Studienarbeit

Suche in anwendungsfachlichen Persistenzmedien am Beispiel der JWAM-Registratur

Björn Ostermann
Gärtnerstraße 105
20253 Hamburg
5osterma@informatik.uni-hamburg.de

Mai 2001

Betreuung: Prof. Dr.-Ing. Heinz Züllighoven

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Die ständig steigende Bedeutung des Computers als „Allzweckwerkzeug“ für die Bearbeitung und Speicherung jeglicher Form von Daten, Materialien und Medien macht es notwendig, sich mit der Verwaltung dieser Daten zu beschäftigen. Wichtig ist zunächst die Möglichkeit, Daten nach der Bearbeitung ablegen zu können, um sie zu einem späteren Zeitpunkt wieder einlesen zu können. Ist die Anzahl der zu sichernden Daten gering, werden meist auf einfache Mechanismen benutzt: Entweder wird jedes Datum in eine einzelne Datei gespeichert, oder alle Daten werden, zum Beispiel bei Java mit Hilfe der Serialisierung, in eine gemeinsame Datei gesichert. Bei größeren Datenbeständen kommen Datenbanken zum Einsatz, da nur sie hohe Zugriffsgeschwindigkeiten garantieren. Unterschieden werden kann hierbei nach dem Paradigma der verwendeten Datenbank (relational, objektorientiert, etc.).
Eine entscheidende Frage bei allen Speicherlösungen ist die Frage nach effizienten Suchmöglichkeiten. Eine Anfrage an das Speichermedium nach allen Daten, die ein vorgegebenes Suchkriterium erfüllen, soll in möglichst kurzer Zeit ein Ergebnis liefern. Dieses Ziel kann dadurch erreicht werden, daß die Suche optimal an das Speichermedium angepaßt ist (beispielsweise durch Einsatz von „Query“-Anfragen bei relationalen Datenbanken).
Andererseits bedeutet die Anpassung einer Anwendung an ein Speichermedium und die dazugehörige Suche eine Festlegung, die meist nicht erwünscht ist. Beim Entwurf einer Anwendung ist oft noch keine Entscheidung für ein konkretes Speichermedium getroffen worden, oder es wird geplant die Anwendung mit verschiedenen Speichermedien zu benutzen. Es ist auch möglich, daß für die Entwicklung nicht das gleiche Speichermedium wie für den späteren Einsatz zur Verfügung steht.
Daher ist ein Lösung erforderlich, mit der es unabhängig von dem eingesetzten Speichermedium möglich ist, Daten zu sichern und nach ihnen zu suchen. Gleichzeitig soll es möglich sein, schnelle Suchfunktionen des Speichermediums, falls vorhanden, auszunutzen.

1.2. Kontext der Arbeit

Diese Studienarbeit entsteht im Rahmen der Entwicklung des JWAM-Rahmenwerkes am Arbeitsbereich Softwaretechnik der Universität Hamburg (näheres hierzu siehe [Züllighoven 98]).
Bei der Entwicklung dieses Rahmenwerkes entstand der Bedarf nach einer Lösung, die über Arbeitsplatzgrenzen hinaus kooperatives Arbeiten ermöglicht. Hierfür wurde von A. Havenstein im Rahmen seiner Studienarbeit die „JWAM-Registratur“ entwickelt (siehe [Havenstein 00]). Es handelt sich hierbei um eine Materialverwaltung, die sich an den fachlichen Konzepten einer Dokumentenverwaltung von (Papier-) Akten orientiert.
Sie erlaubt es, Materialien an einem Arbeitsplatz in die Registratur zu stellen und dann von einem anderen Arbeitsplatz darauf zuzugreifen. Dadurch ist die Möglichkeit einer impliziten Kooperation gegeben. Die ursprüngliche Version der Registratur benutzte nur den von Java zur Verfügung gestellten Serialisierungsmechanismus. Mittlerweile ist die Registratur mehrfach überarbeitet worden – unter anderem wurde einen Anbindung an relationale Datenbanken realisiert.
Ein wichtiger Punkt der bisher im Rahmen der Entwicklung nur unzureichend gelöst wurde, ist die Suche in der Registratur. Das einzige im JWAM-Rahmenwerk zur Verfügung stehende Werkzeug hierfür ist der „Text-Schnüffler“ (Text-Sniffer): Es handelt sich hierbei um ein

Werkzeug zur Volltextsuche über den gesamten Inhalt der Registratur, das die Anforderungen aber oft nur unzureichend erfüllt, da es nur generisch suchen kann. Für erweiterte Suchmöglichkeiten ist es nötig, für jede Anwendung eine eigene Suche zu implementieren.

1.3. Zielsetzung

Ziel der Arbeit ist die Entwicklung und Evaluierung eines Suchmechanismus, der es ermöglicht Anwendung und Persistenzmedium voneinander zu trennen. Gleichzeitig sollen dabei die Vorteile, die das zugrundeliegende Speichermedium bezüglich Optimierung der Zugriffsgeschwindigkeit bietet, möglichst gut ausgenutzt werden.
Dieser Suchmechanismus wird dann in das Konzept der Registratur integriert. Anhand einer Beispielapplikation werden Vor- und Nachteile der alten und der neuen Implementation dargestellt.

1.4. Übersicht über die Arbeit

Kapitel 2 beschäftigt sich mit den Grundlagen des Suchens. Der Begriff des „Suchens“ wird definiert und der Einsatzkontext von Suchfunktionen beschrieben. Die grundlegenden Konzepte von relationalen Datenbanken und der Begriff des Schnüfflers werden erörtert.
Im dritten Kapitel wird die Situation analysiert, wie sie zur Zeit bei Entwicklung mit JWAM und Zugriff auf Persistenzmedien gegeben ist. Hierbei wird zunächst auf die vorhandenen Konzepte wie Registratur und Mapper eingegangen. Anschließend werden verschiedene Beispiele aus dem Kontext von WAM, die Suchfunktionen benutzen, vorgestellt.
Das vierte Kapitel beschäftigt sich mit Lösungen, die nicht aus dem Umfeld von JWAM stammen. Hier wird zunächst allgemein und dann am Beispiel der Anwendung „TopLink“ erklärt, wie Objekte auf relationale Datenbanken abgebildet werden können. Anschließend wird auf einen eventuell zukünftigen Standard für eine Java-Persistenzschnittstelle JDO eingegangen.
Kapitel 5 stellt die während der Arbeit entwickelten Lösungen für eine Erweiterung der Suchfunktionalität der Registratur vor.
Im letzten Kapitel werden die Ergebnisse der Arbeit noch einmal zusammengefaßt und ein Ausblick auf weiterführende und offene Fragen gegeben.

1.5. Danksagung

Danken möchte ich an dieser Stelle Prof. Dr. Heinz Züllighoven, Henning Wolf und insbesondere Martin Lippert, die mir beim Entstehen dieser Studienarbeit durch ihre Hinweise und ihre Kritik behilflich waren.

2. Grundlagen zur Suche

In diesem Kapitel wird allgemein auf die Tätigkeit des Suchens eingegangen. Dazu wird zunächst der Begriff des Suchens definiert. Anschließend folgt eine Beschreibung des Einsatzkontextes von Suchfunktionen. Es wird beschrieben, wie Informationen in relationalen Datenbanken abgelegt sind und wie Suchanfragen an diese aussehen. Abschließend wird erörtert, wie sich Schnüffler in den WAM-Ansatz einordnen lassen.

2.1. Begriffsdefinition „Suchen“

Die vorliegende Studienarbeit beschäftigt sich mit der Suche in anwendungsfachlichen Behältern, insbesondere mit der Suche in der JWAM-Registratur. Diese Tätigkeit, das Suchen, ist wie folgt definiert:
„Suchen: Bezeichnet einen Vorgang, bei dem in einem gegebenen Datenbestand alle Datenobjekte gefunden und dem aufrufenden Benutzer oder Programm zur Verfügung gestellt werden, die einer angegebenen Suchbedingung genügen.“ [Schneider 97]
Der Datenbestand, der durchsucht werden soll, kann sowohl komplett im Hauptspeicher als auch in einem Sekundärspeicher liegen. Für diese Arbeit ist der Fall interessant, bei dem der Datenbestand auf einem Sekundärspeicher, wie zum Beispiel einer Festplatte, liegt.
Bei Datenobjekten kann es sich um jede Art von Daten handeln, die sich in einem Speichermedium sichern lassen. Möglich ist alles von einfachen Strings oder numerischen Werten bis zu komplexen Konstrukten. In den weiteren Betrachtungen sind diese Datenobjekte oft Objekte im Sinne der objektorientierten Programmierung.
Nach der Art der Suchbedingung können Suchvorgänge unterschieden werden (vgl. [Schneider 97]). Die einfachste Art ist das Suchen mit einem Suchschlüssel: Es wird ein Feld und für dieses ein Schlüsselwert vorgegeben. Alle Datensätze die in diesem Feld den angegebenen Wert haben, gehören zum Ergebnis. Ist sichergestellt, daß nur ein Datensatz dieser Bedingung genügt, handelt es sich um einen Primärschlüssel. (Beispiel: „Gesucht ist der Mitarbeiter mit der Personalnummer 4711.“) Bei der Suche mit einer logischen satzinternen Suchbedingung können mehrere Felder eines Datensatzes benutzt werden. Neben Gleichheit sind auch andere Vergleiche wie „größer“, „ungleich“, etc. möglich. Einzelergebnisse können durch „und“, „oder“ bzw. „nicht“ verknüpft werden. (Beispiel: „Gesucht werden alle Mitarbeiter, die weiblich und nicht älter als 30 Jahre sind.“) Schließlich gibt es die Suche mit einer logischen satzübergreifenden Bedingung. Hier sind auch Vergleiche zwischen Feldern aus verschiedenen Datensätzen erlaubt. (Beispiel: „Gesucht sind alle Mitarbeiter, die mehr verdienen als ihr Abteilungsleiter.“)
Die in Kapitel 5 vorgestellten Suchfunktionen gehören in die zweite Kategorie der hier genannten Suchvorgänge. Die Suche mit Suchschlüssel wird praktisch bei allen in dieser Arbeit vorgestellten Lösungen benutzt. Sie wird verwendet, um einzelne Objekte aus der Registratur in den Hauptspeicher zu laden.

2.2. Einsatzkontext von Suchfunktionen

Suchfunktionen werden überall dort benötigt, wo aus einer großen Menge von Daten nur solche benötigt werden, die bestimmte vorgegebene Eigenschaften besitzen. Daher findet man Suchfunktionen in alle Bereichen, in denen Daten gespeichert sind und der Zugriff auf diese wahlfrei erfolgen soll.

Das Einsatzfeld von Suchfunktionen ist extrem vielfältig: Es reicht von Internetsuchmaschinen, bei denen der zu durchsuchende Datenbestand praktisch aus allen zugreifbaren Internetseiten besteht, bis zu einfachen Fahrplanauskünften, die zu einer Uhrzeit die Abfahrt des nächsten Zuges suchen.
Suchfunktionen in Kontext von WAM arbeiten meist auf einem eingrenzbaren Datenbestand, zum Beispiel dem Inhalt einer Datenbank. Gesucht wird nach Objekten, die bestimmte Eigenschaften erfüllen. Diese Eigenschaften können in binärer Form abgefragt werden (z.B. Suche nach allen Kunden, die zuverlässig eingestuft sind) oder mit Hilfe von Vergleichen (z.B. Suche nach allen Kunden, die über 10000 DM Jahresumsatz haben).

2.3. Suchen in Datenbanken

Für die Speicherung und Verwaltung größerer Datenbeständen in elektronischer Form werden oft Datenbanken eingesetzt. Dies ist dadurch begründet, daß sie einfacheren Speichermechanismen, wie etwa der Verwendung einzelner Dateien in einem Dateisystem, überlegen sind. Datenbanken sind schnell bei der Suche und beim Zugriff auf Datensätze, sie können verhindern, daß es zu inkonsistenten Zuständen infolge halb ausgeführter Transaktionen kommt und sie erlauben mehreren Benutzern auf dem selben Datenbestand zu arbeiten. Weitere Vorteile von Datenbanken finden sich zum Beispiel bei Date (vgl. [Date 00]).
Da Datenbanken eines der am häufigsten eingesetzten Speichermedien für große Datenbestände sind, sind sie auch das Einsatzfeld für die meisten Suchfunktionen. Daher soll hier kurz auf den am weitesten verbreitetet Typ von Datenbanken, den relationalen Datenbanken, eingegangen werden.
Die Speicherung von Daten erfolgt in relationalen Datenbanken in Tabellen. Jede Tabelle besteht aus einer festen Anzahl von Spalten, und einer variablen Anzahl von Zeilen. Eine Zeile steht für einen Datensatz, eine Spalte für ein Attribut, das alle Datensätze in dieser Tabelle besitzen. In einem Feld steht für einen Datensatz der Wert eines Attributes.

Kunden:

Kundennr.	Name	Ort	Umsatz
1243	Müller	Hamburg	20.000
1356	Schmidt	Kiel	15.000
2178	Meyer	Hamburg	25.000

Rechnungen:

Rechnungsnr.	Kundennr.	Betrag
8247	1356	1000
8248	2178	3000
8249	1356	2000

Abbildung 1: Beispiel für Tabellen einer relationalen Datenbank

Meist gibt es in einer Tabelle eine Spalte, anhand der jeder Datensatz eindeutig identifiziert werden kann (Kundennummer in der Kundentabelle). Mit Hilfe dieses Primärschlüssels können Datensätze aus verschiedenen Tabellen einander zugeordnet werden (zum Beispiel alle Rechnungen, die zu einem Kunden gehören). Weitere Informationen zu diesem Thema finden sich bei Date (vgl. [Date 00]).
Um auf relationale Datenbanken zuzugreifen, hat sich als Standard die Sprache SQL etabliert (ursprünglich Abkürzung für Structured Query Language, siehe [Date 87]). Diese bereits 1970 von IBM für die eigenen Datenbanksysteme entwickelte Sprache wurde von vielen

anderen Herstellern übernommen und wird heute von praktisch jeder relationalen Datenbank unterstützt.
Für die Suche in einer Datenbank wird in SQL der Befehl `SELECT` verwendet. Ein einfacher SQL-Ausdruck besitzt folgendes Schema:

```
SELECT Attribute FROM Tabelle WHERE Ausdruck
```

Attribute steht für die Attribute, die zum Ergebnis gehören sollen (z.B. Name oder Ort; * steht für alle vorhandenen Attribute). Tabelle bezeichnet den Namen der Tabelle aus denen die Datensätze geladen werden sollen (hier: Kunden). Ausdruck bezeichnet einen logischen Ausdruck, der für die Datensätze, die zum Suchergebnis gehören sollen, wahr sein muß (z.B. Umsatz > 16000). Eine Suchanfrage, die die Namen aller Kunden mit einem Umsatz größer ist als 16000 sucht, sieht folgendermaßen aus:

```
SELECT Name FROM Kunden WHERE Umsatz > 16000
```

Das Ergebnis dieser Anfrage sind die beiden Namen „Müller“ und „Meyer“.
Abfragen, die auf Daten aus mehreren Tabellen zugreifen, sind ebenfalls möglich. Diese Abfrage liefert, die beiden Rechnungsnummern, die zu dem Kunden „Schmidt“ gehören.

```
SELECT Rechnungsnr FROM Rechnungen WHERE
Rechnungen.Kundennr = Kunden.Kundennr AND Name = ‘Schmidt’
```

Wie man an den Beispielen sieht, ist es bei relationalen Datenbanken möglich, einzelne Attribute gezielt zu suchen und auf sie direkt zuzugreifen. Dies erlaubt zwar eine schnelle Suche, durchbricht aber gleichzeitig bei objektorientierten Anwendungen aber das Konzept der Datenkapselung, das ein Zugriff auf einzelne Attribute nur über festgelegte Methoden erlaubt.
Objektorientierte Datenbanken haben dieses Problem nicht, da sie Objekte als Ganzes einschließlich der Zugriffsmethoden speichern. Der Wert eines Attributes kann somit nur über die Methode abgefragt werden, die der Entwickler dafür vorgesehen hat. Sie sind dafür beim Zugriff und bei der Suche langsamer. Außerdem fehlt bis jetzt ein anerkannter, einheitlicher Zugriffsstandard für objektorientierte Datenbanken, der vergleichbar mit SQL bei relationalen Datenbanken ist. Es gibt zwar die standardisierte Anfragesprache OQL (Object Query Language, siehe [Cattell 93]), allerdings wird diese bisher von den wenigsten Datenbank-systemen unterstützt

3. Analyse der aktuellen Situation im JWAM-Rahmenwerk

Nach dem im vorigen Kapitel die Grundlagen zur Suche im allgemeinen dargestellt wurden, wird im nun folgenden Abschnitt auf die Suche im JWAM-Rahmenwerk eingegangen.
Zunächst wird in einem kurzen Überblick die JWAM-Registratur beschrieben, da sie die Grundlage für alle weiteren hier vorgestellten Lösungen ist. Anschließend folgt eine Vorstellung aller im Rahmenwerk vorhandenen Werkzeuge, mit denen eine Suchfunktion realisiert werden kann. Es handelt sich hierbei um das Schnüffler-Konzept und das darauf aufbauende Finder-Werkzeug. Der praktische Einsatz von Schnüfflern wird am Beispiel des SAB-Projektes gezeigt. Thema des nächsten Abschnitts sind Werkzeuge, die eigene Lösungen zur Suche in der Registratur benutzen. Als Beispielanwendung dient hier der HelpDesk.
Abschließend erfolgt ein Vergleich der vorgestellten Lösungen.

