Diplomarbeit

Reengineering von Smalltalk nach Java

Markus Bauer

Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe

14. Oktober 1998

Betreuer:
Dipl.-Inform. Thomas Genßler
Dipl.-Inform. Benedikt Schulz

Verantwortlicher Betreuer:
Prof. Dr. rer. nat. Gerhard Goos

Erklärung

Ich erkläre hiermit, daß ich diese Arbeit eigenhändig, ohne unzulässige Hilfsmittel und unter Angabe aller Quellen angefertigt habe.

Karlsruhe, den 14. Oktober 1998

Markus Bauer

Inhaltsverzeichnis

1 Übersicht **1**

1.1 Aufgabenstellung 1

1.2 Gliederung der Arbeit 2

2 Einführung **3**

2.1 Kontext der Arbeit 3

2.2 Vorgehensweise 7

3 Grundkonzepte von Smalltalk und Java **10**

3.1 Einführung 10

3.2 Ausdrücke und Anweisungen 13

3.3 Blöcke und Kontrollstrukturen 17

3.4 Objekte, Klassen und Methoden 19

3.5 Vererbung 23

3.6 Typisierung und Polymorphie 24

3.7 Metaklassen und Introspektion 28

3.8 Bemerkungen 29

4 Einfache Reengineering-Ansätze **31**

4.1 Vorüberlegungen 31

4.2 Der Reflection-Ansatz 32

4.3 Der südafrikanische Ansatz 34

4.4 Der Interface-Ansatz 36

4.5 Zusammenfassung 39

5 Reengineering mit Hilfe von Typinferenz 40

5.1 Einführung 40
5.2 Statische Programmanalyse 43
5.3 Theoretische Grundlagen 44
5.4 Typinferenz als Datenflußproblem 48
5.5 Grundalgorithmus zur Typinferenz 52
5.6 Verbesserungen des Grundalgorithmus 55
5.7 Der Algorithmus von Agesen 59
5.8 Agesons Algorithmus und Smalltalk 63
5.9 Nutzung der Typinformationen 65

6 Bewertung 78

6.1 Automatisierbarkeit 78
6.2 Eigenschaften des entstehenden Java-Codes 80

7 Zusammenfassung und Ausblick 83

7.1 Zusammenfassung 83
7.2 Ausblick 84

A Quellcode zu `STObject` 86

Abbildungsverzeichnis

2.1 Kosten- und Nutzen verschiedener Varianten des Reengineerings 4

3.1 Verschiedene Smalltalk-Ausdrücke für Methodenaufrufe. 14
3.2 Eine Anweisungsfolge in Smalltalk und Java im Vergleich. 15
3.3 Umsetzung von Methodenaufrufen. 16
3.4 Verwendung von Blöcken in Smalltalk. 17
3.5 Eine Verzweigung in Smalltalk und in Java. 18
3.6 Iteration durch eine `Collection` in Smalltalk. 19
3.7 Umsetzung des Code-Beispiels aus Abb. 3.6 nach Java. 20
3.8 Klassengraph (Vererbungshierarchie) 23
3.9 Klassen zur Modellierung geometrischer Figuren 26
3.10 Smalltalk-Methode `add`, die polymorphen Code enthält. 26
3.11 Java-Variante der Methode `add`. 26
3.12 Die dualen Klassenhierarchien: Klassen und Metaklassen. 29

4.1 Aufruf einer Methode beim *Reflection*-Ansatz. 33
4.2 Stummel in `STObject` für `comma`. 34
4.3 Überblick über den südafrikanischen Ansatz. 35
4.4 Aufruf einer Methode beim südafrikanischen Ansatz. 35
4.5 Interface für `comma` und Fragmente aus `STString`. 37
4.6 Methodenaufruf durch *Casting* nach Interfaces. 37

5.1 Geometrische Figuren und Cowboys 42
5.2 Methode, die geometrische Figuren verwendet. 42
5.3 Prozedur zur Berechnung des Maximums zweier Zahlen 45
5.4 Kontrollflußgraph zur Prozedur aus Abb. 5.3 45
5.5 Worklist–Algorithmus zur Berechnung der MFP-Lösung 49

5.6 Datenflußgraph für einige Smalltalk-Anweisungen 50
5.7 Datenflußgraph mit konkreten Typen 51
5.8 Aufruf von `max` 53
5.9 Berechnung ungenauer Typmengen für `x` und `y` 55
5.10 Genauere Typmengen durch Expansion von Vererbung 56
5.11 Berechnung ungenauer Typmengen bei *1-Level-Expansion*. 58
5.12 Schablonen für den Aufruf `res := a max: b`. 61
5.13 Schablonen für den Aufruf `res := a max: b` (Fortsetzung). . . . 62
5.14 Blöcke in der Methode `maxof:and:`. 64
5.15 Java-Code mit Typ*annotationen*. 66
5.16 Java-Code mit Typ*deklarationen*. 66
5.17 Beispiel zur Fußnote auf Seite 68. 67
5.18 Gemeinsame Oberklasse zu $\{A, B, C\}$. 70
5.19 Methode g ist nicht in O definiert. 70
5.20 Modifikation einer bestehenden Klasse. 73
5.21 Einfügen einer neuen Klasse. 73
5.22 Einfügen eines neuen Interfaces. 73
5.23 Vererbung der Methode f mit Redefinitionen. 74
5.24 Ursprüngliche Vererbungshierarchie von `Test`. 77
5.25 Modifizierte Vererbungshierarchie von `Test`. 77

6.1 Reengineering mit Hilfe von Typinferenz. 79

Kapitel 1

Übersicht

1.1 Aufgabenstellung

Seit Ende der achtziger Jahre gehört die Programmiersprache *Smalltalk* zu den am häufigsten eingesetzten objekt-orientierten Programmiersprachen. Unternehmen aus den verschiedensten Branchen setzen Smalltalk zur Entwicklung von Softwaresystemen unterschiedlichster Art und Größe ein, so daß sich mittlerweile zahlreiche Smalltalk-Applikationen im täglichen Einsatz befinden [Sha95].

Mit der zunehmenden Verbreitung und Reife der Programmiersprache *Java* hat Smalltalk jedoch einen mächtigen, ebenfalls objekt-orientierten Konkurrenten bekommen. In bestimmten Bereichen, in denen früher oft Smalltalk zum Einsatz kam – beispielsweise im Bereich der *Inhouse-Software-Entwicklung* bei Banken und Versicherungen – wird heute bereits häufig mit Java entwickelt.

Unternehmen aus diesen Bereichen, die den Wechsel von Smalltalk zu Java als Entwicklungssprache für neue Anwendungen vollziehen, stehen nun vor der Frage was mit ihren in Smalltalk geschriebenen *Altanwendungen* geschehen soll. Da diese Altanwendungen gewartet und eventuell weiterentwickelt werden sollen, muß in solchen Unternehmen sowohl Smalltalk- als auch Java-Know-How gepflegt und ausgebaut werden. Eine solche Doppelstrategie ist aufwendig und kostenintensiv. Um dies zu vermeiden, ist es von Interesse, ob und wie sich bestehende Smalltalk-Anwendungen kostengünstig in Java-Anwendungen umsetzen lassen.

Aus dieser Fragestellung ergibt sich die Aufgabe der Diplomarbeit, aus einem Vergleich der Programmiersprachen Smalltalk und Java Verfahren zu erarbeiten, mit denen vorhandener Smalltalk-Code nach Java umgesetzt werden kann.

Die zu erarbeitenden Verfahren müssen gewissen Anforderungen genügen, damit sie von Nutzen sind. Einerseits muß mit ihrer Hilfe eine *praxisrelevante* Teilmenge von Smalltalk *korrekt* (unter Erhalt der ursprünglichen Semantik) nach Java umgesetzt werden können. Andrerseits müssen die Verfahren weitgehend *automatisierbar* sein, so daß der Aufwand für die Umsetzung überschaubar bleibt.

Auch der mit Hilfe solcher Verfahren erzeugte Java-Code sollte gewisse Eigenschaften aufweisen: Er sollte *korrekt*, *wartbar*, *erweiterbar* und *effizient* sein. Zur Erfüllung dieser Eigenschaften muß der entstehende Code in dem Sinn *typisch* für Java sein, daß er im Programmierstil und in den verwendeten Konstrukten von Hand geschriebenen Java-Code möglichst nahe kommt.

1.2 Gliederung der Arbeit

Die Arbeit gliedert sich folgendermaßen:

Kapitel 2 gibt eine Einführung in die Aufgabe, Smalltalk-Anwendungen nach Java umzusetzen und stellt sie in den allgemeinen Kontext des Software-Engineerings.

In Kapitel 3 vergleichen wir die grundlegenden Konzepte und Eigenschaften der beiden Programmiersprachen Smalltalk und Java und beschäftigen uns damit, wie diese bei einer Umsetzung von Smalltalk nach Java ineinander überführt werden können. Wir stellen dabei fest, daß die gravierendsten Unterschiede zwischen den beiden Sprachen in der Typisierung und in der Realisierung von Polymorphie liegen.

Daher widmen wir uns in den darauffolgenden Kapiteln 4 und 5 speziell der Umsetzung der Unterschiede in Typisierung und Polymorphie: In Kapitel 4 stellen wir einige Ansätze vor, die die entsprechenden Prinzipien aus Smalltalk direkt nach Java übertragen. In Kapitel 5 gehen wir einen aufwendigeren Weg: Wir extrahieren mit Hilfe von Typinferenz aus dem bestehenden Smalltalk-Code Typinformationen und nützen diese zur Erzeugung typisierten Java-Codes. Im ersten Teil des Kapitels beschäftigen wir uns ausführlich mit Typinferenzverfahren; im zweiten Teil geben wir an, wie wir die gewonnenen Typinformationen für das Reengineering des Smalltalk-Codes nützen können.

In Kapitel 6 evaluieren wir diese Vorgehensweise an einem geeigneten Beispiel. Insbesondere ermitteln wir dabei den Grad der Automatisierbarkeit des Verfahrens.

In Kapitel 7 fassen wir die Erkenntnisse der Arbeit zusammen und geben einen Überblick über weitere Fragestellungen im Zusammenhang mit unserer Reengineering-Aufgabe.

Kapitel 2

Einführung

2.1 Kontext der Arbeit

In diesem Abschnitt werden wir die Aufgabe, Smalltalk-Anwendungen in Java-Anwendungen zu überführen, genauer untersuchen. Zunächst werden wir unsere Aufgabe in den allgemeinen Themenbereich des Software-Reengineerings einordnen und danach begründen, warum diese spezielle Reengineering-Aufgabe von Interesse und praktischem Nutzen ist.

2.1.1 Software-Reengineering

Software-Reengineering umfaßt das Untersuchen und Verändern eines bestehenden Software-Systems, um es in eine verbesserte Neufassung zu bringen [CI90]. Diese Neufassung des Systems ermöglicht dann eine Fortentwicklung und Anpassung des Systems an neue Anforderungen mit geringerem Aufwand, sowie Verbesserungen hinsichtlich Betriebsverhalten (beispielsweise hinsichtlich Zuverlässigkeit oder Performance), Funktionalität und Nutzbarkeit [Til96]. In der Regel sind dazu Änderungen an den Strukturen der Anwendung erforderlich *(Restructuring)*. In objekt-orientierten System geschieht dies meist in Form von Änderungen am Objektmodell der Anwendung.

Die (systematische) Transformation des vorhandenen Systems in die Neufassung kann auf unterschiedliche Weise erfolgen. Eine Möglichkeit besteht darin, den Code der Altanwendung unmittelbar – möglichst werkzeugunterstützt – zu bearbeiten und ihn in die neue, verbesserte Form zu bringen *(Source Code Transformation)*. Wenn wir die unterschiedlichen *Abstraktionsebenen* der Anwendungsentwicklung – Anforderungen, Entwurf, Implementierung – als Maßstab heranziehen, so spielt sich ein solcher Reengineering-Prozeß auf niedrigem Abstraktionsniveau ab.

Im Gegensatz dazu ist auch eine Transformation des Systems möglich, indem mit Hilfe ausführlicher Programmanalysen Entwurfsinformationen aus dem Programmcode der Altanwendung extrahiert werden *(Reverse Engineering)*,

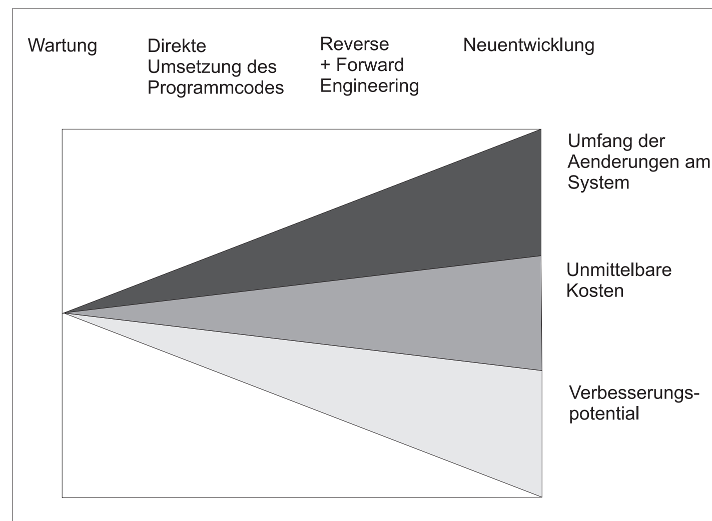

Abbildung 2.1: Kosten- und Nutzen verschiedener Varianten des Reengineerings

und auf Basis dieser Informationen eine Reimplementierung des Systems erstellt wird *(Forward Engineering)*. Durch diese Rückgewinnung von Entwurfsinformationen aus dem Programmcode entsteht eine Erhöhung der Abstraktionsebene, mit deren Hilfe erst umfangreichere Umstrukturierungen möglich werden. Dem möglicherweise größeren Nutzen solch umfangreicher Umstrukturierungen stehen allerdings höhere, unmittelbare Kosten gegenüber. In Abbildung 2.1 sind Umfang der Änderungen am System, Kosten und der gewonnene Spielraum für Verbesserungen (Nutzen) von Wartung, direkter Umsetzung von Programcode, Reverse- und Forward-Engineering, sowie kompletter Neuentwicklung schematisch dargestellt. Welche dieser Varianten zur Modifikation eines Altsystems angewandt werden soll, hängt stark von der Zielsetzung dieser Modifikation ab.

2.1.2 Wechsel der Programmiersprache

Unsere Aufgabe fällt in ein Spezialgebiet des Software-Reengineerings, das *Retargeting*. Unter Retargeting verstehen wir das Übertragen des Systems in eine andere Umgebung, beispielsweise in Form einer anderen Hardwareplattform, eines anderen Betriebssystems, einer anderen Programmiersprache oder eines anderen Datenhaltungssystems.

Die Übertragung von bestehenden Anwendungen in eine neue Programmiersprache ist meist sehr aufwendig. In der Regel sind solche Altanwendungen sehr umfangreich, so daß eine manuelle Umsetzung nicht wirtschaftlich durchzuführen ist.

Eine kostengünstige, automatische Umsetzung des bestehendes Codes ist jedoch oft nicht möglich, da es nicht immer gelingt, Algorithmen anzugeben, die für eine Abbildung der spezifischen Konzepte und Programmiertechniken der Ausgangssprache in solche der Zielsprache sorgen. Wenn die Konzepte von Ausgangs-

und Zielsprache sehr weit auseinanderliegen, verschiebt sich der Reengineering-Vorgang immer weiter von einer direkten Umsetzung auf Basis des Programmcodes zu einer Umsetzung mit Hilfe einer ausführlichen Programmanalyse und anschließender Reimplementierung. Probleme in diesem Zusammenhang entstehen besonders häufig dann, wenn die Ausgangssprache ein geringeres Abstraktionsniveau bietet als die Zielsprache, und die Abstraktionsmöglichkeiten der Zielsprache weitgehend genutzt werden sollen.

Eine wesentliche Erleichterung für eine automatisierte Umsetzung von Programmcode mittels eines Werkzeugs besteht darin, die Ausgangssprache auf eine praxisrelevante Teilmenge einzuschränken. Das Werkzeug wird dann so konstruiert, daß es diese praxisrelevante Teilmenge selbständig umsetzen kann. Die verbleibenden Konstrukte müssen vom Werkzeug erkannt und entsprechend gekennzeichnet werden und können dann von Hand bearbeitet werden. Entscheidend für den Erfolg einer solchen Verfahrensweise ist, daß nur ein geringer Anteil des Programmcodes manuell bearbeitet werden muß.

Wie wir oben angedeutet haben, ist Software-Reengineering aufwendig und teuer. Daher ist es besonders wichtig, daß wir uns des Sinns und Nutzens eines Reengineering-Projektes versichern, bevor wir es beginnen. Wir müssen von dem Ergebnis des Reengineerings profitieren können. Dies gilt natürlich auch für die Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen.

Wir werden daher in den folgenden Abschnitten kurz darstellen, welche Gründe es für den Umstieg von Smalltalk nach Java gibt, und welchen Nutzen die Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen bietet.

2.1.3 Warum Java statt Smalltalk?

Die Erfolgsgeschichte von Smalltalk ist eng mit folgenden Aspekten der Sprache und der zugehörigen Programmierumgebung verbunden [McC97] [Gol95]:

- Smalltalk ermöglicht eine effiziente Anwendungsentwicklung. Insbesondere unterstützt es *Rapid Prototyping* und fördert die Wiederverwendung von Code.
- Moderne Smalltalk-Systeme verfügen über ausgefeilte Klassenbibliotheken und Entwicklungswerkzeuge, unter anderem zur einfachen Konstruktion graphischer Benutzerschnittstellen und zur Anbindung an Datenbanken.
- Smalltalk-Systeme unterstützen verschiedenste Rechnerplattformen.
- Smalltalk ist eine übersichtliche, objekt-orientierte Sprache mit einfachen Kozepten und daher relativ schnell zu beherrschen.

Trotz dieser Vorteile verwenden bereits zahlreiche Unternehmen, die früher Smalltalk eingesetzt haben, Java zur Entwicklung ihrer Anwendungen. Die Gründe für diese Entwicklung sind vielschichtig.

Zum einen reklamiert Java für sich ähnliche Schlüsseleigenschaften wie Smalltalk, auch wenn existierende Java-Umgebungen noch nicht ganz die Reife vergleichbarer Smalltalk-Umgebungen erreicht haben. Wie wir in Kapitel 3 sehen werden, bestehen zum anderen zwischen den beiden Sprachen eine Reihe von Gemeinsamkeiten, die den Umstieg erleichtern.

Ein wesentlicher Erfolgsfaktor für Java ist zudem die Nähe zu den weit verbreiteten Sprachen C bzw. C++. Java bietet dem Software-Entwickler dabei eine vertraute Syntax und Programmierweise, ohne dabei auf Vereinfachungen im Sprachdesign und auf moderne Spracheigenschaften zu verzichten – als Beispiel sei hier nur der Verzicht auf Zeigerarithmetik und die automatische Speicherbereinigung angeführt.

Eine Reihe von weiteren Vorteilen ergibt sich aus der engen Verknüpfung von Java mit den Internet-Technologien, insbesondere dem *World Wide Web*. Zum einen erfährt Java dadurch eine weite beachtliche Popularität und Verbreitung, die dafür sorgt, daß Java eine Unterstützung der IT-Branche zuteil wird. Insbesondere unterstützen zahlreiche Anbieter von Betriebssystemen, Programmierwerkzeugen und Klassenbibliotheken Java. Zum anderen erlangt Java dadurch einen selten erreichten Grad der Standardisierung und Plattformunabhängigkeit.

Aufgrund seiner weiten Verbreitung und seiner Nähe zu C bzw. C++ gelingt es Java, eines der Hauptprobleme von Smalltalk etwas zu entschärfen: den Mangel an qualizierten Software-Entwicklern. Smalltalk dagegen haftet immer noch der Ruf einer etwas exotischen Sprache an, die wenigen Entwicklern vertraut ist, da sich seine Syntax (und einige seiner Konzepte) doch recht deutlich von anderen Programmiersprachen unterscheiden.

Es gibt neben all diesen Aspekten einen gravierenden Unterschied zwischen Smalltalk und Java, der Java für die Entwicklung großer, langlebiger Systeme besonders attraktiv macht – die Art der Typisierung. Smalltalk ist eine *untypisierte* Sprache[1]: Smalltalk verzichtet auf Typdeklarationen für die Variablen des Programms. Dies hat zur Folge, daß der Typ einer Variablen erst zur Laufzeit des Programmes feststeht, und daher die Zulässigkeit von Operationen auf einem durch diese Variable angesprochenen Objekt erst dann geprüft werden kann.

Java dagegen gehört zu den *statisch typisierten* Sprachen. Java-Programme müssen über Typdeklarationen für die Variablen verfügen. Dadurch steht der Typ einer jeden Variablen bereits zur Übersetzungszeit fest und die Zulässigkeit von Operationen auf dem durch eine Variable angesprochenen Objekt kann schon zu diesem Zeitpunkt überprüft werden. Dies ermöglicht es, die Anzahl der Fehler, die zur Laufzeit eines Systemes auftreten können, zu reduzieren und sorgt damit für eine höhere Zuverlässigkeit des Systems. Zudem dienen die Typdeklarationen in Java-Programmen als zusätzliche Dokumentation des Programmcodes.

[1] oft spricht man auch von einer dynamisch typisierten Sprache

Wenn sich ein Unternehmen aufgrund solcher Überlegungen zu einem Wechsel der Entwicklungssprache von Smalltalk zu Java entschlossen hat, stellt sich nun noch die Frage, ob und warum vorhandene Altanwendungen umgesetzt werden sollten.

2.1.4 Nutzen der Umsetzung bestehender Smalltalk-Anwendungen nach Java

Wie wir schon in Abschnitt 1.1 gesehen haben, liegt ein Nutzen der Umsetzung bestehender Smalltalk-Anwendungen nach Java darin, daß Unternehmen den Wechsel der Entwicklungssprache vollziehen können. Dazu werden die Altanwendungen nach Java übertragen und können dann von Java-Entwicklern gepflegt und weiterentwickelt werden. Dadurch wird vermieden, daß sowohl für Java als auch für Smalltalk Know-How gewonnen und bewahrt werden muß. Es kann somit also ein *vollständiger* Technologiewechsel vollzogen werden.

Ein weiterer Nutzen liegt darin, daß sich die resultierenden Java-Anwendungen besonders für den verteilten Einsatz in firmeninternen Netzwerken *(Intranets)* oder auf zusätzlichen, neuen Rechnerplattformen (beispielsweise auch auf mobilen Systemen) eignen.

Wenn es gelingt, den ursprünglich untypiserten Smalltalk-Code in typisierten Java-Code umzuarbeiten, ist darüber hinaus eine Verbesserung der Zuverlässigkeit der Anwendung möglich, da durch die dann durchführbare statische Typprüfung zusätzliche Möglichkeiten zur Konsistenzsicherung gegeben sind.

Die beiden letzteren Aspekte erlangen besondere Bedeutung, wenn wir uns vergegenwärtigen, von welcher Art typische Smalltalk-Anwendungen sind. Eine 1997 durchgeführte Umfrage unter 600 Smalltalk-Entwicklern [McC97] hat ergeben, daß die überwiegende Mehrheit der von ihnen entwickelten Smalltalk-Anwendungen als strategisch wichtige Systeme für das operationale, tägliche Geschäft einer Unternehmung charakterisiert werden können. Zuverlässigkeit und Einsatzfähigkeit auf allen fur das Unternehmen relevanten Rechnerplattformen sind daher von großem Interesse.

Die meisten dieser Systeme sind während ihres Einsatzes kontinuierlich angepaßt und erweitert worden und enthalten eine große Menge (undokumentiertes) Wissen über Unternehmensprozesse, so daß eine vollständige Neuimplementierung kaum durchführbar scheint. Mit Hilfe des Reengineerings kann möglicherweise dennoch eine Migration zu Java vollzogen werden.

2.2 Vorgehensweise

Wir werden im folgenden einen Überblick über die von uns gewählte Vorgehensweise geben, mit der wir Smalltalk-Anwendungen in Java-Anwendungen überführen werden.

2.2.1 Aufgabenbereiche

Zunächst erkennen wir, daß unsere Grundaufgabe darin besteht, zu untersuchen, wie wir die Grundelemente von Smalltalk umsetzen können. Diese Grundelemente umfassen die eigentlichen Konstrukte der Sprache, sowie einen unbedingt notwendigen Teil der Standardklassenbibliothek. Smalltalk umfaßt vergleichsweise wenige Sprachkonstrukte. Wesentliche Elemente der Sprache sind Bestandteil der Standardbibliothek, beispielsweise Methoden zur Definition neuer Klassen und zur Objekterzeugung oder Klassen zur Realisierung von Kontrollstrukturen.

Diese Arbeit konzentriert sich auf diese Grundaufgabe und verwendet diese Erkenntnisse dann, um die eigentlichen Anwendungen zu übertragen.

Zusätzlich erforderlich ist die Umsetzung der übrigen Teile der Standardbibliothek, sowie weitere Bibliotheken, die für die Implementierung der Anwendung genützt wurden. Wir werden in Kapitel 6 sehen, daß diese Aufgabe unabhängig von der Umsetzung der eigentlichen Anwendung betrachtet werden sollte.

2.2.2 Arbeitsschritte

In diesem Abschnitt beschreiben wir das allgemeine Schema, mit dem wir Smalltalk-Anwendungen nach Java umsetzen wollen. Alle Verfahren, die wir in den nachfolgenden Kapiteln beschreiben werden, lassen sich auf dieses Schema zurückführen. Die Umsetzung geschieht in mehreren Phasen:

1. Zunächst müssen wir den Smalltalk-Code in eine Form bringen, in der sich die weiteren Umsetzungsschritte möglichst leicht durchführen lassen. Dazu lesen wir den Smalltalk-Code ein und überführen ihn mit Hilfe von Symbolentschlüsselung und Syntaxanalyse in einen *abstrakten Strukturbaum (AST, Abstract Syntax Tree)* [WG84] [ASU85]. Für diesen Schritt sollten nach Möglichkeit vorhandene Werkzeuge verwendet werden. Die meisten modernen Smalltalk-Umgebungen verfügen bereits über eine fertige Klassenbibliothek zur Lösung dieser Aufgabe.

2. Zur Umsetzung der unterschiedlichen Konzepte der beiden Sprachen Smalltalk und Java sind mehr oder minder aufwendige Analysen des Smalltalk-Codes notwendig. Diese Analysen führen wir anhand des abstrakten Strukturbaumes durch. Die dabei gewonnenen Informationen speichern wir in den Knoten des Strukturbaumes.

   Gegebenenfalls sind Veränderungen an der Struktur des Programmes erforderlich, diese führen wir ebenfalls auf Basis des Strukturbaumes durch.

3. Aus dem im vorigen Schritt angereicherten und gegebenenfalls modifizierten Strukturbaum erzeugen wir dann Java-Code.[2]

---

[2] Wir weisen darauf hin, daß Kommentare, die sich im ursprünglichen Smalltalk-Code befinden, in den neu entstehenden Java-Code übertragen und an passender Stelle eingefügt werden müssen, weil sie einen wesentlichen Beitrag zum Verständnis des Systems leisten.

Wir werden im Verlauf der Arbeit einige Verfahren darstellen (siehe Kapitel 4), die Java-Code erzeugen, der sich sehr eng am ursprünglichen Smalltalk-Code orientiert. Bei diesen Verfahren ist der zweite Schritt entsprechend wenig ausgeprägt. In Kapitel 5 werden wir jedoch ein Verfahren beschreiben, in dem wir ausführliche Programmanalysen benötigen und Modifikationen an der Struktur der Anwendung vornehmen müssen, um typisierten Java-Code zu erhalten.

Wir werden uns jedoch bemühen, die Struktur der ursprünglichen Smalltalk-Anwendung so weit wie möglich zu übernehmen, und Umstrukturierungen nur dann vorzunehmen, wenn dies erforderlich ist. Insbesondere vereinbaren wir, daß wir jede Smalltalk-Klasse der ursprünglichen Anwendung in eine separate Java-Klasse (mit einer entsprechenden Java-Methode für jede der in der ursprünglichen Smalltalk-Klasse vorhandenen Smalltalk-Methoden) in der entstehenden Neufassung der Anwendung umsetzen.

Kapitel 3

Grundkonzepte von Smalltalk und Java

In diesem Kapitel stellen wir die Grundkonzepte der beiden Programmiersprachen Smalltalk und Java zusammen, erarbeiten Unterschiede und Gemeinsamkeiten zwischen diesen Sprachen und skizzieren, wie die grundlegenden Konstrukte Smalltalks auf Java-Konstrukte abgebildet werden können.

3.1 Einführung

Wir beginnen dieses Kapitel, indem wir einen Überblick über die Herkunft, die Entwicklungsgeschichte und die charakteristischen Eigenschaften der beiden Sprachen Smalltalk und Java geben.

3.1.1 Smalltalk

Die Entwicklung Smalltalks begann Anfang der 70er Jahre im Rahmen des Forschungsprojektes *Dynabook* am *Palo Alto Research Center (PARC)* der Firma *Xerox*. Ziel des Dynabook-Projektes war es, ein kompaktes, leicht zu bedienendes Computersystem, das Dynabook, zu schaffen, welches auch von Laien zur Lösung zahlreicher Aufgaben aus beruflichen und privaten Bereichen einsetzbar sein sollte. Das Dynabook selbst wurde nie fertiggestellt, die Konzepte dieses Systems haben die Hardware- und Softwareindustrie jedoch nachhaltig beeinflußt: zahlreiche Ideen und Visionen des Dynabook-Projektes bilden die Grundlage der heutigen *Personal Computer*.

