Vergleich der Anwendungsarchitekturen von Smalltalk und Java im Hinblick auf Migration

Christian Wege

WSI-98-12

Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich Programmierung
Eberhard-Karls-Universität Tübingen
Sand 13
72076 Tübingen
Germany

Email: wege@acm.org

ISSN 0946-3852

Zum Autor:

Christian Wege ist wissenschaftlicher Mitarbeiter der DaimlerChrysler AG. Sein Interessensschwerpunkt liegt im Bereich Architektur verteilter objekt–orientierter Applikationen. Die vorliegende Arbeit entstand unter Betreuung von Prof. Klaeren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1

Einleitung

Die Motivation für die vorliegende Arbeit ist der Wunsch, bestehende Smalltalk–Applikationen nach Java zu migrieren.

In Java sehen viele Entwickler die Lösung für die Gestaltung von Applikationen in einem stark vernetzten, heterogenen Umfeld. Java schließt die Lücke zwischen den portablen, aber sehr langsamen Skript–Sprachen (Bsp. TCL) und den schnellen aber schwer portierbaren Low–Level–Sprachen wie C oder C++ („programming to the bare metal languages“)[1]. Insbesondere diese große Bandbreite von Java macht sie als Entwicklungsplattform sehr interessant.

Zur Entwicklung von Client–Applikationen mit graphischer Benutzeroberfläche wurde innerhalb des Daimler-Benz Konzerns oft Smalltalk verwendet. Zunehmende wirtschaftliche Schwierigkeiten der Hersteller von Smalltalk–Entwicklungsumgebungen und abnehmende Attraktivität von Smalltalk als Entwicklungsplattform insgesamt[2] lassen eine Migrationsstrategie weg von Smalltalk zunehmend notwendig erscheinen. Smalltalk hat es im kommerziellen Einsatz bisher nicht geschafft, große Bedeutung zu erlangen.[3] Zudem ist Smalltalk für den Einsatz auf Servern nicht sehr gut geeignet.

Viele der guten Elemente von Smalltalk sind in Java wiederzufinden („Java [...] and Smalltalk are comparable programming environments offering the richest set of capabilities for software application developers.“[4]). Java bietet zudem eine größere Sicherheit, Portabilität und Einsetzbarkeit in einem vernetzten Umfeld. Durch die große Nähe zwischen Java und Smalltalk und die Vorteile von Java als Entwicklungsplattform, erscheint die Migration von Smalltalk–Anwendungen nach Java als sehr attraktiv.

1.1 Problemstellung

Das Thema dieser Arbeit ist ein Vergleich der Applikationsarchitekturen von Smalltalk und Java im Hinblick auf Migration von Smalltalk nach Java.

Hintergrund dieser Fragestellung ist der Wunsch, eine bestehende Smalltalk–Applikation nach Java zu migrieren. Damit sich die migrierte Applikation nahtlos in die Zielplattform einfügt, ist die Übertragung der Designideen, die hinter einer spezifischen Smalltalk–Implementierung stecken, von besonderer Bedeutung. Die migrierte Applikation soll die Elemente von Java so verwenden, wie

[1] siehe Gosling et al., „The Java Language Environment – A White Paper“

[2] Im August 1998 erschien die letzte Ausgabe der Zeitschrift *Smalltalk Report*

[3] In einer IDC Studie von Anfang 1997 zur vorwiegend eingesetzten Programmiersprache in Betrieben, die objektorientiert entwickeln, gaben 70% der Befragten C++ und nur 7% Smalltalk an.

[4] siehe ebenfalls Java White Paper

es dem Stil von Java entspricht. Dadurch soll die Lesbarkeit und Wartbarkeit nach der Migration erhalten bleiben.

Unter der Architektur einer Applikation verstehen wir im Rahmen dieser Untersuchung die Kombination von Mustern[5] auf den verschiedenen Abstraktionsebenen. Jedes Muster trägt als „Mikroarchitektur“ zur Architektur einer Applikation bei. Die Gesamtarchitektur einer Applikation ergibt sich aus der Summe der eingesetzten bzw. verwendeten Muster.

Die Migration anhand von Mustern hilft, die Applikation in Java so zu implementieren, daß sie dem Stil der Java–Plattform entspricht. Jede Musterimplementierung stellt eine typische Verwendung der Elemente der Plattform dar. Die Migration von Mustern untersucht, welche Elemente auf welche Art in Java verwendet werden, um die gleiche Funktionalität wie in Smalltalk zu bieten. Diese Migration kann Änderungen in der Applikationsarchitektur mit sich bringen. Durch diese Art der Migration erhält man eine für Java typische Applikation mit einer für Java typischen Architektur. Die Applikation erscheint nach der Migration nicht als Smalltalk–Programm in Java–Verkleidung, sondern so, als sei sie direkt in Java geschrieben worden.

1.1.1 Bestehende Migrationsansätze

Nicht nur bei Daimler–Benz besteht der Bedarf nach einer Migration von Smalltalk nach Java. Daher existieren bereits Werkzeuge, die eine mehr oder weniger automatische Migration von Smalltalk nach Java versprechen. Im Folgenden werden kurz einige der bisherigen Ansätze besprochen und deren Einfluß auf die Applikationsarchitektur dargestellt.

Der einfachste Ansatz besteht aus einer rein syntaktischen Konvertierung des Quelltextes nach Java [Gmb97]. Diese Art der Migration ignoriert vollständig die Architektur, die in einer Applikation von der bestehenden Klassenbibliothek induziert wird. Sie spart lediglich Schreibarbeit. Die eigentliche Migrationsleistung muß anschließend von Hand erbracht werden.

Andere Ansätze gehen von der Migration der graphischen Benutzeroberfläche aus unter Belassung der Kernfunktionalität der Applikation in Smalltalk. Hierzu wird die Oberfläche der Applikation in Java nachgebildet. Die Kommunikation mit dem Applikationskern erfolgt beispielsweise über CORBA [Kra98, Obj97]. Diese Art der Migration dient dazu, einer Smalltalk–Applikation einen Webbrowser–basierten Zugang zu verschaffen. Dabei wird allerdings nur ein kleiner Teil der Applikation nach Java konvertiert.

Ein weiterer Ansatz geht wissensbasiert vor. Zunächst wird ein „Mapping“ erstellt von den Klassen der Smalltalk–Klassenbibliothek nach Java. Auf dieser Basis wird die Verwendung von Smalltalk–Klassen durch die Verwendung von Java–Klassen ersetzt. Viele der durch das Fehlen von Codeblocks und die dynamische Typisierung bei der Migration auftretenden Probleme werden mit Hilfe von Interfaces und Wrapper–Klassen gelöst [Ori97]. Die Autoren geben selbst zu, daß ein Programmierer diesen Code „transparenter, eleganter - und damit pflegbarer - erstellt hätte“. Dieser Ansatz bietet zwar den größten Grad an automatischer Umsetzung, der generierten Java–Applikation sieht man ihre Smalltalk–Herkunft aber deutlich an.

1.2 Überblick

Dieser Abschnitt soll dem Leser den Einstieg in die Arbeit erleichtern. Wir beschreiben den Zugang zu der Arbeit auf drei verschiedene Arten. Diese verschiedenen Zugänge sind in Abbildung 1.1 graphisch dargestellt.

Kernaussagen

Kapitel 3 enthält die ausführliche Darstellung der Fragestellung. Kapitel 7 faßt den Gang der Untersuchung der Arbeit zusammen, stellt den Beitrag neuen Wissens heraus und zieht Fazit. Eine

[5] Muster sind Beschreibungen von bewährten Lösungen immer wiederkehrender Probleme bei der Entwicklung von Software. Siehe 2.4.

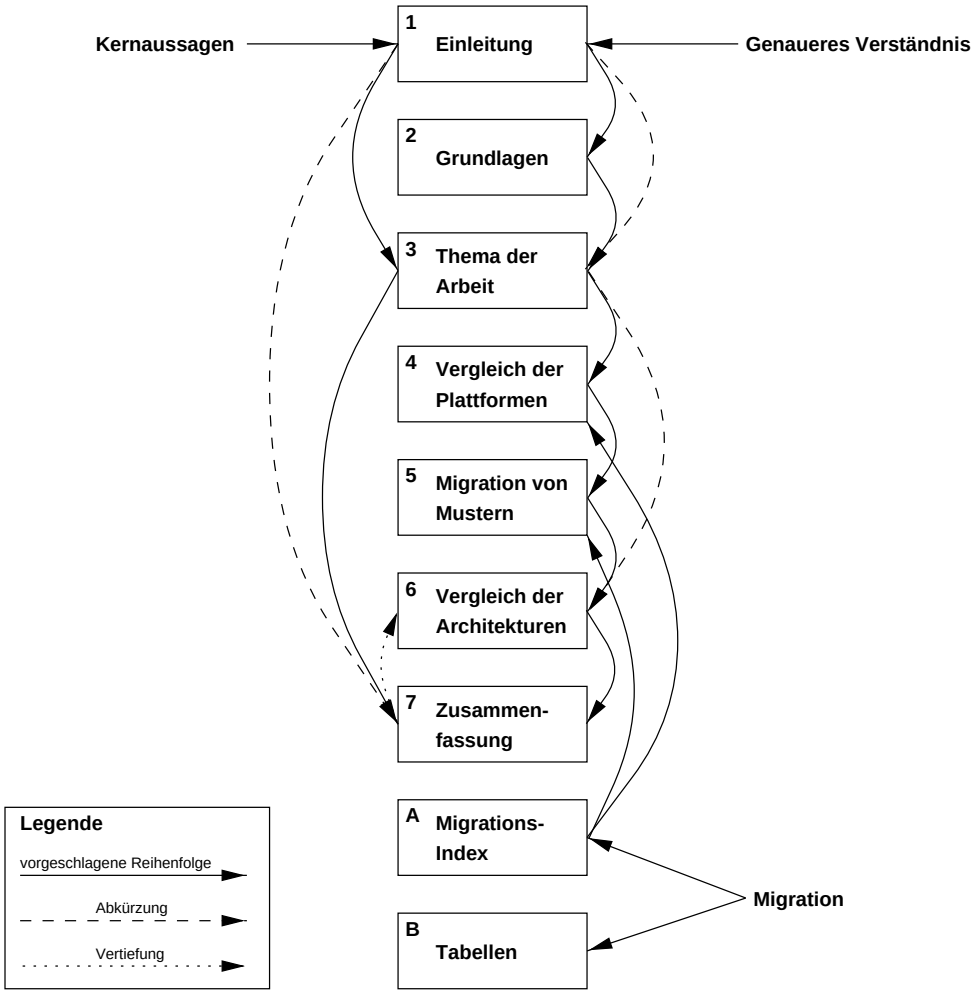


Abbildung 1.1: Arten des Zugangs zur Arbeit

Auflistung der gefundenen Probleme befindet sich in Kapitel 6, welches bei diesem Zugang zur Arbeit der Vertiefung dient.

Genaueres Verständnis

Für ein genaueres Verständnis der Arbeit empfiehlt sich die Lektüre der Arbeit in der Reihenfolge, die von den Kapiteln vorgegeben ist. Kapitel 2 stellt Java und Smalltalk kurz vor und gibt einen Überblick zu Wiederverwendung, Mustern und Softwarearchitektur. Bei Vertrautheit mit der Thematik kann dieses Kapitel übersprungen werden. Kapitel 4 und 5 enthalten die ausführliche Darstellung der Ergebnisse der Untersuchung. Die darin gefundenen Probleme werden im darauffolgenden Kapitel zusammengefaßt und diskutiert.

Migration

Bei der Durchführung der Migration ergibt sich eine gänzlich andere Herangehensweise an die Arbeit. Bei der Migration ist ein schnelles Auffinden der passenden Textstelle aufgrund der Konstrukte im Programmcode wichtig. Der Migrationsindex enthält solche Verweise auf die Kapitel 4 und 5. Anhang B stellt einige Klassen und Methoden von Smalltalk und Java einander gegenüber, die eine ähnliche Funktionalität abdecken.

1.3 Notation

Zur Darstellung der Diagramme der Arbeit wurde so weit wie möglich die *Unified Modelling Language* (UML) verwendet [Fow97]. Die UML hat das Ziel, eine einheitliche Notation für die Entwurfsdokumente bei der Applikationsentwicklung bereitzustellen. Die UML wurde kürzlich von der OMG[6] standardisiert. Abbildung 1.2 stellt die Elemente der UML vor, die in der Arbeit verwendet wurden.

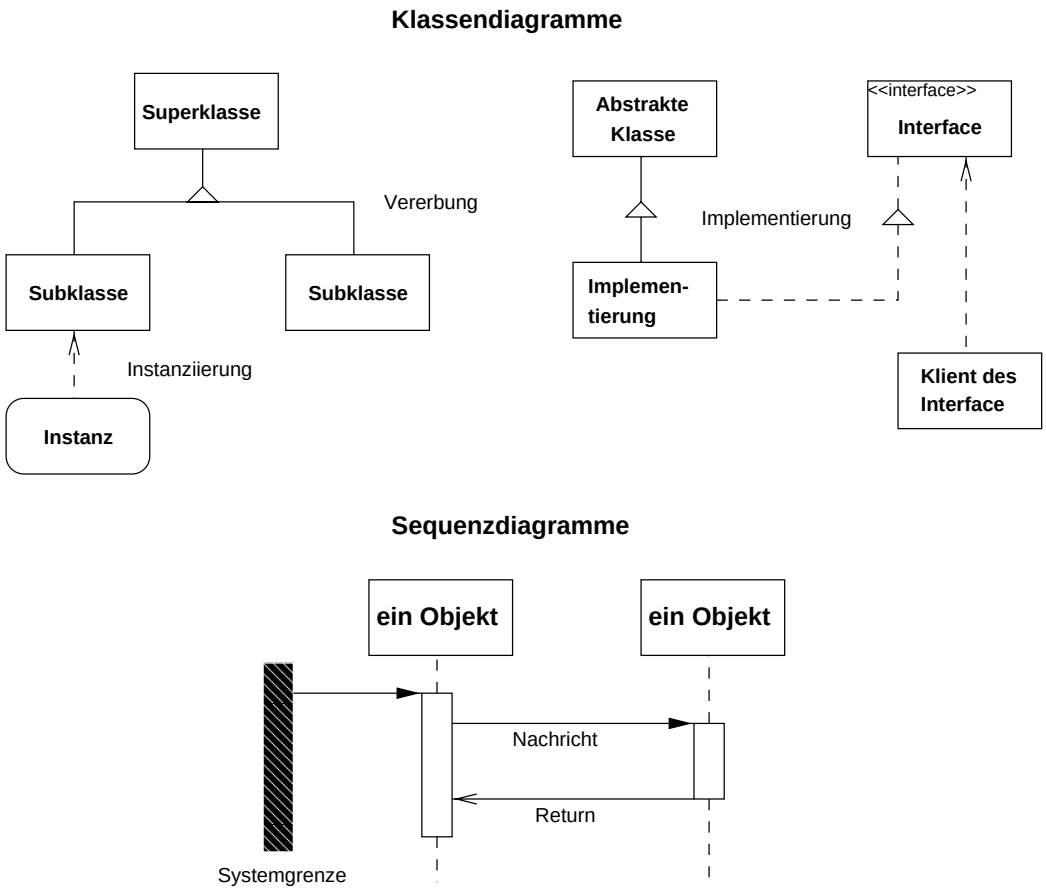


Abbildung 1.2: Elemente der UML

Innerhalb der Arbeit wurden manche Begriffe durch textuelle Hervorhebungen markiert. Begriffe, die neu eingeführt und an dieser Stelle erklärt bzw. definiert werden, sind **fett** gesetzt. Neu eingeführte Begriffe, die jedoch nicht an dieser Stelle erklärt werden, sind *kursiv* gesetzt. Die Erklärung dieser Begriffe findet sich im Glossar. Innerhalb der Beschreibungen zur Mustermigration sind die beteiligten Entitäten bei ihrer ersten Erwähnung ebenso wie allgemeine Betonungen ebenfalls kursiv gesetzt. Codebeispiele sind in `Maschinenschrift` gesetzt.

[6] Object Management Group. Internationale gemeinnützige Vereinigung mit dem Ziel der Weiterentwicklung und Weiterverbreitung von Objekt–Technologie.

2

Grundlagen

2.1 Smalltalk

Geschichte

Die Geschichte der Programmiersprache Smalltalk reicht zurück in die frühen 70er Jahre. Damals war das übliche Ziel beim Entwurf einer Programmiersprache, die Software so zu bauen, daß sie auf der aktuellen Hardware lief.

Bei Smalltalk wurde dagegen zunächst die Software entwickelt und anschließend die Hardware den Anforderungen der Software angepaßt. Die Software sollte die Abbildung der realen Welt auf die Hardware mit den Darstellungsmöglichkeiten der Sprache erlauben und eine Oberfläche zum effektiven Arbeiten mit dem Computer bereitstellen. Der ursprüngliche Ansatz ist mit den heutigen *Thin clients* vergleichbar.

Die Ergebnisse dieser Entwicklungen mündeten damals in Smalltalk–80, welches die Wurzel aller verfügbaren Smalltalk–Implementierungen ist. Die maßgeblichen Ideen der reinen Objektorientierung und der virtuellen Maschine gehen auf Alan Kay zurück, der als Vater von Smalltalk gilt. Die erste kommerzielle Version stammt von Adele Goldberg. Aus diesem Produkt entstand das bekannte VisualWorks.

Die Standardisierung durch ein ANSI Technical Committee hat das Ziel, eine gemeinsame Basis für die Portierung zwischen verschiedenen Sprachimplementierungen zu schafffen. Bislang liegt lediglich eine Vorversion des Standards vor [ANS97].

Eigenschaften

Reine Objektorientierung

Smalltalk zeichnet sich durch seine *reine Objektorientierung* aus. Jedes Sprachkonstrukt, das der Programmierer verwendet, ist ein Objekt — angefangen von Zahlen bis hin zu Teilen der Entwicklungsumgebung.

Kapselung

Kapselung ist ein Zusammenfassen des Zustands und des Verhaltens im Objekt, um die Integrität der Daten garantieren zu können [WBWW90, S.18]. Auf die Daten eines Objekts kann nur über seine Schnittstelle zugegriffen werden. Diese Schnittstelle wird von Methoden realisiert, die durch Nachrichten an das Objekt angestoßen werden. Durch die Trennung von Schnittstelle und Implementierung der Methoden eines Objekts, kann die Implementierung geändert werden, ohne einen Einfluß auf aufrufende Objekte befürchten zu müssen.

Polymorphie

Durch Überschreiben von Methoden in einer Subklasse erreicht Smalltalk *Polymorphie*. Der Sender einer Nachricht kann keine Aussage über die konkrete Klasse des Empfängerobjekts und somit über die Auswahl der implementierenden Methode treffen. Diese Auswahl ist abhängig von

der Klasse des empfangenden Objekts, welche die gleiche Semantik auf andere Art implementieren kann. Der Mechanismus, der dahinter steht, ist *late binding*: Erst beim Versenden der Nachricht ermittelt das System die Methode, welche auf die Nachricht reagiert.

Late binding

Eine *Klasse* dient der Beschreibung der Objekte dieser Klasse. Jedes Objekt ist eine Instanz einer Klasse, welche als Vorlage für weitere Instanzen dient. Die Klasse selbst ist auch wieder eine Instanz einer übergeordneten Klasse.

Klassen

Die *automatische Speicherverwaltung* gibt nicht mehr benötigte Objekte frei. Ein Programmierer spricht ein Objekt über dessen Objektzeiger an, der jedoch transparent gestaltet ist. Es existiert kein Mechanismus zum expliziten Freigeben eines Objekts. Falls keine Referenzen mehr auf ein Objekt existieren, kann es freigegeben werden.

Automatische Speicherverwaltung

Variablen sind in Smalltalk *dynamisch getypt*. D.h. erst das Laufzeitsystem kann die Klasse des Objekts ermitteln, auf das eine Variable zeigt. Der Compiler kann somit zur Übersetzungszeit noch nicht feststellen, ob ein Objekt eine bestimmte Nachricht tatsächlich versteht.

Dynamische Typisierung

Um portabel zu sein, wird ein Smalltalk–Programm nicht in Maschinencode übersetzt, sondern in maschinenunabhängigen Zwischencode. Die Basis für die Ausführung eines übersetzten Programms ist eine *virtuelle Maschine*, die den Zwischencode interpretiert. Erst bei der Ausführung eines Smalltalk–Programms erzeugt ein *inkrementeller Compiler* Maschinencode, der für weitere Aufrufe gespeichert wird.

virtuelle Maschine

inkrementeller Compiler

Entwicklungsumgebungen

Eine der bekanntesten Entwicklungsumgebungen für Smalltalk ist VisualWorks. Sie geht direkt auf die erste Version von Smalltalk–80 zurück und zeichnet sich durch eine enorm große Klassenbibliothek und mächtige Entwicklungswerkzeuge aus. Das Produkt besteht im Wesentlichen aus einer virtuellen Maschine und einem Image, welches sämtlichen Anwendungscode und alle Objekte enthält. Für diese Arbeit spielt VisualWorks eine untergeordnete Rolle und wird lediglich am Rande erwähnt.

VisualWorks

Ein ernstzunehmender Konkurrent von VisualWorks ist VisualAge for Smalltalk. Es besteht aus der Basis IBM Smalltalk und einer visuellen komponentenbasierten Entwicklungsumgebung. IBM Smalltalk besteht ebenfalls aus einer virtuellen Maschine und einem Image, das wieder als Behälter für den Programmcode und die erzeugten Objekte dient. IBM Smalltalk ist die Basis für den Plattformvergleich dieser Arbeit, da sich seine Sprachdefinition sehr eng an bestehende Standards anlehnt.

IBM Smalltalk

2.2 Java

Geschichte

Java ist eine Programmiersprache, die Ende 1995 von der Firma Sun vorgestellt wurde. Sie sollte ursprünglich der Programmierung von Geräten der Unterhaltungselektronik dienen. Da sich C++ als zu problematisch für die gestellten Aufgaben erwies, entwickelte Sun die Sprache Java. Die Syntax orientiert sich eng an C++, verzichtet aber auf viele problembehaftete Eigenschaften und greift Elemente von Smalltalk und Objective–C wieder auf. Zur Zeit bereitet Sun eine Standardisierung von Java durch die ISO vor.

Eigenschaften

Die Sprache vereinigt einige Eigenschaften auf sich, die weit über reine Objektorientierung hinausgehen. Im Folgenden werden diese Eigenschaften aufgelistet und kurz beschrieben [GM95].

objektorientiert

Um *Objektorientierung* zu realisieren, kennt Java die Konzepte Klasse, Instanz, Vererbung,

Kapselung und Polymorphie.

verteilt

Um *verteiltes Rechnen* zu ermöglichen unterstützt Java die Protokolle TCP/IP Sockets, FTP Client und Server, URL Connections und in Zukunft CORBA. Darüberhinaus hat Java ein eigenes Modell zur Verteilung von Java–Objekten (Remote Method Invocation - RMI).

interpretiert

Wie in Smalltalk erzeugt der Compiler keinen Maschinencode, sondern einen Zwischencode (Bytecode), der von einer virtuellen Maschine (JVM) *interpretiert* wird. Durch die Verwendung dieser JVM ist der erzeugte Code in Bezug auf die zugrundeliegende Hardware *architekturneutral* und *portabel*.

architekturneutral
portabel

robust

Robustheit erreicht Java durch eine statische Typüberprüfung, eine automatische Speicherverwaltung (garbage collection) und den Verzicht auf Zeigerarithmetik, welche in C++ ständige Ursache von Problemen war.

sicher

Applets sind kleine Programme, die über einen Web–Browser aufgerufen werden und auf dem Client laufen. Um die *Sicherheit* eines solchen Programmes zu gewährleisten, laufen sie in einer *Sandbox* ab. Eine Sandbox ist eine Ablaufumgebung eines Programms, die ein ablaufendes Programm so überwacht, daß nur erlaubte Operationen ausgeführt werden können, weshalb Java über ein ausgefeiltes Sicherheitsmodell verfügt.

performant

Just–in–Time Compiler und in Zukunft die *Hot Spot*–Technologie sorgen für eine *hohe Performanz* bei der Interpretierung des erzeugten Bytecodes. Falls die Anforderungen an die Ausführungsgeschwindigkeit sehr hoch sind, kann der Zwischencode vor Ausführung des Programms vollständig bis auf Maschinenebene übersetzt werden.

multithreaded

Java definiert ein einfaches *API*, um Threads erzeugen zu können bzw. zu synchronisieren. Die *Threadunterstützung* ist in die Sprache eingebaut und daher aus Programmierersicht unabhängig von der zugrundeliegenden Betriebssystemplattform.

dynamisch

Durch bedarfsgesteuertes Nachladen von Klassen zur Laufzeit entwickelt ein Java–Programm eine hohe *Dynamik*. Die zunächst kleine Zahl an geladenen Klassen wächst zur Laufzeit eines Programms immer weiter an.