3.1. Die JWAM-Registratur

Bei der Registratur handelt es sich um ein Kooperations- und Persistenzmedium, das im Laufe der Entwicklung des JWAM-Rahmenwerkes entstand. Es ist als Ausstattungskomponente im Rahmenwerk realisiert.
Bei der Betrachtung von Arbeitsvorgängen in der realen Welt stellt sich schnell heraus, daß viele dieser Vorgänge nicht von einer Person, sondern von mehreren durch Zusammenarbeit erledigt werden (vgl. [Züllighoven 98]). Um auch solche Vorgänge auf Software abbilden zu können, wurde der WAM-Ansatz so erweitert, daß auch die Unterstützung kooperativer Arbeit möglich ist. Ein Modell zur Kooperation, das aus den Verwaltungswissenschaften stammt, ist die Benutzung einer gemeinsamen Registratur. Jeder Kooperationspartner kann in diese Registratur Materialien hineinlegen, die dann registriert werden. Genauso kann jeder Kooperationspartner Materialien aus der Registratur entleihen. Dabei wird vermerkt, beispielsweise auf einer Fehlkarte, wer das Material entliehen hat. Andere Benutzer der Registratur können sich dann mit dem Entleiher koordinieren, wenn sie ebenfalls Zugriff auf dieses Material benötigen. Da die anderen Kooperationspartner nicht direkt im Benutzungsmodell der Registratur sichtbar sind, sondern nur ihre Spuren (neue Dokumente, Fehlkarten), handelt es sich hierbei um eine Form der impliziten Kooperation. Eine genauere Beschreibung dieser und weiterer Kooperationsmöglichkeiten finden sich bei Züllighoven (vgl. [Züllighoven 98]). In dieser Quelle wird für das Kooperationsmedium noch der Begriff Archiv verwendet. Dieser Begriff ist inzwischen durch Registratur ersetzt worden, da dies den Verwendungszweck besser beschreibt.
Neben der Unterstützung der Kooperation ist eine Registratur auch dazu geeignet, das bis dahin bestehende Problem der Persistenz zu lösen: Daten, die in der Registratur gespeichert sind, bleiben über das Ende der Benutzung einer Anwendung erhalten. Ein erster Entwurf einer Registratur wurde von Skutta und Thiel in ihrer Studienarbeit entwickelt (siehe [Skutta & Thiel 99]). Diese Registratur konnte allerdings nur die Daten der Anwendung, für die sie entwickelt wurde (ein Pausenplan), verarbeiten.

3.1.1. Aufbau der Registratur

Eine anwendungsunabhängige Umsetzung des Registraturkonzeptes wurde durch Havenstein in seiner Studienarbeit entwickelt und implementiert (siehe [Havenstein 00]). Hierbei wurde die Funktionalität in zwei Komponenten aufgeteilt. Die erste Komponente ist die Registratur, ein fachlicher, persistenter Behälter, in dem Materialien jedes Typs und jeder Größe

gespeichert werden können. Die zweite Komponente ist der Registrator, der die Verwaltung über alle in der Registratur gespeicherten Materialien übernimmt. Alle Zugriffe auf die Registratur erfolgen nur über ihn. Damit wird die Konsistenz der Registratur gesichert. Dieses Konzept der Registratur wurde in einer überarbeiteten Version in das JWAM-Rahmenwerk integriert und als Standard zur Materialablage übernommen.
Die Registratur wird, genauso wie das gesamte Rahmenwerk, laufend weiterentwickelt. Daher unterscheidet sich die aktuelle Version mittlerweile erheblich von der ursprünglichen Version.

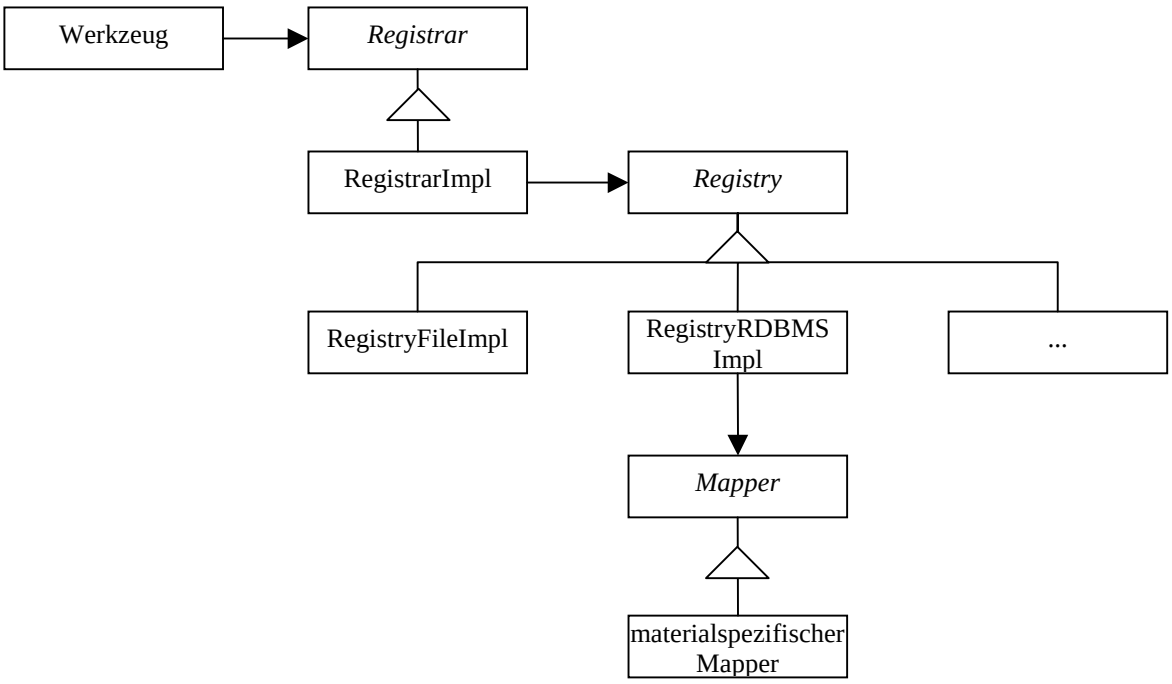


Abbildung 2: Architektur der Registratur

Abbildung 2 zeigt den Aufbau der Registratur in der JWAM-Version 1.5: Ein Werkzeug, das auf die Registratur zugreift, kennt nur die Schnittstelle des Registrators (Registrar). Hierfür ist eine Standardimplementierung (RegistrarImpl) im Rahmenwerk vorhanden. Es können aber auch eigene Registratoren eingesetzt werden, solange sie die Schnittstelle erfüllen. Das Werkzeug stellt seine Anfrage, zum Beispiel nach einem Material, an den Registrator. Der prüft dann in der Registratur, ob dieses Material vorhanden ist, und somit die Anfrage beantwortet werden kann. Der Registrator kennt die genaue Implementierung der Registratur nicht. Er benutzt nur die Methoden, die in der Registratur-Schnittstelle (Registry) vorhanden sind. Dadurch können ohne Änderungen an der Architektur verschiedene Registraturen eingesetzt werden, die auf völlig unterschiedlichen Speichermedien arbeiten. Die einfachste und anfangs einzige Implementierung einer Registratur im Rahmenwerk ist eine Registratur, die auf dem Dateisystem arbeitet (RegistryFileImpl). Sie benutzt die Java-Serialisierung und speichert anschließend alle zu sichernden Daten in einer einzelnen Datei.

3.1.2. Das Mapper-Konzept

Im Zuge der Weiterentwicklung des Rahmenwerkes wurde die Registratur um eine Anbindung an relationale Datenbanken ergänzt. Mit Hilfe dieser Anbindung ist es möglich, neben der bisher genutzten Dateiregistratur, relationale Datenbanken als Persistenzmedium zu

verwenden. Die Anbindung basiert auf einem Abbildungs- (Mapping-) konzept, bei dem Objektattribute Datenbanktabellen zugeordnet werden. Für jede Klasse, von der Objekte in der Datenbank gespeichert werden sollen, muß eine zugehörige Abbildungsklasse existieren (Namenskonvention: *Klassenname*Mapper). In dieser Abbildungsklasse, dem Mapper, ist festgelegt, auf welche Tabelle ein Objekt und auf welche Spalten dieser Tabelle seine Attribute abgebildet werden. Außerdem beinhaltet die Abbildungsklasse den kompletten Programmcode, der den Zugriff auf die Datenbank ausführt. Verwendet werden hierfür SQL-Anweisungen, die an die Datenbank weitergeleitet werden und die gewünschte Funktion in der Datenbank ausführen. Damit kann mit wenigen Änderungen jede relationale Datenbank, die den JDBC-Standard (Java Database Connectivity, siehe [Hamilton et. al. 97]) erfüllt, als Persistenzmedium verwendet werden. Benutzt wird dieser Abbildungsmechanismus von der Relationalen-Datenbank-Registratur (RegistryRDBMSImpl). Jeder konkrete material-bezogene Mapper muß eine Unterklasse der gleichnamigen abstrakten Oberklasse sein. Damit wird sichergestellt, daß die Registrierung mit jedem Mapper zusammenarbeiten kann.
Die Speicherung eines Materials mit in einer relationalen Datenbank läuft folgendermaßen ab: Der Registrator ruft die Speichermethode der Registratur mit dem zu sichernden Material auf. Die Registratur überprüft, ob für das Material ein passender Mapper vorhanden ist. Fällt die Prüfung positiv aus, wird die Speichermethode des zu dem Material passenden Mappers aufgerufen (wieder mit dem Material als Parameter). Die Speichermethode des Mappers fragt alle Attribute des Materials ab und sichert sie in den passenden Spalten der zum Material gehörenden Datenbanktabelle.
Der Nachteil an diesem Konzept ist, daß im Mapper sämtliche Informationen wie und auf welche Tabelle ein Material abgebildet werden soll, vorhanden sein muß. Eine Änderung am Material oder an der Datenbank erfordert immer eine Anpassung des zugehörigen Mappers.

3.2. Suchen mit Hilfe von Schnüfflern

Schnüffler sind kleine Programme, die von JWAM-Anwendungen dazu benutzt werden, auf Behältern nach Objekten mit bestimmten Eigenschaften zu suchen. Sie werden vor ihrem Einsatz mit Suchkriterien konfiguriert und arbeiten nachdem sie gestartet worden sind autonom, bis sie ihre Suche beendet haben.
Eingeführt wurden sie, weil es in der aktuellen Implementierung der Registratur bei großen Datenbeständen nicht praktikabel ist, das gesamte Inhaltsverzeichnis (`dvTableOfContents`) des Behälters einzulesen, um darauf eine Suche durchzuführen. Der Grund hierfür ist, daß im Inhaltsverzeichnis praktisch sämtliche Informationen enthalten sind, die sich in dem zu durch-suchenden Behälter befinden. Dazu kommen noch Verwaltungsinformationen über jedes abgespeicherte Objekt. Dies führt bei größeren Datenbeständen zu unverhältnismäßig großen Inhaltsverzeichnissen.
Liegt der Datenbestand, wie zum Beispiel in verteilten Umgebungen, auf einem anderen Rechner wie der Anwendungsprozeß, müßte, verbunden mit erheblichem Zeitaufwand, das gesamte Inhaltsverzeichnis über das Netz transportiert werden. Schnüffler dagegen werden vom Anwendungsrechner auf den Rechner, auf dem der Datenbestand liegt, weitergeleitet und führen dort den Suchvorgang aus. Somit müssen nur die Ergebnisse, die der Schnüffler gefunden hat, über das Netz geschickt werden.

3.2.1. Wie lassen sich Schnüffler einordnen?

Eine interessante Frage die sich stellt, wenn man sich mit dem Konzept des Schnüfflers beschäftigt ist, wie sich Schnüffler in den WAM-Ansatz einordnen lassen. Kann man Schnüffler als Automat bezeichnen oder muß für sie der WAM-Ansatz um eine neue Metapher erweitert werden. Ein Begriff, der hier sofort ins Blickfeld gerät, ist der Begriff des „Agenten“. Als Agent wird zur Zeit eine Vielzahl von Softwareprodukten bezeichnet, ohne daß wirklich definiert ist, was einen Agenten eigentlich auszeichnet. Es gibt keine anerkannte, allgemeingültige Definition für Agenten. Je nach Fachrichtung der Informatik werden andere Anforderungen gestellt. Eine Suche im Internet erbrachte in kürzester Zeit über zwanzig Definitionen, die sich zum Teil erheblich unterscheiden. Gemeinsamkeiten in diesen Definitionen findet man darin, daß Agenten autonom arbeiten und eine gewisse Eigen-intelligenz besitzen, mit der sie eine ihnen gestellte Aufgabe lösen.
Der Begriff des Automaten ist dagegen, zumindest im Umfeld von WAM, klar definiert: „Automaten sind im Rahmen einer zu erledigenden Aufgabe ein Arbeitsmittel, um Material zu bearbeiten. Sie erledigen lästige Routinetätigkeiten als eine definierte Folge von Arbeitsschritten mit festem Ergebnis ohne weiter äußere Eingriffe.“ ([Züllighoven 98], S. 89)
Die Aufgabe von Schnüfflern, die Suche in verschiedenen Behältern, ist meiner Meinung nach auch eine Routinetätigkeit. Schnüffler liefern ein definiertes Ergebnis, das Suchergebnis, und arbeiten nach ihrer Konfiguration ohne Eingriffe von außen. Die genaue Folge von Arbeitsschritten ist bei der Suche mit Schnüfflern nicht unbedingt festgelegt. Schnüffler können die Suche zum Beispiel abhängig vom zu durchsuchenden Behälter gestalten. Hier aber bereits von intelligentem Verhalten zu sprechen, halte ich für übertrieben.
Meiner Meinung nach ist die Metapher des Automaten, wenn auch mit Einschränkungen, durchaus für Schnüffler passend. Ich verwende daher für Schnüffler den Bergriff des speziellen Automaten, der auch bei Gryczan verwendet wird (vgl. [Gryczan et al. 00]).

3.2.2. Funktionsweise von Schnüfflern

Schnüffler können auf allen Behältern arbeiten, von denen die Schnittstelle „durch-schnüffelbar“ (`sniffable`) implementiert wird. Diese Schnittstelle beinhaltet eine Methode, die ein Inhaltsverzeichnis (`dvTableOfContents`) des Behälters liefert. Es handelt sich hierbei um einen Fachwert, der aus einer Liste von Beschreibungen aller Gegenstände (`dvThingDescription`) besteht, die sich in dem Behälter befinden. Das Inhaltsverzeichnis wird für den Suchvorgang benötigt.

```
public void sniffIn(Sniffable space);
public boolean hasSniffSpace();
public Sniffable sniffSpace();
public dvUserIdentificator owner();
public void runInThread();
public void run();
public boolean isRunning();
public dvTableOfContents result();
```

Abbildung 3: Schnittstelle des Schnüfflers

Die für jeden Schnüffler notwendigen Methoden sind in der Schnittstelle Schnüffler (`Sniffer`) festgelegt (siehe Abbildung 3), die von jedem Schnüffler implementiert werden muß. Auf dieser Schnittstelle basierend können eigene Schnüffler gebaut werden. Es steht

aber auch eine Standardimplementierung (`SnifferImpl`) im Rahmenwerk zur Verfügung, mit deren Hilfe die Konstruktion eines eigene Schnüfflers leichter fällt.
Ein Suchvorgang mit der Standardimplementierung des Schnüfflers läuft nach folgendem Schema ab: Zunächst wird von der Anwendung, die den Suchvorgang auslöst, ein neues Exemplar des Schnüfflers erzeugt (`new SnifferImpl()`). Dieser Schnüffler wird an den Rechner weitergeleitet, auf dem sich das zu durchsuchende Speichermedium (z.B. die Registratur) befindet. (Es kann sich hierbei auch um den selben Rechner handeln.) Danach wird im Schnüffler die Methode aufgerufen, die den zu durchsuchenden Behälter festlegt (`sniffIn (Sniffable space)`). Der eigentliche Schnüffeldurchgang wird anschließend entweder normal (`run()`) oder als eigenständiger Thread (`runInThread()`) gestartet. Ist der Suchvorgang beendet, kann sich die aufrufende Anwendung das Ergebnis der Anfrage in Form eines Inhaltsverzeichnis vom Schnüffler geben lassen (`result()`).

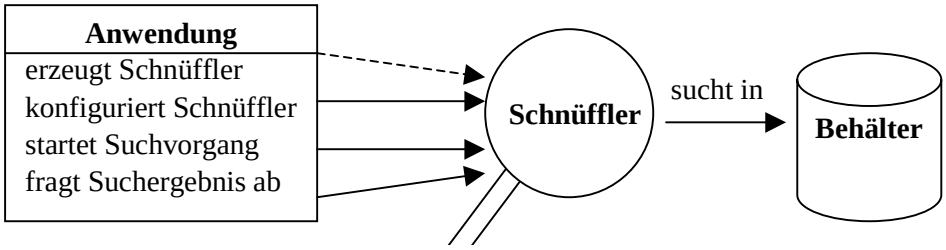


Abbildung 4: grobes Schema eines Suchvorgangs

Wie der eigentliche Suchvorgang abläuft ist nicht festgelegt. Die Standardimplementierung besitzt eine Methode zu Ermittlung des Suchergebnises (`doRun()`), die allerdings das komplette Inhaltsverzeichnis als Ergebnis zurückliefert. Für einen sinnvollen Schnüffler muß diese Methode in einer Unterklasse überschrieben werden. Ein Beispiel dafür ist der Text-Schnüffler (`TextSniffer`). Er ist Bestandteil des Rahmenwerkes und ist als Suchwerkzeug für die Registratur vorgesehen. Bei seiner Erzeugung wird der zu suchende Text als Suchkriterium (`criteria`) übernommen. Außerdem werden Name des Schnüfflers und sein Besitzer festgelegt. Seine `doRun`-Methode implementiert die Suche, indem sie sich das Inhaltsverzeichnis anfordert und dann für jedes Element prüft, ob es mit dem Suchkriterium übereinstimmt. Wenn ja, dann wird es dem Suchergebnis hinzugefügt. Die Übereinstimmung wird durch ein Vergleich mit der Java-Methode „`indexOf`“ gelöst, die prüft, ob ein Teilstring in einem anderen String vorkommt. Daher ist mit dem Textschnüffler nur eine Volltextsuche möglich.

