwei Möglichkeiten. Man kann sich ein kleines Stück aus der Mitte sofort anhören oder die ganze Datei auf der Platte speichern. Schlägt der Download von einem Nutzer aus irgendwelchen Gründen fehl, wird eine Suche nach dieser Daten im ganzen Netz gestartet. Bei erfolgreicher Suche wird der Download wiederaufgenommen. Dabei muss nicht von vorne angefangen werden, die schon übertragenen Daten von dem vorherigen Nutzer können weiter verwendet werden. Ist die Suche nicht erfolgreich, wird der Download schlafen gelegt, um das Ganze zu einem späteren Zeitpunkt noch einmal zu versuchen. Wenn nach fünf solcher Versuche die Datei immer noch nicht vollständig da ist, gibt der Downloader auf.

Die Suche Die Suche unterscheidet sich von der anderen Kommunikation, da man auf jede Anfrage auch eine Antwort erwartet und diese Antwort eindeutig zugeordnet werden muss. Damit die Zuordnung klappt wird der `NotifierSupport` benutzt. Man versendet eine Anfrage und gibt dazu die Zeitspanne an, wie lange man auf die Antworten warten will. Kommt eine Antwort, werden die Zuhörer darüber benachrichtigt. Ansonsten tritt irgendwann der Timeout ein und keine weiteren Antworten werden angenommen.

Plugins Das Anbinden von Plugins an den Netzwerkservice ist ziemlich einfach. Der Notifier schickt eine Benachrichtigung wenn eine Nachricht über das Netzwerk ankommt. Jeder Interessierte kann sich also als Zuhörer anmelden und die für ihn interessanten Sachen rauspicken. Eine andere Methode ist es, eine Nachricht zu verschicken und analog der Suche auf die Antwort zu warten.

4.4.9. Player Service

Eine der zentralen Funktionen von Nemoz ist die des Musikplayers. Der `PlayerService` stellt unter Verwendung des Nemoz-Audiosystems eben diese Funktion zur Verfügung. Grob zusammengefasst ist es so möglich die durch die `Descriptor`en in der `Library` oder einer `Taxonomy` des `LocalUser`s repräsentierten Musikstücke in der durch diese Strukturen induzierten Reihenfolge abzuspielen. Um diese Funktionalität umfassend zu beschreiben, wendet sich dieser Abschnitt zunächst dem zugrundeliegenden Audiosystem zu, wobei hier zunächst dessen Funktionen zum Lesen von Audioformaten und dann erweiterte Funktionen, wie der `Player` vorgestellt werden, erläutert dann die Datenstrukturen, die zur Repräsentation von Playlisten verwendet werden und schließt mit der Verwendung dieser Elemente in Kombination.

Ein Notwendigkeit eines eigenen Audiosystem ergab sich, da Java 1.4.2 mit Javasound von Haus aus zwar potentiell leistungsfähig ist, die Implementierung aber an vielen Stellen zu wünschen übrig ließ. Zur Audioausgabe wird die *low-level*-Funktionalität von Javasound benutzt, doch die Dekodierung von komprimierten Audioformaten, wie z.B. Mp3, wird von Nemoz selbst implementiert. Das Nemoz-Audiosystem liefert also einen Strom von Samples, der von Javasound ohne jede Veränderung auf ein Audiodevice geschrieben wird.

Alle Formen von Audioquellen werden durch das Interface `AudioData` behandelt. Es wird zwischen `AudioClip`s, das sind Audioquellen mit einer festen Länge wie z.B. eine Mp3-Datei, und `Audio-`

Streams wie z.B. einem Shoutcast-Stream unterschieden. Für verschiedene Audioformate existieren jeweils Implementierungen dieser Interfaces. So existieren für Mp3-Dateien Mp3AudioClip und Mp3AudioStream. Diese basieren auf einer modifizierten Version der Mp3 Library JLayer [10]. Um trotz der Unzulänglichkeiten der Implementierung die *high-level*-Funktionalität von Javasound, also das direkte Dekodieren verschiedenster Audioformate, nutzen zu können, existiert eine *Brücke* in Form von JavasoundAudioClip und JavasoundAudioStream.

Die FACTORY AudioDataFactory dient dazu für eine Datei oder URL die richtige Implementierung von AudioClip oder AudioStream zurückzuliefern und ist somit zentrale Anlaufstelle zur Nutzung der Dekodierungsfunktionalität des Audiosystems. Sowohl AudioClip als auch AudioStream werden in einer zu den aus Java bekannten InputStreams analogen Weise benutzt. Es können dekodierte Audiosamples byteweise gelesen werden. Eine gegenüber einfachen InputStreams wesentliche und für die Realisierung eines Audioplayers sehr wichtige Erweiterung stellt die *seek*-Funktion dar, welche es in AudioClips ermöglicht an beliebige Positionen zu springen.

Aufbauend auf die im letzten Abschnitt beschriebene *low-level* Audiodecoding-Funktionalität, bietet das Audiosystem einen Player zum Abspielen aller unterstützten Dateiformate. Dieser orientiert sich in seiner Modellierung an einem Cd-Player. Mit open wird eine AudioData-Instanz *eingelegt* und ein Druck auf Tasten wie play, pause,seek und stop hat *cd-player-typische* Auswirkungen. Der Player beschränkt sich auf diese grundliegenden Funktionen. Zusätzliche Funktionalität, wie eine Playlistenfunktion, kann auf höherer Ebene hinzugefügt werden.

Die Unterstützung von Playlisten wurden unter Verwendung des DECORATOR-Patterns [EGV95] für Library und TaxonomyNode implementiert. Eine Playlist kapselt in Nemoz eine Liste von Descriptoren und next()-,previous() – und skip(pos)-Methoden hinzu. Bei Erstellung einer Playlist wird diese Liste mit allen in der Datenstruktur enthaltenen Descriptoren befüllt. Anschließend wird sie durch über das Observing-Framework empfangene Add- und Remove-Nachrichten synchron gehalten. Playlists horchen zudem am TaxonomyBrowser und halten so die Reihenfolge ihrer Descriptor-Liste synchron zur Sortierung in der GUI.

Der PlayerService bietet nun, unter Nutzung aller in den letzten Abschnitten vorgestellten Komponenten die Möglichkeit, eine Playlist der Library oder einer Taxonomy zu erstellen und diese abzuspielen. Während der Player nur die simplen Funktionen open, play, pause, seek und stop aufweist, fügt der PlayerService nun durch next, previous und skipto Funktionen hinzu, um in der Playlist zu navigieren.

4.4.10. Search Service

Der SearchService erlaubt die lokale und netzwerkweite Suche nach Deskriptoren, Taxonomien und Benutzern. Suchanfragen werden als prädikatenlogische Formeln ohne Negation und Quantoren dargestellt. Abbildung 4.14 gibt einen Überblick über die Klassen des SearchService.

Der SearchService beantwortet SearchQuerys mit SearchResults. Da die Ergebnisse einer netzwerkweiten Suche innerhalb eines längeren, nur durch einen internen *Timeout* beschränkten Zeitraums eintreffen können, erfolgt die Abgabe von Suchanfragen asynchron. Eine Methode des SearchService gibt die bisher eingetroffenen Surchergebnisse zurück. Zusätzlich können interessierte Komponenten durch das Observing-Framework (siehe Abschnitt 4.3.1) über neu eingetroffene Suchergebnisse benachrichtigt werden.

Ein SearchQuery ist ein Predicate, die Konjunktion zweier SearchQuerys (dargestellt als AndQuery) oder die Disjunktion zweier SearchQuerys (dargestellt als OrQuery). Ein Predicate ist ein IsQuery, ein ContainsQuery, ein GreaterQuery, ein LessQuery oder ein SimilarityQuery. Is- und ContainsQuerys kapseln Suchanfragen nach bestimmten Stringmustern, wie sie zum Beispiel bei der Suche nach allen Musikstücken eines bestimmten Interpreten oder der Suche

[10] Eine modifizierte Version von JLayer wurde fest in Nemoz integriert. Weitere Informationen zu JLayer finden sich unter http://www.javazoom.net/javalayer

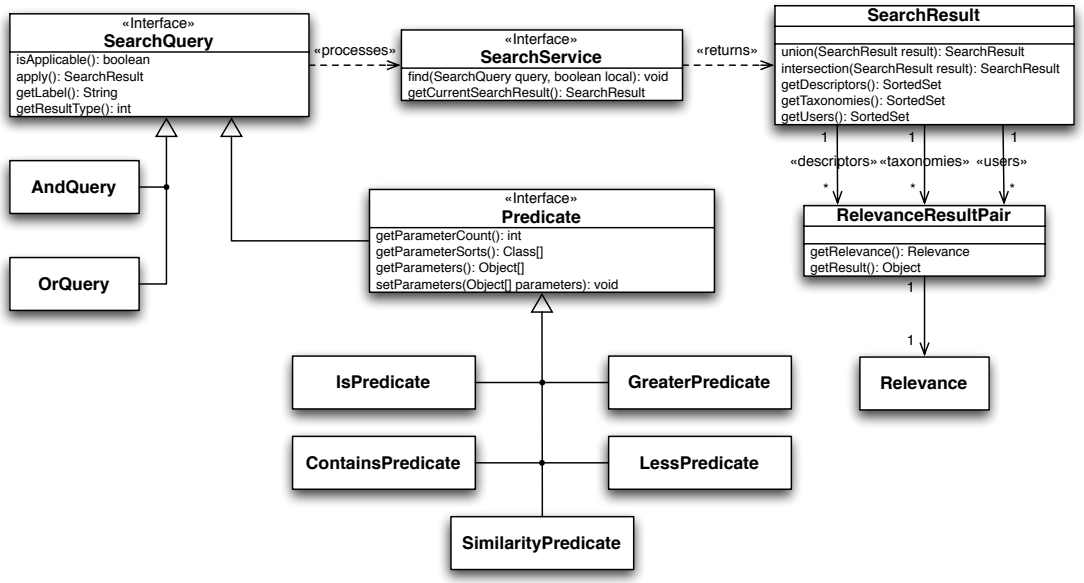


Abbildung 4.14.: Klassen des Search-Service

nach allen Taxonomien, deren Name das Wort „Rock“ enthält, auftreten. `Greater-` und `LessQuerys` stellen Suchanfragen nach Zahlenwerten dar, wie zum Beispiel die Suche nach allen Musikstücken kürzer als 2 Minuten oder die nach allen Benutzern älter als 18 Jahren. `SimilarityQuerys` sind eine Besonderheit von Nemoz. Sie stellen Suchanfragen nach Musikstücken oder Taxonomien dar, die ähnlich zu einem bestimmten Musikstück oder einer bestimmten Taxonomie sind.

Da der Search-Service die gleichzeitige Suche nach Deskriptoren, Taxonomien und Benutzern unterstützt, enthält `SearchResult` drei Ergebnislisten, je eine pro Ergebnistyp. Jeder diese Listen enthält Paare von Ergebnis und Relevanz (`RelevanceResultPairs`) und ist nach der Relevanz sortiert. Diese Relevanzen werden durch „opaque“ VALUE OBJECTS der Klasse `Relevance` dargestellt und sind für jeden Querytyp gesondert definiert. Derzeit hat die Relevanz nur bei der Ähnlichkeitssuche eine Bedeutung, hier steigt sie mit der „Ähnlichkeit“ des Ergebnisobjekts zum Anfrageobjekt. Die Relevanzen der Ergebnisse aller anderen Anfragetypen unterscheiden sich nicht.

4.4.11. Plugin Service

Nemoz bietet für einfache Erweiterbarkeit ein Pluginsystem an. Der `PluginServices` ist verantwortlich dafür, alle vorhandenen Plugins zu laden und zentral zur Verfügung zu stellen. Ein Beispiel-Plugin samt Anleitung und fertig vorbereiteter, auf ANT basierender *build*-Umgebung findet sich im Unterverzeichnis `plugins` im Nemoz-CVS.

4.4.12. GUI Service

Der `GuiService` stellt ein API [11] für Plugins und Erweiterungen dar. Er besteht aus dem Interface „`GuiService`“ und einer simplen Implementierung dieses Interfaces. In dieser Klasse wurde das Pattern des SINGLETONS angewandt. Es gibt also nur eine `GuiService`-Instanz, auf die alle anfragenden Klassen zugreifen. Über diverse `get`-Methoden kann man bestimmte Elemente des GUI errei-

[11] (engl. application programming interface) Eine Programmierschnittstelle ist die Schnittstelle, die ein Softwaresystem anderen Programmen zur Verfügung stellt.

chen. Möchte man z.B. einen neuen Menüpunkt in die `MainMenuBar` einfügen, so erreicht man den `GuiService` über `GuiServiceImplementation.instance();` Nun kann man die Methode `getMainMenuBar()` aufrufen und erhält als Rückgabe die `MainMenuBar` von Nemoz. Da es sich auch bei der `MainMenuBar`, wie bei allen GUI Elementen die nur einmal gebraucht werden, um einen SINGLETON handelt, kann man nun Veränderungen an der `MainMenuBar` vornehmen. Beispielsweise kann man nun also einfach einen weiteren Menüpunkt einfügen. Ähnlich kann man mit allen weiteren GUI Elementen verfahren. Dies sind im Einzelnen: Der `MainFrame`, die `MainMenuBar`, die `MainToolBar`, die `PlayerToolBar` und der `StateManager`. Die Abbildung 4.15 zeigt den schematischen Aufbau des `GuiService`. Näheres zu den einzelnen GUI Elementen und deren Verbindungen findet man im Kapitel 4.6.1. Die einzelnen Elemente werden außerdem ausführlich im Handbuch beschrieben.

