Universität Paderborn

Fachbereich 17 – Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

Implementierung des History-Graph-Mechanismus in Java

Studienarbeit
zur Erlangung des Grades
Bachelor of Computer Science
für den integrierten Studiengang Informatik

von

Vladislav Krasnyanskiy
Peter-Hille-Weg 11
33098 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer

und
Prof. Dr. Hans Kleine Büning

Paderborn, Juli 2001

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 18 Juli 2001

_____

(Vladislav Krasnyanskiy)

Inhaltsverzeichnis

Kapitel 1 Einleitung ---- **6**

Kapitel 2 Grundlagen ---- **8**
2.1 Reengineering ---- 8
2.2 Verified Analysis and Reengineering of Legacy database systems using Equivalence Transformations (VARLET) ---- 10
2.3 Beispiel ---- 12

Kapitel 3 Konzepte und Werkzeuge ---- **16**
3.1 Restrukturierungstransformationen ---- 16
3.2 GXL ---- 17
3.3 JAXP ---- 19

Kapitel 4 Realisierung ---- **21**
4.1 Anforderungen an Benutzer ---- 21
4.2 Der Algorithmus ---- 22
4.3 Programmaufbau ---- 25
4.4 Beispiel zum Programmablauf ---- 29

Kapitel 5 Ausblick ---- **35**

Literatur ---- **36**

Anhang A Graph eXchange Language Document Type Definition ---- **37**

Anhang B GXL Dateien für Beispieldurchlauf ---- **41**

Kapitel 1

Einleitung

In den Zeiten von E-Commerce und Internet sind Datenbanken wichtig. Es gibt heute kaum eine Webseite, die nicht auf eine Datenbank zugreift. Der Trend zum Internet treibt Firmen dazu, ihre relationale Datenbanken in objektorientierte umzusetzen, weil objektorientierte Datenbanken mehr Vorteile als relationale bieten, unter anderem die objektorientierte Modellierung und Entwicklung. Um diese alte Datenbanken in objektorientierte Datenbanken konvertieren zu können, muss man eine Vorstellung haben, wie diese alten Datenbanken funktionieren. Dafür benutzt man einen sogenannten *Reenginering Prozess*, der häufig, wie Erfahrungen zeigen (siehe [8] und [1]), iterativ ausgeführt werden muss. Die Software heutzutage muss einige Eigenschaften bieten: Wartbarkeit, Anpassbarkeit, Erweiterbarkeit und Wiederverwendbarkeit. Durch die Benutzung von objektorientierten Datenbanken entsteht keinen Modellierungsbruch, was eine effiziente Entwicklung verspricht. Weitere Vorteile von objektorientierten Datenbanken sind: die Fähigkeit Typkonstruktoren und komplexe Objekte darzustellen, es wird mit Klassen, Methoden und Kapselung umgegangen, Vererbung, Redefinition und spätere Bindung von Klassen. Das alles erleichtert den Übergang von Modellierung zur Implementierung. Wenn man relationale Datenbanken mit objektorientierten vergleicht stellt man heraus, dass bei der objektorientierten Datenbanken ganzen Pfadausdrücke in Queries passen. Bei den relationalen Datenbanken ist man gezwungen Joins zu benutzen und das kostet Zeit (Benutzerwartezeit). Deshalb ist es so wichtig eine *Datenbankenmigration* vorzunehmen. Datenbankenmigration ist eine Überführung von einer Datenbank zu anderen. Hier wird über die Unterstützung einer Migration einer relationalen Datenbank in eine objektorientierte Datenbank gesprochen.

Ziel der Arbeit

Ziel dieser Studienarbeit ist die Entwicklung eines anwendungsunabhängigen History-Graph-Mechanismus. Ausgangspunkt ist ein bestimmter Startgraph, auf dem Transformationen einer festgelegten Form angewendet wurden. Der Startgraph und die angewendeten Transformationen bilden dann den History-Graph. Unter der Vorraussetzung daß im Startgraphen Veränderungen vorgenommen worden sind, soll mit Hilfe des History-Graphs alle Transformationen die noch anwendbar sind beibehalten werden. Alle Transformationen die nicht mehr angewendet werden können müssen rückgängig gemacht werden.

Gliederung der Arbeit

In Kapitel 2 werden Grundbegriffe erläutert und ein Beispiel aus dem Datenbank-Reengineering vorgestellt.

In Kapitel 3 werden in der Arbeit benutzten Konzepte und Werkzeuge beschrieben.

In Kapitel 4 werden Algorithmen, die für diese Arbeit relevant sind, der Programmaufbau und ein Beispiel zum Programmablauf präsentiert.

In Kapitel 5 wird einen kleinen Ausblick gegeben und auf einige Schwächen des Programms hingewiesen.

Kapitel 2

Grundlagen

2.1 Reengineering [1],[8]

Software Reengineering ist nach [7] die Analyse, von zu verändernden Softwaresysteme (*legacy software systems*), die Überarbeitung und die Implementierung dieser Überarbeitung. Alte Software muss häufig an neue Anforderungen angepasst werden. Beim Reengineering wird in der Regel nicht das ganze System verändert, sondern nur Teile davon. Dieser Vorgang besteht aus zwei Phasen: dem *Reverse-Engineering* und dem *Forward-Engineering*. Beim Reverse-Engineering wird die innere Struktur des Systems erforscht. Und somit das menschliche Verständnis vom System verbessert. In der Reverse-Engineering Phase werden also die nötige Veränderungen modelliert. In der Forward-Engineering Phase werden diese Veränderungen dann implementiert. Die Abbildung 1 illustriert den Reengeneering Prozess. Wir haben also das System, das verändert werden muss (in der Abbildung 1 `legacy system`). Es wird eine Analyse des Systems durch den Entwickler vorgenommen (`Reverse-Engineering`). Dadurch entsteht eine `abstrakte Dokumentation` des Systems. Weiter plant der Entwickler für das System benötigte Veränderungen. Das Resultat der Planungsphase ist der `Plan der benötigten Veränderungen`. Weiter werden durch den Entwickler die im Plan enthaltene Änderungen implementiert (`Forward-Engineering`). Nachdem alle Veränderungen durchgeführt worden sind, bekommt man das veränderte System (`Zielsystem`). Der Reengineering Prozess ist beendet. Eine Iteration der Phasen `Reverse-Engineering` -> `Planung` - >`Forward-Engineering`-> `Reverse-Engineering` kann entstehen, weil das System vielleicht doch nicht die gewünschten Eigenschaften aufweist. Dann beginnt einen neuer Zyklus des Reengineering Prozesses.

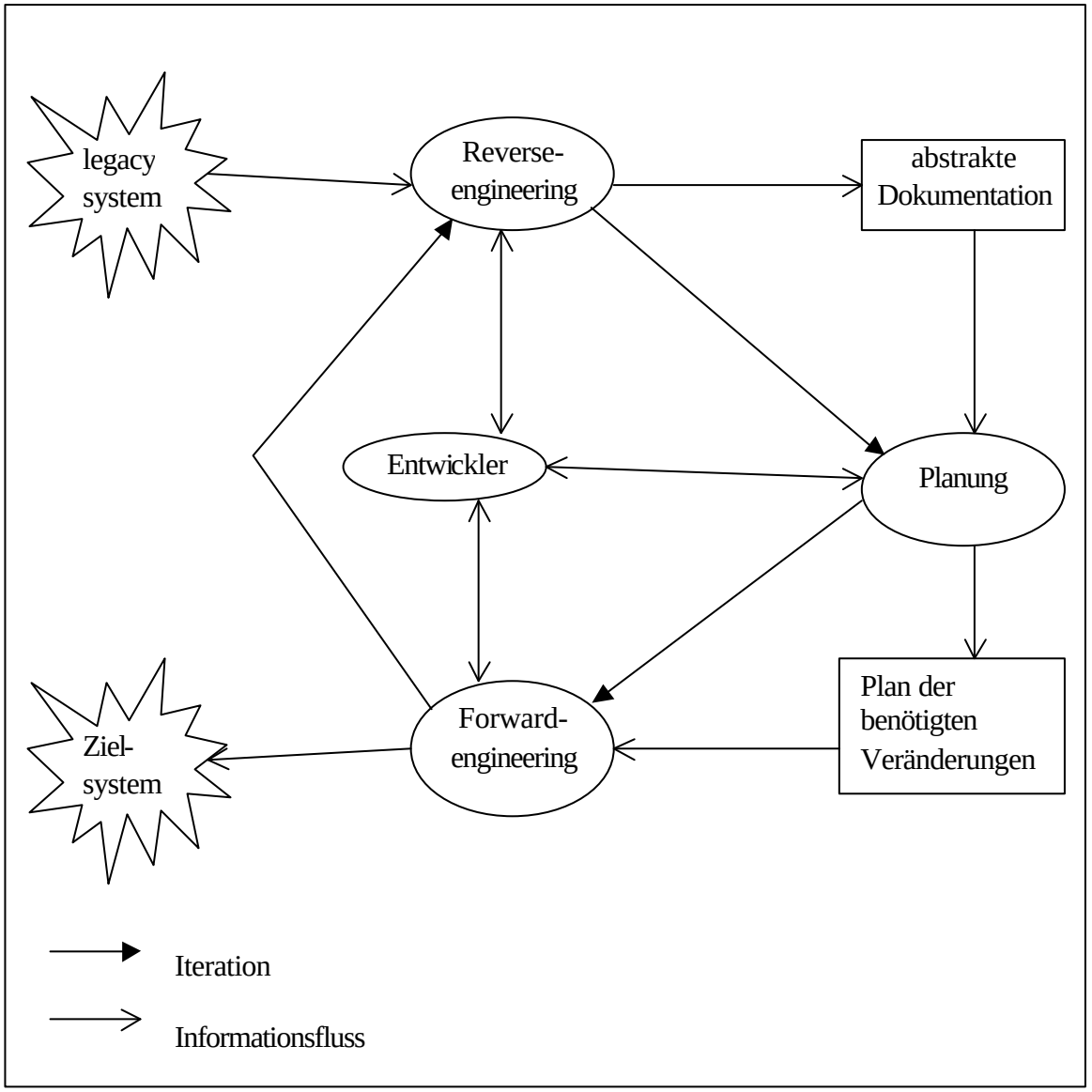


Abbildung 1: Reengineeringsprocess [8]

Der Reengineering Prozess wird unter anderem für Datenbanken benutzt und heißt Datenbanken-Reengineering. Mit Hilfe des Datenbanken-Reengineerings wird das konzeptionelle Schema einer Datenbank wiedergewonnen und verändert.

Das *konzeptionelle* Schema einer Datenbank strukturiert die (Daten-)Objekte und ihre Beziehungen untereinander. Einer der wichtigsten Schritte des Datenbankentwurfs ist die Erstellung des konzeptionellen Schemas. Die Anzahl der Entwickler, die am Entwurf einer Datenbank teilgenommen haben, ist groß. Kaum ein Entwickler, wenn er noch vor Ort ist, kann sich nach einer bestimmten Zeit noch erinnern, wieso er diese oder jene Veränderung durchgeführt hat. Sehr oft fehlt eine Dokumentation dazu, weil die Entwickler häufig unter Druck stehen. Die vorgenommenen Modifikationen beschränken sich meistens auf das *physikalische* Schema und auf den *Applikationscode*. Dies verschärft das Problem der Inkonsistenz zwischen der Dokumentation, dem

physikalischen und dem konzeptionellen Schema. Der Entwickler muss zuerst verstehen, was das System macht und wie es realisiert ist, danach kann er mit der Modifikation beginnen. Bei der Modernisierung alter Systeme wird das konzeptionelle Schema auch verwendet. Die Sprache Java bietet einen leichten Zugang zum Internet, deshalb wird sie bei vielen neuen/erweiternden Implementierungen eingesetzt. Die Realisierung einer Datenbank durch ein Client/Server-System gehört auch dazu. Dabei liegt die Datenbank auf einem Server und die Benutzer können über das Web auf sie zugreifen. Die Datenbank kann zentral verwaltet werden und muss nicht mehr in allen verschiedenen Außenstellen vorhanden sein. Somit ist sie leichter zu warten und für eine größere Anzahl von Benutzern zugänglich. Dadurch wird außerdem eine Plattformunabhängigkeit erreicht, das heißt jeder der einen Internet Zugang besitzt, kann auf die Datenbank zugreifen, wenn er dazu berechtigt ist.

Ein weiteres Anwendungsgebiet, für das konzeptionelle Schema gebraucht wird, ist die *Migration* von relationalen zu objektorientierten Datenbanken. Die Datenbankmigration besteht aus drei Schritten, der erste ist die Analyse. Sie extrahiert Informationen aus der relationalen Datenbank. Der zweite Schritt ist die Schemamigration. Sie übersetzt das relationale in das objektorientierten Schema. Der letzte ist die Applikationsmigration. Sie besteht aus der Datenmigration und der Anfragemigration. Die Datenmigration überführt die Ausprägung der Datenbank. Die Anfragemigration übersetzt die Datenbankanfragen. Legasy-Datenbanksysteme sind verbreitet und oft komplex. Das lässt das Datenbank-Reengineering immer wichtiger werden.

2.2 Verified Analysis and Reengineering of Legacy database systems using Equivalence Transformations (VARLET) [8]

Mit Hilfe von VARLET kann man unter anderem die Überführung (Migration) einer relationalen Datenbank in eine objektorientierte beziehungsweise objekt-relationale Datenbank vornehmen. Dabei wird ein Abbildungsschema verwaltet, das eine eindeutige Zuordnung zwischen den Schemata der Datenmodelle gewährleisten soll. Aus einer konkreten Datenbank wird mittels einer *`initialen Abbildung`* ein äquivalentes für VARLET `internes Datenmodell` konstruiert (siehe Abbildung 2). Dieses Schema kann dann mit Hilfe sogenannten *`Restrukturierungstransformationen`* überarbeitet werden und in die gewünschte konkrete Datenbank abgebildet werden. Es ist auch möglich, aus dem internem Datenmodell wieder in das Ausgangsdatenmodell zurück abzubilden. Die existierenden relationalen Datenbanken beinhalten nicht immer alle *semantischen* Informationen, die für Migration gebraucht werden. Das Schema ist nicht vollständig und nicht eindeutig. Es sind häufig keine Schlüssel und Fremdschlüssel zu erkennen. Deshalb muss das relationale Datenbankschema in einer *`Analyse-Phase`* angereicht werden. Die Abbildung 2 illustriert die Datenbankmigration in VARLET.

