Integration von Entwurfsmustern durch Refactoring-Transaktionen

Alexander Bruder
anib@uni-paderborn.de

Letzte Änderung: 19. Februar 2003

Inhaltsverzeichnis

1 Einführung

Diese Seminararbeit beschäftigt sich mit der Integration von Entwurfsmustern (Design Patterns) in Softwaresysteme mittels Refactoring. Es werden hierbei mehrere existierende Ansätze genannt und auch diskutiert. Dabei wird besonderes Augenmerk auf eine Methode von Mel Ó Cinnéide und Paddy Nixon [CN98] [CN99a] [CN99b] [Cin00] [Cin01] gerichtet, die der vielversprechenste Ansatz ist und genauer anhand eines Beispiels in Abschnitt 2 untersucht wird. Daran anschließend wird ein Entwurfsmuster präsentiert, dass nicht automatisch transformierbar ist. Der vorletzte Abschnitt beschäftigt sich mit einem Programm, das in eingeschränktem Maße Integration von ausgewählten Entwurfsmusteren in bestehenden Software erlaubt. Beendet wird diese Arbeit mit einer Schlussfolgerung.

1.1 Motivation

Warum ist man bestrebt, automatische Codetransformationen durchzuführen und dadurch u.a. Entwurfsmuster zu integrieren?

Jeder der schon einmal eine Software nach dem Wasserfallmodell entwickelt hat weiß, dass das Ergebnis jeder einzelnen Phase dem nächsten Schritt als Grundlage dient. Fehler, die früh gemacht werden oder Änderungen, die erst später einzuarbeiten sind, stellen den Entwicklungsprozess vor Probleme. Man wird feststellen, dass es praktisch unmöglich ist, gleich auf Anhieb die Software richtig zu entwerfen und so problemlos den gesamten Entwicklungszyklus zu durchlaufen.

Es wäre sicherlich einfacher, klein anzufangen und der Software fürs Erste einen überschaubaren Funktionsumfang zu spendieren und dann nach und nach weitere Funktionalität zu implementieren, an die man am Anfang vielleicht noch nicht gedacht hat. Möchte man seiner Anwendung etwas hinzufügen, stellt man schnell fest, dass der aktuelle Code entweder gar nicht die Möglichkeit bietet die Idee einzubauen oder nicht ausreichend flexibel ist, um eine reibungslose Integration zu garantieren.

Verfügt man nun über die Fähigkeit, automatische Codetransformationen anwenden zu können, wird man in die Lage versetzt, schnell grundlegende Änderungen auf der Entwurfsebene vorzunehmen. Dies spart Zeit und lässt außerdem eine iterative Weiterentwicklung der Software zu.

Automatische Codetransformationen auf Entwurfsebene helfen aber nicht nur beim Erstellen von Software sondern auch bei der Wartung und Anpassung. Schon seit den Siebzigern geistert der Begriff der *Softwarekrise* umher und will nicht verstummen. Softwarekrise bedeutet dabei die steigenden Kosten für den Lebenszyklus der Software aus der Sicht des Herstellers. Diese Kostenentwicklung verläuft für Software und Hardware genau entgegengesetzt.

Schlüsselt man die Kosten nach Kostenträgern auf, wie in Abbildung 1 geschehen, so stellt man fest, dass in etwa zweidrittel der Kosten auf Wartung und Erweiterung entfallen. Diese Kostenstelle kann weiter unterteilt werden. Dabei schlagen die Fehlerkorrekturen sowie die Anpassungen mit je 20% zu Buche, die Erweiterung der Software macht 60% aus.

Analyse und Entwurf	8%
Programmierung	10%
Test	10%
Integration	5%
Wartung und Erweiterung	67%

Abbildung 1: Kostenträgeraufschlüsselung nach [zb02]

Unter Zuhilfenahme automatischer Codetransformationen und der Integration von Entwurfsmustern wäre man in der Lage, Wartung und Erweiterung schneller, sicherer und damit auch kostengünstiger durchzuführen.

Nicht zuletzt laden diese neuen Möglichkeiten den Softwarearchitekten auch zum Experimentieren ein, was möglicherweise innovationsfördernd sein kann.

1.2 Entwurfsmuster (Design Patterns)

Entwurfsmuster sind eine der herausragendsten Entwicklungen in der Softwaretechnik des letzten Jahrzehnts. Sie versuchen erprobtes Wissen, dass über die Jahre entstanden ist, zu katalogisieren und wiederverwendbar zu machen. Dies impliziert, dass Entwurfsmuster nicht erfunden, sondern entdeckt werden. So ist es möglich, immer wieder auftretende Probleme von Bekanntem abzuleiten und anzupassen.

Es gibt mittlerweile eine Vielzahl von Entwurfsmustern. Im Weiteren sind die Entwurfsmuster auf den Katalog von Gamma [EGV95] mit seinen 23 Entwurfsmustern beschränkt, da dieser als Standard anerkannt ist.

Die 23 Muster sind in drei Gruppen unterteilt. Gruppe 1 enthält die erzeugenden Entwurfsmuster (*Creational Patterns*):

> Abstract Factory, Builder, Factory Method, Prototype und Singleton

Gruppe 2 besteht aus den strukturellen Entwurfsmustern (*Structural Patterns*):

> Adapter, Bridge, Composite, Decorator, Facade, Flyweight und Proxy

Die letzte Gruppe setzt sich aus Verhaltensmustern zusammen (*Behavioral Patterns*):

> Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method und Visitor

Gamma benutzt in seinem Katalog ein Schema zur Beschreibung eines Entwurfsmusters, dass folgende Informationen enthält:

Intention Welches Problem kann das Entwurfsmuster lösen.

Motivation Beschreibt ein konkretes Szenario, in dem dieses Muster angewand werden kann und zeigt auf, warum es sinnvoll ist, es in diesem Kontext zu benutzen.

Anwendungsbereich Auflistung von Entwurfssituationen, auf die das Muster angewant werden kann.

Struktur UML-Notation des Entwurfsmusters.

Teilnehmer Beschreibung der Klassen/Objekte, die in dem Muster auftreten sowie ihre Aufgaben und Pflichten.

Konsequenzen Zusammenstellung der Vor- und Nachteile (bezogen auf den Entwurf) bei Integration in die Anwendung

Implementierung Skizzierung einer sinnvollen Implementierung.

Beispielprogrammcode Ein Codebeispiel

Bekannte Anwendungfälle Existierende Software, die genau dieses Muster an einer bestimmten Stelle benutzt.

Verwandte Muster Verknüpfung zu Entwurfsmustern, die Artverwandt sind oder aber mit dem konkreten Muster gut zusammenspielen.

Aus diesem allgemein gehaltenen Bauplan kann dann die entsprechende Information extrahiert werden, die für den konkreten Anwendungsfall benötigt wird.

Diese Seminararbeit bedient sich zweier Entwurfsmuster, anhand derer im Fall von *Abstract Method* die erfolgreiche Transformation am Beispiel veranschaulicht wird und im Fall von *Facade* ein Fehlschlag. Nachfolgend wird das erst genannte Muster erklärt, um im zweiten Abschnitt der Seminararbeit die Transformationen besser verstehen zu können. Das Entwurfsmuster *Facade* hingegen wird im Abschnitt 3 erst erläutert.

1.2.1 Factory Method

Factory Method erzeugt über einen Methodenaufruf in einer Unterklasse, die durch eine Schnittstelle oder abstrakte Klasse definiert wird, zur Laufzeit ein Objekt. Dieses Objekt (Produkt) kann widerum eine Instanz einer Unterklasse sein, um unterschiedliche Produkte an unterschiedliche Erzeuger zu knüpfen.

Wie Abbildung 2 deutlich zeigt, besteht dieses Entwurfsmuster aus einem Produzenten (*ConcreteCreator*) und einem Produkt (*ConcreteProduct)*, die jeweils

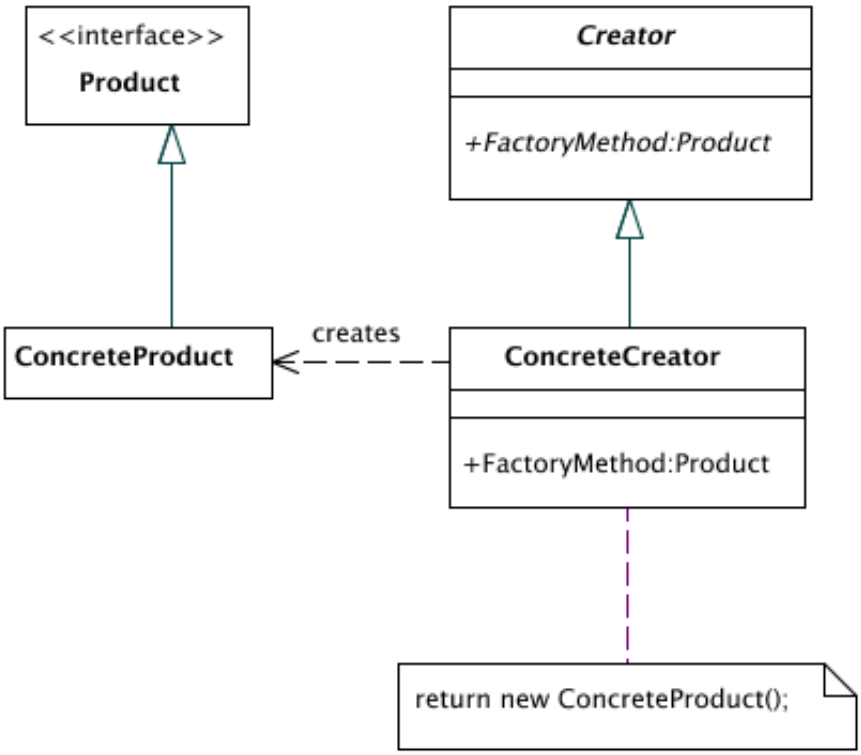

Abbildung 2: Klassendiagramm für *Factory Method*

aus Oberklassen erben. Durch den Aufruf von *FactoryMethod()* erzeugt *ConcreteCreator* zur Laufzeit das entsprechende Produkt.

