Universität
Gesamthochschule Paderborn
Fachbereich 17 · Mathematik - Informatik

Entwurf eines generischen Sichtenkonzeptes für die Entwicklungsumgebung Fujaba

STUDIENARBEIT

für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

von

Carsten Reckord
Bergstraße 53
33415 Verl

vorgelegt bei

Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Johannes Magenheim

Februar 2001

i. Eidesstattliche Erklärung

Ich versichere, daß ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe sowie ohne Benutzung anderer als der angegebenen Quellen angefertigt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Paderborn, den 13. Februar 2001

Carsten Reckord

ii.

Inhaltsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Die *Unified Modeling Language* [UML] wurde erstmals 1997 von Grady Booch, James Rumbaugh und Ivar Jacobson vorgestellt. Sie ist eine grafische, objektorientierte Modellierungssprache, die die Spezifikation statischer und dynamischer Zusammenhänge in einem Softwaresystem ermöglicht. Die UML ist aus der Zusammenführung und Harmonisierung der Sprachen OMT [Rum94] von Rumbaugh, OOSE [Jac95] von Jacobson und Booch-Diagrammen [Boo94] von Grady Booch entstanden, die ihre Sprachen 1994-1995 bei der Rational Software Corporation eingebracht und mit der UML eine zu den Ursprungssprachen weitestgehend kompatible Vereinigung der Vorteile der unterschiedlichen Sprachen geschaffen haben.

Insbesondere bei der Spezifikation großer und komplexer Softwaresysteme spielt die UML mittlerweile eine zentrale Rolle. Sie deckt von der Anforderungsanalyse über den Grobentwurf der Architektur bis zur Spezifikation von Funktionen und Algorithmen einen großen Teil der Phasen moderner Softwareentwicklung ab. Damit kann nahezu die gesamte Modellierung eines Softwaresystems in einer einheitlichen, standardisierten Sprache erfolgen. Hierzu stehen mittlerweile eine Vielzahl unterschiedlicher Werkzeuge zur Verfügung, wie Together [Tog01] von TogetherSoft oder Rational Rose [Ros] von der Rational Software Corporation.

In diesem Zusammenhang wurde 1997 im Rahmen einer Diplomarbeit an der Universität GH Paderborn die Arbeit an der Entwicklungsumgebung FUJABA [FNT98] begonnen. Ein Schwerpunkt bei der Entwicklung von Fujaba ist auf die Analyse und Wahrung der Konsistenz in einzelnen Diagrammen und auch zwischen den Diagrammen einer Spezifikation gelegt worden. FUJABA unterstützt neben zahlreichen Diagrammarten der UML wie Statecharts, Klassen- und Aktivitätsdiagrammen auch die Sprache SDL[SDL96], sowie die Sprache »Story Driven Modelling« (SDM, [FNTZ98]), die eine Beschreibung der dynamischen Änderungen von Objektstrukturen durch Kollaborationsdiagramme, sogenannte Storypatterns, ermöglicht, die in erweiterte Aktivitätsdiagramme, die Storydiagramme, eingebettet sind.
Neben der graphischen Modellierung in UML, SDL und SDM bietet Fujaba auch die Möglichkeit zum Round-Trip-Engineering in Java, das heißt zum Generieren von Quellcode aus UML-Diagrammen und umgekehrt. Damit wird zum einen der Implementationsaufwand, insbesondere für Standardstrukturen wie Assoziationen oder Attribute von Klassen, erheblich reduziert. Zum anderen wird so die Konsistenz zwischen Spezifikation und Implementation gewährleistet und die Analyse und übersichtliche Aufarbeitung von undokumentiertem Quellcode vereinfacht. Dies ist besonders bei großen Softwarepaketen im Falle der Wartung oder Weiterentwicklung interessant, da hier häufig Personen eingesetzt werden, die nicht an der Entwicklung

beteiligt waren und daher vor dem Problem stehen, dieses unbekannte Softwarepaket verstehen zu müssen.

Aber gerade bei großen Spezifikationen ergibt sich schnell das Problem, daß die Diagramme zu komplex und unübersichtlich werden. Umfangreiche Diagramme sind daher meist schwerer verständlich und somit auch schwerer zu modifizieren. Insbesondere Klassendiagramme neigen dazu, schnell unübersichtlich zu werden, besonders wenn alle Klassen einer Software-Spezifikation in einem Diagramm dargestellt werden. So besteht etwa das FUJABA-Softwarepaket zur Zeit aus mehr als 1580 Klassen. Ein Diagramm aus allen Klassen ist hier zu unübersichtlich, um für einen der oben genannten Anwendungsfälle von Nutzen zu sein.
Die UML bietet daher die Möglichkeit, komplexe Diagramme in Teildiagramme aufzuteilen, um so einzelne Strukturen geordneter und übersichtlicher darzustellen. Ein Klassendiagramm könnte etwa nach funktionalen Aspekten der Klassen aufgeteilt oder einzelne Klassenhierarchien könnten getrennt dargestellt werden. Teildiagramme, die nach vorgegebenen Kriterien aus einem anderen Diagramm abgeleitet werden und deren Inhalt in Abhängigkeit von diesem Quelldiagramm bestimmt ist, werden auch als *Sichten* auf dieses Diagramm bezeichnet.

1.2 Zielsetzung

Das Ziel dieser Studienarbeit ist die Realisierung eines Sichtenkonzeptes für die Entwicklungsumgebung FUJABA. Dieses Sichtenkonzept soll eine strukturierte Zerteilung komplexer Diagramme wie oben beschrieben ermöglichen und eine einfache Navigation zwischen den definierten Diagrammen und Sichten ermöglichen. Die in Sichten dargestellten Anteile ihrer Quelldiagramme sollen anhand von Regeln in Abhängigkeit von dem Inhalt der Quelldiagramme bestimmt werden. Dazu sind geeignete Strukturen zu entwerfen, die dem Benutzer die Formulierung solcher Regeln zur Festlegung der relevanten Diagrammanteile erlauben.
Bei der Implementierung sind folgende Anforderungen zu erfüllen:

1. **Generisches Konzept**

   Die Implementierung des Sichtenkonzeptes soll so erfolgen, daß dasselbe Konzept auf alle Diagrammarten anwendbar ist.

2. **Transparente Integration**

   Um dem Benutzer eine einfache Handhabung zu ermöglichen, sollen die Sichten den Quelldiagrammen in der Handhabung weitestgehend entsprechen.

3. **Definition und Kombination von Regeln**

   Neue Regeln zur Festlegung der relevanten Diagrammanteile für eine Sicht sollen zur Laufzeit spezifiziert werden können. Ferner soll die Möglichkeit bestehen, Regeln miteinander zu komplexeren Regeln zu kombinieren. Regeln können zudem Eigenschaften besitzen, die ebenfalls zur Laufzeit veränderbar sind.

4. **Strukturierte Darstellung der Sichten**

   Die verschiedenen Sichten sind in einer geeigneten Weise darzustellen, die für den Benutzer die semantische Zusammengehörigkeit von Diagrammen und den dazuge-

hörigen Sichten verdeutlicht und eine einfache Navigation zwischen verschienen Sichten ermöglicht.

1.3 Struktur der Arbeit

In Kapitel 2 wird zunächst allgemein das Konzept von Sichten auf UML-Diagramme erläutert und mit Sichten auf relationale Datenbanken verglichen. Hierbei wird die Implementierungsidee für das Sichtenkonzept entwickelt und vorgestellt.

In Kapitel 3 werden die verschiedenen Möglichkeiten zur Definition von Regeln zur Bestimmung der relevanten Diagrammteile vorgestellt. Hierbei liegt der Schwerpunkt auf der Bereitstellung generischer Benutzerschnittstellen zur Konfiguration komplexerer Regeln und der Möglichkeit der Kombination mehrerer Regeln.

In Kapitel 4 wird die Definition von Sichten mittels der im vorangegangenen Kapitel definierten Regeln erläutert. Insbesondere wird hier auf die Synchronisation zwischen Quelldiagramm und Sicht bei Änderungen am Quelldiagramm eingegangen, sowie auf Möglichkeiten und Grenzen von Diagrammoperationen auf Sichten.

In Kapitel 5 wird die Erweiterung der grafischen Benutzeroberfläche um Komponenten für die Arbeit mit Sichten vorgestellt. Anschließend wird die Verwendung dieser Komponenten und die Arbeit mit dem in dieser Arbeit entwickelten Sichtenkonzept anhand einer Beispielsitzung in FUJABA erläutert.

In Kapitel 6 werden schließlich die Ergebnisse dieser Arbeit zusammengefaßt und in einem Ausblick mögliche Erweiterungen des Konzeptes diskutiert.

1.4 Ein Beispiel

Zunächst sollen die Anforderungen an das in dieser Arbeit zu entwickelnde Sichtenkonzept anhand eines Beispiels verdeutlicht werden. In diesem Beispiel ist ein Haus modelliert, in dem Personen mit einem Fahrstuhl fahren können, um auf die Etagen zu gelangen, in denen sich ihre Wohnung befindet. Abbildung 1 zeigt das Klassendiagramm des Fahrstuhl-Beispiels, das hier verwendet werden soll.

In dem Klassendiagramm sind verschiedene Lebewesen, wie eine Person deren Haustiere Katze und Hund dargestellt. Jedes dieser Lebewesen lebt in einer Unterkunft, was durch die Assoziation *livesIn* zwischen den Klassen *Lifeform* und *Residence* modelliert ist. Eine Unterkunft kann ein Haus sein, oder ein Appartment in einer Etage eines Hauses. Unterkünfte sind ebenso wie die Etagen eines Hauses Plätze, an denen sich Personen aufhalten können. Dies wird durch die Assoziation *isAt* zwischen der Klasse *Person* und der Klasse *Place* modelliert.

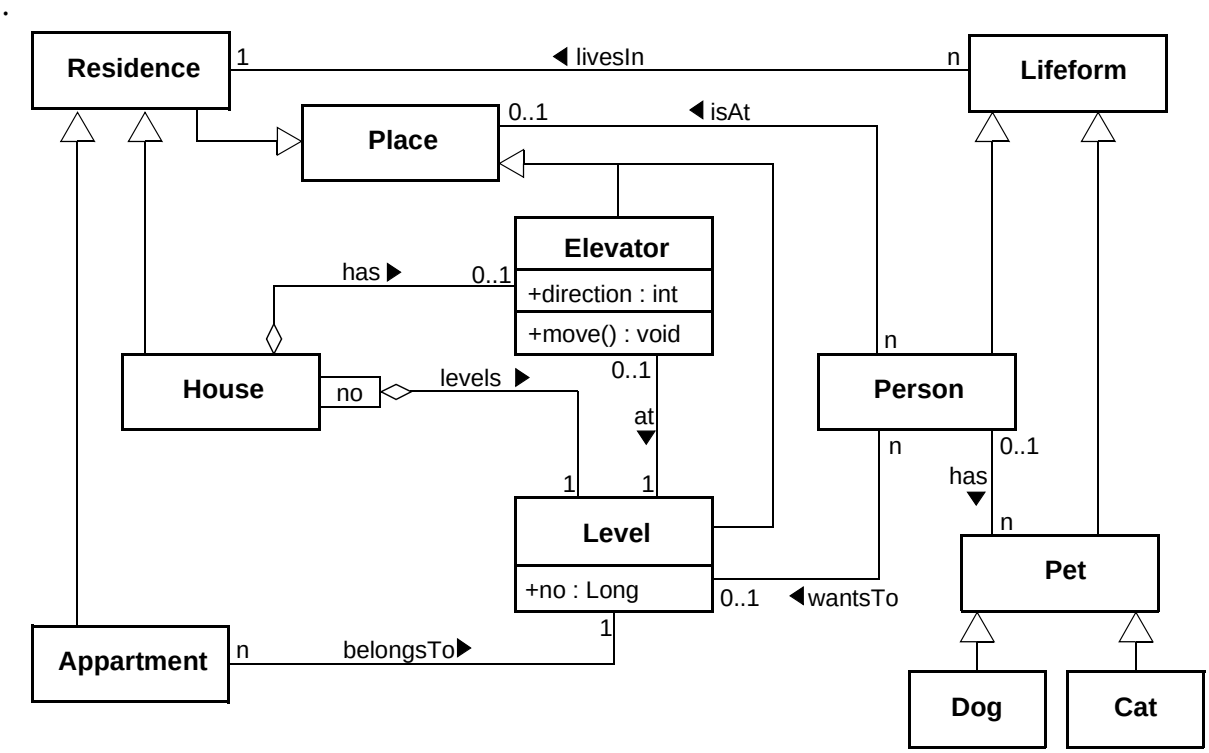

Abbildung 1: Klassendiagramm »Fahrstuhl«

Ein weiterer Platz, an dem sich Personen aufhalten können, ist der Aufzug eines Hauses, der die Etagen des Hauses miteinander verbindet. Dabei befindet er sich zu jedem Zeitpunkt in genau einer Etage des Hauses, was durch die Assoziation *at* zwischen der Klasse *Elevator* und der Klasse *Level* dargestellt ist. Die gewünschte Zieletage wird durch die Assoziation *wantsTo* zwischen *Person* und *Level* festgelegt. Der Aufzug kann durch die Methode *move* in Bewegung gesetzt werden, worauf er sich zur nächsten Etage in der durch das Attribut *direction* angegebenen Richtung bewegt. Nach Erreichen der ersten beziehungsweise letzten Etage eines Hauses wird die Richtung des Aufzuges umgekehrt. Die Funktionsweise der *move*-Methode ist in Abbildung 2 als Storydiagramm dargestellt. Syntax und Semantik von Storydiagrammen sind in [FNTZ98} zu finden.

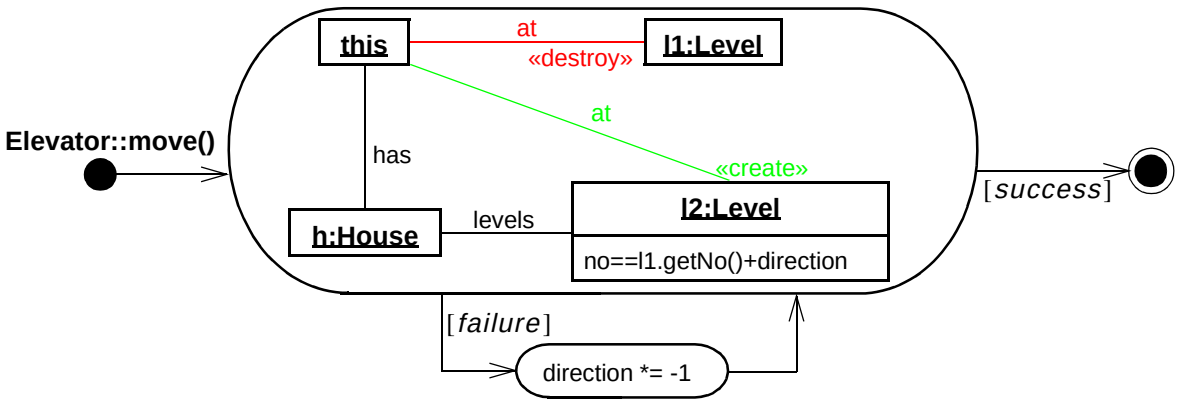

Abbildung 2: Storydiagramm »Elevator::move«

Das in dieser Arbeit zu entwerfende Sichtenkonzept soll es dem Benutzer ermöglichen, zum Beispiel die beiden Klassenhierarchien in dem Klassendiagramm aus Abbildung 1 in unterschiedlichen Sichten darzustellen. Zusätzlich soll es auch möglich sein, etwa die Sicht, die die Klassenhierarchie der Lebensformen darstellt, um die Diagrammelemente zu erweitern, die das Umfeld eines Lebewesens beschreiben. Die resultierende Sicht ist in Abbildung 3 dargestellt.

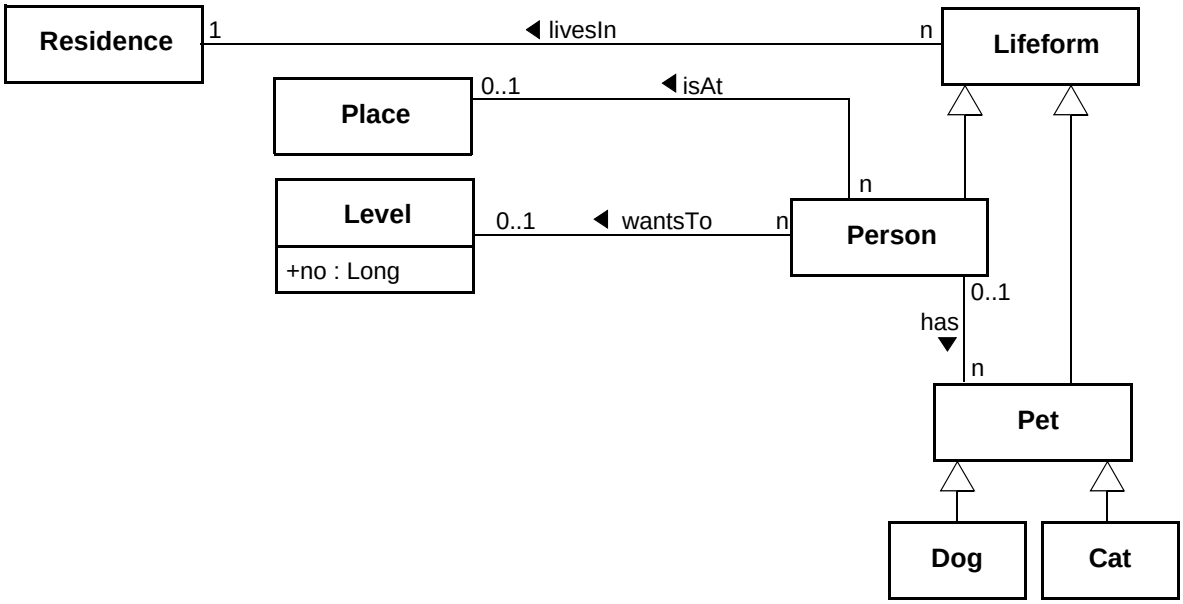

Abbildung 3: Sicht auf die Klassen des Fahrstuhlbeispiels

KAPITEL 2

Das Sichtenkonzept

In Kapitel 1 ist bereits erläutert worden, daß es zum leichteren Verständnis komplexer Diagramme sinnvoll ist, diese in Form mehrerer Teildiagramme darzustellen. Die Aufteilung in Teildiagramme erfolgt dabei nicht willkürlich. Vielmehr sollte jedes Teildiagramm einen bestimmten Teilaspekt des Gesamtmodells darstellen. Aus diesem Grund ist es nicht sinnvoll, den Inhalt dieser Teildiagramme statisch festzulegen, da bei Änderungen am Modell auch der Inhalt der Teildiagramme entsprechend angepaßt werden muß, um weiterhin den entsprechenden Teilaspekt korrekt und vollständig darzustellen. Der Benutzer sollte stattdessen Regeln oder Bedingungen formulieren können, mit denen der Diagramminhalt eines Teildiagramms dynamisch anhand des Inhaltes des Quelldiagramms ermittelt wird. Ein solches Teildiagramm wird als *Sicht* auf das Quelldiagramm bezeichnet.

Ziel dieses Kapitels ist die Herleitung eines Sichtenkonzeptes, das die in Kapitel 1, Abschnitt 1.2 formulierten Anforderungen bezüglich Strukturierung, Generizität und Transparenz erfüllt. Hierzu wird zunächst in Abschnitt 2.1 »Sichten auf relationale Datenbanken« ein bestehendes Sichtenkonzept erläutert, das in relationalen Datenbanken verwendet wird. Anschließend wird in Abschnitt 2.2 »Sichten auf UML-Diagramme« das Sichtenkonzept für UML-Diagramme hergeleitet. Dazu wird zunächst in Unterabschnitt 2.2.1 das allen UML-Diagrammen zugrundeliegende Metamodell beschrieben, das die Basis des zu entwickelnden Sichtenkonzeptes darstellt. In Unterabschnitt 2.2.2 wird schließlich das Sichtenkonzept für UML-Diagramme entwickelt. Hierbei wird auf die Konzepte der Datenbank-Sichten zurückgegriffen und deren Übertragbarkeit auf UML-Diagramme diskutiert. Abschließend werden in Abschnitt 2.3 »Integration in Fujaba« die für das Sichtenkonzept erforderlichen Basisklassen kurz vorgestellt und ihre Integration in das Metamodell beschrieben. Eine genauere Beschreibung der Klassen erfolgt an geeigneter Stelle in den folgenden Kapiteln.

