Ludwig-Maximilians-Universität München
Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik

Fortgeschrittenenpraktikum im Studiengang Informatik

Entwicklung eines Modellierungstools für Aspekt-Orientierte Klassendiagramme

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Gefei Zhang

vorgelegt von Susanne Wagner
suzannew@web.de

München, 13.11.2006

Im Rahmen dieses Fortgeschrittenenpraktikums wurde ein Regelwerk zur Verwebung von Aspekten in ein UML-Klassendiagramm entwickelt und prototypisch in einem UML-Editor umgesetzt.
In der vorliegenden Arbeit wird zunächst eine Einführung in das Konzept der aspekt-orientierten Programmierung gegeben und die verwendete Notation von Aspekten in UML-Diagrammen erläutert. Im Anschluss daran wird das Regelwerk zu Verwebung vorgestellt, das mit Hilfe eines Tools zur Graphtransformation realisiert wurde. Im letzten Abschnitt wird die Erweiterung zum Verweben eines Aspektes in ein Klassendiagramm in einem UML-Editor vorgestellt.

Inhaltsverzeichnis

1 Einleitung 4

2 Aspektorientierung 4

2.1 Grundlagen 4

2.2 Aspekte in UML-Klassendiagrammen 5

3 **Verweben von Aspekten in Klassendiagramme mit Hilfe des Graphtransformationstools AGG 6**

3.1 Der Graphtransfomator AGG 6

3.2 Der Typgraph für vereinfachte UML-Klassendiagramme 9

3.3 Der Weaving-Prozess 11

3.3.1 Verwendete Regeln und NACs 12

3.3.2 Regeln zur Behandlung von Transitivität. 18

3.3.3 Das Mapping 22

4 Prototypische Erweiterung von ArgoUML um Aspektorientierung 23

4.1 Erweiterung in ArgoUML vom Metamodell bis zur Grafik 23

4.2 Implementation des Weaving-Prozesses 26

4.2.1 Übersetzung des UML-Metamodells aus ArgoUML nach AGG und zurück 26

5 **Ausblick 27**

6 Literaturverzeichnis 29

7 Anhang 30

7.1 Beispielhaftes Weaving in ArgoUML und AGG 30

7.2 Verschiedene Weaving-Möglichkeiten 32

1. Einleitung

Aspektorientierte Programmierung setzt sich mit der Verbreitung von AspectJ und zunehmender Integration in Frameworks wie Spring zunehmend durch und ist ein sinnvoller Ansatz, um Separation of Concerns zu erreichen.

Neben der Programmierung fehlen allerdings in der Aspektorientierung noch Methoden zur Modellierung. Ein erster Schritt hierzu ist die Modellierung von Aspekten in Klassendiagrammen, wie sie in [12] definiert ist, dieser Ansatz wird aber momentan von keinem graphischen Modellierungstool unterstützt.

Um es möglich zu machen, in einem gängigen UML-Modellierungstool Aspekte in Klassendiagrammen zu formulieren wird in der vorliegenden Arbeit das UML-Modellierungstool ArgoUML um ein Diagramm erweitert, in dem Aspekte gezeichnet und diese in ein Klassendiagramm verwoben werden können. Dazu wird ein Mechanismus vorgestellt, der den Vorgang des Verwebens mit Hilfe der Graphtransformation realisiert. Dieser Mechanismus nutzt das Graphtransformationstool AGG und baut darin ein Regelwerk zur Übersetzung des Aspekts auf. Durch Ausführen der Transformation wird der Aspekt in das Klassendiagramm verwoben.

1. Aspektorientierung

2.1.Grundlagen

In der Software-Entwicklung versucht der Entwickler aus Effizienzgründen immer, in einem Modul möglichst nur eine einzige Aufgabe zu erledigen. Oft lässt sich dies umsetzen, allerdings nicht, wenn es um Belange geht, die mit anderen Anforderungen eng verwoben sind und sich deshalb durch einen größeren Teil des Codes ziehen. Das Paradebeispiel hierfür ist Logging, bei dem der Programmablauf meist in Logdateien protokolliert wird. [10]

Ziel der aspektorientierten Programmierung ist es, diese logisch vom Rest des Codes unabhängigen Belange, so genannte Querschnittsbelange (engl. Cross Cutting Concerns), vom restlichen Code auch physisch zu separieren. Damit kann Wartbarkeit und Lesbarkeit des Codes verbessert und vor allem Codewiederholung vermieden werden. Aspekte ermöglichen also eine stärkere „Separation of Concerns“, da Querschnittsbelange wie Logging, Transaktionen oder Sicherheitsbelange eigenständig formuliert werden können und der restliche Code von diesen Belangen unberührt bleibt.

Ein Aspekt ist unterteilt in zwei Bereiche, Pointcut und Advice. Im Pointcut wird angegeben, auf welche Stellen im Code der Advice angewendet werden soll, es wird also

ein Einsprungspunkt definiert (Joinpoint), indem die Elemente angegeben werden, die modifiziert werden sollen. Im Advice definiert man die Aufgabe, die vom Aspekt realisiert werden soll, also z.B. das Logging selbst. Die Transformation des vorhandenen Codes, die durch das Ausführen des Aspektes entsteht, wird als Weaving bezeichnet.

Das Paradigma der Aspektorientierung existiert bereits seit einigen Jahren und hat sich als fester Bestandteil in der modernen Software-Entwicklung etabliert. Es existieren bereits viele Erweiterungen für die gängigen Programmiersprachen, bei denen Aspekte in eigenen Dateien definiert werden können und „frühestens zur Übersetzungszeit automatisch in den Programmcode eingefügt werden“ [10]. Das verbreitetste Beispiel ist die Erweiterung AspectJ für Java. Ein sehr guter Ansatz zur Integration von Aspekten und AspectJ findet sich auch im Spring-Framework [9]. Eine wichtige Informationsquelle zur aspektorientierten Programmierung findet sich unter [4].

2.2. Aspekte in UML-Klassendiagrammen

Obwohl Aspekte bereits seit längerem genutzt werden gibt es für sie auf der Ebene des Software-Design noch keine ausreichende Unterstützung. Im wichtigsten Werkzeug des Software-Designs, der UML, existiert bis jetzt noch keine etablierte Möglichkeit, Aspekte zu formulieren.

Möchte man Aspekte in UML-Klassendiagrammen darstellen, so ist zu beachten, dass nur Signaturen modelliert werden, nicht aber das Verhalten. Es ist also nicht sichergestellt, dass Methoden, die den Aspekt realisieren, auch aufgerufen werden. [12]

In diesem Fortgeschrittenenpraktikum wird die in [12] eingeführte, die UML erweiternde Notation verwendet. Diese möglichst einfach gehaltene, das UML-Klassendiagramm erweiternde Notation sieht für einen Aspekt eine eigenes Diagrammelement vor, zwei Beispiele für Aspekte nach dieser Notation sind in Abb. 1 zu sehen. Das Aspekt-Element besteht aus zwei Teilen, die paketähnliche Funktion haben: Pointcut und Advice. Im Pointcut und im Advice können alle Elemente eines Klassendiagramms verwendet werden. Zusätzlich gibt es einen formalen Parameter, dargestellt mit „?“, mit dem man Elemente markieren kann, und die Möglichkeit, Elemente „durchzustreichen“. Diese Möglichkeit wird im folgenden mit „MustNot“ bezeichnet. [12]

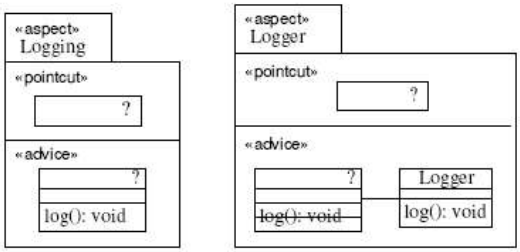

Abbildung 1 Beispiel für Aspekte in Klassendiagrammen aus [12]

Die verwendeten neuen Elemente erweitern die UML. Aspekt, Pointcut und Advice

sind Erweiterungen von Package, der formale Parameter und MustNot sind als Stereotype definiert. [12]
Der formale Parameter wird dazu verwen-det ein Element zu selektieren. Es kann kein formaler Parameter im Aspekt vorkommen oder aber mindestens einer im Pointcut (mehr als einer nur bei einer „or“-Verknüpfung, die in der vorliegenden Arbeit aber nicht behandelt wird) und genau einer im Advice. Bei keiner Selektierung durch einen formalen Parameter wird das Basismodell selektiert und modifiziert. [12]
Ist ein Element im Pointcut mit einem MustNot versehen, so darf dieses Element nicht in der Selektion nicht vorkommen. Im Advice hingegen wird ein Element, das mit einem MustNot versehen ist, bei der Transformation gelöscht. Werden im Advice zum selektierten Objekt neue Elemente hinzugefügt, sind diese also im Pointcut noch nicht vorhanden, so werden diese Elemente bei der Transformation neu erzeugt. [12]
Das Löschen oder Hinzufügen von UML-Attributen oder UML-Operationen kann nur in mit formalem Parameter markierten Elementen stattfinden.
Der Aspekt definiert also eine Modelltransformation, die man als „Weaving“ bezeichnet, wobei der Pointcut zur Auswahl der zu transformierenden Elemente dient und der Advice die Transformation spezifiziert. [12]

3. Verweben von Aspekten in Klassendiagramme mit Hilfe des Graphtransformationstools AGG

Um den Weaving-Prozess auszuführen, d.h. den Aspekt in ein Klassendiagramm zu verweben, wird ein Graphtransformationstool verwendet. Damit sind Basisoperationen und die Freiheit von Seiteneffekten bei der Transformation garantiert.
Im folgenden Abschnitt wird eine kurze Einführung in das verwendete Transformationstool AGG [1] gegeben. Daran anschließend wird erläutert, wie das Weaving mit Hilfe des Transformationstools umgesetzt wurde.

3.1. Der Graphtransformator AGG

AGG ist ein von der TU Berlin entwickeltes Open Source-Tool zur Graphtransformation und besitzt sowohl eine graphische Oberfläche (siehe Abb. 2) als auch eine Schnittstelle für die Verwendung in Java.
Die Abkürzung AGG steht für „Attributed Graph Grammar System“. AGG bietet die Möglichkeit, einen typisierten Graphen, dessen Knoten und Kanten mit Attributen versehen werden können, nach definierten Regeln zu transformieren. Die Regeln des

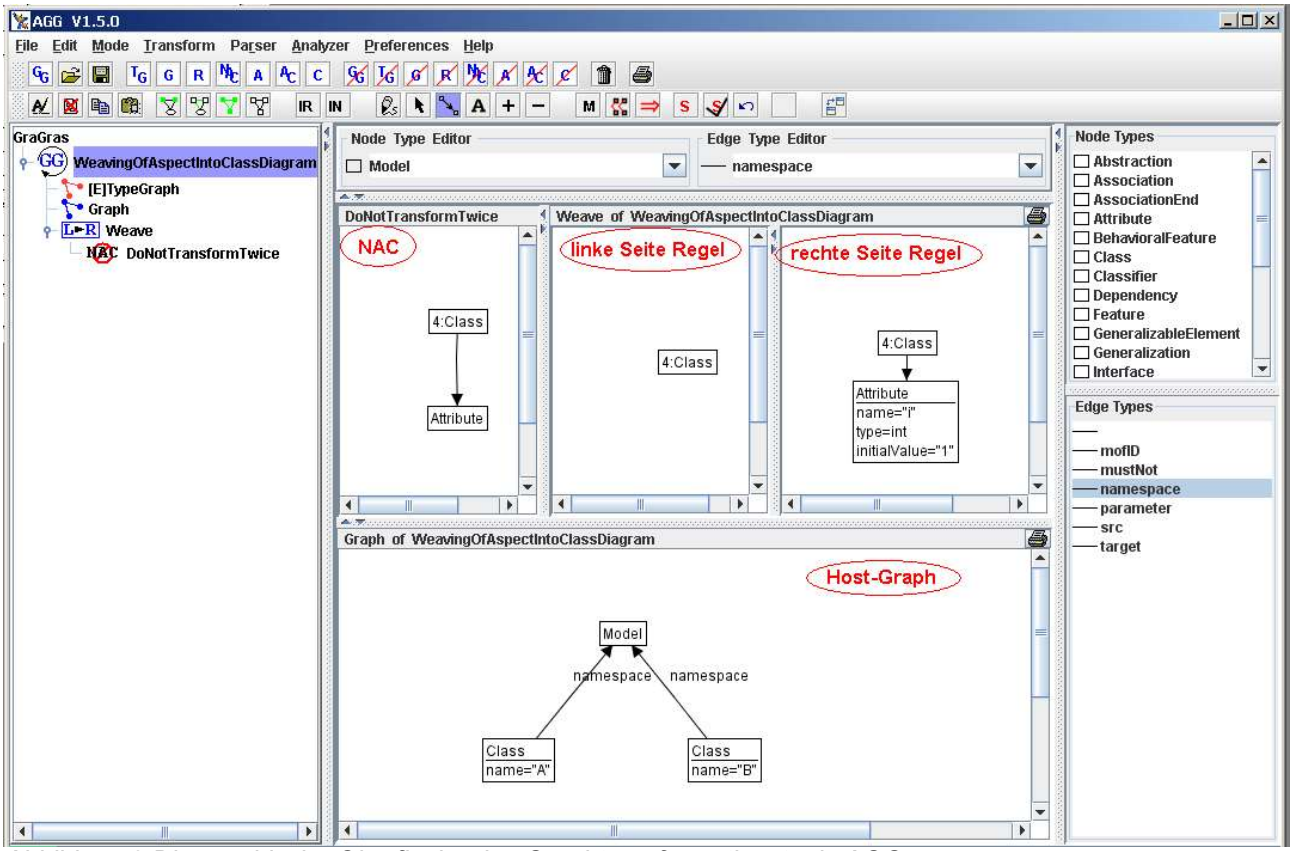

Abbildung 2 Die graphische Oberfläche des Graphtransformationstools AGG

Graphs lassen sich durch eine if-then-Schleifen ähnliche Struktur formulieren [1].

Zur Transformation in AGG sind also mehrere Teile nötig, die in einer mit „GraGra“ benannten Struktur zusammengefasst werden: zunächst muss eine Typisierung in einem sogenannten Typgraph angegeben werden. Danach lässt sich mit Hilfe der Typisierung der zu transformierende Graph zeichnen. Mit Hilfe der Typisierung können nun auch Regeln angegeben werden, wie der Graph transformiert werden soll. Die Regeln sind unterteilt in zwei Seiten, in der linken Seite wird der if-Teil der Regel angegeben, in der rechten Seite der then-Teil, beide Seiten werden über jeweils einen eigenen Teilgraph ausgedrückt. Danach kann die Graphtransformation ausgeführt werden.