3.2.3. Der Einsatz von Schnüfflern am Beispiel des Finders

Der Finder ist kleines generisches Werkzeug zur Suche in der Registratur. In der JWAM-Version 1.5, auf der diese Arbeit basiert, befindet er sich noch im Bereich des Rahmenwerkes, der für noch zu überarbeitende Pakete vorgesehen ist (jwamalpha). Daher können sich bis zur entgültigen Integration Änderungen ergeben.
Das Werkzeug bietet die Möglichkeit, einen Text einzugeben nach dem gesucht wird und die Anzahl der Ergebnisse zu beschränken. Startet man die Suche, wird ein neuer Textschnüffler erzeugt, der den Namen des aktuellen Benutzers und den zu suchenden Text als Parameter hat. Die Suchmethode wird im Registrator (`sniff`) gestartet. Dabei wird als Parameter der oben erzeugte Textschnüffler übergeben. Der eigentliche Schnüffelprozeß läuft wie im vorigen Kapitel beschrieben ab: Der zu durchsuchende Behälter wird festgelegt (`sniffIn`) und anschließend der Schnüffler gestartet (`runInThread`). Außerdem wird der Schnüffler in

einer Liste aller laufenden Schnüffelprozesse registriert. Der Finder wartet so lange, bis die Anfrage an den Registrator ergibt, daß ein wartender Schnüffler, also einer, der den Suchprozeß beendet hat, vorhanden ist. Dieser wird als aktueller Schnüffler gesetzt, aus der Liste registrierten Schnüffler im Registrator entfernt, und es wird das Ereignis bekanntgegeben, daß das Suchergebnis vorliegt. Dieses Ergebnis, es handelt sich dabei um ein Inhaltsverzeichnis (`dvTableOfContents`), fordert der Finder vom Schnüffler an und zeigt es dann an. In Abbildung 5 ist der Ablauf des Suchvorgangs in einem Interaktionsdiagramm dargestellt.

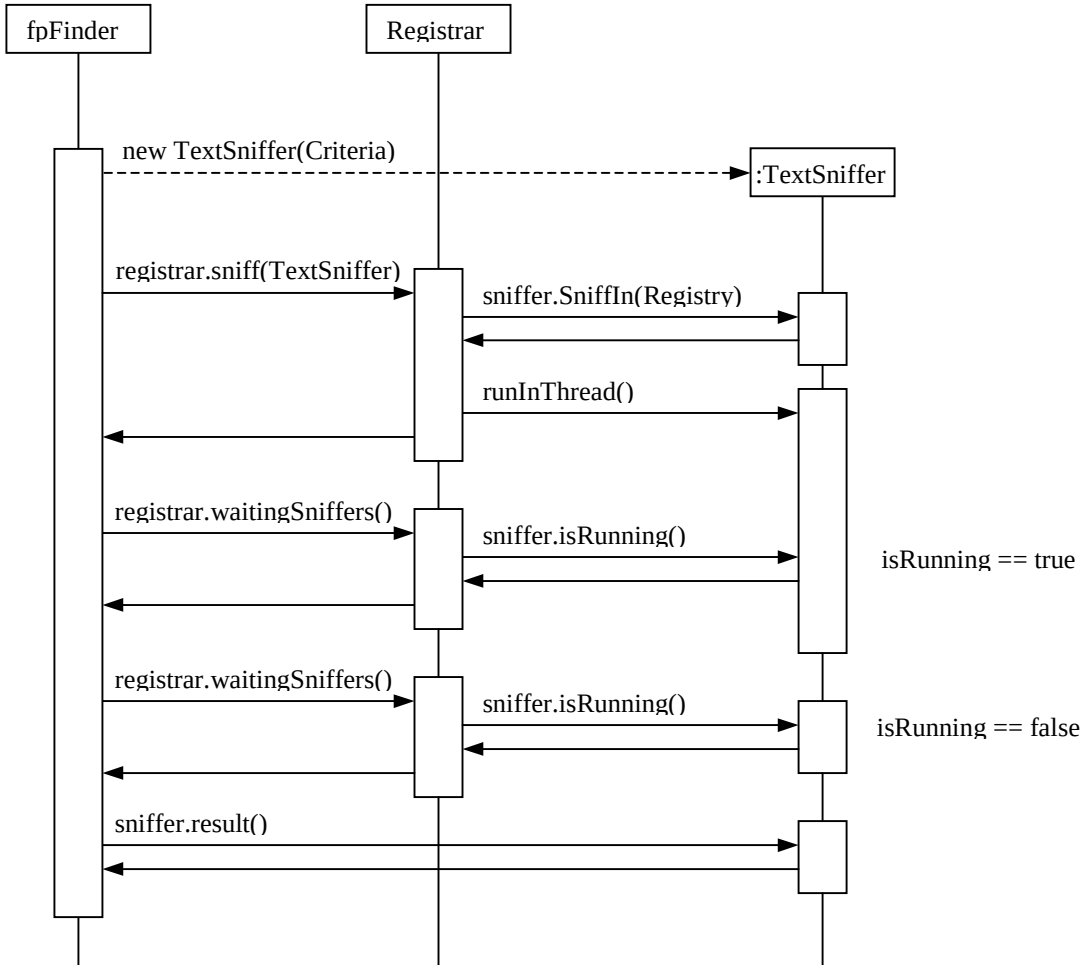


Abbildung 5: Interaktionsdiagramm des Finders

Da der Finder auf dem Text-Schnüffler basiert, besteht auch bei ihm nicht die Möglichkeit, die Suche auf bestimmte Felder zu begrenzen. Zusätzlich besteht die Einschränkung, daß nur eine Volltextsuche unterstützt wird. Daher bekommt man bei den meisten Suchanfragen nur ungenügende Ergebnisse. Sucht man zum Beispiel in einer Adreßkartei alle Kunden aus „Kiel“, so werden auch alle Kunden gefunden, die an einer „Kieler Straße“ oder einem „Kieler Weg“ wohnen. Verstärkt wird dieser Effekt dadurch, daß die Verwaltungsinformationen, die sich in der Registratur befinden (Besitzer, Erzeuger, etc.) auch mit in die Suche einbezogen werden. Dies führt dazu, daß bei Suchanfragen nach kurzen Texten oft alle vorhandenen Materialien vermeintlich die Suchbedingung erfüllen.
Da der Finder als eigenständiges Werkzeug vorliegt, kann er, z.B. über den JWAM-Desktop (siehe [Lippert 99]), direkt auf Behältern wie der Registratur arbeiten. Anpassungen an bestimmte Anwendungen oder Speichermedien sind nicht erforderlich, wenn die Anwendung die Registrator-Schnittstelle und das Speichermedium die Registratur-Schnittstelle benutzen.

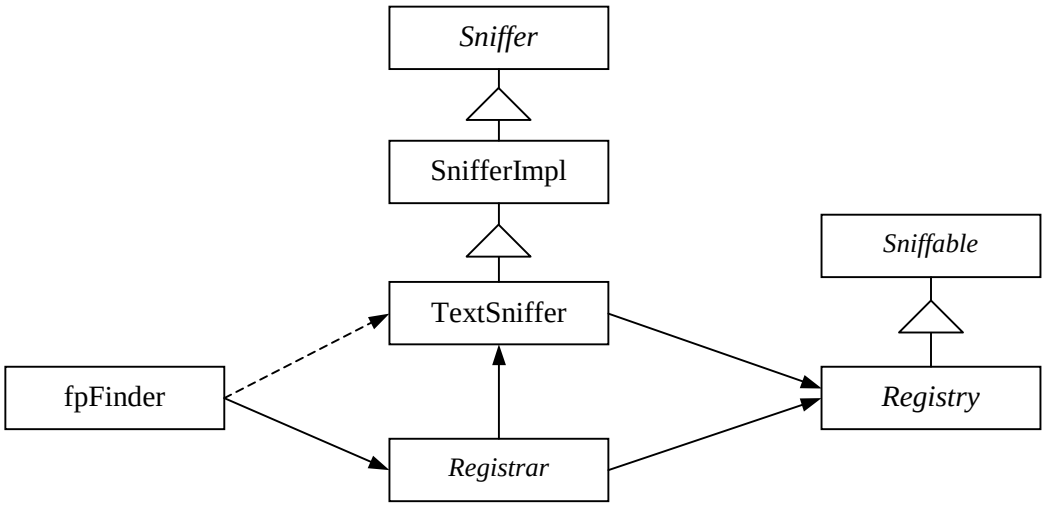
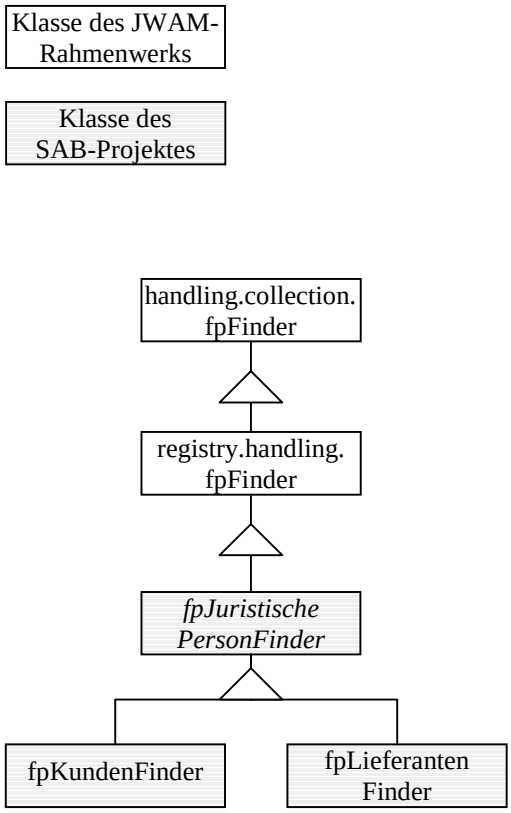


Abbildung 6: Klassendiagramm des Finders

3.2.4. Der Einsatz von Schnüfflern am Beispiel der Saffran-Auftragsbearbeitung

Die Saffran-Auftragsbearbeitung (auch SAB-Projekt genannt) ist eine Software, die aus dem kommerziellen Einsatz von JWAM entstanden ist. Es handelt sich hierbei um eine Kunden- und Auftragsverwaltung eines Werbemittelherstellers. Die Software basiert auf der Alphaversion des JWAM 1.4, benutzt die Relationale-Datenbank-Registratur als Persistenzdienst und setzt JDataStore von Borland als darunterliegende Datenbank ein. Anhand dieser Software soll der praktische Einsatz von Schnüfflern zur Suche in Datenbeständen gezeigt werden.
Das SAB-Projekt besteht aus mehreren, zum Teil unabhängigen Werkzeugen, die den JWAM-Desktop als gemeinsame Benutzungsoberfläche verwenden. Zwei dieser Werkzeuge sind der Kunden- und der Lieferantenverwalter, die sich in ihrer Funktionsweise und ihrem Aufbau nur unwesentlich unterscheiden. Der Kundenverwalter dient dazu, in der Liste aller Kunden, die in der Registratur gespeichert sind, zu suchen. Dies geschieht mit Hilfe eines Kundenfinders, der als Subwerkzeug in den Kundenverwalter integriert ist.
Von den Suchergebnissen kann eines ausgewählt und bearbeitet werden. Hierzu dient der Kundenaktenverwalter, mit dem sich alle Informationen über den jeweiligen Kunden anzeigen und bearbeiten lassen. Teil der Kundenakten sind die Kundenstammblätter, in denen die grundlegenden Daten wie Name, Anschrift, Telefonnummer, etc. gespeichert sind. Entsprechend benutzt der Lieferantenverwalter einen Lieferantenfinder und einen Lieferantenaktenverwalter, der auf Lieferantenakten und Lieferantenstammblättern arbeitet.
Um die Gemeinsamkeiten von Kunden und Lieferanten nicht doppelt implementieren zu müssen, wurde über beide die Abstraktion „juristische Person“ eingeführt. In den Klassen Juristische-Person-Stammblatt, Juristische-Person-Akte und Juristische-Person-Aktenverwalter befinden sich die jeweils identischen Teile der zugehörigen Kunden- bzw. Lieferantenklasse.
Der Juristische-Person-Finder ist eine Verallgemeinerung für den Kunden- und den Lieferantenfinder. Es handelt sich hierbei um eine Spezialisierung des im vorigen Abschnitt beschriebenen allgemeinen Finders. Die zusätzlich in Abbildung 7 dargestellte zweite Finder-Klasse in der Vererbungshierarchie ist beim Wechsel von Version 1.4 zu 1.5 des JWAM-Rahmenwerks weggefallen ist. Der Suchvorgang läuft nach einem ähnlichen Schema ab, wie es im Kapitel über den allgemeinen Finder (Kapitel 3.2.1.) beschrieben ist: Der Finder erzeugt

zunächst einen neuen Schnüffler und ruft mit ihm als Parameter die Suchfunktion des Registrators auf. Der Registrator legt daraufhin das zu durchsuchende Speichermedium fest (die Registratur) und startet den Suchvorgang. Nach dem Ende des Suchvorgangs wird ein Ereignis (Event) ausgelöst, um auf das Suchergebnis wartende Anwendungen zu benachrichtigen. Diese können das Ergebnis am Finder erfragen.



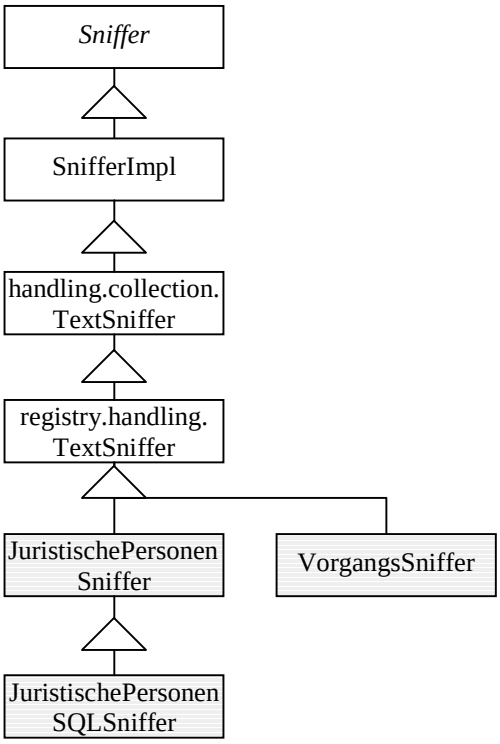


Abbildung 7: Finder-Architektur im SAB-Projekt **Abbildung 8: Sniffer-Architektur im SAB-Projekt**

Die Erzeugung eines neuen Schnüfflers und der eigentliche Suchvorgang sind gegenüber dem Finder verändert. Um einen neuen Schnüffler zu erzeugen, wird in der Version 1.4 des Rahmenwerks die Methode `newSniffer` aufgerufen. Diese Methode wird im spezialisierten Finder (z.B. dem Kundenfinder) durch eine Methode überschrieben, die in seiner Oberklasse (Juristische-Person-Finder) einen geeigneten Schnüffler (`suitableSniffer`) anfordert. Hier wird ein Schnüffler vom Typ juristische Personen-SQL-Schnüffler erzeugt. Es handelt es sich hierbei um einen speziellen Text-Schnüffler (siehe Abbildung 8), der um eine JDataStore spezifische Suchroutine erweitert ist. Diese wird benutzt, wenn es sich bei der eingesetzten Registratur um eine Implementierung für JDataStore handelt (`RegistryJDataStoreImpl`). Ist dies nicht der Fall, wird die Suche mittels der Standardmethode des Text-Schnüfflers (`doRun`) gestartet.
Die JDataStore-spezifische Suche (gestartet mit `doJDataStoreRun`) überprüft zunächst, ob in der Datenbank für den gesuchten Typ (in diesem Fall Kunden- oder Lieferantenakten) eine passende Tabelle vorhanden. Ist dies der Fall, wird dynamisch ein SQL-Ausdruck erzeugt, bei dem es sich um eine Suchanfrage an die Datenbank handelt. Die Feldnamen in diesem SQL-Ausdruck sind statisch, das heißt im Programmcode fest eincodiert, während der Tabellenname an der Registratur erfragt wird und das Vergleichskriterium aus der Suchanfrage stammt.

```
SELECT “Feld1”, “Feld2”, “FeldN” FROM “Tabelle” WHERE
UPPER (“FeldN”) LIKE UPPER (‘%Vergleichskriterium%’)
```

Die kursiv dargestellten Namen werden dynamisch erzeugt, alle anderen sind fest codiert. Der nächste Ausdruck zeigt ein kleines Beispiel:

```
SELECT “object”, “Adresse”, “name” FROM “Kunden” WHERE
UPPER (“name”) LIKE UPPER (‘%mustermann%’)
```

Dieser SQL-Ausdruck wählt aus der Tabelle Kunden alle Objekte, Adressen und Namen aus, bei denen der Name den String „mustermann“ beinhaltet. (Die Unterscheidung von Klein- und Großschreibung wird durch die Verwendung von „`UPPER`“ ignoriert.)
Der SQL-Ausdruck wird in der JDataStore-Registratur ausgeführt (`runSQLQuery`). Die Registratur liefert als Ergebnis eine Menge von Objekten. Zu jedem Objekt wird an diesem die Beschreibung (`ThingDescription`) erfragt und aus allen Beschreibungen ein Inhaltsverzeichnis aufgebaut (`dvTableOfContents`). Dieses Inhaltsverzeichnis ist das Ergebnis der Suchanfrage.
Neben dem Auffinden von Personendaten, dienen im SAB-Projekt Schnüffler zur Suche von Aufträgen. Das in der Vorgangsverwaltung dafür zuständige Werkzeug ist der Positionsfinder. Er benutzt zum Suchen den Vorgangsschnüffler, der wiederum eine Erweiterung des Text-Schnüfflers ist. Der Aufbau ist ähnlich zum juristische Personen-Schnüffler: Es gibt eine gesonderte Routine für die Suche in JDataStore-Registraturen und eine Implementation für alle anderen Suchräume.
Mit dem Einsatz der an die Datenbank JDataStore angepaßten Suchfunktion erfolgt die Suche wesentlich schneller, als bei der Lösung, die auf dem Textschnüffler beruht. Der Grund dafür liegt in der Datenbank, die Suchanfragen im allgemeinen sehr schnell beantworten kann. Die Anfragemöglichkeiten beschränken sich bei beiden Lösungen allerdings weiterhin auf Volltextsuche. Beschränkungen auf bestimmt Felder, in denen gesucht werden soll, sind nicht möglich. Ein Wechsel des Speichermediums ist bei der optimierten Lösung ebenfalls nicht möglich, da die Suchfunktion speziell auf JDataStore angepaßt ist. Die Suchfunktion kann auch in anderen Anwendungen eingesetzt werden, erfordert dann aber Anpassungen an Tabellen- und Objektnamen.