2.1 Sichten auf relationale Datenbanken

Eine relationale Datenbank besteht aus einer Menge von Tabellen, den sogenannten Relationen. Jede Relation besitzt einen eindeutigen Namen und ist eine Ausprägung eines Relationen-Schemas. Ein solches Relationen-Schema besteht aus einer Liste von Attributen und deren zugehörigen Wertebereichen. Jedes Attribut definiert eine Spalte der Relation. Die Gesamtheit der Relationen-Schemata aller in einer Datenbank enthaltenen Relationen wird als Datenbankschema bezeichnet [EN94]. Durch das Datenbankschema wird also das logische Modell der Datenbank beschrieben.

Die in Abbildung 4 dargestellten Relationen sind eine Beispielausprägung des Datenbankschemas *RentDB*, das wie folgt definiert ist

```
RentDB     = (Houses, Appartments, Tenants)
Houses     = (house integer, rent-m2 integer,
              e-kWh double, rad-kWh double)
Appartments = (appt integer, house integer, size integer)
Tenants    = (name char(20), appt integer,
              e-used integer, rad-used integer)
```

Das Datenbankschema besteht aus den Relationenschemata *Houses*, *Appartments* und *Tenants*. Es modelliert eine Datenbank zur Mieterverwaltung. Das Relationenschema *Houses* besteht aus einer eindeutigen Hausnummer, der Quadratmetermiete für die Wohnungen des Hauses, sowie den kWh-Preisen für Strom und Heizkosten. Die Relation *houses* ist eine Ausprägung dieses Schemas. Das Schema *Appartments* besteht aus einer eindeutigen Appartmentnummer, der Nummer des zugehörigen Hauses und der Größe des Appartments. Eine Ausprägung ist in der Relation *appartments* dargestellt. Das Schema *Tenants* beschreibt die Mieter der Appartments durch ihren Namen, das bewohnte Appartment und die verbrauchten kWh Strom und Heizenergie. Die Relation *tenants* ist eine Ausprägung dieses Schemas.

appt	house	size
1	3	550
7	6	730
31	11	55
34	11	80

appartments

name	appt	e-used	rad-used
George W. Bush	1	3467	19683
Bill Gates	7	5321	23761
Hans Meier	31	734	1966
Karl Schulze	34	679	2734

tenants

house	rent-m2	e-kWh	rad-kWh
3	0	0	0
6	130	31,5	3,6
11	11	22,6	5,9

houses

Abbildung 4: Beispielrelationen

Anfragen an eine relationale Datenbank können mittels der relationalen Algebra formuliert werden. Diese Anfragesprache definiert eine Menge von Operationen, die eine oder zwei Relationen als Parameter haben und eine neue Relation als Ergebnis zurückliefern. So gibt es Operationen zur Projektion einer Relation auf bestimmte Attribute, zur Vereinigung oder Differenzbildung zweier Relationen, zur Kombination von Relationen durch ein Kartesisches Produkt, zur Selektion von Tupeln in einer Relation anhand von Bedingungen, sowie zur Umbenennung von Attributnamen. In Erweiterung hierzu können algebraische Operationen und Aggregatfunktionen auf den Tupeln definiert werden. Für eine genaue Beschreibung der relationalen Algebra siehe [EN94].

Da die Formulierung von Anfragen mit Ausdrücken der relationalen Algebra nicht sehr benutzerfreundlich ist, gibt es eine Vielzahl von übersichtlicheren und leichter verständlichen Anfragesprachen, die auf der relationalen Algebra oder alternativen Kalkülen basieren. Im folgenden soll daher die verbreitetste auf der relationalen Algebra basierende Anfragesprache SQL[EN94] verwendet werden. Sie bietet neben oben genannten Operationen viele weitere Operationen zur Formulierung von Anfragen. Außerdem bietet sie neben Anfrageoperationen auch Operationen zur Definition von Relationen-Schemata, zur Modifikation der Daten in der Datenbank und weitere Verwaltungsfunktionen.

Eine Datenbankabfrage in SQL besteht aus drei Klauseln, der **select**-, der **from**- und der **where**-Klausel. Die **select**-Klausel entspricht der Projektion der relationalen Algebra, die **from**-Klausel ermöglicht die Definition eines kartesischen Produktes und die **where**-Klausel entspricht der Selektionsanweisung. Die Umbenennung von Attributen erfolgt durch das Schlüsselwort **as**. Als Beispiel ist in Abbildung 5 das Ergebnis einer Anfrage an die in Abbildung 4 dargestellte Datenbank dargestellt, die durch folgenden SQL-Code definiert ist:

```
select  name, appartments.size as size,
        rent-m2
  from  tenants, appartments, houses
  where tenants.appt=appartments.appt and
        appartments.house=houses.house
```

name	size	rent-m2
George Bush	550	0
Bill Gates	730	130
Hans Meier	55	11
Karl Schulze	80	11

Abbildung 5: Ergebnis der SQL-Anfrage

SQL beziehungsweise. die relationale Algebra ermöglicht also die Auswahl und Zusammenstellung von Daten mehrerer Relationen in einer neuen Relation mittels einer Datenbankanfrage. Eine solche Relation, die nicht Teil des logischen Modells der Datenbank, sondern als *virtuelle Relation* das Ergebnis einer Anfrage ist, wird als *Sicht* bezeichnet. Um einer Datenbank eine Sicht hinzufügen zu können, benötigt die Sicht außerdem, wie jede Relation, einen Namen. Der Befehl zur Definition einer Sicht in SQL sieht daher wie folgt aus:

```
create view <name> as <Abfrageausdruck>
```

Um das Ergebnis aus Abbildung 5 als Sicht mit Namen *rent* zu definieren, lautet die SQL-Anweisung also

```
create view rent as
  select name, appartments.size as size, rent-m2
  from tenants, appartments, houses
  where tenants.appt=appartments.appt and
        appartments.house=houses.house
```

Die in der Sicht enthaltenen Daten werden durch die select-Klausel also aufgrund der Attribute des unterliegenden Schemas, die sie repräsentieren, ausgewählt. Da jedes Attribut im Datenbankschema einen Bestandteileil des durch das Schema ausgedrückten logischen Modells repräsentiert, stellt eine Sicht also einen logischen Teilaspekt des Gesamtmodells dar. Ein Sichtenkonzept, das die Definition solcher »*logischer Sichten*« ermöglicht, wird im Rahmen dieser Arbeit als »*logisches Sichtenkonzept*« bezeichnet.

2.2 Sichten auf UML-Diagramme

2.2.1 Struktur von UML-Diagrammen

UML-Diagramme sind, ebenso wie relationale Datenbanken, Ausprägungen eines zugrundeliegenden Schemas. Dieses Schema, das UML-Metamodell, ist Teil einer 4-Schicht-Metamodellarchitektur, die die UML definiert. Diese Architektur ist in Abbildung 6 dargestellt.

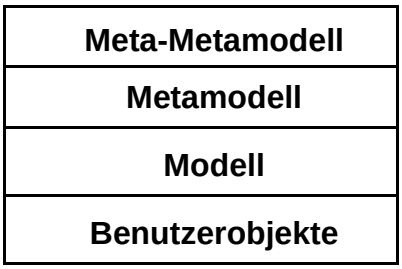

Abbildung 6: Die UML-Metamodell-Architektur

Die erste Schicht dieser Architektur bildet die *Meta-Metamodellebene.* Hier sind Struktur und Semantik der Metamodellebene definiert. Die *Metamodellebene* besteht aus Klassen und Assoziationen zur Beschreibung von UML-Diagrammen der Modellebene. Jedes Metamodell ist eine Instanz eines Meta-Metamodells. Die Modellebene umfaßt die UML-Diagramme aus Benutzersicht. Die Diagrammelemente sind Instanzen der auf Metamodellebene definierten Klassen. Die letzte Ebene ist die Objektebene. Sie enthält konkrete Laufzeitobjekte, die Instanzen der durch den Benutzer auf Modellebene definierten Klassen.

Das UML-Metamodell ist in FUJABA in vereinfachter Form umgesetzt. Abbildung 7 zeigt einen Ausschnitt aus dem Metamodell von FUJABA. Das Fujaba-Metamodell wird auch als *abstrakter Syntaxgraph*, kurz ASG, bezeichnet. Für eine vollständige Beschreibung des ASG und dessen Unterschiede zum UML-Metamodell siehe [FNT98]. Im Folgenden ist mit dem Begriff Metamodell der Fujaba-ASG gemeint.

.

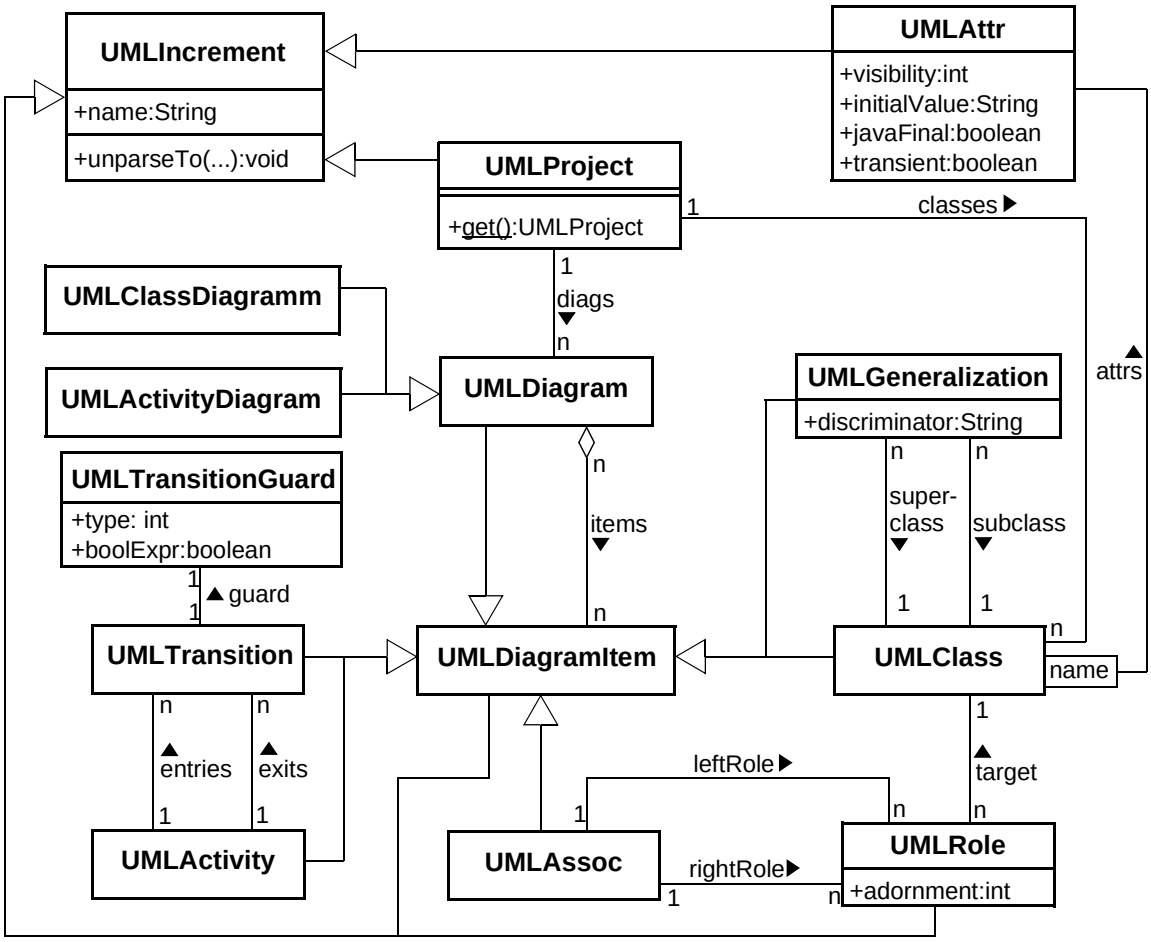

Abbildung 7: Das Metamodell von FUJABA *(Ausschnitt/vereinfacht)*

Die Klasse *UMLDiagram* modelliert ein Diagramm, die höchste Organisationsstruktur in UML. Sie bildet die abstrakte Oberklasse für alle UML-Diagrammarten, wie *UMLClassDiagram* oder *UMLActivityDiagram*. Jedes Diagramm besteht aus einer Vielzahl von Diagrammelementen, wie Klassen, die durch *UMLClass* modelliert werden, oder die verschiedenen Aktivitäten eines Aktivitätsdiagramms, die von der abstrakten Oberklasse *UMLActivity* abgeleitet sind. Die Oberklasse aller Diagrammelemente ist die Klasse *UMLDiagramItem*. Die Zuordnung von Diagrammelementen zu Diagrammen geschieht über die n-zu-n-Aggregation *items* von der Containerklasse *UMLDiagram* zu *UMLDiagramItem. UMLDiagram* selbst ist ebenfalls von *UMLDiagramItem* abgeleitet. Das hat zur Folge, daß ein Diagramm mehrere andere Diagramme enthalten kann. Eine solche Konstruktion ist nach [GHJV95] als »Composite-Pattern« bekannt. Dieses Pattern ist hier leicht modifiziert umgesetzt, da jedes UMLDiagramItem mehreren Diagrammen zugeordnet sein kann. Das Composite-Pattern beschreibt lediglich eine 1-zu-n-Assoziation.

Die Klasse *UMLProject* modelliert ein aus mehreren Diagrammen bestehendes Projekt in Fujaba. Die Diagramme sind einem Projekt durch die *diags*-Assoziation zugeordnet. Dabei ist jedes Diagramm immer nur einem Projekt zugeordnet, da in Fujaba immer nur ein einzelnes

Projekt bearbeitet werden kann. Aus demselben Grund ist die Klasse *UMLProject* als »Singleton« [GHJV95] implementiert. Das heißt, daß es zu jeder Zeit von der Klasse UMLProject höchstens eine Instanz gibt. Auf diese kann von jeder beliebigen Stelle des Systems aus mittels der statischen Methode *get()* zugegriffen werden.

Zur Laufzeit werden UML-Diagramme in Fujaba intern durch Instanzen dieser Metaklassen dargestellt. Jedes Projekt wird durch eine Ausprägung des ASG dargestellt, die aus den Metaklassen-Instanzen für alle Diagramme und Diagrammelemente des Projektes besteht. Im Folgenden sind zwei Beispiele zum besseren Verständnis der Struktur einer ASG-Ausprägung und des Zusammenhangs zwischen Metamodell- und Modellebene aufgeführt.

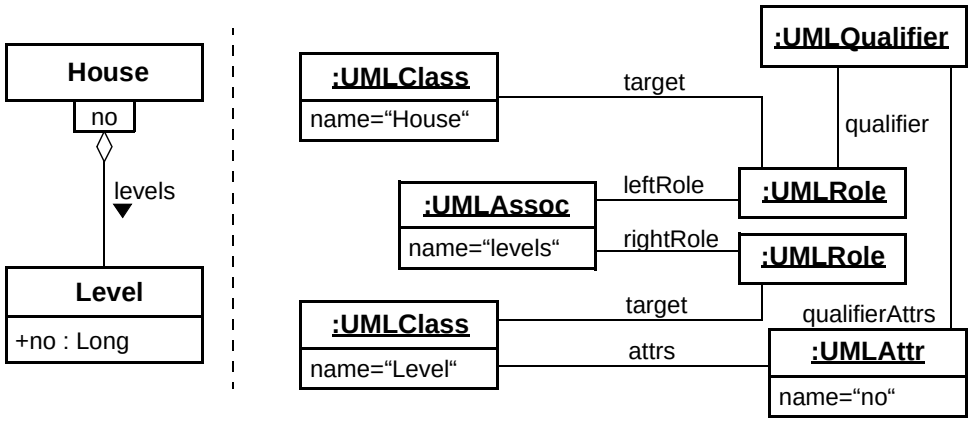

Abbildung 8: Klassendiagramm in FUJABA und interne Darstellung

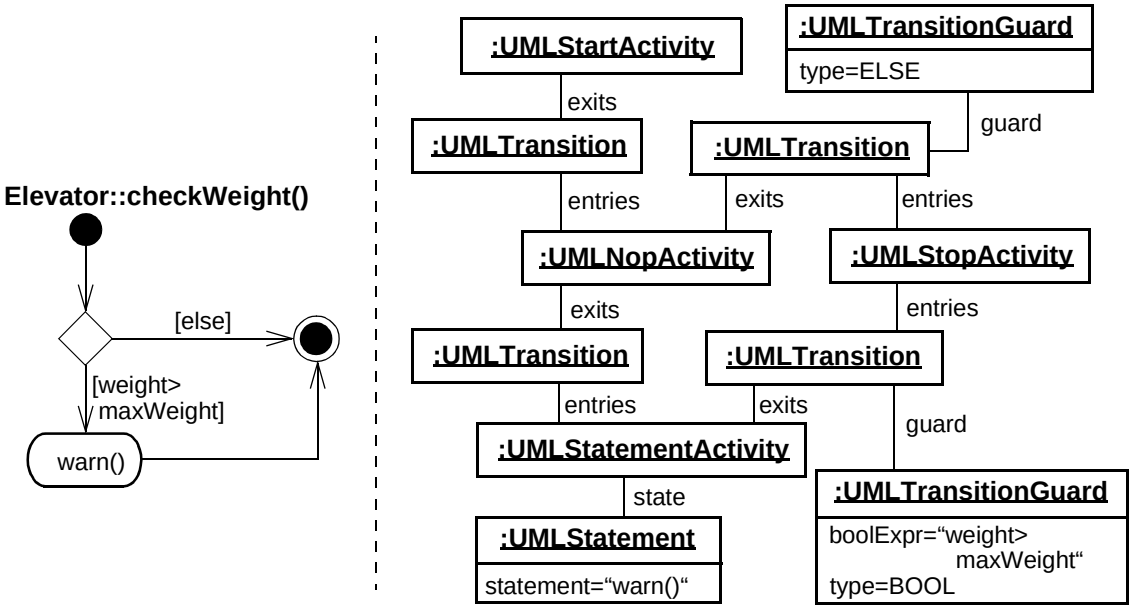

Abbildung 9: Aktivitätsdiagramm in FUJABA und interne Darstellung

2.2.2 Das Sichtenkonzept

Um eine hohe Flexibilität des Sichtenkonzeptes zu gewährleisten, ist es erforderlich, die Schnittstellen der in diesem und den folgenden Kapiteln definierten Klassen des Sichtenkonzeptes allgemein zu halten. Hier sind daher nur die in Abbildung 10 dargestellten Basisklassen des Metamodells zulässig. Entsprechend darf das im Folgenden entwickelte Sichtenkonzept auch nicht auf den Eigenarten einzelner Diagrammarten aufbauen.

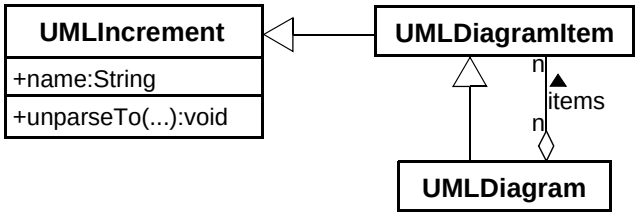

Abbildung 10: Basisklassen

Auf der Basis dieser Informationen ist jedoch die Beschreibung eines logischen Sichtenkonzeptes ähnlich dem von relationalen Datenbanken nicht möglich. Denn zur Definition einer logischen Sicht muß es möglich sein, die Elemente beziehungsweise. Daten nach ihrer logischen Bedeutung innerhalb des gegebenen Modells einzuordnen. In einer relationalen Datenbank wird dies durch die Schemadefinition ermöglicht, die eine Strukturierung der Daten anhand des logischen Modells vorgibt. Die Struktur eines UML-Diagramms ist hingegen durch das Metaschema definiert. Da diese Definition unabhängig von der Bedeutung der Diagrammelemente für das beschrieben Modell ist, läßt sie keine unmittelbare Einordnung der Elemente nach ihrer Bedeutung zu. Auch die Attribute der Klassen aus Abbildung 10 ermöglichen eine solche Zuordnung nicht direkt. Um aus diesen Attributen und den Beziehungen der Diagrammelemente zueinander trotzdem die Modellbedeutung der Elemente ableiten zu können, ist zusätzliches Kontextwissen aus der Domäne des beschriebenen Modells notwendig, das in dem Diagramm nicht enthalten ist. Ein Benutzer, der dieses Wissen hat, kann eine solche Herleitung daher leisten, ein Programm im Allgemeinen nicht. Hierzu ein Beispiel:

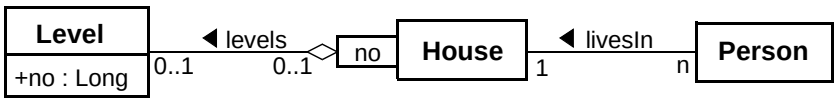

Abbildung 11: Beispiel zu Kontextwissen

Abbildung 11 zeigt einen leicht modifizierten Ausschnitt aus dem Klassendiagramm des Fahrstuhlbeispiels aus Kapitel 1. Will man auf diesem Klassendiagramm eine Sicht definieren, die nur Elemente zur Gebäudebeschreibung enthält, ist hierfür nicht im Diagramm enthaltenes Kontextwissen erforderlich. Um die Modellbedeutung der Klasse *Level* abzuleiten, ist es erforderlich, zu wissen, daß der Wert des Attributes *name* der zugehörigen UMLClass-Instanz eine Etage bezeichnet. Entsprechendes gilt für die Klassen *House* und *Person*. Um eine Zuteilung zu der gewünschten Sicht durchführen zu können, sind weitere Kontextinformationen notwendig. Es muß erkannt werden, daß Etagen und Häuser Elemente der Gebäudebeschreibung im Fahrstuhlbeispiel sind, Personen jedoch nicht.

