Universität Paderborn

Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
D-33098 Paderborn

Entwicklung eines generischen Editors für Schematransformationen

Studienarbeit
für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

Matthias Bothe
Adelheidstr.15
33098 Paderborn

vorgelegt bei
Prof. Schäfer
und
Prof. Keil-Slawik

Paderborn, im September 2001

Inhaltsverzeichn is

23. Oktober 2001

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Neue Anwendungsbereiche wie E-Commerce und Telearbeit stellen neue Anforderungen an die hierfür in Frage kommenden Informationssysteme. Die weit verbreiteten relationalen Datenbanksysteme sind ungeeignet zur effizienten Verwaltung der in den genannten Anwendungsbereichen vozufindenen komplexen

Datenstrukturen. Sie sind deshalb ungeeignet, weil das relationale Datenmodell keine hierarchischen Datenstrukturen zulässt. Das objektorientierte Datenmodell hingegen eignet sich hierfuer. Mit der Einführung des Standards ODMG-93 für objektorientierte Datenbanken und der zunehmenden Verfügbarkeit kommerzieller Produkte in dieser Sparte stellt sich in der Praxis die Frage wie ein , schrittweiser und kostensparender Übergang von den vorhandenen relationalen zu den neuen objektorientierten Systemen vollzogen werden kann. Hierzu werden insbesondere Konzepte und Techniken benötigt, mit deren Hilfe relationale in objektorientierte Systeme überführt (migriert) beziehungsweise integriert werden können. Eine Technik ist der Aufbau einer objektorientierte Zugriffschichten für relationale Datenbanken.

Im Rahmen des Varlet Projektes wurde ein Werkzeug implementiert, das erstens die Analyse eines relationalen Datenbankschemas ermöglicht, zweitens ist eine automatische initiale Abbildung des relationalen Schemas in ein objektorientiertes Schema durchführbar. Drittens kann dieses objektorientierte Schema dann mit Schematransformationen weiter verfeinert werden. Zudem ist ein Konsistenzerhaltungs-Mechanismus in Varlet integriert.

1.2 Ziel der Arbeit

Das Ziel dieser Studienarbeit ist die Enwicklung eines generischen Editors für Transformationen auf einem objektorientierten Schema. Eine Transformation soll im Editor graphisch spezifiziert werden können. Aus der Spezifikation soll der Quelltext einer Methode generiert werden. Diese soll dem Benutzer die Möglichkeit bieten sein objektorientiertes Schema (Klassendiagramm) zu restrukturieren beziehungsweise zu verfeinern. Die Transformation soll für ein Konsistenzkonzept in der GXL (Graph EXchange Language) gespeichert werden. Dieses Konzept wurde im Rahmen der Studienarbeit von [Kra01] realisiert.

1.3 Struktur der Arbeit

In Kapitel 2 werden die für diese Arbeit notwendigen Grundlagen beschrieben.

Kapitel 3 beschreibt die Realisierung der Abbildung des spezifizierten Transformationsdiagramms auf ein Aktivitätendiagramm aus dem der Quellcode der Methode generiert wird.

In Kapitel 4 wird die Speicherung der Transformation mit der GXL beschrieben.

In Kapitel 5 folgt ein kurze Beschreibung der Benutzerschnittstelle des Editors.

Kapitel 6 werden die Ergebnisse zusammengefaßt und ein Ausblick gegeben.

2 Grundlagen

In 2.1 wird der für diese Arbeit wichtige Begriff des Datenbankschemas erklärt . Danach werden Graphtransformationen, die zur Restrukturierung des objektorientierten Schemas benötigt werden, beschrieben. Das objektorientierte Schema wird hier mit UML-Klassendiagrammen dargestellt, die mit der in 2.3 beschriebenen Entwicklungsumgebung Fujaba spezifiziert werden können. In Reddmom wird diese Funktionalität wiederverwendet.

2.1 Datenbankschema

2.1.1 Datenmodelle

Die Struktur von Daten wird mit Hilfe von Datenmodellen beschrieben. Die Strukturbeschreibung einer Datenbank wird Schema genannt. Für diese Arbeit sind das relationale und das objektorientierte Datenmodell wichtig.

Grundlage des relationalen Datenmodells ist der mengentheoretische Begriff der Relation. Eine Relation wird meist in Form einer Tabelle dargestellt. Das Schema einer relationalen Datenbank besteht aus der Beschreibung mehrerer Relationen.

Objektorientierte Datenmodelle basieren auf Objekten, die Zustände und ein Verhalten haben. Der Zustand eines Objektes ist durch die Werte seiner Attribute und Beziehungen zu anderen Objekten definiert, sein Verhalten durch die mit ihm möglichen Operationen. Objekte werden nach Typen klassifiziert, die ihr mögliches Verhalten und ihre Zustandsmenge bestimmen. Ein objektorientiertes Schema besteht aus der Beschreibung mehrerer Objekte. Dies geschieht meist durch die Spezifizierung eines Klassendiagramms.

Eine Relation (Tabelle) in einer relationalen Datenbank kann als eine Klasse im objektorientierten Schema dargestellt werden. Eine Ausprägung der Relation (Zeile, Tupel) entspricht dann einem Objekt (Instanz) der Klasse. Eine detailliertere Beschreibung dieser Datenmodelle findet man in [Rum98].

2.1.2 Abbildung des relationalen Datenbankschemas auf ein objektorientiertes Schema

Diese Arbeit setzt eine Abbildung des relationalen Datenbankschemas auf ein objektorientiertes Schema voraus. Diese Abbildung geschieht automatisch mit Hilfe von Tripelgraphgrammatiken (vgl. [Lef95, JSZ96, Hol97]). Die grundlegende Idee der Tripelgraphgrammatiken basiert auf der Integration von zwei Dokumenten mit Hilfe eines dritten sogenannten Integrationsdokuments. Durch die Angabe einer Menge von Tripelregeln können die zu integrierenden Dokumente eindeutig aufeinander abgebildet werden. Diese Abbildung (Mapping) zwischen den korrespondierenden Inkrementen wird durch das Integrationsdokument verwaltet. So werden Tabellen auf Klassen, Beziehungen zwischen Tabellen auf Assoziationen und Attribute auf Attribute abgebildet. Das resultierende Schema ist objektorientiert, sieht dem relationalen jedoch sehr ähnlich.Das vorhandene objektorientierte

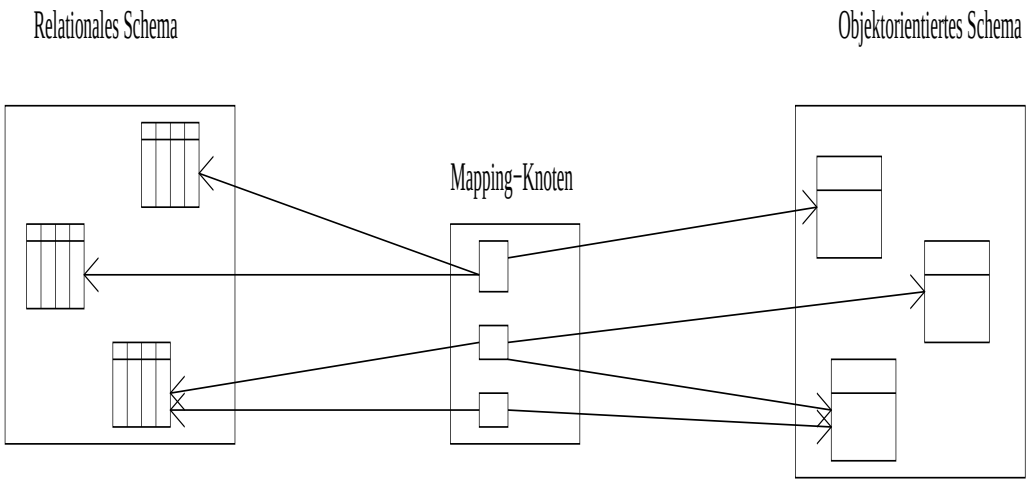


Abbildung 1: Mapping

Schema kann nun mit Transformationen restrukturiert werden, um dieses zu optimieren und die Vorteile der objektorientierten Konzepte zu nutzen. Siehe hierzu Abbildung 1.

