Annotationen zur Anwendung beim Refactoring

Klaus Meffert, Ilka Philippow

Technische Universität Ilmenau
98693 Ilmenau
PF 100565
meffert@rz.tu-ilmenau.de
ilka.philippow@tu-ilmenau.de

Abstract: Die Restrukturierung von Programmcode, das Refactoring, verfolgt das Ziel, die Wartbarkeit eines Softwaresystems zu verbessern. Dabei werden häufig Entwurfsmuster eingesetzt. Um das ursprüngliche Verhalten des Programms bei Umstrukturierung zu erhalten, muss sowohl das Programm als auch die Wirkung der Refactoring-Operation sehr gut verstanden sein. Refactoring-Operationen setzen bestimmte Bedingungen im Programmcode voraus. Eine Möglichkeit, syntaktische und semantische Details für ein spezifisches Codefragment zu bewahren, bieten dokumentierende Annotationen, die einerseits maschinenverarbeitbar und andererseits für den Entwickler lesbar sind. In dieser Arbeit werden Annotationen vorgestellt, die explizite Semantikinformationen enthalten und durch codeevaluierende Werkzeuge oder manuell in den Quelltext eingefügt werden. Mit geeigneten Annotationen ist es möglich, automatisiert Vorbedingungen für eine Refactoring-Operation, speziell auch für die Anwendung von Entwurfsmustern zu prüfen.

1 Einführung

Werkzeugunterstützte Refactoring-Operationen werden verwendet, um Codeartefakte wartbar und erweiterbar zu halten. Typische Refactoring-Operationen können sehr einfach sein, z.B. *Encapsulate Field* (vgl. [Fo99]), aber auch sehr komplex, wie beispielsweise die Anwendung von Entwurfsmustern (z.B. *Composite*, vgl. [Ga95]). Aktivitäten, die zur Qualitätssicherung der Architektur langlebiger Software notwendig sind, kommen aus Zeitgründen oft zu kurz. Dagegen können gut strukturierte und verständliche Programme Aufwand und Kosten für die Wartung beachtlich reduzieren. Werkzeuge für die automatisierte Ausführung von Refactoring-Operationen helfen, den Entwickler von Routinetätigkeiten zu entlasten. Einige Entwicklungsumgebungen (IDEs) unterstützen die Anwendung von Entwurfsmustern, zumindest für deren statische Aspekte (etwa Schnittstellen). Quellcode-Analysewerkzeuge wiederum können Informationen extrahieren, die für ein Refactoring relevant sind (etwa [PMD]).

Für eine korrekte Musteranwendung in einem gegebenen Kontext muss zunächst das geeignete Muster gefunden und dessen Anwendbarkeit geprüft werden. Aufgrund der großen Anzahl bis heute beschriebener Muster kann der Aufwand beträchtlich werden. Die falsche Handhabung von Entwurfsmustern beim Refactoring kann negative

Auswirkungen haben. In [Me06] sind weitere Probleme bei der Anwendung von Mustern beschrieben, so auch das Problem, die genauen Entwurfsentscheidungen für ein Codestück zu erkennen. Quellcode-Analysewerkzeuge und solche, die auf Quellcode operieren, besitzen häufig kein gemeinsames Kommunikationsinterface. Die Analyseergebnisse müssen manuell durch den Entwickler übertragen werden. Dabei passt der Entwickler implizit Analyseergebnisse an die Konzepte der Benutzerwerkzeuge an. Abb. 1 zeigt diesen Vorgang (das *i*-symbol markiert den Informationsfluss).

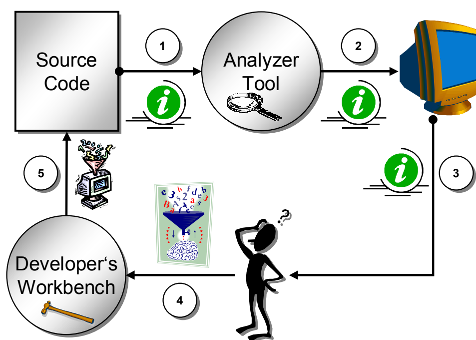

Abbildung 1: Analyse und Modifikation von Quellcode

Lautet das Analyseergebnis für ein Codefragment (Schritt 1, Abb. 1) etwa, dass "die Klasse X eine zyklomatische Komplexität von 17 besitzt" (Schritt 2), dann muss der Entwickler die Bedeutung dieser Aussage bewerten (Schritt 3). Andererseits können IDEs Refactoring-Operationen, wie beispielsweise *Consolidate Conditional Expression* (Schritt 5) ausführen. Auch wenn der Entwickler weiß, dass die Anwendung dieser Operation auf das Codefragment (Schritt 4) die Komplexität senken würde, müssen vor Anwendung einer solchen Operation ihre Vorbedingungen geprüft werden. Ist eine der Bedingungen nicht erfüllt, so kann das das erwünschte Resultat gefährden. Mit dem Ansatz in dieser Arbeit soll ein Entwickler bei seiner Arbeit mit Refactoring-Werkzeugen durch Zusatzinformationen in Codefragmenten unterstützt werden, die außerdem Werkzeugen die Kommunikation untereinander gestatten. Im nächsten Abschnitt werden zunächst existierende Ansätze und Werkzeuge für das werkzeugunterstützte Refactoring vorgestellt.

