Bewertung automatisch erkannter Instanzen von Software-Mustern

von
Dietrich Travkin

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Str. 100
33098 Paderborn

Bewertung automatisch erkannter Instanzen von Software-Mustern

Diplomarbeit
im Rahmen des integrierten Studiengangs
Informatik

von
Dietrich Travkin
Ginsterweg 1
33813 Oerlinghausen

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Dr. Ekkart Kindler

August 2006

Erklärung

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

Ort, Datum	Unterschrift

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Neben der Neuentwicklung ist die Wartung von Software ein Schwerpunkt der modernen Softwareentwicklung. Mit der wachsenden Komplexität und Größe der Softwareprojekte steigen auch der Entwicklungsaufwand und die Kosten. Die Wiederverwendung und Weiterentwicklung von bestehender Software wird damit immer wichtiger. Softwareprodukte sind ständigen Veränderungen ausgesetzt. Sie werden über mehrere Jahre gepflegt, weiterentwickelt und an neue Anforderungen angepasst.

Die ursprünglichen Entwickler sind häufig nicht mehr verfügbar, und neue Entwickler müssen sich in die Softwaresysteme einarbeiten. Hinzu kommt, dass die Dokumentation und das Design der Produkte in vielen Fällen bei Änderungen an der Software nicht aktualisiert werden und somit im Laufe der Zeit veralten oder von Beginn an fehlen. Das führt dazu, dass die nötigen Informationen direkt aus dem Quellcode gewonnen werden müssen, was die Einarbeitung in ein Softwaresystem erschwert und damit den Zeit- und Kostenaufwand für die Wartung von Software erhöht.

Die Analyse eines Systems zur Identifizierung seiner Komponenten und ihrer Beziehungen untereinander sowie zur abstrakteren Repräsentation des Systems wird als *Reverse Engineering* bezeichnet [CC90]. Um sich in ein unbekanntes Softwaresystem einzuarbeiten, versuchen die Entwickler mit Hilfe des Reverse Engineerings, das ursprüngliche Design eines Softwaresystems wiederzugewinnen und so vom Quellcode zu abstrahieren.

Neben diversen UML-Diagrammen, die Strukturen und Beziehungen zwischen Softwarekomponenten beschreiben und deshalb häufig beim Reverse Engineering verwendet werden, eignen sich auch Software-Muster zur abstrakten Beschreibung von Softwaresystemen.

So genannte Entwurfsmuster (auch *Design Patterns* oder Gang-of-Four- beziehungsweise GoF-Patterns genannt) [GHJV95] beschreiben auf Design-Ebene typische Lösungen für wiederkehrende Probleme. Die Beschreibung der Struktur und Funktion der Softwarekomponenten abstrahiert von den Implementierungsdetails. Durch den inzwischen hohen Bekanntheitsgrad der Entwurfsmuster bilden die einprägsamen Musternamen ein gemeinsames Vokabular unter Softwareentwicklern. Das Erkennen eines Entwurfsmusters liefert also eine kompakte Beschreibung der beteiligten Softwarekomponenten und lässt Rückschlüsse auf die Intention der ursprünglichen Entwickler zu.

Ebenso wie Design-Empfehlungen können auch Software-Mängel als Muster beschrieben werden. *Anti Patterns* [BMMM98] bilden das Gegenstück zu Design Patterns. Diese beschreiben typische Problemlösungen, deren negative Konsequenzen

die Vorteile der Lösungen überwiegen. Außer der Darstellung und Erläuterung eines schlechten Softwareentwurfs und seiner Konsequenzen enthält die Beschreibung eines Anti Patterns auch Vorschläge zur Verbesserung des Entwurfs. *Bad Smells* [Fow99, Kapitel 3] geben Hinweise auf problematischen Quellcode oder Entwurfsprobleme und bieten so genannte *Refactorings* zur Verbessung des Codes an. Beispiele für Bad Smells sind unter anderem Klassen mit vielen Attributen, Methoden und Quellcode („Large Class“) oder Methoden, die mehr Daten aus einer anderen Klasse als aus der eigenen verwenden („Feature Envy“).

Während Design Patterns typische Softwareentwürfe beschreiben, deuten Anti Patterns und Bad Smells auf Problemstellen in einem Softwaresystem hin. Die Erkennung solcher Software-Muster ist also eine gute Möglichkeit, vom Quellcode zu abstrahieren oder eine Software auf Mängel zu untersuchen, und damit sehr hilfreich beim Reverse Engineering. Um den Aufwand für die Mustererkennung zu reduzieren, ist eine Werkzeugunterstützung erwünscht.

1.1 Problemstellung

Software-Muster wie Design Patterns und Anti Patterns sind meist informell beschrieben. Sie legen die Struktur und das Verhalten der Komponenten fest, ohne eine konkrete Implementierung zu fordern.

Design Patterns werden allgemein beschrieben, damit die Anwendung des Musters beim *Forward Engineering*[1] an eine spezielle Situation angepasst werden kann. Dem Entwickler werden also viele Freiheiten bezüglich Entwurf und Implementierung überlassen, sodass ein Muster auf viele verschiedene Weisen realisiert werden kann.

Zum Beispiel kann die Implementierung einer Referenz oder Assoziation[2] je nach Programmierstil und -konventionen variieren. Der Programmierstil entscheidet darüber, wie die Referenzen gespeichert werden: in einem Attribut, einem Array oder in einem bestimmten Container. Die Konventionen bestimmen die Namen der zugehörigen Zugriffsmethoden. Außerdem kann die gleiche Semantik durch unterschiedliche Syntax ausgedrückt werden: eine `case`-Anweisung kann durch mehrere `if`-Anweisungen ersetzt werden, Schleifenkonstrukte (`for`, `while`, `do until`) sind untereinander austauschbar.

Die Beschreibung von Anti Patterns und Bad Smells ist noch allgemeiner und knapper gehalten, als die von Design Patterns. Muster wie das Anti Pattern „The Blob“, welches in ähnlicher Form auch als Bad Smell „Large Class“ beschrieben wird, werden durch vage Aussagen, zum Beispiel „fehlender objektorientierter Entwurf“ oder „eine Klasse mit vielen Attributen, Methoden oder beidem“ skizziert. Quantitative Angaben sind unscharf: zum Beispiel „klein“ oder „groß“, „viel“ oder „wenig“. Strukturen werden teilweise nur umgangssprachlich oder metaphorisch

[1] Forward Engineering ist der Prozess, bei dem man aus Designs und Entwürfen auf hoher Abstraktionsebene konkrete Implementierungen erstellt.[CC90]

[2] In [TM05, Tra05] werden verschiedene Implementierungen von Assoziationen diskutiert.

dargestellt. Ein Beispiel ist das Anti Pattern „Lava Flow“, das unter anderem durch „nicht benutzter (toter) Code“ und „nicht dokumentierte, komplexe, wichtig aussehende Funktionen, Klassen oder Segmente, die keinen klaren Bezug zur Architektur des Systems haben“ beschrieben wird.

Die hohe Ungenauigkeit bei der Beschreibung der Muster erschwert eine Werkzeug-gestützte Mustererkennung. Es müssen eine formale Spezifikation der Muster und ein Mustererkennungsalgorithmus so gewählt werden, dass einerseits die vielen Implementierungsvarianten eines Musters berücksichtigt werden und andererseits der Aufwand für die Spezifikation der Muster gering und die Präzision bei der Erkennung hoch ist. Gleichzeitig soll die Skalierbarkeit des Mustererkennungsalgorithmus sichergestellt werden, da die zu untersuchenden Softwaresysteme oft mehrere Hunderttausend oder Millionen Code-Zeilen umfassen.

Eine Möglichkeit, mit den vielen Implementierungsvarianten eines Musters umzugehen, ist, jede einzelne Variante exakt zu spezifizieren. So würde man eine hohe Präzision erreichen, aber es müssten unzählige Spezifikationen erstellt werden, was einen enormen Aufwand für den Reverse Engineer bedeuten würde und deshalb nicht praktikabel ist. Wegen der großen Spezifikationenanzahl wäre auch die Laufzeit einer automatisierten Suche nach allen Mustervarianten entsprechend hoch.

Bei dem an der Universität Paderborn entwickelten Mustererkennungsprozess [NSW+02] wird ein anderer Ansatz verfolgt. Um die Anzahl der Spezifikationen und die Laufzeit der Mustersuche zu reduzieren, werden mehrere Implementierungsvarianten zu einem Muster in nur einer Spezifikation zusammengefasst. Dazu werden ausschließlich die Gemeinsamkeiten der Varianten spezifiziert, alle anderen Informationen werden weggelassen [NWW03]. Durch diese unvollständigen Spezifikationen werden Muster in gewisser Weise „unscharf“ beschrieben. Eine Konsequenz daraus ist der Verlust der Präzision bei der Mustersuche. Die Suchergebnisse können so genannte *False Positives* – also Funde, die nicht das gesuchte Muster repräsentieren – enthalten. Bei großen Softwaresystemen ist außerdem eine große Anzahl an Suchergebnissen zu erwarten. Ohne ein Hilfsmittel würden viele False Positives darunter das Identifizieren der relevanten Musterfunde sehr mühsam machen. Um dem Reverse Engineer einen Hinweis auf die Qualität oder die Verlässlichkeit der Suchergebnisse zu geben, werden die Musterfunde automatisch bewertet.

Die Bewertung der erkannten Musterinstanzen erfolgt mit Hilfe eines so genannten *Vertrauenswertes*, der vom Reverse Engineer vor der Mustererkennung für jede Spezifikation geschätzt wird. Dieser Wert gibt prozentuell an, wie groß der Anteil der korrekten Musterfunde an der Gesamtanzahl der Funde zu einer Spezifikation voraussichtlich sein wird, und beschreibt damit die Präzision der Musterspezifikation. Spezifikationen können bei dem Ansatz in anderen Spezifikationen wiederverwendet werden, um wiederkehrende Strukturen zusammenzufassen. Die Bewertung einer Musterinstanz entspricht – vereinfacht dargestellt – dem Minimum aus dem Vertrauenswert der zugehörigen Musterspezifikation und der Vertrauenswerte der darin verwendeten Spezifikationen. Nach der Mustererken-

nung hat der Reverse Engineer allerdings die Möglichkeit, Bewertungsergebnisse zu korrigieren, wodurch auch die ursprünglich geschätzten Vertrauenswerte automatisch mitkorrigiert werden [NMW04].

Bei dieser Bewertungsmethode werden die Eigenschaften einer Musterinstanz nicht berücksichtigt. Nur die Vertrauenswerte der Musterspezifikationen haben Einfluss auf die Bewertungsergebnisse. Von einer Spezifikation mit geringem Vertrauenswert wird auf schlechte Funde geschlossen, was allerdings für sämtliche Funde zur selben Spezifikation gleichermaßen gilt. Die Bewertungen aller Musterinstanzen zur selben Musterspezifikation sind dadurch (bis auf einige Ausnahmen) gleich, wodurch sie nur eine geringe Aussagekraft für den Reverse Engineer haben.

Eine weitere Möglichkeit, Musterinstanzen zu bewerten, ist in [Wen05a] beschrieben worden. Bei diesem Ansatz werden unter anderem auch unvollständige Musterinstanzen gesucht. Ihre Bewertung gibt durch einen Prozentwert an, wie vollständig ein Musterfund ist. Je mehr durch die Musterspezifikation gestellte Bedingungen der Fund erfüllt, desto vollständiger ist dieser und desto höher wird er bewertet. Die Spezifikation eines Musters besteht bei diesem Ansatz aus mehreren Komponenten, die jeweils mehrere Bedingungen definieren können. Zur Bewertung einer Musterinstanz wird für jede Komponente bestimmt, wie viel Prozent der angegebenen Bedingungen erfüllt sind. Der Durchschnitt aus diesen Prozentwerten entspricht der Bewertung der Musterinstanz.

Bei diesem Verfahren hängt die Bewertung eines Musterfundes im Gegensatz zum vorherigen Ansatz nicht allein von den Eigenschaften der Musterspezifikation, sondern vor allem von denen des Musterfundes ab. Dadurch können verschiedene Instanzen eines Musters unterschiedlich bewertet werden. Allerdings spiegelt die Bewertung einer Musterinstanz aufgrund der Durchschnittsberechnung den Anteil der erfüllten Bedingungen nicht korrekt wieder. Zum Beispiel kann ein Muster aus zwei Komponenten a und b bestehen, wovon a zwei und b zehn Bedingungen enthält. Eine Musterinstanz, bei der die beiden Bedingungen der Komponente a erfüllt sind, aber die zehn der anderen Komponente nicht, wird mit $\frac{1}{2}(100\% + 0\%) = 50\%$ bewertet. Das gleiche gilt auch für die Bewertung einer Musterinstanz, bei der die zehn Bedingungen der Komponente b erfüllt sind, die zwei der anderen jedoch nicht. Der Anteil der erfüllten Bedingungen dagegen ist bei dem ersten Fall $\frac{2}{12} \approx 16.7\%$ und bei dem zweiten Fall $\frac{10}{12} \approx 83.3\%$. Bei diesem Bewertungsverfahren kann also von einer höheren Bewertung nicht auf eine vollständigere Musterinstanz geschlossen werden, was den Nutzen der Bewertung in Frage stellt.

1.2 Lösungsansatz

Ziel dieser Arbeit ist es, ein automatisiertes Bewertungsverfahren für Musterfunde zu entwickeln, welches die Funde abhängig von ihren Eigenschaften bewertet und numerisch angibt, wie gut ein Fund auf die Beschreibung in der zugehörigen Musterspezifikation passt. Die Identifikation der relevanten Funde in den Suchergebnissen soll durch die Bewertung vereinfacht werden.

Das im Rahmen dieser Diplomarbeit entwickelte Bewertungsverfahren basiert auf dem Mustererkennungsverfahren aus [NSW+02, NWW03] und greift die Idee auf, eine Musterinstanz durch angeben ihres Vollständigkeitsgrades zu bewerten.

Wie bereits erwähnt werden bei dem Ansatz aus [NWW03] mehrere Implementierungsvarianten in einer Spezifikation zusammengefasst, indem nur die Gemeinsamkeiten der Implementierungsvarianten beschrieben werden. Da bereits geringe Abweichungen von der Spezifikation dazu führen, dass eine Musterinstanz nicht erkannt wird, bleiben andere Informationen über ein Muster ungenutzt.

Eine Referenz zwischen zwei Klassen zum Beispiel wird immer durch ein Attribut in der referenzierenden Klasse realisiert. Bei einer typischen Implementierungsvariante ist die Sichtbarkeit[3] dieses Attributs eingeschränkt (Information-Hiding-Prinzip) und es sind Methoden für den Lese- und Schreibzugriff implementiert. Dieses ist jedoch nur eine von vielen Implementierungsvarianten. Eine andere, aber unübliche Variante ist die Implementierung mit einem öffentlichen (`public`) Attribut ohne Zugriffsmethoden. Um alle Varianten erkennen zu können, wird bei der Spezifikation des Musters „Referenz“ auf die Verwendung der Informationen darüber, dass bei einer Referenzimplementierung zu dem Attribut meist auch Zugriffsmethoden existieren und die Sichtbarkeit des Attributs eingeschränkt ist, verzichtet.

Die Idee bei dem neu entwickelten Bewertungsverfahren ist es, einen Teil der bisher ungenutzten Informationen zur Bewertung eines Musterfundes zu verwenden und in Form von optionalen Bedingungen in die Musterspezifikation aufzunehmen. Dadurch, dass die Bedingungen optional sind, wird eine Musterinstanz auch dann erkannt, wenn eine dieser Bedingungen nicht erfüllt ist. Ist so eine Bedingung aber erfüllt, so wird die Musterinstanz als ein vollständigerer und damit besserer Fund angesehen. Die Bewertung einer Musterinstanz beschreibt, welcher Anteil der in einer Musterspezifikation enthaltenen Bedingungen erfüllt ist. Je mehr optionale Bedingungen erfüllt sind, desto höher wird eine Musterinstanz bewertet.

Bei der Spezifikation einer Referenz können die Bedingungen, welche die Einschränkung der Sichtbarkeit des Attributs und zugehörige Zugriffsmethoden fordern, als optional gekennzeichnet werden. Auf diese Weise werden auch andere Implementierungsvarianten weiterhin erkannt, allerdings werden die Varianten mit Zugriffsmethoden und der eingeschränkten Sichtbarkeit höher bewertet. Die Spezifikation von optionalen Bedingungen ermöglicht es dem Reverse Engineer, auch diejenigen Informationen für die Mustererkennung zu nutzen, die auf einen Großteil der Implementierungsvarianten, aber nicht auf alle zutreffen.

Die Bewertung eines Musterfundes entspricht dem Anteil der erfüllten Bedingunen. So ist es einem Reverse Engineer selbst bei großen Anzahlen von Suchergebnissen auf einfache Weise möglich, die relevanten Musterfunde zu identifizieren. Güteschranken können die Übersicht bei den Suchergebnissen weiter erhöhen, in-

[3] In objektorientierten Programmiersprachen wie Java kann zum Beispiel angegeben werden, dass auf ein Attribut nur innerhalb seiner Klasse zugegriffen werden kann (`private`). Eine Alternative ist die Möglichkeit, darauf aus einer beliebigen Klasse zuzugreifen (`public`).

dem alle Funde, deren Bewertung die Güteschranke unterschreitet, ausgeblendet werden.

Mehrere Implementierungsvarianten werden wie bei dem Ansatz aus [NWW03] in einer Musterspezifikation zusammengefasst. Außerdem wird der Umfang der angegebenen optionalen Bedingungen gering gehalten. Auf diese Weise wird der Aufwand für die Spezifikation eingeschränkt und die Skalierbarkeit des Mustererkennungsalgorithmus gesichert. Dadurch, dass Musterspezifikationen nur um zusätzliche Informationen in Form von optionalen Bedingungen erweitert werden, gehen keine Informationen verloren. Alle bisher gefundenen Musterinstanzen werden weiterhin erkannt. Die Präzision bei der Mustererkennung bleibt erhalten. Insbesondere wird die Anzahl der so genannten *False Negatives* – also der Musterinstanzen, die nicht als solche erkannt werden – nicht erhöht.

1.3 Struktur der Arbeit

Das Kapitel 2 behandelt die Grundlagen des Mustererkennungsverfahrens, auf dem diese Arbeit aufbaut. Es werden sowohl die Musterspezifikation als auch die Mustersuche nach dem Ansatz aus [NSW+02] erläutert.

Als Nächstes werden in dem Kapitel 3 bereits existierende Bewertungsverfahren für Musterinstanzen vorgestellt. Diese werden untersucht und die mit den Verfahren verbundenen Probleme werden aufgezeigt.

Das Kapitel 4 befasst sich mit dem in dieser Arbeit entwickelten Bewertungsfahren. Nach der Formulierung von allgemeinen Anforderungen an ein Bewertungsverfahren wird der erarbeitete Ansatz vorgestellt. Anschließend werden die für den Ansatz notwendigen Erweiterungen der Musterspezifikationssprache beschrieben. Schließlich wird die Bewertungsfunktion definiert und an Beispielen erklärt.

Die Umsetzung des Verfahrens wird im Kapitel 5 beschrieben. Es werden die Erweiterung der Musterspezifikationssprache und die Anpassung der Mustersuche erläutert.

Darauf folgend werden im Kapitel 6 erste praktische Erfahrungen mit dem Bewertungsverfahren geschildert.

Schließlich werden im Kapitel 7 die Ergebnisse dieser Arbeit zusammengefasst und in einem Ausblick Ideen für mögliche Erweiterungen des Ansatzes präsentiert.

2 Mustererkennung in Fujaba

Eines der heute existierenden CASE[1]-Werkzeuge ist die an der Universität Paderborn entwickelte *Fujaba Tool Suite* (kurz: Fujaba) [Fuj06, FNT98]. Neben dem Forward Engineering bietet diese Entwicklungsumgebung auch eine Unterstützung für Reverse Engineering [NSW+02, Wen01, Nie04].

Fujaba ermöglicht die Modellierung von Softwaresystemen mit UML [Obj04] und stellt eine Java-Code-Generierung aus den erstellten Modellen bereit. Neben zahlreichen Diagrammarten wie Klassen- und Aktivitätsdiagrammen sowie Statecharts werden auch so genannte *Story-Diagramme* [FNTZ98] unterstützt. Diese sind eine Art von Aktivitätsdiagrammen, die zur Beschreibung der dynamischen Änderungen von Objektstrukturen spezielle Kollaborationsdiagramme, so genannte *Story Patterns*, einbetten. Mit Hilfe dieser Diagrammarten, die mit einer formalen Semantik unterlegt sind, ist die Spezifikation von dynamischen und statischen Anteilen eines Softwaresystems sowie die Generierung von zugehörigem Code möglich.

Das Reverse Engineering wird zum einen durch die Möglichkeit, Java-Quellcode in ein UML-Modell zu parsen, und zum anderen durch eine semi-automatische Mustererkennung unterstützt. Die Mustererkennung wird durch mehrere die Fujaba Tool Suite erweiternde Plug-ins realisiert, die zusammen mit Fujaba in der so genannten *Fujaba Tool Suite RE Edition* enthalten sind.

Bevor ein Softwaresystem nach Mustern durchsucht werden kann, werden die Muster in Fujaba mit einer der UML ähnlichen Spezifikationssprache beschrieben und in einem *Musterkatalog* gespeichert. Nachdem der Quellcode des zu untersuchenden Systems zur Repräsentation in einem *abstrakten Syntaxgraphen* geparst wurde, kann mit Hilfe eines speziellen Inferenzalgorithmus und des Musterkatalogs eine Mustersuche auf dem System durchgeführt werden. Dabei wird das beim Parsen erstellte UML-Modell bzw. der abstrakte Syntaxgraph mit Annotationen angereichert, die die Fundstellen der Musterinstanzen markieren. Schließlich findet eine automatische Bewertung der gefundenen Musterinstanzen statt und die Ergebnisse werden angezeigt.

In den folgenden Abschnitten dieses Kapitels werden die einzelnen Schritte der Mustererkennung mit Fujaba genauer erläutert.

[1] Computer Aided Software Engineering

2.1 Abstrakter Syntaxgraph

Zur besseren algorithmischen Handhabung des Quellcodes eines zu untersuchenden Softwaresystems wird dieser durch eine Objektstuktur repräsentiert, die abstrakter Syntaxgraph (kurz: ASG) genannt wird. Das in Fujaba verwendete Modell für abstrakte Syntaxgraphen (siehe Abb. 2.1) ist speziell für objektorientierte Programmiersprachen entwickelt worden und ermöglicht eine nicht-textuelle Repräsentation des Quellcodes auf einer abstrakteren Ebene, unabhängig von der Syntax konkreter Programmiersprachen.

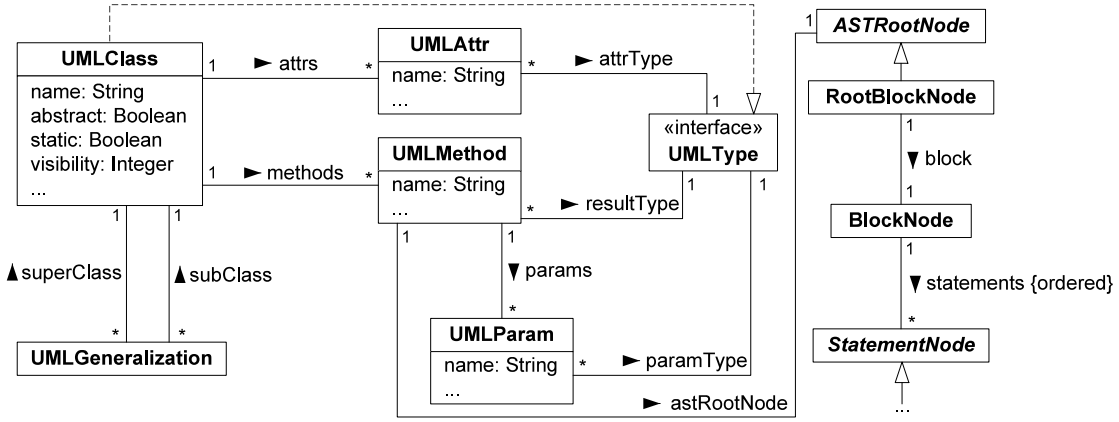


Abbildung 2.1: Ausschnitt des Modells für abstrakte Syntaxgraphen

Ein abstrakter Syntaxgraph beschreibt insbesondere die Struktur eines Softwaresystems. Dazu zählen neben Vererbungs- und anderen Beziehungen auch weitere Informationen über Klassen, Attribute und Methoden, zum Beispiel Name, Sichtbarkeit und Signatur.

Zur Veranschaulichung zeigt die Abbildung 2.3 einen Ausschnitt aus einem abstrakten Syntaxgraphen, der die Struktur des Programmbeispiels aus Abbildung 2.2 repräsentiert. Der Graph ist ein UML-Objektdiagramm. Zur besseren Übersicht wurden einige Attributwerte der Objekte ausgeblendet.

Sämtliche in dem Quellcode vorkommenden Klassen werden durch UMLClass-Objekte repräsentiert, die mehrere Attribute zur Beschreibung des Namens, der Sichtbarkeit und anderer Eigenschaften verwenden. Analog dazu gibt es Objekte für Methoden, Attribute und Parameter. Die Information, dass die Klasse OnlineShop von der Klasse Shop erbt, wird durch ein UMLGeneralization-Objekt und seine Verknüpfungen ausgedrückt. Der Rückgabetyp von Methoden wird durch eine Verknüpfung des UMLMethod-Objekts mit dem UMLClass-Objekt, das den Rückgabetyp der Methode repräsentiert, angegeben.

Bei dem ASG-Ausschnitt in Abbildung 2.3 fällt auf, dass keine Informationen über den Aufbau der Methodenrümpfe enthalten sind, doch auch diese sind Bestandteil von abstrakten Syntaxgraphen. Repräsentiert werden Methodenrümpfe

```
public class OnlineShop extends Shop
{
   private Map articleImagesMap = null;

   public Map getArticleImagesMap()
   {
      ...
   }

   public Image getArticleImage(Article article)
   {
      if (article != null)
      {
         return (Image) this.getArticleImagesMap().get(article);
      }
      else
      {
         return null;
      }
   }
}
```

Abbildung 2.2: Beispiel einer Klassenimplementierung in Java

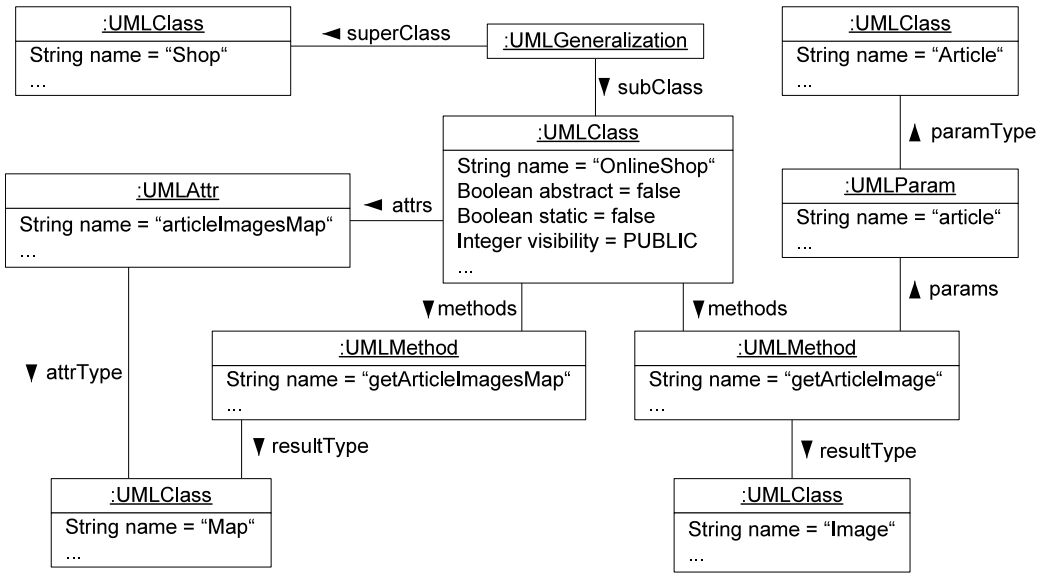


Abbildung 2.3: Ausschnitt des abstrakten Syntaxgraphen zum Code-Beispiel in Abbildung 2.2

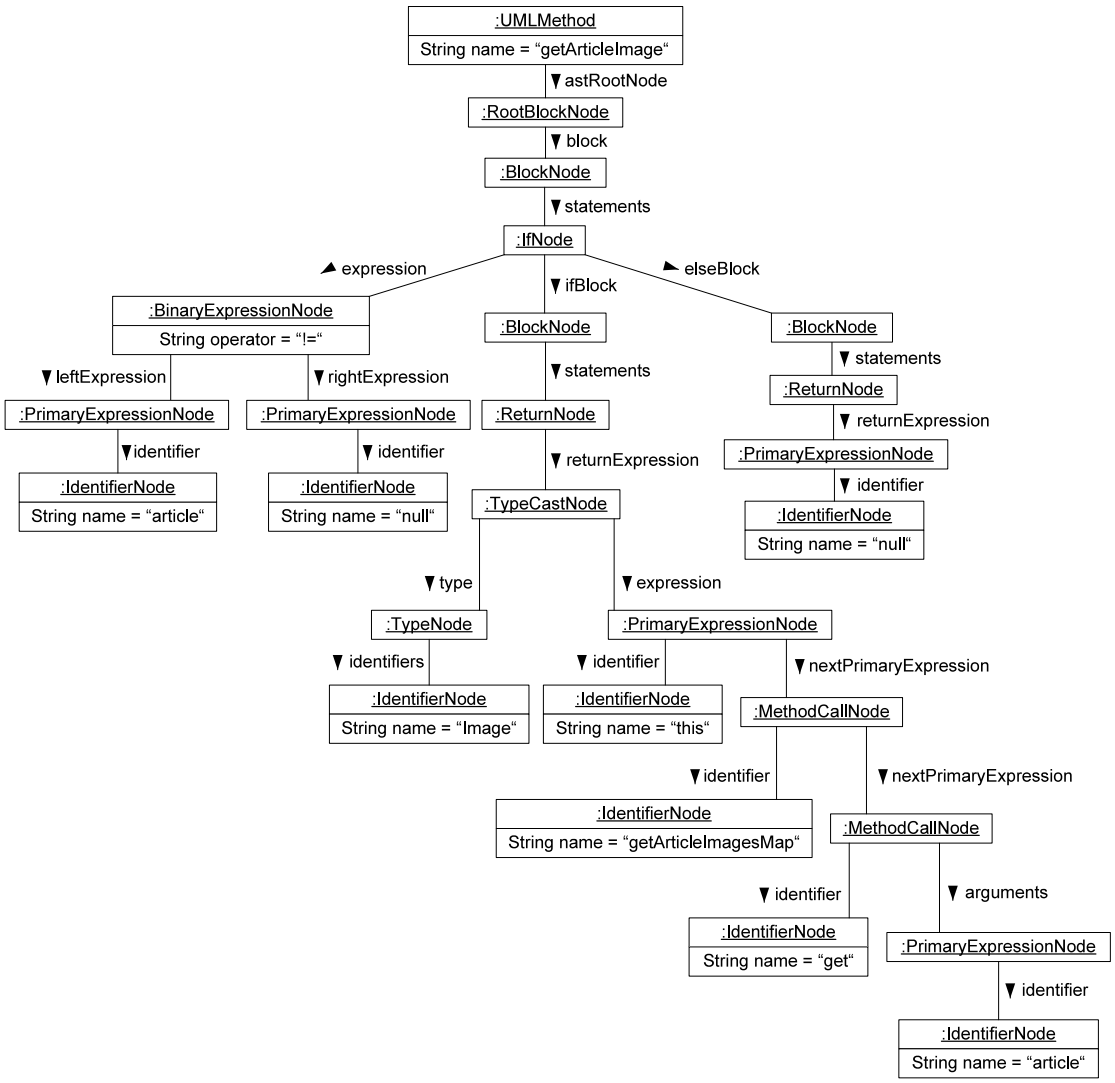


Abbildung 2.4: Abstrakter Syntaxbaum zur Methode getArticleImage aus dem Code-Beispiel in Abbildung 2.2

im ASG durch so genannte *abstrakte Syntaxbäume* (kurz: AST[2]). Diese spiegeln die Struktur der Rümpfe wieder und geben unter anderem Auskunft über enthaltene Methodenaufrufe, Schleifen und bedingte Anweisungen.

Ein Beispiel für einen abstrakten Syntaxbaum ist in einem weiteren ASG-Ausschnitt in der Abbildung 2.4 zu sehen. Dieser Graph repräsentiert die Methode getArticleImage aus dem Code-Beispiel in Abbildung 2.2.

Die if-Anweisung aus dem Code-Beispiel in Abbildung 2.2 wird in dem abstrakten Syntaxbaum durch ein IfNode-Objekt repräsentiert, das mit je einem weiteren Objekt für den if- und else-Zweig sowie einem BinaryExpressionNode-Objekt zur Beschreibung der Bedingung verknüpft ist. Die Methodenaufrufe in dem if-Zweig

[2]Abk. für „Abstract Syntax Tree“

werden durch MethodCallNode-Objekte dargestellt, wobei der Name der aufgerufenen Methode durch das name-Attribut des zugehörigen IdentifierNode-Objekts ausgedrückt wird.

2.2 Musterspezifikation

Software-Muster werden in Fujaba mit Hilfe einer Spezifikationssprache beschrieben, die im Rahmen zweier Diplomarbeiten entwickelt [Pal01] bzw. erweitert [Wen01] wurde und in diesem Abschnitt vorgestellt wird.

2.2.1 Musterspezifikationsdiagramme

Die Spezifikation von Mustern erfolgt in so genannten *Musterspezifikationsdiagrammen*, welche die Eigenschaften und Beziehungen der an einem Muster beteiligten Elemente in Form eines Ausschnitts eines abstrakten Syntaxgraphen darstellen. Somit können Beziehungen zwischen Klassen, Methoden und Attributen einer Software sowie ihre Eigenschaften beschrieben werden. Die dabei verwendete Syntax ist eine Erweiterung der Syntax von UML-Objektdiagrammen.

Ein Musterspezifikationsdiagramm stellt eine Beispielausprägung eines Musters dar. Die darin vorkommenden Objekte sind also als Platzhalter für konkrete Objekte zu sehen. Das Diagramm beschreibt in erster Linie die zu erfüllenden Eigenschaften und Beziehungen der Objekte, die an einer Musterausprägung beteiligt sind, nicht aber eine konkrete Musterinstanz.

Für die Erkennung des spezifizierten Musters müssen Objekte mit den im Diagramm spezifizierten Eigenschaften und Beziehungen im abstrakten Syntaxgraphen der zu untersuchenden Software gefunden werden.

Mehrere Musterspezifikationen können in so genannten Musterkatalogen zusammengefasst werden. Auf diese Weise können die Spezifikationen gruppiert werden, zum Beispiel wurde an der Universität Paderborn je ein Katalog für die Gang of Four - Design Patterns [GHJV95] und für diverse, Software-Mängel beschreibende Muster wie Anti Patterns [BMMM98], Bad Smells [Fow99] und andere erstellt.

Ein Musterspezifikationsdiagramm beschreibt eine *Graphersetzungsregel* (auch *Graphtransformationsregel* genannt, engl. *graph replacement rule* oder *structure replacement rule*) [Roz97], die im Folgenden *Musterregel* genannt wird. Graphersetzungsregeln definieren Graph-Modifikationen. Dazu werden Teile von Graphen beschrieben, die gesucht und durch andere Teilgraphen ersetzt werden sollen. Der zu transformierende Graph wird *Wirtsgraph* genannt.

Ursprünglich bestehen Graphersetzungsregeln aus zwei Graphen, einem linken und einem rechten. Der linke Graph beschreibt die im Wirtsgraphen zu suchende Struktur. Der rechte Graph stellt dar, durch welche Struktur die zu suchende ersetzt werden soll. Dabei können einzelne Knoten und Kanten des linken Graphen entfernt bzw. durch andere ersetzt oder neue Knoten und Kanten zum Graphen hinzugefügt werden.

Bei Musterregeln ist der zu suchende Teilgraph ein Ausschnitt eines abstrakten Syntaxgraphen. Um die Fundstelle zu markieren, wird der abstrakte Syntaxgraph durch Erzeugen eines speziellen Objekts und zugehöriger Verknüpfungen annotiert. Zur kompakteren Darstellung von Musterregeln werden der linke und rechte Graph der Graphtransformationsregel verschmolzen. Die zu erzeugenden Knoten und Kanten werden mit dem Stereotyp „create“ gekennzeichnet und grün dargestellt.

2.2.2 Beispiel einer Musterspezifikation

Eines der 23 in [GHJV95] beschriebenen Entwurfsmuster ist „Template Method“. Anhand dieses Design Patterns wird im Folgenden die Spezifikation von Software-Mustern skizziert.

Das Entwurfsmuster „Template Method“ wird eingesetzt, um die Struktur eines Algorithmus in einer Klasse zu definieren, aber die Implementierung einiger seiner Schritte in die Unterklassen zu verlagern.

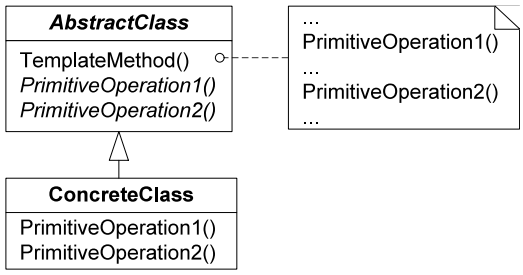


Abbildung 2.5: Struktur des Entwurfsmusters „Template Method“

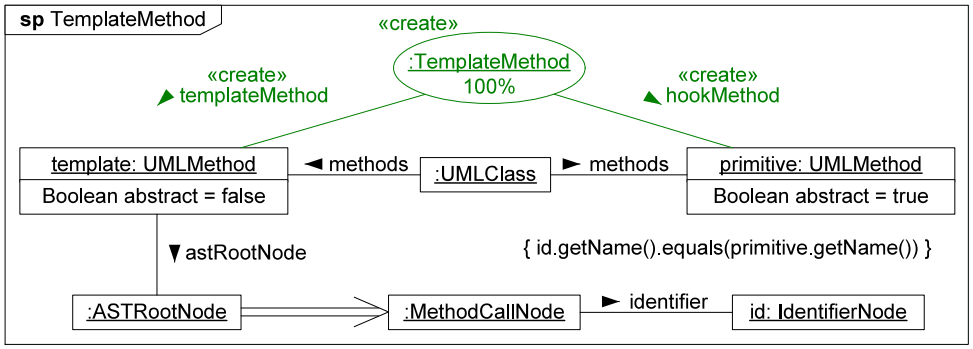


Abbildung 2.6: Spezifikation des Entwurfsmusters „Template Method“

Die Abbildung 2.5 stellt die Struktur des Musters laut [GHJV95] dar. In einer abstrakten Klasse wird ein Algorithmus innerhalb einer so genannten Template-Methode implementiert. Einzelne seiner Schritte werden durch abstrakte Methoden definiert und in Form von Methodenaufrufen von der Template-Methode ver-

wendet. Unterklassen implementieren die verschiedenen Varianten der Algorithmusschritte.

Auf diese Weise kann der gleiche Algorithmus durch Anpassung seiner Einzelschritte in verschiedenen Situationen eingesetzt und wiederverwendet werden.

Ein Beispiel für die Spezifikation des Design Patterns ist in der Abbildung 2.6 zu sehen. Das Diagramm stellt die gesuchte Struktur wie einen Ausschnitt des abstrakten Syntaxgraphen dar (vgl. Abb. 2.3 und 2.4, S. 9/10). Der Name des Musters wird in einem Label in der oberen linken Ecke des das Diagramm umgebenden Rahmens angegeben. Ein mit dem Schlüsselwort „create“ gekennzeichneter Knoten mit dem Namen des Musters repräsentiert das zu erzeugende Annotationsobjekt. Zusätzliche Bedingungen können in geschwungenen Klammern angegeben werden. Diese werden *Constraints* genannt.

Zur Spezifikation des Design Patterns „Template Method“ enthält die abgebildete Musterregel ein UMLClass-Objekt und zwei UMLMethod-Objekte. Diese repräsentieren die abstrakte Klasse mit der Template-Methode und einer „primitiven“ abstrakten Methode, die einen Algorithmusschritt definiert.

Die Anzahl der „primitiven“ Methoden in der abstrakten Klasse wird in der Beschreibung des Design Patterns nicht festgelegt. Deswegen wird bei der Spezifikation nur eine „primitive“ Methode angegeben.

Um zu beschreiben, dass die Template-Methode die „primitive“ Methode aufruft, werden Teile des abstrakten Syntaxbaums der Template-Methode dargestellt. Mit Hilfe eines so genannten *Pfades* – dargestellt durch eine doppelte Linie mit Pfeilspitze – wird ausgedrückt, dass der Methodenrumpf – repräsentiert durch das ASTRootNode-Objekt – einen Methodenaufruf an einer beliebigen Stelle enthält. Der Aufruf wird durch das MethodCallNode-Objekt symbolisiert. Schließlich wird mit Hilfe eines Constraints spezifiziert, dass der Name der aufgerufenen Methode – repräsentiert durch das name-Attribut des IdentifierNode-Objekts – mit dem Namen der „primitiven“ Methode übereinstimmt.

2.2.3 Sprachelemente der Musterspezifikationsdiagramme

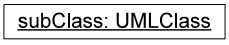
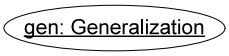
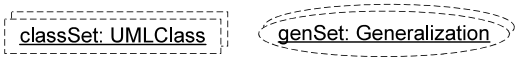
Ein Musterspezifikationsdiagramm wird von einem Rahmen umgeben dargestellt und besitzt ein Label in der oberen linken Ecke des Rahmens (siehe Beispiel in Abbildung 2.6). Dieses Label enthält das Kürzel „sp“ für *Structural Pattern*[3] und den Namen des spezifizierten Software-Musters.

Die Elemente, die in Musterspezifikationsdiagrammen vorkommen können, sind in der Tabelle 2.1 auf Seite 14 aufgelistet, um die Syntax der Diagramme zusammenzufassen. Im Folgenden werden die verschiedenen Elemente und ihre Bedeutung genauer erläutert.

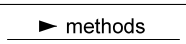
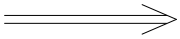
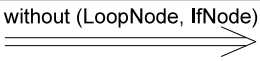
[3] Im Gegensatz zu den in diesem Kapitel beschriebenen Musterregeln, welche die Struktur eines Software-Musters beschreiben (Strukturmuster), gibt es auch Diagramme, die das Verhalten eines Software-Musters beschreiben (Verhaltensmuster). Diese werden in Form von speziellen Sequenzdiagrammen spezifiziert. Das Label dieser Diagramme wird mit dem Kürzel „bp“ für *Behavioural Pattern* gekennzeichnet.

Tabelle 2.1: Syntax von Musterspezifikationsdiagrammen

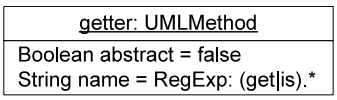
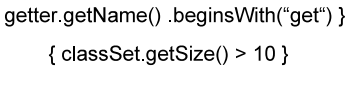
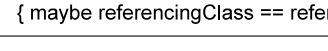
Knoten

	Objektknoten
	Annotationsknoten
	Ein mit dem Stereotyp „create“ ausgezeichneter Annotationsknoten mit Vertrauenswert
	Optionale Objekt- und Annotationsknoten
	Objektmengen- und Annotationsmengenknoten

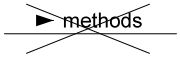
Kanten

	Verknüpfung (auch Link genannt)
	Eine mit dem Stereotyp „create“ ausgezeichnete Verknüpfung (Annotationskante)
	Pfad
	Pfad mit verbotenen ASG-Typen

Bedingungen

	Attributbedingungen zu einem Objektknoten; die zweite definiert einen regulären Ausdruck
	Constraints; enthalten beliebige Java-Ausdrücke mit Bool’schem Wert
	maybe-Constraint

Negation

	Negierte Objekt- und Annotationsknoten
	Negierte Verknüpfung

Knoten

Die Knoten eines Musterspezifikationsdiagramms können Objekte eines abstrakten Syntaxgraphen oder Annotationsobjekte repräsentieren.

Objektknoten repräsentieren Teile einer Software, die in Form eines abstrakten Syntaxgraphen vorliegt. Sie werden – wie in UML-Objektdiagrammen – als Rechteck mit darin ausgeschriebenem und unterstrichenen Typ dargestellt. Vor dem Typ und durch einen Doppelpunkt getrennt kann ein Name für das Objekt angegeben werden. Die Typen entsprechen den Klassen des ASG-Modells (siehe Abb. 2.1 auf S. 8), zum Beispiel UMLClass, UMLMethod oder MethodCallNode.

Eine Annotation, bestehend aus einem Annotationsobjekt und den zugehörigen Verknüpfungen, markiert die Fundstelle einer Musterinstanz im abstrakten Syntaxgraphen. Ein *Annotationsknoten* repräsentiert ein Annotationsobjekt und symbolisiert damit eine Instanz eines Software-Musters. Annotationsknoten werden als Ellipsen dargestellt und genauso wie Objektknoten beschriftet. Die Typen von Annotationsobjekten sind Erweiterungen des ASG-Modells und tragen den Namen der Muster, die sie repräsentieren.

Eine Musterregel beschreibt immer ein einzelnes Muster und wie eine entsprechende Fundstelle annotiert werden soll. Deswegen enthält ein Musterspezifikationsdiagramm genau einen mit dem Stereotyp „create“ ausgezeichneten Annotationsknoten. Dieser ist mit dem Namen des spezifizierten Musters beschriftet. Die von diesem Knoten ausgehenden Kanten beschreiben, welche Objekte im Falle eines Fundes mit dem erzeugten Annotationsobjekt verbunden und auf diese Weise als Fundstelle markiert werden sollen.

Annotationsknoten, die nicht mit dem Stereotyp „create“ ausgezeichnet sind, repräsentieren den Graphen, der durch die zum Knoten gehörende Musterregel beschrieben wird. Auf diese Weise ist eine Wiederverwendung und Komposition von Musterspezifikationen möglich. Kleinere, häufig wiederkehrende Muster müssen dadurch nur einmal spezifiziert werden. Außerdem wird die Übersicht in den Diagrammen erhöht, da mehrere Knoten und Kanten durch einen einzelnen Annotationsknoten ersetzt werden können.

Wenn nicht anders angegeben, sind mit dem Begriff „Annotationsknoten“ im Folgenden ausschließlich diejenigen Annotationsknoten gemeint, die nicht mit dem Stereotyp „create“ gekennzeichnet sind.

Neben einzelnen ASG- und Annotationsobjekten können auch Objektmengen durch einen einzelnen Knoten beschrieben werden. Mengen von ASG-Objekten werden durch *Objektmengenknoten* repräsentiert, während Mengen von Annotationsobjekten durch *Annotationsmengenknoten* repräsentiert werden. Die Objekte, die ein Mengenknoten repräsentiert, haben alle die gleichen Eigenschaften, insbesondere den gleichen Typ und die gleichen Beziehungen. Da eine Menge auch leer sein kann, bedeutet die Verwendung eines Mengenknotens ohne Angabe einer minimalen Anzahl der zu findenden Objekte (zum Beispiel durch ein Constraint), dass das Vorkommen eines entsprechenden Objekts für das Erkennen einer Musterinstanz nicht zwingend erforderlich ist.

Mengenknoten werden analog zu Objekt- und Annotationsknoten dargestellt, allerdings mit einem gestrichelten Rahmen und einem zweiten im Hintergrund versetzt liegenden Rechteck beziehungsweise einer zweiten Ellipse (siehe Beispiele in Tabelle 2.1).

Kanten

Zu den Kanten eines Musterspezifikationsdiagramms gehören *Verknüpfungs-* und *Pfadkanten*.

Verknüpfungskanten stellen Beziehungen zwischen Objekten dar, die durch die verknüpften Knoten repräsentiert werden. Sie werden durch eine durchgezogene Linie repräsentiert. Da Verknüpfungen immer einer Assoziation im UML-Klassendiagramm des ASG-Modells entsprechen (siehe Abb. 2.1 auf S. 8), übernehmen sie den Namen und die Leserichtung der zugehörigen Assoziation.

Die Verknüpfungskanten, die den mit „create“ gekennzeichneten Annotationsknoten mit einem anderen Knoten verbinden, werden auch *Annotationskanten* genannt. Diese sind ebenfalls mit dem Stereotyp „create“ gekennzeichnet.

