Unterstützung von Entwurfsmustern im Quelltext durch bedeutungsorientierte Dokumentation

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

vorgelegt der Fakultät für Informatik und Automatisierung

der Technischen Universität Ilmenau

von Diplom-Informatiker Klaus Meffert

geboren am 12. Juli 1974 in Offenbach am Main

Vorgelegt am 19. November 2008

Tag der wissenschaftlichen Aussprache: 12. August 2009

Gutachter:

1) Prof. Dr.-Ing. habil. Ilka Philippow
2) Prof. Dr. Wilhelm Rossak
3) Privatdozent Dr.-Ing. habil. Jürgen Nützel

urn: nbn:de:gbv:ilm1-2009000244

Vorwort

Diese Arbeit entstand unter Betreuung von Frau Prof. Dr.-Ing. habil. Ilka Philippow. Als externer Doktorand danke ich Frau Prof. Philippow besonders für das entgegen gebrachte Vertrauen, das es mir überhaupt erst ermöglicht hat, diese Arbeit zu beginnen. Weiterhin bin ich ihr für die stets konstruktive und hilfreiche Kritik an meiner Arbeit und die zahlreichen Anregungen dankbar, die mir das Strukturieren und Glätten der Arbeit erleichtert haben.

Herrn Prof. Dr. Wilhelm Rossak und Herrn Privatdozent Dr.-Ing. habil. Jürgen Nützel danke ich für die interessierten, weiter führenden Fragen und konstruktiven Anregungen zur Arbeit und die Möglichkeit zum persönlichen Gespräch.

Die erwiesene Freundlichkeit und Hilfsbereitschaft der mit mir während der Arbeit in Kontakt getretenen Mitarbeiter der Universität in Ilmenau, die mir schon während meines Diplomstudiums positiv aufgefallen sind, möchte ich hier ausdrücklich hervorheben.

Zu persönlichem Dank bin ich meinem engeren sozialen Umfeld verpflichtet, das meine zeitlich eingeschränkte Verfügbarkeit über mehrere Jahre in motivierender Weise gestattet hat.

Klaus Meffert
Mail: kontakt@klaus-meffert.de

Kurzfassung

Die Bedeutung erweiterbarer und wartbarer Software wächst mit zunehmender Komplexität des Technologiemarktes. Es gibt zahlreiche Ansätze, Entwickler bei der Erstellung derartiger Software zu unterstützen. Ein Ansatz sind Entwurfsmuster. Sie helfen bei der Reduzierung des Problems des Architekturzerfalls, der durch naturgemäß steigende Entropie und Komplexität während des Software-Entwicklungsprozesses entsteht. Die Anzahl der dokumentierten Entwurfsmuster nimmt durch neue Publikationen stetig zu. Die Auswahl geeigneter Muster für einen Anwendungsfall wird dadurch immer schwieriger. Weiterhin wird die in der Praxis nur bedingt verfügbare und unzureichende Unterstützung für Entwickler bei der Arbeit mit Entwurfsmustern der intensiven Beschäftigung in der Forschung mit diesem Thema nicht gerecht.
Diese Arbeit leistet einen Beitrag zur besseren Unterstützung des Entwicklers bei der Arbeit mit Mustern jeglicher Granularität. Die formale Dokumentation dieser wird thematisiert, um anhand dessen die Anwendbarkeit eines Musters für einen objektorientierten Quelltext festzustellen und das Muster mit Hilfe von Quelltexttransformationen werkzeuggestützt anzuwenden. Sie stellt eine Methodik bereit für das Anreichern von Programmtext mit semantischen Informationen, die bei der Musterselektion und -anwendung dienlich sind. Die vorgestellte Methodik ist für den Einsatz in Quelltextrezensionen geeignet, deckt also die Forward- und Reengineering-Phase bei der Software-Entwicklung ab.

Schlagwörter: Entwurfsmuster, Annotationen, Quelltext, Transformation, Musterdokumentation, Musterselektion, Musteranwendung, Semantik

Abstract

The significance of extendable and maintainable software rises with the growing complexity of technologies. Numerous approaches support developers to create software with these characteristics. One such approach is given with design patterns. They help in reducing the problem of architectural decay, which comes naturally from escalating entropy and complexity during the process of software development. The number of to-date documented design patterns raises through new publications steadily. Thus, the selection of applicable patterns for a given use case becomes more difficult. Besides, the limited support for developers when working with design patterns in practice does not reflect the massive effort put into this issue by academic research.
This work contributes to a better support of patterns with arbitrary granularity in the developer's daily work. Focus lies on the formal documentation of patterns in order to determine their applicability for a given object-oriented source code and to apply them on a tool-basis with help of source code transformations. This work provides a method for enriching source code with semantic information that is seen as valuable for the processes of selecting and applying a pattern. The present work is suited for source code reviews and thus covers the forward and reengineering phase during software development.

Key words: Design patterns, annotations, source code, transformation, pattern documentation, pattern selection, pattern application, semantics

Inhaltsverzeichnis

KAPITEL 1

1 Einleitung

„Während die Summe dessen, was man gefunden und durch den Druck niedergelegt hat, ins Unermeßliche steigt, wird das, was der einzelne übersieht und weiß, relativ immer geringer."

Max Born, Physiker

Die Einleitung ist aufgeteilt in drei Teile. Zuerst folgt eine Einführung in die Thematik der Entwurfsmuster mit Problemdarstellung. Der daran anschließende Absatz motiviert die Lösung des dargestellten Problems. Im Anschluss folgt eine Gliederung der Arbeit, die einen neuen Ansatz zur Auflösung des skizzierten Problems aufzeigt.

1.1 Grundlagen und Problemdarstellung

Zur effizienten objektorientierten Entwicklung eines Software-Produktes sind umfangreiche technische Kenntnisse und Erfahrungen Voraussetzung. Vorgehensmodelle, formale und semiformale Techniken sowie Werkzeuge erleichtern den Entwicklungsprozess. Konkret geschieht dies beispielsweise durch Refactoring-Werkzeuge, visuell-unterstützte Gestaltung von Applikationen, kontextuelle Hilfesysteme oder etwa durch das Paradigma der modellgetriebenen Entwicklung.
Eine Möglichkeit, im Prozess der Software-Entwicklung von der Entwurfserfahrung anderer zu profitieren, ist der Einsatz von Entwurfsmustern. Der Begriff ist vom englischen Begriff *Design Patterns* abgeleitet. Oft verwendet man verkürzt statt des Begriffes *Entwurfsmuster* einfach nur die allgemeinere Bezeichnung *Muster*.
Die folgende Definition für den Begriff *Entwurfsmuster* entspricht der gängigen Lehrmeinung (vgl. auch [Gamma+ 1995, S. 4] oder [Balzert 1996, S. 874]):

Definition: Entwurfsmuster (Kontext: Informatik)

> Ein Entwurfsmuster ist eine (bewährte) Lösung für ein allgemeines Entwurfsproblem in einem bestimmten Kontext.

Entwurfsmuster ermöglichen die Erstellung einer Software-Architektur, welche Evolutionsprozesse mit nur geringem Aufwand erleichtert und unterstützt (zur Wartung von Software vgl. [Balzert 1996, S. 972], zur Migration siehe [Balzert 1998, ab. S. 212]). Anwendungsgebiet ist sowohl die Forward- als auch die Reengineering-Phase. Forward- und Reengineering-Phase können bzgl. der Musterselektion und -anwendung als gleichwertig angesehen werden, sofern der Anteil der Refactoring-Tätigkeiten vergleichbar ist. Letzt genannte Bedingung tritt in der Realität nach Erfahrung des Autors in kommerziellen Software-Projekten häufiger ein. Hauptzielgruppe von Entwurfsmustern sind Software-Entwickler und Software-Architekten.

Entwurfsmuster sind keine Regeln, die etwas definitiv festlegen (vgl. [Alexander+ 1995, S. X und S. XIV]). Sie machen Vorgaben, die eine flexiblere Software-Architektur ermöglichen und hervorbringen sollen. Die Vorgaben zielen für gewöhnlich auf die nichtfunktionalen Aspekte eines Programms. Aufgrund dieser Vorgaben wird ein bestehender Quelltext so transformiert, dass er eine Form annimmt, die als günstig angesehen wird. Eine flexible Software-Architektur lässt zukünftige Änderungen und Erweiterungen mit weniger Aufwand oder weniger Risiko zu (vgl. [Alexander+ 1995, S. XVI]). Sie ist also leichter wartbar und reduziert das Problem des Architekturzerfalls. Um dieses Ziel zu erreichen, enthält ein Muster idealerweise die Schnittmenge aller wesentlichen Aspekte, die zur Lösung eines für das Muster charakteristischen Problems nicht umgangen werden können (siehe

auch [Alexander+ 1995, S. XIV]). Muster dienen zudem der Dokumentation von Software, sie drücken die Intention des Entwicklers aus.[1]

Besonders in der objektorientierten Entwicklung von Software haben sich Entwurfsmuster in der Informatik etabliert. Dieser Impuls ging maßgeblich von [Gamma+ 1995] aus, wo 23 Muster vorgestellt wurden. Mehrere Jahre zuvor wurde die Idee der Muster für die Architektur durch den Architekten Christopher Alexander populär (vgl. [Alexander+ 1995], [Dols 1978], [Alexander 1979]). Insbesondere wird nicht nur die Grundidee Alexanders an sich in der Informatik adaptiert. Eine Quelle wie [Coram+ 2003] zeigt die deutliche Anlehnung an die Methodik, die im Rahmen der Muster-Sprache für die Architektur durch Alexander entwickelt wurde.

Trotz der hohen Beachtung von Entwurfsmustern in Forschung und Lehre geht der Autor davon aus, dass der ganz überwiegende Teil der Software-Entwickler Entwurfsmuster nicht einsetzt[2]. Nach Informationslage des Autors lässt sich folgende These für Entwickler aufstellen, die mit objektorientierten Sprachen arbeiten:

- 90% aller Entwickler kennt das Konzept der Entwurfsmuster nicht oder wendet Muster nur in nicht nennenswertem Umfang an.
- 10% aller Entwickler wenden Entwurfsmuster nennenswert an.
- Von diesen 10% wenden wiederum nur 10% - oder insgesamt 1% - komplexere oder zusammengesetzte Muster an oder wandeln bestehende Muster ab.

Danach besteht eine deutliche Diskrepanz zwischen dem Wunschzustand, in dem alle Entwickler Entwurfsmuster beherrschen oder weiterentwickeln und der Realität bei der Musteranwendung.

Hauptgrund für die geringe Durchsetzung von Entwurfsmustern in der Praxis ist der hohe Einarbeitungsaufwand der notwendig ist, bis Entwurfsmuster effektiv eingesetzt werden können (vgl. [Vlissides 1999]). Es existiert also eine Diskrepanz zwischen der Rolle von Entwurfsmustern in der Theorie und in der Praxis. Zur Überwindung dieser Kluft ist eine bessere Werkzeugunterstützung für den Entwickler bei der Auswahl und Anwendung von Mustern wünschenswert.
Es ist deshalb entscheidend, die bisherigen Tätigkeiten des Entwicklers bei der Entwurfsmusteranwendung und -auswahl durch geeignete, automatisierbare Verfahren zu unterstützen oder nach Möglichkeit zu ersetzen. Daraus ergibt sich die Gelegenheit, Entwurfsmuster auch über die bisherige Unterstützung in generativen und modellgetriebenen Verfahren hinaus verstärkt in die kommerzielle Software-Entwicklung einfließen zu lassen. Der folgende Absatz 1.1.1 beschreibt ein heute gängiges Verfahren zur werkzeuggestützten Selektion und Anwendung von Entwurfsmustern. Im Anschluss folgt die Auswertung eines Fragebogens zur Arbeit mit Entwurfsmustern. Hiermit soll das Bild zum Thema Entwurfsmuster abgerundet werden.

1.1.1 Heutige Werkzeugunterstützung bei der Musteranwendung

In diesem Abschnitt wird ein heutzutage verbreiteter Weg zur Definition, Auswahl und Anwendung von Entwurfsmustern gezeigt. Beispielhaft wurde das Werkzeug *objectiF* [objectiF 2004] ausgewählt. Es entspricht in etwa dem, was auch andere Programme wie *Rational Rose*, *Together* oder *PSE*

[1] Der Aspekt der Dokumentation soll hier noch einmal besonders betont werden. Betrachtet man die Philosophie des *Test-driven Development* (kurz: TDD), erkennt man in der Fähigkeit, bestehende Programmstrukturen zu dokumentieren, eine Gemeinsamkeit zwischen den qualitätsfördernden Maßnahmen TDD und Entwurfsmustern (vgl. [Beck 2002] und [Fraikin+ 2004, S. 29]).

[2] Diese Erkenntnis hat sich aus Antworten auf die Beiträge [Meffert 2004a] und [Meffert 2004b], persönlichen Gesprächen mit Arbeitskollegen, Entwicklern und Entwurfsmusterspezialisten, dem in Abschnitt 1.1.3 genannten Fragebogen sowie Beobachtungen des Autors in der kommerziellen Software-Entwicklung ergeben.

[Mayr-Kern+ 2002] leisten. All diese Programme bieten im Hinblick auf die Arbeit mit Entwurfsmustern nicht wesentlich mehr als im Folgenden dargestellt wird.
objectiF ist ein Modellierungswerkzeug auf Basis der UML. Der Entwickler hat die Möglichkeit, ein Entwurfsmuster als Entität erster Klasse zu behandeln. Dementsprechend gibt es konzeptionell keinen Unterschied zwischen Entwurfsmustern und einfachen Klassen und Klassenansammlungen. Man erstellt also die Klassen des Entwurfsmusters und legt sie im Repository als definierte Mustervorlage ab. Die Mustervorlage enthält das Grundgerüst des Musters. Die Erstellung einer solchen Vorlage wird als Musterdefinition bezeichnet. Im Weiteren wird anstelle dessen der aus Sicht des Autors präzisere Terminus *formale Musterdokumentation* verwendet. In *objectiF* besteht die formale Musterdokumentation aus statischen Klassen und Schnittstellen sowie unvollständigen dynamischen Teilen. Dynamische Teile sind im von *objectiF* angebotenen Prozess solche, die erst durch Anwendung des Musters und durch manuelle Ausprogrammierung mit Leben erfüllt werden. Beispielsweise kann die Logik einer Methode einer Musterklasse von *objectiF* nicht vorgegeben werden, sofern die Logik abhängig vom Quelltext ist, auf den das Muster angewandt werden soll. Weiterhin kann *objectiF* keine Teile des Ausgangsquelltextes, auf den ein Muster anzuwenden ist, im Rahmen einer Musteranwendung modifizieren. Beispielsweise kann kein Aufruf einer Methode, die durch Anwendung des Musters neu hinzukommt, in den Ausgangsquelltext eingefügt werden.

Die folgenden Abbildungen zeigen die Vorgehensweise der Musteranwendung mit *objectiF* und wurden durch Screenshots bei der Arbeit mit *objectiF* erzeugt. Abbildung 1 stellt einleitend die Auswahl eines Kontextes (einer Menge von Klassen in Form eines Java-Paketes) dar, auf denen ein Muster angewendet werden soll.

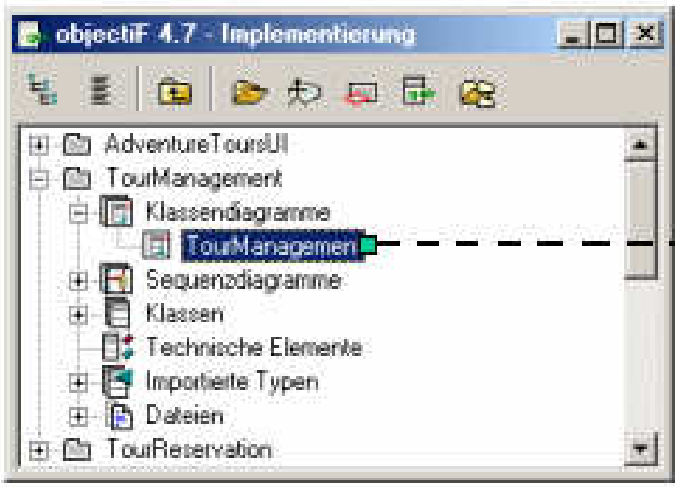

Abbildung 1: Auswahl eines Musterkontextes mit dem Werkzeug *objectiF*.

In der Abbildung 1 ist die Auswahl von Klassen, hier in Form eines Klassendiagramms, dargestellt. Das zugehörige Klassendiagramm verbirgt sich hinter dem Begriff »TourManagement«. Es beinhaltet eine beliebige Menge von Klassen, die zu einander in Beziehung stehen.
Der nächste Schritt ist die manuelle Auswahl des anzuwendenden Musters. Dazu zeigt *objectiF* eine Liste der im Repository gespeicherten Muster an. Eine knappe Beschreibung zu jedem Muster hilft dem Anwender bei der Orientierung (Abbildung 2).

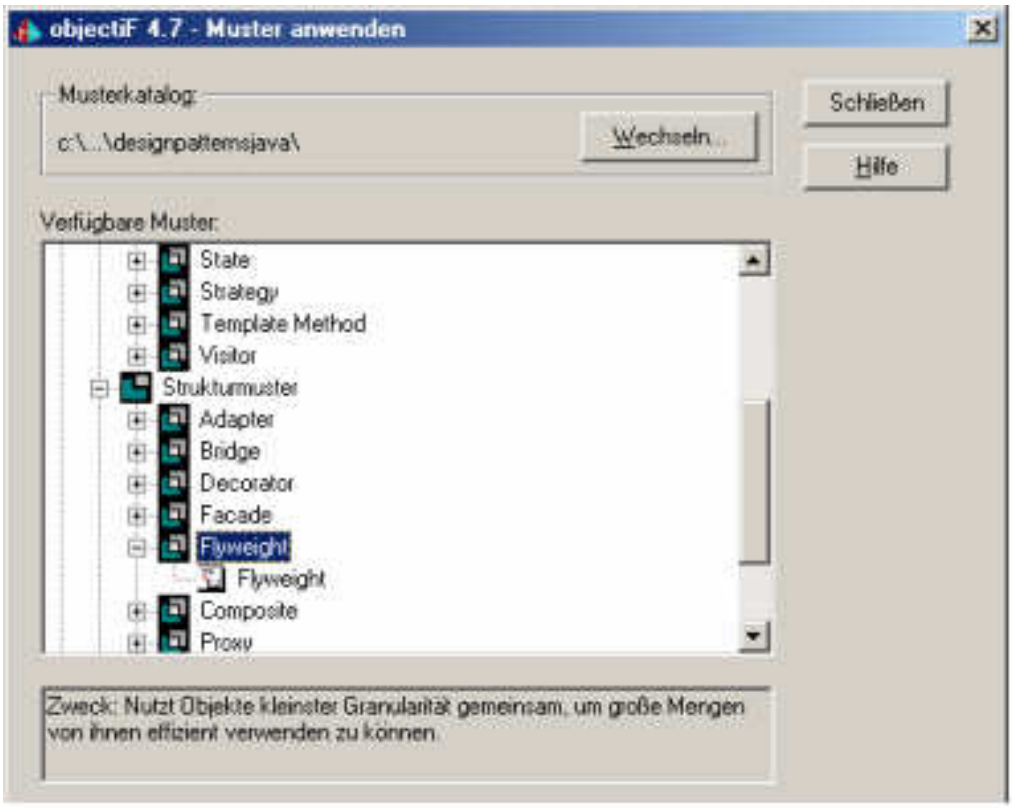

Abbildung 2: Auswahl eines anzuwendenden Musters mit *objectiF*.

Danach werden die im Muster enthaltenen Klassen (Rollen) den Klassen des Kontextes mittels einer Eins-zu-eins-Zuordnung zugewiesen (Abbildung 3).

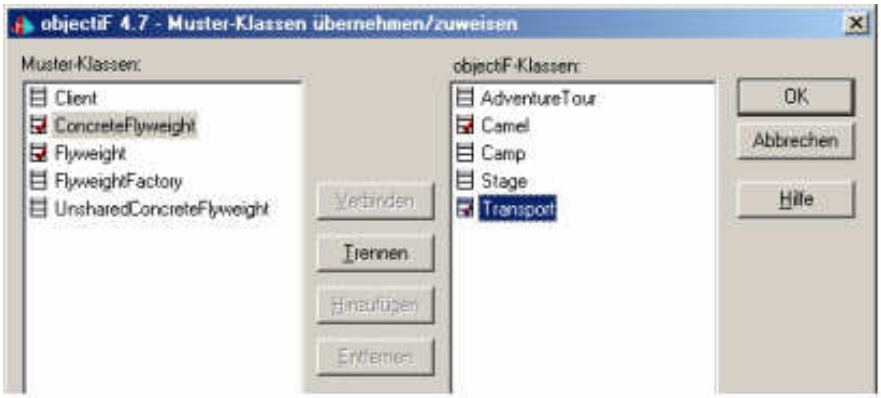

Abbildung 3: Zuordnen von Musterklassen mit zuvor selektierten Klassen mit *objectiF*.

Die Musterklassen sind links in der Abbildung, die Klassen des Kontextes rechts dargestellt. Nach der Zuordnung der Klassen wird der Mustercode mit dem Programmcode durch generative Techniken verwoben. Dieser Vorgang wird auch als Einweben eines Entwurfsmusters bezeichnet.

Definition: Einweben eines Entwurfsmusters

> Das Einweben eines Entwurfsmusters ist der Prozess, bei dem ein für die Anwendung ausgewähltes Muster, welches als Rahmen in Form von statischen und dynamischen Teilen vorliegt, mit dem bestimmten Programmkontext verknüpft wird. Die statischen Teile des Musters ergänzen den ursprünglichen Programmkontext um neue Klassen, Methoden und Attribute, sofern diese noch nicht vorhanden sind. Die dynamischen Teile bewirken eine Veränderung des ursprünglichen Programmtextes.

Die eingewobenen Musterklassen sind meist unvollständig und müssen daraufhin manuell ergänzt werden. Bei Anwendung eines *Flyweight*-Musters sieht der zu vervollständigende Programmtext an einer Stelle etwa so aus (Abbildung 4):

```
Class FlyweightFactory {
      public Flyweight getFlyweight() {
            // put your code here
      }
      …
}
```

Abbildung 4: Durch objectiF generierter Programmtext.

Damit der Entwickler nun den generierten Programmtext durch anwendungsspezifische Teile ergänzen kann, muss er die genaue Bedeutung der Klasse *FlyweightFactory* im Kontext des *Flyweight*-Musters kennen.

1.1.2 Fazit zur heutigen Arbeit mit Entwurfsmustern

Der Prozess der Musterauswahl und -anwendung mit Hilfe heutiger Modellierungswerkzeuge wie *objectiF*, *Rational Rose* oder *Together* befreit den Entwickler nicht davon, sich umfangreiche Kenntnisse zu spezifischen Entwurfsmustern aneignen zu müssen.
Die heutzutage übliche Vorgehensweise bei der Auswahl und Anwendung von Mustern erfordert in jedem Schritt ein hohes Maß an Musterverständnis. Auch die Hilfetexte und zu im Repository verfügbaren Mustern zum besseren Überblick angezeigten UML-Diagramme ändern nichts an der Grundproblematik, dass der Entwickler intensive Erfahrung mitbringen muss. Die formale Dokumentation von Mustern mit einem Modellierungswerkzeug wie *objectiF* (soweit dies mit *objectiF* möglich ist) setzt ebenfalls ein hohes Musterverständnis voraus, wie oben beschrieben.
Ein wichtiger Grund für die ungenügende Werkzeugunterstützung bei der Arbeit mit Entwurfsmusters ist nach Meinung des Autors die fehlende Möglichkeit, die Intention und den Anwendungsbereich eines Musters formal verarbeiten zu können. Bisherige Ansätze gehen hierauf vergleichsweise wenig ein. Werkzeuge, die vergleichbar dem in Absatz 1.1.1 sind, tun dies überhaupt nicht.
Um das Verständnis von Personen, die mit Entwurfsmustern arbeiten, zu vertiefen, wird im folgenden Abschnitt das Ergebnis einer Fragebogenauswertung vorgestellt.

1.1.3 Ergebnisse einer Befragung zur Arbeit mit Entwurfsmustern

Die folgenden Ausführungen sind für das Verständnis der Arbeit nicht unbedingt erforderlich, bei Bedarf kann direkt beim Fazit in Abschnitt 1.1.3.1 weiter gelesen werden.

Der Autor hat einen Fragebogen im Word-Format an 53 Personen per E-Mail geschickt. Der Fragebogen wurde in Deutsch und Englisch verfasst, deutschsprachige Personen erhielten den deutschen, alle anderen den englischen Fragebogen. Die befragten Personen kamen alle schon mit Entwurfsmustern in Berührung. Sie stammen aus verschiedenen Berufsgruppen, darunter Forscher,

Freiberufler und Angestellte in der Software-Entwicklung. Die Personen sind dem Autor teils als ehemalige Kommilitonen, teils durch Besuche bei Konferenzen wie der EuroPLoP bekannt. Einige Personen waren dem Autor nicht bekannt. Sie stammen von Verteilerlisten der EuroPLoP-Konferenzen (Themenschwerpunkt Muster). Jede Person hat denselben Fragebogen ohne personalisierte Kommentare erhalten. Von allen angeschriebenen Personen haben 14 geantwortet. Auch wenn der Autor die Rücklaufquote für diesen komplexen Fragebogen als recht hoch einschätzt, ist die absolute Anzahl an Rücksendungen nicht endgültig befriedigend. Die zurückgesandten Fragebögen waren allesamt vollständig oder fast vollständig beantwortet.
Der Fragebogen beinhaltet 25 Fragen, darunter sechs Fragen mit 2 bis 4 Teilfragen. Alle Fragen waren mit Fließtext zu beantworten und ließen Mehrfachnennungen zu. Einige Fragen ließen faktisch die Beantwortung mit »Ja«, »Nein« oder »Weiß nicht« zu, wenngleich keine Ankreuzfelder o.ä. vorgegeben waren. Zu einigen Fragen wurden zum besseren Verständnis der Fragestellung verschiedene Beispiele aufgezählt.
Sechs Fragen bezogen sich auf den Hintergrund der befragten Person. Dazu zählen etwa die Fragen nach der ersten Berührung mit Entwurfsmustern und nach den bei der Software-Entwicklung verwendeten Werkzeugen. Die weiteren Fragen bezogen sich konkret auf Entwurfsmuster. Sie zielten auf die Arbeitsweise des Befragten mit Mustern sowie dessen Musterverständnis. Zwei Fragen fokussierten Metainformationen in Quelltexten. Dies deshalb, weil solche Metainformationen ein wichtiges Konzept dieser Arbeit sind. Eine Frage bezog sich auf die UML. Der Grund hierfür ist, dass es viele Arbeiten zum Thema Entwurfsmuster gibt, die UML betrachten. Diese Arbeit hingegen kann die UML aufgrund ihrer Hauptausrichtung auf Quelltext und der Komplexität des Themas nicht berücksichtigen. Es galt herauszufinden, inwieweit die UML faktisch eine Rolle bei der Arbeit mit Mustern spielt, um auszuschließen, dass ein fundamentaler Baustein bei der Musterarbeit ausgelassen wird. Die letzte Frage ließ Platz für allgemeine Kommentare zum Fragebogen. Hier wurden allerdings keine nennenswerten Angaben gemacht.

Die Auswertung der ausgefüllten Fragebögen ergab die folgenden Ergebnisse. Sieben Befragte gaben an, erstmals über universitäre Beschäftigung (Vorlesung, Doktorarbeit, Seminar) Zugang zu Entwurfsmustern bekommen zu haben; zwei Befragten eröffnete sich die Materie durch die Arbeit in der kommerziellen Software-Entwicklung, drei Personen gaben das Gamma-Buch [Gamma+ 1995] als Ursprung an; zwei Antworten fehlten. Das Jahr des ersten Kontakts variiert von 1994 bis 2003.
Elf Personen haben durch ihre Tätigkeit im Bereich kommerzielle Software-Entwicklung mit Entwurfsmustern zu tun. Fünf davon waren gleichzeitig im akademischen Bereich mit diesem Thema beschäftigt. Drei Personen standen ausschließlich in akademischer Tätigkeit mit Entwurfsmustern in Kontakt.

Befragt nach der Vorgehensweise bei der Software-Entwicklung, gaben sechs Personen an, immer oder fast immer mit einem Modell zu starten. Konkret wurde – auch im Fortlauf des Fragebogens – häufig die UML als Modellierungssprache genannt. Vier weitere Personen verwenden ein Modell nur als Gedankenstütze, zwei davon gaben an, das Modell informal zu skizzieren bzw. auf Papier zu zeichnen. Von allen Befragten verwenden zwei ein Modell, um daraus Quelltext zu generieren. Der generierte Quelltext wird dabei nicht verändert, sondern erweitert (Konzept der geschützten Regionen, auch als *Protected Regions* bezeichnet). Eine Person verwendet OMT-Diagramme zur Illustration von Programmstrukturen. Zwölf Befragte implementieren größtenteils oder ausschließlich manuell, ohne nennenswerte Generierung aus Modellen. Eine Person gab an, Software nur von Anfang an zu entwickeln, also nicht bestehende Software zu verändern oder zu erweitern. Zwei Personen gaben explizit an, Refactoring in der Erstellungsphase zu verwenden. Der Autor geht davon aus, dass diese Tätigkeit bei den anderen Befragten häufiger vertreten ist, eine entsprechende Nennung aber nicht erfolgt ist.

Weiterhin wurde nach dem Besitz von Musterbüchern und -dokumenten gefragt. Zwölf der Befragten besitzen das Gamma-Buch, nur zwei nicht. Bücher der POSA-Reihe (*Pattern Oriented Software Architecture*, etwa [Buschmann+ 1998]) sind im Besitz von sieben Befragten. Vier Personen besitzen ca. 15 oder mehr Musterbücher, fünf weitere Personen sechs oder mehr dieser Bücher. Die

Büchernennung beinhaltet auch viele wenig populäre Bücher, die teils domänenspezifisch sind (etwa auf Analysemuster – vgl. [Fowler 1998] – spezialisiert). Anhand der Nennung wird ersichtlich, dass der Musterbegriff Interpretationsspielraum bietet und weit gefasst ist.

Es wurde danach gefragt, wie viele Entwurfsmuster so gut bekannt sind, dass sie in einem gegebenen Quelltext oder UML-Diagramm identifiziert werden können. Sieben der Befragten gaben unbedingt an, dass dies fünf Muster seien. Zwei Personen antworteten ohne Quantifizierung mit dem Hinweis, es kommt auf das Ausgangsmaterial an. Vier weiteren Personen nannten 10 bis 25 Muster als für sie identifizierbar. Eine Person gab hier 100 bis 200 Muster an, eine Antwort die aufgrund der intensiven Beschäftigung des Befragten mit Mustern als ernst gemeint gewertet wurde.
Der zweite Teil der Frage betrachtete die Anzahl der Muster, die den Befragten so gut geläufig sind, dass die ohne Nachschlagewerk angewandt werden können. Hier fielen die genannten Zahlen erwartungsgemäß niedriger aus, da die Musteranwendung mehr Kenntnisse erfordert als deren Identifikation (vergleichbar mit einem aktiven und einem passiven Wortschatz). Drei Befragte schauen immer vor der Musteranwendung nach, ein weiterer nur selten. Sieben Personen kennen ca. fünf Muster entsprechend gut, zwei Personen 10 bis 15. Die meisten Muster mit 100 bis 200 kennt wieder derjenige, der im ersten Teil der Frage dasselbe geantwortet hat.

Zum Ermitteln eines geeigneten Entwurfsmusters für ein gegebenes Problem verwenden zehn Personen immer oder meist die Erinnerung. Der Rest der Befragten verwendet Bücher, Tagungsbände, Webseiten oder fragt eine kompetente Person.

Auf die Frage, wie sie Muster anwenden, antworteten sieben, sie tun dies manuell auf Modellebene. Sechs davon implementieren Muster manuell im Quelltext. Sechs weitere Personen verwenden kein Modell (weder als Gedankenstütze, noch als Ausgangspunkt für eine Generierung), sondern implementieren direkt manuell im Quelltext. Eine Person antwortete zu unspezifisch, als dass eine Zuordnung möglich wäre. Eine Person, die direkt manuell im Quelltext implementiert, nutzt gelegentlich auch automatisch generierten Code oder *Copy & Paste* von Beispielquelltext als Basis für die Musterimplementierung.

Gefragt nach den am häufigsten angewandten Entwurfsmustern gab es pro Person im Durchschnitt eine überraschend kleine Liste an Nennungen. Drei Befragte gaben an, es kommt auf den Anwendungskontext an, welche Muster sie verwenden; sie gaben keine konkreten Muster an (bis auf eines, nämlich *Forward-Receiver*). Von den restlichen gaben sieben an, *Factory Method* häufig zu benutzen, ebenso *MVC* (kurz für: *Model-View-Controller*). *Singleton* schlug mit sechs Nennungen zu Buche. *Publish-Subscribe*[3] kam auf vier Nennungen, *Façade*, *Observer,* und *Composite* wurden dreimal genannt, *Proxy*, *Strategy* und *Decorator* zweimal. Einzelnennungen gab es für *Visitor*, *Iterator*, *Adapter*, *Template Method*, *Prototype*, *Mediator, Layers* und *Presentation-Abstraction-Controller*. Eine Person programmiert mittlerweile nicht mehr und konnte hier nicht antworten.

Bei der Entwicklung von Software verwenden acht Personen Modellierungswerkzeuge wie Rational Rose, wenn auch fast immer nur für Modelle ohne formalen Quelltextbezug; Visio und MagicDraw wurden ebenfalls genannt (hiermit wird „nur“ informal gezeichnet). Häufig wurde Eclipse als Entwicklungsumgebung genannt. Microsoft Word wurde öfters für die Dokumentation erwähnt. JUnit[4] wird von den Befragten oft als Testwerkzeug für automatisierte Softwaretests eingesetzt (vgl. auch [Meffert 2006b]). Die im SAP-Bereich Tätigen gaben die zentral verfügbare und so gut wie immer verwendete Funktionalität der SAP-Transaktion SE80 an (eine Transaktion ist im SAP-Umfeld

[3] *Publish-Subscribe* und *Observer* sind vergleichbar und können hier als ein Muster gewertet werden.

[4] JUnit ist ein Java-Framework für den feingranularen automatisierten Test. Ab der Version 4 werden Java-Annotationen verwendet, um Methoden als Testfälle zu kennzeichnen. In Vorgängerversionen wurden Methoden durch eine Namenskonvention (festes Präfix »test«) als JUnit Test markiert.

eine über ein Menü oder ein Kürzel aufrufbare, abgeschlossene Funktionalität auf meist einer Maske).

Ungefähr die Hälfte der Befragten verwendet eigene Frameworks[5] oder solche von Drittanbietern während der Entwicklung, die andere Hälfte nicht oder kaum. Für die Sprache Java wurde das Framework Spring öfters genannt.

Gefragt nach ihrer Meinung, welcher Personenkreis geeignet erscheint, um Entwurfsmuster für ein gegebenes Problem auszuwählen, gaben zehn den erfahrenen Entwickler oder Software-Architekten an. Zwei Befragte waren der Meinung, dies könne jeder tun, eine Person antwortete nicht, einer Person war die Frage nicht klar. Die adäquate Anwendung von Entwurfsmustern sahen neun in Zuständigkeit des erfahrenen Entwicklers oder Software-Architekten, zwei Antworten hielten den durchschnittlichen Entwickler für kompetent genug. Die Identifikation schon implementierter Entwurfsmuster wurde in fast gleich beantwortet wie für die Musteranwendung. Weiterhin wurde jedoch verstärkt der Modellierer als kompetente Person für die Aufgabe der Musteridentifikation angeführt.

Befragt nach dem Zeitpunkt der Musterverwendung und -anwendung, gaben sieben an, dies sei schon in der Analysephase der Fall. Drei Befragte erwähnten, dass es hierfür doch Analysemuster gäbe oder schrieben, sie verwenden dafür nur Analysemuster. Zwei Befragte tun das nur bedingt. Während des Forward Engineerings oder der Prototypphase gaben alle an, Muster anzuwenden, eine Person gab allerdings keine Antwort. Eine von den elf Personen schränkte die Anwendung ein. Zu Dokumentationszwecken verwenden acht Personen Entwurfsmuster, zwei weitere nur bedingt oder selten. Während der Reengineering- oder Wartungsphase verwenden acht Befragte Muster, vier gaben an, es sei zu spät, erst in dieser Phase Muster zu verwenden. Drei Personen antworteten hier mit einem exklusiven »Nein«.

Von den 14 antwortenden Personen gaben 12 an, mit Java schon Entwurfsmuster implementiert zu haben. Vier Personen davon meinten, dass diese Aufgabe mit Java eleganter als mit anderen Sprachen umsetzbar sei, zwei Java-Kundige gaben keine Präferenz an, vier kennen nur Java. Eine Person bevorzugt Smalltalk gegenüber Java (und Ruby). Eine Person, die Java präferiert, kennt zusätzlich C++, C, Python und PHP, eine zweite kennt zusätzlich C++ und C#, eine dritte ABAP[6] und PHP, eine vierte C++ und eine fünfte Python. Eine Person kennt nur C++ und eine nur ABAP.

Vor die Wahl gestellt, ob mehr Werkzeugunterstützung für den Prozess der Musterselektion gewünscht wird, haben sieben mit „Ja“ geantwortet. Vier Antworten waren „Nein“ oder „Eher nein“, eine Antwort lautete „Evtl. ja“. Eine Person aus der Forschung gab an, viele Ansätze gesehen zu haben (inklusive eigener), die allesamt nicht geeignet erschienen. Ein Befragter konnte sich nicht vorstellen, dass ihm ein Werkzeug hier helfen kann. Die Frage nach dem Wunsch nach Werkzeugunterstützung für die Musteranwendung wurde analog beantwortet. Eine Antwort gab einen Hinweis darauf, dass Variationen in Mustern ausreichend gewürdigt werden müssten, eine andere verwies darauf, dass das Werkzeug schon sehr gut sein müsste, um praxistauglich zu sein, eine dritte relativierte abhängig von der Natur des Musters. Ein reiner Vorlagenmechanismus (Einsatz von Quelltext-Fragmenten) wurde in letztgenannter Antwort mehrfach abgelehnt, eine andere Person forderte jedoch genau das Gegenteil, nämlich die Generierung von Code-Rahmen zur Unterstützung bei der Musteranwendung. Der Wunsch nach Unterstützung durch Werkzeuge bei der Mustererkennung herrscht bei sieben Personen vor, einer sieht die Realisierbarkeit aber in weiter

[5] Frameworks spielen nach Meinung des Autors in Verbindung mit Mustern deshalb eine Rolle, weil sie einerseits Muster verwenden und andererseits den Entwickler entlasten, Muster für ein gegebenes Problem finden zu müssen. Es sei angemerkt, dass [Pree 1995] Frameworks teilweise mit Entwurfsmustern gleichsetzt.

[6] ABAP ist eine von SAP erfundene Sprache, die heutzutage in der SAP-Welt die wichtigste Sprache ist und eine objektorientierte Erweiterung namens ABAP/Objects erhalten hat.

Ferne. Eine Person fordert die Erkennbarkeit von Mustern durch Namenskonventionen. Eine Person hält automatische Mustererkennung für unmöglich. Vier Personen antworteten negativ, eine Person macht die Antwort wie in der vorigen Frage abhängig von der Natur des Musters. Für die formale Musterdokumentation wünschen sich acht der Befragten mehr Unterstützung. Die anderen verneinen oder sind mit Büchern und Tagungsbänden zufrieden. Einer findet Prosatext besser als Formalismen, ein anderer wünscht sich stattdessen kompetentere Entwickler.

Werkzeugunterstützung bei der Arbeit mit Entwurfsmustern können sich fünf Personen in Form einer Wissensdatenbank (im weiteren Sinne des Begriffs) vorstellen. Eine Code-Generierung oder *Copy & Paste*-Vorlagen konnten sich fünf Personen vorstellen. Weitere Nennungen in kleinerer Anzahl waren „intelligenter Mechanismus“, „Muster Browser“, „Anzeige von Musterdokumentation“, „Teil eines Bugtracking-Systems“, „Tool muss offen für Aufnahme neuer Muster sein“, „Hypertextrepräsentation“, „Attributierung / Metainformation“. Ein Befragter sieht hier keinen Bedarf.

Befragt danach, ob eine formale Musterdokumentation als möglich erachtet wird, antworteten drei mit einem klaren »Ja«, zwei mit »Beschränkt ja«, einer mit »Wohl ja«, zwei mit einem klaren »Nein«, zwei wussten das nicht einzuschätzen, einer hält fast nichts für unmöglich, zwei finden das Unterfangen schwierig und einer gab keine Antwort. Pro Fragebogen wurde zu dieser Frage nur eine Antwort gegeben. Eine formale Musterdokumentation erachten drei Personen als klar sinnvoll, zwei wissen es nicht, zwei bezweifeln den Zweck der Formalisierung aufgrund einer unerwünschten Spezialisierung, einer sieht keinen Sinn für den industriellen Einsatz, einer fordert ein einheitliches Format für alle Muster, drei antworteten mit »Nein«, einer macht das vom einzelnen Muster abhängig und einer gab keine Antwort. Pro Fragebogen wurde zu dieser Frage nur eine Antwort gegeben. Ein Befragter merkt an, dass „[…]Muster oft auf verschiedenen Abstraktionsebenen dokumentiert [sind] und es unklar [ist], ob jede Community, die Muster nutzt, mit der Formalisierung auch etwas anfangen kann.“

Bis auf einen Befragten ist keinem eine sinnvolle Methodik bekannt, Muster formal zu definieren. Eine Person nannte »Matt Dwyer« als Möglichkeit (siehe [Spec 2007]). Eine Anmerkung lautete: „[Eine solche Formalisierung] kann es auch nicht geben bzw. diese Muster beschreiben dann etwas anderes als das was hinter der Entwurfsmusteridee steckt.“. Ein Befragter antwortete mit „Ja, aber keine sinnvolle.“. Eine Antwort fehlte.

Die Befragten wurden gebeten, die aus ihrer Sicht vorhandenen Variationspunkte von Entwurfsmustern zu benennen. Fünf Personen gaben keine Antwort, eine Person verstand die Frage nicht. Ebenso viele sahen in allem einen potentiellen Variationspunkt. Zwei gaben den Namen von Methoden oder Klassen sowie die Struktur des Musters an. Einer nannte synonyme Muster sowie Musterkombinationen als Variationsmöglichkeit. Eine weitere Person nannte die Problemstellung, Kräfte, Folgen und die Lösung eines Musters.

Die Begriffe *Entwurfsmuster* und *Architekturmuster* konnten neun Personen unterscheiden, die meisten davon gaben eine konkrete Definition an, die anderen waren unsicher und antworteten vage; zwei sehen einen fließenden Übergang zwischen diesen beiden Begriffen; mehrere Nennungen gab es für ein unterschiedliches Abstraktionsniveau. Eine Person hält den Begriff *Architekturmuster* nicht für zielführend, eine Person schrieb „[…]auch Architekturmuster sind Entwurfsmuster“. Das Muster *MVC* haben fünf Personen den Architekturmustern zugeordnet und vier als Hybrid eingestuft (je nach Abstraktionsniveau könne *MVC* ein Architektur- oder ein Entwurfsmuster sein). Die restlichen fünf Personen gaben hierzu keine Auskunft. Gefragt danach, ob eine Unterscheidung zwischen Architektur- und Entwurfsmuster für sinnvoll gehalten wird, antworteten sieben mit „Ja“, fünf mit „Nein“ oder „Eher nicht“ und zwei wussten es nicht einzuschätzen. Eine Anmerkung war, dass der Begriff *Entwurfsmuster* unglücklich sei, weil eine Architektur ebenfalls entworfen werde.

Zu Anwendungsgebieten für Metainformationen in Quelltexten befragt, wurden siebenmal Annotationen genannt und ebenso oft Javadoc (Java-eigener Kommentarstil). Todo-Marker (»Todo« = noch zu erledigen) wurden viermal genannt. Weitere Antworten waren „Verbindung von Komponenten“, „Relationales Mapping, Bedingte Kompilierung“, „Bezug zur Entwicklerdokumentation“, „Fertigstellungsgrad und Qualitätsstatus“ und zweimal „JUnit Tests“.
Auf die Frage nach Möglichkeiten, Metainformationen zu Quelltext hinzuzufügen, wurden mehrmals spezielle Zeichen oder Marker genannt, Namenskonventionen wurden einmal erwähnt, ebenso spezielle Schlüsselwörter. Als Techniken wurden Kommentare, Javadoc, Annotationen und XML angegeben. Zwei Personen sehen als Möglichkeit die automatische Generierung von Metainformationen.

Zehn Befragte kennen Java-Annotationen, vier nicht. Neun Kenner finden Annotationen nützlich (einer davon findet Annotationen „positiv“), einer konnte sich zum Nutzen nicht äußern. Eine Person kritisierte den fehlenden Bezug zwischen Annotationen (sowie Javadoc) und Entwicklerdokumentationen.

Die letzte Frage konzentrierte sich auf die UML. Die Befragten nutzen die UML für ganz unterschiedliche Zwecke, nämlich „[…]zur Modellierung von statischen bzw. dynamischen Aspekten eines Systems“, zur „[…] Kommunikation zwischen Team-Mitgliedern“, als „[…] Ausgangspunkt für automatische Code-Generierung“, zur »Modellierung (ohne Generierung)«, dreimal »Dokumentation« (einmal davon im Nachhinein), zweimal »Design-Modelle«, »Redesign«, »Prozessdarstellung«, »Darstellung von Abhängigkeiten«, »Entwurfsdiskussionen«, »Modellierung von Datenfluss«. Einer merkte an, dass die „[…] UML nicht als Allheilmittel oder Zauberwerkzeug missverstanden werden [soll]“. Er fügte hinzu dass die UML „[…] zur nachträglichen Dokumentation von existierendem Code […] eher ungeeignet [ist]“. Eine Person merkte an, dass sie zwar mit UML Code generiert, aber die Qualität des Generats an der Semantikschwäche der UML leidet.

1.1.3.1 Fazit der Fragebogenauswertung

Die Analyse von nur 14 Fragebögen kann kein sicheres repräsentatives Ergebnis ergeben. Jedoch lassen sich schon aus den gegebenen Antworten einige Rückschlüsse ableiten, zumal die antwortenden Personen allesamt aus der Branche der Software-Entwicklung kommen und nach Kenntnislage des Autors ausgewiesene Experten sind, also eine entsprechende Erfahrung aufweisen können. Eine vom Autor früher durchgeführte, öffentliche Online-Umfrage in der SAP-Community, die hier nicht näher genannt ist, hat zu den Fragen des Nutzens und der Nutzung von Mustern sowie zu deren Bekanntheitsgrad qualitativ das gleiche Ergebnis hervorgebracht.

Die Strahlkraft der universitären Ausbildung hinsichtlich der Popularisierung von Entwurfsmustern scheint sehr groß. Zusammen mit dem Gamma-Buch [Gamma+ 1995] bildet sie die größte Impulsquelle für das Bekanntwerden von Entwurfsmustern.

Modellbasierte Ansätze sind gemäß der Fragebogenauswertung von untergeordneter Stellung. Wenn Modelle verwendet werden, dann oft nur als Gedankenstütze oder Diskussionsgrundlage, nicht als Vorlage für die Generierung von Programmtext. Die UML ist der populärste Vertreter von Modellierungsmöglichkeiten. Die Ergebnisse deuten darauf hin, dass der hauptsächliche Entwicklungsaufwand durch die manuelle Implementierung von Quelltext betrieben wird.
Die Implementierung von Mustern findet meist faktisch ohne Werkzeuge statt. Die bei der täglichen Arbeit verwendeten Werkzeuge bieten keine besonders erwähnenswerte Musterunterstützung. Die Erinnerung dient bei der Implementierung von Mustern in erster Linie als Quelle der Erkenntnis. Diese Erkenntnis wurde zu einem Gutteil durch das Gamma-Buch gewonnen, das ca. 80 Prozent aller Befragten besitzen. So ist es nicht verwunderlich, dass die meisten für die Anwendung bekannten Muster im Gamma-Buch zu finden sind, also etwa *Factory Method* oder *Singleton*. Das komplexe *MVC*-Muster hingegen wurde überraschend oft genannt. Das ist einerseits dadurch zu erklären, dass die POSA-Bücher, die das Muster dokumentieren, ebenfalls weit verbreitet sind. Andererseits ist es

nach Erfahrung des Autors so, dass viele Frameworks und auch die SAP-Technologie namens *Web Dynpro* das *MVC*-Muster bereits enthalten und ohne Mehraufwand bereitstellen. Überrascht hat, dass nur wenige Muster häufiger angewandt werden und die meisten Muster selten oder nie pro Entwickler Verwendung finden. Die Musteranwendung findet meist direkt im Quelltext statt. Wenn ein Muster auf Modellebene eingebracht wird, dann fast immer zu Zwecken der Visualisierung bzw. als Gedankenstütze.

Die Identifikation von Mustern in einem gegebenen Quelltext ist vielen fast gar nicht möglich, nämlich nur für ungefähr fünf Muster. Die Musteranwendung ohne Nachschlagewerk ist gemäß der Auswertung noch schwieriger, was als Indikator für die Komplexität von Mustern gewertet werden kann. Unterstützend wirkt die Angabe der meisten Befragten, dass die Musterauswahl für ein gegebenes Problem nur durch Experten vorgenommen werden sollte. Nur eine Person gab an, extrem viele Muster ohne Hilfsmittel sowohl identifizieren als auch anwenden zu können. Es handelt sich nach Information des Autors um eine Person, die täglich und im Rahmen der Forschung mit Mustern in Berührung kommt. Es verwundert den Autor, dass die Verwendung von Frameworks eine laut den Fragebögen nur geringe Verbreitung bietet (vgl. [Pree 1995], wo die Vorteile von Frameworks, auch in Hinblick auf Entwurfsmuster, herausgestellt werden). Auf spezifische Domänen maßgeschneiderte Frameworks erleichtern nach Erfahrung des Autors und gemäß einer Antwort im Fragebogen die tägliche Arbeit oft erheblich und können den Aufwand zur fehlerfreien und sauberen Umsetzung einer Lösung ganz deutlich reduzieren. Der Zeitpunkt der Musteranwendung wird von allen in der Forward Engineering-Phase gesehen. Ähnlich, wenngleich weniger deutlich war der Tenor bezüglich der Reengineering-Phase[7]. Immerhin die Hälfte der Befragten sieht einen Nutzen von Mustern zur Dokumentation eines Programms. Zehn der Befragten gaben an, Kenntnisse in Java zu haben. C++, C# und Smalltalk wurden fast gar nicht genannt.

Sieben Personen gaben an, sich Unterstützung durch Werkzeuge bei der Musterauswahl zu wünschen. Gleiches gilt für Musteranwendung und -erkennung. Einige Befragte standen der Realisierungsmöglichkeit von werkzeuggestützten Ansätzen für die Unterstützung bei der Arbeit mit Mustern kritisch gegenüber, bis hin zur Aussage, dies sei nicht möglich. Die populärste Lösungsidee war die Verwendung einer Wissensdatenbank, ohne dass hier nähere Details genannt wurden. Die Ratlosigkeit bezüglich konkreter Umsetzungsvorschläge wird weiter deutlich durch die Forderung nach einem „[…] intelligenten Mechanismus“. Einen auf Vorlagen basierenden Ansatz halten einige ebenfalls für lohnenswert. Hier und auch bei anderen Fragen wird die Trivialtechnik Copy & Paste erwähnt. Die Vorschläge »Muster Browser« und »Hypertextrepräsentation« deuten zusammen mit dem Vorschlag der Wissensdatenbank darauf hin, dass durchaus Möglichkeiten gesehen werden, die in die gleiche Richtung weisen. Der Entwickler soll möglichst einfach Zugriff auf relevante Informationen haben, um Muster effektiv anwenden zu können. Dies wird unterstrichen durch die Aussage eines Befragten, der sich kompetentere Entwickler wünscht, eine Forderung die wohl durch keinen akademischen Ansatz erfüllt werden kann.

Die Möglichkeit einer formalen Musterdokumentation wurde kontrovers und nicht eindeutig beantwortet. Dies mag daran liegen, dass der Begriff der formalen Dokumentation nicht erläutert wurde und auch keine allgemein geläufige Definition hierfür existiert. Etwas positiver sehen die Befragten den Nutzen einer formalen Musterdokumentation. Eine Methode zur formalen Musterdokumentation ist niemandem bekannt. Es wurde von Befragten darauf hingewiesen, dass durch Formalisierung Mustervarianten verloren gehen können. Als Variationspunkte von Mustern wurde von einem drittel der Befragten alles angesehen, weniger populär waren die Variation des Namens, der Struktur und der Kombination von Mustern.

Die Unstimmigkeiten bezüglich des Musterverständnisses wurden durch die unterschiedliche Wahrnehmung der Begriffe *Architektur-* und *Entwurfsmuster* deutlich. Im Ergebnis ordneten ebenso

---

[7] Nach Meinung des Autors existiert bei der auf Quelltext fokussierten Entwicklung kein qualitativer Unterschied zwischen Forward und Reengineering-Phase bezüglich der Musteranwendung, sofern die Forward Engineering-Phase nur weit genug fortgeschritten ist. Es ist in manchem industriellen Software-Projekt, das der Autor begleitet hat, nicht möglich gewesen, alleine durch Anschauung des Quelltextes festzustellen, ob sich das Projekt in der Forward- oder der Reengineering-Phase befunden hat.

viele das *MVC*-Muster dem Architekturmuster zu wie es als Hybrid zu bezeichnen. Die fehlende Angabe kam genauso oft vor, bei dem sonst gut ausgefüllten Fragebogen mag das ein Zeichen für die Unsicherheit bei dieser Frage sein.
Viele Befragte gaben an, Annotationen und Javadoc-Kommentare als Möglichkeit für das Anbringen von Metainformationen in Quelltexten zu sehen. Bestehende Anwendungen von Metainformationen wurden insbesondere in Todo-Markern gesehen. Nur einmal genannt, aber nach Erfahrung des Autors ebenfalls populär sind Annotationen zur Erstellung einer Persistenzschicht. XML wurde einmal als Möglichkeit für Metainformationen genannt. Kenntnisse zu Javadoc-Kommentaren sind unter den Befragten weit verbreitet.
Der typische Einsatz der UML kann aus den Fragebogenergebnissen nicht repräsentativ benannt werden, zu unterschiedlich sind die Antworten ausgefallen. Auffällig war jedoch dass UML-Diagramme fast immer nur als Hilfskonstrukt und nicht zur Generierung von Quelltext verwendet werden.

Insgesamt lässt sich feststellen, dass kein Konsens zu herrschen scheint, wenn es um das Einsatzgebiet von Mustern, die Methodik zur Verwendung von Mustern, Begriffsbestimmungen, Nutzen von Mustern, benötigte Unterstützung zur Arbeit mit oder die Dokumentation von Mustern geht.

Neben der in Absatz 1.1.1 genannten suboptimalen Werkzeugunterstützung und den in diesem Absatz thematisierten unterschiedlichen Wahrnehmungen von Mustern existiert eine ganze Reihe von Problemen bei der Arbeit mit Entwurfsmustern, die im nächsten Abschnitt betrachtet werden. Sie sollen helfen, die auf Schwierigkeiten bei der Arbeit mit Mustern hindeutenden Fragebogenergebnisse einzuordnen.

1.1.4 Spezifische Probleme mit Entwurfsmustern

Entwurfsmuster bringen aus Ihrer Grundidee heraus eine ganze Reihe an problematischen Aspekten hervor, die in den nächsten Absätzen diskutiert werden. Die im Weiteren genannten Gründe könnten eine Erklärung für die teils unbefriedigenden Ergebnisse der Fragebogenauswertung hinsichtlich der heutigen Arbeit mit Mustern sein. Der Autor hat zusätzlich zu den unten genannten Quellen eine Reihe von Diskussionen im Netz ausgewertet, die die folgenden Ausführungen unterstützen, siehe auch [Meffert 2004a] oder [Meffert 2004b].

1.1.4.1 Qualität von Mustern

Erfahrung lässt sich nicht objektiv messen oder beurteilen (siehe auch [Fayad 2007a]). Das bedeutet, es ist nicht klar, wann ein Entwickler als erfahren gilt und wann nicht. Alleine die Beschäftigung mit der Software-Entwicklung über einen langen Zeitraum bedeutet nicht automatisch das Vorhandensein von relevanter Erfahrung für die Arbeit mit Entwurfsmustern. Es existieren keine anerkannten und populären Standards für das Abschätzen der Erfahrung von Entwicklern in der industriellen Software-Entwicklung (siehe auch [Fayad 2007a]).
Daraus folgt, dass ein Entwurfsmuster nicht automatisch gut sein muss, nur weil es der Allgemeinheit durch Dokumentation verfügbar gemacht wurde[8].
Eine weitere Folge ist, dass die Auswahl eines dokumentierten Musters für einen gegebenen Kontext durch einen Entwickler mit ungenügenden Voraussetzungen mit dem Risiko verbunden ist, dass das gefundene Muster für den Kontext nicht optimal oder sogar schädlich ist (siehe [Fayad+ 2007b]).

[8] Viele Teilnehmer der Konferenzen der PLoP-Reihe (u.a. PLoP, EuroPLoP, VikingPLoP), deren Hauptthema Entwurfs- und ähnliche Muster sind, fordern drei bekannte Verwendungen, um ein Muster als solches gelten zu lassen.

Bestehende Muster können durch technologischen Fortschritt veralten (siehe auch [Fayad 2007g]). Ist ein Muster aber erst einmal dokumentiert, etwa innerhalb eines Buches, so bleibt es auch nach Veraltung bestehen und vermittelt womöglich den Eindruck der Aktualität. Bei im Internet verfügbaren Musterdokumentationen (siehe auch Abschnitt 1.1.4.3) entschärft sich das Problem der Kennzeichnung veralteter Muster, da ein Online-Beitrag jederzeit modifiziert werden kann und sich den Lesern beim nächsten Abruf in aktualisierter Form präsentiert. Das Problem der Erkennung einer Veraltung bleibt jedoch unverändert bestehen. Die Gründe dafür sind, dass erstens Aktualisierungen auch im Internet fehlen können und zweitens nicht aktuelle Beiträge nicht als veraltet erkennbar sind, wenn sie nicht mit einem entsprechenden Vermerk oder Datum versehen sind.

1.1.4.2 Einsatzgebiet von Mustern

Nach [Fayad+ 2007d] und [Fayad+ 2007e] werden Entwurfsmuster hauptsächlich in der Entwurfsphase eines Software-Projekts eingesetzt. Der Autor dieser Arbeit sieht ein wesentliches Anwendungsgebiet auch im Reengineering. [Fayad+ 2007d] stellt die Frage, ob der Einsatz eines Musters in unterschiedlichen Phasen eines Projekts analog möglich ist bzw. mit vergleichbarem Nutzen oder ob ein Muster leichter in einer als in einer anderen Phase eingesetzt werden kann.

1.1.4.3 Musterdokumentation

Die Musterdokumentation ist die Beschreibung eines Musters, die nach formalem und informalem Charakter unterschieden werden kann. Der Begriff wird in Abschnitt 2.3.1 genauer erläutert.
Die Anzahl dokumentierter Muster ist unüberschaubar hoch (siehe u.a. [Taibi+ 2003], [Fowler 2003a], [Gamma+ 1995]). Gleichwohl wird „[…] eine große Menge von Mustern, die viele Arten von Entwurfsproblemen abdeckt […]“ ([Buschmann+ 1998, S. 22f]), gefordert. Wie [Zimmer 1997, S.35f] beschreibt, ist das Klassifikationsschema der *Gang of Four* (kurz: *GoF,* siehe [Gamma+ 1995]) nicht für eine größere Anzahl Muster (> 100) geeignet.

Generell bedingt die Arbeit mit Entwurfsmustern das Lesen von natürlichsprachigen Musterdokumentationen (vgl. etwa [Gamma+ 1995]). Andererseits basiert das Selektieren und Anwenden eines Musters für einen gegebenen Quelltext sowie das Erkennen vorhandener Muster in diesem Quelltext darauf, eine formale Syntax zu verstehen, die der Quelltext mit sich bringt. Die wechselnde Tätigkeit einerseits mit informalen, andererseits mit formalen Texten, birgt die Gefahr von Missverständnissen. Sie können auftreten beim Versuch, informale Texte, die mächtig im Ausdruck sein können, so zu interpretieren, dass sie mit formalen Quelltexten, deren Bestandteile (Anweisungen einer Programmiersprache) vergleichsweise wenig ausdrucksstark sind in Beziehung gesetzt werden können. Die nachfolgende Abbildung soll das Spannungsfeld zwischen informaler Musterdokumentation und formalem Quelltext verdeutlichen.

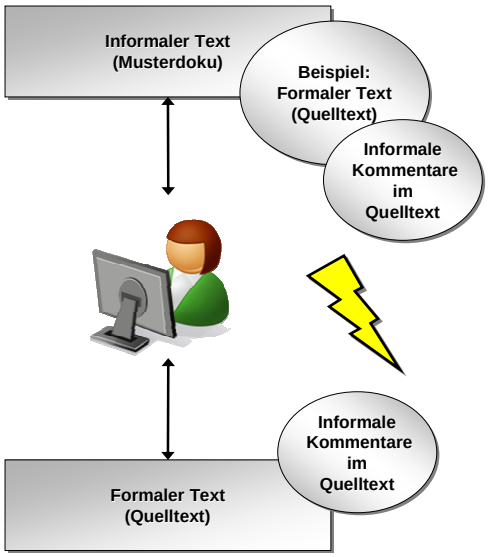

Abbildung 5: Spannungsfeld zwischen informaler Musterdokumentation und formalem Quelltext.

Die Abbildung verweist auf eine Vertiefung des Spannungsfeldes. So sind in informalen, weil überwiegend in Fließtext geschriebenen Musterdokumentationen häufig Beispiele für die Musteranwendung in Form von Quelltextabdrucken zu finden (vgl. etwa [Meffert+ 2008a]). Innerhalb dieser formalen Quelltexte wiederum sind zur Dokumentation der Bedeutung einzelner Quelltextteile Kommentare enthalten, die per se wiederum einen informalen Charakter haben. Kommentare sind weiterhin (und hoffentlich) in formalem Quelltext zu finden, der zu interpretieren ist, wenn ein Muster dafür ausgewählt, darauf angewandt oder innerhalb dessen ein bereits angewandtes Muster erkannt werden soll.

Aufgrund der Vielzahl dokumentierter Muster ist es nicht möglich, sicherzustellen, dass ein neu zu dokumentierendes Muster nicht schon anderswo verzeichnet ist. Das Problem wird durch denkbare kleine Abweichungen zwischen gleichwertigen Mustern erschwert (vergleiche auch [Fayad+ 2007a]).

Ein dokumentiertes Muster in seiner Bedeutung zu erfassen, ist nach Meinung des Autors und nach [Fayad 2007c] eine anspruchsvolle Aufgabe, deren Bewältigung zu einem erheblichen Teil von der Güte der Musterdokumentation abhängt. Letztere liegt ganz überwiegend informell vor (vgl. [Gamma+ 1995], [Buschmann+ 1998], [Fowler 2001] sowie etwa die Tagungsbände der PLoP-Konferenzen, hieraus u.a. [Meffert+ 2008a]).

Die durch Verwendung von Fließtext informelle Musterdokumentation wird in der Regel durch UML-Diagramme ergänzt. Eine Schwäche von UML liegt darin, dass tiefer gehende formale Definitionen nicht mit Standardmitteln möglich sind, sondern etwa durch Definition eigener Stereotypen bewerkstelligt werden müssen. UML eignet sich nach Erfahrung des Autors nicht in jedem Fall gut als Kommunikationsmittel zwischen Fach- und Entwicklungsabteilung, unter anderem aufgrund der zu technischen Ausrichtung, die mancher Mitarbeiter der Fachabteilung nicht teilt. Es eignet sich nach selbiger Erfahrung nur bedingt zur Vermittlung innerhalb der Entwicklungsabteilung. Klassendiagramme direkt anzulegen stellt keinen Zeitvorteil dar gegenüber direkt im Quelltext angelegten Klassen, aus denen Klassendiagramme generiert werden können. Die eingeschränkte Ausdrucksfähigkeit von UML wird durch die häufige Notwendigkeit individuell einzuführender Stereotypen deutlich. Es existiert kein Zwang, UML-Diagramme vollständig abzubilden. Beispielsweise können Kantenbezeichnungen weggelassen werden (siehe auch [Fayad 2007h]). Informale Musterdokumentationen werden durch die Kombination mit semi-formalen und potentiell unvollständigen UML-Diagrammen nicht besser maschinenverarbeitbar.

Ein Muster ist eine abstrakte Lösungsbeschreibung für eine Gruppe verwandter Probleme. Die Implementierung eines Musters ist eine konkrete Beschreibung für ein spezifisches Problem innerhalb der genannten Problemgruppe. Somit besitzt die Musterdokumentation einen anderen Abstraktionsgrad als die Implementierung. Die Beschreibung ist generisch, die Implementierung konkret. Die Musterimplementierung enthält nicht alle Informationen, die im Mustermodell vorhanden sind. Beispiele hierfür sind bekannte Verwendungen, Varianten oder Hinweise, wann das Muster anzuwenden bzw. nicht anzuwenden ist. Durch die Musteranwendung gehen also Informationen verloren, die bei einem Reengineering der Implementierung wichtig sein könnten (vgl. auch [Fayad+ 2007e]).

1.1.4.4 Dokumentationsunschärfen

Die Rollendokumentation innerhalb von Mustern ist kritisch zu betrachten. Es existiert kein verbreiteter Mechanismus der vorgibt, wie Rollen in Mustern zu dokumentieren sind. [Fayad 2007d] benennt ein Beispiel, das die Problematik veranschaulicht. Ein Muster, das zum Entwurf eines Geschäfts zur Computerreparatur geeignet ist, führt die Klasse Computer als Entität ein, die in Ordnung zu bringen ist. Das Muster ist somit auf den ersten Blick übertragbar auf ein Krankenhaus, wo statt Computern Patienten in Ordnung gebracht werden sollen. Diese auf Analogien basierende Gleichsetzung führt im weiteren Implementierungsverlauf wahrscheinlich zu Problemen, da Maschinen und Menschen grundlegend unterschiedliche Eigenschaften besitzen.

Ob ein Muster ein Entwurfsmuster, Architekturmuster, Analysemuster oder (nur) eine Refactoring-Operation ist, lässt sich aufgrund nur unscharf vorhandener Definitionen nicht befriedigend beantworten (siehe auch [Fayad 2007b] und Abschnitt 2.2.1).

1.1.4.5 Musterauswahl

Die Auswahl eines Musters ist insbesondere deshalb intellektuell anspruchsvoll und zeitaufwändig, weil eine formalisierte Auswertung der informell dokumentierten Muster nicht möglich erscheint. Ein Muster alleine anhand seines Namens auszuwählen oder zumindest in Erwägung zu ziehen, ist oft unzuverlässig und unbefriedigend (siehe auch [Fayad 2007f]).

Der Nutzen von Entwurfsmustern wird dadurch abgeschwächt, dass es einer intensiven Einarbeitung bedarf, bis ein Entwickler Muster nutzvoll einsetzen kann ([Fowler 2003b]). Die falsche Auswahl eines Musters kann verheerende Folgen haben.

1.1.4.6 Musteranwendung

Entwurfsmuster sind eine Form der Komplexität, die der Möglichkeit einer einfacheren Lösung entgegenstehen kann.
Das häufige Verwenden einer bestimmten Menge an Basismustern in unterschiedlichen Softwareprojekten kann ein Hinweis dafür sein, dass die verwendete Programmiersprache das eigentliche Problem ist, weil sie die Lösung häufig vorkommender Entwurfsprobleme durch Sprachkonstrukte nicht ausreichend unterstützt.

Die Aussage, Entwurfsmuster erleichterten die Entwicklung von Software, ist laut [Sailer 2002], [Zimmer 1997], [Baroni+ 2003b] und anderen nicht allgemeingültig. Beispielsweise wird in [Topley 2004] die Frage gestellt:

„Weren't object-oriented programming, design patterns and frameworks supposed to make life simpler for the poor sod writing the code?"

John Topley

Die Anwendung eines Musters aus einer abstrakten Beschreibung, der Musterdokumentation, heraus ist per se nicht eindeutig möglich. Der Musteranwender ist vielmehr gefordert, eine konkrete, bisher wahrscheinlich unbekannte Implementierung für sein spezielles Problem zu finden. Dieses Problem befindet sich zwar im abstrakten Kontext des Musters, wodurch das Muster letztendlich für das Problem geeignet erscheint. Die Implementierung der dynamischen Musterteile ist aber für einen speziellen Anwendungsfall prinzipiell nicht an sich bekannt. Beispielhafte Implementierungen für ein beispielhaft gewähltes konkretes Problem, die in der Musterdokumentation angeführt sind, helfen zwar das Verständnis für Implementierungsansätze zu entwickeln; sie stellen aber eben nur Beispiele dar und keine Anleitungen (vgl. auch [Fayad+ 2007e]).

Für das Kombinieren oder Verbinden mehrerer Muster zu einer Architektur existiert kein etablierter formaler Prozess. Vielmehr findet die Musterauswahl und Anwendung nach Erfahrung des Autors und nach [Fayad+ 2007c] ad hoc statt. Der unterschiedliche Abstraktionsgrad von Mustern erschwert eine Kombination (siehe auch [Fayad+ 2007c]).

1.1.4.7 Mustererkennung nach deren Anwendung

Nach Anwendung eines Musters sind die Musterbestandteile im Quelltext formal nur unter größeren Schwierigkeiten oder mit Unschärfen als solche identifizierbar. Die einzige Möglichkeit in einer herkömmlichen Entwicklungsmethodik ist die informelle Auswertung von Kommentaren oder die formelle Suche nach sprechenden Namen[9]. Eine erfolgreiche Identifikation von Musterbestandteilen ist insbesondere dann essentiell, wenn ein Software-System erweitert oder angepasst werden soll. Die Unfähigkeit der formalen Analyse erhöht die Kosten der Identifikation offensichtlich (siehe auch [Fayad 2007d] und [Fayad 2007e]).

1.2 Motivation der Arbeit

Entwurfsmuster vermitteln Expertenwissen zur Bewältigung schon gelöster Probleme. Die Vorteile von Mustern sind vielfältig, insbesondere helfen sie bei der Erstellung wartbarer Software. Aber gerade die Auswahl und Anwendung eines Musters ist nicht einfach und erfordert vom Entwickler ein hohes Problembewusstsein. Hohe technische Anforderungen an das Abstraktionsvermögen des Entwicklers können zu Hemmschwellen in der kommerziellen Software-Entwicklung führen. Nach Erfahrung des Autors ist es besonders in kleinen und mittelständischen Unternehmen schwierig, Überzeugungsarbeit zu leisten und Entwurfsmuster in den allgemeinen Entwicklungsprozess einzuführen (siehe auch die Fragebogenergebnisse in Absatz 1.1.3, die auf signifikant unterschiedliche Wahrnehmungen von Entwurfsmustern hindeuten). Schulungen für technisches Personal sind mit hohen Kosten verbunden und werden oft nicht genehmigt. Die Einsicht zu technischen Schulungen ist seit dem Zusammenbruch des Neuen Marktes Anfang des Jahrtausends im Vergleich zu den Jahren davor an sich zurückgegangen.[10]

Es stellt sich die Frage, wie die bei der Entwurfsmusterauswahl und -anwendung genannten Schwierigkeiten reduziert werden können. Der Entwickler kann bei der Anwendung von Entwurfsmustern in sinnvoller Weise durch ein Software-Werkzeug entlastet werden. Eine Entlastung des Entwicklers sollte insbesondere bei den Aktivitäten

---

[9] Vgl. auch [Pree 1995, S. 117]: „Recall the guideline that names should not only be semantically expressive but also consistent throughout a framework."; Vgl. auch Obfuskatoren wie [Zelix 2003]

[10] In etwa seit 2006 konnte nach Beobachtung des Autors wieder ein Aufschwung verzeichnet werden.

- einschränken in Frage kommender Muster,
- auswählen eines geeigneten Musters,
- abfragen von dem Muster anhaftenden Parametern,
- bestimmen des genauen Kontextes für die Implementierung des ausgewählten Musters,
- implementieren bzw. Einweben des Musters selbst und
- prüfen von Nachbedingungen nach der Musteranwendung

ermöglicht werden.

Aufgrund der Schwierigkeiten bei der Arbeit mit Entwurfsmustern ist daher in Erwägung zu ziehen, das klassische Vorgehen bei der Musterauswahl und -anwendung durch einen werkzeuggestützten Ansatz zu ergänzen.
Die Arbeit mit Mustern kann auf verschiedene Arten geschehen, etwa abstrakt auf der Modellebene (oft UML), der generativen Ebene oder konkret im Quelltext. Diese Arbeit fokussiert sich auf Quelltext. Allgemein ist es möglich, sich zusätzlich der Betrachtung des Modells eines Programms zu widmen, was aufgrund der Komplexität in dieser Arbeit allerdings vernachlässigt wird.

Die für diese Arbeit gestellte Forschungsfrage, die sich aus den dargestellten Problemen und der Motivation dieser Arbeit ableitet, lautet:

Forschungsfrage:

> Wie kann der Entwickler bei der Selektion und Anwendung von Mustern auf Quelltext durch Anreicherung des Quelltextes mit Zusatzinformationen unterstützt werden?

Wie in Abschnitt 1.1.4.3 beschrieben, existiert ein Spannungsfeld zwischen informaler Musterdokumentation und formalem, vorliegendem Quelltext. Diese Arbeit soll einen Beitrag dazu leisten, dieses Spannungsfeld zu reduzieren durch Einführung einer Mediatorschicht als Hilfestellung für den mit Entwurfsmustern arbeitenden Menschen:

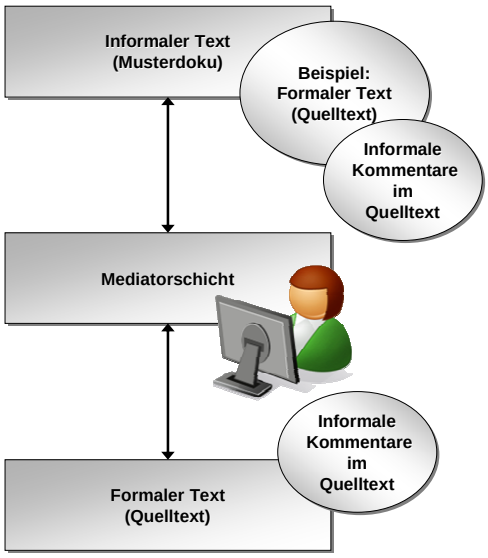

Abbildung 6: Eine Mediatorschicht als Brücke zwischen Musterdokumentation und Quelltext.

Im Gegensatz zu Abbildung 5 soll der Mensch durch den Beitrag dieser Arbeit bei der Arbeit mit Mustern unterstützt und entlastet werden. Dies soll durch die Einführung einer formalen Musterdokumentation geschehen, im Rahmen derer die Lösungsideen von zu dokumentierenden

Mustern in formaler Form erfasst werden. Weiterhin soll für die Musterselektion und -anwendung, die auf der formalen Musterdokumentation basieren sollen, ein Mechanismus zum Anbringen formalisierter Aussagen über Entwicklerintentionen im Quelltext entwickelt werden.

Im Weiteren wird Java als beispielhafte Sprache für die Vorstellung des Ansatzes verwendet. Auch bei der Auswahl bestehender Ansätze (siehe Kapitel 2) wurde der Fokus der Betrachtung auf Java gelegt. Java ist eine moderne und zugleich sehr populäre Sprache. Andere Ansätze wie [Robillard 2003] sind ebenfalls auf Java ausgerichtet. Die Wahl von Java bedeutet keine Einschränkung hinsichtlich der Allgemeinheit des Ansatzes, da beispielsweise die Sprache C# von Microsoft sehr ähnlich zu Java ist und ebenfalls hätte verwendet werden können.

1.3 Gliederung der Arbeit

Die Arbeit ist in vier Kapitel gegliedert.

Kapitel eins ist der Teil bis hierher. Er leitet in die Arbeit ein, vermittelt anhand einer Fragebogenauswertung die Wahrnehmung der Befragten von Entwurfsmustern und definiert die Problemstellung.

Kapitel zwei diskutiert den Stand der Dinge zum Thema Entwurfsmuster. Im Rahmen dessen werden Begriffsdefinitionen durch Rekapitulation vorgenommen sowie bestehende Arbeiten vorgestellt und kritisch gewürdigt. Weil es einerseits sehr viele Ansätze zur Arbeit mit Entwurfsmustern gibt und andererseits einige Ansätze bereits betrachtenswerte Lösungselemente beinhalten, fällt die Betrachtung etwas ausführlicher aus. Diese Arbeit sieht den formalen Ausdruck von Musterintentionen zur Behandlung der Forschungsfrage als signifikant an. Dem wird durch Vorstellung bisheriger Ansätze zu diesem Thema sowie durch Erarbeiten einer Lösung, die auf Semantikinformationen in Quelltexten beruht, Rechnung getragen.
Abschließend wird die Problemstellung unter Berücksichtigung des zuvor dargestellten Stands der Technik sowie der kritischen Würdigung verfügbarer Arbeiten verfeinert. Die für diese Arbeit interessanten Lösungselemente werden herausgestellt.

Kapitel drei beschreibt einen neuen Ansatz zur möglichen Lösung der verfeinerten Problemstellung. Ziel dieses Kapitels ist die Darstellung der Lösungsidee sowie der damit verbundenen Möglichkeiten. Im Rahmen dessen wird die Tauglichkeit des Ansatzes für die Musterdokumentation, Musterselektion sowie Musteranwendung demonstriert und eine Möglichkeit benannt, wie der Ansatz in eine Entwicklungsumgebung integriert werden kann.

Kapitel vier würdigt das Verfahren kritisch und dokumentiert die Vor- und Nachteile des vorgestellten Ansatzes. Anschließend wird die Arbeit zusammengefasst, ein Fazit gezogen sowie ein Ausblick auf zukünftig relevante Aspekte gegeben.

KAPITEL 2

2 Stand der Technik

2.1 Einleitung

Bei der Arbeit mit Entwurfsmustern lassen sich mehrere Schritte unterscheiden. Grundlegend ist die Beschreibung des Musters, die auch Musterdokumentation genannt wird. Synonym für die Dokumentation eines Musters werden auch die Begriffe Musterdefinition oder Musterbeschreibung verwendet. Dadurch kann das Muster in einem Katalog (engl.: *Repository*) gespeichert und später wieder abgerufen werden. Bei der Musterauswahl (auch als Musterselektion bezeichnet) werden für einen gegebenen Programmkontext nun aus einer Menge von Mustern anhand deren Musterdokumentationen die Muster ausgewählt, die für eine Anwendung im Kontext in Frage kommen. Die selektierten Muster werden im Rahmen der Musteranwendung für den gewählten Programmkontext durch Ausimplementierung konkretisiert und dabei mit dem Programmtext verwoben.

Die formale Dokumentation von Mustern ist Voraussetzung für den Ansatz dieser Arbeit, um Muster werkzeuggestützt selektieren und anwenden zu können. Nur mit Hilfe entsprechend definierter Musterdokumentationen kann festgestellt werden, welche Muster für eine Anwendung zu selektieren sind. Die Musterselektion ist das eigentliche Ziel des hier vorgestellten Ansatzes. Die Musteranwendung hängt eng mit der Musterselektion zusammen. Ebenso basiert sie auf der Musterdokumentation. Dementsprechend müssen existierende Ansätze, die auf die genannten Aspekte Musterdokumentation, Musterauswahl und Musteranwendung ausgerichtet sind, berücksichtigt werden.

Für das Hinzufügen von Semantikinformationen zu einem Programm ohne letzteres im Ablauf zu verändern ist eine entsprechende Möglichkeit zu finden. Auch hierzu werden bestehende Ansätze untersucht.

Für notwendige Betrachtungen wird zusätzlich der Aspekt des Reverse Engineering von Mustern berücksichtigt. Die weitergehende Betrachtung von Ansätzen zu dieser Thematik kann Impulse bei der Entwicklung des hier vorgestellten, neuen Ansatzes bringen. Insbesondere kann durch Betrachten von Ansätzen aus dem Bereich Reverse Engineering abgeleitet werden, welche Charakteristika eines Entwurfsmusters signifikant sind. Das Erkennen von schon im ursprünglichen Quelltext angewandten Mustern kann die Qualität der Musterselektion positiv beeinflussen. Denn ein bereits vollständig oder teilweise vorhandenes Muster kann nicht erneut vollständig angewandt werden. Weiterhin können die Schritte Musterselektion und -anwendung insofern als ähnlich angesehen werden als dass sie Musterteile identifizieren. In der Musterselektion sind dies Teile, die als anwendbar angesehen werden, in der Mustererkennung Teile, die bereits angewandt sind.

In jeder Kategorie werden aufgrund der Vielzahl verfügbarer Ansätze nur einige stellvertretend vorgestellt. Jeweils genannte Musterbeispiele entsprechen Beispielen aus den vorgestellten Ansätzen selbst. Die vorgestellten Ansätze sind stellvertretend für eine Vielzahl nicht untersuchter. Keiner der nicht betrachteten Ansätze, die der Autor vorab aussortiert hat, beinhaltete eine Lösungsidee, die dem Ansatz dieser Arbeit vergleichbar wäre. Die diskutierten Ansätze wurden nach intensiver Suche aus verschiedenen Gründen berücksichtigt. Erstens, weil sie oft referenziert wurden (wie etwa [Eden+ 2004]). Zweitens, weil sie Konzepte verwenden, die für den Ansatz dieser Arbeit betrachtenswert erscheinen (etwa [MacDonald+ 2002]). Drittens, weil sie durch ihre Herangehensweise Impulse für die Zielstellung dieser Arbeit vermitteln können (etwa [Mayr-Kern+ 2002]. Viele Ansätze können mehreren der genannten Kategorien zugeordnet werden. Weiter unten im Kapitel wird eine Übersicht über die Aspekte gegeben, die als positiv oder negativ hinsichtlich ihrer Eignung zur Lösung der Zielstellung erscheinen.

Nicht betrachtet werden Verfahren, die auf graphischen Notationen von Programmstrukturen beruhen. In diese Kategorie fällt alles, was beispielsweise die UML als Darstellungsmechanismus verwendet. Diese Arbeit beschränkt sich auf die Betrachtung textueller Programmrepräsentationen. Graphische Notationen sind nur in ihrer textuellen Repräsentation maschinenverarbeitbar. Graphische Notationen können Informationen besitzen, die nicht im Quelltext zu finden sind. Dies ist bei der UML etwa bei Beziehungen zwischen Klassen der Fall. Die Referenzen 0..n, 0..1 und 1..n einer Klasse auf eine andere können etwa nicht in jedem Fall über eine Quelltextanalyse unterschieden werden. Quelltext hingegen besitzt die gesamte Kodierung des Programms, die erforderlich ist, um das Programm zu kompilieren bzw. zu interpretieren und es danach auszuführen. Diagramme können nur Quelltextschnipsel direkt visualisieren oder einbinden. Sie besitzen eine höhere Abstraktionsebene als Quelltext. Da in dieser Arbeit die Implementierungsteile eines Entwurfsmusters von tragender Bedeutung sind, liegt der Fokus auf Programmtext, nicht auf den abstrakteren Modellen. Unterstützend sei [Fowler+ 2000, S. 29] zitiert:

> „Die UML ermöglicht es, einen objektorientierten Entwurf auszudrücken. Entwurfsmuster (engl. *design patterns*) betrachten dagegen die Ergebnisse des Prozesses, nämlich Beispielmodelle."
>
> **Martin Fowler**

Im nächsten Absatz werden grundlegende Begriffsdefinitionen vorgenommen, bevor im Absatz 2.3 vorhandene Arbeiten diskutiert werden.

2.2 Begriffsdefinitionen

Dieser Abschnitt rekapituliert Begriffsdefinitionen, die nach dem heutigen Stand verbreitet und für diese Arbeit relevant sind.

2.2.1 Muster

Die vorliegende Arbeit hat das Ziel, die Arbeit mit Entwurfsmustern zu unterstützen. Alle zu Entwurfsmustern qualitativ vergleichbaren Konzepte sollen ebenfalls durch diese Arbeit abgedeckt werden. Zusammenfassend, wird der Begriff »Muster« gebraucht, der alle in diesem Abschnitt 2.2.1 durch Definitionen herausgestellten Begriffe umfasst.

Unter den Begriff »Muster« fallen in der Software-Entwicklung primär die Begriffe »Entwurfsmuster« und »Architekturmuster«. Die Definition des Begriffs »Entwurfsmuster« wurde bereits in Abschnitt 1.1 vorgenommen und lautet:

Definition: Entwurfsmuster

> Ein Entwurfsmuster ist eine (bewährte) Lösung für ein allgemeines Entwurfsproblem in einem bestimmten Kontext.

Entwurfsmuster wurden insbesondere durch [Gamma+ 1995] populär (vgl. auch die Fragebogenauswertung in Abschnitt 1.1.3). [Hruschka+ 2002] ordnet Entwurfsmustern als Zweck den „[…] Bau der technischen Softwarearchitektur und ihrer Komponenten[…]" zu.

Der Begriff des Architekturmusters, der durch [Buschmann+ 1998] bekannt wurde, ist nicht scharf von dem des Entwurfsmusters abgrenzbar[11]. Vielmehr sind die Übergänge vom weniger komplexen Entwurfsmuster zum komplexeren Architekturmuster fließend. Die Definition für den Begriff »Architekturmuster« ist daher schwierig und kann nur unscharf vorgenommen werden:

Definition: Architekturmuster

> Ein Architekturmuster ist eine bewährte Lösung für ein allgemeines Architekturproblem in einem bestimmten Kontext. Es beschreibt den komplexen Aufbau eines Anwendungsteils, der die Gesamtanwendung maßgeblich beeinflusst. Architekturmuster sind auf höherer Abstraktionsebene angesiedelt als Entwurfsmuster und nicht scharf von diesen abgrenzbar.

Architekturmuster werden laut [Hruschka+ 2002] hauptsächlich „[…] zur Strukturierung, Gliederung und Stabilisierung der grundlegenden, fachlichen Softwarearchitektur […]“ verwendet.
Ein Beispiel für ein Architekturmuster ist *MVC* (Model-View-Controller). Es gibt allerdings keine vollständige Übereinstimmung, ob *MVC* noch ein Entwurfs- oder schon ein Architekturmuster ist. Es wird nach Ansicht des Autors und laut Fragebogenauswertung (Abschnitt 1.1.3) öfter zu den Architekturmustern gezählt. Häufig wird *MVC* auch als Hybrid bezeichnet (siehe ebenfalls Fragebogenauswertung).

Muster treten oft in einer zur ursprünglichen Dokumentation abgewandelten Form auf. Das angewandte Muster ist dann mitunter nicht gleichwertig mit dem dokumentierten. Das ist insbesondere dann der Fall, wenn etwa die dem Muster zugrunde liegende Klassenhierarchie eine andere Form hat. Eine Gleichwertigkeit läge vor, wenn die Abstraktion des angewandten Musters gleich der (abstrakten) Musterdokumentation ist. Von einer Variation kann gesprochen werden, wenn ein Muster eine leicht andere Zielstellung hat als die dokumentierte Ursprungsform des Musters. Außerdem darf die Variation nicht durch die Beschreibung eines anderen Musters erfasst werden können. Gemäß der Fragebogenauswertung aus Abschnitt 1.1.3 sind potentiell alle Teile eines Musters variierbar. Häufig variieren Methodennamen oder Klassennamen eines Musters. Je nach Ausprägung eines Musters kann auch die Struktur anders gestaltet sein, ohne die Funktionalität des Musters zu verändern.

Definition: Mustervariation

> Eine Variation eines Musters liegt vor, wenn entweder eine verhaltensgleiche Form in anderer Implementierung gegeben ist oder wenn eine das Wesen des ursprünglichen Musters nicht verändernde Abwandlung an einer oder mehreren Eigenschaften des Musters vorgenommen wurde. Nicht verändernde Abwandlungen sind insbesondere isomorphe Ersetzungen der Ursprungsform.
>
> Synonym wird der Begriff »Mustervariante« verwendet.

Die in der Definition genannten isomorphen Ersetzungen werden im späteren Abschnitt 3.2.3 ausführlich diskutiert. Beispiel für eine Mustervariation ist eine Erweiterung des *Observer*-Musters durch eine Verwaltungsinstanz, die die Benachrichtigung der registrierten Beobachter steuert anstatt dass die Beobachter direkt und synchron benachrichtigt werden. Diese Variation fällt in die in der Definition zur Mustervariation als zweites genannte Kategorie. Ein Beispiel für die erstgenannte Kategorie ist die Implementierung der Schleife zur Benachrichtigung aller Beobachter. Die

[11] Die in Abschnitt 1.1.3 genannte Befragung hat zum Ergebnis, dass die überwiegende Mehrheit der Befragten eine Trennung zwischen Architektur- und Entwurfsmustern entweder nicht für sinnvoll hält oder selbst nicht vornehmen kann.

Implementierung kann etwa in der Programmiersprache Java durch eine *for*-Schleife oder durch Verwendung des *Iterator*-Musters umgesetzt werden. Zur weiteren Ableitung von Mustervariationen siehe Abschnitt 3.3.3.4 und folgende.

2.2.2 Musterähnliche Konzepte

Den Entwurfs- und Architekturmustern stellt [Hruschka+ 2002] die Idiome bei. Idiome sieht die genannte Quelle „[…] zur Optimierung und Gestaltung von Codesequenzen.“. Die vollständige Definition des Begriffes *Idiom* soll hier wie folgt angegeben werden:

Definition: Idiom

> Ein Idiom ist eine sprachabhängige, bewährte Lösung für ein spezielles Entwurfsproblem in einem bestimmten Kontext.

Die ebenfalls in [Hruschka+ 2002] genannten Analysemuster (vgl. [Fowler 1998]) dienen laut [Hruschka+ 2002] „[…] zur Erstellung von Anforderungsdokumenten mit integrierten Analysemethoden.”. Analysemuster sind daher nicht für diese Arbeit relevant, da sie nicht signifikant auf der Implementierungsebene operieren. Nichtsdestotrotz werden Analysemuster gelegentlich mit Entwurfsmustern in Beziehung gesetzt (vgl. Abschnitt 1.1.3).

Ergänzend zu [Hruschka+ 2002] sollen im Rahmen dieser Arbeit auch Methoden des Refactorings zu den Mustern zählen. Refactoring ist eine Tätigkeit innerhalb des Reengineerings. Beispiele für eine Refactoring-Operation sind das Verschieben einer vorhandenen Methode in eine andere Klasse, das Hinzufügen einer Attributdeklaration zu einer Klasse oder das Umbenennen einer Methode. Der Begriff »Refactoring-Operation« wird in Anlehnung an [Fowler 2001] wie folgt definiert:

Definition: Refactoring-Operation

> Eine Refactoring-Operation ist eine maschinell durchführbare, feingranulare Operation, die auf einem kompilierbaren Quelltext ausgeführt wird. Der Quelltext wird dabei so transformiert, dass er danach wieder kompilierbar ist. Eine Refactoring-Operation wird auf einem bestehenden Quelltext zur Verbesserung von dessen innerer Struktur ausgeführt, ohne das erkennbare Verhalten des Programms zu verändern.
>
> Synonym wird der Begriff »Refactoring« verwendet.

Die Definition von [Fowler 2001] fordert keine Maschinenverarbeitbarkeit, ansonsten ist die hier gegebene Definition vergleichbar.

Refactoring-Mechanismen sind im Sinne der Definition vergleichbar zu behandeln wie Entwurfsmuster oder Idiome. Tätigkeiten im Refactoring unterscheiden sich von der Anwendung von Entwurfsmustern in mehrerlei Hinsicht. Refactoring wird im Rahmen von Code Reviews durchgeführt. Entwurfsmuster können darüber hinaus auch während der Entwicklung außerhalb von Code Reviews angewendet werden. Refactoring-Mechanismen sind punktueller als Entwurfsmuster, deren Granularität gröber ist. Refactoring ist insbesondere dann notwendig, wenn Design-Defekte im kleinen Maßstab zutage treten. Beispielsweise dann, wenn eine Methode zu viele Anweisungen beinhaltet oder wenn deren Parameterliste zu lang wird. Entwurfsmuster hingegen werden bei weniger leicht zu identifizierenden Entwurfsdefekten oder bei wachsenden Anforderungen verwendet. Im Falle des Musters *Bridge* (siehe [Gamma+ 1995]) genau dann, wenn es unterschiedliche Implementierungen für ein Problem geben kann, die unabhängig von einander und gegeneinander austauschbar sind. Im Gegensatz zur Anwendbarkeit von Entwurfsmustern

unterstützen gängige Entwicklungsoberflächen wie Eclipse, NetBeans, Borland JBuilder und IntelliJ IDEA die automatische Durchführung von Refactoring-Mechanismen.

Beispielsweise bewirkt die Regel *Introduce Parameter Object* die Konsolidierung von Eingabeparametern mit dem Effekt, deren Liste später einfacher ändern zu können (in Hinblick auf Anzahl und Typ). Die Definition der Regel lautet (vgl. [Fowler 2001]):

Definition: Introduce Parameter Object (Refactoring-Regel)

> Ersetze innerhalb einer Methodensignatur eine Gruppe zusammengehöriger Eingabeparameter durch ein einzelnes Parameterobjekt.

Weiterhin sind bewährte Praktiken (*Best Practices*) zu nennen, die ähnlich den Idiomen sind. Idiome sind jedoch immer sprachabhängig, *Best Practices* nur auf der konkretisierten (beispielhaften) Ebene. Sofern das Konzept einer *Best Practice* mit dem mehrerer Sprachen übereinstimmt, lässt sie sich auf eben diese Sprachen anwenden. Beispiel für eine solche gute Praxis: Vermeide leere Fehlerbehandlungsblöcke. Konkretisiert auf eine Sprache wie Java oder C++ bedeutet das, in einem *catch*-Block auf jeden Fall eine Anweisung vorzusehen. Ist jedoch der leere Block gewünscht, sollte zur Kennzeichnung der Intention des Entwicklers die leere Anweisung (früher: *NOP* für *No Operation*) „implementiert“ werden. Der Entwickler würde also ein Semikolon setzen. Dies verhindert bei der Anwendung von Prüfwerkzeugen wie *PMD* [PMD 2003] oder *PatternTesting* [Boehm 2004] das Entstehen von lästigen, fallspezifisch unangebrachten Regelverletzungen (siehe auch [Meffert 2006b] für eine weiter führende Betrachtung hierzu).

Minipatterns sind nach [Ó Cinnéide+ 1999a] und [Ó Cinnéide+ 1999b] Konzepte zur feingranularen formalen Dokumentation von Mustern. Sie haben Ähnlichkeit mit Refactoring-Operationen. [Ó Cinnéide+ 1999a] setzt Muster ebenso in Verbindung zueinander wie dies [Buschmann+ 1998] tut und zitiert [Alexander 1979] mit den Worten: „Jedes Muster hängt von den kleineren Mustern ab, die es enthält, und den größeren Mustern, in denen es enthalten ist.“

Quelltexttransformationen sind Operationen zur Überführung eines gegebenen Quelltextes in einen anderen Quelltext. Diese Transformationen können in unterschiedlichen Kontexten ausgeführt werden, insbesondere im Rahmen des Refactoring oder der Musteranwendung.

Definition: Quelltexttransformation

> Eine Quelltexttransformation ist die Überführung eines kompilierbaren Ausgangsquelltextes in einen kompilierbaren Zielquelltext durch Verändern des Ausgangsquelltextes aufgrund einer bestimmten Motivation.
>
> Synonym wird der Begriff »Transformation« verwendet.

Zur Veranschaulichung sei eine Quelltexttransformation an einem Beispiel genannt. Gegeben ist folgender Auszug eines Ausgangsquelltextes (Abbildung 7):

```
public class MyClass {
  private int x;
  ...
  public void berechne(int y) {
    x = y + z;
    ...
    int a = x + 1;
    ...
  }
}
```

Abbildung 7: Beispielhafter Ausgangsquelltext für eine Transformation.

Der Quelltext wird nun derart refakturiert, dass der Zugriff auf die private Klassenvariable *x* durch einen sogenannten *Getter* bzw. *Setter* geschieht. Das Ergebnis der Transformation ist in der Abbildung 8 zu sehen:

```
public class MyClass {
  private int x;
  ...
  public int getX() {
    return x;
  }
  public void setX(int x1) {
    x = x1;
  }
  public void berechne(int y) {
    setX(y + z);
    ...
    int a = getX() + 1;
    ...
  }
}
```

Abbildung 8: Ergebnisquelltext nach beispielhafter Transformation.