In Java ist ein *Package–Mechanismus* definiert, durch den Klassen zusammengefaßt werden können. Dieser Mechanismus verhindert Konflikte im Namensraum und unterstützt die Strukturierung der Sichtbarkeit von Klassen, Methoden und Variablen.

Klassenbibliothek

Zu Java gehört neben der eigentlichen Sprachdefinition eine *umfangreiche Klassenbibliothek*. Diese bildet ein *Framework* zur Applikations- bzw. zur Applet-Entwicklung und bietet eine große Zahl an APIs. Seit der Vorstellung von Java erfuhren sowohl die Klassenbibliothek als auch die Sprache selbst zum Teil tiefgreifende Veränderungen. Durch die Standardisierung ist eine Stabilisierung der Sprachdefinition zu erwarten.

Entwicklungsumgebungen

JDK

Sun stellt das Java Development Kit (JDK) als kostenlose Entwicklungsumgebung zur Verfügung. Es besteht im wesentlichen aus einem Compiler, einer virtuellen Maschine (JVM) und einer inzwischen sehr umfangreichen Klassenbibliothek. Diese Entwicklungsumgebung bietet keinen großen Komfort, ist aber fast als Referenzimplementierung zu betrachten. Der vom Java–Compiler erzeugte Zwischencode ist auf sämtlichen Hardwareplattformen lauffähig, für die eine virtuelle Maschine existiert.

Neben der von Sun kostenlos zur Verfügung gestellten Entwicklungsumgebung gibt es eine Reihe kommerzieller Entwicklungsumgebungen. Diese zeichnen sich in der Regel durch eine größere Benutzerfreundlichkeit aus (Bsp. IBM VisualAge for Java). Daneben existieren nicht–kommerzielle Implementierungen des JDK bzw. der Entwicklungswerkzeuge des JDK (Bsp: Kaffee).

Da sich die vorliegende Arbeit auf einen Vergleich der Sprachplattformen (in Form der Sprache selbst, der Laufzeitumgebung und der Klassenbibliothek) konzentriert, steht das JDK im Mittelpunkt der Untersuchung.

2.3 Wiederverwendung

Ziel von Wiederverwendung

Ein häufig angestrebtes Ziel bei der Entwicklung von Software ist die Wiederverwendung von früher erzielten Ergebnissen. Seit den siebziger Jahren drückt der Begriff Software–Krise aus, wie schwierig es ist, qualitativ hochwertige Software mit vertretbarem Aufwand in einem akzeptablen Zeitrahmen zu entwickeln [Cox90]. Mit der Technik der Wiederverwendung wollte man der Software-Krise etwas entgegensetzen.

Zunächst wurde hierbei lediglich an die Wiederverwendung von bestehendem Programmcode gedacht. Ein Kopieren des Codes und Anpassen an die neuen Bedürfnisse („copy and patch“) ist jedoch eine wenig zufriedenstellende Lösung. Ein weitergehender Ansatz stellt Bibliotheken vorgefertigter Unterroutinen bereit, die Lösungen für immer wieder auftretende Teilprobleme sind. Sie nehmen dem Programmierer die Arbeit ab, immer wieder die gleichen Programmteile implementieren zu müssen.

Objektorientierung ist angetreten, einige der Probleme, die bei der Wiederverwendung auftreten, zu lösen. Jacobson et al. schreiben hierzu:

> Problems with reuse include the finding, understanding and appropriateness of the thing to be reused. Object-orientation gives a completely new technique that strongly supports these issues. [JCJO92, S.27]

Code reuse

Zwei sehr häufig eingesetzte Mechanismen um Wiederverwendung zu realisieren sind Vererbung und Komposition [GHJV95, S.18]. Wiederverwendung beginnt schon auf der Applikationsebene. Hier hilft interne Wiederverwendung, eine Applikation möglichst einfach, wartbar und erweiterbar zu halten.

Auch eine objektorientierte Anwendung wird auf eine Sammlung von Lösungen für Teilprobleme zurückgreifen. Die objektorientierte Terminologie spricht hier von Toolkit [GHJV95, S.26] im Gegensatz zu Unterprogramm–Bibliothek bei prozeduralen Sprachen.

Design reuse

Objektorientierung macht bei der Wiederverwendung von Programmcode jedoch nicht halt sondern versucht, Design wiederzuverwenden. Hierzu existieren objektorientierte Frameworks (siehe 2.5.2). Ein Framework ist eine Menge kooperierender Klassen, die ein wiederverwendbares Design für eine bestimmte Klasse an Software darstellen [Deu89, JF88]. Ein Beispiel für ein Framework ist das Model-View-Controller-Konzept (MVC) von Smalltalk zur Gestaltung interaktiver graphischer Applikationen [KP88].

Frameworks sind eng mit Mustern verküpft (2.4). Einerseits kann ein Framework selbst als ein Muster auf Architekturebene angesehen werden [BMR+96]. Andererseits werden Entwurfsmuster eingesetzt um ein Framework zu beschreiben [Joh92], zu entwickeln [BJ94] und zu implementieren [Sch97]. Frameworks sind für die Anwendungsentwicklung extrem wichtig. Der größte Teil des Designs und des Codes in einer Anwendung wird aus dem Framework stammen oder durch das Framework beeinflußt sein [GHJV95, S.28]. Ein Nachteil der Applikationsentwicklung mit Hilfe von Frameworks ist, daß die Applikation auf das Framework angewiesen ist. Die Portierung auf eine Plattform, auf der das Framework nicht zur Verfügung steht, kostet vermutlich den Aufwand, der bei der Entwicklung durch den Einsatz des Frameworks eingespart wurde [Kap95, S.91].

2.4 Muster

Muster sind Beschreibungen von Lösungen immer wiederkehrender Probleme, die bei der Entwicklung von Anwendungen auftreten. Muster stellen dabei keine komplette Designmethode dar, sondern decken wichtige Praktiken existierender Methoden auf [Cop96, S.3]. Ein Muster ist kein

Stück Programmcode, das unverändert in ein eigenes Programm übernommen werden kann, sondern eine verallgemeinerte Lösung — eingebettet in den Problemkontext und die Konsequenzen, die der Einsatz des Musters hat.

Beschreibung

Muster zeichnen sich durch eine strukturierte Beschreibung aus, die über eine Sammlung an Mustern hinweg konstant ist. Eine graphische Darstellung von Klassen, die das Muster realisieren bzw. ein kurzes Programmstück ist zur Beschreibung von Mustern nicht ausreichend. Eine Darstellung des Problemumfeldes, der Alternativen und eine Begründung für eine spezifische Designentscheidung gehören ebenfalls zur Beschreibung des Musters.

Musterhierarchie

Nach Buschmann et al. [BMR+96] existieren Muster auf verschiedenen Ebenen (Muster–System). Dieser Abschnitt stellt die verschiedenen Ebenen der Idiome, Entwurfsmuster und Architekturmuster vor.

Architekturmuster

Auf höchster Abstraktionsebene existieren **Architekturmuster**. Architekturmuster drücken fundamentale strukturelle Organisationsschemata eines Softwaresystems aus. Ein Architekturmuster definiert eine Anzahl an Subsystemen, sowie deren Aufgaben und Regeln für die Gestaltung der Beziehungen zwischen diesen Subsystemen. Obwohl Architekturmuster auf höchster Ebene die Struktur eines Softwaresystems bestimmen, wird nur in den seltensten Fällen die Verwendung ausschließlich eines einzigen Architekturmusters ausreichen. Nur die Zusammenarbeit mehrerer Architekturmuster kann den verschiedenen Anforderungen an ein System gerecht werden. Allerdings beschreibt auch eine Kombination von Architekturmustern noch keine komplette Softwarearchitektur. Diese Kombination stellt lediglich ein strukturelles Framework dar, welches noch weiter spezifiziert werden muß. Der Anschluß an die eigentliche Applikationslogik und das Zusammenspiel der Komponenten muß noch detaillierter dargestellt werden. Dies geschieht oft mit Hilfe von Entwurfsmustern und Idiomen.

> An architectural framework expresses a fundamental paradigm for structuring software systems. It provides a set of predefined subsystems, as well as rules and guidelines for organizing the relationships between them. [BM95, S.329]

Das strukturelle Framework, welches durch die Architekturmuster definiert wird, findet sich in Smalltalk und Java als konkretes Framework in Form einer Klassenbibliothek wieder. Dieses Framework ist insofern konkret, als daß die Verwendung der Klassen und Objekte des Frameworks zu einer bestimmten Architektur der Applikation führen. Ein Framework ist das programmiersprachliche Mittel, um strukturelles Design wiederverwenden zu können. Architekturmuster haben von den hier beschriebenen Mustern den größten Einfluß auf die Architektur einer Applikation.

Entwurfsmuster

Eine Ebene tiefer sind **Entwurfsmuster** angeordnet. Entwurfsmuster definieren Klassen und Objekte und geben Regeln für die Gestaltung der Beziehungen zwischen diesen Elementen. Sie haben einen großen Einfluß auf die Architektur eines Subsystems — weniger auf die Architektur des Gesamtsystems. Entwurfsmuster sind unabhängig von einem bestimmten Problembereich und von einer spezifischen Implementierung. Allerdings ist die Beschreibung eines Entwurfsmusters manchmal stark von einer bestimmten Implementierungssprache geprägt. Die Beschreibungen der Entwurfsmuster in [GHJV95] sind sehr stark von C++ beeinflußt. In Smalltalk eröffnen die anderen sprachlichen Ausdrucksmöglichkeiten eine zum Teil sehr verschiedene Beschreibung [ABW98].

> The design patterns [...] are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. [GHJV95, S.3]

Entwurfsmuster treten einerseits als strukturierende Merkmale der Sprachplattform auf. Eine Applikation verwendet dieses Design, um sich auf natürliche Weise in das Framework einzufügen. Andererseits bringt die Applikation ihre eigenen Entwurfsmuster mit, die sie zur Strukturierung bestimmter Teilaspekte verwendet. Die Applikation sucht in der Sprachplattform lediglich die Elemente zur Implementierung dieser Entwurfsmuster.

Idiome

Idiome sind Muster auf der niedrigsten Ebene der Muster–Hierarchie. Im Gegensatz zu Entwurfsmustern beschreiben sie konkrete Implementierungen spezifischer Problemlösungen in einer bestimmten Programmiersprache. Die Übergänge von Idiomen zu Entwurfsmustern einerseits und zu Beschreibungen eines Programmierstils andererseits sind fließend. Im Gegensatz zu Programmierstil zeichnen sich Idiome durch die Art der Beschreibung in Form eines Musters und durch eine stärkere Fokusierung auf das Lösen von konkreten Problemen aus. Im Gegensatz zu Entwurfsmustern zeichnen sich Idiome durch eine enge Bindung an die verwendete Implementierungssprache bzw. Plattform aus. Einige der mit Idiomen gelösten Probleme sind so speziell für die verwendete Sprache, daß sich kein verallgemeinertes Problemlösungsschema in Form eines Entwurfsmusters finden ließe.

> An idiom describes how to implement particular components (parts) of a pattern, the components functionality, or their relationships to other components within a given design. They often are specific for a particular programming language. [BM95, S.330]

Idiome sind die unterste Ebene der Verwendung der Elemente einer Sprachplattform. In einer Applikation ist die Verwendung einer bestimmten Plattform durch den Einsatz der spezifischen Idiome gekennzeichnet. Da die Implementierungen der Klassen der Sprachplattform selbst die Plattform verwenden, setzen auch sie sich aus den spezifischen Idiomen zusammen.

2.5 Softwarearchitektur

Softwarearchitektur ist die Beschreibung der Elemente, aus denen ein System aufgebaut ist und der möglichen Interaktionen zwischen diesen Elementen. Darüberhinaus beinhaltet Softwarearchitektur Regeln, wie die Elemente zusammengefügt werden können und hierbei geltende Einschränkungen [SG96, S.1].

Dieser Abschnitt konzentriert sich auf die Aspekte der Softwarearchitektur, die in Bezug auf objektorientierte Softwareentwicklung interessant sind. Dies schließt natürlich Aspekte ein, die auch für andere Architekturstile als Objektorientierung wichtig sind.

2.5.1 Anwendungsschichten und –stufen

Üblicherweise besteht eine Applikation nicht aus einem einzigen monolithischen Block, sondern weist eine innere Struktur auf. Dieser Abschnitt behandelt die logische Schichtung und die Einteilung in physikalische Stufen.

Schichten–Architektur

Schichtung bezeichnet die logische Aufteilung einer Applikation. Jede Schicht ist durch ihre eigenen Zuständigkeiten charakterisiert und weist eine wohldefinierte Schnittstelle zu den umgebenden Schichten auf. Abbildung 2.1 zeigt eine typische Aufteilung einer Applikation in mehrere Schichten:

Domänenobjekt-modell

Das **Domänenobjektmodell** enthält die Objekte der *Problemdomäne*. Dies sind die Entitäten, die durch die Analyse mit Hilfe von Fachexperten gefunden wurden bzw. entstammen dem Anforderungsmodell [JCJO92].

Applikationsmodell

Das **Applikationsmodell** beschreibt, auf welche Art der Benutzer auf das Domänenobjektmodell zugreifen kann. Es stellt die Domänenobjekte zusammen, die der Benutzer für eine bestimmte

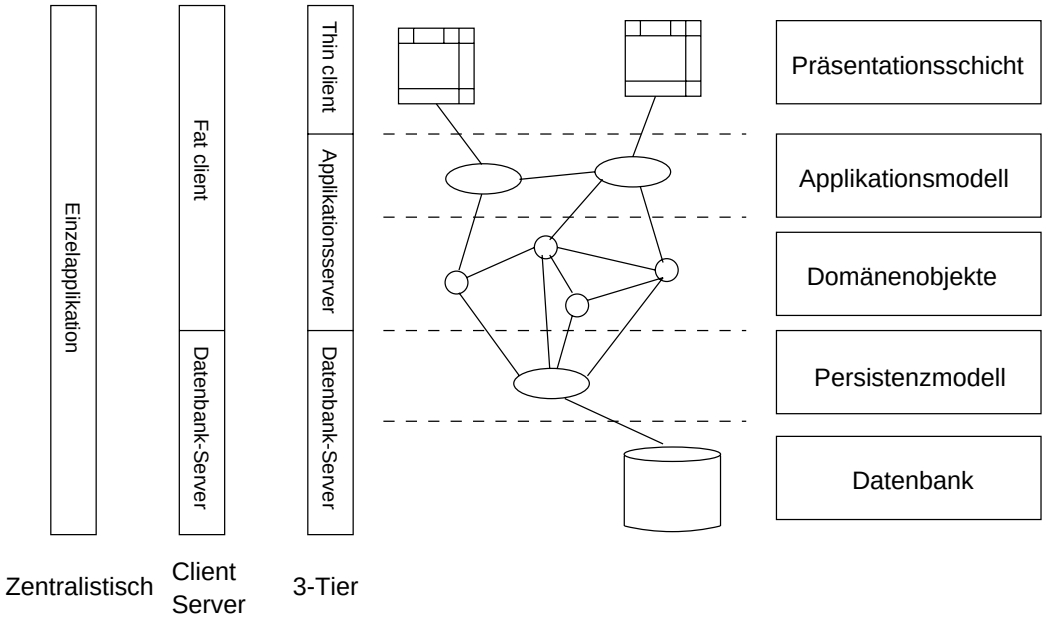


Abbildung 2.1: Schichten– und Stufen–Architektur einer Applikation

Interaktion mit der Applikation benötigt, besorgt das Zusammenspiel der einzelnen Benutzerinteraktionen und kümmert sich um die Zusammenarbeit mit dem Persistenzmodell.

Präsentationsschicht

Die graphische Benutzeroberfläche ist in der **Präsentationsschicht** definiert. Jeder Benutzerdialog arbeitet direkt mit einem Objekt des Applikationsmodells zusammen, welches die relevanten Domänenobjekte zur Verfügung stellt, bzw. die Daten des Benutzerdialoges nach Abschluß einer Aktion des Benutzers entgegennimmt. Reine Dialogsteuerung geschieht ausschließlich in der Präsentationsschicht. Die Abspaltung der Benutzerdialoge vom Applikationsmodell erlaubt es, beide Modelle unabhängig voneinander zu verändern, ohne gleich das Gesamtsystem umstellen zu müssen.

Persistenzmodell

Das **Persistenzmodell** definiert, wie persistente Modelle abgespeichert werden. Die hier betrachtete Applikation verwendet eine relationale Datenbank zur Speicherung solcher Objekte, welche eine Abbildung der Objekte des Domänenobjektmodells auf die Relationen der Datenbank erfordert. Dies ist Aufgabe der objektrelationalen Abbildung, welche zwischen dem Domänenobjektmodell und der eigentlichen Datenbank vermittelt. Üblicherweise enthält diese Schicht ein Transaktionsmodell für die persistenten Objekte, um die Konsistenz der Domänenobjekte gewährleisten zu können.

Datenbank

Die **Datenbank** speichert die persistenten Objekte des Domänenobjektmodells. Diese kann auf dreierlei Weise realisiert werden: als objektorientierte Datenbank, als relationale Datenbank mit einer objektrelationalen Abbildung oder als objektrelationale Datenbank [VC97]. In unserem Beispiel speichert die Datenbank die Objekte in Form von Relationen. Die Applikation greift nicht direkt auf die Datenbank zu, sondern ausschließlich über die Schicht der objekt–relationalen Abbildung.

Stufen–Architektur

Parallel zur Einteilung einer Applikation in logische Schichten liegt die physikalische **Stufung** einer Applikation. Mit Stufung ist die Verteilung einer Applikation auf mehrere miteinander vernetzte

Rechner gemeint. Im Folgenden werden die verschiedenen Paradigmen vorgestellt, nach denen eine Applikation in mehrere Stufen eingeteilt werden kann.

Einzelapplikation

Eine *Einzelapplikation* ist die zentralistische Variante der Stufen–Architektur. In diesem Fall liegt eigentlich keine Stufung vor. Alle Applikationsteile laufen auf einem einzigen Rechner.

Client/Server

Das *Client/Server–Paradigma* verteilt die Rechenleistung auf einen zentralen Server und viele dezentrale Clients, die über ein Netzwerk mit dem Server verbunden sind. Diese Lösung hat den Vorteil, daß große Teile der Applikation direkt beim Benutzer sind, dem dadurch mehr Freiheiten bei der Gestaltung seiner Arbeitsumgebung gegeben werden.

3–tier Architektur

Die *3–tier–Architektur* führt eine dritte Stufe ein: einen oder mehrere Applikationsserver. Die Applikationsserver kommunizieren mit den abgespeckten Clients, die in dieser Architektur ausschließlich für die Präsentation und die Dialogsteuerung zuständig sind und kapseln damit den Zugriff auf den zentralen Datenbankserver. Diese weitere Stufe reduziert die Auslastung des Datenbankservers deutlich, weil er nur noch für die Applikationsserver zuständig ist und nicht mehr für die Gesamtzahl der Clients. Zudem verbessert sich die Skalierbarkeit des Gesamtsystems.

n–tier–Architektur

Die *n–tier–Architektur* ist die Weiterführung der 3–tier–Architektur. Diese verteilt die Applikation auf noch mehr Stufen. Jede Stufe hat ihren eigenen Zuständigkeitsbereich wie Transaktionsmanagement, Benutzerauthentifizierung, Lastausgleich usw. Die einzelnen Stufen sind nicht mehr notwendigerweise auf verschiedene Rechner verteilt, sondern können sich einen gemeinsamen Netzwerkknoten teilen. Damit verschwimmt die Grenze zwischen Schichtarchitektur und Stufenarchitektur einer Applikation.

2.5.2 Frameworks

Ein **Framework** ist eine Technik, um Wiederverwendung in objektorientierten Programmiersprachen zu realisieren. Es gibt verschiedene Definitionen des Begriffs Framework. Nach [Deu89] ist ein Framework ein wiederverwendbares Design des gesamten oder eines Teils eines Systems, das repräsentiert wird durch eine Menge an abstrakten Klassen und die Art, wie ihre Instanzen interagieren. Nach einer anderen Sichtweise ist ein Framework das Skelett einer Applikation, das durch den Applikationsentwickler angepaßt wird [Joh97].

Ein Framework enthält also die Elemente, die allen Applikationen einer bestimmten Domäne gemein sind. Um eine Applikation dieser Domäne zu schreiben, fängt man nicht bei Null an, sondern erweitert ein bestehendes Framework um die applikationsspezifischen Teile. Neue Funktionalität wird in das Framework eingebaut und erweitert die bestehende Funktionalität. Die Architektur der zu erstellenden Applikation ist eine Realisierung der generischen Architektur, die das Framework vorgibt [Joh97].

Die Wiederverwendung von Frameworks geschieht durch Vererbung und Instanziierung [Sch97]. Abbildung 2.2 zeigt schematisch beide Formen der Wiederverwendung.

Wiederverwendung durch Vererbung

Das Framework gibt abstrakte und konkrete Klassen vor. Die abstrakten Klassen repräsentieren das wiederverwendbare Design. Von diesen abstrakten Klassen liegen oft Standardimplementierungen in Form von konkreten Klassen vor. Die applikationsspezifische Funktionalität steckt in Implementierungen der abstrakten Klassen bzw. in Subklassen, die von den Standardimplementierungen erben und deren Funktionalität erweitern oder überschreiben. Diese Form der Wiederverwendung heißt *white box reuse* [GHJV95, S.26].

Wiederverwendung durch Instanziierung

Bei *black box reuse* verwendet die Applikation die vom Framework angebotenen Komponenten ausschließlich über die externe Schnittstelle — nicht durch Vererbung. Für diese Art der Wiederverwendung enthält die Applikation nur „glue code“, um die Einzelteile zusammenzusetzen.

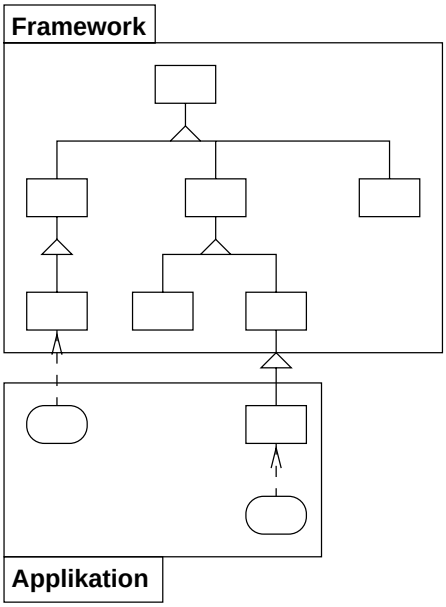


Abbildung 2.2: Framework–Verwendung

3

Thema der Arbeit

Das Thema dieser Arbeit ist ein Vergleich der Applikationsarchitekturen von Smalltalk und Java im Hinblick auf Migration von Smalltalk nach Java.