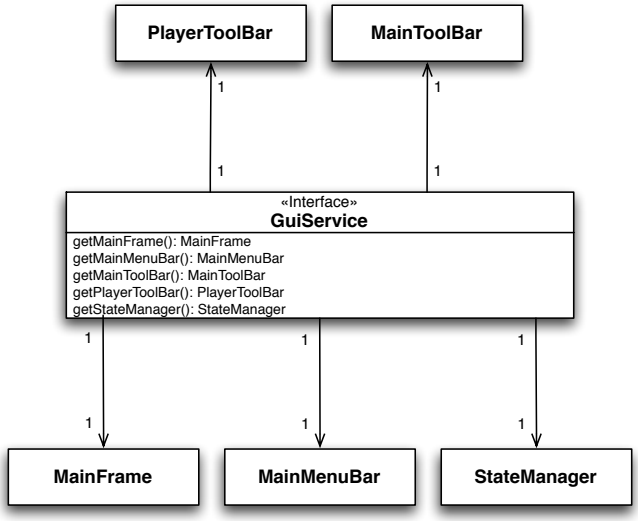


Abbildung 4.15.: Der GUI Service

4.5. Integrationsschicht (`nemoz.operations`)

Im Package `edu.udo.cs.ai.nemoz.data.operations` befinden sich mehrere Klassen, in denen jeweils ähnliche Operationen zusammengefasst wurden. Sinn und Zweck dieses Packages ist es eine Integrationsschicht zu bilden. In den einzelnen Klassen sind alle Operationen vereint, die für bestimmte Aufgaben benötigt werden. Sie stellen eine enorme Vereinfachung dar: Anstatt direkt auf den Datenstrukturen zu arbeiten, werden die einzelnen Operationen mit den entsprechenden Parametern aufgerufen. Um alles weitere kümmert sich dann die jeweilige Operation. Der Vorteil liegt also darin, dass man kein spezielles Wissen über die Datenstrukturen benötigt, um diese zu modifizieren. Möchte man z.B. einen Deskriptor zu einer `TaxonomyNode` hinzufügen, so ruft man lediglich die `add` Methode der Klasse `AddOperation` auf und übergibt dieser den Deskriptor und den `TaxonomyNode`. Die Operation arbeitet dann mit den entsprechenden Services zusammen um die gewünschte Operation auszuführen. Ähnlich verhält es sich bei allen anderen Operationen. Man wählt einfach die entsprechende Operation aus, übergibt die erforderlichen Parameter und der Rest der Arbeit wird von der Operation erledigt. Welche Operationen man genau ausführen kann, wird in diesem Abschnitt beschreiben.

4.5.1. Add Operation

Im Package `edu.udo.cs.ai.nemoz.data.operations` findet sich die Klasse `AddOperation`. In dieser Klasse sind alle Operationen vereint, die Elemente zu lokalen Datenstrukturen hinzufügen. Wann immer man also ein Element, z.B. einen Deskriptor, hinzufügen will, so greift man auf die Methoden dieser Klasse zurück. Der Benutzer aktiviert einen Button in dem GUI [12] oder gibt einen Befehl über die Skript Engine [13] ein. Hierdurch wird die entsprechende Operation mit einigen Parametern aufgerufen. Diese arbeitet dann direkt auf dem Domänenmodell, welches in den Kapiteln 4.2 und 4.4 beschrieben wird.

Mit Hilfe der `AddOperation` ist es möglich folgenden Operationen auszuführen:

- Das Hinzufügen eines `FatDescriptor`s zu der `Library` des lokalen Benutzers.
- Das Hinzufügen eines `Descriptor`s zu einem `TaxonomyNode`.
- Das Hinzufügen eines `LazyDescriptor`s zu einem lokalen `TaxonomyNode`.
- Das Hinzufügen eines `LazyDescriptor`s zu einem lokalen `TaxonomyNode`, und zwar hinter einen übergebenen `LazyDescriptor`.
- Das Hinzufügen eines `FatDescriptor`s zur lokalen `Library` und des entsprechenden `LazyDescriptor`s zu einem lokalen `TaxonomyNode`.
- Das Hinzufügen eines `TaxonomyNode`s zu einem anderen `TaxonomyNode`.
- Das Hinzufügen einer Kopie eines lokalen `TaxonomyNode`s zu einem lokalen `TaxonomyNode`.
- Das Hinzufügen einer Kopie eines lokalen `TaxonomyNode`s zu einer lokalen `Taxonomy`.
- Das Hinzufügen einer `Taxonomy` zum lokalen Benutzer.
- Das Hinzufügen einer `Taxonomy` zu einem Benutzer.
- Das Hinzufügen einer Kopie eines entfernten `TaxonomyNode`s zu einem lokalen `TaxonomyNode`.

Die Methode hat dabei immer den Namen `add`, erst über die entsprechenden Parameter wird bestimmt, um welche Operation es sich speziell handelt. Möchte man also z.B. einen Deskriptor zu einer lokalen Taxonomie hinzufügen, so ruft man die Methode `add(final Descriptor descriptor, final TaxonomyNode taxonomyNode)` mit den beiden Parametern auf. In diesem Fall handelt es sich bei den beiden Parametern um abstrakte Datentypen. In jeder Methode erfolgen selbstverständlich diverse Kontrollabfragen, sodass Fehler früh erkannt und abgefangen werden können.

4.5.2. Move Operation

Zum Verschieben von Elementen gibt es im gleichen Package, die `MoveOperation`. Mit Hilfe dieser Operation kann man:

- Einen lokalen `TaxonomyNode` in einen anderen lokalen `TaxonomyNode` verschieben.
- Einen lokalen `TaxonomyNode` in die `Taxonomy`-Liste des Benutzers verschieben.
- Einen `Descriptor` von einem lokalen `TaxonomyNode` in einen anderen lokalen `TaxonomyNode` verschieben.

[12] siehe Kapitel 4.6.1
[13] siehe Kapitel 4.6.2

- Innerhalb eines TaxonomyNodes einen LazyDescriptor hinter einen anderen LazyDescriptor verschieben.

Analog zur AddOperation wird auch hier die Methode immer move genannt.

4.5.3. Remove Operation

Die Klasse RemoveOperation bietet Operationen, um Elemente aus lokalen Datenstrukturen zu löschen. Folgende Operationen werden dazu bereitgestellt:

- Das Löschen eines Descriptors aus der Library des Benutzers. Zuvor wird der entsprechende Descriptor aus allen lokalen Taxonomien des Benutzers gelöscht.
- Das Löschen eines Descriptors aus einer bestimmten Taxonomie.
- Das Löschen eines TaxonomyNodes aus einer Taxonomie.
- Das Löschen eines Descriptors aus einem TaxonomyNode.
- Das Löschen einer Taxonomie.
- Das Löschen aller Descriptoren und Kindknoten eines TaxonomyNodes.

Auch hier haben die Methoden immer den gleichen Namen, was die Benutzung erheblich vereinfacht. Man schreibt also lediglich remove und übergibt als Argument das entsprechende Element, das entfernt werden soll, evtl. mit weiteren Parametern, z.B. der Taxonomie, aus der es entfernt werden soll.

4.5.4. Change Operation

Die ChangeOperations helfen bei der Änderung von bestimmten, häufig geänderten Werten. Dies ist im Einzelnen:

- Das Ändern des Privacy Modes eines Descriptors.
- Das Ändern des Privacy Modes einer Taxonomie.
- Das Umbenennen einer Taxonomie.
- Das Umbenennen eines TaxonomyNodes.

4.5.5. Merge Operation

In dieser Klasse sind alle Mengenoperationen vereint. Diese können allerdings nur auf lokale TaxonomyNode angewandt werden. Man erhält jeweils eine neue Taxonomie, in der das Ergebnis der Operation gespeichert wird. Es können dabei die folgenden Operationen angewandt werden:

- Die Vereinigung [14] zweier TaxonomyNodes.
- Der Durchschnitt [15] zweier TaxonomyNodes.
- Der Differenz [16] zweier TaxonomyNodes.

[14] Die Vereinigung besteht dabei aus der Vereinigung der abgeflachten TaxonomyNode, d.h. ohne Unterstrukturen. Es werden also lediglich die Deskriptoren vereinigt.
[15] Der Durchschnitt ist dabei analog zur Vereinigung definiert.
[16] Die Differenz ist dabei ebenfalls analog zur Vereinigung definiert.

4.5.6. Intelligent Operation

In dieser Klasse findet man die sog. intelligenten Funktionen. Die beiden Operationen dieser Klasse sind:

Das *Clustern* [17] eines `TaxonomyNode`s. Zum Clustern wird ein neuer Task erzeugt, der die verschiedenen Cluster berechnet. Hierzu wird an die Klasse `FlatClusteringTask` im Package `edu.udo.cs.ai.nemoz.learning` der zu clusterden Knoten und die gewünschte Anzahl der Cluster übergeben. Diese Klasse übernimmt nun die eigentliche Arbeit, während der Kontrollfluß bei der Operation verbleibt. Parallel dazu wird in der Klasse `FlatClusteringTask` ein `ClusterResultListener` erzeugt, der das Ergebnis des Clusterings entgegennimmt. Nun wird ein ClusterExperiment erzeugt und zum `LabService` (4.4.7) übersendet. Dieser berechnet ein `FlatClusterModel` und sendet nach erfolgreicher Arbeit eine `ExperimentSucceeded` Nachricht an den Listener. Der Listener nimmt das `FlatClusterModel` entgegen und modifiziert darauf basierend den zu clusternden Knoten, d.h. er sortiert die Deskriptoren dem Clustermodell entsprechend ein. Dazu greift er selbstverständlich auf die beschriebenen Operationen, wie z.B. `Add` zurück. Die Änderungen am Knoten werden dann über die im GUI Kapitel 4.6.1 beschreibene `stateChanged` Methode propagiert, sodass alle Anzeigeelemente auf den aktuellen Stand gebracht werden können.

Das *Klassifizieren* [18] in `TaxonomyNode`s. Die Methode sortiert `Descriptor`en in eine gegebene Taxonomie ein. Dazu benutzt sie das Klassifikationsmodell, welches in jedem einzelnen Knoten gespeichert wird. Ist in einem Knoten kein Klassifikationsmodell gespeichert, oder das gespeicherte Modell ist veraltet, so wird ein Experiment gestartet um ein neues Modell zu berechnen. Das neue Klassifikationsmodell wird dann in dem entsprechenden Knoten gespeichert. Im Moment wird eine recht triviale Lösung verwendet: Jedes Modell wird neu berechnet, bevor es angewandt wird. Diese Strategie soll aber durch eine bessere ersetzt werden, die eine Neuberechnung nur durchführt, falls diese notwendig ist, d.h. sich der Inhalt des Knoten verändert hat.

Wie funktioniert nun also das Einsortieren mit einem Klassifikationsmodell? Die Methode `classify` der Klasse `IntelligentOperation` erzeugt einen neuen `HierarchicalClassificationTask`. Diese Klasse findet sich im Package `edu.udo.cs.ai.nemoz.learning`. Der `HierarchicalClassificationTask` versucht nun die `Descriptor`en einzusortieren. Hierzu erzeugt er eine Liste von `ClassificationStep`s, ein Step besteht dabei aus einer Menge von `Descriptor`en und einem Zielknoten. Durch diese Liste läuft der Algorithmus nun durch: Zunächst wird für jeden Step geprüft, ob es sich bei dem aktuellen Knoten um ein Blatt handelt, er also keine weiteren Kinder hat. Ist dies der Fall, so werden die entsprechenden Deskriptoren eingefügt. Handelt es sich bei dem Knoten um kein Blatt, so wird zunächst das oben beschriebene Update-Prozedere durchgeführt. Es wird also ein neues Modell für den Knoten gelernt. Hierzu wird die Methode

```
scheduleFlatClassificationModelLearningExperiment(this, destination)
```

des `LabService`s (4.4.7) ausgeführt. Dabei wird der Methode als Parameter der aktuelle Knoten und der Zielknoten übergeben. Nun wird auf das Ergebnis des `LabService`s gewartet, dies geschieht durch die Verwendung einer Semaphore. Wenn es berechnet wurde, wird vom `LabService` eine `ExperimentSucceeded` Nachricht versandt. Die Klasse `ClassificationStep` hat sich beim `LabService` als Listener registriert und empfängt diese Nachricht. Nun wird das Modell auf die `Descriptor`en angewendet. Diese geschieht über die Methode

```
scheduleClassificationExperiment(this,destination,descriptors)
```

des `Labservice`s. Wieder muss mit Hilfe der Semaphore auf das Ergebnis gewartet werden. Ist das Experiment beendet, empfängt man wieder eine `ExperimentSucceeded` Nachricht. In der entsprechenden `hear` Methode des `ClassificationStep`s wird nun versucht die `Descriptor`en dem

[17] Eine ausführliche Beschreibung findet sich im Kapitel 2.3.7.
[18] Eine ausführliche Beschreibung findet sich im Kapitel 2.3.1.

berechneten Modell entsprechend einzusortieren. Gelingt dies nicht mit allen Descriptoren, müssen die restlichen weiterverarbeitet werden. Es werden nun also abschließend weitere Steps erzeugt, die in den folgenden Durchläufen bearbeitet werden. Dies wiederholt sich solange, bis alle Descriptoren einsortiert wurden.

4.5.7. Goggle Operation

Mit Hilfe dieser Operation ist es möglich sich die Descriptoren einer Taxonomie in der Struktur einer anderen Taxonomie zu betrachten. Es wird also eine neue Taxonomie erzeugt, aus der Struktur der einen Taxonomie und mit den Descriptoren der anderen. Eine sehr praktische Funktion, da man so einfach und schnell z.B. seine eigene Musik aus dem Blickwinkel eines anderen Benutzers betrachten kann.

4.5.8. Import Operation

Eine mit der ExportOperation gespeicherte Datei kann mit der ImportOperation wieder importiert werden. Die Methode importXmlFile(File f) erkennt dabei selbst, ob es sich um einen exportierten Descriptor oder um eine Taxonomy handelt.

4.5.9. Export Operation

Die ExportOperation ermöglicht schließlich:

- Das Exportieren der gegebenen Taxonomy als XML Datei.
- Das Exportieren eines gegebenen Descriptors als XML Datei.

4.6. Benutzerschnittstelle

4.6.1. GUI

In diesem Kapitel wird das Konzept, das sich hinter dem Graphical User Interface (GUI) verbirgt, beschrieben. Die Grundphilosophie der GUI läßt sich durch die Begriffe *Direktheit*, *Uniformität* und *Zustandslosigkeit* gut umreissen, [Ung91] gibt eine detaillierte Einführung in die konkrete Bedeutung dieser Begriffe im Kontext graphischer Benutzeroberflächen.

Eine ausführliche Beschreibung der eigentlichen GUI und ihrer Komponenten aus der Sicht des Benutzers findet sich im Handbuch.

Actions

Das Kernstück des GUI-Konzepts bilden die Actions im Package nemoz.gui.actions. Die generelle Idee dabei war, dass die Actions sich weitestgehend selbst verwalten. Es handelt sich hierbei um Klassen, die sowohl von der Klasse AbstractAction erben, sowie das Interface Listener implementieren. Durch die Klasse AbstractAction erhalten die jeweiligen Actions einige Datenfelder, wie z.B. einen Namen, ein Icon [19], eine Long_Description und ein Tastenkürzel. Dies wird alles in dem jeweiligen Konstruktor der Action gesetzt. Wichtig ist, dass es sich bei allen Actions um SINGLETONs handelt, es gibt also jeweils immer nur eine einzige Action, die beim Systemstart erzeugt

[19] Die Icons entstammen alle dem freien Icon Set Nuvola http://www.icon-king.com/goodies.php

wird. Über die Methode `public static final ClusterAction instance()` [20] erhält man die jeweilige Instanz der Action. Diese kann man dann z.B. in ein Menü einfügen. Dies Vorgehen ist sehr komfortabel. Man kann sehr schnell Menüs aufbauen, indem man einfach die jeweiligen Actions über `Action.instance()` einfügt. Da die Actions selbst ihren Namen und ihr Icon verwalten, braucht man keine weiteren Einstellungen vornehmen.