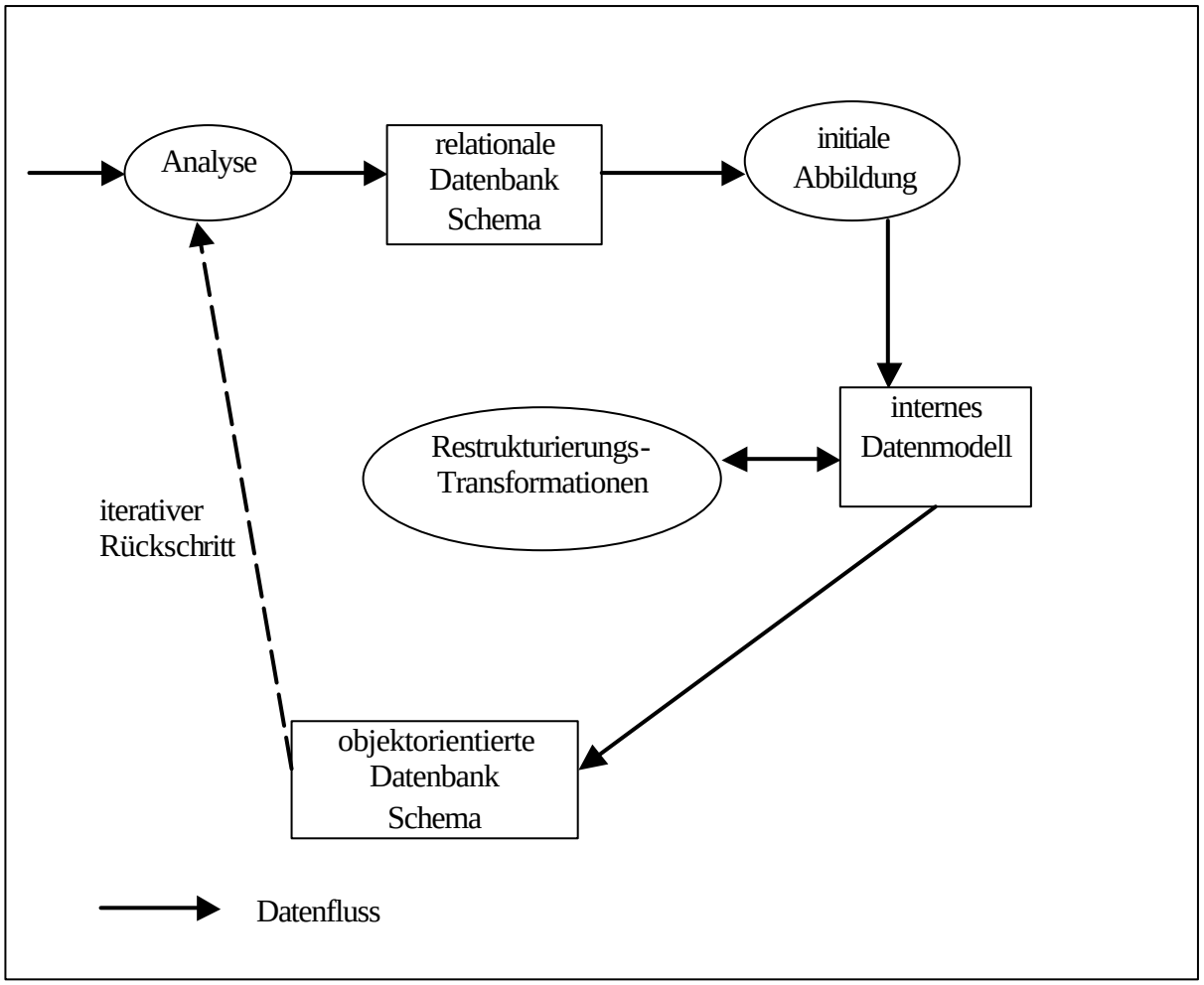


Abbildung 2:Reengineering Prozess in VARLET [1]

Nach der Analyse wird das Datenbankschema mit der *initialen Abbildung* in das internes Schema übersetzt. Danach kann das vorhandene objektorientierte Schema mit Transformationen restrukturiert werden, um die Vorteile der objektorientierten Datenbanken auszunutzen und dieses zu optimieren. Die nach der Restrukturierung gewonnene Erkenntnisse können eine wichtige Rolle in der Analyse spielen. Deshalb ist es an diesem Punkt möglich, die Analyse mit den neuen Informationen erneut zu machen. Änderungen des relationalen Schemas wirken sich auf das objektorientierte Schema aus. Wäre die initiale Abbildung nicht früher ausgeführt worden, hätte sie ein anderes Schema erzeugt. Aber dann hätten vielleicht einige Restrukturierungstransformationen nicht gemacht werden dürfen. Deshalb müssen die initialen Abbildung und die Restrukturierung wiederholt werden, aber nur auf den veränderten Schemakomponenten. Es besteht die Möglichkeit schon ausgeführten Restrukturierungstransformationen erneut ausführen, aber nur solche, die an dieser Stelle zulässig sind. Die initiale Abbildung und die Restrukturierung werden inkrementell wiederholt.

2.3 Beispiel

Um den Reengineering Prozess besser zu verstehen, wird hier ein Beispiel vorgestellt. Das Beispiel ist sehr klein gewählt, damit die Übersicht nicht verloren geht. Das Beispiel beginnt sofort mit Restrukturierungstransformationen und nicht mit der initialen Abbildung, weil in VARLET diese eins zu eins in internes Datenmodell überführt wird. Man muss sich das Beispiel so vorstellen: es gibt eine Universität, der einige Personen zugeordnet sind. Jede Person hat einen Namen und einen Geburtstag. Der Name und der Geburtstag sind von Typ char. Das alles (Universität, Personen, Personname und Persongeburtstag) wird ein Schema genannt. Natürlich bleibt ein Schema nach einer Veränderung ein Schema. Die Schemta werden als Graphen repräsentiert.

Die nachfolgende drei Abbildungen stellen das Beispiel vor. In der Abbildung 3 ist der Startgraph zu sehen, also das Schema, das angereicht werden muss. Im Beispiel ist eine Universität (im Graphen `University`) die Personen (`University_Person`) hat. Personen haben einen Namen (`Person_Name`) und einen Geburtstag (`Person_Birthday`). Der Name und der Geburtstag einer Person sind von Typ char (`University_Char`). In jedem Knoten stehende Bezeichnungen haben folgenden Sinn: eine in normal Schrift geschriebene Bezeichnung ist die interne Repräsentation des Knotens (innere Id des Knotens), die nach außen nicht bekannt ist, eine in kursiv geschriebene und geklammerte Bezeichnung ist die äußere Repräsentation des Knotens (äußere Id des Knotens). Zum Beispiel bei dem Knoten `University` ist die innere Id „University“ und die äußere Id „*University*“. Im weiterem werden die inneren Ids der Kanten und der Knoten benutzt. Die Kanten Ids sind abgekürzt, um die Abbildung übersichtlicher zu halten. Eine kursiv geschriebene Bezeichnung gibt die Semantik der Kante, eine normal geschriebene Bezeichnung ist die innere Id der Kante. Die Abkürzungen für inneren Kanten Ids sind folgendermaßen zu lesen: die klein geschriebenen Buchstaben sind die äußere Id der Kante, klein und groß geschriebenen – innere Id. Die äußere Ids sind kursiv hervorgehoben. Die Richtung einer Kante ergibt sich aus den inneren Id der Kante. Zum Beispiel die Kante zwischen `University` und `Person_Name` hat folgende innere Id: `cUPN` (ausgeschrieben - comprises_University_Person_Name). Das heißt, das die äußere Id *comprises* ist und die Kante geht von `University` nach `Person_Name`. Jetzt kommt die Übersicht, die ausgeschriebene innere Kanten Ids präsentiert.

Abkürzung	voller Name
cUPN	comprises_University_Person_Name
cUUP	comprises_University_University_Person
oUPPN	owns_University_Person_Person_Name
hdPNUC	has_domain_Person_Name_University_Char
hdPBUC	has_domain_Person_Birthday_University_Char
oUPPB	owns_University_Person_Person_Birthday
cUPB	comprises_University_Person_Birthday
oUBPB	owns_University_Birthday_Person_Birthday
cUUB	comprises_University_University_Birthday

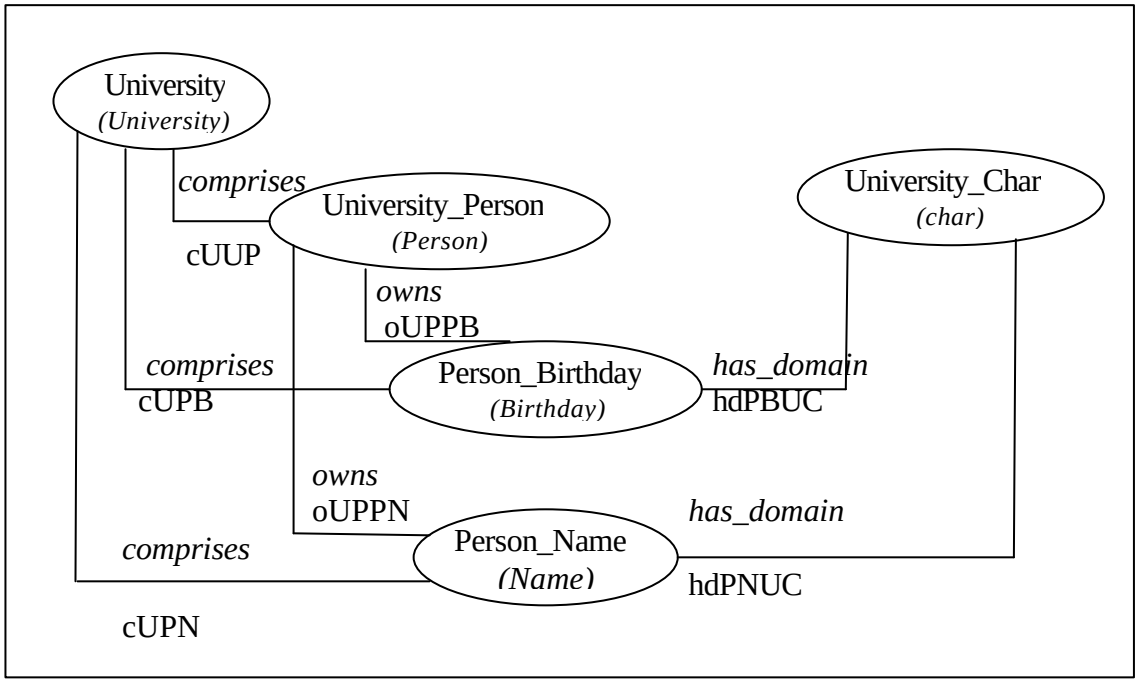


Abbildung 3: Startgraph

In der Abbildung 4 ist dargestellt, wie eine Transformation (Trafo1_SplitClass) den Knoten `University_Person` in zwei Teile zerlegt (der zweite Teil ist `University_Birthday`) und die Kanten zwischen allen Knoten anpasst (vergleiche mit Abbildung 3) werden.

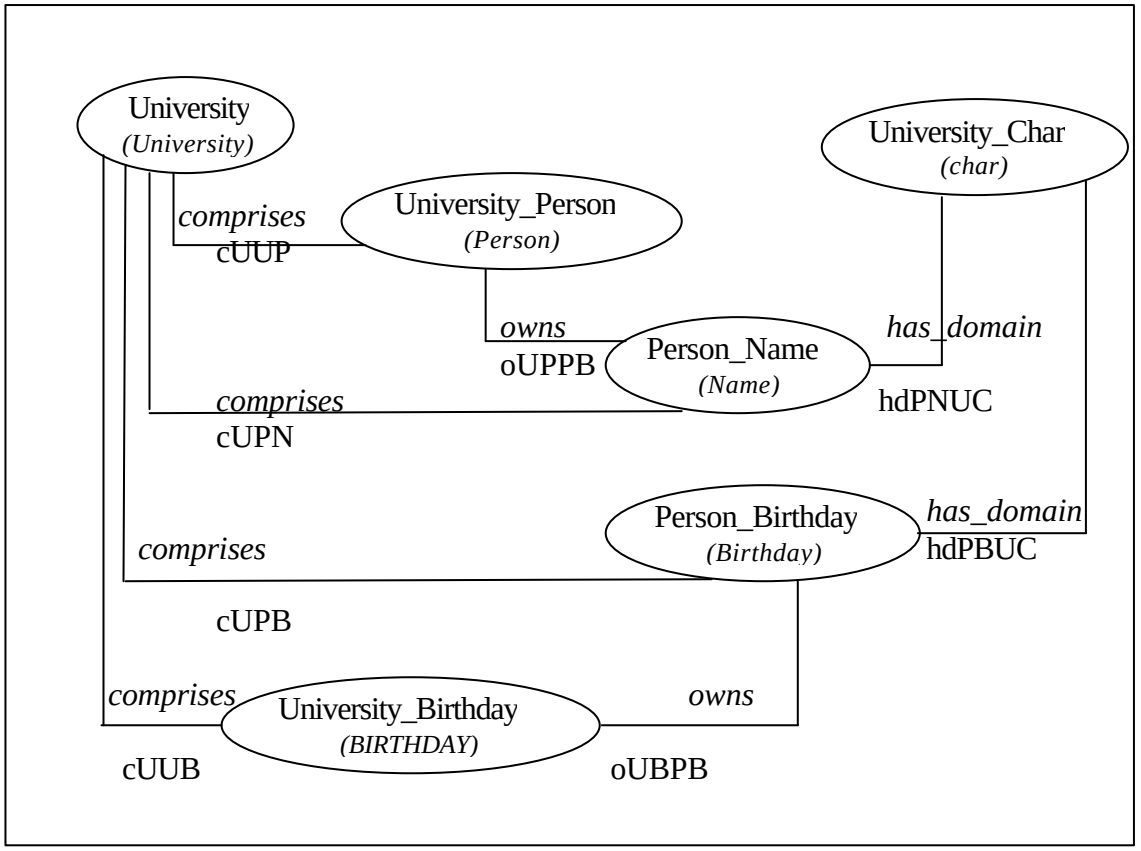


Abbildung 4: Graph nach der Transformation Trafo1_SplitClass

Die nächste Abbildung (Abbildung 5) zeigt die Wirkung einer Transformation (Trafo2_Rename). Diese Transformation ändert die äußere Id des Knoten `University` zur *Uni_Paderborn*. Die Abbildung 5 ist das angereichte Schema des Startgraphen.