Die Transformation vom Ausgangs- in den Ergebnisquelltextes besteht aus folgenden Operationen:

- Einführen einer *Set*-Methode für Variable *x*
- Einführen einer *Get*-Methode für Variable *x*
- Ersetzen aller Lesezugriffe auf *x* durch *getX()*
- Ersetzen aller Zuweisungen an *x* durch *setX(<Ausdruck>)*

Durch Betrachten von Ausgangs- und Ergebnisquelltext (Abbildung 7 und Abbildung 8) kann festgestellt werden, dass beide Quelltexte verhaltensgleich sind. Die Transformation war diesbezüglich also verhaltenskonservierend. In einem übergeordneten Kontext betrachtet, ist die Transformation allerdings nicht verhaltenskonservierend. Denn durch Einführung der Methode namens *setX* ist es für andere Klassen möglich, die Variable *x* in Klasse *MyClass* direkt zu modifizieren.

Eine Transformation beinhaltet nicht selten korrelierte Elemente. So bedingt das Einführen der Methoden *setX* und *getX* in Konsequenz die Ersetzung des Zugriffs auf Variable *x* durch die Aufrufe eben dieser Methode. Diese Art von Korrelation hat auch [Robillard 2003, S. 78] festgestellt.

2.3 Untersuchte Ansätze

Im Folgenden werden ausgewählte Arbeiten diskutiert, die sich mit der formalen Dokumentation, Selektion, Anwendung und Erkennung von Mustern sowie der Anreicherung von Quelltext mit Metainformationen beschäftigen.
Neben diesen Kernaspekten der vorliegenden Arbeit wird auch die Mustererkennung betrachtet. Sie hängt mit den Kernprozessen wie folgt zusammen. Die Musterselektion kann nur Muster auswählen, die nicht schon angewandt wurden und greift dazu auf die Musterdokumentation zurück, in der Erkennungslogiken vorhanden sein können. Weiterhin deuten nur teilweise in einem Quelltext vorhandene Muster darauf hin, dass der Rest des Musters möglicherweise anzuwenden ist. Die Musteranwendung muss die bereits angewandten Musterteile berücksichtigen.
Ansätze die die Anreicherung von Quelltext mit Metainformationen erlauben, werden betrachtet, weil die vorliegende Arbeit auf Metainformationen im Quelltext beruht.
Die vorgestellten Ansätze wurden aus dem verfügbaren Material erstens unter dem Gesichtspunkt der Popularität und zweitens unter dem Aspekt der Verfügbarkeit brauchbarer Dokumentation ausgewählt. Ein drittes Kriterium war die Beurteilung des Ansatzes im Vorfeld hin auf die Möglichkeit, geeignete Lösungsansätze zur Ideengebung für diese Arbeit zu finden.
Wichtig bei der Auswahl der vorzustellenden Ansätze war die Berücksichtigung verschiedenartiger Lösungsansätze, um eine breite Grundlage für die Entwicklung eines eigenen Ansatzes zu schaffen. Zielstellung war es, bereits erarbeitete Konzepte für die Musterbeschreibung zu diskutieren, um sie in den hier vorgestellten Ansatz bei Eignung zu integrieren.

2.3.1 Formale Dokumentation von Entwurfsmustern

Wie bereits in Abschnitt 1.1.4.3 erwähnt, ist die Musterdokumentation die Beschreibung eines Musters, die nach formalem und informalem Charakter unterschieden werden kann.
Die informale (natürlichsprachliche) Musterdokumentation ist die ursprünglich gewählte Form, wie sie etwa in [Gamma+ 1995] zu finden ist. Empfängerkreis für die informale Musterdokumentation sind Entwickler, Architekten und andere mit Mustern beschäftigte Personen.
Die informale Musterdokumentation enthält alle für das Verständnis des betreffenden Musters wesentlichen Informationen. Häufig werden diese Informationen in die Abschnitte Problembeschreibung, Kontext oder Anwendungsgebiet, Empfängerkreis, Lösungsvorschlag, Konsequenzen, Varianten und verwandte Muster unterteilt.
Mit Hilfe einer informalen Musterdokumentation wird der Leser befähigt, ein Muster zu verstehen. Er sollte dann für ein gegebenes Problem entscheiden können, ob das Muster zu diesem Problem passt. Weiterhin sollte er mit Hilfe der Musterdokumentation in der Lage sein, das Muster anzuwenden. Für einen gegebenen Quelltext, für den das Muster bereits angewandt wurde, sollte er die Existenz des Musters erkennen können, wenn die Musteranwendung vergleichbar der in der Dokumentation ist.

Definition: Informale Musterdokumentation

> Als informale Musterdokumentation wird in dieser Arbeit die Beschreibung eines Musters verstanden, wie sie durch Musterbücher wie [Gamma+ 1995] oder [Buschmann+ 1998] populär wurde.

Die informale Musterdokumentation wird häufig von Personen erstellt, die das zu beschreibende Muster sehr gut kennen und es bereits selbst angewendet haben (vgl. Abschnitt 1.1.3). Informale Beschreibungen können von maschinengestützten Algorithmen heutzutage nicht in ausreichendem Maße interpretiert werden. Ein Grund ist die zu informelle und gleichzeitig sehr komplexe, weil Ausnahmen zulassende Grammatik, der beispielsweise die Sprache Deutsch zugrunde liegt.

Die formale Dokumentation eines Musters wird im Rahmen dieser Arbeit auch als Musterdefinition bezeichnet. Einen Standard für die formale Dokumentation eines Musters gibt es nach Information des Autors nicht (vgl. Abschnitt 1.1.3). Es gibt keine Form, die weit verbreitet ist und zudem geeignet ist, ein Muster derart einer Maschine verständlich zu machen, dass sie geeignete Kontexte für das Muster identifizieren, das Muster in weiten Teilen anwenden oder seine frühere Anwendung erkennen kann. Es kann nicht erwartet werden, dass eine Maschine diese Schritte ohne Zutun eines Entwicklers durchführt. Jedoch ist eine nach Meinung des Autors gerechtfertigte Erwartungshaltung, dass der Entwickler bei der Arbeit mit Mustern wesentlich durch Werkzeuge unterstützt werden kann und zwar in einer Art, die durch die Problemstellung in Absatz 1.1 charakterisiert ist und die im nachstehenden Absatz 2.4 verfeinert wird. Die nachfolgend vorgestellten Ansätze beschäftigen sich allesamt mit der Thematik wie die Arbeit mit Mustern durch eine formale Musterdokumentation unterstützt werden kann.

Definition: Formale Musterdokumentation

Als formale Musterdokumentation wird in dieser Arbeit die Beschreibung eines Musters verstanden, die genügend eindeutige und maschinenverarbeitbare Informationen enthält, um einem bestimmten Zweck (Musterselektion, Mustererkennung, Musteranwendung) zu erfüllen. Enthalten sind insbesondere Informationen zur Struktur und Bedeutung des Musters.

Eine formale Musterdokumentation kann auch mehrere Zwecke gleichzeitig erfüllen und enthält dann dementsprechend mehr Informationen.

Synonym wird der Begriff »formale Musterdefinition« verwendet.

Die Dokumentation von Entwurfsmustern gestattet – je nach Formalisierungsgrad – deren Ablage und programmtechnische Verarbeitung, etwa für die Musterselektion oder -anwendung. Grundsätzlich lassen sich die vorgestellten Ansätze anhand des Grades ihrer Formalität unterscheiden. Je formaler eine Beschreibung, desto leichter ist sie maschinenverarbeitbar, aber umso schwieriger ist sie für einen Menschen handhabbar. Halbformale Ansätze unterliegen zwar ebenfalls Formalismen, diese sind maschinell jedoch nicht auswertbar. Vorteil dieser Ansätze ist die leichtere Anwendbarkeit und Nachvollziehbarkeit für den Menschen. Aus diesem Grund werden Ansätze mit unterschiedlich formalem Charakter vorgestellt. Formale Sprachen gehören zur Gruppe der streng formalen Beschreibungsmittel. Im Folgenden werden dazu zwei häufiger zitierte Ansätze vorgestellt. Abschließend folgt ein Fazit zu formalen Sprachen.
Da die vorgestellten Ansätze unterschiedliche Absichten mit der Beschreibung von Mustern verfolgen, wird – wenn dies der eben getroffenen Definition entgegensteht – anstatt von formaler Musterdokumentation von Spezifikation gesprochen.

2.3.1.1 Balanced Pattern Specification Language

Mit der *Balanced Pattern Specification Language* (kurz: *BPSL*) stellt [Taibi+ 2003] eine formale Beschreibungssprache für Entwurfsmuster vor. Die *BPSL* ist speziell auf Entwurfsmuster ausgerichtet. Sie wurde nicht für Analyse-, Architektur- oder Organisationsmuster entwickelt. Die Motivation zur Einführung der *BPSL* war die Unvollständigkeit bisheriger Musterbeschreibungssprachen ([Taibi+ 2003, S. 127]). Der Grund für die Unvollständigkeit bisheriger Verfahren wird in der ausschließlichen Fokussierung entweder auf Verhaltens- oder stattdessen auf Strukturaspekte gesehen [ebd.]. Die *BPSL* will sich beiden Aspekten gleichzeitig widmen. Dafür wurden zwei Komponenten eingeführt. Mit der *First Order Logic* (*FOL*) können strukturelle Aspekte spezifiziert werden. Sie basiert auf Prädikatsausdrücken. Die *Temporal Logic of Actions* (*TLA*) hingegen ist auf die Beschreibung von Verhaltensaspekten ausgerichtet. Die *TLA* eignet sich für die formale Definition von Kooperationen zwischen Objekten. Dazu gehören auch Zustandsänderungen von Variablen und zeitlich begrenzte Objektbeziehungen.

Die *BPSL* heißt balanciert, weil die beiden genannten Aspekte in sich gegenseitig ergänzender Weise unterstützt werden sollen. Sie basiert auf drei Entitäten (vgl. [Taibi+ 2003, S.129f]):

1. Primäre Entitäten: Klassen (kurz: C), Attribute (A), Methoden (M), Objekte (O) und untypisierte Werte (V). Sie werden als nichtreduzierbare Einheiten angesehen.
2. Relationen: Permanente oder zeitlich begrenzte Kollaboration zwischen Entitäten.
3. Aktionen: Atomare Ausführungseinheiten, die mehrere Objekte berühren.

Zwischen den primären Entitäten werden folgende Beziehungen definiert (Tabelle 1):

Name der Beziehung	Beschreibung
Defined-in	Methode, die in einer bestimmten Klasse definiert ist.
	Oder: Attribut, das in einer bestimmten Klasse definiert ist.
Reference-to-one (-many)	Eine Klasse definiert ein »Mitglied« über eine Variable, deren Typ eine Referenz zu einer (mehreren) Instanz(en) der zweiten Klasse ist.
Inheritance	Eine Klasse erbt von einer zweiten.
Creation	In einer Methode wird eine Instanz einer Klasse erzeugt.
	Oder: Eine der Methoden der ersten Klasse erzeugt eine Instanz der zweiten Klasse.
Invocation	Eine Methode ruft eine andere auf.
Argument	Eine Referenz zur Klasse ist ein Argument der Methode.
	Oder: Ein untypisierter Wert ist ein Argument der Methode.
Instance	Ein Objekt ist eine Instanz einer bestimmten Klasse.

Tabelle 1: Beziehungen zwischen Entitäten (nach [Taibi+ 2003, S. 131]).

Mit Hilfe dieser Beziehungen sollen Entwurfsmuster formal spezifiziert werden. Das *Observer*-Muster ist in [Taibi+ 2003, S. 136] auf diese Weise beispielhaft beschrieben. Ein Auszug aus dem Beispiel sei hier wiedergegeben:

$\exists$ subject, concrete-subject, observer, concrete-observer $\in$ C
subject-state, observer-state $\in$ A;
attach, detach, notify, get-state, set-state, update $\in$ M;
o, s $\in$ O;
d $\in$ V;
Defined-in(subject-state, concrete-subject) $\wedge$
Defined-in (observer-state, concrete-observer) $\wedge$
Defined-in(attach, subject) $\wedge$
Defined-in (detach, subject) $\wedge$
Defined-in (notify, subject) $\wedge$
[…]
Attached(concrete-subject[0..1],concrete-observer[*]) $\wedge$
Updated(concrete-subject[0..1],concrete-observer[*]).
Initially: $\neg$Attached(s, concrete-observer).
Attach(s,o) : $\neg$Attached(s,o) $\rightarrow$ Attached'(s,o) $\vee$
[…]

Abbildung 9: Spezifikation des Observer-Musters mit der *BPSL* (Auszug aus [Taibi+ 2003, S. 136]).

Die zu Beginn der dargestellten Spezifikation verwendeten Symbole *A*, *M*, *O* und *V* repräsentieren Entitäten und sind weiter oben erklärt.

Bei Betrachten der Spezifikation aus Abbildung 9 sind mehrere Dinge erkennbar:

1. Die Spezifikation eines einzelnen Musters besteht aus vielen Elementen (einige davon wurden der Übersichtlichkeit halber oben nicht aufgeführt).
2. Die Spezifikation wird vorgenommen durch Verwenden mathematischer Symbole wie ∃ für »es existiert« oder die Symbole für Konjugation und Disjunktion.
3. Die Spezifikation hat einen formalen Charakter, der nicht vergleichbar mit dem Charakter einer modernen Programmiersprache wie Java oder C# ist.

Fazit

Insgesamt erscheinen formale Beschreibungen mit der *FOL* und *TLA* für einen Software-Entwickler nicht leicht verständlich. Die gegensätzliche Ansicht in [Taibi+ 2003, S. 139] ist wahrscheinlich dadurch zu begründen, dass die Autoren eine andere Zielgruppe als den durchschnittlichen Entwickler ansprechen, nämlich Personen, die fähig und willens sind, streng formale Spezifikationen vorzunehmen. Aufgrund der starken Formalität der *BPSL* ist sie im Rahmen einer leicht handhabbaren und für den Entwickler brauchbaren Entwurfsmuster-Dokumentation nicht einsetzbar. Vielleicht gewinnt die *BPSL* in Verbindung mit einer vereinfachenden Makrosicht oder einer graphischen Oberfläche in Zukunft an Bedeutung.
Jedoch ist das Konzept der Entitäten und deren Beziehungen untereinander ein nennenswerter Ansatz zur Analyse eines Programmtextes. Die gleichwertige Betrachtung von Struktur und Verhalten eines Musters, wie sie die *BPSL* fordert und ermöglicht, ist bei der formalen Dokumentation von Mustern zu berücksichtigen. Denn Strukturaspekte sind entscheidend für die Definition des Grundgerüstes eines Musters. Verhaltensaspekte charakterisieren die Lösungsabsicht hinter dem Muster. Zusammen mit einem dritten Aspekt, dem der Anwendbarkeit des Musters, bilden sie die Sichten auf ein Muster. Die Anwendbarkeitssicht ist nicht expliziter Bestandteil der *BPSL*.
Die eben genannten Konzepte, die die BPSL zur formalen Dokumentation von Mustern anwendet, erscheinen für diese Arbeit lohnenswert. Ein zu starker Formalisierungsgrad der formalen Musterdokumentation sollte vermieden werden, um den Ansatz für Software-Entwickler praktikabel zu halten.

2.3.1.2 LePUS

LePUS[12] aus [Eden+ 2004] ist eine deklarative Sprache zur Spezifikation von Mustern. Weil *LePUS* auf die formale Beschreibung der Lösungen fokussiert ist, die ein Entwurfsmuster anbietet, wurde der vom Entwurfsmuster bedeutungsdifferente Begriff der *Lattices* geprägt. Ein *Lattice* wird in [Eden+ 2004, S. 3] definiert als eine Menge von Qualifikationen bezüglich der strukturellen, verhaltensbasierten und relationalen Aspekten von Programmen. Die Prämissen beim Entwurf von *LePUS* sind laut [Eden+ 2004]:

1. Wohldefinierte Semantik.
2. Kompaktes Set an Symbolen.
3. Sparsame Einführung von Erweiterungen zu bestehenden Notationen.
4. Angemessen abstrakt und fähig, die *Lattices* möglichst vieler Entwurfsmuster zu beschreiben.
5. Generisch verwendbar.
6. Strukturell, nicht verhaltensorientiert.
7. Unterstützung einer graphischen Notation.

Um die Schwierigkeit der Arbeit mit einer formalen Deklarationssprache zu verringern, wurde für *LePUS* eine graphische Repräsentation der Deklarationssprache entwickelt.

Als Beispiel sei die graphische Notation des *Observer*-Musters angegeben (Abbildung 10):

[12] LePUS = LanguagE for Pattern Uniform Specification

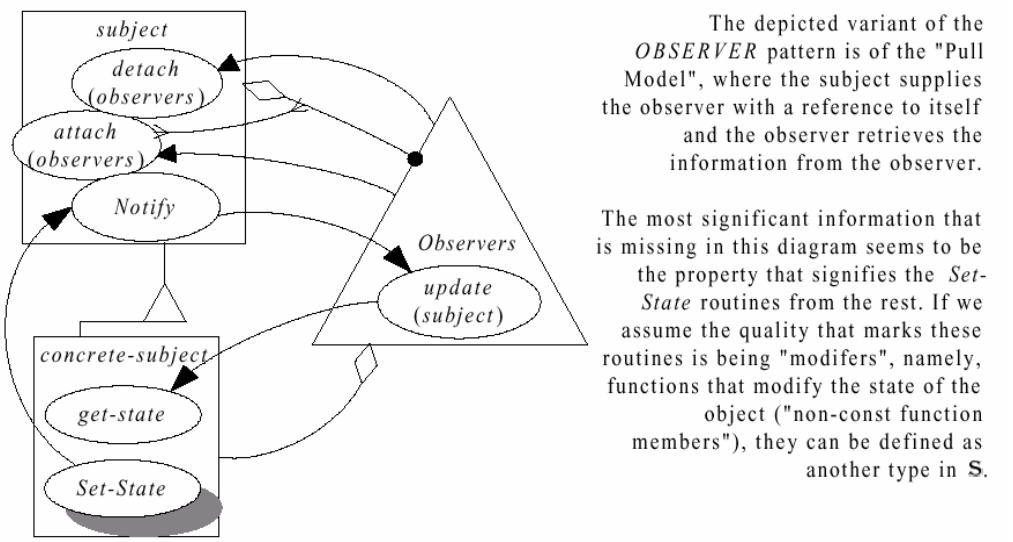

Abbildung 10: Observer-Muster in graphischer LePUS-Notation (aus [Eden+ 2004, S. 29]).

Das Dreieck in der obigen Graphik steht dafür, dass alle *Observer*-Klassen von einer gemeinsamen abstrakten Klasse abgeleitet sind (siehe [Eden+ 2004, S. 16]). Das Rechteck repräsentiert eine Klasse. Die Ellipse steht für eine Funktion. Eine Ellipse mit Schatten stellt eine Menge von Funktionen dar. Die Bedeutung der übrigen Symbole aus der obigen Graphik ist in der nächsten Abbildung 11 dargestellt:

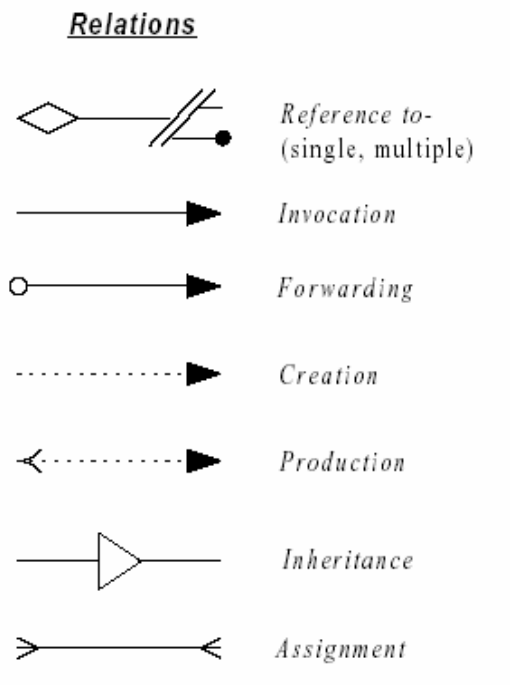

Abbildung 11: Bedeutung von Symbolen in LePUS (aus [PBE 2004a, S. 11]).

Mit der Relation *Reference to* wird ausgedrückt, dass eine Klasse eine oder mehrere andere Klassen referenziert/verwendet. Die Relationen *Invocation*, *Forwarding*, *Creation* und *Production* aus Abbildung 11 beziehen sich auf Funktionen. Die Relation *Inheritance* bedeutet, eine Klasse ist abgeleitet von einer anderen. Die Relation *Assignment* kann verschiedene Bedeutungen haben. Sie beschreibt im obigen Beispiel, dass eine Funktion eine Klasse (in Form eines Objektes) an eine andere übergibt.

Auf ein Beispiel einer formalen Spezifikation mit *LePUS* soll hier wegen der vorigen Vorstellung der *BPSL* verzichtet werden. Grund ist die Ähnlichkeit der beiden formalen Sprachen.

Fazit

LePUS gewichtet Strukturaspekte höher als Verhaltensaspekte. Das steht im Gegensatz zur *BSPL*. [Eden+ 2004] beschreibt, dass *LePUS* nicht in der Lage sein kann, alle Entwurfsmuster bzw. *Lattices* von Entwurfsmustern formal zu beschreiben. Als Beispiele werden die Muster *Iterator* und *Memento* angegeben.
Die unterschiedliche Gewichtung von Struktur und Verhalten ist diskussionswürdig. Zumindest stellt sich die Frage, ob beide Aspekte mit gleichen Mitteln behandelt/definiert werden sollen oder nicht. Die graphische Notation von *LePUS* kann hilfreich sein bei der Modellierung von Abhängigkeiten zwischen Klassen in einem Muster. Sie wird im Rahmen dieser Arbeit als optionale Komponente gesehen, die den Komfort für den Bearbeiter erhöhen kann, aber nicht essentiell ist. Im Gegensatz zu *LePUS* soll diese Arbeit prinzipiell alle Entwurfsmuster unterstützen.

2.3.1.3 Fazit zu formalen Sprachen

Formale Beschreibungssprachen für Entwurfsmuster haben einen anderen Fokus als eine Methodik zur formalen Dokumentation von Mustern in dieser Arbeit. Eine formale Sprache zielt stärker auf die streng automatisierte Musterselektion ab. Die Idee der formalen Sprache ist die exakte und vollständige Beschreibung eines Musters im Hinblick auf sein Verhalten und seine Struktur. Sinnvolles Anwendungsgebiet formaler Sprachen kann die Abdeckung eines Sets von Standardmustern, etwa der 23 *GoF*-Muster, sein. Die formale Musterdokumentation ist erstens mit formalen Sprachen (ohne Zusätze) nicht trivial und zweitens nicht generisch genug, um Variationen hinreichend erkennen zu können.
Diese Arbeit hat hingegen das Ziel, eine für den Software-Entwickler möglichst einfach verständliche formale Dokumentation von Mustern zu ermöglichen. Weiterhin sollen alle Entwurfsmuster abgedeckt werden, was mit den vorgestellten formalen Sprachen alleine nicht zu gelingen scheint. Denn eine Semantik kann mit formalen Sprachen wie *LePUS* (vgl. die Aussage der Autoren selbst [Eden+ 2004, S. 31]) nicht vollständig abgebildet werden.

2.3.1.4 Meta-Patterns

Im Gegensatz zu den eben vorgestellten Konzepten, die formale Sprachen darstellen, handeln die Arbeiten [Pree 1994] und [Pree 1995] von sogenannten *Meta-Patterns*. *Meta-Patterns* dienen der Veranschaulichung von Entwurfsabsichten in Frameworks[13] [Pree 1995, S. 221f]. Pree stellt sieben *Meta-Patterns* vor. Beschreibungen von Mustern, wie sie etwa in [Gamma+ 1995] vorliegen, sind nach [Pree 1994] ausbaufähig (im Sinne von »formalisierbarer«). *Meta-Patterns* erlauben eine abstrakte Beschreibung von Mustern.
Als Hauptanwendungsgebiet werden *Template*- und *Hook*-Methoden angegeben. *Template*-Methoden stellen sogenannte *Frozen Spots* dar. Das sind unveränderliche Stellen, die durch *Hook*-Methoden flexibilisiert werden können. *Hook*-Methoden repräsentieren *Hot Spots*. *Hot Spots* sind veränderbare Punkte innerhalb der Domäne eines Frameworks – also des Framework-Anwendungsbereiches – an denen der Anwender sein System anpassen kann. Diese kritischen Punkte sollen durch den Einsatz von *Meta-Patterns* entschärft werden. Darauf aufbauend kann mit Hilfe der *Meta-Patterns* eine Datenbank mit Beispielen erstellt werden [Pree 1995, S. 167f]. Für jeden *Hot Spot* in einem Framework kann der Entwickler, inspiriert von Beispielen aus der Datenbank, leichter günstige Modifikationen vornehmen. Die dem Entwickler für einen *Hot Spot* präsentierten

[13] [Pree 1995] setzt Frameworks teilweise mit Entwurfsmustern gleich, siehe [Pree 1995, S. 115]: „This sample framework is called Factory Method in the design pattern catalog of Erich Gamma *et al.* (1994).“. In [Pree 1995, S. 121] wird vom „MVC framework“ und vom „State framework“ (gemeint sind die Muster *MVC* und *State*) gesprochen. Siehe auch [Pree 1995, S. 159] für weitere Gleichsetzungen der Begriffe Framework und Entwurfsmuster.

Beispiele werden aufgrund von im Framework angebrachten *Meta-Patterns* ausgewählt. [Pree 1995, S 170] weist darauf hin, dass *Meta-Patterns* nur auf bewährte Frameworks angewandt werden sollten. Ist diese Bedingung erfüllt, tragen *Meta-Patterns* zur Verdeutlichung der Entwurfsabsichten eines Frameworks bei [Pree 1995, S. 179], [Pree 1995, S. 221f]. Die in [Pree 1994] genannten *Meta-Patterns* für Komposition sind (Abbildung 12):

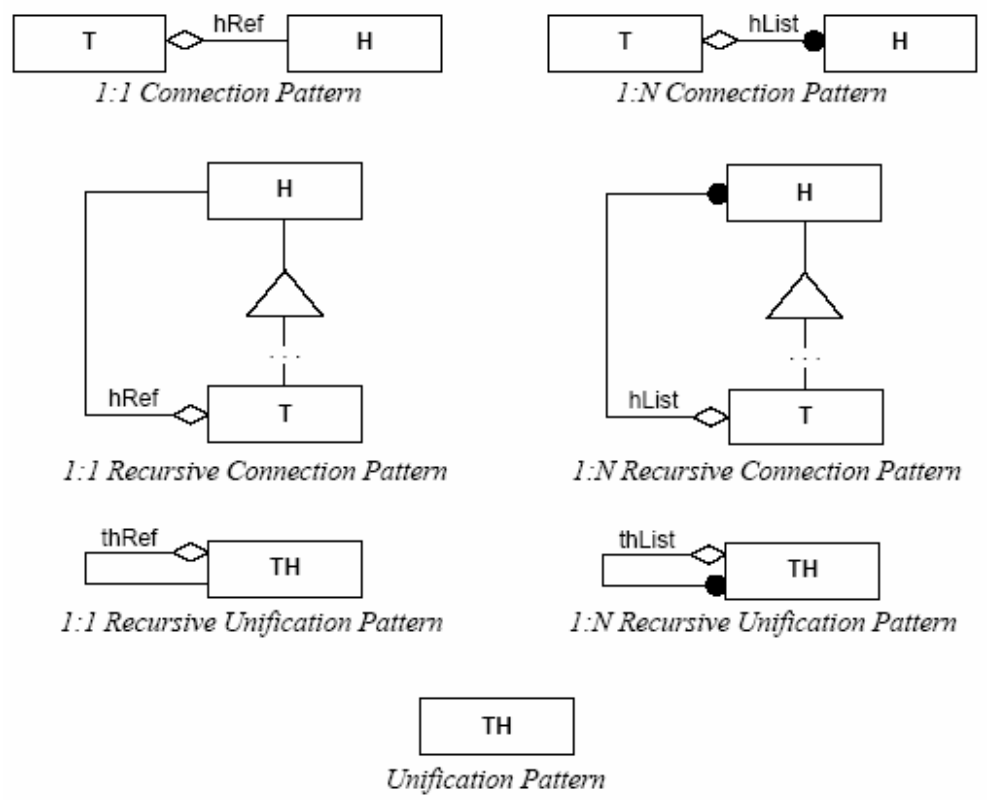

Abbildung 12: Meta Patterns für die Komposition (aus [Pree 2004, S. 8]).

Sechs der oben dargestellten sieben *Meta-Patterns* enthalten zu Beginn ihrer Bezeichnung eine Wertigkeitsangabe. Die linke Seite der Wertigkeit gibt die Multiplizität einer *Template*-Methode an. Die rechte Seite bezieht sich auf die Multiplizität der aufgerufenen *Hook*-Methoden. Zur Veranschaulichung seien hier die *Meta-Patterns* kurz beschrieben:

Das *Unification Pattern* besagt, dass es genau eine *Template*- und eine *Hook*-Methode gibt. Beide befinden sich in derselben Klasse. Dieses Muster bietet sich dann an, wenn die *Hook*-Methode eine Art ausimplementiertes Standardverhalten darstellt, welches wahrscheinlich nicht vom Anwender geändert werden muss. Trifft diese Annahme nicht zu, muss der Anwender zur Modifikation eine neue Klasse implementieren, welche von der *Unification* Klasse erbt. Das folgende Beispiel in Abbildung 13 zeigt, wie die Notation eines *Unification Meta-Patterns* aussehen kann:

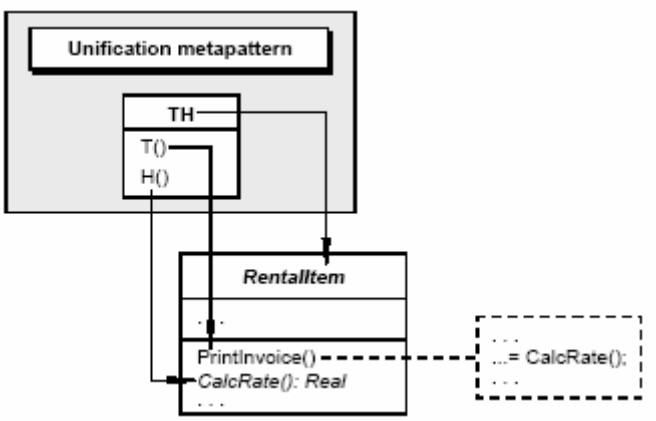

Abbildung 13: Beispiel für ein *Unification Meta-Pattern* (aus [Pree 2004, S. 9]).

Wie zu erkennen ist, existiert eine Klasse *RentalItem*, welche die *Template*-Methode *PrintInvoice* enthält. Letztere ruft innerhalb ihrer Programmlogik die *Hook*-Methode *CalcRate* auf. Ihre Logik kann so an das jeweilige fachliche Umfeld angepasst werden.
Die Notwendigkeit der Subklassenbildung behebt das *1:1 Connection Pattern*. Hier befinden sich *Template*- und *Hook*-Methode in unterschiedlichen Klassen. Das *1:N Connection Pattern* erlaubt pro *Template*-Methode beliebig viele *Hook*-Methoden, die in jeweils unterschiedlichen Klassen liegen. Ob das *1:1 Connection Pattern* oder *1:N Connection Pattern* sinnvoll ist, hängt alleine von den Anforderungen an das Programmdesign ab.
Im Rahmen der *Recursive Connection Patterns* repräsentieren *Template*-Methoden gleichzeitig *Hook*-Methoden, indem sie als Unterklasse der *Hook*-Klasse implementiert werden. Anwendbar sind *Recursive Connection Patterns* insbesondere, wenn ein Netzwerk von Objekten modelliert werden soll, die einheitlich behandelt werden können. Allerdings erhöht sich die Komplexität des Entwurfs bei Anwendung dieses Musters.
Recursive Unification gibt es in zwei Alternativen und ist eine Abwandlung von *Recursive Connection*. Unterschied ist, dass *Template*- und *Hook*-Methode wie beim *Unification Pattern* in derselben Klasse liegen. Dadurch ist die Struktur flexibler, da jede Klasse durch jede andere aufgrund der Typkompatibilität ersetzt werden kann.

Fazit

Meta-Patterns von Pree eignen sich nur für den spezialisierten Einsatz in Frameworks, insbesondere für *Template*- und *Hook*-Methoden. Gerade diese beiden Methodenarten sind in Frameworks, die in ihrer Gestalt generisch sind, vertreten und weniger in Programmcode. Die *Meta-Patterns* können als einfache und unvollständige Ontologie für Entwurfsmuster interpretiert werden. Eine solche Ontologie ist zur Orientierung für den Entwickler sicher hilfreich. Andererseits steigt die Komplexität der formalen Musterdokumentation durch *Meta-Patterns* an. Es ist unklar, welche Entwurfsprobleme sich mit *Meta-Patterns* lösen lassen. Fest steht, dass keine vollständige Abdeckung vorliegt, da mit den genannten *Meta-Patterns* nur bestimmte Strukturen und kein Verhalten ausgedrückt werden kann. Weiterhin ist die Definition von *Meta-Patterns* keine Aufgabe für den durchschnittlichen Software-Entwickler. Das Hilfsmittel der Beispieldatenbank als informaler Mechanismus steht im Gegensatz zu den formalen *Meta-Patterns*. Es wird bezweifelt, ob ein durchschnittlicher Entwickler diesem Paradigmensprung folgen kann und will.
[Pree 1995, S. 114] selbst nennt eine weitere, mit *Meta-Patterns* verbundene Problematik: Klassen sollen demnach nach dem *Narrow Inheritance Principle* entworfen werden. Dieses Prinzip besagt, dass ein Verhaltensaspekt so kompakt wie möglich in einer Klasse gehalten werden soll, um in einer abgeleiteten Klasse mit so wenigen Überschreibungen wie möglich diesen Verhaltensaspekt nach Bedarf anpassen zu können (siehe [Pree 1995, S. 113]).
Das Konzept der *Frozen Spots* und *Hot Spots* kann gemäß [Pree 1995] wie folgt für diese Arbeit übernommen werden: Innerhalb eines Programmtextes existieren bezüglich Entwurfsmustern statische und dynamische Programmteile. Die statischen Teile sind vom Einweben eines

Entwurfsmusters unabhängig, die dynamischen Teile hingegen abhängig. Sie werden durch Annotationen gekennzeichnet. Innerhalb eines Entwurfsmusters ist der Sachverhalt ähnlich: Es existieren statische Teile, die unveränderbar implementiert werden können sowie dynamische Teile, die an die Domäne angepasst werden müssen und durch Annotationen identifiziert werden.

2.3.1.5 PSE-Definition

PSE ist die Abkürzung für *Pattern Support for Eclipse*. *PSE*-Definition (kurz: *PSE-D*) [Wöckl 2002] ist Teil von *PSE*-Base. *PSE*-Base ist ein *Eclipse*[14]-Plugin. Es stellt eine technologische Basis dar, die die Darstellung und Dokumentation von Entwurfsmustern erlaubt. Es stellt Schnittstellen bereit, an die konkrete Werkzeuge angebunden werden können. *PSE*-Base ist Teil von *PSE*.
PSE-D dient der Spezifikation und Verwaltung von Entwurfsmustern mitsamt deren textueller Dokumentation. Die Musterdokumentation orientiert sich an gängigen Notationen und beinhaltet Sektionen wie »Absicht«, »Synonym«, »Motivation«, »Beispielcode« oder »verwandte Muster«.
Für jedes Muster werden die dazu gehörigen Rollen definiert. Jede Rolle wird in einer eigenen XML-Datei abgelegt. Die folgende Abbildung 14 stellt das beschriebene Szenario dar:

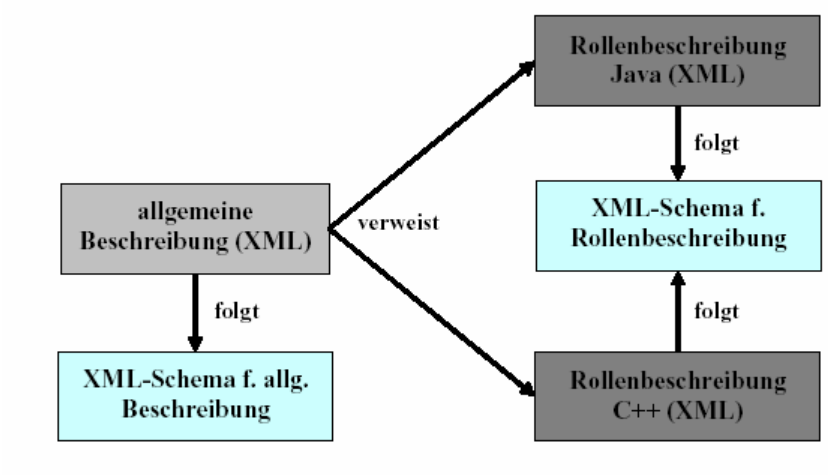

Abbildung 14: Musterverwaltung mit *PSE-D* (aus [Wöckl 2002, S. 32]).

Für jede Rolle werden deren Typ (Schnittstelle oder Klasse) sowie der Rahmen (Code-Kopf und -Fuss) angegeben. Daraus kann *PSE-D* den Mustercode erzeugen. Beispielsweise sieht die Definition der Rolle *AbstractProduct* des Entwurfsmusters *Abstract Factory* so aus wie in Tabelle 2 gezeigt:

Eigenschaft	Ausprägung
Name	AbstractProduct
Beschreibung	Definiert eine Schnittstelle für ein Produkt
Typ	Interface
Codekopf	public interface AbstractProduct{}
Codekörper	
Häufigkeit	1

Tabelle 2: Beschreibung einer Rolle in PSE-D (aus [Wöckl 2002, S. 31]).

Die Eigenschaft »Häufigkeit« gibt an, wie oft eine Rolle bei der Musteranwendung auftreten kann. Hier sind sowohl ganze Zahlen als auch geschlossene und offene Intervalle möglich.

[14] *Eclipse* ist eine Entwicklungsumgebung für Java. Sie nutzt das Konzept der *Plugins* extensiv aus. In *Eclipse* ist alles ein *Plugin*. Dementsprechend erweiterbar ist *Eclipse*.

PSE-D stellt eine Schnittstelle bereit, über die andere Anwendungen auf die Muster-Infrastruktur zugreifen können.

Fazit

Ziel beim Entwurf von *PSE-D* war insbesondere die Unabhängigkeit von Programmiersprachen [Wöckl 2002, S. 33]. Eine Schnittstelle nach außen erlaubt die Nutzung von *PSE-D* als Werkzeugsatz (engl.: *Toolkit*). Die Definition der Rollen eines Musters ist im Hinblick auf Codekopf und -körper textorientiert. Der Benutzer erhält keine graphische Unterstützung. Die Musterbeschreibung erscheint nicht besonders leistungsfähig. Es ist nicht ersichtlich, wie die Intention eines Musters mit *PSE-D* beschrieben werden kann. Lediglich eine textuelle Beschreibung der Intention ist möglich. *PSE-D* orientiert sich zwar bei der Definition von Rollen eines Musters am Quelltext, indem der Entwickler eben diesen zu spezifizieren hat. Allerdings trägt dieses Vorgehen nach Einschätzung des Autors nicht zur Entlastung des Entwicklers bei. Der Entwickler muss nämlich die statischen und dynamischen Programmteile der Rollen eines Musters quasi ohne Hilfsmittel identifizieren und definieren.
Die Konzepte Programmiersprachenunabhängigkeit und Schnittstellenbereitstellung eignen sich prinzipiell zur Berücksichtigung bei der Entwicklung des hier vorgestellten Ansatzes.

2.3.1.6 PADL

Die auf Java zugeschnittene *PADL*[15] [Baroni+ 2003b] ermöglicht die Beschreibung von Entwurfsmustern mit Hilfe eines Metamodells. Mit der *PADL* können Programme und Entwurfsmotive (engl.: *Design motifs*) beschrieben werden. Die *PADL* ist eine Erweiterung der Vorgängerversion *PDL*[16], die derselbe Autor entwickelt hat und die durch die *PADL* obsolet geworden ist. Als Entwurfsmotive bezeichnet [Baroni+ 2003b] eine Repräsentation der Struktur sowie eine Untermenge des Verhaltens einer Lösung eines Entwurfsmusters. Entwurfsmotive können obligatorische Schnittstellen oder Methoden einer Klasse sein, ebenso wie die Art der Beziehung zwischen zwei Klassen.
Grund für die Zerlegung von Mustern in Entwurfsmotive ist die leichtere Identifizierung dieser in einem Programmtext. Denn je feingranularer ein Teil ist, desto größer die Wahrscheinlichkeit, es in einem Kontext wieder zu finden. Auf diese Weise kann die Übereinstimmung einer bestehenden Architektur mit einem in der *PADL* definierten Muster ermittelt werden. Die Erkennung der Entwurfsmotive soll die verbesserte Dokumentation und die automatisierte Anwendung weiterer, zu einem Muster gehöriger und noch fehlender Entwurfsmotive ermöglichen.
Die PADL stellt konkret ein Plugin für *Eclipse* bereit. In der *PADL*-Implementierung sind bereits einige der *GoF*-Muster beispielhaft enthalten. Die Spezifikation des *Composite*-Musters sieht mit der *PADL* folgendermaßen aus (Abbildung 15 nach [Guéhéneuc 2004], Teile der Übersichtlichkeit wegen weggelassen):

[15] PADL = Pattern and Abstract-level Description Language
[16] PDL = Pattern Description Language

```
public class Composite extends StructuralPatternModel {
      public Composite(final IListener patternListener) {

      final IInterface iComponent =
            getFactory().createInterface("Component");
      final IMethod mOperation = getFactory().createMethod("operation");
      iComponent.addActor(mOperation);
      addActor(iComponent);

      final IComposition aComposition =
            getFactory().createCompositionRelationship("children",
                        iComponent, 2);

      cComposite.addImplementedEntity(iComponent);
      cComposite.addActor(aComposition);

      final IDelegatingMethod mDelegation =
            getFactory().createDelegatingMethod("operation", aComposition);

      mDelegation.attachTo(mOperation);
      cComposite.addActor(mDelegation);
      addActor(getFactory().createClass("Composite"));
      addLeaf(new String[] { "Leaf" });
 }

 public void addLeaf(final String[] names) {
      final IClass aClass = this.getFactory().createClass(names[0]);

      aClass.addImplementedEntity((IInterface)
             getActorFromName("Component"));
      aClass.assumeAllInterfaces();
      addActor(aClass);
 }

}
```

Abbildung 15: Spezifikation des *Composite*-Musters mit Hilfe der *PADL*.

Mit Hilfe eines graphischen Werkzeuges, das *PatternsBox* genannt wird, kann eine Meta-Beschreibung wie die obige generiert werden. Aus dieser Meta-Beschreibung generiert *PatternsBox* Quelltext. Die Anwendung eines mit der *PADL* definierten Musters geschieht durch Aufruf von eigens dafür bereitgestellten Geschäftsmethoden der *PADL*.

Im Beispiel ist die strukturartige Spezifikation eines Musters erkennbar. Die dem Muster anhaftenden Lösungsabsichten werden über zuvor zu definierende Geschäftsmethoden angegeben. Sie repräsentieren die Entwurfsmotive. Oben genannte Geschäftsmethoden für Entwurfsmotive sind etwa *createInterface*, *addActor* oder *createCompositionRelation*.

Fazit

Die *PADL* beschreibt die Struktur eines Musters und dessen Lösungsabsicht in einer gemeinsamen Spezifikation. Sie vermischt diese beiden Aspekte durch Einführung des Entwurfsmotivs als beide Sichten abdeckendes Konzept. Die *PADL* eignet sich nicht dazu, direkt dem Entwickler zur Hand gegeben zu werden. Grund ist der abstrakte Charakter, der auf die Einführung einer Meta-Ebene begründet ist und die Vermischung der Aspekte Struktur und Lösungsabsicht eines Musters. Quelltext wird im Rahmen der *PADL* nicht als Entität erster Klasse angesehen (engl.: *First-class citizen*).

Die Idee der Entkopplung der Spezifikationserstellung von der internen Spezifikationsbeschreibung mit Hilfe eines Werkzeugs ist ein für die vorliegende Arbeit sinnvolles Konzept. Es entbindet den Anwender davon, die der Spezifikation zugrunde liegende Sprache lernen zu müssen.

2.3.1.7 Der Generator ANGIE

Mit *ANGIE* [ANGIE 2003] liegt ein Frame-basiertes Generatorsystem vor. Es beinhaltet eine eigene Scriptsprache als Grundlage für den Generationsprozess. Die dem Autor verfügbare Testversion *ANGIE Generation Now!* leistet folgendes: Für jede beliebige Sprache lässt sich ein Quelltext vorgeben (die in der Testversion mitgelieferten Beispiele beziehen sich auf Java und C#). Im Quelltext können Sprachelemente (Anweisungen) ausgezeichnet werden. Der Benutzer kann variable Elemente (engl.: *Slots*) innerhalb eines logischen Rahmens, der als Frame bezeichnet wird, definieren. Es ist dann möglich, alle einen Slot betreffenden Teile, die normalerweise dieselbe Deklaration besitzen, durch einen später als konkreten Wert einfließenden Parameter zu ersetzen. Slots können entweder skalare Werte oder Verweise auf Sub-Frames enthalten. Neben generischen Elementen können Slots auch Defaultwerte zugewiesen werden. Weiterhin lässt sich über Constraints steuern, ob eine Definition innerhalb eines Frames aktiv geschaltet werden soll. Weiter unten ist hierfür ein Beispiel angegeben. Die Idee der Frames und Slots geht auf Minskys Aufsatz »A Framework for Representing Knowledge« aus dem Jahre 1974 (nachgedruckt in [Haugeland 1997, S. 111-142]) zurück. Minsky bezeichnet das Konzept der rekursiven Definition von Strukturen (wie dies etwa bei Sub-Frames der Fall ist), als Mikrowelt. Minsky definiert einen Frame so [Haugeland 1997, S. 111f]:

> „A frame is a data structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed."
>
> **Marvin Minsky**

Slots werden bei Minsky als Terminale einer niedrigeren Ebene bezeichnet [Haugeland 1997, S. 112]. Vergleichbar mit dem von [Alexander+ 1995] eingeführten Mustersystem, definiert Minsky Frame-Systeme. Das sind Sammlungen ähnlicher Frames. Verschiedene Frames eines Systems können dieselben Terminale nutzen. Auch die Möglichkeit der Default-Belegung, wie sie in *ANGIE* zu finden ist, hat Minsky bereits beschrieben („A frame's terminals are normally already filled with 'default' assignments." [Haugeland 1997, S. 112]).

Ein Beispiel für die Verwendung von Slots: Für ein *Singleton* wird der Rückgabetyp für die Methode *getInstance()* als Slot deklariert. Der Klassenname wird ebenfalls diesem Slot zugeordnet. So ist die synchrone respektive konsistente Ersetzung einer Menge Elementplatzhalter durch das einzusetzende Element sichergestellt. Durch die Angabe von *Insertion Points* hat man die Möglichkeit, die bestehenden Strukturen zu erweitern. Das ist deshalb von Bedeutung, da bei einem generativen Ansatz alle manuellen Änderungen verloren gehen, die nicht separat als solche kenntlich gemacht werden.

Zur Verdeutlichung ist das folgende vereinfachte Exempel gedacht, das der Dokumentation von *ANGIE* entnommen wurde. Es handelt sich um die dynamische Definition eines Integer-Elements. Die vorangestellten Ziffern markieren die Zeilennummer:

```
1 .Frame GenNumberElement(Name, MaxValue)

2a .Dim vIntQualifier = (MaxValue > 32767) ? "long" : "short"
2b .Dim sNumbersInitVal

3a // The following Number Element "<!Name!>" is generated
3b // by the frame "GenNumberElement"

4  <!vIntQualifier!> int <!Name!><? = <!sNumbersInitVal!>?>;

5 .End Frame
```

Abbildung 16: Frame-Definition eines Integer-Elements mit ANGIE (nach [ANGIE 2003]).

In Zeile 1 wird der Frame-Header deklariert. In der Deklaration sind der Name des Frame-Typs (*GenNumberElement*) sowie die beiden Konstruktor-Parameter (*Name*, *MaxValue*) enthalten.
Dann folgen in Zeile 2a und 2b zwei Variablendeklarationen. Die erste legt den Variablentyp dynamisch fest. Die dynamische Deklaration bedingt, dass die Variable *vIntQualifier* vom Typ *long* ist, wenn der Wert des Eingabeparameters *MaxValue* größer als 32767 ist und ansonsten vom Typ *short*. Die statische Deklaration in Zeile 2b bezieht sich hier auf ein nicht näher spezifiziertes Element, das außerhalb des Geltungsbereichs dieses Frames initialisiert worden sein kann.

Die Zeilen 3a und 3b beinhalten den zu generierenden Quelltext. In diesem Fall handelt es sich um einen Kommentar. Innerhalb des Kommentars befindet sich die Steueranweisung »<!Name!>«. Die ersten und letzten beiden Zeichen signalisieren *ANGIE* eine eingebettete Variable.

In Zeile 4 findet sich eine kontextabhängige Anweisung (optionaler Codeblock), die durch die Steuerzeichen »<?« und »?>« gekennzeichnet wird. Es handelt sich hierbei um eine Variablendeklaration mit Zuweisung. Die Zuweisung wird nur vorgenommen, wenn die Zuweisungsvariable *sNumbersInitVal* nicht initial, sondern mit einem Wert belegt ist. Das ist ein Beispiel für bedingte Generierung, die durch deklarative Markierungen umgesetzt wird Im Gegensatz dazu stehen prozedurale *If*-Anweisungen, die lokal implementiert sind.

Fazit

Die mit *ANGIE* vorliegende Technologie setzt das Konzept der Frames und Slots von Minsky um. Sie gestattet die Erstellung von Vorlagen im Rahmen eines generativen Prozesses. Es bietet sich an, diese Vorgehensweise für die formale Dokumentation von Mustern prinzipiell zu übernehmen. Denn Quelltext wird innerhalb eines Frames als Entität erster Klasse behandelt, was nach Ansicht des Autors Voraussetzung für eine praktikable Erstellung einer formalen Musterdokumentation durch den Entwickler ist. Dabei wird statischer Quelltext direkt in ein Frame geschrieben und dynamischer Quelltext mit Hilfe der Scriptsprache von *ANGIE* deklariert. Die Verknüpfung von statischen und dynamischen Musterbestandteilen ist wichtig, denn diese fließen durch die Musteranwendung zu einem Ganzen zusammen. Also müssen die Abhängigkeiten zwischen den statischen und dynamischen Teilen in der Musterdokumentation erfasst sein, um sie bei der Musteranwendung berücksichtigen zu können.
ANGIE stellt keine Möglichkeit zur Verfügung, Entitäten eine Bedeutung zuzuordnen. Dies ist für Muster allerdings wichtig, da ein Muster nur innerhalb eines Kontextes anwendbar ist. Diese Lücke muss in dieser Arbeit durch geeignete Mechanismen geschlossen werden, die sich in das Konzept der Frames und Slots integrieren lässt.

2.3.1.8 Pattern By Example

Der nachfolgend vorgestellte Ansatz, *Pattern By Example*, verbirgt die Implementierungsdetails der *ANGIE*-Scriptsprache mit Hilfe einer graphischen Oberfläche vor dem Anwender.

Die Methodik *Pattern By Example* (kurz: *PBE*) [PBE 2004a] konzentriert sich auf die formale Dokumentation von Entwurfsmustern. *PBE* basiert technisch auf der Scriptsprache von *ANGIE*.
Das fehleranfällige und unelegante *Copy & Paste* von vorhandenem, an anderer Stelle neu zu verwendendem Programmtext wird mit Hilfe von *PBE* ersetzt durch einen formaleren und zugleich intuitiveren Prozess. Die Definition geschieht ähnlich dem Prinzip *Query By Example* durch Arbeit auf einem Beispiel-Codefragment, das *Initial Code Fragment* genannt wird. Dabei werden die aus *ANGIE* bekannten Slots festgelegt. Slots sind variable Stellen im Programmtext, etwa der Klassenname. Anhängigkeiten zwischen Code-Bestandteilen können definiert werden. Aus diesen Angaben erstellt *PBE* eine Schablone und speichert diese ab. Später kann aus dieser Schablone und mit Hilfe von Parametrisierungen kontextspezifischer Quelltext erzeugt werden.

PBE prägt den Begriff *Code Pattern*. Er wird vom *Design Pattern* unterschieden. Ein *Code Pattern* ist laut [PBE 2004a, S. 5] ein wiederverwendbares, spezifisches Code-Fragment, das parametrisiert und konfiguriert werden kann. Es kann mit einer oder mehreren Zielsprachen verknüpft werden. Synonym für den Begriff *Code Pattern* wird im Rahmen von *PBE* die Bezeichnung *Code Template* verwendet.

Die formale Dokumentation und die Implementierung eines Musters geschehen in zwei unterschiedlichen Phasen. Während die erste Phase eingängig mit *Pattern Definition* bezeichnet wird, heißt die zweite Phase *Pattern Expansion*. In letzterer wird ein Code-Fragment vom Anwender mit Parametern versorgt und Abhängigkeiten zu anderen Modulen werden festgelegt. Bei der Dokumentation eines Musters mit Hilfe von *PBE* werden die im Code-Fragment festgelegten Slots mit Ausdrücken (engl.: *Expressions*) verknüpft, die die dynamische Festlegung des Slotinhalts bei der Erstellung des Generats steuern. Über das Code-Fragment hinaus zu generierender Code kann durch *Insertion Points* angegeben werden. Abhängigkeiten innerhalb von Code-Fragmenten werden durch *Code Blocks* identifiziert. Ein *Code Block* besitzt einen eindeutigen Namen und ist mit zusätzlichen Bedingungen versehen, die in [PBE 2004a] *Condition Expressions* genannt werden. Sie steuern die Verarbeitung der *Code Blocks*. Blöcke können als optional oder als wiederholbar gekennzeichnet werden. Ein optionaler *Code Block* wird nur berücksichtigt, wenn alle in ihm enthaltenen Slots mit Werten gefüllt sind, also die mit ihnen verknüpften Ausdrücke Rückgabewerte liefern. Die Abbildung 17 zeigt die Definition des Musters *Singleton*.

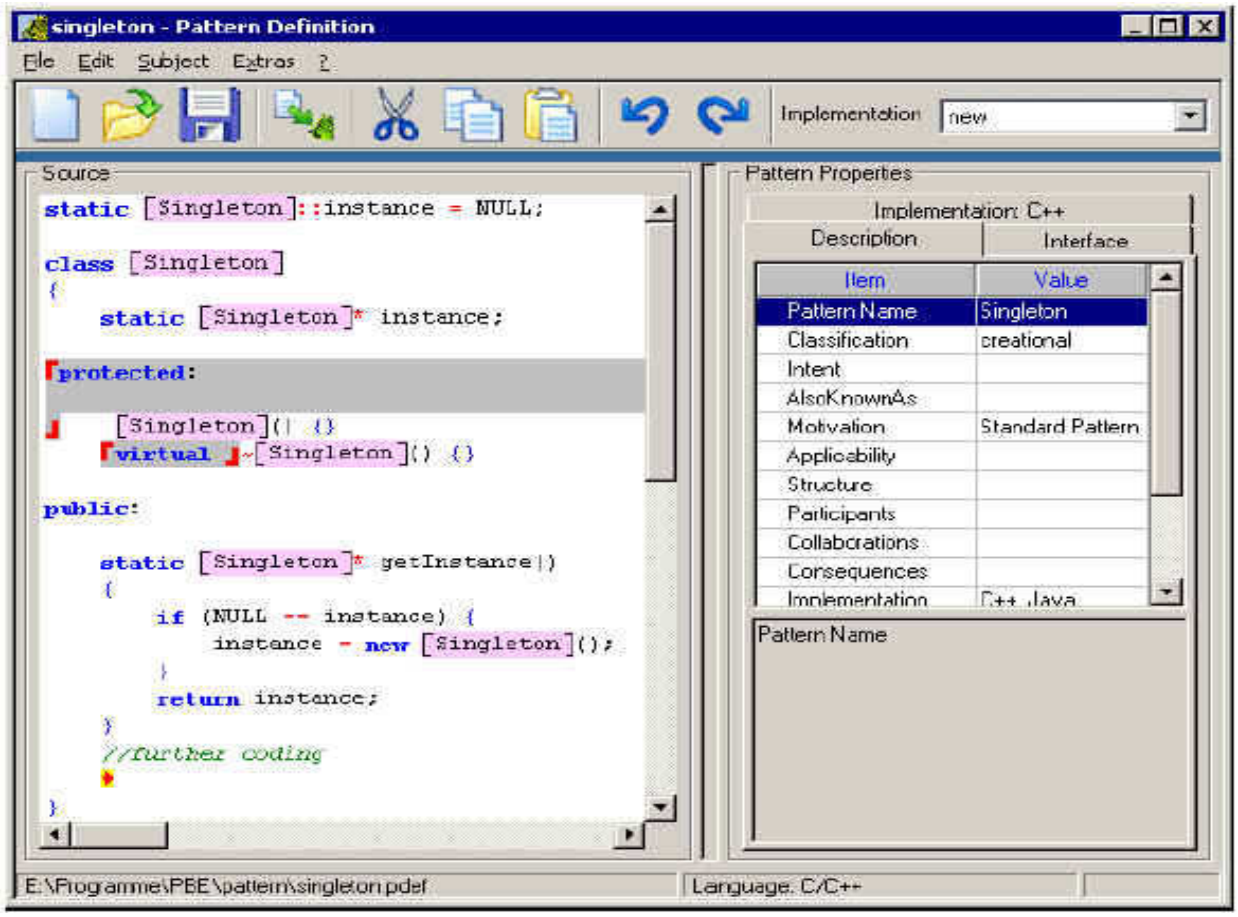

Abbildung 17: Definition eines Musters mit der Methodik *PBE*[17].

Die unterlegten Stellen in der obigen Abbildung repräsentieren Slots. Rechts im Bild ist der Eigenschaftseditor zu sehen. Es passt sich dem Kontext an. Im Bild ist der Kontext durch das Muster insgesamt gegeben. Für Slots bietet der Editor die Möglichkeit der Parametrierung. Eine Präsentation der Musterdefinition ist in [PBE 2004a] zu finden.

Fazit

Die *PBE* zugrunde liegende Idee der Demonstration eignet sich als Konzept für die formale Dokumentation von Mustern. Laut [Mayr-Kern+ 2002, S. 99] ist die Benutzeroberfläche von *PBE* leicht bedienbar und intuitiv. Der Unterschied zwischen der *PBE*-Methodik und der hier intendierten liegt allerdings darin, dass PBE voraussetzt, dass die formale Musterdokumentation bereits vorliegt. In dieser Arbeit ist die Gewinnung einer formalen Musterdokumentation Teil der Problemstellung.

Mit der Methodik *PBE* kann eine Musterdokumentation erstellt werden, um sie später zu instanziieren. Diese Instanziierung ist durch die Möglichkeit der Parametrisierung flexibel möglich. Jedoch kann ein Muster derart nicht in einen vorhandenen Quelltext eingewoben werden. Denn es existiert kein Bezug zwischen den dynamischen Teilen des Musters und den jeweiligen Anknüpfpunkten im Zielquelltext (Zielkontext) für die Musteranwendung. Gerade dieser Bezug wird in der vorliegenden Arbeit als wichtig erachtet und muss geeignet abgebildet werden.

[17] Abbildung entnommen aus http://www.cs.pitt.edu/~chang/231/8pattern/PBE/08.html.

2.3.2 Selektion von Entwurfsmustern

Ein Ziel dieser Arbeit ist die werkzeuggestützte Selektion von Entwurfsmustern, die den Entwickler so stark wie möglich von der Notwendigkeit befreien soll, umfangreiches Wissen über Entwurfsmuster aufbauen zu müssen. Zum Vergleich werden verschiedene Ansätze vorgestellt, die bei der Selektion von Entwurfsmustern für einen gegebenen Kontext unterstützen.

Zwei Ansätze, ein kontextuelles Hilfesystem sowie eine Workbench, dienen als Ergänzung zu werkzeuggestützten Ansätzen. Die Ansätze werden repräsentativ für andere vorgestellt, um die Charakteristik der Konzepte Workbench und kontextuelles Hilfesystem zu betrachten. Dies ist deshalb wichtig für den vorgestellten Ansatz, weil er in ein Werkzeug integriert werden und gut durch eine integrierte, kontextabhängige Hilfe unterstützt werden kann.

Zwei betrachtete Ansätze erlauben die nichttriviale Selektion von Entwurfsmustern anhand eines gegebenen Quelltextes. Diese zwei Ansätze leiten anhand vorhandener Entwurfsdefekte und Unterschiede zwischen zwei korrelierten Programmversionen Vorschläge für anwendbare Entwurfsmuster ab. Die Voraussetzungen für diese dem Verfahren nach prinzipiell interessanten Ansätze sind nach Meinung des Autors schwer zu erfüllen, weil erstens der benötigte Grad der Korrelation nicht klar definiert ist und zweitens zwei korrelierte Programmversionen nicht ohne Weiteres verfügbar sind. Es existieren andere Verfahren mit ähnlichen Charakteristiken. Diese wurden untersucht, als nicht geeignet empfunden und deshalb wird auf diese hier nicht weiter eingegangen.

Ein weiterer Ansatz ergänzt einen technischen Selektionsprozess um die triviale Selektion einzelner Muster mit Hilfe statischer Regeln. Eine solche triviale Selektion ist prinzipiell geeignet zur Ergänzung eines komplexeren Selektionsmechanismus.

Anhand der vorgestellten Ansätze wird deutlich, dass kein zufrieden stellender Mechanismus zur werkzeuggestützten Selektion von Entwurfsmustern existiert. Es handelt sich entweder nur um ergänzende Hilfsmittel oder um nicht allgemein verwendbare Lösungen.

2.3.2.1 Kontextuelles Hilfesystem

Mit [Motelet 2004] wird ein Tutorsystem vorgestellt. Es soll den Entwickler bei der Softwareerstellung unterstützen. Insbesondere kann es bei der Selektion von Mustern hilfreich sein. Das Tutorsystem wurde mit dem Ziel entworfen, die Aspekte

- Entwurfsmuster als Abstraktionsmechanismen,
- intelligente Lehrmethoden und
- kontextuelle Hilfe

abzudecken. Bei dem Tutorsystem handelt es sich also weniger um ein konkretes technisches Hilfsmittel, aus dem produktiver Quelltext hervorgeht. Vielmehr integriert sich das Hilfesystem in den Entwicklungsprozess und bietet eine Nachschlagemöglichkeit für den Entwickler an. Entsprechend des jeweiligen Kontextes werden relevante Hilfethemen angeboten. Hinweise, Informationen und Fragen leiten den Entwickler zur Entscheidung, ein Muster auszuwählen. In [Motelet 2004] wird die Aussage getroffen, Entwurfsmuster können nicht auf systematische Art angewandt werden.

Fazit

Ein kontextuelles Hilfesystem ist keine Unterstützung im technischen Sinne, sondern unterstützt den Entwickler auf organisatorischer Ebene. Daher kann ein Hilfesystem nur eine Ergänzung zu einem werkzeuggestützten Ansatz sein. Weitere Betrachtungen hierzu können sich darauf beschränken, wie ein kontextuelles Hilfesystem in eine Entwicklungsumgebung eingepasst werden kann. Interessant ist ein kontextuelles Hilfesystem in Hinblick auf die Unterstützung des Entwicklers beim Prozessieren

seines Quelltextes mit Blickrichtung Musterdokumentation, -auswahl und -anwendung. Insbesondere kann ein kontextuelles Hilfesystem dem Entwickler helfen, seinen Quelltext in eine Form zu bringen, der für die Anwendbarkeit des hier vorgestellten Ansatzes hilfreich oder notwendig ist.

2.3.2.2 Die Workbench Dali

Gegenstand von [Kazman+ 2002] ist die Rekonstruktion der Architektur einer vorhandenen Legacy-Anwendung. Mittels verschiedener, auch nichtproprietärer Werkzeuge werden entsprechende Informationen aus der vorliegenden Anwendung gewonnen, die bei der sukzessiven Abstrahierung helfen sollen. Oft gelingt die vollständige Rekonstruktion der Anwendungsarchitektur nicht ganz. Erneute Dokumentation anhand gewonnener Informationen erlaubt den Vergleich mit vorhandenen Dokumenten. Aufgrund des Vergleichs können Abweichungen zwischen der tatsächlich vorliegenden und der geplanten Architektur ermittelt werden.
Grundlage für die Rekonstruktion von Informationen ist die Workbench *Dali*, die vom Software Engineering Institute (*SEI*) an der Carnegie Mellon University entwickelt wurde. Es existiert keine einheitliche Definition, was eine Workbench ist. In diesem Kontext lässt sich zumindest soviel sagen: Eine Workbench stellt eine integrierende Oberfläche dar. Sie beinhaltet Werkzeuge, die zur Verrichtung von Tätigkeiten innerhalb einer Domäne geeignet sind. Eine Workbench ist vergleichbar einem Expertenarbeitsplatz. Im Falle von *Dali* handelt es sich um eine technische Oberfläche für einen Entwickler oder Software-Architekten. Durch den Einsatz von sogenannten Plugins in heutigen integrierten Entwicklungsumgebungen (kurz: *IDE*) verschwimmen die Begriffe Workbench und IDE. *Dali* hebt allerdings auf stark spezialisierte Werkzeuge ab, die sich in IDE's eher selten in Form von Plugins, welche hauptsächlich für den Massenmarkt produziert werden, finden lassen. Der im Weiteren verwendete Begriff der Workbench soll diesen Unterschied ausdrücken.

Dali deckt fünf iterative Phasen im Prozess der Rekonstruktion ab:

1. Extraktion von Informationen
2. Erstellung einer Datenbank, die die extrahierten Informationen in standardisierter Form enthält
3. Ermittlung dynamischer Informationen
4. Rekonstruktion der Architektur und Ableiten von Abstraktionen aus den Datenbank-Informationen
5. Analyse der rekonstruierten Architektur

Dali bietet dem Anwender in Form einer Workbench Unterstützung, eine bestehende Architektur besser verstehen zu können. Hauptaugenmerk ist der Abgleich zwischen Soll- und Ist-Architektur. Es handelt sich bei *Dali* eher um einen Ansatz, der Prozesse unterstützt denn um ein eigenständig verwendbares Werkzeug. Allerdings fließen vielfältige Information in die Analysephase ein. So werden in Phase eins nicht nur Quelltexte, *Header Files* und *Build Files* berücksichtigt, sondern auch vorhandene Statistiken wie *Trace*-Dateien. Die Analyse wird unterstützt durch Parser, Werkzeuge für die Analyse von abstrakten Syntaxbäumen, lexikographische Analysewerkzeuge, Profiler, Eingriffe in den Quellcode (*Code Instrumentation*) sowie durch spezifisch erstellte Werkzeuge. Die Workbench *Dali* hilft dem Entwickler bei der Analyse bestehender Strukturen. Sie stellt eine Art Leitfaden mit integrierter Werkzeugunterstützung dar.

Fazit

Eine Workbench bietet keine technische Unterstützung im engeren Sinne. Ähnlich wie ein kontextuelles Hilfesystem kann eine Workbench ein sinnvolles Mittel bei der Anleitung des Entwicklers sein, wie er Programmtext sinnvoll dokumentieren, umgestalten oder erweitern kann.
Die Verwendung einer Workbench ist deshalb sinnvoll, weil durch eine Werkzeugintegration in eine Programmierumgebung die Bedienung vereinfacht werden kann (vgl. [Mayr-Kern+ 2002, S. 27]). Eine Workbench ist hierbei der Rahmen zur Integration von spezifischen Werkzeugen.

Der Vergleich zwischen einer Ist- und Sollarchitektur, wie er in Dali vorgeschlagen wird, kann im Rahmen dieser Arbeit nicht angewandt werden, da ein Ziel dieser Arbeit die Unterstützung des Entwicklers in jeder Phase der Programmerstellung ist, also auch dann, wenn noch keine zwei Programmstände vorliegen, die verglichen werden könnten.

2.3.2.3 Triviale Quelltextanalyse mit PMD

Das Analysewerkzeug *PMD* [PMD 2003] dient der Untersuchung von Java-Quelltext in Hinblick auf die Erkennung von Refactoring-Bedarf. Im Rahmen dessen wird auch der Nutzen, eine Klasse in ein *Singleton* zu konvertieren, untersucht. Der Test darauf fällt positiv aus, wenn alle oder nahezu alle Methoden der untersuchten Klasse als statisch gekennzeichnet sind. Es handelt sich also um einen trivialen Vorschlag für die Anwendbarkeit eines Musters.

Fazit

PMD ist kein Werkzeug, das Entwurfsmustern direkt zuzuordnen wäre. Lediglich für ein Muster, nämlich für *Singleton*, wird dessen Anwendbarkeit in einem konkreten Fall untersucht. Derartige Mechanismen sind nur als Ergänzung anspruchsvollerer Ansätze denkbar, helfen sie doch in feststehender Weise bei der Erkennung trivialer Fälle.

2.3.2.4 Vergleich von korrelierten Programmversionen

[Sang-Uk 2002] verfolgt das Ziel, potentiell anwendbare Entwurfsmuster automatisch zu erkennen und halbautomatisch in bestehenden Programmtext einzuflechten. Der Ansatz vergleicht zwei Programmversionen, um sogenannte *Candidate Spots* für die Anwendung von Entwurfsmustern identifizieren zu können. Die erste Programmversion liegt dabei zeitlich vor der zweiten Version. Die Änderungen zwischen den beiden Versionen werden analysiert und aus dem Analyseergebnis werden Informationen über das Design gewonnen. Die Analyse wird mit Hilfe einer Prolog-Faktenbasis vorgenommen. Motivation von [Sang-Uk 2002] ist die oft hohe Komplexität vorhandener Programme, die eine Identifizierung von der Anwendung eines Musters betroffener Stellen erschwert oder praktisch unmöglich macht.
In [Sang-Uk 2003a] und [Sang-Uk+ 2003b] stellt derselbe Autor, zusammen mit anderen, die Anwendbarkeit des Ansatzes anhand des Musters *Abstract Factory* dar. Letztgenannte Quelle listet zudem die in Prolog vorgesehenen Prädikate auf, die zur Beschreibung des Programmmodels verwendet werden. Diese Regeln sind auf Beziehungen zwischen Klassen beschränkt (»erbt von«, »vererbt an« usw.). Es wird nicht auf spezielle Rollen eines Objektes im Code eingegangen (wie etwa der eines Beobachters im Rahmen des *Observer* Musters).
Die Arbeit wählt Java als Zielsprache. Sie beschränkt sich stark auf Legacy Code, denn nur hier existieren wirklich zwei entsprechend korrespondierende und korrelierte Versionen eines Programms.

Fazit

Nach Aussage von [Sang-Uk 2002] beschränkt sich die Anwendbarkeit des Ansatzes auf einige Erzeugungsmuster. Es ist unklar, inwieweit sich die genannten zwei Programmversionen entsprechen müssen, welche Unterschiede möglich und welche Gemeinsamkeit notwendig sind. Aufgrund dieser Einschränkungen erscheint der Ansatz nicht interessant um anwendbare Entwurfsmuster zu selektieren. Das Konzept der *Candidate Spots* hingegen trägt der Möglichkeit Rechnung, dass ein Muster verteilt über einen Programmtext vorliegen kann und ist an sich als Konzept betrachtenswert.

2.3.2.5 PTIDEJ: Ermittlung und Korrektur von Design Defekten

Die Arbeit [Guéhéneuc+ 2001] beschäftigt sich mit der Erkennung und Korrektur von Entwurfsdefekten. Die Arbeit spezialisiert sich auf die Behebung von Defekten, die beim Zusammenspiel zwischen Klassen existieren. Zur Korrektur der Defekte werden Entwurfsmuster eingesetzt. Zum Zweck der Erkennung von Defekten werden diese klassifiziert. [Guéhéneuc+ 2001] unterscheidet Defekte innerhalb einer Klasse, zwischen Klassen und solche, die dem Verhalten von

Klassen zuzurechnen sind. Manche Defekte können nach Aussage von [Guéhéneuc+ 2001] auch mehreren Kategorien zugeordnet werden, da die Kategorien nicht vollständig disjunkt sind. So führt [Guéhéneuc+ 2001] ergänzend Mischkategorien an aus Defekten, die jeweils zwei der drei Kategorien zuzuordnen sind sowie einer alle drei Kategorien umfassenden Mischkategorie. Die Defekterkennung findet derart statt, dass nach Entwurfsmustern gesucht wird und teils vorhandene, teils fehlende Teile als Defekt interpretiert werden. Anhand des Grades der Übereinstimmung kann die Güte des Programmcodes abgeleitet werden. Finden sich nur wenige Charakteristika eines Entwurfsmusters im Programm wieder und andere davon fehlen, so deutet dies für ein bestimmtes Muster auf ein letztendlich nicht angewandtes hin, dessen (korrekte) Anwendung aber angebracht wäre. Das Werkzeug *PTIDEJ* ermittelt die Unterschiede zwischen aktueller und korrekter/gewünschter Version. Daraus leitet *PTIDEJ* Vorschläge ab, die dem Anwender präsentiert werden und als Arbeitsgrundlage für Refactoring-Maßnahmen dienen können. Dieses Konzept soll hier kurz »Indikationsstrukturen« genannt werden.

Zur Erkennung von Entwurfsdefekten wird ein Metamodell für Entwurfsmustern eingeführt. Es enthält neben Informationen zur Erkennung des Musters auch solche zur Instanziierung desselben. Als Instanziierung bezeichnet [Guéhéneuc+ 2001] den Prozess, der sonst auch als Musteranwendung bezeichnet wird, nämlich die Transformation eines bestehenden Quelltextes, so dass das Ergebnis der Spezifikation des Entwurfsmusters entspricht.

Zur Instanziierung des Musters werden im Metamodell Dienstroutinen bereitgestellt. Für das *Composite* Muster etwa wird eine Methode *addLeaf* im Metamodell aufgenommen. Sie fügt durch Transformation eines gegebenen Quelltextes diesem Quelltext eine Blattklasse mit frei wählbarem Namen hinzu. Weiterhin beinhaltet das Modell Angaben zur Struktur des Musters. Im Fall von *Composite* bedeutet das etwa, dass Komponentenklassen die Schnittstelle *Component* umsetzen. [Guéhéneuc+ 2001] bietet keine reine Strukturdefinition an, sondern lässt eine Unsicherheit darin erkennen, ob Strukturdefinition und Erkennungslogik nicht kombiniert werden sollten. Das darauf hinweisende Beispiel ist die Definition *composite.addShouldImplement(component)*, wobei *component* eine Deklaration der Schnittstelle zur Komponente im *Composite*-Muster darstellt. Der Bestandteil »should« drückt keine unbedingte Definition aus, sondern bedeutet eine Erwartungshaltung, interpretiert man die natürlichsprachliche Bedeutung von »should«. In [Guéhéneuc+ 2001] wird hierzu leider nichts weiter ausgeführt.

Die Logik zur Erkennung von Entwurfsdefekten basiert auf der Definition von Regeln zur Erkennung des Musters, das im Metamodell beschrieben wurde (Instanziierungslogik). Die Erkennungs- und die Instanziierungslogik sind in [Guéhéneuc+ 2001] getrennt aufgestellt. Als Ziel nennen die Autoren die automatische Folgerung der Erkennungslogik aus der Instanziierungslogik. Die Routinen zur Erkennung von Musterteilen sind in Form von Nachbedingungen aufgestellt. Jede Nachbedingung ist in Form einer Scriptsprache, die Ähnlichkeit zu Java-Konstrukten hat, ausgedrückt.
Die mit bestimmten Entwurfsmustern ähnlichen Entitäten werden durch einen *Constraint Solver* ermittelt. Ein solcher wurde von [Guéhéneuc+ 2001] in der Programmiersprache *Claire* verwirklicht. Die Anwendung notwendiger Änderungen findet mittels der eigens entwickelten Java-Spracherweiterung *JavaXL*[18] statt. *JavaXL* führt Quelltexttransformationen so durch, dass bestimmte interne Semantikinformationen im Programmtext erhalten bleiben. Das betrifft insbesondere Kommentare und Deklarationen, die im gegebenen Kontext von Bedeutung sind.

Mit Hilfe der UML gibt [Guéhéneuc+ 2001] eine teilweise Repräsentation von Metamodellen für Entwurfsmuster wieder. Es scheint nicht möglich zu sein, das Metamodell komplett mit UML abzubilden. Jedenfalls tut [Guéhéneuc+ 2001] dies nicht, gibt allerdings auch keine weiteren Informationen zu diesem Umstand an.

[18] JavaXL = Java eXtended Language

Die Autoren selbst kritisieren ihren Ansatz wie folgt:

- Das Metamodell unterliegt Einschränkungen. Es berücksichtigt dynamische Aspekte von Anwendungen zu wenig und ihm mangelt es an Mächtigkeit.
- Das Metamodell ist nicht spezialisiert genug, um Constraints und Transformationsregeln auf feingranularer Ebene kontrollieren zu können.
- Die Regeln zur Erkennung von Entwurfsmustern stehen nicht in Beziehung zum Metamodell. Sie werden von Hand erstellt.
- Aus einem vorhandenen Quelltext lassen sich nicht alle benötigten (Modell-)Informationen ermitteln. Als Beispiel wird eine generische Liste angegeben, deren Ausprägung erst zur Laufzeit bekannt wird.
- Transformationsregeln befinden sich auf einer anderen semantischen Ebene als Constraints.
- *JavaXL* garantiert nicht die Kompilierbarkeit eines transformierten Quelltextes.

Fazit

In [Guéhéneuc+ 2001] werden in einem Metamodell die Aspekte formale Musterdokumentation zum Zweck von dessen Instanziierung mit Hilfe von Transformationen sowie Definition von Regeln zur Erkennung von Musterteilen unterschieden und getrennt von einander unterstützt. Die Musteranwendung durch Transformationen geschieht mit Hilfe von eigens erstellten Dienstroutinen. [Guéhéneuc+ 2001] lässt offen, ob diese Dienstroutinen in nennenswerter Weise wiederverwendet werden können. Das scheint aus Sicht des Autors möglich.
Die Grundidee, eine entartete Version eines Musters im Quelltext vor Anwendung des Originalmusters mit Hilfe von Indikationsstrukturen aufzuspüren ist aus zwei Gründen sinnvoll. Erstens kann so der Bedarf erkannt werden, ein Muster anzuwenden. Zweitens kann bei der Anwendung verhindert werden, dass ein Muster nach dessen Anwendung in Teilen doppelt vorkommt oder nicht angewandt werden kann, weil Musterteile bereits im Quelltext vorliegen. Auch die Beschreibung der Semantik eines Musters mit Hilfe eines Metamodells erscheint dem Autor eine vielversprechende Idee.
Die Abbildung von Erkennungsregeln in Form einer proprietären Scriptsprache hält der Autor für ungünstig, da die Verwendung einer populären Hochsprache wie Java oder C# einen geringeren Einarbeitungsaufwand für die meisten Entwickler darstellt und diese generischen Sprachen mächtiger erscheinen. Die Kategorisierung von Defekten in Klassen gemäß [Guéhéneuc+ 2001] ist eine Hilfestellung bei der Gewichtung von gefundenen Entwurfsdefekten. Sie ist laut [Guéhéneuc+ 2001] selbst noch zu grob, um gefundene Defekte qualitativ über deren Kategorien vergleichen zu können oder andere Ansätze hinsichtlich deren Fähigkeiten gegenüberzustellen. Die Kategorisierung von Defekten sollte im Rahmen eines Verfahrens prinzipiell möglich und erlaubt sein.

2.3.3 Anwendung von Entwurfsmustern

Die dritte Zielstellung dieser Arbeit ist die Anwendung eines zuvor selektierten Entwurfsmusters auf einem Zielkontext. Die werkzeuggestützte Anwendung eines Entwurfsmusters basiert auf einer vorherigen formalen Dokumentation desselben, wofür bereits Ansätze vorgestellt wurden. Mit Hilfe der formalen Musterdokumentation soll erstens festgestellt werden, welche Muster sinnvollerweise anwendbar sind und welche aus bestimmten Gründen nicht. Dazu müssen Vorbedingungen geprüft werden, die als Teil der formalen Musterdokumentation anzusehen sind. Zweitens soll die formale Musterdokumentation zumindest alle statischen Musterbestandteile enthalten sowie die dynamischen in Bezug zu diesen setzen. Die nachfolgend beschriebenen Ansätze müssen also auf einer formalen Dokumentation beruhen, um darauf aufbauend Muster werkzeuggestützt anwenden zu können.

Der erste vorgestellte Ansatz der Minipatterns und Minitransformationen zeigt, wie man die Anwendung eines Musters in Teile zerlegen und das Problem reduzieren kann. In diese Kategorie lässt sich auch die vorgestellte Entwurfsmusteranwendung mittels Operatorsicht einordnen. Beide Ansätze basieren auf einer nichtformalen Definition von Nachbedingungen, die nach der

Musteranwendung erfüllt sein müssen. Alleine daran wird die fehlende Automatisierbarkeit der Verfahren deutlich.
Die generative Musteranwendung ist Gegenstand eines weiteren Ansatzes. Er ist ebenfalls, wie einer der beiden obigen Ansätze, nur auf Frameworks anwendbar. Die Fokussierung auf Frameworks stellt eine entscheidende Einschränkung dar, handelt es sich bei Frameworks doch um ein generisches Fundament in Form einer unfertigen Anwendung. Für ein generisches Programmstück ist die Musteranwendung schon deshalb einfacher als für eine nichtabstrakte Anwendung, weil ein Muster nicht ausimplementiert werden muss, dessen abstrakte Teile also unimplementiert bleiben können.

Ein darüber hinaus betrachteter Ansatz, *PSE-Handler*, verdeutlicht die Vorteile einer Gesamtlösung der Musteranwendung, die die formale Dokumentation und Selektion eines Musters beinhaltet und in eine Entwicklungsumgebung integrierbar ist. Im Rahmen dieser Arbeit ist der Fokus auf einen Ansatz zu setzen, der Musterdokumentation und -selektion integriert und darüber hinaus versucht, die werkzeuggestützte Musteranwendung damit zu verknüpfen.

2.3.3.1 Minipatterns und Minitransformationen

In den Arbeiten [Ó Cinnéide+ 1999a] und [Ó Cinnéide+ 1999b] wird eine Methodologie für die automatisierte Anwendung von Entwurfsmustern in Legacy Code beschrieben. Der Entwickler entscheidet, welches Muster anzuwenden ist.
Das Konzept von sogenannten Minipatterns wie *ABSTRACTION*, *ENCAPSULATE CONSTRUCTION* und *ABSTRACT ACCESS* dient der feingranularen Dokumentation von Mustern. Minitransformationen helfen bei der automatisierten Implementierung von Entwurfsmustern, die vorher manuell ausgesucht wurden.

Das in [Ó Cinnéide+ 1999b] beschriebene Ziel ist die Bereitstellung eines Werkzeuges zur automatisierten Anwendung von Entwurfsmustern in ein bestehendes Programm. Dazu wird zuerst ein Muster prinzipiell manuell ausgewählt. Als zweites muss ein Startpunkt in einem bestehenden Programm angegeben werden, der als *Precursor* bezeichnet wird. Diese beiden Invarianten führen zu einer Zerlegung in Minipatterns. Minipatterns sind niederwertige Konstrukte, die feingranularer sind als ihr Superelement, das Entwurfsmuster selbst.

Für jedes Minipattern muss eine korrespondierende Minitransformation definiert werden. Sie besteht aus einer Reihe von Vor- und Nachbedingungen sowie den durchzuführenden Transformationsschritten; die Idee der Nachbedingungen findet sich u.a. auch in [Zimmer 1997] wieder. Zusätzlich dazu ist in einer Art Demonstration die Beibehaltung des bisherigen Verhaltens zu zeigen. Eine Demonstration ist hier die Anwendung auf einen geeignet gewählten Beispielquelltext, auf den das Minipattern anwendbar ist. Diese Demonstration wird ebenfalls als Bestandteil einer Minitransformation gefordert.

Die Einbringung eines Entwurfsmusters kann nun als Folge auszuführender Minitransformationen beschrieben werden. Da jede Minitransformation das ursprüngliche Programmverhalten konserviert, besitzt der Gesamtvorgang dieselbe Charakteristik. Dieser Ansatz ist in mancher Hinsicht ähnlich dem, die Anwendbarkeit eines noch nicht vorhandenen Entwurfsmusters automatisiert zu prüfen. Zwar beinhaltet die in [Ó Cinnéide+ 1999b] vorgestellte Methodik als ersten Schritt die manuelle Auswahl eines solchen. Die Identifizierung von Elementen in einem vorhandenen Programm, die potentielle Bestandteile eines zu applizierenden Musters sind, ist aber Voraussetzung für das automatisierte Finden anwendbarer Entwurfsmuster. In [Ó Cinnéide+ 1999b] vermieden wird die Frage nach dem Startpunkt für ein Entwurfsmuster, indem das Konzept des sogenannten *Precursors* beschrieben wird. Ein *Precursor* zeichnet eine Stelle im Programm aus, wo eine Implementierung ohne Hilfe eines Entwurfsmusters (zunächst) praktikabel erscheint. Der *Precursor* weist aber durch seine Existenz gleichzeitig auf das Potential hin, als Einstiegspunkt für ein bestimmtes Entwurfsmuster geeignet zu sein.

Ein in [Ó Cinnéide+ 1999b] genanntes Beispiel für das *GoF*-Entwurfsmuster *Factory Method* (vgl. [Gamma+ 1995]) soll zur Veranschaulichung im Original wiedergegeben werden (Abbildung 18):

Name: APPLY FACTORY METHOD

Arguments:

`String creator`: name of the class that creates product objects.

`String product`: name of the product class.

`String absP`: name of the new interface to the `product` class.

`String createP`: name of the new factory method itself.

Description: Applies the Factory Method design pattern to the program.

Preconditions:

```
program.exists(creator) (pre1)
program.exists(product) (pre2)
!program.exists(absP) (pre3)
!program.exists(creator,createP) (pre4)
creator.creates(product) (pre5)
!product.hasPublicField() (pre6)
!product.hasStaticMethod() (pre7)
```

Algorithm:

```
1. ABSTRACTION(product, absP)
2. ENCAPSULATE_CONSTRUCTION(creator, product, createP)
3. ABSTRACT_ACCESS(creator, product, absP, createP)
```

Behaviour Preservation:

We need only show that the preconditions for each of the component minitransformations hold:

1: `pre2` and `pre3` guarantee the preconditions for this step.

2: `pre1`, `pre2` and `pre4` guarantee the preconditions for this step.

3: `pre1`, `pre2`, `pre3`, `pre6`, `pre7` coupled with the postcondition of (1) guarantee the preconditions for this step.

Abbildung 18: Beschreibung einer Minitransformation (aus [Ó Cinnéide+ 1999b, S. 3]).

[Ó Cinnéide+ 1999a] definiert folgende Hilfsfunktionen und Prädikate für Vor- und Nachbedingungen (Tabelle 3):

Name	Beschreibung
ClassCreated	Liefert die Klasse eines Objektes zurück, das durch einen Objekterzeugungsausdruck erzeugt wurde.
ContainingMethod	Liefert die Methode zurück, die eine gegebene Programmentität (Ausdruck, Objektreferenz etc.) enthält.
EqualInterface	Liefert wahr zurück genau dann, wenn die gegebene Schnittstelle exakt die öffentlichen Methoden der gegebenen Klasse reflektiert.
Exists	Bestimmt, ob die gegebene Programmentität existiert.
HasPublicField	Wahr genau dann, wenn die gegebene Klasse ein öffentliches Feld enthält.
HasStaticMethod	Wahr genau dann, wenn die gegebene Klasse eine statische Methode enthält.
Type	Liefert den Typ einer gegebenen Objektreferenz zurück.

Tabelle 3: Hilfsfunktionen und Prädikate für Vor- und Nachbedingungen.

[Ó Cinnéide+ 1999b] sieht die Anwendbarkeit der Methodik besonders für Erzeugungsmuster (*creational patterns*) sowie Strukturmuster (*structural patterns*) gegeben. Für Verhaltensmuster (*behavioural patterns*) ist der Ansatz nach deren Aussage noch nicht ausgereift. Erneut zu betonen ist die Ausrichtung auf Legacy Code, dem durch die Einführung des *Precursor*-Konstruktes besonders Rechnung getragen wird. Es ist davon auszugehen, dass in der vorliegenden Arbeit die drei genannten Musterklassifikationen, Erzeugung, Struktur und Verhalten, ebenfalls unterschiedlich zu betrachten sind.

Die Minipatterns in [Ó Cinnéide+ 1999b] erinnern sowohl an *PatternsBox/PTIDEJ* als auch an die später vorgestellte Vorgehensweise in [Philippow+ 2004]. Dort wendet der Autor das Prinzip der Schlüsselmerkmale an. Er definiert obligatorische Kriterien, die zusammen mit Ausschlusscharakteristiken zur werkzeuggestützten Identifizierung von Entwurfsmustern beitragen.

Fazit

Der Ansatz der Minipatterns besitzt die prinzipiell interessanten Konzepte des Aufteilens eines Problems in Teilprobleme sowie die Durchführung von Einzelschritten unter der Forderung der Verhaltenskonservation. Das Konzept der Vorbedingungen eignet sich zur Beschreibung von Programmstrukturen, die zur Anwendung eines Musters notwendig sind. Die auszugsweise in [Ó Cinnéide+ 1999a] angegebenen Hilfsfunktionen und Prädikate lassen sich für die Zwecke dieser Arbeit modifizieren und ergänzen. Beispielsweise könnte aus dem Prädikat *hasPublicField* eine universelle Funktion abgeleitet werden, die als Eingabeparameter einen bestimmten Feldnamen oder auch einen Objekttypen übergeben bekommt. Die Vorbedingungen aus [Ó Cinnéide+ 1999a] berücksichtigen nicht die Bedeutung des betrachteten Quelltextes. Sie analysieren nur dessen statische Form.
Das Konstrukt der Minipatterns, das konzeptionell ein Charakteristikum eines Musters beschreibt, ist hilfreich für den Aufbau einer Ontologie zur formalen Dokumentation von Entwurfsmustern. Eine solche Ontologie beinhaltet im Idealfall alle möglichen Aspekte eines Musters (etwa Verhaltens-, Struktur- und Erzeugungsaspekte).
Der *Precursor* definiert genau einen Einstiegspunkt für ein Entwurfsmuster. Ein Einstiegspunkt ist nach Ansicht des Autors nicht immer ausreichend, da Entwurfsmuster gelegentlich die Eigenschaft haben, über mehrere Stellen im Programm verteilt zu sein (siehe auch [ANGIE 2003], wo Code-Fragmente respektive Aspekte unterstützt werden; Vgl. auch *Candidate Spots* [Sang-Uk 2002], die mehrere Einstiegspunkte für ein Muster berücksichtigen). Womöglich hat der Autor die Arbeit von Ó Cinnéide auch falsch verstanden und es handelt sich beim *Precursor*-Konstrukt um eine Voraussetzung für ein Muster, die auch aus mehreren Teilen bestehen kann (etwa im übertragenen Sinne von [Kurzweil 1999, S. 19]).
Der Transformationsprozess mit Hilfe von Minitransformationen bedarf insgesamt einer erheblichen manuellen und intellektuellen Arbeit. Er ist konzeptionell für diese Arbeit interessant, muss aber in Hinblick auf den zu leistenden Aufwand günstiger gestalten werden, um für den durchschnittlichen Entwickler praktikabel zu sein.

2.3.3.2 Entwurfsmusteranwendung mittels Operatorsicht

Die Arbeit von [Zimmer 1997] beschreibt ein Konzept zur systematischen Anwendung von Entwurfsmustern und zur Erkennung von Gemeinsamkeiten zwischen Mustern. Entwurfsmuster werden als Operatoren zur Überführung eines Ausgangs- in einen Zielzustand betrachtet. Dazu wird ein Muster in generische und in anwendungsspezifische Teile zerlegt. Generische Teile können automatisiert implementiert werden, anwendungsspezifische erst nach Eingriff des Benutzers, etwa durch Variantenauswahl.
Nach manueller Auswahl eines Teils eines Entwurfes und eines darauf anzuwendenden Musters wird der bisherige Entwurf in einen Zielentwurf transformiert ([Zimmer 1997, S. 43f]). Dieser Zielentwurf muss reorganisiert werden, falls Schnittstellen verändert wurden. Die Operatorsicht steht im Gegensatz zur Bausteinsicht. Bei der Bausteinsicht wird ein gegebener Kontext (Teil eines Entwurfes) als Baustein betrachtet. Die für die Anwendung des gewählten Musters relevante Untermenge wird letztendlich durch einen anderen Baustein ersetzt. Die Transformationsschritte sind nicht oder nur unzureichend bekannt. Die Operatorsicht will dieses Manko beheben. Sie beschreibt Transformationsschritte zur Überführung eines Ausgangsentwurfes in einen Zielentwurf. Im Erfolgsfall enthält der Zielentwurf nach der Transformation und einer eventuellen Reorganisation das anzuwendende Muster und erfüllt die manuell festgelegten Nachbedingungen.

Die folgende Abbildung 19 zeigt das Grundprinzip der Operatorsicht:

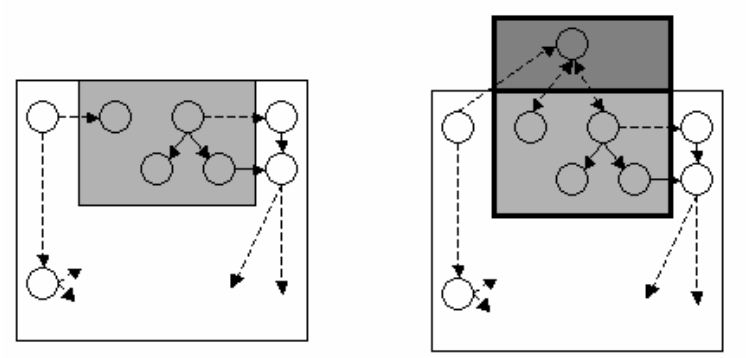

Abbildung 19: Anwendung eines Entwurfsmusters mittels Operatorsicht (nach [Zimmer 1997]).

Die linke Seite zeigt den ursprünglichen Programmzustand mit einem ausgewählten Kontext (hellgrau dargestellt). Der ausgewählte Kontext ist der für die Entwurfsmusteranwendung relevante. Die rechte Seite zeigt den Programmzustand nach Anwendung eines zuvor ausgewählten Entwurfsmusters (als dunkelgrauer Block dargestellt), also den Zustand nach Transformation. Der dunkelgraue Bereich ist durch die Anwendung des Entwurfsmusters hinzugekommen. Er wurde mit dem Zielkontext verwoben, was durch Pfeilverbindungen zwischen den beiden grau schattierten Bereichen auf der rechten Seite angedeutet ist. Ein möglicher Operator für den in der Abbildung dargestellten Übergang ist die Einführung einer Klasse mitsamt einer Methode und einer nichtprivaten Variablen.
Ebenfalls betroffen von der Transformation sind mit dem Zielkontext ursprünglich verbundene Elemente, wie in der Abbildung auch zu erkennen. Im Transformationsprozess ist die Nichtverletzung von Postkonditionen ein wichtiges Ziel und eine Herausforderung. Wichtig sind Nachbedingungen, weil sie helfen, fehlerhafte Transformationen zu erkennen, welche prinzipiell nicht unvermeidbar sind wenn unbekannter Quelltext umgeformt wird. Eine Herausforderung ist die Definition von Nachbedingungen weil sie beliebig komplex werden können. Da die formale Definition von Postkonditionen oft anspruchsvoll ist, geschieht deren Definition bisher meist informell oder durch manuelle Implementierung von Testfällen ([Zimmer 1997, S. 45 u. 104]).
Wie in der Abbildung auf der rechten Seite zu erahnen ist, sind einige nicht offensichtliche Operationen bzw. Transformationen durchzuführen, um das gewählte Muster anzuwenden, sprich: einzuweben.

