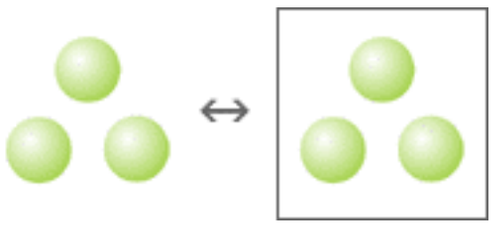
db4oView
for Java



Studienarbeit
SS 2008
Luca Graf 165977

Inhalt

1 Über db4o

Db4o ist ein OpenSource Projekt, das eine objektorientierte Datenbank bereitstellt. Die Datenbank (Engine) ist unter zwei verschiedenen Lizenzen erhältlich, unter der GPL [1], die den freien (kostenlosen) Download und den Einsatz in freien auch unter GPL stehenden Projekten erlaubt. Darüber hinaus gibt es noch eine kostenpflichtige Lizenz, die für den Einsatz in kommerziellen Projekten gedacht ist, mit der man Zugriff auf Developer Tools(Enterprise Package) und professionellen Kundensupport erhält.

Heute kommen immer noch viele Persistenzlösungen wie Hibernate [2] zum Einsatz, die über Mapping versuchen, Objekte und Okjektstrukturen in „gewohnten“ relationale Datenbanken abzubilden. Objektorientierte Datenbanken wie db4o stehen diesen Lösungen gegenüber und versuchen Objekte auf natürliche Art - im Sinne von Objektorientierung - direkt zu speichern. Dies hat viele Vorteile, Mapping-Lösungen wie Hibernate, stoßen schnell an ihre Grenzen, wenn es darum geht, grundlegende Konzepte wie Vererbung, Polymorphie, komplexe Referenzen etc. über Mapping abzubilden. Aber auch die extra Arbeit für die Entwicklung eines Mappings und deren Datenstruktur kosten Zeit, die bei Verwendung objektorientierter Datenbanken wegfällt, da das Datenmodell direkt aus der Anwendung heraus persitiert werden kann.

Auszug von http://db4o.com:

> db4o offers the ultimate persistence solution to store objects of any complexity natively, with only one line of code.

Code Besipiel:

```
ObjectContainer db = Db4o.OpenFile(“C:/tmp/myDB.yap“);
db.Set(new Professor(“Volker“, “Sänger“));
db.Commit() ;
db.Close();
```

Das Beispiel zeigt, wie einfach Objekte direkt und ohne Umwege in der Datenbank abgelegt werden können.

Db4o bietet wie die meisten anderen Datenbanksysteme keine stringbasierte Anfragesprachen (Query Language) wie beispielsweiße SQL.

Db4o bietet 3 verschiedene Anfragekonzepte:

1. Query By Example
2. S.O.D.A
3. Native Queries

1.1 Anfragekonzepte unter db4o

1.1.1 Query By Example:

Ist die einfachste, aber auch ausdrucksschwächste Variante. Es werden alle Objekte, die einem Musterobjekt „ähneln“, zurückgeliefert. QbE ist sehr gut für trivial Suchen geeignet.

Code Besipiel:

```
ObjectContainer db = Db4o.openFile(“C:/tmp/myDB.yap“);
Professor tmp = new Professor();
tmp.name = “Sänger“;
ObjectSet<Professor> res = db.get(tmp);
p1 = (Professor) res.Next();
```

Durch den Aufruf der statischen Methode openFile() erhält man einen ObjectContainer, der die grundlegende Schnittstelle zur Datenbank repräsentiert. Das Ausführen der Methode get() (ausführen eines QbE) auf einem ObjectContainer, liefert ein ObjectSet, das alle Objekte, die dem Musterobjekt „ähneln‘ enthält. Ein ObjectSet implementiert die Java Interfaces List und Iterator und kann somit normal als List/Iterator verwendet werden.

Sonderfälle:

1. Abfrage von Class Objekten(z.B. Professor.class)

   Es werden alle Objekte der Klasse zurückgeliefert.

2. Musterobjekt ist null

   Es werden alle Objekte der Datenbank zurückgeliefert.

Fazit:

QbE ist einfach und elegant, erlaubt aber keine komplexeren Abfragen, etwa unter Verwendung von logischen Verknüpfungen, die nicht formuliert werden können. Dadurch eignet sich QbE eher für den Einstieg oder kleine Spielereien, für den produktiven Einsatz bzw. komplexere Problemstellungen sind S.O.D.A/Native Queries gedacht.

1.1.2 S.O.D.A (Simple Object Database Access):

Das Basiskonzept einer S.O.D.A.[3]-Anfrage ist ein Querygraph. Ein Querygraph kann man sich auch als eine Art Musterobjekt vorstellen, nur stellt er kein lebendiges Anwendungsobjekt, sondern eher eine abstrakte Objektspezifikation (in Form eines Graphen/Baumes) dar.

Code Beispiel:

```
ObjectContainer db = Db4o.OpenFile("C:/tmp/myDB.yap");
Query query = db.query();
query.constrain(Professor.class);
ObjectSet<Professor> res = query.execute();
```

Das Interface Query ist der Kern der S.O.D.A.-API, er kann als Wurzel des Querygraphen gesehen werden. Dem Query können nun sukzessive Suchkriterien hinzugefügt werden. Im Beispiel ein Extent (Professor.class). Die Methode execute() sendet die Abfrage, und liefert als Ergebnis wieder ein ObjectSet. Das Beispiel würde wieder alle Objekte der Klasse Professor liefern.

Wie zuvor schon erwähnt, lassen sich in einem S.O.D.A-Query auch komplexere Suchkriterien vereinigen.