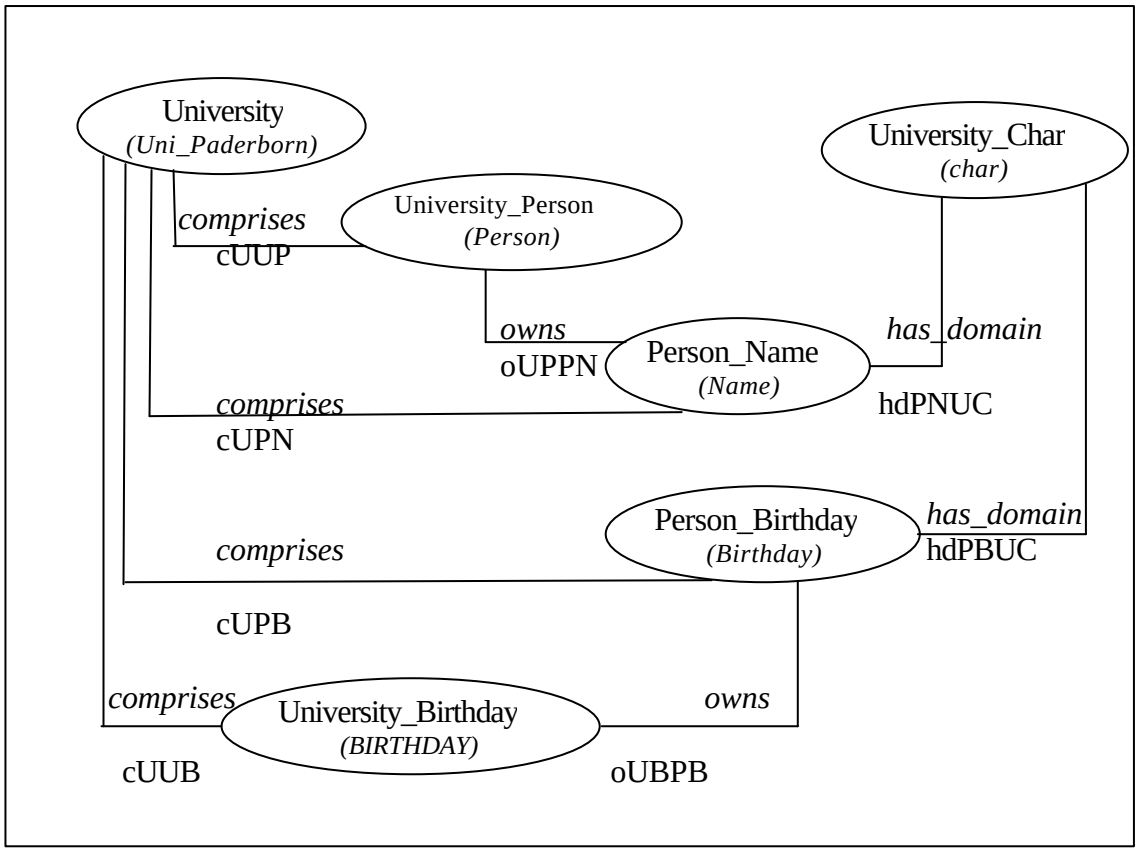


Abbildung 5 : Graph nach der Transformation Trafo2_Rename

Kapitel 3

Konzepte und Werkzeuge

Um den History- Graph-Mechanismus zu implementieren, wurden einige Konzepte und Werkzeuge gebraucht. Die Konzepte stellen das allgemeine Vorgehen vor. Die Werkzeuge sind Hilfsmitteln, womit dieses Vorgehen implementiert wird. Zu den Konzepten gehören insbesondere die Restrukturierungstransformationen oder allgemeiner Graphtransformationen (siehe 3.1). Zu den Werkzeugen gehören GXL (siehe 3.2) und JAXP (siehe 3.3).

Eins der Konzepte sind Graphtransformationen. Diese werden in VARLET (siehe Abbildung 2) für die initiale Abbildung und für die Restrukturierungstransformationen benutzt. Die initiale Abbildung in VARLET ist so realisiert, dass ein relationales Schema 1- zu -1 in ein objektorientiertes überführt wird. Es werden also Tabellen zu Klassen, Beziehungen zwischen Tabellen zu Assoziationen zwischen Klassen und Attribute zu Attribute. Die initiale Abbildung wird in dieser Arbeit nicht betrachtet, wohl aber die Restrukturierungstransformationen.

3.1 Restrukturierungstransformationen

Die *Restrukturierungstransformationen* führen die Veränderungen beim Datenbank-Forward-Reengineering aus. Die Änderungen werden danach, um die Konsistenz der Datenbank zu erhalten, in das relationales Datenbankschema übertragen. Es wird in [1] zwischen folgenden Restrukturierungstransformationen unterschieden:

- Umwandlung einer Klasse in eine Assoziation
- Umwandlung einer Assoziation in eine Klasse
- Generalisierung
- Spezialisierung
- Abspaltung einer Klasse von einer Klasse
- Verschmelzung von zwei Klasse zu einer Klasse
- Umwandlung einer Assoziation in eine Aggregation
- Umwandlung einer Aggregation in eine Assoziation
- Verschiebung von Attributen in einer Vererbungsstruktur
- Veränderung der Kardinalität der Traversierungspfade
- Erzeugen einer Klasse, eines Attributs oder einer Assoziation
- Umbenennung bzw. Typänderung
- Löschen

Nicht alle Restrukturierungstransformationen verändern so das objektorientierte Datenbankschema, dass die Änderungen in das relationale übertragen werden müssen, zum Beispiel *Rename*. Alle Transformationen mit ihren Parameter werden gespeichert. Ablauf einer Transformation sieht folgendermaßen aus: 1) die innere Ids der Knoten (zum Beispiel N1) (Kante E1), die als Eingabe für eine Transformation (zum Beispiel T1) bestimmt sind, werden in N1_T1 (E1_T1) umbenannt; 2) der Transformationsknoten mit allen *in*- und *out*-Kanten und mit *in*- und *out*-Kanten verbundenen Knoten (Kanten) werden in den Graphen eingefügt.

Da die initiale Abbildung und die Restrukturierung inkrementell wiederholt werden, ist es sinnvoll dem Benutzer eine Übersicht aller schon ausgeführten Transformationen und den Restgraphen anzubieten. Damit wird bessere Übersichtlichkeit der Benutzerarbeit erreicht. In VARLET ist dies „hart" codiert, es gibt also keine Möglichkeit das Konzept ohne VARLET zu benutzen. Das in dieser Arbeit entwickelte Werkzeug erlaubt es, das History Graph Konzept unabhängig von VARLET zu benutzen. Um das Werkzeug noch allgemeiner zu machen, ist für die textuelle Graphdarstellung GXL benutzt worden.

3.2 GXL - Graph eXchange Language

Um Graphen bequemer zu bearbeiten, wird im Programm die GXL benutzt. GXL bietet die Möglichkeit Graphen als Text darzustellen was die Arbeit mit Graphen erleichtert.

Es gibt eine große Anzahl von Tools und Programmen, die die Arbeit mit Graphen ermöglichen. Ein Programm kann bestimmte Aufgaben viel besser erledigen als ein anderes. In [6] werden drei Typen von solchen Werkzeuge aufgeführt: 1) *extractors*, diese sammeln Information über software (Werkzeuge); 2) *manipulators*, diese können Form den Daten verändern und 3) *analyzers*, diese bestimmen und zeigen die Information, die sie aus Daten bekommen, an. Um Problemen mit Verträglichkeit (Englisch: interoperability) zwischen Werkzeugen oder Programmen zu entkommen, muss man einen gemeinsamen Format haben. Deshalb haben sich Entwickler zusammengetan und GXL –Standart entworfen. Graph eXchange Language (GXL) ist in einer Zusammenarbeit mehrerer Teams entstanden. GXL ist eine XML (Extensible Markup Language) Sprache, die ihre eigene DTD (Document Type Definition) mitbringt. Sie modelliert einen gerichteten Graphen, der Attribute und Typen hat. Die Datei gxl.dtd (siehe Anhang A) wurde als GXL –Standard klassifiziert. Alle beteiligten Teams kann man unter [4] finden. GXL ist ein „freies" Format. Man braucht für die Benutzung keine Lizenz: weder eine kommerzielle noch eine nicht-kommerzielle. Es muss legendlich auf die Autoren hingewiesen werden.

Weiter wird der GXL –Standart, also die gxl.dtd kurz vorgestellt. Eine ausführliche Beschreibung findet man unter [4]. Die komplette gxl.dtd ist als Anhang A zu finden. GXL unterstützt folgende

Standartdatentypen: *bool*, *int*, *float*, *string*, *seq*, *set* und *bag*. Es gibt auch Typ tup, der Datentupel repräsentiert. Der Typ enum (enumeration) ist auch vertreten. In einer gxl –Datei können mehrere Graphen enthalten sein. Es gibt auch die Möglichkeit einige Teile der Datei außerhalb der Datei zu platzieren. Das erreicht man mit dem Attribut des `gxl` –Knotens xmlns vom Typ xlink. Der `graph` –Knoten kann einen Typ (optional), Attribute, Knoten, Kanten und Relationen als Kinder enthalten. Die Relationen repräsentieren n-äre Beziehungen. Alle Elemente müssen eine eindeutige Id aufweisen. Außerdem kann ein `graph` –Knoten eine Role als Attribut haben. Kanten, Knoten und Relationen können als Kinder auch Graphen aufweisen. Außer von Autoren vorgegebenen Attributen können zu allen Elementen als Kinder beliebige Attribute hinzugefügt werden. Die Abbildung 6 zeigt eine nicht vollständige graphische Übersicht der gxl.dtd. Der vollständige Baum wäre unübersichtlich.

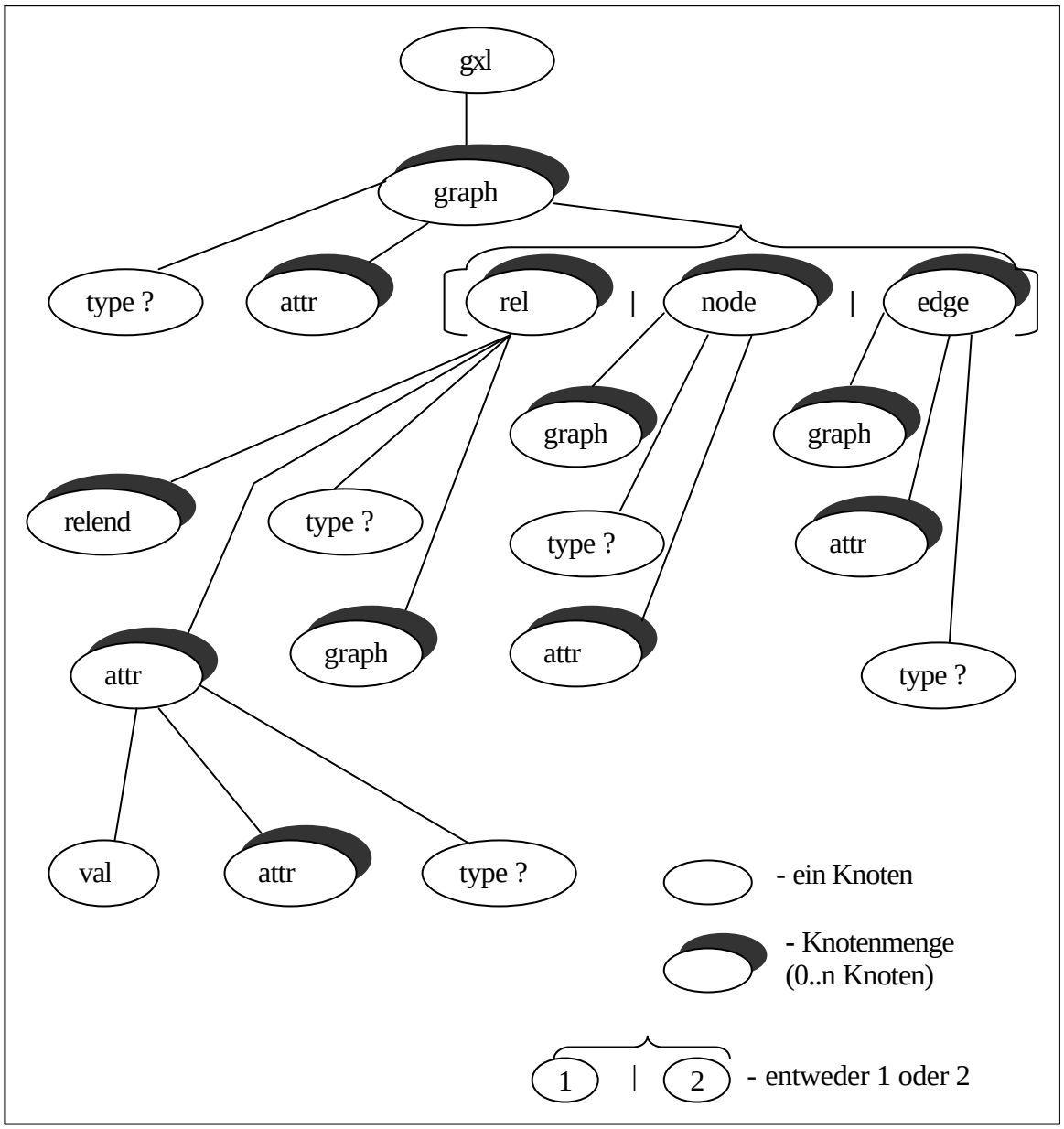


Abbildung 6: nicht vollständiger Baum der gxl.dtd

Hier ist `gxl`-Knoten als Document-Knoten zu sehen, das die ganze gxl-Datei repräsentiert. Es gibt nur einen `gxl`-Knoten pro Datei. Jede gxl-Datei kann mehrere Graphen beinhalten. Jeder Graphen kann folgenden Elemente enthalten:

- optionalen Knoten `type` („?“ heißt in XML optional), der den Typ des Graphen angibt
- mehrere `attr`-Knoten. Der Knoten ist für Null bis mehrere Attribute des Graphen vorgesehen.
- beliebige Kombination von `rel`-, `node`- oder `edge`-Knoten, wobei `rel`-Knoten eine n-äre Beziehung, `node`-Knoten einen Graphknoten und `edge`-Knoten eine Graphkante darstellen.