Neben der kritischen Betrachtung des von der *GoF* [Gamma+ 1995] ausgehenden Ansatzes zur Musterbeschreibung und der Aufzählung damit verbundener Probleme definiert [Zimmer 1997] Typen von Beziehungen zwischen Entwurfsmustern. Folgende Beziehungen sind nach [Zimmer 1997, S. 68] möglich (teilweise wörtlich übernommen, Tabelle 4):

Beziehungstyp	Definition
Muster A ist Variante von Muster B	A ist Teilmenge der Pfade des Ablaufgraphen von B. Jeder Pfad der Teilmenge beginnt in der Wurzel (von A) und endet an einem Blatt.
A benutzt B	Alle Varianten von A benutzen B.
Variante von A benutzt B	Mindestens eine Variante von A benutzt B.
A und B können kombiniert werden	Es gibt ein Entwurfsmuster C, das sowohl A als auch B benutzt.
A spezialisiert B	A ist identisch mit B bis auf in A näher spezifizierte Teile.

Tabelle 4: Beziehungen zwischen Entwurfsmustern (nach [Zimmer 1997, S. 68]).

Der Ansatz setzt einerseits die manuelle Auswahl eines Musters vor dessen Anwendung voraus. Andererseits müssen bezüglich der Anwendung des Musters Nachbedingungen ebenfalls manuell definiert werden, was ein komplexes Problem darstellen kann. Die Beschränkung von [Zimmer 1997] liegt in der Fokussierung auf die Framework-Konstruktion und in der Ausrichtung auf bestehende Entwürfe ([Zimmer 1997, S. 12]). Das hat die möglichst frühzeitige Instanziierung des zu

entwickelnden Frameworks zur Konsequenz ([Zimmer 1997, S. 98]). Eine Forderung an diese Instanziierung, die im frühen Stadium eine Art Demonstrationsanwendung oder *Proof of Concept* darstellt, ist die charakteristische Übereinstimmung mit der letztendlich intendierten Verwendung des Frameworks.

Fazit

Die Beziehungstypen zwischen Entwurfsmustern sind prinzipiell interessant. Muster liegen oft nicht in Reinform vor, sondern als Variationen. Insbesondere bei der Mustererkennung steigern Variantendefinitionen die Möglichkeit, leicht verfremdete Muster noch identifizieren zu können. Bei der Musterauswahl allerdings steht der Nutzen von Beziehungstypen nach Zimmer in Frage. Denn ein Muster, welches ein anderes erweitert, kann einen vollkommen unterschiedlichen Anwendungsbereich haben als das Basismuster.
Nachbedingungen an sich sind ein wichtiges Mittel zur Sicherstellung der Richtigkeit einer Quelltexttransformation. Hierauf kann nicht verzichtet werden, da eine beliebig komplexe Transformation eines beliebigen Quelltextes nicht an sich als richtig funktionierend gelten kann.
Die Definition von Nachbedingungen, die nach der Musteranwendung gelten müssen, in Form von Testfällen abzubilden, erscheint dem Autor als geeigneter Baustein im Rahmen einer Gesamtmaßnahme, die zusätzlich programmatische Definitionen von Nachbedingungen beinhaltet (siehe auch [Meffert 2006b]). Letztere prüfen in Form eines möglichst beliebig automatisch und schnell ausführbaren Programmbausteins die Gültigkeit einer Musteranwendung. Die in [Zimmer 1997] benannten Nachbedingungen beziehen sich lediglich auf die Struktur und Ausprägung einer Implementierung. Sie berücksichtigen nicht die Bedeutung der nach Transformation vorliegenden Implementierung. Die semantische Komponente wird in [Zimmer 1997] nur im Ansatz durch eine beliebig gestaltete, informal formulierte Nachbedingung bedient.
Die frühzeitige Verfügbarkeit einer entsprechend strukturierten *Proof of Concept* Anwendung, die [Zimmer 1997] fordert, ist in der kommerziellen Software-Entwicklung nach Ansicht des Autors in der Mehrzahl der Fälle nicht gegeben.

2.3.3.3 Generative Design Patterns mit CO_2P_2S und Meta- CO_2P_2S

[MacDonald+ 2002] verfolgt das Ziel, Entwurfsmuster bei der Entwicklung von Frameworks für Anwendungen, die parallel auf verteilten Speicher zugreifen, generativ zu erzeugen. [MacDonald+ 2002] haben ein Werkzeug mit dem Namen CO_2P_2S[19] erstellt (siehe [Tan+ 2003]).
Nach der manuellen Auswahl einer Anzahl von Mustern wird für diese ein Framework erzeugt. Die unterstützten domänenspezifischen Entwurfsmuster liegen in Form von Quelltextteilen vor. Dieser Quelltext besitzt sowohl Teile, die bei der Musteranwendung unverändert übernommen werden, als auch solche, die durch *Template*- und *Hook*-Methoden variiert werden können. Mit Hilfe von fest vorgegebenen Parametern werden unterschiedliche Instanziierungen nach dem Baukastenprinzip ausgewählt. Die Instanziierungen selbst sind durch die Parameter weggekapselt, der Anwender erhält also durch Einstellen der Parameter ein zur Konfiguration passendes Ergebnis. Das Ergebnis ergibt sich durch die Auswahl entsprechender *Template*- und *Hook*-Methoden, die zu der eingestellten Konfiguration passen.
In [MacDonald+ 2002] und in anderen Arbeiten (etwa [Zimmer 1997, S. 35f]) wird die informelle, auf einer natürlichen Sprache basierende Beschreibung von Entwurfsmustern kritisiert, wie sie durch die *GoF* [Gamma+ 1995] populär geworden ist. Mit einer derartigen, nicht maschinenlesbaren »Beschreibung« liegt keine formale Dokumentation vor. Daraus resultierend, beschreiben verschiedene Autoren Muster mit demselben Namen in bedeutungsverschiedener Weise. Weitere Probleme, die mit der generischen Anwendung eines Entwurfsmusters zusammenhängen, sind:

- Entwurfsmuster repräsentieren einen relativ allgemeinen Lösungsvorschlag für unterschiedlichste Kontexte. Daher fiele eine universelle generative Lösung sehr generisch

[19] CO_2P_2S = Correct Object-Oriented Pattern-based Programming System

und komplex aus. Dieses Problem tritt bei einer generativen Lösung auf, bevor Adaptionen an die konkrete Problemstellung vorgenommen wurden.
- Der umgekehrte Fall, wenn Anpassungen an eine spezielle Problemstellung vorhanden sind, produziert zu spezialisierte Lösungen von Frameworks.
- Der Kompromiss, während der Adaptionsphase die Code-Generierung vorzunehmen, resultiert in der zufrieden stellenden Abschwächung der ersten beiden Nachteile. Hier wird die Adaptionsphase in zwei Unterphasen aufgeteilt und zwischen diesen Phasen die Generierung vorgenommen. Nachteil ist die höhere Komplexität der Adaption.

Eine Anmerkung zur Adaption: Jedes Entwurfsmuster muss in Hinblick auf Adaptionsparameter untersucht werden. Dann müssen für diese *Constraints*, also Gültigkeitsbedingungen, definiert werden. Die Anzahl der Kombinationen wächst exponentiell mit der Zahl der Parameter, weshalb die Autoren von [MacDonald+ 2002] dafür Software-Unterstützung fordern und in Form von CO_2P_2S selbst anbieten. Mit diesem Werkzeug können Vorlagen für Muster definiert werden. Die Parameter der Muster werden unterteilt in:

- *Lexical*: Spezifizierung einer syntaktischen Struktur (Klassenname etc.)
- *Design*: Beeinflussung der Struktur des Generats
- *Performance*: Leistungsfähigkeit
- *Verification*: Überprüfung von Semantik mittels benutzerspezifiziertem Code

Das Entwurfsmuster *Composite* besitzt beispielsweise folgende sieben Parameter (anfangs jeder Zeile die Kategorie des Parameters):

1. *Lexical*: Name der abstrakten Komponentenklasse
2. *Lexical*: Name der abstrakten *Composite*-Klasse
3. *Lexical*: Name der abstrakten Blatt-Klasse (engl.: *Leaf*)
4. *Lexical*: Name der Muster-Superklasse
5. *Design*: Ort der Kinderoperationen des *Composite*, vergleichbar dem *Precursor*
6. *Design*: Liste von Operationen
7. *Performance*: Die Arten der verwendeten Container (Ablageort der Kinderkomponenten des *Composite*)

Der Code Generator von CO_2P_2S unterstützt Slots und sogenannte geschützte Regionen (*Protected Regions*). Um diese zu kennzeichnen, werden im Quelltext Markierungen in Form von Javadoc-Kommentaren festgelegt (siehe auch [NetWeaver 2003] für ein analoges Vorgehen). Solche Kommentare haben den Vorteil, verhaltensneutral zu sein und direkt von der jeweils verwendeten Entwicklungsumgebung und vom Compiler unterstützt zu werden. Javadoc wird im Folgenden beschrieben, weil dieser Kommentierungs-Mechanismus sowohl im Ansatz [MacDonald+ 2002] als auch in später vorgestellten Ansätzen Verwendung findet und für die vorliegende Arbeit relevant ist.

Javadoc

Javadoc ist ein proprietärer Mechanismus zur Kommentierung von Java-Quelltext. Prinzipiell können solche Kommentare überall im Programmtext stehen. Das Java-eigene Hilfsmittel für die Auswertung dieser Kommentare erlaubt aber nicht die Berücksichtigung von Kommentaren innerhalb von Methoden ([Sun 2004]). Javadoc-Kommentare sind für Entwickler bestimmt. Sie haben das folgende Format:

```
/**
 * Das ist ein Javadoc-Kommentar
 */
```

Abbildung 20: Javadoc-Kommentar in Langform.

Derselbe Kommentar lautet in verkürzter, äquivalenter Schreibweise:

```
/** Das ist ein Javadoc-Kommentar */
```

Abbildung 21: Javadoc-Kommentar in Kurzform.

Innerhalb eines solchen Kommentars können Parameter (sogenannte Tags) verwendet werden, die vergleichbar mit Tags in XML sind. Parameter sind durch einen vorangestellten Klammeraffen gekennzeichnet und bestehen aus dem Namen des Parameters sowie dem zugeordneten Wert:

```
/**@Name Wert*/
```

Abbildung 22: Parameter innerhalb eines Javadoc-Kommentars.

Der hinter dem Klammeraffen stehende Text »Name« repräsentiert den Namen des Parameters. Der Text »Wert« steht stellvertretend für einen beliebigen Wert, der dem Parameter zugeordnet ist.
In Javadoc-Kommentaren werden auch HTML-Elemente unterstützt, beispielsweise:

```
/**
 * Weitere Informationen zu Javadoc finden Sie auf der
 * <a href="http://java.sun.com/j2se/javadoc/">Javadoc Homepage</a>
 */
```

Abbildung 23: Referenz (Link) innerhalb eines Javadoc-Kommentars.

Als standardisierte Kommentarform für die Sprache Java hat sich Javadoc etabliert (vgl. auch Abschnitt 2.3.3.3). Seit längerem ist die Erweiterung von Javadoc Gegenstand der Java Anforderung 260 [JSR260 2004]. Die [JSR260 2004] soll neue Konstrukte einführen und flexiblere Javadoc-Kommentare ermöglichen etwa durch Berücksichtigung unterschiedlicher Typen von Methoden. Auch der semantische Aspekt steht auf der Agenda der [JSR260 2004], dort ist dies angegeben mit dem Aufzählungspunkt „semantical index of classes and packages“. Leider gibt es keine spezifischen Informationen zu diesem Vorhaben und seit Jahren keine Neuigkeiten. Im Rahmen von generativen Ansätzen ist Javadoc ein populärer Mechanismus zur Kennzeichnung von *Protected Regions* (vgl. hierzu [NetWeaver 2003], [ANGIE 2003]).

Meta-CO_2P_2S

Während mit CO_2P_2S Entwurfsmuster über die oben genannten Parameter adaptiert werden können, dient das Werkzeug Meta-CO_2P_2S der formalen Dokumentation von Mustern in Form von Vorlagen. Die Vorlagen werden intern in Form von XML-Dateien persistiert. Wie solche XML-Dateien aussehen, konnte anhand der verfügbaren Literatur nicht ermittelt werden.[20]

[20] Auf der CO_2P_2S-Webseite [CO_2P_2S 2003] ist der Link zum Thema „Pattern Template Repository“ mit einer leeren Seite verknüpft.

Bei der Generation eines ausgewählten Musters wird XSLT[21] als Transformationssprache verwendet. Meta-CO_2P_2S ist nicht beschränkt darauf, nur für CO_2P_2S Eingabedateien bereitzustellen. Es bietet eine allgemeine Basis für andere Werkzeuge. Zu Meta-CO_2P_3S existiert nur wenig Literatur. Die Publikationsliste zu CO_2P_2S und Meta-CO_2P_2S deutet darauf hin, dass die Arbeit an diesen Projekten seit längerem eingestellt wurde. Die Nachfolgerprojekte Meta-CO_2P_3S und Meta-CO_2P_3S (das hinzugekommene »P« steht für »Parallel«) sind zwar etwas neuer, aber ebenso am Ende des Lebenszyklus angelangt, was aus dem Alter verfügbarer Arbeiten hierzu hervorgeht.

Fazit

Die Werkzeugunterstützung durch CO_2P_2S ist für die generative Entwicklung von Frameworks ausgelegt und somit fokussiert auf eine konkrete Domäne. Die Schwierigkeit beim Entwurf von Frameworks ist die für eine bestimmte Domäne geeignete Granularität zu treffen. Ist die Körnigkeit zu fein, ist das Framework zu komplex, ist sie zu grob ist das Framework zu abstrakt und nicht leistungsfähig genug. Zudem ist es schwierig, ein für eine gegebene Domäne gut passendes Framework zu identifizieren.
Bei der Arbeit mit dem Werkzeug muss der Benutzer nicht nur ein Entwurfsmuster manuell auswählen, sondern dieses auch in eine für den Zielentwurf geeignete Form bringen. Deshalb eignet sich der Ansatz nicht für eine weitere Betrachtung für einen domänenübergreifenden Ansatz.
Die Klassifikation von Musterparametern erscheint ebenfalls nicht ausreichend, da diese nicht feingranular genug ist. Die Parameterklasse *Design* etwa ist sehr allgemein auf die Anwendung bzw. Generation von Mustern ausgerichtet. Das Vorgehen zur Markierung von Slots und geschützten Regionen hingegen kann übernommen werden.
Die Generierung von Nachbedingungen für mit dem Muster verknüpfte Parameter erscheint sinnvoll. Es ist zu untersuchen inwieweit eine solche Generierung für domänenübergreifend verwendbare Muster realisierbar ist.

2.3.3.4 PSE-Handler

PSE[22]*-Handler* (kurz: *PSE-H*) [Mayr-Kern+ 2002] erlaubt die Anwendung zuvor definierter Muster. *PSE-H* ist Bestandteil der bereits genannten Muster-Applikation *PSE*-Base, die auch *PSE-D*, *PSE-V* und *PSE-P* enthält. *PSE-H* ist ein *Eclipse*-Plugin. Es kann somit integriert im Entwicklungsprozess verwendet werden.
Die Schritte zur Anwendung eines Entwurfsmusters sind analog denen von *objectiF* (Zuweisung von Rollen zu Instanzen, einfache Plausibilitätsprüfungen, Umsetzen der Zuweisungen im Quelltext). Ein wesentlicher Unterschied ist, dass Muster mittels *PSE-D* definiert werden und nicht auf Basis von UML-Diagrammen. Weiterhin vermerkt *PSE-H* beim Anwenden eines Musters durch Javadoc-Kommentare, welches Entwurfsmuster angewandt und welche Rollen zugewiesen wurden [Mayr-Kern+ 2002, S. 98].

Fazit

Die Möglichkeit der Integration eines Verfahrens (hier: *PSE-H*) in den Entwicklungsprozess ist vorteilhaft. Das prinzipielle Verfahren von *PSE-H* zur Entwurfsmusteranwendung gleicht allerdings dem herkömmlichen Vorgehen. Es setzt intensive Kenntnisse über Entwurfsmuster voraus. Genau diese Voraussetzung ist jedoch ein Grund für die mangelnde Durchsetzung von Entwurfsmustern in der kommerziellen Software-Entwicklung.
Wie in [Mayr-Kern+ 2002, S. 82] beschrieben, war die Entwicklung von Plugins für *Eclipse* teilweise problematisch, da die Stabilität von *Eclipse* noch nicht entsprechend gut war. Mittlerweile ist die Verlässlichkeit von Eclipse akzeptabel. Für eine Umsetzung einer integrierten Unterstützung für den

[21] XSLT = XSL Transformation, (XSL = Extensible Stylesheet Language), dient zur Transformation von XML-Dokumenten.
[22] *PSE* = Pattern Support for Eclipse

Entwickler bei der Arbeit mit Mustern ist die weit verbreitete Plattform *Eclipse* nach aktuellem Stand eine sinnvolle Entscheidung.

2.3.4 Erkennung angewandter Muster in Legacy Code

Nachdem ein Muster auf einem Quelltext angewandt wurde, kann es aus mehreren Gründen sinnvoll sein, dieses Muster später und ohne bisheriges Wissen von dessen Anwendung zu identifizieren. Erstens ist die Erkennung bereits vorhandener oder teilweise oder verfremdet vorhandener Muster bei der Musterselektion wichtig. Denn ein Muster kann nur dann vollständig neu angewandt werden, wenn nicht schon Teile davon im Quelltext vorliegen. Bereits bestehende Teile müssen bei der Musteranwendung berücksichtigt werden. Zweitens ist die Mustererkennung nützlich, um einen gegebenen Quelltext besser verstehen und dokumentieren zu können. Bestehende Muster zu erkennen heisst, die Motivation des Entwicklers für die Wahl seiner Klassenhierarchie und des Aufbaus einzelner Klassen besser nachvollziehen zu können.
Legacy Code gestattet im Gegensatz zu einem in Entwicklung befindlichen Programm die Rekonstruktion einer zugrunde liegenden Architektur. Eine erfolgreiche Rekonstruktion bietet die Möglichkeit, die Intention des Entwicklers besser verstehen zu können. Darin eingeschlossen ist die Erkennung semantischer Strukturen. Dies führt zu einer besseren Dokumentiertheit des Systems. Insbesondere ist es nützlich, im Legacy Code vorhandene Entwurfsmuster detektieren zu können. Das ist speziell beim Anwenden von Entwurfsmustern wichtig, wo die Erkenntnis, dass ein Muster bereits besteht, von Bedeutung ist.
Eng verknüpft mit der Thematik der Architekturrekonstruktion ist das Suchen von Entwurfsdefekten in Quelltext. Diese Bereiche sind verwandt, da beide Gebiete bei der Ermittlung teilweise oder komplett vorhandener Muster in einem Quelltext helfen.

2.3.4.1 Reverse Engineering von Entwurfsmustern

In [Philippow+ 2004] wurde ein Verfahren zur automatischen Erkennung aller *GoF*-Muster (siehe [Gamma+ 1995]) in einem gegebenen Java-Programmtext entwickelt. Das Bestreben von [Philippow+ 2004] ist, durch die Identifizierung von Entwurfsmuster in vorhandenem Programmtext die Dokumentiertheit eines Legacy Systems zu erhöhen und somit die Möglichkeit zu schaffen, dieses System besser verstehen zu können.
Im Gegensatz zu anderen Ansätzen werden laut [Philippow+ 2004] alle 23 *GoF*-Muster mit einer hohen Präzision erkannt. Muster mit eindeutigen Kriterien (etwa *Factory Method*) werden, anders als dies bei dritten, dort betrachteten Ansätzen der Fall ist, immer erkannt. Ein Muster wird sowohl durch Positivkriterien identifiziert als auch durch Negativkriterien von der Erkennung ausgeschlossen. Für jedes Muster werden also Schlüsselmerkmale definiert, die zwingend vorkommen müssen. Verfeinernd dazu, werden alle Muster, die irrtümlich auch aufgrund von Schlüsselmerkmalen eines anderen Musters erkannt wurden, durch Merkmale, die nicht vorhanden sein dürfen, ausgefiltert.
Ausgehend von den Nachteilen anderer, existierender Ansätze, stellt [Philippow+ 2004, S. 45] folgende Forderungen an seinen Ansatz:

1. Alle *GoF*-Muster müssen erkannt werden.
2. Eindeutig definierte Muster wie *Factory Method* müssen sicher erkannt werden.
3. Eine Präzision von 100 Prozent ist angestrebt.

Zu Punkt 3 bemerkt der Autor, dass diese angestrebte Präzision nicht wirklich überprüfbar ist. Als Präzision bezeichnet [5, S. 147] „[…] das Verhältnis von echten zu falsch erkannten Mustern einer Suchanfrage“. Die bei der Suche nach Entwurfsmustern betrachteten Merkmale sind nach [5, S.48ff]:

- Abstrakte Klasse,
- konkrete Klasse,
- Vererbung,
- Attribute (Sichtbarkeit, Typ, Name),

- Operationen (Sichtbarkeit, Polymorphie, Rückgabetyp, Name, Parameter, Abstraktheit),
- Konstruktoren (Sichtbarkeit, Name, Parameter),
- Aggregationsbeziehung (alle Kompositionsbeziehungen werden auch als solche angesehen),
- Friend-Beziehung,
- Objekterzeugung,
- Variablenbenutzung,
- Methodenaufrufe und
- Templates.

Delegation und Assoziationsbeziehungen sind nicht aufgeführt, weil sie durch andere Merkmale abgedeckt sind. Für jedes *GoF*-Muster gibt [Philippow+ 2004] einen Algorithmus in Pseudocode an sowie ergänzend dazu die Laufzeit des Algorithmus. Der Pseudocode des Algorithmus *FindeKompositum* zum Erkennen des Vorhandenseins des *Composite*-Musters lautet (übernommen aus [5, S. 72]):

```
für jede Klasse i (Kompositum) do
  besitzt Klasse i eine 1-zu-n-Aggregation zu einer Oberklasse
  (Komponente)? ja: weiter
  besitzt Klasse i Unterklassen? nein: Muster gefunden
    ja: für jede Unterklasse j (spezialisiertes Kompositum) von
    Klasse i do
      für jede Methode k der Klasse j do
        ruft Methode keine Methode der Klasse j auf, und folgt dem ein
        lokaler Methodenaufruf?
          nein: weiter, ja: Abbruch
      od
    od
  Muster gefunden
od
```

Abbildung 24: Pseudocode für *FindeKompositum* (aus [5, S. 72]).

Die Laufzeit des Algorithmus wird mit O(n) angegeben. Der Pseudocode zeigt, dass sowohl positive Kriterien („ja: weiter“) als auch negative Kriterien („nein: weiter, ja: Abbruch“) berücksichtigt werden.

Fazit

Das Konzept der Positiv- und Negativkriterien lässt sich allgemein auf Auswahlverfahren anwenden. Merkmale, die bei der Suche nach Entwurfsmustern betrachtet werden, sind potentiell interessante Merkmale für die Auszeichnung im Programmtext. Der gesamte Prozess der Mustererkennung ließe sich in den hier vorgestellten Ansatz integrieren, um die Doppelanwendung eines Musters zu vermeiden. Die Definition der Logik zur Identifizierung vorhandener Muster ist allerdings nicht trivial, denn sie basiert darauf, Algorithmen ohne weitere Hilfestellung zu programmieren.
Die Prüfung der Bedeutung eines vorliegenden Programmtextes wird in [Philippow+ 2004] nicht vorgenommen. So kann das Muster *Façade* nur bei Eintreffen sehr eng gefasster Bedingungen korrekt erkannt werden, nicht aber bei kleinsten Abweichungen. Eine solche Abweichung kann etwa sein, dass ein in der Fassadenmethode enthaltener Methodenaufruf auch außerhalb des Fassadenmusters vorkommt. In Konsequenz bedeutet dies, dass die semantische Komponente von Quelltext und Muster stärker zu berücksichtigen ist, um bestehende Probleme, wie sie etwa in [Philippow+ 2004] vorliegen, lösen zu können.

2.3.4.2 PSE-Pattern-Finder

Der *PSE-Pattern-Finder* (kurz: *PSE-P*) [Mayr-Kern+ 2002] ist eine Teilkomponente einer Pattern-Applikation zum Auffinden vorhandener Muster in Legacy Code (siehe *PSE-H*). *PSE-P* erkennt Muster, die mit *PSE-H* angewandt wurden. Weiterhin werden Entwurfsmuster erkannt, die manuell im

Quelltext implementiert wurden, wenn entsprechende Javadoc-Kommentare verwendet wurden (siehe [Mayr-Kern+ 2002, S. 57]).

Fazit

PSE-P zeigt, dass die Einbeziehung von Javadoc-Kommentaren ein praktikabler Ansatz ist. Die Erkennung von Mustern ist hauptsächlich nur möglich, wenn diese zuvor mit einem korrespondierenden Werkzeug angewandt wurden. Damit ist *PSE-P* für Legacy Code mit manuell ausimplementierten Entwurfsmustern relativ uninteressant.
Da *PSE-P* nur Muster findet, die zuvor mit Hilfe von *PSE-H* eingebracht wurden oder analog mit Kommentaren versehen wurden, ist die Anwendbarkeit des Ansatzes zur Musterfindung stark eingeschränkt. Was fehlt, ist zumindest ein Mechanismus, der die vorausgesetzten Javadoc-Kommentare in einer Vorstufe automatisch erzeugt, falls diese nicht vorhanden sind. Das würde das Problem der Mustererkennung jedoch um eine Stufe nach hinten verschieben, da gerade die Ermittlung derartiger Kommentare eine nichttriviale Aufgabe ist.

2.3.4.3 ESC/Java

Das Projekt *ESC/Java*[23] [Flanagan+ 2002] hat sich zum Ziel gesetzt, einen verifizierenden Compiler bereitzustellen. Der Compiler soll – ähnlich den aus objektorientierten Sprachen bekannten Typprüfungen – zur Kompilierzeit formale Verstöße gegen manuell vorgegebene Bedingungen erkennen. *ESC/Java* ist nicht auf Entwurfsmuster spezialisiert, sondern ist allgemein gehalten.
Bedingungen werden mit Hilfe von speziellen Kommentaren im Quelltext angebracht. Solche Kommentare werden Pragmas genannt. Der Sprachumfang der Pragmas orientiert sich an der *JML* (siehe [Leavens+ 2003] und weiter unten). Ein Pragma kann sowohl für einen Konstruktor, eine Methode als auch für Felddeklarationen oder Anweisungen angegeben werden. Das folgende Beispiel zeigt, wie eine Invariante für einen Eingabeparameter angegeben werden kann:

```
class Bag {
...
  //@requires input!=null;
  Bag(int[] input) {
    n = input.length;
    a = input;
  }
...
}
```

Abbildung 25: Annotierter Quelltext für die Analyse mit ESC/Java.

Die fett gedruckte Zeile stellt einen Pragma-Kommentar dar und definiert, dass der Eingabeparameter namens *input* nicht *null* sein darf. Diese Information wird von *ESC/Java* einerseits bei der Prüfung von Aufrufen des *Bag*-Konstruktors verwendet. Andererseits kann *ESC/Java* bei der Prüfung der Logik innerhalb des *Bag*-Konstruktors voraussetzen, dass die Variable *input* nicht *null* ist.

Weitere von *ESC/Java* bereitgestellte Pragmas sind u.a.:

- *assert*: Bedingung die an einer Stelle im Programm erfüllt sein muss
- *invariant e*: Bedingung für Objekt *e*, die immer gelten muss
- *unreachable*: Die so gekennzeichnete Stelle im Programm darf nie erreicht werden
- *assume e*: Das Prädikat *e* wird als geltend angegeben, es kann aber nicht automatisch gefolgert werden

[23] ESC/Java = *Extented Static Checker for Java*

Wird ein Pragma von *ESC/Java* beim Kompilieren als verletzt erkannt, werden Warn- oder Fehlermeldungen ausgegeben.

Fazit

ESC/Java verwendet spezielle Kommentare zum Einbringen von Metainformationen in Quelltext. Da Kommentare die Kompilierbarkeit und das Verhalten eines Programms nicht verändern, bleibt der für *ESC/Java* modifizierte Quelltext für andere Compiler verarbeitbar.
Die von *ESC/Java* bereitgestellten Pragmas erlauben das Festlegen von formalen Bedingungen für Programmteile oder Variablen. Somit ist eine punktuelle Programmverifikation möglich. Flüchtigkeitsfehler und kleinere Entwurfsdefekte können somit erkannt werden. Eine vollständige Programmverifikation kann allerdings nicht erreicht werden. Die angebotenen Mechanismen von *ESC/Java* reichen hierfür nicht aus. Auch können keine Bedeutungen von Programmteilen angegeben werden. Mit dem Mittel der Pragmas in Form von speziellen Kommentaren könnte allerdings die Semantik von Programmteilen spezifiziert werden, wenngleich eine Programmverifikation, die Semantik berücksichtigt, daraus nicht automatisch möglich würde. Spezielle Kommentare sind also eine mögliche Strategie für die Definition von Metainformationen in Quelltexten, gegen die Prüfungen laufen können.

2.3.5 Semantische Auszeichnung von Programmtexten

In einigen diskutierten Ansätzen, etwa in [Ó Cinnéide+ 1999a] oder [Philippow+ 2004], ist das Fehlen der Betrachtung der Bedeutung von Quelltextteilen nach Meinung des Autors ein Grund für eine Schwäche des jeweiligen Ansatzes. Der Autor geht davon aus, dass die Berücksichtigung der Semantik von Quelltext und Muster essentiell für die leistungsfähige Unterstützung bei der Arbeit mit Entwurfsmustern ist.
Daher soll in diesem Abschnitt eine heute schon existierende Möglichkeit diskutiert werden, verhaltensneutrale Informationen mit Quelltexten zu verknüpfen. Diese Möglichkeit der Verknüpfung bietet die sogenannte Auszeichnung von Programmtext. Als Auszeichnung von Programmtext wird im Rahmen dieser Arbeit das Verknüpfen von Informationen mit bestimmten Teilen eines Programms verstanden. Derartige Informationen werden auch Metainformationen genannt, da sie Informationen über Informationen (Teile des Programmtextes) bereitstellen.
Es ist prinzipiell zu unterscheiden zwischen Auszeichnungen, die formalen Regeln genügen, und informalen textuellen oder graphischen Anmerkungen. Letztere werden etwa von Lehrern verwendet, die die Arbeit ihrer Schüler durchsehen. In dieser Arbeit liegt der Fokus alleine auf formalen Auszeichnungen, die möglichst leicht von einem Menschen verstanden werden können.

Eine Auszeichnung wird im Folgenden auch als Annotation bezeichnet und wie nachstehend definiert.

Definition: Annotation (Version 1)

> Eine Annotation ist ein wohldefinierter Kommentar innerhalb eines Programmtextes, der semantische Informationen zu einem Programmelement beinhaltet. Ein wohldefinierter Kommentar ist ein Kommentar, der fest definierten Regeln gehorcht und eindeutig von anderen Kommentaren unterscheidbar ist. Der Gültigkeitsbereich (engl.: *Scope*) der Annotation ist festgelegt entweder durch ihre Ausprägung, die nur einen Gültigkeitsbereich zulässt; oder durch ihre Position, der das in Bezug stehende Programmelement unmittelbar folgt. Die Bedeutung einer Annotation hängt nur von deren Interpretation ab.
>
> Synonym werden die Begriffe »Auszeichnung«[24] und »semantische Auszeichnung« verwendet.

Der Grund für die Wahl des Begriffes »Annotation« (neben dem Begriff »Auszeichnung«) ist darauf begründet, dass es in Java und C# ein Konstrukt mit selbem Namen gibt, das eben das Hinzufügen von Metainformationen zu Quelltext ermöglicht.
Die genannte Definition ist vorläufig und wird später in der Arbeit verfeinert. Die Auszeichnung (oder Annotation) von Programmtext wird im Rahmen der Arbeit mit Entwurfsmustern als wichtig erachtet. Der Grund ist, dass für die formale Musterdokumentation die dem Muster zugrunde liegenden Intentionen sowie der Kontext und die Problemstellung derart definiert werden müssen, dass für einen gegebenen Quelltext entschieden werden kann, ob die Intentionen, der Kontext und die Problemstellung des Musters zum Quelltext passen. In einem Quelltext wiederum sind die Entwurfsintentionen eines Entwicklers für populäre Sprachen wie Java oder C# nicht erkennbar (vergleiche auch die Betrachtungen zu [Ó Cinnéide+ 1999a] oder [Philippow+ 2004]).

Eine Auszeichnung kann an sich jede beliebige Information repräsentieren. Der Zusatz »semantisch« bedeutet, dass der Fokus auf Informationen liegt, die Aussagen über die Semantik des ausgezeichneten Programmteils treffen. Der Begriff »Semantik« steht für die Bedeutungslehre. Er wird wie folgt definiert.

Definition: Semantik (aus [Duden 1993])

> Lehre von der inhaltlichen Bedeutung einer Sprache. In der Informatik bezieht man sich dabei auf Programmiersprachen oder mathematische Kalküle.

Für die semantische Auszeichnung eines Programmtextes ist es notwendig, entsprechende Auszeichnungselemente im Programmtext anzubringen. Diese dürfen ein bestehendes Programm in seiner Wirkungsweise nicht verändern. Möchte man die Kompatibilität mit Entwicklungsumgebungen nicht gefährden, bietet sich die Verwendung von Kommentaren an. Die Definition von Auszeichnungen in einer anderen als der betroffenen Quelltextdatei (die eine Klasse darstellt) hingegen wird als unpraktikabel angesehen, da die Übersicht für den Entwickler bei einer derartigen Externalisierung leidet. Eine andere Möglichkeit, die weiter unten zur Abrundung des Themas vorgestellt wird, sind Spracherweiterungen. Sie ersetzen den ursprünglichen Compiler der jeweils gewählten Programmiersprache durch eine eigene, auf Kompatibilität ausgelegte Variante. Diese Variante stellt eine Obermenge zum Original dar. Anstatt den ursprünglichen Compiler zu verwenden, bindet man die Variante in die Entwicklungsumgebung seiner Wahl ein. Im Folgenden soll das Konstrukt der Annotationen am Beispiel der Programmiersprache Java erläutert werden.

[24] Der Begriff »Auszeichnung« taucht übrigens auch in den Akronymen HTML und XML in Form des Buchstabens »M« (Markup) auf.

Annotationen und Annotationstypen in Java

Eine in 2004 veröffentlichte Erweiterung der Sprache Java wird in [JCP175 2004] im Rahmen eines sogenannten *Java Specification Request* (kurz: *JSR*) gefordert. Die *JSR 175* hat den Titel „A Metadata Facility for the Java™ Programming Language“. Sie fordert die Einführung von Annotationen. Annotationen stellen auch dort Metadaten dar, die im Programmtext angebracht werden. Die Umsetzung der *JSR* geschah im Rahmen der Java Version 1.5, Codename *Tiger*. Die *Reflection API*[25] von Java wurde derart erweitert, dass externe Introspektorprogramme Annotationen, die zur Laufzeit sichtbar sind, abfragen können. Das ist vergleichbar mit der schon vorhandenen API zur Auswertung von Javadoc-Kommentaren.
Neben Java ist C# als populäre Sprache zu nennen, die Annotationen unterstützt. Es handelt sich also um ein sprachübergreifendes Konzept, das nicht auf Java eingeschränkt ist.

Annotationstypen

Jede Annotation ist einem Annotationstyp zugewiesen, der zuvor definiert werden muss. Die Deklaration eines Annotationstypen resultiert zudem in einer Schnittstelle, die zum Auslesen der entsprechenden Annotationen genutzt werden kann. Weiterhin kann ein Annotationstyp zur Definition weiterer solcher Typen verwendet werden. Da ein Annotationstyp genau wie eine herkömmliche Java-Schnittstelle definiert wird, wird er von diesem durch die Verwendung der Annotation *@interface* anstelle des Schlüsselwortes *interface* unterschieden. Es gibt einige Einschränkungen bei der Definition von Annotationstypen im Gegensatz zu Java-Schnittstellen. Beispielsweise darf ein Annotationstyp keinen anderen erweitern. Stattdessen erweitert er implizit die Markerschnittstelle *java.lang.annotation.Annotation*. Methoden dürfen darüber hinaus keine Parameter besitzen und keine Ausnahmen (engl.: *Exceptions*) werfen. Jede Methode eines Annotationstyps stellt einen sogenannten *Member* dessen dar. Die implizit von *java.lang.annotation.Annotation* und von *java.lang.Object* geerbten Methoden werden jedoch nicht als *Member* angesehen.

Ein Beispiel für eine Deklaration des Annotationstypen *RequestForEnhancement* ist in Abbildung 26 gegeben (angelehnt an ein Beispiel aus [JCP175 2004]):

```
/**
 * Describes the "request-for-enhancement" (RFE) that led to the
 * presence of the annotated API element.
 */
public @interface RequestForEnhancement {
    int    id();  // no default - Unique ID number associated with RFE
    String engineer();  // Name of engineer who implemented RFE
    String date()default "[unimplemented]";
}
```

Abbildung 26: Deklaration eines Java-Annotationstyps.

Meta-Annotationstypen

Neben Annotationstypen existieren Meta-Annotationstypen. Sie dienen der Annotation von Annotationstypen. Einige solche werden im Package *java.lang.annotation* bereits vordefiniert, etwa der Metatyp *Target*. Er schränkt die Nutzung der Typen ein, die er annotiert. So kann man festlegen, ob ein Annotationstyp für Typen, Klassen, Annotationen oder etwa Packages verwendet werden darf.

Ein Beispiel für die Beschränkung eines Annotationstyps auf Feld- und Methodendeklarationen:

[25] *API* = Application Programming Interface = Schnittstelle zum Aufrufen von Programmlogik.

```
@Target({FIELD, METHOD })
public @interface Bogus { ... }
```

Abbildung 27: Beispiel für einen eingeschränkten Java-Annotationstypen.

Der Meta-Annotationstyp wird, wie zu sehen, direkt oberhalb des Annotationstypen angegeben. Somit ist der Bezug beider Elemente zueinander definiert.

Annotationen

Eine Annotation ist ein neuer Java-Modifizierer. Er beinhaltet einen Annotationstyp und weist jedem obligatorischen Attribut einen Wert zu. Es gibt drei Arten von Annotationen:

Normal Annotations sind genereller Natur und zur Annotation von Programmelementen vorgesehen. *Marker Annotations* und *Single-member Annotations* sind Kurzformen und dienen nicht der Annotation von Programmelementen. Ein Beispiel für eine *Normal Annotation* (aus [JCP175 2004] entnommen):

```
@RequestForEnhancement(
    id       = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    //date nicht angegeben, wird aus Defaultwert gezogen
)
public static void travelThroughTime(Date destination) { ... }
```

Abbildung 28: Beispiel für die Angabe einer Java-Annotation mit Parametern.

Diese Annotation basiert auf dem zuvor definierten Annotationstyp *RequestForEnhancement*. Wie zu sehen ist, besteht die Annotation aus drei Attributen. Jedem Attribut wird ein seinem Typ entsprechender Wert zugewiesen.

Fazit zu Annotationen

Die *JSR 175* führt auf abstrakter Ebene ein, was im Rahmen dieser Arbeit in Form von Auszeichnungselementen konkret für Entwurfsmuster betrachtet wird. Die direkte Unterstützung von Annotationen in einer populären Sprache wie Java erleichtert die Implementierung eines entsprechenden Analyseprogramms für die hier vorgestellte Auszeichnungsmethodik.
Andererseits können nicht beliebige Programmelemente mit Annotationen der *JSR 175* versehen werden. Als Grund wird die Verkomplizierung der zugrunde liegenden Syntax angegeben. In diesem Punkt weicht die *JSR 175* in ihrer Spezifikation von benötigten Notwendigkeiten ab. Demnach kann die *JSR 175* nur als Konzept übernommen werden, nicht als Implementierung.
Um Verwechslungen zu vermeiden, werden Annotationen gemäß der *JSR 175* im folgenden Text explizit als solche gekennzeichnet. Ansonsten sind prinzipiell Annotationen im Rahmen des hier vorgestellten Ansatzes gemeint.

Spracherweiterungen

Einige Programmiersprachen, darunter Java, erlauben es, Spracherweiterungen (engl.: *Language Extensions*) zu entwickeln. Sie ermöglichen das Einbringen neuer Sprachkonstrukte. Bestehende, originär vorhandene Anweisungen sind davon nicht berührt. Präprozessoren übersetzen einen mit nichtproprietären Befehlen (vergleichbar Makros) angereicherten Code in einen mit dem Standard-Compiler übersetzbaren. Das hat den Nachteil, zwischenzeitlich einen nicht auf normalem Wege übersetzbaren Code zu haben. Praktikabler sind sogenannte Drop-In-Compiler. Sie werden anstelle des normalerweise verwendeten Compilers verwendet. Damit ist es möglich, auch in einer

Entwicklungsumgebung mit Spracherweiterungen zu arbeiten. Zumindest betrifft dies den Kompiliervorgang. Für Entwicklungsumgebungen spezifische Features wie Syntax-Hervorhebung oder Refactoring können derartig eingebrachte Spracherweiterungen nicht verarbeiten. Dies ginge nur dann, wenn die Entwicklungsumgebung an sich einen *Drop-In Compiler* unterstützt.
Eine Implementierung eines *Drop-In-Compilers* liegt mit Kopi vor (siehe [Kopi 2004]). Kopi beinhaltet den KJC[26], einen ebensolchen Compiler, der selbst in Java geschrieben ist. Er ist als Open-Source Projekt frei erhältlich. Er erlaubt es, individuelle Spracherweiterung zu implementieren. Der KJC selbst beinhaltet keine nennenswerten Spracherweiterungen. Er bietet lediglich Features wie Code Optimierung oder Code-Verschönerung (auch *Beautification* genannt, vgl. *BeautyJ* [Gulden 2004]) an. Die erweiterte Version, XKJC[27], ist nicht frei erhältlich. Jedoch kann der KJC aufgrund seiner Erweiterbarkeit an eigene Bedürfnisse angepasst werden, beispielsweise durch Einführung generischer Typen (auch als parametrische Polymorphie bezeichnet), die in Java ab Version 1.5 verfügbar sind.
Andere, frei verfügbare Spracherweiterungen sind beispielsweise der *Java Syntactic Extender* ([Bachrach+ 2003]) oder *OpenJava* ([Tatsubori 2003]).

Fazit zu Spracherweiterungen

Spracherweiterungen eignen sich aufgrund des Vorhandenseins eines nichtkonformen Zwischenprodukts nicht für die formale Dokumentation von Entwurfsmustern oder die Annotation von Programmtext. Diese Alternative ist deswegen uninteressant. Weiterhin bewirkten sie eine Vermischung von Programmcode und Auszeichnungen. Sie sind zu spezifisch, um für einen Ansatz geeignet zu sein, der allgemein anwendbar sein soll. Konzeptionell kann man lediglich die Überlegung übernehmen, den Wortschatz, zu erweitern, der dem Entwickler zur Verfügung steht.
Existierende Spracherweiterungen sind nicht direkt auf die formale Musterdokumentation für die Generation von Programmtext oder die Annotation von Programmelementen ausgerichtet.

2.3.5.1 Die Java Modeling Language

Im Ansatz [Flanagan+ 2002], der in Abschnitt 2.3.4.3 diskutiert wurde, gehorchen die verwendeten Annotationen in Form von Kommentaren der Spezifikation der *JML*[28].
JML ist eine Sprache für die Beschreibung von Verhalten für Java Funktionseinheiten. Zweck ist das Schaffen von Klarheit über die Intentionen des Entwicklers beim Programmentwurf.
Die *JML* basiert auf Annotationen, die als Kommentare im Quelltext angebracht werden. Somit bleibt der Quelltext mit Standardmitteln kompilierbar und behält die ursprüngliche Bedeutung bei. Die *JML* stellt u.a. die folgenden Schlüsselwörter für Annotationen bereit:

- *requires*: Definiert eine Vorbedingung einer Methode
- *ensures*: Definiert eine Nachbedingung einer Methode
- *signals*: Definiert eine Bedingung für die das Generieren einer Ausnahme erlaubt ist
- *assignable*: Legt fest, für welche Felder in der folgenden Methode Werte zugewiesen werden dürfen
- *pure*: Markiert eine Methode, die keine Seiteneffekte verursachen kann
- *invariant*: Definiert eine invariante Eigenschaft einer Klasse
- *assert*: Definiert eine Zusicherung

Weiterhin bietet die *JML* die Möglichkeit, Ausdrücke in Verbindung mit den eben genannten Schlüsselworten zu verwenden. Diese Ausdrücke ermöglichen u.a. logische Verknüpfungen, Vergleiche, die Referenz des Rückgabewertes und den mathematischen Operator $\forall$ ("für alle…").

[26] KJC = Kopi Java Compiler
[27] XKJC = eXtended Kopi Java Compiler
[28] *JML* = Java Modeling Language

Außerdem können in Methodenannotationen Objekte referenziert werden, die im Zugriffsbereich der annotierten Methode liegen.
Ein Beispiel für eine annotierte Methode (der Webseite [Wik 2006] entnommen und gekürzt):

```
public class BankingExample {
    public static final int MAX_BALANCE = 1000;
    private int balance;

    //@ invariant balance >= 0 && balance <= MAX_BALANCE;
    //@ assignable balance;
    //@ ensures balance == 0;
    public BankingExample() { ... }

    //@ requires amount > 0;
    //@ assignable balance;
    ...
    public void credit(int amount) { ... }
    ...
}
```

Abbildung 29: Beispiel für eine mit der JML annotierte Klasse.

Die erste auftretende Annotation in Abbildung 29 lautet:

```
//@ invariant balance >= 0 && balance <= MAX_BALANCE;
```

Sie besagt, dass die Variable *balance* innerhalb der Klasse *BankingExample* immer zwischen dem Wert 0 und dem Wert 1000 liegen muss.
Mit Hilfe von Annotationen der *JML* und einer statischen oder auch dynamischen Programmanalyse kann ein Quelltext auf Korrektheit verifiziert werden. Dedizierte Fehlermeldungen, die etwa die Art der Verletzung einer Zusicherung sowie den Verursacher benennen, können so durch ein Verifikationsprogramm generiert werden.

Fazit

Mit der *JML* können für einen Quelltext Eigenschaften festgelegt werden, die automatisiert nicht folgerbar erscheinen. Dazu bedient sich die *JML* Annotationen in Form von speziellen Kommentaren. Dieses Vorgehen soll im Rahmen dieser Arbeit weiter verfolgt werden. Es soll allerdings im Gegensatz zur *JML* eine mächtigere Menge an Annotationen bereit gestellt werden. Dies kann entweder durch Folgen des *JML*-Ansatzes geschehen, indem weitere Annotationen statisch vorgegeben werden. Oder es ist ein Mechanismus zu schaffen, der die Deklaration eigener Annotationen erlaubt.
Die Ausrichtung der *JML* ist technisch. Eine fachliche Semantik kann mit ihr nicht ausgedrückt werden. Gerade dies ist aber für Entwurfsmuster erforderlich und muss in anderer Form als mit Standardmitteln der *JML* umgesetzt werden.

2.3.5.2 XDoclet

XDoclet [XDoclet 2004] bedient sich Annotationen, um Java-Quelltextteilen Metadaten zuzuordnen, die letztendlich verwendet werden, um Text zu generieren. Annotationen werden für *XDoclet* in Form von speziellen Kommentaren direkt über der zu annotierenden Quelltextstelle angebracht. Unterstützt werden die Programmelemente Klasse, Methode und Feld.

Mit Hilfe von vorher auszuwählenden Vorlagen (engl.: *Templates*) wird anhand der Quelltext-Annotationen ein Generat erzeugt. Das Generat kann beispielsweise ein neuer Quelltext oder eine XML-Datei sein. Die Vorlagen werden in einer von XDoclet eigens eingeführten Vorlagensprache erstellt.

Fazit

XDoclet kann in erster Linie als Generator angesehen werden, der durch einfach anzubringende Annotationen sowie eine vorgegebene Vorlage gesteuert wird. *XDoclet* ist aufgrund der Leistungsfähigkeit als Generator im Java-Umfeld weit verbreitet. *XDoclet* verwendet Annotationen als Ausdrucksmittel für Steueranweisungen, nicht um einem Quelltext Semantik hinzuzufügen. Somit ist *XDoclet* ebenso wie die *JML* rein technisch orientiert und nicht für semantische Aussagen geeignet.

2.3.6 Zusammenfassung

In Kapitel 1 wurde die Frage gestellt, wie die Arbeit mit Mustern auf Quelltextebene erleichtert werden kann. Danach wurden in Kapitel 2 Ansätze vorgestellt, die sich mit der formalen Dokumentation, der Selektion, der Anwendung neuer bzw. der Erkennung vorhandener Entwurfsmuster sowie mit der Anreicherung von Quelltext mit Informationen beschäftigen. Jeder Ansatz wurde im Hinblick auf die Problemstellung dieser Arbeit beurteilt. Bei dieser Betrachtung wurden Ideen und Konzepte identifiziert, die für eine weitere Untersuchung und Berücksichtigung sinnvoll erscheinen. Ebenso wurden in den Ansätzen Bestandteile gefunden, die nach Ansicht des Autors für weitere Überlegungen nicht relevant erscheinen.
Tabelle 5 zeigt die vorgestellten Arbeiten und ihre Ausrichtung im Überblick. Sie gibt das Ergebnis der Betrachtungen in verkürzter Form wieder. Die Tabellenspalten enthalten für die Auswahl und Anwendung eines Entwurfsmusters relevante Aktivitäten. Ein „Ja“ steht für einen Unterstützung der Aktivität durch einen Ansatz, ein „Nein“ für das Gegenteil. Mit einem Fragezeichen wurden Kriterien gekennzeichnet, für die keine Aussage möglich war. Der Begriff „Hilfsmittel“ deutet an, dass der Ansatz als Hilfsmechanismus zur Realisierung der Aktivität angesehen werden kann, nicht aber als Umsetzung der Aktivität. Ein Strich bedeutet, dass die entsprechende Aktivität für den jeweiligen Ansatz nicht relevant ist. Das Prüfen von Nachbedingungen etwa ist bei der Erkennung vorhandener Entwurfsmuster im Rahmen des Reverse Engineering nicht relevant.
Die Aktivität »Reverse Engineering« beschreibt die Fähigkeit eines Ansatzes, in einem bestehenden Programmtext vorhandene Entwurfsmuster identifizieren zu können. Die Aktivität »Nachbed. Prüfen« ist dafür geeignet, nach Anwendung eines Musters zu prüfen, ob die im Programmtext betroffenen Stellen definierten Nachbedingungen gehorchen. Diese Nachbedingungen werden bei den hier vorgestellten Ansätzen durch den Menschen vorgegeben.
Zum Vergleich ist das Ziel dieser Arbeit in der Tabelle 5 ebenfalls angeführt und klassifiziert.

Aktivität → Ansatz ↓	Entwurfsmuster formal definieren	Reverse Engineering	Kontext für ein Muster ermitteln	Muster-Auswahl	Muster-Anwendung	Nachbed. prüfen
[Albin-Am.+ 2001a], [Guéhéneuc 2003]	Ja	Ja	Manuell	Manuell	Ja	-
[Zimmer 1997]	Ja	-	Manuell	Manuell	Ja	Ja
[Pree 1994][Pree 1995]	Teilweise	Nein	Teilweise?	Teilweise	Nein	Nein
CO_2P_2S [Tan+ 2003]	Ja (über Meta- CO_2P_2S)	-	- (Generator)	Manuell	Ja	?
Meta- CO_2P_2S [Tan+ 2003]	Ja	-	-	-	-	-
Diese Arbeit (Ziel)	Ja	Bedingt	Ja	Ja	Ja	Teils
[Ó Cinnéide+ 1999a], [Ó Cinnéide+ 1999b]	Ja	-	Manuell	Manuell	Ja	Ja
[Philippow+ 2004]	Ja	Ja	-	-	-	-
[Sang-Uk 2002], [Sang-Uk 2003a], [Sang-Uk+ 2003b]	-	Ja	Ja	Ja	Ja	?
[Guéhéneuc+ 2001]	Ja	Ja	Ja	Teilweise	Ja	?
ANGIE [ANGIE 2003]	Ja	nein	-	-	Ja	-
[Kazman+ 2002]	-	Ja		-	-	-
[Taibi+ 2003]	Ja	-	-	-	-	-
[Motelet 2004]	-	-	Hilfsmittel	Hilfsmittel	-	-
LePUS [Eden+ 2004]	Ja	Ja	?	Teilweise		?
PSE-D/P/H [Mayr-Kern+ 2002]	Ja	Ja	Nein	Manuell	Ja	-
Annotationen [JCP175 2004]	-	-	Teilweise	-	-	-
[PBE 2004a]	Ja	-	Nein	Manuell	Nein	-
JML [Flanagan+ 2002]	Hilfsmittel	Hilfsmittel	Hilfsmittel	Hilfsmittel	Hilfsmittel	Ja
ESC/Java [Flanagan+ 2002]	-	-	-	Hilfsmittel	-	Ja

Tabelle 5: Eignung vorgestellter Arbeiten für musterbezogene Aktivitäten.

Ein Klassifikationsmaßstab für die Mächtigkeit eines Ansatzes ist der Umfang der unterstützten Entwurfsmuster. Hier gibt es zwei populäre Klassifikationsmengen. Erstens die Menge der 23 *GoF*-Muster [Gamma+ 1995]. Zweitens die Unterteilung unterstützter Muster in Erzeugungs-, Verhaltens- und Strukturmuster nach [Gamma+ 1995]. Dementsprechend schlüsselt Tabelle 6 die vorgestellten Ansätze auf.

Zusammenfassend werden aus den vorgestellten Arbeiten die in Tabelle 7 genannten Ansätze, Konzepte und Ideen als generell betrachtendwert angesehen. In der Spalte »Relevante Aktivität« sind die Aktivitäten innerhalb der Musteranwendung angegeben, für die die links daneben stehenden Elemente bedeutsam sind. Die dargestellten Elemente wurden unter dem Gesichtspunkt der möglichen Eignung für die Entwicklung des hier vorgestellten Ansatzes ausgewählt. Nähere Begründungen finden sich in den weiter oben in diesem Kapitel in den Fazit-Abschnitten der vorgestellten Ansätze.

Zur Abgrenzung sind in Tabelle 8 die Elemente der dargestellten Ansätze aufgelistet, die im Rahmen des hier vorgestellten Ansatzes für eine weitere Verfolgung nicht geeignet erscheinen. Es handelt sich um ein verkürztes Fazit der zuvor diskutierten Ansätze. Die Begründung für die Klassifizierung der nachfolgend dargestellten Elemente als ungeeignet findet sich somit in den Fazit-Abschnitten zum jeweiligen Ansatz.

→Anwendbarkeit Ansatz ↓	Entwicklungs-Phase	Entwurfsmuster-Typus[29]
PADL [Baroni+ 2003b]	Neuentwicklung, Reengineering	B, C, S
[Zimmer 1997]	Neuentwicklung, Reengineering (nur Frameworks)	B, C, S[30]
CO_2P_2S [Tan+ 2003]	Neuentwicklung (nur Frameworks)	B, C, S
Meta- CO_2P_2S [Tan+ 2003]	Neuentwicklung (nur Frameworks)	B, C, S
Diese Arbeit (Ziel)	Alle Phasen	Prinzipiell alle
[Ó Cinnéide+ 1999a], [Ó Cinnéide+ 1999b]	Reengineering	C, S
[Philippow+ 2004]	Reengineering	Alle aus [Gamma+ 1995]
[Sang-Uk 2002], [Sang-Uk 2003a], [Sang-Uk+ 2003b]	Reengineering (mittels Delta-Versionen)	C [Sang-Uk 2002]
[Guéhéneuc+ 2001]	Reengineering	?[31]
ANGIE [ANGIE 2003]	Neuentwicklung Reengineering	B, C, S
[Kazman+ 2002]	Reengineering	?
[Pree 1994][Pree 1995]	Reengineering (Frameworks)	B (*Template Method*)
[Taibi+ 2003]	Nicht relevant, da Musterbeschreibung	B, C, S
[Motelet 2004]	Alle Phasen	Prinzipiell alle
LePUS [Eden+ 2004]	?	B, C, S mit Ausnahmen
PSE-D/P/H [Mayr-Kern+ 2002,Wöckl 2002]	PSE-D/H: Alle Phasen PSE-P: Reengineering	Prinzipiell alle
Annotationen [JCP175 2004]	Alle Phasen	-
[PBE 2004a]	Alle Phasen	Prinzipiell alle
JML [Flanagan+ 2002]	Alle Phasen	Nicht auf Entwurfsmuster festgelegt
ESC/Java [Flanagan+ 2002]	Reengineering	Nicht auf Entwurfsmuster festgelegt

Tabelle 6: Anwendungsbereich vorgestellter Arbeiten.

[29] Vom Ansatz unterstützte Typen: B = *Behavioural*/Verhalten, C = *Creational*/Erzeugung, S = *Structural*/Strukturierung

[30] Annahme, keine explizite Aussage in [Zimmer 1997]

[31] Maßstab sind Entwurfsdefekte der Kategorien *Intra-class* (innerhalb einer Klasse), *Behavioural* (Semantikeigenschaften) und *Inter-class* (zwischen Klassen).

Ansatz	Adaptierbare Elemente	Relevante Aktivität
PADL [Baroni+ 2003b]	Beschreibung von Struktur und Verhalten der Lösung eines Entwurfsmusters; Bereitstellen eines graphischen Werkzeuges zur Entkopplung von der Definitionssprache	Musterdokumentation
[Zimmer 1997]	Konzept der Transformationsschritte	Musterdokumentation
	Konzept der Transformationsschritte, Parametrisierung durch den Benutzer	Musteranwendung
CO_2P_2S [Tan+ 2003]	Festlegen von Typen von Musterparametern	Musterdokumentation
	Festlegen von Typen von Musterparametern	Musteranwendung
Meta- CO_2P_2S [Tan+ 2003]	Entkopplung von Erstellung einer Musterdokumentation und Auswertung dieser	Musterdokumentation
	Entkopplung von Erstellung einer Musterdokumentation und Auswertung dieser	Musterselektion
[Ó Cinnéide+ 1999a], [Ó Cinnéide+ 1999b]	Hilfsfunktionen und Prädikate zur Identifikation notwendiger Programmbestandteile zur Musteranwendung; Zerlegung eines Gesamtproblems in Teilprobleme	Musteranwendung
	Zerlegung eines Gesamtproblems in Teilprobleme; Minipatterns als Bestandteil einer Ontologie zur Beschreibung von Musteraspekten	Musterdokumentation
	Zerlegung eines Gesamtproblems in Teilprobleme	Musterselektion
[Philippow+ 2004]	Positiv- und Negativkriterien; Integration des Reverse Engineering Prozesses an sich zur Erkennung bereits angewandter Muster	Musterselektion
[Sang-Uk 2002], [Sang-Uk 2003a], [Sang-Uk+ 2003b]	Konzept der *Candidate Spots* (realisierbar durch Auszeichnungen)	Musterdokumentation
	Konzept der *Candidate Spots* (realisierbar durch Auszeichnungen)	Quelltextannotation
[Guéhéneuc+ 2001]	Konzept der Suche nach Indikationsstrukturen	Quelltextannotation
ANGIE [ANGIE 2003]	Frame-Technologie, Quelltext ist Entität erster Klasse	Musterdokumentation
	Frame-Technologie	Musteranwendung
[Kazman+ 2002]	Konzept der Workbench	Gesamtprozess
[Pree 1994]	Konzept der *Frozen Spots* und *Hot Spots*	Musterdokumentation

Ansatz	Adaptierbare Elemente	Relevante Aktivität
[Taibi+ 2003]	Spezialisierte Lösung für Entwurfsmuster	Gesamtprozess
	Spezialisierte Lösung für Entwurfsmuster; Konzept der Entitäten	Quelltextannotation
	Unterscheidung zwischen den Aspekten Struktur und Verhalten von Entwurfsmustern; Ausgewogene Berücksichtigung von Struktur und Verhalten eines Musters	Musterdokumentation
[Motelet 2004]	Konzept des kontextuellen Hilfesystems	Gesamtprozess
LePUS [Eden+ 2004]	Konzept der *Lattices*; Unterscheidung zwischen den Aspekten Struktur und Verhalten von Entwurfsmustern	Musterdokumentation
Spracherweiterungen	Möglichkeit der Erweiterung des technischen Wortschatzes zur Erhöhung der Flexibilität	Musterdokumentation
	Möglichkeit der Erweiterung des technischen Wortschatzes zur Erhöhung der Flexibilität	Quelltextannotation
Annotationen [JCP175 2004]	Integrierte, standardisierte Möglichkeit zur semantischen Auszeichnung	Musterdokumentation
	Konkretes Konzept der *JSR 175*; API zur Abfrage von Annotationen im Quelltext	Quelltextannotation
PSE-D/P/H [Mayr-Kern+ 2002, Wöckl 2002]	Integration in Entwicklungsumgebung; Schnittstelle nach außen (*PSE-D*)	Gesamtprozess
	Unabhängigkeit von Programmiersprache (*PSE-D*)	Musterdokumentation
	Unabhängigkeit von Programmiersprache (*PSE-D*); Javadoc-Kommentare zur Deklaration von Semantik (*PSE-P*)	Quelltextannotation
	Erkennung angewandter Muster nach vorhergehender Entwurfsmusteranwendung mit korrespondierendem Werkzeug (*PSE-P*) einfach möglich	Musteranwendung
[PBE 2004a]	Dokumentation von Mustern durch Beispielvorlage; Definition von minimal benötigten Auszeichnungselementen für ein Muster durch Beispielvorlage	Musterdokumentation
JML [Flanagan+ 2002]	Prinzip der Quelltext-Annotationen	Gesamtprozess
ESC/Java [Flanagan+ 2002]	Prinzip der Quelltext-Annotationen; verifizierender Compiler	Musterselektion und -verifikation

Tabelle 7: Betrachtenswerte Elemente vorgestellter Ansätze.

Ansatz	Ungeeignete Elemente	Betroffene Aktivität
PADL [Baroni+ 2003b]	Vermischen von Strukturaspekten- und Lösungsabsichten eines Musters innerhalb einer einzigen Beschreibungsmöglichkeit; Quelltext hat untergeordnete Stellung	Musterdokumentation
[Zimmer 1997]	Einschränkung auf Frameworks	Gesamtprozess
	Beziehungen zwischen Entwurfsmustern	Musterdokumentation
	Notwendigkeit einer Demoanwendung	Musterselektion
	Einschränkung auf Frameworks; Nachbedingungen	Musteranwendung
CO_2P_2S [Tan+ 2003]	Einschränkung auf Frameworks	Gesamtprozess
Meta- CO_2P_2S [Tan+ 2003]	Keine bekannt	
[Ó Cinnéide+ 1999], [Ó Cinnéide+ 1999*]	*Precursor* mit einem Einstiegspunkt; Manuelle Definition von Minitransformationen oder Vergleichbarem	Musteranwendung
[Philippow+ 2004]	Nichttriviale Definition von Erkennungsregeln	Musterselektion
[Sang-Uk 2002],[Sang-Uk 2003a], [Sang-Uk+ 2003b]	Vergleich korrelierter Programmversionen bedeutet eine zu geringe Anwendbarkeit des Ansatzes	Musterselektion
[Guéhéneuc+ 2001]	Verwendung exotischer Hilfsmittel wie der Programmiersprache *Claire*	Musterdokumentation
ANGIE [ANGIE 2003]	Nichtstandardisierte Skriptsprache, die nicht integrierbar ist in verbreitete Entwicklungsumgebungen	Musterdokumentation; Quelltextannotation
[Kazman+ 2002]	Vergleich von Ist- und Sollarchitektur bedeutet stark eingeschränkte Anwendbarkeit des Ansatzes	Musterselektion
[Pree 1994]	Einschränkung auf Frameworks	Gesamtprozess
[Taibi+ 2003]	Eingeschränkte Anwendbarkeit des Ansatzes	Gesamtprozess
	Zu formale Spezifikation	Musterdokumentation
[Motelet 2004]	Keine (da nur als Ergänzung gedacht und geeignet)	
LePUS [Eden+ 2004]	Eingeschränkte Anwendbarkeit des Ansatzes (nicht alle Muster beschreibbar); Zu formale Spezifikation; Unterschiedliche Gewichtung von Struktur- und Verhaltensaspekten	Musterdokumentation
Spracherweiterungen	Nicht standardisiert	Musterdokumentation; Quelltextannotation
Annotationen [JCP175 2004]	Mangelnde Generalität (nicht alle Programmelemente sind annotierbar), keine Standardisierung, eingeschränkte Form	Quelltextannotation

Ansatz	Ungeeignete Elemente	Betroffene Aktivität
JML [Leavens+ 2003]	Eingeschränkte Form, Blöcke nur bedingt annotierbar	Quelltextannotation
PSE-D/P/H [Mayr-Kern+ 2002, Wöckl 2002]	Umfangreiche Kenntnisse über Entwurfsmuster sind erforderlich (*PSE-H*)	Gesamtprozess
	Keine graphische Unterstützung bei der Rollendefinition	Musterdokumentation
[PBE 2004a]	Rein generativer Ansatz, Einweben eines Musters nicht möglich	Musteranwendung
JML [Flanagan+ 2002]	Eingeschränkter Wortschatz; Nicht auf Entwurfsmuster spezialisiert	Gesamtprozess
ESC/Java [Flanagan+ 2002]	Eingeschränkter Wortschatz; Spezifischer Compiler notwendig; Stark eingeschränkte Verifikationsmöglichkeiten	Musterselektion und -verifikation

Tabelle 8: Als ungeeignet erachtete Elemente vorgestellter Ansätze.

Die beschriebenen geeigneten und ungeeigneten Elemente vorgestellter Ansätze werden im Rahmen dieser Arbeit bei der Entwicklung einer Lösung und zur Formulierung der erweiterten Problemstellung berücksichtigt. Die erweiterte Problemstellung folgt im nächsten Absatz, die Lösungsbeschreibung im darauf folgenden Kapitel.

2.4 Erweiterte Problemstellung

Im vorigen Abschnitt 2.3.6 wurden die aus Sicht des Autors positiven und negativen Konzepte der betrachteten Ansätze zusammengestellt. Aus der bisherigen Betrachtung der Ansätze an sich können die jeweils vorhandenen Probleme der Ansätze durch Auswerten der negativen Eigenschaften abgeleitet werden. Die Erkenntnisse der positiven und negativen Konzepte werden mit der bisherigen Problemdarstellung aus Kapitel 1.1 verknüpft, es wird weiterhin eine Konzentration auf ein spezifisch zu lösendes Problem vorgenommen. Im Folgenden resultiert daraus die erweiterte Problemstellung dieser Arbeit. Sie wird in mehrere Aspekte unterteilt wiedergegeben. Der erste betrachtete Aspekt ist die allgemeine Anwendbarkeit des Ansatzes.

2.4.1 Allgemeine Anwendbarkeit des Ansatzes

Der Ansatz soll sich sowohl auf Entwurfs- als auch auf Architekturmuster beziehen, da angenommen wird, dass diese Musterarten analog behandelt werden können (siehe auch Abschnitt 2.2.1). Refactoring-Operationen finden sich als notwendige Transformation einiger Entwurfsmustern wieder[32] und können aufgrund ihrer geringeren Komplexität ohne Weiteres ebenfalls in den Anwendungsbereich des Ansatzes mit einbezogen werden.

Grundproblem bei einigen betrachteten Ansätzen war deren mangelnde Universalität. So werden dort nur Teilmengen von Mustern unterstützt, etwa nur Erzeugungsmuster. In anderen Fällen werden nur Frameworks, nicht aber komplette Programme fokussiert ([Zimmer 1997], [Pree 1994] oder [Tan+ 2003]). Einige Ansätze funktionieren nur in der Reengineering-Phase, weil sie etwa zwei historische Programmversionen benötigen oder auf Altsystemen basieren (etwa [Sang-Uk 2002] oder [Kazman+ 2002]).

Im Rahmen des zu entwickelnden Ansatzes sollen alle Musterarten unterstützt werden. Ferner soll sich das Anwendungsgebiet auf jede Art von Programmen erstrecken, also nicht nur auf Frameworks. Der Ansatz soll sich in jeder Entwicklungsphase, also sowohl im Forward Engineering als auch im Reengineering, einsetzen lassen.

2.4.2 Ausrichtung des Ansatzes

Eine These dieser Arbeit ist es, dass Quelltext in der kommerziellen Software-Entwicklung in vielen Projekten das Hauptwerkstück ist. In anderen Projekten, etwa solche die dem modellgetriebenen Ansatz folgen, sind UML-Modelle das Hauptwerkstück. Die Unterstützung sowohl von Quelltext als auch von UML-Modellen würde den Rahmen dieser Arbeit sprengen. Daher richtet sich diese Arbeit an Quelltext aus und bezieht Modelle nicht mit in die Betrachtung ein. Dies sieht der Autor dieser Arbeit auch dadurch als zulässig an, da fast alle von ihm persönlich beobachteten und begleiteten Projekte keine Modelle verwendet haben. Ein Grund mag sein, dass sich UML-Diagramme, etwa Klassendiagramme, direkt aus Quelltext generieren lassen und hier kein unmittelbarer Zeitvorteil durch direkte Verwendung der UML erkennbar ist. Im Quelltext liegen zudem Implementierungsdetails, die in UML-Diagrammen nicht verfügbar sind. Zustandsdiagramme, die

[32] Beispielsweise die Refactoring-Operation *Extract Interface*, die zum Muster *Factory Method* passt oder die Operation *Form Template Method* und das korrespondierende Muster *Template Method*.

etwa bei der Modellierung von Petrinetzen hilfreich sind, werden in kommerziellen Software-Projekten nach Erfahrung des Autors nur selten eingesetzt.
Die UML erscheint zudem für die formale Modellierung von Entwurfsmustern als nicht mächtig genug, da sie den Quelltextaspekt nicht ausreichend berücksichtigt. Semantische Aussagen lassen sich dort nur mit Hilfe von selbst zu definierenden Stereotypen treffen. Aufgrund der nicht standardisierten Verwendung von Stereotypen können diese nicht formal ausgewertet werden.

2.4.3 Unterstützte musterbezogene Aktivitäten

Ziel dieser Arbeit ist letztendlich die Entwicklung eines praktikablen, möglichst allgemein anwendbaren Ansatzes zur werkzeuggestützten Musteranwendung. Die möglichst zu unterstützenden Aktivitäten sind

- formale Musterdokumentation,
- Musterselektion und
- Musteranwendung.

Die Betrachtung dieser drei Aktivitäten im Einzelnen folgt in den nächsten Abschnitten. Die Mustererkennung (auch als Reverse Engineering von Mustern bezeichnet) wird für sich alleine nicht betrachtet. Sie hängt zwar mit der Musterselektion und -anwendung zusammen, ist aber in ihrer Charakteristik so verschieden von diesen, dass eine weitere Betrachtung hier nicht möglich ist. Der Zusammenhang zwischen Mustererkennung und -selektion sowie -anwendung besteht darin, dass Selektion und Anwendung Informationen benötigen, inwieweit ein zu selektierendes bzw. anzuwendendes Muster bereits vorhanden ist.

2.4.3.1 Formale Musterdokumentation

Die formale Dokumentation von Entwurfsmustern beinhaltet die Definition der statischen und dynamischen Elemente. Diese können in Anlehnung an [Zimmer 1997] und [PBE 2004a] durch Demonstration mit Hilfe eines geeignet gewählten Beispiels definiert werden. Das auf einem Beispiel basierende Verfahren hilft auch bei der Vermeidung von generell manuell zu definierenden Minitransformationen (siehe [Ó Cinnéide+ 1999a]). [Ó Cinnéide+ 1999a] befürwortet mit dem Konzept des *Precursors* die Einführung genau eines Einstiegspunktes bei der Musterdokumentation. Das hält der Autor dieser Arbeit für zu unflexibel und bevorzugt ein an [PBE 2004a] angelehntes Verfahren. Auch [Zimmer 1997] enthält wie [PBE 2004a] das Konzept der Transformationsschritte. Transformationen werden in [Zimmer 1997] als Operationen bezeichnet und stehen im Kontrast zu bausteinartigen Ersetzungen. Vielmehr wird das Anwenden eines Entwurfsmusters in [Zimmer 1997] als Verschmelzungsprozess betrachtet.
Der Versuch, Entwurfsmuster in Beziehung zueinander zu setzen (vgl. [Zimmer 1997]), erscheint allerdings als nicht vielversprechend, da ein abgeleitetes Muster durchaus eine andere Intention verkörpern kann als das Original.
Wichtig bei der Entwicklung einer Möglichkeit, Muster formal zu dokumentieren, ist deren Anwendbarkeit auf alle möglichen Arten von Mustern. Ansätze wie [Eden+ 2004] sind im Kontrast dazu nicht auf alle Entwurfsmuster anwendbar.

Statische und dynamische Quelltextelemente des Musters lassen sich mit einer Scriptsprache wie [ANGIE 2003] beschreiben. Statische Elemente können als *Frozen Spots*, dynamische als *Hot Spots* angesehen werden (vgl. [Pree 1994]).

Meta-Patterns, wie sie [Pree 1994] beschreibt, lassen sich nur für ganz wenige Muster wie *Template Method* einsetzen und sind generell nicht geeignet zur formalen Dokumentation von Mustern innerhalb eines an [PBE 2004a] orientierten Ansatzes. Daher können *Meta-Patterns* nach [Pree 1994] nicht als universeller Bestandteil einer Lösung integriert werden, sondern höchstens als spezielle Ausprägung eines allgemeineren Konzeptes.

Genau wie in [Taibi+ 2003] soll die formale Dokumentation von Entwurfsmustern auf Muster spezialisiert sein. Vorteilhafterweise ist eine an [PBE 2004a] und [ANGIE 2003] angelehnte Methodik nicht auf Entwurfsmuster beschränkt respektive spezialisiert, sondern ließe sich auch im Kontext entwurfsmusterähnlicher Strukturen wie Architekturmuster verwenden.

Durch die Trennung bei der Beschreibung von Struktur- und Verhaltensaspekten durch Verwendung unterschiedlicher Mechanismen (etwa Scriptsprache sowie Annotationen) wird eine Vermischung der beiden Aspekte, wie dies etwa bei [Baroni+ 2003b] geschieht, vermieden.

Die Verwendung einer Scriptsprache wie [ANGIE 2003] resultiert in einer Musterdokumentation, die durch den Menschen relativ leicht lesbar ist. Im Gegensatz dazu stehen Ansätze wie [Taibi+ 2003] und [Eden+ 2004], deren Spezifikation als zu formal für den Entwickler erscheint. Ziel dieser Arbeit ist es, eine möglichst stark an verfügbaren Programmiersprachen angelehnte Beschreibungssprache zu verwenden oder eine vorhandene Sprache geeignet zu erweitern.

Um die Praktikabilität des Ansatzes nicht zu gefährden, soll auf exotische Konstrukte bei der Methodik so weit wie möglich verzichtet werden. Daher sollen möglichst Standardmittel oder angepasste Standards verwendet, jedoch keine eigenen Script- oder Programmiersprachen (vgl. [Guéhéneuc+ 2001] oder [Kopi 2004]) eingeführt werden.

2.4.3.2 Musterselektion

Die Selektion von Entwurfsmustern, die auf einen gegebenen Quelltext sinnvoll anwendbar sind, soll so gestaltet sein, dass sie in ein Werkzeug integriert werden kann. So wird der Entwickler von der Last befreit, sich intensive Kenntnisse zu bestimmten Mustern aneignen zu müssen. Um zu ermitteln, welche Muster für einen gegebenen Kontext anwendbar sind, wird ein Vergleich zwischen annotiertem Quelltext und verfügbaren Musterdokumentationen durchgeführt. Positiv- und Negativkriterien (vgl. [Philippow+ 2004]) helfen bei der Auswahl. Positivkriterien können durch einen an [PBE 2004a] orientierten Ansatz quasi als Abfallprodukt anfallen, da eine beispielhafte Transformation den Vergleich von ursprünglicher und transformierter Version erlaubt und Rückschlüsse auf Gemeinsamkeiten zulässt. Negativkriterien müssen im Rahmen eines an [PBE 2004a] angelehnten Ansatzes explizit definiert werden, sofern nicht schon die definierten Positivkriterien ausreichend sind.
Es wurde kein Ansatz gefunden, der einen Abgleich zwischen im Quelltext angebrachten Semantikinformationen und für eine formale Musterdokumentation gegebene Positiv- und Negativkriterien durchführt, auch wenn dies in [Philippow+ 2004] und [Zimmer 1997] in einem anderen Kontext diskutiert wurde. Hier muss ein eigener Ansatz entwickelt werden.

Ansätze wie die der Workbench *Dali* [Kazman+ 2002] rechtfertigen insofern eine weitere Beachtung, als das eine automatisierte Analyse bestehender Dateien (Trace-, Header-Dateien etc.) bei der Gewinnung von Informationen zu einem bestehenden Quelltext hilfreich sein kann. Muster beispielsweise, die zur Performanzsteigerung eingesetzt werden können, lassen sich durch Analyse zuvor generierter Messungen als sinnvoll ermitteln (etwa *Cache Proxy* oder *Virtual Proxy*).

Da in einem vorliegenden Quelltext bereits Entwurfsmuster vorhanden sein können, ist deren Identifikation vor Selektion neu anzuwendender Muster sinnvoll. Dem kann man durch einen Ansatz wie [Philippow+ 2004] nachkommen. Im Rahmen dieser Arbeit wird die Erkennung bereits vorhandener Muster jedoch nicht über ein Maß hinaus betrachtet als dies mit Hilfe von durch ein Werkzeug im Zuge der Musteranwendung erzeugten Indikatoren möglich ist (vgl. [Mayr-Kern+ 2002], [Wöckl 2002]).
[Guéhéneuc+ 2001] hat einen ähnlichen Nutzen wie [Philippow+ 2004], wenn es um die Erkennung bereits vorhandener Strukturen geht. In [Guéhéneuc+ 2001] werden keine Muster als ganzes detektiert. Vielmehr handelt es sich um feingranularere Strukturen, die auf die Möglichkeit der

Anwendbarkeit eines Musters hindeuten. Es ist denkbar, nach Erkennung derartiger Indikationsstrukturen Annotationen automatisch vorzuschlagen. Dem wird aus Komplexitätsgründen jedoch nicht weiter nachgegangen.

Die Unterstützung des Entwicklers bei der werkzeuggestützten Musterselektion durch ein kontextuelles Hilfesystems erscheint hilfreich (siehe [Motelet 2004] und [Kazman+ 2002]). Eine Erleichterung für den Entwickler ist sicherlich die Möglichkeit, die für die Selektion eines Musters relevanten Informationen präsentiert zu bekommen. In diesem Kontext ist es denkbar, ein adaptives System aufzubauen, das vom Expertenwissen eines menschlichen Tutors profitieren könnte. Später, wenn der Ansatz sich als geeignet erwiesen hat, sollte dieses Konzept weiter untersucht werden.

Die in [Tan+ 2003] genannte Klassifikation von einem Muster anhaftenden Parametern ist zwar sehr grobgranular. Das Konzept eignet sich jedoch grundsätzlich zur Klassifikation von Parametern. So könnte ein Entwickler in einem werkzeuggestützten Prozess angeben, ein Performanzproblem lösen zu wollen. Nun könnte das Werkzeug zunächst nur diejenigen Muster für einen Vorschlag berücksichtigen, die mindestens einen Parameter der Kategorie Performanz beinhalten. Es ist zu untersuchen, ob eine Art Fragenkatalog erstellt werden kann, der den Entwickler bei der Feststellung seines Bedarfs unterstützt.

2.4.3.3 Musteranwendung

Die Musteranwendung folgt der Selektion anzuwendender Muster. Erst nach Selektion eines Musters kann das Muster im gegebenen Programmtext angewandt werden.
Mit Hilfe der analog zu [PBE 2004a] zuvor vorgenommenen formalen Musterdokumentation und dem Bekanntsein der durchzuführenden Transformationsschritte (vgl. [Zimmer 1997] und [PBE 2004a]) kann das Muster implementiert werden. Die Stellen im gegebenen Quelltext, an denen das Muster anzuwenden sind, sollten bei Musteranwendung bekannt und mit den in [Sang-Uk 2002] genannten *Candidate Spots* oder den in [Pree 1994] definierten *Hot Spots* vergleichbar sein. Die Ermittlung eines Kontextes für ein Entwurfsmuster und dessen Selektion bzw. Anwendung sind von einander abhängig. Die Klammer um diese beiden Aktivitäten wird gebildet durch die Transformationsschritte. Sie werden einerseits bei der formalen Musterdokumentation registriert. Andererseits werden sie bei der Selektion zum Abgleich und bei der Musteranwendung zur Implementierung verwendet. Transformationsschritte sind also ein zu berücksichtigendes Konzept bei der Suche nach einer Lösung für die Unterstützung bei der Musteranwendung.

Die anzuwendenden Bestandteile des Musters können vergleichbar zu [ANGIE 2003] definiert sein. Ein Interpreter kann dann die in der Scriptsprache vorliegende abstrakte Musterbeschreibung dem Kontext entsprechend implementieren. Wie auch bei anderen Aktivitäten, erscheint es für den Entwickler günstig, wenn die Musteranwendung innerhalb der Entwicklungsumgebung, die eine Workbench (vgl. [Kazman+ 2002]) repräsentiert, ausgeführt werden kann. Dementsprechend sollte eine zu entwickelnde Lösung eine Beschreibungssprache bereitstellen, die sowohl der von [ANGIE 2003] ähnelt als auch in einer herkömmlichen Entwicklungsumgebung Unterstützung findet.

Die Musteranwendung kann sich an den in [Zimmer 1997] vorhandenen Ausführungen orientieren: Ein Muster wird in generische und in anwendungsspezifische Teile zerlegt. Generische Teile können automatisiert implementiert werden, anwendungsspezifische erst nach Eingriff des Benutzers, etwa durch Variantenauswahl. Für eine Darstellung, welche Informationen bei der Musteranwendung hinzu gezogen werden können, siehe die folgende Abbildung 30.

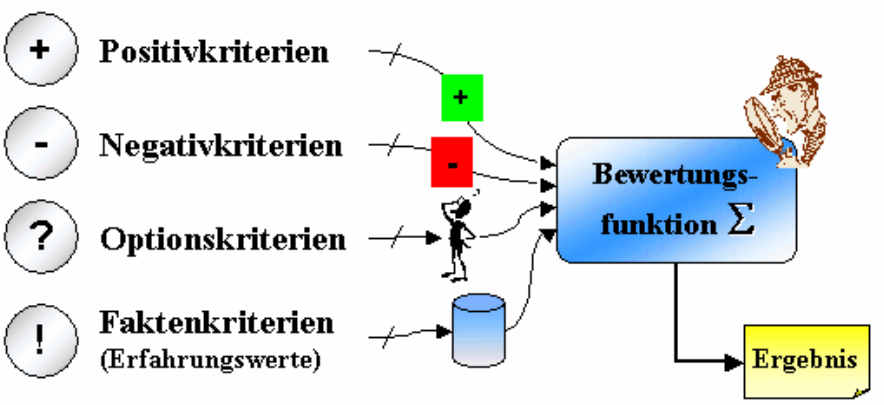

Abbildung 30: Bewertung der Anwendbarkeit eines Entwurfsmusters mittels Kriterien.

Negativkriterien, wie sie auch in [Philippow+ 2004] genannt werden, eignen sich ergänzend zu den Positivkriterien zur Verbesserung der Qualität des Abgleichs. Wie in der Abbildung 30 dargestellt, fließen zusätzlich zu den genannten Positiv- und Negativkriterien weiterhin Benutzerentscheidungen in den Abgleichprozess ein. Liegen etwa Auszeichnungen vor, die zwar ausreichend für eine prinzipielle Musterselektion, aber nicht feingranular genug sind, muss der Benutzer weitere Angaben machen. Einfaches Beispiel für eine derartige Situation sind die verwandten Muster *Abstract Factory* und *Factory Method*. Aufgrund ihrer ähnlichen Ausrichtung liegt es in der Entscheidung des Benutzers, welches der beiden Muster auszuwählen ist. Die Einbeziehung des Benutzers während der Anwendung des Verfahrens findet sich auch in [Zimmer 1997, S 70] wieder.
Faktenkriterien, also auf Erfahrungen basierte Informationen, die dem System vorliegen, können den Grad der Automatisierung des Abgleichprozesses erhöhen. Sie ermöglichen das Präsentieren von Vorschlagswerten, die der Anwender bequem übernehmen kann. Die Auswahl eines passenden Faktums geschieht durch Vergleich zwischen konkret vorliegendem Kontext (Quelltext) und gespeichertem Faktenwissen.

Es ist prinzipiell zu vermeiden, einen rein generativen Ansatz umzusetzen, wie er etwa mit [PBE 2004a] vorliegt. Ein rein generativer Ansatz basiert auf einer Vorlage, die durch ein Werkzeug einmalig umgesetzt wird. Ein flexibler generativer Ansatz hingegen erlaubt nach Umsetzung einer Vorlage deren nachträgliche Änderung, ohne dass diese Änderungen bei erneuter Generierung überschrieben werden. Die beiden genannten Typen der Generation werden in [Albin-Am+ 2001a] als „pure-generative approach“ und „conservative-generative approach“ bezeichnet.
Da ein Muster bei der Anwendung in einen Quelltext oft an mehreren Punkten des Quelltextes zu implementieren ist, ist der Ansatz mit nur einen Implementierungspunkt (vergleichbar einem *Precursor* in [Ó Cinnéide+ 1999a]) nicht ausreichend. Die Interpretation von in einem Quelltext angebrachten Annotationen hingegen erlaubt die Identifizierung aller Ansatzstellen im Quelltext für das anzuwendende Muster.

2.4.4 Integration des Ansatzes in einen Entwicklungsprozess

Für eine bessere Unterstützung des Entwicklers bei der Arbeit mit Mustern ist es sinnvoll, bereitgestellte Konzepte und Techniken als Werkzeug in der Entwicklungsumgebung (kurz: *IDE*) des Entwicklers anzubieten. Eine IDE kann durch das Konzept der Plugins funktional erweitert werden. Diese Plugins sind meist auf allgemeine Aufgaben fixiert.
Es bietet sich an, das Konzept der Workbench ([Kazman+ 2002]), das einen integrierten und auf domänenspezifische Aufgaben ausgerichteten Entwicklungsarbeitsplatz zur effizienten Entwicklung von Software mit Hilfe umfassender domänenspezifischer Funktionalität repräsentiert, zu betrachten. Die Anwendung eines Entwurfsmusters mit einem Werkzeug hat den Vorteil, angewandte Muster später zu rekonstruieren. Das ist besonders in einer Reengineering- oder auch einer Reverse Engineering-Phase nützlich (vgl. auch [Mayr-Kern+ 2002] und [Wöckl 2002]).

Zur Bereitstellung einer kontextuellen Hilfefunktion eignet sich ergänzend dazu ein Mechanismus, wie er in [Motelet 2004] und [Budinsky+ 2003] beschrieben ist.

2.4.5 Praktikabilität des Ansatzes

Der Schwierigkeitsgrad der Verwendung des Ansatzes ist stark abhängig davon, welche Arbeit der Entwickler aufwenden muss, um ein Muster auswählen und anwenden zu können. Die letztgenannten beiden Tätigkeiten wiederum hängen stark von der Güte der formalen Musterdokumentation ab, also davon, welche Aktionen aus einer Musterdokumentation heraus automatisiert werden können.
Die Ansätze [Mayr-Kern+ 2002] und [Wöckl 2002] sowie Modellierungstools wie *objectiF* setzen beim Anwender ein hohes Maß an Musterverständnis voraus. In dieser Arbeit soll versucht werden, den Entwickler, der Muster selektieren und anwenden will, so weit wie möglich zu entlasten. Vielmehr soll ein leider notwendiges, weit reichendes Musterverständnis auf den Personenkreis umgewälzt werden, der mit der grundlegenden Musterdokumentation betraut ist. Das bedeutet auch, dass eine solche Dokumentation, die optimalerweise möglichst intuitiv gewonnen werden kann, leistungsfähig und ausdrucksstark ist. Dazu muss deren Formalisierungsgrad so weit wie möglich fortgeschritten und dennoch praktikabel sein.

KAPITEL 3

3 Ein neuer Ansatz zur Unterstützung von Mustern: TrAQ

3.1 Einleitung

Dieses Kapitel stellt einen neuen Ansatz zur Unterstützung des Entwicklers bei der Arbeit mit Entwurfsmustern im Quelltext vor. Unterstützung soll insbesondere in den Aktivitäten Musterdokumentation, Musterselektion und Musteranwendung gewährt werden. Der Ansatz wird TrAQ genannt, was für Transformation durch Annotation von Quelltext steht.

Ausgangspunkt von TrAQ ist die formale Dokumentation eines Musters. Sie bildet die Grundlage für die werkzeuggestützte Musterselektion, Musteranwendung und Mustererkennung. Ziel der formalen Musterdokumentation ist eine möglichst vollständige Dokumentation des entsprechenden Musters, damit dieses werkzeuggestützt selektiert und angewandt werden kann. Dementsprechend werden die Musterselektion und die Musteranwendung im Rahmen dieser Arbeit weitergehend behandelt. Die Mustererkennung als einzelner Prozess wird im Folgenden aufgrund der Komplexität des Themas nur angerissen. Sie kann im Prinzip ähnlich wie die Aktivitäten Musterselektion und Musteranwendung als werkzeugbasierter Ansatz umgesetzt werden und ist implizit zu einem Gutteil in der später vorgeschlagenen Vorgehensweise bei der Musterselektion vorhanden, wie sich herausstellen wird.

TrAQ beinhaltet einerseits eine Reihe von Konzepten, die in dieser Kombination oder in diesem Kontext neu sind. Andererseits werden geeignete Konzepte bestehender Ansätze, die im vorigen Kapitel 2 diskutiert wurden, integriert.

Basierend auf den Ausführungen aus Abschnitt 2.4 sollen folgende Anforderungen durch den hier vorgestellten Ansatz erfüllt werden. Hauptaugenmerk wird auf die quelltextorientierte Software-Entwicklung gerichtet. Die Herausarbeitung einer Möglichkeit zur formalen Musterdokumentation ist grundlegend. Sie soll erstens geeignet sein, Entwurfs- und Architekturmuster sowie Refactoring-Operationen zu beschreiben. Zweitens soll ein versierter Software-Entwickler mit sehr guten Musterkenntnissen in der Lage sein, formale Musterdokumentationen vorzunehmen. Drittens soll die formale Musterdokumentation möglichst einfach maschinenverarbeitbar sein.
Damit soll die Selektion anwendbarer Muster für einen gegebenen Quelltext besser werkzeugunterstützt werden als bisher. Gleiches gilt für die Anwendung eines selektierten Musters.
Bei der Betrachtung vorhandener Ansätze kristallisierte sich heraus, dass Semantikinformationen im Quelltext und in der Musterdokumentation fehlen, die maschinenverarbeitbar sind. Hier muss eine Lösung gefunden werden, maschinenverarbeitbare Semantikinformationen sowohl im Quelltext als auch in der Musterdokumentation manuell bereitstellen zu können.
Die Erkennung vorhandener Muster oder Musterteile in einem gegebenen Quelltext soll zwar ebenso betrachtet werden, aber aufgrund der Komplexität des Themas nachrangig zur Musterselektion und -anwendung. Es sollen weiterhin Möglichkeiten diskutiert werden, wie der zu entwickelnde Ansatz in einen Software-Entwicklungsprozess integriert werden kann.

Dieses Kapitel ist wie folgt aufgebaut: Zuerst folgt eine Betrachtung, wie ein ideales Verfahren zur Unterstützung des Entwicklers bei der Arbeit mit Mustern im Quelltext aussehen könnte. Danach werden bereits vorgenommene, für TrAQ wichtige Definitionen erweitert und noch benötigte Definitionen neu eingeführt. Als nächstes folgt eine Skizze von TrAQ, die einen Überblick der Lösungsidee vermittelt. Anschließend wird das skizzierte Verfahren detailliert vorgestellt.

3.2 Vorbetrachtung und Definitionserweiterung

Wie könnte ein quasi ideales Verfahren zur Unterstützung des Entwicklers bei der formalen Musterdokumentation, Musterauswahl und Musteranwendung aussehen? Ein derartiges ideales Verfahren könnte folgende Aktivitäten in der angegebenen Reihenfolge beinhalten:

1. Ein Mensch wendet (neu gefundenes) Entwurfsmuster M_i auf einen dafür adäquaten Quelltext Q_a an (in dem Sinne, dass das Muster einer Lösungsidee für ein vorliegendes Problem entspricht).
2. Ein Algorithmus erkennt die Intention des betroffenen Quelltextes Q_a und des Musters M_i sowie den Kontext, in dem M_i angewandt werden kann.
3. Für einen gegebenen, beliebigen Quelltext Q_b erkennt der Algorithmus die Intention.
4. Sind die Intentionen von M_i und Q_b kompatibel (was der Algorithmus zu erkennen hat), so kann M_i analog zu Q_a automatisch auf Q_b angewandt werden.

Ein solches Verfahren kann nach aktuellem Stand nur theoretisch existieren, da die zufriedenstellende maschinelle Erkennung der Intention eines Quelltextes heutzutage nicht möglich ist (vgl. hierzu die in Kapitel 2 vorgestellten Verfahren). Zur leichteren Bewertung der Schwierigkeiten bei der Umsetzung eines solchen Verfahrens soll eine Aufstellung der positiven Datenverarbeitungseigenschaften von Mensch und Computer gegeben werden:

Mensch:

- Gilt als (hoch)intelligent.
- Kann hochkomplexe, auch unbekannte Zusammenhänge und Konzepte erkennen.
- Ist fähig, auch ohne Anleitung zu lernen und sich Erfahrungswissen anzueignen.
- Besitzt unterstützende, ausgereifte, integrierte Sensorik. Eine Folge hiervon ist: Wissensgewinn und Lerneffekt durch Interaktion mit anderen Individuen.
- Für manche Situationen: Massiv parallel rechnend/verarbeitend (vgl. [Mainzer 2003, S. 68]).
- (Re-)Konstruktion und Gewinn von Fakten aus Bruchstücken, unscharfen und auch widersprüchlichen Aussagen (weit über das von der Fuzzy-Logik erbrachte hinausgehend).
- Fehlertolerant, selbstkorrigierend.

Computer:

- Sehr schnell rechnend.
- Zuverlässig, determiniert, objektiv.
- Kapazitativ leicht erweiterbar (Speicher, Verarbeitungsgeschwindigkeit, Ausfallsicherheit).
- Anspruchslos, ausdauernd, heutzutage einfachste Energieversorgung.
- Aufgebaut aus scharf abgrenzbaren Komponenten.
- Beliebig substituierbar, dadurch quasi beliebig lange Lebensdauer.
- Präzise aufgrund digitaler Repräsentation von Daten und aufgrund der dominanten Verwendung von Faktenwissen.
- Genauigkeit bestimmter Aspekte nahezu beliebig steigerbar (Anzahl Nachkommastellen bei Berechnungen, Anzahl Durchläufe bei iterativen Verfahren etc.).

Die Vorteile des Menschen sind im Großen und Ganzen die Nachteile des Computers, und umgekehrt. Die obigen Betrachtungen erscheinen nach Meinung des Verfassers ausreichend, um abzuleiten, dass es zur Zeit keine Maschine gibt, die die in den am Anfang des Absatzes genannten Schritten 2 bis 4 des idealen Verfahrens steckende Aufgabe lösen kann. Es wurde deswegen auch davon abgesehen, in Schritt 1 den menschlichen Faktor durch maschinelle Mechanismen zu ersetzen, was den Grad der Idealisierung eines Verfahrens zur Musterselektion und -anwendung noch steigerte.
Aus der de facto Unmöglichkeit, mit vorhandenen Mitteln einen Quelltext maschinengestützt so auszuwerten, dass für diesen formal eine Musterauswahl oder -anwendung abgeleitet werden

könnte, ist die Idee zu einem neuen Verfahren zur Anreicherung von Quelltext mit Zusatzinformationen entstanden. Das Verfahren TrAQ nutzt Annotationen, um Entwurfsintentionen, Problemangaben und Anforderungen im Quelltext zu verankern. Weiterhin werden Annotationen verwendet, um die formale Musterdokumentation zu unterstützen. Letztere basiert auf der exemplarische Transformation eines für ein zu dokumentierendes Muster geeignet gewählten Ausgangs- in einen Zielquelltext, der unter Anwendung des Musters entsteht. Die Transformationen werden durch Annotationen kenntlich gemacht. Ein Grundprinzip des Verfahrens ist die Repräsentation von Lösungen, die ein Entwurfsmuster realisiert, mit synonymen oder denselben Annotationen wie für die mit den Lösungen korrespondierenden Probleme.

