Universität Paderborn

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

Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets

Diplomarbeit
für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

Lothar Wendehals
Brüderstraße 28
33098 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Gregor Engels

Paderborn, im Oktober 2001

Eidesstattliche Erklärung

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

_____	_____
Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Motivation

1.1. Reverse Engineering

Im praktischen Einsatz befindliche Software unterliegt einer ständigen Veränderung. Sie wird an andere Bedingungen angepasst oder um neue Funktionalität erweitert. Hier kommt es zu Problemen, weil große Softwaresysteme nur sehr umständlich zu warten sind. Erschwerend wirkt sich aus, daß die Softwareentwickler, die das System ursprünglich erstellt haben, oft nicht mehr zur Verfügung stehen. Neue Entwickler müssen sich zum Teil von Grund auf in das Softwaresystem einarbeiten.

Ein anderes, weit verbreitetes Problem ist die fehlende oder zumindest mangelhafte Dokumentation. Zu Beginn der Entwicklung einer Software wird eine Anforderungsanalyse und ein erstes Design erstellt. Während der Weiterentwicklung driftet die Implementierung aber immer weiter von der ursprünglichen Dokumentation ab. Termindruck und fehlendes Bewußtsein zur Notwendigkeit einer Dokumentation lassen die Dokumentation schnell „altern“.

All diese Probleme führen dazu, daß der Aufwand, ein fertiges Softwaresystem zu verstehen und weiterzuentwickeln, enorm wächst. Programme, die den Softwareentwickler bei dieser Arbeit unterstützen, sind also sehr wichtig. Wünschenswert wäre Software, die den Entwickler nicht nur bei der Analyse begleitet, sondern ihn auch bei der Dokumentation unterstützt.

An dieser Stelle setzt das Reverse Engineering ein. Reverse Engineering stellt das genaue Gegenteil zur klassischen Entwicklung dar, bei denen Systeme von Grund auf neu konzipiert und erstellt werden. Elliot J. Chikofsky und James H. Cross II definieren Reverse Engineering wie folgt [CC90]:

> „Reverse Engineering is the process of analyzing a subject to
>
> - identify the system's components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction.“

Der Entwickler hat also die Aufgabe, komplexe Systeme bis in ihre kleinsten Details zu analysieren, um anschließend Veränderungen daran vorzunehmen. Diese Veränderungen können sowohl Erweiterungen als auch Anpassungen an veränderte

Anforderungen enthalten und in ihrem Umfang höchst unterschiedlich sein. Deshalb gewinnt Reverse Engineering in der modernen Softwaretechnik zunehmend an Bedeutung.

Eine große Hilfe bei der Analyse von Software ist die Kenntnis über verwendete Design Patterns, die von Gamma, Helm, Johnson und Vlissides in [GHJV95] vorgestellt worden sind. Sie werden zum Softwaredesign auf einer hohen Abstraktionsebene genutzt, um immer wiederkehrende Probleme zu beschreiben. Dadurch kann der Softwareingenieur Rückschlüsse auf die Funktion gewisser Teile der Software ziehen, die am Design Pattern partizipieren. Er kann also ohne Kenntnis der Details vieler einzelner Teile auf ein zusammenhängendes Ganzes abstrahieren.

Sind bereits bei der ursprünglichen Planung der Software, die der Analyse zugrundeliegt, Prinzipien des Softwaredesigns durch Design Patterns berücksichtigt worden, läßt sich durch die Kenntnis der verwendeten Design Patterns auf Funktionen von ganzen Komponenten schließen. Dadurch entsteht schnell ein Gesamtbild des Softwaresystems.

Aber auch Systeme, bei denen während des Entwurfs nicht explizit Design Patterns benutzt worden sind, können auf Design Patterns hin untersucht werden. Häufig sind unbewußt Design Patterns verwendet worden. Gamma et al. haben immer wieder verwendete und bewährte Lösungsmodelle bestimmter Probleme zusammengetragen und als Katalog sogenannter Design Patterns veröffentlicht. Die Design Patterns wurden also nicht neu erfunden.

Man sieht also, daß das Wissen über Design Patterns in der zu analysierenden Software sehr hilfreich ist. Ein Werkzeug, das dem Softwareentwickler bei der Identifizierung der verwendeten Design Patterns hilft, ist demnach wünschenswert.

1.2. Begriffsklärung

Um eine gemeinsame Basis zur Verständigung zu schaffen, sollten zunächst einige Begriffe erläutert beziehungsweise definiert werden.

Muster

Der Begriff Muster wird in dieser Arbeit als Oberbegriff für Beschreibungen von Problemen und deren Lösungsansätzen verwendet. Eine relativ präzise Definition für den Begriff Muster stammt von Christopher Alexander [AIS+77]:

> „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“

Diese Aussage wurde im Zusammenhang von Häuser- und Städtebau getroffen; trotzdem kann sie auf das Design von Softwaresystemen übertragen werden. Auch hier geht es um komplexe Systeme, deren Entwicklung durch abstrakte Methoden unterstützt wird.

Design Pattern

Als spezielle Muster werden die sogenannten Design Patterns von Gamma, Helm, Johnson und Vlissides [GHJV95] gehandhabt. In ihrem Buch werden 23 Design Patterns vorgestellt, die mittlerweile einen hohen Bekanntheitsgrad erreicht haben. Daher wird im Folgenden der Name Design Pattern als feststehender Begriff für Muster von Gamma et al. verwendet. In der Literatur werden sie auch häufig als GoF-Pattern (Gang of Four-Pattern) bezeichnet.

Cliché

Design Patterns enthalten immer wieder gleiche Teile, die ebenfalls als Muster beschrieben werden können. Diese Teil- oder Hilfsmuster werden hier Clichés genannt. Ein Beispiel für ein solches Cliché ist die *Delegation*. Bei der *Delegation* wird die Ausführung einer Operation von einem Objekt auf ein anderes übertragen. Im Design Pattern *Strategy* kann ein Objekt zum Beispiel eine Operation von unterschiedlichen Strategien ausführen lassen. Indem also wiederkehrende Teilmuster von Design Patterns als Clichés definiert werden, erhält man einen Katalog, der wiederum zur Definition von Design Patterns benutzt werden kann.

Musterausprägung

Wird ein Muster zur Lösung eines Problems angewandt, so muß es in Programmcode umgesetzt werden. Diese Verwendung eines Musters wird als Musterausprägung bezeichnet. Die Definition eines Musters gibt üblicherweise keine Implementierung vor. In ihrem Buch zeigen Gamma et al. beispielsweise Vorschläge zur Implementierung in C++. Daraus ergeben sich abhängig vom Kontext der Benutzung und vom Softwareentwickler beliebig viele Ausprägungen eines Musters. Wird also im Folgenden allgemein von einer Musterausprägung gesprochen, so ist damit keine konkrete Implementierung gemeint.

Mustererkennung

Bei der Mustererkennung werden Musterausprägungen gesucht. Da jedes Muster für ein bestimmtes Problem steht, kann davon ausgegangen werden, daß dieses Problem beim Design eines Programms vorlag. Ein Softwareentwickler, der mit einer automatischen Mustererkennung arbeitet, kann somit das erkannte Muster nutzen,

um Rückschlüsse auf das Design des Programms zu ziehen. Voraussetzung dafür ist natürlich, daß ihm das jeweilige Muster vertraut ist.

1.3. Probleme und Anforderungen

Gamma et al. haben die Design Patterns ursprünglich zum Forward Engineering zusammengestellt. Sie definieren Design Patterns durch vier wesentliche Punkte: Name des Design Patterns, Problembeschreibung, Lösungsansatz und Konsequenzen. Der Name ermöglicht es Entwicklern, ein gemeinsames Vokabular für Muster aufzubauen. Er hilft, sich auf einer höheren Abstraktionsebene mit anderen zu verständigen. Die Problembeschreibung gibt sowohl die Intention, die der Autor verfolgt, als auch den Kontext an, in dem das Design Pattern zu verwenden ist. Der Lösungsansatz beschreibt die Elemente des Musters und ihre Beziehungen untereinander. Es können auch Beispielimplementierungen angegeben werden. Als letztes werden die Konsequenzen, die durch die Verwendung des Design Patterns entstehen, aufgezeigt. Dabei handelt es sich beispielsweise um Laufzeitverhalten oder Folgen, die die Wiederverwendbarkeit der Software betreffen.

Diese Beschreibungen haben einen informellen Charakter und bieten deshalb dem Softwareentwickler viele Freiheitsgrade beim Entwurf und bei der Implementierung. In der Entwurfsphase kann er zum Beispiel wählen, ob Objektbeziehungen durch beidseitig traversierbare Assoziationen oder nur durch einseitige Referenzen realisiert werden. Während der Implementierung kann die gleiche Semantik durch unterschiedliche Syntax ausgedrückt werden. Beispielsweise ist es möglich, für eine Schleife ein `for`- oder ein `while`-Konstrukt zu benutzen.

Im Reverse Engineering führt dies allerdings zu einer großen Vielfalt von Ausprägungen eines einzigen Musters. Theoretisch ist es möglich, für jede Ausprägung ein Muster zu spezifizieren. Allerdings soll eine werkzeugunterstützte Mustererkennung realisiert werden. Um also ein Muster für solch eine Mustererkennung zu spezifizieren, bedarf es einer formalen Methode, die mit möglichst wenigen Spezifikationen für ein Muster diese Vielfalt abdeckt.

Die Präzision einer Mustererkennung spielt für den Benutzer eine große Rolle. Falsch erkannte Musterausprägungen sind zu vermeiden, um dem Benutzer unnötige Arbeit bei der Identifikation korrekt erkannter Musterausprägungen zu ersparen. Vollständig zu verhindern ist eine falsche Erkennung von Musterausprägungen wegen der oben angesprochenen Vielfalt der Musterausprägungen aber nicht.

Die Mustererkennung soll zum Reverse Engineering unterschiedlicher Softwaresysteme einsetzbar sein. Dies erfordert eine hohe Anpassungsfähigkeit sowohl der Musterspezifikationen als auch des Erkennungsprozesses. Die Spezifikationen sollten leicht zu ändern, hinzuzufügen und zu entfernen sein. Hilfreich in diesem Zusammenhang ist ein modularer Aufbau von Musterspezifikationen.

Die bisher erwähnten Probleme und Anforderungen bezogen sich auf die Spezifi-

kation von Mustern, also den statischen Teil der Mustererkennung. Hinzu kommen Anforderungen, die an den dynamischen Teil, den Inferenzmechanismus, gestellt werden. Dazu zählt zunächst die Skalierbarkeit des Algorithmus. Softwaresysteme bestehen in der Realität häufig aus mehreren 100.000 bis 1.000.000 Zeilen Quelltext. Trotzdem muß die Mustererkennung auch bei solchen Systemen handhabbar bleiben.

Wie bereits erwähnt ist eine absolute Präzision bei der Definition von Mustern nicht möglich, wenn eine große Vielfalt von Ausprägungen durch einige wenige Spezifikationen abgedeckt werden soll. Die Güte der Analyseergebnisse sollte deshalb vom Erkennungsprozeß bewertet werden. Der Benutzer kann sich dann bei der Auswertung der Analyseergebnisse auf diese Bewertung stützen.

Um die Präzision der Mustererkennung zu erhöhen, ist es sinnvoll, das Wissen und die Erfahrung des Softwareentwicklers in den Inferenzmechanismus zu integrieren. Die Ergebnisse einer automatischen Mustererkennung können aufgrund der oben erwähnten praktischen Beschränkungen nicht exakt sein. Hier ist eine Interaktion mit dem Benutzer sinnvoll. Die Bewertung der Güte von Analyseergebnissen bietet dem Benutzer eine Grundlage zur Beurteilung. Bei strittigen Ergebnissen sollte er in die Bewertung eingreifen können. Dadurch wird eine Revidierung der Ergebnisse ausgelöst, die zu einer exakteren Mustererkennung führt.

1.4. Aufbau und Struktur der Arbeit

Ein Teil der oben genannten Anforderungen wie formale Beschreibung, Modularität und Anpassungsfähigkeit sind bereits in der Musterspezifikationssprache aus der Diplomarbeit von Marcus Palasdies [Pal01] umgesetzt worden. Im Laufe dieser Diplomarbeit sind jedoch noch einige Erweiterungen hinzugefügt worden, um auch die restlichen Anforderungen zu erfüllen.

Zur Realisierung der dynamischen Anteile der Mustererkennung werden in dieser Diplomarbeit die *Generic Fuzzy Reasoning Nets* (GFRNs) verwendet, die von Jens Jahnke in [Jah99] vorgestellt worden sind. In GFRNs ist die Handhabung von Unschärfe bereits enthalten. Außerdem bieten sie eine Auswertung der Ergebnisse durch *Fuzzy Petrinetze* (FPNs).

Diese beiden Arbeiten werden in Kapitel 2 vorgestellt. Zunächst werden darin aber noch weitere verwandte Arbeiten beschrieben sowie ihre Probleme diskutiert. Am Ende des zweiten Kapitels folgt dann eine kurze Abhandlung über den für diese Arbeit gewählten Lösungsansatz.

Die formale Beschreibung der Muster durch die von Marcus Palasdies eingeführte Musterspezifikationssprache ist Thema des Kapitels 3. Desweiteren werden in Kapitel 3 Erweiterungen der Spezifikationssprache eingeführt. Die Spezifikation eines Musters wird anhand eines Beispiels erläutert.

Im Kapitel 4 wird zunächst die bisherige Realisierung der Mustererkennung in

dem Werkzeug FUJABA beschrieben, in dem auch der Lösungsansatz dieser Arbeit verwirklicht werden soll. Danach wird die Verwendung und die Anpassung der GFRNs erklärt. Der neu entwickelte Algorithmus zur Mustererkennung wird in Kapitel 4 formal durch Pseudocode und informell durch Bearbeitung eines Beispiels vorgestellt. Zum Schluß des Kapitels wird die Auswertung der Analyseergebnisse durch ein Fuzzy Petrinetz behandelt.

Das Thema des Kapitels 5 ist die Interaktion mit dem Benutzer bei der Mustererkennung und bei der Auswertung der Analyseergebnisse. Es werden die Möglichkeiten des Benutzers aufgezeigt, in den Inferenzmechanismus einzugreifen.

Im Kapitel 6 wird die technische Realisierung der in den Kapiteln 3 bis 5 vorgestellten Konzepte behandelt. Dafür werden zunächst die Architektur der Annotationsmaschinen und ihre automatisch generierten Methoden beschrieben. Danach wird das Design zur Umsetzung des GFRNs und des FPNs erläutert.

An einem Beispiel wird in Kapitel 7 die Benutzung der Mustererkennung in der integrierten Entwicklungsumgebung FUJABA demonstriert. Dazu gehört zunächst die Erläuterung der Musterspezifikation und die Erzeugung einer Annotationsmaschine. Weiterhin wird der Ablauf und die Auswertung der Mustererkennung, sowie die Interaktion mit dem Benutzer gezeigt.

Kapitel 8 faßt die hiermit vorgestellte Arbeit abschließend zusammen und bietet einen Ausblick auf mögliche Erweiterungen und weiterführende Anwendungsmöglichkeiten der Mustererkennung.

Im Anhang A ist ein kompletter Katalog aller für die Evaluierung dieser Arbeit erstellten Musterspezifikationen für Clichés und Design Patterns zu finden.

2. Existierende Ansätze

Dieses Kapitel stellt einige Ansätze zur Erkennung von Clichés und Design Patterns vor. Im Abschnitt 2.1 werden verschiedene Lösungen und Lösungsansätze aus ähnlichen Arbeiten sowie ihre Vor- und Nachteile diskutiert. In den Abschnitten 2.2 und 2.3 werden dann Techniken zur Spezifikation und Erkennung von Design Patterns und zum Reverse Engineering von Datenbanken behandelt, die am Lehrstuhl für Softwaretechnik an der Universität Paderborn entwickelt worden sind und dieser Arbeit als Grundlagen dienen.

2.1. Verwandte Arbeiten

Von Linda M. Wills ist Graspr (GRAphbased System for Program Recognition), ein System zur Cliché-Erkennung, vorgestellt worden [Wil96]. Die Definition für Clichés nach Wills weicht etwas von der hier verwendeten Definition ab. Wills konzentriert sich bei ihrem System stärker auf die prozedural-, nicht auf objekt-orientierte Programmierung. Beispiele für Wills Clichés sind häufig benutzte Algorithmen wie Binäre Suche und das Durchlaufen von Listen oder auch Datenstrukturen wie Hashtabellen. Die zu untersuchenden Programme werden in einem Flußgraphen repräsentiert, der Elemente aus Daten- und Kontrollflußgraphen enthält. In einer Bibliothek sind Clichés als attributierte Graphersetzungsregeln gesammelt. Die Repräsentation des Quelltextes in einem Flußgraphen ermöglicht eine Unempfindlichkeit gegenüber Variationen verschiedenster Art. Unterschiede in der Syntax oder in der Modularisierung des Codes stellen keine Probleme bei der Erkennung dar.

Die Suche nach Clichés in Quelltexten wird in Graspr auf eine Teilgraphensuche im Flußgraphen abgebildet. Das Problem der Teilgraphensuche ist aber nach Mehlhorn [Meh84] NP-vollständig. Getestet wurde Graspr nur mit sehr kleinen Programmen von 500-1000 Zeilen Länge. In der Praxis erweist es sich allerdings wegen des exponentiellen Aufwands als nicht tauglich, da heutige Softwaresysteme aus mehreren 100.000 bis 1.000.000 Zeilen Quelltext bestehen.

Christian Krämer und Lutz Prechelt haben an der Universität Karlsruhe das System Pat entwickelt [KP96]. Es basiert auf der Analyse von C++-Header Dateien. Die aus dieser Analyse hervorgehenden Informationen werden in Prolog Fakten

gesichert. Zuvor sind die Design Patterns als Prolog Regeln definiert worden. Durch Unifikation eines Prolog Interpreters werden nun Belegungen für die Regeln aufgrund der Fakten gesucht. Diese Belegungen repräsentieren schließlich Musterausprägungen von Design Patterns in den untersuchten Quelltexten.

Da dieser Ansatz aber von C++-Header Dateien ausgeht, kann er nur die strukturellen Anteile von Klassen untersuchen. Nicht-strukturelle Informationen, die in Methodenrümpfen enthalten sind, wie zum Beispiel Methodenaufrufe, fehlen. Deshalb scheitert dieser Ansatz an vielen Design Patterns wie *State* oder *Chain of Responsibility*, die nur aufgrund einer Analyse der dynamischen Anteile, also Methoden, korrekt identifiziert werden können. Außerdem liefert dieser Ansatz viele falsch erkannte Ausprägungen von Mustern, die sich zu einem großen Teil durch dynamische Vorgänge definieren.

Bei der Design Pattern-Erkennung von Jochen Seemann und Jürgen Wolff von Gudenberg [SG98] wird zunächst ein Graph aus den Quelltexten erzeugt. Der Graph besteht aus den Knotentypen Klasse, Schnittstelle und Methode, verbunden durch Kanten, die Besitzt-, Aufruf-, Referenz- oder Vererbungsbeziehungen repräsentieren. Anschließend wird der Graph gefiltert, um nutzlose Informationen zu entfernen und wichtige Informationen herauszustellen. Filterregeln beschreiben diese Veränderungen. Es werden einerseits Kanten, die verschiedene Relationen symbolisieren, zu neuen Kanten zusammengefasst. So entsteht beispielsweise aus einer Aufruf- und einer Referenzbeziehung eine Assoziationsbeziehung. Weitere abgeleitete Relationen sind Aggregation und Delegation. Andererseits können Superknoten entstehen, die mehrere Knoten zu einer Einheit verbinden. Dazu werden Regeln definiert, die Klassen und bestimmte Relationen zwischen den Klassen zu diesen Superknoten verschmelzen. Mit Hilfe solcher Regeln werden Design Patterns spezifiziert.

Leider ist noch keine Werkzeugunterstützung für diesen Ansatz vorgestellt worden. Es ist also bisher keine praktische Umsetzung realisiert worden. Desweiteren ist wegen des deduktiven Algorithmus von Problemen bei der Skalierbarkeit auszugehen, die die Praxistauglichkeit eines solchen Werkzeugs einschränken werden. Es ist zu erwarten, daß wegen der großen Datenbasis ähnliche Probleme wie bei dem Ansatz von Linda Wills auftreten werden.

Paolo Tonella und Giulio Antoniol stellen in [TA99] eine Design Pattern-Erkennung vor, die auf einer sogenannten Konzeptanalyse basiert. Bei der Konzeptanalyse werden Mengen von Klassen nach gemeinsamen Eigenschaften gruppiert. Die Inferenz beginnt bei Relationen zwischen zwei Klassen wie Assoziation oder Vererbung. Diese beiden Klassen werden dann in einer Klassensequenz zusammengefasst. Durch induktive Kontexterweiterung werden Klassen zu den Sequenzen hinzugefügt, die wiederum zu mindestens einer Klasse der Sequenz in Relation stehen. Methodenaufrufe und Besitzt-Beziehungen zwischen Klassen und Methoden werden ebenfalls als Relationen definiert. Dadurch ergeben sich schließlich Sequenzen von Klassen, die innerhalb eines Teilgraphen in Verbindung stehen. Mengen von Klassen gleicher Eigenschaften werden nochmals in Äquivalenzklassen zusammengefügt.

Repräsentanten der Äquivalenzklassen stellen nach Tonella und Antoniol Musterausprägungen dar.

Bei diesem Ansatz werden jedoch vor der Inferenz keine Muster spezifiziert, nach denen dann gesucht wird. Es werden lediglich möglichst alle Kombinationen von zusammenhängenden Teilgraphen gleicher Eigenschaften gesucht. Ein solcher Teilgraph wird dann als Muster bezeichnet. Welche dieser Muster als Design Patterns nach Gamma et al. zu interpretieren sind, wird leider nicht erläutert. Wie schon bei Krämer und Prechelt fehlt auch hier die Analyse von Methodenrümpfen, die die Erkennung von dynamischen Anteilen ermöglicht.

2.2. Design Pattern-Spezifikation und Erkennung in Fujaba

FUJABA (FROM UML TO JAVA AND BACK AGAIN) wird seit 1997 am Lehrstuhl für Softwaretechnik der Universität Paderborn entwickelt [FNT98]. FUJABA ist eine integrierte Entwicklungsumgebung, die das Design von Softwaresystemen in der Unified Modeling Language (UML) [UML] und dem Story Driven Modeling (SDM) [FNTZ98], sowie deren automatische Übersetzung in die Programmiersprache Java unterstützt. Ebenfalls möglich ist das Reverse Engineering, bei dem Java-Quelltexte eingelesen und als UML-Diagramme dargestellt werden.

Beim Reverse Engineering von Java-Programmen werden die Quelltexte geprasst und in einem abstrakten Syntaxgraphen (ASG) repräsentiert, der durch ein FUJABA-eigenes Metamodell beschrieben ist. Das Metamodell ist unabhängig von der Programmiersprache, so daß prinzipiell durch Austausch des Parsers jede beliebige objektorientierte Programmiersprache unterstützt werden kann. Im ASG sind sowohl strukturelle Informationen über Klassen, Beziehungen zwischen Klassen, Methoden und Attributen von Klassen enthalten, als auch abstrakte Syntaxbäume der Methodenrümpfe, aus denen nicht strukturelle Informationen wie Methodenaufrufe gewonnen werden können.

Auf dieser Basis wurde erstmals durch die Projektgruppe FUJABA eine Erkennung für einige wenige Design Patterns und Clichés realisiert. Allerdings mußte für jedes neue Muster per Hand eine neue Annotationsmaschine implementiert werden. Dies war sowohl bei der Erstellung als auch bei der Wartung sehr aufwendig. Ebenso war es fast nicht möglich, die Annotationsmaschinen an bestimmte Eigenheiten von zu untersuchenden Softwaresystemen anzupassen. Es war also eine starre und sehr unflexible Lösung, die genaue Kenntnisse über die Steuerung der Analyse und die Technik einer Annotationsmaschine erforderte.

Um diese Unzulänglichkeiten zu beseitigen, ist in der Diplomarbeit von Marcus Palasdies eine graphische Spezifikationssprache für Muster entwickelt worden [Pal01]. Sie ist beispielhaft in die FUJABA-Entwicklungsumgebung integriert wor-

den. Mit Hilfe dieser Spezifikationssprache ist es nun relativ einfach möglich, neue Muster zu definieren und automatisch eine entsprechende Annotationsmaschine generieren zu lassen, die sofort in FUJABA eingebunden und ausgeführt werden kann.

Einige grundlegende Prinzipien sind jedoch beibehalten worden. Um im ASG gefundene Musterausprägungen zu kennzeichnen, wird der ASG mit sogenannten Annotationen angereichert. Diese Art von Anreicherung oder Annotation von Graphen kann auf das von Rekers und Schürr in [RS95] beschriebene Verfahren zum graphischen Parsen zurückgeführt werden. Diese Annotationen enthalten Informationen über die Art des erkannten Musters und Referenzen zu beteiligten Objekten im abstrakten Syntaxbaum. Im UML-Klassendiagramm können die gefundenen Annotationen angesehen werden. Sie tragen den Namen des Musters, das sie repräsentieren.

Des weiteren besteht der Erkennungsmechanismus aus einzelnen Maschinen, die jeweils für ein spezifisches Muster zuständig sind. Die Inferenz wird von einer zentralen Maschine gesteuert, die einzelne Objekte aus dem Syntaxbaum an die zuständigen Annotationsmaschinen weiterreicht. Diese untersuchen das an sie übergebene Objekt inklusive seines Kontextes. Wird eine Musterausprägung erkannt, so erstellt die Maschine eine neue Annotation mit den entsprechenden Informationen.

Da gewisse Teilstrukturen wie Assoziationen oder Delegationen bei Mustern immer wieder vorkommen, ist eine Komposition von Mustern vorgesehen. Muster setzen sich aus einem oder mehreren Clichés zusammen und enthalten zusätzliche Informationen über deren Kontext und ihren Beziehungen untereinander. Dadurch ist der Aufbau einer Bibliothek von Hilfsmustern möglich, die die Definition neuer Muster erheblich vereinfacht. So entsteht eine Hierarchie aufeinander aufbauender Muster.

Zusätzlich zur Komposition steht die Vererbung von Mustern zur Verfügung. Annotationen des erbenden Musters annotieren die gleichen Klassen aus dem Metamodell des ASG wie Annotationen des Elternmusters. Das Muster selbst wird dabei nicht vererbt. Das bedeutet, daß Annotationen polymorph verwendet werden können. In Musterausprägungen, deren Spezifikationen die Verwendung einer Annotation eines Elternmusters vorsehen, können Annotationen eines Kindmusters eingesetzt werden. Es entsteht so ein mächtiges Mittel, um eine große Menge von Variationen in der Musterausprägung des zu spezifizierenden Musters abzudecken.

2.3. Reverse Engineering auf Basis von Generic Fuzzy Reasoning Nets

In [Jah99] stellt Jens Jahnke einen Ansatz zum Reverse Engineering von relationalen Datenbanken vor. Ziel dieses Ansatzes ist es, das Schema einer Datenbank aus Tabellen und Quelltexten zu extrahieren und für den Benutzer in Form von

ER-Diagrammen oder objektorientierten Diagrammen aufzubereiten.

Die Spezifikation des Wissens, das für das Reverse Engineering relationaler Datenbanken notwendig ist, erfolgt in sogenannten Generic Fuzzy Reasoning Nets (GFRNs). Zur Handhabung von unsicherem Wissen, wie zum Beispiel bei der Verwendung von Heuristiken, unterstützen sie Unschärfe durch possibilistische Logik und Fuzzy Mengentheorie. Zusätzlich bieten sie eine hohe Flexibilität, da sie leicht anzupassen und zu erweitern sind. Durch die Möglichkeit zur Interaktion mit dem Benutzer lassen sich Analyseergebnisse beeinflussen.

Ein GFRN ist ein gerichteter Graph, dessen Knoten aus Prädikaten und Implikationen bestehen. Kanten existieren nur zwischen Prädikaten und Implikationen und umgekehrt. Implikationen können Bedingungen enthalten. Außerdem legen sie sogenannte Vertrauens- und Schwellwerte fest, die für die spätere Fuzzy-Bewertung notwendig sind. Die Prädikate werden nochmals unterteilt in starke, schwache und aufgeschobene Axiome.

Bei der Analyse einer relationalen Datenbank werden zunächst eine Reihe von Basisfakten generiert. Diese erfüllen starke Axiome - Prädikate, die Quellen im GFRN sind, also keine einlaufende Kante aus einer Implikation besitzen. Durch die Inferenz wird dann versucht, über die Implikationen auf schwache Axiome zu schließen. Fakten, die Prädikate erfüllen, werden durch Stellen in einem Fuzzy Petrinetz (FPN) repräsentiert. Das Fuzzy Petrinetz, welches hier benutzt wird, stellt eine Erweiterung des Modells der Fuzzy Petrinetze von Konar und Mandal [KM96] dar. Die Transitionen im FPN übernehmen die Rolle der Implikationen im GFRN. Sie modellieren die Abhängigkeiten der Fakten untereinander.

Zum Ende der Inferenz wird dann das FPN ausgewertet und die sogenannten Fuzzy Beliefs der Stellen berechnet, die die Glaubwürdigkeit eines Faktums in Prozentwerten ausdrücken. Hier kann der Benutzer eingreifen und die Fuzzy Beliefs verändern. Ist er sich beispielsweise sicher, daß ein bestimmtes Faktum ein Axiom erfüllt, so kann er den zugehörigen Fuzzy Belief auf 100% setzen. Daraus folgt eine Neuberechnung des gesamten FPN, da sich die Werte abgeleiteter Fakten ändern können. Ebenso kann der Benutzer den Fuzzy Belief eines Faktums auf 0% festlegen, um auszudrücken, daß dieses Faktum entgegen der Analyse aus dem GFRN das entsprechende Axiom nicht erfüllt.

GFRNs stellen also eine Möglichkeit dar, um Wissen in einer flexiblen und durch den Menschen leicht lesbaren Form abzubilden. Des weiteren bieten sie die Handhabung von Unschärfe und Interaktion mit dem Benutzer, was bei Analysen, die von Natur aus keine sicheren Ergebnisse liefern können, wünschenswert ist.

2.4. Lösungsansatz

Die Musterspezifikationssprache von Marcus Palasdies bietet bereits die formale Spezifikation von Mustern. Einige der Anforderungen aus Abschnitt 1.3 sind dadurch

erfüllt. So ist durch die Komposition von Mustern Modularität erreicht worden. Durch die graphische Notation ist es sehr einfach, die bereits spezifizierten Muster an neue Anforderungen anzupassen. Weitere Sprachelemente sind der Musterspezifikationssprache hinzuzufügen, um die Variantenvielfalt von Musterausprägungen besser abdecken zu können.

Arbeiten, wie die von Linda Wills zur Clichéerkennung, haben gezeigt, daß reine deduktive Ansätze zu Problemen bei der Skalierbarkeit führen. Deduktive Ansätze, die auch als Bottom Up bezeichnet werden, gehen von Basisfakten aus und versuchen schichtenweise hierarchisch höherliegende Analysen durchzuführen. Dazu werden allerdings zunächst alle Analysen einer Schicht ausgeführt, bevor die nächste Schicht abgearbeitet wird. Im Gegensatz dazu stehen die Top Down-Analysen. Die Top Down-Analyse bedeutet theoretisch, den gesamten Suchraum abzuarbeiten, was praktisch nicht durchführbar ist.

