Rückwertssalto

A05

INSY

2014/2015

Autor: Daniel Melichar

Rückwertssalto

A05

Inhaltsangabe

Angabe und Überlegungen dazu

Erstelle ein Java-Programm, dass Connection-Parameter und einen Datenbanknamen auf der Kommandozeile entgegennimmt und die Struktur der Datenbank als EER-Diagramm und Relationenmodell ausgibt (in Dateien geeigneten Formats, also z.B. PNG für das EER und TXT für das RM)

Verwende dazu u.A. das ResultSetMetaData-Interface, das Methoden zur Bestimmung von Metadaten zur Verfügung stellt.

Zum Zeichnen des EER-Diagramms kann eine beliebige Technik eingesetzt werden für die Java-Bibliotheken zur Verfügung stehen: Swing, HTML5, eine WebAPI, … . Externe Programme dürfen nur soweit verwendet werden, als sich diese plattformunabhängig auf gleiche Weise ohne Aufwand (sowohl technisch als auch lizenzrechtlich!) einfach nutzen lassen. (also z.B. ein Visio-File generieren ist nicht ok, SVG ist ok, da für alle Plattformen geeignete Werkzeuge zur Verfügung stehen)

Recherchiere dafür im Internet nach geeigneten Werkzeugen.

Die Extraktion der Metadaten aus der DB muss mit Java und JDBC erfolgen.

Im EER müssen zumindest vorhanden sein:

- korrekte Syntax nach Chen, MinMax oder IDEFIX
- alle Tabellen der Datenbank als Entitäten
- alle Datenfelder der Tabellen als Attribute
- Primärschlüssel der Datenbanken entsprechend gekennzeichnet
- Beziehungen zwischen den Tabellen inklusive Kardinalitäten soweit durch Fremdschlüssel nachvollziehbar. Sind mehrere Interpretationen möglich, so ist nur ein (beliebiger) Fall umzusetzen: 1:n, 1:n schwach, 1:1
- Kardinalitäten

Fortgeschritten (auch einzelne Punkte davon für Bonuspunkte umsetzbar)

- Zusatzattribute wie UNIQUE oder NOT NULL werden beim Attributnamen dazugeschrieben, sofern diese nicht schon durch eine andere Darstellung ableitbar sind (1:1 resultiert ja in einem UNIQUE)
- optimierte Beziehungen z.B. zwei schwache Beziehungen zu einer m:n zusammenfassen (ev. mit Attributen)
- Erkennung von Sub/Supertyp-Beziehungen

Zeitaufwandschätzung

Funktion	Durchführung		Zeitaufwand		Testung	Kommentar
	[1-3][leicht-schwierig] Schätzung	Tatsächlich	[00:00][hh:mm] Schätzung	Tatsächlich	[ok-fehler]	
Verwendung von Exporter Resourcen			01:00	02:00		relativ einfach da bereits bei Exporter angewendet
Commandline Parser	1	1	00:30	01:00		änderungen der argumente notwending
Connection	1	1	00:30	01:00		keine/kleine Änderungen notwending
Benötigt noch Recherche			05:00	12:00		Hauptsächliche arbeit: schlau werden (libaries, design, etc)
Metadata aus Connection	2		02:00	06:00		Objekt passierend
RM in file	1		00:30	02:00		einfach mit MetaData Objekt
ER-Diagramm	3		02:30	04:00		Guidlines folgen (laut Angabe)
Unit-Testing						
Designüberlegungen				06:00		
Insgesamter Zeitaufwand			12:00	20:00		

Versionierung (über GitHub)

