Ladezeittransformation von Java-Programmen

Diplomarbeit von

Michael Austermann
Stettiner Straße 30
D-53879 Euskirchen

Email: austerm@cs.uni-bonn.de

Institut für Informatik III

Rheinische Friedrich-Wilhelms-Universität Bonn

Professor Dr. A. B. Cremers

27. Dezember 2000

Erklärung: Hiermit erkläre ich, diese Diplomarbeit selbständig durchgeführt zu haben. Alle Quellen und Hilfsmittel, die ich verwendet habe, sind angegeben. Zitate habe ich als solche kenntlich gemacht.

Euskirchen, 27. Dezember 2000, Michael Austermann

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ladezeittransformation

Die Anforderungen an Software sind einem ständigen Wandel unterworfen. Manche Anforderungsänderungen können von den Softwareentwicklern vorhergesehen werden, sodaß im günstigsten Fall die nötige Anpassung durch eine geänderte Konfiguration erreicht werden kann. Es gibt jedoch in der Praxis immer auch nicht antizipierte Anforderungsänderungen, die notfalls eine Programm-Modifikation erfordern. Eine solche Modifikation kann im wesentlichen zu zwei Zeitpunkten erfolgen, nämlich

- **vor** der Kompilierung, und
- **nach** der Kompilierung.

Eine Anpassung vor der Kompilierung entspricht einer Modifikation des zugrundeliegenden Quelltextes. Dieser steht jedoch nicht in jedem Fall zur Verfügung. Mit zunehmender Verwendung von Softwarebibliotheken, Komponentenarchitekturen und Frameworks, wird in der Regel lediglich eine Lizenz für die Verwendung der Software in kompilierter Form erworben, der Quelltext hingegen ist oft nicht erhältlich, oder darf nicht geändert werden.

Als einzige Alternative bleibt daher lediglich eine Anpassung nach der Kompilierung, also die Transformation des Programmes in einer binären Repräsentation. Software die mit den weit verbreiteten Programmiersprachen wie z.B. C, C++ oder Pascal entwickelt wurde, wird jedoch üblicherweise sofort in Maschinencode übersetzt, der spezifisch für ein bestimmtes Betriebssystem mit darunterliegender Hardware ist. Desweiteren besitzt diese binäre Repräsentation oftmals nicht mehr die Struktur und die symbolischen Informationen des Quelltextes, sodaß mit herkömmlichen Mitteln nur stark begrenzte Anpassungen möglich sind.

Im Gegensatz dazu enthalten übersetzte Java-Programme noch soviel symbolische und strukturelle Information, daß eine weitgehende Rekonstruktion des zugrundeliegenden Java-Quelltextes möglich ist. Es bietet sich an, dieses reichhaltige Angebot an Informationen zu nutzen, um auch nach der Kompilierung noch komplexe Anpassungen vorzunehmen.

Die Transformation einer Java-Anwendung sollte sinnvollerweise alle Klassen der Anwendung erfassen. Es ist jedoch nicht möglich, einfach vor dem Start alle

Anwendungsklassen zu transformieren, da die Menge der Klassen, die die Anwendung bilden, vor der Ausführung im allgemeinen nicht vollständig bestimmt werden kann. Das liegt daran, daß die Klassen erst zur Laufzeit, und erst dann, wenn sie wirklich benötigt werden, nachgeladen und gelinkt werden. Dadurch kann eine Anwendung zur Laufzeit dynamisch um beliebige Klassen erweitert werden.

Transformationen sollten also sinnvollerweise erst beim sukzessiven *Laden* der Programmklassen durchgeführt werden, um sicherzustellen, daß auch alle zur Ausführung des Programmes erforderlichen Klassen durch die Transformation erfaßt werden. Das bietet zusätzlich den Vorteil, daß auch sich kurzfristig ändernde Anforderungen berücksichtigt werden können. So ist es denkbar, daß bei einem Programmlauf spezielle Debugging-Anpassungen vorgenommen werden sollen, und in einem darauf folgenden Lauf Anpassungen für das Sammeln von Profiling-Informationen erforderlich sind.

Ladezeittransformationen sind jedoch bisher nur schwer realisierbar. Da die Java-Umgebung eine Ladezeittransformation nicht vorsieht, muß ein eigener Transformationsmechanismus in die vorhandene Architektur integriert werden. Eine solche Integration erfordert ein tiefgehendes Wissen über die zugrundeliegende Architektur, und ist nicht ohne einen erheblichen Aufwand durchzuführen. Vorhandene Lösungsansätze sind stark eingeschränkt, da z.B. Änderungen an Implementationen der Java Virtual Machine (JVM) vorgenommen wurden, oder Transformationen auf einzelne Klassen beschränkt bleiben (mehr dazu in Kapitel 7).

1.2 Ziel der Diplomarbeit

Das Ziel der Diplomarbeit ist die Entwicklung eines Frameworks für die Ladezeittransformation von Java-Programmen, in dem die Programmklassen nicht nur von einer, sondern von vielen verschiedenen Instanzen, im folgenden *Transformer-Komponenten* genannt, transformiert werden. Die Entwicklung und Komposition von Transformer-Komponenten soll möglichst einfach sein.

Dieses Transformationsmodell soll in Java [GJSB00] implementiert und in das JDK1.3 [SUN00a] integriert werden, ohne dabei eine Implementation der Java Virtual Machine zu modifizieren. Abschließend soll das so entstandene Framework daraufhin untersucht werden, welche Zusatzkosten durch die Transformation entstehen.

1.3 Gliederung

Die vorliegende Diplomarbeit ist wie folgt gegliedert:

Kapitel 2 skizziert die Java-Technologie. Einige für diese Arbeit wichtige Themen wie Sicherheit, Class Loader System und Classfile Format werden darin eingehender beschrieben.

Kapitel 3 beschreibt die wesentlichen Anforderungen an das vorliegende Framework für die Ladezeittransformation von Java-Programmen. Anschließend folgt ein Überblick über die Architektur und die Ausarbeitung des zugrundeliegenden Konzepts.

Kapitel 4 beinhaltet die formale Ausarbeitung des in Kapitel 3 beschriebenen Konzepts.

Kapitel 5 beschreibt das verwendete Design für eine Implementation der Architektur aus Kapitel 3 und des formalen Modells aus Kapitel 4. Anschließend werden nicht-offensichtliche, aber wichtige technische Probleme bei der Umsetzung des Designs erläutert.

Kapitel 6 ermittelt, welche Zusatzkosten durch die Ladezeittransformation entstehen und gibt einen Ausblick auf mögliche Erweiterungen der verwendeten Implementation.

Kapitel 7 vergleicht das in dieser Arbeit entwickelte Konzept zur Ladezeittransformation von Java-Programmen mit anderen Arbeiten aus diesem Forschungsbereich.

Kapitel 8 faßt schließlich knapp zusammen, welche Ergebnisse in dieser Arbeit erzielt wurden.

1.4 Danksagungen

Ich danke Herrn Professor Dr. A. B. Cremers für die Betreuung meiner Diplomarbeit. Ein großer Dank geht an Pascal Costanza und Dr. Günter Kniesel für die vielen Stunden, die wir mit der Diskussion von verschiedenen Ansätzen und der Besprechung von Zwischenergebnissen verbracht haben.

Ebenfalls bedanken möchte ich mich bei Angela Schumacher und Alistair Max Bleeck für die gewissenhafte Korrektur dieser Arbeit.

Teil I

Grundlagen

Kapitel 2

Die Java-Technologie

Das folgende Kapitel gibt einen Überblick über die Java-Technologie [SUN00a]. Eine ausführliche Einführung in die Programmiersprache Java und die Java API geben *The Java Programming Language* [AG97] und das *Java Tutorial* [MC98]. Vertiefende Informationen zum Thema Java Classfile Format und Java Virtual Machine sind in *Inside the JAVA 2 Virtual Machine* [Ven99] enthalten.

2.1 Überblick

Die Java-Technologie besteht aus vier verschiedenen, eng miteinander verbundenen Technologien:

- die Java-Programmiersprache,
- die Java Virtual Machine,
- das Java Classfile Format,
- die Java API.

Java-Programme werden in der *Java-Programmiersprache* ausgedrückt, von einem *Java-Kompiler* in das *Java Classfile Format* übersetzt und auf einer *Java Virtual Machine* (JVM) ausgeführt. Zugriffe auf Systemressourcen wie z.B. E/A-Zugriffe werden über Klassen aus der *Java API* realisiert. Die JVM und die Java API bilden zusammen die sogenannte *Java-Plattform*. Java-Programme können auf vielen verschiedenen Plattformen ausgeführt werden, da die Java-Plattform selber ebenfalls in Software implementiert werden kann. Zur Zeit existieren Implementationen der Java-Plattform für viele verschiedene Betriebssysteme, wie z.B. Win32, Solaris, Linux, Macintosh. So wie eine in Hardware realisierte Maschine eine Menge von Maschinenbefehlen hat, so besitzt auch die JVM Maschinenbefehle, die auch *Opcodes* genannt werden. Der in der Java-Programmiersprache ausgedrückte Code einer Methode wird vom Java-Kompiler in die entsprechende Sequenz von Opcodes übersetzt. In diesem Zusammenhang spricht man dann vom *Java-Bytecode* oder kurz *Bytecode*.

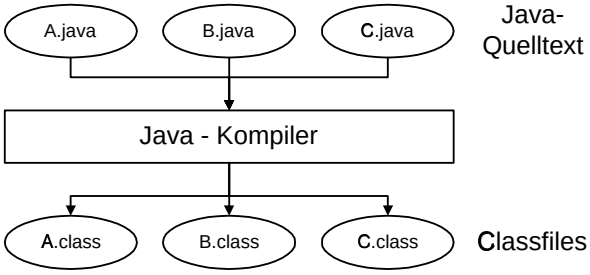

Abbildung 2.1: Java-Quelltext wird in das Classfile Format übersetzt. Dabei erzeugt ein Java-Kompiler pro Klasse ein Classfile.

Java-Programmiersprache Die Java-Programmiersprache ist eine *objektorientierte, architekturneutrale Hochsprache*. Sie ist *typsicher* und besitzt Mechanismen, die einen Einsatz in *verteilten Systemen* ermöglichen. Programme werden ausschließlich in Form von *Klassen* und *Schnittstellen* definiert. Eine Schnittstelle deklariert eine Menge von Methodensignaturen. Eine Klasse deklariert eine Menge von *Feldern* und *Methoden*. Java unterstützt nur *einfache Vererbung*, d.h. jede Klasse besitzt genau eine *direkte Oberklasse*, allerdings kann eine Klasse mehrere Schnittstellen implementieren. Einzige Ausnahme ist die Klasse `java.lang.Object`, die keine Oberklasse besitzt und die *Wurzel der Klassenhierarchie* darstellt. Die genaue Spezifikation der Java Programmiersprache wird in [GJSB00] beschrieben.

Java Virtual Machine Die *Java Virtual Machine* (JVM) ist eine abstrakte Maschine. Wie reale Rechenmaschinen, besitzt auch die JVM eine Menge von Maschinenbefehlen und verschiedene Speicherbereiche während der Laufzeit. Ihre Spezifikation [TL99] definiert bestimmte Eigenschaften, die jede JVM haben muß, läßt den Designern einer JVM-Implementation jedoch auch diverse Auswahlmöglichkeiten. So ist es freigestellt, ob die JVM in Software oder in Hardware implementiert wird. Bei einer Software-Implementation bleibt es den Designern z.B. überlassen, ob der Bytecode interpretiert, oder von einem *Just in Time Compiler* in plattformabhängigen Code übersetzt wird, der dann direkt ausgeführt, und für eine spätere Wiederverwendung in einem Cache vorgehalten wird. Diese flexible Spezifikation erlaubt es, daß die JVM auf vielen verschiedenen Plattformen implementiert werden kann.

Classfile Format Java-Programme werden von einem Java-Kompiler in das sogenannte *Classfile Format* übersetzt. Dieses Format dient als binäre Repräsentation für Java-Programme und wird von allen JVMs erwartet, d.h. es ist unabhängig von der Plattform, auf der die JVM ausgeführt wird. Der Quelltext wird also nicht in eine ausführbare Datei übersetzt, die spezifisch für ein bestimmtes Zielbetriebssystem mit der darunterliegenden Hardware ist.

Von einem Kompiler wird pro Klasse bzw. Schnittstelle genau ein sogenanntes *Classfile* erzeugt (siehe Abbildung 2.1). Da das Objekt-Layout eines Java-Programmes erst zur Laufzeit innerhalb der JVM festgelegt wird, sind die Referenzen auf Klassen, Felder und Methoden innerhalb der Classfiles nur

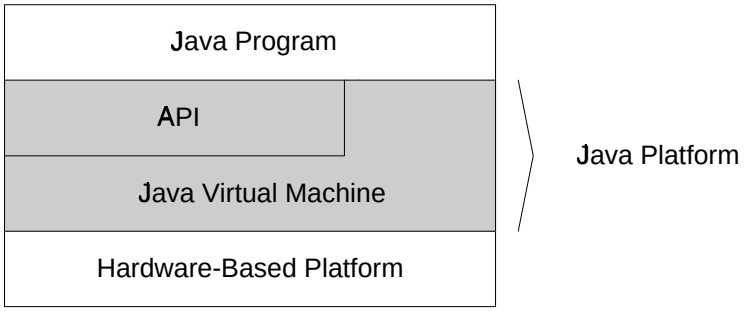

Abbildung 2.2: Die Java-Plattform kapselt die zugrundeliegende Hardware und bietet einem Java-Programm eine einheitliche API (aus [MC98]).

symbolisch. Die Menge an symbolischer Information in einem Classfile ist dabei sogar so groß, daß in den meisten Fällen von sogenannten *Decompilers* ein Classfile zurück in den zugrundeliegenden Java-Quelltext übersetzt werden kann. Da Java-Programme erst zur Laufzeit gebunden werden, und Referenzen immer nur symbolisch sind, wird somit das *Syntactic Fragile Baseclass Problem* [MS98] gelöst. Abschnitt 2.2 geht genauer auf das Classfile Format ein.

Java API Die Java API [Jav] ist eine große Sammlung von fertigen Softwarebibliotheken, die viele nützliche Funktionen bereitstellen. Es werden z.B. Klassen für den Zugriff auf Dateien und zum Aufbau von graphischen Benutzerschnittstellen angeboten. Die Java API ist in *packages* gruppiert, die thematisch verwandte Klassen zu einer Einheit bündeln, so wie dies auch für eigene Bibliotheken oder Programme möglich ist. Abbildung 2.2 zeigt ein Java-Programm, wie z.B. eine *Applikation* oder ein *Applet*, das auf der Java-Plattform ausgeführt wird. Wie aus der Abbildung hervorgeht, isolieren die Java API und die Virtual Machine das Java-Programm vor der zugrundeliegenden Hardware. Zugriffe auf die zugrundeliegende Plattform sind nur über wohldefinierte Schnittstellen (JNI, [SUN97]) möglich. Die Java API ist fester Bestandteil der Java-Plattform. In ihrer Spezifikation werden die sogenannten *Core APIs* festgelegt, die auf jeder Plattform vorhanden sein müssen.

2.2 Classfile Format

Das *Classfile Format* ist ein präzise definiertes, binäres, hardware- und betriebssystemunabhängiges Format für Java-Programme. Seine vollständige Spezifikation ist in der *Java Virtual Machine Specification* [TL99] enthalten. Die Repräsentation einer individuellen Klasse wird als *Classfile* bezeichnet. Sie ist mit einer *Objektdatei* vergleichbar, wie sie z.B. von einem C++ Kompiler erzeugt wird. Jedes Classfile enthält die Definition einer einzigen Klasse oder Schnittstelle. Die Spezifikation des Classfile Formats stellt sicher, daß ein Classfile von einer beliebigen JVM-Implementation gelesen und interpretiert werden kann, unabhängig davon, auf welchem System es erzeugt wurde, und auf welchem System die lesende JVM ausgeführt wird.

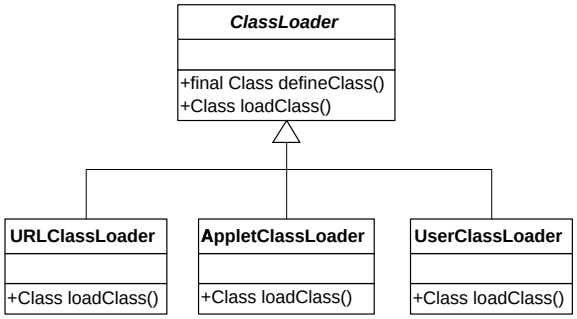

Abbildung 2.3: Class Loaders müssen Unterklasse der Klasse java.lang.ClassLoader sein.

Ein Classfile kann sowohl *Bytecode* als auch *symbolische Referenzen* auf Felder, Methoden und die Namen anderer Klassen enthalten. Ein Beispiel sei die Klasse `C`, die wie folgt deklariert ist:

```
class C {
  void f() {
    D d = new D();
    ...
  }
}
```

Das Classfile das `C` repräsentiert, enthält eine symbolische Referenz auf die Klasse `D`. Solche symbolischen Referenzen werden zur *Link Zeit* (der Klasse `C`) aufgelöst, also durch eine Referenz auf den entsprechenden Klassentyp ersetzt. Dazu muß die JVM das Classfile der Klasse `D` laden und daraus den Klassentyp erzeugen.

Obwohl das Classfile Format eng mit der Sprache Java verbunden ist, wird es nicht ausschließlich in diesem Kontext verwendet. So existieren z.B. Kompiler, die andere Sprachen in das Classfile Format übersetzen können [Tol00]. Desweiteren ist es möglich, gültige Classfiles zu konstruieren, die kein entsprechendes Pendant in der Java-Sprache besitzen.

Weitere Informationen zum Classfile Format geben [Ven99] und [TL99].

2.3 Class Loader System

Die Java-Technologie erfüllt die Bedürfnisse des Internets, da Java-Komponenten und ganze Java-Programme sowohl von der Festplatte, als auch von verschiedenen Servern aus dem Internet auf den lokalen Rechner geladen und dort ausgeführt werden können. Dabei müssen für die Kommunikation mit anderen Rechnern eine Fülle von verschiedenen Netzwerkprotokollen beherrscht werden. Zusätzlich zu den Klassen, die von der lokalen Festplatte oder aus dem Internet geladen werden, existieren auch Java-Lösungen, die Teile ihrer Klassen aus einer Datenbank beziehen oder sogar erst zur Laufzeit erzeugen.

2.3.1 Class Loader

Es stellt sich nun die Frage, wie trotz dieser Fülle von denkbaren Quellen, die JVM immer genau die richtigen Java-Klassen, bzw. deren Classfiles findet und lädt. Verantwortlich für diese Aufgabe ist das *Class Loader System* der JVM. Es besteht aus einer Menge von *Class Loaders*, die sich die Aufgabe des Ladens von Klassen teilen.

Class Loaders sind (bis auf eine Ausnahme) ganz normale Java-Objekte und ermöglichen einem Softwareentwickler, seine Anwendungsklassen auf jede in Java ausdrückbare Art und Weise, z.B. über ein Netzwerk oder vom lokalen Dateisystem, zu laden. Class Loaders sind immer Unterklassen der Systemklasse `java.lang.ClassLoader` (siehe Abbildung 2.3 auf der vorherigen Seite), und es gibt keine Möglichkeit, Klassen anders als über die Class Loaders in das System zu laden.

Class Loaders bekommen, in der Regel von der JVM, die Aufforderung, im System noch unbekannte Klassen zu laden. Dazu wird die von `ClassLoader` geerbte Methode `loadClass` mit dem Namen der zu ladenden Klasse als Parameter aufgerufen. Daraufhin versucht der Class Loader, die angeforderte Klasse auf die in ihm implementierte Art und Weise zu laden, und legt sie in einem Byte-Array im Classfile Format auf dem Heap ab. Anschließend übergibt er eine Referenz auf dieses Byte-Array an die JVM, indem er eine der Varianten der nicht redefinierbaren, von `ClassLoader` geerbten Methoden `defineClass` aufruft. Bei diesem Aufruf wird die plattformunabhängige Darstellung der Klasse im Byte-Array in die plattformabhängige Darstellung der Klasse innerhalb der JVM überführt. Ab diesem Zeitpunkt ist die Klasse im System und der Class Loader hat seine Aufgabe erfüllt.

2.3.2 Bootstrap Class Loader

Da Class Loaders Java-Klassen laden, jedoch ebenfalls Instanzen von Java-Klassen sind, stellt sich die Frage, wer den ersten Class Loader bzw. dessen Klasse lädt. Diese Aufgabe wird vom *Bootstrap Class Loader* erledigt. Er ist Teil der JVM und wird in der Regel in derselben Sprache wie die JVM (also z.B. in C) implementiert. Der Name Bootstrap Class Loader rührt daher, daß dieser Class Loader die Klassen lädt, die für das *Bootstrapping* der JVM notwendig sind.

2.3.3 Parent Delegation Model

Wie bereits erwähnt, teilen sich die Class Loaders im Class Loader System ihre Aufgabe, Klassen zu laden. Bis einschließlich JDK 1.1 gibt es kein einheitliches Modell, wie diese Arbeitsteilung aussehen soll. Daher müssen die Entwickler von Class Loaders selber einen entsprechenden Algorithmus innerhalb der Methode `loadClass` implementieren. Ab Java 2 (JDK 1.2) wird für die Arbeitsteilung das *Parent Delegation Model* vorgeschlagen, welches bereits in der Klasse `ClassLoader` realisiert ist.

Beim Parent Delegation Model besitzt jeder Class Loader eine Referenz auf seinen Vater (Parent). Die `null`-Referenz bezeichnet den Bootstrap Class Loader, der als einziger keinen Vater besitzt und die Wurzel des Class Loader-Baumes darstellt (siehe Abbildung 2.4 auf der nächsten Seite). Der Vater wird

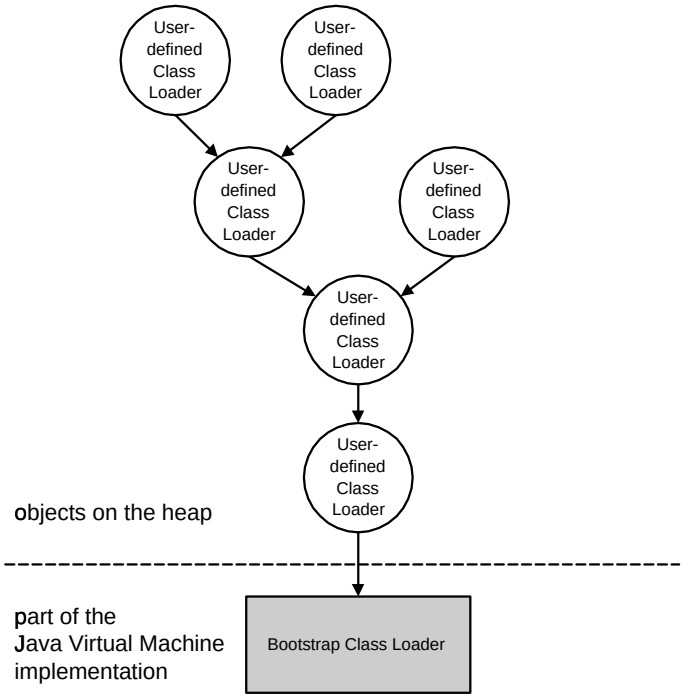

Abbildung 2.4: Das *Parent Delegation Model* des Class Loader Systems (nach [Ven99]). Jeder Class Loader besitzt eine Referenz auf seinen Vater. Die Anfrage eine Klasse zu laden wird immer erst an den Vater delegiert. Nur wenn dieser die Klasse nicht laden kann, versucht der Sohn die Klasse selber zu laden.

bei der Instanziierung eines Class Loaders festgelegt und kann später nicht mehr geändert werden. Wird bei der Instanziierung kein Vater angegeben, so wird automatisch der Class Loader als Vater zugewiesen, der die laufende Applikation geladen hat. Beim Parent Delegation Model wird die Anfrage, eine Klasse zu laden, immer zuerst an den Vater weiterdelegiert. Nur wenn dieser die entsprechende Klasse nicht laden konnte, versucht der Sohn, die Klasse selber zu laden. Abbildung 2.5 auf der nächsten Seite zeigt die Implementation dieses Algorithmus in der Klasse `ClassLoader`.

Bekommt ein Class Loader eine Anfrage für die Klasse `name`, so delegiert er diese Anfrage zuerst per `parent.loadClass(name)` an seinen Vater weiter. Ist die Referenz auf den Vater `null`, so kann der Bootstrap Class Loader mit der nativen Methode `findBootstrapClass0` aufgerufen werden. War der Vater in der Lage, die Klasse zu laden und an die JVM zu übergeben, so gibt der Sohn die von seinem Vater erhaltene Referenz auf die angeforderte Klasse an den Aufrufenden zurück. War sein Vater nicht in der Lage, die Klasse zu laden, so löst dieser eine `ClassNotFoundException` aus, die vom Sohn aufgefangen wird. Daraufhin ruft dieser die in `ClassLoader` deklarierte Methode `findClass` auf. Diese löst standardmäßig ebenfalls eine `ClassNotFoundException` aus. Soll ein eigener Class Loader entwickelt werden, wird typischerweise eine Unterklasse von `ClassLoader` gebildet, und die Methode `findClass` so überschrieben, daß

```
protected synchronized Class loadClass(String name, boolean resolve)
throws ClassNotFoundException
{
  ...
  try {
    if (parent != null) {
      c = parent.loadClass(name, false);
    }
    else {
      c = findBootstrapClass0(name);
    }
  } catch (ClassNotFoundException e) {
    // If still not found, then call findClass in order
    // to find the class.
    c = findClass(name);
  }
  ...
}
```

Abbildung 2.5: Implementation des *Parent Delegation Models* in der Klasse ClassLoader (aus [SUN00a])

sie Klassen auf die erwünschte Art und Weise lädt.

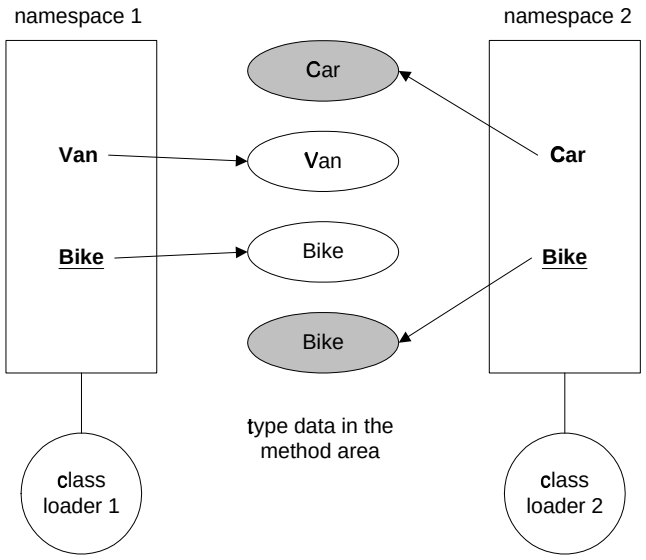

Abbildung 2.6: Durch die Verwendung mehrerer Class Loaders können verschiedene *Namensräume* entstehen (nach [Ven99]). Dadurch ist es möglich, daß sich zwei Klassen mit demselben Namen im System befinden, solange sie in verschiedenen Namensräumen lokalisiert sind.

2.3.4 Namensräume

Jede Klasse, die sich in der JVM befindet, besitzt ihren *definierenden Class Loader*. Das ist der Class Loader, der die Klasse geladen und per `defineClass` an die JVM übergeben hat. Zwei Typen in der laufenden JVM sind genau dann gleich, wenn ihre Bezeichner gleich sind und sie vom selben Class Loader geladen wurden. Dadurch entstehen in der JVM verschiedene *Namensräume* (Namespaces). Jeder Class Loader definiert genau einen Namensraum, in dem sich die Klassen befinden, die er geladen hat. Dadurch ist es möglich, daß sich zwei unterschiedliche Klassen, die denselben Namen besitzen, gleichzeitig in der JVM befinden, wenn sie sich in verschiedenen Namensräumen aufhalten. Abbildung 2.6 zeigt ein Beispiel.

Die Namensräume der Class Loaders ermöglichen somit, mehrere Applikationen gleichzeitig in einer laufenden JVM auszuführen, ohne daß die Klassen der einen Applikation für die Klassen der anderen Applikation *sichtbar* sind. Man betrachte dazu Abbildung 2.7 auf der nächsten Seite. Der Application Class Loader hat einen *Appletviewer*[1] geladen, der nur aus der Klasse `Appletviewer` besteht. Der Appletviewer wiederum hat für jedes *Applet*[2], das er lädt, einen eigenen Class Loader instanziiert. Die Applets `Alpha` und `Beta` befinden sich gleichzeitig im System. Die Klassen von `Alpha` sind jedoch für `Beta` nicht sichtbar und umgekehrt. Dadurch wird zum einen sichergestellt, daß kein Applet Einfluß auf ein anderes nehmen kann, und zum anderen ist es so unmöglich,

[1] Ein Appletviewer ist eine Applikation, die Applets laden und ausführen kann.

[2] Ein Applet ist ein Programm, das in der Regel nicht aus dem lokalen System stammt und zur Ausführung aus dem Netz geladen werden muß. (aus [Gon99a], §2.2)

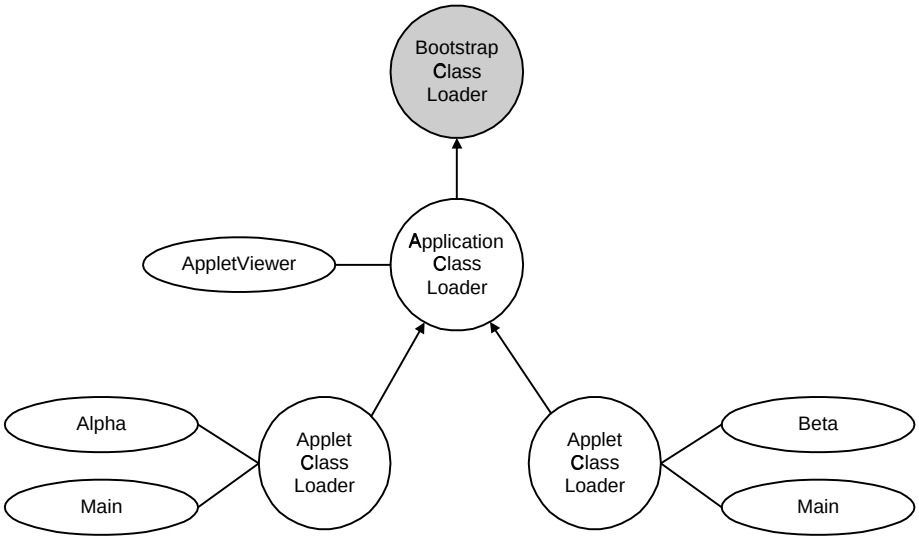

Abbildung 2.7: Aufgrund der aus den Namensräumen resultierenden *Sichtbarkeit* zwischen Klassen, ist die Klasse Beta für die Klasse Alpha nicht sichtbar (und umgekehrt).

daß Konflikte durch gleichnamige Klassen (in diesem Beispiel die Klasse `Main`) entstehen.

Für weitere Informationen zum Thema Class Loaders siehe [Gon99b] und [Ven99], §3.

2.4 Sicherheitsarchitektur

Der folgende Abschnitt gibt einen kurzen Überblick über die Sicherheitsarchitektur von Java. Anschließend wird der *Security Manager*, der ein Teil des gesamten Sicherheitsmodells ist, eingehender beschrieben. Ausführliche Informationen zum Thema Sicherheit sind in [Gon99a] enthalten.

2.4.1 Überblick

Die grundlegende Sicherheitsarchitektur von Java ist darauf ausgelegt, es einem Benutzer zu erlauben, beliebige *Applets* zu laden und auf seinem Rechner auszuführen, ohne daß dabei Risiken für sein lokales System entstehen. Applets werden dynamisch geladen, und das zum Teil, ohne daß sich der Benutzer dessen bewußt ist. Da man in der Regel nicht genau wissen kann, wer der Autor eines Applets ist, kann man nicht blind darauf vertrauen, daß ein Applet nicht versucht, dem lokalen System Schaden zuzufügen. Daher sind alle Aktionen eines Applets auf seine *Sandbox* beschränkt, einem Bereich der innerhalb der JVM speziell diesem Applet zugewiesen ist. Ein Applet darf nur innerhalb seiner Sandbox „spielen“, nicht außerhalb. So ist es z.B. einem Applet in der Regel verboten, auf das lokale Dateisystem lesend oder schreibend zuzugreifen. Die Sicherheitsarchitektur besteht aus mehreren Komponenten:

```
public FileInputStream(String filename) throws FileNotFoundException {
  SecurityManager security = System.getSecurityManager();
  if (security != null) {
    security.checkRead(filename);
  }
  // Hier die gewünschte Datei zum Lesen öffnen.
  ...
}
```

Abbildung 2.8: Zugriffsbeschränkung für das Lesen vom lokalen Dateisystem durch die Verwendung des *Security Managers*

Java-Programmiersprache Die Java-Programmiersprache [GJSB00] ist so entworfen worden, daß sie *typsicher* ist. *Automatische Speicherverwaltung*, *Garbage Collection* ([GJSB00], §12) und *Range Checking* ([GJSB00] §10) beim Zugriff auf Strings und Arrays sind einige Beispiele dafür, wie die Programmiersprache Java einen Entwickler dabei unterstützt, sicheren Code zu schreiben.

Bytecode Verifier Der *Bytecode Verifier* ([TL99], §4.9) stellt sicher, daß nur legitimer Java-Code ausgeführt wird.

Vor der Ausführung eines neu geladenen Java Applets überprüft der Bytecode Verifier, ob es der *Java Virtual Machine Specification* [TL99] entspricht. Darüber hinaus überprüft er, ob die *Speicherverwaltung* verletzt wird, ob *Stack Overflows* oder *Underflows* möglich sind, oder ob illegale *Typecasts* verwendet werden. Solche Aktionen könnten einem Applet ermöglichen, bestimmte Sicherheitsmechanismen zu umgehen. Zusammen mit der JVM gewährleistet der Verifier die Typsicherheit während der Laufzeit.

Namensräume Die bereits in Abschnitt 2.3 beschriebenen *Class Loaders* verhindern mit der Definition von verschiedenen *Namensräumen*, daß Applets unerlaubten Einfluß auf andere in der JVM laufende Applikationen nehmen.

Security Manager Der *Security Manager* ist eine Klasse, die es Applikationen erlaubt eine *Sicherheitspolitik* (security policy) zu implementieren. Dazu muß die Applikation vor der Ausführung einer möglicherweise unsicheren oder sensiblen Operation beim Security Manager überprüfen, ob die Operation im aktuellen Kontext durchgeführt werden darf. Der Security Manager gibt z.B. einem Applet die Grenzen seiner Sandbox vor, indem er den Zugriff auf sensible Ressourcen verweigert. Der folgende Abschnitt geht ausführlicher auf diese Komponente der Java-Sicherheitsarchitektur ein.