Zu den innovativen Konzepten des Dynabooks gehörte eine grafische Benutzeroberfläche in Form eines über eine Maus und Bildschirmmenüs bedienbaren Fenstersystems und eine darin integrierte Programmierumgebung, die es dem Benutzer ermöglichte, Softwarelösungen für seine individuellen Problemstellungen zu entwickeln: das Smalltalk-System.

Smalltalk wurde im Laufe der Zeit konsequent weiterentwickelt und verbessert und gehört heute zu den bedeutendsten objekt-orientierten Programmiersystemen.

Heutige Smalltalk-Systeme basieren auf dem Anfang der 80er Jahre entstandenen Standardsystem *Smalltalk-80*, welches detailliert in [Gol84] und [GR85] bzw. [GR89] beschrieben ist. Solche Systeme bestehen aus mehreren Komponenten:

- einer konzeptionell einfachen, objekt-orientierten *Programmiersprache*[1],
- einer *virtuellen Maschine*, auf der Smalltalk-Programme ausgeführt werden können,
- einer *grafischen Benutzerschnittstelle* zur Interaktion mit dem System,
- einer umfangreichen *Klassenbibliothek*, die die Möglichkeiten des Smalltalk-Systems in eigenen Programmen verfügbar macht und Lösungen für zahlreiche Standardaufgaben anbietet.

Der Schwerpunkt bei der Entwicklung von Smalltalk lag von Anfang an darin, dem Benutzer und Programmierer ein leicht zu erlernendes Werkzeug an die Hand zu geben, mit dem möglichst zügig funktionsfähige Anwendungen erstellt werden können.

Aus diesem Grund ist die eigentliche Programmiersprache des Systems einfach und konsistent aufgebaut: Grundelemente der Sprache sind *Objekte*, deren Funktionalität durch Operationen zur Bearbeitung dieser Objekte, den *Methoden*, erbracht wird. Fast alle Konzepte der Sprache werden realisert, indem an solche Objekte *Nachrichten (Messages)* geschickt werden, die dazu führen, daß bestimmte Methoden des jeweiligen Objektes ausgeführt werden. Smalltalk gilt daher als Programmiersprache, die von Grund auf objekt-orientiert ist.

Die umfangreiche Klassenbibliothek in Verbindung mit den Mechanismus der Vererbung, mit dessen Hilfe die objekt-orientierte Programmiertechnik die *Wiederverwendung* von vorhandenem Programmcode erlaubt, sowie die komfortable, grafische Benutzeroberfläche bewirken kurze Entwicklungszeiten bei der Erstellung von Anwendungen. Aus diesem Grund wird Smalltalk heute gerne in Projekten eingesetzt, die dem Bereich des *Rapid Prototyping* bzw. des explorativen Programmierens zuzuschreiben sind.

Im weiteren Verlauf dieses Kapitels werden wir sehen, daß die Grundkonzepte der Sprache in der Tat darauf ausgerichtet sind, eine komfortable und zügige Anwendungsentwicklung zu ermöglichen.

Vor der Ausführung einer in Smalltalk erstellten Anwendung wird der Smalltalk-Code durch einen *Compiler* in einen plattformunabhängigen Zwischencode *(Bytecode)* übersetzt, der dann von der virtuellen Maschine des

[1] Der Begriff Smalltalk bezeichnet je nach Zusammenhang entweder das komplette Programmiersystem oder nur die Programmiersprache an sich.

Smalltalk-Systems interpretiert und ausgeführt wird. Die virtuelle Maschine sorgt außerdem für eine automatische Speicherverwaltung und bildet die Schnittstelle zum (Hardware-)System, auf dem die Smalltalk-Umgebung läuft.

3.1.2 Java

Die Programmiersprache Java wurde zu Beginn der 90er Jahre von *Sun Microsystems* in einer Projektgruppe um James Gosling entwickelt. Diese Projektgruppe war auf der Suche nach einer geeigneten Sprache zur Programmierung von elektronischen Geräten aus dem Heimgerätemarkt und der Unterhaltungsindustrie, die besondere Anforderungen zu erfüllen hatte [Gos96]: Die Sprache sollte einfach zu erlernen sein und die Entwicklung von portablen, netzwerkfähigen, und robusten Anwendungen ermöglichen.

Die Gruppe um Gosling verwendete zunächst C++, stellte dann jedoch fest, daß diese Sprache einige der Anforderungen nicht im gewünschten Maße erfüllte. Gosling entschied sich daher, eine völlig neue Sprache zu entwickeln, die bewährte Elemente von C++ und anderer Programmiersprachen in sich vereinen und trotzdem übersichtlich bleiben sollte. Das Ergebnis dieser Bemühungen war Java.

Java ähnelt daher C++ (insbesondere in der Syntax), allerdings wurden zahlreiche Vereinfachungen und Verbesserungen vorgenommen, insbesondere verzichtet Java auf Zeigerarithmetik, auf die Möglichkeit, Operatoren zu definieren und zu überladen, und auf Mehrfachvererbung. Um die Entwicklung plattformunabhängiger Anwendungen zu unterstützen, wird genau wie beim Smalltalk-System der Java-Quellcode einer Anwendung durch einen Compiler in einen Bytecode übersetzt, der dann von einer virtuellen Maschine interpretiert und ausgeführt wird. Auch die virtuelle Maschine Javas bietet eine automatische Speicherverwaltung und stellt eine Schnittstelle zur übrigen Systemumgebung zur Verfügung.

Genau wie Smalltalk wird auch Java durch eine Klassenbibliothek ergänzt. Diese Klassenbibliothek deckt bestimmte Grundaufgaben der Programmierung ab, insbesondere stellt sie Komponenten zur Konstruktion einer grafischen Benutzeroberfläche zur Verfügung.

Die Entwicklung der Sprache Java (und insbesondere die der Standard-Klassenbibliotheken) ist noch nicht abgeschlossen – die Definition des jeweils aktuellen Sprachstandards obliegt *Sun Microsystems* und kann in beispielsweise [GJS96] (sowie [LY97] für die virtuelle Maschine) nachgelesen werden.

Obwohl Javas Verbreitung untrennbar mit der Erfolgsgeschichte des Internet und der Intranetze verbunden ist – dort bewährt sich die Sprache insbesondere, weil man mit ihr plattformunabhängige Anwendungen erstellen kann, und wegen diverser Sicherheitskonzepte, die man mit Hilfe der virtuellen Maschine implementieren kann – wurde die Sprache ursprünglich zur Programmierung intelligenter, elektronischer Geräte entworfen. Eine in diesem Bereich einsetzbare Sprache muß einerseits auf einfache Weise die Entwicklung von Anwendungen

für eine Vielzahl von Mikroprozessor-Plattformen unterstützen, andrerseits muß sie den Entwurf *stabiler* und *robuster* Systeme fördern. Wir werden in Abschnitt 3.6 dieses Kapitels sehen, daß Java den zweiten Aspekt besonders betont, indem es sich in Bezug auf sein Typsystem stark von Smalltalk unterscheidet.

3.2 Ausdrücke und Anweisungen

Smalltalk und Java gehören zur Familie der objekt-orientierten Programmiersprachen. Ein mit Hilfe solcher Programmierpsrachen erstelltes Anwendungssystem zur Lösung einer bestimmten Aufgabe enthält in der Regel folgende Elemente:

- *Objekte*, die für die Aufgabe und deren Lösung eine Rolle spielen.
- *Nachrichten*, die sich Objekte gegenseitig zusenden können, um mit miteinander zu kommunizieren.
- *Klassen*, mit Hilfe derer Objekte gleicher Art zu Familien zusammengefaßt und einheitlich beschrieben werden können.
- *Methoden*, die eine Folge von *Anweisungen* enthalten, mittels derer die Objekte auf Nachrichten, die sie erhalten haben, reagieren.

Wir beginnen unsere vergleichende Darstellung von Smalltalk und Java mit den Anweisungen, aus denen der Quellcode für die einzelnen Methoden aufgebaut wird. Diese Anweisungen werden in Form von *Ausdrücken* der jeweiligen Programmiersprache formuliert.

In Smalltalk dienen die Ausdrücke in der Regel dazu, Nachrichten an Objekte zu senden, die über *Variablen* angesprochen werden, oder an Variablen mittels "`:=`" neue Objekte zuzuweisen. Smalltalk kennt unterschiedliche Arten von Variablen: Klassen- und Instanzvariablen, Parameter von Methoden, verschiedene Formen von globalen Variablen, sowie lokale Variablen. Auf Klassen- und Instanzvariablen, sowie auf Parameter von Methoden gehen wir in Abschnitt 3.4 nochmals genauer ein, auf eine tiefergehende Diskussion der verschiedenen Formen von globalen und lokalen Variablen verweisen wir auf die einschlägige Smalltalk-Literatur (beispielsweise [Lew95]).[2]

Ein Smalltalk-Ausdruck (*Expression*) kann die folgenden verschiedenen Formen annehmen:

$$\text{expression} = \{\text{variable } \texttt{":="}\ \} \quad (\text{primary} \mid \text{message_expr}) \tag{3.1}$$

$$\text{primary} = \text{variable} \mid \text{literal} \mid \text{block} \tag{3.2}$$

$$\text{message_expr} = \text{unary_expr} \mid \text{binary_expr} \mid \text{keyword_expr} \tag{3.3}$$

[2] Zum besseren Verständnis einiger Code-Beispiele im weiteren Veraluf dieses Textes rufen wir uns jedoch in Erinnerung, daß lokale Variablen in Smalltalk vor ihrer Verwendung deklariert werden müssen. Das Code-Fragment `|x| x:=...` beispielsweise vereinbart und verwendet eine lokale Variable `x`.

```
3 max: 4  "Keyword Expression"
anArray at: 2 put: 'Hallo'  "Keyword Expression"
4 factorial  "Unary Expression"
3 + 5  "Binary Expression"
'Hello' , ' World'  "Binary Expression"
```

Abbildung 3.1: Verschiedene Smalltalk-Ausdrücke für Methodenaufrufe.

Wir gehen nun die in einem Smalltalk-Ausdruck möglichen Teilausdrücke der Reihe nach durch:

- Der Teilausdruck *message_expr* dient dazu Nachrichten, die anhand eines Namens, des sogenannten *Selectors*, identifiziert werden, an Objekte *(Receiver)* zu senden, worauf die Ausführung der Methode des Objekts (sofern diese existiert) angestoßen wird, die nach dem Selector benannt ist – es erfolgt somit ein Methodenaufruf.

  Smalltalk kennt verschiedene Formen von Nachrichten (bzw. entsprechender Methoden): *Keyword Expressions*, mittels derer Methoden mit einem oder mehreren Parametern aufgerufen werden, *Unary Expressions*, mittels derer Methoden ohne Parameter aufgerufen werden, und *Binary Expressions*, welche Operatoren aufrufen. Beispiele für diese verschiedenen Varianten von Methodenaufrufen sind in Abbildung 3.1 zu sehen.

  Ausdrücke für Methodenaufrufe können beliebig verschachtelt werden. Die beteiligten Teilausdrücke werden dabei in folgender Reihefolge ausgewertet:

  Unary Expressions $\rightarrow$ *Binary Expressions* $\rightarrow$ *Keyword Expressions.*

  Bei gleichrangigen Teilausdrücken haben die weiter links stehenden Ausdrücke stets Vorrang. Mit Hilfe von Klammern "`(`", "`)`" kann eine andere Auswertungsreihenfolge erzwungen werden.

  Der Ausdruck

  ```
  newArray := oldArray copyfrom: i to: i + 2 factorial
  ```

  kopiert beispielsweise aus dem Feld `oldArray` die Elemente $i, \ldots, i+2!$ in ein neues Feld `newArray`.

  Es ist auch möglich, ein und demselben Objekt eine ganze Sequenz von Nachrichten zu schicken *(Cascading)*, dazu werden die einzelnen Teilausdrücke mit "`;`" getrennt:

  ```
  anObject method1: arg1; method2: arg2
  ```

Smalltalk:

```
...
i := i + 1.
anArray at: i put: anObject.
anObject print.
...
```

Java:

```
...
i := i + 1;
anArray.put(i, anObject);
anObject.print();
...
```

Abbildung 3.2: Eine Anweisungsfolge in Smalltalk und Java im Vergleich.

- Mit Hilfe von *Literalen* können in Smalltalk bestimmte Objekte direkt im Quellcode des Programms definiert und an Variablen zugewiesen werden. Mögliche Literale sind: Zahlen, Zeichen, Zeichenketten, Symbole[3] und Inhalte von Feldern (Arrays).
- *Blöcke* fassen mehrere Ausdrücke in einem Objekt zusammen. Diese Ausdrücke können später ausgeführt werden, indem diesem Block die Nachricht `value` geschickt wird. Auf Blöcke gehen wir in Abschnitt 3.3 genauer ein, da mit ihrer Hilfe alle Kontrollstrukturen in Smalltalk realisiert sind.

Mehrere Ausdrücke können kombiniert werden, indem sie mittels "`.`" aneinandergefügt werden. Es entsteht dann eine Anweisungsfolge wie sie beispielsweise im oberen Teil der Abbildung 3.2 zu sehen ist.

In Java sind Anweisungen (bzw. die Ausdrücke mittels derer die Anweisungen formuliert werden) ähnlich aufgebaut wie in Smalltalk. In Abbildung 3.2 sind zu einer Folge von Anweisungen in Smalltalk die entsprechenden Java-Anweisungen zu sehen.

Allerdings gibt es einige Unterschiede, auf die wir hier detailliert eingehen wollen:

- Java kennt nur eine einzige Variante des Methodenaufrufs, nämlich die der Form:

  ```
  anObject.method(arg1,...)
  ```

  Wir müssen daher die verschiedenen Varianten für Methodenaufrufe Smalltalks *(Unary Messages, Binary Messages, Keyword Messages)* auf

[3] Symbole ähneln Zeichenketten, deren Inhalt nicht verändert werden kann.

Smalltalk	Java
`anObject printString`	`anObject.printString()`
`charTable comma`	`charTable.comma()`
`a + b`	`a.plus(b)`
`aString , anotherString`	`aString.comma(anotherString)`
`aBag add: anObject`	`aBag.add_(anObject)`
`myString comma: anotherString`	`myString.comma_(anotherString)`
`anArray at: 1 put: 'Hello'`	`anArray.at_put_(1, "Hello")`

Abbildung 3.3: Umsetzung von Methodenaufrufen.

diese Form abbilden, so daß dabei keine Namenskonflikte entstehen können.

- Eine *Unary Message* bilden wir direkt ab, indem wir in Java einen parameterlosen Methodenaufruf verwenden.
- *Binary Messages* verknüpfen stets ein Objekt (*Receiver*) mit einem weiteren Parameter mittels eines Operators. Da Java keine selbstdefinierten Operatoren kennt, müssen wir eine Ersatzkonstruktion schaffen. Wir verwenden dazu einen textuellen Bezeichner, der dem Operator entspricht, um einen Java-Methodenaufruf mit einem Parameter zu erzeugen.
- *Keyword Messages* senden eine Nachricht zum Aufruf einer Methode mit einem oder mehreren Parametern. Jeder Parameter wird dabei mit einem Schlüsselwort (gefolgt von einem Doppelpunkt) eingeleitet. Um einen solchen Ausdruck nach Java umzusetzen, fassen wir die einzelnen Schlüsselwörter zu einem neuen Schlüsselwort zusammen, wobei wir an den Nahtstellen einen Unterstrich "_" einfügen, und konstruieren daraus einen Java-Methodenaufruf mit einem oder mehreren Parametern.

Einige Beispiele für diese Umsetzung sind in der Tabelle in Abbildung 3.3 angegeben. Diese Abbildung demonstriert am Beispiel der Nachrichten "`,`" und `comma` auch, daß bei der Umsetzung der Methodennamen (Selektoren) keine Namenskonflikte entstehen.

- Ebenso wie in Smalltalk können in Java Methodenaufrufe ineinander verschachtelt werden. Die dabei beteiligten Teilausdrücke werden in Java von links nach rechts ausgewertet. Explizite Klammerungen entfallen, da diese bereits in der Syntax der Methodenaufrufe enthalten sind. Um die Auswertungsreihenfolge einer komplexen Smalltalk-Anweisung in Java nachzubilden, kann es notwendig sein, diese in mehrere Einzelanweisungen zu zerlegen.

```
|i blck1 blck2 |
"Erzeugen des Blocks blck1:"
blck1 := [i := i + 1].
i := 1.
"Ausführen des Blocks blck1:"
blck1 value.
"Hier gilt: i = 2"

"Erzeugen des Blocks blck2:"
blck2 := [:x | x + 1].
"Ausführen des Blocks blck2:"
i := blck2 value: i.
"Hier gilt: i = 3"
```

Abbildung 3.4: Verwendung von Blöcken in Smalltalk.

- Java kennt keine Konstruktion, die Smalltalks *Message Cascading* entspricht. Wir müssen daher Ausdrücke der Form

  ```
  aShape hide; moveto: newLocation; aShape show
  ```

  in eine Folge von Einzelanweisungen übersetzen.

  ```
  aShape.hide(); aShape.moveto_(newLocation); aShape.show();
  ```

- Die Umsetzung von literalen Ausdrücken ist etwas aufwendiger, da dabei passende Objekte für deren Werte erzeugt werden müssen. Wir werden daher auf die Umsetzung von Literalen in Abschnitt 3.4) eingehen, wenn wir uns dort mit der Erzeugung von Objekten auseinandersetzen.

3.3 Blöcke und Kontrollstrukturen

In Smalltalk läßt sich eine Folge von Anweisungen zu *Blöcken* zusammenfassen. Beginn und Ende eines Blockes werden dabei mit eckigen Klammern "`[`", "`]`" markiert. Die im Block enthaltenen Anweisungen werden jedoch nicht unmittelbar an der Definitionsstelle des Blocks ausgeführt, sondern sie erzeugen ein Smalltalk-Objekt (eine sogenannte *Closure*), das diese Anweisungen zu einem späteren Zeitpunkt ausführen kann, wenn ihm die Nachricht `value` geschickt wird. Wenn ein Block ausgeführt wird, hat er stets Zugriff auf die Variablen der Umgebung, in der er definiert wurde. Beispiele für die Definition und Verwendung von Blöcken sind in Abbildung 3.4 gegeben.

Es lassen sich auch Blöcke definieren, die ein Argument erwarten. Dieses Argument wird dem Block bei der Auswertung durch eine `value:`-Methode mit einem Parameter übergeben. Abbildung 3.4 enthält auch dafür ein Beispiel.

Smalltalk:

```
...
(a > b) ifTrue: [max := a] ifFalse: [max := b].
...
```

Java:

```
...
if (a > b) {
  max := a;
}
else {
  max := b;
}
...
```

Abbildung 3.5: Eine Verzweigung in Smalltalk und in Java.

In Smalltalk werden alle Kontrollstrukturen (Verzweigungen, Schleifen,...) mit Hilfe Blöcken, die Argumente von Methodenaufrufen an bestimmte Objekte darstellen, realisiert.

Betrachten wir hierzu das Smalltalk-Äquivalent zu den in anderen Programmiersprachen üblichen *if-then-else*-Konstrukten aus Abbildung 3.5. In Smalltalk entsteht dieses Konstrukt durch einen Methodenaufruf `ifTrue:ifFalse:` an ein Objekt der Klasse `Boolean`. Die Argumente sind dabei Block-Objekte für den Code der beiden möglichen Zweige. Je nach Wert des boolschen Objektes führt die Methode `ifTrue:ifFalse` das passende Block-Objekt aus, indem es diesem eine `value`-Nachricht schickt.

In ähnlicher Weise stellt Smalltalk auch weitere Kontrollstrukturen zur Verfügung, beispielsweise solche für Schleifen:

Boolean `whileTrue:`*Block*

Für die wichtigsten dieser Kontrollstrukturen erzeugt der Smalltalk-Übersetzer *optimierten* Bytecode, indem er sie durch Sprünge implementiert.

Alle anderen Konstruktionen, die Blöcke verwenden, setzt der Smalltalk-Übersetzer in Bytecode um, der für die Blöcke tatsächlich Objekte anlegt und diese mittels `value:` auswertet. Dies gilt insbesondere für Blöcke, die ein Argument erwarten.

Mit Hilfe solcher Block-Objekte lassen sich einfach mächtige benutzerdefinierte Kontrollstrukturen schaffen und verwenden. Blöcke sind daher ein wichtiges Sprachmittel in Smalltalk, welches ausgesprochen oft genutzt wird.

Java kennt keine Block-Objekte. Zur Umsetzung von Block-Konstrukten aus Smalltalk nach Java wenden wir daher folgende Strategien an:

in Klasse Collection:
```
do: aBlock
    Führe für jedes Element elem aus:
        aBlock value: elem
```

Code zur Iteration durch eine Collection:
```
aCollection do: [ :x | x print ]
```

Abbildung 3.6: Iteration durch eine `Collection` in Smalltalk.

- Konstruktionen, aus denen Smalltalk-Übersetzer optimierten Code erzeugen würde, setzen wir mit Hilfe passender Java-Kontrollstrukturen um. `ifTrue:ifFalse:`-Verzweigungen und `whileTrue:`-Schleifen können wir beispielsweise mit Hilfe von `if`-Verzeigungen bzw. `while`-Schleifen übersetzen. Besonderes Augenmerk müssen wir dabei jedoch darauf legen, daß wir die Semantik der Smalltalk-Konstrukte exakt nach Java übertragen – dies bedeutet beispielsweise, daß wir die Regeln Smalltalks zur *Kurzauswertung* von boolschen Ausdrücken mit `and:`, `or:`, ... berücksichtigen müssen.

- In allen anderen Fällen legen wir explizit eine Java-Klasse an, die den Code des Blockes in einer Methode `value()` enthält und verwenden eine Instanz davon als Block-Objekt.

  Wir illustrieren dies an einem Beispiel aus der Smalltalk-80-Klassenbibliothek. Diese stellt eine Vielzahl von Container-Klassen zur Verfügung (in Form von Unterklassen von *Collection*). Jede Container-Klasse verfügt über eine Methode `do:`, die als Argument einen einargumentigen Block erwartet. Dieser Block wird dann auf alle Elemente des jeweiligen Containers angewandt. Mit dieser Hilfe dieser Konstruktion kann komfortabel durch den Container iteriert werden (siehe Abbildung 3.6). Um dies nach Java umzusetzen, definieren wir für den Block eine *anonyme* Klasse, die eine Methode `value()` enthält und übergeben diesen an die `do()`-Methode der nach Java umgesetzten Klasse `Collection` (siehe Abbildung 3.7).

3.4 Objekte, Klassen und Methoden

Wir haben bereits erwähnt, daß eine in objekt-orientierten Programmiersprachen wie Smalltalk und Java geschriebene Anwendung aus weitgehend unabhängigen Einzelbausteinen, den Objekten, besteht, von denen jeder einen (kleinen) Teil zur Gesamtfunktionalität der Anwendung beiträgt.

Jedes Objekt *kapselt* Daten (in Form von Instanzvariablen) und Operationen (Methoden) zur Manipulation dieser Daten. Objekte gleicher Art werden

```
public abstract class STBlock {
   abstract public STObject value_(STObject anObject);
}

public class STCollection {
   ...
   public STObject do_ (STBlock aBlock) {
      Führe für jedes Element elem aus:
         aBlock.value(elem)
   }
   ...
}

Code zur Iteration durch eine Collection:
   aCollection.do_ (new STBlock() {
       public STObject value_ (STObject anObject) {
          anObject.print();
      }
   });
```

Abbildung 3.7: Umsetzung des Code-Beispiels aus Abb. 3.6 nach Java.

zu Klassen zusammengefaßt, damit wir sie beschreiben und implementieren können. Klassen stellen somit gewissermaßen eine Schablone dar, anhand der die individuellen Objekte erzeugt werden können. Jedem Objekt können wir daher genau eine Klasse zuordnen, nämlich die, aus der es erzeugt worden ist.

In einer Klasse müssen sowohl die Instanzvariablen als auch die Methoden beschrieben werden, über die die Objekte dieser Klasse verfügen sollen. In Smalltalk werden Klassen und die zugehörigen Instanzvariablen im *Class Browser* der integrierten Entwicklungsumgebung definiert und im System *(virtual Image)* abgelegt. Anschließend werden den Klassen Methoden und der dazugehörige Programmcode hinzugefügt. Dies geschieht ebenfalls mit Hilfe des *Class Browsers.* Die Syntax von Smalltalk enthält daher keine Konstrukte zur Definition von Klassen. In Java dagegen geschieht die Definition von Klassen und ihrer Methoden mit Hilfe von Syntaxelementen – solche Klassendefinitionen in Java sind beispielsweise in Abbildung 3.7 zu finden. Bei der Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen müssen wir die Definitionen der Klassen der Anwendung aus dem Smalltalk-System extrahieren und entsprechende Java-Syntaxkonstrukte erzeugen.

Zugriffe auf die Daten eines Objektes (Instanzvariablen) sind in Smalltalk nur über Methoden, die in der zugehörigen Klasse definiert sind, erlaubt. In Java dagegen kann der Zugriff auf Instanzvariablen individuell geregelt werden, indem deren Deklarationen mit entsprechenden Schlüsselwörtern versehen werden (`private`, `protected` oder `public`). Instanzvariablen in nach Java umgesetzten

Smalltalk-Klassen deklarieren wir mit dem Schlüsselwort `protected`, da dies die Zugriffsmodalitäten von Smalltalk am besten nachbildet. Der Zugriff auf Methoden *(Sichtbarkeit)* läßt sich in Java ebenfalls durch diese Schlüsselwörter individuell regeln – in Smalltalk sind dagegen alle Methoden eines Objektes aus Methoden *beliebiger* anderer Objekte aufrufbar. Nach Java umgesetzte Smalltalk-Methoden deklarieren wir deshalb mit dem Schlüsselwort `public`.[4]

Zur Definition einer Klasse gehört auch die Angabe von *Methodensignaturen*. Diese beschreiben die Parameter und Rückgabewerte der Methoden. In Smalltalk bestehen solche Signaturen aus dem Namen *(Selector)* der Methode und den Namen der Parameter, die den Methoden beim Aufruf übergeben werden. In Java gehören zur Methodensignatur zusätzlich noch *Typdeklarationen* für die Parameter und den Rückgabewert der Methode. Wir gehen in Abschnitt 3.6 genauer darauf ein, wozu Java diese Typdeklarationen benötigt. Welche Typdeklarationen wir für Java-Methoden angeben, die bei der Umsetzung von Smalltalk-Methoden ohne Typdeklarationen entstehen, werden wir in den Kapiteln 4 und 5 noch genauer erörtern.

Die Semantik der Parameterübergabe beim Methodenaufruf ist in Smalltalk und Java prinzipiell gleich, so daß wir diesen Aspekt bei der Umsetzung von Smalltalk-Code in Java-Code nicht mehr besonders beachten müssen: als Parameter werden Verweise auf Objekte übergeben. Änderungen an den Daten der übergebenen Objekte sind daher nach dem Beenden des Methodenaufrufs im aufrufenden Programmteil verfügbar *(Call by Reference)*. Auch die Rückgabewerte von Methoden werden per Verweis übergeben.[5]

Zum Abschluß dieses Abschnittes betrachten wir noch die verschiedenen Möglichkeiten, Objekte aus den Klassenbeschreibungen zu erzeugen. In Smalltalk wird zur Erzeugung eines Objekts an die entsprechende Klasse die Nachricht `new` geschickt. (Wir können in Smalltalk auch an Klassen Nachrichten schicken, da diese ebenfalls Objekte darstellen – siehe Abschnitt 3.7). Ergebnis des Aufrufs der Methode `new` ist dann ein neues Objekt. So erzeugt

```
aPoint := Point new
```

ein neues Objekt der Klasse `Point` und weist dieses Objekt der Variablen `aPoint` zu.

Nach der Erzeugung muß das Objekt explizit initialisiert werden, das heißt, die Instanzvariablen des Objektes müssen sinnvoll vorbelegt werden. Dies geschieht

[4] Mit den Zugriffsbeschränkungen auf Objekte eng verwandt sind auch Javas *Packages*. Packages ermöglichen die Definition von Sichtbarkeitsregeln und getrennten Namensräumen für zusammengehörende Klassen. Da Smalltalk ein solches Konstrukt nicht kennt, spielt dieser Aspekt für unsere Reengineering-Aufgabe jedoch keine wesentliche Rolle.

[5] Primitive Typen `int`, `float`, ... werden in Java allerdings in Form von Werten übergeben *(Call by Value)*, dies spielt allerdings bei der Umsetzung von Smalltalk-Code in Java-Code keine Rolle, da diese primitiven Typen in Smalltalk in Form vollwertiger Objekte dargestellt werden, und wir diese als ebensolche nach Java übertragen (siehe unten).

durch den Aufruf einer separaten Methode, die der Programmierer der zugrundeliegenden Klasse zur Verfügung gestellt hat. Am Beispiel der Klasse Point könnte dies wie folgt aussehen:

```
aPoint setX: 1.0 Y: 0.0
```

In Java werden Erzeugung und Initialisierung eines Objektes zusammengefaßt. Jede Klasse definiert zur Initalisierung einen *Konstruktor*[6]. Dies ist eine spezielle Methode, die den Namen der Klasse trägt, und zusammen mit der `new`-Anweisung Javas aufgerufen wird.

```
aPoint := new Point(1.0, 0.0);
```

Um die in Smalltalk zur Objekterzeugung und Initialisierung übliche Vorgehensweise in Java nachzubilden, können wir folgenden Code verwenden:

```
aPoint := new Point();  // Standardkonstruktor
aPoint.setX_Y(1.0, 0.0);
```

Eine andere Möglichkeit, die Art der Objekterzeugung Smalltalks nach Java umzusetzen, besteht in der Nutzung von Metaklassen (siehe Abschnitt 3.7). Details hierzu finden sich in [EK98].