Indem man den Unterklassen des Produzenten die Instanzierung der jeweilligen Produkte überträgt, entkoppelt man elegant die in der Abbildung angedeutete Abhängigkeit. Somit ist es auch möglich, unterschiedliche Produkte abhängig von der Unterklasse des Produzenten zu erzeugen.

Ein klassisches Beispiel für die Anwendung von *Factory Method* wäre eine graphische Anwendung, die mehrere unterschiedliche Arten an Dokumenten enhalten kann, z. B. ein Dokument zum Schreiben, eines zum Zeichnen usw. Zu jedem Dokumententyp existiert dann ein Produzent, der die entsprechende Instanz des Dokumentes erzeugt und zurückliefert.

1.3 Refactoring

Im Verlauf dieses Seminars wurde schon eine Einführung in Refactoring gegeben. Um keine Redundanz zu schaffen, wird an dieser Stelle auf eine Einführung und die damit verbundene Erläuterung der Grundlagen zu Gunsten einer Kon-

zentration auf die besonderen Anforderungen von Refactoring, bezogen auf die Integration von Entwurfsmustern, verzichtet.

Wer dennoch Interesse hat und Informationen aus erster Hand wünscht, dem seien folgende beiden Quellen empfohlen: [Opd92] [MFR00]

Wie im vorhergehenden Abschnitt erläutert, sind Entwurfsmuster Lösungen für Probleme in einem bestimmten Kontext. Sie werden auf Entwurfsebene benutzt und haben kein Pendant in der Zielsprache. D. h., sie befinden sich in einer höheren Abstraktionsebene als die Programmiersprache. Dort wird das Entwurfsmuster durch Sprachkonstrukte wie Klassen und Methoden zusammengesetzt, die wiederum die Logik kapseln.

Das hat zur Folge, dass automatische Integration von Entwurfsmustern in der Regel der Fälle nicht durch ein einziges Refactoring erzeugt werden kann. Mit Refactoring ist hier ein atomares Refactoring (*primitive*) gemeint, dass sich nicht weiter aufspalten lässt und korrekt arbeitet, also verhaltenskonservierend transformiert. Dies ist wichtig, da Refactoring umstruktieren und nicht das Verhalten des Systems ändern soll.

Um nun Entwurfsmuster integrieren zu können, müssen mehrere Refactorings hintereinander ausgeführt werden. Diese Refactorings sind mit Vor- und Nachbedingungen versehen, die eingehalten werden müssen, um das nächste Refactoring ausführen zu dürfen. Mel Ó Cinnéide benutzt dazu ein Beweissystem, dass unter [CN00] beschrieben wird. Es wird im Unterabschnitt 2.1 aufgegriffen. Außerdem existiert eine eigene Seminararbeit zu dem Thema *Komposition von Transformationen*.

Sequenzen von Operationen die an Bedingungen geknüpft sind, werden im Datenbankbereich als *Transaktion* bezeichnet. Dieser Begriff bietet sich auch hier an. Die folgende Definition ist eine angepasste Variante der Definition einer Datenbanktransaktion und wurde aus [AH00] entnommen.

1.3.1 Definition von *Refactoring-Transaktionen*

> Unter einer *Refactoring-Transaktion* wird eine Folge von Quellcodetransformationen verstanden, die bzgl. der Integritätsüberwachung als Einheit (atomar) angesehen werden. Daraus folgt, dass der Entwurf und der Quellcode nur vor und nach einer *Refactoring-Transaktion* in zulässigen Zuständen zu sein braucht. *Refactoring-Transaktionen* müssen die sogenannten *ACID-Eigenschaften* wie folgt erfüllen:

A *Atomicity (Atomarität)*: Eine *Refactoring-Transaktionen* wird entweder ganz oder gar nicht ausgeführt. *Refactoring-Transaktion* können keine Zwischenzustände nach einem Abbruch hinterlassen.

C *Consistency (Konsistenz)*: Nach einem erfolgreichen Ende einer *Refactoring-*

Transaktionen muß die Software in einem zulässigen Zustand sein, also das gleiche externe Verhalten für gleiche Eingabe zeigen, wie vor der *Refactoring-Transaktion*.

I *Isolation*: *Refactoring-Transaktionen* laufen im simulierten Einbenutzerbetrieb ab. Eventuell parallel ablaufende andere *Refactoring-Transaktionen* sind isoliert und können sich nicht gegenseitig beeinflussen.

D *Durability (Dauerhaftigkeit)*: Die Wirkung einer einmal erfolgreich beendeten *Refactoring-Transaktion* ist dauerhaft, sofern nicht der komplette Zustand vor der Transaktion gespeichert oder ein Protokoll mitgeschrieben wurde, da eine Transaktion rückwärts ausgeführt ein anderes Ergebnis liefert und nicht mehr mit dem vorherigen Zustand identisch ist.

1.4 Existierende Ansätze

Es existieren zur Zeit insgesamt drei Ansätze zur automatischen Codetransformation, die auch mit Entwurfsmustern umgehen können. Diese Ansätze unterscheiden sich hauptsächlich in zwei Punkten:

- Automatisierbarkeit
- Ausgangssituation

Automatisierbarkeit meint in diesem Zusammenhang, in wie weit der Transformationsprozess von Entscheidungen und Eingaben des Benutzers abhängig ist. D. h., kann eine einmal angestoßene Transformation ohne weiteres Einwirken in jedem Kontext zu dem beabsichtigten Ergebnis führen oder läuft die Transformation interaktiv ab, und wird der Benutzer fortlaufend um Entscheidungen gebeten.

1.4.1 Green Field Situation

Der trivialste Ansatz ist die so genannte *Green Field Situation*. Hierbei wird vorausgesetzt, dass die Komponenten, die in das Entwurfsmuster transformiert werden sollen, in keinem Zusammenhang zueinander stehen und somit auch keine Abhängigkeiten vorweisen. Damit spart man sich viel Programmanalyse. Eine solche Transformation kann fast immer voll automatisch durchgeführt werden.

Der Nachteil liegt aber auf der Hand. Möchte man existierenden Programmcode warten oder erweitern und besteht zwischen einzelnen, zu verändernden Komponenten eine Abhängigkeit, so kann diese Methode nicht angewandt werden, da die Abhängigkeiten nicht verhaltenskonservierend aufgelöst werden können.

Diese Transformationsmethode kann in Teilen auch über das Generator-Prinzip gelöst werden, in dem man bestimmte Situation mit den dazugehörigen Komponenten identifiziert und dann für den erkannten Fall einen Generator erschafft,

der als Eingabe die „losen“ Komponenten übergeben bekommt und daraufhin das Entwurfsmuster generiert.

Wegen der genannten Einschränkung ist dieses Modell für den breiten Einsatz ungeeignet und wird in dieser Seminararbeit nicht weiter behandelt.

1.4.2 Antipatterns

Das Gegenteil zur *Green Field Situation* ist der Ansatz über *Antipatterns*. Als *Antipatterns* kann man grob gesagt Entwürfe verstehen, die ein Entwurfsmuster nicht korrekt, schlecht, kompliziert oder an der falschen Stelle einsetzen.

Mit der *Antipattern*-Methode wird die Annahme verbunden, dass der Softwarearchitekt nicht vertraut mit Entwurfsmustern ist und stattdessen einen „unglücklichen“ Weg genommen hat, um ähnliche Funktionalität einzubauen. Um solche schlechten Lösungswege automatisch in bessere transformieren zu können, müsste jede schlechte Lösung dem Transformationswerkzeug bekannt sein. Dies zu implementieren, ist zur Zeit unmöglich, da man eine Umstrukturierung eines schlechten Entwurfs in einen guten Entwurf nicht generisch lösen kann.

Daher wird auch diesem Modell in dieser Seminararbeit nicht weiter Aufmerksamkeit geschenkt.

1.4.3 Precursor

Der *Precursor*-Ansatz von Mel Ó Cinnéide [Cin01] ist ein Kompromiss aus *Green Field Situation* und *Antipattern*. Er beschreibt eine Entwurfsstruktur, die Abhängigkeiten einzelner Komponenten erlaubt und dabei eine Art Zwischenprodukt oder Vorläufer auf dem Weg zu einem Entwurfsmuster ist, aber gleichzeitig nicht als *Antipattern*, also schlechter Entwurf, charakterisiert werden kann. Damit erreicht man, dass auch ein bestehendes Softwaresystem mit seinen Abhängigkeiten transformiert werden kann. Im Abschnitt 2.3 wird genauer auf diese Idee eingegangen und an einem Beispiel verdeutlicht.