Die Quelltexte von Methodenrümpfen und insbesondere der zugehörige abstrakte Syntaxbaum können sehr umfangreich sein (vgl. Abb. 2.4 auf S. 10). Außerdem können syntaktisch unterschiedliche Methodenrümpfe die gleiche Semantik besitzen. Oft reichen bereits wenige Informationen über einen Methodenrumpf zur Spezifikation eines Musters aus. Zum Beispiel wird in dem Muster in Abbildung 2.5 auf Seite 12 nur verlangt, dass der Rumpf der Template-Methode den Aufruf einer bestimmten Methode enthält, während die Struktur des Methodenrumpfes beliebig sein darf. In solchen Fällen steigt die Anzahl der zu berücksichtigenden Methodenrumpfimplementierungen und damit die Anzahl der möglichen Objektstrukturen im Methoden-AST enorm an. Um die Spezifikation von solchen Informationen zu vereinfachen und dennoch die vielen Implementierungsvarianten zu berücksichtigen, können Pfadausdrücke verwendet werden.

Pfadkanten geben an, dass ein ASG-Objekt direkt oder indirekt von einem anderen ASG-Objekt aus erreichbar ist. Dabei kann der durch den Pfad beschriebene Weg über beliebige Verknüpfungen im abstrakten Syntaxgraphen und andere ASG-Objekte führen. Pfadkanten sind gerichtet und werden durch doppelte Linien mit einer Pfeilspitze dargestellt, die in Richtung des das Zielobjekt repräsentierenden Knotens zeigt.

Mit Hilfe von Pfadkanten kann zum Beispiel die Information, dass ein Methodenrumpf den Aufruf einer bestimmten Methode enthält, auf eine einfache und kompakte Weise angegeben werden. Dazu wird eine Pfadkante von dem ASG-Objekt-Knoten, das die Wurzel des Methoden-AST repräsentiert, zu einem MethodCallNode-Knoten gezeichnet, das den Methodenaufruf beschreibt (siehe Abb. 2.6 auf S. 12). Von der konkreten Methodenrumpfimplementierung wird abstrahiert. Ob der Methodenaufruf zum Beispiel in einer if-Anweisung oder in einer Schleife stattfindet, muss nicht angegeben werden.

Um die erlaubten Wege eines Pfades einzuschränken, kann eine Liste von Ob-

jekttypen angegeben werden, die auf dem Weg zwischen Start- und Zielknoten des Pfades nicht passiert werden dürfen. Zum Beispiel kann angegeben werden, dass eine Methode nicht in einer Schleife aufgerufen wird, indem der ASG-Typ LoopNode, der ein Schleifenkonstrukt repräsentiert, auf dem Pfad zum MethodCallNode-Objekt verboten wird. Dargestellt werden solche Bedingungen, indem an die Pfadkante das Schlüsselwort „without“ gefolgt von einer Aufzählung der zu vermeidenden ASG-Typen, aufgelistet in runden Klammern und durch Kommata getrennt, geschrieben wird.

Zur Vereinfachung werden Verknüpfungs- und Pfadkanten im Folgenden schlicht Verknüpfungen beziehungsweise Pfade genannt.

Bedingungen

Zusätzlich zu den Knoten und Kanten können verschiedene Bedingungen angegeben werden, die von den ASG-Objekten einer Musterinstanz erfüllt werden müssen. Das können Attributbedingungen oder Constraints sein.

Attributbedingungen beziehen sich auf einen Attributwert eines ASG-Objekts und werden analog zur Syntax von UML-Objektdiagrammen innerhalb des Rechtecks angegeben, welches das ASG-Objekt repräsentiert. Dabei wird der Typ und der Name des Attributs gefolgt von einem Operator und einem Ausdruck, der den Wert des Attributs definiert oder einschränkt, in einer Zeile angegeben. Mögliche Operatoren sind „=“, „≠“, „>“, „≥“, „<“ und „≤“. Alternativ kann ein regulärer Ausdruck zur Überprüfung von String-Attributen angegeben werden (siehe Beispiele in Tabelle 2.1).

Constraints sind Bedingungen, die sich keinem Attribut eines ASG-Objekts eindeutig zuordnen lassen und können insbesondere zum Vergleich mehrerer Attributwerte verwendet werden. Sie bestehen aus beliebigen Bool'schen Java-Ausdrücken und sind dadurch bezüglich ihrer Ausdruckskraft mächtiger als Attributbedingungen. Constraints können Methodenaufrufe auf den ASG-Objekten enthalten, die zum Beispiel die Anzahl der zu einem Mengenknoten gefundenen Objekte zurückgeben. Auf diese Weise kann eine Mindest- oder Maximalanzahl der zu einem Mengenknoten zu findenden Objekte spezifiziert werden.

Die Semantik von Musterregeln ist so definiert, dass die Zuordnung der Regelknoten zu den Objekten des abstrakten Syntaxgraphen bei der Mustererkennung injektiv ist (vgl. Semantik von Story-Diagrammen in [Zün02, S. 147]). Das bedeutet, dass verschiedene Knoten der Regel verschiedenen Objekten im abstrakten Syntaxgraphen – Annotationsobjekte eingeschlossen – zugeordnet werden müssen. Einem Objekt kann also höchstens ein Regelknoten zugeordnet werden.

Falls diese Einschränkung unerwünscht ist, kann ein so genanntes maybe-Constraint bei der Spezifikation angegeben werden, das durch das Schlüsselwort „maybe“ gekennzeichnet wird. Im Gegensatz zu anderen Bedingungen, die in Musterregeln vorkommen können, formulieren maybe-Constraints keine Einschränkungen, sondern lockern die von den Objekten zu erfüllenden Bedingungen. Das Constraint „{maybe objA == objB}“ zum Beispiel gibt an, dass den beiden in einer Muster-

regel enthaltenen Knoten mit den Namen „objA“ und „objB“ das gleiche Objekt zugeordnet werden kann.

Negation von Knoten und Kanten

Knoten und Verknüpfungskanten können in Musterregeln negiert vorkommen, um anzugeben, dass ein ASG-Objekt mit bestimmten Eigenschaften oder eine Verknüpfung zwischen zwei Objekten im abstrakten Syntaxgraphen nicht existieren darf.

Die Negation eines Knotens wird durch zwei gekreuzte, diagonal verlaufende Linien dargestellt. Negierte Verknüpfungskanten werden analog dazu abgebildet (siehe Tabelle 2.1 auf Seite 14).

Optionale Knoten

Neben der Negation von Knoten ist es möglich, einen Knoten als optional zu kennzeichnen. Auf diese Weise wird angegeben, dass ein Objekt, das durch einen optionalen Knoten repräsentiert wird, für die Erkennung des spezifizierten Musters nicht gefunden werden muss. Seine Existenz dagegen soll den Fund einer Musterinstanz untermauern. Optionale Knoten können nicht negiert werden.

Als optisches Merkmal wird der Rahmen von optionalen Knoten gestrichelt dargestellt.

Vertrauenswerte

Wie in dem Abschnitt 1.1 beschrieben, ist bei der Spezifikation von Software-Mustern meist eine Vielzahl von Implementierungsvarianten zu berücksichtigen. Um nicht jede mögliche Implementierungsvariante zu spezifizieren, werden mehrere Varianten zusammengefasst, indem nur ihre Gemeinsamkeiten in einer einzelnen Spezifikation beschrieben werden. Dadurch werden die Spezifikationen unvollständig. Eine Konsequenz daraus ist der Verlust von Präzision bei der Mustererkennung. Entscheidungen, ob eine Objektstruktur die Instanz eines spezifizierten Software-Musters ist, können nicht mit Sicherheit getroffen werden. Um solche absoluten Aussagen zu vermeiden und die Unsicherheit auszudrücken, werden die Musterinstanzen bewertet.

Die Bewertung der Musterinstanzen basiert auf Schätzungen der Qualität der zugehörigen Musterregel (siehe Abschnitt 3.1 ab Seite 27). Dazu gibt der Reverse Engineer bei der Spezifikation eines Musters einen so genannten Vertrauenswert an. Dieser sagt aus, wie groß der Anteil der korrekten Musterfunde an der Gesamtanzahl der Funde zu dieser Spezifikation sein wird.

Der Vertrauenswert wird bei jeder Spezifikation in dem mit „create“ gekennzeichneten Annotationsknoten unterhalb des Typs beziehungsweise des Musternamens angegeben.

2.2.4 Musterregelabhängigkeiten

Software-Muster werden meist auf einem hohen Abstraktionslevel beschrieben. Um ihre Struktur zu definieren, werden häufig Referenzen, Assoziationen und Delegationen verwendet. Um solche und andere kleinere wiederkehrende Strukturen nicht bei jeder Musterspezifikation erneut zu definieren, werden sie als Hilfsmuster – auch *Cliché* genannt [Wen01] – einzeln spezifiziert und in anderen Spezifikationen wiederverwendet.

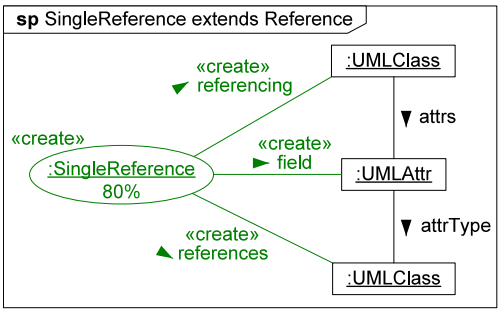


Abbildung 2.7: Musterregel für zu-1-Referenzen

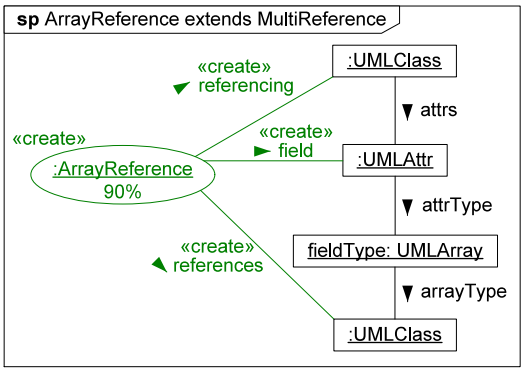


Abbildung 2.8: Musterregel für zu-n-Referenzen, implementiert mit einem Array

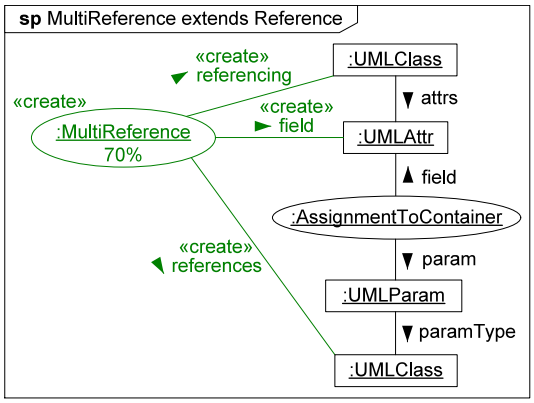


Abbildung 2.9: Musterregel für zu-n-Referenzen, implementiert mit einer Container-Klasse

Referenzen, Assoziationen und Delegationen können auf viele Weisen implementiert werden. Zum Beispiel gibt es bei Referenzen die Unterscheidung zwischen so genannten zu-1- und zu-n-Referenzen. Zu-1-Referenzen können höchstens ein Objekt und zu-n-Referenzen können beliebig viele Objekte referenzieren. Außerdem können zu-n-Referenzen mit Hilfe von Arrays oder Container-Klassen implemen-

tiert werden. Um Referenzen erkennen zu können, müssen für diese Implementierungsvarianten verschiedene Musterregeln spezifiziert werden. Dazu wurden die Regeln „SingleReference“, „MultiReference“ und „ArrayReference“ erstellt (siehe Abbildungen 2.7 bis 2.9).

Eine bidirektionale Assoziation kann durch zwei sich gegenseitig referenzierende Klassen beschrieben werden. Bei der Spezifikation können also die Musterregeln für Referenzen in Form von zwei Annotationsknoten verwendet werden, wobei jeder der Knoten eine der beiden Referenzen repräsentieren würde.

Die Art der Referenzen ist bei der Beschreibung einer beliebigen Assoziation irrelevant. Da jedoch keine Musterregel existiert, die sämtliche Implementierungsvarianten für Referenzen zusammenfasst und dadurch kein Annotationsknoten bei der Spezifikation einer Assoziation angegeben werden kann, der eine beliebige Referenz repäsentiert, müsste zur Spezifikation einer Assoziation für jede Kombination aus je zwei der drei Referenzmusterregeln (zum Beispiel „SingleReference“ und „ArrayReference“ oder „SingleReference“ und „SingleReference“) je eine Assoziationsmusterregel erstellt werden.

Vererbung

Um mehrere semantisch verwandte Musterregeln durch einen Annotationsknoten repräsentieren zu können, wurde das Prinzip der Vererbung zwischen Musterregeln eingeführt. Erbt eine Musterregel von einer anderen, so beschreibt sie einen spezielleren Fall bzw. ein spezielleres Muster. Ein Annotationsknoten zu einer Musterregel, von der andere erben, kann polymorph eingesetzt werden. Das bedeutet, dass es die in der zugehörigen Musterregel oder die in einer der erbenden Musterregeln beschriebene Objektstruktur repräsentiert.

Es ist nur Einfachvererbung möglich. Erbt eine Musterregel von einer anderen, so erhält sie die gleiche Schnittstelle wie die Elternregel. Das heißt, sie muss die gleichen Annotationskanten besitzen und Objekte vom selben Typ annotieren, es können aber auch zusätzliche Annotationskanten hinzugefügt werden. Die übrige Struktur kann von der der Elternregel beliebig abweichen.

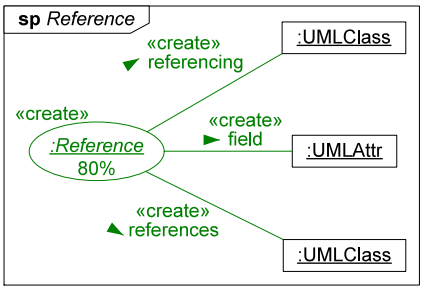


Abbildung 2.10: Abstrakte Musterregel „Reference“

In dem beschriebenen Beispiel wurde eine Musterregel namens „Reference“ wie in Abbildung 2.10 spezifiziert, von der die Regeln „SingleReference“, „MultiRe-

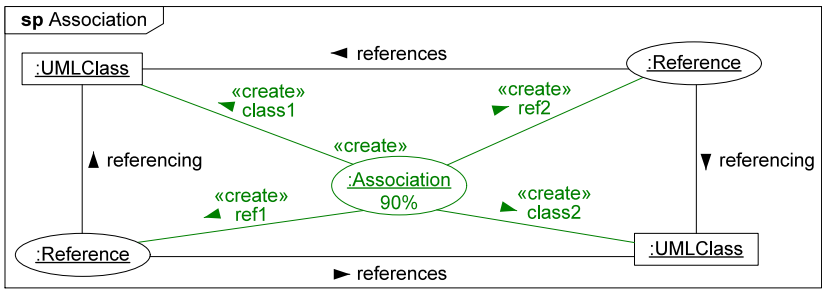


Abbildung 2.11: Musterregel für Assoziationen

ference“ und „ArrayReference“ erben. Ein Annotationsknoten mit dem Typ Reference repräsentiert dadurch eine beliebige der drei Implementierungsvarianten. Eine Assoziation kann somit mit nur einer Musterregel wie in Abbildung 2.11 spezifiziert werden.

Die Angabe, dass eine Musterregel von einer anderen erbt, erfolgt ähnlich wie bei Klassen in objektorientierten Programmiersprachen wie Java. In dem Label eines Musterspezifikationsdiagramms wird hinter dem Namen der erbenden Musterregel das Schlüsselwort „extends“ gefolgt von dem Namen der Regel, von der geerbt wird, angegeben. Um zum Beispiel anzugeben, dass die Musterregel „SingleReference“ von der Regel „Reference“ erbt, enthält das Label des Diagramms in Abbildung 2.7 auf Seite 19 den Ausdruck „SingleReference extends Reference“.

Um zu kennzeichnen, dass es keine konkreten Musterinstanzen zu einer Spezifikation geben soll, kann die Musterregel als abstrakt markiert werden. Abstrakte Musterregeln definieren eine Schnittstelle, also die Annotationskanten und die Typen der zu annotierenden Objekte, aber keine Objektstruktur oder Objekteigenschaften. Zur Hervorhebung wird der Name einer abstrakten Regel kursiv dargestellt.

Die Musterregel „Reference“ wird als abstrakt gekennzeichnet, da konkrete Implementierungsvarianten bei diesem Beispiel nur durch die Regeln „SingleReference“, „MultiReference“ und „ArrayReference“ beschrieben werden.

Musterregelabhängigkeitsgraph

So genannte *Musterregelabhängigkeitsgraphen* (engl. *pattern rules dependency net*) sollen einen Überblick über die Beziehungen der Regeln untereinander verschaffen und stellen unter anderem Vererbungsbeziehungen dar. Ein solcher Graph ist in der Abbildung 2.12 zu sehen.

Musterregeln werden hier durch Rechtecke mit dem Stereotyp „pattern“ symbolisiert. Ihr Vertrauenswert wird unterhalb des Namens angegeben. Vererbungsbeziehungen zwischen Regeln werden wie in UML-Klassendiagrammen durch einen Pfeil mit einer geschlossenen, nicht ausgefüllten Pfeilspitze dargestellt, der in Richtung der Elternregel zeigt.

Außer der Musterregeln können Musterregelabhängigkeitsgraphen auch so ge-

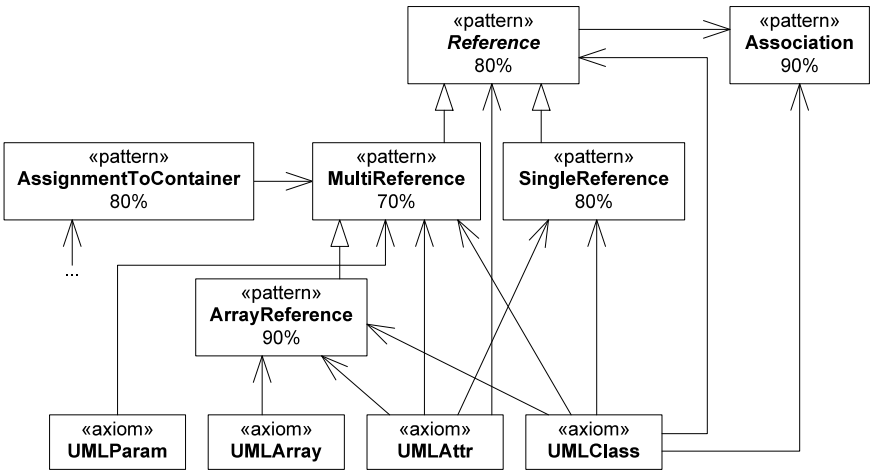


Abbildung 2.12: Musterregelabhängigkeitsgraph zu den Regeln in den Abbildungen 2.7 bis 2.11

nannte *Axiome* enthalten, die mit dem Stereotyp „axiom“ gekennzeichnet werden. Das sind Typen von Objekten aus abstrakten Syntaxgraphen. ASG-Objekte müssen nicht erkannt werden. Sie können direkt als Fakten bei der Mustererkennung verwendet werden.

In Musterregelabhängigkeitsgraphen werden neben der Vererbungsbeziehungen auch Abhängigkeiten zwischen Regeln beziehungsweise zwischen Regeln und Axiomen dargestellt. Eine Abhängigkeit zwischen zwei Musterregeln besteht, wenn eine Regel eine andere wiederverwendet. In solchen Fällen muss erst die Objektstruktur der wiederverwendeten Regel gefunden und annotiert werden, bevor das durch die wiederverwendende Regel beschriebene Muster erkannt werden kann. Abhängigkeiten werden durch einen Pfeil mit offener Pfeilspitze symbolisiert, der in Richtung der wiederverwendenden Regel zeigt.

Der Graph in Abbildung 2.12 stellt die Vererbungsbeziehungen der Musterregeln für Referenzen dar. Außerdem werden die Abhängigkeiten der Regel „Association“ zur Regel „Reference“ und der Regel „MultiReference“ zu „AssignmentToContainer“ sichtbar. Zusätzlich werden die Abhängigkeiten der Regeln zu Axiomen dargestellt, zur besseren Übersicht wurden diese aber für die Regel „AssignmentToContainer“ weggelassen.

2.3 Mustersuche

Software-Muster werden zur Mustererkennung mit Fujaba formal in Form von Musterregeln beschrieben. Bei der Mustererkennung wird versucht, zu jedem Knoten einer Musterregel ein geeignetes Objekt aus dem abstrakten Syntaxgraphen des zu untersuchenden Softwaresystems zu finden, sodass alle in der Regel spezifizierten Bedingungen erfüllt sind. Konnten auf diese Weise allen Knoten einer

Regel Objekte zugeordnet werden[4], so ist eine Objektstruktur im abstrakten Syntaxgraphen gefunden, die eine Instanz des spezifizierten Musters repräsentiert. Die Zuordnung der Objekte zu den Regelknoten wird als *Matching* bezeichnet. Die Abbildung 2.13 stellt beispielhaft ein Matching für das Design Pattern „Template Method“ dar. Die Zuordnungen der Objekte zu den Knoten werden durch dicke Pfeile dargestellt.

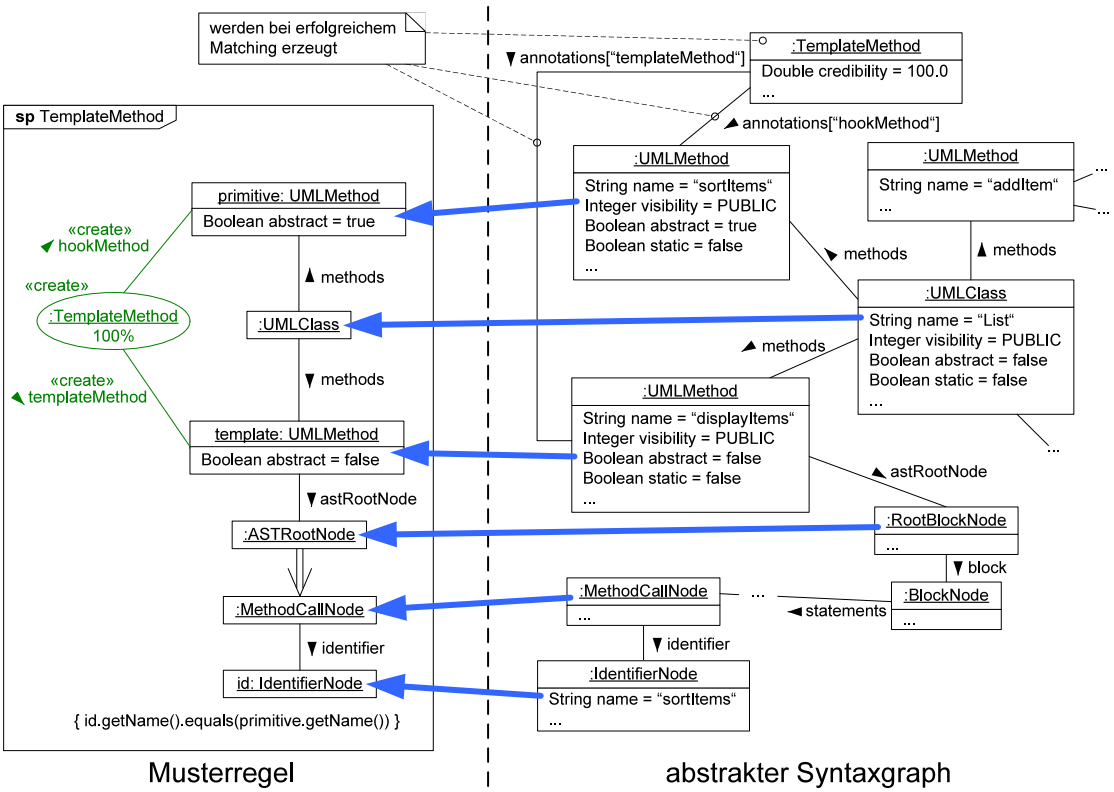


Abbildung 2.13: Beispiel für ein Matching

Die Suche nach Mustern wird in Fujaba durch einen speziellen Inferenzalgorithmus realisiert, der den abstrakten Syntaxgraphen eines zu untersuchenden Softwaresystems mit Hilfe von so genannten *Annotationsmaschinen* analysiert. Dabei werden die Fundstellen der erkannten Musterinstanzen durch Annotationen markiert, die unter anderem in UML-Klassendiagrammen des untersuchten Systems dargestellt werden können.

Eine Annotationsmaschine ist die Implementierung einer Musterregelanwendung, welche aus zwei Teilen besteht. Der erste Teil ist die Suche nach einer geeigneten Anwendungsstelle, also die Suche nach der in der Regel definierten Objektstruktur in einem beliebigen abstrakten Syntaxgraphen. Wird die Struktur gefunden, so kommt der zweite Teil zur Ausführung und es wird das laut der

[4]Ist ein Knoten optional, so müssen diesem keine Objekte zugeordnet werden. Einem Mengenknoten können beliebig viele Objekte zugeordnet werden.

Regel zu erzeugende Annotationsobjekt sowie zugehörige Verknüpfungen erstellt. Auf diese Weise wird der abstrakte Syntaxgraph annotiert.

Das Suchen nach einer Anwendungsstelle für eine Musterregel entspricht dem NP-vollständigen Problem der Teilgraphensuche (vgl. [Bac01, Kapitel 3.2.3 *Subgraph Isomorphie*]). Durch Angabe eines Startknotens, welcher ein Teil des gesuchten Teilgraphen ist, kann laut [Nie04, S. 83] die durchschnittliche Laufzeit für die Teilgraphensuche auf polynomielle Zeit reduziert werden. Diese Eigenschaft wird bei der Mustersuche in Fujaba ausgenutzt.

Der Startknoten bei der Suche nach einer Anwendungsstelle für eine Musterregel ist ein Objekt aus dem zu untersuchenden abstrakten Syntaxgraphen. Dieses hat den Typ, der durch einen der Regelknoten angegeben ist, und wird *Kontext* genannt. Ausgehend von diesem Objekt werden die anderen in der Regel beschriebenen Objekte gesucht und ihre Eigenschaften auf Konformität mit den spezifizierten Bedingungen überprüft. Die Suche erfolgt dabei mit Hilfe eines Backtracking-Algorithmus, der den durch das Kontextobjekt repräsentierten Teilgraphen Objekt für Objekt zu dem in der Musterregel spezifizierten Graphen zu komplettieren versucht.

Um bei der Mustererkennung den Aufwand für den Reverse Engineer zu minimieren, wird der Java-Code für die Annotationsmaschinen automatisch aus Musterregeln generiert und kompiliert. Dabei entsteht für jede nicht abstrakte Regel je eine Annotationsmaschine.

Der Inferenzalgorithmus bekommt einen abstrakten Syntaxgraphen sowie die Annotationsmaschinen als Eingabe. Dieser geht schrittweise vor und prüft in jedem Schritt, ob eine der Annotationsmaschinen mit einem der Knoten des abstrakten Syntaxgraphen als Eingabe erfolgreich ausgeführt, also eine der Musterregeln angewendet und eine Annotation erstellt werden kann.

Eine bei der Anwendung einer Musterregel entstandene Annotation kann Voraussetzung für die erfolgreiche Anwendung einer anderen Musterregel sein. In so einem Fall besteht eine Abhängigkeit zwischen den Musterregeln (vgl. Abschnitt 2.2.4 und Abb. 2.14). Außerdem führen Vererbungsbeziehungen zwischen Musterregeln dazu, dass Annotationsknoten polymorph verwendet werden können. Die Wahl der im nächsten Schritt anzuwendenden Regel wird vom Inferenzalgorithmus unter Berücksichtigung der Regelabhängigkeiten und Vererbungsbeziehungen getroffen.

Zur Auswahl einer Regel wurden zwei Strategien entwickelt [Wen01, Nie04], die Regeln zu möglichst abstrakten Mustern[5] bevorzugen, um so bei dem Inferenzalgorithmus möglichst frühzeitig aussagekräftige Zwischenergebnisse zu produzieren. Auf diese Weise hat der Reverse Engineer die Möglichkeit, einige Ergebnisse zu beurteilen bevor die Suche abgeschlossen ist und bei Bedarf in die Mustererkennung einzugreifen.

Aussagekräftige Zwischenergebnisse können zum Beispiel Entwurfsmuster sein. Bei ihrer Spezifikation werden meist Hilfsmuster, zum Beispiel für Vererbung,

[5]Muster auf einem hohen Abstraktionslevel, nicht aber abstrakte Musterregeln

Assoziationen und Delegation verwendet (siehe Abb. 2.14). Ein Reverse Engineer ist aber weniger an den Hilfsmustern als an den Entwurfsmustern interessiert.

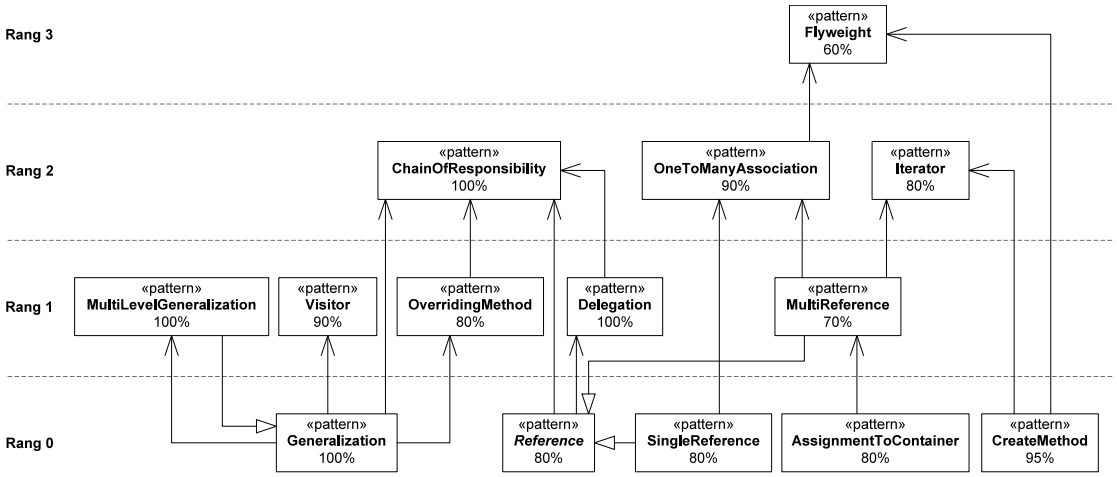


Abbildung 2.14: Musterregelabhängigkeitsgraph zu einigen Entwurfs- und Hilfsmusterspezifikationen mit errechneten Rängen

Die Verwendung von Hilfsmustern bei der Spezifikation eines Musters deutet auf ein abstrakteres Muster hin. Deswegen werden Musterregeln bevorzugt, die bei einer topologischen Sortierung aller Regeln und Axiome bezüglich ihrer Abhängigkeiten möglichst weit weg von den Axiomen liegen. In [Nie04, S. 89] wird zur Beurteilung von Musterregeln ein *Rang* definiert. Der Musterregelabhängigkeitsgraph in Abbildung 2.14 zeigt die Abhängigkeiten einiger der an der Universität Paderborn erstellten Musterregeln für Entwurfsmuster aus [GHJV95]. Für diese Teilmenge der Regeln wurden die Ränge nach der Definition in [Nie04, S. 89] berechnet. Die Axiome wurden bei der Abbildung zur besseren Übersicht weggelassen.

Bei dem Beispiel in Abbildung 2.14 sind die Entwurfsmuster „Chain of Responsibility“, „Flyweight“, „Iterator“ und „Visitor“ vertreten. Die übrigen Muster sind Hilfsmuster. Dem Musterregelabhängigkeitsgraphen kann man entnehmen, dass die Ränge der Entwurfsmusterregeln höher sind als die Ränge der meisten Hilfsmusterregeln.

Eine ausführliche Beschreibung der verschiedenen bisher entwickelten Inferenzalgorithmen befindet sich in [Wen01] und [Nie04].

3 Existierende Bewertungsverfahren

Heute existieren viele Ansätze zur automatischen oder semi-automatischen Erkennung von Software-Mustern. Die meisten haben das Ziel, Entwickler beim Reverse Engineering zu unterstützen.

Die bei den Ansätzen verwendeten Methoden zur Beschreibung und Suche von Software-Mustern sind zum Teil sehr unterschiedlich. Es werden zum Beispiel Prädigatenlogik [KP96, FM04, BBS05], relationale Algebra [BL03] oder Constraint Solver [GJ01, AACGJ01] eingesetzt. Neben der Analyse der strukturellen Eigenschaften der Software-Komponenten werden Software-Metriken dazu benutzt, den Suchraum zu verkleinern [AFC98]. In einigen Fällen wird eine Teilgraphensuche auf der Graphrepräsentation einer Software [NSW+02, FBFL05] durchgeführt. Um auf Basis von speziellen Metriken die Anzahl der False Positives zu verkleinern, wird maschinelles Lernen in Form von Entscheidungsbäumen oder neuronalen Netzen verwendet [FBFL05].

Bis auf einige wenige Ausnahmen wird bei allen genannten Mustererkennungsansätzen nur nach den Musterausprägungen gesucht, die vollständig mit der Musterspezifikation übereinstimmen. Es werden nur Entscheidungen darüber getroffen, ob ein Muster erkannt wurde oder nicht. Die Suchergebisse werden nicht bewertet.

Bei den drei Ansätzen [NWW03], [Wen05c] und [GJ01] dagegen wird eine Aussage über die Qualität der erkannten Musterinstanzen gemacht. Bei dem Ansatz aus [NWW03] werden ebenfalls nur vollständige Musterinstanzen gesucht, diese werden aber basierend auf Schätzungen der Genauigkeit von Musterspezifikationen beurteilt. Auf diese Weise wird die Verlässlichkeit der Suchergebnisse ausgedrückt. Im Gegensatz dazu wird in [Wen05c] und [GJ01] unter anderem auch nach unvollständigen Musterinstanzen gesucht. Diese werden nach ihrem Vollständigkeitsgrad beziehungsweise ihrer Ähnlichkeit zum spezifizierten Muster bewertet.

In den folgenden Abschnitten dieses Kapitels werden die Bewertungsverfahren aus [NWW03] und [Wen05c] im Detail vorgestellt und beurteilt. Wie Musterinstanzen nach dem Ansatz aus [GJ01, AACGJ01] bewertet werden, konnte den Veröffentlichungen leider nicht entnommen werden.

3.1 Schätzung der Präzision einer Musterspezifikation

Bei der Erkennung von Software-Mustern sind zahlreiche Implementierungsvarianten zu berücksichtigen (siehe Abschnitt 1.1). Um bei der Mustererkennung mit

Fujaba nicht für jede der Varianten je eine Musterregel spezifizieren zu müssen, werden bei dem Ansatz [NWW03] mehrere Varianten in einer Regel zusammengefasst. Dazu werden ausschließlich die Gemeinsamkeiten der Varianten spezifiziert. Solche unvollständigen Spezifikationen führen zum Verlust von Präzision bei der Mustererkennung. False Positives – also Objektstrukturen im abstrakten Syntaxgraphen, die fälschlicherweise als Musterinstanzen markiert werden – können nicht ausgeschlossen werden. Um dem Reverse Engineer einen Anhaltspunkt zur Verlässlichkeit der Suchergebnisse zu geben, werden die Musterinstanzen bewertet.

3.1.1 Bewertung von Musterinstanzen

Die Grundlage für die Bewertung von Musterinstanzen bilden Schätzungen der Genauigkeit von Musterregeln beziehungsweise der Verlässlichkeit der zugehörigen Suchergebnisse. Die Schätzungen werden vom Reverse Engineer in Form von so genannten Vertrauenswerten (engl. *credibility value*) für jede Regel abgegeben.

Der Vertrauenswert einer Musterregel trifft eine Aussage darüber, wie groß der Anteil der von der Regel korrekterweise erstellten Annotationen – so genannter *True Positives* – zu der Gesamtanzahl der von der Regel erstellten Annotationen ist und wird in Prozent angegeben. Der Vertrauenswert für eine Regel r kann durch folgende Formel beschrieben werden:

$$Vertrauenswert(r) \quad = \quad \frac{|TruePositives(r)|}{|TruePositives(r)| + |FalsePositives(r)|}$$

Die Verlässlichkeit der Suchergebnisse wird durch so genannte Genauigkeitswerte (engl. *accuracy value*) ausgedrückt. Für jede erstellte Annotation wird jeweils ein solcher Wert bestimmt. Die Berechnung der Genauigkeitswerte soll im Folgenden an einem Beispiel erklärt werden.

Das Entwurfmuster „Iterator“ liefert eine Möglichkeit, zum Beispiel den Inhalt eines Container-Objekts mit Hilfe einer speziellen Klasse zu durchlaufen, ohne die interne Struktur des Container-Objekts zu kennen. Eine entsprechende Musterregel ist in der Abbildung 3.1 dargestellt.

Die Regel beschreibt mit dem aggregate-Knoten eine Klasse mit einer zu-n-Referenz zu einer anderen Klasse. Diese wird durch den iteratedType-Knoten symbolisiert. Das Vorhandensein der Referenz wird durch den Annotationsknoten mit dem Typ MultiReference ausgedrückt. Der iteratedType-Knoten steht für den Typ der zu durchlaufenden Objekte. Die Aggregatklasse enthält eine Methode, die eine Instanz der durch den iterator-Knoten repräsentierten Klasse erstellt und zurückgibt, was durch das Hilfsmuster „CreateMethod“ und den zugehörigen Annotationsknoten beschrieben wird. Die „Iterator“-Klasse wiederum bietet eine Methode, welche als Rückgabetyp die durch den iteratedType-Knoten repräsentierte Klasse hat. Diese Methode ist für das Durchlaufen der referenzierten Objekte zuständig und gibt bei jedem Aufruf das jeweils nächste Objekt zurück.

Die Berechnung der Genauigkeitswerte erfolgt nach der Mustererkennung. Der abstrakte Syntaxgraph des zu untersuchenden Softwaresystems ist zu diesem Zeit-

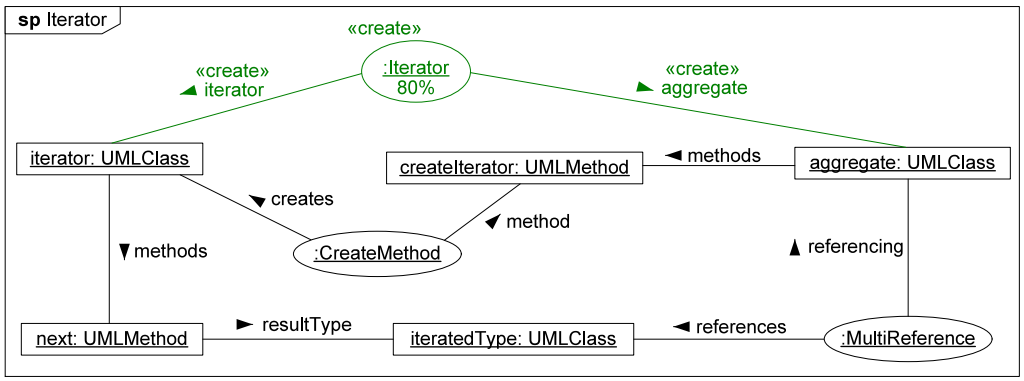


Abbildung 3.1: Musterregel zum Entwurfsmuster „Iterator“

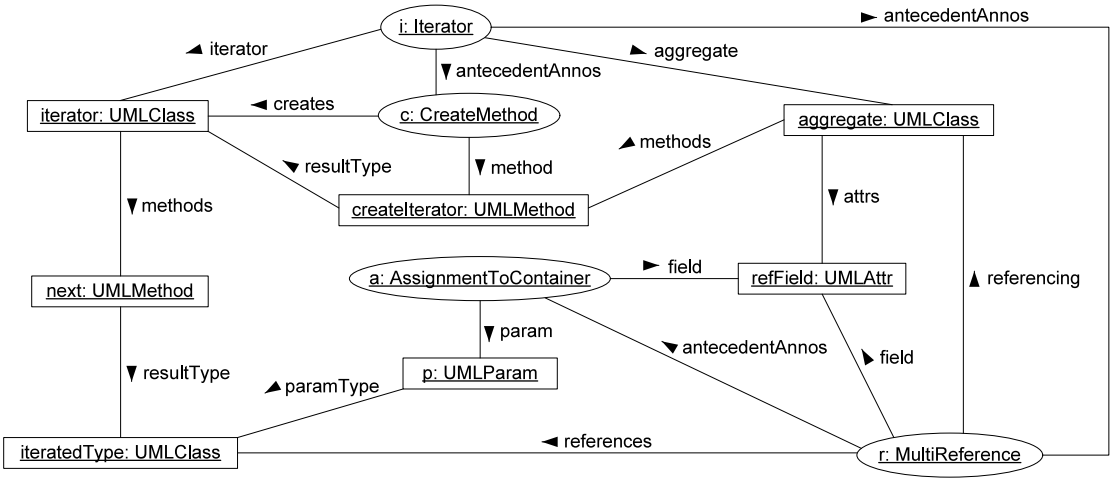


Abbildung 3.2: Ausschnitt eines annotierten abstrakten Syntaxgraphen

punkt bereits annotiert und könnte wie in der Abbildung 3.2 aussehen. Hier sind unter anderem die für die „Iterator“-Musterregel notwendigen Annotationen vom Typ CreateMethod und MultiReference zu sehen. Die Regel „MultiReference“ hängt von der Regel „AssignmentToContainer“ ab (siehe Abb. 2.9 auf S. 19), auch dazu ist ein entsprechendes Annotationsobjekt vorhanden.

Die Bewertung einer Musterinstanz erfolgt abhängig von ihren Vorgängermusterinstanzen. Zu diesem Zweck werden beim Erstellen eines Annotationsobjekts antecedentAnnos-Verknüpfungen zu allen verwendeten Vorgängerannotationsobjekten erstellt. Auch diese Verknüpfungen sind bei dem Beispiel in der Abbildung 3.2 zu sehen.

Fuzzy-Petrinetz

Unter Berücksichtigung der Abhängigkeiten zwischen den Annotationen – beschrieben durch antecedentAnnos-Verknüpfungen – wird zur Berechnung der Genauigkeitswerte ein so genanntes *Fuzzy-Petrinetz* (kurz: FPN) erstellt. Dieses

enthält für jedes Annotationsobjekt im abstrakten Syntaxgraphen je eine Stelle. Ist ein Annotationsobjekt über antecedentAnnos-Verknüpfungen zu Vorgängern verbunden, so erhält das Fuzzy-Petrinetz eine Transition, welche die zum Annotationsobjekt gehörende Stelle im Nachbereich und die Stellen sämtlicher Vorgänger im Vorbereich hat.

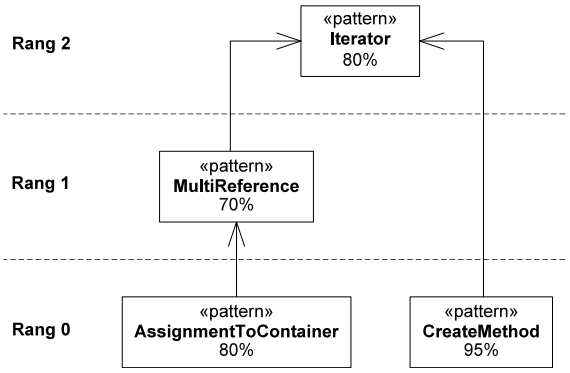


Abbildung 3.3: Musterregelabhängigkeitsgraph für das „Iterator“-Muster

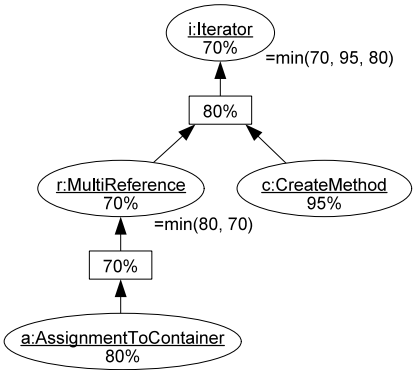


Abbildung 3.4: Ein Fuzzy-Petrinetz zum abstrakten Syntaxgraphen in Abb. 3.2

Ein Beispiel für ein Fuzzy-Petrinetz ist in der Abbildung 3.4 zu sehen. Dieses stellt die Annotationen des abstrakten Syntaxgraphen in Abbildung 3.2 zu einander in Beziehung. Die Annotationen c vom Typ CreateMethod und r vom Typ MultiReference in der Abbildung 3.2 sind direkte Vorgänger der Iterator-Annotation i. Ebenso ist die Annotation a vom Typ AssignmentToContainer direkter Vorgänger der MultiReference-Annotation r. Für jede dieser Vorgänger-Beziehungen ist in dem Fuzzy-Petrinetz in der Abbildung 3.4 eine Verbindung zwischen den zugehörigen Stellen eingezeichnet.

Die Berechnung der Genauigkeitswerte für Musterinstanzen beziehungsweise Annotationen erfolgt durch Ausführung des Fuzzy-Petrinetzes. Dabei werden die errechneten Werte durch Markieren der Stellen festgehalten. Initial werden die Stellen der Annotationen von Rang-0-Musterregeln – bei dem Beispiel sind das „AssignmentToContainer“ und „CreateMethod“ (siehe Abb. 3.3) – mit den Vertrauenswerten der Regeln markiert und alle anderen mit dem Wert 0 (zur Beschreibung von Rängen siehe Abschnitt 2.3). Beim Ausführen des Fuzzy-Petrinetzes werden die Markierungen abhängig von den Vorgängern im Fuzzy-Petrinetz berechnet. Eine Stelle wird dabei mit dem Minimum aus den Markierungen der Vorgängerstellen und dem Vertrauenswert der zur Stelle gehörenden Musterregel markiert.

Die Ausführung des Fuzzy-Petrinetzes ist beendet, wenn das Netz stabil ist, das heißt, wenn sich keine Markierung mehr ändert. Die Markierung einer Stelle in einem stabilen Fuzzy-Petrinetz ist der errechnete Genauigkeitswert der zugehörigen Annotation beziehungsweise der Musterinstanz.

Zur Veranschaulichung des Verfahrens sind die Stellen des Fuzzy-Petrinetzes in der Abbildung 3.4 mit den Namen der zugehörigen Annotationsobjekte aus Abbildung 3.2 und den errechneten Genauigkeitswerten markiert, während die Transitionen mit dem Vertrauenswert der zur Stelle im Nachbereich gehörenden Musterregel gekennzeichnet sind.

Der Genauigkeitswert des Annotationsobjekts r ergibt sich durch Minimumbildung aus dem Genauigkeitswert des Annotationsobjekts a (80%) und dem Vertrauenswert der Musterregel „MultiReference“ (70%). Der Genauigkeitswert für das Annotationsobjekt i ergibt sich durch Minimumbildung aus den Genauigkeitswerten der Annotationsobjekte r (70%) und c (95%) sowie dem Vertrauenswert der Musterregel „Iterator“ (80%).

3.1.2 Adaption der geschätzten Präzisionswerte

Musterinstanzen werden bei dem im Abschnitt 3.1.1 vorgestellten Verfahren auf Basis von Vertrauenswerten der zugehörigen Musterregeln bewertet. Vertrauenswerte machen eine Aussage über die Verlässlichkeit der Suchergebnisse zu einer Musterregel und werden vom Reverse Engineer bei der Spezifikation eines Musters geschätzt.

Um die initial geschätzten Werte und die Bewertung von Musterinstanzen präziser zu machen, hat der Reverse Engineer die Möglichkeit, einen Mustererkennungsprozess jederzeit zu unterbrechen, die Zwischenergebnisse zu untersuchen und die Vertrauenswerte der Musterregeln gegebenenfalls anzupassen[1]. Die Vertrauenswerte bleiben jedoch Schätzungen und damit unpräzise, außerdem ist die manuelle Korrektur der Vertrauenswerte aufwendig.

Im Rahmen einer Diplomarbeit [Rec04] wurde ein heuristisches Verfahren zur halb-automatischen Adaption der Vertrauenswerte entwickelt. Bei diesem Verfahren korrigiert der Reverse Engineer die errechneten Genauigkeitswerte von Musterinstanzen anstatt die Vertrauenswerte der Musterregeln. Ausgehend von der Annahme, dass die vom Reverse Engineer geänderten Genauigkeitswerte präziser sind als die errechneten, werden diese Korrekturen als Basis für die Anpassung der Vertrauenswerte genommen. Die Änderung der Werte erfolgt mit Hilfe eines statistischen Verfahrens, bei dem die Vertrauenswerte schrittweise an ihren Idealwert[2] angenähert werden. Eine vereinfachte Form dieses Verfahrens [NMW04, Nie04] wird im Folgenden vorgestellt.