Auch mittels literaler Ausdrücke (siehe Abschnitt 3.2) können in Smalltalk Objekte erzeugt werden. So erzeugt

```
x := 'Hallo'
```

eine Instanz der Klasse `String`, initialisiert sie mit *"Hallo"* und weist sie der Variablen `x` zu.

Wir bilden dies in Java nach, indem wir die Klasse `STString`, die die Umsetzung der Smalltalk-Klasse `String` enthält, mit einem geeigneten Konstruktor versehen, und den obenstehenden Smalltalk-Code in folgenden Java-Code umsetzen:

```
x := new STString("Hallo");
```

In Java sind ebenfalls Konstrukte mit Literalen möglich. Allerdings sind diese nur zusammen mit primitiven Typen (wie beispielsweise `char`, `int`, ...) verwendbar. Wir merken hierzu an, daß wir Smalltalks Klassen zur Darstellung von Zeichen (`Char`) oder Ganzahlen (`Int`) nicht auf Javas primitive Typen abbilden können, da in den jeweiligen Smalltalk-Klassen weitaus mehr Funktionalität zur Verfügung steht, als in den primitiven Typen Javas. Beispielsweise enthält die Smalltalk-Klasse `Int` ausgefeilte Routinen zur Behandlung von Überläufen, indem eine Konversion zu `LongInt` durchgeführt wird.

[6] Falls der Programmierer der Klasse nicht explizit einen solchen zur Verfügung stellt, verwendet der Java-Übersetzer einen geeigneten Standardkonstruktor.

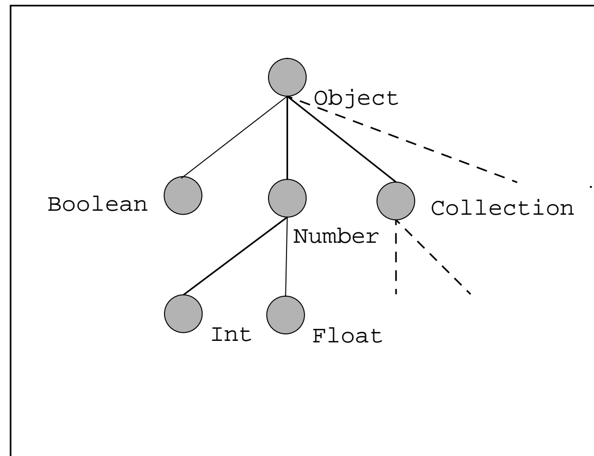

Abbildung 3.8: Klassengraph (Vererbungshierarchie)

3.5 Vererbung

Ein wesentliches Konzept objekt-orientierter Programmiersprachen ist die *Vererbung*. Indem wir eine Klasse B von einer Klasse A erben lassen, erreichen wir, daß die Klasse B die Eigenschaften der Klasse A übernimmt, ohne daß wir diese in der Klasse B nochmals definieren oder implementieren müssen. B können wir dann bequem um zusätzliche Funktionalität ergänzen. Wenn eine Vererbungsbeziehung von A zu B besteht, so benützen wir für A oft den Begriff *Oberklasse von B*, umgekehrt für B den Begriff *Unterklasse von A*. Wenn wir alle Vererbungsbeziehungen, die zwischen den Klassen der Anwendung bestehen, zusammenstellen, so erhalten wir eine Klassenhierarchie, die wir in Form eines Graphen darstellen können. Ein solcher Klassengraph, der Ausschnitte aus dem *Smalltalk-80*-System zeigt, ist in Abbildung 3.8 dargestellt.

Konkret vererben sich in Smalltalk und in Java sowohl Attribute (Instanz- und Klassenvariablen) als auch Methoden einer Oberklasse auf eine entsprechende Unterklasse, so daß wir diesbezüglich keine Unterschiede zwischen Smalltalk und Java feststellen können. Allerdings wird Vererbung in Smalltalk unter Umständen aus anderen Gründen eingesetzt als in Java. Hierauf werden wir in Abschnitt 3.6 noch genauer eingehen.

Sowohl in Smalltalk als auch in Java ist nur Einfachvererbung möglich: Jede Klasse darf lediglich genau eine direkte Oberklasse haben. Java bietet anstelle der Mehrfachvererbung einen Ersatzmechanismus, die sogenannten *Interfaces*. Interfaces können als Klassebeschreibungen angesehen werden, die nur die Schnittstellen von Methoden enthalten. In Java darf eine Klasse beliebig viele Interfaces implementieren ("ererben"). Diese "Mehrfachvererbung" von Interfaces erweist im Zusammenhang mit Javas Konzept zur Typiserung und Polymorphie oft als nützlich. Wir werden davon in Kapitel 5 noch Gebrauch machen.

Verwandt mit dem Begriff der Interfaces sind *abstrakte Klassen*. Wir bezeichnen eine Klasse als abstrakt, wenn sie mindestens eine abstrakte Methode enthält. Eine abstrakte Methode besteht nur aus einer Schnittstellendefinition – eine

Implementierung der Methode ist nicht vorhanden. Abstrakte Klassen können wir nützen, um Gemeinsamkeiten der Klassenschnittstellen verschiedener Klassen in einer Oberklasse zusammenzufassen, ohne daß wir dazu eine gemeinsame Implementierung angeben müssen. Auch dieses Konzept ist im Zusammenhang mit Polymorphie von besonderer Bedeutung (siehe Abschnitt 3.6.

In Java werden abstrakte Methoden mit dem Schlüsselwort `abstract` markiert, in Smalltalk gibt es kein direkt vergleichbares Konzept, abstrakte Methoden können aber nachgebildet werden, wenn die Implementierung einer solchen Methode lediglich aus einem Aufruf der speziellen Methode `subClassResponsibility` besteht. In einer Umsetzung von Smalltalk nach Java werden wir Methoden, die lediglich diesen Aufruf enthalten, und deren Klassen demnach in abstrakte Java-Methoden und -Klassen übersetzen.

3.6 Typisierung und Polymorphie

Wie bereits erwähnt gehört Smalltalk zu den *dynamisch typisierten* Programmiersprachen: Smalltalk kennt keine Typdeklarationen. Variablen in Smalltalk werden zwar im Programmcode deklariert, es wird jedoch dabei kein fester Typ in Form einer Klasse, von der die Objekte, auf die die Variable zeigt, Instanzen sein müssen, vereinbart. Zur Laufzeit kann daher zu einem Zeitpunkt ein Objekt einer bestimmten Klasse A an die Variable zugewiesen werden, zu einem anderen Zeitpunkt ein anderes Objekt einer völlig anderen Klasse B.

Immer, wenn eine Variable in einem Ausdruck verwendet wird, um einem Objekt eine Nachricht zu senden, muß eine passende Methode mit gleicher Signatur – also mit dem Namen der Nachricht und derselben Anzahl von Parametern – in der zum Objekt gehörenden Klasse oder einer deren (direkter oder indirekter Oberklassen) vorhanden sein, damit ein Methodenaufruf stattfinden kann. Da jedoch zur *Übersetzungszeit* mangels Typdeklarationen nicht feststeht, von welcher Klasse das durch die Variable angesprochene Objekt ist, kann dies lediglich zur *Laufzeit* – wenn die Klasse des Objektes bekannt ist – überprüft werden. Das Senden einer Nachricht, auf die das Objekt mangels einer passenden Methode nicht reagieren kann, verursacht daher einen Laufzeitfehler *("Message not understood.")*.

In Smalltalk besteht demnach eine konzeptionelle Trennung zwischen dem Verschicken von Nachrichten an Objekte *(Message Passing)* und dem daraus resultierenden Methodenaufruf: Eine Nachricht führt nur dann zu einem Methodenaufruf, wenn das dabei beteiligte Objekt über eine geeignete Methode verfügt.

Java dagegen ist *statisch typisiert*: Einer Variable wird bereits zur Übersetzungszeit eine feste Klasse zugeordnet – der Variablen können nur Objekte zugewiesen werden, die Instanzen der deklarierten Klasse oder einer deren Unterklassen sind. Die Deklaration einer Variable *beschränkt* daher die Art und Weise, in der sie verwendet werden darf – Verletzungen dieser Beschränkung können bereits zur Übersetzungszeit ermittelt werden. Insbesondere gilt dies für

Methodenaufrufe: Wenn eine Variable in einem Java-Ausdruck für einen Methodenaufruf verwendet wird, und eine passende Methode weder in der Klasse, mit der die Variable deklariert wurde, noch in einer ihrer Oberklassen vorhanden ist, meldet der Java-Compiler einen Übersetzungsfehler.

Dieser Unterschied in der Typisierung hat Auswirkungen auf die Art und Weise, in der die beiden Sprachen *Polymorphie* unterstützen. Wir sprechen von Polymorphie, wenn ein- und derselbe Programmcode mit Objekten unterschiedlicher Art arbeiten kann. Objekt-orientierte Sprachen unterstützen dies, indem Variablen Objekte unterschiedlicher Typen (Klassen) zugewiesen werden können. Die Methodenaufrufe auf solchen Variablen erfolgen dann polymorph, weil für den Empfänger *(Receiver)* des Aufrufs Objekte verschiedener Gestalt möglich sind. Betrachten wir dazu folgendes Beispiel:

Beispiel 3.1 Gegeben seien Klassen zur Darstellung geometrischer Objekte: `Point`, `Line`, `Rectangle`, ..., die jeweils über die Methoden `move` und `draw` zum Verschieben bzw. Zeichnen des jeweiligen Objekts verfügen. Diese Klassen sind in Abbildung 3.9) grau unterlegt dargestellt. Wir nehmen an, daß solche Basisobjekte mit Hilfe einer Klasse `Drawing` zu einer komplexen Zeichnung zusammengefügt werden können, wobei die einzelnen Objekte in einer Containerklasse `objects` gespeichert werden.

Abbildung 3.10 zeigt eine Smalltalk-Methode `add` der Klasse `Drawing` mit der geometrische Objekte der Zeichnung hinzugefügt und anschließend angezeigt werden. Die Methode `add` arbeitet polymorph, weil der Parameter `anOnject` *beliebige* geometrische Objekte ansprechen kann. Wir beachten aber, daß alle diese Objekte die Nachricht `draw` verstehen müssen, ansonsten entsteht ein Laufzeitfehler.

Smalltalk ermöglicht die polymorphe Verwendung von Variablen in völlig flexibler Art und Weise: Jeder Variablen darf jedes beliebige Objekt zugewiesen werden und an die über die Variablen referenzierten Objekte darf jede beliebige Nachricht geschickt werden (evtl. kann jedoch zur Laufzeit ein *"Message not understood"*-Fehler auftreten).

In Java dagegen dürfen einer Variablen nur Objekte aus der Klasse des deklarierten Typs oder aus Unterklassen davon zugewiesen werden. Die polymorphe Verwendung von Variablen ist daher auf Objekte dieser Klassen eingeschränkt.

Dies hat Auswirkungen, wenn wir unser Beispiel 3.1 nach Java übertragen wollen: In `add` benötigen wir einen polymorphen Aufruf der Methode `draw` auf `anObject`, welches ein Objekt aus einer der Klassen `Point`, `Line`, `Rectangle` enthalten kann. Methodenaufrufe auf einer Variablen sind jedoch nur dann erlaubt, wenn die Methoden in der Klasse des deklarierten Typs (oder einer Oberklasse) enthalten sind. Wir benötigen also einen einheitlichen Typ in Form einer Oberklasse `Shape` von `Point`, `Line`, `Rectangle`,..., mit der wir die Variable `anObject` deklarieren können. Diese Oberklasse muß die Methode `draw` enthalten. Eine solche Oberklasse ist in Abbildung 3.9 bereits eingezeichnet, korrekter Java-Code für die Methode `add` ist in Abbildung 3.11 zu sehen.

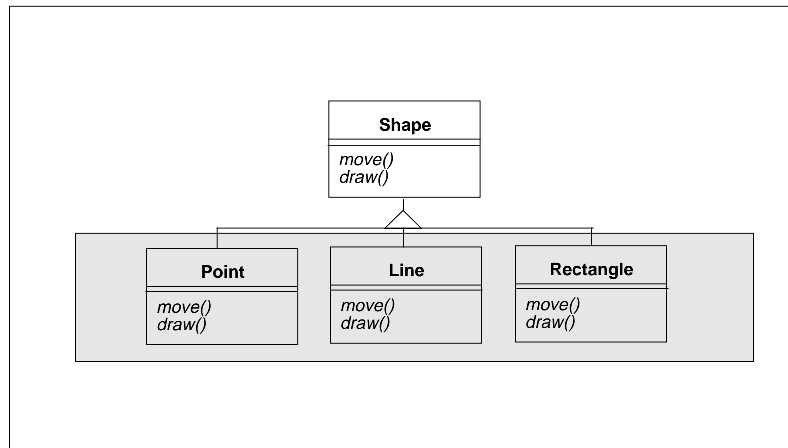

Abbildung 3.9: Klassen zur Modellierung geometrischer Figuren

In Klasse `Drawing`:

```
add: anObject
   objects add: anObject.
   anObject draw.
```

Abbildung 3.10: Smalltalk-Methode `add`, die polymorphen Code enthält.

```
class Drawing {
   //...
   add (Shape anObject) {
      objects.add((Object)anObject);
      anObject.draw()
   }
   //...
}
```

Abbildung 3.11: Java-Variante der Methode `add`.

Allgemein gilt: Polymorpher Code ist in Java nur dann möglich, wenn die polymorph zu verwendenden Klassen über eine *gemeinsame Schnittstelle* in Form einer Oberklasse verfügen und diese Schnittstelle alle Methoden enthält, die in diesem Code polymorph aufgerufen werden. Wir verwenden in Java somit Vererbung (unter anderem) zur Konstruktion von Klassenhierarchien, die Gemeinsamkeiten der Schnittstellen der Klassen modellieren (Stichwort: *"Programming to an Interface"*).

Wir stellen fest, daß statische Typisierung und die Betonung der Schnittstellen der Klassen eng mit der Zielsetzung von Java zusammenhängen – beide fördern die Konstruktion robuster und wartbarer (und damit langlebiger) Anwendungen:

- Typdeklarationen ermöglichen es, Typverträglichkeitsfehler und ungültige Methodenaufrufe schon zur Übersetzungszeit zu identifizieren – sie erhöhen daher die Zuverlässigkeit der Anwendung.
- Typdeklarationen tragen zum Verständnis des Programmcodes bei (zusätzliche Dokumentation), und erleichtern so die Wartung der Anwendung.
- Durchdachte Klassenhierarchien mit passenden Schnittstellen strukturieren die Anwendung und tragen so ebenfalls zur Wartbarkeit der Anwendung bei.

Smalltalks dynamische Typisierung und die damit zusammenhängende Flexibilität bei der polymorphen Verwendung von Programmcode kann hingegen zu einer besonders schnellen Anwendungsentwicklung beitragen, da vorhandener Code in Verbindung mit völlig neuen Klassen wiederverwendet werden kann, ohne daß diese zunächst aufwendig in eine Schnittstellenhierarchie eingebettet werden müssen. Insbesondere im Bereich des explorativen Programmierens kann es zudem schwierig sein, bereits in frühen Phasen der Anwendungsentwicklung solide Klassenhierarchien zu konstruieren – bei prototypischen Implementierungen ist es wichtiger, Funktionalität aus vorhandenen Klassen zu ererben und diese zu modifizieren und zu erweitern, da so die Entwicklungszeiten verkürzt werden können.

Wir erinnern uns daran, daß ein Entwurfsziel von Smalltalk darin bestand, eine Umgebung zu schaffen, in der auf einfache und schnelle Weise Anwendungen entwickelt werden können – insofern sind dynamische Typisierung und uneingeschränkte Polymorphie passende Paradigmen für Smalltalk.

Der in diesem Abschnitt dargestellte Unterschied zwischen Smalltalk und Java – dynamische Typiserung in Smalltalk, statische Typisierung in Java – hat erhebliche Konsequenzen für unser Problem, Smalltalk-Anwendungen nach Java umzusetzen: Javas Mechanismus zum Methodenaufruf erfordert, daß der Typ einer Variablen (in Form einer Klasse) zur Übersetzungszeit bekannt und im Programmcode deklariert ist, damit die Variable in Methodenaufrufen verwendet werden kann. In Smalltalk liegt diese Typinformation jedoch erst zur

Laufzeit vor. Es scheint also zuächst so, daß unsere Reengineering-Aufgabe an diesem Konflikt scheitert.

Glücklicherweise können wir diesen Konflikt auflösen, indem wir eine der folgenden Strategien anwenden:

1. Wir können, wenn wir Smalltalk-Code nach Java übertragen wollen, entweder alle Variablen im Java-Code so deklarieren, daß sie Objekte aller in der Anwendung vorkommender Klassen beeinhalten können, indem wir die Wurzelklasse der Vererbungshierarchie zur Deklaration verwenden. (Die Variablen sind dann praktisch untypisiert.) Dies erfordert aber, daß wir Javas Mechanismus für den Methodenaufruf umgehen und durch einen eigenen Mechansimus ersetzen.

2. Wir können selbst versuchen, korrekte Typen für die Variablen zu bestimmen, indem wir den Smalltalk-Code analysieren. Mit Hilfe dieser Typen können wir dann Typdeklarationen im Java-Code erstellen und so Javas Mechanismus zum Methodenaufruf unverändert nutzen.

In Kapitel 4 werden wir uns mit Ansätzen zur Umsetzung von Smalltalk-Code in Java-Code beschäftigen, die der erstgenannten Strategie folgen, in Kapitel 5 werden wir einen Ansatz entwickeln, der die zuletztgenannte Vorgehensweise realisiert.

3.7 Metaklassen und Introspektion

Wir haben in Abschnitt 3.4 bereits erwähnt, daß Klassen in Smalltalk ebenfalls Objekte sind, und daß wir Instanzen einer Klasse erzeugen können, indem wir diesen Objekten die Nachricht `new` zusenden. Da es zu jedem Objekt eine Klasse geben muß, welche sein Verhalten beschreibt und für seine Erzeugung verantwortlich ist, benötigen wir wiederum Klassen zur Beschreibung von Klassen. In Smalltalk sind dies die sogenannten *Metaklassen*. Zu jeder Klasse existiert in Smalltalk eine Metaklasse, in der Methoden, sogenannte *Klassenmethoden*, die auf Klassen aufgerufen werden können, beschrieben werden. Es existiert somit eine zur primären Klassenhierarchie duale Hierarchie aus Metaklassen (siehe Abbildung 3.12). Die oben angeführte Methode `new` ist ein wichtiges Beispiel für eine Klassenmethode.

Ebenfalls in Metaklassen möglich ist die Definition von Klassenvariablen, die dann von allen Instanzen der zur Metaklasse gehörenden Klasse gemeinsam benutzt werden können.

Klassenmethoden und Klassenvariablen können in bestimmten Situationen sehr nützlich sein (beispielsweise zur Realisierung des *Singleton*-Entwurfsmusters [GHJV95]), so daß sie in Smalltalk-Anwendungen recht häufig eingesetzt werden. Wenn wir Smalltalk-Code, der von diesen Konzepten Gebrauch macht, nach Java umsetzen wollen, so müssen wir die Hierarchie von Metaklassen,

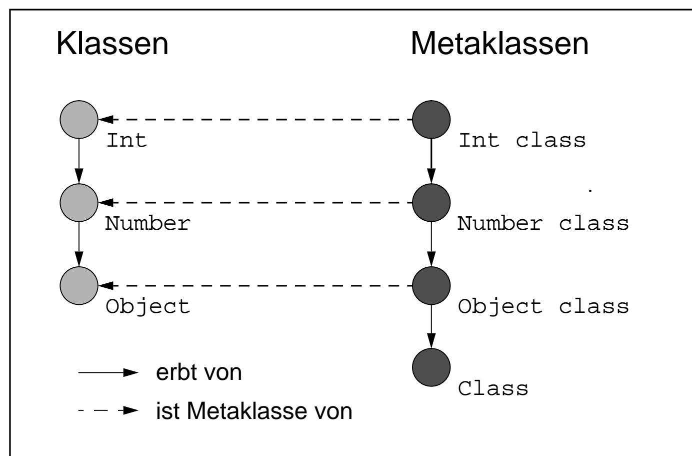

Abbildung 3.12: Die dualen Klassenhierarchien: Klassen und Metaklassen.

die den Klassen der Smalltalk-Anwendung zugrundeliegt, ebenfalls nach Java übertragen. In diesem Zusammenhang stellen sich einige subtile Probleme, auf die in [EK98] detailliert eingegangen wird.

Eng verwandt mit dem Konzept der Metaklassen sind die Fähigkeiten Smalltalks zur *Introspektion*. Darunter versteht man die Möglichkeit, im Programmcode Laufzeitinformationen über die Klasse und die Methoden und Variablen eines Objektes abzufragen und im Programmcode zu verwenden. In Smalltalk implementiert Methoden zur Introspektion mit Hilfe der Metaklassen. Einfache Beispiele für Smalltalk-Methoden zur Introspektion sind in den folgenden Ausdrücken gegeben: `anObject class` zur Bestimmung der Klasse des Objektes `anObject`, `aClass subClasses` zur Ermittlung der Unterklassen der Klasse `aClass` und `anObject perform: #aMethod` zum Aufruf der Methode `aMethod` des Objekts `anObject`.

In Java können wir solche Methoden entweder bereitstellen, indem wir diese geanu wie in Smalltalk mit Hilfe von Metaklassen implementieren, oder in dem wir dazu Javas *Reflection*-API [Sun97] verwenden, die ähnliche Methoden bereitstellt.

3.8 Bemerkungen

Wir haben in diesem Kapitel gesehen, daß die Programmiersprachen Smalltalk und Java zahlreiche Gemeinsamkeiten aufweisen. Dies erleichtert uns die automatisierte Umsetzung von Smalltalk-Anwendungen nach Java sehr. Insbesondere können wir den Code *innerhalb* einzelner Methoden relativ einfach umsetzen. Probleme bereiten uns lediglich die Unterschiede bzgl. Typisierung und Polymorphie aus Abschnitt 3.6.

In den folgenden Kapiteln 4 und 5 werden wir uns daher speziell diesem Themenbereich widmen und verschiedene Ansätze zur Umsetzung dieser Unterschiede erarbeiten. Dennoch bleibt die grundsätzliche Vorgehensweise zur Um-

setzung der Anwendung stets gleich und besteht aus den Arbeitschritten, die wir bereits in Abschnitt 2.2.2 dargestellt haben. Für jede Klasse der ursprünglichen Smalltalk-Anwendung erzeugen wir eine passende Java-Klasse, indem wir die in der Smalltalk-Klasse enthaltenen Methoden gemäß der Überlegungen dieses Kapitels in entsprechende Java-Methoden übertragen. Die verschiedenen Ansätze unterscheiden sich dabei – wie wir sehen werden – ausschließlich

- hinsichtlich der Vererbungsbeziehungen zwischen den entstehenden Java-Klassen,
- bezüglich der Art und Weise, wie Variablen und Methoden im Java-Code deklariert werden
- ob der entstehende Code statisch typisiert ist und die Gültigkeit von Methodenaufrufen zur Übersetzungszeit überprüft werden kann,
- sowie in der Syntax der Methodenaufrufe in nach Java übersetzten Smalltalk-Methoden[7].

Die in den ersten drei Punkten aufgeführten Unterschiede sind besonders gravierend, wenn wir die Ansätze in Kapitel 4 mit dem Ansatz aus Kapitel 5 vergleichen.

[7] Hierin unterscheiden sich insbesonderen die verschiedenen Ansätze aus Kapitel 4.

Kapitel 4

Einfache Reengineering-Ansätze

Wie wir in Kapitel 3 gesehen haben, stellen die Unterschiede bezüglich Typisierung und Polymorphie die größten Hindernisse bei der Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen dar. In diesem Kapitel werden wir daher einige Ansätze vorstellen, mit deren Hilfe diese Unterschiede überwunden werden können, ohne daß dazu eingehende Analysen des ursprünglichen Smalltalk-Codes notwendig sind.

4.1 Vorüberlegungen

Wenn wir den vorliegenden Smalltalk-Code der Anwendung möglichst direkt, also ohne aufwendige Analysen und größere Umstrukturierungen in Java-Code übertragen wollen, müssen wir die in Smalltalk verwendeten Mechanismen für Typprüfung und Polymorphie in Java simulieren.

Wir werden in den folgenden Abschnitten drei Ansätze vorstellen, die dies realisieren, indem sie auf statische Typisierung verzichten und die Zulässigkeit von Methodenaufrufen erst zur Laufzeit überprüfen. Die drei Ansätze unterscheiden sich, wie wir sehen werden, dabei nur in der Realisierung von Methodenaufrufen.

Gemeinsam ist den Ansätzen, daß sie die Struktur der ursprünglichen Smalltalk-Anwendung unverändert übernehmen: Eine Klasse (bzw. eine Methode) der Smalltalk-Anwendung wird in eine entsprechende Klasse (bzw. Methode) der Java-Anwendung umgesetzt. Zu jeder Smalltalk-Klasse und jeder Smalltalk-Methode gibt es nach der Umsetzung genau ein in Java codiertes Gegenstück.

Desweiteren erzeugen alle drei Ansätze *untypisierten* Java-Code: Alle Variablen[1] im durch die Umsetzung entstehenden Java-Code werden mit dem Typ

[1] Immer, wenn wir im folgenden den allgemeinen Begriff Variable verwenden, meinen wir damit alle Arten von Variablen, die in Smalltalk– bzw. Java–Programmen vorkommen können: Globale Variablen, Klassen und Instanzvariablen, Parameter und Rückgabewerte von Methoden, lokale Variablen,...

`STObject` deklariert. Mit `STObject` bezeichnen wir diejenige Java-Klasse, die die Wurzel `Object` der ursprünglichen Smalltalk-Klassenhierarchie realisiert. Der korrekte Typ, den eine Variable während der Ausführung Programms annimmt ist jedoch nicht `STObject`, sondern er ist durch die Klasse gegeben, aus der das durch die Variable adressierte Objekt – das konkret vorliegende Objekt – erzeugt worden ist.

Wenn wir *alle* Variablen mit dem Typ `STObject` deklarieren, müssen wir Mechanismen schaffen, mit deren Hilfe wir Methodenaufrufe an das konkret vorliegende Objekte ausführen können, da der in Java vorgesehene Mechanismus zum Methodenaufruf voraussetzt, daß ein für den jeweiligen Aufruf passender Typ für die Variable bereits zur Übersetzungszeit bekannt ist.

4.2 Der Reflection-Ansatz

4.2.1 Grundidee

Der naheliegendste Ansatz besteht darin, die in Smalltalk verwendeten Mechanismen für Methodenaufrufe direkt nach Java zu übertragen. Wir werden im folgenden darstellen, wie wir einen an Smalltalk angelehnten Mechanismus für den Methodenaufruf in Java implementieren können, indem wir die *Reflection-API* des Java-Systems [Sun97] verwenden.

Den Grundbaustein für diesen Ansatz bildet die Java-Klasse `STObject`, die der Smalltalk-Klasse `Object` entspricht. `STObject` stellt einen Basismechanismus für den Methodenaufruf zur Verfügung. Dazu implementiert sie `perform`-Methoden, mit deren Hilfe wir Nachrichten an ein konkret vorliegendes Objekt, das von `STObject` abstammt, senden können.

Diese `perform`-Methoden überprüfen mit Hilfe der *Reflection-API*, ob das konkret vorliegende Objekt eine Methode zu dieser Nachricht kennt, und führt diese gegebenenfalls aus. Falls das Objekt die Methode nicht kennt, gibt es einen Laufzeitfehler. Die vollständige Implementierung dieses Mechansimus ist dem Quellcode für die Klasse `STObject` zu entnehmen, der im Anhang A beigefügt ist.

4.2.2 Beispiel

Wir illustrieren den Methodenaufruf mittels der `perform`-Methoden anhand eines Beispiels. Gegeben sei eine Klasse `STString` – die Umsetzung der Klasse `String` der Smalltalk-Bibliothek. In `STString` befindet sich die Methode `comma` – die Umsetzung des Smalltalk-Operators `,` – zur Verknüpfung von Zeichenketten. In Abbildung 4.1 ist ein Programmfragment wiedergegeben, welches zwei `STString`s mit Hilfe eines `perform`-Aufrufs zu einem neuen `STString` verknüpft. Aus diesem Beispiel ist auch ersichtlich, daß jede Variable im Programmtext (auch die in der Methodensignatur von `comma` in `STString`) als `STObject` deklariert ist.

```
public class STString extends STObject {
  public STObject comma(STObject aString) {
    return(...); // Implementierung...
  }
}

// Erzeugung zweier String-Objekte
STObject firstString = new STString("Hello");
STObject secondString = new STString(" World.");

// Verknüpfung der String-Objekte
// Dies entspricht der Smalltalk-Anweisung:
// result := firstString , secondString.
STObject result
  = firstString.perform("comma", secondString);
```

Abbildung 4.1: Aufruf einer Methode beim *Reflection*-Ansatz.