2 Die Transformation (Refactoring-Transaktion)

Die Synthese Martin Fowlers und William F. Opdykes Definitionen von Refactoring lassen sich als

> *verhaltenskonservierende Programmumstrukturierung zur Verbesserung der internen Struktur*

zusammenfassen.

Von dieser Definition ausgehend, werden in diesem Abschnitt die einzelnen Schritte beschrieben, die für eine erfolgreiche Transformation notwendig sind. Zuvor wird dazu aber das nötige Grundwissen, sofern es im Zuge dieses Seminars noch nicht vermittelt worden ist, präsentiert. Im Anschluß daran werden anhand eines fortlaufenden Beispiels die eben angekündigten Teilschritte erläutert, so dass am Ende eine vollständige, automatisierbare Transformation steht. Als Beispiel wird das unter 1.2.1 beschriebene Entwurfsmuster *Factory Method* verwendet, welches aus der Dissertation von Mel Ó Cinnéide [Cin01] stammt.

2.1 Grundlagen

Die Grundlagen für die Art und Weise, wie man mehrere Transformationen korrekt hintereinanderschaltet, wird in der Seminararbeit *Komposition von Transformationen* ausführlich beschrieben. Im Folgenden wird lediglich das benötigte Wissen in Teilen kurz rekapituliert.

Jede automatische Codetransformation bzw. Refactoring-Transaktion nach dem Modell von Mel Ó Cinnéide besitzt eine Hierarchie, die in Abbildung 3 zu sehen ist. Dabei bilden Hilfs- und Analysefunktionen zusammen mit den atomaren Refactorings das Fundament. Drauf aufbauend operieren die Minitransformationen, aus denen eine Entwurfsmustertransformation besteht. Die drei obersten Schichten können alle als Refactoring-Transaktionen verstanden werden, wobei ein atomares Refactoring ein Speziallfall ist, in dem die Sequenz einelementig ist.

Damit alle Refactoring-Transaktionen in jeder Situation semantische Korrektheit garantieren, müssen Vor- und Nachbedingungen eingehalten werden. Die folgende Abbildung 4 deutet an, an welchen Stellen in der Sequenz Bedingungen erfüllt werden müssen.

Das Beispiel geht von einer Transaktion mit zwei Refactorings (R_1 und R_3) sowie einer Minitransfomation (M_2) aus. Die Minitransformation ist wiederum eine Refactoring-Transaktion. Die Bedingungen können vereinfacht werden, indem man nur die Vorbedingungen verwendet, da, wie später erläutert wird, die Nachbedingung das Ergbnis des Refactorings ist und somit als Eingabe für den nächsten Schritt dient.

2.1.1 Atomare/Primitive Refactorings (*primitive refactorings*)

Die im Folgenden häufiger benutzten atomaren Refactorings sind die Grundtransformationen für Refactoring-Transaktionen. Sie bilden damit also das Fundament. Diese Sorte der Refactorings besitzen immer eine Vorbedingung, die in Prädikatenlogik der ersten Stufe beschrieben wird und eine Nachbedingung, die

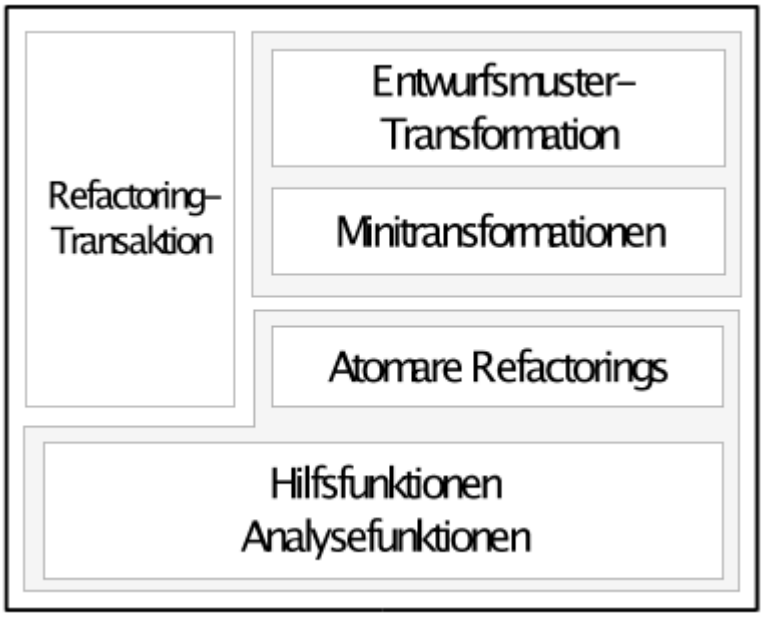

Abbildung 3: Aufbau einer Transformation

das Ergebnis der Transformation definiert. Durch die beiden Bedingungsarten versucht man zu garantieren, dass das Refactoring verhaltenskonservierend arbeitet. Ein vollständig formaler Beweis für die semmantische Korrektheit der Transformation existiert allerdings nicht. Es wird ein nicht formaler Ansatz verfolgt, der lediglich auf Argumentation basiert.

Beispiel: Hinzufügen einer *getter/accessor*-Methode

> void **addGetMethod**(Class *concrete*, String *fieldName*)
> Fügt eine Methode zur Klasse *concrete* hinzu, die einen geschützten Zugriff auf eine private Variable mit dem Namen *fieldName* erlaubt.

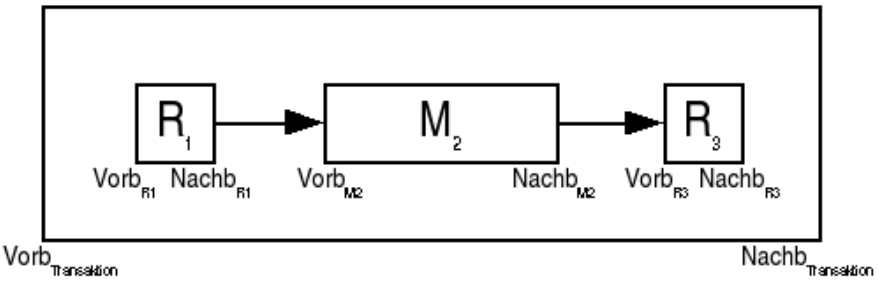

Abbildung 4: Vor und Nachbedingungen

Vorbedingung
Die Klasse *concrete* muß existieren und eine Variable mit dem Namen *fieldName* besitzen.

isClass(*concrete)* ∧ classOf(*fieldName*) = *concrete*

Die Klasse darf keine Methode mit dem Namen „get“ + *fieldName* haben.

∀ m:Method, declares(*concrete*, m) ∘ nameOf(m) ≠ „get“+ *fieldname*

Nachbedingung
concrete verfügt nun über eine Methode mit dem Name „get“ + *fieldName*.

∃ m:Method so dass

classOf’ = classOf[m/*concrete*]
nameOf = nameOf[m/„get“ + *fieldName*]

Rückgabe

returnsObject’=returnsObject[m/*fieldName*]

Korrektheit
Da keine Methode mit dem Namen der Hinzuzufügenden existiert, kann es keine Namenskonflikte geben und auch keine Stelle, an der eine solche Methode hätte aufgerufen werden können.

2.1.2 Analysefunktionen (*analysis functions*)

Analysefunktionen haben zwei Aufgaben. Einerseits dienen sie als Funktionen und Prädikate der Prädikatenlogik erster Stufe für die Vorbedigungen. Andererseits sind sie im Modell direkt als Operationen implementiert, mit denen man, in unserem Fall Java, auf dem Quellcode gearbeitet wird. D. h., die zweite Aufgabe, also die Implementierung, ist somit die Interpretation der ersten Aufgabe der Analysefunktionen. In der Anwendung erhält man durch Analysefunktionen Informationen über den Quellcode, genauer gesagt über dessen Beschaffenheit bzw. Struktureigenschaften.

Beispiel 1: Überprüfung, ob es sich um eine Klasse handelt

Boolean **isClass**(Class *c*)
Liefert *wahr* zurück, wenn es sich bei *c* um eine Klasse handelt, ansonsten *falsch*. Wird jedoch ein String *a* übergeben, wird damit die Existenz einer Klasse *a* im Kontext abgefragt.

Beispiel 2: Überprüfung auf Schnittstellenimplementierung

Boolean **implementsInterface**(Class/Interface *e*, Interface *i*)
Liefter *wahr* zurück, wenn die Klasse oder die Schnittstelle *e* die Schnittstelle *i* implementiert.

2.1.3 Hilfsfunktionen (*helper functions*)

Hilfsfunktionen sind komplexer als Analysefunktion und extrahieren mehr Informationen. Der größte Unterschied besteht aber darin, dass Hilfsfunktionen über Vor- und Nachbedingungen verfügen, somit „richtige“ Funktionen sind und keine Seiteneffekte auslösen können. Ein Teil der Nachbedingung besteht darin, den Ergebniswert zurückzuliefern.