2 Verwandte Arbeiten

In [KR06] werden Annotationen (nach [JSR175]) verwendet, um einen Bezug von Quellcode zur Architekturbeschreibung zu etablieren. So verweist etwa die Annotation *@Component* auf die Verbindung einer Klasse zu einer Komponente, die in der Architekturbeschreibung vorgegeben ist. Nach Aussage der Autoren kann deren Ansatz Codeänderungen für ein Refactoring voraussehen und spätere Konsistenzprüfungen durchführen. LePUS [Ed02] setzt auf die formale Definition von Entwurfsmustern. Der Ansatz ist auf PROLOG begründet und unterstützt weitere Aktivitäten wie Validierung

und Musterdetektion. [TCN03] zielt auf die Musterbeschreibung und unterstützt eine Untermenge davon. [Sa03] versucht, anwendbare Entwurfsmuster durch den Vergleich zweier Versionen von Legacy-Code zu selektieren.

[Jackpot] untersucht Java-Quellcode auf Übereinstimmung mit generischen Regeln. Auf jede Regel wird eine Java-Anweisung abgebildet. Wenn die Abbildung für ein Codefragment möglich ist, dann spezifiziert die Regel, wie das Codefragment zu transformieren ist, um eine Refakturierung durchzuführen. Die Transformation wird ohne Nutzereingriff durch Jackpot ausgeführt. Die Regeln arbeiten auf einem abstrakten Syntaxbaum. Sie sind statisch und können verschachtelt werden. [JSR175] führt Annotationen als valide Sprachkonstrukte in Java ein. [JSR175] erlaubt die Definition von Annotationstypen, die im Quellcode zur Anwendung kommen. Das Konzept beinhaltet u.a. die Definition von Gültigkeitsbereichen für vererbte Annotationen, Parameter, und Defaultwerte für Parameter. Annotationen gelten dabei für Deklarationen und geerbte Annotationen, jedoch nicht für Anweisungen. Ein *annotation processing tool* (*apt*) ermöglicht die Auswertung der Annotationen im Code.

Neben [KR06] gibt es noch weitere Ansätze, die Annotationen verwenden, aber nicht explizit auf Refactoring zielen. JML [LC03] und [XDoclet] führen auf Javadoc basierende Annotation nach demselben Konzept, jedoch aus unterschiedlichen Gründen ein. Das Konzept besteht darin, dass der Anwender manuell syntaktische Beschränkungen und Eigenschaften definiert, wie Multiplizität, Typen, Wertebereiche einer Entität. Beide Ansätze beziehen sich auf Annotationen für Deklarationen, nicht jedoch auf Anweisungen. Weitere Ansätze wie [Ny03] basieren auf natürlicher Sprache, die informal und damit weitaus schwieriger zu interpretieren ist.

[PMD] ist ein regelbasiertes Analysewerkzeug für Javacode. So schlägt z.B. die Regel *UseSingleton* die Anwendung von *Singleton* [Ga95] vor, wenn für eine Klasse nur statische Methoden gefunden wurden. Die PMD-Regel *CyclomaticComplexity* berechnet die gleichnamige Metrik.

In dieser Arbeit wird der Ansatz der Annotationen als Träger verlässlicher Informationen weiterentwickelt und für die Auswahl und Anwendung von Entwurfsmustern ausgebaut.