Damit die Actions sich selbst verwalten können ist es notwendig, dass sie das Interface `Listener` implementieren. Somit können sie nun selbst entscheiden, wann sie aktiv sind und wann nicht. Hierfür hat nahezu jede Action eine `public void hear(final StateMessage message)` Methode. Diese Methode verarbeitet Nachrichten die vom `StateManager` [21] gesendet werden. Die Action bestimmt in dieser Methode anhand des Zustandes des GUI ob sie aktiviert ist, d.h. also, dass sie ausgeführt werden kann, oder nicht. Die oben erwähnte `ClusterAction` ist z.B. aktiviert, wenn eine lokale Taxonomie und genau einer ihrer Knoten und kein Deskriptor ausgewählt ist. Außerdem darf das `ExtensionSet` [22] nicht leer sein. Wenn all diese Bedingungen erfüllt sind aktiviert sich die Action, ist dies nicht der Fall, so deaktiviert sie sich. Eine solche Überprüfung wird immer ausgeführt, wenn der `StateManager` eine Nachricht verschickt, d.h. also immer wenn der Zustand des GUI sich ändert; z.B. wenn ein anderer Knoten mit der Maus ausgewählt wird.

Die `public void actionPerformed(ActionEvent evt)` Methode ist ebenfalls eine Methode die jede Action hat. Hier befindet sich der eigentlich auszuführende Code, wenn die Action ausgewählt wird. D.h. der Benutzer hat durch klicken des Buttons, drücken des Tastenkürzels oder ähnliches die Action gewählt. Nun wird die eigentliche Operation [23] die sich hinter der Action verbirgt, evtl. mit einigen Parameter, aufgerufen. Wenn beispielsweise die `ClusterAction` ausgeführt wird, so erscheint zunächst ein `JOptionPane` [24], in dem die Anzahl der Cluster ausgewählt werden können. Nach dem der Benutzer den „OK“ Button gedrückt hat, wird die `IntelligentOperation.cluster(final LocalTaxonomyNode taxonomyNode, final int numberOfClusters)` Methode aufgerufen. Dabei wird dieser Methode der gerade ausgewählte Taxonomieknoten[25] übergeben, sowie die gewünschte Anzahl der Cluster.

StateManager

Der `StateManager` verwaltet, wie bereits eingangs erwähnt, den Zustand des GUI. D.h. er speichert in mehreren Listen die ausgewählte Quelle, ausgewählte Taxonomien, Ordner und Deskriptoren, sowie den Zustand einiger andere GUI Elemente. Über einige öffentliche Methoden gibt er diese Listen an andere Klassen, z.B. die oben beschriebenen Actions, weiter.

Er befindet sich im Package `edu.udo.cs.ai.nemoz.gui.states`. Es handelt sich bei dieser Klasse ebenfalls um einen SINGELTON, da man nur einen Zustandsspeicher benötigt. Die Klasse implementiert die Interfaces `ActionListener`, `TreeSelectionListener`, `Notifier` und `Listener`. Dadurch, dass es sich um einen Listener handelt, kann der `StateManager` an den verschieden GUI Elementen lauschen und auf Veränderungen reagieren. Ändert sich z.B. die Auswahl im `TaxonomyBrowser`, so wird die Methode `protected void handleSelection(final TreePath selectionPath)` aufgerufen und die Änderungen werden in die entsprechenden Listen eingetragen. Über die Methode `public void stateChanged()` wird diese Veränderung nun an die Listener, die am `StateManager` registriert sind, weiter propagiert, sodass z.B. wie oben beschrieben die Actions prüfen können, ob sie aktiviert werden müssen.

[20] Hier am Beispiel der `ClusterAction`
[21] Im Kapitel 4.6.1 findet sich eine Beschreibung des `StateManagers`.
[22] siehe 4.2.5
[23] Die einzelnen Operationen werden im Kapitel 4.5 erläutert.
[24] `http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/JOptionPane.html`
[25] Den ausgewählten Taxonomieknoten erhält man vom `StateManager`.

Adapter

Im Package `edu.udo.cs.ai.nemoz.gui.adapter` befinden sich einige ADAPTER. Diese haben alle die gleiche Aufgabe: Sie bilden die Schnittstellen der Datenstrukturen und die der GUI ab. Der `TaxonomyAdapter` adaptiert z.B. eine `MutableTaxonomy` zu einem `TreetableModel`, welches dann von der `Treetable` angezeigt werden kann.

Ansichten

Nemoz bietet zwei Alternativen zur Darstellung von Taxonomien:

Treetable Eine Treetable ist eine Kombination aus einem Baum und einer Tabelle. Es ist also möglich, wie in einer Tabelle, mehrere Zeilen und Spalten anzuzeigen. Zusätzlich können Knoten nun Kinder haben, sodass eine Baumstruktur entsteht. Java selbst bietet keinen Treetable als Standardkomponente, allerdings kann man einfach einen `JTree` als Renderer für die Zellen einer `JTable` verwenden und schon hat man eine Treetable. Philip Milne beschreibt dieses Vorgehen ausführlich in dem Tutorial „Creating TreeTables in Swing“ [26].

Treemap Eine Treemap [27] ist eine raumabhängige Visualisierung von hierarchischen Strukturen. Es werden gleichzeitig alle Ebenen und Knoten angezeigt. Mit Hilfe einer Treemap kann man sehr gut Attribute von Blattknoten, in diesem Fall von Deskriptoren, darstellen, indem man die Form und die Farbe der Knoten variiert. Bei Nemoz bedeutet dies, dass die Farbe der einzelnen Deskriptoren von den extrahierten Audiofeatures bestimmt wird. Treemaps erlauben es dem Benutzer Knoten und Subbäume zu vergleichen, auch wenn diese sich auf unterscheidlichen Ebenen befinden. Somit ist es in dieser Ansicht gut möglich Muster, aber auch Ausreisser, zu erkennen. Die Handhabung der Nemoz Treemap wird im Handbuch beschrieben.

Die Treemap wurde von Ben Shneiderman [28] in den 1990er Jahren entwickelt. Einige aktuelle Projekte beschäftigen sich ebenfalls mit Treemaps. Newsmap [29] stellt z.B. die Informationen, die Google News sammelt und strukturiert, in einer Treemap dar.

4.6.2. Script Engine

Die gesamte Funktionalität der Nemoz-Integrationsschicht ist Scheme-skriptbar. Diese Funktion basiert of SISC [30], einem freien Scheme-Interpreter für Java.

Was ist Scheme?

Scheme [31] ist eine moderne Lisp-Variante, die auf konzeptuelle Einfachheit und Ausdrucksstärke hin optimiert wurde. Während viele Programmiersprachen Feature über Feature schichten, kommt Scheme mit einer minimalen Syntax; einer minimalen Menge von Features aus. Scheme ist ein bisschen wie Lego - Man kann eigentlich alles damit bauen und alles was man damit baut schaut nach Lego aus. Lisp, und damit Scheme, ist in gewisser Weise „reine Mathematik“: Es basiert auf dem „Lambda-Kalkül“, einem Kalkül rekursiver Funktionen, der in den 1930er Jahren von Alonzo Church und Stephen Kleene entwickelt wurde. Scheme-Code schaut mit seinen vielen runden Klammern auf den ersten Blick seltsam

[26] Creating TreeTables in Swing `http://java.sun.com/products/jfc/tsc/articles/treetable1/`
[27] siehe auch `http://www.cs.umd.edu/hcil/treemap/`
[28] siehe `http://www.cs.umd.edu/hcil/treemap-history/`
[29] siehe `http://www.marumushi.com/apps/newsmap/`
[30] siehe `http://sisc.sourceforge.net`
[31] siehe `http://www.schemers.org/`

und fremd aus, die Klammern haben jedoch ihren Sinn und werden von einem im Rahmen des Nemoz-Projekts entwickelten Eclipse-Plugins gut in Schach gehalten. Dieses Plugin bietet einen Scheme-Editor, der u.a. zueinander gehörende Klammern hervorhebt und Code automatisch sinnvoll einrückt.

Während Softwareentwicklung in „Mainstream-Sprachen“ wie Java oder C++ in einem „Edit-Compile-Run“-Zyklus verläuft, erlaubt Scheme das interaktive Inspizieren und Editieren von Code und Daten in einem laufenden Programm. Dies vereinfacht „Rapid Prototyping“ und „exploratives Programmieren“ enorm. Scheme ist eine „programmierbare Programmiersprache“: In Scheme ist Code ein Datentyp „wie jeder andere auch“. Makros erlauben es, neue Sprachkonstrukte hinzuzufügen, neue Mittel der linguistischen Abstraktion zu finden und auf diese Weise „domänenspezifische Sprachen“ zu definieren. Oft wird in Scheme “funktional“ programmiert, aber auch „imperativer Stil“, OOP und Logikprogrammierung ist konsistent möglich. Die Sprache hat eine recht grosse, aktive und sehr hilfsbereite Benutzerbasis.

Scheme und Nemoz

Die in Nemoz integrierte Scheme-Script Engine ermöglicht es neue Ideen und Anwendungen für Nemoz durch exploratives Programmieren schnell und einfach prototypisch umzusetzen. Dies wird durch eine rudimentäre, aber erweiterbare Scheme API, sowie einen GUI-Toplevel in Form der Nemoz Console ermöglicht.

Die Scheme-API besteht aus den Modulen `model.scm`, `services.scm` und `operations.scm`, die das Domänenmodell, die Services und, auf höhererer Ebene, die Integrationsschicht kapseln. Die

```
(define pastimes (make-taxonomy "Pastimes"))
(add! pastimes)
```

Abbildung 4.16.: Erstellen und Hinzufügen einer lokalen Taxonomie via Script Engine

Abbildung 4.16 zeigt wie eine neue lokale Taxonomie namens „Pastimes“ auf Modell-Ebene erstellt und danach auf Integrationsschicht-Ebene unter Zuhilfenahme der generischen `add!` Operation zum lokalen Benutzer hinzugefügt wird. Dieses einfache Beispiel unterstreicht eine wichtige Eigenschaft der Nemoz Script Engine: Im Gegensatz zu vielen klassischen Skriptsystemen ermöglicht sie nicht nur das Scripting der GUI, sondern auch aller anderen Schichten, bis hin zu allen Funktionen der Java-Laufzeitumgebung. Um „einfache Dinge einfach und schwierige Dinge möglich zu machen“ ist die Scheme-API selbst in Schichten aufgeteilt, denen die oben genannten Module entsprechen.

Die Nemoz Console der GUI erfüllt zwei verwandte Aufgaben: Sie stellt Nachrichten des Log-Systems in einer Liste dar und bietet eine interaktive Benutzerschnittstelle (Toplevel) zur Script Engine. Scheme-Skripte werden direkt im laufenden Programm eingegeben (bzw. per `load`-Kommando aus einer Datei geladen) und sofort ausgewertet. Eventuelle Rückgabewerte werden durch das Log-System dargestellt.

Beim Start von Nemoz führt die Script Engine das Skript `init.scm` aus, welches vom Benutzer als „Hook“ zur Erweiterung von Nemoz und zum Nachladen weiterer eigener Skripte verwendet werden kann.

5. Experimente

Was Nemoz hauptsächlich von anderen Multimediamanagern unterscheidet sind diejenigen Funktionen, die auf Methoden des Maschinellen Lernens beruhen. In diesem Kapitel gehen wir näher auf diese Problematiken ein. Nach einer kurzen Einführung der verwendeten Experimentierumgebung, beschäftigen wir uns zunächst mit der Extraktion von Merkmalsvektoren aus Audiodaten. Danach erläutern wir die verschiedenen Vorverabeitungsschritte, die wir eingesetzt haben, um uns schließlich den Lernaufgaben selbst zu widmen: Klassifikation und Clustering.

5.1. Die Experimentierumgebung Yale

Yale [Mie04] (Yet Another Learning Environment) [1] wurde am Lehrstuhl für Künstliche Intelligenz der Universität Dortmund entwickelt, um eine Plattform für Anwendungen des Maschinellen Lernens zu haben. Der Schwerpunkt von Yales liegt auf Vorverarbeitung und Klassifikation. So sind auch alle Weka [2]-Lerner eingebunden. Desweiteren stellen Plugins z.B. Clustering und Wertereihenfunktionalität zur Verfügung.

5.2. Vorverarbeitung

5.2.1. Merkmalserzeugung

Audiodateien als solche bilden aufgrund ihres immensen Informationsgehaltes keine sinnvolle Grundlage für Lernverfahren. Aus diesem Grund werden in einem Vorverarbeitungsschritt gezielt Merkmale aus ihnen extrahiert, die zu einem reelwertigen Vektor endlicher Länge (dem Merkmalsvektor) zusammengefaßt werden. Alle Methoden des Maschinellen Lernens werden nun auf diesem Merkmalsvektor ausgeführt. Für eine genauere Einführung in die Merkmalsextraktion siehe z.B. 2.4.2 oder [Mie04].

Die Anzahl von Merkmalen, die aus einem Musikstück extrahiert werden können, ist unendlich. Vor der Merkmalsextrahierung muss also eine feste Menge an Merkmalen ausgewählt werden, auf die man sich schließlich stützt. Das Problem ist zum einen die Größe des Merkmalsraums, und zum anderen die Tatsache, dass für unterschiedliche Probleme verschiedene Merkmalssätze unterschiedliche Ergebnisse liefern. So mag eine bestimmte Menge an Merkmalen z.B. für die Unterscheidung zwischen Klassik und Pop sehr gut geeignet sein, führt bei der Unterscheidung von Rock und Metal aber zu schlechten Ergebnissen.

Merkmalskonstruktion mittels genetischer Programmierung

Die Struktur der Merkmalsbäume und die Größe des Merkmalsraums bedürfen eines Suchverfahrens, das für solche Probleme geeignet ist. [MM05] benutzt und empfiehlt genetische Programmierung. Für die Hintergründe zu genetischer Programierung siehe z.B. [BNKF98].