Ein Graph besteht aus Knoten und gerichteten Kanten. In AGG werden diese als Objekte des Graphen bezeichnet. Alle Graphobjekte können Attribute besitzen, die ähnlich den Variablen in gängigen Programmiersprachen aus einem Namen und einem Typ bestehen und einen Wert des Typs zugewiesen bekommen können. In AGG können als Attributtypen alle in Java vorhandenen Typen oder auch Java-Klassen verwendet werden. ([8], S. 4)

Der Typgraph legt die Arten von Knoten und Kanten fest, die im zu transformierenden Graph und in den Teilgraphen der Regeln verwendet werden können, es wird also durch den Typgraph die mögliche Syntax für Knoten und Kanten festgelegt. Dazu können im

Typgraph Knoten und Kanten erzeugt und mit Namen und Attributen versehen werden. Die Knoten und Kanten im Typgraph geben damit die Objekttypen vor, von denen in den Graphen Instanzen erzeugt werden können. Gleichzeitig wird im Typgraph auch festgelegt, welche Knoten mit welcher Kanten verbunden werden können, gegebenenfalls können Multiplizitäten für die Kanten angegeben werden.
Die Einhaltung der im Typgraph vorgegebenen Struktur kann in unterschiedlichen Stufen vom Programm überprüft und gefordert werden. Die Stufen reichen von keiner Überprüfung über eine Überprüfung der vorgegebenen Verbindungen von Knoten und Kanten bis zu einer Überprüfung der minimal und maximal angegebenen Multiplizitäten der Kanten.
Mit dem Typgraph lässt sich nun der Hauptgraph, der zu transformieren ist, angeben. Diesem Graph zugeordnet lassen sich Regeln zur Transformation erstellen. Bei einer Transformation werden alle dem Graph zugeordneten Regeln abgearbeitet.
Eine Regel in AGG beschreibt eine Zustandsänderung im Graphen, wobei der Teilgraph der linken Seite dem „Vorher“-Graphen entspricht, der Teilgraph der rechten Seite dem „Nacher“-Graph ([8], S. 6). Im Graph auf der linken Seite wird angegeben, auf welche Objekte eine Regel angewendet werden soll, d.h. es werden Informationen über die Vorbedingungen gesammelt, die erfüllt werden müssen, um die Aktion auszuführen ([8], S. 6) . Die in diesem Graph abgebildete Situation muss sich auch im zu transformierenden Graph wiederfinden, sonst kann die Regel nicht angewendet werden. Sind die Vorbedingungen erfüllt, so spricht man von einem „Match“ ([8], S. 6) . Auf der rechten Seite der Regel wird angegeben, wie die im zu transformierenden Graph selektierten Objekte aus der linken Seite der Regel verändert werden sollen. Dabei besteht die Möglichkeit, gleiche Objekte und Assoziationen zu mappen. Diese würden sonst gelöscht und danach neu erzeugt. Gemappte Objekte werden in der GUI von AGG mit einer gleichen Ziffer vor dem Objektnamen dargestellt, getrennt durch Doppelpunkt ([8], S. 6) . Bei Anwendung der Regel werden alle Veränderungen gemappter Elemente von linker zu rechter Seite im zu transformierenden Graph übernommen. Alle nicht gemappten Objekte der linken Seite werden inklusive Assoziationen gelöscht, alle nicht gemappten Objekte und Assoziationen der rechten Seite werden neu erzeugt.
Passen mehrere Teilgraphen des zu transformierenden Graphs auf die linke Seite der Regel, existieren also mehrere Matches, so wird die Regel in sogenannten „Steps“ innerhalb eines Transformationsprozesses so oft angewendet, bis kein passender Teilgraph mehr im zu tranformierenden Graph gefunden wird.
Um Vorbedingungen formulieren zu können, die für die Anwendung einer Regel nicht

zutreffen dürfen, und um die unendliche Anwendung einer Regel zu vermeiden, gibt es die Möglichkeit, Stopbedingungen zu einer Regel zu formulieren. Dabei wird pro Stopbedingung ein zur Regel gehörender zusätzlicher Teilgraph angegeben. Trifft die in diesem Teilgraph angegebene Situation auf ein Match zu, so wird die Regel auf den Teilgraphen des Matches nicht angewandt. Diese Stopbedingungen werden als „Negative Application Conditions“ (kurz „NACs“) bezeichnet.
Auch in NACs müssen Knoten und Kanten mit dem Teilgraph auf der linken Seite der Regel über ein Mapping als gleiche Objekte assoziiert werden.
Eine Regel in AGG setzt sich also aus linker und rechter Seite und den dazugehörigen Stopbedingungen zusammen.
Ein Beispiel für eine Regel in der graphischen Oberfläche von AGG ist in Abb. 2 zu sehen: In der Bildmitte befindet sich die Regel mit linker und rechter Seite. Die Regel hat eine zugehörige NAC, die links von der Regel zu sehen ist. Unter der Regel ist der Graph zu sehen, auf den die Regel angewendet werden soll, der sogenannte „Host-Graph“. Das Beispiel baut auf einem Typgraph auf, der dem UML-Metamodell für Klassendiagramme entspricht. In der Beispielregel sollen alle Instanzen von *Class*, die im zu transformierenden Graph vorkommen mit einem *Attribut* namens „i“ vom Typ „int“ mit dem Wert „1“ versehen werden. Als Stopbedingung (NAC) soll diese Regel nur auf Klassen angewendet werden, die noch kein Attribut, egal welchen Werts, besitzen. Da der Host-Graph zwei Objekt-Instanzen vom Typ *Class* ohne Attribute enthält, würde die Anwendung dieser Regel eine Transformation in zwei Steps zur Folge haben: beide Instanzen vom Typ *Class* würden als Match erkannt und mit jeweils einem wie in der Regel beschriebenen *Attribut* versehen.
Werden für einen Graph mehrere Regeln definiert, so ist es möglich, diese in Schichten, so genannte „Layers“, zu unterteilen. Bei einer Transformation werden die Regeln dann schichtweise von unten nach oben abgearbeitet. Befinden sich mehrere Regeln auf einer Schicht, so werden diese nichtdeterministisch abgearbeitet.

3.2. Der Typgraph für vereinfachte UML-Klassendiagramme

Um ein UML-Klassendiagramm in AGG zu transformieren bzw. einen Aspekt zu verweben muss zunächst das UML-Metamodell im Typgraph abgebildet werden. Da der zur Umsetzung des Prototypen verwendete UML-Editor das Metamodell der UML 1.4 verwendet, basiert der Typgraph auf UML 1.4.2, wie es von der OMG spezifiziert ist [7]. Im Typgraph werden nur Elemente berücksichtigt, die für ein Klassendiagramm nötig sind. Zusätzlich fanden nur Elemente und Strukturen Eingang in den Typgraph, die auch von

dem verwendeten UML-Editor unterstützt werden. Das Modell wird durch diese Einschränkungen deutlich vereinfacht.

Die nötigen Attribute im UML-Metamodell, die vom Typ eines anderen Elementes des UML-Metamodells sind, werden durch eine mit dem Attributnamen benannte Assoziation dargestellt. Dies findet sich beispielsweise beim *Namespace*. Andere Attribute werden vereinfacht nur als Textfeld aufgenommen, weil die Richtigkeit der Angaben schon über den UML-Editor, aus dem das Metamodell übersetzt wird, gewährleistet ist.

Die verwendeten Elemente, also alle Arten von Knoten und ihre Vererbunghierarchie, sind

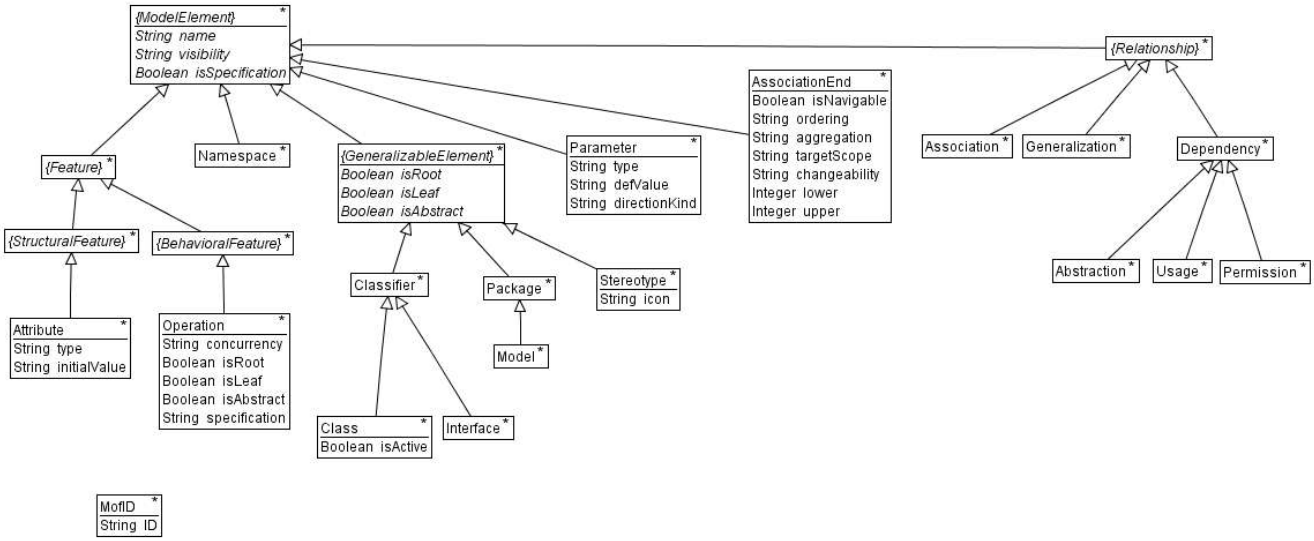

Abbildung 3 Typgraph in AGG ohne Assoziationen

in Abb. 3 dargestellt. Da AGG keine Mehrfachvererbung unterstützt, diese aber in der Spezifikation des UML-Metamodells verwendet wird, wurde versucht die Mehrfachvererbung möglichst logisch und richtig aufzulösen, das heißt aber auch, dass auch andere Varianten des Typgraphs möglich wären. Objekte, die mit {} versehen sind sind abstrakte Objekte und können nicht instantiiert werden. Das Objekt *MofID* wird nur zur richtigen Übersetzung der Objekte aus dem Metamodell nach AGG und zurück benötigt.

Die zwischen den Knoten möglichen Kanten mit Multiplizitäten sind in Abb. 4 dargestellt. Dabei ist zu beachten, dass diese Abbildung die Struktur im Typgraph von AGG wiedergibt und nicht als UML-Diagramm zu lesen ist. In den Graphen in AGG ist eine Navigation sowohl in Kantenrichtung als auch entgegen derselben möglich. Die gerichteten Kanten, die in AGG mit geschlossener Pfeilspitze dargestellt werden und so auch in der schematischen Darstellung verwendet wurden, entsprechen also einer bidirektionalen Assoziation in der UML. Die Verwendung der Multiplizitäten aber entspricht genau der in der UML. Der Typgraph wurch auch bei den Multiplizitäten auf die

Gegebenheiten des verwendeten UML-Editors angepasst, z.B. kann eine *Dependency* nicht mehrere *ModelElements* als *supplier* oder *client* (im Typgraph ausgedrückt mit *src* und *target*) besitzen.

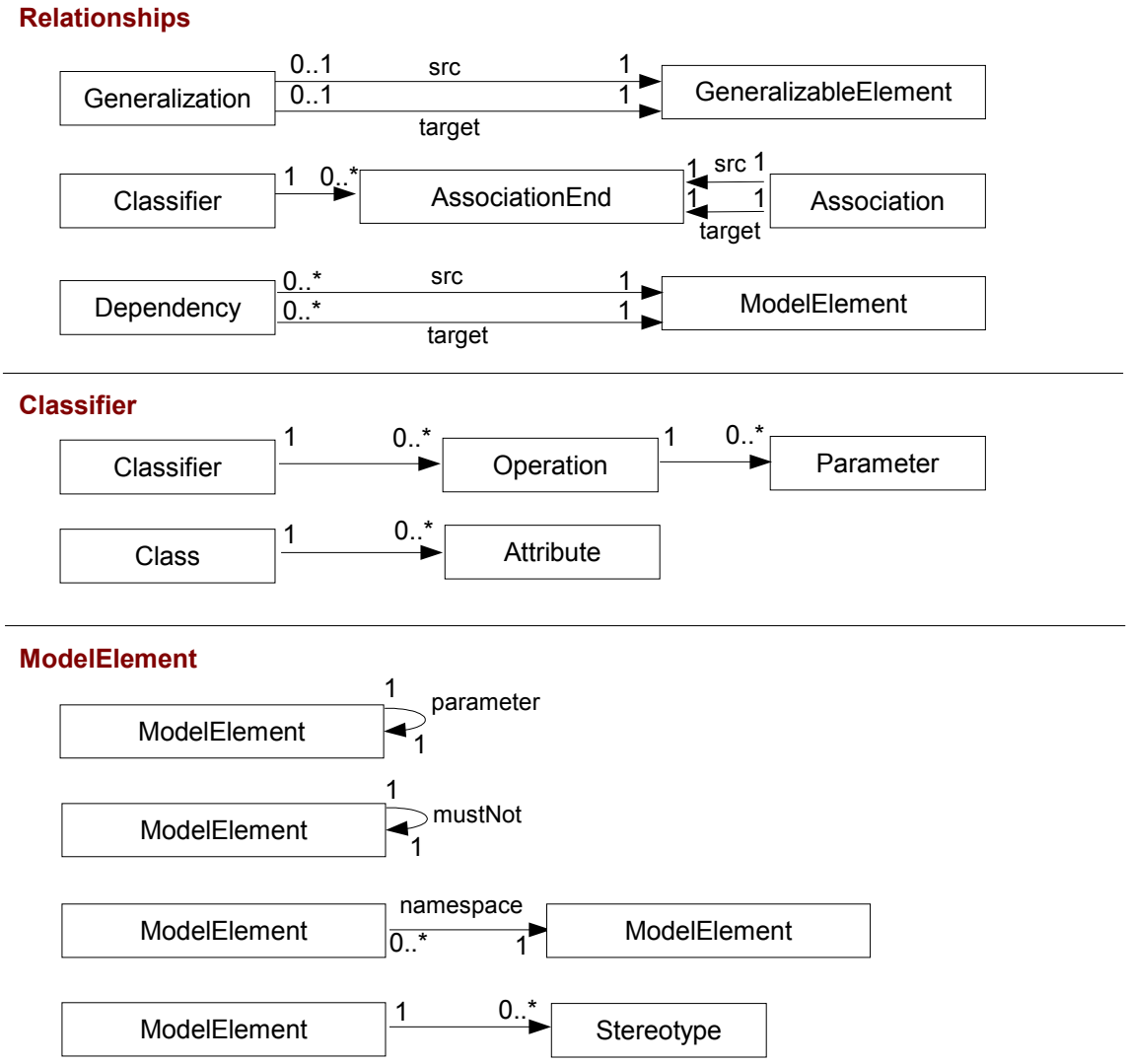

Abbildung 4 Schematische Darstellung der Kanten und Multiplizitäten im verwendeten Typgraph in AGG

Die graphische Darstellung in Abb. 3 ist in Anlehnung an die graphische Darstellung in AGG verfasst worden, allerdings ist aus Gründen der Übersichtlichkeit für jede logische Komponente eine eigene Abbildung angegeben.

Analog zu den unterstützten Operationen im verwendeten UML-Editor gibt es keine Möglichkeit, ternäre Assoziationen in ArgoUML zu erstellen. Für eine Einführung von ternären UML-Relationen in AGG kann [5] herangezogen werden.

3.3. Der Weaving-Prozess

Der zu transformierende Graph, der sogenannte „Host Graph“, besteht im Falle des

Verwebens eines Aspektes aus dem Quell-Klassendiagramm. Das gegebene Klassendiagram wird mit Hilfe des im vorhergehenden Abschnittes angegebenen Typgraphen in einen AGG-Graph übersetzt.
Für die Transformation werden nun noch Regeln erzeugt, die den Aspekt enthalten. Dabei soll sichergestellt werden, dass der Aspekt korrekt in das Klassendiagramm verwoben wird.

3.3.1. Verwendete Regeln und NACs

Soll ein Aspekt in einem Regelwerk in AGG wiedergegeben werden sind mehrere Regeln notwendig.
Die wichtigste Regel ist die Übersetzung des Aspekts selbst. Auf der linken Seite taucht der Pointcut auf, auf der rechten Seite der Advice. Diese Regel, „RuleWeave“, ist Basis für alle anderen Regeln und wird bei Erzeugung der anderen Regeln weiter modifiziert. Haupttransformationsziel dieser Regel ist es, an bestehenden Elementen Attribute zu verändern, neue Elemente zu erzeugen und diese eventuell mit den bestehenden Elementen zu verbinden.
Die Regel „RuleWeave“ ist also zunächst nichts weiter als ein in eine Regel übersetzter Aspekt, bei der der Pointcut in den Graph der linken Seite übertragen wird, der Advice in

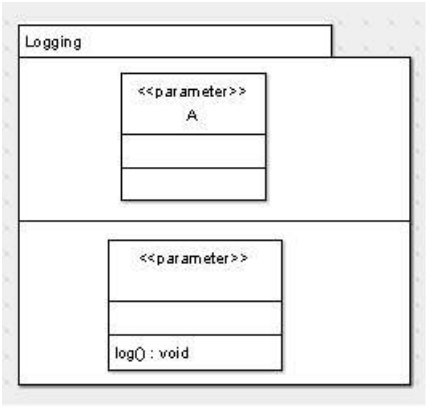

Abbildung 6 Beispiel-Aspekt für RuleWeave

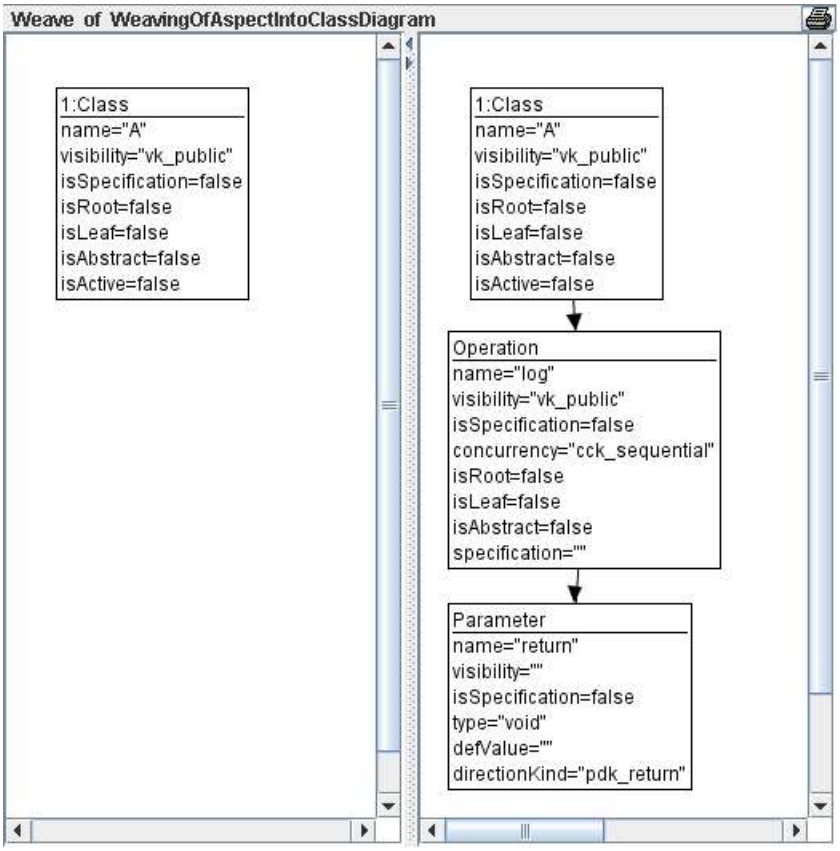

Abbildung 5 RuleWeave in AGG für den Beispiel-Aspekt aus Abb. 5

die rechte Seite. Danach werden alle Elemente und Attribute bestehender Elemente, die auf der linken, aber nicht in der rechten Seite vorkommen und die nicht mit einem MustNot versehen sind, in die rechte Seite der Regel kopiert.