3.3. Suchen mit direkter Persistenzmedienanbindung

Am Beispiel des SAB-Projektes hat sich gezeigt, daß es oft vorteilhaft ist, die Suche direkt auf den Persistenzmedien, hier der Datenbank JDataStore, aufzusetzen, da diese effizientere Anfragemöglichkeiten bieten. Es kann entweder direkt auf der Schnittstelle der Datenbank aufgesetzt werden, oder es können Standardanfragesprachen, die auf Klassen von Datenbanken arbeiten, eingesetzt werden. Direkt auf der Schnittstelle der Datenbank wird man meist entwickeln müssen, wenn man objektorientierte Datenbanksysteme einsetzt. Es gibt zwar die bereits in Kapitel 2.3. erwähnte Anfragesprache OQL, die aber bisher nur wenige Datenbanksysteme unterstützen. Anders ist die Situation bei relationalen Datenbanksystemen: Hier hat sich als Standard die Anfragesprache SQL etabliert, die praktisch von jedem relationalen Datenbanksystem unterstützt wird. Problematisch an SQL ist, daß zwar die Anfragesprache einheitlich ist, die dahinter stehenden Datentypen und deren Definition aber von Datenbank zu Datenbank variieren. Daher müssen beim Wechsel der Datenbank oft Anpassungen vorgenommen werden, obwohl beide mit der gleichen Anfragesprache arbeiten.

Im Beispiel des SAB-Projektes werden die Suchanfragen zwar datenbankunabhängig in SQL formuliert, die eigentliche Anbindung an die Datenbank erfolgt aber über eine JDatastore-spezifische Implementierung der Registratur (RegistryJDataStoreImpl). Bei einem Wechsel der Datenbank müssen die Registratur und der datenbankspezifische Schnüffler komplett neu implementiert werden.
In vielen Projekten ist es nicht möglich, sich im voraus auf ein Persistenzmedium festzulegen. Dies kann notwendig sein, weil die Anwendung auf verschiedenen Persistenzmedien arbeiten soll oder eine Festlegung erst im Laufe des Projektes erfolgt. Dann ist es nur möglich, die in der Registratur-Schnittstelle zur Verfügung gestellten Methoden zu benutzen, um eine Suchfunktion zu realisieren. Die Suche kann in diesem Fall entweder mit Hilfe von Schnüfflern erfolgen (wie bereits beschrieben) oder über eigene Implementierungen, die sich an die Registratur-Schnittstelle halten, gelöst werden. Ein Beispiel, bei dem die Suchfunktionalität komplett in der Anwendung liegt und gleichzeitig die Registratur-Schnittstelle verwendet wird, ist die Suche nach Anfragen im HelpDesk-Projekt. Als Persistenzmedium kommt hier aber auch nur die Dateiregistratur zum Einsatz.

3.3.1. Suche in der Registratur ohne Schnüffler am Beispiel des HelpDesks

Der HelpDesk ist eine Anwendung zur Unterstützung der Bearbeitung und Beantwortung von Hilfeanfragen zu technischen Problemen (vgl. [Otto & Schuler 00]. Entstanden ist er am Arbeitsbereich Softwaretechnik mit dem Ziel, den dortigen Administrator zu entlasten.
Kern des HelpDesks ist das Anfragenwerkzeug (Abbildung 9). Mit ihm werden die Anfragen (`Querys`) verwaltet, es besteht die Möglichkeit Anfragen zu beantworten, weiterzuleiten oder Rückfragen zu stellen. Außerdem bietet das Werkzeug die Möglichkeit, Anfragen gefiltert anzuzeigen. Es können Anfragen angezeigt werden, die zu einer bestimmten Kategorie gehören (z.B. Hardware- oder Softwarefragen), von einem bestimmten Benutzer gestellt wurden oder die einen bestimmten Status haben. Bei dieser gefilterten Auswahl handelt es sich praktisch um eine Suchfunktion, weshalb sie hier kurz beschrieben wird.

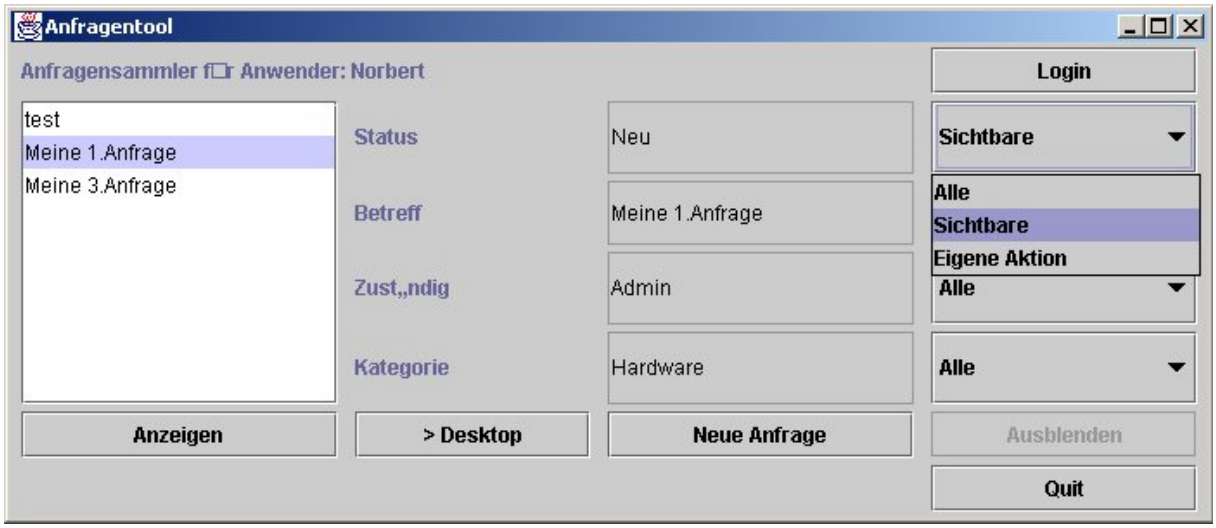


Abbildung 9: Anfragenwerkzeug des HelpDesks

Die Suchfunktion des HelpDesks benutzt zur Suche das Inhaltsverzeichnis (`dvTableOfContents`) aller Anfragen, die in der Registratur gespeichert sind. Um eine Suche durchzuführen, erfragt die Suchfunktion zunächst dieses Inhaltsverzeichnis am Registrator, der es aus den Daten der Registratur aufbaut. Nun wird zu jedem Eintrag im

Inhaltsverzeichnis die zugehörige Anfrage aus der Registratur geladen und mit einem Anfragenfilter (`QueryFilter`) verglichen. Der Anfragenfilter ist identisch aufgebaut wie eine Anfrage, das heißt, er besitzt die gleichen Attribute. Jedes der Attribute des Anfragenfilters wird mit dem zugehörigen Attribut der Anfrage verglichen. Schlägt einer dieser Vergleiche fehl, wird eine Variable „Hinzufügen“ (`add`) falsch gesetzt. Sind alle Vergleiche durchgeführt, wird überprüft, welchen Status die Hinzufügen-Variable besitzt. Ist er wahr (`true`), wird die Anfrage zur Liste der Ergebnisse hinzugefügt, sonst nicht.
Die Suchfunktion des HelpDesks ist, bedingt durch ihren Aufbau, sehr langsam: Zu erst muß das komplette Inhaltsverzeichnis aus der Registratur eingelesen werden. Da die Registratur dieses erst aufbaut, ist hierfür schon ein gewisser Zeitaufwand notwendig. Dann werden für jede Anfrage alle Attribute verglichen. Dies wäre oft gar nicht nötigt, da bereits nach dem ersten negativen Vergleich feststeht, daß die Anfrage nicht zum Suchergebnis gehört.
Durch die Benutzung eines anpaßbaren Filters kann jedes Attribut mit einem dazugehörigen Suchkriterium verglichen werden. Es ist allerdings nur der Vergleich auf exakte Übereinstimmung implementiert. Da der HelpDesk nur über die vom Registrator bereitgestellten Methoden auf die Registratur zugreift, ist ein Wechsel auf ein anderes Speichermedium, soweit es dafür ebenfalls eine Registratur-Schnittstelle gibt, möglich. Die Suchfunktion ist speziell für den HelpDesk geschrieben. Um eine ähnliche Suchfunktion für eine andere Anwendung einzusetzen, müßte diese nahezu komplett neu implementiert werden.

3.4. Vergleich der bisher existierenden Lösungen zur Suche

Keine der vorgestellten Lösungen kann alle Anforderungen erfüllen, die für eine allgemein einsetzbare Suchfunktion notwendig sind. Die Suchmöglichkeiten des Finders und der im SAB-Projekt eingesetzten Lösungen reichen für die meisten Anwendungen nicht aus. Die Suche läßt sich nicht auf bestimmte Attribute eines Objektes einschränken und erlaubt zudem nur eine Volltextsuche. Die Suchfunktion des HelpDesks kann zwar für jedes Attribut ein eigenes Kriterium verwenden, erlaubt dafür aber nur Vergleich auf exakte Übereinstimmung.
Vergleicht man die Geschwindigkeit der Suchfunktionen, sieht man, daß bei großen Datenbeständen nur mit einer Anbindung an eine Datenbank ausreichend schnelle Ergebnisse erzielt werden. Bei vielen kleineren Anwendungen spielt die Suchgeschwindigkeit allerdings keine entscheidende Rolle, da der zu durchsuchende Datenbestand relativ klein ist.
Ein Wechsel auf ein anderes Speichermedium ist nur mit der JDataStore-Lösung nicht möglich. Die anderen Lösungen benutzen die Schnittstelle des Registrators und können somit alle Speichermedien einsetzen, für die eine Implementierung der Registratur vorhanden ist. Während der Finder und der Textschnüffler aus dem SAB-Projekt anwendungsunabhängig sind, müssen beim Einsatz der JDataStore-Lösung Anpassungen an Tabellen- und Objektnamen vorgenommen werden. Ein Einsatz der Suchfunktion des HelpDesks in einer anderen Anwendung würde eine nahezu komplette Neuimplementierung erfordern.
In der folgenden Tabelle werden die Vor- und Nachteile der untersuchten Lösungen noch einmal gegenübergestellt:

	Anfrage-möglichkeiten	Performance	Wechsel des Speichermediums	Wechsel der Anwendung	Implementierungs-Aufwand
Finder	nur Volltextsuche	langsam	möglich	möglich	keiner
SAB mit TextSniffer	nur Volltextsuche	langsam	möglich	möglich	gering
SAB mit doJDataStore-run	nur Volltextsuche	schnell	nicht möglich	nach Anpassungen möglich	hoch
HelpDesk	flexibel	sehr langsam	möglich	nicht möglich	hoch

Abbildung 10: Vergleich der vorgestellten Lösungen zur Suche

4. Existierende Lösungen für Persistenzschnittstellen und darauf aufbauende Suchfunktionen

Im vierten Kapitel dieser Arbeit werden existierende Lösungen zur Anbindung von Persistenzmedien an Anwendungen untersucht, bei denen ein flexibler Austausch des Persistenzmediums möglich ist. Insbesondere wird hierbei auf Suchanfragen eingegangen. Der erste Abschnitt behandelt die Abbildung von Objekten auf Tabellen relationaler Datenbanken im allgemeinen. Hierzu wird im zweiten Abschnitt das Programm „TopLink“ vorgestellt. Der dritte Teil befaßt sich mit dem vielleicht zukünftigen Standard eines Persistenzmodells für Java, JDO.

4.1. Abbildung von Objekten auf relationale Datenbanken

Das Problem des Abbilden (mapping) von Objekten auf Tabellen einer Datenbank tritt dann auf, wenn eine Anwendung nach dem objektorientierten Paradigma entwickelt wird und als Persistenzmedium eine relationale Datenbank verwendet werden soll. Aus entwurfstechnischer Sicht wäre der Einsatz von objektorientierten Datenbanken vorzuziehen, da dann nicht vom verwendeten Paradigma abgewichen werden muß und somit rein objektorientierte Programme entstehen. Da relationale Datenbanken aber wesentlich verbreiteter und ausgereifter sind, werden sie trotzdem häufig eingesetzt. Muß eine neu entwickelte Anwendungen auf eine bestehende Datenbankinfrastruktur zugreifen, scheidet ein Wechsel der Datenbank wegen der hohen Kosten und des Aufwandes für Neuanschaffung und Migration des Datenbestandes auch aus. In vielen Anwendungsfällen (z.B. bei Massendaten mit einfachem Aufbau) bieten objektorientierte Datenbanken nicht die gleiche Leistungsfähigkeit wie relationale, weswegen hier ein Einsatz von objektorientierten Datenbanken nicht möglich ist. Ein weiterer Punkt, der gegen den Einsatz von objektorientierten Datenbanken spricht, ist, daß eine einheitliche, von allen Systemen unterstützte Schnittstelle fehlt. Dagegen wird praktisch jede relationale Datenbank mit einem ODBC-Treiber (ODBC: Open Database Connectivity – Standard, der den einheitlichen Zugriff auf Datenbanken ermöglicht, siehe [Geiger 95]) und mit zunehmender Verbreitung von Java auch mit einem JDBC-Treiber geliefert.
Es ist daher notwendig, Strategien zu entwickeln, um die eigentlich grundlegend verschiedenen Modelle von objektorientierter Programmen und relationalen Datenbanken miteinander zu verbinden. Es sollen dabei weder Änderungen an der Modellierung der Anwendung noch an der Struktur der Datenbank notwendig sein. Um dies zu ermöglichen, muß es ein Abbildungsmechanismus zwischen Anwendung und Datenbank geben. Objekte und die Beziehungen zwischen ihnen (Vererbungen, Referenzen, etc.) müssen auf Tabellen, Spalten und Felder der Datenbank abgebildet werden.
Zunächst muß geklärt werden, wo die Abbildungsfunktionalität angesiedelt ist. Der einfachste Ansatz ist, im Quellcode der Anwendung direkt SQL-Ausdrücke zu verwenden, die über die ODBC- oder JDBC-Schnittstelle an die Datenbank geschickt werden. Hiermit läßt sich relativ schnell und mit wenig Aufwand ein Ergebnis erzielen, solange es sich um kleine

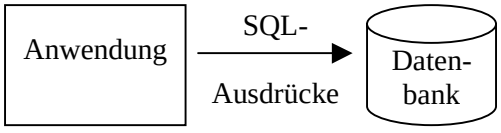


Abbildung 11: Anwendung greift direkt auf Datenbank zu

Anwendungen handelt. Nachteilig an diesem Ansatz ist, daß eine feste Bindung zwischen Programmcode und Datenbank entsteht. Selbst kleine Änderungen in der Datenbank (z.B. Umbenennung oder Erweiterungen) erfordern eine Überarbeitung und anschließende Neuübersetzung der gesamten Anwendung.
Ein besserer Ansatz ist es, die Abbildungslogik zwischen Anwendung und Datenbank in eine oder mehreren „Abbildungsklassen“ zu kapseln. Sollten Änderungen in der Datenbank anstehen, erfordert dies nur noch eine Überarbeitung der Abbildungsklassen. Dieser Ansatz wird im JWAM-Rahmenwerk vom RDBMS-Mapper (siehe Kap. 3.1.) verwendet und kommt auch für die Beispielanwendung „Kundenverwaltung“ (siehe Kap. 5.1) zum Einsatz. Aber auch bei diesem Ansatz zeigt sich, daß die Abbildungsklassen mit steigender Anwendungsgröße unübersichtlich und schwer wartbar werden. Eine vollständige Kapselung der Datenbank besteht weiterhin nicht, und neben der Programmiersprache bleibt eine zweite Anfragesprache mit einem anderen Programmierparadigma bestehen.

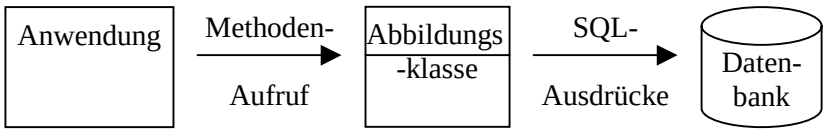


Abbildung 12: Anwendung greift über Abbildungsklasse auf Datenbank zu

Für große Anwendungen besteht ein Lösung darin, eine Persistenzschicht einzuführen, die sich um die Abbildung von Objekten auf das Persistenzmedium kümmert. Diese Persistenzschicht kapselt die Schnittstelle der Datenbank vollständig, so daß es für den Anwender egal ist, ob sich dahinter eine objektorientierte oder eine relationale Datenbank befindet. Sie stellt der Anwendung Funktionen wie leseObjekt, speichereObjekt oder findeObjekte zur Verfügung. Der Anwender greift nur über diese Funktionen auf die Datenbank zu, komplizierte Anfragen in SQL sind damit überflüssig. Die exakte Abbildung von Objekten auf Tabellen wird zum einen durch die generelle Abbildungsstrategie und zum anderen durch eine Art Zuordnungsverzeichnis bestimmt. Durch die generelle Abbildungsstrategie wird festgelegt, wie die Konzepte der objektorientierten Programmierung wie Vererbung, Aggregation und Referenz auf Tabellen abgebildet werden könne. Abbildung 13 zeigt einen Überblick über die verschieden Strategien für die einzelnen Konzepte.