2.4.2 Security Manager

Wird ein Applet auf einem Rechner ausgeführt, so soll diesem Applet in der Regel der Zugriff auf sensible Ressourcen verweigert werden. So wird z.B. in der klassischen Sandbox einem Applet der Zugriff auf das lokale Dateisystem und der Aufbau einer Netzverbindung zu einem anderen als dem eigenen Server verwehrt. Diese *Sicherheitspolitik* wird vom *Security Manager* implementiert. Dazu

wird in den zu schützenden Methoden vor der eigentlichen sensiblen Operation der Security Manager angewiesen, zu überprüfen, ob der *Aufrufer* der Methode laut der aktuellen Sicherheitspolitik das Recht für ihre Ausführung besitzt. Ist kein Security Manager installiert, so findet keine Überprüfung statt, d.h. es existieren keine Zugriffsbeschränkungen.

Funktionsweise Im System existiert nur ein *aktiver* Security Manager. Er muß Instanz der Klasse `SecurityManager` (oder einer Unterklasse) sein, und kann bei der Klasse `java.lang.System` mit den Methoden `setSecurityManager` und `getSecurityManager` gesetzt bzw. ermittelt werden (falls der Aufrufer das Recht dazu besitzt). Ist kein Security Manager installiert, so liefert `System.getSecurityManager()` `null` zurück. Abbildung 2.8 auf der vorherigen Seite zeigt ein Beispiel für die typische Verwendung des Security Managers, bei dem der lesende Zugriff auf das lokale Dateisystem geschützt wird.

Ist ein Security Manager installiert, so kann mit einer seiner Methoden überprüft werden, ob der Aufrufer berechtigt ist, eine Operation auszuführen. Dazu besitzt die Klasse `SecurityManager` die sogenannten `check`-Methoden, die alle mit dem Präfix `check` beginnen, wie z.B. `checkRead` in Abbildung 2.8 auf der vorherigen Seite, und jeweils das Zugriffsrecht auf eine bestimmte Ressource überprüfen. Ist der Zugriff auf eine Ressource erlaubt, so kehrt der Kontrollfluß normal aus der entsprechenden `check`-Methode zurück. Andernfalls löst der Security Manager eine `SecurityException` aus, sodaß der normale Kontrollfluß unterbrochen wird, und dadurch der Zugriff auf die Ressource nicht möglich ist.

Um eine Zugriffskontrolle zu erreichen, müssen also die folgenden zwei Punkte berücksichtigt werden:

- Beim Start der JVM muß ein Security Manager installiert werden, der die gewünschte Sicherheitspolitik implementiert.
- Alle Klassen die sensible Ressourcen verwalten, müssen ihre Methoden so implementieren, daß vor der eigentlichen kritischen Operation eine Überprüfung durch den Security Manager erfolgt.

Installation Im JDK Version 1.0 und 1.1 ist die Klasse `SecurityManager` abstrakt, d.h. jeder, der einen Security Manager verwenden will, muß eine eigene Unterklasse von `SecurityManager` bilden und die gewünschte Sicherheitspolitik entsprechend in Java implementieren. Diese Vorgehensweise ist aufwendig und fehleranfällig, da bei einer versehentlich fehlerhaft erfolgten Implementation eines Security Managers offensichtlich unerwünschte Sicherheitslücken entstehen können.

Seit dem JDK Version 1.2 ist die Klasse `SecurityManager` nicht mehr abstrakt, und es gibt die Möglichkeit, beim Start der JVM einen einfach konfigurierbaren Security Manager zu installieren. Dazu übergibt man der JVM beim Start ein sogenanntes *Policy File*. In dieser ASCII-Datei wird definiert, welche Klassen welche Rechte besitzen sollen. Dazu definiert man eine Menge von Zugriffsrechten pro *Codebase*. Eine Codebase ist eine URL, und alle Klassen, die von dieser URL geladen werden, bekommen die angegebenen Rechte. Das Programm `policytool`, ein mit dem JDK ab Version 1.2 [SUN00a] ausgeliefertes Hilfsprogramm, vereinfacht die Erstellung einer solchen Datei.

Abbildung 2.9 zeigt das Policy File `sample.policy`, bei dem alle Klassen, die vom Server `www.cs3.org` aus dem Verzeichnis `classes` stammen, alle lokalen Dateien lesen und die Ausführung der JVM beenden dürfen. Klassen aus dem lokalen Verzeichnis `/home/austerm/` dürfen alle lokalen Dateien lesen, schreiben, löschen und ausführen.

```
/* sample.policy */
grant codeBase "http://www.cs3.org/classes/" {
  permission java.io.FilePermission "<<ALL FILES>>", "read";
  permission java.lang.RuntimePermission "exitVM";
};

grant codeBase "file:///home/austerm/" {
  permission java.io.FilePermission "<<ALL FILES>>", "read,
    write, delete, execute";
};
```

Abbildung 2.9: Ein *Policy File*

Um die in `sample.policy` definierte Sicherheitspolitik zu installieren, wird der JVM beim Start über Kommandozeilenparameter mitgeteilt, daß ein Security Manager installiert werden soll und in welchem Policy File seine Sicherheitspolitik definiert ist:

```
java -Djava.security.manager
     -Djava.security.policy=sample.policy <class>
```

Weitere Informationen zum Thema Security Manager enthalten [Gon99a] und [Ven99].

2.5 Java Reflection API

Die *Java Reflection API* repräsentiert bzw. reflektiert die Klassen und Schnittstellen in der laufenden Java Virtual Machine als Objekte, auf die innerhalb eines Java-Programmes zugegriffen werden kann. Mit der Reflection API kann man

- die Klasse eines Objektes ermitteln,
- Informationen über die Annotationen, Felder, Methoden, Konstruktoren und die Oberklassen einer Klasse ermitteln,
- herausfinden, welche Konstanten und Methodendeklarationen zu einer Schnittstelle gehören,
- eine Klasse instanziieren, deren Name erst zur Laufzeit bestimmt wird,
- das Feld eines Objektes manipulieren, auch wenn der Name des Feldes erst zur Laufzeit bestimmt wird,
- die Methode eines Objektes ausführen, sogar dann, wenn die Methode erst zur Laufzeit feststeht,
- ein neues Array erzeugen, dessen Typ nicht vor der Laufzeit feststeht.

Die Reflection API macht die Java-Plattform zusammen mit dem Link-Modell zu einer sehr dynamischen Laufzeitumgebung. Mittels der Class Loaders können zur Laufzeit Klassen zu einem Programm hinzugeladen werden, die zum Zeitpunkt der Kompilierung des Programmes vielleicht noch gar nicht existierten, und durch die Reflection API können diese Klassen instanziiert und die so erzeugten Objekte manipuliert werden.

Aufgrund der Reflection API läßt sich vor der Ausführung eines Programmes im allgemeinen nicht durch eine *statische Analyse* vollständig bestimmen, welche Klassen, Methoden und Felder im Laufe der Ausführung des Programmes verwendet werden.

Weitere Informationen zum Thema Reflection API befinden sich in [MC98] und [JDK00].

2.6 Binary Compatibility

Binary Die *Java Language Specification* [GJSB00] fordert, daß Klassen und Schnittstellen von einem Java-Kompiler in das Classfile Format, oder eine Repräsentation, die von einem Class Loader auf das Classfile Format abgebildet werden kann, übersetzt werden müssen. Diese kompilierte Repräsentation einer Klasse bzw. einer Schnittstelle wird als *Binary* bezeichnet. Jede Klasse bzw. jede Schittstelle muß in ein eigenes Binary übersetzt werden. Die Referenzen auf andere Klassen, Methoden und Felder sind innerhalb der Binaries immer nur symbolisch. Das wohl bekannteste Format für ein Binary ist das in 2.2 beschriebene *Classfile Format* selber, denn es erfüllt alle geforderten Eigenschaften. Für weitere Details zum Thema Java-Binaries siehe [GJSB00], §13.

Binary Compatibility Entwickler von Paketen und Klassen, die in einem weit verteilten System Verwendung finden, werden unter anderem mit dem folgenden Problem konfrontiert. In einem weit verteilten System, wie z.B. dem Internet, ist es oft unmöglich, automatisch alle existierenden Binaries zu rekompilieren, die direkt oder indirekt von einer zu ändernden Klasse abhängen. Um diesem Umstand gerecht zu werden, wird in der Spezifikation der *Binary Compatibility* eine Menge von Änderungen definiert, die an einem Paket oder einer Klasse vorgenommen werden können, ohne die Kompatibilität zu existierenden Binaries zu verletzen. Dazu macht die Spezifikation eine Aussage darüber, welche Transformationen an einem Programm bzw. an einem Typ vorgenommen werden können, sodaß für Binaries, die vor der Transformation erfolgreich gelinkt werden konnten, dies auch nach der Transformation ohne Fehler möglich ist. Beispiele für solche Transformationen sind:

- Modifikation der Implementation einer Methode, um ihre Performance zu erhöhen,
- Modifikation der Implementation einer Methode dahingehend, daß sie einen Wert für solche Parameter zurückgibt, bei denen vorher eine Ausnahme ausgelöst wurde, oder die zu einer Endlosschleife führten,
- Hinzufügen von neuen Feldern und Methoden zu existierenden Klassen,
- Einfügen eines neuen Klassen- oder Schnittstellentyps in die Typhierarchie.

Für weitere Informationen zur Binary Compatibility siehe [GJSB00], §13.

Teil II

Framework

Kapitel 3

Konzept

Im folgenden Kapitel wird das Ziel dieser Arbeit detailliert beschrieben und das Konzept für dessen Realisierung vorgestellt. Dieses Konzept bildet die Grundlage für die folgenden Kapitel 4 und 5, die die formale Ausarbeitung, das Design und die Implementation des Transformations-Frameworks beschreiben.

3.1 Konzeptuelle Anforderungen

Unter der *Ladezeittransformation* von Java-Klassen versteht man, daß bei der Ausführung eines Java-Programmes, die Programmklassen nach dem Ladevorgang und vor der Übergabe an die JVM transformiert werden. Wie die Klassen transformiert werden, ist von der jeweiligen Verwendung bzw. dem jeweiligen Ziel abhängig. Alle bestehenden Vorschläge zur Ladezeittransformation von Java-Klassen, wie z.B. BCA [KH98] und Javassist [Chi00], verfolgen jedoch immer den Ansatz, daß genau *eine* zentrale Instanz den Vorgang der Transformation der Klassen bestimmt. Desweiteren beschränkt sich der Fokus der Transformation immer nur auf *eine* Klasse. Das hat zur Folge, daß zwei Klassen nicht in *gegenseitiger* Abhängigkeit voneinander transformiert werden können.

Im Gegensatz dazu wird in dieser Arbeit ein *Multi-Transformer-Modell* für die Ladezeittransformation von Java-Klassen vorgeschlagen, welches die folgenden Punkte berücksichtigt:

- Multiple Transformer,
- Unabhängig entwickelte Transformer,
- Anwendung der Transformer auf Mengen von Klassen, und dadurch
- Transformation unter Berücksichtigung von gegenseitigen Abhängigkeiten der Klassen.

Multiple Transformer Die Klassen eines Programmes werden *gemeinsam* von mehreren *Transformer-Komponenten* transformiert. Die Transformer-Komponenten bestimmen dabei die Transformationen, die an den Programmklassen durchgeführt werden sollen. Im folgenden werden die Transformer-Komponenten auch kurz als *Transformer* bezeichnet.

Die Komposition der Transformer ermöglicht der *Kompositionsalgorithmus*. Er *koordiniert* und *komponiert* die Ausführung der einzelnen Transformer.

Anwendung der Transformer auf Mengen von Klassen Der Fokus der Transformation beschränkt sich nicht auf *eine* Klasse, sondern es können ganze *Mengen* von Klassen gleichzeitig transformiert werden. Dadurch können gegenseitige *Abhängigkeiten* von Klassen bei der Transformation berücksichtigt werden.

Unabhängig entwickelte Transformer Die Transformer können *unabhängig voneinander* entwickelt und miteinander komponiert werden.

Konfliktbehandlung Wenn unterschiedliche Transformer unterschiedliche Ziele verfolgen, die sich unter Umständen überschneiden oder sogar gegensätzlich sind, ist es offensichtlich, daß dabei Konflikte auftreten können. Diese Konflikte sollten erkannt und falls möglich aufgelöst werden.

Dazu überwacht der Kompositionsalgorithmus die Aktivitäten der einzelnen Transformer, und erkennt und behandelt dabei eine bestimmte Klasse von Konflikten (siehe Abschnitt 3.4.6 auf Seite 54).

Framework Ausgehend von diesem *Multi-Transformer-Modell* wird in Kapitel 5 ein Framework abgeleitet, welches dem Benutzer eine einfache Schnittstelle zur Entwicklung, Einbindung und Kombination eigener und dritter Transformer-Komponenten bietet. Das Framework soll die Komplexität kapseln, die zum einen aus der Integration eines Transformationsmechanismus in die Java-Umgebung, und zum anderen aus der Koordinierung der Ausführung der einzelnen Transformer-Komponenten entsteht.

Integration in die Java-Umgebung Das Framework wird so in die Java-Umgebung integriert, daß alle Klassen eines Programmes zur Transformation erreicht werden, ohne dabei Änderungen an einer Implementation der JVM vorzunehmen (siehe Abschnitt 3.2.2 auf Seite 34).

Hinweis Die Sprache Java unterstützt sowohl Klassen, als auch Schnittstellen. Um zwischen der Schnittstelle einer Klasse und dem Typ Schnittstelle zu unterscheiden, wird für den Schnittstellentyp im folgenden der englische Begriff *Interface* verwendet. Desweiteren werden die Begriffe Klasse und Interface in der Regel zusammengefaßt, wenn z.B. von den Klassen eines Programmes die Rede ist, eigentlich jedoch die Klassen *und* Interfaces eines Programmes gemeint sind. Darüberhinaus werden die `implements`-Klausel bei Klassen, und die `extends`-Klausel bei Interfaces, unter dem Begriff `implements`-Klausel zusammengefaßt. Die Methoden und Konstruktoren einer Klasse werden unter dem Begriff Methode zusammengefaßt. Wenn Abweichungen von diesen vereinfachenden Begriffen notwendig ist, um auf die spezifischen Eigenheiten aufmerksam zu machen, so wird darauf im Text explizit hingewiesen.

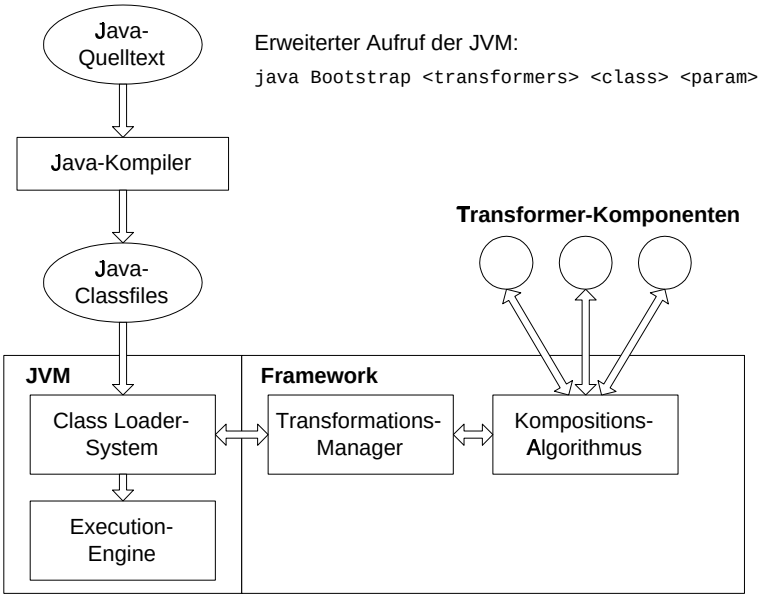

Abbildung 3.1: Überblick über die Architektur der Java-Umgebung und des Frameworks. Die geladenen Classfiles werden vor der Übergabe an die Execution-Engine vom Class Loader System an das Framework übergeben. Dort werden sie von den Transformer-Komponenten transformiert.

3.2 Architektur

3.2.1 Überblick

Abbildung 3.1 gibt einen Überblick über die Architektur der Java-Umgebung und des Frameworks. Der *Transformations-Manager* bildet die Schnittstelle zwischen der Java-Umgebung mit ihrem *Class Loader System* (Abschnitt 2.3) auf der einen und dem *Kompositionsalgorithmus* auf der anderen Seite. Er kapselt die in Kapitel 2 angesprochene Komplexität des Class Loader Systems mit seinen *Namensräumen* [Gon99b] vor dem Kompositionsalgorithmus, enthält jedoch keine eigene Logik, die Manipulationen an den Programmklassen vornimmt. Diese Aufgabe wird vom Kompositionsalgorithmus und den Transformer-Komponenten übernommen.

Durch die Aufteilung in den Transformations-Manager und den Transformationsalgorithmus ist es möglich, auch völlig andere Algorithmen zur Transformation von Klassen oder Programmen in das System zu integrieren, bzw. den im folgenden vorgestellten Algorithmus auch in anderen Kontexten, z.B. als Postprozessor nach der Kompilierung von Programmen einzusetzen.

Das Framework wird aktiviert, indem man die JVM mit einer bereitgestellten *Bootstrap-Klasse* aufruft. Diese Klasse erwartet mindestens zwei Aufrufparameter. Der erste Parameter definiert den Namen einer lokalen Datei, die die Angaben über die zu aktivierenden Transformer im *XML-Format* enthält. Der zweite Aufrufparameter definiert den Namen der Programmklasse, die gestartet werden soll. Die Bootstrap-Klasse wertet die lokale Datei aus, instanziiert

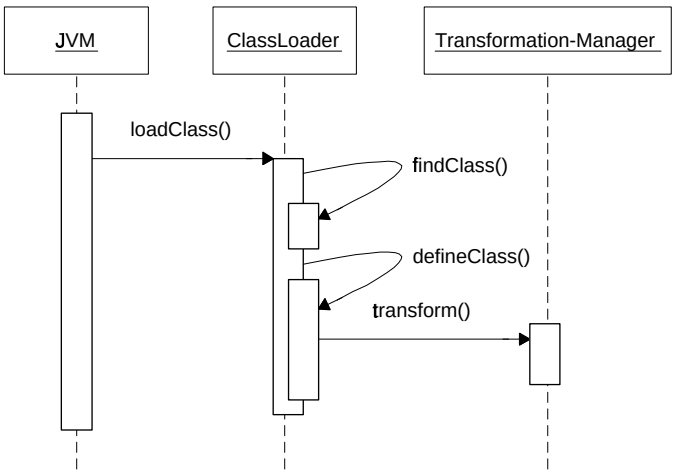

Abbildung 3.2: Integration des Transformations-Manager-Aufrufs in die Klasse ClassLoader

die darin angegebenen Transformer-Komponenten, übergibt diese an das Framework, lädt anschließend die angegebene Klasse des auszuführenden Programmes und führt deren `main`-Methode aus.

3.2.2 Integration in die Java-Umgebung

Von zentraler Bedeutung bei der Integration des Frameworks in die Java-Umgebung sind die *Class Loaders*. Abschnitt 2.3 gab bereits einen ausführlichen Überblick über diese Architektur. Daher werden an dieser Stelle nur noch einmal die wesentlichen Merkmale wiederholt.

Java-Programme werden von einem Java Kompiler in das *Classfile Format* (siehe Abschnitt 2.2) übersetzt, und zwar genau ein *Classfile* pro Java-Klasse bzw. Interface. Bei der Ausführung des Programmes werden diese Classfiles nicht von der JVM direkt, sondern vom *Class Loader System* geladen. Das Class Loader System besteht aus einer Menge von Class Loaders, die, bis auf den Bootstrap Class Loader, ganz normale Java-Objekte sind, und sich ihre Aufgabe teilen.

Class Loader bekommen in der Regel von der JVM die Aufforderung, im System noch unbekannte Klassen zu laden. Daraufhin delegiert der beauftragte Class Loader diese Aufgabe an einen anderen Class Loader, oder versucht selber, die Klasse auf die in ihm implementierte Art und Weise zu laden, und legt sie in einem Byte-Array im Classfile Format auf dem Heap ab. Anschließend übergibt er eine Referenz auf dieses Byte-Array an die JVM, indem er eine der Varianten der nicht redefinierbaren, von `ClassLoader` geerbten Methoden `defineClass` aufruft. Ab diesem Zeitpunkt ist die Klasse im System, und der Class Loader hat seine Aufgabe erfüllt.

Da es keine Möglichkeit gibt, Klassen anders als über `defineClass` in das System zu laden, bietet es sich an, den Aufruf des Transformations-Managers in

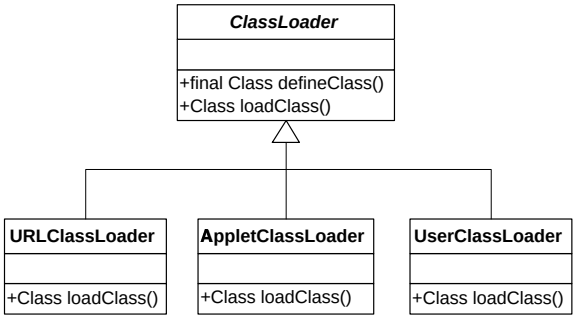

Abbildung 3.3: Da die defineClass-Methoden nicht redefiniert werden können, ist es für Subklassen von ClassLoader unmöglich, die Transformation zu umgehen.

dieses „Nadelöhr“ zu integrieren. Dazu werden die Varianten von `defineClass` in der Klasse `ClassLoader` so abgeändert, daß das Byte-Array vor der Übergabe an die JVM transformiert wird. Abbildung 3.2 auf der vorherigen Seite verdeutlicht diesen Ablauf. Da die `defineClass`-Methoden nicht redefiniert werden können, ist es für Subklassen von `ClassLoader` unmöglich, die Transformation zu umgehen. Siehe dazu Abbildung 3.3.

Die genaue Integration wird im Kapitel über das Design und die Implementation (Kapitel 5) beschrieben. Dabei zeigt sich, daß neben der Integration des Transformations-Manager-Aufrufs auch die Sichtbarkeiten der Klassen untereinander, die aus den *Namensräumen* der Class Loader resultieren, abgebildet werden müssen.

3.3 Transformationen

3.3.1 Das Java-Programm

Dieser Abschnitt führt die in diesem Kapitel verwendete Definition der Datenstruktur *Java-Programm* ein.

Definition 3.1 (Java-Klasse) Eine *Java-Klasse* ist eine Zeichenkette, die gemäß der *Java Language Specification* [GJS96] aufgebaut ist. Sei $\mathcal{K}$ die *Menge aller Java-Klassen.*

Definition 3.2 (Java-Programm) Ein *Java-Programm* ist eine Menge von Java-Klassen, sodaß innerhalb der Menge keine zwei Klassen denselben Namen besitzen. Sei $\mathcal{P}$ die *Menge aller Java-Programme*, also $\mathcal{P} = \{p \in \mathfrak{P}(\mathcal{K}) \mid \forall k \in p \ \nexists k' \in p : \ k \text{ und } k' \text{ haben denselben Namen}\}$.

Der Begriff *Java-Programm* weckt intuitiv das Verständnis, daß damit *alle* Programmklassen gemeint sind. Wie jedoch später gezeigt wird, kann die Menge der Klassen, die das auszuführende Programm bilden werden, weder zur Ladezeit, noch zu irgendeinem Zeitpunkt während der Laufzeit vollständig bestimmt werden, da es in Java jederzeit möglich ist, Klassen dynamisch nachzuladen (siehe dazu z.B. Abschnitt 2.5 auf Seite 27). Man kann aber zu jedem Zeitpunkt eine

Aussage darüber machen, welche Klassen *wenigstens* zum Programm gehören. Das sind die Klassen, die über statische symbolische Referenzen erreicht werden können, und alle Klassen, die sich zu diesem Zeitpunkt im System befinden. Wenn im folgenden von einem Programm die Rede ist, wird damit immer genau diese Menge von Klassen bezeichnet.

3.3.2 Potentielle Transformationen

Eine *Transformation* ist eine atomare Programm-Modifikation. Es folgt eine Auflistung aller potentiellen Transformationen:

- Hinzufügen einer Klasse, einer Methode oder eines Feldes,
- Entfernen einer Klasse, einer Methode oder eines Feldes,
- Umbenennen einer Klasse, einer Methode oder eines Feldes,
- Änderung einer Methodensignatur,
- Änderung einer Methoden-Throws-Klausel,
- Änderung eines Methodenrückgabetyps,
- Änderung eines Feldtyps,
- Änderung der direkten Oberklasse einer Klasse,
- Änderung der implements-Klausel einer Klasse,
- Hinzufügen und Entfernen von Annotationen einer Klasse, einer Methode oder eines Feldes,
- Modifikation der Implementation einer Methode.

Die Menge der potentiellen Transformationen stellt die Modifikationen dar, die von den Transformer-Komponenten durchgeführt werden könnten. Diese Menge wird in den folgenden Abschnitten jedoch eingeschränkt.

3.3.3 Ausschlußkriterien

Im folgenden wird jede potentielle Transformation daraufhin überprüft, welche Konsequenzen ihre Durchführung auf die Ausführbarkeit des zu transformierenden Programmes hat. Eine Transformation wird als nicht sinnvoll erachtet und daher ausgeschlossen, wenn ihre Durchführung die Ausführbarkeit des zu transformierenden Programmes im *worst case* unbedingt negativ beeinflußt, also das Programm in einen inkonsistenten Zustand überführt. Dazu wird die Spezifikation der *Binary Compatibility* [GJSB00] zu Hilfe genommen. Diese kann wie folgt zur Untersuchung der Auswirkungen von Transformationen auf das zu transformierende Programm verwendet werden.

Binary Compatibility Die Binary Compatibility definiert eine Menge von Transformationen für Pakete und Klassen, deren Durchführung auf keinen Fall weitere Transformationen direkt oder indirekt abhängiger Pakete und Klassen nach sich ziehen muß, damit das Linken dieser Pakete und Klassen auch weiterhin ohne Fehler durchgeführt werden kann (siehe hierzu Abschnitt 2.6 auf Seite 27).

Vorhersehbarkeit Transformationen von Paketen und Klassen, die eventuell weitere Transformationen abhängiger Pakete und Klassen nach sich ziehen müssen, um diese in einen konsistenten Zustand zu überführen, sind insofern problematisch, als die Menge aller direkt oder indirekt abhängiger Pakete und Klassen zur Ladezeit nicht vollständig bestimmt werden kann.

Das liegt zum einen daran, daß aus dem Binary einer Klasse A im allgemeinen nicht geschlossen werden kann, das z.B. eine Klasse B von dieser Klasse A abhängig ist. Da in diesem Fall die Transformation von Programmen betrachtet wird, könnte man jedoch annehmen, daß dieser Umstand durch die Berechnung der *transitiven Hülle* der Abhängigkeiten von Programmklassen behoben werden könnte.

Die transitive Hülle der Abhängigkeiten kann jedoch zur Ladezeit nicht vollständig vorhergesehen werden. Das liegt daran, daß für die Bestimmung der im Programm verwendeten Klassen lediglich die *statischen*, symbolischen Referenzen in den Binaries zur Verfügung stehen. Darüber hinaus gibt es in Java die Möglichkeit, Klassen *dynamisch* zu referenzieren. Dazu muß lediglich eine zur Laufzeit berechnete Zeichenkette an die Methode `forName` der Systemklasse `Class` übergeben werden. Diese Methode veranlaßt daraufhin, die entsprechende Klasse durch das Class Loader System nachzuladen, falls sie sich noch nicht im System befindet, und gibt dem Aufrufer eine Referenz auf die gewünschte Klasse zurück, die z.B. für eine Instanziierung weiterverwendet werden kann.

Reflection API Aufgrund der *Java-Reflection API* (Abschnitt 2.5 auf Seite 27) kann nicht einmal eine Aussage darüber gemacht werden, ob eine Methode oder ein Feld einer Klasse von derselben oder anderen, sich ebenfalls im System befindlichen Klassen verwendet wird. Das liegt daran, daß Methoden bzw. Felder ebenfalls dynamisch über eine zur Laufzeit zu berechnende Zeichenkette referenziert und ausgeführt bzw. referenziert und manipuliert werden können.

Die Berücksichtigung der Java-Reflection API an dieser Stelle ist von großer Wichtigkeit, da sie eine breite Verwendung findet. So werden z.B. in *JUnit* [Gam], einem Framework zum automatischen Testen von Klassen, automatisch solche Methoden ausgeführt, deren Namen ein bestimmtes Muster aufweisen. Bei den *Java-Beans* [Ham97], dem Komponentenmodell von Java, werden die *Eigenschaften* der einzelnen Komponenten zur Design-Zeit *ausschließlich* über die Reflection API manipuliert. Die für *Datenbankzugriffe* verwendeten JDBC-Treiber [WH99] werden ebenfalls typischerweise dynamisch zur Laufzeit nachgeladen.

Fazit Die Binary Compatibility wurde für eine Situation spezifiziert, in der die Menge der von einer Klasse abhängigen Klassen nicht vorhersehbar ist. In unserem Transformer-Modell kann ebenfalls die Menge von abhängigen Klassen, Methoden und Feldern aufgrund der Reflection API nicht vorhergesehen werden. Dies führt zum folgenden Ausschlußkriterium für Transformationen:

Kriterium 3.1 *Eine Transformation wird ausgeschlossen, wenn ihre Durchführung die Binary Compatibility verletzt oder verletzen könnte, und diese im* worst case *auch durch weitere, nachfolgende Transformationen nicht vollständig wiederhergestellt werden kann.*

Als Beispiel für eine Transformation, die zwar die Binary Compatibility verletzt, die jedoch durch weitere, nachfolgende Transformationen wiederhergestellt werden kann, siehe die Transformation einer Instanz- in eine Klassenmethode im folgenden Abschnitt.

Transformationen, die das oben aufgeführte Kriterium verletzen bzw. nicht verletzen, werden im folgenden als *illegale* bzw. *legale Transformation* bezeichnet.

3.3.4 Legale Transformationen

Dieser Abschnitt beginnt mit einer zusammenfassenden Übersicht über die legalen und illegalen Transformationen. Die Gründe für die jeweilige Einordnung der Transformationen werden anschließend aufgeführt. In Abschnitt 3.4 werden die erlaubten Transformationen der Transformer-Komponenten auf die legalen Transformationen eingeschränkt.

Regel 3.1 *Die folgenden Transformationen verletzen Kriterium 3.1 nicht und sind somit* legal*:*

- *Hinzufügen einer Klasse, einer Methode oder eines Feldes,*
- *Änderung einer Methoden-Throws-Klausel,*
- *Änderung der direkten Oberklasse einer Klasse gemäß Regel 3.9,*
- *Änderung der implements-Klausel einer Klasse gemäß Regel 3.10,*
- *Hinzufügen und Entfernen von Annotationen einer Klasse, einer Methode oder eines Feldes gemäß Regel 3.11 (siehe hierzu auch Abbildung 3.4 auf der nächsten Seite),*
- *Modifikation der Implementation einer Methode.*

Die folgenden Transformationen verletzen Kriterium 3.1 und sind somit illegal*:*

- *Entfernen einer Klasse, einer Methode oder eines Feldes,*
- *Umbenennen einer Klasse, einer Methode oder eines Feldes,*
- *Änderung einer Methodensignatur,*
- *Änderung eines Methodenrückgabetyps,*
- *Änderung eines Feldtyps,*

Im folgenden wird jede potentielle Transformation daraufhin überprüft, ob sie Kriterium 3.1 verletzt. Jede Überprüfung endet mit einer Regel, die das Ergebnis der Prüfung zusammenfaßt.

Hinzufügen einer Klasse, einer Methode oder eines Feldes Das Hinzufügen von Klassen zu Programmen bzw. Methoden oder Feldern zu Klassen verletzt die Binary Compatibility nicht ([GJSB00], §13.3, §13.4.5, §13.5.3).

Regel 3.2 *Das Hinzufügen einer Klasse, einer Methode oder eines Feldes ist eine* legale *Transformation.*

Element	*Annotation*	*Hinzufügen*	*Entfernen*
Klasse	abstract		✓
	final		✓
Methode	abstract		✓
	static	✓	
	synchronized	✓	✓
	native	✓	✓
	strictfp	✓	✓
	final (Instanzmethode)		✓
	final (Klassenmethode)	✓	✓
Feld	final		✓
	static	✓	
	transient	✓	✓
	volatile	✓	✓

Abbildung 3.4: Zusammenfassende Übersicht über legale Transformationen von Annotationen.

Entfernen einer Klasse, einer Methode oder eines Feldes Nach der Binary Compatibility dürfen nur paketweit sichtbare Klassen gelöscht werden, und dies auch nur dann, wenn sie innerhalb des Paketes nicht mehr referenziert werden ([GJSB00], §13.3). Die Entscheidung, ob eine Klasse innerhalb eines Paketes referenziert wird, kann zur Ladezeit nicht getroffen werden, da eventuell zu einem späteren Zeitpunkt Klassen des entsprechenden Paketes nachgeladen werden, die diese Klasse referenzieren. Das Entfernen von Klassen wird daher ausgeschlossen.

Nach der Binary Compatibility dürfen nur als `private` deklarierte Felder und Methoden einer Klasse gelöscht werden ([GJSB00], §13.4.5). Dies geschieht typischerweise genau dann, wenn sie innerhalb der Klasse nicht mehr benötigt werden. Im Rahmen der Ladezeittransformation kann eine solche Annahme von einer Transformer-Komponente jedoch nicht gemacht werden, da die Methode oder das Feld eventuell über die Reflection API verwendet werden.

Regel 3.3 *Das Entfernen einer Klasse, einer Methode oder eines Feldes ist eine* illegale *Transformation.*

Umbenennen einer Klasse, einer Methode oder eines Feldes Da Klassen, Methoden und Felder innerhalb eines Binaries *ausschließlich* über ihren Namen und ihre Signatur referenziert werden (siehe Abschnitt 2.6 auf Seite 27), entspricht die Umbenennung eines Elementes dessen Entfernung mit anschließender Einfügung des umbenannten Elementes. Das Entfernen von Elementen wurde in Regel 3.3 ausgeschlossen.

Regel 3.4 *Das Umbenennen einer Klasse, einer Methode oder eines Feldes ist eine* illegale *Transformation.*

Änderung einer Methoden-Throws-Klausel Änderungen von Methoden-Throws-Klauseln verletzen die Binary Compatibility nicht ([GJSB00], §13.4.19),

da die Korrektheit der `throws`-Deklaration einer Methode nur zum Zeitpunkt der Kompilierung überprüft wird.

Regel 3.5 *Die Änderung der* `throws`*-Klausel einer Methode ist eine* legale *Transformation.*

Änderung einer Methodensignatur Wird der Typ eines Methodenparameters geändert, ein Parameter zu der Methode hinzugefügt oder ein Parameter der Methode entfernt, so erzeugt dies eine Methode mit einer neuen Signatur. Da Methoden über ihre Signatur eindeutig identifiziert werden, hat dies den Effekt der Entfernung der Methode mit der alten Signatur und dem anschließenden Einfügen der Methode mit der neuen Signatur. Das Entfernen von Methoden wurde in Regel 3.3 ausgeschlossen.