Hintergrund dieser Fragestellung ist der Wunsch, eine bestehende Smalltalk–Applikation nach Java zu migrieren. Diese Migration soll dabei mehr bieten als eine reine syntaktische Übersetzung des Quellcodes. Damit sich die migrierte Applikation nahtlos in die Zielplattform einfügt, ist die Übertragung der Designideen, die hinter einer spezifischen Smalltalk–Implementierung stecken, von besonderer Bedeutung. Die migrierte Applikation soll die Elemente von Java so verwenden, wie es dem Stil von Java entspricht. Dadurch soll die Lesbarkeit und Wartbarkeit nach der Migration erhalten bleiben.

Unter Architektur einer Applikation verstehen wir im Rahmen dieser Untersuchung die Kombination von Mustern auf den verschiedenen Abstraktionsebenen. Jedes Muster trägt als „Mikroarchitektur“ zur Gesamtarchitektur bei. Die Gesamtarchitektur der Applikation ergibt sich aus der Summe der eingesetzten bzw. verwendeten Muster.

> Moreover, applications that use frameworks must conform to the frameworks' design and model of collaboration, so the framework causes patterns in the applications that use it. [Joh97]

Um die Fragestellung genauer zu spezifizieren, beschreibt dieses Kapitel zunächst die verwendete Plattformdefinition. Es zeigt die Faktoren auf, die Einfluß auf die Applikationsarchitektur nehmen. Anschließend grenzt es den Untersuchungsgegenstand dieser Arbeit ab und beschreibt den Gang der Untersuchung.

3.1 Plattformdefinition

Dieser Abschnitt beschreibt die Aufteilung eines objektorientierten Systems und definiert anschliessend den Begriff Sprachplattform (vgl. [Lew95, S.19]).

Ein objektorientiertes System teilt sich in vier Komponenten auf: die Sprache selbst, die Klassenbibliothek, die Laufzeitumgebung und die Entwicklungswerkzeuge (siehe Abbildung 3.1).

In Smalltalk ist die Sprache selbst über die verschiedenen Dialekte hinweg fast vollständig portabel gestaltet, die Klassenbibliothek zu einem großen Teil. Das „Blue Book“ [GR83] definiert die Sprache und große Teile der Klassenbibliothek. Es ist die Basis für die meisten Smalltalk–Dialekte. Darüberhinaus definiert jeder Dialekt eigene Klassen, die die Klassenbibliothek ergänzen. Die Entwicklungswerkzeuge bestimmen, wie der Programmierer auf die Sprache und die Klassenbibliothek

Abbildung 3.1: Komponenten eines OO Systems

zugreifen kann. Die verschiedenen Dialekte haben alle in der Funktionalität vergleichbare Entwicklungswerkzeuge (Browser, Debugger, Inspector, etc.), im Detail unterscheiden sich diese Grundkomponenten jedoch wesentlich. Die Laufzeitumgebung stellt die Dienste bereit, die ein Programm bei der Ausführung benötigt.

Von Java gibt es (noch) keine Dialekte[1]. Die Sprache, die Laufzeitumgebung und die Klassenbibliothek bilden die Basis, die über die verschiedenen Java–Systeme hinweg identisch ist. Die Entwicklungsumgebungen hingegen weisen eine sehr große Variationsbreite auf. Angefangen von der Kommandozeilen–Version bis zu einer Browser–basierten Smalltalk–ähnlichen Entwicklungsumgebung ist alles vorhanden.

Im Rahmen dieses Vergleichs betrachten wir die Sprache, Laufzeitumgebung und Klassenbibliothek von Java und einer spezifischen Smalltalk–Implementierung. Diese Komponenten fassen wir unter dem Begriff **Sprachplattform** zusammen.

3.1.1 Betrachtete Plattformen

Im Falle von Smalltalk betrachten wir neben dem ANSI–Standard und dem „Blue Book“ [GR83] das Produkt IBM Smalltalk in der Version 4.0. Dieser Dialekt wird zusammen mit der visuellen Entwicklungsumgebung IBM VisualAge for Smalltalk ausgeliefert. Das hierin definierte Komponentenmodell und der Mechanismus der visuellen Programmierung sind derart speziell für diesen Dialekt, daß diese Teile im Rahmen des Vergleiches außen vor bleiben.

Im Falle von Java betrachten wir das SUN JDK 1.1, da die Version 1.2 zum Zeitpunkt der Erstellung der Arbeit nur als Beta–Version vorlag. Hinzu kommen Swing und die Collection–API. Swing ist eine Sammlung neuer GUI–Elemente, welche die Elemente des Abstract Windowing Toolit (AWT) der Version 1.1 ersetzen soll. Swing ist aufgrund des verwendeten *Model–View–Controller*–Konzepts ein moderneres und flexibleres GUI–Framework als das ursprüngliche AWT. Die Collection–API ist ein Subframework zur Erstellung von Behälter–Klassen und enthält schon einige Collection–Implementierungen.

[1] Die Firma Microsoft bietet zwar einen Dialekt von Java an, der nicht 100% kompatibel ist, wird aber wegen der Verwendung des Namens „Java“ zur Zeit gerichtlich belangt.

3.2 Einflüsse auf die Applikationsarchitektur

Die Architektur einer Applikation unterliegt verschiedenen Einflüssen. Von diesen Einflüssen betrachten wir im Rahmen dieser Arbeit die Sprachplattform und die dadurch in der Applikation induzierten Muster. Abbildung 3.2 zeigt die Faktoren, die Einfluß auf die Architektur einer Applikation nehmen:

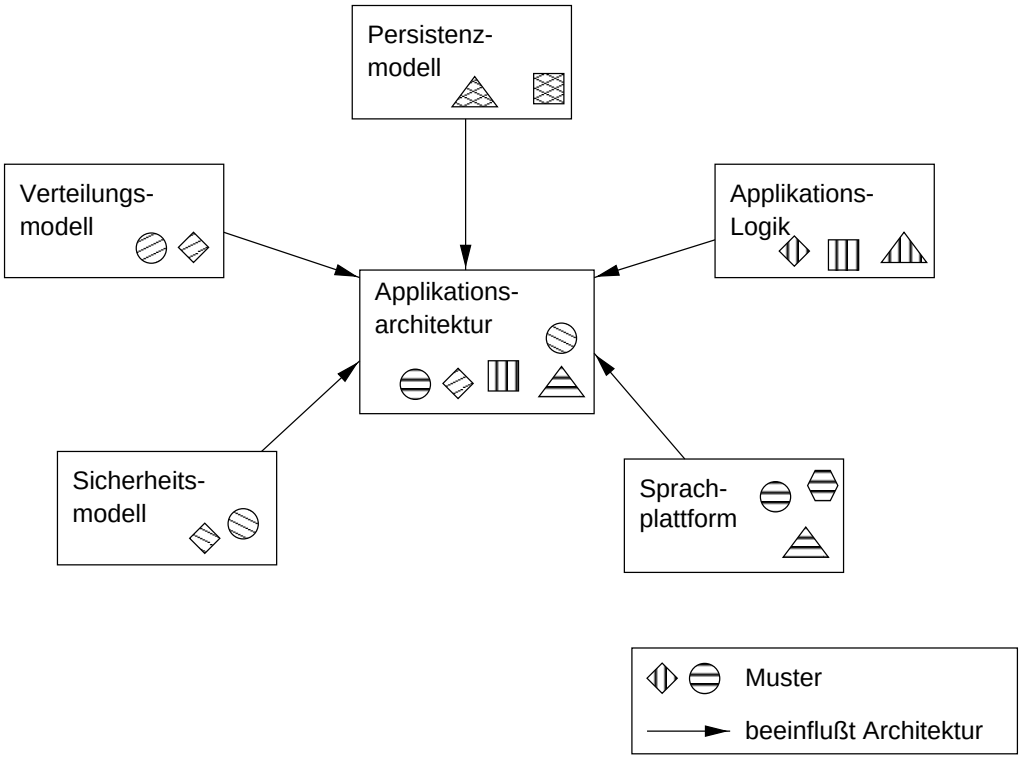


Abbildung 3.2: Einflußfaktoren auf Applikationsarchitektur

Modelle

Die *Applikationslogik* ist die Basis für die Applikationsarchitektur (vgl. Analysemodell in [JCJO92, S.130]). Es ist eine Umsetzung der Ergebnisse der Anforderungsanalyse in eine implementierbare Form. Es besitzt eine gegenüber Änderungen robuste Struktur, welche implementationsunabhängig gestaltet ist. Die Einflüsse der jeweiligen Programmiersprache bzw. Sprachplattform werden erst in den nachfolgenden Modellen sichtbar. Aus diesem Grund brauchen wir den Einfluß der Applikationslogik im Rahmen dieser Arbeit nicht zu betrachten.

Das *Verteilungsmodell* beschreibt die Verteilung der Applikationslogik auf mehrere vernetzte Rechner [FRS96, S.44]. Die Gründe für die Verteilung sind unter anderem Skalierbarkeit, Performance und Fehlertoleranz. Der Einfluß des Verteilungsmodells auf die Applikation ist zunächst unabhängig von der verwendeten Programmiersprache, weshalb es im Rahmen dieser Arbeit nicht weiter betrachtet wird.

Das *Sicherheitsmodell* beschreibt die Mechanismen, wie ein Programm auf Ressourcen zugreifen kann und welchen Einschränkungen es dabei unterliegt. Insbesondere durch die ansteigende

Vernetzungsdichte und zunehmende Vernetzung lokaler Netzwerke mit öffentlichen Netzen erlangt das Thema Sicherheit zunehmend Bedeutung [FRS96, S.49]. Die Klassenbibliothek von Smalltalk enthält keinen Mechanismus, der explizit zur Realisierung eines Sicherheitsmodells vorgesehen ist. Daher behandeln wir diesen Aspekt hier nicht.

Das *Persistenzmodell* sorgt für die Eigenschaften eines Objekts, seine Lebensdauer über die Lebensdauer der verwendenden Applikation hinaus auszudehnen [Boo94]. Dazu gehört die Form, in der das Objekt abgespeichert wird und die Abbildung aus dem Objektmodell auf diese persistente Form. Zwar wird der Anschluß an die Datenbank von beiden Klassenbibliotheken unterstützt, jedoch kein Mechanismus zur objektrelationalen Abbildung. Daher betrachten wir die Einflüsse dieses Modells nicht weiter.

Die *Sprachplattform* enthält diejenigen Komponenten, aus denen sich der Applikationscode zusammensetzt. Diese Elemente der Plattform sind jedoch mehr als eine Aneinanderreihung von Bausteinen zum Aufbau einer Applikation. In großen Teilen repräsentieren sie ein wiederverwendbares Design [Joh97], welches die Architektur der Applikation diktiert [GHJV95, S.26]. Die Untersuchung der Sprachplattformen ist folglich ein zentrales Thema dieser Arbeit.

Muster

All diesen Modellen ist gemein, daß sie bestimmte Muster verwenden. Muster sind wiederverwendbare Designs, welche auf den Ebenen der Idiome, der Entwurfsmuster und der Architekturmuster auftreten. Sie decken einen Bereich ab, der von der Strukturierung einzelner Implementierungsdetails bis zur Gesamtarchitektur einer Applikation reicht. Weil diese Muster je nach Sprachplattform ein anderes Erscheinungsbild haben und Einfluß auf die Architektur der Applikation nehmen (vgl. [BMR+96, S.22]), sind sie das zweite zentrale Thema dieser Arbeit.

3.2.1 Abgrenzung der Einflußfaktoren

Die Motivation hinter dieser Arbeit ist der Wunsch, eine Applikation von Smalltalk nach Java unter Beibehaltung der Funktionalität zu migrieren. Daher sind im Rahmen dieser Arbeit ausschließlich die Einflußfaktoren interessant, die eine hohe Implementationsabhängigkeit aufweisen. Dies sind zum einen die *Sprachplattform* selbst. Sie ist die eigentliche Quelle der Implementierungsabhängigkeit. Durch das enthaltene Framework nimmt sie gleichzeitig großen strukturierenden Einfluß auf die Applikation. Zum anderen sind dies die zum Einsatz kommenden *Muster*. Jedes der beteiligten Modelle prägt der Applikationsarchitektur seine spezifischen Muster auf. Muster sind vom Ansatz her sprachunabhängig, jedoch sind die Musterimplementierungen zum Teil sehr speziell auf die verwendete Sprachplattform zugeschnitten. Diese Implementierungsabhängigkeit ist für die Migration ein sehr wichtiges Thema.

3.3 Prämissen

In dieser Arbeit kann keine vollständige Antwort auf die Frage nach der Migrierbarkeit einer Applikation gegeben bzw. ein vollständiger Weg zur Migration einer Applikation von Smalltalk nach Java aufgezeigt werden. Hierzu müßte das Umfeld der Applikationsentwicklung in die Untersuchung miteinbezogen werden. Einige in diesem Zusammenhang zu erwähnenden Punkte sind das Fachwissen der Entwickler, die verwendeten Werkzeuge, der verwendete Entwicklungsprozeß und die vorhandene bzw. verfügbare Infrastruktur, etc. Diese Aspekte werden in dieser Arbeit nicht behandelt.

Das Thema dieser Arbeit ist die Untersuchung, welchen Einfluß die Migration einer Applikation von Smalltalk nach Java auf ihre Architektur hat. Dabei konzentrieren wir uns auf die Kernfunktionalität der Smalltalk–Plattform. Dies ist im wesentlichen die vom „Blue Book“ und vom

ANSI–Standard definierte Funktionalität und das GUI–Framework der untersuchten Smalltalk–Plattform.

Als weitere Einschränkung geht diese Arbeit davon aus, daß sich der Applikationscode nahezu vollständig aus Idiomen und Entwurfsmustern zusammensetzt bzw. die in der Klassenbibliothek vorhandenen Architekturmuster und Entwurfsmuster gemäß ihrer Intention verwendet. Idealerweise sollte eine Applikation aus diesen Mustern aufgebaut sein. Sie sind das Destillat langjähriger Entwicklungspraxis und stellen bewährtes Design dar.

3.4 Gliederung der Untersuchung

Die Untersuchung gliedert sich in folgende Schritte:

1. *Vergleich der Plattformelemente.* Ausgehend von der Struktur der Smalltalk–Plattform, wird jedes Plattformelement dahingehend untersucht, welches Element der Java–Plattform die gleiche oder annährend gleiche Funktionalität erfüllt. Dies soll aufzeigen, welches Plattformelement in Java statt des Smalltalk–Konstrukts bei der Migration zu verwenden ist. Zudem soll diese Betrachtung solche Plattformelemente herausstellen, für die in Java nur schwer eine Entsprechung zu finden ist.

2. *Migration von Mustern.* Anschließend werden typische Implementierungen in Smalltalk in Form von Mustern behandelt. Dieser Ansatz geht auch wieder von Smalltalk aus, allerdings steht diesmal das Programm im Mittelpunkt. Ein Programm verwendet Muster zum Lösen typischer Probleme oder setzt sich mehr oder weniger vollständig aus Mustern zusammen, die ihm von den verschiedenen Modellen aufgeprägt werden.

3. *Architekturunterschiede einer Applikation zwischen Smalltalk und Java.* Dieser Teil führt die Erkenntnisse der beiden vorherigen Schritte zusammen. Ziel ist die komprimierte Aussage über die Änderungen in der Architektur einer Smalltalk–Applikation, die nach Java übersetzt werden soll bzw. die Probleme, die die Migration erschweren oder der Migration sogar im Weg stehen können.

Als Hilfmittel für die Migration enthält der Anhang einen Migrationsindex. Dieser geht von den in Smalltalk–Programmen zu findenden Methoden, Klassen und sonstigen Sprachkonstrukten aus und gibt Hinweise auf möglicherweise beteiligte Plattformelemente bzw. Muster. Dies soll bei der Migration ein schnelles Auffinden der passenden Textstelle innerhalb der Arbeit ermöglichen.

4

Vergleich der Plattformen

In diesem Kapitel werden die beiden untersuchten Sprachplattformen miteinander auf den Ebenen Sprache, Toolkit und Framework verglichen.

4.0.1 Ebenen der Sprachplattform

Eine Sprachplattform teilt sich folgendermaßen auf:

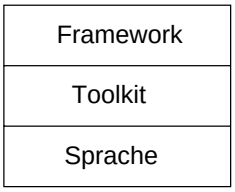


Abbildung 4.1: Ebenen einer Plattform

Die **Sprachebene** beschreibt die Sprache selbst und die Teile der Klassenbibliothek, die für die Definition der Sprache essentiell sind. Die Sprachebene bestimmt, wie ein Programm „aufgeschrieben“ wird und was es bedeutet (4.1). Die **Toolkit–Ebene**[1] besteht aus einer Sammlung von Hilfsklassen, die allgemein einsetzbare Funktionalität anbieten wie die Collection–Klassen (4.2). Die **Framework–Ebene**[2] besteht aus mehreren Frameworks, die sich aus kooperierenden Klassen zusammensetzen (4.3). Die Grenze zwischen Framework– und Toolkit–Ebene ist nicht immer klar zu fassen: Zum einen verwendet ein Framework die Klassen des Toolkits; zum anderen existieren Frameworks, um eigene Hilfsklassen zu bauen.

4.0.2 Vergleichskriterien

Der Plattformvergleich stellt, ausgehend von Smalltalk, die Elemente der Plattformen einzeln einander gegenüber. Wichtig ist das Erkennen der Unterschiede im Design dieser Elemente. Anschließend wird untersucht, wie eine Smalltalk–Implementierung eines Plattformelements nach Java migriert werden kann. Die migrierte Version soll sich auf natürliche Weise in die Java–Plattform

[1] Ein Toolkit ist das objektorientierte Gegenstück zu einer Unterprogrammbibliothek in einer prozeduralen Sprache (2.3).

[2] Ein Framework ist eine halbfertige Applikation, die die Applikationsentwicklung erleichtert (2.5.2).

einfügen und die Funktionalität der Plattform so nutzen, wie es der Plattformentwurf vorsieht. Damit ist eine Migration der Smalltalk–Basisklassen ausgeschlossen. Das migrierte Programm wäre ein Smalltalk–Programm in einer Java–Verkleidung.

Von besonderem Interesse sind die Probleme, die einer Migration im Weg stehen können. Insbesondere wenn eine Funktionalität in Java sehr viel anders realisiert wird bzw. überhaupt nicht vorhanden ist, ist das darzustellen.

Da der Vergleich im Hinblick auf Migration von Smalltalk nach Java unternommen wird, orientiert sich die Struktur der Arbeit an der Smalltalk–Plattform. Die Java–Plattform wird nicht vollständig behandelt, sondern nur insoweit als sie für die Migration von Belang ist. D.h. der Vergleich führt nur die Elemente der Java–Plattform auf, die zur Darstellung der Funktionalität der betrachteten Elemente der Smalltalk–Plattform benötigt werden.

4.0.3 Smalltalk–Plattform

Die Struktur der Smalltalk–Plattform orientiert sich am vorgeschlagenen Smalltalk–ANSI–Standard [ANS97], am Blue Book [GR83] und an der Klassenbibliothek von IBM Smalltalk. Abbildung 4.2 zeigt die in diesem Vergleich verwendete Strukturierung der Sprachplattform.

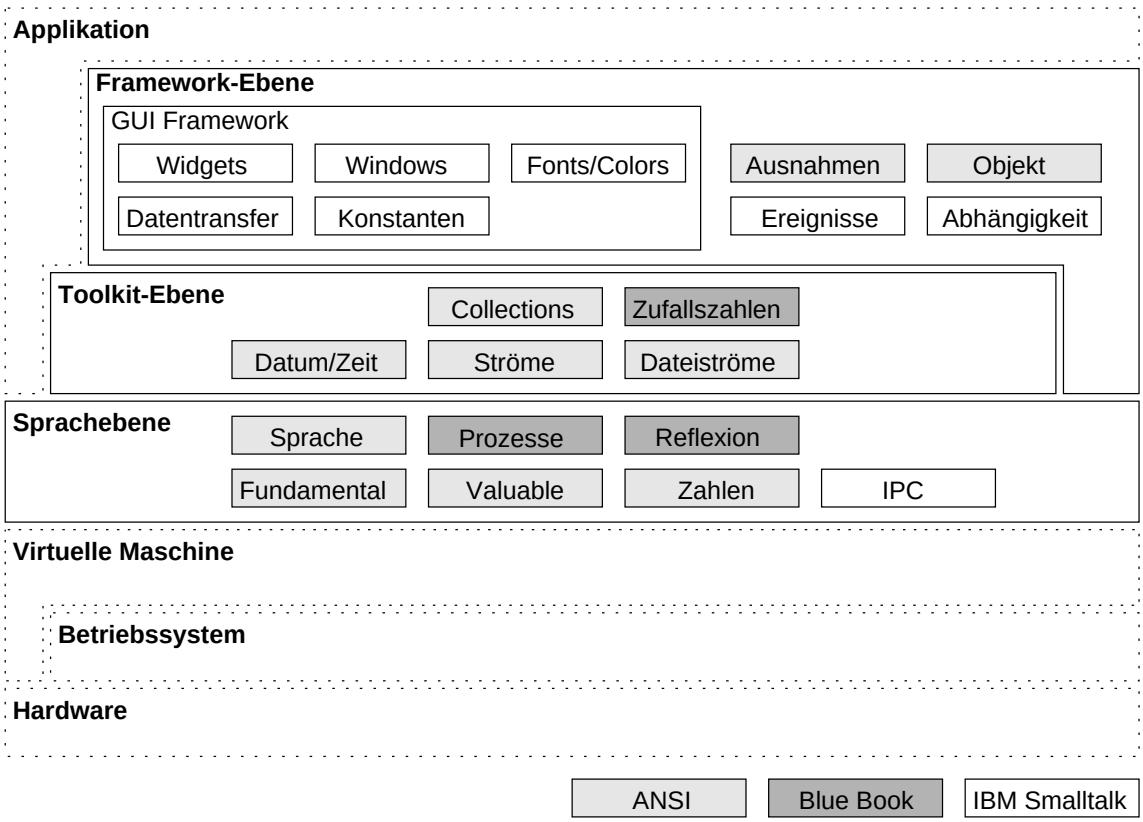


Abbildung 4.2: Struktur der Smalltalk–Plattform

Der ANSI–Standard umfaßt die Elemente von Smalltalk, die auf allen existierenden Plattformen zu finden sind.

> The scope of the library is roughly an intersection of existing Smalltalk products' libraries. That is, it includes numbers, data structures (collections), basic objects (nil, booleans, etc.), blocks, exceptions and files. [ANS97, S.5]

Der ANSI–Standard definiert keine konkreten Klassen, sondern lediglich Protokolle. Über eine mögliche Implementierung dieser Protokolle macht der Standard keine Aussage. Diese Protokolle werden zu Protokollklassen zusammengefaßt. Im Rahmen der vorliegenden Arbeit werden diese Protokollklassen zur Strukturierung der Klassenbibliothek verwendet. In Abbildung 4.2 sind dies die hellgrau hinterlegten Elemente.

Darüberhinaus definiert das „Blue Book" [GR83] weitere Protokollklassen, die ebenfalls in den meisen Smalltalk–Dialekten vorhanden sind. In der Abbildung sind sie dunkelgrau hinterlegt.

Die restlichen Elemente sind spezifisch für IBM Smalltalk. Hier sind die größten Unterschiede zu anderen Smalltalk–Plattformen zu finden, weshalb der Vergleich mit Java auf diesen Ebenen sehr IBM Smalltalk–spezifisch ist.

4.0.4 Virtuelle Maschine

Beide Sprachen verwenden eine virtuelle Maschine zum Ausführen des übersetzten Zwischencodes. Die virtuelle Maschine abstrahiert von der Hardware und ermöglicht so ein Ablaufen der übersetzten Programme auf jeder Hardware, die über eine virtuelle Maschine verfügt. Sie stellt die Verbindung zum Betriebssystem dar und ist damit die Basis der Laufzeitumgebung . Die virtuelle Maschine wird ausführlicher in 5.3.1 behandelt.

