Diplomarbeit

Vergleich der Programmierkonzepte Vererbung, Generizität und Reflexion in Java und Eiffel

ausgeführt
am

Institut für Computersprachen
Technische Universität Wien

unter Anleitung von
ao. Univ.Prof. Dipl.-Ing. Dr.tech. Franz Puntigam

durch

Mathias Ziehmayer
Matr. Nr. 9525996

Wien, 2004 _____

Zusammenfassung

In dieser Arbeit wird die Umsetzung der Programmierkonzepte Vererbung, Generizität und Reflexion in den objektorientierten Programmiersprachen Java und Eiffel miteinander verglichen. Dabei steht die Verwendbarkeit (*Usability*) dieser Konstrukte in den beiden Programmiersprachen im Vordergrund.

Die einzelnen Sprachkonstrukte werden anhand aussagekräftiger Beispiele analysiert, die die Stärken und Schwächen der jeweiligen Umsetzung aufzeigen. Zusätzlich werden die dahinterliegenden Konzepte erläutert und alternative Ansätze in anderen Programmiersprachen gezeigt.

Eine Besonderheit von Eiffel ist die Möglichkeit, Eingangsparameter von Methoden in einer Vererbungshierarchie kovariant neu zu definieren, was ausdrucksstark ist, aber die Typsicherheit stark einschränkt; Java verfolgt in diesem Bereich einen konservativeren Ansatz.

Während in Eiffel Generizität von Anfang an in die Sprache integriert war, wurde diese in Java erst nachträglich hinzugefügt, wodurch einige Kompromisse in der Umsetzung nötig waren.

Reflexion wurde in die meisten verbreiteten objektorientierten Programmiersprachen erst relativ spät hinzugefügt. Java bietet mittlerweile eine recht gute Unterstützung von Reflexion, während Eiffel mit einem sehr ungewöhnlichen Ansatz eine stark limitierte Form der Reflexion bietet.

Abstract

This thesis compares the implementation of the programming concepts inheritance, generics and reflection in the object-oriented languages Java and Eiffel. The main focus is on the usability of these constructs.

The language constructs are analyzed by meaningful examples that show the strengths and weaknesses of the respective implementation. The underlying concepts are illustrated and alternative approaches in other programming languages are shown.

An unusal feature of Eiffel is that in an inheritance hierarchy covariant redefinitions of a method's input parameters are allowed. This is expressive, but compromises type safety; Java is more conservative in that regard.

While generics were always part of Eiffel, they were introduced to Java retrospectively, which made some compromises necessary. Reflection was added to most object-oriented languages relatively late. New versions of Java have good support for reflection, while Eiffel has very limited support for reflection using an unorthodox approach.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Zielsetzung und Vorgehensweise

Es soll verglichen werden, wie Vererbung zusammen mit Untertypbeziehungen, Generizität und Reflexion in Java und Eiffel umgesetzt werden. Die Konzepte, die hinter diesen Sprachkonstrukten stehen, sollen herausgearbeitet werden.

Der zentrale Aspekt dieser Arbeit ist, wie verständlich und nützlich diese Sprachkonstrukte für den Programmierer sind. Dabei werden deren Mächtigkeit und Verständlichkeit gegeneinander abgewogen. Auf die Integration in die Sprache und deren Standard-Libraries, versteckte Fehlerquellen und die Eignung, reale Anforderungen zu modellieren, wird das größte Augenmerk gelegt. Als Ergebnis soll die Verwendbarkeit oder *Usability* der einzelnen Sprachkonstrukte in Java und Eiffel deutlich werden.

Der Vergleich zwischen Eiffel und Java ist deshalb besonders interessant, weil beide Sprachen am „Reißbrett“ entstanden sind und nicht evolutionäre Weiterentwicklungen ursprünglich nicht-objektorientierter Sprachen sind, die gewisse Kompromisse bei der Umsetzung der zu vergleichenden Sprachkonstrukte eingehen müssen. Zu dem Zeitpunkt, als die beiden Sprachen entworfen wurden, insbesondere im Fall von Java, gab es bereits umfangreiche Erfahrungswerte mit anderen objektorientierten Programmiersprachen.

Um den Umfang der Arbeit im gebotenen Rahmen zu halten und gleichzeitig eine ausführliche Behandlung der einzelnen Themen zu ermöglichen, wurde der Vergleich auf drei Programmierkonzepte beschränkt. Vererbung wurde deshalb gewählt, weil diese ein wesentlicher Aspekt der objektorientierten Programmierung ist. Generizität ist ein für den Programmierer äußerst nützlicher Abstraktionsmechanismus, der von verschiedenen objektorientierten Programmiersprachen auf sehr unterschiedliche Art und Weise aufgegriffen wurde. Reflexion ist ein Konzept, das ursprünglich aus der Welt der funktionalen Programmiersprachen stammt, aber auch zunehmend in objektorientierten Sprachen Verbreitung findet.

Im Mittelpunkt des Vergleichs stehen aussagekräftige, praxisnahe Beispiele, die so-

wohl Stärken als auch Limitierungen der Umsetzung in den beiden Programmiersprachen aufzeigen. Die theoretischen Konzepte, die hinter den Sprachkonstrukten stehen, werden anhand dieser Beispiele erläutert. Technische Details treten dabei etwas in den Hintergrund und werden nur dort ausführlich behandelt, wo diese aus reiner Anwendersicht Konsequenzen für die Verwendbarkeit der zu vergleichenden Sprachkonstrukte haben, etwa wenn Implementierungsdetails versteckte Fallen oder unintuitive Nebeneffekte bedingen.

Die Beispiele wurden so gewählt, dass sie Problemstellungen repräsentieren, die sich dem Programmierer in der Praxis häufig stellen. Dabei wird verglichen, welche Lösungsansätze Java und Eiffel dem Programmierer für diese Probleme bieten und welche, möglicherweise unvorhergesehenen Konsequenzen sich daraus ergeben. Zusätzlich werden alternative Ansätze und die Implementierung der zu vergleichenden Konstrukte in anderen Programmiersprachen ausgeleuchtet, um die Stärken und Schwächen der konkreten Umsetzung in Java und Eiffel besser einordnen zu können.

1.2 Überblick über die zu vergleichenden Sprachen

1.2.1 Java

Java wurde 1995 von SUN Microsystems eingeführt. Ursprünglich ist Java aus der Programmiersprache *Oak* entstanden, die von Sun unter der Leitung von James Gosling als vereinfachte Version von C++ [30] entwickelt wurde. Java ist eine statisch typisierte, objektorientierte Sprache, die interpretiert wird.

Folgende Zielsetzungen sind laut Sun beim Design von Java ausschlaggebend gewesen:

- Einfachheit
- Vollständige Objekt-Orientierung
- Robustheit
- Sicherheit
- Plattformunabhängigkeit / Architekturneutralität
- Interpretierung durch eine virtuelle Maschine
- Gute Performance

Der Java-Compiler erzeugt einen maschinenneutralen Zwischencode, den sogenannten *Bytecode*, der auf dem Zielsystem von einem Laufzeitsystem, der *Java Virtual Machine (JVM)* interpretiert wird. Zusätzlich verfügt Java über standardisierte Libraries, die architekturspezifische Details abstrahieren. Durch diese Mechanismen ist Java plattformunabhängig, d.h. jede Plattform, die über eine entsprechende JVM

verfügt, kann den Bytecode von Applikationen ausführen; der Quellcode muss dabei nicht vorhanden sein.

Java verwendet einen Syntax, der sehr stark an C angelehnt ist und relativ kompakten Code ermöglicht. Die Erfahrungen, die mit C++ gesammelt wurden, waren ein wichtiger Einfluss auf das Design der Sprache. Java wird oft als eine stark vereinfachte, „programmiererfreundliche“ Version von C++ bezeichnet.

Java unterstützt keine Pointer, nur Referenzen auf Objekte. Pointerarithmetik ist nicht möglich, da dies als fehleranfällig und nicht portabel angesehen wurde. Ein wesentlicher Aspekt von Java ist auch die Verwendung von *Garbage Collection*, d.h. das Memorymanagement wird durch das Laufzeitsystem der Sprache und nicht explizit vom Programmierer gesteuert. Globale Funktionen und Variablen sind nicht erlaubt, als eine Alternative dazu können Klassen aber über statische Methoden und Felder verfügen.

Seinen anfänglichen Erfolg verdankt Java einer guten, plattformunabhängigen Unterstützung von Netzwerkoperationen und vor allem *Applets*, das sind Applikationen, die in einem Webbrowser laufen können. Alle gängigen Browser sind heute in der Lage, Applets auszuführen. Die Bedeutung von Applets für Java ist jedoch im Laufe der Zeit zurückgegangen, und Java hat als Programmiersprache für Applikationen, Libraries und Middleware aller Art Fuß fassen können. Da Java interpretiert wird, ist die Performance nicht mit der von Sprachen wie C oder C++ vergleichbar, allerdings ist der Abstand durch die Verwendung guter Optimierungen und die Verwendung von *Just in Time (JIT)*-Compilern, die Bytecode während der Ausführung in Maschinencode übersetzen, kleiner geworden.

Java ist auch für Anwendungen auf Plattformen mit limitierten Ressourcen wie etwa *Personal Digital Assistents (PDAs)*, Mobiltelefonen, oder *JavaCards* verbreitet; dafür wird ein spezielles, standardisiertes Subset der Java-Libraries, die *Java Micro Edition (JME)* verwendet.

Der offizielle Compiler für Java, der aus Portabilitätsgründen selbst in Java geschrieben ist, diverse Entwicklungstools und die JVM für eine Vielzahl von Plattformen wird von Sun kostenlos zur Verfügung gestellt. Zusätzlich existieren Entwicklungsumgebungen (*Integrated Development Enviroment (IDE))* von Drittanbietern wie IBM oder Microsoft.

Eine Vielzahl von Dokumenten zu Java ist auf der Website von Sun [26] erhältlich; die Spezifikation von Java ist in [**?**] detailliert beschrieben.

1.2.2 Eiffel

Eiffel wurde ab 1985 von Bertrand Meyer und Jean Marc Nerson konzipiert. 1986 wurde der erste Eiffel-Compiler vorgestellt. Die treibende Kraft hinter dem Design und der Entwicklung von Eiffel ist Bertrand Meyer.

Eiffel wurde als komplett objektorientierte Programmiersprache entworfen, in der viele Konzepte, die zu dieser Zeit relativ neu waren, integriert wurden. Das wesentliche Designziel war eine effiziente, robuste Sprache, die für die Erstellung hochqualitativer Applikationen geeignet ist.

Auch Eiffel wurde stark von den Erfahrungen beeinflusst, die mit C++ gemacht wurden. Ziel war es, die Sprache übersichtlicher und konsistenter zu gestalten, ohne dabei auf die Effizient zu verzichten, die C++ bietet. Eine weitere wichtige Inspiration war die Programmiersprache ADA [37], insbesondere deren generische Konstrukte.

Eiffel ist eine komplett objektorientierte, statisch typisierte Programmiersprache; alle Typen, auch Integer oder Gleitkommazahlen, sind Objekte. Die wesentlichen Designziele von Eiffel waren:

- Volle Objekt-Orientierung
- Umfangreiche Zusicherungen auf Sprachebene (*Design by Contract (DBC)*), die die Entwicklung zuverlässiger und robuster Programme unterstützen
- Effizienz
- Volle Integration von Generizität in die Sprache und Libraries
- Ein ausdrucksstarkes Vererbungssystem

Eiffel verfügt, wie Java, nur über Referenzen. Pointer und Pointerarithmetik sind nicht möglich. Auch Eiffel benutzt *Garbage Collection*, da manuelles Memorymanagement, wie es C++ verwendet, als zu komplex und fehleranfällig angesehen wurde. Die Syntax von Eiffel orientiert sich mehr an Pascal als an C und ist weniger kompakt und „wortreicher“ als die von Java. Globale Variablen oder statische Methoden sind in Eiffel nicht erlaubt, auch explizite Typumwandlungen zur Laufzeit sind nicht möglich.

Insbesondere bei DBC und der konsistenten Integration von Vererbung und Generizität hat Eiffel eine Vorreiterrolle eingenommen. Am Ende der 80er und Beginn der 90er Jahre war Eiffel auch kommerziell einigermaßen erfolgreich, konnte aber nie die gleiche Verbreitung wie C++ finden. Durch die zunehmende Dominanz von Java und später C# wurde Eiffel weiter zurückgedrängt und führt heute im Bereich der kommerziellen Softwareentwicklung eher ein Schattendasein.

Nichtsdestotrotz ist Eiffel äußert einflussreich und hat das Design vieler neuerer Programmiersprachen maßgeblich beeinflusst. Zahlreiche Features, die in Eiffel von Anfang an integriert waren, wurden in andere objektorientierte Programmiersprachen erst nachträglich hinzugefügt.

Eiffel leidet vor allem unter starker Kritik an seinem Typsystem, das zwar ausdrucksstark aber inhärent unsicher ist, und in gewissen Situationen Laufzeitfehler nicht verhindern kann. Trotz vieler Bemühungen, diese Schwäche zu beseitigen, ist das Problem bis heute ungelöst.

Standardisierte Libraries sind ein Bestandteil von Eiffel. Diese sind aber im Vergleich

zu anderen Programmiersprachen wenig umfangreich; auch existieren kaum Libraries von Drittanbietern, was der Popularität von Eiffel weiter abträglich ist.

Ein kommerzieller Compiler für Eiffel ist von der Firma von Bertrand Meyer, *Interactive Software Engineering (ISE)* erhältlich; für nicht kommerzielle Anwendungen ist eine kostenlose Version erhältlich, die für diese Arbeit verwendet wurde. Zusätzlich existiert noch *GNU SmallEiffel*, eine freie OpenSource Implementierung von Eiffel, die sich allerdings aus Lizenzgründen in einigen Libraries von denen in ISE Eiffel unterscheidet.

Die wichtigste Referenz zu Eiffel ist das Buch von Bertrand Meyer, *Eiffel: The Language* [23].

1.3 Überblick über die zu vergleichenden Programmierkonzepte

1.3.1 Vererbung

Vererbung wird oft als eines der entscheidenden Features objektorientierter Programmiersprachen angesehen. Dabei handelt es sich um einen Mechanismus, der es erlaubt, Klassen aus bereits existierenden Klassen abzuleiten und deren Struktur und Verhalten zu erweitern und/oder abzuändern. Dabei werden nur die Unterschiede zur bereits existierenden Klasse angegeben. Alle von einer Klasse geerbten Eigenschaften werden so behandelt, als ob sie in der erbenden Klasse selbst definiert würden.

Diese Vorgehensweise erlaubt eine effiziente und strukturierte Wiederverwendung (*Reuse*) von bereits vorhandenem Code. Die Oberklasse ist von den Abänderungen und Erweiterungen in einer abgeleiteten Klasse nicht betroffen; für Benutzer von Oberklassen ist dieser Vorgang völlig transparent.

Zusätzlich zu Vererbung existieren in praktisch allen objektorientierten Sprachen noch Untertypbeziehungen, die auch als *enthaltender Polymorphismus* bezeichnet werden. Stehen zwei Klassen zueinander in einer Untertypbeziehung, so kann eine Instanz der Unterklasse anstelle einer Instanz der Oberklasse verwendet werden (*Ersetzbarkeitsprinzip*). Vererbung und Untertypbeziehungen sind zwei verschiedene Konzepte, die aber in den allermeisten populären Programmiersprachen miteinander verknüpft und als ein einziges Sprachkonstrukt realisiert sind: wenn eine Kasse von einer anderen erbt, so ist sie gleichzeitig auch deren Untertyp. Auf das Verhältnis zwischen Vererbung und Untertypbeziehungen wird in dieser Arbeit detailliert eingegangen.

Die Art und Weise, wie in einer abgeleitete Klasse Features der Oberklasse abgeändert werden können, ist in allen objektorientierten Programmiersprachen gewissen Regeln unterworfen. Prinzipiell ergeben sich folgende Möglichkeiten:

Erweiterung: In der abgeleiteten Klasse werden neue Eigenschaften hinzugefügt.

Überschreiben: Features der Oberklasse werden in der Unterklasse abgeändert. Die entsprechenden Versionen, die die Oberklasse definiert können zusätzlich bestehen bleiben oder auch nicht.

Entfernen: Die Unterklasse entfernt gewisse Eigenschaften, über die die Oberklasse verfügt. Da diese Vorgehensweise in jedem Fall das Ersetzbarkeitsprinzip verletzt, ist dies nur in wenigen verbreiteten Programmiersprachen möglich (Eiffel erlaubt dies).

Je nachdem, ob eine abgeleitete Klasse von einer oder mehreren Oberklassen erben kann, spricht man von *Einfach-* oder *Mehrfachvererbung*. Sprachen wie C++ oder Eiffel unterstützen echte Mehrfachvererbung, während etwa C# oder Java nur eine Mischform unterstützen, die zwar nur Einfachvererbung, aber mehrfache Untertypbeziehungen erlaubt.

Die erste Programmiersprache, die Klassen und Vererbung als wesentliches Sprachkonstrukt eingeführt hat, war Simula-67 [10], einer auf Algol basierenden Sprache. Diese Konzepte wurden weiterentwickelt und in Smalltalk-80 [14], einer dynamisch typisierten Sprache, erstmals konsequent als wesentliches Paradigma umgesetzt. Die erste Version von Smalltalk wurde in Simula geschrieben. Parallel dazu wurden fast alle populären modularen Programmiersprachen (z.B. C, Pascal, BASIC, Modula, Ada) um objektorientierte Sprachkonstrukte erweitert. Besonders die Popularität von C++ verhalf der objektorientierten Programmierung Mitte der 80er Jahre zum Durchbruch. Ein moderner Nachfolger von Simula ist BETA [21], diese Sprache ist allerdings außerhalb der akademischen Welt kaum verbreitet.

1.3.2 Generizität

Generizität und Vererbung sind zwei Konzepte, die aus der Sicht des Programmierers einen gewissen Überschneidungsbereich haben. Als Generizität wird die Möglichkeit bezeichnet, Klassen oder Methoden durch *Typparameter* zu parametrisieren. Dies wird auch als *generischer Polymorphismus* bezeichnet.

Typparameter sind dabei nur Platzhalter für konkrete Typen, die später anstelle der Typarameter verwendet werden; die entsprechende Bindungen müssen bei der Verwendung generischer Konstrukte angegeben werden. Wenn durch einen Typparameter selbst gewisse Einschränkungen bezüglich der Typen, die ihn ersetzen können impliziert werden, spricht man von *gebundener Generizität*.

Generizität erlaubt es daher, gewissermaßen Vorlagen oder Schablonen für Klassen oder Methoden zu erzeugen, die dann für alle möglichen Ersetzungen der Typparameter verwendbar sind, was ein mächtiges Abstraktionskonzept darstellt. Besonders häufig wird Generizität bei Containerklassen verwendet. Etwa könnte eine Klasse `Stack` definiert werden, die über einen Typparameter `T` verfügt, der angibt, welchen Typ die Elemente haben, die von diesem Stack verwaltet werden. Ein Integerstack würde etwa durch den Ausdruck `Stack<int>`, ein Stack von Strings durch

`Stack<String>` repräsentiert werden. Die internen Mechanismen zur Verwaltung des Stacks müssen nur einmal, nicht für jede mögliche Ersetzung von `T` implementiert werden.

Die erste Programmiersprache, die umfangreiche Unterstützung für Generizität erlaubte, war ADA [37]. C++ verfügt mit *Templates* über mächtige aber relativ komplexe generische Konstrukte; die Popularität von C++ hat wesentlich zur Verbreitung von Generizität in anderen Programmiersprachen beigetragen. Eiffel war eine der ersten objektorientierten Programmiersprachen, die (gebundene) Generizität konsequent und übersichtlich in die Sprache und auch die Standard-Libraries integrierte. In Java und anderen neueren objektorientierten Sprachen wie etwa C# oder VB.NET, wurde Generizität erst später hinzugefügt, wodurch sich Probleme mit der Rückwärtskompatibilität zu früheren Sprachversionen ergeben. Auf diese Problematik wird in dieser Arbeit genauer eingegangen.

1.3.3 Reflexion

Reflexion ist die Möglichkeit einer Programmiersprache, zur Laufzeit Informationen über die Struktur und das Verhalten des Programms selbst herauszufinden und/oder abzuändern. In den meisten traditionellen, modularen Programmiersprachen gehen diese Metainformationen verloren, wenn ein Programm in die Zielsprache (normalerweise Maschinencode) übersetzt wird. Für die Unterstützung von Reflexion ist es daher notwendig, diese Informationen als Metadaten in der Zielsprache zu erhalten.

Die Metainformationen über die Struktur und das Verhalten eines Systems werden durch Reflexion als reguläre Sprachkonstrukte zur Verfügung gestellt; diesen Vorgang nennt man *Reifikation (Vergegenständlichung)*. Informationen über eine Klasse werden beispielsweise in einer zugeordneten Metaklasse zugänglich gemacht (*reifiziert*). Diese Metaklasse erlaubt es dann etwa, alle zur Klasse gehörigen Methoden, Konstruktoren und Felder zu ermitteln und diese auch über die Metaklasse aufzurufen oder zu verändern. Es wäre auch denkbar, die Struktur und das Verhalten einer Klasse zur Laufzeit abzuändern oder zu erweitern; dies ist allerdings in den meisten Programmiersprachen nicht möglich.

Für statisch typisierte Programmiersprachen, die in Maschinencode übersetzt werden, ist eine gute Unterstützung für Reflexion sehr schwierig. Interpretierte Sprachen können normalerweise wesentlich mächtigere Reflexionsmechanismen zur Verfügung stellen. Modernere objektorientierten Sprachen wie Java oder C#, die in einen plattformunabhängigen Zwischencode übersetzt werden, der dann am Zielsystem in einer virtuellen Maschine interpretiert wird, können relativ einfach gute Unterstützung für Reflexion bieten.

Die erste Programmiersprache, die umfangreiche Unterstützung für Reflexion erlaubte, war LISP [15]. LISP ist eine interpretierte, funktionale Programmiersprache die vor allem im Bereich der künstlichen Intelligenz (KI) verbreitet ist. Viele Varianten von LISP erlauben die nahezu uneingeschränkte Abänderung des Programms selbst

während der Ausführung. Im Bereich der objektorientierten Programmiersprachen war Smalltalk-80 ein Vorreiter bei der Unterstützung von Reflexion; Smalltalk ist dynamisch typisiert und wird interpretiert. Eiffel und C++ bieten nur minimale reflexive Mechanismen.

1.4 Gliederung der Arbeit

Die restliche Arbeit gliedert sich in folgende Abschnitte:

Vererbung leuchtet die Sprachkonstrukte für Vererbung und Untertypbeziehungen in Java und Eiffel detailliert aus. Abstrakte Klassen, Einfach- und Mehrfachvererbung sowie Zusicherungen und Objektverhalten werden beschrieben.

Generizität beschreibt die Unterstützung generischer Sprachkonstrukte in den beiden Sprachen. Die Behandlung primitiver Typen, das verwendete Übersetzungsschema und die Probleme, die sich im Zusammenhang mit Rückwärtskompatibilität in Java ergeben, werden behandelt. Die praktische Verwendung von Generizität wird anhand eines umfangreicheren Beispiels in beiden Sprachen gezeigt.

Reflexion zeigt, welche Ansätze Java und Eiffel bezüglich reflexiver Sprachkonstrukte verfolgen. Die wesentlichen Unterschiede und Einschränkungen werden herausgearbeitet.

Zusammenfassung ist ein Überblick über die Ergebnisse der Arbeit.

Am Ende jedes Kapitels werden die Ergebnisse des Vergleiches zusammengefasst, zusätzlich wird gezeigt, welche alternativen Ansätze in anderen Programmiersprachen existieren.