Code Beispiel:

```
ObjectContainer db = Db4o.OpenFile(„C:/tmp/myDB.yap");
Query query = db.query();
query.constrain(Professor.class);
query.descend("name")
      .constrain("Sänger")
      .or(
          query.descend("name")
                .constrain("Hammer");
ObjectSet<Professor> res = query.execute();
```

Über die Methode descend(), können Objektfelder dem Extent (Professor.class) hinzugefügt werden, constrain() enthält die dazu gesuchten Feldwerte. Descend() liefert einen zusätzlichen (untergeordneten) Queryknoten zu unserem Extent (siehe Abb1 Seite 8). Dadurch ist man in der Lage, „Musterobjekte" sukzessive aufzubauen, die auch logische Bedingungen erfüllen, da sich mehrere descend() Anfragen mit and,or,not verknüpfen lassen. Darüber hinaus bieten S.O.D.A-Anfragen die Möglichkeit, constrains mit Vergleichsoperatoren wie beispielsweise greater() zu verwenden und Querys nach Feldwerten sortieren zu lassen.

Sollte sich eine Anfrage nicht in S.O.D.A formulieren lassen, gibt es die Möglichkeit, beliebigen Code in die Anfrage einzubetten (Stichwort: Evaluation[4]) .

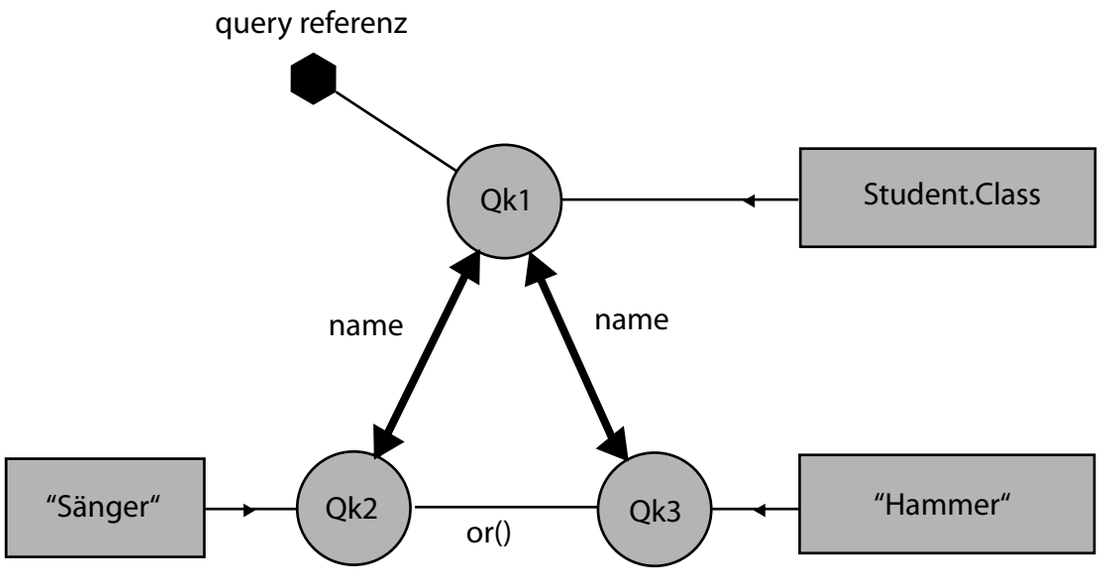


Abb1 Querygraph

Fazit:

S.O.D.A. ist eine von db4o unabhänige Queryspezifikation. Sie wurde in db4o eingebettet und wird teilweise von db4o Entwicklern betreut, aber unabhänig von db4o entwickelt. Durch das sukzessive Aufbauen von Objektbäumen, mit denen Abfragen eingeschränkt werden, ist es möglich komplexe Abfragen zu formulieren. Intern verwendet db4o ausschließlich S.O.D.A d.h., alle Anfragekonzepte werden intern in S.O.D.A Querys umgewandelt.

1.1.3 Native Queries:

Native Queries sind ein neuartiger Ansatz, der es erlaubt, Anfragen in der zugrundeliegenden Programmiersprache der Anwendung zu formulieren. Diese Art der Anfrage ist ein recht neuartiges Konzept, das auf ein Whitepaper von Cook/Rosenberger(Native Queries for Persistent Objects)[5] zurückgeht. Native Queries soll die zukünftige Hauptanfragesprache von db4o sein.

Code Beispiel:

```
ObjectSet<Professor> res = db.query(new Predicate<Professor>() {
        public boolean match(Professor p) {
                return p.getName() == “Sänger“
                        || p.getName() == “Hammer“;
        }
});
```

Hier wird der Querymethode eine Implementierung eines Predicate Objekts übergeben, der erwartete Extent Typ wird als Generic festgelegt. Die eigentliche Anfrage wird in der (abstrakten) Methode match() spezifiziert. Das Beispiel liefert alle Professoren mit dem Namen Sänger oder Hammer.

Fazit:

Auch wenn die Vorteile von Native Queries nicht auf den ersten Blick ersichtlich sind, sind sie gerade im Hinblick auf Wartbarkeit bemerkenswert. Da es sich hier um keine stringbasierte „Abfragemethode“ handelt, können Tippfehler von beispielsweise Feldnamen direkt durch die IDE oder spätestens zum Kompilierzeitpunkt des Codes abgefangen werden (was sich sonst erst durch Laufzeitfehler bemerkbar macht). Zusätzlich lassen sich alle Abfragen durch IDE Refactoring erfassen, und der Anwendungscode, speziell die Abfragen sind lange nicht mehr so fragil gegenüber Änderungen, wie es bei stringbasierten Ansätzen der Fall war. Ein weiterer Pluspunkt ist, dass sich der Entwickler nicht erst in neue Abfragesprachen einarbeiten muss, sondern mit der verwendeten Programmiersprache schnell und sicher Abfragen schreiben kann.