In Abb. 5 ist ein Aspekt zu sehen, bei dem sich im oberen Bereich der Pointcut befindet, im unteren Bereich der Advice. In Abb. 6 findet sich die dazugehörige in AGG formulierte Regel „RuleWeave“ mit Abbildung der linken und rechten Seite der Regel.

Analog zu der in Abschnitt 2 dargestellten Definition von Aspekten ergeben sich folgende zusätzliche Punkte, deren Behandlung jeweils zu eigenen Regeln führen:

Alle Knoten und deren Kinder, die im Advice vorkommen und ein *MustNot* haben sollen aus dem Zielgraph gelöscht werden. Das Löschen von *Attributen* und *Operationen* ist dabei nur in mit *formalem Parameter* markierten Elementen möglich, denn nur diese sollen ja Gegenstand einer Veränderung sein. Diese Aktion ist ausgelagert in eine eigene Regelart „RulesDeleteNodeWithMustNotInAdvice“, weil mit *MustNot* markierte Element im Advice im zu transformierenden Graph auch nicht vorkommen können und dann für die Auswahl eines Teilgraphen für das Ausführen des eigentlichen Weavings, also der Ausführung von „RuleWeave“, keine Rolle spielen sollen. Es wird für jedes *MustNot* im

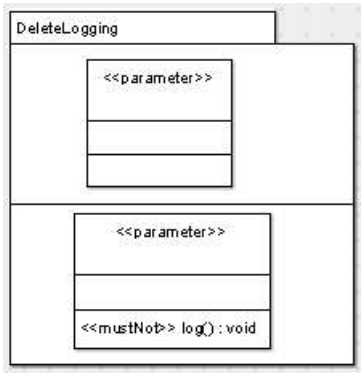

Abbildung 8 Beispiel-Aspekt für RulesDeleteNodeWith-MustNotInAdvice

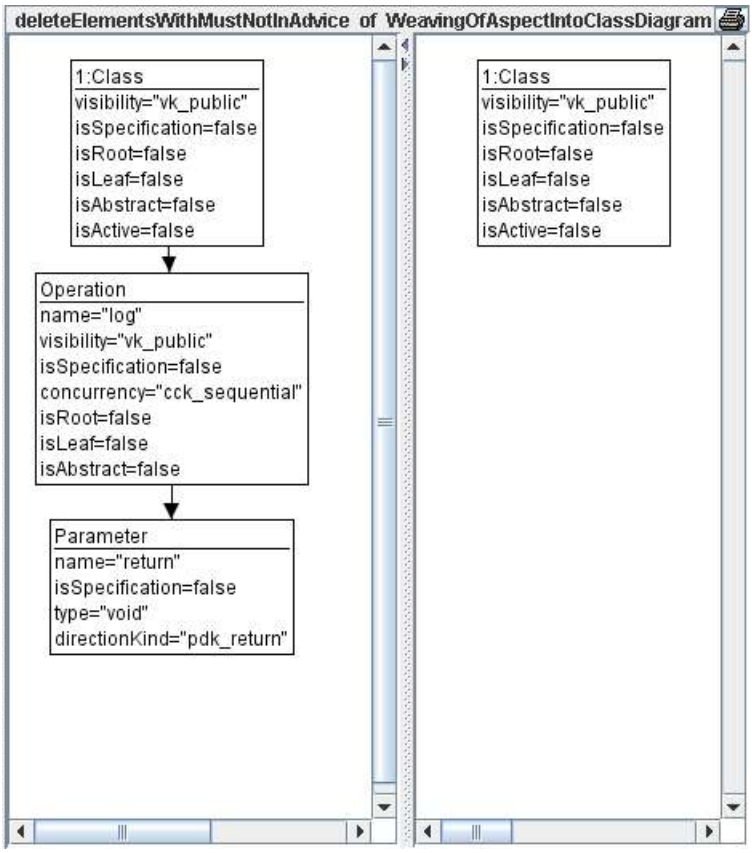

Abbildung 7 RuleDeleteNodeWithMustNotInAdvice in AGG für den Beispiel-Aspekt aus Abb. 7

Advice eine Regel erstellt, deren linke Seite das mit *MustNot* markierte Elemente und alle Unterelemente desselben enthält und deren rechte Seite leer ist. Das zu löschende, also mit *MustNot* markierte Element inklusive aller Unterelemente wird im Anschluss aus der rechten Seite der Hauptregel „RuleWeave“ gelöscht.
Als Beispiel ist in Abb. 8 ein Aspekt angegeben, Abb. 7 zeigt die dazugehörige konkrete in AGG formulierte Regel „RuleDeleteNodeWithMustNotInAdvice“.

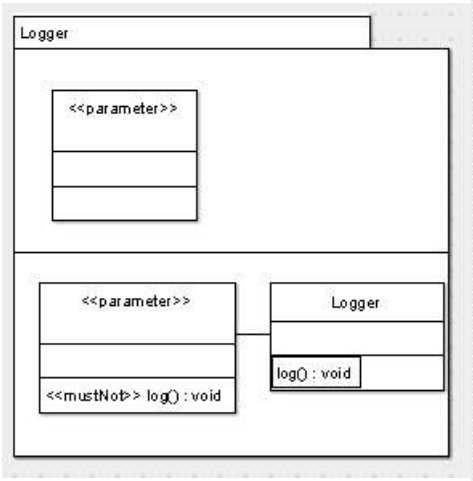

Abbildung 10 Beispiel-Aspekt für RuleCreateElementsOnlyOnce

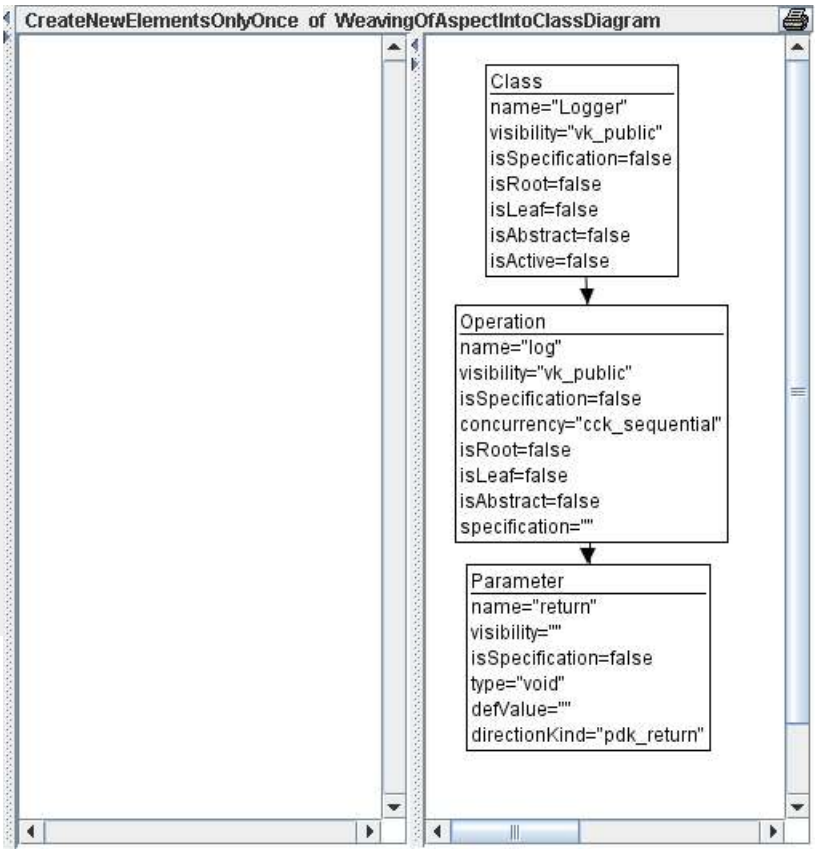

Abbildung 9 RuleCreateElementsOnlyOnce in AGG für den Beipiel-Aspekt aus Abb. 8

Elemente, die nur ein Mal erzeugt werden sollen, egal wie oft die Transformationsregel angewendet wird, müssen in einer neuen Regel erzeugt werden. Das betrifft alle Elemente, die im Advice, aber nicht im Pointcut vorkommen, kein *MustNot* haben und vom Typ *Class, Interface, Package, Model* oder *Stereotype* sind. Im folgenden wird diese Regel mit „RuleCreate-ElementsOnlyOnce“ bezeichnet. In der Regel befinden sich auf der rechten Seite alle neu und einmalig zu erzeugenden Elemente, die linke Seite ist leer oder enthält die für die neuen Elemente nötigen und bereits bestehenden *Namespaces*. Gibt es Elemente, die nur ein Mal erzeugt werden sollen, so muss auch die Hauptregel verändert werden: Die neu und nur einmalig zu erzeugenden Elemente müssen in die linke Seite mitaufgenommen werden.
In Abb. 9 findet sich für diese Regel ein Aspekt als Beispiel, in Abb. 10 die dazugehörige

in AGG übersetzte Regel „RuleCreateElementsOnlyOnce“.

Beim Erzeugen eines neuen Elementes vom Typ *Package* und Befüllen des *Packages* mit bereits bestehenden Elementen müssen die *Namespaces* der bestehenden Elemente geändert werden. Das ist innerhalb des normalen Weavings problematisch, weshalb es in „RuleCreateElementImport“ ausgelagert wird. In dieser neuen Regel befindet sich in der linken Seite der vorherige *Namespace* und die zu ändernden Elemente, auf der rechten Seite die Elemente mit dem beabsichtigte neue *Namespace*.

In Abb. 11 ist ein Aspekt als Beispiel angegeben, bei dem alle Klassen in ein neues Package gegeben werden sollen. Die zugehörige in AGG formulierte Regel „RuleCreate-ElementImport“ ist in Abb. 12 zu sehen.

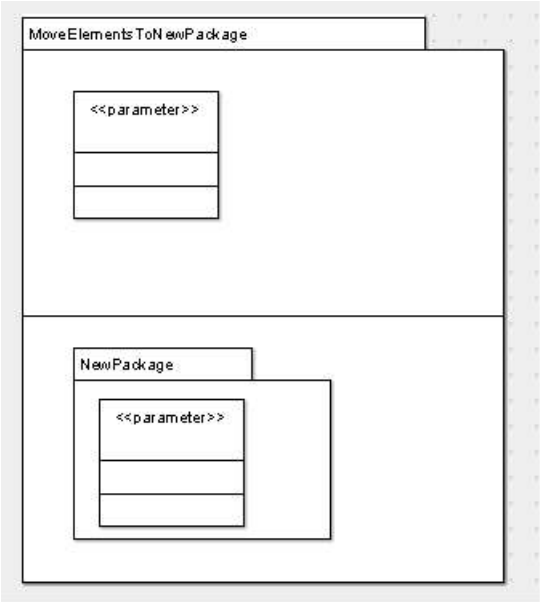

Abbildung 11 Beispiel-Aspekt für RuleCreateElementImport

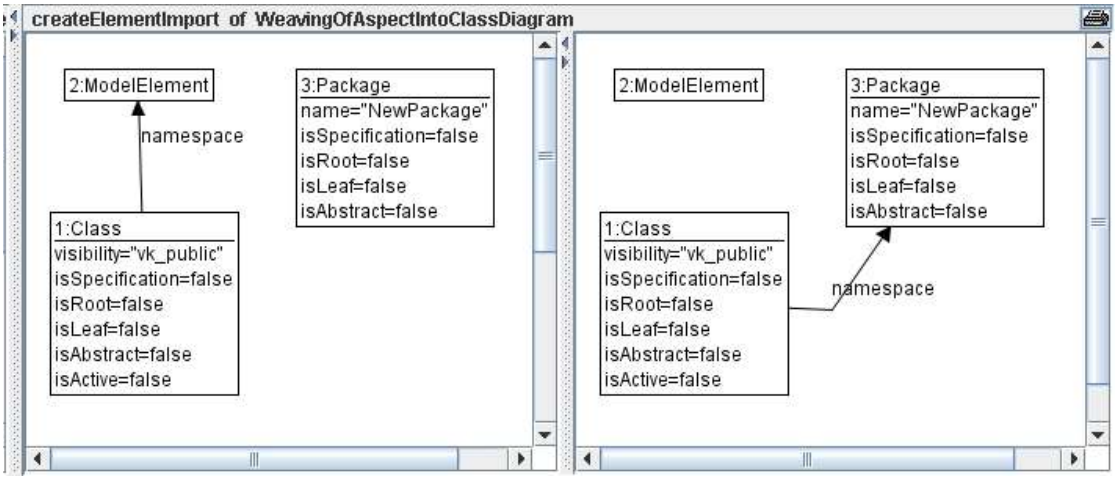

Abbildung 12 RuleCreateElementImport in AGG für den Beispiel-Aspekt aus Abb. 10

Am Ende der Transformation werden alle Elemente gelöscht, die eine Assoziation in der UML darstellen und denen eine Seite, d.h. Quelle oder Ziel der Assoziation, fehlt. In den Regeln „RulesDeleteLooseRelationships“ werden also Elemente vom Typ *Relationship* oder *AssociationEnd*, die nicht Verbindungen in zwei Richtungen haben, gelöscht. Dadurch wird auch die Erfüllung der vom Typgraph geforderten Struktur wiederhergestellt. Regeln von diesem Typ sind statisch und in jeder Transformation enthalten.
Im entstandenen Regelwerk wird jeder Regelart eine eigene Schicht bzw. eigene Schichten zugewiesen:

1. Schicht: RulesDeleteNodeWithMustNotInAdvice
2. Schicht: RuleCreateElementsOnlyOnce
3. Schicht: RuleCreateElementImport
4. Schicht: RuleWeave
5. -7. Schicht: RulesDeleteLooseRelationships

Einige der Regeln würden ohne eine Stop-Bedingung nicht terminieren. Dafür werden „Negative Application Conditions“ (NACs) für die erzeugten Regeln erstellt, die eine Ausführung der Regel verhindern, wenn die in der NAC beschriebenen Bedingungen auf die Selektion zutreffen. Die verwendeten NACs lassen sich wiederum in Gruppen zusammenfassen:
Einige Regeln sollen auf eine Situation nur einmalig angewendet werden, verändern aber die Selektierungssituation unter Umständen nicht, hätten also die unendliche Ausführung einer Regel zur Folge. In diesem Fall wird die rechte Seite der Regel in eine NAC, benannt „DoNotTransformTwice“, kopiert. Dies ist nötig bei der Hauptregel „RuleWeave“, bei „RuleCreateElementImport“ und bei „RuleCreateOnlyOnce“.
Befindet sich im Pointcut ein *MustNot*, so sollen die mit *MustNot* markierten Elemente nicht zur Selektion herangezogen werden. Dazu wird das mit *MustNot* markierte Element inklusive Umgebung in eine zu „RuleWeave“ gehörende NAC kopiert und anschließend das mit *MustNot* markierte Element aus „RuleWeave“ gelöscht.
Ein Beispiel ist im Aspekt in Abb. 13 zu sehen: Es sollen nur Klassen ausgewählt werden, die keine log-Operation haben. In Abb. 14 ist die zugehörige Regel „RuleWeave“ mit linker Seite der Regel in der Mitte, rechter Seite der Regel rechts und auf der linken Seite der Abbildung die NAC mit der gemappten Klasse und der Log-Operation zu sehen.