2.2 Graphtransformationen

Ein Datenbankschema kann als Graph dargestellt werden. Zur Änderung des Schemas können daher Graphtransformationen eingesetzt werden. In dieser Arbeit werden Restrukturierungstransformationen zur Graphtransformation benötigt. Restrukturierungstransformationen werden durch Graphersetzungsregeln spezifiziert. Eine Graphersetzungsregel besteht aus einer linken und einer rechten Regelseite. Die linke beschreibt den zu modifizierenden Teilgraphen. Bei Anwendung der Regel wird dieser im Wirtsgraphen gesucht. Bei erfolgreicher Suche wird er durch den auf der rechten Regelseite beschriebenen Teilgraphen ersetzt. In VARLET wurde PROGRES (PROgrammierte GRaphErsetzungsSysteme) [Zün95] zur Beschreibung der Graphersetzungsregeln eingesetzt.

2.2.1 Restrukturierungstransformationen

Restrukturierungstransformationen verändern das objektorientierte Schema. Ein Beispiel für eine Restrukturierungstransformation ist das Abspalten einer Klasse (splitClass) von einer Klasse. Das heißt, es wird im Schema eine neue Klasse erzeugt und ein Teil der Eigenschaften (Attribute) der alten Klasse werden in die neue Klasse verschoben. In [Wad98] werden folgende Restrukturierungstransformationen aufgeführt.

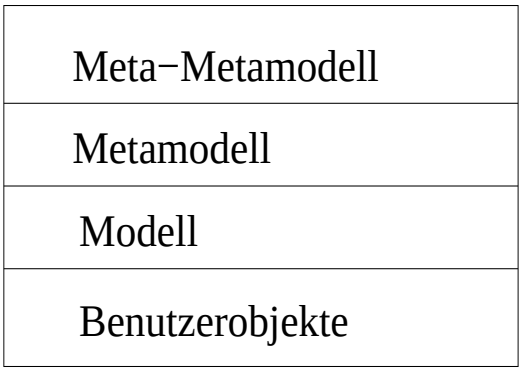
- Umwandlung einer Klasse in eine Assoziation
- Umwandlung einer Assoziation in eine Klasse
- Generalisierung
- Spezialisierung
- Verschmelzen von zwei Klassen zu einer Klasse
- Umwandlung einer Assoziation in eine Aggregation
- Umwandlung einer Aggregation in eine Assoziation
- Verschiebung von Attributen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

Restrukturierungstransformationen verändern das Schema und somit auch die Informationskapazität. Eine Klassifizierung der Transformationen, in Bezug auf den Begriff der Informationskapazität, ist u.a. in [BP96, JZ98] zu finden.

2.3 UML und Fujaba

Die Unified Modeling Language [UML] ist eine grafische, objektorientierte Modellierungssprache zur Spezifikation statischer und dynamischer Strukturen in einem Softwaresystem. UML wird insbesondere bei der Spezifikation großer und komplexer Softwaresystem eingesetzt. Sie wird in fast allen Phasen der Softwareentwicklung benutzt. Nahezu die gesamte Modellierung eines Softwaresystems kann in dieser standardisierten Sprache erfolgen.

Aus diesem Grund wurde in der Arbeitsgruppe Softwaretechnik der Universität Paderborn die Softwareentwicklungsumgebung Fujaba entwickelt [FNT98]. Fujaba ermöglicht dem Benutzer die Spezifikation von Softwaresystemen in UML-Notation und die Generierung von Java-Quellcode daraus. Der Entwurf der Datenstrukturen zur Speicherung von UML-Sprachelementen in Fujaba erfolgt in Anlehnung an das UML-Metamodell. Die UML-Metamodell-Architektur besteht aus vier Schichten und ist in Abbildung 2 dargestellt.



Jede Schicht in dieser Architektur ist eine Instanz der über ihr liegenden Schicht . Das Meta-Metamodell wird durch natürliche Sprache und Elementen aus der Metamodellebene beschrieben. Sie definiert so Struktur und Semantik der Metamodellebene . Die Metamodellebene beschreibt durch Klassen und Assoziationen

die UML-Diagramme der Modellebene. Die Modellebene enthält die vom Benutzer definierten UML-Diagramme. Die Diagrammelemente sind Instanzen der auf der Metamodellebene beschriebenen Klassen. Die letzte Schicht beschreibt die Benutzerobjekte. Sie sind die Instanzen der durch den Benutzer definierten Klassen auf der Modellebene.

Zur Speicherung aller UML-Sprachkonstrukte ist in Fujaba ein abstrakter Syntaxgraph implementiert. Der abstrakte Syntaxgraph stellt ein vereinfachtes UML-Metamodell plus Erweiterungen für Story Driven Modeling dar. Abbildung 3 zeigt einen Ausschnitt des abstrakten Syntaxgraphen für Klassendiagramme.

Die wichtigsten Strukturierungseinheiten in UML sind Diagramme. Diese werden in Fujaba durch die Klasse UMLDiagram modelliert. Sie bildet die abstrakte Oberklasse für alle Diagrammarten, wie UMLClassDiagram oder UMLActivityDiagram. Jedes Diagramm besteht aus mehreren Elementen. Ein Klassendigramm besteht zum Beispiel aus Klassen und Assoziationen, die durch UMLClass und UMLAssoc modelliert werden. Die Klasse UMLDiagramItem ist die Oberklasse aller Diagrammelemente. Die Container-Klasse UMLDiagram aggregiert die Objekte der Klasse UMLDiagramItem. UMLDiagram ist ebenfalls von UMLDiagramItem abgeleitet. Das heißt, dass ein Diagramm wieder Diagramme enthalten kann. Diese Konstruktion wird in [GHJV95] Composite-Pattern genannt.

Die Wurzel des abstrakten Syntaxgraphen ist eine Instanz der Klasse UMLProjekt. Die Klasse UMLProjekt modelliert ein aus mehreren Diagrammen bestehendes Projekt. Jedem Diagramm ist durch die diags-Assoziation genau ein Projekt zugeordnet. Die Klasse UMLProjekt ist als Singelton [GHJV95] implementiert, da in Fujaba immer nur genau ein Projekt bearbeitet werden soll. Das heißt, es existiert zu jeder Zeit höchstens eine Instanz der Klasse UMLProjekt, auf die von jeder Stelle des Systems mittels der statischen Methode get zugegriffen werden kann. Zur Verdeutlichung wird in Abbildung 4 die Struktur einer ASG-Ausprägung und ihre grafische Repräsentation in Fujaba gegenübergestellt.

