Unabhängige Erweiterbarkeit für Aspekt-Orientierte Systeme

Michael Austermann, Pascal Costanza und Günter Kniesel
Universität Bonn, Institut für Informatik III
Römerstraße 164, 53117 Bonn
{austerm|costanza|gk}@cs.uni-bonn.de

1 Einleitung

Die Anforderungen an Software sind einem ständigen Wandel unterworfen. Ein besonderes Problem stellen dabei nicht antizipierte Anforderungsänderungen dar. Im Rahmen herkömmlicher objekt-orientierter Technologie erfordern sie eine Änderung des zugrundeliegenden Quellprogramms, mit all den damit verbundenen Wartungsproblemen. Als Alternative findet in den letzte Jahren die Idee wieder stärkere Beachtung, solch unantizipierte Änderungen inkrementell als *Programm-Transformationen* zu beschreiben und automatisch ausführen zu lassen. In diesem Kontext sehen wir zwei Trends als besonders bedeutungsvoll:

- **Aspekt-Orientierte Programmierung (AOP)** [1] fasst semantisch zusammengehörende Transformationen in eigenen Programm-Modulen, sogenannten Aspekten, zusammen. AOP bietet somit ein alternatives Konzept zur Modularisierung objekt-orientierter Programme und zum Umgang mit Programmänderungen.

- **Ladezeittransformationen** Spätestens seit dem wegweisenden BCA-Ansatz von Keller und Hölzle [2] ist es klar, dass unvorhergesehene Transformationen am besten zur *Ladezeit* des zu transformierenden Programms durchzuführen sind. So lassen sich z. B. auch Klassen transformieren, deren Quellcode nicht zur Verfügung steht oder die erst zur Laufzeit bestimmt werden.

Im folgenden werden wir auf Schwächen bestehender AOP-Ansätze im Hinblick auf unabhängige Erweiterbarkeit eingehen und dazu einen partiellen Lösungsvorschlag vorstellen. Danach werden wir ein Framework zur Ladezeit-Transformation von Java-Programmen beschreiben, das unseren Lösungsvorschlag umsetzt und gleichzeitig als „back end“ beliebiger AOP-Systeme dienen kann.

2 Probleme der unantizipierten Aspekt-Komposition

Der unantizipierte gemeinsame Einsatz von unabhängig entwickelten Aspekten ist ein bisher in AOP-Systemen nicht zufriedenstellend gelöstes Problem: Wenn unabhängig entwickelte Aspekte ohne besonderes Wissen über ihre Interna gemeinsam eingesetzt werden, kann es zu unerwünschten Seiteneffekten kommen. Eine explizite Komposition von Aspekten kann solche Seiteneffekte verhindern, ist aber nicht anwendbar, wenn Aspekte komponiert werden sollen, deren gemeinsame Verwendung von keinem der beteiligten Programmierer antizipiert worden ist. Gesucht ist also alternativ ein automatisches Verfahren zur Komposition von Aspekten, das keinerlei Wissen über die beteiligten Aspekte voraussetzt und trotzdem unerwünschte Seiteneffekte verhindert.

Das beschriebene Problem lässt sich in zwei Teilprobleme aufspalten, die auf der Suche nach einer Lösung im folgenden diskutiert werden.

Um nicht auf einen bestimmten AOP-Ansatz festgelegt zu erscheinen, wird im folgenden nur allgemein von *Programmtransformationen* gesprochen. Genau genommen geht es um *positiv getriggerte* Programmtransformationen – Transformationen können nur durch das Vorhandensein (nicht durch das Fehlen) einer Programmeigenschaft ausgelöst werden. Alle uns bekannten AOP-Ansätze weisen diese Eigenschaft auf.

2.1 Gegenseitiges Triggern

Das erste Problem der unantizipierten Komposition von Aspekten ist die Möglichkeit des „gegenseitigen Triggerns“: Eine spät zum Zuge kommende Transformation fügt möglicherweise Eigenschaften zu einem Programm hinzu, die die erneute Anwendung bereits durchgeführter Transformationen notwendig machen.