PSEUDOCODE zur Erzeugung der Weaving-Regeln:

```
Rule ruleWeave = create RuleWeave (aspect.getElementsInPointcut(), aspect.getElementsInAdvice);
      // linke Seite: Elements in Pointcut
      // rechte Seite:  Elements in Advice
   foreach (element in ruleWeave.getElementsInLeft()) {
      if ( ! (ruleWeave.getElementsInRight().contains(element) {
         if ( ! (hasMustNot (element) ) {
            copyElementToRightSide(element, ruleWeave);
         }
      }
   }
createNACDoNotTransformTwice(ruleWeave);
   // NAC: rechte Seite der Regel

//MustNot in Pointcut:
if ( hasLeftSideElementsWithMustNot (ruleWeave)) {
   foreach (elementWithMustNot in leftSideElements) {
      createNAC(elementWithMustNot, ruleWeave);
        //NAC: elementWithMustNot + Kinder
      deleteElementFromLeftSide (elementWithMustNotIncludingChildren, ruleWeave);
   }
}

//MustNot in Advice:
if (mustNotInAdvice) {
   foreach ( elementWithMustNotInAdvice ) {
       create RuleDeleteMustNotInAdvice();
         // linke Seite: ElementWithMustNot in Advice + Kinder
         // rechte Seite: leer
      deleteElementIncludingChildrenFromLeftSide (elementWithMustNotInAdvice, ruleWeave);
   }
}

//Namespace-Berichtigung für bestehende Elemente bei neuem Package:
List newPackages = getPackagesInRightButNotInLeftSide (ruleWeave);
forEach (newPackage in newPackages) {
   create RuleCreateElementImport();
      // linke Seite: ElementsInNewPackage + oldNamespace
      // rechte Seite: ElementsInNewPackage + newPackage als Namespace
   createNACDoNotTransformTwice(ruleCreateElementImport);
      // NAC: rechte Seite der Regel
}

//Elemente, die nur ein Mal erzeugt werden sollen:
List elementsToBeCreatedOnlyOnce = new ArrayList();
foreach (element in ruleWeave.getElementsInRight()) {
   if (! hasMustNot (element) {
      if ( isOfTypeClass(element) || isOfTypeInterface(element) || isOfTypePackage(element)
                || isOfTypeStereotype(element) || isOfTypeModel(element) ) {
         elementsToBeCreatedOnlyOnce.add(element);
      }
   }
}
create RuleCreateElementsOnlyOnce(elementsToBeCreatedOnlyOnce);
   // linke Seite: leer
   // rechte Seite: elementsToBeCreatedOnlyOnce
copyElementsToLeftSide(elementsToBeCreatedOnlyOnce, ruleWeave);
createNACDoNotTransformTwice(ruleCreateOnlyOnce);
   // NAC: rechte Seite der Regel

//statische Regeln zum Löschen unvollständiger Relationships:
create RulesDeleteLooseRelationships();
```

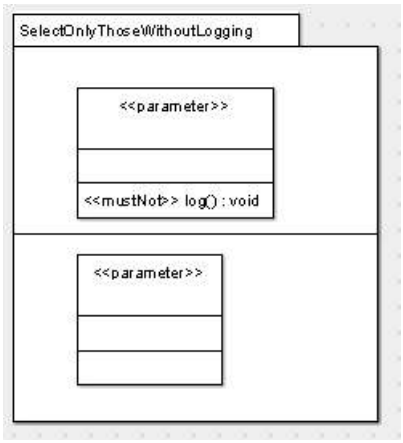

Abbildung 13 Beispiel-Aspekt für NAC bei MustNot in Pointcut

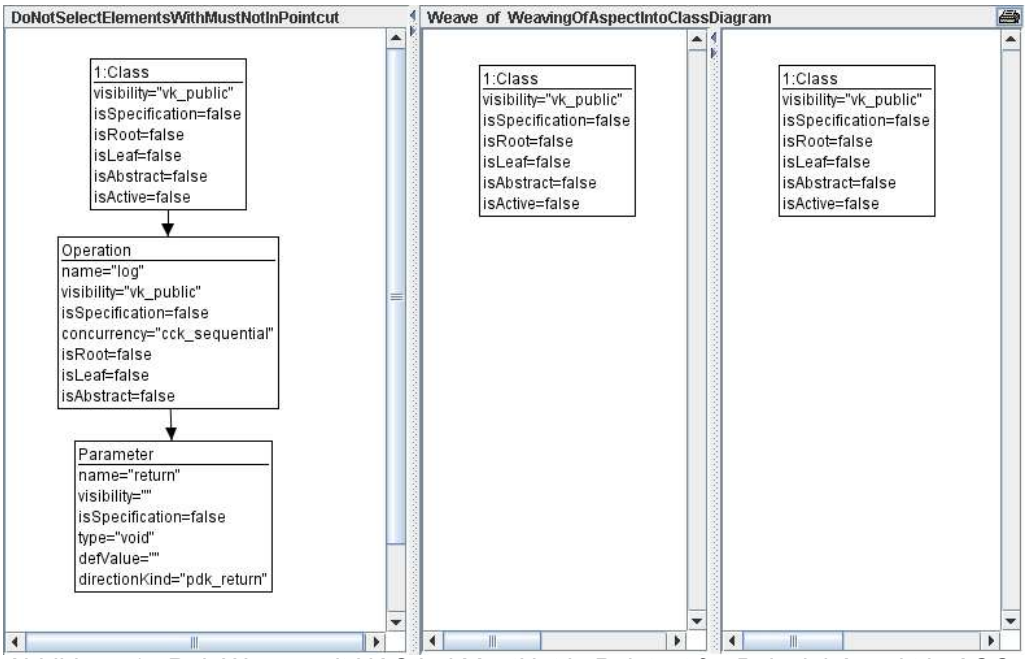

Abbildung 14 RuleWeave mit NAC bei MustNot in Pointcut für Beispiel-Aspekt in AGG

Bei den statischen Regeln zum Löschen von „losen“ *Relationships* sollen nur die *Relationships* zum Löschen selektiert werden, denen auch tatsächlich eine Verbindung fehlt, mittels einer NAC müssen also alle korrekten *Relationships* und *AssociationEnds* ausgeschlossen werden.

Pseudocode, der den Aufbau des Regelwerkes zur Behandlung von Aspekten in AGG veranschaulicht ist auf Seite 17 abgebildet.

3.3.2. Regeln zur Behandlung von Transitivität

Ist ein Element über Assoziationen nicht direkt, sondern indirekt über mehrere Assoziationen und Zwischenelemente mit einem anderen Objekt verbunden, so könnte

man das als eine transitive Verbindung bezeichnen. Eine solche Verbindung könnte für die Formulierung in einem Aspekt von Nutzen sein, weshalb hier ein Regelwerk aufgezeigt werden soll, mit dem man eine solche Verbindung mit Hilfe von AGG finden und behandeln kann.

Eine transitive Verbindung sei also eine Verbindung zwischen zwei *ModelElementen*, wobei die beiden Elemente entweder direkt mittels der in der Verbindung angegebenen Verbindung vom Typ *Relationship* verbunden sind oder aber indirekt über mehrere andere *ModelElements*, wobei alle Verbindungen zwischen den beiden *ModelElements* vom selben Typ wie der in der transitiven Verbindung angegebenen sein müssen, zudem sollen sie die gleiche Richtung aufweisen.

Eine mögliche Notation könnte sein die Verbindung mit einem * zu versehen, um sie als transitiv zu kennzeichnen.

Zwei Beispiele für verschiedene transitive Verbindungen finden sich in Abb. 15 und 18. In Abb. 15 hat eine Klasse A über eine beliebig lange Kette von Assoziationen eine Verbindung zur Klasse B. Abb. 16 und Abb. 17 zeigen konkrete Beispiele für eine transitive Verbindung zwischen Klasse A und Klasse B. In Abb. 18 erbt eine Klasse C über eine beliebig lange Kette von Vererbungen von Klasse D. Das triviale Beispiel dazu findet sich in Abb. 19, ein komplexeres Beispiel ist in Abb. 20 zu sehen.

Abbildung 15 Notation für die transitive Association zwischen einer Klasse A und einer Klasse B

Abbildung 16 Triviales Beispiel fürdie transitive Association aus Abb. 15

Abbildung 17 Komplexes Beispiel fürdie transitive Association aus Abb. 15

Abbildung 18 Notation für die transitive Vererbung zwischen einer Klasse C und einer Klasse D

Abbildung 19 Triviales Beispiel fürdie transitive Vererbung aus Abb. 18

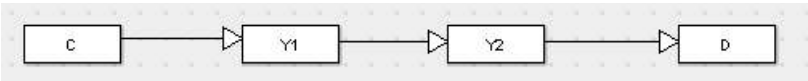

Abbildung 20 Komplexes Beispiel fürdie transitive Vererbung aus Abb. 18

Für die Behandlung einer transitiven Verbindung zwischen zwei Elementen auf Klassenebene sind zwei neue Regelarten nötig:

1. Eine Regelart „SelectNodeWithTransitiveConnection“, die prüft, ob die angegebene transitive Verbindung vom Typ *Relationship* mit der angegebenen Richtung zwischen den beiden angegebenene ModelElements besteht
2. Eine Regelart „DoActionWhenConnectivityIsFound“, die bei Bestehen der Verbindung die gewünschte Aktion ausführt.

Bei der ersten Regelart gibt es eine unterschiedliche Behandlung für *Relationships* vom Typ *Association* und alle anderen *Relationships*.

Bei allen *Relationships*, die nicht vom Typ *Association* sind kann das UML-Modell für die Verbindung direkt in den linken Teil einer Regel übersetzt werden. Danach muss lediglich der Endknoten, im Beispiel aus Abb. 18 der Knoten D, durch einen leeren allgemeinen Knoten, der für das *Relationship* als Endknoten spezifiziert ist (siehe Schema in Abb. 4), angegeben werden. Im Beispiel aus Abb. 18 ist das, da es sich um eine *Generalization* handelt, ein leerer Knoten vom Typ *GeneralizableElement.* Um diese Behandlung in AGG

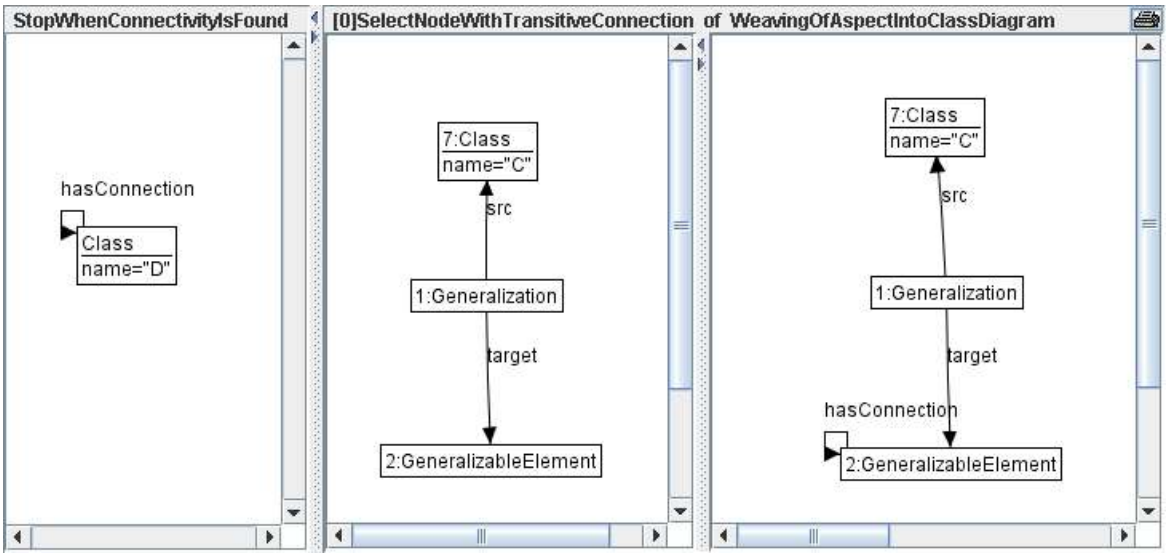

Abbildung 21 Beispielhafte Regel „SelectNodeWithTransitiveConnection“ für Transitivität bei Vererbung

möglich zu machen muss der in Abb. 3 angegebene Typgraph angepaßt werden: Die für *Relationships* nötigen Endknoten *GeneralizableElement*, *Classifier* und *ModelElement* dürfen nicht abstrakt sein. Der entstandene Graph wird nun auch in die rechte Seite der Regel kopiert. Danach werden der Ausgangsknoten auf der linken Seite und der Endknoten der rechten Seite mit einem Marker versehen, hier eine Kante, die wieder auf den Ausgangsknoten zeigt und mit „hasConnection“ benannt ist. Der Marker wird also so lange weitergeschoben, bis ein Knoten gefunden ist, der dem gesuchten Knoten entspricht. Damit die Regel terminiert, braucht sie eine NAC, bei der der Marker am

gesuchten Endknoten, im Beispiel also D, hängt. Diese Regel ist in Abb. 21 zu sehen.

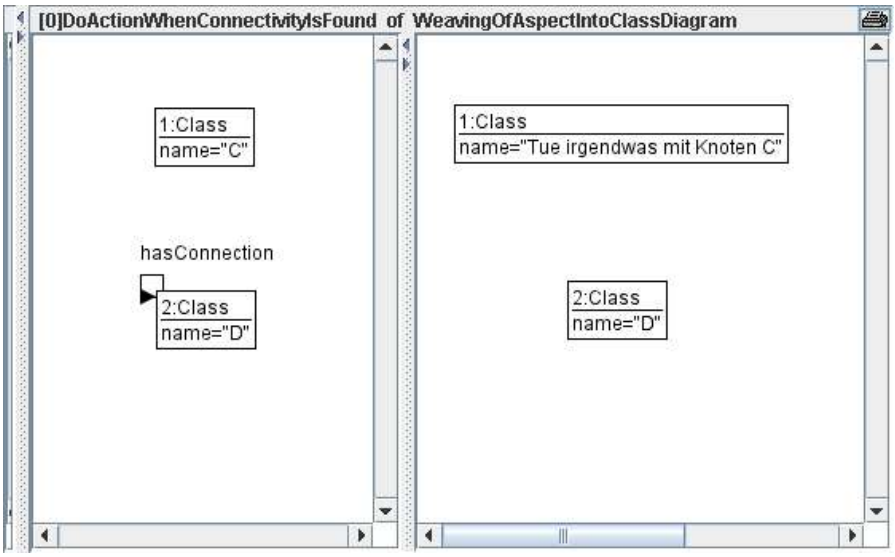

Abbildung 22 Beispielhafte Regel zum Entfernen des Markers und Durchführen einer Aktion bei vorliegender transitiven Beziehung

Ist der gesuchte Knoten gefunden, so ist er mit dem Marker versehen. In diesem Fall kommt die zweite Regel zur Anwendung, die Transitivität besteht. In der zweiten Regel wird der Marker entfernt und die Aktion ausgeführt, die bei bestehender Transitivität ausgeführt werden sollte, im Beispiel soll das die Umbenennung des Knotens „C“ in „Tue irgendwas mit Knoten C“ sein, siehe Abb. 22.

Das Auffinden einer transitiven Verbindung bei einer *Association* ist etwas komplizierter. Hier muss auch der Fall betrachtet werden, dass eine *Association* in beide Richtungen navigierbar sein kann, und deshalb das *AssociationEnd* des Ausgangsknotens sowohl an der Source-Kante als auch an der Target-Kante hängen kann (veranschaulicht in Abb. 23), weshalb man in diesem Fall zwei Regeln braucht. Zudem darf bei einer gesuchten nur in einer Richtung navigierbaren *Association* die Navigierbarkeit in die Gegenrichtung auf keinen Fall angegeben werden, weil sonst alle beidseitig navigierbaren Verbindungen nicht selektiert würden.

Die beiden Regelarten „SelectNodeWithTransitiveConnection“ und „DoActionWhen-ConnectivityIsFound“ müssen wiederum in Schichten abgearbeitet werden, Regelart 1 hat im Beispiel Schicht 0, Regelart 2 die Schicht 1.

Transitivität wird hier nur theoretisch behandelt und ging nicht in die prototypische Implementation mit ein.

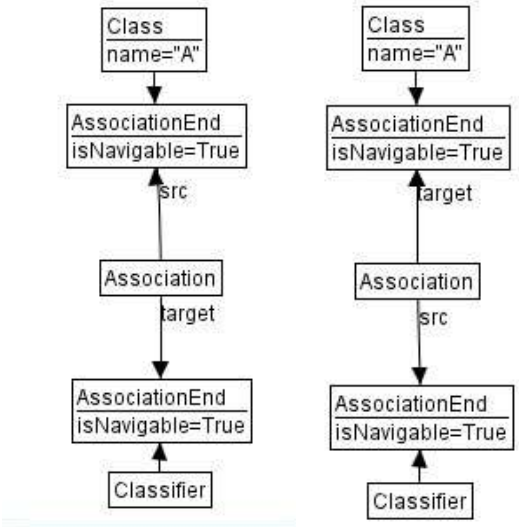

Abbildung 23 Zwei Möglichkeiten für in beide Richtungen navigierbare Association

3.3.3. Das Mapping

Für alle Regeln muss ein Mapping durchgeführt werden, d.h. alle Elemente, die das gleiche Objekt bezeichen und in den verschiedenen Teilgraphen einer Regel auftauchen müssen ein Mapping erhalten. Andernfalls werden sie als unterschiedliche Objekte interpretiert und beispielsweise bei einem Vorkommen in linker und rechter Seite der Regel gelöscht und neu erzeugt. Mappings werden für die Identifikation gleicher Objekte auf linker und rechte Seite einer Regel benötigt und für Objekte der linken Seite einer Regel und jeder zur Regel gehörigen NAC.

Zu mappende Elemente sind Elemente, die mit einem formalen Parameter versehen sind oder Elemente, die gleich sind. Gleichheit bedeutet auf jeden Fall, dass Knotentyp und alle Attributwerte übereinstimmen müssen. Zusätzlich müssen bei Knoten vom Typ *Attribut* oder *Operation* die Elternknoten gleich sein, bei Knoten vom Typ *Parameter* die Elternknoten über zwei Generationen und bei Knoten vom Typ *Relationship* oder *AssociationEnd* müssen die an den beiden Seiten des Knotens hängenden Knoten ebenfalls gleich sein, im Fall einer *Association* zusätzlich die Knoten an den AssociationEnds. Sind diese Bedingungen erfüllt, so werden die beiden verglichenen Elemente gemappt.

Anschließend werden alle Kanten gemappt, deren an der Kante hängende Knoten bereits gemappt sind.

Dieser Mapping-Algorithmus kann beim Kopieren von Knoten und Kanten umgangen werden, in diesem Fall werden Kopie und Vorlage sofort gemappt.