Die Genauigkeitswerte werden mit Hilfe eines Fuzzy-Petrinetzes berechnet. Dieses enthält zu jeder Musterinstanz beziehungsweise Annotation je eine Stelle und spiegelt die Abhängigkeiten der Annotationen wieder. Wird der Genauigkeitswert einer Annotation geändert, so wird der neue Wert als so genannter *virtueller Genauigkeitswert* in einer Menge $\widehat{G}_s$ der zugehörigen Stelle s im Fuzzy-Petrinetz

[1] Außer den Vertrauenswerten kann auch die Struktur einer Musterregel geändert werden.

[2] Nach dem Prinzip der Regression ist der Vertrauenswert der Regel r ideal, wenn die Summe $\sum_a (g_a - g'_a)^2$ über alle zu r erstellten Annotationen a der quadrierten Differenzen zwischen dem errechneten Genauigkeitswert g_a und dem geänderten Genauigkeitswert g'_a minimal ist.

gespeichert. Nachdem der Reverse Engineer alle seine Änderungen vorgenommen hat, werden die Vertrauenswerte auf Basis der virtuellen Genauigkeitswerte angepasst.

Die Details bei der Berechnung sollen an dem Fuzzy-Petrinetz aus Abbildung 3.4 auf Seite 30 erläutert werden. Die Abbildungen 3.5 und 3.6 stellen eine Erhöhung und eine Verringerung eines Genauigkeitswertes dar. Die Stellen sind mit den Genauigkeitswerten aus Abbildung 3.4 markiert.

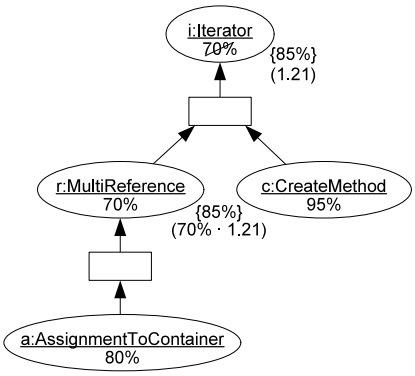


Abbildung 3.5: Erhöhung des Genauigkeitswerts am Beispiel

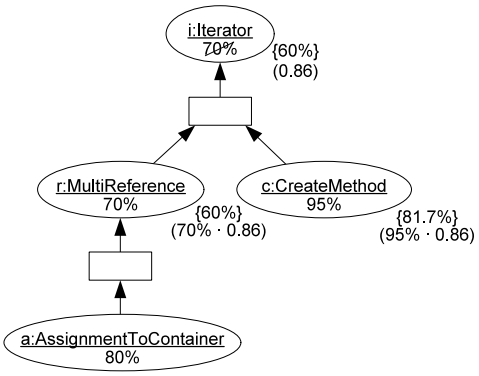


Abbildung 3.6: Verringerung des Genauigkeitswerts am Beispiel

Wird der Genauigkeitswert der Annotation i wie in Abbildung 3.5 von 70% auf 85% erhöht, so wird der neue Wert in der Menge $\widehat{G}_s$ der zugehörigen Stelle s im Fuzzy-Petrinetz gespeichert. Um den neuen Genauigkeitswert auch mit dem Fuzzy-Petrinetz berechnen zu können, müssen die Genauigkeitswerte der Vorgängerannotationen erhöht werden. Aus diesem Grund wird die relative Änderung des Genauigkeitswertes – in diesem Fall mit Faktor $\frac{85}{70} \approx 1.21$ – in Form weiterer virtuellen Werte auf die direkten Vorgänger der Annotation übertragen, bei denen der errechnete Genauigkeitswert kleiner als der neue Wert 85% ist. Bei dem Beispiel in Abbildung 3.5 wird also der Stelle zur Annotation r der virtuelle Genauigkeitswert $70\% \cdot 1.21 \approx 85\%$ hinzugefügt. In Fällen, bei denen der errechnete virtuelle Genauigkeitswert größer als 100% ergeben würde (zum Beispiel bei $90\% \cdot 1.21 \approx 109\%$), wird dieser auf 100% reduziert und dadurch beschränkt.

Wenn ein Genauigkeitswert wie in Abbildung 3.6 von 70% auf 60% verringert wird, so werden nur für diejenigen Vorgängerannotationen virtuelle Werte berechnet, bei denen der errechnete Genauigkeitswert größer ist als der neue Wert 60%. Das übrige Vorgehen ist analog.

Nachdem der Reverse Engineer alle erwünschten Änderungen an den Genauigkeitswerten vorgenommen hat, startet er die automatische Berechnung der neuen Vertrauenswerte, die in zwei Schritten erfolgt.

Beim ersten Schritt wird für jede Stelle s der durchschnittliche virtuelle Genauigkeitswert $\overline{g_s}$ bestimmt. Falls keine virtuellen Werte existieren, wird der zuvor

errechnete Genauigkeitswert g_s verwendet:

$$\overline{g_s} = \begin{cases} \frac{1}{|\widehat{G}_s|} \cdot \sum_{\widehat{g} \in \widehat{G}_s} \widehat{g} & \text{falls } \widehat{G}_s \neq \emptyset \\ g_s & \text{sonst} \end{cases}$$

Im zweiten Schritt wird der neue Vertrauenswert v'_r einer Musterregel r berechnet. Dazu wird der Durchschnitt aus dem für r spezifizierten Vertrauenswert v_r und den durchschnittlichen virtuellen Genauigkeitswerten $\overline{g_s}$ aller Stellen $s \in \mathit{Stellen}(r)$ gebildet. Dabei ist $\mathit{Stellen}(r)$ als die Menge aller Stellen definiert, die zu den Annotationen der Regel r erstellt wurden:

$$v'_r = \frac{v_r + \sum_{s \in \mathit{Stellen}(r)} \overline{g_s}}{1 + |\mathit{Stellen}(r)|}$$

Bei diesem Vorgehen werden bei der Berechnung des neuen Vertrauenswertes sämtliche zu einer Regel erstellten Annotationen und die zugehörigen Stellen berücksichtigt. Das hat den Effekt, dass bei sehr vielen Annotationen und nur wenigen geänderten Genauigkeitswerten sich der Vertrauenswert nur langsam an den Idealwert annähert. Um die Annäherung zu beschleunigen, kann $\mathit{Stellen}(r)$ als die Menge aller zu den Annotationen der Regel r erstellten Stellen s mit $\overline{g_s} \neq g_s$ definiert werden. So würden nur die Stellen der Annotationen berücksichtigt, dessen errechneter Genauigkeitswert g_s von dem durchschnittlichen virtuellen Genauigkeitswert $\overline{g_s}$ abweicht.

3.1.3 Beurteilung des Verfahrens

Bei dem vorgestellten Verfahren wird die Verlässlichkeit der Suchergebnisse nach der Mustererkennung bewertet. Dazu gibt der Reverse Engineer bei der Spezifikation von Software-Mustern für jede Musterregel eine Schätzung darüber ab, wie wahrscheinlich es ist, dass ein Fund tatsächlich eine Instanz des spezifizierten Musters ist. Mit Hilfe dieser Schätzungen und unter Berücksichtigung der Vorgängerbeziehungen von Annotationen werden die als Musterinstanzen markierten Objektstrukturen im abstrakten Syntaxgraphen bewertet.

Das Schätzen der Vertrauenswerte bei der Spezifikation eines Musters ist schwierig, erfordert Erfahrung und Kenntnis über das zu untersuchende Softwaresystem. Diese bekommt der Reverse Engineer aber erst während oder nach der Analyse der Software. Außerdem sind Schätzungen ungenau und führen zu unpräzisen Bewertungsergebnissen.

Um die Genauigkeit der geschätzten Vertrauenswerte zu erhöhen wurde ein semi-automatisches, heuristisches Adaptionsverfahren entwickelt, welches im Abschnitt 3.1.2 vorgestellt wurde. Initiiert durch vom Reverse Engineer durchgeführte Korrekturen der für Annotationen berechneten Genauigkeitswerte werden die geschätzten Vertrauenswerte angepasst.

Die Adaption der Vertrauenswerte erfordert mehrere Iterationen und viele Korrektureingaben vom Reverse Engineer. Dadurch wächst der Aufwand für die Mustererkennung erheblich an. Auch nach vielen Iterationen können die Vertrauenswerte ungenau bleiben, insbesondere dann, wenn die Adaption der Vertrauenswerte nicht konvergiert. Hinzu kommt das Problem, dass auch die Korrekturen unpräzise oder gar fehlerhaft sein können. Außerdem wirken sich die Korrekturen der Genauigkeitswerte einzelner Musterinstanzen durch die Adaption der Vertrauenswerte auf die Bewertung sämtlicher Instanzen zu einer Musterregel aus, was die zuvor korrekten Bewertungen verfälschen kann.

Bei dem vorgestellten Verfahren fällt auf, dass für die Bewertung von Musterinstanzen hauptsächlich Informationen über Musterregeln genutzt werden. Dazu zählen die Vertrauenswerte und die Abhängigkeiten der Regeln untereinander. Informationen über die als Musterinstanz markierten Objektstrukturen im abstrakten Syntaxgraphen bleiben bis auf die Vorgängerbeziehungen der Annotationen unberücksichtigt. Zum Beispiel wirkt sich die Anzahl der zu einem optionalen oder einem Mengenknoten gefundenen Objekte nicht auf die Bewertung einer Musterinstanz aus. Außerdem wird durch die Schätzung eines Vertrauenswertes eine Aussage über alle Instanzen zu einer Musterregel gleichermaßen gemacht. Das führt dazu, dass die Bewertungsergebnisse der Musterfunde zu einer Musterregel kaum variieren, wodurch die Übersicht bei den Suchergebnissen verloren geht. Eine Sortierung der Musterfunde nach ihrem Genauigkeitswert bringt nur wenig Erkenntnis, da sie auf diese Weise wenige Gruppen von gleich bewerteten Funden bilden.

Für Instanzen zum Muster „Iterator“ zum Beispiel gibt es nur zwei mögliche Bewertungsergebnisse, nämlich eine Bewertung mit 70% oder 80%. Die Abbildungen 3.4 (S. 30) und 3.7 zeigen, wie die beiden Bewertungsergebnisse zu Stande kommen können.

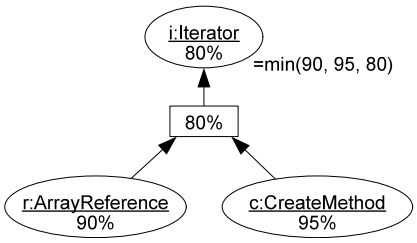


Abbildung 3.7: Alternative Bewertung einer Instanz zum Muster „Iterator“

Zur Erkennung des Musters „Iterator“, wie es in der Regel in Abbildung 3.1 (S. 29) spezifiziert wurde, wird unter anderem eine Annotation vom Typ MultiReference benötigt. Da die Musterregel „ArrayReference“ von der Regel „MultiReference“ erbt, kann eine ArrayReference-Annotation an Stelle einer MultiReference-Annotation verwendet werden. Somit beschreibt die Musterregel „Iterator“ implizit zwei alternative Objektstrukturen, eine mit einer MultiReference- und eine mit einer ArrayReference-Annotation. Da die Bewertungsergebnisse von Musterinstan-

zen nur von den Vertrauenswerten und den Annotationsabhängigkeiten beeinflusst werden und es keine Musterregeln gibt, die von der Regel „CreateMethod“ erben, stellen die Abbildungen 3.4 (S. 30) und 3.7 die einzigen möglichen Bewertungen von Instanzen zum „Iterator“-Muster dar.

Erbt eine Musterregel von einer anderen, so stellt diese immer eine alternative Objektstruktur dar. Aus dem Beispiel wird deutlich, dass die Bewertung von Musterinstanzen nur dann variieren kann, wenn solche Alternativen bei Hilfsmusterregeln existieren. Die Anzahl der möglichen Bewertungsergebnisse für die Funde zu einer Musterregel ist dadurch stark eingeschränkt.

Das bisher in Fujaba verwendete und in den Abschnitten 3.1.1 und 3.1.2 vorgestellte Bewertungsverfahren basiert auf vom Reverse Engineer abgegebenen Schätzungen und ist dadurch unpräzise. Die semi-automatische Adaption der Vertrauenswerte erhöht die Präzision zwar, erfordert aber viele manuell durchzuführende Korrekturen der Genauigkeitswerte und ist dadurch sehr aufwendig. Vom Prinzip her werden weniger die Musterinstanzen, sondern vielmehr die Musterregeln bewertet. Dadurch sind die Bewertungsergebnisse für die Instanzen zu einer Musterregel meist gleich, was den Nutzen der Bewertung in Frage stellt. Da die Eigenschaften der Musterfunde bei ihrer Bewertung nahezu unberücksichtigt bleiben, ist keine zufrieden stellende Aussage über die Qualität der Suchergebnisse möglich.

3.2 Berechnung des Vollständigkeitsgrades einer Musterinstanz

In Rahmen seiner Diplomarbeit [Wen05a] hat Sven Wenzel einen Mustererkennungsansatz entwickelt, bei dem – im Gegensatz zu dem bei Fujaba verfolgten Ansatz – auch unvollständige Musterinstanzen gesucht und bezüglich ihrer Vollständigkeit bewertet werden [Wen05b, Wen05c].

Das Ziel dabei ist, einen Softwareentwickler beim Forward Engineering zu unterstützen, indem ihm in der Design-Phase Hilfestellungen bei der Vervollständigung eines Software-Musters, zum Beispiel eines Design Patterns [GHJV95], gegeben werden. Dazu wird nicht der Quellcode nach Vorkommen von Musterinstanzen durchsucht, sondern das noch unvollständige UML-Klassendiagramm einer Software. Wird ein unvollständiges Muster gefunden, so wird der Entwickler darauf hingewiesen und es werden Vorschläge gemacht, wie er das Muster komplettieren kann.

Wie bei Fujaba werden auch bei diesem Ansatz Software-Muster durch erweiterte UML-Objektdiagramme beschrieben. Jedes dieser Diagramme stellt die an einem Muster beteiligten Komponenten (Klassen, Methoden, Attribute, Assoziationen und andere) in Form von Objekten dar. Jedes Objekt in dem Diagramm ist ein Platzhalter für ein Element des zu untersuchenden Klassendiagramms und wird in der Diplomarbeit *Rolle* genannt.

Die Typen der Objekte beziehungsweise Rollen legen die Art der geeigneten

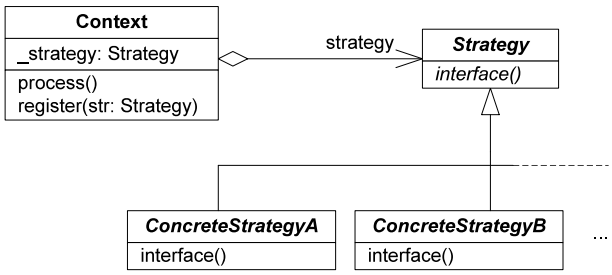


Abbildung 3.8: Struktur des Design Patterns „Strategy“ laut [Wen05a]

Klassendiagrammelemente fest, so gibt es zum Beispiel die Typen Class, Operation (repräsentiert eine Methode) und Parameter. Jede Rolle kann mehrere Unterrollen enthalten, so kann zum Beispiel eine Class-Rolle Operation-Rollen enthalten und diese wiederum Parameter-Rollen. Diese Enthaltensbeziehungen werden wie Aggregationen in UML-Klassendiagrammen dargestellt. Als oberste Rolle in der Hierarchie erhält jede Musterspezifikation eine so genannte *virtuelle Rolle*, die eine Musterinstanz repräsentiert.

Die Abbildung 3.9 stellt als Beispiel die Spezifikation des Design Patterns „Strategy“ [GHJV95] laut [Wen05a] dar. Zum Vergleich ist die beschriebene Struktur auch als UML-Klassendiagramm abgebildet (siehe Abb. 3.8). Diese enthält einige Erweiterungen gegenüber der Abbildung in [GHJV95].

Um weitere Eigenschaften der Elemente eines Musters zu spezifizieren, werden Constraints der *Object Constraint Language* (OCL) verwendet. Diese können angeben, dass eine Klasse abstrakt sein muss, oder ihre Sichtbarkeit festlegen. Außerdem können Beziehungen zwischen Rollen beschrieben werden, zum Beispiel Vererbung, Überschreiben von Methoden oder der Typ eines Parameters. Zur besseren Übersicht sind die Constraints in der Abbildung 3.9 nur informell in Form von Kommentaren dargestellt.

Ein spezieller Algorithmus versucht jeder Rolle ein geeignetes Element des Klassendiagramms zuzuordnen. Die Zuordnung eines Klassendiagrammelements zu einer Rolle wird *Binding* genannt. Bei der Entscheidung, ob ein Klassendiagrammelement eine bestimmte Rolle spielen kann, werden Constraints normalerweise ignoriert. Stattdessen werden sie für die Bewertung der einzelnen Bindings verwendet. Es kann allerdings auch ein so genannter *Necessary Flag* für ein Constraint gesetzt werden. Dieser verhindert, dass ein Binding erstellt wird, wenn das Constraint nicht erfüllt ist.

Wenn nichts anderes spezifiziert wurde, wird versucht, jeder Rolle genau ein Klassendiagrammelement zuzuordnen. Durch Angeben der Multiplizität einer Rolle wie bei Assoziationsenden in UML-Klassendiagrammen kann jedoch eine Mindest- und eine Maximalanzahl der zu einer Rolle zugeordneten Elemente spezifiziert werden.

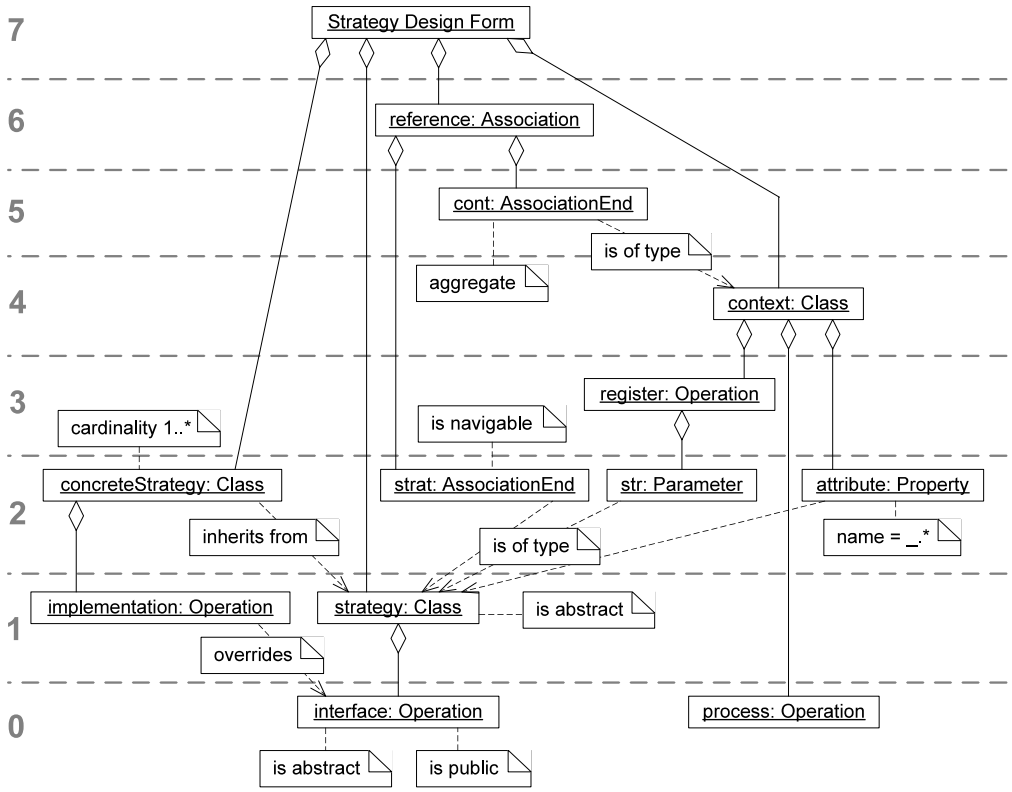


Abbildung 3.9: Spezifikation des Design Patterns „Strategy“

3.2.1 Bewertung von Musterinstanzen

Vor der Bewertung von Bindings und Musterinstanzen werden die Rollen bezüglich ihrer Abhängigkeiten sortiert. Dabei werden sowohl die Unterrollenbeziehungen als auch die durch OCL-Constraints entstehenden Abhängigkeiten berücksichtigt. In der Abbildung 3.9 sind die Rollen bereits entsprechend sortiert dargestellt. Anschließend werden die Bindings bottom-up bewertet, das heißt mit den Rollen beginnend, die von keinen anderen abhängen.

Die Qualität eines Bindings wird durch einen Prozentwert ausgedrückt, der angibt, wie gut ein Klassendiagrammelement eine bestimmte Rolle spielen kann. Dazu wird überprüft, wie viele der für eine Rolle spezifizierten Constraints von dem Element im Klassendiagramm erfüllt werden.

Ist eine Rolle wie process in der Abbildung 3.9 unabhängig von anderen und wurden für sie keine Constraints spezifiziert, so wird ein zugehöriges Binding b immer mit 100% bewertet. Sind dagegen Constraints spezifiziert, so wird ein Binding mit dem Durchschnitt der Erfülltheitsgrade der Constraints bewertet. Die folgende, aus [Wen05a] entnommene Formel beschreibt die Bewertung eines Bindings zu einer unabhängigen Rolle:

$$q(b) = \begin{cases} \frac{1}{|C|} \cdot \sum_{c \in C} eval(c), & \text{falls } |C| > 0 \\ 100\%, & \text{sonst} \end{cases} \tag{3.1}$$

Hierbei bezeichnen $q(b)$ die errechnete Qualität des Bindings b, C die Menge der für die Rolle des Bindings b spezifizierten Constraints und $eval(c)$ den Erfülltheitsgrad des Constraints c für das Binding b. Es gilt $eval(c) = 0\%$, falls das Constraint c nicht erfüllt ist und $eval(c) = 100\%$, falls das Constraint voll erfüllt ist. Bei Constraints, die auch zum Teil erfüllt werden können, kann der Erfülltheitsgrad auch einen Wert zwischen 0% und 100% einnehmen.

In dem Fall, dass eine Rolle Unterrollen besitzt, werden neben der Erfülltheitsgrade der Constraints auch die Bewertungen der Bindings für Unterrollen analog zu Constraints berücksichtigt. Die Bewertung eines Bindings b zu einer Rolle mit Unterrollen wird laut [Wen05a] durch folgende Formel ausgedrückt:

$$q(b) = \frac{1}{|C| + |U|} \cdot \left(\left(\sum_{c \in C} eval(c) \right) + \left(\sum_{u \in U} q(u) \right) \right) \tag{3.2}$$

Hier bezeichnet U die Menge der Unterrollen der zu b gehörenden Rolle und $q(u)$ die Qualität des Bindings für die Unterrolle u. Ist für eine Unterrolle u kein Binding vorhanden, so gilt $q(u) = 0\%$. Die übrigen Bezeichner entsprechen denen in der Formel 3.1.

Die Bewertung einer Musterinstanz erfolgt analog zur Formel 3.2 abhängig von der Qualität der Bindings zu den Unterrollen der virtuellen, das Muster repräsentierenden Rolle.

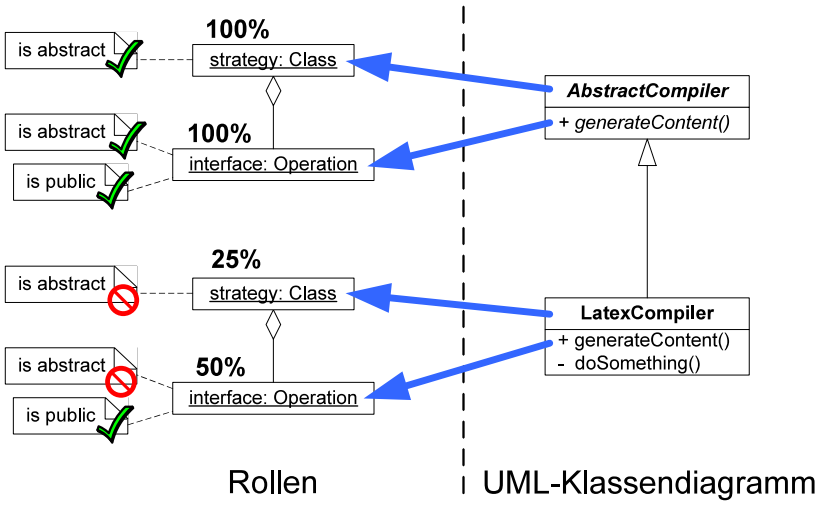


Abbildung 3.10: Bewertung von Bindings

Ein aus der Diplomarbeit von Sven Wenzel entnommenes Beispiel für die Bewertung von Bindings ist in der Abbildung 3.10 dargestellt. Hier ist ein Ausschnitt aus einem zu untersuchenden UML-Klassendiagramm sowie die Bindings der Rolle strategy und ihrer Unterrolle interface dargestellt (siehe Spezifikation des Design Patterns „Strategy“ in Abbildung 3.9). Die Bindings werden durch die dick dargestellten Pfeile repräsentiert. Die Rollen sind je einmal für jedes mögliche Binding

mit einer Bewertung größer 0% abgebildet und sind mit den jeweiligen Bewertungen markiert. Ein Haken symbolisiert ein erfülltes und ein Verbotsschild ein nicht erfülltes Constraint.

Da die generateContent-Methode der LatexCompiler-Klasse in dem Beispiel nicht abstrakt ist, ist eines der beiden Constraints der Rolle interface nicht erfüllt, das Binding wird also mit 50% bewertet:

$$\begin{aligned} q(b) &= \frac{1}{|C|} \cdot \sum_{c \in C} eval(c) \\ &= \frac{1}{2} \cdot (0\% + 100\%) \\ &= 50\% \end{aligned}$$

Das Constraint der Rolle strategy ist ebenfalls nicht erfüllt, weil die Klasse LatexCompiler nicht abstrakt ist. Aus diesem Grund wird das zugehörige Binding wie folgt bewertet:

$$\begin{aligned} q(b) &= \frac{1}{|C| + |U|} \cdot \left(\left(\sum_{c \in C} eval(c) \right) + \left(\sum_{u \in U} q(u) \right) \right) \\ &= \frac{1}{1+1} \cdot (0\% + 50\%) \\ &= 25\% \end{aligned}$$

Bei der Klasse AbstractCompiler werden alle Constraints der Rollen strategy und interface erfüllt, entsprechend werden die beiden Bindings mit jeweils 100% bewertet.

Die Rollen und Constraints können bei dem beschriebenen Ansatz auch mit Gewichten versehen werden, um ihre Wichtigkeit bei der Mustererkennung und Musterinstanzbewertung zu beschreiben. Sind gewichtete Rollen oder Constraints in einer Musterspezifikation enthalten, so ändert sich die Art der Berechnung. Die Formeln 3.1 und 3.2 werden in diesem Fall durch folgende ersetzt:

$$q(b) = \begin{cases} \frac{1}{f(C)} \cdot \sum_{c \in C} w(c) \cdot eval(c), & \text{falls } |C| > 0 \\ 100\%, & \text{sonst} \end{cases} \tag{3.3}$$

$$q(b) = \frac{1}{f(C) + f(U)} \cdot \left(\left(\sum_{c \in C} w(c) \cdot eval(c) \right) + \left(\sum_{u \in U} w(u) \cdot q(u) \right) \right) \tag{3.4}$$

Hierbei bezeichnet $w(x)$ das Gewicht einer Rolle oder eines Constraints x, während f definiert ist durch $f(X) = \sum_{x \in X} w(x)$.

3.2.2 Beurteilung des Verfahrens

Das vorgestellte Bewertungsverfahren hat entscheidende Vorteile gegenüber dem bisher verwendeten Verfahren bei Fujaba.

Im Gegensatz zu dem im Abschnitt 3.1 vorgestellten Ansatz wird bei dem Bewertungsverfahren von Sven Wenzel ein Fund bezüglich seiner Vollständigkeit bewertet. Anstatt der Eigenschaften einer Musterspezifikation werden die Eigenschaften einer Musterinstanz bewertet, nämlich erfüllte oder nicht erfüllte Constraints. Diese Art von Bewertung hat eine wesentlich höhere Aussagekraft. Je mehr Constraints ein Binding erfüllt, desto höher fällt die Bewertung des Bindings und damit auch die der zugehörigen Musterinstanz aus.

Die Bewertung wird völlig automatisch durchgeführt. Es sind keinerlei Benutzereingaben für die Bewertung notwendig. Insbesondere müssen die Bewertungsergebnisse nicht korrigiert werden, um präzisere Bewertungen zu erhalten.

Dadurch, dass kein Constraint erfüllt werden muss, außer derer mit Necessary Flag, entsteht eine Vielzahl an möglichen Kombinationen aus erfüllten und nicht erfüllten Constraints. Aus diesem Grund können sich die Bewertungsergebnisse stärker unterscheiden und sind präziser als bei dem bisher verwendeten Verfahren bei Fujaba.

Neben den Vorteilen des Ansatzes gibt es auch einen entscheidenden Kritikpunkt. Bei der Bewertung eines Bindings zu einer Rolle wird der Durchschnitt der Unterrollenbewertungen und der Erfülltheitsgrade der Constraints berechnet. Diese Art der Berechnung spiegelt nicht den Anteil der erfüllten Constraints bei einer Musterinstanz wieder.

Bei dem Beispiel aus Abbildung 3.10 sind insgesamt drei Constraints für die Rolle strategy und ihre Unterrolle interface angegeben. Fasst man diese beiden Rollen als ein eigenständiges Muster auf, so sind bei einem der beiden dargestellten Funde zwei von drei Mustereigenschaften nicht vorhanden. Der Anteil der von der Musterinstanz erfüllten Constraints ist in diesem Fall $\frac{1}{3} \approx 33.3\%$ und nicht 25%. Die Konsequenz aus dieser Bewertungsmethode ist, dass beim Vergleich mehrerer Musterinstanzen nicht von einer höheren Bewertung auf eine vollständigere Musterinstanz geschlossen werden kann. Das Verfahren kann also nicht zum Vergleich der Qualität mehrerer Musterinstanzen genutzt werden. Das folgende Beispiel soll dieses verdeutlichen.

In der Abbildung 3.11 sind die Bindings von zwei Funden zum gleichen Muster dargestellt. Das Muster wird durch eine virtuelle und zwei Class-Unterrollen beschrieben. Für eine der beiden Unterrollen ist ein Constraint spezifiziert, für die andere Rolle eine deutlich größere Anzahl an Constraints. Bei beiden Funden sind alle Constraints zu einer der beiden Unterrollen erfüllt, die Constraints der jeweils anderen Unterrolle nicht.

Für beide Musterinstanzen erhält man $\frac{1}{2} \cdot (0\% + 100\%) = 50\%$ als Bewertungsergebnis, obwohl der Anteil der erfüllten Constraints und damit auch der Grad der Vollständigkeit bei den beiden Musterinstanzen sehr unterschiedlich sein kann (zum Beispiel 1% im Vergleich zu 99% der spezifizierten Constraints erfüllt). Die

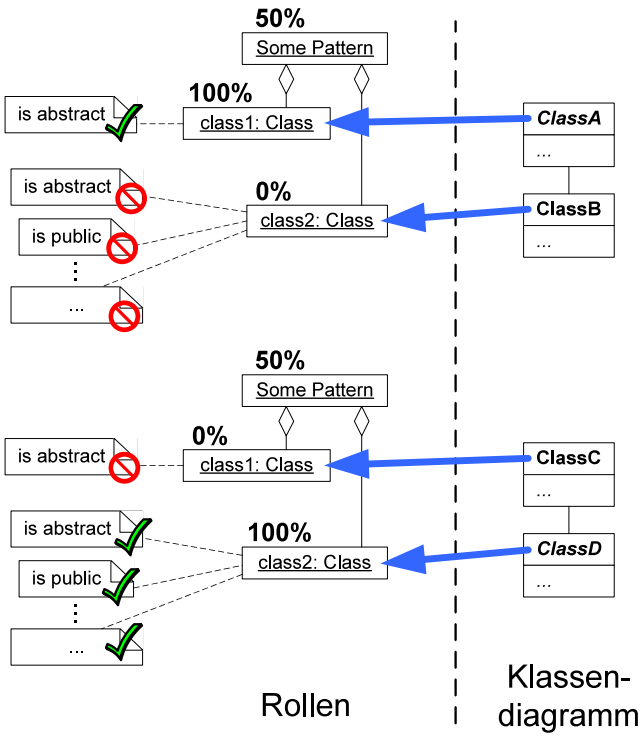


Abbildung 3.11: Beispiel für eine ungeeignete Bewertung

Bewertung der Musterinstanzen entspricht nicht dem Grad ihrer Vollständigkeit beziehungsweise ihrer Qualität und kann dadurch nicht zum Sortieren der Suchergebnisse nach Relevanz genutzt werden.

3.3 Zusammenfassung

Wie bereits zu Beginn dieses Kapitels beschrieben, gibt es nur wenige Ansätze zur Erkennung von Software-Mustern. Zwei der drei genannten Ansätze wurden in den vorhergehenden Abschnitten vorgestellt und bezüglich ihres Nutzens bei der Mustererkennung untersucht. Beide Bewertungsverfahren machen keine zufrieden stellende Aussage über die Qualität einer Musterinstanz.

Bei dem in Fujaba verwendeten Ansatz (siehe Abschnitt 3.1) schätzt ein Reverse Engineer die Genauigkeit seiner Musterspezifikationen. Musterinstanzen werden ausschließlich auf diesen Schätzungen und den Abhängigkeiten der Musterregeln basierend bewertet. Die Eigenschaften der Musterinstanzen bleiben bei ihrer Bewertung größtenteils unberücksichtigt. Dadurch stimmen die Bewertungen (nahezu) aller Musterinstanzen zur selben Regel überein. Die Qualität eines Musterfundes wird sehr unpräzise ausgedrückt und die Bewertung hat wenig Aussagekraft.

Die Idee bei dem Ansatz von Sven Wenzel (siehe Abschnitt 3.2) ist es, den Grad der Vollständigkeit einer Musterinstanz als Bewertung zu verwenden. Die Art der Berechnung – Bilden des Durchschnitts aus den Bewertungen der Unterrollen

und den Erfülltheitsgraden der Constraints – verfehlt allerdings dieses Ziel. Der Anteil der erfüllten Constraints und damit auch der Grad der Vollständigkeit einer Musterinstanz entspricht im Allgemeinen nicht der Musterinstanzbewertung, wodurch die Bewertungsergebnisse wenig Erkenntnis bringen.

4 Neuentwicklung eines Bewertungsverfahrens

Die im vorhergehenden Kapitel beschriebenen Bewertungsverfahren für Musterinstanzen weisen erhebliche Mängel auf. Im Rahmen dieser Arbeit wurde ein neuer Ansatz entwickelt, der die genannten Probleme weitestgehend löst. Dieser basiert auf dem im Kapitel 2 vorgestellten Mustererkennungsverfahren [NSW+02, NWW03] und greift die Idee auf, den Vollständigkeitsgrad eines Musterfundes als seine Bewertung zu verwenden.

Bevor der Ansatz im Detail vorgestellt wird, werden Anforderungen an ein Bewertungsverfahren formuliert und die notwendigen Eigenschaften einer beliebigen, für das Mustererkennungsverfahren von Fujaba geeigneten Bewertungsfunktion erarbeitet. Der Beschreibung des entwickelten Bewertungsverfahrens folgt eine Zusammenfassung.

4.1 Allgemeine Anforderungen

Eines der Ziele von Werkzeugen zur automatischen Erkennung von Software-Mustern ist, den Zeitaufwand beim Reverse Engineering zu reduzieren. Um dieses Ziel zu erreichen, werden diverse Anforderungen an ein Mustererkennungsverfahren gestellt.

Der Prozess der Mustererkennung kann in drei Teilaufgaben unterteilt werden: die manuelle Spezifikation der Software-Muster, die Werkzeug-gestützte Suche nach Vorkommen von Musterinstanzen beziehungsweise Ausprägungen eines Musters und die manuelle Analyse der Suchergebnisse. An alle drei Aufgaben wird die Anforderung gestellt, den menschlichen Aufwand so weit wie möglich zu reduzieren. Trotz der vielen zu berücksichtigenden Implementierungsvarianten von Software-Mustern (siehe Abschnitt 1.1) soll der Aufwand für die Spezifikation der Muster gering gehalten werden. Damit die Identifikation der korrekten Musterfunde in den Suchergebnissen möglichst wenig Zeit beansprucht, ist bei der Mustersuche eine hohe Präzision erwünscht (wenige False Negatives und False Positives). Außerdem soll die Mustersuche auch bei sehr großen zu untersuchenden Softwaresystemen anwendbar bleiben und die Laufzeit minimiert werden.

Unter Berücksichtigung der oben genannten Anforderungen an ein Werkzeug-gestütztes Mustererkennungsverfahren werden im Folgenden die Anforderungen an ein automatisches Bewertungsverfahren für Musterfunde formuliert.

1. **Bewertungskriterium:** Die Bewertung eines Musterfundes soll eine numerische Aussage über die Qualität des Fundes machen. Idealerweise wird bewertet, wie gut der Fund auf die Beschreibung eines Musters passt. Als Beschreibung dient die Spezifikation des Musters.

2. **Präzision der Bewertung:** Die Qualität der Musterfunde ist durch eine möglichst präzise Bewertung auszudrücken, um die Identifikation der relevanten Funde zu vereinfachen. Die Bewertung soll eine Sortierung der Musterfunde nach ihrer Qualität ermöglichen, unabhängig davon, ob die Funde Ausprägungen verschiedener Muster sind.

3. **Präzision der Suche:** Durch das Bewertungsverfahren darf die Präzision der Mustererkennung nicht gefährdet werden, insbesondere darf die Anzahl der False Negatives nicht erhöht werden.

4. **Effizienz:** Die Bewertungsfunktion soll möglichst effizient berechenbar sein. Eine wesentliche Erhöhung der Laufzeit für die Mustererkennung aufgrund der Bewertung von Musterinstanzen ist zu vermeiden.

5. **Menschlicher Aufwand:** Eine Erhöhung des Aufwands für den Reverse Engineer aufgrund der Werkzeug-gestützten Bewertung von Musterfunden ist möglichst zu vermeiden.

4.2 Eigenschaften einer Bewertungsfunktion

Im Folgenden werden die Eigenschaften einer Bewertungsfunktion formuliert, die für die effiziente Bewertung von Musterinstanzen in Verbindung mit dem in Fujaba verwendeten und im Kapitel 2 vorgestellten Mustererkennungsverfahren notwendig sind.

Bei diesem Verfahren werden Muster durch Musterregeln beschrieben und die dazu gefundenen Musterinstanzen mit Hilfe von Annotationen im abstrakten Syntaxgraphen markiert. Die Annotationen enthalten Informationen über die Elemente einer Musterinstanz, die zur Bewertung der Musterinstanz verwendet werden können. Ein notwendiger Parameter einer möglichen Bewertungsfunktion ist also die Annotation einer Musterinstanz.

Zur Bewertung einer Musterinstanz sollen die Eigenschaften der annotierten Objektstruktur im abstrakten Syntaxgraphen und die Eigenschaften der Objektstruktur, die in der zur Annotation gehörenden Musterregel beschriebenen ist, verglichen werden. Um eine effiziente Berechnung zu erzielen (siehe Anforderung 4 im Abschnitt 4.1), sollen die Bewertungen der Vorgänger einer Annotation wiederverwendet werden, was durch eine rekursive Funktion erreicht wird.

Insgesamt ergibt sich die folgende Struktur einer Bewertungsfunktion v:

$$v : A \rightarrow W \quad \text{mit} \quad v(a) = F\left(E_1\left(a\right), E_2\left(\text{Regel}\left(a\right)\right), v\left(a_1\right), \ldots, v\left(a_k\right)\right)$$

Hierbei bezeichnen

- A die Menge der Annotationen
- W eine geordnete Menge, zum Beispiel die reellen Zahlen $\mathbb{R}$
- $a \in A$ die Annotation, welche die zu bewertende Musterinstanz markiert
- Regel(a) die zur Annotation a gehörende Musterregel (Spezifikation des gesuchten Musters)
- $a_1, \ldots, a_k \in A$ die direkten Vorgänger der Annotation a
- $E_1(a) \in \mathbf{E}_1$ eine Menge von Eigenschaften der Annotation a
- $E_2(\text{Regel}(a)) \in \mathbf{E}_2$ eine Menge von Eigenschaften der Musterregel Regel(a)
- F eine Funktion, die beschreibt, wie die Eigenschaften der Annotation a und der Musterregel Regel(a) mit den Bewertungen der Vorgängerannotationen verrechnet werden. Diese hat die Form $F : \mathbf{E}_1 \times \mathbf{E}_2 \times W^k \rightarrow W$.

4.3 Ansatz

Software-Muster wie Design Patterns oder Anti Patterns werden meist informell, oft in natürlicher Sprache und ungenau beschrieben (siehe Abschnitt 1.1). Die Spezifikation der Muster nach dem in Fujaba verwendeten Ansatz (siehe Kapitel 2) kann dagegen nur präzise durch Angeben einer Objektstruktur und zugehöriger Bedingungen gemacht werden. Eine Musterinstanz wird nur dann erkannt, wenn die angegebene Struktur vollständig gefunden und sämtliche spezifizierten Bedingungen erfüllt sind. Bereits geringe Abweichungen von der Spezifikation eines Musters führen dazu, dass eine Musterinstanz nicht erkannt wird.

Um die Anzahl der False Negatives gering zu halten, werden bei der Spezifikation eines Musters bisher ausschließlich „sichere“ Informationen verwendet, also diejenigen, die auf jede Musterinstanz zutreffen [NWW03].

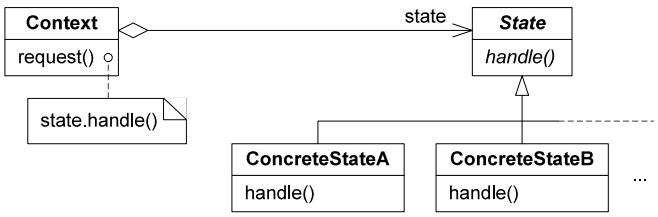


Abbildung 4.1: Struktur des Entwurfsmusters „State“

Als Beispiel kann hier das Entwurfsmuster „State“ [GHJV95, S. 305] genannt werden, dessen Struktur in der Abbildung 4.1 dargestellt wird. Dieses beschreibt eine Struktur, bei der ein Objekt, der Kontext, sein Verhalten aufgrund von Zustandsänderungen wechseln kann. Jeder der möglichen Zustände wird dabei durch

je eine Klasse realisiert, die das Verhalten des Objekts in dem jeweiligen Zustand implementiert. Die Schnittstelle der Zustandsklassen wird durch eine abstrakte Oberklasse definiert. Das Kontextobjekt hält eine Referenz auf ein Zustandsobjekt, an das alle zustandsabhängigen Aufgaben delegiert werden.

Laut Musterbeschreibung kann der Kontext als Argument an das Zustandsobjekt übergeben werden. So könnte bei Bedarf auf den Kontext zugegriffen werden, um zum Beispiel den Zustand zu wechseln. Auch wenn es andere Implementierungsvarianten gibt, würde das Vorhandensein eines solchen Arguments mit einer größeren Sicherheit auf eine Instanz des Musters „State“ hindeuten als ohne dieses Argument. Der Fund würde besser auf die Beschreibung des Musters passen. Auch der Begriff „state“ in dem Namen der abstrakten Zustandsklasse würde ein Suchergebnis untermauern, allerdings muss diese Bedingung nicht zwingend erfüllt sein. In der Praxis kommt es außerdem vor, dass bei der Anwendung eines Software-Musters kleinere Abweichungen zur Musterbeschreibung entstehen, zum Beispiel könnte bei dem „State“-Muster vergessen worden sein, die Oberklasse für Zustände abstrakt zu machen.

Um sicherzustellen, dass auch eine Musterinstanz ohne die genannten Eigenschaften erkannt wird, werden solche „unsicheren“ Informationen – wie die Information, dass ein Kontextobjekt an ein Zustandsobjekt übergeben werden kann – bei der Spezifikation eines Musters nach dem bisherigen Ansatz [NWW03] entweder weggelassen oder es werden mehrere die verschiedenen Alternativen beschreibende Musterregeln spezifiziert. Dadurch, dass ein Teil der Informationen bei der Mustererkennung ungenutzt bleibt, sinkt die Präzision bei der Erkennung, die Anzahl der False Positives steigt.

Die Idee bei dem in dieser Arbeit vorgestellten Bewertungsverfahren ist es, einen Teil der bisher ungenutzten, „unsicheren“ Informationen zur Bewertung der Musterfunde zu verwenden. Dazu werden die Musterregeln um diese Informationen in Form von optionalen Bedingungen erweitert. Optionale Bedingungen – wie die Forderung danach, dass ein Kontextobjekt an ein Zustandsobjekt übergeben wird – können, müssen aber nicht erfüllt werden. Das Erfüllen dieser Bedingungen ist jedoch erwünscht. Eine größere Anzahl an erfüllten Bedingungen bei einem Musterfund wird als vollständigeres und damit besseres Suchergebnis interpretiert. Je mehr optionale Bedingungen von einer Musterinstanz erfüllt werden, desto besser passt die Beschreibung beziehungsweise die Spezifikation des Musters auf die Instanz und desto höher wird der Fund bewertet.

Ein besonderes Augenmerk ist darauf zu richten, dass hier die Bedeutung des Begriffs „optional“ ein wenig von der ursprünglichen Bedeutung abweicht[1], was dem Reverse Engineer bei der Spezifikation der Muster bewusst sein muss.

Dadurch, dass bisherige Musterspezifikationen um optionale Bedingungen erweitert und bisher verwendete Elemente einer Musterregel erhalten werden, gehen

[1] „Optional“ im eigentlichen Sinne würde bedeuten, dass es egal ist, ob eine optionale Bedingung erfüllt ist oder nicht. Hier erhält der Begriff „optional“ die Bedeutung: Es ist nicht notwendig aber erwünscht, dass eine optionale Bedingung erfüllt ist.

keine Informationen verloren. Die Anzahl der False Negatives bei einer Mustererkennung wird also nicht erhöht, womit die Anforderung 3 (siehe Abschnitt 4.1) erfüllt wird.

Zur Bewertung einer Musterinstanz werden die in der zugehörigen Musterregel definierten Eigenschaften als Bedingungen angesehen und der Anteil der von der Musterinstanz erfüllten Bedingungen wird berechnet. Vereinfacht dargestellt kann die Bewertung einer Musterinstanz, die durch eine Annotation a repräsentiert wird, als der Quotient aus der Anzahl der von ihr erfüllten und den erfüllbaren Bedingungen beschrieben werden:

$$Bewertung(a) = \frac{|ErfüllteBedingungen(a)|}{|ErfüllbareBedingungen(a)|} \tag{4.1}$$

Der Wert macht eine Aussage darüber, wie gut eine Musterinstanz, repräsentiert durch eine Objektstruktur im abstrakten Syntaxgraphen, auf die zugehörige Musterspezifikation passt. Die erste der im Abschnitt 4.1 formulierten Anforderungen ist damit erfüllt.

Als Bedingungen, die in einer Musterregel spezifiziert sind, werden unter anderem Constraints, Attributbedingungen und Knoten (Finden eines ASG-Objekts zu dem Knoten) angesehen. Bisher können jedoch nur Knoten als optional gekennzeichnet werden. Auf diese Weise können nur wenige „unsichere“ Informationen spezifiziert werden, also Informationen, die nicht auf jede Musterinstanz zutreffen müssen. Die Bewertungen mehrerer Musterinstanzen zur gleichen Musterregel können sich deswegen nur wenig unterscheiden. Um die Anzahl der möglichen Bewertungsergebnisse und die Präzision bei der Bewertung zu erhöhen (siehe Anforderung 2 im Abschnitt 4.1), wird die Musterspezifikationssprache um weitere optionale Elemente wie Attributbedingungen und Constraints erweitert.