Da das System diese Leistung nicht erbringen kann, ist ein möglicher Ansatz, den Benutzer die für eine Sicht relevanten Diagrammteile vollständig selbst auswählen zu lassen. Dieser Ansatz widerspricht jedoch der Zielsetzung, durch Sichten nachträglich große und unübersichtliche Spezifikationen übersichtlicher und lesbarer zu machen. Denn um die Diagramme solcher Spezifikationen in Sichten zu unterteilen, müßte der Benutzer diese zunächst verstehen und die Diagrammelemente auf ihre Bedeutung hin analysieren. Dies führt wiederum auf das ursprüngliche Problem - die Unübersichtlichkeit der Diagramme - zurück. Desweiteren wäre ein solcher Ansatz sehr anfällig gegenüber Änderungen an dem Quelldiagramm einer Sicht. Wird ein Diagramm etwa um neue Elemente erweitert, so müssen diese jeweils manuell den entsprechenden Sichten zugeordnet werden. Aus diesem Grund ist ein regelbasierter Ansatz für ein Sichtenkonzept flexibler.

Bei einem regelbasierten Ansatz definiert der Benutzer anstelle einer direkten Auswahl der Diagrammelemente Regeln, die für die Sicht relevante Diagrammteile bestimmen. Dies entspricht der Formulierung eines Anfrageausdrucks bei der SQL-Sichtendefinition. Wie bereits dargestellt, kann mittels solcher Regeln nicht die Auswahl eines logischen Teilaspektes des Modells auf der Basis der im ASG enthaltenen Informationen formuliert werden. Durch die Objekte einer ASG-Ausprägung und ihre Assoziationen zueinander wird jedoch ein Graph definiert, der die strukturellen Beziehungen der Diagrammelemente zueinander beschreibt. Damit können Regeln formuliert werden, die aufgrund der direkten oder indirekten strukturellen Beziehung von Diagrammelementen zueinander die sichtrelevanten Diagrammteile ausgehend von einer Startmenge bestimmen. Hierzu müssen diese Regeln geeignete Traversierungsvorschriften für den durch die ASG-Elemente und ihre Assoziationen definierten Graph enthalten. Die Definition einer Sicht kann dann durch eine Kombination einer beliebigen Anzahl solcher Regeln erfolgen. Dazu zunächst nur ein Beispiel:

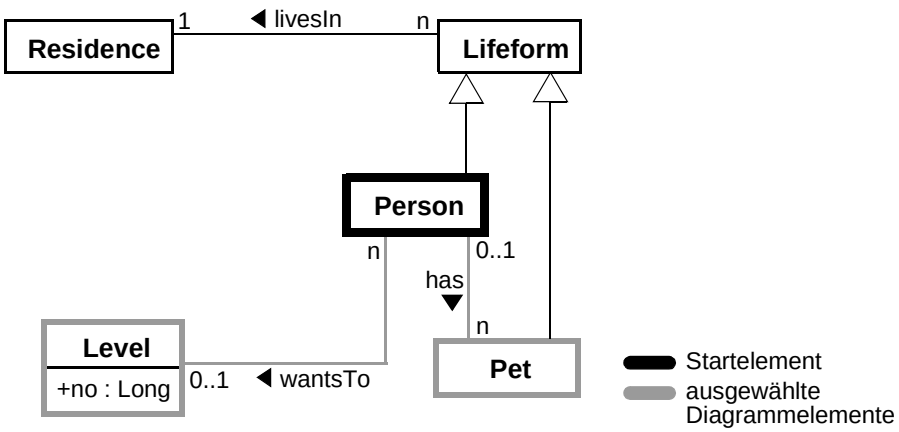

Abbildung 12: Regelbasierte Auswahl von Diagrammelementen

In Abbildung 12 ist das Ergebnis einer Regel dargestellt, die von der Klasse *Person* ausgehend alle Assoziationen dieser Klasse und ihre Assoziationspartner auswählt. Die Traversierung des ASG, auf der diese Regel basiert, ist in Abbildung 13 dargestellt. In Schritt 1 werden jeweils über die Assoziationen *target* und *leftRole* beziehungsweise *rightRole* die Assoziationen

bestimmt, mit denen die Klasse verbunden ist. In Schritt 2 wird dann von der Assoziation ausgehend die Partnerklasse ausgewählt.

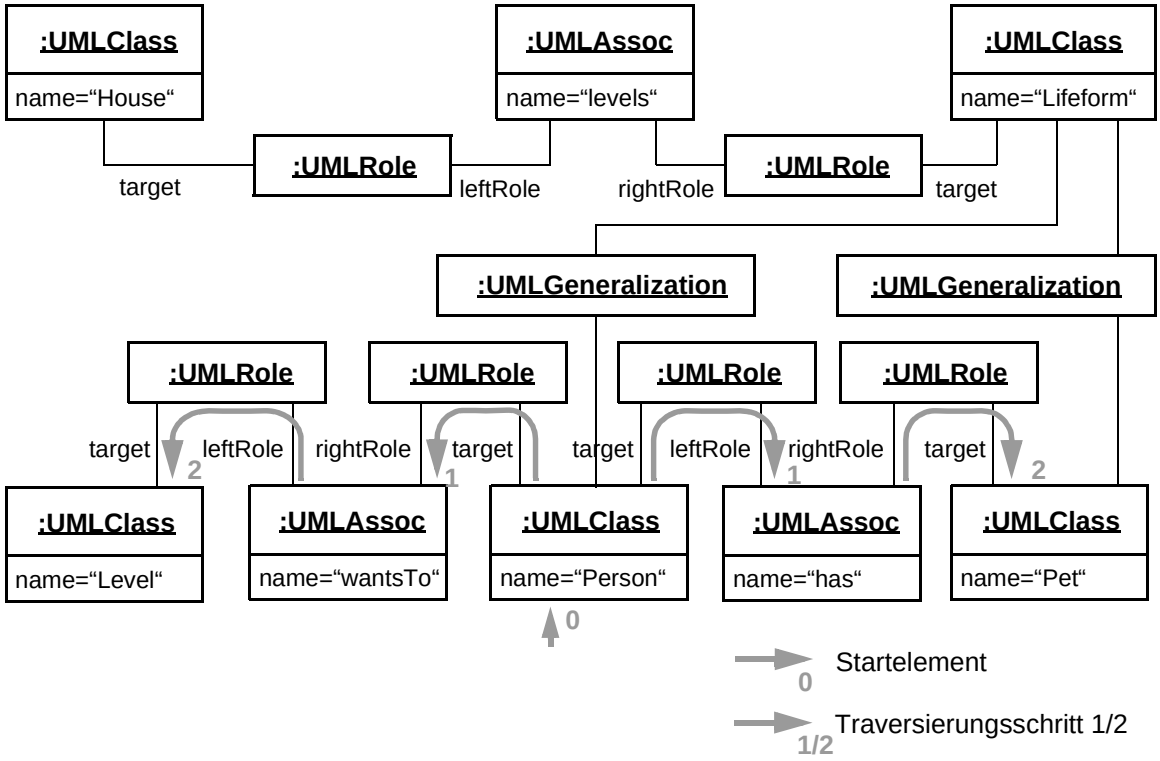

Abbildung 13: Die Traversierungsvorschrift

In dem durch ein derartiges Regelwerk festgelegten Sichtenkonzept erfolgt die Auswahl der für eine Sicht relevanten Diagrammteile unabhängig von ihrer logischen Bedeutung ausschließlich aufgrund der Struktur des Quelldiagramms. Ein solches Sichtenkonzept wird daher im Rahmen dieser Arbeit als »*strukturelles Sichtenkonzept*« bezeichnet. Die so definierten Sichten sind einfache *Teildiagramme* ohne Beschränkung auf einen bestimmten logischen Teilaspekt des Quelldiagramms. Durch die Definition mittels Regeln wird jedoch erreicht, daß ihr Inhalt nicht statisch ist, sondern in Abhängigkeit von ihrem Quelldiagramm dynamisch berechnet wird.

2.3 Integration in FUJABA

Um mit einer Sicht in FUJABA arbeiten zu können, muß diese durch ein entsprechendes Objekt im ASG repräsentiert werden. Dies kann nicht durch eine neue Instanz der Klasse des Quelldiagramms erfolgen. Denn in Fujaba ist jeder Diagrammart eine Analysemaschine zugeordnet, die die entsprechenden Diagramme auf die Einhaltung der statischen Semantik dieser Diagrammart überprüft. Eine Sicht ist aber unter Umständen bezüglich der statischen Semantik ihres Quelldiagrammes fehlerhaft, da erforderliche Diagrammteile fehlen. Daher ist eine Erweiterung des

FUJABA-Metamodells um eine neue Diagrammart zur Darstellung von Sichten erforderlich. Diese Erweiterung ist in Abbildung 14 dargestellt.

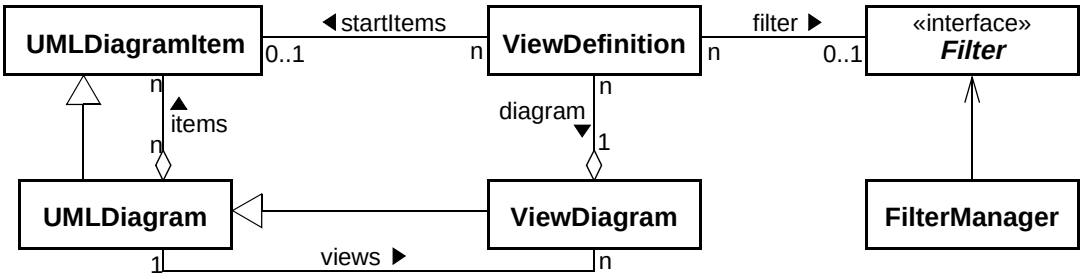

Abbildung 14: Metamodell, erweitert um Sichten

Eine Sicht wird durch die Klasse *ViewDiagram* modelliert. Sie ist über die Assoziation *views* einem Quelldiagramm zugeordnet. Durch die Ableitung der Klasse *ViewDiagram* von *UMLDiagram* ist es zudem möglich, Sichten auf Sichten zu definieren. Dies entspricht dem Composite-Pattern nach [GHJV95]. Außerdem wird hierdurch erreicht, daß auf Sichten dieselben Operationen ausgeführt werden können, wie auf ihren Quelldiagrammen. Dies vereinfacht eine für den Benutzer transparente Integration des Sichtenkonzeptes in die Entwicklungsumgebung, so daß er auf einer Sicht genauso arbeiten kann, wie auf dem entsprechenden Quelldiagramm.

Die Klasse *Filter* kapselt eine Regel zur Sichtendefinition. Auf diese Klasse und die Klasse *FilterManager* wird in Kapitel 3 näher eingegangen. Die zur Anwendung einer Regel erforderlichen Informationen sind in der Klasse *ViewDefinition* gekapselt. Der Inhalt einer Sicht wird durch eine oder mehrere Instanzen dieser Klasse definiert. Dazu werden diese Instanzen mittels der *diagram*-Assoziation an das *ViewDiagram* gebunden. Die Klasse *ViewDefinition* und die Definition einer Sicht durch Instanzen dieser Klasse wird in Kapitel 4 beschrieben.

Zur Darstellung von Diagrammen in Fujaba muß die logische Darstellung, die durch die Instanzen der Metamodell-Klassen gegeben ist, in eine geeignete grafische Darstellung umgesetzt werden. Diese Umsetzung geschieht durch einen Unparsing-Mechanismus, durch den jedem logischen Objekt eine geeignete grafische Darstellung in Form mehrerer *Display-Objekte* zugeordnet wird. Diese Display-Objekte bilden eine Ausprägung des sogenannten *Display-Graphen.* Ihre Erzeugung geschieht in der Methode *unparseTo*, die in der Klasse *UMLIncrement* definiert ist und für alle abgeleiteten Klassen redefiniert wird. Die Aktualisierung der Display-Objekte bei Änderungen an den logischen Objekten geschieht ebenfalls in der unparseTo-Methode.

Da das Unparsing einer Sicht genauso erfolgt, wie das Unparsing jedes anderen Diagramms, ist eine Anpassung des Unparsing-Mechanismus für die Implementierung des Sichtenkonzeptes nicht erforderlich. Allerdings erlaubt die Struktur des Display-Graphen nicht die Darstellung von Sichten auf Diagramme, die in andere Diagramme eingebettet sind, wie dies zum Beispiel bei Storypatterns der Fall ist. Da parallel zu dieser Arbeit die Umstellung von Fujaba auf ein

neues Display-Konzept erfolgt, das unter anderem eine solche Darstellung von „inneren Sichten“ erlaubt, ist im Rahmen dieser Arbeit auf eine umfangreiche Anpassung des Display-Graphen zu diesem Zweck verzichtet worden.

2.4 Zusammenfassung

In diesem Kapitel sind zwei verschiedene Sichtenkonzepte vorgestellt worden, das *logische* und das *strukturelle Sichtenkonzept*.

Das *logische Sichtenkonzept* ermöglicht die Aufteilung eines Modells in Sichten anhand der logischen Bedeutung der Elemente für das Modell. Dieses Konzept ist in relationalen Datenbanken verwirklicht. Die Umsetzung ist hier durch die Definition des Datenbankschemas und der dadurch erfolgenden Strukturierung der Daten nach ihrer Bedeutung möglich.

Für UML-Diagramme ist eine Umsetzung dieses Konzeptes jedoch nicht möglich, weshalb hier stattdessen das *strukturelle Sichtenkonzept* verwendet wird. Dieses Konzept ermöglicht eine Auswahl der relevanten Diagrammteile durch Regeln auf Basis der Diagrammstruktur. Die für die Umsetzung dieses Konzeptes erforderlichen Klassen wurden aus dem Konzept abgeleitet und kurz vorgestellt. Eine genauere Beschreibung ist Inhalt der folgenden Kapitel.

KAPITEL 3

Regeln und Filter

In Kapitel 2 ist dargestellt worden, daß die für eine Sicht relevanten Elemente des Quelldiagramms durch Regeln beschrieben werden sollen. Ferner ist geklärt worden, daß diese Regeln auf Basis von Traversierungsvorschriften für den durch die Elemente des abstrakten Syntaxbaums und ihrer Assoziationen definierten Graph zu definieren sind.

Im Folgenden werden diese Regeln und ihre Umsetzung in geeignete Programmstrukturen beschrieben. Dazu wird zunächst in Abschnitt 3.1 »Graphtraversierung durch Kontextregeln« der Aufbau geeigneter Traversierungsvorschriften auf einem allgemeinen Graphen erläutert. Anschließend wird das dort definierte Verfahren in den Abschnitten 3.2 bis 3.6 zur Definition zunehmend komplexerer Regeln und ihre Implementierung in Unterklassen der Klasse Filter angewendet. In Abschnitt 3.7 »Der FilterManager« wird schließlich eine »Factory«-Klasse vorgestellt, die die verfügbaren Filterklassen verwaltet und Instanzen dieser Klassen zur Verfügung stellt. Abschließend werden die Ergebnisse dieses Kapitels in Abschnitt 3.8 zusammengefaßt.

3.1 Graphtraversierung durch Kontextregeln

Eine einfache Möglichkeit zur Beschreibung einer Traversierungsvorschrift auf einem Graphen ist die Definition einer Nachbarschaftsbeziehung. Ist die direkte Nachbarschaft eines Knotens bestimmbar, so kann durch wiederholtes Ausnutzen dieser Beziehung auf die neu zu der Nachbarschaftsmenge hinzugekommenen Knoten diese Menge erweitert werden. Die Verfolgung der Wege vom Ausgangsknoten zu den Knoten der Nachbarschaftsmenge entspricht dann einer Traversierungsvorschrift auf dem Graphen. Dies wird im folgenden formal definiert.

Sei $G = (V,E)$ mit $E = \{\{u,v\}, u, v \in V\}$ ein beliebiger ungerichteter Graph und $S \subseteq V$ eine Menge beliebiger Startknoten aus V. Dann wird der Kontext von S bestimmt durch die Funktion $k_n : V \rightarrow V$, $n \in \boldsymbol{N}$, die rekursiv definiert ist durch

$$k_1(s) \subseteq V, s \in S$$

$$k_1(S) = \bigcup_{s \in S} k_1(s)$$

$$k_n(S) = k_{n-1}(S) \cup k_1(k_{n-1}(S)), n > 1$$

Damit wird der Kontext von S durch die Abbildungsvorschrift von k_1 bestimmt. k_1 wird als 1-Kontext von S bezeichnet, k_n als n-Kontext. Sei $n' \in \boldsymbol{N}$ die kleinste Zahl mit $k_{n'} = k_{n'+1}$. Dann wird $k_n(S)$ mit $n \geq n'$ als transitive Hülle von G bezüglich S und k_1 bezeichnet.

Sei für das folgende Beispiel die Abbildungsvorschrift von k_1 gegeben durch

$$k_1(S) = \{v \in V, \exists u \in S, \{u, v\} \in E\}$$

k_1 beschreibt die natürliche Nachbarschaft von S. In Abbildung 15 ist ein Beispielgraph mit $S = \{v_0\}$ dargestellt. Die Knoten v_1-v_5 bilden den 1-Kontext, v_6-v_{11} den 2-Kontext und v_{12} den 3-Kontext von S.

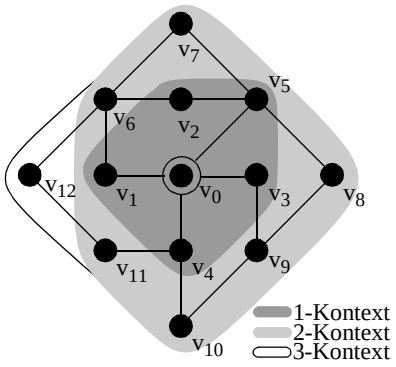

Abbildung 15: Graph mit Kontextmengen

Durch eine Markierungsfunktion $m_i: V \rightarrow \{0, 1\}$ kann zudem die zur Erweiterung des Kontextes von k_{n-1} auf k_n verwendete Menge auf eine Teilmenge von k_{n-1} beschränkt werden. Dazu ist die Definition von k_n wie folgt anzupassen:

$$k_n(S) = k_{n-1}(S) \cup k_1(S'), S' = \{s \in k_{n-1}(S), (m_{n-1}(s) = 1)\}$$

Dadurch kann unter anderem erreicht werden, daß zur Kontexterweiterung k_1 nicht mehr auf alle Elemente von k_{n-1} angewandt werden muß, sondern nur auf die in k_{n-1} neu hinzugekommenen. Dazu ist m_i zu definieren als

$$m_i(s) = \begin{cases} 1, s \in k_i(S) \wedge s \notin k_{i-1}(S) \\ 0, sonst \end{cases}$$

Für die im Folgenden beschriebene Implementierung von Regeln bringt dies neben einer Performancesteigerung auch die Möglichkeit, in m_i weitere Bedingungen zur Traversierung zu definieren.