4. Prototypische Erweiterung von ArgoUML um Aspektorientierung

Für die Umsetzung des im letzten Kapitel vorgestellten Weavings eines Aspektes in ein Klassendiagramm wurde im Rahmen der Arbeit ein UML-Editor prototypisch um die Möglichkeit erweitert, Aspekte in einem Diagramm zu erstellen und in ein Klassendiagramm zu verweben. Als Editor wurde ArgoUML [2], ein in Java geschriebener Open-Source-Editor, gewählt. Die Erweiterung ist ausgelagert in ein eigenes Modul „AspectUML“.

Ein ausführliches Beispiel für das Weaving mit AspectUML inklusive der darunterliegenden Transformation in AGG und einige weitere Weaving-Beispiele finden sich im Anhang.

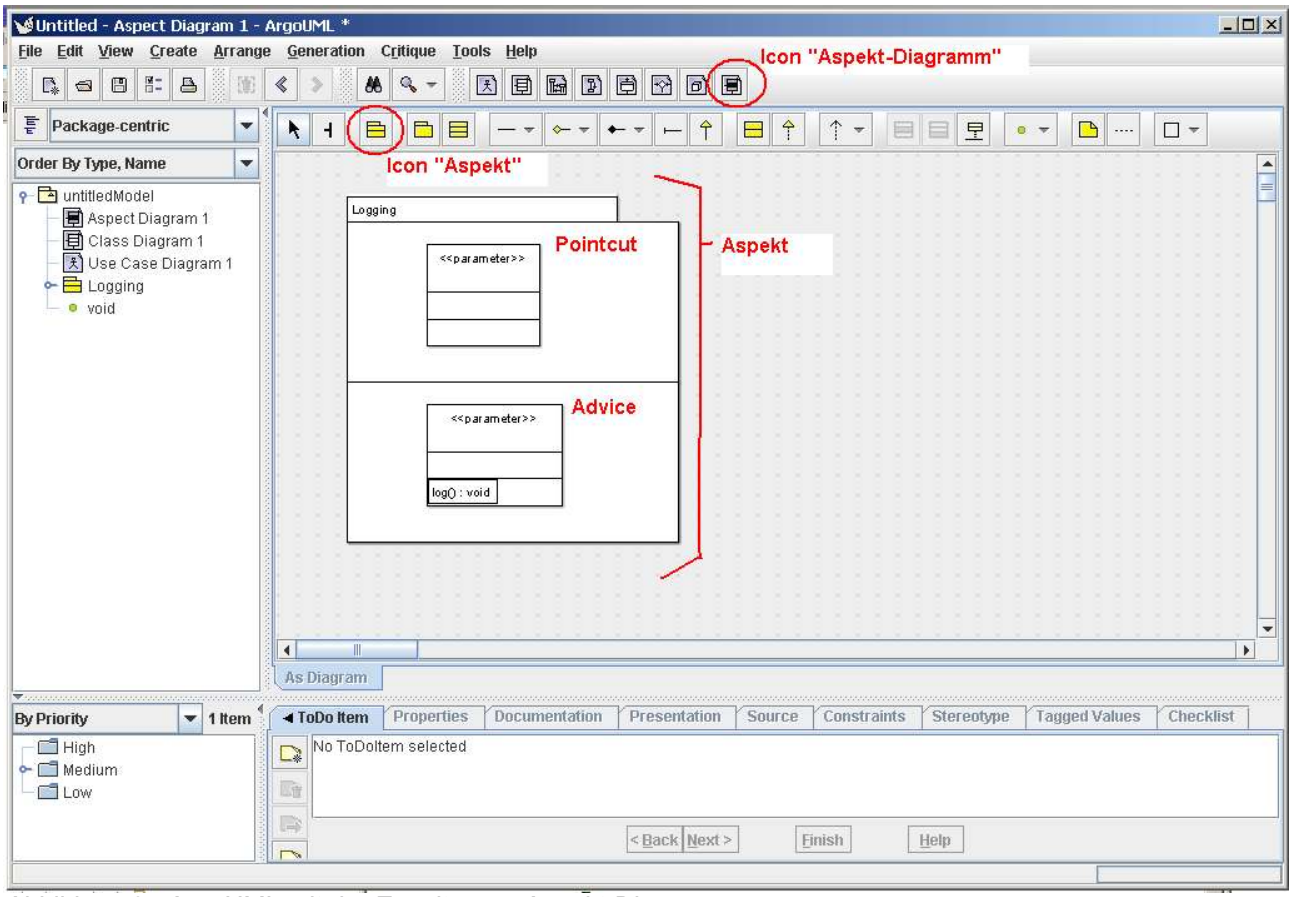

Abbildung 24 ArgoUML mit der Erweiterung Aspekt-Diagram

4.1. Erweiterung in ArgoUML vom Metamodell bis zur Grafik

Das Modul „AspectUML“ führt ein neues Diagramm, das Aspect-Diagramm, ein, in dem man Aspekte zeichnen kann. Dieses Diagramm basiert auf dem UML-Klassendiagramm und erweitert es lediglich um ein neues Element, den Aspekt, und die Möglichkeit, alle anderen Elemente mit *MustNot*, dargestellt mit „<<mustNot>>“ oder einem *formalen Parameter*, dargestellt mit „<<parameter>>“, zu versehen. Ein Screenshot der GUI des

neuen Diagramms findet sich in Abb. 24, in der auch ein Aspekt im geöffneten Aspekt-Diagramm zu sehen ist.
Die Erweiterung in ArgoUML läuft durch alle Schichten, aus denen ArgoUML aufgebaut ist. Als erstes musste das von ArgoUML verwendete Metamodell um den Aspekt inklusive Advice und Pointcut, die alle drei Erweiterungen von *UML-Package* sind, und die Stereotypen *mustNot* und *formalParameter* erweitert werden. Bei der Kompilierung von ArgoUML wird das Metamodell aus einer XMI-Dateien ausgelesen und es werden mit Hilfe von Netbeans MDR Java-Dateien erstellt, die die einzelnen Objekte des Metamodells repräsentieren. Leider werden nicht nur diese Java-Dateien, sondern auch die XMI-Quelldatei zur Laufzeit weiter gelesen, weshalb eine saubere Trennung in das erweiternde Modul und den Source-Code von ArgoUML schon hier Schwierigkeiten bereitet hat.
Nach der Erweiterung des Metamodells und Generierung der dazugehörigen Java-Klassen fehlt nun noch die Erweiterung der Schnittstellen für die neuen Metamodell-Elemente. Diese finden sich im Package „model“ und erweitern die in ArgoUML vorgegebenen Klassen.
Danach wurde über die Main-Klasse des neuen Moduls AspectUML eine Option zum neu Anlegen des Aspekt-Diagramms in der Iconleiste und im Menü unter „Create“ hinzugefügt.
Bei Entwicklung der GUI für das neue Diagramm wurde auf das Cookbook von ArgoUML [3] und eine veraltete Anleitung zum Schreiben von Modulen für ArgoUML [6] zurück-gegriffen. Der Code zur Erweiterung von ArgoUML um das neue Diagramm erweitert im wesentlichen den Code für die GUI des Klassendiagramms. Dabei wird eine neue sogenannte „Fig“, d.h. eine Figur für die Darstellung des Elements Aspekt, eingeführt, und das über Rechtsklick zu erreichenden Menü für alle anderen Figs, die die Knoten und Kanten des Diagramms repräsentieren, um die Möglichkeit zum Hinzufügen und Entfernen von MustNot und formalem Parameter ergänzt.
Im Aspekt-Diagramm ist es nun möglich, einen Aspekt zu zeichnen, diesen in Pointcut (oben) und Advice (unten) mit Klassendiagrammen zu befüllen und die Elemente in den Klassendiagrammen über durch Rechtsklick erscheindende Menü mit *MustNot* oder *formalem Parameter* zu markieren. Bei Rechtsklick auf den Aspekt erscheint ein Menüpunkt „Weave“, bei dem nach Wählen eines Quell-Klassendiagramms der Aspekt in das Klassendiagramm verwoben und das Ergebnis in einem neuen Klassendiagramm angezeigt wird.
Die entstandene Paketstruktur für das neue Modul findet sich in Abb. 25. Im Weave-Paket finden sich alle Klassen für die Umsetzung des Weaving, alle anderen Pakete sind die für

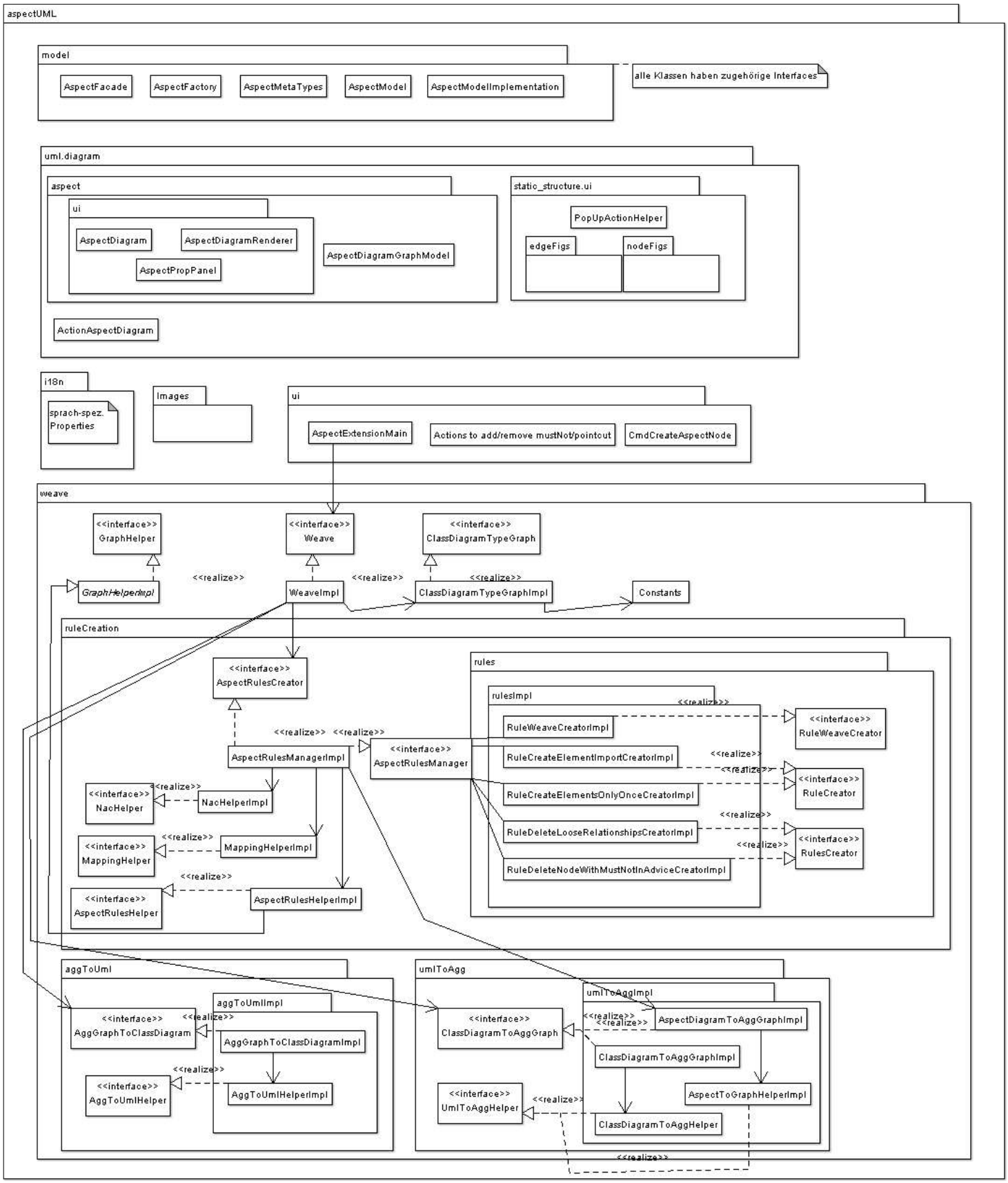

Abbildung 25 Übersicht über die Pakete der ArgoUML-Erweiterung AspectUML

die Erweiterung des Klassendiagramms zum Aspektdiagramm nötigen Klassen, wobei die Struktur der in ArgoUML entspricht.

4.2.Implementation des Weaving-Prozesses

Die Paketstruktur zur Implementation des Weaving-Prozesses befindet sich im Paket „Weave“ und ist in drei Bereiche unterteilt: die Übersetzung des UML-Metamodells aus ArgoUML nach AGG, die Transformation selbst und die Übersetzung von AGG zurück ins von ArgoUML verwendete Metamodell. Der Prozess des Weaving wurde analog zu den in Abschnitt 3 erläuterten Regeln unter Benutzung des AGG-Tools implementiert.
Im Paket „Weave“ findet sich außerdem eine Klasse zum Erstellen des Typgraphen und die Klasse Weave, die die Übersetzungen und die Transformation managt und das neue Zieldiagramm erzeugt und anzeigt.

4.2.1.Übersetzung des UML-Metamodells aus ArgoUML nach AGG und zurück

Zusätzlich zum eigentlichen Weaving-Prozess muss das UML-Metamodell, das in ArgoUML mit Hilfe von MDR aus XMI-Dateien generiert wird und in Form von Java-Klassen vorliegt, in AGG übersetzt werden. Dies geschieht mittels einer händischen Übersetzung der einzelnen Knoten und danach der Erzeugung aller Assoziationen, ebenfalls auf statischem Wege. Genauso wird für den im Aspect-Diagramm gezeichneten Aspekt bzw. dessen Inhalt verfahren.
Nach der Transformation muss der entstandene Graph von AGG in das UML-Metamodell von ArgoUML zurückübersetzt werden. Dies geschieht, ebenso wie die Übersetzung nach AGG, durch statische Methoden.
Die Richtigkeit der AGG-Graphen wird nach der vollständigen Übersetzung von Klassendiagramm und Aspekt und nach erfolgter Transformation mit Hilfe des Typgraphen inklusive der dort angegebenen Multiplizitäten überprüft.
Alle Objekte im Klassendiagramm müssen für die Übersetzung unterschiedlich sein, d.h. im Metamodell mindestens einen unterschiedlichen Attributwert aufweisen, sonst kann es zu Fehlern kommen. Es darf also beispielsweise keine zwei *Klassen* geben, die genau gleiche Eigenschaften haben.
Ein weiteres nicht behandeltes Problem in der Implentation entsteht, wenn im Quell-Klassendiagramm eine *Assoziation* erstellt wird, die von *Klasse* A nach *Klasse* B gezogen wird, im Aspekt aber von *Klasse* B nach *Klasse* A. Dann stimmen im Metamodell die Zugehörigkeiten von Quelle und Ziel der *Assoziation* nicht überein und es kann keine

Übereinstimmung festgestellt werden. Die Gleichheit der *Assoziation* wird also nicht erkannt und es erfolgt kein Mapping.
Nach erfolgreicher Durchführung des Weavings wird ein neuen Klassendiagramm erstellt, das das Ergebnis des Weavings zeigt. Da dieses Klassendiagramm automatisch durch das Modul erstellt und befüllt wird befinden sich die Elemente alle auf einem Punkt im Diagramm, sie liegen in der graphischen Oberfläche übereinander. Deshalb wird nach Beendigung des Weaving-Prozesses ein Algorithmus ausgeführt, der sich um die graphische Anordnung der Elemente des Klassendiagramms kümmert. Die Objekte werden auf dem Klassendiagramm verteilt, *UML-Packages* vergrößert und die Inhalte der *Packages* in den *Packages* verteilt.

5. Ausblick

Die in dieser Arbeit geleistete prototypische Implementierung ist eine Umsetzung der Grundfunktionalitäten des Weaving. Es bleibt weiteren Arbeiten vorbehalten, zusätzliche Features zu implementieren.
Nicht umgesetzt ist das in der Arbeit von Zhang vorgestellten „or“, auch Erweiterungen wie eine Wildcard auf *Operationen* im Pointcut, d.h. weder Parameter noch Rückgabewert interessieren bei der Auswahl einer *Operation*, sind denkbar.
Zudem sind in der vorhandenen prototypischen Implementierung keine Prüfroutinen im Editor implementiert, d.h. Benutzereingaben werden nicht auf Richtigkeit überprüft und die Benutzeroberfläche weist einige kleinere Schönheitsfehler auf. Ebenso fehlen automatisierte Tests.
Generell lässt sich leider sagen, dass ArgoUML als Open-Source-Editor nicht unbedingt der zur Erweiterung geeignete UML-Editor war. Große Schwierigkeiten bereitete die Erweiterung des Editors in einem eigenen Modul, ohne dass Teile von ArgoUML angefasst werden müssen – dies hat sich als für diese Aufgabe unmöglich herausgestellt. Das liegt zum einen an der Umsetzung der Erzeugung von Klassen für das UML-Metamodell als auch an der Programmierung selbst. Einige Klassen beispielsweise sind einfach nicht explizit mit public im Konstruktor und in der Klassendeklaration versehen, was eine Erweiterung aus einem anderen Package, wie es in einem Modul nun einmal der Fall ist, unmöglich macht. Zudem sind einige Strukturen statisch deklariert und nur in der jeweiligen Klasse erweiterbar, so dass das Metamodellpackage der Aspekterweiterung dort ebenfalls statisch eingefügt werden musste (ExtensionMechanismsHelperMDRImpl).
Außerdem unterliegt ArgoUML starken Änderungen und Anpassungen, was prinzipiell

begrüßenswert ist, auch weil im Laufe der Arbeit einige Fehler behoben wurden, die in der Entwicklung des Moduls problematisch waren, aber es ist zeitaufwändig, wenn nach einem Update zunächst einmal das entwickelte Modul nicht mehr läuft oder fehlerhafter Code eingecheckt wurde, der ArgoUML kurzzeitig lahm legt.
Für zukünftige Arbeiten empfiehlt sich also eventuell ein stabilerer und unabhängiger zu erweiternder Open-Source-Editor.
Die Umsetzung der Transformation mit AGG hat sich als sehr erfolgreich erwiesen. Die statische Übersetzung des Metamodells nach AGG und die eigene modifizierte Erstellung des Typgraphs ist verbesserungswürdig, hier könnte man vielleicht eine automatisierte Möglichkeit entwickeln, den Typgraph aus dem UML-Metamodell zu erstellen. Die von AGG bereitgestellten Möglichkeiten zur Regelerstellung und Transformation konnten sehr gut bei der Umsetzung des Weaving-Prozesses verwendet werden.
Alles in allem stellt diese Arbeit nicht nur die prototypische Implementierung der Erweiterung eines UML-Editors, sondern vor allem ein grundlegendes Regelwerk für die Verwebung von Aspekten in UML-Klassendiagramme mit Hilfe der Graphtransformation zur Verfügung.