[1] http://www-ai.cs.uni-dortmund.de/SOFTWARE/YALE/index.html
[2] http://www.cs.waikato.ac.nz/ml/weka/

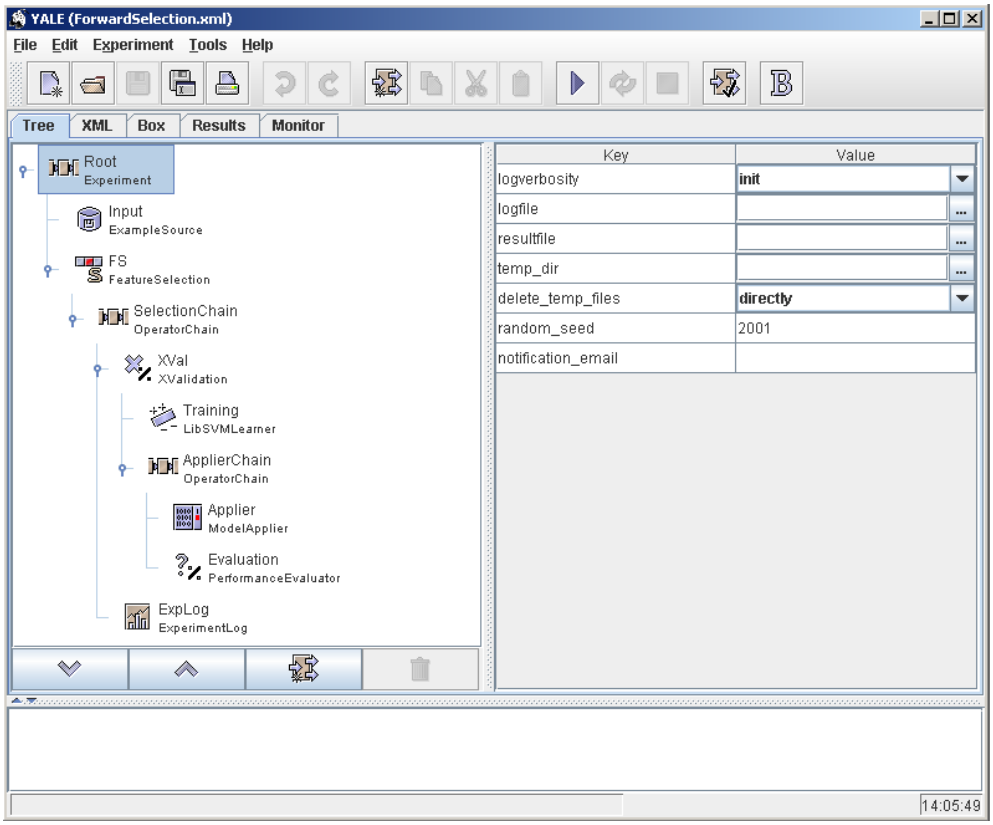


Abbildung 5.1.: Die Experimentierumgebung Yale

Verwendete Merkmalssätze

Für die Klassifikation in Nemoz wird ein Merkmalssatz benötigt, der auf beliebigen Problemen hinreichend gut funktioniert. Zwar ist es auch denkbar, für jedes Problem in Nemoz selbst einen eigenen Merkmalssatz zu konstruieren, doch sind die Verfahren dazu zu zeitaufwendig, als das sie für den Benutzer akzeptabel wären.

Zwei Merkmalssätze waren bereits vorhanden: zum einen der in [Mie04] entwickelte (*'Mierswa'*), zum anderen der in [MUT+05] entwickelten (*'Mörchen'*).

Desweiterem wurde von einigen PG Teilnehmern Musikstücke nach (*Genre*, *Tempo*, *Color*) vorklassifiziert. Die Klassifizierung in *Genre* beinhaltet die Klassen: (*Blues*, *Pop*, *Metal*, *Rock*). Die Musikstücke in *Tempo* wurden nach den Klassen (*Schnell*, *Mittel*, *Langsam*) sortiert. Die Aufteilung in *Color* besteht aus den Klassen (*Accoustic*, *Electric*, *Electronic*, *Orchestra*). Für diese Klassifizierungsaufgaben konstruierten wir, mit den oben gennaten Verfahren Merkmale und vereinigten alle Merkmalssätze der drei Klassifizierungsaufgaben, mit denen von Mierswa und Mörchen und erhielten so einen großen Merkmalssatz bestehend aus 144 Merkmalen(*'Fat'*).

Das YALE-Experiment zur Merkmalserzeugung zeigt Abb. 5.2.

Die Accuracy der einzelnen Klassifizierungsaufgaben mit den besten Individuen(Merkmalssätze) ist in Tabelle 5.1 zu sehen. Bei der Accuracy-Berechnung wurde eine zehnfache Kreuzvalidierung durchgeführt.

Wie in der Tabelle zu sehen ist, sind diese Ergebnisse nicht zufrieden stellend. Der Grund für diese schlechten Werte liegt darin, dass zu Beginn der Experimente, die Lernaufgabe aus mehreren Klassen

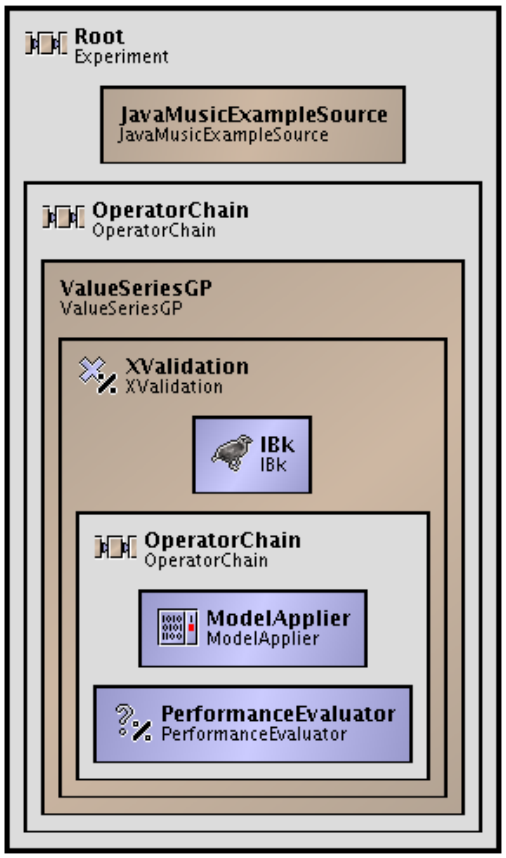


Abbildung 5.2.: Experiment zur Merkmalskonstruktion

Aufteilung	Accuracy
Tempo	0.52
Genre	0.44
Color	0.45

Tabelle 5.1.: Accuracy der einzelnen Datensätze nach durchschnittlich 150 Generationen mit einen k-NN Lerner und 10-facher Kreuzvalidierung

nicht auf ein binäres Problem reduziert wurde. Dieser schwerwiegende Fehler wurde erst am Ende der PG bemerkt. Das Problem ist, dass die Fitness der einzelnen Individuen mit F-measure berechnet wird, welche jedoch nur für binäre Klassifikationsprobleme definiert ist. Aus diesem Grund waren die gefundenen Individuen nicht optimal auf das Problem abgestimmt und haben die Ergebnisse verfälscht. Ein Neustart des Experiments ist in Plannung.

5.2.2. Merkmalsgewichtung und -selektion

Merkmalsgewichtung

Nachdem man einen festen Merkmalssatz gefunden hat, läßt sich dieser weiter optimieren, indem man die Merkmale unterschiedlich gewichtet. Die Idee dabei ist, dass Merkmale, die für das Experiment unnötig oder unwichtig sind, geringer gewichtet werden als diejenigen, die wichtig sind. Die Merkmale werden dabei mit einem Wert zwischen 0 und 1 multipliziert. Gerade bei einfachen Lernverfahren wie k-NN führt dies oft zu einer Erhöhung der Performanz.

Für die Bestimmung der Gewichte gibt es folgenden Verfahren:

- **Forward Weighting** Beim Forward Weighting werden die Merkmale zuerst einzeln evaluiert. Danach werden greedy die besten Merkmale ausgewählt und in diskreten Schritten wird Ihnen immer mehr Gewicht zugeteilt, bis die Performanz nicht weiter steigt.

- **Backward Weighting** Beim Backward Weighting werden in diskreten Schritten den Merkmalen immer weniger Gewicht zugeteilt bis die Performanz nicht weiter steigt.

- **Genetischer Algorithmus** Randomisiertes Verfahren zur Optimierung der Merkmalsgewichte. Die Individuen bestehen aus unterschiedlichen Gewichtsvektoren w$\in \Re^m$ für m Merkmale. Mutationen werden durch Addition mit einer normalverteilten Größe realisiert.

Merkmalsselektion

Merkmalsselektion ist ein Spezialfall der Merkmalsgewichtung. Statt jedes Merkmal mit einem Wert zwischen 0 und 1 zu gewichten, werden Merkmale mit 0 oder 1 gewichtet, also entweder vollständig berücksichtigt oder gar nicht.

Wir verwendeten die Merkmalsselektion zur Erstellung eines vierten Merkmalssatz. Ausgehend vom Merkmalssatz *'Fat'*, führten wir verschiedene Merkmalsselektionen durch: Jeweils Forward Selection, Backward Selection, Selektion per Genetischem Algorithmus für *Genre, Tempo* und *Color*. Daraus errechneten wir diejenigen 44 Merkmale, die am häufigsten ausgewählt worden waren. Diese ergaben den Merkmalssatz *'Selection'*.

5.3. Klassifikation

Eine der großen Herausforderung aktueller Musikmanager besteht darin, den User bei der Verwaltung seines (teilweise sehr großen) Musikbestandes zu unterstützen. In Nemoz werden die Deskriptoren von Songs in Taxonomien gespeichert. Taxonomien sind Baumstrukturen, deren einzelne Knoten Deskriptoren enthalten können, und deren Inhalte mit zunehmender Tiefe des Baumes spezifischer werden. Als Beispiel in diesem Kapitel diene uns die *Genre*-Taxonomie, die in 5.3 gezeigt ist.

Klassifikation ist das Zuordnen von Dingen zu bestimmten Klassen. In Nemoz handelt es sich um Songdeskriptoren in Taxonomieknoten. Mathematisch betrachtet geht es also darum, eine Abbildung $\phi : \mathcal{X} \rightarrow \mathcal{C}$ zu finden, die jedem Songdeskriptor einen Knoten der Taxonomie zuweist, in die er einsortiert werden soll. Da diese Funktion unbekannt ist, muss man sich meist mit einer Approximation $\overline{\Phi}$ zufrieden geben. Diese Approximation nennt sich *Klassifizierungsfunktion*, in Yale handelt es sich um den Operator *ModelApplier* in Kombination mit einem gelernten Modell.

Verschiedene Lernverfahren sind entwickelt worden, um diese Klassifizierungsfunktion zu finden. Im nächsten Abschnitt werden kurz diejenigen vorgestellt, die bei den Experimenten verwendet wurden.

5.3.1. Flache Klassifikation

Die Standardlernverfahren lernen eine Klassifizierungsfunktion $\phi : \mathcal{X} \rightarrow \mathcal{C}$, die jedem Beispiel eine Klasse zuordnet. Es handelt sich um flache Klassifikation, da keine hierarchische Struktur der Klassen vorliegt. Für Nemoz heißt das, dass die hierarchische Struktur der Taxonomien zerschlagen wird, um diese Lerner auf Taxonomien anzuwenden. Ein Beispiel zeigt Abbildung 5.4: jeder Knoten der *Genre*-Taxonomie wird zu einer Klasse, die Struktur ist jedoch nicht mehr zu erkennen.

Was die verschiedenen Lernverfahren angeht, so haben wir uns bemüht, die gesamte Bandbreite an Methoden abzudecken: Support Vector Machines, Neuronale Netze, probabilistische Lernverfahren und Entscheidungsbaumlerner. Mit einer 10-fachen Kreuzvalidierung wurde die Güte der einzelnen Lernverfahren getestet. Als Gütemaß wurde Accuracy gewählt. Für eine genaue Übersicht über die einzelnen Lerner siehe 5.3.1. An dieser Stelle beschränken wir uns auf eine Beschreibung der Operatoren und Parameter.

IBk

Die Weka-Implementierung eines k-nächste Nachbarn Klassifizierers. Für weitere Informationen siehe 2.3.1. Für unsere Experimente beließen wir die Einstellungen so wie sie standardmässig vorgegeben sind, schalteten nur die Normalisierung aus, und variierten die Anzahl der nächsten Nachbarn.

Multilayer Perceptron

Dieser Vetreter der Neuronalen Netze wird im Input Layer mit einem Trainingsbeispiel $X = (x_1, x_2, x_3, ..., x_i)$ gefüttert. In der Trainingsphase werden gewichte der Kanten zwischen den einzelnen Layers angepasst, so dass der Lerner die richtige Klasse vorhersagen kann.

Benutzt wurden die Standardeinstellungen: Vier Knoten im einzigen Hidden Layer, die Sigmoid-Funktionen repräsentieren.

Radial Basis Function Neural Network (RBFNN)

Ein weiterer Vertreter der Neuronalen Netzwerke. Jedoch wird nicht wie beim Multilayer Perceptron Lerner eine Hyperebene oder mehrere Hyperebenen durch den Eingangsvektorraum gelegt sondern im Raum Gaussglocken plaziert, die deren Umgebung klassifizieren.