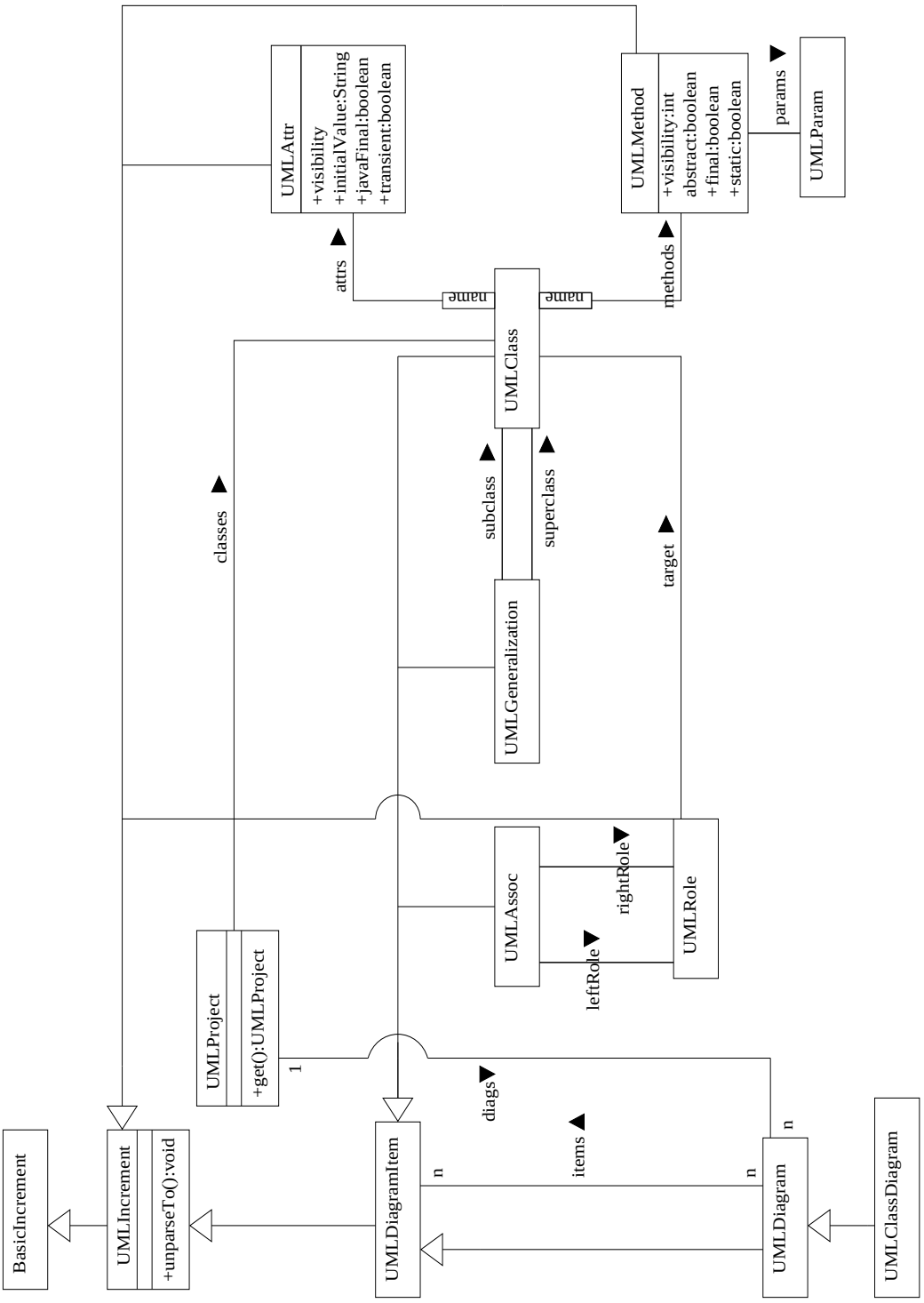


Abbildung 2: Der abstrakte Syntaxgraph für Klassendiagramme

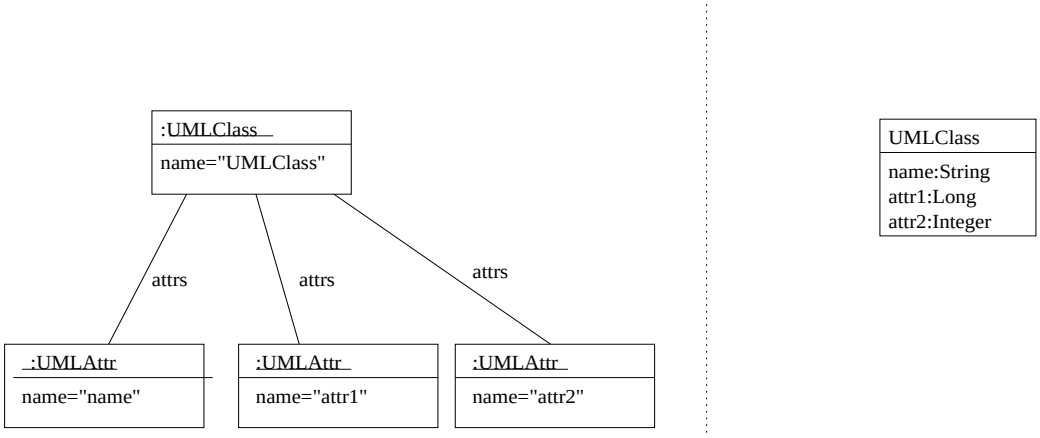


Abbildung 3: graphische Repräsentation eines Klassendiagramms in Fujaba und interne Darstellung

Im folgenden soll auf Story-Diagramme [JSZ98] beziehungsweise Aktivitätsdiagramme eingegangen werden. Story-Diagramme sind erweiterte Aktivitätsdiagramme, wobei die Aktivitäten entweder Story Pattern oder Java Quellcode enthalten. Story-Pattern definieren Transformationen auf der dynamischen Objektstruktur und entsprechen in ihrer Ausdrucksmächtigkeit in etwa den Graphersetzungsregeln von Progres. Da es der UML diesbezüglich an Spezifizierungsmöglichkeiten fehlte wurden sie in Fujaba als veränderte UML-Kollaborationsdiagramme implementiert. Ein Vergleich von Story-Diagrammen und Progres findet man in [FNT98]. Da ein Aktivitätsdiagramm zur Spezifizierung einer Methode einer Klasse dient, muß die die Methode enthaltende Klasse sowie die Klassen aller verwendeten Objekte auf einem Klassendiagramm im Projekt zu finden sein. Die Abbildung 5 zeigt ein Story-Pattern, Abbildung 6 seine interne Repräsentation und Abbildung 7 den abstrakten Syntaxgraphen für Aktivitätsdiagramme in Fujaba.