Kapitel 2

Vererbung

2.1 Abstraktionskonzepte

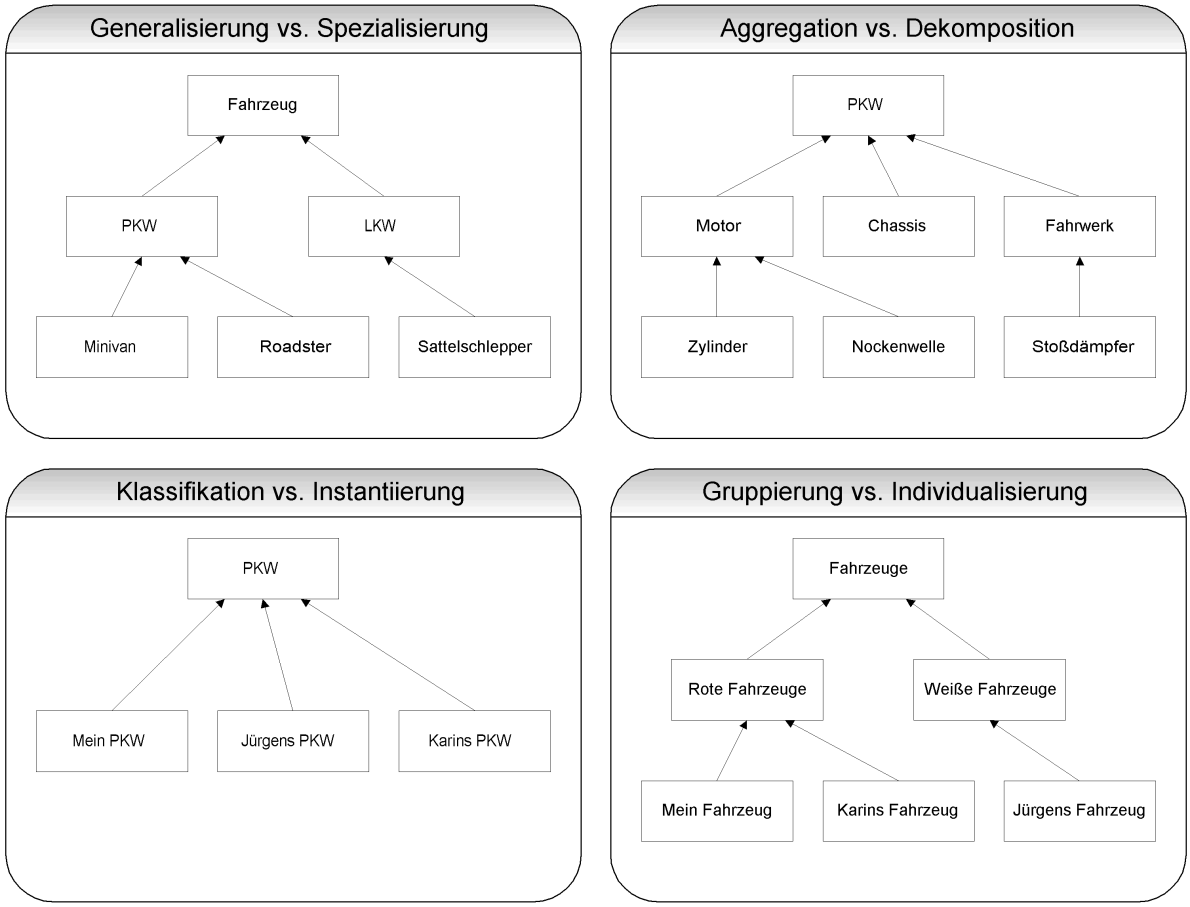
Antero Taivaalsaari [32] teilt die am häufigsten verwendeten Abstraktionsmechanismen in folgende Klassen:

Generalisierung / Spezialisierung: Generalisierung betont die gemeinsamen Eigenschaften von Objekten und vernachlässigt Unterschiede zwischen Objektkategorien. Spezialisierung stellt das entgegengesetzte Konzept dar.

Aggregation / Dekomposition: Aggregation betont die Details einer Beziehung als Ganzes und vernachlässigt Details von Komponenten. Dekomposition stellt das entgegengesetzte Konzept dar.

Klassifikation / Instantiierung: Klassifikation betont die Eigenschaften einer Klasse als Ganzes und vernachlässigt spezifische Details von Instanzen. Instantiierung stellt das entgegengesetzte Konzept dar.

Gruppierung / Individualisierung: Gruppierung betont die Eigenschaften von Objekten als Gruppe und vernachlässigt Details der einzelnen Objekte. Individualisierung stellt das entgegengesetzte Konzept dar.



Normalerweise wird durch Vererbung eine *Spezialisierung* modelliert. Allerdings bieten objektorientierte Programmiersprachen normalerweise keine Möglichkeit, sicherzustellen, dass Vererbung tatsächlich für konzeptuelle Spezialisierung verwendet wird; in der Praxis wird Vererbung häufig zur Modellierung anderer Abstraktionskonzepte zweckentfremdet. Besonders häufig ist dies bei *Mehrfachvererbung* der Fall, die oftmals zur Modellierung einer Aggregationsbeziehung verwendet wird.

Weiters gibt es derzeit wenige Möglichkeiten, durch eine Programmiersprache selbst zu erzwingen, dass Vererbung zur Modellierung von Spezialisierungsbeziehungen verwendet wird. Einige Versuche sind unternommen worden, mittels geeigneter Regeln die konzeptuellen Beziehungen zwischen Objekten zu garantieren. Peter Wegner [36] hat beispielsweise vier Kompatibilitätsregeln (*Compatibility Rules*) entworfen:

Überschreibung ist die schwächste dieser Kompatibilitätsregeln und erlaubt die freie Neudefinition und sogar Entfernung von Operationen in einer Subklasse.

Namenskompatibilität erlaubt es Subklassen, Operationen neu zu definieren, verlangt aber, dass deren Namen konsistent bleiben.

Signaturkompatibilität verlangt völlige syntaktische Kompatibilität zwischen Klassen und deren Subklassen

Verhaltenskompatibilität verlangt, dass das *Verhalten* von Operationen in Klassen und deren Subklassen konsistent bleibt.

Die ersten drei dieser Anforderungen werden auch als „Nicht-strikte Vererbung“ bezeichnet, und sind relativ leicht durch die Programmiersprache selbst zu garantieren. Verhaltenskompatibilität wird auch als „Strikte Vererbung“ bezeichnet, ist jedoch weniger einfach zu erzwingen. Java ist laut dieser Einteilung auf der Ebene der Signaturkompabtibilität angesiedelt, während Eiffel mittels *Design by Contract* (siehe Abschnitt 2.6.1) versucht, auch Verhaltenskompatibilität zu erreichen.

2.2 Vererbung, Untertypbeziehungen und *ist-ein*-Beziehungen

Folgende Beziehungen zwischen Klassen treten in der Praxis häufig auf. [19]

Vererbung: Hier wird eine Klasse durch die Erweiterung und/oder Abänderung einer anderen Klasse erzeugt. Dieser Mechanismus zielt in erster Linie auf die Wiederverwendung von Code der Oberklasse ab.

Untertypbeziehungen: Untertypbeziehungen beschreiben die Ersetzbarkeit von Objekten in einer Objekthierarchie. Eine Instanz einer abgeleiteten Klasse kann anstelle einer Instanz der Oberklasse verwendet werden. Ob die abgeleitete Klasse auch die Implementierung der Oberklasse weiterverwenden kann, ist dabei irrelevant.

Ist-ein-Beziehungen: Eine *ist-ein-Beziehung* modelliert die konzeptuelle Spezialisierung von Klassen. Dabei stellt die abgeleitete Klasse eine speziellere Variante der Oberklasse dar, so wie es sich aus der Aufgabenstellung bzw. der „realen Welt“ ergibt. *Ist-ein-Beziehungen* können, müssen aber nicht durch Untertypbeziehungen darstellbar sein; dies hängt auch stark von der verwendeten Programmiersprache ab.

Ist-ein-Beziehungen stellen also die konzeptuelle Modellierung der Aufgabenstellung dar. Diese müssen vom Softwareentwickler auf geeignete Art und Weise auf die ihm zur Verfügung stehenden Sprachkonzepte abgebildet werden.

Vererbung und Untertypbeziehungen werden in vielen Programmiersprachen durch ein einziges Sprachkonstrukt realisiert, weshalb *Ist-ein-Beziehungen* häufig durch dieses Sprachkonzept abgebildet werden. Manche Programmiersprachen, in denen nur Einfachvererbung (siehe Abschnitt 2.4) möglich ist, bieten trotzdem die Möglichkeit von mehrfachen Untertypbeziehungen.

2.2.1 Untertypen in Java

In Java ist ein Typ U ein Untertyp des Typs T, wenn er mittels `extends` oder `implements` als solcher deklariert ist und die folgenden Eigenschaften erfüllt:

- Für jede Variable in T existiert eine entsprechende Variable identischen Typs in U
- Für jede Methode in T gibt es eine entsprechende Methode gleichen Namens in U, wobei
  - die Typen aller Parameter der Methoden identisch sein müssen
  - der Rückgabewert der Methode in U identisch mit dem Typ des Rückgabewertes der Methode in T sein muss. In Java 1.5 kann der Rückgabewert der Methode in U ein Untertyp der Methode in T sein.
  - die Methode in U nicht mehr Exceptions als die Methode in T deklarieren darf, und jede Exception in U ein Untertyp der entsprechenden Exception in T sein muss

Für Arrays gilt:

- Ist U ein Untertyp von T, so ist ein Array von U ein Untertyp eines Arrays von T.

2.2.2 Untertypen (Konformität) in Eiffel

In der Eiffel-Terminologie wird anstatt des Begriffs *Untertyp* häufig *Konformität* verwendet. Ein Typ X ist zum Typ Y konform, wenn X ein Untertyp von Y ist. In Eiffel gelten folgende Regeln für Untertypbeziehungen [23]:

Ein Typ X ist dann zu einem Typ Y konform, wenn genau eine der folgenden Regeln zutrifft:

- X und Y sind identisch.
- X und Y sind Klassen, X hat keine generischen Parameter und Y führt X in seiner `inherit`-Sektion an.
- X und Y sind Klassen, wobei X die Form *P[U1, U2,...,Un]* hat und die `inherit`-Sektion von Y *P[V1, V2,...,Vn]* enthält, wobei jedes *Vi* zum entsprechenden *Ui* konform ist.
- Y hat die Form `like` *anchor* und der Typ von *anchor* ist zu X konform.
- Es existiert ein Typ Z, sodass sowohl Y zu Z als auch Z zu X konform sind.

2.3 Abstrakte Klassen und Interfaces

2.3.1 Interfaces in Java

Zur Modellierung reiner Untertypbeziehungen ohne Wiederverwendung von Code können in Java *Interfaces* verwendet werden. In einem Interface werden eine Reihe von Methoden deklariert, jedoch ohne eine konkrete Implementierung; diese Methoden werden in Java als *abstrakt* bezeichnet. Das Schlüsselwort `abstract` ist im Falle von Interfaces optional, da alle Methoden implizit abstrakt sind.

Eine Klasse, die ein Interface implementiert, muss die Implementierung aller im Interface deklarierten, abstrakten Methoden enthalten, ansonsten ist die Klasse selbst abstrakt und kann nicht instantiiert werden. In Java kann jede Klasse beliebig viele Interfaces implementieren. Wenn eine Klasse ein Interface implementiert, so ist sie ein Untertyp des jeweiligen Interfaces.

Das folgende Besipiel veraunschaulicht diese Vorgehensweise:

```
public interface Vergleichbar {
  public boolean gleich(Object a);
  public boolean groesserAls(Object a);
}
```

Hier wird das Interface *Vergleichbar* deklariert, das die Methoden *gleich* und *groesserAls* enthält. Der Parameter dieser Methoden ist jeweils vom Typ `Object`, da nicht eingeschränkt werden soll, welche Klassen dieses Interface später einmal implementieren.

```
public class Vektor2D implements Vergleichbar {
  public Vektor2Da(int aX, int aY) {
    x = aX;
    y = aY;
  }

  public boolean gleich(Object a) {
    if(a instanceof Vektor2D) {
      Vektor2D other = (Vektor2D) a;
      if(x == other.x && y == other.y) {
        return true;
      } else {
        return false;
      }
    // typen nicht gleich
    } else {
      return false;
    }
  }

  public boolean groesserAls(Object a) {
    if(a instanceof Vektor2Da) {
```

```
      Vektor2D other = (Vektor2D) a;
      if((x * x + y * y) > (other.x * other.x + other.y * other.y)) {
        return true;
      } else {
        return false;
      }
    // typen nicht gleich
    } else {
      return false;
     // raise exception?
    }
  }

protected double x;
protected double y;
}
```

Die Klasse *Vektor2D* implementiert das Interface und stellt eine konkrete Implementierung der abstrakten Methoden zur Verfügung. *Vektor2D* ist damit ein Untertyp von *Vergleichbar* und kann an dessen Stelle verwendet werden.

In Java ist eine kovariante Neudefinition von Methoden nicht möglich, d.h. die Parameter einer Methode eines Untertyps müssen den gleichen Typ haben wie die entsprechenden Parameter des Obertyps. In der Klasse *Vektor2D* ist der Typ der Eingangsparameter der Vergleichsmethoden durch das Interface *Vergleichbar* auf `Object` festgelegt.

Dadurch ergibt sich in der Praxis häufig das Problem, dass zahlreiche Typüberprüfungen und -umwandlungen zur Laufzeit nötig sind. Das Interface *Vergleichbar* etwa soll in einer Vielzahl von Klassen implementierbar sein, um eine möglichst große Ausdrucksstärke zu erzielen. Da aber nicht festgelegt ist, welche Klassen dieses Interface später einmal implementieren werden und Parameterneudefinitionen in Java in Untertypen invariant sind, muss der Parameter der Methode *gleich* ganz allgemein als `Object` deklariert werden. Diese Klasse verfügt allerdings nicht über die zu vergleichenden Eigenschaften, daher kann im Methodenrumpf auf diese nicht direkt zugegriffen werden. Der Parameter von *gleich* muss daher zur Laufzeit in den passenden Typ umgewandelt werden. Mit dem `instanceof`-Operator wird sichergestellt, dass als Eingangsparameter der richtige Typ übergeben wurde.

Folgende Probleme ergeben sich dabei:

- Der Compiler kann nicht feststellen, ob der Aufruf der Methode mit dem richtigen Parameter erfolgt. Da in Java alle Objekte von der Oberklasse `Object` abgeleitet sind, ist es möglich, dass die Typumwandlung zur Laufzeit scheitert, wenn nicht der richtige Typ übergeben wird.
- Ein Laufzeitfehler kann zwar vermieden werden, wenn vor der Typumwandlung im Methodenrumpf die Kompatibilität des Parameters mittels des `instanceof` Operators überprüft wird, eine statische Überprüfung durch den Compiler ist aber auch mit dieser Methode nicht möglich.

- Das Objektverhalten im Falle einer Typinkompatibilität ist nicht befriedigend zu lösen. Denkbar ist das Auslösen einer Exception, oder aber das Zurückliefern eines speziellen Rückgabewertes, der unter Umständen sogar missverständlich sein kann (z.B. Verwendung des logischen Rückgabewerts *nicht gleich* auch im Falle eines Typfehlers, wobei die Unterscheidung zu ungleichen Objekten des korrekten Typs nicht möglich ist). Jedenfalls ist keiner dieser Lösungsansätze eine adäquate Alternative zur statischen Typüberprüfung durch den Compiler.

Ab Java 1.5 [26] sind auch *generische* Interfaces Bestandteil der Sprache; diese bieten einige Lösungsansätze zu den oben geschilderten Problemen. Generische Interfaces werden detailliert im Abschnitt 3.9 behandelt.

2.3.2 Abstrakte Klassen als Interfaces in Eiffel

Eiffel bietet mit abstrakten oder, in Eiffel-Terminologie, *deferred* Klassen ein mit Interfaces vergleichbares Sprachkonstrukt. Da in Eiffel Mehrfachvererbung möglich ist, ist keine Unterscheidung zwischen Interface und abstrakter Klasse notwendig. Wird ein Feature mit dem Schlüsselwort `deferred` deklariert, so bedeutet dies, dass kein Methodenrumpf vorhanden ist. Sind ein oder mehrere Features einer Klasse als deferred deklariert, so muss die gesamte Klasse selbst als deferred markiert werden, und kann nicht instantiiert werden. Abgeleitete Klassen müssen eine konkrete Implementation aller als deferred deklarierter Methoden zur Verfügung stellen, um instantiiert werden zu können. Dieser Mechanismus stellt also eine Obermenge von Interfaces / abstrakten Klassen in Java dar.

Da Eiffel kovariant (siehe Abschnit 2.5.1) ist, sind Typumwandlungen und -überprüfungen zur Laufzeit wie in Java nicht nötig (und auch gar nicht möglich). Das nächste Beispiel zeigt diesen Mechanismus:

```
deferred class
  VERGLEICHBAR

feature
  gleich(a: like Current): BOOLEAN is deferred end

feature
  groesser_als(a: like Current): BOOLEAN is deferred end

end
```

Die Klasse *VERGLEICHBAR* wird als `deferred` deklariert und kann wie ein Interface in Java verwendet werden. Der Eingangsarameter der Vergleichsmethoden kann als `like Current` deklariert werden; man spricht in diesem Fall von *Anchored Types*. Dies bedeutet, dass eine abgeleitete Klasse in einer konkreten Implementierung der Methode diesen Parameter durch einen Untertyp der Klasse *VERGLEICHBAR* ersetzen kann, wie im folgenden Beispiel durch *VEKTOR2D*:

```
  class
    VEKTOR2D inherit VERGLEICHBAR
    redefine gleich, groesser_als end

  create
    make


  feature make (a_x, a_y: DOUBLE) is
    do
      x := a_x
      y := a_y
    end

  feature gleich(a: VEKTOR2D): BOOLEAN is
    do
      if
        a.x = x AND a.y = y
      then
        Result := true
      else
        Result := false
      end
   end

  feature groesser_als(a: VEKTOR2D): BOOLEAN is
    do
      if
        (x * x + y * y) > (a.x * a.x + a.y * a.y)
      then
        Result := true
      else
        Result := false
    end
  end


feature {VEKTOR2D}
    x: DOUBLE
    y: DOUBLE
end
```

Die kovariante Neudefinition der Eingangsparameter der Methoden *gleich* und *groesser_als* erlaubt ohne die Verwendung von Casts den direkten Zugriff auf die zu vergleichenden Eigenschaften. Obwohl *Anchored Types* auf den ersten Blick einen großen Vorteil gegenüber den Typumwandlungen zur Laufzeit in Java darstellen, ist diese Vorgehensweise inherent unsicher, da Laufzeitfehler, sogenannte *Catcalls*, auftreten können. Im Gegensatz zu unerlaubten Typumwandlungen in Java verursachen diese Catcalls keine klar definierten, abfangbaren Exceptions, sondern führen zu völlig unspezifiziertem Programmverhalten. Catcalls werden detailliert im Abschnitt 2.5.2 behandelt.

2.4 Einfach- und Mehrfachvererbung

2.4.1 Einleitung

Mehrfachvererbung ist ein äußerst kontroversielles Sprachkonstrukt. Über Vor- und Nachteile von Mehrfachvererbung ist sehr viel diskutiert worden, ohne dass dabei ein Konsens erzielt wurde [29].

Mehrfachvererbung bedeutet, dass eine Klasse von mehreren Oberklassen erben kann, während bei Einfachvererbung eine Klasse nur eine einzige Oberklasse haben kann. Viele Programmiersprachen, wie etwa Java, können zwar nur von der Implentierung einer einzigen Oberklasse erben, können aber mittels anderer Sprachkonstrukte (Interfaces in Java) zu mehreren Klassen in einer Untertypbeziehung stehen. Eiffel unterstützt echte Mehrfachvererbung und relativ mächtige Sprachkonstrukte, um Mehrdeutigkeiten aufzulösen.

2.4.2 Vorteile Mehrfachvererbung

Ghan Bir Singh [29] beschreibt folgende wichtige Anwendungen von Mehrfachvererbung:

- Multiple Independent Protocols
- Mixin-Vererbung (Mix and Match)
- Submodularität
- Trennung zwischen Interface und Implementierung

Multiple Independent Protocols beschreibt Situationen, in denen eine Klasse durch die Kombination voneinander unabhängiger, eigenständiger Klassen erzeugt wird. Das unten angeführte Beispiel, in denen eine neue Klasse *Amphibienfahrzeug* aus den eigenständigen Klassen *Auto* und *Boot* erzeugt wird, fällt in diese Kategorie. Der Vorteil dieser Vorgehensweise ist, dass vorhandener Code produktiv wiederverwendet werden kann und dass Coderedundanz und Inkonsistenzen vermieden werden können.

Mixin-Vererbung [3] ist eine Technik, bei der Klassen extra zum Zweck der späteren Kombinierung in eine andere Klasse erzeugt werden. Häufig handelt es sich dabei um generelle Eigenschaften oder Funktionalität, die für viele Klassen nützlich sein kann. Die Verwendung von Mixin-Vererbung unterstützt modulares Design und die Wiederverwendung von häufig verwendetem Code. Ein Beispiel wäre eine Klasse *Serialize*, die jeder Klasse, die von ihr erbt, erlaubt, ihren Zustand auf ein permanentes Speichermedium zu transferieren (wozu jedoch eine gewisse Sprachunterstützung z.B. durch Reflexion nötig ist).

Submodularität beschreibt eine Vorgehensweise, bei der eine „Zielklasse“ speziell durch das Erben von eigens dafür erzeugten Oberklassen erzeugt wird. Die Klas-

sen, die zur Zielklasse kombiniert werden, müssen für sich selbst nicht unbedingt sinnvoll verwendbar sein und eine Untertypbeziehung ist unter Umständen gar nicht nötig. Der Vorteil dieser Vorgehensweise besteht darin, dass komplexe Klassen in handhabbare, in sich geschlossene Abschnitte aufspalten werden können, die die Modularisierung des Designs unterstützt. Der Übergang zur *Mixin-Vererbung* ist dabei fließend.

Unter **Trennung zwischen Interface und Implementierung** versteht man das Aufspalten einer Klasse in die eigentliche Implementierung und das reine Interface. In Java wird dies direkt durch den Interfacemechanismus, wie in Abschnitt 2.3.1 beschrieben, erreicht; in Eiffel kann dies durch die Verwendung von *deferred* Klassen erreicht werden.

2.4.3 Probleme Mehrfachvererbung

Folgende Hauptnachteile von Mehrfachvererbung sind erkennbar:

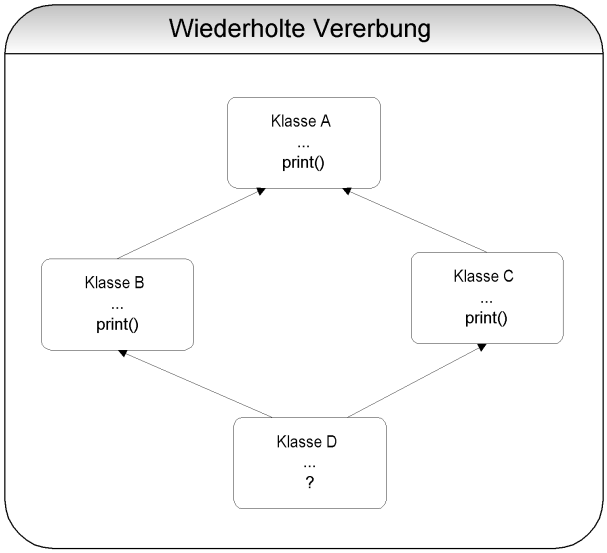
- Namenskonflikte
- Methodenkombination
- Komplexität
- Mißbrauch
- Wiederholte Vererbung

Namenskonflikte treten auf, wenn eine Klasse von zwei oder mehreren Klassen erbt, die Methoden oder Eigenschaften mit dem jeweils gleichen Namen besitzen. Laut Bertrand Meyer [23] stellt dies jedoch kein konzeptuelles Problem dar, sondern ist ein Resultat einer ungünstiger Wahl von Namen; daher ist das Problem syntaktischer Natur und nicht eine inherente Eigenschaft von Klassen und Vererbung. Weiters merkt Meyer an, dass dieses Problem nicht in den Oberklassen liegt (da diese für sich ja korrekt und konsistent sind), sondern in der Klasse begründet ist, die die inkompatiblen Oberklassen vereint. Daher muss dieses Problem auch in der abgeleiteten Klasse gelöst werden. Eiffel bietet zu diesem Zweck die Möglichkeit, Features in einer abgeleiteten Klasse umzubenennen.

Häufig haben Klassen Initialisierungsmethoden (Konstruktoren). Wenn eine Klasse von mehreren Oberklassen erbt, so muss die Möglichkeit der **Methodenkombination** vorgesehen sein, d.h. die abgeleitete Klasse muss gezielt alle Initialisierungsmethoden gleichen Namens aller Oberklassen aufrufen können.

Ein weiteres Problem von Mehrfachvererbung ist **Mißbrauch**. Der häufigste Fall ist der Missbrauch von Mehrfachvererbung für eine konzeptuelle Aggregation. Beispielsweise könnte eine Klasse *Auto* erzeugt werden, indem von den Klassen *Motor*, *Chassis* und *Fahrwerk* geerbt wird; konzeptuell richtig wäre in diesem Fall allerdings die Aggregation der Einzelkomponenten in *Auto*. *Auto* stellt keinen sinnvollen Untertyp

seiner Komponenten dar. Da dieses Problem auf der konzeptuellen Ebene angesiedelt ist, gibt es keinen vernünftigen Weg, solchen Missbrauch zu verhindern.



Wiederholte Vererbung tritt auf, wenn eine Klasse ein Feature von einer Oberklasse mehr als einmal erbt. Dieses Problem wird auch als *Diamond Inheritance* bezeichnet. Im skizzierten Beispiel erben die Klassen B und C jeweils von der Klasse A; die Klasse D hingegen erbt von sowohl B als auch C. D erbt also Features von A mehrfach; besonders problematisch ist die Situation wenn die duplizierten Features in einer oder beiden der Klassen B oder C verändert werden. Die häufigsten Lösungsansätze für dieses Problem sind:

- Wenn die duplizierten Features identisch sind, so werden die „überschüssigen“ entfernt, und der Konflikt ist beseitigt. Sind die duplizierten Features hingegen nicht identisch, so ist dies ein Fehler. Die Klasse D muss in diesem Fall die die Inkompatibilität explizit auflösen. In Eiffel oder aber auch in *Extended Smalltalk* wird dieser Konflikt durch Umbenennung von Features beseitigt.
- Der Vererbungsgraph kann linearisiert werden; in diesem Fall ist eine Duplizierung von Features nicht möglich, jedoch hängt die konkrete Ausprägung der Features in der Klasse D von der Reihenfolge der Linearisierung ab, weshalb dieser Ansatz äußerst unbefriedigend ist.
- Es ist möglich, die wiederholt vererbten Klassen zu duplizieren; im obigen Fall würden implizit die Klassen *A1* und *A2* entstehen. Dies beseitigt zwar das Problem der wiederholten Vererbung, wandelt es allerdings im Wesentlichen in ein Namenskonfliktproblem um. *CommonObjects* verwendet dieses Lösungsansatz.