3 Lösungsansatz

Ein Problem beim Refactoring ist die fehlende Werkzeugunterstützung für die Bestimmung sinnvoller Operationen und ihre Anwendung. Ein Entwickler ist heute kaum in der Lage, relevante und u.U. komplexe Informationen in den Quellcode einzufügen, die durch Standardwerkzeuge (IDEs) verarbeitet werden können. Ein erster Ansatz in dieser Richtung ist in [JSR175] für Java vorgestellt. Mit dem hier beschriebenen Ansatz wird das Ziel verfolgt, die Selektion geeigneter Refactoring-Operationen zu vereinfachen. Das soll durch die Einbettung einer Kommunikationsschicht in den Quellcode erreicht werden, deren Aufgabe im Informationstransport zwischen unterschiedlichen Entitäten besteht. Eine Entität ist ein Analysewerkzeug, eine Workbench oder ein Entwickler. Der Prozess in Abb. 1 wird so

erweitert, dass zwischen Analysewerkzeug und Entwicklerwerkzeug der annotierte Quelltext als Kommunikationsschicht geschaltet wird. Die sich ergebenden Schritte sind:

- Ausführung einer konventionellen Analyse des Quelltextes.
- Dokumentation der Analyseresultate durch Einfügen geeigneter Annotationen.
- Auswertung und Evaluierung der Annotationen durch die IDE.
- Anzeige der Auswertungsergebnisse für den Entwickler, wenn möglich verbunden mit Empfehlungen für das Refactoring.
- Bewertung der Auswertungsergebnisse und Empfehlungen durch den Entwickler.
- Auswahl der Refactoring-Operation.
- Ausführung der gewählten Operation durch die IDE unter Einbeziehung der durch die Annotationen erhaltenen Informationen.

Annotationen können bereits während der Codeentwicklung eingefügt werden. Zusammenfassend kann die hier zugrunde liegende Idee mit den folgenden Schritten beschrieben werden:

- Einfügen von syntaktischen Informationen in den Quellcode, die sonst durch AST-Analyse (AST = Abstrakter Syntaxbaum) extrahiert werden müssen.
- Einfügen von semantischen Informationen in den Quellcode, die sonst automatisch nicht zu finden sind.
- Einsatz eines Formates, welches sowohl für den Menschen lesbar als auch von der Maschine verarbeitbar ist.
- Verbinden eines jeden Informationselements mit einem Programmelement.
- Definition des Gültigkeitsbereiches (engl.: *scope*) für jede Annotation.
- Verbinden der syntaktischen und semantischen Informationen im Quelltext mit der Musterdokumentation .
- Bestimmung der Anwendbarkeit eines Musters durch den Abgleich syntaktischer und semantischer Informationen zwischen Quellcode und Musterdokumentation.

3.1 Einführung von Annotationen

Annotationen werden durch einen Entwickler oder ein Werkzeug in den Quellcode eingesetzt. Eine Annotation etabliert eine Korrelation zwischen einer syntaktischen oder semantischen Information und einem Codesegment. Um Korrelation herzustellen, werden Annotationen oberhalb des Programmfragmentes platziert. Syntaktische Informationen kommen aus einer AST-Analyse oder aus der Kenntnis eines Entwicklers zu einem Codefragment. Semantische Informationen können in der Regel nur aus dem Kontext eines Codefragments gewonnen werden und fordern in der Regel den Entwickler.

Eine Annotation muss sowohl für den Menschen verständlich als auch maschinenverarbeitbar sein. Das kann prinzipiell durch s.g. duale Annotationen realisiert werden. Eine duale Notation besitzt nur eine Aussage und folgt damit [JSR175], [LC03] oder [Me06]. Eine duale Annotation wird dual genannt, weil durch sie die zweifache Zielstellung verfolgt wird, eine Aussage aufzustellen, die gleichzeitig für den Menschen

lesbar und durch die Maschine verarbeitbar ist. Hier wird vorgeschlagen, aus zwei Teilen bestehende, zusammengesetzte Annotationen zu verwenden: einen Teil für die maschinelle Auswertung einer Annotation, der zweite als Informationsträger für den Entwickler. Die Unterscheidung beider Teile kann durch die Einführung eigener Identifizierer (IDs) erfolgen. Eine ID kann als s.g. *GUID* (*Globally Unique Identifier*) definiert sein, um möglichst weltweit einzigartig zu sein. Die maschinenlesbare Seite einer Annotation (abgekürzt mit MID) besteht aus:

1. ID,
2. Parameter (jeder Parameter besitzt einen Namen und einen Typ).

Die an Entwickler gerichtete Seite (abgekürzt mit HID) besteht aus:

1. der ID der dazugehörenden MID,
2. der Beschreibung in natürlicher Sprache, einschließlich der Parameter,
3. eines Parameters zur Sprachidentifikation.