3.2 Einfache Filter

Durch die Definition der Abbildungsvorschrift eines 1-Kontextes und die Anwendung der rekursiven Definition des n-Kontextes kann also eine Traversierungsvorschrift für einen Graphen beschrieben werden. Im Folgenden sei dieser Graph definiert durch die Instanzen der Metaklassen des ASG und ihre Assoziationen zueinander. Eine Regel wird dann definiert durch die Abbildungsvorschrift des 1-Kontextes auf diesem Graphen und eine zugehörige Markierungsfunktion. Die Abbildungsvorschrift wird im Folgenden als *1-Regel* bezeichnet

Regeln sind in FUJABA durch die Interfaceklasse *Filter* (Abbildung 16) gekapselt. Sie stellt die Methoden *get1Context* und *getNContext* zur Verfügung, die den 1-Kontext beziehungsweise n-Kontext der gekapselten Regel zurückliefern. Wird der Methode getNContext als zu berechnender Kontext der Wert -1 übergeben, so wird die transitive Hülle über die gegebene Regel berechnet. Die Methode grow1Context definiert die 1-Regel. Als Parameter hat sie das Diagramm, auf dem der Kontext bestimmt werden soll, das Diagrammelement *item*, für das der 1-Kontext bestimmt werden soll, sowie den bisherigen Kontext *context*. Durch Ausführung von *grow1Context* werden die Elemente aus dem 1-Kontext von *item* in *context* eingefügt. Der Rückgabewert von *grow1Context* ist eine Menge von Startelementen für eine nachfolgende Kontexterweiterung. Durch diese Menge ist also die Markierungsfunktion aus Abschnitt 3.1 bestimmt. Zusätzlich besitzt jeder Filter einen Namen und stellt über die Methode *getDescription* eine textuelle Kurzbeschreibung seiner Funktion zur Verfügung. Durch die Methode *isForDiagram* kann festgestellt werden, ob ein *Filter* eine Regel für ein gegebenes Diagramm definiert. Die Klasse *AbstractFilter* stellt Standardimplementierungen für die Methoden *get1Context* und *getNContext*, sowie für die Zugriffsmethoden auf das Attribut *name* zur Verfügung. Zur Berechnung des Kontextes wird in get1Context und getNContext die abstrakte Methode grow1Context aufgerufen, die von Unterklassen mit einer Implementierung ihrere 1-Regel zu überschreiben ist. Dieses Vorgehen entspricht dem »Template Method«-Pattern aus [GHJV95]. Um eine einfache und generische Speicherung zu ermöglichen, ist die Klasse *AbstractFilter* zudem von der Klasse *BasicIncrement* abgeleitet. In dieser Klasse ist ein generisches Speicherverfahren implementiert, das eine Speicherung der Objekteigenschaften ermöglicht

Die Umsetzung des Filters durch eine Interfaceklasse und eine abgeleitete abstrakte Klasse mit Standardimplementierungen für die Methoden der Interfaceklasse entspricht der Umsetzung vieler Standardschnittstellen des Java Runtime Environment. Sie ermöglicht die Umgehung von Mehrfach-Vererbungsproblemen in Java. So wird eine höhere Flexibilität bei der Implementierung neuer Filter erreicht.

.

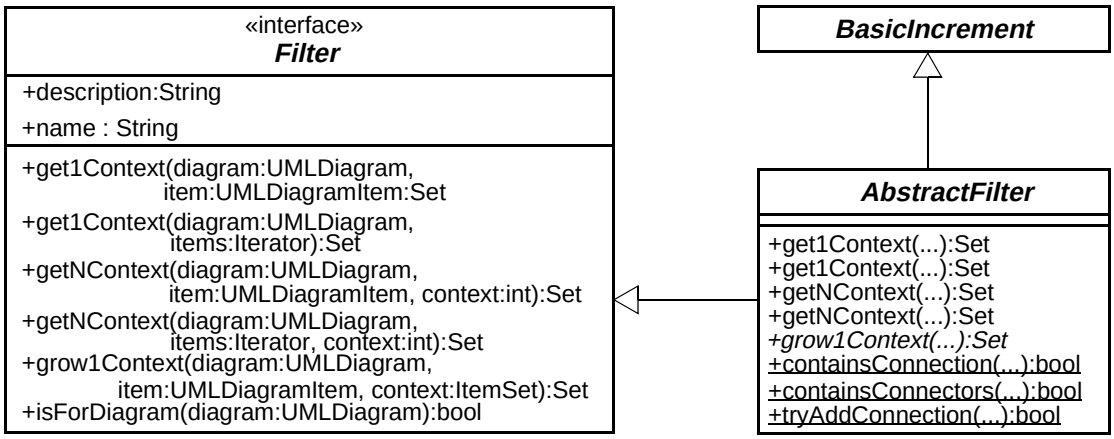

Abbildung 16: Die Klassen Filter und AbstractFilter

Durch die Einführung der Klasse *UMLConnection* aus Abbildung 17 als abstrakte Oberklasse aller Metamodell-Klassen, die Kanten in einem UML-Diagramm modellieren, ist es möglich, diagrammart-unabhängige Operationen zur Aufnahme von Kanten aus UML-Diagrammen in Sichten zu definieren. Diese sind als statische Methoden in der Klasse *AbstractFilter* implementiert (siehe Abbildung 16) und erlauben die Vereinfachung der Implementierung von *grow1Context*. Die Methode *containsConnectors* überprüft ein gegebenes Diagramm oder eine gegebene Menge von *UMLDiagramItems* darauf, ob sie *source*- und *targetConnector* einer *UMLConnection* enthalten. *containsConnection* überprüft entsprechend das Enthaltensein einer *UMLConnection* und ihrer Endobjekte. Die Methode *tryAddConnection* fügt ein Objekt vom Typ UMLConnection und die beiden mit ihr verbundenen UMLDiagramItem-Objekte in eine gegebene Kontextmenge ein, falls mindestens eines dieser drei Objekte bereits in der Menge enthalten ist.

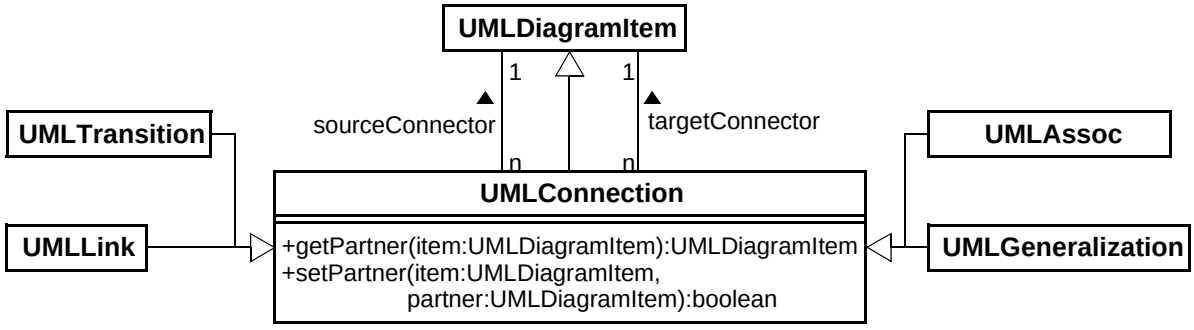

Abbildung 17: Die Klasse UMLConnection

Um eine neue Regel zu definieren, ist lediglich die Ableitung einer neuen Filterklasse von AbstractFilter notwendig. Zur vollständigen Beschreibung der Regel ist nur die Methode *grow1Context* zu redefinieren und im Methodenrumpf die Abbildungsvorschrift für den 1-Kontext zu formulieren. Dadurch ist eine sehr einfache und kompakte Implementierung neuer Filter möglich. Dies kann etwa in Fujaba durch Aktivitätsdiagramme oder Kollaborationsdiagramme

erfolgen. Um eine noch einfachere Modellierung durch Storydiagramme in Fujaba zu ermöglichen, ist in der Klasse *ItemSet* die Schnittstelle der Java-Standardklasse *Set* um eine Referenz auf die Klasse *UMLDiagramItem* mit den entsprechenden Standardzugriffsmethoden nach dem Fujaba-Styleguide erweitert. Erst dadurch ist es in Fujaba möglich, diesem Container in einem Storypattern Objekte vom Typ UMLDiagramItem hinzuzufügen.

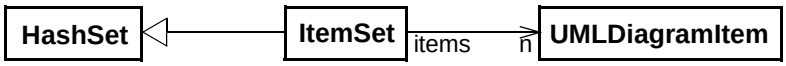

Abbildung 18: Die Klasse ItemSet

In Abbildung 19 ist beispielhaft die Methode *grow1Context* eines Filters dargestellt, der die Auswahl von Klassen eines Klassendiagramms entlang der Vererbungshierarchie in Richtung der Subklassen ermöglicht.

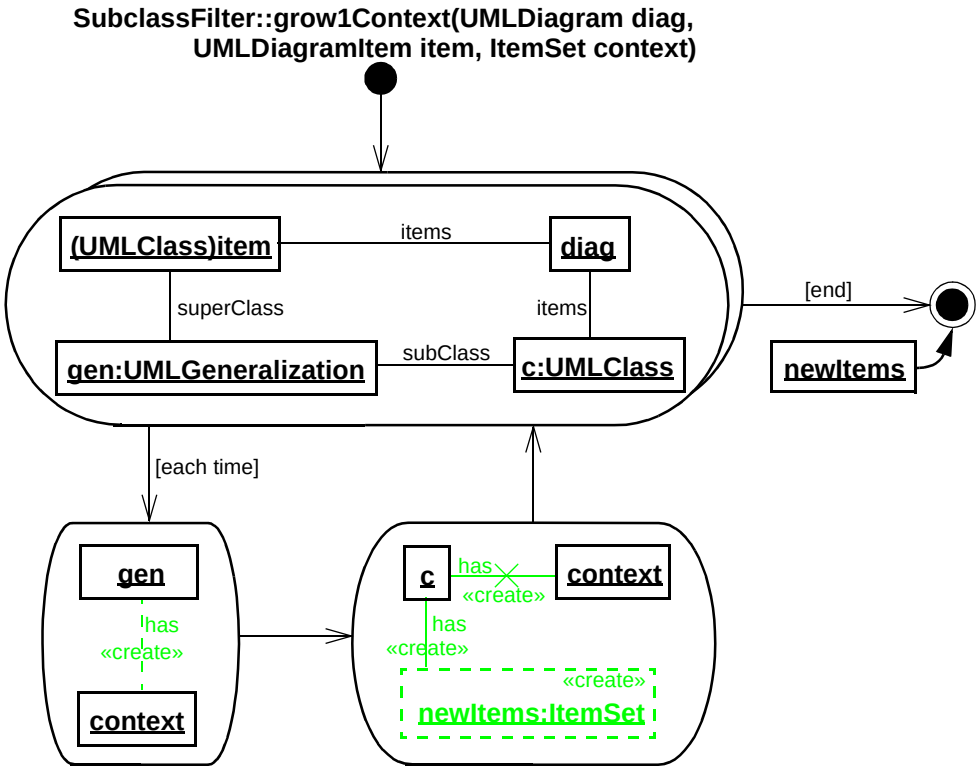

Abbildung 19: Regel als Storydiagramm

3.3 Parametrisierte Filter

In Abschnitt 3.2 wurde der grundlegende Aufbau eines Filters vorgestellt. Dieser Aufbau ermöglicht die statische Definition von Regeln zur Kontextbildung in der Methode *grow1Context*.

Da es viele Anwendungsfälle gibt, in denen die benötigten Regeln zur Sichtendefinition sich nur geringfügig unterscheiden, ist es wünschenswert, diese Fälle mit einer einzelnen Filterklasse

abzudecken, anstatt für jeden Spezialfall eine neue Klasse zu definieren. So müßten zum Beispiel für die Verfolgung von Vererbungsbeziehungen analog zu dem durch Abbildung 19 definierten SubclassFilter ein SuperclassFilter und ein SubAndSuperclassFilter definiert werden. Es ist wünschenswert, diese drei Klassen durch eine Klasse InheritanceFilter zu ersetzen. Dazu ist eine Parametrisierung der 1-Regel erforderlich. Die Parameter der 1-Regel sind dafür als Attribute der Filterklasse zu definieren. Im Falle des *InheritanceFilters* könnten dies etwa zwei boolean-Attribute *includeSubclasses* und *includeSuperclasses* sein. Um dem Benutzer zur Laufzeit die Anpassung dieser Attribute zu ermöglichen, muß ein solcher parametrisierter Filter zudem eine geeignete Benutzerschnittstelle zur Verfügung stellen.

Die Basisklasse für parametrisierte Filter ist die Interfaceklasse *ConfigurableFilter*. Sie stellt über die Assoziation *editor* eine Instanz der Interfaceklasse *FilterEditor* zur Verfügung, die eine geeignete Benutzerschnittstelle zusammen mit einem einheitlichen Interface zum Zugriff auf diese Benutzerschnittstelle verkapselt. Dazu sind die Methoden *getEditorPanel*, die die Benutzerschnittstelle in Form einer grafische Komponente vom Typ *JPanel*[1] zurückgibt, sowie die Methoden *getNewFilter*, *getValues* und *setValues* zur Synchronisation von Benutzerschnittstelle und Filterparametern definiert. Die Methode *setValues* initialisiert die Benutzerschnittstelle mit den Werten der Filterparameter. Die Methode *getValues* liest die aktuellen Werte für die Parameter aus der Benutzerschnittstelle aus und aktualisiert die Werte der Parameter entsprechend. Die Methode *getNewFilter* liefert eine neue Instanz des bearbeiteten Filters mit den durch die Benutzerschnittstelle definierten Werten. Um dies zu ermöglichen, implementiert *ConfigurableFilter* das Interface *Cloneable*.

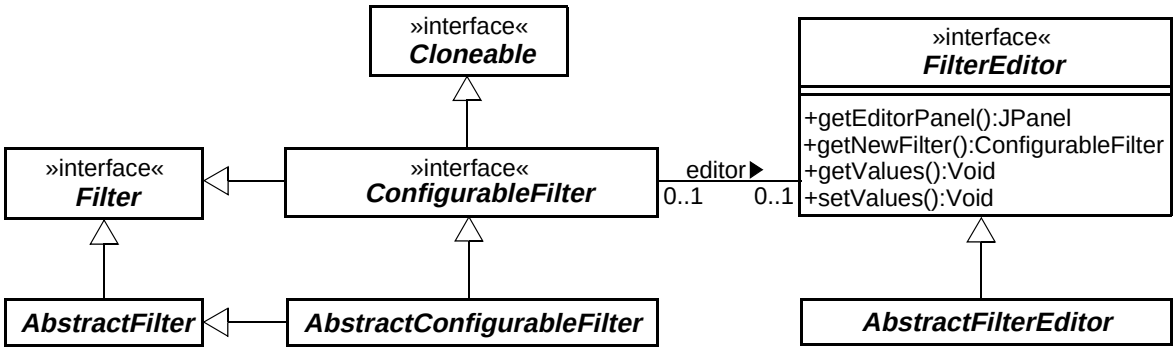

Abbildung 20: Die Klassen ConfigurableFilter und FilterEditor

AbstractConfigurableFilter und AbstractFilterEditor sind von den Interfaces abgeleitete abstrakte Klassen, die Standardimplementierungen der meisten Methoden der Interfaces zur Verfügung stellen.

1. Grafische Flächenkomponente aus der *Swing*-Bibliothek des *Java Runtime Environment*

3.4 Der generische Filtereditor

Die Implementierung einer grafischen Benutzerschnittstelle ist im Allgemeinen eine umfangreiche und komplexe Aufgabe. Müßte eine solche Schnittstelle für jeden neuen *ConfigurableFilter* mitimplementiert werden, wäre eine einfache Definition neuer Filterregeln nicht mehr gegeben. Eine Modellierung neuer komplexer Filterklassen in Fujaba wäre kaum zu leisten. Daher liefert die Standardimplementierung der Methode *getEditor()* in *AbstractConfigurableFilter* eine Instanz der Klasse *DefaultFilterEditor* zurück. Diese Klasse stellt mit der Methode *getEditorPanel* einen auf Basis der Attribute des Filters generisch erzeugten Editor zur Verfügung.

Die generische Erzeugung des Editors geschieht dabei mit Hilfe der Java Core Reflection API [Ref98] und der JavaBeans-Technologie [Bea97].

3.4.1 Die Core Reflection API

Die *Core Reflection API* ist eine „kleine und sichere API, die die interne Betrachtung von Klassen und Objekten der Java Virtual Machine unterstützt“ (übersetzt aus [Ref98]). Die Core Reflection API bietet eine Schnittstelle für die Abfrage von Informationen über Klassen und die darin definierten Felder und Methoden. Zur Laufzeit kann in einem konkreten Objekt durch einen Aufruf der Methode *getClass* auf eine Instanz der Klasse *Class* zugegriffen werden. Diese ermöglicht den Zugriff auf alle für das Objekt definierten Felder, Methoden und Konstruktoren in Form von Instanzen der Klassen *Field*, *Method* und *Constructor*. Die Klasse *Field* gibt Auskunft über Namen und Typ eines Feldes und ermöglicht die Abfrage und Änderung des Feldwertes für ein gegebenes Objekt. Analog kann mit Hilfe der Klasse *Method* die Signatur der jeweiligen Methode abgefragt und die Methode auf einem konkreten Objekt aufgerufen werden. Die Klasse *Constructor* erlaubt entsprechend die Erzeugung neuer Instanzen der definierenden Klasse. Instanzen dieser Klassen der Java Reflect API können nicht direkt erzeugt werden, sondern werden von der Java Virtual Machine beim Laden der Klassen angelegt.

3.4.2 Die JavaBeans-Technologie

Die *JavaBeans* Technologie ist eine Sammlung von Klassen und Standardtechniken für die komponentenbasierte Softwareentwicklung, insbesondere zur Einbindung von grafischen Komponenten. Als *Komponente* wird in diesem Zusammenhang ein abgeschlossenes, wiederverwendbares Paket aus einer oder mehreren Klassen bezeichnet. Eine Komponente, eventuell ergänzt um weitere Klassen, die Informationen zu der Komponente bereitstellen, wird als *Bean* bezeichnet. Die Bean-API der Java Plattform stellt eine Vielzahl von Klassen zur Einbindung von Beans in ein Softwaresystem zur Verfügung. Diese ermöglichen die Benutzung von Beans ohne direkten Zugriff auf deren Schnittstellen, was die dynamische Zusammenstellung und Benutzung von Beans zur Laufzeit erlaubt. Die Zugriffe auf die Schnittstellen des Beans erfolgen in den Klassen der Bean-API durch geeignete Mechanismen aus der Reflection-API.

Die Klasse *BeanInfo* ist die Ausgangsklasse für Informationen über ein Bean. Sie erlaubt den Zugriff auf Instanzen von Klassen, die detailliertere Informationen zur Verfügung stellen. Dies sind vor allem der *BeanDescriptor* und eine Liste von *PropertyDescriptoren*. Der *BeanDescriptor* bietet allgemeine Informationen über das Bean, wie den Namen und, sofern vorhanden, einen *Customizer*. Ein *Customizer* ist ein speziell für das Bean geschriebenes grafisches Benutzerinterface. Zusätzlich ermöglicht der *BeanDescriptor* das Auslesen und Setzen von Attributen des Beans über den Attributnamen. Attribute sind in der Bean-API definiert durch eine Zugriffsmethode der Form *get<Attributname>()* und eine optionale *set*-Methode. Alternativ kann auf Attribute des Beans auch mit den von der *BeanInfo* zurückgelieferten *PropertyDescriptor*-Instanzen zugegriffen werden. Ein *PropertyDescriptor* liefert Informationen über ein Attribut, wie den Typ des Attributes und die zugehörigen Zugriffsmethoden. Damit erlaubt er den direkten Zugriff auf Reflect-Ebene. Zusätzlich stellt er einen *PropertyEditor* zur Verfügung. Die Klasse *PropertyEditor* enthält verschiedene Methoden zur Darstellung und Änderung des Attributwertes auf einer höheren Ebene als *PropertyDescriptor*. Unterklassen dieser Klasse sind daher in der Regel auf einen bestimmten Attributtyp beschränkt. Sie ermöglichen das direkte Lesen oder Setzen des Attributwertes, das Lesen und Setzen als Text, sowie eine Methode zur grafischen Darstellung. Zusätzlich kann ein *PropertyEditor* eine geeignete GUI-Komponente zum Edieren des Attributwertes, einen sogenannten *CustomEditor*, zur Verfügung stellen. *PropertyEditoren* für verschiedene Klassen werden durch statische Methoden bei der Klasse *PropertyEditorManager* registriert und sind dort anhand des Attributtyps abfragbar. Standardmäßig bietet Java *PropertyEditoren* für die Standarddatentypen *boolean*, *double*, *int* und *long* und die Klasse *String*. Neben den Methoden zur Manipulation des Attributwertes bietet die Klasse *PropertyEditor* zudem die Möglichkeit der Überwachung des Attributwertes durch ein Listener-Konzept vergleichbar mit dem »Observer-Pattern« aus [GHJV95]. Instanzen der Klasse *PropertyChangeListener* können sich bei dem *PropertyEditor* registrieren und werden bei jeder Änderung des Attributwertes durch eine Instanz der Klasse *PropertyChangeEvent* über den alten und neuen Wert informiert.