Keine dieser Ansätze kann allerdings das fundamentale Problem der wiederholten Vererbung wirklich lösen.

2.4.4 Mehrfachvererbung in Eiffel

Das folgenden Beispiel zeigt, wie eine Klasse *Amphibienfahrzeug* von sowohl *Auto* als auch *Boot* abgeleitet werden kann:

```
class
  AUTO
  create
      make

  feature make is
      do
      km_total := 0
      end

  feature fahre_nach(km: INTEGER) is
    do
      km_total := km_total + km
    end

  feature get_versicherung_preis: INTEGER is
    do
      Result := km_total * 5
    end

  feature {AUTO}
  km_total: INTEGER

  end
```

Die Klasse *AUTO* definiert die Features *fahre_nach*, *get_versicherung_preis* und *km_total*. Die Kosten der Versicherung richten sich dabei nach dem Gesamtkilometerstand.

```
  class
      BOOT

  feature make is
    do
      sm_total := 0
    end


  feature schwimme_nach(sm: INTEGER) is
    do
      sm_total := sm_total + sm
    end

  feature get_versicherung_preis: INTEGER is
    do
      Result := sm_total * 2
    end
```

```
  feature {AUTO}
  sm_total: INTEGER

end
```

BOOT definiert die Features *schwimme_nach*, *get_versichung_preis* und *sm_total*. Sowohl *BOOT* als auch *AUTO* sind konkrete Klassen, die für sich alleine verwendbar sind.

```
class
  AMPHIBIENFAHRZEUG
inherit
  AUTO redefine make, get_versicherung_preis end
  BOOT undefine make, get_versicherung_preis end

  create
      make

  feature make is
    do
      km_total := 0
      sm_total := 0

    end

  feature get_versicherung_preis:INTEGER is
    do
      Result := km_total * 5 + sm_total * 3
    end

end
```

Die Klasse *AMPHIBIENFAHRZEUG* erbt von sowohl *AUTO* als auch *BOOT*. Da beiden Oberklassen über die namesgleichen Methoden *make* und *get_versicherung_preis* verfügen, muss dieser Konflikt aufgelöst werden; dazu werden die Schlüsselworte `redefine` und `undefine` verwendet. In diesem Fall werden die Versionen von *AUTO* verwendet und durch die passende Implementierung für ein Amphibienfahrzeug ersetzt. Die abgeleitete Klasse *AMPHIBIENFAHRZEUG* verfügt über die Features *fahre_nach*, *schwimme_nach*, *km_total*, *sm_total* und *get_versicherung_preis*. Einzig der Konstruktor(*make*) sowie die Methode zur Berechnung des Versicherungspreises müssen abgeändert werden, die restlichen Features werden direkt von den Oberklassen geerbt.

2.4.5 Mehrfachvererbung und Interfaces in Java

In Java ist keine direkte Mehrfachvererbung von Code möglich. Bis zu einem gewissen Grad kann dieses Verhalten durch Interfaces simuliert werden:

```
public interface Schwimmfaehig {
  public void schwimmeNach(int sm);

  // nicht moeglich
  // protected int smTotal;

}
```

Das Interface *Schwimmfaehig* deklariert die abstrakte Methode *schwimmeNach*. Ein Feld, das die Anzahl der von einer schwimmfähigen Klasse zurückgelegten Seemeilen enthält, kann das Interface allerdings nicht enthalten, da nur Konstanten in Interfaces deklariert werden können. Alternativ wäre es möglich, eine Methode *getSmTotal* zu deklarieren.

```
public class Auto {
    public Auto() {
      kmTotal = 0;
  }

  public void fahreNach(int km) {
    kmTotal = kmTotal + km;
  }

  public int getVersicherungPreis() {
    return kmTotal * 5;
  }

  protected int kmTotal;
}
```

Auto ist eine nicht abstrakte Klasse, die äquivalent zu *AUTO* in Eiffel über die Methoden *fahreNach*, *getVersicherungPreis* und ein Feld für den Gesamtkilometerstand verfügt.

```
public class Amphibienfahrzeug extends Auto implements
Schwimmfaehig {
  public Amphibienfahrzeug() {
    super();
    smTotal = 0;
  }

  // Implementierung kann nicht von Schwimmfaehig geerbt werden
  public void schwimmeNach(int sm) {
    smTotal = smTotal + sm;
  }

  public int getVersicherungPreis() {
    return kmTotal * 5 + smTotal * 3;
  }

  // kann nicht von Schwimmfaehig geerbt werden
```

```
    protected int smTotal;
}
```

Amphibienfahrzeug erbt im obigen Beispiel die Implementierung von *Auto* und implementiert zusätzlich das Interface *Schwimmfaehig*. Die Implementierung einer möglichen Klasse *Boot* kann aber nicht vererbt werden, diese muss von *Amphibienfahrzeug* selbst zur Verfügung gestellt werden. Der Nachteil dieser Methode gegenüber der echten Mehrfachvererbung von Eiffel liegt darin, dass existierender, funktionierender Code nicht weiterverwendet werden kann und dupliziert werden muss.

Eine Möglichkeit, Mehrfachvererbung zu simulieren, besteht darin, die Vererbungshierarchie durch das Einführen von Zwischentypen zu linearisieren. Das kann aber zu einer sehr großen Anzahl zusätzlichen Klassen führen, die unter Umständen für sich allein keinen Sinn ergeben, und die Wartbarkeit stark einschränken.

Eine weitere Möglichkeit, Mehrfachvererbung zu simulieren, ist die *Delegation*. Dabei werden die Oberklassen in einer Wrapper-Klasse durch Aggregation miteinander kombiniert, was auch automatisiert geschehen kann. Ein solches Tool für Java ist *Jamie*. [35]

2.5 Kovarianz vs. Invarianz

In einem Untertyp können die Typen der Elemente des Obertyps abgeändert werden. Prinzipiell ergeben sich dabei drei Möglichkeiten:

Invarianz: Der Typ eines Elements im Untertyp ist identisch zum Typ des entsprechenden Elements im Obertyp.

Kovarianz: Der Typ eines Elements im Untertyp ist ein Untertyp des entsprechenden Elements im Obertyp.

Kontravarianz: Der Typ eines Elements in Untertyp ist der Obertyp des entsprechenden Elements im Obertyp.

In diesem Zusammenhang ist die kovariante Neudefinition von *Eingangsparametern* einer Methode von besonderer Bedeutung; kovariante Neudefinitionen von Ergebnistypen sind unproblematisch. Wenn im Folgenden ohne genauere Einschränkung von *Kovarianz* gesprochen wird, so ist stets Kovarianz von Eingangsparametern gemeint.

Eiffel ist durchgängig kovariant; dies war ein wesentliches Designziel, da kovarinate Neudefinitionen eine große Ausdrucksstärke besitzen und in vielen Situationen eine „natürliche“ Modellierung ermöglicht. Der entscheidende Nachteil dieser Vorgehensweise ist, dass unsichere Vorgänge möglich sind, die zum Zeitpunkt der Kompilierung nur sehr schwer erkannt werden können. Diese Problematik wird im Abschnitt 2.5.2 detailliert behandelt.

In Java sind die meisten Neudefinitionen von Elementen in Untertypen invariant. Eine Ausnahme stellen Exceptions und Arrays dar: diese sind kovariant, d.h. eine Methode des Untertyps darf eine speziellere Exception (Untertyp) als die entsprechende Methode der Oberklasse auslösen; bei Arrays ist ein Array eines Untertyps selbst ein Untertyp eines Arrays des Obertyps. Ab Java 1.5 sind auch Ergebnistypen von Methoden kovariant. Dieser Ansatz bietet nicht die gleiche Ausdrucksstärke wie die Mechanismen in Eiffel, ist jedoch (mit Ausnahme der kovarianten Arrays, siehe Abschnitt 2.5.5) inherent sicherer.

2.5.1 Kovarianz in Eiffel

Ein typisches Problem, bei dem eine kovariante Neudefinition von Eingangsparametern sinnvoll ist, sind *binäre Methoden*. Im folgenden Beispiel haben die Klassen *VEKTOR2D* und *VEKTOR3D* jeweils die Methode `skalar_produkt`, die als Eingangsparameter eine Klasse vom selben Typ verlangen. In Eiffel lässt sich dieser Umstand einfach modellieren.

```
class
  VEKTOR_2D

  create
      make_2D

  feature make_2D(a_x, a_y: DOUBLE) is
      do
      x := a_x;
      y := a_y;
      end

  feature skalar_produkt(other: VEKTOR_2D):DOUBLE is
      do
      Result := x * other.x + y * other.y
      end

  feature{VEKTOR_2D}
  x: DOUBLE; y: DOUBLE;
end

class
  VEKTOR_3D

inherit VEKTOR_2D redefine skalar_produkt end

  create
    make_3D

   feature make_3D(a_x, a_y, a_z: DOUBLE) is
      do
          make_2D(a_x, a_y)
          z := a_z
```

```
      end

   feature skalar_produkt(other: VEKTOR_3D): DOUBLE is
     do
       Result := x * other.x + y * other.y * z * other.z
     end


  feature{VEKTOR_3D} z: DOUBLE
end
```

Die Klasse *Vektor_3D* ist ein Untertyp der Klasse *Vektor_2D*, diese Klassen stellen einen zwei- bzw. dreidimensionalen Vektor dar. Beide verfügen über eine Methode `skalar_produkt`, die das Skalarprodukt zweier Vektoren der gleichen Dimension berechnet. Durch die kovariante Neudefinition des Eingangsparameters *other* der Methode `skalar_produkt` lässt sich dieser Umstand sehr einfach modellieren. Die Problematik dieses Ansatzes wird im nächsten Abschnitt näher erläutert.

2.5.2 Catcalls

Ein *CAT(Changing Availability or Type)-call*, oder kurz *Catcall* ist ein polymorpher Methodenaufruf, bei dem ein Typfehler mit undefiniertem Programmverhalten zur Laufzeit auftritt, obwohl die statische Typüberprüfung erfolgreich war. Folgende Ursachen für Catcalls sind möglich [16], [24]:

- Eine kovariante Neudefinition von Methodenparametern oder Variablen
- Verwendung generischer Parameter (siehe Abschnitt 3.8)
- Deaktivieren von Features (*Feature Hiding* siehe Abschnitt 2.8.1)

Das folgende Beispiel löst einen Catcall aus:

```
class
  CATCALL

  feature do_catcall is
    local
      v_2D: VEKTOR_2D
      v_3D: VEKTOR_3D
      poly_2D: VEKTOR_2D
    do
      create v_2D.make_2D(3.0, -2.0)
      create v_3D.make_3D(-6.0, 1.0, 2.0)

      -- ok
      print(v_2D.skalar_produkt(v_2D))
      -- ok
      print(v_3D.skalar_produkt(v_3D))
      -- type error
```

```
      print(v_3D.skalar_produkt(v_2D))
      -- ok, aber...
      print(v_2D.skalar_produkt(v_3D))
      -- catcall
      poly_2D := v_3D
      print(poly_2D.skalar_produkt(v_2D))
    end
  end
end
```

Hier wird durch die Anweisung `print (poly_2D.skalar_produkt(v_2D))` nach der Zuweisung `poly_2D := v_3D` ein Catcall ausgelöst. Die Variable `poly_2D` enthält eine Referenz auf ein Objekt vom Typ *Vektor3D*, dessen Methode *skalar_produkt* aufgerufen wird; als Parameter wird ein Objekt vom Typ *Vektor2D* übergeben. Im Methodenrumpf wird dann versucht, auf das nicht vorhandene Feature *z* zuzugreifen, was ein undefiniertes Programmverhalten zur Folge hat; jedenfalls kann kein korrektes Ergebnis ermittelt werden, da der Aufruf logisch falsch ist.

Das Grundproblem im obigen Beispiel ist die kovariante Neudefinition des Parameters `other` in der Methode `skalar_produkt`. Zur Compilezeit kann nicht festgestellt werden, welchen Typ die Variable `poly_2D` zum Zeitpunkt des Methodenaufrufs referenziert. In Java oder anderen Programmiersprachen, die in bezug auf Eingangsparametertypen nicht kovariant sind, wäre dieser Fehlerfall nicht möglich.

2.5.3 Lösungsansätze

Es sind zahlreiche Lösungsansätze für das Problem der Catcalls vorgeschlagen worden [16], [9]. Diese können wie folgt zusammengefasst werden:

Compilerunterstützung: Es wird versucht, Catcalls mit Unterstützung von intelligenten Compilern zu erkennen, die zur Compilezeit einen Fehler verursachen, wenn eine unsichere Operation erkannt wird. Diese Vorgehensweise wäre zwar die ideale Lösung des Problems, stellt sich letztlich aber als unrealistisch heraus. Die Komplexität der Überprüfung ist enorm, und prinzipiell kann nicht der letztendliche Ausführungspfad des Programms vorhergesagt werden. Alternativ kann (etwa mit Heuristiken) eine Voraussage getroffen werden, welche Operationen gefährlich sein *könnten*, dies würde aber unweigerlich auch legitime und sichere Aufrufe betreffen; außerdem sind deterministische und klar definierte Regeln eine Grundanforderung an statisch typisierte Programmiersprachen. Eine weitere Alternative ist, alle Operationen, die einen Catcall auslösen können, nicht zu unterbinden, sondern lediglich ein Warning auszugeben. Der Nachteil liegt darin, dass allein durch Warnings keine Typsicherheit zu erzielen ist. Desweiteren sind kovariante Neudefinitionen inherent unsicher, weshalb auch für korrekte Operationen Warnings ausgegeben würden, was möglicherweise das generelle Ignorieren dieser Warnings zur Folge hätte. Ein weiteres grundsätzliches Problem dieser Vorgehensweise besteht darin, dass für diese Analysen der

gesamte Sourcecode zur Verfügung stehen muss, was bei der Verwendung von Libraries oftmals nicht der Fall ist.

Kovarianz als Überladen: In diesem Ansatz ersetzen kovariante Neudefinitionen nicht das Original, sondern es wird zusätzlich zur ursprünglichen Definition eine überladene Version erzeugt (überladene Methoden sind in Eiffel normalerweise nicht möglich). Im oben anführten Beispiel hätte die Klasse *Vektor3D* zwei Versionen der Methode `skalar_produkt`; in der einen Version wäre der Typ des Parameters *other* *Vektor2D*, in der anderen Version wäre der Typ *Vektor3D*. Welche der beiden Methoden aufgerufen wird, wird dynamisch durch den Typ des Parameters bestimmt, was Catcalls komplett verhindern würde. Der Hauptnachteil dieses Lösungsansatzes ist, dass es für den Programmierer undurchsichtig ist, welche Ausprägung einer Methode aufgerufen wird. Auch können die überladenen Methoden jeweils unterschiedliche Vor- und Nachbedingungen haben, was die Nachvollziehbarkeit von Zusicherungen deutlich erschwert.

Expandierte Vererbung: Expandierte (oder nicht-konforme) Vererbung, ist eine reine Implementationsvererbung; eine abgeleitete Klasse erbt zwar die Implementierung und die Features der Oberklasse, ist aber nicht deren Untertyp. Dies verhindert polymorphe Referenzen auf die entsprechenden Klassen, und damit auch Catcalls. Dieser Ansatz ist keine generelle Lösung des Problems, da eine ausschließliche Anwendung von expandierter Vererbung extreme Einschränkungen mit sich bringen würde, ist aber eine gute Lösung in Situationen, in denen zwar Vererbung, aber nicht unbedingt eine Untertypbeziehung erwünscht ist (z.B. *Submodularität*, siehe Abschnitt 2.4). Bertrand Meyer ist der Ansicht, dass die konsequente Anwendung von expandierter Vererbung in der Praxis die meisten Catcalls verhindern kann, da diese Art der Vererbung sehr häufig verwendet wird [16].

Recast: Um Catcalls zu verhindern, muss in diesem Ansatz für jede kovariante Neudefinition eine spezielle Methode definiert werden, die die Typumwandlung handhabt. Dafür wird das neue Schlüsselwort `recast` eingeführt, dass nach jeder kovarianten Neudefinition eine Methode anführen muss, die die Typumwandlung explizit durchführt. Wenn eine Umwandlung nicht sinnvoll möglich ist, so kann der Programmierer in dieser Methode eine Exception auslösen; jedenfalls wird ein definiertes Verhalten erzwungen. Diese Vorgehensweise löst das Problem des undefinierten Programmverhaltens bei Catcalls, allerdings ist diese Strategie sehr ähnlich zu expliziten Typumwandlungen bei Sprachen, die keine Kovarianz erlauben.

Keiner dieser Ansätze ist jedoch wirklich befriedigend; Unterstützung durch den Compiler scheint der optimale Weg zu sein, doch aufgrund der enormen Komplexität gibt es noch keinen realen Compiler, der Catcalls zuverlässig erkennt.

2.5.4 Invarianz in Java

In Java ist eine kovariante Neudefinition des Eingangsparameters der Methode `skalarProdukt` nicht möglich.

```
public class Vektor2D {
  public Vektor2D(double aX, double aY) {
    x = aX;
    y = aY;
  }

  public double skalarProdukt(Vektor2D other) {
    double skalar;
    skalar = x * other.x + y * other.y;
    return skalar;
  }

  protected double x;
  protected double y;
}

public class Vektor3D extends Vektor2D {
  public Vektor3D(double aX, double aY, double aZ) {
    super(aX, aY);
    z = aZ;
  }

  public double skalarProdukt(Vektor3D other) {
    double skalar;
    skalar = x * other.x + y * other.y + z * other.z;
    return skalar;
  }

  protected double z;
}
```

Im obigen Beispiel wird die Methode *skalarProdukt* nicht durch die abgeleitet Klasse *Vektor3D* ersetzt, sondern *überladen*. Das bedeutet, es existieren zwei Versionen der Methode nebeneinander; welche Methode aufgerufen wird, hängt von Typ des Parameters *other* zum Zeitpunkt des Methodenaufrufs ab. Dadurch werden zwar Catcalls unmöglich gemacht, allerdings sind logisch falsche Aufrufe möglich. Beispielsweise könnte *skalar_produkt* von *Vektor3D* mit einem zweidimensionalen Vektor berechnet werden, da *Vektor3D* ja beide Ausprägungen der Methode besitzt. Solche Fehler können sehr schwer zu entdecken sein.

2.5.5 Kovariante Arrays in Java

In Java stellen Arrays eine interessante Ausnahme im Typsystem dar: diese sind kovariant, d.h. `String[]` ist ein Untertyp von `Object[]`, was zu einer ähnlichen Problematik wie in Eiffel führt (siehe Abschnitt 2.5.2).

```
public class ArrayCov {
  public ArrayCov() {

    strArray = new String[3];
    objArray = strArray;
  }

  public void failure() {
    // ok
    objArray[0] = new String("String 1");
    // run time failure
    objArray[1] = new Double(2.25);
    // run time failure
    objArray[2] = new Object();
  }

  protected String[] strArray;
  protected Object[] objArray;
}
```

Hier ist *strArray* ein Untertyp von *objArray*, da `String` ein Untertyp von `Object` ist. Die Zuweisung `objArray = strArray` im Konstruktor ist daher legal, die Anweisungen `objArray[1] = new Double(2.25)` und `objArray[2] = new Object()` lösen allerdings eine Exception zur Laufzeit aus. Zur Compilezeit kann nicht vorhergesagt werden, welchen Typ *objArray* zum Zeitpunkt der Zuweisung referenziert (`String[]` im obigen Beispiel). Dies stellt eine Lücke im Typsystem von Java dar, und es ist nicht leicht nachzuvollziehen, warum dies in Java überhaupt erlaubt ist, da im Gegensatz zu Eiffel nie der Versuch unternommen wurde, durchgängig Kovarianz in die Sprache zu integrieren.

2.6 Objektverhalten

Reine Untertypbeziehungen reichen in vielen Fällen nicht aus, um das konsistente Verhalten von Objekten in einer Vererbungshierarchie zu beschreiben. Es gibt verschiede Ansätze, um dies zu erreichen. Häufig werden Anforderungen an Klassen einfach in Kommentaren oder der Programmdokumentation festgehalten; viele Programmiersprachen verfügen über *Assertions*, das sind boolsche Ausdrücke, die, wenn sie zu *false* evaluieren, einen Fehler auslösen. Andere Programmiersprachen, wie Eiffel, verfügen über noch mächtigere Konstrukte, um das Objektverhalten formal zu

spezifizieren. Dadurch können in einem gewissen Maß diese Anforderungen auch durch die Programmiersprache selbst überprüft werden.

2.6.1 Design by Contract(DBC) in Eiffel

Bertrand Meyer führte *Design by contract (DBC)* als mächtiges Werkzeug zur Entwicklung zuverlässiger Software ein [23]. DBC ist in Eiffel durchgängig und konsistent auf Sprachebene integriert. DBC verwendet Assertions, um die korrekte Implementierung von Klassen, Komponenteninterfaces und internen Datenzuständen zu gewährleisten. Die drei zentralen Bestandteile von DBC sind Vorbedingungen (Preconditions), Nachbedingungen (Postconditions) und Invarianten (Invariants). Vor- und Nachbedingungen gehören zu dabei zu einer einzelnen Methode, während Invarianten sich auf eine Klasse als Ganzes beziehen.

Vorbedingungen beschreiben, welche Bedingungen erfüllt sein müssen, bevor die zugehörige Methode aufgerufen werden kann. Für die Einhaltung von Vorbedingungen ist grundsätzlich der *Client*, d.h. der Verwender einer Softwarekomponente verantwortlich. Vorbedingungen werden mit dem Schlüsselwort `require` am Anfang des Methodenrumpfes definiert.

Nachbedingungen legen fest, welche Bedingungen erfüllt sein müssen, nachdem eine Methode abgeschlossen ist. Für die Einhaltung von Nachbedingungen ist der *Supplier*, d.h. der Author / Hersteller einer Softwarekomponente verantwortlich. Nachbedingungen werden mittels des Schlüsselwortes `ensure` am Ende des Methodenrumpfes festgelegt.

Invarianten sind Bedingungen, die zu jedem Zeitpunkt, d.h. nach dem Aufruf der *Create-Methode* und dann vor und nach dem Aufruf jeder Methode, erfüllt sein müssen. Für die Einhaltung von Invarianten ist der Supplier zuständig. Invarianten werden durch das Schlüsselwort `invariant` definiert.

Im folgenden Beispiel wird eine Klasse zur Repräsentation gerader, ganzer Zahlen dargestellt, wobei diese Eigenschaften mittels DBC sichergestellt werden.

```
class
    EVEN_INT

create
  make

feature make(a: INTEGER) is
  require
    is_even: a \\ 2 = 0
   do
     i := a
   end

feature add(a: INTEGER) is
```

```
  require
    operand_even: a \\ 2 = 0
    operand_positive: a > 0
  do
      i := i + a
  ensure
    updated: i = old i + a
  end

feature multiply(a: INTEGER) is
  require
    operand_positive: a > 0
   do
     i := i * a
   ensure
     updated: i = old i * a
   end

feature {NONE}
i: INTEGER

invariant
  is_even: i \\ 2 = 0

end
```

Die Vorbedingungen `operand_even: a \\ 2 = 0` und `operand_positive: a > 0` legen fest, dass die Argumente der entsprechenden Methoden gerade und positiv sein müssen, wofür der Client der Klasse Sorge zu tragen hat. Die Nachbedingungen, wie beispielsweise `updated: i = old i + a` stellen sicher, dass die Methode den gewünschten Effekt hatte. Das Schlüsselwort `old` vor einer Variable ist dabei der Wert der jeweiligen Variable vor dem Methodenaufruf. Nachbedingungen dienen häufig mehr zu Dokumentationszwecken, sind aber auch ein probates Mittel, um den gewünschten Effekt einer Methode sicherzustellen. Die Invariante `is_even: i \\ 2 = 0` stellt sicher, dass die repräsentierte Zahl zu jedem Zeitpunkt gerade ist. Dies hat, wie die Gültigkeit der Nachbedingungen, der Supplier sicherzustellen.

2.6.2 Zusicherungen durch Dokumentation in Java

In Java gibt es kein mit DBC vergleichbares Konstrukt. Eine häufig verwendete Möglichkeit (auch in zahlreichen anderen Programmiersprachen) ist, in der Programmdokumentation bzw. in Kommentaren diese Zusicherungen zu beschreiben. Der Hauptnachteil besteht natürlich darin, dass die Einhaltung dieser Zusicherungen nicht garantiert werden kann, sondern vom Client / Supplier „freiwillig“ eingehalten werden muss. Es ist dabei auch möglich, dass diese Zusicherungen unwissentlich verletzt werden, da die formale Prüfung entfällt. Insgesamt erschwert dieser Ansatz die Wartbarkeit, Weiterverwendbarkeit und Zuverlässigkeit der Software sehr stark. Nichtsdestotrotz ist dieser Ansatz der in der Praxis am häufigsten verwendete.