4.2.3 Konsequenzen

Der Ansatz orientiert sich sehr stark an den in Smalltalk verwendeten Mechanismen. Insbesondere verzichtet er auf statische Typisierung und die Überprüfung zulässiger Methodenaufrufe zur Übersetzungszeit. Da der *Reflection*-Ansatz Smalltalks Prinzip zur Verwendung von Polymorphie ermöglicht, sind keine Änderungen an der Struktur des ursprünglichen Smalltalk-Codes nötig – wir können diesen direkt in Java-Code nach dem Muster aus Abbildung 4.1 übersetzen. Dieser Ansatz eignet sich daher besonders gut für eine *automatische Übersetzung* von Smalltalk-Code in Java-Code.

Indem wir die Mechanismen aus Smalltalk direkt auf den Java-Code übertragen, übernehmen wir auch die in Kapitel 3 beschriebenen Nachteile der dynamischen Typisierung.

Der mit Hilfe dieses Ansatzes entstehende Code genügt zudem den in der Aufgabenstellung in Abschnitt 1.1 geforderten Eigenschaften nicht:

- Der so erzeugte Java-Code ist in dem Sinne *untypisch* für Java, daß wesentliche Eigenschaften der Programmiersprache Java keine Berücksichtigung finden: Aussagekräftige Typdeklarationen, statische Typisierung und die Nutzung von Vererbung zur Konstruktion von Schnittstellen-Hierachien gehören zu den wesentlichen Merkmalen typischen Java-Codes, werden aber vom *Reflection*-Ansatz nicht verwendet.

- Die Syntax des Methodenaufrufs unterscheidet sich wesentlich von der üblicherweise in Java verwendeten. Der Programmcode wird schwerer lesbar (und damit auch *schwerer wartbar* und *erweiterbar*, da die tatsächliche

```
public class STObject {

   // Stummel für den Operator ,
   STObject comma(STObject anObject) {
      signalMethodNotStood("comma");
   }

   // weitere Stummel...
}
```

Abbildung 4.2: Stummel in `STObject` für `comma`.

Bedeutung eines Methodenaufrufs, die sich dem Programmierer durch den Namen der gerufenen Methode repräsentiert, in die Argumentliste einer `perform`-Methode verschoben wird.

- Das Nachschlagen und Ausführen von Methoden mittels der *Reflection-API* führt zusätzlich zu Einbußen in der Performance der Applikation.[2] Untersuchungen in [EK98] haben ergeben, daß ein Methodenaufruf mittels der *Reflection-API* etwa das 200-fache der Zeit eines normalen Methodenaufrufs in Java benötigt.

4.3 Der südafrikanische Ansatz

4.3.1 Grundidee

Aufgrund der Performance-Probleme des *Reflection*-Ansatzes wurde an der Universität von Pretoria, Südafrika, ein anderer Absatz entwickelt [EK98]:

Alle in der Anwendung vorkommenden Smalltalk-Methodensignaturen werden in der Wurzelklasse `STObject` der Vererbungshierarchie in Form eines *Stummels (Stub)* definiert. Ein solcher Stummel für den Operator `,` aus der Smalltalk-Klasse `String` ist in Abbildung 4.2 in der Methode `comma` zu sehen. Dieser Stummel enthält eine Anweisung, die einen Laufzeitfehler (*"Message not understood!"*) erzeugt.

In den Java-Klassen, in deren ursprünlichen Smalltalk-Klasse eine Methode definiert wird, wird der zugehörige Stummel einfach überschrieben, so daß der entsprechende Code der Methode ausgeführt und kein *"Message not understood!"*–Fehler mehr angezeigt wird. In unserem Beispiel wird in der Klasse `STString` der Stummel `comma` aus 4.2 überschrieben.

Der Aufruf der Methoden kann dann in der für Java-Code üblichen Form erfolgen. In Abbildung 4.4 ist ein Beispiel hierfür zu finden.

[2] Die Ursachen sind sowohl in der *Reflection-API* selbst zu suchen, als auch in unserer prototypischen Implementierung der `perform`-Methoden in `STObject` (siehe Anhang A).

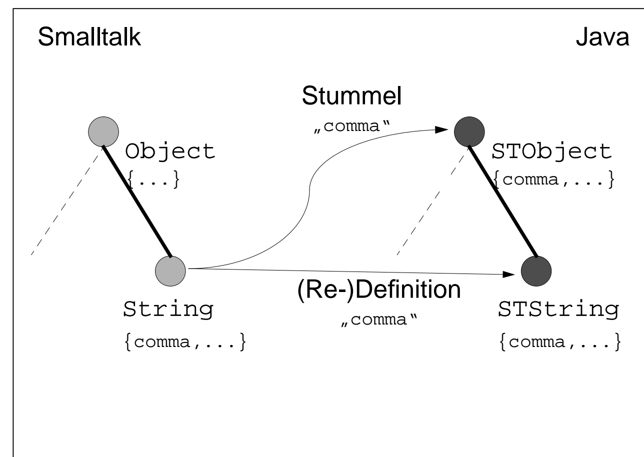

Abbildung 4.3: Überblick über den südafrikanischen Ansatz.

```
public class STString extends STObject {
   // Überschreiben des Stummels durch die eigentliche Methode
   public STObject comma(STObject aString) {
      return(...); // Implementierung...
   }
}

// Erzeugung zweier String-Objekte
STObject firstString = new STString("Hello");
STObject secondString = new STString(" World.");
// Verknüpfung der String-Objekte
// Dies entspricht der Smalltalk-Anweisung:
// result := firstString , secondString.
STObject result
   = firstString.comma(secondString);
```

Abbildung 4.4: Aufruf einer Methode beim südafrikanischen Ansatz.

Immer, wenn ein Objekt eine Methode, die auf ihm aufgerufen wird, nicht versteht, ist noch der aus `STObject` geerbte Stummel sichtbar, und löst die *"Message not understood!"*-Fehlermeldung aus.

4.3.2 Bewertung

Der südafrikanische Ansatz stützt sich bei der Realisierung des Methodenaufrufs ausschließlich auf die Mechanismen des Java-Systems. Im Vergleich mit dem *Reflection*-Ansatz entstehen daraus einige Vorteile:

- Die Performance-Probleme beim Methodenaufruf durch die *Reflection-API* werden ausgeräumt. Die Kosten für einen Methodenaufruf entsprechen denen gewöhnlicher Methodenaufrufe in Java.
- Die Lesbarkeit des entstehenden Java-Pogrammcodes nimmt deutlich zu, da die Methodenaufrufe in authentischer Java-Syntax formuliert werden.

Allerdings bietet auch der südafrikanische Ansatz keinerlei Verbesserungen hinsichtlich statischer Typisierung und Überprüfbarkeit von Methodenaufrufen zur Übersetzungzeit.

Ein zusätzlicher Nachteil des südafrikanischen Ansatzes besteht darin, daß die Wurzelklasse `STObject` durch die Definition der Stummel erheblich aufgebläht wird. `STObject` wird dadurch bei größeren Systemen sehr umfangreich und unübersichtlich. Das ursprüngliche Design der Anwendung wird verwässert, da Methoden nicht mehr nur in den Klassen bekannt sind, in denen sie auch tatsächlich definiert werden, sondern auch in `STObject`.

Bei der Pflege der Anwendung muß mit jeder Methode (genauer: mit jeder neuen Methodensignatur), die dem System hinzugefügt wird, die Wurzelklasse modifiziert werden. Ebenso müssen Änderungen an beliebigen Methodensignaturen des Systems in der Wurzelklasse nachgezogen werden. Durch diese zusätzliche Abhängigkeit des Systems von der Wurzelklasse kann der Übersetzungsvorgang der Anwendung erheblich verzögert werden, da sämtliche Klassen, die von `STObject` abhängen, neu übersetzt werden müssen.

Möglicherweise können die Nachteile der aufgeblähten Wurzelklasse aber durch ein geeignetes Werkzeug zur automatischen Verwaltung von `STObject`, welches selbständig die Signaturen in `STObject` einträgt, modifiziert oder löscht, ausgeglichen werden.

4.4 Der Interface-Ansatz

4.4.1 Grundidee

In [Bot96] wird ein weiterer Ansatz vorgeschlagen, der mit dem südafrikanischen Ansatz sehr eng verwandt ist und den wir im folgenden *Interface-Ansatz* nennen wollen.

```
public interface has_comma {
   // Signatur für den Operator ,
   public STObject comma(STObject anObject) {
      signalMethodNotStood("comma");
   }
}

public class STString
   extends STObject
   implements has_append, has_size, has_comma
      //... und weitere Interfaces für die Methoden dieser Klasse
{
   public STObject comma(STObject aString) {
      // Implementierung...
   }
}
```

Abbildung 4.5: Interface für `comma` und Fragmente aus `STString`.

```
// Erzeugung zweier String-Objekte
STObject firstString = new STString("Hello ");
STObject secondString = new STString(" World.");
// Verknüpfung der String-Objekte
// Dies entspricht der Smalltalk-Anweisung:
// result := firstString , secondString.
STObject result = ((has_comma)aString).comma(secondString);
```

Abbildung 4.6: Methodenaufruf durch *Casting* nach Interfaces.

Die Grundidee des *Interface*-Ansatzes besteht darin, für jede Signatur einer Smalltalk-Methode der Anwendung ein eigenes Java-Interface zu definieren. Jede Klasse, die eine Methode einer bestimmten Signatur definiert, muß das entsprechende Java-Interface für diese Signatur implementieren.

Ein Java-Interface für den Operator `,` aus der Smalltalk-Klasse `String` ist in Abbildung 4.5 gegeben, ebenso ein Fragment aus der Java-Klasse `STString`, die aus der Umsetzung der Smalltalk-Klasse `String` entstanden ist.

Der Methodenaufruf erfolgt dann, indem eine explizite Typkonvertierung *(Typcast)* in das zur Signatur der Methode passende Interface durchgeführt wird. Abbildung 4.6 illustriert die anhand unseres Standardbeispiels, der String-Verknüpfung mittels der `comma`-Methode.

4.4.2 Konsequenzen

Der *Interface*-Ansatz hat ähnliche Eigenschaften wie der südafrikanische Ansatz: Er orientiert sich relativ eng an Smalltalk, insbesondere verzichtet auch er auf statische Typprüfungen. Die Überprüfung von Typverträglichkeit und Methodenaufrufen geschieht zur Laufzeit durch die virtuelle Maschine des Java-Systems. Genau wie der südafrikanische Ansatz eignet sich der *Interface*-Ansatz daher besonders für die automatische Umsetzung von Smalltalk-Code in Java-Code. Die Performance des *Interface*-Ansatzes entspricht der des südafrikanischen Ansatzes.

Der *Interface*-Ansatz vermeidet die aufgeblähte Wurzelklasse des südafrikanschen Ansatzes, allerdings erzeugt er stattdessen eine Vielzahl von Java-Interfaces, die gegebenenfalls mit Hilfe eines Werkzeugs verwaltet werden müssen. Zudem stellt sich heraus, daß die für die Methodenaufrufe notwendigen Typecasts bei verketteten Methodenaufrufen Code erzeugen, der nur schwer lesbar und wartbar ist.

Gegenüber dem südafrikanischen Ansatz hat der *Interface*-Ansatz jedoch einen deutlichen Vorteil: Der Ansatz läßt sich mit Hilfe von Analysen des Programmcodes verbessern, so daß lesbarerer und wartbarer Code entsteht. Wir werden eine solche Verbesserungsmöglichkeit im nächsten Abschnitt kurz skizzieren.

4.4.3 Verbesserung

Wenn wir anhand einer Analyse des Programmcodes feststellen können, daß ein bestimmter Satz von Methoden nur in Klassen eines bestimmten Teils der Vererbungshierarchie definiert wird, dann können wir die Signaturen zu einem gemeinsamen Interface zusammenfassen (*Clustering* von Interfaces). Dieses Interface ist in der Regel durch eine bereits vorhandene Klasse des Systems gegeben.

Beispiel 4.1 In der Klassenbibliothek von Smalltalk-Systemen kommen die Methoden `append:`, `size` und einige weitere nur in *Container*-Klassen vor, also in Nachkommen von `Collection` (wie z.B. `Bag`, `Set`, `String`). Für die Signaturen dieser Methoden müssen keine separaten Java-Interfaces definiert werden, da die Typcasts für die Methodenaufrufe mit Hilfe von `STCollection`, der Umsetzung von `Collection`, durchgeführt werden können. Ein Aufruf der Methode `append:` ergibt dann den folgenden Code:

```
STObject result = ((Collection)aString).append(anotherString);
```

Allerdings löst auch diese Verbesserung nicht das Hauptproblem dieses Ansatzes: den Verzicht auf die statische Typisierbarkeit des resultierenden Java-Codes und die dadurch bedingten Vorteile.

4.5 Zusammenfassung

Die in diesem Kapitel geschilderten Ansätze orientieren sich alle sehr eng an den Mechanismen von Smalltalk. Insbesondere übernehmen sie aus Smalltalk die dynamische Typisierung, die Überprüfung von Methodenaufrufen zur Laufzeit und die Art und Weise, in der Polymorphie verwendet wird. Sie alle zeichnen sich deshalb insbesondere dadurch aus, daß sie sich für eine direkte, automatische Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen eignen.

Allerdings nützt keiner dieser Ansätze die Möglichkeiten, die Java aufgrund seines statischen Typsystems bietet. Insbesondere sind dies:

- Dokumentation der Anwendung durch Typdeklarationen.
- Hohe Zuverlässigkeit der Anwendung durch Konsistenzsicherung in Form Typprüfungen bereits zur Übersetzungszeit.

Im nächsten Kapitel werden wir daher einen Ansatz vorstellen, der diese Nachteile nicht mehr aufweist und mit dem wir authentischen, statisch typisierten Java-Code erzeugen können. Dieser Ansatz erfordert allerdings aufwendige Analysen des bestehenden Smalltalk-Codes.

Kapitel 5

Reengineering mit Hilfe von Typinferenz

In diesem Kapitel werden wir einen anderen Weg beschreiten, um die Unterschiede zwischen Smalltalk und Java bezüglich Typisierung und Polymorphie zu überwinden. Zunächst werden wir dazu mit Hilfe einer statischen Programmanalyse aus dem untypisierten Smalltalk-Code Typinformationen extrahieren. Diese Typinformationen werden wir dann dazu nutzen, um statisch typisierbaren Java-Code zu erzeugen.

5.1 Einführung

Die Ansätze aus Kapitel 4 haben die unbefriedigende Eigenschaft, daß sie eine der Grundeigenschaften von Java nicht berücksichtigen – die statische Typisierung. Der mit ihrer Hilfe entstehende Java-Code verzichtet auf aussagekräftige Typdeklarationen für Variablen und auf die Möglichkeit, Typprüfungen schon zur Übersetzungszeit des Programms vornehmen zu können.

Sowohl Typdeklarationen als auch statische Typprüfung gehören aber zu den Schlüsselkonzepten von Java. Code, der solche Konzepte nicht ausreichend berücksichtigt, verletzt die Anforderung aus Abschnitt 1.1, daß der durch das Reengineering entstehende Code typisch für die Zielsprache sein soll.

Aufgabe eines geeigneten Verfahrens für die Umsetzung von Smalltalk-Code in Java-Code muß es somit sein, aus dem Smalltalk-Code geeignete Typinformationen zu bestimmen, und diese zur Erzeugung guten Java-Codes zu verwenden.

Das Berechnen von Typinformationen für ein untypisiertes Programm bezeichnet man als *Typinferenz* [PS92]. Typinferenzsysteme haben ihren Ursprung in funktionalen Sprachen [Mil78], die später durch *Typklassen (Abstract Types)* erweitert wurden (siehe beispielsweise [HHJW93]). Eine Typklasse besteht aus einer Menge von Signaturen für Funktionen und Operatoren, die von Instanzen dieser Typklasse definiert werden müssen. Ein Typinferenzsystem für solche

Sprachen ermöglicht es, mit Hilfe von Schlußregeln die Typklassen für Parameter und Ergebnisse von beliebigen Funktionen (also die vollständige Signatur) aus ihrer Implementierung heraus zu bestimmen.

Beispiel 5.1 In der funktionalen Sprache Haskell [HJW+92] sind die Operationen der Grundrechenarten `+`, `-`, `*` ... in einer Typklasse `Num` deklariert. Konkrete Datentypen `Int`, `Float` sind Instanzen dieser Typklassen.

Zu einer Funktion `square x = x * x` kann das Haskell-System auf folgende Signatur schließen: `Num a => square :: a -> a`. Diese Signatur besagt, daß für die Funktion `square` Instanzen der Klasse `Num` als Argument und für den Rückgabewert erforderlich sind.

Typinferenzsysteme dieser Art sehen vor, daß Funktionen eindeutig *einer* Typklasse zugeordnet werden können. Es ist daher nicht erlaubt, Funktionen mit gleichem Namen in verschiedenen Klassen zu definieren, wenn diese nicht in einer Unterklassenbeziehung miteinander stehen.

Beispiel 5.2 Definiert man den Operator `*` neben `Num` auch noch in einer Typklasse `hasMult`, so kann das System für die `square`-Funktion nicht mehr entscheiden, welcher Typklasse `x` angehört. Daher liefert das System schon bei der Definition der Typklasse `hasMult` einen Fehler.

Die Situation, daß Methoden gleichen Namens in völlig unterschiedlichen Klassen (die in keiner Unterklassenbeziehung zueinander stehen) definiert sind, finden wir in Smalltalk-Systemen recht häufig, daher ist ein solches Typinferenzsystem für Smalltalk-Code ungeeignet.

Ein weitaus bedeutenderes Problem von Inferenzsystemen für abstrakte Typen zeigt jedoch das folgende Beispiel[1]:

Beispiel 5.3 Betrachten wir nochmals die Klassen zur Darstellung geometrischer Objekte (vgl. 3.1): `Point`, `Line`, `Rectangle`, ... Jede Klasse verfügt jeweils über die Methoden `move` und `draw` zum Verschieben bzw. Zeichnen des jeweiligen Objekts verfügen (siehe Abbildung 5.1). Wir nehmen wie in Beispiel 3.1 an, daß solche Basisobjekte mit Hilfe einer Klasse `Drawing` zu einer komplexen Zeichnung zusammengefügt werden können. Abbildung 5.2 zeigt nochmals die (untypisierte) Methode `add` aus `Drawing` mit der geometrische Objekte der Zeichnung hinzugefügt und anschließend angezeigt werden.

Wir interessieren uns nun in unserer Reengineering-Aufgabe dafür, welche Typen der Parameter `anObject` der Methode `add` annehmen kann. Ein Typinferenzsystem, welches uns abstrakte Typinformationen bestimmt, liefert für `anObject` alle Klassen, in denen eine Methode `draw` definiert ist – also alle Klassen für geometrische Objekte (`Point`, `Line`, `Rectangle`, ...), zusätzlich aber auch noch die ebenfalls in Abbildung 5.1 dargestellte Klasse `Cowboy`, da

[1] Dieses Problem entsteht nur, wenn man die oben geschilderte Forderung nach eindeutiger Zuordnung von Funktionsnamen auf Klassen aufgibt

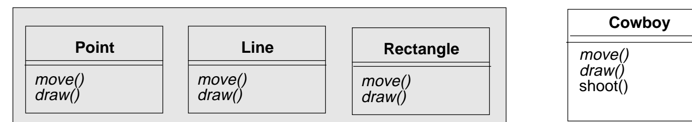

Abbildung 5.1: Geometrische Figuren und Cowboys

```
class Drawing {
   //...
   add (anObject) {
      objects.add(anObject);
      anObject.draw()
   }
   //...
}
```

Abbildung 5.2: Methode, die geometrische Figuren verwendet.

diese auch über eine Methode `draw` verfügt. Diese Klasse kann aber in der Methode `add` gar nicht sinnvoll verwendet werden, weil das Senden der Nachricht `draw` an ein `Cowboy`-Objekt eine völlig andere Bedeutung hat, als das Senden dieser Nachricht an ein geometrisches Objekt.

Um solche semantischen Fehlinformationen der Typanalyse zu vermeiden, werden wir daher sogenannte *konkrete Typinformationen (Concrete Types)* berechnen. Die konkrete Typinformation einer Variablen[2] besteht aus einer Menge von Klassen, deren Instanzen die Variable während der Laufzeit des Programmes beeinhalten kann.

In einem korrekt funktionierenden Programm können wir im Beispiel 5.3 für den Parameter `anObject` der Methode `add` ausschließen, daß er jemals zur Laufzeit ein Objekt der Klasse `Cowboy` enthält.

Wir werden daher in den folgenden Abschnitten ein Verfahren erarbeiten, das für jede Variable eines Smalltalk-Programmes eine möglichst kleine, aber vollständige Menge von Klassen aller Instanzen, die die Variable enthalten kann, berechnet.

Diese Typmengen werden wir dann in Abschnitt 5.9 in für Java geeignete Typ-

[2]Es kann sich dabei um eine Klassen- oder Instanzvariable, eine lokale Variable, einen formalen Parameter einer Methode oder einen Rückgabewert handeln.

deklarationen umsetzen, so daß wir statisch typisierbare Java-Programme erzeugen können.

5.2 Statische Programmanalyse

Die Information, welchen konkreten Typ eine Variable oder ein Ausdruck einer untypisierten Smalltalk-Anwendung annimmt, gehört zur Menge der *Laufzeitinformationen.* Solche Informationen entstehen erst während der Ausführung des entsprechenden Programmcodes.

Wenn wir uns für solche Informationen über ein Programm interessieren, stoßen wir auf ein prinzipielles Problem: Informationen, die wir während der Laufzeit eines Programmes ansammeln könnten, sind meistens unbefriedigend, weil sie ungenau sind. Dies liegt daran, daß wir im allgemeinen nicht sicherstellen können, daß alle Programmteile, die zur gewünschten Information beitragen, auch tatsächlich ausgeführt werden. Welche Programmteile ausgeführt werden, hängt unter anderem von der (unendlichen) Menge der Eingaben in das Programm ab – daraus ergibt sich in der Regel eine ebenfalls unendliche Menge von Ablaufpfaden durch das Programm (zum Beispiel durch Schleifen).

Um dieses Problem zu lösen, setzen wir die Technik der *statischen Programmanalyse* ein. Wir versuchen dabei, solche Informationen, die eigentlich zur Laufzeit anfallen, zu berechnen, ohne das betreffende Programm auszuführen, indem wir die Semantik und die innere Struktur des Programmes analysieren und aus dieser heraus Schlüsse über sein Laufzeitverhalten zu ziehen.

Zusätzlich verwenden wir eine vereinfachende, abstrakte Sicht auf die Laufzeitinformationen des Programmes, indem wir unsere Analyse beispielsweise mit abstrakten Werten aus einem endlichen Wertebereich für die Programmvariablen anstelle mit konkreter Werte und unendlich großem Wertebereich durchführen. Durch diese abstrahierende Perspektive auf die Berechnungen des Programmes ist eine Reduktion der Komplexität möglich. Wir sprechen dann von *abstrakter Interpretation* [CC77]. Abstrakte Interpretation liefert eine Aproximation an das tatsächliche Laufzeitverhalten eines Programms unter einem bestimmten Blickwinkel. Ein wesentliche Forderung dabei ist die Zuverlässigkeit dieser Approximation. Informationen über das Programm, die wir durch die Analyse gewonnen haben, dürfen durch keinen möglichen echten Programablauf widerlegt werden. Um eine solche abstrakte Interpretation durchführen zu können, muß die Semantik der Programmanweisungen in Bezug auf diese abstrakten Werte angegeben werden.

Konkret berechnen wir die abstrahierten Informationen über das Laufzeitverhalten eines Programmes mit Hilfe der Modellierung seines Informationsflusses (*Daten- und Kontrollfluß*) in Form von Flußgraphen und eines zugehörigen mathematischen Rahmens. Diesen Rahmen werden wir im nächsten Abschnitt kurz vorstellen – eine detailliertere Beschreibung dieser Konzepte findet sich in [MR90] und [Mar97]. Diesen Rahmen werden wir dann in Abschnitt 5.4 dazu nutzen, um unser Typinferenzproblem präzise zu formulieren und zu lösen.

5.3 Theoretische Grundlagen

In diesem Abschnitt legen wir die theoretischen Grundlagen, die den Rahmen der statischen Programmanalyse bilden.

Im Mittelpunkt unserer Analyse stehen *Flußgraphen*, die den Informationsfluß durch das Programm modellieren. Dabei repräsentieren die Knoten bestimmte Programmpunkte (Prozeduren, Anweisungen oder Ausdrücke) und die Kanten den Fluß der Informationen zwischen diesen.

Definition 5.1 *Ein* Flußgraph *ist ein Graph $G = (E, K, S)$, bestehend aus einer Menge E von Knoten (Ecken), einer Menge $K \subseteq E \times E$ von Kanten, sowie einer Menge $S \subset E$ von Startknoten.*

Sei $(e, e') \in K$. Dann ist e ein Vorgänger *von e' und e' ein* Nachfolger *von e. Jeder Startknoten $s \in S$ hat keinen Vorgänger.*

Definition 5.2 *Ein* Pfad *π in $G = (E, K, S)$ ist eine Folge von Kanten $(k_i)_{i=1...n}$, $k_i \in K$, die mit einem Knoten $e_1 \in E$ beginnt und mit einem Knoten $e_n \in E$ endet: $\pi = (e_1, e_2)(e_2, e_3) \cdots (e_n, e_{n+1})$.*

Wir verabreden weiter, daß jeder Knoten $e \in E \setminus S$ eines Flußgraphen von mindestens einem Startknoten $s \in S$ aus erreichbar sein soll. (Es gibt dann einen Pfad von s nach e.)

Beispiel 5.4 Der Kontrollfluß durch eine Prozedur eines Programms kann durch einen Kontrollflußgraphen dargestellt werden, dabei modellieren die Knoten des Graphen die einzelnen Anweisungen des Programms und die Kanten den Kontrollfluß zwischen den Anweisungen.

In Abbildung 5.4 ist ein solcher Kontrollflußgraph für die Prozedur zur Berechnung des Maximums zweier Zahlen aus Abbildung 5.3 dargestellt.

Jeder Knoten des Flußgraphen trägt Informationen aus dem (abstrakten) Wertebereich V. Wir bezeichnen die Information zum Knoten e mit $v(e)$.

Vom abstrakten Wertebereich V verlangen wir, daß er mit der *Ordnungsrelation* $\sqsubseteq$ einen *vollständigen Verband* bildet:

Definition 5.3 *Eine Halbordnung $(V, \sqsubseteq)$ heißt* vollständiger Verband[3], *wenn gilt: Jede Teilmenge M von V hat eine* größte untere Schranke (Infimum) *$inf(M) \in V$ und eine* kleinste obere Schranke (Supremum) *$sup(M) \in V$.*

[3] engl. *complete Lattice*

```
function max(a, b : Integer) : Integer;
  var res : Integer;
  begin
    if (a >= b) then
      res := a
    else
      res := b;
    max := res;
  end;
```

Abbildung 5.3: Prozedur zur Berechnung des Maximums zweier Zahlen

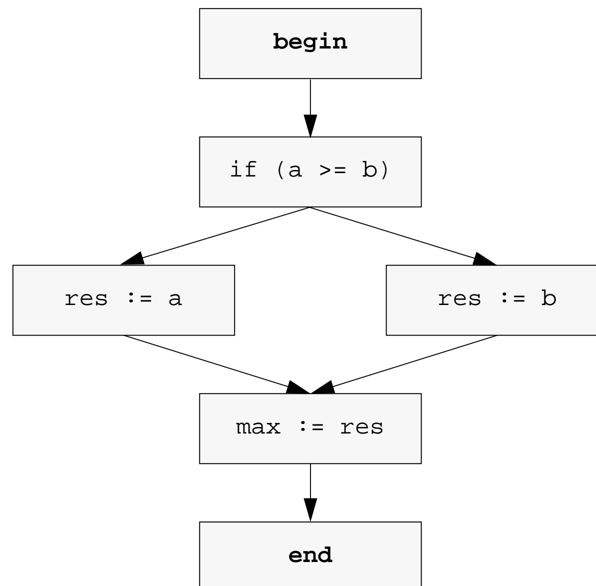

Abbildung 5.4: Kontrollflußgraph zur Prozedur aus Abb. 5.3

Dabei ist $inf(M)$ definiert durch

1. *für alle $y \in M$ gilt: $inf(M) \sqsubseteq y$ und*
2. *aus $z \sqsubseteq x$ für alle $x \in M$ folgt: $z \sqsubseteq inf(M)$.*

Entsprechend ist $sup(M)$ definiert durch

1. *für alle $y \in M$ gilt: $y \sqsubseteq sup(M)$ und*
2. *aus $x \sqsubseteq z$ für alle $x \in M$ folgt: $sup(M) \sqsubseteq z$.*

Insbesondere existieren in V auch das kleinste Element $\bot = inf(V)$ *und das* größte Element $\top = sup(V)$.

Für $inf(M)$ schreiben wir gelegentlich auch $\bigwedge M$, für $sup(M)$ $\bigvee M$.