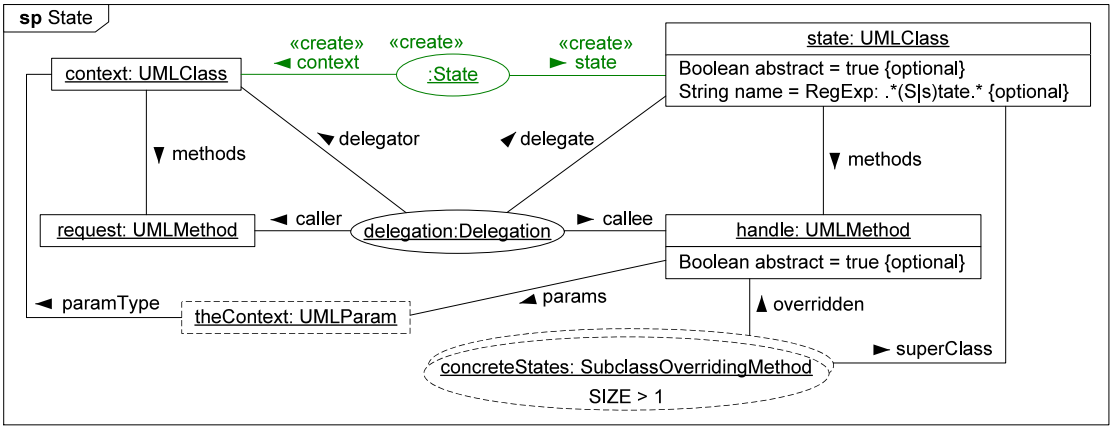


Abbildung 4.2: Spezifikation des Entwurfsmusters „State“

Mit Hilfe der optionalen Elemente und einiger weiterer Erweiterungen kann das „State“-Muster nun wie in der Abbildung 4.2 spezifiziert werden. Dass der Kontext als Argument an ein Zustandsobjekt übergeben werden kann, wird durch den

optionalen Knoten theContext definiert, der einen Parameter der handle-Methode repräsentiert. Die Bedingungen, dass die Zustandsoberklasse den Begriff „state“ im Namen enthält und diese Klasse ebenso wie die das Verhalten definierende handle-Methode abstrakt ist, werden als optional gekennzeichnet. Die Klassen, die die Zustände implementieren, werden durch einen Annotationsmengenknoten vom Typ SubclassOverridingMethod repräsentiert. Das Hilfsmuster „Subclass-OverridingMethod“ beschreibt Unterklassen, die eine bestimmte Methode überschreiben, in diesem Fall die handle-Methode (siehe Abb. B.3). Mit der Bedingung „SIZE > 1“ wird eine Mindestanzahl von zwei Zuständen vorausgesetzt.

Die bei dem bisherigen Bewertungsverfahren verwendeten Vertrauenswerte (siehe Abschnitt 3.1.1) werden nicht mehr für die Bewertung von Musterfunden benötigt und entfallen deswegen bei der Spezifikation in Abbilding 4.2 und allen folgenden. Sie können aber neben dem neuen Bewertungsverfahren für Musterinstanzen zur Bewertung der Qualität von Musterregeln verwendet werden.

Durch die optionalen Attributbedingungen und den optionalen Knoten in der Musterregel „State“ werden auch „unsichere“ Informationen spezifiziert, die zur Bewertung von Musterinstanzen verwendet werden. Eine höhere Anzahl an erfüllten optionalen Bedingungen führt zu einer höheren Bewertung einer Musterinstanz und drückt eine höhere Verlässlichkeit des Suchergebnisses aus.

Um bei der Spezifikation eines Musters angeben zu können, wie wichtig ein Knoten oder eine Bedingung in einer Musterregel für die Erkennung einer Musterinstanz ist, wurde eine Gewichtung der Musterregelelemente eingeführt. Wird zum Beispiel der Gewichtsfaktor 2 für eine Attributbedingung spezifiziert, so hat die Erfülltheit dieser Bedingung im Verhältnis zu einer Spezifikation ohne diesen Faktor den doppelten Einfluss auf die Bewertung einer Musterinstanz (hat das Gewicht von zwei Bedingungen).

Mit der bisher verwendeten Musterspezifikationssprache ist es nur möglich, absolute Aussagen über die Erfülltheit einer Bedingung zu machen: sie ist entweder erfüllt oder nicht. Als Ergänzung werden nun „unscharfe“, auf Software-Metriken basierende Bedingungen eingeführt. Diese können auch teilweise erfüllt werden. Der Grad, zu dem so eine Bedingung erfüllt ist, wird durch einen Wert aus dem Intervall $[0, 1]$ ausgedrückt. Ist sie nicht erfüllt, so ist der Erfülltheitsgrad 0. Ist sie voll erfüllt, so ist der Grad 1. Mit Hilfe der Erfülltheitsgrade von „unscharfen“ Bedingungen können genauere Aussagen zum Anteil der von einer Musterinstanz erfüllten Bedingungen gemacht werden, was die Präzision bei der Bewertung erhöht (vgl. Anforderung 2 im Abschnitt 4.1).

In der Gleichung 4.1 wird der Erfülltheitsgrad von Bedingungen, die teilweise erfüllt werden können (zum Beispiel der von „unscharfen“ Bedingungen), nicht berücksichtigt. Außerdem hat hier die Gewichtung von Bedingungen keinen Einfluss auf die Bewertung. Die Formel muss also verfeinert werden. Anstatt die erfüllten Bedingungen schlicht zu zählen werden bei der Bewertung der durch eine Annotation a repräsentierten Musterinstanz die gewichteten Erfülltheitsgrade der Bedingungen aufsummiert. Bedingungen, die nur erfüllt oder nicht erfüllt sein können, erhalten den Erfülltheitsgrad 1, wenn sie erfüllt sind und den Grad 0,

wenn sie nicht erfüllt sind. Das Gewicht einer Bedingung b wird durch den Wert $w_b \in \mathbb{R}^+$ beschrieben. Ist für die Bedingung b kein Gewicht spezifiziert worden, so gilt $w_b = 1$.

$$Bewertung(a) = \frac{\sum_{b \in ErfüllbareBedingungen(a)} Erfülltheitsgrad(b) \cdot w_b}{\sum_{b \in ErfüllbareBedingungen(a)} w_b} \tag{4.2}$$

Die Gleichung 4.2 stellt die Bewertung von Musterinstanten vereinfacht dar. Diese soll eine erste, grobe Vorstellung von dem erarbeiteten Bewertungsverfahren vermitteln. Ein ausführliche Beschreibung des Verfahrens folgt im Abschnitt 4.5.

Musterregeln können unter anderem Mengenknoten enthalten, die in der Formel oben noch nicht berücksichtigt wurden. Ein solcher Knoten repräsentiert eine Menge von Objekten oder Annotationen im abstrakten Syntaxgraphen. Neben der Information über die Erfülltheit von optionalen Bedingungen kann auch die Anzahl der zu einem Mengenknoten gefundenen Objekte einen Hinweis auf die Qualität eines Musterfundes liefern. So würden bei dem Muster „State“ (siehe Abb. 4.2, S. 47) zum Beispiel zehn Zustände mit einer höheren Sicherheit auf eine Instanz des Musters hindeuten als zwei oder weniger. Aus diesem Grund wird bei dem erarbeiteten Bewertungsverfahren eine höhere Anzahl an zu einem Mengenknoten gefunden Objekten als vollständigeres und damit besseres Suchergebnis interpretiert und führt zu einer höheren Bewertung der zugehörigen Musterinstanz. Mit Hilfe von Gewichten und „unscharfen“ Bedingungen kann das Verhalten bei der Bewertung von Mengen auch geändert werden. Näheres ist dem Abschnitt 4.5 zu entnehmen.

Bei dem in dieser Arbeit vorgestellten Ansatz können mehrere Implementierungsvarianten eines Musters wie in dem Abschnitt 1.1 beschrieben (siehe auch [NWW03]) in einer Spezifikation zusammengefasst und so die Anzahl der Spezifikationen reduziert werden. Als Verbesserung ist es aber möglich, einige Informationen über ein Muster mit Hilfe von optionalen Bedingungen als „unsicher“ zu kennzeichnen und für die Bewertung der Musterinstanzen zu nutzen, anstatt sie, wie in [NWW03] beschrieben, zu verwerfen.

Der Aufwand für die Spezifikation der Muster mit ihren zahlreichen Varianten (Anforderung 5 im Abschnitt 4.1) soll ebenso wie die Laufzeit der Mustererkennung (Anforderung 4) möglichst gering gehalten werden. Dazu wird angelehnt an das Vorgehen aus [NWW03] ein Mittelweg zwischen der vollständigen Spezifikation sämtlicher Implementierungsvarianten eines Musters in je einer Musterregel und der ausschließlichen Spezifikation der Gemeinsamkeiten aller Varianten in einer einzigen Musterregel gegangen. Diese Arbeit und die darin erarbeiteten Konzepte können jedoch nur das Werkzeug und einige Beispiele liefern. Die Musterspezifikation wird von einem Reverse Engineer, dem Anwender, durchgeführt. Dieser entscheidet, welchen Teil eines Musters er spezifiziert.

In den folgenden Abschnitten wird das im Rahmen dieser Diplomarbeit entwickelte Bewertungsverfahren und die zugehörigen Erweiterungen der Musterspezifikationssprache genauer erläutert und anhand von Beispielen veranschaulicht.

4.4 Erweiterung der Musterspezifikationssprache

Bevor das entwickelte Bewertungsverfahren im Detail vorgestellt werden kann, werden in diesem Abschnitt die notwendigen Erweiterungen der Musterspezifikationssprache beschrieben. Dazu gehören optionale Teilgraphen und Bedingungen, auf Metriken basierende und „unscharfe“ Bedingungen sowie die Gewichtung von Teilgraphen, Knoten und Bedingungen.

4.4.1 Optionale Elemente

Mit der im Abschnitt 2.2 vorgestellten Musterspezifikationssprache können bisher nur Objekt- und Annotationsknoten als optional gekennzeichnet werden. Um dem Reverse Engineer mehr Möglichkeiten zu bieten, Teile einer Musterregel als optional zu markieren, wird die Sprache um weitere optionale Elemente erweitert. Zusätzlich zu Knoten werden auch Constraints, Attribut- und die im folgenden Abschnitt vorgestellten *Metrikbedingungen* als optional kennzeichenbar. Außerdem können Teilgraphen einer Musterregel als optional spezifiziert werden.

Constraints, Attribut- und Metrikbedingungen werden durch das Schlüsselwort „optional“ gekennzeichnet, das in geschweiften Klammern hinter die Bedingung geschrieben wird. Optionale Teilgraphen werden – analog zu *Combined Fragments* in UML-Sequenzdiagrammen [Obj04] – durch ein Rechteck dargestellt, welches das Label „optional“ trägt und die zum Teilgraphen gehörenden Knoten umschließt. Dieses Rechteck wird im Folgenden *optionales Fragment* genannt.

Neben beliebigen Knoten kann ein optionales Fragment auch Constraints enthalten. Außerdem kann eine Kante einen Knoten innerhalb des Fragments mit einem Knoten außerhalb des Fragments verbinden. Überlappungen mehrerer optionaler Fragmente sind nicht erlaubt.

Bei Anwendung einer Regel wird die größtmögliche Anzahl an optionalen Bedingungen erfüllt, wobei in diesem Zusammenhang sämtliche optionalen Elemente (optionale Knoten, Teilgraphen, Constraints, Attributbedingungen,...) als optionale Bedingungen angesehen werden. Ein optionaler Teilgraph ist gefunden, wenn sämtliche in dem zugehörigen optionalen Fragment enthaltenen, nicht optionalen Elemente – ausgehende Kanten eingeschlossen – gefunden und die nicht optionalen Bedingungen erfüllt sind.

Die Abbildung 4.3 zeigt eine Spezifikation des Hilfsmusters für zu-1-Referenzen, das einige der neuen optionalen Elemente enthält. Hier wurden zusätzlich zu den Informationen, die in der bisher verwendeten Musterregel (vgl. Abb. 2.7, S. 19) verwendet wurden, auch „unsichere“ Informationen benutzt, also Informationen, die nicht auf jede Implementierungsvariante zutreffen.

Eine zu-1-Referenz wird gewöhnlicherweise mit Hilfe eines Attributs und zugehöriger Zugriffsmethoden realisiert. Da es aber auch Ausnahmen gibt, wird die Forderung nach dem Vorhandensein von Zugriffsmethoden mit Hilfe der beiden optionalen Fragmente, die je eine der Methoden beschreiben, abgeschwächt. Nach dem Information-Hiding-Prinzip sollte die Sichtbarkeit des Attributs möglichst

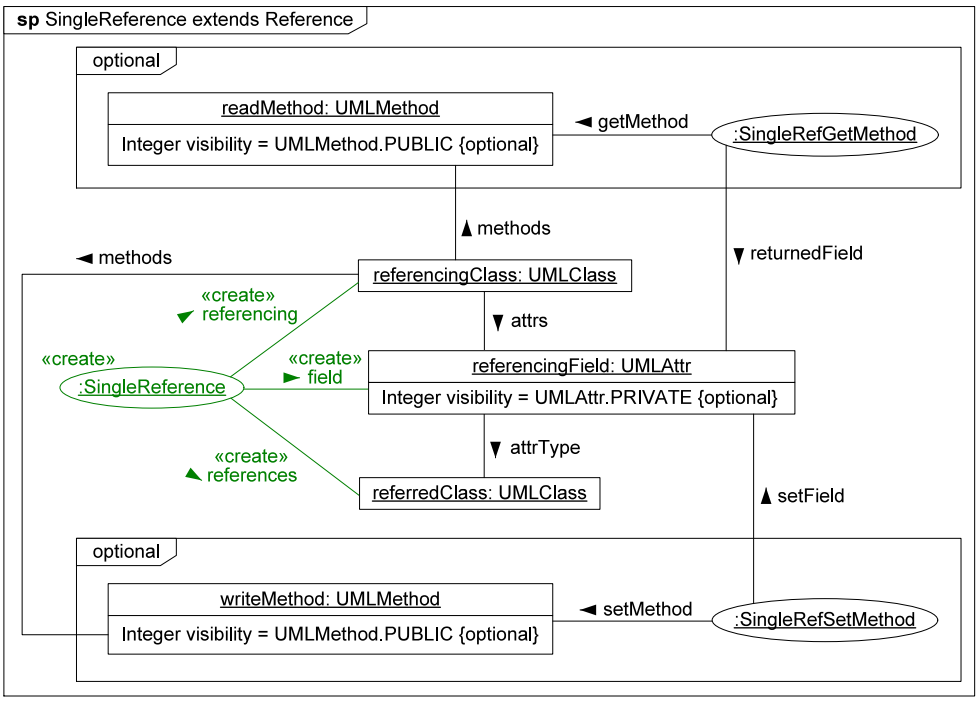


Abbildung 4.3: Spezifikation des Musters „SingleReference“ mit optionalen Elementen

weit eingeschränkt sein (private) und der Zugriff darauf über öffentliche (public) Methoden erfolgen. Auch diese Eigenschaften treffen nicht zwingend auf jede Implementierungsvariante zu, die zugehörigen Bedingungen sind deshalb optional.

Die Informationen über die Erfülltheit der optionalen Bedingungen werden für die Bewertung der Qualität einer Musterinstanz verwendet. Je mehr optionale Bedingungen erfüllt sind, desto mehr stimmt die Instanz eines Musters mit der zugehörigen Spezifikation überein und desto höher wird sie bewertet.

4.4.2 Metriken

Software-Metriken bieten eine Möglichkeit, ein Softwaresystem zu analysieren und quantitative Aussagen darüber zu machen. Zum Beispiel können sie Auskunft über die Anzahl der Code-Zeilen, der Methoden und der Attribute in einer Klasse oder die Tiefe einer Vererbungshierarchie geben. Metriken können unter anderem zur Identifikation von Implementierungs- oder Entwurfsmängeln eingesetzt werden und sind dadurch hilfreich bei der Beschreibung von Anti Patterns [BMMM98] oder Bad Smells [Fow99].

Um dem Reverse Engineer ein weiteres Mittel zur Beschreibung von Software-Mustern zu geben, wird die Musterspezifikationssprache um Metriken erweitert. Jedem Knoten einer Musterregel, der ein Objekt des abstrakten Syntaxgraphen repräsentiert, können Bedingungen hinzugefügt werden, die je eine Einschränkung bezüglich einer oder mehrerer Metriken formulieren. Zum Beispiel kann in einem

Tabelle 4.1: Einige der verwendbaren Metriken

Akronym	Kurzbeschreibung	anwendbar auf
CC	zyklomatische Komplexität nach McCabe	UMLMethod
COND	Anzahl der else if- und catch-Anweisungen	UMLMethod
LOC	Anzahl Code-Zeilen	UMLMethod
NI	Anzahl der Methodenaufrufe	UMLMethod
NOP	Anzahl der Parameter	UMLMethod
NOS	Anzahl der Anweisungen	UMLMethod
AvgCC	durchschnittliche zyklomatische Komplexität	UMLClass
DIT	Tiefe einer Klasse in ihrer Vererbungshierarchie	UMLClass
NAM	Anzahl abstrakter Methoden	UMLClass
NMO	Anzahl der überschriebenen Methoden	UMLClass
NOA	Anzahl der Attribute	UMLClass
NOC	Anzahl direkter Unterklassen	UMLClass
NOM	Anzahl der Methoden	UMLClass
WLOC	Summe der Code-Zeilen aller Methoden	UMLClass
WNOC	Anzahl direkter und indirekter Unterklassen	UMLClass
NOCL	Anzahl der Klassen	UMLProject
PLOC	Summe über WLOC aller Klassen	UMLProject
SIZE	Kardinalität einer Menge	Objektmenge

Knoten für Objekte vom Typ UMLClass die Bedingung „NAM ≥ 10“ angegeben werden, um auszudrücken, dass die Anzahl der abstrakten Methoden in der Klasse mindestens Zehn betragen muss (NAM = Number of Abstract Methods).

Die Angabe von Metrikbedingungen erfolgt analog zu Attributbedingungen (vgl. Abschnitt 2.2.3 und Tabelle 2.1 auf Seite 14). Anstatt des Attributnamens wird ein Akronym für die Metrik verwendet, zum Beispiel „NAM“ oder „LOC“ (Lines of Code). Erlaubte Operatoren sind wie bei Attributbedingungen „$=$“, „$\neq$“, „$>$“, „$\geq$“, „$<$“ und „$\leq$“. Zusätzlich kann eine Bedingung mit Hilfe logischer Operatoren („&“ für *und*, „|“ für *oder*, „!“ für *nicht*) aus mehreren zusammengesetzt werden, zum Beispiel „$(\text{LOC} \leq 20) \mid (\text{LOC} \geq 500)$“.

In der Tabelle 4.1 ist ein Teil der verwendbaren Metriken aufgelistet. Es werden das Akronym einer Metrik, eine Kurzbeschreibung und der Typ der Objekte angegeben, auf die eine Metrik angewendet werden kann. Weitere verwendbare Metriken sind in [MN05, Rot05] beschrieben.

Ein Beispiel für die Verwendung von Metriken stellt die Musterregel aus Abbildung 4.4 dar. Diese spezifiziert den Bad Smell „Large Class“, der in [Fow99] als eine besonders große und dadurch unübersichtliche, schwer wartbare Klasse beschrieben wird.

Mit Hilfe des UMLClass-Knotens und der drei Metrikbedingungen wird angegeben, dass durch die Musterregel eine Klasse mit „vielen“ Attributen, Methoden

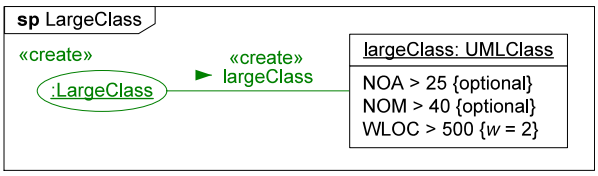


Abbildung 4.4: Musterregel zum Bad Smell „Large Class“ mit Metrikbedingungen

und Code-Zeilen gesucht wird. Die verwendeten Metriken sind „NOA“ (Number of Attributes), „NOM“ (Number of Methods) und „WLOC“ (Lines of Code). Der Begriff „viel“ wird hier durch die Schranken „NOA $>$ 25“, „NOM $>$ 40“ und „WLOC $>$ 500“ beschrieben. Für die letzte der drei Bedingungen wird durch den Ausdruck $w = 2$ ein höheres Gewicht spezifiziert. Die ersten beiden Bedingungen sind optional, wodurch zum Beispiel auch Klassen mit vielen Methoden aber nur wenigen Attributen erkannt werden.

Mit den bisherigen Mitteln der Musterspezifikationssprache konnten Bedingungen bezüglich der Anzahl der zu einem Mengenknoten gefundenen Objekte nur mit Hilfe von Java-Ausdrücken innerhalb von so genannten Constraints angegeben werden. Um die Angabe solcher Bedingungen zu vereinfachen, wird eine spezielle Metrik für Mengenknoten mit dem Akronym „SIZE“ eingeführt. Diese gibt die Anzahl der zu einem Mengenknoten gefundenen Objekte einer Musterinstanz an.

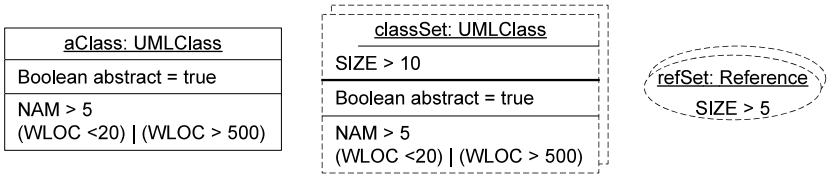


Abbildung 4.5: Beispiele für Metrikbedingungen

Um bei der Darstellung eines Objektknotens die Metrikbedingungen von den Attributbedingungen zu trennen, wird der für Bedingungen reservierte Bereich des Knotens durch eine waagerechte Linie unterteilt (siehe Abb. 4.5). In dem oberen Teil werden Attribut- und in dem unteren Metrikbedingungen angegeben. Bei Mengenknoten gibt es einen zusätzlichen Teil für Bedingungen mit der „SIZE“-Metrik, weil sich diese im Gegensatz zu den anderen Metriken und Bedingungen auf die Objektmenge und nicht auf ein einzelnes Objekt bezieht.

Ferner können Metriken auch in Constraints verwendet werden. Dazu wird bei Angabe der Metrik der Name des Knotens, auf das sich die Metrik bezieht, durch einen Punkt getrennt vor das Akronym der Metrik gestellt. Enthält eine Musterregel einen Mengenknoten mit dem Namen „set“ und einen Objektknoten vom Typ UMLClass mit dem Namen „c“, so kann zum Beispiel ein Constraint der Form „$\{\ (\text{set.SIZE} \geq 5)\ \&\ (\text{c.WLOC} > 500)\ \}$“ erstellt werden.

4.4.3 Fuzzy-Bedingungen

Die informelle Beschreibung von Software-Mustern enthält oft quantitative Angaben in Form von vagen Ausdrücken der natürlichen Sprache wie „viel“ oder „wenig“, „klein“ oder „groß“. Bisher können solche Information bei der Musterspezifikation nur durch Angabe von Schranken verwendet werden, welche keine treffende Beschreibung der vagen Ausdrücke sind.

Zum Beispiel wird bei der Musterregel „LargeClass“ aus Abbildung 4.4 eine Klasse mit *vielen* Attributen durch die Metrikbedingung „NOA $>$ 25“ beschrieben. Der Wert 25 ist ein Erfahrungswert. Die durch die Bedingung angegebene Schranke umschreibt den Begriff „viel“ im Zusammenhang mit dem Bad Smell „Large Class“ nur bedingt.

Ein anderes Problem ist, dass mit den bisherigen Mitteln der Musterspezifikationssprache nur Bedingungen formuliert werden können, die entweder erfüllt oder nicht erfüllt sind. Für die Bewertung von Musterinstanzen sind Bedingungen, die zum Teil erfüllt werden können, besser geeignet und deshalb erwünscht. Anstatt der Aussage, ob eine Bedingung erfüllt ist, könnte ihr Erfülltheitsgrad als Indiz für die Qualität einer Musterinstanz verwendet werden. Dadurch wäre die Bewertung präziser (siehe Anforderung 2 im Abschnitt 4.1).

Zur Lösung der beiden oben genannten Probleme soll eine Erweiterung der Musterspezifikationssprache beitragen, die es ermöglicht, quantitative Bedingungen mit Hilfe so genannter *Fuzzy-Mengen* (engl. *fuzzy sets*, siehe auch [TU97, Kapitel 2] und [Ser94, Kapitel 3]) „unscharf“ zu formulieren. Solche Bedingungen werden im Folgenden *Fuzzy-Bedingungen* genannt und bilden eine Alternative zur Spezifikation von Schranken.

Im Gegensatz zu „scharfen“ Mengen der traditionellen Mengenlehre, bei der ein Element entweder in der Menge ist oder nicht, wird durch eine Fuzzy-Menge („unscharfe“ Menge) M für eine Menge X der Grad der Zugehörigkeit von Elementen $x \in X$ zur Menge M beschrieben. Dieser wird durch eine *Zugehörigkeitsfunktion* der Form $\mu_M : X \rightarrow [0, 1]$, $x \mapsto \mu_M(x)$ angegeben.

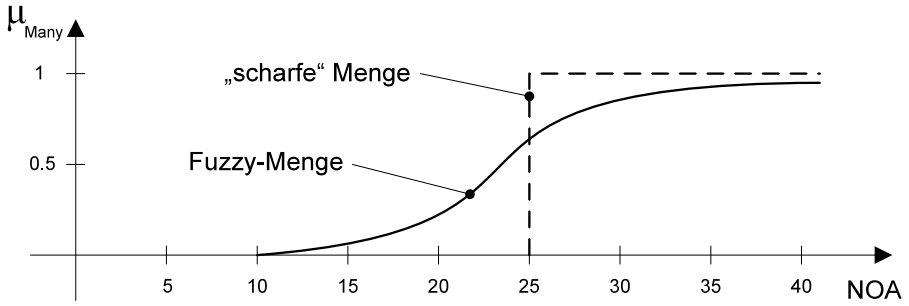


Abbildung 4.6: Beispiel für die Zugehörigkeitsfunktion einer Fuzzy-Menge

Bei dem Bad Smell „Large Class“ zum Beispiel kann eine Menge *Many* der Klassen mit vielen Attributen definiert werden. Die Zugehörigkeitsfunktion μ_{Many} würde in diesem Fall jeder möglichen Anzahl an Attributen einen Wert aus dem

Intervall $[0, 1]$ zuordnen. Der Definitionsbereich von μ_{Many} wäre die Menge der natürlichen Zahlen mit Null $\mathbb{N}_0$. Der Graph der Zugehörigkeitsfunktion kann wie in der Abbildung 4.6 aussehen. Auf der X-Achse ist hier die Anzahl der Attribute, beschrieben durch die Metrik „NOA“, aufgetragen. Zum Vergleich ist auch die „scharfe“ Menge der Klassen mit mehr als 25 Attributen abgebildet, was der Spezifikation in Abbildung 4.4 entspricht.

Mit Hilfe der Zugehörigkeitsfunktion μ_{Many} kann zwischen den Erfülltheitsgraden der Bedingung, eine Klasse hätte viele Attribute, besser differenziert und der Ausdruck „viel“ treffender beschrieben werden, als mit einer Schranke wie „NOA $>$ 25“. Der Wert der Zugehörigkeitsfunktion $\mu_{\text{Many}}(x)$ für die Attributanzahl x einer Klasse kann für die Bewertung einer Instanz des Musters „Large Class“ verwendet werden. Klassen mit mehr als 25 Attributen müssen nicht alle gleich bewertet werden, stattdessen können Klassen mit einer besonders hohen Attributanzahl höher als andere bewertet werden. Auch Klassen mit 25 oder weniger Attributen können als Bad Smell „Large Class“ erkannt werden. Ihre durch die Zugehörigkeitsfunktion definierte Bewertung fällt aber entsprechend gering aus. Der Reverse Engineer muss sich nicht mehr auf eine Schranke festlegen, um den Begriff „viel“ zu beschreiben.

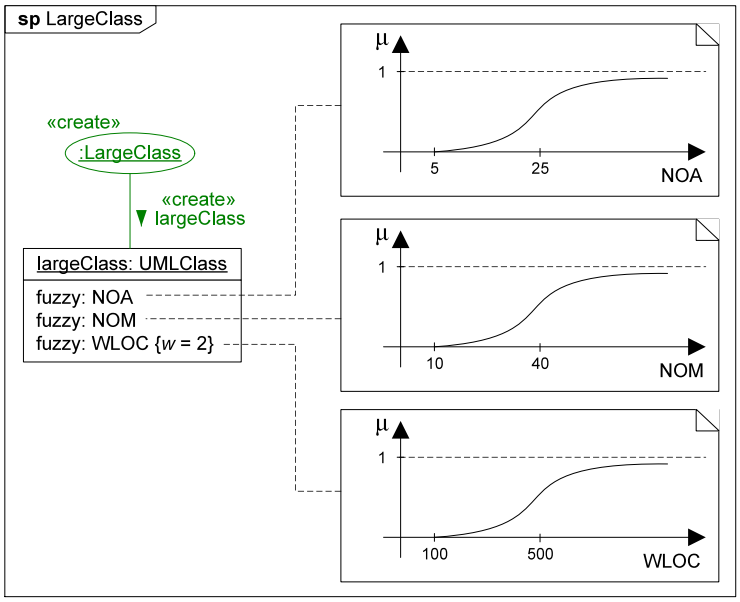


Abbildung 4.7: Musterregel zum Bad Smell „Large Class“ mit Fuzzy-Bedingungen

Zur Spezifikation einer Fuzzy-Bedingung in einer Musterregel wird eine Zugehörigkeitsfunktion angegeben. Diese bezieht sich immer auf eine Metrik wie „NOA“ oder „SIZE“ und wird durch Angabe der Funktionsgleichung beschrieben (siehe Beispiel im Abschnitt 6.3). Der Erfülltheitsgrad der Bedingung wird durch die Zugehörigkeitsfunktion bestimmt. Eine Fuzzy-Bedingung ist (zumindest zum Teil) erfüllt, wenn ihr Erfülltheitsgrad größer ist als Null, andernfalls ist die Bedingung nicht erfüllt.

Der Bad Smell „Large Class“ kann nun wie in der Abbildung 4.7 spezifiziert werden. Die Zugehörigkeitsfunktionen sind hier nur informell durch Kommentare und Funktionsgraphen beschrieben. Praktisch werden die Zugehörigkeitsfunktionen durch Wahl und Parametrisierung einer von mehreren implementierten mathematischen Funktionen angegeben (siehe Abschnitte 5.2 und 6.3).

4.4.4 Gewichte

Um dem Reverse Engineer eine Möglichkeit zu geben, bestimmte Teile einer Musterregel als besonders wichtig oder unwichtig hervorzuheben, können Elemente für die Bewertung der zugehörigen Musterinstanzen mit einem frei wählbaren nichtnegativen Faktor $w \in \mathbb{R}^+$ gewichtet werden. Dieser Faktor wird im Folgenden *Gewichtsfaktor* genannt. Wird zum Beispiel der Gewichtsfaktor 2 für eine Attributbedingung spezifiziert, so wird diese bei der Bewertung einer Musterinstanz doppelt gewertet.

Tabelle 4.2: Mit einem Gewicht versehbare Elemente

Element	mit einem Gewicht versehbar
Attributbedingung	•
Metrikbedingung	•
Fuzzy-Bedingung	•
Constraint (ausgenommen maybe-Constraints)	•
Objektknoten	•
Annotationsknoten	•
Objektmengenknoten	•
Annotationsmengenknoten	•
optionales Fragment (Teilgraph)	•
Verknüpfung	○
Pfad	○

Legende: • = ja, ○ = nein

Zu den Elementen, die gewichtet werden können, gehören Knoten, Attribut-, Metrik- und Fuzzy-Bedingungen sowie Constraints – ausgenommen maybe-Constraints – und optionale Fragmente (siehe Tabelle 4.2). Der Gewichtsfaktor wird in geschweiften Klammern hinter dem Objektnamen, der Bedingung beziehungsweise dem Label eines optionalen Fragments in der Form „$\{w = 0.7\}$“ (für das Gewicht 0.7) angegeben.

4.5 Bewertung von Musterfunden

Im Abschnitt 4.3 wird die Bewertung von Musterinstanzen durch die stark vereinfachte Formel 4.2 auf Seite 49 skizziert. Aus der Formel wird allerdings nicht

klar, welchen Einfluss die Bewertung der Vorgänger einer Annotation auf die der Annotation selbst hat. Außerdem wird nicht beschrieben, wie sich die zu einem Mengenknoten gefundenen Objekte auf die Bewertung einer Musterinstanz auswirken. Diese und andere Details bei der Berechnung werden in diesem Abschnitt angesprochen. Ebenso werden die bei der Wahl einer Bewertungsfunktion getroffenen Entscheidungen motiviert und an Beispielen erläutert.

4.5.1 Bewertbare Bedingungen

Die Tabelle 4.3 listet alle Arten von Elementen auf, die in einer Musterregel vorkommen können. Hierbei werden auch die Erweiterungen der Musterspezifikationssprache berücksichtigt. Jedes Element dieser Art in einer Musterregel kann als je eine Bedingung angesehen werden.

Tabelle 4.3: In Musterregeln vorkommende Elemente

Element	spezifizierbar als optional	negiert
Attributbedingung	●	○
Metrikbedingung	●	○
Fuzzy-Bedingung	○	○
Constraint (ausgenommen maybe-Constraints)	●	○
Objektknoten	●	●
Annotationsknoten	●	●
Objektmengenknoten	○	○
Annotationsmengenknoten	○	○
optionales Fragment (Teilgraph)	●	○
Verknüpfung	○	●
Pfad	○	○

Legende: ● = ja, ○ = nein

Optionale Fragmente fassen einen Teil der durch eine Musterregel beschriebenen Objektstruktur zusammen. Sie können den enthaltenen Teilgraphen für die Bewertung speziell gewichten, definieren aber keinerlei Einschränkungen bezüglich der Eigenschaften der zu suchenden Objektstruktur. Deswegen werden optionale Fragmente nicht als Bedingungen angesehen.

Im Rahmen dieser Arbeit konnte kein sinnvolles Beispiel für ein Muster gefunden werden, bei dem auf eine der spezifizierten Verknüpfungen in der gesuchten Objektstruktur verzichtet werden kann. Wird bei der Mustersuche ein Objekt zu einem Knoten gefunden, so müssen auch für sämtliche von dem Knoten ausgehende Kanten[2] entsprechende Verknüpfungen im abstrakten Syntaxgraphen exi-

[2]Für Kanten zu nicht optionalen Knoten müssen immer Verknüpfungen existieren. Für Kanten zu optionalen Knoten müssen diese existieren, wenn ein Objekt zu dem optionalen Knoten existiert.

stieren. Aus diesem Grund werden Verknüpfungen und Pfade bei dem gewählten Ansatz nicht als optional gekennzeichnet und nicht als eigenständige Bedingungen betrachtet. Stattdessen beschreiben diese einen Teil der durch einen Knoten spezifizierten Bedingung, die angibt, dass ein Objekt mit einem bestimmten Typ und bestimmten Verknüpfungen gefunden werden soll.

Alle übrigen Elementarten, nämlich Attribut-, Metrik-, Fuzzy-Bedingungen, Constraints (außer der maybe-Constraints[3]), Objekt- und Annotationsknoten, stellen Bedingungen an die gesuchte Objektstruktur. Kommt in einer Musterregel eine dieser Bedingungen vor, so wird ihr Erfülltheitsgrad für jede Musterinstanz überprüft. Zur Bewertung eines Musterfundes wird der Anteil der erfüllten an den spezifizierten beziehungsweise erfüllbaren Bedingungen bestimmt.

4.5.2 Bewertungsfunktion

Die Signatur einer beliebigen Bewertungsfunktion wird bereits im Abschnitt 4.2 beschrieben. Als einzigen Parameter erhält die Funktion die Annotation, welche die zu bewertende Musterinstanz repräsentiert. Der Bildbereich der Bewertungsfunktion wird allerdings nur durch eine geordnete Menge beschrieben. Bei dem gewählten Ansatz wird eine Musterinstanz durch Bestimmen des Anteils der erfüllten an den erfüllbaren Bedingungen bewertet. Als Bildbereich der Bewertungsfunktion ergibt sich in diesem Fall also das Intervall $[0, 1] \subset \mathbb{R}$.

Kompakt ausgedrückt, kann die Bewertung einer Annotation a durch folgende Formel beschrieben werden:

$$\mathit{Bewertung}(a) = \frac{\mathit{Erfülltheit}(\mathit{Regel}(a), a)}{\mathit{Gewicht}(\mathit{Regel}(a))} \tag{4.3}$$

Hierbei bezeichnet der Ausdruck $\mathit{Regel}(a)$ die Musterregel, welche die durch die Annotation a markierte Objektstruktur im abstrakten Syntaxgraphen beschreibt. $\mathit{Erfülltheit}(\mathit{Regel}(a), a)$ bezeichnet für die Annotation a die gewichtete Summe der Erfülltheitsgrade aller in der Regel $\mathit{Regel}(a)$ definierten Bedingungen, während der Ausdruck $\mathit{Gewicht}(\mathit{Regel}(a))$ die Summe der Gewichte dieser Bedingungen bezeichnet (vgl. Gleichung 4.2, S. 49). Die Berechnung der Werte, die durch die Ausdrücke $\mathit{Erfülltheit}(\mathit{Regel}(a), a)$ und $\mathit{Gewicht}(\mathit{Regel}(a))$ repräsentiert werden, wird im Folgenden beschrieben.

Die Elemente in einer Musterregel sind hierarchisch angeordnet, zum Beispiel kann ein optionales Fragment Objektknoten enthalten, diese wiederum können Attributbedingungen enthalten. Aus diesem Grund werden im Folgenden die Ausdrücke $\mathit{Erfülltheit}(\mathit{Regel}(a), a)$ und $\mathit{Gewicht}(\mathit{Regel}(a))$, welche zusammen mit der Gleichung 4.3 die erarbeitete Bewertungsfunktion beschreiben, induktiv über den hierarchischen Aufbau einer Musterregel definiert.

Um die Komplexität bei der Beschreibung der Bewertungsfunktion zu reduzieren, wird diese im Folgenden zuerst auf einer Teilmenge der in Musterregeln

[3] Da maybe-Constraints keine Einschränkungen bei der Mustererkennung formulieren, sondern diese lockern, werden sie nicht als Bedingungen betrachtet.

vorkommenden Elementarten definiert. Anschließend wird diese Menge sukzessive um weitere Elementarten erweitert und die Formeln zur Beschreibung der Bewertungsfunktion ergänzt.

Alternativ dazu befindet sich im Anhang A ab Seite 113 eine kompakte Beschreibung dieser Bewertungsfunktion. Die Beschreibung ist insbesondere zum Nachschlagen gedacht. Sie enthält keine Erklärungen und definiert die Funktion von Anfang an auf sämtlichen in einer Musterregel vorkommenden Elementarten.

Zur Definition von Gewicht und Erfülltheit wird im Folgenden der Gewichtsfaktor benötigt. Dieser kann für ein Musterregelelement spezifiziert werden, um seinen Einfluss auf die Bewertung von Musterinstanzen zu erhöhen oder zu verringern. Der Gewichtsfaktor eines Elements e wird mit w_e bezeichnet und entspricht dem in der Musterregel für e angegebenen Wert, zum Beispiel durch den Ausdruck „$\{w = 0.7\}$“. Der Wertebereich von w_e ist die Menge der nicht negativen reellen Zahlen $\mathbb{R}^+$. Die Elementarten, die mit einem Gewicht versehen werden können, sind in der Tabelle 4.2 auf Seite 56 aufgelistet. Ist kein Gewichtsfaktor spezifiziert, so gilt $w_e = 1$.

4.5.3 Objektknoten, Attribut- und Metrikbedingungen

Als Erstes werden das Gewicht und die Erfülltheit induktiv auf der Menge der Musterregeln, der nicht negativen Objektknoten sowie der Attribut- und Metrikbedingungen definiert.

Das Gewicht einer Regel wird durch die Summe der Gewichte der darin enthaltenen Objektknoten bestimmt, während das Gewicht eines Knotens durch seinen Gewichtsfaktor und die Summe der Gewichte der in dem Knoten spezifizierten Attribut- und Metrikbedingungen beschrieben wird. Auf diese Weise werden die Gewichte sämtlicher in einer Regel spezifizierten Bedingungen aufsummiert. Im Folgenden wird das Gewicht induktiv definiert.

Da Attribut- und Metrikbedingungen keine weiteren Elemente enthalten können, wird das Gewicht einer Attribut- oder Metrikbedingung b als ihr Gewichtsfaktor w_b definiert:

$$\mathit{Gewicht}(b) = w_b \tag{4.4}$$

Ein Objektknoten k beschreibt ein Objekt eines bestimmten Typen, welches, wie in der Musterregel spezifiziert, mit anderen Objekten verknüpft sein muss. Diese Eigenschaften des Objekts werden als eine Bedingung gewertet und erhalten den Gewichtsfaktor w_k des Knotens als Gewicht. Attribut- und Metrikbedingungen können weitere Eigenschaften des Objekts fordern. Aus diesem Grund wird das Gewicht eines nicht negativen Objektknotens k als die Summe der Gewichte aller zum Knoten gehörenden Bedingungen definiert:

$$\mathit{Gewicht}(k) = w_k + \sum_{e \in \mathit{Elemente}(k)} \mathit{Gewicht}(e) \tag{4.5}$$

Hierbei bezeichnet *Elemente*(k) die Menge der in dem Knoten k spezifizierten Attribut- und Metrikbedingungen.

Schließlich wird das Gewicht einer Musterregel r als die Summe der Gewichte der darin enthalten Elemente definiert:

$$\mathit{Gewicht}(r) = \sum_{e \in \mathit{Elemente}(r)} \mathit{Gewicht}(e) \tag{4.6}$$

Elemente(r) bezeichnet hier die Menge der in der Regel r spezifizierten Objektknoten.

Ähnlich wie das Gewicht wird auch die Erfülltheit einer Annotation zu einer Regel definiert. Sie entspricht der Summe der für eine Annotation errechneten Erfülltheitsgrade der einzelnen in der Musterregel spezifizierten Bedingungen. Um die Gewichte der einzelnen Bedingungen zu berücksichtigen, werden die Erfülltheitsgrade der einzelnen Bedingungen beim Bilden der Summe mit dem zugehörigen Gewicht multipliziert. Die Erfülltheit einer Annotation ist stets kleiner oder gleich dem Gewicht der zugehörigen Regel und wird im Folgenden induktiv definiert.

Attribut- und Metrikbedingungen können nur erfüllt oder nicht erfüllt sein. Die einzig möglichen Erfülltheitsgrade sind damit 0 und 1. Da die Bedingungen auch mit einem Gewichtsfaktor versehen werden können, wird der Erfülltheitsgrad mit dem Gewichtsfaktor der Bedingung multipliziert.

Für eine Attribut- oder Metrikbedingung b und ein Objekt o im abstrakten Syntaxgraphen wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(b, o) = \begin{cases} w_b, & \text{falls die Bedingung } b \text{ für das Objekt } o \\ & \text{erfüllt ist,} \\ 0, & \text{sonst} \end{cases} \tag{4.7}$$

Konnte bei der Mustersuche einem Objektknoten ein Objekt zugeordnet werden, so ist die durch den Knoten angegebene Bedingung, die angibt, dass ein Objekt eines bestimmten Typs gefunden werden soll, erfüllt. Analog zur Berechnung des Gewichts eines Objektknotens k erhält diese Bedingung auch hier das Gewicht w_k. Attribut- und Metrikbedingungen des Knotens werden durch Addieren ihrer gewichteten Erfülltheitsgrade berücksichtigt. Ist kein Objekt zu dem Knoten gefunden worden, so sind alle diese Bedingungen nicht erfüllt und die Erfülltheit beträgt 0.

Für einen nicht negativen Objektknoten k, der in der Musterregel r spezifiziert wurde, sowie eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert

durch:

$$\mathit{Erfülltheit}(k,a) = \begin{cases} w_k + \sum_{e \in \mathit{Elemente}(k)} \mathit{Erfülltheit}(e,o), & \text{falls beim Erstellen der Annotation } a \text{ dem Knoten } k \text{ ein Objekt } o \text{ zugeordnet wurde,} \\ 0, & \text{sonst} \end{cases} \tag{4.8}$$

Wie bei der Definition des Gewichts eines Objektknotens beschrieben, bezeichnet hier $\mathit{Elemente}(k)$ die zum Knoten k gehörenden Attribut- und Metrikbedingungen. Ihre gewichteten Erfülltheitsgrade werden hier durch die Summe ihrer Erfülltheiten beschrieben.

Für eine Annotation und die zugehörige Regel wird die Erfülltheit als die gewichtete Summe der Erfülltheitsgrade aller in der Regel spezifizierten Bedingungen definiert. Diese Summe entspricht der Summe der Erfülltheiten der in der Regel spezifizierten Objektknoten.

Für eine Musterregel r und eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(r,a) = \sum_{e \in \mathit{Elemente}(r)} \mathit{Erfülltheit}(e,a) \tag{4.9}$$

Die Elemente der Regel r sind hier die in r spezifizierten Objektknoten.

Beispiel

Die Bewertungsfunktion ist nun vollständig auf Musterregeln definiert, die ausschließlich nicht negative Objektknoten sowie Attribut- und Metrikbedingungen enthalten können. Wie Musterinstanzen mit Hilfe dieser Funktion bewertet werden, wird im Folgenden an einem kleinen Beispiel vorgeführt.

In der Abbildung 4.4 auf Seite 53 ist eine Musterregel zu dem Bad Smell „Large Class“ dargestellt. Diese enthält nur einen Objektknoten sowie drei zum Knoten gehörende Metrikbedingungen.

Angenommen, bei der Mustererkennung wäre eine Instanz des Bad Smells „Large Class“ gefunden worden. In diesem Fall wird durch Annotieren eines UMLClass-Objekts in dem abstrakten Syntaxgraphen der untersuchten Software das Vorkommen der Musterinstanz markiert. Wenn für die durch das Objekt repräsentierte Klasse die Metrikwerte NOA = 10 (NOA = Number of Attributes), NOM = 45 (NOA = Number of Methods) und WLOC = 700 (WLOC = Lines of Code) gelten, ergibt sich folgende Rechnung zur Bewertung der Musterinstanz. Die erstellte Annotation wird im Folgenden mit a, das annotierte Objekt mit o und die Musterregel „Large Class“ abkürzend mit r bezeichnet.

Als Erstes soll das Gewicht der Musterregel r bestimmt werden. Dazu werden die Gewichte der einzelnen in r enthaltenen Elemente bestimmt und aufsummiert.

Laut Definition entspricht das Gewicht einer Metrikbedingung ihrem Gewichtsfaktor. Da für die Bedingungen „NOA > 25“ und „NOM > 40“ in der Musterregel r („LargeClass“) kein Gewichtsfaktor angegeben wurde, wird der Standardgewichtsfaktor 1 verwendet. Für die Bedingung „WLOC > 500“ ist der Wert 2 als Gewichtsfaktor spezifiziert worden.

Für den Knoten largeClass (siehe Abb. 4.4, S. 53) wurde ebenfalls kein Gewichtsfaktor spezifiziert, damit gilt $w_{\mathsf{largeClass}} = 1$. Sein Gewicht entspricht der Summe aus den Gewichten der Metrikbedingungen und des Gewichtsfaktors $w_{\mathsf{largeClass}}$:

$$\begin{aligned} \mathit{Gewicht}(\mathsf{largeClass}) &= w_{\mathsf{largeClass}} + \sum_{e \in \mathit{Elemente}(\mathsf{largeClass})} \mathit{Gewicht}(e) \\ &= 1 + (\underbrace{1}_{\text{NOA>25}} + \underbrace{1}_{\text{NOM>40}} + \underbrace{2}_{\text{WLOC>500}}) \\ &= 5 \end{aligned}$$

Da die Musterregel r („LargeClass“) nur einen Knoten enthält, entspricht das Gewicht der Regel dem des Knotens largeClass:

$$\begin{aligned} \mathit{Gewicht}(r) &= \sum_{e \in \mathit{Elemente}(r)} \mathit{Gewicht}(e) \\ &= \mathit{Gewicht}(\mathsf{largeClass}) \\ &= 5 \end{aligned}$$

Als Nächstes soll die Erfülltheit der Annotation a bestimmt werden, welche die zu bewertende Instanz des Musters „Large Class“ repräsentiert. Dazu werden die Erfülltheiten der einzelnen Elemente der Regel bezüglich der Annotation bestimmt.

Da die annotierte Klasse die Metrikwerte NOA = 10, NOM = 45 und WLOC = 700 besitzt, ist die optionale Bedingung „NOA > 25“ nicht erfüllt, während die Bedingungen „NOM > 40“ und „WLOC > 500“ erfüllt sind. Die Erfülltheit der nicht erfüllten Bedingung entspricht dem Wert 0, die der anderen Bedingungen entspricht dem jeweiligen Gewichtsfaktor.