Das folgende Beispiel veranschaulicht diese Vorgehensweise:

```
public class EvenInt {

  // a must be even and positive
  public EvenInt(int a) {
      i = a;
  }

  // a must be even and positive
  public void add(int a) {
      i = i + a;
  }

  // a must be positive
  public void multiply(int a) {
      i = i * a;
  }

  // i must always be even
  private int i;
}
```

Hier werden die Zusicherungen einfach als Kommentar zu den Methoden hinzugefügt. Der Programmierer ist für deren Einhaltung verantwortlich, dies kann aber auf Sprachebene nicht erzwungen werden.

Häufig werden die Zusicherungen auch in die externe Programmdokumentation aufgenommen. Diese Methode ist bei kleinen Projekten, die kaum externe Softwarekomponenten verwenden, durchaus praktikabel, desweiteren ist diese Vorgehensweise mangels Alternativen in vielen Programmiersprachen, auch in sehr großen Softwareprojekten, weit verbreitet. Meist wird die Einhaltung der Zusicherungen durch festgeschriebene Codier- und Dokumentationsrichtlinien erreicht.

2.6.3 Assertions in Java

Eine weitere Möglichkeit besteht darin, Zusicherungen explizit durch *Assertions* zu erzwingen. Assertions sind erst seit Java 1.4 Bestandteil der Sprache [26], obwohl diese schon in einer frühen Spezifikation der Sprache enthalten waren, aber aus Zeitmangel nicht berücksichtigt wurden. James Gosling hat dies in einem Interview als einen schweren Fehler bezeichnet. Sun hat auch in Erwägung gezogen, ein vollständiges DBC anstellte des Assertion-Mechanismus der Sprache hinzuzufügen, dies aber mit der Begründung abgelehnt, dies hätte zu schwerwiegende Auswirkungen auf bestehende Libraries und Softwaresysteme; außerdem würde die Komplexität der Sprache dadurch auf ein unerwünschtes Niveau steigen [4]. Vergleichbare Sprachkonstrukte existieren in einer Vielzahl andere Programmiersprachen, etwa das `ASSERT()` Makro in C bzw. C++.

Eine Assertion ist ein boolscher Ausdruck, der, wenn er nicht erfüllt wird (zu *false*

evaluiert), zur Laufzeit eine Exception auslöst. Assertions werden in Java mit `assert expr1;` deklariert, wobei *expr1* der zu evaluierende boolsche Ausdruck ist. Um detaillierte Fehlermeldungen zu erzeugen kann auch die Form `assert expr1: expr2;` verwendet werden; *expr2* ist in dieser Variante ein Ausdruck beliebigen Typs, dessen Wert dem Konstruktor einer Exception übergeben wird. Assertions können deaktiviert werden, daher ist es wichtig, dass diese keine Nebeneffekte enthalten (was aber grundsätzlich als schlechter Programmierstil gilt).

Gegenüber DBC sind folgende Nachteile erkennbar

- Assertions sind weniger formal, sie können überall im Code vorkommen. Die Semantik von Vorbedingungen, Nachbedingungen und Invarianten wird nicht erzwungen.
- Assertions sind schlechter lesbar als die vergleichbaren Konstrukte in DBC, die in Eiffel an prominenter Stelle im Code zu finden sind und die durch automatische Dokumentationstools in die Klassendefinition eingebunden sind.
- Invarianten müssen am Anfang und am Ende jeder einzelnen Methode dupliziert werden, was die Wartbarkeit bei Änderungen der Invarianten erschwert. Obwohl diese natürlich in eine eigene Methode ausgelagert werden kann, ist nicht sichergestellt, dass jede Methode dieser Konvention folgt, d.h. einzelne Methoden könnten die Invarianten nicht überprüfen.
- Assertions stellen nicht wirklich ein neues Sprachkonstrukt dar, sie können einfach durch `if` Statements und Exceptions emuliert werden. In vielerlei Hinsicht sind Assertions nur eine syntaktisch vereinfachte Schreibweise bestehender Konstrukte.
- Assertions werden nicht vererbt. Siehe Abschnitt 2.7.1.

Im folgenden Beispiel werden Assertions verwendet, um sicherzustellen, dass Zusicherungen eingehalten werden:

```
public class EvenIntWithAssertions {

  // a must be even and positive
  public EvenIntWithAssertions(int a) {
      // precondition
      assert(a % 2 == 0); // is_even
      i = a;
      // invariant
      assert(i % 2 == 0); // is_even
  }

  // a must be even and positive
  public void add(int a) {
      int oldI = i;
      // precondition
      assert(a % 2 == 0); // is_even
      assert(a > 0);      // is positive
```

```
        i = i + a;
        // postcondition
        assert(i == oldI + a); // updated
        // invariant
        assert(i % 2 == 0); // is_even
    }

    // a must be positive
    public void multiply(int a) {
        int oldI = i;
        // precondition
        assert(a > 0);      // is positive
        i = i * a;
        // postcondition
        assert(i == oldI * a); // updated
        // invariant
        assert(i % 2 == 0); // is_even
    }

    private int i;
}
```

Hier werden die Vor- und Nachbedingungen sowie die Invarianten der Klasse, die eine gerade ganze Zahl darstellt, durch Assertions modelliert. Der Ausdruck `assert(a % 2 == 0)` etwa stellt sicher, dass der Parameter a gerade ist, `assert(a > 0)` garantiert, dass a positiv ist. Klasseninvarianten werden am Ende jeder Methode wiederholt, für die Überprüfung der Postcondition muss der alte Wert jeder zu überprüfenden Variable aufgehoben werden.

In Eiffel werden Vorbedingungen *vor* dem Aufruf der Methode von Client überprüft und lösen dort eine Exception aus; in der oben gezeigten Emulation in Java wird die Exception im Supplier ausgelöst und könnte sogar versehentlich durch einen `catch`-Block im Methodenrumpf abgefangen werden.

Obwohl ein ähnlicher Effekt erzielt werden kann wie in Eiffel, ist deutlich zu sehen, dass dieser Ansatz wesentlich unübersichtlicher und unsicherer ist als die Verwendung von DBC.

2.7 Objektverhalten und Vererbung

Zusicherungen in einer Klasse sollten sich in einer geeigneten, konsistenten Art und Weise auf abgeleitete Klassen übertragen. Dabei ist folgende Semantik sinnvoll, wenn U ein Untertyp von T ist [23]

- Jede Vorbedingung in T impliziert eine Vorbedingung in U
- Jede Nachbedingung in U impliziert eine Nachbedingung in T
- Jede Invariante in U impliziert eine Invariante in T

Vorbedingungen in Untertypen können also schwächer, dürfen aber nicht stärker als die entsprechende Vorbedingung im Obertyp sein, während Nachbedingungen und Invarianten in Untertypen stärken sein können, aber nicht schwächer sein dürfen als die im Obertyp. Der Gedanke dahinter ist, dass Untertypen, die anstelle von Obertypen verwendet werden können, nicht mehr vom Client verlangen dürfen und auch nicht weniger Funktionalität bieten dürfen, da sonst das Ersetzbarkeitsprinzip verletzt würde. In Eiffels DBC werden diese Regeln durch die Sprache selbst erzwungen, Assertions in Java garantieren diese semantischen Regeln nicht.

2.7.1 Design by Contract und Vererbung

DBC wurde von Anfang an für die Verwendung in Vererbungshierarchien konzipiert und verwendet die oben genannte Semantik.

Wenn eine Methode einer abgeleiteten Klasse keine Vor- oder Nachbedingung enthält, so erbt sie unverändert die jeweiligen Vor- oder Nachbedingungen der Oberklassen. Dieser Fall tritt in der Praxis am häufigsten auf, üblicherweise werden diese Zusicherungen in einer abstrakten Oberklasse definiert.

Soll die Vorbedingung einer Methode einer abgeleiteten Klasse modifiziert werden, so muss die neue Vorbedingung mit `require else` anstatt von `require` definiert werden. Semantisch entspricht das einer *or else* Verknüpfung der neuen Vorbedingung mit den entsprechenden Vorbedingungen der Oberklassen. Analog dazu muss eine modifizierte Nachbedingung mit `ensure then` anstatt von `ensure` definiert werden; dies entspricht dann einer *and then* Verknüpfung mit den jeweiligen Nachbedingungen der Oberklasse. Invarianten werden implizit mittels *and* mit den Invarianten der Oberklasse verknüpft.

Das folgende Beispiel zeigt wie Zusicherungen in Eiffel vererbt werden:

```
deferred class
  FAHRZEUG

  feature fahre_nach(km: INTEGER) is deferred
    require
      non_neg: km > 0
      fahrzeug_ok: km_bis_service > km
    ensure
      updated: km_bis_service = old km_bis_service - km
    end

  feature service: INTEGER is deferred
    ensure
      service_ok: km_bis_service >= old km_bis_service
    end

  feature
    km_bis_service: INTEGER;
```

```
  feature {FAHRZEUG} invariant
    legal: km_bis_service >= 0
end
```

FAHRZEUG ist eine abstrakte Klasse. Trotzdem können in dieser die Vorbedingungen *fahrzeug_ok*, *non_neg*, die Nachbedinguen *service_ok*, *updated* sowie die Klasseninvariante *legal* definiert werden, die an Unterklassen vererbt werden.

```
class
  PKW inherit FAHRZEUG
  redefine fahre_nach, service end

  create
    make

  feature make(a_diesel: BOOLEAN) is
    do
      diesel := a_diesel
      -- 10000 km Service
      km_bis_service := 10000
   end

  feature fahre_nach(km: INTEGER) is
    do
      km_bis_service := km_bis_service - km
    end

  feature service: INTEGER is
    do
      km_bis_service := 10000
      -- kosten
      if diesel
        then
          Result := 1000
        else
          Result := 500
        end
      end

  feature {PKW} diesel: BOOLEAN
end
```

PKW und *LKW* sind von *FAHRZEUG* abgeleitet und erben die dort gemachten Zusicherungen.

```
class
  LKW inherit FAHRZEUG
  redefine fahre_nach, service end

  create
    make

  feature make(a_achsen: INTEGER) is
```

```
    require
      achsen_ok: a_achsen >= 2 AND a_achsen <= 5
    do
      achsen := a_achsen
      maut_kosten := 0
      -- 5000 km service
      km_bis_service := 5000
    end

  feature fahre_nach(km: INTEGER) is
    do
      km_bis_service := km_bis_service - km
      -- mautkosten 3 cent pro achse und km
       maut_kosten := maut_kosten + km * achsen * 3
    end

  feature service: INTEGER is
    do
      km_bis_service := 5000
      -- Kosten: 1000 Euro pro Achse
      Result := achsen * 1000
    end

  feature {LKW}
    achsen: INTEGER
    maut_kosten: INTEGER
end
```

Im oben angeführten Beispiel werden in einer abstrakten Oberklasse *FAHRZEUG* Zusicherungen definiert, die sich auf die abgeleiteten Klassen *LKW* und *PKW* übertragen. Ein konsistentes Objektverhalten wird durch die Sprache selbst garantiert, unter der Voraussetzung, dass die Vor-, Nachbedingungen und Invarianten geeignet gewählt wurden. Da die Zusicherungen in der abstrakten Oberklasse gemacht wurden, sind diese für den Programmieren übersichtlich an einer einzigen Stelle zusammengefasst, was auch für die Dokumentation von Vorteil ist.

2.7.2 Kovarianz und DBC

Kovarianz und DBC sind die wichtigsten Konzepte, die Eiffel von anderen Programmiersprachen unterscheiden. Beide Konzepte wurden bewusst von Anfang an in das Sprachdesign eingebunden und sind konsequent in Libraries integriert.

Jedoch ist anzumerken, dass Kovarianz und die Regeln von DBC prinzipiell widersprüchlich sind. DBC verlangt, dass Vorbedingungen in Untertypen nicht verschärft werden dürfen. Die Typen von Eingangsparametern einer Methode können durchaus als Vorbedingungen dieser Methode angesehen werden; wird nun in einer abgeleiteten Klasse eine speziellere Variante (Untertyp) eines Eingangsparameters erwartet, so verletzt dies die Grundregeln von DBC. Da beide Konzepte tief in der Sprache verwurzelt sind, ist dieser Konflikt kaum zu lösen.

2.7.3 Assertions und Vererbung

Der Hauptnachteil von Assertions besteht darin, dass sie nicht vererbt werden können. Sie müssen in abgeleiteten Klassen neu definiert werden. Dadurch ergibt sich in umfangreichen Vererbungshierarchien viel Coderedundanz, außerdem kann mit dieser Vorgehensweise nicht garantiert werden, dass die Regeln für die Zusicherungen eingehalten werden, etwa könnte eine Vorbedingung in einem Untertyp gestärkt werden. Weiters können die Zusicherungen nicht, wie in Eiffel, in einer abstrakten Oberklasse gemacht werden, weshalb dieser Ansatz gegenüber DBC in Eiffel deutlich unterlegen ist.

Das nächste Beispiel zeigt, wie Assertions auf eine Vererbungshierarchie angewendet werden:

```
public abstract class Fahrzeug {
  public abstract void fahreNach(int km);
  public abstract int service();

  protected int kmBisService;
}
```

Im Interface *Fahrzeug* können keine Assertions verwendet werden.

```
public class PKW extends Fahrzeug {

  public PKW(boolean aDiesel) {
    diesel = aDiesel;
    // 10000 km Service
    kmBisService = 10000;
    // invariant
    assert(kmBisService >= 0); // legal
  }

  public void fahreNach(int km) {
    // precondition
    assert(km > 0);              // non - neg
    assert(kmBisService > km);  // fahrzeug_ok

    int oldKmBisService = kmBisService;
    kmBisService = kmBisService - km;
    // postcondition
    assert(kmBisService == oldKmBisService - km); // updated
    // invariant
    assert(kmBisService >= 0); // legal
  }

  public int service() {
    int oldKmBisService = kmBisService;
    int kosten;
    kmBisService = 10000;
    // kosten
    if(diesel) {
```

```
      kosten = 1000;
    } else {
      kosten = 500;
    }
    // postcondition
    assert(kmBisService >= oldKmBisService); // updated
    // invariant
    assert(kmBisService >= 0); // legal
    return kosten;
  }

  protected boolean diesel;
}
```

Die Klassen *PKW* und *LKW* implementieren das Interface *Fahrzeug*; jede Klasse muss seine eigenen Assertions definieren.

```
public class LKW extends Fahrzeug {

  public LKW(int aAchsen) {
    achsen = aAchsen;
    mautKosten = 0;
    // 5000 km Service
    kmBisService = 5000;
    // invariant
    assert(kmBisService >= 0); // legal
  }

  public void fahreNach(int km) {
    // precondition
    assert(km > 0);              // non - neg
    assert(kmBisService > km);  // fahrzeug_ok

    int oldKmBisService = kmBisService;
    kmBisService = kmBisService - km;
    // Mautkosten: 3 Cent pro km und Achse
    mautKosten = mautKosten + km * achsen * 3;
    // postcondition
    assert(kmBisService == oldKmBisService - km); // updated
    // invariant
    assert(kmBisService >= 0); // legal
  }

  public int service() {
    int oldKmBisService = kmBisService;
    int kosten;

    kmBisService = 5000;
    // kosten 1000 Euro pro Achse
    kosten = achsen * 1000;

    // postcondition
    assert(kmBisService >= oldKmBisService); // updated
    // invariant
```

```
    assert(kmBisService >= 0); // legal
    return kosten;
  }


  protected int achsen;
  protected int mautKosten;
}
```

Wie man aus dem Beispiel erkennt, müssen die Assertions in jeder Klasse wiederholt werden. Um die korrekte Aktualisierung von Variablen in Nachbedingungen zu prüfen, muss der alte Wert extra in einer Variable zwischengespeichert werden, was den Code noch zusätzlich länger macht. Auch ist nicht sofort ersichtlich, ob und wo diese Assertions abgeändert werden; dies muss durch Kommentare oder die Programmdokumentation erfolgen. Außerdem werden im Gegensatz zu DBC Vorbedingungen erst nach dem Methodenaufruf überprüft.

2.7.4 Alternative Ansätze in Java

Es sind einige Versuche unternommen worden, Java um einen Mechanismus zu erweitern, der DBC entspricht. Einige dieser Erweiterungen sind:

Jass [2] unterstützt Vor- und Nachbedingungen, Klasseninvariaten und Schleifeninvarianten. Ausdrücke in Zusicherungen dürfen keine Nebeneffekte haben. Auf den ursprünglichen Wert einer Variablen vor dem Methodenaufruf und den Rückgabewert der Methode kann in der Nachbedingung zugegriffen werden.

jContractor [17] ist eine Java-Library, die DBC unterstützt. Die Zusicherungen werden in speziell benannten Methoden gemacht, die vom Classloader umgesetzt werden. Auf Werte von Variablen vor einem Methodenaufruf kann zugegriffen werden.

Handshake [12] ist eine dynamisch gelinkte Library, die den Filezugriff der Java Virtual Machine abfängt. Handshake erlaubt es, externe Zusicherung zu existierenden Klassen und Interfaces hinzuzufügen, ohne die Klassen selbst zu ändern. Handshake ist nicht in Java geschrieben und muss daher auf verschiedene Plattformen portiert werden.

FJC [18] ist eine Spracherweiterung für Java, die neben Generizität auch Vor- und Nachbedingungen sowie Invarianten und Assertions einführt. Auf alte Werte von Variablen vor dem Methodenaufruf und auf den Rückgabewert einer Methode kann in der Nachbedingung zugegriffen werden.

2.8 Objekthierarchien

Oft entsprechen reale Objekthierarchien nicht denen, die sich einfach mit Hilfe objektorientierter Sprachen modellieren lassen. Beispielsweise könnte eine Unterklasse eine Eigenschaft nicht haben, über die die überwältigende Mehrheit der anderen Klassen in dieser Vererbungshierarchie verfügt. Oft ergeben sich solche Einschränkungen auch als nachträgliche Anforderungen, weshalb sie nicht immer beim Systemdesign berücksichtigt werden können.

Auch wenn man diese Problematik ganz generell als schlechtes Design abtun könnte, tritt sie in der Praxis häufig genug auf, um spezielle Sprachmechanismen zu rechtfertigen. Oftmals treten solche Probleme auch als Kompromiss zwischen sauberem Design und effizienter Implementierung auf.

2.8.1 Feature Hiding in Eiffel

Eiffel erlaubt es abgeleiteten Klassen, Features von Oberklassen nicht zugänglich zu machen. Dies wird auch als *Feature Hiding* oder *Descendant Hiding* bezeichnet. In Eiffel wird dies durch die Anweisung `export {NONE} methode` in der `inherit`-Sektion der Klasse bewerkstelligt:

```
class
  SAEUGETIER

  create
    make

  feature make(a_alter: INTEGER) is
    do
       alter := a_alter
    end

  feature fortpflanzen: like Current is
  do
    Result.make(0)
  end

  feature schlafen is
  do
     -- schlafen legen
  end

  feature {SAEUGETIER} -- Implementation alter: INTEGER
end

class
  MAULTIER
  -- Maultier kann sich nicht fortpflanzen
  inherit SAEUGETIER export {NONE} fortfplanzen end
```

```
  feature trage_last(gewicht: INTEGER) is
  do
    -- trage die last
    if gewicht > 100
      then
        current.schlafen
    end
  end
end
```

Im diesem Beispiel ist *Maultier* das einzige Tier einer großen Hierarchie, das sich nicht fortpflanzen kann. Daher kann die Methode `fortpflanzen` deaktiviert werden. Diese Lösung ist zwar einfach und pragmatisch, allerdings wird dadurch das Ersetzbarkeitsprinzip verletzt. Ein polymorpher Aufruf der Methode `fortpflanzen` kann einen Catcall auslösen.

2.8.2 Möglichkeiten in Java

Im Java ist ein Vorgehen wie in Java nicht möglich. Die Methode *fortpflanzen* ist zwangsläufig vorhanden:

```
public class Saeugetier {

   public Saeugetier(int aAlter) {
       alter = aAlter;
   }

   public Saeugetier fortpflanzen() {
       return new Saeugetier(0);
   }

   public void schlafen() {
       // schlafen legen
   }

   protected int alter;
}

public class Maultier extends Saeugetier {

   public Maultier(int aAlter) {
       super(aAlter);
   }

   public void trageLast(int gewicht) {
       // trage last
       if(gewicht > 100) {
         schlafen();
       }
```

```
    }

    public Maultier fortpflanzen() {
        // return null ?
        // raise exception ?
        // do nothing ?
        return null;
    }
}
```

Im obigen Beispiel wird in der Methode *fortpflanzen* eine Null-Referenz zurückgeliefert, da sich ein Maultier nicht fortpflanzen kann; dies kann zu Laufzeitfehlern führen, wenn in anderen Komponenten davon ausgegangen wird, dass die Methode immer einen gültigen Wert zurückgibt. Alternativ könnte auch eine Exception ausgelöst werden, diese müsste aber rechtzeitig in das Design integriert werden. Welche Vorgehensweise für den Benutzer die Beste ist, bleibt fraglich.

2.8.3 Lösungsansätze

Feature Hiding ist ein kontroversielles Sprachkonstrukt. Einerseits erlaubt es eine einfache, schnelle Lösung von Inkonsistenzproblemen in Vererbungshierarchien, andererseits gefährdet es das korrekte Systemverhalten. Ist das Deaktivieren von Features nicht möglich, so muss zwangsläufig eine Implementierung angegeben werden, die ein semantisch korrektes Verhalten nicht erreichen kann.

Der offensichtlichste Lösungsansatz ist das Ummodellieren der Vererbungshierarchie. Dies ist aber häufig nicht möglich, da entweder die Inkonsistenz zu einem späten Zeitpunkt auftritt, sodass ein komplett neues Design nicht zu rechtfertigen wäre, oder die Vorteile einer ansonsten konsistenten Hierarchie die Nachteile von gewissen Ausnahmen rechtfertigen.

Unproblematisch ist Feature Hiding allerdings im Falle einer reinen Implementationsvererbung ohne Untertypbeziehung. In Eiffel lässt sich dies durch *expanded* Vererbung erreichen; in Java ist es hingegen nicht möglich, dass eine Klasse von einer anderen erbt, ohne deren Untertyp zu sein.

2.9 Sather

Sather [28, 31] ist eine ursprünglich auf Eiffel basierende Programmiersprache, die in erster Linie auf Einfachheit und größtmögliche Effizienz abzielt. Dabei hat sich ein sehr interesanntes Typsystem entwickelt, das sich deutlich von dem in Eiffel unterschiedet. Die wichtigsten Eigenschaften sind:

- Sather unterscheidet strikt zwischen Vererbung und Untertypbeziehungen; diese

werden durch zwei verschiedene Sprachkonstrukte implementiert und sind von einander komplett unabhängig.

- Es sind sowohl Mehrfachvererbung als auch mehrfache Untertypbeziehungen möglich.
- In Sather gibt es *abstrakte* und *konkrete* Typen; abstrakte Typen werden mit einem `$` Präfix gekennzeichnet.
- Abstrakte Typen dürfen keine Implementierung enthalten, können aber Untertypen haben. Konkrete Typen müssen für alle in ihren Obertypen deklarierten Methoden eine Implementierung bereitstellen, dürfen aber ihrerseits keine Untertypen haben. Konkrete Typen können instantiiert werden, abstrakte nicht.
- Eine Variable kann von einem abstrakten oder konkreten Typ deklariert werden. Variablen, die einen konkreten Typ haben, können ausschließlich diesen referenzieren. Bei Variablen, die einen abstrakten Typ haben, ist Polymorphismus möglich.
- Im Unterschied zu Eiffel sind Neudefinitionen in Untertypen nicht kovariant, sondern kontravariant. Eine Ausnahme bildet der Typ `SAME`, der `like Current` in Eiffel entspricht, und immer den Typ der Klasse selbst hat. Da konkrete Typen keine Untertypen haben können, ist Sather statisch typsicher; Catcalls sind ausgeschlossen.
- Abstrakte Typen können sich sogar als Obertypen anderer Klassen deklarieren.
- Vererbung in Sather ist eine reine Implementationsvererbung. Nur konkrete Typen können voneinander erben, diese müssen aber zueinander in keinerlei Beziehung stehen. Vererbung wird durch das Schlüsselwort `include` realisiert, und es findet tatsächlich eine rein textuelle Inkludierung der Implementation der angegebenen Klasse statt. Namenskonflikte müssen wie in Eiffel durch das explizite Umbenennen von Features aufgelöst werden.
- Vor- und Nachbedingungen sowie Klasseninvarianten sind analog zu Eiffel möglich, allerdings können diese durch die völlige Trennung von Vererbung und Untertypbeziehungen nur in konkreten Typen verwendet werden.