Die Klasse *Introspector* stellt durch die Methode *getBeanInfo(Class beanClass, ...)* alle verfügbaren Informationen über ein Bean in Form einer Instanz der Klasse *BeanInfo* zur Verfügung. Der Bean-Entwickler kann diese Informationen explizit zur Verfügung stellen, indem er dem Bean-Paket eine Unterklasse von *BeanInfo* mit dem Namen *<Name der Beanklasse>BeanInfo* hinzufügt. Falls eine solche Klasse nicht im Klassenpfad gefunden wird, wird ein neues *BeanInfo*-Objekt durch Analyse des Beans mit Reflect-Methoden erzeugt.

3.4.3 Der generische Editor

Auf Basis der JavaBean-Technologie kann zur Laufzeit eine Editor-Komponente für einen parametrisierten Filter zusammengestellt werden. Dazu wird durch einen Aufruf der Methode *getBeanInfo* der *Introspector*-Klasse ein *BeanInfo* für den Filter erzeugt. Falls dieses *BeanInfo* in seinem *BeanDescriptor* einen *Customizer* zur Verfügung stellt, wird dieser als Editor benutzt.

Steht kein *Customizer* zur Verfügung, so werden die *PropertyEditoren* der im *BeanInfo* zur Verfügung gestellten *PropertyDescriptor*-Instanzen zur Zusammenstellung eines Editors benutzt. Dazu wird zunächst für jedes Attribut des Filters geprüft, ob der zugehörige *PropertyDescriptor* explizit einen *PropertyEditor* angibt. Ist dies nicht der Fall, so wird im *PropertyEditorManager* nach einem geeigneten *PropertyEditor* gesucht. Bleibt auch dies erfolglos, so wird für dieses Attribut lediglich der aktuelle Wert als Text dargestellt und kann nicht ediert werden. Wird ein *PropertyEditor* gefunden, so wird dafür eine geeignete Editorkomponente ermittelt. Dazu wird zunächst überprüft, ob der *PropertyEditor* durch die Methode *getCustomEditor()* eine solche Komponente zur Verfügung stellt. Ist dies nicht der Fall, so werden die Methoden *getAsText()* und *setAsText(String value)* des PropertyEditors benutzt, um das Attribut durch ein Textfeld darzustellen. Handelt es sich um eine sogenannte *tagged property*, kann das Attribut also nur eine Menge von Werten annehmen, die durch die Methode *getTags* des *PropertyEditors* definiert sind, so wird statt des Textfeldes eine ComboBox mit den möglichen Werten benutzt. Die so erhaltene Komponente wird dann zusammen mit einem Label, das den Namen des Attributs enthält, dem Editor hinzugefügt. Abbildung 21 zeigt als Beispiel einen Editor für die nebenstehende Filterklasse.

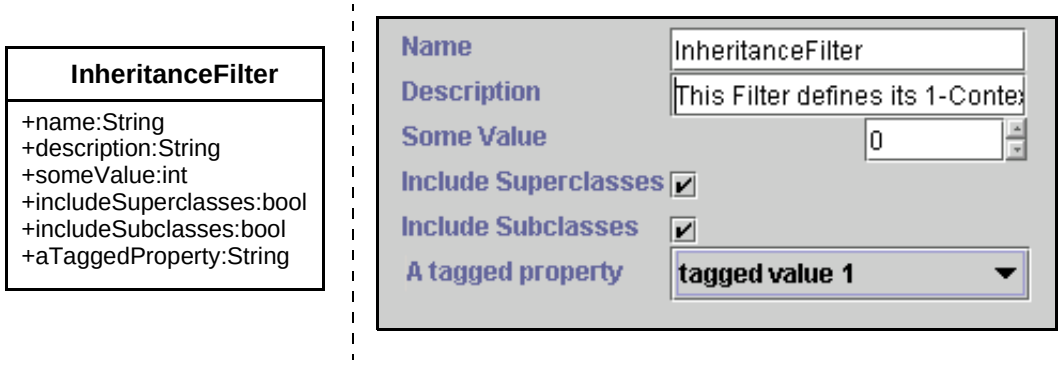

Abbildung 21: Klasse und zugehöriger automatisch generierter Editor

Das vorgestellte Verfahren für die automatische Zusammenstellung eines generischen Editors ist für die meisten Anwendungsfälle ausreichend. Falls es nicht ausreicht, bietet das Bean-Konzept dem Benutzer unterschiedliche Möglichkeiten, verschiedene Teile des Editors selbst zu definieren. Um weitere Attributklassen zu unterstützen, kann der Entwickler entsprechende *PropertyEditoren* mit *CustomEditor*-Komponenten oder *set-/getAsText()*-Funktionen definieren. Diese können dann beim *PropertyEditorManager* registriert werden und werden anschließend automatisch bei der Zusammenstellung des Editors benutzt. Die Registrierung kann etwa beim Laden der Filterklasse durch einen sogenannten *Klassenkonstruktor* geschehen. Alternativ kann auch eine vollständige Benutzerschnittstelle implementiert und als *Customizer* einer entsprechenden *BeanInfo*-Klasse zur Verfügung gestellt werden.

3.5 Standardfilter

Im Rahmen dieser Arbeit sind vier Standardfilter für Storypattern, Aktivitäts- und Klassendiagramme implementiert worden.

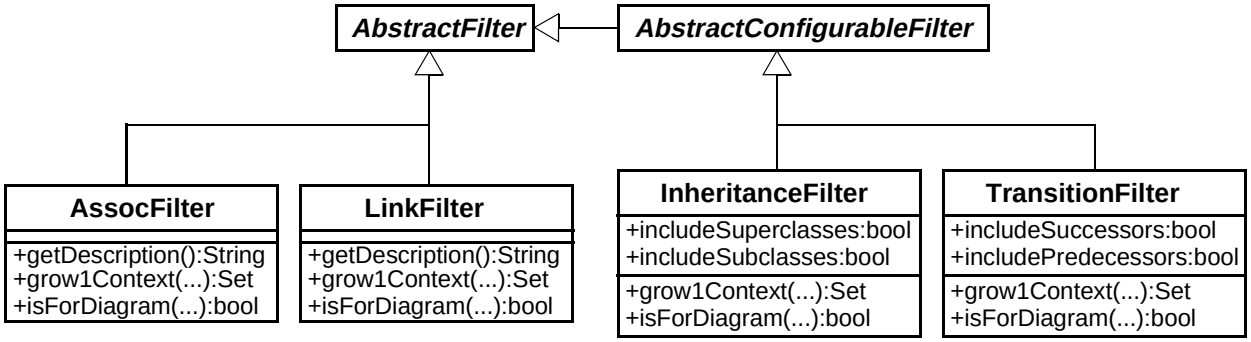

Abbildung 22: Die Standardfilter

Die Klasse *AssocFilter* definiert eine Regel zur Kontext-Erweiterung entlang von Assoziationen zwischen den Klassen eines *Klassendiagramms*. In der Klasse *InheritanceFilter* ist eine Regel definiert, die die Verfolgung von Vererbungskanten in Abhängigkeit von den Attributen *includeSubclasses* und *includeSuperclasses* erlaubt. Die in der Klasse *LinkFilter* definierte Regel beschreibt die Verfolgung von Objektlinks in einem StoryPattern. Und die Klasse *TransitionFilter* ermöglicht die Verfolgung von Transitionen in einem Aktivitätsdiagramm in Abhängigkeit von den Attributen *includeSuccessors* und *includePredecessors*. Die Storydiagramme zu diesen Regeln sind in Anhang D dargestellt.

3.6 Der CompositeFilter

In Abschnitt 3.2 und Abschnitt 3.3 sind Filter vorgestellt worden, die einfache statische oder parametrisierte Regeln zur Graphtraversierung beschreiben. Die so definierten Filter sind relativ kompakt und einfach zu implementieren, aber im allgemeinen auch trotz Parametrisierung auf ein bestimmtes Problem, etwa die Darstellung einer Klassenhierarchie, beschränkt. Es ist daher wünschenswert, mehrere Filter durch Mengenoperationen zu einem komplexeren Filter zu verknüpfen.

Die Klasse *CompositeFilter* erlaubt die Definition einer beliebig komplexen Regel durch ein Netzwerk aus Verknüpfungsoperatoren und Filtern. Dieses Netzwerk kann in dem *FilterEditor*

des *CompositeFilters* als Diagramm spezifiziert werden. Dazu ist dem *CompositeFilter* durch die Assoziation *diagram* eine Instanz der Klasse *CFDiagram* zugeordnet.

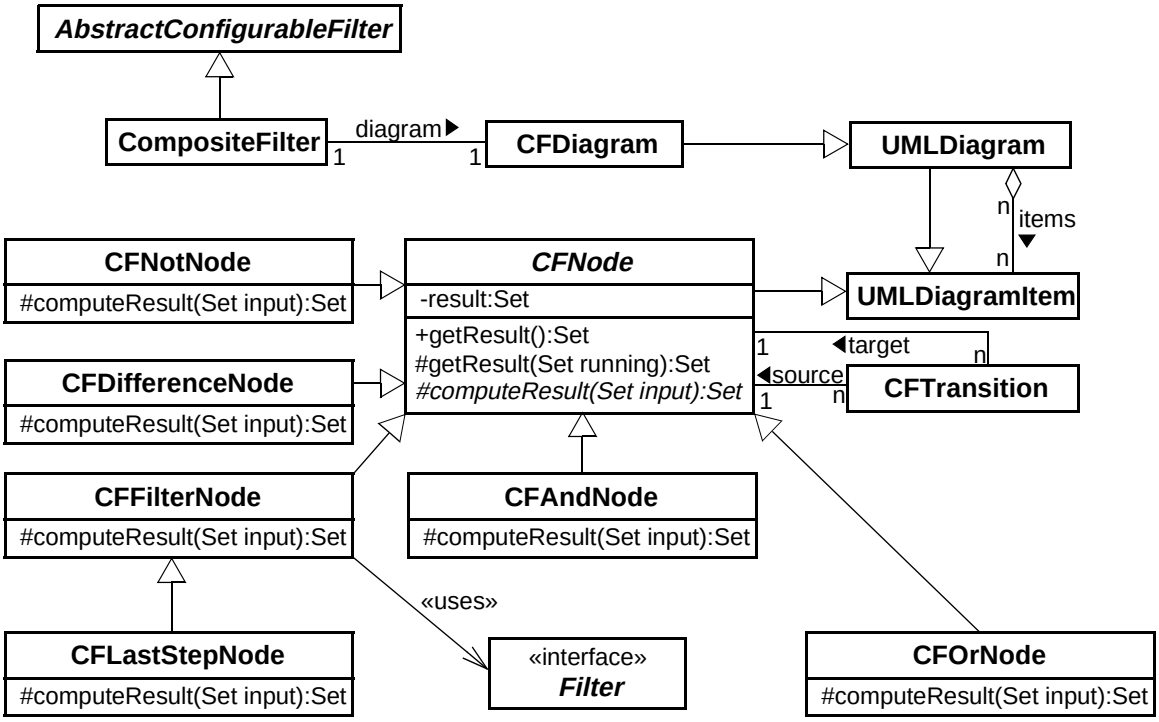

Abbildung 23: Der CompositeFilter und das CFDiagram

Ein *CFDiagram* besteht aus Instanzen der Klasse *CFNode*, die von *UMLDiagramItem* abgeleitet ist. Die Zuordnung zum *CFDiagram* erfolgt durch die Assoziation *items* in *UMLDiagram*. Die Klasse *CFNode* wird für jeden Knotentyp spezialisiert. Die Klasse *CFTransition* repräsentiert eine gerichtete Kante von einem Quell- zu einem Zielknoten in dem Diagramm. Das Ergebnis der Operation des Quellknotens wird entlang der *CFTransition* zum Zielknoten propagiert. Die Menge aller einlaufenden Transitionen in einen Knoten definiert die Parametermenge des Knotens. Die Klassen *CFAndNode*, *CFOrNode* und *CFDifferenceNode* repräsentieren die entsprechenden Mengenoperationen Schnitt, Vereinigung und Differenz. *CFNotNode* repräsentiert die Negation, die durch die Differenz von der Menge aller Elemente des Quelldiagramms definiert ist. *CFFilterNode* kapselt einen *Filter* und einen vordefinierten Kontext, der durch diesen *Filter* zu berechnen ist. Die Startmenge für die Regel wird durch die einlaufenden Transitionen bestimmt. *CFLastStepNode* entspricht *CFFilterNode*, jedoch werden nur die Diagrammelemente zurückgegeben, die im letzten Schritt der Kontextberechnung neu hinzugekommen sind. *CFStartNode* repräsentiert die Startmenge für die Regel, *CFEndNode* das Ergebnis. Ein *CFDiagram* muß genau eine Instanz der Klasse *CFStartNode* und eine Instanz der Klasse *CFEndNode* enthalten. Die grafische Representation der Diagrammelemente ist in Abbildung 24 dargestellt. Sie ist angelehnt an bestehende UML-Notationen und die Darstellung logischer Gatter in Blockschaltbildern.

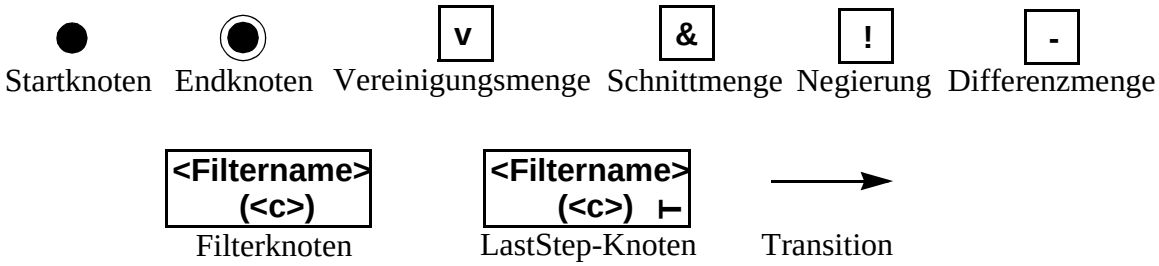

Abbildung 24: Die Elemente eines CFDiagram

Zur Berechnung des 1-Kontextes, der durch ein solches Diagramm definiert wird, müssen die Werte der einzelnen Knoten in geeigneter Reihenfolge berechnet werden. Um nur für solche Knoten Zwischenergebnisse zu berechnen, die für das Gesamtergebnis von Bedeutung sind, wird diese Berechnung rekursiv vom Endknoten aus definiert. Die Berechnungsvorschrift ist in Abbildung 25 als Aktivitätsdiagramm dargestellt.

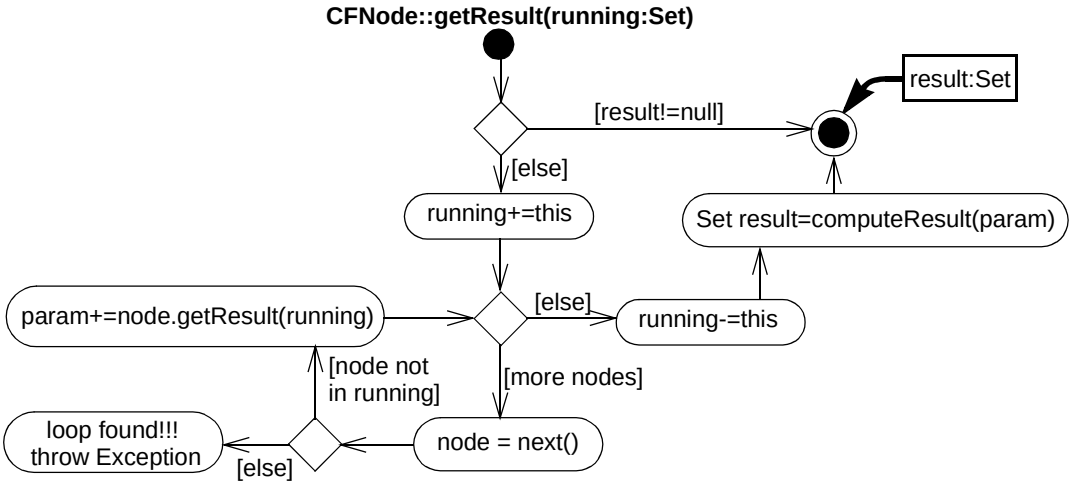

Abbildung 25: Aktivitätsdiagramm für CFNode::getResult

Falls für einen Knoten bereits ein Ergebnis vorliegt, so wird dies zurückgegeben. Sonst wird der aktuelle Knoten der Menge der Knoten hinzugefügt, die zur Zeit rechnen. Dann werden über alle einlaufenden Transitionen die Ergebnisse der verbundenen Knoten abgefragt. Wird dabei ein Knoten entdeckt, der in der Menge der rechnenden Knoten enthalten ist, so gibt es einen Zyklus im Graph und die Berechnung bricht ab. Sonst wird mit diesen Ergebnissen als Parameter die eigentliche Berechnung durchgeführt. Nach Abschluß der Berechnung wird der Knoten aus der Menge der rechnenden Knoten gelöscht und das Ergebnis in dem Feld result gespeichert. Diese Berechnungsvorschrift terminiert immer, da entweder der Graph Zyklen enthält, die wie oben beschrieben zum Abbruch führen, oder früher oder später Knoten ohne einlau-

fende Transitionen erreicht werden. Damit terminiert die Rekursion an dieser Stelle, da deren Ergebnis unmittelbar berechnet werden kann. Ein Beispiel:

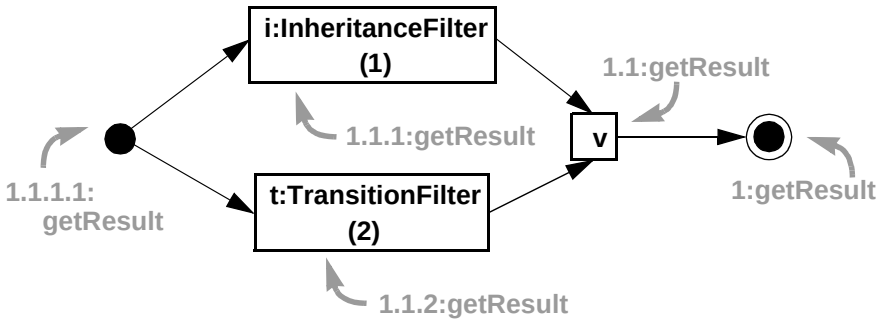

Abbildung 26: Beispieldiagramm mit Auswertungsreihenfolge

Das Beispiel zeigt eine Regel, deren 1-Kontext durch die Vereinigung des 2-Kontextes eines TransitionFilters mit Namen t und des 1-Kontextes eines InheritanceFilters mit Namen i definiert wird. Die Aufrufreihenfolge ist grau dargestellt.

3.7 Der FilterManager

In den vorangegangenen Abschnitten wurde die Definition neuer Regeln durch verschiedene Filter beschrieben. Zur Laufzeit werden Instanzen dieser Filterklassen in einer Factory-Klasse, dem FilterManager, zur Verfügung gestellt.

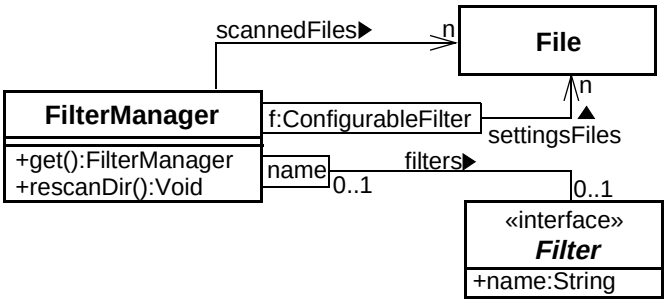

Abbildung 27: Die Klasse FilterManager

Der *FilterManager* ist als »Singleton« implementiert, so daß die einzige Instanz der Klasse von jeder Stelle des Systems aus erreichbar ist. Er verwaltet eine Menge von »Prototyp«-Instanzen [GHJV95] von Filterklassen. Auf Kopien dieser Instanzen kann mittels der Methode *getFilter(name:String)* über den Namen des Filters zugegriffen werden. Bei jedem Aufruf dieser Methode wird der interne Cache nach einer entsprechenden Prototyp-Instanz durchsucht. Falls keine solche Instanz gefunden wird, wird ein Cache-Update durchgeführt, bei dem nach neuen Filterklassen oder gespeicherten Filtereinstellungen gesucht wird. Danach wird der Cache erneut nach einer passenden Prototypinstanz durchsucht. Wird eine passende Instanz gefunden, so wird mittels Reflect-Methoden eine Kopie dieser Instanz erzeugt und zurückgegeben.