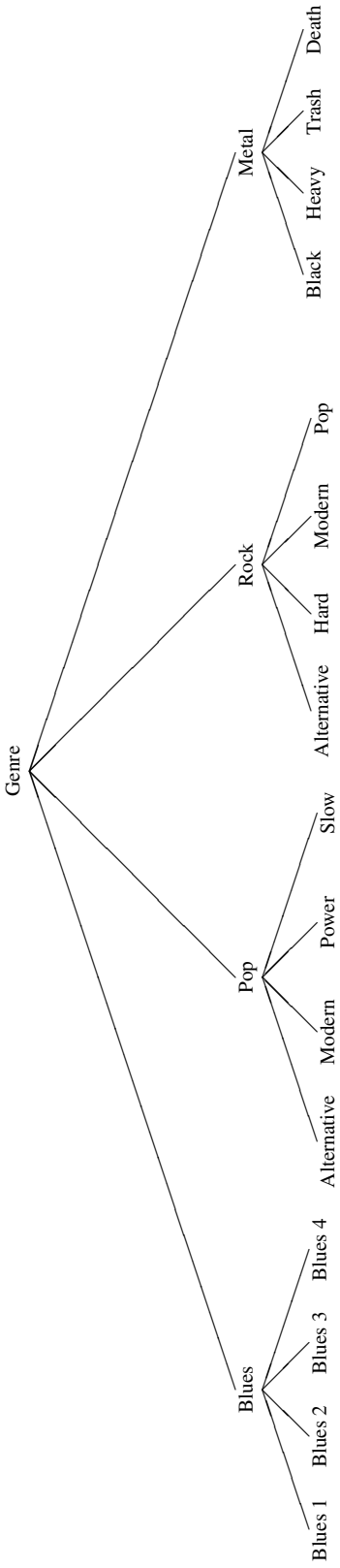


Abbildung 5.3.: Die *Genre*-Taxonomie

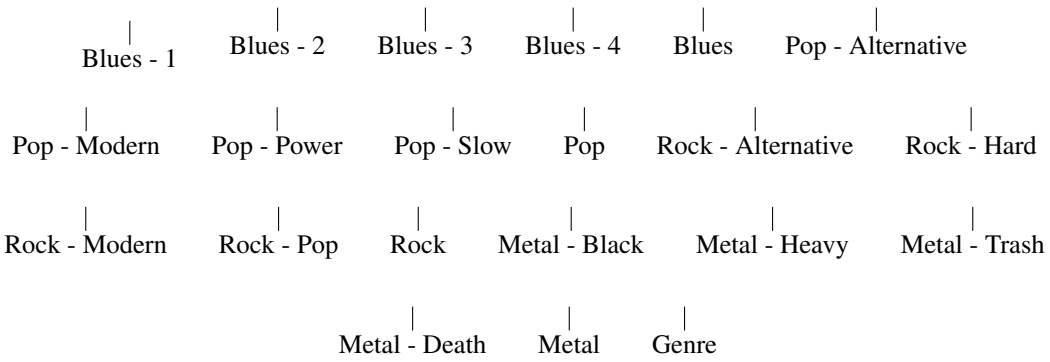


Abbildung 5.4.: Die flache *Genre*-Taxonomie

Da dieser Lerner alle Attribute automatisch normalisiert, liegen für ihn keine Ergebnisse unnormalisierter Experimente vor.

Naive Bayes & Multinomial Naive Bayes

Ein Bayes-Klassifikator ist ein auf der statistischen Entscheidungstheorie basierendes Klassifikationsverfahren. Dabei wird angenommen, dass sich die Zugehörigkeit zu einer Klasse im Merkmalsraum als Dichtefunktion einer Verteilung angeben lässt. Ein Objekt wird der Klasse zugeordnet, zu der es mit der höchsten Wahrscheinlichkeit angehört.[3]

Im Unterschied zu anderen Lernern, ist es bei Naive Bayes Lernern notwendig, die Eingabeattribute zwischen 0 und 1 zu normalisieren. Aus diesem Grund liegen keine Ergebnisse für unnormalisierte Attribute vor.

Support Vector Machine (SVM)

Das Prinzip der Klassifikation bzw. Mustererkennung mit Support-Vector-Maschinen (SVM) beruht auf dem Finden einer optimal trennenden Hyperebene in einem hochdimensionalen Merkmalsraum - typischerweise mit wesentlich höherer Dimension als der ursprüngliche Merkmalsraum (letzterer wird auch als Eingaberaum bezeichnet). Die Idee dahinter besteht darin, dass mit einer passenden nichtlinearen Abbildung in eine ausreichend hohe Dimension, Daten aus zwei Kategorien stets mit Hilfe einer Hyperebene getrennt werden können.[4]

Für unsere Experimente verwendeten wir den bereits in Yale vorhandene LibSVM-Lerner von Chih-Chung Chang and Chih-Jen Lin.

J48 (C4.5)

Bei diesem Lerner handelt es sich um die Weka-Implementierung des C4.5 Entscheidungsbaumlerners ([Qui93]). Die automatische Normalisierung wurde deaktiviert, denn die Normalisierung wurde immer nach den Laden des Examplesets durchgeführt um zu gewährleisten, dass bei jedem Experiment dieselbe Normalisierung durchgeführt wird.

[3] `http://de.wikipedia.org/wiki/Bayes`
[4] `http://de.wikipedia.org/wiki/SVM`

5.3.2. Hierarchische Klassifikation

Flache Klassifikation würde ein neues Beispiel direkt einer dieser Klassen zuordnen. Hierarchische Klassifikation verläuft entlang der Pfade im Baum. Beginnend bei der Wurzel wird in jedem Knoten entschieden, in welchen Teilbaum weitergeleitet wird, oder ob das Beispiel in diesem Knoten bleibt. Im Falle der Genre-Taxonomie würde zunächst (in der Wurzel) entschieden, ob das Beispiel zu Rock, Pop, Metall, Blues oder keinem dieser Genre angehört. Je nachdem, in welchen Kindknoten das Beispiel einsortiert wird, erfolgt eine erneute Entscheidung. Wird das Beispiel in den Knoten Rock einsortiert, folgt danach z.B. Alternative Rock, Hard Rock, Modern Rock, Pop Rock oder bleiben. Dieser Vorgang wird solange wiederholt, bis entweder ein Blatt erreicht ist, oder das Beispiel in einem Knoten bleibt. Als Lerner innerhalb der Knoten werden KNN und J48 verwendet. Für eine gute Übersicht über Hierarchische Klassifikation von Dokumenten siehe [Gra03].

Flache Klassifikation

Der erste Teil des Experiments besteht aus dem Laden der Datensätze. Hier werden z.B. die ExampleSets geladen, die die *Mierswa*-Features verwenden, und die drei verschiedenen Klassifizierungsaufgaben *Genre*, *Color* und *Tempo* repräsentieren. Danach folgt klassische Kreuzvalidierung mit den verschiedenen Lernern. Siehe 5.5 links.

Hierarchische Klassifikation

Hierarchische Klassifikation verwendet als Taxonomiestrukturen Clustermodels aus dem Clustering-Plugin von Yale. Dementsprechend wird nicht nur ein ExampleSet geladen, sondern auch das jeweilige Clustermodel, das die Taxonomie repräsentiert. Nach der gleichen Vorverarbeitung wie bei flacher Klassifikation werden die im Clustermodel enthaltenen Informationen mit in das ExampleSet geschrieben (CM2PathTaggedES), woraufhin wieder Kreuzvalidierung folgt. Diesmal mit dem Operator Hierarchische Klassifikation, der als inneren Operator den in den Knoten verwendeten Lerner enthält. Zur Bestimmung der *TreeDistance* dient der ClassLabelEvaluator. Siehe 5.5 rechts.

5.3.3. Ergebnisse

Merkmalssätze

Die Ergebnisse zu verschiedenen Merkmalssätzen lassen sich nur schwer interpretieren. Die bisherigen Ergebnisse deuten daraufhin, dass es egal ist, welchen Merkmalssatz man verwendet. Es gibt scheinbar keinen, der wirklich zuverlässig mit allen Problemen zurechtkommt. Betrachtet man Tabelle A.1, würde man sagen: *Fat* für die Taxonomie *Color*, *Mierswa* für die beiden anderen. Sobald man das ganze mit Normierung betrachtet, scheint *Mörchen* am besten geeignet für *Tempo*, *Selection* für *Color* und für *Genre* ergibt sich kein einheitliches Bild.

Wir werden später noch einmal auf das Problem zurückkommen, wenn es um die Wahl des besten Lernverfahrens geht.

Normierung, Gewichtung

Für den Vergleich zwischen unnormalisierten Daten, normalisierten Daten und gewichteten Daten betrachten wir exemplarisch die folgenden Werte des IBk Lerners, bei dem nur ein Nachbar berücksichtigt wurde und der Gewichtsvektor mit Forward Weighting berechnet wurde.

Die Steigerung des Lernergebnisses ist deutlich zu erkennen. Es empfiehlt sich also, nur mit normierten Daten zu arbeiten. Bei der Gewichtung ist der Gewichtsvektor speziell auf die jeweilige Taxonomie trai-

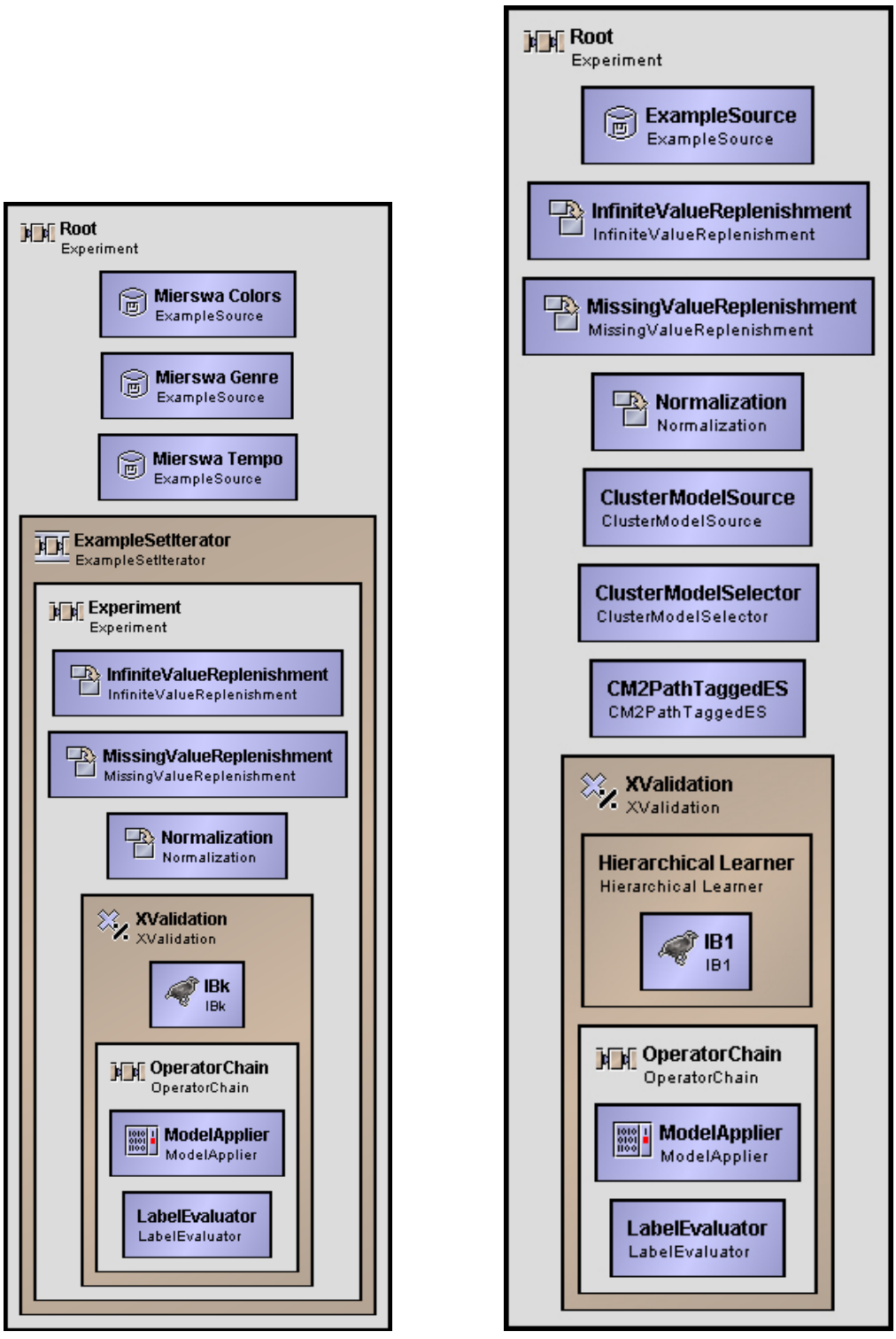


Abbildung 5.5.: Die zur Klassifikation benutzten Experimente

		unnormiert	normiert	gewichtet
Fat	Tempo	45,85	58,93	63,10
	Genre	36,67	38,67	40,67
	Color	43,92	48,65	49,48

Tabelle 5.2.: Accuracy normiertes und gewichtetes IBk

niert worden. Folgende Tabelle zeigt, dass genetische Algorithmen zu unvorhergesehenen Ergebnissen führen können.

		Gewichtsvektoren		
		Tempo	Genre	Color
Fat	Tempo	59,52	63,69	55,36
	Genre	36,67	44,67	46,00
	Color	49,32	50,00	58,11

Tabelle 5.3.: Gewichtungen

Zu erwarten wäre wohl gewesen, dass die beste Performanz da auftritt, wo Gewichtsvektor und Problem zusammen gehören. Die Tabelle zeigt jedoch, dass das in zwei von drei Fällen nicht der Fall nicht gewesen ist. Wir führen das darauf zurück, dass die Gewichtung mittels genetischer Algorithmen erzeugt worden sind, deren Parameter nicht optimal eingestellt waren. Spätere Tests haben ergeben, dass sich die Performanz z.B. mittels *Forward Weighting* weiter erhöhen läßt.

Verschiedene Lernverfahren

Auch für verschiedenen Verfahren ist es schwer, ein klares Ergebnis zu erkennen. Verschiedene Lerner funktionieren mit verschiedenen Parametern unterschiedlich gut auf unterschiedlichen Problem. Den besten Eindruck macht eine SVM mit C=100 mit dem Merkmalssatz *'Fat'*. Diese erzeugt auf allen drei Taxonomien die besten Ergebnisse, wenn es auch Lerner gibt, die auf einzelnen Taxomomien besser sind.

Fazit Hierarchische & Flache Klassifikation

Den Experimenten zur Klassifikation nach zu urteilen, die wir bisher durchgeführt haben, führt hierarchische Klassifikation nicht unbedingt zu einer Verbesserung der Klassifizierungsgüte. Tabelle 5.4 zeigt ausgewählte Beispiele. Diese Beispiele sind von PG Teilnehmern erzeugte Taxonomien. Es stellt sich allerdings die Frage, ob man das Ergebnis nicht signifikant verbessern kann, indem man z.B. Merkmalsgewichtung in jedem Knoten ausführt.

Das Zusammenspiel zwischen Merkmalsatz, Gewichtung und Lernverfahren bedarf wohl noch der genaueren Untersuchung. Selbst die besten Ergebnisse von ungefähr 70% Accuracy sind für den Benutzer nicht akzeptabel.

5.4. Clustering

Beim Clustering werden die Instanzen von einer Menge X in einer Menge von Clustern C partitioniert, sodass die Extension für jedes c aus C eine Teilmenge von X ist und die Instanzen in einem Cluster eine höhere Ähnlichkeit aufweisen als die Instanzen aus verschiedenen. In diesem Abschnitt werden die benutzten Standardmethoden kurz Erläutert. Für eine detalierte Erläuterung verweisen wir auf [JMF99].