Obwohl Sather seine Ursprünge als ein effizientes, „abgespecktes“ Derivat von Eiffel hat, ist im Laufe der Zeit eine eigenständige Sprache entstanden, die außer einer syntaktischen Ähnlichkeit nicht allzu viele Gemeinsamkeiten mit Eiffel hat. Das Typsystem von Sather ist ungewöhnlich, aber statisch sicher, pragmatisch und scheint für den praktischen Einsatz in größeren Projekten durchaus geeignet. Leider hat Sather ausserhalb der akademischen Welt nie Fuß fassen können und wird seit 1996 vom *International Computer Science Institute (ICSI) Berkeley* nicht mehr weiterentwickelt.

2.10 Resultate

Der größte und bedeutsamste Unterschied zwischen Java und Eiffel ist die durchgängige Verwendung von Kovarianz in Eiffel, inbesonders der Möglichkeit, Eingangsparameter von Methoden kovariant neu zu definieren. Bertrand Meyer ist der Ansicht, Kovarianz erlaube eine natürlichere, ausdrucksstärkere Modellierung von Anforderungen an ein System. In vielen Fällen ist die Möglichkeit der kovarianten Neudefinition offensichtlich von Vorteil; etwa bei der Implementierung von binären Methoden in Vererbungshierarchien. Hier bietet Kovarianz einen deutlichen Vorteile gegenüber den in Java notwendigen häufigen Typüberprüfungen und Typumwandlungen zur Laufzeit.

Der Nachteil, der Kovarianz mit sich bringt, ist allerdings beträchtlich: Typfehler zur Laufzeit, sogenannte *Catcalls* können nicht sicher ausgeschlossen werden; diese führen darüberhinaus zu einem undefiniertem Programmverhalten. Trotz vielfacher Diskussionen und Lösungsvorschläge ist bisher keine zufriedenstellende Lösung gefunden worden, und es ist anzunehmen, dass es keine allgemeingültige Lösung gibt, die das Typsystem von Eiffel nicht stark verkomplizieren und damit auch die „Natürlichkeit“ dieses Ansatzes zunichte machen würde.

Während Eiffel echte Mehrfachvererbung bietet, ist in Java nur Einfachvererbung möglich; mehrfache Untertypbeziehungen ohne Vererbung von Code sind jedoch durch Interfaces möglich, die in erster Linie zur Trennung von Schnittstelle und Implementierung dienen. Mit *deferred* Klassen, die Interfaces in Java entsprechen, und Mehrfachvererbung bietet Eiffel eine Obermenge zu den entsprechenden Sprachkonstrukten in Java. Mehrfachvererbung wird häufig aufgrund ihrer Komplexität und Mehrdeutigkeiten kritisiert. Alle entstehenden Mehrdeutigkeiten müssen in Eiffel durch Umbenennen der miteinander in Konflikt stehenden Features aufgelöst werden.

In anderen Sprachen, wie etwa in C++, dürfen Mehrdeutigkeiten prinzipiell bestehen bleiben, ein Fehler tritt erst dann auf, wenn eine Operation nicht eindeutig aufgelöst werden kann. Ein weiteres Problem von Mehrfachvererbung ist deren missbräuchliche Verwendung, besonders häufig werden Aggregationsbeziehungen unnötigerweise durch Mehrfachvererbung modelliert.

Eiffel bietet ein mächtiges Werkzeug, um ein konsistentes Objektverhalten zu gewährleisten: *Design By Contract* (DBC). Durch Vor- und Nachbedingungen sowie Klasseninvarianten wird ein „Vertrag “ zwischen Supplier und Client einer Klasse geschlossen, der dann durch Eiffel erzwungen wird. Die Zusicherungen vererben sich auch auf sinnvolle Art und Weise auf Untertypen, was das Design von umfangreichen, konsistenten Vererbungshierarchien unterstützt.

Java bietet kein mit DBC vergleichbares Sprachkonstrukt. Zwar können diese Zusicherungen in Kommentaren und Programmdokumentation angeführt werden, die Sprache kann aber deren Einhaltung nicht garantieren. Java 1.4 führt zwar *Assertions* ein, diese stellen aber lediglich eine Minimallösung dar, vor allem, da sich Assertions nicht auf Untertypen übertragen.

Eiffel bietet auch die Möglichkeit des *Feature Hiding*, d.h. eine abgeleitete Klasse kann geerbte Features nicht zur Verfügung stellen. In Java ist dies nicht möglich. Zwar bietet Feature Hiding eine einfache und schnelle Lösung, um Ausnahmen in Vererbungshierarchien zu ermöglichen, allerdings wird dadurch das Ersetzbarkeitsprinzip verletzt, wodurch Catcalls möglich werden.

Abschließend ist anzumerken, dass zwei sehr wichtige Sprachkonstrukte von Eiffel, Kovarianz und DBC, eigentlich nicht miteinander vereinbar sind. DBC verbietet es, in Untertypen Vorbedingungen von Methoden zu verschärfen, allerdings kann der Typ von Eingangsparametern durchaus als Vorbedingung angesehen werden, was im Falle einer kovarianten Neudefinition von Eingangsparametern einer Verletzung der Vorbedingung entsprechen würde.

2.10.1 Übersicht

	Java	*Eiffel*
Mehrfache Untertypbeziehung	`Ja`	`Ja`
Mehrfache Vererbung	`Nein`	`Ja`
Abstrakte Klassen	`Ja (interface)`	`Ja (deferred)`
Vererbung ohne Untertypbeziehung	`Nein`	`Ja (expanded)`
Explizite Typumwandlung	`Ja`	`Nein/durch Zuweisung`
Kovarianz	`Arrays, Ergebnistypen`	`Durchgängig`
Typsicherheit	`+`	`-`
Mechanismus Objektverhalten	`Assertions`	`DBC`
Objektverhalten und Vererbung	`-`	`++`
Feature hiding	`Nein`	`Ja`
Ausdruckstärke	`+`	`++`

Kapitel 3

Generizität

3.1 Eiffel

In Eiffel war Generizität von Anfang an integraler Bestandteil der Sprache. Eiffel war eine der ersten objektorientierten Sprachen, die (Mehrfach-) Vererbung und Generizität konsequent miteinander verbunden haben; in vielen anderen Programmiersprachen hat sich Generizität als zusätzliches Feature später entwickelt und ist oft nicht durchgängig integriert. Zahlreiche Standard-Libraries in Eiffel machen umfangreichen Gebrauch von generischen Parametern.

Eiffel erlaubt die Verwendung von Typparametern ausschließlich als Parameter von Klassen; statische oder globale Methoden existieren in Eiffel nicht.

3.2 Java

In Java war Generizität ursprünglich nicht vorgesehen. Ab der Version 1.5 ist Unterstützung für Generizität nachträglich in die Sprache integriert worden, wobei Abwärtskompatibilität zu früheren Versionen eine unbedingte Designanforderung war. Die wichtigsten Evolutionsschritte waren:

Pizza [27] ist eine Obermenge von Java, in der drei Erweiterungen hinzugefügt wurden: gebundene Generizität, *First Class* Funktionen, und algebraische Datentypen. First Class Funktionen entsprechen im Wesentlichen Funktionspointern wie z.B. in C/C++ oder *Agenten* in Eiffel (siehe Abschnitt 4.3). Algebraische Datentypen haben ihre Wurzeln in funktionalen Programmiersprachen, und erlauben es, einen Typ als eine Reihe von Fallunterscheidungen darzustellen.

Generic Java (GJ) [6] hat sich aus Pizza entwickelt. Algebraische Datentypen und First Class Funktionen wurden nicht weiter verfolgt und sind nicht mehr Bestandteil von Generic Java. Wesentliche Unterschiede bei der gebundenen Generizität sind bessere Einbindung in vorhandene Libraries und die limitierte

Möglichkeit der Reflexion für generische Typen. Des weiteren erlaubt Generic Java nur die Verwendung von Referenztypen als Typparameter, primitive Typen wie *int* oder *float* können nicht verwendet werden.

Java 1.5 [13] Die Spracherweiterungen im Bereich Generizität in Java 1.5 entspricht im wesentlichen GJ. Java 1.5 erlaubt im Gegensatz zu GJ nicht die Verwendung generischer Exceptions (Untertypen von *Throwable* können keine Typparameter haben), dafür ist es möglich, in `throws`-Klauseln Typparameter zu verwenden.

In Java können sowohl Klassen als auch (statische) Methoden über generische Typparameter verfügen. Bei Methoden werden die Typparameter vor der Methodendefinition angegeben.

3.3 Gebundene Generizität

Typparameter sind sowohl in Java als auch in Eiffel gebunden. Das bedeutet, dass eine „Schranke“ angegeben werden muss, und nur Untertypen dieser Schranke können den Typparameter ersetzen. Wird keine Schranke angegeben, so wird implizit `Object` bzw. `ANY` angenommen. Gleichzeitig kann nur auf Features des Schrankentyps zugegriffen werden, was zwar eine einfache Typüberprüfung und eine effiziente Umsetzung durch den Compiler ermöglicht, allerdings die Flexibilität doch einschränkt. In anderen Programmiersprachen, wie etwa in C++ (siehe Abschnitt 4.5.1) oder ADA 95 ist diese Einschränkung nicht vorhanden. In Abschnitt 4.5.1 wird Generizität in C++ näher beschrieben.

In Java werden Typparameter mittels `extends` gebunden. Dies bedeutet, dass der Typparameter nur durch Klassen ersetzt werden kann die ein Untertyp der Schranke sind (*extends*). Als Schranke können auch mehrere Interfaces angegeben werden, es ist auch möglich, sowohl eine Klasse als auch mehrere Interfaces gleichzeitig zu verwenden. Zusätzlich ist es möglich, anonyme Typparameter, sogenannte *Wildcards*, zu verwenden (siehe Abschnitt 3.7).

In Eiffel wird die Schranke mittels des Operators `->` hinter dem Typparameter angegeben; es kann nur eine Schranke angegeben werden.

3.4 Homogene vs. heterogene Übersetzung

Ein wesentlicher Aspekt bei der Implementierung von Generizität in einer Programmiersprache ist die Wahl des Übersetzungsschemas. Bei der *heterogenen* Übersetzung wird für jede konkrete Ersetzung eines Typparameters eine gesonderte Übersetzung in die Zielsprache durchgeführt. Bei einer *homogenen* Übersetzung wird für jede parametrisierte Klasse oder Methode nur eine Version in der Zielsprache erzeugt, die alle möglichen Ersetzungen von Typparametern handhabt. Dies setzt bei statisch

typisierten Sprachen die Verwendung von gebundener Generizität voraus, da Typinformationen über generische Parameter vorhanden sein müssen.

- Die Vor- und Nachteile einer heterogenen Übersetzung sind:
  - \+ Eine sehr effiziente Optimierung ist möglich, da zum Zeitpunkt der Übersetzung der konkrete Typ von generischen Parametern bekannt ist.
  - \+ Es ist nicht notwendig, gebundene Generizität zu verwenden; ob eine konkrete Ersetzung eines Typparameters möglich ist, muss nicht vorhergesagt werden. Daher erlauben heterogene Übersetzungen eine wesentlich größere Ausdrucksstärke als homogene.
  - – Unter Umständen müssen sehr viele Spezialisierungen generischer Konstrukte erzeugt und in die Zielsprache übersetzt werden; dies führt zu langen Kompilationszeiten und die Größe des erzeugten Codes ist beträchtlich.
  - – Bei der Verwendung von generischen Libraries muss normalerweise der Quellcode vorliegen.
- Die Vor- und Nachteile einer homogenen Übersetzung sind:
  - \+ Die Kompilationszeit und die Größe des entstehenden Codes sind unabhängig von der Anzahl der Ersetzungen.
  - \+ Im Fall von Java erlaubt diese Vorgehensweise gemeinsam mit Type-Erasure (siehe Abschnitt 3.6) die Rückwärtskompatibilität zu bestehenden Libraries.
  - – Die Integration primitiver Typen ist schwierig.
  - – Das Laufzeitverhalten ist schlechter als bei einer heterogenen Übersetzung, da Typumwandlungen eingefügt werden müssen und spezielle Optimierungen nicht leicht möglich sind.

Sowohl Java als auch Eiffel verwenden eine homogene Übersetzung generischer Konstrukte; Eiffel führt zusätzlich spezielle Optimierungen für primitive Typen durch, da diese in der Sprache nicht gesondert behandelt werden (siehe Abschnitt 3.5). C++ ist eine Sprache, die eine rein heterogene Übersetzung verwendet; C# (siehe Abschnitt 3.10.2) verfolgt eine gemischten Ansatz: Typparameter, die Klassen sind, werden homogen übersetzt, primitive Typen heterogen.

3.5 Behandlung primitiver Typen

In Eiffel gibt es keine Unterscheidung zwischen primitiven Typen und Klassen; auch Integer oder Gleitkommazahlen sind Klassen, die Untertypen von `ANY` sind; die entsprechenden Klassen sind `BOOLEAN`, `CHARACTER`, `INTEGER`, `REAL` und `DOUBLE`. Diese sind allerdings expandiert (*expanded*), das bedeutet, es können keine Referenzen auf diese Klassen erstellt werden. Das erlaubt es dem Compiler, diese bei Bedarf in die

entsprechenden Maschinentypen umzuwandeln, damit Performanceeinbußen vermieden werden; für den Programmierer ist dieser Vorgang allerdings transparent. Daher können alle Typen ohne Einschränkung als generische Parameter verwendet werden.

Java hingegen unterscheidet zwischen primitiven Typen (`bool`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`) und Referenztypen (Klassen). Zwischen primitiven Typen untereinander und zwischen primitiven Typen und Klassen können keinerlei Untertypbeziehungen bestehen, insbesondere sind primitive Typen nicht von `Object` abgeleitet. Für jeden primitiven Typ existiert eine zugehörige Wrapperklasse, die diesen als Klasse repräsentiert und über die entsprechenden Methoden zur Konvertierung verfügt.

In Eiffel existieren zusätzlich zu jeder Klasse, die einen primitiven Typ repräsentiert, eine zugehörige nicht expandierte Klasse, auf die Referenzen zulässig sind; die nicht expandierten Klassen sind durch ein `_REF` Suffix gekennzeichnet, z.B. `INTEGER_REF` oder `REAL_REF`. Dabei kann ein Konvertierung nur von der expandierten Klasse zur Referenzklasse erfolgen, nicht aber umgekehrt.

Bei der Erweiterung von Java um Generizität ist diskutiert worden, ob primitive Typen als generische Parameter zugelassen werden sollten [4]. Dies wurde letztlich abgelehnt; die Gründe hierfür waren:

- Java unterscheidet streng zwischen primitiven Typen und Klassen. Es gibt keinen Kontext, in dem primitive Typen und Klassen austauschbar verwendbar sind; die einzige Ausnahme sind Arrays, die allerdings in Java eine Sonderstellung einnehmen (siehe auch Abschnitt 2.5.5).
- Die Zulassung primitiver Typen als generische Parameter würde eine homogene Übersetzung unmöglich machen.
- Homogene Übersetzung ist die einzige Möglichkeit um Rückwärtskompatibilität sicherzustellen.
- Für jeden primitiven Typ existiert eine entsprechende Wrapper-Klasse. Die Umwandlung in diese erlaubt die Verwendung als generische Parameter, auch wenn dies mit einem gewissen Programmieraufwand und Performanceeinbußen verbunden ist.

Allerdings ist die Verwendung primitiver Typen in Collections (z.B. `Stack<int>`) sehr häufig, daher wurde in Java 1.5 eine automatische Konvertierung zwischen primitiven Typen und den entsprechenden Wrapperklassen eingeführt. Dies wird als *Autoboxing* bezeichnet (unter *Boxing* versteht man die Umwandlung eines primitiven Typs in die entsprechende Wrapperklassen, unter *Unboxing* den umgekehrten Vorgang).

In Java 1.5 werden primitive Typen in einem Kontext, in dem Referenztypen (Klassen) erwartet werden, automatisch in die entsprechenden Wrapperklassen konvertiert; auch der umgekehrte Vorgang wurde automatisiert: Wrapperklassen werden in einem Kontext, in dem die zugehörigen primitiven Typen erwartet werden, ebenfalls automatisch konvertiert. Java unterstützt also genau genommen *Autoboxing* und *Autoun-*

boxing (C# etwa unterstützt nur Autoboxing, nicht aber Autounboxing). Dadurch können generische Collections sehr einfach mit primitiven Typen verwendet werden, ohne dass zusätzliche Arbeit für den Programmierer entsteht:

```
public class Autobox {
  Stack<int> intStack;

  public static void main(String[] args) {
    intStack = new Stack<int>();
    // autoboxing
    intStack.push(12);
    // autounboxing
    int i = intStack.pop();
  }
}
```

Im obigen Beispiel werden implizit mehrere Typumwandlungen durchgeführt. Zuerst wird `Stack<int>` in `Stack<Integer>` umgewandelt, da als generische Parameter nur Klassen zulässig sind. In der Anweisung `intStack.push(12)` wird das Integerliteral *12* in die eine Wrapperklasse vom Typ `Integer` umgewandelt und dann auf den Stack gelegt (Autoboxing). In der Anweisung `int i = intStack.pop()` wird ein Element vom Typ `Integer` vom Stack geholt; in der folgenden Zuweisung zur Variable i wird dieses in den primitiven Typen `int` umgewandelt (Autounboxing). Diese Konvertierungen führen allerdings auch zu Performanceeinbußen.

Ein interessantes Problem in diesem Zusammenhang ist die *Identität* von Wrapperklassen, d.h. sind die Wrapperklassen von identischen Werten primitiver Typen auch die gleiche Instanz dieser Wrapperklassen?

```
public void test() {
 Integer int1 = 4224;
 Integer int2 = 4224;
 if(int1 == int2) {
   //...
 }
}
```

In diesem Beispiel würde der Vergleich `if(int1 == int2)` fehlschlagen, da durch das Autoboxing zwei verschiedene Instanzen von `Integer` erzeugt würden. Der korrekte Vergleich wäre `if(int1.equals(int2))`, wobei die *equals* Methode zum Test der Gleichheit verwendet wird.

Es wäre auch denkbar, die Identität der Wrapperklassen zu gewährleisten. Folgende Probleme ergeben sich dabei:

- Wenn Wrapperklassen mit einem öffentlichen Konstruktor erzeugt werden, kann die Identität nicht garantiert werden; es müssten statische Factory-Methoden verwendet werden, die einen Cache über die bereits erzeugten Werte halten.
- Da Java seit der Version 1.0 öffentliche Konstruktoren für Wrapperklassen verwendet, ist eine nachträgliche, komplette Umstellung auf Factory-Methoden

kaum vorstellbar.

- Manche primitiven Datentypen, insbesondere `long` sowie `float` und `double` verfügen über einen sehr großen Wertebereich, weshalb der zugehörige Cache extrem groß werden könnte.

In Java 1.5 wurden sämtliche Wrapperklassen um statische Factory-Methoden erweitert, die beim Autoboxing verwendet werden. Die öffentlichen Konstruktoren bleiben aber weiterhin erhalten, um Rückwärtskompatibilität zu gewährleisten. Die Identität von Wrapperklassen wird garantiert, wenn beim Boxing der Wert des primitiven Typs `true`, `false`, ein `byte`, ein ASCII-Zeichen oder ein Integer zwischen -127 und 128 ist.

Da es in Eiffel nicht möglich ist, auf Klassen, die primitive Typen repräsentieren, Referenzen zu erstellen, stellt sich das Problem der Identität nicht. Die zugehörigen Referenzklassen (z.B. `INTEGER_REF`) hingegen garantieren keine Identität; diese werden allerdings nur selten verwendet.

3.6 Type Erasure in Java

Rückwärtskompatibilität zu früheren Versionen ist in Java unverzichtbar [4]. Das Ziel der Java-Implementation von Generizität ist die vollständige Ersetzung von Teilen der API durch neue, generische Versionen („Retrofitting“). Daher war die Einführung von Generizität in der Version 1.5 auch mit erheblichen Schwierigkeiten verbunden, da sichergestellt werden musste, dass vorhandener Code und Libraries mit von Java 1.5 erzeugtem Bytecode für die *Java Virtual Machine (JVM)* weiterhin ausgeführt werden können. Umgekehrt müssen die zahlreichen Bibliotheksklassen, die auf eine generische Variante umgestellt wurden, weiterhin mit altem Code kompatibel sein.

Aufgrund dieser Anforderungen wird in Java eine Technik namens *Type-Erasure* verwendet. Dabei werden generische Sprachkonstrukte allein vom Compiler in einem Art Präprozessordurchlauf in nicht-generischen Code umgewandelt; für die JVM ist dieser Vorgang völlig transparent. Da die JVM selbst keine generischen Typen kennt, werden im Zuge dieses Compilerdurchlaufs parametrisierte Typen in *Raw-Types* umgewandelt. Als *Raw-Type* bezeichnet man in diesem Zusammenhang den entsprechenden nicht generischen „Basistyp“.

Der Vorgang kann ungefähr so zusammengefasst werden:

- Wenn eine Klasse keine Typparameter verwendet, wird durch Erasure die Klassendefinition nicht geändert.
- Parametrisierte Typen verlieren ihre Typparameter; diese werden durch den stärksten Schrankentyp ersetzt, den der Compiler verwendet kann. Ein Typparameter `<T>` wird etwa durch `Object` ersetzt, während `<T extends Fahrzeug>` durch `Fahrzeug` ersetzt wird.
- Zuletzt werden, wo sie nötig sind, Casts eingefügt.

Dieser Vorgang ist auch für den Programmierer völlig transparent; dadurch ergeben sich allerdings auch einige unvorhergesehene Konsequenzen:

```
public class A<T extends Fahrzeug> {
  // ...
}

public class A<T extends LKW> {
  // ...
}
```

Da im obigen Beispiel nach dem Erasure die beiden Klassen den gleichen Namen, *A*, haben, ist dies unzulässig. Nach dem Entfernen und Ersetzen der Typparameter kann der Compiler zwischen den beiden Versionen der Klasse *A* nicht mehr unterschieden. In Eiffel ist dies allerdings auch nicht möglich, dort wurde diese Entscheidung aber bewusst getroffen. Grundsätzlich wäre es möglich, in diesem Fall immer die speziellste Variante einer parametrisierten Klasse zu verwenden; dies ist aber für den Programmierer undurchsichtig und kann bei der Verwendung mehrerer Typparameter zu Mehrdeutigkeiten führen.

Ein ähnliches Problem ergibt sich auch im Zusammenhang mit überladenen Methoden:

```
public class A<T extends Fahrzeug, U extends Fahrzeug> {
  public void drucken(T fahrzeug) {
    // ...
  }
  public void drucken(U fahrzeug) {
    // ...
  }
}
}
```

Hier würden durch die Substitution der Typparameter zwei Varianten der Methode *drucken* erzeugt, die die gleiche Signatur haben. Auch dieser Fall ist in Eiffel ebenfalls nicht gestattet, da Eiffel das Überladen von Methoden überhaupt nicht erlaubt.

Bei der Vermischung von generischen und nicht generischen Varianten einer Klasse kann es sogar zu Typfehlern zur Laufzeit kommen:

```
class A {
  public Lkw laufzeitFehler() {
    List<Lkw> listLkw = new LinkedList<Lkw>();
    List listAlias = listLkw;
    listAlias.add(new String("abc")); // compiler warning
    return listLkw.iterator().next();
  }
}
```

Hier wird ein Alias einer nicht generischen Liste (deren Elemente vom Typ `Object` sind), auf eine parametrisierte Liste, die Einträge vom Typ `LKW` hat, erzeugt. Nach

dem Type-Erasure ist der Typ der beiden Listen nicht mehr zu unterschieden; dadurch kann in die Liste ein `String` eingefügt und danach versucht werden, `LKW` zu extrahieren, was zu einem Laufzeitfehler führt.

Der Compiler gibt bei jedem möglichen unsicheren Vorgang ein *unchecked* Warning aus. Programme, die ohne diese Warnings kompiliert werden können, sind laut Sun [5] immer typsicher. Ähnlich wie bei Catcalls in Eiffel ist aber anzumerken, dass solche Warnings auch in Fällen ausgegeben werden, in denen es nicht zu einem Laufzeitfehler kommt; da Warnings häufig ignoriert werden, stellen sie kein geeignetes Mittel dar, statische Typsicherheit zu garantieren.

Als weitere Konsequenz ergibt sich, dass die Verwendung des `instanceof` Operators oder der *getClass()* Methode von `Object` bei parametrisierten Klassen äußert problematisch ist, da etwa `Vector<Vector<LKW>>>` und `Vector<LKW>` nach dem Erasure den gleichen Typ haben, obwohl sie nicht austauschbar verwendet werden können.

Es ist auch nicht möglich, Typparameter direkt zu instantiieren. Da Java eine homogene Übersetzung verwendet, kann nach dem Type-Erasure nicht mehr festgestellt werden, ob für jede mögliche Ersetzung für den Typparameter T ein entsprechender Konstruktor verfügbar ist:

```
class Singleton<T> {
  private T instance;

  public T getInstance() {
    if (instance == null)
     instance = new T();

     return instance;
  }
}
```

Die obige Implementierung des *Singleton* Design-Patterns ist in Java nicht zulässig.

In Eiffel gibt es eine entsprechende Problematik nicht, da kein Type-Erasure durchgeführt wird und der Compiler somit über die notwendige Typinformation verfügt.