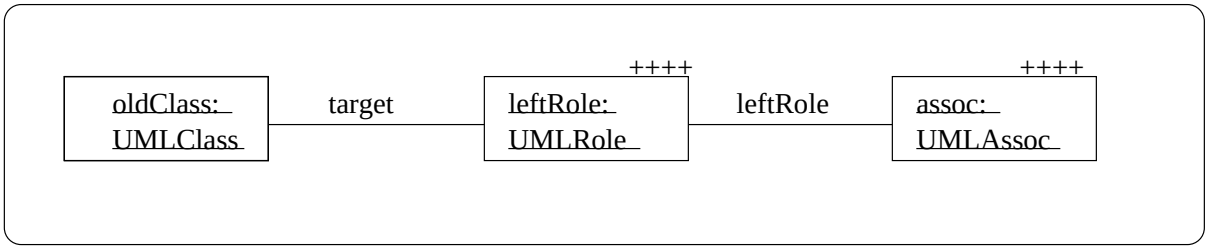


Abbildung 4: Story-Pattern

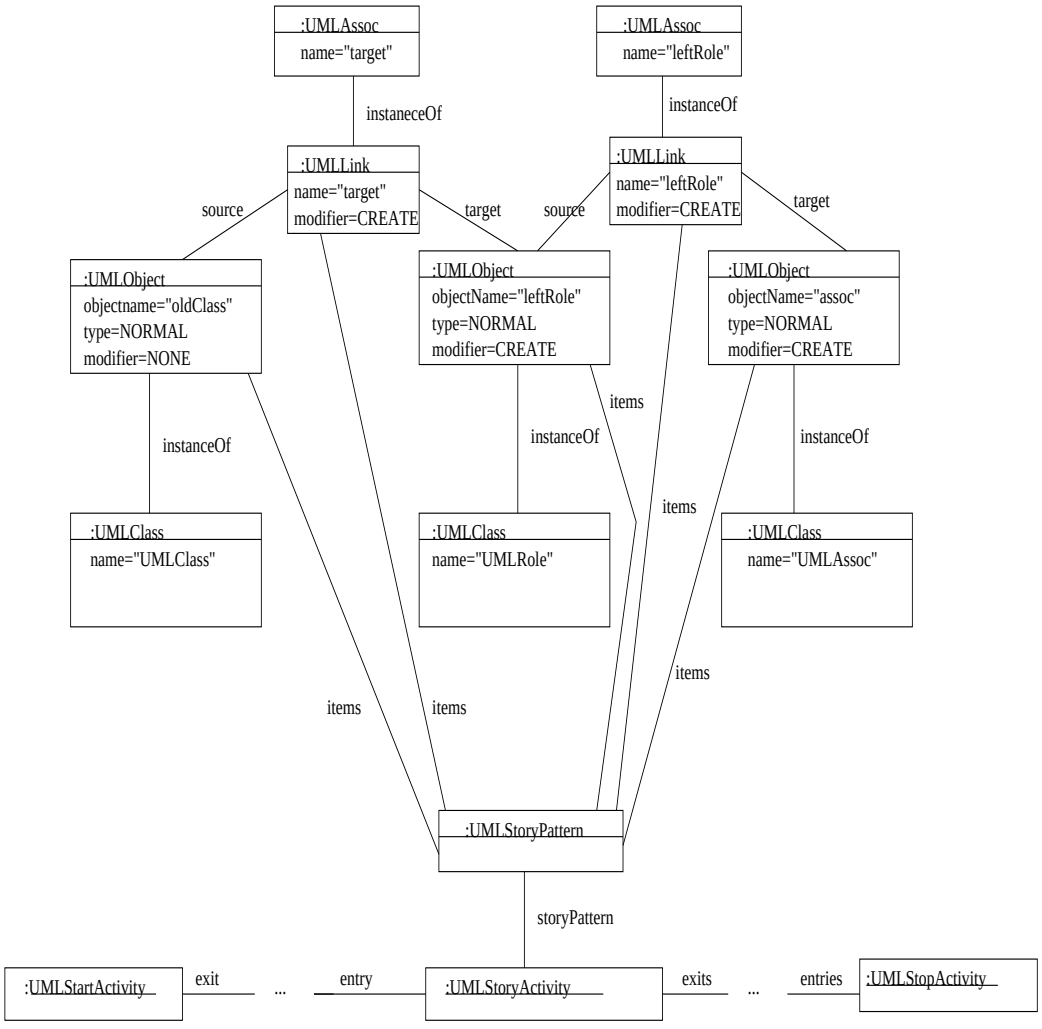


Abbildung 5: Interne Darstellung des Story-Pattern

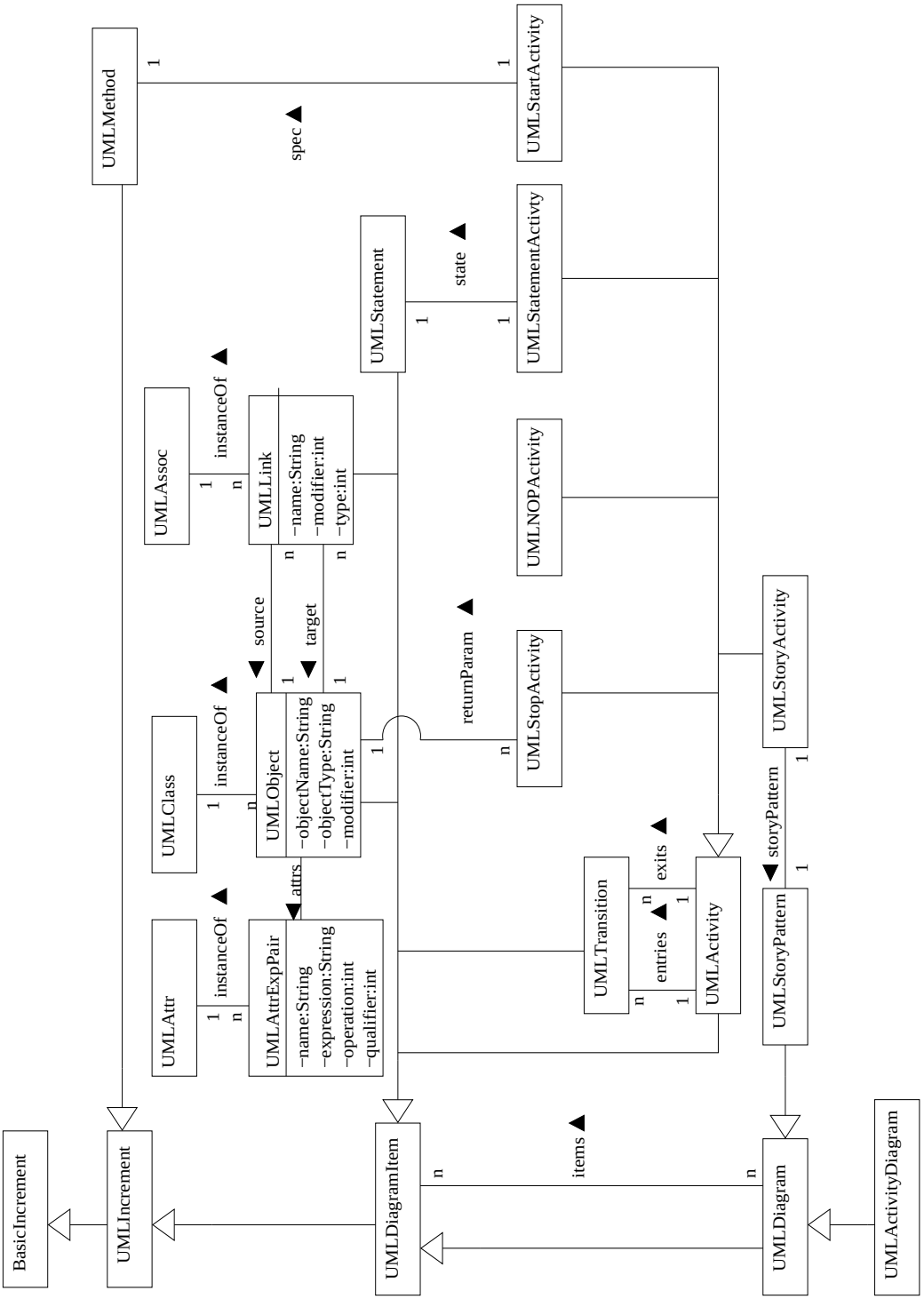


Abbildung 6: Abstrakter Syntaxgraph für Aktivitätsdiagramme

3 Realisierung

Die Erklärung der Abbildung vom Transformationsdiagramm auf das Aktivitätendiagramm wird im folgenden gemäß der Struktur des Aktivitätendiagramms in Abbildung 10 durchgeführt. Zunächst wird die Abbildung auf das Story-Pattern beschrieben, danach die Attributverschiebung.