Für die Parameter in MID und HID werden die gleichen Namen verwendet und durch ein Präfix markiert (hier wird dafür @ verwendet). Der Sprachparameter erlaubt HID-Sektionen in verschiedenen Sprachen zu integrieren. Ein Beispiel ist in Abb. 2a gezeigt. Die in Abb. 2a verwendeten Typen (String) und die Sprache (Englisch) sind Konstanten, die einmal definiert werden müssen. Die Verbindung und Referenzierung von MID und HID erfolgt über die ID der MID und durch die gleiche Namensgebung verwendeter Parameter.

a) zusammengesetzte Annotation

```
MID:    1234(("var", String), ("class", String))
HID:    MID=1234, "The local variable @var is not used in class @class",
        English
```

b) Angewandte syntaktische Annotation

```
    /*
     * @syntactic
     * MID: 1234("surname", "x")
     * HID: Local variable surname is not used in class x
     */
    private String surname;
```

c) Annotation als Endemarker

```
    /*
     * @syntactic
     * ID=4711
     * MID:...
     * HID:...
     */
    int x = getAmount();
    doSomethingWith(x);
    print(x);
    /* @end-marker ID=4711*/
```

Abbildung 2: Definition von Annotationen

Zusammengesetzte Annotationen besitzen den Vorteil, dass nicht zwanghaft – wie bei einfachen Annotationen gefordert – nach Aussagen gesucht werden muss, die gleichzeitig lesbar als auch maschinenverarbeitbar sind. Denn die Aussagekraft von MID und HID ist die gleiche, das Ausdrucksmittel dafür kann unabhängig voneinander formuliert werden. Ähnlichkeiten zwischen zusammengesetzten Annotationen müssen wiederum explizit definiert werden, z.B. durch den Aufbau einer Tabelle mit Paaren von ähnlichen Ausdrücken. Der Vorteil einer solchen expliziten Definition besteht in der Möglichkeit, zu jeder Ähnlichkeitsbeziehung weitere Zusatzinformationen anzugeben. Bei einfachen Annotationen wäre dies auch nur über eine entsprechende Ähnlichkeitstabelle möglich.

Annotationen im Quellcode

Jede Annotation im Quellcode bezieht sich auf ein Programmelement oder einen Block solcher Elemente. Annotationen können als Sprachkonstrukt (vlg. [JSR175]) oder als Kommentar (vgl. [XDoclet], [LC03] und [Me06]) realisiert werden. Die Entscheidung für Annotationen als wohldefinierte Kommentare erleichtert die Arbeit, da dadurch keinesfalls das Verhalten des annotierten Codes beeinflusst wird und alle aktuellen Werkzeuge, Compiler und Interpreter genutzt werden können. Javadoc-Annotationen passen gut in das Konzept der Java-Plattform. In früheren Java-Versionen kamen sie als individuelle Erweiterung zur breiten Anwendung (vgl. [XDoclet]).

Zur Unterscheidung einer Annotation von einem Kommentar beginnt jede Annotation mit dem @-Symbol als Anfangsmarker. In unserem Ansatz werden syntaktische und semantische Typen von Annotationen durch *@syntactic* und *@semantic* unterschieden (Abb. 2b).

Gültigkeit von Annotationen

Eine Annotation kann prinzipiell oberhalb eines jeden Programmelements (Deklaration, Einzelanweisung, Methode, Block) eingesetzt werden und besitzt mit dieser Festlegung exakt einen Gültigkeitsbereich. Eine Annotationsdefinition dagegen kann potenziell mehrere Gültigkeitsbereiche erlauben, zumindest aber einen. Möglich sind: Anwendung, Paket, Klasse, innere Klasse, Methode, Anweisung und Felddefinition. Für jedes dieser Programmelemente kann eine syntaktische bzw. semantische Annotation formuliert werden. In Abb. 2b ist der angewandte Gültigkeitsbereich der Annotation die private Deklaration einer Variablen. Demgemäß ist die Annotation MID 1234 im Kontext der Abb. 2b sinnvoll. Sie ist auch sinnvoll für eine statische Deklaration, z.B. `private static String surname`. Diese statische Deklaration hat einen anderen AST als die analoge nichtstatische Deklaration. Daher ist eine leistungsfähige Definition des Gültigkeitsbereichs von Annotationen essentiell. Wie in [Me06] vorgeschlagen, können Isomorphe für die Bereiche von Annotationen verwendet werden. Ein Isomorph ist ein Programmelement, welches als Äquivalent zu einem anderen angesehen wird. Beispielsweise kann eine Methode mit deklarierter Ausnahme als isomorph zu einer anderen Nicht-Exception-Methode erscheinen, wenn es um die Annotation geht. In diesem Fall werden beide Methodensignaturen hinsichtlich derer ASTs als äquivalent betrachtet.