3.7 Anonyme Typparameter und Wildcards in Java

Java bietet die Möglichkeit, auch anonyme Typparameter zu verwenden [5]; diese werden als *Wildcards* bezeichnet. Wird ein Typparameter einer Klasse oder Methode im Rumpf nicht verwendet, dann kann anstelle einer Typvariable ein *Wildcard* verwendet werden. Dieser wird durch `?` angegeben. Dies hat zuerst einmal den Vorteil, dass nicht Typvariablen deklariert werden müssen, die später nicht referenziert werden. Auch bei Wildcards ist es wie bei normalen Typparametern möglich, einen Schrankentyp mittels `extends` oder sogar eine untere Schranke durch `super` anzugeben.

Das folgende Beispiel zeigt die Verwendung von Wildcards:

```
// ohne wildcards
void printFahrzeuge1(Collection <T extends Fahrzeug> fahrzeuge) {
  for (Fahrzeug nextFahrzeug: fahrzeuge) {
    System.out.println(nextFahrzeug);
  }
}

// mit wildcards
public void printFahrzeuge2(Collection <? extends Fahrzeug> fahrzeuge) {
  for (Fahrzeug nextFahrzeug: fahrzeuge) {
      System.out.println(nextFahrzeug);
  }
}
```

In der *printFahrzeuge1()* Methode wird der Typparameter T zwar deklariert, aber nie verwendet. In diesem Fall kann stattdessen, wie in *printFahrzeuge2()* dargestellt, eine *Wildcard* verwendet werden, was im obigen Fall zunächst einmal einfach eine syntaktische Vereinfachung darstellt. Allerdings erhöht eine konsequente Verwendung von Wildcards wie oben beschrieben die Übersichtlichkeit des Codes, da direkt ersichtlich ist, ob ein Typparameter später noch einmal verwendet wird, oder lediglich ein allgemeiner Platzhalter ist.

Eine Besonderheit von Wildcards ist es, dass ein Schrankentyp mittels `super` angegeben werden kann, was bedeutet, dass an dieser Stelle nur Obertypen der angegebenen Schranke erlaubt sind. Dies ist bei normalen Typparametern nicht möglich.

```
private Collection<? super Fahrzeug> liste;
```

`Collection<Object>` würde dieser Definition genügen, `Collection<LKW>` jedoch nicht, da `LKW` kein Obertyp von `Fahrzeug` ist.

Das folgende Beispiel veranschaulicht, in welchen Situationen mittels `super` gebundene Wildcards nützlich sein können:

```
public class Util {
  public static <T> void listeKopieren
  (List<T> ziel, List<T> quelle) {

    for (int i = 0; i < quelle.size(); i++)
      ziel.set(i, quelle.get(i));
  }
}
```

Die statische Methode *listeKopieren()* kopiert eine generische Liste in eine andere. Dabei wird jedoch vorausgesetzt, dass die Typparameter von `quelle` und `ziel` identisch sind.

```
List<Object> ziel = new ArrayList<Object>();
List<String> quelle = new ArrayList<String>();
```

```
Util.listeKopieren(ziel, quelle);  // Fehler!
```

Der obige Aufruf von *listeKopieren()* würde vom Compiler nicht erlaubt werden, obwohl der Kopiervorgang prinzipiell möglich ist. Durch die Verwendung gebundener Wildcards kann die Methode so umgestaltet werden, dass sie weniger restriktiv aber dennoch typsicher ist:

```
public class Util {
  public static <T> void listeKopieren
  (List<? super T> ziel, List<? extends T> quelle) {

    for (int i = 0; i < quelle.size(); i++)
      ziel.set(i, quelle.get(i));
  }
}
```

Diese Implementierung von *listeKopieren()* setzt lediglich voraus, dass es einen Typ `T` gibt, der ein Untertyp des Elementtyps der Liste `ziel` und ein Obertyp des Elementtyps der Liste `quelle` ist. Dadurch wird der vorher von Compiler zurückgewiesene Aufruf möglich, während unsichere Vorgänge vermieden werden:

```
List<String> ziel = new ArrayList<String>();
List<Integer> quelle = new ArrayList<Integer>();
Util.listeKopieren(ziel, quelle);  // Fehler!
```

Dieser Aufruf von *listeKopieren()* ist nicht möglich, da es keinen Typ `T` gibt, der Untertyp von `String` und gleichzeitig Obertyp von `Integer` ist.

Der Hauptvorteil von Wildcards liegt allerdings darin, dass sie einen typsicheren Lösungsansatz in Situationen bieten, in denen Kovarianz bei generischen Typen erwünscht ist. In Java ist etwa (im Gegensatz zu Eiffel) `List<LKW>` kein Untertyp von `Collection<Fahrzeug>`. Es gilt folgende Regel:

- Ein generischer Typ `U` ist genau dann ein Untertyp des generischen Typs `T`, wenn die Typparameter identisch sind und der Raw-Type von U ein Untertyp des Raw-Types von `T` ist.

Das Problem wird im folgenden Beispiel verdeutlicht:

```
class A {
  public static void main(String[] args) {
    ArrayList<LKW> fuhrpark = new ArrayList<LKW>;
    // <init>
    drucken(fuhrpark);
    // illegaler cast
    drucken((Collection<Fahrzeug>)fuhrpark);
    // drucken
  }

  private drucken(Collection<Fahrzeug>) {
    // drucken
```

```
    }
  }
```

Die obige Vorgehensweise führt nicht zum gewünschten Resultat; die Methode *drucken* kann nicht aufgerufen werden, da `ArrayList<LKW>` kein Untertyp von `Collection<Fahrzeug>` ist. Auch ein expliziter Cast ist nicht erlaubt. Das Problem lässt sich durch die Verwendung von Wildcards umgehen:

```
class A {
public static void main(String[] args) {
  ArrayList<LKW> fuhrpark = new ArrayList<LKW>;
  // <init>
  drucken(fuhrpark);
}

private drucken(Collection<? extends Fahrzeug>) {
  // drucken
}
}
```

Diese Version funktioniert wie erwartet. Durch die Verwendung von Wildcards im Parameter von *drucken* wird je nach Aufruf ein entsprechender Typ generiert; im vorherigen Beispiel ist das `Collection<LKW>`, der entsprechend der oben angegebenen Regel ein Obertyp von `ArrayList<LKW>` ist.

Dieser Mechanismus erlaubt auch typsichere kovariante Zuweisungen generischer Typen:

```
ArrayList<LKW> fuhrpark = new ArrayList<LKW>;
Collection<? extends Fahrzeug> fahrzeuge = fuhrpark;
```

Diese Zuweisung ist legal und typsicher, da durch die Verwendung von Wildcards ein neuer, passender Typ erzeugt und anstelle der Wildcard verwendet wird.

Die folgende Zuweisung hingegen wäre nicht möglich:

```
ArrayList<LKW> fuhrpark = new ArrayList<LKW>;
Collection<Fahrzeug> fahrzeuge = fuhrpark;
```

In Eiffel gibt es keine Möglichkeit, anonyme Typparameter zu verwenden. In Java werden Wildcards aber hauptsächlich in Situationen verwendet, in denen kovariante Zuweisungen oder Untertypbeziehungen zwischen generischen Klassen von Vorteil wären; da in Eiffel auch generische Klassen konsequent kovariant sind, besteht kaum Bedarf für ein solches Sprachkonstrukt.

3.8 Generizität und Kovarianz in Eiffel

In Eiffel ist eine parametrisierte Klasse ein Untertyp einer anderen, wenn jeder Parameter der einen Klasse ein Untertyp des entsprechenden Parameters der anderen ist. (Eine genaue Definition ist im Abschnitt 2.5.1 angegeben). Z.B. ist `LIST[STRING]` ein Untertyp von `LIST[ANY]`, da `STRING` ein Untertyp von `ANY` ist.

Dabei kann es zu Catcalls kommen:

```
class
  GEN_COVARIANCE

feature cat_call is
  local
    container_vektor_2D: ARRAY[VEKTOR_2D]
    container: ARRAY[ANY]
    vektor_2D: VEKTOR_2D -- polymorphe Referenz
    str: STRING
    skalar: DOUBLE
  do
     create container_vektor_2D.make(0, 9)
     create vektor_2D.make_2D(2.0, 3.0)

    container := container_vektor_2d
    -- ok
    container.put(vektor_2D, 0)
    -- catcall
    container.put("String", 1)
    skalar := container_vektor_2D.item(1).skalar_produkt(vektor_2D)
  end
end
```

Im obigen Beispiel wird ein Catcall ausgelöst. Die Zuweisung `container := container_vektor_2d` ist zulässig, da `Vektor_2D` ein Untertyp von `ANY` ist. Anschließend wird in das Array ein String eingefügt, und danach versucht, die Methode *skalar_produkt()* auszuführen, was fehlschlägt, da ein String nicht über diese Methode verfügt. Diese Problematik ist im wesentlich identisch zu kovarianten Arrays in Java (siehe Abschnitt 2.5.5).

3.9 Generizität und Interfaces

Eine einfache und häufige Anwendung für Generizität sind Interfaces bzw. abstrakte Klassen, die binäre Methoden beinhalten. Die im Abschnitt 2.3 gezeigten Vorgehensweisen haben entscheidende Nachteile; generische Interfaces bieten hier für viele Fälle eine praktikable Lösung.

3.9.1 Eiffel

VERGLEICHBAR_GEN[T] ist eine abstrakte (deferred) Klasse in Eiffel, die dem *VERGLEICHBAR* Interface aus Abschnitt 2.3.2 entspricht. Anstelle der Definition der Eingangsparameter der Methoden mit dem Typ `like Current` und der späteren kovarianten Neudefinition dieser Parameter wird hier allerdings der Typparameter T verwendet:

```
deferred class
    VERGLEICHBAR_GEN[T]

feature
    gleich(a: T): BOOLEAN is deferred end

feature
    groesser_als(a: T): BOOLEAN is deferred end

end
```

3.9.2 Anwendungsbeispiel in Eiffel

Ein prioritätsgesteuertes Mailssystem, das Nachrichten senden und empfangen kann, und dabei höherpriore Nachrichten vor solche mit einer niedrigeren Priorität stellt, soll als Beispiel für die Anwendung generischer Interfaces und Klassen dienen.

Die Klasse *PRIORITY_MESSAGE* abstrahiert dabei die Nachrichten, die in diesem System versendet werden können.

```
class
    PRIORITY_MESSAGE inherit VERGLEICHBAR_GEN[PRIORITY_MESSAGE]

  -- die deferred features von VERGLEICHBAR_GEN
    feature
    groesser_als(a: PRIORITY_MESSAGE):BOOLEAN is
        do
            result := (priority > a.priority)
        end

    feature
    gleich(a: PRIORITY_MESSAGE):BOOLEAN is
        do
            result := (priority = a.priority)
        end

    feature {NONE}
    priority: INTEGER
end
```

Da die Nachrichten untereinander nach Priorität geordnet werden sollen, wird von der Nachrichtenklasse das Interface *VERGLEICHBAR_GEN* implementiert. Als Typpa-

rameter wird dabei *PRIORITY_MESSAGE* verwendet, also die Nachrichtenklasse selbst, da diese untereinander verglichen werden sollen (die Vergleichsmethoden sind *binär*). Die Priorität der Nachricht wird durch die Variable `priority: INTEGER` modelliert.

Diese Vorgehensweise erhöht die Typsicherheit, da in der Klasse, die das Interface implementiert, auf kovariante Neudefinitionen verzichtet werden kann. Aus diesem Grund ist dieses Idiom in Eiffel weit verbreitet. Auch andere Programmiersprachen verwenden diesen Ansatz, um in gewissem Umfang Kovarianz zu ermöglichen.

Die Klasse *PRIORITY_MAIL* selbst ist für das Senden, Empfangen und Verwalten der Messages zuständig, wobei über die konkrete Implementierung der Nachrichten nichts bekannt sein muss.

```
class
    PRIORITY_MAIL[T -> VERGLEICHBAR_GEN[T]]

feature
    send(msg: T) is
        local
          insert_pos: INTEGER
        do
        -- finde einfuegeposition
        from
            insert_pos := 0
        until
            insert_pos = idx OR
                storage.item(insert_pos).groesser_als(msg) OR
                storage.item(insert_pos).gleich(msg)
        loop
            insert_pos := insert_pos + 1
        end

        --  message einfuegen

    end

feature{NONE} -- Implementation
  storage: ARRAY[T]
```

PRIORITY_MAIL verwendet den gebundenen Typparameter *T*, der anstelle einer konkreten Nachrichtenklasse verwendet wird. Die Nachrichten selbst werden im Array `storage: ARRAY[T]` gehalten. Da die Methode *send* beim Einfügen der Nachrichten in dieses Array die Prioritäten der Nachrichten berücksichtigen muss, ist der Typparameter *T* durch den Typ `VERGLEICHBAR_GEN[T]` beschränkt, d.h. jede Klasse, die, wie durch das Interface *VERGLEICHBAR_GEN* definiert, mit sich selbst vergleichbar ist, kann als Nachricht verwendet werden.

Eine konkrete Ausprägung des Mailssystems wird durch die Variable *mail* deklariert:

```
  mail: PRIORITY_MAIL[PRIORITY_MESSAGE]
```

Als Nachricht wird dabei die oben beschriebene Klasse *PRIORITY_MESSAGE* verwendet. Dieser Ansatz ist äußerst flexibel, die Abhängigkeiten der Klassen, die Bestandteil des Nachrichtensystems sind, werden auf ein Minimum beschränkt und diese sind auch klar definiert und für den Programmierer übersichtlich dargestellt. Die Funktionalität des Nachrichtensystems kann problemlos für alle Arten von Messages verwendet werden:

```
voice_mail: PRIORITY_MAIL[VOICE_MESSAGE]
```

Die einzige Einschränkung ist hierbei, dass *VOICE_MESSAGE* das Interface *VERGLEICHBAR_GEN* implementieren muss.

Diese Art der Verwendung von generischen Klassen hat einen gewissen Überschneidungsbereich mit der Abstraktion durch Vererbung. In vielen Fällen ist Generizität zur Modellierung von Aggregationsbeziehungen (siehe Abschnitt 2.1) vorteilhaft.

3.9.3 Java

VergleichbarGen ist die generische Variante des Interfaces *Vergleichbar* aus dem Abschnitt 2.3.1. Hier wird der Typparameter T eingeführt, um Casts in Klassen, die das Interface implementieren, zu vermeiden.

```
public interface VergleichbarGen<T> {
  public boolean groesserAls(T a);
  public boolean gleich(T a);
}
```

Auf den ersten Blick scheint dies äquivalent zu der Vorgehensweise in Eiffel zu sein. Allerdings gibt es durch die Verwendung von Type-Erasure zur Implementierung von Generizität in Java signifikante Einschränkungen, die nicht intuitiv klar sind. Auf diese Probleme wird im Abschnitt 3.9.5 detailliert eingegangen.

3.9.4 Anwendungsbeispiel in Java

Das prioritätsgesteuerte Mailsystem soll auch in Java implementiert werden. Die Nachricht wird durch die Klasse *PriorityMessage* dargestellt.

```
public class PriorityMessage
  implements VergleichbarGen<PriorityMessage> {

  private int priority;

  public void trace() {
      System.out.println(body + " priority: " + priority);
  }

  // interface VergleichbarGen
  public boolean groesserAls(PriorityMessage a) {
```

```
        return (priority > a.priority);
    }

    public boolean gleich(PriorityMessage a) {
        return (priority == a.priority);
    }
  }
```

Auch hier wird die Ordnung der Messages untereinander durch die Implementierung der Methoden des generisches Interfaces *VergleichbarGen* bestimmt.

Die Klasse *PriorityMail* und die Methode *send* werden in Java wie folgt implementiert:

```
public class PriorityMail<T extends VergleichbarGen<T>> {
  public void send(T msg) {
    int insertPos;

    // finde einfuegeposition
    for(insertPos = 0;
      insertPos < idx && storage.elementAt(insertPos).groesserAls(msg);
      ++insertPos) {
        //
    }

    // message einfuegen
  }
}
```

Das komplette Mailsystem wird unter Verwendung der Klasse, die den Typparameter ersetzt, erzeugt:

```
  PriorityMail<PriorityMessage> mail =
    new PriorityMail<PriorityMessage>();
```

3.9.5 Problem im Anwendungsbeispiel in Java

Auf den ersten Blick sind die Implementierungen des Mailsystems in Java und Eiffel konzeptuell sehr ähnlich. Leider ist die Verwendung von generischen Interfaces zur Modellierung binärer Methoden in Java nur eingeschränkt möglich.

Soll etwa das System um eine Klasse *VoiceMessage*, die ein Untertyp von *PriorityMail* ist, erweitert werden, könnten folgende Klassen definiert werden:

```
  public class PriorityMessage
    implements VergleichbarGen<PriorityMessage> {
    // ...
  }

  public class VoiceMessage extends PriorityMail
```

```
  implements VergleichbarGen<VoiceMessage> {
  // ..
}
```

Nach dem Type-Erasure durch den Compiler würden die Klassendefinitionen folgendermaßen aussehen:

```
public class PriorityMessage implements VergleichbarGen {
  // ...
}

public class VoiceMessage extends PriorityMail
  implements VergleichbarGen {
  // ..
}
```

Dadurch würde die Klasse *VoiceMail* die gleiche Ausprägung des Interfaces *VergleichbarGen* zweimal implementieren, was der Compiler nicht erlaubt. Das bedeutet, dass nur eine Oberklasse in einer Vererbungshierarchie ein solches Interface implementieren kann. Für alle abgeleiteten Klassen wird der Typ des Eingangsparameters für eine Methode wie *gleich* durch die Oberklasse bestimmt, was unter Umständen Typumwandlung zur Laufzeit, wie in Abschnitt 2.3.1 beschrieben, bedingt. Die Verwendung von Interfaces kann in solchen Fällen trotzdem sinnvoll sein , da durch Oberklasse der Eingangsparameter zumindest stärker beschränkt wird als durch `Object` im Falle von nicht generischen Interfaces.

3.10 Generizität in anderen Programmiersprachen und alternative Ansätze

3.10.1 C++

C++ unterstützt Generizität durch *Templates* [30]. Typparameter sind nicht gebunden, d.h. es gibt keine Einschränkungen, welche Features Typen, die Typparameter ersetzen, verwenden können. Für jede konkrete Ersetzung wird vom Compiler ein spezieller Code erzeugt (heterogene Übersetzung), und danach wird eine Typüberprüfung vorgenommen. Dies hat die in Abschnitt 3.4 beschriebenen Nachteile zur Folge; auf der anderen Seite ist dieser Ansatz wesentlich flexibler und ausdrucksstärker als die Verwendung von gebundener Generizität und erlaubt auch die Erzeugung von effizienterem Code, da zur Laufzeit keine Typumwandlungen durchgeführt werden müssen. Mit der *Standard Template Library (STL)* steht eine umfangreiche und mächtige Templatebibliothek zur Verfügung.

Templates müssen im Quellcode vorliegen, eine separate Kompilierung von Templates

ohne eine konkrete Ersetzung der Typparameter ist nicht möglich. Eine wesentliche Schwäche dieses Ansatzes ist, dass für den Programmierer nicht direkt ersichtlich ist, welche Features des Typs, der den Typparameter ersetzt, letztlich verwendet werden; die kann auch nicht formal spezifiziert werden, wie es etwa bei gebundener Generizität möglich ist. Änderungen in der Implementierung von Templates können dazu führen, dass existierender Code nicht mehr funktioniert.

3.10.2 C#

C# ist eine Programmiersprache, die von *Microsoft* speziell für das *.NET-Framework* entworfen wurde. Das .NET-Framework beinhaltet eine virtuelle Maschine (*CLR-Common Language Runtime*) die einen speziellen, maschinenunabhängigen Zwischencode (*MSIL-Microsoft Intermediate Language*) ausführt; dieser entspricht dem Bytecode der JVM. Zusätzlich existiert eine umfangreiche Klassenbibliothek. Die Unterschiede zwischen den generischen Varianten von Java und C# werden in [1] beschrieben.

Ähnlich wie in Java wurde Generizität in C# nicht von Anfang an unterstützt sondern wird nachträglich zur Sprache hinzugefügt. C# unterstützt, wie Java, Einfachvererbung und mehrfache Untertypbeziehungen mittels Interfaces. Die generische Version von C# verwendet gebundene Generizität; als Schrankentyp kann dabei eine Klasse und/oder mehrere Interfaces angegeben werden. Auch primitive Typen sind als generische Parameter erlaubt, da sich diese von einen gemeinsamen Obertypen `Object` ableiten. Generische Parameter sind in Klassen, Interfaces, Methoden und *Delegates* erlaubt.

Im Gegensatz zu Java wird durch die Einführung von Generizität die MSIL (und damit auch die CLR) verändert und um generische Typen erweitert. Auch sind neue generische Klassen in den .NET-Libraries nicht rückwärtskompatibel zu ihren nicht generischen Varianten. Das alte .NET Framework bleibt neben dem erweiterten bestehen. Die Einführung generischer Typen in die MSIL erlaubt es, Instantiierungen generischer Konkstrukte bei Bedarf zur Laufzeit durchzuführen. Der entscheidende Vorteil dieser Vorgehensweise ist, dass Probleme im Zusammenhang mit Type-Erasure werden vermieden werden, außerdem müssen generische Libraries nicht wie in C++ im Quellcode vorliegen. Der Nachteil ist, dass eine Rückwärtskompatibilität unmöglich ist.

C# verwendet eine Mischung zwischen homogener und heterogener Übersetzung. Wird ein primitiver Typ als generischer Parameter verwendet, so wird eine heterogene Übersetzung verwendet, ansonsten homogene.

3.10.3 Virtual Types

Virtual Types [33] basieren auf den *Virtual Patterns* der Programmiersprache *BETA* [20]. Das Grundprinzip dabei ist, dass Klassendefinitionen um virtuelle Typdekla-

rationen erweitert werden, die einen neuen Typnamen als Alias für einen existierenden Typ einführen. Mit der Anweisung `typedef` *`Name`* `as` *`Type`* wird ein Typalias mit dem Namen *Name* für den Typ *Type* erzeugt, ähnlich wie in C/C++. Diese Aliase können in der Klasse anstelle konkreter Typen verwendet werden, so wie Typparameter in Eiffel oder Java.

In Unterklassen können Aliase verändert werden, und zwar kann *Type* durch einen Untertyp ersetzt werden, was einer kovarianten Neudefinition des Aliases entspricht. *Name* kann hingegen nicht verändert werden. Zusätzlich wird ein spezieller virtueller Typ, *This*, der die umschließende Klasse repräsentiert, eingeführt.

Dieser Ansatz eignet sich gut für rekursive Klassenstrukturen, wie sie bei *Design Patterns* häufig auftreten. Auch binäre Methoden können effizient und übersichtlich behandelt werden.

In [34] werden *Structural Virtual Types* vorgestellt, ein Ansatz, der gebundene Generizität und Virtual Types miteinander verbindet.

3.11 Resultate

Grundsätzlich scheinen die Ansätze, die bei der Integration von Generizität in Java und Eiffel verwendet werden, sehr ähnlich. Beide Sprachen verwenden gebundene Generizität, dass heißt, dass Typen, die zu einer angegebenen Schranke in einer Untertypbeziehung stehen, generische Parameter ersetzen können. Dieser Ansatz erlaubt es beiden Sprachen, eine homogene Übersetzung generischer Konstrukte durchzuführen.

Ein Unterschied der beiden Sprachen tritt bei der Verwendung von Generizität besonders hervor: die Integration primitiver Typen. In Eiffel sind primitive Typen (expandierte) Klassen, die von *ANY* abgeleitet sind. Es gibt keinerlei Einschränkung, welche Typen als generische Parameter verwendet werden können.

In Java existiert eine strenge Unterscheidung zwischen Klassen (Referenztypen) und primitiven Typen. Für jeden primitiven Typ existiert eine entsprechende Wrapperklasse, die in Kontexten verwendet werden kann, in denen nur Referenztypen zulässig sind. Als generische Parameter können ausschließlich Klassen verwendet werden. Da dies, insbesondere bei der Verwendung generischer Collections, eine starke Einschränkung bedeutet, wurde eine automatische Umwandlung primitiver Typen in ihre enstprechende Wrapperklasse und umgekehrt eingeführt (Autoboxing). Dies ist zwar mit einer gewissen Einbuße an Laufzeiteffizienz verbunden, und nicht elegant und konsequent wie in Eiffel, stellt aber sicher eine brauchbare Möglichkeit dar. Da die Unterscheidung zwischen primitiven Typen und Klassen tief in Java verwurzelt ist, ist dieser Ansatz auch konzeptuell verständlich.

Während Eiffel Generizität von Anfang an als wesentlichen Bestandteil in die Sprache integriert hat, ist in Java Generizität erst mit der Version 1.5. eingeführt worden. Um ein vollständige Rückwärtskompatibilität zu der enormen Menge an existierendem

Code und Libraries zu garantieren, mussten dabei einige Kompromisse geschlossen werden. Da Klassendefinitionen in der JVM nicht geändert wurden, werden generische Parameter in einem Compilerdurchlauf entfernt (Type-Erasure). Dies führt in einigen Fällen zu für den Programmierer schwer nachvollziehbaren Einschränkungen.