Es folgt eine vertiefende Beschreibung von Konzepten, die für TrAQ relevant sind. Zunächst werden in Kapitel 3.2.1 Motive, Quelltexttransformationen und Refactoring-Operationen diskutiert. Anschließend folgt in Kapitel 3.2.2 nach weitergehender Diskussion der Vorschlag einer Form von Annotationen die für den vorgestellten Ansatz geeignet erscheint. Danach werden isomorphe Programmblöcke in Kapitel 3.2.3 vorgestellt, das sind Quelltextteile, die vergleichbar zueinander sind und signifikante Bedeutung im Prozess der Quelltexttransformation erlangen. Die danach diskutierte Vergleichbarkeit von Quelltexten ist für die Prozesse Musterselektion und Musteranwendung bedeutend. Im Anschluss daran und auf Basis dessen wird das Verfahren selbst vorgestellt.

3.2.1 Motive, Transformationen und Refactoring-Operationen

Hinter jeder Quelltexttransformation steht eine Motivation für die Transformation. Es gibt keine Transformation ohne Motivation. Eine solche Motivation im Rahmen der Musteranwendung wird im Folgenden als Motiv bezeichnet. Es begründet, warum eine Transformation durchzuführen ist bzw. durchgeführt wurde. Eine Transformation wiederum wird durch Ausführen einer oder mehrerer feingranularer Refactoring-Operationen realisiert.

Die folgende Abbildung 31 zeigt den Zusammenhang zwischen den Konzepten Motiv, Transformation und Refactoring-Operation schematisch.

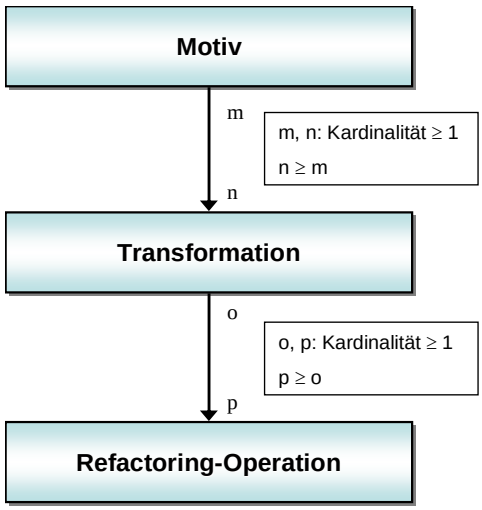

Abbildung 31: Zusammenhang zwischen Motiv, Transformation und Refactoring-Operation.

Wie in der Abbildung dargestellt, kann ein Motiv mehrere Transformationen bedingen und eine Transformation mehrere Refactoring-Operationen. Weiterhin kann dieselbe Transformation mehreren Motiven zugeordnet sein und dieselbe Refactoring-Operation mehreren Transformationen. Eine Wiederverwendbarkeit von Transformationen und Refactoring-Operationen ist also gegeben.

Motive

Ein Motiv für eine Transformation bezieht sich auf einen Quelltextteil. Ein Motiv kann technisch oder fachlich orientiert sein. Ein technisches Motiv ergibt sich aus Notwendigkeiten der Programmiersprache. Beispielsweise kann ein technisches Motiv das Einführen eines privaten Konstruktors sein. Das ist etwa sinnvoll für das *Singleton*-Muster, um eine Mehrfachinstanziierung weitgehend auszuschließen[33]. Ein fachliches Motiv hingegen ergibt sich aus einer Intention, die ein Muster in sich trägt. Ein Beispiel für ein fachliches Motiv des *Observer*-Musters ist die Benachrichtigung von registrierten Beobachtern.

Definition: Motiv für eine Transformation

> Ein Motiv für eine Transformation ist eine fachliche (funktionale) oder technische (nichtfunktionale) Motivation, die zur Transformation geführt hat. Diese Motivation ergibt sich im Kontext der Musteranwendung aus den Absichten des Musters.
>
> Ein fachliches Motiv kann technische oder andere fachliche Motive bedingen. Ein technisches Motiv hingegen kann nur andere technische Motive bedingen.

Motive sind erstens nützlich zur fachlichen Dokumentation eines Musters. Sie lassen sich in einer informellen Musterdokumentation wie [Gamma+ 1995] identifizieren und stellen idealerweise ein wertvolles Extrakt einer textuellen Musterdokumentation dar, das die Vorteile und Fähigkeiten eines Musters prägnant dokumentiert. Zweitens helfen Motive, verschiedene Quelltextteile mit einander in Beziehung zu setzen. Alle Quelltextteile, die aufgrund desselben Motivs bei der Anwendung eines Musters transformiert werden, hängen von einander ab. Diese Abhängigkeit zu kennen ist wichtig bei der Quelltexttransformation. Wird nämlich ein Quelltextteil transformiert, muss der abhängige Teil zumindest auf Transformierbarkeit hin untersucht werden. Drittens helfen Motive formal über Annotationen zu dokumentieren, was die Beweggründe für eine Transformation und somit auch für eine Refactoring-Operation waren.

Damit verschiedene Motive unterscheidbar sind, müssen sie über verschiedene Muster hinweg unterschiedlich bezeichnet sein, wenn sie verschieden sind und gleich, wenn sie gleich sind.

Ein Motiv kann andere Motive bedingen (vgl. auch [Robillard 2003, S. 69]). So ist ein fachliches Motiv des *Singleton*-Musters die Verhinderung der Mehrfachinstanziierung. Das sich daraus ergebende technische Motiv ist die Einführung eines privaten Konstruktors.
Ein technisches Motiv kann allerdings kein fachliches Motiv bedingen, denn die technische Ebene liegt unterhalb der fachlichen. Die technische Ebene "kennt" die fachliche Ebene nicht: Ein Programm kann ohne fachliche Motivation ablaufen, aber eine fachliche Motivation kann nur durch ein Programm abgebildet werden, das technisch korrekt ist.
Ein fachliches Motiv kann weitere fachliche Motive beinhalten. Letztere sind dann von feinerer Granularität als das enthaltende Motiv. Ein technisches Motiv kann anlog weitere technische Motive beinhalten. Ein fachliches Motiv kann rein logisch sein, wenn es keine eigene Implementierung besitzt, sondern diese sich aus der Umsetzung der enthaltenen Motive ergibt (vgl. Motiv *a* in Anhang B - Composite).

Aspekte eines Motivs

Ein Motiv kann sich im Besonderen auf einen ganz bestimmten Teil einer Programmentität beziehen. Beispielsweise referenziert das folgende Motiv in Abbildung 32 in erster Linie den Eingabeparameter:

[33] Mit Hilfe der Reflexion (engl.: *Reflection*) kann in Java eine Klasse beliebig instanziiert werden, ungeachtet privater Konstruktoren.

```
/**@@motiv a: Registriere Beobachter*/
public void register(Object a_observer) {
  ...
}
```

Abbildung 32: Beispiel 1 für einen Motivaspekt.

Das genannte Motiv bezieht sich auf den Eingabeparameter *a_observer*, der hier einen zu registrierenden Beobachter darstellt. Die Information, auf welche Teile einer Programmentität ein Motiv sich inhaltlich bezieht, ist insbesondere beim automatisierten Anbringen von Annotationen hilfreich. So kann das eben genannte Beispiel erweitert werden (Abbildung 33, Zeilennummern diesmal mit angegeben):

```
010 public class Subject {
020   private MyObject observer;

030   /**@@motiv a: Registriere Beobachter*/
040   public void register(MyObject a_observer) {
050     observer = a_observer;
060   }

070   public static void main(String[] args) {
080     Subject subject = new Subject();
090     MyObject observer = new MyObject();
100     subject.register(observer);
110   }
120 }
```

Abbildung 33: Beispiel 2 für einen Motivaspekt.

Im Quelltext aus Abbildung 33 sind die Zeilen 020, 040, 050 und 100 von einander wie folgt abhängig: In Zeile 100 wird Zeile 040 gerufen, in Zeile 050 wird der Eingabeparameter aus Zeile 040 der Variable zugewiesen, die in Zeile 020 deklariert wurde. Aus diesem Grund können die abhängigen Zeilen ebenfalls mit dem genannten Motiv annotiert werden.

Der in Java und verwandten Sprachen generell herrschende Zusammenhang zwischen Deklaration und Aufruf bzw. Zuweisung kann ausgebeutet werden, um aus der Annotation in Zeile 030 weitere Annotationen zu gewinnen. So kann Zeile 100 genauso wie Zeile 030 annotiert werden, weil Aufruf und Deklaration der Methode *register* ein logisch zusammen gehöriges Paar bilden. Analog können die Zeile 020 und 050 (Deklaration und Zuweisung) in Zusammenhang gesetzt werden. Zeile 050 kann allerdings nur mit Zeile 040 in Verbindung gebracht werden, wenn aus der Annotation in Zeile 030 hervorgeht, dass in Zeile 040 der Eingabeparameter den Beobachter repräsentiert.
Die Abhängigkeit zwischen verschiedenen Programmentitäten soll am folgenden Bild, das zwei Java-Klassen schematisiert, verdeutlicht werden:

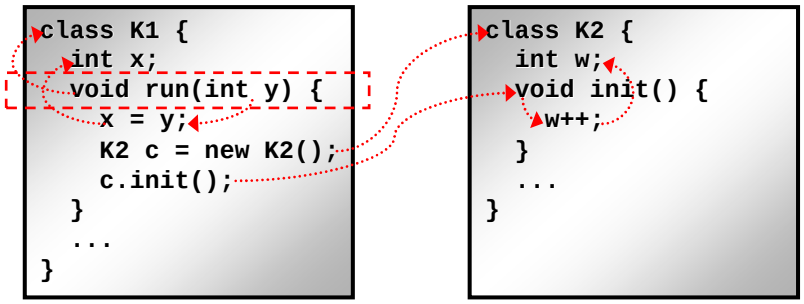

Abbildung 34: Automatisiert folgerbare Abhängigkeiten zwischen Programmentitäten.

Die in der Abbildung 34 gerahmte Programmentität, die Methode *run*, soll hier als Ausgangspunkt der Abhängigkeitsbetrachtung dienen. Dieser Fall ist äquivalent damit, dass diese Methode mit einem Motiv annotiert ist. Es gilt nun, alle von dieser Entität abhängigen Entitäten zu ermitteln, unter Berücksichtigung der Besonderheiten der verwendeten Programmiersprache. Die Abhängigkeiten sind mit roten, gestrichelten Linien und einer Richtung, ausgedrückt durch einen Pfeil, gekennzeichnet. Die Richtung gibt die Ausgangsentität an, von der die Betrachtung ausgeht und die Zielentität, die daraus ermittelt werden kann.
Die Methode *run* liegt innerhalb der Klasse K1. Also ist *run* von der Existenz von K1 abhängig. K1 andererseits ist Mitspieler bei der Realisierung des Motivs, das *run* zugeordnet ist. Der Eingabeparameter der Methode *run* hat den Namen *y*. Dieser wird in der folgenden Zeile der Klassenvariablen *x* zugewiesen. Also ist diese Zuweisung korreliert und somit im Sinne der Betrachtung zum Motiv abhängig. Daraus folgt eine Abhängigkeit dieser Zuweisung zur Deklaration von *x* auf Klassenebene. Innerhalb von *run* wird eine Instanz von Klasse K2 erzeugt. Somit besteht eine Abhängigkeit zwischen Klasse K1, in der *run* deklariert ist, und K2. Ferner wird Methode *init* der generierten Klasseninstanz gerufen (woraus ebenfalls eine Abhängigkeit zu K2 folgt). Innerhalb dieser Methode wird die Klassenvariable *w* inkrementiert, womit die Abhängigkeit zur Deklaration von *w* gegeben ist.
All diese Abhängigkeiten können automatisiert gefunden werden. Dazu ist lediglich eine Analyse der abstrakten Syntaxbäume der beiden Klassen erforderlich.

These: Ermittlung abhängiger Programmentitäten zu einem Motiv

---

Ist eine Programmentität mit einem Motiv versehen, können in vielen Fällen davon abhängige Programmentitäten ermittelt werden. Das erlaubt es, automatisiert Annotationen anzubringen.

Insbesondere kann eine Abhängigkeit bei Variablenzuweisungen, Aufrufen von Methoden aus fremden Klassen, Anweisungen innerhalb von annotierten Methoden und Klassenreferenzen durch Typdeklarationen automatisiert festgestellt werden.

Zurück zu den Motivaspekten und dem Quelltext aus Abbildung 33: Die Kennzeichnung des Teils der Programmentität in Zeile 040, die den Beobachter für das Motiv »Registriere Beobachter« darstellt, kann beispielsweise durch eine Zuordnung eines eindeutigen Begriffs mit einer Suchroutine abgebildet werden. Als eindeutiger Begriff für den Aspekt des Beobachters im Motiv wird »Beobachter« gewählt, damit ein menschlicher Leser dessen Bedeutung direkt erfassen kann. Die zugehörige Suchroutine wird nun so gestaltet, dass der erste Eingabeparameter im abstrakten Syntaxbaum einer Methodendeklaration erkannt wird. Die Routine sollte allerdings so gestaltet sein, dass komplexere Fälle ebenfalls korrekt behandelt werden können.
So könnte es mehrere Eingabeparameter für eine Methode geben, wovon nur derjenige den Beobachter darstellt, der einen bestimmten Typ besitzt oder von einem bestimmten Typ abgeleitet

ist. Andererseits muss ein Beobachter immer ein nichtprimitiver Typ sein. So kann eine Suchroutine auch ein Beobachterobjekt identifizieren, wenn es mit dem generischen Java-Typ *Object* angegeben ist und ansonsten nur primitive Typen in der Methodensignatur auftauchen.
Damit mehrere solche Zuweisungen für ein Motiv möglich sind, kann eine Tabelle von Tupeln (Begriff, Suchroutine) geführt werden. Fügt man in die Tabelle noch das Motiv als Schlüsselfeld ein, reicht eine Tabelle für alle Motive als Kriterium aus.

Um die Bedeutung eines Aspekts (etwa »Beobachter«) zu gestalten, ist eine konkrete Behandlerroutine erforderlich. Damit können automatisiert Annotationen angebracht werden, so wie eben beschrieben. Diese Routine besitzt pro definiertem Aspekt eine konkrete Logik, wie dieser Aspekt zu interpretieren ist. Andererseits können bestimmte Zusammenhänge zwischen Programmentitäten nur durch ein Motiv alleine, ohne weitere Berücksichtigung von deren Aspekten, hergestellt werden. Dies ist im obigen Beispiel für die Zeilenpaare (040, 100) sowie (020, 050) der Fall. Um dem Rechnung zu tragen, sollte die eben genannte Behandlerroutine neben dem Aspekt auch das Motiv bekannt gemacht werden. Dies auch aus dem Grund, weil ein Aspekt in zwei verschiedenen Motiven eine unterschiedliche Behandlung bedingen kann. Das Motiv ist hierbei eine Mussangabe, wohingegen der Aspekt eine optionale Komponente darstellt. Wird der Aspekt nicht mit übergeben, so führt dies zu einem positiven Ergebnis, wenn er irrelevant ist, andernfalls bricht die Behandlerroutine mit einer Fehlermeldung ab oder fragt beim Anwender nach einer Entscheidung.

Musterteile

Teile eines Musters können durch Motive identifiziert werden. Ein Musterteil wiederum kann in unterschiedlicher Gestalt umgesetzt sein. Jeder Variationspunkt im Muster kann bei einem werkzeuggestützten Selektionsprozess, der vor der Musteranwendung steht, zu einer dem Entwickler zu präsentierenden Auswahlmöglichkeit führen. Variationspunkte können als variable Musterteile im Rahmen des Verfahrens verstanden werden. Besteht ein Muster aus mehreren Variationspunkten, kann es durch je ein Musterteil pro Variationspunkt repräsentiert werden.

Definition: Musterteil

> Als Musterteil werden die Teile eines Musters verstanden, die zusammen genommen ein Motiv umsetzen. Pro Motiv ergibt sich ein Musterteil. Variationsmöglichkeiten eines Musterteils ergeben sich aus den bekannten und als sinnvoll erachteten Implementierungsmöglichkeiten pro Motiv.

Pro Musterteil kann eine Annotation definiert werden. Sie korrespondiert mit je einem Motiv des Musters, entsprechend der obigen Aufzählung der Motive.

Musterteile können als statisch oder dynamisch angesehen werden. Statische Musterteile sind kontextunabhängige Teile, etwa die Schnittstelle für Beobachter im *Observer*-Muster oder das Hinzufügen einer Schnittstellen zu einer Klassensignatur als Markierung, dass die Klasse die jeweilige Schnittstelle umsetzt. Statische Musterteile sind bei der Musteranwendung unverändert zu übernehmen. Einzig die Benennung von Methoden, Klassen oder Variablen kann variiert werden. Dynamische Musterteile sind alle Teile, die nicht statisch sind.
Welcher Musterteil statisch ist und welcher nicht, kann teilweise über Refactoring-Operationen ermittelt werden. Operationen, die etwas Löschen betreffen keine statischen Musterteile, da der gelöschte Teil kein Bestandteil des Musters ist. Operationen, die etwas Hinzufügen, können potentiell statische Musterteile betreffen. Dies trifft in besonderem Maße auf Deklarationen von Variablen oder auf Schnittstellen zu. Denn gerade diese beiden Programmentitäten sind nicht variabel bzgl. einer Musterausprägung. Werden diese Entitäten in Variationen desselben Musters anders implementiert, sind die in der variierten Form ausgeprägten Entitäten statisch bzgl. der Mustervariante.

Der Nutzen über die Kenntnis, ob ein Musterteil statisch ist oder nicht, liegt darin, dass Mustervariationen hierdurch leichter von dediziert anderen Mustern unterschieden werden können. Ist nämlich ein als statisch angesehener Musterteil in einer potentiell als Mustervariation einzustufenden Ausprägung qualitativ anders implementiert, kann davon ausgegangen werden, dass keine Mustervariation vorliegt, sondern stattdessen ein anderes Muster. Weiterhin können statische Musterteile in einer Musterdokumentation, wie sie etwa durch [Gamma+ 1995] populär wurde, anders dargestellt werden. Diese Darstellung kann sich sowohl auf UML-Diagramme als auch auf textuelle Beschreibungen oder auf Kommentare in Quelltextbeispielen beziehen.

Transformationen

Eine Transformation ergibt sich aus einem Motiv. Sie wird manuell ausgeführt und ist vergleichbar einem Minipattern (vgl. [Ó Cinnéide+ 1999a]). Eine Transformation T überführt einen Quelltext Q_a in einen Quelltext Q_z. Die kurze Schreibweise hierfür ist $T(Q_a) \rightarrow Q_z$.
Eine Transformation T kann Transformationen T_i enthalten, in diesem Fall definiert sich T ausschließlich über alle in ihr enthaltenen T_i. Diese T_i liegen dann in einer fest definierten Reihenfolge vor, die innerhalb von T unverändert bleibt. Eine Transformation T_i kann elementar sein. Elementar ist eine Transformation, wenn sie nicht durch andere Transformationen T_{ij} definiert wird.

Besteht T nicht aus mehreren T_i, so ist T elementar. Eine Transformation T_i kann ebenfalls aus mehreren anderen Transformationen T_{ij} bestehen, sie muss es aber nicht.

Eine elementare Transformation T_i kann Teil einer übergeordneten Transformation T sein und dann innerhalb T ausgeführt werden. Sie kann aber auch für sich alleine ausgeführt werden. Ist T elementar, wird T direkt durch Aufruf der Transformationslogik ausgeführt. Ist T nicht elementar, sondern besteht aus n Teiltransformationen T_i (i = 0..n-1), so wird T gemäß folgendem Pseudocode ausgeführt (Q_{temp} ist ein temporär gehaltener Quelltext):

```
Qtemp = Qa
Für i = 0 bis n-1
  Ti(Qtemp) → Qz
  Qtemp = Qz
```

Abbildung 35: Pseudocode für die Ausführung komponierter Transformationen.

Jedes T_i, das nicht elementar ist, wird auf analoge Weise ausgeführt. Dies entspricht dem Prinzip des Musters *Composite* [Gamma+ 1995], das eine Rekursion abbildet.

Ist eine Transformation nichtelementar, hängt sie von den in ihr enthaltenen Transformationen ab, die also vor ihr auszuführen sind. Ist eine Transformation nicht auf diese Weise von einer anderen abhängig, dann ist diese Abhängigkeit in einer Abhängigkeitstabelle zu verzeichnen. Beispielsweise kann eine Anweisung durch Transformation erst in eine Klasse verschoben werden, wenn diese Klasse durch eine andere Transformation neu erzeugt wurde.
Eine Abhängigkeitstabelle enthält pro Mustervariante in einer Musterdokumentation (Abschnitt 3.3) eine Zuordnung einer Transformation zu einer davon abhängigen Transformation. Bei der Ausführung von Transformationen kann deren Reihenfolge dann mit Hilfe der Abhängigkeitstabelle bestimmt werden. Abhängigkeiten wie etwa die einer Transformation, die eine Methode anlegt, von der Klasse, in die die Methode gehört, können auch automatisiert ermittelt werden, da es sich dabei um allgemein gültige Grundsätze der Programmiersprache Java (und anderer) handeln.

Eine Transformation wird mit Hilfe von Annotationen im Ausgangs- und Zielquelltext definiert. Korrespondierende Annotationen im Ausgangs- und Zielquelltext markieren den Zustand vor und nach einer modifizierenden Transformation eindeutig. Existiert für eine Annotation im Ausgangsquelltext keine korrespondierende Annotation im Zielquelltext, handelt es sich um eine

Löschoperation. Ist umgekehrt für eine Annotation im Zielquelltext keine korrespondierende Annotation im Ausgangsquelltext vorhanden, handelt es sich um eine Einfügeoperation.

Mögliche Transformationsoperationen für Java-Quelltext

Die Ausführung einer Transformation kann durch die Anwendung einer Operation ausgedrückt werden. Die möglichen Transformationsoperationen werden anhand der in Java vorhandenen Entitäten ermittelt. In der Programmiersprache Java gibt es folgende Entitäten:

- Das Programm als Ganzes
- Paket (logische Hülle um eine Menge von Klassen)
- Klasse bzw. Schnittstelle
- Methode
- Klassendeklaration
- Lokale Deklaration
- Anweisung
- Block von Klassendeklarationen
- Außerhalb einer Methode: Block von Anweisungen, inklusive lokalen Deklarationen
- Innerhalb einer Methode: Block von Anweisungen, inklusive lokalen Deklarationen
- Innere Klassen
- Labels: hier nicht weiter berücksichtigt

Denkbare Transformationsoperationen für Quelltextteile sind:

- Einfügen
- Löschen
- Verändern
- Umbenennen
- Kopieren
- Verschieben
- Ersetzen

Die Operation Kopieren ist ein Spezialfall des Einfügens. Verschieben und Ersetzen können als Kombination aus Löschen und Einfügen abgebildet werden. Umbenennen ist ein Spezialfall des Veränderns, Ersetzen kann auch als solcher aufgefasst werden.

Pro Entität (Programm, Klasse, Anweisung etc.) gibt es unterschiedliche zulässige Transformationsoperationen, die aus den Gegebenheiten der Sprache Java und den an sich möglichen Operationen resultieren. Die vom Autor ermittelten Kombinationsmöglichkeiten sind in folgender Tabelle 9 dargestellt.

Entität	Operation	Bemerkungen
Paket	Umbenennung	
Klasse	Neueinführung	
	Löschung	Entfernt oder ersetzt durch andere Klasse
	Modifikation	Äquivalentes Verhalten vs. Logikänderung
	Ersetzung	
	Umbenennung	
	Verschiebung	in anderes Paket
Methode	Neueinführung	
	Löschung	
	Ersetzung	
	Modifikation	Äquivalentes Verhalten vs. Logikänderung
	Verschiebung	In andere Klasse
	Umbenennung	
Deklaration / Dekl.block	Neueinführung	
	Löschung	
	Ersetzung	
	Modifikation	etwa durch Typänderung
	Verschiebung	In andere Klasse, Methode oder statischen Block
	Umbenennung	
Anweisung / Anw.block	Neueinführung	
	Löschung	
	Ersetzung	
	Modifikation	Äquivalentes Verhalten vs. Logikänderung
	Verschiebung	In andere Methode, Klasse oder statischen Block

Tabelle 9: Mögliche Transformationsoperationen für Java-Entitäten.

Für ein Programm als Ganzes gibt es keine Transformationsoperation, da es sich sonst um ein völlig neues Programm handelte. Für Pakete gibt es nur eine Transformationsoperation, die Umbenennung. Das Neueinführen sowie Löschen von Paket geschieht nicht explizit, sondern implizit über das Neueinführen von Klassen in ein Paket sowie das Entfernen aller Klassen aus einem Paket. Eine Anweisung oder ein Block kann hingegen nicht umbenannt werden, weil Anweisung und Block nicht mit einem symbolischen Namen verknüpft sind.

Refactoring-Operationen

Eine Transformation ist eine manuell durchgeführte Operation auf einem Quelltext. Sie kann aus mehreren Schritten bestehen. Um Transformationen in eine maschinenverarbeitbare Form zu bringen, werden sie in Refactoring-Operationen überführt. Eine Refactoring-Operation ist eine feingranulare Änderung auf Quelltextebene, die prinzipiell maschinell durchgeführt werden kann. Ergebnis der Operation ist ein kompilierbarer Quelltext.

Refactoring-Operationen ergeben sich aus manuell durchgeführten Transformationen. Es liegt in der Hand einer kompetenten Person, zu einer Transformation alle Refactoring-Operationen zu ermitteln, die diese Transformation realisieren. Die Maschinenausführbarkeit von Refactoring-Operationen und die Anforderung der Wiederverwendbarkeit führen zu einer weiteren Forderung für die Gestalt von Refactoring-Operationen. Sie lautet, dass eine Refactoring-Operation möglichst feingranular sein soll.

Um Refactoring-Operationen für zukünftige Verwendungen definieren zu können, müssen folgende Informationen pro Refactoring-Operation festgehalten werden:

- Name der Operation: Beliebig wählbar, informeller Text für den menschlichen Leser

- Eingabeparameter: sämtliche benötigten Informationen zur Durchführung der Operation. Dies beinhaltet etwa den Namen einer Klasse, wenn diese durch die Refactoring-Operation neu anzulegen ist.
- Vorbedingungen: Angabe von Vorbedingungen, die vor Anwendung der Refactoring-Operation gelten müssen. Die Angabe kann in gleicher Form erfolgen wie für Geltungsbereiche in Annotationstypen (vgl. Abschnitt 3.2.2).
- Operationslogik: Referenz auf eine Programmlogik, die die Refactoring-Operation mit Hilfe der bereitgestellten Parameter ausführen kann.

In Abbildung 36 ist beispielhaft die Definition einer Refactoring-Operation angegeben:

```
Name: Introduce_Interface @interface
Eingabeparameter:
   @interface: Einzuführende Schnittstelle
Vorbedingungen:
   Schnittstelle @interface noch nicht vorhanden
Ausgangsannotation: fehlt
Zielannotation: "motiv a"
Operation: Schnittstelle anlegen
```

Abbildung 36: Beispielhafte Definition einer Refactoring-Operation.

Die Definition beinhaltet die Abschnitte *Name*, *Eingabeparameter*, *Vorbedingungen*, *Ausgangsannotation*, *Zielannotation* und *Operation*. Alle bis auf die Annotationen betreffenden Abschnitte sind kontextfrei. Sie sind elementarer Bestandteil der Refactoring-Operation. Die Annotationen betreffende Abschnitte sind abhängig von einem konkreten Bezug, nämlich den dort genannten Annotationen. Somit repräsentiert Abbildung 36 eine kombinierte Darstellung einer allgemeinen und einer musterabhängigen Definition. Diese verkürzte Form der Darstellung wird im Folgenden weiter verwendet. Zum Verständnis sei gesagt, dass sie aufgelöst werden könnte, indem alle bis auf die Annotationen betreffenden Bereiche aus Abbildung 36 als allgemeine, kontextfreie Definition einer Refactoring-Operation angesehen werden und die beiden Annotationsabschnitte (Ausgangs- und Zielannotation) plus eine Referenz auf die Refactoring-Operation mittels deren Namen (in Abbildung 36 ist das *Introduce_Interface*) als Zuordnung einer Refactoring-Operation zu einem Kontext.

Eingabeparameter von Refactoring-Operationen werden mit einem vorangestellten Klammeraffe »@« markiert, in der Abbildung 36 ist dies etwa der Parameter mit dem Namen *interface* zur Eigenschaft *Name*. In der Abbildung 36 wurden die Vorbedingungen lediglich in textueller Form angegeben. Letztendlich sollte jede Vorbedingung nach Möglichkeit formalisiert werden durch Angabe einer Prüfroutine (vgl. [Ó Cinnéide+ 1999b]). Die Angaben zu Ausgangs- und Zielannotation in Abbildung 36 geben an, welcher Teil des Ausgangsquelltextes im Rahmen der Musterdokumentation in welchen Teil des Zielquelltextes überführt wurde. Der jeweilige Teil wird durch eine Annotation eindeutig bestimmt: Eine Annotation bezieht sich auf die ihr direkt folgende Programmentität. Die angegebene Zielannotation *motiv a* wurde der Übersichtlichkeit wegen in Kurzform angegeben. Eine vollständige Angabe enthielte von Maschinen und von Menschen verarbeitbare Teile, so wie in Abschnitt 3.2.2 dargestellt. Ausgangs- und Zielannotation korrespondieren letztendlich mit Motiven (vgl. Abschnitt 3.3.3.3).

Ein Beispiel für den Zusammenhang der Begriffe

Anhand der Motive »Registrierung zu beliebigem Zeitpunkt« und »einheitlicher Registrierungsmechanismus« soll demonstriert werden, wie Annotationen, Motive, Transformationen und Refactoring-Operationen zusammen hängen.
Jedes Motiv wird im Quelltext durch eine Annotation angebracht. Die Position der Annotation bestimmt sich durch die Anweisungen, die dem Motiv zuzuordnen sind. Jede Anweisung, deren

Existenz durch das Motiv motiviert werden kann, wird mit einer Annotation versehen, die mit dem Motiv korrespondiert.
Aus dem Motiv »Registrierung zu beliebigem Zeitpunkt« ergibt sich, dass ein zu registrierendes Objekt nicht im Konstruktor eines registrierenden Objekts übergeben werden darf. Eine zweite, sich aus dem Motiv ergebende Schlussfolgerung ist, dass die Registrierung über eine Methode erfolgen muss, denn nur so kann eine Registrierung zu einem beliebigen Zeitpunkt stattfinden.
Das Motiv »einheitlicher Registrierungsmechanismus« bedingt, dass das zu registrierende Objekt einer Schnittstelle (oder auch einer abstrakten Klasse) gehorchen muss. Eine weitere Schlussfolgerung lautet konsequenterweise, dass der Eingabeparameter im Registraturmechanismus für registrierende Objekte vom Typ der einheitlichen Schnittstelle sein muss.
Die beiden eben genannten Motive zusammengenommen, ergeben die Notwendigkeit einer Registraturmethode, die als Eingabe einen Schnittstellentyp fordert.

Aus der Schlussfolgerung einer wie eben bezeichneten Registraturmethode ergeben sich je nach Ausgangsquelltext unterschiedliche Transformationen. Im einfachsten Fall ist eine Registraturmethode im Ausgangsquelltext exakt so vorhanden wie sie aus einem Motiv hervorgeht. Ist eine Registraturmethode zwar vorhanden, aber bekommt sie das zu registrierende Objekt in Form eines konkreten Klassentypen übergeben, so ergeben sich mehrere notwendige Transformationen. Diese Transformationen bestehen aus der Deklaration einer Schnittstelle, der Implementierung der Schnittstelle beim zu registrierenden Objekt sowie der Anpassung der Signatur der Methodenregistratur. Wird im Ausgangsquelltext eine Registrierung hingegen über einen Konstruktor vorgenommen, muss die Registraturmethode eingeführt, an geeigneter Stelle aufgerufen und der Konstruktor sowie dessen Aufruf angepasst werden.

Für eine Transformation hängen die durchzuführenden Refactoring-Operationen von der Form des Ausgangsquelltextes ab. Die Transformation »Implementierung der Schnittstelle« etwa bedingt für die Erweiterung der Klassensignatur eine andere Refactoring-Operation für eine Klasse, die bereits eine Schnittstelle implementiert als für eine Klasse, die noch keine Schnittstelle implementiert. Die nachfolgenden zwei Abbildungen stellen diese beiden Fälle beispielhaft dar:

```
public class C1 {                    public class C1 implements I1 {
...                              →   ...
}                                    }
```

Abbildung 37: Einführung einer Schnittstelle in eine Klasse (Fall eins).

Die Abbildung 37 zeigt auf der linken Seite die Ausgangssituation, eine Klasse C1 ohne Schnittstellendeklaration. Auf der rechten Seite der Abbildung ist das durch Refactoring-Operation zu gewinnende Ergebnis zu sehen, die Implementierung der Schnittstelle I1. In der folgenden Abbildung 38 ist die Ausgangssituation eine andere:

```
public class C1 implements I2 {      public class C1 implements I1, I2 {
...                              →    ...
}                                     }
```

Abbildung 38: Einführung einer Schnittstelle in eine Klasse (Fall zwei).

Hier implementiert die Klasse C1 bereits die Schnittstelle I2. Es ist ersichtlich, dass eine Refactoring-Operation zur Implementierung von Schnittstelle I1 anders arbeiten muss als im vorigen Fall. Dies ist alleine schon daran zu sehen, dass das Schlüsselwort *implements* bereits im Ausgangsquelltext enthalten ist. Um geänderten Ausgangssituationen wie diesen Rechnung zu tragen, wurden unterhalb von Transformationen die Refactoring-Operationen als konzeptionelle Ebene hinzugefügt.

3.2.2 Weitergehende Diskussion von Annotationen

Der Begriff der Annotation wurde in Abschnitt 2.3.5 einleitend definiert. An dieser Stelle soll der Begriff weiter präzisiert werden. Annotationen stellen im Kontext dieser Arbeit Semantikinformationen dar. Sie werden im Quelltext angebracht und beziehen sich auf Programmentitäten. Programmentitäten sind fundamentale Bestandteile von Quelltexten und werden hier wie folgt definiert:

Definition: Programmentität

> Eine Programmentität ist ein valides Konstrukt der einem Programm zugrunde liegenden Programmiersprache, entweder eine Deklaration oder eine Anweisung. Elementar wird eine Programmentität genannt, wenn sie entweder eine Deklaration darstellt oder eine Anweisung, die keine weiteren Anweisungen oder Deklarationen enthält. Nichtelementar sind beispielsweise Blöcke oder Schleifenanweisungen.
>
> Synonym wird der Begriff »Programmelement« verwendet. Eine Programmzeile mit genau einer Anweisung oder Deklaration entspricht ebenfalls einer Programmentität.

Programmentitäten können unterschiedliche Granularitäten haben. Die Granularität einer Programmentität drückt die Abstraktionsebene einer Programmentität aus. Die Granularität ist für atomare Entitäten wie für eine einzelne Anweisung oder Felddeklaration am feinsten. Eine nichtatomare Entität hingegen – beispielsweise eine Methode oder eine Klasse – hat eine geringere Granularität, weil sie feingliedrigere Entitäten enthalten kann. Granularität ermöglicht die Reduktion der Komplexität eines Annotationsverfahrens, weil im Bedarfsfall eben nur eine grobgranulare Entität annotiert wird, wenn der Restkontext vernachlässigt werden kann. Die folgende Abbildung stellt die Granularität unterschiedlicher Programmentitäten dar.

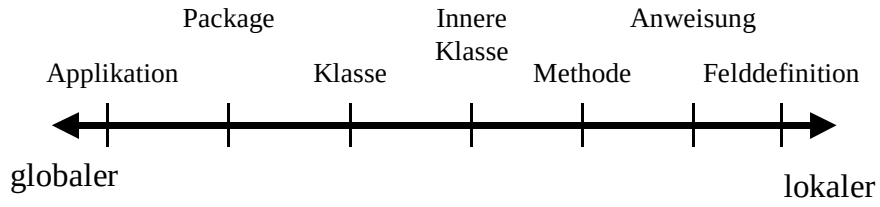

Abbildung 39: Granularität von Programmentitäten.

Eine Annotation ordnet einer Programmentität eine Bedeutung zu. Eine Bedeutung kann entweder technischer oder fachlicher Natur sein. Beides kann unter dem Begriff *Semantik* zusammengefasst werden. Annotationen werden gewöhnlich manuell durch den Entwickler im Programmtext angebracht. Mit Hilfe von zusätzlich zu entwickelnden Werkzeugen können manuell angebrachte Annotationen durch automatisch generierte ergänzt werden.

Ein sehr einfaches Beispiel für die Auszeichnung einer Klasse mit Hilfe einer Annotation im Javadoc-Format sei zur Veranschaulichung in Abbildung 40 genannt:

```
/**@@motiv instances 1*/
public class ApplicationManager {
...
  public void doSomething() {
  ...
  }
}
```

Abbildung 40: Semantische Auszeichnung einer Klasse.

In der ersten Zeile ist die Annotation in Form eines bedeutungsrelevanten Kommentars angebracht, der einer zu definierenden Grammatik gehorcht und Bestandteil eines festgelegten Wortschatzes ist. Die Bedeutung o.g. Annotation */**@@motiv instances 1*/* könnte für die Klasse *ApplicationManager* sein, dass die Klasse nur maximal eine Instanz besitzt, gleich eines *Singletons*. Das Schlüsselwort *motiv* nach dem doppelten Klammeraffen bezeichnet, dass die Annotation ein Motiv ausdrückt.

3.2.2.1 Einfache Form von Annotationen

Ein Javadoc-Kommentar wie er in Abbildung 40 verwendet wurde, hat eine einfache Form. Diese Form wird mit leichter Variation im späteren Abschnitt 3.3 zur Illustration von TrAQ verwendet, damit die Ausführungen prägnanter und somit verständlicher gehalten werden können. Dies ist deshalb unkritisch, weil die Beispiele nicht maschinell verarbeitet werden sollen, sondern lediglich zur Illustration für den Leser gedacht sind und weil die einfache Form leicht in eine komplexere Form überführt werden kann, wie später gezeigt wird.

Zwei Änderungen am Format aus Abbildung 40 werden vorgenommen, um für das Verfahren notwendige Informationen aufnehmen zu können. Die endgültige einfache Form von Annotationen lautet wie folgt (Abbildung 41):

```
/**@@motiv <motiv>(<index>/<unterindex>): <freitext> */
```

Abbildung 41: Verkürzte Form für Annotationen.

Die in spitzen Klammern stehenden Teile der Abbildung 41 stellen Variablen dar. Da jede Annotation genau ein Motiv ausdrückt, wird in der Annotation das Kürzel des Motivs (im Rahmen dieser Arbeit ein Kleinbuchstabe) untergebracht, und zwar an der mit *<motiv>* bezeichneten Stelle in der Abbildung. Danach folgt ein Index, der pro Motiv geführt wird und an der mit *<index>* gegebenen Stelle einzusetzen ist. Der Index einer Annotation ist eine bei eins beginnende und fortlaufende Zahl. Er dient dazu, einen Vergleich zwischen zwei Phasen des Verfahrens (etwa Phase A und Phase B, vgl. Abschnitt 3.3.4.3) zu ermöglichen. Ist etwa eine Programmentität in Phase A annotiert, wird sie nur mit einer solchen Programmentität in Phase B verglichen, die eine Annotation zum gleichen Motiv mit demselben Index besitzt.
Ein optionaler Unterindex wird durch einen Schrägstrich (»Slash«) vom Index getrennt. Er erlaubt es, mehrere zum selben Motiv gehörende Anweisungen zu annotieren.
Ein ausschließlich als informale Information für den menschlichen Leser gedachter Freitext wird für den Platzhalter *<freitext>* eingesetzt. So kann eine verkürzte Annotation beispielsweise lauten:

```
/**@@motiv a(1): Behandle Beobachterklassen einheitlich*/
```

Abbildung 42: Beispiel einer Annotation in verkürzter Form.

Wichtig ist in dieser Arbeit, dass verschiedene Annotationen maschinell von einander unterscheidbar sind. Das bedeutet, verschiedene Annotationen dürfen nicht gleich benannt sein und müssen einer einheitlichen Form folgen.

3.2.2.2 Komplexe Form von Annotationen

Die eben genannte einfache Form von Annotationen ist nicht geeignet, um alle im Folgenden diskutierten Anforderungen zur Maschinenverarbeitbarkeit und Erweiterbarkeit zu erfüllen. Daher wird im Weiteren eine leistungsfähigere Form vorgeschlagen. Die nun genannten Anforderungen an Annotationen sollen bewirken, dass Annotationen von einer Machart sind, die deren Anbringung, Verarbeitung und Interpretation mit Hilfe von Werkzeugen einfach gestattet.

3.2.2.3 Präzision und Signifikanz von Annotationen

„As complexity rises, precise statements lose meaning and meaningful statements lose precision."

Lotfi Zadeh, Begründer der Fuzzy-Logik

Es gilt, Annotationen so präzise wie möglich zu halten, um die Aussagekraft gegenüber dem Entwickler aber auch gegenüber der Maschine größtmöglich zu gestalten. Die Aussagekraft einer Annotation hängt nicht nur von der Granularität der Programmentität ab, die mit der Annotation versehen ist, sondern auch von der Granularität der Annotation selbst. Die Granularität einer Programmentität ergibt sich aus dem Typ der Entität. Beispielsweise ist eine Methode feingranularer als eine Klasse aber grobgranularer als eine Anweisung innerhalb der Methode. Die Granularität einer Annotation drückt den Detaillierungsgrad des Bezuges zum Bezugselement aus. Der feingranularste Bezug findet sich etwa bei einer Annotation, die sich nur auf einen atomaren Teil der Anweisung bezieht. Zur Veranschaulichung ist folgende Anweisung gegeben:

```
int x = 2;
```

Abbildung 43: Beispiel für die Veranschaulichung der Granularität einer Annotation.

Diese einzelne Anweisung, die eine Deklaration und einer Zuweisung enthält, besteht aus vier atomaren Teilen, nämlich:

- Typisierungsdeklaration: *int*
- Variablenbenennung: *x*
- Zuweisungsoperator: *=*
- Zuweisungswert: *2*

Als Impuls für weitere Ausführungen soll ein Satz von Matisse dienen.

„Precision is not truth."

Henri Matisse

Die Wirklichkeit auszudrücken, ist ein nicht erreichbares Ziel. Denn sie existiert nicht in einer eindeutigen Form und hängt zumindest von der Art der Beobachtung ab (vgl. etwa die Quantenphysik und die Heisenberg'sche Unschärferelation). Wieder ist man auch bei Annotationen darauf beschränkt, eine sinnvolle Implementierung zu wählen. Im Falle von Quelltext, der einer wohldefinierten Syntax genügt, lassen sich jedenfalls die Bestandteile eines Quelltextes sowie deren Deklarations- und, Anweisungsarten problemlos und eindeutig auseinandersetzen. Eine Programmentität als Teil eines Quelltextes lässt sich beliebig genau in ihre Bestandteile auflösen. Es ist beliebig präzise möglich, auf eine Programmentität oder jeden sie ausmachenden Teil Bezug zu nehmen. Dies geht derart, dass eine beliebig kleine Sequenz von Zeichen gewählt wird, die noch eine Bedeutung trägt. Im Falle des Gleichheitsoperators sind dies zwei Zeichen in Java (»==«), der Negationsoperator besteht aus einem Zeichen (»!«).

Allerdings ist es nicht von vornherein möglich, eine semantische Aussage beliebig präzise zu formulieren. Der Grund liegt darin, dass eine Aussage an sich eine beliebige Gestalt annehmen kann, für die man zwar eine (präzise) Syntax fordern kann. Jedoch ist die Interpretation einer wohl formulierten Aussage ein willkürlicher Akt, der auf vorigen Definitionen beruht. Diese Definitionen sind nicht präziser als das, was der Mensch ihnen mitgibt.

Nicht weniger wichtig als die Präzision einer Annotation ist deren Signifikanz. Eine Annotation ist signifikant im Sinne dieser Arbeit, wenn sie Informationen bereitstellt, die beim Abgleichprozess zwischen einem gegebenen Quelltext und einer Musterdokumentation als Entscheidungskriterium für oder gegen ein bestimmtes Entwurfsmuster nützlich sind.

Einem erfahrenen Entwickler sollte es möglich sein, die Charakteristik eines Musters ausreichend signifikant mit Annotationen auszudrücken. Dies sollte alleine deshalb möglich sein, weil eine informale Musterdokumentation alle relevanten Teile eines Musters enthält und die Kernaussagen zum Muster stichpunktartig benannt werden können. Jeder Stichpunkt kann als Annotation ausgedrückt werden. Vorbedingung ist die Verfügbarkeit eines geeigneten Satzes von Aussagen, die mit Annotationen ausgedrückt werden können. Andererseits sollte es möglich sein, benötigte aber noch fehlende Aussagen hinzuzufügen. Letzteres ist insbesondere wichtig, weil es unrealistisch erscheint, alle möglichen relevanten Aussagen zu Mustern in einem Schritt erfassen zu können. Signifikanz kann also nicht alleine durch die Syntax einer Annotation erreicht werden. Vielmehr kann Signifikanz nur durch einen geeigneten Inhalt einer Annotation erlangt werden.

3.2.2.4 Kennzeichnung und Notation von Annotationen

Zur einfachen Unterscheidung eines gewöhnlichen Kommentars von einer Annotation wird ein spezielles Präfix eingeführt. Das Präfix wird symbolisiert durch das doppelte Klammeraffe-Symbol. Der einfache Klammeraffe wird nicht verwendet, weil er schon im Rahmen der standardisierten Java-Annotationen Verwendung findet.
Nach dem zweifachen Klammeraffe-Symbol folgt ein Schlüsselwort. So ist es möglich, dass bei Kollision des Präfixes mit einem anderen Analysewerkzeug zumindest über das nachgestellte Post-Präfix festgestellt werden kann, ob es sich um eine Annotation im Sinne dieser Arbeit handelt oder nicht. Das Post-Präfix führt sozusagen eine Ebene der Indirektion ein. Indirektionen verkomplizieren ein Gefüge, machen es aber leistungsfähiger.

Notation einer Annotation

Eine wichtige Eigenschaft für Annotationen ist die Eingängigkeit der Syntax. Nur so hat ein Entwickler die Chance, Annotationen effizient anzuwenden. Der Lernaufwand für ein Element, das syntaktischen Regeln zu folgen hat, ist aufgrund der Gewohntheit insbesondere dann gering, wenn die Syntax der der Umgangssprache möglichst nahe kommt. Weiterhin wirkt das Vermeiden komplexer Ausdrücke der Gefahr entgegen, die Praktikabilität der Ausdrücke unterhalb der Akzeptanzschwelle zu haben. Die Vorteile der Verwendung einer Syntax, die ähnlich der der Umgangssprache ist, gegenüber der von formalen Sprachen sind bei einem Prozess mit menschlichem Akteur offensichtlich. [Mainzer 2003, S. 96] führt hierzu aus:

> „Vom Standpunkt eines Informationssystems arbeitet sprachliche Kommunikation mit hoher Redundanz und Unbestimmtheit (*Fuzziness*). Darin wird zunehmend die Überlegenheit natürlicher Sprache gegenüber formalen Sprachen gesehen, die eine Herausforderung für die KI-Forschung darstellt."
>
> **Klaus Mainzer**

Einerseits gilt die natürliche Sprache als mächtig und umfassend, andererseits als ungenau und mehrdeutig. Die ursprüngliche Überlegung des Autors war, die Syntax einer Annotation in einer zur JSR 175 [JCP175 2004] vergleichbaren Form zu gestalten, etwa derart:

```
/**@@motiv unify traversal of similar_elements*/
```

Abbildung 44: Beispiel für eine mögliche Form einer Annotation.

Dies hat allerdings den Nachteil, dass je nach Wahl der Syntax der Annotation entweder die Maschinenverarbeitbarkeit oder die Menschenlesbarkeit leiden. Eine Annotation soll aber sowohl für den Menschen verständlich als auch maschinenverarbeitbar sein. Dies soll gelten, möglichst ohne dass die Verbesserung eines dieser beiden Aspekte eine Verschlechterung des anderen Aspekts bedingt. Aufgrund dessen wird das im Weiteren als duale Annotation bezeichnete Konzept eingeführt.

Duale Annotationen und Annotationstypen

Eine gleichzeitige Maschinenverarbeitbarkeit und Menschenlesbarkeit von Annotationen kann durch die im Folgenden beschriebenen dualen Annotationen realisiert werden. Eine duale Notation besitzt nur eine Aussage und folgt damit [JCP175 2004], [Leavens+ 2003] oder [Meffert+ 2006c]. 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. Die folgende Abbildung 45 veranschaulicht das Prinzip.

Abbildung 45: Maschinen- und menschenlesbarer Teil eines Auszeichnungselementes.

Gemäß einem Warenetikett, das einen Scannercode für die Maschinenverarbeitung und ein frei gestaltbares Etikett für den potentiellen Käufer besitzt, besteht die zusammengesetzte Annotation aus einem Teil für die Maschinenverarbeitung und einem Teil für die Interpretation durch den Menschen.

Ein dualer Annotationstyp definiert die mögliche Ausprägung aller Annotationen eines Typs. Er entspricht einem Schema, dem zugrunde liegende Annotationen gehorchen müssen. Die Definition des Begriffes *dualer Annotationstyp* wird wie folgt vorgenommen:

Definition: Dualer Annotationstyp

> Ein dualer Annotationstyp enthält Informationen über den Aufbau einer Annotation. Diese Informationen beinhalten alle für die Annotation zulässigen Teile. Dazu gehören Parameternamen und -typen sowie die Art jedes Parameters (obligatorisch oder nicht).
>
> Im Annotationstyp ebenfalls angegeben ist der Geltungsbereich, den Annotationen dieses Typs besitzen.
>
> Ein dualer Annotationstyp beinhaltet eine Definition für die maschinenverarbeitbare Komponente (MID) und eine damit korrespondierende menschenlesbare Komponente (HID).
>
> Ein dualer Annotationstyp kann mehreren Annotationen zugeordnet sein.

Annotationstypen können ohne Weiteres hierarchisch organisiert werden. So kann in Anlehnung an die Vererbungshierarchie von Klassen und Schnittstellen in Java pro Annotationstyp ein übergeordneter Annotationstyp angegeben werden. Wird kein solcher Superannotationstyp spezifiziert, gilt die Konvention, dass immer ein einziger einmalig festzulegender Basisannotationstyp mit der MID 0 gilt. Dieses Vorgehen entspricht der Tatsache, dass in Java jede Klasse ohne Angabe

einer Superklasse implizit immer von der Klasse *java.lang.Object* erbt. Der Superannotationstyp kann beliebige, allgemein benötigte Eigenschaften besitzen. Beispielsweise kann er Parameter für Metadaten in Form einer Kollektion von Tupeln bereitstellen. Somit wäre es ohne zusätzliche Deklarationen möglich, in jeder Annotation beliebige Metadaten zu manifestieren. Dazu gehören etwa der Name des Erfassers der Annotation, sowie der Zeitpunkt dieses Vorgangs.
Architektonisch noch flexibler kann die Hierarchie von Annotationstypen gestaltet werden, wenn in einer Konfiguration abgelegt werden kann, welcher Superannotationstyp prinzipiell als Supertyp eingesetzt werden soll. Alle in der Hierarchie über diesem Typen liegenden Annotationstypen dürfen dann nicht explizit als Supertypen verwendet werden. Der Vorteil hierbei ist, dass es sowohl einen über alle Projekte gemeinsamen Supertypen gibt als auch einen projektabhängigen Supertypen, der letztendlich vom projektunabhängigen Supertypen erbt.

Die Verknüpfung zwischen dualem Annotationstyp und Annotationen geschieht über einen Identifizierer. Dieser Identifizierer ist sowohl in Annotationstypendeklarationen als auch in Annotationen enthalten. Der Identifizierer tritt in einem Annotationstypen zweimal auf, zum ersten im maschinenlesbaren Teil und zum zweitem im menschenlesbaren Teil. Die maschinenlesbare Seite (abgekürzt mit MID für *Machine Identification*) eines dualen Annotationstyps besteht aus:

1. Identifizierer (kurz: ID),
2. Parameter (jeder Parameter besitzt einen Namen und einen Typ) und
3. Geltungsbereich zugrunde liegender Annotationen.

Die an den Entwickler gerichtete Seite (abgekürzt mit HID für *Human Identification*) besteht aus:

1. Identifizierer der dazugehörenden MID,
2. Beschreibung in natürlicher Sprache, einschließlich der Parameter und
3. Parameter zur Sprachidentifikation.

Ein Identifizierer muss in irgendeiner Weise erzeugt werden. Eine maschinelle Erzeugung bietet sich an, da ein Identifizierer eindeutig und prägnant sein soll. Ein Identifizierer kann in Form einer s.g. *GUID* (*Globally Unique Identifier*) definiert sein, um möglichst weltweit einzigartig zu sein. Eine *GUID* wird oft mit einem Zufallsalgorithmus und unter Hinzunahme einer so genannten MAC-Adresse (Seriennummer einer eingebauten Netzwerkkarte) erzeugt, die die weltweite Einzigartigkeit mit einer hohen Wahrscheinlichkeit erreicht.
Für die Parameter in MID und HID werden die gleichen Namen verwendet. In der HID werden Parameter durch ein Präfix markiert (hier wird dafür das Klammeraffe-Zeichen »@« verwendet). Der Sprachparameter erlaubt HID-Sektionen in verschiedenen Sprachen zu integrieren. Ein Beispiel ist in Abbildung 46 gegeben.

```
MID:  1234{("var", String, obligatory), ("class", String)}
HID:  "The local variable @var is not used in class @class", English
SCOPES: (PROTECTED_METHOD OR PUBLIC_METHOD) AND METHOD_WITH_EXCEPTION
```

Abbildung 46: Definition eines Annotationstyps.

Der in dieser Abbildung verwendete Typ (*String*) und die Sprache (*English*) können als Konstanten gestaltet werden, die einmal definiert werden müssen. Andere denkbare Typen sind: *Character*, *Integer*, *Boolean*, *Float*, *Date* (vgl. die vorhandenen Java-Typen). Die Verbindung und Referenzierung von MID und HID erfolgt über die ID der MID und durch die gleiche Namensgebung verwendeter Parameter. Groß- und Kleinschreibung werden im Rahmen dieser Arbeit innerhalb von Annotationen nicht unterschieden, da der Autor in einer derartigen Unterscheidung keine Vorteile sieht.

Eine konkrete Annotation, die zum Annotationstyp aus Abbildung 46 passt, ist in Abbildung 47 dargestellt.

```
/*
 * @@motiv
 * MID: 1234("surname", "x")
 * HID: "The local variable @var is not used in class @class", English
 */
private String surname;
```

Abbildung 47: Angewandte Annotation.

Neben MID und HID ist in Abbildung 46 ein Abschnitt *SCOPES* enthalten. Er definiert in Form einer Liste boolescher Ausdrücke, welche Geltungsbereiche (engl.: *Scopes*) für Annotationen des deklarierten Annotationstyps erlaubt sind. Die Geltungsbereiche werden als symbolische Bezeichner angegeben. Jeder symbolische Bezeichner ist einer Prüfroutine zugeordnet, die einen Wahrheitswert zurückliefert. Eine Prüfroutine kann beliebige Prüfungen gegen Quelltext vornehmen. So könnte im Beispiel in Abbildung 46 der Geltungsbereich *PROTECTED_METHOD* mit einer Prüflogik verknüpft sein, die nur Methoden mit der Sichtbarkeit *protected* für Annotationen erlaubt, indem sie nur in diesem Fall *true* zurückliefert. Weiterhin kann jede Geltungsbereich-Prüfroutine alle Parameter mit übergeben bekommen, die in der MID definiert sind. Welche Parameter an eine Prüfroutine übergeben werden müssen, hängt von der Prüfroutine ab. Wie in Abbildung 46 kann es auch Prüfroutinen ohne Parameter geben. Eine Prüfroutine mit einem Parameter, etwa einem Variablennamen, könnte dann in Abwandlung von Abbildung 46 so deklariert sein wie in Abbildung 48 gezeigt:

```
MID:  1234{("var", String, obligatory), ("class", String)}
HID:  "The local variable @var is not used in class @class", English
SCOPES: (PROTECTED_METHOD OR PUBLIC_METHOD) AND METHOD_WITH_EXCEPTION(@var)
```

Abbildung 48: Definition eines Annotationstyps mit parametergesteuerter Prüfroutine.

In Abbildung 48 bekommt die Prüfroutine namens *METHOD_WITH_EXCEPTION* einen Parameter übergeben, nämlich den mit dem Namen *var* und dem Typen *String*. Eine generische Implementierung von Prüfroutinen mit unterschiedlichen Eingabeparametern kann durch Einführung eines einzelnen generischen Parameters realisiert werden. Dieser eine Parameter bildet eine Liste von Schlüssel-Wert-Paaren ab. Eine solche Liste kann etwa durch den eingebauten Java-Typ *java.util.Hashtable* realisiert werden.

Die dargestellte Form von Prüfroutinen erlaubt es nicht nur festzustellen, ob eine Annotation auf eine gegebene Programmentität anwendbar ist. Sie gestattet es mit Hilfe der eingeführten Variablen ebenfalls, die im Kontext der Annotation bedeutungsrelevanten Teile der annotierten Entität zu identifizieren. Im Beispiel der o.g. Annotation ist das die lokale Variable, die nicht verwendet wird. Zusätzlich kann eine weitere Routine vorgesehen werden, die im Falle von nicht angegebenen Variablen den relevanten Teil einer Programmentität durch Analyse des abstrakten Syntaxbaums ermittelt. Im Falle einer annotierten Variablendeklaration etwa kann der Name der deklarierten Variablen über den Syntaxbaum einfach automatisiert identifiziert werden.

Der Autor hat eine größere Anzahl von Gruppen von Operatoren und Anweisungs- bzw. Deklarationstypen für Geltungsbereiche definiert. Es handelt sich um die nach Meinung des Autors offensichtlichsten Elemente für die Sprache Java, zuzüglich einiger weniger Elemente, die aus der Erfahrung heraus hinzugenommen wurden (etwa Persistenzklasse, Datencontainer, Berechnung in der folgenden Aufzählung). Die Gruppen umfassen u.a.:

- Interface-Typen: Marker vs. normale Schnittstellen
- Methodentypen: Konstruktor vs. Instanzmethode vs. statische Methode
- Sichtbarkeiten: *private*, *protected*, *package*, *public*

- Klassentypen: abstrakt vs. konkret; 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.

Verarbeitung von Annotationen

Die maschinelle Verarbeitung von Annotation geschieht über eine Behandlungslogik. Je eine Annotation ist mit einer Behandlerklasse (engl.: *Handler*) verknüpft. Jede dieser Klassen implementiert dieselbe Schnittstelle. Die Schnittstelle enthält eine Methode zum Auswerten der Annotation. Die Methode bekommt als Eingabe alle benötigten Informationen, darunter eine Referenz auf den abstrakten Syntaxbaum des Quelltextes sowie auf die zu prozessierende Annotation.
Welcher Behandler für welche Annotation verantwortlich ist, ergibt sich aus der Identifikation der Annotation, die im Klassennamen kodifiziert ist. Die Namen der Behandlerklassen folgen einer Konvention, die ein festes Präfix vorschreibt, gefolgt von der Identifikation des Annotationstypen, für den die jeweilige Behandlerklasse zuständig ist.
Ein Behandler für das Beispiel aus Abbildung 47 interpretiert den ersten Parameter *@var* als Name einer Variablen und die Aussage der Annotation so, wie im HID-Teil der Abbildung angegeben. Derart kann beispielsweise eine mit dem Werkzeug PMD [PMD 2003] ermittelte verwaiste Variable gekennzeichnet und bei Bedarf verarbeitet werden.

Annotation mehrerer Programmentitäten

Bisher wurde nur die Annotation einer einzelnen Programmentität diskutiert. Eine Sequenz von Programmentitäten kann nicht minder bedeutend sein. Insbesondere bei Anweisungen kann es sein, dass eine Sequenz dieser Anweisungen als Ganzes eine Bedeutung besitzt. Diese Bedeutung sollte durch eine Annotation ebenfalls ausgedrückt werden können.

Gemäß Definition bezieht sich eine Annotation auf die ihr folgende Programmentität. Mehrere sequentiell aufeinander folgende Programmentitäten können somit nicht annotiert werden. Es gibt mehrere Möglichkeiten, dies dennoch zu bewerkstelligen. Erstens kann ein Block eingeführt werden, der keine weitere Auswirkung als die logische Gruppierung der eingeschlossenen Programmelemente besitzt. Dies sieht in Java wie in Abbildung 49 aus:

```
/*
 * @@annotation
 * MID: 1234(...)
 * HID: "...", English
 */
{
  int x = getAmount();
  doSomethingWith(x);
  print(x);
}
```

Abbildung 49: Annotation mehrerer Programmentitäten mittels Blockeinführung.

Die Blockanweisung besteht im Beispiel aus der öffnenden und schließenden geschweiften Klammer. Da die Blockanweisung eine Programmentität ist, kann sie ebenfalls annotiert werden.
Die Einführung einer Blockanweisung zum Zweck der Annotation mehrerer Programmentitäten hat Nachteile. Erstens erschließt sich der Zweck der Blockanweisung nicht unmittelbar. Eine solche Blockanweisung könnte von einem anderen Entwickler wieder entfernt werden, weil er ihr keine offensichtliche Bedeutung beimisst. Ein Analysewerkzeug wie PMD [PMD 2003] könnte ferner die Blockanweisung als unnötig bemängeln. Zweitens kann das Finden der korrespondierenden

schließenden Klammer der Blockanweisung für den Leser des Quelltextes schwierig sein, wenn geschachtelte Blockanweisungen verwendet werden. Außerdem grenzt eine Blockanweisung den Geltungsbereich von darin deklarierten Variablen ein.
Ein Vorschlag für eine semantisch korrekte Lösung ist die Einführung einer Meta-Annotation, die das Ende einer Sequenz annotierter Programmentitäten markiert. Die Abbildung 50 zeigt das Vorgehen.

```
/*
 * @@annotation
 * MID: 1234(...)
 * HID: "...", English
 */
int x = getAmount();
doSomethingWith(x);
print(x);
/*@@end-marker ID=1234*/
```

Abbildung 50: Annotation mehrerer Programmentitäten mit Endemarkierung.

Die Endemarkierung besteht hier aus der Annotation */*@@end-marker ID=1234*/*. Die Referenz des Endemarkers auf die zugehörige Annotation geschieht über die Angabe der Identifikationsnummer der Annotation.
Die Nachteile der o.g. Blockanweisung sind somit behoben. Allerdings ist die manuelle Anbringung einer solchen Endemarkierung nicht so schnell möglich wie das Anbringen einer Blockanweisung. Daher sollte eine zur Annotation des Quelltextes verwendete Entwicklungsumgebung entsprechende Unterstützung bieten (etwa: Markieren einer zu annotierenden Sequenz; Eingabe einer Annotation; die Entwicklungsumgebung fügt die Endemarkierung automatisch ein). Vergleiche auch [Motelet 2004] und [Kazman+ 2002] zur Unterstützung des Entwicklers durch Werkzeuge.

Verfeinerte Definition des Begriffs Annotation

Aufbauend auf dem Begriff des Annotationstypen und der Einführung von MID und HID kann die Definition des Begriffs *Annotation* wie folgt präzisiert werden:

Definition: Annotation (Version 2)

> Eine Annotation ist ein spezieller Kommentar innerhalb eines Programmtextes. Der Kommentar beginnt mit den Zeichen »@@« und einem Schlüsselwort.
>
> Eine Annotation wird eindeutig definiert durch einen maschinenverwertbaren Identifikator (kurz: MID). Die Bedeutung einer Annotation hängt von der Interpretation der MID ab.
>
> In einer Annotation können Parameterwerte festgelegt werden, die dem MID zugeordnet werden, indem sie hinter den Identifikator des MID gestellt werden. Die Art der zulässigen Parameter wird definiert durch einen Annotationstypen, der der Annotation zugeordnet ist. Der zulässige Geltungsbereich einer Annotation wird ebenfalls innerhalb eines Annotationstypen definiert. Jeder Annotation ist genau ein Annotationstyp zugeordnet.
>
> Jedem MID ist eine Behandlungslogik zugeordnet. Sie kann die Validität der Bestandteile der behandelten Annotation (Gültigkeit der angegebenen Parameterwerte) prüfen.
>
> Zur Vereinfachung der Lesbarkeit einer Annotation kann jeder Annotation ein zum MID korrespondierender Identifikator für den Menschen zugeordnet werden (kurz: HID). Der HID enthält alle Parameterwerte, die schon im MID angegeben sind. Die Parameterwerte sind im HID in einen Fließtext eingebettet, der natürlichsprachlich gestaltet sein sollte.
>
> Synonym für »Annotation« kann der Begriff »Auszeichnung« verwendet werden.

Zusammengesetzte Annotationen besitzen den Vorteil, dass nicht zwanghaft nach einer Syntax gesucht werden muss, die gleichzeitig menschenlesbar und maschinenverarbeitbar ist. Denn die Aussage von MID und HID ist äquivalent, die Syntax 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 Annotationen wie in [JCP175 2004] wäre dies auch nur über eine entsprechende Ähnlichkeitstabelle möglich.

3.2.2.5 Zusatzinformationen in Annotationen

Die bisher beschriebene Form von Annotationen ist leicht um zusätzliche Informationen erweiterbar. Im Folgenden werden einige Informationen beschrieben, die zur Verbesserung des Ansatzes beitragen können. Die letztendlich noch zu erbringende Leistung ist die Berücksichtigung und Verarbeitung dieser Informationen innerhalb von Annotationen für die Prozesse Musterdokumentation, -selektion und -anwendung. Dies wird im Rahmen der vorliegenden Arbeit nicht geleistet, weil Zusatzinformationen als Erweiterung auch für andere Anwendungsfälle, etwa Code-Generatoren, angesehen werden können und zudem eine Perspektive geben sollen, wie TrAQ erweitert werden kann.

Metainformationen für Annotationen

Um zu zeigen, welche Entität (Werkzeug, Entwickler) eine Annotation eingeführt hat, kann der Name der Entität angeben werden. So könnte etwa das Werkzeug [PMD 2003] durch die Zeichenkette *tool-pmd* identifiziert werden, respektive ein Entwickler z.B. mit *developer-peter*. Solche Informationen können bei Diagnosen und Empfehlungen hilfreich sein. Ein zusätzliches Freitextfeld ermöglicht einer annotierenden Person, Kommentare abzugeben, weshalb oder nach welchen Regeln eine Annotation eingefügt wurde.

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 könnte es etwa 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 der Analyse eines abstrakten Syntaxbaums
- **Diagnose**: Anweisung zum Kontext
- **Empfehlung**: kann auf eine Diagnose erfolgen

Erkennung der Veralterung von Annotationen

Eine Annotation kann ihre Gültigkeit für die zugehörige Programmentität verlieren, wenn letztere verändert wird. Um das erkennbar zu gestalten, kann für das annotierte Programmelement ein Hash-Wert errechnet werden, der in der Annotation vermerkt wird (siehe auch [Meffert+ 2006c]). Eine integrierte Entwicklungsumgebung, die Annotationen unterstützt, wird so in der Lage sein, Programmänderungen je nach Güte des Hash-Wertes zu erkennen. Indirekte Abhängigkeiten sind diesbezüglich hingegen schwieriger auszuwerten. Beispielsweise müsste für eine Annotation, die sich auf eine ganze Klasse bezieht, jede andere Klasse, die die annotierte Klasse referenziert oder von ihr referenziert wird, mit berücksichtigt werden. Der skizzierte Aspekt der Veralterung von Annotationen hilft bei der Verbesserung des Verfahrens, ist aber generell nicht für die Beschreibung der Funktionsweise erforderlich. Aufgrund der Komplexität dieses Aspekts kann hierauf nicht weiter eingegangen werden. Es ist jedenfalls festzustellen, dass Annotationen nicht schon veralten können wenn die Musterselektion und -anwendung direkt nach Annotation des Quelltextes ausgeführt wird. Insofern ist die Funktionsweise des Verfahrens auch nicht a priori gefährdet.

3.2.3 Isomorphe Programmblöcke und Vergleichbarkeit von Quelltext

Zwei Programmblöcke werden in dieser Arbeit isomorph genannt, wenn sie bedeutungsgleich sind. Bedeutungsgleichheit zweier Programmteile heisst, dass ein Programmteil durch den anderen ersetzt werden kann, ohne das Verhalten des Programms zu verändern. Bedeutungsgleichheit ist im Transformationsprozess zur Musterselektion und Musteranwendung relevant. Ist bekannt, dass Q_a isomorph zu Q_b ist und existiert eine Transformation von Q_b nach Q_z, dann kann Q_a durch vorgelagertes Umformen nach Q_b anschließend nach Q_z transformiert werden. Damit kann eine Transformation für einen gegebenen Programmblock durchgeführt werden, ohne vorher für diesen definiert worden zu sein. Entscheidend ist allerdings auch, dass Isomorphie nicht in jedem Fall gegeben ist, sondern von Bedingungen abhängen kann.

Ein Beispiel für zwei Programmblöcke Q_a und Q_b, die nur unter bestimmten Bedingungen isomorph sind, ist im Folgenden dargestellt. Die nachstehenden Programmblöcke haben die Aufgabe, alle Elemente einer Liste der Reihe nach auszulesen. Programmblock Q_a lautet gemäß Abbildung 51:

```
List l = ...; // Liste konstruieren
int size = l.size();
for(int i = 0; i < size; i++) {
  ... // variabler Teil
}
```

Abbildung 51: Programmblock Q_a für Isomorphiebetrachtungen.

Der als variabel gekennzeichnete Teil ist nicht explizit definiert, sondern wird nur durch Bedingungen (siehe weiter unten) umschrieben. Auf diese Weise ist die Definition eines Programmblocks generisch möglich. Eine derartige generische Definition passt potentiell zu vielen verschiedenen Kontexten. Sie ist effizienter und einfacher als die Angabe möglichst vieler relevanten Alternativen unter unendlich vielen.
Eine generische Definition, die bestimmte Teile berücksichtigt und bestimmte nicht, ist etwa durch Vorgabe eines abstrakten Syntaxbaums möglich, in dem generische Elemente weggelassen oder mit einem Joker-Zeichen markiert werden

Ein zu Programmblock Q_a aus Abbildung 51 isomorpher Programmblock Q_b hat folgende Gestalt (Abbildung 52):

```
List list = ...; // Liste konstruieren
for(int i = 0; i < list.size(); i++) {
  ... // variabler Teil
}
```

Abbildung 52: Programmblock Q_b für Isomorphiebetrachtungen.

Wird zu Programmblock Q_a und Q_b nur der jeweilige abstrakte Syntaxbaum gespeichert, so kann die unterschiedliche Wahl von Bezeichnern homogenisiert werden. Im Fall von Q_a und Q_b betrifft das die Variablen *l* und *list*. Beide Programmblöcke führen eine Schleife über alle Elemente einer Liste durch. In Programmblock A wird der maximale Wert der Schleifenvariable *i* außerhalb der Schleife ermittelt (siehe Variable *size*). Dagegen findet dies in Programmblock Q_b innerhalb der Schleife statt (siehe Ausdruck *list.size()*).
Funktional können Q_a und Q_b zumindest gleich sein, sofern die Bedingung zutrifft, dass innerhalb der Schleife keine Elemente in der Liste hinzugefügt oder entfernt werden. Es ist in diesem Beispiel unerheblich, dass die Liste in Q_a den Namen *l* hat und in Q_b den Namen *list*.
Die Ablaufgeschwindigkeit von A ist als höher anzunehmen als die von Q_b. Dies gilt, wenn durch den verwendeten Compiler keine statischen und beim Programmablauf keine dynamischen Optimierungen durch die Laufzeitumgebung stattfinden. Dann nämlich wird die Größe der Liste in Q_a nur einmalig ermittelt, bei Q_b hingegen in jedem Schleifendurchlauf.

Die Quelltexte Q_a und Q_b haben eine Chance isomorph zu sein, wenn die folgenden Bedingungen für beide Logiken zutreffen. Diese Bedingungen, die der Autor selbst ermittelt hat, sind:

1. Innerhalb der Schleife wird kein Element der Liste entfernt oder hinzugefügt.
2. Die Liste, die über die Schleife ausgelesen werden soll, ist vom Typ »Liste« (etwa Java-Typ *java.util.List*). Das erste Element einer solchen Liste hat den Index 0.
3. Die Schleifenvariable *i* wird pro Schleifendurchlauf um eins erhöht.
4. Abbruchbedingung ist, dass *i < size*, mit *size* = Anzahl der Elemente der Liste.
5. Variable *i* läuft bis zum Schleifenende.
6. Innerhalb der Schleife wird auf die Listenvariable über den Index *i* zugegriffen.
7. Der variable Teil aus Q_a und Q_b ist funktional äquivalent.

Aus diesen Bedingungen können Varianten gebildet werden, etwa für rückwärts zählende Schleifen oder solche, die nur jedes zweite Element einer Liste berücksichtigen. Die siebte Bedingung, die Betrachtung des variablen Teils innerhalb der Schleife, kann ebenfalls über Isomorphenvergleiche geprüft werden.
Schwieriger sind Isomorphe zu erfassen, die funktional identisch sind, aber qualitativ abweichende Implementierungen besitzen. In diese Kategorie fallen beispielsweise zwei Schleifen, wobei die erste wie in Q_a und Q_b realisiert ist (Abbildung 53):

```
List list = ...; // Liste konstruieren
for(int i = 1; i <= list.size(); i++) {
  ... // variabler Teil

  // Zugriff auf Liste über normierten Index
  ... = list.get(i-1);

  ... // variabler Teil
}
```

Abbildung 53: Programmblock Q_c für Isomorphiebetrachtungen.

Programmblock Q_c ist nahezu identisch mit Q_b, bis auf den verschobenen Index. Die Isomorphiebeziehung zu Q_a oder zu Q_b ist nicht so einfach feststellbar wie zwischen Q_a und Q_b. Der Grund ist die Korrelation in Q_c zwischen dem Startindex eins und dem Schleifenabbruchkriterium im Schleifenkopf sowie dem Listenzugriff innerhalb der Schleife. Diese Korrelation ist nicht ohne Weiteres im Rahmen einer Maschinenverarbeitung erkennbar.
Eine zweite Schleife zum Vergleich mit Q_a und Q_b sei (Abbildung 54):

```
List list = ...; //Liste konstruieren
int index = 0;
while (index < list.size()) {
  ...
  index++;
  ...
}
```

Abbildung 54: Programmblock Q_d für Isomorphiebetrachtungen.

Für Programmblock Q_d aus Abbildung 54 kann ohne Einschränkungen die Isomorphie zum Programmblock Q_b festgestellt werden. Dies ist anzunehmenderweise auch für die nichtfunktionale Anforderung der Ablaufgeschwindigkeit der Fall. Hier kann nur eine – wenn auch starke – Annahme getroffen werden, da die interne Arbeitsweise des jeweiligen Compilers und der Laufzeitumgebung das Ergebnis beeinflussen können. Ausgangspunkt der festgestellten Isomorphie zwischen Q_b und Q_d sind wiederum die oben genannten Bedingungen, die auch schon für die Isomorphiebetrachtung zwischen Q_a und Q_b herangezogen wurden.

Ein weiterer, auf den ersten Blick zu Q_a bis Q_d isomorpher Programmtext Q_e ist in folgender Abbildung 55 gegeben:

```
List list = ...; //Liste konstruieren
Iterator it = list.iterator();
while (it.hasNext()) {
  ...
  ... = it.next();
  ...
}
```

Abbildung 55: Programmblock Q_{ϵ} für Isomorphiebetrachtungen.

Um eine Isomorphie zwischen Q_e und beispielsweise Q_a feststellen zu können, muss eine neue Bedingung eingeführt werden. Sie ist asymmetrisch, weil sie für Q_e anders lautet als für einen Programmtext wie Q_a, wo auf eine Listenvariable innerhalb der Schleife zugegriffen werden muss. Sie lautet für Q_e, dass auf die Listenvariable innerhalb der Schleife nicht zugegriffen werden darf, da der Zugriff bereits gekapselt durch den Iterator stattfindet und ein Doppelzugriff negative Seiteneffekte verursachen kann (etwa Konflikte beim sogenannten gleichzeitige Zugriff, engl.: *Concurrent access*). Für andere Programmtexte wie Q_a lautet sie, dass auf die Listenvariable innerhalb der Schleife nur über die Schleifenvariable zugegriffen werden darf, um das jeweils nächste Element aus der Liste auszulesen. Es darf nicht auch das übernächste oder das einen Durchlauf zuvor gelesene Element ausgewertet werden. Wäre das der Fall, widerspräche das den Möglichkeiten, die mit der Logik in Programmblock Q_e realisierbar sind.

Anhand der Betrachtungen zu den wenig komplexen Programmblöcken Q_a bis Q_e wird deutlich, dass selbst für einen einfach anmutenden Quelltext intensive Überlegungen angestellt werden müssen, um dessen Bedeutungsgleichheit zu einem anderen Quelltext zu manifestieren.

Zusammenfassend kann festgehalten werden: Um die Isomorphie zweier Blöcke Q_a und Q_b festzustellen, bedarf es der manuellen Vorgabe von Vorbedingungen. Nur wenn diese Vorbedingungen eintreten besteht die Möglichkeit zur Feststellung der Isomorphie. Es kann im Rahmen dieser Arbeit nicht beantwortet werden, welche Vorbedingungen notwendig sind, um für Q_a die Isomorphie zu Q_b sicher festzustellen. Ausgangspunkt dieses Problems sind sowohl die geforderte Praktikabilität einer Handlungsanweisung für Vorbedingungen als auch die beliebige Komplexität von Q_a.
Es wird vorgeschlagen, aufgrund der beliebigen Komplexität und Gestalt von Vorbedingungen, eine Realisierung solcher Bedingungen mit Hilfe von Prüfroutinen in einer Hochsprache wie Java vorzunehmen. Eine Scriptsprache einzuführen ist nach Meinung des Autors aufgrund der möglicherweise geringeren Leistungsfähigkeit sowie des höheren Lernaufwands nicht sinnvoll (siehe auch Absatz 2.3.2.5). Häufig benötigte Routinen sollten in eine Bibliothek oder ein Framework ausgelagert werden, um deren Wiederverwendbarkeit zu gewährleisten.

Definition: Isomorphie von Programmentitäten

> Zwei Programmentitäten Q_a und Q_b heißen isomorph, wenn sie bedeutungsgleich sind. Bedeutungsgleichheit bedeutet Funktionsgleichheit bei vergleichbaren nichtfunktionalen Anforderungen.
>
> Bedeutungsgleichheit zwischen Q_a und Q_b ist eine Zuordnung, die manuell oder bei Möglichkeit mit Hilfe eines Algorithmus vorzunehmen ist.
>
> Q_a und Q_b sind unbedingt isomorph, wenn die Isomorphie unabhängig von Vorbedingungen und Programmzuständen gilt. Ansonsten sind Q_a und Q_b bedingt isomorph. Besteht zwischen Q_a und Q_b eine funktionale Übereinstimmung, aber keine nichtfunktionale, so wird der Isomorphiebeziehung zwischen Q_a und Q_b das Attribut »funktional« zugeordnet.
>
> Sind Q_a und Q_b bedeutungsgleich und gilt die Isomorphie auch für Q_b und Q_c, so sind Q_a und Q_c ebenfalls isomorph.

Durch die Notwendigkeit der Beachtung von nichtfunktionalen Anforderungen sinkt der Grad der Formalisierbarkeit von Isomorphen. Daher müssen Isomorphe und zugehörige Bedingungen manuell deklariert werden – ein weiterer Hinweis darauf, dass Programmtransformationen nicht vollständig automatisiert durchgeführt werden können[34].

Gemäß der eben genannten Definition wird es erforderlich sein, für jedes mögliche variable funktionale und nichtfunktionale Charakteristikum in Quelltext A eine Vorbedingung zu definieren. Das ist aufgrund der Vielzahl möglicher Charakteristika aber nicht praktikabel. Daher werden die Hauptmerkmale von Q_a in den Vordergrund der Betrachtung gestellt, mit der Möglichkeit unvollständiger Vorbedingungen. Als Hauptmerkmal wird eine signifikante und für einen erfahrenen Entwickler schnell ersichtliche Eigenschaft eines Programmelements verstanden.
Annahme ist dann, dass ein Quelltext Q_a, der aufgestellten Vorbedingungen entspricht, sehr wahrscheinlich keine signifikante Abweichung bzgl. der Isomorphie zu Quelltext Q_b besitzt in nicht durch die Vorbedingungen geprüften variablen Teilen.

In Konsequenz kann aus dieser Annahme die These formuliert werden:

These: Nichterfassbarkeit schlechter Implementierungen durch Vorbedingungen

> Zu schlechte Implementierungen mit signifikanter Komplexität können von Vorbedingungen nicht erfasst werden.

Wenn die Isomorphie zwischen A und B bekannt ist, kann Q_a durch Q_b ersetzt werden. Diese Ersetzung ist eine Transformation. Somit ist die Feststellung der Isomorphie zwischen Q_a und Q_b gleichzeitig die Definition der Transformation von Q_a zu Q_b. Die Definition der Vorbedingungen zur Isomorphie ist in diesem Fall äquivalent zur Definition der Transformation.
Zur letztendlichen Identifizierung von Isomorphen wird vorgeschlagen, über eine Erkennungslogik identifizierte Isomorphe mit Quelltext-Annotationen zu kennzeichnen. Manuell identifizierte Isomorphe können genauso gekennzeichnet werden. Dann ist es möglich, in einem Prozess der von der Information über Isomorphe profitieren kann – etwa bei der Musteranwendung – Annotationen auszuwerten. Das Problem der Isomorphenerkennung kann so vorgelagert und in andere Module ausgelagert werden.

[34] Dies ist mindestens dann gültig, wenn Transformationen auch auf manuell zu deklarierenden Isomorphien zwischen Programmentitäten basieren, was in dieser Arbeit als Voraussetzung angesehen wird.

3.2.3.1 Vergleichbarkeit von Quelltexten

Quelltexte mit derselben Funktion können sehr stark von einander variieren. Das liegt daran, dass es keinen Zwang gibt, eine bestimmte Sequenz von Anweisung für ein bestimmtes Problem zu implementieren. Wenn funktionsgleiche Quelltexte immer gleich wären, könnten dieselben Transformationsregeln für alle diese Quelltexte angewandt werden. Das entspräche einer enormen Erleichterung. Die Notwendigkeit, für minimale Abweichungen im Ausgangsquelltext unterschiedliche Transformationen definieren zu müssen, fiele weg.

Um die Durchführbarkeit einer Transformation T für einen gegebenen Quelltext Q_b feststellen zu können, könnte ein Quelltext Q_a hinzugezogen werden, für den die Durchführbarkeit bereits bekannt ist (etwa dadurch, dass er in einer beispielhaften Transformation verwendet wurde, siehe Abschnitt 3.3.3.2). Die Frage ist, ob Q_b als äquivalent zu Q_a hinsichtlich der Musteranwendung angesehen werden kann. Ist dem so, kann Q_b faktisch durch Q_a ersetzt und die Transformation in alt bekannter Weise durchgeführt werden.
Dies bedeutet, dass ein Algorithmus einen Vergleich zwischen Q_a und Q_b anstellen muss. Das Ergebnis des Vergleichs ist die möglichst gesicherte Feststellung, welche Teile in Q_b isomorph zu denen in Q_a bzgl. T zu behandeln sind. Eine Isomorphie zwischen Q_a und Q_b existiert, wenn

a) Q_a und Q_b gleich sind, oder wenn die nicht gleichen Teile nach Normierung
b) entweder gemäß Vorbedingungen isomorph sind oder
c) als gleich gekennzeichnet sind.

Nicht kompilierrelevante Teile sind etwa Kommentare, überflüssige Leerzeichen und Leerzeilen. Annotationen gehören dieser Menge nicht an, weil sie als Träger von vergleichsrelevanten Informationen beim Vergleich zu berücksichtigen sind.

Definition: Gleichheit von Quelltextteilen

> Zwei Quelltexte Q_a und Q_b sind gleich, wenn sie nach Entfernen nicht kompilierrelevanter Teile, nach Formatierung mit Hilfe einer fest vorgegebenen Regel und nach Normierung von verwendeten Bezeichnern im zeichenweisen Vergleich exakt übereinstimmen.

Die in der Definition erwähnte Formatierungsregel dient dazu, Zeilenumbrüche, die Verwendung von Leerzeichen u.ä. zu normieren. Dies ist wichtig bei einem zeichenweisen Vergleich. Aus demselben Grund findet die Normierung von Bezeichnern statt. Anhand des folgenden Beispiels soll die Notwendigkeit einer Zeichennormierung demonstriert werden. Gegeben sind zwei Quelltexte Q_a und Q_b:

```
public class MeineKlasse {
  private int zahl;

  public void meineMethode(String param1) {
    zahl = Integer.parseInt(param1);
  }

}
```

Abbildung 56: Quelltext Q_a - Beispielhafte Klassenimplementierung.

Der folgende Quelltext Q_b in Abbildung 57 entspricht in seiner Bedeutung exakt dem von Quelltext A, hat aber eine deutlich andere Form (Benennung, Formatierung).

```
public class Converter {
private int number;
public void convert ( String numberString ){
number = Integer . parseInt ( numberString ) ;
}
}
```

Abbildung 57: Quelltext Q_b - Beispielhafte Klassenimplementierung (analog Quelltext Q_a).

Eine Normierung von Q_a und Q_b, die die Gleichheit zwischen beiden Quelltexten veranschaulicht, könnte jeweils für Q_a und Q_b das in Abbildung 58 gezeigte Ergebnis bringen:

```
public class a {
  private int b;
  public void convert(String c) {
    b = Integer.parseInt(c);
  }
}
```

Abbildung 58: Quelltext Q_c – Normierte Form von Quelltext Q_a und Q_b.

Eine Normierungsvorschrift kann mit Hilfe von Pseudocode angegeben werden (Abbildung 59):

```
  Für jede Klasse K:
    Für alle Bezeichner B in K:
      Wenn B nichts bereits ersetzt wurde:
        Ersetze B durch den nächsten freien Namen V.
        Finde alle anderen Vorkommnisse von B (auch außerhalb von K)
         und ersetze sie durch V.
        Vermerke B als bereits ersetzt.
```

Abbildung 59: Pseudocode zur Normierung von Quelltext.

Die Vorschrift aus Abbildung 59 ersetzt alle Bezeichner so, dass jeder Bezeichner gemäß der Reihenfolge des Auftretens durch einen determinierten Namen ersetzt wird.
Bei der Normierung von Quelltext ist weiterhin das Entfernen von unnötigen Programmanweisungen denkbar, beispielsweise von leeren Anweisung (in Java das Semikolon ohne vorhergehende Anweisung) oder von Variablen, die implizit eingesetzt werden könnten, sofern der Wert einer Variablen unverändert bleibt. Ein Beispiel für letzteres (Abbildung 60):

```
  int x = 2;
  int y = x * 3;
```

Abbildung 60: Beispielhafter Ausgangsquelltext vor Normierung.

Der Ausgangsquelltext aus Abbildung 60 kann wie folgt umgeformt werden (Abbildung 61):

```
  int y = 2 * 3;
```

Abbildung 61: Normierte Form des beispielhaften Ausgangsquelltextes.

Das auf dieses sehr einfache Beispiel anwendbare Prinzip kann auf beliebig komplexe Quelltexte angewandt werden. Zumindest für sequentiell ablaufende Quelltexte (ohne *goto*-Anweisung o.ä.)

und für einfache Fälle wie in Abbildung 60 gezeigt kann eine solche Variableneliminierung umgesetzt werden.

3.2.3.2 Vergleich von Quelltext mittels Annotationen

Eben wurde dargestellt, wie zwei Quelltextteile miteinander verglichen werden können. Isomorphe oder nach Normierung identische Quelltexte können einfach identifiziert werden. In allen anderen Fällen ist der Vergleich schwieriger.

Da eine Maschine heutzutage den Zweck eines Programms im Allgemeinen nicht erkennen kann, führt diese Arbeit eine Hilfestellung ein. Diese Hilfestellung besteht aus Annotationen, die im Quelltext manuell oder automatisiert eingefügt werden.

Gemäß den Betrachtungen in Absatz 3.2, ist ein Computer fähig, formalisierte Aussagen mit Hilfe eines Katalogs (im Sinne von »Repository« oder »Datenbank«) zu interpretieren und Katalogaussagen zu treffen. Um einer Software das weitergehende Verständnis der Intention eines Quelltexts zu ermöglichen, werden im vorgestellten Ansatz Annotationen verwendet. Diese Annotationen werden in einem Quelltext Q_a angebracht und ermöglichen es, die Bedeutung einer Programmstelle wiederzugeben. Ein Quelltext Q_b kann nun mit denselben Mitteln annotiert werden, um Analogien zwischen Q_a und Q_b herzustellen. Wird die Repräsentation einer Musterdokumentation geeignet gewählt, lässt sich die Annotationsmöglichkeit in selber Form auf diese Vorlage übertragen.

Zwei Quelltextteile Q_a und Q_b gelten nun als gleich, wenn Q_a durch Annotation A_1 ausgezeichnet ist und Q_b durch Annotation A_2 und A_1 als äquivalent zu A_2 angesehen wird. Zwei Annotationen werden als äquivalent angesehen, wenn dies so definiert wurde. Diese Definition kann derart stattfinden, dass eine Liste von Paaren von äquivalenten Annotationen manuell aufgestellt wird. Zwei Annotationen sind dann gleich, wenn sie entweder als Paar in der Liste zu finden sind, oder deren Gleichheit transitiv ermittelt werden kann. Das heißt, wenn die Annotationen A_1 und A_2 gleich sind und A_2 gleich A_3, dann ist A_1 auch gleich zu A_3. Weiterhin ist die Gleichheit symmetrisch, also wenn A_1 gleich A_2, dann auch A_2 gleich A_1. Es gilt ferner, wenn A_1 ungleich A_2 und A_2 gleich A_3, dann ist A_1 ungleich A_3 sowie A_3 ungleich A_1, A_1 ungleich A_2 und A_2 ungleich A_1.

Das Problem, zwei Quelltexte als gleich zu erkennen, kann also reduziert werden auf das wesentlich einfachere Problem, zwei Annotationen als gleich zu erkennen. Es folgt ein Beispiel für zwei annotierte Quelltexte, die über den Vergleich von deren Annotationen als gleich erkannt werden können. Die Feststellung der Gleichheit beider Quelltexte über deren direkten Vergleich erscheint hier wesentlich schwieriger, da jede Anweisung an sich, zusammen mit ihrer Position in Relation zu den anderen Anweisungen, verglichen werden müsste und Umformungsregeln angewandt werden müssten. Der erste Quelltext Q_a lautet gemäß Abbildung 62:

```
/**@@motiv simple_calculation*/
{
  int x = 5;
  int y = 4;
  for(int i = 0; i < 3; i++) {
    y = x * y;
    y = y – x;
    y = y + x;
  }
}
```

Abbildung 62: Annotierter Quelltext Q_a.

Der folgende Quelltext Q_b in Abbildung 63 ist selbst mit Hilfe von komplexen Umformungsregeln und ohne Programmablaufanalyse nicht ohne Weiteres als funktionsgleich zu Q_a aus Abbildung 62 erkennbar. Ohne eine dynamische Analyse durch Programmablauf wäre ein Erfolg unwahrscheinlich.

```
/**@@motiv simple_calculation*/
{
  int y = 4;
  int x = 5;
  y = y * x * x * x;
}
```

Abbildung 63: Zu Quelltext Q_a als funktionsgleich annotierter Quelltext Q_b.

Anstelle der Annotation *motiv simple_calculation* in Q_a oder Q_b hätte auch eine andere Annotation verwendet werden können, wenn sie in der oben erwähnten Liste von Paaren als gleichwertig zur jeweils anderen verwendeten Annotation in Q_b bzw. Q_a enthalten ist.

These: Erkennung funktionsgleicher Anweisungssequenzen mit Annotationen

> Zwei Anweisungssequenzen, die mit als äquivalent angesehenen Annotationen versehen sind, lassen sich als funktionsgleich identifizieren. Die Äquivalenz zweier Annotationen muss manuell definiert werden.

Eine Problematik, die sich beim Vergleich von Annotationen anstatt von Quelltext ergibt, ist die mögliche Veralterung des annotierten Quelltextes (hierauf wurde in Absatz 3.2.2.5 eingegangen).

3.3 Das neue Verfahren TrAQ im Detail

3.3.1 Überblick

TrAQ basiert auf Semantikinformationen, die explizit in Form von Annotationen in einen Quelltext eingefügt werden. Denn gerade das semantische Verständnis eines Programms fehlt einer Maschine. Einerseits werden diese Informationen bei der formalen Musterdokumentation verwendet. Andererseits können Annotationen in einem bestehenden Quelltext eingeführt werden, um in diesem Entwurfsprobleme und -intentionen zu verknüpfen (vgl. [Meffert+ 2007b]). Damit sollen sowohl die Selektion anwendbarer Muster sowie die Anwendung eines selektierten Musters ermöglicht werden-

Das Gesamtverfahren wird in der folgenden Abbildung 64 im Überblick gezeigt.

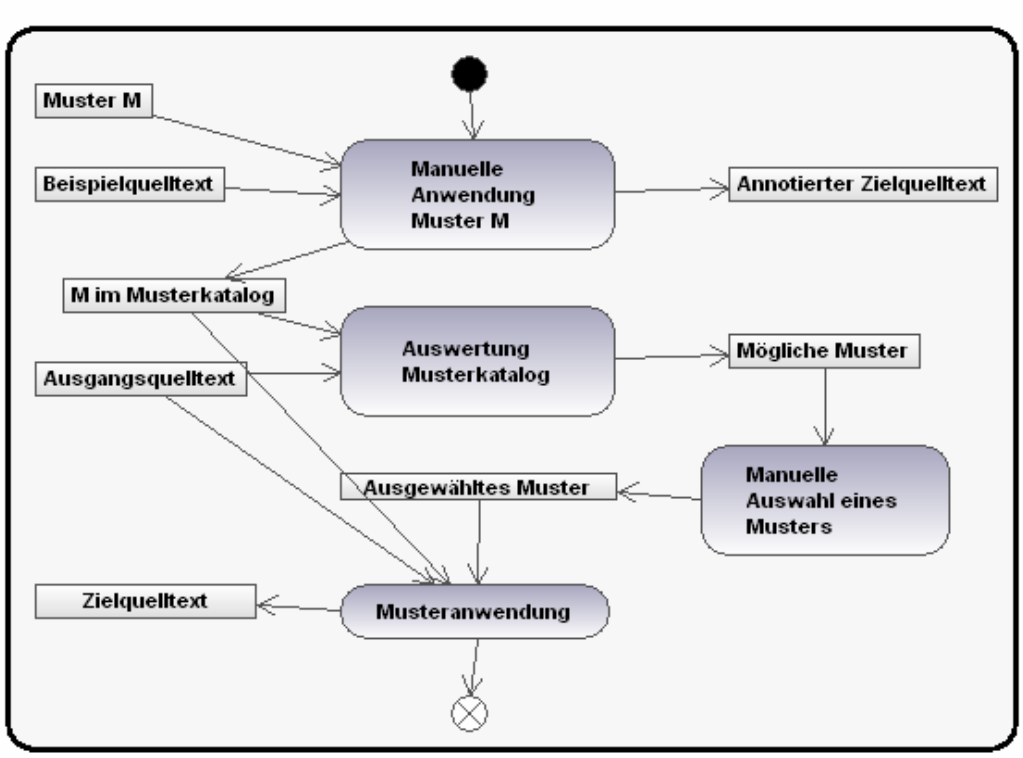

Abbildung 64: Grundprinzip des Verfahrens TrAQ im Überblick.