Es wurde ein Objekt o zu dem Objektknoten largeClass gefunden, damit ergibt sich die Erfülltheit des Knotens bezüglich der erstellten Annotation a wie folgt:

$$\begin{aligned} \mathit{Erfülltheit}(\mathsf{largeClass}, a) &= w_{\mathsf{largeClass}} + \sum_{e \in \mathit{Elemente}(\mathsf{largeClass})} \mathit{Erfülltheit}(e, o) \\ &= 1 + (\underbrace{0}_{\text{NOA>25}} + \underbrace{1}_{\text{NOM>40}} + \underbrace{2}_{\text{WLOC>500}}) \\ &= 4 \end{aligned}$$

Analog zum Gewicht der Regel r ergibt sich auch ihre Erfülltheit bezüglich der

erstellten Annotation a:

$$\begin{aligned} \mathit{Erfülltheit}(r, a) &= \sum_{e \in \mathit{Elemente}(r)} \mathit{Erfülltheit}(e, a) \\ &= \mathit{Erfülltheit}(\mathsf{largeClass}, a) \\ &= 4 \end{aligned}$$

Als Bewertung der Musterinstanz, die durch die Annotation a repräsentiert und durch die Musterregel r beschrieben wird, ergibt sich durch folgende Rechnung der Wert 0.8 beziehungsweise 80%:

$$\begin{aligned} \mathit{Bewertung}(a) &= \frac{\mathit{Erfülltheit}(\mathit{Regel}(a), a)}{\mathit{Gewicht}(\mathit{Regel}(a))} \\ &= \frac{\mathit{Erfülltheit}(r, a)}{\mathit{Gewicht}(r)} \\ &= \frac{4}{5} \\ &= 0.8 = 80\% \end{aligned}$$

Um die Bewertungsfunktion vollständig zu beschreiben, soll die Menge der Elementarten, auf denen die Bewertungsfunktion nun definiert ist, im Folgenden sukzessive erweitert werden.

4.5.4 Fuzzy-Bedingungen

Das Gewicht einer Fuzzy-Bedingung b wird analog zu dem Gewicht von Attribut- und Metrikbedingungen als der Gewichtsfaktor von b definiert:

$$\mathit{Gewicht}(b) = w_b \tag{4.10}$$

Eine Fuzzy-Bedingung bezieht sich immer auf eine Metrik, zum Beispiel die Metrik „NOA“ (Number of Attributes) für Klassen. Der Erfülltheitsgrad einer Fuzzy-Bedingung ist im Gegensatz zu dem von Attribut- und Metrikbedingungen nicht auf die Werte 0 und 1 beschränkt, sondern wird durch einen Wert aus dem Intervall $[0, 1]$ angegeben. Die Zugehörigkeitsfunktion μ der Bedingung definiert für jeden möglichen Metrikwert, zu welchem Grad die Bedingung erfüllt wäre, indem sie einen Metrikwert wie NOA = 25 auf einen Wert im Intervall $[0, 1]$ abbildet. Um das Gewicht der Bedingung zu berücksichtigen, wird der Erfülltheitsgrad mit dem Gewichtsfaktor der Bedingung multipliziert.

Die Erfülltheit einer Fuzzy-Bedingung b bezüglich einer Metrik m, eine für b definierte Zugehörigkeitsfunktion μ_b, ein Objekt o im abstrakten Syntaxgraphen und den Metrikwert $x_{m,o}$ für die Metrik m und das Objekt o wird damit wie folgt definiert:

$$\mathit{Erfülltheit}(b, o) = w_b \cdot \mu_b(x_{m,o}) \tag{4.11}$$

Fuzzy-Bedingungen können unter anderem für Objektknoten definiert werden. Aus diesem Grund bezeichnet von nun an der Ausdruck *Elemente*(k) in den Gleichungen 4.5 und 4.8 (auf den Seiten 59 und 61) für einen Objektknoten k die Menge der in dem Knoten k spezifizierten Attribut-, Metrik- und Fuzzy-Bedingungen.

Beispiel

Eine Musterregel, bei der Fuzzy-Bedingungen verwendet werden, ist die in der Abbildung 4.7 (S. 55) dargestellte Regel zum Bad Smell „Large Class“. Diese gleicht der Regel, die im vorhergehenden Abschnitt als Beispiel verwendet wurde (siehe Abschnitt 4.5.3 ab Seite 61). Der einzige Unterschied ist, dass anstatt der Metrikbedingungen Fuzzy-Bedingungen verwendet wurden, die den Begriff „viel“ (viele Attribute, Methoden und Code-Zeilen) durch eine nicht lineare Funktion beschreiben (vgl. Abb. 4.4, S. 53).

Die Bewertung einer Instanz zu dem Muster „Large Class“ kann analog zu der Rechnung im vorhergehenden Abschnitt durchgeführt werden. Angenommen, eine als Instanz des Musters „Large Class“ erkannte Klasse hätte, wie im vorhergehenden Beispiel, die Metrikwerte NOA = 10, NOM = 45 und WLOC = 700. Dann wird diese wie folgt bewertet. Die Annotation, welche die Fundstelle markiert, wird im Folgenden mit a, das annotierte UMLClass-Objekt mit o und die Musterregel „Large Class“ mit r bezeichnet.

Als Gewicht der Regel erhält man wieder den Wert 5. Die Erfülltheiten der drei Fuzzy-Bedingungen erhält man durch einsetzen der Metrikwerte in die jeweiligen Zugehörigkeitsfunktionen, die im Folgenden mit μ_{NOA}, μ_{NOM} und μ_{WLOC} bezeichnet werden. Für die Werte $\mu_{\text{NOA}} = 0.1$, $\mu_{\text{NOM}} = 0.7$ und $\mu_{\text{WLOC}} = 0.8$ wird die Erfülltheit des Objektknotens zur Annotation r wie folgt berechnet:

$$\begin{aligned} \mathit{Erfülltheit}(\mathsf{largeClass}, a) &= w_{\mathsf{largeClass}} + \sum_{e \in \mathit{Elemente}(\mathsf{largeClass})} \mathit{Erfülltheit}(e, o) \\ &= 1 + (1 \cdot \mu_{\text{NOA}}(10) + 1 \cdot \mu_{\text{NOM}}(45) + 2 \cdot \mu_{\text{WLOC}}(700)) \\ &= 1 + (0.1 + 0.7 + 2 \cdot 0.8) \\ &= 3.4 \end{aligned}$$

Die Erfülltheit der Annotation a wird analog zum vorhergehenden Beispiel berechnet. Als Ergebnis erhält man den Wert 3.4. Die Bewertung der Annotation a ist somit $\frac{3.4}{5} = 0.68 = 68\%$.

4.5.5 Constraints

Ein Constraint enthält keine weiteren Bedingungen. Das Gewicht eines Constraints c wird deswegen als sein Gewichtsfaktor definiert:

$$\mathit{Gewicht}(c) = w_c \tag{4.12}$$

Die Erfülltheit eines Constraints wird analog zu der von Attribut- und Metrikbedingungen definiert. Für ein Constraint c, das in der Musterregel r spezifiziert wurde, sowie eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(c, a) = \begin{cases} w_c, & \text{falls das Constraint } c \text{ für die Annotation } a \\ & \text{erfüllt ist,} \\ 0, & \text{sonst} \end{cases} \tag{4.13}$$

Constraints werden außerhalb von Objekten in einer Musterregel definiert. Der Ausdruck $\mathit{Elemente}(r)$ in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints und der nicht negativen Objektknoten.

4.5.6 Annotationsknoten

Ein Annotationsknoten in einer Musterregel beschreibt einen Teil der in einem abstrakten Syntaxgraphen gesuchten Objektstruktur. Diese wird durch eine weitere, zum Annotationsknoten gehörende Musterregel spezifiziert, wodurch eine Abhängigkeit zwischen den Musterregeln und den zugehörigen Annotationen entsteht.

Annotationsabhängigkeiten und die Polymorphie-Eigenschaft von Annotationsknoten erschweren die Bewertung von Musterinstanzen. Aus diesem Grund wird vor der Definition von Gewicht und Erfülltheit bei Annotationsknoten die Problematik bei der Bewertung von Musterinstanzen zu Musterregeln mit Annotationsknoten beschrieben. Die bei der Wahl der Bewertungsfunktion getroffenen Entscheidungen werden motiviert und das Vorgehen bei der Bewertung an Beispielen erläutert.

Weil ein Annotationsknoten eine durch eine andere Musterregel spezifizierte Struktur beschreibt, definiert dieser eigentlich nicht nur eine, sondern mehrere Bedingungen, nämlich die der zugehörigen Regel. Zum Beispiel definiert der Delegation-Knoten der Musterregel „State“ (siehe Abb. 4.2, S. 47) alle durch die Regel „Delegation“ (siehe Abb. B.7, S. 123) spezifizierten Bedingungen. Aus diesem Grund soll bei der Bewertung einer State-Annotation ihre Vorgängerannotation vom Typ Delegation einen entsprechend hohen Einfluss auf das Bewertungsergebnis erhalten. Dazu geht ein Delegation-Annotationsknoten mit dem Gewicht der Musterregel „Delegation“ – also mit der Summe der Gewichte aller in der Musterregel „Delegation“ spezifizierten Bedingungen – in die Bewertung ein und erhält dadurch ein höheres Gewicht als zum Beispiel eine Attributbedingung oder ein Objektknoten. Auf diese Weise spiegelt die Bewertung einer Musterinstanz – im Gegensatz zu dem Ansatz aus Abschnitt 3.2 – den Anteil der erfüllten Bedingungen korrekt wieder. Die Bestimmung des Gewichts eines Annotationsknotens

erweist sich jedoch im Allgemeinen aufgrund seiner Polymorphie-Eigenschaft als schwierig.

Musterregeln können von einander erben. Zum Beispiel erbt die Musterregel „ArrayReference“ von „MultiReference“ (siehe Abb. 2.12, S. 22). Dadurch kann die durch einen Annotationsknoten vom Typ MultiReference beschriebene Objektstruktur entweder durch eine MultiReference- oder eine ArrayReference-Annotation repräsentiert werden. Die Gewichte der Regeln „MultiReference“ und „ArrayReference“ sind jedoch unterschiedlich und die Summe der erfüllbaren Bedingungen hängt von der dem Annotationsknoten zugeordneten Annotation ab. Ist dieser Knoten optional, so muss ihm keine Annotation zugeordnet werden. In dem Fall, dass solch eine Annotation fehlt, kann das Gewicht der dadurch nicht erfüllten Bedingung nicht bestimmt werden.

Als Ausweg wurde entschieden, das Gewicht eines Annotationsknotens mit dem Durchschnitt der Gewichte aller in Frage kommenden Regeln zu beschreiben (siehe Gleichung 4.14). Für jeden Typ, den eine dem Knoten zugeordnete Annotation haben kann, wird das Gewicht der zugehörigen Musterregel berechnet. Anschließend wird aus diesen Gewichten der Durchschnitt gebildet. Einem Annotationsknoten vom Typ MultiReference zum Beispiel könnte aufgrund der Vererbungshierarchie eine MultiReference- oder eine ArrayReference-Annotation zugeordnet werden. Das Gewicht dieses Knotens wird als der Durchschnitt aus den Gewichten der beiden Regeln bestimmt. Um den Gewichtsfaktor zu berücksichtigen, der für einen Annotationsknoten spezifiziert werden kann, wird der Durchschnitt der Regelgewichte mit diesem Faktor multipliziert.

Das Gewicht eines nicht negativen Annotationsknotens k wird als das Produkt aus dem Gewichtsfaktor von k und dem Durchschnitt der Gewichte von allen nicht abstrakten Unterregeln der zum Annotationsknoten k gehörenden Musterregel definiert:

$$Gewicht(k) = w_k \cdot \frac{\sum_{r \in NAUnterregeln(k)} Gewicht(r)}{|NAUnterregeln(k)|} \tag{4.14}$$

Hierbei bezeichnet der Ausdruck $NAUnterregeln(k)$ für einen Annotationsknoten k und die Musterregel r, welche die durch den Knoten k beschriebene Objektstruktur im abstrakten Syntaxgraphen spezifiziert, die Menge der nicht abstrakten Unterregeln von r, die Regel r eingeschlossen.

Durch diese Definition erhält ein Annotationsknoten ein Gewicht, das den durch den Knoten definierten Bedingungen entspricht. Das Gewicht ist bei der Bewertung sämtlicher Annotationen gleich, wodurch ein Vergleich der Bewertungen zweier Annotationen auch dann möglich ist, wenn einem Annotationsknoten Annotationsobjekte unterschiedlichen Typs zugeordnet wurden (zum Beispiel eine MultiReference-Annotation in dem einen Fall und eine ArrayReference-Annotation in dem anderen). Außerdem kann das Gewicht auch dann bestimmt werden, wenn eine zu einem optionalen Annotationsknoten zugeordnete Annotation fehlt.

Zur Berechnung der Erfülltheit einer Annotation müssen auch ihre Vorgängerannotationen untersucht werden. Bei der Spezifikation des Design Patterns „State“

zum Beispiel (siehe Abb. B.1, S. 121) wird die Hilfsmusterregel „Delegation“ (siehe Abb. B.7, S. 123) verwendet. Diese enthält unter anderem einen Annotationsknoten vom Typ SingleReference, wodurch eine entsprechende Annotation in der gesuchten Objektstruktur vorausgesetzt wird. Eine Delegation-Annotation hat also immer eine SingleReference-Annotation als direkten Vorgänger. Wäre der Annotationsknoten optional, so könnte dieser Vorgänger auch fehlen.

Jede Annotation wird einzeln bewertet. Wird eine SingleReference-Annotation gering bewertet, zum Beispiel weil eine Zugriffsmethode nicht erkannt wurde (siehe Musterregel „SingleReference“ in Abb. B.8, S. 123), so ist ein Teil der durch die Musterregel „SingleReference“ definierten Bedingungen nicht erfüllt. Ist diese Annotation der Vorgänger einer Delegation-Annotation, so ist auch ein Teil der durch die Musterregel „Delegation“ definierten Bedingungen nicht erfüllt, was durch eine entsprechend niedrige Bewertung der Delegation-Annotation berücksichtigt werden soll.

Um eine effiziente Berechnung zu ermöglichen (siehe Anforderung 4 im Abschnitt 4.1), sollen Bewertungsergebnisse wiederverwendet werden. Eine rekursive Bewertungsfunktion verwendet dazu – wie im Abschnitt 4.2 beschrieben – die Bewertungen der Vorgänger einer Annotation wieder. Der Erfülltheitsgrad der durch einen Annotationsknoten spezifizierten Bedingung wird durch die Bewertung der dem Knoten zugeordneten Annotation (diese liegt stets in dem Intervall $[0, 1]$) ausgedrückt (falls so eine Annotation existiert). Bei einer fehlenden Annotation ist der Erfülltheitsgrad 0. Die Bewertung einer Annotation hängt auf diese Weise unter anderem auch von den Bewertungen ihrer Vorgängerannotationen ab. Um auch bei der Erfülltheit das Gewicht eines Annotationsknotens zu berücksichtigen, wird der Erfülltheitsgrad mit dem Gewicht des Knotens multipliziert.

Für einen nicht negativen Annotationsknoten k, der in der Musterregel r spezifiziert wurde, sowie eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(k, a) = \begin{cases} \mathit{Gewicht}(k) \cdot \mathit{Bewertung}(o), & \text{falls beim Erstellen der Annotation } a \text{ dem Knoten } k \text{ eine Annotation } o \text{ zugeordnet wurde,} \\ 0, & \text{sonst} \end{cases} \tag{4.15}$$

Da nun das Gewicht und die Erfülltheit auch für Annotationsknoten definiert sind, kann die Menge der Elementarten, auf denen die Bewertungsfunktion definiert ist, um Annotationsknoten erweitert werden. Der Ausdruck $\mathit{Elemente}(r)$ in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints sowie der nicht negativen Objekt- und Annotationsknoten.

Beispiel

Im Folgenden wird die Bewertung einer Musterinstanz an der Beispielmusterregel aus Abbilding 4.8 vorgeführt.

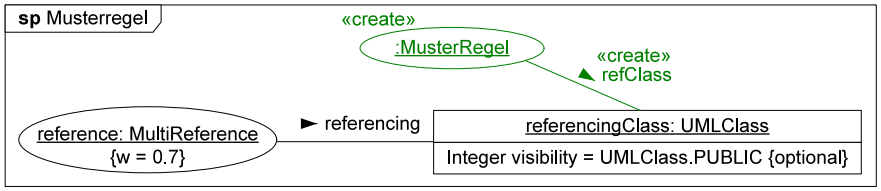


Abbildung 4.8: Beispielmuster mit Annotationsknoten

Um das Gewicht des Annotationsknotens reference zu berechnen, wird als Erstes bestimmt, welche Typen ein Annotationsobjekt annehmen kann, das dem Knoten zugeordnet wird. Die einzige von der Regel „MultiReference“ erbende Regel ist „ArrayReference“, die möglichen Typen des Annotationsobjekts sind damit MultiReference und ArrayReference. Als Nächstes wird das Gewicht der Regeln „MultiReference“ und „ArrayReference“ bestimmt. Bei diesem Beispiel wird von den Gewichten 21 für die Regel „MultiReference“ und 15 für die Regel „ArrayReference“ ausgegangen. Das Gewicht des Knotens ergibt sich als der Durchschnitt aus den Gewichten der Regeln „MultiReference“ und „ArrayReference“ multipliziert mit dem Gewichtsfaktor des Knotens. In der folgenden Rechnung werden diese Regeln abkürzend mit mRef und aRef bezeichnet.

$$
\begin{aligned}
\mathit{Gewicht}(\mathsf{reference}) &= w_{\mathsf{reference}} \cdot \frac{\sum_{r \in \mathit{NAUnterregeln}(\mathsf{reference})} \mathit{Gewicht}(r)}{|\mathit{NAUnterregeln}(\mathsf{reference})|} \\
&= w_{\mathsf{reference}} \cdot \frac{\mathit{Gewicht}(\mathsf{mRef}) + \mathit{Gewicht}(\mathsf{aRef})}{2} \\
&= 0.7 \cdot \frac{21 + 15}{2} \\
&= 12.6
\end{aligned}
$$

Die Summe der Gewichte der beiden Regelknoten ergibt das Gewicht der Regel, die im Folgenden abkürzend mit r bezeichnet wird. Das Gewicht des Knotens referencingClass ist 2. Man erhält $\mathit{Gewicht}(r) = 12.6 + 2 = 14.6$.

Die Berechnung der Erfülltheit soll an einer Annotation a zur Regel aus der Abbildung 4.8 vorgeführt werden. Diese ordnet dem Knoten reference ein Annotationsobjekt m vom Typ ArrayReference mit der Bewertung $\mathit{Bewertung}(o) = 91\%$ zu, während dem Knoten referencingClass ein Objekt o zugeordnet wird, das die optionale Attributbedingung nicht erfüllt.

Die Erfülltheit des Knotens reference bei der Annotation a wird wie folgt be-

rechnet:

$$\begin{aligned} \mathit{Erfülltheit}(\mathsf{reference}, a) &= \mathit{Gewicht}(\mathsf{reference}) \cdot \mathit{Bewertung}(o) \\ &= 12.6 \cdot 0.91 \\ &= 11.466 \end{aligned}$$

Die Attributbedingung des Knotens referencingClass ist nicht erfüllt, damit entspricht die Erfülltheit des Knotens seinem Gewichtsfaktor $w_{\mathsf{referencingClass}} = 1$. Für die Regel r ergibt sich damit:

$$\begin{aligned} \mathit{Erfülltheit}(r, a) &= \sum_{e \in \mathit{Elemente}(r)} \mathit{Erfülltheit}(e, a) \\ &= \mathit{Erfülltheit}(\mathsf{reference}, a) + \mathit{Erfülltheit}(\mathsf{referencingClass}, a) \\ &= 11.466 + 1 \\ &= 12.466 \end{aligned}$$

Als Bewertung der Annotation a erhält man schließlich $\frac{12.466}{14.6} \approx 0.85 = 85\%$.

Bei diesem Beispiel wird deutlich, dass ein Annotationsknoten bei der Bewertung ein Gewicht erhält, das den Anteil der durch den Knoten spezifizierten Bedingungen wiederspiegelt. Dadurch erhält ein Annotationsknoten einen höheren Einfluss auf die Bewertung als ein Objektknoten oder eine Attributbedingung. Mit Hilfe eines Gewichtsfaktors kann dieser Einfluss erhöht oder reduziert werden. Im Gegensatz zu dem Ansatz aus [Wen05a] entspricht die Bewertung einer Musterinstanz dem Anteil der erfüllten Bedingungen (vgl. Abschnitt 3.2.2).

4.5.7 Negative Knoten

Objekt- und Annotationsknoten können in einer Musterregel auch negiert vorkommen. Negative Knoten beschreiben einzelne Objekte oder Objektstrukturen (repräsentiert durch je eine Annotation) im abstrakten Syntaxgraphen, die zur Erkennung des spezifizierten Musters nicht existieren dürfen.

Um ein Objekt genauer zu beschreiben, können bei einem Objektknoten zusätzlich zu dem Typ des Objekts unter anderem auch Attributbedingungen spezifiziert werden. Handelt es sich um einen Annotationsknoten, so repräsentiert dieser eine Objektstruktur, die durch eine Musterregel mit mehreren darin spezifizierten Bedingungen (Knoten, Attributbedingungen, Constraints und andere) beschrieben wird.

Je größer die Anzahl der durch einen negativen Knoten definierten Bedingungen ist, desto spezieller ist das durch den Knoten beschriebene Objekt beziehungsweise die Objektstruktur. Die Anzahl der Einschränkungen für die gesuchte Struktur sinkt. Wird zum Beispiel ein negativer Objektknoten vom Typ UMLClass in einer Musterregel verwendet, so darf an der Stelle des Knotens in der gesuchten Objektstruktur kein UMLClass-Objekt existieren, ganz unabhängig von seinen Eigenschaften. Wird dagegen zusätzlich zu dem negativen Knoten durch eine Attributbedingung spezifiziert, dass die durch das UMLClass-Objekt beschriebene

Klasse abstrakt ist, so wird die durch den Knoten gestellte Einschränkung an die gesuchte Struktur verringert. An der Stelle des Knotens darf nur eine abstrakte Klasse nicht existieren, der Fall ist also spezieller.

Bei der Bewertung einer Musterinstanz soll eine kleinere Einschränkung an die gesuchte Struktur auch ein geringeres Gewicht erhalten, denn es werden weniger Bedingungen gestellt. Je mehr Bedingungen ein negativer Knoten beschreibt, desto kleiner müsste das Gewicht für diesen Knoten bei der Bewertung einer Musterinstanz ausfallen.

Um die Berechnung des Gewichts eines negativen Knotens zu vereinfachen, erhält der Knoten immer das gleiche Gewicht wie eine „atomare“ Bedingung, zum Beispiel eine Attributbedingung oder ein Constraint. Im Fall eines negativen Objektknotens bleiben sämtliche Attribut-, Metrik- und Fuzzy-Bedingungen bei der Bewertung einer Musterinstanz unberücksichtigt, während im Fall eines negativen Annotationsknotens die durch die zugehörige Musterregel definierten Bedingungen unberücksichtigt bleiben. Nur der für den Knoten spezifizierte Gewichtsfaktor hat Einfluss auf die Bewertung einer Musterinstanz.

Das Gewicht eines negativen Objekt- oder Annotationsknotens k wird als der Gewichtsfaktor von k definiert:

$$\mathit{Gewicht}(k) = w_k \tag{4.16}$$

Konnte eine Annotation zu der Musterregel, die einen negativen Knoten enthält, erstellt werden, so ist die durch den Knoten definierte Bedingung erfüllt. Da ein negativer Knoten nicht als optional gekennzeichnet werden kann, gibt es keine Annotation, bei der diese Bedingung nicht erfüllt ist. Aus diesem Grund wird die Erfülltheit eines negativen Knotens durch das Gewicht des Knotens definiert.

Für einen negativen Objekt- oder Annotationsknoten k, der in der Musterregel r spezifiziert wurde, sowie eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(k, a) = w_k \tag{4.17}$$

Die Menge der Elementarten, auf denen die Bewertungsfunktion definiert ist, wird nun um negative Knoten erweitert. Der Ausdruck $\mathit{Elemente}(r)$ in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints sowie aller Objekt- und Annotationsknoten.

4.5.8 Mengenknoten

Musterregeln können unter anderem Mengenknoten enthalten. Ein solcher Knoten repräsentiert eine Menge von Objekten oder Annotationen in einem abstrakten Syntaxgraphen und kann Metrik- und Fuzzy-Bedingungen bezüglich der Metrik

„SIZE“ enthalten. Da diese Metrik nicht auf einzelne Objekte oder Annotationen anwendbar ist, muss die Erfülltheit solcher Bedingungen abweichend von den Gleichungen 4.7 (S. 60) und 4.11 (S. 63) definiert werden. Die Definition des Gewichts stimmt dagegen mit denen aus den Gleichungen 4.4 (S. 59) und 4.10 (S. 63) überein und entspricht dem Gewichtsfaktor der Bedingung.

Für eine Metrikbedingung b bezüglich der Metrik „SIZE“ sowie eine Objekt- oder Annotationsmenge M wird die Erfülltheit wie folgt definiert (vgl. Gleichung 4.7, S. 60):

$$\mathit{Erfülltheit}(b, M) = \begin{cases} w_b, & \text{falls die Bedingung } b \text{ für die Menge } M \\ & \text{erfüllt ist,} \\ 0, & \text{sonst} \end{cases} \tag{4.18}$$

Für eine Fuzzy-Bedingung b bezüglich der Metrik „SIZE“, eine für b definierte Zugehörigkeitsfunktion μ_b, eine Objekt- oder Annotationsmenge M sowie den Metrikwert $x_{m,M}$ für die Metrik m und die Menge M wird die Erfülltheit wie folgt definiert (vgl. Gleichung 4.11 S. 63):

$$\mathit{Erfülltheit}(b, M) = w_b \cdot \mu_b(x_{m,M}) \tag{4.19}$$

Die Berechnungen des Gewichts und der Erfülltheit bei Objektmengen- und Annotationsmengenknoten werden nach dem gleichen Prinzip durchgeführt, die Formeln unterscheiden sich jedoch aufgrund der Unterschiede zwischen Objektmengen- und Annotationsmengenknoten. Im Folgenden wird zuerst das Prinzip bei der Bewertung von Mengenknoten erklärt und die Formeln für Annotationsmengenknoten vorgestellt. Anschließend wird die Berechnung bei Objektmengenknoten beschrieben.

Annotationsmengenknoten

Um zur Bewertung einer Musterinstanz das Verhältnis aus den erfüllten und den erfüllbaren Bedingungen bilden zu können, muss unter anderem die gewichtete Summe der erfüllbaren Bedingungen bestimmt werden, die durch einen Annotationsmengenknoten definiert werden. Diese Summe entspricht dem Gewicht des Knotens.

Die Anzahl der Annotationen, die in einer Menge liegen können, ist unbeschränkt. Eine Mindest- oder Höchstzahl wird nicht immer spezifiziert. Um dennoch ein Gewicht für einen Annotationsmengenknoten angeben zu können, wird dieses auf die gleiche Weise bestimmt wie bei einem Knoten, der eine einzelne Annotation anstatt einer Menge repräsentiert. Es wird also der Durchschnitt der Gewichte von allen nicht abstrakten Unterregeln der zum Annotationsmengenknoten gehörenden Musterregel bestimmt und mit dem Gewichtsfaktor des Knotens multipliziert (vgl. Gleichung 4.14, S. 66). Somit erhält ein Annotationsmengenknoten

den gleichen Einfluss auf die Bewertung einer Musterinstanz wie ein Annotationsknoten.

Um auch die Metrik- und Fuzzy-Bedingungen zu berücksichtigen, die in einem Annotationsmengenknoten zusätzlich spezifiziert werden können (diese beziehen sich immer auf die Metrik „SIZE“), wird die Summe der Gewichte dieser Bedingungen addiert. Das Gewicht eines Annotationsmengenknotens k wird also durch folgende Gleichung definiert:

$$\begin{aligned} \mathit{Gewicht}(k) &= w_k \cdot \frac{\sum_{r \in \mathit{NAUnterregeln}(k)} \mathit{Gewicht}(r)}{|\mathit{NAUnterregeln}(k)|} \\ &+ \sum_{e \in \mathit{MengenBedingungen}(k)} \mathit{Gewicht}(e) \end{aligned} \tag{4.20}$$

Hierbei bezeichnet der Ausdruck $\mathit{MengenBedingungen}(k)$ für den Annotationsmengenknoten k die Menge der in dem Knoten k spezifizierten Mengenbedingungen, also der Metrik- und Fuzzy-Bedingungen bezüglich der Metrik „SIZE“.

Die Erfülltheit der durch einen Annotationsmengenknoten definierten Bedingung soll durch die dem Knoten zugeordneten Annotationsobjekte bestimmt werden. Wie schon in dem Abschnitt 4.3 beschrieben, wird bei dem in dieser Arbeit entwickelten Bewertungsverfahren eine höhere Anzahl der zu einem Mengenknoten gefundenen Objekte (in diesem Fall Annotationsobjekte) als vollständigeres und damit besseres Suchergebnis interpretiert und führt zu einer höheren Bewertung der zugehörigen Musterinstanz.

Bewertungen von Annotationen können sich aufgrund von optionalen Bedingungen unterscheiden. Einem Mengenknoten, der wie der Knoten concreteStates in der Musterregel „State“ (siehe Abb. 4.2, S. 47) eine Menge von Annotationen beschreibt, können somit Annotationsobjekte mit unterschiedlichen Bewertungen zugeordnet werden. Die Bewertungen der Annotationen, die einem Annotationsmengenknoten zugeordnet wurden, sollen genauso wie bei einem Annotationsknoten berücksichtigt werden. Je höher diese ausfallen, desto höher ist die Qualität der zugehörigen Musterinstanz und desto höher soll diese bewertet werden.

Außerdem ist bei Mengenknoten darauf zu achten, dass eine höhere Qualität der zu einem Mengenknoten gefundenen Objekte bei gleich bleibender Anzahl der Objekte ebenso zu einer höheren Bewertung der Musterinstanz führt wie eine größere Anzahl bei gleich bleibender Qualität. Zum Beispiel soll eine Menge von drei Annotationsobjekten mit den Bewertungen 90%, 70% und 30% ebenso wie eine Menge von zwei Annotationsobjekten mit den Bewertungen 90% und 80% zu einer höheren Bewertung der jeweiligen Musterinstanz führen als eine Menge von zwei Annotationsobjekten mit den Bewertungen 90% und 70%.

Um sowohl die Qualität als auch die Anzahl der zu einem Mengenknoten gefundenen Objekte bei der Bewertung einer Musterinstanz zu berücksichtigen, werden die Elemente einer Menge einzeln durch je einen Wert aus dem Intervall $[0, 1]$ bewertet. Die Bewertungen der einzelnen Annotationsobjekte, die einem Annotationsmengenknoten zugeordnet wurden, werden durch die Formel 4.3 auf Seite 58

bestimmt. Die Bewertung einer Annotation a ist also $\mathit{Bewertung}(a) \in [0, 1]$. Anschließend wird die Summe aus den Bewertungen gebildet. Der so entstehende Wert beschreibt die Qualität einer Menge absolut. Die Anzahl und die Qualität der Elemente in einer Menge werden dabei gleichermaßen berücksichtigt.

Um den Anteil der erfüllten Bedingungen korrekt zu beschreiben, muss die Erfülltheit für eine Menge, die einem Mengenknoten zugeordnet wurde, kleiner oder gleich dem Knotengewicht sein. Die Summe der Bewertungen der in der Menge enthaltenen Elemente – diese beschreibt die Qualität der Menge – ist aber nach oben unbeschränkt, da eine Menge beliebig viele Elemente enthalten kann.

Um die Qualität einer Menge nicht absolut, sondern relativ durch einen Wert zwischen 0 und 1 auszudrücken, wird die Summe der Elementbewertungen der Menge durch eine spezielle Skalierfunktion s auf einen Wert im Intervall $[0, 1]$ abgebildet. Die Funktion s ist streng monoton steigend, geht durch den Punkt $(0, 0)$ und hat den Grenzwert $\lim_{x\to\infty} s(x) = 1$.

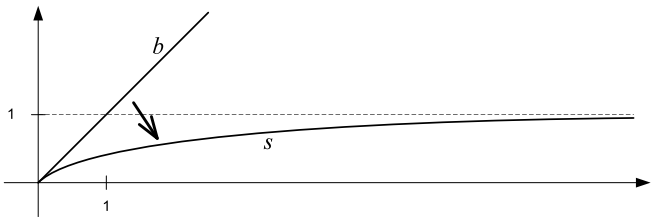


Abbildung 4.9: Abbildung von $\mathbb{R}^+$ auf das Intervall $[0, 1] \subset \mathbb{R}$

Die Graphik 4.9 stellt diese Abbildung dar. Die Summe der Elementbewertungen einer Menge ist eine nicht negative reelle Zahl und liegt damit auf der in der Graphik dargestellten Geraden b. Durch die Funktion s wird dieser Wert auf einen in $[0, 1]$ abgebildet und dadurch skaliert. Die Ordnung der abgebildeten Werte bleibt dabei erhalten. Die Qualität einer beliebig großen Mengen kann so durch einen Wert im Intervall $[0, 1]$ ausgedrückt werden.

Um als Erfülltheit einer Menge zu einem Mengenknoten k einen Wert in dem Intervall $[0, \mathit{Gewicht}(k)]$ zu erhalten, wird zunächst der Wert, der durch Skalieren der Summe der Elementbewertungen einer Menge entsteht, mit dem Gewicht des Mengenknotens multipliziert. An dieser Stelle werden allerdings bei der Berechnung des Mengenknotengewichts die Mengenbedingungen, also die Bedingungen bezüglich der Metrik „SIZE“, ausgeschlossen. Das liegt daran, dass sich diese Bedingungen nicht auf einzelne Objekte beziehen und bei der Bewertung der einzelnen Elemente in einer Menge nicht berücksichtigt werden (vgl. Gleichung 4.25).

Für einen Annotationsmengenknoten k wird das Gewicht des Knotens ohne Mengenbedingungen durch folgende Formel beschrieben (vgl. Gleichung 4.21 auf Seite 72):

$$G(k) = w_k \cdot \frac{\sum_{r \in \mathit{NAUnterregeln}(k)} \mathit{Gewicht}(r)}{|\mathit{NAUnterregeln}(k)|} \tag{4.21}$$

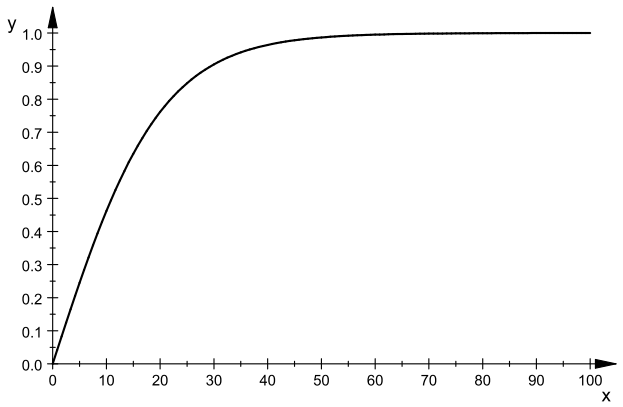


Abbildung 4.10: Graph der Funktion s

Um schließlich auch die Mengenbedingungen zu berücksichtigen, wird zu dem oben beschriebenen Wert die Summe der Erfülltheiten der Mengenbedingungen addiert.

Für einen Annotationsmengenknoten k, der in der Musterregel r spezifiziert wurde, eine Annotation a mit $r = \mathit{Regel}(a)$ sowie der Menge A der beim Erstellen der Annotation a dem Knoten k zugeordneten Annotationen wird die Erfülltheit durch folgende Formel definiert:

$$\begin{aligned} \mathit{Erfülltheit}(k, a) &= s\left(\sum_{b \in A} \mathit{Bewertung}(b)\right) \cdot G(k) \\ &\quad + \sum_{e \in \mathit{MengenBedingungen}(k)} \mathit{Erfülltheit}(e, A) \end{aligned} \tag{4.22}$$

Es gibt viele Möglichkeiten, die Funktion s zu bestimmen, die zum Abbilden der Summe der Bewertungen der Elemente in einer Menge auf das Intervall $[0, 1]$ verwendet wird. Da jedoch die Anzahl der Elemente in einer Menge in den meisten Fällen im Intervall $[0, 50]$ erwartet wird, wird die Funktion so definiert, dass Werte in diesem Intervall auf möglichst weit auseinander liegende Werte im Bildbereich der Funktion s abgebildet werden.

Die Funktion s mit der Signatur $s : \mathbb{R}^+ \to [0, 1)$, $x \mapsto s(x)$ wird wie folgt definiert:

$$s(x) = \frac{2}{1 + e^{-\frac{x}{10}}} - 1 \tag{4.23}$$

Der Graph der Funktion ist in der Abbildung 4.10 dargestellt und zeigt, dass die Werte im Bereich 0 bis 50 auf Werte im Intervall 0 bis ca. 0.99 abgebildet werden.

Da nun das Gewicht und die Erfülltheit auch für Annotationsmengenknoten definiert sind, kann die Menge der Elementarten, auf denen die Bewertungsfunktion definiert ist, um diese Knotenart erweitert werden. Der Ausdruck $\mathit{Elemente}(r)$

in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints, Objekt- und Annotationsknoten sowie Annotationsmengenknoten.

Das vorgestellte Verfahren zur Bewertung von Objektmengen zu einem Mengenknoten bewertet sowohl die Qualität als auch die Anzahl der Objekte in einer Menge. Mit Hilfe der Skalierfunktion s kann die Qualität eines Mengenknotens relativ, also durch einen Prozentwert ausgedrückt werden, obwohl die Kardinalität der Menge unbeschränkt ist. Die Konsequenz daraus ist, dass Annotationen zu Musterregeln mit Mengenknoten nicht mit 100% bewertet werden können. Abhängig von der Anzahl und Qualität der Objekte, die einem Mengenknoten zugeordnet wurden, kann die Bewertung einer Annotation jedoch beliebig nah an den Wert 100% herankommen.

Laut Definition der Bewertungsfunktion werden Musterinstanzen hoch bewertet, wenn die Anzahl der zu einem Annotationsmengenknoten zugeordneten Annotationen und die Qualität der Annotationen hoch ist. Ist dieses Verhalten ausnahmsweise nicht erwünscht, zum Beispiel weil nur die Anzahl der Elemente in einer Menge relevant ist, so können durch Angeben des Gewichtsfaktors 0 für den Annotationsmengenknoten die Bewertungen der dem Knoten zugeordneten Annotationen ignoriert werden. Die Anzahl der Elemente in der Menge kann jedoch weiterhin durch eine in dem Knoten spezifizierte Metrik- oder Fuzzy-Bedingung zur Metrik „SIZE“ (mit einem Gewichtsfaktor größer als 0) zur Bewertung von Musterinstanzen verwendet werden. Das Gleiche gilt auch für die im Folgenden behandelten Objektmengenknoten.

Objektmengenknoten

Analog zu der Berechnung des Gewichts und der Erfülltheit bei Annotationsmengenknoten ist auch das Vorgehen bei Objektmengenknoten.

Das Gewicht eines Objektmengenknotens k wird auf die gleiche Weise wie bei Objektknoten bestimmt und durch folgende Gleichung definiert (vgl. Gleichung 4.5 auf Seite 59):

$$\mathit{Gewicht}(k) = w_k + \sum_{e \in \mathit{Elemente}(k)} \mathit{Gewicht}(e) \tag{4.24}$$

Hierbei bezeichnet der Ausdruck $\mathit{Elemente}(k)$ die Menge der in dem Knoten k spezifizierten Attribut-, Metrik- und Fuzzy-Bedingungen (Bedingungen bezüglich der Metrik „SIZE“ eingeschlossen).

Bei der Berechnung der Erfülltheit eines Objektmengenknotens und der dem Knoten zugeordneten Objekte soll wie bei Annotationsmengenknoten die Qualität der Objekte berücksichtigt werden und durch einen Wert aus dem Intervall $[0, 1]$ ausgedrückt werden. Objektmengenknoten können optionale Bedingungen enthalten, die von den Objekten in einer Menge erfüllt werden sollen, aber nicht müssen. Je mehr optionale Bedingungen ein Objekt erfüllt, desto besser passt dieses auf die Beschreibung und desto höher ist seine Qualität.

Bei Annotationsmengenknoten werden die dem Knoten zugeordneten Annotationen durch *Bewertung*(a) für eine Annotation a bewertet (siehe Formel 4.3 auf Seite 58). Ein Objekt o zu einem Objektmengenknoten k dagegen wird durch die folgende Formel bewertet (vgl. Gleichungen 4.8, S. 61 und 4.5, S. 59):

$$B(k,o) = \frac{w_k + \sum_{e \in NichtMengenBedingungen(k)} Erfülltheit(e,o)}{w_k + \sum_{e \in NichtMengenBedingungen(k)} Gewicht(e)} \tag{4.25}$$

Der Ausdruck *NichtMengenBedingungen*(k) bezeichnet hierbei die Menge der zum Objektmengenknoten k definierten Attribut-, Metrik- und Fuzzy-Bedingungen, die sich nicht auf die Metrik „SIZE“ beziehen, da diese Metrik nur auf eine Menge angewendet werden kann.

Die übrigen Schritte zur Berechnung der Erfülltheit eines Objektmengenknotens und der dem Knoten zugeordneten Objekte sind identisch zu denen bei Annotationsmengenknoten: Die Summe der Bewertungen der einem Objektmengenknoten zugeordneten Objekte wird durch die Funktion s auf einen Wert im Intervall $[0,1]$ abgebildet und das Ergebnis mit dem Gewicht des Objektknotens ohne Mengenbedingungen multipliziert. Anschließend wird zu diesem Produkt die Summe der Erfülltheiten der Mengenbedingungen addiert.

Das Gewicht eines Objektmengenknotens k ohne Mengenbedingungen wird durch folgende Formel beschrieben (vgl. Gleichung 4.24, S. 75):

$$G(k) = w_k + \sum_{e \in NichtMengenBedingungen(k)} Gewicht(e) \tag{4.26}$$

Die Erfülltheit wird für einen Objektmengenknoten k, der in der Musterregel r spezifiziert wurde, eine Annotation a mit $r = Regel(a)$ sowie die Menge O der beim Erstellen der Annotation a dem Knoten k zugeordneten Objekte wie folgt definiert:

$$\begin{aligned} Erfülltheit(k,a) &= s\left(\sum_{o \in O} B(k,o)\right) \cdot G(k) \\ &\quad + \sum_{e \in MengenBedingungen(k)} Erfülltheit(e,O) \end{aligned} \tag{4.27}$$

Der Ausdruck *MengenBedingungen*(k) bezeichnet hier die Menge der zum Objektmengenknoten k spezifizierten Mengenbedingungen, also der Metrik- und Fuzzy-Bedingungen bezüglich der Metrik „SIZE“. Hierbei ist ein besonderes Augenmerk darauf zu richten, dass bei Objektmengenknoten *MengenBedingungen*$(k) \cup$ *NichtMengenBedingungen*$(k) =$ *Elemente*(k) gilt.

Die Menge der Elementarten, auf denen die Bewertungsfunktion definiert ist, wird nun auch um Objektmengenknoten erweitert. Der Ausdruck *Elemente*(r) in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints, Objekt- und Annotationsknoten sowie Objektmengen- und Annotationsmengenknoten.

Beispiel

Im Folgenden wird an der Beispielmusterregel aus Abbildung 4.11 vorgeführt, wie eine Musterinstanz zu einer Musterregel mit Mengenknoten bewertet wird. Die Musterregel beschreibt eine Klasse mit Methoden von besonders hoher Komplexität, was mit Hilfe der McCabe-Metrik[4] CC (zyklomatische Komplexität) [Sof06] ausgedrückt wird.

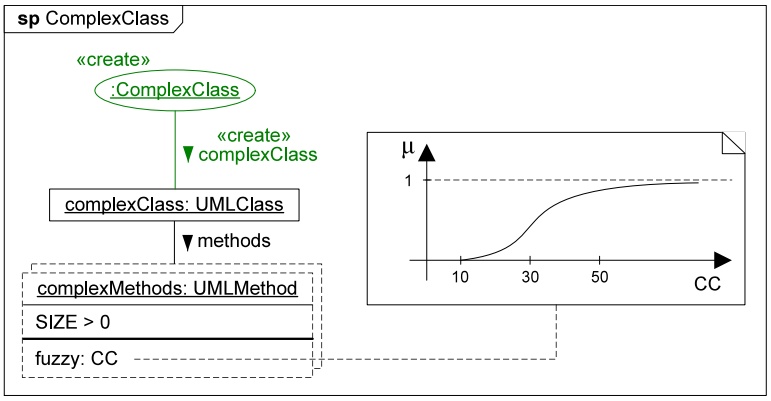


Abbildung 4.11: Beispielmuster mit Mengenknoten

Das Gewicht des Mengenknotens complexMethods entspricht der Summe aus dem Gewichtsfaktor des Knotens und den Gewichtsfaktoren der darin spezifizierten Bedingungen. Da bei dem Beispiel keine Gewichtsfaktoren spezifiziert wurden, wird jeweils der Standardwert 1 verwendet und man erhält den Wert 3 als Gewicht. Das Gewicht des Knotens complexClass ist 1. Damit ist das Gewicht der Regel $3 + 1 = 4$.

Ausgehend von einer Annotation a, die dem Knoten complexMethods drei Objekte o_1, o_2 und o_3 sowie dem Knoten complexClass ein Objekt c zuordnet, wird im Folgenden die Erfülltheit von a berechnet. Für die Objekte werden dabei die folgenden Metrikwerte angenommen: $\mathrm{CC} = 57$ bei Objekt o_1, $\mathrm{CC} = 43$ bei Objekt o_2 und $\mathrm{CC} = 17$ bei Objekt o_3.

Um die Erfülltheit des Knotens complexMethods zu berechnen, werden als Erstes die dem Knoten zugeordneten Objekte einzeln bewertet. Dabei werden nur die Bedingungen berücksichtigt, die sich auf ein Objekt und nicht auf eine Menge beziehen, in diesem Fall wird also nur die Fuzzy-Bedingung berücksichtigt. Diese wird im Folgenden mit b und die Zugehörigkeitsfunktion mit μ_{CC} bezeichnet. Die Bewertung eines Objekts o zu einem Mengenknoten k wird durch folgende Formel bestimmt:

$$B(k,o) = \frac{w_k + \sum_{e \in NichtMengenBedingungen(k)} Erfülltheit(e,o)}{w_k + \sum_{e \in NichtMengenBedingungen(k)} Gewicht(e)}$$

[4]Die zyklomatische Komplexität einer Methode beschreibt die Anzahl der möglichen Pfade durch den Kontrollflussgraphen der Methode. Diese hängt insbesondere von den bedingten Anweisungen in der Methode ab.

Mit den Werten $\mu_{\mathrm{CC}}(57) = 0.9$, $\mu_{\mathrm{CC}}(43) = 0.8$ und $\mu_{\mathrm{CC}}(17) = 0.1$ werden die drei Objekte o_1, o_2 und o_3 wie folgt bewertet:

$$\begin{aligned}
B(\mathsf{complexMethods}, o_1) &= \frac{w_{\mathsf{complexMethods}} + \mathit{Erfülltheit}(b, o_1)}{w_{\mathsf{complexMethods}} + \mathit{Gewicht}(b)} \\
&= \frac{1 + \mu_{\mathrm{CC}}(57)}{1 + 1} \\
&= \frac{1 + 0.9}{2} \\
&= 0.95 \\
B(\mathsf{complexMethods}, o_2) &= \frac{1 + \mu_{\mathrm{CC}}(43)}{2} \\
&= \frac{1 + 0.8}{2} \\
&= 0.9 \\
B(\mathsf{complexMethods}, o_3) &= \frac{1 + \mu_{\mathrm{CC}}(17)}{2} \\
&= \frac{1 + 0.1}{2} \\
&= 0.55
\end{aligned}$$

Nach der Bewertung der einzelnen dem Mengenknoten zugeordneten Objekte wird die Summe der Bewertungen mit Hilfe der Funktion s (siehe Formel 4.23, S. 74) auf einen Wert im Intervall $[0, 1]$ abgebildet. Dabei erhält man den Wert $s(0.95 + 0.9 + 0.55) = s(2.4) \approx 0.12$. Anschließend wird dieser Wert mit dem Gewicht des Knotens complexMethods (ohne die Bedingung SIZE > 0) multipliziert: $0.12 \cdot 2 = 0.24$. Schließlich erhält man durch Addieren der Erfülltheit der Bedingung SIZE > 0 die Erfülltheit des Mengenknotens: $0.24 + 1 = 1.24 = \mathit{Erfülltheit}(\mathsf{complexMethods}, a)$.

Die Erfülltheit des Knotens complexClass beträgt 1. Die Erfülltheit der Regel für die Annotation a entspricht damit dem Wert $1.24 + 1 = 2.24$. Die Annotation wird also mit $\frac{2.24}{4} = 0.56 = 56\%$ bewertet.

Die Bewertung einer Annotation mit identischen Eigenschaften aber dem Metrikwert CC $= 31$ für das Objekt o_3 und $\mu_{\mathrm{CC}}(31) = 0.5$ führt zu dem Ergebnis 56.5%. Bewertet man eine Annotation mit identischen Eigenschaften, bei der dem Knoten complexMethods 30 weitere Objekte mit dem Metrikwert CC $= 57$ zugeordnet wurden, so erhält man 95.5% als Bewertungsergebnis.