Eiffel ist auch bei generischen Klassen kovariant. Das hat zwar intuitive Untertypbeziehungen zur Folge, allerdings kann dadurch die statische Typsicherheit nicht garantiert werden und Catcalls sind möglich. Java hat bei Untertypbeziehungen zwischen generischen Klassen restriktivere Regeln, die unintuitiv sein können; etwa ist `Stack<String>` kein Untertyp von `Collection<Object>`. Damit werden Catcalls wie in Eiffel verhindert.

Allerdings sind Typfehler zur Laufzeit auch in Java möglich, da durch das Type-Erasure wichtige Informationen verloren gehen. Der Java-Compiler gibt für alle potentiell unsicheren Vorgänge ein Warning aus. Sun garantiert, dass alle Programme, die ohne diese Warnings kompilieren, typsicher sind; dies ist aber kein geeigneter Ansatz, um statische Typsicherheit zu gewährleisten.

In Java besteht auch die Möglichkeit zur Verwendung anonymer Typparameter (Wildcards). Diese haben einigen praktischen Nutzen; in vielen Fällen sind sie jedoch nötig, um Einschränkungen, die durch das Type-Erasure begründet sind, zu umgehen.

3.11.1 Übersicht

	Java	*Eiffel*
Art der Generizität	`gebunden`	`gebunden`
Übersetzungsschema	`homogen`	`homogen`
Integration in Sprache	`-`	`++`
Integration in Libraries	`+`	`++`
Typinformation generischer Parameter	`--` `(Type-Erasure)`	`++`
Behandlung primitiver Typen	`+`	`++`
Anonyme Typparameter	`Ja`	`Nein`
Typsicherheit	`+`	`-`
Ausdrucksstärke	`+`	`+`

Kapitel 4

Reflexion

4.1 Klassifizierung

Reflexionsmechanismen können nach verschiedenen Kriterien eingeteilt werden [22]:

Die einfachste Ausprägung ist *Implementations-Reflexion*. Dabei existieren Metaobjekte, die eigens dafür existieren, gewisse Aspekte der Systemimplementierung zu inspizieren und auch zu verändern, wie etwa die Strategie eines Garbage-Collectors oder die Prioritäten beim Scheduling von Threads. Dabei kann der Zugriff auf implementierungsspezifische Details als der Metabereich angesehen werden.

Üblicherweise ist mit dem Begriff der Reflexion aber *Computational* Reflexion gemeint. In diesem Ansatz stellt die Programmiersprache Metaklassen zur Verfügung, die Mechanismen vergegenständlichen, die normalerweise implizit in der Sprache vorhanden sind, wie etwa Methodenaufrufe.

Computational Reflexion wird noch genauer in folgende Kategorien unterteilt [8]:

Metaklassenmodell: Im Metaklassenmodell ist für jede Klasse eine Metaklasse vorhanden, die deren Struktur und Verhalten beschreibt. Verschiedene Instanzen der gleichen Klasse haben das gleiche Metaobjekt.

Metaobjektmodell: Das Metaobjektmodell ist eine Variante des Metaklassenmodells; allerdings existieren Metainformationen nicht nur pro Klasse, sondern jede Instanz einer Klasse verfügt über ein eigenes Metaobjekt.

Metakommunikationsmodell: Im Metakommunikationsmodell werden nicht Klassen oder Objekte, sondern Nachrichten an Objekte reifiziert.

Bei der Art der Implementierung von Reflexion in einer Programmiersprache kann noch zwischen *expliziter* und *impliziter* Reflexion unterschieden werden [7]. Explizite Reflexion bedeutet, dass gezielt, etwa durch die Verwendung der entsprechenden API, auf Metainformationen zugegriffen wird. Bei der impliziten Reflexion wird diese

immer an speziellen, vorherbestimmten Stellen der Programmausführung, etwa bei Methodenaufrufen, durchgeführt.

4.2 Features

4.2.1 Eiffel

Reflexion wird in der Eiffel-Terminologie auch als *Introspektion* bezeichnet. Ursprünglich war Reflexion in Eiffel nicht vorgesehen; in einem Proposal von Bertrand Meyer [25] wird eine Erweiterung der Sprache um *Agenten* (siehe Abschnitt 4.3) und Reflexion vorgeschlagen, wobei diese beiden Konzepte auf eine etwas undurchsichtige Art miteinander verknüpft sind.

Während Agenten in der aktuellen Version von ISE Eiffel und SmallEiffel bereits gut unterstützt werden, ist Reflexion nur sehr eingeschränkt möglich. Das dynamische Laden von Klassen ist ebensowenig möglich wie dynamische Methodenaufrufe durch Angabe des Methodennamens, allerdings bieten Agenten dafür eine Alternative.

Derzeit ist nicht absehbar, ob und wann eine vollständigere Unterstützung für Reflexion in Eiffel verfügbar sein wird.

4.2.2 Java

Java unterstützt Reflexion seit der Version 1.1. Seitdem wurden an der Reflexions-API zahlreiche Änderungen durchgeführt; die meisten Neuerungen sind durch die Einführung von Generizität in Java 1.5 bedingt. *Java Beans*, ein Komponentenmodell für Java sowie Java *Remote Method Invocation (RMI)* machen ausgiebig von Reflexion gebrauch.

Java bietet im Vergleich zu anderen statisch typisierten, objektorientierten Sprachen eine sehr gute und umfangreiche Unterstützung von Reflexion. So lassen sich dynamische, indirekte Methodenaufrufe durch Angabe des Methodennamens erreichen, auch das dynamische Laden von Klassen zur Laufzeit ist möglich; dies hat allerdings eine erhebliche Verschlechterung der Performance zur Folge.

4.3 Agenten in Eiffel

Agenten [25] sind ein Sprachkonstrukt in Eiffel, das es erlaubt, Methodenaufrufe als *First-Class Objekte* zu verwenden, d.h. Methodenaufrufe selbst haben einen formalen Typ, können in Variablen gespeichert und an andere Methoden übergeben werden. Dies ist sehr ähnlich zu Funktionspointern in C/C++ oder *Closures* in LISP.

Eine Besonderheit von Agenten ist, dass Eingangsparameter beim Erzeugen des Methoden-Objekts entweder an einen konkreten Wert gebunden oder offen gelassen werden können. Gebundene Argumente sind nicht veränderbar, für jedes offene Argument muss, wenn die Methode aufgerufen wird, ein Wert angegeben werden.

Solche Methoden-Objekte werden durch das Schlüsselwort `agent`, gefolgt von der darzustellenden Methode erzeugt. Wird ein Parameter offen gelassen, so wird anstelle eines Wertes `?` angegeben. Existiert etwa eine Methode *f1()*, die über 3 `INTEGER` Parameter verfügt, so wird durch den folgenden Ausdruck ein Agent erzeugt, der über keine offenen Parameter verfügt:

```
agent f1(1, 2, 3)
```

Der Ausdruck

```
agent f1(1, ?, ?)
```

instantiiert einen Agenten, bei dem die letzten beiden Eingangsparameter offen gelassen werden. Es ist möglich, alle Parameter offen zu lassen.

Agenten werden durch die folgenden drei Klassen dargestellt:

- `ROUTINE [BASE_TYPE, OPEN_ARGS -> TUPLE]` ist die abstrakte Oberklasse für die konkrete Ausprägung eines Agenten.
- `PROCEDURE [BASE_TYPE, OPEN_ARGS -> TUPLE]` stellt eine Methode ohne Rückgabewert dar.
- `FUNCTION [BASE_TYPE, OPEN_ARGS -> TUPLE, RESULT_TYPE]` repräsentiert eine Methode, die einen Wert zurückliefert.

Die generischen Parameter dieser Klassen geben dabei den Typ der Klasse, zu der die Methode gehört (`BASE_TYPE`) und die offenen Parameter (`OPEN_ARGS`) an. `TUPLE` ist eine Containerklasse, die die Typen der offen gelassenen Argumente ihrerseits als generische Parameter enthält. Liefert die Methode einen Wert zurück, so wird dessen Typ durch `RESULT_TYPE` dargestellt. `BASE_TYPE` spielt dabei im Zusammenhang mit Reflexion eine wichtige Rolle (siehe 4.3.1).

Der Typ des Agenten

```
agent f1(1, 2, 3)
```

wäre daher

```
f: FUNCTION [ANY, TUPLE, REAL]
```

falls die Methode *f1()* eine Gleitkommazahl als Rückgabewert hat.

Der Typ von

```
agent f1(?, ?, 3)
```

wäre

```
f: FUNCTION [ANY, TUPLE[INTEGER, INTEGER], REAL]
```

Die wichtigsten Features der von `ROUTINE` abgeleiteten Klassen sind:

- *call()* ruft eine Methode ohne Rückgabewert auf
- *item()* ruft eine Methode auf und liefert den Rückgabewert zurück
- *precondition()* und *postcondition()* überprüfen, ob die Vor- bzw. Nachbedingung der Methoden erfüllt ist
- Das Array *open_args* enthält alle offenen Parameter

Agenten können in vielen Situationen nützlich sein:

```
class AGENT

feature num_integrator(f: FUNCTION [ANY, TUPLE[REAL], REAL];
  lower: REAL; upper: REAL; step: REAL): REAL
  is
      local x: REAL;
  do
    from
      x := lower
    until
      x >= upper
    loop
      -- check precondition
      if
        f.precondition([x])
      then
        Result := Result + step * f.item([x])
      end

      x := x + step
    end
  end
end
```

Die Methode *num_integrator()* kann ganz allgemein die numerische Integration einer mathematischen Funktion in einem (geschlossenen) Intervall durchführen, wobei die Grenzen des Intervalls (*lower* und *upper*) sowie die Granularität der Berechnung (*step*) angegeben werden. Zusätzlich muss als erster Parameter ein Agent übergeben werden, der die (mathematische) Funktion enthält, die integriert werden soll. Dabei wird erwartet, dass diese Funktion ein offenes Argument und einen Rückgabewert vom Typ `REAL` hat. Der offene Parameter wird dabei bei der Berechnung in einer Schleife durch einen konkreten Wert ersetzt, die Rückgabewerte aufaddiert, und vor dem Aufruf wird noch überprüft, ob die Vorbedingung des Agenten erfüllt ist. Offene Parameter werden in eckigen Klammern an Werte gebunden.

Der Verwendung würde so aussehen:

```
class AGENT

feature f(x: REAL; a: REAL): REAL is
  require
    non_negative: x >= 0
  do
    -- a * sqrt(x)
    Result := sqrt(x) * a
  end

feature calc is
  local
    result: REAL
  do
     -- numerische integration
     result := num_integrator(agent f(?, 3.0), 0.0, 10.0, 0.01)
     print(result)
  end

end
```

In der Methode *calc()* wird die numerische Integration durchgeführt. Dabei wird das Integral

$$\int_l^u f(ax^2)\,dx.$$

im Intervall von [0, 10] berechnet, und a wird an die Konstante 3 gebunden. Anstatt von *f()* könnte jede beliebige andere Funktion verwendet werden, unter der Voraussetzung, dass der Agent nach dem Binden von Argumenten einen offenen Parameter und einen Rückgabewert von Typ `REAL` hat.

Die Verwendung von Agenten im vorherigen Beispiel ist sehr flexibel, da über die verwendete Funktion nur die offenen Argumente und der Typ der Rückgabewerts bekannt sein muss. Allerdings ist anzumerken, dass dieses Sprachkonstrukt den schon in Eiffel vorhandenen Abstraktionskonzepten entgegenläuft. Durch die Verwendung eines Interfaces, das die Methodensignatur der zu integrierenden Funktion definiert, ließe sich der gleiche Effekt erzielen. Diese Vorgehensweise hat auch den Vorteil, dass die Anforderungen an diese Methode für den Programmierer expliziter dargestellt sind.

4.3.1 Agenten und Reflexion in Eiffel

Agenten können im Proposal von Bertrand Meyer auch für Reflexion verwendet werden. Dabei spielt die Methode *generator()* von `ROUTINE` eine wichtige Rolle: diese liefert ein Metaobjekt der zum Agenten gehörige Klasse zurück. Dieses Objekt hat

den Typ `TYPE` und bildet die Ausgangsbasis für Reflexion in Eiffel. Folgende Features stehen zur Verfügung:

- *name* ist der Name der Klasse als String
- Das Array *generics* enthält alle generischen Parameter der Klasse
- Das Array *routines* stellt die Methoden der Klasse dar. Wird über diesen Weg auf `ROUTINE` zugegriffen, so sind alle Parameter offen.
- Das Array *attributes* erlaubt den Zugriff auf die Attribute (Felder) der Klasse.
- *invariant* gibt an, ob die Klasseninvarianten erfüllt sind.

Derzeit wird `TYPE` aber von keinem Eiffel-Compiler unterstützt. In ISE Eiffel liefert *generator* einen String zurück, der einfach den Namen der Klasse enthält.

Im Proposal von Bertrand Meyer ist die einzige Möglichkeit, ein Objekt vom Typ `TYPE` zu erhalten, indirekt über die Verwendung von Agenten. Dies ist nicht leicht nachzuvollziehen, da Agenten und Reflexion sehr wenig miteinander zu tun haben. Leider existieren außer diesem Proposal kaum weitere Informationen zu Reflexion in Eiffel, weshalb auch die Begründung für diese etwas eigenwillige Verknüpfung nicht in Erfahrung zu bringen ist.

Es würde meiner Meinung nach nichts dagegen sprechen, das Feature *generator* mittels `ANY` allen Klassen zur Verfügung zu stellen. Interessanterweise verfügt in ISE Eiffel 5.4 `ANY` über dieses Feature, es wird allerdings nur ein String mit dem Namen der Klasse zurückgeliefert. Immerhin wäre dadurch ein Ansatz zur Reflexion ähnlich wie in C++ (siehe 4.5.1) möglich.

Auch ohne konkrete Implementierung von `TYPE` ist derzeit eine limitierte Form der Reflexion für Agenten möglich:

```
class
  INTROSPECTOR

feature introspect_agent(r: ROUTINE[ANY, TUPLE]) is
  -- introspektion durchfuehren
  local
    i: INTEGER
    ops: TUPLE;
    o: ANY;
  do
    -- base type
    io.put_string("Base type: ")
    io.put_string(r.base_type.generator)

    -- target
    io.put_string("Target: ")
    io.put_string(r.target.generator)

    -- Argumente
```

```
    io.put_string("Number open args: ")
    io.put_integer(r.open_count)

    ops := r.operands
    from
      i := 0
    until
      i > (ops.count - 1)
    loop
      -- argument type
      io.put_string(ops.item(i + 1).generator)
      i := i + 1
    end
  end

feature test is
  do
    introspect_agent(agent introspect_agent(?))
  end

end
```

4.4 Reflexion in Java

Die Package `java.lang.reflect` enthält die Reflexions-API von Java. Die wichtigsten Klassen darin sind:

- `Class` ist der Ausgangspunkt für Reflexion in Java und repräsentiert einen Typ.
- `Field` stellt ein Feld einer Klasse dar. Eine Veränderung zur Laufzeit ist über Methoden dieser Klasse möglich.
- `Method` repräsentiert eine Methode. Ein Methodenaufruf ist möglich.
- `Constructor` stellt einen Konstruktor einer Klasse dar. Mit Hilfe dieser Klasse können Objekte indirekt erzeugt werden.
- `Modifier` gibt den Zugriffstyp einer Klasse, Methode, eines Konstruktors oder Felds an.
- `Array` stellt ein Array dar.

In Java 1.5 sind noch einige Klassen und Interfaces hinzugefügt worden, die generische Parameter repräsentieren. Diese werden in Abschnitt 4.4.6 detailliert beschrieben.

Der Ausgangspunkt für Reflexion in Java ist `Class`, das Metainformationen für jeden Typ enthält; eine Instanz dieser Klasse kann für jeden Typ, auch für primitive Typen und sogar für `void`, abgerufen werden.

Jede Klasse, die in die JVM geladen wird, hat verfügt über eine Instanz von `Class`, die das zugehörige `.class` File repräsentiert. Wird ein Objekt erzeugt, so wird von

der JVM auf diese Information zurückgegriffen.

Eine Instanz von `Class` kann auf verschiedene Arten erhalten werden:

- Die *getClass()* Methode von `Object` liefert den Laufzeittyp des jeweiligen Objekts.
- Über die statische Factory- Methode *forName()* von `Class` kann durch Angabe des Namens ein Metaklassenobjekt erzeugt werden (siehe Abschnitt 4.4.2).
- *Klassenliterale* (*Class Literals*) erlauben es, durch Anhängen von `.class` an einen Typnamen eine entsprechende Instanz von `Class` abzurufen. Dies ist auch bei primitiven Typen möglich.
- Wrapperklassen für primitive Typen verfügen über das statisches Feld *TYPE*, das das selbe Ergebnis wie das Klassenliteral des zugehörigen primitiven Typs liefert.

Mittels `Class` stehen dem Programmierer dann folgende Metadaten zur Verfügung:

- Den Namen der Klasse und das Package, in dem sie enthalten ist
- Die Oberklasse
- Die implementierten Interfaces
- Konstruktoren
- Methoden
- Felder
- Den Elementtyp von Arrays

Informationen über Methoden, Felder und Konstruktoren werden in der Praxis am häufigsten verwendet und im Folgenden detaillierter beschrieben.

4.4.1 Konstruktoren

Das folgende Beispiel zeigt, wie zur Laufzeit auf Informationen über die vorhanden Konstruktoren zugegriffen werden kann:

```
public class Reflection {
  public void reflectCtors(Class refClass) {

    Constructor ctors[] = refClass.getConstructors();
    Constructor ctor;

    for(int i = 0; i < ctors.length; ++i) {
      ctor = ctors[i];
      // name
      System.out.println("Constructor: " + ctor.getName());
      // deklarierende klasse
```

```
        Class decl = ctor.getDeclaringClass();
        System.out.println("Deklariert von: " + decl.getName());
        // modifiers
        int mod = ctor.getModifiers();
        System.out.println("Modifier: " + Modifier.toString(mod));
        // parameter
        Class params[] = ctor.getParameterTypes();
        for(int j = 0; j < params.length; ++j) {
          System.out.println("Parameter: " + params[j].getName());
        }
        // exceptions
        Class exceptions[] = ctor.getExceptionTypes();
        for(int j = 0; j < exceptions.length; ++j) {
          System.out.println("Exception: " + exceptions[j].getName());
        }
      }
    }
  }
```

Im obigen Beispiel wird der Methode *reflectCtors()* ein Parameter vom Typ `Class` übergeben, der die zu reflektierenden Klasse darstellt. Mit der Anweisung `Constructor ctors[] = refClass.getConstructors()` wird ein Array von `Constructor` erstellt, das die Konstruktoren der Klasse beinhaltet.

Es gibt dabei zwei Möglichkeiten, das Array zu erstellen:

- *getConstructors()* liefert alle öffentlichen Konstruktoren, über welche die Klasse verfügt
- *getDeclaredConstructors()* liefert *alle* Konstruktoren, auch private (*private*) oder geschützte (*protected*), die von der Klasse deklariert werden

Diese Unterscheidung ist auf den ersten Blick wenig sinnvoll, da der Zugriffstyp des Konstruktors sehr leicht gesondert ermittelt werden kann. Sie wurde jedoch als Analogie zu den entsprechenden Methoden von `Method` und `Field` eingeführt, wo diese Differenzierung sehr wichtig ist (siehe Abschnitt 4.4.3).

Danach wird im Beispiel in einer Schleife über die Konstruktoren iteriert und folgende Informationen ermittelt:

- *getName()* ermittelt den Namen des Konstruktors.
- *getDeclaringClass()* liefert die Klasse, die den Konstruktor deklariert. Da in Java Konstruktoren nicht vererbt werden können, ist diese immer die Klasse selbst, die als Parameter der Methode *reflectCtors()* übergeben wurde.
- *getModifiers()* ermittelt den Zugriffstyp (*private*, *public*, *protected*) des Konstruktors. Dieser wird durch Integerkonstanten, die in `Modifier` definiert sind, dargestellt.
- *getParameterTypes()* liefert ein Array, das die Typen der Parameter des Konstruktors in der Reihenfolge der Deklaration enthält.

- *getExceptionTypes()* gibt ein Array zurück, das alle vom Konstruktor deklarierten Exceptions enthält.

Die Klasse `Constructor` erlaubt auch die Instantiierung von Klassen über den Reflexionsmechanismus.

```
public class Reflection {
  public void bigIntCtor1() {
    // Konstruktor erzeugen
    Class[] types = new Class[] { String.class, int.class };
    Constructor ctor = BigInteger.class.getConstructor(types);
    // Argumente
    Object[] args = new Object[] { "999939", 256 };
    // BigInteger instantiieren
    BigInteger bigInt = ctor.newInstance(args);
  }
}
```

Im obigen Beispiel wird ein `BigInteger` indirekt über den Reflexionsmechanismus erzeugt. Dazu wird ein `Constructor` unter Angabe der Typen seiner Parameter (`String` und `int`) erzeugt. Dieser verfügt über die Methode *newInstance()*, die eine neue Instanz von `BigInteger` erzeugt; als Parameter wird ein Array der Werte der Eingangsparameter des Konstruktors übergeben.

4.4.2 Dynamisches Laden von Klassen

Es ist sogar möglich, eine Instanz einer Klasse nur unter Angabe ihres Namens zu erzeugen:

```
public class Reflection {
  public void bigIntCtor2() {
    // Konstruktor erzeugen
    Type bigIntType = Class.forName("java.math.BigInteger");
    Class[] types = new Class[] { String.class, int.class };
    Constructor ctor = bigIntType.getConstructor(types);
    // Argumente
    Object[] args = new Object[] { "123999939", 16 };
    // BigInteger instantiieren
    BigInteger bigInt = ctor.newInstance(args);
  }
}
```

Die Methode *bigIntCtor2()* unterscheidet sich von der vorherigen Version dadurch, dass das benötigte Objekt vom Typ `Class` durch die statische Factory-Methode *forName()* erzeugt wird. Dabei wird der Klassenname einfach als String angegeben. Es können auch Klassen verwendet werden, die zur Compilezeit nicht bekannt sind. Dadurch ist ein komplett dynamisches Laden von Klassen in die JVM möglich. Dies ist ein äußert flexibler und mächtiger Mechanismus, der normalerweise nur in interpretierten (Skript-) Sprachen zu finden ist; allerdings ist der Preis dafür eine äußerst

schlechte Performance. Außerdem können natürlich Typfehler zur Laufzeit auftreten. Diese Typfehler lösen Exceptions aus.

In Eiffel ist eine vergleichbare Vorgehensweise nicht möglich.

4.4.3 Methoden

Das folgende Beispiel zeigt, wie zur Laufzeit Informationen über Methoden einer Klasse abgerufen werden können:

```
public class Reflection {
  public void reflectMethods(Class refClass) {
    // Variante 1
    Method methoden[] = refClass.getMethods();
    // Variante 2
    Method methoden[] = refClass.getDeclaredMethods();

    Method methode;

    for(int i = 0; i < methoden.length; ++i) {
      methode = methoden[i];
      // name
      System.out.println("Methode: " + methode.getName());
      // deklarierende klasse
      Class decl = methode.getDeclaringClass();
      System.out.println("Deklariert von: " + decl.getName());
      // modifiers
      int mod = methode.getModifiers();
      System.out.println("Modifier: " + Modifier.toString(mod));
      // r\"{u}ckgabewert
      Class ret = methode.getReturnType();
      System.out.println("R\"{u}ckgabewert: " + ret.getName());
      // parameter
      Class params[] = methode.getParameterTypes();
      for(int j = 0; j < params.length; ++j) {
        System.out.println("Parameter: " + params[j].getName());
      }
      // exceptions
      Class exceptions[] = methode.getExceptionTypes();
      for(int j = 0; j < exceptions.length; ++j) {
        System.out.println("Exception: " + exceptions[j].getName());
      }
    }
  }
}
```

In *reflectMethods()* wird die Reflexion mit Hilfe der Klasse `Method` durchgeführt. Die Vorgehensweise ist dabei sehr ähnlich, wie in *reflectConstructors()*. Die wichtigsten Unterschiede sind:

- *getMethods()* liefert ein Array aller öffentlichen Methoden, über die die zu re-

flektierende Klasse verfügt. Darin sind auch alle Methoden enthalten, welche von Oberklassen geerbt werden.

- *getDeclaredMethods()* ermittelt alle Methoden, unabhängig vom Zugriffstyp, die in dieser Klasse deklariert werden; geererbte Methoden werden *nicht* berücksichtigt.
- Zusätzlich kann mit *getReturnType()* der Ergebnistyp der Methode abgefragt werden.

Methoden können auch reflexiv unter Angabe des Methodennamens aufgerufen werden. Falls beim Aufruf die Anzahl oder Typen der Parameter nicht mit der Methodendeklaration übereinstimmen, so wird eine Exception ausgelöst:

```
public class Reflection {
  public void reflectSelf() {

    Class classSelf = this.getClass();

    // methodenobjekt erzeugen
    Class pTypes[] = new Class[] {Class.class};
    Method methode = classSelf.getMethod("reflectMethods", pTypes);


    // eigentlicher aufruf
    Object pVals[] = new Object[] {this};
    methode.invoke(this, pVals);
  }
}
```

Hier wird Methode *reflectMethods* aus dem vorherigen Beispiel aufgerufen. Zuerst muss ein `Method`-Objekt unter Angabe des Methodennamens und einem Array mit den Typen der Eingangsparameter erzeugt werden. Dieses Array hat ein Element vom Typ `Class`, da dies der Typ des Eingangsparameters von *reflectMethods* ist. Danach wird ein korrespondierendes Array erstellt, das die konkreten Bindungen der Eingangsparameter an Objektreferenzen enthält. Da die Klasse `Reflection` selbst reflektiert werden soll, wird im Beispiel *this* verwendet. Danach kann mit *invoke()* vom vorher erzeugten `Method`-Objekt die gewünschte Methode aufgerufen werden.