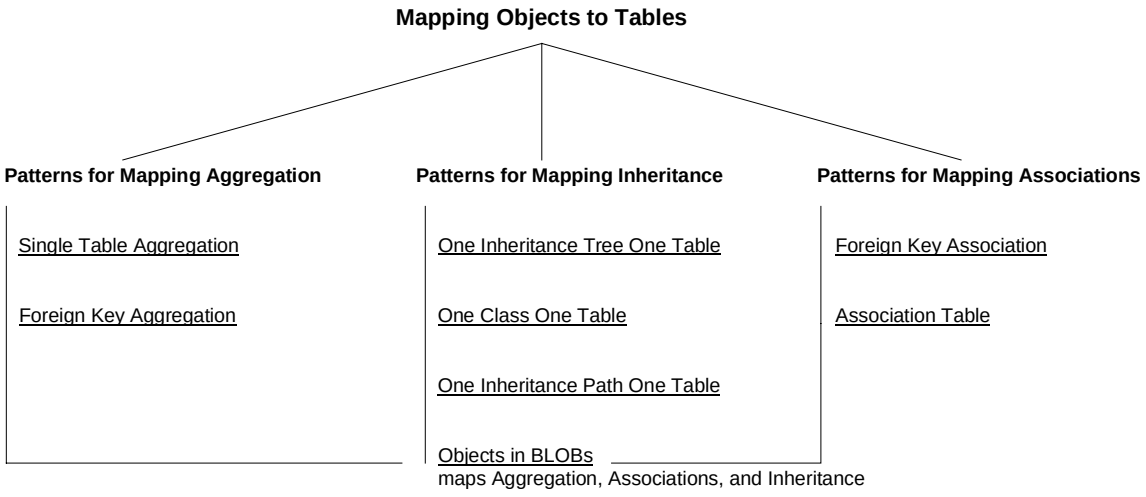


Abbildung 13: Abbildungsstrategien für Konzepte der objektorientierten Programmierung (aus [Keller 97])

Die in Abbildung 13 genannten Abbildungsstrategien werden in [Keller 97] genauer erläutert und bewertet.
In der Beispielanwendung „Kundenverwaltung“ wurde die Lösung gewählt, Objekte als BLOBs (binary large objects) zu speichern, da hiermit der Wechsel zwischen einer relationalen und einer objektorientierten Datenbank relativ einfach zu realisieren ist.
Hat man sich für eine Abbildungsstrategie entschieden, muß die exakte Zuordnung von Objekten auf Tabellen festgelegt werden. Dazu dient ein Verzeichnis, in dem für jede Klasse die zugehörige Tabelle und für jedes Attribut die zugehörige Spalte dieser Tabelle angegeben ist. Die Abbildungsklasse benutzt die Informationen dieses Abbildungsverzeichnisses, um Anwendungsobjekten Datenbanktabellen zuzuordnen. Dieses Verzeichnis ist die einzige Stelle, in der Veränderungen vorgenommen werden müssen, wenn es zu einem Wechsel der Datenbank oder größeren Änderungen in ihrer Struktur kommt. Der Anwendungscode und der Code der Abbildungsklasse bleiben hiervon unberührt.
Da der Aufwand eine eigene Persistenzschicht zu entwickeln groß ist, lohnt es sich nur, hierfür Entwicklungszeit zu investieren, wenn das Ergebnis hinterher für weitere Anwendungen weiterverwendet werden kann. Als Alternative hierzu bietet sich an, ein externe Lösung zu verwenden. Es gibt eine große Anzahl an verschiedenen, zum Teil kommerziellen Produkten, die die Aufgabe einer Persistenzschicht übernehmen (siehe [Ambysoft 00]). Als eines der bekanntesten Produkte soll im folgenden „TopLink“ von der Firma „The Object People“ genauer betrachtet werden und insbesondere auf das Stellen von Suchanfragen mit diesem Programm eingegangen werden.

4.2. TopLink

TopLink ist ein Rahmenwerk, das die im vorigen Abschnitt allgemein beschriebenen Konzepte nutzt, um objektorientierten Anwendungen den Zugriff auf relationale Datenbanken zu erlauben. Es handelt sich hierbei um ein kommerzielles Produkt, das von der Firma „The Object People“ entwickelt wurde und mittlerweile von „WebGain“ vermarktet wird (siehe [WebGain 01]). TopLink ist in drei Varianten erhältlich, eine für den BEA WebLogic Server, eine für Smalltalk und eine für Java. Die Java-Version ist Gegenstand aller weiterer Betrachtungen.

4.2.1. Funktionsweise von TopLink

TopLink stellt für Anwendungen eine Schnittstelle bereit, die über alle Funktionen verfügt, die zum Zugriff auf eine Datenbank notwendig sind. Die Aufrufe dieser Funktionen übersetzt TopLink in SQL-Ausdrücke, die über die JDBC-Schnittstelle an die Datenbank weitergeleitet werden (siehe Abbildung 14).

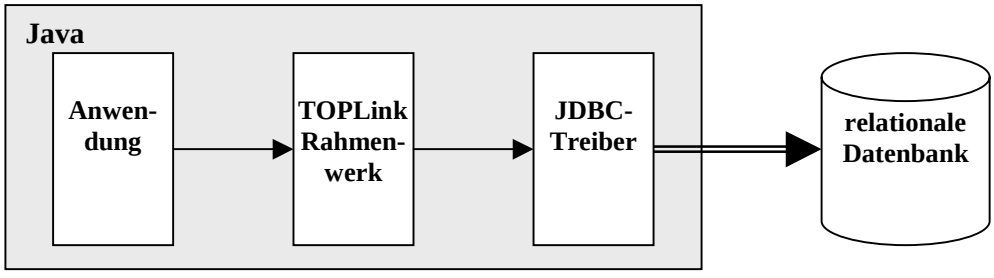


Abbildung 14: Einordnung des TopLink-Rahmenwerks

Für TopLink ist es nicht nötig, daß zu speichernde Klassen von einer gemeinsamen Oberklasse erben, wie es zum Beispiel bei der objektorientierten Datenbank Versant der Fall ist. Stattdessen wird eine Art „Zwischenübersetzung“ verwendet. Das zentrale Werkzeug von TopLink, der TopLinkBuilder, übernimmt bestehenden Java-Programmcode und fügt die Erweiterungen, die für den Zugriff auf eine Datenbank notwendig sind, ein. Das Ergebnis ist wiederum Java-Quellcode, der weiter bearbeitet werden kann. Um Objekte zu speichern, benötigt TopLink Zugriff auf alle Attribute. Daher muß beim JDK 1.1 für jedes Attribut eine öffentliche Zugriffsfunktion vorhanden sein. Ab JDK1.2 läßt sich diese Problem mit Hilfe von Core Reflection umgehen.
Der Ablauf der Entwicklung mit TopLink sieht folgendermaßen aus: Als erstes muß die grundlegende Architektur der Anwendung, also welche Klassen sie besitzt und wie sie zueinander in Beziehung stehen, entwickelt, implementiert und in Java-Programmcode übersetzt werden. Danach muß festgelegt werden, wie die Datenbank aufgebaut ist, also welche Tabellen, Spalten und Felder sie besitzt. Alternativ können hier auch die Informationen aus einem bestehenden Altsystem übernommen werden. Mit dem TopLinkBuilder wird die Klassenstruktur der Anwendung und das Schema der Datenbank importiert. Aus diesen Informationen kann für jede Klasse ihre genaue Abbildung auf Tabellen der Datenbank erstellt werden. Dies geschieht entweder von Hand oder unterstützt durch ein Werkzeug. Die Informationen über die Abbildungen speichert TopLink in „Descriptor“-Dateien, um sie später wieder einlesen zu können. Diese Dateien entsprechen dem in Kapitel 4.1. genannten Zuordnungsverzeichnis. Zusätzlich baut der TopLinkBuilder diese Informationen in den Anwendungscode ein, den er zuvor importiert hat. Der so entstandene neue Programmcode muß als Quellcode wieder exportiert werden. In diesem Quellcode muß der Entwickler die Methoden, die den Zugriff auf die Datenbank übernehmen (Speichern von Objekten, Stellen von Suchanfragen), ausimplementieren. Nach einer weiteren Übersetzung des ergänzten Quellcodes ist die Anwendung lauffähig.

4.2.2. Suchanfragen mit TopLink

TopLink unterscheidet drei grundlegende Arten von Suchanfragen (Querys): Sitzungsanfragen (Session Querys), Anfragenobjekte (Query Objects) und SQL-Anfragen (SQL-Queries). Für alle Suchanfragen gilt, daß TopLink neben allen Exemplaren der gesuchten Klasse auch alle Exemplare von Unterklassen dieser Klasse liefert. Es ist möglich, dieses voreingestellte Verhalten so zu ändern, daß nur Exemplare der gesuchten Klasse gefunden werden. TopLink unterstützt Suchanfragen nach Schnittstellen. Alle Exemplare von Klassen, die eine Schnittstelle implementieren, werden als Ergebnis zurückgeliefert.

Sitzungsanfragen

Die einfachste Art der drei Anfragen sind die Sitzungsanfragen. Ihre Funktionalität wird direkt durch die Datenbanksitzung (DatabaseSession) zur Verfügung gestellt. Jede Anwendung, die mit Hilfe von TopLink auf eine Datenbank zugreifen will, muß ein Exemplar der Datenbanksitzungsklasse erzeugen. Eine Sitzung repräsentiert die Verbindung zwischen Anwendung und Datenbank. In der Datenbanksitzungsklasse werden alle Informationen gespeichert, die für den Zugriff notwendig sind. Dies sind Paßwörter und Konfiguration der Datenbank, die verwendete JDBC-Verbindung und die Descriptorklassen für die Abbildungsinformationen. Jede Datenbanksitzung stellt alle grundlegende Funktionen für das Lesen und Verändern zur Verfügung. Es handelt sich hierbei um Lese-, Schreib-, Einfüg-, Update- und Löschmethoden. Die für das Stellen von Suchanfragen interessanten Methoden sind `readObject()` und `readAllObjects()`. Die `readObject()`-Methode muß mit der

Klasse des zu lesenden Objekts aufgerufen werden. Eine einfache Leseoperation sieht folgendermaßen aus:

```
session.readObject(Customer.class);
```

Dieser Aufruf liefert das erste Exemplar der Klasse Customer, das sich in er Tabelle befindet, in der Customer-Objekte gespeichert sind. Die Methode `readAllObjects()` ließt alle Objekte einer Klasse und liefert als Ergebnis einen Vector von Objekten.
Um nach Objekten suchen zu können, die ein bestimmtes Kriterium erfüllen, wird die Lesemethode um Ausdrücke (Expressions) erweitert. Ein Ausdruck kann alles von einer Konstanten bis zu einer komplexen boolschen Klausel sein. Typischerweise besteht ein Ausdruck aus einem Attribut (z.B. einer in der Datenbank abgebildeten Klasse), einem Operator (gleich, größer, ähnlich, etc.) und einer Vergleichskonstanten. Ausdrücke müssen von der zu ihr gehörigen Fabrikklasse (ExpressionBuilder) erzeugt werden.
Ein Anfrage, mit der alle Kunden gesucht werden, deren Umsatz größer als 10000 DM ist, hat folgenden Aufbau:

```
ExpressionBuilder custBuilder = new ExpressionBuilder();
Vector importantCustomers = session.readAllObjects(Customer.class,
    custBuilder.get(“turnover”).greaterThan(10000));
```

Einfache Anfragen können durch Kombination mit den boolschen Operatoren AND, OR und NOT zu komplexeren Anfragen zusammengesetzt werden. Folgender Code sucht von allen Kunden, deren Namen mit A anfängt und deren Umsatz größer als 10000 DM ist, die Telefonnummer:

```
ExpressionBuilder custBuilder = new ExpressionBuilder();
Expression customerWithA, importantCustomer, complexExpression;
cutomerWithA = custBuilder.get(“name”).like(“A%”);
importantCustomer = custBuilder.get(“turnover”).greaterThan(10000);
complexExpression = customerWithA.and(importantCustomer);
Vector importantNumbers = session.readAllObjects(Customer.class,
complexExpression);
```

TopLink unterstützt standardisierte Datenbankfunktionen wie zum Beispiel `like(String)`, `toUpperCase()`, `currentDate()`, etc.. Über die Methode `getFunction()` können eigene Funktionen aus der Datenbank aufgerufen werden. Abfragen über mehrer Tabellen, die in der Anfragesprache SQL durch „Joins“ gelöst werden, sind ebenfalls möglich:

```
customer.get(“address”).get(“city”).equal(“Hamburg”);
```

Neben den genannten Anfragefunktionen gibt es noch viele weitere, die ihrem Äquivalent aus SQL entsprechen (parametrisierte Anfragen, etc.).

Anfragenobjekte

Normalerweise werden Suchanfragen, wie oben beschrieben, mit Hilfe von Sitzungsmethoden wie `readObject()` durchgeführt. Intern erzeugen diese Methoden Anfrageobjekte, initialisieren sie und benutzen sie, um auf die Datenbank zuzugreifen. Anfrageobjekte sind die Java-Abstraktion von SQL-Aufrufen. Anfrageobjekte erlauben weitreichendere Anpassungen, als es mit Sitzungsanfragen möglich sind. So ist es zum Beispiel möglich, Anfragen mit Hilfe des Cursor-Prinzips auszuführen oder mehrer Anfragen im Batchbetrieb an die Datenbank zu

stellen, um die Zugriffshäufigkeit auf die Datenbank zu verringern. Anfragenobjekten kann ein Name zugeordnet werden, so daß sie später wieder benutzt werden können.
Die Initialisierung und die Ausführung eines Anfrageobjektes ist allerdings aufwendiger als eine Anfrage innerhalb einer Sitzung. Folgendes Beispiel liefert alle Kunden mit Umsatz größer als 10000 DM.

```
ReadAllQuery query = new ReadAllQuery();
query.setReferenceClass(Customer.class);
query.setSelectionCriteria
     (new ExpressionBuilder.get(“turnover”).greaterThan(10000));
Vector customers = (Vector) session.executeQuery(query);
```

Die zu suchende Klasse und das Suchkriterium müssen bei Objektanfragen explizit zugewiesen werden. Danach kann der Aufruf zum Ausführen der Anfrage erfolgen.
Im folgenden Beispiel wird einer Anfrage ein Name zugewiesen, so daß sie parametrisiert aufgerufen werden kann.

Der Ausdruck, der das Suchkriterium für die Anfrage ist, wird erzeugt:

```
ExpressionBuilder custBuilder = new ExpressionBuilder();
nameExpression =
   custBuilder.get(“name”).equal(cust.getParameter(“name”);
```

Das neue Anfragenobjekt wird erzeugt und parametrisiert. Dazu wird am Anfragenobjekt eingestellt, nach Objekten welcher Klasse gesucht werden soll:

```
ReadObjectQuery query = new ReadObjectQuery();
query.setReferenceClass(Customer.class);
query.setSelectionCriteria(nameExpression);
```

Ein Parameter wird der Anfrage hinzugefügt, für den beim Aufruf ein String eingesetzt werden muß:

```
query.setArgument(“name”);
```

Die Anfrage wir zur Sitzung hinzugefügt:

```
session.addQuery(“getName”, query);
```

Jetzt kann die Anfrage mit einem Text für den Parameter (hier „Mustermann“) aufgerufen werden:

```
Customer customer = (Customer) session.executeQuery(“getName”,
     “Mustermann”);
```

SQL-Anfragen

Die dritte Form der von TopLink unterstützen Anfragen sind SQL-Anfragen. In jeder Anfrage kann statt eines Ausdrucks ein SQL-Aufruf stehen. In diesem Fall muß der SQL-Aufruf alle Daten liefern, die für den Aufbau eines Exemplars der zu suchenden Klasse notwendig sind.

```
Customer customer = (Customer) session.readObjekt
 (Customer.class,  “SELECT * FROM CUSTOMER WHERE TURNOVER > 10000”);
```

Schließlich können auch Anfragen gestellt werden, die nicht auf Objektebene, sondern auf Datenebene arbeiten.

```
Vector rows = session.executeSQL(“SELECT NAME FROM CUSTOMER”);
```

Diese Anfrage liefert einen Vektor von aller Kundennamen, ohne das eigentliche Kundenobjekt einzulesen.