Das Beispiel zeigt, dass sowohl die Anzahl als auch die Qualität der Objekte, die einem Mengenknoten zugeordnet sind, bei der Bewertung berücksichtigt werden. Mit Hilfe der Skalierfunktion s kann die Bewertung einer Annotation durch einen Prozentwert ausgedrückt werden, obwohl die Anzahl der Objekte, die einem Mengenknoten zugeordnet werden können, unbeschränkt ist.

4.5.9 Optionale Fragmente

Ein optionales Fragment fasst Knoten und Constraints einer Musterregel zusammen und beschreibt, dass diese nur als Ganzes gefunden werden dürfen. Das Fragment stellt aber keine Anforderungen an die Eigenschaften der gesuchten Objektstruktur. Aus diesem Grund werden optionale Fragmente nicht als Bedingungen gezählt. Ein für ein optionales Fragment spezifizierter Gewichtsfaktor kann jedoch den Einfluss der darin enthaltenen Musterregelelemente auf die Bewertung von Musterinstanzen ändern. Dazu werden das Gewicht und die Erfülltheit eines optionalen Fragments als das Produkt aus dem Gewichtsfaktor des Fragments und der Summe der Gewichte beziehungsweise Erfülltheiten der darin spezifizierten Elemente definiert.

Das Gewicht eines optionalen Fragments f wird wie folgt definiert:

$$\mathit{Gewicht}(f) = w_f \cdot \sum_{e \in \mathit{Elemente}(f)} \mathit{Gewicht}(e) \tag{4.28}$$

Hierbei bezeichnet $\mathit{Elemente}(f)$ die Menge der in dem optionalen Fragment f spezifizierten Constraints, Objekt- und Annotationsknoten sowie Objektmengen- und Annotationsmengenknoten.

Für ein optionales Fragment f, das in der Musterregel r spezifiziert wurde, sowie eine Annotation a mit $r = \mathit{Regel}(a)$ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(f, a) = w_f \cdot \sum_{e \in \mathit{Elemente}(f)} \mathit{Erfülltheit}(e, a) \tag{4.29}$$

Die Menge der Elementarten, auf denen die Bewertungsfunktion definiert ist, wird nun um optionale Fragmente erweitert. Der Ausdruck $\mathit{Elemente}(r)$ in den Gleichungen 4.6 und 4.9 (auf den Seiten 60 und 61) bezeichnet von nun an die Menge der in der Musterregel r spezifizierten Constraints, Objekt-, Annotations-, Objektmengen- und Annotationsmengenknoten sowie optionalen Fragmente.

Beispiel

Welche Auswirkungen der Gewichtsfaktor eines optionalen Fragments auf die Bewertung der darin enthaltenen Elemente hat, wird im Folgenden an der Beispielmusterregel aus Abbildung 4.12 gezeigt.

Die Regel – diese wird im Folgenden mit r bezeichnet – enthält drei Knoten ohne spezifizierten Gewichtsfaktor. Die Gewichte der Knoten sind also jeweils 1. Zwei Knoten sind in einem Fragment f mit dem Gewichtsfaktor $w_f = 0.5$ enthalten.

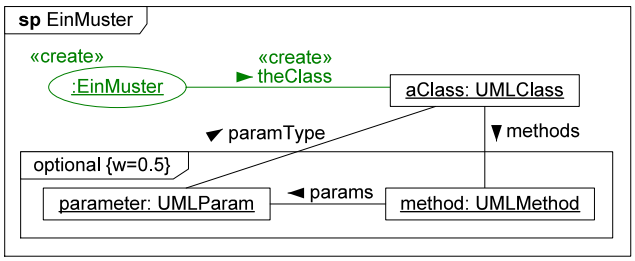


Abbildung 4.12: Beispielmuster mit optionalen Fragmenten

Als Gewicht der Regel r erhält man somit:

$$\begin{aligned}
\mathit{Gewicht}(r) &= \sum_{e \in \mathit{Elemente}(r)} \mathit{Gewicht}(e) \\
&= \mathit{Gewicht}(\mathsf{aClass}) + \mathit{Gewicht}(f) \\
&= 1 + w_f \cdot \sum_{e \in \mathit{Elemente}(f)} \mathit{Gewicht}(e) \\
&= 1 + 0.5 \cdot (\mathit{Gewicht}(\mathsf{parameter}) + \mathit{Gewicht}(\mathsf{method})) \\
&= 1 + 0.5 \cdot (1 + 1) \\
&= 2
\end{aligned}$$

Eine Annotation a, bei der dem Knoten aClass ein Objekt o zugeordnet wurde, während den anderen Knoten keine Objekte zugeordnet wurden, wird wie folgt bewertet.

$$\begin{aligned}
\mathit{Erfülltheit}(r, a) &= \sum_{e \in \mathit{Elemente}(r)} \mathit{Erfülltheit}(e, a) \\
&= \mathit{Erfülltheit}(\mathsf{aClass}, a) + \mathit{Erfülltheit}(f, a) \\
&= 1 + w_f \cdot \sum_{e \in \mathit{Elemente}(f,a)} \mathit{Erfülltheit}(e, a) \\
&= 1 + 0.5 \cdot (\mathit{Erfülltheit}(\mathsf{parameter}) + \mathit{Erfülltheit}(\mathsf{method}, a)) \\
&= 1 + 0.5 \cdot (0 + 0) \\
&= 1
\end{aligned}$$

Als Bewertung der Annotation erhält man damit $\frac{1}{2} = 50\%$. Mit dem Gewichtsfaktor $w_f = 1$ für das optionale Fragment wäre die Bewertung $\frac{1}{3} \approx 33.3\%$

4.6 Zusammenfassung

In diesem Kapitel wurde ein neues Bewertungsverfahren für Musterinstanzen vorgestellt. Bei diesem werden Musterinstanzen durch Angeben des Anteils der erfüllten an den spezifizierten Bedingungen bewertet. Auf diese Weise wird beschrieben, wie gut eine Musterinstanz auf die Spezifikation des gesuchten Musters passt

(Anforderung 1 im Abschnitt 4.1). Im Gegensatz zu dem im Abschnitt 3.1 beschriebenen Ansatz werden die Eigenschaften der Musterinstanzen anstatt der Eigenschaften der Musterregeln bewertet, was die Aussagekraft der Bewertungsergebnisse erheblich steigert.

Um dem Reverse Engineer mehr Möglichkeiten zu bieten, Teile einer Musterspezifikation als optional zu kennzeichnen, wurde die Musterspezifikationssprache erweitert. Dadurch ist es möglich, zusätzlich zu den bisher spezifizierten Informationen auch Eigenschaften von Musterinstanzen zu beschreiben, die mit einer höheren Sicherheit auf einen korrekten Fund hinweisen, jedoch nicht zwingend für jede Musterinstanz gelten. Je mehr dieser Eigenschaften eine Musterinstanz besitzt, desto höher wird diese bewertet.

Durch die Erweiterung der Musterspezifikationssprache um Metrikbedingungen lassen sich auch Software-Metriken zur Beschreibung von Software-Mustern verwenden. Mit Hilfe von Gewichtsfaktoren kann ein Reverse Engineer angeben, dass bestimmte Teile einer Musterspezifikation einen besonders hohen oder einen besonders niedrigen beziehungsweise keinen Einfluss auf die Bewertung von Musterinstanzen haben sollen.

Die Präzision der Musterinstanzbewertungen hängt von der Musterspezifikation ab. Je mehr optionale Teile eine Musterregel enthält, desto mehr Kombinationen gibt es aus erfüllten und nicht erfüllten Bedingungen und desto mehr unterschiedliche Bewertungsergebnisse sind möglich. Um die Präzision zu erhöhen (Anforderung 2), wurden auf Software-Metriken basierende Fuzzy-Bedingungen eingeführt. Im Gegensatz zu anderen Bedingungen kann ihr Erfülltheitsgrad durch einen beliebigen Wert aus dem Intervall $[0, 1]$ ausgedrückt werden. Mit Hilfe einer frei wählbaren Zugehörigkeitsfunktion $\mu : \mathbb{R} \rightarrow [0, 1]$ kann ein Reverse Engineer beliebig genau angeben, wie mögliche Metrikwerte bewertet werden sollen.

Durch die Berücksichtigung der Anzahl aller spezifizierten Bedingungen, die ein Muster beschreiben, die Bedingungen aus Hilfsmusterregeln eingeschlossen, entspricht die Bewertung einer Musterinstanz im Gegensatz zu dem im Abschnitt 3.2 beschriebenen Ansatz dem Anteil der von der Instanz erfüllten Bedingungen. Dadurch kann von einer höheren Bewertung auf eine Musterinstanz mit höherer Qualität geschlossen werden (der Fund passt besser auf die Musterbeschreibung). Außerdem ist auf Basis der Bewertungen ein Vergleich mehrerer Musterinstanzen möglich, unabhängig davon, ob sie Ausprägungen verschiedener Muster sind (Anforderung 2).

Neben optionalen Bedingungen wird auch die Anzahl und Qualität der Objekte bewertet, die einem Mengenknoten zugeordnet werden. Obwohl die Anzahl der Objekte im Allgemeinen nach oben unbeschränkt ist, werden Musterinstanzen immer relativ bewertet.

Die Präzision der Mustererkennung ist nicht gefährdet. Dadurch, dass die bisherigen Musterregeln um optionale Knoten, Fragmente und Bedingungen erweitert werden, steigt bei der Mustersuche weder die Anzahl der False Negatives noch die Anzahl der False Positives (Anforderung 3). Die optionalen Elemente werden nur für die Bewertung der Musterinstanzen genutzt und haben keinen Einfluss darauf,

ob eine Musterinstanz erkannt wird oder nicht.

Um eine effiziente Berechnung zu ermöglichen (Anforderung 4), ist eine rekursive Bewertungsfunktion gewählt worden. Auf diese Weise können Bewertungsergebnisse wiederverwendet werden.

Durch Erweitern der Musterspezifikationen um optionale Bedingungen steigt der menschliche Aufwand im Vergleich zu dem Ansatz aus [NWW03] (siehe auch Abschnitt 1.1). Dieser beschränkt sich jedoch auf die einmalige Spezifikation der Muster und kann dadurch vernachlässigt werden. Außerdem entscheidet ein Reverse Engineer selbst, wie viel er zusätzlich zu den bisher verwendeten Informationen spezifiziert und in vielen Fällen reichen bereits wenige optionale Bedingungen oder Knoten, um ein Muster treffend zu spezifizieren (vgl. Spezifikation des Musters „State“ in Abbildung 4.2 auf Seite 47). Im Vergleich zu dem im Abschnitt 3.1.2 beschriebenen Bewertungsverfahren ist der menschliche Aufwand bei der Analyse der Ergebnisse geringer, weil der Reverse Engineer keine Korrekturen vornehmen muss, um die Präzision der Bewertung zu erhöhen. Außerdem ist aufgrund der präziseren Bewertung die Identifikation der Musterinstanzen mit dem größten Anteil an erfüllten Bedingungen einfacher möglich als bei den im Kapitel 3 beschriebenen Ansätzen.

Zusammenfassend lässt sich sagen, dass mit dem neuen Ansatz eine präzisere Bewertung mit mehr Aussagekraft als bei bisherigen Verfahren erreicht wurde.

5 Technische Realisierung

In diesem Kapitel wird die Realisierung des im Rahmen dieser Diplomarbeit entwickelten Bewertungsverfahrens für Musterinstanzen beschrieben. Im ersten Abschnitt wird ein grober Überblick über das Vorgehen bei der Mustererkennung und der anschließenden Bewertung der Funde gegeben. Die beiden nachfolgenden Abschnitte befassen sich mit der Anpassung der Mustersuche und des Meta-Modells für Musterspezifikationsdiagramme.

5.1 Architektur

Die automatische Erkennung von Software-Mustern erfolgt in Fujaba in mehreren Schritten, die zur Umsetzung des im Kapitel 4 vorgestellten Bewertungsverfahrens angepasst wurden. Die Abbildung 5.1 stellt die wichtigsten Schritte der Mustersuche und den Datenfluss dar (siehe auch Abschnitt 2.3).

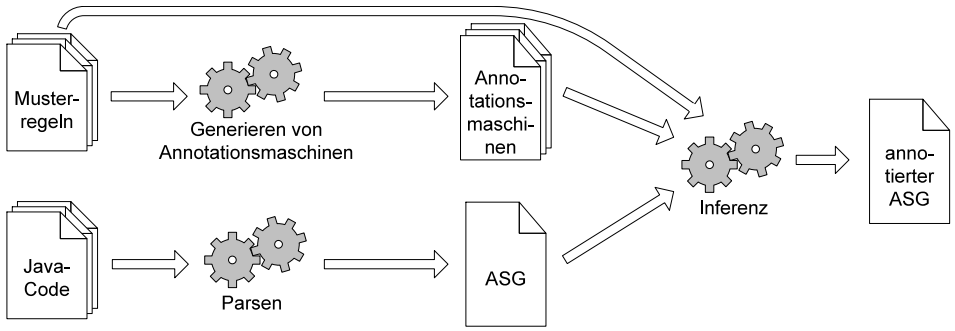


Abbildung 5.1: Datenfluss der automatischen Mustererkennung in Fujaba

Aus den Musterregeln wird die Implementierung von Annotationsmaschinen generiert und der Code kompiliert. Der Quellcode des zu untersuchenden Softwaresystems wird geparst und daraus der abstrakte Syntaxgraph (ASG) des Systems erstellt. Dieser wird anschließend mit Hilfe der Annotationsmaschinen, der Musterregeln und des Inferenzalgorithmus nach den spezifizierten Mustern durchsucht und an den Fundstellen annotiert. Schließlich können die in Form von Annotationen vorliegenden Suchergebnisse tabelarisch und in UML-Klassendiagrammen der analysierten Software dargestellt werden.

Die genannten Schritte werden mit Hilfe mehrerer die Fujaba Tool Suite [Fuj06] erweiternder Plug-ins realisiert. Ein Teil der Plug-ins mit ihren Abhängigkeiten untereinander ist in der Abbildung 5.2 dargestellt. Zusätzlich zu den dargestellten

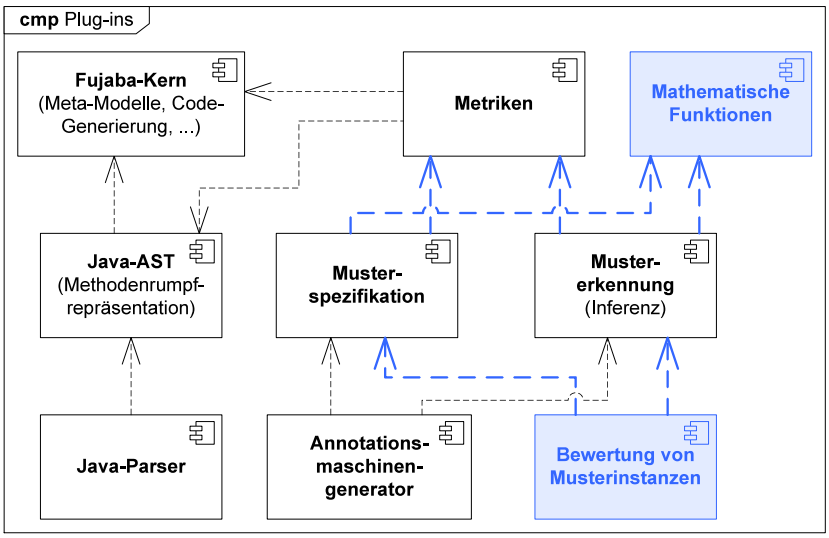


Abbildung 5.2: Plug-ins und Abhängigkeiten

exisitieren weitere Plug-ins für die jeweiligen Benutzerschnittstellen. Diese wurden für die Entwicklungsumgebung *Eclipse* [Ecl06] implementiert.

Der Fujaba-Kern stellt unter anderem Meta-Modelle für Klassen- und so genannte Story-Diagramme [FNTZ98] (eine Art von Aktivitätsdiagrammen) zur Verfügung. Mit Hilfe dieser Diagramme kann sowohl die Struktur als auch das Verhalten einer Software beschrieben und anschließend lauffähiger Java-Code daraus generiert werden.

Das Parsen von Java-Code wird durch die Plug-ins Java-AST und Java-Parser realisiert. Das Musterspezifikations-Plug-in stellt das Meta-Modell für Musterspezifikationsdiagramme (Musterregeln) und Musterkataloge bereit. Das Plug-in Annotationsmaschinengenerator unterstützt die Generierung von Annotationsmaschinen aus Musterregeln, indem ein Klassendiagramm und mehrere Story-Diagramme erstellt werden, aus denen mit Hilfe des Fujaba-Kerns Java-Code generiert wird. Neben mehreren Strategien für die Mustersuche ist in dem Inferenz-Plug-in auch die Bewertung von Musterinstanzen nach dem im Abschnitt 3.1 beschriebenen Ansatz implementiert.

Um die bisherige Funktionalität zu erweitern und anzupassen, wurden weitere Plug-ins entwickelt und existierende angepasst. Die dabei entstandenen Plug-ins und die neuen Plug-in-Abhängigkeiten sind in der Abbildung 5.2 durch einen dunkleren Hintergrund beziehungsweise dickere Linien hervorgehoben. Angepasst wurden die Musterspezifikation, die Generierung von Annotationsmaschinen und die Inferenz. Zur Bewertung von Musterinstanzen wurde ein neues Plug-in entwickelt[1].

[1]Zur Bewertung von Musterinstanzen nach dem vorgestellten Verfahren ist ein Zugriff auf die Musterspezifikationen notwendig. Da das Inferenz-Plug-in von der Musterspezifikation unabhängig bleiben sollte, wurde ein neues Plug-in entwickelt, welches das Verhalten der Inferenz anpasst und von dem Musterspezifikations-Plug-in abhängt.

Das Meta-Modell für Musterspezifikationsdiagramme wurde erweitert. Bei der Realisierung von Metrikbedingungen konnte das Metrik-Plug-in verwendet werden. Dieses ist zur Berechnung von Software-Metriken und ihrer Darstellung in so genannten *polymetrischen Sichten* entwickelt worden [MN05, Rot05] und wird bei der Inferenz und der Musterinstanzbewertung zur Überprüfung von Metrikbedingungen genutzt. Fuzzy-Bedingungen, welche durch je eine Zugehörigkeitsfunktion beschrieben werden, wurden mit Hilfe eines neuen Plug-ins für mathematische Funktionen realisiert. Dieses stellt mehrere parametrisierte Standard-Funktionen (lineare Funktion, e-Funktion und andere) zur Verfügung, die sowohl bei der Musterspezifikation als auch bei der Inferenz und der Bewertung von Musterfunden verwendet werden.

In den folgenden Abschnitten werden die zur Realisierung des neuen Bewertungsverfahrens durchgeführten Anpassungen genauer beschrieben.

5.2 Erweiterung des Meta-Modells für Musterspezifikationsdiagramme

Die Erweiterungen des Meta-Modells für Musterspezifikationsdiagramme und Musterregeln sind in der Abbildung 5.3 durch einen dunkleren Hintergrund beziehungsweise dickere Linien hervorgehoben. Das Klassendiagramm zeigt das Meta-Modell des Musterspezifikations-Plug-ins vereinfacht dar. Zusätzlich sind die Klassen des neuen Plug-ins für mathematische Funktionen abgebildet.

Alle Elemente einer Musterregel können mit einem Gewichtsfaktor versehen werden. Dazu erhält die Klasse PSDiagramItem das Attribut weight: double.

Optionale Fragmente werden durch die Klasse PSOptionalFragment realisiert, während die Elemente, die darin enthalten sein können, durch die Klasse PSFragmentItem beschrieben werden. Ein optionales Fragment muss mindestens ein Element enthalten.

Die Bedingungen, die zu einem Knoten spezifiziert werden können, sind durch die Unterklassen von PSNodeExpression implementiert. Attribut- und Metrikbedingungen können mit Hilfe des Attributs optional: boolean der Klasse PSNodeExpressionOptional als optional gekennzeichnet werden. Analog dazu erhält auch die Klasse PSConstraint ein solches Attribut, um optionale Constraints zu ermöglichen.

Die Zugehörigkeitsfunktion einer Fuzzy-Bedingung wird mit Hilfe der Klassen PSFuzzySetMembershipFunction, PSInterval und PSFunction modelliert. Um die Spezifikation der Zugehörigkeitsfunktion einer Fuzzy-Bedingung zu vereinfachen, werden die als Bibliothek verwendeten Funktionsimplementierungen des Plug-ins für mathematische Funktionen genutzt. Der Definitionsbereich einer Zugehörigkeitsfunktion – dieser ist immer das Intervall $(-\infty, \infty)$ – kann in mehrere Intervalle aufgeteilt werden. Für jedes Intervall wird eine mathematische Funktion ausgewählt und geeignet parametrisiert. Zum Beispiel kann beschrieben werden,

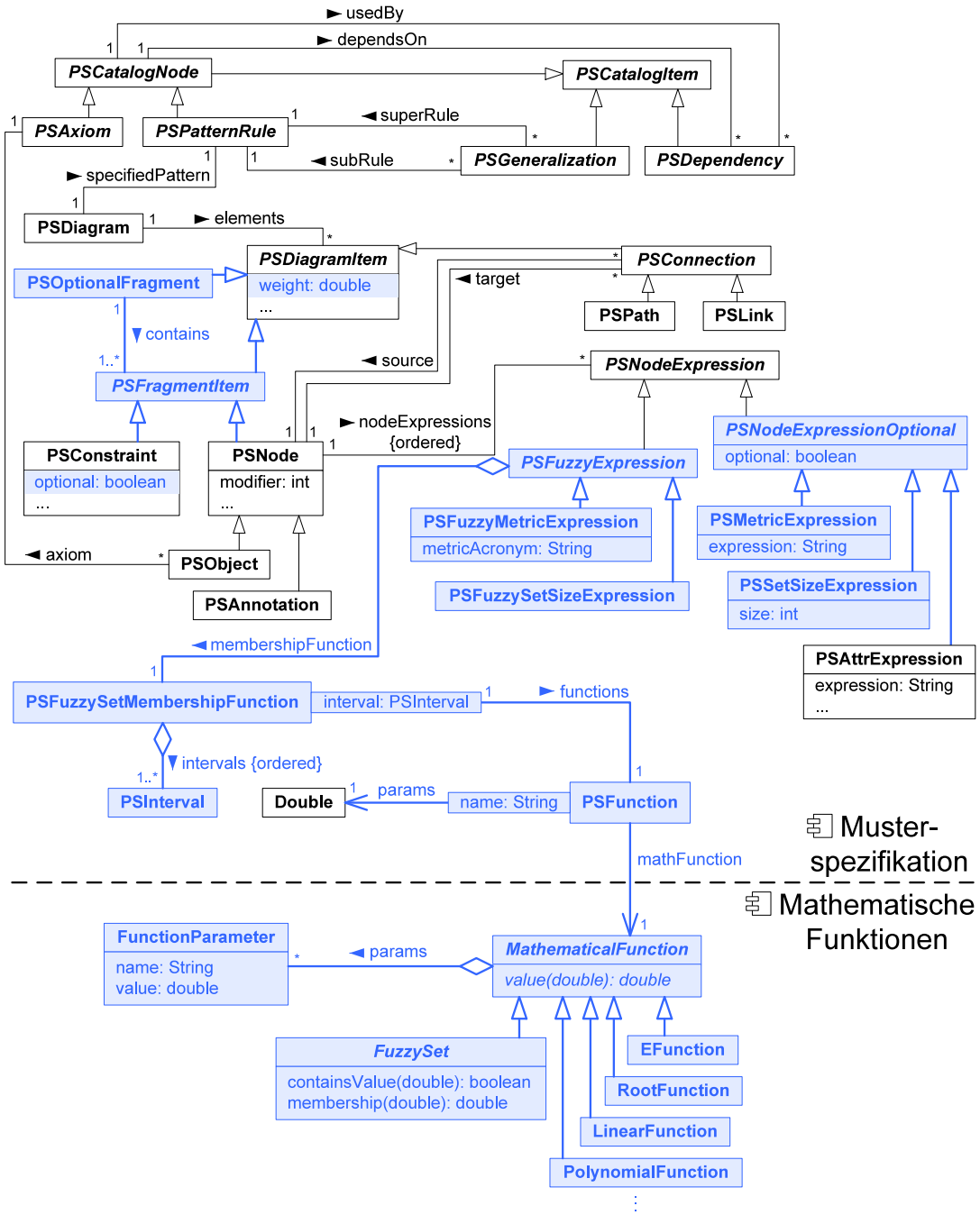


Abbildung 5.3: Meta-Modell der Musterspezifikationsdiagramme (Musterregeln)

dass eine Zugehörigkeitsfunktion auf dem Intervall $(-\infty, 10)$ durch die Funktion $f(x) = 0$ und auf dem Intervall $[10, \infty)$ durch die Funktion $g(x) = (x - 10)^2$ definiert wird.

5.3 Anpassung der Mustersuche

Zur Realisierung des neuen Bewertungsverfahrens wurde das Inferenz-Plug-in erweitert und die Generierung von Annotationsmaschinen angepasst. Diese Schritte werden im Folgenden genauer erläutert.

5.3.1 Erweiterung des Inferenz-Plug-ins

Um die Bewertung von Musterinstanzen anzupassen, wurde der bestehende Inferenzmechanismus modifiziert und erweitert. In dem UML-Klassendiagramm in Abbildung 5.4 sind die an der Implementierung des Mechanismus beteiligten Klassen und ihre Assoziationen dargestellt. Auch hier sind die Erweiterungen hervorgehoben.

Die Klasse InferenceEngine verwendet abhängig von der aktuellen Einstellung einen der implementierten Inferenzalgorithmen. Diese sind nach dem Design Pattern „Strategy“ realisiert. Die abstrakte Klasse InferenceStrategy implementiert mit der Methode run den Teil des Inferenzalgorithmus, der für alle Strategien gleich ist. Die Unterklassen bestimmen, in welcher Reihenfolge die Annotationsmaschinen zur Ausführung kommen und wie die einzelnen Musterfunde bewertet werden.

Die bisher verwendeten Inferenzstrategien benötigen für die Mustersuche nur die Musterregelabhängigkeiten in Form eines Musterregelabhängigkeitsgraphen (Klasse PatternRulesDependencyNet) sowie die generierten Annotationsimplementierungen und Annotationsmaschinenimplementierungen. Um eine als Musterinstanz erkannte Objektstruktur im abstrakten Syntaxgraphen mit der in einer Musterregel spezifizierten Objektstruktur vergleichen und die spezifizierten Bedingungen überprüfen zu können, erhält der Inferenzmechanismus zusätzlich zu den Regelabhängigkeiten auch das Fujaba-Projekt (Klasse FProject) mit den Musterspezifikationen.

Bei der iterativen Inferenzstrategie wird jede neu erzeugte Annotation durch einen Aufruf der Methode evaluateNewAnnotation der Klasse IterativeStrategy bewertet. Die Bewertung erfolgt durch Aufbauen eines Fuzzy-Petrinetzes, welches im Anschluss an die Mustersuche ausgeführt wird (siehe Abschnitt 3.1.1). Um das Bewertungsverfahren anzupassen, wurde eine neue Inferenzstrategie in der Unterklasse IterativeEvaluatingStrategy implementiert. Diese Klasse überschreibt die Methode evaluateNewAnnotation und verwendet das im Kapitel 4 beschriebene Bewertungsverfahren. Dieses ist ebenfalls nach dem „Strategy“ Design Pattern implementiert und ist dadurch beliebig austauschbar.

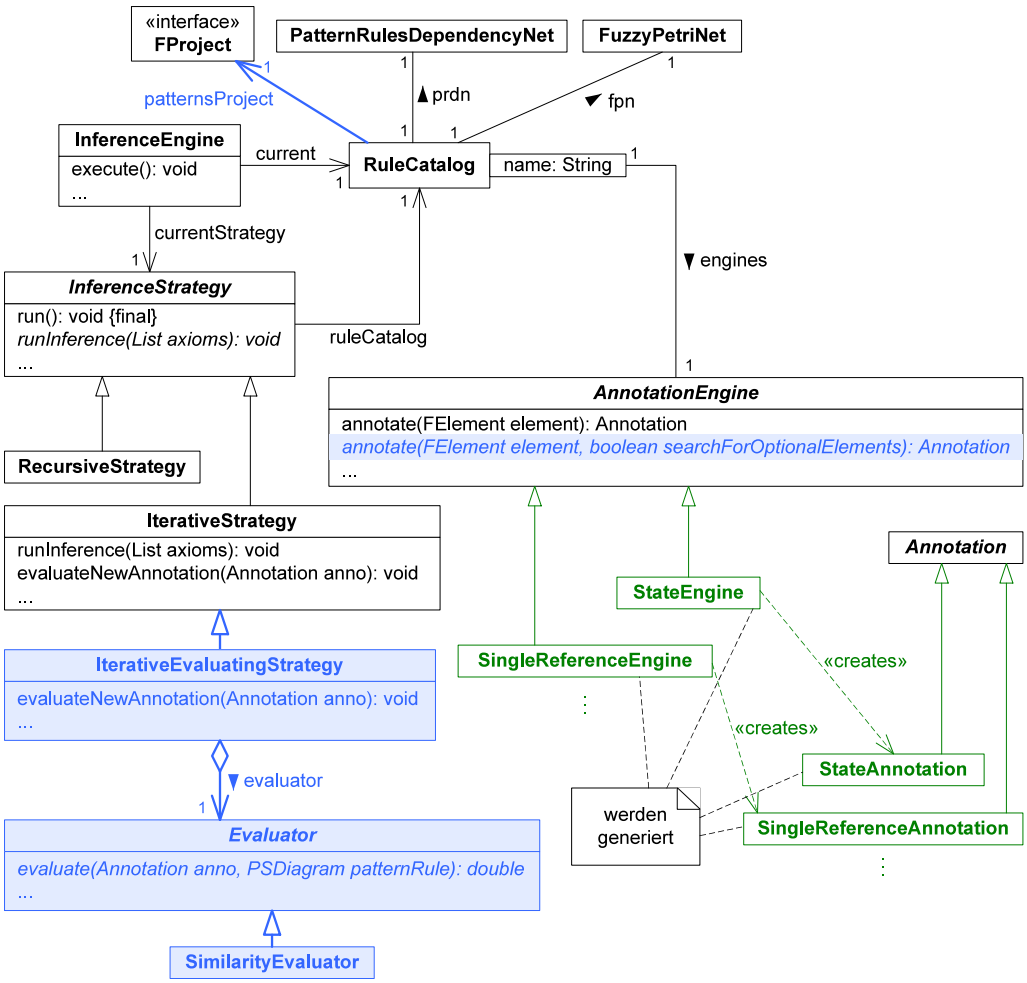


Abbildung 5.4: Erweiterung des Inferenzmechanismus

Das Bewertungsverfahren wird durch die abstrakte Klasse Evaluator repräsentiert. Ihre Unterklassen implementieren die konkreten Bewertungsstrategien. Ein Evaluator erhält die neu erzeugte Annotation und das Diagramm der zugehörigen Musterregel, bewertet die Annotation und gibt das Bewertungsergebnis als double-Wert zurück. Anschließend wird dieser Wert für die spätere Darstellung der Suchergebnisse in der Annotation gespeichert. Zur Zeit ist nur das in dieser Arbeit beschriebene Bewertungsverfahren in der Klasse SimilarityEvaluator implementiert. Andere Strategien können jedoch auf einfache Weise hinzugefügt werden.

5.3.2 Modifikation der Annotationsmaschinen

Annotationsmaschinen implementieren die Suche nach der durch eine Musterregel spezifizierten Objektstruktur. Für jede nicht abstrakte Musterregel wird je eine Unterklasse von AnnotationEngine und Annotation generiert (siehe Abb. 5.4). Die Suche wird durch Implementieren der abstrakten Methode annotate der Klasse

AnnotationEngine realisiert. Diese Methode erhält ein Objekt des abstrakten Syntaxgraphen als Argument und versucht, ausgehend von diesem Objekt als Teil des gesuchten Matchings den Rest des in der Musterregel spezifizierten Graphen zu finden. Bei Erfolg wird eine Annotation erstellt, um die Fundstelle zu markieren.

Die Suche nach Objekten zu optionalen Knoten und Fragmenten sowie die Überprüfung von optionalen Bedingungen erhöht die Laufzeit der Mustersuche. Um diese bei Bedarf reduzieren zu können, wurde die Suche nach der in einer Musterregel spezifizierten Objektstruktur in zwei Schritte unterteilt: die Suche nach dem „Pflichtteil“ und die Suche nach dem optionalen Teil der Objektstruktur. Mit Hilfe des Parameters searchForOptionalElements: boolean in der Methode annotate der Klasse AnnotationEngine (siehe Abb. 5.4) kann die Suche nach dem optionalen Teil, also die Suche nach Objekten zu optionalen Knoten und Fragmenten sowie die Überprüfung optionaler Bedingungen, ausgelassen werden. In diesem Fall wird allerdings davon ausgegangen, dass Objekte zu optionalen Knoten nicht gefunden und optionale Bedingungen nicht erfüllt sind, was in einer entsprechend niedrigen Bewertung der Musterfunde resultiert.

Die Generierung von Annotationsmaschinen erfolgt in drei Schritten. Im ersten Schritt wird ein UML-Klassendiagramm generiert, das alle zu generierenden Klassen enthält. Für jede Methode dieser Klassen wird je ein Story-Diagramm [FNTZ98] generiert, welches den Methodenrumpf beschreibt (siehe auch [Wen01, Abschnitt 6.2]). Im nächsten Schritt wird aus den Diagrammen mit Hilfe des Code-Generierungsmechanismus von Fujaba Java-Code generiert. Anschließend wird der Code kompiliert.

Für jede Rolle, die ein Objekt des abstrakten Syntaxgraphen in einer Musterregel einnehmen kann, werden je zwei Methoden in der Annotationsmaschinenklasse generiert. Die Suche nach dem „Pflichtteil“ wird in der Methode anotate_role_<Rollenname> implementiert und die Suche nach dem optionalen Teil in der Methode findOptionalElements_role_<Rollenname>. Die generierte Methode annotate probiert für ein gegebenes Objekt des abstrakten Syntaxgraphen alle Rollen nacheinander aus. In dem Fall, dass eine Annotation erstellt werden konnte, wird abhängig von dem Parameter searchForOptionalElements auch nach optionalen Knoten und Fragmenten gesucht. Schließlich wird die erstellte Annotation beziehungsweise `null` zurückgegeben.

Der Methodenrumpf der generierten anotate_role_<Rollenname>-Methode ist in der Abbildung 5.5 als Story-Diagramm vereinfacht dargestellt (eine ausführliche Beschreibung befindet sich in [Wen01, Abschnitt 6.2]). Als Beispiel ist das für die Musterregel „State“ und die Rolle der Kontextklasse (Objektknoten context in dem Diagramm in Abbildung B.1 auf Seite 121) generierte Story-Diagramm abgebildet (siehe Abb. 5.6, S. 91).

Die mit 2 und 3 markierten Aktivitäten realisieren die Suche nach allen Objektstrukturen, welche auf die Beschreibung in der Musterregel passen. Hierbei bleiben alle optionalen Elemente (Knoten, Fragmente, Bedingungen, Constraints) unberücksichtigt. Aus diesem Grund fehlen bei dem Beispiel in Abbildung 5.6 die in der Musterregel „State“ spezifizierten optionalen Attributbedingungen und der

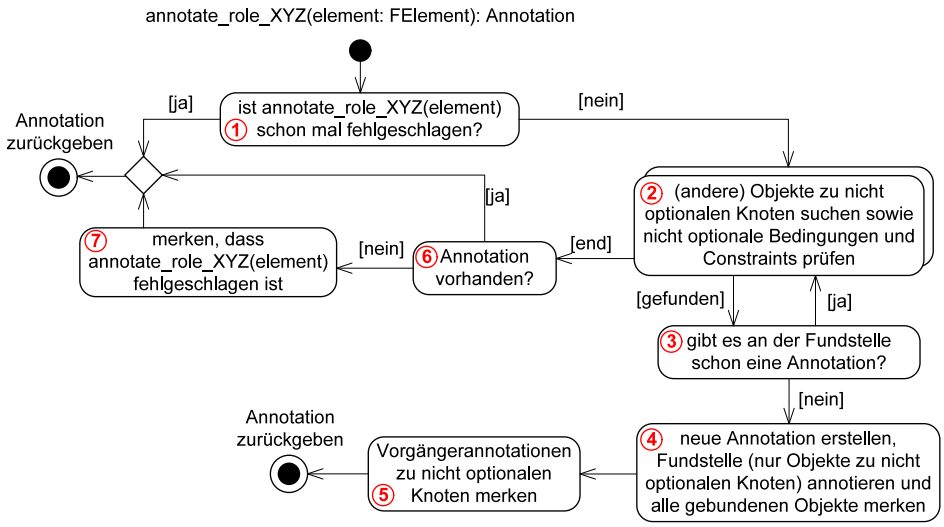


Abbildung 5.5: Suche nach nicht optionalen Elementen

optionale Knoten. In der Methode wird für jede Fundstelle überprüft, ob diese bereits annotiert ist. Falls an der Stelle keine Annotation existiert, wird in den Aktivitäten 4 und 5 eine neue Annotation erstellt und mit allen Objekten des Matchings (Zuordnung der gefundenen Objekte zu den Knoten der Musterregel) durch eine Verknüpfung verbunden.

Um die Überprüfung von Metrik- und Fuzzy-Bedingungen durch die generierten Story-Diagramme zu ermöglichen, werden spezielle Ausdrücke und Anweisungen in den Diagrammen verwendet. Metrikbedingungen werden analog zu Attributbedingungen durch Ausdrücke repräsentiert, aus denen bei der Generierung von Java-Code ein Methodenaufruf erstellt wird, der die jeweilige Bedingung mit Hilfe des Metrik-Plug-ins überprüft.

Fuzzy-Bedingungen werden mit Hilfe spezieller Klassen überprüft, welche die Zugehörigkeitsfunktion der jeweiligen Bedingung implementieren. Diese Klassen werden für jede Fuzzy-Bedingung generiert und sind Spezialisierungen der abstrakten Klasse FuzzySet (siehe Abb. 5.3, S. 86). Für jedes Intervall der spezifizierten Zugehörigkeitsfunktion wird eine der Unterklassen von MathematicalFunction instatiiert und entsprechend der Musterregelangaben parametrisiert. Die Methode value(double): double der generierten Klasse delegiert jeden Aufruf an die laut Definition der Zugehörigkeitsfunktion zuständige mathematische Funktion. Zur Überprüfung einer Fuzzy-Bedingung wird in den Story-Diagrammen der zugehörigen Annotationsmaschine eine spezielle Anweisung generiert. Diese bestimmt mit Hilfe des Metrik-Plug-ins den zu überprüfenden Metrikwert, zum Beispiel die Anzahl der Attribute einer Klasse bei der Metrik „NOA“ (Number of Attributes). Anschließend wird durch einen Aufruf der containsValue-Methode der für die Fuzzy-Bedingung generierten FuzzySet-Unterklasse bestimmt, ob die Bedingung erfüllt ist, also ob der Erfülltheitsgrad größer als Null ist.

Ist durch das Argument searchForOptionalElements der annotate-Methode ange-

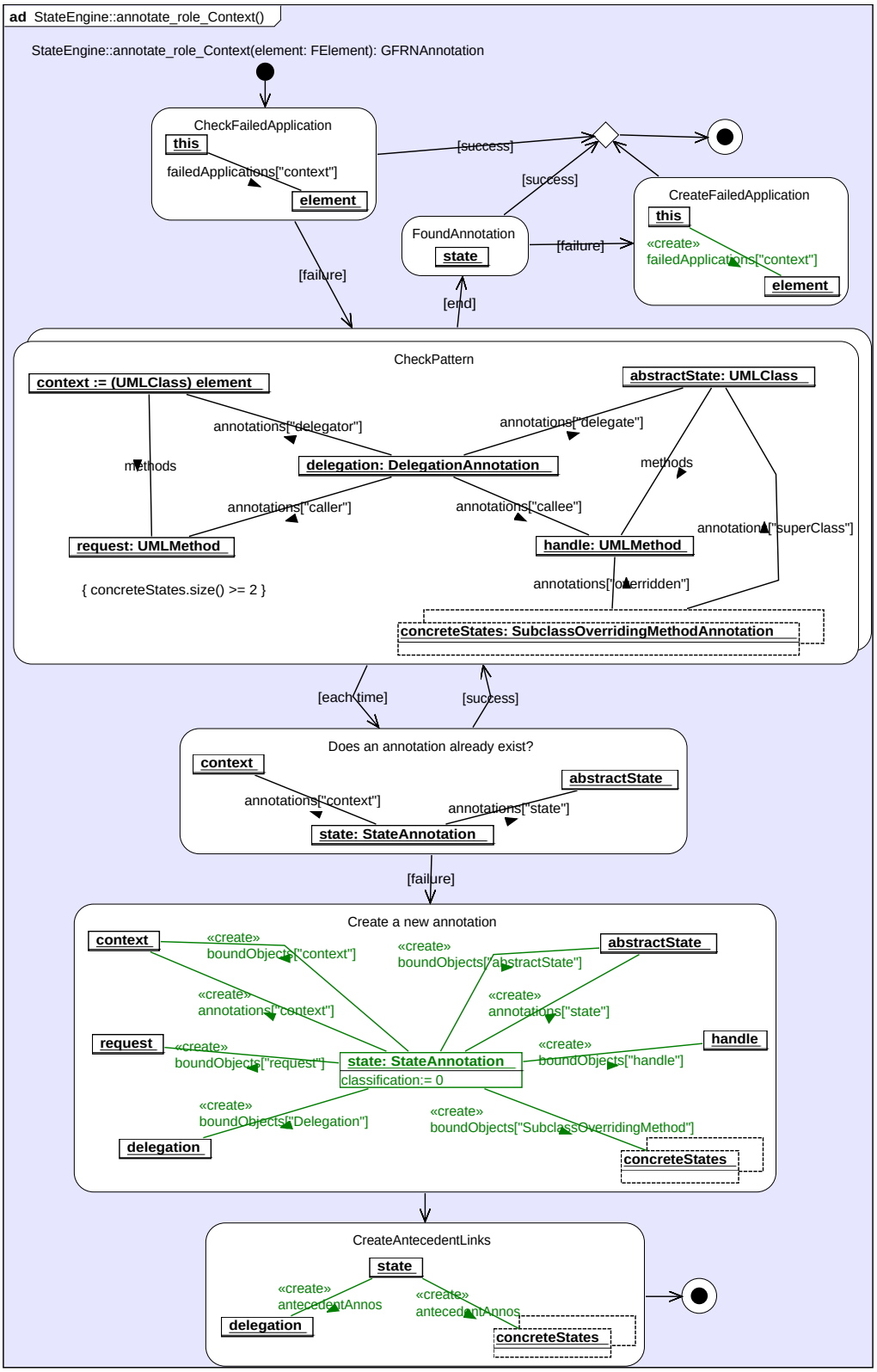


Abbildung 5.6: Story-Diagramm der Methode StateEngine.annotate_role_Context

geben worden, dass auch nach dem optionalen Teil gesucht werden soll, so wird nach dem Aufruf der anotate_role_<Rollenname>-Methode auch die zugehörige Methode findOptionalElements_role_<Rollenname> aufgerufen. Diese sucht nach geeigneten Objekten zu allen optionalen Knoten und Fragmenten und überprüft alle zugehörigen nicht optionalen Bedingungen. Jedes gefundene Objekte wird durch eine Verknüpfung mit der Annotation verbunden und das Matching dadurch erweitert.

Um die Komplexität der generierten Annotationsmaschinen zwecks Performance und Wartbarkeit gering zu halten, wird ein Großteil der optionalen Bedingungen erst bei der Bewertung einer Annotation durch eine Evaluator-Unterklasse (siehe Abb. 5.4) überprüft. Attributbedingungen und Constraints bilden hierbei eine Ausnahme. Diese können beliebige Java-Ausdrücke enthalten, deren nachträgliche Überprüfung sehr aufwendig ist. Aus diesem Grund werden Attributbedingungen und Constraints im Gegensatz zu Metrik- und Fuzzy-Bedingungen[2] bereits bei der Mustersuche in den generierten findOptionalElements_role_<Rollenname>-Methoden überprüft. Ist ein Constraint oder eine Attributbedingung erfüllt, so wird das durch einen entsprechenden Flag in der Annotation festgehalten.

Der Methodenrumpf der generierten findOptionalElements_role_<Rollenname>-Methode ist in der Abbildung 5.7 allgemein als Story-Diagramm dargestellt. Hier werden alle möglichen Fälle berücksichtigt. Mehrere wiederkehrende Schritte sind durch gleiche Farben und durch einen gestrichelten Rahmen gekennzeichnet.

Aus dem Diagramm wird deutlich, dass die Suche nach Objekten zu optionalen Knoten und Fragmenten sowie die Überprüfung von optionalen Bedingungen und Constraints voneinander weitestgehend unabhängig erfolgt. Zum Beispiel wird jedes Constraint einzeln überprüft. Ist dieses erfüllt, so wird ein entsprechender Flag in der Annotation gesetzt. Ob ein Constraint erfüllt ist, hat jedoch keinerlei Auswirkung auf die Überprüfung anderer optionaler Bedingungen oder Constraints.

Als Beispiel ist in der Abbildung 5.8 das generierte Story-Diagramm der Methode findOptionalElements_role_Context der ebenfalls generierten Klasse StateEngine dargestellt. Nachdem die Objekte des Matching, das zuvor durch die Methode annotate_role_Context erstellt wurde, gebunden wurden, werden die optionalen Attributbedingungen nacheinander überprüft. Anschließend wird nach dem optionalen Objekt gesucht. Bei Erfolg wird dieses in das Matching aufgenommen.

Durch die Art der Generierung von Annotationsmaschinen (Erstellen der beschriebenen Story-Diagramme) kann bei der Mustersuche zur Zeit nicht sichergestellt werden, dass bei mehreren möglichen Zuordnungen eines Objekts zu einem Knoten das Objekt ausgewählt wird, welches die meisten optionalen Bedingungen erfüllt beziehungsweise die höchstmögliche Bewertung der zugehörigen Annotation erzielt. Der in Fujaba implementierte Code-Generierungsmechanismus für Story-Diagramme, der hier genutzt wird, erstellt Methodenrümpfe die bei der Suche nach einem geeigneten Objekt zu einem Knoten im Story-Diagramm das erste gefundene Objekt wählen, das alle (nicht optionalen) Bedingungen erfüllt. Um

[2]Hier ist der Erfülltheitsgrad gemeint.