6. Literaturverzeichnis

[1] AGG Website. http://tfs.cs.tu-berling.de/agg (Stand 13.10.2006).

[2] ArgoUML Website. http://argouml.tigris.org (Stand 13.10.2006).

[3] ArgoUML Cookbook. http://argouml-stats.tigris.org/documentation/defaulthtml/cookbook/ (Stand 13.10.2006).

[4] Aspect-Oriented Software Development Conference Site. http://www.aosd.net/ (Stand 13.10.2006).

[5] Büttner, Fabian und Gogolla, Martin (2004): Realizing UML Metamodell Transformations with AGG. Dept. Of Computer Science, University of Bremen, Germany. In Reiko Heckel, editor, *Proc. ETAPS Workshop Graph Transformation and Visual Modeling Techniques (GT-VMT'2004)*. Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier. http://www.db.informatik.uni-bremen.de/publications/Buettner_2004_GTVMT.ps.gz (Stand 13.10.2006).

[6] De Lamotte, Florent: Making a new module for ArgoUML. http://argopno.tigris.org/documentation/tutoriel/new_module.html (Stand 13.10.2006).

[7] OMG: UML Specification 1.4.2 http://www.omg.org/cgi-bin/apps/doc?formal/04-07-02.pdf (Stand 13.10.2006).

[8] Rudolf, Michael und Taentzer, Gabriele (1999): Introduction to the Language Concepts of AGG. TU Berlin, Germany. http://tfs.cs.tu-berlin.de/agg/Diplomarbeiten/Rudolf.ps.gz (Stand 13.10.2006).

[9] Springframework. http://www.springframework.org (Stand 13.10.2006).

[10] Wikipedia: Aspektorientierte Programmierung. http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung (Stand 13.10.2006).

[11] XMI Reference. http://www.zvon.org/xxl/XMI/Output/index.html (Stand 13.10.2006).

[12] Zhang, Gefei (2005): Towards Aspect-Oriented Class Diagrams. In *Proc. 12th Asia-Pacific Software Engineering Conf. (APSEC'05)*, pages 763-768. IEEE Computer Society. http://www.pst.ifi.lmu.de/veroeffentlichungen/zhang:apsec:2005.pdf (Stand: 13.10.2006).

7. Anhang

7.1. Beispielhaftes Weaving in ArgoUML und AGG

Quellklassendiagramm + Aspekt → resultierendes Klassendiagramm

Abbildung 28 Quell-Klassendiagramm

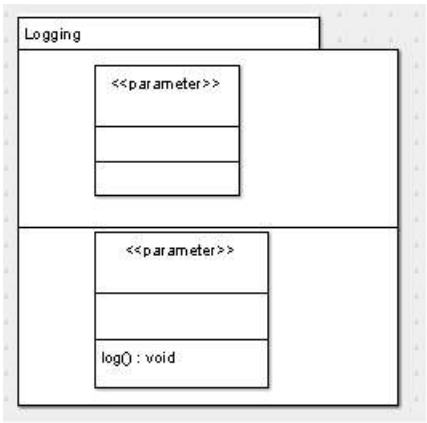

Abbildung 26 Aspekt

Abbildung 27 Aus dem Weaving resultierendes Klassendiagramm

Darunterliegende AGG-Transformation:

Quellgraph + Aspekt-Regelwerk → resultierender Graph

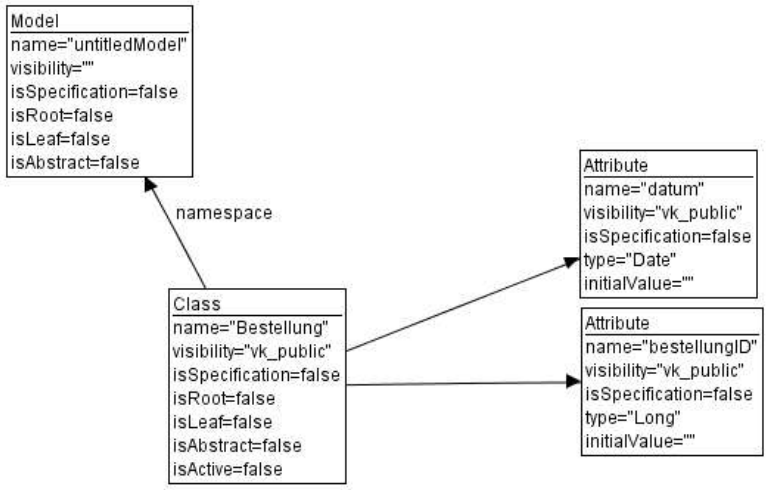

Abbildung 29 AGG-Quellgraph

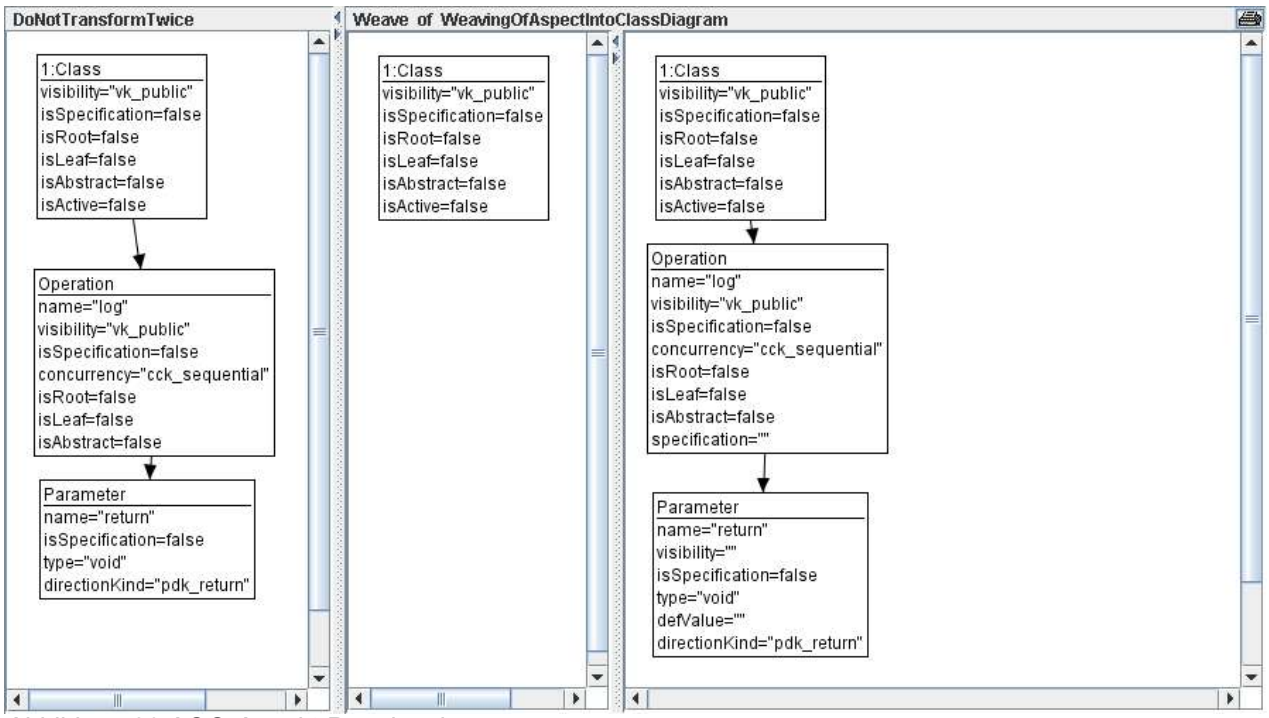

Abbildung 30 AGG-Aspekt-Regelwerk

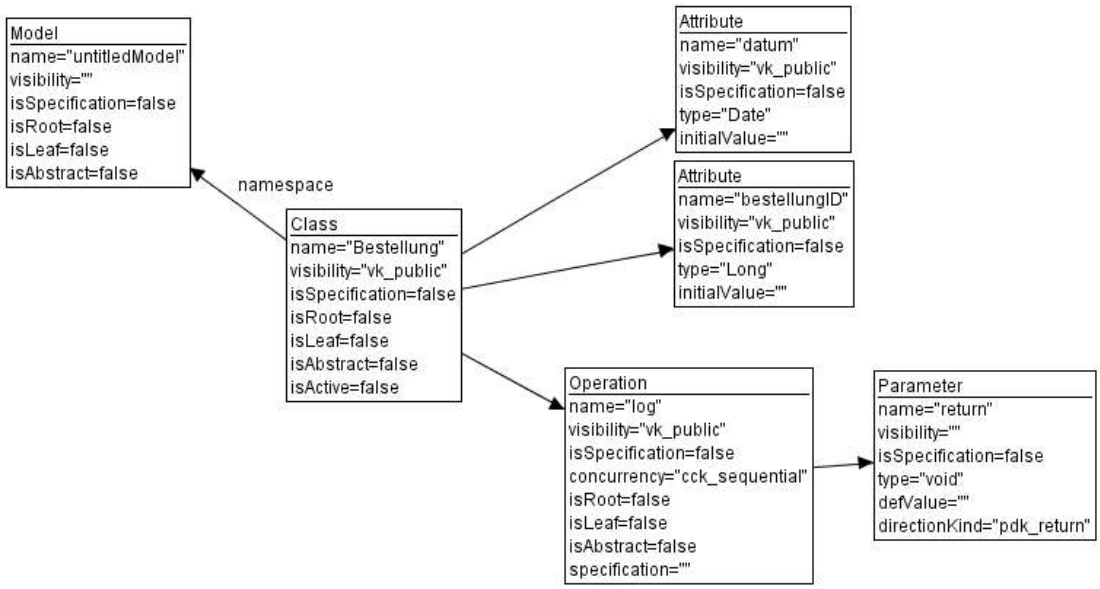

Abbildung 31 AGG-resultierender Graph

7.2.Verschiedene Weaving-Möglichkeiten

(Ist kein Quell-Klassendiagramm explizit für das Beispiel angegeben, so wird das vorhergehende Quell-Klassendiagramm benutzt)

1. Logging

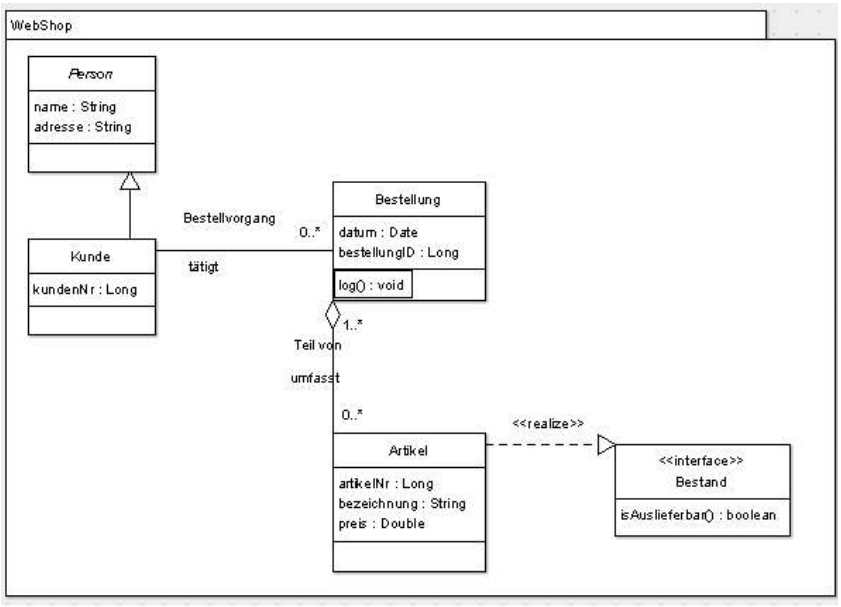

Abbildung 32 Quell-Klassendiagramm "WebShop"

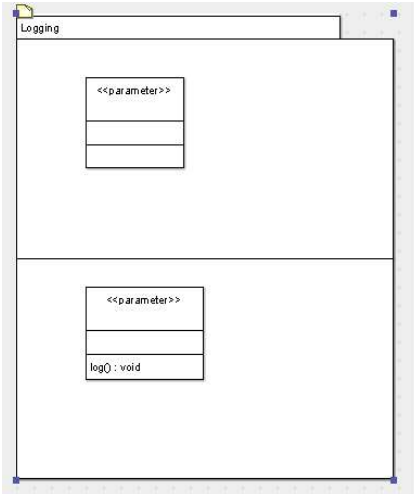

Abbildung 33 Aspekt "Logging"

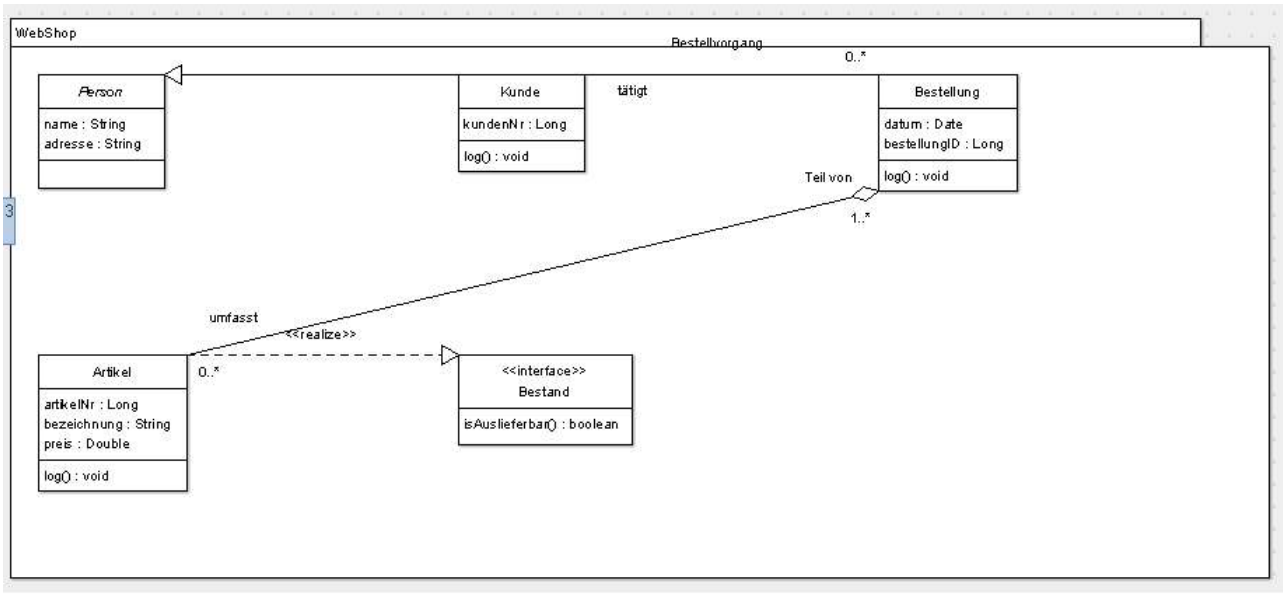

Abbildung 34 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "Logging"

2. Logger

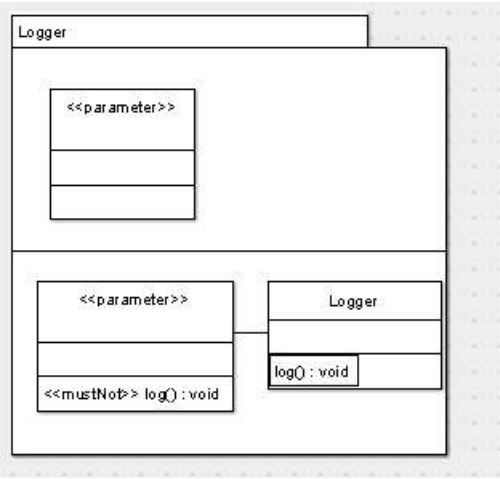

Abbildung 35 Aspekt "Logger"