4.2.3. Bewertung

TopLink bietet einen großen Funktionsumfang, mit dem es möglich ist, praktisch jede Art von Anwendungszugriff auf Datenbanken abzubilden. Es ist mit wenig Aufwand möglich, die Verbindung zwischen Datenbank und Anwendung herzustellen. Dazu muß allerdings ein Grundwissen über die verschiedenen Abbildungsarten vorhanden sein.
TopLink bietet drei verschiedene Arten, Suchanfragen zu stellen: Für einfache Anfragen dienen Sitzungsanfragen, für komplexere Anfragevorgänge werden Anfrageobjekte verwendet und als Ausweichlösung besteht die Möglichkeit, Anfragen in SQL zu formulieren. Durch diese drei Arten von Anfragen ist es möglich, sich die Art der Anfrage auszuwählen, die für die eigene Anwendung am besten geeignet ist. Damit verbunden ist aber auch ein großer Einarbeitungsaufwand, da jede einzelne der drei Anfragearten bereits recht komplex ist. Auf SQL-Anfragen sollte möglichst verzichtet werden, wenn auf rein objektorientierte Programmierung Wert gelegt wird.
Allerdings bauen alle drei Abfragearten auf den gleichen Voraussetzungen aus: Es muß in der Anwendung das Wissen vorhanden sein, wie und worauf die einzelnen Objektattribute abgebildet werden. Dies widerspricht dem Prinzip der Kapselung der inneren Repräsentation in der objektorientierten Programmierung.
Weitere Einschränkungen von TopLink liegen im technischen Bereich: TopLink arbeitet nur mit relationalen Datenbanken zusammen, es ist nicht möglich, objektorientierte Datenbanken einzusetzen. TopLink ist abhängig von den JDBC-Treibern, die mit den eingesetzten Datenbanken geliefert werden. Sind diese fehlerhaft oder haben nicht den Funktionsumfang, den TopLink benötigt, kann die dazugehörige Datenbank nicht eingesetzt werden. Beispielsweise war es nicht möglich, mit TopLink auf der Datenbank JDataStore zu arbeiten, weil dessen JDBC-Treiber eine Funktion zum Einlesen der Tabellen fehlt.
TopLink ist mit einem Preis von 4000 Dollar für eine Entwicklerlizenz nur für professionelle Anwender interessant.

4.3. Java Data Objects (JDO)

JDO ist eine Architektur, die eine Schnittstelle zum Zugriff auf Persistenzmedien für Java-Anwendungen beschreibt. Die Anzahl der Persistenzmedien, die mit JDO zusammenarbeiten sollen, ist, im Gegensatz zu TopLink, wesentlich größer. Neben relationalen sollen auch objektorientierte oder hierarchische Datenbanken und Dateisysteme von JDO unterstützt werden.
Jede Anwendung soll auf jedem Persistenzmedium arbeiten können, wenn beide die im JDO-Standard vorgegebenen Schnittstellen erfüllen. Neben der Definition von einheitlichen Schnittstellen hat JDO das Ziel, das persistente Abspeichern von Objekten transparent zu machen. Objekte sollen nicht mehr mittels expliziter Anweisungen (`read-/ write Object()`) gelesen oder gespeichert werden. Die JDO-Implementierung soll erkennen, wenn

ein Objekt benötigt wird und es dann automatisch in den Speicher laden. Einsetzbar soll JDO in einem weiten Bereich von Anwendungen sein: Von eingebetteten Systemen bis zu Firmen-Informations-Systemen (Enterprise Information Systems) reicht das geplante Einsatzspektrum. Zu bemerken ist, daß es sich bei JDO erst um den Entwurf eines geplanten Standards handelt. Fraglich ist noch, wie viele Hersteller wirklich JDO unterstützen werden und eine JDO-Schnittstelle in ihre Produkte einbauen.

4.3.1. Warum JDO?

Bisher existiert keine standardisierte Lösung, mit der es unter Java möglich ist, Objekte in Persistenzmedien zu speichern. Stattdessen muß der Entwickler einer Anwendung mit proprietären Schnittstellen arbeiten, die ein Austausch des Persistenzmediums unmöglich machen, etwa bei objektorientierten Datenbanken oder bei Programmen wie TopLink. Als Alternative bleibt nur, sich selbst um die Persistenz zu kümmern. Wenn keine Zeit für den Aufbau einer eigenen anwendungsunabhängigen Persistenzschicht ist, bietet es sich an die zu speichernden Objekte auf Tabellen abzubilden, um sie in einer relationalen Datenbank zu speichern. Zur Anbindung von Java-Programmen an relationale Datenbanken wird die JDBC-Schnittstelle verwendet, die zwar eine einheitliche Anfragesprache, aber kein einheitliches Typsystem definiert. Daher kann es auch bei Verwendung von JDBC zu Inkompatibilitäten kommen. Die Abbildung von Objekten, besonders wenn es sich um komplizierter aufgebaute handelt, ist umständlich und nicht besonders effektiv. Außerdem liegt ein Bruch in der Entwicklung vor, da in zwei Sprachen gleichzeitig, Java und einer Anfragesprache (meist SQL), entwickelt wird.
Um einen neuen Java-Standard zu schaffen, der für die genannten Probleme eine Lösung bietet, wurde im August 1999 eine Expertengruppe gegründet, die in einem „Java Community Process“ eine Entwurf für die Spezifikation von JDO erarbeitet hat. (Beim Java Community Process (JCP) handelt es sich um einen formalisierten, offen Prozeß, mit dem Sun Microsystems Java Technologie-Spezifikationen zusammen mit der internationalen Java-Gemeinschaft weiterentwickelt.) Dieser Entwurf wurde nach mehreren internen Revisionen im Juli 2000 als JDO Draft 0.8 veröffentlicht. Für das vierte Quartal 2000 war eine Referenzimplementierung und eine Testumgebung geplant, die bis jetzt aber noch nicht veröffentlicht wurden.

4.3.2. Architektur von JDO

JDO ist für ein sehr breites Spektrum von Anwendungen vorgesehen: Von Minimalsystemen, die als Persistenzmedium nur ein einfaches Dateisystem besitzen, bis zur Firmen-Informations-Systemen, die ihren Datenbestand verteilt auf verschieden Systemen verwalten, soll JDO einsetzbar sein. Um diesen Anforderungen gerecht zu werden, wird bei JDO nach verwalteten (managed) und nicht verwalteten (unmanaged) Umgebungen unterschieden. Verwaltete Umgebungen sind Umgebungen, bei denen sich das Persistenzmedium selbst oder ein vorgeschaltetes Programm um die Verwaltung der zu speichernden Daten kümmert. In nicht verwalteten Umgebungen kapselt JDO das Persistenzmedium komplett und kümmert sich um Lesen, Speichern und Abbilden der Daten. Der Anwendung werden Daten nur in Form von Java-Klassen zur Verfügung gestellt. In verwalteten Umgebungen bietet JDO zusätzlich die Möglichkeit, der Anwendung transparenten Zugriff auf das Persistenzmedium auf Systemebene zu gewähren, um die Verwaltung der Daten zum Beispiel einer Middleware

zu überlassen. Die weiteren Ausführungen beschränken sich auf den einfachen Fall einer nicht verwalteten Umgebung.
Für den einfachen Fall einer Zwei-Schichten-Architektur stellt JDO zwei grundlegende Schnittstellen zu Verfügung. Auf der einen Seite steht die JDO-Implementierung. Sie erlaubt den Zugriff auf das Persistenzmedium. Die JDO-Implementierung kann entweder ein fester Bestandteil des Persistenzmediums sein oder ein Programm, das JDO-Aufrufe in Aufrufe des Persistenzmediums übersetzt. Jede JDO-Implementierung muß die Schnittstelle `PersistenceManager` implementieren. Sie stellt die Schnittstellen zu Verfügung, die für Anfragen (Query) und Transaktionen (Transaction) notwendig sind.

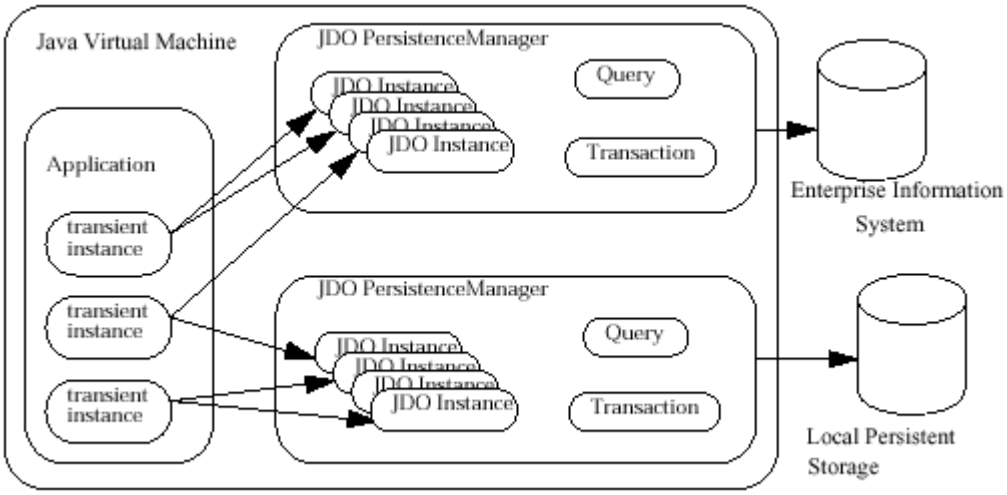


Abbildung 15: nicht-verwaltete JDO-Umgebung (aus [JDO 00])

Auf der anderen Seite stehen JDO-Exemplare (JDO-Instances). Ein JDO-Exemplar ist ein Java-Objekt, das ein oder mehrere Daten, die in einem Persistenzmedium gespeichert sind, repräsentiert. Es kann sich hierbei um ein einzelnes Objekt einer objektorientierten Datenbank, um eine Zeile aus einer relationalen Datenbank oder auch das Ergebnis einer Anfrage, bestehend aus mehreren Zeilen einer Datenbank, handeln. JDO-Exemplare müssen die Schnittstelle `PersistenceCapable` implementieren. Dies kann entweder explizit durch den Entwickler der Klasse geschehen oder implizit mit Hilfe des JDO-Enhancers. Der JDO-Enhancer ist ein Programm, das den Bytecode von Java-Klassen so anpaßt, daß sie von JDO-Implementierungen verarbeitet werden können. Bis auf Systemklassen wie File, Thread, Socket oder System können alle Klassen mit Hilfe von JDO persistent gemacht werden.

4.3.3. Anfragen mit JDO

Das persistente Objektmodell von JDO, das im Rahmen dieser Studienarbeit nicht weiter ausgeführt ist (siehe [JDO 00]), erlaubt das dynamische Nachladen von Objekten wenn Referenzen von anderen Objekten bestehen. Allerdings benötigt man hierfür ein Ausgangsobjekt. Um Objekte von einem Persistenzmedium zu bekommen, bietet JDO drei Möglichkeiten. Besitzt man die gültige ObjectId, ist es möglich, das gesuchte Objekt mit der `getObjectById()`-Methode des Persistenzmanagers zu bekommen. Die zweite Möglichkeit ist die Methode `getExtent()`, die ebenfalls in der Schnittstelle des Persistenzmanagers definiert ist. Sie liefert alle Exemplare einer angegebenen Klasse (optional auch aller Unterklassen).
Die dritte Möglichkeit besteht in der Nutzung der JDO-Query-Schnittstelle. Sie wurde mit dem Ziel entworfen, unabhängig von der darunterliegenden Anfragesprache der Datenbank,

egal ob SQL, OQL oder sonstigen, zu sein. Gleichzeitig soll die Möglichkeit bestehen, Optimierung der Datenbankanfragesprache zu nutzen. Fabriken für Objekte, die die Query-Schnittstelle implementieren, sind alle Objekte, die `PersistenceManager` implementieren.
Eine Anfrage besteht aus mindestens drei Elemente. Das erste ist die Klasse des Suchergebnisses. Alle zurückgelieferten Objekte sind entweder von dieser oder einer Unterklasse. Das zweite Element ist der Behälter, der durchsucht werden soll. Es muß sich hierbei um eine Klasse handeln, die die Schnittstelle `java.util.collection` implementiert. Das dritte Element ist der Anfragenfilter (query filter). Es handelt sich hierbei um einen boolschen Ausdruck in Java, der darüber entscheidet, ob eine Objekt die Suchanfrage erfüllt oder nicht. Übergeben wird dieser boolsche Ausdruck als String. Ein Objekt erfüllt die Suchanfrage, wenn der boolsche Ausdruck im zugehörigen Filter `true` ergibt. Zugelassen sind logische, arithmetische und vergleichende Operatoren. Neben den genannten drei gibt es noch weitere optionale Elemente in einer Anfrage: Es ist möglich, Parameter zu spezifizieren, an die bei der Ausführung der Anfrage Werte gebunden werden, es besteht die Möglichkeit, ungebundene Variablen zu spezifizieren, die beispielsweise in Filtern eingesetzt werden und man kann festzulegen, in welcher Ordnung das Ergebnis zurückgeliefert wird.

Folgendes Beispiel zeigt eine Anfrage nach allen Kunden mit über 10000 DM Umsatz.

```
Collection clnCustomer =
     persistenceManager.getExtent(customerClass);
String filter =  “turnover > 10000”;
Query q = persistenceManager.newQuery
     (Customer.class, clnCustomer, filter);
Collection goodCustomers = q.execute();
```

Das zweite Beispiel zeigt eine Anfrage, die alle Kunden liefert, die aus einer als Parameter übergebenen Stadt kommen.

```
Collection clnCustomer = persienceManager.getExtent (Customer);
String filter = “Customer.city == paraCity”;
String parameter = “String paraCity”;
Query q = persistenceManager.newQuery
     (Customer.class, clnCustomer, filter);
q.declareParameters (parameter);

String myCity = “Hamburg”;
Collection customersOfHamburg = (Collection) q.execute (myCity);
```

4.3.4. Bewertung

JDO ist ein sehr umfangreiches Konzept, das nahezu alle Bereiche, die bei der Anbindung von Anwendungen an Persistenzmedien relevant sind, abdeckt. Das Anfragenkonzept ermöglicht alle typischen Standardanfragen und läßt sich darüber hinaus auch für parametrisierte Anfragen einsetzen. Damit dürfte es die Anforderungen von normalen Java-Anwendungen erfüllen. Im Vergleich zu TopLink ist es komplizierter, eine Anfrage zu formulieren und auszuführen. Dafür kommt JDO aber mit nur einer Art von Anfragen aus. Unschön ist, daß Filter oder auch Parameter als reine Strings definiert sind. Ein jeweils eigener Typ, wie bei TopLink, würde mehr Sicherheit vor Fehlern bieten.

Anwendungsklassen können ohne manuelle Veränderungen am Quellcode an JDO angepaßt werden. Hierfür steht der Referenz-Enhancer zur Verfügung, der Bestandteil der Referenzimplementierung von JDO sein soll.
Die Implementierung der JDO-Schnittstelle auf Datenbankseite ist dagegen komplizierter, da hier die in [JDO 00] vorgestellten umfangreichen Konzepte für die jeweilige Datenbank entwickelt werden müssen. Hier ist man auf die Datenbankhersteller angewiesen, von denen zwar viele JDO unterstützen wollen, aber bisher keiner ein Produkt auf den Markt gebracht hat. Die einzige Datenbank, die bisher JDO unterstützt, ist eine nichtöffentliche Betaversion von „Judo“ der Firma Versant (siehe [Versant 00]).
Wenn die Entwicklung von JDO, die jetzt schon hinter dem angekündigten Zeitplan liegt, abgeschlossen ist und gleichzeitig eine größere Anzahl Hersteller ihre Produkte mit JDO-Unterstützung versieht, könnte JDO der zukünftige Standard zum Zugriff auf Persistenzmedien unter Java werden. Dies wäre ein wichtiger Schritt, um das Einsatzfeld von Java zu erweitern, ohne auf proprietäre Software, wie zum Beispiel TopLink, setzen zu müssen.

5. Vergleich verschiedener realisierter Lösungen

In diesem Kapitel werden verschiedene Lösungsansätze vorgestellt, die während der Implementierungsphase dieser Studienarbeit realisiert wurden. Zunächst wird die Beispielanwendung „Kundenverwaltung“ vorgestellt, mit der die darauf folgenden drei Lösungen zur Suche getestet wurden. Die erste Lösung benutzt das Inhaltsverzeichnis der Registratur, um darauf die Suche durchzuführen. Die zweite Lösung ergänzt die Registratur um eine neue, erweiterte Suchfunktion. Die dritte Lösung integriert diese erweiterte Suchfunktion in einen Schnüffler. Abschließend folgt eine Bewertung der drei Lösungen.