Deshalb ist eine kombinierte Bottom Up/Top Down-Analyse wie in den GFRNs von Jens Jahnke wünschenswert. Top Down-Analysen sind in der bisherigen Form der GFRNs aber nur auf Basisfakten möglich. Diese Einschränkung ist aufzuheben. Desweiteren soll der Inferenzmechanismus der GFRNs von einer optimierten Suchreihenfolge ausgehen, die in dem GFRN enthalten ist. Gezielte Suchpfade zu Prädikaten sollen es ermöglichen, schnell zu gewünschten Analyseergebnissen zu gelangen, ohne erst den gesamten Suchraum abarbeiten zu müssen.

Die Handhabung von Unschärfe beziehungsweise unsicheren Wissens ist in den GFRNs bereits enthalten. Die Fuzzy Petrinetze stellen eine Grundlage dar, um Analyseergebnisse auf Basis von Unschärfe zu bewerten. Jedoch stellt die Musterspezifikationssprache noch keine Sprachelemente zur Verfügung, um Bewertung durch Unschärfe zu unterstützen. Die Sprache ist also in diese Richtung zu erweitern.

Für die Mustererkennung wird der Inferenzmechanismus der GFRNs adaptiert. Zum einen werden Änderungen an der Syntax vorgenommen, um GFRNs mit der Musterspezifikationssprache zu verbinden. Zum anderen sind Änderungen an der Semantik notwendig, um eine gute Skalierbarkeit des Erkennungsprozesses zu erreichen.

3. Cliché- und Design Pattern-Spezifikation

Die folgenden Abschnitte stellen Techniken vor, die in Abschnitt 1.3 diskutierten Probleme und Anforderungen anzugehen. Abschnitt 3.1 behandelt die Repräsentation des Quelltextes in einem Graphen. Eine ausführliche Beschreibung der Musterspezifikationssprache von Marcus Palasdies, die bereits in 2.2 angesprochen wurde, folgt in Abschnitt 3.2. Diese Sprache ist erweitert worden, um zusätzliche Anforderungen zu erfüllen. Die Erweiterungen sind in 3.3 zu finden. Der Abschnitt 3.4 bildet dann schließlich den Abschluß des Kapitels mit einem Beispiel zur Spezifikation eines Design Patterns.

3.1. Der abstrakte Syntaxgraph

In der integrierten Entwicklungsumgebung Fujaba wird der Quelltext eines Softwaresystems zunächst geparst und durch einen abstrakten Syntaxgraphen (ASG) repräsentiert. Der abstrakte Syntaxgraph basiert auf einem Metamodell, das an das UML-Metamodell angelehnt ist. Aus dem ASG kann vor allem die Struktur des Softwaresystems abgelesen werden. Es sind Informationen über Methoden und Attribute von Klassen enthalten. Benutzt- und Vererbungsbeziehungen zwischen Klassen sind ebenso zu finden.

Um jedoch dynamische Beziehungen zwischen Klassen erkennen zu können, müssen die Methodenrümpfe analysiert werden. Diese sind ebenfalls in abstrakten Syntaxgraphen repräsentiert. In dem Syntaxgraphen einer Methode lassen sich unter anderem Aufrufe von Methoden anderer Klassen finden, die ein wichtiges Indiz für die Art der Interaktion zwischen Klassen beziehungsweise Objekten dieser Klassen sind.

Des weiteren bieten die abstrakten Syntaxbäume eine teilweise Normierung der Quelltexte bezüglich der Syntax. Die konkrete Ausprägung einer `for`-, `while`- oder `do while`- Schleife beispielsweise wird zur abstrakten Information einer Schleife reduziert. Ein Teil der Implementierungsvarianten kann somit zusammengeführt und gleich behandelt werden.

3.2. Die Musterspezifikationssprache

Eine graphische Spezifikationssprache für Muster ist bereits von Marcus Palasdies in seiner Diplomarbeit [Pal01] vorgestellt worden. Mit Hilfe dieser Sprache werden Beispielausprägungen eines Musters spezifiziert. Eine Beispielausprägung stellt jedoch keine konkrete Musterausprägung dar. Sie gilt als Repräsentant einer Äquivalenzklasse bestehend aus Musterausprägungen mit den in der Beispielausprägung definierten Eigenschaften.

Die Syntax der Spezifikationssprache orientiert sich an der Syntax von UML-Objektdiagrammen. Wie bei den UML-Objektdiagrammen mit ihren zugehörigen UML-Klassendiagrammen gibt es auch in der Spezifikationssprache für Muster eine Unterscheidung in Objekt- und Klassendiagramm. Der Benutzer arbeitet hauptsächlich mit dem Objektdiagramm. Hierin definiert er die Beispielausprägung eines Musters. Sie besteht aus einem Graphen. Die Knoten des Graphen sind ASG-Objekte, Musterobjekte oder Operatoren, die Kanten werden Verknüpfungen genannt.

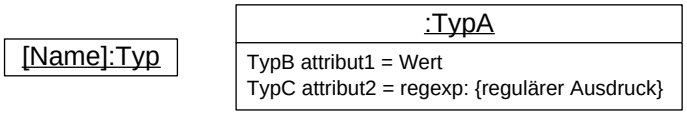

Abbildung 3.1.: ASG-Objekte

ASG-Objekte repräsentieren in der Spezifikationssprache Objekte aus dem abstrakten Syntaxgraphen eines geparsten Softwaresystems. Sie werden durch Rechtecke dargestellt. Der Typ des ASG-Objektes wird in der oberen Hälfte des Rechtecks hinter einem Doppelpunkt angegeben. Vor dem Doppelpunkt kann optional der Name des Objektes genannt werden. Das linke ASG-Objekt in Abbildung 3.1 zeigt die Syntax.

Boolsche Bedingungen können ebenfalls optional zu den ASG-Objekten formuliert werden. Diese sind in der unteren Hälfte des Rechtecks, durch einen Querstrich vom Namen und Typ getrennt, zu finden. Die Bedingungen beinhalten Vergleiche mit konkreten Werten, mit Werten von Attributen anderer ASG-Objekte oder auch mit regulären Ausdrücken für Zeichenketten. Das rechte ASG-Objekt in Abbildung 3.1 ist ein Beispiel mit zwei Bedingungen, die zweite Bedingung ist ein Vergleich mit einem regulärem Ausdruck.

Ein weiterer wichtiger Knotentyp sind die Musterobjekte. Musterobjekte modellieren die Wiederverwendung von schon vorhandenen Musterspezifikationen. Wird ein Musterobjekt in einer Musterspezifikation verwendet, so kann man sich an dieser Stelle die zum Musterobjekt gehörende Musterspezifikation eingesetzt denken. Ein Musterobjekt erspart also zum einen die Spezifikation von immer wiederkehrenden Teilmustern und zum anderen verbessert es die Lesbarkeit der Spezifikation.

Abbildung 3.2.: Musterobjekte

Musterobjekte sind Repräsentanten für Annotationen des entsprechenden Musters im ASG. Im Graphen werden sie als Ovale mit dem Musternamen im Inneren dargestellt. Abbildung 3.2 zeigt zwei Musterobjekte.

Das rechte Musterobjekt aus Abbildung 3.2 spielt in der Spezifikation eines Musters eine besondere Rolle. Es ist mit dem Stereotyp «*create*» ausgezeichnet. Pro Spezifikation gibt es nur ein solches Element. Es trägt den Namen des zu spezifizierenden Musters und steht für die Annotation, die im Falle einer gefundenen Musterausprägung erzeugt werden soll.

Der dritte Knotentyp sind Operatoren. Es gibt insgesamt acht verschiedene Operatoren: *equal*, *non-equal*, *equal-set*, *non-equal-set*, *left-subset*, *right-subset*, *left-subset-equal* und *right-subset-equal*. Operatoren sind Platzhalter für ASG-Objekte. Sie repräsentieren entweder einzelne Objekte oder im Falle der *set*-Operatoren Mengen von Objekten. Eine genaue Definition aller Operatoren findet sich in [Pal01].

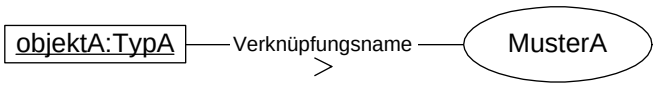

Abbildung 3.3.: Verknüpfung von Knoten

Kanten werden in der Spezifikationssprache Verknüpfungen genannt. Verknüpfungen sind ungerichtet. Sie können beliebige der drei Knotentypen miteinander verbinden. Verknüpfungen werden benannt und ihr Name mit einem Pfeil, der die Leserichtung des Namens angibt, versehen. Sie besagen, daß in einer passenden Musterausprägung eine Verbindung zwischen den verknüpften Elementen über eine bestimmte Assoziation, deren Name die Verknüpfung trägt, bestehen muß. Die Syntax ist in Abbildung 3.3 anschaulich dargestellt.

Verknüpfungen, die von dem mit dem Stereotyp «*create*» versehenen Musterobjekt ausgehen, bilden einen Sonderfall. Sie werden ebenfalls mit dem Stereotyp «*create*» gekennzeichnet, da sie im Falle einer positiven Suche die Verbindung der erzeugten Annotation zu den gefundenen Objekten herstellen und deshalb neu erzeugt werden müssen.

Sowohl Verknüpfungen als auch ASG- und Musterobjekte lassen sich negieren. Dies geschieht, indem das entsprechende Element durchkreuzt (Abbildung 3.4) wird.

Abbildung 3.4.: Negierung von ASG-Objekt, Musterobjekt und Verknüpfung

Negierte Verknüpfungen verbieten eine Verbindung in der Musterausprägung zwischen den verknüpften Elementen. Negierte ASG- oder Musterobjekte beschreiben, daß ein entsprechendes Objekt in einer passenden Musterausprägung nicht vorhanden sein darf.

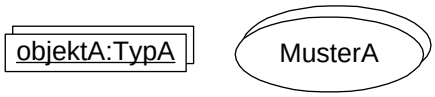

Abbildung 3.5.: Mengendefinition eines ASG- und eines Musterobjekts

ASG- und Musterobjekten können als Mengen definiert werden. Dadurch repräsentieren sie nicht nur ein Objekt in einer passenden Musterausprägung, sondern eine ganze Menge von Objekten. Abbildung 3.5 zeigt die Syntax für jeweils ein als Menge gekennzeichnetes ASG- und Musterobjekt.

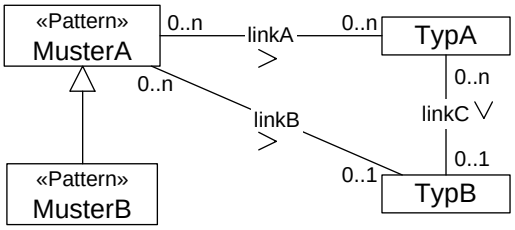

Abbildung 3.6.: Klassendiagrammausschnitt der Musterspezifikationssprache

Das Klassendiagramm einer Musterspezifikation wird zum Teil automatisch durch Fujaba generiert. Dazu gehören Klassen für alle Knoten, sowie Assoziationen für die Verknüpfungen aus dem Objektdiagramm. Musterklassen, also Klassen für Musterobjekte, sind in dem Klassendiagramm durch den Stereotyp «*Pattern*» gekennzeichnet.

Der Benutzer hat die Möglichkeit, in dem Klassendiagramm einer Musterspezifikation Vererbungsbeziehungen zwischen Musterklassen zu definieren. Nur Einfachvererbung ist erlaubt. Abbildung 3.6 zeigt einen Ausschnitt eines Klassendiagramms. Zu sehen sind die Assoziationen zwischen den ASG- und Musterobjekten sowie die

Vererbung zwischen Musterobjekten. Das erbende Muster übernimmt die Assoziationen des Elternmusters.

Musterobjekte sind in den Spezifikationen gleichzeitig Stellvertreter für alle Musterobjekte von erbenden Mustern. Dies ist gewährleistet durch eine polymorphe Bindung von Annotationen an die Musterobjekte während der Suche nach Musterausprägungen.

Die Vererbung ist somit ein mächtiges Mittel, um eine größere Vielfalt von Musterausprägungen durch eine einzige Spezifikation zu beschreiben.

3.3. Erweiterungen der Spezifikationssprache

Einige der Anforderungen aus 1.3 sind mit den bisherigen Möglichkeiten der Musterspezifikationssprache bereits erfüllt. So bietet sie eine formale Beschreibung von Mustern, sowie Modularität und Anpassungsfähigkeit.

Allerdings hat sie noch Schwächen bei der Abdeckung einer großen Vielfalt von Musterausprägungen. Insbesondere bei der Analyse von Methodenrümpfen fehlt es an einer geeigneten Notation zur Handhabung der vielen syntaktischen Unterschiede. Dieses sollen vor allem die Pfadausdrücke leisten, die in Abschnitt 3.3.1 beschrieben werden. Optionale Knoten sind ein zusätzliches Mittel, um eine größere Vielfalt von Musterausprägungen zu unterstützen. Sie werden in Abschnitt 3.3.2 behandelt.

Als Grundlage zur Handhabung von Unschärfe bei der Erkennung und Bewertung von Musterausprägungen sind die Vertrauens- und Schwellwerte (Abschnitt 3.3.3) eingeführt worden. Um zusätzliche Informationen für den Erkennungsprozeß in die Musterspezifikationen aufzunehmen, gibt es sogenannte Trigger. Eine kurze Erklärung zur Spezifikation von Triggern findet sich in Abschnitt 3.3.4, die Funktionsweise wird aber erst im Kapitel 4 über den Inferenzmechanismus erläutert.

3.3.1. Pfadausdrücke

Pfadausdrücke sind als neuer Kantentyp in die Musterspezifikationssprache eingeführt worden. Sie sind gerichtet und besitzen somit ein Start- und ein Zielknoten. Mit ihrer Hilfe lassen sich Beziehungen zwischen Objekten des ASG spezifizieren, die nicht direkt in Verbindung stehen.

Das Objekt aus der Musterausprägung, das an den Startknoten gebunden ist, ist das Startobjekt des Pfades. Alle Objekte, die vom Startobjekt über beliebige Wege, also Verknüpfungen, erreichbar und vom Typ des Zielknotens sind, sind Zielobjekte. Objekte, die auf dem Weg zwischen Start- und Zielobjekt liegen, müssen nicht näher spezifiziert werden.

Es ist möglich, Typen von Objekten anzugeben, über die ein Weg zwischen Start- und Zielobjekt nicht führen darf. Das heißt, gibt es keinen Weg vom Start- zum Zielobjekt, der Objekte des angegebenen Typs meidet, so wird der Pfad nicht erfüllt.

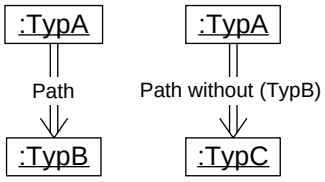

Abbildung 3.7.: Pfadausdrücke

Abbildung 3.7 zeigt die Syntax anhand zweier Pfadausdrücke. Der linke Pfadausdruck ist ohne Einschränkung, der rechte unterbindet Objekte vom Typ *TypB* auf dem Pfad zwischen *TypA* und *TypC*. Weitere zu unterdrückende Typen können in den Klammern angegeben werden.

Semantisch gleiche Quelltextfragmente können sich stark in der Syntax unterscheiden. Methoden, die die gleiche Funktion haben, können also unterschiedliche abstrakte Syntaxbäume besitzen. Oftmals genügen aber wenige Informationen als Aussagen über Methoden, zum Beispiel ein Methodenaufruf, der in einer Schleife stattfindet. Durch gestaffelte Pfadausdrücke lassen sich solche Bedingungen leicht formulieren. Eine große Vielfalt syntaktisch unterschiedlicher Methoden wird somit zusammengefasst.

3.3.2. Optionale Knoten

ASG- und Musterobjekte können als optional ausgezeichnet werden. In einer zu der Spezifikation passenden Musterausprägung muß zu einem optionalen Knoten kein Objekt gefunden werden. Optionale Knoten dürfen für die Definition eines Musters nicht notwendig sein. Sie drücken nur die Möglichkeit der Existenz aus. Ihre Existenz kann die resultierende Annotation durch zusätzliche Informationen untermauern.

Die gleiche Semantik wie durch Verwendung von optionalen Knoten kann man durch die Definition mehrerer Musterspezifikationen erreichen. Es sind alle Kombinationen aus dem nicht optionalen Teil der Spezifikation und den optionalen Knoten durch Hinzunahme der optionalen Knoten als nicht optionale Knoten beziehungsweise durch Weglassen dieser Knoten zu bilden. Alle Musterspezifikationen dieser Menge zusammengenommen decken die gleichen Ausprägungen ab, wie die Spezifikation mit optionalen Knoten.

Abbildung 3.8.: Optionale Knoten

Abbildung 3.8 zeigt ASG- und Musterobjekte, die als optional definiert worden

sind. Als optisches Merkmal werden sie grau gezeichnet.

3.3.3. Vertrauens- und Schwellwert

Eine Anforderung an die Mustererkennung waren Aussagen über die Güte der Analyseergebnisse. Musterausprägungen sollen mit einer gewissen Unschärfe bewertet werden, um absolute Aussagen zu vermeiden. Fälschlicherweise gefundene Musterausprägungen können nicht vermieden werden. Aus diesem Grund sind Behauptungen, eine Objektstruktur sei die Ausprägung eines bestimmten Musters, eher irreführend für den Benutzer. Um solche absoluten Aussagen zu umgehen, sind sogenannte Vertrauens- und Schwellwerte eingeführt worden. So muß zu jeder Musterspezifikation ein solches Wertepaar definiert werden.

Der Vertrauenswert trifft eine Aussage darüber, wie sicher der Softwaretechniker ist, daß die von ihm definierte Musterspezifikation das entsprechende Cliché oder Design Pattern beschreibt. Dabei sollte dieser Wert Erfahrungswerte des Softwareingenieurs widerspiegeln. Bei der späteren Auswertung der Ergebnisse der Cliché- und Design Pattern-Erkennung durch ein Fuzzy Petrinetz wird auf den Vertrauenswert der Musterspezifikation zurückgegriffen und als Grundlage für weitere Berechnungen verwendet.

Der Schwellwert legt das Minimum der Güte aller Musterausprägungen fest, die bei der Suche nach einer Ausprägung des spezifizierten Musters benutzt werden. Liegt die Güte einer Musterausprägung, ausgedrückt durch den Fuzzy Belief der Annotation, unterhalb des Schwellwerts, so kann sie nicht für das zu suchende Muster verwendet werden. Der Schwellwert soll also die Anzahl falsch erkannter Musterausprägungen vermindern.

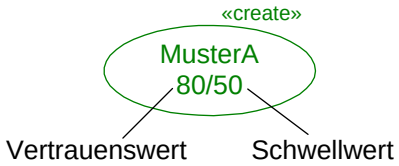

Abbildung 3.9.: Spezifikation des Vertrauens- und Schwellwerts

Das Wertepaar Vertrauens- und Schwellwert wird zu einer Musterspezifikation in dem Musterobjekt mit dem Stereotyp «*create*», das pro Spezifikation einmalig ist, abgelegt. Abbildung 3.9 zeigt ein Musterobjekt mit diesen beiden Werten unterhalb des Musternamens. Der erste Wert ist der Vertrauenswert, der zweite der Schwellwert. Beides sind prozentuale Werte zwischen 0 und 100.

3.3.4. Trigger

Der Inferenzmechanismus, der im Kapitel 4 erläutert wird, benötigt Informationen, welche Objektstrukturen auf Musterausprägungen hin zu untersuchen sind. Dazu sind sogenannte Trigger eingeführt worden. ASG- und Musterobjekte können als Trigger ausgezeichnet werden. Pro Spezifikation muß mindestens ein Trigger vorhanden sein.

Erhält der Inferenzmechanismus Objekte, die vom Typ des Triggers sind, lösen sie die Untersuchung ihres Kontextes auf das Vorhandensein einer Musterausprägung aus. In unterschiedlichen Musterspezifikationen können Trigger vom gleichen Typ vorhanden sein. Ein Objekt kann demnach die Überprüfung auf verschiedene Spezifikationen auslösen.

Abbildung 3.10.: Als Trigger ausgezeichnete ASG- und Musterobjekte

Trigger werden durch fette Umrahmungen ihrer graphischen Objekte dargestellt. In Abbildung 3.10 ist je ein ASG- und ein Musterobjekt als Trigger gekennzeichnet zu sehen. Eine tiefergehende Erläuterung über Zweck und Funktion der Trigger wird in Abschnitt 4.3 gegeben.

3.4. Beispielspezifikation eines Design Patterns

In diesem Abschnitt wird als Beispiel für das Design Pattern *Composite* nach Gamma et al. eine Musterspezifikation erstellt. In Abbildung 3.11 ist das Klassendiagramm des Design Patterns in UML-Notation zu sehen. Das *Composite*-Muster beschreibt die Organisation von Objekten in einer Baumstruktur. Sowohl Blätter als auch Knoten in der Baumstruktur können gleichartig behandelt werden. Die *Component*-Klasse ist abstrakt, sie definiert die Schnittstelle für alle Elemente des Baumes. Einen Knoten, der beliebige Elemente des Baumes aufnehmen kann, stellt die Klasse *Composite* dar. Die Klasse *Leaf* ist schließlich stellvertretend für alle Blattklassen. *Composite* und alle Blattklassen erben von *Component*, so daß sie die gleiche Schnittstelle zusichern. Eine Klasse, die eine solche Baumstruktur benutzt, besitzt üblicherweise eine Referenz auf die Wurzel des Baumes. Es genügt hierbei eine Referenz auf die Klasse *Component*, wie bei der Klasse *Client*.

Die wesentlichen Merkmale des Design Patterns *Composite* stellen die beiden Klassen *Component* und *Composite*, sowie ihre Beziehungen untereinander dar. Zunächst einmal erbt die Klasse *Composite* von der Klasse *Component*. Entlang dieser Vererbungshierarchie verläuft eine Assoziation zur abstrakten Oberklasse. Sie

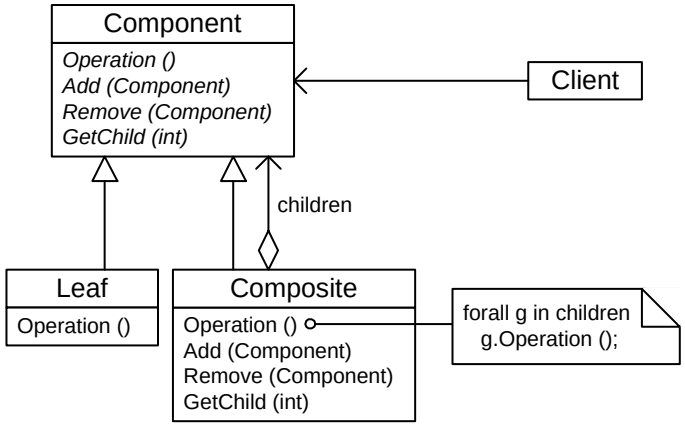

Abbildung 3.11.: Das Design Pattern *Composite* nach Gamma et al.

modelliert eine Kind-Beziehung zwischen der Knotenklasse und Baumelementen, die an diesem Knoten hängen. Desweiteren existiert eine mehrfache Delegation zwischen der Knotenklasse *Composite* und der Basisklasse *Component*. Operationen, die auf alle Elemente des Baumes angewandt werden sollen, werden auf dem Wurzelobjekt aufgerufen und durch die Knotenklasse an alle Kinder weitergereicht.

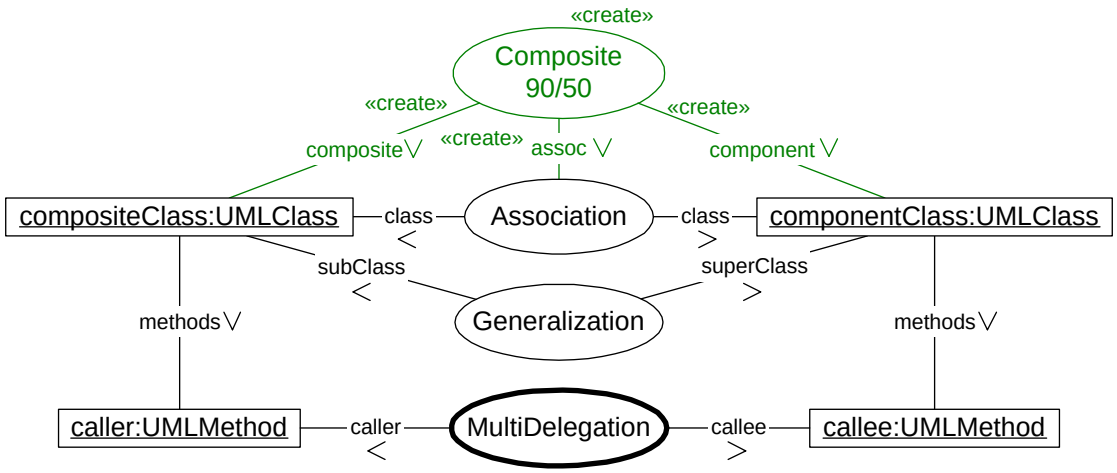

Abbildung 3.12.: Spezifikation des Design Pattern *Composite*

Abbildung 3.12 zeigt die Musterspezifikation des Design Patterns *Composite*. Seine oben aufgezählten Merkmale sind hier wiederzufinden. Es existieren zwei ASG-Objekte, die die Klassen *Component* und *Composite* repräsentieren. Zwischen ihnen gibt es die Musterobjekte *Association* und *Generalization*, die bereits als eigene Clichés spezifiziert worden sind. Die Mehrfachdelegation ist ebenfalls durch ein Mu-

sterobjekt zwischen zwei ASG-Objekten spezifiziert, die Platzhalter für die aufrufende und die aufgerufene Methode sind.

Die Spezifikation enthält kein Objekt, das die Klasse *Leaf* aus Abbildung 3.11 repräsentiert. Diese Klasse ist nicht wesentlich für die Definition eines *Composite*-Patterns. In einer Musterausprägung können mehrere solcher Blattklassen verwendet werden, aber auch ein Design aus nur einer *Component*- und einer *Composite*-Klasse stellt noch immer ein *Composite*-Design Pattern dar. Es muß also in der Musterspezifikation keine explizite Blattklasse existieren. Ebenfalls ausgelassen wurde die Klasse *Client*, die nur die Benutzung des Design Patterns verdeutlichen soll und somit nicht zum eigentlichen Design Pattern gehört.

Der schwarze Teil des Graphen beschreibt eine Situation, die von der Mustererkennung im abstrakten Syntaxgraphen des zu untersuchenden Softwaresystems wiederzufinden ist. Bei einer positiven Suche werden alle mit dem Stereotyp «*create*» ausgezeichneten Elemente des Graphen produziert. Das heißt, es wird ein Annotationsobjekt angelegt und mit den Objekten aus dem abstrakten Syntaxgraphen über Verknüpfungen verbunden. Das Annotationsobjekt enthält die Information, daß zwischen den annotierten Objekten eine Ausprägung des Design Patterns *Composite* vorliegt.

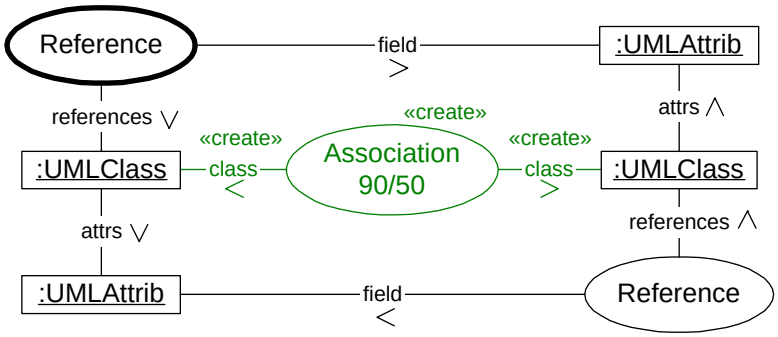

Abbildung 3.13.: Spezifikation des Clichés *Association*

Die Musterspezifikation des Clichés *Association* ist in Abbildung 3.13 zu sehen. Es beschreibt die gegenseitige Referenzierung zweier Klassen. Eine Referenzierung ist wiederum als eigenes Cliché spezifiziert worden. Demzufolge wird es in der Musterspezifikation der Assoziation zweimal verwendet.

Mehrfachdelegation Die Mehrfachdelegation ist ein gutes Beispiel, um die Verwendung von Pfadausdrücken zu erläutern. Als Delegation wird die Weiterreichung einer Aufgabe von einem delegierenden Objekt an ein empfangendes Objekt verstanden. Technisch realisiert wird dies durch den Aufruf einer Methode des empfangenden Objektes durch eine Methode des delegierenden Objektes. Üblicherweise tragen die beiden Methoden den gleichen Namen. Bei einer Mehrfachdelegation existieren mehrere empfangende Objekte, auf denen die Methode aufgerufen wird.

Diese Semantik kann jedoch durch beliebig unterschiedliche Syntax implementiert werden. Die Musterspezifikation einer Mehrfachdelegation hat also unendlich viele Ausprägungsmöglichkeiten zu handhaben. Die einzigen Syntaxelemente, die wahrscheinlich bei allen Varianten existieren, sind eine Schleife und ein Methodenaufruf innerhalb dieser Schleife.

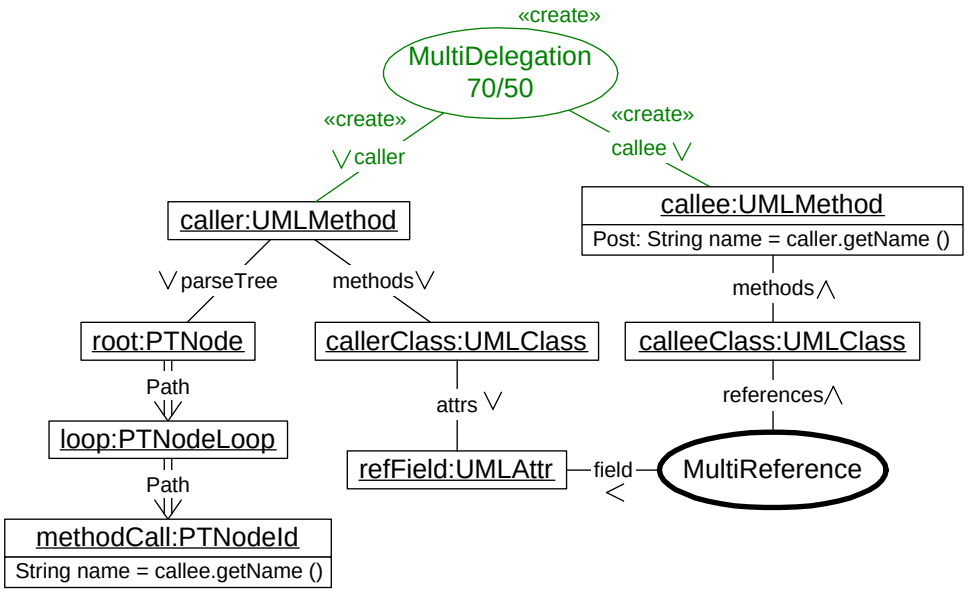

Abbildung 3.14.: Spezifikation des Clichés *MultiDelegation*

Abbildung 3.14 zeigt die Musterspezifikation dieses Clichés. Es gibt zwei Pfadausdrücke, die zusammengenommen die beschriebene Situation ergeben. Von der Wurzel der aufrufenden Methode ausgehend, dargestellt durch das ASG-Objekt *root:PTNode* muß ein Pfad zu einer Schleife existieren und von dort aus ein Pfad zu einem Objekt vom Typ *PTNodeId*, das den Methodenaufruf repräsentiert. Alle Elemente auf dem Weg zwischen Wurzel und Schleife, sowie zwischen Schleife und Methodenaufruf werden ignoriert und müssen somit nicht spezifiziert werden.