Beispiel: Abstrahiere Schnittstelle von Klasse

Interface **abstractClass** (Class *c*, String *newName*)
Erzeugt eine Schnittstelle mit dem Namen *newName*, die alle öffentlichen (public) Methoden der Klasse *c* enthält.

Vorbedingung
Die Klasse *c* muß existieren.

isClass(c)

Nachbedingung
Die zurückgelieferte Schnittstelle enthält alle öffentlichen Methoden der Klasse *c*.

isInterface' = isInterface[*inf*/true]
equalInterface' = equalInterface[(*c*,*inf*)/true]

Der Name der erzeugten Schnittstelle muß *newName* sein.

nameOf' = nameOf[*inf*/*newName*]

2.1.4 Annahmen und Beschränkungen des beschriebenen Modells

Um sinnvolle und korrekte Ergebnisse nach dem Modell von Mel Ó Cinnéide zu erzielen, müssen gewisse Einschränkungen gemacht werden.

1. Es wird von Java-Programmen ausgegangen, da die Sprache einfacher ist als z. B. C++ und breit eingesetzt wird.
2. Der zu transformierende Quellcode muß am Anfang korrekt zu kompilieren sein.

   Beispiel: Wendet man z. B. das Refactoring *addMethod* im Zuge einer Refactoring-Transaktion auf ein nicht kompilierbares Programm an und die Refactoring-Transaktion fügt ausgerechnet die Methode hinzu, die zur Kompilierung benötigt wird, so ändert sich das Verhalten des Programms. In diesem Fall von *nicht kompilierbar* in *kompilierbar*. Auch eine solche Änderung ist nicht verhaltenskonservierend.

3. Programme, die folgenden Code enthalten, können nicht transformiert werden:

   ```
   obj.getClass().getMethod("foo", null).invoke(obj);
   ```

   In diesem Fall ist es nicht eindeutig, ob die Methode *foo* z. B. umbenannt werden muß.

4. Es wird angenommen, dass Objekte ausschließlich mit **new** erzeugt werden können. Klonen von Objekten scheidet damit aus.

5. Private Klassen werden der Einfachheit halber nicht zugelassen.

6. *Packages* werden ignoriert. Es wird angenommen, dass eine Klasse oder Schnittstelle sicher über den Namen identifiziert werden kann.

7. In einer Schnittstellendefinition wird eine Methode durch den Namen, den Rückgabewert und die Parameter beschrieben. Etwaige Ausnahmen (Exceptions) werden der Einfachheit halber ignoriert.

2.2 Entwicklung einer Transformation für ein Entwurfsmuster

Die Entwicklung einer Transformation für ein Entwurfsmuster geht in vier bis fünf Schritten von statten, wie Abbildung 5 zeigt.

1. Auswahl des Design Patterns, dass transformiert werden soll.

2. Festlegen eines *Precursor* als Startpunkt, der eine Verknüpfung zwischen der Struktur des Entwurfsmuster und der des aktuellen Systems herstellt.

3. Aufschlüsselung des Weges vom *Precusor* zum Entwurfsmuster in *Minipatterns*. *Minipatterns* werden unter 2.4 beschrieben.

4. Sofern noch keine entsprechenden *Minitransformationen* (siehe 2.4) existieren, müssen sie erst noch definiert werden.

5. Wenn alle benötigten *Minitransformationen* vorhanden sind, werden sie zu einer Refactoring-Transaktion gebündelt.

Im weiteren Verlauf dieses Abschnittes werden die eben beschriebenen Schritte anhand des Entwurfsmusters *Factory Method* wie angekündigt erklärt

2.3 Ausgangsbedingung / Precursor

Der *Precursor* ist das zentrale Element in dem Verfahren zur automatischen Codetransformation nach Mel Ó Cinnéide. Wie schon weiter oben angedeutet,

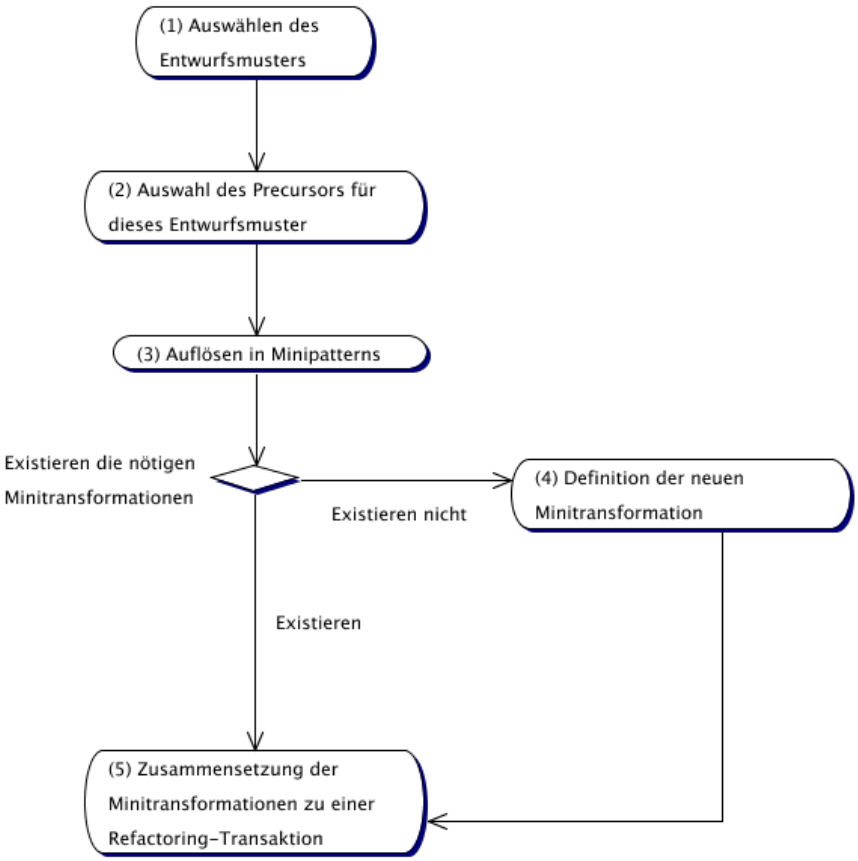

Abbildung 5: Aktivitätsdiagramm zum Ablauf der Entwicklung einer Transformation für ein Entwurfsmuster

ist der *Precursor* eine Entwurfsstruktur, die die Intention, welche hinter einem Entwurfsmuster steckt, in vereinfachter Weise widerspiegelt. Mit diesem Ansatz ist es möglich, einen Mittelweg zwischen *Green Field Situation* (siehe 1.4.1) und *Antipattern* (siehe 1.4.2) zu schaffen.

Diesen Ansatz zu finden, ist nicht immer einfach. Man muß eine ausgewogene Mischung zwischen Vollständigkeit und Einfachheit der Struktur treffen. Denn ist die Struktur nicht vollständig genug, werden Situationen im Kontext gefunden, die nicht im Entferntesten den Zusammenhang repräsentieren, auf den das Entwurfsmuster abzielt. Auf der anderen Seite muß der *Precursor* aber auch ausreichend einfach sein, so dass die Programmanalyse den *Precursor* im Quellcode antreffen kann und nicht durch zu viele benötigte Bedingungen nur in Spezialfällen einen Treffer landet.

Beispiel:

Um den *Precursor* für das Entwurfsmuster *Factory Method* zu finden, ist es

sinnvoll das Klassendiagramm (siehe Abbildung 2) dieses Musters zu vereinfachen und auf einen Stand zu bringen, der es erlaubt, nur jeweils einen Hersteller und ein Produkt zu benutzen. Es ergibt sich damit das Klassendiagramm in Abbildung 6.

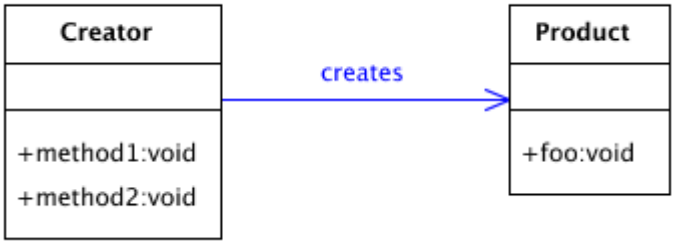

Abbildung 6: Vereinfachtes Klassendiagramm um den Precursor anzudeuten

In diesem Fall kann man den *Precursor* einfach herleiten:

> Der Hersteller muß eine Instanz des Produkts erzeugen.

Um dieses Vorstadium des Entwurfsmusters im Quelltext zu finden, wird noch die richtige Analysefunktion benötigt.

> creates(creator, product)

Damit hat man alle Information für den Precursor zusammen.

2.4 Minipattern und Minitransformation

Als nächster Schritt steht nun die Unterteilung der Transformation in *Minipatterns* an. Zuvor allerdings muß noch geklärt werden, was es mit einem *Minipattern* auf sich hat.

Minipatterns sind immer wiederkehrende Muster innerhalb der Entwurfsmuster. Diese Unterteilung ist sinnvoll, da man so Entwurfsmuster aus einer Grundmenge von Fragmenten zusammensetzten kann, die alle Entwurfsmuster gemeinsamen haben. Zusammengefasst in einer Bibliothek, können die *Minipatterns* vom Transformationssystem jederzeit wiederverwendet werden, was viel Arbeit erspart, da nicht für jedes Entwurfsmuster ein komplett neuer Algorithmus zur automatischen Transformation entwickelt werden muß. Stattdessen versucht man, dass Entwurfsmuster auf eine endliche Menge von *Minipatterns* zu reduzieren, diese dann miteinander als Refactoring-Transaktion zu vereinen und wieder zu verwenden.