Das Cache-Update wird durch die Methode *rescanDir()* ausgelöst. Dabei wird ein in den Einstellungen von FUJABA definiertes Verzeichnis durchsucht. Dabei werden jar- und zip-Archive, sowie Unterverzeichnisse rekursiv durchsucht. Bei dieser Suche gefundene class-Dateien werden mittels eines modifizierten *ClassLoaders*, der das dynamische Nachladen von Klassen anhand ihrer class-Datei erlaubt, geladen, instantiiert und dem Prototyp-Cache hinzugefügt. Zusätzlich wird für jede gefundene Datei, die die abgespeicherten Einstellungen eines parametrisierten Filters enthalten, eine neue Instanz dieses Filters mit den Einstellungen aus der Datei erzeugt und dem Cache hinzugefügt. Eine Zuordnung der Datei zu der neuen Filterinstanz wird in der qualifizierten Assoziation *settingsFiles* gespeichert, um bei späteren Änderungen an den Parametern des Filters diese wieder sichern zu können. Um bereits untersuchte Dateien bei einem weiteren Update nicht erneut zu untersuchen, werden sie durch die Assoziation *scannedFiles* an den *FilterManager* gebunden.

Der *FilterManager* erlaubt also den Zugriff auf alle Filterklassen und gespeicherten Filtereinstellungen in dem Filterverzeichnis. Dadurch können insbesondere neue Filterklassen zur Laufzeit der Fujaba-Entwicklungsumgebung hinzugefügt werden. Nachdem zum Beispiel ein neuer Filter in FUJABA durch ein Storydiagramm modelliert worden ist, kann der Benutzer in FUJABA den zugehörigen Javacode exportieren, die Klasse kompilieren und sofort benutzen, um eine neue Sicht in seinem Projekt zu definieren.

3.8 Zusammenfassung

In diesem Kapitel ist die Formulierung von Regeln zur Definition der relevanten Diagrammanteile für Sichten durch Traversierung eines Graphen beschrieben worden. Dazu ist zunächst in Abschnitt 3.1 der Kontext als Möglichkeit zur rekursiven Definition von Traversierungsvorschriften auf Graphen vorgestellt worden. Zur Formulierung einer Kontext-Vorschrift reicht es dabei aus, eine Abbildungsvorschrift für den 1-Kontext anzugeben. In Abschnitt 3.2 ist dieses Konzept auf den Graphen aus den Instanzen des ASG und ihren Assoziationen umgesetzt worden. Dazu ist die Interfaceklasse *Filter* eingeführt worden, die die Implementierung der Traversierungsvorschrift kapselt. Die Konzentration auf die Definition eines 1-Kontextes erlaubt dabei die einfache Implementierung neuer Filter, zum Beispiel durch Modellierung der Filter-Regeln in FUJABA. In Abschnitt 3.3 sind parametrisierte Filter eingeführt worden, die eine Anpassung der Filterregel in Abhängigkeit von Attributen des Filters erlauben. Zur Veränderung dieser Attribute durch den Benutzer ist ein generischer Editor auf JavaBeans-Basis vorgestellt worden. Mit dem Ziel, die Definition neuer Filter noch flexibler zu gestalten, ist in Abschnitt 3.6 der *CompositeFilter* eingeführt worden, dessen 1-Kontext-Regel durch die Kombination verschiedener Filter durch Mengenoperationen definiert ist. Zur Formulierung dieser 1-Kontext-Regel wird dabei ein Flußdiagramm verwendet, dessen Syntax und Ausführungssemantik ebenfalls erläutert worden ist. Die verschiedenen Filter werden durch den in Abschnitt 3.7 vorgestellten FilterManager verwaltet, der das dynamische Nachladen von neu definierten Filterklassen zur Laufzeit ermöglicht.

KAPITEL 4

Definition von Sichten

In Abschnitt 4.1 »Definition und Synchronisation von Sichten« dieses Kapitels wird zunächst beschrieben, wie mit Hilfe der in Kapitel 3 vorgestellten Regeln eine Sicht definiert wird.

Werden Diagrammelemente in einem Diagramm gelöscht oder dem Diagramm hinzugefügt, so müssen die Sichten auf dieses Diagramm gemäß den sie definierenden Regeln aktualisiert werden. Die hierfür erforderlichen Kontrollstrukturen werden ebenfalls in Abschnitt 4.1 beschrieben.

In Abschnitt 4.2 »Änderungen auf Sichten« werden einige Besonderheiten bei Änderungen auf Sichten, wie dem Einfügen oder Löschen von Diagrammelementen, erläutert. Es wird darauf eingegangen, warum solche Änderungsoperationen, im Gegensatz etwa zu relationalen Datenbanken, für das hier vorgestellte Sichtenkonzept für UML-Diagramme unproblematisch sind. Zudem wird eine Erweiterung der Kontrollstrukturen aus Abschnitt 4.1 erläutert, die dem Benutzer eine transparentere Bearbeitung von Sichtendiagrammen ermöglicht.

4.1 Definition und Synchronisation von Sichten

Das in Kapitel 2 vorgestellte Sichtenkonzept erlaubt die Definition einer Sicht durch eine beliebige Menge voneinander unabhängiger Bedingungen zur Bestimmung der sichtrelevanten Anteile des Quelldiagramms. Diese Bedingungen werden durch Regeln formuliert, deren grundsätzlicher Aufbau in Kapitel 3 beschrieben worden ist. Sie sind umgesetzt in Unterklassen der Klasse *Filter*. Zur Formulierung dieser Filterregeln ist der *Kontext* eines Knotens oder einer Knotenmenge eines Graphen definiert worden, der eine Nachbarschaftsbeziehung der Knoten definiert. Um mittels der vorgestellten Regeln eine Sicht zu definieren, muß eine solche Definition neben dem Filter und dem Quelldiagramm auch die gewünschte Kontextgröße und Ausgangsmenge für die Kontextbestimmung auf dem Quelldiagramm festlegen.

Die in Abbildung 28 dargestellte Klasse *ViewDefinition* modelliert eine solche Definition einer Bedingung. Um sie mit dem Projekt abspeichern zu können, ist sie von der Klasse BasicIncrement abgeleitet. Die Zuordnung des Filters, der die Regel für eine solche Definition beschreibt, geschieht durch die Assoziation *filter*. Der durch den Filter zu berechnende Kontext wird durch das Attribut *context* festgelegt. Die Ausgangselemente für die Kontextberechnung sind durch die Assoziation *startItems* an die Klasse *ViewDefinition* gebunden. Durch die Methode *update* wird die Berechnung des Kontextes für diese Ausgangselemente angestoßen. Die so ermittelten Diagrammelemente werden der *ViewDefinition*-Instanz durch die Assoziation *items* zugeord-

net. Die Bindung an die Sicht erfolgt durch die Assoziation *diagram*. Die für eine Sicht relevanten Elemente des Quelldiagramms sind damit bestimmt durch die Vereinigung aller *UMLDiagramItem*-Objekte, die an die *ViewDefinition*-Instanzen der Sicht gebunden sind.

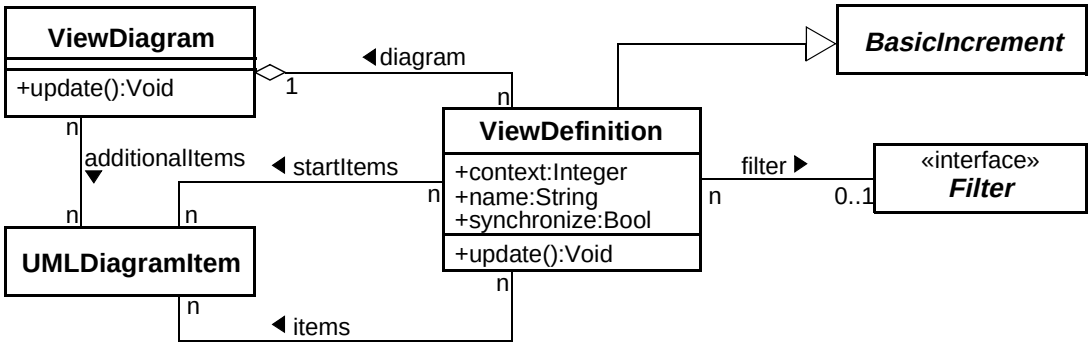

Abbildung 28: Die Klassen ViewDiagram und ViewDefinition

Durch die Möglichkeit, einer Sicht jederzeit weitere Bedingungen in Form neuer Instanzen der Klasse ViewDefinition hinzuzufügen, ist eine systematische Erweiterung bestehender Sichten möglich. So kann etwa in einem ersten Schritt eine Sicht auf das Klassendiagramm des Fahrstuhl-Beispiels aus Abbildung 29 definiert werden, die die Vererbungshierarchie der *Lifeform*-Klasse darstellt. In einem weiterem Schritt kann diese Sicht dann zum Beispiel um die Klassen erweitert werden, die mit den bereits in der Sicht enthaltenen Klassen durch eine Assoziation verbunden sind.

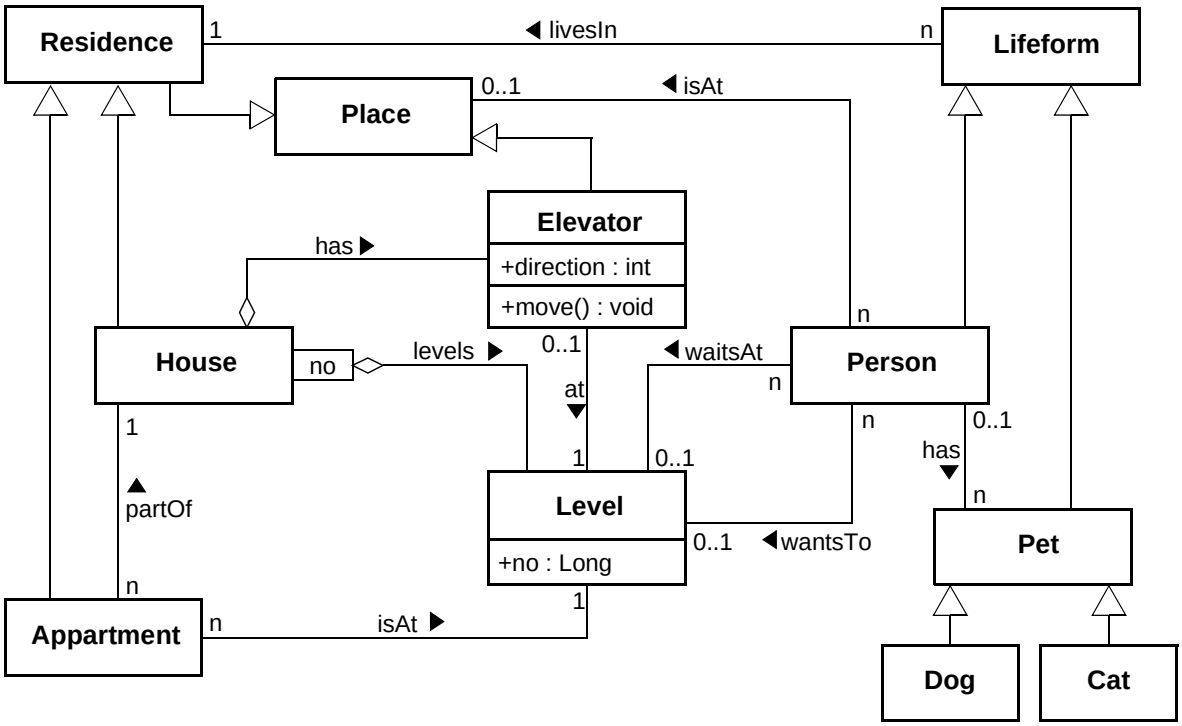

Abbildung 29: Klassendiagramm des Fahrstuhl-Beispiels

Um eine Sicht gemäß der Kontextregeln, durch die sie definiert ist, mit ihrem Quelldiagramm konsistent zu halten, muß der relevante Diagrammanteil bei Änderungen an dem Quelldiagramm neu berechnet werden. Dazu sind die Zugriffsmethoden für die Assoziation *items* in der Klasse *UMLDiagram* so modifiziert worden, daß beim Hinzufügen oder Löschen von Diagrammelementen durch einen Aufruf der Methode *update* für alle auf dem Diagramm definierten Sichten eine Neuberechnung des Inhalts ausgelöst wird. Dies entspricht dem »Observer«-Pattern aus [GHJV95]. In der Methode *update* werden zunächst alle zu synchronisierenden *ViewDefinition*-Instanzen durch Aufruf ihrer *update*-Methode aktualisiert. Durch das Attribut *synchronize* kann für jede Instanz der Klasse *ViewDefinition* individuell festgelegt werden, ob eine solche Synchronisation durchgeführt werden soll. Anschließend wird die Vereinigungsmenge über die Diagrammelemente aller *ViewDefinition*-Instanzen bestimmt und mit den aktuell an die Sicht gebundenen Elementen verglichen. Alle Elemente der Menge, die noch nicht Teil der Sicht sind, werden ihr hinzugefügt. In der Sicht enthaltene Elemente, die nicht Teil der Menge sind, werden aus der Sicht entfernt.

Bei Änderungen an den Attributen einer *ViewDefinition*-Instanz wird lediglich der durch diese Instanz definierte Kontext neu berechnet und mit dem alten Ergebnis verglichen. Neu hinzugekommene Elemente werden der Sicht hinzugefügt. Elemente des alten Ergebnisses, die in dem neu berechneten Kontext nicht mehr enthalten sind, werden darauf überprüft, ob andere *ViewDefinition*-Instanzen der gleichen Sicht diese Elemente beinhalten. Ist dies nicht der Fall, so werden sie aus der Sicht entfernt.

4.2 Änderungen auf Sichten

Das hier vorgestellte Sichtenkonzept erlaubt, im Gegensatz etwa zu relationalen Datenbanken, auch Diagrammoperationen auf Sichten, wie das Hinzufügen oder Löschen von Diagrammelementen.

In relationalen Datenbanken ist das Hinzufügen oder Löschen von Datentupeln in Sichten im Allgemeinen nicht erlaubt. Hierdurch kann der Datenbestand in einen inkonsistenten Zustand gebracht werden. Es können zum Beispiel bei der Definition einer Sicht einzelne Attribute der ursprünglichen Relation durch Projektion ausgeblendet werden. Soll in eine so definierte Sicht ein neues Datentupel eingefügt werden, so fehlen die Werte für diese Attribute, um das gegebene Tupel in die der Sicht zugrundeliegende Relation einzufügen. Außerdem kann eine Sicht aus Attributen mehrerer Relationen zusammengestellt sein. In diesem Fall ist die Abbildung von Einfüge- oder Löschoperationen von der Sicht auf die zugrundeliegenden Relationen nicht mehr eindeutig.

In dem hier vorgestellten Sichtenkonzept ist das Hinzufügen und Löschen von Diagrammelementen in Sichten hingegen problemlos möglich, da durch die eindeutige Festlegung eines einzelnen Quelldiagramms die Operationen auf der Sicht direkt auf das Quelldiagramm umgesetzt werden können. Das Löschen von Diagrammelementen erfolgt einfach durch Löschen dieser Elemente aus dem Quelldiagramm.

Entsprechend werden in einer Sicht neu erzeugte Diagrammelemente dem Quelldiagramm hinzugefügt. Solche Elemente zählen allerdings in vielen Fällen nach den Filterregeln nicht zu den sichtenrelevanten Diagrammelementen. Um dem Benutzer eine transparente Arbeit auf Sichten zu ermöglichen, sollten diese in einer Sicht erstellten Diagrammelemente trotzdem in der Sicht angezeigt werden. Dazu werden diese Elemente den Instanzen der Klasse *ViewDiagram* durch die Assoziation *additionalItems* zugeordnet. Die Bindung eines Elementes an eine *ViewDiagram*-Instanz durch diese Assoziation geschieht immer dann, wenn das Element zum Zeitpunkt der Bindung an die Sicht durch die *items*-Assoziation von *UMLDiagram* nicht Teil des Quelldiagramms ist. Solche Elemente sind augenscheinlich nicht aufgrund einer der die Sicht definierenden Filterregeln an die Sicht gebunden worden, sondern in der Sicht neu erzeugt worden. Durch die Assoziation *additionalItems* an ein *ViewDiagram*-Objekt gebundene Diagrammelemente sind unabhängig von den Filterregeln der Sicht immer in der Sicht enthalten. Sie können nur manuell vom Benutzer, etwa nach Beendigung der Arbeit auf der Sicht, aus dieser Liste entfernt und der normalen Auswertung durch die Filterregeln unterzogen werden.

4.3 Zusammenfassung

Wie in Kapitel 2 dargestellt worden ist, wird der Inhalt einer Sicht durch eine Menge von Regeln auf dem Quelldiagramm bestimmt. Zur Auswertung dieser in Kapitel 3 eingeführten Regeln sind zusätzliche Parameter anzugeben. Dies sind der zu berechnende Kontext und die Startelemente für die Kontextberechnung. Die in Abschnitt 4.1 vorgestellte Klasse *ViewDefinition* stellt diese Parameter zusammen mit einer Instanz einer Filterklasse zur Verfügung. Der Inhalt einer Sicht kann damit durch mehrere Instanzen dieser Klasse definiert werden, die dazu an das Sichtendiagramm gebunden werden. Ferner ist in Abschnitt 4.1 ein Synchronisationsmechanismus beschrieben worden, der den Inhalt einer Sicht bei Änderungen am Quelldiagramm aktualisiert. In Abschnitt 4.2 ist abschließend beschrieben worden, wie Änderungen auf Sichten auf ihr Quelldiagramm zurückgeführet werden.

KAPITEL 5

Die Benutzerschnittstelle

Um dem Benutzer die Definition und Bearbeitung von Sichten in Fujaba mittels der in den vorangegangenen Kapiteln vorgestellten logischen Strukturen und Klassen zu ermöglichen, ist die grafische Benutzerschnittstelle von Fujaba um Dialoge zur Definition und Manipulation von Filtern und Sichten zu erweitern. Diese Dialoge werden in Abschnitt 5.1 »Die Benutzerschnittstelle« vorgestellt. Anschließend wird in Abschnitt 5.2 »Operationen auf Sichten« die Arbeit mit Sichten aus Benutzersicht anhand einer Beispielsitzung vorgestellt.

5.1 Die Benutzerschnittstelle

Im Rahmen dieser Arbeit ist die Ergänzung der grafischen Benutzerschnittstelle von FUJABA um Dialoge zum Erzeugen und Bearbeiten von Sichten und zur Verwaltung der vorhandenen Filter erforderlich. Zur Implementierung dieser Dialoge wird die Swing-Bibliothek [ELW98] verwendet, die auch für die grafische Benutzerschnittstelle von Fujaba benutzt wird. Die Swing-Bibliothek ermöglicht die Entwicklung von grafischen Benutzeroberflächen in Java. Dazu werden zahlreiche Standardkomponenten, wie Containerflächen, Schaltflächen, Menüs und Fenster zur Verfügung gestellt, aus denen komplexe Benutzerschnittstellen zusammengestellt werden können. Swing ist Bestandteil der Java Foundation Classes (JFC), die im Java Development Kit seit Version 1.2 enthalten sind, und ist seitdem als Standard-API für die Entwicklung grafischer Benutzeroberflächen in Java etabliert.

5.1.1 Der FilterEditor-Dialog

Um die Implementierung neuer Dialoge zu vereinfachen und allen Dialogen in FUJABA einen einheitlichen Aufbau zu geben, stellt die Fujaba-API die Klasse *FujabaDialog* zur Verfügung. Diese Klasse erweitert die Swing-Komponente *JDialog* und stellt ein Grundgerüst für die Implementierung neuer Dialoge zur Verfügung. Hierzu bietet sie die Methoden *guiPanelOkCancelHelp* und *guiPanelCloseHelp* zur Erzeugung von Standardschaltflächen für den Dialog. Durch die Methode *guiBuild* wird der Dialog aufgebaut. Durch Überschreiben dieser Methode können Unterklassen Komponenten zu dem Dialog hinzufügen. Durch die Methode *unparse* werden die Komponenten eines Dialoges mit den zu edierenden Werten initialisiert. Dazu ist diese Methode in allen Unterklassen von *FujabaDialog* zu redefinieren. Analog werden die Werte der Dialogkomponenten durch die Methode *parse* ausgelesen, die zu diesem Zweck ebenfalls in Unterklassen zu redefinieren ist.