Die virtuellen Maschinen beider Sprachen enthalten eine automatische Speicherverwaltung, die nicht mehr benötigte Objekte automatisch freigibt. Dadurch entfällt für die Applikation die Notwendigkeit, diese Verwaltung selbst übernehmen zu müssen. Eine solche Verwaltung hätte starken Einfluß auf den Entwurf einer Applikation. Zudem enthält die virtuelle Maschine den Mechanismus zur Parallelausführung (4.1.5).

In der vorliegenden Arbeit werden die virtuellen Maschinen nicht weiter untersucht.

4.1 Vergleich auf Sprachebene

Im Vergleich auf Sprachebene wird die Sprache selbst untersucht und diejenigen Klassen der Klassenbibliothek, die für die Sprache von essentieller Bedeutung sind. Diese Klassen werden auch als *Kernel–Klassen* bezeichnet [Lew95, S.63]. In Java sind dies vorwiegend die Klassen im Package `java.lang`. In IBM Smalltalk sind dies die Klassen der Subsysteme *Common Language Data Types* und *Common Language Implementation*. Diese Aufteilung in Subsysteme ist allerdings spezifisch für IBM Smalltalk.

4.1.1 Sprache

Dieser Abschnitt untersucht die eigentliche Sprache. Wir betrachten der Reihe nach die in der Abbildung aufgeführten Teilaspekte.

Datentypen

Ein Datentyp definiert den Wertebereich einer Variablen und die Operationen, welche darauf ausgeführt werden können. An dieser Stelle heben wir auf die Unterscheidung zwischen primitiven Typen und Referenztypen ab. Zahlen werden in 4.1.4 behandelt, Zeichen und boolesche Werte in 4.1.2.

In Smalltalk ist jedes Sprachkonstrukt ein Objekt. Eine Applikation greift auf ein Objekt über seine Referenz zu. Diese Referenz ist ein Zeiger auf den Bereich im Hauptspeicher, in dem das Objekt lebt. Eine Anweisung kann diese Referenz ausschließlich dazu verwenden, dem Objekt Nachrichten zu schicken oder die Referenz einer Variablen zuzuweisen bzw. sie als Argument zu verwenden. Beim Kopieren einer Variablen wird lediglich die Referenz kopiert. Beide Variablen zeigen nach dem Kopieren auf das gleiche Objekt. Das Kopieren eines Objektes erfolgt explizit durch die Nachricht `copy`.

Java unterscheidet im Gegensatz zu Smalltalk zwischen **primitive Datentypen** und **Referenztypen**. Eine Variable eines primitiven Typs speichert den Wert direkt in der Variablen. Bei einem Referenztyp wird dagegen der Zeiger auf das Objekt oder das Array abgespeichert. Die primitiven Typen sind `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`. Sämtliche Referenztypen außer Arrays sind von der Klasse `Object` direkt oder indirekt abgeleitet. Um mit einen Element eines primitiven Typs als Objekt arbeiten zu können, muß es von einem Wrapperobjekt gekapselt werden. Das Package `java.lang` stellt für jeden primitiven Typ eine Wrapper–Klasse zur Verfügung.

Die Unterscheidung zwischen primitiven Typen und Referenztypen sorgt bei der Migration für folgende Probleme:

- Zuweisungen primitiver Elemente kopieren das Element selbst. Zuweisungen von Referenztypen kopieren lediglich die Referenz. Das kann bei der Migration zu Problemen führen (Siehe Zuweisungen in diesem Abschnitt).
- Höhere Datenstrukturen in Java wie `Vector` oder die Klassen des Collection–Toolkits können nur Referenztypen verarbeiten. Lediglich Arrays können mit primitiven Datentypen arbeiten; allerdings sind Arrays nicht so komfortabel zu verwenden wie Collections. Zur Verwendung in einer Collection müssen primitive Elemente in Wrapper gepackt werden bzw. vor der Verwendung als primitives Element aus dem Wrapper herausgeholt werden.

  Bei der Migration wird eine Smalltalk–Collection entweder zu einer Java–Collection oder zu einem Array. Diese Entscheidung ist von den Typen der gespeicherten Elemente abhängig.

Variablen

In diesem Abschnitt werden die verschiedenen Aspekte von Variablen wie Typisierung, Sichtbarkeit und Initialisierung behandelt. Außerdem wird in diesem Abschnitt auf indizierte Variablen in Smalltalk eingegangen.

Typisierung Smalltalk und Java unterscheiden sich darin, zu welchem Zeitpunkt der Typ einer Variablen festgelegt wird.

In Smalltalk resultiert der Typ einer Variablen aus dem Verhalten des Programms zur Laufzeit. Daher spricht man in Smalltalk von dynamischer Typisierung. Der Compiler kann den Typ einer Variablen oder eines Arguments zur Übersetzungszeit nicht feststellen. In einer Variablen kann somit ein Objekt eines beliebigen Typs gespeichert werden. Dadurch kann der Compiler die Gültigkeit einer Nachricht an ein Objekt nicht überprüfen.

In Java steht der Typ einer Variablen dagegen schon zur Übersetzungszeit fest. Die Typisierung geschieht explizit durch den Programmierer. Der Compiler kann dadurch die Gültigkeit der

Speicherung eines Wertes in einer Variablen und die Gültigkeit eines Methodenaufrufs bereits zu Übersetzungszeit überprüfen. Aufgrund der statischen Typisierung benötigen Java–Programme an manchen Stellen eine explizite Typumwandlung. Höhere Datenstrukturen greifen oft nur über die Schnittstelle einer Superklasse auf ein Objekt zu. Um über die Schnittstelle der tatsächlichen Klasse des Objekts zuzugreifen, muß die Applikation eine Typumwandlung auf die Unterklasse durchführen (*Downcast*). In Smalltalk ist eine solche explizite Typumwandlung aufgrund des dynamischen Typsystems nicht notwendig. Ein weiteres Konstrukt von Java ist der Interface–Mechanismus. Der Interface–Mechanismus ist mit Mehrfachvererbung zu vergleichen, wobei ein Interface jedoch ausschließlich abstrakte Methoden definiert. Interfaces werden ausführlicher in 4.1.1 besprochen.

Für die Migration bedeutet dies, daß die Typen der Variablen und Parameter ermittelt werden müssen, die sie zur Laufzeit haben. Im Java–Programm müssen diese Typen statisch angegeben werden. Bei dieser Laufzeitanalyse können die Namen der formalen Parameter der Methoden in Smalltalk sein. Oft wird im Parameternamen der Typ des erwarteten Arguments kodiert. Falls eine Klasse auf ein Objekt zugreifen muß, dessen Klasse sie bei ihrer eigenen Übersetzung nicht kennt, geschieht der Zugriff über ein Interface. Die Klasse des später erstellen Objekts muß dieses Interface implementieren, um die statische Typsicherheit herzustellen. Falls die Applikation auf Methoden eines Objekts zugreift, die nur in einer Subklasse definiert sind, muß durch die Migration ein expliziter Downcast eingefügt werden.

Sichtbarkeit In Smalltalk ist die Sichtbarkeit von der Art der Variablen abhängig. Die Sichtbarkeit von Variablen in Java ist nur in Teilen mit der von Smalltalk identisch. In diesem Abschnitt werden zunächst die verschiedenen Variablenarten diskutiert. Abbildung 4.3 veranschaulicht anschließend die Migration von Instanzvariablen, Klassenvariablen und Klasseninstanzvariablen.

Temporäre Variablen sind nur innerhalb einer Methode sichtbar. Sie müssen zu Beginn der Methode deklariert werden. In Java können temporäre Variablen nicht nur zu Beginn einer Methode, sondern an nahezu beliebiger Stelle deklariert werden. Bei der Migration müssen temporäre Variablen lediglich auf die Java–Syntax umgeschrieben werden.

Instanzvariablen sind in Smalltalk nur innerhalb der Instanz sichtbar. Ihre Lebendauer spannt sich von der Erzeugung bis zur Freigabe des Objekts. Instanzvariablen merken sich ihren Zustand über die Methodenaufrufe hinweg. Sie sind von außen nicht sichtbar, sondern nur innerhalb der Instanz selbst. Auf Instanzvariablen wird von außehalb der Instanz über *Getter–* und *Setter–Methoden* zugegriffen. In Java läßt sich die Sichtbarkeit von Instanzvariablen sehr flexibel über die Modifikatoren `public`, `protected` und `private` gestalten. Durch die Migration werden in Java vorwiegend private Instanzvariablen erzeugt. In einfachen Fällen kann in Java auf die Getter– und Setter–Methoden verzichtet werden. Die Sichtbarkeit der betroffenen Variablen muß dann durch einen Modifikator erweitert werden.

Klassenvariablen sind in Smalltalk innerhalb der deklarierenden Klasse selbst und in allen Unterklassen sichtbar. Auch auf Klassenvariablen kann von außerhalb der Instanz nur über Getter– und Setter–Methoden zugegriffen werden. In Java ist eine Klassenvariable, die nicht als `private` deklariert wird, in den Subklassen sichtbar. Auf sie kann auch direkt zugegriffen werden — ohne Verwendung von Getter– und Setter–Methoden. Durch die Migration wird auch in Java eine Klassenvariable erzeugt. In einfachen Fällen kann in Java auf die Getter– und Setter–Methoden verzichtet werden. Die Sichtbarkeit der betroffenen Variablen muß dann durch einen Modifikator erweitert werden.

Klasseninstanzvariablen werden in Smalltalk ebenso wie Klassenvariablen im Klassenobjekt deklariert. Allerdings existiert für jede Unterklasse und die Superklasse je eine eigene Kopie einer Klasseninstanzvariablen. Klasseninstanzvariablen existieren in Java nicht. Sämtliche Klassen einer Subklassenhierarchie teilen sich eine einzige Klassenvariable. Falls jede Klasse eine eigene

Kopie der Klassenvariablen besitzen soll, muß sie in jeder Klasse einzeln deklariert werden. Da dies in Subklassen aber nicht durch eine Variablendeklaration erzwungen werden kann, benötigt die Java–Lösung abstrakte Getter– und Setter–Methoden. Diese Methoden werden in der Superklasse definiert und müssen in jeder Subklasse implementiert werden. Sie arbeiten auf einer jeweils lokal deklarierten Instanzvariablen. Durch die Migration müssen in der Superklasse die abstrakten Getter– und Setter–Methoden definiert und in den Subklassen implementiert werden. Die Deklaration der Klasseninstanzvariablen fällt in Java weg. Dagegen muß in jeder Subklasse eine lokale Klassenvariable deklariert werden.

Globale Variablen und **Poolvariablen** sind in Smalltalk global zugänglich. Smalltalk speichert globale Variablen im Verzeichnis `Smalltalk`, auf das jedes Objekt Zugriff hat. Poolvariablen existieren in Variablenpools (z.B. `CwConstants`). Falls ein Objekt auf solche Variablen zugreifen will, muß die Klasse diesen Variablenpool in der Klassendefinition angeben. Für das Verzeichnis globaler Variablen und sonstiger Variablenpools existiert in Java kein Gegenstück. Eine Variable ist immer einer Klasse oder einer Instanz zugeordnet. Für die Migration bedeutet das, daß Variablen, die einer beliebigen Menge an Objekten zugänglich sein sollen, in einer Klasse zusammengefaßt werden müssen. Für jeden Variablenpool und das globale Verzeichnis erzeugt die Migration also je eine eigene Klasse, wobei die Variablen eine `public`– Markierung tragen. Die verwendenden Objekte importieren dann diese Klassen.

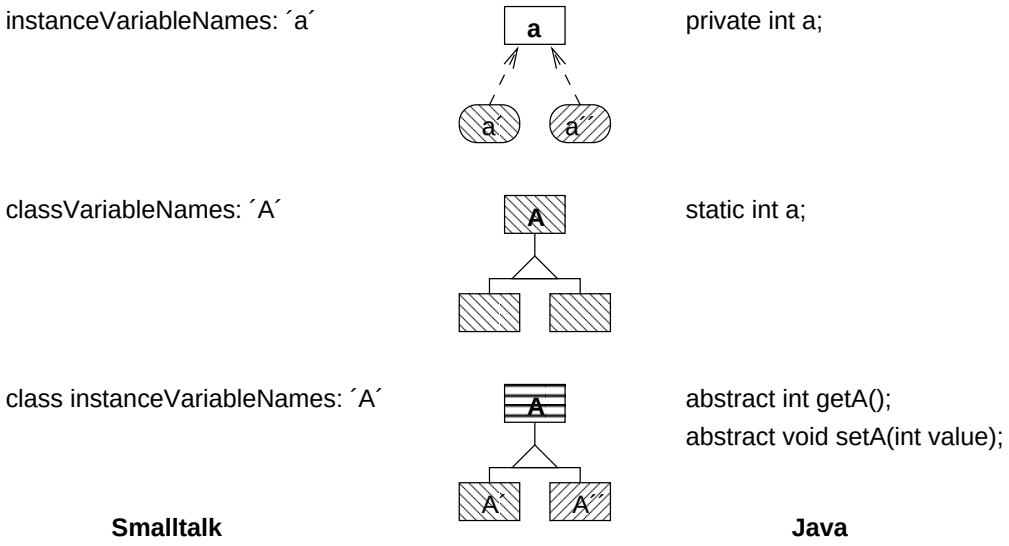


Abbildung 4.3: Migration von Variablen verschiedener Sichtbarkeit

Abbildung 4.3 stellt die verschiedenen Sprachkonstrukte zur Deklaration der verschiedenen Variablenarten in Smalltalk und ihre Entsprechungen in Java dar. Zusätzlich wird die Sichtbarkeit der verschiedenen Variablen durch unterschiedliche Schraffuren veranschaulicht.

Initialisierung In Smalltalk wird die Initialisierung von Variablen nicht durch ein spezielles Sprachkonstrukt unterstützt. Daher verwendet ein Smalltalk–Programm zur Initialisierung von Variablen oft ein Idiom. Die Schwierigkeit liegt im Erkennen des Initialsierungscodes.

Java unterstützt die Variableninitialisierung durch mehrere Mechanismen. Jede Variable kann bei ihrer Deklaration mit einem Initialwert versehen werden. Neben dem Konstruktor kann zur

komplexeren Initialisierung von Instanzvariablen ein *instance initializer* verwendet werden. Klassenvariablen können über einen *static initializer* initialisiert werden.

Die Migration von Variableninitalisierungen wird bei den Idiomen *Explicit Initialization* (5.1.11) und *Lazy Initialization* (5.1.12) besprochen.

Indizierung Neben den gewöhnlichen Instanzvariablen, auf die ein Objekt über den Namen zugreift, definiert Smalltalk indizierte Variablen. Das Anlegen einer Klasse entscheidet darüber, ob eine Klasse eine *fixed class* oder eine *variable class* ist. Jede Instanzvariable einer fixed class besitzt einen Namen, der ihrer Identifizierung dient. Die Instanzvariablen einer variable class werden ausschließlich über ihren Index angesprochen. Eine indizierte Variable speichert entweder eine Objektreferenz oder einen Bytewert. Die Zahl der Instanzvariablen eines Objekts kann sich zur Laufzeit ändern. Dieser Mechanismus dient der Implementierung der Klassen des Collection–Toolkit, die diesen Zugriff auf indizierte Variablen vor dem Benutzer kapseln.

In Java kann jede Klasse neben einer Reihe namensreferenzierter Variablen eine Reihe an indizierten Variablen enthalten. Indizierte Variablen werden durch Arrays realisiert. Ein Array speichert eine Menge an primitiven Elementen oder Objektreferenzen. Jedes Array verfügt über einen eigenen Namen.

Die Migration einer variable class besteht folglich im Anlegen einer Array–Instanzvariablen in der Java–Klasse. Ein einmal erzeugtes Array kann in Java seine Größe nicht mehr ändern. Folglich muß das Array bei Größenänderungen in ein größeres Array kopiert werden, das dann seinen Platz einnimmt.

Nachrichten

Eine Nachricht (oder Methodenaufruf) dient der Repräsentation von Interaktionen zwischen Objekten [GR83, S.24]. Zunächst besprechen wir, wie einer Nachricht Argumente mitgegeben werden. Anschließend wird die Verkettung und Kaskadierung von Nachrichten besprochen. Abschließend werden die polymorphen Eigenschaften von Nachrichten und dynamische Methodenaufrufe diskutiert.

Parameter Die Parameter eines Methodenaufrufs sind in Smalltalk mit Positionsnamen versehen, deren Position in der Parameterliste feststeht. Parameter sind in Smalltalk dynamisch getypt.

Methodenparameter sind in Java dagegen reine Positionsparameter. Parameter sind statisch getypt.

Durch die Migration werden die Positionsnamen der Parameter weggelassen. Der Versender einer Nachricht muß beachten, daß die Argumente einer Nachricht den korrekten statischen Typ haben müssen. Der Compiler überprüft in Java die korrekte Typisierung der Argumente (Siehe auch Typisierung in diesem Abschnitt).

Verkettung Verkettung ist das Verschicken einer Nachricht an das Ergebnis eines vorherigen Methodenaufrufs. In Smalltalk können aufgrund der Namensparameter viele Klammern notwendig werden.

In Java ist jeder Methodenaufruf geklammert. Zusätzlich verlangt Java in manchen Fällen eine explizite Typumwandlung (Downcast) bei der Verkettung von Nachrichten, was in diesem Fall zu weiteren Klammern führt.

Die Migration von Verkettung bedeutet ein Umformatieren des Quelltextes.

Kaskadierung Ein Kaskade ist das Verschicken mehrerer Nachrichten an ein einziges Objekt. Solche Methodenaufrufe werden in Smalltalk durch ein Semicolon voneinander getrennt. Falls am Ende einer Kaskade das Empfängerobjekt zurückgeliefert werden muß, kann die Nachricht `yourself` notwendig werden.

Kaskadierte Nachrichten können in Java nicht dargestellt werden.

Kaskaden müssen in Java mit Hilfe einer temporären Variablen und voneinander getrennten Nachrichten nachgebildet werden (siehe 5.1.16). Durch den Wegfall von kaskadierten Nachrichten wird die Nachricht `yourself` überflüssig (siehe 5.1.17).

Polymorphie Polymorphie ist die Eigenschaft, daß ein Name Objekte vieler verschiedener Klassen bezeichnen kann [Boo94].

In Smalltalk sind zwei Arten polymorpher Methodenaufrufe möglich. Bei der ersten Art ist die Klasse des Empfängers eine Subklasse derjenigen Klasse, die das verwendete Protokoll definiert. Der Klient weiß nicht, zu welcher konkreten Subklasse der Emfänger gehört. Bei der zweiten Art stehen die möglichen Empfängerobjekte in keinem direkten Zusammenhang zueinander. Sie zeichnen sich dadurch aus, daß alle Empfänger eine Methode gleichen Namens definieren. Wegen des dynamischen Typsystems ist auch in diesem Fall ein polymorpher Methodenaufruf möglich.

Java ist hier restriktiver als Smalltalk, weil der Übersetzer schon zur Übersetzungszeit den Typ des Empfängers überprüft. Ein in einer Variablen gespeichertes Objekt muß entweder eine Instanz der in der Variablendeklaration angegebenen Klasse oder einer ihrer Subklassen sein oder muß das Interface implementieren, welches als Typ der Variablen deklariert ist.

Ein polymorpher Methodenaufruf der ersten Art ist direkt nach Java zu übertragen, indem in Java die gleich Klassenstruktur aufgebaut wird. Bei einem polymorphen Methodenaufruf der zweiten Art muß in Java ein Interface erstellt werden, um dem Compiler die statische Typüberprüfung zu ermöglichen. Der Methodenaufruf erfolgt über das im Interface definierte Protokoll.

Dynamischer Methodenaufruf Ein Methodenaufruf ist dann dynamisch, wenn zur Übersetzungszeit der Selektor der aufgerufenen Methode noch nicht bekannt ist.

In Smalltalk schickt der Klient dem Empfänger die Nachricht `perform:` und übergibt den Selektor der gewünschten Methode. Aufgrund des dynamischen Typsystem erfolgt die Überprüfung der Gültigkeit des Aufrufs erst beim Empfang der Nachricht. Falls ein ungültiger Selektor übergeben wurde, ruft der Empfänger die Fehlermethode `doesNotUnderstand` auf (siehe Fehlermethoden in 4.1.2).

In Java ist das Ausführen eines dynamischen Methodenaufrufs etwas komplizierter als in Smalltalk. Java verlangt hierzu zunächst das Erzeugen eines `Method`–Objekts, welches den dynamischen Methodenaufruf kapselt. Der Klient muß dieses `Method`–Objekt über das Klassenobjekt des Empfängers erfragen. Falls Parameter übergeben werden sollen, ist viel Schreibarbeit nötig.

Die Migration hat mit dem statischen Typsystem in Java zu kämpfen. Die Applikation muß das Methodenobjekt vor dem dynamischen Aufruf abfragen. Dies kann entweder direkt davor geschehen oder zu einem früheren Zeitpunkt. Um die Ausnahmebehandlung möglichst spezifisch gestalten zu können, sollte das Methodenobjekt so früh wie möglich abgefragt werden. Diese Abfrage kann an ganz anderer Stelle nötig sein als im Versender des dynamischen Methodenaufrufs. In vielen Fällen kann ein dynamischer Methodenaufruf in Java durch einen statischen Methodenaufruf ersetzt werden. Dazu definiert der Sender ein Interface, welches der Empfänger implementieren muß. Der Sender kommuniziert über die im Interface definierten Methoden mit dem Empfänger.

Operatoren

Operatoren sind in Smalltalk Nachrichten, die auf den Doppelpunkt am Ende des Parameternamens verzichten (siehe auch Nachrichten in diesem Abschnitt). Daher gelten für Operatoren genau die

gleichen Assoziationsregeln wie für gewöhnliche Nachrichten. Beispielsweise liest Smalltalk den Ausdruck `index + offset * 2` als `(index + offset) * 2`. Eine Anwendung kann in Smalltalk eigene Operatoren definieren. Smalltalk unterscheidet unäre und binäre Operatoren.

In Java verwenden Operatoren die Infix–Notation, die der gewöhnlichen mathematischen Schreibweise entspricht. Die Definition eigener Operatoren und das Überschreiben vorhandener Operatoren ist in Java nicht möglich. In Java sind sämtliche Operatoren in der Sprache und bis auf Gleichheit und `instanceof` nur für primitive Typen definiert. Java unterscheidet ebenfalls unäre und binäre Operatoren.

Operatoren können in zweierlei Hinsicht ein Problem für die Migration darstellen. Die unterschiedlichen Assoziativitätsregeln können bei einer Migration zu einer Semantikveränderung von Ausdrücken führen. Diese Semantikveränderung kann durch ein Umformulieren oder Klammern beseitigt werden. Da Operatoren in Java vorwiegend für primitive Typen definiert sind, kann die Migration eine Umformulierung der Operation in einen Methodenaufruf bedeuten. Selbstdefinierte Operatoren müssen als Methoden definiert werden. Falls die Operanden in Wrapper–Klassen gekapselt sind (siehe 4.1.1), müssen sie zunächst ausgepackt werden.

Methoden

Dieser Abschnitt untersucht die Typisierung der Methodenparameter und unterscheidet zwischen Instanzmethoden und Klassenmethoden.

Typisierung der Parameter Eine Smalltalk–Methode muß keine Typen für ihre Parameter definieren. Daher kann sie Parameter eines beliebigen Typs akzeptieren. Die Methodenauswahl ist nicht abhängig vom Typ des erhaltenen Arguments. Falls diese Auswahl nötig ist, um die richtige Operation ausführen zu können, wird in Smalltalk das Idiom *Double Dispatch* (5.1.9) verwendet.

Eine Java–Methode dagegen muß für jedes Argument einen Typ deklarieren. Durch das statische Typsystem ist die Methodenauswahl innerhalb eines Objekts von den Typen der Parameter abhängig. Durch Überschreiben (overriding) einer Methode erreicht ein Objekt Methodenpolymorphie innerhalb einer Klasse.