Das Cliché *Generalization* aus Abbildung 3.15 beschreibt eine Vererbung zwischen zwei Klassen im ASG. An diesem Beispiel läßt sich die Vererbung von Mustern leicht erklären. Häufig findet sich statt der erwarteten direkten Vererbung zwischen zwei Klassen im ASG eine Vererbungshierarchie über mehrere Klassen. Im Design Pattern *Composite* würde das bedeuten, daß zwischen der Klasse *Component* und der Klasse *Composite* mindestens eine weitere Klasse in der Vererbungshierarchie liegt. Trotzdem liegt noch immer eine Ausprägung des *Composite*-Design Patterns vor.

Zu lösen ist dieses Problem, indem ein weiteres Muster spezifiziert wird, das eine solche Vererbungshierarchie beschreibt. Abbildung 3.16 zeigt dieses Cliché. Erbt

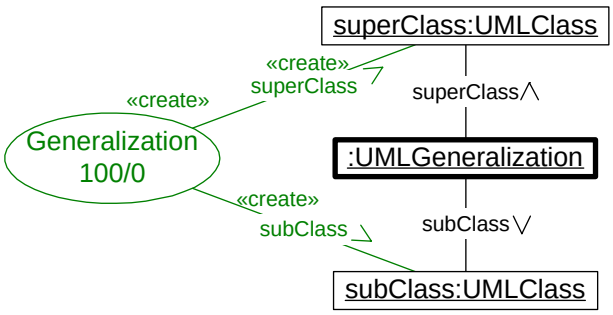

Abbildung 3.15.: Spezifikation des Clichés *Generalization*

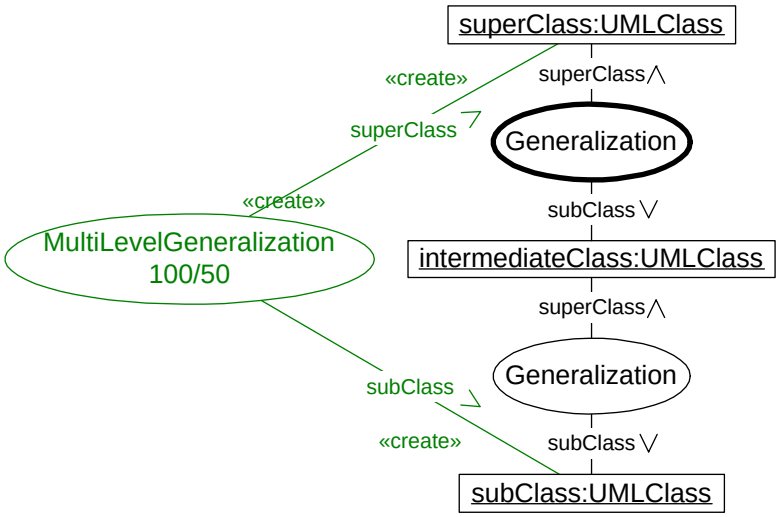

Abbildung 3.16.: Spezifikation des Clichés *MultiLevelGeneralization*

nun das Cliché *MultiLevelGeneralization* von *Generalization*, so kann zum Beispiel in der Musterspezifikation des Design Patterns *Composite* das Cliché *Generalization* polymorph verwendet werden.

Die Vererbung von Mustern läßt sich nur in einem Klassendiagramm der Musterspezifikationen definieren. In Abbildung 3.17 ist das Klassendiagramm zu sehen. Hier wurde eine Vererbung zwischen *Generalization* und *MultiLevelGeneralization* eingetragen.

Alle Clichés und Design Patterns, die im Laufe dieser Arbeit spezifiziert wurden, sind im Anhang A zu finden. In dem Musterkatalog sind unter anderem die bereits aufgeführten Muster, als auch zusätzliche Clichés enthalten, die für weitere Design Patterns benötigt wurden, aber nicht näher erläutert werden.

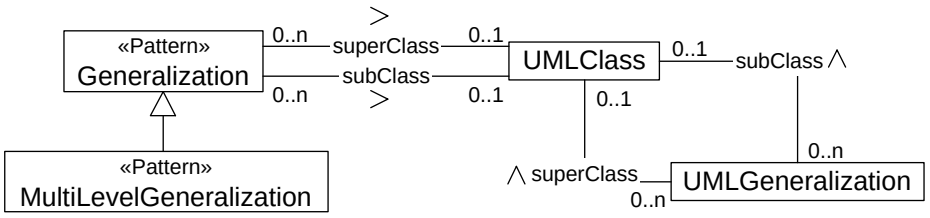

Abbildung 3.17.: Klassendiagramm des Clichés *MultiLevelGeneralization*

4. Der Inferenzmechanismus

Nach der statischen Musterspezifikationssprache muß nun ein Algorithmus zur Erkennung der Musterausprägungen definiert werden. In Abschnitt 4.1 werden zunächst der Algorithmus, der bisher in FUJABA zur Design Pattern-Erkennung gedient hat, und die damit auftretenden Probleme vorgestellt. Zur Erklärung des neuen Ansatzes werden im Abschnitt 4.2 die Generic Fuzzy Reasoning Nets (GFRNs) sowie einige notwendige syntaktische und semantische Änderungen an diesen eingeführt. Der Inferenzmechanismus besteht aus zwei Teilen. Der erste Teil, der Erkennungsprozeß, ist Thema des Abschnitts 4.3. Der zweite Teil besteht aus der Auswertung eines Fuzzy Petrinetzes, das eine Bewertung der Analyseergebnisse liefert. Diese Auswertung ist Inhalt des Abschnitts 4.4.

4.1. Bisheriger Inferenzmechanismus in Fujaba

Der bisherige Erkennungsprozeß wird durch eine zentrale Inferenzmaschine gesteuert. Für jedes spezifizierte Muster existiert eine Annotationsmaschine, die sich bei der zentralen Inferenzmaschine anzumelden hat. Desweiteren teilt jede Annotationsmaschine der Inferenzmaschine die Typen der Objekte mit, die sie für die Mustererkennung initial benötigt.

Die Musterspezifikationen sind bereits modular aufgebaut. Daraus ergibt sich eine Abhängigkeit zwischen den Spezifikationen. Die jeweiligen Annotationsmaschinen müssen diese Abhängigkeiten berücksichtigen. Zu diesem Zweck gibt jede Annotationsmaschine an, welche anderen Annotationsmaschinen zuvor ihre Analysen abgeschlossen haben müssen.

Die zentrale Inferenzmaschine startet in der Reihenfolge, die durch die Abhängigkeiten vorgegeben ist, die Annotationsmaschinen. An eine Annotationsmaschine werden jeweils alle Objekte des von ihr benötigten Typs übergeben.

Bei diesem Algorithmus zur Mustererkennung treten verschiedene Probleme auf. Zunächst einmal muß jede Annotationsmaschine alle anderen Annotationsmaschinen kennen, von denen sie abhängig ist. Werden neue Muster eingeführt, müssen Annotationsmaschinen geändert werden. Ein neues Muster, das beispielsweise von einem bereits existierenden Muster erbt, kann bei der Mustererkennung durch Polymorphie an die Stelle des Elternmusters treten. Alle Annotationsmaschinen, die

abhängig von der Annotationsmaschine des Elternmusters sind, müssen dahingehend angepaßt werden, daß sie auch die Annotationsmaschine des Kindmusters als Vorgänger bei der Inferenzmaschine eintragen.

Diesem Problem wird durch die Abspaltung der Informationsverwaltung über Abhängigkeiten aus den Annotationsmaschinen begegnet. Die Informationen werden von nun an in einem GFRN gehalten, aus dem der Inferenzmechanismus sie auslesen kann.

Ein weitere Schwachstelle des bisherigen Inferenzmechanismus ist die fehlende Überprüfung von Vorbedingungen. Die Annotationsmaschinen werden unabhängig von der Tatsache gestartet, ob Vorgängermaschinen bei der Suche erfolgreich waren. Die Annotationsmaschinen überprüfen also auch dann die an sie übergebenen Objekte, wenn die Vorbedingungen nicht erfüllt sind, also benötigte Annotationen durch Vorgängermaschinen nicht erzeugt worden sind.

In dem GFRN, welches die Informationen über Abhängigkeiten der Muster hält, sind gezielte Implikationsketten enthalten. Daraus können Vorbedingungen und Folgerungen abgelesen werden. So können die Annotationsmaschinen abhängig von den Vorbedingungen gestartet werden und die Folgerungen bei einer positiven Suche untersucht werden.

4.2. Generic Fuzzy Reasoning Net

Generic Fuzzy Reasoning Nets wurden an der Universität Paderborn von Jens Jahnke entwickelt [Jah99]. Sie dienen dazu, Kenntnisse und Erfahrungswerte über das Reverse Engineering relationaler Datenbanken formal zu repräsentieren.

Ein GFRN ist ein gerichteter Graph bestehend aus Prädikaten und Implikationen. Prädikate werden als Ovale dargestellt, Implikationen als Rechtecke. Prädikate sind mit Implikationen durch gerichtete Kanten verbunden. Kanten mit einer ausgefüllten Pfeilspitze negieren eine Schlußfolgerung.

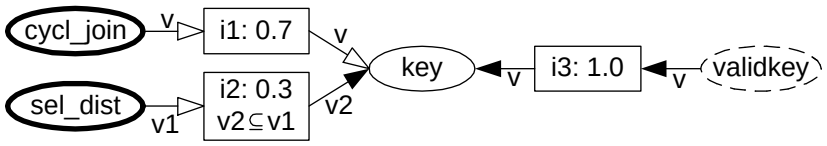

Abbildung 4.1.: Ein einfaches Generic Fuzzy Reasoning Net

Abbildung 4.1 zeigt ein einfaches GFRN aus vier Prädikaten und drei Implikationen. Die Implikationen besitzen einen Namen, einen Vertrauenswert zwischen 0 und 1, sowie optionale Bedingungen. In den Bedingungen können formale Parameter der ein- und ausgehenden Kanten verwendet werden. Die Implikation i_2 enthält bei-

spielsweise die Bedingung $v_2 \subseteq v_1$, zusammengesetzt aus den formalen Parametern v_1 der einlaufenden und v_2 der auslaufenden Kante.

Die Prädikate lassen sich in drei Kategorien aufteilen, die starken, schwachen und aufgeschobenen Axiome. Starke Axiome sind nicht widerlegbare Fakten, ihre Prädikate dürfen keine von einer Implikation einlaufenden Kanten haben. Während der Auswertung bleiben ihre Instanzen, Stellen im FPN, unverändert. In der graphischen Notation eines GFRN sind sie mit einem dickeren Rahmen gekennzeichnet. Im Beispiel sind die Prädikate *cycl_join* und *sel_dist* starke Axiome.

Schwache Axiome stellen die im Laufe der Inferenz zu untersuchenden Annahmen dar. Sie werden durch ein Oval mit einem durchgezogenen dünnen Rahmen dargestellt. Das Prädikat *key* repräsentiert ein solches schwaches Axiom.

Die aufgeschobenen Axiome stellen aufwendige Analyseoperationen dar. Sie werden nur überprüft, wenn sich starke Hinweise auf ihre Gültigkeit ergeben. Sie sind durch gestrichelte Ovale dargestellt. In Abbildung 4.1 ist *validkey* ein solches aufgeschobenes Axiom.

Das Cliché *cycl_join* steht für eine Verbund-Anweisung aus der zu untersuchenden Datenbank über Attribute derselben Tabelle. Anhand eines Attributs findet eine Unterscheidung der Tupel statt. Das deutet darauf hin, daß dieses Attribut eine Schlüsseleigenschaft besitzt. Implikation i_1 modelliert diese Schlußfolgerung mit einem Vertrauenswert von 0.7, da es kein sicherer Schluß ist.

Das Schlüsselwort *distinct* in einer Anfrage an die Datenbank besagt, daß kein Tupel in der Ergebnisrelation doppelt vorhanden sein darf. Das Cliché *sel_dist* beschreibt diese Anfrage. Die Existenz einer solchen Anfrage läßt vermuten, daß keine Teilmenge der beteiligten Attribute ein Schlüssel sein kann. Implikation i_2 fließt also negativ in das schwache Axiom *key* ein.

Die Implikation i_3 schließt von dem aufgeschobenen Axiom *valid_key* mit Vertrauenswert 1.0 negativ auf die Schlüsseleigenschaft. Sollte das aufgeschobene Axiom überprüft und festgestellt werden, daß es nicht gilt, so gilt die Schlüsseleigenschaft ebenfalls nicht.

Um GFRNs für die Inferenz bei der Cliché- und Design Pattern-Erkennung auf Software-Quelltexten nutzen zu können, müssen sie angepaßt werden. Sowohl die Syntax als auch die Semantik zur Auswertung der GFRNs sind davon betroffen. Im Folgenden wird auf diese Änderungen eingegangen.

4.2.1. Anpassung der Syntax

Die Dreiteilung der Prädikate ist bei der Anpassung aufgehoben worden. Es existieren nur noch die starken Axiome, die im Folgenden lediglich als Axiome bezeichnet werden. Schwache Axiome werden Prädikat genannt.

Aufgeschobene Axiome stellen sogenannte „goal-driven“-Analysen dar. Diese Analysen sind die Top Down-Anteile im Inferenzmechanismus der GFRNs. Im Gegensatz dazu gibt es „data-driven“-Analysen, die Bottom Up-Anteile. Die GFRNs

sind ursprünglich für das Anwendungsgebiet der relationalen Datenbanken entwickelt worden. Daher werden die Bedingungen in Implikationen des GFRNs durch eine relationale Algebra formuliert. Diese Bedingungen lassen sich häufig nicht in die Rückrichtung schließen. Die „goal-driven"-Analysen können aus diesem Grund nur auf Basisfakten ausgeführt werden. Schwache Axiome, die nicht auf Basisfakten aufbauen, sondern Vorbedingungen besitzen, können nicht als Top Down-Analysen überprüft werden. Diese Einschränkung wird aufgehoben. Eine Unterscheidung zwischen schwachen und aufgeschobenen Axiomen ist also nicht mehr notwendig. Sie verschmelzen zu den Prädikaten.

Axiome stehen für Fakten, die im abstrakten Syntaxbaum bereits nach dem Parsen der Quelltexte vorhanden sind. Von ihnen ausgehend kann mit der Mustererkennung begonnen werden. In der graphischen Darstellung eines GFRN werden sie durch fett umrandete Ovale repräsentiert. Ihre Namen entsprechen denen von Klassen aus dem ASG-Metamodell. Prädikate repräsentieren Muster, die zu untersuchen sind. Sie werden durch dünn umrandete Ovale im GFRN gezeichnet.

Die Implikationen enthalten als Bedingungen keine relationalen Ausdrücke mehr. Statt dessen enthalten sie den schwarzen Teilgraphen einer Musterspezifikation. Dieser Teilgraph stellt die Bedingung dar, die erfüllt sein muß, damit die Implikation schließen kann. Vorbedingungen für eine Implikation sind zum einen alle in der Musterspezifikation als Musterobjekte enthaltenen Clichés. Sie werden als Prädikate in das GFRN aufgenommen. Zum anderen werden alle ASG-Objekte, die in der Musterspezifikation als Trigger markiert sind, als Axiome zu Vorbedingungen der Implikation. Der Vertrauens- und der Schwellwert aus der Musterspezifikation werden ebenfalls in der Implikation angegeben.

Dienen Objekte in der Musterspezifikation als Trigger, so sind die von den entsprechenden Axiomen beziehungsweise Prädikaten in die Implikation der Musterspezifikation einlaufenden Kanten fett gezeichnet. Diese Kanten werden im Folgenden als Triggerkanten bezeichnet.

In Abbildung 4.2 ist ein Ausschnitt eines modifizierten GFRN zu sehen. Es ist die Regel, die auf das Design Pattern *Composite* schließt. Die drei Clichés *Association*, *Generalization* und *MultiDelegation* sind als vorbedingende Prädikate wiederzufinden. Die Implikation enthält sowohl den Vertrauens- und den Schwellwert, als auch den im ASG zu suchenden Teilgraphen. Wird dieser gefunden, kann auf die Existenz eines *Composite*-Design Patterns geschlossen werden.

Eine zusätzliche Kante ist zwischen Prädikaten eingeführt worden. Sie modelliert eine Vererbungsbeziehung zwischen Mustern. Das Prädikat des Kind-Musters verweist mit Hilfe eines Pfeils auf das Prädikat des Eltern-Musters. In Abbildung 4.3 ist eine solche Vererbungskante zwischen den Prädikaten *MultiLevelGeneralization* und *Generalization* zu finden.

Die Implikationen in dieser Abbildung sind vereinfacht dargestellt. Die Graphen innerhalb der Implikationen werden wegen der kompakteren Form weggelassen. Nur die Vertrauens- und Schwellwerte werden in die Implikationen übernommen.

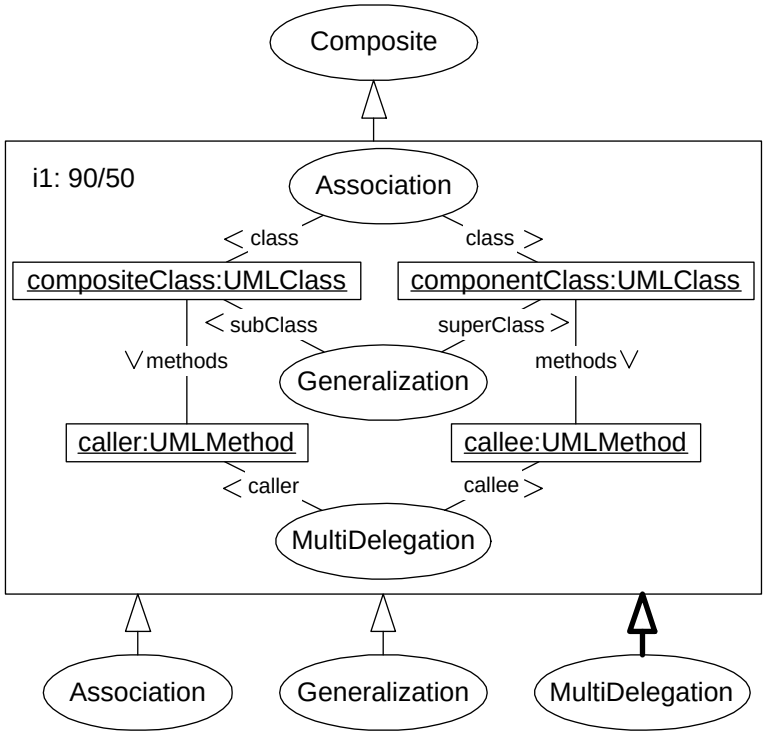

Abbildung 4.2.: Die Musterspezifikation des Design Pattern *Composite* im GFRN

4.2.2. Anpassung der Semantik

Wie bereits im vorigen Abschnitt erwähnt, gibt es nur noch eine zweiteilige Unterscheidung in Axiome und Prädikate. Die aufgeschobenen Axiome aus den ursprünglichen GFRNs, die „goal-driven"-Analysen kennzeichnen, entfallen. Dadurch ergibt sich eine neue Definition für den Top Down-Anteil des Inferenzmechanismus. Alle Prädikate in der angepassten Version der GFRNs können auch in einer Top Down-Analyse überprüft werden.

Möglich ist diese Änderung durch die Verwendung von Graphtransformationen anstatt der relationalen Algebra als Bedingungen der Implikationen. Eine Graphtransformation, die in diesem Zusammenhang die Erkennung eines Musters darstellt und durch die Musterspezifikation gegeben ist, fügt nur neue Informationen zum abstrakten Syntaxgraphen hinzu, sie zerstört keine Informationen. Eine Graphtransformation benötigt einen Trigger als Bezugspunkt. Bei der Auslösung der Mustererkennung muß also ein Trigger vorhanden sein. Soll die oben genannte Forderung erfüllt werden, muß sowohl in der Bottom Up-, als auch in der Top Down-Analyse solch ein Trigger zur Verfügung stehen. Dies stellt die Annotationsmaschine sicher, wie in Abschnitt 2 näher erläutert werden wird.

Die Entscheidung, welche Teile des GFRN in einer Bottom Up- und welche in

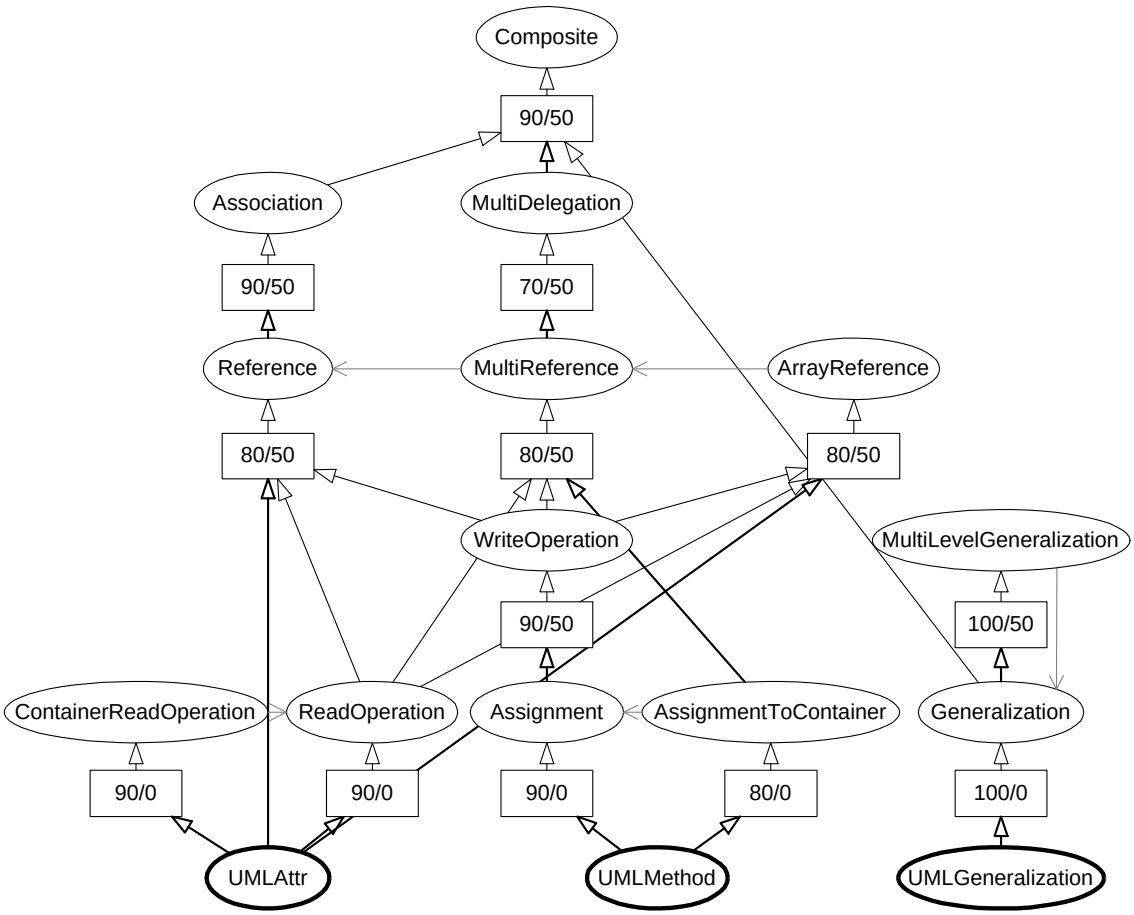

Abbildung 4.3.: Ausschnitt eines GFRNs

einer Top Down-Analyse zu überprüfen sind, wird nicht mehr statisch im GFRN festgelegt, sondern vom Erkennungsprozeß getroffen, der im folgenden Abschnitt erläutert wird.

4.3. Der Erkennungsprozeß

Der erste Teil des Inferenzmechanismus ist der Mustererkennungsprozeß. Da weder reine Bottom Up- noch Top Down-Analysen vielversprechende Ergebnisse hinsichtlich der Skalierbarkeit der Algorithmen liefern, ist eine kombinierte Bottom Up/Top Down-Analyse implementiert worden. Als Grundlage für die Analyse dient ein GFRN, welches alle Abhängigkeiten der Musterspezifikationen enthält.

Das GFRN ist ist mit Hilfe mehrerer Klassen realisiert worden. Die beiden wichtigsten Klassen sind *Predicate*, die die Prädikate und Axiome des GFRN modelliert,

```
Predicate::evaluate (UMLIncrement incr, boolean bottomUp)
 1: if predicate is axiom
 2:   evaluateConsequentPredicates (incr)
 3: else
 4:   engine = GFRNInferenceEngine.getFromEngines (predicateName)
 5:   if not engine.allCasesEvaluated (incr)
 6:     for all antecedent implications antImpl do
 7:       antImpl.evaluate (incr, false, engine)
 8:     annotation = engine.annotate (incr)
 9:     if annotation exists
10:       evaluateConsequentPredicates (annotation)
11:       if generalized predicate genPred exists
12:         genPred.evaluateConsequentPredicates (annotation)
13:     if not annotation exists and not bottomUp
14:       for all specialized predicates specPred do
15:         specPred.evaluate (incr, false)
```

Abbildung 4.4.: Die Methode *evaluate* der Klasse *Predicate*

```
Predicate::evaluateConsequentPredicates (UMLIncrement incr)
1:  for all consequent implications consImpl do
2:    if consImpl is triggered
3:      consImpl.evaluate (incr, true)
```

Abbildung 4.5.: Die Methode *evaluateConsequentPredicates* der Klasse *Predicate*

```
Implication::evaluate (UMLIncrement incr, boolean bottomUp,
                       Engine engine)
1:  if bottomUp
2:    for all consequent predicates consPred do
3:      consPred.evaluate (incr, true)
4:  else
5:    for all antecedent predicates antPred do
6:      if not antPred is axiom
7:        triggers = engine.getTrigger (incr, antPred)
8:        for all trigger in triggers do
9:          antPred.evaluate (trigger, false)
```

Abbildung 4.6.: Die Methode *evaluate* der Klasse *Implication*

und *Implication*, die die Implikationen modelliert. In den Abbildungen 4.4, 4.5 und 4.6 sind die wichtigsten Methoden der beiden Klassen in Pseudocode verfasst zu sehen. Sie beschreiben den kombinierten Bottom Up/Top Down-Algorithmus zur Mustererkennung.

Die Mustererkennung wird gesteuert durch die Inferenzmaschine. Sie verwaltet alle Annotationsmaschinen, die zur Mustererkennung notwendig sind. Desweiteren hält sie eine Instanz des GFRN, in dem die Abhängigkeiten der Annotationsmaschinen enthalten sind.

Die Axiome des GFRN repräsentieren Klassen aus dem Metamodell des abstrakten Syntaxgraphen. Die Inferenzmaschine beginnt damit, im ASG alle Objekte aufzusammeln, die vom Typ dieser Klassen sind. Diese Objekte bilden die Fakten, auf denen der Erkennungsprozeß aufsetzen kann. Die Inferenzmaschine durchläuft alle Fakten nacheinander. Sie startet mit jeweils einem Faktum auf dem Axiom, das den entsprechenden Typ des Faktums repräsentiert, eine Iteration des Erkennungsprozesses, indem sie die Methode *Predicate::evaluate* aufruft. Als aktuelle Parameter werden das Faktum, also ein Objekt des ASG vom Typ *UMLIncrement*[1], und der boolsche Wert *true* übergeben. Der Algorithmus beginnt also mit der Bottom Up-Phase der Analyse.

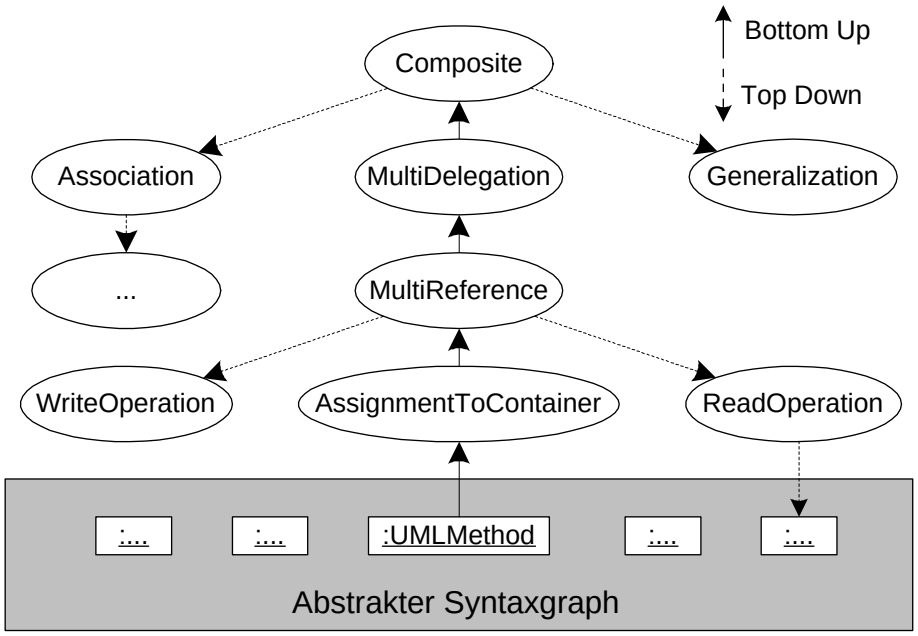

Abbildung 4.7.: Inferenz am Beispiel des *Composite*

Als Beispiel soll die Erkennung einer *Composite*-Musterausprägung dienen. Ab-

[1] *UMLIncrement* ist die Basisklasse aller Klassen im Metamodell des abstrakten Syntaxgraphen

bildung 4.2 zeigt ein GFRN, das alle Muster aus dem Katalog aus Anhang A enthält, die zu einer Erkennung des Design Patterns *Composite* notwendig sein können. In Abbildung 4.7 ist eine graphische Beschreibung dieser Beispielerkennung zu sehen. Bei der Beschreibung des Algorithmus wird im Folgenden davon ausgegangen, daß alle Mustererkennungen positiv verlaufen, also Ausprägungen des jeweiligen Musters gefunden werden.

Die Inferenzmaschine startet eine Iteration mit dem Aufruf der Methode *evaluate* auf dem Axiom *UMLMethod* mit einem Objekt der Klasse *UMLMethod* aus dem abstrakten Syntaxbaum als Parameter. In den Zeilen 1 und 2 der Methode *Predicate::evaluate* aus Abbildung 4.4 ist beschrieben, daß nun die konsequenten Prädikate ausgewertet werden müssen. Aus dem GFRN in Abbildung 4.2 ist abzulesen, daß dazu das Prädikat *AssignmentToContainer* zählt.