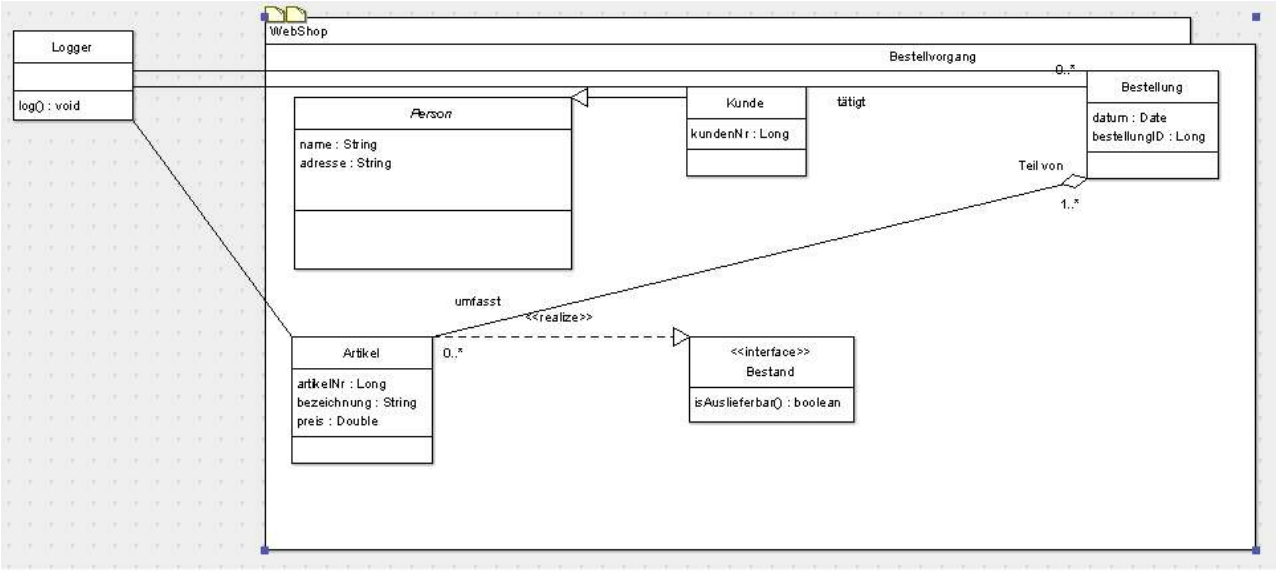

Abbildung 36 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "Logger"

3. Download

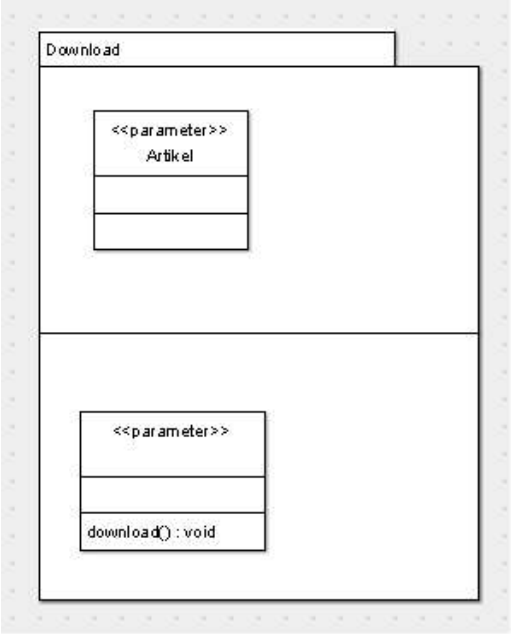

Abbildung 37 Aspekt "Download"

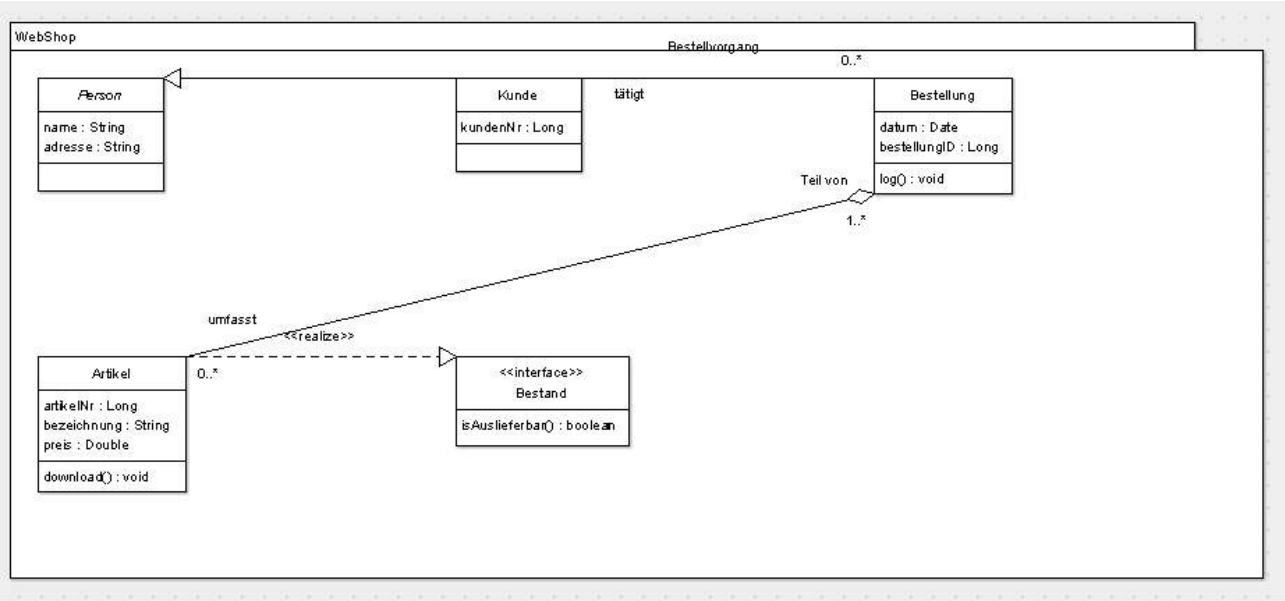

Abbildung 38 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "Download"

4. Subclass

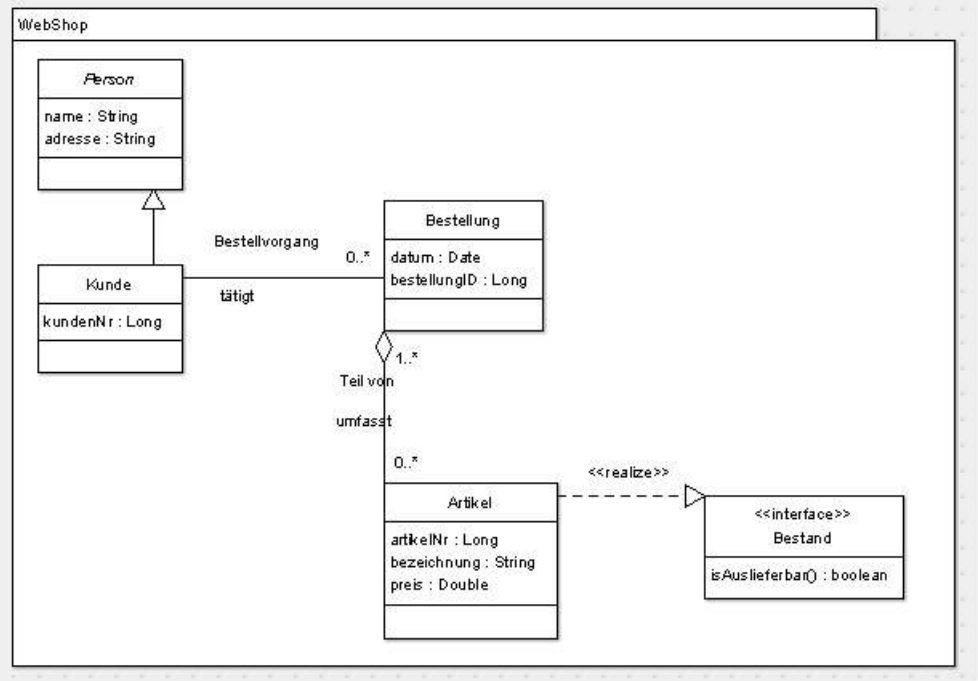

Abbildung 39 Quell-Klassendiagramm

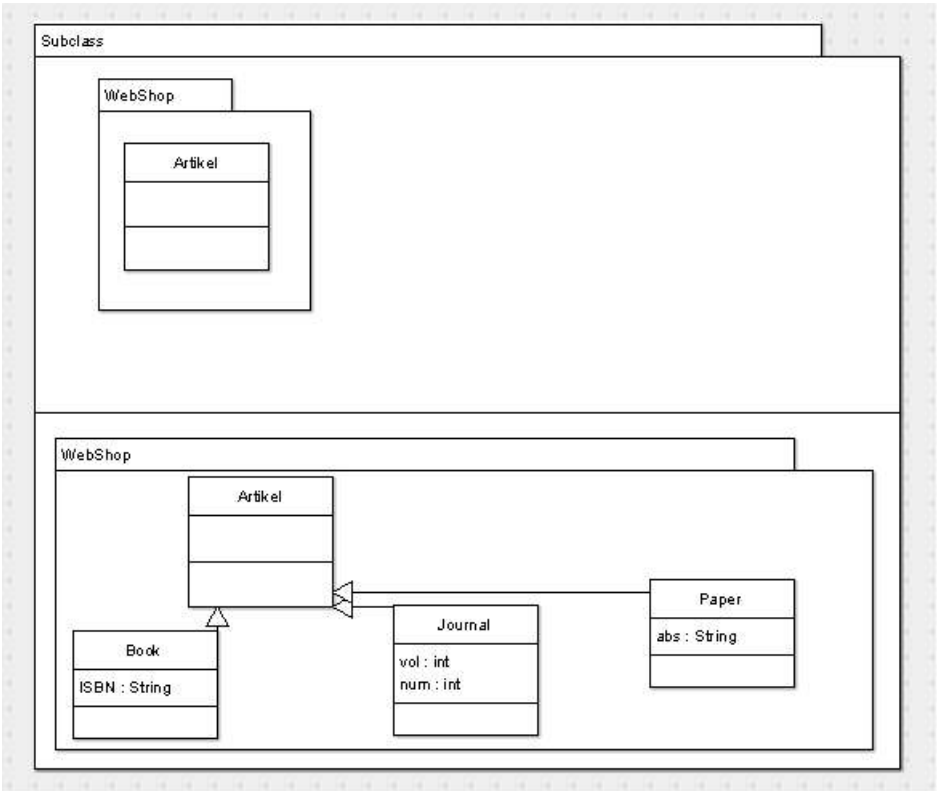

Abbildung 40 Aspekt "Subclass"

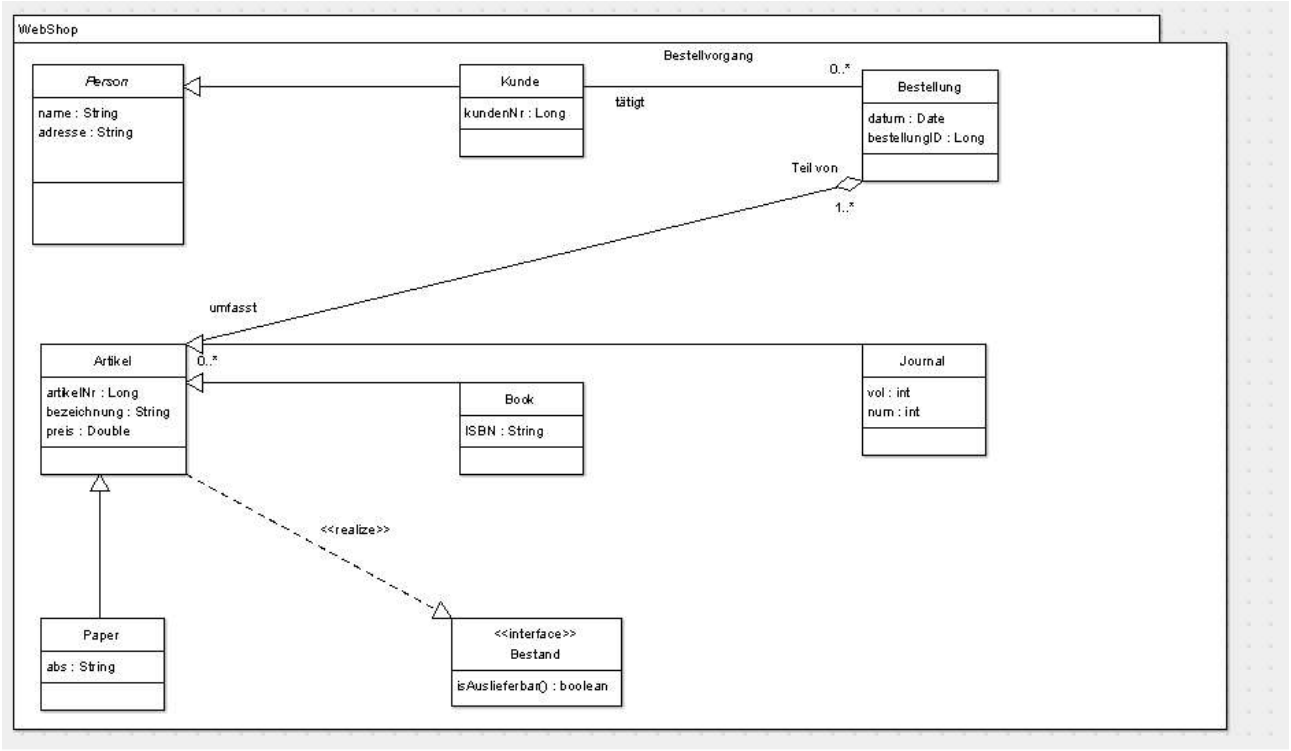

Abbildung 41 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "Subclass"

5. Move Elements to New Package

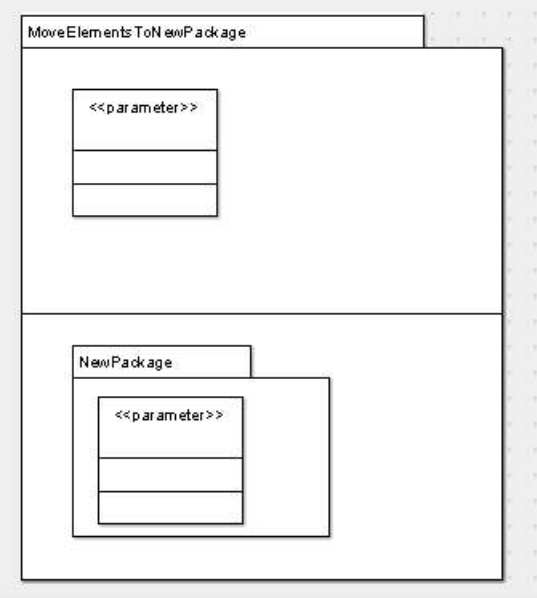

Abbildung 42 Aspekt "MoveElementsToNewPackage“

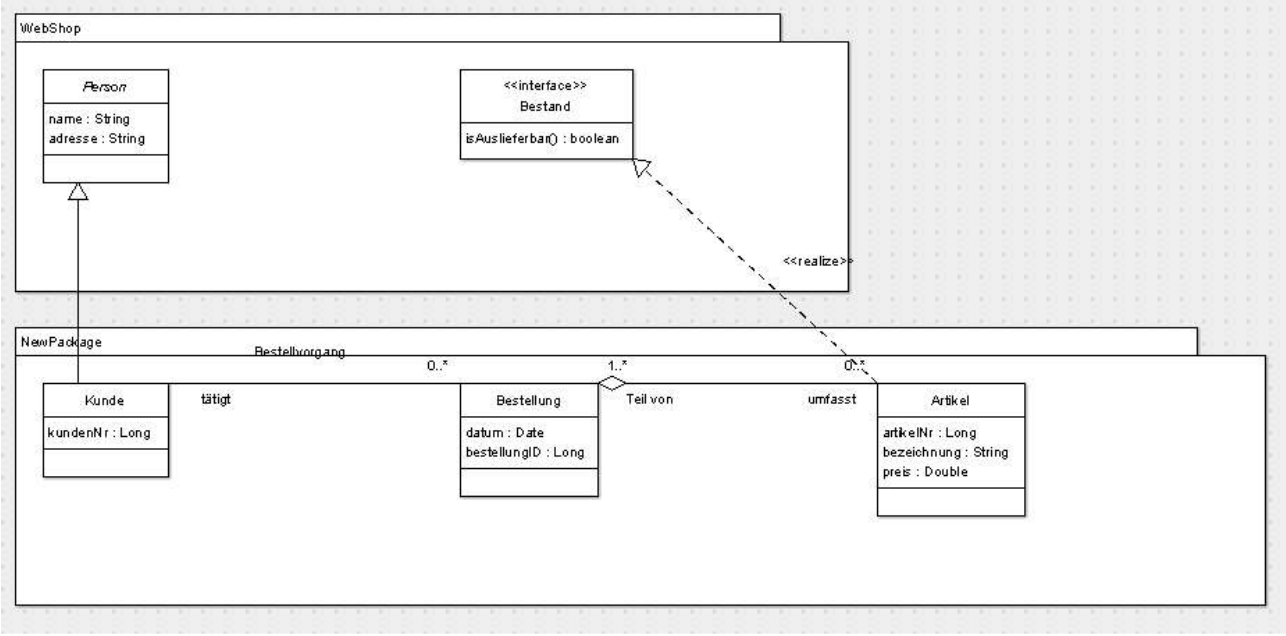

Abbildung 43 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "MoveElementsToNewPackage"