Bei der Migration muß die Deklaration einer Methode um die Typen der Parameter erweitert werden. Nur eine Laufzeitanalyse kann die Typen der Parameter ermitteln. Die Positionsnamen können für die Analyse hilfreich sein, weil sie in Smalltalk oft den Typ des Parameters kodieren. (Siehe auch Datentypen in diesem Abschnitt) Weil die Methodenauswahl in Java von den Parametertypen abhängig gemacht werden kann, entfällt in Java die Notwendigkeit, das Idiom Double Dispatch zu implementieren. Die Parameterabhängigkeit wird durch Überschreiben gelöst. Das Entfernen dieses Idioms führt auch zum Entfernen von den Methoden, die in anderen Klassen durch dieses Idiom eingeführt wurden.

Instanzmethoden Eine Instanzmethode arbeitet im Kontext ihrer Instanz und hat Zugriff auf alle sichtbaren Variablen.

In Smalltalk sind sämtliche Methoden global sichtbar. Sie definieren die Schnittstelle eines Objektes. Jede Methode einer Klasse kann mit einem Protokollkommentar versehen werden. Allerdings haben diese Protokollkommentare keinen Einfluß auf die Sichtbarkeit einer Methode (Siehe auch Protokolle in diesem Abschnitt).

Java kennt eine ausgefeiltere Kontrolle der Sichtbarkeit. Durch die Modifikatoren `public`, `protected` und `private` kann die Sichtbarkeit einer Methode sehr fein eingestellt werden. Protokollkommentare existieren in Java nicht.

Instanzmethoden stellen für die Migration keine große Herausforderung dar. Da in Smalltalk alle Methoden global sichtbar sind, können sie in Java mit `public` markiert werden. Falls eine

Methode den Protokollkommentar `private` trägt, wird sie in Java mit `private` markiert. Dadurch verhindert der Compiler einen Aufruf dieser Methode von außen.

Klassenmethoden Eine Klassenmethode arbeitet im Kontext ihrer Klasse und hat Zugriff auf alle Klassenvariablen und Klasseninstanzvariablen.

Klassenmethoden sind in Smalltalk ebenfalls global sichtbar. In Smalltalk muß bei Aufruf einer Klassenmethode erst das Klassenobjekt erfragt werden. Eine Instanz kann den Aufruf einer Klassenmethode nicht direkt verstehen. Die Klassenmethoden definieren die Schnittstelle des Klassenobjekts.

In Java kann eine Klassenmethode entweder über die Klasse oder über eine Instanz dieser Klasse aufgerufen werden. Der Aufruf über die Klasse erfordert nicht das Vorhandensein einer Instanz. Das Klassenobjekt kann hierzu nicht verwendet werden — es wird vorwiegend für Reflexion verwendet (siehe 4.1.7). Die Sichtbarkeit von Klassenmethoden kann ebenfalls durch Modifikatoren sehr fein eingestellt werden.

Durch die Migration wird eine Klassenmethode in Java mit `public` markiert. Falls sie den Protokollkommentar `private` trägt, wird sie in Java mit `private` markiert. Dadurch verhindert der Compiler einen Aufruf dieser Methode von außen. Durch die andere Art des Aufrufs einer Klassenmethode müssen sämtlich Klienten einer Klassenmethode verändert werden. Entweder erfolgt der Aufruf über die vorhandene Instanz oder direkt über die Klasse. Falls durch den Aufruf eine neue Instanz angelegt werden soll, muß statt der Klassenmethode in Java der passende Konstruktor aufgerufen werden (siehe Objekt–Erzeugung in 4.1.2).

Ausdrücke

Ein Ausdruck ist in Smalltalk entweder eine Zuweisung, ein Methodenaufruf oder eine Return–Anweisung.

Zuweisungen Eine Zuweisung kopiert in Smalltalk lediglich die Referenz auf ein Objekt. Das Objekt, auf das die Referenz zeigt, bleibt unangetastet.

In Java geschieht dies nur bei Referenztypen. Zuweisungen primitiver Typen kopieren den Wert selbst. Siehe ebenso die Diskussion zum Kopieren von Objekten 4.3.1.

Obwohl Java zwischen primitiven und Referenztypen unterscheidet, spielt das für die Migration einer Zuweisung keine Rolle. Die Klassen in Smalltalk, die den primitiven Typen in Java entsprechen, lassen ein Verändern des einmal erzeugten Objekts nicht zu. Daher kann aus der Aliasbildung, die durch die Zuweisung entsteht, kein Problem erwachsen.

Methodenaufrufe Methodenaufrufen ist ein eigener Abschnitt gewidmet (siehe Nachrichten in diesem Abschnitt).

Return–Anweisungen Wenn eine Smalltalk–Methode keine explizite Return–Anweisung enthält, gibt sie ihre Instanz zurück. Ob dieses Objekt benötigt wird, kann nicht durch lesen der Klasse selbst entschieden werden, sondern nur durch Studium der Verwendungen der Instanzen dieser Klasse.

Java verlangt dagegen eine explizite Return–Anweisung und eine Spezifikation des Typs des Rückgabewerts im Methodenkopf. Der Compiler überprüft das Vorhandensein einer Returnanweisung, falls ein Rückgabetyp im Methodenkopf definiert ist.

Eine Schwierigkeit bei der Migration sind die Methoden, die keine Return–Anweisungen enthalten, deren Klienten sich aber auf die Rückgabe der Instanz verlassen. Da eine Smalltalk–Methode,

die keine Return–Anweisung enthält, den Empfänger zurückgibt, kann die Notwendigkeit zum Einfügen einer Return–Anweisung in Java nicht immer leicht erkannt werden.

Programmdefinition

Ein Programm besteht aus den Klassen und Objekten, die während der Entwicklung erzeugt wurden. Es stellt sich die Frage, wie eine solches Programm gegen den Rest des Systems abgegrenzt wird.

In Smalltalk leben diese Objekte im *Image* des Systems. Dieses Image ist die Datei des Smalltalk–Systems, das sämtliche Instanzen enthält. Da in Smalltalk Klassen lediglich Instanzen sind, leben auch die Klassen in diesem Image. Das Programm und das Smalltalk–System sind eng miteinander verbunden. Die erzeugten Klassen sind Bestandteil des Gesamtsystems und liegen nicht als separate Dateien vor. Insbesondere kann ein Smalltalk–Programmierer das bestehende System für seine persönlichen Belange modifizieren. Dadurch wird die Abgrenzung eines Programmes in Smalltalk schwierig. IBM Smalltalk verwaltet aus diesem Grund Applikationen, die zusammengehörende Klassen verwalten. Damit ist bei der Migration klar, welche Klassen zusammengehören. Um eine Smalltalk–Applikation auszuliefern, wird eine gestrippte Version des Images erzeugt.

In Java ist die Abgrenzung eines Programms eindeutig. Die erzeugten Klassen liegen in Form von Dateien vor, die getrennt vom System verwaltet werden. Eine Java–Applikation hat die Möglichkeit, mehrere Klassen zu einem Package zusammenzufassen. Das System selbst wird durch die Erstellung der Applikation nicht verändert. Zur Laufzeit der Applikation benötigte Objekte müssen von der Applikation explizit angelegt werden.

Bei der Migration einer Applikation muß zunächst festgestellt werden, welche Klassen und Objekte zu der Applikation gehören. Falls Smalltalk diese Elemente als Applikation verwaltet, sind die Klassen leicht festzustellen. Schwieriger ist die Identifizierung der Instanzen, die vorhanden sein müssen. Eine Java–Applikation muß diese Instanzen explizit anlegen.

Namensraum

Ein Namesraum definiert die Sichtbarkeit von Namen. Die Gestaltung des Namensraums hat Einfluß darauf, ob Nameskonflikte auftreten können und wie diese behoben werden können.

Smalltalk hat für Klassen einen einzigen globalen Namensraum. Jede definierte Klasse ist überall sichtbar. Um Namenskonflikte zu vermeiden, werden Klassen oft mit einem Präfix versehen (beispielsweise `Cw...` für die Klassen des Common Widget Subsystem). Der Variablenpool `Smalltalk` enthält Variablen, die global sichtbar sind. Daneben existieren Variablenpools, deren Variablen nicht ohne weiteres sichtbar sind. Um die Variablen eines Variablenpools verwenden zu können, muß eine Klasse diesen Pool bei ihrer Deklaration angeben.

Jede Klasse in Java gehört eindeutig zu einem Package. Eine Klasse gibt die Packagezugehörigkeit bei der Klassendeklaration an. Um eine Klasse verwenden zu können, muß ihr vollständig qualifizierter Name angegeben werden. Eine unqualifizierte Verwendung ist nur möglich, wenn die Klasse vor ihrer Verwendung importiert wird. Durch diesen Mechanismus werden Konflikte im Namensraum effektiv verhindert. Namespräfixe wie in Smalltalk sind damit überflüssig. Da global sichtbare Variablen und Konstanten nur in Klassen deklariert werden, lassen sich Namenskonflikte hierbei ebenfalls auflösen.

Bei der Migration muß entschieden werden, welchem Package eine Klasse angehören soll. Am einfachsten wird für die migrierte Applikation ein eigenes Package definiert. Dadurch sind alle Klassen der Applikation im gesamten Package sichtbar. Die in IBM Smalltalk vorhandenen Subapplikationen lassen sich in Java als Subpackages darstellen. Da in Smalltalk der Programmierer durch geeignete Maßnahmen Namenskonflikte verhindern muß, entstehen durch die Migration keine

neuen Namenskonflikte. Allerdings müssen die entsprechenden Packages in Java explizit importiert werden. Idealerweise sollten die nach der Migration überflüssig gewordenen Präfixe entfernt werden.

Klassenbeziehungen

Dieser Abschnitt behandelt die statischen Beziehungen der Klassen untereinander.

Klassenhierarchie Die statischen Beziehungen zwischen den Klassen sorgen für die Strukturierung der Klassen in einer Hierarchie. In Smalltalk wie Java ist die Klasse `Object` die Wurzel dieser Hierarchie. Jede Klasse im System muß von genau einer Superklasse abgeleitet sein; Mehrfachvererbung existiert weder in Smalltalk noch in Java. In Smalltalk kann eine Klasse beliebig viele Subklassen besitzen. Eine Java–Klasse kann mit `final` markiert werden, was ein Erben von dieser Klasse verhindert. Smalltalk definiert zusätzlich eine Hierarchie von Metaklassen (4.1.7).

Eine statische Beziehunge von Klassen definiert gleichzeitig eine Vererbungsbeziehung. In Smalltalk erbt eine Subklasse sämtliche Instanzvariablen und Methoden. Eine in der Superklasse definierte Klassenvariable wird von sämtlichen Subklassen geteilt. Von einer Klasseninstanzvariablen erbt jede Subklasse eine eigene Kopie. In Java wird die Vererbung durch Modifikatoren beeinflußt; prinzipiell kann aber jede Variable an die Subklasse vererbt werden (siehe auch Variablen in diesem Abschnitt). Die Verbung von Methoden wird in Java ebenso von Modifikatioren beeinflußt. Prinzipiell kann aber jede Instanz– oder Klassen–Methode an Subklassen vererbt werden. Im Gegensatz dazu werden Konstruktoren in Java nicht vererbt. In Smalltalk sind die Konstruktormethoden gewöhnliche Klassenmethoden und werden daher vererbt (5.1.1). Variablen und Methoden können in Java durch die Markierung `final` vor einem überschreiben in Subklassen geschützt werden.

Abstrakte Klassen Eine abstrakte Superklasse tritt nicht nur als Anbieter von Elementen auf, sondern kann den Subklassen auch die Pflicht auferlegen, bestimmte Methoden anzubieten. In Smalltalk besteht eine solche Methode in der Superklasse aus einem Aufruf der Methode `subclassResponsibility`. Diese Methode gehört zu den Fehlermethoden der Klasse `Object`. Aufgrund des dynamischen Typsystems kann das Einhalten dieser Bedingung erst zur Laufzeit festgestellt werden. Eine Java–Klasse markiert solche Methoden mit `abstract`. Dadurch kann der Compiler statisch sicherstellen, daß alle Subklassen diese Methode implementieren.

Klassenerweiterung Neben der Erweiterung einer bestehenden Klasse durch Vererbung definiert Smalltalk den Mechanismus der Klassenerweiterung (base class extension). Damit kann eine bestehende Klasse um neue Methoden erweitert werden. Für einen Klienten der Klasse spielt es keine Rolle, ob die Methode in der Klasse selbst, oder in der Klassenerweiterung definiert ist. Java kennt keinen vergleichbaren Mechanismus.

Interfaces und innere Klassen In Java existieren zusätzlich zu den schon behandelten Klassen sogenannte Interfaces und innere Klassen. Ein Interface ist eine Art abstrakter Klasse. Falls eine gewöhnliche Klasse ein Interface in der Klassendefinition angibt, muß sie sämtliche Methoden implementieren, die in dem Interface definiert sind. Ein Interface enthält selbst keinen Code, sondern nur Methodendefinitionen. Damit ist eine Art Mehrfachvererbung in Java möglich. Allerdings wird dadurch kein Verhalten vererbt, sondern nur Design. Interfaces werden an vielen Stellen aufgrund des statischen Typsystems von Java nötig.

Eine innere Klasse ist eine Klasse, die im Sichtbarkeitsbereich einer umgebenden Klasse definiert wird. Dieser Mechanismus ist mit der Prozedurschachtelung von Pascal vergleichbar. Eine innere Klasse kann auf den internen Zustand von Instanzen der definierenden Klasse zugreifen. Falls die

innere Klasse direkt nach ihrer Definition instanziiert wird, muß sie nicht einmal einen eigenen Namen erhalten (anonymous class).

Für die Migration bedeuten die Unterschiede in den Beziehungen der Klassen untereinander folgendes: Da beide Sprachen nur Einfachvererbung definieren, sind hierdurch nur geringe Umstrukturierungen der Klassenhierarchie zu erwarten. Die Vererbung von Variablen und Methoden läßt sich leicht von Smalltalk nach Java übertragen. Konstruktoren bilden hier die einzige Ausnahme. Diese müssen in jeder Klasse explizit implementiert sein (abgesehen vom Default–Konstruktor). Klassenerweiterungen entfallen in Java dagegen komplett. D.h. die Methoden, die auf die Klassenerweiterung angewiesen sind, müssen so umgeschrieben werden, daß sie die Funktionalität der Klassenerweiterung auf andere Art nutzen. Das kann den Programmcode komplizierter und weniger elegant machen.

Der Interface–Mechanismus wird bei der Migration mancher Konstrukte aufgrund des statischen Typsystems von Java benötigt: die Ereignisbehandlung definiert Listener–Interfaces (4.3.3), polymorphe Methodenaufrufe benötigen je nach Kontext ein Interface (siehe Nachrichten in diesem Abschnitt), dynamische Methodenaufrufe können manchmal mit Hilfe eines Interfaces durch statische Methodenaufrufe ersetzt werden (siehe Nachrichten in diesem Abschnitt), ein Protokoll kann durch ein Interface explizit gemacht werden (siehe Protokolle in diesem Abschnitt).

Literale

Smalltalk kennt die Literalklassen Zeichen, Strings, Zahlen, Symbole, Selektoren und Felder. Dieser Abschnitt behandelt die Literalklassen ausschließlich in Bezug auf Darstellung im Programmtext. Jeder Paragraph enthält einen Verweis auf die ausführliche Darstellung der Funktionalität.

Zeichen Zeichen unterscheiden sich in Smalltalk und Java hinsichtlich ihrer Darstellung. Tabelle 4.1 stellt die Darstellungsarten von Zeichen einander gegenüber. Zudem kann ein Java–Programm ein Zeichen in Unicode–Kodierung enthalten. Siehe auch 4.1.2.

Smalltalk	Java
`$m`	`'m' '\\' '\u0138'`

Tabelle 4.1: Darstellung von Zeichen

Strings Strings unterscheiden sich in Smalltalk und Java hinsichtlich ihrer Darstellung. Tabelle 4.2 stellt die Stringdarstellungen einander gegenüber. Siehe auch den Abschnitt über Strings in 4.2.1.

Smalltalk	Java
`'foobar'`	`"foobar"`

Tabelle 4.2: Darstellung von Strings

Zahlen Die Literale für Zahlen unterscheiden sich hinsichtlich ihrer Darstellung im Programmtext in Smalltalk und Java. Tabelle 4.3 stellt die Darstellungsarten von Zahlen einander gegenüber. Siehe auch 4.1.4.

	Smalltalk	Java
Integer	`3`	`3 3L`
Hexadecimal	`16rFF`	`0xFF`
Oktal	`8r153`	`0153`
Fließkomma	`30.45 1.58e5`	`30.45 1.58e5`

Tabelle 4.3: Darstellung von Zahlen

Symbole Smalltalk definiert zusätzlich zu Strings Symbole. Symbole sind insofern einzigartig, als nie zwei verschiedene Symbole die gleiche literale Darstellung haben. Symbole werden unter anderem dazu verwendet, bei einem dynamischen Methodenaufruf die gewünschte Methode zu identifizieren. Siehe auch 4.2.1. Symbole existieren in Java nicht und werden daher oft durch Strings ersetzt. Allerdings muß dadurch eine Applikation die Einzigartigkeit eines Strings selbst garantieren. Die Verwendung von Symbolen in Smalltalk zur Identifizierung von Methoden wird in Java durch ein Methodenobjekt ersetzt (`java.lang.reflection.Method`).

Selektoren Ein Selektor dient der Auswahl der Methode des Empfängerobjekts eines statischen Methodenaufrufs. Seine Darstellung ist ausschließlich dadurch gekennzeichnet, daß er hinter einem Ausdruck steht, dessen Wert ein Objekt ist. Ein Selektor ist in Java durch einen vorangestellten Punkt gekennzeichnet.

Falls ein Selektor als Argument verwendet wird oder in einer Variablen abgespeichert wird, ist seine Darstellung die eines Symbols. Da dies in Java nicht möglich ist, fällt diese Darstellungsart weg. Ein Selektor in Form eines Symbols wird meist zum Ausführen eines dynamischen Methodenaufrufs verwendet.

Felder Die literale Darstellung von Arrays unterscheidet sich in Smalltalk und Java in zweierlei Hinsicht. Zum einen müssen die Elemente eines literalen Arrays in Java alle den gleichen Typ haben. Zum anderen unterscheidet sich die Syntax (siehe Tabelle 4.4). Bei der Migration müssen also Arrays, deren Elemente gemischte Typen aufweisen, entweder auf mehrere Arrays in Java aufteilt werden, oder sämtliche Elemente müssen als Objekte verwendet werden. Diese Verwendung kann das Einpacken eines primitiven Elements in seine Wrapper–Klasse bedeuten. Einschachtelungen von Arrays sind in beiden Sprachen möglich. Siehe auch 4.2.1.

Smalltalk	Java
`#(3 'nine' $c #(3 5))`	`{1,2,{3,4}}`

Tabelle 4.4: Darstellung von Arrays

Kommentare

Smalltalk schließt Kommentare durch doppelte Anführungszeichen ein. Diese werden in Java problemlos mit `/* */` markiert.

Protokolle

In diesem Abschnitt werden Protokolle und Protokollkommentare besprochen.

Protokollkommentare dienen der Strukturierung der Methoden einer Klasse, um dem Anwender das Verstehen der Klasse zu erleichtern. Diese Aufteilung der Methoden hat allerdings keinen Einfluß auf deren Verhalten, sondern hat lediglich Kommentarcharakter. Ein direktes Gegenstück existiert in Java nicht. Im Java–Programm erscheint diese Markierung als gewöhnlicher Kommentar.

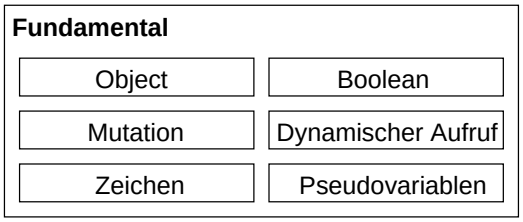
Neben diesen Protokollkommentaren existieren informelle Protokolle. Diese bezeichnen die Gesamtheit der Methoden, über die zwei Objekte miteinander kommunizieren. Nach Gamma „werden Smalltalk–Protokolle in Java durch Interfaces dargestellt“ [Gam97]. Für eine ausführliche Diskussion von Protokollen siehe Mediating Protocols (5.1.10).

Kategorien

Kategorien dienen der Strukturierung der Klassen einer Klassenbibliothek, um dem Anwender das Verstehen der Bibliothek zu erleichtern. Diese Aufteilung der Klassen in Kategorien hat allerdings keinen Einfluß auf deren Verhalten, sondern lediglich Kommentarcharakter. Java stellt Kategorien üblicherweise durch Packages dar. Diese fassen Klassen zu einer Einheit zusammen, was allerdings Einfluß auf die Sichtbarkeit von Variablen und Methoden hat.

Zusätzlich separieren Packages den Namensraum und helfen somit bei der Vermeidung von Namenskonflikten (siehe Namensraum in diesem Abschnitt). Das verhindert zum einen Konflikte von Klassennamen im globalen Namensraum. Zum anderen werden dadurch Namenspräfixe überflüssig. Die hierarchische Strukturierung in Applikationen und Subapplikationen in IBM Smalltalk bildet Java durch die ebenfalls hierarchische Aufteilung in Packages und Subpackages nach.

4.1.2 Fundamental–Protokollklasse



Die Klassen der Fundamental–Protokollklasse sind Teil der Sprachdefinition selbst. Wir behandeln der Reihe nach die in der Abbildung dargestellten Teilaspekte dieser Protokollklasse.

Die Klasse `Object`

Die Klasse `Object` hat in Smalltalk wie Java die gleiche Funktion — sie ist die Superklasse aller anderer Klassen und definiert daher das Protokoll, das alle Objekte verstehen. In dieser Arbeit teilen wir die Funktionalität von `Object` in einen Sprachteil und einen Frameworkteil auf. Die Frameworkfunktionalität wird in 4.3.1 besprochen. Dieser Abschnitt behandelt die Erzeugung, Initialisierung, Freigabe, den Test der Funktionalität und die Fehlermethoden eines Objekts.

Erzeugung Die Erzeugung eines neuen Objekts geschieht in Smalltalk nicht durch Aufruf eines Konstruktors, sondern durch Aufruf einer Klassenmethode. Der Mechanismus zum Erzeugen eines neuen Objekts wird durch das Meta–Objekt–Protokoll zur Verfügung gestellt (siehe Abschnitt Reflexion 4.1.7). Abbildung 4.4 zeigt das Erzeugen eines Objekts in Smalltalk anhand eines Beispiels. Die Klasse `Foo` kann die Methode `new` überschreiben, um das neu angelegte Objekt zu konfigurieren.

Die Implementierung von `new` in der Klasse `Behaviour` legt das Objekt an. Die Klasse `Behaviour` ist Teil des Meta–Objekt–Protokolls. Die gezeigten Klassen sind durch eine Vererbungsbeziehung der jeweiligen Metaklassen miteinander verbunden.