Regel 3.6 *Die Änderung der Signatur einer Methode ist eine* illegale *Transformation.*

Änderung eines Methodenrückgabetyps Die Änderung eines Methodenrückgabetyps verletzt die Binary Compatibility ([GJSB00], §13.4.13), da dies dem Entfernen der alten Methode und dem Einfügen der geänderten Methode entspricht.

Regel 3.7 *Die Änderung eines Methodenrückgabetyps ist eine* illegale *Transformation.*

Änderung eines Feldtyps Die Binary Compatibility macht über die Änderung eines Feldtyps keine Aussage. Ähnlich der Änderung eines Methodenrückgabetyps entspricht die Änderung eines Feldtyps dem Entfernen des Feldes mit anschließendem Hinzufügen des geänderten Feldes. Das Entfernen eines Feldes wurde jedoch bereits ausgeschlossen.

Regel 3.8 *Die Änderung eines Feldtyps ist eine* illegale *Transformation.*

Änderung der direkten Oberklasse einer Klasse Die Binary Compatibility macht über die Änderung der direkten Oberklasse einer Klasse die folgende Aussage ([GJSB00], §13.4.4):

> „Changing the direct superclass or the set of direct superinterfaces of a class type will not break compatibility with pre-existing binaries, provided that the total set of superclasses or superinterfaces, respectively, of the class type loses no members.“

Diese Aussage führt zu folgender Regel:

Regel 3.9 *Die Änderung der direkten Oberklasse einer Klasse ist genau dann eine* legale *Transformation, wenn die gesamte Menge aller Oberklassen dadurch kein Element verliert.*

Abbildung 3.5 auf der nächsten Seite zeigt eine erlaubte Änderung der direkten Oberklasse.

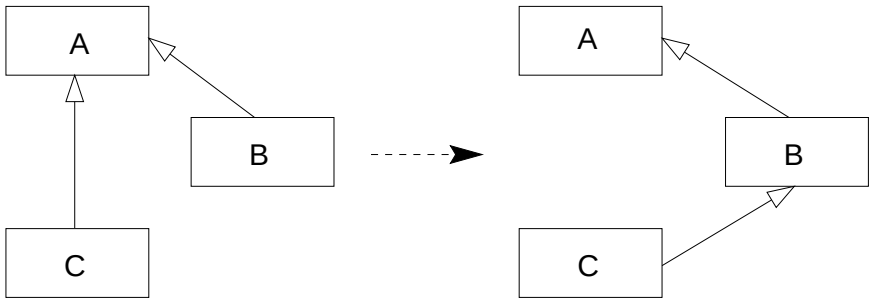

Abbildung 3.5: Beispiel für eine erlaubte Oberklassenänderung.

Änderung der implements-Klausel einer Klasse Wie oben zitiert, wird die Binary Compatibility nicht verletzt, wenn durch eine Änderung der `implements`-Klausel einer Klasse die gesamte Menge aller Superinterfaces der Klasse kein Element verliert.

Regel 3.10 *Die Änderung der* `implements`*-Klausel einer Klasse ist genau dann eine* legale *Transformation, wenn die gesamte Menge aller Superinterfaces der Klasse kein Element verliert.*

Hinzufügen und Entfernen von Annotationen einer Klasse, einer Methode oder eines Feldes Im folgenden werden die Annotationen, die die Sichtbarkeit eines Elementes deklarieren und alle sonstigen Annotationen getrennt betrachtet.

Zugriffsannotationen In der Sprache Java ist die einzige Zugriffsannotation für Klassen die Annotation `public`. Eine nicht als `public` deklarierte Klasse ist automatisch *paketweit* sichtbar.

Methoden und Felder können in Java mit den Annotationen `public`, `protected` und `private` deklariert werden. Eine Methode bzw. ein Feld, das nicht als `public`, `protected` oder `private` deklariert ist, ist implizit paketweit sichtbar.

Die Verringerung der Sichtbarkeit eines Elementes verletzt die Binary Compatibility, und kann zur Laufzeit einen Link-Fehler auslösen ([GJSB00], §13.4.3, §13.4.6).

Die Erweiterung der Sichtbarkeit eines Elementes hingegen verletzt die Binary Compatibility nicht.

Sonstige Annotationen Unter den sonstigen Annotationen werden im folgenden alle Annotationen für Klassen, Methoden und Felder zusammengefaßt, die nicht deren Sichtbarkeit festlegen. Diese werden im folgenden für Klassen, Methoden und Felder getrennt betrachtet. Die Tabelle in Abbildung 3.4 auf Seite 39 faßt die im folgenden beschriebenen Ergebnisse zusammen.

Klassen können als `abstract` und `final` deklariert werden:

abstract Eine Klasse wird als *abstract* deklariert, wenn sie abstrakte Methoden besitzt, oder vor einer Instanziierung geschützt werden soll.

Wird die Annotation `abstract` einer Klasse entfernt, so darf sie keine abstrakten Methoden besitzen, da in diesem Fall bei der Übergabe an die JVM ein `VerifyError` ausgelöst wird. Besitzt sie hingegen keine abstrakten Methoden, so wird kein `VerifyError` ausgelöst. In diesem Fall wird die Binary Compatibility nicht verletzt ([GJSB00], §13.4.1).

Die nachträgliche Deklaration einer Klasse als `abstract` verletzt die Binary Compatibility, da zur Laufzeit keine Instanzen dieser Klasse erzeugt werden können ([GJSB00], §13.4.1).

final Die Annotation `final` kennzeichnet eine Klasse als vollständig. Es können keine Unterklassen einer finalen Klasse gebildet werden.

Wird die Annotation `final` einer Klasse entfernt, so hat dies keine negativen Auswirkung auf die Programmausführung. Allenfalls die vom Programmierer der Klasse unerwünschte Unterklassenbildung wird so wieder ermöglicht. Die Binary Compatibility wird dadurch nicht verletzt ([GJSB00], §13.4.2).

Wird die Annotation `final` zu einer Klasse hinzugefügt, so führt dies zur Zurückweisung ihrer Unterklassen durch den *Verifier*. Da weder zur Ladezeit noch zur Laufzeit entschieden werden kann, ob Unterklassen dieser Klassen in das System geladen werden sollen, verletzt diese Transformation die Binary Compatibility ([GJSB00], §13.4.2).

Methoden können mit den Annotationen `abstract`, `static`, `final`, `synchronized`, `strictfp` und `native` deklariert werden:

abstract Eine als `abstract` deklarierte Methode kennzeichnet diese als Bestandteil einer Klasse und stellt ihre Signatur, ihren Rückgabetyp und ihre `throws`-Klausel, jedoch nicht ihre Implementation bereit.

Wird eine Methode von einer Transformer-Komponente nachträglich als abstrakt deklariert, so wird ihre Klasse automatisch auch abstrakt. Eine Klasse nachträglich als abstrakt zu deklarieren, wurde jedoch bereits ausgeschlossen.

Die Transformation einer abstrakten Methode in eine nicht abstrakte Methode hat keine negativen Auswirkungen auf die Ausführbarkeit des Programmes, wenn ebenfalls ihre Implementation bereitgestellt wird. Diese Transformation verletzt die Binary Compatibility nicht ([GJSB00], §13.4.14).

static Eine als `static` deklarierte Methode ist eine Klassenmethode. Eine nicht als `static` deklarierte Methode ist eine Instanzmethode.

Die Binary Compatibility verbietet zwar die Transformation einer nicht privaten Instanzmethode in eine Klassenmethode und umgekehrt ([GJSB00], §13.4.7), man kann in diesem Fall jedoch die folgende Überlegung anstellen:

Eine Instanzmethode kann zu einer Klassenmethode transformiert werden, wenn in ihrer Implementation alle Zugriffe auf `this` entfernt werden. Eine Klassenmethode kann ohne Anpassung ihrer Implementation zu einer Instanzmethode transformiert werden.

Um die Ausführbarkeit des Programmes weiterhin zu gewährleisten, müssen bei einer solchen Transformation jedoch global alle Aufrufe der transformierten Methode angepaßt werden. Die statischen Aufrufe, die innerhalb der Classfiles in den Implementationen anderer Methoden stehen, stellen dabei kein Problem dar. Die Aufrufe der Methoden über die Reflection API können jedoch auch hier im Vorfeld nicht vollständig bestimmt werden. Bei der Transformation einer Instanz- in eine Klassenmethode müssen diese jedoch gar nicht modifiziert werden.

Der Zugriff auf eine Methode gestaltet sich in der Reflection API unabhängig davon, ob es sich um eine Klassen- oder Instanzmethode handelt. Der Aufruf per `invoke(...)` erwartet neben den Methodenparametern zusätzlich die Übergabe der Pseudovariablen `this`. Ihr Wert wird bei Klassenmethoden jedoch ignoriert. Im Falle der Umwandlung einer Instanz- in eine Klassenmethode müssen die Aufrufe der Methode, die über die Reflection API realisiert werden, daher nicht angepaßt werden.

Der umgekehrte Fall funktioniert jedoch nicht ohne Modifikation der Methodenaufrufe, die mittels der Reflection API realisiert wurden, und scheidet daher aus.

final Wird eine Methode als `final` deklariert, so kann sie in Unterklassen nicht überschrieben oder verdeckt werden.

Wird die Annotation `final` einer finalen Instanzmethode entfernt, so verletzt dies die Binary Compatibility nicht ([GJSB00], §13.4.15). Für den umgekehrten Fall gilt dies jedoch nicht, da auch hier zur Ladezeit nicht bestimmt werden kann, ob Unterklassen in das System gelangen werden, die die entsprechende Methode überschreiben.

Das Hinzufügen und Entfernen der Annotation `final` einer Klassenmethode verletzt die Binary Compatibility nicht ([GJSB00],§13.4.15), da Klassenmethoden nicht überschrieben werden können.

synchronized Eine als `synchronized` deklarierte Methode setzt automatisch einen *monitor lock* bevor sie ausgeführt wird.

Das Hinzufügen oder Entfernen der Annotation `synchronized` verletzt die Binary Compatibility nicht ([GJSB00], §13.4.18).

strictfp In einer als `strictfp` deklarierten Methode sind alle Ausdrücke der Methode *FP-strict*[1].

Die Binary Compatibility macht über das Hinzufügen oder Entfernen der Annotation `strictfp` keine Aussage. Da diese Annotation jedoch lediglich einen Einfluß auf die Floating Point Operationen der Methodenimplementation hat, wird die Ausführbarkeit des Programmes nicht negativ beeinflußt.

[1] „Within an FP-strict expression, all intermediate values must be elements of the float value set or the double value set, implying that the results of all FP-strict expressions must be those predicted by IEEE 754 arithmetic on operands represented using single and double formats. Within an expression that is not FP-strict, some leeway is granted for an implementation to use an extended exponent range to represent intermediate results; the net effect, roughly speaking, is that a calculation might produce “the correct answer“ in situations where exclusive use of the float value set or double value set might result in overflow or underflow.“ [GJSB00], §15.4

native Eine Methode kann als `native` deklariert werden, wenn sie nicht in Java, sondern in einer plattformabhängigen Sprache wie z.B. C oder C++ implementiert werden soll.

Das Hinzufügen oder Entfernen der Annotation `native` verletzt die Binary Compatibility nicht ([GJSB00], §13.4.16). Dabei müssen jedoch zwei Punkte beachtet werden:

- Wird eine Methode durch eine Transformer-Komponente nachträglich als `native` deklariert, so muß diese auch sicherstellen, daß die plattformabhängige Implementation der Methode der JVM zur Verfügung steht.
- Wird eine `native` Methode durch eine Transformer-Komponente nachträglich als nicht `native` deklariert, so muß diese lediglich die Methode in der üblichen Form als eine Liste von *Opcodes* implementieren. Die der JVM eventuell zur Verfügung gestellte plattformabhängige Implementierung der Methode wird von der JVM in diesem Fall ignoriert.

Felder können mit den Annotationen `final`, `static`, `transient` und `volatile` deklariert werden:

final Wird ein Feld als `final` deklariert, so behält es nach der Initialisierung immer denselben Wert. Bei dem Versuch, einem finalen Feld nach der Initialisierung einen anderen Wert zuzuweisen, wird eine Laufzeitausnahme ausgelöst.

Die nachträgliche Deklaration eines Feldes als `final` verletzt die Binary Compatibility ([GJSB00], §13.4.8), da Zuweisungen an das Feld während der Laufzeit zu einer unerwarteten Ausnahme führen.

Das nachträgliche Entfernen der Annotation `final` eines Feldes verletzt die Binary Compatibility nicht.

static Ist ein Feld als `static` deklariert, so existiert genau eine Inkarnation dieses Feldes, unabhängig davon, wieviele Instanzen der Klasse erzeugt wurden. Ein als `static` deklariertes Feld, auch *Klassenvariable* genannt, wird inkarniert, wenn die Klasse initialisiert wird.

Wird die Annotation `static` zu einem Feld hinzugefügt oder entfernt, so verletzt dies die Binary Compatibility ([GJSB00], §13.4.9).

Bei Feldern gilt jedoch dieselbe Argumentation bezüglich dieser Annotation wie bei den Methoden. Wird eine Instanzvariable zu einer Klassenvariablen transformiert, so reicht es, die Feldzugriffe, die über statische, symbolische Referenzen in den Methodenimplementationen erfolgen, zu transformieren. Der bei Zugriffen über die Reflection API erforderliche `this`-Parameter wird während der Ausführung ignoriert.

transient Felder können als `transient` deklariert werden, um anzuzeigen, daß sie nicht Teil des persistenten Zustandes eines Objektes sind. Diese Annonation hat insbesondere im Rahmen der *object serialization* [ser] eine besondere Bedeutung.

Das Entfernen oder Hinzufügen der Annotation `transient` verletzt die Binary Compatibility nicht ([GJSB00], §13.4.10).

volatile Wird ein Feld als `volatile` deklariert, so muß ein Thread seine Arbeitskopie dieses Feldes vor jedem Zugriff mit der Masterkopie in Einklang bringen.

Die Binary Compatibility macht über das Hinzufügen oder Entfernen der Annotation `volatile` keine Aussage. Da diese Annotation jedoch lediglich einen Einfluß auf die Ausführung der Threads hat, wird die Ausführbarkeit des Programmes nicht negativ beeinflußt.

Regel 3.11 faßt die erlaubten Transformationen der Annotationen zusammen:

Regel 3.11 *Die Änderung der Sichtbarkeit einer Klasse, einer Methode oder eines Feldes ist genau dann eine* legale *Transformation, wenn dadurch die Sichtbarkeit erhöht wird.*

Die Änderung der sonstigen Annotationen eines Elementes ist genau dann eine legale *Transformation, wenn sie der Tabelle in Abbildung 3.4 entspricht.*

Modifikation der Implementation einer Methode Die Modifikation der Implementation einer Methode verletzt die Binary Compatibility nicht ([GJSB00], §13.4.20). Dies liegt unter anderem auch daran, daß in der Spezifikation der Binary Compatibility das sogenannte *Method-Inlining* für jegliche Methoden, also auch für finale Methoden, durch einen Kompiler ausgeschlossen wird[2].

Bemerkung Der völlige Ausschluß des Method-Inlinings existiert erst seit Version 2 der *Java Language Specification* [GJSB00]. In der 1. Version ist Method-Inlining in einigen wenigen Situationen erlaubt (siehe [GJS96], §13.4.21).

Regel 3.12 *Die Modifikation der Implementation einer Methode ist eine* legale *Transformation.*

3.4 Komposition von Transformationen

Dieser Abschnitt hat zum Ziel, das für die gesamte Arbeit grundlegende Konzept der Transformer-Komponenten und deren Komposition durch den Kompositionsalgorithmus vorzustellen. Dazu werden in Abschnitt 3.4.1 die Aufgaben und Anforderungen des Kompositionsalgorithmus und der Transformer-Komponenten detailliert beschrieben. In Abschnitt 3.4.2 folgt ein erster Versuch für die Definition einer Transformer-Komponente und den Entwurf eines Kompositionsalgorithmus. Bei genauerer Betrachtung (Abschnitt 3.4.3) stellt sich jedoch heraus, daß diese noch unerwünschte Eigenschaften besitzen. Eine Überarbeitung (Abschnitt 3.4.4 und 3.4.5) führt schließlich in Abschnitt 3.4.6 zu dem Konzept, welches die Grundlage für eine präzise, formale Ausarbeitung in Kapitel 4 ist.

[2] „We note that a compiler cannot expand a method inline at compile time.“ [GJSB00], §13.4.20

3.4.1 Aufgaben und Anforderungen

Eine Transformer-Komponente hat die Aufgabe, ein Java-Programm zu transformieren. Der Kompositionsalgorithmus hat im Prinzip eine ähnliche Aufgabe. Er soll das Programm jedoch nicht selber transformieren, sondern vielmehr die gemeinsame Programmtransformation durch mehrere Transformer-Komponenten ermöglichen.

Jede Transformer-Komponente soll die Programmklassen bezüglich eines bestimmten Aspektes transformieren. Durch die Kombination unterschiedlicher Komponenten mithilfe des Kompositionsalgorithmus können dann unterschiedliche Aspekte miteinander kombiniert werden. Ein Aspekt kann in diesem Zusammenhang z.B. die Ergänzung aller Methodenimplementationen um die Ausgabe von Debug-Informationen oder die Anpassung von Klassen an geänderte Bedingungen sein, wie dies etwa bei der *Binary Component Adaptation* [KH98] passiert.

Da jede Komponente bei der Programmtransformation ein eigenes Ziel verfolgt, und der Kompositionsalgorithmus zur Aufgabe hat, die aktiven Komponenten das Programm gemeinsam transformieren zu lassen, muß er ihre Ausführung steuern, ihre Transformationen überwachen, Konflikte zwischen den Transformationen einzelner Komponenten erkennen und diese falls möglich auflösen.

Eine weitere wichtige Anforderung an den Kompositionsalgorithmus ist, daß das Ergebnis *eindeutig* sein soll. Das bedeutet in diesem Zusammenhang, daß bei gleichen aktiven Transformer-Komponenten und gleichem Eingabeprogramm immer dasselbe transformierte Programm erzeugt wird.

Der Kompositionsalgorithmus hat nicht zur Aufgabe, den *Java-Verifier* (Abschnitt 2.4) zu ersetzen. Im folgenden werden der Kompositionsalgorithmus und die Transformer-Komponenten zwar so aufgebaut, daß grobe Verstöße gegen die Java Language Specification bzw. das Classfile Format verhindert werden, die Überprüfung der Klassen auf ihre Konformität hinsichtlich dieser Spezifikationen wird und soll jedoch auch weiterhin vom Verifier durchgeführt werden.

Diese Aufgaben und Anforderungen lassen sich wie folgt zusammenfassen:

- **Transformer-Komponente:**
  - Transformation von Java-Programmen.
- **Kompositionsalgorithmus:**
  - Komposition von Transformer-Komponenten,
  - Erkennung und Behandlung von Konflikten,
  - Eindeutigkeit des Ergebnisses.

3.4.2 Naive Komposition

In diesem Abschnitt wird ein erster Versuch unternommen, die Transformer-Komponente zu definieren und einen Kompositionsalgorithmus zu entwerfen.

Transformer-Komponente

Wie oben erwähnt, hat eine Transformer-Komponente die Aufgabe, Java-Programme zu transformieren. Auf der konzeptuellen Ebene kann man sich einen

Transformer als eine Abbildung vorstellen, die ein Java-Programm auf ein transformiertes Programm abbildet. Um die Konsistenz des zu transformierten Programmes weitgehend zu gewährleisten, soll ein Transformer nur legale Transformationen durchführen.

Darüber hinaus müssen Transformer eine weitere wichtige Forderung beachten. Da das zu transformierende Programm zu jedem Zeitpunkt in der Transformationsphase modifiziert werden kann, darf eine Transformer-Komponente eine Transformation nicht durchführen, weil das Programm eine bestimmte Eigenschaft nicht besitzt, die es jedoch durch eine legale Transformation im weiteren Verlauf erlangen könnte. Abbildungen, die diese Forderung beachten, werden im folgenden als *wohlgeformt* bezeichnet.

Definition 3.3 Eine Abbildung $f : \mathcal{P} \longrightarrow \mathcal{X}$, wobei $\mathcal{X}$ eine beliebige Menge sei, heißt genau dann *wohlgeformt*, wenn das Ergebnis $f(p)$ eines Programmes $p \in \mathcal{P}$ nicht von solchen Eigenschaften abhängt, die p *nicht* besitzt, jedoch durch legale Transformationen erlangen könnte.

Das folgende Beispiel zeigt eine *nicht wohlgeformte* Abbildung:

Beispiel 3.1 Die Abbildung a fügt zu einer Klasse k des Programmes p genau dann eine Methode `m` hinzu, wenn k *kein* Feld besitzt.

a ist deshalb keine wohlgeformte Abbildung, weil das Hinzufügen eines Feldes eine legale Transformation darstellt.

Es folgt die Definition der Transformer-Komponente:

Definition 3.4 (Transformer-Komponente) Eine *Transformer-Komponente* ist eine wohlgeformte Abbildung $\kappa : \mathcal{P} \longrightarrow \mathcal{P}$, die nur legale Transformationen durchführt. Im folgenden wird die Menge aller Transformer-Komponenten mit $\mathbb{K}$ bezeichnet.

Beispiel 3.2 Eine Abbildung, die ein Programm auf ein Programm mit optimierten Methodenimplementationen abbildet, ist eine Transformer-Komponente.

Beispiel 3.3 Eine Abbildung, die zu jeder Programmklasse einen Instanzzähler hinzufügt, ist eine Transformer-Komponente.

Der Kompositionsalgorithmus NAIV

Im folgenden wird ein erster Ansatz für einen Kompositionsalgorithmus vorgestellt, bei dem die Konflikterkennung und -behandlung zur Vereinfachung nicht berücksichtigt wird. Sie wird zu einem späteren Zeitpunkt integriert (siehe Abschnitt 3.4.6).

Die Eingabe des Algorithmus ist eine Menge von Transformer-Komponenten und ein Java-Programm. Der Algorithmus hat nun die Aufgabe, die Ausführung der Komponenten zu steuern. Da die Konflikterkennung noch unberücksichtigt bleibt, beschränkt sich dies darauf, das Programm nacheinander durch die Komponenten transformieren zu lassen. Dabei ist zu beachten, daß die Transformation durch eine Komponente eventuell weitere Transformationen durch andere

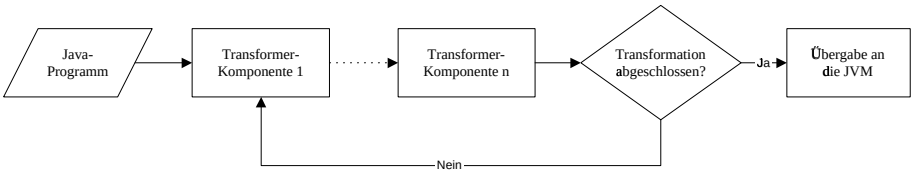

Abbildung 3.6: Der Kompositionsalgorithmus *NAIV*

Komponenten nötig macht. Das Programm muß also mehrfach von den einzelnen Komponenten bearbeitet werden. Dies entspricht der Berechnung eines *Fixpunktes*, falls dieser existiert.

Der folgende Algorithmus *NAIV* komponiert die Transformer-Komponenten, und wendet pro Iteration die Komposition einmal auf das Programm an. Der Algorithmus iteriert solange, bis der Fixpunkt erreicht wird, d.h. das Ergebnis der letzten Iteration gleich dem zu transformierenden Programm am Anfang der Iteration ist. Bei nicht existierendem Fixpunkt hält der Algorithmus nicht.

Da die Transformer-Komponenten keine vorgegebene Ordnung haben, wird vom Algorithmus *NAIV* für die Komposition eine zufällige Reihenfolge gewählt.

Algorithmus 3.1 (NAIV)
Eingabe:

- $\kappa_1, \ldots, \kappa_n \in \mathbb{K}$ die Transformer,
- $p \in \mathcal{P}$ das zu transformierende Programm.

Berechnung:

1. do
2. $\quad p' = p$;
3. $\quad p = \kappa_n \circ \kappa_{n-1} \circ \ldots \circ \kappa_1(p')$;[3]
4. until $p' = p$;
5. return p;

3.4.3 Auswirkungen der Kompositionsreihenfolge

Eine wesentliche Forderung an den Kompositionsalgorithmus ist die Eindeutigkeit des Ergebnisses.

Das Ziel dieses Abschnitts ist es, zu untersuchen, ob die Reihenfolge, in der die Komponenten komponiert werden, einen Einfluß auf das Ergebnis, also das transformierte Programm hat. Dazu dient das folgende Beispiel:

Beispiel 3.4 Ein Programm, bestehend aus der Klasse `C`, soll von zwei Transformer-Komponenten $\kappa_{access}, \kappa_{counter} \in \mathbb{K}$ transformiert werden. κ_{access} erweitert eine Klasse für jedes nicht primitive Feld um zwei Zugriffsmethoden und

[3]Die Notation $\circ$ bedeutet die *Komposition* zweier Abbildungen und ist wie folgt definiert: b $\circ$ a (x) := b(a(x)).

ersetzt alle direkten Zugriffe auf diese Felder durch die entsprechenden Methodenaufrufe. $\kappa_{counter}$ erweitert eine Klasse für jedes Feld um einen Zähler, der die Zugriffe auf dieses Feld zählt.

Für den Algorithmus *NAIV* existieren nun genau 2 Möglichkeiten, die zwei Transformer-Komponenten κ_{access} und $\kappa_{counter}$ mit einander zu komponieren:

1. $\kappa_{access} \circ \kappa_{counter}$,
2. $\kappa_{counter} \circ \kappa_{access}$.

Als zu transformierendes Programm wird die folgende Klasse `C` verwendet, die ein Feld `b` des Typs `B` und eine Methode `manipulateB` besitzt, in der auf `b` zugegriffen wird:

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

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

Startet man den Algorithmus *NAIV* mit der Komposition $\kappa_{access} \circ \kappa_{counter}$, so kann folgender Ablauf beobachtet werden:

1. $\kappa_{counter}$ stößt auf das Feld `b` und fügt daraufhin der Klasse `C` ein neues Feld `b_counter` hinzu, das als Zähler für die Zugriffe auf `b` dienen soll. Da in der Methode `manipulateB` auf `b` zugegriffen wird, wird an dieser Stelle der Code zur Erhöhung des Zählers `b_counter` eingefügt.
2. κ_{access} stößt auf das Feld `b` und fügt daraufhin der Klasse `C` die Methoden `getB` und `setB` zum Zugriff und zur Manipulation von `b` hinzu. Der direkte Zugriff auf `b` in `manipulateB` wird durch einen Methodenaufruf von `getB` ersetzt.
3. $\kappa_{counter}$ stößt auf den Feldzugriff auf `b` in `getB` und fügt daraufhin Code zur Erhöhung des Zählers `b_counter` ein.
4. Transformation abgeschlossen.

Die Klasse `C` ist zur folgenden Klasse `C'` transformiert worden:

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

    private void setB(B b) {        // neu      in Schritt   2
        this.b = b;                 // neu      in Schritt   2
    }                               // neu      in Schritt   2
    private B getB() {              // neu      in Schritt   2
        b_counter++;                // neu      in Schritt     3
        return b;                   // neu      in Schritt   2
    }
    public void manipulateB() {
```

```
        b_counter++;                // neu      in Schritt 1
        getB().doSomething();       // geändert in Schritt   2
    }
}
```

Startet man den Algorithmus *NAIV* mit der Komposition $\kappa_{counter} \circ \kappa_{access}$, so kann folgender Ablauf beobachtet werden:

1. κ_{access} stößt auf das Feld `b` und fügt daraufhin der Klasse `C` die Methoden `getB` und `setB` zum Zugriff und zur Manipulation von `b` hinzu. Der direkte Zugriff auf `b` in `manipulateB` wird durch einen Methodenaufruf von `getB` ersetzt.

2. $\kappa_{counter}$ stößt auf das Feld `b` und fügt daraufhin der Klasse `C` ein neues Feld `b_counter` hinzu, das als Zähler für die Zugriffe auf `b` dienen soll. Da in der Methode `getB` auf `b` zugegriffen wird, wird an dieser Stelle der Code zur Erhöhung des Zählers `b_counter` eingefügt.

3. Transformation abgeschlossen.

Die Klasse `C` ist zur folgenden Klasse `C''` transformiert worden:

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

    private void setB(B b) {        // neu      in Schritt 1
        this.b = b;                 // neu      in Schritt 1
    }                               // neu      in Schritt 1
    private B getB() {              // neu      in Schritt 1
        b_counter++;                // neu      in Schritt   2
        return b;                   // neu      in Schritt 1
    }
    public void manipulateB() {
        getB().doSomething();       // geändert in Schritt 1
    }
}
```

Die Klassen `C'` und `C''` unterscheiden sich offensichtlich in der Methode `manipulateB`. Die Ausführung von `manipulateB` erhöht den Zähler `b_counter` in `C'` doppelt so oft wie in `C''`.

Beobachtung 3.1 Anhand von Beispiel 3.4 können zwei wesentliche Beobachtungen gemacht werden:

1. Der Methoden-Code in `manipulateB` ist abhängig von der Komposition der Transformer-Komponenten.

2. Die Schnittstellen der transformierten Klassen `C'` und `C''` sind unabhängig von der Komposition der Transformer-Komponenten.

3.4.4 Aufteilung der Transformation

Beispiel 3.4 hat gezeigt, daß die Reihenfolge, in der die Transformer-Komponenten komponiert werden, einen wesentlichen Einfluß auf das Ergebnis der Transformation hat. Dieser Effekt ist jedoch unerwünscht, da gefordert wurde, daß das Ergebnis der Transformation eindeutig sein muß. Dieser Mißstand soll nun behoben werden. Dazu kann die folgende Beobachtung gemacht werden:

Beobachtung 3.2 Die Schnittstelle einer Klasse ist eine ungeordnete Menge von Namen + Signaturen. Die Implementation einer Methode hingegen ist eine geordnete Liste von Befehlen. Da eine ungeordnete Menge keine definierte Reihenfolge der Elemente hat, hat die Reihenfolge, in der die Elemente in die Menge eingefügt werden, keinen Einfluß auf das Ergebnis. Die Reihenfolge, in der neue Elemente in eine Liste eingefügt werden, macht jedoch einen wesentlichen Unterschied im Ergebnis aus.

Übertragen auf die Transformation von Java-Programmen heißt dies, daß es sinnvoll ist, den Vorgang der Transformation in zwei Phasen aufzuteilen:

- **Phase eins**, in der Mengen modifiziert werden, und
- **Phase zwei**, in der Listen transformiert werden.

Nun werden noch einmal die Transformationen betrachtet, die von einer Transformer-Komponente durchgeführt werden dürfen. Dies sind:

- Hinzufügen einer Klasse, einer Methode oder eines Feldes,
- Änderung einer Methoden-Throws-Klausel,
- Änderung der direkten Oberklasse einer Klasse gemäß Regel 3.9,
- Änderung der implements-Klausel einer Klasse gemäß Regel 3.10,
- Hinzufügen und Entfernen von Annotationen einer Klasse, einer Methode oder eines Feldes gemäß Regel 3.11,
- Modifikation der Implementation einer Methode.

Bis auf die Modifikation der Implementation einer Methode, handelt es sich um Änderungen von Mengen, die insgesamt die Schnittstelle (engl. Interface) einer Klasse bilden. Wird nun die Transformation des Programmes in die Phasen

- Interface-Transformation und
- Code-Transformation

aufgeteilt, so muß vom Anwender des Frameworks lediglich für die Code-Transformation die Reihenfolge der Transformationen vorgegeben werden. Die Interface-Transformation geht der Code-Transformation voraus, und ist weiterhin unabhängig von der Reihenfolge der Transformationen und entspricht auch künftig der Berechnung eines Fixpunktes.

Die Menge der Transformationen, die die Schnittstelle einer Klasse manipulieren, bzw. neue Klassen zu einem Paket hinzufügen, wird im folgenden mit der Menge der *Interface-Transformationen* bezeichnet, und ist wie folgt definiert:

Definition 3.5 (Interface-Transformationen) Sei $\mathcal{E}$ die Menge der legalen Interface-Transformationen. Dies sind:

- Hinzufügen einer Klasse, einer Methode oder eines Feldes,
- Änderung einer Methoden-Throws-Klausel,
- Änderung der direkten Oberklasse einer Klasse gemäß Regel 3.9,
- Änderung der implements-Klausel einer Klasse gemäß Regel 3.10,
- Hinzufügen und Entfernen von Annotationen einer Klasse, einer Methode oder eines Felds gemäß Regel 3.11.

Der Kompositionsalgorithmus PROGRESS

Aufgrund der Aufteilung der Transformation in zwei Phasen, muß auch die Transformer-Komponente in zwei Teile aufgeteilt werden:

- Die *Interface-Transformer-Komponente* ist verantwortlich für die Manipulation der Schnittstellen der Klassen.
- Die *Code-Transformer-Komponente* ist verantwortlich für die Manipulation der Methodenimplementationen.

Definition 3.6 (Interface-Transformer-Komponente) Eine *Interface-Transformer-Komponente* ist eine wohlgeformte Abbildung $\kappa : \mathcal{P} \longrightarrow \mathcal{P}$, die nur Interface-Transformationen durchführt. Die Menge aller Interface-Transformer-Komponenten wird im folgenden mit $\mathbb{K}_i$ bezeichnet.

Diese Definition einer Interface-Transformer-Komponente verbietet lediglich die Transformation bestehender Methodenimplementationen. Beim Einfügen einer nicht abstrakten Methode in eine Klasse, darf und muß jedoch die *initiale* Methodenimplementation ebenfalls eingefügt werden.

Definition 3.7 (Code-Transformer-Komponente) Eine *Code-Transformer-Komponente* ist eine Abbildung $\xi : \mathcal{P} \longrightarrow \mathcal{P}$, die nur Methodenimplementationen transformiert. Im folgenden wird mit $\mathbb{K}_c$ die Menge aller Code-Transformer-Komponenten bezeichnet.

Ausgehend von diesen Definitionen muß nun ein neuer Kompositionsalgorithmus entworfen werden, der die Transformation in den oben genannten zwei Phasen durchführt.

Es gibt zwei wesentliche Punkte, die bei seinem Entwurf zu berücksichtigen sind:

- Da die Interface-Transformation im wesentlichen der Erweiterung von Mengen entspricht, ist die Reihenfolge der Komposition der Interface-Transformer-Komponenten unerheblich. Es gilt jedoch weiterhin, daß die Transformationen einer Komponente eventuell Transformationen anderer Komponenten auslösen, sodaß die Interface-Transformation weiterhin iterativ durchgeführt wird.