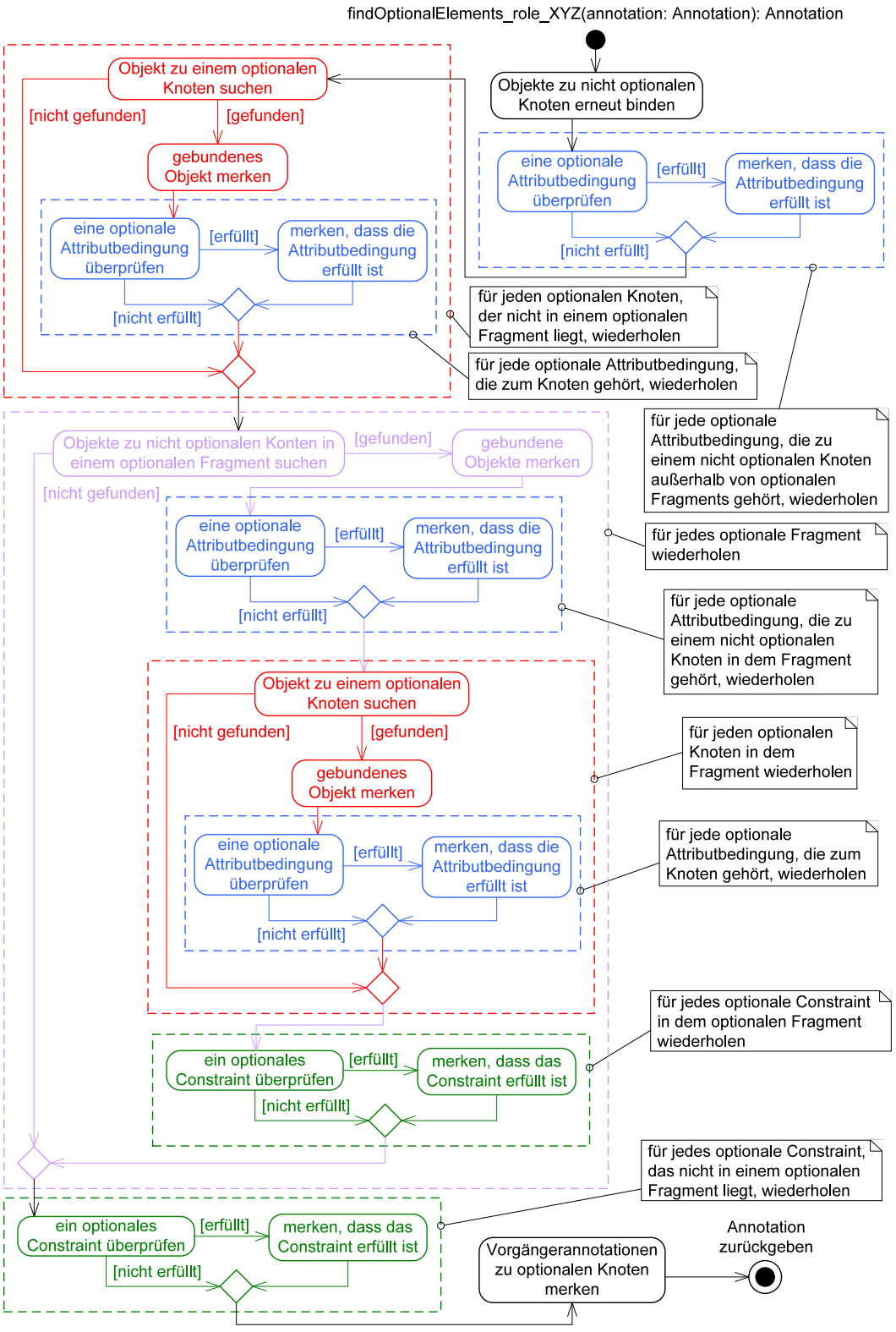


Abbildung 5.7: Suche nach optionalen Elementen

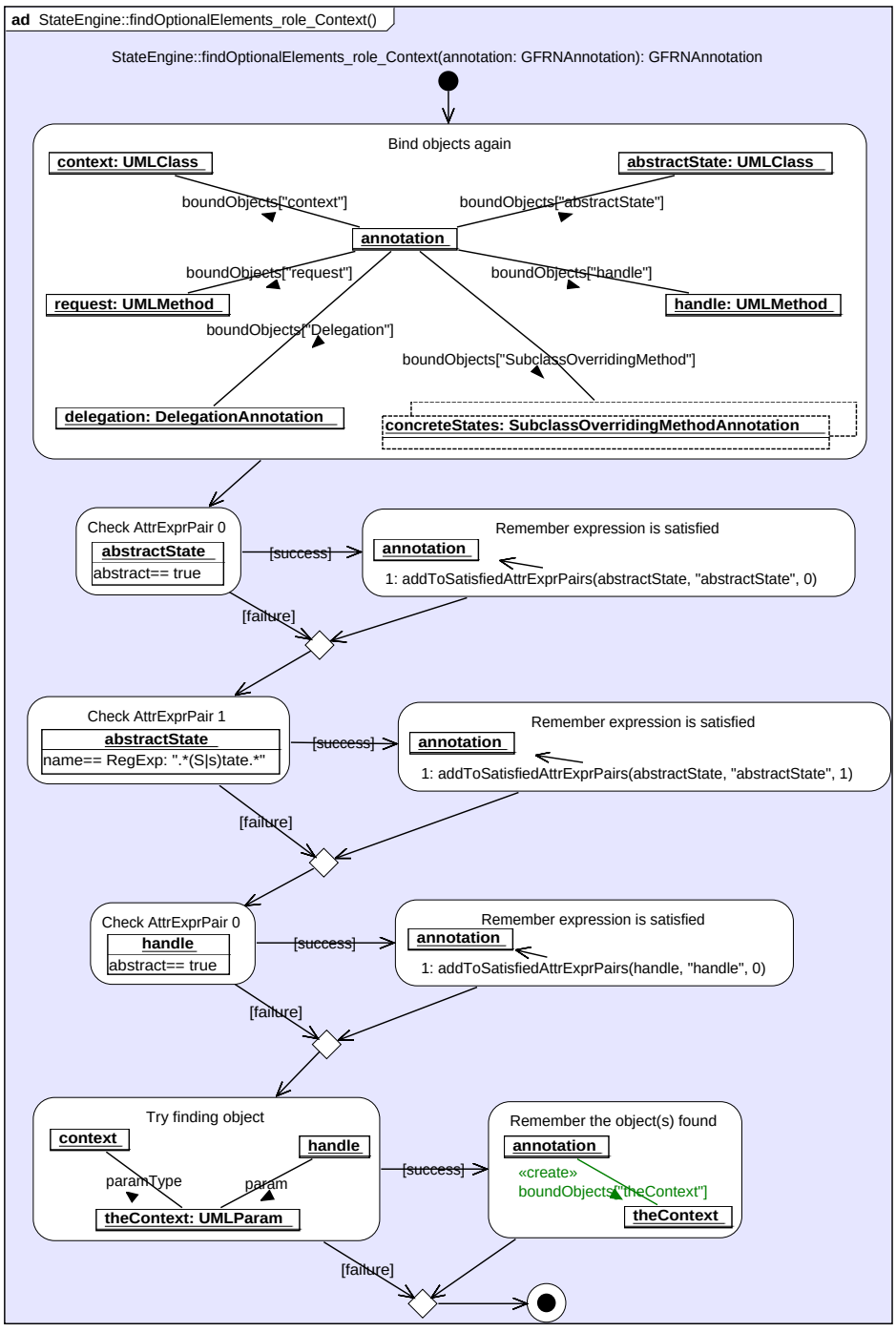


Abbildung 5.8: Story-Diagramm der Methode StateEngine.findOptionalElements_role_Context

in jedem Fall die bestmögliche Bewertung einer Annotation zu erzielen, muss in Zukunft auch der Code-Generierungsmechanismus von Fujaba angepasst werden.

5.4 Darstellung der Suchergebnisse

Die Ergebnisse einer Mustersuche werden in einer Tabelle aufgelistet. Darin sind der Name des gefundenen Musters, die Bewertung der Annotation und die annotierten Elemente aufgelistet. Zusätzlich können die Annotationen auch in Klassendiagrammen dargestellt werden.

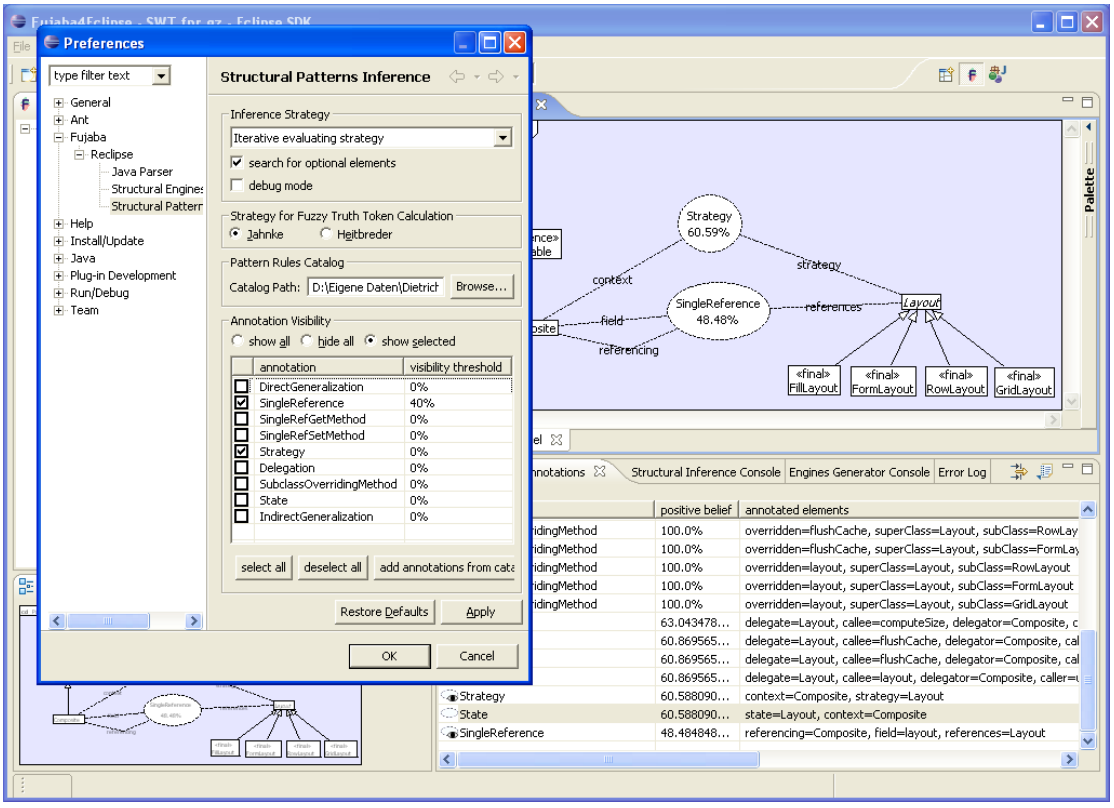


Abbildung 5.9: Einschränkung der sichtbaren Annotationen

Die Ellipsen, welche die Annotationen in einem Klassendiagramm repräsentieren, enthalten neben dem Namen der zugehörigen Musterregel auch die Annotationsbewertung. Sie sind durch gestrichelte Linien mit den Klassen verbunden, die annotiert sind oder annotierte Elemente wie Methoden und Attribute enthalten (siehe Abb. 5.9).

Um die Übersicht zu erhöhen, kann ein Teil der Annotationen in Klassendiagrammen und der Annotationstabelle unabhängig voneinander ausgeblendet werden. Die darzustellenden Annotationen können durch Selektieren von Annotationstypen und durch Güteschranken angegeben werden (siehe Abb. 5.9). Es werden nur die Annotationen angezeigt, deren Typ selektiert und deren Bewertung höher

ist als die angegebene Güteschranke (visibility threshold). In der Annotationstabelle können zusätzlich zu den in Klassendiagrammen dargestellten Annotationen auch ausgeblendete aufgelistet werden. Ein Augensymbol markiert die im Diagramm sichtbaren Annotationen.

Neben der Bewertung von Musterfunden ist die Auswahl der darzustellenden Annotationen ebenfalls sehr hilfreich bei der Identifikation der relevanten Suchergebnisse. Zum Beispiel kann so die Sicht auf besonders hoch bewertete Annotationen eingeschränkt werden. Um einen Überblick über erkannte Design Patterns zu erhalten, können Annotationen zu Hilfsmustern ausgeblendet werden.

6 Praktische Erfahrungen

Das entwickelte Bewertungsverfahren wurde an mehreren Programmbeispielen und verschiedenen Software-Mustern getestet. Daraus wurden die Muster „State“, „Strategy“ und „Large Class“ herausgegriffen, um daran die Bewertung von Musterfunden zu demonstrieren. Im Folgenden werden die unterschiedlichen Bewertungsergebnisse dreier Implementierungsvarianten des Musters „State“ erläutert und die Bewertung einer Ausprägung des Musters „Strategy“ in der Entwicklungsumgebung Eclipse [Ecl06] beschrieben. Anschließend werden die Ergebnisse einer Suche nach großen Klassen mit Hilfe der Musterregel „Large Class“ vorgeführt und die Vorteile von Fuzzy-Bedingungen aufgezeigt.

6.1 Bewertung verschiedener Implementierungsvarianten im Vergleich

In den Abbildungen 6.1 bis 6.3 sind drei mögliche Implementierungen der Bedienoberfläche eines Medien-Players durch UML-Klassendiagramme skizziert. In allen drei Fällen wurde das Design Pattern „State“ [GHJV95, S. 305] angewandt (vgl. Struktur des Musters, Abb. 4.1, S. 45).

Die Benutzeroberfläche, repräsentiert durch die Klasse PlayerUI, hält eine Referenz auf den aktuellen Zustand. Dieser wird durch die Klasse PlayerState beschrieben, welche die Methode handleButtonPressedEvent definiert. Die konkreten Zustände werden durch Unterklassen von PlayerState implementiert und überschreiben die Methode, um das Verhalten in dem Zustand zu definieren. Ein PlayerUI-Objekt delegiert jeden Aufruf der Methode buttonPressed an sein Zustandsobjekt.

Die drei Implementierungsvarianten wurden mit Hilfe der im Anhang B dargestellten Musterregeln als je eine Musterinstanz erkannt und anschließend unter Verwendung des im Kapitel 4 vorgestellten Verfahrens bewertet. Die Ergebnisse der Bewertungen sind in der Tabelle 6.1 auf Seite 100 aufgelistet.

Da das Design Pattern „State“ strukturell dem Design Pattern „Strategy“ gleicht (vgl. Abb. 4.1, S. 45 und Abb. 6.5), wurden die „State“-Musterinstanzen auch als „Strategy“-Musterinstanzen erkannt. Die in der Tabelle ebenfalls aufgeführten Bewertungen der Strategy-Annotationen unterscheiden sich von denen der State-Annotationen nur aufgrund des Begriffs „State“ in dem Namen der Klasse PlayerState (vgl. Musterregeln in Abb. B.1 und Abb. B.2, S. 121).

Neben den Bewertungen der gefundenen Musterinstanzen ist auch der Anteil eines Hilfsmusters an dem gesuchten Entwurfsmuster aufgeführt. Dieser setzt das

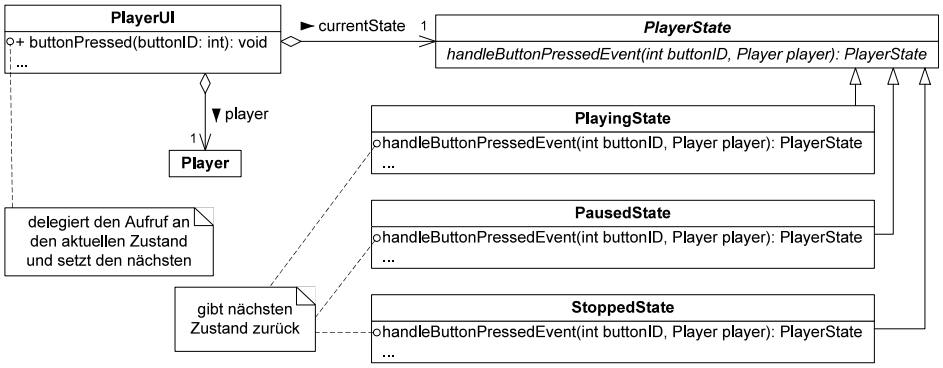


Abbildung 6.1: Implementierung des „State“-Musters, Variante 1

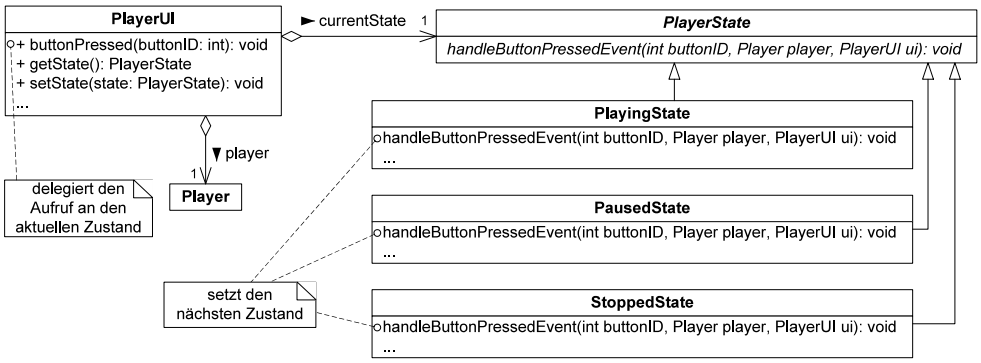


Abbildung 6.2: Implementierung des „State“-Musters, Variante 2

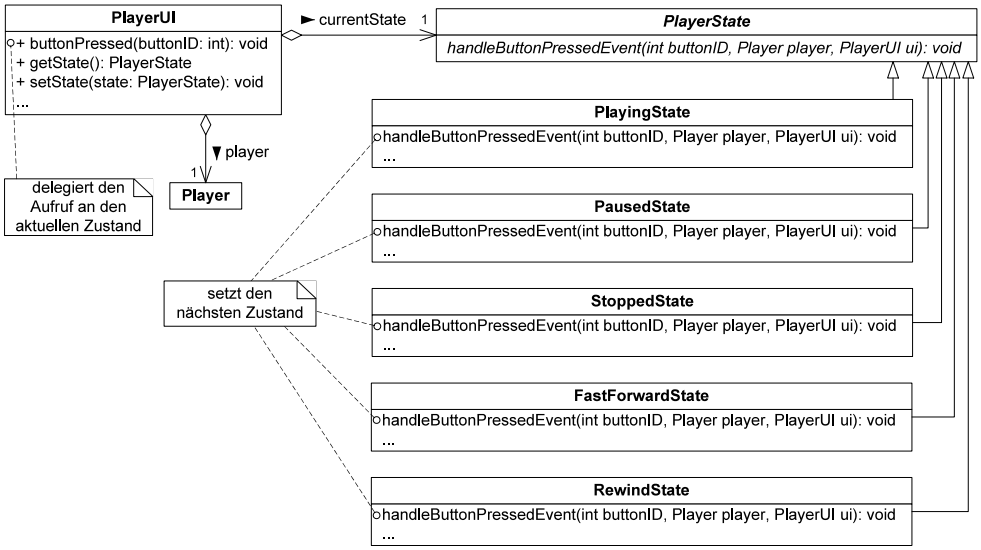


Abbildung 6.3: Implementierung des „State“-Musters, Variante 3

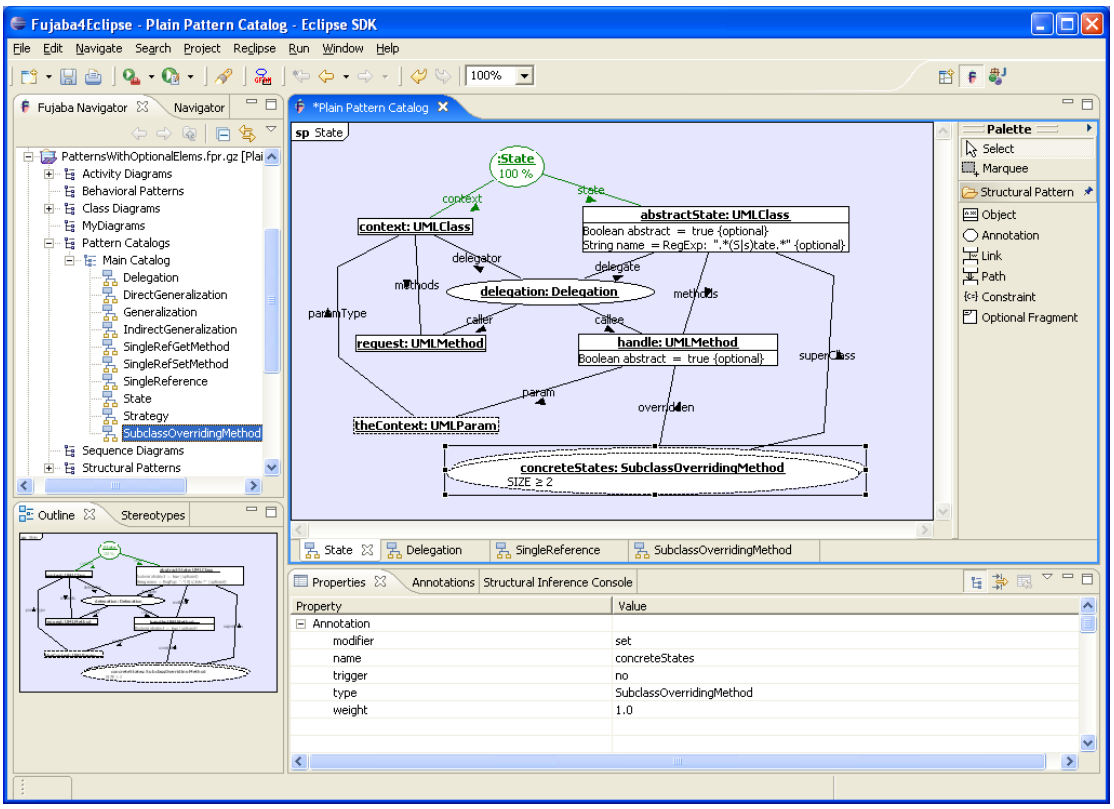


Abbildung 6.4: Spezifikation des Musters „State“

Gewicht einer Hilfsmusterregel ins Verhältnis zum Gewicht der Regel „State“ beziehungsweise „Strategy“ und gibt damit einen Hinweis auf den Einfluss der Hilfsmusterregel auf die Bewertung einer State- beziehungsweise Strategy-Annotation.

Die Abbildung 6.1 stellt eine Implementierung des „State“-Musters dar, bei der die Referenz auf den aktuellen Zustand durch ein Attribut der Klasse PlayerUI realisiert ist. Methoden für den Schreib- und Lesezugriff auf das Attribut fehlen. Ein konkreter Zustand gibt nach Behandlung des Ereignisses (gedrückter Knopf) den Folgezustand zurück. Der Zustandswechsel erfolgt, indem das PlayerUI-Objekt die Referenz auf den neuen Zustand setzt.

Diese „State“-Implementierung wurde mit 39.59% bewertet. Der Grund für

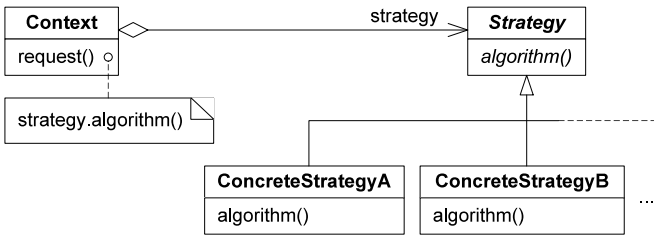


Abbildung 6.5: Struktur des Design Patterns „Strategy“

Tabelle 6.1: Beispiele für Bewertungen von „State“-Musterinstanzen

Musterinstanz	Anteil an der Entwurfsmusterinstanz	Bewertung		
		Fall 1	Fall 2	Fall 3
State	100%	39.59%	86.47%	87.82%
Strategy	100%	38.03%	84.91%	86.26%
Delegation	71.88%	34.78%	97.83%	97.83%
SingleReference	51.56%	12.12%	100%	100%
SingleRefGetMethod	17.19%	-	100%	100%
SingleRefSetMethod	21.88%	-	100%	100%

die relativ niedrige Bewertung sind insbesondere die fehlenden Zugriffsmethoden für das Attribut, das eine Referenz auf den aktuellen Zustand hält. Diese sind durch die Hilfsmuster „SingleRefGetMethod“ und „SingleRefSetMethod“ spezifiziert und kommen in der Musterregel „SingleReference“ (siehe Abb. B.8, S. 123) in Form von Annotationsknoten vor. Die in der Regel „SingleReference“ und ihren Hilfsmusterregeln spezifizierten Knoten und Bedingungen machen mehr als die Hälfte der zur Beschreibung des „State“-Musters verwendeten Knoten und Bedingungen aus. Dadurch fällt die niedrige Bewertung der SingleReference-Annotation bei der Bewertung der State-Annotation besonders stark ins Gewicht.

In der Abbildung 6.6 ist das Ergebnis der Mustersuche für dieses Beispiel dargestellt. Die erstellten Annotationen sind in dem Klassendiagramm als Ellipsen dargestellt, welche neben dem Namen der Musterregel auch die Annotationsbewertung enthalten. Gestrichelte Linien verbinden die annotierten Elemente (zum Beispiel Klassen, Methoden und Attribute) mit der Annotation. Um die Übersicht zu erhöhen, wurde ein Teil der Annotationen in dem Klassendiagramm ausgeblendet (zum Beispiel die SubclassOverridingMethod-Annotationen). Zusätzlich zu der Darstellung der Annotationen im Diagramm sind diese auch in einer Tabelle aufgelistet (unten im Bild).

Die in den Abbildungen 6.2 und 6.3 (S. 98) skizzierten Anwendungen des Design Patterns „State“ unterscheiden sich nur durch die Anzahl der nicht abstrakten Zustandsklassen. Im Gegensatz zu dem vorherigen Fall wird hier der Zustandswechsel nicht durch das PlayerUI-Objekt, sondern durch ein Zustandsobjekt vollzogen. Dazu wird die Methode setState auf dem als Parameter übergebenen PlayerUI-Objekt aufgerufen.

Diese beiden Implementierungsvarianten wurden mit 86.47% und 87.82% bewertet. Zu dem Zustandsattribut existieren in diesen Fällen beide Zugriffsmethoden: je eine für den Lese- und Schreibzugriff. Außerdem sind nahezu alle optionalen Bedingungen erfüllt. Nur ein optionales Constraint der Musterregel „Delegation“ ist nicht erfüllt. Dieses fordert für die Methoden buttonPressed und handleButtonPressedEvent gleiche Namen. Die Implementierungsvariante aus Abbildung 6.3 ist höher bewertet worden als die aus Abbildung 6.2. Der Grund dafür ist die höhere Anzahl an Zustandsklassen.

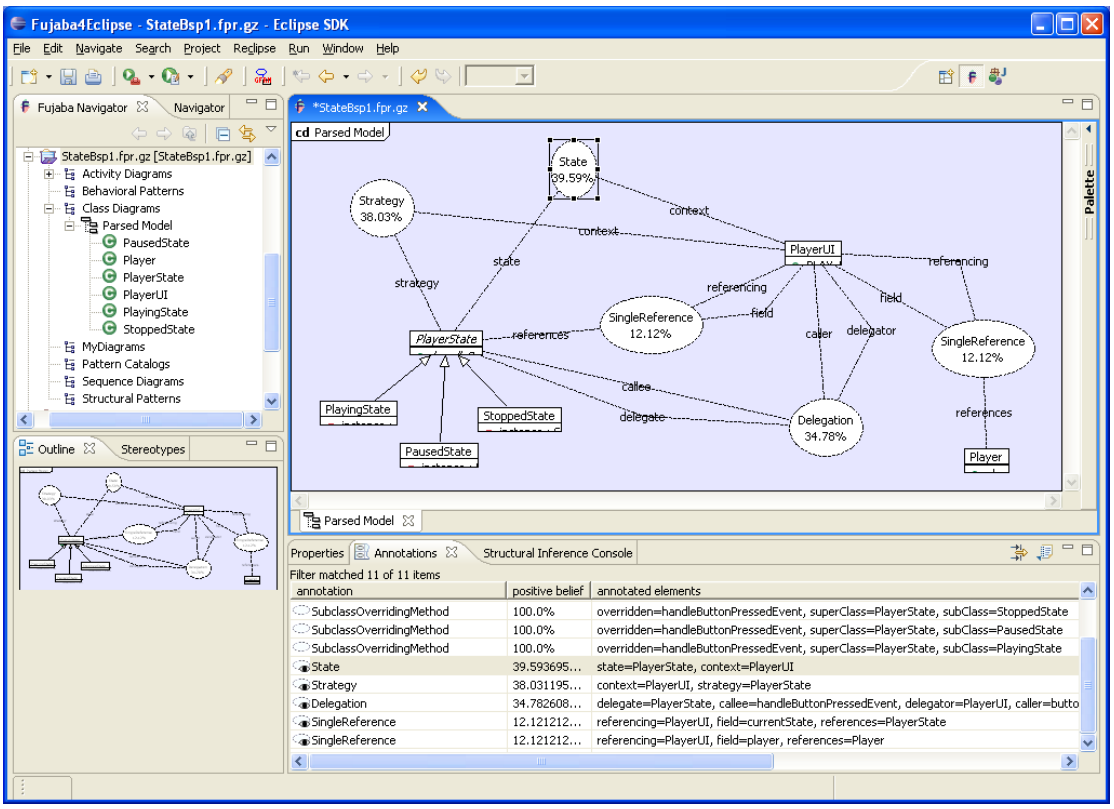


Abbildung 6.6: Bewertungsergebnisse des ersten Beispiels

Zum Vergleich wurden die drei Instanzen des Musters „State“ auch nach dem im Abschnitt 3.1 beschriebenen Verfahren bewertet. Bei diesem Ansatz entspricht die Bewertung einer Annotation ihrem Genauigkeitswert. Dieser wird durch Bilden des Minimums aus dem vom Reverse Engineer geschätzten Vertrauenswert der zugehörigen Musterregel und den Genauigkeitswerten der Vorgängerannotationen rekursiv berechnet.

In dem bisher verwendeten Musterkatalog (ohne optionale Elemente) ist die Musterregel „State“ mit dem Vertrauenswert 80% spezifiziert worden. Alle zur Beschreibung des Musters verwendeten Hilfsmusterregeln haben keinen niedrigeren Vertrauenswert. Der einzig mögliche Genauigkeitswert für eine Instanz des „State“-Musters ist somit ebenfalls 80%, was auch die Bewertung der drei Musterausprägungen aus den Abbildungen 6.1 bis 6.3 ergab.

Die Beispiele zeigen, dass mit dem neuen Verfahren eine wesentlich präzisere Bewertung als bei dem bisher in Fujaba verwendeten Ansatz (siehe Abschnitt 3.1) möglich ist. Bereits wenige optionale Bedingungen reichen aus, um eine aussagekräftige Bewertung zu ermöglichen. Die Unterschiede der verschiedenen Musterfunde werden mit Hilfe optionaler Bedingungen durch die Bewertung hervorgehoben.

6.2 Bewertung einer Strategy-Musterausprägung in Eclipse

Die Beispiele im vorhergehenden Abschnitt sind konstruiert. Um das Bewertungsverfahren auch an einem realistischen Anwendungsbeispiel vorzuführen, wurde ein Teil des Eclipse Frameworks [Ecl06] nach Mustervorkommen durchsucht.

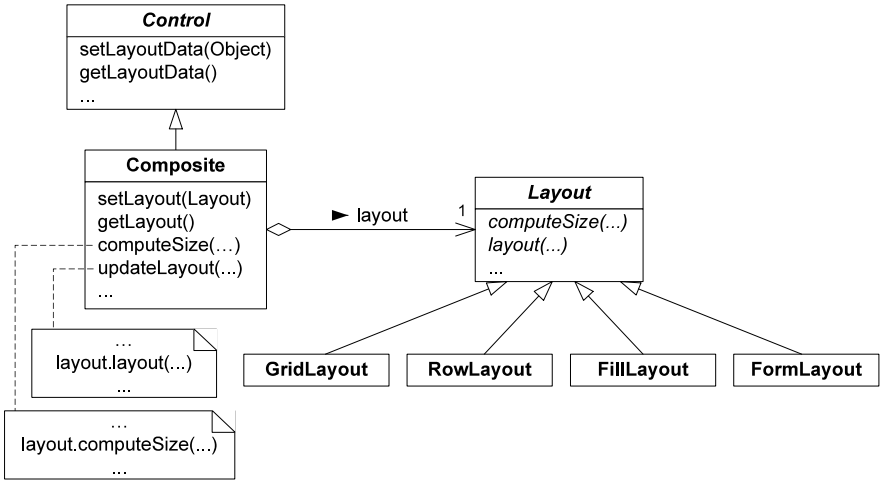


Abbildung 6.7: Struktur der Strategy-Musterausprägung im SWT

In der Dokumentation ist nachzulesen, dass in der SWT-Bibliothek (Standard Widget Toolkit) von Eclipse das Muster „Strategy“ angewendet wurde [GB03, S. 330-333]. Eine Klasse Composite verwendet verschiedene Layout-Algorithmen, um die darin enthaltenen Elemente zu platzieren und ihre Größe zu bestimmen. Die Algorithmen sind als Unterklassen von Layout implementiert. Ein Composite-Objekt delegiert die Berechnungen an sein Layout-Objekt durch Aufrufen der Methoden computeSize und layout. In der Abbildung 6.7 wird die Implementierung durch ein UML-Klassendiagramm skizziert.

Mit Hilfe der Musterregeln, die im Anhang B aufgeführt sind, wurde eine Mustersuche auf Teilen der SWT-Bibliothek durchgeführt[1]. Die Suchergebnisse sind der Abbildung 6.8 zu entnehmen. Um die Übersicht zu erhöhen, wurden in dem Klassendiagramm und in der Annotationstabelle einige der Annotationen ausgeblendet. Im Klassendiagramm wurden ausschließlich Strategy-Annotationen dargestellt, während in der Annotationstabelle nur die Annotationen aufgelistet wurden, die eine der Klassen aus Abbildung 6.7 annotieren.

Die beschriebene Musterausprägung wurde erkannt und mit 60.59% bewertet. Die relativ geringe Bewertung ist dadurch entstanden, dass die Zugriffsmethode Composite.setLayout aufgrund von Abweichungen von der Methodenrumpfbeschreibung in der Musterregel aus Abbildung B.10 nicht als Zugriffsmethode er-

[1] Es wurden die Pakete org.eclipse.swt.widgets und org.eclipse.swt.layout untersucht. Hier befinden sich die am Muster beteiligten Klassen.

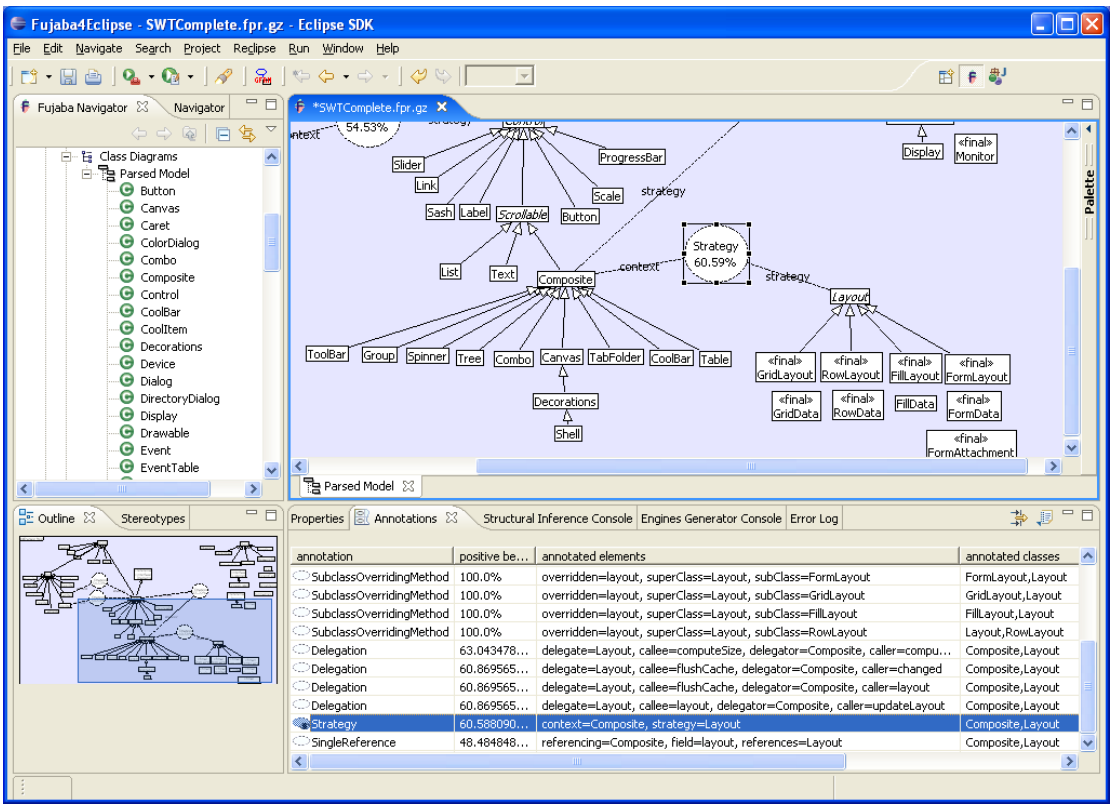


Abbildung 6.8: Bewertung der Strategy-Musterausprägung im SWT

kannt wurde. Aus dem gleichen Grund sind auch die SingleReference-Annotation und die davon abhängigen Delegation-Annotationen relativ gering bewertet worden.

Die Auflistung der gefundenen Annotationen zeigt, dass unter anderem die beiden Delegationen aus Abbildung 6.7 erkannt wurden. Die Delegation an die Methode Layout.computeSize wurde mit 63.04% bewertet, die an die Methode Layout.layout mit 60.87%. Die höhere Bewertung der ersten von beiden genannten Delegationen ergibt sich aufgrund der Namensgleichheit der delegierenden Methode computeSize in der Klasse Composite und der aufgerufenen Methode in der Klasse Layout. Diese Bedingung ist in der Musterregel „Delegation“ in Abbildung B.7 durch ein optionales Constraint angegeben worden.

Neben der in der Literatur beschriebenen Ausprägung des Musters „Strategy“ sind auch drei weitere Stellen im SWT als Instanz dieses Musters markiert worden. Die zugehörigen Annotationen sind samt ihrer Bewertungsergebnisse in der Abbildung 6.9 dargestellt. Eine manuelle Untersuchung des Quellcodes hat ergeben, dass es sich hierbei um False Positives handelt, die aufgrund der Vererbungshierarchien der Klassen Control und Composite und der darin implementierten beziehungsweise überschriebenen Methoden der Struktur des Design Patterns „Strategy“ ähneln. Wegen der geringeren Übereinstimmung mit der Spezifikation

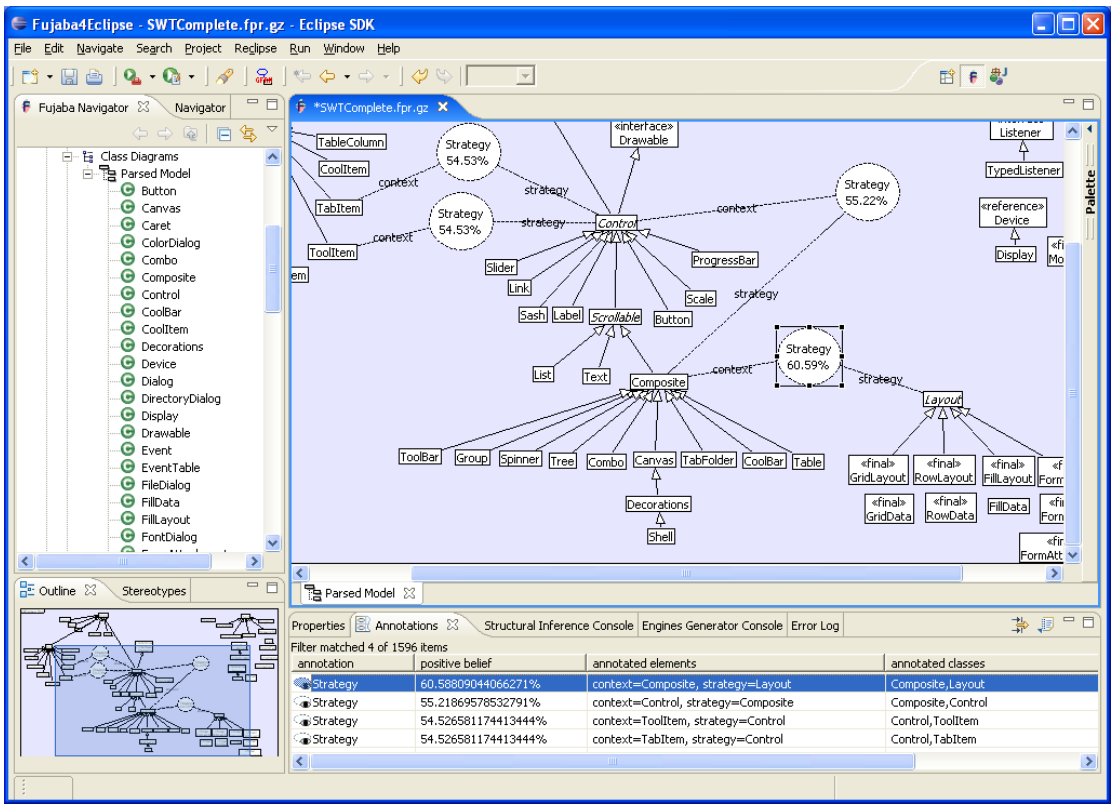


Abbildung 6.9: Bewertung von Strategy-Musterfunden im SWT

des Musters „Strategy“ sind diese jedoch geringer bewertet als die Annotation, welche die tatsächlich vorhandene Musterausprägung repräsentiert.

Ein Grund für die geringere Bewertung eines der False Positives ist, dass die als Strategie markierte Klasse Composite nicht abstrakt ist. Diese Eigenschaft wird in der Musterregel „Strategy“ in Abbildung B.2 (S. 121) durch eine optionale Attributbedingung gefordert. Die Bedingung ist als optional gekennzeichnet worden, um auch von der Musterspezifikation abweichende Musterausprägungen finden zu können. Bei der Implementierung könnte zum Beispiel vergessen worden sein, eine Klasse oder Methode als abstrakt zu markieren. Dennoch ist diese Eigenschaft ein wichtiger Bestandteil des Musters. Ist diese bei einem Fund nicht vorhanden, so ist es möglich, aber unwahrscheinlich, dass es sich um eine Ausprägung des gesuchten Musters handelt, was mit Hilfe eines Gewichtsfaktors ausgedrückt werden kann.

Bisher sind optionale Bedingungen bei den verwendeten Regeln (siehe Anhang B) mit keinem Gewichtsfaktor versehen, wodurch der Standardwert 1 verwendet wird. Um die Wichtigkeit der Forderung nach einer *abstrakten* Strategieoberklasse mit einer *abstrakten*, darin definierten Methode für den zu implementierenden Algorithmus auszudrücken, wurden die Gewichte der beiden Bedingungen durch Angeben des Gewichtsfaktors $w = 10$ erhöht. Anschließend wurde die Bewertung mit den neuen Gewichten wiederholt. Die dabei entstandenen Bewer-

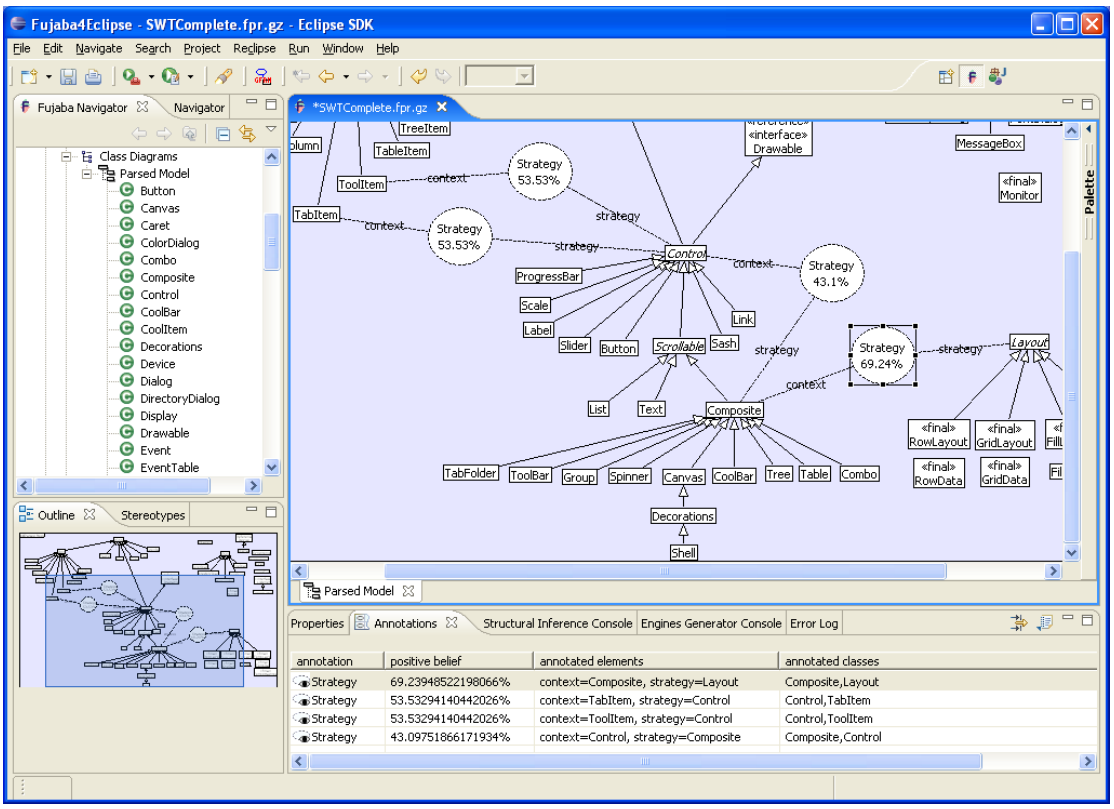


Abbildung 6.10: Alternative Bewertung von Strategy-Musterfunden im SWT

tungsergebnisse sind der Abbildung 6.10 zu entnehmen.

Aufgrund der erhöhten Gewichte fällt die Bewertung des False Positives, bei dem die nicht abstrakte Klasse Composite als Strategie markiert wurde, mit dem Wert 43.1% deutlich geringer aus, als bei dem korrekten Fund, der in diesem Fall mit 69.24% bewertet wurde. Durch Angeben von Gewichtsfaktoren für optionale Bedingungen lassen sich also Musterfunde, welche die Bedingungen erfüllen, in den Bewertungsergebnissen stärker hervorgeheben. Werden Musterfunde zu verschiedenen Musterregeln betrachtet, so kann durch eine andere Gewichtung auch die Ordnung der Annotationen geändert werden, da sich der errechnete Anteil an erfüllten Bedingungen ändert.

In diesem Abschnitt wurde gezeigt, dass Musterinstanzen nach dem neuen Bewertungsverfahren präzise bewertet werden. Eine Sortierung der Musterfunde hebt die Funde hervor, die mit der Musterspezifikation am meisten übereinstimmen, und vereinfacht damit die Analyse der Suchergebnisse. Durch Angeben von Gewichtsfaktoren kann der Einfluss bestimmter optionaler Bedingungen auf die Bewertungsergebnisse gezielt erhöht und damit das Vorhandensein bestimmter Eigenschaften betont werden.

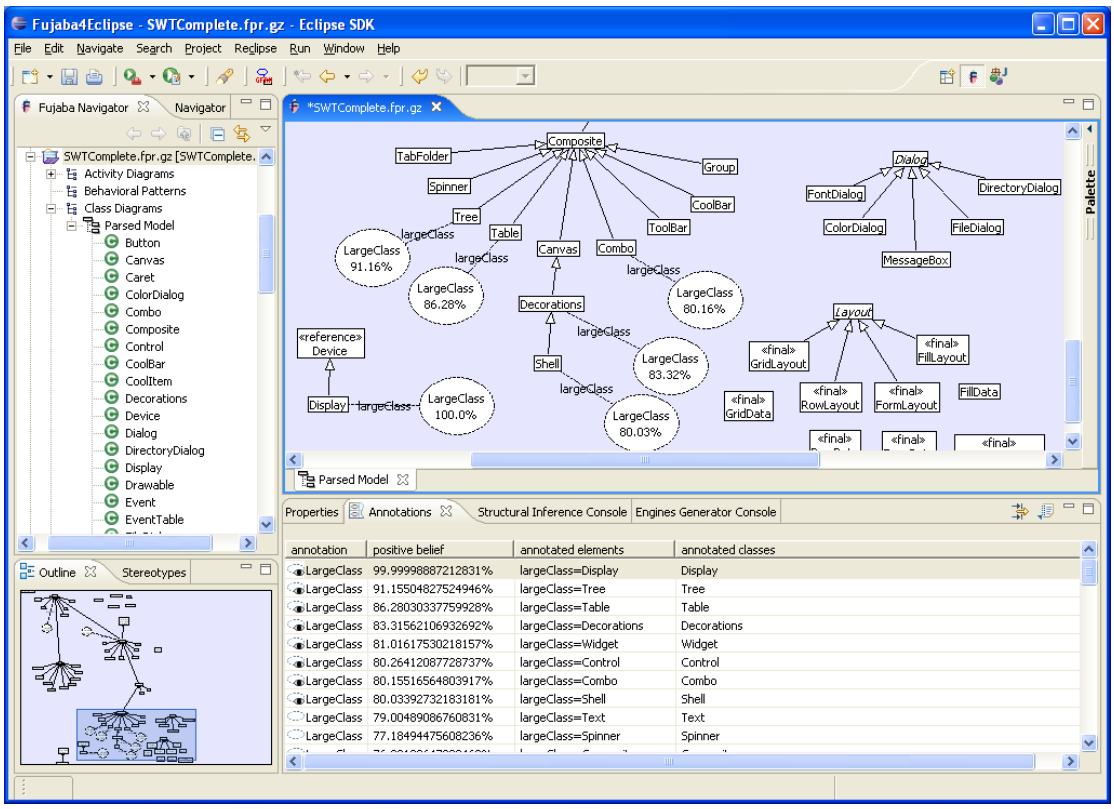


Abbildung 6.11: Musterinstanzen zum Bad Smell „Large Class“

6.3 Suche nach großen Klassen mit Hilfe von Fuzzy-Bedingungen

Ein Anwendungsbeispiel für Fuzzy-Bedingungen ist die im Abschnitt 4.4.3 vorgestellte Musterregel zum Bad Smell „Large Class“ (siehe Abb. 4.7, S. 55). Mit Hilfe dieser Musterregel wurde der Quellcode der Bibliothek SWT (Standard Widget Toolkit im Eclipse Framework [Ecl06]) nach besonders großen Klassen durchsucht.