3.1 Abbildung auf das Story-Pattern

In der Abbildung 8 zeigt das mit (1) markierte Bild ein mit dem Schematransformationseditor spezifiziertes Diagramm. Es sieht einem Klassendiagramm ähnlich. Die Diagrammelemente sind Repräsentationen von Klassen und Assoziationen, die im folgenden T-Klassen und T-Assoziationen genannt werden. Der ASG für Transformations-Diagramme (Abbildung 9) hat eine ähnliche Struktur wie der ASG für Klassendiagramme. Zum Beispiel besteht eine T-Assoziation, analog zu Assoziationen, aus zwei Rollenobjekten ((8) und (10)) und einem Assoziationsobjekt (9), eine T-Klasse aus einem Objekt vom Typ TransformationObject. Ein Attribut einer T-Klasse ist kein einzelnes Attribut, sondern ein Behälter (Vector) von Attributen, der beim Aufruf der durch dieses Diagramm spezifizierten Methode übergeben wird. Die Diagrammelemente können um eine Notation für Modifikationen durch Story-Pattern erweitert werden. Die Pluszeichen an einem Diagrammelement bedeuten, dass dieses Element im zugrundeliegenden Klassendiagramm (Wirtsgraphen) zu erzeugen ist. Die Modifikationsinformationen werden ebenfalls in den T-Diagrammelementen (2) gespeichert. Unter (3) sind die Objekte aufgeführt auf die die Objekte des Transformations-Diagramms abgebildet werden. Aus der Transformations-Diagramm Spezifikation (1) wird das in (5) gezeigte Story-Pattern erzeugt. (5) zeigt die der Story-Pattern-Variable "new" zugrundeliegende interne Repräsentation. Auf diese wird das TransformationObject (7) abgebildet. Das der Story-Pattern-Variablen zugrundeliegene Klassendia-