Startpunkt von TrAQ ist der Prozess der Musterdokumentation, der im oberen Bereich der Skizze dargestellt ist. Die Musterdokumentation wird von einem erfahrenen Entwickler oder Architekten durchgeführt. Der Prozess beginnt mit der manuellen Auswahl eines zu dokumentierenden Musters M sowie eines Beispielquelltextes, auf den M sinnvoll angewandt werden kann. Die Auswahl des Musters kann sich etwa auf ein Buch stützen, in dem das Muster dokumentiert ist oder auf ein Muster, das noch gar nicht (auch nicht in einem Buch) dokumentiert wurde. Ergebnis ist die Dokumentation des Musters M sowie ein annotierter Zielquelltext, auf den M manuell angewandt wurde. Die bei diesem Prozess manuell vorgenommenen Quelltexttransformationen werden vom System aufgezeichnet. Eine Transformation entsteht während der Bearbeitung des Quelltextes und kann durch einen Vorher-Nachher-Vergleich festgestellt werden. Eine Transformation ist etwa das Hinzufügen einer Methode oder das Entfernen einer Anweisung. Das Ergebnis der Aufzeichnung ist ein im Musterkatalog dokumentiertes Muster M. Der Musterkatalog verzeichnet somit alle formal dokumentierten Muster.
Mittels des Musterkatalogs kann danach ein Anwender für einen bisher unbekannten Ausgangsquelltext feststellen lassen, ob ein Muster aus dem Katalog anwendbar ist. Dies geschieht unter anderem mit Hilfe von Annotationen. Annotationen wurden einerseits im Rahmen der Musterdokumentation etabliert. Andererseits werden sie im gegebenen Ausgangsquelltext gesucht und mit denen der Musterdokumentation verglichen[35]. Ist eine genügend große Ähnlichkeit für ein Muster gegeben, kann es als anwendbar angesehen werden und dem Anwender als ein mögliches Muster präsentiert werden. Die Musterselektion und die folgende Musteranwendung können also im Gegensatz zur Musterdokumentation von einem durchschnittlich begabten Entwickler durchgeführt werden.
Wird ein anwendbares Muster zur Anwendung vom Entwickler aus der Menge der als von TrAQ als geeignet selektierten Alternativen ausgewählt, kann es unter Auswertung der Transformationsschritte, die im Musterkatalog zur Musterdokumentation hinterlegt sind, angewandt werden. Ergebnis der Anwendung ist der links unten in der Abbildung 64 dargestellte Zielquelltext.

[35] Die Annotation eines Quelltextes als Grundlage für die Musterselektion wird in Abschnitt 3.3.7 beschrieben.

TrAQ basiert also darauf, dass es unterschiedliche Rollen der menschlichen Akteure bei der Musterdokumentation einerseits und der Musterselektion und -anwendung andererseits gibt. Die Rolle, die die durchführende Person bei der Musterdokumentation einnimmt, setzt tiefer gehende Kenntnisse in der Entwicklung und Arbeit mit Entwurfsmuster voraus. Im Gegensatz dazu erfordert die Rolle der Person, die die Musterselektion oder -anwendung durchführt, eine geringere Kompetenz voraus. TrAQ stellt dem weniger erfahrenen Entwickler mit Hilfe der Musterdokumentation die Kompetenz eines erfahrenen Entwicklers zur Verfügung.

Im Folgenden wird eine Übersicht über die Aktivitäten Musterdokumentation, Musterselektion und Musteranwendung gegeben. Im Anschluss wird das Verfahren zur werkzeuggestützten Realisierung der eben genannten Aktivitäten im Detail dargestellt.

3.3.1.1 Musterdokumentation

Ausgangspunkt für die formale Musterdokumentation ist ein erkanntes, zu dokumentierendes Muster. Dieses Muster muss der Person, die die Musterdokumentation durchführt, so gut wie möglich bekannt sein. Durch Erstellen der Dokumentation zu weiteren Mustern füllt sich der Katalog formal dokumentierter Muster, die somit für die Musterselektion und -anwendung zur Verfügung stehen.
Eine gängige Forderung für Entwurfs- und verwandte Muster ist, dass ein Muster nur dann als solches angesehen wird, wenn es mindestens in drei Projekten erfolgreich angewandt wurde (vgl. Abschnitt 1.1.3). Ein Muster sollte also bereits mehrmals angewandt worden sein, bevor es formal dokumentiert wird.
Die in dieser Arbeit formale Musterdokumentation basiert auf der exemplarischen Anwendung des Musters. Die folgende Abbildung 65 stellt die zugrunde liegende Idee dar:

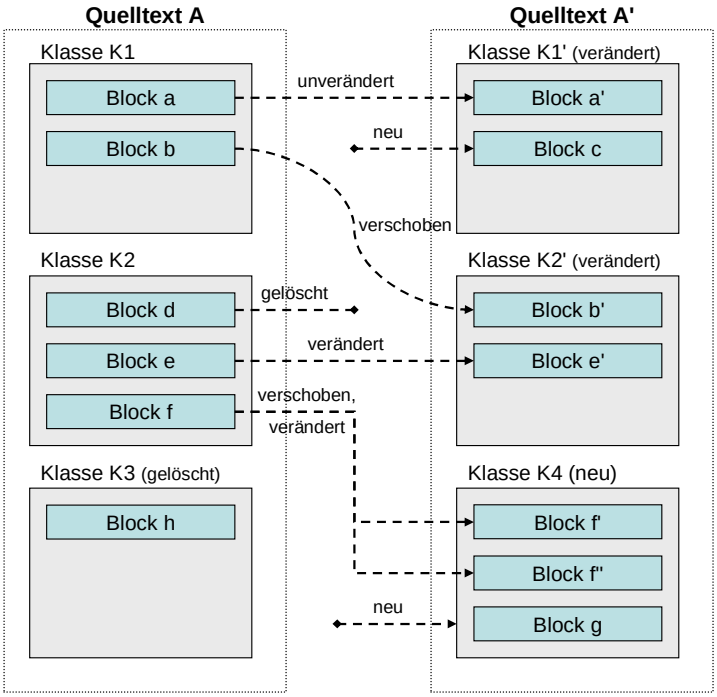

Abbildung 65: Konzept der Musterdokumentation durch Transformation.

In der Abbildung sind zwei Quelltexte A und A' dargestellt. Jeder Quelltext besteht aus mehreren Klassen, jede Klasse aus einem oder mehreren Blöcken (repräsentiert durch Methoden oder Deklarationen und Anweisungen auf Klassenebene).
Ausgangspunkt der Abbildung 65 ist der Beispielquelltext A. Er wird manuell von einem erfahrenen Entwickler gewählt. Weiterhin wird entweder ein formal zu dokumentierendes Muster M gewählt, das entweder bereits informal (wie in [Gamma+ 1995]) oder noch gar nicht dokumentiert ist. Dieses

Muster M wird auf A angewandt. Das geschieht faktisch durch manuelle Ausführung von Transformationsschritten. Ein solcher Schritt ist etwa das Hinzufügen einer Methode oder das Löschen einer Anweisung. Ergebnis des manuellen Transformationsprozesses ist der Quelltext A'. Er entspricht Quelltext A plus angewandtem Muster M.
Die bei der Transformation anfallenden Informationen werden extrahiert, insbesondere die Zuordnung eines Quelltextteils in A zu einem Teil in A', der durch Transformation erzeugt wird. Zur Erleichterung dieser Zuordnung werden manuell Annotationen während des Transformationsprozesses in den Quelltext eingebracht. Diese Annotationen erlauben es weiterhin, die Bedeutung eines Quelltextteils, etwa einer Anweisung oder einer Deklaration, im Rahmen der Musterselektion und -anwendung zu definieren. Die nächste Abbildung 66 zeigt schematisch das Prinzip.

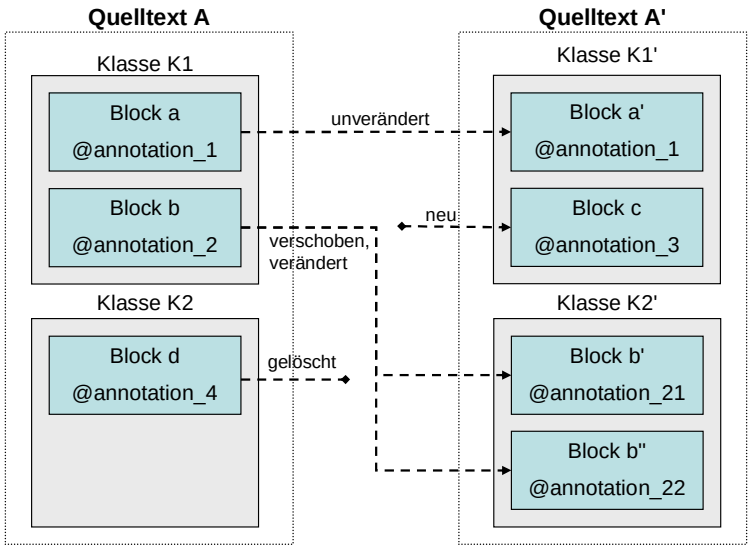

Abbildung 66: Annotation von transformierten Quelltextteilen.

Auf der linken Seite sind zwei Klassen K1 und K2 aus Quelltext A abgebildet, die die Ausgangssituation darstellen. Sie werden durch Quelltextänderung transformiert in die Klassen K1' und K2'. Gegenstand der Transformation sind jeweils Quelltextblöcke. Die in der Abbildung angedeuteten Annotationen markieren die Korrespondenz zwischen Ausgangs- und Zielsituation in einer maschinell verarbeitbaren Weise. Die Art der Korrespondenz zwischen Quelltext A und A' gibt die durchgeführte Transformation an. Im Falle von Block a (Abbildung 66, linke Seite) handelt es sich um ein unverändertes Übernehmen von Quelltext A nach A'. Block b (selbe Abbildung) wird verschoben in eine neue Klasse und dabei auf zwei Teilblöcke aufgeteilt. Block c (rechts in Abbildung 66) wird im Zielquelltext A' neu eingeführt ohne eine Korrespondenz in Quelltext A zu haben. Block d wird aus Quelltext A ersatzlos gelöscht. Diese Feststellungen können durch Suche von in A bzw. vorhandenen korrespondierenden Annotationen im jeweils anderen Quelltext A' bzw. A gemacht werden. Ist etwa ein Block b in Quelltext A mit einer Annotation markiert, die auch in Quelltext A' existiert, ist eine Verschiebung, veränderte oder unveränderte Übernahme von Block b aus A nach A' folgerbar.
Mit Hilfe der im Transformationsprozess von A nach A' gewonnenen Informationen soll ein bisher unbekannter, gegebener Quelltext B mit A in Einklang gebracht werden, so dass M für B selektiert und auf B angewandt werden kann. Das Ergebnis der Anwendung von M auf B ist B'. Dieses Ergebnis soll durch möglichst gut automatisierte Transformationen, die während der Musteranwendung gewonnen werden, herbei geführt werden.

3.3.1.2 Musterselektion

Das Vorhandensein eines Musterkatalogs, der formal dokumentierte Muster enthält, ermöglicht die werkzeuggestützte Selektion eines dieser Muster für einen gegebenen Quelltext. Die

Musterselektion soll dabei so automatisiert wie möglich ablaufen. Da es Mustervariationen gibt, also ähnliche Ausprägungen eines Musters, die dem Kernmuster zuzuordnen sind, muss der Mensch alleine schon zur Auswahl einer Mustervariante einbezogen werden. Er muss ebenfalls konsultiert werden, um ein maschinell als geeignet für die Anwendung selektiertes Muster zu bestätigen. Zu groß sind die Unsicherheiten bei der Interpretation eines Quelltextes durch eine Maschine, als das damit zu rechnen wäre, dass dies ohne Fehler gelingt.
Das Verfahren zur Musterselektion ist in der folgenden Abbildung 67 skizziert:

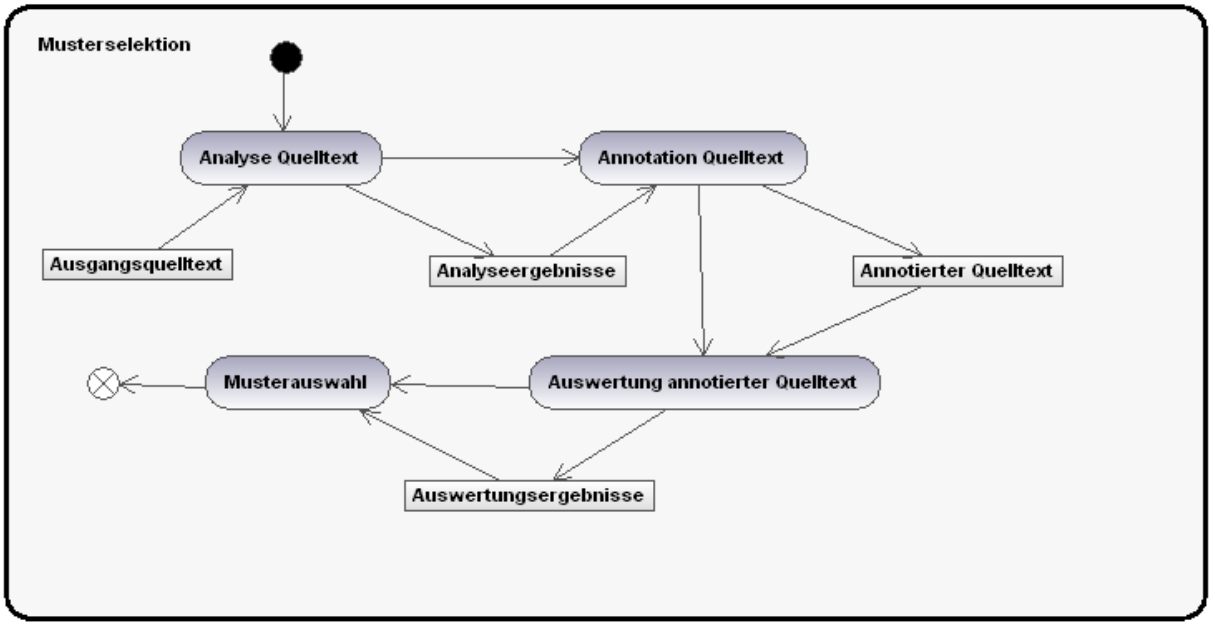

Abbildung 67: Prinzip der Musterselektion mit Annotationen im Aktivitätsdiagramm.

Ausgangspunkt des in Abbildung 67 skizzierten Vorgehens zur Musterselektion ist ein gewöhnlicher Quelltext, der gemäß dem hier vorgestellten Ansatz analysiert wird. Ergebnis der Analyse sind Vorschläge für im Quelltext anzubringende Annotationen. Mit Hilfe dieser Vorschläge und eigener Überlegungen kann der gegebene Quelltext annotiert werden, entweder durch einen Entwickler oder ein Werkzeug, welches die Analyseergebnisse verarbeitet. Das Ergebnis ist in jedem Fall ein annotierter Quelltext. Der nächste Schritt ist die Auswertung des annotierten Quelltextes. Die Auswertungsergebnisse werden dem Entwickler im nächsten Schritt präsentiert. Die Auswertung basiert auf der Berücksichtigung von Quelltext-Annotationen und isomorphen Programmblöcken. Die Auswertung berücksichtigt alle im Musterkatalog vorhandenen Muster. Pro Muster wird ermittelt, inwieweit Annotationen im vorliegenden Quelltext mit solchen in einer im Musterkatalog vorliegenden formalen Musterdokumentation übereinstimmen und ob Gleichheiten zwischen Quelltext und Musterdokumentation existieren oder durch isomorphe Programmblöcke abgebildet werden. Dementsprechend kann dem Entwickler eine Liste von Mustern angeboten werden. Pro Muster werden die Übereinstimmungen und Unterschiede zwischen Quelltext und Musterdokumentation präsentiert. Der Entwickler wählt aus den für die Anwendung als sinnvoll ermittelten Mustern eines aus, um es letztendlich anzuwenden.

Indem bereits vorhandene Musterteile im gegebenen Quelltext ermittelt werden können, findet im Rahmen der hier beschriebenen Musterselektion eine implizite Mustererkennung statt. Denn durch den Vergleich mit der Musterdokumentation und unter Berücksichtigung von Annotationen, Isomorphen, Synonymen und Ähnlichkeitstabellen für Annotationen können bereits implementierte Musterteile identifiziert werden.

3.3.1.3 Musteranwendung

Die Musteranwendung folgt der Musterselektion. Auch bei der Musteranwendung ist eine vollständige Automatisierung nach Meinung des Autors nicht vorstellbar[36].

These: Automatisierbarkeit der Anwendung von Mustern

> Für ein beliebiges Muster, das den Entwurfsmustern oder ähnlichen Mustern zuzuordnen ist, kann eine vollautomatisierte Anwendung dieses Musters auf einen gegebenen, unbekannten Quelltext in einer objektorientierten Hochsprache wie Java oder C# nicht garantiert werden.

Einerseits wird die Einschränkung der Automatisierbarkeit angenommen, weil die Transformation eines Quelltextes, die für eine Musteranwendung nötig ist, für Hochsprachen wie Java oder C# nicht für beliebige Quelltexte im Voraus definiert werden kann. Das hat zur Folge, dass Transformationen in einzelnen Fällen fehlerhafte Zielquelltexte hervorbringen können, die etwa nicht kompilierbar oder funktional vom Soll abweichend sind. Zweiter Grund zur Einbeziehung des Menschen bei der Musteranwendung ist, dass es Variationspunkte in Mustern gibt. Einfachstes Beispiel sind Bezeichnungen von Klassen, Methoden und Variablen.

These: Einbeziehung des Menschen bei der Selektion und Anwendung von Mustern

> Für ein beliebiges Muster, das den Entwurfsmustern oder ähnlichen Mustern zuzuordnen ist, ist es sowohl bei der Selektion des Musters für einen gegebenen unbekannten Quelltext in einer objektorientierten Hochsprache wie Java oder C# als auch für die Musteranwendung eines gewählten Musters nicht möglich, auf die Einbeziehung des Menschen zu verzichten.

Ergebnis der Musterselektion ist die Festlegung eines anzuwendenden Musters. Zur Anwendung sind Quelltexttransformationen durchzuführen, die in der formalen Musterdokumentation definiert sind. Der Entwickler kann diese Transformationen bewerten, indem er einige abwählt sowie eigene Transformationen ergänzt. Im nächsten Schritt startet der Entwickler den Anwendungsprozess des Musters durch Bestätigung der Korrektheit aller Angaben. Mit Hilfe der Musterdokumentation wird die Musteranwendung durchgeführt, indem der Ausgangsquelltext um neue Teile angereichert und in bestehenden verändert wird, so dass letztendlich ein Quelltext entsteht, der das angewandte Muster enthält.

Die Transformation des gegebenen, bisher unbekannten Ausgangsquelltextes basiert auf Annotationen, die in diesem Quelltext anzubringen sind. Werden im Ausgangsquelltext Annotationen gefunden, die mit denen aus der Musterdokumentation korrespondieren, so kann eine Verknüpfung zwischen Musterdokumentation und Ausgangsquelltext hergestellt werden. Für die annotierte Stelle im Quelltext kann dann eine in der Dokumentation hinterlegte Transformation für die Ausführung ermittelt werden. Nicht über Annotationen zwischen Dokumentation und Ausgangsquelltext verknüpfbare Quelltextstellen können eventuell über eine Syntaxanalyse in Relation gestellt werden

[36] Vgl. auch den Artikel "Was, bitte, bedeutet Objektorientierung?" von Stefan Jähnichen und Stephan Herrmann im Informatikspektrum 8 (Band 25, Heft 4,August 2002), S. 269: "[…] Man darf niemals vergessen, dass gute Software nur entwickelt werden kann, wenn die Menschen, die daran arbeiten, gute Arbeit leisten. Programmiersprachen können nur einen äußeren Rahmen schaffen. Methoden versuchen deutlich konstruktivere Hilfe anzubieten, sind allerdings von deutlich heuristischer, d.h. auch vager Natur."

(siehe auch [Meffert 2007a]). Gelingt auch dies nicht, muss das tiefer gehende Erfahrungswissen des Anwenders[37] herangezogen werden.
Mit der Musterdokumentation, der Musterselektion und der Musteranwendung sind die Kernaspekte des Gesamtverfahrens skizziert, die alle auf Annotationen basieren.

3.3.2 Prozessphasen von TrAQ

Zunächst folgt eine Übersicht der Prozessphasen. TrAQ besteht aus den Phasen A bis F, deren Zusammenhang in der folgenden Abbildung 68 dargestellt ist:

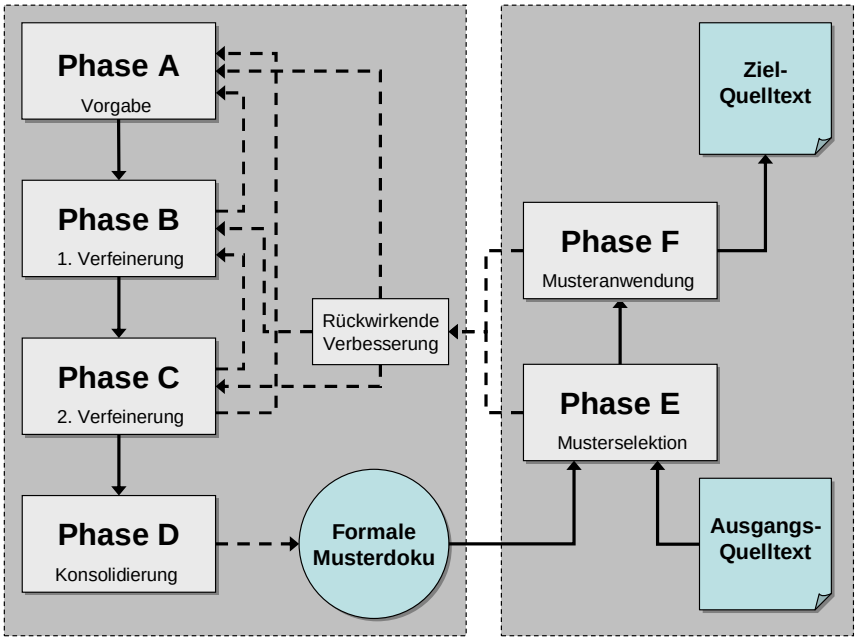

Abbildung 68: Die Phasen des Verfahrens im Überblick.

Die in der Abbildung dargestellt Phasen sind im Einzelnen:

Muster aussuchen: Phase A wird initiale Dokumentationsphase genannt. Hier wird von einem Entwickler, der die Musterdokumentation durchführt, entschieden, welches Muster formal zu dokumentieren ist. Weiterhin wird manuell ein beispielhafter Ausgangsquelltext ausgewählt, auf den dieses Muster anwendbar ist. Danach wird das Muster manuell auf dem Ausgangsquelltext angewandt.

Ausgangsquelltext verändern: In Phase B, der ersten Verfeinerungsphase, wird der Ausgangsquelltext modifiziert, um der Tatsache Rechnung zu tragen, dass bei der Musterauswahl und -anwendung unterschiedlichste Quelltexte zu berücksichtigen sind. Weiterhin wird das Muster auf den modifizierten Ausgangsquelltext angewandt.

Muster verändern: Phase C stellt die zweite Verfeinerungsphase dar. Hier wird der Ausgangsquelltext aus Phase A verwendet. Das in Phase A gewählte Muster wird variiert und dann auf den Quelltext angewandt. Motivation hierfür ist die häufige Variation eines Musters durch den anwendenden Entwickler. Beispielsweise gibt es das *Singleton*-Muster in einfacher Ausführung, die nicht Thread-fähig ist, und in der nach Erfahrung des Autors weniger verbreiteten Thread-fähigen Variante.

[37] Erfahrungswissen kann möglicherweise auch formalisiert abgelegt und maschinell ausgewertet werden, was im Rahmen dieser Arbeit nicht weiter untersucht werden kann.

Musterdokumentation zusammenstellen: Phase D widmet sich der Konsolidierung der Ergebnisse aus den vorigen Phasen. In dieser Phase wird die formale Musterdokumentation aufgestellt und in den Musterkatalog gestellt.

Ab hier ist das Muster so dokumentiert, dass es für die Musterselektion und -anwendung im Produktivbetrieb verwendet werden kann:

Musterselektion: In Phase E findet die Musterselektion für einen gegebenen Ausgangsquelltext mit Hilfe von formalen Musterdokumentationen statt. Jedes im Musterkatalog vorhandene Muster kann bei der Musterselektion berücksichtigt werden.

Musteranwendung: Phase F führt die Anwendung des in Phase E ausgewählten Musters durch und nimmt hierbei die formale Dokumentation des Musters zu Hilfe.

Die einzelnen Phasen werden im Folgenden der Reihe nach beschrieben.

3.3.3 Phase A: Initiale Musterdokumentationsphase

Ziel dieser in Abbildung 69 skizzierten Phase ist es, ein formal zu dokumentierendes Muster manuell auszuwählen und es auf einen ebenfalls manuell zu wählenden exemplarischen Ausgangsquelltext anzuwenden.

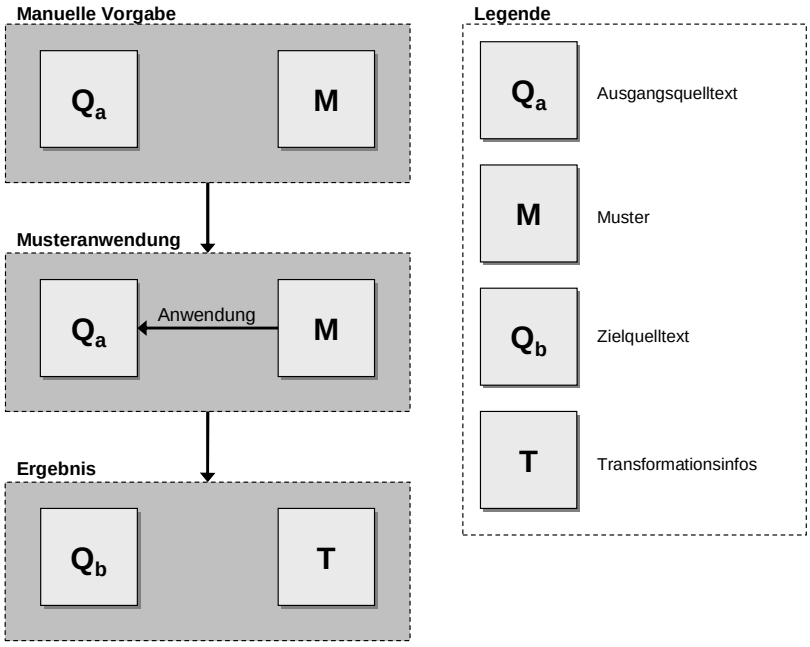

Abbildung 69: Phase A von TrAQ - Initiale Dokumentationsphase.

Die Person, die die Schritte dieser Phase unternimmt, wird weiterhin Dokumentator genannt. Der Dokumentator ist ein Entwickler, der sich hervorragend mit Entwurfsmustern auskennt und überdurchschnittliche Fähigkeiten in der Software-Entwicklung besitzt.
Zu Beginn dieser ersten Phase von TrAQ wird von einem Dokumentator ein formal zu dokumentierendes Muster M gewählt, das nicht bereits dokumentiert wurde. Passend zu M wird vom Dokumentator ein Ausgangsquelltext Q_a gewählt. Q_a muss so beschaffen sein, dass M sinnvoll darauf angewandt werden kann. Sind Q_a und M vorgegeben, kann M auf Q_a manuell vom Dokumentator angewandt werden. Aus dieser exemplarischen Anwendung von M geht ein Zielquelltext Q_z hervor. Weiteres Produkt dieser Anwendung sind Transformationen T, die nötig waren um von Q_a nach Q_z zu gelangen. Die Schritte der Phase werden nacheinander an einem komplexen Beispiel beschrieben.

3.3.3.1 Auswahl eines zu dokumentierenden Musters

Für die Auswahl des Musters gibt es keine speziellen Regeln. Jedes Muster, was dem Dokumentator geeignet erscheint, kommt in Frage. Hier bietet das vorgestellte Verfahren keine Hilfestellung. Es wäre immerhin denkbar, zu dokumentierende Muster vorher auf Ihre allgemeine Güte bzgl. der Eignung in der Software-Entwicklung hin zu untersuchen. Dies soll aber nicht Gegenstand dieser Arbeit sein.

3.3.3.2 Auswahl eines geeignetes Ausgangsquelltextes

Als nächstes ist vom Dokumentator ein Quelltext Q_a auszuwählen. Anforderung an Q_a ist, dass Q_a sich eignet, um Muster M darauf anzuwenden. Auch hierzu wird an dieser Stelle keine verfahrenstechnische Unterstützung geboten. Es existieren ja üblicherweise für ein Muster - wie weiter oben gefordert - mindestens drei Anwendungsfälle. Aus diesen kann ein Beispiel gewählt werden. Alternativ kann ein geeigneter Quelltext konstruiert werden.

3.3.3.3 Anwendung des zu dokumentierenden Musters

Zuvor wurde das Muster M ausgewählt und dafür ein geeigneter, beispielhafter Ausgangsquelltextes Q_a gefunden. Nun wird M auf Q_a tatsächlich angewandt. Dies geschieht in Teilschritten, die als Transformationen bezeichnet werden. Jeder Schritt wird dabei für den Ansatz dokumentiert. Die vom Dokumentator manuell durchgeführten Transformationen $T(Q_a, M) \rightarrow Q_z$ sind so vom System festzuhalten, dass später möglichst umfänglich automatisiert $T(Q_b, M) = Q_z'$ durchgeführt werden kann, mit Q_b als bisher unbekanntem Quelltext und Q_z' als Zielquelltext mit angewandtem Muster M.

3.3.3.4 Beispiel für die initiale Musterdokumentationsphase

Auswahl eines zu dokumentierenden Musters

Nach der theoretischen Beschreibung der initialen Phase A folgt in diesem Abschnitt die Demonstration anhand eines Beispiels. Als Muster wurde *Observer* gewählt. Die Wahl fiel zum einen auf dieses häufig verwendete Muster, weil es ausreichend komplex ist und eine Reihe von beteiligten Klassen vorweisen kann. Zum anderen sind Variationen des Musters dokumentiert und ebenfalls populär. *Observer* ist zudem nicht den Erzeugungsmustern zuzurechnen, welche durch die einfach identifizierbare Kopplung mit Konstruktoren und Konstruktionsbefehlen (wie *new* in Java) wesentlich weniger komplex und leichter handhabbar sind (vgl. das *Singleton* in Anhang A). Dass sich das Beispiel an *Observer* orientiert, stellt keine Einschränkung dar. Durch die Wahl dieses Musters ergibt sich zwar ein bestimmter Ausgangsquelltext (siehe folgenden Abschnitt). Die verfahrenstechnische Behandlung des Ausgangsquelltextes durch die generischen Annotationen sowie durch Analyse von abstrakten Syntaxbäumen ist allerdings musterunabhängig. Die im Anhang dargestellten Muster *Singleton* und *Composite* zeigen, dass eine Behandlung anderer Muster mit Hilfe von Motiven und Annotationen sowie die Transformation im Rahmen der Musterdokumentation an sich musterübergreifend möglich ist (siehe auch *Iterator* und *Composite* in [Meffert 2006a] und [Meffert 2007a]). Das Verfahren kann in der beschriebenen Weise für jedes Muster angewandt werden. Für Flexibilität sorgen die verwendeten Motive und Annotationen, die prinzipiell jede Semantik abbilden können. Die weiterhin für Programmentitäten eingeführten Isomorphe sind a priori kontextfrei und somit für jedes Muster verwendbar. Die im Rahmen der Musterdokumentation durchzuführende Quelltexttransformation ist für jedes Muster auf die beschriebene Art durchführbar. Die sich an die Musterdokumentation anschließenden Prozess der Musterselektion und -anwendung sind unabhängig von einem bestimmten Muster. Sie stützen sich auf einen Katalog, der sämtliche bereits dokumentierten Muster enthält.

Alle folgenden Beispiele beziehen sich auf die Programmiersprache Java.

Auswahl eines geeignetes Ausgangsquelltextes

Passend zum gewählten Muster *Observer* wurde ein Quelltext ermittelt, auf den das Muster angewandt werden kann. Der Java-Quelltext Q_a hat folgende Gestalt (zur einfacheren Bezugnahme werden die Zeilennummern mit angegeben; nach manchen Zeilennummern und vor der eigentlichen Programmzeile folgen Marker wie *acd*, die Motive ausdrücken und später erläutert werden):

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Observierer
040        PatDetails bo2 = new PatDetails();
050        // Konstruiere Observierten und registriere Observierer
060 acd    PatSelector bo1 = new PatSelector(bo2);
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110    public class PatSelector{
120      // Der Observierende
130 ab   private PatDetails dependent;

140      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
150      // benötigt)
160      private Object state;

170      // Konstruiere Objekt und registriere einen Observierenden
180 abcd public PatSelector(PatDetails dependent) {
190 abd    this.dependent = dependent;
200      }

210      public void start() {
220        //... tue irgendwas hier ...
230        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
240        doWorkChangingState();
250        //... tue auch noch irgendwas anderes hier ...
260      }

270      // Diese Methode tut etwas, was den Zustand dieses Objekts signifikant
280      // beeinflusst. Daher wird in ihr der Observierende informiert. Ihm wird
290      // der aktuelle Zustand dieses Objekts mit übergeben, damit er den Zustand
300      // auswerten kann.
310      public void doWorkChangingState() {
320        System.out.println("Ich tue etwas, was meinen Zustand verändert");
330        System.out.println("Deshalb informiere ich PatDetails darüber");
340 abef   dependent.refresh(state);
350      }
360    }

370 a  public class PatDetails {
380 ef   public void refresh(Object state) {
390        // Aktualisiere Zustand und Darstellung
400        // ...
410        System.out.println(reagiere auf Zustandsänderung von PatSelector");
420      }
```

```
430     }
```

Abbildung 70: Exemplarischer Ausgangsquelltext für Anwendung des Observer-Musters.

Der Quelltext aus Abbildung 70 repräsentiert ein komplettes, kompilierbares und sinnvoll lauffähiges Java-Programm. In diesem Quelltext sind drei Klassen angegeben, nämlich *PVAnwendung* (*PV* = Patientenverwaltung), *PatSelector* und *PatDetails*. Jede der drei Klassensignaturen ist zur besseren Übersichtlichkeit fett gedruckt. Die Klassen stellen einen Teil einer Patientenverwaltungsanwendung dar. Die Klasse *PVAnwendung* stellt den Einstiegspunkt der Anwendung dar, weil die Anwendung durch sie gestartet wird. Daher wird *PVAnwendung* auch als Klientklasse bezeichnet. Die Klasse *PatSelector* ist mit einem graphischen Selektionselement verknüpft, welches hier der Einfachheit halber nicht angegeben wurde. In ihm kann ein Anwender nach einem Patienten suchen, etwa über dessen Namen oder dessen Versichertennummer. Die Klasse *PatDetails* repräsentiert einen graphischen Detailbereich, in dem Informationen zum selektierten Patienten dargestellt werden, etwa Geburtsdatum, Adressdaten sowie die Krankenakte.
Das Quelltextbeispiel an sich ist nicht domänenspezifisch, sondern domänenübergreifend und insofern abstrakt. Die eben benannten Klassen lassen sich zwar aufgrund ihrer sprechenden Namen einem bestimmten Geschäftsprozess zuordnen. Deren Verwendung in der dargestellten Implementierung kann aber auch ohne Weiteres in anderen Geschäftsprozessen erfolgen, weil diese letztendlich nur die Benachrichtigung einer Klasse durch eine andere bei deren Zustandsänderung realisierten.
Der Quelltext aus Abbildung 70 stellt eine Art Vorstufe eines angewandten *Observer*-Musters dar, denn die Implementierung der Lösung entspricht nur in etwa dem, was *Observer* propagiert. Eine Abweichung im gegebenen Quelltext vom *Observer*-Muster ist, dass nur genau ein Beobachter unterstützt wird, anstatt generisch und einheitlich eine beliebige Anzahl an Beobachtern zu ermöglichen. Der Beobachter muss zudem vom Typ *PatDetails* sein, wohingegen das *Observer*-Muster beliebige Klassen erlaubt, wenn sie nur eine entsprechende Schnittstelle umsetzen.
Der gegebene Ausgangsquelltext wird nun durch manuelles Anwenden des *Observer*-Musters umgeformt.

Anwendung des zu dokumentierenden Musters

Observer wird nachfolgend in seiner einfachen Form angewandt (vgl. [Gamma+ 1995]). Der Zustand des Subjekts wird im Weiteren bei der Benachrichtigung der Beobachter direkt an diese weitergegeben. In Phase B (Abschnitt 3.3.4) hingegen findet die Zustandsübergabe über eine dedizierte Methode namens *getState()* statt.
Die Schritte zur Anwendung von *Observer* auf den Quelltext aus Abbildung 70 werden im Folgenden wiedergegeben. Jeder Umformungsschritt stellt eine Transformation dar. Das Ergebnis dieser Umformung ist später in Abbildung 82 zu sehen. Die Zeilen in Abbildung 70, bei denen direkt nach der Zeilennummer Motive in Form von Buchstaben stehen, sind von modifizierenden Transformationen (Änderung, Löschung) betroffen. Die Motive für die Anwendung des *Observer*-Musters werden im folgenden Absatz benannt. Darauf folgen die durch die Motive bedingten Transformationen.

Fachliche Motive der durchzuführenden Transformationen

In Abbildung 70 sind Motive mit den Kleinbuchstaben *a*, *b*, *c*, *d*, *e* und *f* angegeben. Sie stehen für Motive von Transformationen, die die jeweilige Zeile betreffen. Eine Transformation kann mehr als eine Zeile betreffen. Dies ist im Beispiel in Abbildung 70 für einige Motive gegeben, wie an den verschiedenen, mit demselben Motiv gekennzeichneten Zeilen abgelesen werden kann. Die Motive der genannten Abbildung sind zum Überblick in Tabelle 10 zusammengestellt:

Motiv	Bedeutung
a	Behandle Beobachterklassen einheitlich (führe Schnittstelle ein) → gleichbedeutend mit dem Entfernen von Abhängigkeiten zwischen Beobachtern und Subjekt
b	Erlaube es, beliebig viele Beobachter zu registrieren
c	Gestatte Registrierung eines Beobachters zu beliebigem Zeitpunkt
d	Stelle einen einheitlichen Registrierungsmechanismus bereit
e	Subjekt benachrichtigt alle registrierten Beobachter bei Zustandsänderung des Subjekts
f	Bei Benachrichtigung durch Subjekt können Beobachter den Zustand des Subjekts auslesen

Tabelle 10: Fachliche Motive durchzuführender Transformationen.

Jedes Motiv bezieht sich auf die Bedeutung einer Transformation. Die durch die Motive ausgedrückten Bedeutungen spiegeln sich im Zielquelltext wider, der durch Transformation des Ausgangsquelltextes entsteht. Analog zu diesen Motiven, die sich auf durch das Muster herbeigeführte Lösungen beziehen, hätten auf Probleme bezogene Motive benannt werden können. Dies hätte im Weiteren keinen Unterschied gemacht, da eine 1:1 Zuordnung zwischen Lösungs- und Problemmotiv existiert. Dies gilt, wenn man die Semantik einer Aussage über eine Lösung als Synonym für die Aussage über ein Problem ansieht. Das ist hier zulässig, weil ein Anwender des Verfahrens durch Annotation entweder ein Problem ausdrückt was er gelöst haben möchte oder eine Aussage über die Lösung macht, die er durch Musteranwendung realisiert wissen will.

Die genannten Motive können auf zwei verschiedene Wege ermittelt werden. Der erste Weg ist das Interpretieren einer in informaler Form vorliegenden Musterdokumentation (wie etwa in [Gamma+ 1995]). Das Finden von Motiven, die in einer informalen Musterdokumentation explizit oder indirekt genannt sind, ist eine rein intellektuelle Tätigkeit. Im Rahmen dieser Tätigkeit liest ein erfahrener Entwickler die Musterdokumentation, wertet sie aus und extrahiert daraus die Kernaussagen über das Muster. Ergebnis sind die dem Muster zugrunde liegenden Motive. Der zweite Weg ist das Transformieren eines Ausgangsquelltextes in einen Zielquelltext, der weiter unten beschrieben wird. Bei diesem Transformationsprozess ist für jede transformierte Quelltextzeile mindestens ein Motiv zu finden, das die Transformation fachlich begründet.
In dieser Arbeit wurde eine Kombination beider Wege gewählt. Dazu wurde zunächst gemäß dem ersten Weg eine Aufstellung der Motive zum *Observer*-Muster versucht. In einem Iterationsschritt wurde für jede transformierte Zeile geprüft, ob ein geeignetes Motiv existiert um die Transformation zu begründen. War dies nicht der Fall, wurde ein Motiv modifiziert oder neu hinzugefügt.
Die Angabe der Motive wurde vorweggenommen, um einerseits alle Motive anfänglich in einer Übersicht darstellen zu können und zweitens schon vor Ausführung jeder Transformation zum besseren Verständnis den Überblick über deren fachliche Motivation zu haben. Die zu jedem Motiv gehörenden Motivaspekte werden jeweils bei der ersten Verwendung eines Motivs im Rahmen der durchzuführenden Transformationen mit angegeben.

Jedes in natürlicher Sprache formulierte Motiv kann prinzipiell auf viele Arten umgesetzt werden. Von diesen Möglichkeiten sind nur einige sinnvoll und effizient zugleich. Alle anderen denkbaren Möglichkeiten sind entweder nicht sinnvoll, weil sie ästhetischen Ansprüchen oder einer guten Wartbarkeit im Wege stehen. Oder sie sind ineffizient, weil sie unnötige Anweisungen enthalten.

Ein Motiv wird durch Anbringen einer Annotation im Quelltext ausgedrückt, so wie weiter unten in diesem Abschnitt für Transformationen dargestellt. Das Anbringen von Annotationen erst während des Transformationsprozesses ist eine verkürzte Vorgehensweise. Äquivalent dazu, aber umfangreicher in der Prozessbeschreibung, ist das getrennte Anbringen von Annotationen bevor Transformationen auf deren Basis durchgeführt werden. Neben der kürzeren Darstellung gibt es einen weiteren Grund, warum Annotationen nicht initial vor den darauf basierenden Transformationen angebracht werden, sondern währenddessen. Der Grund ist, dass erst während der Transformationsphase gegenwärtig wird, wo sich im Ausgangsquelltext Transformationen auswirken. Weiterhin kann bei einfügenden Transformationen (vgl. Abschnitt 3.3.3.3) erst für den

Zielquelltext eine Annotation angegeben werden, da die mit der Annotation verknüpfte Programmentität im Ausgangsquelltext noch nicht vorhanden ist.

Die in diesem Abschnitt genannten Motive *a* bis *f* lassen teils keinen Spielraum bei der Implementierung, teils eben schon.
Motiv *a* lässt keine qualitativ unterschiedlichen Implementierungsmöglichkeiten zu, da eine einheitliche Behandlung von Klassen nur mit Hilfe einer Schnittstelle oder einer Basisklasse bewerkstelligt werden kann. Die Umsetzung mit Hilfe einer Basisklasse soll hier nicht erachtet werden. Ein Argument für eine Schnittstelle und gegen eine Basisklasse ist die Abstraktheit der Definition, die universell ist und keine Implementierungsdetails vorweg nimmt.
Die Motive *c* und *d* in Kombination können nur mit Hilfe einer Methode als Registrierungsmechanismus umgesetzt werden. Ein Konstruktor wie in Abbildung 70 hingegen erlaubt nur eine Registrierung zum Zeitpunkt der Instanziierung.
Zu den Motiven *b*, *e* und *f* gibt es qualitativ unterschiedliche Implementierungsmöglichkeiten, wovon die offensichtlichsten Varianten aus Sicht des Autors folgende sind:

Motiv *b*: Registrierung der Beobachter – Varianten:
1. ein Aufruf pro Beobachter
2. Übergabe aller Beobachter auf einmal per Liste

Motiv *e*: Durchlaufen aller Beobachter – Varianten:
1. Schleife über alle Beobachter per *Iterator*
2. Schleife über alle Beobachter per *for*-Schleife
3. Verwendung eines Containers, der die Schleife kapselt

Motiv *f*: Zustandsübergabe von Subjekt an Beobachter – Varianten:
1. Zustand des Subjekts wird jedem Beobachter direkt bei Benachrichtigung übergeben
2. Zustand kann von jedem Beobachter bei Bedarf über eigens eingeführte Methode im Subjekt erhalten werden

Motiv *b* beruht auf der Verwaltung der (registrierten) Beobachter durch das Subjekt. Die Verwaltung wiederum kann ebenfalls auf verschiedene Art und Weise implementiert werden, etwa:

1. Speichern der Beobachter in einer Liste, die im Subjekt gehalten wird.
2. Speichern der Beobachter in einer Liste, die in einer separaten Container-Klasse gehalten wird, die vom Subjekt referenziert wird.
3. Speichern der Beobachter in einer Kollektion, die im Subjekt gehalten wird.
4. Speichern der Beobachter in einer Kollektion, die in einer separaten Container-Klasse gehalten wird, die vom Subjekt referenziert wird.

Die Kombinationsmöglichkeiten der genannten Varianten spannen einen Variantenraum auf. Dieser Raum stellt die bekannten unterschiedlichen Möglichkeiten dar, das zugehörige Muster umzusetzen.

Durchzuführende Transformationen

Zu jedem im vorigen Absatz genannten Motiv gehört eine Anzahl an Transformationen, pro Motiv mindestens eine. Um das zu einer Transformation gehörige Motiv auszudrücken, wird eine Annotation im Rahmen jeder Transformation zum transformierten Quelltext hinzugefügt. Das heißt, jede im Quelltext modifizierte Programmentität wird annotiert. Diese Annotation findet während der Transformation statt. Ist ein vorhandener Teil des Ausgangsquelltextes von der Transformation betroffen, wird dieser mit einer Annotation versehen. Dies ist etwa der Fall bei Verschiebungen oder Änderungen, nicht aber bei Neueinfügungen. Wird durch die Transformation etwas hinzugefügt, wird für die hinzugefügte Stelle ebenfalls eine Annotation angefügt. Das ist etwa der Fall bei Neueinfügungen, Verschiebungen oder Änderungen, nicht aber bei Löschungen. Bei Verschiebungen oder Änderungen wird also sowohl der Ausgangs- als auch der durch Transformation resultierende Zielquelltext annotiert.

Pro durch Annotation manifestiertem Motiv werden weiterhin die in Abschnitt 3.2.1 benannten Motivaspekte ermittelt, die am Ende der Phase A zusammengefasst als Ergebnis präsentiert werden.

Während der Durchführung der Transformationen kann ein unkompilierbarer Quelltext entstehen. Dies ist dadurch bedingt, dass nicht vorgegeben ist, in welcher Reihenfolge die Transformationen durchzuführen sind. Hintergrund ist, dass eine Transformation auf einer anderen basieren kann, weil letztere etwa eine Variable einführt, die erstere referenziert. Wichtig ist nur, dass das Ergebnis der Transformation kompilierbar ist. Das sollte immer der Fall sein, da die Forderung an das Ergebnis ist, ein Muster auf einen zuvor schon kompilierbaren Quelltext korrekt angewandt vorzufinden.
Im Folgenden wird die Transformation des Ausgangsquelltextes aus Abbildung 70 durch Anwendung des Musters *Observer* ausgeführt. In der Übersicht in Tabelle 11 sind diese Transformationen samt zugehörigen Motiven benannt:

Transformation	Motive
T1: Schnittstelle *IObserver* hinzufügen	a, e, f
T2: Schnittstelle *IObserver* durch Klasse *PatDetails* implementieren	a
T3: Methode *register* zu Klasse *PatSelector* hinzufügen	a, b, c
T4: Variable *observers* zu Klasse *PatSelector* hinzufügen	b
T5: Bisherige Registrierung von Beobachter *PatDetails* entfernen	a, c, d
T6: Registrierung von Beobachter *PatDetails* hinzufügen	b, c, d
T7: Benachrichtigungsmechanismus für Beobachter ändern	a, b, e, f
T8: Konstruktor von Klasse *PatSelector* ändern	a, c, d
T9: Alten Speichermechanismus für einzelnen Beobachter löschen	d

Tabelle 11: Transformationen in Phase A für das Observer-Muster samt zugehöriger Motive.

Diese Transformationen zur Anwendung von *Observer* folgen nun im Einzelnen.

Transformation T1: Schnittstelle IObserver hinzufügen

Die Schnittstelle *IObserver* aus dem *Observer*-Muster ist von jeder Beobachter-Klasse zu realisieren. Sie ist im gegebenen Beispiel statisch. Eine (oft abstrakte) Basisklasse oder eine Schnittstelle, die im Muster verwendet wird, nicht aber im Ausgangsquelltext vorkommt, kann statisch aus dem Muster übernommen werden. Die Schnittstelle *IObserver* wird wie in Abbildung 71 definiert:

```
499 /**@@motiv a(1): Behandle Beobachterklassen einheitlich*/
500 public interface IObserver {
509   /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
509   /**@@motiv f(3): Übergabe Zustand an Beobachter*/
510   void refresh(Object state);
520 }
```

Abbildung 71: Transformation T1: Schnittstelle IObserver hinzufügen.

Die Zeilennummern entsprechen der letztendlichen Position im Zielquelltext und sind im Prinzip willkürlich wählbar, da deren Zuordnung zum Ausgangsquelltext über Annotationen definiert ist. Die Annotation in Zeile 499 der Abbildung 71 ordnet die transformierte Stelle dem Motiv *a* zu. Das Motiv *a* begründet, warum die zugehörige Transformation T1 durchzuführen ist bzw. durchgeführt wurde. Gleiches gilt für die Motive, die direkt über Zeile 510 stehen. Sie besagen, dass Zeile 510 notwendig ist, weil im Rahmen von *Observer* erstens jeder Beobachter einzeln zu benachrichtigen ist und zweitens bei dieser Benachrichtigung der Zustand des Subjekts an jeden Beobachter übergeben wird. Da zu einem Motiv mehr als eine Transformation gehören kann, wird der Index der Transformation nach dem Motivnamen in Klammern geschrieben, um die Annotationen von einander unterscheiden zu können. Dieses Vorgehen ist analog zur [JSR305]. Die Indizes sind in den angegebenen Transformationen pro Motiv nicht unbedingt chronologisch geordnet, dies ist nur für deren Auftreten

im Zielquelltext weiter unten der Fall. Die Reihenfolge ist allerdings nicht wichtig, weil später bei der Quelltextannotation für die Musterselektion von den Indizes abstrahiert wird.
In Abbildung 71 sind zwei Annotationen für Zeile 510 angegeben, nämlich pro Motiv eine. Da eine Annotation nicht als Programmentität gilt und eine Annotation sich gemäß Definition (siehe Abschnitt 3.2.2) auf die nächste Programmentität bezieht, über der sie steht, beziehen sich die zwei Annotationen auf die Anweisung in Zeile 510.

Für jedes der Motive aus Abbildung 71 wird nun der jeweils dazu gehörende Motivaspekt ermittelt. Eine Zusammenstellung der ermittelten Motivaspekte wird am Ende der Phase A gegeben.
Für Motiv *a(1)* ist der Aspekt »Schnittstellenname« anzugeben, der durch den nach dem Schlüsselwort *interface* stehenden Namen ausgedrückt wird. Motiv *e(3)* besitzt keinen Motivaspekt, da das Motiv de facto durch Aufruf einer Methode, die sich in Java durch genau eine Klasseninstanz darstellt, abgebildet wird. Motiv *f(3)* bezieht sich auf den Zustand des Subjekts. Also kann der Aspekt mit dem Begriff »Zustand« bezeichnet werden. Er bezieht sich in Zeile 510 auf den ersten Eingabeparameter.

Mit der in Abbildung 71 angegebenen Erweiterung des Ausgangsquelltextes um eine Schnittstelle ist die erste Transformation beschrieben.

Die Transformation könnte verfeinernd in zwei feingliedrigere Transformationen aufgespaltet werden. Die erste dieser Transformationen lautet (Abbildung 72):

```
499 /**@@motiv a(1): Behandle Beobachterklassen einheitlich*/
500 public interface IObserver {
520 }
```

Abbildung 72: Transformation T1a: Deklaration der Schnittstelle IObserver.

Die zweite dazu gehörige Transformation kann angegeben werden mit (Abbildung 73):

```
509 /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
509 /**@@motiv f(3): Übergabe Zustand an Beobachter*/
510 void refresh(Object state);
```

Abbildung 73: Transformation T1b: Methode refresh zu Schnittstelle IObserver hinzufügen.

Es ist nicht unbedingt erforderlich, eine Transformation in Unterschritte aufzuteilen. Das kann jedoch bei späteren Definitionen helfen, da feingranulare Transformationen besser wiederverwendet und leichter adaptiert werden können.

Transformation T2: Schnittstelle IObserver durch Klasse PatDetails implementieren

Die über die vorige Transformation eingeführte Schnittstelle *IObserver* kann nun durch Klasse *PatDetails*, die einen Beobachter von Klasse *PatSelector* darstellt, implementiert werden. In einem ersten Schritt wird die Klassensignatur um die Schnittstelle erweitert, wie in Abbildung 74 dargestellt:

```
369 /**@@motiv a(2): Behandle Beobachterklassen einheitlich*/
370 public class PatDetails implements IObserver {
```

Abbildung 74: Transformation T2: Schnittstelle IObserver zu Klasse PatDetails hinzufügen.

Zeile 369 drückt ein Motiv aus, das äquivalent zu dem aus Zeile 499 der Transformation T1 ist. Das ist dadurch zu erklären, dass eine Schnittstelle erstens definiert (Transformation T1) und zweitens implementiert werden muss (Transformation T2).

Der Motivaspekt zur Annotation in Zeile 369 kann mit »Schnittstellenname Motiv a(1)« angegeben werden, weil grundsätzlich eine Beziehung zwischen der Schnittstellendeklaration und deren Implementierung besteht. In der Behandlerroutine zum Motivaspekt kann diese Abhängigkeit durch Vergleichen der annotierten Programmentitäten beider Motivteile ausgewertet werden.

Im gewählten Beispiel ist die Methode *refresh* aus Schnittstelle *IObserver* bereits in Klasse *PatDetails* vorhanden. Sie muss nicht neu eingeführt oder durch Umbenennen einer vorhandenen Methode erzeugt werden. Demnach muss nur die Klassensignatur um die Schnittstellenangabe erweitert und die Einführung der Methode kann übersprungen werden.

Transformation T3: Methode register zu Klasse PatSelector hinzufügen

Um jedem Beobachter seine Registrierung beim Subjekt zu ermöglichen, wird die Methode *register* im Subjekt eingeführt (Abbildung 75):

```
201 /**@@motiv a(3): Behandle Beobachterklassen einheitlich*/
201 /**@@motiv b(1): Registrierungsmöglichkeit vieler Beobachter*/
201 /**@@motiv c(1): Registrierung zu beliebigem Zeitpunkt*/
201 /**@@motiv d(4): Einheitlicher Registrierungsmechanismus*/
202 public void register(IObserver observer) {
204   observers.add(observer);
206 }
```

Abbildung 75: Transformation T3: Methode register zu Klasse PatSelector hinzufügen.

Motiv *a* in der Abbildung korrespondiert mit der über die Transformationen T1 und T2 eingeführten Schnittstelle *IObserver*. Motiv *b* vermerkt zu Methode *register* (Zeile 202), dass diese dazu dient, einem Beobachter eine Registrierungsmöglichkeit beim Subjekt zu bieten. Motiv *c* macht deutlich, dass mit Hilfe der Methode *register* eine Registrierung von Beobachtern zu einem beliebigen Zeitpunkt möglich ist. Im Gegensatz zur Registrierung per Methode bietet die Registrierung von Beobachtern durch deren Übergabe an einen Konstruktor nicht die Möglichkeit der Registrierung zu einem beliebigen Zeitpunkt, da ein Konstruktoraufruf nur einmalig pro Objektinstanz erfolgen kann. Motiv *d* kennzeichnet die Registrierungsmöglichkeit als einheitlich, weil nur über die dedizierte Methode *register* eine dedizierte Registrierung möglich ist.
Der Motivaspekt zu Motiv *a(3)* kann mit »Schnittstellenname Motiv *a(1)*« benannt werden, analog zu Motiv *a(2)*. Für Motiv *b(1)* muss kein expliziter Aspekt eingeführt werden, weil die Methode an sich den Aspekt verkörpert. Gleiches gilt für Motiv *c(1)*, weil eine Methode als Möglichkeit angesehen wird, eine Registrierung zu einem beliebigen Zeitpunkt durchzuführen, im Gegensatz zu einem Konstruktor. Der Aspekt zu Motiv *d(4)* muss analog zu den vorigen beiden Motiven nicht explizit benannt werden, da das Vorhandensein genau einer Methode und zugleich die Nichtexistenz eines Konstruktors zur Registrierung den einheitlichen Registrierungsmechanismus darstellt.

Transformation T4: Variable observers zu Klasse PatSelector hinzufügen

Zur Speicherung von beliebig vielen Beobachtern wird eine neue Variable eingeführt, die eine Liste darstellt, repräsentiert durch den Java-Typ *Vector* (Abbildung 76).

```
114 /**@@motiv a(7): Behandle Beobachterklassen einheitlich*/
114 /**@@motiv b(2): Registrierungsmöglichkeit vieler Beobachter*/
115 private List observers = new Vector();
```

Abbildung 76: Transformation T4: Variable observers zu Klasse PatSelector hinzufügen.

In Zeile 115 wird eine Variable namens *observers* mit dem Typ *List* deklariert. Gleichzeitig (als *Inline* bezeichnet) wird die Variable über den Konstruktoraufruf von *Vector* instanziiert. *Vector* ist eine Java-Klasse, die die Schnittstelle *List* implementiert (genauer: *java.util.List* und *java.util.Vector*).

Der relevante Teil der Programmentität zu Motiv *a(7)* ist durch den Java-Typ *List* gegeben, der, allgemein gesagt, eine Datenkollektion repräsentiert. Demnach kann der Motivaspekt mit »Datenkollektion« angegeben werden. Für Motiv *b(2)* kann derselbe Aspekt angegeben werden, weil nur durch eine Datenkollektion eine (komfortable) Speicherung mehrerer Beobachter möglich ist.

Transformation T5: Bisherige Registrierung von Beobachter PatDetails entfernen

Bisher wurde Klasse *PatDetails* über den Konstruktor der Klasse *PatSelector* registriert. Dies ist im *Observer*-Muster nicht erwünscht, da das Muster erstens einen einheitlichen Registrierungsmechanismus und zweitens die Registrierung zu beliebigem Zeitpunkt fordert. Die bisherige Registrierung über den Konstruktor wird also entfernt, indem die Konstruktion von *PatSelector* ohne Eingabeparameter stattfindet (Motiv *c*, Abbildung 77):

```
059 /**@@motiv a(6): Behandle Beobachterklassen einheitlich*/
059 /**@@motiv c(4): Registrierung zu beliebigem Zeitpunkt*/
059 /**@@motiv d(3): Einheitlicher Registrierungsmechanismus*/
060 PatSelector bo1 = new PatSelector();
```

Abbildung 77: Transformation T5: Bisherige Beobachterregistrierung entfernen.

Motiv *a*, die einheitliche Behandlung von Beobachterklassen, erklärt sich folgendermaßen. Im Ausgangsquelltext wurde ein Eingabeparameter vom Typ *PatDetails* erwartet. Somit war es im Ursprungsentwurf nur möglich, einen Beobachter vom eben genannten Typ zu registrieren. Da diese Beschränkung weggefallen ist (siehe Methode *register* in Transformation T3), wurde Motiv *a* für Transformation T5 vergeben. Gleiches gilt für Motiv *d*, der Forderung nach einem einheitlichen Registrierungsmechanismus. Einheitlich ist ein Registrierungsmechanismus nur dann, wenn er jeden Beobachter gleich behandelt. Das war nicht der Fall, als nur Beobachter vom nicht abstrakten, sondern konkret ausimplementierten Typ *PatDetails* registriert werden konnten. Faktisch hätten zwar Unterklassen von *PatDetails* gebildet werden können. Diese hätten allerdings keine privaten oder finalen Methoden der Superklasse überschreiben können. Generell wäre durch diese Unterklassenbildung ein entarteter Programmentwurf entstanden, da unnötigerweise voll ausimplementierte Methoden hätten überschrieben werden müssen. Das positive Gegenbeispiel hierzu ist die in Transformation T1 eingeführte Schnittstelle *IObserver*.

Für Motiv *a(6)* ist ausschlaggebend, dass im Konstruktor des Subjekts (Klasse *PatSelector*) keine Instanz eines Beobachters übergeben wird. Daher kann der Aspekt mit »Fehlende Beobachterinstanz« benannt werden. Er bezieht sich auf die Parameterliste, die in Zeile 060 der Abbildung 77 leer ist und somit keine Beobachterinstanz enthalten kann. Für Motiv *c(4)* kann analog argumentiert werden, mit dem Zugeständnis, dass die Forderung »Registrierung zu beliebigem Zeitpunkt« nicht den Zeitpunkt der Objekterzeugung des Subjekts mit einschließt. Der Aspekt für Motiv *d(3)* ist analog zu *a(6)* und *c(4)* zu benennen.

Die folgende Transformation fügt einen neuen Registrierungsmechanismus anstelle des eben entfernten ein.

Transformation T6: Registrierung von Beobachter PatDetails hinzufügen

Der Beobachter wird im *Observer*-Muster über Methode *register* bekannt gemacht. Im Beispiel wird somit die Registrierung von *PatDetails* als Beobachter von *PatSelector* über den Konstruktor letztgenannter Klasse durch eine Registrierung mittels der dedizierten Methode *register* ersetzt (Abbildung 78).

```
062 // Registriere Observierer
063 /**@@motiv b(3): Registrierung Beobachter bo2*/
063 /**@@motiv c(2): Registrierung zu beliebigem Zeitpunkt*/
063 /**@@motiv d(1): Einheitlicher Registrierungsmechanismus*/
064 bo1.register(bo2);
```

Abbildung 78: Transformation T6: Registrierung von Beobachterklasse PatDetails hinzufügen.

Zeile 64 in Abbildung 78 wurde neu eingefügt. Damit wird ein Beobachter (Objektinstanz *bo2* der Klasse *PatDetails*) beim Subjekt (Objektinstanz *bo1* der Klasse *PatSelector*) im Sinne des *Observer*-Musters registriert.
Für Motiv b(3) lautet der Aspekt »Übergabe Beobachter (Schnittstelle Motiv a(1))«, der sich auf die Parameterliste der Methode in Zeile 064 bezieht. Der Motivaspekt zu c(2) muss analog zu c(1) nicht angegeben werden, der für d(1) analog zu d(4).

Transformation T7: Benachrichtigungsmechanismus für Beobachter ändern

Der Mechanismus zur Benachrichtigung der registrierten Beobachter wird geändert. Bisher gab es nur einen Beobachter, das *Observer*-Muster unterstützt allerdings beliebig viele. Die neue Benachrichtigungslogik lautet gemäß Abbildung 79:

```
~~340 dependent.refresh(state)~~
341 /**@@motiv a(4): Behandle Beobachterklassen einheitlich*/
341 /**@@motiv e(1): Benachrichtige alle Beobachter*/
342 Iterator it = observers.iterator();while (it.hasNext()) {
343   /**@@motiv a(4/1): Behandle Beobachterklassen einheitlich*/
343   /**@@motiv e(1/1): Benachrichtigung einzelner Beobachter*/
344   IObserver observer = (IObserver)it.next();
345   /**@@motiv a(4/2): Behandle Beobachterklassen einheitlich*/
345   /**@@motiv e(1/2): Benachrichtigung einzelner Beobachter*/
345   /**@@motiv f(1): Übergabe Zustand an einzelnen Beobachter*/
346   observer.refresh(state);
348 }
```

Abbildung 79: Transformation T7: Benachrichtigungsmechanismus für Beobachter ändern.

Zeile 340 wird also faktisch durch die Zeilen 341 bis 348 ersetzt. In Zeile 342 der Abbildung 79 befinden sich zwei Anweisungen. Dies ist eine verkürzte Schreibweise, da sich eine Annotation nur auf die direkt folgende Programmentität beziehen kann. Korrekterweise hätten die beiden Anweisungen in Zeile 342 auf zwei Zeilen aufgeteilt werden und mit einer Blockanweisung (in Java: geschweifte Klammer auf/zu) umgeben werden können. Dann wäre die Blockanweisung mit den Annotationen für Zeile 342 versehen worden.

Mit Hilfe einer Schleife über alle registrierten Beobachter (Zeilen 342 bis 348 in Abbildung 79) werden alle Beobachter des Subjektes benachrichtigt. In Zeile 343 und 345 sind Teilmotive angebracht, um die Zusammensetzung eines Motivs zu demonstrieren und um eine feingliedrigere Definition zu erhalten. Die Benachrichtigung der Beobachter findet über den Aufruf in Zeile 346 statt. Der Aufruf in dieser Zeile stellt aus zwei Gründen eine wichtige Stelle dar. Erstens, weil hier ein Fall dargestellt ist, bei dem sich ein Motiv *f* innerhalb eines anderen Motivs *e* befindet. Dies wirkt sich beim Aufstellen der zur Transformation gehörenden Refactoring-Operationen aus. Zweitens kann Zeile 346 durch ein Teilmotiv von Motiv *e* ausgedrückt werden. Zur Benachrichtigung aller Beobachter ist es nämlich erforderlich, jeden dieser Beobachter einzeln zu benachrichtigen, was als Teilmotiv aufgefasst werden kann.

Der Aspekt zu Motiv *a(4)* kann ähnlich zu *a(7)* angegeben werden mit »Iteration über Datenkollektion«. Für *e(1)* kann derselbe Aspekt angegeben werden, ebenfalls mit Bezug auf die

nichtprimitive Variable *observers*. Die Teilmotive *a(4/1)*, *a(4/2)* sowie *e(1/1)* und *e(1/2)* stellen optionale Verfeinerungen dar, für die hier kein Motivaspekt angegeben wird.
Der Aspekt zu Motiv *f(1)* wird abgebildet durch den Eingabeparameter *state*, der gleich zu *f(3)* mit »Zustand« bezeichnet werden kann.

Transformation T8: Konstruktor von Klasse PatSelector ändern

Aufgrund der Änderung des Registrierungsmechanismus' für Beobachter muss der Konstruktor der Klasse *PatSelector* samt dem Methodenkörper geändert werden (Abbildung 80).

```
179 /**@@motiv a(5): Behandle Beobachterklassen einheitlich*/
179 /**@@motiv c(3): Registrierung zu beliebigem Zeitpunkt*/
179 /**@@motiv d(2): Einheitlicher Registrierungsmechanismus*/
180 public PatSelector() {
190    this.dependent = dependent;
```

Abbildung 80: Transformation T8: Konstruktor von Klasse PatSelector ändern.

Der Konstruktor enthält nun keinen Eingabeparameter mehr. Zuvor war der Eingabeparameter ein Beobachter. Diese Konstruktor-basierte Registrierung ist durch die Forderung nach einem einheitlichen Registrierungsmechanismus nicht mehr aufrecht zu erhalten, da ein Konstruktor pro konstruierter Objektinstanz nur einmal aufgerufen wird. Auch aufgrund der Forderungen des *Observer*-Musters nach einheitlicher Behandlung von Beobachterklassen und nach einer Registrierung der Beobachter zu einem beliebigen Zeitpunkt kann die Änderung begründet werden.
Der Aspekt zu Motiv *a(5)* ist analog zu dem von Motiv *a(6)*, der von *c(3)* analog zu *c(4)* und der von *d(2)* analog zu *d(3)*.

Transformation T9: Alten Speichermechanismus für einzelnen Beobachter löschen

Durch diese Transformation werden nicht mehr benötigte Programmzeilen gelöscht. Dies betrifft die folgenden Zeilen 120 und 130 in Klasse *PatSelector* (Abbildung 81):

```
118 /**@@motiv d(5): Einheitlicher Registrierungsmechanismus*/
120 // Der Observierende
130 private PatDetails dependent;
```

Abbildung 81: Transformation T9: Alten Speichermechanismus für einzelnen Beobachter löschen.

Die gelöschte Zeile 130 enthielt einen Verweis auf den im Ausgangsquelltext einzig registrierten Beobachter, der vom fixen Typ *PatDetails* sein musste. Im Zielquelltext, der gemäß dem *Observer*-Muster gestaltet wird, werden die nun in beliebiger Anzahl zulässigen Beobachter gemäß der Transformation aus Abbildung 76 gehalten.

Zu Motiv *d(5)* kann der Aspekt angegeben werden mit »Keine Referenz auf konkreten Beobachter«, was sich auf die gesamte Subjektklasse (sowie deren Superklassen und referenzierte Hilfsklassen) bezieht.

Ergebnisquelltext nach Transformation

Nach Durchführung aller oben angegebenen Transformationen wird aus dem Ausgangsquelltext (Abbildung 70) der in Abbildung 82 dargestellte Zielquelltext:

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Observierten
040        PatDetails bo2 = new PatDetails();
```

```
050         // Konstruiere Observierer
059         /**@@motiv a(6): Behandle Beobachterklassen einheitlich*/
059         /**@@motiv c(4): Registrierung zu beliebigem Zeitpunkt*/
059         /**@@motiv d(3): Einheitlicher Registrierungsmechanismus*/
060 acd     PatSelector bo1 = new PatSelector();
062         // Registriere Observierer
063         /**@@motiv b(3): Registrierungsmöglichkeit vieler Beobachter*/
063         /**@@motiv c(2): Registrierung zu beliebigem Zeitpunkt*/
063         /**@@motiv d(1): Einheitlicher Registrierungsmechanismus*/
064 bcd     bo1.register(bo2);
070         // Starte Logik des Observierten
080         bo1.start();
090       }
100     }

110     public class PatSelector {
114       /**@@motiv a(7): Behandle Beobachterklassen einheitlich*/
114       /**@@motiv b(2): Registrierungsmöglichkeit vieler Beobachter*/
115 ab    private List observers = new Vector();
140       // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
150       // benötigt).
160       private Object state;

170       // Konstruiere Objekt und registriere einen Observierenden
179       /**@@motiv a(5): Behandle Beobachterklassen einheitlich*/
179       /**@@motiv c(3): Registrierung zu beliebigem Zeitpunkt*/
179       /**@@motiv d(2): Einheitlicher Registrierungsmechanismus*/
180 acd   public PatSelector() {
200       }

201       /**@@motiv a(3): Behandle Beobachterklassen einheitlich*/
201       /**@@motiv b(1): Registrierungsmöglichkeit vieler Beobachter*/
201       /**@@motiv c(1): Registrierung zu beliebigem Zeitpunkt*/
201       /**@@motiv d(4): Einheitlicher Registrierungsmechanismus*/
202 abcd  public void register(IObserver observer) {
204         observers.add(observer);
206       }

210       public void start() {
220         //... tue irgendwas hier ...
230         // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
240         doWorkChangingState();
250         //... tue auch noch irgendwas anderes hier ...
260       }

270       // Diese Methode tut etwas, was den Zustand dieses Objekts signifikant
280       // beeinflusst. Daher wird in ihr der Observierende informiert. Ihm wird
290       // der aktuelle Zustand dieses Objekts mit übergeben, damit er den Zustand
300       // auswerten kann.
310       public void doWorkChangingState() {
320         System.out.println("Ich tue etwas, was meinen Zustand verändert");
330         System.out.println("Deshalb informiere ich alle Observer darüber");
341         /**@@motiv a(4): Behandle Beobachterklassen einheitlich*/
341         /**@@motiv e(1): Benachrichtige alle Beobachter*/
```

```
342 ae     Iterator it = observers.iterator(); while (it.hasNext()) {
343          /**@@motiv a(4/1): Behandle Beobachterklassen einheitlich*/
343          /**@@motiv e(1/1): Benachrichtigung einzelner Beobachter*/
344 ae       IObserver observer = (IObserver)it.next();
345          /**@@motiv a(4/2): Behandle Beobachterklassen einheitlich*/
345          /**@@motiv e(1/2): Benachrichtigung einzelner Beobachter*/
345          /**@@motiv f(1): Übergabe Zustand an einzelnen Beobachter*/
346 aef      observer.refresh(state);
348        }
350      }
360    }

369    /**@@motiv a(2): Behandle Beobachterklassen einheitlich*/
370 a  public class PatDetails implements IObserver {
379      /**@@motiv e(2): Benachrichtigung einzelner Beobachter*/
379      /**@@motiv f(2): Übergabe Zustand an einzelnen Beobachter*/
380 ef   public void refresh(Object state) {
390        // Aktualisiere Zustand und Darstellung
400        // ...
410        System.out.println("Reagiere auf Zustandsänderung von PatSelector");
420      }
430    }

499    /**@@motiv a(1): Behandle Beobachterklassen einheitlich*/
500 a  public interface IObserver {
509      /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
509      /**@@motiv f(3): Übergabe Zustand an einzelnen Beobachter*/
510 ef   void refresh(Object state);
520    }
```

Abbildung 82: Zielquelltext mit angewandtem Observer-Muster in Phase A.

Der Übersichtlichkeit halber sind zu den durch Annotationen ausgedrückten Motiven in Abbildung 82 die Motive zusätzlich links neben jeder annotierten Zeile angegeben. Zu diesem Zielquelltext folgen zunächst Anmerkungen zu den dort enthaltenen Annotationen. Danach werden die Relationen der durch jeweils ein Motiv annotierten Stellen erarbeitet. Anschließend werden aus den durchgeführten Transformationen Refactoring-Operationen gewonnen. Danach sind alle Informationen vorhanden, die zum Aufstellen einer formalen Musterdokumentation erforderlich wird. Die gewonnenen Informationen werden im Phase A abschließenden Abschnitt 3.3.3.5 zusammengestellt.

Annotationen im Zielquelltext

Der Zielquelltext aus Abbildung 82 enthält eine Reihe von Annotationen. Sie wurden überall dort eingeführt, wo eine Transformation vorgenommen wurde. Ist für eine Programmentität keine Annotation vorhanden, kann einer der folgenden Gründe verantwortlich sein:

1. Die Annotation wurde vergessen.
2. Die Annotation ist im Rahmen des gewählten Beispiels nicht erforderlich.
3. Die Programmentität muss nicht annotiert werden, weil sie nicht für das Muster relevant ist.
4. Die Programmentität muss nicht annotiert werden, weil sie auch ohne Annotation automatisch verarbeitet werden kann.

Der erste Fall kann unter Umständen in einer Verfeinerungsphase entdeckt werden. Er kann auch bei Verwendung der Musterdokumentation zum Zweck der Musterselektion oder Musteranwendung

erkannt werden, wenn die Selektion oder Anwendung von der Erwartungshaltung abweicht und Nachforschungen den Fehler zutage bringen.
Der zweite Fall kann in einer Verfeinerungsphase entdeckt werden, wenn die nicht annotierte Programmentität dann relevant wird.
Der dritte Fall sollte nicht auftreten, da durch Transformation veränderte Quelltextteile immer eine Musterrelevanz haben sollte. Ausnahme können Kommentare sein.
Der vierte Fall sollte dadurch vermieden werden, dass alles annotiert wird, was zu annotieren ist. Eine Verfeinerung des Verfahrens, die eine Komforterhöhung bedeutete, könnte diesen Fall zwecks Arbeitserleichterung für den Menschen explizit erlauben. In Abschnitt 3.3.7.1 wird auf automatisch ermittelbare Annotationen weiter eingegangen.
Weiter oben wurde erwähnt, dass im Beispiel zur einfacheren Nachvollziehbarkeit eine Kurzform für Annotationen gewählt wurde. Es wurde gesagt, dass diese Kurzform nachträglich durch eine exakte Schreibweise ersetzt werden kann. Ein Beispiel sei mit der Annotation aus Zeile 63 der Abbildung 82 gegeben. Die dort in Kurzform geschriebene Annotation lautet (Abbildung 83):

```
/**@@motiv b: Registrierung Beobachter bo2*/
```

Abbildung 83: Kurzform einer angebrachten Annotation.

Die Langform der Annotation kann nach Aufstellung des Annotationstyps angegeben werden. Ein geeigneter Annotationstyp kann lauten (Abbildung 84):

```
MID:    4711{("mot", Character, obligatory), ("obs", String)}
HID:    Motiv @mot: Registrierung Beobachter @obs", Deutsch
SCOPES: DEREFERENCED_METHOD_CALL(@bo)
```

Abbildung 84: Definition eines Annotationstyps zu einer gegebenen Annotation.

Die Langform der Annotation lautet unter Bezug auf diesen Annotationstypen wie in Abbildung 85 angegeben:

```
/*
 * @@motiv
 * MID: 4711("b", "bo2")
 * HID: "Motiv @mot: Registrierung Beobachter @obs", Deutsch
 */
```

Abbildung 85: Langform zur angebrachten Annotation.

Relationen zwischen Programmentitäten desselben Motivs

Häufig verursacht ein Motiv in der Musterdokumentation mehrere Transformationen, wie im Beispiel dieses Abschnitts deutlich wurde. In der Musterdokumentation werden die Annotationen zum selben Motiv mit einem fortlaufenden Index versehen. In der Musterselektion hingegen, wo ein Abgleich der Annotationen des zu untersuchenden Quelltextes mit denen der Musterdokumentation stattfindet, können keine Indizes nach dem Motivnamen verwendet werden, da gar nicht bekannt ist, hinsichtlich welches Musters annotiert wird (genau das soll ja im Rahmen der Musterselektion erst entschieden werden). Deshalb ist es in der Musterselektion notwendig herauszufinden, welche Annotation im betrachteten Quelltext (Quelltext-Annotation) mit welcher Annotation zum selben Motiv in einer betrachteten Musterdokumentation passt. Es gibt maximal so viele Möglichkeiten, wie Annotationen zum selben Motiv in der Musterdokumentation vorhanden sind.
Das Finden der richtigen Annotation der Musterdokumentation zur Quelltext-Annotation lässt sich durch verschiedene Überlegungen vereinfachen. Erstens kann der Geltungsbereich der Annotationen zum Motiv herangezogen werden. Passt für die Programmentität, zu der die Quelltext-Annotation gehört, nur ein Geltungsbereich ist die korrespondiere Annotation der Musterdokumentation bereits gefunden. Gibt es mehrere Annotationen in der Musterdokumentation, deren Geltungsbereich zur

genannten Programmentität passt, sind eben diese Kandidaten für eine Korrespondenz zur Quelltext-Annotation (aus diesen kann aber nur genau eine relevant sein, sofern kein Fehler beim Annotieren des Quelltextes gemacht wurde).
Ausgehend von der zuletzt geschilderten Mehrdeutigkeit kann eine korrespondiere Annotation dadurch gefunden werden, dass Relationen zwischen den annotierten Programmentitäten ermittelt werden. Nach Betrachtung des Ausgangsquelltextes im Beispiel (Abbildung 70) fällt auf, dass es keine Mehrdeutigkeiten gibt. Beispielsweise sind alle Programmentitäten für Motiv *a* im Geltungsbereich von einander unterscheidbar, wie die folgende Aufstellung der betroffenen Zeilen samt der Art der Programmentität verdeutlicht:

- Zeile 060: Konstruktoraufruf
- Zeile 130: Deklaration
- Zeile 180: Konstruktordeklaration
- Zeile 190: Variablenzuweisung
- Zeile 340: Methodenaufruf
- Zeile 370: Klassendeklaration

Alle diese Arten von Programmentitäten sind durch Analyse des abstrakten Syntaxbaums eindeutig von einander unterscheidbar, weil sie vom Compiler als unterschiedliche Typen im Syntaxbaum repräsentiert werden. Weil auch für die anderen Annotationen im Beispiel keine Mehrdeutigkeit existiert, ist davon auszugehen dass diese häufig nicht gegeben ist[38]. Dies kann auch deshalb angenommen werden, weil die Transformationen eines Motivs abhängige Programmentitäten betreffen. Wie in Abschnitt 3.2.1 dargelegt (vgl. auch Abbildung 34), sind die abhängigen Programmentitäten überwiegend von einem unterscheidbaren Typ. Eine Ausnahme stellen Methoden dar. Diese ergibt sich daraus, dass es für das beobachtbare Verhalten eines Programms unerheblich ist, ob eine Logik innerhalb einer einzigen Methode ausgeführt wird, oder ob diese Methode intern eine weitere Hilfsmethode aufruft. Nun könnten sowohl die erste Methode als auch die Hilfsmethode zum selben Motiv gehören. Allerdings kann für Methoden derselben Klasse gefordert werden, dass nur die erste, übergeordnete Methode als relevant für die Annotation anzusehen ist, da diese die gesamte Logik einschließt. Für alle Muster bis auf die, wo gerade die Indirektion, die durch das Aufrufen einer Methode aus einer anderen Methode derselben Klasse entstehen, ist diese Verfahrensweise möglich. Eine Methode aus Klasse K1, die eine Methode aus Klasse K2 aufruft (so wie in Abbildung 34) ist unterscheidbar durch die unterschiedliche Klasse, in der die jeweilige Methode definiert ist. Je nachdem, welche weiteren Charakteristika in K1 und K2 durch Analyse von eventuell vorhandenen Annotationen sowie abstrakten Syntaxbäumen bestimmt werden können, kann eine Unterscheidung der Methoden über die Charakteristika der jeweils zugehörigen Klasse herbeigeführt werden. Gleiches gilt für die in Abbildung 34 dargestellten Variablendeklarationen, von denen jeweils eine in den beiden dargestellten Klassen zu sehen ist. Es kann die Variablendeklaration als zum Motiv gehörend angesehen werden, die in Abhängigkeit zur Methode steht, welche die Variable anspricht. Andererseits ist es denkbar, beide Variablendeklarationen zu betrachten und zu prüfen, welche davon besser mit einer Annotation aus der Musterdokumentation korrespondiert. Letzteres kann durch Hinzuziehen der automatisiert ableitbaren Abhängigkeiten zur annotierten Programmentität unterstützt werden.

Sollen explizit Abhängigkeiten in der Musterdokumentation zwischen annotierten Programmentitäten desselben Motivs definiert werden, kann dies über Prüflogiken in reinem Java-Code bewerkstelligt werden. Eine Tabelle verzeichnet weiterhin, welche Prüflogik für welche

[38] Vgl. auch Anhang A und B: Für das *Singleton* existiert eine offensichtliche Eindeutigkeit der Geltungsbereiche pro Motiv. Für das *Composite* sind zwar für manche Motive (etwa *b*) gleiche Geltungsbereiche vorhanden; diese beziehen sich aber auf jeweils eine Programmentität die beliebig oft vorkommen kann und somit generell nicht eindeutig ist (beispielsweise sind dies beliebig viele verschiedene Klassen, die mögliche geometrische Figuren auf einem Bild repräsentieren).

Programmentität (die durch eine Annotation zu einem Motiv samt einem eindeutigen Index gekennzeichnet ist) zu ziehen ist. Diese Definition expliziter Abhängigkeiten wurde im Beispiel nicht vorgenommen. Der erste Grund ist, dass es sich als nicht erforderlich herausgestellt hat (siehe weiter oben die Typen von Programmentitäten zu Motiv *a*, die allesamt von einander unterscheidbar sind). Der zweite Grund ist die Komplexität der Umsetzung einer solchen Prüflogik und somit deren Benennung an dieser Stelle. Letztendlich handelt es sich „nur“ um einen Quelltextanalysator, der abstrakte Syntaxbäume und bei Bedarf zur Verfeinerung Annotationen untersucht. Hier ist es denkbar, eine Bibliothek bzw. ein Framework aufzubauen, womit zukünftige Prüflogiken einfach erstellt und über die Musterdokumentation in den Prozess der Musterselektion eingebunden werden können.

Als letzter wesentlicher Schritt zur Gewinnung einer formalen Musterdokumentation ist das Aufstellen von Refactoring-Operationen erforderlich. Dies wird im nächsten Abschnitt beschrieben.

Refactoring-Operationen zu Transformationen

Der weiter oben gegebene Zielquelltext aus Abbildung 82 entstand durch Ausführung der davor genannten Transformationen. Jede Transformation bedingt gemäß Abbildung 31 eine oder mehrere Refactoring-Operationen. Eine Refactoring-Operation ist ein eindeutig definierter, maschinell durchführbarer Transformationsschritt. Im Gegensatz dazu ist eine Transformation für gewöhnlich nur manuell und nicht maschinell durchführbar, weil eine Transformation aus einer Motivation heraus geschieht, eine Refactoring-Operation aber fest vorgegeben ist. Refactoring-Operationen werden aus Transformationen abgeleitet. Dieser Schritt ist notwendig, um eine Maschinenanwendbarkeit einer Transformation zu erhalten. Die genannte Ableitung ist möglich, wenn eine Vorschrift existiert, die einer Transformation eine Sequenz von Refactoring-Operationen zuordnet. Die Identifikation einer Überführungsvorschrift einer Transformation in eine oder mehrere Refactoring-Operationen kann durch Analyse von abstrakten Syntaxbäumen stattfinden. Mit Hilfe einer solchen Analyse kann für einen Ausgangsquelltext einer Transformation als auch für den daraus resultierenden Zielquelltext zweifelsfrei festgestellt werden, welcher Typ von Programmentität von der Transformation betroffen war. Ein solcher Typ kann beispielsweise eine Schnittstelle, eine Methode, ein Methodenparameter oder ein Anweisungsteil sein. Durch Kennzeichnung mit Annotationen kann im Fall einer Modifikationstransformation zweifelsfrei markiert werden, welcher Typ einer Programmentität in welchen Typ durch Transformation überführt wurde. Annotationen helfen auch festzustellen, ob eine Einfügung, eine Löschung oder eine Verschiebung stattgefunden hat.

Zur Veranschaulichung werden für die im Abschnitt weiter oben genannten neun Transformationen die jeweils zugehörigen Refactoring-Operationen beschrieben. Jede im Folgenden dargestellte Refactoring-Operation wurde durch Nachdenken aus je einer Transformation ermittelt. Dieser intellektuelle Prozess kann für zukünftige Transformationen, die vergleichbar den folgenden sind, maschinell unterstützt werden. Dazu kann als Basis für ein Vorschlagswesen für Refactoring-Operationen beispielsweise die Ähnlichkeit einer neuen Transformation mit einer bereits bestehenden ermittelt werden (siehe weiter unten).

Refactoring-Operationen für T1 (Schnittstelle IObserver hinzufügen)

Die zur Transformation T1 zugehörige Refactoring-Operation wird an dieser Stelle (willkürlich) mit *Introduce_Interface* benannt. Die Transformation entspricht hier dieser einen Refactoring-Operation, weil eine neue Entität, nämlich eine Schnittstelle angelegt wird. Wäre die Schnittstelle im zu transformierenden Ausgangsquelltext schon vorhanden, müsste eine andere Refactoring-Operation verwendet werden. Diese andere Operation würde die bestehende Schnittstelle umformen in die gewünschte. Das könnte beispielsweise durch Umbenennen einer bereits in der Schnittstelle deklarierten Methode geschehen, falls die Methode qualitativ dasselbe tut wie die benötigte Methode, aber einen anderen Namen trägt.

Zusammenfassend, kann die Refactoring-Operation *Introduce_Interface* wie Abbildung 86 angegeben werden:

```
Name: Introduce_Interface @interface
Eingabeparameter:
   @interface: Einzuführende Schnittstelle
Vorbedingungen:
   Schnittstelle @interface noch nicht vorhanden
Ausgangsannotation: fehlt
Zielannotation: "motiv a(1)" (Zeile 499)
Operation: Hinzufügen Klasse, die Schnittstelle @interface einführt
```

Abbildung 86: Refactoring-Operation Introduce_Interface.

Ein im Ausgangsquelltext vorhandener abstrakter Syntaxbaum als Vorbedingung für die Anwendung der Refactoring-Operation muss in Abbildung 86 nicht angegeben werden, weil mit *Introduce_Interface* ein Programmteil neu eingeführt wird. Die sprachlich formulierte Vorbedingung, dass die einzuführende Schnittstelle noch nicht vorhanden sein darf, wird in der Praxis durch eine Prüflogik ersetzt. Diese Prüflogik kann eine Java-Routine sein, die für einen gegebenen Quelltext feststellen kann, ob eine Schnittstelle vorhanden ist oder nicht. Derartige Prüfroutinen können bei Bedarf ergänzt werden und wachsen so im Lauf der Zeit zu einer mächtigen Bibliothek heran. Generell gilt, dass in Refactoring-Operationen genannte Vorbedingungen nur als Versuch einer möglichst umfassenden Behandlung relevanter Fälle anzusehen sind. Es ist in vielen Fällen nicht ohne Weiteres möglich, alle signifikanten oder gar alle nötigen Vorbedingungen anzugeben (vgl. auch [Ó Cinnéide+ 1999b]).

Für diese Refactoring-Operation *Introduce_Interface* soll im Folgenden beschrieben werden, wie sie aus einer vorliegenden Transformation bei zukünftigen Musterdokumentationen automatisch ermittelt werden kann. Ausgangs- und Zielannotation sind dazu mit einander zu vergleichen. Der in Abbildung 86 vorliegende Fall, dass die Ausgangsannotation fehlt und eine Zielannotation angegeben ist, ist einfach zu interpretieren: Bei der durchgeführten Transformation handelt es sich um eine Einfügeoperation. Eine Transformation, die eine solche Konstellation von Ausgangs- und Zielannotation vorweist ist ein potentieller Kandidat für die genannte Refactoring-Operation. Zur Feststellung, welcher Typ von Programmentität eingefügt wurde, kann der Geltungsbereich der Zielannotation verwendet werden (vgl. Abschnitt 3.2.2).

Refactoring-Operationen für T2 (Schnittstelle IObserver zu Klasse PatDetails hinzufügen)

Im Gegensatz zur vorigen Transformation, ist das Implementieren einer Schnittstelle durch eine Klasse eine Modifikation einer bestehenden Entität. Weiterhin besteht diese eine Transformation möglicherweise aus mehreren Refactoring-Operationen.
Die erste Refactoring-Operation ist die Erweiterung der Klassensignatur um die zu implementierende Schnittstelle (Abbildung 87):

```
Name: Add_Interface_To_Signature @interface @class
Eingabeparameter:
   @interface: Zu implementierende Schnittstelle
   @class:     Klasse, die Schnittstelle implementieren soll
Vorbedingungen:
   Schnittstelle @interface vorhanden
   Klasse @class vorhanden
   Klasse @class deklariert Schnittstelle @interface noch nicht
Ausgangsannotation: "motiv a(2)" (Zeile 369)
Zielannotation: "motiv a(2)" (Zeile 369)
Operation: Signatur von Klasse @class um Schnittstelle @interface
```

Abbildung 87: Refactoring-Operation Add_Interface_To_Signature.

Die zweite Operation ist die Implementierung aller in der Schnittstelle angegebenen Methoden. Für jede zu implementierende Methode wird eine Refactoring-Operation durchgeführt, um die Methode mit entsprechender Signatur und leerem Körper anzulegen. Eine weitere Refactoring-Operation fügt Quelltext in eine neu angelegte Methode hinzu. Dies beides ist jedoch im gegebenen Beispiel nicht notwendig, da die Klasse *PatDetails* bereits die Methode *register* implementiert.

Die Refactoring-Operation aus Abbildung 87 kann wie folgt aus Transformationen abgeleitet werden. Zunächst ist festzustellen, dass Ausgangs- und Zielannotation gleich sind. Das bedeutet, es handelt sich nicht um das Einfügen einer neuen oder das Entfernen einer vorhandenen Klasse. Vielmehr wird eine bestehende Klasse modifiziert. Die Modifikation besteht daraus, zu einer Klassensignatur eine Schnittstellenangabe hinzufügen. Das Hinzufügen einer Schnittstellenangabe kann durch Vergleich von abstrakten Syntaxbäumen (vorher/nachher) festgestellt werden.

Refactoring-Operationen für T3 (Methode register zu Klasse PatSelector hinzufügen)

Um es Beobachtern zu erlauben, beim Subjekt registriert zu werden, wird in Klasse *PatSelector* die Methode *register* neu eingefügt. Die erste zu dieser Einfügetransformation gehörige Refactoring-Operation ist in Abbildung 88 dargestellt:

```
Name: Add_Interface_Method @interface @class @method
Eingabeparameter:
   @interface: Zu implementierende Schnittstelle
   @class:     Klasse, die Methode der Schnittstelle implementieren soll
   @method:    Zu implementierende Methode
Vorbedingungen:
   Schnittstelle @interface vorhanden
   Klasse @class implementiert Methode @method von Schnittstelle
              @interface noch nicht
Ausgangsannotationen: keine
Zielannotation: "motiv a(3)" (Zeile 201)
Operation: Methode @method in Klasse @class mit Signatur gemäß
           Schnittstelle @interface und mit leerem Körper anlegen
```

Abbildung 88: Refactoring-Operation Add_Interface_Method.

Die Operation aus Abbildung 88 legt zunächst eine leere Methode an. Die folgende Refactoring-Operation fügt eine entsprechende Logik zu dieser bisher noch leeren Methode *register* hinzu (Abbildung 89).

```
Name: Fill_Method_With_Code @class @method @code
Eingabeparameter:
   @class:     Klasse, die Methode enthält
   @method:    Methode, die mit Logik befüllt werden soll
   @code:      Logik (Quelltext) für die Methode
Vorbedingungen:
   Methode @method in Klasse @class vorhanden
Ausgangsannotation: "motiv a(3)"
Zielannotation: "motiv a(3)"
Operation: Methodenkörper mit Quelltext @code besetzen
```

Abbildung 89: Refactoring-Operation Fill_Method_With_Code.

Die im Parameter *@code* anzugebende Logik ist bekannt durch die in Abbildung 75 angegebene Transformation. Sie lautet demnach (Abbildung 90):

```
observers.add(observer);
```

Abbildung 90: Quelltextparameter für Refactoring-Operation Fill_Method_With_Code.

Dass es sich hier nur um eine Zeile handelt, ist nicht als Einschränkung zu verstehen. Prinzipiell können beliebig viele Quelltextzeilen in eine Refactoring-Methode wie die o.g. eingegeben werden.
Es ist zu beachten, dass in Abbildung 89 die Ausgangsannotation konkret angegeben wurde, während sie in Abbildung 88 noch fehlte. Der Grund ist, dass nach Anwendung der ersten Operation (Abbildung 88) die dort angegebene Zielannotation zur Ausgangsannotation für die darauf folgende, verkettet ausgeführte Operation (Abbildung 89) wird.
Weiterhin ist anzumerken, dass Zeile 202 des Zielquelltextes mit drei verschiedenen Annotationen versehen ist, weil drei Motive für die zugrunde liegende Transformation des Ausgangsquelltextes existieren. Letztendlich ist es egal, welche Annotation als Bezug in Abbildung 88 verwendet wird, sofern die Annotationen sowohl im Ausgangs- als auch im Zielquelltext bezüglich der Refactoring-Operation analog vorhanden sind. Dies ist für die drei Annotationen der Fall, weil alle drei sowohl im Ausgangsquelltext in Zeile 201 fehlen als auch im Zielquelltext in Zeile 201 neu hinzugekommen sind. Wichtig ist nur, dass für jedes Motiv erkannt wird, dass die Operation auszuführen ist. Dies wiederum kann abgeleitet werden, weil die transformierte Zeile von drei Motiven abhängig war.

Refactoring-Operationen für T4 (Variable observers zu Klasse PatSelector hinzufügen)

Die beim Subjekt registrierten Beobachter werden in einer neu einzuführenden Variablen namens *observers* gehalten. Diese Variable stellt eine Liste dar. Sie wird durch die Operation in Abbildung 91 eingeführt.

```
Name: Add_Declaration_in_Class @class @code
Eingabeparameter:
   @class:     Klasse, in die Deklaration einzufügen ist
   @code:      Quelltext, der die Deklaration enthält
Vorbedingungen:
   Klasse @class vorhanden
   Deklaration aus Quelltext @code nicht in Klasse @class vorhanden
Ausgangsannotation: keine
Zielannotation: "motiv b(2)" (Zeile 114)
Operation: Quelltext @code in Klasse @class einfügen
```

Abbildung 91: Refactoring-Operation Add_Declaration_in_Class.

Gemäß der zugehörigen Transformation besitzt der Parameter *@code* folgenden Inhalt (Abbildung 92):

```
private List observers = new Vector();
```

Abbildung 92: Eingabequelltext für Refactoring-Operation Add_Declaration_in_Class.

Für zukünftige, mit der Ursprungstransaktion vergleichbare Transformationen kann die Anwendbarkeit dieser Refactoring-Operation wie folgt maschinell gefolgert werden. Weil keine Ausgangs- zur Zielannotation existiert, handelt es sich um eine Einfügeoperation. Der Bezug der Zielannotation ist eine Variablendeklaration auf Klassenebene. Es muss also geprüft werden, ob durch eine Transformation eine solche Variablendeklaration hinzugefügt wurde. Ist das der Fall, kann die Transformation mit der Refactoring-Operation aus Abbildung 91 verknüpft werden.

Refactoring-Operationen für T5 (bisherige Registrierung von Beobachter PatDetails entfernen)

Die bisherige Konstruktion von Klasse *PatSelector* muss gemäß Transformation ersetzt werden.
Zunächst wird dazu der Konstruktoraufruf von *PatSelector* geändert. Dies geschieht mit folgender Refactoring-Operation, die für Klasse *PVAnwendung* ausgeführt wird (Abbildung 93):

```
Name: Modify_Instantiation @class @line @code
Eingabeparameter:
   @class:      Klasse, in der Instanziierung zu ändern ist
   @line:       Zeile, die Objektinstanziierung enthält
   @param_list: Liste der Eingabeparameter für zu rufenden Konstruktor
Vorbedingungen:
   Klasse @class vorhanden
   Zeile @line enthält genau einen Konstruktoraufruf
   Aufgerufener Konstruktor mit Parametertypen aus @param_list existiert
Ausgangsannotation: "motiv c(4)" (Zeile 59)
Zielannotation: "motiv c(4)" (Zeile 59)
Operation: Konstruktoraufruf anpassen an Verwendung der Parameter aus
           @param_list
```

Abbildung 93: Refactoring-Operation Modify_Instantiation.