Die Methode *Predicate::evaluateConsequentPredicates*, abgedruckt in Abbildung 4.5, zeigt, welche Prädikate in der Bottom Up-Phase, die durch die Evaluierung der Konsequenzen eines Prädikates geprägt ist, auszuwerten sind. Nur Prädikate, deren Implikationen durch das aktuelle Prädikat beziehungsweise Axiom getriggert werden, sind auszuwerten. Die Triggerung einer Implikation ist dem GFRN durch die fett gezeichneten Triggerkanten zu entnehmen. Die Evaluierung, aufgerufen auf der getriggerten Implikation, wird durch die Methode *Implication::evaluate* (Abbildung 4.6, Zeilen 1-3) an die Prädikate weitergereicht.

Nun beginnt die erste Auswertung eines Prädikates. Da das aktuelle Prädikat *AssignmentToContainer* kein Axiom ist, wird zunächst die Annotationsmaschine für dieses Prädikat von der Inferenzmaschine angefordert (Abbildung 4.4, Zeile 4). Als nächstes wird in Zeile 5 überprüft, ob das aktuelle Objekt, hier eine Methode aus dem ASG, bereits von der Maschine untersucht worden ist. Diese Überprüfung verhindert Endlosschleifen bei der Analyse. Die technische Realisierung der Überprüfung wird in Abschnitt 6.2.2 behandelt. Im Folgenden ist davon auszugehen, daß alle Objekte zum ersten Mal von der jeweiligen Annotationsmaschine untersucht werden.

Wurde das Objekt noch nicht von der Maschine untersucht, wird die Evaluierung fortgesetzt. Ansonsten endet die Evaluierung des Prädikates hier, da sie bereits stattgefunden hat und keine neuen Erkenntnisse zu erwarten sind. In den Zeilen 6 und 7 wird sichergestellt, daß alle vorbedingenden Prädikate evaluiert worden sind. Zur Auswertung der Vorbedingungen wird in die Top Down-Phase umgeschaltet. Im Algorithmus ist dies in Zeile 7 an den aktuellen Parametern des Methodenaufrufs zu sehen. Der formale Parameter *bottomUp* der Methode *Implication::evaluate* wird mit dem boolschen Wert *false* belegt.

Aus dem GFRN ist abzulesen, daß das Prädikat *AssignmentToContainer* keine weiteren Vorbedingungen als das Axiom *UMLMethod* hat. Die Top Down-Phase bricht also in den Zeilen 5 und 6 der Methode *Implication::evaluate* ab. Da die Vorbedingungen geprüft worden sind, arbeitet der Algorithmus in der Zeile 8 aus Abbildung 4.4 weiter. Hier wird der Annotationsmaschine das zu untersuchende Objekt übergeben. Die technische Umsetzung der Methode *annotate* einer Annota-

tionsmaschine wird in Abschnitt 2 erläutert.

Die Annotationsmaschine erzeugt bei erfolgreicher Suche eine Annotation, die zurückgegeben wird. Da in diesem Beispiel davon ausgegangen wird, daß die Mustererkennungen erfolgreich verlaufen, fährt der Algorithmus mit Zeile 10 fort. Hier wird nun wieder die Auswertung aller Konsequenzen des Prädikates angestoßen. Als aktueller Parameter wird die neu erzeugte Annotation übergeben. Im GFRN ist wieder die Information zu finden, daß das Prädikat *MultiReference* von *AssignmentToContainer* getriggert wird.

Da das Prädikat *MultiReference* jedoch weitere vorbedingende Prädikate besitzt, wird in die Top Down-Phase gewechselt, die aber nicht sofort wie bei dem Prädikat *AssignmentToContainer* abbricht. Es sind zunächst die Prädikate *ReadOperation* und *WriteOperation* zu evaluieren. Für die Auswertung der Vorbedingungen sind Trigger notwendig. Die Annotationsmaschine des Musters *MultiReference* liefert diese Trigger, abhängig von dem aktuell zu untersuchenden Objekt und dem vorbedingenden Prädikat. Trigger kann das Objekt selber sein, aber auch Objekte aus seinem direkten Kontext sind möglich. In Abschnitt 2 wird die Technik erläutert, die notwendig ist, um potentielle Trigger für Vorbedingungen zu bestimmen.

Für alle von der Annotationsmaschine zurückgelieferten Trigger sind die vorbedingenden Prädikate zu untersuchen. Der Kontext des aktuell zu untersuchenden Objektes ist nun auf Musterausprägungen passend zu den Vorbedingungen untersucht und eventuelle Annotationen wurden erzeugt. Es wird wieder zurück in die Bottom Up-Phase gewechselt und die eigentliche Evaluierung des Prädikates *MultiReference* kann durchgeführt werden.

Von dem Prädikat *MultiReference* wird auf das Prädikat *MultiDelegation* geschlossen. Bei Existenz einer Ausprägung für die *MultiDelegation* kann mit der Evaluierung des Design Patterns *Composite* begonnen werden. Außer dem Prädikat *Association* hat das Prädikat *Composite* auch noch die *Generalization* als Vorbedingung. Wenn für beide jeweils eine Annotation erzeugt wurde, kann die Annotationsmaschine eine Ausprägung des Design Patterns *Composite* suchen und annotieren.

Da es keine Konsequenzen für ein *Composite* gibt, ist hiermit eine Iteration des Erkennungsprozesses abgeschlossen. Die Inferenzmaschine kann nun die nächste Iteration mit einem weiteren Objekt des abstrakten Syntaxbaumes starten.

In der Methode *Predicate::evaluate* sind zwei Sonderfälle im Zusammenhang mit der Vererbung von Mustern enthalten. Der erste Fall ist mit den Zeilen 11 und 12 abgedeckt. Ist eine Annotation für das aktuelle Prädikat erzeugt worden, das heißt, wurde eine Musterausprägung gefunden, so müssen die Konsequenzen des eventuell vorhandenen Elternmusters in der Bottom Up-Phase evaluiert werden. Die für das Kindmuster erzeugte Annotation kann auch für Konsequenzen des Elternmusters verwendet werden. Es gibt nur ein Elternmuster, da bei der Mustervererbung nur die Einfachvererbung erlaubt ist.

Eine ähnliche Situation tritt in der Top Down-Phase auf. Konnte keine Annotation bei der Evaluierung eines Prädikates gefunden werden, so müssen alle spe-

zialisierten Prädikate, also die Prädikate der erbenden Muster, untersucht werden. Wird eine Annotation durch die Auswertung eines Kindmusters gefunden, so kann sie anstelle der Annotation des Elternmusters verwendet werden. Dies geschieht in den Zeilen 13 bis 15.

4.4. Auswertung durch ein Fuzzy Petrinetz

Nachdem die Mustererkennung abgeschlossen ist, kann der zweite Teil des Inferenzmechanismus gestartet werden. Die Bewertung der Analyseergebnisse wird mit Hilfe eines Fuzzy Petrinetzes berechnet. Während des Erkennungsprozesses wird für jede erzeugte Annotation eine Stelle in einem Fuzzy Petrinetz angelegt. Die Implikationen, die zu den Annotationen geführt haben, werden durch Transitionen im FPN repräsentiert. Eine formale Definition der Fuzzy Petrinetze ist in [Jah99] zu finden.

Als Grundlage zur Bewertung der Analyseergebnisse dienen die Vertrauens- und Schwellwerte aus den Musterspezifikationen. Die Bewertung eines Analyseergebnisses, also einer Annotation, wird durch den Fuzzy Belief ausgedrückt. Im Gegensatz zu dem Vertrauenswert der Musterspezifikation ist der Fuzzy Belief nicht statisch vorgegeben, sondern wird während der Auswertung des FPNs in den Stellen berechnet.

Die Stellen, die Quellen im FPN sind, also keine von einer Transition einlaufende Kante besitzen, erhalten als Fuzzy Belief initial den Vertrauenswert der Implikation, die zu der korrespondierenden Annotation geführt hat. Diese Annotationen sind direkt aus der Implikation eines Axioms im GFRN entstanden, da sie ebenfalls keine Vorgängerannotationen besitzen. Alle Transitionen übernehmen sowohl Vertrauens- als auch Schwellwert ihrer Implikation. Nachdem der Erkennungsprozeß beendet wurde, kann die Auswertung des FPNs durchgeführt werden.

Abbildung 4.8 zeigt den initialen Zustand eines FPN vor der Auswertung. Als Beispiel wurde die Erkennung einer *Composite*-Ausprägung aus dem letzten Abschnitt genommen. Die Stellen enthalten ihren Fuzzy Belief. Neben den Stellen des FPN ist jeweils der Typ der zugehörigen Annotation angegeben. Die Vertrauens- und Schwellwerte stehen ebenfalls neben den Transitionen.

Die Berechnung eines Fuzzy Petrinetzes erfolgt in mehreren Iterationen, so lange, bis das Netz stabil ist. Bei jeder Iteration können sich die Fuzzy Beliefs der Stellen ändern. Jede Iteration wiederum besteht aus zwei Phasen. In der ersten Phase wird für jede Transition ein *Fuzzy Truth Token* (FTT) berechnet. Zunächst wird das Minimum aus allen Fuzzy Beliefs der Stellen aus dem Vorbereich einer Transition und aus dem Vertrauenswert gebildet. Ist dieser Wert größer oder gleich dem Schwellwert der Transition, so bekommt das FTT diesen Wert, ansonsten ist das FTT gleich Null. Bei einem Wert ungleich Null gilt die Transition als aktiviert.

In der zweiten Phase einer Iteration schalten alle aktivierten Transitionen. Dabei wird für jede Stelle das Maximum aller FTTs der Transitionen aus dem Vorbereich

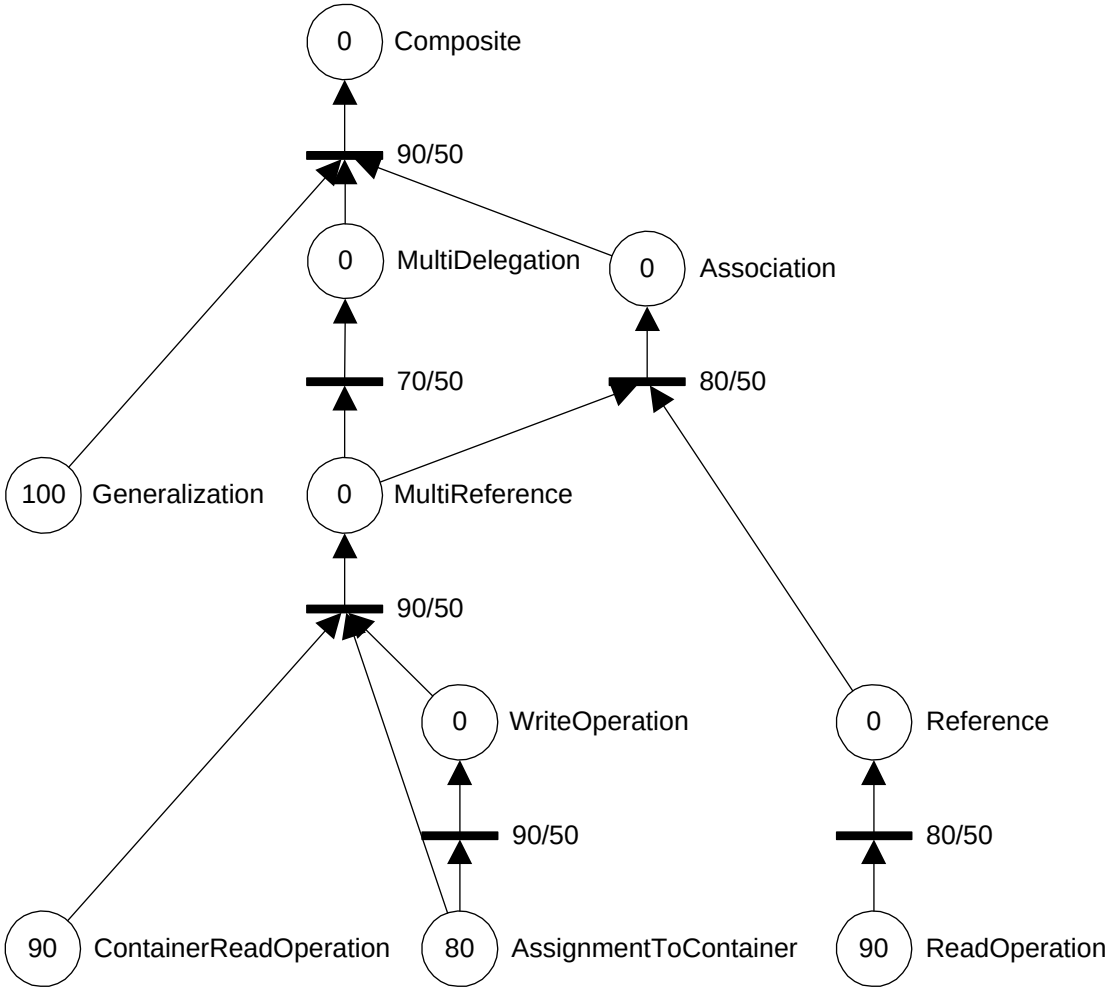

Abbildung 4.8.: Initialer Zustand eines FPN

der Stelle berechnet. Dieser Wert ist der neue Fuzzy Belief der Stelle.

Das Netz ist stabil, wenn in einer Iteration kein Fuzzy Belief einer Stelle mehr verändert wurde. Die Fuzzy Beliefs der Stellen können dann von ihren zugehörigen Annotationen übernommen werden. Sie drücken dann die Bewertung der Glaubwürdigkeit der Annotation aus. In Abbildung 4.9 ist das FPN in einem stabilen Zustand zu sehen.

Sobald das FPN in einem stabilen Zustand ist, gilt der Inferenzmechanismus als abgeschlossen. Nun kann der Benutzer in die Bewertung der Annotationen eingreifen. Diese Interaktion ist Thema des nächsten Kapitels.

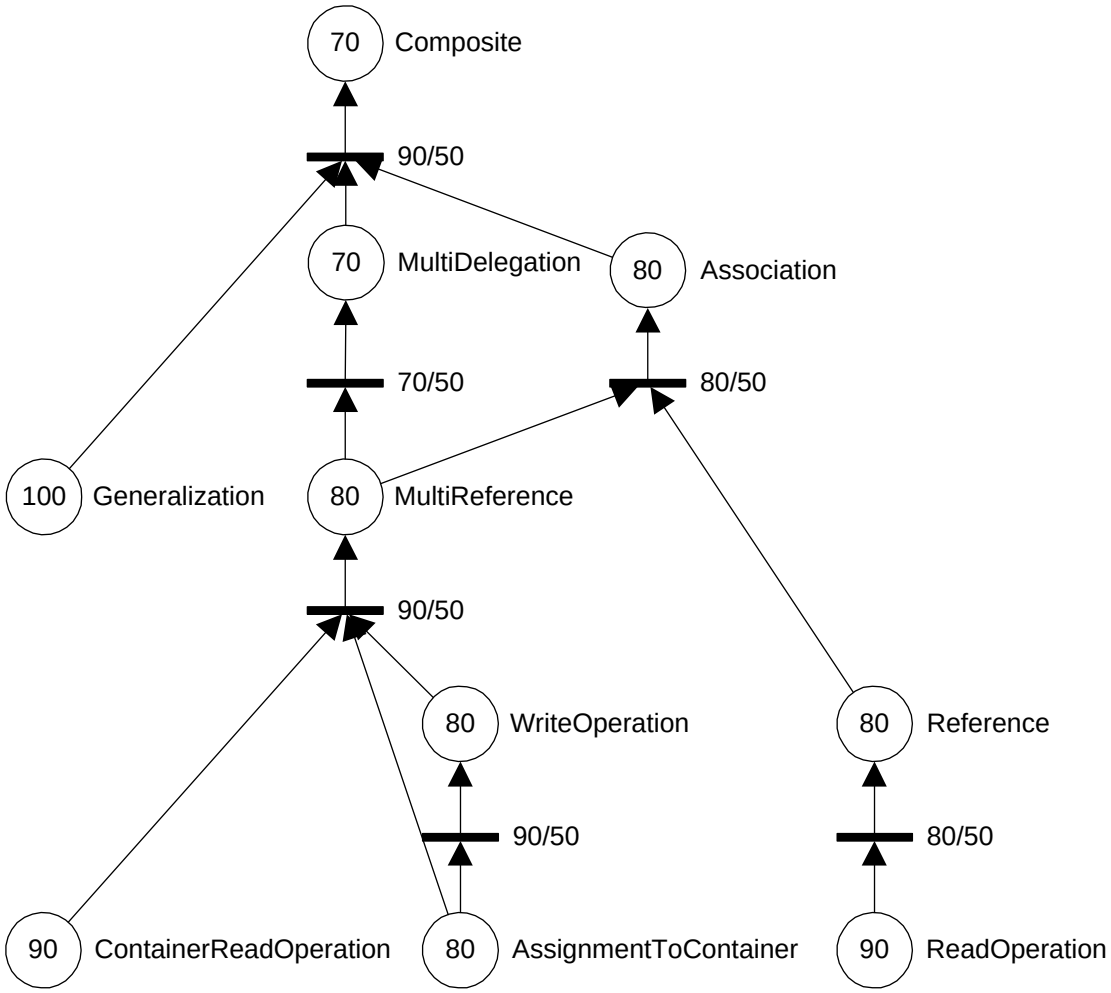

Abbildung 4.9.: Stabiler Zustand eines FPN

5. Interaktion mit dem Benutzer

5.1. Unterbrechung des Erkennungsprozesses

Jede Iteration des Erkennungsprozesses ist in sich abgeschlossen. Eine Iteration beginnt mit dem Starten der Evaluierung auf einem Axiom mit einem Objekt des abstrakten Syntaxbaumes. Sie endet, wenn alle sich ergebenden Konsequenzen abgearbeitet sind. Dies kann relativ schnell bei der Auswertung des ersten Prädikates erfolgen, sollte es nicht erfüllt sein. Aber auch länger andauernde Iterationen mit sehr vielen Wechseln zwischen Bottom Up- und Top Down-Phasen sind möglich. Bei einem Abbruch des Erkennungsprozesses innerhalb einer Iteration kommt es zu Informationsverlusten, da eventuell Konsequenzen nicht mehr überprüft würden. Allerdings kann der Erkennungsprozeß zwischen den Iterationen angehalten werden. Es sind dann nur ein Teil der Objekte aus dem abstrakten Syntaxbaum untersucht worden. An der gleichen Stelle kann der Prozeß dann aber auch wieder fortgesetzt werden.

Diese Unterbrechung kann bei großen zu untersuchenden Softwaresystemen von Vorteil sein. Hier kann der Mustererkennungsprozeß unter Umständen langwierig sein. Es sollte dem Benutzer allerdings möglich sein, sich bereits Zwischenergebnisse anzusehen. Aus diesem Grund wird dem Benutzer die Möglichkeit der Unterbrechung und des Abbruchs des Prozesses geboten. Bei einem Abbruch kann der Inferenzmechanismus nicht mehr fortgesetzt werden.

Bei einer Unterbrechung wie bei einem vollständigen Abbruch wird das bereits bestehende Fuzzy Petrinetz ausgewertet. Das ist möglich, weil das FPN nach jeder Iteration wieder in sich abgeschlossen ist. Die bereits vorhandenen Annotation sind also in dem gleichen Zustand, wie nach einem vollständigen Durchlauf des Inferenzmechanismus.

5.2. Änderung der Fuzzy Beliefs

Wie schon in der Motivation erläutert, ist es nicht möglich, bei einer werkzeugunterstützten Mustererkennung eine absolute Präzision zu erreichen. Es kann vorkommen, daß Musterausprägungen falsch erkannt werden. Ebenso kann der Inferenzmechanismus korrekt erkannte Musterausprägungen mit einem zu geringen Fuzzy

Belief versehen. Hier kann die Erfahrung und das Wissen des Benutzers einfließen.

Nachdem der Inferenzmechanismus abgearbeitet worden ist, wird der Benutzer die Analyseergebnisse auf für ihn interessante Daten hin untersuchen. Dabei wird er auch den Quelltext, also die Basisfakten der Analyse, ansehen. Bei einigen Annotationen kann er zu dem Schluß kommen, daß sie nicht korrekt sind, oder eventuell falsch bewertet worden sind. Die Fuzzy Beliefs der Annotationen sind deshalb vom Benutzer veränderbar. Glaubt der Benutzer, eine Annotation sei falsch, kann er den Fuzzy Belief auf 0% setzen. Ist er der Meinung, die Annotation sei vollkommen korrekt, setzt er den Fuzzy Belief auf 100%. Natürlich sind alle Werte zwischen 0% und 100% ebenfalls denkbar.

Nach einer oder mehrerer Korrekturen der Fuzzy Beliefs kann der Benutzer eine Revidierung des Fuzzy Petrinetzes anstoßen. Dies führt zu erneuten Iterationsrunden, so lange, bis das Netz wieder stabil ist. So können sich die Änderungen einiger weniger Fuzzy Beliefs auf Fuzzy Beliefs vieler anderer Annotationen auswirken.

In dem Beispiel aus Abbildung 4.9 würde sich die Änderung des Fuzzy Beliefs der Annotation vom Typ *MultiDelegation* auf die Annotation, die ein *Composite*-Design Pattern markiert, auswirken. Wenn der Benutzer sicher ist, daß eine Mehrfachdelegation vorliegt, ändert er den Fuzzy Belief auf 100%. Daraus ergibt sich für die Annotation des *Composite* nach der Neuberechnung ein Fuzzy Belief von 80%. Lehnt der Benutzer die Annotation der Mehrfachdelegation ab, weil nach seiner Analyse keine Mehrfachdelegation gegeben ist, so setzt er den Fuzzy Belief der entsprechenden Annotation auf 0%. Bei einer anschließenden Neuberechnung des FPNs würde in diesem Fall die Annotation des *Composite*-Musters einen Fuzzy Belief von 0% erhalten, da der Schwellwert der Transition nicht mehr erreicht würde. Die vorliegende Situation wäre dann keine Musterausprägung eines *Composite*-Design Patterns mehr.

Der Benutzer ist somit also in der Lage, die Bewertungen von Analyseergebnissen zu beeinflussen. Er kann durch eine Neuberechnung des FPNs die Konsequenzen seiner Änderungen nachvollziehen. Sollten sich Konsequenzen ergeben, die der Benutzer nicht akzeptieren will, kann er seine Änderungen auch wieder rückgängig machen und so den alten Zustand wiederherstellen.

6. Technische Realisierung

Dieses Kapitel behandelt die technische Umsetzung der in den Kapiteln 3 bis 5 vorgestellten Konzepte. Einen groben Überblickt über die verwendete Architektur der Mustererkennung gibt der Abschnitt 6.1. Die nächsten Abschnitte gehen dann detaillierter auf die einzelnen Teile ein. In Abschnitt 6.2 wird das Design der Annotationsmaschinen erläutert. Zusätzlich wird gezeigt, wie eine Annotationsmaschine aus der Musterspezifikation erstellt wird. Der Abschnitt 6.3 behandelt das Design eines Generic Fuzzy Reasoning Nets mit seinen Anpassungen. Die technische Umsetzung des Fuzzy Petrinetzes ist Thema des Abschnitts 6.4. In Abschnitt 6.5 wird auf die Steuerung des Inferenzmechanismus eingegangen.

6.1. Architektur

Die Klassen, die zur technischen Umsetzung dieser Diplomarbeit benötigt werden, sind in insgesamt vier Pakete aufgeteilt und in die FUJABA-Architektur integriert worden. Der Basispaketname für FUJABA ist *de.uni_paderborn.fujaba*. Alle weiteren Paketnamen werden im Folgenden relativ zu diesem Basispaket angegeben. Die die Inferenz steuernde Klasse sowie einige Hilfsklassen sind im Paket *patDetection* zu finden. Die anderen drei Gruppen der Klassen für die Annotationsmaschinen, das GFRN und das FPN sind in die Unterpakete *patDetection.engines*, *patDetection.gfrn* und *patDetection.fpn* gegliedert.

Abbildung 6.1 gibt einen Überblick über die Architektur der Mustererkennung. Ein zentraler Bestandteil der Mustererkennung ist der Musterkatalog. Aus dem Katalog wird pro Musterspezifikation eine Annotationsmaschine und eine Annotationsklasse generiert. Diese werden in das Paket *patDetection.engines* integriert. Aus dem gesamten Katalog werden die Abhängigkeiten der Musterspezifikationen extrahiert und in einem GFRN abgebildet. Von dem GFRN wird eine textuelle Repräsentation in XML-Notation als Datei gesichert. Diese Datei wird vom Inferenzmechanismus vor jedem Start einer neuen Inferenz eingelesen und das GFRN zurückgewonnen.

Aus dem GFRN erhält der Inferenzmechanismus die Information, welche Annotationsmaschinen benötigt werden, und erzeugt Instanzen der Maschinen. Er sammelt alle Basisfakten aus dem ASG, also Objekte des ASG, und startet anhand der Schlußfolgerungen aus dem GFRN die Annotationsmaschinen.

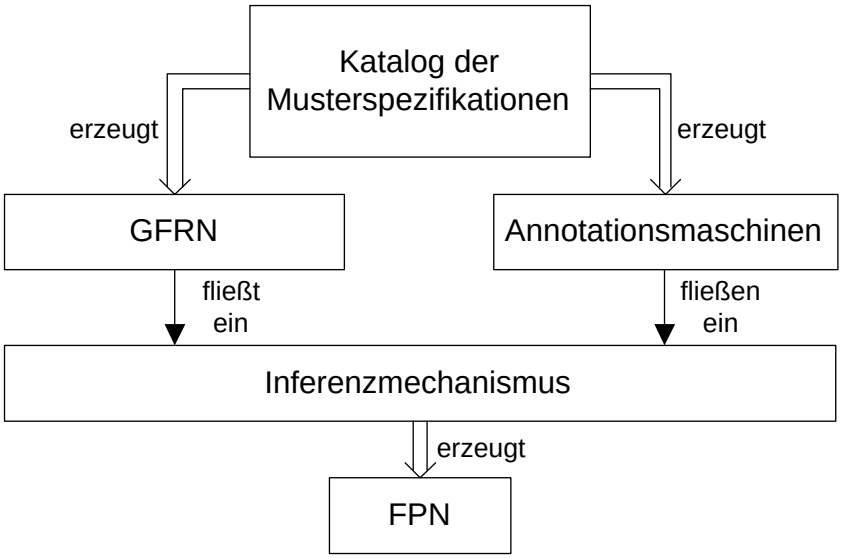

Abbildung 6.1.: Architektur der Mustererkennung

Während der Inferenz werden Annotationen und zugehörige Stellen im FPN erzeugt. Transitionen verbinden die Stellen im FPN analog zu den Schlußfolgerungen, die zu den Annotationen geführt haben. Zum Ende des Inferenzmechanismus wird das FPN ausgewertet und die Ergebnisse den Annotationen zugeordnet.

6.2. Die Annotationsmaschinen

6.2.1. Das Design der Annotationsmaschinen

Das Paket der Annotationsmaschinen besteht aus zwei abstrakten Oberklassen und ihren jeweiligen Implementierungen. Abbildung 6.2 zeigt in einem UML-Klassendiagramm einen Ausschnitt des Paketes *patDetection.engines*.

Zunächst einmal ist eine Klasse *GFRNDetectionEngine* als abstrakte Oberklasse aller Annotationsmaschinen eingeführt worden. Sie definiert zum einen die gemeinsame Schnittstelle aller Maschinen durch die abstrakten Methoden *allCasesEvaluated*, *annotate* und *getTrigger*, die im nächsten Abschnitt behandelt werden. Zum anderen bietet sie eine Methode *checkRegExpression* zur Auswertung regulärer Ausdrücke, wie sie in den Bedingungen von ASG-Objekten der Musterspezifikationen vorkommen. Zusätzlich verwaltet diese Klasse eine Menge von Objekten der Klasse *UMLIncrement*, die von der Maschine bereits untersucht worden sind. Alle Annotationsmaschinen erben direkt von *GFRNDetectionEngine* und müssen die drei abstrakten Methoden implementieren. Zu jeder Musterspezifikation existiert jeweils eine Annotationsmaschine, die den Namen der Spezifikation trägt, also des Musters,

erweitert durch das Suffix *Engine*.

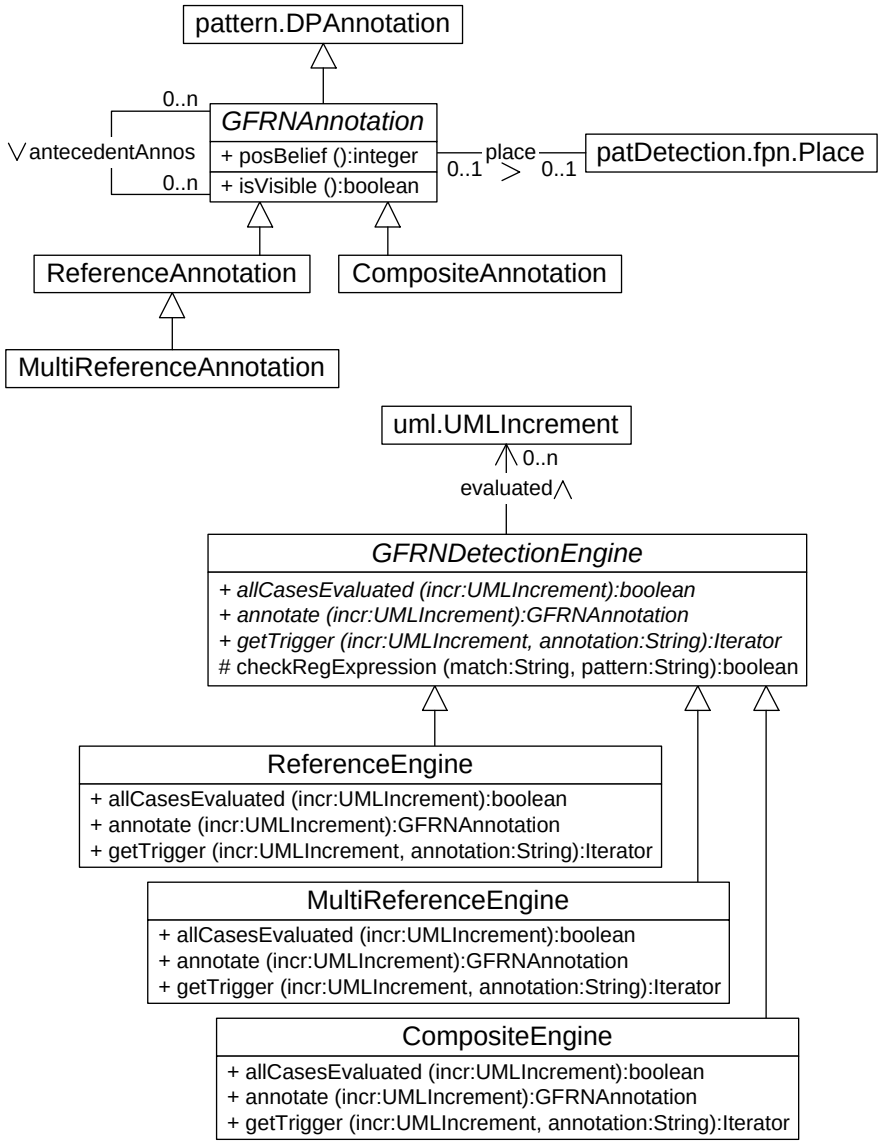

Abbildung 6.2.: Ausschnitt des Paketes *patDetection.engines*

Die Klasse *GFRNAnnotation* bildet die abstrakte Basisklasse aller Annotationsklassen des Inferenzmechanismus. Sie erbt von der Klasse *DPAnnotation*, die allgemeine Eigenschaften zur Annotation des abstrakten Syntaxgraphen zur Verfügung stellt und Teil des ASG-Metamodells von Fujaba ist. Die Klasse *GFRNAnnotation* enthält das Attribut *posBelief*, das den Fuzzy Belief der Annotation speichert. Die Methode *isVisible* gibt an, ob die Annotation in einem Diagramm von Fujaba angezeigt werden soll. Jede Annotation referenziert eine Stelle im Fuzzy Petrinetz,

die die Annotation bei der Auswertung der Analyseergebnisse repräsentiert. Desweiteren kennt eine Annotation alle vorbedingenden Annotationen, auf denen sie aufbaut. Dieses sind Annotationen, die zu der Musterausprägung gehören und Musterobjekte aus der Spezifikation abdecken. Die vorbedingenden Annotationen sind über die Assoziation *antecedentAnnos* zu erreichen.