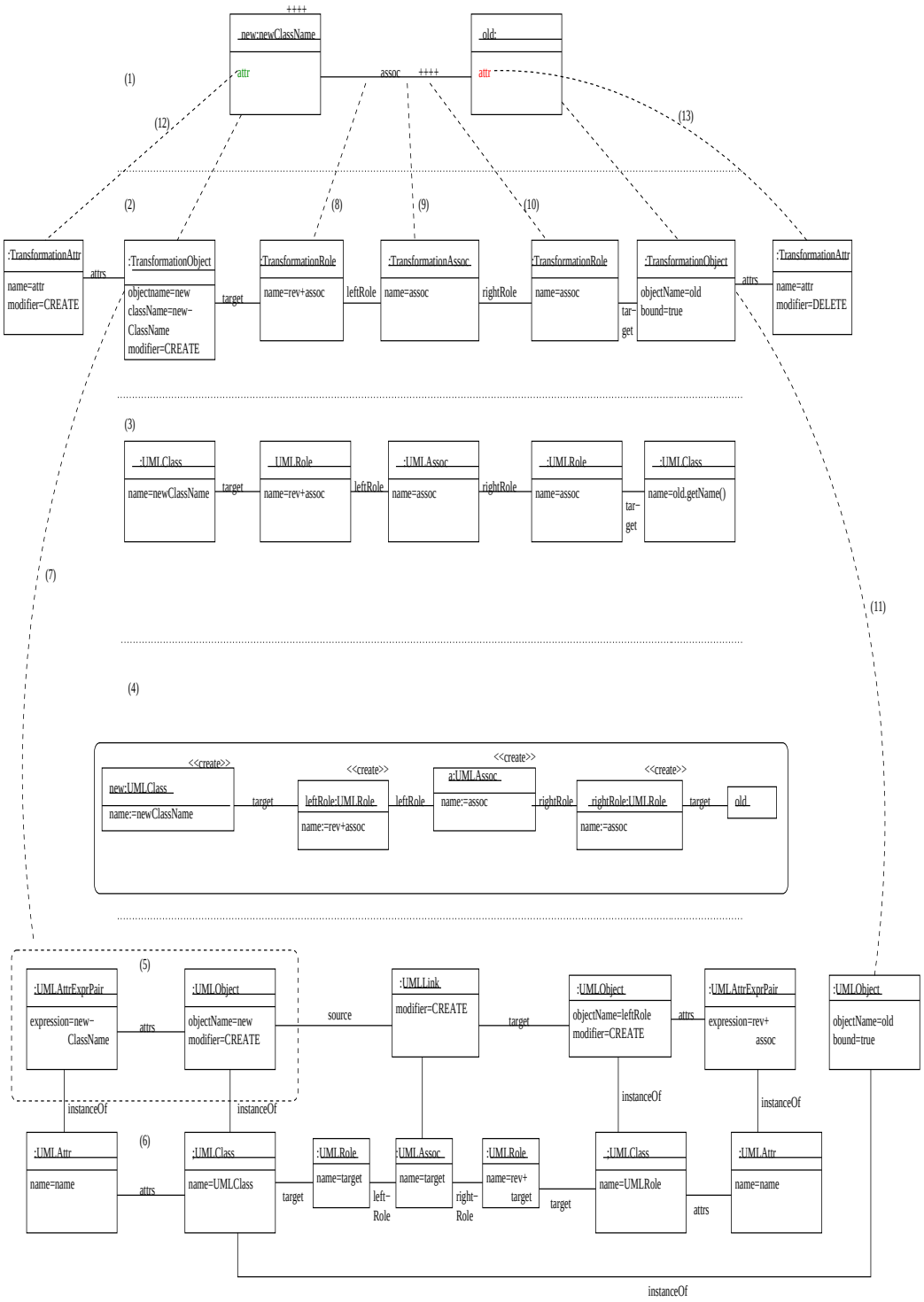


Abbildung 7: Abbildung auf Story-Pattern

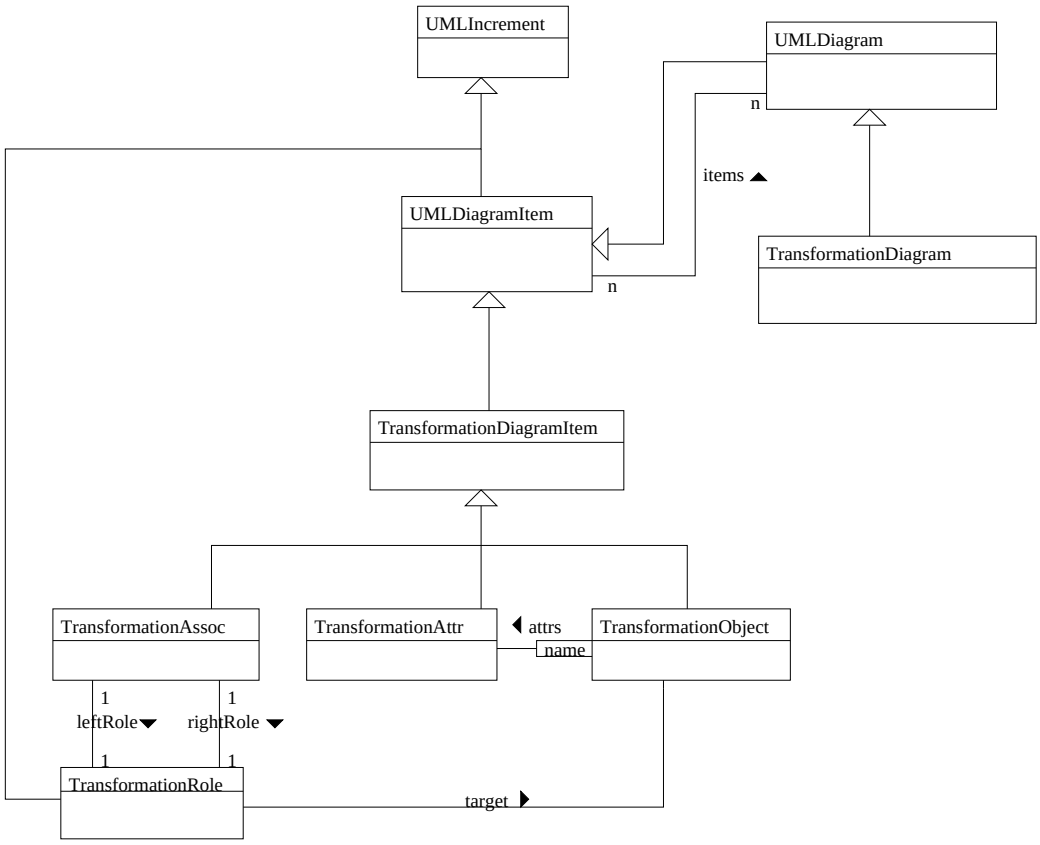


Abbildung 8: ASG für Transformations-Diagramme

gramm ist unter (6) zu sehen. Es entspricht der Struktur des ASG für Klassendiagramme in Fujaba. Die T-Klasse old ist an ein Objekt, das der Methode übergeben wird, gebunden, deshalb ist sie auf eine gebundene Variable (11) im Story-Pattern abzubilden.

3.2 Attributverschiebung

In der Abbildung 8 hat jede T-Klasse ein Attribut (1). Das grün dargestellte Attribut (12) ist in der Klasse zu erzeugen, das rote (13) zu löschen. Diese Information wird in dem Attribut modifier der Klasse TransformationAttr gespeichert. Die

Abbildung Struktur des Aktivitätendiagramms zeigt, daß nach erfolgreicher Anwendung des Story-Patterns, auf das die Klassen und Assoziationen abgebildet wurden, eine Codesequenz zur Attributverschiebung ausgeführt wird. Der Code dafür wird in einer Statement-Aktivität gespeichert.

Im folgenden wird ein Ausschnitt des Quelltextes der Abbildungs-Methode, der im Anhang aufgeführt ist, beschrieben .

Der in den Klammern der Methode append stehende Text, ist der in der Statement-Aktivität des erzeugten Aktivitätsdiagramms enthaltende Programmcode. Nach der Codegenerierung aus dem Aktivitätendiagramm, erscheint dieser unverändert an entsprechender Stelle im Rumpf der Methode. In der Zeile 8 wird ein Array erzeugt, das alle Referenzen auf Objekte vom Typ UMLClass, die im Story-Pattern verwendet wurden, speichert (Zeile 41). Die Arrays in Zeile 9 und 10 speichern die zu löschenden und zu erzeugenden Attributbehälter vom Typ Vector, der unter demselben Index gespeicherten "Klasse" (UMLClass-Objekt). In der while-Schleife von Zeile 54 bis 74 werden die Attribute nacheinander für jede Klasse erzeugt(Zeile 69) und gelöscht(Zeile 63).

4 Speicherung der Transformationen

4.1 GXL

GXL ist ein standardisiertes Austauschformat für Graphen und eine Teilsprache von XML. Ihre Syntax ist durch ein XML DTD beschrieben. Dieses Austaschformat ermöglicht die Interoperabilität zwischen graphbasierten Werkzeugen. GXL wurde insbesondere zur Herstellung von Kompatibilität zwischen Software-Reengineering-Werkzeugen entwickelt. Dadurch wird es möglich eine einheitliche Reengineering-Arbeitsumgebung aus unterschiedlichen Werkzeugen zusammenzustellen . Als

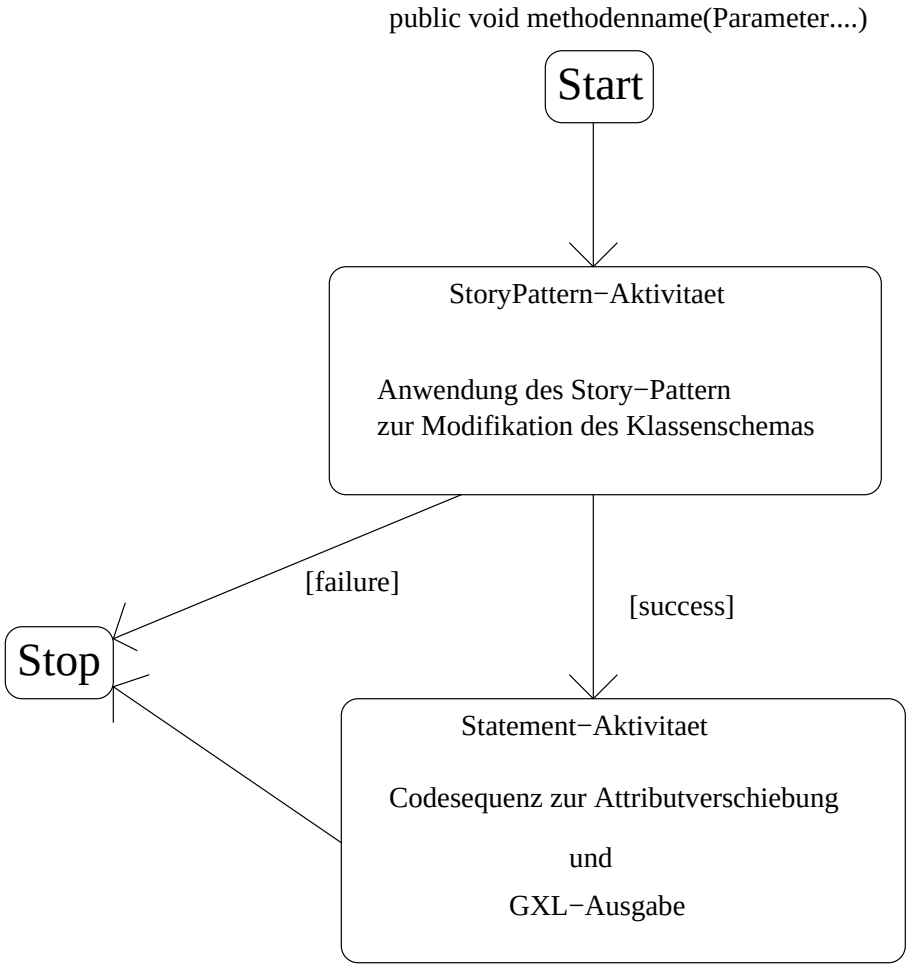


Abbildung 9: Struktur des Aktivitätendiagramms

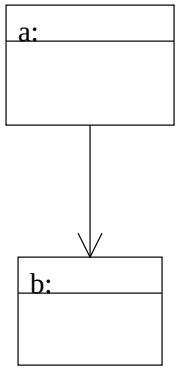
ein allgemeines Austauschformat für Graphen, kann es für jeden Austausch von graphbasierten Daten, wie zum Beispiel dem Austausch von Modellen zwischen CASE-Tools, eingesetzt werden. Eine ausführlichere Beschreibung und weitere Verwendungsmöglichkeiten von GXL findet man unter [GXL]. Es folgt eine kurze Beschreibung der Syntax (DTD) von GXL.