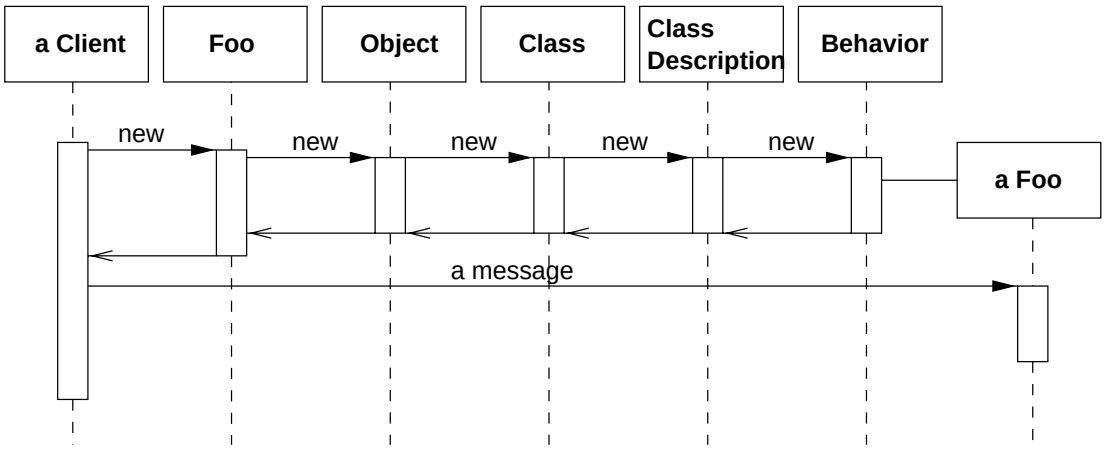


Abbildung 4.4: Erzeugen eines Objekts in Smalltalk

Eine Java–Applikation erzeugt ein neues Objekt in den allermeisten Fällen durch das Schlüsselwort `new` im Programmtext. Abbildung 4.5 zeigt das Erzeugen eines Objekts in Java anhand eines Beispiels. Der Konstruktor der Klasse wird implizit nach der Erzeugung des Objekts aufgerufen. Durch die Definition mehrerer Konstruktoren kann das erzeugt Objekt vom Klienten durch einen in die Sprache integrierten Mechanismus konfiguriert werden

Neben der Möglichkeit, ein Objekt durch das Schlüsselwort `new` anzulegen, kann auch das Klassenobjekt verwendet werden. Dadurch ist eine Auswahl der gewünschten Klasse zur Laufzeit möglich. Diese Art der Instanzerzeugung ist allerdings die Ausnahme.

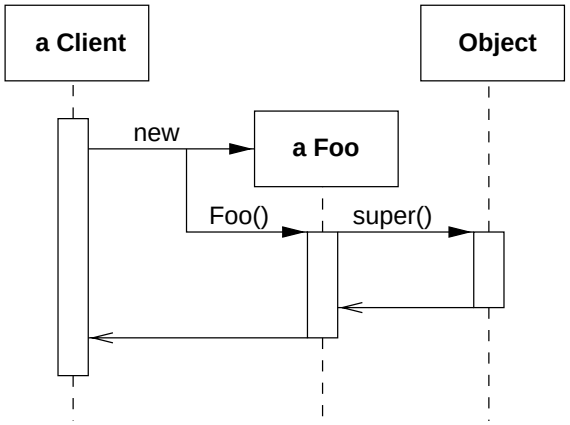


Abbildung 4.5: Erzeugen eines Objekts in Java

Die Schwierigkeit bei der Migration einer Instanzerzeugung liegt in der Vielfalt der Möglichkeiten, die Smalltalk zu Instanzerzeugung über das Meta–Objekt–Protokoll bietet. Innerhalb der

Konstruktormethode hat der Programmierer in Smalltalk mehr Freiheiten als in Java. In Java ist sichergestellt, daß vor dem Abarbeiten des Konstruktors der zu instanziierenden Klasse der Konstruktor der Superklasse aufgerufen wird. Diese Einschränkung in Java kann zu einer Aufteilung der Konstruktormethode führen bzw. die Berechnung von Parametern für den Konstruktor im Klientencode notwendig machen.

Initialisierung Smalltalk definiert kein Sprachkonstrukt zur Initialisierung von Instanzen einer Klasse. Daher verwendet eine Smalltalk–Applikation oft das Idiom *Constructor Method* (5.1.1). Die Erzeugung von Instanzen geschieht in Smalltalk durch Klassenmethoden. Diese enthalten den Initialisierungscode für Instanzen. Da Klassenmethoden an Subklassen vererbt werden, steht der Initialisierungscode auch in Subklassen zur Verfügung.

In Java kann eine Instanz durch zwei Mechanismen initialisiert werden. Der Konstruktor einer Instanz kann Initialisierungscode enthalten. Weil Konstruktoren allerdings nicht vererbt werden, steht der Initialisierungscode in einer Subklasse nicht ohne weiteres zur Verfügung. Hierzu muß die Subklasse den Konstruktor der Superklasse explizit aufrufen. Der zweite Mechanismus verwendet das Sprachkonstrukt eines instance initializers. Dieser enthält den Initialisierungscode einer Instanz. Instance initializers werden an Subklassen vererbt.

Die Identifizierung des Initialisierungsteils und die Eigenschaften der Instanzerzeugung in Java können bei der Migration von Smalltalkcode ein Problem sein. Die Identifizierung des Initialisierungscodes ist in Smalltalk aufgrund des Fehlens eines Sprachkonstruktes nicht immer einfach. Eine Konstruktormethode (5.1.1) kann Initialisierungscode enthalten. Falls in Smalltalk innerhalb der Konstruktormethode umfangreiche Berechnungen *vor* Aufruf des Konstruktors der Superklasse erfolgen, müssen diese in Java an anderer Stelle erfolgen. Evtl. muß der Klient diese Berechnungen vor der Instanzerzeugung ausführen, um sie dem Konstruktoraufruf als Argument mitzugeben.

Freigabe Smalltalk und Java besitzen beide eine automatische Speicherverwaltung (4.0.4), die nicht mehr referenzierte Objekte freigibt.

Das IBM Smalltalk Framwork definiert für Klassen die Methode `exiting`, die vor dem Ausschalten des Systems ausgeführt wird, um zusätzliche Ressourcen wie Dateien oder Netzwerkverbindungen freizugegeben. Ein gewöhnliche Instanz wird in Smalltalk nicht über ihre Freigabe informiert.

In Java ruft die Speicherverwaltung die `finalize()`–Methode vor Freigabe eines Objekts auf. Allerdings bezieht sich diese Methode im Gegensatz zu Smalltalk auf Instanzen. Java definiert keinen Mechanismus zum Aufruf von Applikationscode vor dem Verlassen des Systems.

Meistens läuft die Freigabe von Objekten in Smalltalk und Java vollständig transparent ab. Die Migration muß lediglich eingreifen, falls die Applikation in den Freigabeprozeß durch oben beschriebenen Mechanismus eingreift. Allerdings muß der klassenbasierte Aufruf von Smalltalk durch einen instanzbasierten Aufruf ersetzt werden.

Test der Funktionalität Die Funktionalität eines Objektes wird durch seine Klasse bestimmt [GR83, S.95]. In beiden Sprachen existieren Mechanismen zum Erfragen der Funktionalität einer Instanz.

In Smalltalk kann eine Applikation die Klasse eines Objekts bzw. die Zugehörigkeit eines Objekts zu einer Klasse abfragen. Daneben kann ermittelt werden, ob ein Objekt eine bestimmte Methode definiert. In Smalltalk genügt hierzu ein Aufruf von `respondsTo:`.

Eine Java–Applikation fragt diese Informationen über das Klassenobjekt, das in `Object` definierte Protokoll und die Operatoren `instanceof` und `==` ab.

Die in Smalltalk vorhandenen Tests der Funktionalität einer Instanz sind in Java ebenfalls verfügbar; sie werden lediglich auf andere Art angesprochen.

Fehlermethoden Die Klasse `Object` stellt in Smalltalk ein einfaches Protokoll zur Fehlerbehandlung bereit. Die darin behandelten Fehler sind zum Teil sehr smalltalk–spezifisch. Darüberhinaus definieren Smalltalk und Java einen zusätzlichen Mechanismus zur Ausnahmebehandlung, welcher in 4.3.4 beschrieben ist.

`doesNotUnderstand` zeigt in Smalltalk an, daß der Empfänger eine Nachricht erhalten hat, die er nicht verarbeiten kann. Aufgrund des dynamischen Typsystems stellt erst der Empfänger einer Nachricht fest, ob er eine Implementierung für die Nachricht kennt. Java kennt diese Fehlermethode prinzipiell nicht. Bei einem statischen Methodenaufruf stellt der Compiler fest, ob eine ungültige Methode aufgerufen werden soll. Bei einem dynamischen Methodenaufruf erzeugt die virtuelle Maschine eine `NoSuchMethodException`, falls der Empfänger die Methode nicht implementiert. Die Behandlung dieses Fehler wird in Smalltalk also vom Empfänger übernommen, währen sie in Java vom Sender übernommen wird. Manchmal ist der Aufruf dieser Methode in Smalltalk erwünscht, um zur Laufzeit Methodenaufrufe konfigurieren zu können wie z.B. im Proxy–Muster (5.2.8).

`primitiveFailed` zeigt an, daß die virtuelle Maschine eine Primitive nicht fehlerfrei verarbeiten konnte. Java erzeugt hier die Ausnahme `VirtualMachineError`. In Java ist die Ausnahmebehandlung an einen Codeblock gebunden, während sie in Smalltalk durch die Methode `primitiveFailed` für die gesamte Klasse definiert werden kann. Folglich muß jeder Aufruf des betroffenen Objekts in Java mit einem eigenen Ausnahmebehandler versehen werden.

`shouldNotImplement` stellt eine Einschränkung des Protokolls dar, das die Superklasse definiert. Java kennt diesen Mechanismus nicht. Das Verwenden dieser Methode deutet ein möglicherweise notwendiges Refaktorieren der Superklasse an. In der Smalltalk–Klassenbibliothek wird diese Methode manchmal verwendet, um das Protokoll der Superklasse einzuschränken (z.B. in der Collection–Hierarchie). Java löst das i.d.R. durch eine exaktere Aufteilung der definierten Interfaces.

`subclassResponsibility` zeigt in Smalltalk an, daß es sich um eine abstrakte Klasse handelt (siehe Klassenbeziehungen in Abschnitt 4.1.1). Abstrakte Klassen bzw. abstrakte Methoden werden in Java mit `abstract` markiert.

Eine solche Methode ist dazu gedacht, in den Subklassen überschrieben zu werden. In Smalltalk überprüft der Compiler allerdings nicht, ob die Methode tatsächlich überschrieben wurde. Daher kann es in Java nötig sein, eine Implementierung anzugeben, obwohl sie im Smalltalk–Programm fehlt.

Die in `Object` definierten Fehlermethoden sind z.T. sehr smalltalk–spezifisch und daher bei der Migration wie oben beschrieben besonders zu berücksichtigen.

Mutation

Smalltalk kennt die Methode `become:` zur Mutation eines Objekts. Diese Methode tauscht zwei Objekte und sämtliche darauf zeigenden Referenzen gegenseitig aus.

Java kennt dieses Konzept prinzipiell nicht. Ein Objekt kann nicht durch ein anderes ersetzt werden.

Die Verwendung von `become:` in Smalltalk führt bei der Migration zu größten Schwierigkeiten. Die Verwendung kann eine umfangreiche Refaktorierung des Codes nötig machen. Smalltalk–Applikationen verwenden `become:` beispielsweise zur Realisierung des Proxy–Musters (5.2.8). Dieses wird in Java vollständig anders implementiert.

Zeichen

Zeichen sind in Smalltalk vollwertige Objekte. D.h. sie können beispielsweise in einer Collection gespeichert werden.

In Java ist `char` in primitver Typ. Daher können Zeichen nicht ohne weiteres wie Objekte behandelt werden. Die Wrapperklasse `java.lang.Character` dient dazu, ein Zeichen als Objekt behandeln zu können. Die Verwendung von gewrappten Zeichen bedeutet allerdings mehr Schreibarbeit als bei den Literalen selbst.

Da die Zeichenklasse in Smalltalk keine Methoden zum Verändern eines Zeichens definiert, ist die Verwendung eines Zeichens in Smalltalk sehr ähnlich zu der in Java. Lediglich wenn die Objekteigenschaften eines Zeichens in Smalltalk verwendet werden, muß das verwendete Zeichen in seine Wrapperklassen eingepackt und später wieder ausgepackt werden. Ansonsten unterschieden sich Zeichen im wesentlichen durch die Darstellung der Literale (siehe Literale in Abschnitt 4.1.1).

Boolesche Werte

Boolesche Werte sind in Smalltalk ebenfalls Objekte. Allerdings sind die beiden vom System erzeugten Instanzen `true` und `false` nicht veränderbar. Meist wird die Objekt–Eigenschaft von booleschen Werten in Smalltalk nicht ausgenutzt.

In Java sind boolesche Werte Elemente des primitiven Typs `boolean`. Sie können nicht wie vollwertige Objekte behandelt werden (beispielsweise bei Speicherung in einer Collection). Hierzu benötigt man die Wrapperklasse `java.lang.Boolean`. Repräsentiert werden boolesche Werte in beiden Sprachen durch die Literale `true` und `false`.

Da die Verwendung von booleschen Werten in Smalltalk deren Objekteigenschaft nur selten ausnutzt, ist die Migration meist unproblematisch. Bei Verwendung als Objekt muß der Wert in die Wrapperklasse eingepackt und später ausgepackt werden.

Pseudovariablen

Pseudovariablen sind in Smalltalk Sprachkonstrukte, die zwar lesend wie gewöhnliche Variablen behandelt werden können, deren Wert aber von der Laufzeitumgebung gesetzt wird. Smalltalk definiert folgende Pseudovariablen:

Null–Objekt Smalltalk behandelt das Nullobjekt `nil` wie jedes andere Objekt. Eine Java–Applikation kann `null` ebenfalls anstelle eines beliebigen Objekts verwenden. In Smalltalk unterstützt `nil` das Protokoll zum Anlegen einer Klasse, die keine Superklasse besitzt. Beispielsweise nutzt das Proxy–Muster (5.2.8) diese Eigenschaft. In Java hat jede Klasse eine Superklasse. Diese Verwendung führt also zu einer größeren Umstrukturierung, allerdings vom Einzelfall abhängt.

Selbstreferenz Eine Methode greift auf die eigene Instanz über `self` zu. Dieser Zugriff funktioniert in Instanz– und Klassenmethoden. In Java kann `this` nur in Instanzmethoden verwendet werden, weil der Zugriff auf das Klassenobjekt in dieser Form nicht definiert ist. Die Verwendung von `this` ist in Java nicht nötig, weil die Klassenmethoden ohne die Angabe eines Empfängers aufgerufen werden können.

Referenz auf Superklasse `super` erlaubt in Smalltalk den direkten Zugriff auf eine Methode, welche in der Superklasse implementiert ist. Dieser Aufruf wird meist verwendet, wenn die Subklasse die geerbte Methode überschrieben hat, um auf deren Funktionalität zuzugreifen. In Java können nur Instanzmethoden `super()` verwenden, weil der Zugriff auf das Klassenobjekt in dieser Form nicht definiert ist. Die in Smalltalk häufigste Verwendung von `super` in Klassenmethoden

geschieht in Konstruktormethoden. Dieser Aufruf erfolgt in Java innerhalb des Konstruktors. Weil in Java der Konstruktor der Superklasse jedoch automatisch aufgerufen wird, ist dieser Aufruf in Java meist überflüssig. In Java ist die Verwendung `super()` in Klassenmethoden nicht erlaubt.

4.1.3 Valuable

Die Valuable–Protokollklasse des ANSI–Standards beschreibt Objekte, die mit einer Variante der Nachricht `value` ausgewertet werden können. Für die hier betrachtete Smalltalk–Plattform sind dies ausschließlich Codeblöcke.

Die Migration von Blöcken gestaltet sich schwierig, weil sich das Konzept nicht eins-zu-eins nach Java übertragen läßt. Je nach Art der Anwendung eines Blockes in Smalltalk muß eine Applikation ein anderes Sprachkonstrukt in Java wählen, um die gleiche Semantik zu implementieren.

In diesem Abschnitt werden Blöcke zunächst definiert. Anschließend werden verschiedene Einsatzbereiche von Blöcken in Smalltalk vorgestellt.

Was ist ein Block?

Ein Smalltalk–Block repräsentiert eine Sequenz von Aktionen, deren Auswertung erst zu einem späteren Zeitpunkt erfolgt [GR83, S.31]. Den Wert erfährt man durch Versenden der Nachricht `value`. Blöcke sind in Smalltalk Objekte erster Klasse. D.h. durch das Block–Konzept ist es möglich, ein Stück Code zu speichern oder als Argument einer Nachricht zu verwenden. Die in Java definierten Code–Blöcke dienen lediglich der Strukturierung des Quellcodes. Sie sind keine Objekte erster Klasse.

In Java sind Instanzen einer inneren Klasse am ehesten geeignet, das Verhalten von Smalltalk–Blöcken nachzuempfinden. Eine Anwendung kann eine Klasse als innere Klasse definieren. Das bedeutet, daß Instanzen dieser inneren Klasse im Kontext der Instanz der umschließenden Klasse existieren. Ein solches Objekt hat folglich Zugang zu den privaten Variablen und Methoden des Objekts. Um in manchen Fällen Schreibarbeit zu sparen, besitzt Java die Möglichkeit, eine anonyme innere Klasse zu definieren. Eine solche Klasse wird nicht über einen Namen angesprochen, sondern definiert und direkt instanziiert.

Die Darstellung von Codeblöcken durch Instanzen innerer Klassen benötigt zusätzlich ein Interface, welches die Kommunikation zwischen den kommunizierenden Objekten regelt (siehe Abbildung 4.6). Dieses Interface definiert die Methode, welche vom Empfänger des Blockobjekts aufgerufen wird. Ohne das Interface wüßte der Empfänger nicht, welche Methode des Blockobjekts er aufrufen müßte.

Kontrollstrukturen

Die fundamentale Kontrollstruktur von Smalltalk ist die Sequenz [GR83, S.32], die eine Reihe von Ausdrücken sequentiell verarbeitet. Alle anderen Kontrollstrukturen in Smalltalk sind nicht Bestandteil der Sprache, sondern eine Kombination von Nachrichten und Codeblöcken. Dies sind Auswahl und bedingte Wiederholung.

In Java sind Kontrollstrukturen Bestandteil der Sprache [Gra97, S.180].

Die Migration von Kontrollstrukturen ist kein Problem. Die verwendeten Methodenaufrufe zur Realisierung einer Kontrollstruktur sind eindeutig. Der Code des Blocks wird in Java zum Rumpf der Kontrollstruktur. Der Block innerhalb einer Kontrollstruktur in Java definiert einen formalen Parameter, um beispielsweise den Schleifenzähler zu erhalten. Dieser Parameter wird in Java durch eine temporäre Variable ersetzt und ebenfalls als Schleifenzähler eingesetzt.

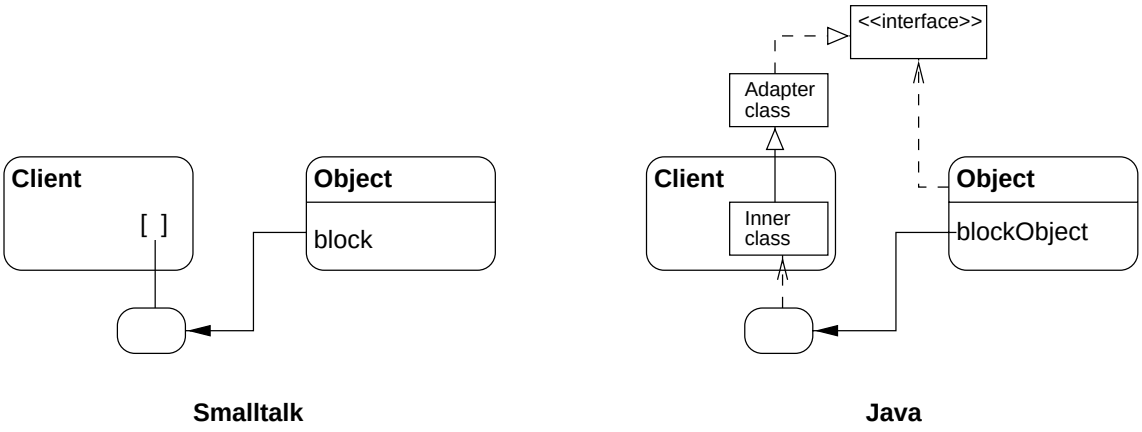


Abbildung 4.6: Ersatz eines Blocks durch Instanz einer inneren Klasse

Multiprogrammierung

Die Ausführungseinheiten von parallel laufenden Prozessen sind Blöcke. Multiprogrammierung wird in 4.1.5 behandelt.

Ausnahmebehandlung

Blöcke sind die Smalltalk–Konstrukte, für die eine Anwendung einen Ausnahmebehandler registrieren kann. Ausnahmebehandlung wird in 4.3.4 diskutiert.

4.1.4 Zahlen

Smalltalk kapselt die interne Darstellung von Zahlen bei arithmetischen Berechnungen weitgehend. Eine Java–Applikation muß hier sehr viel mehr selbst machen. Die bedeutet einen Aufwand bei der Migration, weil die Umwandlungen, die im Smalltalk–Code nicht zu sehen sind, in Java explizit gemacht werden müssen.

Dieser Abschnitt stellt zunächst die verfügbaren *Datentypen* vor und bespricht anschließend *Typumwandlungen*.

Verfügbare Datentypen

Die in Smalltalk verfügbaren Datentypen sind Bestandteil der Klassenbibliothek. In Java verteilen sich die Datentypen auf die primitiven Typen, die Bestandteil der Sprache sind und die Klassen des Package `java.math`.

Rationale Zahlen Die Klasse `Integer` repräsentiert rationale Zahlen. Die beiden Subklassen `SmallInteger` und `LargeInteger` dienen der Unterscheidung in der Ausführungsgeschwindigkeit und Genauigkeit. `SmallInteger` hat einen eingeschränkten Wertebereich, führt Berechnungen aber schneller aus als `LargeInteger`, welche keinen Einschränkungen des Wertebereichs unterliegt. Smalltalk wählt die jeweils beste Darstellung automatisch aus [GR83, S.127].

Java definiert die Datentypen `byte`, `short`, `int` und `long`, welche alle nur über einen eingeschränkten Wertebereich verfügen. Die Klasse `BigInteger` definiert rationale Zahlen in Java, die

keiner Beschränkung des Wertebereichs unterliegen. Die primitiven Elemente werden mit Operatoren verwendet; bei der Klasse müssen Methoden aufgerufen werden.

Reelle Zahlen `Float` ist in Smalltalk durch Mantisse und Exponent beschränkt und ist dadurch mit `float` und `double` in Java vergleichbar. `Fraction` ist ein Repräsentation von rationalen Zahlen in Smalltalk, die immer korrekt ist [GR83, S.127]. In Java ist kein vergleichbarer Datentyp definiert. IBM Smalltalk definiert zusätzlich die Klasse `Decimal`, welche mit Festkommazahlen bis zu 30 Nachkommastellen exakt rechnen kann. Die Klasse `BigDecimal` erlaubt in Java das Rechnen mit rationalen Zahlen beliebiger Länge und Genauigkeit.

Typumwandlungen

Typumwandlungen geschehen in Smalltalk automatisch. Die Methoden `coerce`, `generality` und `retry` dienen der expliziten Typumwandlung, werden in Smalltalk–Applikationen aber kaum verwendet. IBM Smalltalk ersetzt diese Methoden durch das Protokoll `lessGeneralThan` und `moreGeneralThan`, um Darstellungsdetails noch weiter vor dem Programmierer zu verbergen.

Eine Java–Applikation muß jede Typumwandlung explizit angeben. D.h. die Stellen an denen in Smalltalk eine automatische Typumwandlung geschieht, müssen bei der Migration erkannt werden. Außerdem geschieht die Typumwandlung in Java immer durch Angabe des gewünschten Typs.

4.1.5 Multiprogrammierung

In diesem Abschnitt werden Smalltalk–Prozesse mit Java–Threads verglichen. In beiden Fällen wird der Prozeß, in dem die Applikation läuft, nicht verlassen. Kommunikation mit anderen Prozessen im Betriebssystem sind Thema des Abschnitts 4.1.6. Dieser Abschnitt behandelt die *Ablaufarchitektur*, das *Erzeugen von Prozessen*, die *Synchronisation* von Prozessen, die *Kommunikation* dazwischen und den *Mechanismus der zeitlichen Verzögerung*.

Ablaufarchitektur

Die Ablaufeinheit eines Prozesses in Smalltalk ist ein Block. Ein Block läuft im Sichtbarkeitskontext der erzeugenden Methode und hat somit Zugriff auf deren lokale Variablen. Jeder parallel ablaufende Block wird in Smalltalk als Prozeß bezeichnet. Der `ProcessScheduler` verteilt die Rechenzeit der virtuellen Maschine auf die aktiven Smalltalk–Prozesse innerhalb des Betriebssystem–Prozesses (siehe Abbildung 4.7).