Innerhalb der Hierarchie der Annotationen sind die Vererbungsbeziehungen zwischen den Mustern widergespiegelt. Erben zwei Muster voneinander, so ist dies auch bei ihren Annotationsklassen der Fall. In Abbildung 6.2 ist beispielsweise die Vererbung zwischen den Mustern *MultiReference* und *Reference* in der Vererbung ihrer Annotationsklassen zu erkennen. Bei der Suche nach Musterausprägungen können so durch Polymorphie auch Annotationen erbender Muster verwendet werden.

In Abbildung 6.2 sind als Beispiel nur die Klassen für die Annotationsmaschinen und die Annotationen der drei Muster *Reference*, *MultiReference* und *Composite* übernommen worden. Das Paket enthält jedoch alle Annotationsmaschinen und Annotationsklassen der Clichés und Design Patterns aus dem im Anhang A zu findenden Katalog.

6.2.2. Erzeugung einer Annotationsmaschine

Die Erzeugung einer Annotationsmaschine geschieht in zwei Schritten. Zunächst einmal werden aus der Musterspezifikation automatisch UML- und SDM-Diagramme [FNTZ98] konstruiert. In einem UML-Klassendiagramm wird die Klasse der Annotationsmaschine und die Klasse der zugehörigen Annotation erzeugt. Des weiteren werden Vererbungsbeziehungen und Assoziationen zu Hilfsklassen festgelegt. Dazu gehören unter anderem die abstrakten Oberklassen für alle Annotationsmaschinen und Annotationen.

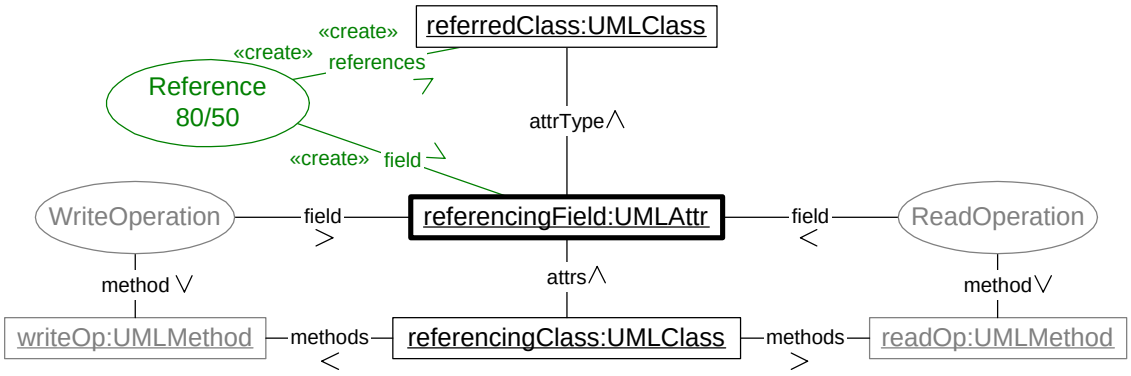

Abbildung 6.3.: Spezifikation des Clichés *Reference*

Im zweiten Schritt wird aus den UML- und SDM-Diagrammen Java-Quelltext erzeugt, der anschließend übersetzt werden kann. Es entstehen zwei neue Klassen, die

Annotationsmaschine und die dazugehörige Annotationsklasse. Diese beiden Klassen können dann sofort in die Cliché- und Design Pattern-Erkennung eingebunden werden, indem aus dem gesamten Katalog inklusive der neuen Spezifikation ein Generic Fuzzy Reasoning Net erzeugt wird. Dieses Netz wird in einer Datei abgelegt und beim Start des Inferenzmechanismus geladen.

In den nächsten Abschnitten werden die Methoden *allCasesEvaluated*, *annotate* und *getTrigger* der Annotationsmaschine am Beispiel der *ReferenceEngine* erläutert. Die Annotationsmaschine wurde auf Basis der Musterspezifikation aus Abbildung 6.3 erzeugt. Die Methoden werden in jeweils einem SDM-Diagramm spezifiziert.

Die Methode allCasesEvaluated

Die Annotationsmaschine verwaltet eine Liste aller Objekte, die sie bereits untersucht hat. Allerdings gibt es unterschiedliche Rollen, in der die Annotationsmaschine ein Objekt untersuchen kann. In dem Cliché *Generalization* aus Abbildung 3.15 zum Beispiel kann ein Objekt des Typs *UMLClass* in der Rolle *superClass* oder *subClass* verwendet werden. Der Inferenzmechanismus muß sicherstellen, daß ein Objekt für jede Rolle, die es in der zu untersuchenden Spezifikation einnehmen kann, evaluiert wird. Die Methode *allCasesEvaluated* übernimmt diese Überprüfung für ein gegebenes Objekt.

In Abbildung 6.4 ist ein Beispiel dieser Methode für die Annotationsmaschine des Musters *Reference* zu sehen. Einige Aktivitäten sind in diesem und in den folgenden Abbildungen zur besseren Erläuterung durchnumeriert. In Aktivität 1 wird zunächst das Gesamtergebnis der Methode auf den boolschen Wert *true* gesetzt. Die Methode testet dann, welchen Typ das übergebene Objekt hat. Es wird nur auf die Typen der Trigger einer Musterspezifikation und der Objekte überprüft, die annotiert[1] werden.

Sollte das Objekt dem jeweiligen Typ entsprechen, wird in Aktivität 2 getestet, ob das Objekt in der Rolle, in der der Typ verwendet werden kann, bereits untersucht wurde. Das Ergebnis dieses Tests wird in der boolschen Variablen *fujaba_Success* festgehalten. Der Wert dieser Variablen wird in Aktivität 3 dann mit dem Gesamtergebnis der Methode *und*-verknüpft. Dieser Test wird für jeden der möglichen Typen wiederholt.

Der Rollenname ist entweder „*trigger*“ für Trigger, die nicht annotiert werden, oder der Name des Links, über den das Objekt annotiert wird. In Beispiel 6.4 sind die Rollennamen „*references*“ und „*field*“.

Die Methode annotate

Die Methode *annotate* ist dafür zuständig, ein an sie übergebenes Objekt vom Typ *UMLIncrement* aus dem FUJABA-Metamodell und seinen Kontext auf eine Mu-

[1] Objekte, die eine «*create*»-Kante zu dem mit «*create*» gekennzeichneten Musterobjekt besitzen.

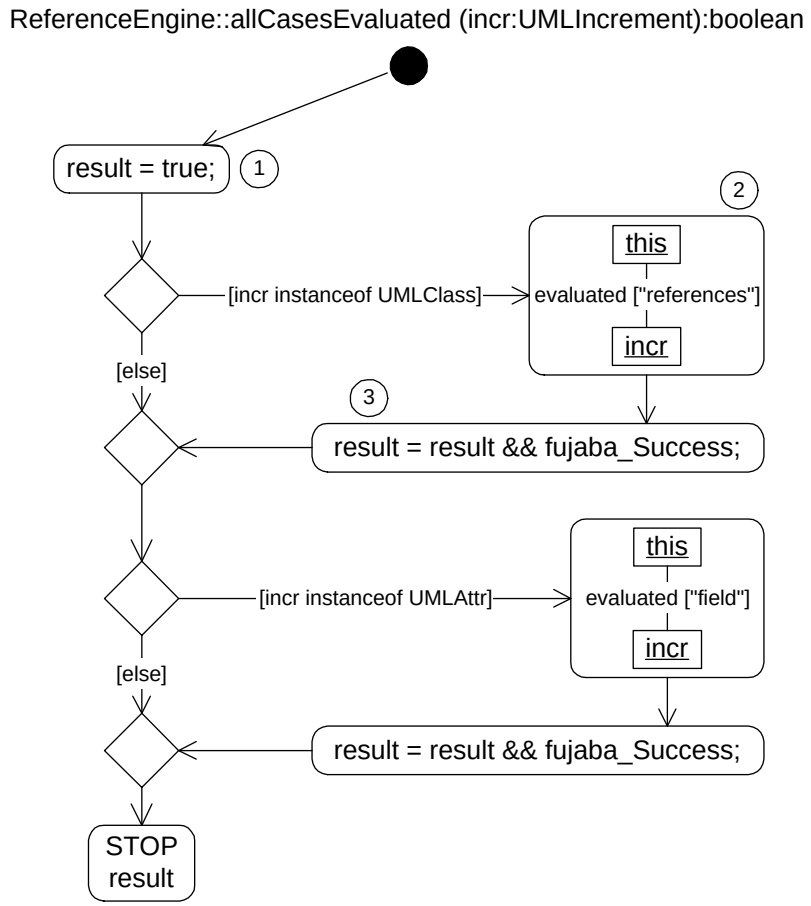

Abbildung 6.4.: Methode *allCasesEvaluated* der Klasse *ReferenceEngine*

sterausprägung hin zu untersuchen. Abbildung 6.5 zeigt die Methode *annotate* der Annotationsmaschine *ReferenceEngine* als SDM-Aktivitätsdiagramm.

In Aktivität Nummer 1 wird zunächst überprüft, ob das Objekt in der Rolle „*field*“ schon untersucht wurde. Falls ja, kann zur nächsten Rolle übergegangen werden. Ansonsten wird Aktivität 2 ausgeführt. Diese Aktivität beschreibt eine Objektwelt, wie sie, ausgehend vom übergebenen Objekt in der Rolle „*field* “, gefunden werden muß, damit eine Ausprägung des gesuchten Musters vorliegt. Kann eine solche Objektwelt nicht gefunden werden, dann wird in Aktivität 3 das Objekt in die Liste der in der Rolle „*field*“ überprüften Elemente aufgenommen und zur nächsten Rolle übergegangen.

Wenn eine Übereinstimmung zur beschriebenen Objektwelt im Kontext des zu untersuchenden Elementes gefunden werden konnte, liegt eine Ausprägung des beschriebenen Musters vor. Dann wird in Aktivität 4 zunächst eine eventuell bestehende Annotation gesucht, die diesen Tatbestand ausdrückt.

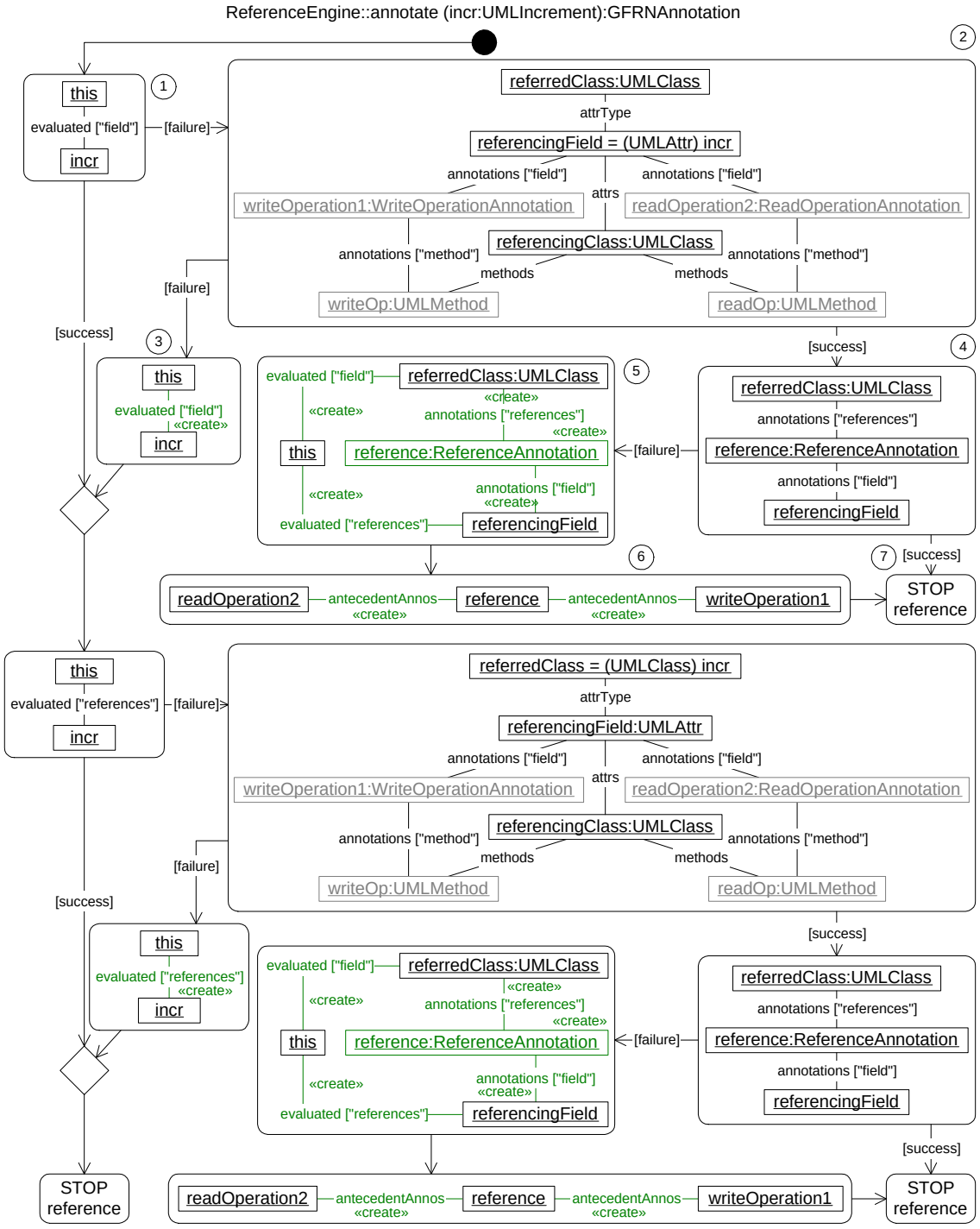

Abbildung 6.5.: Methode *annotate* der Klasse *ReferenceEngine*

Wird sie gefunden, kann die Methode in der Aktivität 7 erfolgreich beendet und die gefundene Annotation zurückgegeben werden. Im negativen Fall muß eine entsprechende Annotation erzeugt und mit den zu annotierenden Objekten in Aktivität 5 verknüpft werden. Desweiteren merkt sich die Annotationsmaschine, daß die annotierten Objekte in den jeweiligen Rollen evaluiert wurden. Aktivität 6 verknüpft die zuvor erzeugte Annotation mit allen Annotationen, die in der gefundenen Musterausprägung verwendet worden sind. Die Assoziation *antecedentAnnos* hält alle vorbedingenden Annotationen. Anschließend kann die Methode mit der Rückgabe der erzeugten Annotation verlassen werden.

Für jedes triggernde und jedes zu annotierende Objekt einer Musterspezifikation wird eine Fallunterscheidung, wie sie oben beschrieben worden ist, in die Methode *annotate* eingebaut. Im Beispiel gibt es noch einen zweiten Fall, der das übergebene Element auf die Rolle „*references*“ hin überprüft.

Die Methode getTrigger

Wird ein Prädikat für ein gegebenes Objekt evaluiert, so müssen vor der Mustererkennung zunächst die Vorbedingungen erfüllt sein. Um die Vorbedingungen in der Top Down-Phase zu überprüfen, werden Triggerobjekte für die entsprechenden Prädikate bestimmt. Dies übernimmt die Methode *getTrigger*. Als Parameter erhält sie ein Objekt vom Typ *UMLIncrement* und eine Zeichenkette mit dem Namen eines Musters, der für eine Annotation steht. Sie liefert eine Iteration über Objekte zurück, die als potentielle Trigger für die Annotationsmaschine der gegebenen Annotation dienen. Abbildung 6.6 zeigt die Methode *getTrigger* der Klasse *ReferenceEngine*.

Der Algorithmus beginnt in Aktivität 1 damit, eine leere Menge für die potentiellen Triggerobjekte anzulegen. Dann wird festgestellt, für welche Annotation und damit für welches Prädikat Trigger gesucht werden. In dem Beispiel wird von dem Fall ausgegangen, daß Trigger für das Prädikat *WriteOperation* notwendig sind. Da die Rolle des zu untersuchenden Objektes in der Musterausprägung vor der Erkennung nicht festzustellen ist, wird auf alle Rollen hin untersucht.

In Aktivität 2 wird davon ausgegangen, daß das Objekt die Rolle „*field*“ hat und vom Typ *UMLAttr* ist. In der Musterspezifikation aus Abbildung 6.3 wird nun ein Weg über Kanten und Objekte zu der übergebenen Annotation - hier *WriteOperation* - gesucht. Zufällig wird das Objekt in der Rolle „*field*“ von der *WriteOperation* annotiert. Es ist also ein Trigger für das Prädikat *WriteOperation* und wird der Menge der potentiellen Trigger hinzugefügt.

In Aktivität 3a wird angenommen, daß das zu untersuchende Objekt von Typ *UMLClass* ist. Von dem ASG-Objekt *referredClass* gelangt direkt zum Objekt *referencingField*, welches wiederum von der *WriteOperation* annotiert wird. Die Aktivität 3 ist eine sogenannte „*for each*“-Aktivität. Jedes Objekt *referencingField*, das über die Verknüpfung *attrType* mit dem Objekt *referredClass* verbunden ist (3a), wird in die Menge der potentiellen Trigger aufgenommen (3b). Aktivität 4 been-

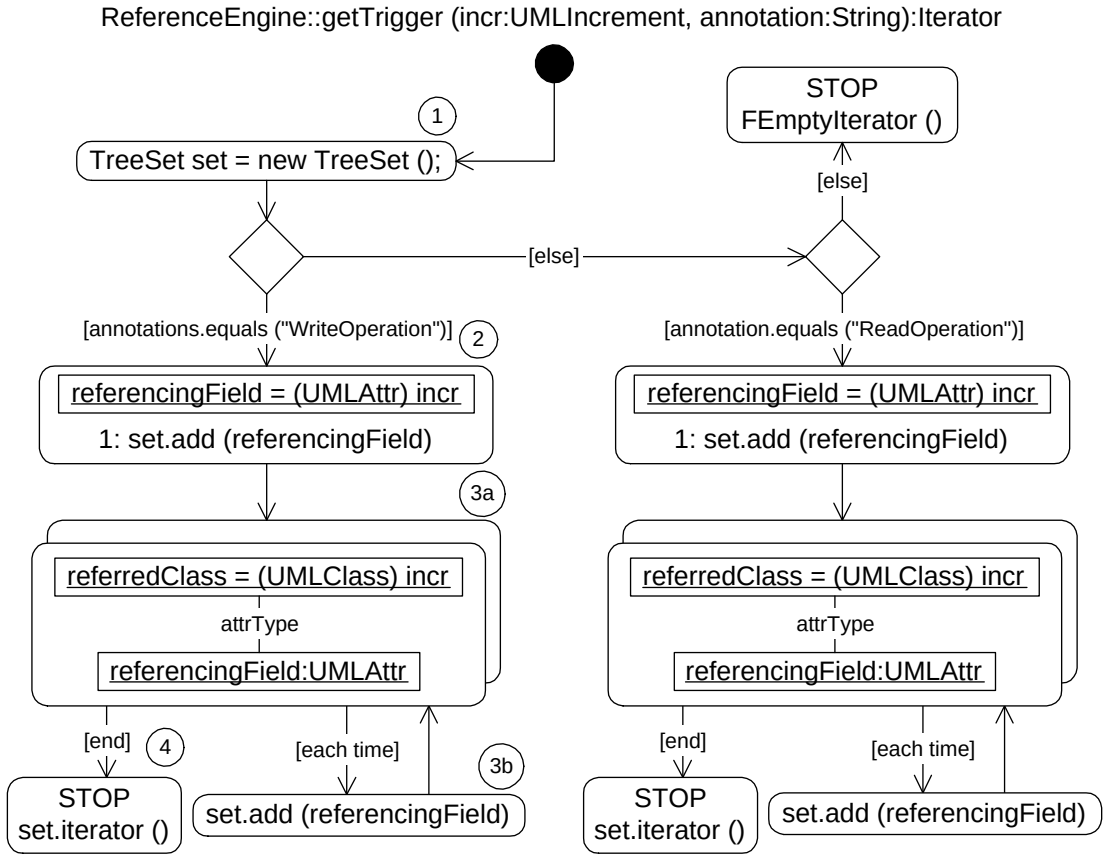

Abbildung 6.6.: Methode *getTrigger* der Klasse *ReferenceEngine*

det schließlich den Methodenaufruf und übergibt einen Iterator über die Menge der potentiellen Trigger.

Für jedes weitere Musterobjekt einer Spezifikation wird eine neue Fallunterscheidung eingeführt, die die als Parameter übergebene Zeichenkette *annotation* mit dem Namen des Musterobjekts vergleicht. Die potentiellen Trigger werden, wie bereits oben am Beispiel beschrieben, gesammelt, indem Wege von dem ebenfalls als Parameter übergebenen Objekt *incr* zu potentiellen Triggerobjekten der Musterobjekte angegeben werden. Pro Weg wird eine Aktivität erzeugt, die jeweils die Zielobjekte des Weges zu der Menge der potentiellen Trigger hinzufügt.

Der zweite Fall ist jedoch in diesem Beispiel für die *ReadOperation* identisch zur *WriteOperation*. Von beiden Musterobjekten wird das Objekt *referencingField* annotiert. Der Weg von dem Objekt *incr* zu den Triggerobjekten ist also in beiden Fällen gleich. Hier liegt eine Symmetrie innerhalb der Spezifikation vor. Weitere Musterobjekte existieren in der Musterspezifikation des Clichés *Reference* nicht, die Methode enthält demnach nur diese beiden Fallunterscheidungen.

6.3. Das Generic Fuzzy Reasoning Net

Das Paket *patDetection.gfrn* enthält die technische Umsetzung der Generic Fuzzy Reasoning Nets. Abbildung 6.7 zeigt das Paket in einem UML-Klassendiagramm. Die Klasse *GFRN* bildet die zentrale Schnittstelle zu diesem Paket. Sie hält Referenzen auf alle Prädikate und alle Implikationen eines Netzes. Über den Namen erhält man von der Klasse *GFRN* aus Zugriff auf ein Prädikat. Die Methode *generateXLMDocument* liefert eine textuelle Repräsentation des GFRN in XML-Notation. Alle Knoten des GFRN implementieren das Interface *GFRNNode*, das eine Methode *generateXMLTag* vorschreibt, die für den jeweiligen Knoten ein textuelle Repräsentation in Form eines XML-Tags zurückliefert. Die XML-Repräsentation dient zur Speicherung des GFRN.

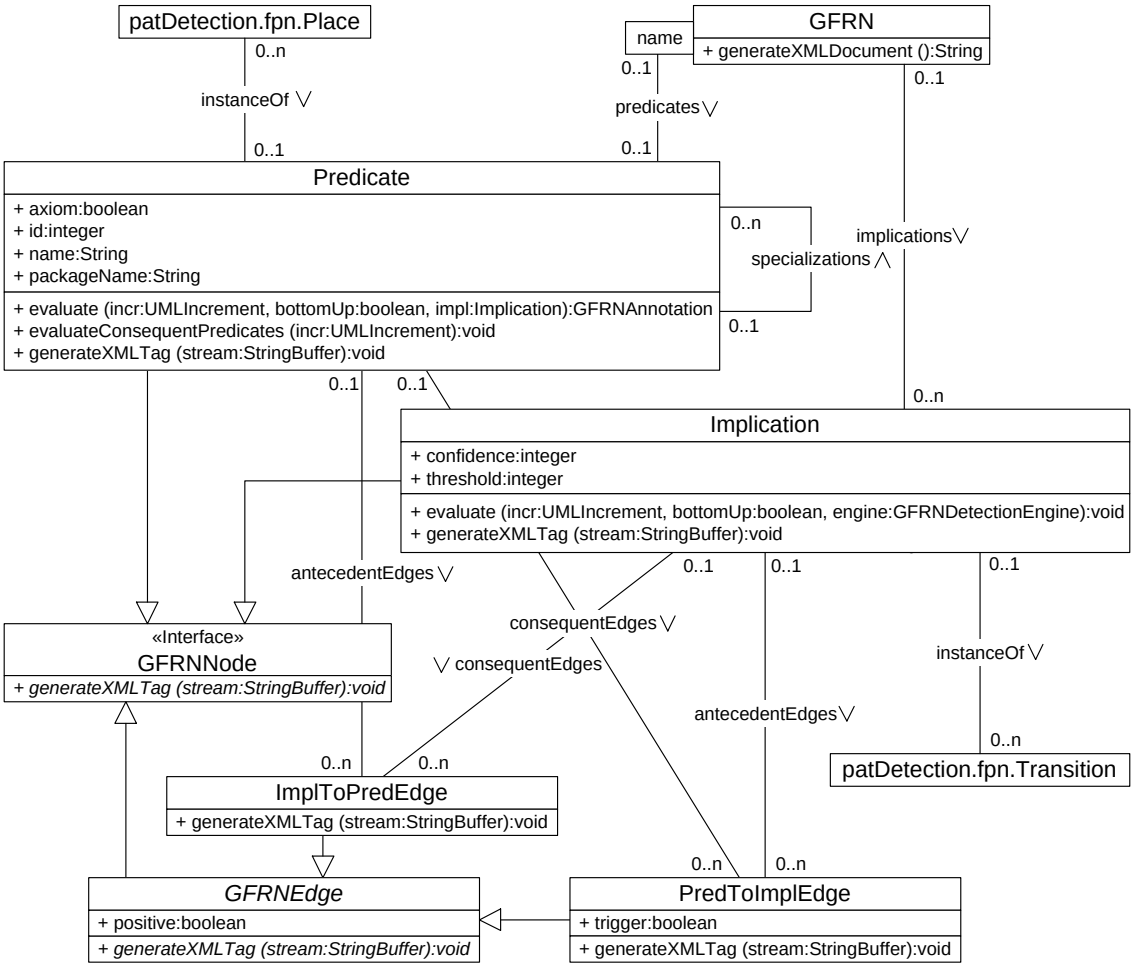

Abbildung 6.7.: Das Paket *patDetection.gfrn*

Die Prädikate eines GFRN werden durch die Klasse *Predicate* modelliert. Sie enthält die Methoden *evaluate* und *evaluateConsequentEdges*, die bereits im Abschnitt 4.3 ausführlich behandelt worden sind. Desweiteren enthält die Klasse einige Attribute. Das boolsche Attribut *axiom* gibt an, ob das Prädikat ein Axiom ist. In *id* wird ein Integer-Wert als eindeutiger Identifizierer gespeichert. Außerdem hat ein Prädikat einen Namen und ein Paketnamen, der im Falle eines Axioms das Paket der ASG-Klasse, die es repräsentiert, und sonst das Paket der zugehörigen Annotationsmaschine angibt.

Die Assoziation *instanceOf* hält Referenzen auf alle Stellen des FPN, die Instanzen des Prädikats sind, also durch das Prädikat erzeugt worden sind. Die Vererbungskanten in einem GFRN, die in Abschnitt 4.3 eingeführt worden sind, werden durch die Assoziation *specializations* modelliert. Gemäß der Einfachvererbung kann ein Prädikat viele Spezialisierungen haben, aber nur eine Generalisierung.

Die Klasse *Implication* repräsentiert die Implikationen eines GFRN. Sie speichert in den Attributen *confidence* und *threshold* den Vertrauens- und den Schwellwert der Musterspezifikation.

Die Kanten eines GFRN werden in der technischen Umsetzung nicht als Assoziation zwischen Prädikaten und Implikationen implementiert, sondern als eigenständige Klassen. Eine gemeinsame abstrakte Oberklasse *GFRNEdge* faßt die Kanten zusammen. Das Attribut *positive* dieser Klasse besagt, ob ein Prädikat positiv oder negativ in eine Implikation eingeht, beziehungsweise ob eine Implikation auf das Prädikat positiv oder negativ schließt. Die Klasse *ImplToPredEdge* bildet die Kanten zwischen Implikation und Prädikat ab. Die Gegenrichtung von einem Prädikat zu einer Implikation modelliert die Klasse *PredToImplEdge*. Sie enthält weitere Attribute. So besagt das Attribut *trigger*, ob das Prädikat die Implikation triggert.

6.4. Das Fuzzy Petrinetz

Das Design der technischen Realisierung eines Fuzzy Petrinetzes ist analog zu dem Design des GFRN aufgebaut. Das UML-Klassendiagramm ist in 6.8 abgebildet. Die Schnittstelle zum FPN bildet hier die Singleton-Klasse *FPN*. Alle Stellen und Transitionen des Netzes sind von dem Singleton-Objekt der Klasse aus zu erreichen. Die Klasse bietet die Factory-Methoden *createPlace*, *createTransition* und *createEdge* zur Erzeugung von Stellen, Transitionen und Kanten zwischen diesen. Außerdem kann von einem Objekt dieser Klasse aus die Evaluierung des FPNs durch die Methode *evaluate* gestartet werden.

Die Stellen eines FPN faßt die Klasse *Place* zusammen. Der Fuzzy Belief der Stelle wird in dem Attribut *posBelief* gespeichert. Der Wert eines vom Benutzer geänderten Fuzzy Beliefs enthält das Attribut *userBelief* und die Information, ob der Benutzer den Fuzzy Belief geändert hat, modelliert das boolsche Attribut *userDefined*. Die Methode *initialize* wird zur Initialisierung des FPNs aufgerufen. Die

Fuzzy Beliefs aller Stellen, die Instanzen eines Axioms sind, werden mit dem Vertrauenswert des Axioms belegt, die Beliefs der anderen Stellen werden auf Null gesetzt. Während der Evaluierung des FPNs werden die Fuzzy Beliefs der Stellen durch Aufruf der Methode *calculateFBMarking* berechnet.