Der Umstand, dass Transformationen sich gegenseitig triggern können, ist ein Problem, weil in bekannten AOP-Ansätzen jede Transformation *nur einmal* angewendet wird – es kann somit sein, dass die resultierenden Programme unvollständig sind, da die Bestandteile

```
    public class C' {
       public B b = new B();
(1)    private int b_counter = 0;

(2)    public void setB(B _b) {
          this.b = _b;
       }
(2)    public B    getB()    {
(3)       b_counter++;
          return this.b;
       }

       public void manipulateB() {
(1)       b_counter++;
(2)       getB().doSomething();
       }
    }
```

(1) Counter, (2) Access, (3) Counter, ...

```
    public class C'' {
       public B b = new B();
(2)    private int b_counter = 0;

(1)    public void setB(B _b) {
          this.b = _b;
       }
(1)    public B    getB()    {
(2)       b_counter++;
          return this.b;
       }

       public void manipulateB() {

(1)       getB().doSomething();
       }
    }
```

(1) Access, (2) Counter, ...

Abbildung 1. Je nachdem, welcher Aspekt zuerst auf die Klasse C angewendet wird, entstehen unterschiedliche Ergebnisse.

fehlen, die erst durch eine wiederholte Anwendung von Transformationen entstehen würden.

Offensichtlich müssten Transformationen so lange iteriert werden, bis die sich daraus ergebenden Änderungen keine Trigger-Bedingung mehr feuern, m. a. W. ein Fixpunkt erreicht ist. Das kann im Extremfall sogar dann notwendig sein, wenn eine *einzige* Transformation beteiligt ist, jedoch das Programm, auf das sie angewendet wird, hinsichtlich der für die Tranformation relevanten Eigenschaften eine zyklische Struktur aufweist. Ein entsprechendes Beispiel wird in Abschnitt 4 vorgestellt.

Die automatische Iteration von Transformationen bzw. Aspekt-Anwendungen wirft jedoch sofort zwei neue Fragen auf:

1. Terminiert die Iteration?
2. Liefert sie ein eindeutiges Ergebnis?

2.2 Reihenfolge-Abhängigkeit

Leider ist die Antwort auf beide Fragen *im allgemeinen Fall* negativ. Wir zeigen hier nur anhand des folgenden Beispiels, dass reihenfolgeabhängig verschiedene Ergebnisse möglich sind.

Angenommen folgendes Programm soll von zwei Aspekten Access und Counter, transformiert werden:

```
public class C {
   public B b = new B();

   public void manipulateB() {
      b.doSomething(); }
}
```

Der Aspekt Access erweitert eine Klasse für jedes public Feld um zwei Zugriffsmethoden und ersetzt alle direkten Zugriffe auf dieses Feld durch die entsprechenden Methodenaufrufe. Der Aspekt Counter erweitert eine Klasse für jedes Feld um einen Zähler, der die lesenden Zugriffe auf diese Feld zählt.

Wie oben beschrieben werden diese beiden Aspekte solange abwechselnd angewendet, bis keine Änderungen mehr am Programm erfolgen. Je nachdem, mit welchem der beiden Aspekte diese Iteration beginnt, ergeben sich unterschiedlichen Ergebnisse, wie in Abbildung 1 ersichtlich ist: Der Unterschied zwischen C' und C" besteht darin, dass in der Methode manipulateB() in der ersten Variante der Zähler für den Zugriff auf das Feld b inkrementiert wird, während dies in C" nicht passiert. Das ergibt sich aus der Tatsache, dass im zweiten Fall durch Access der Zugriff auf b im ersten Schritt durch den Aufruf einer Zugriffsmethode ersetzt wird, *bevor* Counter ihn im zweiten Schritt erkennen kann. Im Ergebnis wird das Feld b_counter in C' doppelt so oft erhöht wie in C".

3 Aufteilung von Transformationen

Am obigen Beispiel lässt sich neben dem dargestellten negativen Ergebnis eine weitere wichtige Beobachtung machen: Der resultierende *Methodenrumpf* von manipulateB ist abhängig von der Reihenfolge der Transformationen. Die resultierenden *Schnittstellen* der Klassen C' und C" sind jedoch in beiden Fällen gleich, also offensichtlich unabhängig von der Reihenfolge der Transformationen.

Tatsächlich lässt sich nachweisen, dass die Transformation von Schnittstellen immer reihenfolgenunabhängig ist. Dies ergibt sich aus der Tatsache, dass die Schnittstelle einer Klasse eine *ungeordnete Menge* von Namen und Signaturen ist. Dagegen ist die Implementation einer Methode eine *geordnete Liste* von Befehlen. Es ist unmittelbar klar, dass die Reihenfolge, in der Elemente in eine Menge eingefügt werden, keinen Einfluss auf die resultierende Menge hat, während diese Reihenfolge bei einer Liste einen wesentlichen Unterschied ausmacht.