		J48, flach		J48, hier.	
Fat	Instruments	0,33	1,63	0,56	1,12
	Genres	0,40	1,73	0,28	1,56
	MP3s	0,20	2,04	0,16	1,99
	BigVoice	0,26	1,40	0,14	2,67
	no name (1)	0,67	0,91	0,68	0,99

Tabelle 5.4.: Accuracy und TreeDistance, hierarchische & flache Klassifikation

5.4.1. Standardmethoden

Es existiert eine große Anzahl an Clustering Algorithmen. Die Wahl eines Clustering Algorithmus hängt von Eigenschaften der betrachteten Daten und vom Ziel der Anwendung ab. Clustering Methoden lassen sich wie folgt einteilen:

- **Partitionierende Methoden:**
  Bei einer gegebenen Menge von n Elementen konstruiert ein partitionierender Algorithmus k Partitionen, wobei jede Partition ein Cluster repräsentiert und $k <= n$ gilt. Das heißt, es klassifiziert die Elemente in k Gruppen mit den folgenden Eigenschaften:

  1. Jede Gruppe beinhaldet mindestens ein Element.
  2. Jedes Element gehört exakt zu einer Gruppe.

  Um das globale Optimum bei partitionierenden Methoden zu finden, müsste man alle mögliche Partitionen berechnen. Die meisten Applikationen jedoch benutzten die Heuristiken **K-Means** oder **K-Medoids**. Sei k die Anzahl der Partitionen, die erstellt werden sollen. Diese Heuristiken kreieren zuerst eine Initial Partition. In der Iterationsphase werden Elemente von einer Partition einer anderen, zu der das Element am nähesten ist, zugeordnet, bis ein Abbruchkriterium, z.B. es erfolgt keine Gruppen Änderung mehr, erfüllt ist. Der Unterschied zwischen den beiden Heuristiken ist wie folgt:

  – **K-Means** Jedes Cluster wird durch den Durchschnittwert der Elemente im Cluster representiert.
  – **K-Medoids** Jedes Cluster wird durch das Element representiert, das sich dem Clusterzentrum am nähesten befindet.

  Diese Heuristiken erzeugen Cluster, die eine kugelförmige Form haben, was nicht immer sinnvoll ist.

- **Hierarchische Methoden:** Die hierarchischen Clusteringmethoden erzeugen eine hierarchische Zerlegung der gegebenen Elementenmenge. Eine hierarchische Methode kann als agglomerative oder teilend klassifiziert werden. Die agglomerative Methode benutzt einen *bottom-up* Ansatz, in der jedes Element zu Beginn ein eigenes Cluster bildet. Sukzessive werden ähnliche Cluster miteinander verschmolzen, bis nur noch eine Cluster übrig bleibt oder ein Abbruchkriterium erfüllt ist.

  Sei $|p - p'|$ der Abstand zwischen zwei Elementen p und p', m_i der Zentroid für Cluster C_i und n_i die Anzahl der Elemente in C_i. Die meist verbreitesten Maße für die Abstandsberechnung zwischen zwei Clustern sind wie folgt definiert:

1. **Single Link (Minimaler Abstand)**

$$d_{min}(C_i, C_j) = min_{p \in C_i, p' \in C_j} |p - p'|$$

2. **Complete Link (Maximaler Abstand)**

$$d_{max}(C_i, C_j) = max_{p \in C_i, p' \in C_j} |p - p'|$$

3. **Zentroid Abstand**

$$d_{mean}(C_i, C_j) = |m_i - m_j|$$

Die größte Schwäche der hierarchischen Methoden ist, dass eine Verschmelzung nicht mehr Rückgängig gemacht werden kann. Eine teilende Methode benutzt einen *top-down*-Ansatz, in der alle Elemente zuerst im selben Cluster liegen. Sukzessive werden die Cluster in kleinere Cluster gesplittet wofür partitionierende Methoden, wie K-Means benutzt werden.

		Buttom Up CL	Buttom Up SL	Top Down
Fat	Unnormiert	20,289	11,249	1,572
	Normiert	30	11,019	1,572
Mörchen	Unnormiert	14,938	10,587	1,572
	Normiert	27,784	10,873	1,795
Mierswa	Unnormiert	13,960	10,764	1,734
	Normiert	27,23	10,995	1,910
Selection	Unnormiert	21,713	11,407	1,875
	Normiert	25,222	10,890	1,572

Tabelle 5.5.: Durchschnittlicher Absoluter Abstand der Cluster Modelle mit den Erzeugten Cluster

5.4.2. Umsetzung in Yale

Die Merkmalssätze, die für die Experimente von Clustering benutzt wurden, sind identisch zu den Merkmalsätzen bei der Klassifikation. Desweiteren wurde von den PG Teilnehmer hierarchische Taxonomien erstellt, die auch als Eingabe für die Bewertung der Clustering Algorithmen benutzt wurden. Diese hierarchische Taxonomien sind baumartige Strukturen und werden von nun an als Cluster Modell bezeichnet. Ein Cluster Modell hat die Eigenschaft, dass der Wurzelknoten alle Instanzen des Baum beinhaltet. Eine Instanz wird jedoch in dem tiefsten Teilbaum gekennzeichnet in dem sich die Instanz befindet.

Der Ablauf der Clustering Experimente ist wie folgt. Eine Anfragemenge $X_q \subseteq X$ die geclustert werden soll, beinhaltet alle Instanzen aus einem Cluster Modell.

Danach wird versucht die selbe hierarchische Struktur des Cluster Modell zu erzeugen. Auf diese Weise sind wir in der Lage ein benutzerdefiniertes Cluster Modell mit einem von Clustering erzeugten Cluster Modell zu vergleichen. Zwei Clustermodelle werden in 2 Schritten verglichen. Zuerst wird die Tree Distanz zwischen den einzelnen Instanzen berechnet. Um die Tree Distanz zwischen zwei Instanzen i und j zu berechnen, müssen die Kanten gezählt werden die von Knoten X (beinhaltet Instanz i) zu Knoten Y (beinhaltet Instanz y) führen. Diese Tree Distanz Werte werden in einer Matrix zusammengefasst. Auf dieselbe Art wird noch eine Matrix für das erzeugte Cluster Modell erstellt. Dann wird die durchschnittliche Abweichung zwischen beiden Matrizen berechnet. Der Durchschnitt aus diesen Abstandswerten ist der Absolute Abstand zwischen zwei Cluster Modellen.

$$L_{abs}(cm_k, cm_l) = \frac{1}{|X_q|^2} \sum_{i,j \in X_q} |dis_k(i,j) - dis_l(i,j)|$$

Ein weiteres Ähnlichkeitsmaß um zwei Cluster Modelle zu vergleichen ist die Korrelation der Tree Distanz. Dieser Abstandswert wurde in den Experimenten nicht berechnet.

5.4.3. Fazit

Für das Clustern von Audiodaten ist eine Normalisierung als Vorverarbeitung nicht zu wählen. Eine Hierarische Clustering Aufgabe mit den Top-Down Ansatz liefert wie in Tabelle 5.4.1 zu sehen die besten Werte. Eine weiterer Grund, dieses Verfahren in Nemoz anzuwenden, ist, dass eine hierarschische Clusteringaufgabe mit dem Top-Down Ansatz in Echtzeit arbeitet. Die erzeugten Bäume mit dem Buttom-Up Ansatz sind sehr tief. Aus diesem Grund sind die Abstandswerte in der Tabelle wesentlich schlechter als die des Top-Down Ansatzes.

6. Ausblick

Nemoz ist ein großes System und ein vielschichtiges System. Es eröffnet eine Vielzahl von Möglichkeiten und läßt sich in viele Richtungen weiterentwickeln. Dies ist dank der recht sauberen Architektur konzeptuell und dank des Plugin-Systems technisch auch einfach möglich. Es ist also einzig den begrenzten zeitlichen Resourcen der Projektgruppe geschuldet, wenn viele interessante Erweiterungen noch nicht implementiert werden konnten. Dies soll jedoch im Rahmen des Open Source Projekts Nemoz und hoffentlich auch im Rahmen zahlreicher Diplomarbeiten nachgeholt werden.

Dieses Kapitel gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen, der natürlich keinen Anspruch auf Vollständigkeit erheben will oder kann. Der erste Abschnitt beschäftigt sich mit rein technischen Aspekten, welche in erster Linie die Wartberkeit und Performanz des Systems betreffen. Im zweiten Abschnitt folgt eine Aufzählung möglicher konzeptueller Erweiterungen und neuer Features. Der letzte Abschnitt ist bewusst weitaus weniger konkret und realistisch geraten als die vorhergehenden. Er handelt von der durch JXTA und IRC inspirierten Idee, Nemoz zur „Nemoz World“, einer „freien, virtuellen Community für Musikfans“ auszubauen.

6.1. Technisches

Portierung auf Java 1.5 Derzeit ist Nemoz in Java 1.4.2 implementiert, welches zu Projektbeginn die aktuelle, stabile Version darstellte und somit eine vernünftige Wahl war. Inzwischen hat jedoch Java 1.5 (oder Java 5, wie es von der Sun-Marketingabteilung seit kurzem genannt wird) eine Stabilität erreicht, die eine Portierung von Nemoz lohnenswert erscheinen läßt. Java 1.5 bietet zuerst einmal im Multimediabereich viele Vorteile gegenüber der Vorgängerversion, da es zum einen eine flexiblere und leistungsfähigere JavaSound-API mitbringt und zum anderen ein besseres Thread-Scheduling bietet. Ohne Codeänderungen läuft Nemoz so auch auf langsameren Rechnern flüssig und Feature-Extraktion bei gleichzeitigem Abspielen einer MP3-Datei rückt in den Bereich des Möglichen.

Ein weiterer für Nemoz relevanter Vorteil von Java 1.5 ist die rudimentäre Unterstützung für generische Programmierung durch Generics und Variance [1]. Während dieses mit Hilfe von Erasure im Compilerfrontend implementierte Framework in vielerlei Hinsicht zu wünschen übrig läßt, ermöglicht es doch weitaus besser lesbare APIs in und um Nemoz, da von einer Schnittstellenmethode erwartete oder zurückgegebene Collection-Typen nun explizit deklariert werden können.

Optimierung der YALE-Anbindung YALE hat sich als exzellentes Werkzeug zur explorativen Entwicklung und Anpassung von Clustering- und Klassifikationsmethoden erwiesen. Die derzeitige, vollständige Einbettung in Nemoz ist allerdings nicht optimal. Zum einen ist das YALE-System prinzipbedingt recht groß, zum anderen werden von Nemoz nur wenige der angebotenen Funktionen wirklich genutzt. Der sicherlich größte Nachteil der bestehenden Lösung ist aber die fehlende Threadsicherheit eines „Standard-YALE“, wodurch zeitaufwändige Verfahren in Nemoz nur sehr bedingt im Hintergrund ausgeführt werden können.

Ein möglicher Ausweg wäre eine Implementierung der genutzten Verfahren im Nemoz-Kern, wobei diese dabei threadsicher gemacht und auf ihr spezielles Einsatzgebiet optimiert werden sollten. Die zwischen dem Domänenmodell von YALE und dem Modell von Nemoz vermittelnde Schicht könnte

[1] `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`

dann wegfallen, was Performanz und Einfachheit des Systems weiter verbessern würde. Optimierte Verfahren können im Gegenzug wieder in YALE einfließen.

Portierung des Netzwerkservices auf JXTA JXTA hat während der Entwicklungszeit von Nemoz grundlegende Verbesserungen an Dokumentation und Implementierung erfahren, so daß es nun für den Einsatz in Nemoz geeignet scheint. Eine direkte Umsetzung der Funktionalität das Nemoz-Netzwerkservice in JXTA ist relativ einfach möglich und bietet einige direkte Vorteile. Hierzu zählen die leichtere Wartbarkeit des Codes (grundlegende Netzwerkfunktionalität muss nicht mehr selbst implementiert und gewartet werden) und bessere Unterstützung von Peers hinter Firewalls. Mittelfristig läßt sich JXTA dann zum Aufbau eines Internetweiten Nemoz-Netzes nutzen, dazu jedoch mehr in Abschnitt 6.3.

6.2. Konzeptuelles

6.2.1. Klassifikation

Selbst wenn die bisherigen Ergebnisse noch nicht wirklich zufriedenstellend sind, so gibt es doch viele mögliche Erweiterungen und Modifikationen, die die Qualität noch erhöhen könnten.

Unterschiedliche Verfahren pro Knoten

Zwar läuft die Klassifikation hierarchisch ab, in jedem Knoten wird aber stets der gleiche Lerner mit den gleichen Merkmalen verwendet. Wie im Abschnitt 5.2.1 bereits erwähnt, ist aber nicht jeder Merkmalssatz für jedes Klassifizierungsproblem geeignet. Für jeden Knoten (oder zumindest für jede Taxonomie) einen Merkmalssatz zu konstruieren scheitert aber - bisher - an der Laufzeit der Merkmalskonstruktion. Merkmalsgewichtung dauert weniger lange, und mag - für jeden Knoten ausgeführt - das Ergebnis verbessern.

Eine weitere Möglichkeit wäre, den Lerner oder seine Paramter für jeden Knoten anzupassen. Yale stellt bereits einen Operator zur Parameteroptimierung bereit, ein solches Experiment wäre also leicht durchzuführen.

Meta-Learning

Meta-Learning beschäftigt sich mit dem Problem, dass Merkmalskonstruktion zu zeitaufwendig ist, um sie immer wieder neu auszuführen. Statt immer wieder alle Merkmale zu extrahieren, benutzt man zunächst einmal nur einen kleinen Satz an Merkmalen. Aus einer Gewichtung dieser Merkmale lernt man einen passenden größeren Datensatz für das jeweilige Experiment, und extrahiert dann die nötigen fehlenden Merkmale. Somit muß man keine neue Merkmalskonstruktion ausführen, sondern nur ein paar weitere Merkmale extrahieren.

Erweiterte Merkmalssätze

Ebenfalls naheliegend wäre, die Merkmalssätze um zusätzliche Attribute zu erweitern, die nicht aus der Audiospur extrahiert wurden, z.B. Metainformationen wie sie im ID3-tag enthalten sind.

6.2.2. Clustering

Non-Redundant Clustering & Multi-View Clustering

Für eine kurze Übersicht siehe Abschnitt 2.3.7

Verteiltes Kollaboratives Clustering

Für die Klassifikation von Audiodaten müssen vordefinierte Beispiele existieren. Das Vordefinieren von Beispielen ist eine sehr arbeitsintensive Aufgabe. Die Alternative, Clustering, versagt bei der Aufteilung der Musikstücke wegen der großen Anzahl an Merkmalen mit den bekannten Standartverfahren.