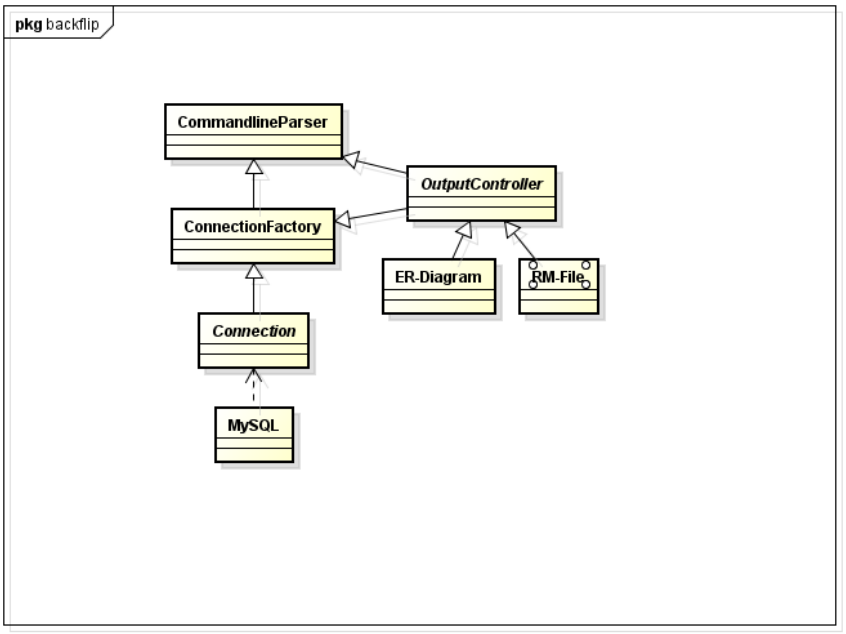
Die Versionierung fand auf einen privaten Repository, welches dann zu Aufgabenende public gemacht wurde, statt.

Link: github.com/dmelichar-tgm/jdbc

Designüberlegungen

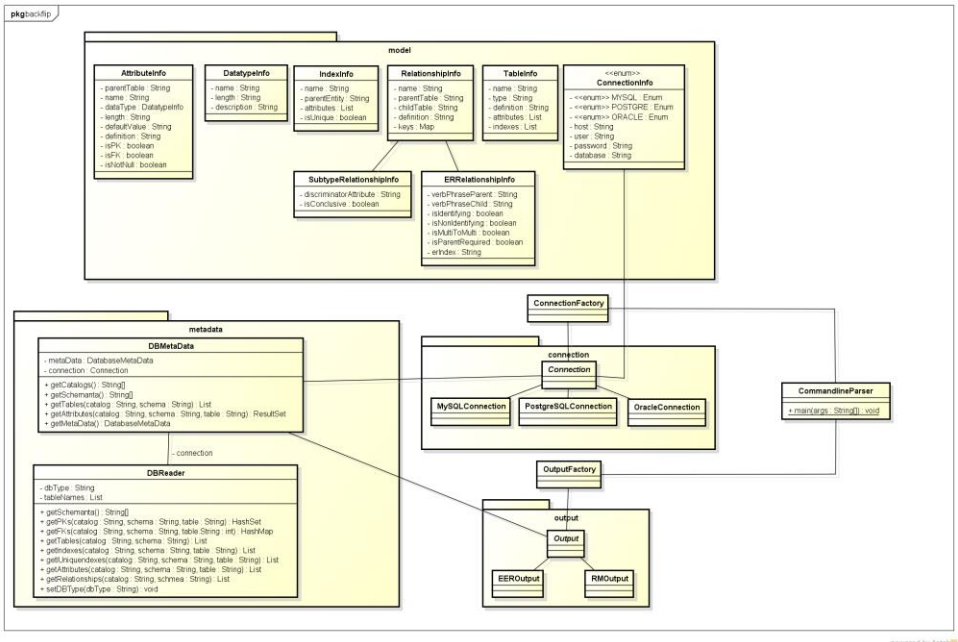
Bitte beachten: dies lediglich Überlegungen sind, viele Methoden, Attribute und dergleichen wurden "on-the-go" erstellt.

Erste Designüberlegung



Bei der ersten Designüberlegung wurde davon ausgegangen, dass man einfach eine veränderte Version des Exporters nehmen kann. Daher ist diese Klassendiagram ähnlich dem des Exporters mit dem einzigen Unterschied der Output Varianten

Zweite Designüberlegung:



Bei der zweiten Designüberlegung, welche nach einigen Vorschlägen von Herrn Prof. Borko getätigt wurde, hat sich das Klassendiagram um einiges erweitert. Hierbei ist vor allem die Modularität der Applikation sowie die Verwendung von Designpatterns zu beachten.