Daher ist es sinnvoll, Programmtransformationen in zwei Klassen zu unterteilen:

- Codetransformationen und
- Schnittstellentransformationen

Codetransformationen verändern existierenden Methodencode. Zu den *Schnittstellentransformationen* zählen wir das Hinzufügen von Klassen, Vererbungsbeziehungen, Feldern und Methoden (einschließlich initialem Methodencode).

Im Gegensatz zu Feldern, für die in Java Defaultwerte angenommen werden, wenn kein expliziter Initialwert definiert ist, kann für Methoden im allgemeinen kein sinnvolles „Defaultverhalten“ definiert werden. Daher muss für nicht-abstrakte Methoden explizit ein Initialcode angegeben werden, der später durch Codetransformationen noch verändert werden kann. Das Hinzufügen von Methoden mitsamt Initialcode wird daher als reine Schnittstellentransformation betrachtet.

Ebenso wird die Änderung von Annotationen als Schnittstellenänderung betrachtet. Hierbei werden jedoch bei Zugriffsrechten (public, protected, etc.) nur Änderungen zugelassen, die die Sichtbarkeit eines Elements erweitern.

Allgemein werden nur *monotone* Schnittstellentransformationen zugelassen. Das bedeutet, dass nur neue Teile zu einem Programm hinzugefügt, oder bestehende Teile allgemeiner zugänglich gemacht werden können, aber nie etwas gelöscht oder in der Sichtbarkeit eingeschränkt wird. *Echte* Veränderungen können nur als Codetransformationen ausgedrückt werden.

In Bezug auf unsere ursprüngliche Problemstellung können wir obiges Ergebnis wie folgt zusammenfassen:

- Aspekte, die sich auf Schnittstellentransformationen beschränken, können unabhängig entwickelt und automatisch komponiert werden.
- Aspekte, die Codetransformationen beinhalten können nach dem bisherigen Stand unserer Erkenntnisse nicht automatisch komponiert werden.

Diese Erkenntnisse wurden im Rahmen des TAILOR Projekts am Institut für Informatik III der Universität Bonn [5] in ein Werkzeug zur Ladezeit-Transformation von Java-Programmen umgesetzt, das im folgenden kurz vorgestellt wird.

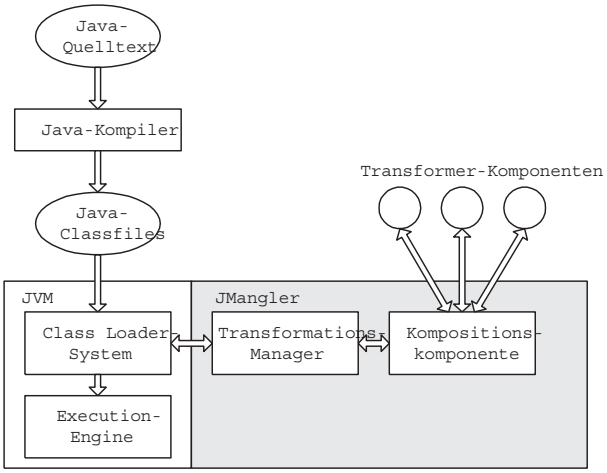

Abbildung 2. Architektur des JMangler-Frameworks

4 JMangler

JMangler ist ein in Java entwickeltes Framework, das Transformationen von kompilierten Java-Klassen während des Ladens in eine Java Virtual Machine erlaubt. Programmierer können eigene *Transformer-Komponenten*[1] in Java schreiben. Diese erhalten von JMangler die geladenen Java-Klassen als objektorientierte Repräsentationen zur Analyse und entscheiden, welche Transformationen darauf von JMangler durchgeführt werden sollen.

JMangler bietet keine eigene Spezifikationssprache für Transformationen an. Stattdessen muss die Analyse und Transformation von Java-Klassen in reinem Java erfolgen. Hierzu ist eine genaue Kenntnis des Java Class File Formats notwendig. Die Realisation eines Übersetzers, beispielsweise von AspectJ in Transformer-Komponenten von JMangler, ist jedoch ohne weiteres denkbar.

Der Vorgang der Transformation von Programmen wird in zwei Phasen aufgeteilt. In der ersten Phase werden Schnittstellen-Tranformationen (in beliebiger Reihenfolge) solange durchgeführt, bis sich keine Änderungen mehr ergeben. In dieser Phase prüft das Framework, ob die von den Komponenten angeforderten Transformationen zulässig sind (in Hinblick auf Einhaltung der Java Binary Compatibility Spezifikation), entscheidet über die Ausführungsreihenfolge und führt die Transformationen selbst aus.