Beim kollaborativen Clustering werden Taxonomien von anderen Peers benutzt, um eine Menge von Objekten zu gruppieren. Dabei broadcastet ein Knoten zunächst alle zu gruppierenden Objekte an seine peers und erhält daraufhin passende (Teil-)taxonomien. In diese sortiert er die fehlenden Objekte durch hierarchische Klassifikation ein und präsentiert dem Nutzer verschiedene, alternative Ergebnisse. [WMM05]

6.3. Eine kurze Reise um die Nemoz World

Die Nemoz World, eine „freie, virtuelle Community für Musikfans“ ist ähnlich dem Internet-Relay-Chat (IRC) Netzwerk aufgebaut, kommt aber im Gegensatz zu diesem ohne zentrale Server aus (siehe Abbildung 6.1 Benutzer können sogenannten *Zones* beitreten oder diese selbst neu erstellen, indem sie einer nicht einer noch nicht existierenden Zone beitreten. Zones sind Gruppen von Nemoz-Peers, wobei jeder Peer Mitglied beliebig vieler Zones sein darf. Die in Aufteilung in Zones wird von den Benutzern also selbst organisiert. Es ist zu erwarten, dass Zones für Fans verschiedener Musikgenres („Rock“) entstehen oder Zones geographisch benachbarter Peers („Studentendorf“).

Nicht zuletzt würde ein internetweites Netz von Nemoz-Peers eine ideale Versuchsumgebung für neue kollaborative Clustering- und Klassifikationsmethoden darstellen.

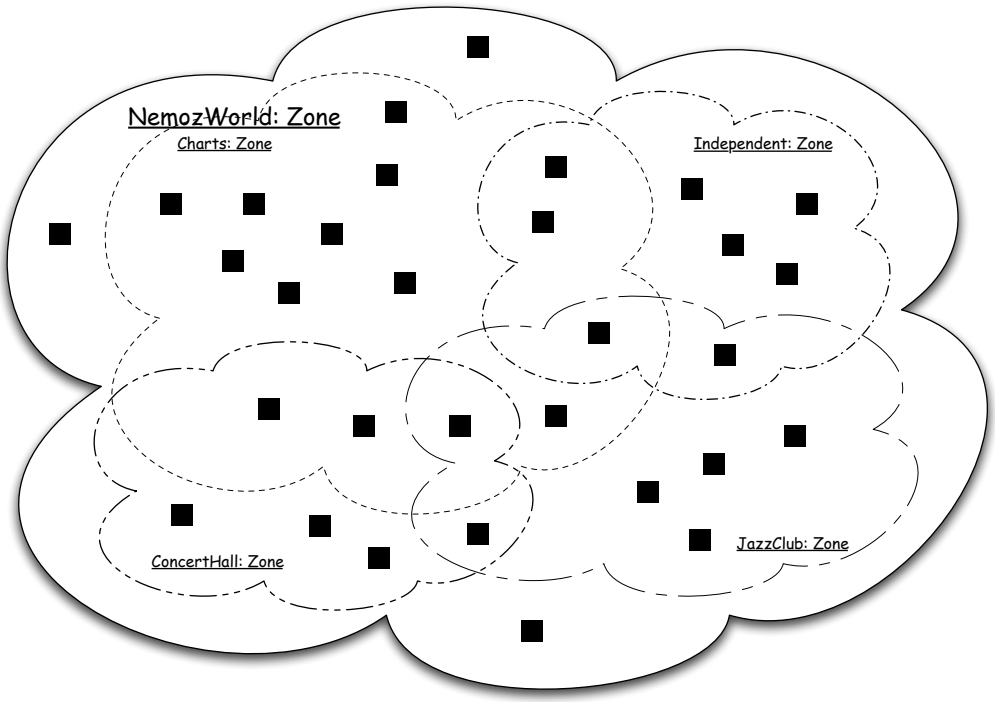


Abbildung 6.1.: Die Nemoz World

A. Experimentergebnisse

Bei der Klassifikation und Clustering wurden die Features von **Mörchen** nach [MUT+05], **Mierswa** nach [Mie04] und der *Fat* Datensatz, siehe Abschnitt 5.2.1, benutzt.

Bei der Merkmalsselektion wurden die Merkmale aus den Fat Features mit den Klassifikationsaufgaben **Genre**, **Color** und**Tempo** selektiert mit Forward Selection, Backward Selection und Evolutionärer Selektion. Als innerer Lerner wurde Nearest Neighbout und SVM angewendet.

		IBk(1)	IBk(3)	IBk(5)	IBk(7)	IBk(9)	IBk(11)	IBk(13)	IBk(15)
Fat	Tempo	45,85	50,6	48,21	44,64	44,64	46,43	48,81	48,81
	Genre	36,67	37,33	37,33	32,67	34,00	40,00	42,67	43,33
	Color	43,92	44,59	50,00	53,38	53,38	54,05	53,38	50,68
Mörchen	Tempo	52,33	45,93	47,09	50,58	51,74	52,91	50,58	52,91
	Genre	34,42	35,06	29,87	29,22	33,77	31,82	35,06	36,36
	Color	44,52	43,84	42,47	45,21	41,10	44,52	42,47	42,47
Mierswa	Tempo	50,00	49,03	53,88	55,34	53,40	52,92	51,94	52,43
	Genre	30,40	40,80	48,00	43,20	44,00	42,40	43,20	43,20
	Color	35,14	43,92	44,59	48,65	50,68	47,30	47,30	43,92
Selection	Tempo	26,21	26,19	26,21	26,21	26,21	26,21	26,21	26,21
	Genre	12	12	12,67	12,67	12,67	12,67	12,67	12,67
	Color	33,81	33,78	33,81	33,81	33,81	33,81	33,81	33,81

Tabelle A.1.: Accuracy unnormiertes IBk

		IBk(1)	IBk(3)	IBk(5)	IBk(7)	IBk(9)	IBk(11)	IBk(13)	IBk(15)
Fat	Tempo	58,93	61,31	65,48	63,69	66,07	64,29	61,90	63,10
	Genre	38,67	39,33	48,67	52,00	50,67	53,33	56,67	52,67
	Color	48,65	54,05	59,46	63,51	60,81	60,14	60,81	54,73
Mörchen	Tempo	66,86	63,95	70,35	70,35	71,51	69,19	68,60	68,02
	Genre	35,70	37,32	45,45	46,10	44,16	44,81	41,56	48,05
	Color	48,63	47,95	50,00	55,48	58,22	57,53	54,79	54,79
Mierswa	Tempo	64,56	67,48	66,99	69,90	68,45	66,50	67,48	67,48
	Genre	44,00	49,60	49,60	48,80	49,60	45,60	45,60	48,80
	Color	43,24	43,92	45,95	50,00	49,32	52,70	50,68	51,35
Selection	Tempo	61,67	60,11	65,48	66,65	64,89	65,48	61,29	61,31
	Genre	40,00	41,33	50,00	53,33	52,67	52,67	53,33	53,33
	Color	48,71	55,48	61,57	63,53	60,24	60,14	58,12	58,11

Tabelle A.2.: Accuracy normiertes IBk

		IBk(1)	IBk(3)	IBk(5)	IBk(7)	IBk(9)	IBk(11)	IBk(13)	IBk(15)
Fat	Tempo	63,10	66,07	67,28	68,45	67,26	67,26	69,64	69,05
	Genre	40,67	42,67	50,67	52,67	51,33	52,67	53,33	54,67
	Color	49,48	56,76	63,57	64,19	64,86	64,86	66,22	65,54
Mörchen	Tempo	71,51	70,98	74,42	73,26	73,26	73,84	73,84	73,84
	Genre	41,56	47,40	47,43	48,05	48,70	50,65	52,60	52,60
	Color	48,69	57,73	56,16	58,22	59,67	61,64	60,96	59,59
Mierswa	Tempo	68,45	70,87	74,27	72,82	72,82	70,09	70,87	70,87
	Genre	39,77	38,07	43,75	46,59	46,59	47,16	49,43	47,16
	Color	46,62	50,00	55,41	57,43	56,76	56,76	58,11	58,11
Selection	Tempo	—	—	—	—	—	—	—	—
	Genre	—	—	—	—	—	—	—	—
	Color	—	—	—	—	—	—	—	—

Tabelle A.3.: Accuracy normiertes und gewichtetes IBk

		J48	RBFNN
Fat	Tempo	69,05	63,69
	Genre	46,00	48,00
	Color	45,95	61,49
Mörchen	Tempo	64,53	68,01
	Genre	40,53	42,21
	Color	**52,05**	59,11
Mierswa	Tempo	62,62	66,29
	Genre	40,34	46,59
	Color	43,97	46,62
Selection	Tempo	66,69	66,67
	Genre	47,33	53,33
	Color	50,81	58,78

Tabelle A.4.: Accuracy unnormierte Lernverfahren

		J48	RBFNN	MLPerceptron	Bayes	Bayes Multi
Fat	Tempo	64,88	63,69	58,78	55,36	60,71
	Genre	48,00	40,00	46,67	47,36	48,67
	Color	48,65	50,68	65,48	62,16	**62,16**
Mörchen	Tempo	65,12	68,02	66,86	67,44	59,88
	Genre	40,26	37,66	41,56	45,45	46,10
	Color	52,05	58,22	58,90	47,95	47,95
Mierswa	Tempo	60,19	65,05	70,39	54,37	59,22
	Genre	52,00	54,40	46,02	44,80	48,80
	Color	47,97	52,70	59,46	58,78	55,41
Selection	Tempo	67,26	64,93	64,88	26,29	26,69
	Genre	49,33	40,00	48,67	12,67	12,67
	Color	54,70	58,19	58,11	33,78	33,78

Tabelle A.5.: Accuracy normierte Lernverfahren

		C=10	C=50	C=100	C=150
Fat	Tempo	63,10	68,45	70,83	70,24
	Genre	54,00	54,67	48,67	48,67
	Color	60,14	58,78	62,16	60,81
Mörchen	Tempo	71,51	66,28	69,19	69,77
	Genre	44,16	39,61	37,66	38,96
	Color	53,42	58,22	55,48	56,85
Mierswa	Tempo	62,62	60,19	67,48	56,08
	Genre	50,00	44,89	46,02	46,59
	Color	56,00	58,11	58,11	69,42
Selection	Tempo	42,68	42,86	42,86	42,86
	Genre	28,00	28,00	28,00	28,00
	Color	13,51	13,51	13,51	13,51

Tabelle A.6.: Accuracy normierte SVM

		IBk(3)		IBk(5)		IBk(7)		J48	
		Acc	TD	Acc	TD	Acc	TD	Acc	TD
Fat	Instruments	0,51	1,20	0,53	1,16	0,60	0,99	0,33	1,63
	Moods	0,12	1,51	0,18	1,49	0,17	1,65	0,09	1,98
	Genres	0,28	1,57	0,37	1,30	0,60	1,28	0,40	1,73
	Nice Things	0,68	0,81	0,73	0,69	0,74	0,64	0,66	0,87
	My Music	0,10	2,98	0,14	2,85	0,17	2,76	0,13	2,88
	mp3s	0,17	2,18	0,20	2,11	0,25	1,95	0,20	2,04
	music	0,20	2,47	0,32	2,10	0,29	2,16	0,26	2,27
	moods	0,33	1,35	0,24	2,53	0,23	2,62	0,18	2,89
	Party	0,71	0,96	0,73	0,92	0,74	0,90	0,60	1,30
	Big Voice	0,32	1,52	0,34	1,44	0,33	1,47	0,26	1,40
	no title	0,13	2,94	0,13	2,98	0,18	2,70	0,12	2,85
	no name (1)	0,75	0,70	0,78	0,63	0,79	0,61	0,67	0,91
	no name (2)	0,28	1,79	0,18	1,97	0,18	2,00	0,21	1,97

Tabelle A.7.: Accuracy und durchschnittliche TreeDistance flache Klassfikation, Garagebanddaten

		IBk(3)		IBk(5)		IBk(7)		J48	
		Acc	TD	Acc	TD	Acc	TD	Acc	TD
Fat	Instruments	0,51	1,20	0,60	0,98	0,57	1,04	0,56	1,12
	Moods	0,13	1,50	0,15	1,72	0,09	2,03	0,09	1,83
	Genres	0,30	1,47	0,45	1,18	0,45	1,20	0,28	1,56
	Nice Things	0,71	0,71	0,74	0,64	0,75	0,65	0,70	0,70
	My Music	0,14	2,74	0,15	2,71	0,21	2,53	0,11	2,93
	mp3s	0,16	2,16	0,17	2,18	0,24	1,97	0,16	1,99
	music	0,31	2,09	0,20	2,40	0,24	2,28	0,19	2,47
	moods	0,33	1,35	0,21	2,56	0,18	2,54	0,17	2,96
	Party	0,72	0,95	0,71	0,96	0,73	0,70	0,63	1,31
	Big Voice	0,33	1,44	0,27	1,55	0,28	1,50	0,22	1,57
	no title	0,12	2,84	0,16	2,84	0,20	2,63	0,14	2,67
	no name (1)	0,78	0,65	0,78	0,60	0,78	0,60	0,68	0,99
	no name (2)	0,24	1,87	0,23	1,92	0,25	1,87	0,18	2,08

Tabelle A.8.: Accuracy und durchschnittliche TreeDistance hierarchische Klassfikation, Garagebanddaten

		Buttom Up CL	Buttom Up SL	Top Down
Fat	Unnormiert	20,289	11,249	1,572
	Normiert	30	11,019	1,572
Mörchen	Unnormiert	14,938	10,587	1,572
	Normiert	27,784	10,873	1,795
Mierswa	Unnormiert	13,960	10,764	1,734
	Normiert	27,23	10,995	1,910
Selection	Unnormiert	21,713	11,407	1,875
	Normiert	25,222	10,890	1,572

Tabelle A.9.: Durchschnittlicher Absoluter Abstand der Cluster Modelle mit den Erzeugten Cluster

		Buttom Up CL	Buttom Up SL	Top Down
Fat	Unnormiert	19,155	10,155	1,870
	Normiert	31,227	11,019	2,004
Mörchen	Unnormiert	12,821	9,599	1,788
	Normiert	30,534	10,227	1,982
Mierswa	Unnormiert	12,127	9,760	1,810
	Normiert	30,060	10,816	1,950
Selection	Unnormiert	20,570	10,277	1,832
	Normiert	28,066	10,889	1,911

Tabelle A.10.: Durchschnittlicher Absoluter Abstand der Cluster Modelle mit den erweiterten Merkmalsvektoren