Minipatterns sind eine Struktur der Entwurfsebene. Die benötigte Transformation, die mit diesem Minipatterns beschrieben wird, ist ein festzulegender Algorithmus. Dieser wird *Minitransformation* genannt und ist nichts anderes als eine spezielle Refactoring-Transaktion.

Für das hier benutzte Beispiel benötigt man insgesammt vier *Minitransformationen.*

1. *Abstraction*: Fügt dem *Produkt* eine Schnittstelle hinzu, über das der *Hersteller* das *Produkt* benutzen kann, welches er erzeugt.
2. *EncapsulateConstruction*: Die Konstruktion des *Produktes* innerhalb des *Herstellers* wird in einer dafür bereitgestellten Methode gekapselt.
3. *AbstractAccess*: Objektreferenzen vom Typ *Produkt* werden im *Hersteller* durch die Schnittstelle ersetzt.
4. *PartialAbstraction*: Der *Hersteller* wird in eine abstrakte Oberklasse verwandelt, die aktuelle Implementierung wird in einer Unterklasse realisiert.

Die vier Minipatterns sind in den folgenden Unterabschnitten der Reihenfolge nach definiert.

2.4.1 Abstraction Minitransformation

Die *Abstraction Minitransformation* fügt eine Schnittstelle zu einer Klasse hinzu, um einen abstrakteren Zugang zu der Klasse zu gewinnen.

Abstraction(Class *c*, String *newName*) {

    Interface inf = abstractClass(*c*, *newName*);
    addInterface(inf);
    addImplementLink(*c*, inf);

}

Zuerst erzeugt man eine Schnittstelle *inf*, die alle öffentlichen Methoden der Klasse *c* enthält. Danach wird die Schnittstelle dem Programm hinzugefügt und die *implements* Anweisung für ein Interface in das Java Programm integriert.

Die Vor- und Nachbedingung haben folgende Form:

Vorbedingung
Die Klasse muß existieren

    isClass(*c*)

und außerdem darf keine Klasse und auch kein Interface mit dem Namen *newName* im System vorhanden sein.

$$\neg\text{isClass}(newName) \wedge \neg\text{isInterface}(newName)$$

Nachbedingung

Es muß ein Interface mit dem Namen *newName* eingeführt worden sein.

$$\text{nameOf'} = \text{nameOf}[inf/newName]$$
$$\text{isInterface'} = \text{isInterface}[inf/\text{true}]$$

Die Klasse c und das Interface *inf* haben die gleiche öffentliche Schnittstelle

$$\text{equalInterface'} = \text{equalInterface}[(c,inf)/\text{true}]$$

Die Klasse c ist mit dem neuen Interface verknüpft

$$\text{implementsInterface'} = \text{implementsInterface}[(c,inf)/\text{true}\ ]$$

Beispiel: Anwendung von *Abstraction Minitransformation*

Wird nun die Minitransformation auf das Beispiel angewandt, so erhält man das Klassendiagramm in Abbildung 7.

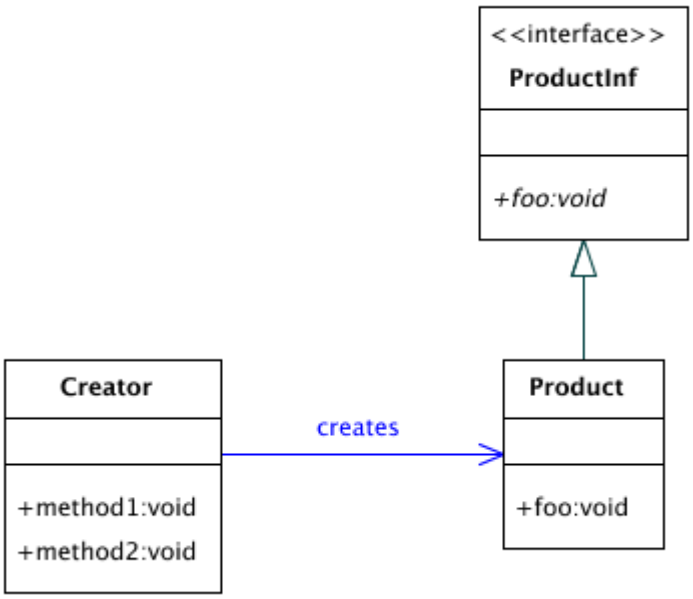

Abbildung 7: Klassendiagramm nach Anwendung der Minitransformation *Abstraction*

2.4.2 EncapsulateConstruction Minitransformation

Um die Erzeugung eines Produktes durch die Hersteller-Klasse in eine dafür abgestellte Funktion zu verlagern, benötigt man die Minitransformation *EncapsulateConstruction*. Dabei müssen auch alle Stellen innerhalb des *Herstellers*, die bisher über den Operator **new** die Produkte erzeugt haben, in einen Aufruf der neuen erzeugten Methode geändert werden.

Da diese Minitransformation relativ komplex ist und unter anderem auch Schleifen benutzt, wird an dieser Stelle auf die Berechnung der Vor- und Nachbedingungen innerhalb der Transformation verzichtet und auf [Cin01] (Seite 49ff) verwiesen.

EncapsulateConstruction(Class *creator*, Class *product*, String *createProduct* {

ForAll *c*:Constructor, classOf(*c*) = product {
Method m = makeAbstract(*c*, *createProduct*);
addMethod(*creator*, m);
}
ForAll e:ObjectCreationExprn, classCreated(e) = *product*
∧ containingClass(e) = *creator* ∧ nameOf(containingMethod(e))
≠ *createProduct* {
replaceObjCreationWithMethInvocation(e, *createProdcut*);
}

}

Nach erfolgreicher Anwendung der Minitransformation ergibt sich folgendes Klassendiagramm in Abbildung 8.

2.4.3 AbstractAccess Minitransformation

Nachdem wir nun die Erzeugung der *Produkte* in einer (abstrakten) Methode gekapselt haben, müssen wir dafür sorgen, dass innerhalb des *Herstellers* ausschließlich die Produkte über das Interface *ProductInf* angesprochen werden und nicht mehr über *Product* um endgültig eine Entkopplung zu erreichen.

Praktisch umgesetzt entsteht nachfolgende Refactoring-Transaktion:

AbstractAccess (Class *context*, Class *concrete*, Interface *inf*, SetOfString *skipMethods*) {

ForAll o:ObjectRef, typeOf(o)=*concrete*, containingClass(o)=*context*, nameOf(containingMethod(o)) ∉ *skipMethods* {

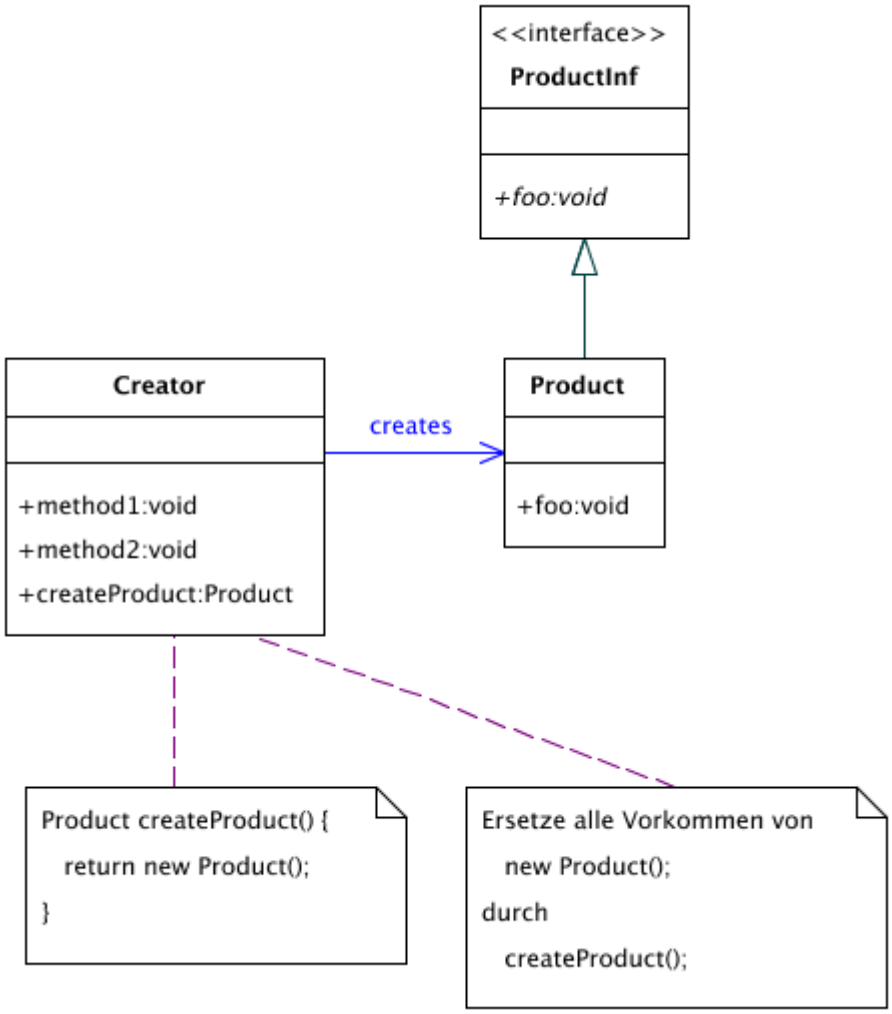

Abbildung 8: Klassendiagramm nach Anwendung der Minitransformation *Encapsulate Construction*

```
            replaceClassWithInterface(o, inf);
        }
    }
```

Die Transformation ersetzt in der Klasse *context* alle Objektreferenzen in den Methoden (welche nicht in *skipMethods* angegeben sind) und dabei auf die Klasse *concrete* verweisen, durch die Schnittstelle *inf*.