Durchführungsüberlegungen

Recherche nach Tools / Libaries

Die Hauptsächliche Arbeit bei dieser Aufgabenstellung war die Suche nach Tools beziehungsweise Libaries, welche die Anwendung vereinfachen. Für die Recherche wurde Hauptsächlich das Internet verwendet, aber teilweise auch Unterredungen mit anderen Mitschülern.

Prinzipiell wurde nach folgenden Dingen gesucht:

- Libary welche aus Metadata Grafiken erstellen kann
- Applikationen (die in Java geschrieben wurden) welche der Aufgabenstellung ähneln
- Plugins für bestehende Grafikprogramme (Astah, Dia, etc.)

SchemaSpy und SchemaCrawler

Bei der Recherche wurde auf SchemaSpy und SchemaCraler gestoßen. Beide basieren auf Graphviz (siehe unten). Der Grund warum beide nicht verwendet wurden war weil beide, meiner Meinung nach, nicht genügend Dokumentation für die Verwendung hatten.

SchemaSpy bietet viel Funktionalität, beispielsweise die Erstellung einer HTML-Page auf welcher sehr viel über die Datenbank eingesehen werden kann. Jedoch kann diese SchemaSpy nicht verwendet werden, da es einfach nur ein Runnable-Jar als Download gibt. Dies würde bedeuten, dass die Aufgabenstellung dann auf Argumente übergeben reduziert gewesen wäre.

Graphviz

Nachdem SchemaSpy und SchemaCrawler nicht wirklich vielversprächend waren, habe ich mich entschlossen zur Quelle beider zu gehen: Graphviz.

Von der Graphviz-Website (www.graphviz.org)

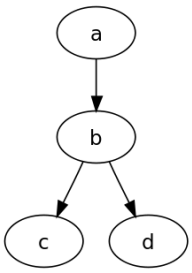
Graphviz is open source graph visualization software. Graph visualization is a way of representing structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and web design, machine learning, and in visual interfaces for other technical domains.

Graphviz bietet eine Reihe an Funktionalitäten an, welche bei der Aufgabenstellung notwendig oder hilfreich sind. Unglücklicherweise gibt es keine wirkliche Libary welche Verwendet werden kann, jedoch kann man in DOT die Semantik welche man in dem Graphen haben will angeben, und diese dann über Grapviz darstellen. Zunächst habe ich mich nicht auf Graphviz fixiert, und weiter nach anderen Tools gesucht, nachdem ich aber keine Gefunden habe welche mich angesprochen haben oder andere Probleme in die Quere stellen würden (z.B. ein Astah-Plugin welches Lizenrechtlich nicht möglich wäre), habe ich mich letztendlich dazu entschlossen das EER-Diagramm über dieses Tool durchzuführen.

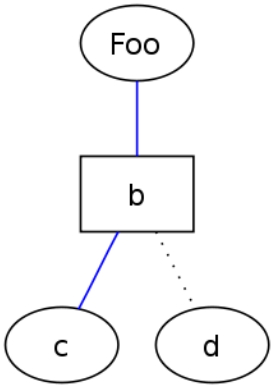
DOT (Programmier-)sprache

Die Dot-Sprache ist prinzipiell nicht schwer zu verstehen. Hier sind einige Beispiele von Wikipedia welche die Sprache gut darstellen.

```
digraph graphname {
    a -> b -> c;
    b -> d;
}
```




```
graph graphname {
    // This attribute applies to the graph itself
    size="1,1";
    // The label attribute can be used to change the label of a node
    a [label="Foo"];
    // Here, the node shape is changed.
    b [shape=box];
    // These edges both have different line properties
    a -- b -- c [color=blue];
    b -- d [style=dotted];
}
```



Wichtig hierbei sind die Labels. Diese können nämlich mit HTML-Tags ausgestatet werden um die Darstellung ein wenig zu Verbessen um bei z.B. der (min-max)-Notation die einzelnen Notation darzustellen oder um bei einer Beziehung, einen Namen auf den Strich zu stellen.