Die Parameter der Operation haben für das Beispiel aus Abbildung 70 folgende Werte:

- *@class*: PVAnwendung
- *@line*: 60
- *@param_list*: leer

Somit wird aus einem Konstruktoraufruf mit einem Parameter ein Konstruktoraufruf ohne Parameter.

Refactoring-Operationen für T6 (Registrierung von Beobachter PatDetails hinzufügen)

Nach Entfernen der bisherigen Konstruktor-basierten Registrierung wird eine neue Zeile in Klasse *PVAnwendung* eingefügt, um den Beobachter beim Subjekt gemäß dem *Observer*-Muster zu registrieren (Abbildung 94):

```
Name: Insert_Line @class @line @code
Eingabeparameter:
   @class:     Klasse, in der Zeile einzufügen ist
   @line:      Zeile, vor der Quelltextzeile einzufügen ist
   @code:      Quelltextzeile, die einzufügen ist
Vorbedingungen:
   Klasse @class vorhanden
   Klasse @class besitzt mindestens @line Zeilen, beginnend bei 1
Ausgangsannotation: keine
Zielannotation: "motiv c(2)" (Zeile 63)
Operation: Quelltext @code in Klasse @class vor Zeile @line einfügen
```

Abbildung 94: Refactoring-Operation Insert_Line.

Die einzufügende Zeile ist die im Zielquelltext in Abbildung 82 dargestellte Zeile 064. Der Parameter *@line* bekommt also den Wert 070 (gemäß der Zählweise im Ausgangsquelltext in Abbildung 70). Der Parameter *@code* ist gleich dem Inhalt der Zeile 063 aus Abbildung 82.
Diese Refactoring-Operation namens *Insert_Line* kann generisch verwendet werden, etwa auch anstatt der im vorigen Abschnitt benannten Operation *Add_Declaration_in_Class*. Der Vorteil letztgenannter Operation ist allerdings, dass sie spezialisierter ist und somit Vorbedingungen spezifischer geprüft werden können. Außerdem ist die Verarbeitung der Operation in diesem Fall einfacher möglich als mit der generischen Variante. Der Grund ist, dass bei *Add_Declaration_in_Class* eine einzufügende Deklaration auf Klassenebene an jeder beliebigen Stelle einer Klasse außerhalb von Code-Blöcken und Methoden eingefügt werden kann. Dahingegen muss bei *Insert_Line* eine Einfügestelle explizit angegeben werden. Der konkrete Aufruf von *Add_Declaration_in_Class* ist also einfacher als von *Insert_Line*.

Die mit *Insert_Line* in Klasse *PVAnwendung* einzufügende Zeile lautet gemäß Transformation (Abbildung 95):

```
bo1.register(bo2);
```

Abbildung 95: Mit Insert_Line einzufügende Quelltextzeile in Klasse PVAnwendung.

Damit ist die bisherige Konstruktor-basierte Registrierung des Beobachters beim Subjekt durch eine *Observer*-konforme ersetzt worden.

Refactoring-Operationen für T7 (Benachrichtigungsmechanismus für Beobachter ändern)

Im Ausgangsquelltext war die Benachrichtigung nur eines Beobachters möglich. Das *Observer*-Muster soll aber beliebig viele Beobachter unterstützen. Deshalb muss der bisherige Mechanismus zur Benachrichtigung der registrierten Beobachter geändert werden. Die Refactoring-Operation besteht darin, die Benachrichtigungslogik in Methode *doWorkChangingState* zu ersetzen. Die Operation in Abbildung 96 ersetzt eine Zeile durch eine oder mehrere andere Zeilen.

```
Name: Replace_Line @class @line @code
Eingabeparameter:
   @class:     Klasse, in der Zeile zu ersetzen ist
   @line:      Zeile, die zu ersetzen ist
   @code:      Quelltext, der einzusetzen ist
Vorbedingungen:
   Klasse @class vorhanden
   Klasse @class besitzt mindestens @line Zeilen, beginnend bei 1
Ausgangsannotation: "motiv e(1)" (Zeile 341)
Zielannotation: "motiv e(1)" (Zeile 341)
Operation: Zeile @line in Klasse @class durch Quelltext @code ersetzen
```

Abbildung 96: Refactoring-Operation Replace_Line.

Durch Anwendung der Operation *Replace_Line* wird die folgende Zeile ersetzt (Abbildung 97):

```
340     dependent.refresh(state);
```

Abbildung 97: Durch Operation Replace_Line ersetzte Zeile.

Die stattdessen einzusetzenden Zeilen lauten (Abbildung 98):

```
342     Iterator it = observers.iterator();while (it.hasNext()) {
344       IObserver observer = (IObserver)it.next();
346       observer.refresh(state);
348     }
```

Abbildung 98: Durch Operation Replace_Line einzusetzende Zeilen.

Wie in der zugehörigen Transformation aus Abbildung 79 erwähnt, ist in Motiv *e* das Motiv *f* enthalten (Zeile 346). Die Transformation in Abbildung 98 ist demnach noch nicht unbedingt vollständig. Zeile 346 muss zu Motiv *f* (Übergabe Zustand an einzelnen Beobachter) passen. Dies ist schon der Fall, da Zeile 346 bereits eine Zustandsvariable namens *state* an den Beobachter übergibt. In Abschnitt 3.3.4 wird ein Beispiel gegeben, in dem die Sachlage anders ist und eine durch ein Motiv innerhalb eines anderen Motivs bedingte entsprechende Ersetzung notwendig ist.

Somit wurde die Benachrichtigung genau eines konkreten Beobachters für den Ausgangsquelltext aus Abbildung 70 mit Hilfe der Refactoring-Operation *Replace_Line* durch einen generischen Mechanismus ersetzt.

Refactoring-Operationen für T8 (Konstruktor von Klasse PatSelector ändern)

Aufgrund der Änderung des Registrierungsmechanismus' für Beobachter muss der Konstruktor der Klasse *PatSelector* geändert werden. Über den Konstruktor soll keine Beobachterregistrierung mehr stattfinden, daher soll der entsprechende Eingabeparameter des Konstruktors entfernt werden. Die Refactoring-Operation, mit der dies geschehen kann, ist in Abbildung 99 angegeben:

```
Name: Remove_Input_Parameter @class @method @paramname
Eingabeparameter:
   @class:     Klasse, in der Methode @method sich befindet
   @method:    Methode, in der Eingabeparameter zu entfernen ist
   @paramname: Name des zu entfernenden Parameters der Methode @method
Vorbedingungen:
   Klasse @class vorhanden
   Methode @method in Klasse @class vorhanden
   Parameter @paramname existiert für Methode @method
Ausgangsannotation: "motiv c(3)" (Zeile 179)
Zielannotation: "motiv c(3)" (Zeile 179)
Operation: Parameter @paramname der Methode @method in Klasse @class
           entfernen
```

Abbildung 99: Refactoring-Operation Remove_Input_Parameter.

Die Refactoring-Operation *Remove_Input_Parameter* ist insofern generisch, als dass sie sich sowohl auf gewöhnliche Methoden bezieht als auch auf Konstruktoren (letztere können als spezielle Methoden aufgefasst werden).
Da der Konstruktor von Klasse *PatSelector* nur einen Parameter besitzt, wird nach Anwendung von Operation *Remove_Input_Parameter* der einzige Parameter entfernt. Übrig bleibt ein Konstruktor ohne Parameter.

Refactoring-Operationen für T9 (alten Speichermechanismus für einzelnen Beobachter löschen)

Die zugehörige Transformation löscht zwei Programmzeilen aus Klasse *PatSelector*, nämlich:

```
120     // Der Observierende
130     private PatDetails dependent;
```

Abbildung 100: In Klasse PatSelector zu löschende Zeile.

Eine Refactoring-Operation, die genau eine Zeile löscht, kann wie in Abbildung 101 angegeben werden:

```
Name: Remove_Line @class @line
Eingabeparameter:
   @class:     Klasse, in der Zeile zu entfernen ist
   @line:      Zeile, die zu entfernen ist
Vorbedingungen:
   Klasse @class vorhanden
   Klasse @class beinhaltet Zeile @line
Ausgangsannotation: "motiv b"
Zielannotation: Keine
Operation: Zeile @line in Klasse @class entfernen
```

Abbildung 101: Refactoring-Operation Remove_Line.

Die Operation aus Abbildung 101 muss zum Löschen der in Abbildung 100 angegebenen zwei Zeilen demzufolge zweimal aufgerufen werden. Anstelle dessen wäre auch eine Refactoring-Operation *Remove_Lines* denkbar, die es gestattet, mehr als eine Zeile auf einmal zu entfernen.
Anzumerken ist, dass in Abbildung 101 die Ausgangsannotation mit Motiv *b* angegeben wurde. Motiv *b* lautet gemäß Abschnitt 3.3.3.4 »Erlaube es, beliebig viele Beobachter zu registrieren«. Die Ausgangsannotation ist nicht im Ausgangsquelltext aus Abbildung 70 vorhanden, wird aber in der Transformation eingeführt. Es ist so zu sehen, dass diese Annotation im zu transformierenden Quelltext anzubringen ist, bevor die Refactoring-Operation *Remove_Line* ausgeführt wird. Dies ist

deshalb sinnvoll, weil das Erkennen durchzuführender Refactoring-Operationen für spätere, unbekannte Quelltexte analog geschieht (nämlich: zuerst einen Quelltext annotieren, dann über anwendbare Refactoring-Operationen entscheiden).

Damit ist die formale Dokumentation des *Observer*-Musters mit Hilfe des vorgestellten Verfahrens im Grundsatz abgeschlossen. Der nächste Abschnitt diskutiert die hieraus gewonnenen Ergebnisse, bevor anschließend eine Verfeinerung der formalen Dokumentation des *Observer*-Musters vorgenommen wird.

3.3.3.5 Diskussion der bisherigen Ergebnisse

Bis hierhin wurde die Phase A der formalen Musterdokumentation beschrieben. Als Beispiel zur Illustration wurde das *Observer*-Muster gewählt. Wie anfangs der Phase dargelegt, ist das Verfahren prinzipiell geeignet für jedes Entwurfsmuster sowie alle verwandten Musterarten, die auf Quelltext basieren (wie etwa Architekturmuster, nicht aber Analysemuster). Die Phase A beinhaltete

- das Auswählen eines zu dokumentierenden Musters M,
- das Auswählen eines Ausgangsquelltextes Q_a, auf den M anwendbar ist,
- das Anwenden von M auf Q_a durch Quelltexttransformationen sowie
- das Aufstellen von Refactoring-Operationen zu jeder Quelltexttransformation.

Die durch die Ausführung der dargestellten Transformationsschritte gewonnenen Informationen sind

- ein Zielquelltext Q_z, der Q_a samt angewandtem Muster M darstellt,
- Korrelierte Programmteile in Q_a und Q_z, insbesondere die durch Anwendung von M
    - hinzugekommenen,
    - modifizierten und
    - weggefallenen Teile,
- *Observer*-basierte Motive für Transformationen,
- pro Motiv die Relationen der damit annotierten Programmentitäten,
- pro zwischen Q_a und Q_z unterschiedlichem Teil die zugehörige Transformation,
- pro Transformation die zugehörigen Motive,
- pro Transformation die zugehörigen Refactoring-Operationen,
- pro Refactoring-Operation die dafür notwendigen Vorbedingungen sowie die Logik zur Ausführung der Operation,
- statische und dynamische Musterteile.

Die gewonnenen Informationen zu Motiven, Annotationen, Transformationen und Refactoring-Operationen werden in den nachfolgenden Abschnitten abschließend zusammengefasst.

Ermittelte Motive und Motivaspekte

Zum *Observer*-Muster wurden die in Abschnitt 3.3.3.4 genannten Motive ermittelt. Motive zu finden ist nur durch intellektuelle Tätigkeit möglich. Dieser Prozess kann durch informale Hilfestellungen unterstützt werden. Eine solche Hilfestellung ist etwa das Untersuchen der Korreliertheit zweier Motive.
Das Aufstellen von Motivaspekten löste den Zwang aus, die für das Motiv signifikanten Teile einer annotierten Programmentität zu benennen. Die zuvor entwickelten Motivaspekte sind in der folgenden Tabelle 12 zusammengestellt.

Motiv	Aspekt	Zeile	Element
a(1)	Schnittstellenname	500	Name Schnittstelle
a(2)	Schnittstellenname Motiv a(1)	370	Name Schnittstelle
a(3)	Schnittstellenname Motiv a(1)	202	Name Schnittstelle
a(4)	Iteration über Datenkollektion	342	Variable (nichtprimitiv)
a(5)	Fehlende Beobachterinstanz	180	Parameterliste
a(6)	Fehlende Beobachterinstanz	060	Eingabeparameter
a(7)	Datenkollektion	115	Deklarationstyp
b(1)	-	-	-
b(2)	Datenkollektion	115	Deklarationstyp
b(3)	Übergabe Beobachter (Schnittstelle Motiv a(1))	064	Parameterliste
c(1)	-	-	-
c(2)	-	-	-
c(3)	Fehlende Beobachterinstanz	180	Parameterliste
c(4)	Fehlende Beobachterinstanz	060	Parameterliste
d(1)	-	-	-
d(2)	Fehlende Beobachterinstanz	180	Parameterliste
d(3)	Fehlende Beobachterinstanz	060	Parameterliste
d(4)	-	-	-
d(5)	Keine Referenz auf konkreten Beobachter	130	gesamte Klasse
e(1)	Iteration über Datenkollektion	342	Variable (nichtprimitiv)
e(3)	-	-	-
f(1)	Zustand	346	1. Eingabeparameter
f(3)	Zustand	510	1. Eingabeparameter

Tabelle 12: Ermittelte Motivaspekte in Phase A.

Die erste Spalte beschreibt das Teilmotiv zu dem der in der zweiten Spalte bezeichnete Motivaspekt zugeordnet wurde. Die dritte Zeile gibt die Zeile im Quelltext aus Abbildung 82 an, auf die sich der Aspekt bezieht. Keine Angaben in der zweiten und dritten Spalte (gekennzeichnet durch den Strich) deuten darauf hin, dass ein Aspekt nicht zu benennen ist, weil er irrelevant ist. Der genaue Teil der Zeile aus Spalte drei ist in der vierten Tabellenspalte (»Element«) angegeben. Zur Feststellung des Vorhandenseins dieses Elements ist jeweils eine Logik zu erstellen wie in Abschnitt 3.2.1 diskutiert.

Die in Tabelle 12 fehlenden Motive sind jene, für die keine Transformation vom Ausgangs- in den Zielquelltext erforderlich war, für deren zugehörige Programmelemente beide Quelltexte also gleich und musterkonform waren, weil der Ausgangsquelltext in dieser Hinsicht das Motiv bereits implementierte. Für diese Motive zeigt die folgende Tabelle 13 die nachträglich durch manuellen Vergleich ermittelten Motivaspekte:

Motiv	Aspekt	Zeile	Element
e(2)	-	-	-
f(2)	Zustand	380	1. Eingabeparameter

Tabelle 13: Nachträglich ermittelte Motivaspekte zu Phase A.

Der Aspekt zu *e(2)* ist analog zu *e(3)* gewählt worden und der zu *f(2)* analog zu *f(3)* (vgl. Deklaration Schnittstelle mit deren Implementierung in Abbildung 70).

Anhand des Beispiels zur Anwendung des *Observer*-Musters kann deutlich gemacht werden, dass eine scheinbar einfache Funktionalität beliebig kompliziert werden kann. Das genannte Motiv *c* (Registrierung zu beliebigem Zeitpunkt) kann etwa in einem in mehreren Strängen (engl.: *Threads*) ablaufenden Programm einen Synchronisationsmechanismus in der Logik des *Observer*-Musters erforderlich machen. Selbst wenn der Nutzen der Anwendung eines einfach wirkenden Musters

gering erscheint, kann dieser Nutzen in komplexeren Situationen ganz erheblich gesteigert werden. Gesetzt den Fall, dass eine formale Musterdokumentation für verschiedene *Observer*-Varianten vorliegt, nämlich für die vorgestellte, als auch für eine *Thread*-fähige Variante, können unterschiedliche *Observer*-Varianten unter Verwendung derselben Motive angewandt werden. Der Anwender muss gegebenenfalls die Entscheidung treffen, welche Variante er benötigt. Diese Entscheidung könnte auch durch Anbringen zusätzlicher Motive mit Hilfe von Annotationen im Quelltext formal kundgetan werden.

Annotationen

Jedes Motiv wurde durch eine Annotation ausgedrückt. Das Finden der Annotationen ist somit durch das Ermitteln der Motive abgewickelt. Zur Verkürzung des komplexen Beispiels wurden Annotationen in Kurzform geschrieben. Ein Annotationstyp wurde für sie nicht definiert. Letzteres kann ohne Weiteres nachgeholt werden, wie auch im Beispiel demonstriert (siehe Abschnitt 3.3.3.4).

Anhand von in Ausgangs- und Zielquelltext vorhandenen Annotationen und dem Vergleich des jeweiligen Ausgangsteils mit dem zugehörigen Zielteil kann formal nachvollzogen werden, welcher Teil des Ausgangstextes in welchen Teil des Zieltextes überführt wurde, welche Teile weggefallen und welche neu hinzu gekommen sind. Die Information über den relevanten Teil pro Annotation kann teilweise aus den Motivaspekten (Tabelle 12) abgelesen werden, da diese die für die zugehörige Annotation relevante Programmentität bereits benennen. Daraus kann in Konsequenz der Geltungsbereich pro Annotation bestimmt werden. Für Motive, die Transformationen bedingten, welche Zeilen aus dem Ausgangsquelltext entfernt haben (wie etwa Transformation T9), kann aus deren Motivaspekt allerdings nicht direkt der Geltungsbereich der zugehörigen Transformation ermittelt werden. Stattdessen muss er in diesem Fall manuell bestimmt werden.
Den Geltungsbereich von Annotationen festzulegen ist einerseits nützlich, damit später geprüft werden kann, ob eine in einem gegebenen Quelltext angebrachte Annotation gültig ist. Andererseits kann dadurch bei der Musterselektion eingegrenzt werden, welche Annotation der Musterdokumentation mit einer bestimmten Annotation im Quelltext gemeint ist, wenn es mehrere Annotationen pro Motiv gibt.

Für die Definition des Geltungsbereichs einer Annotation sind zwei Fälle zu unterscheiden. Der erste bezieht sich auf den Ausgangsquelltext und ist analog zu den Motivaspekten. Zunächst erwartet man, dass eine als problematisch angesehene Programmstelle in einem Quelltext genauso annotiert wird wie der Ausgangsquelltext in der Musterdokumentation. Daher ist analog diesem der Geltungsbereich pro Annotation zu bestimmen. Andererseits kann es sein, dass eine Annotation für eine Programmentität angebracht wurde, die bereits mit der Lösungsidee des Musters korrespondiert. Für derartige Annotationen ist keine Transformation der annotierten Programmentität durchzuführen. Weiterhin sind sie auch gültig und nicht zu bemängeln. Daher sind auch Geltungsbereiche für die Annotationen des Zielquelltextes der Musterdokumentation zu definieren.
Das Bestimmen der Geltungsbereiche für Annotationen im Ausgangs- und im Zielquelltext ist analog. Weil nur erstere tatsächlich zwingend sind und letztere die Optimierung des Verfahrens betreffen (etwa zur Erkennung von Fällen, wenn ein Muster bereits mit dem Verfahren angewandt wurde), werden im Weiteren nur die Geltungsbereiche für die Annotationen des Ausgangsquelltextes angegeben.
Der Geltungsbereich sollte so stark eingeschränkt werden, dass er der annotierten Programmentität entspricht. Er sollte weiterhin so flexibel gewählt werden, dass er weiteren, bereits absehbar gültigen Programmentitäten nicht im Wege steht. Ob weitere Programmentitäten abgesehen werden können, hängt von der Erfahrung des Entwicklers ab, der die Musterdokumentation erstellt. Im Rahmen von iterativen Verbesserungen der Musterdokumentation kann der Geltungsbereich einer Annotation allerdings modifiziert werden. Es ist also nicht erforderlich, den Geltungsbereich bereits perspektivisch auszurichten, wenngleich das die Güte der Musterdokumentation erhöht. Für die erste transformierte Zeile des Ausgangsquelltextes soll der Geltungsbereich zur Demonstration bestimmt werden. Die entsprechende Zeile lautet:

```
060 acd    PatSelector bo1 = new PatSelector(bo2);
```

Abbildung 102: Erste transformierte Programmzeile im Ausgangsquelltext.

Die zu Zeile 060 gehörenden Motive lauten *a*, *c* und *d*, sie ergeben sich aus der Transformation, die für diese Zeile durchgeführt wurde. Motiv *a* zielt auf die einheitliche Behandlung von Beobachterklassen. Demnach ist dafür alleine der Eingabeparameter *bo2* relevant, der dem entgegen wirkt und die Transformation hin zum Zielquelltext erst notwendig macht. Dabei kann als gleichwertig angesehen werden, ob der Eingabeparameter an einen Konstruktor (wie in Abbildung 70) oder an eine Methode übergeben wird. Motiv *c* handelt von der Registrierung eines Beobachters zu einem beliebigen Zeitpunkt. Es zielt also auf die Verwendung des Konstruktors ab. Hieran wird schon deutlich, dass zu unterschiedlichen Motiven gehörende Annotationen derselben Programmentität potentiell unterschiedliche Geltungsbereiche besitzen. Motiv *d* (»Einheitlicher Registrierungsmechanismus«) kann analog zu Motiv *c* begründet werden.

Nach Untersuchung der Motivaspekte aus Tabelle 12 und Untersuchung des Ausgangsquelltextes erhält man die in Tabelle 14 angegebenen Geltungsbereiche für Annotationen:

Zeile	Motiv	Geltungsbereich
060	a	Eingabeparameter einer gerufenen Methode oder eines Konstruktors
060	c	Konstruktoraufruf
060	d	Konstruktoraufruf
130	a	Variablendeklaration (Klassentyp)
130	b	Variablendeklaration (ungleich Datencontainer)
180	a	Eingabeparameter einer gerufenen Methode oder eines Konstruktors
180	c	Eingabeparameter in Konstruktordeklaration
180	d	Eingabeparameter in Konstruktordeklaration
340	a	Methodenaufruf fremde Klasse
340	b	Methodenaufruf in konkreter anderer Klasse
340	e	Methodenaufruf in konkreter anderer Klasse
340	f	Methodenaufruf in konkreter anderer Klasse
370	a	Klassendeklaration
380	e	Methodendeklaration
380	f	Methodendeklaration

Tabelle 14: Geltungsbereiche für Annotationen zum Ausgangsquelltext.

Die Geltungsbereiche aus Tabelle 14 können teilweise noch feiner aufgestellt werden. Beispielsweise kann für Zeile 060 und Motiv *a* gefordert werden, dass der relevante Eingabeparameter genau dem Beobachter entspricht. Welcher Klassentyp allerdings als Beobachter gelten kann, ist nicht trivial zu erkennen. Ist eine entsprechende Annotation in der Beobachterklasse vorhanden, ist die Erkennung möglich. Ansonsten bleibt nur die Wahl, entweder ein anderes Verfahren wie [Philippow+ 2004] ergänzend hinzuzuziehen oder den Anwender im Rahmen der Musterselektion zu befragen, welche Klasse die Beobachterklasse darstellt.

Transformationen

Jede während der Anwendung des *Observer*-Musters durchgeführte Transformation ersetzte einen Teil im Ausgangsquelltext durch einen anderen Teil im Zielquelltext. Jede Transformation stellt also einem Teil des Ausgangsquelltextes einen Teil des Zielquelltextes gegenüber. Pro Transformation wurden alle sie begründenden bzw. bedingenden Motive angegeben. Gelingt es nun für einen zukünftigen, unbekannten Ausgangsquelltext, analoge Motive zu identifizieren, ist eine analoge Transformation für diesen Ausgangsquelltext angebracht. Eine Transformation wird durchgeführt durch Anwendung von Refactoring-Operationen.

Für bestimmte Quelltexte kann es schwierig sein, eine Transformation automatisiert auszuführen. Dies betrifft insbesondere Anweisungssequenzen innerhalb von Methoden, welche a priori beliebig gestaltet sein können. Siehe auch die Methode *create* im Ausgangsquelltext von Anhang B - Composite.

Refactoring-Operationen

Folgende Refactoring-Operationen wurden bisher im Rahmen der formalen Musterdokumentation gewonnen (Tabelle 15):

Name der Operation	Beschreibung
Introduce_Interface	Schnittstelle anlegen
Add_Interface_To_Signature	Klassensignatur um Schnittstelle erweitern
Add_Interface_Method	Methodendeklaration aus Schnittstelle zu Klasse hinzufügen
Fill_Method_With_Code	Leere Methode mit Logik befüllen
Add_Class_Declaration	Felddeklaration auf Klassenebene hinzufügen
Insert_Line	Beliebige Quelltextzeile in Klasse oder Schnittstelle einfügen
Replace_Line	Beliebige Quelltextzeile in Klasse oder Schnittstelle durch andere Quelltextzeilen ersetzen
Remove_Line	Beliebige Quelltextzeile in Klasse oder Schnittstelle restlos entfernen
Remove_Input_Parameter	Einen Eingabeparameter aus Methode oder Konstruktor entfernen

Tabelle 15: Bisher gewonnene Refactoring-Operationen.

Es wird sich in der ersten Verfeinerungsphase B in Abschnitt 3.3.4 zeigen, dass diese bisher gewonnenen Operationen teilweise wiederverwendet werden können. Andererseits wird sich dann zeigen, ob Operationen generischer gestaltet werden müssen, um auf einen anderen Ausgangsquelltext angewandt werden zu können.

Fazit

In Phase A wurde ein Muster auf einen geeignet gewählten Ausgangsquelltext angewandt. Die Musteranwendung wurde in Quelltexttransformationen aufgeschlüsselt. Jede dieser Transformationen besteht aus einer oder mehreren Refactoring-Operationen. Jeder im Rahmen der Musteranwendung transformierte Quelltextteil wurde durch eine Annotation gekennzeichnet. Jede Annotation wiederum repräsentiert ein Motiv. Somit sind für jede Transformation Motive angegeben, die diese Transformation begründen. Diese Motive sind gleichzeitig Ausdruck der Lösungsidee des angewandten Musters. Die von jeweils einem Motiv betroffenen Programmzeilen wurden derart verschränkt, dass deren Relation zueinander benannt wurde.

Wie zu erwarten war, zeigen die Ergebnisse, dass für die Umsetzung der Phase A eine erhebliche intellektuelle Leistung nötig ist. Die Eignung eines Ausgangsquelltextes kann nur durch einen erfahrenen Entwickler festgestellt werden. Zudem können Motive beliebig komplex sein und nur durch einen menschlichen Leser bewertet werden. Weiterhin kann nur der erfahrene Entwickler beurteilen, ob aufgestellte Refactoring-Operationen als wiederverwendbar angesehen werden können und ob die mit den Operationen verknüpften Vorbedingungen ausreichend erscheinen. Die nachträgliche Begutachtung der Güte einer erstellten Musterdokumentation gemäß dem beschriebenen Prozess obliegt somit letzten Endes einem kompetenten Entwickler und kann nicht in Gänze maschinell verifiziert werden. Maschinell können nur Inkonsistenzen festgestellt werden.

3.3.4 Phase B: Erste Verfeinerungsphase

Nachdem die initiale Phase A in Abschnitt 3.3.3 beschrieben wurde, widmet sich dieser Abschnitt der ersten Verfeinerungsphase von TrAQ zur Erlangung einer formalen Musterdokumentation.
Ziel einer Verfeinerungsphase ist die Verbesserung der Musterdokumentation. Eine Musterdokumentation kann dadurch verbessert werden, dass leicht vom ursprünglich beispielhaft

gewählten Ausgangsquelltext abweichende Quelltexte homogen behandelt werden können. In der Praxis wird es nur selten vorkommen, dass ein gegebener, bisher unbekannter Quelltext exakt mit einem in der Musterdokumentation erfassten Quelltext Q_a übereinstimmt. Daher wird in dieser Phase der ursprünglich gewählte Ausgangsquelltext Q_a leicht variiert. Diese Variation muss vom Dokumentator so gewählt werden, dass das resultierende Q_a' immer noch eine Anwendung von Muster M erlaubt. Demgemäß wird in Phase B der in Phase A gewählte Ausgangsquelltext Q_a modifiziert. Ansonsten sind die Phasen A und B gleich. Abbildung 103 stellt die Phase B samt der Unterschiede zu Phase A dar.

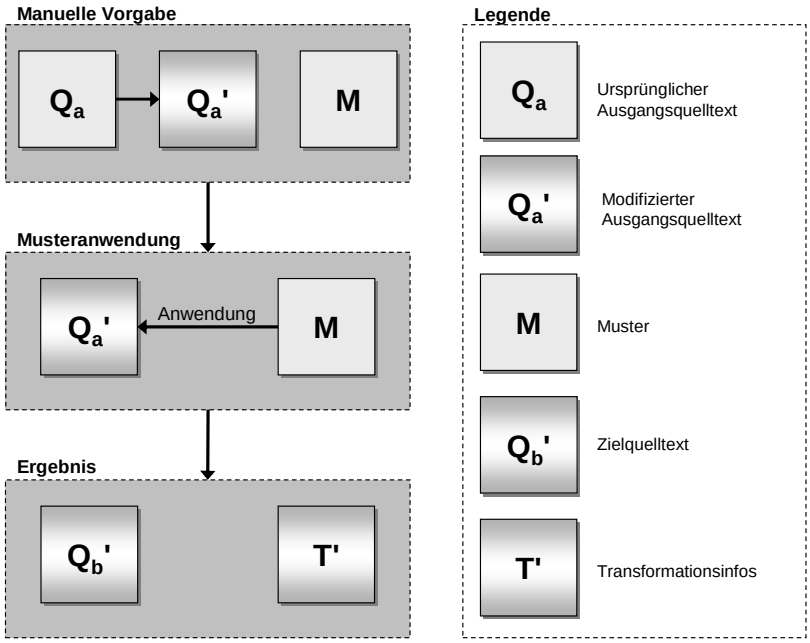

Abbildung 103: Phase B von TrAQ - Erste Verfeinerungsphase.

Die zu Phase A unterschiedlichen Teile sind in der Abbildung 103 mit einem Farbverlauf hinterlegt. Aufgrund des unterschiedlichen Ausgangsquelltextes ergeben sich sowohl ein anderer Zielquelltext (Q_z' statt Q_z aus Phase A) als auch andere Transformationen (T' statt T).

Phase A wurde in Abschnitt 3.3.3 sowohl theoretisch als auch am Beispiel erläutert und zwar getrennt von einander. Phase B und die folgende Phase C basieren auf den gleichen Prinzipien wie Phase A. Die theoretischen und die praktischen Ausführungen dazu werden daher im Folgenden gemeinsam dargestellt.

3.3.4.1 Variieren des Ausgangsquelltextes

Änderungen am Ausgangsquelltext der Phase A sind so vorzunehmen, dass das Muster anwendbar bleibt. Die gegenüber dem ursprünglichen Ausgangsquelltext geänderten Teile sind:

- Zeile 30 und 40: entfernt
- Zeile 60: geändert
- Zeile 180 und 190: geändert
- Zeile 340: geändert
- Zeile 371: neu eingefügt
- Zeilen 372 bis 376: neu eingefügt
- Zeile 380: geändert
- Zeile 395: neu eingefügt

Abbildung 104 zeigt diesen Quelltext für das *Observer*-Beispiel.

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
050        // Konstruiere Observierten
060        PatSelector bo1 = new PatSelector();
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110    public class PatSelector {
120      // Der Observierende
130 ab   private PatDetails dependent;

140      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
150      // benötigt. Er besteht etwa aus einer Handvoll Variablenbelegungen).
160      private Object state;

170      // Konstruiere Objekt und registriere einen Observierenden
180 abcd public PatSelector() {
188        // Konstruiere Observierer und registriere ihn
190 abcF   dependent = new PatDetails(this);
200      }

210      public void start() {
220        //... tue irgendwas hier ...
230        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
240        doWorkChangingState();
250        //... tue auch noch irgendwas anderes hier ...
260      }

270      // Diese Methode tut etwas, was den Zustand dieses Objekts signifikant
280      // beeinflusst. Daher wird in ihr der Observierende informiert. Ihm wird
290      // der aktuelle Zustand dieses Objekts nicht mit übergeben, er muss
300      // mittels getState() explizit abgefragt werden.
310      public void doWorkChangingState() {
320        System.out.println("Ich tue etwas, was meinen Zustand verändert");
330        System.out.println("Deshalb informiere ich PatDetails darüber");
340 e      dependent.update();
350      }

354      public Object getState() {
356        return state;
358      }
360    }

370 a  public class PatDetails {
371      private PatSelector subject;
372      public PatDetails(PatSelector a_subject) {
374        subject = a_subject;
376      }
380 ae   public void update() {
390        // Aktualisiere Zustand und Darstellung
395 F      Object state = subject.getState();
```

```
400         // ...
410         System.out.println("Reagiere auf Zustandsänderung von PatSelector");
420       }
430     }
```

Abbildung 104: Modifizierter Ausgangsquelltext für Anwendung des Observer-Musters.

Zu beachten ist, dass statt dem Motivkürzel *f* das Kürzel *F* verwendet wurde. Der Grund wird in nächsten Abschnitt erläutert. Dort soll auch demonstriert werden, wie mit diesem modifizierten Ausgangsquelltext die bisher gewonnene formale Musterdokumentation überprüft und zudem universeller gestaltet werden kann. Im Gegensatz zu Phase A soll in Phase B keine weitere Benennung von Motivaspekten vorgenommen werden, weil das Vorgehen analog zu dem in Phase A ist und diese Informationen für weitere Betrachtungen nicht notwendig sind.

In Abbildung 104 wird der Beobachter nun innerhalb des Subjekts erzeugt und dabei direkt registriert. Weiterhin wird der Zustand des Subjekts beim Benachrichtigen des Beobachters nicht direkt übergeben, was bei der Transformation beibehalten werden soll (etwa weil der Zustand zeitnah abgefragt werden soll). Der Zustand muss vom Beobachter abgefragt werden. Dazu wird dem Beobachter beim Konstruieren eine Referenz auf das Subjekt übergeben (Zeile 190). Diese Referenz wird im Beispiel als harte Verdrahtung der Klasse *PatSelector* angegeben und wird bei der Transformation aus Gründen der Kürze nicht verändert (in der Praxis würde eine Schnittstelle für das Subjekt verwendet). Die Methode, die zur Benachrichtigung des Beobachters aufgerufen wird, wurde ferner umbenannt, um zu zeigen, wie dies mit der bisher gleich und nun anders benannten Methoden in der Musterdokumentation in Einklang zu bringen ist.

Die Änderung der Zustandsübermittlung von Subjekt an Beobachter hat zur Folge, dass das in Phase A definierte Motiv *f* nicht in seiner ursprünglichen Fassung für Phase B gilt. In Phase A hatte Motiv *f* die Aussage »Bei Benachrichtigung durch Subjekt können Beobachter den Zustand des Subjekts auslesen«. In dieser Phase B ist das Motiv verändert, es lautet stattdessen »Beobachter können nach Benachrichtigung durch Subjekt bei Bedarf den Zustand des Subjekts auslesen« und wird statt mit *f* nun mit *F* bezeichnet. Die Änderung im Ausgangsquelltext sorgt also dafür, dass faktisch eine Mustervariation zustande kommt, nämlich die Veränderung der Zustandsabfrage des Subjekts durch den Beobachter. Dies wird dadurch kenntlich gemacht, dass das Motiv *f* im Folgenden in variierter Form als Motiv *F* geführt wird. In der Praxis könnte statt *F* eine fortlaufende Nummer hinter den Ursprungsnamen des Motivs gestellt werden. Dann hieße das variierte Motiv *f2*. Weil in den Quelltexten und Transformationen das Motivkürzel zwischen Zeilennummer und Inhalt der Programmzeile steht und mit einem Zeichen besser unterzubringen ist als mit zwei Zeichen, wird im Weiteren *F* statt *f2* verwendet

Folgende Transformationen aus dem ursprünglichen Beispiel (Abbildung 70) sind auch für den geänderten Quelltext ohne Änderung durchführbar:

1. Schnittstelle *IObserver* hinzufügen
2. Schnittstelle *IObserver* zu Klasse *PatDetails* hinzufügen
3. Variable *observers* zu Klasse *PatSelector* hinzufügen
4. Registrierung von Beobachter *PatDetails* hinzufügen
5. Benachrichtigungsmechanismus für Beobachter ändern
6. Alten Speichermechanismus für einzelnen Beobachter löschen

Obwohl der Ausgangsquelltext erheblich geändert wurde, können die gerade genannten Transformationen ausgeführt werden. Dies kann durch Betrachten der zu den Transformationen gehörigen Refactoring-Operationen festgestellt werden. Jede dieser Operationen besitzt Vorbedingungen, die ein guter Indikator für die Durchführbarkeit der Operation sind. Konkret kann

das an den folgenden für die eben genannten Transformationen relevanten Refactoring-Operationen nachvollzogen werden (Nummerierung analog zu den Transformationen):

1. *Introduce_Interface* (Abbildung 86)
2. *Add_Interface_To_Signature* (Abbildung 87)
3. *Add_Declaration_in_Class* (Abbildung 91)
4. *Insert_Line* (Abbildung 94)
5. *Replace_Line* (Abbildung 96)
6. *Remove_Line* (Abbildung 101)

Für diese Operationen sind deren Vorbedingungen auch für den geänderten Quelltext zutreffend. Im nächsten Abschnitt wird sich allerdings herausstellen, dass die Refactoring-Operation *Introduce_Interface* modifiziert werden sollte, um eine Mustervariation abzubilden.

Transformation des Ausgangsquelltextes

Die zusätzlich zu den genannten Transformationen durchzuführenden sind der Übersichtlichkeit halber in der Tabelle 16 samt zugehörigen Motiven dargestellt. Diese Transformationen sind genauso ermittelt worden wie in Phase A, nämlich durch Anwendung des Musters auf den nun modifizierten Ausgangsquelltext und unter Vorgabe eines anzuwendenden Musters.

Transformation	Motive
T10: Verschiebe Instanziierung von *PatDetails*	g
T11: Methode *update* umbenennen in *refresh*	a, e
T12: Benachrichtigungsmechanismus für Beobachter ändern	a, e

Tabelle 16: Zusätzliche Transformationen in Phase B für Observer samt zugehöriger Motive.

Um die Transformationen in dieser Phase mit denen in der vorhergehenden Phase A gegenüber stellen zu können, werden in dieser Phase die Indizes der Annotationen gleich gewählt zur vorigen Phase. D.h., wenn für ein bestimmtes Motiv eine Transformation durchgeführt wird die korrespondiert mit einer Transformation aus Phase A (die dasselbe Motiv haben muss), wird die Annotation im Zielquelltext in Phase B mit demselben Index versehen wie in Phase A. Dies ist deshalb notwendig, weil es pro Motiv mehrere Annotationen im Zielquelltext geben kann (vgl. Phase A) und nur mit Hilfe eines gleichen Indexes ein direkter Vergleich möglich ist. Transformationen, die keine Entsprechung zur Phase A haben, sich aber auf ein Motiv beziehen das auch in Phase A referenziert wurde, werden mit einem Index versehen, der größer ist als der höchste Index aus Phase A zu diesem Motiv.

Aufgrund des geänderten Ausgangsquelltexts können mehrere eben nicht genannte, aber in Phase A diskutierte Transformationen nicht in ihrer ursprünglichen Form ausgeführt werden. Diese Transformationen lauten:

1. Konstruktor von Klasse *PatSelector* ändern
2. Methode *register* zu Klasse *PatSelector* hinzufügen
3. Bisherige Registrierung von Beobachter *PatDetails* entfernen

Für die erste dieser Transformationen lautet die zugehörige Refactoring-Operation *Remove_Input_Parameter*. Ihre Vorbedingung *Methode @method in Klasse @class vorhanden* ist für den modifizierten Quelltext verletzt, weil dort in Klasse *PatSelector* kein Konstruktor implementiert ist.

Transformation T10: Verschiebe Instanziierung von PatDetails

Demgemäß muss die Instanziierung von Klasse *PatDetails*, die in Abbildung 104 innerhalb von Klasse *PatSelector* stattfindet, in Klasse *PVAnwendung* vorgenommen werden. Die erste beispielhafte

Transformation hat diesen Fall nicht erfasst, weil dort eine entsprechende Transformation nicht notwendig war. Über die erste Verfeinerungsphase wird hiermit deutlich, dass die ursprüngliche Musterdokumentation um eine – im ersten Beispiel redundante – Transformation zu ergänzen ist.
Das Ergebnis der Transformation lautet (Abbildung 105):

```
039      /**@@motiv g: Instanziiere Beobachter im Klienten*/
040 g    PatDetails dependent = new PatDetails();
...
~~130 ab   private PatDetails dependent;~~
...
~~190 F    dependent = new PatDetails(this);~~
```

Abbildung 105: Transformation T10: Verschieben Instanziierung von PatDetails.

Die Transformation verschiebt eine Instanziierung. Dabei ist zu beachten, dass die Instanziierung aus zwei Teilen besteht. Zum Ersten ist das die Deklaration der Instanzvariablen. Zum Zweiten handelt es sich um die Instanziierung selbst, deren Ergebnis der Instanzvariablen zugewiesen wird.
Das zugehörige, neu eingeführte Motiv *g* ist hier mit »Instanziiere Beobachter im Klienten« benannt worden. Es ist ebenfalls Kern des *Observer*-Musters und Bestandteil der Musterbeschreibung in [Gamma+ 1995]. In der initialen Phase A der Musterdokumentation wurde dieses Motiv noch nicht gefunden. Die Phase B hat dieses Motiv mit Hilfe des modifizierten Ausgangsquelltextes erst zu Tage gefördert.
Die zur Transformation T10 passenden Refactoring-Operationen sorgen dafür, dass die Instanziierung von *PatDetails* durch Klasse *PatSelector* zunächst entfernt und dann wieder an anderer Stelle, nämlich in Klasse *PVAnwendung*, eingefügt wird. Dies geschieht durch zwei Operationen. Die erste schneidet eine Anweisung aus, die zweite Operation fügt diese Anweisung an anderer Stelle wieder ein. Das Ausschneiden und Einfügen bedient sich dabei einer Zwischenablage. Die erste Operation zum Ausschneiden einer Anweisung lautet (Abbildung 106):

```
Name: Cut_Line @class @line
Eingabeparameter:
   @class:    Klasse, die auszuschneidende Anweisung enthält
   @line:     Zeile, in der auszuschneidende Anweisung steht
Vorbedingungen:
   Klasse @class existiert
   Zeile @line der Klasse @class existiert und enthält eine Anweisung
Ausgangsannotation: "motiv g" (Zeile 130 bzw. 190)
Zielannotation: siehe Paste_Line (weiter unten)
Operation: Ausschneiden einer Anweisung in eine Zwischenablage
```

Abbildung 106: Refactoring-Operation Cut_Line.

Die Operation *Cut_Line* wird zweimal aufgerufen, einmal für das Ausschneiden der Instanzvariable in Zeile 130 (Abbildung 104) und zum anderen für das Ausschneiden der Instanziierung von *PatDetails* in Zeile 190 (selbe Abbildung). Für den ersten Aufruf von *Cut_Line* sind die Parameter wie folgt zu belegen. Parameter *@class* wird auf den Wert *PatSelector* gesetzt, das ist die Klasse, in der sich die auszuschneidende Zeile befindet. Parameter *@line* erhält den Wer *130*, die Zeilennummer der Anweisung.

Anschließend kann die in die Zwischenablage ausgeschnittene Anweisung an anderer Stelle durch eine weitere Refactoring-Operation eingefügt werden (Abbildung 107):

```
Name: Paste_Line @class @line
Eingabeparameter:
   @class:    Klasse, die Anweisung einzufügen ist
   @line:     Zeile, vor der die Anweisung einzufügen ist
Vorbedingungen:
   Klasse @class existiert
   Zeile @line der Klasse @class existiert
Ausgangsannotation: siehe Cut_Line (weiter oben)
Zielannotation: "motiv g" (Zeile 40)
Operation: Einfügen einer Anweisung aus der Zwischenablage
```

Abbildung 107: Refactoring-Operation Paste_Line.

Für das Einfügen der Instanzvariablen ist Parameter *@class* auf den Wert *PVAnwendung* zu setzen und Parameter *@line* auf den Wert *39*.
Nun kann unter erneuter Verwendung von *Cut_Line* und *Paste_Line* die zweite Anweisung aus Zeile 190 ausgeschnitten und in Zeile 40 eingefügt werden. Danach lautet der durch Transformation entstandene Quelltext (Abbildung 108):

```
039        private PatDetails dependent;
040 g      dependent = new PatDetails(this);
```

Abbildung 108: Vorläufiges Ergebnis einer Transformation mit Cut_Line und Paste_Line.

Es gibt ein Problem mit einer logischen Zweiteilung einer Operation in zwei Teiloperationen, wie dies mit *Cut_Line* und *Paste_Line* getan wurde. Wie in Abbildung 106 und Abbildung 107 zu sehen, konnte jeweils nur eine Annotation angegeben werden. Der Grund ist, dass für die Ausschneideoperation (*Cut_Line*) die Zwischenablage das Ziel war, in der keine Annotationen existieren; für die Einfügeoperation (*Paste_Line*) hingegen war die Zwischenablage die Quelle. Daher ist es logisch einfacher, die beiden Refactoring-Operationen *Cut_Line* und *Paste_Line* zu einer Operation *Move_Line* zusammenzufassen. *Move_Line* tut nichts anderes, als zuerst *Cut_Line* und danach *Paste_Line* auszuführen. Die Deklaration von *Move_Line* kann angegeben werden als (Abbildung 109):

```
Name: Move_Line @src_class @src_line @dst_class @dst_line
Eingabeparameter:
   @src_class:    Klasse, die auszuschneidende Anweisung enthält
   @src_line:     Zeile, in der auszuschneidende Anweisung steht
   @dst_class:    Klasse, die Anweisung einzufügen ist
   @dst_line:     Zeile, vor der die Anweisung einzufügen ist
Vorbedingungen:
   Klasse @src_class existiert
   Klasse @dst_class existiert
   Zeile @src_line der Klasse @src_class existiert und enthält eine
                 Anweisung
   Zeile @dst_line der Klasse @dst_class existiert
Ausgangsannotation: "motiv ab" (Zeile 130) bzw. "motiv abc" (Zeile 190)
Zielannotation: "motiv g" (Zeile 39) bzw. "motiv g" (Zeile 40)
Operation: Verschieben einer Anweisung
```

Abbildung 109: Refactoring-Operation Move_Line.

In Abbildung 109 sind der Kürze wegen für zwei Aufrufe von *Move_Line* die Parameter für Ausgangs- und Zielannotation angegeben. Tatsächlich sind zwei Aufrufe der Operation mit je einer Parameterangabe notwendig.

Die Anwendung von *Move_Line* ist analog zur kombinierten Ausführung von *Cut_Line* und *Paste_Line*, mit der Ausnahme, dass *Move_Line* eine in sich geschlossene, atomare Operation repräsentiert. Der Vorteil ist, dass Ausgangs- und Zielannotation nun direkt angegeben werden können. Verglichen mit dem gewünschten Ergebnis aus Abbildung 105 existieren drei Unterschiede in Abbildung 108:

a) Die Deklaration der Instanzvariablen weicht durch den Modifizierer *private* ab.
b) Die Instanziierungsanweisung ist auf zwei Zeilen verteilt anstatt in einer Zeile zu stehen.
c) Der Konstruktor von *PatDetails* wird mit dem Parameterwert *this* aufgerufen anstatt ohne Parameter.

Der erste Unterschied erklärt sich dadurch, dass eine Deklaration, die auf Klassenebene vorlag, auf die Methodenebene verschoben wurde. Er kann nur durch eine Korrekturlogik behoben werden. Diese Korrekturlogik, die universell innerhalb einer Programmiersprache anwendbar ist, prüft gegen bestimmte Konstellationen und modifiziert nach fest vorgegebenen Regeln. Diese Regeln spiegeln die Syntax der Programmiersprache wider. Im Falle der Abbildung 108 kann eine Regel greifen die besagt, dass innerhalb von Methoden keine Sichtbarkeitsmodifizierer für Felddeklarationen erlaubt sind. Für Java lauten diese Modifizierer *private*, *protected*, *package* und *public*.

Der zweite Unterschied kann durch eine Umformungslogik, die ebenfalls als universell innerhalb einer Programmiersprache anzusehen ist, egalisiert werden. Die relevante Regel lautet, dass die Deklaration einer Variablen mit der ersten Anweisung, in der die Variable Verwendung findet, zusammengelegt werden kann, wenn letztere Anweisung im selben Block wie die Deklaration steht. Stünde die Anweisung in einem untergeordneten Block, etwa innerhalb einer *If*-Anweisung, wäre die Sichtbarkeit der Deklaration nach Umformung niedriger. Daher ist die Block-Bedingung für Java generell wichtig.

Der dritte Unterschied entstand originär durch die Transformation, die den Parameterwert *this* beim Verschieben eines Programmteils aus einer Klasse in eine andere Klasse nicht angepasst hat. Statt über die *this*-Referenz muss eine Referenz auf das Subjekt mittels der Instanzvariablen *bo2* hergestellt werden, die in Klasse *PVAnwendung* bekannt ist. Dies kann im aktuellen Beispiel durchaus von einem Algorithmus erkannt und durchgeführt werden. Denn *this* ist prinzipiell klassenabhängig. Eine Verschiebung von *this* in eine andere Klasse bedeutet immer, dass Nacharbeit nötig ist. Diese Nacharbeit bedeutet entweder die Ersetzung von *this* oder das Verschieben von relevanten Teilen aus der Ursprungsklasse in die Klasse, wo *this* hinverschoben wurde. Letzteres ist nur fallweise durchführbar und nicht relevant im aktuellen Beispiel. Demnach ist die eben entstandene Zeile 040 noch nicht in ihrer endgültigen Form. Vielmehr muss im Konstruktoraufruf der Eingabeparameter entfernt werden. Mittels einer neuen Refactoring-Operation *Remove_Input_Parameter* kann dies bewerkstelligt werden. Die Operation lautet:

```
Name: Remove_Input_Parameter @class @line @index
Eingabeparameter:
   @class:     Klasse, in der Methode @method sich befindet
   @line:      Zeile, die den zu erweiternden Methodenaufruf enthält
   @index:     Index des Parameterwerts, der zu entfernen ist
Vorbedingungen:
   Klasse @class vorhanden
   Zeile @line in Klasse @class vorhanden
   In Zeile @line existiert genau ein Methodenaufruf
   Es existieren mindestens @index Parameterwerte im Methodenaufruf
Ausgangsannotation: "motiv g" (Zeile 40)
Zielannotation: "motiv g" (Zeile 40)
Operation: Parameter an durch @index bezeichneten Stelle in Zeile @line der
           Klasse @class entfernen.
```

Abbildung 110: Refactoring-Operation Remove_Input_Parameter.

Die Klasse *@class* lautet *PVAnwendung*, die Zeilennummer *@line* lautet 40, der Parameter *@index* bekommt den Wert 1 zugewiesen. Die neu entstandene Zeile 040 heißt dann (der Übersichtlichkeit halber zusammen mit Zeile 039):

```
039         private PatDetails dependent;
040 g       dependent = new PatDetails();
```

Abbildung 111: Ergebnis einer Transformation mit Remove_Input_Parameter.

Transformation T11: Methode update umbenennen in refresh

Weil im modifizierten Ausgangsquelltext die in Phase A ursprünglich mit *refresh* benannte Methode nun *update* heißt, ist eine weitere Transformation vorzunehmen, die in Phase A nicht notwendig war. Die Methode *update* in Zeile 380 ist umzubenennen in *refresh*, um mit *Observer* konform zu sein (vgl. Schnittstelle *IObserver*):

```
380 ae   public void refresh() {
```

Abbildung 112: Ergebnis von Transformation T11.

Die zugehörige Refactoring-Operation zum Umbenennen einer Methode kann wie in der Abbildung 113 angegeben werden:

```
Name: Rename_Method @class @line @new_name
Eingabeparameter:
   @class:    Klasse, die umzubenennende Methode enthält
   @line:     Zeile, in der umzubenennende Methode steht
   @new_name: neuer Name der Methode
Vorbedingungen:
   Klasse @class existiert
   In Zeile @line der Klasse @class steht eine Methodendeklaration
Ausgangsannotation: "motiv e" (Zeile 380)
Zielannotation: "motiv e" (Zeile 380)
Operation: Umbenennen einer Methode
```

Abbildung 113: Refactoring-Operation Rename_Method.

Die Parameter sind wie folgt zu belegen. Parameter *@class* erhält den Wert *PatDetails*, Parameter *@line* den Wert *380* und Parameter *@new_name* den Wert *refresh*. Ergebnis ist die Umbenennung der Methode in Zeile 380 von *update* nach *refresh*.
Durch die Umbenennung müssen alle Anweisungen, die die umbenannte Methode rufen, ebenfalls modifiziert werden. Das kann direkt über die Refactoring-Operation *Rename_Method* geschehen. Dieses Vorgehen, das kombinierte Verändern der Methodendeklaration und aller Aufrufer, entspricht dem, was moderne Java-Entwicklungsumgebungen bieten. Somit wird der Aufruf von Methode *update* in Zeile 340 durch *Rename_Method* automatisch angepasst, so dass danach Methode *refresh* gerufen wird.

Neben dem im Vergleich zum Ursprungsbeispiel (Abbildung 70) unterschiedlichen Methodennamen ist auch die Signatur der nun mit *refresh* (vorher mit *update*) benannten Methode verschieden. Die Signatur enthält keinen Eingabeparameter. Im Ursprungsbeispiel repräsentierte der einzige Eingabeparameter den Zustand des Subjekts, der vom Subjekt direkt in jeden benachrichtigten Beobachter übergeben wurde. Im Ausgangsquelltext aus Abbildung 104 fiel dieser Parameter zugunsten einer neu eingeführten Methode *getState()* weg. Letztere liefert einem Beobachter nach expliziter Anfrage den Zustand des Subjekts. Vorteile dieses Vorgehens können insbesondere dann entstehen, wenn die Zustandsermittlung des Subjekts zeitaufwändig oder speicherintensiv ist oder wenn sich der Subjektzustand sehr häufig ändern kann und eine aktuelle Sicht auf diesen Zustand gewünscht ist. Sollen diese möglichen Vorteile konserviert werden, muss *getState()* bei Bedarf vom Beobachter aufgerufen werden. Ist diese Konservierung nicht gewünscht, kann *getState()* im Subjekt in Methode *doWorkChangingState()* direkt und einmalig vor Benachrichtigung der Beobachter aufgerufen werden, um dann bei Aufruf der Beobachter mitgegeben zu werden (analog zum Ursprungsbeispiel in Phase A). Abhängig von der Entscheidung bzgl. der Verwendung von *getState()* fällt die Transformation im Rahmen dieses Beispiels aus. An dieser Stelle wird klar, dass eine explizit durch den Beobachter gerufene Methode *getState()* eine Mustervariante zum *Observer*-Muster aus Phase A darstellt. Zur Veranschaulichung der Handhabbarkeit einer Mustervariante innerhalb einer formalen Musterdokumentation soll *getState()* explizit im Beobachter gerufen werden. Dies passiert bereits im Ausgangsquelltext und bedarf keiner weiteren Transformation. Stattdessen muss eine Variation von Schnittstelle *IObserver* vorgenommen werden. Dort muss der Eingabeparameter der Methode *refresh* entfernt werden, damit eine Konformität zwischen der Schnittstelle und der Zeile in Abbildung 112 hergestellt ist. In Abschnitt 3.3.3.4 wurde bereits die mögliche Aufspaltung einer Transformation diskutiert, die zum damaligen Zeitpunkt nicht notwendig war. Aufgrund der Mustervariation in Schnittstelle *IObserver* ist diese Notwendigkeit nun gegeben. Die zu Abbildung 73 variierte Form der Teiltransformation *T1b* lautet (Abbildung 114):

```
509   /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
510   void refresh();
```

Abbildung 114: Transformation T1b: Modifizierte Methode refresh ohne Eingabeparameter.

Der Unterschied zwischen Abbildung 73 und Abbildung 114 ist der Wegfall des Eingabeparameters in letzterer Abbildung. Die Refactoring-Operation *Introduce_Interface* aus Abbildung 86 kann hierfür verwendet werden. Der Refactoring-Operation wird die einzuführende Schnittstelle übergeben, die sich in Phase B gegenüber Phase A geändert hat.

In Phase A wurde für Zeile 380 der Abbildung 82 keine dedizierte Refactoring-Operation oder Vorbedingung angegeben. Der Grund ist, dass diese Zeile im Ausgangsquelltext zufällig bereits die richtige Form hatte und unverändert in den Zielquelltext übernommen werden konnte. Durch Variation des Ausgangsquelltextes in Phase B konnte aufgedeckt werden, dass es durchaus sinnvoll ist, eine Erweiterung der Musterdokumentation hinsichtlich der Implementierung der Benachrichtigungsmethode im Beobachter vorzunehmen, um die Abhängigkeit des eben benannten Dreigespanns auszudrücken.
Eine generelle Möglichkeit, die Abhängigkeit zwischen Schnittstellendeklaration und Implementierung der Schnittstelle zu manifestieren, ist das Vorsehen einer Regel im Werkzeug, mit Hilfe dessen die formale Musterdokumentation vorgenommen wird. Diese Abhängigkeit ist in Java prinzipiell gegeben und kann maschinell ausgewertet werden. Eine zweite Möglichkeit, die in Ergänzung zur ersten umgesetzt werden kann, ist das explizite Definieren der Abhängigkeit innerhalb des Musters. Der Vorteil dieser expliziten Definition ist die Verbesserung der Klarheit der Musterdokumentation. Einem Entwickler kann bei Bedarf direkt angezeigt werden, dass diese Abhängigkeit besteht. Das kann einerseits für das Musterverständnis vorteilhaft sein, andererseits dann, wenn eine automatisiert durchgeführte Transformation nicht fehlerfrei ausgeführt werden konnte.

Transformation T12: Benachrichtigungsmechanismus für Beobachter ändern

Die nächste Transformation betrifft die Benachrichtigung aller registrierten Beobachter durch das Subjekt bei Zustandsänderung des Subjekts. Wie in Phase A wird eine Schleife mittels eines Iterators eingeführt. Sie sorgt für eine entsprechende Benachrichtigung. Das Ergebnis der Transformation ist in Abbildung 115 dargestellt:

```
341     /**@@motiv a(4): Behandle Beobachterklassen einheitlich*/
341     /**@@motiv e(1): Benachrichtige alle Beobachter*/
342     Iterator it = observers.iterator();while (it.hasNext()) {
343       /**@@motiv a(4/1): Behandle Beobachterklassen einheitlich*/
343       /**@@motiv e(1/1): Benachrichtigung einzelner Beobachter*/
344       IObserver observer = (IObserver)it.next();
345       /**@@motiv a(4/2): Behandle Beobachterklassen einheitlich*/
345       /**@@motiv e(1/2): Benachrichtigung einzelner Beobachter*/
346       observer.refresh();
348     }
```

Abbildung 115: Transformation T12: Benachrichtigungsmechanismus für Beobachter ändern.

Die bereits in Phase A eingeführte Refactoring-Operation *Replace_Line* führt die Transformation aus. Sie ersetzt die Zeile 340 des Ausgangsquelltexts durch die Observer-Logik zur Benachrichtigung der Beobachter. Die zu ersetzende Zeile lautet (Abbildung 116):

```
340 be     dependent.refresh();
```

Abbildung 116: Durch Operation Replace_Line ersetzte Zeile in Phase B.

Die stattdessen einzusetzenden Zeilen lauten (Abbildung 117):

```
342     Iterator it = observers.iterator();while (it.hasNext()) {
344       IObserver observer = (IObserver)it.next();
346       observer.refresh();
348     }
```

Abbildung 117: Durch Operation Replace_Line einzusetzende Zeilen in Phase B.

Somit ist die Transformation des modifizierten Ausgangsquelltexts abgeschlossen. Die zu den Transformationen korrespondierenden Refactoring-Operationen wurden ermittelt und angegeben. Der resultierende Zielquelltext ist in folgender Abbildung 118 angegeben:

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Beobachter
039        /**@@motiv g(1): Instanziiere Beobachter im Klienten*/
040 g      PatDetails bo2 = new PatDetails();
050        // Konstruiere Observierten
059        /**@@motiv a(6): Behandle Beobachterklassen einheitlich*/
059        /**@@motiv c(4): Registrierung zu beliebigem Zeitpunkt*/
059        /**@@motiv d(3): Einheitlicher Registrierungsmechanismus*/
060 acd    PatSelector bo1 = new PatSelector();
063        /**@@motiv b(3): Registrierungsmöglichkeit vieler Beobachter*/
063        /**@@motiv c(2): Registrierung zu beliebigem Zeitpunkt*/
063        /**@@motiv d(1): Einheitlicher Registrierungsmechanismus*/
065 bcd    bo1.register(bo2);
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110    public class PatSelector {
114      /**@@motiv a(7): Behandle Beobachterklassen einheitlich*/
114      /**@@motiv b(2): Registrierungsmöglichkeit vieler Beobachter*/
115 ab   private List observers = new Vector();
140      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
150      // benötigt. Er besteht etwa aus einer Handvoll Variablenbelegungen).
160      private Object state;

170      // Konstruiere Objekt und registriere einen Observierenden
179      /**@@motiv a(5): Behandle Beobachterklassen einheitlich*/
179      /**@@motiv c(3): Registrierung zu beliebigem Zeitpunkt*/
179      /**@@motiv d(2): Einheitlicher Registrierungsmechanismus*/
180 acd  public PatSelector() {
200      }

201      /**@@motiv a(3): Behandle Beobachterklassen einheitlich*/
201      /**@@motiv b(1): Registrierungsmöglichkeit vieler Beobachter*/
```

```
201      /**@@motiv c(1): Registrierung zu beliebigem Zeitpunkt*/
201      /**@@motiv d(4): Einheitlicher Registrierungsmechanismus*/
202 abcd public void register(IObserver observer) {
204        observers.add(observer);
206      }

210      public void start() {
220        //... tue irgendwas hier ...
230        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
240        doWorkChangingState();
250        //... tue auch noch irgendwas anderes hier ...
260      }

270      // Diese Methode tut etwas, was den Zustand dieses Objekts signifikant
280      // beeinflusst. Daher wird in ihr der Observierende informiert. Ihm wird
290      // der aktuelle Zustand dieses Objekts nicht mit übergeben, er muss
300      // mittels getState() explizit abgefragt werden.
310      public void doWorkChangingState() {
320        System.out.println("Ich tue etwas, was meinen Zustand verändert");
330        System.out.println("Deshalb informiere ich PatDetails darüber");
341        /**@@motiv a(4): Behandle Beobachterklassen einheitlich*/
341        /**@@motiv e(1): Benachrichtige alle Beobachter*/
342 ae     Iterator it = observers.iterator(); while (it.hasNext()) {
343          /**@@motiv a(4/1): Behandle Beobachterklassen einheitlich*/
343          /**@@motiv e(1/1): Benachrichtigung einzelner Beobachter*/
344 ae       IObserver observer = (IObserver)it.next();
345          /**@@motiv a(4/2): Behandle Beobachterklassen einheitlich*/
345          /**@@motiv e(1/2): Benachrichtigung einzelner Beobachter*/
346 ae       observer.refresh();
348        }
350      }

353      /**@@motiv F(5): Beobachter können Subjekt-Zustand auslesen*/
354 F    public Object getState() {
356        return state;
358      }
360    }

369    /**@@motiv a(4): Behandler Beobachterklassen einheitlich*/
370 a  public class PatDetails implements IObserver {
371      /**@@motiv F(3): Beobachter können Subjekt-Zustand auslesen*/
371 F    private PatSelector subject;
372      /**@@motiv F(4): Beobachter können Subjekt-Zustand auslesen*/
372 F    public PatDetails(PatSelector a_subject) {
374        subject = a_subject;
376      }
379      /**@@motiv a(2): Behandler Beobachterklassen einheitlich*/
379      /**@@motiv e(2): Benachrichtigung einzelner Beobachter*/
380 ae   public void refresh() {
390        // Aktualisiere Zustand und Darstellung
394        /**@@motiv F(1): Übergabe Zustand an einzelnen Beobachter*/
395 F      Object state = subject.getState();
400        // ...
410        System.out.println("Reagiere auf Zustandsänderung von PatSelector");
```

```
420       }
430     }

499     /**@@motiv a(1): Vereinheitliche Beobachterklassen*/
500 a   public interface IObserver {
509       /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
510 e     void refresh();
520     }
```

Abbildung 118: Zielquelltext nach Anwendung des Observer-Musters in Phase B.

Im Folgenden werden die Unterschiede zwischen Phase A und B diskutiert. Zuerst werden die Motive beider Phasen betrachtet, danach die Zielquelltexte verglichen.

3.3.4.2 Vergleichen der Motive aus Phase A und B

Das Motiv g wurde in Phase B neu gefunden. Weiterhin wurde Motiv *f* zu *F* variiert, das wie folgt lautet: »Beobachter können nach Benachrichtigung durch Subjekt bei Bedarf den Zustand des Subjekts auslesen«.

Das zusätzlich in Phase B gefundene Motiv *g* heißt: »Instanziiere Beobachter im Klienten«. Ein neu hinzugekommenes Motiv in Phase B erfordert eine weitergehende Untersuchung der Ursache. In Phase B ist das Motiv neu eingeführt worden, weil es zur Begründung einer Transformation notwendig war. Diese Transformation war nicht relevant in Phase A, weil dort der entsprechende Quelltextteil bereits in seiner Zielform war. Im Gegensatz dazu mussten in Phase B die logisch zusammen gehörenden Zeilen 130 und 190 transformiert werden. Da jede Transformation mittels eines Motivs begründet werden muss und keines der bereits in Phase A gefundenen Motive *a* bis *f* zur Beschreibung geeignet waren, musste Motiv *g* eingeführt werden. Im Beispiel in Phase B bedeutet das Motiv *g* letztendlich nur die Verfeinerung der formalen Musterdokumentation. In der späteren Phase C (Abschnitt 3.3.5) wird gezeigt, dass ein Motiv auch aufgrund einer Mustervariation neu einzuführen ist.

Die in Phase A und B entstandenen Zielquelltexte werden im Rahmen des nächsten Abschnitts unter Berücksichtigung der dort durch Annotationen angebrachten Motive verglichen.

3.3.4.3 Vergleichen der Zielquelltexte

Der Vergleich der Zielquelltexte aus Phase A und B dient der Kontrolle, ob alle entdeckten Unterschiede durch Motive, Transformationen und Refactoring-Operationen abgedeckt wurden. Die Unterschiede der Zielquelltexte der beiden Phasen beschränken sich auf zwei Fälle. Erstens handelt es sich um die leicht unterschiedliche Deklaration des Beobachters in Klasse *PVAnwendung*. Dieser Unterschied ist erklärbar durch die Verschiebung der Deklaration aus einem Klassenkontext hinein in einen Methodenkontext. Zweitens liegt ein Unterschied vor, der sowohl Schnittstelle *IObserver* als auch Klasse *PatDetails* betrifft. Letztgenannte Klasse ist abhängig von *IObserver*, weil sie diese Schnittstelle implementiert. So ist es auch zu erklären, dass die Methode *refresh* in beiden Klassen ohne Eingabeparameter vorkommt, im Gegensatz zum Zielquelltext in Phase A. Dieser Unterschied gründet auf einer Mustervariation. In Phase B wurde also eine andere Schnittstelle als in Phase A unter dem Namen *IObserver* realisiert. Technisch kann das so abgebildet werden, dass für den geänderten Schnittstellencode (Methode *refresh* hat keinen Eingabeparameter mehr) entweder ein neuer Schnittstellenname gewählt wird, beispielsweise *IObserver2*. Oder es wird eine Versionierung eingeführt, die es erlaubt, zu jeder Schnittstelle zu ein und demselben Namen mehrere Versionen zu spezifizieren. Eine entsprechende Versionsnummer müsste dann entweder in der mit der Mustervariante verknüpften Refactoring-Operation direkt angegeben werden. Oder es wird eine separate Zuordnungstabelle geführt, in der zu jeder Mustervariation vermerkt wird, welche Version

pro betroffenem Musterteil (faktisch ein Stück Quelltext, etwa eine Schnittstelle, eine abstrakte Basisklasse oder eine Anweisungssequenz) ihr zugeordnet ist.

Die Motive *a* bis *e* sind in den Zielquelltexten der Phasen A und B vertreten. Die annotierten Quelltextteile werden im Rahmen dieses Abschnitts verglichen. Dazu wird jeder mit einem Motiv versehene Quelltextteil aus Phase A mit einem identisch annotierten Teil aus Phase B verglichen. Das gleiche findet in umgekehrter Weise, also ausgehend von Phase B, statt. Dies ist notwendig, da zwar dieselben Motive, nicht aber unbedingt die gleiche Anzahl an Annotationen pro Motiv in den Zielquelltexten der Phasen A und B vorkommen. Wurden pro Motiv mehrere Annotationen vorgenommen, so wurde in jeder Annotation zum Motiv ein Index mitgeführt, der bei eins startet und für jede weitere Annotation zum Motiv um eins erhöht wurde. Beispielsweise wurde im Zielquelltext von Phase A (Abbildung 82) in Zeile 499 die Annotation */**@@motiv a(1): Vereinheitliche Beobachterklassen*/* angebracht. Die Zahl in Klammern ist eben besagter Index.

Die folgende Tabelle 17 stellt die mit gleichem Motiv annotierten, aber unterschiedlich ausgeprägten Quelltextteile der Zielquelltexte aus Phase A und B gegenüber. Die Zahl am Anfang jeder Spalte repräsentiert die Zeilennummer des jeweiligen Zielquelltextes.

Phase A	Phase B
`346: observer.refresh(state);`	`346: observer.refresh();`
`380: public void refresh(Object state) {`	`380: public void refresh() {`
`510: void refresh(Object state);`	`510: void refresh();`

Tabelle 17: Annotierte Programmentitäten mit gleichem Motiv und abweichender Form.

Die Zeilennummern dieser Teile stimmen deshalb überein weil das Beispiel günstig gestaltet wurde, um einen Vergleich der Quelltexte aus den Phasen A und B zu begünstigen. Eine Übereinstimmung der Zeilennummern ist aber nicht erforderlich, da Annotationen als Referenz ausreichen.

Für die in Tabelle 17 genannten Zeilen gilt, dass zusätzlich zur Abweichung in der Gestalt der annotierten Programmentitäten ein Motiv in Phase B fehlt, das in Phase A vorhanden ist. Es handelt sich um das Motiv *f*, welches die Zustandsübergabe an die Beobachter ausdrückt. Genau diese Zustandsübergabe findet in Phase B aber nicht explizit bei der Benachrichtigung der Beobachter statt. Alle damit zusammenhängenden Stellen sind mit der Motivvariante *F* annotiert. Die Zustandsübergabe ist in die Beobachter verlagert, die bei Bedarf den Zustand des Subjekts mittels der in Phase B neu eingeführten Methode *getState()* erfragen. Diese Änderung in der Zustandsbehandlung war aber gerade Gegenstand der Variante in Phase B. Somit kann die Abweichung in den Motiven konsistent erklärt werden, ebenso wie die Abweichung der Gestalt der annotierten Programmentitäten gemäß Tabelle 17. Wäre eine Erklärung derart nicht möglich gewesen, deutete dies auf eine unvollständige oder inkonsistente Dokumentation in Phase A bzw. B hin. In diesem Fall müsste bei Bedarf beispielsweise ein neues Motiv eingeführt, weitere Annotationen im entsprechenden Zielquelltext vorgenommen oder bestehende modifiziert werden.

Weiterhin gibt es Annotationen im Zielquelltext aus Phase B, die nicht in Phase A vorkommen. Die zugehörigen Anweisungen sind (vgl. Abbildung 118):

1. Zeile 190: `dependent = new PatDetails(this);`
2. Zeile 371: `private PatSelector subject;`
3. Zeilen 372-376: `public PatDetails(PatSelector a_subject) { … }`

Alle diese Programmstellen weichen deshalb von Phase A ab, weil die Zustandsübergabe des Subjekts an die Beobachter in Phase B weggefallen und durch Methode *getState()* ersetzt wurde. Der Grund ist also derselbe wie für die Zeilen aus Tabelle 17. Im Übrigen können sind diese Programmstellen mit der Motivvariante *F* annotiert.

Beim Vergleich der Zielquelltexte müssen Feinheiten der Programmiersprache mit einbezogen werden. So ist es etwa in Java nicht erforderlich, einen Konstruktor ohne Parameter zu implementieren, wenn dieser keine Logik trägt. Ein Konstruktor ohne Parameter wird auch Default-Konstruktor genannt. Es spricht allerdings nichts dagegen den Default-Konstruktor ohne Logik dennoch zu implementieren.
Existiert in einem Zielquelltext also ein leerer Default-Konstruktor und im anderen Zielquelltext nicht, so sind die Zielquelltexte bezüglich des Konstruktors als identisch zu bewerten. Dies gilt nicht, wenn der leere Default-Konstruktor annotiert ist, was allerdings nach Meinung des Autors nicht sinnvoll erscheint.
Anders verhält es sich mit leeren Konstruktoren, die Eingabeparameter beinhalten. Existiert nämlich mindestens ein Konstruktor mit Parameter und ist der Default-Konstruktor nicht deklariert, so ist der Default-Konstruktor auch implizit nicht mehr vorhanden! Ist also ein leerer Konstruktor mit Eingabearameter in einem Zielquelltext vorhanden und in einem anderen nicht, liegt qualitativ ein Unterschied vor, weil der Default-Konstruktor im erstgenannten Quelltext nicht (mehr) implizit vorhanden ist, im anderen aber schon. Ein leerer Konstruktor mit Eingabeparameter macht in Java nach Kenntnislage des Autors nur Sinn, wenn der Default-Konstruktor explizit ausgeschlossen werden soll.

Im Zielquelltext aus Phase A existieren keine Programmteile, die nicht (auch nicht in modifizierter Form) in Phase B vorhanden sind. Existierten diese Programmteile, wären sie genauso zu untersuchen wie dies in diesem Abschnitt weiter oben für in Phase B neu hinzugekommene Programmteile beschrieben wurde. Es gilt also bei jedem Unterschied zwischen den Zielquelltexten zweier Phasen eine Erklärung für diesen Unterschied zu finden. Ist der Unterschied durch unterschiedliche Ausgangsquelltexte oder durch Verwendung einer Mustervariante erklärbar, ist er gerechtfertigt. In allen anderen Fällen muss die Konsistenz der Musterdokumentation geprüft und entsprechend korrigiert bzw. vervollständigt werden.

Beim Vergleich der Zielquelltexte von Phase A und B sind dem Autor Unterschiede aufgefallen, die durch ungünstig gewählte Transformationen entstanden sind. So war beispielsweise die Zeile 040 des Zielquelltextes aus Phase B ursprünglich in Zeile 062. Dies kam dadurch zustande, dass die an sich frei wählbare Zeilennummer für das Ziel der zugehörigen Transformation entsprechend gewählt wurde. Im Vergleich ist es aber ungünstig, unterschiedliche Zeilennummern für Anweisungen zu haben, die wahlweise an verschiedenen Stellen im Programm stehen können ohne dessen Verhalten zu ändern. Es ist etwa egal, ob zuerst die Anweisung der Zeile 040 aus Abbildung 118 ausgeführt wird und dann die aus Zeile 060 oder umgekehrt. Beim Vergleich erschweren Zeilennummern, die zwischen den Vergleichsquelltexten unnötigerweise unterschiedlich sind, die Rechtfertigung der Unterschiede. Deshalb ist der Vergleich der Zielquelltexte ein gutes Mittel, um die formale Musterdokumentation über die unterschiedlichen Phasen so konsistent wie möglich zu gestalten.

3.3.4.4 Fazit

Die erste Verfeinerungsphase des Verfahrens, die Phase B, hat folgende Ergebnisse gebracht. Die Musterdokumentation konnte durch Variation des Ausgangsquelltextes auf eine Mustervariante ausgeweitet werden. Im Zuge dessen wurde ein neues Motiv entdeckt, dass durch eine in Phase A zufällige Übereinstimmung von Ausgangs- und Zielquelltext nicht gefunden werden konnte. Weiterhin wurde eine Motivvariation gefunden. Die formale Musterdokumentation wurde also insgesamt dadurch verbessert, dass eine Mustervariation dokumentiert wurde. Weiterhin wurde der Fall berücksichtigt, dass ein Methodenname (im Beispiel lautete er *update*) nicht identisch mit dem Methodennamen ist, der in einer für das Muster einzuführenden Schnittstelle (hier: *IObserver*) deklariert ist. Die Musterdokumentation wurde also flexibilisiert. Im Rahmen der Transformationen in Phase B wurden Refactoring-Operationen eingeführt, die in Phase A noch nicht definiert wurde. Damit wurde der verfügbare Satz an wiederverwendbaren Operationen aufgestockt. Davon kann in zukünftig zu erstellenden Musterdokumentationen profitiert werden. Mit Hilfe der Refactoring-Operationen *Cut_Line*, *Paste_Line* und der daraus zusammengesetzten Operation *Move_Line* wurde

demonstriert, dass eine Refactoring-Operation aus mehreren anderen komponiert werden kann. Die Wiederverwendbarkeit von Refactoring-Operationen erhöht sich dadurch letztendlich. Der Vorteil der Ausführung einer zusammengesetzten Refactoring-Operation anstatt der Einzelausführung der von benutzten Einzeloperationen wurde beschrieben.

An die gerade beschriebene erste Verfeinerungsphase B schließt sich die zweite Verfeinerung, die Phase C, an.

3.3.5 Phase C: Zweite Verfeinerungsphase

Eine Musterdokumentation kann dadurch verbessert werden, dass sie verschiedene Variationen eines Musters erfassen kann. Durch Variation des Musters können die bisher gewonnenen Refactoring-Operationen sowie Motive und Annotationen auf Ihre Verwendbarkeit hin überprüft werden. Fehlt etwa eine Annotation oder ist die Vorbedingung einer Refactoring-Operation nicht vollständig, kann dies möglicherweise durch Mustervariation festgestellt werden.
Das Grundprinzip der Phase C ist gleich dem der Phasen A und B, mit dem Unterschied, dass eine Variation des Musters betrachtet wird. In folgender Abbildung 119 ist Phase C samt den Unterschieden zu Phase A dargestellt:

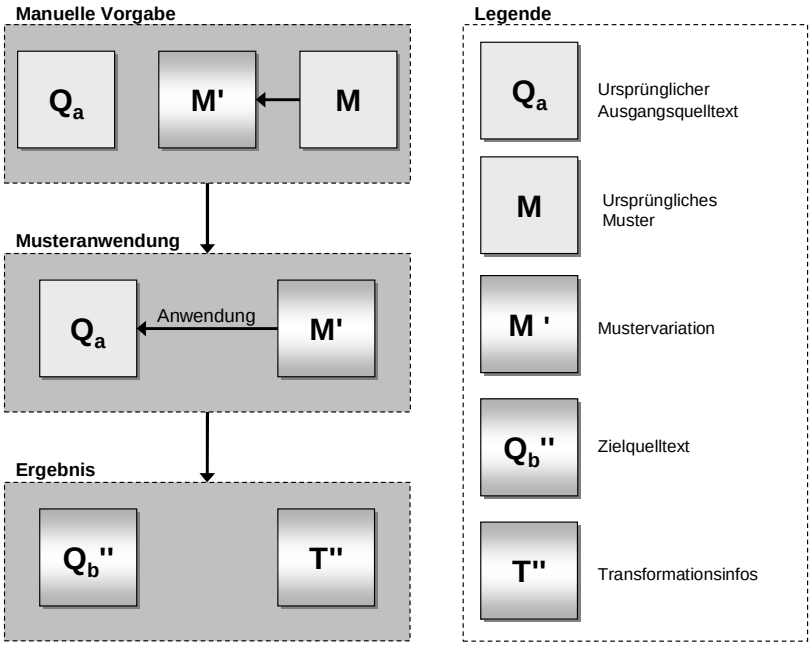

Abbildung 119: Phase C von TrAQ - Zweite Verfeinerungsphase.

Die zu Phase A unterschiedlichen Teile sind in der Abbildung 119 mit einem Farbverlauf hinterlegt.
Auf den Ausgangsquelltext Q_a, der gleich dem aus Phase A ist, wird nun also die Mustervariante M' angewandt. In einem späteren Vergleich zwischen Phase A und C können dann die Auswirkungen dieser Mustervariation untersucht werden. Statt Q_a hätte auch Q_a' aus Phase B verwendet werden können. Das macht qualitativ dann keinen Unterschied, wenn später die Ergebnisse zwischen Phase C und B anstatt zwischen Phase C und A verglichen werden.
Analog zu Phase A und B wird M' nun auf Q_a angewandt. Als Ergebnis entstehen wieder ein Zielquelltext, Q_z'', sowie Transformationen T''.

3.3.5.1 Variation des Musters

Das ursprünglich in den Abschnitten 3.3.3 und 3.3.4 vorgegebene und angewandte Muster *Observer* wird in dieser Phase der Musterdokumentation in einem Punkt verändert. Bisher wurden alle registrierten Beobachter durch das Subjekt selbst benachrichtigt. Die Benachrichtigungsstrategie war

direkt im Subjekt kodiert. In der in diesem Schritt eingeführten Mustervariation werden die Beobachter durch einen sogenannten *Change Manager* benachrichtigt. Dieser *Change Manager* wird vom Subjekt referenziert. Das Subjekt delegiert die Benachrichtigung der Beobachter an den *Change Manager*, anstatt diese Logik selbst zu implementieren. Der *Change Manager* bietet den Vorteil, ein frei konfigurierbares Strategieobjekt zu verwenden, das die Beobachterbenachrichtigung auf unterschiedliche Weise realisieren kann, ohne dass das Subjekt geändert werden muss.

Die einzelnen Transformationen und Refactoring-Operationen werden für Phase C nicht mehr angegeben. Das Prinzip der Ermittlung und Durchführung dieser ist analog zu den vorigen Phasen A und B.