Die Ablaufeinheit eines Threads in Java ist eine Instanz. Jeder Thread verfügt somit über seinen eigenen lokalen Speicher. Zusätzlich kann ein Thread auf die Klassenvariablen seiner Klasse zugreifen (siehe Abbildung 4.7). Damit ist der Datenaustausch zwischen mehreren Threads möglich. Java implementiert die Funktionalität des Schedulers in der virtuellen Maschine (JVM). Es bleibt der JVM überlassen, die Java–Threads mit Hilfe von Betriebssystem–Threads zu realisieren oder einen eigenen Scheduling–Mechanismus zu implementieren. Für die Verwendung von Threads spielt diese Implementierungsentscheidung keine Rolle.

Für die Migration ist entscheidend, daß die Ablaufeinheiten in Smalltalk und Java grundsätzlich verschiedener Natur sind. Der Code eines parallel ablaufenden Blockes und sämtliche Methoden, die ausschließlich von diesem Block aus aufgerufen werden, kommen in Java in eine eigene Klasse. Diese Klasse muß entweder das `Runnable`–Interface implementieren oder von der Klasse `Thread` abgeleitet sein. Da die Kommunikation zwischen mehreren Blöcken nicht mehr über lokale Variablen geschieht, müssen zu diesem Zweck Klassenvariablen angelegt werden oder andere Mechanismen zu Thread–Kommunikation verwendet werden.

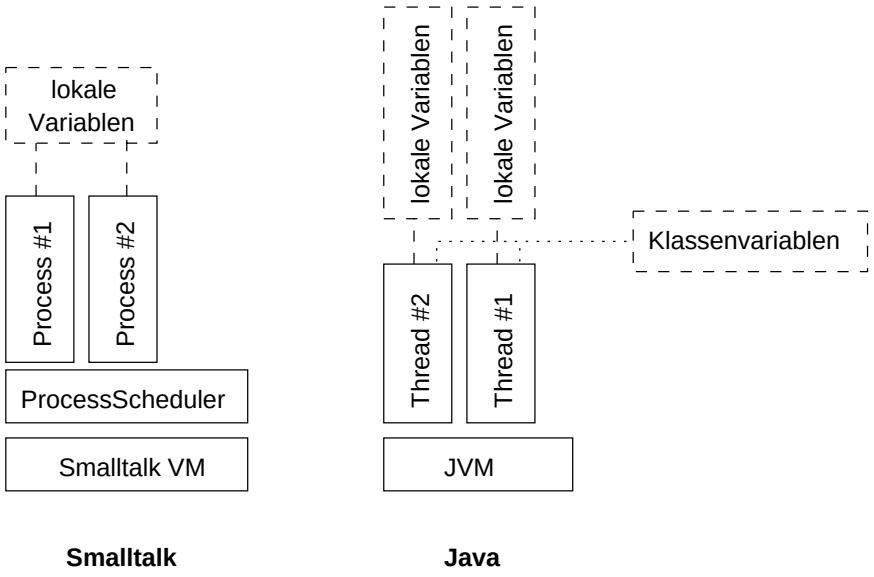


Abbildung 4.7: Prozesse und Threads

Erzeugen eines Prozesses

Ein Block wird in Smalltalk durch die Nachricht `fork` zum Prozeß. Die Abarbeitung der Anweisungen im Prozeß geschieht genau wie bei der Nachricht `value`. Der Unterschied ist, daß `value` wartet, bis das Ergebnis des Blockes vorliegt, während `fork` sofort zurückkehrt.

Ein Java–Thread ist eine Instanz von `Thread` oder einer Klasse, die das `Runnable`–Interface implementiert. Die Applikation startet den Thread durch die Nachricht `run()`.

Da sich die Ablaufeinheit ändert, ist auch die Erzeugung von Threads grundsätzlich anders.

Synchronisation

Die Synchronisation von Prozessen dient der Koordinierung des Zugriffs auf gemeinsame Ressourcen mehrerer parallel ablaufender Prozesse.

In Smalltalk dienen Instanzen von `Semaphore` der Synchronisation von Prozessen. Die Methoden `wait` und `signal` definieren das Semaphor–Protokoll. Gegenseitigen Ausschluß stellt eine Smalltalk–Applikation durch die Methode `critical:` dar, die einen Block vor gleichzeitigem Zugriff schützen kann.

In Java erfüllt jedes Objekt die Semaphor–Semantik mit den Methoden `wait()` und `notify()`. Den Schutz vor gleichzeitigem Zugriff erreicht Java durch das Schlüsselwort `synchronized`, welches sich auf Methoden und Blöcke beziehen kann.

Die Synchronisationsmechanismen in Smalltalk setzen der Migration nach Java keinen Widerstand entgegen.

Kommunikation

Zur sicheren Kommunikation zwischen zwei Prozessen definiert Smalltalk neben den Synchronisationsmechanismen die Klasse `SharedQueue`.

Die Java–Plattform kennt keine vergleichbare Klasse, kann die selbe Semantik aber mit Hilfe einer `synchronized` Collection darstellen, die als Queue verwendet wird [OW97, S.88].

Obwohl die Smalltalk–Klasse in Java nicht definiert ist, läßt sich der Mechanismus in Java leicht nachbilden.

Zeitliche Verzögerung

Eine zeitliche Verzögerung erzeugt eine Smalltalk–Applikation mit einer Instanz von `Delay`, welche eine Auflösung im Millisekunden–Bereich hat.

Eine Java–Applikation verwendet hierfür keine eigene Klasse, sondern die Methode `sleep()`, welche für alle Objekte definiert ist.

Die in Smalltalk vorhandenen Mechanismen zur Multiprogrammierung lassen sich alle nach Java übertragen, auch wenn die Implementierung z.T. sehr verschieden erfolgt. Die unterschiedlichen Ablaufeinheiten (Block in Smalltalk im Gegensatz zu Klasse in Java) bedeuten einige Arbeit. Allerdings ergibt sich daraus kein prinzipelles Problem bei der Migration.

4.1.6 IPC

IBM Smalltalk faßt den Mechanismus zum Zugriff auf Betriebssystem–Prozesse und Umgebungsvariablen unter dem Begriff „Interprozeßkommunikation“ zusammen. Diese Prozeßbehandlung steht im Gegensatz zu Prozessen und Threads innerhalb des Smalltalk– bzw. Java–Prozesses (siehe 4.1.5).

Betriebssystem–Prozesse

Um Funktionen des Betriebssystems zu nutzen, wird in Smalltalk ein eigener Betriebssystem–Prozeß erzeugt (`UNIXProcess`). Mit diesem Prozeß wird über Ströme kommuniziert, die den C-Strömen `stdin`, `stdout` und `stderr` entsprechen (`UNIXReadPipeStream` und `UNIXWritePipeStream`).

In Java beschreibt die Klasse `Process` einen Prozeß, der außerhalb der Java–Umgebung läuft. Dieser wird durch `Runtime.exec()` erzeugt. Auch dieser Prozeß definiert einen Eingabe–, einen Ausgabe– und einen Fehlerstrom — allerdings in plattformunabhängiger Weise.

Der Zugriff auf Betriebssystem–Prozesse ist in Smalltalk sehr eng am Prozeßmodell von UNIX orientiert. Obwohl Java an dieser Stelle mehr Wert auf Betriebssystemunabhängigkeit legt, ist die Verwandtschaft zum UNIX–Modell so groß, daß die Migration kein großes Problem ist.

Zugriff auf Umgebungsvariablen

Die Umgebung eines Prozesses unter UNIX besteht aus einer Menge an Paaren der Form `aKey = aValue`. Mit Hilfe dieser Variablen werden z.B. der Suchpfad, das Heimatverzeichnis etc. gesetzt. `UNIXEnvironment` ist die Smalltalk–Schnittstelle zu diesen Umgebungsvariablen.

Der Zugriff auf Umgebungsvariablen ist in Java über die Klasse `Property` plattformunabhängig gehalten. `System.getProperty()` bzw. `System.getProperties()` holt die Umgebungsvariablen.

Der Zugriff auf Umgebungsvariablen des Betriebssystems ist in Smalltalk sehr eng an UNIX orientiert. Obwohl Java an dieser Stelle mehr Wert auf Betriebssystemunabhängigkeit legt, ist die Verwandtschaft zum UNIX–Modell so groß, daß die Migration kein großes Problem ist.

4.1.7 Reflexion

Reflexion ist die Eigenschaft einer Sprache, einer Applikation den Zugriff auf Informationen über sie selbst zu ermöglichen. Smalltalk wie Java besitzen reflexive Eigenschaften, die sich jedoch deutlich voneinander unterscheiden. Reflexion ist ein Architekturmuster, welches in 5.3.2 beschrieben wird.

In Smalltalk werden die reflexiven Eigenschaften durch das Meta–Objekt–Protokoll realisiert. Das Meta–Objekt–Protokoll von Smalltalk ist vorwiegend dazu gedacht, Entwicklungswerkzeuge zu bauen. Für die Applikationsentwicklung ist das Metaobjekt–Protokoll, abgesehen von den reflexiven Eigenschaften, von untergeordneter Bedeutung (manche Probleme lassen sich über das Meta–Objekt–Protokoll jedoch sehr elegant lösen).

Zentral für das Funktionieren des Meta–Objekt–Protokolls ist das Konzept der Metaklasse. Weil alle Objekte in Smalltalk die Instanz einer Klasse sind, sind Klassen selbst Instanzen ihrer Metaklasse. Metaklassen verhalten sich wie gewöhnliche Klassen und sind daher auch in einer Vererbungshierarchie angeordnet. Diese Vererbungshierarchie der Metaklassen liegt parallel der Vererbungshierarchie der gewöhnlichen Klassen. Aufgabe der Metaklassen ist beispielsweise die Instanzerzeugung. Instanzerzeugung erfolgt in Smalltalk nicht durch ein Sprachkonstrukt, sondern wird in der Metaklasse `Behaviour` von einer Methode implementiert. Über das Meta–Objekt–Protokoll kann das Laufzeitsystem selbst verändert werden. Beispielsweise kann damit für eine beliebige Klasse ein spezifischer Compiler registriert werden. Für die Migration nach Java sind jedoch die reflexiven Eigenschaften interessant. Über das Meta–Objekt–Protokoll können die Variablen, Methoden, Subklassen, etc. einer Klasse abgefragt werden.

Java kennt das Konzept der Metaklasse nicht. Allerdings bietet das Reflection–API die Möglichkeit zum Zugriff auf Metainformation der bestehenden Klassen. Damit können die Variablen, Methoden, implementierten Interfaces, etc. einer gegebenen Klasse erfragt werden.

Applikationen, die das Meta–Objekt–Protokoll intensiv verwenden, lassen sich kaum nach Java migrieren. Lediglich der Abruf von Metainformation ist mit dem Reflection–API in Java nachzubilden.

4.1.8 Zusammenfassung

Smalltalk und Java weisen auf sprachlicher Ebene eine Reihe an Unterschieden auf:

- Der augenfälligste Unterschied ist die Typisierung von Variablen und Parametern, welche in Smalltalk dynamisch getypt sind, während sie in Java statisch getypt sind. Dieser Unterschied führt an vielen Stellen zu einer komplizierteren Lösung in Java aber auch zu einer besseren Überprüfbarkeit der Typisierung durch den Compiler.

- Java unterscheidet zwischen primitiven Typen und Referenztypen, während in Smalltalk ausschließlich Referenztypen definiert sind. Das macht in manchen Fällen die Verwendung von Wrapper–Klassen notwendig.

- Die Erzeugung von Objekten geschieht in Smalltalk über Klassenmethoden, während hierfür in Java Konstruktoren definiert sind. Dies führt zu einer Umstrukturierung des Erzeugungs– und Initialisierungscodes in der Klasse und zu leichten Modifikationen bei Klienten dieser Klasse.

- Die Verwendung von Objekt–Mutation in Smalltalk erfordert zum Teil umfangreiche Veränderungen in den verwendenden Klassen.

- Die Ausführung eines dynamischen Methodenaufrufs wird in Java, abhängig vom Einzelfall, durch verschiedene Mechanismen abgebildet. Dies hat allerdings keinen großen Einfluß auf die Gesamtarchitektur der Applikation.

- Eine umfangreiche Verwendung des Meta–Objekt–Protokolls macht meist eine komplette Neuentwicklung in Java notwendig.

4.2 Vergleich auf Toolkit–Ebene

In diesem Abschnitt werden die Klassen der Klassenbibliothek untersucht, die dem Toolkit zuzurechnen sind. Sie zeichnen sich dadurch aus, daß sie oft benötigte Funktionalität in wiederverwendbaren Klassen anbieten, aber nicht dirket zur Definition der Sprache benötigt werden

4.2.1 Collections

Eine Collection ist ein Behälter für mehrere Objekte. Smalltalk und Java bieten jeweils eine Reihe an Collections an.

Die Collections zählen zu den wichtigsten Teilen des Smalltalk–Toolkits. Applikationen verwenden Collections sehr oft aufgrund ihrer universellen Einsetzbarkeit. Die verschiedenen in Smalltalk vorhandenen Collections unterscheiden sich voneinander darin, wie die enthaltenen Objekte organisiert sind und über welches Protokoll auf sie zugegriffen werden kann.

Im einfachsten Fall verwendet eine Java–Applikation die Klassen `Vector`, `Stack` und `Hashtable` zur Darstellung einer Sammlung an Objekten. Darüberhinaus definiert das Collections–API ein umfangreiches Framework zur Erstellung und Verwendung von Collections [Sof98]. Java stellt nicht nur ein Collection–Toolkit zur Verfügung, das mit dem in Smalltalk vergleichbar ist. Es bietet auch ein Framework zur Einwicklung eigener Collections, die sich nahtlos in das bestehende Toolkit einfügen können.

Collection–Protokoll

Die `Collection`–Klasse ist die Wurzel der Collection–Hierarchie. Sie definiert das Protokoll, das alle Smalltalk–Collections verstehen. Das Collection–Protokoll definiert Methoden zum Zugriff auf die Elemente und zur Konvertierung in eine andere Klasse. Außerdem sind interne Iteratoren definiert, um eine Möglichkeit zu haben, über die Elemente zu iterieren. Daneben erweitert jede Subklasse das Protokoll um spezifische Methoden. Externe Iteratoren werden durch Ströme dargestellt (siehe 4.2.3).

In Java ist das Collection–Protokoll durch mehrere Interfaces definiert. Das `Collection`–Interface ist die Wurzel der Collection–Hierarchie. Weitere Interfaces erweitern dieses Protokoll durch spezifische Methoden. Das Collection–Toolkit enthält neben diesen Interfaces gebräuchliche Standardimplementierungen. Außerhalb des Collection–Toolkit stehen die Klassen `Vector`, `Stack` und `Hashtable`, welche zwar eine ähnliche Funktionalität wie die Collections aufweisen, deren Protokoll aber nicht verstehen. Java kennt ausschließlich externe Iteratoren, die durch die Klassen `Enumeration`, `Iterator` und `ListIterator` implementiert werden.

Da das Protokoll des Collection–Toolkit in Java dem Collection–Protokoll von Smalltalk sehr nahe kommt, ist bei der Migration mit keinen großen Problemen zu rechnen.

Collection–Klassen

Für die meisten Klassen der Collection–Hierarchie in Smalltalk gibt es ein Äquivalent in Java (siehe Tabelle 4.5).

Allerdings zählen nicht alle Gegenstücke der Klassen der Collection–Hierarchie von Smalltalk in Java zum Collection–Toolkit. Im einzelnen sind dies:

`String` Die Klasse `String` geört in Java nicht zur Collection–Hierarchie, sondern ist eigenständiger Bestandteil von `java.util`, worin auch `StringBuffer` definiert ist. Diese Klasse unterstützt Stringmanipulationen.

Smalltalk	Java	Bemerkung
`Linked List`	`LinkedList`	
`Array`	`ArrayList`	und primitives Array
`RunArray`	`ArrayList`	nicht so effizient
`String`	`String`	
`Symbol`	`String`	keine Eindeutigkeit
`Text`		Package `java.lang.text`
`ByteArray`	`ArrayList`	nicht so effizient
`Interval`		nicht benötigt
`OrderedCollection`	`ArrayList`	
`SortedCollection`	`TreeSet`	
`Bag`	`ArrayList`	
`MappedCollection`		Implementierung von `Map` als Wrapper
`Set`	`HashSet`	
`Dictionary`	`HashMap`	
`IdentityDictionary`	`HashMap`	keine Eindeutigkeit

Tabelle 4.5: Abbildung von Smalltalk–Collections

`Symbol` Die Klasse `Symbol` ist in Java kein Bestandteil der Sprache. Eine Java–Applikation wird daher `String` verwenden und die Eindeutigkeit selbst garantieren müssen.

`Interval` Java definiert keine Intervalle. Ein Intervall wird in Java durch eine Schleife dargestellt.

`MappedCollection` Eine mapped collection bildet ihre Elemente auf die Elemente einer darunterliegenden Collection ab. Das Konzept einer mapped collection kennt Java nicht. Allerdings kann eine Applikation das `Map`–Interface des Collection–Toolkits verwenden, um einen Wrapper zu implementieren.

4.2.2 Datum und Uhrzeit

Die `Date` und `Time`–Klassen in Smalltalk sind Teil der `Magnitude`–Subhierarchie, die eine Ordnung auf den Elementen ermöglicht. Ein `Date` repräsentiert einen spezifischen Tag im julianischen Kalender. `Time` ist die Zeit in Sekunden seit Mitternacht. [GR83, S.107]

Die Java–Klasse `Date` vereinigt die beiden oben beschriebenen Smalltalk–Klassen. `Calendar`, `TimeZone` und `DateFormat` bieten das Umfeld für die Datumsrechnung. Java liefert als Standardimplementierung des `Calendar`–Interface den `GregorianCalendar`.

Datum und Uhrzeit machen bezüglich Migration keine Schwierigkeiten.

4.2.3 Ströme

Streams dienen dazu, über die Elemente einer `Collection` zu iterieren [GR83, S.195].

Allerdings sind sie im Gegensatz zu den von der `Collection`–Klasse angebotenen internen Iteratoren unterbrechbar. D.h. eine Instanz von `Stream` merkt sich das letzte Element, auf das zugegriffen wurde. Streams sind somit viel fexibler zu verwenden als die von `Collection` angebotenen internen Iteratoren.

Java definiert die externen Iteratoren `Enumeration`, `Iterator` und `ListIterator`. Die Klasse `Enumeration` kann ausschließlich lesend auf die Elemente zugreifen, über die sie iteriert. Die Klas-

se `Iterator` erlaubt zusätzlich das Löschen eines Elementes. Ein `ListIterator` kann außerdem Elemente einfügen, erlaubt jedoch keine Positionierung auf der zugrundeliegenden Collection wie bei `PositionableStream` in Smalltalk.

Die Funktionalität von Strömen wird in Java nur teilweise abgedeckt. Bei Bedarf muß der Klient einer Collection durch direkten Zugriff auf die Collectionelemente das Verhalten von Smalltalk nachbilden.

4.2.4 Dateiströme

Auf beiden Plattformen sind die Zugriffsmechanismen auf externe Dateien definiert.

IBM Smalltalk bietet eine Implementierung des POSIX.1–Standard, um auf Dateien in portabler Art und Weise zuzugreifen. Darauf aufbauend wurden die in [GR83] beschriebenen Dateiströme realisiert. Die Zeichenkodierung des zugrundeliegenden Dateisystems ist für den Smalltalk–Programmierer transparent gehalten. Die Dateiströme in Smalltalk sind byte–weise stromorientiert. Ein wahlfreier Zugriff auf den Inhalt einer Datei ist über die primitiven Dateimethoden möglich.

Auch eine Java–Applikation greift auf Dateien über Dateiströme zu. Sie sind ebenfalls byte–orientiert. Zusätzlich können Filter in einen Dateistrom eingebaut werden. Diese Filter erlauben beispielsweise eine Konversion in das Unicode–Format oder das direkte Abspeichern eines Objekts. Zusätzlich zu Dateiströmen ist ebenfalls ein wahlfreier Zugriff auf den Inhalt einer Datei möglich. Zur Kommunikation zwischen Threads dienen Pipeline–Ströme. Die Klassen in `java.io` sind so ausgelegt, daß auf die plattformspezifischen Eigenschaften in transparenter Art und Weise zugegriffen werden kann [Fla97, S.396].

Dateiströme haben in Smalltalk und Java ähnliche Eigenschaften wie Byte–Orientierung. Allerdings muß der Dateizugriff aufgrund des unterschiedlichen Aufbaus der zum Dateizugriff verwendeten Objekte komplett neu geschrieben werden, was jedoch keinen großen Einfluß auf die Architektur der Applikation hat.

4.2.5 Random

Smalltalk wie Java definieren einen Mechanismus zum Erzeugen von Zufallszahlen.

Der Smalltalk–Klasse `Random` stehen in Java die Methode `Math.random()` und die Klasse `java.util.Random` gegenüber. Beide Plattformen erzeugen eine Reihe von Pseudozufallszahlen aufgrund eines Startwertes der zu Beginn gesetzt wird. Java kann Zufallszahlen erzeugen, die einer Gaußverteilung gehorchen.

Die Migration von Zufallszahlen birgt keine großen Schwierigkeiten.

4.2.6 Zusammenfassung

Bis auf kleine Ausnahmen ist die Verwendung des Smalltalk–Toolkits nach Java leicht zu bewerkstelligen. Im einzelnen bleibt festzuhalten:

- Die Klassen der Collection–Hierarchie können ausreichend gut durch die Klassen des Collection–Toolkits in Java ersetzt werden. Einige Klassen der Collection–Hierarchie sind in Java an anderer Stelle implementiert oder existieren in Java nicht.

- Datum und Uhrzeit sind für die Migration kein Problem.

- Die Funktionalität der vom Stream–Toolkit definierten Iteratoren wird von den in Java definierten externen Iteratoren nur teilweise abgedeckt. Aufgrund der fehlenden Positionierbarkeit der Iteratoren in Java kann es nötig sein, die zugrundeliegende Collection offenzulegen.

- Der Zugriff auf Dateien muß komplett neugeschrieben werden, allerdings ohne großen Einfluß auf die Gesamtarchitektur.
- Zufallszahlen sind für die Migration kein Problem.

4.3 Vergleich auf Framework–Ebene

In diesem Abschnitt werden die Klassen der Klassenbibliothek untersucht, die in Form von Frameworks in Erscheinung treten. Die Klassen der Framework–Ebene unterscheiden sich von Hersteller zu Hersteller. Aus diesem Grund ist der angestellte Vergleich sehr spezifisch für IBM Smalltalk und nicht ohne weiteres auf andere Smalltalk–Dialekte (Bsp. VisualWorks) übertragbar.

4.3.1 Objekt–Framework

Die Klasse `Object` ist in beiden Sprachen die Wurzel der Klassenhierarchie. Das in `Object` definierte Protokoll gilt also für alle Objekte des Systems. Dieser Abschnitt betrachtet die Protokolle zum *Kopieren*, *Drucken* und *Speichern* eines Objekts und zum *Vergleichen zweier Objekte*. Für die sprachrelevanten Eigenschaften von `Object` siehe 4.1.2.

Kopieren Eine Zuweisung von Referenztypen kopiert in Smalltalk und Java lediglich die Referenz. Daneben bieten beide Sprachen einen Mechanismus zum Kopieren des Objekts selbst.

Durch den Aufruf `shallowCopy`, wird in Smalltalk nur der Empfänger dieser Nachricht kopiert, wohingegen durch `deepCopy` auch die Inhalte dessen Variablen kopiert werden. Zusätzlich bietet das Protokoll `copy` an, welches standardmäßig `shallowCopy` verwendet. Falls beim Kopieren eines Objekts nur manche Variablen kopiert werden, muß `copy` überschrieben werden.