Bei der Spezifikation des Musters wurde für jede Fuzzy-Bedingung eine Zugehörigkeitsfunktion angegeben, indem eine von mehreren vorimplementierten mathematischen Funktionen ausgewählt und geeignet parametrisiert[2] wurde (für Details siehe Abschnitt 5.2). In allen drei Fällen handelte es sich dabei um die folgende streng monoton steigende Funktion mit dem Grenzwert 1:

$$\mu(x) = \frac{1+\varepsilon}{1+e^{(-x+\Delta x)\cdot c}} + \Delta y \tag{6.1}$$

Die Graphen der verwendeten, parametrisierten Funktion sind zusammen mit der Musterregel auf Seite 55 abgebildet.

[2]Es können auch die voreingestellten Werte der implementierten Funktionen verwendet werden.

Tabelle 6.2: Bewertungen von „Large Class“-Musterinstanzen

Klasse	Bewertung	NOA/$\mu_{\text{NOA}}(x)$	NOM/$\mu_{\text{NOM}}(x)$	WLOC/$\mu_{\text{WLOC}}(x)$
Display	100%	97 / 1.0	140 / 1.0	1796 / 1.0
Tree	91.16%	26 / 0.557798	100 / 0.999954	2428 / 1.0
Table	86.28%	22 / 0.31403	107 / 0.999986	1965 / 1.0
Decorations	83.32%	19 / 0.17425	80 / 0.998716	979 / 0.996408
Widget	81.02%	16 / 0.086303	87 / 0.9996	843 / 0.982453
Control	80.26%	10 / 0.013207	251 / 1.0	1726 / 0.999999
Combo	80.16%	11 / 0.019605	79 / 0.998484	948 / 0.994835
Shell	80.03%	14 / 0.050688	81 / 0.998913	816 / 0.976048

Die Werte sind auf sechs Nachkommastellen gerundet. NOA = Number of Attributes, NOM = Number of Methods, WLOC = Lines of Code. Siehe auch Abbildung 4.7 auf Seite 55.

Durch eine Zugehörigkeitsfunktion lässt sich der Begriff „viel“ treffender und präziser beschreiben als mit Schranken (vgl. Musterregeln aus den Abbildungen 4.4 und 4.7, Seiten 53 und 55). Für jeden möglichen Metrikwert, zum Beispiel 10 oder 97 Attribute in einer Klasse, wird durch diese Funktion der Erfülltheitsgrad der zugehörigen Fuzzy-Bedingung angegeben und dazu verwendet, den Grad der Übereinstimmung des Fundes mit dem gesuchten Muster zu bestimmen.

Die Untersuchung des SWT-Quellcodes mit Hilfe der Musterregel „Large Class“ aus Abbildung 4.7 (S. 55) hat mehrere große Klassen aufgedeckt. Die mit über 80% bewerteten Klassen sind in der Tabelle 6.2 aufgelistet und in der Abbildung 6.11 dargestellt. Zusätzlich zu der Bewertung der Musterfunde sind in der Tabelle die zugehörigen Metrikwerte und die Erfülltheitsgrade der jeweiligen Fuzzy-Bedingungen (ausgedrückt durch die Zugehörigkeitsfunktion μ) aufgeführt.

Die verschiedenen Bewertungsergebnisse zeigen die hohe Präzision, mit der die Musterfunde mit Hilfe der Fuzzy-Bedingungen bewertet werden. Die Klassen mit der höchsten Übereinstimmung mit der Musterspezifikation – also die Klassen mit der höchsten gewichteten Summe der Erfülltheitsgrade – werden durch eine hohe Bewertung hervorgehoben. Zum Beispiel wurde die Klasse Display aufgrund der 97 darin enthaltenen Attribute, 140 Methoden und 1796 Zeilen Code mit 99.99998887 . . . % besonders hoch bewertet.

In diesem Abschnitt werden die Vorteile der Fuzzy-Bedingungen gegenüber den optionalen Attribut- und Metrikbedingungen mit Schranken gezeigt. Im Gegensatz zu optionalen Bedingungen, über die nur absolute Aussagen möglich sind (eine optionale Bedingung ist entweder erfüllt oder nicht), kann der Erfülltheitsgrad von Fuzzy-Bedingungen durch eine Funktion beschrieben werden. Solche Funktionen geben an, wie mögliche Metrikwerte zu bewerten sind, und können[3] die Präzision bei der Bewertung von Musterfunden erheblich erhöhen.

[3] Auch durch Fuzzy-Bedingungen können Schranken formuliert werden.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst die Ergebnisse der Diplomarbeit zusammen. Anschließend werden einige Ideen vorgestellt, die in Zukunft verwirklicht werden können.

7.1 Zusammenfassung

Die unvollständige Spezifikation von Software-Mustern nach dem an der Universität Paderborn entwickelten Mustererkennungsverfahren [NSW+02, NWW03] führt zu False Positives bei der Mustererkennung. Um dem Reverse Engineer einen Hinweis auf die Verlässlichkeit beziehungsweise Qualität der Musterfunde zu geben und dadurch die Auswertung der Suchergebnisse zu erleichtern, werden diese automatisch bewertet.

Im Rahmen dieser Diplomarbeit ist ein neuer Ansatz zur Bewertung von Musterfunden entwickelt und prototypisch realisiert worden. Dieser basiert auf dem an der Universität Paderborn entwickelten Mustererkennungsverfahren und greift die Idee auf, Funde nach dem Grad ihrer Übereinstimmung mit der Musterspezifikation zu bewerten.

Bei der Spezifikation von Mustern werden zusätzlich zu den bisher genutzten Informationen auch Aussagen verwendet, die auf einen Großteil der Musterausprägungen, jedoch nicht auf alle zutreffen. Solche Aussagen werden in Form von optionalen Bedingungen angegeben, wodurch auch von der Spezifikation abweichende Musterinstanzen gefunden werden. Erfüllte optionale Bedingungen drücken eine höhere Wahrscheinlichkeit dafür aus, dass ein Fund tatsächlich eine Instanz des gesuchten Musters ist und untermauern damit das Suchergebnis. Die Bewertung einer Musterinstanz gibt das Verhältnis der erfüllten zu den spezifizierten Bedingungen wieder. Je mehr Bedingungen erfüllt sind, desto mehr stimmt ein Musterfund mit der Spezifikation des Musters überein und desto höher wird dieser bewertet.

Im Gegensatz zu den beiden im Kapitel 3 beschriebenen Bewertungsverfahren werden sowohl die Eigenschaften der Musterfunde berücksichtigt als auch der Anteil der erfüllten Bedingungen beziehungsweise der Grad der Übereinstimmung mit der Musterspezifikation korrekt wiedergegeben. Zuvor ungenutzte Informationen über ein Muster werden nun zur Bewertung von Musterfunden verwendet. Die Präzision des bisherigen Mustererkennungsverfahrens bleibt erhalten.

Die Ausdruckskraft der Musterspezifikationssprache wurde durch optionale Bedingungen, Software-Metriken und so genannte Fuzzy-Bedingungen erhöht. Während Software-Metriken hilfreich bei der Beschreibung von Software-Mängeln wie

Anti Patterns und Bad Smells sind, stellen Fuzzy-Bedingungen eine Alternative für Schranken dar. Insbesondere bei der Spezifikation von Bedingungen wie „viele Attribute“ oder „wenige Zeilen Code“ hat der Reverse Engineer die Möglichkeit, die Begriffe „viel“ und „wenig“ treffender auszudrücken. Dazu wird eine Funktion angegeben, die alle möglichen Metrikwerte auf das Intervall $[0, 1]$ abbildet und damit beschreibt, wie diese zu bewerten sind.

An den Beispielen aus Kapitel 6 wurde gezeigt, dass die Unterschiede, die durch verschiedene von Musterfunden erfüllten Bedingungen entstehen, durch die Bewertung hervorgehoben werden, was mit Hilfe von Gewichtsfaktoren verstärkt werden kann. Außerdem zeigen die Beispiele, dass bereits wenige optionale oder Fuzzy-Bedingungen ausreichen, um eine aussagekräftige Bewertung zu erreichen.

Das neu entwickelte Verfahren zur Bewertung von Musterfunden beseitigt die Mängel der bisherigen Bewertungsmethoden und erleichtert durch höhere Präzision die Identifikation der relevanten Suchergebnisse.

7.2 Ausblick

Zur Zeit ist die Mustersuche so implementiert, dass einem Knoten das erste Objekt zugeordnet wird, das alle nicht optionalen Bedingungen erfüllt. In Zukunft soll die Mustersuche modifiziert werden, um unter allen geeigneten Objekten das Objekt auszuwählen, mit dem die höchstmögliche Bewertung erzielt werden kann.

Das in dieser Arbeit vorgestellte Bewertungsverfahren wird bisher ausschließlich zum Sortieren der Suchergebnisse nach der Mustererkennung genutzt und hat keinen Einfluss auf den Suchalgorithmus. In Zukunft kann die Mustersuche mit Hilfe der Bewertungsfunktion und einer Inferenzstrategie, die eine *Best-First-Suche* implementiert, beschleunigt werden. Es können auch weitere Bewertungsfunktionen realisiert und verglichen werden.

Bei dem Best-First-Suchalgorithmus wird zu jeder Zeit unter allen bisher vorhandenen Teillösungen diejenige für die weitere Suche ausgewählt, die den größten Erfolg verspricht. Die Erfolgsaussicht einer Teillösung wird mit Hilfe einer Bewertungsfunktion quantifiziert. Im Kontext der Mustererkennung entspricht eine Teillösung dem Fund einer Musterinstanz. Dieser kann für die Suche anderer Musterinstanzen Vorraussetzung sein. Durch Angabe von Güteschranken können Funde, welche die Güteschranke nicht erreichen, bei der Suche verworfen und der Suchraum auf diese Weise verkleinert werden.

Schon jetzt gibt es die Möglichkeit, die Suche zu beschleunigen, indem ausschließlich nicht optionale Bedingungen einer Musterregel geprüft werden. Dadurch fallen aber die Bewertungen der Funde niedrig aus, da für sämtliche optionalen Bedingungen angenommen wird, dass sie nicht erfüllt sind. Außerdem sind dann die Bewertungen aller Instanzen zu einem Muster gleich. Bei einer Best-First-Suche mit Güteschranken dagegen würden nur Musterfunde mit den meisten erfüllten Bedingungen weiter verfolgt werden. Zum Beispiel kann durch eine Güteschranke angegeben werden, dass nur nahezu vollständige Instanzen des

Hilfsmusters „SingleReference“ für die Suche nach abstrakteren Mustern wie „Delegation“ und „Strategy“ verwendet werden. Außerdem würden bei der Mustersuche die am höchsten bewerteten Funde als Erstes gefunden werden, sodass die Suche frühzeitig abgebrochen werden kann und dennoch gute Ergebnisse liefern würde.

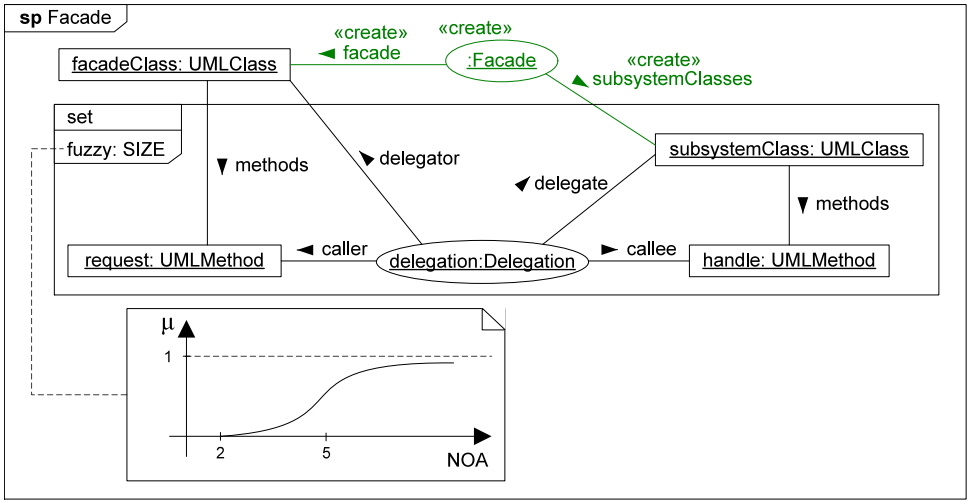


Abbildung 7.1: Mögliche Spezifikation des Design Patterns „Facade“

Zur Zeit können mehrere Vorkommen gleichartiger Objektstrukturen im abstrakten Syntaxgraphen nicht durch eine einzige Musterregel beschrieben werden. Es muss eine Hilfsmusterregel erstellt werden, welche die Struktur spezifiziert. Mit Hilfe eines Annotationsmengenknotens kann anschließend angegeben werden, dass diese mehrfach vorkommt. Durch die zusätzlichen Musterregeln steigen die Laufzeit und der Speicherbedarf bei der Mustersuche. Es wird unter anderem auch an unnötigen Stellen nach Vorkommen der durch die Hilfsmusterregel beschriebenen Struktur gesucht und zusätzliche Annotationen werden erstellt. Außerdem sinkt die Übersicht bei den Suchergebnissen.

Analog zu optionalen Fragmenten und Mengenknoten können Fragmente eingeführt werden, die eine Menge von gleichartigen Objektstrukturen repräsentieren. Es wären keine Hilfmusterregeln mehr nötig, um diese Strukturen zu beschreiben. Die Anzahl der Musterregeln kann auf diese Weise reduziert werden. Die Qualität und die Anzahl der mehrfach vorkommenden Objektstrukturen können wie bei Objekten zu einem Mengenknoten bewertet werden. Außerdem können Metrik- und Fuzzy-Bedingungen zur Metrik „SIZE“ die Anzahl der Strukturen gesondert bewerten.

Die Spezifikation des Design Patterns „Facade“ [GHJV95, S. 185] zum Beispiel kann dann wie in Abbildung 7.1 aussehen. Durch das set-Fragment wird spezifiziert, dass die darin enthaltene Objektstruktur mehrfach gefunden werden soll. Die Anzahl der durch das Fragment repräsentierten Strukturen wird durch eine Fuzzy-Bedingung bewertet.

Ferner kann für jede Musterregel mit optionalen Elementen eine Statistik darüber geführt werden, wie oft eine optionale Bedingung erfüllt beziehungsweise nicht erfüllt war. Wird eine optionale Bedingung nie oder sehr selten erfüllt, so bringt diese keinen beziehungsweise wenig Nutzen für die Bewertung und kann entfernt werden, um die Laufzeit zu reduzieren. Der Reverse Engineer kann auf solche Bedingungen aufmerksam gemacht werden.

Anhang A

Kompakte Beschreibung der Bewertungsfunktion

In diesem Kapitel wird die entwickelte Bewertungsfunktion definiert. Um die Funktion möglichst knapp zu beschreiben, wird hier auf Erklärungen verzichtet. Eine ausführliche Beschreibung befindet sich im Abschnitt 4.5, wo auch die einzelnen bei der Wahl der Funktion getroffenen Entscheidungen begründet sind.

A.1 Ausdrücke und Begriffe

Im Folgenden werden die für die Definition der Bewertungsfunktion benötigten Begriffe und Ausdrücke definiert.

Zu einer Annotation gehörende Musterregel

Der Ausdruck $\mathit{Regel}(a)$ bezeichnet die Musterregel, welche die durch die Annotation a markierte Objektstruktur im abstrakten Syntaxgraphen beschreibt.

Gewichtsfaktor

Der Gewichtsfaktor, der für ein Element e einer Musterregel spezifiziert werden kann, um seinen Einfluss auf die Bewertung von Musterinstanzen zu erhöhen oder zu verringern, wird mit w_e bezeichnet. Der Wertebereich von w_e ist die Menge der nicht negativen reellen Zahlen $\mathbb{R}^+$. Die Elementarten, die mit einem Gewicht versehen werden können, sind in der Tabelle 4.2 auf Seite 56 aufgelistet. Wurde kein Gewichtsfaktor spezifiziert, so gilt $w_e = 1$.

Hierarchie der Musterregelelemente

Die Elemente einer Musterregel können hierarchisch angeordnet sein, zum Beispiel kann ein optionales Fragment Objektknoten enthalten, diese wiederum können Attributbedingungen enthalten. Um diese Beziehungen zu beschreiben, wird der Ausdruck $\mathit{Elemente}(e)$ für eine Musterregel oder ein in einer Musterregel enthaltenes Element e induktiv definiert:

- Die Elemente eines Constraints, einer Attribut-, Metrik- oder Fuzzy-Bedingung b ist die leere Menge: $\mathit{Elemente}(b) = \emptyset$.
- Die Elemente eines Objektknotens oder Objektmengenknotes o ist die Menge aller zu dem Knoten o spezifizierten Attribut-, Metrik- und Fuzzy-Bedingungen.
- Die Elemente eines Annotationsknotens a ist die leere Menge.
- Die Elemente eines Annotationsmengenknotens a ist die Menge aller zu dem Knoten spezifizierten Metrik- und Fuzzy-Bedingungen.
- Die Elemente eines optionalen Fragments f ist die Menge der innerhalb des Fragments spezifizierten Knoten und Constraints (ausgenommen maybe-Constraints).
- Die Elemente einer Musterregel r ist die Menge aller in der Regel spezifizierten Constraints (ausgenommen maybe-Constraints), Knoten und optionaler Fragmente.

Nicht abstrakte Unterregeln

Für einen Annotationsknoten k und die Musterregel r, welche die durch den Knoten k beschriebene Objektstruktur im abstrakten Syntaxgraphen spezifiziert, wird der Ausdruck $\mathit{NAUnterregeln}(k)$ definiert als die Menge der nicht abstrakten Unterregeln von r, die Regel r eingeschlossen.

Mengenbedingungen

Der Ausdruck $\mathit{MengenBedingungen}(k)$ bezeichnet die Menge der zum Mengenknoten k spezifizierten Mengenbedingungen, also der Metrik- und Fuzzy-Bedingungen bezüglich der Metrik „SIZE“. Der Ausdruck $\mathit{NichtMengenBedingungen}(k)$ bezeichnet die Menge der zum Mengenknoten k definierten Attribut-, Metrik- und Fuzzy-Bedingungen, die sich nicht auf die Metrik „SIZE“ beziehen. Es gilt $\mathit{MengenBedingungen}(k) \cup \mathit{NichtMengenBedingungen}(k) = \mathit{Elemente}(k)$.

Skalierfunktion s

Die Skalierfunktion s mit der Signatur $s : \mathbb{R}^+ \to [0, 1)$, $x \mapsto s(x)$ wird wie folgt definiert:

$$s(x) = \frac{2}{1 + e^{-\frac{x}{10}}} - 1$$

Der Graph der Funktion ist in der Abbildung 4.10 auf Seite 74 dargestellt.

A.2 Bewertung einer Annotation

Eine Annotation a wird durch folgende Formel bewertet:

$$Bewertung(a) = \frac{Erfülltheit(Regel(a), a)}{Gewicht(Regel(a))}$$

Hierbei bezeichnet $Erfülltheit(Regel(a), a)$ für die Annotation a die gewichtete Summe der Erfülltheitsgrade aller in der Regel $Regel(a)$ definierten Bedingungen, während der Ausdruck $Gewicht(Regel(a))$ die Summe der Gewichte dieser Bedingungen bezeichnet (vgl. Gleichung 4.2, S. 49). Diese werden im Folgenden induktiv über den hierarchischen Aufbau einer Musterregel definiert.

A.2.1 Gewicht von Musterregeln und enthaltenen Elementen

Das Gewicht einer Musterregel und der darin enthaltenen Elemente wird wie folgt induktiv definiert:

Gewicht von Constraints, Attribut-, Metrik- und Fuzzy-Bedingungen

Das Gewicht eines Constraints, einer Attribut-, Metrik- oder Fuzzy-Bedingung b wird als der Gewichtsfaktor von b definiert:

$$Gewicht(b) = w_b$$

Gewicht von Objektknoten und Objektmengenknoten

Das Gewicht eines Objektknotens oder Objektmengenknotens k wird wie folgt definiert:

$$Gewicht(k) = \begin{cases} w_k + \sum_{e \in Elemente(k)} Gewicht(e), & \text{falls der Knoten } k \text{ nicht negativ ist,} \\ w_k, & \text{sonst} \end{cases}$$

Gewicht von Annotationsknoten

Das Gewicht eines Annotationsknotens k wird als das Produkt aus dem Gewichtsfaktor von k und dem Durchschnitt aller Gewichte der nicht abstrakten Unterregeln der zum Annotationsknoten k gehörenden Musterregel definiert:

$$Gewicht(k) = \begin{cases} w_k \cdot \frac{\sum_{r \in NAUnterregeln(k)} Gewicht(r)}{|NAUnterregeln(k)|}, & \text{falls der Knoten } k \text{ nicht negativ ist,} \\ w_k, & \text{sonst} \end{cases}$$

Gewicht von Annotationsmengenknoten

Das Gewicht eines Annotationsmengenknotens k, wird wie folgt definiert:

$$\mathit{Gewicht}(k) = w_k \cdot \frac{\sum_{r \in \mathit{NAUnterregeln}(k)} \mathit{Gewicht}(r)}{|\mathit{NAUnterregeln}(k)|} + \sum_{e \in \mathit{MengenBed}(k)} \mathit{Gewicht}(e)$$

Gewicht von optionalen Fragmenten

Das Gewicht eines optionalen Fragments f wird als das Produkt aus dem Gewichtsfaktor des Fragments f und der Summe der Gewichte von darin spezifizierten Elementen definiert:

$$\mathit{Gewicht}(f) = w_f \cdot \sum_{e \in \mathit{Elemente}(f)} \mathit{Gewicht}(e)$$

Gewicht von Musterregeln

Das Gewicht einer Musterregel r wird als die Summe der Gewichte von darin spezifizierten Elementen definiert:

$$\mathit{Gewicht}(r) = \sum_{e \in \mathit{Elemente}(r)} \mathit{Gewicht}(e)$$

A.2.2 Erfülltheit von Bedingungen

Die Erfülltheit aller durch eine Musterregel definierten Bedingungen für eine Musterinstanz wird im Folgenden induktiv definiert:

Erfülltheit von Constraints

Für eine Annotation a und ein in der Musterregel $\mathit{Regel}(a)$ spezifiziertes Constraint c wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(c, a) = \begin{cases} w_c, & \text{falls das Constraint } c \text{ für die Annotation } a \text{ erfüllt} \\ & \text{ist,} \\ 0, & \text{sonst} \end{cases}$$

Erfülltheit von Attribut- und Metrikbedingungen

Für ein Objekt o im abstrakten Syntaxgraphen und eine Attribut- oder Metrikbedingung b wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(b, o) = \begin{cases} w_b, & \text{falls die Bedingung } b \text{ für das Objekt } o \text{ erfüllt ist,} \\ 0, & \text{sonst} \end{cases}$$

Für eine Objekt- oder Annotationsmenge M und eine Metrikbedingung b bezüglich der Metrik „SIZE“ wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(b, M) = \begin{cases} w_b, & \text{falls die Bedingung } b \text{ für die Menge } M \text{ erfüllt ist,} \\ 0, & \text{sonst} \end{cases}$$

Erfülltheit von Fuzzy-Bedingungen

Für ein Objekt o im abstrakten Syntaxgraphen und eine Fuzzy-Bedingung b bezüglich einer Metrik m, eine für b definierte Zugehörigkeitsfunktion μ_b und dem Metrikwert $x_{m,o}$ für die Metrik m und das Objekt o wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(b, o) = w_b \cdot \mu_b(x_{m,o}),$$

Für eine Objekt- oder Annotationsmenge M, eine Fuzzy-Bedingung b bezüglich der Metrik „SIZE“, eine für b definierte Zugehörigkeitsfunktion μ_b und dem Metrikwert $x_{m,M}$ für die Metrik m und die Menge M wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(b, M) = w_b \cdot \mu_b(x_{m,M}),$$

Erfülltheit von Objektknoten

Für eine Annotation a und einen in der Musterregel $\mathit{Regel}(a)$ spezifizierten Objektknoten k wird die Erfülltheit definiert durch:

$$\mathit{Erfülltheit}(k, a) = \begin{cases} w_k, & \text{falls der Knoten } k \text{ negativ ist,} \\ w_k + \sum_{e \in \mathit{Elemente}(k)} \mathit{Erfülltheit}(e, o), & \text{falls } k \text{ nicht negativ ist und beim Erstellen der Annotation } a \text{ dem Knoten } k \text{ ein Objekt } o \text{ zugeordnet wurde,} \\ 0, & \text{sonst} \end{cases}$$

Erfülltheit von Objektmengenknoten

Für eine Annotation a, einen in der Musterregel $Regel(a)$ spezifizierten Objektmengenknoten k sowie der Menge O der beim Erstellen der Annotation a dem Knoten k zugeordneten Objekte wird die Erfülltheit definiert durch:

$$\begin{aligned} Erfülltheit(k,a) &= s\left(\sum_{o\in O} B(k,o)\right)\cdot G(k) \\ &\quad + \sum_{e\in MengenBedingungen(k)} Erfülltheit(e,O) \end{aligned}$$

mit

$$B(k,o) = \frac{w_k + \sum_{e\in NichtMengenBedingungen(k)} Erfülltheit(e,o)}{w_k + \sum_{e\in NichtMengenBedingungen(k)} Gewicht(e)}$$

und

$$G(k) = w_k + \sum_{e\in NichtMengenBedingungen(k)} Gewicht(e)$$

Erfülltheit von Annotationsknoten

Für eine Annotation a und einen in der Musterregel $Regel(a)$ spezifizierten Annotationsknoten k wird die Erfülltheit definiert durch:

$$Erfülltheit(k,a) = \begin{cases} w_k, & \text{falls der Knoten } k \text{ negativ ist,} \\ Gewicht(k)\cdot Bewertung(o), & \text{falls beim Erstellen der Annotation } a \text{ dem Knoten } k \text{ eine Annotation } o \text{ zugeordnet wurde,} \\ 0, & \text{sonst} \end{cases}$$

Erfülltheit von Annotationsmengenknoten

Für eine Annotation a, einen in der Musterregel $Regel(a)$ spezifizierten Annotationsmengenknoten k sowie der Menge A der beim Erstellen der Annotation a dem Knoten k zugeordneten Annotationen wird die Erfülltheit definiert durch:

$$\begin{aligned} Erfülltheit(k,a) &= s\left(\sum_{b\in A} Bewertung(b)\right)\cdot G(k) \\ &\quad + \sum_{e\in MengenBedingungen(k)} Erfülltheit(e,A) \end{aligned}$$

mit

$$G(k) = w_k \cdot \frac{\sum_{r\in NAUnterregeln(k)} Gewicht(r)}{|NAUnterregeln(k)|}$$

Erfülltheit von optionalen Fragmenten

Für eine Annotation a und ein in der Musterregel $Regel(a)$ spezifiziertes optionales Fragment f wird die Erfülltheit definiert durch:

$$Erfülltheit(f, a) = w_f \cdot \sum_{e \in Elemente(f)} Erfülltheit(e, a)$$

Erfülltheit von Musterregeln

Für eine Annotation a und die Musterregel $r = Regel(a)$ wird die Erfülltheit definiert durch:

$$Erfülltheit(r, a) = \sum_{e \in Elemente(r)} Erfülltheit(e, a)$$

Anhang B

Musterregeln

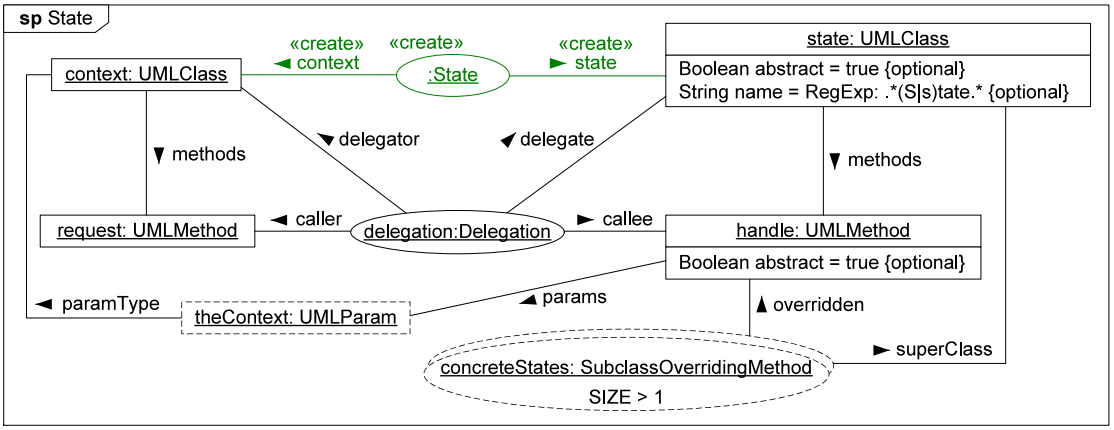


Abbildung B.1: Musterregel zum Entwurfsmuster „State“

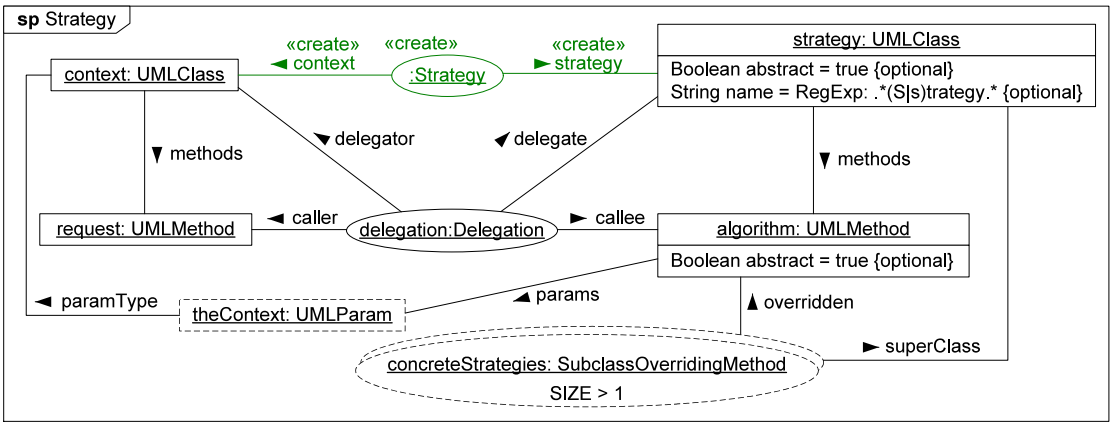


Abbildung B.2: Musterregel zum Entwurfsmuster „Strategy“

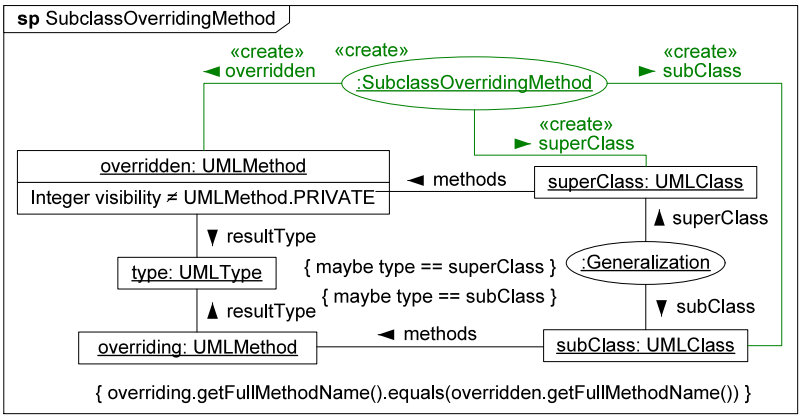


Abbildung B.3: Musterregel „SubclassOverridingMethod“

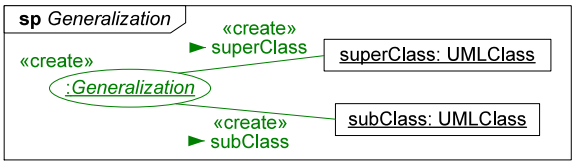


Abbildung B.4: Musterregel „Generalization“

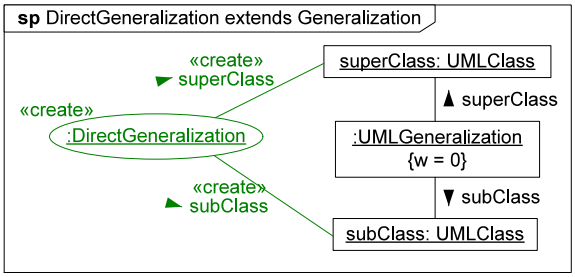


Abbildung B.5: Musterregel „DirectGeneralization“

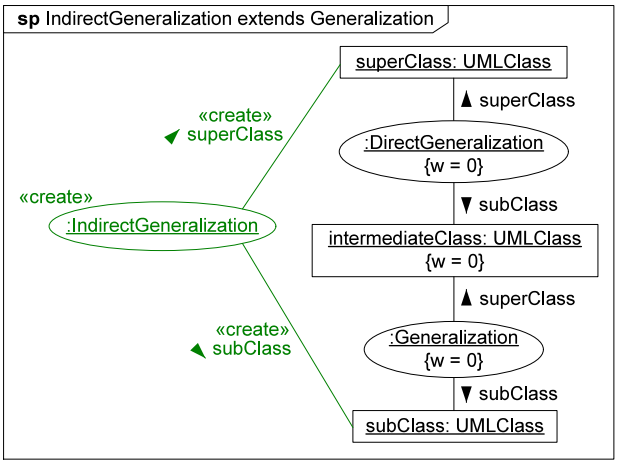


Abbildung B.6: Musterregel „IndirectGeneralization“

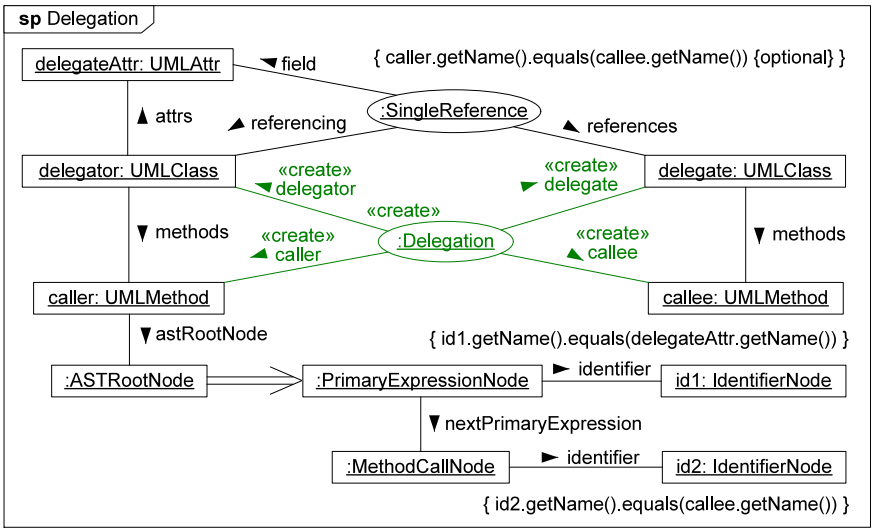


Abbildung B.7: Musterregel „Delegation“

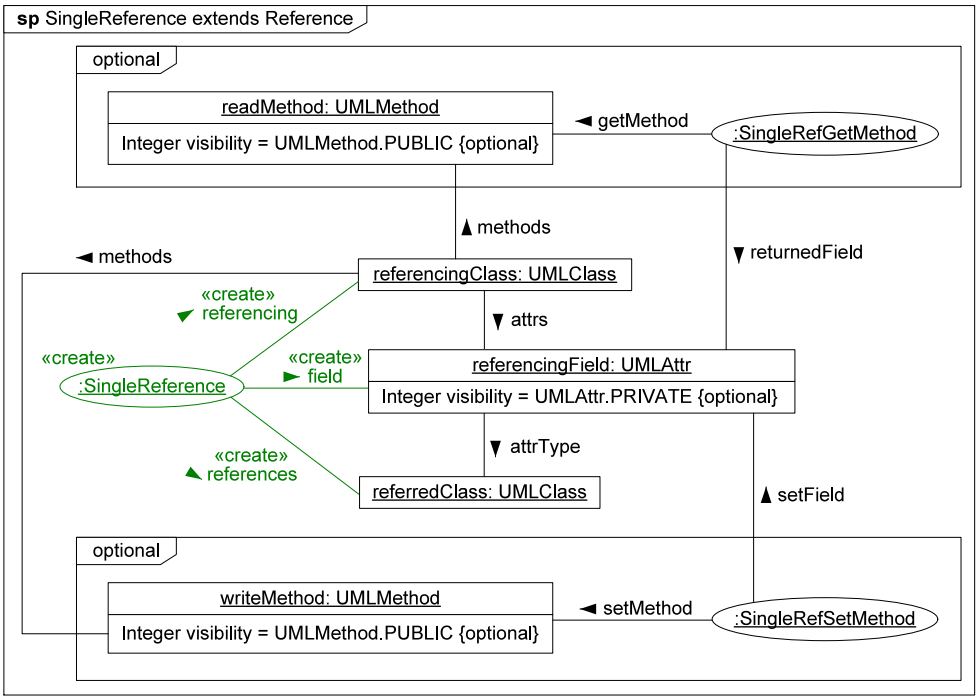


Abbildung B.8: Musterregel „SingleReference“

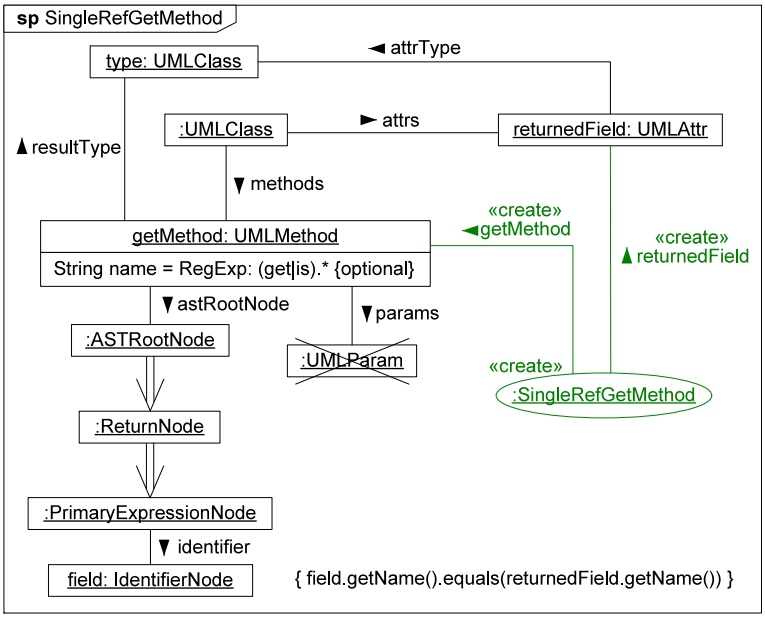


Abbildung B.9: Musterregel „SingleRefGetMethod“

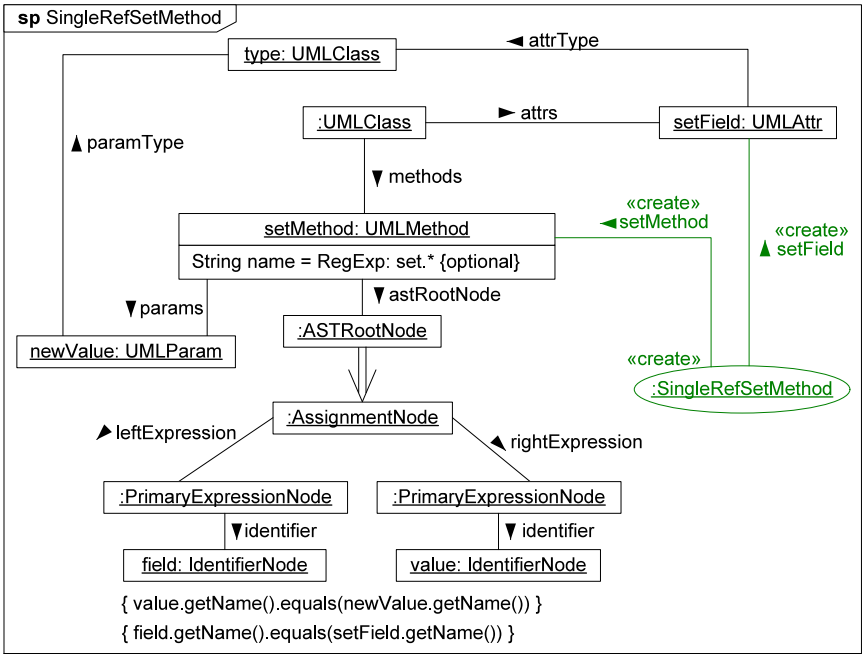


Abbildung B.10: Musterregel „SingleRefSetMethod“

Literatur

[AACGJ01] Albin-Amiot, Hervé; Cointe, Pierre; Guéhéneuc, Yann-Gaël; Jussien, Narendra: Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together. In: *Proc. of the* 16^{th} *International Conference on Automated Software Engineering (ASE), San Diego, USA*, IEEE Computer Society Press, November 2001, S. 166–173

[AFC98] Antoniol, Giuliano; Fiutem, Roberto; Cristoforetti, Lucas: Design Pattern Recovery in Object-Oriented Software. In: *Proc. of the* 6^{th} *International Workshop on Program Comprehension (IWPC), Ischia, Italy*, IEEE Computer Society Press, Juni 1998, S. 153–160

[Bac01] Bachl, Sabine: *Erkennung isomorpher Subgraphen und deren Anwendung beim Zeichnen von Graphen*, Universität Passau, Dissertation, 2001

[BBS05] Blewitt, Alex; Bundy, Alan; Stark, Ian: Automatic Verification of Design Patterns in Java. In: *Proc. of the* 20^{th} *International Conference on Automated Software Engineering (ASE), Long Beach, USA*, 2005, S. 224–232

[BL03] Beyer, Dirk; Lewerentz, Claus: CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs. In: *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, IEEE Computer Society Press, Mai 2003, S. 294–295

[BMMM98] Brown, William J.; Malveau, Raphael C.; McCormick III, Hays W.; Mowbray, Thomas J.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1998

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

[Ecl06] Eclipse Foundation Inc. *Eclipse Project.* Online unter: http://www.eclipse.org/eclipse/. Stand: August 2006

[FBFL05] Ferenc, Rudolf; Beszédes, Árpád; Fülöp, Lajos; Lele, János: Design Pattern Mining Enhanced by Machine Learning. In: *Proc. of*

the 21st International Conference on Software Maintenance (ICSM), Budapest, Hungary, 2005, S. 295–304

[FM04] Fabry, Johan; Mens, Tom: Language Independent Detection of Object-Oriented Design Patterns. In: *Computer Languages, Systems and Structures* (2004), Februar

[FNT98] Fischer, Thorsten; Niere, Jörg; Torunski, Lars: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, Universität Paderborn, Diplomarbeit, Juli 1998

[FNTZ98] Fischer, Thorsten; Niere, Jörg; Torunski, Lars; Zündorf, Albert: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, G. (Hrsg.); Rozenberg, G. (Hrsg.): *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer Verlag, November 1998 (LNCS 1764), S. 296–309

[Fow99] Fowler, Martin: *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1999

[Fuj06] Fujaba Development Group. *Fujaba Tool Suite.* Online unter: http://www.fujaba.de. Stand: August 2006

[GB03] Gamma, Erich; Beck, Kent: *Contributing to Eclipse – Principles, Patterns and Plug-ins.* Addison-Wesley, 2003

[GHJV95] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: Elements of Reusable Object Oriented Software.* Addison-Wesley, 1995

[GJ01] Guéhéneuc, Yann-Gaël; Jussien, Narendra: Using Explanations for Design Patterns Identification. In: Bessière, Christian (Hrsg.): *Proc. of the 1st IJCAI workshop on Modeling and Solving Problems with Constraints, Seattle, USA*, AAAI Press, August 2001, S. 57–64

[KP96] Krämer, Christian; Prechelt, Lutz: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: *Proc. of the 3rd Working Conference on Reverse Engineering, Monterey, USA*, 1996, S. 208–215

[MN05] Meyer, Matthias; Niere, Jörg: Calculation and Visualization of Software Product Metrics. In: Giese, Holger (Hrsg.); Zündorf, Albert (Hrsg.): *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany* Bd. tr-ri-05-259, University of Paderborn, September 2005, S. 41–44

[Nie04] Niere, Jörg: *Inkrementelle Entwurfsmustererkennung*, Universität Paderborn, Dissertation, Juni 2004

[NMW04] Niere, Jörg; Meyer, Matthias; Wendehals, Lothar: User-Driven Adaption in Rule-Based Pattern Recognition / Universität Paderborn. 2004 (tr-ri-04-249). – Forschungsbericht

[NSW+02] Niere, Jörg; Schäfer, Wilhelm; Wadsack, Jörg P.; Wendehals, Lothar; Welsh, Jim: Towards Pattern-Based Design Recovery. In: *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, ACM Press, Mai 2002, S. 338–348

[NWW03] Niere, Jörg; Wadsack, Jörg P.; Wendehals, Lothar: Handling Large Search Space in Pattern-Based Reverse Engineering. In: *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, IEEE Computer Society Press, Mai 2003, S. 274–279

[Obj04] Object Management Group, Inc. *UML 2.0 Superstructure Specification*. Oktober 2004

[Pal01] Palasdies, Marcus: *Design-Pattern Spezifikation und Erkennung auf Basis von Story-Diagrammen*, Universität Paderborn, Diplomarbeit, Mai 2001

[Rec04] Reckord, Carsten: *Optimierung von Genauigkeitswerten unscharfer Regeln*, Universität Paderborn, Diplomarbeit, Mai 2004

[Rot05] Roth, Lukas: *Metrik-basierte Analyse von UML-Modellen*, Universität Paderborn, Studienarbeit, August 2005

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation*. Bd. 1. World Scientific, 1997

[Ser94] Seraphin, Marco: *Neuronale Netze und Fuzzy-Logik*. Franzis-Verlag, 1994

[Sof06] Software Engineering Institue, Carnegie Mellon University, USA. *Cyclomatic Complexity - Software Technology Roadmap*. Online unter: http://www.sei.cmu.edu/str/descriptions/cyclomatic_body.html. Stand: August 2006

[TM05] Travkin, Dietrich; Meyer, Matthias: Generation of Type Safe Association Implementations. In: Giese, Holger (Hrsg.); Zündorf, Albert (Hrsg.): *Proc. of the* 3^{rd} *International Fujaba Days 2005, Paderborn, Germany* Bd. tr-ri-05-259, Universität Paderborn, September 2005, S. 63–66

[Tra05] Travkin, Dietrich: *Generierung typsicherer Implementierungen für Assoziationen in UML-Modellen*, Universität Paderborn, Studienarbeit, Februar 2005

[TU97] Tsoukalas, Lefteri H.; Uhrig, Robert E.: *Fuzzy and Neural Approaches in Engineering.* John Wiley and Sons, 1997

[Wen01] Wendehals, Lothar: *Cliché- und Mustererkennung auf Basis von Generic Fuzzy Reasoning Nets*, Universität Paderborn, Diplomarbeit, Oktober 2001

[Wen05a] Wenzel, Sven: *An Approach to the Automatic Recognition of Design Forms*, Universität Dortmund und Tampere University of Technology, Diplomarbeit, September 2005

[Wen05b] Wenzel, Sven: Automatic Detection of Incomplete Instances of Structural Patterns in UML Class Diagrams. In: *Proc. of the* 3^{rd} *Nordic Workshop on UML and Software Modeling (NWUML), Tampere, Finland*, Universität Tampere, August 2005

[Wen05c] Wenzel, Sven: Detection of Incomplete Patterns Using Fujaba Principles. In: Giese, Holger (Hrsg.); Zündorf, Albert (Hrsg.): *Proc. of the* 3^{rd} *International Fujaba Days 2005, Paderborn, Germany* Bd. tr-ri-05-259, Universität Paderborn, September 2005, S. 33–40

[Zün01] Zündorf, Albert: *Rigorous Object Oriented Software Development*, Universität Paderborn, Habilitation, 2001

[Zün02] Zündorf, Albert: *Rigorous Object Oriented Software Development.* Universität Paderborn, Mai 2002. – Erweiterung von [Zün01], Entwurf, Version 0.3, online unter http://www.uni-paderborn.de/fachbereich/AG/schaefer/Personen/Ehemalige/Zuendorf/AZRigSoftDraft_0_2.pdf und http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Zuen02.pdf, Stand: August 2006

Index