`Edge`- und `node`-Knoten sind gleich aufgebaut. Sie können folgenden drei Elemente beinhalten:

- einen Graphen
- optionalen Knoten `type`, der den Typ des Knotens angibt
- mehrere `attr`-Knoten. Der Knoten ist für Null bis mehrere Attribute des Knotens vorgesehen.

Einen `rel`-Knoten sieht fast genau so aus, wie `edge`- und `node`-Knoten. Es kommt lediglich einen `relend`-Knoten dazu. Alle Attribut-Knoten (`attr`) sind identisch aufgebaut. Sie bestehen aus:

- optionalen Knoten `type`, der den Typ des Knotens angibt
- mehrere `attr`-Knoten. Der Knoten ist für Null bis mehrere Attribute des Knotens vorgesehen
- `val`-Knoten. Der Knoten ist für Inhalt des `attr`-Knoten reserviert. Jeder `attr`-Knoten kann nur einen `val`-Knoten aufweisen.

3.3 JAXP

Java™ APIs for XML Processing (JAXP) ist von SUN implementierter XML Parser. JAXP ermöglicht Arbeit mit dem sogenannten DOM (Document Object Model) und einigen anderen XML Darstellungen (siehe [5]). In dieser Arbeit wurde nur DOM benutzt. DOM ist eine Möglichkeit XML als einen Baum darzustellen. Dies erleichtert die Arbeit mit XML. Man muss legendlich eine XML Datei als einen Baum durchzulaufen und in der XML Datei gespeicherte Inhalte auszulesen, zu speichern oder zu verändern. Die Baumstruktur verbraucht aber viel CPU und Speicher. Die DOM – Architektur ist in W3C –Gruppe entstanden [3]. Die Abbildung 7 zeigt DOM API.

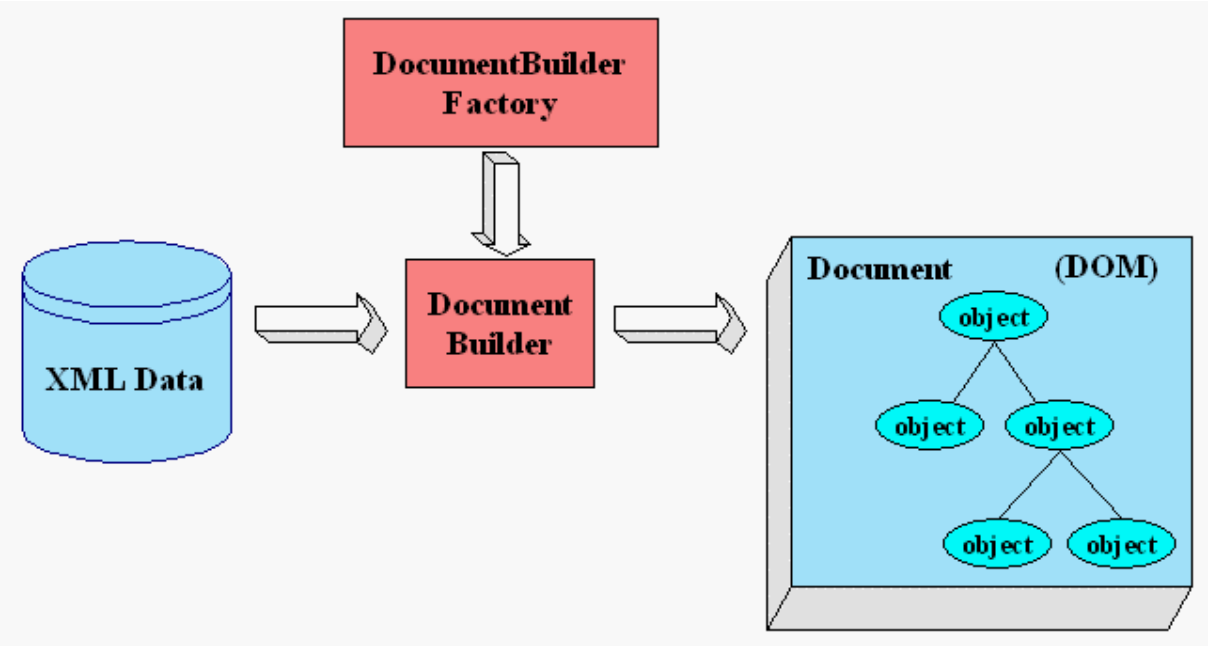


Abbildung 7: Document Object Model (DOM) APIs [5]

JAXP parset und transformiert XML Dokumente. Um JAXP zu benutzen, benötigt man den JDK 1.1.8 oder höher. Es muss keinen Treiber angebunden werden. Alles, was für die Arbeit nötig ist, liefert SUN mit. Die Wahl fiel auf JAXP, weil JAXP von SUN kommt und deshalb sehr gut mit der Programmierungssprache Java verträglich ist. Jetzt kommt eine kleine JAXP –Beschreibung. Es wird nur DOM behandelt, weil es für Arbeit relevant ist.

Der Kern der API ist in dem Package `javax.xml.parsers` definiert. Das Package hat zwei Factoryklassen : *SAXParserFactory* und *DocumentBuilderFactory*. Sie liefern einen SAXParser bzw. einen DocumentBuilder. Mit Hilfe eines DocumentBuilder's bekommt man ein DOM –Document, das einen DOM –Baums Document –Knoten repräsentiert. Die Factoryklasse gibt die Möglichkeit, Implementierung einen anderen XML-Parseranbieter zu benutzen, ohne der Code des Programms zu ändern. JAXP beinhaltet folgende Packages: `javax.xml.parsers, org.w3c.dom, org.xml.sax` und `javax.xml.transform`. Über `javax.xml.parsers` wurden schon oben geschrieben. `org.w3c.dom` implementiert die DOM APIs. `org.xml.sax` implementiert die SAX APIs. `javax.xml.transform` implementiert XSLT APIs, womit man XML zu andern Formaten (HTML und andere) konvertieren kann.

Kapitel 4

Realisierung

Für die erfolgreiche Arbeit mit dem Werkzeug müssen einige Einschränkungen des Werkzeugs vorgestellt werden. Die Benutzer müssen bestimmte Regeln beachten und eine Vorstellung über innere Aufbau des Tools haben, um gewünschte Resultate zu erzielen.

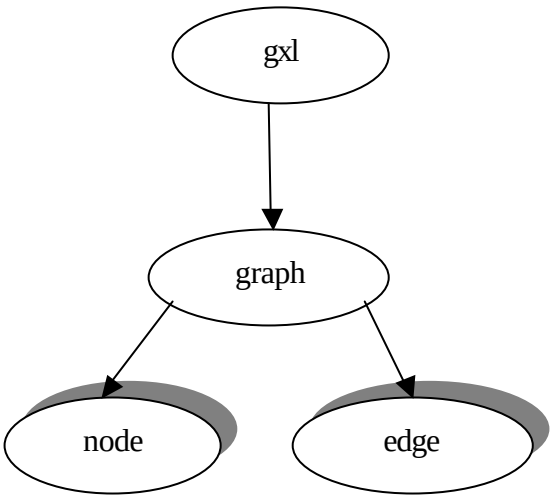
4.1 Anforderungen an Benutzer

An dieser Stelle werden bestimmte Regel beschrieben, die beim Benutzen des Werkzeugs berücksichtigen muss.

1. Die Knoten und Kanten müssen eindeutige inneren IDs haben. Das ist sowohl für den Algorithmus als auch für GXL eine Vorraussetzung.
2. Die Dateien müssen das korrekte GXL-Format einhalten, damit man die Dateien in DOM Dokumente umwandeln kann. Die ersten zwei Zeilen jeder Datei müssen wie unten aufgeführt aussehen:

   ```
   <?xml version="1.0"?>
   <!DOCTYPE gxl SYSTEM "gxl.dtd">.
   ```

   Danach kommt eine Graphbeschreibung. (siehe Anforderung 3)
3. Der gxl- Knoten des Graphen muss abweichend von der klassischen Vorstellung folgende Struktur aufweisen:

   Unter klassischen Graphdarstellung ist hier eine rekursive Definition des Graphen gemeint.
4. Es kann nur einen Graphen pro Klasse geben, weil es immer auf dem Graphen gearbeitet wird. Damit wird die Konsistenzerhaltung gewährleistet.

5. Bei der Eingabe einer Transformation müssen alle Knoten und Kanten, zu denen eine *out*-Kante führt, vorhanden sein.
6. Es muss zuerst den Graphen erzeugt bzw. hinzugefügt werden, danach Transformationen und Veränderungen.
7. Vor dem Aufrufen der Methode createHistory muss mindestens ein Graph vorhanden sein.
8. Die Id des Transformationsknoten muss mit „Trafo“ anfangen, die Id der *in* –Kante und die Id der *out* –Kante mit „in“, beziehungsweise mit „out“ anfangen, damit man den Knoten beziehungsweise die Kanten findet.

4.2 Der Algorithmus.

Bevor der Programmaufbau beschrieben wird, wird ein grober Überblick über den Algorithmus, den das Programm implementiert, gegeben.

Das zu verändernde Schema wird mit Hilfe der Restrukturierungstransformationen angereicht. In weiterem wird zur Vereinfachung das Wort Transformation statt Restrukturierungstransformation verwendet. Beim Einfügen einer Transformation (z.B. T1) wird der Knoten (die Kante), den die Transformation verändert, in IdDesKnotens_T1(IdDerKante_T1) umbenannt und als Eingangsparametern für Transformation T1 gespeichert. Die Knoten (die Kanten), die mit der von T1 out- Kante verbunden sind, werden einfach hinzugefügt. Es wird keine der Bedingungen, die gefordert wurden verletzt (siehe 4.1). Die Abbildung 8 illustriert, den oben beschriebenen Vorgang.

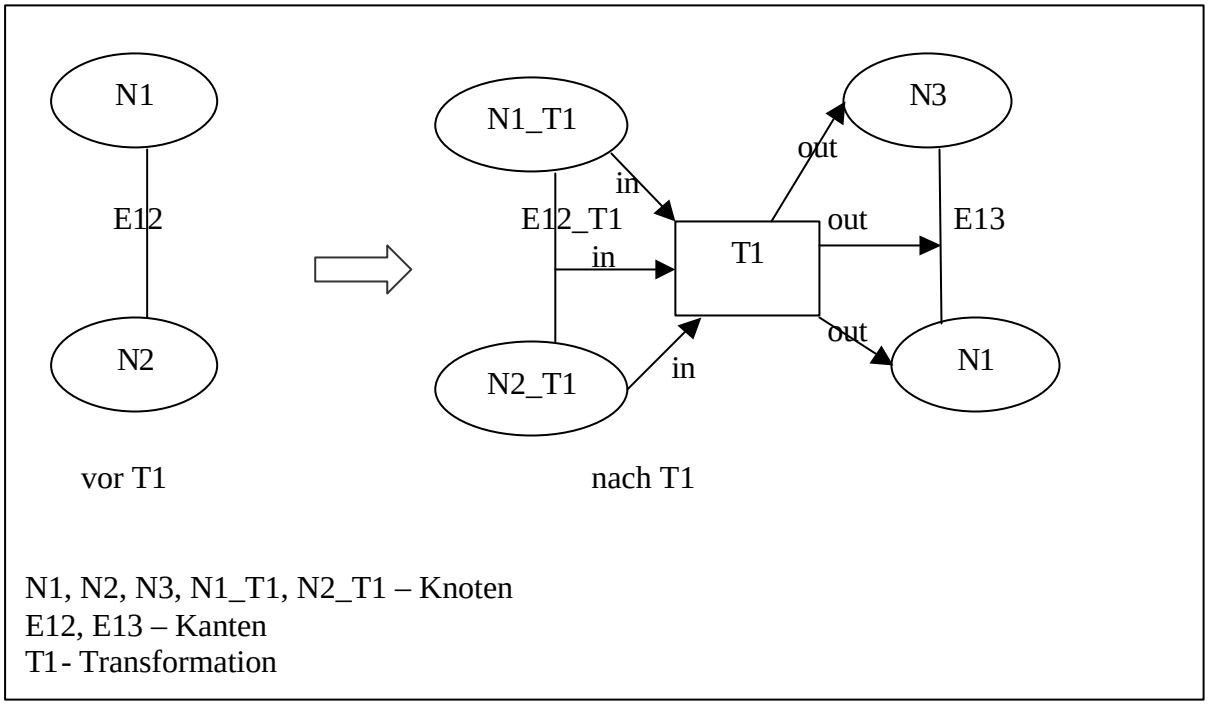


Abbildung 8: Einfügen einer Transformation

Wenn eine Änderung zum Beispiel beim dem Knoten N2_T1 vorgenommen wurde (siehe Abbildung 9 der Fall nach T1), müssen die Transformationen, die N2_T1 als Eingangsparameter haben (in der Abbildung 9 T1), zurückgesetzt und noch ein Mal ausgeführt werden. Dafür müssen die Knoten (Kanten), die von dieser Transformation (T1) erzeugt oder verändert wurden, zum Löschen markiert (siehe Abbildung 9) und der Zustand vor der Transformation wiederhergestellt (siehe Abbildung 10).