1.2 Weitere Features

Wie viele relationale Datenbanksysteme bringt db4o auch die Möglichkeit mit, Aktionen (Abfolgen von Lese-/Schreiboperationen) in Transaktionen zu kapseln. Db4o bietet ACID Transaktionen mit dem Isolationsgrad READ COMITTED.

Db4o stellt außerdem eine API für den Client-Server-Betrieb[6] bereit, mit dem es möglich ist, über die Grenzen einer einzelnen VM hinaus zu operieren.

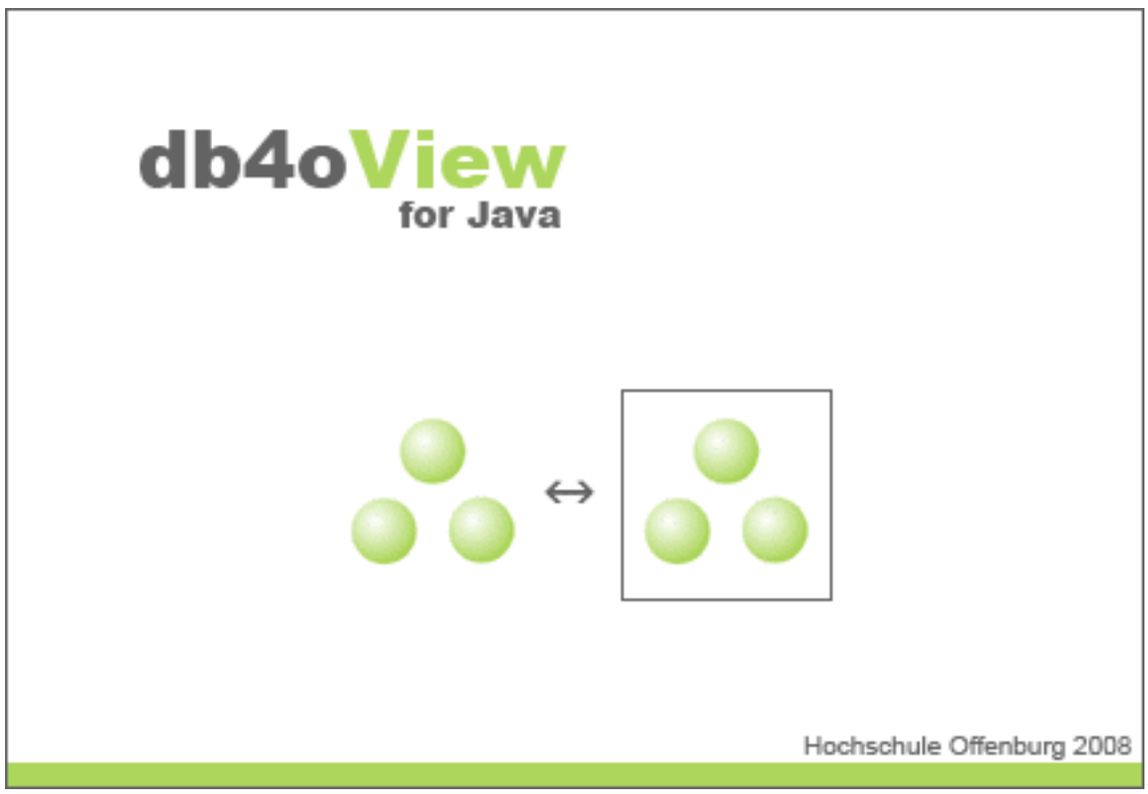
Ein weiteres feature ist Replikation[7], db4o erlaubt es anhand von Objektzuständen Veränderungen an Objekten zu erkennen, und alle Replikaten dieses Objekts zu aktualisieren. In Bezug auf Datenbanken lässt sich Replikation wie folgt verstehen: Ein Objekt aus „Datenbank A“ wird in „Datenbank B“ dupliziert. Bei eventuellen Änderungen wird das Replikat automatisch auf den neusten Stand gebracht.

2 Was ist db4oView

Objektorientierte Datenbanken sind ein spannendes Thema. Da ich meine Kenntnisse in Java und in OO-Programmierung weiter vertiefen wollte, entschied ich mich für ein Projekt mit db4o.

So entstand db4oView. Wie der Name schon erahnen lässt, handelt es sich um einen Viewer, der es erlaubt, Inhalte von db4o Datenbanken einzusehen. Aufgabe bzw. Intension war es, bestehende Datenbanken „anschaubar“ zu machen. Es sollten bestehende Klassen/Objekte übersichtlich dargestellt werden, und es sollte möglich sein, durch Objektstrukturen zu „navigieren“. Darüber hinaus sollten objektorientierte Konzepte wie Vererbung und Objekt-Zusammensetzung erkennbar sein.

Der grundlegende Aufbau der Anwendung besteht aus dem MVC-Pattern (Model-View-Controler) , auf das im nächsten Kapitel eingegangen wird.