Ein GXL-Dokument kann als Graph (Baum) dargestellt werden. Die Wurzel des Dokuments bildet der Dokumentknoten gxl, der alle Elemente in der GXL-Datei enthält. Das heißt alle anderen Elemente in der Datei stehen zwischen den beiden Tags <gxl> (start-tag) und </gxl>(end-tag). Jeder Knoten hat ein Anfangtag und ein Endetag, die die Kindelemente des Knotens einschließen. Das nächste Strukturierungslement ist ein Graph(-knoten). Jede gxl-Datei kann mehrere Graphen beinhalten. Ein Graph beziehungsweise ein graph-Element kann folgende Elemente enthalten:

- ein optionales Element type zur Angabe des Graphtyps
- eine Menge von Attributelementen
- eine beliebige Kombination von node, rel und edge Knoten, wobei edge-Knoten eine Graphkante, ein rel-Knoten eine n-aere Beziehung und ein node-Knoten einen Graphknoten darstellt.

Das edge- und node-Element haben dieselbe Struktur und enthalten jeweils die folgenden Elemente.

- ein Graphelement
- ein optionales Element type zur Angabe des Knotentyps
- eine Menge von Attributelementen.




```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM "gxl1.0.dtd">
<gxl>
 <graph>
   <node id= "a">
   </node>
   <node id ="b">
   </node>
   <edge from= "a" to "b">
   </edge>
</graph>
</gxl>
```

Abbildung 10: Beispielgraph als GXL-Dokument

Das Element rel enthaelt zusaetzlich zu den gerade aufgefuehrten Elementen ein relend-Element. Ein Attributelement enthaelt:

- ein val-Element zur Angabe des Attributwerts
- ein optionales type-Element zur Angabe des Attributtyps
- eine Menge von Attributelementen

In der Abbildung “Beispielgraph als GXL-Dokument” sind ein einfacher Graph und seine Repraesentation durch ein GXL-Dokument dargestellt. Jeder Graphenknoten hat eine eindeutige ID, die im Attribut id des entsperechenden node-Elements gespeichert wird. Eine Kante wird durch die Angabe der IDs von Quell- und Zielknoten im edge-Element dargestellt. Es folgt eine kurze Beschreibung der Realisierung der GXL-Ausgabe in dieser Arbeit.

4.2 Generierung

Nach erfolgreicher Ausfuehrung des Story-Patterns (Abbildung auf Story-Pattern) werden alle darin erzeugten Instanzen als GXL-Knoten in einen StringBuffer ge-

schrieben. Da die Instanzen und deren OID erst zur Laufzeit, das heisst bei Anwendung der Methode, bekannt sind, wird der Code zur Erzeugung des String-Buffers und das Schreiben des GXL-Dokuments darin, zur Statement-Aktivitaet hinzugefuegt. Die Verbindungen (links) zwischen den Objekten, werden durch zwei Kanten im GXL-Dokument realisiert, da sie Instanzen bidirektionaler Assoziationen sind. In der Abbildung Codesequenz zur Attributverschiebung befindet sich in Zeile 71-72 der Code fuer das Schreiben der erzeugten Links zwischen den UMLClass-Objekten und den UMLAttr-Objekten in den GXL-StringBuffer.

5 Die Benutzerschnittstelle

Im folgenden werden die zur Erzeugung der Diagrammelemente dienenden Dialoge des Schemtransformationseditors kurz beschrieben.

Im T-Klassen-Dialog kennzeichnet

- Delete , das Löschen einer Klasse
- Create, das Erzeugen einer Klasse
- None, keine Modifikation der Klasse
- Normal, dass die Klasse vorhanden sein muß.
- Optional, die Klasse kann bei Anwendung der Transformation vorhanden sein, muß aber nicht.
- Negative, die Klasse darf nicht vorhanden sein.
- Bound, die Klasse wurde als Parameter (Ansatzpunkt in der Objektstruktur) übergeben.

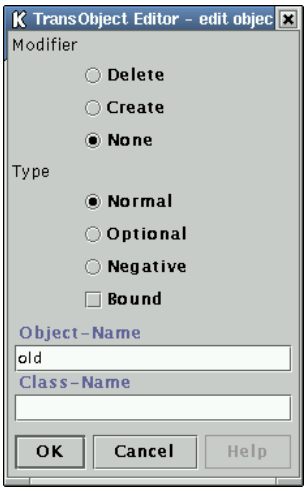


Abbildung 11: T-Class-Dialog

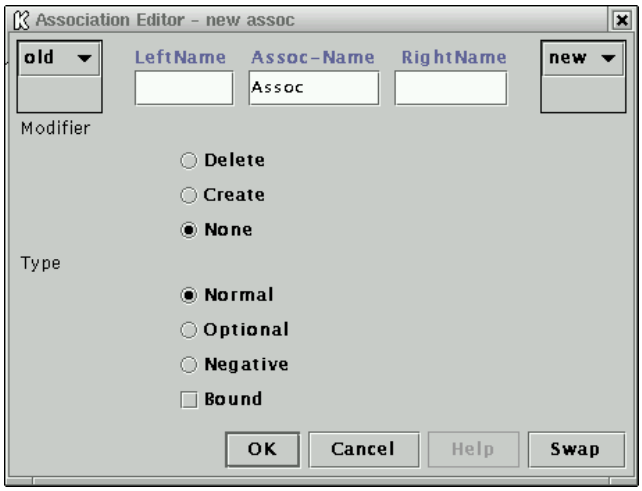


Abbildung 12: T-Assoc-Dialog

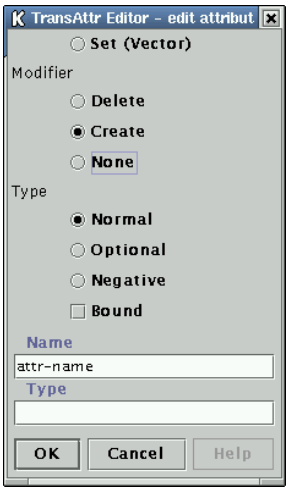


Abbildung 13: T-Attr-Dialog

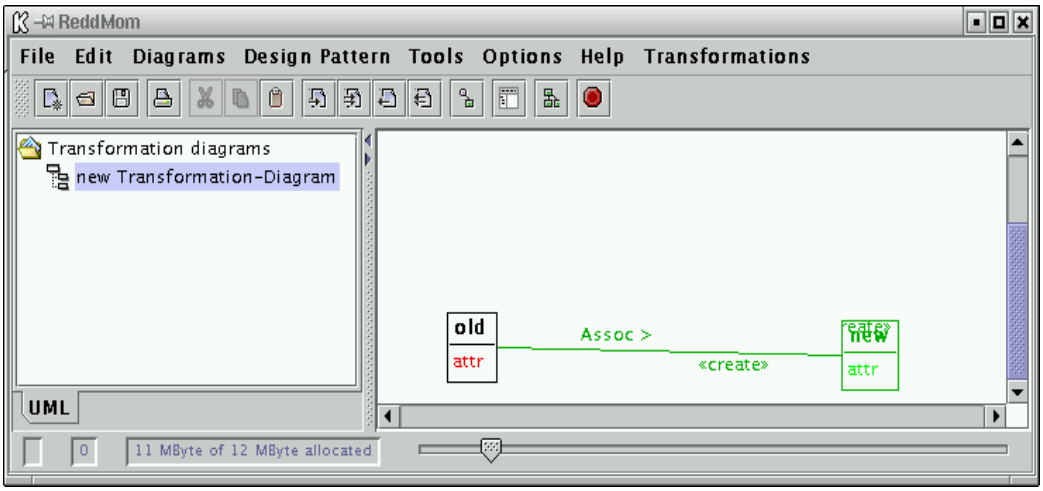


Abbildung 14: T-Diagramm

Die anderen Dialoge sind analog aufgebaut.