Nach Anwendung der *Observer*-Variation auf den Ausgangsquelltext aus Abbildung 70 entsteht folgender Zielquelltext (Abbildung 120):

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Observierten
040        PatDetails bo2 = new PatDetails();
050        // Konstruiere Observierer
059        /**@@motiv c(4): Registrierung zu beliebigem Zeitpunkt*/
059        /**@@motiv d(3): Einheitlicher Registrierungsmechanismus*/
060 cd     PatSelector bo1 = new PatSelector();
062        // Registriere Observierer
063        /**@@motiv b(3): Registrierung Beobachter bo2*/
063        /**@@motiv c(2): Registrierung zu beliebigem Zeitpunkt*/
063        /**@@motiv d(1): Einheitlicher Registrierungsmechanismus*/
064 bcd    bo1.register(bo2);
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110    public class PatSelector {
115 ab   private List observers = new Vector();
120      private ChangeManager changeMan;

140      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
150      // benötigt. Er besteht etwa aus einer Handvoll Variablenbelegungen).
160      private Object state;

170      // Konstruiere Objekt und registriere einen Observierenden
180 bcd  public PatSelector(PatDetails dependent) {
190        changeMan = new ChangeManager();
195        changeMan.register(dependent);
200      }

202 abc  public void register(IObserver observer) {
204        observers.add(observer);
206      }

210      public void start() {
220        //... tue irgendwas hier ...
230        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
240        doWorkChangingState();
```

```
250         //... tue auch noch irgendwas anderes hier ...
260       }

310       public void doWorkChangingState() {
320         System.out.println("Ich tue etwas, was meinen Zustand verändert");
330         System.out.println("Deshalb informiere ich PatDetails darüber");
339         /**@@motiv f(4): Übergabe Zustand an einzelnen Beobachter*/
339         /**@@motiv e(4): Benachrichtigung einzelner Beobachter*/
340 e       changeMan.notify(state);
350       }
360     }

369     /**@@motiv a(2): Vereinheitliche Beobachterklassen*/
370 a   public class PatDetails implements IObserver {
379       /**@@motiv e(2): Benachrichtigung einzelner Beobachter*/
379       /**@@motiv f(2): Übergabe Zustand an einzelnen Beobachter*/
380 ef    public void refresh(Object state) {
390         // Aktualisiere Zustand und Darstellung
400         // ...
410         System.out.println("Reagiere auf Zustandsänderung von PatSelector");
420       }
430     }

499     /**@@motiv a(1): Vereinheitliche Beobachterklassen*/
500 a   public interface IObserver {
509       /**@@motiv e(3): Benachrichtigung einzelner Beobachter*/
509       /**@@motiv f(3): Übergabe Zustand an einzelnen Beobachter*/
510 ef    void refresh(Object state);
520     }

599     /**@@motiv h(1): Erlaube konfigurierb. Benachrichtigung d. Beobachter*/
600     public class ChangeManager() {
610       private List observers = new Vector();
620       private PatSelector subject;
630       public ChangeManager(PatSelector subject) {
640         this.subject = subject;
650       }
660       public void register(IObserver observer) {
670         observers.add(observer);
680       }

690       public void notify(Object state) {
705         /**@@motiv a(4): Vereinheitliche Beobachterklassen*/
705         /**@@motiv e(1): Benachrichtige alle Beobachter*/
710 ae      Iterator it = observers.iterator();while (it.hasNext()) {
719           /**@@motiv a(4/1): Vereinheitliche Beobachterklassen*/
719           /**@@motiv e(1/1): Benachrichtigung einzelner Beobachter*/
720 ae        IObserver observer = (IObserver)it.next();
725           /**@@motiv a(4/2): Vereinheitliche Beobachterklassen*/
725           /**@@motiv e(1/2): Benachrichtigung einzelner Beobachter*/
725           /**@@motiv f(1): Übergabe Zustand an einzelnen Beobachter*/
730 aef       observer.refresh(state);
740         }
750       }
```

```
760     }
```

Abbildung 120: Zielquelltext mit angewandter Variante des Observer-Musters in Phase C.

In Zeile 190 des Quelltextes wird der *Change Manager* der Einfachheit des Beispiels halber vom Subjekt selbst erzeugt. Flexibler ist die Instanziierung des *Change Managers* vom Klienten (siehe Klasse *PVAnwendung*) mit anschließender Übergabe der konstruierten Objektinstanz an das Subjekt über dessen Konstruktor.

Für die in Phase A und B gewonnene Refactoring-Operation sowie Motive soll geprüft werden, inwieweit Ergänzungen oder Veränderungen zur Berücksichtigung der Mustervariation aus dieser Phase erforderlich sind. Da sich Annotationen direkt aus Motiven ergeben und Transformationen lediglich dazu dienen, Refactoring-Operationen zu gewinnen, müssen beim Vergleich nur Motive betrachtet werden.

3.3.5.2 Vergleich der Mustervariationen

Die in Phase C verwendete Variante des *Observer*-Musters besitzt folgende Unterschiede zum ursprünglich verwendeten Muster:

1) Die Benachrichtigung der Beobachter geschieht nicht mehr direkt durch das Subjekt. Stattdessen delegiert das Subjekt die Verantwortung dafür an einen *Change Manager*.
2) Bei Benachrichtigung der Beobachter wird der Zustand nicht direkt übergeben. Stattdessen bietet das Subjekt eine Methode *getState()* an. Diese kann ein Beobachter zum Auslesen des Subjektzustands aufrufen.

In Konsequenz ist in Phase C eine neue Klasse *Change Manager* hinzugekommen. Weiterhin besitzt die Methode *refresh* in Schnittstelle *IObserver* nun keinen Eingabeparameter mehr. Aus letzterem folgt automatisch, dass die Beobachter (die die Schnittstelle implementieren) ebenfalls die Methode *refresh* ohne Parameter implementieren (müssen).

Allen in Phase C gegenüber Phase A geänderten Teilen muss ein Motiv zugeordnet werden. Existiert aus den Vorgängerphasen A und B noch kein geeignetes Motiv, muss ein neues eingeführt werden. Im gewählten Beispiel können die geänderten Teile wie folgt mit den bestehenden Motiven versehen werden:

1) Die Benachrichtigung der Beobachter durch den *Change Manager* kann mit Motiv *e* markiert werden.
2) Die Methode *refresh* in der Beobachterklasse behält ihre Bedeutung, die Abfrage des Subjektzustands findet allerdings innerhalb der Methode statt anstelle über einen Eingabeparameter.
3) Die Abfrage des Zustands innerhalb von Methode *refresh* über Methode *getState* wird mit Motiv *F* versehen.

Der *Change Manager* dient zur konfigurierbaren Benachrichtigung der Beobachter. Dies kann durch ein neu einzuführendes Motiv *h* ausgedrückt werden. Motiv *h* kann etwa lauten: »Erlaube konfigurierbare Benachrichtigung der Beobachter«. Demgemäß kann Zeile 600 im Zielquelltext in Abbildung 120 wie folgt annotiert werden (Abbildung 121):

```
599 /**@@motiv h(1): Erlaube konfigurierb. Benachrichtigung d. Beobachter*/
600 public class ChangeManager() {
```

Abbildung 121: Annotation des Change Managers.

Die Anwendung der *Observer*-Variante mit *Change Manager* hat im Zielquelltext die Eigenschaft der Konfigurierbarkeit der Beobachterbenachrichtigung hinzugefügt, die im Ausgangsquelltext nicht vorhanden war. Der Bedarf nach dieser Eigenschaft wurde auch nicht durch Motive im Ausgangsquelltext ausgedrückt. Die Konfigurierbarkeit der Beobachterbenachrichtigung ist also durch die Musteranwendung ohne explizite Benutzeranforderung entstanden. In zukünftigen Ausgangsquelltexten kann durchaus der Bedarf nach dieser Konfigurierbarkeit bestehen. Gerade dann entfaltet sich der Nutzen der Mustervariation in Phase C, die es erst ermöglicht hat, das Motiv *h* zu entdecken.

3.3.5.3 Vergleich der Zielquelltexte

Zunächst folgt der Vergleich zwischen den Zielquelltexten aus Phase A und C, danach zwischen denen aus Phase B und C.

Gegenüber dem Zielquelltext aus Phase A (Abbildung 82) sind folgende Unterschiede in Abbildung 120 vorhanden. Die Zeilen 120, 190, 195 und alle Zeilen ab 600 sind hinzugekommen. Alle diese neuen Zeilen sind entstanden durch die Einführung der Klasse *Change Manager*. Gleiches gilt für die durch eine einzelne Anweisung ersetzten Zeilen 340 bis 348. Diese Ersetzung trägt dem Rechnung, dass die Benachrichtigung der Beobachter nicht mehr durch das Subjekt, sondern exklusiv durch den *Change Manager* erfolgt. Alle anderen Zeilen sind gleich. Insbesondere haben sich die Klassen *PVAnwendung*, *PatDetails* und *IObserver* gegenüber dem Zielquelltext mit angewandtem *Observer*-Muster ohne *Change Manager* nicht geändert. Aufgrund dessen sind also keine Inkonsistenzen oder Unvollständigkeiten in der Musterdokumentation in den Phasen A und C erkennbar.

Der Vergleich zwischen den Zielquelltexten aus Phase B (siehe Abbildung 118) und C liefert analoge Ergebnisse. In Phase C sind folgende Unterschiede gegenüber Phase B festzustellen. Die Zeilen 120, 190, 195 und alle Zeilen ab 600 sind durch Einführung der Klasse *Change Manager* hinzugekommen. Die Ersetzung der Zeilen 341 bis 348 durch einen einzelne Zeile 340 (Abbildung 120) kann ebenso begründet werden.
Weil in Phase C der Zustand des Subjekts direkt bei der Benachrichtigung an die Beobachter übergeben wird, ergeben sich Unterschiede in den Zeilen 040, 380 und 510. Dies ist allerdings analog zu den Unterschieden zwischen Phase A und B, die in Phase B bereits diskutiert wurden. Die Analogie ist dadurch zu erklären, dass der Ausgangsquelltext aus Phase C gleich dem Ausgangsquelltext aus Phase A ist. Demnach können zwischen Phase B und C durchaus Unterschiede existieren, die ebenfalls zwischen Phase B und A vorhanden sind.
Auch aufgrund des Vergleichs der Zielquelltexte der Phasen B und C können keine Inkonsistenzen oder Unvollständigkeiten festgestellt werden.

3.3.5.4 Fazit

Mit Hilfe dieser Phase konnte die Dokumentation einer neuen Variante des *Observer*-Musters vorgenommen werden. Im Zuge dessen wurde ein neues Motiv eingeführt. Durch einen weiteren Vergleich, in den die Zielquelltexte aus den Phasen A und B involviert wurden, konnte die Wahrscheinlichkeit für inkonsistente oder bis dato unvollständige Musterdokumentationen verringert werden.

Optional können zusätzlich zu den in den Abschnitten 3.3.4 und 3.3.5 beschriebenen ersten zwei Verfeinerungsphasen B und C beliebig viele weitere durchgeführt werden. In jeder weiteren Verfeinerungsphase ist eine noch nicht behandelte Variation des Ausgangsquelltextes oder des zu

dokumentierenden Musters zu prozessieren. Es entsteht je ein Zielquelltext sowie mitunter eine bis dahin noch nicht dokumentierte Mustervariante. Je mehr Verfeinerungsphasen durchlaufen werden, umso mehr Varianten eines Musters oder Ausgangsquelltextes können potentiell durch eine formale Musterdokumentation abgedeckt werden.

3.3.6 Phase D: Konsolidierungsphase

Diese Phase widmet sich der Konsolidierung der in den vorigen Phasen gewonnenen Informationen. Dies beinhaltet einerseits das Aufstellen der formalen Musterdokumentation mit dem Zweck, diese für die Musterselektion und -anwendung verwenden zu können. Andererseits werden die gewonnenen Annotationen, Transformationen, Refactoring-Operationen sowie Motive hinsichtlich ihrer Güte untersucht und können bei Bedarf umgestaltet werden. Zur Verfeinerung der Musterdokumentation werden weiterhin Isomorphe zu annotierten Programmelementen identifiziert. Als erstes folgt die Angabe der formalen Musterdokumentation zum *Observer*-Muster. Danach folgt die Verfeinerung der Musterdokumentation.

3.3.6.1 Aufstellung der formalen Musterdokumentation

Die formale Musterdokumentation kann nach Durchführung der initialen Dokumentationsphase A sowie den beiden Verfeinerungsphasen B und C angegeben werden. Die in den vorigen Phasen gewonnenen Informationen beinhalten:

- Musternamе
- Pro Verfeinerungsphase
  - Exemplarischer Ausgangsquelltext
  - Durch exemplarische Musteranwendung resultierender Zielquelltext
- Motive
  - Motivaspekte
  - Annotationen pro Motiv
    - Geltungsbereiche pro Annotation
  - Transformationen pro Motiv
    - Refactoring-Operation pro Transformation
      - Parameter zur Operation
      - Informale Beschreibung der Operation
      - Vorbedingungen pro Refactoring-Operation

Die formale Musterdokumentation wird auf Grundlage dieser Informationen aufgestellt. In den Phasen A bis C wurden Annotationen, Motive und Transformationen während des Gebrauchs ad hoc eingeführt. Um diese Elemente für die Musterselektion und -anwendung auswerten zu können, müssen sie strukturiert formalisiert abgelegt werden.

Bevor die einzelnen Bestandteile der Musterdokumentation aus den bisherigen Informationen abgeleitet und zusammengestellt werden, folgt in Abbildung 122 eine Übersicht der Bestandteile.

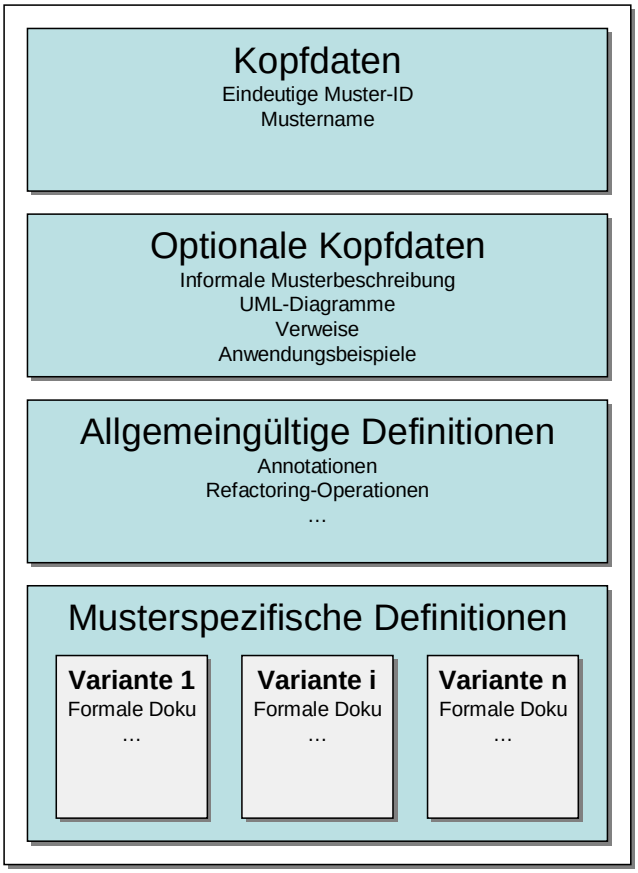

Abbildung 122: Aufbau der formalen Musterdokumentation.

Im Folgenden werden die zur formalen Musterdokumentation von *Observer* nötigen Daten in der Reihenfolge der Abschnitte gemäß Abbildung 122 erörtert. Die Ablage der Daten zur formalen Musterdokumentation kann prinzipiell in jedem als geeignet erachteten Format erfolgen. Pro Datenabschnitt werden Hinweise zu Umsetzungsmöglichkeiten gegeben. Um alle Informationen zur Musterdokumentation logisch in Zusammenhang zu bringen, kann eine Deklarationsdatei verwendet werden, die eine Referenz auf alle Datenabschnitte beinhaltet. Auch die Speicherung in einer Datenbank ist möglich. Der Bezug der einzelnen Teile kann über das als nächstes diskutierte Schlüsselfeld der eindeutigen Muster-Identifikation erfolgen.

Erster Bestandteil der formalen Musterdokumentation sind die Kopfdaten. Die Kopfdaten des *Observer*-Musters können einfach angegeben werden (Abbildung 123):

```
Eindeutige Muster-ID: Observer
Musternname: Observer
```

Abbildung 123: Kopfdaten des Observer-Musters in der formalen Musterdokumentation.

Die eindeutige Identifikation (kurz: ID) zum Muster ist frei wählbar, genau wie der Musternname. Die ID muss zwingend über alle Musterdokumentationen eindeutig sein, im Gegensatz zum Musternamen. Der Musternname sollte zwar ebenfalls eindeutig sein; weil er aber dem Anwender nur zur Information dargestellt wird, ist er beliebig wählbar. Die ID ist deshalb zusätzlich zum Musternamen eingeführt, weil sie bei einer Referenzierung sinnvollerweise möglichst bündig und ohne Sonderzeichen aufgestellt werden sollte. Wäre sie gleich dem Musternamen, wäre dies nicht in jedem Fall möglich.

Die optionalen Kopfdaten enthalten ausschließlich Angaben, die für die Darstellung gegenüber dem interessierten Entwickler oder einer anderen Person gedacht sind. Diese Angaben können formal oder informal sein. Sie sind jedenfalls nicht für die Auswertung durch den in dieser Arbeit beschriebenen Ansatz geeignet. Für das *Observer*-Muster können etwa alle Informationen aus [Gamma+ 1995] im Abschnitt der optionalen Kopfdaten angegeben werden, also textuelle Beschreibungen, Quelltextbeispiele sowie UML-Diagramme. Auch Links auf Ressourcen im Internet haben in diesem Abschnitt ihren Platz. Das Format der Speicherung der optionalen Kopfdaten kann nach Belieben vonstatten gehen. Beispielsweise kann eine HTML-Datei mit Verweisen erstellt und diese per Link aus der formalen Musterdokumentation referenziert werden.

Nach Kopfdaten und optionalen Kopfdaten folgen die allgemeinen Definitionen, auf denen die formalen Musterdokumentation des konkreten Musters (hier: *Observer*) beruht. Die allgemeinen Definitionen werden praktischerweise so abgelegt, dass sie für alle Musterdokumentationen zugänglich sind und referenziert werden können. Genauer gesagt, werden die allgemeinen Definitionen so abgelegt, dass sie aus den musterspezifischen Definitionen heraus referenziert werden können und so, dass ein Algorithmus auch die allgemeinen Definitionen im Zugriff haben kann, sobald er die musterspezifischen Definitionen verarbeitet.

Für Transformationen gilt prinzipiell: Alle für ein Muster relevanten Teile müssen durch je mindestens eine Transformation erfasst worden sein. Das bedeutet, dass beispielhafte Ausgangsquelltexte (Phase A bis C) so zu wählen sind, dass jeder im Muster relevante Teil über alle Ausgangsquelltexte je mindestens einmal nicht in der benötigten Form vorhanden ist. Beispielsweise ist es für die in Phase A diskutierte *Observer*-Form erforderlich, dass in der Benachrichtigungsmethode *notify* des Beobachters der Zustand übergeben wird. Dieser Eingabeparameter des Zustands in der *notify*-Methode des Beobachters sollte in mindestens einem Ausgangsquelltext der Phasen A bis C fehlen. Ist dies nicht gegeben, können zukünftig betrachtete unbekannte Ausgangsquelltexte in den fraglichen Teilen auf keinen Fall automatisiert transformiert werden.
Andererseits ist es nicht unbedingt erforderlich, die Benachrichtigungsmethode selbst im Beobachter in einem der Ausgangsquelltexte wegzulassen. Zumindest ist dies dann vertretbar, wenn davon auszugehen ist, dass Benachrichtigungen der Beobachter bei Zustandsänderung des Subjekts durch Methodenaufruf erfolgen. Die Benachrichtigung durch Methodenaufruf entspricht schließlich dem weit verbreiteten Vorgehen. Es ist gute Praxis, dies so zu tun. Eine etwas abwegige, aber möglicherweise im Einzelfall angebrachte andere Benachrichtigungsart ist das Setzen einer Variablen im Beobachter. Diese Variable könnte vom Beobachter beispielsweise zyklisch abgefragt werden, etwa innerhalb eines *Threads*. Nun könnte hier weiter argumentiert werden, dass im Rahmen von *Observer* die Benachrichtigung den Beobachter sofort erreichen soll und nicht nach einer Latenzzeit, die im genannten Beispiel von der Machart des *Threads* bestimmt wird. Weiterhin könnte gegen das etwas exotisch gewählte Beispiel gehalten werden, dass das Setzen einer Variablen für gewöhnlich nicht direkt, sondern über eine Setter-Methode stattfindet. Damit ist aber die Benachrichtigung wieder näher an die methodenbasierte, intuitiv als nach Meinung des Autors einzig sinnvoll anzusehende Benachrichtigungsart gerückt.
Sollte die Musterdokumentation für *Observer* also Ausgangsquelltexte mit anderen Benachrichtigungsmechanismen als via Methode erfassen, so muss ein entsprechend repräsentativer exemplarischer Ausgangsquelltext in mindestens einer der Phasen A bis C gewählt werden.

Andererseits kann mittels der bisher entwickelten formalen Musterdokumentation durchaus festgestellt werden, dass in einem vorgegebenen Ausgangsquelltext eine Benachrichtigungsmethode vorhanden zu sein hat. In den in den Phasen A bis C entstandenen Zielquelltexten ist jeweils die Methode *notify* in der Beobachterklasse mit einer Annotation gekennzeichnet. Diese Annotation drückt ein Motiv aus. Ein Motiv bedingt eine oder mehrere Transformationen. Eine Transformation wiederum ist als notwendige Maßnahme gegenüber einem Quelltext aufzufassen, damit dieser Quelltext dem *Observer*-Muster entsprechen kann. Ein Quelltext, der einer Anforderung des Musters nicht entspricht, weicht demnach vom Muster ab. Eine Abweichung deutet aber auf fehlende

Musterteile im Quelltext hin. Gerade das kann also durch eine formale Musterdokumentation festgestellt werden.

Alle Quelltextteile, die durch dasselbe Motiv (etwa *a*, *b* oder *c*) bedingt sind, hängen von einander ab! Eine Abhängigkeit kann meist automatisiert ermittelt werden, beispielsweise beim Aufruf einer Methode, weil Aufrufstelle und Deklarationsstelle der Methode technisch von einander abhängig sind.

Alle durch die Phasen A bis C definierten Mustervarianten werden in derselben Musterdokumentation zusammengefasst. Die logische Klammer um die Mustervarianten ist der Name des Musters, das in Variationen dokumentiert wurde. Jede Mustervariation wird getrennt von den anderen Varianten abgelegt. Ein Vergleich der Varianten ist möglich durch Betrachten der Motive pro Variante sowie der Transformationen bzw. Refactoring-Operationen pro Variante. Die wiederverwendbaren Teile einer formalen Musterdokumentation werden in einem öffentlichen Bereich (auch *Repository* genannt) abgelegt. Diese Teile beinhalten Motive, Annotationsdefinitionen, Transformationen, Refactoring-Operationen und alle sonstigen kontextfrei einsetzbaren Teile. Eine Mustervariation muss dann lediglich eine Referenz auf ein Teil im öffentlichen Bereich halten.

Definition von Annotationen

Eine Annotation ist die Anwendung eines abstrakten Annotationstypen (vgl. Abschnitt 2.3.5). Daher sind nicht Annotationen, sondern deren jeweils zugehöriger Annotationstyp zu definieren.
Für zwei Annotationen der formalen Musterdokumentation aus Phase A (Abschnitt 3.3.3) wird der zugehörige Annotationstyp im Folgenden zur Veranschaulichung aufgestellt. Da zur besseren Lesbarkeit während der formalen Musterdokumentation Annotationen in Kurzform geschrieben wurden, werden die betrachteten Annotationen nachstehend zusätzlich in ihrer Langform angegeben.

Die erste Annotation, für die ein Annotationstyp aufzustellen ist, lautet in Kurzform und zusammen mit der annotierten Anweisung (Abbildung 124):

```
/**@@motiv a(1): Behandle Beobachterklassen einheitlich*/
public interface IObserver {
```

Abbildung 124: Annotation in Kurzform, für die ein Annotationstyp definiert werden soll.

Da die Langform einer Annotation auf einen Annotationstypen verweist, muss letzterer zuerst definiert werden. Der Annotationstyp beinhaltet eine eindeutige, frei wählbare Identifikation, eine Parameterliste, eine menschenlesbare Beschreibung sowie die Angabe gültiger Geltungsbereiche. Die Identifikation soll hier beispielhaft mit 4711 gewählt werden. Die Liste der Parameter ist leer, da die Annotation keine Parameter besitzt.
Die menschenlesbare Beschreibung entspricht dem Text, der in der gegebenen Kurzform der Annotation vorliegt. Sie lautet somit »Behandle Beobachterklassen einheitlich«. Die Sprache der Beschreibung ist demnach Deutsch. Die genannte Annotation wurde für eine öffentliche Schnittstellendeklaration angebracht, also kann der Geltungsbereich in symbolischer Schreibweise mit *PUBLIC_INTERFACE* bezeichnet werden. Im Falle eines Musters ist prinzipiell davon auszugehen, dass dort definierte Schnittstellen öffentlich sind (und nicht nur die Sichtbarkeit innerhalb eines Java-Paketes haben). Wären sie nicht öffentlich, wäre deren Zweck fraglich, da sie nicht in den Kontext bildenden Anwendungsteilen, die immer außerhalb des abstrakten Musterkerns liegen, anwendbar wären.

Somit kann der Annotationstyp zur genannten Annotation angegeben werden mit (Abbildung 125):

```
MID:  4711{("motiv", String, obligatory), ("index", int, obligatory)}
HID:  "Motiv @motiv(@index): Behandle Beobachterklassen einheitlich"
SCOPES: PUBLIC_INTERFACE
```

Abbildung 125: Definition des Annotationstyps 4711.

Für die HID wurde keine Sprache angegeben, als Defaultsprache wird Deutsch festgesetzt. An dieser Stelle soll noch einmal die Symmetrie von Geltungsbereichen herausgestellt werden. Da eine Schnittstelle erstens deklariert und zweitens von einer Klasse implementiert wird, sind die Geltungsbereiche zur Deklaration der Schnittstelle und zur Implementierung der Schnittstelle (*class MyClass implements MyInterface*) als gleichwertig anzusehen. In dieser Arbeit wird daher von einer Symmetriebeziehung zwischen Geltungsbereichen gesprochen, die sich aus programmiersprachlichen Vorgaben ableitet. Zu Geltungsbereich *PUBLIC_INTERFACE* (Abbildung 125) ist der symmetrische Geltungsbereich der, der eine Klassensignatur mit Schnittstellendeklaration beschreibt. Er könnte mit *ANY_CLASS_DECLARATION_WITH_INTERFACE* benannt werden. Bei der Angabe der Geltungsbereiche für einen Annotationstypen ist es daher nicht zwingend erforderlich, den Geltungsbereich explizit vollständig anzugeben. Die zu angegebenen Geltungsbereichbestandteilen (hier nur einer, nämlich *PUBLIC_INTERFACE*) symmetrischen Geltungsbereiche (hier wäre das *ANY_CLASS_DECLARATION_WITH_INTERFACE*) können anhand von einmalig aufzustellenden, von der Programmiersprache abhängigen Regeln, automatisiert abgeleitet werden.
Soll keine implizite Ergänzung von Geltungsbereichen durch Symmetrie vorgenommen werden, kann dies durch Vorsehen eines Steuerkennzeichens erreicht werden. Beispielsweise könnte ein vor einen Bestandteil des Geltungsbereichs gestelltes Minuszeichen die Symmetrie ausschließen, also etwa so:

```
SCOPES: -PUBLIC_INTERFACE
```

Die Langform der Annotation kann nun ebenfalls aufgestellt werden. Sie lautet:

```
/*
 * @@annotation
 * MID: 4711("a", 1)
 * HID: "Motiv a(1): Behandle Beobachterklassen einheitlich"
 */
```

Abbildung 126: Langform einer Annotation zum Annotationstypen 4711.

Die zweite zu betrachtende Annotation lautet zusammen mit der annotierten Anweisung:

```
/**@@motiv f(3): Übergabe Zustand an Beobachter*/
void refresh(Object state);
```

Abbildung 127: Weitere Annotation in Kurzform, für die Annotationstyp aufzustellen ist.

Im Unterschied zur ersten Annotation bezieht sich das durch die Annotation ausgedrückte Motiv spezifisch auf einen Teil der annotierten Anweisung, nämlich auf den ersten Eingabeparameter der Methode. Dies muss durch den Geltungsbereich im Annotationstypen ausgedrückt werden. Der Annotationstyp kann angegeben werden mit (Abbildung 128):

```
MID:  4712{("motiv", String, obligatory), ("index", int, obligatory)}
HID:  "Motiv @motiv(@index): Übergabe Zustand an Beobachter"
SCOPES: METHOD_PARAMETER
```

Abbildung 128: Definition des Annotationstyps 4712.

Die Unterschiede zum vorher dargestellten Annotationstypen 4711 sind die MID, die Parameterausprägung sowie der Geltungsbereich. Letzterer bezieht sich auf den Eingabeparameter einer Methode.

Die Langform der Annotation kann nun ebenfalls aufgestellt werden. Sie lautet:

```
/*
 * @@annotation
 * MID: 4712("f", 3)
 * HID: "Motiv f(3): Übergabe Zustand an Beobachter"
 */
```

Abbildung 129: Langform einer Annotation zum Annotationstypen 4712.

Für alle anderen Annotationen in Kurzform aus den Phasen A bis C können analog deren Langform sowie der zugehörige Annotationstyp angegeben werden.

Aliase für Annotationen

Ein Alias für eine Annotation erlaubt es, die Annotation auf eine andere Art und Weise auszudrücken. In der Musterselektion können die Aliase dann beim Abgleich verwendet werden. So macht es Sinn, für die Annotation für Motiv *a* einen Alias zu definieren. Der Alias zum ursprünglichen Motiv »Behandle Beobachterklassen einheitlich« wird angegeben mit »Entferne Abhängigkeit zwischen Beobachtern und Subjekt« (Alias 1). Alle als äquivalent angesehenen sprachlichen Ausgestaltungen dieses Aliases oder des Ursprungsmotivs können ebenfalls als gleichwertiger Alias angegeben werden. Der Alias 1 kann als Problem ausgedrückt werden mit »Abhängigkeit zwischen Beobachtern und Subjekt ist problematisch« (Alias 2). Als Feststellung formuliert lautet Alias 1 »Abhängigkeit zwischen Beobachtern und Subjekt liegt vor« (Alias 3). Ausgehend vom Ursprungsmotiv können weitere Aliase angegeben werden, nämlich »Uneinheitliche Beobachterklassen sind problematisch« (Alias 4), »Beobachterklassen sind uneinheitlich« (Alias 5) und beispielsweise auch »Einheitliche Beobachterklassen benötigt« (Alias 6).

Transformationen und Refactoring-Operationen

Die Quelltexttransformationen und Refactoring-Operationen wurden bereits in den Phasen A bis C genannt. Um die Informationen daraus nutzbar machen zu können, müssen die Transformationen und somit die zugeordneten Refactoring-Operationen auf andere Quelltexte übertragbar sein. Dies gelingt erstens durch Vergleich von Annotationen (samt deren Geltungsbereichen) der Musterdokumentation und eines vorgegebenen annotierten Quelltextes. Zweitens kann eine Refactoring-Operation aus der Musterdokumentation als relevant für einen gegebenen Quelltext angesehen werden, wenn die Vorbedingungen der Refactoring-Operation erfüllt sind. Sind diese nicht erfüllt, liegt das entweder an einem für die Operation ungeeigneten Quelltext oder daran, dass die Refactoring-Operation aufgrund der Gestalt des Quelltextes nicht notwendig ist. Letzterer Fall kann durch Vergleich des gegebenen Quelltextes und des Zielquelltextes in der Musterdokumentation erkannt werden.
Alle Refactoring-Operationen einer Transformation stellen eine Einheit dar. Ist also eine Vorbedingung einer Refactoring-Operation verletzt und liegt das nicht daran, dass die Refactoring-Operation angewandt werden muss, dann können die anderen Refactoring-Operationen derselben Transformation nicht ohne Weiteres ausgeführt werden. Stattdessen bleibt dann nur die Einbeziehung des Anwenders zur Beurteilung möglicher Spielräume.

Transformationen

Eine Transformation kann also nur durch Anwendung aller in ihr enthaltenen und zugleich notwendigen Refactoring-Operationen ausgeführt werden, nicht aber durch Auslassen einer notwendigen Operation. Daher ist es erforderlich, die Zuordnung von Transformation und der sie abbildenden Refactoring-Operationen in der Musterdokumentation zu vermerken. Diese Zuordnung kann einfach durch eine Liste abgebildet werden. Da der Name einer Refactoring-Operation eindeutig zu wählen ist, kann er als Schlüssel für die Operation verwendet werden. Die Transformationen selbst erhalten einen eindeutigen Schlüssel, der beliebig wählbar ist. Es wird vorgeschlagen, eine Kombination aus dem Namen des Musters, in dem die Transformation erstmalig aufgestellt wurde und einem pro Muster bei eins beginnenden Index als Schlüssel für Transformationen zu wählen. Weiterhin wird vorgeschlagen, jeder Transformation als Sekundärindex die Namen aller Refactoring-Operationen der Transformationen, die jeweils mit einem Trennzeichen abgegrenzt werden, zu wählen. Mit dem Sekundärindex kann geprüft werden, ob eine Transformation mit denselben Refactoring-Operationen wie eine neu zu definierende bereits vorhanden ist. Hier wird deutlich, dass die Reihenfolge der Refactoring-Operationen, die im Zuge einer Transformation auszuführen sind, relevant ist. Einerseits kann es sein, dass die Reihenfolge für bestimmte Operationen einer Transformation unwichtig ist, andererseits können andere Operationen dieser Transformation abhängig von einer Reihenfolge sein. Denkbar ist auch, dass eine Operation irgendwann, aber nach einer zweiten und vor einer dritten Operation zu exekutieren ist. Dies ist in geeigneter Weise pro Transformation zu definieren. Es wird vorgeschlagen, für abhängige Operationen eine Spezifikation der Abhängigkeit vorzunehmen.
Weiterhin soll für das Nachvollziehen des Zweckes der Transformation eine informelle Beschreibung in die Deklaration jeder Transformation aufgenommen werden. Die Transformation T3 aus Phase A kann dann etwa so deklariert sein (Abbildung 130):

```
Transformation: Observer-3

Beschreibung: Führt eine Schnittstellen-Methode in eine Klasse ein

Sekundärschlüssel: Add_Interface_Method;Fill_Method_With_Code

Refactoring-Operationen:
   Add_Interface_Method
   Fill_Method_With_Code ←Add_Interface_Method
```

Abbildung 130: Definition einer Transformation mit zugehörigen Refactoring-Operationen.

In Abbildung 130 ist für die Operation *Fill_Method_With_Code* die Abhängigkeit zu *Add_Interface_Method* in der Abbildung durch den nach links gerichteten Pfeil zwischen abhängiger und referenzierter Operation angegeben.
Sofern notwendig, kann für zuerst im Rahmen der Transformation auszuführende Operationen ein Reihenfolgeindex angegeben werden. Dies ist im Beispiel in Abbildung 130 nicht erforderlich, da dort nur zwei Operationen genannt sind, wovon die zweite von der ersten abhängig und alleine schon daraus nach der zuerst für die Transformation deklarierten auszuführen ist.

Um sicherzustellen, dass die Transformation für den über die Musterdokumentation erhaltenen Ausgangsquelltext anwendbar ist, kann dieser Ausgangsquelltext sowie der durch die manuelle Transformation entstandene Zielquelltext in der Transformationsdeklaration mitgegeben werden. Nun kann die Transformation im Rahmen eines Validierungslaufes für den genannten Ausgangsquelltext ausgeführt werden. Entsteht nicht der gegebene Zielquelltext, sondern eine abweichende Form, oder schlägt eine Vorbedingung einer der Refactoring-Operationen fehl, deutet das auf eine fehlerhafte Ausprogrammierung der Refactoring-Operationen zur Transformation oder auf eine inkorrekte Transformationsdeklaration hin.

Refactoring-Operationen

Refactoring-Operationen als Bestandteile von Transformationen wurden bereits abstrakt im Rahmen der Phasen A bis C benannt. Eine Refactoring-Operation kann als Funktion verstanden werden, die bestimmten Ausgangsquelltexten davon abhängige Zielquelltexte zuordnet. Die Regelung der Zuständigkeit für Ausgangsquelltexte geschieht mit Hilfe von Geltungsbereichen sowie von Vorbedingungen. Geltungsbereiche sind ausreichend in Phase A bis C definiert worden. Vorbedingungen wurden textuell beschrieben. Im Endeffekt muss diese informelle Beschreibung in Form einer Prüflogik implementiert und der Operation zugeordnet werden. Eine solche Prüflogik sollte aufgrund ihrer beliebigen Komplexität nach Meinung des Autors durch vollwertige Java-Methoden umgesetzt werden und nicht durch weniger leistungsfähige Scriptsprachen.
Die Implementierung der Prüflogik soll an dieser Stelle nicht angegeben werden. Sie beinhaltet letztendlich die Auswertung des abstrakten Syntaxbaumes, der für die Stelle im Ausgangsquelltext ermittelt werden kann, für die eine Annotation als Zuständigkeitskennzeichnung für die Refactoring-Operation vorliegt. Die Durchführung der Operation selbst geschieht durch Umformung des abstrakten Syntaxbaumes und somit des dadurch repräsentierten Quelltextes. Eine Möglichkeit für die Beschreibung der Aktivitäten von *Introduce_Interface* ist:

1. Ermittle abstrakten Syntaxbaum der Klasse, in die die Schnittstelle einzuführen ist.
2. Ermittle Teil des abstrakten Syntaxbaums, der die Klassensignatur repräsentiert (dass es sich um die Klassensignatur handeln muss, geht aus Ausgangs- und Zielquelltext sowie den Annotationen hervor, die bei der exemplarischen Ausführung der Refactoring-Operation vorlagen).
3. Modifiziere Teil des abstrakten Syntaxbaums aus Schritt 2 durch Ergänzung einer Schnittstellenimplementierung.
4. Generiere Quelltext für den Teil des abstrakten Syntaxbaums aus Schritt 3.
5. Setze Quelltext anstelle des ursprünglichen im Ausgangsquelltext ein und realisiere somit die Schnittstelleneinführung.

Dieser Pseudocode ist durch Aufbau eines geeigneten Frameworks bzw. einer Bibliothek von Hilfsroutinen zu implementieren. Beispiele für sinnvolle Hilfsroutinen für *Introduce_Interface* sind:

- Ermittlung des abstrakten Syntaxbaums zu einer Programmentität,
- Ermittlung des abstrakten Syntaxbaums, der eine bestimmte Programmentität repräsentiert oder der durch eine bestimmte Annotation gekennzeichnet ist,
- Hinzufügen einer Schnittstellenimplementierung für eine Klasse auf einem abstrakten Syntaxbaum sowie
- Generierung von Quelltext aus einem gegebenen abstrakten Syntaxbaum.

3.3.6.2 Verfeinerung der formalen Musterdokumentation

Dieser Abschnitt beschreibt, wie die zuvor aufgestellte Musterdokumentation leistungsfähiger gestaltet werden kann.

Ermittlung unabhängiger Motive

Die Identifikation von von einander unabhängigen Motiven ist hilfreich bei der Auswertung von Annotationen im Rahmen der Musterselektion. Sind zwei Motive unabhängig von einander, so drücken sie unterschiedliche Lösungsideen aus, die getrennt von einander existieren können. Im Falle von *Observer* bedeutet das, das Muster realisiert verschiedene Lösungen, von denen jede (unabhängige) für sich bereits lohnenswert wäre.

Von den gefundenen Motiven *a* bis *g* sind einige Motive von anderen unabhängig. Eine Unabhängigkeit ist etwa für mehrere Tupel der Motive *a*, *b*, *c* und *d* gegeben. Motiv *a* ist unabhängig von Motiv *d*, weil ein einheitlicher Registrierungsmechanismus auch ohne einheitliches Behandeln von Beobachtern umsetzbar ist. Umgekehrt gilt dies allerdings nicht.

Motiv *a* ist unabhängig von Motiv *c*, weil die Vereinheitlichung von Beobachterklassen nicht im direkten Zusammenhang mit der Umsetzung einer Registrierungsmöglichkeit zu beliebigem Zeitpunkt steht, und umgekehrt.
Motiv *c* ist unabhängig von Motiv *d*, weil die Registrierung eines Beobachters zu beliebigem Zeitpunkt auch ohne einen einheitlichen Registrierungsmechanismus umsetzbar ist, und umgekehrt.
Ferner ist Motiv *b* von Motiv *a* aus demselben Grund unabhängig wie bei Motiv *c* und *a*.

Definition von synonymen Motiven

Für die gefundenen Motive sollten Synonyme definiert werden. Dies ist insbesondere sinnvoll, weil es verschiedene Sichtweisen auf einen Quelltext geben kann. Die Sicht des Dokumentators eines Musters ist lösungsorientiert, wohingegen die Sicht des Entwicklers, der seinen Quelltext annotiert, auch problemorientiert sein kann. So kann beispielsweise zum lösungsorientierten Motiv *b* »Erlaube es, beliebig viele Beobachter zu registrieren« ein problemorientiertes Synonym mit »Direkte Abhängigkeit zwischen Subjekt und Beobachter« benannt werden.

Definition von Isomorphen

Die formale Musterdokumentation enthält bis hierhin die Deklaration von Refactoring-Operationen auf Basis von Programmentitäten und Annotationen. Um auch Programmentitäten durch eine Operation transformierbar zu machen, die nicht mit denen im beispielhaften Ausgangsquelltext gegebenen exakt übereinstimmen, sind isomorphe Programmentitäten zu definieren.
Durch die Definition von Isomorphen können unterschiedliche Quelltextteile mit gleicher Bedeutung vergleichbar gemacht werden. Isomorphe können durch Angabe von synonymen abstrakten Syntaxbäumen für einen anderen abstrakten Syntaxbaum definiert werden.
Somit kann eine Refactoring-Operation alle abstrakten Syntaxbäume, die als Synonym für einen abstrakten Syntaxbaum definiert wurden, intern durch letztgenannten Syntaxbaum ersetzen und somit vereinheitlichen. Die Menge der transformierbaren Programmentitäten steigt in dem Maße wie Synonymdefinitionen vorliegen. Stellt sich heraus, dass ein Synonym nur bei Vorliegen bestimmter Bedingungen, etwa einer als final deklarierten Klasse, gültig ist, ist dies durch eine Vorbedingung auszudrücken. Eine solche Vorbedingung kann analog definiert werden wie für Refactoring-Operationen.
Durch Vergleich der in der ersten Verfeinerungsphase B eingeführten Variation des Ausgangsquelltextes können Isomorphe mitunter entdeckt werden. Der Dokumentator kann die Variation bewusst so wählen, dass sie Isomorphe offensichtliche enthält, beispielsweise einen Iterator anstatt einer *for*-Schleife.

Für Annotationen kann analog zu für Java originären Programmentitäten eine Synonymdefinition vorgenommen werden. Dies ist beispielsweise sinnvoll um unterschiedliche Betrachtungswinkel auf einen Quelltext zu berücksichtigen. Während der Dokumentator in der Musterdokumentation Motive zum Ausdrücken von Lösungen anbringt, wird der Entwickler, der einen Quelltext zwecks Musterselektion und -anwendung annotiert, ggfs. Motive zum Ausdrücken von Problemen wählen. Beispielsweise könnte das vom Dokumentator gewählte Motiv *a* »Behandle Beobachterklassen einheitlich« vom Entwickler stattdessen mit »Beobachterklassen werden nicht einheitlich behandelt« benannt werden. Die Definition von Synonymen für Annotationen bzw. deren zugrunde liegenden Motiven erlaubt das homogene Behandeln dieser verschiedenen Betrachtungswinkel.

Unterteilen von Transformationen

Prinzipiell erhöht es die Wahrscheinlichkeit der Wiederverwendbarkeit und Anpassbarkeit von Transformationen bei Muster- oder Quelltextvariationen, wenn Transformationen so feingranular wie möglich aufgestellt werden. Die Aufteilung von Transformationen ist ein manueller Prozess, bei dem sinnvolle Aufteilungen zu eruieren sind. Eine Aufteilung macht insbesondere Sinn, wenn eine ganze Klasse (beispielsweise eine abstrakte Basisklasse) oder eine Schnittstelle (wie *IObserver* für das *Observer*-Muster) neu einzufügen sind. Alle Methoden innerhalb einer solchen Klasse oder Schnittstelle können sinnvollerweise als Teiltransformationen ausgedrückt werden. So kann die gesamte Transformation nach Anpassung einer Teiltransformation wiederverwendet werden, wie

dies in Phase B gezeigt wurde. Weitere mögliche Variationspunkte betreffen Logik innerhalb von Blöcken. Blöcke können in Java durch geschweifte Klammern gebildet werden oder durch Blockanweisungen (wie *if* oder *for*) sowie durch Methoden (der Methodenrumpf bildet einen Block). Es obliegt dem erfahrenen Entwickler, ähnlich der Muster *Template Method* oder *Strategy* [Gamma+ 1995] Anweisungssequenzen sinnvoll in neu einzuführende Methoden auszulagern.

Validierung von Transformationen

Um die Gefahr von Transformationen, die ein inkorrektes Ergebnis oder einen unkompilierbaren Quelltext erzeugen zu reduzieren, können optional Nachbedingungen implementiert werden. Diese prüfen nach Ausführung der Transformation, ob erwartete Charakteristika des Zielquelltextes vorhanden sind. Solche Merkmale können beispielsweise bestimmte Klassen- oder Methodensignaturen oder bestimmte abstrakte Syntaxbäume innerhalb einer Methode sein. Schlägt eine Nachbedingung fehl, kann dies dem Anwender gemeldet werden. Weiter gehend, kann eine Korrekturlogik umgesetzt werden, die auf bestimmte Fehlsituationen reagieren kann, indem der vorliegende und als inkorrekt erkannte abstrakte Syntaxbaum modifiziert wird. Die Umsetzung einer Korrekturlogik obliegt dabei dem Intellekt eines kompetenten Entwicklers.
Nachbedingungen können analog zu Vorbedingungen als vollwertige Java-Methoden und inter Zuhilfenahme geeigneter Hilfsroutinen implementiert werden.

3.3.6.3 Fazit

Die Kombination von Quelltext und Annotationen erlaubt die dedizierte Beschreibung sowohl der Struktur als auch des Verhaltens eines Entwurfsmusters in Form einer formalen Dokumentation. Grundlage dafür ist die gleichzeitige Möglichkeit, vorliegenden Quelltext durch Annotationen zusätzlich zur durch die Programmelemente beschriebenen Struktur im Verhalten zu beschreiben.

In diesem Abschnitt wurde erläutert, wie die in den Phasen A bis C gewonnenen Informationen genutzt werden können, um eine formale Musterdokumentation aufzustellen. Insbesondere wurde diskutiert, wie das Grundgerüst einer Musterdokumentation aussehen kann und welche Teile es auf Basis der bisherigen Aktivitäten aus den Phasen A bis C enthält. Weiterhin wurde beschrieben wie Transformationen samt der zugehörigen Refactoring-Operationen sowie Abhängigkeiten zwischen den Operationen deklariert werden können.
Zu den Refactoring-Operationen wurden keine Implementierungsdetails angegeben. Stattdessen wurde ergänzend zu deren informeller Beschreibung aus den Phasen A bis C beschrieben, wie Implementierungen für Refactoring-Operationen sowie Vor- und Nachbedingungen gestaltet werden können. Weiterhin wurde die Isomorphendefinition für Programmentitäten und Annotationen und deren Nutzen für die Musterselektion und -anwendung erläutert.
Die Vorschrift, alle im Rahmen der Musterdokumentation transformierten Quelltextteile mit Motiven zu versehen, lässt den Schluss zu, dass die nicht mit Motiven bzw. Annotationen versehenen Teile eines Ausgangsquelltextes im Rahmen der Musterdokumentation nicht relevant sind.

3.3.7 Phase E: Musterselektion

Im Rahmen der Musterselektion wird für einen gegebenen Ausgangsquelltext festgestellt, welche Muster auf diesen Quelltext sinnvollerweise angewandt werden können. Für die Selektion können die Muster berücksichtigt werden, die durch Musterdokumentation im Musterkatalog aufgenommen wurden. Verschiedene Muster treten bei der Musterselektion ebenso in Konkurrenz zueinander wie unterschiedliche Varianten eines Musters.

Die Musterselektion hängt eng mit der Musteranwendung zusammen. Einerseits muss bereits in der Musterselektion geprüft werden, ob ein Muster anwendbar ist. Denn nur später anwendbare Muster können sinnvoll für eine Anwendung selektiert werden. Andererseits sind die in der Musterselektion ermittelten Informationen, warum ein Muster geeignet erscheint, für die Musteranwendung relevant. Denn die Musteranwendung benötigt Anhaltspunkte, welche Stellen des

Ausgangsquelltextes wie zu transformieren sind. Definitionen der Transformationen wiederum sind bereits in der Musterdokumentation enthalten.

An dieser Stelle soll ein schon bekannter Quelltext als Grundlage dienen, um das Prinzip des Verfahrens bei der Musterselektion zu demonstrieren. Bisher aufgestellte Ausgangsquelltexte sind die aus Phase A bzw. C (Abbildung 70) und B (Abbildung 104). Beispielhaft wird der Quelltext aus Phase A herangezogen und leicht modifiziert (Abbildung 131).

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Observierer
040        PatDetails bo2 = new PatDetails();
050        // Konstruiere Observierten und registriere Observierer
060        PatSelector bo1 = new PatSelector(bo2);
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110    public class PatDetails {
120      public void refresh(Object state) {
130        // Aktualisiere Zustand und Darstellung
140        // ...
150        System.out.println("Reagiere auf Zustandsänderung von PatSelector");
160      }
170    }

200    public class PatSelector {
210      // Der Observierende
220      private PatDetails dependent;

230      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
240      // benötigt. Er besteht etwa aus einer Handvoll Variablenbelegungen).
250      private Object state;

260      // Konstruiere Objekt und registriere einen Observierenden
270      public PatSelector(PatDetails dependent) {
280        this.dependent = dependent;
290      }

300      public void start() {
310        //... tue irgendwas hier ...

320        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
330        doWorkChangingState();
340        //... tue auch noch irgendwas anderes hier ...
350      }

360      public void doWorkChangingState() {
370        System.out.println("Ich tue etwas, was meinen Zustand verändert");
380        System.out.println("Deshalb informiere ich PatDetails darüber");
390        dummyMethod();
400      }
410      private void dummyMethod() {
```

```
420         dependent.refresh(state);
430       }
440     }
```

Abbildung 131: Quelltext für die Prüfung der Tauglichkeit des Verfahrens.

Unterschiede in Abbildung 131 gegenüber dem Ausgangsquelltext aus Phase A sind die geänderte Zeile 340 des Ausgangsquelltextes (neue Zeilennummer 390) und die neu eingeführten Zeilen 410 bis 430. Einige Kommentare wurden entfernt. Die Zeilennummern weichen ab Zeile 110 vom ursprünglichen Quelltext ab, weil Klasse *PatDetails* vor Klasse *PatSelector* gezogen wurde.
Die programmtechnische Änderung des Originalquelltextes (Zeile 340) besteht darin, dass statt des direkten Aufrufs des Beobachters zuerst eine private Methode gerufen wird, die den Beobachter letztendlich benachrichtigt. In der Praxis wird dieses Vorgehen wenig Sinn machen, zumindest nicht, wenn wie in Methode *dummyMethod* nur eine Anweisung steht. Das Beispiel nimmt hierauf keine Rücksicht, da es alleine die Verarbeitung eines Quelltextes demonstrieren will, der in seiner Gesamtheit noch nicht bekannt war. Zudem kommt es auch in der Praxis oft vor, dass eigentlich unsinnige Programmierungen vorgenommen werden, sei es aufgrund mangelnder Kompetenz des Entwicklers oder aufgrund von Zeitdruck.
Die in den späteren Tabelle 18 und Tabelle 19 genannten Ankerpunkte sind in Abbildung 131 durch unterstrichene Zeilen angegeben.

Anhand dieses Quelltextes soll gezeigt werden, wie erkannt werden kann, dass der Quelltext geeignet ist, um das *Observer*-Muster dafür auszuwählen. Dazu wird zuerst der Quelltext mit Annotationen versehen. Weiterhin wird demonstriert, wie für die Anwendung von *Observer* durchzuführende Transformationen ermittelt und ausgeführt werden können.

3.3.7.1 Annotation des Quelltextes

In Rahmen der formalen Musterdokumentation wurden alle Quelltextteile mit Annotationen versehen, die von Transformationen betroffen waren. Es gilt nun, für einen gegebenen Ausgangsquelltext wie in Abbildung 131 möglichst die Stellen zu identifizieren, die bei einer Musteranwendung von einer Transformation betroffen wären. Im Idealfall bedeutet dass, das genau die Stellen im Ausgangsquelltext mit Annotationen versehen sind, die für die Musterselektion und die Musteranwendung relevant sind. Zwei Arten von Annotationen sind hierbei zu unterscheiden. Erstens sind dies Annotationen, die ein Problem des Anwenders ausdrücken, das er durch Musteranwendung gelöst haben möchte. Zweitens sind dies Annotationen, die unterstützend zu erst genannten Annotationen vorhanden sein sollten, um im Rahmen der Musterselektion eine größtmögliche Übereinstimmung mit den Annotationen einer formalen Musterdokumentation eines potentiell anwendbaren Musters zu ermöglichen.
Ausgangspunkt der Annotation eines Ausgangsquelltextes ist das Identifizieren des nichtfunktionalen Problems mit dem Quelltext. Nachdem das Problem identifiziert wurde, ist zumindest eine Stelle im Quelltext zu finden, die das Problem verursacht. Diese Stelle ist zu annotieren.

Weil vor der Musterselektion nicht bekannt ist, welches Muster selektiert und angewandt werden wird, können nur Handlungsanweisungen gegeben werden, wie ein gegebener Quelltext zu annotieren ist, um Musterselektion und -anwendung dafür erfolgreich durchführen zu können.
Für Erzeugungsmuster ist die Objekterzeugung von grundlegender Bedeutung. Das heisst, für ein Erzeugungsmuster muss ein Konstruktoraufruf oder die Deklaration eines Konstruktors annotiert werden. Welcher Konstruktor oder Konstruktoraufruf zu annotieren ist, hängt davon ab, welche Anforderungen der Anwender an die Verbesserung seines Quelltextes hat. Soll etwa eine Klasse *X* nur maximal einmal erzeugt werden, so ist der Konstruktor der Klasse *X* oder der Aufruf dieses Konstruktors mit einer Annotation zu versehen. Korrespondiert diese Annotation etwa mit einer aus der Musterdokumentation zum *Singleton*, so ist es möglich, *Singleton* als anzuwendendes Muster zu selektieren. Ist stattdessen die Flexibilisierung der Erzeugung einer Klasse *Y* vom Anwender gewünscht, so muss er eine andere Annotation, die dies ausdrückt, für einen Konstruktoraufruf oder

den Konstruktor von Klasse *Y* anbringen. Findet sich etwa in der Musterdokumentation zu *Factory Method* oder zu *Abstract Factory* eine korrespondierende Annotation, können beide Muster als anwendbar vorgeschlagen werden.
Schwieriger ist das Annotieren von Quelltext mit der Möglichkeit, andere Arten von Mustern als Erzeugungsmuster zu selektieren. Denn im Gegensatz zu Erzeugungsmustern basieren Mustertypen wie Struktur-, Verhaltens- oder Architekturmuster auf nicht leicht im Quelltext identifizierbaren Kriterien wie der Objekterzeugung, für die es in Java dedizierte Konstrukte, nämlich Konstruktoren sowie den *new*-Operator, gibt.

Annotationswürdige Stellen im Quelltext

Für die Musterselektion werden Annotationen im Ausgangsquelltext mit solchen in der Musterdokumentation abgeglichen. Daher sind a priori nur Programmstellen im Ausgangsquelltext für eine Annotation interessant, die auch in der Musterdokumentation annotiert wurden. Daher soll rückwärts gehend von den gefundenen Annotationen in der Musterdokumentation zu *Observer* im Folgenden entwickelt werden, welche Programmentitäten im Besonderen einer Annotation würdig sind und welche nicht.

Die Motivation für die Einführung eines Musters kann verschiedene Ursachen haben. Zum einen kann ein generelles Entwurfsproblem vorliegen, das elegant gelöst werden soll. Beispielsweise kann dies die Abbildung der Möglichkeit sein, mehrere Beobachter statt nur einem in einer generischen Weise zu unterstützen, so wie dies *Observer* realisiert. Zum anderen kann für eine konkrete Programmentität eine nichtfunktionale Anforderung identifiziert worden sein, die es gilt umzusetzen. Beispiel hierfür ist die performantere Umsetzung einer bereits gegebenen Logik.

Für Entwickler, die den Ausgangsquelltext annotieren, ist zu empfehlen, dass sie sich zunächst vergegenwärtigen, welche Programmstellen im Besonderen von dem aktuellen Entwurfsproblem bzw. der gewünschten nichtfunktionalen Verbesserung betroffen sein könnten. Im Falle eines Problems bzw. einer Anforderung mit konkretem Bezug auf eine Programmentität ist eine Annotation für eben diese Programmentität vorzunehmen. Soll stattdessen ein allgemeines Entwurfsproblem gelöst werden, ist zunächst eine Anweisung oder Deklaration zu finden, die als Aufhänger für das Entwurfsproblem anzusehen ist. Auch in der Forward Engineering Phase können solche Anweisungen oder Deklarationen bereits umgesetzt sein, wenn von einer Quelltext-getriebenen Entwicklung ausgegangen wird oder ein durch ein Modell erstelltes Quelltextgenerat vorliegt.

Das Entwurfsproblem im Ausgangsquelltext aus Abbildung 131 liegt insbesondere in der direkten Abhängigkeit zwischen den Klassen *PatSelector* und *PatDetails* begründet, welche sich an den Zeilen 220 und 270 der Abbildung feststellen lässt. Dies kann entweder manuell durch Betrachten des Quelltextes oder ohne Weiteres automatisch durch ein Analysewerkzeug herausgefunden werden.
Demnach sind die beiden genannten Zeilen zu annotieren. Es reicht hier im Prinzip aus, Zeile 270 zu annotieren, da in Zeile 280 auf die Variable zugegriffen wird, die in Zeile 220 deklariert wurde (ebenfalls durch herkömmliche Quelltext-Analysewerkzeuge vollständig automatisch ermittelbar).
Da Zeile 270, die Deklaration des Konstruktors, annotiert wurde, kann auch der Aufruf des Konstruktors in Zeile 060 analog annotiert werden. Dies kann aufgrund der generellen Abhängigkeit zwischen Deklaration und Aufruf automatisiert geschehen.
Ein nicht direkt ins Auge springendes Entwurfsproblem im Ausgangsquelltext ist die Tatsache, dass der Beobachter nur zu einem Zeitpunkt, nämlich bei der Konstruktion des Subjektes, beim Subjekt registriert werden kann. Dieses Entwurfsproblem wird, ebenso wie das vorige, durch *Observer* gelöst.

Die Annotation der Zeile 270 könnte folgendes Bild ergeben (Abbildung 132):

```
      /**@@motiv Einheitliche Beobachterklassen benötigt*/
270   public PatSelector(PatDetails dependent) {
```

Abbildung 132: Angebrachte Annotation für die Musterselektion.

Die angebrachte Annotation entspricht einem in Abschnitt 3.3.6.1 definierten Alias. Lediglich die Zuordnung zu Motiv *a* fehlt in Abbildung 132, weil eine derartige Motivbenennung nur im Rahmen der Musterdokumentation geleistet wird.

Weil in der Musterdokumentation die Motive *a* und *c* als unabhängig von einander deklariert wurden, ist es nicht erforderlich, dass Annotationen für beide Motive vorhanden sind. Unabhängige Motive drücken unterschiedliche Lösungsideen aus, die getrennt von einander existieren können. Analog verhält es sich mit den Motivpaaren *a* und *d* bzw. *c* und *d*.
Die Schwierigkeit bei der Annotation von Zeile 270 besteht darin, eine für den Abgleich mit der Musterdokumentation geeignete Annotation zu finden. Um dies zu unterstützen, kann in der verwendeten Entwicklungsumgebung eine Hilfestellung für die Annotierung von Quelltext eingebaut werden (in dem Java-Werkzeug *Eclipse* etwa mittels Plugins). Diese Hilfestellung bietet dem Entwickler für eine Anweisung nur die Annotationen an, die in mindestens einer Musterdokumentation vorkommen und einen zur Anweisung passenden Geltungsbereich haben. Siehe hierzu auch Abschnitt 3.4.

Im Ausgangsquelltext aus Abbildung 131 müssen Klassen und Schnittstellen nicht annotiert werden. Denn diese wurden auch nicht in der Musterdokumentation mit Annotationen bedacht. Es schadet allerdings nicht, diese Annotation dennoch vorzunehmen, da überschüssige Annotationen beim Abgleich der Annotationen nicht negativ ins Gewicht fallen (sie stimmen ja selbst und hinsichtlich der annotierten Programmentität mit der Musterdokumentation überein und können somit ignoriert werden). Generell hilft es sogar, pro Klasse jede der Rollen, die die Klasse einnehmen kann, durch Annotation zum Ausdruck zu bringen. Wenn in der Musterdokumentation analog verfahren wird, können so Fälle behandelt werden, in denen etwa eine Schnittstelle im Ausgangsquelltext bereits vorhanden ist oder in einer im Vergleich zum Muster modifizierten Form.
Weiterhin können insbesondere Hilfsroutinen (Aufruf sowie Deklaration) ignoriert werden, wenn der Entwickler sicher ist, dass diese bei der aktuellen Problemstellung irrelevant sind.

Automatisiertes Anbringen von Annotationen

Zusätzlich zu den manuell anzubringenden Annotationen können bestimmte Annotationen automatisiert im Ausgangsquelltext angebracht werden. Für den Quelltext aus Abbildung 131 kann aus der in Abbildung 132 angegebenen Definition die Annotation der Zeile 220 gefolgert werden. Dies ist möglich, da sowohl Zeile 270 als auch Zeile 220 eine Referenz auf die Beobachterklasse *PatDetails* besitzen und zudem in der Musterdokumentation der Motivaspekt »Beobachter« definiert wurde.

Annotationen können insbesondere automatisiert für Programmelemente angebracht werden, die mit bereits annotierten Programmelementen korrespondieren. Beispielsweise kann eine Annotation für einen Methodenaufruf angebracht werden, wenn die gerufene Methode selbst annotiert ist. Ein Methodenaufruf setzt nämlich voraus, dass die Methode implementiert ist. Ist die Methode Teile einer Schnittstelle, muss die Schnittstelle auch in der Klasse deklariert sein. Der Methodenaufruf selbst ist der dritte notwendige Teil der Implementierungsstücke. Ist nur die Deklaration einer Methode annotiert, kann darauf eine Annotierung des Methodenaufrufs abgeleitet werden. Im einfachsten Fall wird die Annotation der Methodendeklaration kopiert, so dass der Methodenaufruf identisch annotiert ist. Vorausgesetzt, die Annotation der Methodendeklaration drückt aus, welche Bedeutung hinter der Methode steckt, ist das Annotieren des Methodenaufrufs in identischer Weise wie das Annotieren der Methodendeklaration durchaus angebracht.

Für Erzeugungsmuster sind insbesondere Programmstellen relevant, an denen Objekte erzeugt werden. Diese Stellen lassen sich in Sprachen wie Java oder C# durch Suche nach Konstruktoraufrufen finden. Wenn der Anwender mit Hilfe eines hierarchisch gegliederten Fragenkatalogs (vgl. [Tan+ 2003]) geführt wird, etwa von der Abfrage der Problemklasse (Problem mit der Geschwindigkeit, mit der Objekterzeugung, mit der Wiederverwendbarkeit etc.) bis hin zu Detailfragen (etwa Mustervarianten betreffend), kann eine Annotation aus der Menge der mit Hilfe der Fragen eingeschränkten Muster(dokumentationen) als Vorschlag präsentiert werden.

Weiterhin können Annotationen für Programmentitäten automatisiert ermittelt werden, die exakt definierten Konventionen gehorchen. Beispielsweise kann festgelegt werden, dass die Java-Schnittstellen *java.util.List* und *java.util.Map* sowie deren Implementierungen Datenkollektionen darstellen. Dies kann nützlich sein für Muster die auf Datenkollektionen basieren, wie etwa *Composite* (vgl. auch Anhang B).

Als wahrscheinlich häufig wiederkehrend angesehene Quelltextteile können zusammen mit einer als sinnvoll erachteten Annotation in einem Repository gespeichert werden. Quelltextteile sind hier nicht nur Anweisungen oder Anweisungssequenzen, sondern auch Programmteile in unterschiedlichen Kontexten (Klassen, Methoden) eines Programms, die durch Referenz o.ä. in Zusammenhang stehen.

3.3.7.2 Abgleich von Annotationen

Der Abgleich von Annotationen bedeutet den Vergleich von Annotationen im Ausgangsquelltext mit denen je einer Musterdokumentation mit dem Ziel, die Eignung eines dokumentierten Musters mit dem Ausgangsquelltext festzustellen. Pro Musterdokumentation wird dieser Abgleich durchgeführt, so dass als Ergebnis pro dokumentiertem Muster ein Grad an Übereinstimmung ermittelt werden kann. Annotationen sind ein starker Indikator für oder gegen ein Muster, weil sie explizit und mit definierter Semantik angebracht werden. Im Gegensatz dazu stehen originäre Programmentitäten, die prinzipiell kontextfrei sind und denen nicht ohne Weiteres eine spezielle Bedeutung beigemessen werden kann.

Als geeignet empfundene Muster werden im Rahmen der Musterselektion für die Anwendung vorgeschlagen. In Abbildung 131 wurde Zeile 270 direkt annotiert. Diese Annotation kann automatisiert für die Zeilen 280 und 220 übernommen werden, weil der Eingabeparameter aus Zeile 270 in Zeile 280 einer Klassenvariablen zugewiesen wird und letztere Variable in Zeile 220 deklariert wird. Die Abhängigkeit dieser Programmelemente kann mit Hilfe der Motive und Motivaspekte in der Musterdokumentation sowie durch Ausbeuten der generellen Abhängigkeiten in der Sprache Java (Deklaration und Aufruf bzw. Zuweisung oder Referenz) und der explizit festgelegten Relationen von Programmentitäten eines Motivs festgestellt werden. Jedes Motiv, das in der Musterdokumentation in mehreren Teilen vorhanden ist, zeigt auf, dass die mit dem Motiv annotierten Programmentitäten korreliert sind. Für Motiv *a* sind zur Veranschaulichung in der folgenden Abbildung die im Ausgangsquelltext annotierten Programmzeilen angegeben:

```
060 PatSelector bo1 = new PatSelector(bo2);

130 private PatDetails dependent;

180 public PatSelector(PatDetails dependent) {
190 this.dependent = dependent;

370 public class PatDetails {
```

Abbildung 133: Abhängige Programmentitäten zu einem Motiv.

Zeile 060 und Zeile 370 hängen von einander ab, weil in Zeile 060 die Variable *bo2* vom Typ *PatDetails* verwendet wird, der in Zeile 370 deklariert ist. Zeile 060 und Zeile 180 hängen von einander ab, weil Zeile 060 den Konstruktor, der in Zeile 180 deklariert ist, ruft. Zeile 180 und Zeile 370 sind korreliert, weil in Zeile 180 eine Referenz auf den in Zeile 370 eingeführten Typen *PatDetails* vorkommt. Zeile 180 und Zeile 190 hängen zusammen, weil der Eingabeparameter *dependent* (Zeile 180) in Zeile 190 auf der rechten Seite der Zuweisung steht. Letztere Zeile besitzt eine Abhängigkeit zu Zeile 130, weil die dort deklarierte Variable einen Wert zugewiesen bekommt.

Mit Hilfe der in der Musterdokumentation zu *Observer* vorliegenden Annotationen sowie von festgelegten Aliasen für diese Annotationen kann nun ein Vergleich vorgenommen werden. Es wird vorausgesetzt, dass der vorliegende Quelltext kompilierbar ist.

Der Vergleich zweier Annotationen ist erfolgreich, wenn entweder beide Annotationen gleich sind oder eine Übereinstimmung auf Basis von Aliasen gefunden werden kann. Weiterhin müssen die mit den verglichenen Annotationen versehenen Programmentitäten zu dem zulässigen Geltungsbereich der jeweiligen Annotation passen. Da ein und dieselbe Annotation in der Musterdokumentation und im Ausgangsquelltext an mehreren verschiedenen Stellen zum Einsatz kommen kann, müssen ferner die mit den zu vergleichenden Annotationen versehenen Programmentitäten bezüglich ihres abstrakten Syntaxbaumes übereinstimmen oder sich durch eine isomorphe Anweisung ersetzen lassen, so dass eine Übereinstimmung gegeben ist. Das hat zur Konsequenz, dass alle nicht durch Isomorphe abbildbaren Fälle in der Musterdokumentation als Variationen des Musters festzuhalten sind.

Der Vergleich der Annotationen im Beispiel dieses Abschnitts mit denen der Musterdokumentation von *Observer* liefert folgendes Ergebnis. Die direkt annotierte Zeile 270 stimmt mit dem Alias der Annotation in der Musterdokumentation überein. Daraus kann abgeleitet werden, dass sich die Annotation auf Motiv *a* bezieht. Aufgrund der automatisierten Annotation der Zeilen 220 und 280 ist deren Abhängigkeit zu Zeile 180 bekannt. Daraus ergibt sich dasselbe Motiv für die Zeilen 280, 220 sowie 060.

Ermittelte Übereinstimmungen zwischen Dokumentation des selektierten Musters und gegebenem Ausgangsquelltext werden hier Ankerpunkte genannt. Ein Ankerpunkt verknüpft eine Programmentität des Ausgangsquelltextes mit einer Transformation in der Musterdokumentation.

Die eben für *Observer* ermittelten Ankerpunkte sind (Tabelle 18):

Nr.	Zeile Musterdoku	Zeile Quelltext	Motiv
1	060	060	a
2	130	220	a
3	180	270	a
4	190	280	a

Tabelle 18: Ankerpunkte für Motiv a bei der Musterselektion.

Beim Vergleich der in Tabelle 18 genannten Programmzeilen von Quelltext der Musterdokumentation (»Zeile Musterdoku«) und gegebenem Quelltext (»Zeile Quelltext«) fällt auf, dass kein Unterschied existiert. Mit Hilfe der Annotationen konnte dieser Vergleich einfach ausgeführt werden, da damit von Zeilennummern abstrahiert werden konnte. Das *Observer*-Muster kommt also aufgrund dieses Vergleichs für den gegebenen Ausgangsquelltext in Betracht.

Anhand einer alternativen Annotation soll gezeigt werden, wie sich die Musterselektion für den gegenüber dem Ausgangsquelltext der Musterdokumentation geänderten Quelltextteil gestaltet. Es handelt sich um die Programmzeile 390 bis 430 aus Abbildung 131. In Zeile 420 wird ein einzelner Beobachter gerufen. Dies ist eine kritisch zu bewertende Implementierung, weil eine direkte Abhängigkeit zwischen Subjekt und einem konkreten Beobachter existiert. Diese Zeile könnte wie folgt annotiert werden:

```
410  private void dummyMethod() {
419    /**@@motiv b: Direkte Abhängigkeit zwischen Subjekt und Beobachter*/
420    dependent.refresh(state);
430  }
```

Abbildung 134: Beispielhafte Annotation eines Methodenaufrufs.

Alternativ könnte die Zeilen 410 oder 390 gleichsam annotiert werden, denn auch diese Zeilen sind (indirekt) für die Benachrichtigung des konkreten Beobachters verantwortlich. Dies kann auch automatisch gefolgert werden. Dazu muss eine Regel vorgegeben werden, die die Annotation einer Anweisung, die sich alleinig in einer Methode befindet (Zeile 420), auf die Methode selbst (Zeile 410) und den Aufruf dieser (Zeile 390) überträgt.

Die Ankerpunkte zu diesem Motiv lauten dann gemäß Tabelle 19 (Nummerierung von Tabelle 18 fortgesetzt):

Nr.	Zeile Musterdoku	Zeile Quelltext	Motiv
5	340	420	b
6	340	410	b
7	340	390	b

Tabelle 19: Ankerpunkte für Motiv b bei der Musterselektion.

Die in Spalte »Zeile Musterdoku« von Tabelle 19 genannten Zeilennummern sind allesamt identisch. Das erklärt sich daraus, dass aus einer Zeile im ursprünglichen Quelltext durch Einführung einer neuen Methode um eine bestehende Anweisung herum drei Zeilen wurden (Aufruf neuer Methode, Deklaration neuer Methode, Aufruf ursprünglicher Anweisung). Da in der Musterdokumentation zur Annotation von Zeile 340 nur eine Geltungsbereichsdefinition existiert, kann es sein, dass nicht alle Annotationen im Quelltext damit übereinstimmen. Somit wird ein bisher nicht vorgesehener Fall erkennbar. Der Anwender kann so darauf aufmerksam gemacht werden, dass eine vorzunehmende Quelltexttransformation mit einer gewissen Wahrscheinlichkeit fehlschlägt. Der Grund kann dem Anwender ebenfalls vermittelt werden, nämlich dass aus einer Anweisung drei wurden. Der Anwender muss entscheiden, welche Zeile letztendlich durch die in der Musterdokumentation definierte Transformation T7 ersetzt werden soll. Im Prinzip muss er sich nur zwischen Zeile 390 und

420 entscheiden, da nur diese Methodenaufrufe darstellen und somit dem Geltungsbereich von T7 entsprechen. Der Anwender wählt im Beispiel die Zeile 420. Das hätte allerdings auch automatisiert gefolgert werden können, weil die Methode *dummyMethod* nur eine Anweisung enthält und somit verdichtet werden kann.

Allgemein kann der Grad der Übereinstimmung zwischen gegebenem Quelltext und einer konkret betrachteten Musterdokumentation mit Hilfe der ermittelten Ankerpunkte auf unterschiedliche Weise bestimmt werden. Beim Vergleich der Annotationen können mehrere Kriterien berücksichtigt werden. Diese ergeben sich daraus, ob korrespondierende Annotationen aus dem gegebenen Quelltext in einer bestimmten Musterdokumentation gefunden werden und wenn ja, mit welchem Grad sie übereinstimmen. Weiterhin ist entscheidend, ob Annotationen der Musterdokumentation im gegebenen Quelltext fehlen.
Die Bestimmung einer Gewichtung aus diesen Kriterien kann auf verschiedene Arten vorgenommen werden, ebenso wie die Ermittlung des Grades der Übereinstimmung zweier Annotationen. Letzteres hängt von der Übereinstimmung der Komponenten einer Annotation ab. Im Wesentlichen sind das das Motiv, das die Annotation ausdrückt sowie der abstrakte Syntaxbaum des annotierten Programmelements. Bei im Motiv verschiedenen Annotationen kann die Ähnlichkeit nur manuell entschieden werden, beispielsweise durch sukzessives Aufstellen einer Ähnlichkeitsmatrix für Annotationen. Weiterhin können sprachlich verschieden ausgedrückte Motive durch eine manuell aufzustellende Synonymtabelle als gleich erkannt werden.
Eine Annotation in einem Quelltext, die ein Programmelement annotiert das laut Annotationsdefinition nicht erlaubt ist, sind vom System als ungültig zu kennzeichnen und abzulehnen. Für diese Annotation findet keine Gewichtung statt, der Prozess sollte dadurch abgebrochen werden. Ungültige Annotationen sollten vielmehr durch das zur Annotation verwendete Werkzeug (etwa die integrierte Entwicklungsumgebung) unterbunden werden. Gleiches gilt für Annotationen, deren Motiv gänzlich unbekannt ist, also weder durch Annotationsdefinition noch durch Synonymtabelle vorgegeben ist.

Zusammenfassend werden Ähnlichkeiten und Synonyme zwischen Motiven in Annotationen einmalig manuell definiert. Dies hat durch einen kompetenten Entwickler zu geschehen, etwa von demjenigen, der die ursprünglichen Annotationsdefinitionen vorgenommen hat. Weiterhin sind ungültige Annotationen auszugrenzen. Zudem sind im Quelltext fehlende Annotationen der Musterdokumentation zu berücksichtigen. Im Quelltext vorhandene Annotationen, die nicht in einer speziellen Musterdokumentation vorliegen, deuten auf ein weiteres, für den Quelltext benötigtes Muster hin. Diese können zunächst ignoriert werden, bis der Vergleich mit einer Musterdokumentation stattfindet, in der diese Annotationen vorliegen.
Graduelle Unterschiede zwischen einer Annotation im Quelltext und der in der betrachteten Musterdokumentation können somit nur aus unterschiedlichen abstrakten Syntaxbäumen der jeweils annotierten Programmentität resultieren.

Für im Quelltext gegebene Annotationen sind ferner die Motive zu ermitteln, die für die aktuell betrachtete Musterdokumentation nicht als unabhängig zu den Motiven deklariert sind, die durch die Quelltext-Annotationen manifestiert sind. Für diese ermittelten Motive sollten die zugehörigen Annotationen der Musterdokumentation auch im Quelltext wiederzufinden sein. Sind sie das nicht, deutet das auf eine unvollständige Annotation des Quelltextes hin. Der Grad dieser Unvollständigkeit ist ein weiteres Maß für die Übereinstimmung eines annotierten Quelltextes mit einer Musterdokumentation und somit mit dem zugrunde liegenden Muster.

Folgende Kriterien können also für den Vergleich eines annotierten Quelltextes mit einer bestimmten Musterdokumentation bemüht werden:

1. Welche Annotationen stimmen zwischen Quelltext und Musterdokumentation überein?
2. Welche Annotationen liegen nicht im Quelltext, aber in der Musterdokumentation vor (nur abhängige Motive!)?

3. Welche Annotationen im Quelltext weichen von denen in der Musterdokumentation durch annotierte Programmentitäten mit unterschiedlichem abstraktem Syntaxbaum ab?

Für Kriterium eins (Positivkriterium) können im Vergleich Pluspunkte vergeben werden. Für die Kriterien zwei und drei (Negativkriterien) hingegen werden Minuspunkte vergeben. Eine exakt übereinstimmende Annotation sollte so stark gewichtet werden, dass mehrere Negativkriterien ein weniger starkes Gegengewicht darstellen. Diese Forderung basiert auf der starken Stellung eines exakt zutreffenden Motivs, das quasi eine gesicherte Aussage darstellt, wohingegen unzutreffende Kriterien lediglich eine (von der Maschine nicht sicher begründbare) Abweichung vom Soll oder eine Unvollständigkeit darstellen.

Kriterium zwei tritt dann ein, wenn unvollständig annotiert wurde. Das kann mehrere Gründe haben, die durchaus ihre Berechtigung haben. Dazu gehören beispielsweise Zeitdruck, Bequemlichkeit oder auch Komplexität des Prozesses der Annotation.
Kriterium drei hingegen deutet darauf hin, dass entweder eine falsche Quelltext-Annotation vorliegt oder eine Mustervariante angebracht wäre die nicht formal dokumentiert ist oder ein anderes Muster geeigneter ist. Daher ist Kriterium drei eine höhere negative Gewichtung zuzuweisen als Kriterium zwei.

Nachdem für jedes dokumentierte Muster die Übereinstimmung mit dem vorliegenden Quelltext anhand von Annotationen und Isomorphen durchgeführt wurde, kann dem Anwender das Ergebnis in Form einer Liste präsentiert werden. Diese Liste enthält zu jedem dokumentierten Muster den ermittelten Grad der Übereinstimmung, ggfs. aufgeteilt in Positiv- und Negativpunkte. Bei Bedarf können die festgestellten Gemeinsamkeiten und Unterschiede zwischen Musterdokumentation und vorliegendem Quelltext ebenfalls präsentiert werden. Das vom Anwender darauf hin gewählte Muster ist das für den folgenden Schritt der Musteranwendung relevante.

3.3.7.3 Fazit

Der Prozess der Musterselektion kann durch den Abgleich von Annotationen aus einem gegebenen Quelltext mit denen einer Musterdokumentation durchgeführt werden. Annotationen homogenisieren hierbei mögliche Unterschiede zwischen Musterdokumentation und gegebenem Quelltext. Die Definition von Aliasen für Annotationen sowie von isomorphen Programmentitäten erhöhen das Maß verarbeitbarer Varianz zwischen gegebenem und erwartetem (weil dokumentiertem) Quelltext. Der bloße Vergleich lediglich auf Basis von Quelltext kann dies nicht leisten. In diesem Sinne können Annotationen als generelle und Isomorphe als spezielle Abstraktion angesehen werden.

Da die Musterselektion auf dem Vorhandensein von Musterdokumentationen basiert, ist die Qualität letzterer entscheidend für die Güte des Selektionsprozesses. Nicht dokumentierte Muster können nicht selektiert werden. Ebenso können nicht dokumentierte Varianten von Mustern nicht selektiert werden. Ein vorliegender Quelltext kann umso besser mit Musterdokumentationen abgeglichen werden, je ähnlicher ein in der Musterdokumentation vorliegender exemplarischer Quelltext – unter Berücksichtigung der homogenisierenden Annotationen und Isomorphe – diesem ist. Vom Anwender vorgegebene Quelltext-Annotationen können nur identifiziert werden, wenn sie exakt so oder in Form von Aliasen definiert wurden. Andererseits können Annotationen für solche Programmentitäten automatisiert ermittelt werden, die syntaktisch mit bereits annotierten Entitäten in Bezug stehen, beispielsweise eine Variablendeklaration einerseits und der Zugriff auf die deklarierte Variable andererseits.

3.3.8 Phase F: Musteranwendung

Im Rahmen der Musteranwendung wird ein zuvor selektiertes Muster auf einen gegebenen Ausgangsquelltext angewandt.

Genau wie die zuvor beschriebene Musterselektion ist die Musteranwendung abhängig von im Ausgangsquelltext vorhandenen Annotationen. Die Selektion und Anwendung sind letztendlich von einander abhängig, denn die Selektion bestimmt ein anwendbares Muster dadurch, dass noch nicht vorhandene Musterteile sowie teils vorhandene Musterbestandteile ermittelt werden. Im Rahmen der Musteranwendung muss das Wissen über diese Teile verfügbar sein, um das Muster möglichst automatisiert anwenden zu können.

Mit Hilfe der Ankerpunkte aus der Musterselektion können die Transformationen durchgeführt werden, die letztendlich zur Anwendung des Musters führen. Eine Transformation wird durch Ausführen der ihr zugrunde liegenden Refactoring-Operationen realisiert. Eine Transformation ist nur dann durchführbar wenn eine Korrespondenz zwischen Ausgangsquelltext und Musterdokumentation mit Hilfe der Ankerpunkte hergestellt werden kann. Die Durchführbarkeit einer Transformation kann partiell eingeschränkt sein, wenn Vorbedingungen von manchen der ihr zugrunde liegenden Refactoring-Operationen verletzt sind. Eine Entscheidungsmöglichkeit für eine Transformation ist das Kriterium, ob für alle Refactoring-Operationen deren Vorbedingungen erfüllt sind. Ist mindestens eine nicht erfüllt, wird die Transformation abgelehnt. Hier ist es denkbar, den Anwender mit dem Grund der Ablehnung zu konfrontieren, etwa durch Nennung der fehlgeschlagenen Vorbedingung. Dann hat der Anwender die Möglichkeit, Änderungen im Quelltext vorzunehmen, um verletzte Vorbedingungen zu bereinigen. Komplexer ist die Möglichkeit der partiellen Ausführung einer Transformation durch Berücksichtigung lediglich der Refactoring-Operationen, deren Vorbedingung erfüllt ist. Diese Möglichkeit kann mit dem ersten Ansatz des Alles-oder-Nichts Prinzips von Transformationen kombiniert werden, indem der Anwender selbst bei verletzten Vorbedingungen eine Transformation ausführen lassen kann. Grundsätzlich ist der Ansatz, dass eine Transformation nicht generell zu einem syntaktisch oder semantisch korrekten Zielquelltext führt. Daher muss sowieso in Kauf genommen werden, dass der Anwender das Ergebnis einer Transformation kritisch untersucht und bei Bedarf Korrekturen von angebrachten, aber inkorrekten Transformationen durchführt.
Eine generell als durchführbar angesehene Transformation ist letztendlich nur notwendig, wenn der Ausgangsquelltext in dem eventuell zu transformierenden Teil nicht mit dem Zielquelltext der Musterdokumentation übereinstimmt.

Anhand der Ankerpunkte aus Tabelle 18 und Tabelle 19 können für die Musteranwendung durchzuführende Quelltexttransformationen ermittelt werden, indem die Zeilen des Ausgangsquelltextes der Musterdokumentation herangezogen werden. Die jeweiligen Zeilen ergeben sich aus den Ankerpunkten.
Für Motiv *a* (siehe Tabelle 18) sind demnach die Zeilen 060, 130, 180 und 190 im Ausgangsquelltext der Musterdokumentation (Abbildung 70) zu betrachten. Für diese Zeilen wurden in Phase A die Transformationen T5 (Zeile 060), T9 (Zeile 130) sowie T8 (Zeilen 180 und 190) durchgeführt. Eben diese Transformationen haben die genannten Zeilen berührt. Außerdem wurde für diese Zeilen in der Musterdokumentation Motiv *a* vorgesehen. Nun sind alle weiteren Zeilen zu ermitteln, die in der Musterdokumentation zu Motiv *a* gehören. Daraus können wiederum die zugehörigen Transformationen bestimmt werden, die dann auch für den gegebenen Quelltext anzuwenden sind. Diese Zeilen im Zielquelltext der Musterdokumentation (Abbildung 82) samt Transformationen lauten (Tabelle 20):

Zeile	Neu /Geändert	Transformation
060	Geändert	T5
115	Neu	T4
180	Geändert	T8
202 - 206	Neu	T3
340 - 348	Geändert	T7
370	Geändert	T2
500 - 520	Neu	T1

Tabelle 20: Neu hinzugekommene und geänderte Zeilen der Musterdokumentation samt Transformationen für Motiv a.

In Tabelle 20 ist in Spalte »Neu/Geändert« ebenfalls angegeben, ob die angegebene Zeile des Zielquelltextes im Ausgangsquelltext noch nicht (»Neu«) oder bereits in anderer Form vorhanden war (»Geändert«). Da ein Motiv in der Dokumentationsphase nur für transformierte Quelltextteile angebracht wird, kann eine Übereinstimmung von annotierten Zeilen zwischen Ausgangs- und Zielquelltext nicht vorkommen. Für die angegebenen Zeilen wurde keine aus dem Ausgangsquelltext gelöscht.

Gemäß Tabelle 19 ist für Motiv *b* die Zeile 340 im Ausgangsquelltext relevant. Die dazu gehörende Transformation ist T7. Alle in der Musterdokumentation zusätzlich mit Motiv *b* versehenen Zeilen samt Transformationen sind:

Zeile	Neu/Geändert	Transformation
064	Neu	T6
115	Neu	T4
202 - 206	Neu	T3

Tabelle 21: Neu hinzugekommene und geänderte Zeilen der Musterdokumentation samt Transformationen für Motiv b.

Durch Vergleich von Tabelle 20 mit Tabelle 21 ist ersichtlich, dass es Transformationen gibt, die den beiden Motiven *a* und *b* gemeinsam sind (nämlich T3 und T4). Jede Transformation ist allerdings nur einmal auszuführen. Die Transformationen T1, T2, T5, T7 und T8 sind nur aufgrund von Motiv *a*, die Transformation T6 nur aufgrund von Motiv *b* auszuführen.

Aus den Transformationen, die sich direkt aus annotierten Quelltextzeilen ergeben (siehe Tabelle 20 und Tabelle 21), leiten sich weitere Transformationen ab. Diese abgeleiteten Transformationen ergeben sich aus den Programmzeilen in der Musterdokumentation, die mit Motiv *a* (bzw. mit Motiv *b* im zweiten Fall) versehen sind. Sind diese Programmzeilen jeweils mit weiteren Motiven versehen, so sind diese weiteren Motive ebenfalls zu berücksichtigen, indem die dazu gehörenden Transformationen ebenfalls für die Ausführung verzeichnet werden. Diese Notwendigkeit ergibt sich aus der Tatsache, dass eine Programmentität von der Existenz einer anderen abhängig sein kann und dass beide Programmentitäten prinzipiell mit einem anderen Motiv versehen sein können.
Im Zielquelltext aus Abbildung 82 ergeben sich so aus Motiv *a* die Motive *b*, *c*, *d*, *e* und *f* (vgl. etwa die Zeilen 202 und 346 des Zielquelltextes). Weiterhin kann festgestellt werden, dass die Signaturen der Klassen *PatDetails* und *IObserver* mit Motiv *a* und in diesen Klassen enthaltene Programmentitäten mit den Motiven *e* und *f* versehen sind. Daraus kann abgelesen werden, dass die Motive *e* und *f* quasi in Motiv *a* enthalten sind. Letztgenannte Motive könnten automatisiert aus Motiv *a* gefolgert werden, weil sie direkt abhängig davon sind (da sie sich auf Programmentitäten beziehen, die in neu eingeführten Klassen liegen, welche mit Motiv *a* annotiert sind).
Im der Musterdokumentation zum *Observer* gibt es nur eine Transformation, die noch nicht zu Motiv *a* oder *b* ermittelt wurde, nämlich Transformation T9. Diese ist aufgrund der Korrelation von Motiv *d* zu *a* ebenfalls für die Ausführung vorzusehen. Alle anderen Motive sind durch die Transformationen T1 bis T8 bereits abgedeckt. Dies deckt sich auch mit den in Tabelle 11 genannten Transformationen.

Die ermittelten Transformationen können in flexibler Reihenfolge ausgeführt werden, sollten sich aber an der Reihenfolge der Musterdokumentation orientieren. Letzteres ist dann relevant, wenn Programmentitäten zu einer Transformation innerhalb einer Programmentität einer anderen Transformation liegen (Beispiel: Methodensignatur innerhalb einer Schnittstelle). Wichtig ist allerdings, dass eine Transformation bestenfalls nur ausgeführt wird, wenn sie nicht von einer anderen abhängig ist und deren Vorbedingungen erfüllt sind. Daher sind zuerst die Transformationen mit erfüllten Vorbedingungen und ohne Abhängigkeiten zu realisieren. Danach kann für Transformationen mit bisher nicht erfüllten Vorbedingungen erneut deren Anwendbarkeit geprüft werden, so dass bei Vorhandensein aller relevanten Vorbedingungen jede Transformation in der richtigen Reihenfolge abhängig zu den anderen ausgeführt wird. Das Führen einer Abhängigkeitstabelle für Transformationen ist nur notwendig, wenn zwei Transformationen entweder dieselbe Programmanweisung betreffen oder wenn eine Transformation eine Programmentität generiert, die die andere (abhängige) Transformation modifiziert (siehe oben).

Die Durchführung einer als relevant ermittelten Transformation aus der Musterdokumentation zur sukzessiven Anwendung des zugrunde liegenden Musters kann ohne Anwendereingriff nur erfolgen, wenn erstens die Vorbedingungen zur jeweiligen Transformation erfüllt sind und zweitens alle abhängigen Transformationen bereits ausgeführt wurden. Liefert eine Transformation trotz erfüllter Vorbedingungen der zugehörigen Refactoring-Operationen ein unerwartetes (suboptimales oder falsches) Ergebnis, so gibt das Anlass, die Musterdokumentation zu überarbeiten. Erst danach sollte die Musteranwendung erneut durchgeführt werden. Zuvor sollte allerdings die Musterselektion wiederum durchlaufen werden, um sicherzustellen, dass das bisher als am besten anwendbar angesehene Muster nach der Korrektur ebenfalls der Top-Kandidat ist.
Die wie beschrieben ermittelten Transformationen sind in Tabelle 22 samt den im Ausgangsquelltext aus Abbildung 131 (Aktionen »Geändert«, »Gelöscht«) und den im Zielquelltext aus Abbildung 135 (Aktionen »Geändert«, »Neu«) betroffenen Zeilen genannt:

Transformation	Zeilen	Aktion
T1	500 – 520	Neu
T2	110	Geändert
T3	291 – 296	Neu
T4	220 222	Gelöscht Neu
T5	060	Geändert
T6	062 – 064	Neu
T7	420 422 – 428	Gelöscht Neu
T8	270 280	Geändert Gelöscht
T9	220	Gelöscht

Tabelle 22: Durchgeführte Transformationen bei der Musteranwendung.

Mit Hilfe der Anwenderentscheidung über Methode *dummyMethod* sowie aufgrund der Auswertung der Ähnlichkeit des gegebenen Ausgangsquelltextes mit der selektierten Musterdokumentation kann die Musteranwendung erfolgen. Nach Ausführung der ermittelten Transformationen auf dem Quelltext aus Abbildung 131 lautet der sich ergebende Zielquelltext gemäß Abbildung 135:

```
010    public class PVAnwendung {
020      public static void main(String[] args) {
030        // Konstruiere Observierer
040        PatDetails bo2 = new PatDetails();
050        // Konstruiere Observierten und registriere Observierer
```

```
060 acd    PatSelector bo1 = new PatSelector();
062        // Registriere Observierer
064 bcd    bo1.register(bo2);
070        // Starte Logik des Observierten
080        bo1.start();
090      }
100    }

110 a  public class PatDetails implements IObserver {
120      public void refresh(Object state) {
130        // Aktualisiere Zustand und Darstellung
140        // ...
150        System.out.println("Reagiere auf Zustandsänderung von PatSelector");
160      }
170    }

200    public class PatSelector {

222 ab   private List observers = new Vector();

230      // Zustand dieses Objekts (normalerweise erst zusammengestellt, wenn
240      // benötigt. Er besteht etwa aus einer Handvoll Variablenbelegungen).
250      private Object state;

260      // Konstruiere Objekt und registriere einen Observierenden
270 acd  public PatSelector() {
290      }

291      /**@@motiv a(3): Behandle Beobachterklassen einheitlich*/
291      /**@@motiv b(1): Registrierungsmöglichkeit vieler Beobachter*/
291      /**@@motiv c(1): Registrierung zu beliebigem Zeitpunkt*/
291      /**@@motiv d(4): Einheitlicher Registrierungsmechanismus*/
292 abcd public void register(IObserver observer) {
294        observers.add(observer);
296      }

300      public void start() {
310        //... tue irgendwas hier ...

320        // Jetzt tue etwas, was den Zustand dieses Objekts signifikant ändert
330        doWorkChangingState();
340        //... tue auch noch irgendwas anderes hier ...
350      }

360      public void doWorkChangingState() {
370        System.out.println("Ich tue etwas, was meinen Zustand verändert");
380        System.out.println("Deshalb informiere ich PatDetails darüber");
390        dummyMethod();
400      }
410      private void dummyMethod() {
422 ae     Iterator it = observers.iterator();while (it.hasNext()) {
424 ae       IObserver observer = (IObserver)it.next();
426 aef      observer.refresh(state);
```

```
428        }
430      }
440    }
500 a  public interface IObserver {
510 ef   void refresh(Object state);
520    }
```

Abbildung 135: Zielquelltext nach der Musteranwendung.

In Abbildung 135 sind die transformierten Zeilen durch Angabe der zugehörigen Motive neben der jeweiligen Zeilennummer kenntlich gemacht. Durch die Abweichung der Zeilennummern vom Ausgangsquelltext wird deutlich, dass eine Invarianz bezüglich der Position von Programmentitäten im Ausgangsquelltext relativ zur Musterdokumentation besteht.

3.3.8.1 Fazit

Auf Basis des Selektionsprozesses wurde das dort gewählte Muster angewandt. Die Ergebnisse aus der Musterselektion konnten für die Musteranwendung verwendet werden. Das war möglich, weil die Grundlage durch die formale Musterdokumentation gegeben ist. Die Musteranwendung konnte mit Hilfe von Ankerpunkten aus der Musterselektion durchgeführt werden. Jeder Ankerpunkt hat dabei einen Teil des Quelltextes sowie eine dazu gehörige Transformation bezeichnet. Die Musteranwendung konnte durch sequentielles Ausführen der Transformationen jedes Ankerpunktes abgebildet werden. Der Anspruch war, eine möglichst weitgehende Automatisierung der Quelltexttransformation vorzunehmen. Es wurde notgedrungen in Kauf genommen, dass einzelne Transformationen Teile im Ergebnisquelltext erzeugen, die entweder nicht kompilierbar oder semantisch nicht einwandfrei sind.
Die Transformation von Programmentitäten innerhalb von Methoden hat sich als besonders kritisch erwiesen, da die Ausgangssituation beliebig geartet sein kann und auch aus überflüssigen, suboptimalen oder sogar fachlich falschen Anweisungen bestehen kann. Die Komplexität derartiger Logik wird In Anhang B für das *Composite*-Muster in der Methode *create* des Ausgangsquelltextes deutlich. Ein Ausweg bietet sich an, indem zusammengehörige Sequenzen von Anweisungen entweder zu einem Programmblock zusammen gefasst oder in eine neue Methode ausgelagert werden (vgl. Refactoring-Operation *Extract Method* [Fowler 2001]). Diese neu gebildeten Einheiten (Blöcke oder Methoden) können dann wiederum mit einem spezifischen Motiv versehen werden und in einer Transformation im Rahmen der Musterdokumentation feingranular berücksichtigt werden können. In der Transformationslogik muss dann zusätzlich eine Analysefunktion für spezielle Anweisungssequenzen vorgesehen werden, die die signifikanten Informationen extrahiert. Im Falle der Methode *create* des *Composite*-Musters ist festzustellen, dass es eine Fallunterscheidung gibt und die einzelnen Zweige der *if*-Bedingung die unterschiedlichen Komponenten des Musters *Composite* mit der gleichnamigen Rolle *Composite* (vgl. [Gamma+ 1995]) betreffen.

3.4 Integration von TrAQ in Entwicklungswerkzeuge

Das in dieser Arbeit beschriebene Verfahren TrAQ kann mit Hilfe von Werkzeugen unterstützt werden. Grundlage ist das Vorhandensein einer heutzutage üblichen integrierten Entwicklungsumgebung, vgl. auch das Konzept der Workbench ([Kazman+ 2002]) und das des kontextuellen Hilfesystems ([Motelet 2004]). Eine IDE erlaubt das Editieren und Kompilieren von Quelltext. Populäre IDE's für Java, wie Eclipse oder NetBeans, erlauben das Einbinden von Erweiterungen in Form so genannter Plugins (vgl. [Mayr-Kern+ 2002], wo Eclipse verwendet wird). Mit Hilfe dieser Plugins können individuelle Logiken, statische Menüs und Kontextmenüs umgesetzt werden. Weiterhin kann mit Plugins speziell auf Ereignisse reagiert werden, die von der IDE erzeugt werden. Beispielsweise kann so das Einfügen oder Ändern einer Programmzeile überwacht und behandelt werden.

Eine Werkzeugunterstützung bei der Anwendung des Verfahrens ist insbesondere deswegen realisierbar, weil TrAQ auf formalisierten und somit von einer Maschine verarbeitbaren Konstrukten basiert.

Das Erkennen von Quelltextteilen, die zu annotieren sind, kann durch Werkzeuge unterstützt werden. Beispielsweise können Vergleiche zwischen einem vorliegenden Quelltext und vorhandenen formalen Musterdokumentationen durchgeführt werden, um für eine Annotation potentielle Quelltextstellen zu erkennen. Hier können auch Isomorphe berücksichtigt werden. Geltungsbereiche von Annotationen können abgeprüft werden und dazu führen, dass Annotationen nur für zulässige Programmstellen vorgeschlagen werden. Durch statische Programmanalyse können für Erzeugungsmuster relevante Konstruktoraufrufe und sprechende Namen identifiziert werden. Weiterhin können Hilfetexte und Tutorien den Entwickler für die Annotation sensibilisieren. So kann etwa ein kontextuelles Hilfesystem nach Auswahl des Themas »Benachrichtigung einer Klasse durch eine andere, wenn sich deren Zustand ändert« auf Annotationen abzielen, die für entsprechende Muster (wie *Observer*, *Publish-Subscribe* oder *MVC*) relevant sind.
Entsprechende Werkzeuge sind besonders dienlich, wenn sie eine graphische Oberfläche bereitstellen. Die Einführung einer graphischen Oberfläche zur Ermöglichung der Annotation entkoppelt den Vorgang der Annotation an sich von der Definition der Annotationsgrammatik (vgl. auch [Baroni+ 2003b]).

Im Folgenden wird beschrieben, wie der vorgestellte Ansatz in den Entwicklungsprozess eingebunden werden kann. Prinzipiell kann Quelltext zu jedem beliebigen Zeitpunkt annotiert werden. Ebenso können zu jeder Zeit formal dokumentierte Muster selektiert und angewandt werden. Eine Berücksichtigung des Software-Lebenszyklus ist daher an dieser Stelle nicht nötig.

Vielmehr wird vorgeschlagen, spezialisierte Werkzeuge in der täglich genutzten Entwicklungsumgebung einzubinden. Dieses Konzept folgt dem der Workbench, wie es heute weit verbreitet ist (vgl. die populären Java-Entwicklungsumgebungen wie Eclipse, NetBeans oder JBuilder) und auch in [Kazman+ 2002] beschrieben wird. Weiterhin wird vorgeschlagen, das Konzept der Workbench mit dem des kontextuellen Hilfesystems (siehe [Motelet 2004]) zu kombinieren. Das bedeutet, dass dem Entwickler kontextabhängige Informationen und Eingabehilfen zur Unterstützung der Annotation von Quelltext sowie der Selektion und Anwendung von Mustern angeboten werden.

Eine auf den Ansatz abgestimmte Arbeitsoberfläche kann folgende Elemente beinhalten (Abbildung 136):

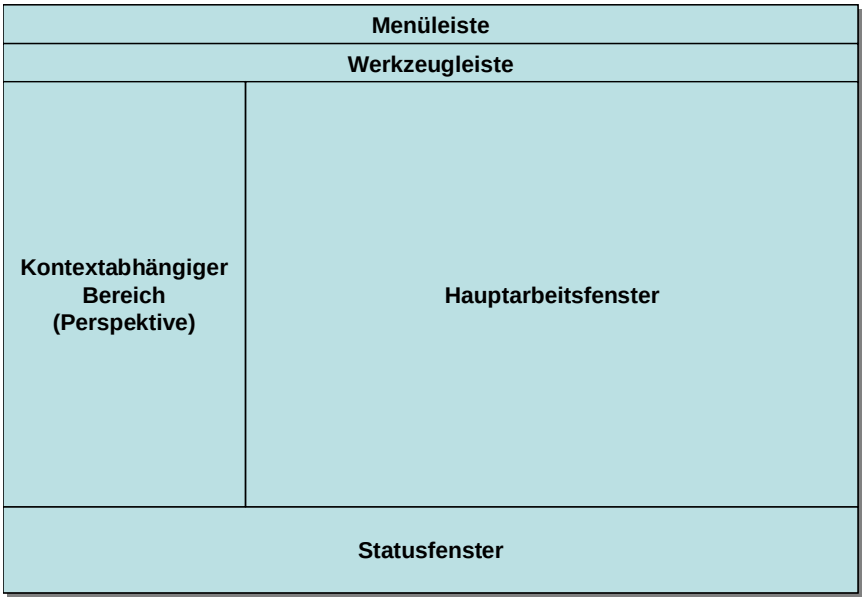

Abbildung 136: Elemente einer Workbench.

Die im Bild dargestellten Elemente sind:

- **Menüleiste**: Überwiegend kontextunabhängige Funktionen, wie Speichern, Kompilieren oder Operationen auf der Zwischenablage
- **Werkzeugleiste**: Individuell gestaltbare Anordnung von häufig benötigten Funktionen, die über Drucktasten aufgerufen werden können
- **Kontextabhängiger Bereich**: Spezielle Sicht auf Informationen, wird typischerweise vom Anwender im Bedarfsfall aus einer Menge von verfügbaren Perspektiven gewählt (vgl. die Java-Entwicklungsumgebung Eclipse)
- **Hauptarbeitsfenster**: Typischerweise wird hier der Quelltext angezeigt, den der Entwickler bearbeiten kann
- **Statusfenster**: Meldungen nach durchgeführter Operation, gegliedert in Informations-, Warn- und Fehlermeldungen

3.4.1 Vorschlag für die Gestaltung der Workbench

Eine konkrete Ausprägung der allgemein beschriebenen Elemente einer Workbench für den Ansatz dieser Arbeit kann wie in Abbildung 137 gestaltet sein:

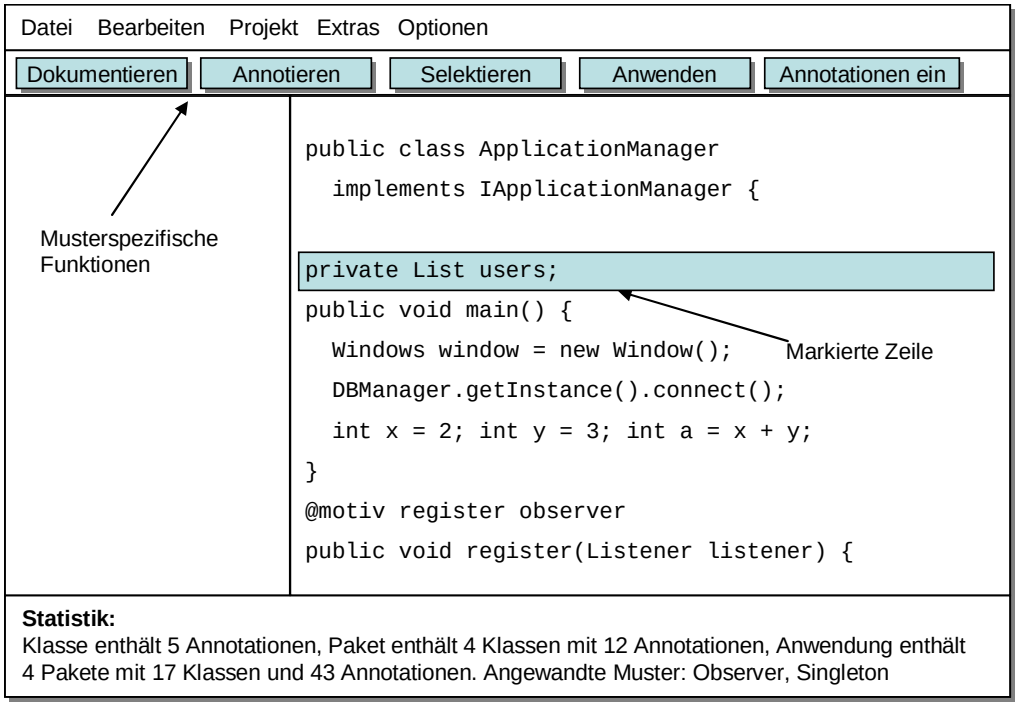

Abbildung 137: Konkrete Ausprägung einer Workbench für die Arbeit mit Mustern.

Die Aufteilung der Abbildung 137 entspricht der vorigen Abbildung 136. In der Menüleiste sind allgemeine Funktionen angedeutet. Sie haben hier keine spezielle Bedeutung. Die Werkzeugleiste ist mit fünf wichtigen Funktionen bestückt. Dies sind:

a) Dokumentieren: Führt den Chefentwickler in einen vom normalen Entwicklungsbetrieb separaten Prozess der formalen Musterdokumentation
b) Annotieren: Funktion für den Entwickler, um seinen Quelltext im Hauptarbeitsfenster mit Annotationen zu versehen
c) Selektieren: Erlaubt es zum vorliegenden Quelltext ein anzuwendendes Muster zu selektieren
d) Anwenden: Nach Durchführung des vorigen Schritts kann das Muster hiermit angewandt werden
e) Annotationen ein: Umschaltbutton, der im Quelltext vorhandene Annotationen ein- oder ausblendet. Der Text des Buttons ändert sich nach Betätigen in »Annotationen aus«

Der kontextabhängige Bereich wird erst in der weiteren Beschreibung spezifischer Funktionen konkret dargestellt.
Im Hauptarbeitsfenster ist ein Quelltext angedeutet, auf dem der Entwickler arbeitet. Zur Festlegung eines Kontextes ist es möglich, Zeilen zu markieren, wie im Bild ebenfalls angedeutet. Eine Zeile kann etwa vom Entwickler markiert werden um sie im Anschluss zu annotieren.

Der Statistikbereich enthält nach Durchführung einer Funktion der Werkzeugleiste Informationen über die durchgeführte Funktion. In Abbildung 137 ist ein Analyseergebnis angedeutet, dessen Schwerpunkt auf dem Quantifizieren von Annotationen liegt. Dieses kann im Hintergrund, etwa bei Programmstart und ohne dass ein Anwendereingriff erforderlich ist, zusammengestellt und ausgegeben werden. Denkbar ist eine Verlinkung von einzelnen Textelementen der Analyseausgabe, so dass der Entwickler in ein Detailbild mit weiteren Informationen zum gewählten Element gelangt.

Die Funktionen der Werkzeugleiste werden nun im Einzelnen beschrieben. Für die Funktion »Dokumentieren« soll hier nicht diskutiert werden, wie deren werkzeuggestützte Umsetzung aussehen kann. Im Kapitel 3 wurde stattdessen ausführlich beschrieben, wie eine formale Musterdokumentation erlangt werden kann. Ein Werkzeug kann hier unterstützen, indem es

durchgeführte Transformationen vom Ausgangs- in den Zielquelltext möglichst umfänglich analysiert und manuelle Routinearbeiten reduziert. Die Ausgestaltung einer konkreten Werkzeugunterstützung ist zukünftigen Arbeiten vorbehalten.