6. Add Behavior (1)

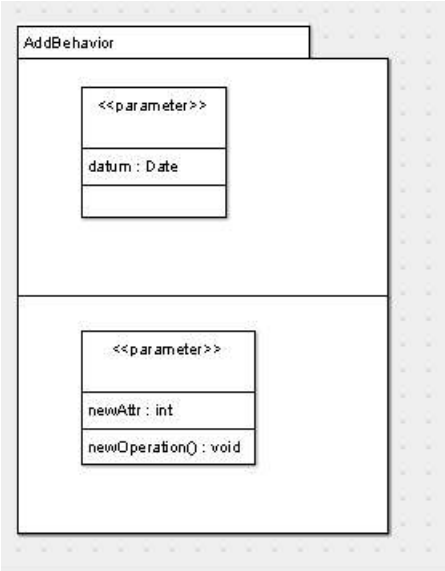

Abbildung 44 Aspekt "AddBehavior"

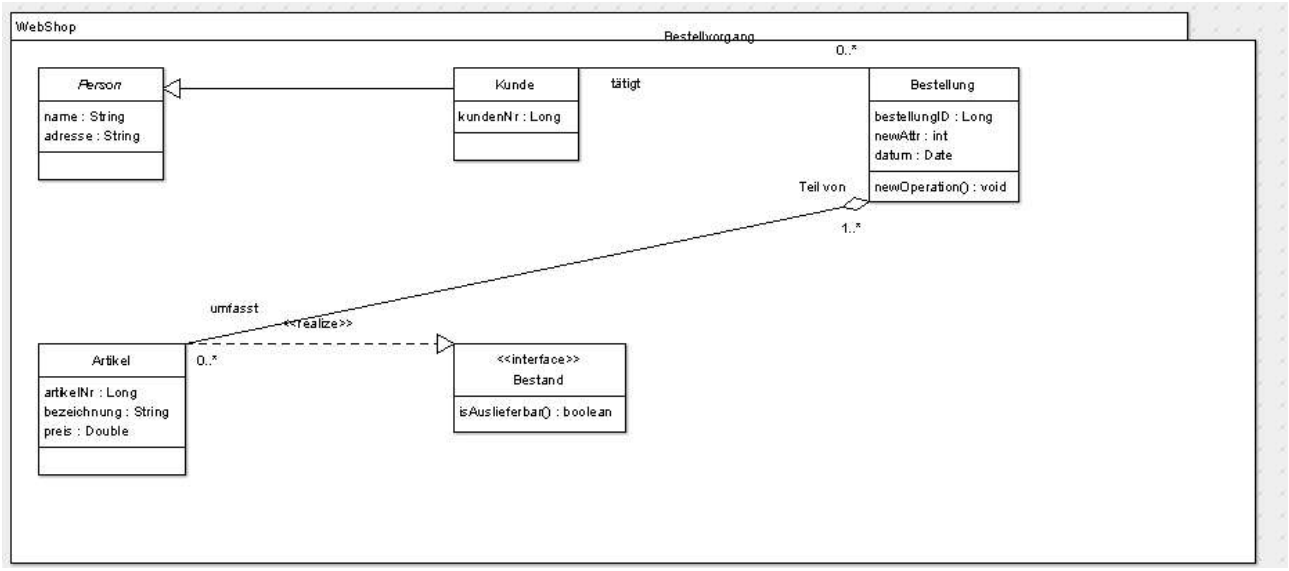

Abbildung 45 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "AddBehavior"

7. Add Behavior (2)

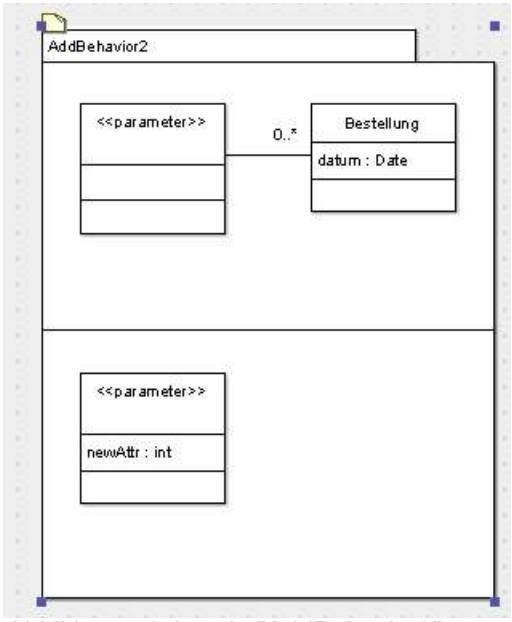

Abbildung 46 Aspekt "AddBehavior2"

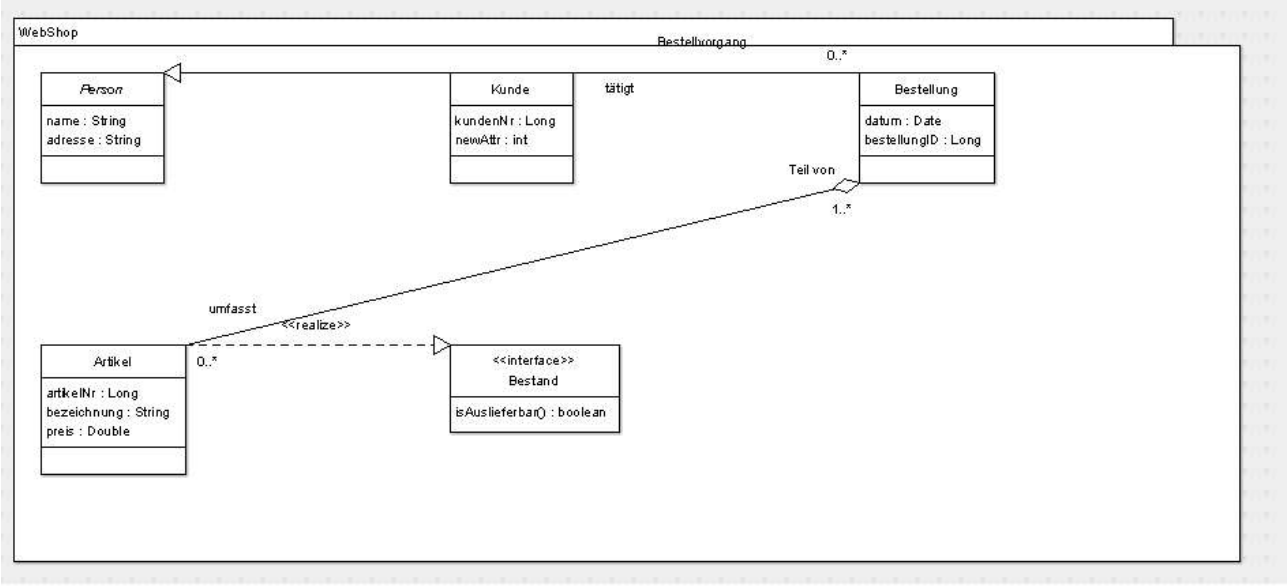

Abbildung 47 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "AddBehavior2"

8. Delete Class

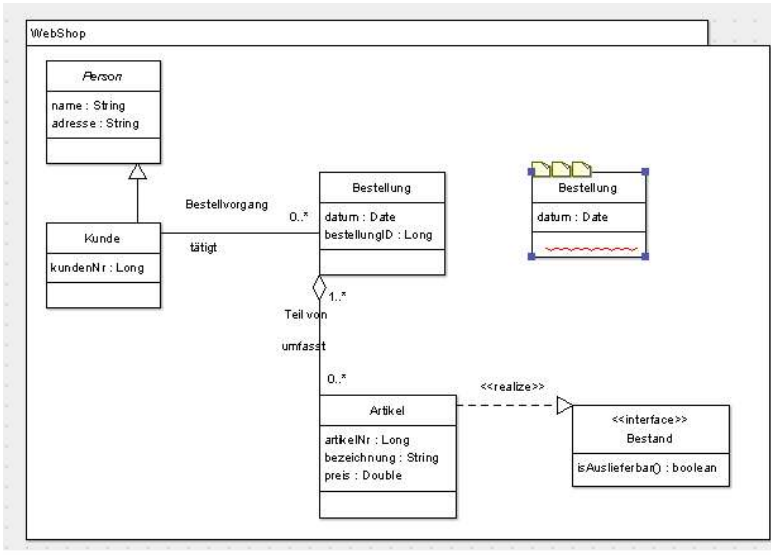

Abbildung 48 Quell-Klassendiagramm

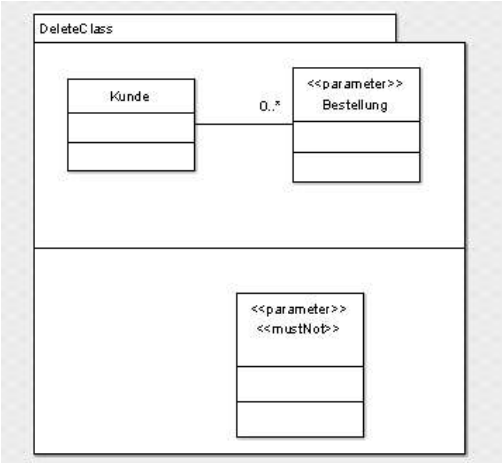

Abbildung 49 Aspekt "DeleteClass"

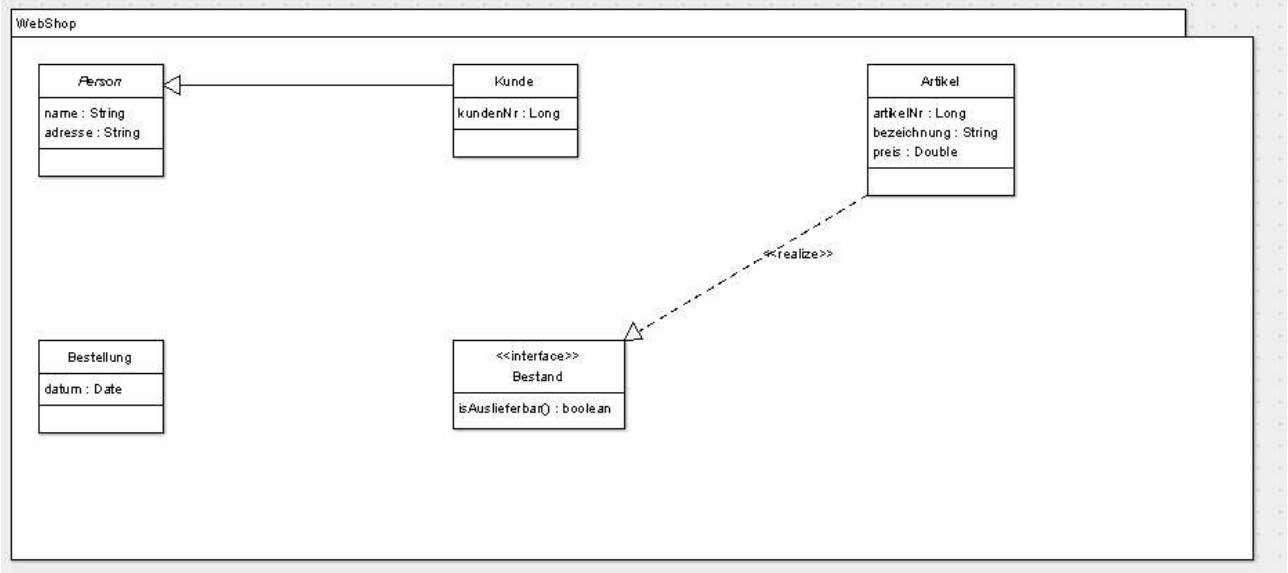

Abbildung 50 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "DeleteClass"

9. Delete Operation

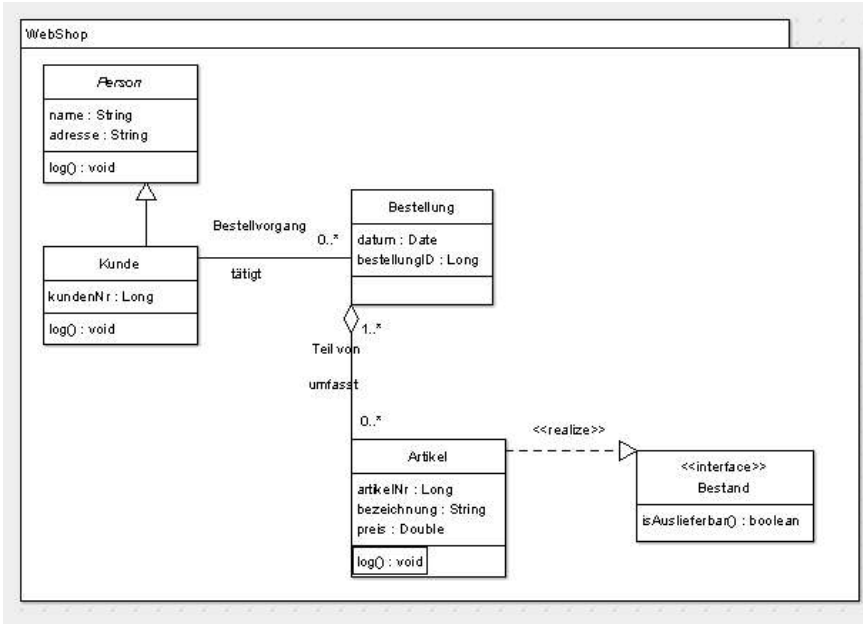

Abbildung 51 Quell-Klassendiagramm

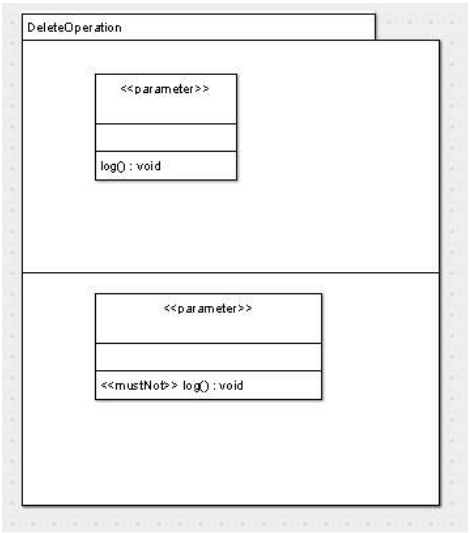

Abbildung 52 Aspekt „DeleteOperation“

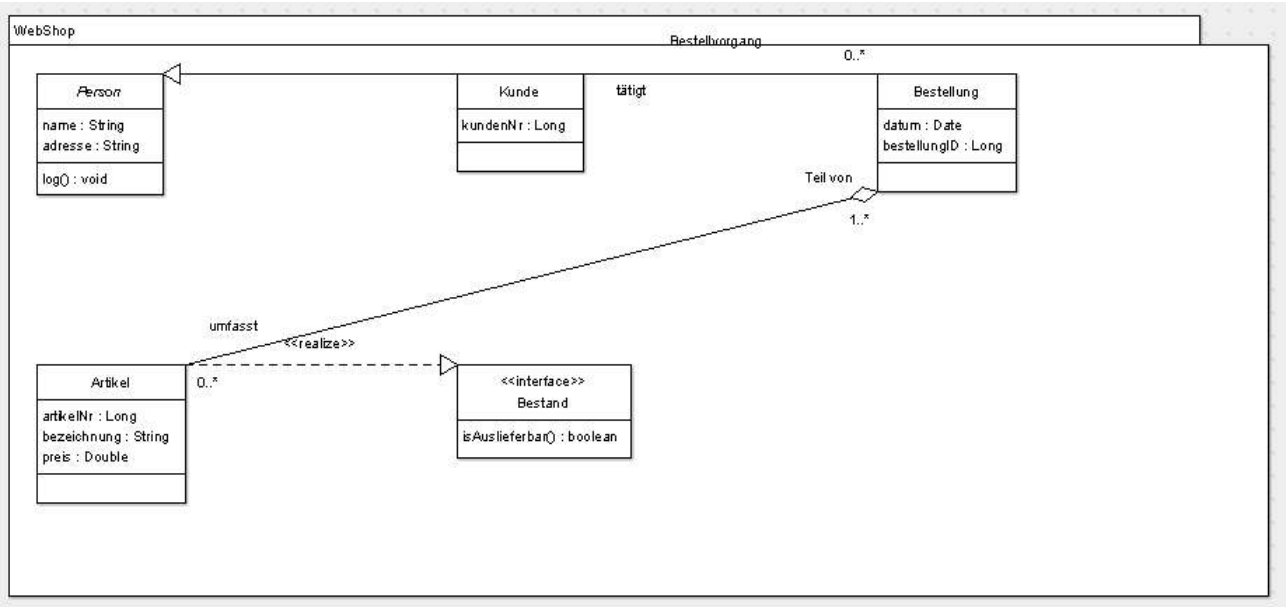

Abbildung 53 Ergebnis-Klassendiagramm nach Weaving mit Aspekt "DeleteOperation"