- Da die Code-Transformation der Manipulation von Listen entspricht, hat die Reihenfolge der Komposition der Code-Transformer-Komponenten einen wesentlichen Einfluß auf das Ergebnis. Die Reihenfolge darf daher nicht vom Algorithmus willkürlich gewählt werden, sondern muß in der Eingabe festgelegt werden. Auch bei den Code-Transformern existiert die Möglichkeit, daß die Transformation durch einen Transformer die Transformation durch einen weiteren Transformer nötig macht. Dieser Umstand soll jedoch durch die vorgegebene Festlegung der Reihenfolge behoben werden. Dabei kann ein und derselbe Code-Transformer auch mehrfach in der Reihenfolge vorkommen. Aufgrund dieser Festlegung wird die Code-Transformation nicht iterativ durchgeführt.

Der Kompositionsalgorithmus *PROGRESS* berücksichtigt die oben genannten Punkte.

Algorithmus 3.2 (PROGRESS)
Eingabe:

- $\kappa_1, \ldots, \kappa_n \in \mathbb{K}_i$ die Menge der Interface-Transformer,
- $(\xi_1, \ldots, \xi_m) \in \mathbb{K}_c^m$ das geordnete Tupel der Code-Transformer,
- $p \in \mathcal{P}$ das zu transformierende Programm.

Berechnung:

1. do
2. $\quad p' = p;$
3. $\quad p = \kappa_n \circ \kappa_{n-1} \circ \ldots \circ \kappa_1(p');$
4. until $p' = p;$
5. $p = \xi_m \circ \ldots \circ \xi_2 \circ \xi_1(p);$
6. return $p;$

3.4.5 Richtung der Transformation

Die Phase der Interface-Transformation besteht im wesentlichen aus der Erweiterung von Mengen. Einzige Ausnahme bildet die Möglichkeit, Annotationen von Klassen, Methoden und Feldern zu *entfernen*. Kann eine Annotation entweder nur entfernt oder nur hinzugefügt werden, so hat auch hier die Reihenfolge, in der Transformationen durchgeführt werden, keinen Einfluß auf das Ergebnis.

Dagegen stellen solche Annotationen ein Problem dar, deren Entfernung *und* Hinzufügung eine legale Transformation ist. In diesem Fall hat die Reihenfolge der Transformationen einen Einfluß auf das Ergebnis nach einer Iteration, und damit unter Umständen auch auf das Gesamtergebnis. Darüber hinaus kann hier eine unerwünschte Situation auftreten, wenn ein Interface-Transformer κ_1 eine Annotation zu einem Element hinzufügt, und ein anderer Interface-Transformer κ_2 in einer darauffolgenden Iteration diese Annotation wieder entfernt. Die Annahme von κ_1, das transformierte Element sei nun mit der erwünschten Annotation deklariert, wird durch κ_2 zerstört. Die Umkehrung (κ_2 fügt hinzu, was κ_1 entfernt hat) ist gleichermaßen problematisch.

Dieser Mißstand kann durch die Einführung einer *Richtung der Transformation* behoben werden. Die Richtung der Transformation besagt, ob die Annotation eines Elementes entweder entfernt oder hinzugefügt werden darf. Eine solche Richtung kann durch eine *Halbordnung* auf der Menge der Programme $\mathcal{P}$ modelliert werden. Eine Programmtransformation hält genau dann die Richtung der Transformation ein, wenn für das Programm vor der Iteration p_{vor} und das Programm nach der Iteration p_{nach}, $p_{vor} \sqsubseteq p_{nach}$ gilt.

Fazit Durch die Einführung der Transformationsrichtung kann sichergestellt werden, daß die Annahmen der Interface-Transformer nicht verletzt werden. Darüber hinaus stellt sie sicher, daß die Reihenfolge der Interface-Transformationen keinen Einfluß auf das Endergebnis hat. Die Überprüfung der Transformationsrichtung wird im folgenden Abschnitt 3.4.6 in den Kompositionsalgorithmus integriert.

3.4.6 Konflikterkennung

Ein Aspekt, der bisher noch unberücksichtigt blieb, ist die Erkennung von Konflikten durch den Kompositionsalgorithmus. Dem soll nun Rechnung getragen werden.

Die Konflikterkennung beschränkt sich dabei auf die Interface-Transformation, da sich die *Code-Transformation* auf die Modifikation der Methodenimplementationen beschränkt, und Konflikte dabei durch die Vorgabe der Kompositionsreihenfolge vermieden werden können.

Um Konflikte bei der *Interface-Transformation* zu erkennen, hat der Transformationsalgorithmus grundsätzlich zwei Möglichkeiten:

1. Pro Iteration läßt der Algorithmus jede Komponente denselben Programmstand transformieren. Aus dem ursprünglichen Programm vor der Iteration und den von den Komponenten transformierten Programmen können dann die Transformationen der einzelnen Komponenten abgeleitet werden. Diese Transformationen müssen dann auf Konflikte hin überprüft werden. Liegen keine Konflikte vor, oder können alle Konflikte behoben werden, müssen alle diese Transformationen gemeinsam auf das Programm angewendet werden. Das Ergebnis dieser Anwendung ist dann das Programm, das an die nächste Iteration übergeben wird.

2. Der Algorithmus läßt die Komponenten das Programm nicht direkt transformieren. Stattdessen analysieren die Komponenten das Programm nur, und geben dem Kompositionsalgorithmus bekannt, welche Transformationen sie vornehmen möchten. Er sammelt diese Daten, überprüft sie auf Konflikte, löst diese falls möglich auf, und transformiert anschließend das Programm entsprechend.

Im folgenden wird die zweite Möglichkeit weiter verfolgt, da sie den Vorteil bietet, daß die von den Komponenten vorgenommenen Transformationen nicht umständlich aus einem Vergleich von zwei Programmen abgeleitet werden müssen. Desweiteren können vom Kompositionsalgorithmus von vornherein alle Transformationen ausgeschlossen werden, die nicht legal sind.

Eine Interface-Transformer-Komponente ist nun keine Abbildung von einem Programm auf ein Programm mehr, sondern eine Abbildung von einem Programm auf eine Menge von Transformationen:

Definition 3.8 (Interface-Transformer-Komponente) Eine *Interface-Transformer-Komponente* ist eine wohlgeformte Abbildung $\kappa : \mathcal{P} \longrightarrow \mathcal{E}$. Sei $\mathbb{K}_i$ die Menge aller Interface-Transformer-Komponenten.

Konflikte

Die folgende Klasse von Konflikten zwischen den Transformationen zweier Interface-Transformer kann erkannt werden:

- Zwei Felder mit gleichem Namen sollen eingefügt werden.
- Zwei Methoden mit gleicher Signatur sollen eingefügt werden.
- Zwei Klassen mit gleichem Namen sollen eingefügt werden.
- Die direkte Oberklasse einer Klasse soll auf zwei unterschiedliche Klassen geändert werden.

Diese Konflikte werden nun daraufhin untersucht, ob eine Auflösung möglich ist.

Feldkonflikt Sollen von zwei Interface-Transformer-Komponenten zwei Felder mit gleichem Namen eingefügt werden, so kann dieser Konflikt nicht aufgelöst werden. Im Multi-Transformer-Modell weiß im allgemeinen keine Komponente von der Existenz bzw. Aktivität einer anderen Komponente. Wüßten sie etwas darüber, so könnte man annehmen, daß nicht beide Komponenten versuchen, gleichnamige Felder zu einer Klasse hinzuzufügen. Da also eine Komponente nichts von der Aktivität der anderen weiß, kann man davon ausgehen, daß die Felder von den zwei Komponenten für zwei unterschiedliche Zwecke eingefügt werden sollten. Wird ein Feld von zwei Komponenten für verschiedene Zwecke verwendet, ohne daß diese sich der Verwendung des jeweils anderen bewußt sind, so kann es zu unerwartetem Verhalten während der Programmausführung kommen[4].

Methodenkonflikt Beim Einfügen zweier Methoden mit gleicher Signatur, gilt eine ähnliche Argumentation wie beim Feldkonflikt. Da die Implementationen der zwei einzufügenden Methoden nicht einfach gemischt werden können, kann nur eine der beiden verwendet werden, wodurch sich auch hier unerwartete Konsequenzen bei der Programmausführung ergeben können.

Klassenkonflikt Beim Versuch, zwei Klassen mit gleichem Namen einzufügen, gilt ebenfalls das Argument, daß die zwei Klassen im allgemeinen von unterschiedlichen Transformern für unterschiedliche Zwecke eingefügt werden. Ein solcher Konflikt kann daher gleichfalls nicht aufgelöst werden.

[4] In der Implementation des Frameworks (Kapitel 5) werden diese Konflikte dadurch vermieden, daß Transformer *eindeutige* Bezeichner vom Framework anfordern können.

Oberklassenkonflikt Möchte eine Interface-Transformer-Komponente die direkte Oberklasse einer Klasse A auf die Klasse B ändern, und eine andere Komponente auf die Klasse C, so kann dieser Konflikt genau dann aufgelöst werden, wenn B Oberklasse von C ist, oder umgekehrt. Ist B Oberklasse von C, so wird C die neue direkte Oberklasse von A. Ist C Oberklasse von B, so wird B die neue direkte Oberklasse von A.

Stehen die Klassen B und C nicht in einer gegenseitigen Oberklassenbeziehung, so kann der Konflikt nicht aufgelöst werden, da in Java keine *multiple Vererbung* möglich ist.

Fazit Feld-, Methoden- und Klassenkonflikte können zwar erkannt, jedoch nicht aufgelöst werden. Die Transformation sollte in diesem Fall abgebrochen werden. Ein Oberklassenkonflikt kann erkannt, und in bestimmten Fällen aufgelöst werden. Liegt ein nicht auflösbarer Oberklassenkonflikt vor, sollte auch in diesem Fall die Transformation abgebrochen werden.

Der Kompositionsalgorithmus TAU

Der folgende Kompositionsalgorithmus *TAU* beinhaltet nun alle Ideen und Regeln, die in diesem Kapitel gesammelt wurden. Er bildet die Grundlage für eine formale, ausführliche Ausarbeitung des Transformationsmodells in Kapitel 4.

Algorithmus 3.3 (TAU)
Eingabe:

- $\kappa_1, \ldots, \kappa_n \in \mathbb{K}_i$ die Menge der Interface-Transformer,
- $(\xi_1, \ldots, \xi_m) \in \mathbb{K}_c^m$ das geordnete Tupel der Code-Transformer,
- $p \in \mathcal{P}$ das zu transformierende Programm,
- $(\mathcal{P}, \sqsubseteq)$ die Transformationsrichtung.

Berechnung:

1. do // Interface-Transformation
2. for all κ_i do $\delta_i = \kappa_i(p)$;
3. if $\mathit{conflict}(\delta_1, \ldots, \delta_n)$ then return ϵ;
4. $\delta_{ges} = \mathit{merge}(\delta_1, \ldots, \delta_n)$;
5. $p' = p$;
6. $p = \mathit{apply}(\delta_{ges}, p')$
7. if $\neg(p' \sqsubseteq p)$ then return ϵ;
8. until $p' = p$;
9. $p = \xi_m \circ \ldots \circ \xi_2 \circ \xi_1(p)$ // Code-Transformation
10. return p

Dabei haben die verwendeten Abbildungen *conflict*, *merge* und *apply* die folgenden Aufgaben:

- *conflict* erkennt in $\delta_1, \ldots, \delta_n$ die weiter oben beschriebenen Konflikte.
- *merge* faßt $\delta_1, \ldots, \delta_n$ zu einem δ_{ges} zusammen, und nimmt dabei die Konfliktauflösung von Oberklassenänderungen vor.
- *apply* wendet die in δ_{ges} definierten Transformationen auf das Programm p' an.

3.4.7 Beispiele

Im folgenden werden zwei Beispiele für die Transformation von Programmen mit dem Algorithmus *TAU* gegeben.

Beispiel 3.5 Gegeben seien zwei Interface-Transformer:

- κ_{bca} ist eine Variante der *Binary Component Adaptation* [KH98], die zur Klasse `A` die Methode `doSomethingElse` hinzufügt.
- $\kappa_{interface}$ ist ein Transformer, der aus jeder Klasse ihre öffentliche Schnittstelle extrahiert, ein Interface bildet, das genau diese Schnittstelle deklariert, und die Klasse dieses Interface implementieren läßt.

Die Halbordnung, die die Richtung der Transformation bestimmt, ist bei diesem Beispiel irrelevant und kann beliebig gewählt werden.

Gegeben sei zusätzlich das folgende Programm, das nur aus der Klasse `A` besteht. `A` besitzt lediglich die Methode `doSomething`:

```
public class A {
   public void doSomething() { ... }
}
```

Startet man den Algorithmus mit $\{\kappa_{bca}, \kappa_{interface}\}$ und der Klasse `A` als Eingabe, so kann folgender Ablauf beobachtet werden:

1. κ_{bca} gibt *TAU* bekannt, daß es `A` um die Methode `doSomethingElse` erweitern möchte. $\kappa_{interface}$ sieht, daß das Programm nur aus der Klasse `A` besteht, extrahiert deren Interface (das zu diesem Zeitpunkt noch nicht `doSomethingElse` enthält) und bildet daraus das neue Interface `$A$`, das nur die Methode `doSomething()` deklariert. Ferner gibt sie bekannt, daß sie die `implements`-Klausel von `A` um `$A$` erweitern möchte.
2. *TAU* stellt fest, daß keine Konflikte vorliegen, vereinigt die Transformationen von κ_{bca} und $\kappa_{interface}$ und wendet diese auf das Programm an.
3. κ_{bca} hat seine Transformation beendet. $\kappa_{interface}$ sieht, daß `A` nun auch die Methode `doSomethingElse` enthält. Sie gibt daher *TAU* bekannt, daß sie `$A$` um die Methode `doSomethingElse` erweitern möchte.
4. *TAU* stellt fest, daß keine Konflikte vorliegen, und wendet die Transformationen von $\kappa_{interface}$ auf `$A$` an.
5. $\kappa_{interface}$ hat seine Transformation beendet, da sich an `A` nichts geändert hat.

6. Interface-Transformation abgeschlossen.

7. Code-Transformation abgeschlossen.

Die Klasse `A` ist zu der folgenden Klasse `A'` transformiert worden:

```
public class A' implements $A$ {          // geändert in Schritt 1
   public void doSomething() { ... }
   public void doSomethingElse() { ... } // neu      in Schritt 1
}
```

Aus der Transformation ist zusätzlich das Interface `$A$` hervorgegangen:

```
public interface $A$ {                    // neu  in Schritt 1
   public void doSomething();             // neu  in Schritt 1
   public void doSomethingElse();         // neu  in Schritt   3
}
```

Beispiel 3.6 Im folgenden Beispiel wird noch einmal die Klasse `C` aus Abschnitt 3.4.3 aufgegriffen. Es seien die folgenden Transformer gegeben:

- κ_{access} erweitert eine Klassen für jedes nicht primitive Feld um zwei Zugriffsmethoden.
- $\kappa_{counter}$ erweitert eine Klasse für jedes Feld um einen Zugriffszähler.
- ξ_{access} ersetzt jeden direkten Zugriff auf ein nicht primitives Feld durch die von κ_{access} eingefügten Zugriffsmethoden.
- $\xi_{counter}$ fügt vor jedem Zugriff auf ein Feld Code ein, der den von $\kappa_{counter}$ eingefügten Zugriffszähler erhöht.

Die Halbordnung, die die Richtung der Transformation bestimmt, ist bei diesem Beispiel irrelevant und kann beliebig gewählt werden.

Das folgende Programm besteht lediglich aus der Klasse `C`. `C` besitzt ein Feld `b` vom Typ `B` und eine Methode `manipulateB`, die auf `b` zugreift.

```
public class C {
    private B b = new B();
    public void manipulateB() {
        b.doSomething();
    }
}
```

Startet man den Algorithmus mit $\{\kappa_{counter}, \kappa_{access}\}$, $(\xi_{access}, \xi_{counter})$ und der Klasse `C` als Eingabe, so kann folgender Ablauf beobachtet werden:

1. $\kappa_{counter}$ bemerkt, daß `C` das Feld `b` besitzt und gibt daraufhin bekannt, daß sie ein neues Feld `b_counter` einfügen möchte. κ_{access} findet das nicht primitive Feld `b` in der Klasse `C` und gibt daraufhin bekannt, daß sie die Methoden `setB` und `getB` zu `C` hinzufügen möchte.

2. *TAU* stellt fest, daß keine Konflikte vorliegen, vereinigt die Transformationen von $\kappa_{counter}$ und κ_{access} und wendet diese auf das Programm an.

3. $\kappa_{counter}$ und κ_{access} haben ihre Transformation abgeschlossen.
4. Interface-Transformation abgeschlossen.
5. ξ_{access} ersetzt den Zugriff auf `b` in `manipulateB` durch den Aufruf von `getB`.
6. $\xi_{counter}$ fügt in `getB` eine Anweisung hinzu, die den Zähler `b_counter` erhöht.
7. Code-Transformation abgeschlossen.

Die Klasse `C` ist zu der folgenden Klasse `C'` transformiert worden:

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

    private void setB(B b) {        // neu      in Schritt 1
        this.b = b;                 // neu      in Schritt 1
    }                               // neu      in Schritt 1
    private B getB() {              // neu      in Schritt 1
        b_counter++;                // neu      in Schritt     6
        return b;                   // neu      in Schritt 1
    }
    public void manipulateB() {
        getB().doSomething();       // geändert in Schritt   5
    }
}
```

Aus der Transformation sind keine zusätzlichen Interfaces oder Klassen hervorgegangen.

3.5 Zusammenfassung

In dem vorgestellten Konzept für ein Ladezeittransformations-Framework gibt es die Möglichkeit, die Programmtransformationen von mehreren, unabhängig voneinander entwickelten *Transformer-Komponenten* bestimmen zu lassen, die durch den *Kompositionsalgorithmus* miteinander komponiert werden. Dabei können ganze Mengen von Klassen gleichzeitig transformiert werden, sodaß bei der Transformation gegenseitige Abhängigkeiten der Klassen untereinander berücksichtigt werden können.

Durch die Aufteilung des Frameworks in den *Transformations-Manager* und den *Transformationsalgorithmus*, existiert eine saubere Trennung zwischen der Integration in die *Java-Umgebung* bzw. das *Class Loader System* und dem Vorgang der Transformation durch die Transformer. Wegen der Integration des Transformations-Manager-Aufrufs in die Systemklasse `ClassLoader`, werden vom Kompositionsalgorithmus garantiert alle Klassen erreicht, denn `ClassLoader` ist die Oberklasse aller möglichen Class Loader.

Da die Menge der Transformationen, die von einem Transformer durchgeführt werden dürfen, auf die *legalen* Transformationen beschränkt wurde, kann

die Konsistenz des transformierten Programmes bis zu einem gewissen Grad garantiert werden, und grobe Fehler durch unbedachte Transformationen durch Transformer werden damit ausgeschlossen.

Durch die Aufteilung der Transformation in die Phasen *Interface-Transformation* und *Code-Transformation* wurde ausgenutzt, daß die Schnittstellen von Klassen als Mengen aufgefaßt werden können, die Interface-Transformation somit der Manipulation von Mengen entspricht, und daher die Reihenfolge der Operationen beim Vorhandensein einer *Transformationsrichtung* keine Rolle spielt. Aufgrund dessen ist es nicht nötig, die Reihenfolge in der das Programm durch die Interface-Transformer transformiert wird, vorzugeben. Lediglich die Kompositionsreihenfolge der Code-Transformer muß von außen vorgegeben werden, da die Manipulation der Methodenimplementationen als Manipulation von geordneten Listen aufgefaßt werden kann, sodaß die Reihenfolge der Transformationen einen Einfluß auf das Endergebnis hat.

Da die Interface-Transformer im Algorithmus TAU das Programm nicht direkt transformieren, sondern vielmehr ihre erwünschten Transformationen in jeder Iteration dem Kompositionsalgorithmus bekanntgeben, kann dieser die Transformationsmengen der einzelnen Interface-Transformer analysieren und dabei eine bestimmte Klasse von Konflikten erkennen und behandeln.

Kapitel 4

Das formale Transformationsmodell

In diesem Kapitel wird das in Kapitel 3 vorgestellte Konzept für die Ladezeittransformation von Java-Programmen in ein formales Modell überführt. Zu Beginn wird dazu ein algebraisches Modell der Java-Klasse eingeführt. Darauf aufbauend folgt die Definition der Interface- und Code-Transformer-Komponenten. Mithilfe des abschließend eingeführten Kompositionsalgorithmus TAU können diese miteinander komponiert werden.

4.1 Klassen

Die folgenden Definitionen führen zum algebraischen Modell einer Klasse und eines Programmes. Dabei werden die speziellen Eigenheiten des *Classfile Formats* (Abschnitt 2.2) wie z.B. Attribute berücksichtigt. Es werden nur solche Eigenschaften modelliert, die für eine formale Betrachtung von Klassen im Kontext der Transformation wichtig sind. Das Klassenmodell erhebt daher nicht den Anspruch, daß alle möglichen Elemente der *Java Language Specification* [GJSB00] genügen.

Definition 4.1 Sei $\mathcal{B}$ die Menge aller *Klassenbezeichner*. Ein *Klassenbezeichner* ist eine Zeichenkette, die den Namen und das Paket einer Klasse bezeichnet.

Definition 4.2 Sei $\mathcal{S}$ die Menge aller *Signaturen*. Eine *Signatur* ist eine Zeichenkette, zusammengesetzt aus dem Feld- bzw. Methodennamen und dem Feldtyp bzw. der Anzahl und den Typen der Methodenparameter. Die Vorschrift zur Bildung eindeutiger Signaturen wird in der *Java Virtual Machine Specification* [TL99] §4.3.2 und §4.3.3 beschrieben.

Die folgenden Annotationen entsprechen den in der Sprache Java spezifizierten Zugriffsannotationen, ausgenommen der hier neu eingeführten Annotation *package*, die innerhalb dieser formalen Ausarbeitung den *paketweiten* Zugriff auf ein Element repräsentiert.

Definition 4.3 Sei $\mathbb{Z} = \{$*public*, *protected*, *package*, *private*$\}$ die Menge aller *Zugriffsannotationen*.

Definition 4.4 Sei $\mathbb{A} = \{abstract, static, final, native, transient, volatile, strictfp\}$ die Menge aller *sonstigen Annotationen*.

Im Java Classfile Format werden die Methodenimplementationen, Debug-Informationen, die den Opcodes der Methodenimplementationen die ursprünglichen Zeilennummern zuordnen, und andere Informationen in den sogenannten *Attributen* gespeichert. Siehe dazu auch Abschnitt 2.2.

Definition 4.5 Sei $\mathbb{J}$ die Menge der *Java-Attribute*, wie z.B. `ExceptionTable`-Attribute, `LineNumberTable`-Attribute, `Code`-Attribute usw.

Es gibt im Classfile Format genau zwei Attribut-Typen, die die Implementation von Methoden beschreiben. Dies sind die `LineNumberTable`-Attribute und die `Code`-Attribute, die in den folgenden Mengen zusammengefaßt werden:

Definition 4.6 Sei $\mathbb{J}_c \subset \mathbb{J}$ die Menge der `Code`-Attribute und $\mathbb{J}_l \subset \mathbb{J}$ die Menge der `LineNumberTable`-Attribute.

Die folgende Definition faßt die Java-Annotationen und die Java-Attribute zur Menge der *Elementeigenschaften* zusammen.

Definition 4.7 Sei $\mathcal{A} = \mathbb{Z} \cup \mathbb{A} \cup \mathbb{J}$ die Menge der *Elementeigenschaften*.

Eine Methode besitzt eine *Signatur* s, eine Menge von *Elementeigenschaften* a und eine Menge von *Bezeichnern* in der `throws`-Klausel t:

Definition 4.8 Sei $\mathcal{M} = \{(s, a, t) \mid s \in \mathcal{S},\ a \in \mathfrak{P}(\mathcal{A}),\ t \in \mathfrak{P}(\mathcal{B})\}$ die Menge aller *Methoden*.

Ein Feld besitzt eine *Signatur* s und eine Menge von *Elementeigenschaften* a:

Definition 4.9 Sei $\mathcal{F} = \{(s, a) \mid s \in \mathcal{S},\ a \in \mathfrak{P}(\mathcal{A})\}$ die Menge aller *Felder*.

Eine Java-Klasse besitzt einen Bezeichner, eine direkte Oberklasse, implementiert eine Menge von Interfaces und besitzt Methoden und Felder, usw. Dies führt zum algebraischen Modell einer Klasse:

Definition 4.10 Sei $\mathcal{K}$ die Menge aller *Klassen*.

$$
\begin{array}{rll}
\mathcal{K} = \{(b, o, I, F, M, A) \mid & b \in \mathcal{B}, & \text{der Klassenbezeichner} \\
& o \in \mathcal{B} \cup \{\epsilon\}, & \text{der Oberklassenbezeichner} \\
& I \in \mathfrak{P}(\mathcal{B}), & \text{die implementierten Interfaces} \\
& F \in \mathfrak{P}(\mathcal{F}), & \text{die Felder} \\
& M \in \mathfrak{P}(\mathcal{M}), & \text{die Methoden} \\
& A \in \mathfrak{P}(\mathcal{A})\} & \text{die Elementeigenschaften}
\end{array}
$$

Ist $o = \epsilon$, so hat die entsprechende Klasse keine Oberklasse.

Da im folgenden nur solche Klassen betrachtet werden, die keine zwei Methoden oder Felder mit gleicher Signatur besitzen, wird die Menge der *gültigen* Klassen eingeführt. Ihre Elemente besitzen diese gewünschte Eigenschaft.

Definition 4.11 Sei $\mathcal{K}_g \subset \mathcal{K}$ die Menge aller *gültigen Klassen*, also der Klassen, deren Felder und Methoden paarweise verschiedene Signaturen besitzen.

$$\begin{aligned}\mathcal{K}_g = \{k \in \mathcal{K} \mid & (\ f_1, f_2 \in k.F,\ \ f_1.s = f_2.s \ \Rightarrow\ f_1 = f_2) \ \wedge \\ & (\ m_1, m_2 \in k.M,\ \ m_1.s = m_2.s \ \Rightarrow\ m_1 = m_2)\ \}.\end{aligned}$$

Definition 4.12 Sei $\mathcal{P}$ die Menge aller *Programme*, d.h. die Menge aller Mengen von gültigen Klassen, deren Klassenbezeichner paarweise verschieden sind und deren Vererbungsbeziehungen azyklisch sind.

$$\begin{aligned}\mathcal{P} = \{p \in \mathfrak{P}(\mathcal{K}_g) \mid & (\forall\ k_1 \in p :\ \nexists\ k_2 \in p :\ \ k_1.b = k_2.b) \ \wedge \\ & (\forall\ k \in p : k \text{ ist nicht Oberklasse von sich selber})\}.\end{aligned}$$

4.2 Transformer

Um Konflikte zwischen den Transformationen einzelner Transformer effizienter erkennen zu können, wurde im Konzept (Kapitel 3) festgelegt, daß die Interface-Transformer ihre Transformationen nicht selber durchführen, sondern dem Kompositionsalgorithmus lediglich bekanntgeben. Die folgende Definition modelliert die Menge der *Programmerweiterungen*. Eine Programmerweiterung faßt dabei eine beliebige Menge von legalen Transformationen zusammen.

Definition 4.13 Eine *Programmerweiterung* ist ein Tupel, das eine Menge von legalen Transformationen für ein Programm zusammenfaßt. Die Menge aller Programmerweiterungen $\mathcal{E}$ ist wie folgt definiert:

$$\begin{aligned}\mathcal{E} = \{(e_k, e_o, e_i, e_f, e_m, e_t, e_{ek+}, e_{ef+}, e_{em+}, e_{ek-}, e_{ef-}, e_{em-}) \mid\ & e_k \in \mathfrak{P}(\mathcal{E}_{klasse}), \\ & e_o \in \mathfrak{P}(\mathcal{E}_{ober}), \\ & e_i \in \mathfrak{P}(\mathcal{E}_{interface}), \\ & e_f \in \mathfrak{P}(\mathcal{E}_{feld}), \\ & e_m \in \mathfrak{P}(\mathcal{E}_{methode}), \\ & e_t \in \mathfrak{P}(\mathcal{E}_{throws}), \\ & e_{ek+} \in \mathfrak{P}(\mathcal{E}_{kEigenschaft}), \\ & e_{ef+} \in \mathfrak{P}(\mathcal{E}_{fEigenschaft}), \\ & e_{em+} \in \mathfrak{P}(\mathcal{E}_{mEigenschaft}), \\ & e_{ek-} \in \mathfrak{P}(\mathcal{E}_{kEigenschaft}), \\ & e_{ef-} \in \mathfrak{P}(\mathcal{E}_{fEigenschaft}), \\ & e_{em-} \in \mathfrak{P}(\mathcal{E}_{mEigenschaft})\}.\end{aligned}$$

Die Mengen $\mathcal{E}_{klasse}, \mathcal{E}_{ober}, \mathcal{E}_{interface}, ...$ sind dabei wie folgt definiert:

$\mathcal{E}_{klasse}$	$= \{(k) \mid k \in \mathcal{K}_g\}$	Klasse
$\mathcal{E}_{ober}$	$= \{(b, o) \mid b, o \in \mathcal{B}\}$	Oberklasse
$\mathcal{E}_{interface}$	$= \{(b, i) \mid b, i \in \mathcal{B}\}$	Implementiertes Interface
$\mathcal{E}_{feld}$	$= \{(b, f) \mid b \in \mathcal{B},\ f \in \mathcal{F}\}$	Feld
$\mathcal{E}_{methode}$	$= \{(b, m) \mid b \in \mathcal{B},\ m \in \mathcal{M}\}$	Methode
$\mathcal{E}_{throws}$	$= \{(b_k, s, b_e) \mid b \in \mathcal{B},\ s \in \mathcal{S},\ b \in \mathcal{B}\}$	Methoden-throws-Klausel
$\mathcal{E}_{kEigenschaft}$	$= \{(b, a) \mid b \in \mathcal{B},\ a \in \mathcal{A}\}$	Klasseneigenschaft
$\mathcal{E}_{fEigenschaft}$	$= \{(b, s, a) \mid b \in \mathcal{B},\ s \in \mathcal{S},\ a \in \mathcal{A}\}$	Feldeigenschaft
$\mathcal{E}_{mEigenschaft}$	$= \{(b, s, a) \mid b \in \mathcal{B},\ s \in \mathcal{S},\ a \in \mathcal{A}\}$	Methodeneigenschaft

$\delta_\emptyset := (\emptyset, \ldots, \emptyset)$ ist die *leere Erweiterung*.

Die einzelnen Tupel haben folgende Bedeutung:

(k)	$\in \mathcal{E}_{klasse}$	Ergänze die Klasse k.
(b, o)	$\in \mathcal{E}_{ober}$	Setze in der Klasse mit Bezeichner b die direkte Oberklasse auf o.
(b, i)	$\in \mathcal{E}_{interface}$	Füge das Interface i zur `implements`-Klausel der Klasse mit Bezeichner b hinzu.
(b, f)	$\in \mathcal{E}_{feld}$	Füge das Feld f zur Klasse mit Bezeichner b hinzu.
(b, m)	$\in \mathcal{E}_{methode}$	Füge die Methode m zur Klasse mit Bezeichner b hinzu.
(b_k, s, b_e)	$\in \mathcal{E}_{throws}$	Füge den Bezeichner b_e zur `throws`-Klausel der Methode mit der Signatur s in der Klasse mit Bezeichner b_k hinzu.
(b, a)	$\in \mathcal{E}_{kEigenschaft}$	Ergänze bzw. entferne die Elementeigenschaft a zur bzw. von der Klasse mit Bezeichner b.
(b, s, a)	$\in \mathcal{E}_{fEigenschaft}$	Ergänze bzw. entferne die Elementeigenschaft a zur bzw. von dem Feld mit der Signatur s in der Klasse mit dem Bezeichner b.
(b, s, a)	$\in \mathcal{E}_{mEigenschaft}$	Ergänze bzw. entferne die Elementeigenschaft a zur bzw. von der Methode mit der Signatur s in der Klasse mit dem Bezeichner b.

Definition 4.14 Eine *Interface-Transformer-Komponente* ist eine Abbildung

$$\kappa : \mathcal{P} \longrightarrow \mathcal{E}.$$

Sei $\mathbb{K}_i$ die Menge aller Interface-Transformer-Komponenten.

Definition 4.15 Eine *Code-Transformer-Komponente* ist eine Abbildung

$$\xi : \mathcal{P} \longrightarrow \mathcal{P},$$

die nur Methodenimplementationen transformiert. Sei $\mathbb{K}_c$ die Menge aller Code-Transformer-Komponenten.

4.3 Richtung der Transformation

Die Transformationsrichtung wurde eingeführt, damit die Annahmen der Interface-Transformer nicht verletzt werden (siehe Abschnitt 3.4.5). Darüber hinaus stellt sie sicher, daß die Reihenfolge der Interface-Transformationen keinen Einfluß auf das Endergebnis hat.

Die Richtung der Transformation wird formal durch eine *Halbordnung* auf der Menge der Programme modelliert. Im Kompositionsalgorithmus *TAU* wird die Einhaltung der Transformationsrichtung nach jeder Iteration überprüft, indem festgestellt wird, ob das Programm vor der Iteration p_{vor} und nach der Iteration p_{nach} der Halbordnung entspricht, also $p_{vor} \sqsubseteq p_{nach}$ gilt.

Definition 4.16 Die Abbildung $get : \mathcal{P} \times \mathcal{B} \longrightarrow \mathcal{K}_g$ sei wie folgt definiert:

$$get(p,b) = \begin{cases} k \in p \text{ mit } k.b = b & \text{falls } \exists k \in p : \ k.b = b \\ \epsilon, & \text{sonst.} \end{cases}$$

Definition 4.17 Die Abbildung $ober : \mathcal{P} \times \mathcal{B} \longrightarrow \mathfrak{P}(\mathcal{B})$ ermittelt für eine Klasse die gesamte Menge ihrer Oberklassen. Sie ist wie folgt rekursiv definiert:

$$ober(p,b) = \begin{cases} \emptyset & \text{falls } get(p,b) = \epsilon \\ \{get(p,b).o, \ \ ober(p, get(p,b).o)\} & \text{sonst.} \end{cases}$$

Die Halbordnung auf der Menge der Programme wird nun *bottom-up* aufgebaut:

Definition 4.18 Sei $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ eine reflexive *Halbordnung* auf der Menge der Elementeigenschaften $\mathcal{A}$, die nur *legale* Transformationen zuläßt.

Die Halbordnung auf den Elementeigenschaften definiert, welche Annotationen und Attribute entfernt und welche hinzugefügt werden dürfen. Ihre genaue Definition wird an dieser Stelle offen gelassen, und muß erst beim Start des Transformationsalgorithmus *TAU* (siehe Abschnitt 4.5) durch den Benutzer genau festgelegt werden. Dadurch wird eine höhere Flexibilität erreicht. In Abschnitt 4.4 wird ein Beispiel für eine mögliche Halbordnung geben. Abbildung 4.1 auf der nächsten Seite skizziert ihren Aufbau.

Definition 4.19 Definiere $(\mathcal{F}, \leq_{\mathcal{F}})$ eine reflexive *Halbordnung* auf der Menge der Felder $\mathcal{F}$.

$$f_1 \leq_{\mathcal{F}} f_2 \text{ gdw. } f_1.s = f_2.s \quad \text{und} \quad f_1.a \sqsubseteq_{\mathcal{A}} f_2.a.$$

Definition 4.20 Definiere $(\mathcal{M}, \leq_{\mathcal{M}})$ eine reflexive *Halbordnung* auf der Menge der Methoden $\mathcal{M}$.

$$m_1 \leq_{\mathcal{M}} m_2 \text{ gdw. } m_1.s = m_2.s \quad \text{und} \quad m_1.a \sqsubseteq_{\mathcal{A}} m_2.a.$$

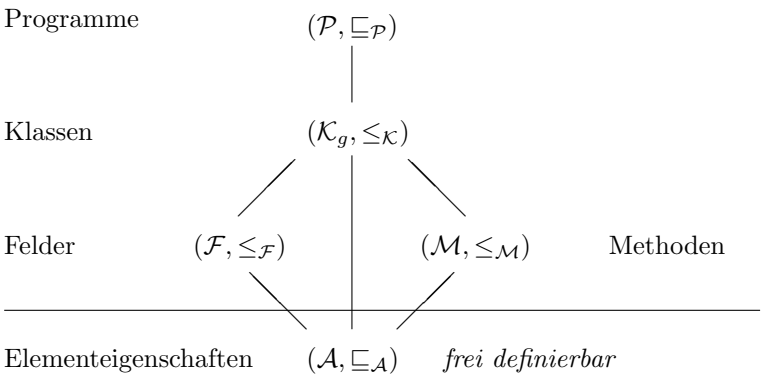

Abbildung 4.1: Aufbau der Halbordnung

Definition 4.21 Definiere $(\mathcal{K}_g, \leq_{\mathcal{K}})$ eine reflexive *Halbordnung* auf der Menge der gültigen Klassen $\mathcal{K}_g$.

$$
\begin{aligned}
k_1 \leq_{\mathcal{K}} k_2 \text{ gdw. } & k_1.b = k_2.b && \text{und}\\
& k_1.I \subseteq k_2.I && \text{und}\\
& \forall f_1 \in k_1.F: \ \exists f_2 \in k_2.F: \ f_1 \leq_{\mathcal{F}} f_2 && \text{und}\\
& \forall m_1 \in k_1.M: \ \exists m_2 \in k_2.M: \ m_1 \leq_{\mathcal{M}} m_2 && \text{und}\\
& k_1.A \sqsubseteq_{\mathcal{A}} k_2.A. &&
\end{aligned}
$$

Definition 4.22 Definiere $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ eine reflexive *Halbordnung* auf der Menge der Programme $\mathcal{P}$.

$$
\begin{gathered}
p_1 \sqsubseteq_{\mathcal{P}} p_2 \text{ gdw.}\\
\forall k_1 \in p_1: \ \exists k_2 \in p_2: \ k_1.b = k_2.b \ \wedge \ k_1 \leq_{\mathcal{K}} k_2 \ \wedge \ \mathit{ober}(k_1) \subseteq \mathit{ober}(k_2).
\end{gathered}
$$

Sie definiert die in Abschnitt 3.4.5 eingeführte *Richtung* der Transformation.

4.4 Beispielhalbordnung

Es folgt ein Beispiel für eine sinnvolle Halbordnung auf der Menge der Elementeigenschaften $\mathcal{A}$. Sie läßt eine Zunahme der Sichtbarkeit von Elementen und das Hinzufügen von Attributen zu Elementen zu. Desweiteren darf die Annotation `native` zu Methoden hinzugefügt, jedoch nicht entfernt werden und die Annotation `transient` darf von Feldern entfernt, jedoch nicht hinzugefügt werden. Alle anderen Transformationen werden nicht zugelassen.

Definition 4.23 Sei $(\mathbb{Z}, \leq)$ eine *Halbordnung* auf der Menge der Zugriffsanno-

tationen $\mathbb{Z}$, sodaß gilt:

$$private \leq protected \leq public$$
$$\text{und}$$
$$private \leq package \leq public.$$

Dabei stehen *package* und *protected* in keiner Relation.

Definition 4.24 (REDUCE) Die Abbildung *REDUCE* faßt eine Menge von Zugriffsannotationen zu einer Zugriffsannotation zusammen, sofern dies möglich ist.

$$REDUCE : \mathfrak{P}(\mathbb{Z}) \longrightarrow \mathbb{Z} \cup \{\epsilon\},$$
$$REDUCE(Z) = \begin{cases} public & \text{falls } public \in Z \\ \epsilon & \text{falls } public \notin Z \ \wedge \ package, protected \in Z \\ protected & \text{falls } public, package \notin Z \ \wedge \ protected \in Z \\ package & \text{falls } public, protected \notin Z \ \wedge \ package \in Z \\ private & \text{sonst.} \end{cases}$$

Definition 4.25 Sei $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ eine Halbordnung auf der Menge der Elementeigenschaften $\mathcal{A}$ wie folgt definiert:

$$\begin{aligned} A \sqsubseteq_{\mathcal{A}} A' \text{ gdw. } REDUCE(A \cap \mathbb{Z}) &\leq REDUCE(A' \cap \mathbb{Z}) \text{ und} \\ (A \setminus \{native, transient\}) \cap \mathbb{A} &= (A' \setminus \{native, transient\}) \cap \mathbb{A} \text{ und} \\ A \cap \{native\} &\subseteq A' \cap \{native\} \text{ und} \\ A \cap \{transient\} &\supseteq A' \cap \{transient\} \text{ und} \\ A \cap \mathbb{J} &\subseteq A' \cap \mathbb{J}. \end{aligned}$$

Bemerkung 4.1 Bei der Übergabe an die JVM kann die effektive Sichtbarkeit eines Elements (Klasse, Methode oder Feld) mittels *REDUCE* aus den Elementeigenschaften ermittelt werden.

4.5 Komposition von Transformationen

Dieser Abschnitt beschreibt den bereits in Kapitel 3 eingeführten Kompositionsalgorithmus *TAU* formal. Dazu wird zu Beginn eine Menge von Abbildungen und Algorithmen definiert, die in *TAU* Verwendung finden.

Definition 4.26 (APPLY) Die Abbildung *APPLY* erweitert ein Programm $p \in \mathcal{P}$ um die in $\delta \in \mathcal{E}$ definierten Programmerweiterungen zu einem Programm $p' \in \mathfrak{P}(\mathcal{K})$.

$$APPLY : \mathcal{P} \times \mathcal{E} \longrightarrow \mathfrak{P}(\mathcal{K}); \qquad (p, \delta) \longmapsto p'.$$

Da durch die Erweiterungen von p um δ eventuell ungültige Klassen bzw. ein ungültiges Programm entsteht, ist p' nicht unbedingt ein Element von $\mathcal{P}$, sondern eventuell nur von $\mathfrak{P}(\mathcal{K})$.

Definition 4.27 (CONFLICT$_n$) Die Abbildung CONFLICT$_n$ überprüft n Erweiterungen darauf, ob zwei oder mehr Erweiterungen ein Programm um dieselbe Klasse, bzw. eine Klasse um dasselbe Feld oder dieselbe Methode erweitern wollen.

$$CONFLICT_n: \ \mathcal{E}^n \longrightarrow \{true, false\}$$

$$(\delta_0, \ldots, \delta_n) \longmapsto \begin{cases} true & \text{falls } \exists i,j \leq n, \ i \neq j: \ \delta_i.e_k \cap \delta_j.e_k \neq \emptyset \\ true & \text{falls } \exists i,j \leq n, \ i \neq j: \ \delta_i.e_m \cap \delta_j.e_m \neq \emptyset \\ true & \text{falls } \exists i,j \leq n, \ i \neq j: \ \delta_i.e_f \cap \delta_j.e_f \neq \emptyset \\ false & \text{sonst.} \end{cases}$$

Definition 4.28 (EXISTS) Die Abbildung *EXISTS* überprüft, ob ein Programm p durch eine Erweiterung δ um ein Element erweitert werden soll, welches bereits im Programm enthalten ist.

$$EXISTS: \mathcal{P} \times \mathcal{E} \longrightarrow \{true, false\}$$

Definition 4.29 (UNION$_n$) Die Abbildung $UNION_n$ faßt n Erweiterungen komponentenweise zu einer Erweiterung zusammen:

$$UNION_n: \mathcal{E}^n \longrightarrow \mathcal{E},$$
$$UNION_n(\delta_1, ..., \delta_n) = (\bigcup_{i=1}^{n} \delta_i.ek, \ \ldots \ , \bigcup_{i=1}^{n} \delta_i.e_{am}).$$

Algorithmus 4.1 (SUPERMERGE)
Der Algorithmus *SUPERMERGE* faßt Oberklassenerweiterungen zusammen, und nimmt dabei eine Konfliktüberprüfung vor.

$$SUPERMERGE: \mathcal{P} \times \mathcal{E} \longrightarrow \mathcal{E} \cup \{\epsilon\}.$$

Sei $p \in \mathcal{P}$ und $\delta \in \mathcal{E}$ die Eingabe, dann werden die Oberklassenerweiterungen in $\delta.e_o$ auf Konflikte überprüft und ggf. zusammengefaßt.

Eingabe:

- $p \in \mathcal{P}$
- $\delta \in Erweiterungen$

Berechnung:

1. *// alle Klassen, deren Oberklasse geändert werden soll* $\rightarrow ID$
2. $ID = \{b \in \mathcal{B} \mid \exists o \in \mathcal{B} : \ (b, o) \in \delta.e_o\}$;
3. $e'_o = \emptyset$;
4. for all $b \in ID$ do begin
5. *// alle neuen Oberklassen von* $b \rightarrow X$
6. $X = \{o \in \mathcal{B} \mid (b, o) \in \delta.e_o\}$;
7. seien $o_1, \ldots, o_n$ die Elemente aus X;
8. *// alle Oberklassen einer neuen Oberklasse* $\rightarrow O_i$
9. for $i = \{1, \ldots, n\}$ do $O_i = ober(p, o_i)$;
10. finde einen Vektor $a = (a_1, \ldots, a_n)$, sodaß $O_{a_1} \subseteq \ldots \subseteq O_{a_n}$;
11. *// speziellste Oberklasse auswählen*
12. if (a existiert) then $e'_o = e'_o \cup \{(b, o_{a_n})\}$;
13. else return ϵ;
14. end;
15. $\delta.e_o = e'_o$;
16. return δ;

Da die Transformation in die Phasen *Interface-Transformation* und *Code-Transformation* aufgeteilt wurde, werden die Methodenimplementationen erst nach den Schnittstellen der Klassen transformiert. Daher dürfen die Methodenimplementationen in der Interface-Transformationsphase nicht sichtbar sein, damit kein Transformer seine Transformation in Abhängigkeit eben dieser (eventuell später modifizierten) Implementationen vornimmt.

Durch die im folgenden definierten Abbildungen $EXTRACT$ und $FILL$ kann die Interface-Transformationsphase im Kompositionsalgorithmus vor den Implementationsdetails der Klassen abgeschirmt werden. Dazu stellen sie die Funktion bereit, die Methodenimplementationen eines Programmes entfernen und später wieder hinzufügen zu können.

Definition 4.30 (EXTRACT,FILL) Die Abbildung $EXTRACT$ extrahiert und entfernt die `Code`- und `LineNumberTable`-Attribute aus den Methoden der Klassen, die in einem Eingabeprogramm p enthalten sind.

$$EXTRACT : \mathcal{P} \longrightarrow \mathcal{P} \times \mathfrak{P}(\mathcal{B} \times \mathcal{S} \times \mathfrak{P}(\mathbb{J}_c \cup \mathbb{J}_l))$$

Die Abbildung *FILL* füllt `Code`- und `LineNumberTable`-Attribute in die Methoden der Klassen, die in einem Eingabeprogramm p enthalten sind.

$$FILL : \mathcal{P} \times \mathfrak{P}(\mathcal{B} \times \mathcal{S} \times \mathfrak{P}(\mathbb{J}_c \cup \mathbb{J}_l)) \longrightarrow \mathcal{P},$$

dabei gilt:

$$\forall p \in \mathcal{P} : \quad FILL(EXTRACT(p)) = p.$$

Algorithmus 4.2 (TAU)
Der *Kompositionsalgorithmus TAU* ist eine Abbildung einer Menge von Interface-Transformer-Komponenten $\kappa_1, \ldots, \kappa_n \in \mathbb{K}_i$, eines geordneten Tupels von Code-Transformer-Komponenten $(\xi_1, \ldots, \xi_m) \in \mathbb{K}_c^m$, einer Halbordnung auf der Menge der Elementeigenschaften $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$ und eines Programmes $p \in \mathcal{P}$ auf ein transformiertes Programm $p' \in \mathcal{P}$ oder das leere Wort ϵ. Dabei wird das Eingabeprogramm p bezüglich der Komponenten $\kappa_1, \ldots, \kappa_n$ und $\xi_1, \ldots, \xi_m$ zum Programm p' transformiert. Tritt dabei ein Konflikt auf, so ist die Ausgabe das leere Wort ϵ.

$$TAU : \mathbb{K}_i^n \times \mathbb{K}_c^m \times \mathcal{P} \longrightarrow \mathcal{P} \cup \{\epsilon\}.$$

Eingabe:

- $\kappa_1, \ldots, \kappa_n \in \mathbb{K}_i$
- $(\xi_1, \ldots, \xi_m) \in \mathbb{K}_c^m$
- $p \in \mathcal{P}$
- $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$

Berechnung:

1. Vervollständige die Halbordnung $(\mathcal{P}, \sqsubseteq_{\mathcal{P}})$ durch $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$;
2. *// entferne die Methodenimplementationen*
3. $(p, Code) = EXTRACT(p)$;
4. do
5. *// alle Programmerweiterungen bestimmen*
6. for $j = \{1, \ldots, n\}$ do $\delta_j = \kappa_j(p)$;
7. *// auf Konflikte prüfen*
8. if $(CONFLICT_n(\delta_1, \ldots, \delta_n))$ then return ϵ;
9. *// Gesamterweiterungsmenge bilden*
10. $\delta_{ges} = UNION_n(\delta_1, \ldots, \delta_n)$;
11. $\delta_{ges} = SUPERMERGE(p, \delta_{ges})$;
12. *// bei Oberklassenkonflikt abbrechen*
13. if $\delta_{ges} = \epsilon$ then return ϵ;
14. *// prüfe, ob Element aus δ_{ges} bereits in p enthalten ist*
15. if $(EXISTS(\delta_{ges}, p))$ then return ϵ;
16. $p' = APPLY(p, \delta_{ges})$;
17. *// wenn kein gültiges Programm, abbrechen*
18. if $p' \notin \mathcal{P}$ then return ϵ;
19. *// wenn Transformationsrichtung nicht eingehalten wird, abbrechen*
20. if $\neg(p \sqsubseteq_{\mathcal{P}} p')$ then return ϵ;
21. $(p, Code') = EXTRACT(p')$;
22. $Code = Code \cup Code'$;
23. until $(\delta_{ges} = \delta_{\emptyset})$;
24. *// Methodenimplementationen wieder ins Programm aufnehmen*
25. $p = FILL(p, Code)$;
26. *// Code-Transformation*
27. $p = \xi_n \circ \ldots \circ \xi_1(p)$;
28. return p;

Das folgende Korollar beweist, daß das Ergebnis der Transformation tatsächlich unabhängig von der Reihenfolge der Interface-Transformer in der Eingabe ist. Das birgt den Vorteil, daß Interface-Transformer ohne weitere Angaben miteinander komponiert werden können, wodurch die Verwendung des Algorithmus vereinfacht wird. Desweiteren wird bewiesen, daß das transformierte Programm und das ursprüngliche Programm tatsächlich entweder in der durch die Halbordnung vorgegeben Beziehung stehen, oder vom Algorithmus das leere Wort ϵ ausgegeben wird.

Korollar 4.1 Der Algorithmus *TAU* besitzt die folgenden Eigenschaften:

1. $p \sqsubseteq_{\mathcal{P}} TAU(p)$ oder $TAU(p) = \epsilon$,
2. das Ergebnis ist unabhängig von der Reihenfolge der Interface-Transformer-Komponenten in der Eingabe, also

   $$\forall p \in \mathcal{P} : \ \forall \sigma \in PERM_n : \ TAU(\kappa_{\sigma_1}, \ldots, \kappa_{\sigma_n}, \xi, p) = TAU(\kappa_1, \ldots, \kappa_n, \xi, p).$$

Dabei ist $PERM_n$ die Gruppe der Permutationen der Zahlen 1 bis n (siehe [Kli92], Definition 3.5.9), die auch als *symmetrische Gruppe* bezeichnet wird.

Beweis Die Eigenschaft 1 gilt aufgrund der Zeilen 18, 20 offensichtlich. Die Eigenschaft 2 gilt ebenfalls, denn die Bildung der δ_i ist unabhängig von der Reihenfolge der κ_i in der Eingabe, da alle κ_i dasselbe Programm p als Eingabe erhalten. Die Abbildung CONFLICT ist per Definition unabhängig von der Reihenfolge der δ_i in der Eingabe und die Abbildung UNION ist eine komponentenweise Mengenvereinigung, die kommutativ ist. □

Kapitel 5

Design und Implementation

Das folgende Kapitel beschreibt das Design und die Implementation des Frameworks. Es wurde auf der *Java 2 Platform, Standard Edition, Version 1.3* [SUN00a] von *Sun Microsystems* entwickelt und in diese integriert.

5.1 Ziele

Beim Design und der Implementation des im Kapitel 4 eingeführten formalen Transformationsmodells wurden primär die folgenden Ziele verfolgt:

- Keine Änderung einer spezifischen JVM-Implementation, sondern eine allgemeingültige Lösung,
- Gewährleistung der Sicherheit,
- Verwendung einer vorhandenen API zur Manipulation von Classfiles,
- Einfache Entwicklung und Einbindung eigener und dritter Transformer-Komponenten.

Durch eine vollständig in Java realisierte Implementation des Frameworks, muß keine Implementation einer JVM geändert werden. Das bietet den Vorteil, daß das Framework auf allen Plattformen, für die eine JVM-Implementation existiert, eingesetzt werden kann, und somit die Plattformunabhängigkeit von Java auch auf das Framework übertragen wird. Trotzdem soll das Framework nicht durch einen eigenen *Class Loader* in den Ladevorgang der Klassen integriert werden, da hierdurch Applikationen, die ebenfalls eigene Class Loader verwenden, nicht oder nur eingeschränkt mit dem Framework funktionieren.

Durch die Verwendung einer bereits vorhandenen API zur Manipulation von Classfiles kann der Aufwand zur Implementation des Frameworks wesentlich verringert werden.

5.2 Design

Der folgende Abschnitt beschreibt das Design des Frameworks. Die Auswahl der verwendeten API zur Manipulation der Klassen beeinflußt sowohl das Design

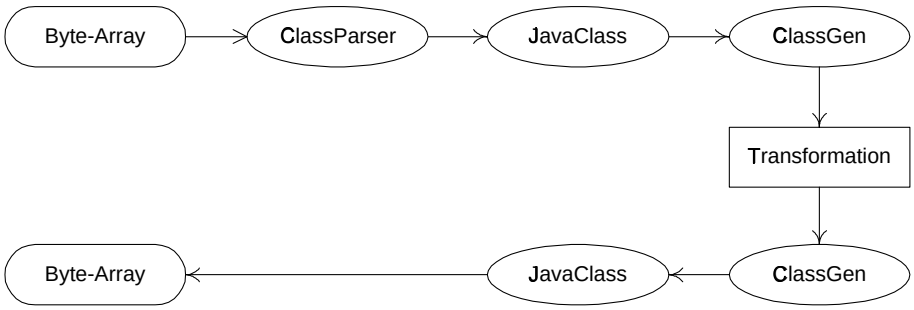

Abbildung 5.1: Mittels der JavaClass API wird aus einem Byte-Array eine *ClassGen-Repräsentation* konstruiert. Diese kann anschließend einfacher transformiert werden und bietet eine Funktion zur Rekonstruktion des entsprechenden Byte-Arrays an.

des Frameworks, als auch die Entwicklung von Transformern, und wird daher zu Beginn behandelt.

Anschließend folgt eine Übersicht über die Pakete des Frameworks, die thematisch verwandte Klassen bündeln. Die zwei Pakete, die die wesentlichen Funktionen des Frameworks beinhalten, werden daraufhin detailliert beschrieben.

5.2.1 Manipulation der Klassen

Die Programmklassen werden vom zuständigen *Class Loader* als Byte-Array im Classfile Format an die JVM übergeben. Zur Transformation der Klassen muß dieses Byte-Array manipuliert werden. Eine direkte Manipulation des Arrays ist zwar theoretisch möglich, jedoch in der Praxis eher umständlich, und konträr zu der Forderung, daß sich die Entwicklung von Transformer-Komponenten möglichst einfach gestalten soll. Da das Classfile Format einen graphenartigen Aufbau besitzt, bietet es sich an, dieses Format in einen Objektgraphen zu überführen, in dem sich die Manipulation einfacher gestaltet.

Auswahl der Bibliothek Dieser Teil muß nicht neu implementiert werden, da es mittlerweile eine größere Auswahl von Klassenbibliotheken gibt, die diese Aufgabe erfüllen. Die bekanntesten, in Java implementierten Bibliotheken sind:

- JOIE: The Java Object Instrumentation Environment,
- Jikes Bytecode Toolkit,
- JavaClass API,
- BIT: Bytecode Instrumenting Tool.

In dieser Arbeit kommt die *JavaClass API* [Dah99b] von Markus Dahm zum Einsatz, deren Quelltext zur Verfügung steht, und die durch regelmäßige Updates weitergepflegt wird. Das *Jikes Bytecode Toolkit* [Laf00] bietet ebenfalls eine mächtige API zur Manipulation von Classfiles an, steht jedoch zur Zeit nur in

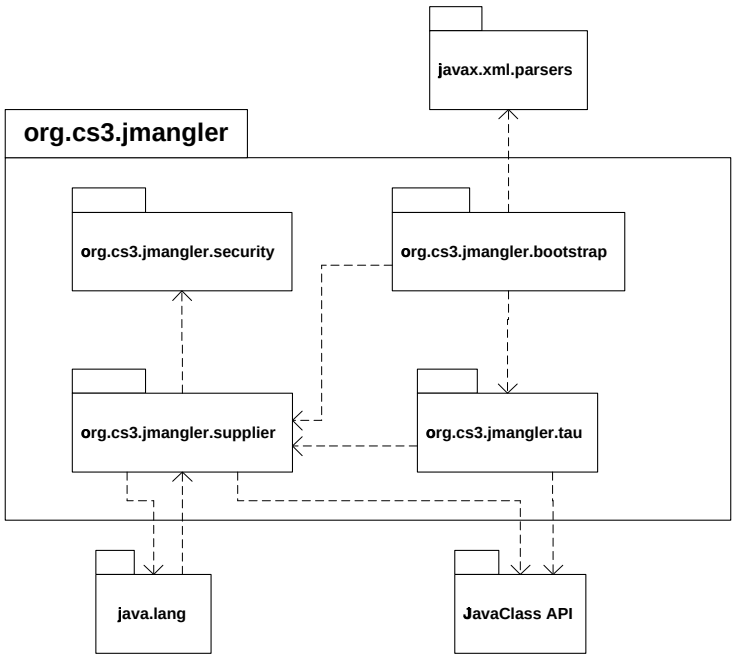

Abbildung 5.2: Paketdiagramm des Frameworks

einer 90-tägigen Evaluationslizenz zu Verfügung. Das *Java Object Instrumentation Environment* [CCK98] und das *Bytecode Instrumenting Tool* [LZ97] sind nur in Beta-Versionen verfügbar und werden nicht mehr weitergepflegt.

JavaClass API In der JavaClass API wird ein Byte-Array im Classfile Format mithilfe der Klasse `ClassParser` in eine Instanz der Klasse `JavaClass` überführt. Eine Instanz dieser Klasse repräsentiert eine Java-Klasse. Aus einem `JavaClass`-Objekt kann dann wiederum eine Instanz der Klasse `ClassGen` generiert werden, die eine komfortablere Schnittstelle zur Manipulation der repräsentierten Klasse besitzt. Aus dieser Repräsentation kann dann wiederum über den Umweg der Klasse `JavaClass` ein Byte-Array im Classfile Format erzeugt werden. Abbildung 5.1 auf der vorherigen Seite gibt einen Überblick über diese Vorgehensweise. Im folgenden wird für Repräsentationen von Klassen bzw. ihren Classfiles der Begriff *ClassGen-Repräsentation* verwendet, wobei jeweils eine entsprechende Instanz der Klasse `ClassGen` gemeint ist. Weitere Details über die Manipulation von Classfiles mithilfe der JavaClass API finden sich in [Dah99a, Dah99b].

5.2.2 Paket-Übersicht

Abbildung 5.2 gibt einen Überblick über die Pakete des Frameworks und deren Abhängigkeiten. Das Framework besteht im wesentlichen aus den Paketen `bootstrap`, `security`, `supplier` und `tau`. Das Paket `security` enthält nur die Klasse `JManglerPermission`, die zur Implementation der Sicherheitsvorkehrungen verwendet wird. Das Paket `bootstrap` enthält die Klasse `Start`, die das

Framework konfiguriert, aktiviert und daraufhin das Laden und Ausführen der Hauptanwendungsklasse initiiert. Das Paket `javax.xml.parsers` enthält einen XML-Parser, der für das Einlesen der Transformer-Konfiguration vom Paket `bootstrap` verwendet wird. Die Verwendung dieser Klassen wird in Abschnitt 5.3 eingehender beschrieben.

Im folgenden werden hauptsächlich die Pakete `java.lang`, `supplier` und `tau` beschrieben, da diese den überwiegenden Teil des Frameworks beinhalten.

Das Paket `java.lang` ist Teil der Java API und enthält Klassen, die fundamental für das Design der Programmiersprache Java sind. Dazu gehört auch die Klasse `ClassLoader`, die Oberklasse aller möglichen *Class Loader* ist. Wie in Abschnitt 3.2.2 bereits skizziert, wird in diese Klasse der Aufruf des Frameworks integriert.

Das Paket `supplier` kapselt im wesentlichen die Komplexität der Class Loaders mit ihren Namensräumen vor dem Transformationsalgorithmus. Dazu bietet es eine Schnittstelle an, über die eine Menge von Klassen in der ClassGen-Repräsentation manipuliert und erweitert werden kann.

Darüber hinaus verwaltet das Paket die ClassGen-Repräsentationen aller Anwendungsklassen und implementiert Sicherheitsvorkehrungen, die die unerlaubte Registrierung eines Objektes als Transformationsalgorithmus verhindert.

Durch die Trennung von Transformationsalgorithmus und der Bereitstellung einer Infrastruktur durch das Paket `supplier`, kann ein Transformationsalgorithmus ohne tiefgehendes Wissen oder besondere Aufmerksamkeit bezüglich des Class Loader Systems entwickelt werden.

Das Paket `tau` setzt auf dem Paket `supplier` auf, und beinhaltet im wesentlichen eine Implementation des in Kapitel 4 eingeführten Kompositionsalgorithmus *TAU* und die Definition der Schnittstellen, die von Transformer-Komponenten implementiert werden müssen.

Bemerkung zur Terminologie Im folgenden wird sowohl der Begriff *Transformationsalgorithmus*, als auch der Begriff *Kompositionsalgorithmus* verwendet. Ein Algorithmus wird als Transformationsalgorithmus bezeichnet, wenn er für die Transformation einer Menge von Klassen verwendet werden kann. Der Kompositionsalgorithmus TAU aus Kapitel 4 ist ein solcher Transformationsalgorithmus, denn zusammen mit seinen aktiven *Transformer-Komponenten* transformiert er eine Menge von Klassen.

5.2.3 Kapselung des Class Loader Systems

Der Kompositionsalgorithmus TAU des formalen Modells aus Kapitel 4 erwartet als Eingabe unter anderem das zu transformierende Java-Programm, also eine Menge von Klassen.

In der Java-Umgebung werden die Anwendungsklassen von den Class Loaders bezogen, die, wie in Abschnitt 2.3 auf Seite 18 beschrieben, eine nicht zu vernachlässigende Komplexität besitzen.

Das Paket `supplier` soll diese Komplexität vor einem Transformationsalgorithmus verbergen, und ihm stattdessen eine einfache Schnittstelle zur Verfügung stellen, die ihm eine zu transformierende Menge von Klassen zugänglich macht. Dazu wird das in Abbildung 5.3 auf der nächsten Seite gezeigte Design verwendet.

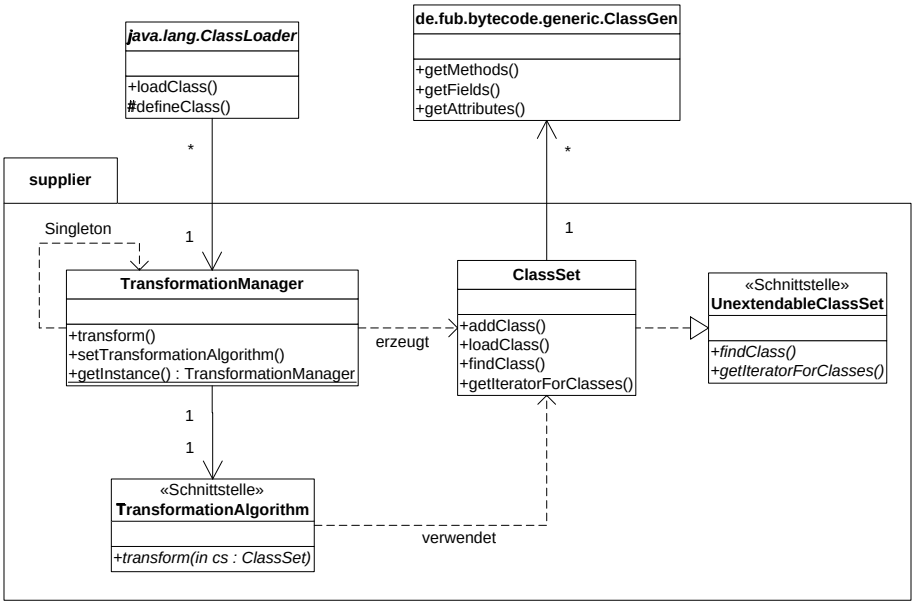

Abbildung 5.3: Das Design des Pakets *supplier*

Instanzen von `de.fub.bytecode.generic.ClassGen` sind die bereits bekannten ClassGen-Repräsentationen von Klassen aus der JavaClass API.

Eine Menge von Klassen wird durch die Klasse `ClassSet` repräsentiert, die eine beliebige Anzahl von Referenzen auf ClassGen-Repräsentationen besitzt. Sie enthält Methoden zum Hinzufügen und Auffinden von Klassen. Ein `ClassSet`-Objekt entspricht damit einem Element aus der Menge der Programme $\mathcal{P}$ im formalen Modell. `ClassSet` implementiert die Schnittstelle `UnextendableClassSet`. Diese enthält nur Methoden zum Auffinden von Klassen, jedoch keine zum Erweitern der Menge. Sie spielt im Paket `tau` eine Rolle.

Damit man ein Objekt als Transformationsalgorithmus verwenden kann, muß seine Klasse die Schnittstelle `TransformationAlgorithm` implementieren, die lediglich die Operation `transform(ClassSet cs)` besitzt, und dem Objekt somit eine Menge von Klassen zur Transformation zur Verfügung stellt. Durch dieses Design kann der zur Transformation verwendete Algorithmus flexibel ausgetauscht werden, was einer Anwendung des *Strategy-Patterns* [GHJV95] entspricht.

Die Verwaltung des zu verwendenden Transformationsalgorithmus übernimmt die Klasse `TransformationManager`. Diese Klasse ist als *Singleton* [GHJV95] implementiert. Dadurch existiert genau eine Instanz dieser Klasse, die im folgenden kurz als *Transformations-Manager* bezeichnet wird. Daß der Transformations-Manager nur einmal existiert, ist sinnvoll, da alle Klassen der Anwendung von demselben Transformationsalgorithmus transformiert werden sollen. Ein Objekt kann sich beim Transformations-Manager als Transformationsalgorithmus registrieren, wenn dessen Klasse die oben erwähnte Schnittstelle `TransformationAlgorithm` implementiert.

Die Systemklasse `ClassLoader` wird so verändert, daß alle Instanzen eine

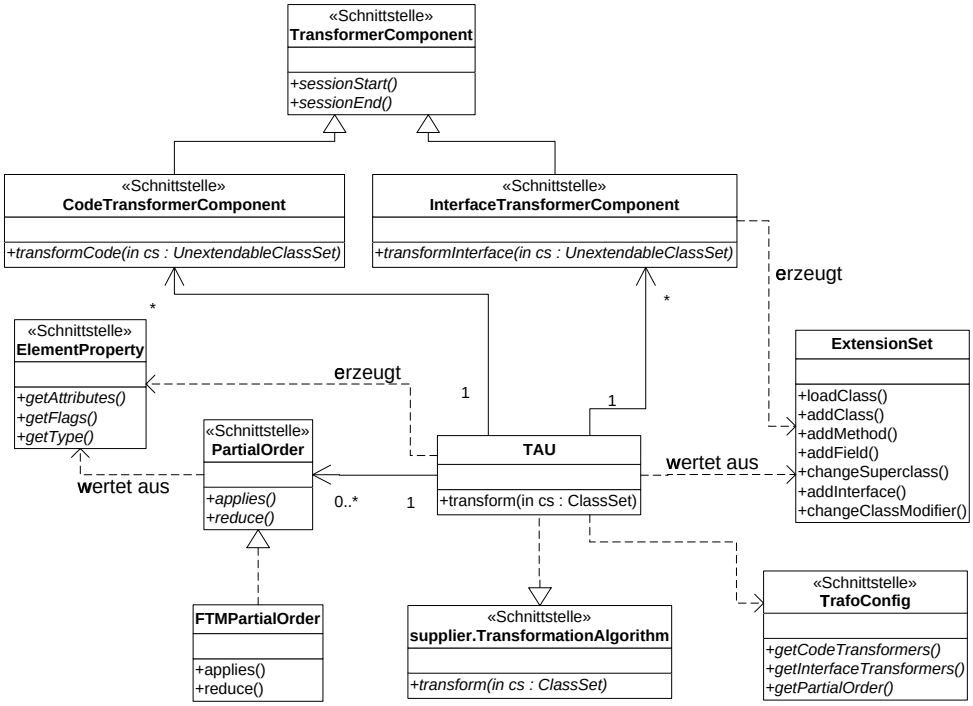

Abbildung 5.4: Das Paket *tau*

Referenz auf den Transformations-Manager besitzen, und die geladenen Klassen vor der Übergabe an die JVM von diesem transformieren lassen, indem sie dessen Methode `transform` aufrufen. Dieser delegiert wiederum die Aufgabe der Transformation an den registrierten Transformationsalgorithmus.

5.2.4 Der Algorithmus TAU

Der im formalen Modell eingeführte Kompositionsalgorithmus TAU spiegelt sich im Paket `tau` wider. Abbildung 5.4 zeigt ein Klassendiagramm dieses Pakets. Im Mittelpunkt steht die Klasse `TAU`, die eine Implementation des Algorithmus TAU ist. Sie implementiert die Schnittstelle `TransformationAlgorithm`, sodaß Instanzen dieser Klasse sich als Transformationsalgorithmus beim Transformations-Manager registrieren können.

Elemente der im formalen Modell eingeführten Menge der *Programmerweiterungen* $\mathcal{E}$ werden durch Instanzen der Klasse `ExtensionSet` repräsentiert. Diese Klasse bündelt somit eine Menge von *legalen Transformationen*.

Die Schnittstellen `CodeTransformerComponent` und `InterfaceTransformerComponent` entsprechen den Code- und Interface-Transformer-Komponenten des formalen Modells. Sie enthalten eine Methode `transformCode` bzw. `transformInterface`, die jeweils ein `UnextenableClassSet`-Objekt übergeben bekommen. Klassen, die `CodeTransformerComponent` implementieren, dürfen die Methodenimplementationen der im `UnextendableClassSet`-Objekt übergebenen Klassen direkt transformieren. Klassen die `InterfaceTransformerComponent`

implementieren, dürfen bei einem Aufruf von `transformInterface` die übergebenen Klassen nicht direkt transformieren, sondern müssen ein `ExtensionSet`-Objekt zurückgeben, daß die erwünschten legalen Transformationen enthält. Da `UnextendableClassSet` keine Methoden zur Erweiterung der Menge um weitere Klassen enthält, müssen auch solche Erweiterungen über das `ExtensionSet`-Objekt realisiert werden.

`TAU` besitzt Referenzen auf `CodeTransformerComponent`-Objekte und `InterfaceTransformerComponent`-Objekte, die es nach der Instanziierung aus einem `TrafoConfig`-Objekt ausliest und für die Transformation der Klassen verwendet. Während einer Iteration der Interface-Transformation sammelt es die von den `InterfaceTransformerComponent`-Objekten zurückgegebenen `ExtensionSet`-Objekte, wertet diese entsprechend Algorithmus 4.2 aus und wendet sie auf das aktuelle `ClassSet`-Objekt an.

Neben den Transformer-Komponenten besitzt `TAU` auch eine Referenz auf ein `PartialOrder`-Objekt. Eine Klasse, die die Schnittstelle `PartialOrder` implementiert, kann als partielle Ordnung fungieren, und bestimmt damit die aus dem formalen Modell bekannte *Richtung* der Transformation. Die Klasse `FTMPartialOrder`[1] ist eine Implementation dieser Schnittstelle.

5.3 Implementation

Dieser Abschnitt beschreibt folgende, nicht-offensichtliche, aber wichtige technische Probleme bei der Umsetzung des beschriebenen Designs:

- Nachladen von Klassen während der Transformation, ohne daß diese dabei bereits an die JVM übergeben werden,
- Zwischenspeichern von Klassen nach der Transformation, wobei die Sichtbarkeiten der Klassen untereinander, die aus den Namensräumen der Class Loaders resultieren, entsprechend berücksichtigt werden müssen,
- Konfiguration und Aktivierung des Frameworks beim Start der JVM,
- Sicherheitsvorkehrungen.

5.3.1 Nachladen von Klassen

Ein wesentlicher Vorteil des Frameworks gegenüber vorhandenen Lösungen ist es, daß mehrere Klassen gleichzeitig und somit in Abhängigkeit voneinander transformiert werden können. Bei der Implementation ergibt sich jedoch das folgende Problem:

Problem Die Klasse `ClassLoader` enthält keine Methode, die nur den Bytecode[2] einer Klasse liefert, ohne daß dieser auch an die JVM übergeben wird, sodaß sich die entsprechende Klasse danach im System befindet. Wenn der Aufruf des Frameworks wie geplant in die Methode `defineClass` integriert wird,

[1] FTM steht für „Formales Transformationsmodell“, da die Implementation der Klasse sich eng an die Vorgaben des formalen Modells hält.

[2] Eine Zeichenkette im Classfile Format wird oft einfach als *Bytecode* bezeichnet, obwohl Bytecode streng genommen die Bezeichnung für eine Sequenz von JVM-Maschinenbefehlen ist.

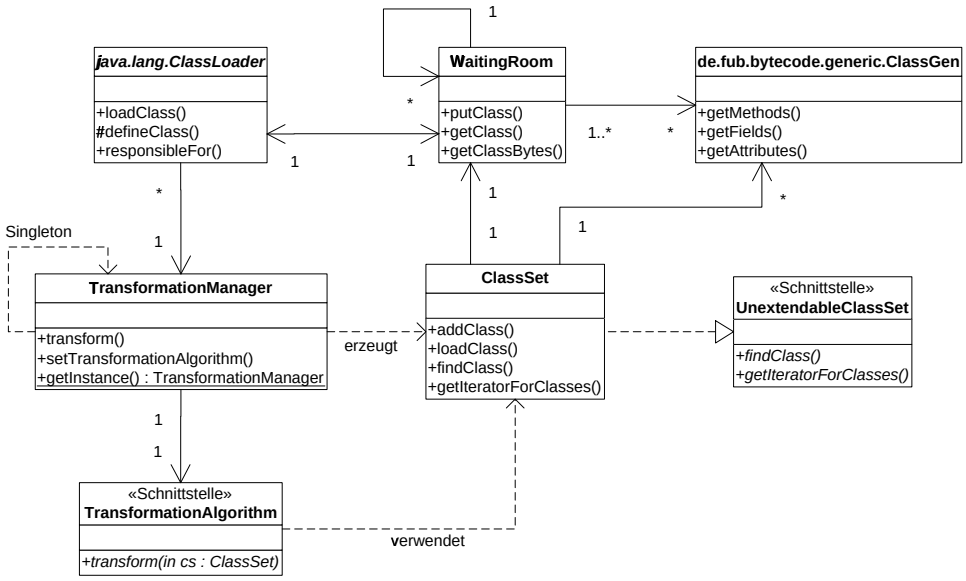

Abbildung 5.5: Das Paket *supplier* wurde für die Implementation um die Klasse WaitingRoom erweitert.

dann steht innerhalb von `defineClass` lediglich der Bytecode der aktuell zu definierenden Klasse für die Transformation zur Verfügung. Sollen während der Transformation dieser Klasse weitere Klassen zum Transformieren nachgeladen werden, so ist dies aufgrund des angesprochenen Fehlens einer allgemeinen Methode für das ausschließliche Laden von Bytecode nicht ohne weiteres möglich. Dabei kommt erschwerend hinzu, daß aufgrund des Delegationsmodells des Class Loader Systems (Abschnitt 2.3 auf Seite 18) im Vorfeld nicht entschieden werden kann, welcher Class Loader eine Klasse bzw. deren Bytecode laden wird.

Lösung Zur Lösung dieses Problems wurde folgender Umstand ausgenutzt: Die Methode `defineClass` in `ClassLoader` löst einen `ClassFormatError`[3] aus, wenn der übergebene Bytecode von der JVM zurückgewiesen wird. Beim Aufruf von `defineClass` steht also der Bytecode der angeforderten Klasse zur Verfügung, dessen Übergabe an die JVM durch das Auslösen eines `ClassFormatError` unterbunden werden kann.

Die Klasse `ClassLoader` wurde nun so erweitert, daß sie sich in einem von zwei möglichen Zuständen befinden kann:

- ClassLoading,
- ByteCodeFetching.

Befindet sie sich im Zustand *ClassLoading*, so verhalten sich alle Instanzen von `ClassLoader` normal, und versuchen bei einem Aufruf von `loadClass` den Byte-

[3] „Thrown when the Java Virtual Machine attempts to read a class file and determines that the file is malformed or otherwise cannot be interpreted as a class file.“, aus [JDK00]

code der angegebenen Klasse zu laden und an die JVM zu übergeben.

Befindet sie sich im Zustand *ByteCodeFetching*, so versuchen ebenfalls alle Instanzen bei einem Aufruf von `loadClass` den Bytecode der angegebenen Klasse zu laden und per `defineClass` an die JVM zu übergeben. Innerhalb von `defineClass` wird jedoch der Bytecode gesichert und ein `ClassFormatError` ausgelöst.

Der Bytecode einer Klasse kann also wie folgt erhalten werden, ohne daß die Klasse an die JVM übergeben wird:

```
public class ClassLoader {
  ...
  private byte[] safeBytes; // Sicherungsfeld
  byte[] fetchByteArray(String name) {
    try {
      setByteCodeFetching(true); // Zustand ändern
      Class c = loadClass(name);
      // Wenn der Kontrollfluß hierher gelangt, ist
      // die Klasse schon in der JVM
      return c.getClassLoader();
    }
    catch(ClassFormatError e) {
      // Bytecode geladen, die entsprechende Klasse
      // befindet sich noch nicht in der JVM
      return safeBytes;
    }
    finally {
      setByteCodeFetching(false); // Zustand ändern
    }
  }
  ...
}
```

5.3.2 Zwischenspeichern von Klassen

Problem Eine weitere zu lösende Aufgabe stellt der Umstand dar, daß während der Transformation eventuell mehrere Klassen transformiert werden, nach Abschluß der Transformation jedoch nur eine Klasse direkt an die JVM übergeben wird. Die anderen Klassen, deren Bytecode in der oben beschriebenen Art und Weise beschafft wurde, müssen also bis zu ihrer Übergabe an die JVM zwischengespeichert werden. Dabei müssen die Namensräume der einzelnen Class Loaders beachtet werden, denn es ist möglich, daß sich zwei Klassen mit demselben Namen gleichzeitig in der JVM befinden, wenn sie sich in unterschiedlichen Namensräumen aufhalten. Die Zwischenspeicherung der Klassen sollte also pro Class Loader, der den Bytecode der Klasse geladen hat, geschehen.

Lösung Als Zwischenspeicher fungiert im Framework die Klasse `WaitingRoom`, die eine Menge von ClassGen-Repräsentationen für genau einen Class Loader verwaltet. Abbildung 5.5 auf der vorherigen Seite zeigt das Klassendiagramm des Pakets `supplier` in der für die Implementation um die Klasse `WaitingRoom`

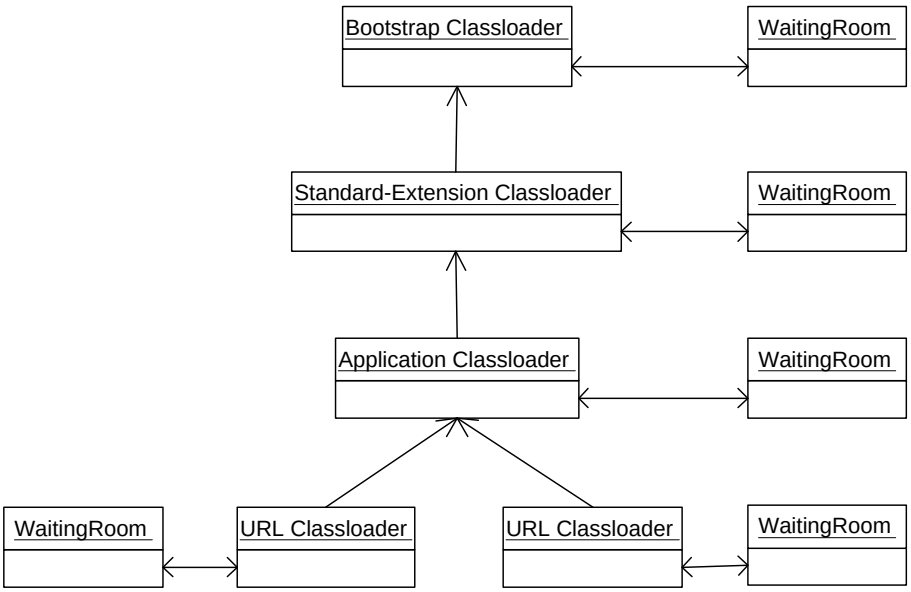

Abbildung 5.6: Class Loader-Hierarchie mit assoziierten *Waitingrooms*. Jedes WaitingRoom-Objekt verwaltet die ClassGen-Repräsentationen der Klassen, die sein assoziierter Class Loader geladen hat.

erweiterten Form. Demnach hat jede Instanz eines Class Loaders genau eine Referenz auf ein `WaitingRoom`-Objekt und umgekehrt. Abbildung 5.6 verdeutlicht diesen Umstand.

Die Klasse `ClassSet` bekommt bei einer Instanziierung ebenfalls eine Referenz auf das `WaitingRoom`-Objekt, dessen Class Loader die initiale Klasse des `ClassSet`-Objektes geladen hat. Soll das `ClassSet`-Objekt um Klassen erweitert werden, so können diese beim `WaitingRoom`-Objekt angefordert werden. Dieses prüft, ob die gewünschte Klasse bereits vorhanden ist und fordert diese gegebenenfalls bei ihrem Class Loader wie oben beschrieben an. Abbildung 5.7 auf der nächsten Seite zeigt ein Sequenzdiagramm, daß diese Aufrufsequenz graphisch beschreibt. Abbildung 5.8 auf Seite 84 verdeutlicht noch einmal das Zusammenspiel der einzelnen Klassen bei der Transformation.

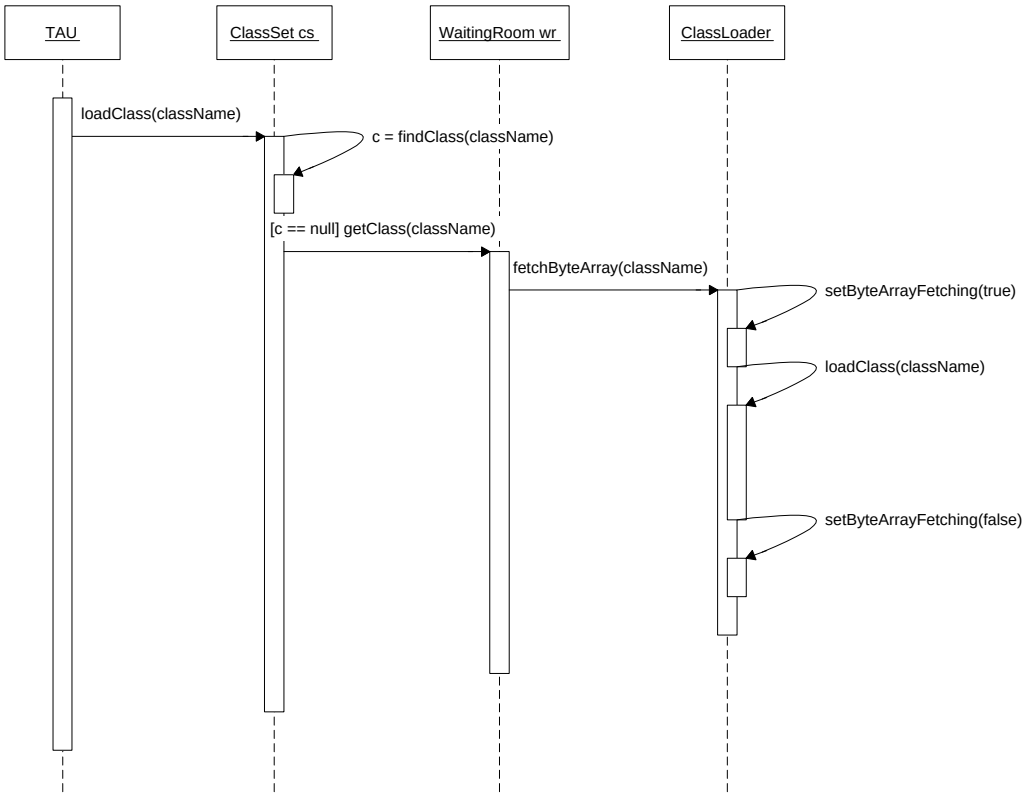

Abbildung 5.7: Dieses Diagramm zeigt das *Nachladen von Klassen*. Bekommt ein ClassSet-Objekt die Nachricht, eine Klasse nachzuladen, wird diese beim aktuellen WaitingRoom-Objekt angefordert. Das WaitingRoom-Objekt delegiert diese Aufgabe an seinen assoziierten Class Loader, indem es ihm die Nachricht *fetchByteArray* schickt. Daraufhin wird der in Abschnitt 5.3.1 auf Seite 79 beschriebene Vorgang gestartet.

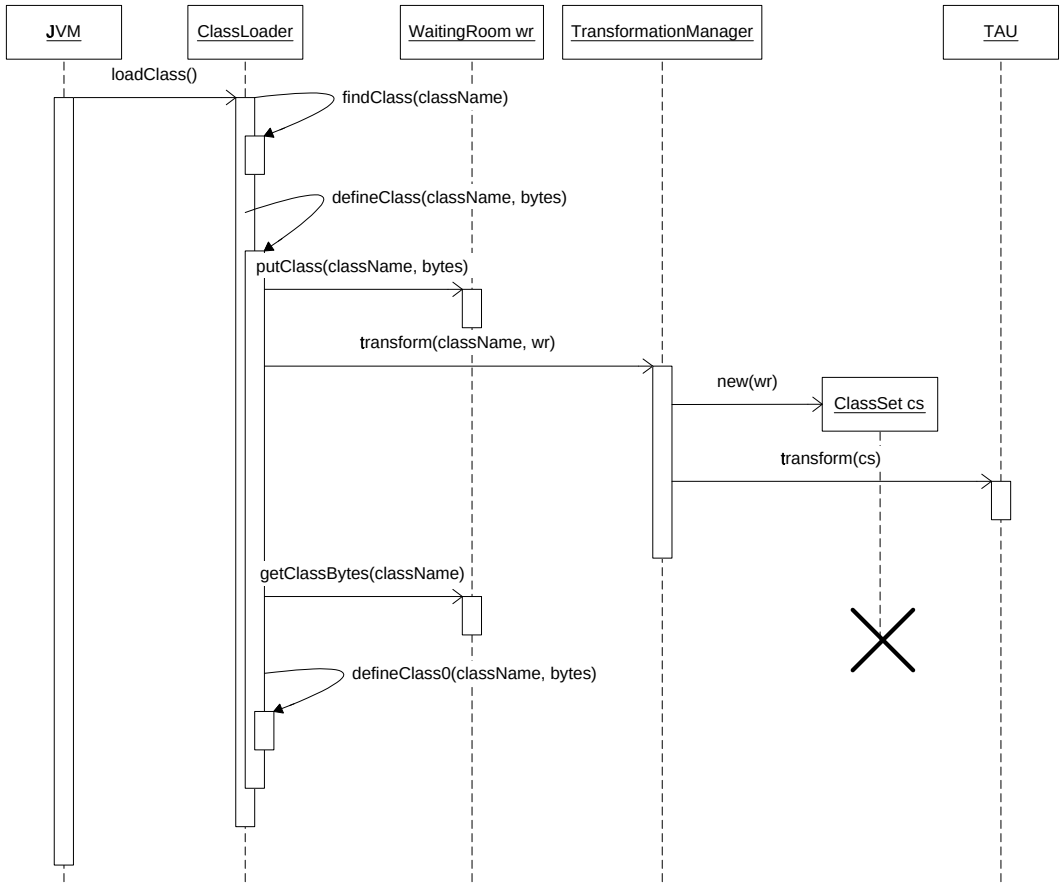

Abbildung 5.8: Dieses Diagramm beschreibt den *Aufruf des Frameworks*. Bekommt ein Class Loader die Nachricht *loadClass*, so lädt er das entsprechende Classfile und übergibt es per *defineClass* scheinbar direkt an die JVM. Die *defineClass*-Methoden in java.lang.ClassLoader wurden jedoch so modifiziert, daß das Classfile erst dem assoziierten WaitingRoom übergeben wird. Anschließend wird der Transformations-Manager aufgefordert, dieses Classfile bzw. dessen ClassGen-Repräsenstation zu transformieren. Abschließend wird das transformierte Classfile per *getClassBytes* wieder beim WaitingRoom angefordert und dann wirklich an die JVM übergeben.

5.3.3 Sicherheitsvorkehrungen

Durch die Integration des Frameworks in die Java-Umgebung entstehen zwei neue Sicherheitslücken, die von einem „Angreifer“ ausgenutzt werden könnten:

1. Registriert sich ein Objekt als Transformationsalgorithmus, so gelangen alle Programmklassen vor der Übergabe an die JVM an dieses Objekt. Ist es einem Angreifer ohne weiteres möglich, sich als Transformationsalgorithmus zu registrieren, so kann er anschließend eventuell vorhandene Sicherheitsvorkehrungen in den Programmklassen entfernen und sich so unter Umständen unbefugten Zugriff auf schützenswerte Objekte verschaffen.

2. Erhält ein Angreifer eine Referenz auf ein `WaitingRoom`-Objekt, so ist er in der Lage, die zwischengespeicherten ClassGen-Repräsentationen der Klassen, die noch nicht an die JVM übergeben wurden, in der oben beschriebenen Art und Weise zu manipulieren, um sich so ebenfalls unbefugten Zugriff auf schützenswerte Objekte zu verschaffen.

Diese zwei Sicherheitslücken werden wie folgt geschlossen:

Punkt 1 wird gelöst, indem in den Implementationen der Methoden `setActive` und `setTransformationAlgorithm` der Klasse `TransformationManager` vor der eigentlichen Aktion überprüft wird, ob der Aufrufer der entsprechenden Methode, laut der aktuellen *Security-Policy* (siehe Abschnitt 2.4 auf Seite 23), das Recht für ihre Ausführung besitzt. Ist dies nicht der Fall, löst der *Security-Manager* eine `SecurityException` aus, sodaß die Ausführung des nachfolgenden Codes abgebrochen wird. Abbildung 5.9 zeigt den Code, der dies sicherstellt.

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
    sm.checkPermission(
       new JManglerPermission("TransformerAlgorithm","set"));
}
```

Abbildung 5.9: Sicherheits-Überprüfung in der Klasse TransformationManager durch den Security Manager.

Punkt 2 wird gelöst, indem nur die Class Loaders die Referenzen auf die `WaitingRoom`-Objekte verwalten, und das entsprechende Feld in `ClassLoader` als `private` deklariert wird. Die anderen Klassen des Pakets `supplier` sind so implementiert, daß keine Referenz auf ein `WaitingRoom`-Objekt nach außen gelangt.

5.3.4 Konfiguration und Aktivierung des Frameworks

Aktivierung Das Framework wird aktiviert, indem die Klasse `Start` aus dem Paket `bootstrap` ausgeführt wird. Deren `main`-Methode konfiguriert das Framework wie weiter unten beschrieben, und aktiviert es, indem es die Methode

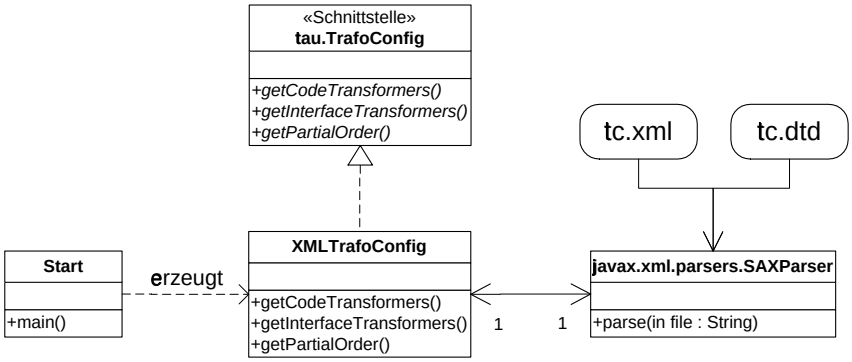

Abbildung 5.10: Verwendung eines XML-Parsers bei der Konfiguration des Frameworks

`setActive` des Transformations-Managers aufruft. Daraufhin lädt sie die eigentlich auszuführende Programmklasse und ruft über die Reflection API deren `main`-Methode auf. Der Name der auszuführenden Programmklasse wird an `Start` über einen Kommandozeilenparameter übergeben.

Konfiguration Für das Einlesen der Konfiguration wird die *Extensible Markup Language*, abgekürzt XML, eingesetzt. Informationen zu diesem Thema sind in [BM98] zu finden.

Im Design des Pakets `tau` wurde bereits beschrieben, daß Instanzen der Klasse `TAU` ihre Konfiguration, also die Code- und Interface-Transformer und die partielle Ordnung, über ein `TrafoConfig`-Objekt beziehen.

In der vorgestellten Implementation des Frameworks wird diese Konfiguration aus einer Datei eingelesen, deren Name beim Start als Kommandozeilenparameter übergeben wird. Diese Datei enthält ein XML-Dokument, das die zu verwendenden Transformer und die partielle Ordnung definiert.

Der Aufbau dieses Dokumentes muß der im Rahmen dieser Arbeit entworfenen *Transformer Configuration Language* (TCL) entsprechen. Sie wurde in Form einer *Document Type Definition* (DTD) beschrieben. Anhang A auf Seite 113 enthält ihre Definition.

Abbildung 5.10 gibt einen Überblick über die implementierte Vorgehensweise. Die Konfigurationsdatei wird von einem XML-Parser eingelesen, der dabei überprüft, ob sie *wohlgeformt* und *gültig* im Sinne der TCL ist. Die eingelesenen Daten werden an eine Instanz der Klasse `XMLTrafoConfig` weitergegeben, die sie auswertet und die entsprechenden Objekte instanziiert. Im Anschluß übergibt die Klasse `Start` dieses `XMLTrafoConfig`-Objekt an das `TAU`-Objekt.

Der XML-Parser wird über die *Java API for XML Parsing* bedient. [Dav00] enthält die Spezifikation dieser API. In der vorliegenden Implementation wurde die Referenzimplementation dieser API von Sun Microsystems verwendet [SUN00b].

5.4 Einschränkung gegenüber dem formalen Modell

Die vorgestellte Implementation des Frameworks hat gegenüber dem formalen Modell aus Kapitel 4 eine wesentliche Einschränkung. Da der Aufruf des Frameworks in die Klasse `ClassLoader` integriert wurde, können die Klassen, die vom *Bootstrap Class Loader* geladen werden, nicht transformiert werden. Der Bootstrap Class Loader lädt alle Klassen, die sich im *Bootstrapclasspath* befinden. Dies sind die sogenannten *Systemklassen*, also z.B. die Klassen der Pakete `java.lang` und `java.io`. Der Bootstrap Class Loader ist Bestandteil der JVM und als solcher in der Regel nativ implementiert. Es gibt keine Möglichkeit, ohne eine Änderung der JVM-Implementation Einfluß auf ihn zu nehmen und sich in den Ladevorgang, so wie bei den in Java implementierten Class Loaders, einzuklinken. Die Änderung einer JVM-Implementation zur Integration des Frameworks wurde jedoch ausdrücklich ausgeschlossen.

Kapitel 6

Evaluation

6.1 Ziele

Laufzeitzusatzkosten Die Ladezeittransformation von Java-Programmen führt zu Laufzeitzusatzkosten während des Ladens der Klassen. Nach dem Laden und Transformieren der Klassen wird deren Bytecode jedoch nach wie vor mit der vollen Geschwindigkeit ausgeführt. Semantische Änderungen der Klassen, die aus Code-Transformationen resultieren, können sich auf die Laufzeit des Programmes sowohl positiv, als auch negativ auswirken. Eine Analyse der Auswirkungen solcher Transformationen ist jedoch nicht Ziel dieses Kapitels und wird nur betrieben, wenn es für die Analyse der aus der Ladezeittransformation resultierenden Zusatzkosten notwendig ist.

Um den Einfluß des Frameworks, der verwendeten Transformer-Komponenten und der Applikationsklassen selber auf die Zusatzkosten zu untersuchen, werden die Laufzeiten der verschiedenen Transformationsphasen und die Gesamtlaufzeit gemessen, sowie die Anzahl der durchgeführten Transformationen ermittelt. Aufgrund dieser Werte kann dann z.B. festgestellt werden, welche Transformationsphasen die meiste Zeit beanspruchen und welche Transformationen teurer als andere sind.

Speicherzusatzkosten Neben den Laufzeitzusatzkosten ist auch ein erhöhter Speicherbedarf zu verzeichnen, der im wesentlichen auf die sich ansammelnden Klassenrepräsentationen und die Instanzen der Framework- und Transformer-Klassen zurückzuführen ist. Für die Evaluation war beabsichtigt, den durchschnittlichen Speicherzusatzaufwand pro transformierter Klasse zu bestimmen. Für die Bestimmung des gesamten zur Verfügung stehenden und des freien Heaps in der JVM, stehen während der Ausführung eines Programmes die Methoden `totalMemory` und `freeMemory` der Systemklasse `java.lang.System` zur Verfügung. `freeMemory` liefert laut Spezifikation der Java-API [JDK00] jedoch nur einen approximativen Wert. Während der Evaluation hat sich dieser als zu ungenau herausgestellt, um eine aussagekräftige Interpretation zuzulassen. Die Tabellen in Anhang B belegen, daß in einigen Tests der mit den oben erwähnten Methoden ermittelte Speicherverbrauch ohne Ladezeittransformation höher ist, als mit Ladezeittransformation (siehe dazu z.B. Tabelle B.11, mit bis zu 44% geringeren Speicherkosten bei *aktivem* Framework). Dieser Umstand läßt sich

auch nicht mit einem von der JVM initiierten *Garbage-Collector*-Lauf erklären, da diese Funktion mit der Kommandozeilenoption `-Xnoclassgc` ausgeschaltet war. Aus diesem Grund werden die Speicherzusatzkosten in diesem Kapitel nicht analysiert.

6.2 Beschreibung der Tests

Der folgende Abschnitt beschreibt die für die Evaluation verwendeten Testapplikationen, Transformer-Komponenten, Konfigurationen des Frameworks und die eingesetzte Plattform.

6.2.1 Transformierte Applikationen

Für die Evaluation wurden Applikationen ausgewählt, die ausschließlich in Java implementiert sind, also keine nativen Methoden enthalten. Da das Framework die Laufzeitzusatzkosten während des Ladens der Klassen erzeugt, hängen die relativen Mehrkosten zum einen von der Anzahl und Größe der zu transformierenden Klassen, und zum anderen von der Laufzeit der Applikation selber ab. Für Applikationen, die aus relativ vielen Klassen bestehen und deren Ausführung eine kurze Laufzeit hat, wird ein verhältnismäßig hoher Laufzeitzusatzaufwand erwartet, wohingegen für Applikationen mit wenig Klassen und einer langen Laufzeit ein geringer relativer Laufzeitzusatzaufwand zu erwarten ist. Tabelle 6.1 zeigt die verwendeten Applikationen. Es wurden ausschließlich „realistische“ Applikationen verwendet. Die Anzahl ihrer Klassen variiert von 24 bis 255, und ihre Laufzeiten auf der verwendeten Plattform von einigen Zehntelsekunden bis über 90 Sekunden, sodaß in den Messungen die oben angesprochenen Effekte zu beobachten sein sollten.

Applikation	**Eingabe**	**Beschreibung**	**geladene Klassen**	**KBytes Bytecode**
javac [SUN00a]	Eigener Quelltext	Java-Kompiler	140	565
JLex [BA00]	sample.lex	Lexical analyser generator	24	85
CUP [HFA+99]	parser.cup	Parser generator	30	147
javadoc [SUN00a]	javac-Quelltext	Dokumentations-Tool	255	936
JGL Benchmark [Obj00]	-	Benchmarks für das JGL-Paket	30	56

Tabelle 6.1: Beschreibung der Benchmark-Applikationen

6.2.2 Eingesetzte Transformer-Komponenten

Neben der Anzahl der Applikationsklassen und der Laufzeit haben auch die im Framework registrierten Transformer-Komponenten einen wesentlichen Einfluß auf die erzeugten Mehrkosten. Die Zeit, die von den einzelnen Transformern für

die Analyse der Programmklassen und die Definition ihrer Transformationen verbraucht wird, kann vom Framework nicht beeinflußt werden. Es kann lediglich die von den Transformern vorgegebenen Transformationen effizient ausführen.

Für die Evaluation werden die in Tabelle 6.2 gezeigten Transformer eingesetzt. Ihre Komplexität reicht von einer einfachen Analyse der direkten Oberklasse beim *ClassGen-Transformer* bis zum ausgiebigen Hinzufügen von Methoden und dem Transformieren vieler Methodenimplementationen beim *Accessor-Transformer*.

Transformer	**Interface**	**Code**	**Beschreibung**
GenericObject	✓		Ändert die Oberklasse aller Klassen von `java.lang.Object` auf `tests.GenericObject`.
Printable	✓		Fügt zu jeder Klasse, die eine Systemklasse als direkte Oberklasse besitzt, das Interface `Printable` und die Methode `print` hinzu, falls nicht bereits vorhanden.
Accessor	✓	✓	Fügt für jedes Feld eine `get-` bzw. `set`-Methode ein und ersetzt alle direkten Zugriffe auf Felder durch die entsprechenden Methodenaufrufe.
Cloneable	✓		Jede Klasse, die `java.lang.Cloneable` implementiert (auch indirekt durch eine Oberklasse), wird um die Methode `copy` ergänzt, deren Rückgabetyp die aktuelle Klasse ist.
SystemExit		✓	Ersetzt `System.exit()` durch das Auslösen einer Ausnahme.

Tabelle 6.2: Beschreibung der verwendeten Transformer-Komponenten

Bis auf den Accessor-Transformer nehmen die vorgestellten Transformer keine semantischen Änderungen an den Klassen vor, die einen meßbaren Einfluß auf die Laufzeiten der Applikationen haben. Die vom Accessor-Transformer durchgeführte Code-Transformation, die jeden Feldzugriff durch den Aufruf einer von ihm erzeugten Zugriffsmethode ersetzt, bewirkt jedoch signifikante Laufzeitzusatzkosten, die, wie in Tabelle 6.3 gezeigt, von 7% bei CUP bis zu 61% beim javac-Kompiler variieren. Dieser Effekt muß bei der Bestimmung der Zusatzkosten, die durch die Ladezeittransformation der Klassen entsteht, berücksichtigt werden, damit die Ergebnisse nicht verfälscht werden.

6.2.3 Testkonfigurationen

Für die Messungen werden die folgenden Transformer-Konfigurationen eingesetzt:

- Empty,
- GenericObject,
- Printable,
- Accessor,
- Cloneable,
- SystemExit,
- Accessor und Cloneable,
- Altogether.

Die Konfiguration *Empty* bedeutet, daß das Framework zwar aktiv ist, jedoch keine Transformer registriert sind. Diese Konfiguration läßt auf die Zusatzkosten rückschließen, die alleine auf das Framework, ohne den Einfluß von aktiven Transformern zurückzuführen sind. Der hier zu erwartende Zusatzaufwand wird im wesentlichen auf das Erzeugen der ClassGen-Repräsentationen aus den Classfiles und zurück, und dem Verwaltungsaufwand des Frameworks zurückzuführen sein.

Die Konfiguration *Altogether* bezeichnet die gleichzeitige Registrierung aller in Tabelle 6.2 angegebenen Transformer.