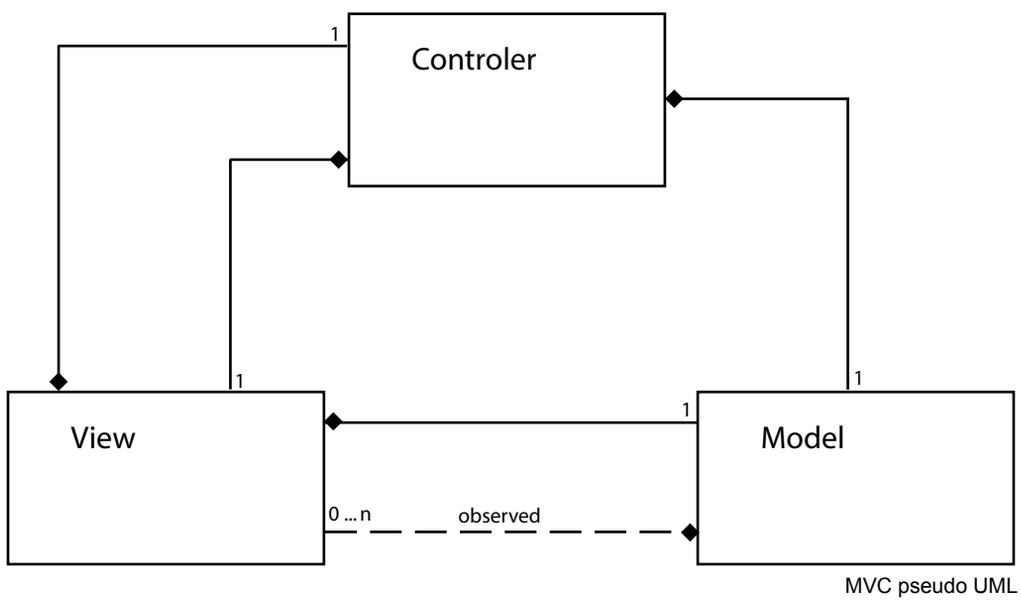
db4o Startscreen

3 MVC-Pattern

MVC (Model-View-Controler) beschreibt ein Entwurfsmuster zur logischen Strukturierung einer Anwendung (in drei Hauptteile). Ein Datenmodell (Model), das alle darzustellenden Daten und Datenhaltungslogik enthält, einer Präsentation (View) welche die Daten visualisiert und einem Controler der Benutzeraktionen des Views entgegennimmt und dementsprechend agiert (z.B. das Model auffordert, neue Daten für die View bereitzustellen und bestimmte Steuerelemente des Views zu verändern). Ändert das Model seine Daten, wird die View darüber informiert, und holt sich anschließend die aktuellen Daten selbständig vom Model ab und ändert gegebenfalls seine Darstellung. Dieser Vorgang wird mit Hilfe des Observer(Beobachter)-Muster[8] realisiert.

Durch die logische Trennung erhält man einen flexiblen und für Erweiterungen/Änderungen offenen Entwurf. Die View ist dadurch beispielsweise vom Model entkoppelt und in keiner Weise mehr davon abhängig. Dadurch wäre es möglich, das Model gegen ein beliebiges anderes Datenmodel auszutauschen, und die View und den Controler unangetastet weiterzuverwenden (vorausgesetzt beide Modelle implementieren das gleiche Interface).

Ziel des Pattern ist es, ein flexibles, für Erweiterungen offenes Design zu erhalten.



MVC pseudo UML

4 db4oView UML-Diagramm

5 Aufbau db4oView

5.1 Model

5.1.1 Observable Funktion

Das Model implementiert das Interface IObservable, das die benötigten Methoden festlegt, um das Model „beobachten“ zu können (Beobachter hinzufügen,entfernen und benachrichtigen). Es wurde hier bewusst nicht die von Java bereitgestellte Klasse java.util.Observable zur Realisierung des Observer-Musters verwendet, sondern eine eigene, für unseren Zweck zugeschnitte Implementierung.

Dafür implementiert die View vier verschiedene Observer-Interfaces, die jeweils eine Updatefunktion für eine Datenänderung beinhalten (Ein Interface pro veränderlichen Teil der Anwendung bsp. Objekttabelle). Die View registriert sich beim Model viermal als Beobachter, einmal pro Interfacetyp (das Model hält nun für jeden dieser Interfaces eine Referenz mit dem expliziten Interfacetyp). Bei Datenupdates kann die View nun je nach Art der Änderung explizit benachrichtigt werden und, es wird immer nur die spezielle Updatefunktion eines Interface aufgerufen. Dies erspart in der View viele if/else statements um herauszufinden, welcher Teil der Anwendung aktualisiert werden muss. Der Code bleibt dadurch geschlossen gegen Veränderungen, aber offen für Erweiterungen (Hinzufügen eines neuen Interfaces bei Bedarf).

5.1.2 Datenhaltungslogik

Zusätzlich implementiert es das Interface IModel, das den Zugriff von View und Controler auf das Model definiert. Hier sind die grundlegenden Methoden festgelegt, die der Controler bei Datenupdates aufruft und die „Gettermethoden“ mit denen die View auf die Daten des Models zugreifen kann.

Der Hauptteil des Models besteht aus der Logik, wie Daten aus der Datenbank geholt und für die weitere Nutzung (Darstellung) gehalten werden. Da es sich um eine generische Anwendung für Datenbanken handelt, war es ein Problem, dass die Klassen von Objekten der jeweiligen Datenbank zur Laufzeit nicht bekannt waren. Dies war aber nur auf den ersten Blick ein Problem, und ließ sich mit Hilfe von Metainformationen, die db4o über eine Datenbank bereitstellt, elegant lösen. Um den Entwurf weiterhin flexibel zu halten, wurde eine eigene Daten-/Objektstruktur (bestehend aus: IDbClass, IDbObject, IDbObjectMember (siehe UML-Diagramm)) entwickelt, um Klassen und Objekte unabhänig vom db4o API abzubilden (Um View/Controler unabhänig vom db4o API verwenden zu können).

Db4o bietet Zugriff auf Metainformationen der Datenbank, die Informationen über Klassen (StoredClass) und deren Felder (StoredField) beschreiben. Mit Hilfe dieser Informationen ist es möglich, ohne Kenntnisse über die eigentlichen Klassen die interne Klassenstruktur für eine Datenbank aufzubauen. Db4oView erstellt seine Klassenstruktur beim Öffnen einer Datenbank. Zu Beginn besteht die Struktur nur aus den Klassen, deren Feldern (Name und Typ) und einigen weiteren Informationen über die Klassen selbst (Superclass, Childclass). Die Daten über die einzelnen Instanzen der Klasse werden aus Performancegründen erst beim expliziten Auswählen einer Klasse geladen und dem Klassenobjekt hinzugefügt.