Für jede Annotation kann zunächst der Gültigkeitsbereich exemplarisch („by example“) identifiziert werden. Danach wird für jedes erkannte Beispiel eine mögliche isomorphe Repräsentation gesucht. Auf diese Weise können die Gültigkeitsbereiche von Annotationen bestimmt werden. Die wahlweise Gruppierung von Programmkonstrukten kann durch die Technik *Programming by Example* (vgl. [Li01]) unterstützt werden. Wir haben eine größere Anzahl von Gruppen von Operatoren und Anweisungs- bzw. Deklarationstypen definiert, die zu vielen Permutationen führen können. Diese Typen umfassen u.a.:

- **Interface-Typen**: Marker vs. normale Schnittstellen
- **Methodentypen**: Konstruktor vs. Instanzmethode vs. statische Methode;
- **Sichtbarkeiten**: private, protected., package, public
- **Klassentypen**: abstrakte vs. konkrete; Position in der Hierarchie; Sichtbarkeit; Funktion einer Klasse: Datencontainer vs. Hilfsklasse vs. Persistenzklasse
- **Anweisungstypen**: Zuweisung: ja/nein; Methodenaufruf vs. Konstruktion vs. Berechnung vs. Schlüsselwort usw.; Block vs. Einzelanweisung
- **Operatortypen**: Arithmetisch vs. Boolescher Vergleich vs. Boolesche Bedingung vs. Logisch vs. Bit-Verschiebung usw.

Für jede dieser Gruppen kann ein Isomorph separat bestimmt werden. Ein definiertes Isomorph erhält einen eigenen Namen (eine Art ID), so dass es aus einer anderen Annotationsdefinition referenziert werden kann. Jeder einzelne Name wird mit einer Routine verbunden, die für ein konkretes Programmelement ermittelt, ob es zum Gültigkeitsbereich gehört oder nicht. Durch die Möglichkeit, Gültigkeitsbereiche mit Hilfe Boolescher Funktionen zu gruppieren, muss ein Gültigkeitsbereich nicht durch ein einzelnes Isomorph definiert werden, es reicht wenn eine Boolesche Kombination mehrerer Isomorphe dafür existiert.

Damit eine Gruppe von Programmelementen für eine Annotation spezifiziert werden kann, wurde eine weitere Annotation eingeführt, der Endemarker (siehe Abb. 2c). Sie muss keine HID besitzen, da sie selbstsprechend ist. Der Endemarker referenziert die korrespondierende Annotation durch die konkrete ID.

3.2 Untertypen von Annotationen

Syntaktische und semantische Annotationen können auf unterschiedliche Aspekte verweisen, die als Untertyp der Annotation erscheinen, z.B. *@syntactic diagnosis* oder *@semantic intention.* Für syntaktische Annotationen kann es folgende Typen geben:

- **Diagnose**; z.B. Identifikation eines Isomorphs
- **Empfehlung**: z.B. Angabe einer Refactoring-Operation
- **Anforderung**: z.B. Aufforderung zu einer Refactoring-Operation

Semantische Informationen beschreiben unterschiedliche Aspekte des annotierten Codefragmentes:

- **Zweck**: die Bedeutung des Codefragmentes

- **Anforderung**: zur Anwendung einer Refactoring-Operation oder eines Entwurfsmusters
- **Problem**: verweist auf ein im Code enthaltenes Problem
- **Schlussfolgerung**: Anweisung, z.B. als Ergebnis einer AST-Analyse
- **Diagnose**: Anweisung zum Kontext
- **Empfehlung**: kann auf eine Diagnose erfolgen

Um zu zeigen, welche Entität (Werkzeug, Entwickler) eine Annotation eingeführt hat, sollte der Name der Entität angeben werden. So könnte etwa das Werkzeug [PMD] durch die Zeichenkette *tool-pmd* identifiziert werden, respektive ein Entwickler z.B. mit *developer-peter*. Solche Informationen sind bei Diagnosen und Empfehlungen hilfreich. Ein zusätzliches Freitextfeld ermöglicht einem Annotator, Kommentare abzugeben, weshalb oder nach welchen Regeln eine Annotation eingefügt wurde.

Eine Annotation kann ihre Gültigkeit für ein Codefragment über die Zeit verlieren, wenn dieses verändert wird. Um das erkennbar zu gestalten, wird für das annotierte Programmelement ein Hashcode-Wert errechnet, der in der Annotation vermerkt wird. Eine IDE, die Annotationen unterstützt, wird so in der Lage sein, Programmänderungen je nach Güte des Hashwertes zu erkennen. Indirekte Abhängigkeiten sind diesbezüglich hingegen schwieriger auszuwerten. Zusammenfassend kann eine Annotation folgende Elemente enthalten:

- Anfangsmarker: *@syntactic <subtype>*, *@semantic <subtype>*
- Endemarker: *@end-marker*,
- ID der Annotation,
- Hashcode der annotierten Elemente,
- ID des Annotators,
- optionaler Kommentar des Annotators,
- MID: Maschinenlesbare Teil der Annotation,
- HID: Menschenlesbare Teil der Annotation.