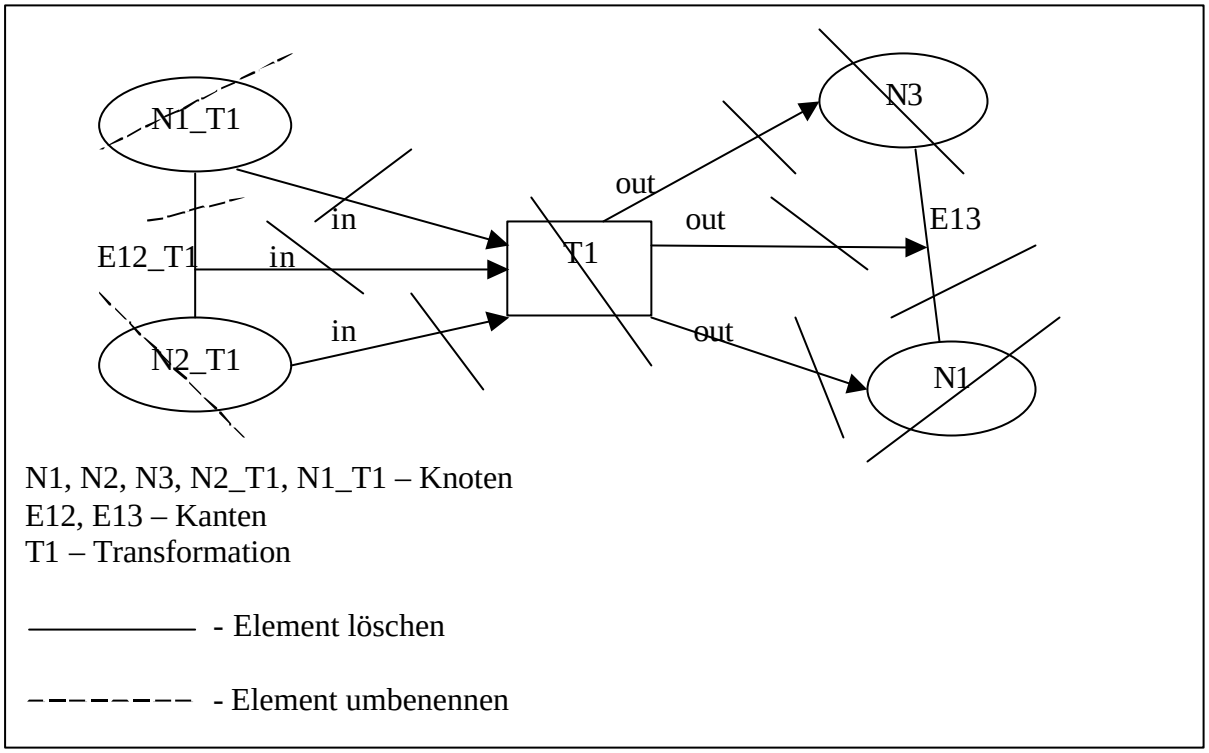


Abbildung 9: Markierung den zu löschenden und umzubenennenden Elemente

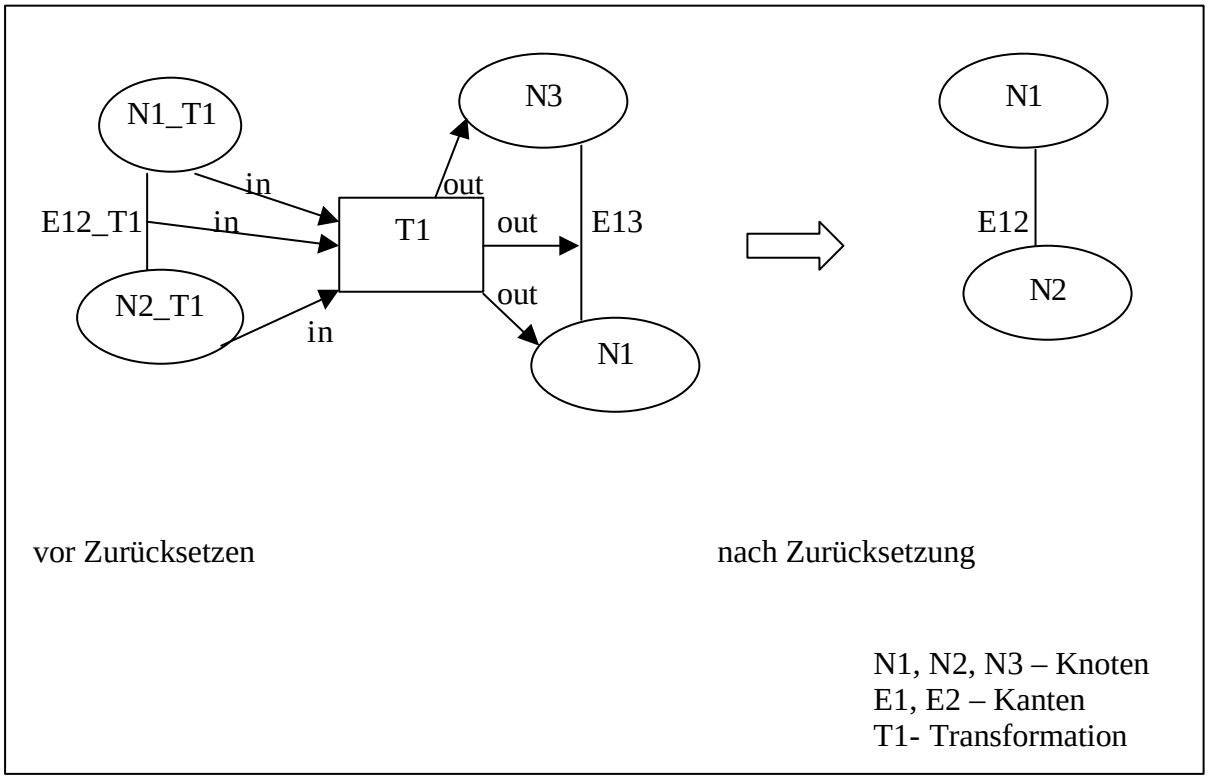


Abbildung 10: Zurücksetzen einer Transformation

Bei Zurücksetzen einer von den mehreren Transformationen werden alle von der zurückgesetzten Transformation erzeugten und veränderten Kanten (Knoten) und die Transformationen, die auf diesen

Knoten (Kanten) aufbauen, gelöscht (siehe Abbildung 11). Die Knoten (Kanten), die von der Transaktion unabhängig (von der Transformation nicht verändert oder nicht erzeugt) sind, werden nicht berührt. Die Abbildung ist folgendermaßen zu verstehen: der Startgraphen bestand aus `N1`, `N2`, `N3`, `E12` und `E13`. Danach wurden `T1`, `T2`, `T3`, `T4` ausgeführt. Naher wurde der Knoten `N2` verändert, das Löschen und Umbenennung der kenngezeichneten Knoten und Kanten verursachte.

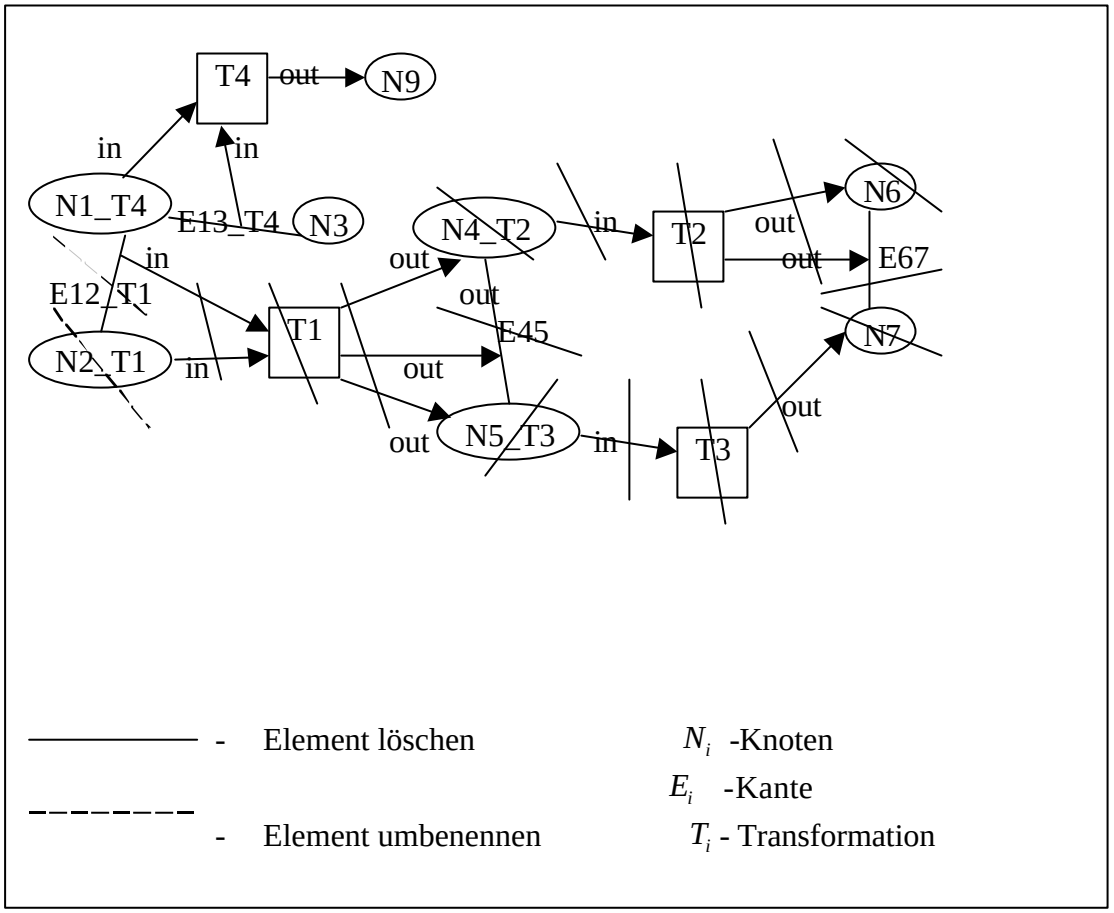


Abbildung 11: kaskadiertes Zurücksetzen einer Transformation (hier T1)

4.3 Programmaufbau

Das Programm (Tool) besteht aus zwei Klassen: GXLHistory und Main. Die Main -Klasse definiert die Benutzerschnittstelle, also hier wird die GXLHistory –Klasse erzeugt und benutzt. Die GXLHistory -Klasse leistet die eigentliche Arbeit.

Die Klasse GXLHistory stellt für den Benutzer vier Methoden zur Verfügung.

- *addGraph*. Die Methode addGraph erzeugt einen DOM-Baum aus der übergebenen GXL-Datei. Es ist noch ein Mal zu betonen, dass es nur einen Graphen pro Klasse (GXLHistory – Klasse) existieren darf. Mit andern Worten der Graph ist ein *Singleton*. Zu dem Graphen werden immer Subgraphen addiert oder aus dem Graphen gelöscht.

- *addTransformation.* Die Methode addTransformation fügt eine Datei mit einer Transformation zu den Graphen hinzu. Wenn man mehrere Transformationen hinzufügen möchte, muss man addTransformation mehrmals anwenden.
- *addChange.* Die Methode addChange fügt eine Datei mit Veränderungen zu den Graphen hinzu. In der Datei können mehrere Veränderungen zusammengefasst werden.
- *createHistory*. Die Methode createHistory stößt den History-Mechanismus an. Sie sammelt zuerst die zu löschenden Knoten und Kanten. Danach werden die markierten Knoten und Kanten gelöscht. Naher werden die Hilfsknoten und -Kanten zur *report* hinzugefügt (siehe unten). Dann wird jeweils in eine Datei der übriggebliebenen Graphen und die Liste *report* geschrieben.

Die Abbildung 12 zeigt den Ablauf des Programms.