Abbildung 14 zeigt ein mit dem Schematransformationseditor erstelltes Diagramm in Reddmom. Auf der Arbeitsfläche sind zwei durch eine T-Assoziation verbundene T-Klassen zu sehen. Die im Klassenschema zu erzeugende Elemente sind mit dem Text <<create>> gekennzeichnet. Die beiden Attribute in den T-Klassen haben denselben Namen und bezeichnen somit denselben Attributcontainer. Die linke T-Klasse hat ein rot markiertes (löschen) und die rechte ein grün (erzeugen) markiertes Attribut. Diese Spezifikation entspricht dem Verschieben von Attributen zwischen den der Transformation zugrundeliegenden Klassen. Diese Transformation entspricht der in Kapitel 2 eingeführten Restrukurierungstransformation “splitClass”.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Editor für Schematransformationen implementiert, der dem Benutzer die Möglichkeit bietet Transformationen auf Klassendiagrammebene zu spezifizieren. Dieses wurde durch eine Abbildung der intern aufgebauten Strukturen auf ein Story-Diagramm realisiert. Aus diesem kann mit Fujaba Java-Quellcode für eine Methode generiert werden, die dann eine Transformation auf einem Klassendiagramm ausführen kann. Anschließend wurde die Transformation in GXL gespeichert. Dadurch wurde die Zustandänderung protokolliert. Dieses Protokoll kann in einer anderen Arbeit zur Konsistenzerhaltung verwendet werden.

Eine Erweiterung des Editors um eine dynamische Anbindung der generierten Methode an die Benutzeroberfläche als Editierfunktion für ein Klassendiagramm wäre komfortabel.

7 Anhang

```
1  //move attributs
2   UMLStatementActivity statementActivity = new UMLStatemen-
tActivity();
3  UMLStatement statement1 = new UMLStatement();
4
5  int numberOfTrafoObjects = diag.getNumberOfTrafoObjects();
6
7   statement1.append("int numberOfClasses ="+numberOfTrafoOb-
jects+";");
8  statement1.append("UMLClass[] arrayOfClasses = new UMLClass[number-
OfClasses];");
9  statement1.append("Vector[] attrsToDelete = new Vector [numberOf-

Classes];");
10  statement1.append("Vector[] attrsToCreate = new Vector [numberOf-

Classes];");
11
12  Iterator trafoObjIter = diag.iteratorOfTrafoObjects();
13  Iterator tmpTrafoAttrsIter;
14
15  int i=0;
16
17  TransformationAttr tmpTrafoAttr;
18  TransformationObject tmpTrafoObj;
19
20  while (trafoObjIter.hasNext())
21  {
```

```
22  tmpTrafoObject = (TransformationObject) trafoObjIter.next();
23  tmpTrafoAttrsIter = tmpTrafoObject.iteratorOfTrafoAttrs();
24
25  statement1.append("tmpAttrsToDeleteVector = new Vector();");
26   statement1.append("tmpAttrsToCreateVector = new Vector();");
27
28  while (tmpTrafoAttrsIter.hasNext())
29  {
30   tmpTrafoAttr = (TransformationAttr) tmpTrafoAttrsIter.next();
31  if (tmpTrafoAttr.markedAsDeleted())
32   {
33  statement1.append("tmpAttrsToDeleteVector.addElement("+tmpTrafoAttr.getName()+");");
34   }
35   if (tmpTrafoAttr.markedAsCreated())
36   {
37  statement1.append("tmpAttrsToCreateVector.addElement("+tmpTrafoAttr.getName()+");");
38   }//while
39  }//while
40
41 statement1.append("arrayOfClasses["+i+"]="+tmpTrafoObject.getObjectName()+";");
42 statement1.append("attrsToCreate["+i+"]= tmpAttrsToDeleteVec-
tor;");
43 statement1.append("attrsToDelete["+i+"]= tmpAttrsToDeleteVec-
tor;");
44
45  i=i+1;
46
47  }//while
48
49  statement1.append("int j=0");
```

```
50  statement1.append("Iterator tmpIterCreatedAttrs;");
51  statement1.append("UMLAttr tmpAttr;");
52  statement1.append("UMLClass tmpClass;");
53
54  statement1.append("while (j<numberOfClasses){");
55
56  statement1.append("tmpClass = arrayOfClasses[j];");
57 statement1.append("tmpIterCreatedAttrs= attrsToCreate[j].iterator();");
58 statement1.append("tmpIterDeletedAttrs= attrsToDelete[j].iterator();");
59
60  statement1.append("while(tmpIterDeletedAttrs.hasNext()){");
61
62 statement1.append("tmpAttr = (UMLAttr) tmpIterDeletedAttrs.next();");
63  statement1.append("tmpAttr.setParent(null);}");
64
65
66  statement1.append("while(tmpIterCreatedAttrs.hasNext()){");
67
68 statement1.append("tmpAttr = (UMLAttr) tmpIterCreatedAttrs.next();");
69  statement1.append("tmpAttr.setParent(tmpClass);")
70  //GXL———
71   statement1.append("gxlBuffer.append(\"<edge from =\"+tmpAt-
tr+\" to =
\"+tmpClass+\"></edge>);");
72  statement1.append("gxlBuffer.append(\"<edge from =\"+tmpClass+\"
to =
\"+tmpAttr+\"></edge>);}");
71  //GXL———
72  statement1.append("j=j+1;}");
73
```

8 Literatur

[UML] www.rational.com/uml/index.jsp.

[Kra01] http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/Stud/beschreibungen/laufende/HG_Mechanis

[Rum98] http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/VARLET/documents/Rum98.pdf

[Zün95] A. Zündorf. PROgramierte GRaphErsetzungsSysteme (Spezifikation, Implementie-rung und Anwendung einer integrierten Entwicklungsumgebung), Dissertation RWRH Aachen. Deutscher Universitätsverlag (1995)

[Wad98] http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/VARLET/documents/Wad98.pdf

[BP96] M. Blaha and W. Premerlani. A Catalog of Object Model Transformation. Presented at 3 rd Working Conference on Reverse Engineering, Monterey, California. November 1996

[FNT98] Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling Diplomanden: T. Fischer, J. Niere, L. Torunski http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/FUJABA/doc/DiplomTFischerJNiereLT

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides; Design Pattern Addison-Wesley, Bonn, 1995

[JZ98] J. Jahnke and A. Zündorf. Using Graph Grammars for Building the Varlet Database Re-verse Engineering Environment. Accepted for 1998 Intl. Workshop on Theory and Applications of Graph Grammars, Paderborn, Germany. November 1998.

[GXL] www.gupro.de/GXL

[JSZ96] J. Jahnke, W. Schäfer and A. Zündorf. A Design Environment for Migrating Relational to Object Oriented Database Systems. In Proc. of the 1996 Int Conference on Software Mainte-nance (ISCM96). IEE Computer Society, 1996.

[Lef95] M. Lefering. Integrationswerkzeuge in einer Softwareentwicklungsumgebung. Infor-matik, Verlag Shaker, 1995.

[Hol97] J. Holle. Ein Generator für integrierte Werkzeuge am Beispiel der object-relationalen Datenbankschemamigration. Diplomarbeit an der Universität-Gesamthochschule Paderborn, Juli 1997.