3.3 Definition von Annotationen

Annotationen müssen definiert werden, bevor sie zum Quellcode hinzugefügt werden können. Möglich ist die Definition von Annotationen

1. durch Analyse von Aufzeichnungen oder Anforderungen,
2. durch Transformation eines Beispielcodes,
3. durch die Wiederverwendung von Definitionen in einem öffentlichen Repository oder einem Serviceprovider.

Die erste Alternative ist geeignet, wenn im Voraus bekannt ist, welche Annotationen im Kontext eines möglichen Entwurfsmusters benötigt werden. Das kann etwa die Intention eines Entwurfsmusters betreffen. Die zweite Alternative ist der komplexeste Weg, um Annotationen für ein gegebenes Muster zu finden. Ausgangspunkt ist ein passendes Fragment eines Beispielcodes. Die Anwendung des Musters erfolgt durch eine Folge

manueller Transformationen, wobei jede Transformation zu einer neuen Annotation für das Muster führen kann. Nach jedem Transformationsschritt ist der Entwickler gefordert, zu entscheiden, ob das Einfügen einer weiteren Eine Annotation wird dann als notwendig erachtet, wenn sie Informationen betrifft, die nicht im AST-Code enthalten sind, aber benötigt werden, um Schlussfolgerungen zur Anwendbarkeit eines Musters ziehen zu können. Das betrifft ganz speziell Entwurfsabsichten, die durch Annotationen ausgedrückt werden können. Die dritte Alternative ist äquivalent zur zweiten, mit dem Unterschied, dass eine externe Quelle für die Auswahl und Wiederverwendung von Annotationen existiert. Um einen Quellcode nicht mit Annotationen zu überschwemmen, sollte gelten, so viel wie nötig, so wenig wie möglich. Die Information muss jedoch ausreichend sein, um sowohl erfahrene Entwickler als auch Werkzeuge zu befähigen, Muster auswählen und anzuwenden zu können. Um die Informationsmenge zu bestimmen, die dieser Anforderung genügt, muss die technische Definition des Musters (*pattern template*) betrachtet werden (vgl. [Me06]). In ausgewählten Fällen können Annotationen automatisch durch Werkzeuge erzeugt werden. Beispielsweise kann ein Analysewerkzeug *for*- oder *while*-Schleifen identifizieren, für die sich die Anwendung des *Iterator*-Musters anbietet. Viele Muster sind jedoch nicht so trivial wie *Iterator*. Durch Werkzeuge im Quelltext angebrachte Annotationen geben Empfehlungen für die Anwendung einer Refactoring-Operation oder eines Entwurfsmusters. Die Entscheidung der Anwendung einer Operation verbleibt letztendlich beim Entwickler.

4 Annotationen für Refactoring-Operationen und Entwurfsmuster

In Analogie zu [JSR250], wo allgemeine Annotationen für Java empfohlen werden, werden in diesem Abschnitt Annotationen für Refactoring-Operationen, einschließlich von Entwurfsmustern, vorgestellt. Syntaktische Annotationen und enthalten in der Regel Ergebnisse aus AST-Analysen. Die im Folgenden gelisteten Beispiele (Auszug) ergaben sich aus der Untersuchung der Ausgaberesultate des Werkzeuges [PMD], der Refactoring-Operationen, die in [Fo99] benannt werden, der Entwurfsmuster aus [Ga95] und weiterer Publikationen, die sich auf Refactoring-Operationen beziehen.

- **Diagnose**: Variable / Methodenparameter / Methode *x* wird nicht verwendet (vgl. [PMD])
- **Diagnose**: Klasse x besitzt nur statische Methoden (vgl. [PMD])
- **Diagnose**: das nächste Element erzeugt eine Instanz der Klasse x (vgl. [Sa03])
- **Diagnose**: der folgende Codeblock ist ein Isomorph zu *x* (vgl. [Sa03])
- **Diagnose**: die Methodensignatur enthält zu viele Parameter

Semantische Annotationen sollten Diagnosen und/oder Empfehlungen enthalten (siehe Abb. 3). Für einfache Refactoring-Operationen ist es in vielen Fällen ausreichend, eine Empfehlung dafür abzugeben, eine Operation auf ein einzelnes Programmelement (z.B. auf eine Klasse) anzuwenden. Um ein Muster anzuwenden sind dagegen meistens mehrere Operationen auszuführen.

```
/*
 * @semantic suggestion
 * ... header fields and MID skipped
 * HID: introduce Iterator for variable "list"
 */
java.util.List list = buildList();
for(int i=0;i<list.size();i++) {...}
```