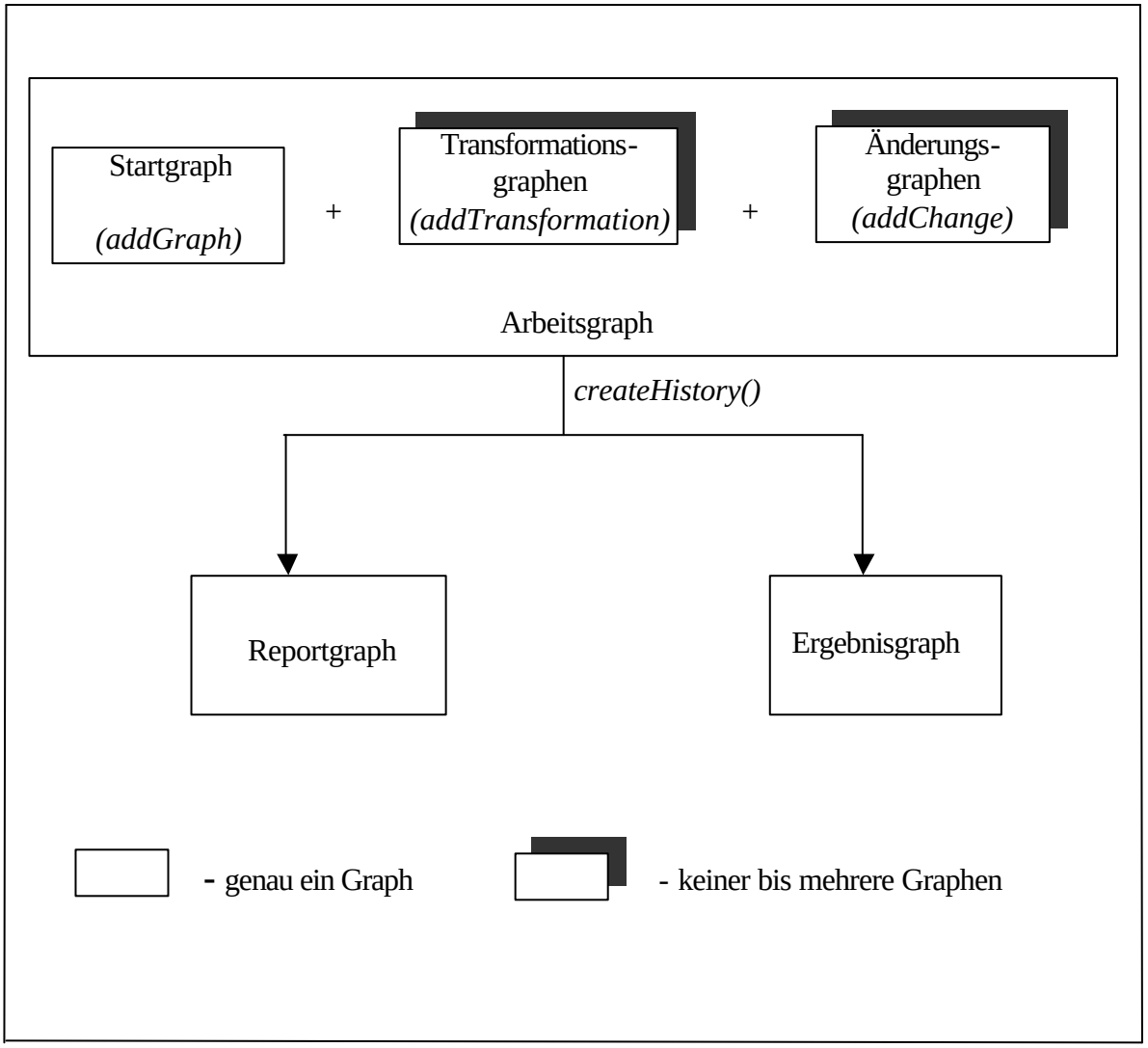


Abbildung 12: Programmablauf

In dem `Reportgraphen` werden alle gelöschten Knoten und Kanten und die Knoten und Kanten, von denen eine *in* –Kante ausgeht gehalten. Der `Ergebnisgraph` ist ein Graph, der die restlichen Knoten und Kanten beinhaltet, also die Knoten und Kanten, die nach dem Zurücksetzten einer oder mehreren Transformationen übriggeblieben sind.

Die GXLHistory –Klasse hat vier Klassenvariablen von Java-Typ LinkedList:

- *report*. In der Liste report werden die gelöschten Knoten/Kanten aufbewahrt. Bei jeder Veränderung müssen einige Knoten und Kanten gelöscht werden. Diese werden in report gespeichert. Diese Liste wird vor jeder neuen Veränderung geleert.
- *stamps*. Die Liste stamps beinhaltet alle aktive Transaktionen. Wird eine Transaktion zum Graphen dazugetan, wird sie auch zu der Liste stamps angehängt. Wird eine Transaktion zurückgesetzt, wird sie aus der Liste stamps gelöscht.

- *changes*. Liste changes speichert, wie der Name schon verrät, alle Veränderungen, die vor der Methode createHistory hinzugefügt wurden.
- *toRename*. In der Liste toRename werden alle die umzubenannten Knoten/Kanten eingesammelt. Die Liste wird zweierlei verwendet: beim Einfügen einer Transformation und beim Bearbeitung der Veränderungen.

Es gibt in der Klasse eine Klassenvariable von Java –Typ Hashtable *newOldIds*. In der Hashtable werden als Schlüssel die neue Id und als Wert die zugehörige alte Id abgelegt. Das passiert nach jeder hinzugefügten Transformation. Die Hashtable ist nötig, um beim Zurücksetzen einer Transformation alte Knoten/Kanten –Namen wiederherzustellen.

In der Klasse GXLHistory gibt es Methoden, die für den Benutzer verbogen bleiben.
Die Methode *writeGXLFile()* schreibt den übriggebliebenen Graphen in eine Datei mit dem Namen *nameOfOutputFile*. Der Graph wird einfach durchgelaufen, die Knoten und Kanten werden geschrieben.

Die Methode *writeResultFile()* leistet dasselbe. Der einziger Unterschied besteht darin, dass die Methode eine Liste durchläuft, nicht einen Graphen. Mit der Methode werden die gelöschten Knoten und Kanten geschrieben.

Die Methode *makeDOMTree()* erzeugt aus einer Datei *content* die DOM –Architektur. Derselbe Effekt erreicht man mit einem XML –Parser für Java. Die Methode unterstützt legendlich den Programmier bei der Benutzung des Parsers.

Die Methode *getGraphNode()* gibt den Graph –Knoten des Graphen zurück (s. 3.1 3).

Die Methode *getNodeWithId()* gibt einen Knoten oder eine Kanten mit bestimmten Id zurück.

Die Methode *removeNamedNode* löscht aus dem Graphen *graph* einen bestimmten Knoten oder eine bestimmte Kante.

Die Methode *renameNamedNode()* ändert Knoten/Kanten –Ids um.

Die Methode *getChangedAfter()* gibt eine Transformation zurück, die einen Knoten oder eine Kanten nach Transformation mit Id *trafo* geändert hat.

Die Methode *addHelpNodesToReport()* fügt die von *in* –Kanten ausgehende Knoten und Kanten. Dies ist nötig, um dem Benutzer eine übersichtlichere Datei der gelöschten Knoten und Kanten zu bieten.

Es gibt in der Klasse eine Methode Namens *updateNodes()*. Sie fügt den veränderten Knoten oder die veränderten Kanten zu dem Graphen hinzu. Wenn das Zurücksetzen einer Transformation mit Hilfe der oben beschriebenen Methoden erfolgen würde, würde der Algorithmus nicht korrekt implementiert. Es würden zum Beispiel in der Abbildung 11: die Kanten und Knoten `E45`, `E67`, `E78` und `N8` nicht gelöscht, was aber dem Algorithmus widerspräche. Die Methode *addNotWithInBoundedToReport()* fügt solche Knoten und Kanten zur den gelöschten Knoten und Kanten hinzu.

Weiter werden Methoden wie die *pickToDeleted()* beschrieben. Die Methode *pickToDeleted()* sammelt die Knoten, die gelöscht werden müssen in der Liste *report*. Es werden alle Veränderungen durchgelaufen und für alle Veränderungen dasselbe gemacht. Es werden zunächst die entsprechende Knoten und Kanten mit *undoRename()* umbenannt. Danach wird die Transformation, die mit geänderten Knoten oder mit geänderten Kante verbunden ist gefunden. Danach werden alle *in* –Kanten mittels *saveEdgesToTrafo()* gespeichert. Naher werden alle Knoten und Kanten laut Algorithmus mittels *saveOtherNodes()* gefunden.

Die weitere Methode, die hier vorgestellt wird heißt *undoRename()*. Es werden zuerst alle Knoten und Kanten, von denen eine *in* –Kanten ausgeht in der Liste *toRename* eingesammelt. Für jede solche Kante oder jeden solchen Knoten wird geschaut, ob die Transformation, die ihn oder sie geändert hat die letzte ist, die ihn oder sie geändert hat. Wenn es der Fall sein sollte, wird diese Kante oder dieser Knoten gelöscht und die Kanten mit den Namen *IdDerKante_IdDerTransformationsknoten* oder der Knoten mit den Namen *IdDesKnoten_IdDerTransformationsknoten* in *IdDerKante* bzw. *IdDesKnoten* umbenannt. Wenn es nicht der Fall sein sollte, wird die Transformation mittels *getChangedAfter()*, die zweitletzte den Knoten aus *toRename* geändert hat. Danach wird der Knoten oder die Kante mit den Namen *IdDesKnoten_IdDerLetztenTrafo* bzw. *IdDerKante_IdDerLetztenTrafo* in *IdDesKnoten_IdDerVorletztenTrafo* bzw. *IdDerKante_IdDerVorletztenTrafo* umbenannt. Vorher wird aber *IdDesKnoten_IdDerVorletztenTrafo* bzw. *IdDerKante_IdDerVorletztenTrafo* gelöscht.

4.4 Beispiel zum Programmablauf

An dieser Stelle wird das Beispiel aus Kapitel 3 (Abbildungen 3,4,5) aufgegriffen. Der Beispielablauf ist folgender:

1) erzeuge Startgraphen (Abbildung 3);
2) füge 2 Transformationen (`Trafo1_SplitClass` und `Trafo2_Rename`) hinzu (Abbildungen 4,5);
3) bekomme einen veränderten Knoten (innere Id `University_Person`);
4) setze die betroffenen Transformationen zurück.
5) Schreibe den Reportgraphen und Ergebnisgraphen in Dateien aus.

In folgendem Abschnitt werden einzelne Schritte detaillierter beschrieben.

1) Es wird einfach der Graph aus Abbildung 3 in DOM-Baum umgewandelt mit Hilfe der addGraph() Methode.
2) Es werden Transformationen `Trafo1_SpitClass` und `Trafo2_Rename` in DOM-Baum übersetzt und zu dem Graphen hinzugefügt. Dabei werden betroffene Kante `owns_University_Person_Person_Birthday` nach `owns_University_Person_Person_Birthday_Trafo1_SplitClass` und Knoten `University_Person` und `Person_Birthday` nach `University_Person_Trafo1_SplitClass` beziehungsweise nach `Person_Birthday_Trafo1_SplitClass für die Trafo1_SplittClass` und Knoten `University` nach `University_ Trafo2_Rename` für `Trafo2_Rename` umbenannt. Die Abbildung 13 zeigt den Graphen nach zwei den Transformationen.

   Die Liste stamps hat jetzt zwei Elementen: `Trafo1_SplitClass` und `Trafo2_Rename`.

   Die Liste toRename wird zweimal gefüllt (mit `owns_University_Person_Person_Birthday`, `University_Person`, `Person_Birthday und mit University`) und einmal geleert.

   Die Hashtable hat vier Ids (`owns_University_Person_Person_Birthday`, `University_Person, Person_Birthday`, `University`).
3) Es wird ein DOM Baum aus dem Veränderungsgraphen gemacht, das nur aus `University_Person` besteht.

4) Jetzt wird geschaut, welche der Transformationen als Eingabeparameter den Knoten `University_Person` hat. Das ist die Transformation `Trafo1_SplitClass`. Sie muss also zurückgesetzt werden. Dafür wird sie aus der Liste stamps gelöscht, die Knoten (`University_Person` und `Person_Birthday`), von Trafo1 erzeugten Knoten und Kanten, Transformationsknoten, alle in- und out-Knoten, die zu Trafo1 beziehungsweise von Trafo1 führen und Kante (`owns_University_Person_Person_Birthday`) werden gelöscht. Die in der Luft hängenden zwei Knoten und eine Kante werden den Platz den gelöschten den Knoten und der Kante einnehmen. Die gelöschten Knoten und Kante werden in report festgehalten.
5) Es werden der Reportgraphen und der Ergebnisgraphen in Dateien geschrieben. Der Reportgraph beinhaltet alle von Trafo1 erzeugten und veränderten Knoten und Kanten. Der Ergebnisgraph hat alle übriggebliebenen Knoten und Kanten. Siehe Abbildungen 14 und 15 und Anhang B.