Merkmale	Häufigkeit
peak 0 value(FrequencyPeaks (2))	12
log(std(rob5(abs(diff 1(SpectralCentroid)))))	10
log(kurt(diff 2(BandEnergyRatio)))	11
pc2(PhaseSpace 2 7(MFCC1))	10
mean(rob5(MFCC3))	11
auto correlation(MFCC33) 5.0	10
log(pc2(PhaseSpace 2 2(MelBin2)))	10
log(centroid(mag(FFT(Hanning weight(MelBin29)))))	10
average quadratic mean(RMS-Loudness (2))	10
peak 2 index(FrequencyPeaks (2))	10
peak 3 value(FrequencyPeaks (2))	11
peak 3 index(FrequencyPeaks (2))	11
interval cluster0values(lowpass(extrema(series)))type0(lowPass(extrema(series)))	11
interval cluster1empty(lowpass(extrema(series)))end1(lowPass(extrema(series)))	10
lin reg discrepancy(bark scale(FFT(Hanning weight(series))))	10
average arithmetic mean(FrequencyWindowAverage (2))	11
length(LengthGenerator (2))	11
absolute mean(AbsoluteAverage (2))	11
log(amplitude index(AmplitudeIndex (2)))	10
lin reg gradient(series)	10
length(LengthGenerator (3))	11
peak 0 value(PeakFinder (3) (2))	10
peak 1 width(PeakFinder (3) (2))	10
min(Min (3) (2))	11
log(length(LengthGenerator (2) (2) (2)))	10
average arithmetic mean(Average (2) (4))	10
log(std(rob5(log(NormChromaF))))	9
sqrt(centroid(mag(FFT(Hanning weight(LowSpectralEnergy)))))	9
log(bandwidth(mag(FFT(Hanning weight(MelBin27)))))	9
deviation arithmetic mean(ZeroCrossingsAverage (2))	9
deviation arithmetic mean(DistanceAverage (2))	9
peak 0 index(FrequencyPeaks (2))	9
/(max(SpecMax (2)),average arithmetic mean(SpecArithAvg (2)))	9
interval cluster 1 empty(series) type 1(series)	9
interval cluster 1 empty(series) end 1(series)	9
interval cluster 2 values(series) type 2(series)	9
interval cluster 2 values(series) start 2(series)	9
interval cluster 2 values(series) end 2(series)	9
average arithmetic mean(Average)	9
absolute mean(AbsoluteAverage (3))	9
amplitude index(AmplitudeIndex (2) (2))	9
peak 0 index(PeakFinder (3) (2))	9
peak 1 value(PeakFinder (3) (2))	9
peak 1 index(PeakFinder (3) (2))	9

Tabelle A.11.: Die Gewählten Merkmale nach 18 Feature Selektion Experimenten

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AIS77] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language*. Oxford University Press, 1977.

[And73] M. R. Anderberg. *Cluster analysis for applications*. Academic Press, 1973.

[ATS04] M. Z. Ashrafi, D. Taniar, and K. Smith. ODAM: An Optimized Distributed Association Rule Mining Algorithm. *IEEE Distributed Systems Online*, 2004.

[AwJS96] H. Abelson and G. J. Sussman with J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[Baa03] F. Baader. *The description logic handbook*. Cambridge University Press, 2003.

[Bal00] H. Balzert. *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag, 2000.

[BBK01] C. Boehm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys Journal*, 2001.

[Bec00] K. Beck. *Extreme Programming explained: embrace change*. Addison-Wesley, 2000.

[Ber02] P. Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, 2002.

[BF01] K. Beck and M. Fowler. *Extreme Programming planen*. Addison-Wesley, 2001.

[BG04] K. Beck and E. Gamma. *Contributing to Eclipse: Principles, Patterns and Plug-Ins*. Addison-Wesley, 2004.

[BKK96] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Databases*, 1996.

[BNKF98] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming*. Morgan Kaufmann Publishers, Inc., 1998.

[Bra97] J. Bradshaw. *Software Agents*. AAAI/MIT Press, 1997.

[BS04] S. Bickel and T. Scheffer. Multi-view clustering. In *Proceedings of the IEEE International Conference on Data Mining*, 2004.

[Car97] R. Caruana. Multitask learning. *Machine Learning Journal*, 1997.

[CHT99] N. Craswell, D. Hawking, and P. Thistlewaite. Merging results from isolated search engines. In *Proceedings of 10th Australasian Database Conference*, 1999.

[Com79] D. Comer. Ubiquitous b-tree. *ACM Computing Surveys Journal*, 1979.

[CPZ97] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, 1997.

[CSWH00] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[CSY99] W. Cohen, R. E. Schapire, and Y.Singer. Learning to order things. *Journal of Artificial Intelligence Research*, 1999.

[DMDH04] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology Matching: A Machine Learning Approach. In *Handbook on Ontologies in Information Systems*. Springer, 2004.

[EGV95] R. Johnson E. Gamma, R. Helm and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[EMH03] M. Eisenhardt, W. Müller, and A. Heinrich. Classifying documents by distributed P2P clustering. In *Proceedings of Informatik 2003*, 2003.

[Eva04] E. Evans. *Domain-Driven Design*. Addison-Wesley, 2004.

[FH03] J. Fürnkranz and E. Hüllermeier. Pairwise preference learning and ranking. In *Proceedings of the 14th European Conference on Machine Learning (ECML '03)*, 2003.

[FJ03] M. J. Fonseca and J. A. Jorge. Indexing highdimensional data for content-based retrieval in large databases. In *Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, 2003.

[FKMR02] Simon Fischer, Ralf Klinkenberg, Ingo Mierswa, and Oliver Ritthoff. Yale: Yet Another Learning Environment – Tutorial. Technical Report CI-136/02, Collaborative Research Center 531, University of Dortmund, Dortmund, Germany, Juni 2002. ISSN 1433-3325.

[Foo97] J. Foote. Content-based retrieval of music and audio. in multimedia storage and archiving systems 2. In *Proceedings of SPIE*, 1997.

[FPB78] Jr. Frederick P. Brooks. *The Mythical Man-Month: Essays on Software-Engineering*. Addison-Wesley, 1978.

[FPSS96] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From Data Mining to Knowledge Discovery: An overview. In *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

[Fuh04] N. Fuhr. *Information Retrieval, Skriptum zur Vorlesung*. 2004.

[FY99] B. Foote and J. Yoder. Big ball of mud. Technical report, Fourth Conference on Patterns Languages of Programs (PLoP 97/EuroPLoP 97), 1999.

[Gab91] R. Gabriel. Lisp: Good news, bad news, how to win big. In *Proceedings of the First European Conference on the Practical Application of Lisp*, 1991.

[GG98] V. Gaede and O. Guenther. Multidimensional access methods. *ACM Computing Surveys Journal*, 1998.

[GH04] D. Gondek and T. Hofmann. Non-redundant data clustering. In *Proceedings of the 4th IEEE International Conference on Data Mining*, 2004.

[GLCS95] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith. Query by humming: Musical information retrieval in an audio database. In *Proceedings of ACM Multimedia*, 1995.

[GR89] A. Goldberg and D. Robson. *Smalltalk 80: the language*. Addison-Wesley, 1989.

[Gra03] M. Granitzer. Hierarchical text classification using methods from machine learning. Master's thesis, Graz University of Technology, 2003.

[Gru93] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *Presented at the Padua workshop on Formal Ontology Journal*, 1993.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. ACM Press, 1984.

[Hah01] M.F.H. Hahn. Algorithmen für datamining. Master's thesis, Christian-Albrechts-Universität, Kiel, 2001.

[Har99] J. Hartung. *Multivariate Statistik*. Oldenbourg Verlag, 1999.

[HH89] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz*. Springer Verlag, 1989.

[HTF01] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Verlag, 2001.

[JKP04a] E. Januzaj, H. P. Kriegel, and M. Pfeifle. DBDC: Density-based distributed clustering. In *Proc. 9th Int. Conf. on Extending Database Technology (EDBT)*. Kluwer Academic Publishers, 2004.

[JKP04b] E. Januzaj, H. P. Kriegel, and M. Pfeifle. Scalable density-based distributed clustering. In *8th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD)*. Springer-Verlag Berlin Heidelberg, 2004.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 1999.

[KC01] F. Kurth and M. Clausen. Full-text indexing of very large audio data bases. In *Proceedings of the 110th Convention of the Audio Engineering Society*. Kluwer Academic Publishers, 2001.

[KKPR04] H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz. Effective decompositioning of complex spatial objects into intervals. In *Proceedings of International Conference on Data-bases and Applications (DBA)*, 2004.

[Knu98] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1998.

[Lad03] R. Laddad. *AspectJ in action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[LMM98] Y. Lashkari, M. Metral, and P. Maes. *Collaborative Interface Agents in: Readings in Agents*. AAAI Press, 1998.

[LWC98] Z. Liu, Y. Wang, and T. Chen. Audio feature extraction and analzsis for scene segmentation and clssification. *VLSI Signal Processing System Journal*, 1998.

[Mae97] P. Maes. *Modeling adaptive autonomous agents*. MIT Press, 1997.

[McC93] S. McConnell. *Code Complete*. Microsoft Press, 1993.

[Mie04] I. Mierswa. Automatisierte Merkmalsextraktion aus Audiodaten. Master's thesis, Fachbereich Informatik, Universität Dortmund, 2004.

[Mit97] T. Mitchell. *Machine Learning*. McGraw-Hill Press, 1997.

[MKFR03] Ingo Mierswa, Ralf Klinkenberg, Simon Fischer, and Oliver Ritthoff. A Flexible Platform for Knowledge Discovery Experiments: YALE – Yet Another Learning Environment. In *LLWA 03 - Tagungsband der GI-Workshop-Woche Lernen - Lehren - Wissen - Adaptivität*, 2003.

[MKL+02] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraga, J. Pruyne, B. Richard, S. Rollings, and Z. Xu. Peer-to-peer computing. Technical report, HP Laboratories Palo Alto, 2002.

[ML99] W. Meng and K.-L. Liu. Building efficient and effective metasearch engines. Technical report, Dept. of Computer Science, SUNY at Binghamton, NY, 1999.

[MM05] Ingo Mierswa and Katharina Morik. Automatic feature extraction for classifying audio data. *Machine Learning Journal*, 58:127–149, 2005.

[MUT+05] F. Mörchen, A. Ultsch, M. Thies, I. Löhken, M. Nöcker, C. Stamm, N. Efthymiou, and M. Kümmerer. Musicminer: Visualizing timbre distances of music as topographical maps. Technical report, 2005.

[MW05] I. Mierswa and M. Wurst. Efficient case based feature construction exploting constructed features without data. Technical report, Collaborative Research Center 531, University of Dortmund, Dortmund, Germany, Juni 2005.

[MWJ96] J. P. Müller, M. Wooldridge, and N. R. Jennings. Intelligent agents iii. In *Proceedings of the 1996 Workshop on Agent Theories, Architectures, and Languages (ATAL)*, 1996.

[NB02] J. Noble and R. Biddle. Notes on postmodern programming. Technical report, School of Mathematical and Computing Sciences, Victoria University, Wellington, New Zealand, 2002.

[Nwa96] H. S. Nwana. Software agents: An overview in knowledge engineering review. *Knowledge Engineering Review Journal*, 1996.

[Oes98] B. Oestereich. *Objektorientierte Softwareentwicklung*. Oldenbourg Verlag, 1998.

[O'L98] D. E. O'Leary. Enterprise knowledge management. *IEEE Computer Journal*, 1998.

[Omi01] A. Omicini. *Coordination of Internet Agents*. Springer Verlag, 2001.

[Pet96] C. J. Petrie. Agent-based engineering, the web and intelligence. *IEEE Expert Intelligent Systems Their Applications Journal*, 1996.

[Qui93] J. R. Quinlan. *Programs for machine Learning*. McGraw-Hill Press, 1993.

[Rad04] M. Radmacher. Sicherheits- und scwachstellenanalyse entlang des wireless-lan-protokollstacks. 2004.

[Rec04] J. Rech. *Wireless LANs*. Heise Verlag, 2004.

[RFH+01] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable and content-adressable network. In *Proceedings of ACM SIGCOMM*, 2001.

[RIS+94] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: An open architecture for collaborative filtering of netnews. In *Proceedings of CSCW 94 Conference on Computer Supported Cooperative Work*. ACM Press, 1994.

[Ris00] L. Rising. *The Pattern Almanac 2000*. Addison-Wesley, 2000.

[RK02] A. Ribbrock and F. Kurth. Full-text retrieval approach to content-based audio identification. In *Proceedings of International Workshop on Multimedia Signal Processing*, 2002.

[RKF+01] O. Ritthoff, R. Klinkenberg, S. Fischer, I. Mierswa, and S. Felske. Yale: Yet another machine learning environment. In *LLWA 01 – Tagungsband der GI-Workshop-Woche Lernen – Lehren – Wissen – Adaptivität*, 2001.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.

[Rot02] J. Roth. *Mobile Computing*. Dpunkt Verlag, 2002.

[SF00] R. G. Smith and A. Farquhar. The road ahead for knowledge management. *AI Perspective, AI Magazine Journal*, 2000.

[SGG00] A. Silberschatz, P. Galvin, and G. Gagne. *Applied operating system concepts*. Wiley, 2000.

[SL01] A. Sun and E.-P. Lim. Hierarchical text classification and evaluation. In *Proceedings of the ICDM 2001*, 2001.

[SNB01] W. Shen, H. Norrie, and J-P. A. Barthes. *MAS for concurrent intelligent design and manufactoring*. Taylor and Francis Press, 2001.

[Som01] I. Sommerville. *Software Engineering*. Addison-Wesley, 2001.

[Tan03] A. S. Tanenbaum. *Computernetzwerke*. Prentice Hall Press, 2003.

[Thr95] S. Thrun. Lifelong leraning: A case study. Technical Report CS-95-208, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, 1995.

[TM95] S. Thrun and T. Mitchell. Learning one more thing. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann, 1995.

[TO98] S. Thrun and J. O'Sullivan. *Clustering learning tasks and the selective cross-task transfer of knowledge*. Kluwer Academic Publishers, 1998.

[TP98] S. Thrun and L.Y. Pratt. *Learrning to learn: Introduction*. Kluwer Academic Publishers, 1998.

[Tza02] G. Tzanetakis. *Manipulation, Analysis and Retrieval Systems for Audio Signals*. PhD thesis, Computer Science Department, Princeton University, June 2002.

[Ung91] D. Ungar. Self: The power of simplicity. *Lisp and Symbolic Computation: An Internationl Journal*, 1991.

[Wei99] G. Weiss. *Multiagent Systems*. MIT Press, 1999.

[Wes04] F. Westphal. *Akzeptanztest mit Fit*. Dpunkt Verlag, 2004.

[WF01] I. Witten and E. Frank. *Data Mining: Praktische Werkzeuge und Techniken fuer das maschinelle Lernen*. Carl Hanser Verlag, 2001.

[WMM05] M. Wurst, I. Mierswa, and K. Morik. Structuring music collections by exploiting peers' processing. Technical report, Sonderforschungsbereich 475, Universität Dortmund, 2005.

[Zak99] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 1999.