Beispiel 5.5 Die Potenzmenge $\mathcal{P}(U)$ einer endlichen Menge U bildet mit der Mengeninklusion $\subseteq$ einen endlichen vollständigen Verband, wobei für ein Mengensystem $\mathcal{M} \subseteq \mathcal{P}(U)$ gilt:

$$\bigwedge \mathcal{M} = \bigcap_{M \in \mathcal{M}} M, \ \bigvee \mathcal{M} = \bigcup_{M \in \mathcal{M}} M$$

$$\bot = \bigcap_{M \in \mathcal{P}(U)} M \ = \ \phi, \top = \bigcup_{M \in \mathcal{P}(U)} M \ = \ U$$

.

Die Information in den Knoten verteilt sich entlang der Kanten durch den Flußgraphen. Zur Modifikation der Information während des Flusses durch den Graphen versehen wir jede Kante mit einer *Transferfunktion* $tf(k) : V \longrightarrow V$.

Um den Beitrag der Information $v(e)$ eines Knotens e zur Information $v(e')$ eines Knotens e' entlang *eines* Pfades π von e nach e' zu berechnen, benötigen wir die *Verkettung* der Transferfunktionen tf der einzelnen Kanten zur Transferfunktion $[\pi]_{tf}$ des Pfades π.

Definition 5.4 *Die* Transferfunktion $[\pi]_{tf}$ eines Pfades π *ist gegeben durch:*

$$[\pi]_{tf} = \begin{cases} id & , \textit{für } \pi = \epsilon \\ tf(k) & , \textit{für } \pi = k, k \in K \\ [k_2 \cdots k_n]_{tf} \circ tf(k_1) & , \textit{für } \pi = k_1 \dots k_n, k_i \in K \end{cases} \quad (5.1)$$

Mit diesen Begriffen können wir ein *Flußproblem* formulieren:

Gegeben sind:

- ein Flußgraph $G = (E, K, S)$,
- ein abstrakter Wertebereich V in Form eines Verbandes,
- eine Menge von Transferfunktionen $\{tf(k) : V \longrightarrow V \mid k \in K\}$ für alle Kanten des Graphen,
- eine initiale Belegung der Startknoten mit Werten $v_0(s) \in V$ für alle $s \in S$.

Gesucht sind die durch den Fluß im Graphen entstehenden Belegungen der Knoten e des Graphen:

$$v(e) = \bigvee \{[\pi]_{tf}(v_0(s)) \mid s \in S, \quad \pi \text{ ist Pfad von } s \text{ nach } e\ \} \tag{5.2}$$

$v(e)$ in Gleichung 5.2 nennt man *"Meet Over All Paths"*–Lösung $MOP(e)$ des Flußproblems für den Knoten e. [4]

$MOP(e)$ kombiniert alle Informationen, die über alle möglichen Pfade von allen Startknoten aus zu e fließen.

Da es in der Regel eine unendliche Anzahl von Pfaden in G gibt (durch Zyklen in G), können wir $MOP(e)$ nicht wie in Gleichung 5.2 dargestellt berechnen. Daher verwenden wir die sogenannte *minimale Fixpunktlösung* $MFP(e)$, die durch folgende iterative Berechnungsvorschrift gegeben ist:

$$MFP(e) = \begin{cases} v_0(e) & \text{, für } e \in S \\ \bigvee \{tf(k)(MFP(e')) \mid k = (e', e) \in K\} & \text{, sonst} \end{cases} \tag{5.3}$$

Bevor wir angeben können, unter welchen Voraussetzungen diese Lösung $MFP(e)$ berechenbar ist, benötigen wir noch folgende Definitionen:

Definition 5.5 *Eine* aufsteigende Kette (x_i) *in einer Halbordnung* $(V, \sqsubseteq)$ *ist eine Folge* $x_0, x_1, \ldots$ *für die gilt:* $x_j \sqsubseteq x_{j+1}$ *, für alle* j*.*

Definition 5.6 *Gibt es für eine aufsteigende Kette* (x_i) *ein* $n_0 \in \mathbf{N}$*, so daß für alle* $n \geq n_0$ $x_n = x_{n_0}$ *gilt, dann liegt eine sich* stabilisierende Kette *vor.*

Definition 5.7 *Eine Funktion* $f : V \longrightarrow V$ *heißt* monoton*, wenn für alle* $x, x' \in V$ *gilt: aus* $x \sqsubseteq x'$ *folgt* $f(x) \sqsubseteq f(x')$*.*

Definition 5.8 *Eine Funktion* $f : V \longrightarrow V$ *heißt* strukturerhaltend *bzgl.* $\vee$*, wenn für alle* $x, x' \in V$ *gilt:* $f(x \vee x') = f(x) \vee f(x')$*.*

[4] Je nach Anwendungsbereich, für den das Problem formuliert wird, wird in Gleichung 5.2 statt $\vee$ bisweilen $\wedge$ verwendet. Diesen Fall wollen wir hier aber nicht weiter verfolgen, aber die nachfolgenden Überlegungen dieses Abschnitts lassen sich entsprechend übertragen [MR90].

Gemäß der *"Safety and Coincidence"*-Theoreme von Kam und Ullman [KU77], [KS91] ist $MFP(e)$ berechenbar, wenn die Funktionen $tf(e)$ monoton in V sind, und sich alle aufsteigenden Ketten in V stabilisieren.[5] In diesem Fall gilt:

$$MOP(e) \sqsubseteq MFP(e), \text{ für alle } e \in E. \tag{5.4}$$

Sind die Funktionen $tf(e)$ sogar strukturerhaltend in V bzgl. $\vee$, dann gilt:

$$MOP(e) = MFP(e), \text{ für alle } e \in E. \tag{5.5}$$

Im nächsten Abschnitt werden wir diese Erkenntnisse nutzen und auf unser Typinferenzproblem übertragen, indem wir die Berechnung konkreter Typinformationen als Flußproblem formulieren.

Zuvor schließen wir diesen Abschnitt aber mit einem Algorithmus zur Berechnung der MFP-Lösung des Flußproblems ab. Der Algorithmus orientiert sich an [Mar97] und ist in Abbildung 5.5 wiedergegeben. Er arbeitet folgendermaßen:

Zunächst bekommt jeder Knoten des Graphen eine Anfangsbelegung. Für die Startknoten ist diese Belegung durch v_0 gegeben, für alle anderen Knoten verwenden wir $\perp$. Alle Knoten, die der Algorithmus noch bearbeiten muß, werden in einer *Worklist* geführt. Der Algorithmus wählt dann aus dieser Worklist immer wieder einen Knoten aus, entfernt ihn aus der Liste und berechnet dessen Information aus der der Vorgängerknoten. Wenn sich dadurch der Wert des Knotens geändert hat, müssen alle Nachfolgerknoten neu betrachtet werden (*Propagation*), weil deren Information von der des aktuellen Knotens abhängt, und werden in die Worklist aufgenommen. Wenn sich die Information im aktuellen Knoten nicht verändert hat, ist auch keine erneute Betrachtung der Nachfolgerknoten nötig. Wenn die Worklist schließlich leer ist, hat der Flußgraph einen stabilen Zustand erreicht, und die so entstandene Belegung v des Graphen ist gerade die MFP–Lösung (5.3).

Die Auswahl des als nächstes zu bearbeitenden Knotens aus der Worklist kann nach verschiedenen Strategien erfolgen, beispielsweise durch Tiefen- oder Breitensuche, oder nach Zusammenhangskomponenten.

5.4 Typinferenz als Datenflußproblem

In diesem Abschnitt werden wir die Berechnung von konkreten Typinformationen als Datenflußproblem formulieren, dabei werden wir auf die Konzepte aus Abschnitt 5.3 zurückgreifen.

Zunächst stellen wir fest, daß wir den Datenfluß durch ein Programm oder eine Methode[6] durch einen Flußgraphen G modellieren können, indem wir die Variablen und Ausdrücke des Programms als Knoten, und die Datenabhängigkeiten

[5]Dies ist in endlichen vollständigen Verbänden immer der Fall.

[6]Wir gehen zunächst davon aus, daß das Programm oder die Methode keine Methodenaufrufe enthält. Wir beschränken uns somit auf ein intraprozedurales Datenflußproblem.

Name: Worklist-Algorithmus

Eingabe: Ein Flußgraph $G = (E, K, S)$, ein vollständiger Verband V, eine Menge von Transferfunktionen tf und eine Anfangsbelegung der Startknoten v_0.

Ausgabe: Die durch den Fluß entstehende Belegung $v : E \longrightarrow V$.

```
begin
   (* Initialisierung der Startknoten *)
   forall e ∈ S do
      v(e) := v_0(e);
   od;
   (* ... und der übrigen Knoten. *)
   forall e ∈ E \ S do
      v(e) := ⊥;
   od;
   (* Initialisierung der Worklist *)
   worklist := {e | (s, e) ∈ K, s ∈ S};

   (* Iteration *)
   while worklist ≠ φ do
      (* Wähle einen Knoten aus der Worklist *)
      let e ∈ worklist ;
      worklist := worklist \e;
      X := ⋁ {tf(k)(v(e')) | k = (e', e) ∈ K};
      if X ≠ v(e) then
         v(e) := X;
         (* v(e) hat sich verändert, also müssen alle Nachfolger
         von e neu betrachtet werden *)
         worklist := worlist ∪ {e' | (e, e') ∈ K};
      fi;
   od;
end.
```

Abbildung 5.5: Worklist–Algorithmus zur Berechnung der MFP-Lösung

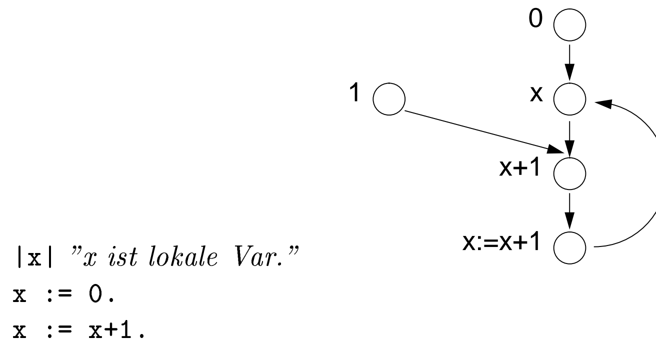

Abbildung 5.6: Datenflußgraph für einige Smalltalk-Anweisungen

zwischen den Ausdrücken als Kanten darstellen. Datenabhängigkeiten bestehen dann, wenn Werte zwischen den Ausdrücken fließen. Abbildung 5.6 illustriert dies für ein Smalltalk-Programmstück.

Gemäß Abschnitt 5.2 abstrahieren wir von den konkreten Werten der Variablen und Ausdrücke, da wir uns nur für deren Typen interessieren. Wir ordnen daher jedem Ausdruck (und damit jedem Knoten des Graphen) eine Typmenge zu, die diejenigen Klassen der Anwendung enthält, deren Instanzen mögliche Werte des Ausdrucks darstellen. Wir bezeichnen diese Typmenge als *konkreten Typ* des Ausdrucks.[7] Wenn wir die Menge aller Klassen der Anwendung mit U bezeichnen, dann bildet $V = \mathcal{P}(U)$ unseren abstrakten Wertebereich, also die Menge, die alle theoretisch möglichen konkreten Typen des Programmes enthält. Wie wir in Beispiel 5.5 gesehen haben, ist V ein vollständiger Verband bzgl. der Mengeninklusion $\subseteq$ bzw. der Vereinigung $\cup$.

Für die Transferfunktionen $tf(k)$ wählen wir für alle Kanten die identische Abbildung id, da mit den Werten auch die Typinformationen unverändert durch die Ausdrücke fließen. Die identische Abbildung id über V ist sowohl monoton als auch strukturerhaltend bzgl. $\subseteq$.

Wir wollen nun, ausgehend von einer geeigneten Ausgangsbelegung des Graphen für die Startknoten, die konkreten Typmengen aller Knoten berechnen. Im allgemeinen stellen die Startknoten des Graphen *literale Ausdrücke* dar, für die eine geeignete Typinformation bzw. Anfangsbelegung v_0 unmittelbar abgeleitet werden kann. Im Graphen der Abbildung 5.7 besteht diese Anfangsbelegung darin, den Knoten für die Konstanten 0 und 2.5 die Typmengen {`Int`} und {`Float`} zuzuordnen.

Die konkreten Typen der einzelnen Ausdrücke sind gerade die Typmengen, die durch die MOP-Lösung (5.2) des Datenflußproblems mit den eben dargestellten Komponenten G, V, tf und v_0 gegeben sind. Gemäß Abschnitt 5.3 können wir diese durch die MFP-Lösung (5.3) ersetzen, und mit Hilfe des

[7] Wenn ein Ausdruck a vom konkreten Typ T_1 ist, dann ist a auch vom konkreten Typ T_2, falls $T_1 \subseteq T_2$.

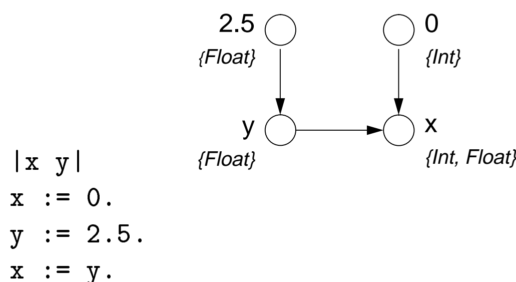

Abbildung 5.7: Datenflußgraph mit konkreten Typen

Worklist-Algorithmus aus Abbildung 5.5 berechnen. Das Ergebnis einer solchen Typinferenz, also die Belegung des Flußgraphen nach Anwendung des Worklist-Algorithmus, ist in Abbildung 5.7 zu sehen.

Noch zu klären bleibt uns nun, wann Daten (und damit Typinformationen) zwischen den einzelnen Ausdrücken des Programms fließen, d.h. zwischen welchen Knoten Kanten in den Flußgraphen eingefügt werden müssen. In objektorientierten Programmiersprachen lassen sich folgende allgemeine Regeln dafür angeben [Age94]:

- *Zuweisungen* erzeugen einen Datenfluß vom Ausdruck auf der rechten Seite zur Zielvariablen auf der linken Seite.
- *Variablenzugriffe* erzeugen einen Datenfluß von der Variable, auf die zugegriffen wird, in den umgebenden Ausdruck, in dem der Variablenzugriff verwendet wird.
- *Methodenaufrufe* erzeugen jeweils Datenflüsse von den konkreten Ausdrücken für die Argumente zu den formalen Parametern der aufgerufenen Methode, sowie vom Rückgabewert der Methode in den umgebenden Ausdruck des Methodenaufrufes.

Zusammenfassend besteht das Grundschema unseres Typinferenzverfahrens aus folgenden Schritten:

1. Erzeuge für alle Variablen und Ausdrücke des Programmes Knoten in einem Datenflußgraphen.
2. Verbinde die Knoten der Ausdrücke, zwischen denen ein Datenfluß besteht, durch Kanten.
3. Belege die Startknoten des Flußgraphen, die literale Ausdrücke enthalten, mit passenden Typinformationen.
4. Löse das Datenflußproblem mit dem Worklist-Algorithmus aus Abbildung 5.5, mit dessen Hilfe die Typinformationen vollständig durch den Flußgraphen propagiert werden.

In der Literatur (beispielsweise [PS91], [OPS92], [Age94]) wird in diesem Zusammenhang bisweilen auch der Begriff *Constraint* verwendet. Durch einen Datenfluß entsteht immer auch ein Constraint, also eine Bedingung, für den konkreten Typ eines Ausdrucks. Beispielsweise entsteht aus einer Zuweisung `var := expr` folgende Bedingung für die Typmengen $T(\texttt{var})$, $T(\texttt{expr})$ von `var` und `expr`: $T(\texttt{var}) \subseteq T(\texttt{expr})$.

Die Lösung unseres Datenflußproblems entspricht der Lösung eines Systems von Constraints, da wir jedes Constraint in Form einer Kante im Flußgraphen berücksichtigen.

In unseren bisherigen Überlegungen haben wir Methodenaufrufe, also Datenflüsse über Methoden- bzw. Prozedurgrenzen hinweg, noch nicht einbezogen, dies werden wir im folgenden Abschnitt nachholen.

5.5 Methodenaufrufe und Typinferenz – Grundalgorithmus

Sobald ein Programm polymorphe Methodenaufrufe enthält, gestaltet sich die datenflußbasierte Typinferenz etwas komplizierter. Wir werden in diesem und den nächsten Abschnitten untersuchen, welche Probleme im Zusammenhang mit Methodenaufrufen entstehen, und wie wir diese lösen können.

Zunächst führen wir dazu *Schablonen (Templates)* zur Modellierung des Datenflusses innerhalb einer Methode ein. Wir betrachten dazu den Flußgraphen G_m, der gemäß unseres Verfahrens aus Abschnitt 5.4 den Datenfluß für eine einzelne Methode m modelliert. In eine Schablone für m übernehmen wir nun alle Knoten, die formale Parameter, Rückgabewerte, lokale Variablen und sonstige Ausdrücke repräsentieren. Knoten, die Klassenvariablen, Instanzvariablen und globale Variable darstellen, nehmen wir dagegen nicht in die Schablone mit auf. Wir vervollständigen die Schablone nun um alle die Kanten aus G_m, die zur Darstellung des Datenfluß zwischen ihren Knoten erforderlich sind.

Mit Hilfe dieser Schablonen für die Methoden können wir nun Datenflüsse über Methodengrenzen hinweg modellieren, indem wir im Falle eines Aufrufs einer Methode m die Aufrufstelle mit allen passenden Schablonen[8] für m in folgender Weise verbinden:

- die Knoten für die Ausdrücke der konkreten Argumente des Aufrufs mit den Knoten der formalen Parameter der Schablonen,
- die Knoten für den Rückgabewert in den Schablonen mit dem Knoten für den umgebenden Ausruck, in dem der Aufruf stattfindet.

Abbildung 5.8 zeigt dies für den Aufruf einer Methode zur Berechnung des Maximums zweier Zahlen. Zusätzlich sind dann noch Kanten erforderlich, die

[8] Wir werden gleich sehen, daß in der Tat für *einen* Methodenaufruf *mehrere* Schablonen in der geschilderten Weise in den Flußgraphen eingebunden werden

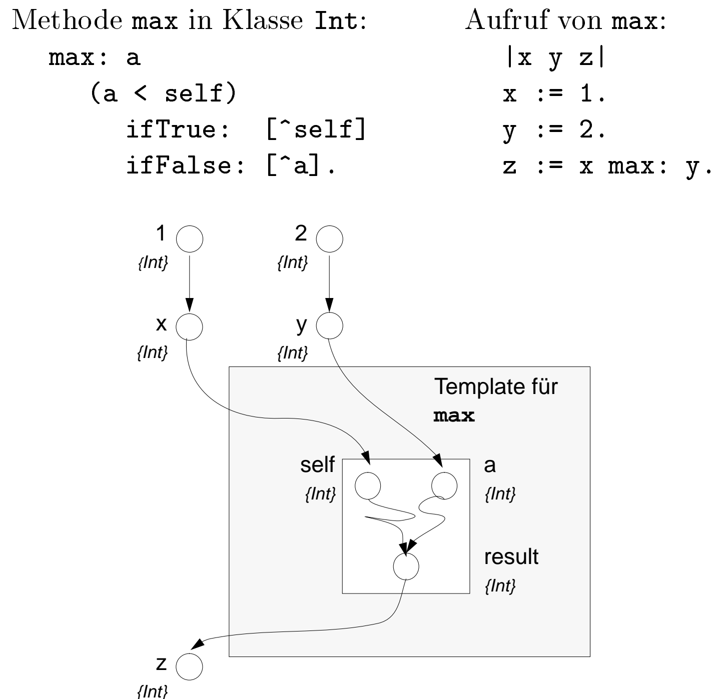

Abbildung 5.8: Aufruf von `max`

den Datenfluß zwischen den Knoten der Schablonen einerseits und den Knoten für Instanzvariable und globale Variable andrerseits modellieren.

Wir können nun den Grundalgorithmus von Palsberg für die Typinferenz [PS91] formulieren:

- *Löse Vererbungsbeziehungen auf.* Dies tun wir, indem wir die Implementierungen der Methoden einer Klasse in ihre Unterklassen kopieren. Existiert dort bereits eine Implementierung der Methode, weil sie dort redefiniert wurde, so benennen wir die Kopie aus der Oberklasse um. Aufrufe von Methoden aus der Oberklasse (`super m: a`) werden durch Aufrufe dieser umbenannten Methoden ersetzt.

- *Erstelle den Flußgraphen und berechne die MFP-Lösung des zugehörigen Flußproblems.* Wir beginnen dazu bei einer ausgezeichneten Methode, die den Start der Anwendung ermöglicht, und bauen Schritt für Schritt aus den Ausdrücken des Programms den Flußgraphen auf (siehe Abschnitt 5.4).

  Wenn wir auf einen Methodenaufruf der Form `var := recv m: arg` stoßen, erzeugen wir für *jede* Methode mit der Signatur `m:` in *jeder* Klasse aus der Typmenge T(`recv`) jeweils eine Schablone. Falls eine der Schablonen schon aus der Analyse bereits betrachteter Methodenaufrufe existiert, verwenden wir diese. Diese Schablonen verbinden wir gemäß des obenstehenden Schemas mit dem Graphen. Wir benötigen daher *während* der

Konstruktion des Flußgraphen immer schon möglichst vollständige Typmengen für die Ausdrücke, insbesondere für $T($`recv`$)$. Deshalb ziehen wir nun (anders als bei der Grundform des Worklist-Algorithmus aus Abbildung 5.5) die Typinformationen immer sofort nach, sobald neue Kanten zum Graphen hinzukommen.

Wenn sich im Verlauf des Algorithmus die Typmenge $T($`recv`$)$ ändert, müssen wir weitere Schablonen für die neu hinzugekommenen Elemente in $T($`recv`$)$ einbinden, d.h. es kommen weitere Kanten zum Graphen hinzu. Dementsprechend müssen wir auch wieder Typinformationen durch den Graphen propagieren. Insgesamt fügen wir also ständig neue Kanten zum Graphen hinzu, ziehen die Flußinformation nach, worauf wieder neue Kanten erforderlich werden, usw... Dies wiederholen wir solange, bis alle potentiell aufgerufenen Methoden der Anwendung in den Flußgraphen eingegangen sind und sich die Informationen in den Knoten stabilisiert haben. Wir realisieren dies durch eine entsprechend erweiterte Form des Worklist-Algorithmus.

Wir werden nun diesen Algorithmus kurz bewerten, dabei betrachten wir folgende Kriterien:

Korrektheit: Unser Typinferenz-Verfahren arbeitet korrekt, wenn es stets *vollständige* Typmengen für die Ausdrücke des Programmes liefert. Es darf also nicht vorkommen, daß ein Wert eines Ausdrucks während eines tatsächlichen Programmablaufs einer Instanz einer Klasse entspricht, die nicht in der berechneten Typmenge enthalten ist.

Die Korrektheit des Verfahrens ist gewährleistet, weil *alle* möglichen Datenflüsse im Graphen berücksichtigt werden. Es können so keine Typinformationen verloren gehen.

Genauigkeit: Wenn unser Algorithmus zu große Typmengen berechnet, diese also Klassen enthalten, die zur Laufzeit in den entsprechenden Ausdrücken gar keine Rolle spielen, dann bezeichnen wir das Inferenzverfahren als ungenau.

Es gibt Situationen, in denen der Grundalgorithmus ungenau ist. Ein Beispiel hierfür ist in Abbildung 5.9 angegeben. In der Klasse `Test` gibt es eine Methode `maxof:and:`, die das Maximum zweier Zahlen berechnet. Sie ist polymorph verwendbar: Ihre Argumente können sowohl beide vom Typ `Int` als auch beide vom Typ `Float` sein.

Betrachten wir nun die Methode `main`. Nach dem ersten Aufruf von `maxof:and:` mit `Int`-Argumenten ist die für das Ergebnis `x` berechnete Typmenge {`Int`} noch genau. Da für den zweiten Aufruf aber dieselbe Schablone für `maxof:and:` verwendet wird, obwohl jetzt `Float`-Argumente vorliegen, vermischen sich die Typen. Sowohl für `x` als auch für `y` wird nun eine ungenaue Typmenge {`Int, Float`} berechnet.

```
Klasse Test:
  maxof: a and: b
    (a < b) ifTrue: [^b] ifFalse: [^a].

  main
    |x y|
    x := self maxof: 1   and: 2.
    y := self maxof: 2.5 and: 3.5.
```

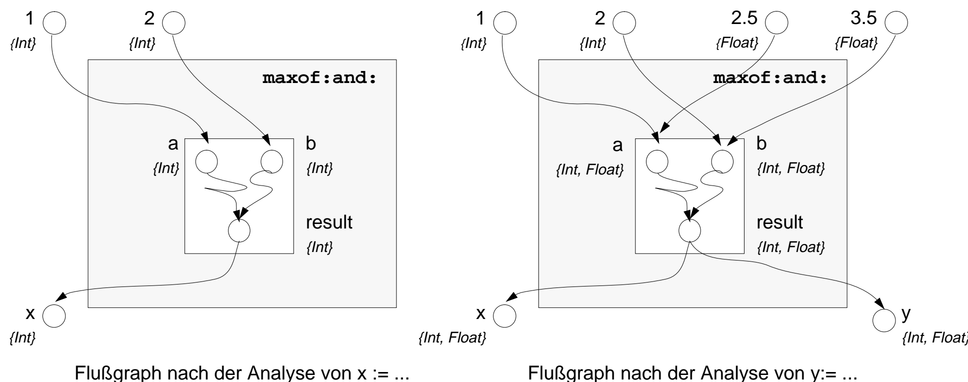

Abbildung 5.9: Berechnung ungenauer Typmengen für `x` und `y`

Effizienz: Der Algorithmus ist nicht effizient, da an vielen Stellen durch die Expansion der Vererbung Methoden mehrfach analysiert werden müssen.

Wir stellen aber fest, daß die Expansion der Vererbung in vielen Situationen die Genauigkeit der Typinferenz verbessert. Betrachten wir dazu das Beispiel aus Abbildung 5.10. Gegeben sei eine in der Klasse `Number` definierte Methode `max:`, die eine gemeinsame Oberklasse von `Int` und `Float` ist. Durch die Expansion der Vererbung bekommen die Klassen `Int` und `Float` jeweils eine Kopie von `max:`. Unser Typinferenzverfahren, angewandt auf die Methode `main`, verwendet folglich verschiedene Schablonen für diese Kopien. Daher vermischen sich hier die Typen der Argumente des `Int`- und des `Float`-Aufrufs nicht und es können präzise Typmengen für `x` und `y` berechnet werden.

5.6 Verbesserungen des Grundalgorithmus

Wir haben in Abschnitt 5.5 gesehen, daß der Grundalgorithmus in bestimmten Fällen unpräzise Typmengen berechnet, da die Typinformationen für verschiedene Aufrufe einer Methode vermischt werden.

Eine solche Vermischung von Flußinformationen aus verschiedenen Aufrufsituationen ist ein typisches, altbekanntes Problem in der *interprozeduralen Datenflußanalyse*. Es existieren eine Reihe von Verfahren, um eine solche Vermi-

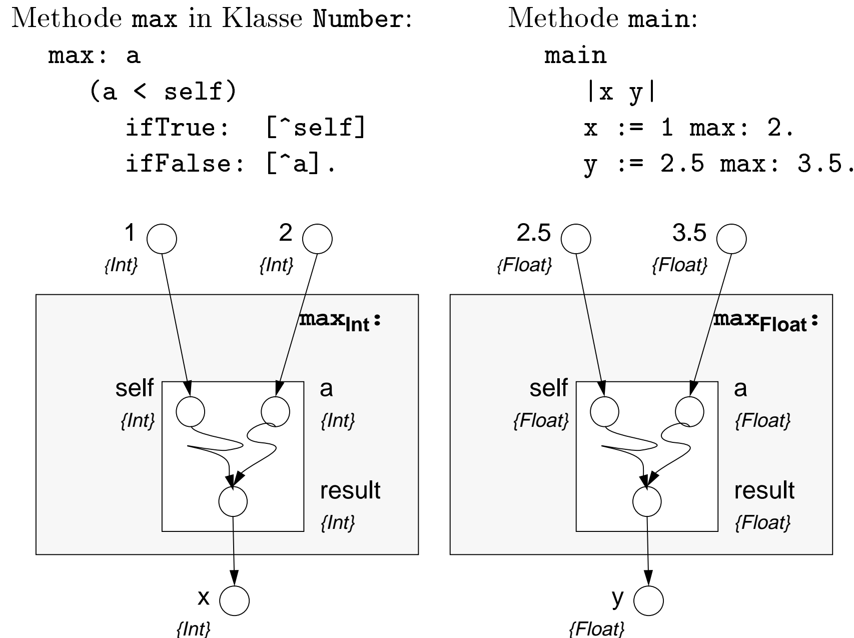

Abbildung 5.10: Genauere Typmengen durch Expansion von Vererbung

schung von Flußinformationen aus verschiedenen Aufrufsituationen zu vermeiden [Mar97].

Das einfachste Verfahren besteht im *Inlining* von Prozeduraufrufen. Dabei wird der Code der aufgerufenen Prozeduren direkt an die Stellen der Prozeduraufrufe in den Code des Programmes eingefügt. Dadurch entsteht ein prozedurloses Programm, welches mit den Methoden der intraprozeduralen Datenflußanalyse bearbeitet werden kann. Inlining funktioniert allerdings nur für Programme, die keine rekursiven Prozeduraufrufe enthalten.