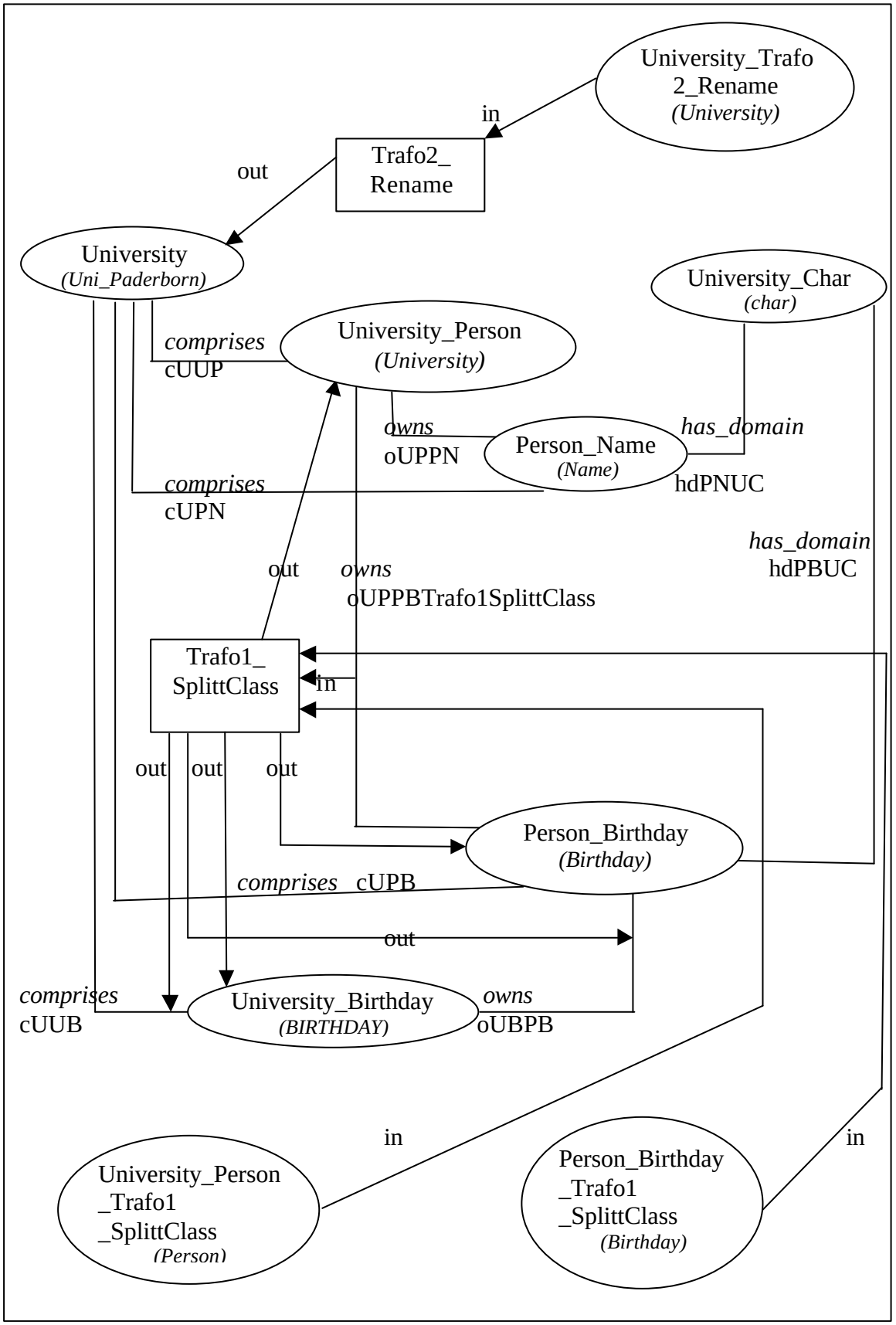


Abbildung 13:Arbeitsgraph

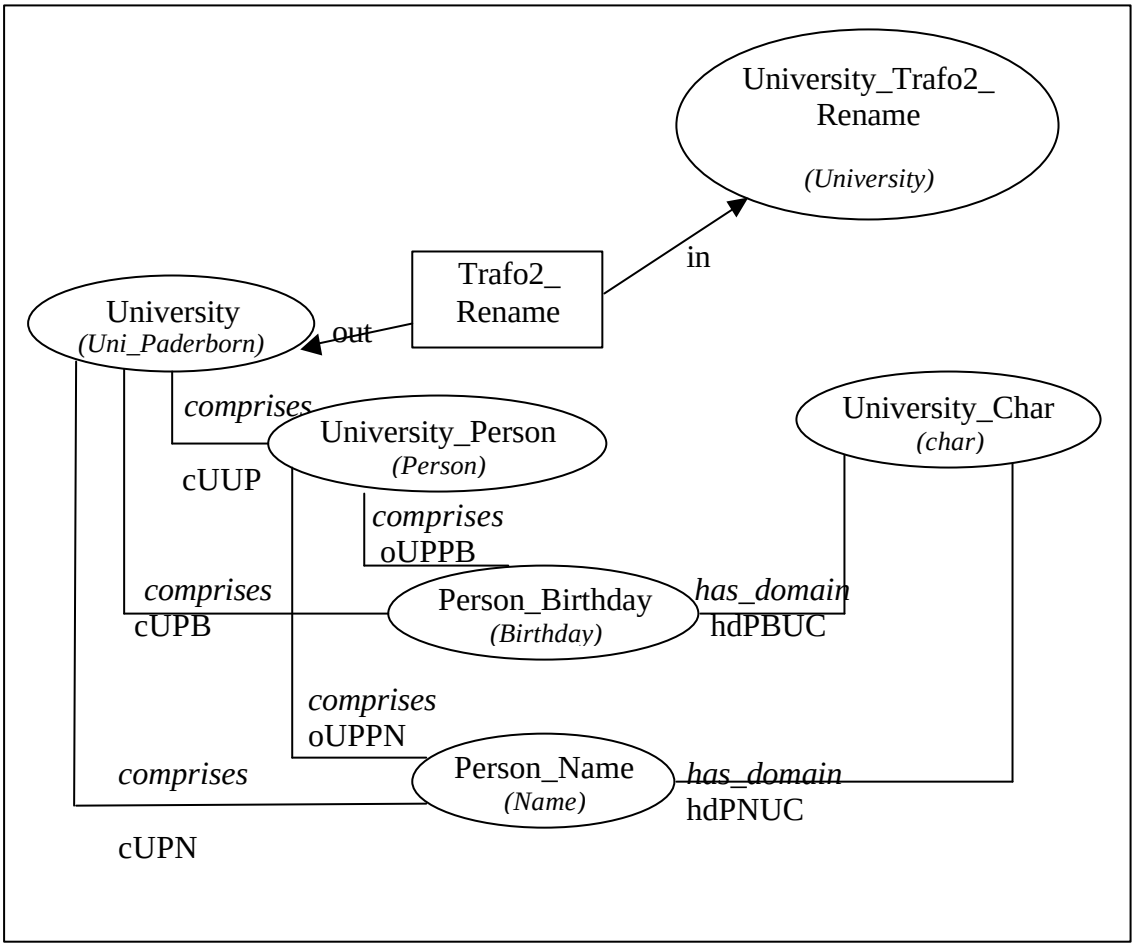


Abbildung 14: Ergebnisgraph

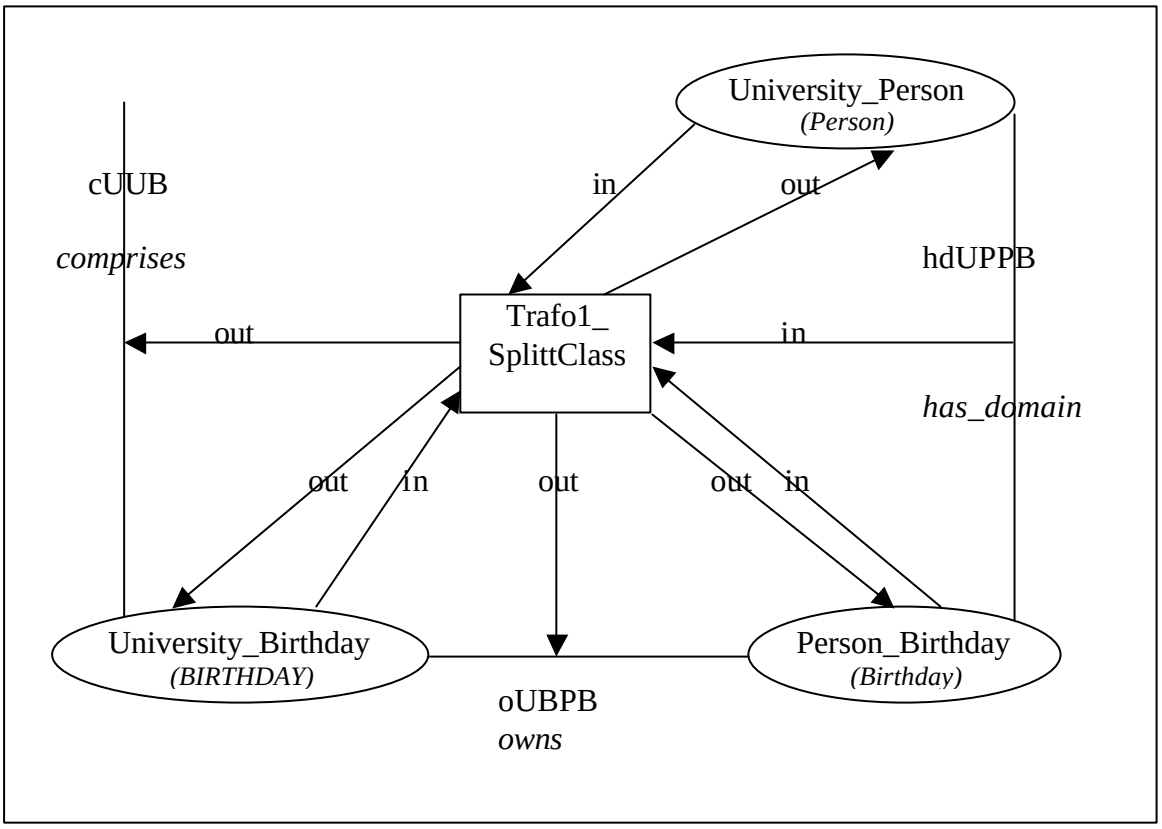


Abbildung 15: Reportgraph

Kapitel 5

Ausblick

Das Werkzeug, das in dieser Arbeit implementiert worden ist, erfüllt die Anforderungen, die dem Autor gestellt wurden. Es wäre aber wünschenswert ein benutzerfreundlicheres Werkzeug zu programmieren. Eine graphische Oberfläche wäre besser als die Eingabe von Operationen in der Main –Klasse. In großen GXL –Dateien etwas zu suchen ist schwer. Sogar einer erfahrener XML Leser verliert schnell die Übersicht, weil die Formatierung nicht immer gut ist. Deshalb ist es vielleicht sehr gut, erstens eine Formatierung vorzunehmen und zweitens eine Möglichkeit zu schaffen die Graphen graphisch darzustellen. Eine lesbare Formatierung zu bekommen, bedeutet langes Jonglieren mit JAXP. Es wäre auch von Vorteil, wenn eine Rückgabe des Graphen vor Transformation X und die Rückgabe einer Transformationsliste von dem Werkzeug angeboten würde.

Literatur:

1. J. P. Wadsack. “Inkrementelle Konsistenzerhaltung in der transformationsbasierten Datenbankmigration“, Diplomarbeit, Universität Paderborn. September 1998.
2. Prof. Dr. S. Böttcher Vorlesungsfolien zur Vorlesung „Datenbanken II“, WS 00/01.
3. World Wide Web Consortium (W3C) http://www.w3c.org/
4. Graph eXchange Language (GXL) http://www.gupro.de/GXL
5. JAXP http://java.sun.com/xml/xml_jaxp.html
6. R.C. Holt, A. Winter, A. Schürr „GXL: Toward a Standard Exchange Format”.
7. E. J. Chikofsky und J. H. Cross II, Reverse Engineering and Design recovery:{A} Taxonomy. In journal IEEE Software, volume 7, number 1, pages 13-17, January 1990.
8. J.-H. Jahnke. „Managing Uncertainty and Inconsistency in Database Reengineering Processes”. Ph.D. Dissertation, University of Paderborn, Germany,1999.

Anhang A

Graph eXchange Language Document Type Definition

gxl.dtd

```
<!-- GXL (1.0)
     Document Type Definition
     (February 14, 2001)

copyright by

       Andy Schuerr
         Institute for Software Technology
         University of the German Federal Armed Forces Munich
         85577 Neubiberg, Germany
         Andy.Schuerr@unibw-muenchen.de

       Susan Elliott Sim
         Department of Computer Science
         University of Toronto
         Toronto M5S 3G4, Canada
         simsuz@cs.utoronto.ca

       Ric Holt
         Department of Computer Science
         University of Waterloo
         Waterloo N2L 3G1, Canada
         holt@plg.uwaterloo.ca

       Andreas Winter
         Institute for Software Technology
         University of Koblenz-Landau
         Rheinau 1, D-56075 Koblenz, Germany
         winter@uni-koblenz.de
-->

<!-- Extensions -->

<!ENTITY % gxl-extension     "" >
<!ENTITY % graph-extension   "" >
<!ENTITY % node-extension    "" >
<!ENTITY % edge-extension    "" >
<!ENTITY % rel-extension     "" >
<!ENTITY % value-extension   "" >
<!ENTITY % relend-extension  "" >

<!ENTITY % gxl-attr-extension     "" >
<!ENTITY % graph-attr-extension   "" >
<!ENTITY % node-attr-extension    "" >
<!ENTITY % edge-attr-extension    "" >
<!ENTITY % rel-attr-extension     "" >
<!ENTITY % relend-attr-extension  "" >
```

```
<!-- Attribute values -->

<!ENTITY % val "
           locator |
           bool    |
           int     |
           float   |
           string  |
           enum    |
           seq     |
           set     |
           bag     |
           tup
           %value-extension;">


<!-- gxl -->

<!ELEMENT gxl  (graph* %gxl-extension;) >
<!ATTLIST gxl
    xmlns:xlink CDATA    #FIXED    "www.w3.org/1999/xlink"
    %gxl-attr-extension;
>


<!-- type -->

<!ELEMENT type EMPTY>
<!ATTLIST type
    xlink:type  (simple)    #FIXED  "simple"
    xlink:href  CDATA       #REQUIRED
>


<!-- graph -->

<!ELEMENT graph (type? , attr* , ( node | edge | rel )* %graph-extension;)
>
<!ATTLIST graph
    id          ID       #REQUIRED
    role        NMTOKEN  #IMPLIED
    edgeids     ( true | false ) "false"
    hypergraph  ( true | false ) "false"
    edgemode    ( directed | undirected |
                    defaultdirected | defaultundirected) "directed"
    %graph-attr-extension;
>


<!-- node -->

<!ELEMENT node (type? , attr*, graph* %node-extension;) >
<!ATTLIST node
    id          ID       #REQUIRED
    %node-attr-extension;
>
```

```
<!-- edge -->

<!ELEMENT edge (type?, attr*, graph* %edge-extension;) >
<!ATTLIST edge
    id            ID       #IMPLIED
    from          IDREF    #REQUIRED
    to            IDREF    #REQUIRED
    fromorder     CDATA    #IMPLIED
    toorder       CDATA    #IMPLIED
    isdirected    ( true | false ) #IMPLIED
    %edge-attr-extension;
>


<!-- rel -->

<!ELEMENT rel (type? , attr*, graph*, relend* %rel-extension;) >
<!ATTLIST rel
    id          ID            #IMPLIED
    isdirected    ( true | false ) #IMPLIED
    %rel-attr-extension;
>


<!-- relend -->

<!ELEMENT relend (attr* %relend-extension;) >
<!ATTLIST relend
    target      IDREF                #REQUIRED
    role        NMTOKEN              #IMPLIED
    direction   ( in | out | none)   #IMPLIED
    startorder  CDATA                #IMPLIED
    endorder    CDATA                #IMPLIED
    %relend-attr-extension;
>


<!-- attr -->

<!ELEMENT attr (type?, attr*, (%val;)) >
<!ATTLIST attr
    id     IDREF    #IMPLIED
    name   NMTOKEN  #REQUIRED
    kind   NMTOKEN  #IMPLIED
>


<!-- locator -->

<!ELEMENT locator  EMPTY >
<!ATTLIST locator
    xlink:type  (simple)    #FIXED  "simple"
    xlink:href  CDATA       #IMPLIED
>
```

```
<!-- atomic values -->

<!ELEMENT bool    (#PCDATA) >
<!ELEMENT int     (#PCDATA) >
<!ELEMENT float   (#PCDATA) >
<!ELEMENT string  (#PCDATA) >


<!-- enumeration value -->

<!ELEMENT enum    (#PCDATA) >


<!-- composite values -->

<!ELEMENT seq     (%val;)* >
<!ELEMENT set     (%val;)* >
<!ELEMENT bag     (%val;)* >
<!ELEMENT tup     (%val;)* >
```