Code Beispiel:

```
StoredClass[] storedClasses = db.ext().storedClasses();
loadedClasses = new TreeMap<String, IDbClass>();

for (int i=0;i<storedClasses.length;i++) {
        List<IDbObjectMember> classMember = new ArrayList<IDbObjectMember>();

        StoredField[] members = storedClasses[i].getStoredFields();
        for(int j=0;j<members.length;j++) {
            IDbObjectMember member = ObjectMemberFactory.createMember();
            classMember.add(member);
        }

        loadedClasses.put(storedClasses[i].getName(),
                          new DbClass( storedClasses[i].getName(),classMember)
                           );
}
```

Das Beispiel zeigt den Zugriff auf die Metainformationen einer db4o Datenbank. Über die Datenbankreferenz „db“ können die in der Datenbank abgelegten Klassen abgefragt werden (db.ext(). storedClasses()). Man erhält ein Array mit StoredClass Objekten, welche die zuvor genannten Informationen enthalten.

Anhand dieser Informationen wird die in db4oView verwendete Datenstruktur aufgebaut (Eine Map (loadedClasses), die IDbClass Objekte enthält, die wiederum eine Liste aus IDbObjectMember Objekte enthält, welche die Klassenfelder repräsentieren). Hiermit wird Klassenstruktur der Datenbank in der db4oView eigenen Klassenstruktur abgebildet. Diese Daten werden beispielsweise von der View verwendet, um den Klassenbaum aufzubauen. Die Objektfelder (IDbObjectMember) werden mit Hilfe einer Factory erstellt, da sie je nach Datentyp unterschiedliche Funktionalitäten implementieren. Die Factory wird im nächsten Abschnitt beschrieben.

Das Codebeispiel wurde der Übersichtlichkeit halber etwas vereinfacht.

Jedes IDbClass Objekt enthält darüber hinaus ein Array mit IDbObject Objekten, welche die eigentlichen Instanzen jeder Klasse darstellen (werden bei Bedarf geladen und dem Klassen Objekt hinzugefügt). Jedes IDbObject enthält wiederum IDbObjectMember Objekte, welche die einzelnen Felder und deren Werte der Instanz beschreiben.

Code Beispiel:

```
Query instancesQuery = db.query();
instancesQuery.constrain(currentClass);
ObjectSet<GenericObject> instancesResult = instancesQuery.execute();

IDbObject[] instances = new IDbObject[instancesResult.size()];
int arrayIndex = 0;

while(instancesResult.hasNext()) {
        GenericObject obj = (GenericObject) instancesResult.next();

        List<IDbObjectMember> classMember = new ArrayList<IDbObjectMember>();

        for(int i=0;i<loadedClasses.get(currentClassName).getClassMember().size();i++) {
                IDbObjectMember member = ObjectMemberFactory.createMember()
                classMember.add(member);
        }

        IDbObject object = new DbObject(db.ext().getID(obj),classMember);
        instances[arrayIndex] = object;
        arrayIndex++;
}

loadedClasses.get(className).setInstances(instances);
```

Es werden über ein S.O.D.A Query alle Instanzen einer Klasse abgefragt. Für jede Instanz wird ein IDbObject erstellt, für jede Klassenvariable ein IDbObjectMember Objekt, die in einem Array abgelegt und ihrer Instanz (IDbObject Objekt) zugewiesen werden. Sind alle Objekte erstellt worden, werden sie ihrer Klasse (IDbClass Objekt) zugewiesen.

Das Codebeispiel wurde der Übersichtlichkeit halber etwas vereinfacht.

Das sind die grundlegenden Machanismen die notwendig waren, um eine Datenbankstruktur unabhänig vom db4o API abzubilden. Die Code Beispiele sind wie zuvor schon erwähnt stark vereinfacht, und sollen nur das grundlegende Konzept verdeutlichen. Für den interessierten Leser ist der komplette Quellcode auf der beigelegten CD einsehbar.

5.1.3 ObjectMemberFactory

Die Factory dient ausschließlich dazu um zu bestimmen, welcher Objekttyp für ein bestimmtes Objektfeld erzeugt wird. Da an verschiedenen Stellen im Quellcode Memberobjekte erzeugt werden, wurde der Code in ein Factory Objekt verlagert, um ihn zentral verwalten zu können. Das Factory Objekt stellt eine statische Methode createMember() bereit, die ein IDbObjectMember Objekt zurückliefert. Die Unterteilung in unterschiedliche IDbObjectMember wie beispielweise IDbCollectionMember war notwenig, da Collections anders wie triviale Datentypen in der Objekttabelle dargestellt werden (bsp. Collection: 3Itmes), anstatt dem eigentlichen Feldwert. Durch die Verwendung eines Interface, können alle IDbObjectMember in der restlichen Anwendung gleich behandelt werden, verwenden allerdings je nach Datentyp des Feldes, teilweise unterschiedliche Implementierungen.

Code Beispiel:

```
public static IDbObjectMember createMember(Object obj) {

        IDbObjectMember member = null;

        if(obj instanceof GenericObject) {
                member = new DbObjectMember(obj);
        }
        else if(obj instanceof java.util.Collection) {
                member = new DbCollectionMember(obj);
        }
        else if(obj instanceof java.util.Map) {
                member = new DbMapMember(obj);
        }

        ...

        else {
                member = new DbSimpleMember(obj);
        }
        return member;
}
```