Für die Analyse objekt-orientierter Programme mit polymorphen Methodenaufrufen, bei denen erst zur Laufzeit feststeht, welche Methode welcher Klasse tatsächlich gerufen wird, ist Inlining nicht durchführbar, da unklar ist, aus welcher Methode der Code, der an der Aufrufstelle eingefügt werden soll, zu entnehmen ist.

Andere Verfahren benützen die *Historie* der Prozeduraufrufe in Form eines statischen Aufrufgraphen (*Call Graph*), um verschiedene Aufrufsituationen unabhängig von einander zu analysieren, und so eine Vermischung der Flußinformationen zu vermeiden. Allerdings lassen sich diese Verfahren ebenfalls schlecht auf objekt-orientierte Programme übertragen, da die Konstruktion eines Aufrufgraphen wegen der polymorphen Methodenaufrufe nur schwer möglich ist.

In [OPS92] wird daher eine Verbesserung des Grundalgorithmus vorgeschlagen, der versucht, die Inlining-Technik auf objekt-orientierte Programme zu übertragen. Dieses erweiterte Verfahren verwendet für jeden Methodenaufruf[9] des

[9] jede syntaktische Aufrufstelle im Programmcode

Programms eine komplett *neue* Menge von Schablonen und verknüpft nur diese neu erzeugten Schablonen mit der Aufrufstelle.[10]

Im Beispiel aus Abbildung 5.9 wird also für die beiden `max:`-Aufrufe in der Methode `main` jeweils eine eigene Schablone verwendet und so die Vermischung der `Int`- und `Float`-Typinformationen vermieden.

In [OPS92] ist dieses Verfahren mit Hilfe von semantik-erhaltenden Quelltext-transformationen formuliert, die auf das zu analysierende Programm angewandt werden: Gegeben sei ein Programm, das n Aufrufstellen $s_1, \ldots, s_n$ einer Methode mit Signatur m enthält, die in k Klassen vorliegt. Die Quelltexttransformation erzeugt nun in jeder der k Klassen n Kopien der Methode $m : m_1, \ldots, m_n$. Der Aufruf von m an der i-ten Aufrufstelle s_i wird durch einen Aufruf von m_i ersetzt. Auf das so transformierte Programm kann dann der Grundalgorithmus angewandt werden. Wegen dieser Expansion des Programmes ist das erweiterte Verfahren auch unter dem Namen *1-Level-Expansion* bekannt.

Allerdings bietet auch dieses erweiterte Verfahren nur geringfügige Verbesserungen. Die Vermischung der Typinformationen für verschiedene Aufrufsituationen kann nur dann vermieden werden, wenn es im Programm keine polymorphen *Aufruffolgen* gibt. Schon wenn eine polymorph aufgerufene Methode eine andere Methode ebenfalls polymorph aufruft, werden wieder Typinformationen vermischt.

Ein Beispiel hierfür ist in Abbildung 5.11 angegeben: Für die Methode `square:` selbst werden zwar für `Int` und `Float` separate Schablonen verwendet, weil die Aufrufe von zwei verschiedenen Stellen aus erfolgen. Der Aufruf von `mult:with:` in `square:` führt aber zu einer Vermischung von Typinformationen. Es wird hier nämlich eine gemeinsame Schablone für `Int` und `Float` verwendet, da die Aufrufe von derselben Stelle aus stattfinden. Für `x` und `y` werden daher ungenaue Typmengen berechnet.

Sollen auch tiefere polymorphe Aufrufketten noch präzise analysiert werden, können wir den Algorithmus verallgemeinern, indem wir mehrere Expansionstransformationen nacheinander durchführen. Jede weitere Expansion erhöht die Länge der präzise zu analysierenden polymorphen Aufruffolgen um eins. Allgemein: p-fache Expansion *(p-Level-Expansion)* ermöglicht es, polymorphe Aufruffolgen der Länge p präzise zu analysieren.

Allerdings quadriert sich mit jeder Expansion die Größe des zu analysierenden Programmes. Experimente [PS94] mit einfachen, aus wenigen Anweisungen bestehenden Smalltalk-Programmen haben ergeben, daß bereits Expansionen mit $p \geq 2$ kaum mehr durchführbar sind.[11] Längere polymorphe Aufrufketten (zum Beispiel solche der Länge 4) kommen jedoch in realen Smalltalk-Anwendungen häufig vor [Age94].

---

[10] Man spricht dabei auch vom Duplizieren bzw. *Cloning* von Schablonen. Dagegen verwendet die Grundform des Algorithmus schon vorhandene Schablonen wieder: *Sharing* von Schablonen.

[11] Dies liegt daran, daß bereits für einzelne, einfache Anweisungen wie `3+4` große Teile der Smalltalk-Bibliothek ebenfalls analysiert werden müssen.

Klasse `Test`:

```
mult: a with: b
   ^(a * b)

square: a
   ^self mult: a with: a

main
   |x y|
   x := self square: 2.
   y := self square: 2.0.
```

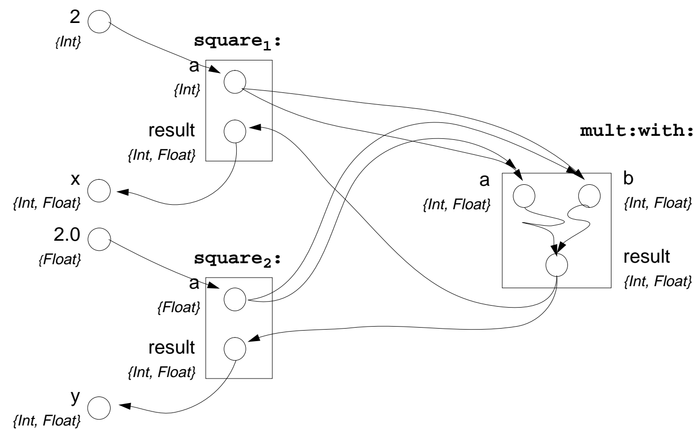

Abbildung 5.11: Berechnung ungenauer Typmengen bei *1-Level-Expansion*.

Eine Weiterentwicklung des Verfahrens ist in [PC94] beschrieben. Ziel dieses Verfahrens ist es, eine erhöhte Genauigkeit bei der Typinferenz zu erzielen, ohne dazu *alle* Schablonen duplizieren zu müssen. Das Verfahren berechnet dazu die Typinformationen mit Hilfe mehrerer Durchläufe *(Iterationen)* durch das Programm.

Die erste Iteration besteht einfach darin, den Palsberg'schen Grundalgorithmus auf das Programm anzuwenden. Es wird also genau eine Schablone pro Methode angelegt (nach Expansion der Vererbung) und von allen Aufrufstellen im Programm gemeinsam verwendet. In folgenden Iterationen versucht das Verfahren mit Hilfe der Typmengen aus der jeweils vorangehenden Iteration zu entscheiden, ob Schablonen dupliziert werden müssen, um Vermischungen der Typinformationen zu vermeiden.

Zwei Aufrufe `recv1 max: arg1` und `recv2 max: arg2` dürfen nämlich in der p-ten Iteration nur dann eine gemeinsame Schablone nutzen, wenn für die Typmengen T_{p-1} der vorangehenden Iteration folgende Bedingung gilt:

$$T_{p-1}(\texttt{recv1}) = T_{p-1}(\texttt{recv2}) \text{ und } T_{p-1}(\texttt{arg1}) = T_{p-1}(\texttt{arg2})$$

Nach p Iterationen entspricht die Genauigkeit dieses Verfahrens der des p-Level-Expansion-Verfahrens, ohne allerdings dessen Aufwand zu erfordern.

In [Age94] werden jedoch einige Nachteile dieses Verfahrens genannt:

- Es ist unklar, wie p bestimmt werden kann, so daß das Verfahren für praxisrelavante Programme exakte Typmengen liefert.
- Die Implementierung des Verfahrens gestaltet sich kompliziert, da die Typmengen der jeweils vorausgehenden Iteration stets verfügbar sein müssen.
- Aufgrund der Iterationen ist das Verfahren immer noch nicht frei von redundanten Berechnungen.

Im nächsten Abschnitt werden wir daher ein Verfahren betrachten, das einen etwas anderen Ansatz verfolgt, und das sowohl konzeptionell einfach, als auch präzise und effizient ist.

5.7 Der Algorithmus von Agesen

Für die experimentelle Programmiersprache SELF[12], die viele Konzepte mit Smalltalk gemein hat, war ein leistungsfähiges Typinferenzverfahren für kon-

[12] SELF ist wie Smalltalk ein integriertes System zur explorativen Programmierung, bestehend aus einer konzeptionell kleinen Sprache, die von einer mächtigen Bibliothek und einer komfortablen Umgebung zur Unterstützung des Programmieres ergänzt wird. Die Philosophie und Konzepte zu SELF werden in [US91] und [SU95] erklärt.

krete Typen erforderlich. Daher wurden im Rahmen des SELF-Projekts zahlreiche Inferenzverfahren untersucht [Age94], unter anderem die, die wir bereits in den Abschnitten 5.5 und 5.6 betrachtet haben.

Wie wir gesehen haben, sind diese Verfahren entweder unpräzise, ineffizient oder kompliziert. Daher wurde für das SELF-System ein neues Verfahren entwickelt, das sowohl konzeptionell einfach, als auch präzise und effizient ist [Age95].

Das Verfahren baut auf Palsbergs Grundalgorithmus auf. Es unterscheidet sich von diesem nur hinsichtlich des Erzeugens und Einbindens von Schablonen bei Methodenaufrufen. Die Grundidee besteht darin, daß immer genau dann neue Schablonen erzeugt werden *(Cloning)*, wenn Typeninformationen verschiedener Aufrufe vermischt würden – ansonsten werden bereits bestehende Schablonen wiederverwendet *(Sharing)*.

Wie können wir dies gewährleisten, ohne daß wir dazu mehrere Iterationen benötigen und auf die Typinformationen der vorausgehenden Iteration aufbauen müssen?

Betrachten wir dazu einen Methodenaufruf der Form `res := recv m: arg`. Sei $R = T(\texttt{recv})$ und $A = T(\texttt{arg})$. Wir nehmen zunächst an, daß die Mengen $R = \{r_1, \ldots, r_s\}$ und $A = \{a_1, \ldots, a_t\}$ bereits vollständig bekannt sind. (Dies ist natürlich während der Konstruktion des Typgraphen nicht der Fall, wir werden aber später sehen, daß wir auch mit zunächst unvollständigen Mengen R und A arbeiten und diese schrittweise ergänzen können, ohne daß wir bisher durchgeführte Berechnungen, die sich auf die unvollständigen Mengen stützen, verwerfen müssen.)

Um diesen Aufruf zu analysieren, berechnen wir das *kartesische Produkt* der Mengen R und A:

$$R \times A = \{(r_1, a_1), \ldots, (r_1, a_t), \ldots, (r_i, a_1), \ldots, (r_i, a_t), \ldots, (r_s, a_1), \ldots, (r_s, a_t)\}$$

Für jedes Paar $(r_i, a_j) \in R \times A$ verwenden wir eine eigene Schablone zu `m:`, verknüpfen diese mit der Aufrufstelle und beschicken sie mit den Wertepaar (r_i, a_j). Dabei verwenden wir eine vorhandene Schablone wieder, wenn für `m:` bereits eine solche für (r_i, a_i) existiert, andernfalls erzeugen wir eine neue. Von den Knoten für den Rückgabewert in jeder dieser Schablonen ziehen wir eine Kante zum Knoten für `res`. Dies ist in Abbildung 5.12 am Beispiel des Aufrufes `x := a max: b` der `max:`-Methode aus Abbildung 5.10 und fiktiver Typmengen $R = T(\texttt{a}), A = T(\texttt{b})$ illustriert.

Wir müssen jetzt noch die Frage klären, wie wir das kartesische Produkt $R \times A$ berechnen können, obwohl die beiden Typmengen R und A während der Konstruktion des Flußgraphen noch gar nicht vollständig bekannt sind. Wir nützen dazu die *Monotonie* des kartesischen Produktes:

Sei $P = R \times A$. Fügen wir nun zu R ein weiteres Element r hinzu, so kommen entsprechend neue Elemente zu P hinzu – nämlich alle Elemente aus $\{r\} \times A$ – ohne daß dabei bereits in P enthaltene Elemente verändert werden müssen.

`x := a max: b.`
$R = T(\mathtt{a}) = \{\mathtt{Int}, \mathtt{Float}\}$
$A = T(\mathtt{b}) = \{\mathtt{Int}, \mathtt{BigInt}\}$
$R \times A = \{(\mathtt{Int}, \mathtt{Int}), (\mathtt{Int}, \mathtt{BigInt}), (\mathtt{Float}, \mathtt{Int}), (\mathtt{Float}, \mathtt{BigInt})\}$

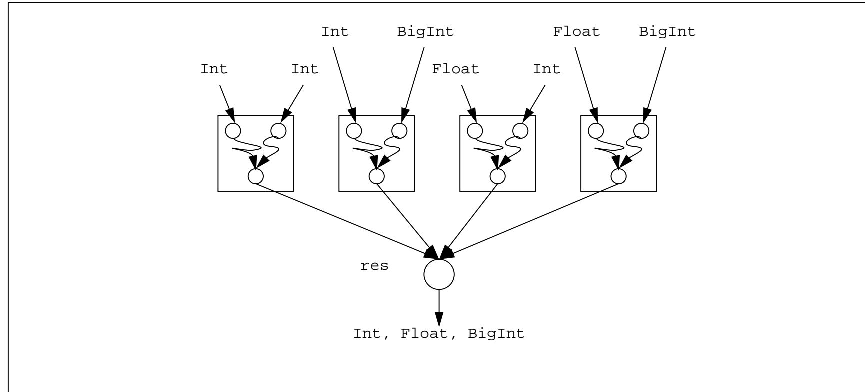

Abbildung 5.12: Schablonen für den Aufruf `res := a max: b`.

Entsprechendes gilt wegen der Symmetrie des kartesischen Produktes auch für die Hinzunahme eines neuen Elements zu A.

Wir können also, wenn im Verlauf unseres Algorithmus eine der beiden Typmengen $R = T(\mathtt{recv})$ oder $A = T(\mathtt{arg})$ durch neue Klassen ergänzt wird, einfach auch die Menge $R \times A$ entsprechend ergänzen und passende zusätzliche Schablonen für `m:` hinzunehmen. In Abbildung 5.13 ist eine solche Situation dargestellt. Die Typmenge $T(\mathtt{b})$ wurde hier um die Klasse `Rational` erweitert, dementsprechend ergänzen wir die Menge $R \times A$ (unterstrichen dargestellt) und erzeugen neue Templates.

Mit diesen Erkenntnissen können wir nun den Algorithmus von Agesen[13] zusammenhängender, allgemeiner Form angeben.

Wir benötigen dazu für jede Methode mit der Signatur m und den formalen Parametern $a_1, \ldots, a_k$ eine Tabelle zur Verwaltung ihrer Schablonen. Alle Schablonen für m werden in dieser Tabelle gespeichert, unabhängig davon, welche Aufrufstelle zu ihrer Erzeugung geführt hat. Auf die Einträge der Tabelle kann über den Schlüssel $(r, a_1, \ldots, a_k)$ zugegriffen werden.

Der Algorithmus läßt sich dann folgendermaßen formulieren:

- Konstruiere einen Datenflußgraphen nach dem Grundalgorithmus von Palsberg (siehe Abschnitt 5.5).
- Behandle dabei jeden Methodenaufruf `res := recv m:` $a_1, \ldots, a_k$ wie folgt:

[13] Wir benennen den Algorithmus hier der Einfachheit halber nach seinem Erfinder Ole Agesen, einem der Entwickler von SELF. In [Age95] wird der Name *Cartesian Product Algorithm* verwendet.

```
x := a max: b.
```

$R = T(\mathtt{a}) = \{\mathtt{Int}, \mathtt{Float}\}$

$A = T(\mathtt{b}) = \{\mathtt{Int}, \mathtt{BigInt}, \underline{\mathtt{Rational}}\}$

$R \times A = \{(\mathtt{Int}, \mathtt{Int}), (\mathtt{Int}, \mathtt{BigInt}), (\mathtt{Float}, \mathtt{Int}), (\mathtt{Float}, \mathtt{BigInt}),$
$\underline{(\mathtt{Int}, \mathtt{Rational})}, \underline{(\mathtt{Float}, \mathtt{Rational})}\}$

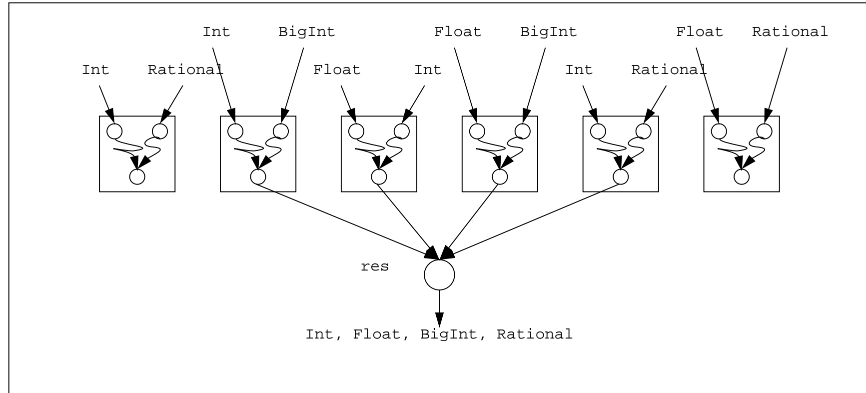

Abbildung 5.13: Schablonen für den Aufruf `res := a max: b` (Fortsetzung).

1. Erzeuge das kartesische Produkt $P = T(\mathtt{recv}) \times T(a_1) \times \ldots \times T(a_k)$ (oder erweitere es wie oben dargestellt).
2. Suche für jedes $(k+1)$-Tupel des kartesischen Produktes P in der Tabelle zu m nach einer passenden Schablone. Gibt es keine solche Schablone für das Tupel, dann erzeuge eine, füge diese der Tabelle hinzu und belege ihre Eingangsknoten mit den Elementen des Tupels.
3. Verbinde die Knoten für den Rückgabewert der Schablonen, die zu den Tupeln des kartesischen Produktes P gehören, mit dem entsprechenden Knoten für `res` aus der Aufrufstelle.

Wie wir es schon in Abschnitt 5.5 für den Grundalgorithmus von Palsberg getan haben, werden wir nun auch den Algorithmus von Agesen kurz bewerten:

Korrektheit: Die Korrektheit des Algoritmus von Agesen ist gewährleistet, weil – genau wie beim Grundalgorithmus – *alle* möglichen Datenflüsse im Graphen berücksichtigt werden und somit keine Typinformationen verloren gehen können.

Genauigkeit: Der Algorithmus ist in dem Sinne genau, daß er Programme mit beliebig tiefen polymorphen Aufrufgraphen analysieren kann, ohne daß Typinformationen vermischt werden – es werden aufgrund der Konstruktion des Algorithmus nie Argumente unterschiedlicher Typen in dieselbe Schablone propagiert. Insbesondere liefert der Algorithmus exaktere Typmengen als der Grundalgorithmus bzw. die p-Level-Expansion-Algorithmen.

Effizienz: Der Algorithmus ist effizienter als der erweiterte Grundalgorithmus, da die Codeduplikationen (bzw. das entsprechende Cloning der Schablonen) in vielen Fällen entfallen können: Immer wenn Schablonen gemeinsam genutzt werden können, ohne daß eine Vermischung von Typinformationen auftritt, tut dies der Algorithmus.

Im nächsten Abschnitt werden wir den bisher allgemein formulierten Algorithmus auf Smalltalk übertragen.

5.8 Anpassung von Agesens Algorithmus auf Smalltalk

Wir haben in den voranstehenden Abschnitten ein allgemeines Verfahren erarbeitet, mit dem wir konkrete Typinformationen aus in einer objekt-orientierten Sprache geschriebenem Programmcode extrahieren können: den Algorithmus von Agesen. Um das Verfahren praktisch nutzen zu können, muß es jedoch zunächst an die Programmierpsrache angepaßt werden, in der die zu analysierenden Programme vorliegen.

Da wir den Algorithmus für unser Reengineering-Problem nutzen wollen, das darin besteht, Smalltalk-Anwendungen nach Java zu überführen, skizzieren wir im folgenden, wie wir die spezifischen Merkmale von Smalltalk berücksichtigen müssen, um ein funktionsfähiges Verfahren zu erhalten.

Wir müssen dazu angeben, welche Datenflüsse die verschiedenen Konstrukte der Programmiersprache Smalltalk erzeugen, damit wir den der Typinferenz zugrundeliegenden Datenflußgraphen korrekt aufbauen können.

Wie wir in Kapitel 3 festgestellt haben, sind die wesentlichen Konstrukte in Smalltalk Ausdrücke, die eine der folgenden Formen annehmen:

1. Zugriffe auf Variablen bzw. Literale (siehe Abschnitt 3.2).
2. Blöcke bzw. Closures (siehe Abschnitt 3.3).
3. Methodenaufrufe *(Message Sends)*, deren Argumente jeweils Ausdrücke der Formen 1.–3. sind.
4. Zuweisungen, deren rechte Seiten jeweils aus einem der Ausdrücke 1.–3. bestehen.

In Abschnitt 5.4 (siehe Seite 51) haben wir bereits angegeben, welche Datenflüsse durch Zugriffe auf Variablen (bzw. Literale), durch Methodenaufrufe und durch Zuweisungen erzeugt werden.

Wir müssen also im folgenden noch klären, wie wir Blöcke bei der Typinferenz berücksichtigen.

```
maxof: a and: b
   (a < b)
      ifTrue: [^b]   "Block (1)"
      ifFalse: [^a]. "Block (2)"
```

Abbildung 5.14: Blöcke in der Methode `maxof:and:`.

Ausdrücke in Blöcken können auf Variablen des lexikalischen Kontextes zugreifen, in dem sie definiert sind, obwohl sie in einem anderen Kontext ausgewertet werden. Wir müssen dies während der Typinferenz entsprechend berücksichtigen. Betrachten wir dazu nochmals die Methode `maxof:and:` aus Abbildung 5.9, die wir der Übersichlichkeit halber in Abbildung 5.14 nochmals wiedergeben.

Wenn wir die Ausdrücke der Blöcke (1) bzw. (2) im Kontext der Methode `ifTrue:ifFalse:` auswerten, so müssen wir dabei auf die Variablen `a` und `b` aus dem Kontext der Methode `maxof:and:` zugreifen. Da wir jedoch für `maxof:and:` möglicherweise unterschiedliche Schablonen für `Int`– und `Float`–Typen benötigen, müssen wir einen Mechanismus nützen, der es uns ermöglicht, auf die Datenflußknoten für `a` und `b` in der jeweils *richtigen* Schablone von `maxof:and:` zuzugreifen. Wir legen dazu für die Blöcke Closure-Objekte an, die einen Verweis auf die gerade aktuelle Schablone von `maxof:and:` enthalten. Wenn wir dann die Anweisungen innerhalb des Closure-Objektes analysieren, können wir diesen Verweis dazu benützen, um auf die Variablenknoten der jeweils passenden Schablone von `maxof:and:` zuzugreifen. Diese Konstruktion entspricht dem Verfahren, welches das Smalltalk-System zur Laufzeit verwendet (siehe Abschnitt 3.3) – mit dem Unterschied, daß zur Laufzeit ein Verweis auf ein *Activation Record* statt auf eine Schablone von `maxof:and:` verwendet wird.

Unser Typinferenzverfahren berücksichtigt nun alle der oben angeführten Konstrukte der Programmiersprache Smalltalk, wir haben somit die Grundlage für eine Implementierung des Algorithmus von Agesen für Smalltalk gelegt.

Zum Abschluß dieses Abschnittes halten wir nochmals fest, daß unser Verfahren ausgehend von einer ausgezeichneten Startmethode, mit der die Ausführung der zu analysierenden Smalltalk-Anwendung gewöhnlich beginnt, alle von dieser Methode aus erreichbaren Teile der Anwendung in den Datenflußgraphen integriert und Typinformationen für alle in diesen Teilen des Systems vorhandenen Variablen berechnet. Dazu ist es erforderlich, daß *alle* von der Ausgangsmethode aus erreichbaren Teile der Anwendung im Quelltext (bzw. in Form eines AST) vorliegen.

In Smalltalk-Systemen kann dies gewährleistet werden, da auch Bibliotheken (inklusive der Standard-Klassenbibliothek) im Smalltalk-Quellcode geliefert werden. Eine Ausnahme bilden die sogenannten *Primitive*, spezielle Methoden, die Systemfunktionen der virtuellen Maschine eines Smalltalk-Systemes aufrufen. Für diese Systemfunktionen liegt kein geeigneter Smalltalk-Code vor, daher ist es erforderlich, deren Parameter und Rückgabewerte vorab anhand

der Spezifikationen der virtuellen Maschine mit geeigneten Typinformationen zu versehen. Wenn wir dies unterlassen, kann es zu fehlerhaften Ergebnissen bei der Bestimmung der Typmengen von Ausdrücken kommen, die Ergebnisse aus Primitivenaufrufen nutzen.

5.9 Nutzung der Typinformationen für das Reengineering

Wir sind nun mit Hilfe des Algorithmus von Agesen in der Lage, für jede Variable des Programms geeignete konkrete Typinformationen zu berechnen. Diese Typinformationen wollen wir im folgenden nutzen, um typisierten Java-Code zu erzeugen.

5.9.1 Vorüberlegungen

Die Typinformation zu einer Variable liegt uns in Form einer im allgemeinen mehrelementigen Typmenge vor. Wir können diese Typinformationen dazu nützen, um unseren Code mit *Typannotationen* zu versehen. Ein Beispiel für Java-Code mit Typannotationen ist in Abbildung 5.15 gegeben – es basiert auf dem Smalltalk-Code aus Abbildung 5.11.[14]

Wir beachten übrigens, daß wir für die Typannotationen in Methodendeklarationen die Typinformationen aus den verschiedenen Typinferenz-Schablonen der jeweiligen Methode zusammenführen müssen – trotzdem nützen wir die volle Genauigkeit des Typinferenzalgorithmus für die zurückgelieferten Werte aus Methodenaufrufen. Beispielsweise entstehen die Typmengen {`Int`,`Float`} für die Parameter `a` und `b` in der Methode `mult_with_` durch das Zusammenführen der Typinformationen aus der (`Int`,`Int`)– und der (`Float`,`Float`)–Schablone. Zur Bestimmung der Typmenge {`Int`} für die Variable `x` bzw. der Typmenge {`Float`} für die Variable `y` in `main` waren jedoch separate Schablonen der Methoden `square_` bzw. `mult_with_` für `Int`– und `Float`–Argumente erforderlich (siehe Seite 58).

Um reinen Java-Code zu erhalten, müssen wir die Typannotationen in Typdeklarationen übersetzen. Dazu müssen wir die Typmengen auf eine einzelne Klasse abbilden, die wir dann für die Deklaration der betreffenden Variable oder zur Erzeugung einer Methodensignatur nutzen können. In Abbildung 5.16 ist eine solche Übersetzung für unser Beispiel aus Abbildung 5.15 angegeben.

Wir werden nun ein Verfahren zur Umsetzung von Typannotationen in Typdeklarationen systematisch darstellen. Wir werden dabei jedoch feststellen, daß wir diese Umsetzung nicht vollständig automatisieren können.

[14]Für dieses Beispiel und alle weiteren im Rest dieses Kapitels werden wir Java-Syntax verwenden. Die Umsetzung von Smalltalk-Konstrukten in entsprechende Java-Konstrukte haben wir bereits in Kapitel 3 skizziert, so daß hier nur noch Fragen bezüglich der Typisierung der Programme von Interesse sind.

```
class Test {

   {Int,Float} mult_with_({Int,Float} a, {Int,Float} b) {
      return(a*b);
   }

   {Int,Float} square_({Int,Float} a) {
      return mult_with_(a, a);
   }

   {} main() {
      {Int} x;
      {Float} y;
      x := square_(2);
      y := square_(2.0);
   }

}
```

Abbildung 5.15: Java-Code mit Typ*annotationen*.

```
class Test {

   Number mult_with_(Number a, Number b) {
      return(a*b);
   }

   Number square_(Number a) {
      return mult_with_(a, a);
   }

   void main() {
      Int x;
      Float y;
      x := (Int) square_(2);
      y := (Float) square_(2.0);
   }

}
```

Abbildung 5.16: Java-Code mit Typ*deklarationen*.

```
divide: x by: y
   (x class == Float)
      ifTrue: [^x divFloat: y]   "x ist Float"
      ifFalse: [^x divInt: y].   "x ist Int"
```

Abbildung 5.17: Beispiel zur Fußnote auf Seite 68.

Zentrales Element dieser Umsetzung ist eine Abbildung d von der Menge aller in der Anwendung vorkommenden Typmengen V' in die Menge der Klassen U, aus denen die Anwendung besteht:

$$d : V' \longrightarrow U \qquad V' \subseteq V, V = \mathcal{P}(U)$$

In den Abbildungen 5.15 und 5.16 ordnet d beispielsweise der Typmenge {`Int`,`Float`} die Klasse `Number` zu.