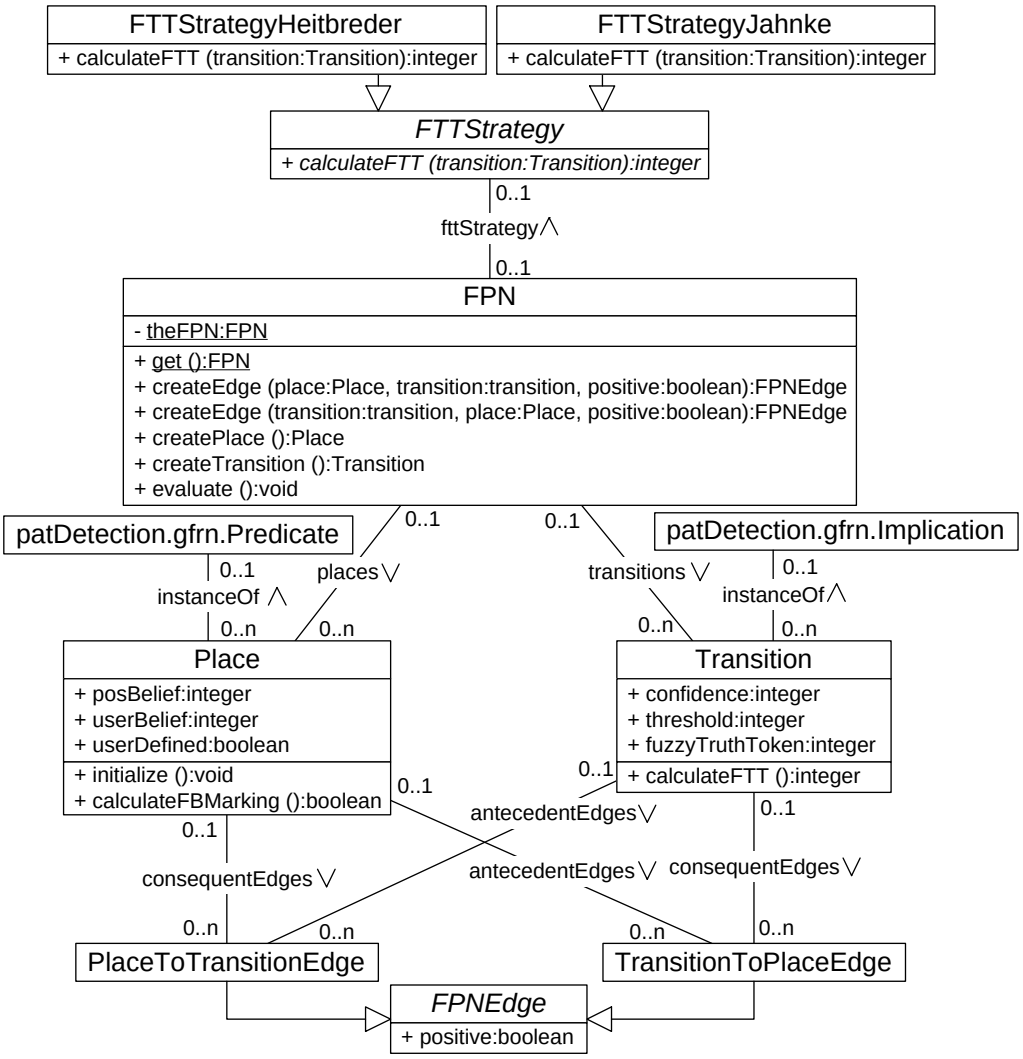

Abbildung 6.8.: Das Paket *patDetection.fpn*

Die Klasse *Transition* ist für alle Transitionen eines FPNs. Die Werte ihrer Attribute *confidence* und *threshold* sind Kopien der gleichnamigen Attribute ihrer Implikations-Instanz und dienen demnach ebenfalls zur Speicherung des Vertrauens- und des Schwellwerts. Das Attribut *fuzzyTruthToken* enthält das Fuzzy Truth Token einer Transition und wird durch die Methode *calculateFTT* berechnet.

Allerdings berechnet die Methode *calculateFTT* den Wert des Fuzzy Truth To-

kens nicht direkt, sondern delegiert die Berechnung an die Methode *calculateFTT* der abstrakten Klasse *FTTStrategy*. Diese Klasse bildet die Oberklasse eines *Strategy*-Design Patterns mit den beiden konkreten Strategien *FTTStrategyHeitbreder* und *FTTStrategyJahnke*. Die beiden Strategien zur Berechnung des Fuzzy Truth Tokens sind den Arbeiten von Melanie Heitbreder [Hei98] und Jens Jahnke [Jah99] entnommen worden.

Die Kanten des FPN sind wie beim Design des GFRN als eigenständige Klassen implementiert worden. Das Attribut *positive* der gemeinsamen abstrakten Oberklasse *FPNEgde* der Kanten ist analog zu dem gleichnamigen Attribut der Klasse *GFRNEdge*.

6.5. Der Inferenzmechanismus

Das Paket *patDetection* (Abbildung 6.9) besteht im wesentlichen aus der Klasse *GFRNInferenceEngine* und zwei Hilfsklassen zum Laden und Speichern des GFRN. Die Klasse *GFRNInferenceEngine* modelliert die Inferenzmaschine, die den Inferenzmechanismus steuert.

Das Attribut *halted* kennzeichnet, daß die Inferenz anzuhalten ist. Der Wert dieses Attributs wird vor jeder neuen Iteration eines Mustererkennungsprozesses überprüft. Bevor jedoch der Inferenzmechanismus angehalten wird, muß die aktuelle Iteration abgeschlossen sein. Die Methode *abort* bricht den Inferenzmechanismus vollständig ab, allerdings auch erst nach Beendigung der aktuellen Iteration. Da die Klasse *GFRNInferenceEngine* von der Klasse *Thread* erbt, muß sie die Methode *run* implementieren, die den Inferenzmechanismus als eigenen Thread startet.

Die Klasse *GFRNInferenceEngine* hält eine Referenz auf das GFRN, um darauf die Mustererkennung auszuführen. Auf alle Annotationsmaschinen, die für die Mustererkennung notwendig sind, kann von *GFRNInferenceEngine* über ihren Namen zugegriffen werden. Zur Auswertung der Analyseergebnisse referenziert die Inferenzmaschine das FPN.

Bevor der Inferenzmechanismus gestartet werden kann, muß das GFRN geladen werden. Diese Aufgabe übernimmt die Singleton-Klasse *GFRNFactory*. Mit Hilfe der Klasse *GFRNSaxHandler* lädt sie das GFRN aus einer XML-Repräsentation. Die Klasse *GFRNFactory* ist außerdem dafür zuständig, aus den Musterspezifikationen das GFRN zu generieren und als textuelle Repräsentation zu speichern.

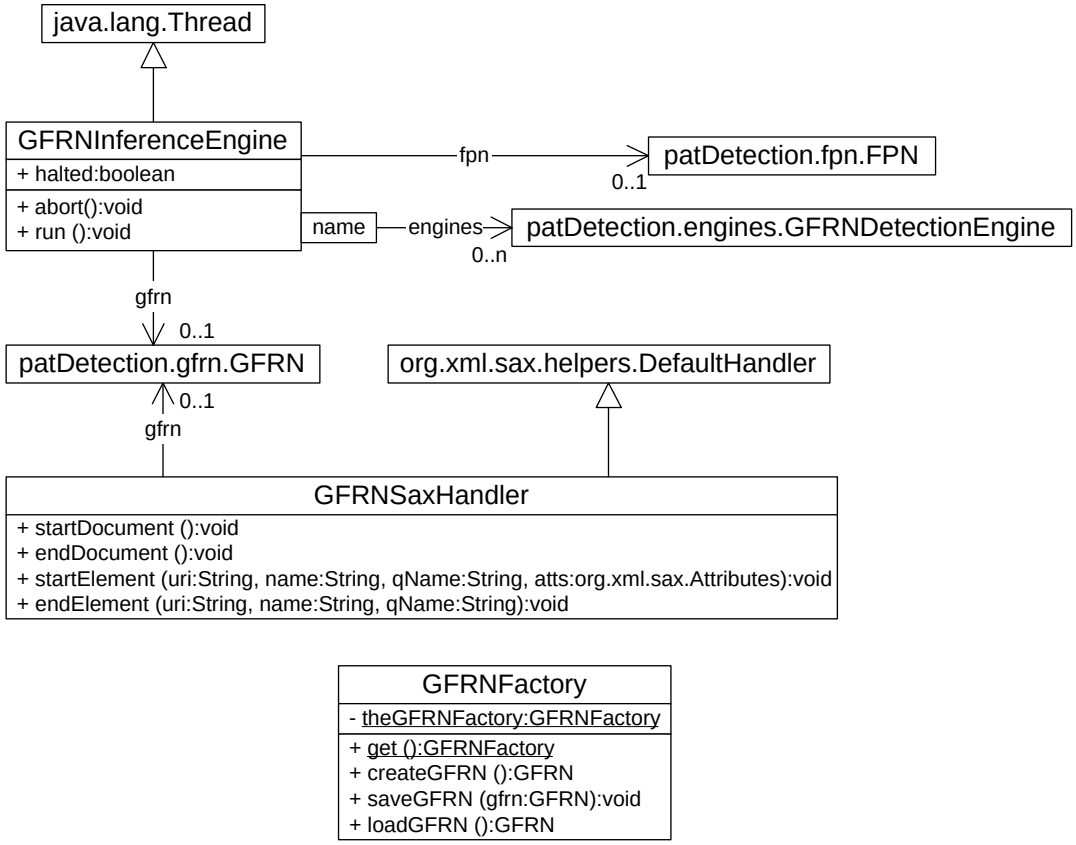

Abbildung 6.9.: Das Paket *patDetection*

7. Beispielsitzung

In den folgenden Abschnitten wird anhand eines Beispiels demonstriert, wie die technische Umsetzung der Mustererkennung in der integrierten Entwicklungsumgebung Fujaba zu benutzen ist. Zunächst wird im Abschnitt 7.1 mit der Spezifikation eines Musters in Fujaba begonnen. Der Abschnitt 7.2 behandelt dann die Integration von neuen Musterspezifikation in das GFRN und die Erzeugung einer Annotationsmaschine. Im Abschnitt 7.3 folgt dann der Erkennungsprozeß und in Abschnitt 7.4 die Interaktion des Benutzers bei der Bewertung der Analyseergebnisse. Die technische Umsetzung ist durch umfangreiche Tests sowohl an Quelltexten, die durch Fujaba erzeugt worden sind, als auch an Quelltexten fremder Softwaresysteme getestet worden. Abschnitt 7.5 gibt einen Überblick darüber.

7.1. Spezifikation

In der Diplomarbeit von Marcus Palasdies [Pal01] ist bereits detailliert die Erzeugung einer Musterspezifikation in Fujaba beschrieben worden, deswegen wird dieser Abschnitt nur kurz auf die Musterspezifikation und Erzeugung von Annotationsmaschinen eingegangen. Abbildung 7.1 zeigt die Spezifikation des Clichés *MultiDelegation*, das bereits in 3.4 vorgestellt worden ist.

In der Abbildung sind außerdem alle Spezifikationen des Kataloges auf der linken Seite in einer Liste zu sehen. Die Liste ist in zwei Hälften aufgeteilt. Der erste Teil der Liste führt die Objektdiagramme der Musterspezifikationen auf. In diesem Teil ist die Musterspezifikation des Clichés *MultiDelegation* ausgewählt. Der zweite Teil enthält die Klassendiagramme sowohl jeder einzelnen Spezifikation, als auch ein Klassendiagramm, das alle Spezifikationen zusammenfaßt. Dieses Klassendiagramm zeigt Abbildung 7.2.

In diesem Katalog-Klassendiagramm sind alle ASG-Metaklassen abgebildet, die in den Musterspezifikationen verwendet werden. Alle relevanten Assoziationen zwischen diesen Klassen sind mit ihren Kardinalitäten und Rollennamen eingetragen. Außerdem sind Klassen mit dem Stereotyp «*Pattern*» zu sehen. Sie repräsentieren die Muster aus dem Katalog. Vererbungen zwischen Mustern sind in diesem Diagramm zu spezifizieren. Die ASG-Metaklassen, die von einem Muster annotiert werden, sind mit der Musterklasse über eine Assoziation verbunden.

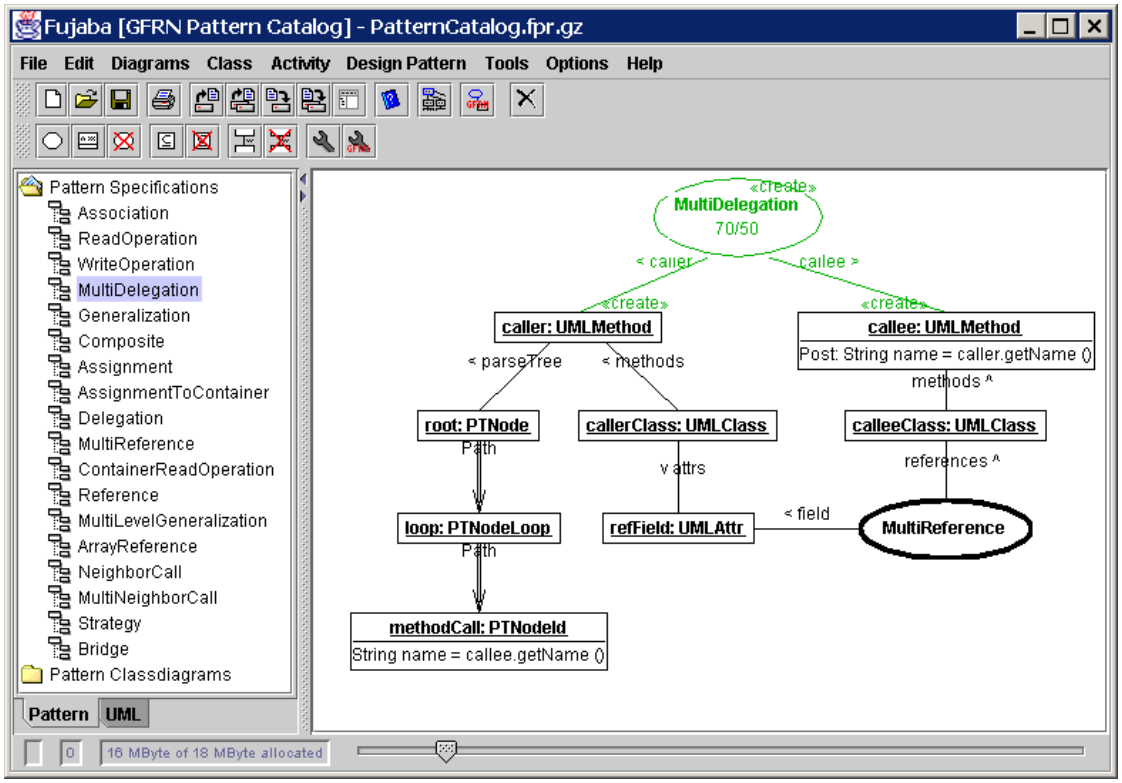

Abbildung 7.1.: Spezifikation des Clichés *MultiDelegation* in FUJABA

Wenn die Spezifikation eines neuen Musters beziehungsweise die Änderung einer Spezifikation abgeschlossen ist, kann die Annotationsmaschine für dieses Muster generiert werden. Abbildung 7.3 zeigt das Menü *Design Pattern*, in dem alle notwendigen Operationen zu finden sind. Die Generierung einer Annotationsmaschine geschieht in zwei Schritten. Zunächst werden mit Hilfe des Menüeintrags *Create GFRN Pattern Recognition Engine* ein UML-Klassendiagramm und SDM-Diagramme zur Definition der in 6.2.2 vorgestellten Methoden erzeugt. Anschließend werden die generierten Klassen in Java-Quelltext exportiert. Diese Klassen können dann übersetzt und in dem Paket *patDetection.engines* abgelegt werden.

7.2. Integration einer Musterspezifikation

Das Generieren und Übersetzen einer Annotationsmaschine reicht nicht aus, um sie in den Inferenzprozeß zu integrieren. Dazu muß erst noch ein neues GFRN erzeugt werden. Der Musterkatalog muß zu diesem Zweck geladen sein, da die Abhängigkeiten zwischen allen Musterspezifikationen in das GFRN einfließen. Alle Musterspezifikationen des Kataloges werden als Prädikate in das GFRN aufgenommen. Die

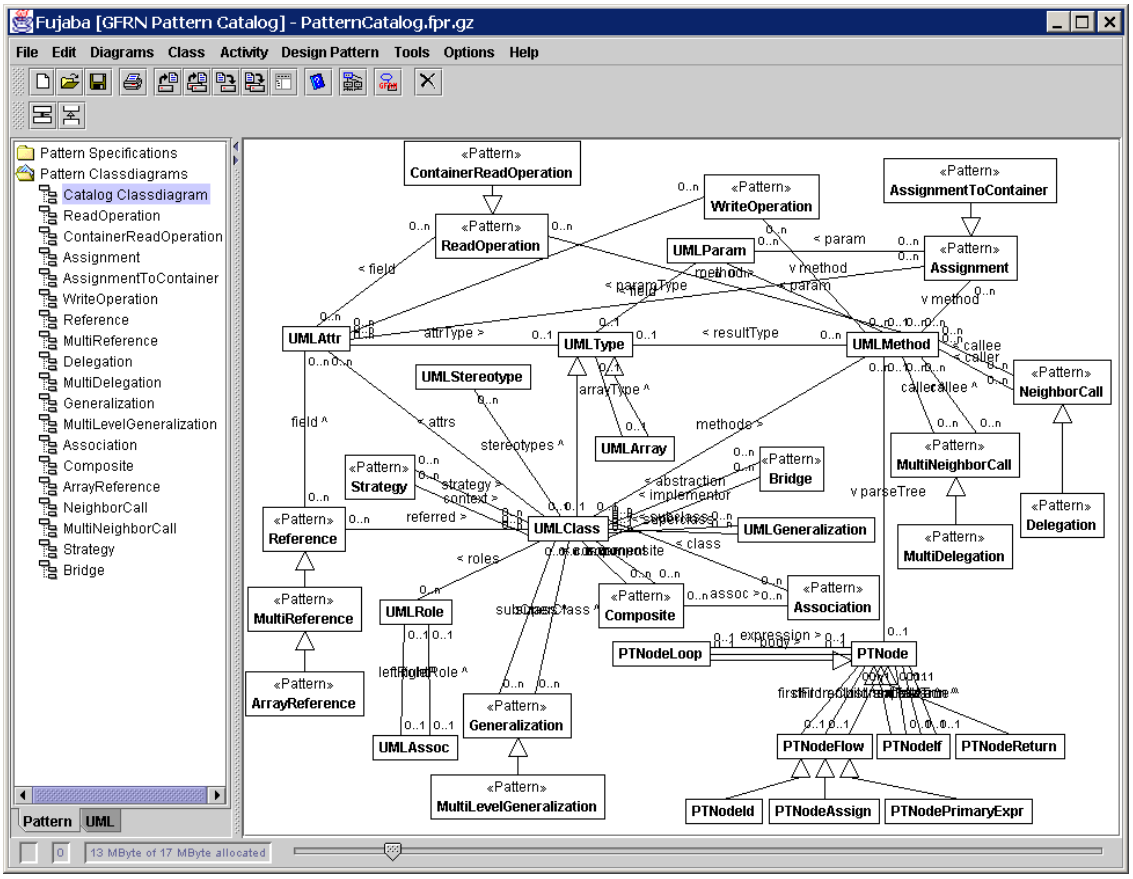

Abbildung 7.2.: Der gesamte Musterkatalog im Klassendiagramm

Abhängigkeiten zwischen den Musterspezifikationen werden auf Implikationen abgebildet. Vererbungen unter den Mustern werden durch Vererbungskanten im GFRN modelliert.

Es besteht durch diese Architektur die Möglichkeit, verschiedene GFRNs für den Inferenzmechanismus zu erzeugen. Auch aus Teilkatalogen lassen sich GFRNs produzieren, sofern die Abhängigkeiten zwischen den Mustern im GFRN in sich abgeschlossen sind, und keine Inkonsistenzen entstehen, weil benötigte Muster fehlen. Auch vollkommen unterschiedliche Kataloge können verwendet werden. Durch Austausch des GFRNs arbeitet der Inferenzmechanismus dann auf einem anderen Katalog.

Zur Erzeugung eines GFRN wird der Menüpunkt *Build And Save GFRN* (Abbildung 7.3) ausgewählt. Es wird ein neues GFRN konstruiert und als textuelle Repräsentation in XML-Notation in der Datei *GFRN.xml* im Paket *patDetection* abgelegt.

Vor jeder Inferenz wird diese Datei geladen und das GFRN daraus erzeugt.

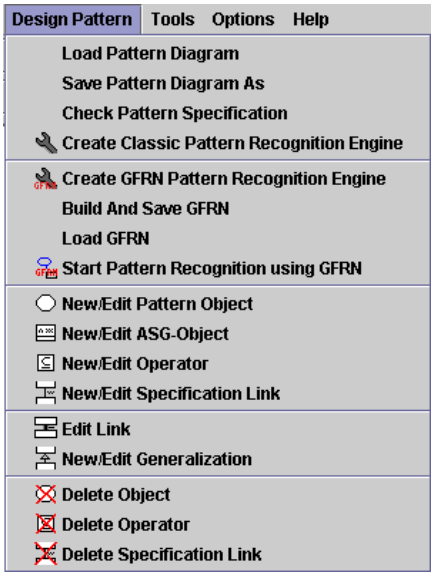

Abbildung 7.3.: Das Menü *Design Pattern* in FUJABA

So muß der Musterkatalog nicht geladen werden. Aus dem GFRN lassen sich die benötigten Annotationsmaschinen ablesen, die durch Instantiierung der Annotationsmaschinen in das System zur Mustererkennung eingebunden werden.

7.3. Der Erkennungsprozeß

Als Beispiel wird für den Erkennungsprozeß eine Implementierung des *Composite*-Design Patterns in der Programmiersprache Java verwendet. Im Unterschied zu dem klassischen Beispiel aus [GHJV95] ist in der Vererbungshierarchie zwischen der *Component*-Klasse und der *Composite*-Klasse eine weitere Klasse *IntermediateClass* eingefügt worden. In den Abbildungen 7.4 bis 7.6 sind Ausschnitte aus den Quelltexten dieser Klassen zu sehen.

Bevor die Mustererkennung gestartet werden kann, müssen zunächst die zu untersuchenden Quelltexte in FUJABA eingelesen werden. Dies geschieht über die Import-Funktion. Nach dem Parsen der Quelltexte wird ein Klassendiagramm wie in Abbildung 7.7 angezeigt. Alle Informationen, die durch einfaches Parsen den Quelltexten entnommen werden können, sind in dem Klassendiagramm dargestellt. Dazu gehören die Klassen mit ihren Attributen und Methoden. In Abbildung 7.7 sind die Klassenpiktogramme eingeklappt, das heißt, Attribute und Methoden der Klassen werden nicht angezeigt. Des weiteren enthält das Klassendiagramm die Vererbungshierarchie der drei Klassen, da diese direkt den Quelltexten zu entnehmen ist. Die Assoziation zwischen den Klassen *Component* und *Composite* kann nicht direkt aus

dem Quelltext geschlossen werden, sondern muß durch den Inferenzprozeß erkannt werden.

```
1:  public abstract class Component {
2:    private Composite parent;
3:    public void setParent (Composite value) {...}
4:    public Composite getParent () {...}
5:    public abstract void delegate ();
6:  }
```

Abbildung 7.4.: Ausschnitt der Klasse *Component*

```
1:  public class IntermediateClass extends Component {
2:    public abstract void delegate ();
3:  }
```

Abbildung 7.5.: Ausschnitt der Klasse *IntermediateClass*

```
 1: public class Composite extends IntermediateClass {
 2:   private HashSet children;
 3:   public void addToChildren (Component value) {...}
 4:   public Iterator iteratorOfChildren () {...}
 5:   public void delegate () {
 6:     Iterator iter = iteratorOfChildren ();
 7:     while (iter.hasNext ()) {
 8:       Component comp = (Component) iter.next ();
 9:       comp.delegate ();
10:     }
11:   }
12: }
```

Abbildung 7.6.: Ausschnitt der Klasse *Composite*

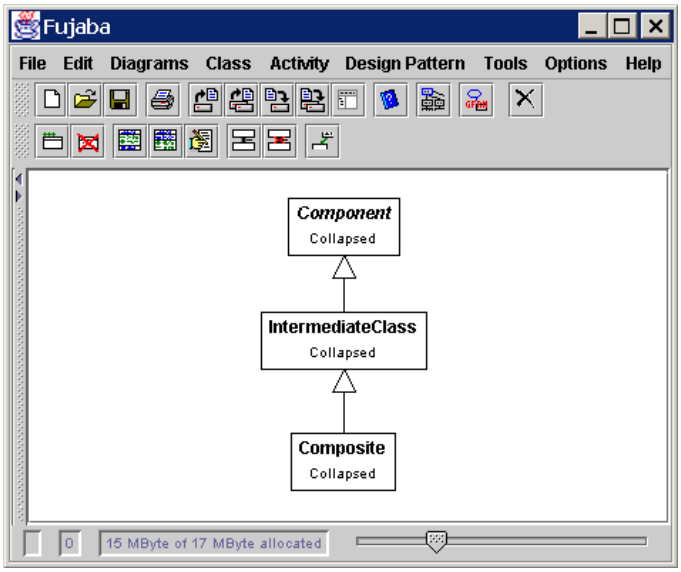

Abbildung 7.7.: Die eingelesenen Klassen im Klassendiagramm

Mit Hilfe des Eintrags *Start Pattern Recognition using GFRN* aus dem Menü *Design Pattern* wird in FUJABA der Inferenzmechanismus gestartet. Vor der eigentlichen Inferenz wird zunächst das GFRN geladen und die Annotationsmaschinen instantiiert. Ein Dialog zeigt den Fortschritt der Inferenz an. Er bietet zusätzlich an, den Erkennungsprozeß anzuhalten oder vollständig zu stoppen. Abbildung 7.8 zeigt diesen Dialog. Die bereits erzeugten Annotationen tragen alle noch einen Fuzzy Belief von 0%, da das zugehörige FPN noch nicht ausgewertet wurde.

Läßt der Benutzer die Erkennung über den Dialog anhalten, so wird zunächst die aktuelle Iteration des Erkennungsprozesses beendet. Dann werden die Bewertungen für die bereits erkannten Annotationen durch das FPN berechnet. Über den Dialog läßt sich die Inferenz wieder fortsetzen.

Abbildung 7.10 zeigt den Zustand des Klassendiagramms nach abgeschlossener Mustererkennung. Alle Annotationen, die gefunden worden sind, werden angezeigt. Wie leicht zu erkennen ist, gibt es keine direkte *Generalization*-Annotation zwischen den Klassen *Component* und *Composite*. Das bedeutet, daß für das *Composite*-Design Pattern die Annotation *MultiLevelGeneralization* benutzt worden ist. Die Redefinition von Mustern durch Vererbung hat also in diesem Beispiel ermöglicht, ein *Composite*-Design Pattern zu erkennen, das von der klassischen Implementierung abweicht.

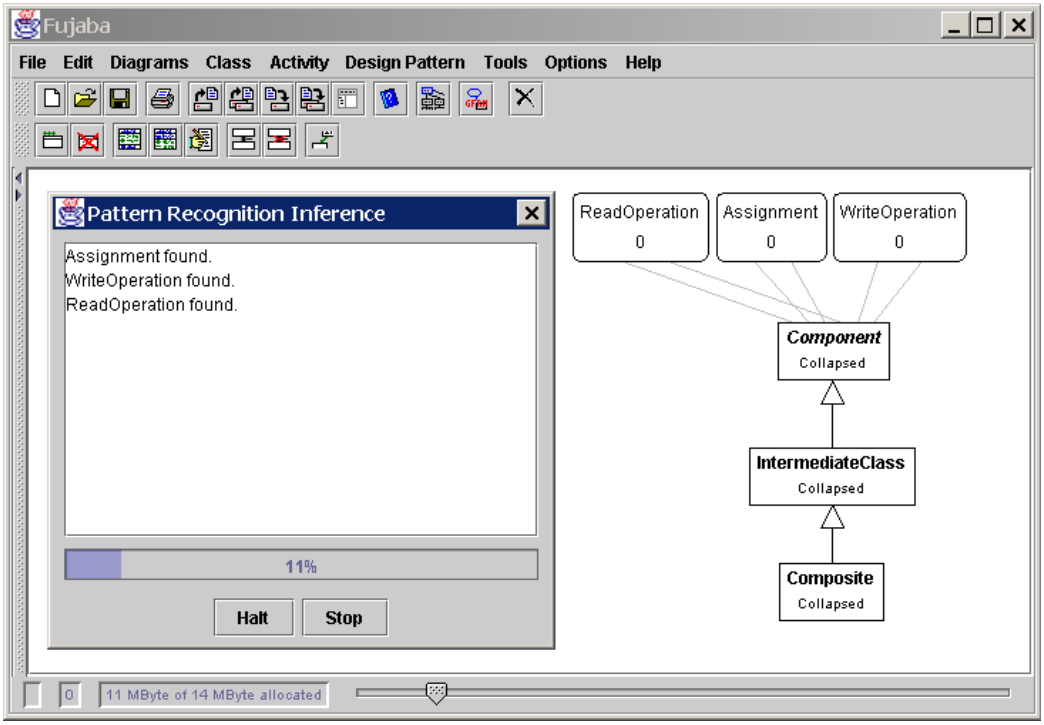

Abbildung 7.8.: Die Mustererkennung in FUJABA

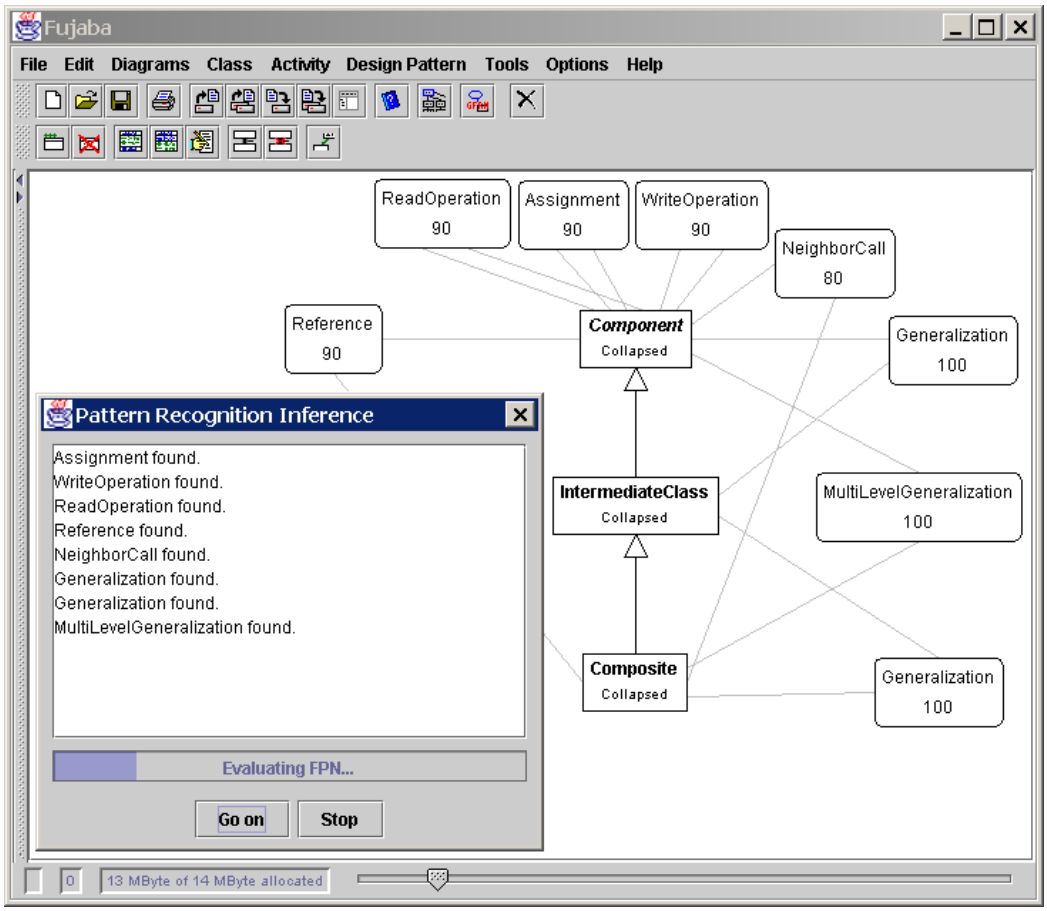

Abbildung 7.9.: Eine angehaltene Mustererkennung

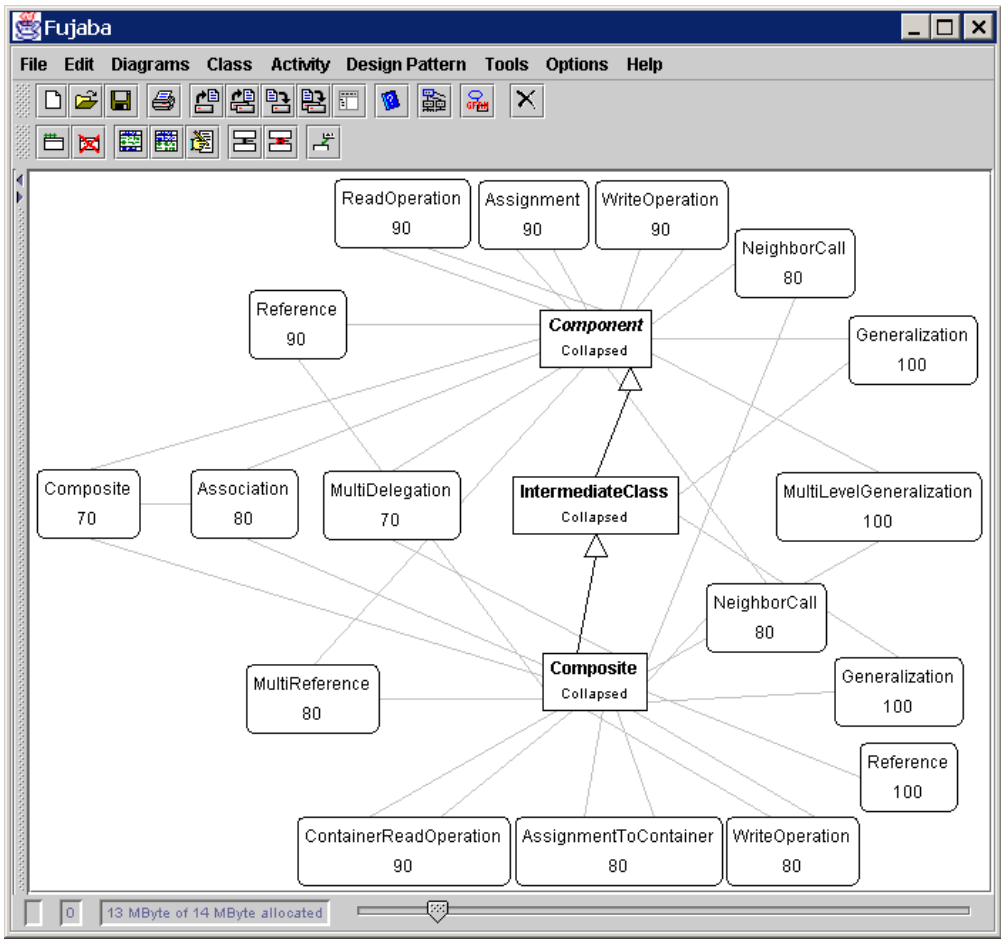

Abbildung 7.10.: Das Ergebnis der Mustererkennung

7.4. Die Interaktion

In der Praxis werden bei komplexen Systemen so viele Annotationen erzeugt, daß der Benutzer die Übersicht verliert, wenn alle angezeigt würden. Meist interessiert sich der Benutzer auch nur für ganz bestimmte Muster wie Design Patterns. Zu diesem Zweck kann der Benutzer in einem Optionen-Dialog die wichtigsten Muster auswählen, deren Annotationen durch FUJABA anzuzeigen sind. Abbildung 7.11 stellt diesen Dialog dar. Desweiteren kann zu jedem Muster ein Schwellwert angegeben werden. Diese Wert hat zur Folge, daß nur Annotationen angezeigt werden, deren Fuzzy Belief oberhalb des Schwellwerts liegt. Alternativ lassen sich auch alle Annotationen ausblenden.