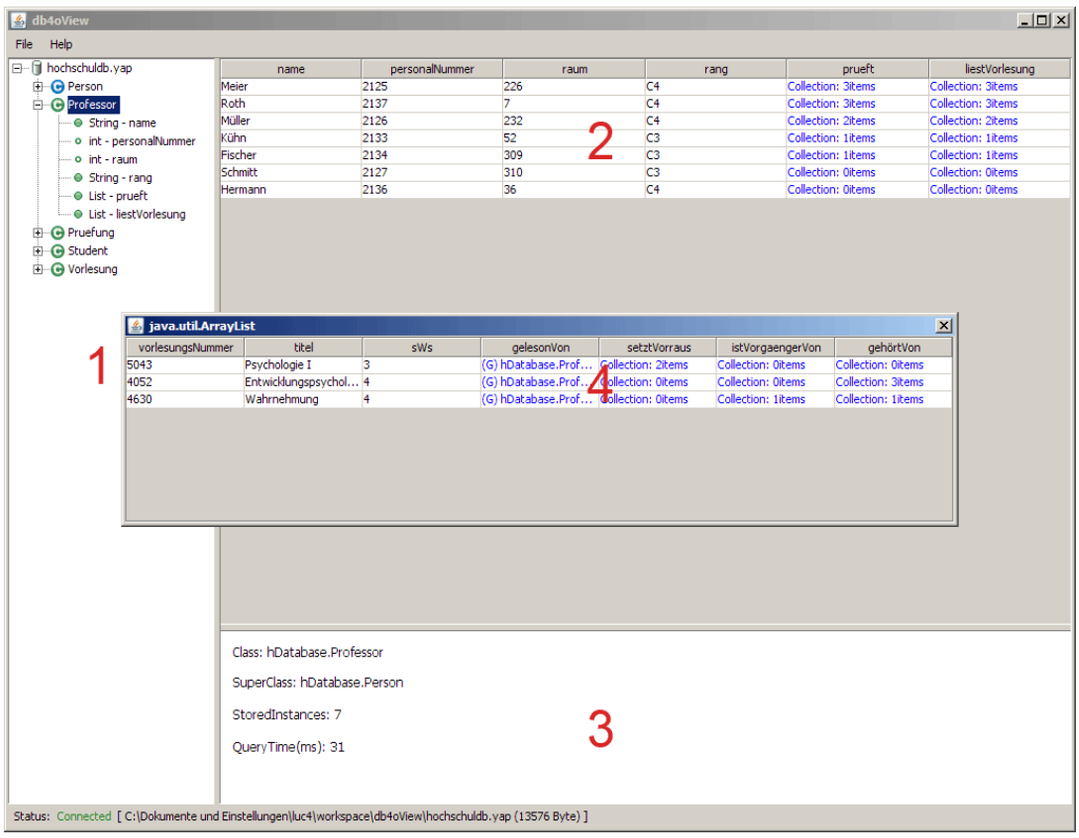
In der Methode createMember() werden die Feldwerte auf ihren Klassen-/Datentyp untersucht, und je nach Ergbnisse unterschiedliche Implementierungen von IDbObjectMember (Bsp. DbCollectionMember) zurückgeliefert.

Das Codebeispiel wurde der Übersichtlichkeit halber etwas vereinfacht.

5.2 View

Die View besteht aus vier Hauptteilen:

1. Klassenbaum
2. Objekttabelle
3. KlassenMetainformationen
4. Objekt/Collection Popup



Die View wurde mit Jigloo[9] erstellt und nutzt das Look&Feel JGoodiePlastic. Sie beinhaltet die vier zuvor genannten Hauptteile.

Die View implementiert wie schon erwähnt vier Interfaces(IClassTreeObserver,IObjectTableObserver,IMetaInfoObserver,IPopUpWindowObserver), die für das Observable-Muster benötigt werden. Das Model ist dadurch in der Lage, je nach Datenupdate eine der entsprechende updateX() Methode des Views aufzurufen, und somit eine gezielte Aktualisierung der Oberfläche einzuleiten.

Folgend werden kurz die einzelnen Teile der GUI und deren grundlegender Aufbau besprochen.

5.2.1 Klassenbaum

Der Klassenbaum stellt die Modeldaten (loadedClasses) grafisch dar. Er wurde mit dem von Java mitgelieferten JTree realisiert. Er benutzt ein selbst implementiertes TreeModel[10], mit dessen Hilfe es für den JTree möglich ist, einen Baum aus den Daten (loadedClasses) automatisch aufzubauen. Der JTree verwendet intern immer ein TreeModel. Das TreeModel stellt dem JTree die Daten plus spezielle „Helfer Methoden“ bereit, um aus den Daten einen Baum aufzubauen. Somit ist man in der Lage, mit Hilfe des TreeModel Interfaces ein TreeModel selbst für eine beliebige Datenquelle zu implementieren.

Code Beispiel:

```
public class ClassTreeModel implements TreeModel  {

        private String dbFileName;
        private Map<String, IDbClass> loadedClasses;

        public Object getRoot() { return dbFileName; }

        public int getChildCount(Object node) {
                if(node.equals(getRoot()) && loadedClasses!=null) {
                        return loadedClasses.size();
                }
                else if(node instanceof IDbClass) {
                        IDbClass dbCLass = (IDbClass)node;
                        return loadedClasses.get(dbClass.getClassName()).size();
                }
                else { return 0; }
        }

        public boolean isLeaf(Object node) {
                if(getChildCount(node)>0) return false;
                else                      return true;
        }

        public Object getChild(Object node, int nodeIndex) {}
        public int getIndexOfChild(Object parent, Object child) {}
```

Das Beispiel zeigt die Funktionen eines TreeModels mit deren Hilfe ein JTree in der Lage ist, aus einer beliebigen Datenquelle (hier loadedClasses) einen Baum aufzubauen.

Das Codebeispiel wurde der Übersichtlichkeit halber etwas vereinfacht.

Zur verbesserten Darstellung wurde zusätzlich ein eigener TreeCellRenderer[11] geschrieben, der beispielsweise Superklassen von normalen Klassen unterscheiden kann, und diese gesondert darstellt. Benutzeraktionen werden mithilfe eines TreeSelectionListener[12] realisiert, der gegebenfalls eine Aktion auf dem Controler auslöst. Updates auf den Klassenbaum werden durch die Methode updateClassTree() (Implementierung von IClassTreeObserver) ausgeführt.