Anhang B

GXL Dateien für Beispieldurchlauf

Startgraph

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
      <graph id="SQL-Meta-Schema">

            <node id="University">
                  <type xlink:href ="sql-meta-schema.gxl#SCHEMA"/>
                  <attr name="University"></attr>
            </node>

            <node id="University_Char">
                  <type xlink:href ="sql-metaschema.gxl#STD_DOMAIN"/>
                  <attr name="char"></attr>

            </node>

            <node id="University_Person">
                  <type xlink:href ="sql-meta schema.gxl#ENTITY_TYPE"/>
                  <attr name="PERSON"></attr>
            </node>

            <edge id = "comprises_University_University_Person"
                  from = "University" to = "University_Person">
                  <type xlink:href = "sql-meta-schema.gxl#comprises"/>
                  <attr name = "comprises"></attr>
            </edge>

            <node id="Person_Name">
                  <type xlink:href ="sql-meta-schema.gxl#ATTRIBUTE"/>
                  <attr name="Name"></attr>
            </node>

            <edge id = "comprises_University_Person_Name"
                  from = "University" to = "Person_Name">
                  <type xlink:href = "sql-meta-schema.gxl#comprises"/>
                  <attr name = "comprises"></attr>
            </edge>

            <edge id = "owns_University_Person_Person_Name"
                  from = "University_Person" to = "Person_Name">
                  <type xlink:href = "sql-meta-schema.gxl#owns"/>
                  <attr name = "owns"></attr>
            </edge>

            <edge id = "has_domain_Person_Name_University_Char"
                  from = "Person_Name" to = "University_Char">
                  <type xlink:href = "sql-meta-schema.gxl#has_domain"/>
                  <attr name = "has_domain"></attr>
            </edge>
```

```
            <node id="Person_Birthday">
                  <type xlink:href ="sql-meta-schema.gxl#ATTRIBUTE"/>
                  <attr name="Birthday"></attr>
            </node>

            <edge id = "comprises_University_Person_Birthday"
                  from = "University_" to = "Person_Birthday">
                  <type xlink:href = "sql-meta-schema.gxl#comprises"/>
                  <attr name = "comprises"></attr>
            </edge>

            <edge id = "owns_University_Person_Person_Birthday"
                  from = "University_Person" to = "Person_Birthday">
                  <type xlink:href = "sql-meta-schema.gxl#owns"/>
                  <attr name = "owns"></attr>
            </edge>

            <edge id = "has_domain_Person_Birthday_University_Char"
                  from = "Person_Birthday" to = "University_Char">
                  <type xlink:href = "sql-meta-schema.gxl#has_domain"/>
                  <attr name = "has_domain"></attr>
            </edge>

      </graph>
</gxl>
```

Trafo1_SplittClass

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
      <graph id="Trafo_1">

            <node id="University_Person">
                  <type xlink:href ="sql-meta-schema.gxl#ENTITY_TYPE"/>
                  <attr name="PERSON"></attr>
            </node>

            <node id="Person_Birthday">
                  <type xlink:href ="sql-meta-schema.gxl#ATTRIBUTE"/>
                  <attr name="Birthday"></attr>
            </node>

            <edge id = "owns_University_Person_Person_Birthday"
                  from = "University_Person" to = "Person_Birthday">
                  <type xlink:href = "sql-meta-schema.gxl#owns"/>
                  <attr name ="owns"></attr>
            </edge>

            <node id="Trafo1_SplitClass">
                  <type xlink:href ="sql-meta-schema.gxl#TRANSFORMATION"/>
                  <attr name="SplitClass"></attr>
            </node>

            <edge id = "in_University_Person"
                  from = "University_Person" to = "Trafo1_SplitClass">
                  <type xlink:href = "sql-meta-schema.gxl#in"/>
                  <attr name = "in"></attr>
            </edge>
```

```
        <edge id = "in_Person_Birthday"
              from = "Person_Birthday" to = "Trafo1_SplitClass">
              <type xlink:href = "sql-meta-schema.gxl#in"/>
              <attr name = "in"></attr>
        </edge>

        <edge id = "in_owns_University_Person_Person_Birthday"
              from = "owns_University_Person_Person_Birthday"
              to = "Trafo1_SplitClass">
              <type xlink:href = "sql-meta-schema.gxl#in"/>
              <attr name = "in"></attr>
        </edge>

        <node id="University_Birthday">
              <type xlink:href ="sql-meta-schema.gxl#ENTITY_TYPE"/>
              <attr name="BIRTHDAY"></attr>
        </node>

        <edge id = "comprises_University_University_Birthday"
              from = "University" to = "University_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#comprises"/>
              <attr name = "comprises"></attr>
        </edge>

        <edge id = "owns_University_Birthday_Person_Birthday"
              from = "University_Birthday" to = "Person_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#owns"/>
              <attr name = "owns"></attr>
        </edge>

        <edge id = "out_University_Person"
              from = "Trafo1_SplitClass" to = "University_Person">
              <type xlink:href = "sql-meta-schema.gxl#out"/>
              <attr name = "out"></attr>
        </edge>

        <edge id = "out_University_Birthday"
              from = "Trafo1_SplitClass" to = "University_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#out"/>
              <attr name = "out"></attr>
        </edge>

        <edge id = "out_Person_Birthday"
              from = "Trafo1_SplitClass" to = "Person_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#out"/>
              <attr name = "out"></attr>
        </edge>

        <edge id = "out_owns_University_Birthday_Person_Birthday"
              from = "Trafo1_SplitClass"
              to = "owns_University_Birthday_Person_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#out"/>
              <attr name = "out"></attr>
        </edge>

        <edge id = "out_comprises_University_University_Birthday"
              from = "Trafo1_SplitClass"
              to = "comprises_University_University_Birthday">
              <type xlink:href = "sql-meta-schema.gxl#out"/>
              <attr name = "out"></attr>
        </edge>

  </graph> </gxl>
```

Trafo2_Rename

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
      <graph id="Trafo_2">



            <node id="Trafo2_Rename">
                  <type xlink:href ="sql-meta-schema.gxl#TRANSFORMATION"/>
                  <attr name="Rename"></attr>
            </node>

            <edge id = "in_University"
                  from = "University" to = "Trafo2_Rename">
                  <type xlink:href = "sql-meta-schema.gxl#in"/>
                  <attr name = "in"></attr>
            </edge>

            <node id="University">
                  <type xlink:href ="sql-meta-schema.gxl#SCHEMA"/>
                  <attr name="Uni_Paderborn"></attr>
            </node>

            <edge id = "out_University"
                  from = "Trafo2_Rename" to = "University">
                  <type xlink:href = "sql-meta-schema.gxl#in"/>
                  <attr name = "in"></attr>
            </edge>

      </graph>
</gxl>
```

Veränderung

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
    <graph id="Change_1">
         <node id="University_Person">
              <type xlink:href ="sql-meta-schema.gxl#ENTITY_TYPE"/>
              <attr name="STUDENT"></attr>
         </node>

    </graph>
</gxl>
```

Ergebnisgraphen

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
  <graph id="SQL-Meta-Schema">

     <node id="University_Trafo2_Rename">
        <type xlink:href="sql-meta-schema.gxl#SCHEMA" />
        <attr name="University" />
     </node>

    <node id="University_Char">
      <type xlink:href="sql-meta-schema.gxl#STD_DOMAIN" />
      <attr name="char" />
    </node>

    <edge id="comprises_University_University_Person"
      from="University" to="University_Person">
      <type xlink:href="sql-meta-schema.gxl#comprises" />
      <attr name="comprises" />
    </edge>

    <node id="Person_Name">
      <type xlink:href="sql-meta-schema.gxl#ATTRIBUTE" />
      <attr name="Name" />
    </node>

    <edge id="comprises_University_Person_Name"
      from="University" to="Person_Name">
      <type xlink:href="sql-meta-schema.gxl#comprises" />
      <attr name="comprises" />
    </edge>

    <edge id="owns_University_Person_Person_Name"
       from="University_Person" to="Person_Name">
       <type xlink:href="sql-meta-schema.gxl#owns" />
       <attr name="owns" />
    </edge>

    <edge id="has_domain_Person_Name_University_Char"
       from="Person_Name" to="University_Char">
       <type xlink:href="sql-meta-schema.gxl#has_domain" />
       <attr name="has_domain" />
    </edge>

    <node id="Person_Birthday">
       <type xlink:href="sql-meta-schema.gxl#ATTRIBUTE" />
       <attr name="Birthday" />
    </node>

    <edge id="comprises_University_Person_Birthday"
       from="University" to="Person_Birthday">
       <type xlink:href="sql-meta-schema.gxl#comprises" />
       <attr name="comprises" />
    </edge>

    <edge id="owns_University_Person_Person_Birthday"
       from="University_Person" to="Person_Birthday">
       <type xlink:href="sql-meta-schema.gxl#owns" />
       <attr name="owns" />
    </edge>
```

```
    <edge id="has_domain_Person_Birthday_University_Char"
       from="Person_Birthday" to="University_Char">
       <type xlink:href="sql-meta-schema.gxl#has_domain" />
       <attr name="has_domain" />
    </edge>

    <node id="Trafo2_Rename">
       <type xlink:href="sql-meta-schema.gxl#TRANSFORMATION" />
       <attr name="Rename" />
    </node>

    <edge id="in_University" from="University" to="Trafo2_Rename">
       <type xlink:href="sql-meta-schema.gxl#in" />
       <attr name="in" />
    </edge>

    <node id="University">
       <type xlink:href="sql-meta-schema.gxl#SCHEMA" />
       <attr name="Uni_Paderborn" />
    </node>

    <edge id="out_University" from="Trafo2_Rename" to="University">
       <type xlink:href="sql-meta-schema.gxl#in" />
       <attr name="in" />
    </edge>

    <node id="University_Person">
       <type xlink:href="sql-meta-schema.gxl#ENTITY_TYPE" />
       <attr name="STUDENT" />
    </node>
</graph>
</gxl>
```

Reportgraphen

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl.dtd">
<gxl>
<graph>
   <edge id="in_University_Person"
       from="University_Person" to="Trafo1_SplitClass">
       <type xlink:href="sql-meta-schema.gxl#in" />
       <attr name="in" />
   </edge>

   <edge id="in_Person_Birthday"
       from="Person_Birthday" to="Trafo1_SplitClass">
       <type xlink:href="sql-meta-schema.gxl#in" />
       <attr name="in" />
   </edge>

   <edge id="in_owns_University_Person_Person_Birthday"
      from="owns_University_Person_Person_Birthday" to="Trafo1_SplitClass">
       <type xlink:href="sql-meta-schema.gxl#in" />
       <attr name="in" />
   </edge>

   <node id="Trafo1_SplitClass">
       <type xlink:href="sql-meta-schema.gxl#TRANSFORMATION" />
       <attr name="SplitClass" />
   </node>
```

```
  <edge id="out_University_Person"
     from="Trafo1_SplitClass" to="University_Person">
     <type xlink:href="sql-meta-schema.gxl#out" />
     <attr name="out" />
  </edge>

  <edge id="out_University_Birthday"
       from="Trafo1_SplitClass" to="University_Birthday">
       <type xlink:href="sql-meta-schema.gxl#out" />
       <attr name="out" />
  </edge>

  <node id="University_Birthday">
     <type xlink:href="sql-meta-schema.gxl#ENTITY_TYPE" />
     <attr name="BIRTHDAY" />
  </node>

  <edge id="out_owns_University_Birthday_Person_Birthday"
    from="Trafo1_SplitClass”
    to="owns_University_Birthday_Person_Birthday">
    <type xlink:href="sql-meta-schema.gxl#out" />
    <attr name="out" />
 </edge>

 <edge id="owns_University_Birthday_Person_Birthday"
    from="University_Birthday" to="Person_Birthday">
    <type xlink:href="sql-meta-schema.gxl#owns" />
   <attr name="owns" />
 </edge>

 <edge id="out_Person_Birthday"
    from="Trafo1_SplitClass" to="Person_Birthday">
    <type xlink:href="sql-meta-schema.gxl#out" />
    <attr name="out" />
 </edge>

 <edge id="out_comprises_University_University_Birthday"
    from="Trafo1_SplitClass"
    to="comprises_University_University_Birthday">
    <type xlink:href="sql-meta-schema.gxl#out" />
    <attr name="out" />
 </edge>

 <edge id="comprises_University_University_Birthday"
     from="University" to="University_Birthday">
     <type xlink:href="sql-meta-schema.gxl#comprises" />
     <attr name="comprises" />
 </edge>

 <node id="University_Person">
    <type xlink:href="sql-meta-schema.gxl#ENTITY_TYPE" />
    <attr name="PERSON" />
 </node>

 <node id="Person_Birthday">
    <type xlink:href="sql-meta-schema.gxl#ATTRIBUTE" />
    <attr name="Birthday" />
</node>
```

```
  <edge id="owns_University_Person_Person_Birthday"
    from="University_Person" to="Person_Birthday">
    <type xlink:href="sql-meta-schema.gxl#owns" />
    <attr name="owns" />
  </edge>
</graph>
</gxl>
```