Abbildung 3: Annotation einer semantischen Empfehlung

Die folgenden Beispiele zeigen Annotationsmöglichkeiten für einfache Refactoring-Operationen:

- **Diagnose**: der Name der Variable/Klasse/Methode x ist zu lang (vgl. [PMD])
- **Diagnose**: die Methode ist zu lang (vgl. [Fo99])
- **Empfehlung**: verschieben einer Methode in die Superklasse (vgl. [Fo99])
- **Diagnose**: Methode x aus Klasse y nicht in Klasse z verwendet (vgl. [Fo99])

4.1 Annotationen für Entwurfsmuster

Die Anwendung von Entwurfsmuster benötigt semantische Annotationen, die neben der Empfehlung zur Anwendung auch eine konkrete Musterbeschreibung, Diagnoseinformationen und die Bedingungen für eine Musteranwendung angegeben. Die vorgeschlagenen Empfehlungen zu Entwurfsmustern orientieren sich an der Musterbeschreibung in [Ga95].

- *Observer*, *Mediator*: **Anforderung**: entkopple Kollaborationen in Klasse x und y
- *Flyweight*, *Proxy*: **Diagnose**: die Erzeugung einer Instanz der Klasse x verbraucht zu viel Speicher
- *Proxy*: **Anforderung**: füge Caching/Protokollierung/Zugriffskontrolle zur Klasse x hinzu, ohne diese zu verändern
- *Interpreter*: **Diagnose**: die interpretierte Grammatik ist simpel
- *Interpreter*: **Diagnose**: die Performance der interpretierten Grammatik ist unkritisch
- *Interpreter*: **Anforderung**: liefere ein Konzept zum Aufbau eines Grammatikinterpreters

Das letzte Beispiel für *Interpreter* ist ein Vertreter dafür, dass sich Annotationen manchmal auf Aussagen in anderen Annotationen stützen. Im *Interpreter*-Fall muss nämlich herausgefunden werden, welche Klasse ein Literal, einen wiederholbaren Ausdruck u.s.w. repräsentiert. Es wäre einfacher, eine Annotation zu definieren und in der Klasse anzubringen, die ein Literal in einem bestimmten Kontext repräsentiert (hier ist der Kontext die zu interpretierende Grammatik). So kann die Information des Zwecks der Klasse sofort ausgelesen werden anstatt dass ein Tool dies zuerst analysieren müsste.

4.2 Auswahl von Refactoring-Operationen

Mit Annotationen können *candidate spots* (vgl. [Sa03]), d.h. Identifikatoren für Refactoring-Operationen gesetzt werden. Dazu ist es notwendig, Annotationen in Korrelation zu Musterdefinitionen zu setzen. Für die Unterstützung der Entscheidung für oder gegen eine Refactoring-Operation werden *candidate spots* mit Vorbedingungen ergänzt. Mit Annotationen können auch erfüllte oder verletzte Vorbedingungen für die

Anwendung einer Refactoring-Operation, und im Besonderen eines Entwurfsmusters, ausgedrückt werden. Eine Vorbedingung könnte eine semantische Einschränkung sein, z.B. eine strikte Objekt Identität ist nicht gefordert (vgl. *Flyweight* [Ga95]), oder eine syntaktische, wie eine Referenz von Klasse *x* zu Klasse *y*.

In [Me06] wird die Idee korrespondierender Elemente zwischen Quellcode und Musterdefinitionen im Detail beschrieben. Um die Musterdefinition zu extrahieren, müssen sowohl die syntaktischen als auch die semantischen Einschränkungen bestimmt werden. Die textuelle, nicht durch eine Maschine verarbeitbare Beschreibung aus [Ga95] kann durch Annotationen in einer maschinenlesbaren Form abgelegt werden. Diese Transformation wird durch das Konzept der zusammengesetzten Annotationen (HID und MID) stark vereinfacht.

5 Schlussfolgerungen und Ausblick

Refactoring-Operationen gewinnbringend zu nutzen, bedeutet deren sorgfältige Auswahl und Anwendung. Dafür muss ein gegebenes Codefragment sorgfältig analysiert werden. Die heute verfügbaren Werkzeuge unterstützen nur unbefriedigend die Auswahl einer Refactoring-Operation und ihre kontextsensible Anwendung. Existierende Analysewerkzeuge für Quellcode sind in der Lage, Informationen zu ermitteln, die für das Selektieren passender Refactoring-Operationen zur Verbesserung des Quellcodes relevant sind. Andererseits sind Werkzeuge, die eine Refactoring-Operation ausführen, nicht in der Lage, diese Analyseresultate zu verstehen.