In Phase zwei werden ausschliesslich Code-Transformationen durchgeführt und zwar genau einmal, in der vom Programmierer vorgegebenen Reihenfolge. Eine Wiederholung findet hier nicht automatisch, sondern nur wie vorgegeben statt.

[1] Diese Transformer-Komponenten sind das Äquivalent zu Aspekten aus anderen Aspekt-Orientierten Systemen.

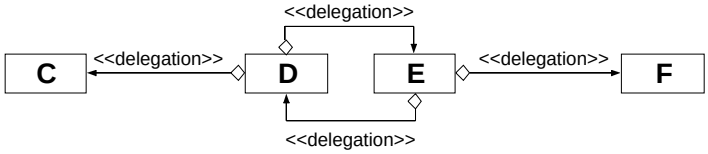

Abbildung 3. Ein Beispiel für eine zyklische Programmstruktur, die eine Iteration von Transformationen erfordert.

Damit eine Transformer-Komponente als Schnittstellentransformer eingesetzt werden kann, muss sie das InterfaceTransformerComponent Interface implementieren. Entsprechend muss sie das CodeTransformerComponent Interface implementieren, um als Codetransformer eingesetzt werden zu können. Eine Transformer-Komponente kann auch beide Interfaces implementieren und somit als Schnittstellentransformer und als Codetransformer eingesetzt werden. Dies hat keinen Einfluss auf die Reihenfolge der Phasen: Die Rolle als Schnittstellentransformer kann ausschließlich in der ersten Phase wahrgenommen werden, die Rolle als Codetransformer nur in der zweiten Phase. Auf diese Weise ist es aber möglich, innerhalb einer Transformer-Komponente Schnittstellentransformationen und Codetransformationen aufeinander abzustimmen.

Die Kompositionskomponente von JMangler, die die Aufteilung der Transformationen in die verschiedenen Phasen steuert, ist in Abbildung 2 zusammen mit den anderen wesentlichen Bestandteile von JMangler dargestellt. Die Information darüber, welche Transformer-Komponenten eingesetzt und in welcher Reihenfolge die Codetransformer ausgeführt werden sollen, erhält die Kompositionskomponente in Form einer Konfigurationsdatei in einem XML Format.

5 Anwendungsbeispiel

JMangler ist im Rahmen des TAILOR Projekts für die Implementierung von LAVA, einer Erweiterung der Programmiersprache Java, eingesetzt worden. Damit die LAVA-spezifischen Spracherweiterungen auch bei fremden, bereits übersetzten Java-Klassen greifen können, übernehmen entsprechende Transformer-Komponenten die notwendigen Modifikationen an deren Bytecode.

Für die Nutzung von JMangler als back-end des LAVA-Compilers ist die Tatsache von entscheidender Bedeutung, dass Interfacetransformationen unabhängig komponiert und nach Bedarf automatisch iteriert werden können. Dies wird an folgendem, stark vereinfachtem Beispiel klar.

Zur Implementation objektbasierter Vererbung erfordert LAVA u. a. die automatische Generierung von lokalen Forwarding-Methoden für alle im deklarierten Typ eines entsprechend markierten Feldes sichtbaren Methoden. Beispielsweise werden in folgender Klasse C Forwarding-Methoden für alle Methoden erzeugt, die in der Schnittstelle der Klasse D enthalten sind, da das Feld d als delegatee deklariert ist.

```
public class C {
   public delegatee D d;

   // wenn D die Methode m enthaelt,
   // bewirkt delegatee, dass hier
   // (in etwa) folgendes erzeugt wird

   // public void m() { d.m(); }
}
```

Eine Transformer-Komponente Forward ist zuständig für die Bestimmung der sichtbaren Methoden des Feldtyps und die Erzeugung entsprechender Forwarding-Methoden in der Klasse, die das Feld enthält.

Je nach Programmstruktur kann es erforderlich sein, diese Komponenten zu iterieren. Ein Beispiel ist in Abbildung 3 skizziert. In diesem Beispiel gibt es Forwarding-Relationen von D zu C, von D zu E, von E zu D und von E zu F. Bei einmaliger Bearbeitung aller Klassen würde Forward zunächst versuchen, Forwarding-Methoden für D zu erzeugen. Dieser Versuch würde aber ein unvollständiges Ergebnis liefern, da das Interface von D noch nicht vollständig bestimmt ist. Dies gilt analog für den Fall, dass zunächst E von Forward bearbeitet wird. Dieses Problem ist folglich nur zu lösen, wenn Forward iterativ und mehrfach auf alle beteiligten Klassen angewendet wird.