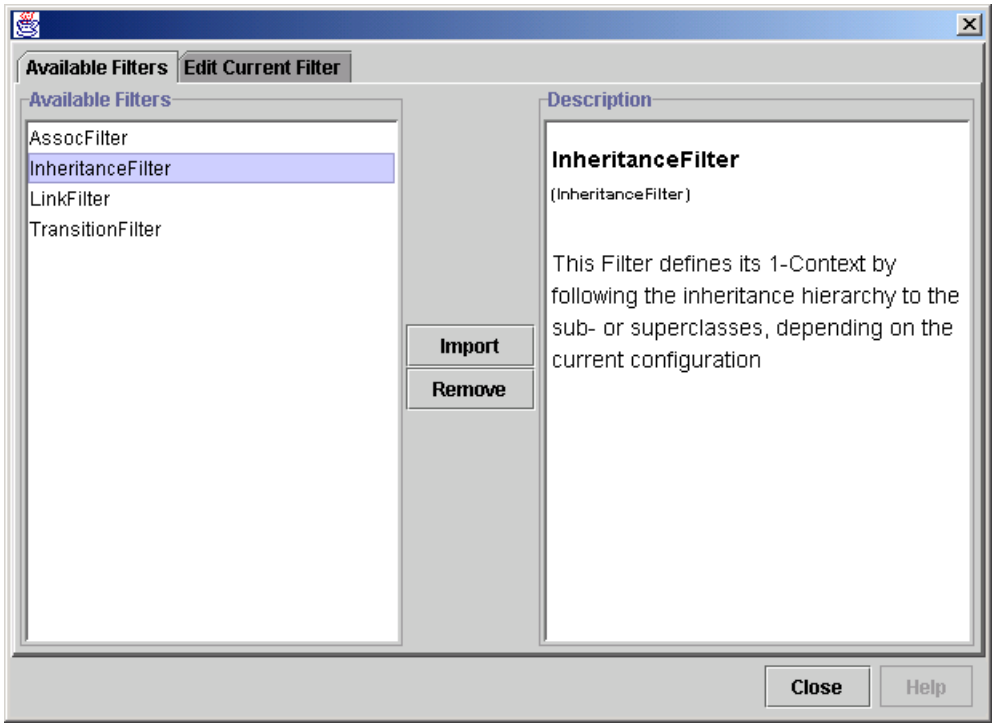

Abbildung 30: Dialog zur Filterverwaltung

Auf der Basis dieser Klasse ist im Rahmen dieser Arbeit der in Abbildung 30 dargestellte Dialog für die Verwaltung der Filter des in Kapitel 3 vorgestellten FilterManagers implementiert worden. Der Inhalt des Dialogs besteht aus zwei Registerkarten zur Auswahl und Bearbeitung der zur durch den FilterManager zur Verfügung gestellten Filter. Die in Abbildung 30 dargestellte Registerkarte besteht aus einer Listenkomponente auf der linken Seite, einem Textfeld auf der rechten Seite und einer Gruppe von Schaltflächen in der Mitte. In der Listenkomponente können die zur Verfügung stehenden Filter ausgewählt werden. In dem Textfeld zur Rechten wird für den aktuell selektierten Filter eine Kurzbeschreibung angezeigt. In der ersten Zeile ist der Name des Filters angegeben, der durch das Attribut *name* in der Klasse *Filter* definiert ist. Die zweite Zeile gibt den Namen der Filterklasse an und der folgende Text ist der Wert des Attributes *description* des Filters. Die Schaltflächen in der Mitte des Dialoges erlauben das Löschen und Importieren von Filtern.

Ist der selektierte Filter eine Instanz der Klasse *ConfigurableFilter*, so kann diese in der zweiten Registerkarte konfiguriert werden. Dazu enthält diese Registerkarte im oberen Teil den von dem Filter zur Verfügung gestellten Editor (vergleiche Kapitel 3, Abschnitt 3.3). Darunter befinden sich Schaltflächen zum Übernehmen und Speichern der vorgenommenen Einstellungen, zum Erzeugen einer Kopie des Filters mit den Einstellungen des Editors und zum Zurücksetzen der Editorfelder auf die Werte des Filters. Diese Funktionen werden von der in Kapitel 3 vorgestellten Klasse *FilterEditor* zur Verfügung gestellt. In Abbildung 31 ist diese Registerkarte mit dem in Kapitel 3 vorgestellten generischen Editor für eine Instanz der Klasse *InheritanceFilter* dargestellt.

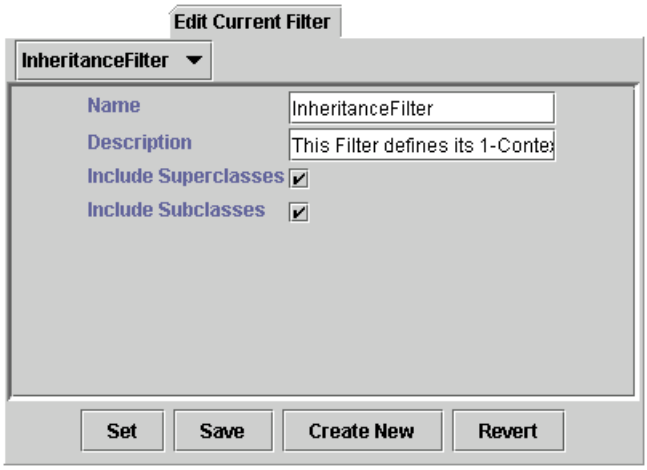

Abbildung 31: Registerkarte »Edit Current Filter«

Die beiden Registerkarten sind von der Klasse *FujabaDialogPanel* abgeleitet. Diese erweitert die Zeichenfläche *JPanel* der Swing-Bibliothek um die Methoden unparse und refreshDialog, die ähnlich den Methoden *parse* und *unparse* der Klasse *FujabaDialog* die Initialisierung und das Auslesen der auf dem Panel enthaltenen Komponenten ermöglichen. Die Klassen des Dialoges sind in Abbildung 32 vereinfacht dargestellt.

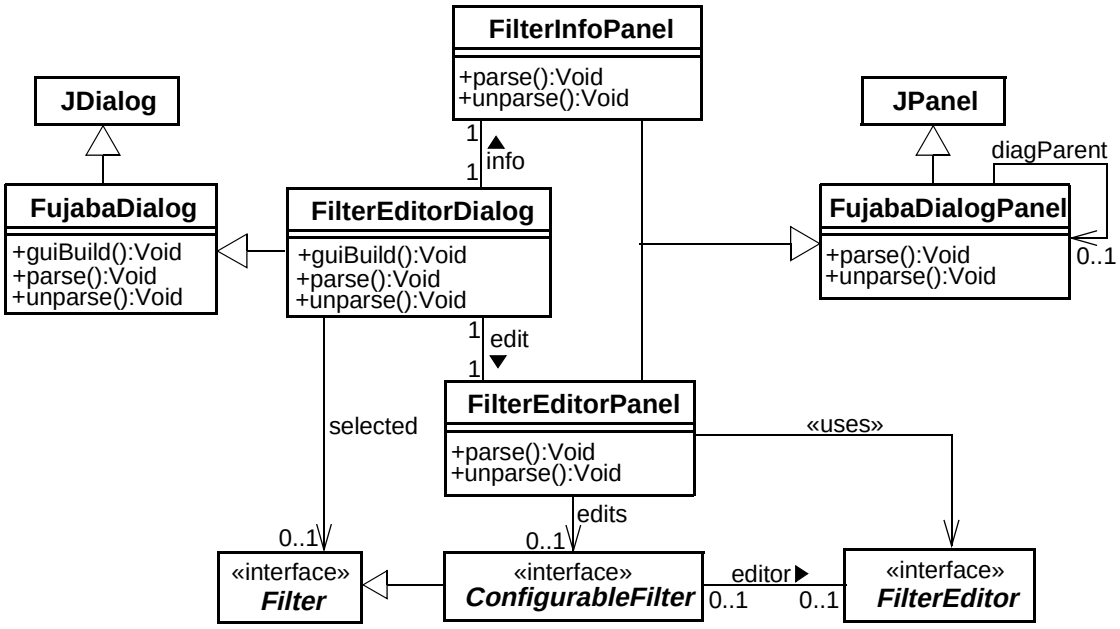

Abbildung 32: Klassendiagramm »FilterEditorDialog«

5.1.2 Der Sichtendialog

Neben der Verwaltung der im System bekannten Filter muß auch für das Anlegen und Verändern von Sichten ein Dialog zur Verfügung gestellt werden. Dieser Dialog basiert im Gegensatz zu dem in Unterabschnitt 5.1.1 vorgestellten *FilterEditorDialog* nicht auf der Klasse *Fujaba-*

Dialog. Denn für Dialoge, die der Bearbeitung von Instanzen des Fujaba-ASG dienen, wird eine andere Basisklasse in der Fujaba-API zur Verfügung gestellt. Die Klasse *PropertyEditor* stellt genau wie die Klasse *FujabaDialog* das Grundgerüst eines Dialoges zur Verfügung, um eine einfache Implementierung neuer Dialoge zu ermöglichen und ein einheitliches Aussehen dieser Dialoge sicherzustellen. Die Klasse *PropertyEditor* stellt daher bereits die Schaltflächen »Ok« und »Cancel«, sowie eine Textzeile für Statusinformationen und kurze Hilfetexte am unteren Rand zur Verfügung. In Unterklassen von *PropertyEditor* können dem Dialog weitere Schaltflächen durch Überschreiben der Methode *additionalButtons* hinzugefügt werden. Der Hauptbereich des Dialogs wird in der Methode *additionalProperties* aufgebaut. Die in diesen Bereich eingefügten Komponenten werden, ähnlich wie dies in der Klasse *FujabaDialog* geschieht, durch Aufruf der Methode *unparse* initialisiert. Die hierfür benötigten Daten werden der zu edierenden *UMLIncrement*-Instanz entnommen, die dem Dialog im Konstruktor oder durch die Methode *setIncrement* zugeordnet wird. Durch Aufruf der Methode *parse* werden die geänderten Werte wieder aus den Dialogkomponenten ausgelesen und in die *UMLIncrement*-Instanz zurückgeschrieben.

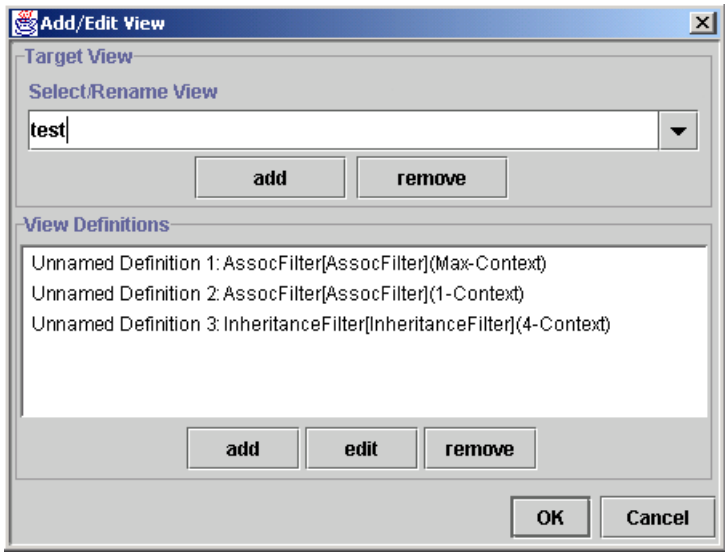

Abbildung 33: Dialog zur Bearbeitung von Sichten

Der Dialog für die Erzeugung und Änderung von Sichten ist in Abbildung 33 dargestellt. Im oberen Teil steht ein Auswahlelement, eine sogenannte Combobox, zur Auswahl der zu bearbeitenden Sicht zur Verfügung. In ihr sind alle auf dem aktuellen Diagramm definierten Sichten enthalten. Ist das aktuelle Diagramm ebenfalls eine Sicht, so ist sie hier ebenfalls enthalten. Die Combobox ist edierbar. Das heißt, daß der aktuell angezeigte Text verändert werden kann. Dadurch können bestehende Sichten umbenannt werden. Mit den unter der ComboBox befindlichen Schaltflächen kann eine neue Sicht angelegt beziehungsweise die in der ComboBox selektierte Sicht gelöscht werden. In dem Bereich unterhalb dieser Schaltflächen befindet sich ein Listenelement, das die Instanzen der Klasse *ViewDefinition* (siehe Kapitel 4) enthält, die den Inhalt der in der Combobox selektierten Sicht festlegen. Mit Hilfe der darunter befindlichen Schaltflächen kann eine in der Liste selektierte Definition aus der Liste gelöscht, bearbeitet oder eine neue Definition hinzugefügt werden.

Wird eine der Schaltflächen »Add« oder »Edit« betätigt, so wird der in Abbildung 34 dargestellte Dialog zur Festlegung der Eigenschaften der Definition geöffnet. Da die Klasse *ViewDefinition* keine Unterklasse von *UMLIncrement* ist, basiert dieser Dialog wiederum auf der Klasse *FujabaDialog*. Am oberen Rand des Dialoges kann der Name der Definition angegeben werden. Im daran anschließenden Bereich kann der zu verwendende Filter ausgewählt und angepaßt werden. Daneben wird der zu berechnende Kontext und der Wert des *synchronize*-Attributes des *ViewDefinition*-Objektes festgelegt. Im unteren Bereich des Dialoges befinden sich zwei Listenelemente zur Auswahl der Startelemente für die Kontextberechnung. Die linke Liste enthält alle noch nicht zugeordneten Diagrammelemente des Quelldiagramms. Mit den daneben befindlichen Schaltflächen können Elemente in die rechte Liste verschoben oder aus ihr entfernt werden. Diese Liste enthält die Startelemente für die Kontextberechnung.

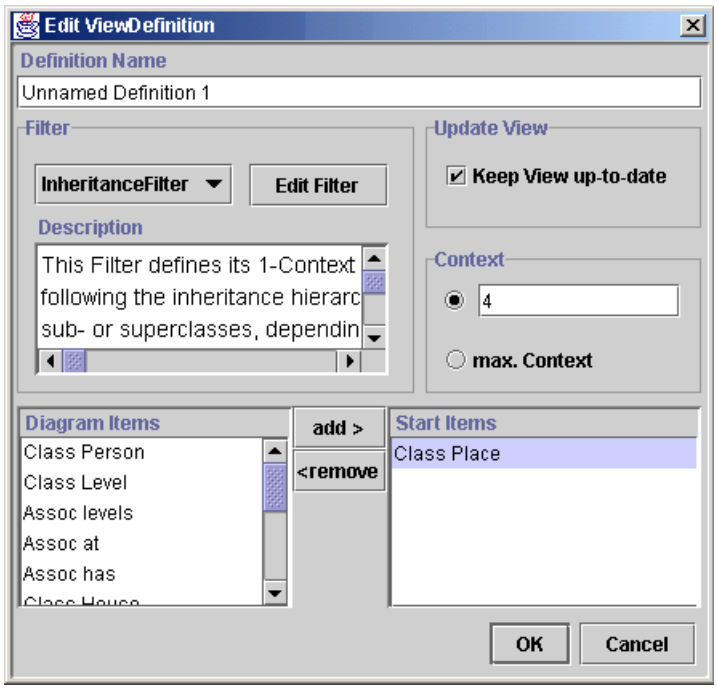

Abbildung 34: Dialog »Edit ViewDefinition«

5.2 Operationen auf Sichten

Nachdem in den vorangegangenen Kapiteln alle für die Umsetzung des in Kapitel 2 vorgestellten Sichtenkonzeptes erforderlichen Klassen vorgestellt worden sind und in Abschnitt 5.1 die Dialoge für die Arbeit mit Sichten bereits erläutert worden ist, soll in diesem Abschnitt die Arbeit mit diesem Sichtenkonzept aus dem Blickwinkel eines Benutzers vorgestellt werden. Dies geschieht in einer Beispielsitzung anhand des Fahrstuhlbeispiels aus Kapitel 1. Abbildung 35 zeigt das Klassendiagramm des Fahrstuhlbeispiels in FUJABA

.

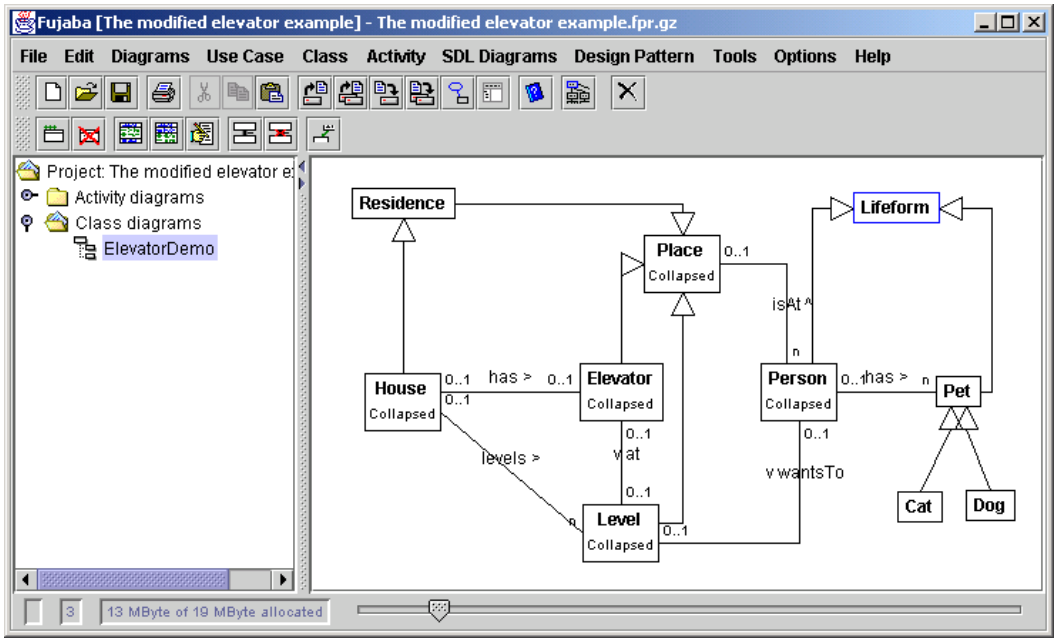

Abbildung 35: Klassendiagramm in FUJABA

Bevor hierauf eine Sicht definiert wird, soll zunächst einer der vordefinierten Filter angepaßt werden. Der Dialog hierzu, der in Abschnitt 5.1 bereits vorgestellt wurde, kann über den Menüpunkt »View Filters« im Menü »Tools« aufgerufen werden. Wählt man in diesem Menü den Filter *InheritanceFilter* aus, so wird die Registerkarte »Edit Filter« verfügbar. In dem auf dieser Registerkarte dargestellten Editor kann nun zum Beispiel die Option »Include Superclasses« deselektiert werden (vergleiche Abbildung 36)