Dabei müssen auch hier Bedingungen eingehalten werden, um die externe Semantik beizubehalten.

Vorbedingung

Die Schnittstelle *inf* sowie die Klassen *context* und *concrete* müssen existieren. Des Weiteren ist es notwendig, dass *concrete* die Schnittstelle benutzt.

> isInterface(*inf*) $\wedge$ isClass(*context*) $\wedge$ isClass(*concrete* $\wedge$ implementsInterface(*concrete*, *inf*)

Keine der zu ändernden Objektreferenzen darf statische Methoden aus der *concrete*-Klasse benutzen.

> $\forall$ m:Method, m $\in$ *concrete*, isStatic(m) $\circ$ $\forall$ o:ObjectRef, typeOf(o)=concrete, containingClass(o)=*context* $\circ$ $\neg$ uses(o,m)

Gleiches gilt für öffentliche Variablen in *concrete*.

> $\forall$ f:Field, f $\in$ *concrete*, isPublic(f) $\circ$ $\forall$ o:ObjectRef, typeOf(o)=concrete, containingClass(o)=*context* $\bullet$ $\neg$ uses(o,f)

Nachbedingung

Alle angegebenen Referenzen auf *concrete* in *context* verweisen nun auf die Schnittstelle.

> $\forall$ o:ObjectRef, typeOf(o)=*concrete*, containingClass(o)=*context*, nameOf(containingMethod(o)) $\notin$ *skipMethods* $\bullet$ typeOf' = typeOf[*o*/*inf*]

Durch diese Minitransformation ergibt sich Abbildung 9

2.4.4 PartialAbstraction Minitransformation

Mit der vierten und letzten Minitransformation möchten wir erreichen, dass auch der *Creator* verallgemeinert werden kann. Dazu schaffen wir eine abstrakte Oberklasse, in die ausgewählte Methoden verschoben werden; die Oberklasse bekommt den Namen *AbsCreator*. Die schon bekannte Klasse *Creator* erweitert jene neu gebildete Oberklasse und überschreibt die abstrakte Methode *createProduct()*. Das Ergebnis soll dem UML-Klassendiagram in Abbildung 10 entsprechen.

Der Aufbau der Minitransformation ist damit einfach:

> **PartialAbstraction**(Class *concrete*, String *newName*, SetOfStrings *abstractMethods*) {
>
> > Class abstract = createEmptyClass/*newName*);
> > addClass(abstract, superclass(*concrete*), *concrete*);
> > ForAll m:Method, m $\in$ *concrete*, nameOf(m) $\in$ *abstractMethods* {

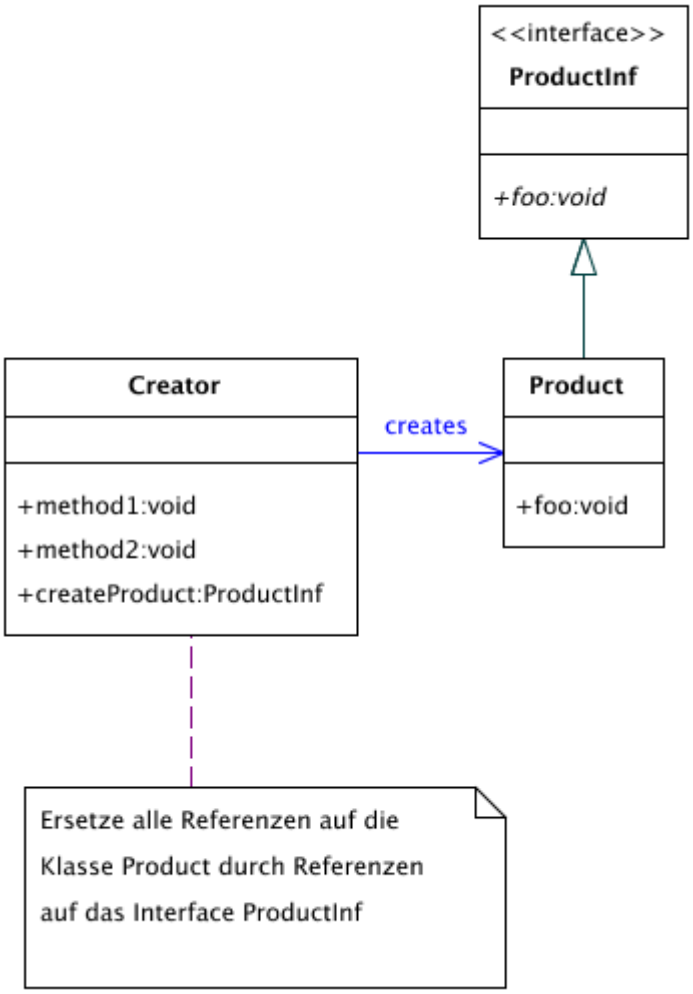

Abbildung 9: Klassendiagramm nach Anwendung der Minitransformation *AbstractAccess*

```
        Method absMethod = abstratMethod(m);
        addMethod(abstract, absMethod);
    }
    ForAll m:Method, m ∈ concrete, nameOf(m) ∉ abstract-
    Methods {
        pullUpMethod(m);
    }
}
```

Um den Transformationsprozess nicht in die Länge zu ziehen, wird für Berechnung der Vor- und Nachbedingungen auf [Cin01] verwiesen. Da der Nachweis

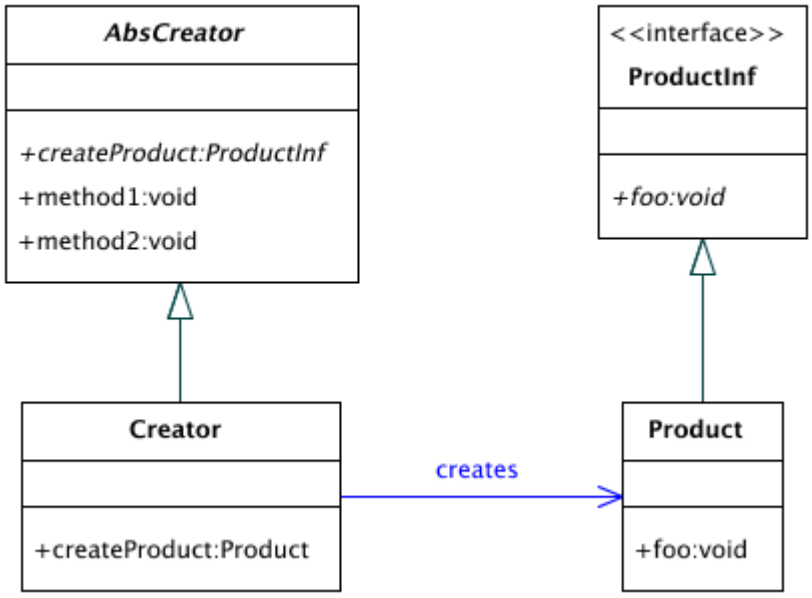

Abbildung 10: Klassendiagramm nach Anwendung der Minitransformation *PartialAbstraction*

keine neuen Erkenntnisse liefert.

2.5 Die vollständige Factory Method Refactoring-Transaktion

Da wir nun alle Minitranformationen zusammen getragen und anhand des Beispiels gezeigt haben, wie die ganze Transformation aus Teilen zusammengestezt wird, wird hier das Endergebnis in Form der *Factory Method* Refactoring-Transaktion präsentiert und noch leicht vereinfacht.

Wie weiter oben schon beschrieben ist, benötigen wir für den Nachweis der Verhaltenskorrektheit die Nachbedingungen nicht, da sie ja das Ergebnis des erfolgreichen Transformationsunterschrittes sind. Daher muß nur gezeigt werden, dass vor einem Teilschritt alle Bedingungen wahr sind.

applyFactoryMethod(Class *creator), Class product, String productInf, String absCreator, String createProduct)* {

Abstraction(*product)*, *productInf*);
EncapsulateConstruction(*creator*, *product*, *createProduct*);
AbstractAccess(*creator*, *product*, *prodcutInf*, createProduct);
PartialAbstraction(*creator*, *absCreator*, *createProduct*);

}

Ausreichende Vorbeding ist damit:

1. Die Klassen *creator* und *product* existieren. Außerdem sind keine Klassen oder Schnittstellen mit den Namen *absCreator* und *productInf* im System vorhanden.

   isClass(*creator*) $\wedge$ isClass(*product*) $\wedge$
   $\neg$ isClass(*absCreator*) $\wedge$ $\neg$ isInterface(*absCreator*) $\wedge$
   $\neg$ isClass(*productInf*) $\wedge$ $\neg$ isInterface(*productInf*)

2. Des Weiteren darf *creator* keine Methoden mit dem Namen *createProduct*, welche die gleiche Signatur wie der Konstruktor der Klasse *product* haben, besitzen.