Das Konzept der zusammengesetzten Annotationen ermöglicht die Etablierung von syntaktischen und semantischen Informationen im Quellcode, die zum einen durch Werkzeuge ausgetauscht, und somit zur Auswahl und Ausführung von Refactoring-Operationen verwendet werden können. Zum anderen werden die Entwickler in die Lage versetzt, Statements zu einem durch sie evaluierten Code darzulegen. Durch die Kopplung eines maschinenverarbeitbaren (MID) mit einem durch den Menschen lesbaren (HID) Teil werden Probleme mit der Definition von Annotationen vermieden. Die Unterscheidung von syntaktischen und semantischen Typen und deren Untertypen dienen der besonderen Betonung unterschiedlicher Aspekte und Aussagen. Ein Schwerpunkt wurde auch darauf gelegt, die Validität von Notationen über die Zeit, z.B. bei einer Codeänderung zu bewahren bzw. zu prüfen. Durch Hinzufügen eines Hashcodes an eine Notation können relevante Veränderungen ermittelt und erkannt werden. Für das Parsen von Mustervorlagen wurde ein Prototyp entwickelt, der Annotationen im Quellcode untersucht und ASTs durch Auswerten des Quelltexts erzeugt.

In künftigen Arbeiten müssen existierende Werkzeuge wie [PMD] und AST-Analysatoren für die Unterstützung der hier beschriebenen Annotationen weiterentwickelt werden. Eine Menge relevanter Annotationen muss standardisiert werden, um den Austausch zwischen Werkzeugen zu ermöglichen. Die Methode, die den Entwickler bei der Definition von Annotationen für Refactoring-Operationen unterstützt, einschließlich der Vorbedingungen und Transformationen, muss weiter konkretisiert

werden. Ähnlichkeitsbeziehungen zwischen unterschiedlichen MIDs sind wichtig, um s.g. *near misses* anwendbarer Operatoren erkennen zu können. Wir schlagen vor, später für standardisierte Annotationen eine Ähnlichkeitsmatrix aufzustellen Ein Algorithmus wie der in [Me06] beschriebene hat dann die Möglichkeit, *near misses* zu erkennen. Die Einheit aus Annotationen und Beschreibung der Entwurfsmuster kann zur Detektion angewandter Muster verwendet werden. Einige Muster wie z.B. *Façade* [Ga95] könnten ohne explizite Information in Annotationen kaum gefunden werden, da sie schwer oder gar nicht von gewöhnlichem Quellcode unterschieden werden können.

Literaturverzeichnis

[Ed02] Eden, A. H.: LePUS: A Visual Formalism for Object-Oriented Architectures. The 6th World Conference on Integrated Design and Process Technology. Pasadena, CA, 26.-30. Juni 2002.

[Fo99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[Ga95] Gamma, E. et al.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Jackpot] Jackpot. http://jackpot.netbeans.org/index.html.

[JSR175] Java Specification Request 175: A Metadata Facility for the JavaTM Programming Language. http://www.jcp.org/en/jsr/detail?id=175.

[JSR250] Java Specification Request 250: Common Annotations for the JavaTM Platform. http://www.jcp.org/en/jsr/detail?id=250.

[KR06] Krahn, H.; Rumpe, B.: Towards Enabling Architectural Refactorings through Source Code Annotations. In: Proceedings der Modellierung 2006. 22.-24. März 2006, Innsbruck. GI-Edition - Lecture Notes in Informatics, LNI P-82, ISBN 3-88579-176-5, 2006.

[LC03] Leavens, G. T.; Cheon, Y.: Design by Contract with JML. http://www.jmlspecs.org, 2003.

[Li01] Lieberman, H.: Your Wish Is My Command: Programming By Example. Morgan Kaufman, 2001.

[Me06] Meffert, K.: Supporting Design Patterns with Annotations. *ecbs*, pp. 437-445, 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), 2006.

[Ny03] Nyberg, E. et al.: The JAVELIN Question-Answering System at TREC 2003: A Multi-Strategy Approach with Dynamic Planning, Proceedings of TREC 12, Nov. 2003.

[PMD] PMD. http://pmd.sourceforge.net.

[Sa03] Sang-Uk, J.: An Approach to Automatically Identifying Design Structure for Applying Design Pattern. Korea, 2003.

[TCN03] Taibi, T.; Chek Ling Ngo, D.: Formal Specification of Design Patterns – A Balanced Approach. In: Journal of Object Technology, vol. 2, no. 4, Juli-August 2003, S. 127-140.

[Xdoclet] XDoclet. http://xdoclet.sourceforge.net.