5.1. Beispielanwendung Kundenverwaltung

Um die erweiterte Suchfunktion anhand eines praktischen Beispiels testen zu können, wurde im Rahmen der Arbeit als Beispielanwendung eine kleine Kundenverwaltung entwickelt (Abbildung 16). Sie erlaubt es, Daten wie Name, Adresse und Umsatz zu verwalten. Zur Verfügung stehen Funktionen zum Anlegen neuer Kunden, zum Ändern der Daten eines bestehenden Kunden, zum Löschen von Kunden und zum Suchen von Kunden. Die Suche erlaubt es, mehrere Suchkriterien gleichzeitig zu verwenden. Das Ergebnis besteht aus den Kunden, die alle Kriterien erfüllen. Die Anzahl der Kriterien ist zunächst auf drei beschränkt. Es kann nur nach Namen, Orten und Umsätzen gesucht werden. Bei der Suche nach Texten,

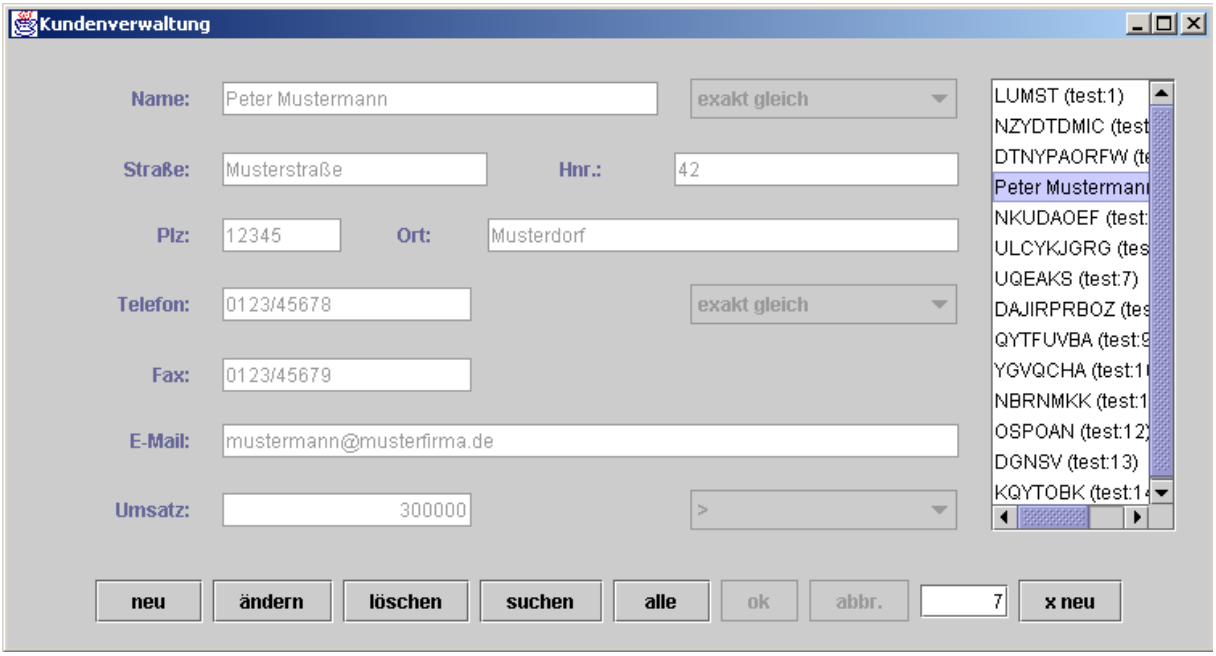


Abbildung 16: Kundenverwaltung

in diesem Fall Name und Ort, kann bestimmt werden, ob eine exakte Übereinstimmung, eine Übereinstimmung des Wortanfangs oder eine Übereinstimmung innerhalb des Wortes vorliegen soll. Bei des Suche nach numerischen Werten, hier Umsatz, kann der vergleichende Operator festgelegt werden (<, <=, =, >= , >). Außerdem gibt es eine Funktion, mit der eine Anzahl vorher festgelegter Datensätze, bestehend aus zufälligen Zeichenketten bzw. Zahlen, erzeugt werden kann. Diese Funktion dient dazu, die Kundenverwaltung mit einer großen Anzahl von Datensätzen zu befüllen, um die Performance der Suchfunktion bei größeren Datenbeständen zu testen. In einer Liste werden die Name aller vorhandenen Kunden

angezeigt. Das Markieren eines Kunden erlaubt es, diesen zu editieren oder zu löschen. Wurde eine Suche durchgeführt, werden in der oben genannten Liste alle Namen der Kunden angezeigt, die das Suchkriterium oder die Suchkriterien erfüllen.
Um die Kundendatensätze persistent zu machen, benutzt die Kundenverwaltung die Registrator-Schnittstelle der Registratur. Damit ist es prinzipiell möglich, jedes Persistenzmedium anzuschließen, für das es eine Implementierung der Registratur gibt. In diesem Fall wurde die Funktion mit der Dateiregistratur (RegistryFileImpl) und den beiden Datenbanken „JDataStore“ und „Cloudscape“, die über die Registratur für relationale Datenbanken (RegistryRDBMSImpl) angeschlossen sind, getestet.

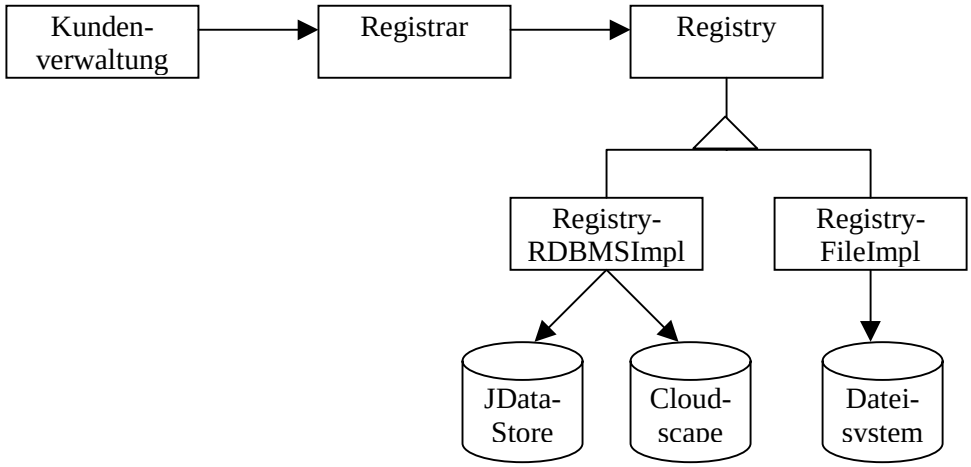


Abbildung 17: Architektur der Kundenverwaltung

5.2. Beschreibung der verschiedenen Lösungen für die Suche

Das Ziel der Entwicklung ist es, für die JWAM-Registratur eine universelle Suchfunktion zu schaffen, die unabhängig von dem eingesetzten Persistenzmedium funktionieren soll. Sie soll das gleiche Ergebnis liefern, egal ob die Dateiregistratur oder eine Datenbank als Persistenzmedium verwendet werden. Gleichzeitig muß aber die Möglichkeit bestehen, die Vorteile eines schnelleren Mediums, wie einer Datenbank, zu nutzen. Die für dieses Ziel entwickelten Konzepte werden im folgenden anhand der Beispielanwendung „Kundenverwaltung“ vorgestellt.

5.2.1. Suche auf dem Inhaltsverzeichnis der Registratur

Um zunächst einmal einen ersten Prototypen für die Suche in der Kundenverwaltung zu bekommen, wurde eine Suchfunktion entwickelt, die auf der bereits bestehenden Schnittstelle der Registratur aufbaut. Die Funktionskomponente der Kundenverwaltung wird um eine Suchmethode (`search()`) erweitert, in der die komplette Suchfunktionalität liegt. Die Kriterien, nach denen gesucht werden soll, werden als Parameter mit übergeben. Ein Suchvorgang läuft folgendermaßen ab: Das komplette Inhaltsverzeichnis (`dvTableOf Contents`), das sämtliche für die Suche relevante Informationen beinhaltet, wird aus der Registratur eingelesen. Von jedem Eintrag wird die Beschreibung (`dvThingDescription`) darauf überprüft, ob Suchkriterien und Beschreibungen der einzelnen Elemente über-

einstimmen. Im positiven Fall wird dieses Element zu der Liste der Suchergebnisse hinzugefügt. Diese Liste wird in Form eines neuen Inhaltsverzeichnisses als Ergebnis zurückgeliefert.
Diese Lösung hat den Nachteil, daß alle Daten, nach denen gesucht wird, im Inhaltsverzeichnis enthalten sein müssen. Somit werden immer alle Datensätze in den Speicher eingelesen, um dort mit den Suchkriterien verglichen zu werden. Ist das Inhaltsverzeichnis größer als der physikalische Speicher im Rechner, wird der Suchvorgang aufgrund von Auslagerungsvorgängen des Betriebssystems extrem langsam oder scheitert ganz. Das verwendete Persistenzmedium wird dagegen nur dazu benutzt, das Inhaltsverzeichnis zu liefern. Die Suche findet nur auf den Informationen dieses Inhaltsverzeichnis, nicht aber auf den eigentlichen Daten statt. Ein weiteres Problem besteht darin, daß die gesamte Suchfunktionalität in der Funktionskomponente liegt. Daher muß für jede Anwendung dieser Teil neu entwickelt werden, was eigentlich vermieden werden sollte.
Insgesamt ist diese Lösung nur für kleine Anwendungen, die die Dateiregistratur verwenden, praktikabel, da es hierbei nur die Möglichkeit der Suche auf dem Inhaltsverzeichnis gibt. Setzt man eine Datenbank ein, ist diese Lösung unsinnig, da die Datenbank als solche gar nicht verwendet wird, sondern weiterhin nur die Informationen aus dem Inhaltsverzeichnis ausgewertet werden.

5.2.2. Erweiterung der Registratur um eine eigene Suchfunktion

Der zweite Lösungsansatz verlagert die Suchfunktionalität von der Funktionskomponente der Anwendung in die Registratur. Die Schnittstelle der Registratur wird um eine Suchmethode erweitert, die in der Implementierung der Registratur auf das verwendete Persistenzmedium abgestimmt werden kann. Suchanfragen werden von Anwendungen an den Registrator gestellt, dessen Schnittstelle ebenfalls um eine Suchmethode erweitert wird. Diese Anfragen werden an die Registratur weitergeleitet.

Aufbau von Suchanfragen

Die Signatur der Suchmethode des Registrators sieht folgendermaßen aus:

```
public dvTableOfContents search
     (Class materialClass, Criteria[] criteria);
```

Die Suchmethode wird mit der Klasse zu suchenden Ergebnisse (`materialClass`) und einem Feld von Suchkriterien (`Criteria`) aufgerufen. Die Klasse des Materials wird bei der Suche auf relationalen Datenbanken benötigt, um eine passende Abbildungsklasse (einen Mapper) auszuwählen.
Jedes Kriterium beinhaltet Informationen über ein Attribut, das relevant für die Suche ist. Es besteht aus dem Namen des zu suchenden Attributs (repräsentiert durch einen String), dem Vergleichstext oder –wert (ein Fachwert) und dem Vergleichsoperator (eine Zeichenkonstante). Implementiert sind für Vergleichstexte ein exakte Übereinstimmung (`EQUALS` ≅ 1), ein gleicher Anfang (`BEGINS_WITH` ≅ 2) und eine Übereinstimmung an beliebiger Stelle (`CONTAINS` ≅ 3). Für numerische Werte sind die Operatoren größer (4), größer oder gleich (5), gleich (6), kleiner oder gleich (8) und kleiner (7) implementiert. Zusätzlich gibt es in der Schnüffler-Lösung einen allgemeinen Operator, der benutzt werden kann, um Eigenschaften an Objekten zu erfragen (`EVALUATES_TO` ≅ 9 – siehe Kapitel 5.2.3.). Alle Suchkriterien sind mit „und“ verknüpft.

```
public Criteria
     (String name, DomainsValue searchPhrase, int comparator)
public String name()
public DomainValue searchPhrase()
public int comparator()
public static final int EQUALS = 1;
public static final int BEGINS_WITH = 2;
public static final int CONTAINS = 3;
public static final int GREATER = 4;
public static final int GREATER_OR_EQUAL = 5;
public static final int EQUAL = 6;
public static final int SMALLER = 7;
public static final int SMALLER_OR_EQUAL = 8;
public static final int EVALUATES_TO = 9;
```

Abbildung 18: Schnittstelle der Klasse Criteria

Im folgenden Beispiel wird nach allen Kunden gesucht, deren Name mit A beginnt und deren Umsatz größer als 100.000 DM ist.

```
Criteria criteria[] = new Criteria[2];
criteria[0] = new Criteria
     ("NAME", dvString.Factory.valueOf("A"), BEGINS_WITH);
criteria[1] = new Criteria
     ("TURNOVER", dvInteger.Factory.valueOf(100000), GREATER);
_searchResult = _registrar.search(Customer.class, criteria);
```

Verzichtet wurde auf komplexere Anfragemöglichkeiten wie „oder“-Verknüpfungen, Verschachtelung von Anfragen oder reguläre Ausdrücke, da diese Konzepte dann auch auf allen Implementierungen der Registratur umgesetzt werden müßten. Dies würde insbesondere bei der Dateiregistratur nur mit unverhältnismäßig hohem Aufwand möglich sein.
Um andere, wie die oben genannten Vergleichsoperatoren (zum Beispiel „ungleich)“ zu verwenden, sind an Anpassungen an verschiedenen Stellen des Quellcodes notwendig. Der neue Vergleichsoperator muß als Konstante in der Klasse der Suchkriterien (Criteria) definiert werden. Außerdem müssen die Suchfunktionen für die verschiedenen Persistenzmedien um den neuen Vergleichsoperator ergänzt werden. Für die Relationale-Datenbank-Registratur sind hierbei Anpassungen am Mapper notwendig, für die Dateiregistratur ist deren Klasse (RegistryFileImpl) zu ergänzen. Verwendet man einen Schnüffler (siehe 5.2.3), so muß in der Schnüfflerklasse der neue Vergleichsoperator ergänzt werden.

Anforderungen an zu suchende Objekte in der Registratur

In vielen Anwendungsfällen, in denen komplexe Objekte abgespeichert werden, ist es nicht notwendig, alle Attribute durch eine Suchfunktion auffindbar zu machen. Es sind meist nur wenige Attribute, nach denen gesucht wird. Beispielsweise reicht es bei einer Telefonkartei, zu einem Namen die passende Nummer zu finden. Der Fall, daß zu einer Nummer der passende Name gesucht wird, tritt normalerweise nicht auf. Daher reicht es aus, nur bestimmte Attribute eines Objektes für eine Suche zugänglich zu machen. Man erreicht damit eine höhere Suchgeschwindigkeit und muß weniger Verwaltungsinformationen für die Suche bereithalten. Die Information, nach welchen Attributen gesucht werden kann, befindet sich bei der Relationalen-Datenbank-Registratur in der an die Datenbank angepaßten Abbildungs-klasse. Da bei Verwendung der Dateiregistratur hierzu keine Möglichkeit besteht, wird diese Information zusätzlich im Material selber hinterlegt. Genutzt wird hierfür das in jeder

Materialbeschreibung (dvThingDescription) vorhandene „Attributes“-Feld. Dieses Feld ist bisher meist ungenutzt oder wird für Beschreibungen verwendet. Im folgenden Beispiel werden drei Attribute eines Kunden für die Suche zugreifbar gemacht:

```
Map attributes = new Hashtable();
attributes.put("NAME",dvString.Factory.valueOf(_customerName));
attributes.put("CITY",dvString.Factory.valueOf(_address.city()));
attributes.put
   ("TURNOVER",dvInteger.Factory.valueOf(_customerTurnover));
```

Implementierung der Suche in der Dateiregistratur

Die Suche in der Dateiregistratur funktioniert nach dem gleichen Prinzip, wie die in Kapitel 5.2.1. beschriebene Lösung zur Suche auf einem Inhaltsverzeichnis. Auch hier wird zunächst das komplette Inhaltsverzeichnis der Registratur in den Speicher geladen, um darauf die Suche durchzuführen. Die Suchanfragen müssen das zu Beginn dieses Abschnitts beschriebene Format haben. Es sind somit flexiblere Anfragen, als bei der Lösung aus Kapitel 5.2.1. möglich. Für den eigentlichen Suchvorgang wird eine dreifach verschachtelte Schleife benutzt. Die äußere Schleife durchläuft alle Einträge des eingelesenen Inhaltsverzeichnisses. Ein zweite Schleife innerhalb der ersten durchläuft alle beim Aufruf der Methode übergebenen Suchkriterien (criteria). Die dritte und innere Schleife durchläuft für jedes Suchkriterium alle Einträge des Attribut-Feldes der Materialbeschreibung. Für jeden Attributeintrag wird überprüft, ob der Name identisch mit dem des Suchkriteriums ist. Ist dies der Fall, wird geprüft, abhängig vom Vergleichsoperator, ob Wert oder Text des Eintrages in der Attributliste den Vergleich mit Wert oder Text des Suchkriteriums erfüllen. Für jeden Eintrag, der alle Vergleiche erfüllt, wird die zugehörige Materialbeschreibung in das Inhaltsverzeichnis des Suchergebnisses aufgenommen. Dieses Inhaltsverzeichnis wird als Ergebnis der Suchanfrage zurückgeliefert.

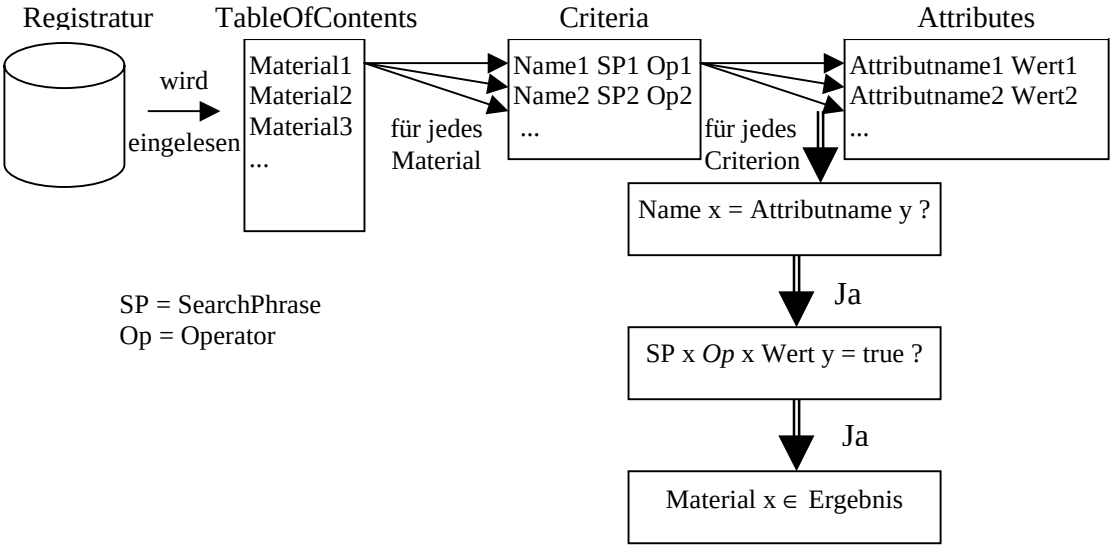


Abbildung 19: Ablauf des Suchvorgangs in der Dateiregistratur

Die Möglichkeit Suchanfragen zu stellen, bei denen die Suche auf die zu den Attributen passenden Felder beschränkt ist, erkauft man sich durch zusätzlichen Aufwand bei der Erstellung der Materialien, die später gespeichert werden sollen. Da jedes Material, das später durch eine Suchanfrage auffindbar sein soll, ein Attribut-Feld benötigt, ist ein Anpassung

dieser Materialien notwendig. Es ist außerdem nicht möglich, die Liste der Attribute, nach denen gesucht werden kann, nachträglich zu erweitern.