Zum Beispiel:

```
B [label=<<B>bold label B</B>>];
C [label=<<b>bold label C</b>>];
```

Erstellen der Grafik

Um von dem DOT-File auf die eigentliche Grafik zu kommen wurden über Graphviz einige executeables zur Verfügung gestellt. Diese müssen dann einfach in Java mit den korrekten Parametern aufgerufen werden und es wird die Grafik erstellt. Es ist zu beachten, dass die verschiedenen Executeables (oder auch *Roadmaps*) verschiedene Output varianten haben. Zum Beispiel stellt dot eine hierarchische Ansicht da, neato stellt *spring models* da, usw.

Letztendlich habe ich mich entschlossen neato zu verwenden, da es für mich am besten aussieht.

j

Durchführung

Bei Durchführung habe ich sehr viel Zeit mit den Prototypen verbracht. Ich habe verschiedenste Varianten versucht um quick und dirty auf die jeweilige Lösung zu kommen. Zugegebenermaßen sind die Prototypen sehr unüberschaubar, dies habe ich dann bei meinen Designüberlegungen versucht zu berücksichtigen und möglichst viele Model-Klassen zu erstellen.

Die Model-Klassen wurden als erstes händisch ausprogrammiert, da ich zu dem Zeitpunkt über keine kommerzielle Lizenz für Astah hatte. Prinzipiell sind alle Modelle auf die gleiche Art und Weise Strukturiert: Attribute und dazugehörige Getter- und Setter-Methoden. Weiters behinhlten die Modele überschriebene Equals-, Hashcode-, und toString-Methoden. Die Methoden wurden über die IDE generiert und bei Bedarf ein wenig Modifiziert. Viele der Modelle bzw. Methoden der Modelle werden gar nicht verwendet, jedoch war mir die Modularität und spätere Erweiterbarkeit der Applikation sehr wichtig.

Danach wurden die bei dem Exporter bereits erstellten Klassen so umgeschrieben, dass sie die Modelle verwenden. Zum Beispiel stellt der CommandLineController nun auch die Möglichkeit da ein ConnectionInfo-Objekt zu liefern, welche alle Information beinhaltet und für die Connection benötigt wird.

Am Anfang waren noch weitere Optionen zur Verfügung gestellt bei Aufruf des JARs, diese habe ich aber dann entfernt und über ein Properties-File gelöst, da die Angaben, welche in jenen File stehen, meist nicht geändert werden bei Öfteren Verwenden der Applikation (z.B. Ordner zu Graphviz).

Die Hauptsächliche Arbeit für die Applikation ist, meiner Meinung nach, in meinem Fall der DBReader. Hier werden alle benötigten Informationen über die Datenbank gesammelt und in einem guten Format zur Verfügung gestellt. Die Verwendung von Collections war hierfür das Beste. Auch beim DBReader habe ich versucht möglichst viele Erweiterungen möglich zu machen. Der Reader liest alles heraus, was für die Aufgabenstellung notwendig ist und funktioniert so gesehen sehr gut.

Beim Erstellen des RM-Outputs gab es nur leichte bis gar keine Probleme. Der Prototyp der hierfür erstellt wurde könnte beinahe 1:1 übernommen werden. Die Syntax in welcher das RM ausgegeben wird hat sich ein wenig geändert und ist nun, meiner Meinung nach, übersichtlicher.

Bei der Erstellung des EER-Outputs gab es aber die größten Probleme. Ein Dot file zu generieren stellte keine Schwierigkeiten da, dieses zu einem PDF zu konvertieren ebenso nicht, aber ein *richtiges* Dot File zu erstellen war relativ kompliziert. Prinzipiell wurde das DOT File über die Runtime und Neato auf ein PDF konvertiert und in den angegeben Ordner ausgegeben. Die Runtime hat mir hin und wieder Probleme gemacht und hat Neato nicht korrekt geschlossen, weswegen es hin und wieder kein PDF gab. Leider gibt Neato keine Rückmeldungen bzw. Errors aus, d.h. es musste selbst herausgefunden werden was der Fehler sein könnte wenn kein PDF Ausgegeben wurde. Meist handelte es ich um syntaktische Fehler im DOT File.