3.4.2 Die Funktion »Annotieren« in der Workbench

Die Funktion »Annotieren« bietet dem Entwickler mehrere Alternativen eine Quelltextannotation vorzunehmen, die in Abbildung 138 angedeutet sind:

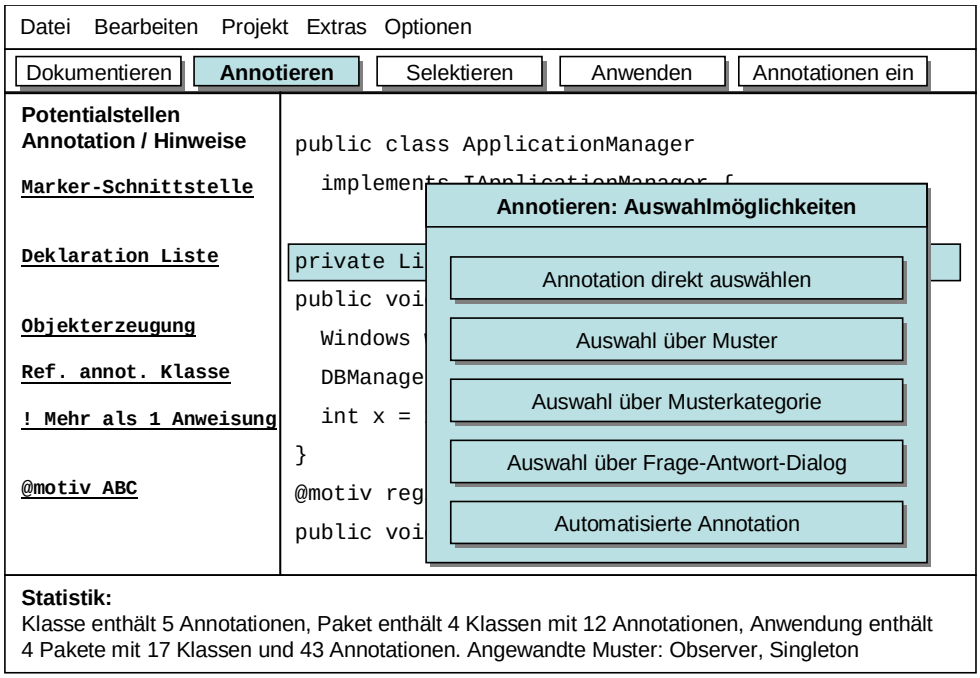

Abbildung 138: Alternativen zum Annotieren eines Quelltextes.

In Abbildung 138 sind im kontextabhängigen Bereich (links neben dem Quelltext) Analyseergebnisse des Quelltextes hinsichtlich Annotierbarkeit dargestellt. Diese Ergebnisse können entweder im Hintergrund oder bei Bedarf, etwa bei Betätigen der Option »Annotation direkt auswählen« innerhalb der Funktion »Annotieren« zusammengestellt werden.
Die Analyseergebnisse sind zur Kennzeichnung von potentiell für die Annotation relevanten Programmentitäten gedacht. So ist in der Abbildung als erstes der Hinweis auf eine Marker-Schnittstelle namens *IApplicationManager* angedeutet. Solche dedizierten Schnittstellen sind meist leer und haben quasi nur die Funktion einer Markierung. In Java beispielsweise ist die Schnittstelle *java.io.Serializable* so ein Marker, der dafür sorgt, dass eine damit gekennzeichnete Klasse automatisch serialisierbar ist, ohne weiteres Zutun des Entwicklers. Derartige Marker-Schnittstellen sind manuell in einer Tabelle festzuhalten, damit das System Kenntnis davon hat.
Eine weitere in der Abbildung genannte Potentialstelle ist die Deklaration einer Liste als spezielle Datenkollektion, etwa relevant für das Muster Composite. Derartige Typen (in Java: *java.util.List*) sind ebenfalls manuell und einmalig in einer Tabelle zu erfassen.
Eine zusätzlich genannte Potentialstelle ist die Objekterzeugung, die wichtig für Erzeugungsmuster ist.

Im kontextabhängigen Bereich sind ferner Hinweise und Warnungen dargestellt. Eine Warnung bezieht sich auf die Tatsache, dass mehr als eine Anweisung in einer Zeile implementiert wurde. Das ist bei der Annotation kritisch, da die Annotation sich nur auf die nächstfolgende Anweisung bezieht und es schwer zu erfassen ist, dass die darauf folgende Anweisung in derselben Zeile nicht annotiert ist.

Ein Hinweis deutet darauf hin, dass an einer Stelle eine bereits annotierte Klasse referenziert wird. Das kann hilfreich sein, wenn Annotationen in der referenzierenden Klasse hinzugefügt werden sollen.
Die in Abbildung 138 genannten Alternativen sind als Arbeitserleichterung zu verstehen, um das Hinzufügen von Annotationen durch direktes Eintippen in den Quelltext vorzunehmen. Allerdings kann selbst beim Eintippen eine Hilfe angeboten werden, nämlich durch Vervollständigung. Nahezu alle populären Entwicklungsumgebungen bieten dieses als »Code Completion« bezeichnete Prinzip für Konstrukte der jeweils unterstützten Programmiersprache bereits an.

Die in Abbildung 138 als erstes genannte Möglichkeit, eine Annotation direkt auszuwählen, führt den Anwender zu einer Liste aller Annotationen. Hier ist es entscheidend, ob bereits eine Zeile im Quelltext markiert wurde oder nicht. Wurde keine Zeile markiert, werden alle bereits im Repository definierten Annotationen angeboten. Nach Auswahl einer Annotation werden die Stellen im Quelltext hervorgehoben, an denen die Annotation aufgrund ihres Geltungsbereichs angebracht werden kann. Ist andererseits vom Entwickler bereits eine Zeile im Quelltext markiert, so werden nur die dafür zulässigen Annotationen zur Auswahl angeboten.

Die zweite Möglichkeit, eine Annotation über ein Muster auszuwählen, führt zunächst zur Auswahl eines Musters aus einer Liste. Es ist denkbar, einen mehrstufigen Dialog zu implementieren, der vorliegende Zuordnungen von Mustern zu Gruppen (etwa Erzeugungsmuster) auswertet. Nach Auswahl eines Musters durch den Anwender zeigt der folgende Dialog nur die Annotationen an, die für das jeweils gewählte Muster in dessen formaler Dokumentation vorliegen. Anstatt hier auf eine eventuell markierte Quelltextzeile einzuschränken werden vom System alle Zeilen im Quelltext markiert, für die eine Annotation zum Muster möglich ist. Es ist auch denkbar, den Anwender zuerst eine Annotation wählen zu lassen, um dann zu sehen für welche Programmzeilen diese anwendbar ist. Oder es werden tatsächlich nur die Annotationen angezeigt, die zu einer vom Anwender zuvor markierten Programmzeile passen.
Nachdem ein Quelltext annotiert wurde, kann ein Muster dafür selektiert werden. Der Prozess der Musterselektion wurde in Abschnitt 3.3.7 beschrieben. In der folgenden Abbildung 139 ist dargestellt wie ein Analyseergebnis präsentiert werden kann. Ausgehend davon kann der Benutzer ein anzuwendendes Muster auswählen.

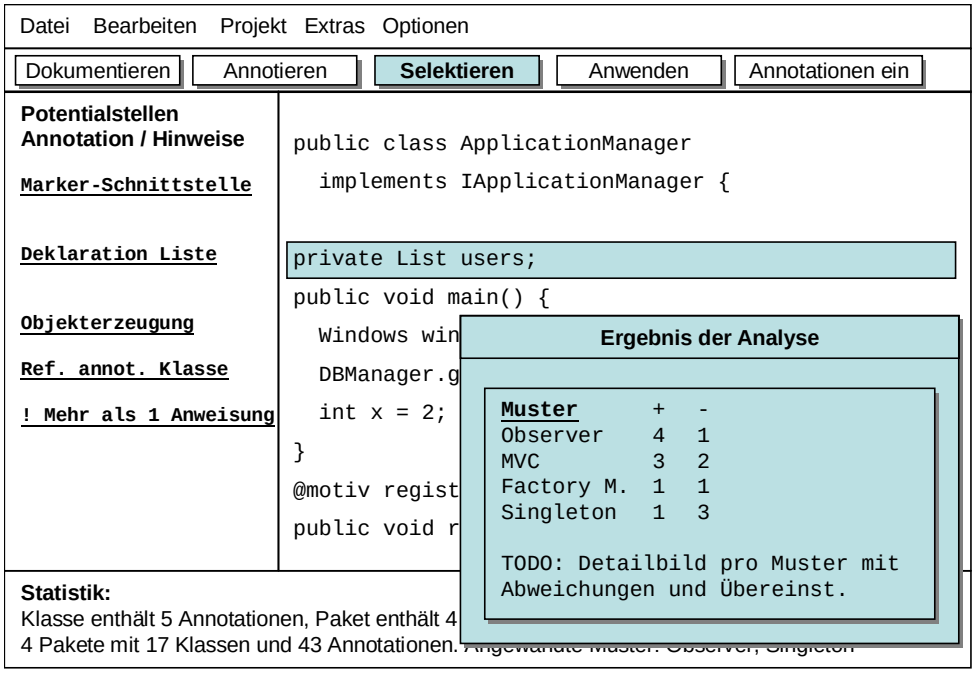

Abbildung 139: Selektieren eines anzuwendenden Musters.

In der Abbildung ist eine Rangliste der auf Anwendbarkeit untersuchten Muster dargestellt. Es sollten nur die Muster in der Liste präsentiert werden, die eine als ausreichend angesehene Anwendbarkeit besitzen. Die Anwendbarkeit ist gekennzeichnet durch die Anzahl der übereinstimmenden Kriterien und die Anzahl der nicht übereinstimmenden Kriterien. Als Kriterium wird eine Annotation im gegebenen Quelltext aber auch in der Musterdokumentation angesehen. Ist eine Annotation des Quelltextes in der Musterdokumentation vorhanden, zählt dies als Übereinstimmung. Graduelle Unterscheidungen können bei Abweichungen des Geltungsbereiches der Annotation sowie bei Ähnlichkeitsbeziehung zwischen Annotation im Quelltext und der im der Musterdokumentation vorgenommen werden. Ist eine Annotation im Quelltext, aber nicht in der Musterdokumentation vorhanden, oder umgekehrt, so zählt dies negativ. Die Entscheidung, ab wann ein Muster würdig ist auf der Rangliste zu erscheinen, sollte durch eine Option steuerbar sein (Verhältnis der positiven zu den negativen Treffern sowie Mindestanzahl an positiven Treffern). Zu beachten ist, dass die in der Trefferliste dargestellten Muster eine konkret dokumentierte Musterausprägung repräsentieren. Das bedeutet, ein dargestelltes Muster ist entweder das einzige Muster dieses Namens oder eine von mehreren dokumentierten Mustervarianten mit unterschiedlichen Namen (und somit de facto ein eigenständiger Eintrag in der Trefferliste).

Zur Unterstützung des Anwenders kann jedes Muster mit einer Verknüpfung auf zusätzliche Informationen versehen werden. Diese Informationen beinhalten informelle Musterdokumentationen wie in [Gamma+ 1995], UML-Diagramme, Beispielquelltexte sowie die Möglichkeit, die zum Muster gehörige formale Musterdokumentation samt Annotationen, Ausgangs- und Zielquelltext sowie Transformationen und Motivbeschreibungen anzusehen.

Hat der Anwender sich aufgrund der präsentierten Informationen für ein Muster entschieden, kann dessen formale Musterdokumentation für die Anwendung auf dem Quelltext herangezogen werden. In der Entwicklungsumgebung ist nach Selektion des Musters lediglich der Button »Anwenden« zu betätigen (Abbildung 140):

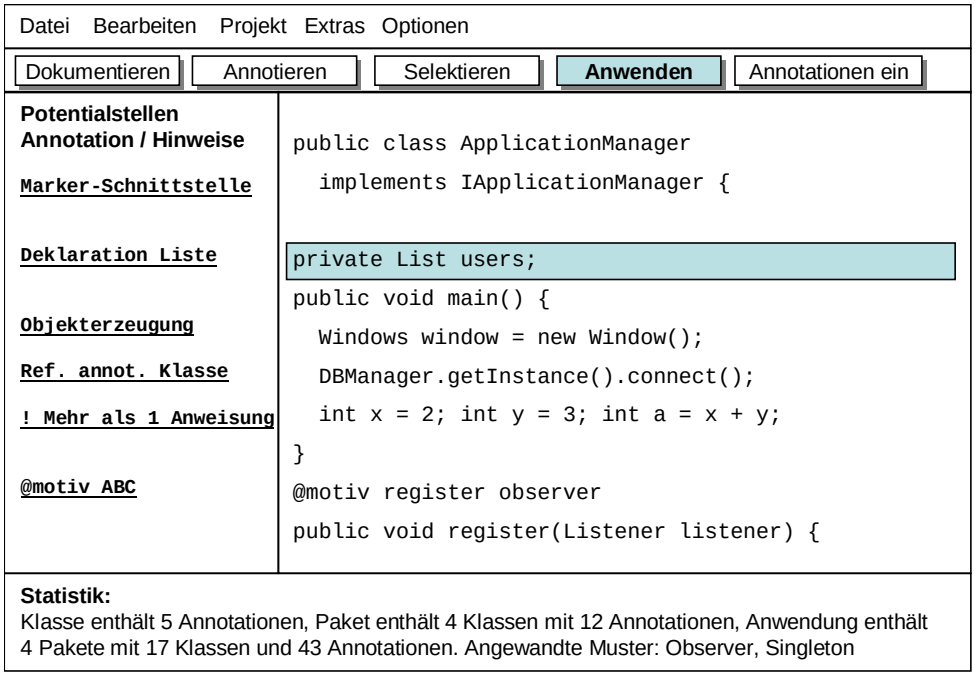

Abbildung 140: Anwenden eines selektierten Musters.

Die in der Musterselektion gewonnenen Informationen zu Übereinstimmungen und Abweichungen zwischen Quelltext und Musterdokumentation werden ebenfalls herangezogen. Im Anwendungsschritt ist das Anbieten eines Konfigurators denkbar, der etwa das Umbenennen von neu eingeführten Klassen, Methoden und Variablen erlaubt.

Die Workbench präsentiert nach Betätigen des Buttons »Anwenden« (Abbildung 140) den resultierenden Quelltext. Weiterhin werden Warnungen und Hinweise dargestellt, die während des Transformationsprozesses gewonnen wurden. Warnungen beinhalten insbesondere Transformationen, die aufgrund von nicht zutreffenden Vorbedingungen nicht ausgeführt wurden. Weiterhin kann dem Anwender mitgeteilt werden, wenn ein nicht kompilierbarer Quelltext entstanden ist. Hinweise können statistische Angaben enthalten, wie etwa die Anzahl der ausgeführten Quelltext-Transformationen.

3.5 Zusammenfassung

In diesem Kapitel 3 wurde TrAQ im Detail vorgestellt. Anfangs wurde eine Übersicht über das Verfahren samt weitergehenden Begriffsdefinitionen gegeben. Als elementarer Bestandteil des Verfahrens sind Motive und Annotationen herausgearbeitet worden. Auf Basis dieser Bestandteile wurden die Prozesse Musterdokumentation, Musterselektion und Musteranwendung beschrieben und am Beispiel des Musters *Observer* demonstriert. Die Eignung von TrAQ für andere Muster als *Observer* wurde durch den Einsatz der zunächst kontextfreien Konstrukte Motiv und Annotation sowie durch das Definieren von Transformationen bzw. Refactoring-Operationen mit Hilfe der generisch einsetzbaren Sprache Java gewährleistet. Im Anhang wurde ferner für die beiden Muster *Singleton* und *Composite* angegeben, wie deren Ausgangs- und Zielquelltext in der Musterdokumentation aussieht, welche Motive die Muster bedingen und welche Zeilen in den genannten Quelltexten von den Motiven betroffen sind.

Als Voraussetzung für das Verfahren wurde das hauptsächliche Arbeiten mit Quelltext benannt. Im Rahmen einer modellgetriebenen Entwicklung, die durch den kontinuierlichen Einsatz von Code-Generatoren bestimmt ist, kann das Verfahren nicht verwendet werden.

Es wurde gezeigt, dass TrAQ sich eignet um Muster formal in einer Weise zu definieren, die eine maschinengestützte Selektion und Anwendung der dokumentierten Muster erlaubt. Zur Unterstützung des Menschen bei der Musterdokumentation wurde vorgeschlagen, anhand einer beispielhaften Transformation eines Ausgangsquelltextes durch Anwendung des zu dokumentierenden Musters einen Zielquelltext zu erzeugen. Die dabei abfallenden Informationen zu Transformationen, Motiven und Annotationen können zur Musterdokumentation genutzt werden.

Produkt einer formalen Musterdokumentation ist ein Satz von prägnanten Aussagen über die Absichten des dokumentierten Musters in Form von Motiven. Weiterhin können durch die Verfeinerungsphasen Varianten des Musters definiert werden, die somit explizit benannt sind. Die Verfeinerungsphasen bieten durch beliebig häufige Iteration die Möglichkeit, eine Musterdokumentation mit zunehmend investiertem Zeitaufwand umfänglicher zu gestalten. Für komplexe Muster können die Phasen E und F zusätzlich als Verfeinerungsphasen durchlaufen werden, um die Qualität der Musterdokumentation zu verbessern. Erst nach Abschluss dieser zusätzlichen Iteration wird die Musterdokumentation dann für die produktive Nutzung freigegeben.

Weiterhin wurde gezeigt, wie auf Basis der formalen Musterdokumentation eine Musterselektion für einen bisher unbekannten Quelltext durchgeführt werden kann. Das Ergebnis der Selektion ist die Aussage über den Grad der Eignung dokumentierter Muster für diesen Quelltext. Das neue Verfahren TrAQ entlastet den Entwickler bei der Musterselektion, somit von der bisher notwendigen Arbeit, umfänglich mögliche Muster in Form informeller Musterbeschreibungen wie in [Gamma+ 1995] zu studieren.

Hinsichtlich der Musteranwendung wurde gezeigt, wie die Informationen aus dem Vorgängerprozess der Musterselektion verwendet werden können, um die Anwendung des Musters durchzuführen. Die Musteranwendung auszuführen bedeutet danach die Ermittlung anwendbarer Transformationen, die in der Musterdokumentation definiert wurden, sowie deren anschließende Ausführung.
Es wurde festgestellt, dass automatisiert ausgeführte Quelltexttransformationen nicht mit absoluter Sicherheit so ausgeführt werden können, dass ein kompilierbarer oder gar semantisch korrekter Quelltext entsteht. Vielmehr ist der Entwickler angehalten, das Ergebnis der Musteranwendung manuell zu kontrollieren und ggfs. zu korrigieren.

Das Problem des Vergleiches eines Quelltextes mit einer wie auch immer gearteten Beschreibung eines Musters ist insofern vereinfacht worden, als dass mit TrAQ nur noch Annotationen verglichen werden müssen. Das daraus entstehende Problem der Annotation von Quelltext wurde einerseits entschärft durch die Angabe von Richtlinien, wie Quelltext zu annotieren ist und weiterhin durch das Aufzeigen von Möglichkeiten, Annotationen im Quelltext automatisiert anzubringen. Drittens wurde die Möglichkeit erwähnt, bestehende formale Musterdokumentationen als Hilfestellung beim Annotieren heranzuziehen. Die in [Ó Cinnéide+ 1999a] vorgestellten Hilfsfunktionen und Prädikate zur Identifikation notwendiger Programmbestandteile zur Musteranwendung sind im Prinzip durch die vorgeschriebene Methodik der Musterdokumentation obsolet geworden.

Weiterhin wurde skizziert, wie TrAQ in eine Entwicklungsumgebung integriert werden kann samt einer möglichen Form der Darstellung.

3.6 Berücksichtigung bestehender Ansätze

TrAQ besitzt eine Reihe von Gemeinsamkeiten mit den in Kapitel 2 vorgestellten Ansätzen. Die Berührungspunkte werden in diesem Abschnitt benannt, gefolgt von den umgangenen, als negativ angesehenen Eigenschaften der betrachteten Ansätze.

In Übereinstimmung mit [Taibi+ 2003] ist der Ansatz dieser Arbeit auf Entwurfsmuster und verwandte Musterformen spezialisiert.

Die Grundidee, sowohl Struktur als auch Verhalten eines Programms zu beschreiben, entspricht dem in [Baroni+ 2003b], [Taibi+ 2003] und [Eden+ 2004] dargestellten. [Eden+ 2004] führt sogenannte *Lattices* ein, die strukturelle, relationale und verhaltensorientierte Aspekte darstellen. TrAQ beschreibt die Struktur einer Musterlösung durch den während der Dokumentationsphase transformierten Quelltext. Nicht offensichtliche Abhängigkeiten zwischen Programmentitäten (wie Deklaration und Zuweisung einer Variablen) können mit Hilfe von Annotationen ausgedrückt werden. Annotationen helfen auch, die Bedeutung und somit das Verhalten von Musterteilen auszudrücken. [Baroni+ 2003b] verwendet Entwurfsmotive (*Design motifs*), die vorliegende Arbeit realisiert mit Motiven etwas Vergleichbares. [Sang-Uk 2002] führt das Konzept der *Candidate Spots* für die Musterdokumentation und -selektion ein. In [Pree 1994] wird von *Frozen Spots* und *Hot Spots* im Kontext von Frameworks gesprochen. *Hot Spots* sind im Prinzip mit *Candidate Spots* vergleichbar. Das neue Verfahren TrAQ realisiert dieses Konzept durch Einführung von Annotationen. Mit [Mayr-Kern+ 2002] und [Wöckl 2002] gemeinsam ist das Verankern von Bedeutung im Quelltext mittels Kommentaren. Das generelle Mittel der Annotationen stimmt mit [JCP175 2004] überein, wenngleich in [JCP175 2004] eine Spracherweiterung eingeführt und keine Kommentarform gewählt wurde. [JCP175 2004] führt weiterhin ein Konstrukt zur Abfrage von Annotationen zur Entwurfs- oder Laufzeit ein, was im vorliegenden Ansatz nicht konkret geleistet, sondern nur beschrieben wurde. Analog zu [Kopi 2004] können neue „Sprachkonstrukte“ in Form von neuen Annotationen mit benötigter, aber bisher noch nicht vorhandener Bedeutung entworfen und eingeführt werden.
Annotationen als konkrete Ausgestaltung eines Bedeutung beschreibenden Mittels wurden in [Flanagan+ 2002] erwähnt. Letzt genannter Ansatz realisiert zudem einen verifizierenden Compiler. Der neue Ansatz integriert eine automatisierte Validierung mit Hilfe von Vor- und Nachbedingungen sowie Unit Tests für Refactoring-Operationen, Geltungsbereichen von Annotationen sowie durch Motivaspekte.

[Baroni+ 2003b] schlägt ein graphisches Werkzeug zur Definition von Mustern vor, um den Anwender von den Details einer Definitionssprache fernzuhalten. Im vorgestellten Ansatz wird dies erreicht durch exemplarische Musterdokumentation sowie durch Einsatz der vollwertigen und populären Programmiersprache Java für Refactoring-Operationen sowie Vor- und Nachbedingungen bzw. Unit Tests. Analog zu [PBE 2004a] basiert TrAQ auf der Dokumentation durch Ausführen eines Beispiels. In [PBE 2004a] eingeführte Vorlagen (*Code Templates*) finden sich in der formalen Musterdokumentation dieser Arbeit wieder.

In [Zimmer 1997] wird das Konzept der Transformationsschritte verfolgt, das im vorgestellten Ansatz über die Musterdokumentation Einzug erhält. Die vorliegende Arbeit erlaubt die Auswahl von Mustervarianten, genau wie Zimmer. Auch im Ansatz von Zimmer muss ein Muster vor dessen Anwendung manuelle gewählt werden. Die strikte Trennung der Prozesse Musterdokumentation und Musterselektion bzw. -anwendung findet sich in der vorliegenden Arbeit genauso wie in [Tan+ 2003] wieder.

In [Ó Cinnéide+ 1999a] werden Hilfsfunktionen und Prädikate zur Identifikation von notwendigen Programmbestandteilen für die Musteranwendung eingeführt. Analog dazu werden im vorgestellten Ansatz Annotationen, Vorbedingungen von Transformationen sowie das Verfügbarmachen eines Ausgangsquelltextes in der Musterdokumentation (abstrahiert durch Annotationen, Isomorphe und die Betrachtung abstrakter Syntaxbäume) verwendet. Gemäß [Ó Cinnéide+ 1999a] können Hilfsfunktionen für das Aufstellen von Vor- und Nachbedingungen sowie für die Erkennung von Geltungsbereichen durch eine Dienstschicht bereitgestellt werden. [Ó Cinnéide+ 1999a] zerlegt die Prozesse der Dokumentation, Selektion und Anwendung von Mustern jeweils in Teilprobleme. Der neue Ansatz tut dies ebenso. In der Musterdokumentation wird exemplarisch Schritt für Schritt transformiert. Annotationen werden getrennt von Programmentitäten betrachtet. Die Selektion vergleicht einzelne Annotationen und Anweisungen. Die Musteranwendung basiert auf der schrittweisen Transformation eines gegebenen Quelltextes.

Genau wie in [Ó Cinnéide+ 1999] müssen Quelltexttransformationen manuell definiert werden. Die vorliegende Arbeit lindert diesen Umstand etwas ab, indem die Definition von Transformationen auf einer exemplarischen Musteranwendung basiert.

In [Philippow+ 2004] wird das Konzept der Positiv- und Negativkriterien zur Selektion eines Musters beschrieben. Der vorliegende Ansatz leistet ebendieses durch entsprechende Gewichtung übereinstimmender und abweichender Annotationen beim Vergleich dieser im Rahmen der Musterselektion. [Guéhéneuc+ 2001] sucht nach Indikationsstrukturen und Defekten, um Muster zu selektieren. Umgesetzt wird dies in TrAQ mit Hilfe von Annotationen, die mit der Musterdokumentation abgeglichen werden. Die in [Guéhéneuc+ 2001] genannte Klassifikation von Defekten ist im vorliegenden insofern abgebildet als dass durch Annotationen Defekte ausgedrückt werden können.
Vergleichbar mit [Pree 1994] können im Ansatz dieser Arbeit unveränderliche und veränderliche Teile bei der Musterdokumentation und -anwendung unterschieden werden. Pree bezeichnet dies mit *Frozen Spots* und *Hot Spots*, die allerdings nur auf Frameworks angewandt werden. *Hot Spots* können insbesondere mit den Annotationen dieser Arbeit ausgedrückt werden. Ähnlich ist das Konzept der Frames in [ANGIE 2003] zu verstehen, das flexible und statische Teile eines Musters definiert, allerdings ohne ein semantisches Ausdrucksmittel bereitzustellen. [ANGIE 2003] behandelt Quelltext als Entität erster Klasse, was in dieser Arbeit ebenfalls umgesetzt ist. In [Pree 1995] wird der Entwickler durch Beispiele aus einer Datenbank inspiriert. Dieser gedankliche Prozess wird im vorliegenden Ansatz weitergeführt, indem eine formale Musterdokumentation gewissermaßen als Quelle der Inspiration für einen Automatismus dient.

In [Tan+ 2003] werden unterschiedliche Typen von Musterparametern erarbeitet. In der vorliegenden Arbeit ist dies in ähnlicher Weise wiederzufinden. Die Struktur des Musters findet über Ausgangs- und Zielquelltext sowie Transformationen Berücksichtigung. Die Semantik wird durch Annotationen abgebildet. Die Überprüfung des Generats findet wie schon beschrieben statt. Der in [Tan+ 2003] genannte Design-Aspekt kann im vorliegenden Ansatz teilweise durch die Auswahl von Mustervarianten beeinflusst werden, gleiches gilt für das Kriterium der Performanz.

Wie in [Kazman+ 2002] und [Motelet 2004] findet das Konzept der Workbench bzw. des kontextuellen Hilfesystems zur Unterstützung des Entwicklers im vorgestellten Ansatz ebenfalls Berücksichtigung. Es wurde in Grundzügen skizziert. Analog zu [Mayr-Kern+ 2002] und [Wöckl 2002] wird ein Mechanismus zur Anwendung des Ansatzes in die Entwicklungsumgebung eingebunden. Eine Schnittstelle nach außen, wie sie in den beiden genannten Ansätzen existiert, wurde allerdings nicht vorgesehen, wenngleich dies möglich ist. Ebenfalls nicht weiter im Rahmen dieser Arbeit betrachtet wurde die Möglichkeit, bereits angewandte Muster zu erkennen. Durch Berücksichtigung von Annotationen wäre dies für den vorgestellten Ansatz relativ leicht für Muster möglich, die annotationsbasiert angewandt wurden. Die Unabhängigkeit von der Programmiersprache wiederum ist den Ansätzen gemeinsam. Dies ist für den aktuellen Ansatz gegeben, sofern die Sprache vergleichbar mit Java ist. Das trifft beispielsweise auf C# oder Delphi zu.

Von den bisher betrachteten Verfahren wurden einige als nachteilig angesehene Eigenschaften umgangen.

Im Gegensatz zu [Zimmer 1997], [Pree 1994] und [Tan+ 2003] ist eine Einschränkung auf Frameworks nicht gegeben. Vielmehr werden beliebige Quelltexte unterstützt. Die Notwendigkeit einer Demoanwendung gibt es im vorgestellten Ansatz ebenfalls nicht. In [Zimmer 1997] müssen Beziehungen zwischen Entwurfsmustern definiert werden. Diese Notwendigkeit besteht im aktuellen Ansatz nur bedingt. Nur für Mustervariationen gibt es Abhängigkeiten, die aber durch getrennt abgelegte Musterdokumentationen aufgelöst werden. Bei Fehlern in einer Dokumentation zu einem Muster müssen allerdings die Dokumentationen zu den Varianten ebenfalls angefasst werden. [Zimmer 1997] fordert die Definition von Nachbedingungen. Diese dienen im beschriebenen Ansatz dazu, die Qualität von Quelltexttransformationen zu erhöhen. Nachbedingungen sind ein

notwendiges Übel, sie verringern die Wahrscheinlichkeit, dass eine inkorrekte Transformation unentdeckt bleibt. In [Zimmer 1997] wird die komplett manuelle Auswahl eines Musters gefordert, der aktuelle Ansatz schwächt die manuelle Komponente zur Musterauswahl erheblich ab.

[Eden+ 2004] besitzt eine Einschränkung derart, dass nicht alle Muster beschreibbar sind. Die vorliegende Arbeit erlaubt das formale Beschreiben aller Muster, da über Annotationen beliebige Bedeutungen ausgedrückt werden können. Transformationen werden über vollwertige Programmlogik umgesetzt, die prinzipiell als uneingeschränkt zu werten ist. Im Gegensatz zu [Eden+ 2004] und [Taibi+ 2003] wurde eine zu formale Spezifikation vermieden, indem bei der Dokumentation ein beispielhaftes Verfahren bereitgestellt wurde. Auch wurde die in [Eden+ 2004] gegebene unterschiedliche Gewichtung von Struktur- und Verhaltensaspekten nicht umgesetzt. In [Baroni+ 2003b] werden Strukturaspekte- und Lösungsabsichten eines Musters mit Hilfe einer einzigen Beschreibungsmöglichkeit vermischt. Der vorliegende Ansatz stellt unterschiedliche, leicht von einander abgrenzbare Mittel dafür bereit. Zum einen sind dies Annotationen, die explizit und abgrenzbar zu einer gegebenen Struktur (exemplarischer Ausgangsquelltext) Lösungsabsichten hinzufügt. Zum anderen wird die Struktur eines Musters im Rahmen eines auf Beispiel basierenden Transformationsprozesses quasi auf natürliche Weise entwickelt.

[Guéhéneuc+ 2001] verwendet eine exotische Sprache zur Dokumentation von Mustern, [Sang-Uk 2002] verwendet das unter Entwicklern nicht verbreitete Prolog, [ANGIE 2003] benutzt eine nichtstandardisierte Beschreibung, die nicht ohne Weiteres in eine Entwicklungsumgebung integriert werden kann. Dies konnte durch das beispielbasierte Verfahren zur Erlangung einer Musterdokumentation sowie durch duale Annotationen und das ausschließliches Verwenden von vollwertigem Java-Code für Refactoring-Operationen und Vor- bzw. Nachbedingungen bzw. Unit Tests umgangen werden. [Kopi 2004] verwendet mit einem *Drop-In-Compiler* eine Komponente, die Standards umgeht. Der neue Ansatz kann als Ergänzung zu bestehenden Mechanismen verstanden werden und unterstützt alle relevanten Technologien (Compiler, Sprachkonstrukte).

Die in [Leavens+ 2003] und auch in [JCP175 2004] eingeführten Mittel zur Quelltextannotation erlauben nicht die Annotation beliebiger Programmentitäten. So können beispielsweise keine Blöcke annotiert werden. Der vorliegende Ansatz hebt diese Einschränkung durch Verwendung von Kommentaren, anstatt von Sprachkonstrukten erster Klasse, auf.

Der in [Sang-Uk 2002] und [Kazman+ 2002] eingeschlagene Weg, einen Vergleich korrelierter Programmversionen vorzunehmen, bedeutet eine zu geringe Anwendbarkeit des Ansatzes, weil diese Programmstände schwer ermittelbar und verfügbar sein müssen. Dies konnte in dieser Arbeit durch Annotationen sowohl in Musterdokumentation als auch im Quelltext umgangen werden. Allerdings könnte es für zukünftige Betrachtungen interessant sein, die in [Kazman+ 2002] vorgeschlagene Betrachtung von Protokolldateien, etwa die aus einem Versionsverwaltungssystem, unterstützend hinzuzuziehen, um die Mächtigkeit von TrAQ zu erhöhen.

In [Philippow+ 2004] müssen Erkennungsregeln für Muster auf nichttriviale Weise definiert werden. Im Rahmen dieser Arbeit wurde die Mustererkennung nur im Ansatz für die Musterselektion behandelt. Die Suche nach Indikationen für oder gegen ein Muster wird durch Annotationen vereinfacht und durch Isomorphe leistungsfähiger gestaltet. Der Abgleich von Annotationen ist zwar auch nichttrivial, muss aber grundsätzlich nur einmal gelöst werden. Eventuell unterschiedliche Gewichtungen von Positiv- und Negativindikatoren müssen allerdings pro Muster ermittelt und ausbalanciert werden.

In [Ó Cinnéide+ 1999] wird in einem gegebenen Quelltext mit dem Begriff des *Precursors* genau ein Einstiegspunkt (Ankerpunkt) für die Musteranwendung ermöglicht. Dies entspricht quasi nur einer annotierten Programmstelle für die Musterselektion im Rahmen dieser Arbeit. Durch die Möglichkeit, mehrere Annotationen vorzunehmen und weitere Annotationen automatisiert abzuleiten, können

viele Ankerpunkte realisiert werden. Jeder weitere Ankerpunkt bedeutet dabei eine Erhöhung der Qualität der Musteranwendung, weil diese so flexibilisiert wird.

Der Ansatz von [Flanagan+ 2002] ist nicht auf Muster spezialisiert und damit einerseits umfassender als die aktuelle Arbeit, andererseits weniger leistungsfähig auf dem speziellen Gebiet der Muster. In [Flanagan+ 2002] wird ein nur eingeschränkter Wortschatz für das Benennen von Bedeutungen angeboten, der den Einsatz eines spezifischen Compilers notwendig macht. Annotationen hingegen können beliebige Bedeutungen ausdrücken und bedürfen keines angepassten Compilers, wenngleich ein eigenes Programm geschrieben werden muss, was den Ansatz umsetzt.
Aufgrund der allgemeinen Ausrichtung von [Flanagan+ 2002], existieren nur stark eingeschränkte Verifikationsmöglichkeiten. Im vorgestellten Ansatz bieten Geltungsbereiche von Annotationen, Vor- und Nachbedingungen zu Transformationen, die Aufteilung von Transformationen in kleinteilige und leicht zu erfassende Refactoring-Operationen, Unit Tests sowie Motivaspekte die Möglichkeit zur automatisierten Verifikation.

[Mayr-Kern+ 2002] und [Wöckl 2002] beschreiben keine graphische Unterstützung bei der Rollendefinition von Mustern. Die auf einem Beispiel basierende Musterdokumentation in der vorliegenden Arbeit ist zwar nicht grafisch, unterstützt den Dokumentator aber erheblich, weil die Transformation hin zum Zielquelltext auf Basis eines Beispiels vorgenommen wird.

[PBE 2004a] diskutiert einen rein generativen Ansatz zum Anwenden eines Musters. Das Adaptieren eines bestehenden Quelltextes durch Transformation ist im Gegensatz zu TrAQ nicht vorgesehen.

KAPITEL 4

4 Kritische Würdigung

Im vorigen Kapitel wurde die Arbeitsweise des Verfahrens TrAQ ausführlich dargestellt und die Integration in den Entwicklungsbetrieb skizziert. Bei kritischer Würdigung der im Rahmen dieser Arbeit beschriebenen Kernprozesse formale Musterdokumentation, Musterselektion und Musteranwendung können Vor- und Nachteile des vorgestellten Verfahrens festgestellt werden.

4.1 Vorteile von TrAQ

Der TrAQ-Ansatz unterstützt die Arbeit mit Entwurfsmustern auf Quelltextebene. Quelltext ist nach Ansicht des Autors das Hauptwerkstück in der kommerziellen Software-Entwicklung und somit in besonderem Maße relevant.

Im Rahmen der Musterdokumentation wird ein Muster formal dokumentiert. Das bedeutet konkret, es wird eine Beschreibung des Musters erstellt, die informelle Beschreibungen wie in [Gamma+ 1995] ergänzt. Die Musterdokumentation wird insbesondere durch Verwendung von Annotationen formalisiert. Annotationen erlauben das Ausdrücken beliebiger, zuvor festgelegter Bedeutungen, insbesondere von Entwurfsintentionen. Sie sind so gestaltet, dass sie sowohl maschinen- als auch menschenlesbar sind und somit einerseits automatisiert verarbeitet und andererseits vom Menschen leicht erfasst und verstanden werden können. Annotationen stellen daher ein Dokumentationswerkzeug für Quelltext dar, das einen Zusatznutzen zu informellen Quelltextkommentaren bietet. Mit Hilfe der Abstraktion von Annotationen durch Annotationstypen ist eine Definition von Annotationen durch Wiederverwendung bereits bestehender Annotationstypen mit wenig Aufwand möglich. Bereits definierte Behandlerroutinen für Geltungsbereiche erhöhen den Grad der Wiederverwendbarkeit bereits getätigter Definitionen.
Annotationen als wohldefinierter Kommentar innerhalb von Quelltexten sind seit einigen Jahren in Sprachen wie Java oder C# verbreitet. Das kann als Vorteil gesehen werden, da Programmierer somit Zeit hatten, das Konzept der Annotationen zu verinnerlichen und deren Anwendungsbereich für sich selbst zu evaluieren. Die starke Verbreitung von Annotationen in Java und C# kann als Beleg für deren Praxistauglichkeit interpretiert werden.

Der auf dem exemplarischen Transformieren von Quelltext basierende Ansatz der Dokumentation erzwingt das Benennen jedes transformierten, also zwischen Ausgangs- und Zielquelltext veränderten Programmstücks. Ein geeigneter Ausgangsquelltext kann relativ leicht gefunden werden, wenn das Muster bereits informal dokumentiert ist. Die informalen Dokumentationen enthalten meist beispielhafte Quelltexte. Zudem muss für ein zu dokumentierendes Muster ein geeigneter Ausgangsquelltext vorliegen, weil das Muster sonst keines wäre (in verschiedenen Projekten wird mindestens dreimaliges Vorkommen gefordert, damit das Muster als solches bezeichnet wird). Die Benennung der transformierten Teile in Form von Motiven erzeugt eine prägnante Aufzählung der Musterintentionen, quasi der Lösungsabsichten des Musters. Durch die Herausarbeitung von Motivaspekten werden einerseits signifikante Teile einer annotierten, weil durch Musteranwendung transformierten Programmentität benannt. Zum Zweiten offenbart ein Vergleich der mit Aspekten versehenen Motive und der insgesamt eingeführten Motive die Programmentitäten, welche im Rahmen der Musterdokumentation nicht transformiert wurden. Derartige Programmstellen zeigen Potentiale für Variationen des Ausgangsquelltextes in Verfeinerungsphasen auf.
Durch das Benennen von unabhängigen Motiven wird einerseits die Qualität der Musterbeschreibung insgesamt erhöht, da so verschiedene, nicht korrelierte Lösungsabsichten des Musters genau dokumentiert werden. Weiterhin erleichtern unabhängige Motive die Musterselektion, da ein Vergleich zwischen einer Annotation in einem gegebenem Quelltext und den Annotationen in der Musterdokumentation sich auf das annotierte Motiv sowie die davon nicht unabhängigen Motive beschränken kann.

Die Einführung von Teilmotiven erlaubt die feinteilige formale Dokumentation von nichtatomaren Strukturen wie Schleifen. Wird für eine Schleife als Ganzes ein Motiv angebracht, können die innerhalb der Schleife befindlichen Programmentitäten durch vom eigentlichen Motiv abhängige Teilmotive dokumentiert werden.

Zu jeder manuell transformierten Stelle im Quelltext fordert der Prozess der formalen Musterdokumentation das Definieren einer formalen Transformation. Eine solche Transformation kann im Rahmen der Musteranwendung verwendet werden, um das Muster Stück für Stück anzuwenden. Eine Transformation basiert auf einer Sequenz von feingranularen Refactoring-Operationen. Dies erlaubt es, eine erstmals definierte Refactoring-Operation für verschiedene Transformationen wieder zu verwenden. Einige Refactoring-Operationen sind zudem in modernen Entwicklungsumgebungen verankert und können schon heute auf Knopfdruck automatisiert ausgeführt werden.

Mit Hilfe der Verfeinerungsphasen wird einerseits veränderlichen Ausgangsquelltexten und andererseits Mustervariationen Rechnung getragen. Insbesondere durch die Einbeziehung von variierten Ausgangsquelltexten wird gewürdigt, dass jeder zukünftig im Rahmen der Musterselektion und -anwendung betrachtete Quelltext potentiell unterschiedlich von den zuvor betrachteten sein kann.

Zur Verifikation der Funktionsfähigkeit der Transformationen in der Musterdokumentation ist es möglich, automatisierte Unit Tests zu implementieren. Diese können beliebig oft und quasi ohne zusätzlichen Zeitaufwand jederzeit ausgeführt werden. Der Charakter von Unit Tests unterstützt einen ständig wachsenden Katalog von Tests, der wiederum mit wachsendem Umfang eine bessere Qualitätssicherung bedeutet.

Auf Basis der formalen Musterdokumentation kann ein Katalog von Mustern erstellt werden. Jedes im Katalog vertretene Muster kann bei der Selektion und der danach folgenden Musteranwendung berücksichtigt werden.

Die Musterselektion basiert auf einem vorgegebenen, mit Annotationen versehenen Quelltext. Diese Annotationen können einerseits manuell angebracht werden. Mögliche Hilfestellungen durch Werkzeuge wurden in dieser Arbeit beschrieben. Ebenso ist es möglich, automatisiert Annotationen anzubringen. Dies ist einerseits umsetzbar für markante Stellen im Programmcode, etwa für bekannte Schnittstellen oder Datentypen. Andererseits können Annotationen automatisiert für Programmentitäten angebracht werden, die in Beziehung zu einer anderen, bereits manuell oder automatisiert annotierten Programmentität stehen. Hier hilft erstens die generell in einer Programmiersprache wie Java vorhandene Abhängigkeit zwischen Deklaration und Aufruf bzw. Zuweisung oder Referenz. Zweitens können Abhängigkeiten über in der Musterdokumentation mit demselben Motiv versehenen Programmentitäten festgestellt werden, unterstützt durch Motivaspekte und die dadurch ausgedrückten signifikanten Teile der jeweils annotierten Programmentität.

Ein annotierter Quelltext ist durch die Annotationen um Beschreibungen ergänzt. Da eine Annotation aus einem maschinen- und einem menschenlesbaren Teil besteht, ergänzt eine Annotation sowohl die formale als auch die informale Dokumentation des Quelltextes. Annotationen geben durch ihre definierte Gestalt eine Konvention vor, die zu einheitlicher Dokumentation führen kann.

Um bei der Musterselektion beliebig variierende Quelltexte besser handhaben zu können, wurde das Konzept der Isomorphe eingeführt. Auf Ebene der Annotationen wurde vorgeschlagen, sowohl Synonyme als auch eine Ähnlichkeitsmatrix zwischen Annotationen aufzustellen.

Mit Hilfe des Vergleiches eines gegebenen Quelltextes und allen im Katalog verzeichneten Musterdokumentationen kann eine Rangliste erstellt werden. Diese gibt an, in welchem Maße die

Anwendbarkeit aller verzeichneten Muster für den Quelltext angebracht erscheint. Beim Vergleich kann mit Hilfe der Annotationen abstrahiert werden von Zeilennummern und vom zeichenweisen Gegenüberstellen zweier Quelltextteile. Isomorphe und Synonyme tragen zur Abstraktion von Vergleichen auf Basis konkreter Programmentitäten ebenfalls bei.

Nachdem ein Muster aufgrund einer in der Musterselektion ermittelten Rangliste gewählt wurde, kann es angewandt werden. Die Informationen aus der Musterselektion können dafür ausgebeutet werden. Das betrifft insbesondere die Übereinstimmungen und Unterschiede zwischen Quelltext und Musterdokumentation. Zu den Übereinstimmungen können Motive zum Muster nebst abhängigen Motiven ermittelt werden. Diese Motive besitzen eine Verknüpfung zu Transformationen, welche nach Ausführung im Falle einer korrekten Transformation die Anwendung des Musters bewirken. Transformationen sind automatisiert ausführbar und über Vorbedingungen sowie Nachbedingungen (Unit Tests, Prüflogik) verifizierbar.
Dadurch, dass bei der Musteranwendung nicht pauschal, sondern individuell transformiert wird, wird einem bereits in Teilen vorhandenen Muster Rechnung getragen. Eine gesonderte Feststellung dessen durch eine unabhängige Mustererkennung ist nicht erforderlich.

Aufgrund der durchgängig in allen Musterphasen verwendeten Entitäten Annotationen sowie Quelltext werden die Phasen von der Dokumentation bis hin zur Anwendung auf eine gemeinsame Basis gestellt. Sie werden nicht getrennt von einander behandelt, sondern basieren auf einander und verschmelzen so.

4.2 Nachteile von TrAQ

Die formale Musterdokumentation als Ausgangspunkt von TrAQ erfordert als Autor einen kompetenten, deutlich überdurchschnittlich erfahrenen und begabten Entwickler. Der Aufwand zur Erstellung einer Musterdokumentation ist hoch. Dies ist analog zu anderen Ansätzen wie [Mayr-Kern+ 2002] und [Wöckl 2002]. Generell lässt sich dieses Kompetenzproblem nach Ansicht des Autors nicht vermeiden. Immerhin können mit Hilfe der formalen Musterdokumentation die notwendigen Musterkenntnisse in der Selektions- und Anwendungsphase reduziert werden.

In den Dokumentationsphasen A bis C müssen Annotationen zum selben Motiv über alle Mustervarianten hinweg konsistent durchnumeriert werden. Das bedeutet einen zusätzlichen Aufwand für den Dokumentator. Außerdem müssen unterschiedliche Motive in Mustern differenziert werden und gleiche Motive einheitlich benannt werden. Das einzuhalten bedeutet das Prüfen bereits bestehender Motive. Dies kann zwar durch eine automatisierte Suche teilweise vereinfacht werden, bedeutet aber einen nicht zu vermeidenden manuellen Restaufwand.
Fehler in der Musterdokumentation wirken sich negativ auf die Musterselektion und Musteranwendung aus. Die Möglichkeit der Korrektur der Musterdokumentation ist gegeben, wenngleich deren Komplexität nicht unterschätzt werden darf. Die Einführung von automatisierten Prüfmechanismen, wie Unit Tests, scheint erforderlich um praktikabel eine hohe Qualität von Musterdokumentationen gewährleistet zu können.

Das Verfahren bedingt, dass jede Mustervariation explizit definiert werden muss, sofern sie nicht durch Isomorphe oder Synonyme abgedeckt wurde. Weiterhin müssen unterschiedliche Ausgangsquelltexte entweder durch Isomorphiedefinitionen oder durch Angabe einer beispielhaften Transformation abgedeckt werden. Eine befriedigende Abdeckung zu erhalten ist zeitaufwändig. Nach Meinung des Autors ist dieser Aufwand leider nicht vermeidbar.

TrAQ basiert auf im Quelltext vorhandenen Annotationen. Diese anzubringen bedeutet einen Zusatzaufwand. Da potentiell jeder Programmteil annotiert werden kann, ist der durchschnittliche Anwender auf Hilfestellungen angewiesen. Diese zusätzliche Komplexität birgt das Potential manche Anwender abzuschrecken.

Der Abgleich von Annotationen im Quelltext und denen in der Musterdokumentation kann komplex werden, wenn die zu einem Motiv annotierten Programmentitäten in ihrem syntaktisch Typ (etwa: Methodendeklaration, Methodenaufruf, Variablendeklaration, Zuweisung) nicht von einander abgrenzbar sind. Das Aufstellen einer entsprechenden Prüflogik auf Basis der Analyse von abstrakten Syntaxbäumen ist zwar technisch einfach möglich, erfordert jedoch eine erhebliche Mehrarbeit bei der Musterdokumentation.
Genau wie bei Spracherweiterungen sind auch die in dieser Arbeit genannten Ausprägungen für Annotationen nicht standardisiert. Allerdings ist dem Autor kein Standard für das Ausdrücken von Semantik in Quelltexten bekannt. Dieser muss sich, analog zu [JCP175 2004], im Laufe der Zeit erst einstellen.
Im Quelltext angebrachte Annotationen können veralten oder falsch werden. Einerseits wenn die Annotationsdefinition sich ändert, andererseits wenn der ursprünglich annotierte Quelltext verändert wird. Dies zu erkennen ist zwar möglich, erfordert aber manuelle Nacharbeiten.

Die Gewichtung von Unterschieden und Gemeinsamkeiten zwischen Musterdokumentation und gegebenem Quelltext im Rahmen der Musterselektion kann auf beliebig viele Arten vorgenommen werden. Ein optimales Verfahren hierzu zu benennen erscheint aufgrund der Komplexität des Problems nicht möglich. Vielmehr ist ein möglichst guter Algorithmus zu entwickeln, der wahrscheinlich nur durch langfristige empirische Untersuchungen entstehen kann.

Bei der Musterselektion ist es nach Meinung des Autors notwendig, dass der Mensch die letzte Entscheidung über das danach anzuwendende Muster zu treffen hat. Dies kann nachteilig beurteilt werden, weil es eine gewisse Kenntnis über die zur Auswahl stehenden Musters erfordert.

Aufgrund der Komplexität von Quelltexttransformationen kann eine korrekte Musteranwendung nicht garantiert werden. Allerdings können Quelltexttransformationen wohl nicht vollständig automatisiert werden. Es bleibt – unabhängig von diesem Verfahren – die Unsicherheit, dass eine Transformation fehlerhaft durchgeführt wird und etwa zu unkompilierbaren oder sogar semantisch falschen Quelltexten führt. Die Sicherstellung der Korrektheit des Quelltextes nach automatisierter Transformation ist für den Anwender eine schwierige Aufgabe. Nicht kompilierbare Quelltextteile sind zwar leicht zu identifizieren. Die Fehlerbehebung für diese setzt allerdings ein gewisses Verständnis über das anzuwendende Muster voraus, das nur durch im Rahmen des Verfahrens angebotene Informationen zum Muster unterstützt werden kann. Kompilierbare, aber dennoch inkorrekte Quelltextteile sind weit schwieriger zu identifizieren. Sie zu erkennen bedeutet die Untersuchung des gesamten Quelltextes, der für die Musteranwendung relevant ist. Die relevanten Teile können immerhin aufgrund der annotierten Quelltextteile sowie davon abhängigen und von während der Transformation berührten Teilen benannt werden.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Mit der vorliegenden Arbeit wurde ein Verfahren namens TrAQ zur Unterstützung von Entwurfsmustern und verwandten Mustern bei der Entwicklung auf Quelltext vorgestellt. Es wurde herausgearbeitet, was die aktuellen Probleme mit Entwurfsmustern sind. Dazu wurde einerseits ein Fragebogen erstellt, an Personen aus dem Gebiet der Software-Entwicklung verteilt und ausgewertet. Andererseits wurden bestehende Ansätze untersucht, um die vorhandenen Lösungen und die damit wiederum verbundenen Einschränkungen und Probleme zu diskutieren. Es wurde festgestellt, dass die Unterstützung des Entwicklers bei der Arbeit mit Mustern durch bestehende Verfahren nicht ausreichend gegeben ist. Aus Zeitgründen konnte keine praktische Evaluierung des Ansatzes vorgenommen werden.

Im Themenkreis der Arbeit wurden Begriffsdefinitionen vorgenommen und neue Konzepte eingeführt. Insbesondere wurden Motive als abstraktes Ausdrucksmittel und duale Annotationen samt Annotationstypen als konkretes Mittel benannt.

Zur Auflösung der Forschungsfrage wurden formale Dokumentationen von Quelltext vorgenommen. Erstens wurde eine formale Musterdokumentation vorgeschlagen, zweitens die Annotation eines für die Musterselektion und -anwendung relevanten Quelltextes.
Beginnend mit der Musterdokumentation wurden darauf aufbauend die Prozesse der Musterselektion und -anwendung beschrieben. TrAQ stützt sich auf eine formale Musterdokumentation, die als notwendig angesehen wird, um die folgenden Prozesse zu unterstützen. Im Rahmen dieser werden Annotationen eingebracht und definiert. Die Annotationen repräsentieren die essentiellen Lösungsmotive zu einem Muster in präziser Form. Analog zur Annotation der Musterdokumentation findet die Annotation eines gegebenen Quelltextes statt, für den Muster selektiert und angewandt werden sollen. Es wurde diskutiert, dass anstatt von Aussagen über Lösungen auch Aussagen aus anderen Perspektiven, wie der Problemseite oder der Anforderungsseite, sinnvoll sein können und vom Verfahren unterstützt werden (vgl. [Meffert+ 2007b] und [Meffert 2006a]).

Die Aufstellung einer formalen Musterdokumentation hilft somit beim Verständnis, aus welchen Teilen ein Muster letztendlich besteht. Durch die Einführung einer semantischen Ebene, die technisch und fachlich in Form von Motiven eingesetzt werden kann, wurden informale Musterbeschreibungen, wie sie etwa in [Gamma+ 1995] zu finden sind, formalisiert. Aus Motiven und Annotationen heraus wurden Transformationen begründet, welche wiederum aus feinteiligen Refactoring-Operationen zusammen gesetzt sind. Mit Hilfe von Motivaspekten und Relationen von Programmentitäten desselben Motivs wurden die essentiellen Musterausprägungen explizit und präzise benannt.

TrAQ bietet die Möglichkeit, durch die Dokumentation von immer mehr Mustern den Musterkatalog zu erweitern, und somit die dokumentierten Muster für die Selektion und Anwendung bereitzustellen. Dieser zeitintensive Prozess liegt in der Hand erfahrener Entwickler und Architekten, die damit eine Vorarbeit leisten, um den davon profitierenden Entwicklern sowohl intellektuell als auch zeitlich die Arbeit zu erleichtern. Zur Erleichterung der Musterdokumentation wurde ein beispielhaftes Vorgehen vorgeschlagen. Anhand des *Observer*-Musters wurde das Vorgehen demonstriert, die Allgemeingültigkeit des Ansatzes wurde festgestellt, u.a. durch die zusätzlich genannten Muster *Singleton*, *Composite* sowie *Iterator*. Weiterhin wurde diskutiert wie der Entwickler bei der Annotation von Quelltext unterstützt werden kann.

Das Verwenden von Annotationen aus der Musterdokumentation bei der späteren Entwicklung hilft dem weniger erfahrenen Entwickler, die für den Quelltext geeigneten Muster herauszufinden. Ein Nebenprodukt der Annotation ist die Anreicherung von Quelltext mit Entwicklerintentionen. Diese Intentionen sind sowohl von einem Menschen als auch von der Maschine interpretierbar.
Neben diesem Selektionsprozess profitiert ebenfalls der Prozess der Musteranwendung von dem Abgleich zwischen Musterdokumentation und Quelltext. Die Musteranwendung kann durch entsprechende Musterdokumentationen so flexibel gestaltet werden, dass bei der Anwendung des Musters bereits vorhandene Musterteile im Quelltext berücksichtigt werden. TrAQ bietet insbesondere Arbeitserleichterung bei der Suche nach einem für die Anwendung geeigneten Muster. Der Entwickler wird entlastet vom umfangreichen Studium der immer mächtiger werdenden Literatur zu einzelnen Mustern. Weiterhin helfen automatisiert durchgeführte Transformationen bei der Musteranwendung, da ein Teil der manuellen Implementierungsarbeit wegfällt. Nichtsdestotrotz ist es nur dem Entwickler möglich, die Korrektheit der automatisiert durchgeführten Transformationen zu prüfen.

Die Möglichkeit TrAQ in den Entwicklungsprozess einzubinden, wurde konzeptionell durch die Beschreibung einer Werkzeugintegration beschrieben. Die Formalisierbarkeit der verwendeten Elemente – Musterdokumentation, Annotationen, Quelltext – erleichtert die praktische Umsetzung.

5.2 Ausblick

In diesem Abschnitt wird ein Ausblick auf Aspekte gegeben, die sich im Rahmen dieser Arbeit ergeben haben, aber nicht abschließend beantwortet werden konnten. Weiterhin werden Möglichkeiten benannt, den Ansatz dieser Arbeit einzusetzen oder ihn mit anderen Arbeiten zu kombinieren.

Die exemplarische Transformation eines Ausgangsquelltextes in einen Zielquelltext zur Aufstellung einer formalen Musterdokumentation ist ein elementarer Bestandteil des vorgestellten Verfahrens TrAQ. Während der Transformation ist es wichtig, die Kriterien der Transformation dem aufzeichnenden System deutlich zu machen. Die zukünftige Realisierung eines solchen Systems in Form einer Werkzeugs, am besten in die Entwicklungsumgebung integriert, kann die Erstellung von Musterdokumentationen erheblich erleichtern.

Da die Erstellung von formalen Musterdokumentationen zeitaufwändig ist, ist die Möglichkeit eines Austausches solcher Dokumentationen wünschenswert. Auf einer global verfügbaren Plattform im Internet kann dies mit heutigen Mitteln umgesetzt werden. Gleichzeitig ist eine Kollaborationsplattform denkbar, mit deren Hilfe Musterdokumentationen gegenseitig verbessert werden können. Auf diese Weise können auch Isomorphe, Refactoring-Operationen und Annotationsdeklarationen untereinander getauscht und diskutiert werden.

Die Beurteilung der Güte einer aufgestellten Musterdokumentation kann maschinell nur auf Ebene der Verifikation durchgeführt werden. Möglichkeiten hierzu wurden genannt, insbesondere ergeben sich diese aus dem formalen Charakter von Annotationen und Quelltext an sich. Darüber hinaus könnte versucht werden, etwa mit Hilfe eines Künstlichen Neuronalen Netzes, Konzepte aus verschiedenen Musterdokumentationen zu extrahieren und diese in neu erstellten, noch nicht bewährten Musterdokumentationen zu suchen. Gelänge dies, könnten gemeinsame und abweichende Konzepte ermittelt werden. Ähnlich der Musterselektion, die Gemeinsamkeiten und Unterschiede zwischen Musterdokumentation und gegebenem Quelltext berücksichtigt, könnte so womöglich eine Abweichung von der Norm festgestellt werden, woraus dann mit einer gewissen Wahrscheinlichkeit qualitative Mängel in der neu aufgestellten Musterdokumentation folgerbar wären.

Zur Erhöhung der Wahrscheinlichkeit, dass bei der Musterselektion bereits implementierte Musterteile erkannt werden, ist es denkbar, die Musterdokumentation um Erkennungsroutinen wie

in [Philippow+ 2004] zu erweitern. Potentiell kann so eine höhere Wahrscheinlichkeit bei der Erkennung vorhandener Musterteile realisiert werden. Andererseits sollte es bei einer gut ausgereiften Musterdokumentation nicht passieren, dass die genannten zusätzlichen Erkennungsroutinen Fälle abdecken, die nicht durch die in dieser Arbeit beschriebene Musterdokumentation selbst erkannt werden. Letztendlich können solche Erkennungsroutinen also als eine Art Qualitätssicherung (ähnlich einem »Unit Test«) aufgefasst werden, die beim Finden von Lücken in der Musterdokumentation dienlich sein können. Sie können allerdings auch als eine Möglichkeit angesehen werden, die Zuordnung einer Quelltext-Annotation zu genau einer von mehreren Annotationen desselben Motivs in der Musterdokumentation umfassender zu bestimmen. Die Erkennung dieser Zuordnung ist ein Problem, das nicht bei allen Mustern auftritt, aber aufgrund seiner Komplexität ohne Einbeziehung des Menschen als Entscheidungsinstanz nicht trivial ist.

Bei der Musterselektion ist mit Hilfe eines Bewertungsverfahrens zu ermitteln, wie geeignet ein dokumentiertes Muster für einen gegebenen Quelltext erscheint. Die Bewertung wurde grundsätzlich thematisiert. Interessant wäre die weitergehende Betrachtung von Verfahren zur möglichst automatisierten und adaptierenden Gewichtung von Übereinstimmungen und Abweichungen zwischen Musterdokumentation und Quelltext. Denkbar wären etwa Evolutionäre Algorithmen wie die Genetische Programmierung (vgl. [Meffert+ 2008b]).

Das automatisierte Anbringen von Annotationen wurde grundsätzlich diskutiert. Ansatzpunkte zur Umsetzung wurden genannt. Was noch fehlt ist die Umsetzung selbst mit Hilfe eines Werkzeuges. Annotationen als Ausdrucksmittel sind außerhalb von TrAQ potentiell für viele weitere Anwendungsgebiete interessant. Sie können nicht nur zum Anbringen von Entwurfsintentionen oder anderen semantischen Aussagen, sondern etwa auch für Steueranweisungen verwendet werden. Da Annotationen leicht maschinenverarbeitbar sind und in der vorgestellten Form keinen neuen Compiler benötigen, eignen sie sich insbesondere für die Steuerung von Programmgeneratoren beliebiger Art. Auch zur Unterstützung von Analyse- und Refactoring-Werkzeugen sind Annotationen denkbar. In diesem Feld erscheint es sinnvoll, Beziehungen zwischen Klassen (Assoziation, Referenz, Komposition…) mit Hilfe von Annotationen auszudrücken. Dies macht insbesondere Sinn, wenn ein Analysewerkzeug diese Informationen ermittelt und automatisch die dazu passenden Annotationen im bereits teilweise annotierten Quelltext anbringt. Bei der Generierung von UML-Diagrammen können die in den Annotationen steckenden Informationen, die über die bisher automatisiert ableitbaren Informationen hinausgehen, ausgenutzt werden. So könnten direkt bei der UML-Generierung aus Quelltext heraus vielfältige Stereotypen erzeugt werden.

Die Musteranwendung, realisiert durch die Transformation von Quelltext, kann hinsichtlich der Aussage über die Korrektheit der ausgeführten Transformationen verbessert werden. Denkbar sind Unit Tests, verifizierende Verfahren, aber auch das Aufstellen von Metriken (vgl. [Balzert 1998, S. 225ff]). Der Vergleich von Metriken des Quelltextes vor und nach der Musteranwendung könnte Aufschluss darüber geben, ob eine korrekte Transformation stattgefunden hat oder nicht. Beispielsweise könnte der in [Balzert 1998, S. 263ff] genannte Goal-Question-Metric-Ansatz zur Prüfung der Software-Qualität zum Einsatz kommen.

Annotationen als relativ neues Mittel des Entwicklers, um Quelltext mit Semantikinformationen anzureichern, müssen sich im Lauf der Zeit erst bewähren. Die Eignung von Annotationen hängt erstens davon ab, wie mächtig Annotationen in Sprachen wie Java eingebunden werden; dazu zählt insbesondere die Syntax, die eine möglichst universelle und dennoch präzise Formulierung von Meta-Informationen gestatten sollte. Zweitens ist es entscheidend, wie stark sich Annotationen als tägliches Mittel bei der Programmierung durchsetzen. Drittens kommt es darauf an, inwiefern Annotationen von populären Entwicklungsumgebungen in Zukunft direkt unterstützt werden. Und viertens ist der wichtigste Aspekt, wie Annotationen interpretiert werden, sowohl vom menschlichen Verwender als auch vom analysierenden System. Eine Annotation ohne Interpretation ist an sich nutzlos.

Zur Erhöhung des Nutzens von Annotationen, sei es zur Abbildung technischer oder fachlicher Aussagen, kann das Aufstellen einer Ontologie hilfreich sein. Diese erlaubt es, verschiedene Aussagen zueinander in Beziehung zu setzen und Überbegriffe zu identifizieren. Mit Hilfe dieser kann eine Ähnlichkeitsmatrix für Annotationen zudem unterstützt werden.

Zur Erkennung der Veralterung von Annotationen kann die Entwicklungsumgebung bei der Modifikation von Quelltext durch den Entwickler oder durch das Ausführen von Plugins prüfen, ob davon annotierte Programmteile betroffen sind. Ist dies der Fall, kann durch Isomorphievergleiche festgestellt werden, ob der modifizierte Teil mit dem ursprünglichen als äquivalent anzusehen ist. Ist dies nicht der Fall, kann die Entwicklungsumgebung den Entwickler mit einem Dialog konfrontieren, der eine Prüfung der möglicherweise veralteten Annotation fordert.

Für modellgetriebene Verfahren spielen UML-Diagramme eine bedeutende Rolle. Es ist denkbar, UML-Diagramme mit Annotationen zu versehen, um eine Verknüpfung mit annotiertem Quelltext herzustellen bzw. diesen aus dem UML-Modell zu generieren. Anstelle von Annotationen im Modell könnten auch definierte UML-Stereotypen herangezogen werden.

Zur Reduzierung der Komplexität der Annotation von Quelltext sowie der Musterselektion und -anwendung ist es denkbar, den Ausgangsquelltext vorab derart zu normieren, dass als nicht essentiell angesehene Anweisungen und Deklarationen entfernt werden. Dieser Prozess kann einerseits kontextfreie Operationen beinhaltet, beispielsweise das Entfernen von nicht gerufenen Methoden. Andererseits sind kontextspezifische Normierungsregeln denkbar, die mit dem anzuwendenden Muster verknüpft sind (siehe auch [Fayad 2007i]). Insbesondere letzteres kann helfen, das Problem, welche Anweisungen im Quelltext annotiert werden müssen, zu reduzieren.

Die technische Entwicklung von Java war während der Erstellung dieser Arbeit rasant. Damals hatten die Java Versionen 1.3 und 1.4 lange Bestand. Die Version 1.4 war zum Beginn der Erstellung dieser Arbeit die aktuelle. Doch innerhalb einer im Vergleich zu früheren Intervallen der Produkteinführung kurzen Zeit wurden die Versionen 5 (entspricht 1.5) und 6 (auch 1.6 genannt) eingeführt. Kurz nachdem Java Version 6 publiziert wurde, gab es als inoffiziell Beta-Version bereits Java in Version 7. Der Zug von Version 1.4 auf Version 7 (Beta) vollzog sich innerhalb von ca. zwei Jahren. Unter der Annahme, dass der Fortschritt bei der Weiterentwicklung von Programmiersprachen in gleichem oder höherem Tempo vonstatten geht, ist es schwierig, den Besonderheiten dieser Weiterentwicklungen Rechnung zu tragen. Das betrifft insbesondere Annotationen, denen ein Geltungsbereich zugeordnet werden muss, der auf den in der Programmiersprache verfügbaren Konstrukten basiert. Gleiches gilt für die Analyse von abstrakten Syntaxbäumen.

Trotz und gerade wegen der Komplexität des Themas der Quelltextanalyse, die für anspruchsvolle Transformationen nach Auffassung des Autors das Erkennen von Entwurfsintentionen mit einschließen muss, erscheint es lohnenswert, Semantik mit Hilfe von Annotationen oder anderen geeigneten Mechanismen im Quelltext auszudrücken und somit eine maschinelle Auswertung zu ermöglichen. Aus Sicht des Autors erscheint der einzige Weg dies zu umgehen, das Verfahren zur Entwicklung von zeitgemäßer Software auf eine höhere Ebene zu verlagern. Ansätze hierzu sind bereits seit einiger Zeit in Arbeit, etwa die modellgetriebene Entwicklung. Die Arbeit mit einem Modell als abstrakter Instanz scheint allerdings ebenfalls von einer explizit mitgeführten Semantik zu profitieren und angewiesen zu sein, worauf auch die Verwendung von Stereotypen in UML-Diagrammen hinweist. Die Vorteile der Abstraktion, komplizierte Sachverhalte und Detailfragen möglichst wegzukapseln, sind zugleich deren Nachteile. Erstens verhindert eine Abstraktion ein umfassendes Verständnis über das zugrunde liegende System. Zweitens sind die Eingriffsmöglichkeiten bei prinzipiell zu erwartendem Anpassungsbedarf beschränkt. Die industrielle Software-Entwicklung ist seit deren Bestehen nach wie vor auf die Arbeit mit Quelltext fokussiert, so dass ein Paradigmenwechsel noch in Frage steht.

Anhang A – Singleton

Ausgangsquelltext

```
010     public class Klient {

020       public void init() {
030 a       ZentralVerwalter verwalter = new ZentralVerwalter();
040         int zeit = verwalter.getTime();
050         ...
060       }

070       ...

080     }

100     public class ZentralVerwalter {

110       private static int instanzen;

120 a     public ZentralVerwalter() {
130 a       if (instanzen > 0) {
140 a         throw new RuntimeException("Maximal eine Instanz erlaubt");
150 a       }
160 a       instanzen++;
170 a     }

180       public int getTime() {
190         return 4711;
200       }

210     }
```

Abbildung 141: Ausgangsquelltext für das Singleton.

Zielquelltext

```
010     public class Klient {

020       public void init() {
030 a       ZentralVerwalter verwalter = ZentralVerwalter.getInstance();
040         int zeit = verwalter.getTime();
050         ...
060       }

070       ...

080     }

100     public class ZentralVerwalter {

110 a     private static ZentralVerwalter instanz = new ZentralVerwalter;

120 a     private ZentralVerwalter() {
130 a     }

140 a     public static ZentralVerwalter getInstance() {
150 a       return instanz;
160 a     }

170       public int getTime() {
180         return 4711;
190       }

200     }
```

Abbildung 142: Zielquelltext für das Singleton.

Motive

a) Erlaube maximal eine Klasseninstanz

Geltungsbereiche pro Motiv

Zeile (Ausgangsquelltext)	Motiv	Geltungsbereich
030	a	Konstruktoraufruf
120 – 170	a	Konstruktordeklaration

Tabelle 23: Geltungsbereiche für Motive des Singleton.

Anhang B - Composite

Ausgangsquelltext

```
010     public class Klient {

020 ad     public static List elem;

030        public Klient() {
030          elem = new Vector();
040          Grafik g = new Grafik();
050          Bild p = new Bild();
060 ad       elem.add(p);
070 ad       elem.add(new Text("Text"));
080 ad       elem.add(new Linie(8, 1, 9, 14));
090 abc      g.create();
100        }

110     }

200 abe public class Grafik {

210 abc   public void create() {
220 ace     int size = Klient.elem.size();
230 ace     for (int i=0; i < size; i++) {
240 abce      Object o = Klient.elem.get(i);
250 abce      Class clazz = o.getClass();
260 abce      if (clazz == Bild.class) {
270 abce        ( (Bild) o).paint();
280 abce      }
290 abce      else if (clazz == Text.class) {
300 abce        ( (Text) o).print();
310 abce      }
320 ace       else { ... }
330 ac      }
340       }

350     }

400 ab  public class Bild {

410 ac    public void paint() {
420         // Zeichenroutine für das Bild
430         ...
440       }

450     }

500 ab  public class Text  { ... } // analog zu Klasse Bild

600 ab  public class Linie { ... } // analog zu Klasse Bild
```

Abbildung 143: Ausgangsquelltext für das Composite.

Zielquelltext

```
010      public class Klient {
020       public Klient() {
030          Grafik g = new Grafik();
040          Bild p = new Bild();
050 ad       g.add(p);
060 ad       g.add(new Text("Text"));
070 ad       g.add(new Linie(8, 1, 9, 14));
080 abc      g.operation();
090        }
100      }
200 abe public class Grafik implements IComponent {
210 ad     private List elem = new Vector();
220 ad     public void add(IComponent c) {
230 ade      if(c.getClass().isAssignableFrom(Grafik.class)){
240 ade       throw new Exception("Grafik nicht zu Grafik hinzufügen!");
250 ade      }
260 ad       elem.add(c);
270 ad     }
280 ac      public void operation() {
290 ac       for (int i=0; i < elem.size(); i++) {
300 abc        IComponent c = (IComponent) elem.get(i);
310 abc        c.operation();
320 ac       }
330         }
340      }
400 ab   public class Bild implements IComposite {
410 ad     private List elem = new Vector();
420 ad     public void add(IComponent c) {
430 ad       elem.add(c);
440 ad     }
450 ac     public void operation() {
460 ac       paint();
470 ac       for (int i=0; i < elem.size(); i++) {
480 ac         IComponent c = (IComponent) elem.get(i);
490 ac         c.operation();
500 ac       }
510 ac     }
520 ac     public void paint() {
530          // Zeichenroutine für das Bild
540          ...
550        }
560      }
600 ab   public class Text  implements ... // analog zu Klasse Bild
700 ab   public class Linie implements ... // analog zu Klasse Bild
800 ab   public interface IComponent {
810 ad     void add(IComponent c);
820 ac     void operation();
830 ab   }
900 abe public interface IComposite extends IComponent { } //erbt nur!
```

Abbildung 144: Zielquelltext für das Composite.

Motive

a) Unterstütze rekursive Komponenten
b) Vereinheitliche Komponenten
c) Führe Operation einer Komponente aus
d) Füge Komponente zu einer anderen hinzu
e) Stelle Basiskomponente bereit

Geltungsbereiche pro Motiv

Zeile (Ausgangsquelltext)	Motiv	Geltungsbereich
020	d	Statische Variablendeklaration
060	d	Listenoperation »Hinzufügen«
070	d	Listenoperation »Hinzufügen«
080	d	Listenoperation »Hinzufügen«
090	b	Methodenaufruf
090	c	Methodenaufruf
200	b	Klassendeklaration
200	e	Klassendeklaration
210 – 330	c	Jede Anweisung oder lokale Deklaration
240 – 310	b	Jede Anweisung oder lokale Deklaration
220 – 320	e	Jede Anweisung oder lokale Deklaration
400	b	Klassendeklaration
410	c	Methodendeklaration
500	b	Klassendeklaration
600	b	Klassendeklaration

Tabelle 24: Geltungsbereiche für Motive des Composite.

Abkürzungsverzeichnis

A_i	Bestimmte Annotation aus einer Menge von Annotationen
M	Muster
M'	Muster, Variante zu M
M_i	Bestimmtes Muster aus einem Musterkatalog
Q_a	Ausgangsquelltext
Q_a'	Ausgangsquelltext, Variante zu Q_a
Q_b	Ausgangsquelltext, potentiell unterschiedlich zu Q_a
Q_z	Zielquelltext
Q_z'	Zielquelltext, potentiell unterschiedlich zu Q_z
Q_z''	Zielquelltext, potentiell unterschiedlich zu Q_z und zu Q_z'
T	Quelltexttransformation
T'	Quelltexttransformation, potentiell verschieden zu T
T''	Quelltexttransformation, potentiell verschieden zu T und T'
$T(Q_a) \rightarrow Q_z$	Transformation von Quelltext Q_a nach Quelltext Q_z
$T(Q_a, M) \rightarrow Q_z$	Transformation von Quelltext Q_a unter Anwendung von Muster M nach Quelltext Q_z
T_i	Teiltransformation von T
Ti	Transformation i, verschieden von anderen Transformationen bei der Musterdokumentation oder -anwendung
T_{ij}	Teiltransformation j innerhalb von Teiltransformation i.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Albin-Am.+ 2001a] Albin-Amiot, Hervé; Guéhéneuc, Yann-Gaël: Meta-modeling Design Patterns: Application to Pattern Detection and Code Analysis. Workshop on Adaptative Object-Modelsand Metamodeling Techniques. ECOOP (European Conference on Oriented Programming), 2001. http://www.emn.fr/x-info/jussien/publications/albin-amiot-ASE01.pdf. Abrufdatum: 22.09.2003

[Alexander 1979] Alexander, Christopher: The Timeless Way of Building (1979)

[Alexander+ 1995] Alexander, Christopher; Ishikawa, Sara; Silverstein, Murray: Eine Muster-Sprache: Städte, Gebäude, Konstruktion. Löcker Verlag, Wien (1995)

[ANGIE 2003] Delta Software Technology GmbH: ANGIE. http://www.d-s-t-g.com/neu/pages/pagesger/et/common/techn_angie_frmset.htm. Abrufdatum: 25.11.2003

[Bachrach+ 2003] Bachrach, Jonathan; Playford, Keith: Java Syntactic Extender. http://www.ai.mit.edu/~jrb/jse/index.htm. Abrufdatum: 01.12.2003

[Balzert 1996] Balzert, Helmut: Lehrbuch der Software-Technik: Software-Entwicklung. Band 1. Spektrum Akademischer Verlag, Heidelberg u.a. (1996)

[Balzert 1998] Balzert, Helmut: Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung. Band 2. Spektrum Akademischer Verlag, Berlin (1998)

[Baroni+ 2003a] Baroni, Aline Lúcia; Guéhéneuc, Yann-Gaël; Albin-Amiot, Hervé: Design Patterns Formalization (2003). http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf . Abrufdatum: 25.06.2004

[Baroni+ 2003b] Baroni, Aline Lúcia; Albin-Amiot, Hervé; Guéhéneuc,Yann-Gaël: Design Patterns Formalization. Research Report 03/3/INFO, Nantes (2003). http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf. Abrufdatum: 02.02.2004

[Beck 2002] Beck, Kent: Test-Driven Development by Example. Addison-Wesley, Reading (2002)

[Boehm 2004] Boehm, Oliver: PatternTesting. http://patterntesting.sourceforge.net/whatis.html. Abrufdatum: 16.10.2004

[Brooks 1995] Frederick P. Brooks, Jr. The Mythical Man Month: Essays on Software Engineering. Addison-Wesley, Reading, Ma., anniversary edition (1995)

[Budinsky+ 2003] Budinsky, F. J.; Finnie, M. A.; Vlissides, J. M; Yu, P. S.: Automatic code generation from design patterns. http://www.research.ibm.com/journal/sj/352/budinsky.html. Abrufdatum: 01.11.2003

[Bünter 1992] Bünter, Toni: Eine Architektur für ein Software-Wartungssystem. Universität Zürich, Inaugural-Dissertation, Zürich (1992)

[Buschmann+ 1998] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal, Michael: Pattern-orientierte Software-Architektur – Ein Pattern-System. Addison-Wesley (1998)

[Coram+ 2003] Coram, Todd; Lee, Jim: Experiences - A Pattern Language for User Interface Design. http://www.maplefish.com/todd/papers/experiences/Experiences.html#Modeless%20Feedback%20Area. Abrufdatum: 26.11.2003

[Dols 1978] Dols, José A.; Alexander, Christopher: Moderne Architektur – Fundamente, Funktionen, Formen. Rowohlt Taschenbuch Verlag GmbH, Reinbeck bei Hamburg (1978)

[Duden 1993] Engesser, Hermann (Hrsg.): Duden Informatik. Dudenverlag, Mannheim, 1993.

[Eden+ 2004] Eden, Amnon; Hirshfeld, Yoram; Yehudai, Amiram: LePUS: A Declarative Pattern Specification Language. http://citeseer.ist.psu.edu/112816.html. Abrufdatum: 01.06.2004

[Fayad 2007a] Fayad, M.E.: Skill and Experience: The Magical Wands. http://pattern.ijop.org/?p=16. Abrufdatum: 13.10.2007

[Fayad 2007b] Fayad, M.E.: Drawing a Fine Line between an Analysis Pattern and a Design Pattern. http://pattern.ijop.org/?p=23. Abrufdatum: 13.10.2007

[Fayad 2007c] Fayad, M.E.: Keep it Very Simple! http://pattern.ijop.org/?p=24. Abrufdatum: 13.10.2007

[Fayad 2007d] Fayad, M. E.: Untraceable Patterns-The Pitfalls of the Pattern Systems. http://pattern.ijop.org/?p=30. Abrufdatum: 13.10.2007

[Fayad 2007e] Fayad, M. E.: No Guidelines for Extracting Patterns. http://pattern.ijop.org/?p=31. Abrufdatum: 13.10.2007

[Fayad 2007f] Fayad, M. E.: Chasing an Elusive Vocabulary: Importance of Vocabulary in Pattern Development. http://pattern.ijop.org/?p=32. Abrufdatum: 13.10.2007

[Fayad 2007g] Fayad, M. E.: Outdated and Jaded Patterns or Fresh and Everlasting Patterns: Choose Your Pick! http://pattern.ijop.org/?p=33. Abrufdatum: 13.10.2007

[Fayad 2007h] Fayad, M. E.: Modeling Problems are Nagging Obstacles to Creating Meaningful Patterns. http://pattern.ijop.org/?p=34. Abrufdatum: 13.10.2007

[Fayad 2007i] Fayad, M. E.: Focus! Focus! Focus! Golden Words for Developing Meaningful Patterns. http://pattern.ijop.org/?p=25. Abrufdatum: 09.03.2008

[Fayad+ 2007a] Fayad, M.E.; Srikanth, G. K.: Same Problem, but Multiple Patterns! The Common Problem of Duplication. http://pattern.ijop.org/?p=19. Abrufdatum: 13.10.2007

[Fayad+ 2007b] Fayad, M.E.; Srikanth, G. K.: Choosing the Right Pattern – Real Challenges. http://pattern.ijop.org/?p=20. Abrufdatum: 13.10.2007

[Fayad+ 2007c] Fayad, M.E.; Srikanth, G. K.: A Brief Summary of Pattern Compositions: Understanding the Insider's Secrets. http://pattern.ijop.org/?p=27. Abrufdatum: 13.10.2007

[Fayad+ 2007d] Fayad, M.E.; Srikanth, G. K.: Lack of Patterns Connectivity across Development Phases. http://pattern.ijop.org/?p=28. Abrufdatum: 13.10.2007

[Fayad+ 2007e] Fayad, M. E.; Wu, Shasha: Reinventing the Wheel: An Undesirable and Dangerous Development. http://pattern.ijop.org/?p=35. Abrufdatum: 13.10.2007

[Flanagan+ 2002] Flanagan, C.; Leino, K.; Lillibridge, M.; Nelson, G.; Saxe, J.; Stata, R.: Extended Static Checking for Java**.** Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)

[Fowler 1998] Fowler, Martin: Analysemuster. Wiederverwendbare Objektmodelle. Addison-Wesley, München (1998)

[Fowler+ 2000] Fowler, Martin; Scott, Kendall: UML konzentriert. Addison-Wesley, München (2000)

[Fowler 2001] Fowler, Martin: Refactoring: improving the design of existing code; Addison-Wesley, Reading, 6. Auflage (2001)

[Fowler 2003a] Fowler, Martin: Patterns für Enterprise Application-Architekturen. mitp-Verlag, Bonn, 1. Auflage (2003)

[Fowler 2003b] Fowler, Martin: PatternsAreNothingNew. http://martinfowler.com/bliki/PatternsAreNothingNew.html. Abrufdatum: 26.11.2003

[Fraikin+ 2004] Fraikin, Frank; Hamburg, Matthias; Jungmayr, Stefan; Leonhardt, Thomas; Schönknecht, Andreas; Spillner, Andreas; Winter, Mario: Die trügerische Sicherheit des grünen Balkens. In: Objektspektrum, Ausgabe Januar/Februar 2004, S. 25-29, Sigs-Datacom (2004)

[Gamma+ 1995] Gamma, E.; Helm, R.; Johnson R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Co., Reading, MA (1995)

[Guéhéneuc+ 2001] Yann-Gel Guéhéneuc; Albin-Amiot, Hervé: Using Design Patterns and Constraints to Automate the Detection and Correction of Inter-class Design Defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems, pages 296-305. IEEE Computer Society Press (July 2001)

[Guéhéneuc 2003] Guéhéneuc, Yann-Gaël: PatternsBox. http://www.yann-gael.gueheneuc.net/Work/Research/Ptidej/Demo/. Abrufdatum: 03.11.2003

[Guéhéneuc 2004] Guéhéneuc, Yann-Gaël: PatternsBox – PADL Download (2004). http://www.yann-gael.gueheneuc.net/Work/Research/PatternsBox/Download/. Abrufdatum: 13.08.2004

[Gulden 2004] Gulden, Jens: BeautyJ (2004). http://beautyj.berlios.de/. Abrufdatum: 15.10.2004

[Haugeland 1997] Haugeland, John (ed.): Mind Design II – Philosophy, Psychology, Artificial Intelligence. MIT Press, Cambridge, London, 2. Auflage (1997)

[Hruschka+ 2002] Hruschka, Peter; Rupp, Chris: Agile Softwareentwicklung für Embedded Real-Time Systems mit der UML. Carl Hanser Verlag, München, Wien (2002)

[JCP175 2004] Java Community Process 175: A Metadata Facility for the Java™ Programming Language. http://www.jcp.org/en/jsr/detail?id=175. Abrufdatum: 16.10.2004

[JSR260 2004] Java Specification Request 260: Javadoc™ Tag Technology Update. Abrufdatum: 10.10.2007

[JSR305 2006] Java Specification Request 305: Annotations for Software Defect Detection http://www.jcp.org/en/jsr/detail?id=175. Abrufdatum: 10.10.2007

[Kazman+ 2002] Kazman, Rick; O'Brian Liam; Verhoef, Chris: Architecture Reconstruction Guidelines, 2nd Edition. Technical Report CMU/SEI-2002-TR-034. Carnegie Mellon Software Engineering Institute (2002)

[Kurzweil 1999] Kurzweil, Ray: The Age of Spiritual Machines. Penguin Books, New York et al. (1999)

[Kopi 2004] The Kopi Project. http://www.dms.at/kopi/. Abrufdatum: 04.02.2004

[Leavens+ 2003] Leavens, G. T.; Cheon, Y.: Design by Contract with JML. http://www.jmlspecs.org. Abrufdatum: 14.11.2003

[MacDonald+ 2002] MacDonald, S.; Szafron, D.; Schaeffer, J.; Anvik, J.; Bromling, S.; Tan, K.: Generative Design Patterns, 17th IEEE International Conference on Automated Software Engineering (ASE) September 2002, Edinburgh, UK, pp. 23-34 (2002)

[Mainzer 2003] Mainzer, Klaus: KI – Künstliche Intelligenz. Grundlagen intelligenter Systeme. Wissenschaftliche Buchgesellschaft, Darmstadt (2003)

[Mayr-Kern+ 2002] Mayr-Kern, Irmgard Anna: Erweiterung einer Programmierumgebung zur Unterstützung entwurfsmusterbasierter Softwareentwicklung. Diplomarbeit, Johannes Kepler Universität Linz (2002)

[Meffert 2004a] Meffert, Klaus: Design Patterns – Boondoggle or State of the Art. https://weblogs.sdn.sap.com/pub/wlg/606. Abrufdatum: 13.09.2004

[Meffert 2004b] Meffert, Klaus: Forendiskussion in der SAP Community. http://www50.sap.com/community/int/forums/ShowPost.aspx?PostID=165831. Abrufdatum: 13.09.2004

[Meffert 2006a] Meffert, Klaus: 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)

[Meffert 2006b] Meffert, Klaus: JUnit Profi - Tipps – Software erfolgreich testen. entwickler press (2006)

[Meffert+ 2006c] Meffert, Klaus; Philippow, Ilka: Supporting Program Comprehension for Refactoring-Operations with Annotations. In: Fujita, H., Mejri, M. (eds.): New Trends in Software Methodologies, Tools and Techniques - Proceedings of the fifth SoMeT_06, Vol. 147, pp. 48-67 (2006)

[Meffert 2007a] Meffert, Klaus: Supporting the Selection of Design Patterns by Formal Definition and Considering Semantics Proceedings of the 11th European Conference on Pattern of Programs, S. 655-667, UVK Universitätsverlag Konstanz GmbH (2007)

[Meffert+ 2007b] Meffert, Klaus; Philippow, Ilka: Towards a new code-based software development concept enabling code patterns. 2nd International Conference on Software and Data Technologies, ICSOFT 2007, Spain (2007)

[Meffert+ 2008a] Meffert, Klaus; Philippow, Ilka: Configuration Provider: A Pattern for Configuring Threaded Applications. Proceedings of the 12th European Conference on Pattern of Programs, S. 479-488, UVK Universitätsverlag Konstanz GmbH (2008)

[Meffert+ 2008b] Meffert, Klaus et al.: JGAP – Java Genetic Algorithms and Genetic Programming Package. http://jgap.sf.net. Abrufdatum: 16.09.2008

[Motelet 2004] Motelet, Olivier: A Contextual Help System for Assisting Object-Oriented Software Designers in using Design Patterns; EMOOSE Master Thesis (2000)

[Philippow+ 2004] Philippow, Ilka; Streitferdt, Detlef; Riebisch, Matthias; Naumann, Sebastian: An approach for reverse engineering of design patterns. Software and Systems Modeling, pp. 55-70 (2004)

[NetWeaver 2003] SAP AG: SAP NetWeaver. http://service.sap.com/netweaver. Abrufdatum: 20.11.2003

[Ó Cinnéide+ 1999a] Ó Cinnéide, Mel; Nixon, Paddy: A Methodology for the Automated Introduction of Design Patterns; IEEE International Conference on Software Maintenance. Oxford, England. http://www.computer.org/proceedings/icsm/0016/00160463abs.htm. Abrufdatum: 17.10.2004

[Ó Cinnéide+ 1999b] Ó Cinnéide, Mel; Nixon, Paddy: Automated Application of Design Patterns to Legacy Code. http://www.cs.ucd.ie/staff/meloc/home/papers/ECOOP99.pdf. Abrufdatum: 17.10.2004

[objectiF 2004] microTOOL GmbH : objectiF Homepage. http://www.microtool.de/objectif/de/index.asp. Abrufdatum: 16.10.2004

[PBE 2004a] Delta Software Technology GmbH: Pattern By Example. April 2002. www.d-s-t-g.com/neu/media/pdf/facts_e/DLT21551.pdf. Abrufdatum: 27.06.2004

[PBE 2004b] Delta Software Technology GmbH: PBE Demo. http://www.d-s-t-g.com/neu/pages/pagesger/et/common/techn_pbe_pbe6_frmset.htm. Abrufdatum: 17.10.2004

[PMD 2003] InfoEther: PMD. http://pmd.sourceforge.net/. Abrufdatum: 25.11.2003.

[Pree 1994] Pree, Wolfgang: Meta Patterns – A Means for Capturing the Essentials of Reusable Object-Oriented Design. In M. Tokoro und R. Pareschi (Hrsg.), Proceedings ECOOP ,94, LNCS 821, pp. 150-162, Bologna, Italy, July 1994. Springer Verlag (1994)

[Pree 1995] Pree, Wolfgang: Design Patterns for Object-Oriented Software Development. Addison-Wesley Publishing Co., Reading, MA (1995)

[Pree 2004] Pree, Wolfgang: State-of-the-art Design Pattern Approaches - An Overview. http://citeseer.ist.psu.edu/440891.html. Abrufdatum: 17.10.2004

[Robillard 2003] Robillard, Martin P.: Representing Concerns in Source Code. Ph.D. Thesis. Department of Computer Science, University of British Columbia (2003)

[Sailer 2002 Sailer, Hartmut: Was sind Design Patterns? http://www.unilog-integrata.de/training/news/news2002_4_8.html. Abrufdatum: 20.01.2004

[Sang-Uk 2002] Sang-Uk, Jeon: An Approach to Automatically Identifying Design Structure for Applying Design Pattern, 2002.

http://salmosa.kaist.ac.kr/~sujeon/paper/Thesis_Jeon.pdf. Abrufdatum: 15.11.2003

[Sang-Uk 2003a] Sang-Uk, Jeon: Pattern-based Automatic Design Evolution of Java Program. http://salmosa.kaist.ac.kr/~sujeon/ppt/KSEJW2002.ppt. Abrufdatum: 15.11.2003

[Sang-Uk+ 2003b] Sang-Uk, Jeon; Joon-Sang Lee; Doo-Hwan Bae: An Automated Refactoring Approach To Design Pattern-Based Program Transformations in Java Programs. http://salmosa.kaist.ac.kr/~sujeon/paper/Jeon_DesignPattern.pdf. Abrufdatum: 15.11.2003

[Seguin+ 2004] Seguin, Chris; Atkinson, Mike: JRefactory. http://jrefactory.sourceforge.net/. Abrufdatum: 19.09.2004

[Spec 2007] SAnToS laboratory: Specification Patterns. http://patterns.projects.cis.ksu.edu. Abrufdatum: 01.12.2007

[Sun 2004] Sun: How to Write Doc Comments for the Javadoc™ Tool. http://java.sun.com/j2se/javadoc/writingdoccomments/index.html. Abrufdatum: 19.09.2004

[Taibi+ 2003] Taibi, Toufik; Chek Ling Ngo, David: Formal Specification of Design Patterns – A Balanced Approach. In: Journal of Object Technology, vol. 2, no. 4, July-August 2003, pp. 127-140. http://www.jot.fm/issues/issue_2003_07/article4. Abrufdatum: 03.02.2004

[Tan+ 2003] Tan, K; Szafron, D; Schaeffer, J; Anvik, J; Mac-Donald, S: Using generative design patterns to generate parallel code for a distributed memory environment. In PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 203–215, New York, NY, USA, 2003. ACM Press (2003)

[Tatsubori 2003] Tatsubori, Michiaki: OpenJava. http://www.csg.is.titech.ac.jp/openjava/. Abrufdatum: 01.12.2003

[Topley 2004] Topley, John: Learning to K.I.S.S. http://www.johntopley.com/archive/2004/01/06/index.html. Abrufdatum: 18.01.2004

[Vlissides 1999] Vlissides, John: Designing with Patterns. http://www.research.ibm.com/designpatterns/pubs/dwp-tutorial.pdf. Abrufdatum: 05.12.2003

[Wik 2006] Wikipedia: Java Modeling Language. http://en.wikipedia.org/wiki/Java_Modeling_Language. Abrufdatum: 13.06.2006

[Wöckl 2002] Wöckl, Andreas: Erweiterung einer Entwicklungsumgebung zur flexiblen Definition, Verwaltung und Visualisierung von Entwurfsmustern und deren Dokumentation (2002). http://www.swe.uni-linz.ac.at/people/sametinger/research/pse/pse-base.dipl.pdf. Abrufdatum: 29.06.2004

[XDoclet 2004] XDoclet Team: XDoclet. http://xdoclet.sourceforge.net/. Abrufdatum: 16.10.2004

[Zelix 2003] Zelix Pty Ltd: Zelix KlassMaster - Java Name Obfuscation. http://www.zelix.com/klassmaster/featuresNameObfuscation.html. Abrufdatum: 23.11.2003

[Zimmer 1997] Zimmer, Walter: Frameworks und Entwurfsmuster. Dissertation Universität Karlsruhe (Technische Hochschule), Informatik 1997. Shaker Verlag, Aachen (1997)

Thesen

1. Quelltext-zentrierte Entwicklung spielt eine bedeutende Rolle in der heutigen Software-Entwicklung.
2. Der Entwickler wird bei der Arbeit mit Entwurfsmustern nicht ausreichend unterstützt, um Muster mit geringem Aufwand sinnvoll auswählen und korrekt anwenden zu können.
3. Die meisten Entwickler wenden Entwurfsmuster nicht in signifikantem Maße an.
4. Werkzeuggestützte Quelltext-Transformationen werden aktuell nicht in dem Maße unterstützt, wie dies wünschenswert wäre.
5. Ohne Hinzuziehen eines Menschen kann nicht garantiert werden, dass beliebige Quelltext-Transformationen ein korrektes Ergebnis liefern.
6. Quelltext-Transformationen können unter Zuhilfenahme von explizit im Quelltext angebrachten Informationen erstens vereinfacht und zweitens leistungsfähiger gestaltet werden.
7. Die Bedeutung eines Programmteils (Deklaration, Anweisung oder Menge davon) innerhalb eines Programms kann im Allgemeinen nicht automatisiert ermittelt werden.
8. Die Programmiersprache Java ist unter pragmatischen Gesichtspunkten als beispielhafte Implementierungssprache geeignet. Java ist sehr populär und vergleichbar mit der ebenfalls sehr weit verbreiteten Sprache C#. Java ist ein typischer Vertreter moderner objektorientierter Sprachen und besitzt keine nennenswerten Eigenheiten, die Betrachtungen, welche für Java gültig sind, für andere vergleichbare Sprachen wie C# ungültig erscheinen ließe.
9. Die Begriffe Entwurfsmuster und Architekturmuster können nicht scharf von einander abgegrenzt werden.

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Weitere Personen waren an der inhaltlich-materiellen Erstellung der vorliegenden Arbeit nicht beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch angesehen wird und den erfolglosen Abbruch des Promotionsverfahrens zur Folge hat.

Goldbach, den 11. November 2008 Klaus Meffert