5.2.2 Objekttabelle

Die Objekttabelle stellt die Instanzen einer Klasse (loadedClass.get(className).instances) dar. Sie basiert, ähnlich wie der Klassenbaum, auf einer von Java ausgelieferten Klasse, dem JTable[13], der ein ähnliches Interface (TableModel[14]) bereitstellt, um die Tabelle elegant aus vorhandenen Datenstrukturen aufzubauen. Auch hier wurden CellRenderer benutzt, um einfache Datentypen von speziellen Datentypen wie beispielsweise Objekten/Collections zu unterscheiden. Updates auf die Objekttabelle werden durch die Methode updateObjectTable() (Implementierung von IObjectTableObserver) ausgeführt.

5.2.3 Objekt/Collection Popup

Das Popup Fenster stellt Daten der nicht trivialen Datentypen detailiert dar. Es ist vom Prinzip her gleich wie die Objekttabelle aufgebaut und verwendet das selbe Tablemodell, sowie die gleichen CellRenderer, nur das der Table in einem gesonderten „Popup“ Fenster dargestellt wird. Updates auf diesen Teil werden durch die Methode updatePopUpWindow() (Implementierung von IPopUpWindowObserver) ausgeführt.

5.2.4 KlassenMetainformationen

Die KlassenMetainformationen sind einfache Textfelder (JLabel), die auf einem JPanel gruppiert sind. Updates auf diesen Teil werden durch die Methode updateMetaInfo() (Implementierung von IMetaInfoObserver) ausgeführt.

5.3 Controler

Der Controler nimmt wie zuvor schon erwähnt Benutzeraktionen des Views entgegen, und entscheidet, welche Aktion/en dieses Ereignis auslösen. Der View ist nur für die visuellen Aspekte der Anwendung zuständig, alle Entscheidungen über das Verhalten delegiert er an den Controler. Er kann somit als Strategy der View gesehen werden (Strategy-Muster[15]). Der Controler führt nicht nur Aktionen wie Datenupdates auf dem Model aus, sondern ruft auch Methoden auf der View auf, um beispielsweise Navigationselemente zu manipulieren.

Er implementiert das Interface IControler, das die verschiedenen Methoden für die Benutzeraktionen festlegt(Bsp. openDb4oFile).

Code Beispiel:

```
public void openDb4oFile(String filePath) {
        try {
                model.loadDbFile(filePath);
                view.disableOpenFileItem();
                view.enableCloseFileItem();
                view.editRightTopLabel(„Use the Left Panel to select a Class ...“);
        }
        catch (DatabaseFileLockedException e) {
                view.showErrorDialog(„Database already in use!“);
        }
}
```

Hier wird die Aktion „öffne Datenbank“ von der View an den Controler delegiert, der daraufhin eine Reihe von Aktionen ausführt. Er veranlasst das Model die Daten aus der Datenbank zu laden, und verändert einige Elemente des Views.