6.2.4 Zu ermittelnde Werte

Die Implementation des Frameworks wurde für die Evaluation so erweitert, daß während der Ausführung und Transformation einer Applikation die folgenden Werte ermittelt werden:

- Transformationen:
  - Anzahl geänderter Oberklassen,
  - Anzahl hinzugefügter Interfaces,
  - Anzahl hinzugefügter Methoden,
  - Anzahl geänderter Codesequenzen.
- Statistik über Iterationen, Sessions[1] und `ClassSet`-Objekte:
  - Anzahl der Sessions,
  - Anzahl der Iterationen von TAU insgesamt,
  - Minimale Anzahl von Iterationen pro Session,
  - Maximale Anzahl von Iterationen pro Session,
  - Durchschnittliche Anzahl von Iterationen pro Session,
  - Minimale Anzahl von Klassen in einem `ClassSet`-Objekt,
  - Maximale Anzahl von Klassen in einem `ClassSet`-Objekt,
  - Durchschnittliche Anzahl von Klassen in einem `ClassSet`-Objekt.

[1] Mit einer *Session* wird *ein* Aufruf des Transformationsalgorithmus bezeichnet.

- Zeitwerte:
  - Zeit für die Auswertung des Bootstrapclasspath[2],
  - Zeit für das Erzeugen der ClassGen-Repräsentationen aus den Classfiles,
  - Zeit für das Erzeugen der Classfiles aus den transformierten ClassGen-Repräsentationen,
  - Zeit für die Interface-Transformation gesamt,
  - Zeit für die Code-Transformation gesamt,
  - Zeit für die einzelnen Transformer,
  - Zeit für das Einlesen der Transformer-Konfiguration,
  - Gesamtlaufzeit.

6.2.5 Verwendete Plattform

Die folgenden Messungen wurden mit der JVM aus dem *SUN JDK 1.3.0 Beta Refresh* durchgeführt. Jeder Lauf wurde fünf mal auf einem Linux-PC mit Kernel 2.2.10, 200 MHz und 64 MB Ram ausgeführt. Gewertet wurde jeweils die minimale Zeit aus den fünf Messungen. Die JVM bekam durch den Kommandozeilenparameter `-Xms32M` einen *Heap* von ca. 32 MByte zugewiesen, damit eine Beeinflussung der Meßwerte durch das Anfordern von zusätzlichem Speicher durch die JVM vermieden werden konnte.

6.3 Durchführung ohne Ladezeittransformation

Da der Accessor-Transformer, wie oben erwähnt, Laufzeitzusatzkosten erzeugt, die nicht durch die Transformation der Klassen selber, sondern durch die Änderung ihrer Semantik entstehen, müssen die Referenzwerte für die Applikationen jeweils einmal für die originalen Klassen und einmal für die vom Accessor-Transformer transformierten Klassen bestimmt werden. Tabelle 6.3 zeigt die gemessenen Werte und die aufgetretenen Differenzen.

[2]Das Framework stellt den Transformern eine Utility-Klasse zur Verfügung, die Auskunft darüber erteilt, ob es sich beim Namen einer Klasse um den Namen einer Systemklasse handelt. Diese Information kann für Transformer wichtig sein, da das Framework die Transformation von Systemklassen nicht ermöglicht. Vergebliche Versuche von Transformern, Systemklassen in das aktuelle `ClassSet`-Objekt zu laden, können durch die Verwendung der angesprochenen Utility-Klasse vermieden werden. Diese Klasse wiederum wertet bei ihrer ersten Verwendung den *BootstrapClasspath* aus, um im Vorfeld die Namen der Systemklassen zu bestimmen.

Applikation	original	transformiert	Differenz	Zusatzkosten
javac	26583	42814	16231	61%
JLex	2121	2636	515	24%
CUP	3594	3836	242	7%
javadoc	90265	98334	8069	9%
JGL Benchmark	33231	39624	6393	19%

Tabelle 6.3: Zeitdifferenz zwischen Laufzeit der originalen Applikation und der durch den Accessor-Transformer transformierten Applikation (Zeitangaben sind in der Einheit *msec* notiert)

Die Spalte *original* zeigt die Laufzeit der jeweiligen Applikation ohne Modifikation der Klassen und ohne aktives Framework. Die Spalte *transformiert* zeigt die Laufzeit der jeweiligen Applikation, deren Klassen zuvor vom Framework mit aktivem Accessor-Transformer transformiert, anschließend auf der Festplatte zwischengespeichert, und dann für die Zeitmessung erneut ohne aktives Framework ausgeführt wurden, wobei die zwischengespeicherten, transformierten Klassen verwendet wurden. Die Spalte *Zusatzkosten* zeigt die relativen Zusatzkosten an, die durch die angesprochenen semantischen Änderungen der Applikationsklassen entstanden.

Für die folgenden Zeitmessungen mit aktivem Framework und Ladezeittransformation wird dann jeweils der Referenzwert für die originalen oder die Accessor-transformierten Klassen verwendet, je nachdem, ob der Accessor-Transformer aktiv ist oder nicht.

6.4 Durchführung mit Ladezeittransformation

Im folgenden wird nur ein zusammenfassender Überblick über die gesammelten Werte gegeben, da eine ausführliche Auflistung aller Ergebnisse den Umfang dieses Kapitels sprengen würde. Anhang B enthält Tabellen mit allen in Abschnitt 6.2.4 aufgeführten zu ermittelnden Werten.

6.4.1 Konstante Kosten

Beim Ausführen der Applikationen mit aktivem Framework werden für das Einlesen und Auswerten der Transformer-Konfiguration und für das Auswerten des *Bootstrapclasspath* in etwa konstante Laufzeitkosten erzeugt, die weitestgehend unabhängig von der Anzahl der Klassen und den aktiven Transformern sind. Die folgende Tabelle zeigt den durchschnittlichen Laufzeitzusatzaufwand, der auf diese Programmteile zurückzuführen ist:

Programmteil	Zusatzkosten
Einlesen der Konfiguration	1235
Auswerten des Bootstrapclasspath	2046

Tabelle 6.4: Konstante Zusatzkosten (Zeitangaben sind in der Einheit *msec* notiert)

6.4.2 Relative und absolute Zusatzkosten

Die folgenden Tabellen geben die relativen und absoluten Laufzeitzusatzkosten für die jeweilige Konfiguration an, die aus der Ladezeittransformation der Klassen durch das Framework und der Transformer resultieren. Die Tabellen 6.5 und 6.6 geben die gesamten Zusatzkosten an, die Tabellen 6.7 und 6.8 die Zusatzkosten ohne den in 6.4.1 erwähnten konstanten Anteil, der aus dem Einlesen der Transformer-Konfiguration und der Analyse des Bootstrapclasspath resultiert.

Applikation	**Referenzwert**	**Empty**		**GenericObject**		**Printable**		**Cloneable**		**SystemExit**	
javac	26,6	18%	4,9	48%	12,8	26%	7,1	25%	6,9	21%	5,7
JLex	2,1	111%	2,4	329%	7,0	219%	4,7	213%	4,5	113%	2,4
CUP	3,6	72%	2,6	203%	7,3	134%	4,8	126%	4,6	104%	3,8
javadoc	90,3	14%	13,5	22%	20,6	16%	14,4	16%	14,6	14%	13,6
JGL Benchmarks	33,2	6%	2,2	12%	4,3	13%	4,5	13%	4,6	7%	2,6

Tabelle 6.5: Relative und absolute Laufzeitzusatzkosten **mit** konstantem Anteil (Zeitangaben sind in der Einheit *sec* notiert)

Applikation	**Referenzwert**	**Accessor**		**Accessor u. Cloneable**		**Altogether**	
javac	42,8	65%	28,0	66%	28,5	84%	36,3
JLex	2,6	425%	11,2	423%	11,2	585%	15,4
CUP	3,8	268%	10,3	267%	10,3	389%	15,0
javadoc	98,3	43%	42,8	43%	43,0	53%	53,0
JGL Benchmarks	39,6	17%	7,0	18%	7,5	23%	9,2

Tabelle 6.6: Relative und absolute Laufzeitzusatzkosten **mit** konstantem Anteil (Zeitangaben sind in der Einheit *sec* notiert)

Applikation	Referenzwert	Empty		GenericObject		Printable		Cloneable		SystemExit	
javac	26,6	13%	3,7	43%	11,6	14%	3,9	13%	3,7	16%	4,4
JLex	2,1	55%	1,2	271%	5,8	66%	1,4	59%	1,3	54%	1,2
CUP	3,6	39%	1,4	169%	6,1	45%	1,6	37%	1,3	70%	2,5
javadoc	90,3	13%	12,3	21%	19,4	12%	11,2	12%	11,3	13%	12,3
JGL Benchmarks	33,2	3%	1,0	9%	3,1	3%	1,3	3%	1,3	3%	1,3

Tabelle 6.7: Relative und absolute Laufzeitzusatzkosten **ohne** konstantem Anteil (Zeitangaben sind in der Einheit *sec* notiert)

Applikation	Referenzwert	Accessor		Accessor u. Cloneable		Altogether	
javac	42,8	57%	24,7	59%	25,3	77%	33,0
JLex	2,6	308%	8,1	298%	7,9	459%	12,1
CUP	3,8	184%	7,1	182%	7,0	301%	11,6
javadoc	98,3	39%	38,9	40%	39,7	50%	49,6
JGL Benchmarks	39,6	9%	3,8	10%	4,2	14%	5,9

Tabelle 6.8: Relative und absolute Laufzeitzusatzkosten **ohne** konstantem Anteil (Zeitangaben sind in der Einheit *sec* notiert)

Wie erwartet variieren die relativen Zusatzkosten sehr stark und liegen (inkl. der konstanten Kosten), je nach Applikation und Konfiguration, zwischen 6% (JGL bezüglich der Konfiguration *Empty*) und 585% (JLex bezüglich der Konfiguration *Altogether*). Die relativen Zusatzkosten sind bei den *JGL Benchmarks* immer am niedrigsten, da diese Applikation nur aus 30 Klassen besteht, und trotzdem eine verhältnismäßig lange Laufzeit von über 33 Sekunden besitzt. Selbst bei der Konfiguration *Altogether* fallen die Zusatzkosten von mehr als 9 Sekunden durch die Ladezeittransformation nur mit 23% ins Gewicht. Zusätzlich zu der geringen Anzahl von Klassen, besitzen diese gemeinsam lediglich eine Größe von 56 KBytes. Die durchschnittliche Größe beträgt somit nur gut 1,8 KByte, wohingegen bei allen anderen Applikationen dieser Wert mindestens doppelt so hoch liegt. Dadurch sind die Kosten für das Erzeugen der ClassGen-Repräsentationen und für das Analysieren und Transformieren der Klassen im Schnitt deutlich geringer als bei allen anderen getesteten Applikationen.

Die höchsten relativen Zusatzkosten sind für *JLex* zu verzeichnen. Trotz der verhältnismäßig geringen Anzahl von nur 24 geladenen Klassen, belaufen sich bei der Konfiguration *Altogether* die Zusatzkosten auf 585%. Das liegt an der sehr kurzen Laufzeit von JLex, die gerade einmal 2,1 Sekunden beträgt, sodaß der Zusatzaufwand von 12,1 Sekunden für die Ladezeittransformation stark ins

Gewicht fällt. Diese Werte bestätigen die zu Beginn geäußerten Erwartungen.

Neben der Applikationslaufzeit und der Anzahl und Größe der geladenen Klassen, bestätigen die Meßwerte ebenfalls, daß die Zusatzkosten auch mit der Komplexität der Transformationen zunehmen. Je mehr Transformer-Komponenten aktiv sind, und je höher der Aufwand für Analyse und Transformation der Klassen wird, desto größer werden die Zusatzkosten. So entstehen für die Ladezeittransformation der Applikation *javadoc* lediglich 13% Mehrkosten bei einer Transformation durch den *SystemExit-Transformer*. Demgegenüber stehen 39% Mehrkosten bei einer Ladezeittransformation durch den *Accessor-Transformer*.

Sei nun $p \in \mathcal{P}$ die betrachtete Applikation, t_{ohne} deren Laufzeitkosten ohne Ladezeittransformation, C_{konst} seien die in 6.4.1 betrachteten konstanten Zusatzkosten, $\kappa_1, \ldots, \kappa_n \in \mathbb{K}_i$ die aktiven Interface-Transformer und $(\xi_1, \ldots, \xi_m) \in \mathbb{K}_c^m$ die aktiven Code-Transformer, dann läßt sich für die relativen Laufzeitzusatzkosten C_{rel} zusammenfassend die folgende Relation aufstellen:

$$C_{rel} \sim \frac{C_{konst} + [g(p) \cdot k(\{\kappa_1, \ldots, \kappa_n\}, (\xi_1, \ldots, \xi_m))]}{t_{ohne}}$$

Dabei sei die Abbildung g ein Maß für die Größe und Anzahl der Programmklassen und k ein Maß für die Komplexität der Transformer-Komponenten.

Aus obiger Relation ist ersichtlich, daß mit zunehmender Laufzeit t_{ohne} die relativen Zusatzkosten immer geringer werden.

6.4.3 Durchgeführte Transformationen

Tabelle 6.9 zeigt die von den einzelnen Transformer-Komponenten durchgeführten Transformationen. Daraus ist z.B. ersichtlich, daß die Zusatzkosten von 28 Sekunden bzw. 65% für die Transformation des `javac`-Kompilers durch den Accessor-Transformer daraus resultieren, daß 1000 Methoden hinzugefügt und 8602 Codesequenzen transformiert werden.

		GenericObject			Printable			Accessor			Cloneable			SystemExit
Applikation		Oberklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Oberklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Oberklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Oberklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert
javac		48	0	0	0	0	51	51	0	0	0	1000	8602	0	0	0	0	2
JLex		23	0	0	0	0	23	23	0	0	0	326	2810	0	0	0	0	0
CUP		20	0	0	0	0	21	21	0	0	0	158	870	0	0	0	0	7
javadoc		52	0	0	0	0	61	58	0	0	0	1070	7852	0	0	0	0	3
JGL Benchmarks		16	0	0	0	0	19	19	0	0	0	62	695	0	0	0	0	0

Tabelle 6.9: Durchgeführte Transformationen der jeweiligen Transformer-Komponenten.

6.4.4 Kosten für Oberklassenänderungen

Die folgende Tabelle 6.10 zeigt die gesamte Laufzeit der Interface-Transformationsphase bezüglich der Konfiguration GenericObject und die Laufzeit des GenericObject-Transformers alleine.

Applikation	**Interface-Transf. ges.**	**GenericObject**	**Superklassen geändert**
javac	7953	61	48
JLex	4778	26	23
CUP	4738	20	20
javadoc	7882	90	52
JGL Benchmarks	2181	18	16

Tabelle 6.10: Laufzeiten der Interface-Transformation insgesamt und des GenericObject-Transformers im Vergleich (Zeitangaben sind in der Einheit *msec* notiert)

Es fällt auf, daß für die Interface-Transformation insgesamt eine wesentlich größere Laufzeit zu verzeichnen ist, als für die Ausführung des GenericObject-Transformers alleine, obwohl dieser in der betrachteten Konfiguration die einzige registrierte Transformer-Komponente ist. Dieser Umstand zeigt, daß die Durchführung der vom GenericObject-Transformer vorgegebenen Oberklassen-Transformationen sehr *teuer* ist. Der Grund dafür liegt in der aufwendigen Analyse und Modifikation aller Methoden der transformierten Klasse, da alle `super`-Aufrufe transformiert werden müssen. Dieser Aufwand ist im Vergleich zu den anderen legalen Transformationen überdurchschnittlich hoch, da z.B. bei der Konfiguration *Accessor* die Laufzeit des *Accessor-Transformers* in etwa zwei Drittel der gesamten Interface-Transformation beträgt (siehe dazu die Tabellen B.13 und B.14 im Anhang).

6.4.5 Kosten für das Erzeugen von Klassenrepräsentationen

Da die direkte Manipulation der Klassen im Classfile Format zwar möglich, jedoch eher umständlich, fehleranfällig und kompliziert ist, wurde im Design des Frameworks eine Abbildung dieses Formats auf die in 5.2.1 eingeführte ClassGen-Repräsentation gewählt. Die folgende Tabelle zeigt die Laufzeitzusatzkosten, die durch diese Abbildung und die Erzeugung von Classfiles aus den transformierten ClassGen-Repräsentation resultiert.

Applikation	**Anzahl Klassen**	**Durchschnittliche Klassengröße**	**ClassGen nach Classfile gesamt**	**ClassGen nach Classfile pro Klasse**	**Classfile nach ClassGen gesamt**	**Classfile nach ClassGen pro Klasse**	**Anteil an Zusatzkosten bzgl. Empty**	**Anteil an Zusatzkosten bzgl. Altogether**
javac	140	4,0	890	6,3	2123	15,1	68%	8%
JLex	24	3,5	241	10,0	801	33,4	49%	6%
CUP	30	4,9	327	10,9	990	33,0	54%	8%
javadoc	255	3,7	1172	4,6	3278	12,9	34%	9%
JGL Benchmarks	30	1,9	192	6,4	684	22,8	43%	8%
Mittel		*3,6*		*7,6*		*23,4*	*50%*	*8%*

Tabelle 6.11: Kosten für das Erzeugen von ClassGen-Repräsentationen aus Classfiles und zurück (Zeitangaben sind in der Einheit *msec* notiert, die durchschnittliche Klassengröße in *KByte*)

Die Zusatzkosten belaufen sich bei einer durchschnittlichen Klassengröße von 3,6 KBytes im Classfile Format auf 23,4 msec für das Abbilden des Classfiles auf die entsprechende ClassGen-Repräsentation und 7,6 msec für das erneute Erzeugen eines Classfiles aus der transformierten ClassGen-Repräsentation. Die Kosten für das Erzeugen der ClassGen-Repräsentationen sind wesentlich höher als für das Erzeugen der Classfiles, da zusätzlich zum Parsen der Classfiles auch die entsprechenden Objekte für den Objektgraphen instanziiert werden müssen. Beim abschließenden Erzeugen der Classfiles entfallen diese Kosten, sodaß dieser Verarbeitungsschritt nur etwa ein Drittel der Zeit beansprucht.

Insgesamt variiert der Anteil für das Erzeugen von Klassenrepräsentationen je nach Konfiguration zwischen etwa 50%, wenn gar keine Transformer-Komponente aktiviert wurde und etwa 8% bei der Konfiguration *Altogether*. Mit zunehmender Komplexität der Ladezeittransformation wird also auch dieser Anteil an den Gesamtzusatzkosten immer geringer.

6.5 Ausblick

Der folgende Abschnitt gibt einen Überblick über optionale Erweiterungen und Änderungen der in Kapitel 5 vorgestellten und in diesem Kapitel evaluierten Implementation des Frameworks, um es im Hinblick auf die durch die Ladezeit-

transformation entstehenden Zusatzkosten zu optimieren.

6.5.1 Verwendung eines anderen XML-Parsers

Aus Abschnitt 6.4.1 geht hervor, daß schon beim Start des Frameworks gut eine Sekunde für das Einlesen und Auswerten der Transformer-Konfiguration verbraucht wird. Dieser Wert wird zwar mit zunehmender Rechnergeschwindigkeit immer geringer, er wirkt sich jedoch gerade bei Applikationen mit einer sehr kurzen Laufzeit sehr negativ auf die relativen Zusatzkosten aus.

In der in Kapitel 5 vorgestellten Implementation des Frameworks wird die von Sun Microsystems zur Verfügung gestellte Referenzimplementation der *Java API for XML Parsing* [Dav00] verwendet. Es existieren jedoch mittlerweile einige weitere Implementationen dieser Spezifikation, wie z.B. der *XML Parser for Java* von IBM Alphaworks [Alp00] oder der *Xerces Java Parser* von der Apache Software Foundation [Apa00]. Durch die Verwendung eines anderen XML-Parsers können die konstanten Zusatzkosten eventuell gesenkt werden.

Sollten auch diese Anpassungen nicht zu einer Kostenoptimierung führen, kann auch ein proprietäres Format zur Konfiguration des Frameworks entwickelt und verwendet werden, wodurch natürlich die Vorteile eines Einsatzes von XML verloren gehen.

6.5.2 Keine Erzeugung von Klassenrepräsentationen

In der getesteten Implementation des Frameworks werden die ClassGen-Repräsentationen der geladenen Klassen unabhängig davon erzeugt und gesammelt, ob überhaupt eine Transformer-Komponente registriert ist. Durch diese Vorgehensweise konnten die Zusatzkosten, die alleine auf das Framework zurückzuführen sind, ermittelt werden. Diese Zusatzkosten können beim Fehlen registrierter Transformer-Komponenten auf annähernd Null gesenkt werden, wenn dieser Schritt ausbleibt. In der Implementation müssen dazu nur wenige Zeilen Code ergänzt werden.

Da der Einsatz des Frameworks nur dann von Vorteil ist, wenn Transformer-Komponenten registriert sind, ist diese Änderung nur dann sinnvoll, wenn auch die Transformer-Konfiguration zu Beginn dynamisch erzeugt wird, und daher das Vorhandensein bzw. Nichtvorhandensein von Transformer-Komponenten nicht vorhergesehen werden kann.

6.5.3 Event-getriebene Transformation

Während der Interface-Transformationsphase muß ein Interface-Transformer unter Umständen in jeder Iteration des Kompositionsalgorithmus TAU (Algorithmus 4.2) die gesamten im `ClassSet`-Objekt enthaltenen Klassen daraufhin untersuchen, ob sie von anderen aktiven Interface-Transformern so modifiziert wurden, daß weitere, eigene Transformationen nötig sind.

In diesem Zusammenhang gibt es die Option, eine Art Event-Modell in das Framework zu integrieren, bei dem sich ein Transformer beim Kompositionsalgorithmus als *Observer* für die Benachrichtigung über bestimmte Transformationen registrieren kann[3]. Sind für einen Interface-Transformer z.B. nur die

[3] Dies entspricht dem Observer-Pattern [GHJV95]

Oberklassen-Transformationen anderer Transformer relevant, so kann er sich für diese Klasse von Transformationen beim Kompositionsalgorithmus registrieren und wird dann entsprechend benachrichtigt. Dadurch könnten die Kosten für die Analyse der im `ClassSet`-Objekt befindlichen Klassenrepräsentationen für die Transformer-Komponenten gesenkt werden. Dabei ist jedoch zu beachten, daß durch die Verwaltung der Observer und deren Benachrichtigungen wieder neue Zusatzkosten entstehen, die insgesamt niedriger ausfallen müßten als ohne Event-Modell, damit man von einer Optimierung sprechen kann.

Da die Transformationen der Klassen ausschließlich vom Kompositionsalgorithmus TAU durchgeführt werden, ist die Korrektheit der Event-getriebenen Transformation nicht von den Transformer-Komponenten abhängig, da der Algorithmus TAU alleine für die Auslösung der richtigen Events verantwortlich wäre. Wären die Komponenten für die Auslösung der Events verantwortlich, könnte ein fehlerhafter Transformer eine korrekte Transformation verhindern, indem er falsche oder zu wenige Events auslöst.

Der Aufwand für eine Implementation dieser Option ist relativ gering, da nur wenige Klassen des Frameworks geändert werden müßten, und vorhandene Transformer-Komponenten auch ohne Anpassungen weiterhin funktionieren würden. Ohne Anpassung würden vorhandene Transformer die neuen Funktionen jedoch nicht nutzen, sodaß deren Vorteile nicht zum Tragen kämen.

Kapitel 7

Related Work

Im folgenden wird das in dieser Arbeit entwickelte Framework zur Ladezeittransformation von Java-Programmen mit anderen Arbeiten aus diesem Forschungsbereich verglichen.

7.1 Binary Component Adaptation

Die Binary Component Adaptation (BCA) [KH98] von Ralph Keller und Urs Hölzle erlaubt es, Komponenten in binärer Form während des Ladens anzupassen, und ermöglicht so Adaptionen ohne Zugriff auf die zugrundeliegenden Quelltexte. Es existiert eine Implementation für Java, die in die SUN JDK 1.1 Virtual Machine integriert wurde.

Anpassungen an Klassen werden in Form einer Java-ähnlichen Sprache definiert und von einem speziellen Kompiler in sogenannte *Delta-Files* übersetzt. Beim Aufruf der BCA-JVM werden diese Delta-Files als Parameter übergeben, und während des Ladens der Anwendung auf die entsprechenden Klassen angewandt. Abbildung 7.1 auf der nächsten Seite gibt einen Überblick über ein BCA-System. Mit BCA ist es z.B. möglich, neue Methoden und Felder zu einer Klasse hinzuzufügen, Interfaces um neue Operationen zu erweitern und für alle Klassen, die dieses Interface implementieren, gegebenenfalls eine Standardimplementation der neuen Operationen anzugeben.

Durch die vollständige Integration in die JVM erreicht BCA eine wesentlich höhere Geschwindigkeit als das vorgestellte Transformations-Framework, wenn ähnliche Transformationen vorgenommen werden. Desweiteren werden dadurch die Systemklassen nicht von der Transformation ausgeschlossen.

Die Deltas werden in einer leicht zu erlernenden, Java-ähnlichen Syntax beschrieben, sodaß keine exakten Kenntnisse des Classfile Formats bzw. einer API zur Manipulation von Classfiles erforderlich sind. Das hat für einen Benutzer zwar zum einen den Vorteil, daß der Einarbeitungsaufwand für die Verwendung von BCA verhältnismäßig gering ausfällt, zum anderen bleibt die Menge der möglichen Modifikationen auf die von der Sprache angebotenen beschränkt, die nur eine Teilmenge aller möglichen Classfile-Modifikationen abbildet. Zusätzlich gibt es, aufgrund der im Vorfeld statisch definierten Anpassungen, keine Möglichkeit, während des Transformierens der Klassen dynamisch über eventuelle Modifikationen zu entscheiden, um so z.B. die Abhängigkeiten von Klas-

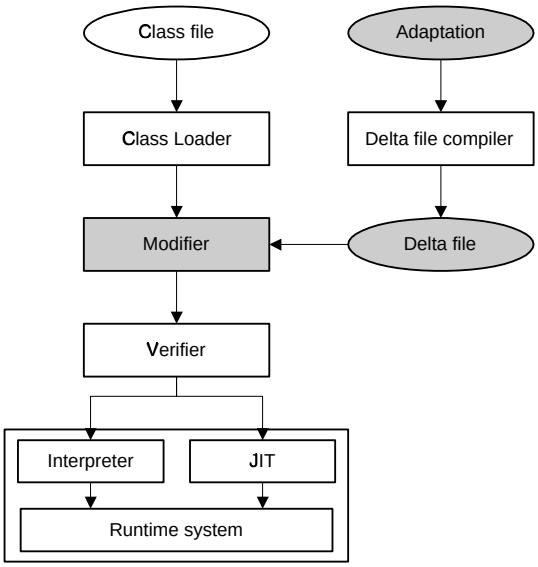

Abbildung 7.1: Überblick über ein Binary Component Adaptation System (aus [KH98])

sen untereinander berücksichtigen zu können. Damit erlaubt das vorgestellte Framework zur Ladezeittransformation eine wesentlich mächtigere Klasse von Modifikationen.

Durch die Integration von BCA in eine JVM-Implementation wird zwar eine wesentlich höhere Geschwindigkeit erreicht und die Transformation von Systemklassen ermöglicht, der große Vorteil der Plattformunabhängigkeit von Java geht dabei jedoch verloren, da Applikationen, die nur auf einem BCA-System ausführbar sind, auf die BCA-unterstützten Plattformen beschränkt bleiben. Konzeptionell spricht nichts dagegen, eine Variante des vorgestellten Transformations-Frameworks zu entwickeln, die ebenfalls direkt in eine JVM-Implementation integriert ist.

Fazit Das in dieser Arbeit vorgestellte Framework ist konzeptuell mächtiger, jedoch auch komplexer als BCA. So wäre es möglich, eine Transformer-Komponente zu entwickeln, die zu Beginn ein BCA-Delta-File zugewiesen bekommt, und die darin enthaltenen Anpassungen auf die Klassen des auszuführenden Programmes anwendet. Der wesentliche Vorteil von BCA ist darin zu sehen, daß der Einarbeitungsaufwand für einen Benutzer um ein Vielfaches geringer ist, und somit einfache Anpassungen von Klassen relativ schnell definiert werden können.

7.2 Java Object Instrumentation Environment

Das *Java Object Instrumentation Environment* (JOIE) [CCK98] von Geoff Cohen ist ein Toolkit zur Ladezeittransformation von Java-Klassen. Transformationen werden ebenfalls in Form von *Transformern* vorgegeben, die bei einem

speziellen Class Loader registriert werden können. Dieser Class Loader erzeugt nach dem Laden des Bytecodes ein sogenanntes `ClassInfo`-Objekt, das die geladene Klasse repräsentiert. Dieses Objekt wird dann allen registrierten Transformern der Reihe nach zur Manipulation übergeben. Da JOIE komplett in Java implementiert ist, kommt es ohne eine geänderte JVM-Implementation aus.

Die `ClassInfo`-Objekte beinhalten ähnlich der JavaClass API [Dah99b] ebenfalls eine Objektgraph-Repräsentation des zugrundeliegenden Classfiles. Über die von JOIE zur Verfügung gestellte API können diese Objekte beliebig modifiziert werden.

JOIE zielt in eine ähnliche Richtung wie diese Arbeit. Es existieren jedoch einige grundlegende Unterschiede. Wegen des Einsatzes eines eigenen Class Loaders können Applikationen mit JOIE nicht verwendet werden, wenn sie ebenfalls eigene Class Loaders einsetzen, da hierdurch die Transformation von Klassen umgangen würde.

Desweiteren können mit JOIE keine Klassen während der Transformation hinzugefügt werden und die Modifikationen der Transformer sind immer auf die aktuelle Klasse beschränkt, weshalb Abhängigkeiten zwischen Klassen bei der Transformation nicht berücksichtigt werden können.

Einen weiteren wesentlichen Unterschied stellt der Umstand dar, daß die Komposition von vielen unterschiedlichen Transformern nur rudimentär behandelt wird. Die aktuell zu transformierende Klasse wird den registrierten Transformern lediglich der Reihe nach, geordnet nach den beim Registrieren anzugebenden Prioritäten der Transformer, zur Transformation übergeben. Eine wie in dieser Arbeit vorgestellte automatische Komposition von Transformationen durch einen Kompositionsalgorithmus ist in JOIE nicht vorhanden.

Es fehlt ebenfalls eine Schnittstelle, die die komfortable Konfiguration des Toolkits ohne Code-Modifikationen und erneutes Kompilieren der Quelltexte zuläßt, wie dies z.B. mit der in 5.3.4 vorgestellten XML-Schnittstelle des Transformations-Frameworks oder den Delta-Files bei BCA möglich ist.

Fazit JOIE ist ein Vorläufer des in dieser Arbeit vorgestellten Frameworks und hat Einfluß auf dessen Entwicklung gehabt. Das in Kapitel 4 vorgestellte Transformationsmodell ist wesentlich mächtiger als JOIEs Modell. Der Schwerpunkt von JOIE ist in der Bereitstellung einer eigenen API zur Manipulation von Klassenrepräsentationen zu sehen. Es wäre möglich gewesen, diese API ohne den JOIE-eigenen Class Loader und dessen Transformer-Modell in der Implementation dieser Arbeit zu verwenden, und somit JOIE mit den Vorteilen des in Kapitel 4 vorgestellten Transformationsmodells zu verbinden.

7.3 Javassist

Javassist [Chi00] von Shigeru Chiba ist eine Klassenbibliothek, die *strukturelle Reflektion* (*structural reflection*) in Java ermöglicht. Strukturelle Reflektion erlaubt es einem Programm, die Definition von Datenstrukturen wie z.B. Klassen und Methoden zu verändern. Sie wird von einigen Sprachen wie z.B. *Smalltalk* [GR83] und *CLOS* [KdB91] angeboten. Javassist kommt aus dem Bereich der *Metaobject Protocols* [KdB91]. Es gibt dort verschiedene Ansätze, zu welchen Zeitpunkten Modifikationen an den Datenstrukturen vorgenommen werden

können. Javassist verfolgt dabei den Ansatz, die Modifikationen zur Ladezeit vorzunehmen, und gleicht in dieser Hinsicht dem vorgestellten Framework.

Javassist ist vollständig in Java implementiert und kommt ohne ein eigenes Laufzeitsystem oder einen Kompiler aus. Es erlaubt strukturelle Reflektion von Klassen vor der Übergabe an die JVM. Dazu werden Änderungen durch strukturelle Reflektionen in äquivalente Bytecode-Transformationen der zugrundeliegenden Classfiles übersetzt. Nach der Transformation werden die Classfiles in die JVM geladen, sodaß weitere Modifikationen nicht mehr möglich sind.

Die Integration der strukturellen Reflektion in den Ladevorgang der Anwendungsklassen erfolgt ähnlich wie in JOIE in einem eigenen Class Loader. Vor der Übergabe eines Classfiles an die JVM kann daraus ein sogenanntes *Compile time class*-Objekt der entsprechenden Klassen erzeugt werden. Dieses bietet für Modifikationen eine Quelltextabstraktion der zugrundeliegenden Klasse an, sodaß Änderungen ohne Kenntnisse des Classfile Formats durchgeführt werden können. Dadurch wird jedoch auch die Menge der möglichen Transformationen stark beschränkt. So können z.B. keine völlig neuen Methoden zu einer Klassen hinzugefügt, sondern nur Methoden von einer Klasse zu einer anderen kopiert werden.

Javassist wird ebenfalls durch einen speziellen Class Loader in den Ladevorgang der Anwendungsklassen integriert. Dadurch entsteht das von JOIE bekannte Problem, daß nicht alle Klassen transformiert werden können, wenn Applikationen ebenfalls eigene Class Loaders einsetzen.

Javassist besitzt kein Transformer-Modell, sondern geht davon aus, daß der Class Loader, der eine Klasse lädt, alleine über die Modifikationen entscheidet bzw. diese Aufgabe an ein anderes Objekt delegiert. Es besitzt keine Unterstützung für multiple Transformer oder eine Schnittstelle für die Konfiguration des Class Loaders ohne erneutes Übersetzen des zugrundeliegenden Quelltextes.

Fazit Das in dieser Arbeit vorgestellte Transformations-Framework ist konzeptuell mächtiger, jedoch auch komplexer als Javassist. Strukturelle Reflektion bzw. Transformation kann zwar ohne größeren Einarbeitungsaufwand mit Javassist realisiert werden, die Menge der möglichen Modifikationen ist jedoch, wie oben erwähnt, sehr stark eingeschränkt. Es wäre möglich, das Transformations-Framework mit einer zu entwickelnden Transformer-Komponente auszustatten, die eine ähnliche API wie Javassist zur Verfügung stellt, um somit die strukturelle Reflektion mit den Vorteilen des Frameworks zu paaren.

7.4 Bytecode Engineering Tools

In Abschnitt 5.2.1 wurde bereits erwähnt, daß eine Reihe von Bibliotheken zur Repräsentation und Manipulation von Classfiles existieren, deren Einsatz im Rahmen dieser Arbeit als Ersatz für die JavaClass API [Dah99b] möglich gewesen wäre. Ein Vergleich mit dem vorgestellten Transformations-Framework ist daher nicht sinnvoll, sie werden jedoch der Vollständigkeit halber im folgenden kurz beschrieben.