JMangler erlaubt solche Transformationen. Da es sich bei Forward um eine reine Interface-Transformation handelt, treten zwischen dieser und weiteren Transformer-Komponenten, die für andere Eigenschaften der Sprache LAVA zuständig sind, keine störenden Interaktionen auf. Daher können alle beteiligten Interfacetransformer-Komponenten ohne implizites Wissen voneinander problemlos gemeinsam eingesetzt werden.

6 Fazit

Der unantizipierte gemeinsame Einsatz von unabhängig entwickelten Aspekten ist ein bisher in Aspekt-Orientierten Systemen nicht zufriedenstellend gelöstes Problem. Die Interaktion unabhängig entwickelter Aspekte muss durch eine Vermittlungsinstanz geregelt werden, wenn man unerwünschte Interaktionen vermeiden will. Diese Tatsache schränkt den Anwendungsbereich aspekt-orientierter Systeme stark ein. Insbesondere im Bereich der Komponenten-Technologie, wo unvorhergesehende Kombinierbarkeit unabhängig entwickelter Komponenten eine zentrale Rolle spielt [4], ist Aspekt-Orientierte Programmierung in der jetzigen Form kaum anwendbar.

In dem hier vorgestellten Konzept für eine Aufteilung von Transformationen auf Schnittstellentransformationen und Codetransformationen gibt es die Möglichkeit, Programmtransformationen von mehreren, unabhängig voneinander entwickelten Transformer-Komponenten bestimmen und automatisch miteinander kombinieren zu lassen. Dabei ist es nicht nötig, die Reihenfolge vorzugeben, in der ein Programm durch die Schnittstellentransformer transformiert wird. Die Kompositionsreihenfolge der Codetransformer muss jedoch von außen vorgegeben werden, da sie das Endergebnis beeinflusst.

Dieses Konzept bildet die Grundlage für JMangler, ein Framework für die Transformation von Java Klassen zur Ladezeit. Eine detaillierte Beschreibung dieses Tools befindet sich in [3].

Ein wesentliches Ergebnis unserer Arbeit ist der Nachweis, dass unabhängige Erweiterbarkeit im Sinne von komponentenorientierter Software für Schnittstellentransformationen realisierbar ist. Dieses Ergebnis kann ohne weiteres auf andere Aspekt-Orientierte Systeme übertragen werden.

Es bleibt die offene Frage, ob für Codetransformationen feinere Kriterien gefunden werden können, anhand derer ein noch höherer Grad an unabhängiger Erweiterbarkeit erreicht werden kann. Zum Einen ist vorstellbar, dass nur eine bestimmte Klasse von Codetransformationen zugelassen werden, so dass auch hier unabhängige Transformationen ermöglicht werden. Es ergibt sich die Fragestellung, wo die Grenze zwischen zulässigen und unzulässigen Codetransformationen gezogen werden soll. Dynamische Aspekte, die durch Datenflussanalyse bestimmt werden, können hierbei eine wichtige Rolle spielen.

Zum Anderen wäre es interessant zu untersuchen, inwiefern es möglich ist, Codetransformationen mit Spezifikationen ihrer Komponierbarkeit zu versehen, die verifizierbar sind und somit automatische Kompositionen ermöglichen würden. Wir würden diese Fragen gern im Rahmen des Workshops diskutieren.

Das TAILOR Projekt wird von Prof. Dr. A. B. Cremers geleitet und von der Deutschen Forschungsgemeinschaft (DFG) unter Projektnummer CR 65/13 finanziert.

Literatur

[1] Aspect-Oriented Programming Home Page. http://www.parc.xerox.com/csl/projects/aop/. 1998-2000.

[2] Ralph Keller, Urs Hölzle. *Binary Component Adaptation*. in: Eric Jul (Ed.). *ECOOP '98 – Object-Oriented Programming*. Conference Proceedings, Springer LNCS 1445, 1998.

[3] Günter Kniesel, Pascal Costanza, and Michael Austermann. *JMangler – A Framework for Load-Time Transformation of Java Class Files*. eingereicht für: *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*.

[4] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[5] The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/. 2000-2001.