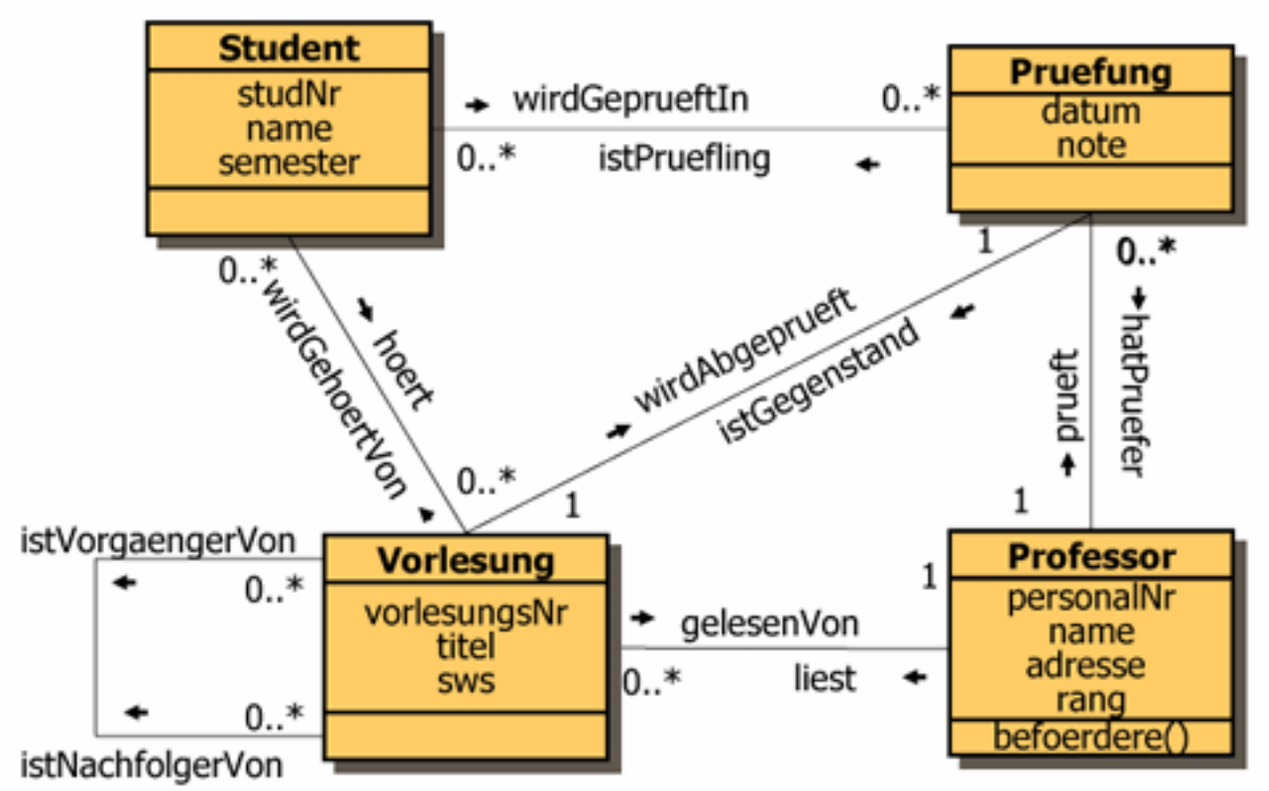
6 Hochschuldatenbank

Zur Veranschaulichung der Anwendung wurde die Hochschuldatenbank, die in den Vorlesungen oft als Beispiel diente, in db4o implementiert. Es wurde eine Objektstruktur, bestehend aus Professoren, Studenten, Vorlesungen und Prüfungen angelegt und beispielhaft initalisiert. Die mit Beispielwerten initalisierten Objekte wurde in einer db4o Datenbank abgelegt.

Code Beispiel:

```
//erstelle Objekte
Professor p1 = new Professor(2125, „Meier“,“C4“,226);
Student s1 = new Student(42001, „Becker“,8);
Vorlesung v1 = new Vorlesung(5001, „Mathematik I“,4,p7);

//setze Voraussetzungen
v2.setSetztVorraus(v1);
v1.setIstVorgaengerVon(v2);

//setzen Professoren zu Vorlesungen
p1.addVorlesung(v3);

//setze Studenten zu Vorlesungen
v1.addStudent(s2);
s2.addVorlesung(v1);

//setze Noten
Pruefung k1 = new Pruefung(v2,s2,new Date(),1.3);
s2.setPruefung(k1);


try {
        ObjectContainer db=Db4o.openFile(„C:/hochschuldb.yap“);

        //Speichere Objekte in Datenbank
        db.set(p1);
        db.set(s1);
        db.set(v1);
        ...
}
catch(DatabaseFileLockedException e) {
        e.printStackTrace();
}
```

Das Codebeispiel wurde der Übersichtlichkeit halber etwas vereinfacht.



Hochschuldatenbank UML-Diagramm

7 Fazit

Db4o macht Spass! Es ist den Entwicklern wirklich gelungen, eine Umgebung zu schaffen, in der es möglich ist, Objekte beliebiger Komplexität mit einer einzelnen Zeile Code zu speichern. Man ist nicht mehr zum ständigen Kontextwechsel zwischen objektorientierter Programmierung und relationalen Datenbankensystemen gezwungen, was die Arbeit deutlich angenehmer gestaltet.

Db4o ist intuitiv zu bedienen, und durch ein einfaches und gut gelungenes Einsteigertutorial[15], gestaltet sich der erste praktische Einstieg recht einfach. Auch die weiterführenden Tutorials[16], bieten einen guten Überblick über die Funktionen und Besonderheiten von db4o.

Trotz dieser Tatsachen ist es verwunderlich, dass db4o den „Durchbruch“ noch nicht geschafft und in der Praxis immer noch Mapping-Lösungen für relationalen Datenbanken bevorzugt werden, die oft einem flexiblen objektorientierten Entwurf entgegenstehen. Zusätzlich bringen diese Lösungen einen nicht zu unterschätzenden Overhead mit sich, der die Performance, Entwicklungsdauer und Wartung der Software zusätzlich einschränken.

Leider ist db4o anzumerken, dass es noch nicht flächendeckend eingesetzt wird, dementsprechend hält sich die Aktivität der Community zumindest in den Foren in Grenzen.

Der einzige wirkliche Minuspunkt, der mir während meiner Arbeit mit db4o auffiel, ist das die API zumindest an manchen Stellen doch eher dürftig bzw. wenig aussagekräftig dokumentiert ist, und man sich nur mit Hilfe engagierter Forennutzer oder „testen“ gewisse Informationen beschaffen konnte.

Abschliessend kann ich sagen, dass db4o ein sehr spannendes und interessantes Thema ist, das auch wirklich innovative Ansätze wie bsp. Native Queries mitbringt. Db4o muss sich meiner Meinung nach in keinster Weise hinter relationalen Datenbanken/Mapping-Lösungen verstecken, da es die meisten „wichtigen“ Features wie ACID Transaktionen, Replikation ebenso beinhaltet, aber eben den großen Vorteil bietet objektorientiert zu „denken“.

8 Verweise

[1] http://www.gnu.org/licenses/gpl-3.0.html
[2] http://www.hibernate.org/
[3] http://sodaquery.sourceforge.net/
[4] http://developers.db4o.com/resources/api/db4o-java/com/db4o/query/Evaluation.html
[5] http://www.odbms.org/download010.01%20Cook%20Native%20Queries%20for%20Persistent%20Objects%20August%202005.pdf
[6] http://developer.db4o.com/Resources/view.aspx/Reference/Client-Server
[7] http://developer.db4o.com/Resources/view.aspx/Reference/Db4o_Replication_System_DRS
[8] http://de.wikipedia.org/wiki/Observer_%28Entwurfsmuster%29
[9] http://www.cloudgarden.com/jigloo/index.html
[10] http://java.sun.com/javase/6/docs/api/javax/swing/tree/TreeModel.html
[11] http://java.sun.com/javase/6/docs/api/javax/swing/tree/TreeCellRenderer.html
[12] http://java.sun.com/javase/6/docs/api/javax/swing/event/TreeSelectionListener.html
[13] http://java.sun.com/javase/6/docs/api/javax/swing/JTable.html
[14] http://java.sun.com/javase/6/docs/api/javax/swing/table/TableModel.html
[15] http://de.wikipedia.org/wiki/Strategie_(Entwurfsmuster)
[16] http://developer.db4o.com/Resources/view.aspx/Formula_One_Tutorial