   $\forall$ c:Constructor, c $\in$ *product* $\bullet$ $\neg$ defines(*creator*, *createProduct*, sigOf(c))

3. Der Hersteller kann Produkte erzeugen.

   creates(*creator*, *product*)

4. Öffentlich zugängliche Felder in der Produkt-Klasse dürfen durch kein Produkt-Objekt referenziert werden, die widerum in der Hersteller-Klassen referenziert werden.

   $\forall$ f:Field, f $\in$ *product* $\bullet$ $\forall$ o:ObjectRef, typeOf(o) = *product*, containingClass(o) = *creator* $\bullet$ $\neg$ uses(o,f)

5. Keine der Variablen in der Klasse *product*, die von Methoden benutzt werden, dürfen öffentlich sein.

   $\forall$ f:Field, m:Method, f $\in$ *concrete*, m $\in$ *concrete* , uses(m,f) $\bullet$ $\neg$ isPublic(f)

6. Statische Methoden in *product* dürfen nicht in Objekten dieser Klassen referenziert werden, wenn diese im Hersteller referenziert werden.

   $\forall$ m:Method, m $\in$ *product*, isStatic(m) $\bullet$ $\forall$ o:ObjectRef, typeOf(o==*product*, containingClass(o) = *creator*, $\bullet$ $\neg$ uses(o,m)

Damit ist die automatische Codetransformation erfolgreich abgeschlossen.

3 Nicht transformierbares Entwurfsmuster

Als Beispiel für ein Entwurfsmuster, dass nicht automatisch transformierbar ist, wird das Muster *Facade* benutzt. Die Intention hinter dem Entwurfsmuster ist eine vereinheitlichte Schnittstelle für ein Subsystem aus mehreren Klassen anzubieten. Der Precursor leitet sich aus dieser Beschreibung ab: Mehrere Klassen (Clients) benutzen über eine einzelne Klasse (die spätere *Facade*-Klasse) die Klassen aus dem Subsystem, wie in Abbildung 11 zu sehen ist.

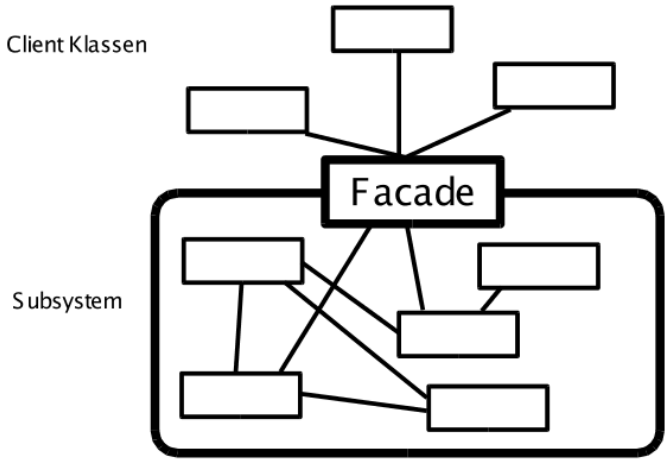

Abbildung 11: Entwurfsmuster *Facade*

Das Entwurfsmuster kann praktisch zum Beispiel für eine Schnittstelle zu einem Compilersystem benutzen werden. Dabei besteht das Subsystem aus den ganzen Komponenten, die ein Compiler braucht, z. B. Scanner, Parser, Generator usw. Als *Facade*-Klasse verwendet man die Klasse *Compiler*, die mit dem Methodenaufruf *compile()* den Quellcode übersetzt und eigenständig die Klassen im Subsystem benutzt.

Bei dem Versuch, eine Transformation zu finden, stößt man unweigerlich auf ein nicht lösbares Problem.

Angenommen, ein *Client* benutzt mehrere Instanzen des Subsystem in unterschiedlicher Weise. Vom oben genannten praktischen Einsatz ausgehend, gebraucht irgendeine Klasse (Client) bestimmte Klassen aus einem sehr komplexen Compilersystem, dass nach einer erfolgreichen Transformation, über die allgemeine Schnittstelle einmal angesprochen werden soll. Im Moment gibt es aber noch keine einheitliche Schnittstelle, so dass jeder Client auf unterschiedliche Art und Weise den Parser, Generator und andere Elemente benutzt. Dabei sind einige Zugriffe auf den Compiler einfach, andere aber extrem verschachtelt und komplex. Hier liegt der Knackpunkt. Für die Schnittstelle brauchen wir einen gemeinsamen Nenner für die Zugriffe von Clients auf das Compilersystem. Sind diese Aufrufe sehr komplex, kann trotz einer guten Analyse nicht dieser kleinste gemeinsame Nenner gefunden werden, der als Schnittstelle gebraucht wird.

Damit ist dieses Entwurfsmuster nicht automatisch transformierbar.

4 Unterstützung durch Werkzeuge

Automatische Integration von Entwurfsmustern auf Entwurfsebene in bestehende Software wird heute zum Teil schon von modernen Modellierungswerkzeugen unterstützt. Ein Exemplar, dass dieses Problem teilweise lösen kann, ist die Software *Together* (http://www.togethersoft.com).

4.1 Together

Together ist ein Werkzeug, dass hauptsächlich auf Entwurfsebene über UML-Diagramme arbeitet. Inzwischen kann diese Programm alles, was man von einer Entwicklungsumgebung verlangt. Im Rahmen dieser Seminararbeit haben wir daran aber kein weiteres Interesse.

Wir wollen in diesem Abschnitt versuchen, dass Beispiel aus dem zweiten Abschnitt mit Hilfe dieses Werkzeuges automatisch zu transformieren. Bisher haben wir uns nicht besonders für die Quellcodeebene interessiert, da wir die Methode der automatischen Integration theoretisch beschrieben haben und die Entwurfssicht dafür ausreichend war. Sollte die Transformation aber funktionieren, ist es sicherlich von Interesse, wie der Quellcode danach aussieht.

Wir beginnen mit der Situation, die wir für die Erklärung des *Precursors* benutzt haben. Das UML-Klassendiagramm dazu ist in Abbildung 2.3 zu sehen. Der Quellcode sieht für die beiden beteiligten Klassen wie folgt aus.

```
public class Creator {
    public void method1() {
        Product p = new Product();
    }

    public void method2() {
    }
}
```

```
public class Product {
    public void foo() {
    }
}
```

Ablauf der Transformation

Hat man nun in dem Programm eine Klasse im Klassendiagramm ausgewählt und öffnet das Kontextmenü, so kommt man über den Menüpunkt *Choose Pattern* in einen Dialog, der in Abbildung 12 zu sehen ist.

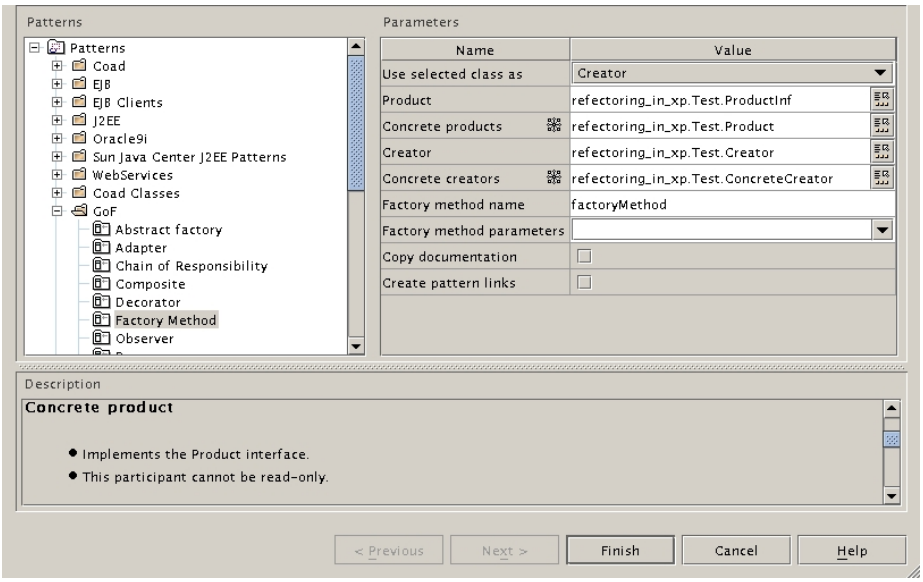

Abbildung 12: Dialog zur Auswahl des Entwurfsmusters

Dort füllt man die Felder entsprechend aus. Dabei kann angegeben werden, welche der aktuellen Klassen später den einzelnen Komponenten des Entwurfsmusters entsprechen soll. Befolgt man die Anleitung im unteren Teil des Fensters und führt die Transformation aus, so erhält man als Ergebnis das Klassendiagramm in Abbildung 13.