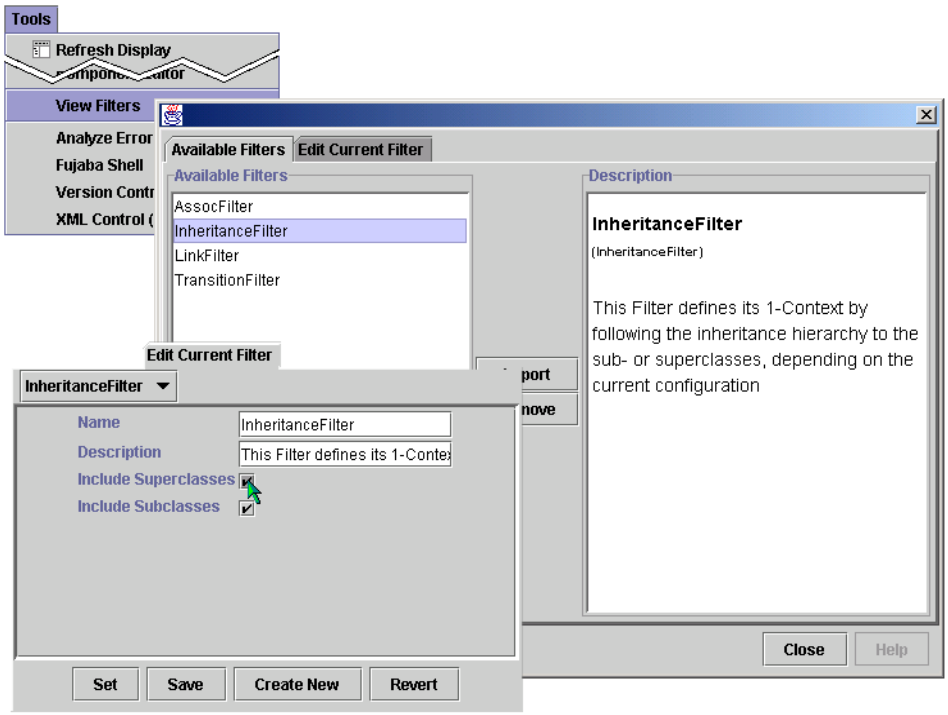

Abbildung 36: Verwendung des »View Filters«-Dialoges

Anschließend kann mit Hilfe dieses Filters eine neue Sicht definiert werden. Diese Sicht soll die Klassenhierarchie der Klasse *Lifeform* darstellen. Daher wird zunächst die Klasse Lifeform im Klassendiagramm selektiert. Durch Druck der rechten Maustaste auf dem Diagramm kann dann das Kontextmenü für Klassendiagramme aufgerufen werden. Durch Wahl des Menüpunktes »Add/Edit Views« wird der aus Abschnitt 5.1 bekannte Dialog zur Bearbeitung von Sichten geöffnet. Durch Drücken der Schaltfläche »New« unter der ComboBox wird eine neue Sicht erzeugt und in der ComboBox als „Unnamed View“ angezeigt. Durch einen Mausklick in die ComboBox kann dieser Name geändert werden, etwa zu „Lifeforms“. Bislang sind für diese neue Sicht noch keine Regeln zur Auswahl der relevanten Diagrammanteile definiert. Dies geschieht durch Anwählen der Schaltfläche »New«. Hierdurch wird das Dialogfenster »Edit ViewDefinitions« geöffnet, in dem nun der anfangs bearbeitete InheritanceFilter ausgewählt werden kann. Für den Kontext ist die Option »max. Context« zu wählen, um die gesamte Klassenhierarchie von *Lifeform* in die Sicht aufzunehmen. Die Schaltfläche zur Festlegung des Update-Verhaltens ist bereits standardmäßig selektiert. Hierdurch wird festgelegt, daß bei Änderungen an dem Quelldiagramm diese Regel neu ausgewertet wird. Im unteren Teil des Dialogs wird die Startmenge für die Kontextberechnung ausgewählt. Die Klasse Lifeform, die im Vorfeld im Klassendiagramm selektiert wurde, ist bereits markiert. Durch Drücken der »Add«-Schaltfläche wird sie der Liste der Startelemente hinzugefügt. Bevor der Dialog nun über die Schaltfläche »Ok« verlassen wird, erhält die *ViewDefinition*-Instanz noch den Namen „LifeformHierarchie“. Siehe hierzu auch Abbildung 37

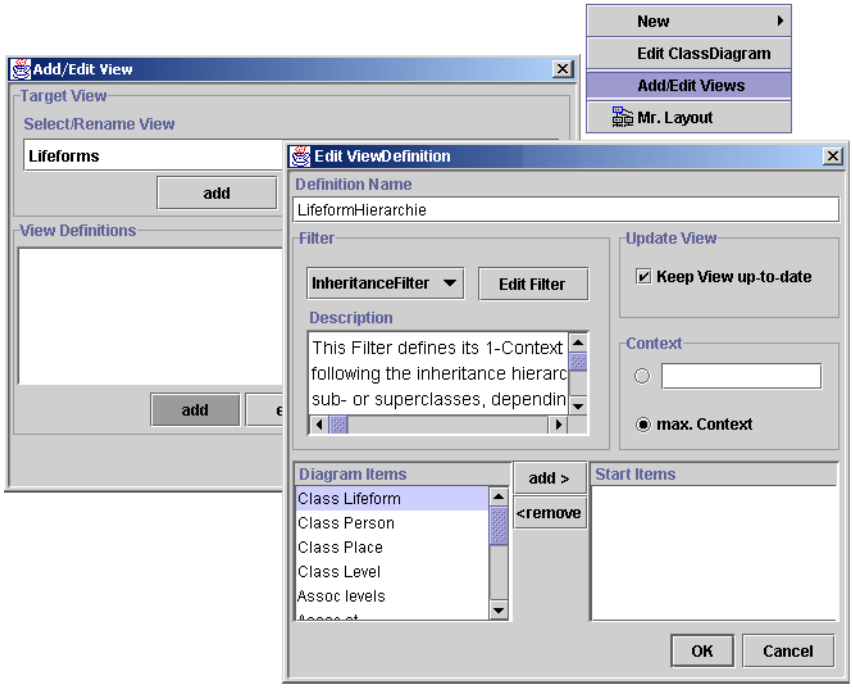

Abbildung 37: Anlegen einer Sicht

Vor Verlassen des Sichtendialogs wird anschließend noch eine weitere *ViewDefinition*-Instanz erzeugt. Diesmal wird der AssocFilter gewählt und der 1-Kontext auf den Klassen Lifeform und Person bestimmt. Diese Instanz erhält den Namen „LifeformAssocs“. Nach Verlassen des Sichten-Dialogs kann die Sicht über den Projektbaum im linken Teil der FUJABA-Oberfläche ange-

wählt werden. Der Eintrag für die Sicht ist dem Eintrag für das ursprüngliche Diagramm untergeordnet und durch ein Augen-Piktogramm gekennzeichnet. Die Sicht ist in Abbildung 38 dargestellt. Dort ist auch zu sehen, daß für das Sichtendiagramm dieselben Werkzeugleisten zur Verfügung stehen, wie für das Quelldiagramm, so daß die Sicht genauso bearbeitet werden kann, wie das Quelldiagramm.

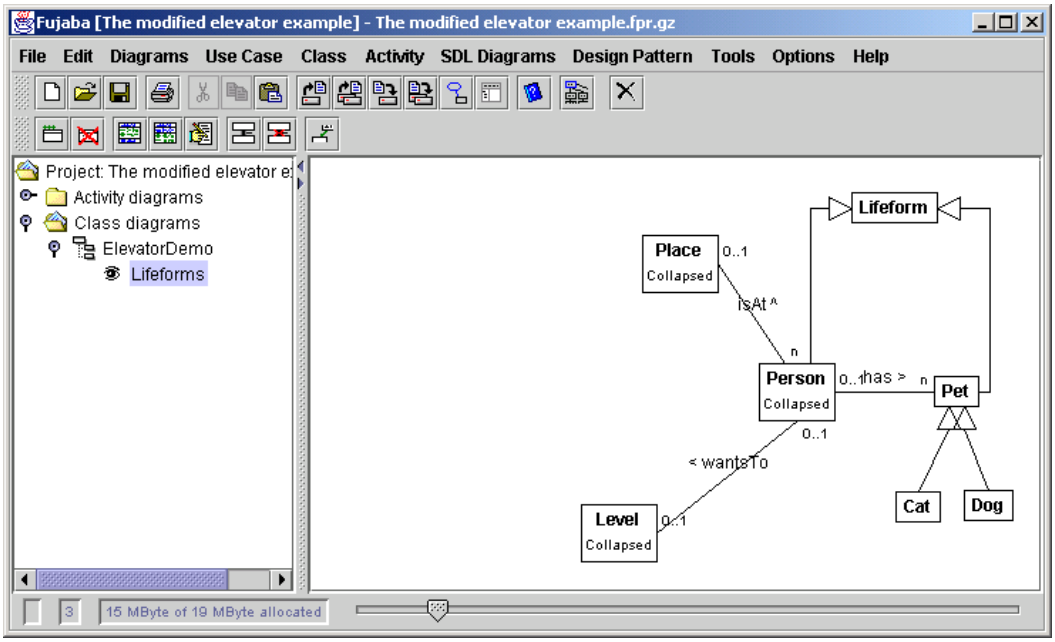

Abbildung 38: Die Sicht „Lifeforms“ in FUJABA

Als nächstes wird nun die Assoziation *livesIn* in das Quelldiagramm eingefügt, wie in Abbildung 39 dargestellt, und die Assoziation *isAt* gelöscht. Dann ist anschließend in der Sicht die Klasse *Residence* hinzugekommen, da sie nach dem Einfügen von *livesIn* nun im 1-Kontext der Klasse *Lifeform* liegt, der durch die ViewDefinition „LifeformAssocs“ bestimmt ist. Die Klasse *Place* ist hingegen nicht mehr Teil der Sicht, da sie durch Löschen der Assoziation *isAt* nicht mehr in dem 1-Kontext von *Person* oder *Lifeform* liegt.

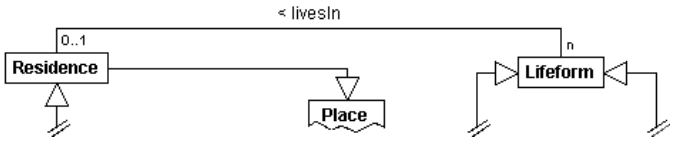

Abbildung 39: Einfügen der Assoziation livesIn

Nun soll in der Sicht die Klasse *Appartment* angelegt werden, die von *Residence* erben soll. Abbildung 40 zeigt die Sicht nachdem dies geschehen ist. Hier ist zu beachten, daß die Klasse *Appartment* weder durch „LifeformAssocs“ noch durch „LifeformHierarchie“ als sichtenrelevant ausgezeichnet wird. Die Klasse und der Generalisierungspfeil zu *Residence* werden nur angezeigt, weil sie in dieser Sicht erzeugt worden sind und deshalb in der *additionalItems*-Menge des Sichtendiagramms enthalten sind. Die Elemente dieser Menge werden unabhängig von den Filterregeln immer angezeigt, um die Arbeit auf einer Sicht zu erleichtern (vergleiche

Abschnitt 4.2). Ohne diese Maßnahme wäre die Klasse *Appartment* zwar ebenfalls erzeugt und dem Quelldiagramm „ElevatorDemo“ hinzugefügt worden. In der Sicht wäre sie trotzdem nicht erschienen, da sie nicht Teil einer der beiden durch „LifeformAssocs“ und „LifeformHierarchie“ bestimmten Kontextmengen ist.

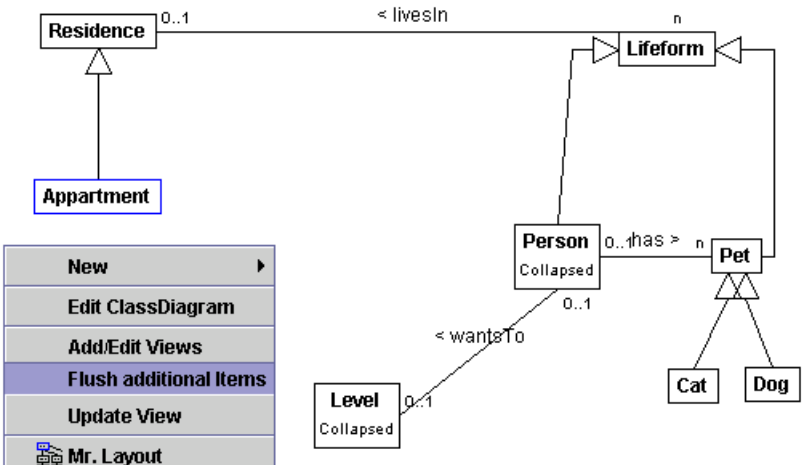

Abbildung 40: Die bearbeitete Sicht

Das Kontextmenü einer Sicht enthält zwei zusätzliche Menüpunkte. Über den Menüpunkt »Flush additional Items« wird die additionalItems-Menge der Sicht geleert und die Sicht neu berechnet. In diesem Beispiel hätte das das Entfernen der Klasse Appartment aus der Sicht zur Folge. Der Menüpunkt »Update View« löst eine Neuberechnung des Sichteninhalts aus. Hierbei werden im Gegensatz zu der automatischen Neuberechnung, die bei Änderungen am Quelldiagramm ausgelöst wird, auch solche ViewDefinition-Instanzen neu ausgewertet, deren synchronize-Flag nicht gesetzt ist. So ist es etwa möglich, eine Sicht als eine Art „Snapshot“ zu definieren, indem für alle ViewDefininition-Instanzen in dem Dialog aus Abbildung 37 die Schaltfläche »keep view up-to-date« deselektiert wird. Durch Anwahl des Menüpunktes »Flush additional Items« kann die Sicht dann zu jedem beliebigen Zeitpunkt manuell aktualisiert werden.

5.3 Zusammenfassung

In diesem Kapitel sind die Elemente der grafischen Benutzeroberfläche von Fujaba vorgestellt worden, die für die Arbeit mit Sichten benötigt werden. In Abschnitt 5.1 sind zunächst die Dialoge für die Vewaltung der Filter und das Anlegen und Edieren von Sichten vorgestellt worden. Anschließend sind diese Dialoge und einige weitere Elemente der grafischen Benutzeroberfläche im Rahmen einer Beispielssitzung genauer erläutert worden. In dieser Beispielsitzung ist ausgehend von dem Klassendiagramm des Fahrstuhlbeispiels aus Kapitel 1 eine Sicht mit zwei Instanzen der Klasse ViewDefinition zur Festlegung des Sichteninhalts erzeugt worden. Anschließend sind auf dem Quelldiagramm und der Sicht einige Diagrammelemente gelöscht und neu erzeugt worden, um die Synchronisation von Quelldiagramm und Sicht darzustellen.

Zur vollständigen Integration des Sichtenkonzeptes in die grafische Oberfläche von FUJABA sind neben den hier vorgestellten neuen Dialogen und Menüeinträgen noch einige kleinere Anpassungen an der Fujaba-Benutzeroberfläche vorgenommen worden, auf die hier nicht im Einzelnen eingegangen worden ist. So mußte zum Beispiel der Code zur Bereitstellung der Kontextmenüs so angepaßt werden, daß für Sichten das Kontextmenü des Quelldiagramms angezeigt wird. Ähnliches gilt für die Werkzeugleisten zur Diagrammbearbeitung.

KAPITEL 6

Zusammenfassung und Ausblick

Ziel dieser Arbeit ist die Konzeption und Implementierung eines Sichtenkonzeptes für die Entwicklungsumgebung FUJABA gewesen. Ein solches Sichtenkonzept soll die systematische Gliederung großer und komplexer UML-Diagramme in mehrere Teildiagramme ermöglichen, um so eine bessere Verständlichkeit und Übersichtlichkeit großer Spezifikationen zu erreichen.

In Kapitel 2 ist zunächst das grundlegende Sichtenkonzept definiert worden. Es ist festgestellt worden, daß die direkte Festlegung der in eine Sicht aufzunehmenden Elemente des Quelldiagramms nicht sinnvoll ist, da bei einem solchen Ansatz nach mehreren Änderungen an dem Quelldiagramm eine Sicht keinen sinnvollen Ausschnitt des Quelldiagramms mehr darstellt. Stattdessen ist ein regelbasierter Ansatz zur Bestimmung der sichtenrelevanten Teile des Quelldiagramms gewählt worden. Die hierfür erforderlichen Regeln, die in Kapitel 3 vorgestellt worden sind, formulieren Bedingungen auf der Struktur des zugrundeliegenden Diagrammes. Das hier vorgestellte Sichtenkonzept wird daher auch als *strukturelles Sichtenkonzept* bezeichnet, da die Elemente einer Sicht aufgrund ihres strukturellen Zusammenhanges ausgewählt werden.

Die Definition der Regeln für dieses Sichtenkonzept erfolgt durch die Festlegung einer Abbildungsvorschrift für den sogenannten Kontext einer ausgezeichneten Menge von Startelementen des Quelldiagramms. Der Kontext beschreibt in diesem Zusammenhang eine Nachbarschaftsbeziehung zwischen den Instanzen des UML-Metamodells beziehungsweise des abstrakten Syntaxgraphen von FUJABA. Der Kontext ist rekursiv definiert, was eine einfache und kompakte Implementierung der Regeln zur Sichtendefinition in Unterklassen der Klasse *Filter* erlaubt. Angefangen mit einfachen Filtern, die statisch eine bestimmte Regel kapseln, sind mehrere Filterklassen unterschiedlicher Komplexität vorgestellt worden. Die komplexeste Filterklasse ist dabei der *CompositeFilter* gewesen, der die Verknüpfung mehrerer *Filter* durch Mengenoperationen erlaubt.

Um mit den in Kapitel 3 vorgestellten Regeln den Inhalt einer Sicht beschreiben zu können, müssen neben dem Filter auch die zur Auswertung der Filterregel erforderlichen Parameter zur Verfügung gestellt werden. In Kapitel 4 ist zu diesem Zweck die Klasse *ViewDefinition* vorgestellt worden. An dieser Stelle ist auch auf die Synchronisation zwischen Quelldiagramm und Sicht auf der Basis dieser Klasse eingegangen worden.

Abschließend sind in Kapitel 5 die Komponenten der grafischen Benutzeroberfläche von Fujaba, die für die Arbeit mit Sichten in FUJABA benötigt werden, im Rahmen einer Beispielsitzung vorgestellt worden.

Die in dieser Arbeit vorgestellten Konzepte und Strukturen sind vollständig in der Fujaba-Entwicklungsumgebung umgesetzt worden, so daß die Arbeit mit Sichten in Fujaba wie in der Beispielsitzung in Kapitel 5 beschrieben bereits möglich ist. In der Praxis hat sich bereits gezeigt, daß durch die Verwendung des hier vorgestellten Sichtenkonzeptes die Übersichtlichkeit umfangreicher Diagramme in FUJABA deutlich verbessert werden kann. Es hat sich jedoch auch gezeigt, daß noch zahlreiche Ergänzungen und Erweiterungen zu dem vorgestellten Konzept zu erwägen sind. So ist zum Beispiel eine Ergänzung um sogenannte negative Regeln denkbar, also um Regeln, durch die bestimmte Diagrammteile aus einer Sicht ausgeschlossen werden können. Eine weitere mögliche Ergänzung zu dem hier vorgestellten Sichtenkonzept besteht in der Bereitstellung von Funktionen zur automatischen Generierung von häufig benötigten Sichten. So könnten die Vererbungshierarchien eines Klassendiagramms etwa getrennt in verschiedenen automatisch generierten Sichten dargestellt werden. Zur flexiblen Gestaltung einer solchen automatischen Sichtengenerierung könnten die zu generierenden Sichten auch anhand des Kontextes der Diagrammelemente bestimmt werden. So kann etwa für jede Klasse eines Klassendiagramms der 1-Kontext aufgrund der Vererbungsbeziehungen definiert und dann für jede Klasse, in deren 1-Kontext keine Oberklasse, aber mindestens eine Unterklasse enthalten ist, eine neue Sicht mit der Vererbungshierarchie dieser Klasse erzeugt werden.

ANHANG A

Abbildungsverzeichnis

ANHANG B

Literaturverzeichnis

[Bea97] Sun Microsystems, Inc. *JavaBeans API Specification 1.01, Online at http:// java.sun.com/products/javabeans/docs*, 1997.

[Boo94] Grady Booch. *Objektorientierte Analyse und Design*. Addison Wesley, 1994.

[ELW98] R. Eckstein, M. Loy, and D. Wood, editors. *Java Swing*. O'Reilly, 1998.

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/ Cummings, Redwood City, 2nd edition, 1994.

[FNT98] T. Fischer, J. Niere, and L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling (in german)*. Master's thesis, University of Paderborn, Paderborn, Germany, July 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In G. Engels and G.Rozenberg, editors, *Proc. of the 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995.

[Jac95] I. Jacobson. *Object oriented software engineering: a use case driven approach*. Addison Wesley, 1995.

[Ref98] Sun Microsystems, Inc. *Java Core Reflection API, Online Documentation at http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/ java-reflectionTOC.doc.html*, 1998.

[Ros] Rational. *Rose, the Rational Rose case tool. Online at http://www.rational.com*.

[Rum94] James Rumbaugh. *Objektorientiertes Modellieren und Entwerfen*. Hanser, 1994.

[SDL96] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 1994 + Addendum 1996.

[Tog01] TogetherSoft. *Together 4.2, UML Case-Tool, Online at http://www.togethersoft.com*, 2001.

[UML] Rational Software Corporation. *UML documentation version 1.3 (1999). Online at http://www.rational.com*.