Wenn wir diese Abbildung d geeignet konstruieren, können wir mit ihrer Hilfe jede Typannotation im mit Typinformationen versehenen Code (vgl. Abbildung 5.15) in eine geeignete Typdeklaration umformen. Wir beachten dabei, daß die Abbildung d Typannotationen stets auf die gleiche Klasse abbildet, wenn die zugrundeliegenden Typmengen gleich sind – der Programmkontext einer Typannotation spielt also bei der Umsetzung der Typinformationen keine Rolle.

Die Klasse O, die d einer Typmenge T zuordnet, muß die Gemeinsamkeiten der Klassen in T zum Ausdruck bringen. Insbesondere betrifft dies die Menge der Nachrichten, die von O verstanden werden (bzw. die dementsprechende Menge von Methoden, die in O und deren Oberklassen definiert sind).[15]

Unter diesem Aspekt liegen der Zuordnung einer Klasse O zur Typmenge T, $d : T \longmapsto O$, folgende Überlegungen zugrunde:

- Ist eine Variable `x` mit der (mehrelementigen) Typmenge T annotiert, so wird `x` bezüglich der Klassen aus T polymorph verwendet. Die Klasse O, die wir für die Deklaration von `x` nützen wollen, muß diese Polymorphie auch in Java ermöglichen. Wie wir in Kapitel 3 gesehen haben, muß sie daher gemeinsame *Oberklasse* der Klassen aus T sein[16] und alle Methoden der Klassen aus T enthalten, bezüglich derer diese Klassen polymorph verwendet werden.

[15] Wir nehmen zur Kenntnis, daß wir uns bei der Betrachtung der Gemeinsamkeiten der Klassen in T tatsächlich auf die von den Klassen akzeptierten Nachrichten (bzw. die entsprechenden Methodendefinitionen) beschränken können, da Methoden die einzigen Elemente eines Objektes sind, die nach außen hin sichtbar sind.

[16] O kann auch selbst in T enthalten sein. Dann muß O Oberklasse aller *anderen* Klassen in T sein.

- In einem korrekten Programm dürfen an `x` nur Nachrichten geschickt werden, die *alle* Klassen aus T verstehen.[17] Wir können dies zur Übersetzungszeit überprüfen, indem wir `x` als Variable eines Typs O deklarieren, der folgende Bedingung erfüllt:

$$M(O) = \bigcap_{X \in T} M(X) \tag{5.6}$$

wobei wir mit $M(X)$ die Menge aller Nachrichten, die eine Klasse X versteht (bzw. der Methoden, die in X definiert sind), bezeichnen.

Allerdings werden wir noch sehen, daß wir in bestimmtem Ausnahmefällen eine Klasse O verwenden, die die Bedingung (5.6) nicht erfüllt.

Wir geben nun Regeln zur Konstruktion der Abbildung d an. Dabei lassen wir uns von den oben stehenden Erkenntnissen leiten. Es wird sich herausstellen, daß wir bisweilen die Anwendung umstrukturieren müssen, damit wir jeder Typmenge T im Programmcode eine passende Klasse O zuordnen können.

5.9.2 Vereinfachung der Typmengen

Sowohl in der Abbildung d, als auch in den Typannotationen des Programms können die Typmengen vereinfacht werden, indem überflüssige Unterklassen aus ihnen entfernt werden.

Wir entfernen aus einer Typmenge T die überflüssigen Unterklassen (und erhalten als Resultat T') durch folgende Vorschrift:

$$T' = \{X \mid X \text{ hat keine direkte oder indirekte Oberklasse in } T\}$$

Dies können wir tun, da sich eine dadurch vereinfachte Typmenge bzgl. Bedingung (5.6) nicht anders verhält als die urpsrüngliche Typmenge, es gilt nämlich:

$$\bigcap_{X \in T'} M(X) = \bigcap_{X \in T} M(X)$$

Durch diese Vereinfachung entstehen kompaktere und übersichtlichere Typmengen.

[17] Diese Aussage ist nicht ganz richtig: Es lassen sich korrekte Smalltalk-Programme finden (insbesondere solche, in denen Introspektion verwendet wird), für die dies nicht gilt. Ein Beispiel hierfür ist in 5.17 angegeben. Für dieses Beispiel liegen für `x` und `y` jeweils die Typmengen $T = \{\texttt{Int}, \texttt{Float}\}$ vor, die Methode `divInt:` sei jedoch nur in `Int` definiert, und die Methode `divFloat:` nur in `Float`. An `x` werden in diesem Fall Nachrichten geschickt, die nicht alle Typen in T verstehen. Das in diesem Kapitel vorgestellte Verfahren funktioniert jedoch auch in diesem Fall.

5.9.3 Abbildung der Typmengen in einzelne Klassen

Abbildung einelementiger Typmengen. Einelementige Typmengen treten immer dann bei Variablen auf, wenn diese Variablen nicht polymorph verwendet werden. Einelementige Typmengen $T = \{X\}$ werden in d durch folgende, naheliegende Zuordnung berücksichtigt:

$$d : \{X\} \longmapsto X$$

Abbildung mehrelementiger Typmengen. Mehrelementige Typmengen T müssen auf eine gemeinsame Oberklasse O abgebildet werden, die in der Regel Bedingung (5.6) erfüllt.

Eine mehrelementige Typmenge tritt nur bei polymorpher Verwendung von Variablen auf. Wie wir in Kapitel 3 gesehen haben, unterstützt Smalltalk Polymorphie, ohne daß für die polymorph verwendeten Klassen eine gemeinsame Oberklasse, die die polymorph genutzten Methoden enthält, existieren muß. Aus diesem Grund können wir nicht davon ausgehen, daß für jede mehrelementige Typmenge eines annotierten Programms eine geeignete Oberklasse O existiert, mit deren Hilfe wir eine Zuordnung $d : T \longmapsto O$ konstruieren können. Gegebenenfalls müssen wir eine solche Oberklasse erst noch konstruieren (oder eine bestehende Oberklassen modifizieren) und in die Klassenhierarchie der Anwendung geeignet einbinden. Es sind somit bei der Konstruktion der Zuordnung $d : T \longmapsto O$ zwei Szenarien möglich, die wir im folgenden detailliert betrachten:

1. *Nutzung einer vorhandenen Oberklasse O.* Zur Nutzung einer bereits vorhandenen gemeinsamen Oberklasse O der Klassen in T muß diese zunächst einmal identifiziert werden.

   Dazu durchlaufen wir von jeder Klasse in T die Vererbungshierarchie der Anwendung in Richtung Wurzel[18]. Diejenige Klasse O, die dabei mit den kürzesten Pfaden von allen Klassen in T erreicht wird, ist Kandidat für die gesuchte Klasse. Ein Beispiel hierzu ist in Abbildung 5.18 zu sehen.

   Wir müssen nun überprüfen, ob O tatsächlich eine geeignete Klasse ist, die als Ersatz für die Typmenge T dienen kann:

   (a) Erfüllt diese Klasse die Bedingung (5.6), so haben wir bereits eine geeignete Klasse gefunden, und wir können O der Menge T zuordnen.

   (b) Gilt dagegen

   $$M(O) \subset \bigcap_{X \in T} M(X) \tag{5.7}$$

   so kennt O weniger Methoden, als wir erwartet haben.

[18] Wir erinnern uns daran, daß in Smalltalk die Vererbungshierarchie immer die Form eines Baumes hat.

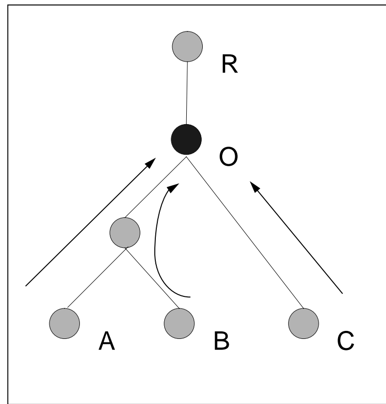

Abbildung 5.18: Gemeinsame Oberklasse zu $\{A, B, C\}$.

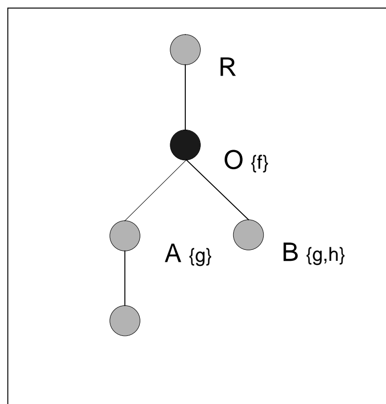

Abbildung 5.19: Methode g ist nicht in O definiert.

Abbildung 5.19 zeigt eine solche Situation:[19] In der Klasse O ist die Methode g nicht definiert, obwohl sie in den beiden Unterklassen A und B definiert ist.

Wenn ein solcher Fall vorliegt, müssen wir die Methoden, die in O "fehlen" (am Beispiel also g), genauer betrachten:

Einerseits kann es sich dabei um Methoden handeln, die zwar die gleiche Smalltalk-Signatur haben (also den gleichen Namen und die gleiche Anzahl von Parametern), aber semantisch völlig unterschiedlich sind und auch nicht polymorph verwendet werden. Dann sind keine Änderungen an der Vererbungshierarchie der Anwendung nötig und wir können O der Typmenge T zuordnen.

Andrerseits kann es sich um solche Methoden der Klassen aus T handeln, die polymorph genutzt werden, ohne daß diese Methoden in

[19] Wir haben die Klassen in Abbildung 5.19 mit Methodenmengen versehen. Diese Methodenmengen geben an, welche Methoden in der jeweiligen Klasse (re-)definiert werden. Eine Klasse kennt demnach alle Methoden, die sich aus der Vereinigung ihrer Methodenmenge und derer ihrer direkten und indirekten Oberklassen ergeben.

die Schnittstelle der gemeinsamen Oberklasse aufgenommen worden sind. Wir müssen diese polymorphe Nutzung von Methoden in Java explizit ausdrücken, indem wir uns eine geeignete Oberklasse für die Klassen in T *konstruieren*.

Dies können wir tun, indem wir die "fehlenden", polymorph genutzten Methoden der Klassen in T zu einer bestehenden Oberklasse in Form von abstrakten Methoden hinzufügen, oder indem wir eine völlig neue Oberklasse konstruieren und in die Vererbungshierachie der Anwendung mit aufnehmen. Beide Modifikationen an der Struktur der Anwendungen können wir jedoch im allgemeinen nicht automatisch durchführen. Wir werden diese gesondert weiter unten (siehe 2.) betrachten.

Wie können wir entscheiden, ob die in O fehlenden Methoden polymorph verwendet werden?

Wir benötigen dazu zunächst die Menge $D(T)$ aller Variablen der Anwendung, die mit der Typmenge T annotiert werden. Wir bestimmen dann für jede Variable x aus $D(T)$ die Menge $S(x)$ derjenigen Methoden, von denen es Aufrufe gibt, die x als *Receiver*-Objekt verwenden.

Für die Methoden, die in der Methodenmenge

$$S(T) = \bigcup_{x \in D} S(x), \qquad S(T) \subseteq \bigcap_{X \in T} M(X) \tag{5.8}$$

enthalten sind, liegt eine polymorphe Nutzung vor. Wenn wir solche Methoden vorliegen haben, die nicht in O enthalten sind, dann dürfen wir T nicht einfach durch O ersetzen, sondern müssen eine der unter 2. beschriebenen Änderungen an der Klassenstruktur vornehmen.

Beispiel 5.6 In unserem Code-Beispiel aus 5.15 ist

$$D(\{\texttt{Int, Float}\}) = \{\mathtt{a}_{\mathtt{mult_with_}}, \mathtt{b}_{\mathtt{mult_with_}}, \mathtt{a}_{\mathtt{square_}}\}$$

und somit ist $S(\{\texttt{Int, Float}\}) = S(\mathtt{a}_{\mathtt{mult_with_}}) = \{\mathtt{*}\}$.[20]

2. *Konstruktion einer geeigneten Oberklasse O.* Dies kann einerseits dadurch geschehen, indem wir eine vorhandene Oberklasse O erweitern. Dabei definieren wir die polymorph genützten Methoden $S(T)$ der Klassen in T in Form abstrakter Methoden in O. Andrerseits kann es auch notwendig sein, eine komplett neue Oberklasse O in die Vererbungshierachie der Anwendung einzufügen, die die polymorph genützten Methoden $S(T)$ der Klassen in T (ebenfalls als abstrakte Methoden) enthält.

   Für das Herstellen bzw. Modifizieren einer gemeinsamen Oberklasse geben wir kein algorithmisches Verfahren an, statt dessen skizzieren wir die grundlegende Vorgehensweise anhand einiger typischen Situationen.

[20] Wir behandeln hier den Operator `*` wie eine gewöhnliche Methode.

Betrachten wir zunächst Abbildung 5.20. Hier werden die Methoden f und g der Klassen A und B polymorph verwendet, g ist aber nicht in der gemeinsamen Oberklasse O, sondern jeweils separat in A und B definiert. O bildet eine geeignete Abstraktion *beider* Klassen A und B, daher können wir g in O als abstrakte Methode definieren.

Wir können dies jedoch nicht einfach tun, wenn O noch andere direkte Unterklassen besitzt, die g nicht definieren. Dies ist in Abbildung 5.21 illustriert: Hier erbt neben unseren Klassen A und B auch noch die Klasse C von O, C enthält aber die Methode g nicht. Wir müssen daher eine neue Klasse O' einfügen, die lediglich Oberklasse von A und B ist, nicht aber von C. Wir können dann g in dieser Oberklasse O' abstrakt definieren, und die Zuordnung $\{A, B\} \longmapsto O'$ in die Abbildung d aufnehmen.

Betrachten wir noch ein viel allgemeineres Szenario, welches in Abbildung 5.22 dargestellt ist: In Smalltalk können Klassen einer Typmenge T (hier $T = \{A, B\}$) polymorph verwendet werden, die überhaupt keine Beziehung im Vererbungsbaum zueinander haben (abgesehen von einer gemeinsamen Wurzel oder einer anderen gemeinsamen Klasse weit oben im Vererbungsbaum, R). In diesem Fall müssen wir ein *Interface* I erzeugen, welches die polymorph genutzten Methoden $S(T)$ enthält. Wir können dann die Zuordnung $\{A, B\} \longmapsto I$ in d aufnehmen. Dies ist an einem Beispiel in Abbildung 5.22 dargestellt.

All diesen Szenarien ist gemeinsam, daß wir, vereinfacht ausgedrückt, die Schnittstellen der polymorph genutzten Methoden $S(T)$ von den Klassen in T in eine gemeinsame Oberklasse O (bzw. ein Interface I) *verschieben*.

Welche Typannotationen verwenden wir für die Signatur der neuen, abstrakten Methoden in O (bzw. I), die durch dieses "Verschieben" entstehen?

Betrachten wir dazu nochmals das Beispiel in Abbildung 5.20. Wir nehmen folgende Signaturen für die Methode g in den Klassen A und B an:[21]

$$\begin{aligned} g_A &: A \times P_{g_A} \longrightarrow R_{g_A} \\ g_B &: B \times P_{g_B} \longrightarrow R_{g_B} \end{aligned}$$

Wenn wir nun g in O abstrakt definieren, dann benötigt g folgende Signatur:

$$g_O : \quad O \quad \times \quad P_{g_A} \cup P_{g_B} \quad \longrightarrow \quad R_{g_A} \cup R_{g_B}$$

Wir benötigen in den Ausdrücken für den Parameter P_{g_O} und den Rückgabewert R_{g_O} jeweils den Vereinigungsoperator $\cup$, da g_O sich sowohl für

[21] Die $P_{f_A}, R_{f_A}, \ldots$ stehen dabei für Typmengen. In einer Signatur $f : A \times P_{f_A} \longrightarrow R_{f_A}$ bezeichnet A die Klasse, in der die Methode f definiert ist, P_{f_A} einen Parameter der Methode und R_{f_A} den Rückgabewert.

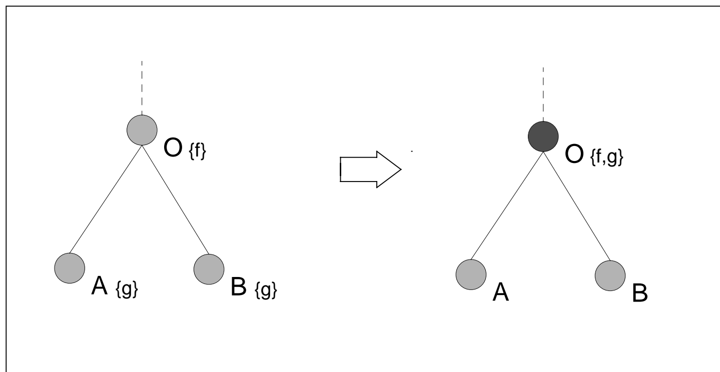

Abbildung 5.20: Modifikation einer bestehenden Klasse.

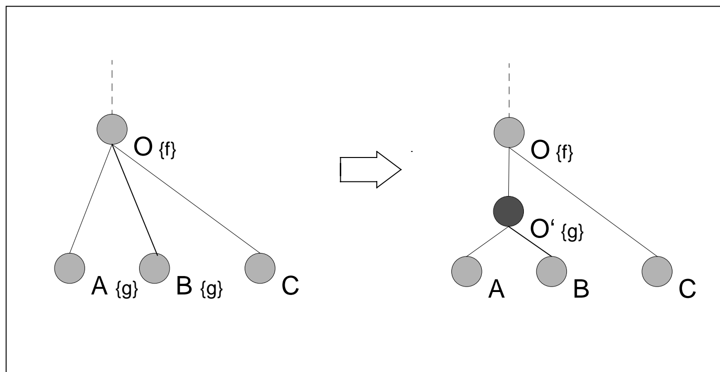

Abbildung 5.21: Einfügen einer neuen Klasse.

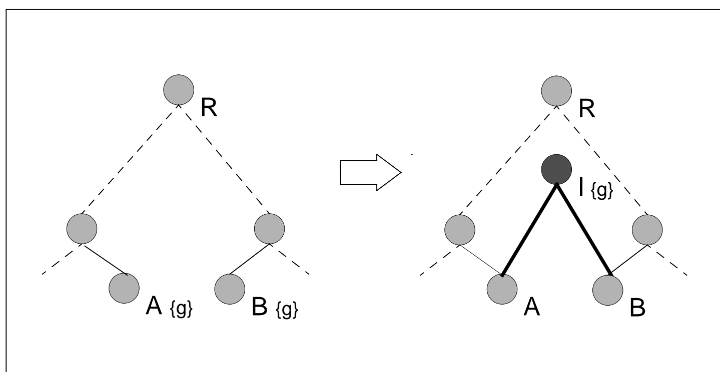

Abbildung 5.22: Einfügen eines neuen Interfaces.

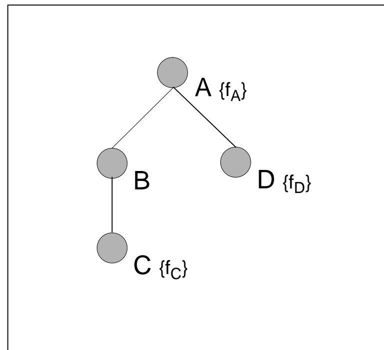

Abbildung 5.23: Vererbung der Methode f mit Redefinitionen.

die konkreten Typen von Parameter und Rückgabewert von g_A als auch von g_B eignen muß.

Wir stellen fest, daß durch die vereinigten Typmengen für die Parameter und Rückgabewerte möglicherweise neue Typmengen entstanden sind, die noch nicht in V' enthalten sind. Wir müssen diese Typmengen dann der Menge V' hinzufügen und für sie auch entsprechende Zuordnungen unter d definieren.

5.9.4 Angleichen von Methodensignaturen

Da Java nur invariante Methodenvererbung kennt, müssen die Methodensignaturen von redefinierten Methoden, die zu Klassen gehören, die in einer Vererbungsbeziehung stehen, entsprechend aneinander angeglichen werden.

Betrachten wir zur Illustration Abbildung 5.23. Die Methode f wird in Klasse A definiert, und in den Unterklassen C und D jeweils redefiniert (überschrieben). B erbt die Methode f von A, ohne sie zu redefinieren.

Wir müssen nun jede der Methoden f in diesem Vererbungsbaum mit denselben Typannotationen versehen. Wenn wir diese Typannotationen schließlich mit Hilfe von d in Java-Signaturen abbilden, genügen diese der von Java geforderten typinvariante Methodenvererbung.

Um Parameter und Rückgabewerte der Methoden f jeweils mit gleichen Typmengen versehen zu können, verwenden wir (wie schon oben bei der Konstruktion der Signaturen für die "hochzuziehenden", polymorphen Methoden) jeweils die Vereinigungen der Typmengen aus den bestehenden Signaturen.

Betrachten wir dazu zunächst die Signaturen der Methoden f, wie sie uns vor der Angleichung vorliegen:[22]

[22]Wir gehen hier davon aus, daß f eine Methode mit einem Parameter ist. Die hier angeführten Überlegungen lassen sich aber ohne weiteres sowohl auf Methoden mit keinem Parameter als auch auf Methoden mit mehreren Parametern übertragen.

$$
\begin{aligned}
f_A &: A \times P_{f_A} \longrightarrow R_{f_A} \\
f_C &: C \times P_{f_C} \longrightarrow R_{f_C} \\
f_D &: D \times P_{f_D} \longrightarrow R_{f_D}
\end{aligned}
$$

Für jede der Klassen A, C und D bekommen wir dann folgende Typmengen für den Parameter P_f und den Rückgabewert R_f:

$$
P_f = \bigcup_{X \in \{A,C,D\}} P_{f_X}, \qquad R_f = \bigcup_{X \in \{A,C,D\}} R_{f_X}
$$

und als gemeinsame Signatur für alle redefinierten Methoden f:

$$
f : X \times P_f \longrightarrow R_f, \qquad X \in \{A, C, D\}
$$

Durch die Verwendung des Vereinigungsoperators in den Ausdrücken für P_f und R_f stellen wir sicher, daß die neue, angeglichene Signatur für f zu *allen* konkreten Typen von Parameter und Rückgabewert der ursprünglichen Signaturen paßt.

Durch die Angleichung der Signaturen können gegebenenfalls völlig neue Typmengen enstehen, die dann ebenfalls in V' integriert und mit Hilfe der Abbildung d abgebildet werden müssen. Wir müssen daher die in den Abschnitten 5.9.2–5.9.4 angeführten Regeln solange immer wieder anwenden, bis alle Typmengen in d berücksichtigt worden, und alle Typmengen in invariant vererbten Methodensignaturen vereinheitlicht sind.[23]

Sobald wir *alle* Typmengen (sowohl die, die wir mittels unseres Typinferenzverfahrens ermittelt haben, als auch die, die durch die Angleichung der Methodensignaturen entstanden sind) in die Abbildung d integriert haben, können wir d direkt dazu nutzten, die Typannotationen des Programms in Typdeklarationen umzuwandeln.

5.9.5 Zusammenfassendes Beispiel

Wir illustrieren die Überlegungen dieses Abschnitts am Beispielprogramm `Test` aus Abbildung 5.15. Dem Beispiel liegt die Klassenstruktur aus Abbildung 5.24 zugrunde.

Im Programm treten die Typmengen $V' = \{\{\texttt{Int}\}, \{\texttt{Float}\}, \{\texttt{Int}, \texttt{Float}\}\}$ auf. Wir konstruieren nun die Abbildung d. Die Typmengen $\{\texttt{Int}\}$ und $\{\texttt{Float}\}$ sind trivial umzusetzen:

[23] Dieser Zustand kann immer erreicht werden, da mit der Menge U der Klassen der Anwendung auch stets die Menge $V' \subseteq \mathcal{P}(U)$ der in der Anwendung auftretetenden Typmengen endlich ist.

$$d \ : \ \{\texttt{Int}\} \longmapsto \texttt{Int}$$
$$d \ : \ \{\texttt{Float}\} \longmapsto \texttt{Float}$$

Interessanter ist die Umsetzung der zweielementigen Typmenge $\{\texttt{Int},\texttt{Float}\}$. `Int` und `Float` werden in `Test` polymorph verwendet. $S(\{\texttt{Int},\texttt{Float}\}) = \{*\}$ ist die Menge der in diesem Zusammenhang polymorph aufgerufenen Methoden (siehe Beispiel 5.6 auf Seite 71). Wir müssen die Methode `*` daher in ein gemeinsames Interface `Number` verschieben, und können die Zuordnung

$$d : \{\texttt{Int},\texttt{Float}\} \longmapsto \texttt{Number}$$

zu d hinzufügen. Abbildung 5.25 zeigt die durch das Interface `Number` modifizierte Vererbungsstruktur des Beispiels.

Aus den Signaturen für die Methode `*`

$$*_{\texttt{Int}} \ : \ \texttt{Int} \times \{\texttt{Int}\} \longrightarrow \{\texttt{Int}\}$$
$$*_{\texttt{Float}} \ : \ \texttt{Float} \times \{\texttt{Float}\} \longrightarrow \{\texttt{Float}\}$$

entsteht gemäß Abschnitt 5.9.4 die vereinheitlichte Signatur

$$*_X : X \times \{\texttt{Int},\texttt{Float}\} \longrightarrow \{\texttt{Int},\texttt{Float}\}, \quad X \in \{\texttt{Number},\texttt{Int},\texttt{Float}\}$$

Diese wird dann zur Java-Signatur `Number mult (Number a);` in den Klassen `Number`, `Int` und `Float`, wenn wir auf diese die Namensregeln aus Kapitel 3 und Abbildung d anwenden.

Wenn wir d schließlich auf die Typannotationen in Abbildung 5.15 anwenden, so erhalten wir den Programmcode aus Abbildung 5.16.

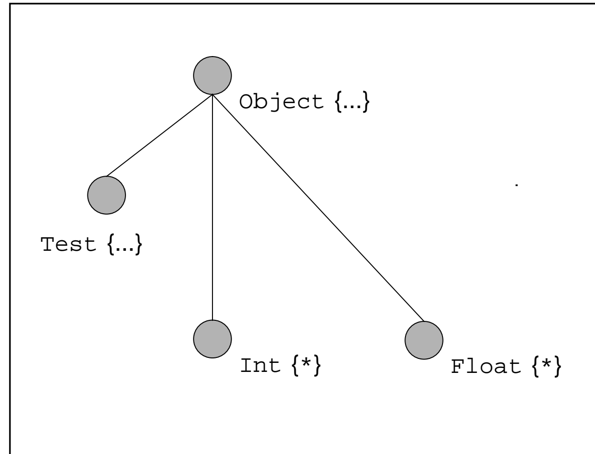

Abbildung 5.24: Ursprüngliche Vererbungshierarchie von `Test`.

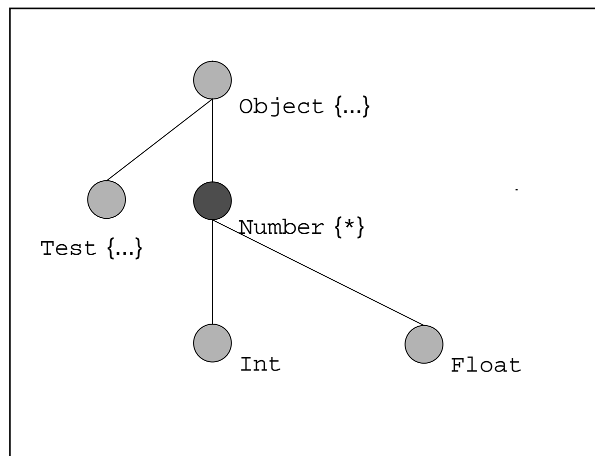

Abbildung 5.25: Modifizierte Vererbungshierarchie von `Test`.

Kapitel 6

Bewertung

In diesem Kapitel werden wir unseren Ansatz aus Kapitel 5 für das Reengineering von Smalltalk-Anwendungen nach Java bewerten. Dabei berücksichtigen wir die Anforderungen aus Abschnitt 1.1. Wir haben dort einerseits gefordert, daß unser Verfahren *automatisierbar* sein muß, andrerseits haben wir festgelegt, daß der entstehende Java-Code bestimmte Eigenschaften erfüllen muß: er soll *typischem* Java-Code entsprechen und *effizient*, *wartbar* und *erweiterbar* sein.

6.1 Automatisierbarkeit

Unser Verfahren zur Umsetzung von Smalltalk-Code in Java-Code kann nur dann sinnvoll eingesetzt werden, wenn es weitgehend automatisierbar ist. Wir werden daher im folgenden unser Verfahren daraufhin untersuchen.

Rufen wir uns dazu noch einmal die einzelnen Schritte des Verfahrens in Erinnerung, wie sie in Abbildung 6.1 in grafischer Form dargestellt sind. Zunächst lesen wir den Smalltalk-Code ein und stellen ihn in Form eines abstrakten Syntaxbaumes dar – dieser erste Schritt ist vollständig automatisierbar, in der Tat liegen den meisten modernen Smalltalk-Systemen bereits entsprechende Klassenbibliotheken, die diese Aufgabe übernehmen, bei. Mit Hilfe des Typinferenz-Algorithmus von Agesen berechnen wir dann für die Variablen des Smalltalk-Codes Typannotationen (siehe Abschnitte 5.1 bis 5.8). Auch dieser Schritt ist vollautomatisch durchführbar, eine prototypische Implementierung des Algorithmus von Agesen für das *VisualWorks-Smalltalk-System* existiert bereits [Li98]. Die Typannotationen nutzen wir schließlich dazu, Typdeklarationen für die Variablen zu erzeugen (siehe Abschnitt 5.9), so daß wir in einem letzten Arbeitsschritt – ebenfalls automatisiert – typisierten Java-Code erzeugen können. (Wie wir Smalltalk-Ausdrücke in Java-Konstrukte übersetzen können, haben wir in Kapitel 3 skizziert.)