7.4.1 Jikes Bytecode Toolkit

Das *Jikes Bytecode Toolkit* (JikesBT) [Laf00] ist eine Java-Klassenbibliothek, die eine *High-Level* API zum Lesen, Erzeugen, Manipulieren und Schreiben von Java Classfiles zur Verfügung stellt. JikesBT bietet eine *logische Repräsentation* von Classfiles an, wohingegen die meisten anderen Bibliotheken zur Classfile-Repräsentation eine direktere und detailliertere *Low-Level-Repräsentation* bieten. So verbirgt JikesBT zum Beispiel den in Classfiles enthaltenen *Constant-Pool* auf Wunsch vollkommen vor dem Benutzer, wodurch Transformationen von Classfiles wesentlich vereinfacht werden. JikesBT steht zur Zeit nur unter einer 90-tägigen Evaluationslizenz zur Verfügung.

JikesBT soll als Grundlage für solche Applikationen dienen, die Classfiles analysieren, erzeugen oder manipulieren müssen, wie dies z.B. vom vorgestellten Framework zur Ladezeittransformation oder einem Kompiler durchgeführt wird.

7.4.2 Bytecode Instrumenting Tool

Das *Bytecode Instrumenting Tool* (BIT) [LZ97] von Han Bok Lee war eine der ersten Java-Klassenbibliotheken zum Bearbeiten von Classfiles. Der Hauptfokus von BIT liegt in der Modifikation des in den Classfiles enthaltenen Bytecodes, also den Implementationen der einzelnen Methoden.

Zu dem Zeitpunkt als BIT entwickelt wurde, existierten bereits auf vielen verschiedenen Plattformen Tools zur Analyse des *dynamischen Verhaltens* von Programmen, einzig für die Java-Plattform waren solche Tools noch nicht verfügbar. BIT versteht sich als eine Art Backend-Tool, das für die Entwicklung von solchen Analyseprogrammen verwendet werden kann. Mit BIT können Analyseprogramme vereinfacht Methodenaufrufe in vorhandene Classfiles einfügen, um so bei deren Ausführung an *Profiling-Informationen* zu gelangen, oder das *Debugging* von Programmen zu vereinfachen. BIT bietet daher auch keine Klassen an, die die Einbindung zur Ladezeittransformation vereinfachen oder ein Transformer-Modell implementieren. Die von BIT angebotene API ist bei weitem nicht so mächtig wie die JavaClass API oder die JikesBT API. Die von ihr angebotenen Funktionen sind im wesentlichen auf die angesprochenen Modifikationen von Methodenimplementationen beschränkt.

Kapitel 8

Fazit

Transformationsmodell Das in dieser Arbeit vorgestellte Transformationsmodell ermöglicht die *dynamische Transformation* von Java-Programmen durch *multiple Transformer-Komponenten*.

Transformer-Komponenten analysieren die zu transformierenden Klassen und entscheiden über die anzuwendenden Transformationen. Es wurde ein spezieller Algorithmus entwickelt, der in der Lage ist, eine beliebige Menge von Transformer-Komponenten *automatisch* zu komponieren.

Ein wesentlicher Fortschritt dieser Arbeit gegenüber existierenden Ansätzen ist die Möglichkeit, beliebig viele Programmklassen gleichzeitig transformieren zu können, sodaß sogar Abhängigkeiten von Klassen untereinander während der Transformation berücksichtigt werden können. Dadurch ist es z.B. möglich, die Schnittstelle einer Klasse A dynamisch durch die Schnittstelle einer Klasse B zu erweitern.

Um eine gewisse Güte des transformierten Java-Programmes zu gewährleisten, wurden solche Transformationen ausgeschlossen, deren Anwendung ein Programm im worst case in einen inkonsistenten Zustand bringt, der auch durch weitere Transformationen nicht wieder in einen konsistenten Zustand überführt werden kann.

Die Transformer-Komponenten wurden in die sogenannten *Interface-* und die *Code-Transformer-Komponenten* unterteilt, die unterschiedliche Kategorien von Transformationen durchführen können. Code-Transformer-Komponenten transformieren ausschließlich Methodenimplementationen, Interface-Transformer-Komponenten führen alle anderen erlaubten Transformationen durch, also z.B. das Hinzufügen von Methoden oder Feldern zu Klassen.

Es wurde bewiesen, daß durch diese Klassifizierung der Transformer-Komponenten und eine entsprechende Aufteilung der Transformation des Programmes, bei der Komposition der Interface-Transformer-Komponenten keine Abhängigkeiten der Komponenten untereinander berücksichtigt werden müssen. Daher können Interface-Transformer völlig unabhängig voneinander entwickelt werden, und deren automatische Komposition wird ohne Informationen über ihre Funktionsweise bzw. ihre offensichtlichen oder verborgenen Abhängigkeiten möglich. Lediglich die Komposition der Code-Transformer-Komponenten benötigt die Festlegung einer Kompositionsreihenfolge durch den Anwender, da gezeigt wurde, daß diese einen wesentlichen Einfluß auf das Ergebnis der Transformation besitzt.

Während der Transformation werden die Aktionen der einzelnen Transformer-Komponenten überwacht, sodaß eine bestimmte Klasse von Konflikten zwischen den Transformationen einzelner Transformer erkannt und in bestimmten Fällen aufgelöst werden kann.

Framework Das Transformationsmodell wurde in der Programmiersprache Java implementiert, und das so entstandene Framework zur Ladezeittransformation von Java-Programmen in das JDK1.3 integriert. Dabei konnte auf die Modifikation einer Java Virtual Machine Implementation verzichtet werden, sodaß die Plattformunabhängigkeit der Java-Technologie auch beim Einsatz des Frameworks zum Tragen kommt.

Der Vorgang der Transformation ist für die transformierte Applikation völlig transparent. Es existieren keine Einschränkungen für die Applikation, wie z.B. die Verwendung spezieller Class Loaders, wie dies bei vielen anderen Ansätzen der Fall ist. Die Entwicklung neuer Transformer-Komponenten gestaltet sich sehr einfach, da lediglich eine definierte Schnittstelle implementiert werden muß. Die Transformation der Klassen erfolgt über eine komfortable API und die Komplexität des Class Loader Systems wird durch das Framework vor den Transformer-Komponenten verborgen. Für die Ladezeittransformation einer Applikation wird kein Zugriff auf die zugrundeliegenden Quelltexte benötigt, da das Classfile Format die Struktur und die symbolischen Informationen der ursprünglichen Java-Quelltexte nahezu vollkommen widerspiegelt.

Die Konfiguration des Frameworks erfolgt über eine einfach zu bedienende XML-Schnittstelle, über die die Registrierung und Parametrisierung der Transformer-Komponenten festgelegt werden kann.

Evaluation Das Ziel der Evaluation war, die *Zusatzkosten* zu ermitteln, die durch die Ladezeittransformation der Programmklassen entstehen. Dazu wurden Transformer-Komponenten mit unterschiedlich komplexen Aufgabenstellungen entwickelt, die von dem einfachen Ersetzen weniger Code-Sequenzen bis zum Hinzufügen einiger 100 Methoden und dem Transformieren mehrerer 1000 Code-Sequenzen reichen. Diese Transformer wurden in verschiedenen Konfigurationen auf fünf repräsentative Applikationen, wie z.B. einem Java-Kompiler, angewendet.

Aus den Messwerten konnten die Faktoren ermittelt werden, die einen wesentlichen Einfluß auf die Zusatzkosten besitzen. So variieren die relativen Laufzeitzusatzkosten sehr stark mit der Anzahl der zu transformierenden Anwendungsklassen, der Gesamtlaufzeit der Anwendung und der Anzahl und Komplexität der aktiven Transformer-Komponenten. Da die Anzahl und Größe der zu transformierenden Anwendungsklassen ein Maß für die Eingabegröße der Transformation durch die Transformer darstellt, war es nicht überraschend, daß die Zusatzkosten mit zunehmender Anzahl und Größe der Anwendungsklassen steigen. Demgegenüber steht, daß die Laufzeitzusatzkosten mit zunehmender Gesamtlaufzeit einer Anwendung immer weniger ins Gewicht fallen, und somit die relativen Laufzeitzusatzkosten in den Messungen mit denselben aktiven Transformer-Komponenten, jedoch Anwendungen unterschiedlicher Gesamtlaufzeit, z.B. zwischen 23% und 585% variieren.

Die Ergebnisse der Evaluation haben desweiteren gezeigt, daß eine Klasse von Transformationen, nämlich die Änderung der direkten Oberklasse, beson-

ders teuer ist, da sie die Analyse und Modifikation vieler Methodenimplementationen nach sich zieht.

Anwendung Das vorgestellte Framework wird im Rahmen des Tailor-Projekts [Tai00] am Institut für Informatik III der Universität Bonn eingesetzt. Das Tailor-Projekt untersucht und entwickelt Sprachkonstrukte zur Unterstützung von dynamischer Komponentenanpassung. Die zwei wesentlichen Voraussetzungen hierfür sind

- die Fähigkeit dynamisch neue Komponenten in ein laufendes System zu laden, sodaß sie die Funktion vorhandener Komponenten beeinflussen [Kni99].
- die Möglichkeit eine neue Komponente für Klienten unter derselben Objektidentität erscheinen zu lassen wie die durch sie angepasste Komponente [CS01].

Das Framework wird zur Zeit eingesetzt, um eine Spracherweiterung von Java um objektbasierte Vererbung [Kni00, CKC99] zu implementieren, die die Voraussetzungen von Punkt 1 erfüllt. Durch den Einsatz des Frameworks können die Transformationen, die die Spracherweiterungen implementieren, auch auf Klassen von Drittanbietern angewendet werden, die nicht als Quelltext vorliegen.

Anhang A

Transformer Configuration Language

Für das Einlesen der Konfiguration des Frameworks wird die *Extensible Markup Language* eingesetzt. Informationen zu diesem Thema sind in [BM98] enthalten. Die folgende *Document Type Definition* definiert die im Rahmen dieser Arbeit entwickelte *Transformer Configuration Language* (TCL). Ein XML-Dokument muß der TCL genügen, um als Konfiguration des Frameworks akzeptiert zu werden (siehe Abschnitt 5.3.4 auf Seite 85).

```
<?xml version='1.0' encoding='us-ascii'?>

<!--
    DTD for the (T)ransformer (C)onfiguration (L)anguage
-->

<!ELEMENT TrafoConfig (Transformers, InterfaceTransformers,
                       CodeTransformers, PartialOrder, Settings?)>
<!ELEMENT Transformers (Transformer*)>
<!ELEMENT Transformer (ClassName, Id, Parameter)>
<!ELEMENT ClassName (#PCDATA)>
<!ELEMENT Id (#PCDATA)>
<!ELEMENT Parameter (#PCDATA)>
<!ELEMENT InterfaceTransformers (Id*)>
<!ELEMENT CodeTransformers (Sequence | Graph)>
<!ELEMENT Sequence (Id*)>
<!ELEMENT Graph (Vertice*,Edge*)>
<!ELEMENT Vertice (Id)>
<!ELEMENT Edge (Id,Id)>
<!ELEMENT PartialOrder (ClassName)>
<!ELEMENT Settings (verboseON?, startUpMessage?, statisticInfoMessage?,
                    checkPartialOrder?, dumpClassSet?, classSetSecurityLevel?,
                    maxIterations?)>
<!ELEMENT verboseON (#PCDATA)>
<!ELEMENT startUpMessage (#PCDATA)>
<!ELEMENT statisticInfoMessage (#PCDATA)>
```

```
<!ELEMENT checkPartialOrder (#PCDATA)>
<!ELEMENT dumpClassSet (#PCDATA)>
<!ELEMENT maxIterations (#PCDATA)>
<!ELEMENT classSetSecurityLevel (#PCDATA)>
```

Anhang B

Meßwerte der Evaluation

Die folgenden Tabellen enthalten alle Meßwerte die während der Evaluation gesammelt wurden, und bildeten die Grundlage für Kapitel 6.

Applikation	javac	JLex	CUP	javadoc	JGL Benchmarks
Superklassen geändert	0	0	0	0	0
Interfaces hinzugefügt	0	0	0	0	0
Methoden hinzugefügt	0	0	0	0	0
Codesequenzen geändert	0	0	0	0	0
Sessions gesamt	140	24	30	255	30
Iterationen gesamt	140	24	30	255	30
Min Iterationen pro Session	1	1	1	1	1
Max Iterationen pro Session	1	1	1	1	1
Durchschnitt Iterationen pro Session	1.00	1.00	1.00	1.00	1.00
Min Klassen im ClassSet	1	1	1	1	1
Max Klassen im ClassSet	1	1	1	1	1
Durchschnitt Klassen im ClassSet	1.00	1.00	1.00	1.00	1.00

Tabelle B.1: Transformations-Statistik bzgl. der Konfiguration *Empty*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	0	863	2455	0	0	1191	31467	26583	4884	18%	30276	26583	3693	13%
JLex	0	213	951	0	0	1177	4479	2121	2358	111%	3302	2121	1181	55%
CUP	0	286	1120	0	0	1182	6182	3594	2588	72%	5000	3594	1406	39%
javadoc	0	1231	3334	0	0	1180	103718	90265	13453	14%	102538	90265	12273	13%
JGL Benchmarks	0	184	743	0	0	1188	35390	33231	2159	6%	34202	33231	971	2%

Tabelle B.2: Zeitmessungen bzgl. der Konfiguration *Empty*.

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	14.1	8.1	5.9	73%	9.7	4.6	5.1	110%
JLex	2.5	1.8	0.7	42%	1.0	0.3	0.7	231%
CUP	3.4	1.1	2.3	198%	1.9	0.7	1.1	147%
javadoc	16.8	7.4	9.4	127%	15.0	6.3	8.6	135%
JGL Benchmarks	1.7	2.3	-0.5	-22%	0.9	0.3	0.6	207%

Tabelle B.3: Speicherverbrauch bzgl. der Konfiguration *Empty*.

Applikation	Superklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Sessions gesamt	Iterationen gesamt	Min Iterationen pro Session	Max Iterationen pro Session	Durchschnitt Iterationen pro Session	Min Klassen im ClassSet	Max Klassen im ClassSet	Durchschnitt Klassen im ClassSet
javac	48	0	0	0	141	189	1	2	1.34	1	1	1.00
JLex	23	0	0	0	25	48	1	2	1.92	1	1	1.00
CUP	20	0	0	0	31	51	1	2	1.64	1	1	1.00
javadoc	52	0	0	0	256	308	1	2	1.20	1	1	1.00
JGL Benchmarks	16	0	0	0	31	47	1	2	1.51	1	1	1.00

Tabelle B.4: Transformations-Statistik bzgl. der Konfiguration *GenericObject*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	0	856	2195	7953	0	1218	39418	26583	12835	48%	38200	26583	11617	43%
JLex	0	193	823	4778	0	1218	9104	2121	6983	329%	7886	2121	5765	271%
CUP	0	380	997	4738	0	1211	10901	3594	7307	203%	9690	3594	6096	169%
javadoc	0	1328	3806	7882	0	1210	110887	90265	20622	22%	109677	90265	19412	21%
JGL Benchmarks	0	185	762	2181	0	1219	37545	33231	4314	12%	36326	33231	3095	9%

Tabelle B.5: Zeitmessungen bzgl. der Konfiguration *GenericObject*.

Applikation	Interface-Transformation
	GenericObject
javac	61
JLex	26
CUP	20
javadoc	90
JGL Benchmarks	18

Tabelle B.6: Laufzeiten der Transformer bzgl. der Konfiguration *GenericObject*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	14.8	8.1	6.6	81%	9.8	4.6	5.1	111%
JLex	2.5	1.8	0.7	39%	1.1	0.3	0.8	252%
CUP	3.4	1.1	2.3	200%	2.2	0.7	1.4	190%
javadoc	16.1	7.4	8.7	118%	15.0	6.3	8.6	136%
JGL Benchmarks	2.4	2.3	0.1	5%	0.9	0.3	0.6	203%

Tabelle B.7: Speicherverbrauch bzgl. der Konfiguration *GenericObject*.

Applikation	javac	JLex	CUP	javadoc	JGL Benchmarks
Superklassen geändert	0	0	0	0	0
Interfaces hinzugefügt	51	23	21	61	19
Methoden hinzugefügt	51	23	21	58	19
Codesequenzen geändert	0	0	0	0	0
Sessions gesamt	140	24	30	255	30
Iterationen gesamt	191	47	51	316	49
Min Iterationen pro Session	1	1	1	1	1
Max Iterationen pro Session	2	2	2	2	2
Durchschnitt Iterationen pro Session	1.36	1.95	1.70	1.23	1.63
Min Klassen im ClassSet	1	1	1	1	1
Max Klassen im ClassSet	1	1	1	1	1
Durchschnitt Klassen im ClassSet	1.00	1.00	1.00	1.00	1.00

Tabelle B.8: Transformations-Statistik bzgl. der Konfiguration *Printable*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	2045	925	2251	2673	0	1197	33685	26583	7102	26%	30443	26583	3860	14%
JLex	2045	247	730	2352	0	1205	6782	2121	4661	219%	3532	2121	1411	66%
CUP	2001	436	930	2317	0	1215	8434	3594	4840	134%	5218	3594	1624	45%
javadoc	2049	1122	3456	2772	0	1231	104731	90265	14466	16%	101451	90265	11186	12%
JGL Benchmarks	2030	185	649	2397	0	1206	37789	33231	4558	13%	34553	33231	1322	3%

Tabelle B.9: Zeitmessungen bzgl. der Konfiguration *Printable*.

Applikation	Interface-Transformation
	Printable
javac	2420
JLex	2272
CUP	2244
javadoc	2478
JGL Benchmarks	2240

Tabelle B.10: Laufzeiten der Transformer bzgl. der Konfiguration *Printable*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	15.2	8.1	7.0	86%	9.9	4.6	5.2	113%
JLex	1.8	1.8	-0.0	-0%	1.1	0.3	0.8	271%
CUP	2.3	1.1	1.2	105%	2.0	0.7	1.2	158%
javadoc	15.4	7.4	8.0	108%	15.1	6.3	8.7	137%
JGL Benchmarks	1.2	2.3	-1.0	-44%	1.0	0.3	0.7	248%

Tabelle B.11: Speicherverbrauch bzgl. der Konfiguration *Printable*.

Applikation	Superklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Sessions gesamt	Iterationen gesamt	Min Iterationen pro Session	Max Iterationen pro Session	Durchschnitt Iterationen pro Session	Min Klassen im ClassSet	Max Klassen im ClassSet	Durchschnitt Klassen im ClassSet
javac	0	0	1000	8602	140	252	1	2	1.80	1	1	1.00
JLex	0	0	326	2810	24	41	1	2	1.70	1	1	1.00
CUP	0	0	158	870	30	55	1	2	1.83	1	1	1.00
javadoc	0	0	1070	7852	255	398	1	2	1.56	1	1	1.00
JGL Benchmarks	0	0	62	695	30	40	1	2	1.33	1	1	1.00

Tabelle B.12: Transformations-Statistik bzgl. der Konfiguration *Accessor*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	1991	856	1919	3367	20462	1244	70722	42814	27908	65%	67487	42814	24673	57%
JLex	1808	285	767	1266	7641	1251	13839	2636	11203	425%	10780	2636	8144	308%
CUP	1970	311	911	693	7320	1240	14133	3836	10297	268%	10923	3836	7087	184%
javadoc	2617	1290	2934	3611	28676	1293	141100	98334	42766	43%	137190	98334	38856	39%
JGL Benchmarks	1960	201	674	368	4836	1255	46636	39624	7012	17%	43421	39624	3797	9%

Tabelle B.13: Zeitmessungen bzgl. der Konfiguration *Accessor*.

Applikation	Interface-Transformation	Code-Transformation
	Accessor	Accessor
javac	2624	20455
JLex	977	7641
CUP	518	7317
javadoc	2815	28654
JGL Benchmarks	241	4832

Tabelle B.14: Laufzeiten der Transformer bzgl. der Konfiguration *Accessor*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	16.8	9.6	7.1	74%	11.5	4.5	7.0	155%
JLex	2.2	1.4	0.7	50%	1.6	0.1	1.4	1258%
CUP	4.1	2.3	1.8	76%	2.4	0.5	1.8	326%
javadoc	18.0	6.8	11.2	166%	16.3	6.1	10.1	164%
JGL Benchmarks	2.2	1.8	0.4	24%	1.0	0.1	0.9	787%

Tabelle B.15: Speicherverbrauch bzgl. der Konfiguration *Accessor*.

Applikation	javac	JLex	CUP	javadoc	JGL Benchmarks
Superklassen geändert	0	0	0	0	0
Interfaces hinzugefügt	0	0	0	0	0
Methoden hinzugefügt	0	1	0	102	0
Codesequenzen geändert	0	0	0	0	0
Sessions gesamt	134	24	30	224	28
Iterationen gesamt	213	25	39	413	30
Min Iterationen pro Session	1	1	1	1	1
Max Iterationen pro Session	3	2	2	6	2
Durchschnitt Iterationen pro Session	1.58	1.04	1.30	1.84	1.07
Min Klassen im ClassSet	1	1	1	1	1
Max Klassen im ClassSet	2	1	1	3	2
Durchschnitt Klassen im ClassSet	1.04	1.00	1.00	1.13	1.07

Tabelle B.16: Transformations-Statistik bzgl. der Konfiguration *Cloneable*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	2027	957	2036	2436	0	1204	33483	26583	6900	25%	30252	26583	3669	13%
JLex	2059	249	730	2240	0	1210	6652	2121	4531	213%	3383	2121	1262	59%
CUP	2006	292	970	2098	0	1200	8145	3594	4551	126%	4939	3594	1345	37%
javadoc	2027	1103	3354	3618	0	1204	104842	90265	14577	16%	101611	90265	11346	12%
JGL Benchmarks	2107	184	690	2258	0	1220	37838	33231	4607	13%	34511	33231	1280	3%

Tabelle B.17: Zeitmessungen bzgl. der Konfiguration *Cloneable*.

Applikation	Interface-Transformation
	Cloneable
javac	2119
JLex	2192
CUP	2039
javadoc	2555
JGL Benchmarks	2135

Tabelle B.18: Laufzeiten der Transformer bzgl. der Konfiguration *Cloneable*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	14.7	8.1	6.6	81%	9.8	4.6	5.1	110%
JLex	1.5	1.8	-0.2	-13%	1.1	0.3	0.8	260%
CUP	2.1	1.1	0.9	82%	1.9	0.7	1.1	149%
javadoc	15.4	7.4	7.9	107%	15.1	6.3	8.7	137%
JGL Benchmarks	2.3	2.3	0.0	2%	1.0	0.3	0.7	235%

Tabelle B.19: Speicherverbrauch bzgl. der Konfiguration *Cloneable*.

Applikation	Superklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Sessions gesamt	Iterationen gesamt	Min Iterationen pro Session	Max Iterationen pro Session	Durchschnitt Iterationen pro Session	Min Klassen im ClassSet	Max Klassen im ClassSet	Durchschnitt Klassen im ClassSet
javac	0	0	0	2	140	140	1	1	1.00	1	1	1.00
JLex	0	0	0	0	24	24	1	1	1.00	1	1	1.00
CUP	0	0	0	7	30	30	1	1	1.00	1	1	1.00
javadoc	0	0	0	3	255	255	1	1	1.00	1	1	1.00
JGL Benchmarks	0	0	0	0	30	30	1	1	1.00	1	1	1.00

Tabelle B.20: Transformations-Statistik bzgl. der Konfiguration *SystemExit*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	0	1043	2206	0	667	1256	32248	26583	5665	21%	30992	26583	4409	16%
JLex	0	218	932	0	16	1256	4538	2121	2417	113%	3282	2121	1161	54%
CUP	0	296	1021	0	1218	1251	7366	3594	3772	104%	6115	3594	2521	70%
javadoc	0	928	3423	0	943	1260	103794	90265	13529	14%	102534	90265	12269	13%
JGL Benchmarks	0	183	690	0	15	1259	35788	33231	2557	7%	34529	33231	1298	3%

Tabelle B.21: Zeitmessungen bzgl. der Konfiguration *SystemExit*.

Applikation	Code-Transformation
	SystemExit
javac	662
JLex	12
CUP	1216
javadoc	918
JGL Benchmarks	11

Tabelle B.22: Laufzeiten der Transformer bzgl. der Konfiguration *SystemExit*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	14.7	8.1	6.6	80%	9.7	4.6	5.1	110%
JLex	2.5	1.8	0.7	42%	1.0	0.3	0.7	232%
CUP	3.7	1.1	2.5	219%	1.9	0.7	1.1	146%
javadoc	16.9	7.4	9.5	128%	14.9	6.3	8.5	134%
JGL Benchmarks	2.6	2.3	0.3	13%	0.9	0.3	0.6	222%

Tabelle B.23: Speicherverbrauch bzgl. der Konfiguration *SystemExit*.

Applikation	Superklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Sessions gesamt	Iterationen gesamt	Min Iterationen pro Session	Max Iterationen pro Session	Durchschnitt Iterationen pro Session	Min Klassen im ClassSet	Max Klassen im ClassSet	Durchschnitt Klassen im ClassSet
javac	0	0	1000	8602	134	253	1	3	1.88	1	2	1.04
JLex	0	0	327	2810	24	41	1	2	1.70	1	1	1.00
CUP	0	0	158	870	30	55	1	2	1.83	1	1	1.00
javadoc	0	0	1172	7852	224	461	1	6	2.05	1	3	1.13
JGL Benchmarks	0	0	62	695	28	38	1	2	1.35	1	2	1.07

Tabelle B.24: Transformations-Statistik bzgl. der Konfiguration *AccessorCloneable*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	2027	825	1814	5561	18768	1246	71352	42814	28538	66%	68079	42814	25265	59%
JLex	2050	286	711	3301	5671	1245	13811	2636	11175	423%	10516	2636	7880	298%
CUP	2014	304	1020	2699	5241	1249	14112	3836	10276	267%	10849	3836	7013	182%
javadoc	2003	1040	2996	6816	26062	1293	141288	98334	42954	43%	137992	98334	39658	40%
JGL Benchmarks	2017	196	652	2458	2755	1253	47135	39624	7511	18%	43865	39624	4241	10%

Tabelle B.25: Zeitmessungen bzgl. der Konfiguration *AccessorCloneable*.

	Interface-Transformation		Code-Transformation
Applikation	Cloneable	Accessor	Accessor
javac	2136	2777	18754
JLex	2082	941	5670
CUP	2045	428	5241
javadoc	2411	2725	26047
JGL Benchmarks	2041	225	2754

Tabelle B.26: Laufzeiten der Transformer bzgl. der Konfiguration *AccessorCloneable*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	16.6	9.6	7.0	73%	11.5	4.5	7.0	155%
JLex	2.1	1.4	0.6	44%	1.5	0.1	1.3	1180%
CUP	4.2	2.3	1.8	77%	2.4	0.5	1.8	328%
javadoc	18.4	6.8	11.6	171%	16.5	6.1	10.3	168%
JGL Benchmarks	2.7	1.8	0.9	49%	1.0	0.1	0.9	801%

Tabelle B.27: Speicherverbrauch bzgl. der Konfiguration *AccessorCloneable*.

Applikation	Superklassen geändert	Interfaces hinzugefügt	Methoden hinzugefügt	Codesequenzen geändert	Sessions gesamt	Iterationen gesamt	Min Iterationen pro Session	Max Iterationen pro Session	Durchschnitt Iterationen pro Session	Min Klassen im ClassSet	Max Klassen im ClassSet	Durchschnitt Klassen im ClassSet
javac	48	52	1052	8604	134	313	1	4	2.33	1	2	1.05
JLex	23	24	351	2810	24	71	1	4	2.95	1	2	1.04
CUP	20	22	180	877	30	81	2	4	2.70	1	2	1.03
javadoc	52	62	1231	7855	224	525	1	6	2.34	1	3	1.14
JGL Benchmarks	16	20	82	695	28	66	1	4	2.35	1	2	1.10

Tabelle B.28: Transformations-Statistik bzgl. der Konfiguration *Altogether*.

Applikation	BCP auswerten	ClassGen nach Bytecode	Bytecode nach ClassGen	Interfacetransformation	Codetransformation	TC einlesen	Laufzeit	Referenzwert	Differenz	Overhead	Laufzeit ohne konst. Overhead	Referenzwert	Differenz	Overhead
javac	2083	792	2106	14074	18258	1295	79160	42814	36346	84%	75782	42814	32968	77%
JLex	2029	235	763	8229	4969	1297	18078	2636	15442	585%	14752	2636	12116	459%
CUP	2081	309	953	7479	4975	1298	18796	3836	14960	389%	15417	3836	11581	301%
javadoc	2074	1334	2921	14927	26489	1297	151306	98334	52972	53%	147935	98334	49601	50%
JGL Benchmarks	2037	219	617	4471	2342	1297	48862	39624	9238	23%	45528	39624	5904	14%

Tabelle B.29: Zeitmessungen bzgl. der Konfiguration *Altogether*.

Applikation	Interface-Transformation				Code-Transformation	
	Accessor	GenericObject	Cloneable	Printable	SystemExit	Accessor
javac	2894	72	122	2436	242	18006
JLex	883	23	29	2236	5	4961
CUP	360	30	30	2337	608	4361
javadoc	2542	124	414	2543	478	25986
JGL Benchmarks	177	19	31	2229	1	2338

Tabelle B.30: Laufzeiten der Transformer bzgl. der Konfiguration *Altogether*

Applikation	Speicher vor GC	Referenzwert	Differenz	Overhead	Speicher nach GC	Referenzwert	Differenz	Overhead
javac	17.2	9.6	7.5	78%	11.6	4.5	7.1	158%
JLex	2.4	1.4	1.0	68%	1.6	0.1	1.4	1262%
CUP	2.9	2.3	0.5	22%	2.3	0.5	1.8	320%
javadoc	18.5	6.8	11.7	172%	16.7	6.1	10.5	170%
JGL Benchmarks	2.2	1.8	0.3	22%	1.1	0.1	1.0	888%

Tabelle B.31: Speicherverbrauch bzgl. der Konfiguration *Altogether*.

Literaturverzeichnis

[AG97] Ken Arnold and James Gosling. *The Java Programming Language*. Java Series. Addison Wesley, 1997.

[Alp00] XML Parser for Java. http://www.alphaworks.ibm.com/tech/xml4j, 2000.

[Apa00] The Xerces Java Parser. http://xml.apache.org/xerces-j/index.html, 2000.

[BA00] Elliot Berk and C. Scott Ananian. JLex: A Lexical Analyzer Generator for Java(tm). http://www.cs.princeton.edu/∼appel/modern/java/JLex/, 2000.

[BM98] Henning Behme and Stefan Mintert. *XML in der Praxis*. Addison-Wesley, 1998.

[CCK98] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, 1998. USENIX Association.

[Chi00] Shigeru Chiba. Load-Time Structural Reflection in Java. In Elisa Bertino, editor, *Proceedings of ECOOP2000*, LNCS 1850. Springer, 2000.

[CKC99] Pascal Costanza, Günter Kniesel, and Armin B. Cremers. Lava: Sprachерweiterung für Delegation in Java. In *Java-Informations-Tage 1999*, pages 233–242. Springer, 1999.

[CS01] Pascal Costanza and Oliver Stiemerling. Dynamic Recomposition of Components and Object Identity (Zur Veröffentlichung angenommen). In *TOOLS Europe 2001*, Freiburg, Schweiz, 2001. IEEE Computer Press.

[Dah99a] Markus Dahm. Byte Code Engineering. In *Java-Informations-Tage 1999*, pages 267–277. Springer, 1999.

[Dah99b] Markus Dahm. Byte Code Engineering with the JavaClass API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, 1999.

[Dav00] James Duncan Davidson. Java API for XML Parsing. http://java.sun.com/aboutJava/communityprocess/final/jsr005/index.html, 2000.

[Gam] Erich Gamma. JUnit, Testing Resources for Extreme Programming. http://www.junit.org.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1995.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GJSB00] Jar Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, 2000.

[Gon99a] Li Gong. *Inside Java 2 Platform Security*. The Java Series. Addison-Wesley, 1999.

[Gon99b] Li Gong. Securely loading classes. In *Inside Java 2 Platform Security*, The Java Series, pages 71–83. Addison-Wesley, 1999.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Ham97] Graham Hamilton. JavaBeans. http://java.sun.com/beans/docs/spec.html, 1997.

[HFA+99] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew W. Appel. CUP Parser Generator for Java. http://www.cs.princeton.edu/∼appel/modern/java/CUP/, 1999.

[Jav] Java 2 Platform, Standard Edition, V1.2.2 API Specification.

[JDK00] Java 2 Platform, Standard Edition, v 1.3 API Specification. http://java.sun.com/j2se/1.3/docs/api/index.html, 2000.

[KdB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[KH98] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In Eric Jul, editor, *Proceedings ECOOP '98*, LNCS 1445, 1998.

[Kli92] Wilhelm Klingenberg. *Lineare Algebra und Geometrie*. Springer, 1992.

[Kni99] Günter Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *Proceedings ECOOP 99*, 1999.

[Kni00] Günter Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, Universität Bonn, Institut für Informatik III, 2000.

[Laf00] Chris Laffra. Jikes Bytecode Toolkit. http://www.alphaworks.ibm.com/tech/jikesbt, 2000.

[LZ97] Han Bok Lee and Benjamin G. Zorn. BIT: A tool for instrumenting Java bytecodes. In USENIX, editor, *USENIX Symposium on Internet Technologies and Systems Proceedings, Monterey, California, December 8–11, 1997*, pages 73–82, Berkeley, CA, USA, 1997. USENIX.

[MC98] Kathy Walrath Mary Campione. *The Java Tutorial Second Edition: Object-Oriented Programming for the Internet.* Java Series. Addison Wesley, 1998.

[MS98] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In Eric Jul, editor, *Proceedings ECOOP '98*, LNCS 1445, 1998.

[Obj00] JGL Version 3.1. http://www.objectspace.com/jgl/prodJGL.asp, 2000.

[ser] The Java Tutorial: Object Serialization. http://java.sun.com/docs/books/tutorial/essential/io/serialization.html.

[SUN97] Java Native Interface Specification. http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html, 1997.

[SUN00a] Java 2 Platform, Standard Edition, Version 1.3. http://java.sun.com/j2se/1.3/, 2000.

[SUN00b] Java Technologie and XML. http://java.sun.com/xml/, 2000.

[Tai00] The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/, 2000.

[TL99] Frank Yellin Tim Lindholm. *The Java Virtual Machine Specification (2nd Ed).* Java Series. Addison Wesley, 1999.

[Tol00] Robert Tolksdorf. Programming Languages for the Java Virtual Machine. http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html, 2000.

[Ven99] Bill Venners. *Inside the Java 2 Virtual Machine.* Mc Graw Hill, 1999.

[WH99] Seth White and Mark Hapner. JDBC 2.1 API. Technical report, Sun Microsystems Inc., 1999.