Ein Schönheitsfehler fällt beim ersten Betrachten sofort auf. Die öffentliche Methode *foo()* sollte eigentlich in der Schnittstelle untergebracht sein, um einen abstrakteren Zugriff auf Produkte zu bekommen. Hier muß also Hand angelegt und die Methode manuell in die Schnittstellendefinition verlagert werden.

Nun wollen wir uns den generierten Quellcode ansehen.

```
abstract public class Creator {
    public void method1() {
        Product p = new Product();
    }

    public void method2() {}

    public abstract ProductInf factoryMethod();
}

public class ConcreteCreator extends Creator {
    public ProductInf factoryMethod() {
        return new Product();
    }
```

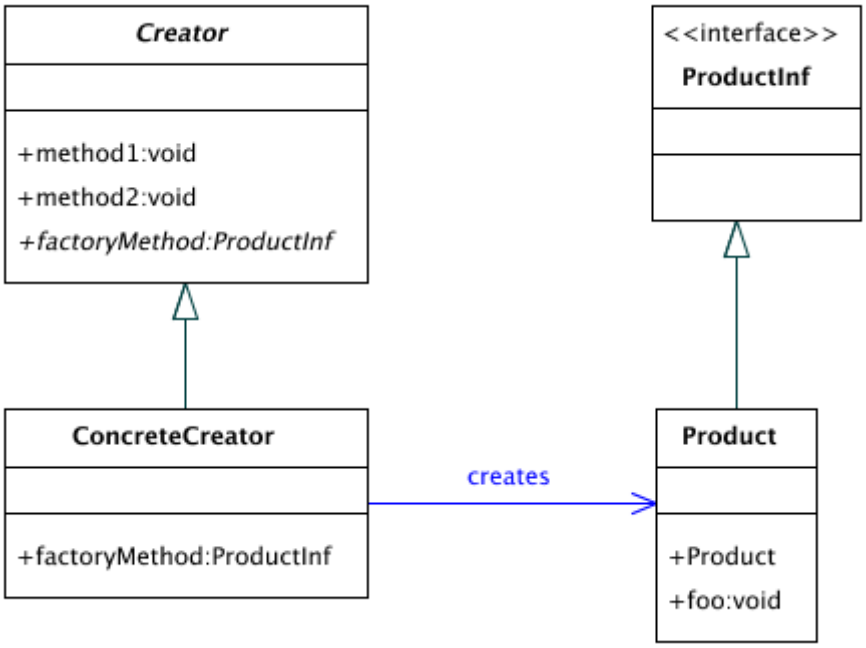

Abbildung 13: UML-Klassendiagramm nach der Transformation durch *Together*

```
}

public interface ProductInf {
}

public class Product implements ProductInf {
    public Product(){}

    public void foo() {}
}
```

Gleich in der ersten Klasse ist ein Problem auszumachen. In der Methode *method1()* werden nach wie vor über den Operator *new* Instanzen von *Product* gebildet. Diese Zeile müßte korrekt folgendermaßen aussehen:

```
    ProductInf p = factoryMethod();
```

Der Rest des generierten Programmcodes ist in Ordung.

Es ist also festzuhalten, dass eine vollständig korrekte Transformation nicht gelungen ist, da nicht alle Seiteneffekte bei der Transformation durch das Programm bedacht worden sind. Es ist nach wie vor notwendig, manuell Korrekturen durchzuführen.

5 Schlussfolgerung

Im letzten Abschnitt dieser Seminararbeit wird eine kurze Schlussfolgerung gezogen, wie gut das vorgestellte Modell funktioniert und wo die Schwächen liegen. Außerdem wird ein Überblick über die Automatisierbarkeit der Transformation von Entwurfsmustern aus dem Katalog von Gamma gegeben.

5.1 Schlussfolgerung

Wie in Abschnitt 2 gezeigt, ist es in der Theorie möglich bestimmte Entwurfsmuster automatisch zu transformieren. Leider zeigt der darauf folgende Abschnitt 3, dass es auch Muster gibt, die nicht automatisch transformierbar sind. In der nachfolgenden Tabelle sind alle Entwurfsmuster von Gamma [EGV95] aufgelistet und nach Durchführbarkeit der automatischen Transformierbarkeit bewertet.

Der Automatisierungsgrad wird dabei in der gleichnamigen Spalte in insgesamt drei Fälle unterteilt.

vollständig : Die vorgestellte Methode ließ sich sehr gut anwenden und es wurde ein funktionsfähiger *Precursor* gefunden.

teilweise : Es existiert eine Refactoring-Transaktion für dieses Entwurfsmuster. Das produzierte Ergebnis weisst aber Mängel auf, da die Transformation nicht vollständig ist.

nicht transformierbar : Es wurde keine Transformation gefunden, oder die Ergebnisse der entsprechenden Transformation weisen erhebliche Mängel auf, die dazu führen, dass die Methode nicht praktisch einsetzbar ist.

Name des Entwurfsmusters,	Typ	Automatisierungsgrad
Abstract Factory	Erzeuger	vollständig
Builder	Erzeuger	vollständig
Factory Method	Erzeuger	vollständig
Prototype	Erzeuger	vollständig
Singleton	Erzeuger	vollständig
Adapter	Struktur	vollständig
Bridge	Struktur	vollständig
Composite	Struktur	vollständig
Decorator	Struktur	teilweise
Facade	Struktur	nicht transformierbar
Flyweight	Struktur	nicht transformierbar
Proxy	Struktur	teilweise
Chain of Responsibility	Verhaltensstruktur	vollständig
Command	Verhaltensstruktur	teilweise
Interpreter	Verhaltensstruktur	nicht transformierbar
Iterator	Verhaltensstruktur	teilweise
Mediator	Verhaltensstruktur	nicht transformierbar
Memento	Verhaltensstruktur	teilweise
Observer	Verhaltensstruktur	nicht transformierbar
State	Verhaltensstruktur	teilweise
Strategy	Verhaltensstruktur	vollständig
Template Method	Verhaltensstruktur	vollständig
Visitor	Verhaltensstruktur	nicht transformierbar

Diese Übersicht zeigt, dass alle erzeugenden Entwurfsmuster transformierbar sind. Bei den beiden anderen Arten ist das Ergebnis durchwachsen und reicht von vollständig *transformierbar* bis *nicht transformierbar*. Dies ist aber nicht ganz überraschend, da der Ansatz der benutzten Methode, der *Precursor*, wie auch die Überprüfung der Vor- und Nachbedingungen auf eine Analyse der Programmstruktur zurückgeführt wird. Verhalten kann man mit dieser Methode daher schlecht bis gar nicht erfassen, warum auch die Verhaltensstrukturen nur begrenzt transformierbar sind.

Die Ursache für das durchschnittliche Abschneiden der Struktur-Entwurfsmuster liegt darin begründet, dass manche Entwurfsmuster in Teilen auch Aspekte von Verhaltens-Entwurfsmustern besitzen. So zum Beispiel das Muster *Facade*, welches unter 3 schon beschrieben worden ist.

Bisher haben wir die vorgestellte Methode nur auf dem Papier angewandt. Es soll aber nicht verschwiegen werden, dass Mel Ó Cinnéide im Rahmen seiner Dissertation auch einen Prototypen entwickelt hat, der automatische Transformationen ausführen kann. Das so genannte DPT (Design Pattern Tool) kann vier Entwurfsmuster nach Angabe von Mel Ó Cinnéide erfolgreich anwenden. Leider ist dieses Programm nicht öffentlich verfügbar, weshalb keine genaueren Aussagen über die Möglichkeiten in der Praxis gemacht werden können.

Literatur

[AH00] Gunter Saake Andreas Heuer. *Datenbanken: Konzepte und Sprachen 2., aktualisierte und erweiterte Auflage*. mitp, 2000. 6

[Cin00] Mel Ó Cinnéide. *Automated Refactoring to Introduce Design Patterns*. In *Proceedings of the International Conference on Software Engineering (Doctoral Workshop)*, Limerick, 2000. 2

[Cin01] Mel Ó Cinnéide. *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, 2001. 2, 8, 9, 18, 21

[CN98] Mel Ó Cinnéide and Paddy Nixon. *Program Restructuring to Introduce Design Patterns*. In *Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering*, pages 79–80, Brussels, Jul 1998. European Conference on Object- Oriented Programming. 2

[CN99a] Mel Ó Cinnéide and Paddy Nixon. *A Methodology for the Automated Introduction of Design Patterns*. In *IEEE International Conference on Software Maintance*, pages 463–472. IEEE Computer Society Press, 1999. 2

[CN99b] Mel Ó Cinnéide and Paddy Nixon. *Automated Application of Design Patterns to Legacy Code*. In *Proceedings of the Workshop on Experiences in Object-Oriented Re-Engineering*, Lisbon, Jun 1999. European Conference on Object-Oriented Programming. 2

[CN00] Mel Ó Cinnéide and Paddy Nixon. *Composite Refactorings for Java Programs*. In *Proceedings of the Workshop on Formal Techniques for Java Programs*. European Conference on Object-Oriented Programming, 2000. 6

[EGV95] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 3, 28

[MFR00] John Brant William Opdyke Martin Fowler, Kent Beck and Don Roberts. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 2000. 6

[Opd92] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. 6

[zb02] zeroCode build. *Software Lifecycle Costs*. zeroCode-build, May 2002. 3