Implementierung der Suche in der Registratur für relationale Datenbanken

Neben der Dateiregistratur ist die Registratur für relationale Datenbanken (RegistryRDBMS-Impl) das zweite Persistenzmedium, für das eine Suchfunktion realisiert wurde. Die Lösung unterscheidet sich dabei erheblich, von der im vorigen Abschnitt vorgestellten Lösung.
Die Materialien, die durch diese Suchfunktion auffindbar sein sollen, werden in der verwendeten relationalen Datenbank folgendermaßen gespeichert: Jedem Material ist eine Tabelle in der Datenbank zugeordnet. In dieser Tabelle sind alle Informationen über dieses Material in verschiedenen Spalten gespeichert. Für jedes Attribut, das durch eine Suchanfrage auffindbar sein soll, ist ein Spalte vorhanden. Weiterhin gibt es für alle Verwaltungsinformationen wie Besitzer und Erzeuger des Materials, Erzeugungs- und letzter Zugriffszeitpunkt sowie Registrierungsnummer eine eigene Spalte. Schließlich sind je nachdem wie der Mapper Materialien auf die Datenbank abbildet, Spalten für jedes Attribut vorhanden. Alternativ kann bei objektrelationalen Datenbanken das ganze Material in serialisierter Form auch als BLOB in nur einer Spalte gespeichert werden. Neben der Tabelle, die dem Material zugeordnet ist, gibt es noch eine gemeinsame Tabelle, für alle Materialien (Thingdirectory-table). Diese Tabelle dient der Zuordnung von Objekt-IDs und Registrierungsnummern. Sie ist für die Suche nicht relevant.

Name	City	...	ID	Possessor	Creator	Creationtime	...	Regno
Mustermann	Musterdorf		39e706a4...		0;default User	Dec 11 19:46:05		test:1
Meyer	Hamburg		39e706a4...		0;default User	Dec 12 20:47:06		test:2
Müller	Bremen		39e706a4...		0;default User	Mar 29 13:47:22		test:3

Abbildung 20: Ausschnitt aus einer Datenbanktabelle der Beispielanwendung „Kundenverwaltung“

Der Ablauf einer Suchanfrage an die Relationale-Datenbank-Registratur sieht folgendermaßen aus: Die Suchanfrage, wird von der Relationalen-Datenbank-Registratur an einen dem Material zugeordneten Mapper weitergeleitet. Diesen Mapper bestimmt sie anhand des Ergebnisobjektes, das bei der Suchanfrage als Parameter mitübergeben wurde. Der Mapper übernimmt die weitere Ausführung der Suchanfrage.
Die Suchmethode durchläuft in einer Schleife alle Suchkriterien und erstellt aus jedem einen passenden Teil-SQL-String für eine SQL-Suchanfrage. Für das erste Suchkriterium entsteht ein String mit folgendem Aufbau:

```
select * from TABLE where NAME op 'TEXT'
```

op steht für den Vergleichsoperator (>,<,=, etc.)
Für jedes weitere Suchkriterium wird dieser String weiter ergänzt:

```
and NAME op 'TEXT'
```

Dadurch entsteht ein SQL-String, der die komplette Suchanfrage beinhaltet. Das Beispiel aus der Dateiregistratur würde folgendermaßen aussehen:

```
select * from CUSTOMERTABLE where NAME LIKE 'A%'
                                 and TURNOVER >= 100000
```

Diese Anfrage wird an die Datenbank übergeben und anschließend ausgeführt. Die Datenbank liefert als Ergebnis eine Ergebnismenge (ResultSet) zurück. Zu jedem Element dieser Ergebnismenge wird die Materialbeschreibung aus der Datenbank gelesen und alle Beschreibungen in Form eines Inhaltsverzeichnises als Ergebnis der Suchanfrage zurückgeliefert.

5.2.3. Integration der erweiterten Suchfunktion in einen Schnüffler

Um die Vorteile einer flexibleren und schnelleren Suchfunktion mit der Erweiterbarkeit und dem im objektorientierten Sinne besseren Aufbau des Schnüfflers zu kombinieren, wurden die im vorigen Abschnitt vorgestellten Suchmethoden in den Standard-JWAM-Schnüffler integriert. Auch bei diesem neuen Schnüffler (UniversalSniffer) liegt die Suchfunktionalität in der `doRun`-Methode, die mit der neuen Implementierung der Suche überschrieben wird (siehe Kapitel 3.2.).
Zunächst wird geprüft, ob es sich bei dem verwendeten Behälter um eine Datei- oder um eine Relationale-Datenbank-Registratur handelt. Liegt eine Dateiregistratur vor, so wird die Suche, wie in Kapitel 5.2.2. beschrieben, auf dem Inhaltsverzeichnis ausgeführt. Handelt es sich um eine Registratur für relationale Datenbanken, wird die Anfrage an den zum gesuchten Material passenden Mapper weitergeleitet und von diesem, wie im vorigen Kapitel beschrieben, ausgeführt.
Das Schnüffler-Konzept wurde aufgegriffen, da es hiermit möglich ist, Anfragen an gespeicherte Objekte zu stellen, die über das Suchen nach vorher festgelegten Attributen hinausgehen. An ein gespeichertes Objekt kann die Anfrage gestellt werden, ob es eine bestimmte Eigenschaft erfüllt. Hierzu wird an dem entsprechenden Objekt eine sondierende Methode aufgerufen, die als Ergebnis entweder wahr oder falsch liefert und damit entscheidet, ob die gesuchte Eigenschaft erfüllt ist oder nicht. Um eine Methode eines Objektes aufzurufen, muß dies zuerst komplett aus dem Persistenzmedium eingelesen werden. Für eine Suchanfrage ist somit ein Einlesen jedes vorhandenen Objektes notwendig. Hieraus folgt, daß Suchanfragen dieser Art nicht besonders schnell sind.
Da der Aufruf der sondierenden Methode nur ausgeführt werden kann, wenn der konkrete Name bekannt ist, läßt sich dieses Konzept nicht in einen allgemeinen Schnüffler integrieren. Für jedes zu suchende Objekt muß daher ein eigener Schnüffler erstellt werden, der das Grundkonzept des universellen Schnüfflers übernimmt. Um diese neue Art von Anfragen stellen zu können, wird ein zusätzlicher Vergleichsoperator „`EVALUATES_TO`“ eingeführt. Beinhaltet eine Suchkriterium diesen Operator, wird eine dem Namen des Suchkriteriums zugeordnete Methode aufgerufen und geprüft, ob das Ergebnis mit dem Vergleichswert übereinstimmt. Folgendes Codebeispiel ist Teil einer Anfrage an ein Kundenobjekt, ob die Eigenschaft „guter Kunde“ erfüllt ist. Es stammt aus der `doRun`-Methode des neuen universellen Schnüfflers (UniversalSniffer).

```
if (comparator == Criteria.EVALUATES_TO)
{
  if (name == "goodCustomer")
  {
    // Objekt einlesen
    Registerable reg = ((Registry)sniffSpace()).thing(
      ((dvRegisterableDescription)toc.description(tocIndex)).
      registrationNumber());
```

```
    // Aufrufen der Kundenmethode
    if (((Customer)reg).isGoodCustomer() ==
      ((dvBoolean)searchPhrase.value()))
    {
      fulfilsSearchForOneParameter = true;
    }
  }
}
```

Das hierzu gehörige Suchkriterium lautet:

```
criteria[0] = new Criteria
  ("goodCustomer",dvBoolean.Factory.valueOf("true"), EVALUATES_TO);
```

Der erste Parameter wird benutzt, um die sondierende Methode zu identifizieren, die am Objekt aufgerufen wird. Der Name dient hier nur als Auswahlkriterium, wird sonst aber nicht weiter benutzt. Der zweite Parameter gibt den boolschen Wert (true oder false) an, den die sondierende Methode des Objektes als Ergebnis haben soll, wenn das Objekt zum Suchergebnis gehört. Mit dem dritten Parameter wird festgelegt, um welche Art Anfrage es sich handelt. Ist dieser Parameter die Konstante `EVALUATES_TO`, so handelt es sich um einen Aufruf einer sondierenden Methode an einem Objekt.
Bei dem gesamten Konzept handelt es sich bisher nur um erste Ideen, die im Rahmen der Studienarbeit nicht mehr vollständig evaluiert wurden. Es sollte nur gezeigt werden, daß auch Anfragen möglich sind, die über das eigentlich vorgesehene Schema hinausgehen. Daher muß bis jetzt auch für jede Art von Anfrage der Schnüffler angepaßt werden. Generische Anfragen sind nicht möglich.

5.3. Bewertung der Lösungen

Mit dem neuentwickelten Schnüffler und der erweiterten Registratur ist es möglich, Anfragen zu stellen, die eine wesentlich genauere und eingegrenztere Suche erlauben, als es bisher möglich war. Zusätzlich bietet die neue Lösung eine größere Flexibilität bei der Auswahl des Speichermediums. Eine Anwendung kann ohne Anpassungen die Dateiregistratur oder jede Datenbank, für die es ein Registraturanschluß gibt, benutzen.
Fraglich ist aber, ob das Schema der Suchanfragen ausreichend ist oder ob eine SQL-artige Anfragesprache notwendig ist. Ein Nachteil der vorgestellten Lösung liegt darin, daß im voraus festgelegt werden muß, nach welchen Attributen später gesucht werden soll und es auch nicht möglich ist, die Liste dieser Attribute nachträglich zu erweitern. Es muß im Werkzeug Wissen über die Attribute des Materials vorhanden sein. Damit sind auch immer Anpassungen am Werkzeug notwendig, wenn es zu Änderungen im Material kommt. Dies widerspricht dem Prinzip der Datenkapselung in der objektorientierten Programmierung.
Betrachtet man die Geschwindigkeit der Ausführung von Suchanfragen, so stellt sich heraus, daß diese nahezu unabhängig von der Anzahl der Suchparameter und der Größe des Persistenzmediums ist. Bei Verwendung der Relationalen-Datenbank-Registratur in Verbindung mit der Datenbank „JDataStore“ hängt die Ausführungszeit nur von der Anzahl der gefundenen Ergebnisse, nicht aber von der Komplexität der Anfrage oder der Anzahl vorhandener Objekte in der Datenbank ab. Wird als Persistenzmedium die Dateiregistratur eingesetzt, sind die Auswirkungen einer Vergrößerung des zu durchsuchenden Datenbestandes ebenfalls gering. Der größte Teil der Ausführungszeit einer Suchanfrage wird für das Einlesen und Wiederaufbauen der gespeicherten Objekte und ihrer Material-

beschreibungen benötigt. Daraus läßt sich schließen, daß weitere Optimierungen in Bezug auf die Ausführungsgeschwindigkeit zu erst an diesen Stellen notwendig sind, bevor eine weitere Optimierung der Suchanfragen notwendig wird.

6. Zusammenfassung und Ausblick

Ziel der Arbeit war es, die JWAM-Registratur um eine erweiterte Suchfunktionalität zu ergänzen, die eine gezieltere und gleichzeitig schnellere Suche erlaubt und dabei unabhängig vom verwendeten Persistenzmedium ist.
Hierfür wurden zunächst allgemeine Grundlagen zur Suche und zu den verwendeten Datenbanken dargestellt. Es folgte eine Analyse der bestehenden Komponenten und Konzepte des JWAM-Rahmenwerkes auf die die erweiterte Suche aufbaut. Die Suche muß sich in diese bestehende Architektur einfügen, ohne große Änderungen erforderlich zu machen.
Anschließend wurden verschiedener Lösungen, die bis jetzt zur Suche in der Registratur eingesetzt wurden, verglichen. Es zeigte sich, daß sowohl das zum Rahmenwerk gehörende Konzept des Schnüfflers als auch die an die jeweilige Anwendung angepaßten Such-funktionen keine ausreichend präzisen Suchanfragen erlauben. Da diese Suchfunktionen zum Teil auch nur auf einem Persistenzmedium arbeiten, wird deutlich, daß der Bedarf einer Neuentwicklung besteht.
Im vierten Kapitel wurden daher Persistenzschnittstellen, die von externer Seite entwickelt wurden und die dazugehörenden Suchfunktionen, untersucht. Hier wurde zunächst allgemein und dann am Beispiel des Programms „TopLink“ vorgestellt, wie Objekte auf relationale Datenbanken abgebildet werden können. Es folgte eine Analyse von JDO, dem zukünftigen Standard einer Persistenzschnittstelle für Java. Beide vorgestellten Lösungen, TopLink und JDO, bieten eine sehr umfangreiche Schnittstelle für jede Art von Suchanfragen. Die Handhabung der Anfragen ist bei JDO einfacher, da es nur eine Art Anfragen gibt, während TopLink drei verschiedene Arten von Anfragen benutzt. Dafür sind bei TopLink die Anfragen durchgehend objektorientiert, während bei JDO Strings für einige Parameter verwendet werden. TopLink erlaubt nur den Einsatz von relationalen Datenbanken und ist zudem, da es sich um ein kommerzielles Produkt handelt, nicht allgemein verfügbar. JDO mangelt es bis jetzt noch an Unterstützung von einem Großteil der Datenbankhersteller. Dies soll sich aber in Zukunft ändern, wenn man den Ankündigungen verschiedener Hersteller glauben darf.
Im fünften Kapitel wurden verschiedene Lösungen für eine erweiterte Suchfunktionalität vorgestellt, die im Rahmen der Studienarbeit entwickelt wurden. Die Funktionalität wurde mit Hilfe einer Beispielanwendung überprüft. Als Beste der drei vorgestellten Lösungen hat sich die dritte Lösung, die als Basis einen Schnüffler verwendet, herausgestellt. Sie erlaubt Suchanfragen, die für den Bedarf vieler Anwendungen ausreichend sind, läßt dabei aber auch individuelle Anpassungen für Anfragen, die über das Standardschema hinausgehen, zu. Sie arbeitet unabhängig vom verwendeten Persistenzmedium und ist bei Verwendung der Dateiregistratur ausreichend schnell. Nachteilig an der vorgestellten Lösung ist, daß es notwendig ist, Anpassungen am Material vornehmen zu müssen und daß es nicht möglich ist, die Suche nachträglich anzupassen.
Es ist zu überlegen, ob das entwickelte Schema für Suchanfragen wirklich ausreichend ist, oder ob ein mächtigeres Hilfsmittel, wie etwa eine SQL-artige Anfragesprache, nötig ist.
Sollte sich JDO als Standard etablieren, wäre die Entwicklung einer Anbindung der Registratur an diese Schnittstelle sehr interessant. Die Entwicklung von Mappern, die an das jeweilige Persistenzmedium angepaßt sind, wäre dann nicht mehr nötig. Gleichzeitig stünde eine mächtige Schnittstelle für die Erstellung von Suchanfragen zur Verfügung, die auf allen angeschlossenen Persistenzmedien arbeiten könnte.

Literaturverzeichnis

[Ambysoft 00]
Ambysoft Persistence Layer Web-Site:
http://www.ambysoft.com/persistenceLayer.html

[Cattell 93]
R. G. G. Cattell (Hrsg.): „The Object Database Standard: ODMG-93“. San Mateo, Calif.: Morgan Kaufmann Publishers, 1993.

[Date 87]
C. J. Date: „A guide to the SQL standard“. Reading, Mass.: Addison-Wesley, 1987.

[Date 00]
C.J. Date: “An Introduction to Database System”. Reading, Mass.: Addison-Wesley, 2000.

[Geiger 95]
K. Geiger: „Inside ODBC”. Redmond, Wash.: Mircrosoft Press, 1995.

[Gryczan et al. 00]
G. Gryczan, A. Havenstein, S. Roock, I. Wetzel, H. Züllighoven: „Frameworkbasierte Anwendungsentwicklung (Teil 5)“, in: OBJEKTspektrum 1/2000.

[Hamilton et al. 97]
G. Hamilton, R. Cattell, M. Fisher: „JDBC database access with Java“. Reading, Mass.: Addison-Wesley, 1997.

[Havenstein 00]
A. Havenstein: „Unterstützung kooperativer Arbeit durch eine Software-Registratur“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 2000.

[JDO 00]
Sun Mircosystems, Inc.: „Java Data Objects – JSR 000012, Version 0.8“. Palo Alto, Calif., 2000.

[Keller 97]
W. Keller: „Mapping Objects to Tables“: A Pattern Language, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, 1997.

[Lippert 99]
M. Lippert: „Die Desktop-Metapher in Systemen nach dem Werkzeug- und Material-Ansatz“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1999.

[Otto & Schuler 00]
M. Otto, N. Schuler: „Fachliche Services: Geschäftslogik als Dienstleistung für verschiedene Benutzungsschnittstellen-Typen“, Diplomarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 2000.

[Schneider 97]
H.-J. Schneider (Hrsg.): „Lexikon Informatik und Datenverarbeitung“. München: R. Oldenbourg Verlag, 1997.

[Skutta & Thiel 99]
Michael Skutta, Olaf Thiel: „Verwendung und Erweiterung des JWAM Rahmenwerkes“, Studienarbeit am Arbeitsbereich Softwaretechnik, Fachbereich Informatik, Universität Hamburg, 1999.

[Versant 00]
Versant Judo Web-Site:
http://www.versant.com/developer/components/judo/index.html

[WebGain 01]
WebGain TopLink Web-Site:
http://www.webgain.com/products/toplink.html

[Züllighoven 98]
H. Züllighoven: „Das objektorientierte Konstruktionshandbuch“. Heidelberg: dpunkt.verlag, 1998.