Wir stellen fest, daß lediglich bei der Abbildung der Typannotationen in Typdeklarationen Handarbeit erforderlich ist: Wir müssen hierbei jeder der in der Regel mehrelementigen Klassenmengen, die als Typannotationen der Variablen

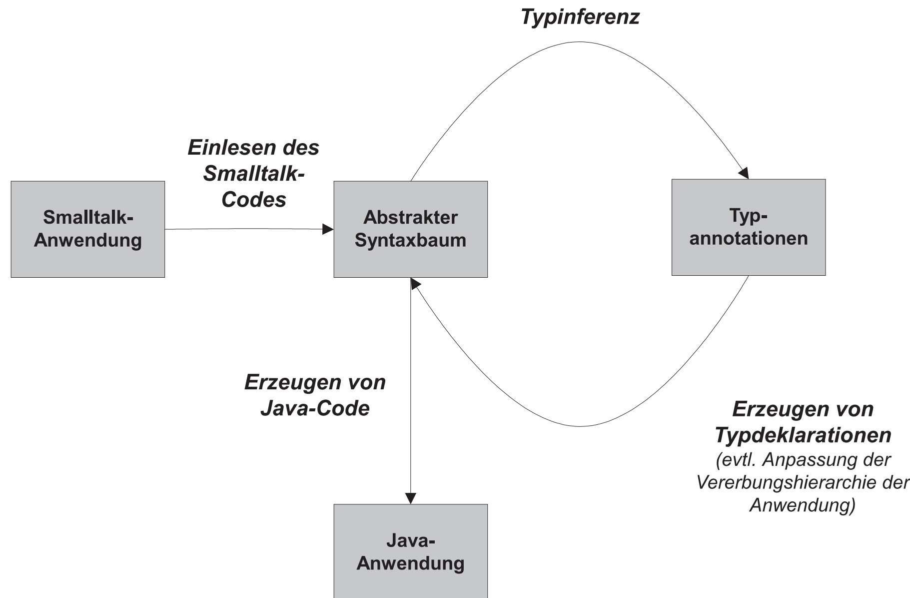

Abbildung 6.1: Reengineering mit Hilfe von Typinferenz.

der Anwendung auftreten, eine einzige Klasse zuordnen, die eine gemeinsame Abstraktion der in der Typannotation enthaltenen Klassen darstellt. Wenn die Anwendung bereits eine solche Klasse enthält, dann können wir diese automatisch identifizieren (siehe Seite 69ff in Abschnitt 5.9.3). Handarbeit ist nur in den Fällen erforderlich, in denen wir uns eine solche Klasse erst noch konstruieren müssen, indem wir eine vorhandene Klasse modifizieren, oder eine Klasse bzw. ein Interface in die Klassenhierarchie einfügen.

Dies ist jedoch nur dann der Fall, wenn der Smalltalk-Code der Anwendung Klassen polymorph verwendet, ohne daß sich dies in einer entsprechenden Klassenhierarchie widerspiegelt. In sorgfältig konstruierten Anwendungssystemen, die einen stabilen Entwicklungsstand erreicht haben (bei denen somit ein Reengineering-Projekt besonders lohnenswert wäre), treten solche Situationen jedoch nur selten auf. Erfahrene Smalltalk-Entwickler tendieren nämlich dazu, Vererbung zur Konstruktion von Interface-Hierarchien einzusetzen *("Programming to an Interface)*, indem sie die Gemeinsamkeiten der Schnittstellen polymorph verwendbarer Klassen mit Hilfe gemeinsamer (abstrakter) Oberklassen ausdrücken. Dies führt zu übersichtlicheren und besser strukturierten Anwendungen (siehe [GHJV95]): Die Klassenhierarchie der Anwendung dokumentiert, welche Klassen sich auf gleiche Weise verwenden lassen.[1]

Smalltalk-Anwendungen, deren zugrundeliegende Klassenhierarchien strikt nach dem Entwurfsprinzip *"Programming to an Interface"* aufgebaut sind, un-

[1]Im Bereich des *Rapid Prototyping* wird das Entwurfsprinzip *"Programming to an Interface"* allerdings nicht immer verfolgt. Hier wird Vererbung eher dazu eingesetzt, um Programmcode der Oberklassen wiederzuverwenden und anzupassen. In solchen Fällen wird unser Reengineering-Verfahren jedoch ohnehin selten angewendet werden, da Prototypen oft durch eine vollständig neu entworfende Anwendung ersetzt werden.

termauern diese Behauptung: So sind beispielsweise sowohl in den Standardklassen eines *Smalltalk-80*-Systems als auch in den Werkzeugen der *VisualWorks*-Entwicklungsumgebung für polymorph nutzbare Klassen immer gemeinsame (abstrakte) Oberklassen vorhanden, die die polymorph aufrufbaren Methoden zusammenfassen.

Wir halten fest: Handarbeit ist nur in den *seltenen* Fällen erforderlich, in denen wir die Klassenhierarchie der Anwendung so modifizieren müssen, daß diese die Gemeinsamkeiten der Klassenschnittstellen geeignet wiedergibt. Falls solche Modifikationen notwendig sind, beschränken sich diese auf das Einfügen einer Klasse in die Klassenhierarchie und das Verschieben von Methoden zwischen Klassen, die in einer Vererbungsbeziehung stehen (vgl. Seite 71ff in Abschnitt 5.9.3). Ein Programmierer kann diese Modifikationen ausführen, indem er lediglich den Programmcode der beteiligten Klassen (die der betreffenden Typannonation und die entsprechenden Oberklassen) betrachtet – es ist dazu nur ein *lokales* Verständnis der Anwendung erforderlich.

Wir folgern, daß unser Verfahren eine beträchtliche Erleichterung darstellt, wenn es darum geht, ausgereifte Smalltalk-Anwendungen nach Java zu übersetzen, da ein Großteil des Smalltalk-Codes überhaupt nicht von Hand bearbeitet werden muß.

6.2 Eigenschaften des entstehenden Java-Codes

In diesem Abschnitt untersuchen wir, ob der mittels unseres Verfahrens aus Kapitel 5 entstehende Java-Code der Anwendung den in Abschnitt 1.1 geforderten Eigenschaften genügt.

Grundeigenschaften. Im Gegensatz zu den in Kapitel 4 dargestellten Verfahren erzeugt das auf Typinferenz basierende Verfahren aus Kapitel 5 typisierten Java-Code.

Typisierter Code hat gegenüber untypisiertem Code die folgenden Vorteile [PS92]:

- *Bessere Lesbarkeit*: Typdeklaration tragen zur besseren Lesbarkeit und zum Verständnis des Programmcodes bei, indem sie die Verwendung von Variablen (insbesondere von Parametern von Methoden) dokumentieren.
- *Höhere Zuverlässigkeit*: Typisierter Java-Code ermöglicht es, bereits zur Übersetzungszeit Typfehler und unzulässige Methodenaufrufe zu entdecken. Dadurch können Laufzeitfehler vermieden werden – die Anwendung arbeitet zuverlässiger.

Außerdem verwendet der resultierende Code Konstrukte, die in gewöhnlichen Java-Anwendungen vorkommen. Insbesondere werden Methodenaufrufe in der üblichen Java-Syntax formuliert und haben auch deren Semantik. Daraus ergeben sich weitere wünschenswerte Eigenschaften des Java-Codes.

- *Wartbarkeit:* Der entstehende Code ist in seiner Beschaffenheit Java-Programmierern vertraut und kann daher leicht verstanden und bearbeitet werden. Dies ist eine wesentliche Voraussetzung für die Wartbarkeit des Systems.
- *Effizienz:* Bei der Ausführung des Codes werden die in der virtuellen Maschine des Java-Systems bereitgestellten effizienten Mechansimen zum Methodenaufruf genützt. Dadurch werden Performance-Probleme, wie sie beispielsweise beim *Reflection*-Ansatz aus Abschnitt 4.2 bestehen, vermieden.

Erweiterbarkeit. Eine weitere Anforderung an den entstehenden Code besteht darin, daß er Weiterentwicklungen und Erweiterungen der Anwendung ermöglichen sollte. Wir wir gleich erläutern werden, ist der durch unser Verfahren aus Kapitel 5 erzeugte Code in dieser Hinsicht nicht optimal.

Es ergeben sich nämlich Einschränkungen daraus, daß wir mit Hilfe von Typinferenz *konkrete* Typinformationen bestimmen und für das Reengineering nutzen. Wenn wir diese konkreten Typinformationen mit Hilfe der Techniken aus Abschnitt 5.9 in Typdeklarationen transformieren, erhalten wir Typdeklarationen für die Variablen, die die Nutzung der Variablen in der bestehenden Smalltalk-Anwendung möglichst genau erfassen. In von Hand implementierten Java-Anwendungen werden Code-Stücke jedoch oft flexibel gehalten, indem entsprechend allgemeine Typdeklarationen gewählt werden. Der durch unser Verfahren erzeugte Code läßt eine solche Flexibilität dagegen nicht zu. Wir illustrieren dies an einem Beispiel.

Beispiel 6.1 Betrachten wir eine Containerklasse `Array`, die beliebige Objekte speichern soll. Mittels einer Methode `put` können ihr Objekte hinzugefügt werden, mittels `get` können Objekte ausgelesen werden.

Da Java (genau wie Smalltalk) keine Generizität unterstützt, werden die Methoden `put` und `get` der Klasse `Array` gewöhnlich so konzipiert, daß sie mit Objekten vom allgemeinen Typ `Object` umgehen können. Sie erhalten daher das folgende `Interface`:

```
void put(Object anObject, int i);
Object get(int i);
```

Um mit der Methode `put` Objekte einer beliebigen Klasse der Anwendung in `Array` zu speichern, müssen diese nach `Object` konvertiert werden, beim Entnehmen mittels `get` müssen sie wieder in Onjekte der ursprünglichen Klassen zurückgewandelt werden.

Unser Typinferenzverfahren bestimmt für untypisierte Varianten von `put` und `get` Typannotationen, die sich aus der tatsächlichen Nutzung von `Array` ergeben: Wenn wir `Array` in der Anwendung lediglich zur Speicherung von Zahlen in Form der Klassen `Int` und `Float` verwenden, so erhalten wir für `put` und `get` die Signaturen

```
void put({Int, Float}anObject, int i);
{Int, Float} get(int i);
```

die dann gemäß Abschnitt 5.9 in

```
void put(Number anObject, int i);
Number get(int i);
```

umgesetzt werden. Die so erzeugte Klasse `Array` läßt sich in diesem Fall nicht mehr mit beliebigen Objekten verwenden, sondern nur mit solchen, die der Klasse `Number` oder einer deren Unterklassen angehören. Die Klasse `Array` büßt damit in erheblichem Maße Flexibilität und Wiederverwendbarkeit ein.

Wir fassen zusammen: Die mit Hilfe unseres Verfahrens aus Kapitel 5 erzeugten Typdeklarationen werden zu speziell, so daß dieses Verfahren für Code, der flexibel verwendet werden soll – insbesondere also Code aus Bibliotheken und Frameworks – oder für Code, der intensiv weiterentwickelt werden soll, weniger geeignet ist.

Solcher Code kann allerdings recht einfach von Hand nachbearbeitet werden, so daß seine ursprüngliche Flexibilität wieder hergestellt werden kann. Wir betonen in diesem Zusammenhang jedoch nochmals, das wir Bibliotheken *ohne* eine Anwendung, die diese nutzt, mit unserem Typinferenz-Verfahren nicht bearbeiten können: Unser Typinferenzverfahren basiert auf dem Prinzip der abstrakten Interpretation (siehe Abschnitt 5.2). Diese kann nur mit dem vollständigen Quellcode ausführbarer Anwendungen durchgeführt werden.

Kapitel 7

Zusammenfassung und Ausblick

In diesem Kapitel fassen wir die Ergebnisse der Arbeit zusammen und führen einige weitere Arbeitsbereiche und Probleme an, die im Zusammenhang mit dieser Arbeit von Interesse sind.

7.1 Zusammenfassung

Ziel der vorliegenden Arbeit war es, Verfahren für die Umsetzung bestehender Smalltalk-Anwendungen nach Java zu erarbeiten.

Eine solche Umsetzung erscheint erfolgversprechend, da sich die Grundkonzepte der Sprachen stark ähneln. Es existiert jedoch ein für die Umsetzung bedeutsamer Unterschied: Smalltalk und Java verwenden unterschiedliche Ansätze zur *Typisierung* des Programmcodes: Smalltalk ist eine dynamisch typisierte Programmiersprache, wohingegen Java statisch typisiert ist: In Java steht der Typ einer jeden Variable bereits zur Übersetzungszeit des Programmes fest, und es läßt sich überprüfen, ob auf dem durch die Variable abgesprochenen Objekt bestimmte Operationen aufgerufen werden dürfen. In Smalltalk ist dies erst zur Laufzeit des Programmes möglich. Dieser Unterschied bringt Konsequenzen hinsichtlich *Polymorphie* mit sich: anders als in Smalltalk sind in Java polymorphe Methodenaufrufe nur für Instanzen von Klassen möglich, bei denen diese Methoden in einer gemeinsamen Oberklasse definiert sind.

Der Schwerpunkt der Arbeit lag demzufolge darin, zu untersuchen, wie diese Unterschiede bei der Umsetzung von Smalltalk-Anwendungen in Java-Anwendungen berücksichtigt werden müssen.

Dazu haben wir eine Reihe von Ansätzen betrachtet, die aus dem untypisierten Smalltalk-Code ebenso untypisierten Java-Code erzeugen. Da der dadurch entstehende Java-Code dem ursprünglichen Smalltalk-Code sehr ähnlich ist, können die meisten Smalltalk-Konstrukte direkt in die entsprechenden Java-Konstrukte übersetzt werden, ohne daß dazu umfangreichere Umstrukturie-

rungsmaßnahmen erforderlich sind. Wir können mit Hilfe dieser Ansätze daher relativ einfach Werkzeuge zur automatischen Übersetzung von Smalltalk-Code in Java-Code konstruieren. Solche Werkzeuge können dann von Nuzten sein, wenn wir bestehende Smalltalk-Anwendungen auf weitere Rechnerplattformen zu übertragen wollen, auf denen kein Smalltalk-System, sondern lediglich eine Java-Umgebung *(JVM, Java Virtual Machine)* vorhanden ist.

Der mit Hilfe dieser Verfahren entstehende Java-Code verwendet jedoch eine Mischung von Smalltalk- und Java-Konzepten. Insbesondere verzichtet er auf statische Typisierung und die damit verbundenen Vorteile (höhere Zuverlässigkeit, bessere Lesbarkeit des Programmcodes). Aus diesen Gründen eignen sich diese Verfahren kaum dazu, um Smalltalk-Anwendungen in echte Java-Anwendungen umzusetzen, die kontinuierlich gepflegt und weiterentwicklet werden sollen.

Wir haben daher ein weiteres Verfahren erarbeitet, mit dem wir authentischen, typisierten Java-Code erzeugen können. Dieses Verfahren besteht aus zwei Schritten:

Zunächst unterziehen wir die bestehende Smalltalk-Anwendung einer datenflußbasierten statischen Programmanalyse, in der wir für alle Variablen im Programmcode konkrete Typinformationen berechnen. Die konkrete Typinformation zu einer Variable ist eine Typmenge, die die Klassen als Elemente enthält, auf deren Instanzen die Variable zur Laufzeit verweisen kann. Zur Bestimmung dieser Typinformationen verwenden wir das Typinferenzverfahren von Agesen, da dieses auch für polymorphen Programmcode zuverlässig arbeitet.

In einem zweiten Schritt erzeugen wir mit Hilfe dieser Typinformationen typisierten Java-Code mit geeigneten Typdeklarationen. Dazu verwenden wir eine Abbildung, die der Typmenge jeder Variablen eine einzelne Klasse zuordnet, die eine gemeinsame Abstraktion der Klassen in der Typmenge in Form einer Oberklasse darstellt. Diese Klasse verwenden wir dann, um die Variable zu deklarieren. Allerdings läßt sich in seltenen Fällen keine solche Klasse finden, dann müssen wir die Struktur der Anwendung modifizieren und eine geeignete Klasse konstruieren.

Auch dieses Verfahren ist automatisierbar, lediglich zur Modifikation der Struktur der Anwendung ist Handarbeit erforderlich. Der mit Hilfe dieses Verfahrens entstehende Java-Code ist typisiert, gut lesbar und wartbar – so umgesetzte Anwendungen können daher ohne Probleme in Java gepflegt und weiterentwickelt werden.

7.2 Ausblick

Im Zusammenhang mit der Umsetzung von Smalltalk-Anwendungen nach Java gibt es noch einige Fragestellungen, die wir in dieser Arbeit nicht behandelt haben.

- Wir sind mit Hilfe unserer Verfahren zwar in der Lage, Smalltalk-Code nach Java zu übersetzen. Damit der so entstehende Java-Code jedoch funktionieren kann, benötigen wir eine gewisse *Infrastruktur*: Wir müssen die Kernelemente eines Smalltalk-Systems (die in Form der Smalltalk-Klassenbibliothek gegeben sind) in Java nachbilden, so daß die Grundfunktionen einer Smalltalk-Umgebung (Objekterzeugung, Systemfunktionen, arithmetische Operationen,...) dem automatisch nach Java transformierten Code zur Verfügung stehen.

- Wünschenswert wäre es ferner, wenn wir den Automatisierbarkeitsgrad des in Kapitel 5 erarbeiteten Verfahrens zur Erzeugung echten, typisierten Java-Codes noch weiter erhöhen könnten. Fortschritte in diesem Bereich lassen sich möglicherweise dadurch erzielen, daß wir die Umstrukturierungen an der Vererbungshierarchie, die dabei erforderlich sein können, in Form von Grundoperationen (vergleichbar den *Refactorings* aus [Opd92]) zusammen mit bestimmten Situationen, in denen sie anzuwenden sind, spezifizieren.

- Die in dieser Arbeit dargestellten Ansätze beschränken sich auf Quellcode-Transformationen zwischen Smalltalk und Java. Eine andere Möglichkeit, Smalltalk-Anwendungen in Java-Umgebungen zu übertragen, besteht darin, Smalltalk-Quellcode (oder evtl. auch Smalltalk-Bytecode) direkt in Java-Bytecode zu übersetzen. Allerdings erhalten wir damit keinen Java-Code, den wir weiterpflegen können, sondern wir müssen weiterhin mit dem Smalltalk-Quellecode arbeiten. Dafür erschließen wir uns alle Systemplattformen, auf denen eine Java-Umgebung verfügbar ist.

Zum Abschluß halten wir jedoch fest, daß es uns insbesondere mit dem Typinferenz-Ansatz aus Kapitel 5 gelungen ist, erfolgversprechende Methoden zum Reengineering bestehender Smalltalk-Anwendungen nach Java zu erarbeiten. Die Praxistauglichkeit dieser Methoden muß jedoch mit Hilfe eines experimentellen Werkzeuges zur Umsetzung von Smalltalk-Code in Java-Code und anhand von tatsächlich eingesetzten Smalltalk-Anwendungen untermauert werden.

Anhang A

Die Klasse `STObject` des Reflection-Ansatzes

Wir geben hier eine einfache Implementierung der Klasse **`STObject`** für den Reflection-Ansatz aus Kapitel 4 an.

```
/**
 * STObject.java
 *
 * Diese Klasse implementiert teilweise die Funktionalitaet der
 * Klasse 'Object' der Smalltalk-Standardbibliothek.
 *
 * Mit Hilfe dieser Klasse wird das in Smalltalk realiserte Konzept
 * des Methodenaufrufs in Java simuliert. Dazu werden
 * 'perform'-Methoden zur Verfuegung gestellt, mit deren Hilfe eine
 * Methode anhand eines 'Selectors' aufgerufen werden kann. Die Typen
 * fuer Receiver und Parameter muessen dabei zur Uebersetzungszeit
 * nicht feststehen.
 *
 */

import java.lang.reflect.*;

public class STObject {

    private static Class stObjClass;

    /**
     * Konstruktor.
     */
    public STObject () {

        // Erstelle ein Klassenobjekt von STObject für die
        // Elemente der Signatur einer aufzurufenden Methode
        // (siehe perform()-Methoden).
        try {
            stObjClass = Class.forName("STObject");
```

```
    }
    catch (ClassNotFoundException e) {
        System.err.print("Class 'STObject' not found.");
        System.exit(1);
    }
}

/**
 * Ruft eine parameterlose Methode des Objekts auf.
 * @param selector Name der aufzurufenden Methode.
 * @return         Ergebnis (Rueckgabewert) des Aufrufs.
 */
public STObject perform (String selector) {

    STObject args[] = new STObject[0];
    return perform (selector, args);
}

/**
 * Ruft eine Methode des Objekts mit einem Parameter auf.
 * @param selector Name der aufzurufenden Methode.
 * @param arg      Parameter
 * @return         Ergebnis (Rueckgabewert) des Aufrufs.
 */
public STObject perform (String selector, STObject arg) {

    STObject args[] = new STObject[1];
    args[0] = arg;
    return perform (selector, args);
}

/**
 * Ruft eine Methode des Objekts mit zwei Parametern auf.
 * @param selector Name der aufzurufenden Methode.
 * @param arg1     erster Parameter
 * @param arg2     zweiter Parameter
 * @return         Ergebnis (Rueckgabewert) des Aufrufs.
 */
public STObject perform (String selector,
                         STObject arg1, STObject arg2) {

    STObject args[] = new STObject[2];
    args[0] = arg1;
    args[1] = arg2;
    return perform (selector, args);
}

/**
 * Ruft eine Methode des Objekts auf, die ueber beliebig viele
 * Parameter verfuegen darf.
 * @param selector Name der aufzurufenden Methode.
 * @param args     Parameterliste
 * @return         Ergebnis (Rueckgabewert) des Aufrufs.
```

```
 */
public STObject perform (String selector, STObject[] args) {

    Method method = null;

    // Vorbereiten der Methodensignatur
    Class formalParameters[] = new Class[args.length];
    for (int i = 0; i < args.length; i++) {
        formalParameters[i] = stObjClass;
    }

    // Gibt es eine Methode mit dem Namen 'selector'?
    try {
        method = this.getClass().getMethod(selector,
            formalParameters);
    }
    catch (NoSuchMethodException e) {
        signalMessageNotUnderstood(selector);
    }

    // Versuche, die Methode aufzurufen
    Object returnValue = null;
    try {
        returnValue = method.invoke(this, args);
    }
    catch (Exception e) {
        signalMessageNotUnderstood(selector);
    }

    return (STObject) returnValue;

}

/**
 * Liefert eine STString-Repraesentation des Objekts.
 *
 * Diese Methode sollte in Unterklassen
 * passend ueberschrieben werden.
 * @return String-Repraesentation des Objekts.
 */
public STObject printString () {
    return (new STString (getClass().getName()));
}


/**
 * Gibt eine Fehlermeldung aus, die besagt, dass
 * der Aufruf der Methode 'selector' gescheitert ist
 * und beendet das Programm.
 * @param selector Name der Methode, deren Aufruf scheitert.
 */
private void signalMessageNotUnderstood(String selector) {
```

```
        System.err.print("Message ");
        System.err.print(selector);
        System.err.println(" not understood.");
        System.exit(1);
    }
}
```

Literaturverzeichnis

[Ada95] Jim Adamczyk. Smalltalk reaches crossroads in the insurance industry. *Communications of the ACM*, 38(10):107–109, October 1995.

[Age94] Ole Agesen. Constrained-based type inference and parametric polymorphism. In *Proceedings of the First International Static Analysis Symposium (SAS '94)*, volume 864 of *LNCS*. Springer-Verlag, 1994.

[Age95] Ole Agesen. The Cartesian product algorithm. In Walter Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *LNCS*, pages 2–26, Berlin, GER, August 1995. Springer-Verlag.

[ASU85] A. V. Aho, Ravi Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1985.

[BG93] Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. *ACM SIGPLAN Notices*, 28(10):215–230, October 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).

[Bot96] Per Bothner. Translating Smalltalk to Java. Technical report, Cygnus Solutions, 1996.

[Bud91] Timothy Budd. *Object-Oriented Programming*. Addison-Wesley, New York, N.Y., 1991.

[Bur92] Steve Burbeck. *Applications Programming in Smalltalk-80: How to use Model-View-Controller*, 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[Chu92] Michael Chung. Comparing Java with Smalltalk. `http://www.chisolutions.com/papers/compare/compare.htm`, 1992.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[EK98] R. L. Engelbrecht and D. G. Kourie. Issues in translating Smalltalk to Java. In Kai Koskimies, editor, *Compiler Construction 98*, volume 1383 of *LNCS*. Springer, 1998.

[Fla97] David Flanagan. *Java in a Nutshell: a desktop quick reference*. A Nutshell handbook. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, second edition, 1997.

[Fus97a] Mark L. Fussell. Java and Smalltalk syntax compared. Technical report, ChiMu Publications, 1997.

[Fus97b] Mark L. Fussell. SmallJava: Using language transformation to show language differences. Technical report, ChiMu Publications, 1997.

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

[GJ90] Justin O. Graver and Ralph E. Johnson. A type system for Smalltalk. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 136–150, San Francisco, California, January 17–19, 1990. ACM SIGACT-SIGPLAN, ACM Press.

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

[Gol84] Adele Goldberg. *Smalltalk-80 — The Interactive Programming Environment*. Addison-Wesley, Reading (MA), 1984.

[Gol95] Adele Goldberg. Why Smalltalk. *Communications of the ACM*, 38(10):105–107, October 1995.

[Gos96] James Gosling. The Java language. Technical report, Javasoft, 1996.

[GR85] Adele Goldberg and David Robson. *Smalltalk-80 — The Language and its Implementation*. Addison-Wesley, Reading (MA), 1985.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80 — The Language*. Addison-Wesley, Reading (MA), 1989.

[HH95] Trevor Hopkins and Bernard Horan. *Smalltalk: an introduction to appclication development using Visual Works*. Prentice Hall, 1995.

[HHJW93] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. Technical report, University of Glasgow, 1993.

[HJW+92] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzman, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partian, and J. Peterson. Report on the programming language haskell, version 1.2. *Sigplan*, 27(5), May 1992.

[KS91] Jens Knoop and Bernhard Steffen. The interprocedural coincidence theorem. Technical Report 91-27, Technical University of Aachen (RWTH Aachen), 1991.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, January 1977.

[Lew95] Simon Lewis. *The Art and Science of Smalltalk.* Prentice Hall, 1995.

[Li98] Jinhua Li. Maintenance support for untyped object-oriented systems. `http://www.informatik.uni-stuttgart.de/ifi/se/mitarbeiter/jinhua/`, 1998.

[Lou93] Kenneth C. Louden. *Programming Languages: Principles and Practice.* PWS Publishing Company, 1993.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, 1997.

[Mar97] Florian Martin. PAG — an efficient program analyser generator. Technical report, Universität des Saarlandes, Postfach 151150, 66041 Saarbrücken, Germany, 1997.

[McC97] Steve McClure. Smalltalk strengths stand out. Technical report, IDC, 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.

[MNC+91] Gerald Masini, Amedo Napoli, Dominique Colnet, Daniel Leanard, and Karl Tombre. *Object Oriented Languages.* Academic Press Limited, 1991.

[MR90] T. J. Marlowe and B. G. Ryder. Properties of Data Flow Frameworks. *Acta Informatica*, 28:121–161, 1990.

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

[OPS92] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 329–349. Springer-Verlag, New York, N.Y., 1992.

[Ori98] Orisa GmbH. Konvertierung kompletter Smalltalk-Projekte nach Java. `http://www.orisa.de/`, 1998.

[PC94] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. *ACM SIGPLAN Notices*, 29(10):324–324, October 1994.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. Technical Report DAIMI PB - 345, Computer Science Department, Aarhus University, March 1991.

[PS92] Jens Palsberg and Michael I. Schwartzbach. Three discussions on object-oriented typing. *ACM OOPS Messenger*, 3(2):31–38, 1992.

[PS94] G. Phillips and T. Shepard. Static typing without explicit types. Technical report, Dept. of Electrical Engineering and Computer Science, Royal Military College of Canada, 1994.

[RB98] Don Roberts and John Brant. Good enough analysis for refactoring. Technical report, Department of Computer Science, University of Illinois, 1304 W. Springfiled Ave., Urbana, IL 61801, 1998.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Sytems*, 3(4):253–263, 1997.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. In *Theory and Practice of Object Systems*, 1998.

[Sha95] Yen-Ping Shan. Introduction: Smalltalk on the rise. *Communications of the ACM*, 38(10):102–104, October 1995.

[SU95] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for self. In Walter Olthoff, editor, *ECOOP '95 - Object-Oriented Programming 9th European Conference, Aarhus, Denmark*, number 952 in Lecture Notes in Computer Science, pages 303–330. Springer-Verlag, New York, N.Y., 1995.

[Sun97] Sun Microsystems. *JDK 1.1.5 Documentation — the Java Reflection API*, 1997.

[Til96] Scott R. Tilley. Perspectives on legacy systems reengineering. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA, 1996.

[US91] D. Ungar and R. B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

[WG84] W. M. Waite and G. Goos. *Compiler Construction.* Springer, New York, 1984.