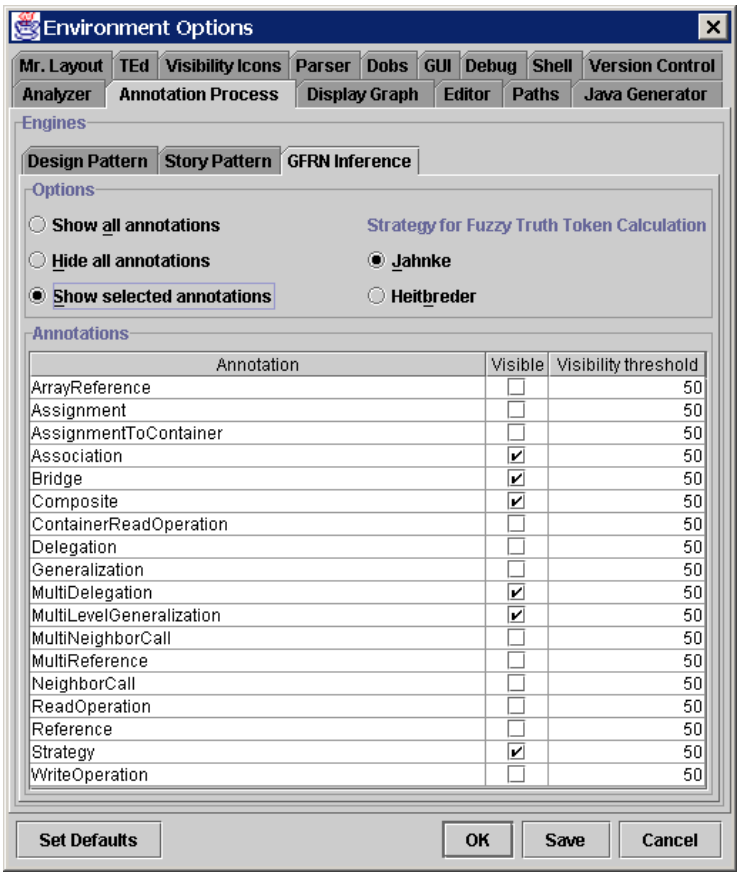

Abbildung 7.11.: Der Optionen-Dialog

In diesem Dialog besteht außerdem die Möglichkeit, die Strategie anzugeben, die angewandt wird, um die Fuzzy Truth Token der Transitionen zu berechnen. Zur Auswahl stehen die beiden Strategien nach Jahnke [Jah99] und Heitbreder [Hei98], die bereits in Abschnitt 6.4 erwähnt worden sind.

Im Folgenden werden mit Hilfe der oben genannten Einstellmöglichkeiten nur die *Composite*-Annotation und die Annotationen angezeigt, die direkt zur *Composite*-Annotation geführt haben. Dieses sind die *Association*, die *MultiDelegation* und die *MultiLevelGeneralization*. Abbildung 7.12 zeigt diese Ansicht des Klassendiagramms.

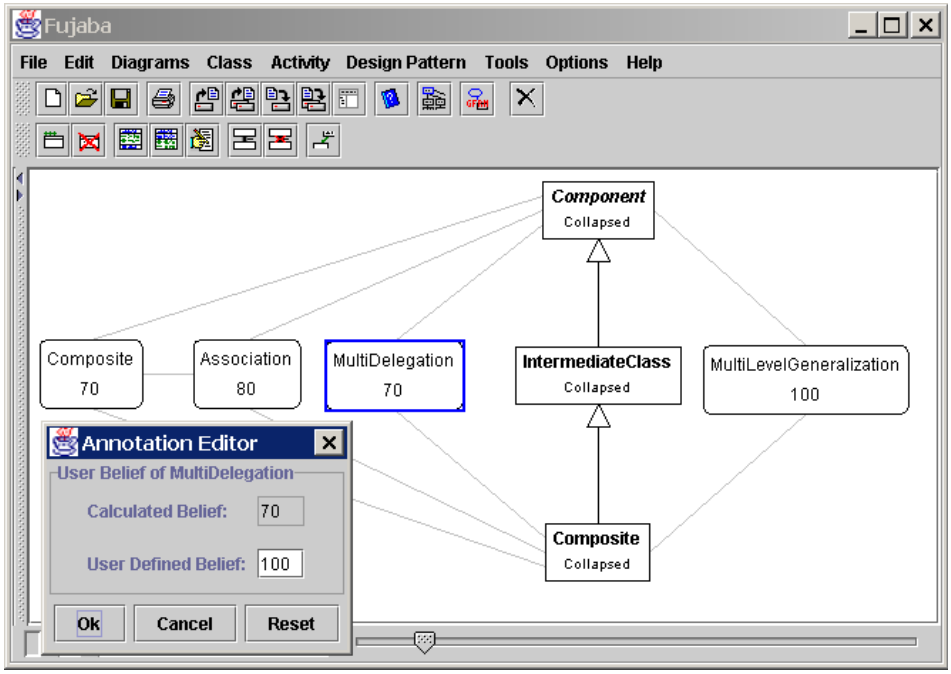

Abbildung 7.12.: Der Änderungsdialog für den Fuzzy Belief

Der in Abschnitt 5.2 beschriebene Eingriff in den Inferenzmechanismus durch den Benutzer ist durch einen Dialog zur Veränderung des Fuzzy Beliefs einer Annotation verwirklicht worden. Dieser Dialog ist ebenfalls in Abbildung 7.12 zu sehen. Aufzurufen ist er über das kontextsensitive Menü einer Annotation.

Angenommen, der Benutzer sieht sich einige der Annotationen genauer an. Er stellt fest, daß es eine *MultiDelegation* gibt, die die Methode *delegate* der Klasse *Composite* als *caller* und die gleichnamige Methode aus der Klasse *Component* als *callee* annotiert[1]. Bei der Analyse dieser Methoden im Quelltext stellt er fest, daß die Methode *delegate* aus 7.6 tatsächlich eine Mehrfachdelegation ist. Er hat nun die Möglichkeit, den Fuzzy Belief der entsprechenden Annotation zu ändern. Da der Benutzer sich sicher ist, daß eine Mehrfachdelegation vorliegt, ändert er den Belief auf 100%.

[1]Die Rollen, unter denen Elemente annotiert werden, sind sichtbar, wenn die Annotationskanten selektiert werden.

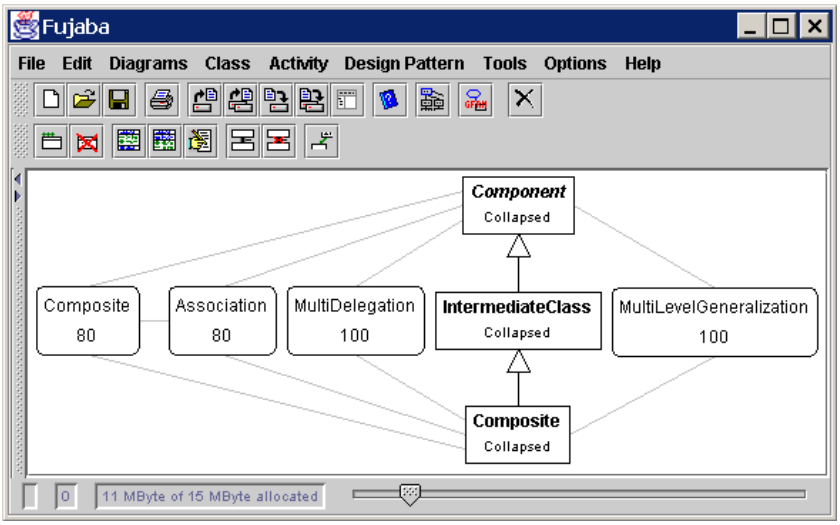

Abbildung 7.13.: Die Auswirkungen der Fuzzy Belief Änderung

In Abbildung 7.13 ist zu sehen, welche Konsequenzen diese Veränderung hat. Vor der Veränderung war der Fuzzy Belief der *MultiDelegation* der limitierende Faktor für den Fuzzy Belief der *Composite*-Annotation. Der Fuzzy Belief der *MultiDelegation* wird erhöht und das FPN neu ausgewertet. Daraufhin steigt der Fuzzy Belief des *Composite* ebenfalls. Der limitierende Fuzzy Belief ist nun der Belief der *Association*.

Dieses Beispiel ist sehr klein und die Konsequenzen der Änderung leicht vorherzusehen. Allerdings sind auch weitreichendere Konsequenzen denkbar, die vom Benutzer nicht mehr vorhersehbar sind. Zu diesem Zweck bietet FUJABA eine Unterstützung bei der Veränderung der Fuzzy Beliefs und eine automatische Reevaluierung des FPN.

7.5. Evaluierung

Zur Evaluierung der Mustererkennung ist ein Katalog mit 15 Clichés und drei Design Patterns aufgebaut worden. Die Clichés sind geeignet, die Spezifikationen von weiteren Design Patterns durch Modularität zu unterstützen. Die Muster des Kataloges sind sowohl an FUJABA, als auch an fremden Softwaresystemen positiv getestet worden.

Diese Softwaresysteme sind das JAVA ABSTRACT WINDOW TOOLKIT (AWT) [AWT] mit einem Umfang von etwa 114.000 Zeilen Quelltext und die JAVA GENERIC LIBRARY (JGL) [JGL] mit etwa 36.000 Zeilen Quelltext. In [NSW+01] sind die Ergebnisse dieser Evaluierung zu finden. Tabelle 7.1 ist aus [NSW+01] übernommen worden.

Source code	KLOC	Patterns	Complete analysis	First GoF Pattern
AWT (part)	8,7	JPC1	41 sec.	0,5 sec.
AWT (part)	8,7	JPC2	100 sec.	56 sec.
AWT (all)	114,4	JPC2	1307 sec.	13 sec.
JGL (all)	36,5	JPC1	43 sec.	3,5 sec.
JGL (all)	36,5	JPC2	73 sec.	24 sec.

Tabelle 7.1.: Analyseergebnisse der Evaluierung aus [NSW+01]

Die Tabelle enthält in der ersten Spalte das analysierte Softwaresystem. Spalte 2 gibt den Umfang des Systems in tausend Zeilen Quelltext an. Die dritte Spalte gibt Auskunft über den verwendeten Musterkatalog. Die letzten beiden Spalten enthalten die Dauer der kompletten Analyse und die Zeit bis zur ersten Erkennung eines Design Patterns.

Es sind zwei verschiedene Musterkataloge zur Evaluierung verwendet worden. Der erste Katalog JPC1 entspricht teilweise dem im Anhang beigefügten Katalog. Allerdings sind Clichés, die Methodenrümpfe analysieren, herausgenommen beziehungsweise modifiziert worden. Es sind nur Muster enthalten, die ausschließlich strukturelle Informationen der Quelltexte analysieren. Aus dem Cliché *MultiDelegation* aus Abbildung 3.14 ist beispielsweise die Analyse des Methodenrumpfes entfernt worden, so daß nur noch auf Namensgleichheit zweier Methoden unterschiedlicher Klassen geprüft wird. Dieser Katalog wurde dem Ansatz von Krämer und Prechelt [KP96] nachempfunden, der ebenfalls nur strukturelle Informationen aus den C++-Header Dateien extrahiert und analysiert. Der zweite Katalog JPC2 entspricht exakt dem im Anhang beigefügten Musterkatalog. Hier werden Analysen inklusive der Methodenrümpfe ausgeführt.

Bei der Verwendung des ersten Katalogs sind erwartungsgemäß sehr viele Muster falsch erkannt worden. Die Untersuchung hat gezeigt, daß der Verzicht auf die Analyse von Methodenrümpfen zwar einen Geschindigkeitsvorteil bringt, sich allerdings negativ auf die Präzision der Mustererkennung auswirkt. Im Vergleich dazu ist die Analyse der JGL bei Verwendung des zweiten Katalogs etwa 70% langsamer, jedoch ist die Präzision sehr viel höher. Der Benutzer spart sehr viel mehr Zeit, wenn er nicht erst sehr viele falsche Ergebnisse auszusortieren hat. Der menschliche Faktor ist bei der Auswertung der Analyseergebnisse also schwerer zu gewichten, als die Laufzeit des Algorithmus.

Der Zeitunterschied von 70% relativiert sich noch stärker, wenn man die absoluten Laufzeiten des Inferenzmechanismus betrachtet. Die vollständige Abarbeitung der Mustererkennung benötigte beim AWT mit 114.000 Zeilen Quelltext weniger als 22 Minuten. Bei einen kleineren Teil des AWT von etwa 8700 Zeilen Quelltext, der aus den wesentlichen Klassen besteht, dauerte die Inferenz nur 100 Sekunden bei

Verwendung des zweiten Katalogs.

Als weiteres Ergebnis der Evaluierung ist festzuhalten, daß die Skalierbarkeit der Mustererkennung erreicht wurde. Im Gegensatz zu dem NP-vollständigen Ansatz von Linda Wills [Wil96] ist der Zeitaufwand dieses Ansatzes auch bei großen Systemen von 100.000 Zeilen Quelltext Umfang vertretbar.

8. Zusammenfassung und Ausblick

8.1. Zusammenfassung

In dieser Arbeit ist die Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets in dem integrierten Entwicklungswerkzeug Fujaba verwirklicht worden. Die Mustererkennung unterstützt den Softwareentwickler beim Reverse Engineering komplexer Systeme. Sie hilft dabei, die Verwendung von Design Patterns in der Implementierung dieser Systeme zu identifizieren. Der Benutzer erhält dadurch einen Überblick über das System und kann Rückschlüsse auf Teile des Systems ziehen, ohne Details untersuchen zu müssen.

Um die Mustererkennung in die Praxis umzusetzen, wurden zwei Ansätze weiterentwickelt beziehungsweise angepasst. Zunächst ist die Musterspezifikationssprache von Marcus Palasdies [Pal01] um Sprachelemente erweitert worden. Einige der Anforderungen aus Abschnitt 1.3 sind bereits durch die Musterspezifikationssprache erfüllt gewesen. Dazu gehört die Modularität und Anpassungsfähigkeit der Spezifikationen. Außerdem unterstützt die Vererbung von Mustern die Abdeckung vieler Varianten von Musterausprägungen durch eine einzige Spezifikation.

Einige neue Sprachelemente sind in dieser Arbeit hinzugefügt worden. So erleichtern Pfadausdrücke die Auswertung von Methodenrümpfe. Dynamische Anteile in Mustern sind nun einfacher zu spezifizieren. Sie erhöhen die Variantenvielfalt einer Musterspezifikation und die Präzision der Mustererkennung. Die Auswertung der Methodenrümpfe hat zur Folge, daß weniger Musterausprägungen durch den Inferenzmechanismus falsch erkannt werden.

Desweiteren sind Sprachelemente eingeführt worden, die eine Verbindung der Musterspezifikationssprache zu den GFRNs schaffen. Dazu gehören die Trigger, die gezielte Schlußfolgerungsketten innerhalb der GFRNs vorgeben. Aber auch die Vertrauens- und Schwellwerte, die die Handhabung von Unschärfe und unsicheren Wissens ermöglichen.

Die GFRNs von Jens Jahnke [Jah99] genügten den Anforderungen, die an die Mustererkennung gestellt worden sind, ebenfalls nicht vollständig. So ist die Syntax der GFRNs durch Einschränkung auf nur zwei Prädikattypen und Hinzunahme von Vererbungskanten zwischen Prädikaten geändert worden.

Um die Skalierbarkeit zu verbessern, sind auch Veränderungen an der Semantik

der GFRNs vorgenommen worden. Der dritte Prädikattyp, der aus den ursprünglichen GFRNs entfernt worden ist, die aufgeschobenen Axiome, kennzeichnen den Top Down-Anteil des GFRNs. Allerdings waren Top Down-Analysen nur auf Basisfakten ausführbar. Durch die Verwendung von Graphtransformationen konnte diese Einschränkung umgangen werden, so daß innerhalb des GFRNs nun beliebig durch den Inferenzmechanismus sowohl Bottom Up-, als auch Top Down-Analysen durchgeführt werden können.

Die Güte der Analyseergebnisse wird auch weiterhin durch die FPNs auf Basis der Vertrauens- und Schwellwerte berechnet. Somit ist die Handhabung von Unschärfe sichergestellt.

In dieser Arbeit wurde der Schwerpunkt der Mustererkennung auf die Verwendung von Design Patterns in Softwareimplementierungen gelegt. Vorstellbar ist jedoch auch eine Anwendung der Mustererkennung in anderen Domänen. Die Verwendung von Graphen als Repräsentation der initialen Daten läßt Freiraum für andere Anwendungen. Jegliche Daten, die in Graphen umgesetzt werden können, lassen sich mit diesem Ansatz auf Muster hin untersuchen. Auch die Musterspezifikationssprache ist in diese Richtung offen ausgelegt. Die Analyse von Quelltexten ist nur eine Anwendung unter vielen.

8.2. Ausblick

Es existieren verschiedene Ansätze zur Erweiterung der Mustererkennung. So ist zu untersuchen, inwieweit die in dieser Arbeit eingeführten optionalen Objekte der Musterspezifikationssprache zur Bewertung der Analyseergebnisse verwendet werden können. Optionale Objekte können die Präzision von Musterspezifikationen verbessern, ohne die Spezifikation von den Objekten abhängig zu machen. Die Existenz optionaler Elemente in einer Musterausprägung sollte also Einfluß auf die Bewertung der Güte haben, indem sie diese untermauert.

Die ursprünglichen GFRNs bieten die Möglichkeit von negativen Schlußfolgerungen. Prädikate können die Ungültigkeit von anderen Prädikaten implizieren. Das führt dazu, daß sowohl positive, als auch negative Fuzzy Beliefs für Stellen im FPN berechnet werden. Ein Faktum besitzt dann eine positive Bewertung, die seine Gültigkeit belegt, und eine negative Bewertung, die den Grad seiner Ungültigkeit festlegt. Desweiteren kann auch die Ungültigkeit eines Prädikates zu Implikationen herangezogen werden.

Als Beispiel für negative Schlußfolgerungen können die Clichés *MultiDelegation* und *WriteOperation* genannt werden. Wird eine Methode als *WriteOperation* erkannt, so deutet das darauf hin, daß keine *MultiDelegation* vorliegt oder umgekehrt.

Negative Schlußfolgerungen und negative Prädikate sind eventuell Hilfsmittel, um die Mustererkennung auf sogenannte *Pattern Languages* auszuweiten. In den *Pattern Languages* werden nicht nur einzelne Muster spezifiziert, sondern auch Zu-

sammenhänge zwischen den Mustern definiert. Die Existenz einer Musterausprägung kann zum Beispiel bedeuten, daß keine Ausprägung eines bestimmten zweiten Musters in dem gleichen Kontext vorhanden sein kann. Die technischen Voraussetzungen zur Realisierung der genannten Erweiterungen sind bereits in der Architektur der GFRNs und FPNs enthalten.

Die Mustererkennung in ihrer bisherigen Form unterstützt nur die Erkennung von Designteilen wie Design Patterns. Die nächste Abstraktionsstufe ist die Rückgewinnung gesamter Architekturen, also Informationen über Zusammenhänge von Designteilen und eventuelle Schlußfolgerungen. Die Anwendung von Pattern Languages kann in diesem Zusammenhang hilfreich sein.

In der Quälitätssicherung kann die Erkennung von Design Patterns in Quelltexten eine Rolle spielen. So deutet die Verwendung von Design Patterns in der Regel auf ein gutes Gesamtdesign hin. Allerdings kann die Verwendung einzelner Design Patterns auch ganz im Gegenteil zu schlechter Qualität führen. Im Zusammenhang mit der Programmierung von Java Smart Cards beispielsweise, die nur einen eng begrenzten Speicher besitzen, kann ein *Composite*-Pattern zu Problemen führen. Die aus diesem Muster gebildeten Datenstrukturen können dynamisch wachsen, was eventuell zu einem Speicherüberlauf führt.