Kommentare

- Die Applikation benötigt eine Graphviz Installation
- Leider habe ich es nicht hin bekommen die *config.properties* Datei und die dazu gehörige Klasse so hin zu bekommen wie es ursprünglich geplant war. Es kann sein das die Datei neue Werte beim Wiederaufruf nicht liest und diese stattdessen mit den hart gecodeten überschreibt und diese dann auch verwendet. Der Grund dafür war das ich nicht genau wusste wie ich festlegen kann feststellen kann, dass das File bereits erstellt wurde.
- Ich habe versucht möglichst nach dem IDEFIX-Notation-Standard zu gehen. Ich habe dies nicht zu 100% hin bekommen, da die Erstellung von Graphviz Diagrammen nur bis zu einem gewissen Punkt die Möglickeit hierfür gibt.
- Es wurde versucht Möglichst nach Design Patterns vorzugehen um die Modularität und Erweiterbarkeit zu gewährleisten.

Statistik:

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
DBReader.java	400	267	67%	74	18%	59	15%
CommandLineController.java	359	236	66%	74	21%	49	14%
EEROutput.java	209	165	79%	18	9%	26	12%
Diagram_Prototype.java	186	131	70%	21	11%	34	18%
DBConnection.java	165	83	50%	55	33%	27	16%
AttributeInfo.java	138	108	78%	5	4%	25	18%
ERRelationshipInfo.java	135	106	79%	5	4%	24	18%
ConnectionInfo.java	119	93	78%	5	4%	21	18%
Output_Prototype.java	118	81	69%	9	8%	28	24%
TableInfo.java	113	87	77%	5	4%	21	19%
Attribute.java	108	38	35%	58	54%	12	11%
RelationshipInfo.java	106	81	76%	5	5%	20	19%
RMOutput.java	103	76	74%	13	13%	14	14%
ConfigUtil.java	87	55	63%	18	21%	14	16%
IndexInfo.java	87	65	75%	5	6%	17	20%
DatatypeInfo.java	75	54	72%	5	7%	16	21%
Table.java	75	26	35%	40	53%	9	12%
SubtypeRelationshipInfo.java	70	51	73%	5	7%	14	20%
AbstractConnection.java	68	35	51%	24	35%	9	13%
OutputController.java	64	37	58%	16	25%	11	17%
DBStaticFields.java	54	12	22%	32	59%	10	19%
ConnectionFactory.java	32	13	41%	12	38%	7	22%
MySQLConnection.java	32	15	47%	11	34%	6	19%
DatabaseTypes.java	26	16	62%	5	19%	5	19%
Test.java	23	12	52%	7	30%	4	17%
Main.java	20	12	60%	4	20%	4	20%
package-info.java	11	1	9%	10	91%	0	0%
package-info.java	11	1	9%	10	91%	0	0%
Output.java	10	4	40%	5	50%	1	10%
package-info.java	9	1	11%	8	89%	0	0%
package-info.java	8	1	12%	7	88%	0	0%
package-info.java	7	1	14%	6	86%	0	0%
package-info.java	6	1	17%	5	83%	0	0%
package-info.java	6	1	17%	5	83%	0	0%
Total:	**3040**	**1966**	**65%**	**587**	**19%**	**487**	**16%**

Quellen und lessons learned

Lessons Learned:

- JDBC Metadata
- Java Runtime
- Graphviz und DOT
- IDEFIX Notation Standard

Quellen und Ressourcen:

Graphviz Dokumentation: http://graphviz.org/Documentation.php

Runtime Propleme: http://www.javaworld.com/article/2071275/core-java/when-runtime-exec---won-t.html

Java DatabaseMetaData: http://docs.oracle.com/javase/7/docs/api/java/sql/DatabaseMetaData.html