Das obige Beispiel hat dabei den gleichen Effekt wie ein Aufruf von `this.reflectMethods()`. Auch in diesem Fall tritt bei der Verwendung von Reflexion eine erhebliche Verschlechterung der Performance auf.

4.4.4 Felder

Den reflexiven Zugriff auf Felder einer Klasse veranschaulicht das nächste Beispiel:

```
public class Reflection {
  public void reflectFields(Class refClass) {
    // Variante 1
```

```
    Field fields[] = refClass.getFields();
    // Variante 2
    Field fields[] = refClass.getDeclaredFields();
    Field field;

    for(int i = 0; i < fields.length; ++i) {
      field = fields[i];
      // name
      System.out.println("Field: " + field.getName());
      // deklarierende klasse
      Class decl = field.getDeclaringClass();
      System.out.println("Deklariert von: " + decl.getName());
      // modifiers
      int mod = field.getModifiers();
      System.out.println("Modifier: " + Modifier.toString(mod));
      // typ
      Class typ= field.getType();
      System.out.println("Typ: " + typ.getName());
    }
  }
}
```

Informationen über Felder werden in *reflectFields()* mit Hilfe eines Arrays von `Field` abgerufen. Je nachdem, ob dieses mit der Methode *getFields()* oder *getDeclaredFields()* von `Class` ermittelt wird, sind alle öffentlichen Felder inklusive der vom Obertyp geerbten Felder oder alle in genau dieser Klasse deklarierten Felder enthalten. `Fields` ist sehr ähnlich zu `Methods` und `Constructors`; die wichtigsten Methoden sind *getName()* und *getType()*, die den Namen bzw. Typ eines Feldes liefern.

Auf die Werte von Feldern kann zur Laufzeit zugegriffen werden:

```
public class Reflection {
  public void incIntFeld(String feldName, Object obj) {
    Class type = obj.getClass();
    Field feld = type.getDeclaredField(feldName);
    int value = feld.getInt(obj);
    field.setInt(obj, value + 1);
  }
}
```

Die Methode *incIntFeld()* inkrementiert den Wert eine (Integer-) Feldes; als Parameter wird der Name des Feldes sowie eine Refenerz auf das zugehörige Objekt angegeben. Die Klasse `Field` verfügt für Referenztypen und jeden primitiven Typen über entsprechende *get-* und *set-* Methoden (z.b. *setInt()* oder *getObject()*).

4.4.5 Arrays

Auch Arrays stellen zur Laufzeit Metainformationen zur Verfügung:

```
public class Reflection

  public Object resizeArray(Object array, int size) {
    Class type = array.getClass().getComponentType();
    Object newArray = Array.newInstance(type, size);
    System.arraycopy(array, 0, newArray, 0,
        Math.min(Array.getLength(array), size));
    return newArray;
  }
}
```

Im obigen Beispiel wird gezeigt, wie zur Laufzeit die Größe eines Arrays verändert werden kann; ohne die Verwendung von Reflexion wäre dies in Java nicht möglich. Die Methode *getComponentType()* liefert dabei den Typ der Elemente des Arrays, der anschließend in *newInstance()* von `Array` verwendet wird, um das neue Array mit der angegebenen Größe zu erstellen.

4.4.6 Reflexion und Generizität

Im Java 1.5 wurden `java.lang.reflect` um folgende Klassen und Interfaces erweitert, um die (limitierte) Möglichkeit zu bieten, Reflexion auf generischen Konstrukten durchzuführen:

- `GenericDeclaration` ist ein Interface, das von `Constructor`, `Method` und `Class` implementiert wird und anzeigt, dass diese Typparameter enthalten können. Mit der Methode *getTypeParamaters()* werden diese in einem Array zurückgeliefert.
- `TypeVariable` stellt Typparameter dar. Der Name des Typparameters sowie der Schrankentyp, an den dieser gebunden ist, können abgerufen werden.
- `ParameterizedType` repräsentiert eine parametrisierte Klasse oder Methode als ganzes, etwa `Stack<T>`. Die Typparameter(T) sowie der Raw-Type (`Stack`) können ermittelt werden.
- `GenericArrayType` stellt ein generisches Array (z.B. `T[]`)dar und erlaubt es, den Komponententyp des Arrays zu ermitteln (`T`).
- `WildcardType` repräsentiert einen Typ, der Wildcards verwendet. Auf die obere (z.B. `? extends Fahrzeug`) bzw. untere Schranke(z.B. `? super Fahrzeug`) kann zugegriffen werden.

Durch die Verwendung von Type-Erasure (siehe Abschnitt 3.6) ist die Reflexion parametrisierter Typen nur eingeschränkt möglich. Insbesondere ist es unmöglich, Informationen über die tatsächliche Bindung von Typparametern abzurufen, auch nicht, wenn eine eigene Methode eingeführt wird, die scheinbar genau diese Information liefert:

```
public class GenericClass<T, U extends Fahrzeug> {

   public Class<T> getType1() {
      return T.class;
   }

   public Class<U> getType2() {
      return U.class;
   }
}
```

Da beim Type-Erasure generische Parameter entfernt werden, aber das Klassenliteral `.class` statisch aufgelöst wird ist das obige Beispiel in Java nicht zulässig und würde sich nicht kompilieren lassen. Durch welchen konkreten Typ die generischen Parameter T und U ersetzt wurden, lässt sich nach dem Type-Erasure nicht mehr feststellen.

4.5 Reflexion in anderen Programmiersprachen

4.5.1 C++

In Standard C++ [30] wird eine limitierte Form der Reflexion unterstützt, die als *Run Time Type Information (RTTI)* bezeichnet wird. Dabei wird der Operator `typeid()` verwendet, dem als Argument ein Typname oder ein Ausdruck (in diese Fall wird der Typ des Ausdrucks zur Laufzeit verwendet) übergeben wird. Dieser liefert ein Objekt vom Typ `type_info` zurück, das einige wenige Typinformationen beinhaltet:

- Die Methode *name()*, die den Typnamen als String angibt. Bei primitiven Typen wird ein `NULL` Pointer zurückgeliefert.
- Die Operatoren `==` und `!=`, die die Gleichheit zweier Typen überprüfen können
- Die Methode *before()*, die eine Ordnungsrelation zwischen Typen herstellt, und benutzt werden kann, um Typen zu sortierten

Diese Informationen sind aber sehr spärlich und reichen normalerweise für sich alleine nicht aus, um konkrete Anforderungen an Reflexion umzusetzen. Laut Bjarne Stroustroup ist dies aber Absicht, da jede Applikation unterschiedliche Anforderungen stellt. RTTI stellt daher nur die Ausgangsbasis für auf die Applikation maßgeschneiderte Reflexionsmechansismen dar.

Dazu wird `type_info` als Schlüssel in einer *Map* verwendet, die eine Verknüpfung mit den gewünschten zusätzlichen Typinformationen herstellt. In der Praxis bedeutet dies, dass für jede Klasse, über die Metainformationen vorhanden sein sollen, eine spezielle Metaklasse implementiert werden muss. Diese kann dann genau die benötigten Informationen enthalten. Dieser Ansatz ist zwar flexibel und laufzeiteffizient, aber

sehr aufwändig und stellt nur eine Minimallösung dar. Er ist kaum als Reflexion auf Sprachebene zu bezeichnen.

In C++ selbst wird diese Typinformationen für den `dynamic_cast` Operator verwendet. Dieser wandelt zur Laufzeit einen Typ in einen anderen um, wenn dies möglich ist; ist die Konvertierung nicht möglich, so wird ein `NULL` Pointer zurückgeliefert, wodurch eine gewisse Fehlertoleranz erreicht wird. Dabei ist auch ein *upcast*, also eine Typumwandlung in einen Obertypen, möglich.

4.5.2 C#

C# [11] verfügt über eine gute Unterstützung für Reflexion. Die Reflexions-API ist im namespace `System.Reflection` enthalten und ist strukturell relativ ähnlich der entsprechenden Klassenstruktur in Java. Die wichtigsten Klassen darin sind:

- `Type` ist der Ausgangspunkt für Reflexion. Jeder Typ in C#, auch primitive Typen, werden durch diese Klasse dargestellt.
- `FieldInfo` stellt Felder einer *struct* oder eines *enum* dar und erlaubt auch die Manipulation dieser.
- `MemberInfo` repräsentiert die Felder einer Klasse; auch auf diese ist ein Zugriff zur Laufzeit möglich.
- `MethodInfo` repräsentiert Methoden einer Klasse. Diese können mittels *invoke()* aufgerufen werden.
- `ParameterInfo` stellt Informationen über Parameter einer Methode zur Verfügung.

Die Methode *CreateInstance()* der Klasse `Activator` erlaubt es, Klassen dynamisch unter Angabe ihres Namens zu erzeugen. Dies führt allerdings, wie in Java, zu Performanceeinbußen. C# erlaubt es, festzustellen, welche Typen in einem *Assembly* definiert werden; dabei stehen relativ mächtiger Such- und Filterfunktionen zur Verfügung.

C# erlaubt mittels der Klassen im Namespace `System.Reflection.Emit` sogar, neue Typen zur Laufzeit zu *definieren*, in die MSIL zu übersetzen und diese anschließend zu instantiieren und auszuführen. Dieser Mechanismus ist sehr mächtig und in erster Linie für die Anwendung in Compilern und Skriptmodulen gedacht. Eine dynamische Abänderung von Typen ist allerdings nicht möglich, da dies eine statische Typüberprüfung unmöglich machen würde. Auf dynamisch erzeugte Typen kann nur indirekt über die Reflexions-API zugegriffen werden; Variablen eines solchen Typs können nicht deklariert werden, da dieser zur Compilezeit ja unbekannt ist.

4.6 Resultate

Bei der Implementierung von Reflexion sind die Unterschiede zwischen Java und Eiffel sehr groß. Beide Sprachen verwenden das Metaklassenmodell; Java bietet für eine statisch typisierte, objektorientierte Sprache eine relativ umfangreiche Unterstützung. Reflexion ist ab Java 1.1 Bestandteil der Sprache und insbesondere Java Beans machen umfangreich davon Gebrauch. In Eiffel ist Reflexion derzeit nur sehr eingeschränkt möglich; in einem Proposal von Bertrand Meyer, das vom ISE Compiler und SmallEiffel teilweise umgesetzt wird, wird eine Spracherweiterung um Agenten und Reflexion (Introspektion) beschrieben.

In Java wird jeder Typ durch ein Metaobjekt vom Typ `Class` dargestellt, das recht umfangreiche Möglichkeiten der Reflexion bietet. Methoden, Konstruktoren, Felder und Arrays, die einen eigenen Typen darstellen, können als Objekte reifiziert werden. Dabei ist es möglich, mit Hilfe dieser Objekte Methoden- bzw. Konstruktorenaufrufe durchzuführen oder die Werte von Feldern zu verändern, was allerdings beträchtliche Performancenachteile mit sich bringt, trotzdem ist die Flexibilität groß. Es nicht möglich, zur Laufzeit Methoden oder Felder hinzuzufügen oder zu entfernen, oder deren Parameter bzw. Typ zu verändern, da dadurch eine statische Typüberprüfung unmöglich gemacht würde. Auch die Deklaration neuer Typen zur Laufzeit ist nicht möglich.

Eiffel führt ein neues Sprachkonstrukt, Agenten, ein, die es erlauben, Methoden als Objekte zu repräsentieren und über diesen Mechanismus Aufrufe zur Laufzeit durchzuführen. Dabei können als Besonderheit Parameter gebunden oder offen gelassen werden; nur offenen Parameter fließen in den Typ ein, der die Methode repräsentiert, wodurch dieser Ansatz recht flexibel ist. Offene Parameter, der Rückgabetyp sowie die Klasse, zu der die Methode gehört, können reflektiert werden. Jeder Agent ist an eine Klasse gebunden, die die repräsentierte Methode definiert. Diese Klasse wird durch eine Instanz von `TYPE` dargestellt und würde reflexiven Zugriff auf deren Features erlauben; auch generische Parameter und deren Bindungen wären reflektierbar. Kein derzeit verfügbarer Compiler implementiert jedoch `TYPE`; in ISE Eiffel wird stattdessen einfach ein String mit dem Typnamen verwendet. Der Zugang zu reflexiven Mechanismen ist laut dem Proposal nur über Agenten möglich; dies ist unintuitiv und die Gründe hierfür sind nicht leicht nachzuvollziehen.

In Java 1.5 sind zahlreiche Änderungen in der Reflexions-API vorgenommen worden, um auch generische Konstrukte zu erfassen. Neue Klassen und Interfaces zu Repräsentation von Typparameter, Wildcards und generischen Arrays wurden hinzugefügt. Da allerdings durch das Type-Erasure viele Typinformationen verloren gehen, ist die Reflexion generische Konstrukte nur eingeschränkt möglich. Viele Einschränkungen sind ohne ein genaues Verständnis dafür, wie Type-Erasure durch Java implementiert wird, nicht vorherzusehen und auch mit dem restlichen Reflexionsmechanismus inkonsistent. Solche Implementierungsdetails sollten eigentlich vor dem Programmierer versteckt werden, aber aus Gründen der Rückwärtskompabilität war Java gezwungen, diesen Weg zu gehen.

4.6.1 Übersicht

In dieser Übersicht wird für die Bewertung von Reflexion in Eiffel das Proposal von Bertrand Meyer angenommen.

	Java	*Eiffel*
Reflexionsmodell	`Metaklassen`	`Metaklassen`
Integration in Sprache	`+`	`--`
Reflexion generischer Konstrukte	`-`	`+`
Reflexive Methodenaufrufe	`Ja`	`Nein`
Dynamisches Laden von Klassen	`Ja`	`Nein`
Deklaration Typen zur Laufzeit	`Nein`	`Nein`
Ausdrucksstärke	`+`	`-`

Kapitel 5

Zusammenfassung

5.1 Zusammenfassung

Eiffel unterstützt echte Mehrfachvererbung; Java unterstützt nur Einfachvererbung, erlaubt aber mittels Interfaces mehrfache Untertypbeziehungen. In manchen Situationen bietet die Verwendung von Mehrfachvererbung Vorteile, da bestehender Code effizient weiterverwendet werden kann. Dem stehen jedoch eine höhere Komplexität und auch die häufige missbräuchliche Verwendung gegenüber. Eine gemischter Ansatz, in dem nur Einfachvererbung aber mehrfache Untertypbeziehungen mit Interfaces möglich ist, scheint ein guter und brauchbarer Kompromiß zu sein, da viele problematische Verwendungen vermieden werden und die Mächtigkeit nicht allzu stark eingeschränkt wird.

Ein bedeutsamer Unterschied zwischen Java und Eiffel ist die Art und Weise, in der Features von Klassen in Unterklassen abgeändert werden können. Eiffel ist dabei durchgängig kovariant, während Java in den allermeisten Fällen invariant ist. Kovarianz bietet den Vorteil einer großen Ausdrucksstärke, die in vielen Situationen eine direkte Modellierung von Problemstellungen ermöglicht. Der Nachteil an dieser Lösung ist, dass die statische Typsicherheit nicht garantiert werden kann und Catcalls zur Laufzeit auftreten können. Eine komplette Lösung des Problems ist in Eiffel nicht absehbar, auch wenn durch einige Techniken, wie etwa die konsequente Verwendung von expandierter Vererbung, das Risiko gemindert werden kann. Wenn, wie in Java, die Möglichkeit von kovarianten Neudefinitionen nicht besteht, müssen in vielen Fällen explizite Typumwandlungen zur Laufzeit durchgeführt werden, was ebenfalls das Risiko von Typfehlern zur Laufzeit mit sich bringt; diese führen jedoch nicht zu undefiniertem Programmverhalten sondern zu Exceptions, die auch abgefangen werden können.

Eiffel bietet mit DBC einen mächtigen und gut in die Sprache integrierten Mechanismus, um ein konsistentes Objektverhalten zu gewährleisten, der auch einfach auf ganze Vererbungshierarchien angewendet werden kann. Java bietet kein vergleichbares Konstrukt; Assertions sind stellen lediglich eine Minimallösung dar, die insbesondere

im Zusammenhang mit Vererbung unübersichtlich, redundant und fehleranfällig sind.

Generizität wurde in Eiffel und seine Libraries von Anfang an von sehr durchgängig integriert. Auch bei generischen Konstrukten ist Eiffel durchgängig kovariant, was wiederum die Typsicherheit gefährdet. Java unterstützt Generizität erst seit der Version 1.5; zahlreiche Libraries werden in dieser Version auf generische Varianten umgestellt. Um volle Rückwärtskompatibilität zu garantieren, mussten bei der Implementierung einige Kompromisse eingegangen werden. Da die JVM selbst keine generischen Typen kennt, verwendet Java eine Technik namens Type-Erasure, bei der Typinformationen über generische Konstrukte verloren gehen. Obwohl dies im Großen und Ganzen gut funktioniert, ergeben sich doch einige Einschränkungen und verstecke Fehlerquellen, die für den Programmierer nicht leicht zu erkennen sind.

Sowohl Java als auch Eiffel verwenden für die Umsetzung von Generizität eine homogene Übersetzung. In Eiffel sind auch primitive Typen von `ANY` abgeleitet, daher kann als generischen Parameter jeder beliebige Typ verwendet werden. In Eiffel sind nur Klassen (Referenztypen) als generische Parameter zulässig, daher wurde eine automatische Konvertierung von primitiven Typen in ihre entsprechende Wrapperklasse (Autoboxing) eingeführt. Dies ist für den Programmierer sehr praktisch und fast völlig transparent, kann aber zu leichten Performanceeinbußen führen.

Reflexion ist in den beiden Programmiersprachen sehr unterschiedlich implementiert. Java bietet ab der Version 1.1 eine für eine statisch typisierte, objektorientierte Programmiersprache sehr gute Unterstützung für Reflexion. Jeder Typ verfügt über eine zugeordnete Metaklasse, über die zahlreiche Informationen über den Typ abgerufen und teilweise auch manipuliert werden können. Reflexive Methodenaufrufe und auch das dynamische Laden von Klassen sind möglich.

Eiffel verbindet aus schwer nachvollziehbaren Gründen Reflexion mit einem anderen neuen Sprachkonstrukt, Agenten. Agenten erlauben in gewissem Rahmen die Reifikation von Methodenaufrufen, sind also durchaus als reflexiver Mechanismus anzusehen. Weitere Informationen über Klassen, wie etwa Felder oder deklarierte Konstruktoren, sind im Proposal von Bertrand Meyer über die Metaklasse `CLASS` verfügbar. Dies ist allerdings noch in keinem verfügbaren Compiler umgesetzt.

5.1.1 Tabellarische Gesamtübersicht

Hier werden noch einmal die tabellarischen Zusammenfassungen der verglichenen Sprachkonstrukte aufgelistet:

Vererbung

	Java	*Eiffel*
Mehrfache Untertypbeziehung	Ja	Ja
Mehrfache Vererbung	Nein	Ja
Abstrakte Klassen	Ja (interface)	Ja (deferred)
Vererbung ohne Untertypbeziehung	Nein	Ja (expanded)
Explizite Typumwandlung	Ja	Nein/durch Zuweisung
Kovarianz	Arrays, Ergebnistypen	Durchgängig
Typsicherheit	+	-
Mechanismus Objektverhalten	Assertions	DBC
Objektverhalten und Vererbung	-	++
Feature hiding	Nein	Ja
Ausdruckstärke	+	++

Generizität

	Java	*Eiffel*
Art der Generizität	gebunden	gebunden
Übersetzungsschema	homogen	homogen
Integration in Sprache	-	++
Integration in Libraries	+	++
Typinformation generischer Parameter	-- (Type-Erasure)	++
Behandlung primitiver Typen	+	++
Anonyme Typparameter	Ja	Nein
Typsicherheit	+	-
Ausdrucksstärke	+	+

Reflexion

	Java	*Eiffel*
Reflexionsmodell	Metaklassen	Metaklassen
Integration in Sprache	+	--
Reflexion generischer Konstrukte	-	+
Reflexive Methodenaufrufe	Ja	Nein
Dynamisches Laden von Klassen	Ja	Nein
Deklaration Typen zur Laufzeit	Nein	Nein
Ausdrucksstärke	+	-

Literaturverzeichnis

[1] Maria Lucia Barron-Estrada and Ryan Stansifer. A Comparisson of Generics in Java and C#. *Proceedings 41st Annual ACM SoutheastConference, Savannah, Georgia*, 2003.

[2] Deltlef Bertetzko. Parallelitaet und Vererbung beim „Programmieren mit Vertrag“ - Weiterentwicklung von JaWa. 1999.

[3] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems*, pages 303–311, Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 1990.

[4] Gilad Bracha. Adding Generics to the Java Programming Language: Participant Draft Specification. 2004.

[5] Gilad Bracha, Neal Gafter, Mards Torgersen, Christian Plesner Hansen, Erik Ernst, and Peter von der Ahe. Adding Wildcards to the Java Programming Language. 2004.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future safe for the Past: Adding Genericity to the Java Programming Language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[7] Gerald Brose. Reflection in Java, CORBA und JacORB.

[8] Walter Cazzola. Evaluation of Object-oriented Reflective Models. In *Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming ECOOP '98), Brussels, Belgium, on 20th-24th July 1998.

[9] W. R. Cook. A Proposal for making Eiffel type-safe. *The Computer Journal*, 32(4):305–311, 1989.

[10] O.J Dahl and K. Nygaard. Simula, an Algol-based Simulation Language. *Communications of the ACM 9, 9*, pages 671–678, 1966.

[11] Microsoft C# Devoloper Center. *http://msdn.microsoft.com/vcsharp/*.

[12] Andrew Duncan and Urs Hoelzle. Adding Contracts to Java with Handshake. *Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barabara*, 1998.

[13] Gilad Bracha et. al. Adding Generics to the Java Programming Language. *http://java.sun.com/developer/earlyAccess/addinggenerics/*, 2001.

[14] Adele Goldberg and David Robson. *SmallTalk-80 The Language and its Implementation*. 1983.

[15] Jr. Guy L. Steele. An Overview of COMMON LISP. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 98–107. ACM Press, 1982.

[16] Mark Howard, Eric Bezault, Bertrand Meyer, Dominique Colnet, Emmanuel Stapf, Karine Arnout, and Markus Keller. Type-safe Covariance: Competent Compilers can catch all Catcalls, 2003.

[17] Murat Karoarman, Urs Hoelzle, and John Bruno. jContractor: A reflective Java Library to support Design by Contract. *Technical Report, Department of Computer Science, University of California, Santa Barbara*, 1998.

[18] Martin Lackner. Extendig Java. *Diplomarbeit, TU Wien*, 2001.

[19] Wilf LaLonde and John Pugh. Subclassing != Subtyping != Is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, 1991.

[20] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. Virtual Classes: A powerful Mechanism in Object-oriented Programming. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1989.

[21] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-oriented Programming in the BETA Programming Language*. Addison-Wesly, 1993.

[22] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155. ACM Press, 1987.

[23] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Europe, 1992.

[24] Bertrand Meyer. Static typing. In *Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 20–29. ACM Press, 1995.

[25] Bertrand Meyer. Agents, Iteration and Introspection. *archive.eiffel.com/doc/manuals/language/agent/agent.pdf*, 2000.

[26] SUN Microsystems. *http://java.sun.com*.

[27] M. Odersky and P. Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, pages 146–159. ACM Press, New York (NY), USA, 1997.

[28] Heinz W. Schmidt and Stephen M. Omohundro. CLOS, Eiffel, and Sather: A comparison. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 181–213. MIT Press Cambridge, Massachusetts, London, England, 1993.

[29] Ghan Bir Singh. Single versus Multiple Inheritance in Object-oriented Programming. *SIGPLAN OOPS Mess.*, 6(1):30–39, 1995.

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 1991.

[31] Clemens Szyperski, Stephen Omohundro, and Stephan Murer. Engineering a Programming Language: The Type and Class System of Sather. In *Programming Languages and System Architectures*, pages 208–227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993.

[32] Antero Taivalsaari. On the Notion of Inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[33] Kresten Krab Thorup. Genericity in Java with Virtual Types. *Lecture Notes in Computer Science - ECOOP 97*, pages 444–470, 1997.

[34] Kresten Krab Thorup and Mads Torgersen. Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes. *Lecture Notes in Computer Science*, 1628:186, 1999.

[35] John Viega. Automated Delegation. *Masters Thesis, University of Virginia*, 1998.

[36] Peter Wegner. Concepts and Paradigms of Object-oriented Programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.

[37] William D. Young and Donald I. Good. Generics and Verification in ADA. *1980 SIGPlan Symposiun on the Ada Programing Language*, pages 123–127, 1980.