A. Katalog der Musterspezifikationen

A.1. Clichés

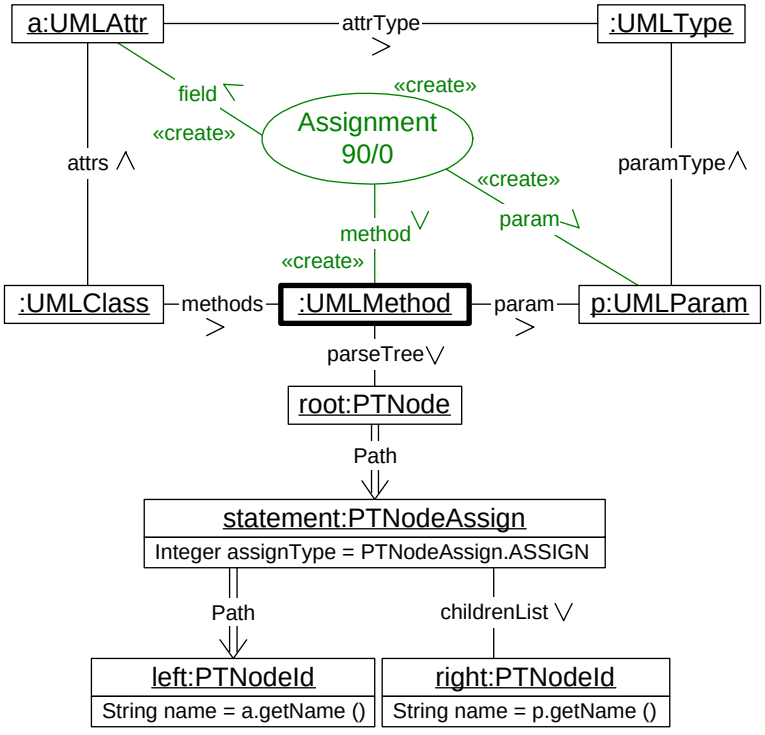

Abbildung A.1.: *Assignment*

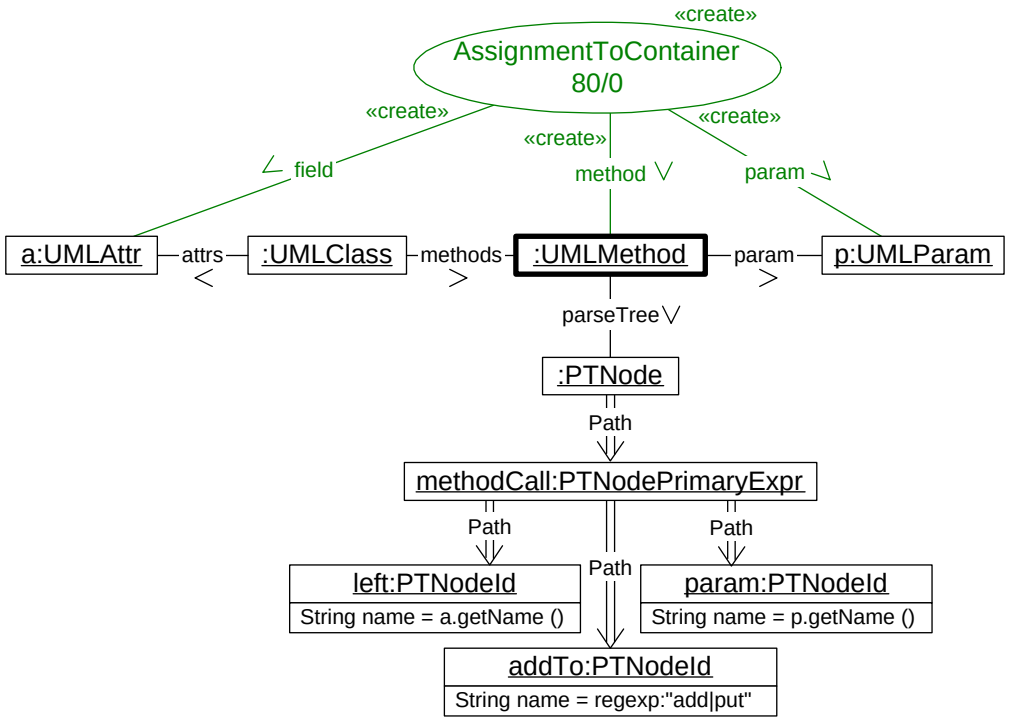

Abbildung A.2.: *AssignmentToContainer*

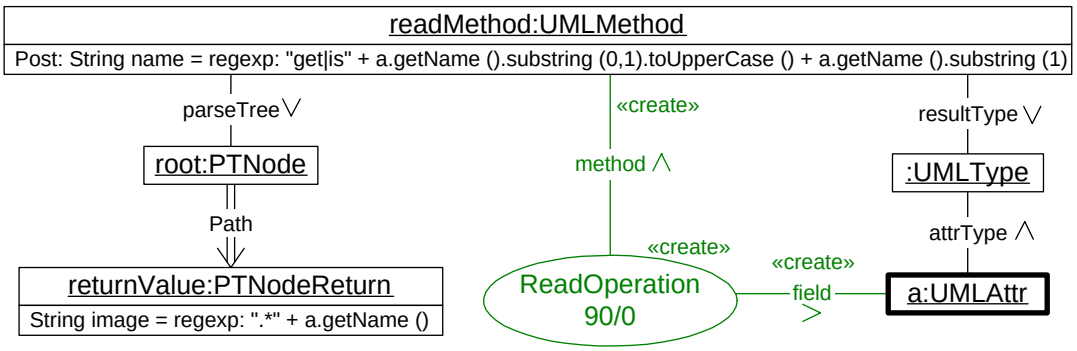

Abbildung A.3.: *ReadOperation*

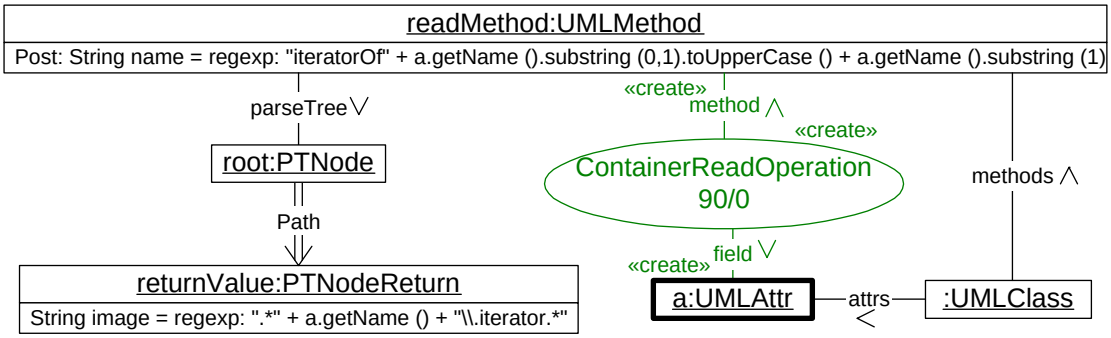

Abbildung A.4.: *ContainerReadOperation*

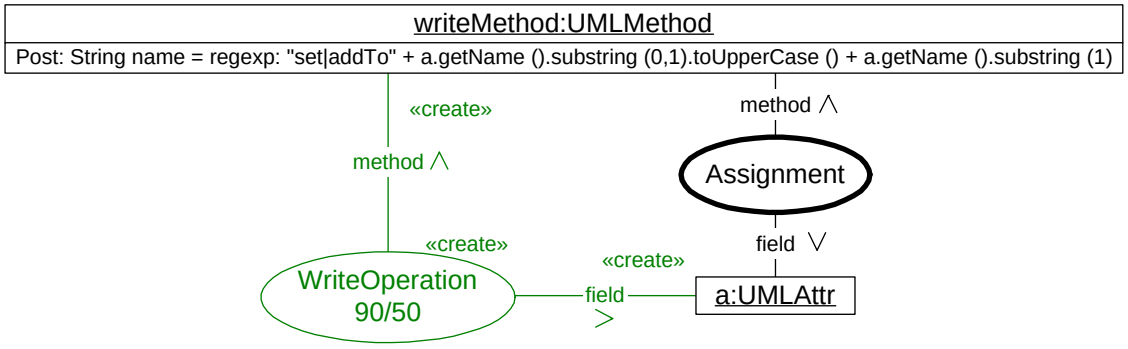

Abbildung A.5.: *WriteOperation*

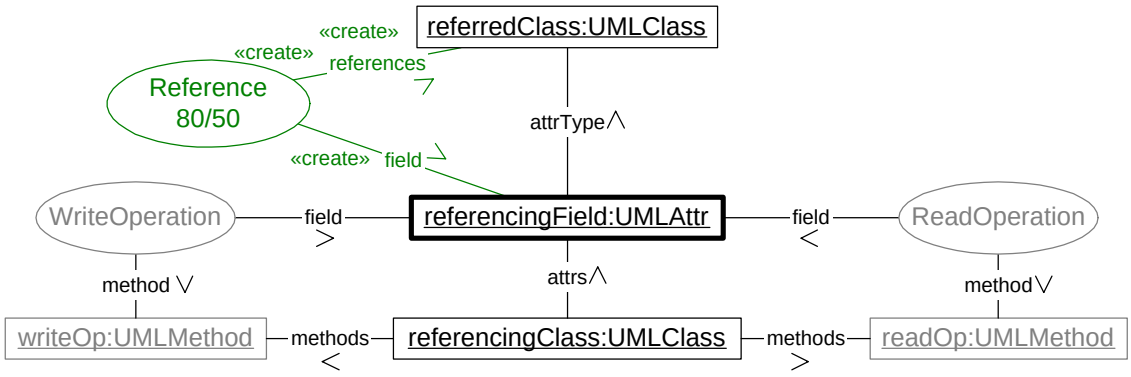

Abbildung A.6.: *Reference*

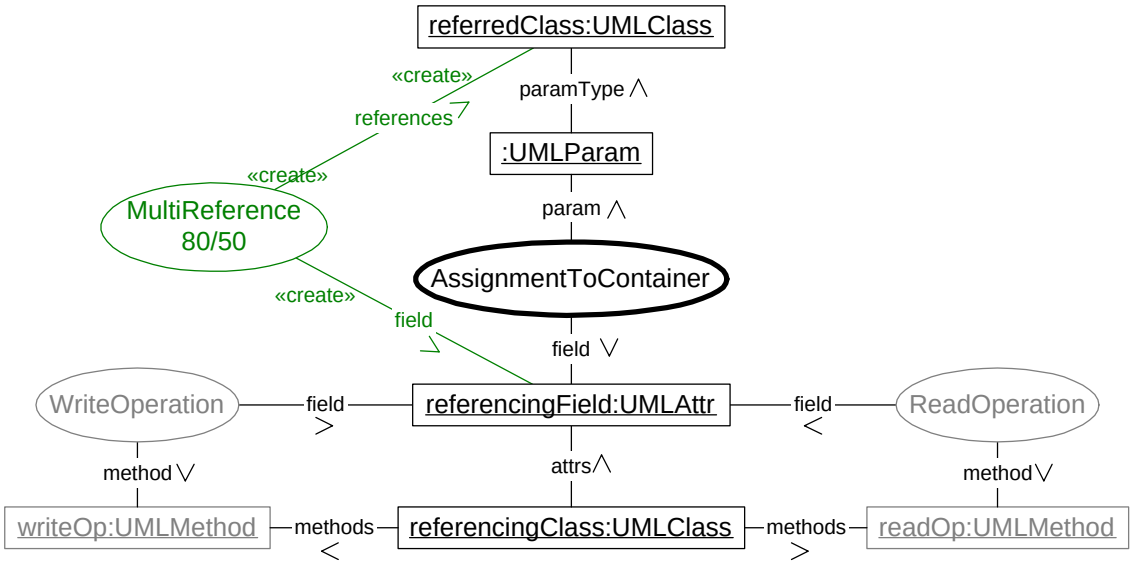

Abbildung A.7.: *MultiReference*

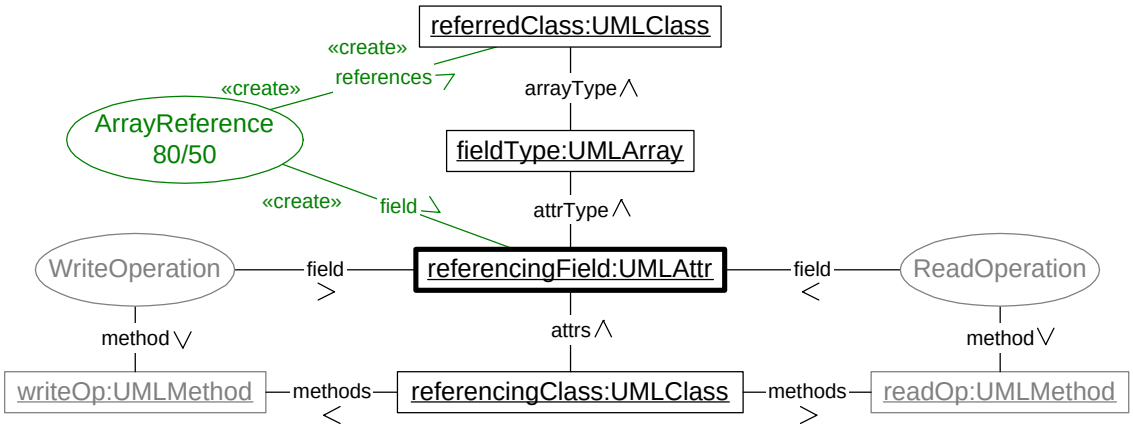

Abbildung A.8.: *ArrayReference*

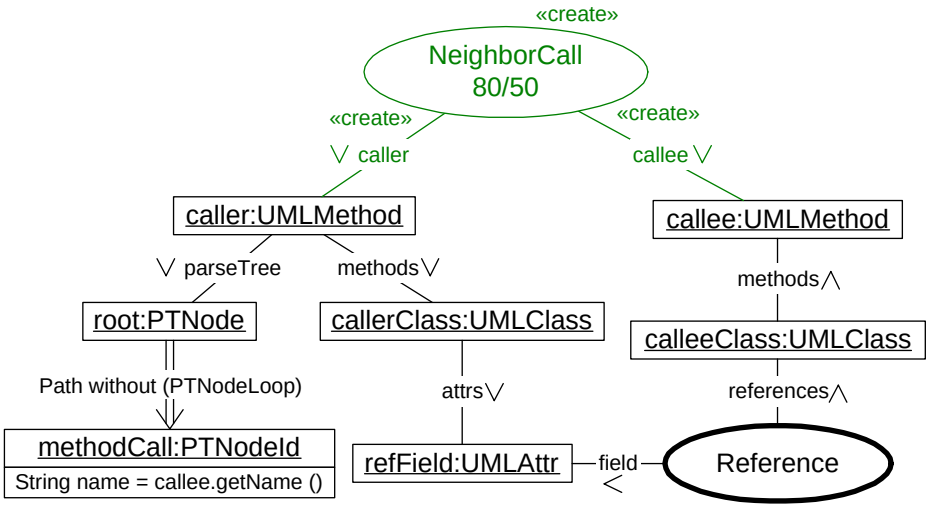

Abbildung A.9.: *NeighborCall*

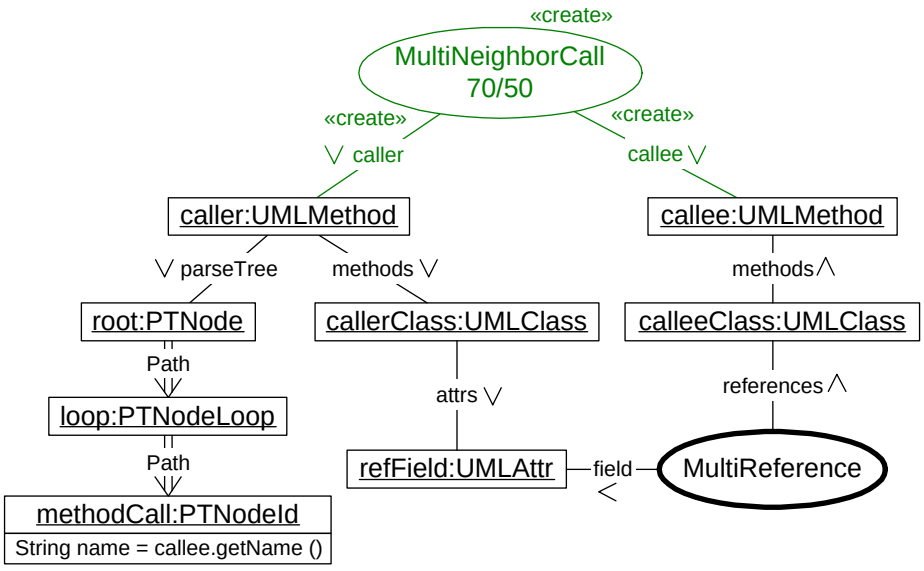

Abbildung A.10.: *MultiNeighborCall*

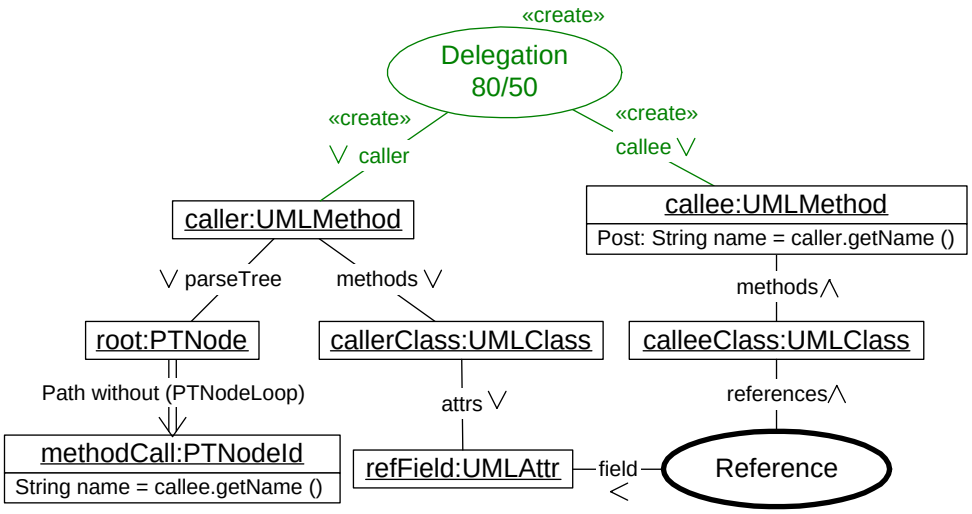

Abbildung A.11.: *Delegation*

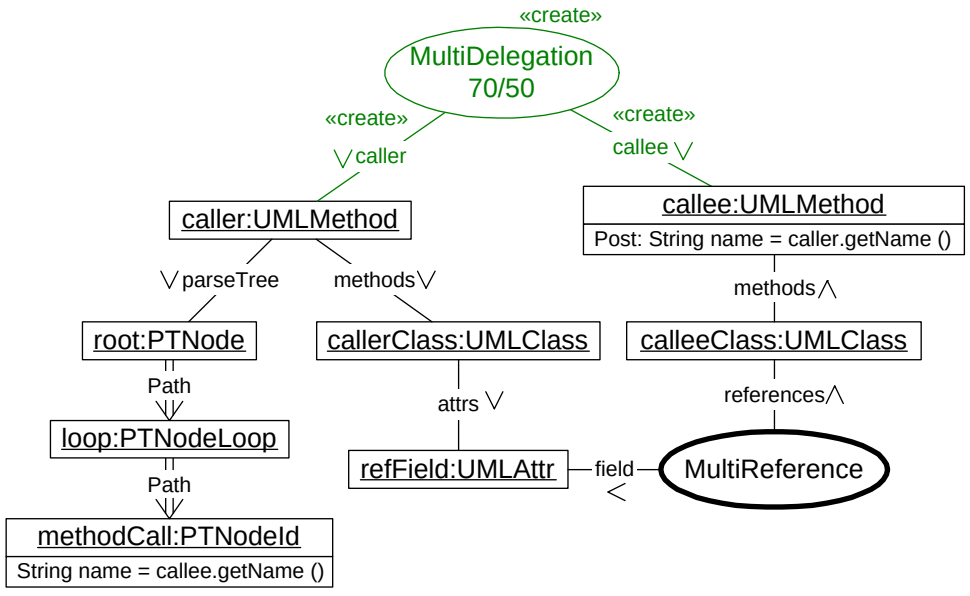

Abbildung A.12.: *MultiDelegation*

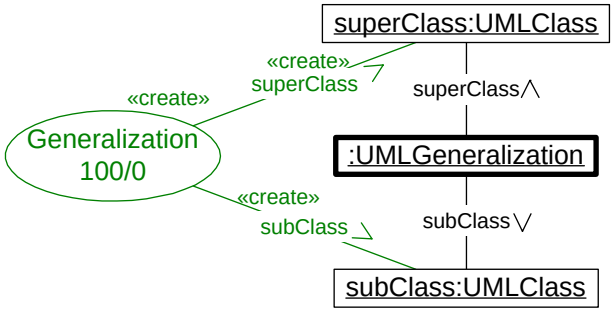

Abbildung A.13.: *Generalization*

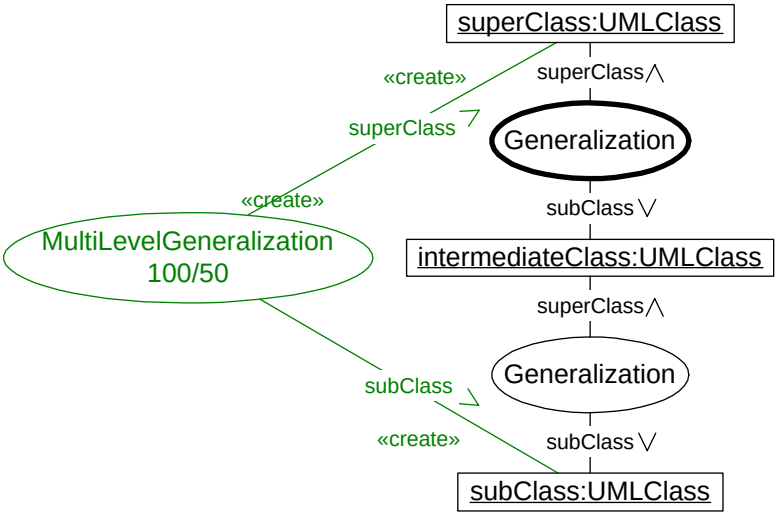

Abbildung A.14.: *MultiLevelGeneralization*

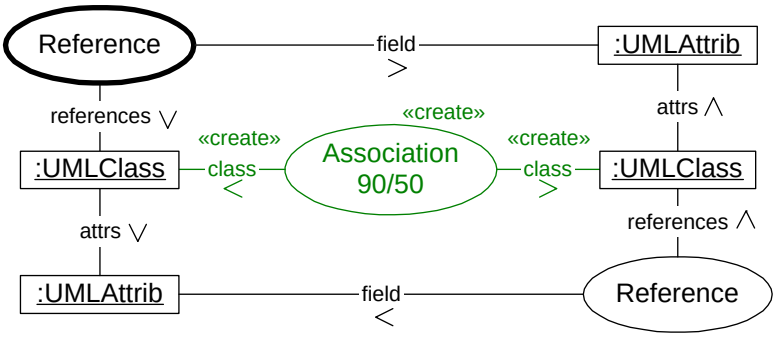

Abbildung A.15.: *Association*

A.2. Design Patterns

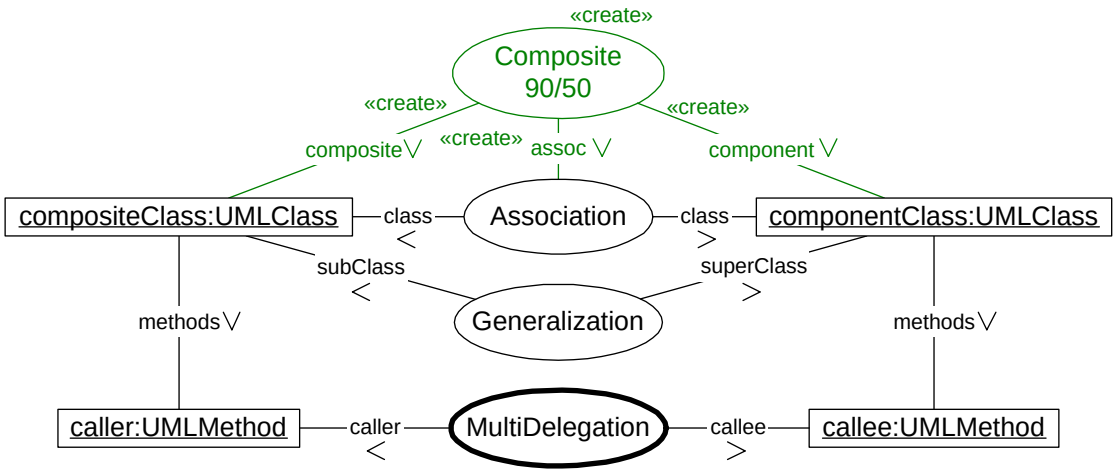

Abbildung A.16.: *Composite*

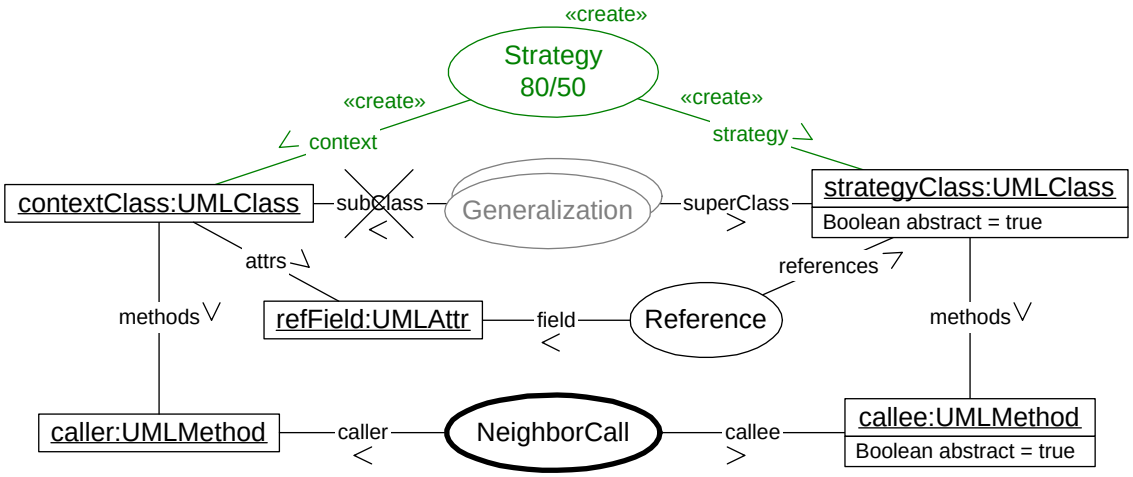

Abbildung A.17.: *Strategy*

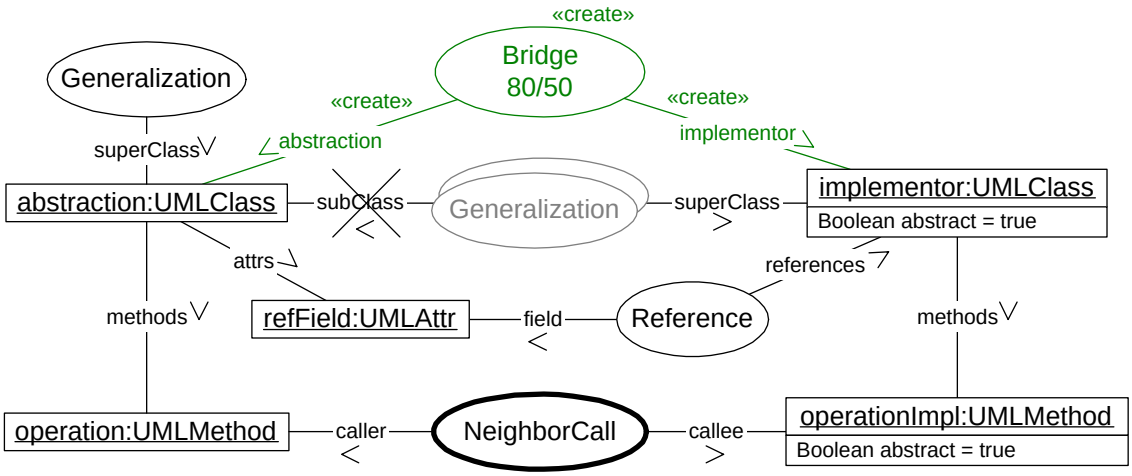

Abbildung A.18.: *Bridge*

A.3. Vererbungsbeziehungen

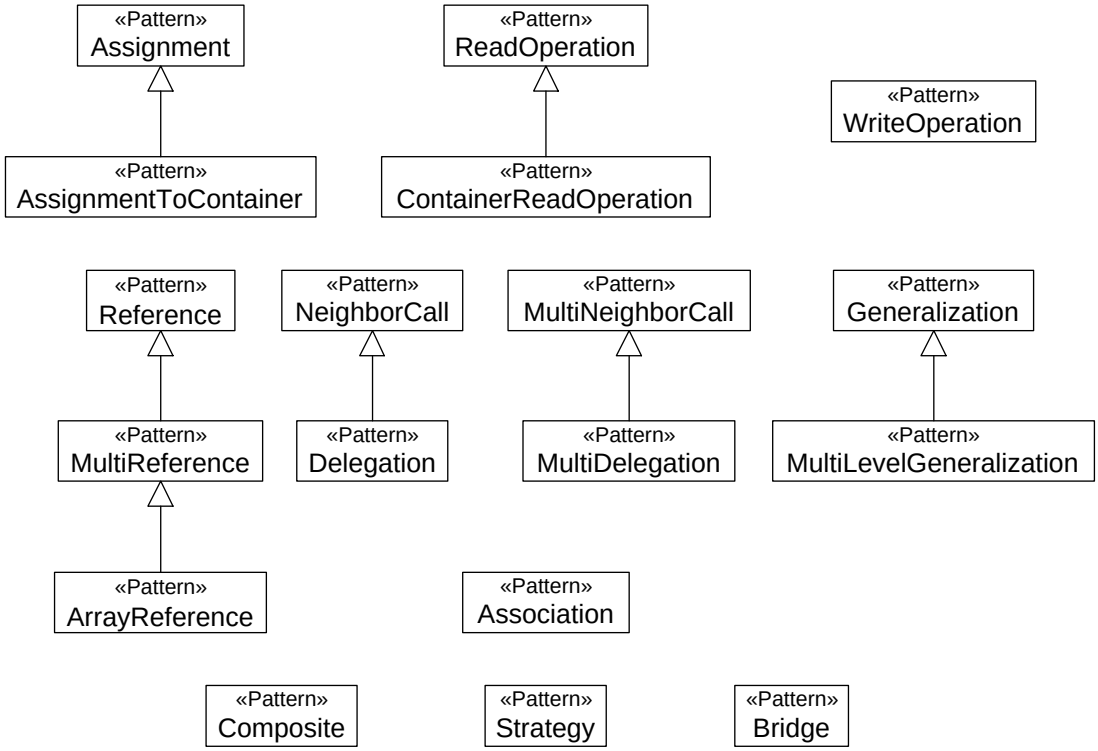

Abbildung A.19.: Hierarchie der Muster

A.4. Generic Fuzzy Reasoning Net

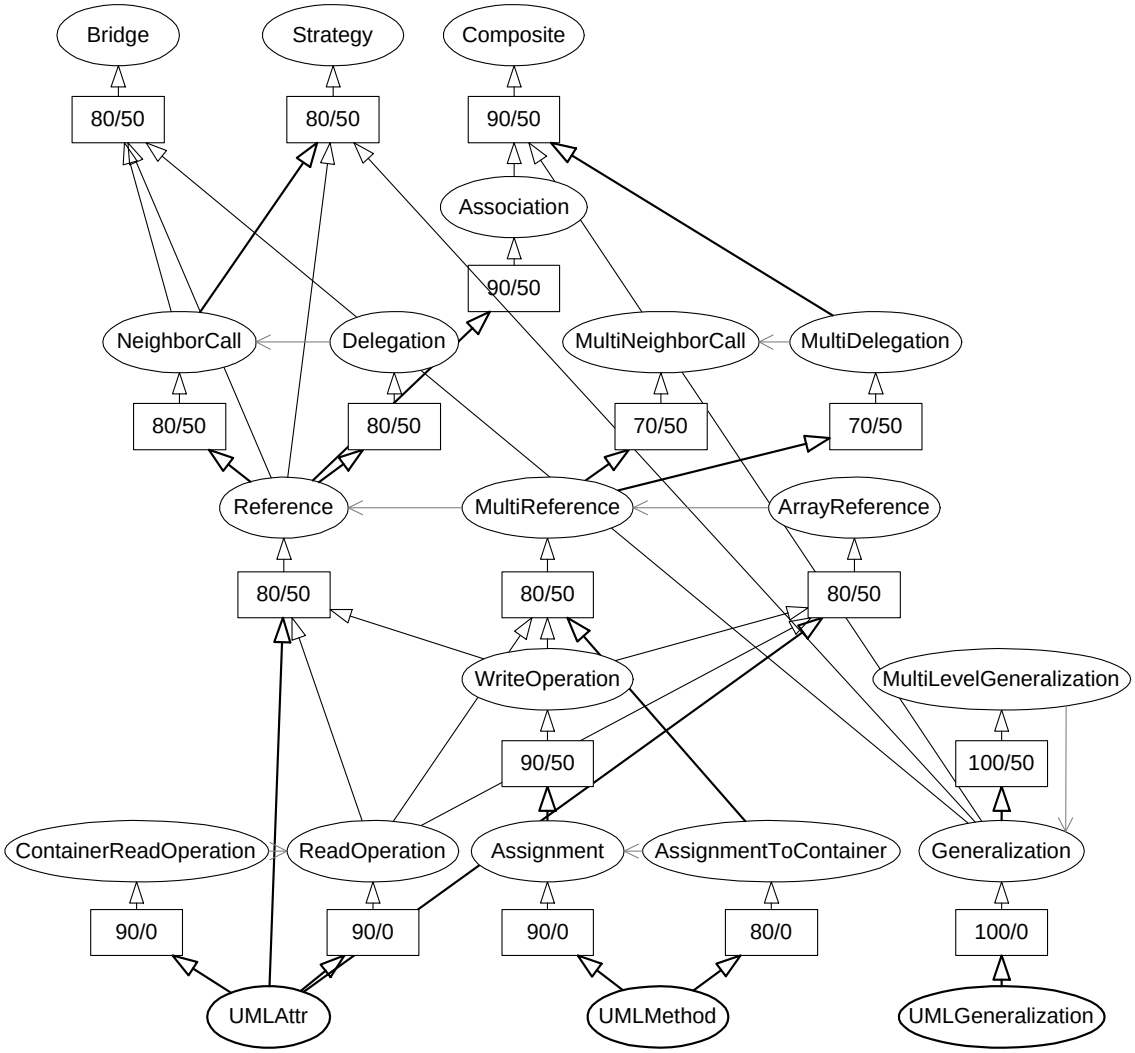

Abbildung A.20.: Alle Muster des Kataloges organisiert in einem GFRN

Literatur

[AIS+77] Alexander, C. ; Ishikawa, S. ; Silverstein, M. ; Jacobson, M. ; Fiksdahl-King, I. ; Angel, S.: *A Pattern Language*. Oxford University Press, 1977

[AWT] SUN Microsystems: *AWT, the SUN Java Abstract Window Toolkit. Online at http://java.sun.com/products/jdk/awt*

[CC90] Chikofsky, Elliot J. ; Cross II, James H.: Reverse Engineering and Design Recovery: A Taxonomy. In: *IEEE Software* 7 (1990), January, Nr. 1, S. 13–17

[FNT98] Fischer, T. ; Niere, J. ; Torunski, L.: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, July 1998

[FNTZ98] Fischer, T. ; Niere, J. ; Torunski, L. ; Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, G. (Hrsg.) ; G.Rozenberg (Hrsg.): *Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer Verlag, 1998 (LNCS 1764)

[GHJV95] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA : Addison-Wesley, 1995

[Hei98] Heitbreder, M.: *Eine Ausführungsmaschine für Generic Fuzzy Reasoning Nets auf Basis unscharfer Petrinetze*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, July 1998

[Jah99] Jahnke, J.H.: *Management of Uncertainty and Inconsistency in Database Reengineering Processes*, University of Paderborn, Paderborn, Germany, Diss., September 1999

[JGL] ObjectSpace, Inc.: *JGL, the ObjectSpace (Voyager) Java Generic Library. Online at http://www.objectspace.com*

[KM96] Konar, A. ; Mandal, A. K.: Uncertainty Management in Expert Systems Using Fuzzy Petri Nets. In: *IEEE Transactions on Knowledge and Data Engineering* 8 (1996), February, Nr. 1, S. 96–105

[KP96] Krämer, C. ; Prechelt, L.: Design recovery by automated search for structural design patterns in object-oriented software. In: *Proc. of the 3^{rd} Working Conference on Reverse Engineering (WCRE), Monterey, CA*, IEEE Computer Society Press, November 1996, S. 208–215

[Meh84] Mehlhorn, K.: *Graph Algorithms and NP-Completeness*. 1^{st}. Springer Verlag, 1984

[NSW+01] Niere, J. ; Schäfer, W. ; Wadsack, J.P. ; Wendehals, L. ; Welsh, J.: Towards Pattern-Based Design Recovery / University of Paderborn. Paderborn, Germany, 2001. – Forschungsbericht. (to appear)

[Pal01] Palasdies, M.: *Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, May 2001

[RS95] Rekers, J. ; Schürr, A.: A Graph Grammar Approach to Graphical Parsing. In: *Proc. of the IEEE Symposium on Visual Languages, Darmstadt, Germany*, IEEE Computer Society Press, 1995

[SG98] Seemann, J. ; von Gudenberg, J.W.: Pattern-Based Design Recovery of Java Software. In: *ACM SIGSOFT Software Engineering Notes* 23 (1998), November, Nr. 6

[TA99] Tonella, P. ; Antoniol, G.: Object Oriented Design Pattern Inference. In: *Proc. of the 5^{th} Symposium on Software Development Environments (SDE5)*, IEEE Computer Society Press, September 1999, S. 230–238

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

[Wil96] Wills, L.M.: Using Attributed Flow Graph Parsing to Recognize Programs. In: *Proc. of International Workshop on Graph Grammars and Their Application to Computer Science*. Williamsburg, Virginia, 1994 : Springer Verlag, November 1996 (LNCS 1073)

Index