In Java existiert ausschließlich die Methode `clone()`. Kopierbare Objekte müssen mit dem `Cloneable`–Interface markiert werden. Die Standardimplementierung von `clone()` verhält sich wie `shallowCopy`. Ein `deepCopy` kann also nur durch ein `Überschreiben` von `clone()` erreicht werden. `Object` deklariert die Methode `clone` als `protected`. Da die meisten Klassen der Klassenbibliothek diese Methode nicht überschreiben bzw. die Methode nicht als `public` deklarieren, können nur abgeleitete Klassen eine Kopie anlegen [Eck98, S.439].

Weil Java nur eine Methode zum Kopieren eines Objekts definiert, muß bei der Migration die fehlende Funktionalität nachgebildet werden. Dies führt zu Problemen, falls in Smalltalk nicht durchgängig immer die eine Methode verwendet wird, sondern beide innerhalb der gleichen Klasse zum Einsatz kommen. Eine Klasse muß sich in Java auf eine Kopiervariante festlegen. Außerdem sind einige Objekte der Java–Klassenbibliothek nicht direkt kopierbar. Hingegen ist in Smalltalk jedes Objekt kopierbar. Dadurch können Probleme bei einem Nachbilden von `deepCopy` entstehen.

Beschreibung Bezüglich Erzeugen der Beschreibung eines Objekts sind Smalltalk und Java sehr ähnlich. In Smalltalk muß eine Applikation die Methoden `printString` und `printOn:` überschreiben. Java definiert hierfür die Methode `toString()`.

Speichern Beide Sprachen definieren einen Mechanismus zum Speichern eines Objekts.

In Smalltalk definieren die Methoden `storeString` und `storeOn:` das Protokoll zum Erzeugen eines Strings, aus dem das abgespeicherte Objekt wiederhergestellt werden kann.

In Java speichert eine Applikation Objekte mit Hilfe der *Object Serialization*. Ein `ObjectOutputStream` erlaubt die Umwandlung eines Objekts in einen Zeichenstrom. Ein `ObjectInputStream` kann daraus das ursprüngliche Objekt wiederherstellen.

Die Mechanismen zum Abspeichern eines Objekts unterscheiden sich in Smalltalk und Java sehr und werden unterschiedlich verwendet. Für die Migration ist das kein Problem.

Äquivalenz und Gleichheit Das Smalltalk–Protokoll definiert Methoden zur Äquivalenz und Gleichkeit. == stellt fest, ob beide Objekte dasselbe Objekt sind. = stellt fest, ob beide Objek-

te die gleiche Komponente repräsentieren [GR83, S.96]. Da Parameter in Smalltalk dynamisch typisiert sind, überprüft ein Test auf Objektgleichheit zunächst auf Typäquivalenz beider Objekte.

Java stellt Gleichheit durch Aufruf der Methoden `equal()` fest. Äquivalenz wird durch den in der Sprache definierten Operator `==` festgestellt. Da der formale Parameter der `equal()`–Methode in Java den Typ `Object` hat, muß auch in Java zunächst der dynamische Typ des Arguments überprüft werden, bevor die Gleichheit festgestellt werden kann.

Die Protokolle unterscheiden sich in Smalltalk und Java kaum und deren Implementierung in neuen Klassen ist sehr ähnlich. Smalltalk wie Java definieren die Methode `hash` zur Ermittlung eines Hash–Wertes. Bei Überschreiben des Gleichheitstests muß diese Methode ebenfalls überschrieben werden, um die Gleichung `a = b` $\Rightarrow$ `a hash = b hash` zu erfüllen.

Vergleich von Objekten bzw. Elementordnung Zum Vergleich von Objekten siehe das Idiom „Comparing Method“ (5.1.6).

4.3.2 Abhängigkeitsmechanismus

In Smalltalk kann sich ein Objekt bei jedem beliebigen Objekt registieren lassen, um über alle relevanten Änderungen informiert zu werden. Der Dependency–Mechanismus wurde in frühen Versionen von Smalltalk in der GUI dazu verwendet, die Anbindung der View an das Modell zu gestalten. Dieser Mechanismus wurde inzwischen jedoch vom Ereignismodell abgelöst, der im Abschnitt 4.3.3 behandelt wird.

In Java dagegen muß eine Klasse, die sich beobachten lassen will, von `Observable` erben. Ein Beobachter muß das `Observable`–Interface implementieren.

Dies ist zwar eine exakte Nachbildung des Abhängigkeitsmechanismus von Smalltalk, aber die Notwendigkeit, von `Observable` zu erben, kann die Migration schwierig machen. Nur wenn die zu beobachtende Klasse in Smalltalk von `Object` erbt, muß sie nicht refaktoriert werden. Ansonsten kann es nötig sein, daß eine Superklasse weiter oben in der Hierarchie von `Observable` erben muß. Die Verwendung des Abhängigkeitsmechanismus kann in Java eine Umstrukturierung erfordern, wenn die Klasse nicht von `Object` erbt. Jedoch wird der Abhängigkeitsmechanismus oft durch das Ereignismodell ersetzt. Diese Ersetzung kann auch bei der Migration geschehen, weil eine Umstrukturierung der Klassenhierarchie evtl. nicht zu vermeiden ist.

4.3.3 Ereignisbehandlung

Durch den Mechanismus der Ereignisbehandlung kann einem Objekt mitgeteilt werden, daß in einem anderen Objekt ein interessantes Ereignis eingetreten ist. Ein solches Ereignis kann beispielsweise eine Eingabe durch den Benutzer sein. In diesem Abschnitt behandeln wir die verschiedenen *Aufrufmechanismen*, die beiden *Abstraktionsebenen* des Ereignismodells, die Unterschiede *wie Ereignisse zusammengefaßt werden* und besprechen die *vorhandenen Klassen*.

Aufrufmechanismus

Eine Smalltalk–Applikation registriert bei der ereignisauslösenden Instanz ein Objekt (meist sich selbst) und den Methoden–Selektor der Behandlermethode. Bei Eintreten des Ereignisses führt diese Instanz einen dynamischen Methodenaufruf aus, um die Behandlermethode aufzurufen (siehe Abbildung 4.8). Die Behandlermethode erhält als Argument eine Instanz von `CwAnyCallbackData` oder einer Subklasse davon. Hierüber erfährt der Behandler von den spezifischen Daten des Ereignisses.

Eine Java–Anwendung registriert dagegen ein Objekt, das ein bestimmtes Interface implementieren muß. Das Ereignis ruft eine der Methoden auf, die über dieses Interface definiert sind (siehe

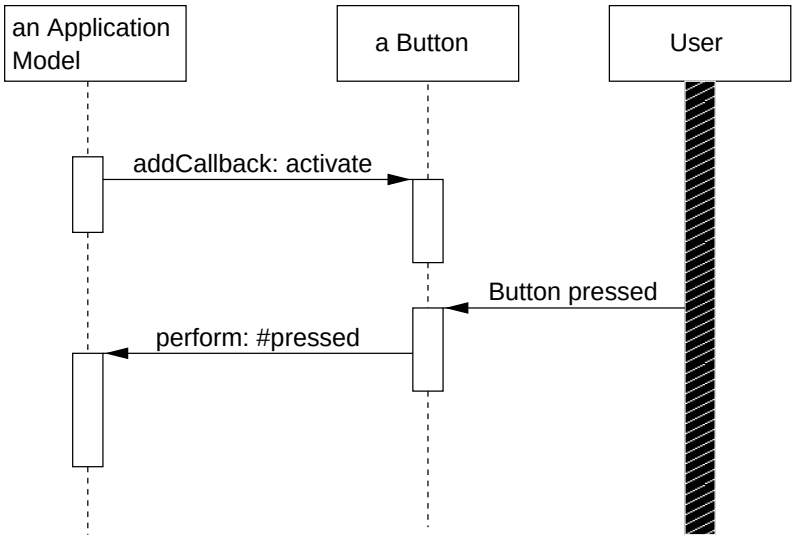


Abbildung 4.8: Callback in Smalltalk

Abbildung 4.9). Um diesem Objekt Zugriff auf die lokalen Variablen zu geben, muß es als Instanz einer innere Klasse realisiert sein (siehe auch 4.1.3). Eine Behandlermethode erhält eine Instanz von `EventObject` oder einer ihrer Unterklasse als Argument.

Smalltalk und Java verwenden verschiedene Mechanismen zum Aufruf des Ereignisbehandlers. In den meisten Fällen läßt sich jedoch die Verwendung des Smalltalk–Mechanismus leicht nach Java übertragen. Eine ereignisbehandelnde Methode kann in Java keinen beliebigen Namen haben. Eine Klasse, die als Ereignisbehandler eingesetzt werden soll, muß ein Listener–Interface implementieren. Dieses Interface definiert die Namen der Behandlermethoden. Bei der Migration ist folglich eine Umbenennung der entsprechenden Methoden nötig. Die Methoden zur Implementierung eines Listener–Interface werden oft in einer eigenen Klasse zusammengefaßt. Um dieser Klasse den Zugriff auf nicht–öffentliche Elemente des am Ereignis interessierten Objekts zu ermöglichen, muß diese Klasse als „innere Klasse“ definiert sein (siehe 4.1.3).

Abstraktionsebenen

Ereignisse werden in IBM Smalltalk in **Callbacks** und **Events** unterschieden. Callbacks behandeln logisch höhere Aktionen des Benutzers (wie das Aktivieren eines `CwPushButton`) während Events primitive Ereignisse der Benutzerschnittstelle behandeln (Bewegen des Mauszeigers, Mausknopfbenutzung, Tastendruck).

Das Ereignismodell von Java ist logisch gesehen mit Smalltalk–Callbacks vergleichbar.

> [...] the new event handling model is essentially a „callback“ model. [Fla97, S.144]

Die Funktionalität der Smalltalk–Events ist in Java ebenfalls durch das Ereignismodells abgedeckt. Java kennt beim Ereignismodell keine Trennung zwischen callbacks und events wie Smalltalk. Das *low–level event–handling model* von Java setzt noch tiefer an als Smalltalk–Events und verwendet einen anderen Aufrufmechanismus.

Die Aufteilung der Ereignisse in Smalltalk nach callbacks und events spiegelt sich in Java nicht wieder. Das Ereignismodell faßt in Java beide Abstraktionsebenen zusammen. Das low–level

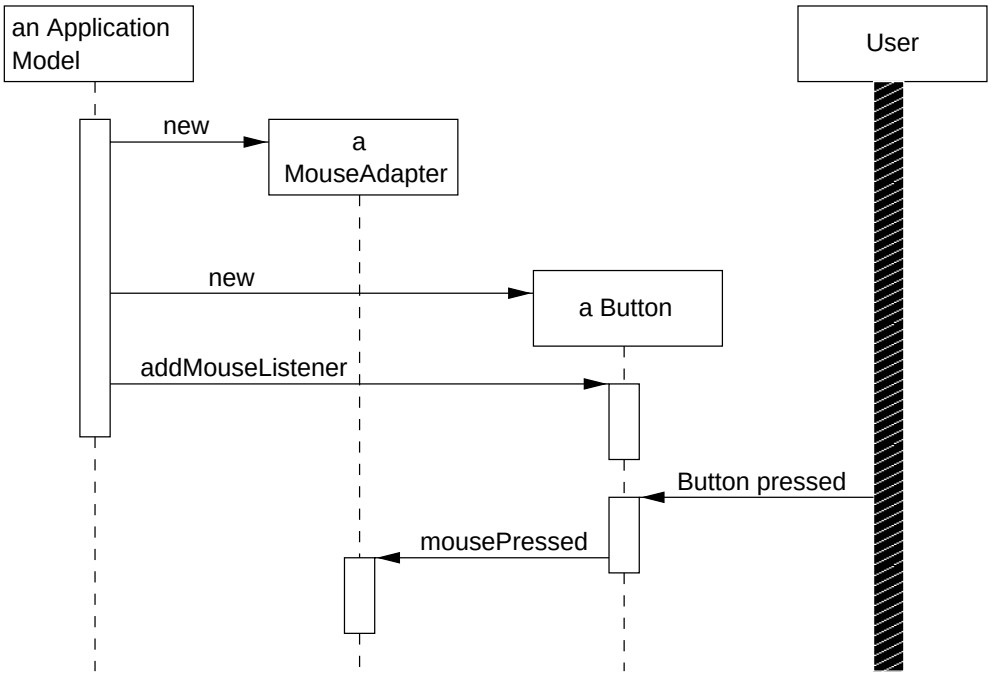


Abbildung 4.9: Event in Java

event–handling model wird für die Applikationsentwicklung nicht benötigt. Da sich der Aufrufmechanismus zwischen callbacks und events nicht wesentlich unterscheidet, hat diese Aufteilung in Smalltalk keinen Einfluß auf die Migration.

Granularität

In Smalltalk kann für jedes Ereignis ein eigener Behandler registriert werden. In Java werden mehrere Behandlermethoden dagegen durch ein Listener–Interface zusammengefaßt. Die Zusammenfassung mehrerer Behandlermethoden in einem Behandlerobjekt in Java wird schwierig, falls die Behandlermethoden in Smalltalk über mehrere Klassen verteilt sind. Insbesondere wenn eine Behandlermethode nicht über die externe Schnittstelle auf das registrierte Objekt zugreift, kann durch das Herauslösen der Behandlermethode eine Änderung der externen Schnittstelle der Klasse notwendig werden, um der Behandlermethode Zugriff auf benötigte Informationen zu geben.

Vorhandene Klassen

Tabelle B.12 im Anhang stellt die in Smalltalk vorhandenen Callbacks, Tabelle B.9 die Events den entsprechenden Konstrukten in Java gegenüber. Für die meisten Ereignisse definiert Java Entsprechungen. Die Smalltalk–Ereignisse, für die keine Entsprechung in Java existiert, sind Teil des GUI–Frameworks. Sie werden entweder in Java nicht benötigt (`cascading`, `popdown`, `popup` und `simple`) oder werden in Java durch einen anderen Mechanismus abgedeckt (`help`).

Typisierung

Die Ereignisbehandlung funktioniert in IBM Smalltalk und Java sehr ähnlich. Allerdings bedeutet deren Verwendung aufgrund des statischen Typsystems einen höheren Programmieraufwand. Für jedes Ereignis müssen drei bis vier Klassen und Java–Interfaces geschrieben werden: [Kra97]

- eine Klasse für das Ereignis,
- ein Interface für den Ereignis–Sender,
- ein Interface für den Ereignis–Empfänger und
- evtl. eine Klasse für den Multicast von Ereignissen.

In Smalltalk ist hierfür lediglich eine Klasse zu implementieren.

4.3.4 Ausnahmebehandlung

Fehlersituation werden durch Ausnahmen repräsentiert. Eine Ausnahme ist eine Situation, die vom erwarteten Fluß des Programms abweicht. Falls eine solche Ausnahme nicht behandelt wird, wird die Ausführung des Programms unterbrochen. Smalltalk unterscheidet zwischen *lokaler Ausnahmebehandlung* und *globaler Ausnahmebehandlung*.

Lokale Ausnahmebehandlung

Bei lokaler Ausnahmebehandlung wird in Smalltalk einem Methodenaufruf zusätzlich ein Codeblock mitgegeben, welcher für die Behandlung von Ausnahmesituationen beim Abarbeiten dieser Methode zuständig ist (Bsp: `remove:ifAbsent:`). Diese Art der Ausnahmebehandlung ist von Fall zu Fall spezifisch für die aufgerufene Methode [Ple97, S.96].

In Java ist lokale Ausnahmebehandlung nicht definiert. Einem Methodenaufruf kann kein Behandlerblock mitgegeben werden. Die aufrufende Methode muß in aller Regel den Rückgabewert der aufgerufenen Methode auswerten und dann selbst die Fehlerbehandlung durchführen.

Die Migration von lokaler Ausnahmebehandlung führt in aller Regel zu Umstrukturierung des Programmcodes. Eine Abfrage des Fehlercodes muß eingefügt werden, um den Code des Behandlerblockes im Rumpf der Abfrage lokal abarbeiten zu können.

Globale Ausnahmebehandlung

Bei globaler Ausnahmebehandlung registiert die Applikation für einen Block einen Ausnahmebehandler, bevor der Block ausgeführt wird. Dieser Abschnit behandelt das Registrieren eines Ausnahmebehandlers, die Propagierung einer Ausnahme über den Aufrufstapel und vergleicht die vorhandenen Ausnahmen miteinander.

Registrierung von Ausnahmebehandlern In Smalltalk registriert die Methode `when:do:` einen Ausnahmebehandler für einen Codeblock. Die Registrierung von mehreren Ausnahmebehandlern geschieht durch Wiederholung von `when:do:`. In Smalltalk existieren Ausnahme–Collections, mit denen ein Ausnahmebehandler für eine Gruppen an Ausnahmen registriert werden kann. Die Ausnahmebehandlung von Smalltalk kennt *completion blocks*, die am Ende eines Blockes auf jeden Fall aufgerufen werden, auch wenn innerhalb des Blockes eine Ausnahme auftrat.

Java registriert einen Ausnahmebehandler für einen Block mit den Schlüsselwörtern `try` und `catch`. Mehrere Ausnahmebehandler registriert Java durch eine Wiederholung der `catch`–Klausel. Ausnahme–Collections wie in Smalltalk existieren nicht. Der completion block steht in Java in der `finally`–Klausel.

Die Registrierung von Ausnahmebehandlern verläuft in Smalltalk und Java sehr ähnlich, weil sich in beiden Sprachen die Ausnahmebehandler auf einen Block beziehen. Ebenso lassen sich completion blocks in beiden Sprachen registrieren. Nachteil von Java ist, daß es keine Ausnahme–Collections definiert.

Propagierung von Ausnahmen Ausnahmen propagieren entlang der lexikalischen Blockstruktur und entlang des Aufrufstapel bis zum passenden Ausnahmebehandler (siehe Abbildung 4.10).

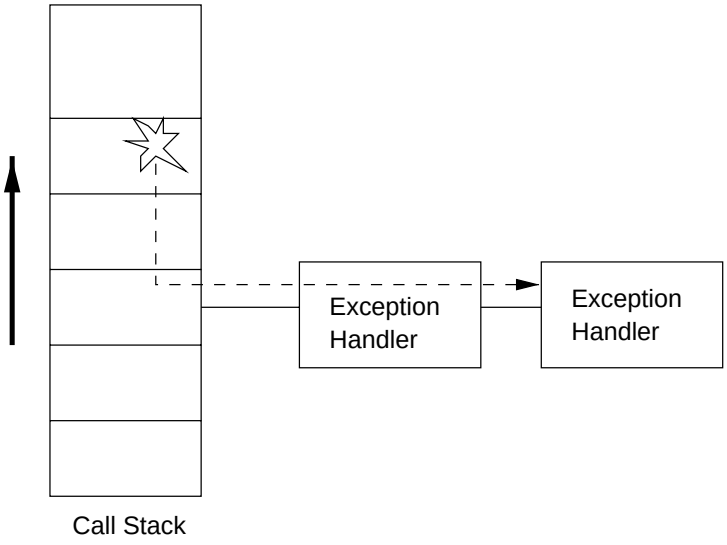


Abbildung 4.10: Globale Ausnahmebehandlung

Falls eine Ausnahme nicht vom auslösenden Block behandelt wird, sucht sie den nächst umschließenden Block. Falls in Smalltalk kein Behandler für diese Ausnahme registriert wurde, propagieren Ausnahmen bis zum nächsten Default–Behandler.

In Java existieren keine Default–Behandler. Eine Ausnahme propagiert in diesem Fall bis zur `main()`–Methode.

Da sich Smalltalk und Java bei der Propagierung von Ausnahmen sehr ähnlich sind, ist aus dieser Richtung kein Problem für die Migration zu erwarten. Die Installation eines Behandlers in der `main()`–Routine bildet die Smalltalk–Funktionalität ausreichend gut nach.

Vorhandene Ausnahmen Die Ausnahmeklassen sind in Java über die gesamte Klassenbibliothek verteilt während in Smalltalk die Ausnahmeinstanzen über die gesamte Applikation verteilt sind.

In Smalltalk bilden die Ausnahmen eine Instanzen–Hierarchie (siehe Abbildung 4.11). D.h. sämtliche Ausnahmen sind von der Instanz `ExAll` abgeleitet. Eine Applikation erzeugt neue Ausnahmen durch die Nachricht `newChild` an eine der bestehenden Instanzen. Der Variablenpool `SystemExceptions` speichert die vom System definierten Ausnahmeinstanzen [IBM97, S.28].

In Java bilden die Ausnahmen eine Hierarchie von Klassen (siehe Abbildung 4.11). Bei Auftreten einer Ausnahmesituation erzeugt das System oder die Applikation eine Instanz einer dieser Klassen. Sämtliche Ausnahmen sind Subklassen von `Exception` oder `Error`. Ausnahmen der `Error`–Hierarchie sollten nicht durch Ausnahmebehandler abgefangen werden, da sie kritische Fehlersituationen anzeigen, die den Abbruch der Applikation erfordern (vergleichbar mit `ExHalt`).

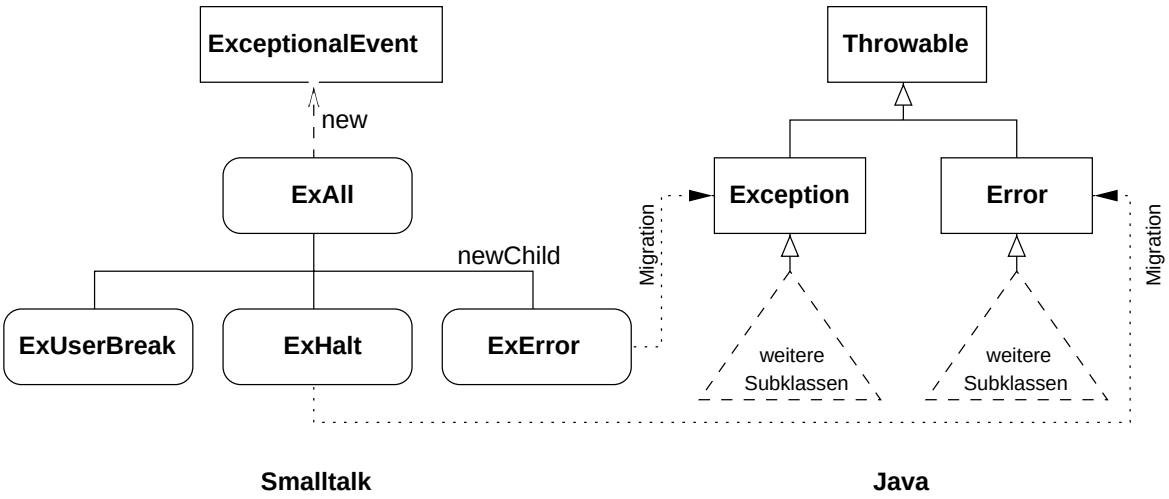


Abbildung 4.11: Ausnahme–Hierarchie in Smalltalk und Java

Instanzen von `Exception` zeigen Ausnahmesituationen an, die die Applikation behandeln kann (vergleichbar mit `ExError`).

Für die Migration ist entscheidend, wie die Sichtbarkeit von selbst definierten Ausnahmen gesteuert wird. In Smalltalk muß eine Applikation hierzu Variablenpools, globale Variblen, Klassenvariablen oder explizite Argumentübergabe verwenden. In Java dagegen sind Ausnahmen Teil der Klassenhierarchie, wodurch die Packages die Sichtbarkeit von Ausnahmen regeln. Das bedeutet bei der Migration ein Entfernen dieser Übergabemechanismen und ein direkter Verweis auf die Klassenhierarchie. Die Erzeugung neuer Ausnahmeinstanzen in Smalltalk durch `newChild` muß in Java durch das Anlegen einer neuen Klassen in der Throwable–Hierarchie dargestellt werden. Die Behandlung von `ExUserBreak` in Smalltalk kann durch die Migration ersatzlos gestrichen werden. Diese Ausnahme ist in Smalltalk deshalb notwendig, weil in Smalltalk das System und die Applikation sehr eng miteinander verbunden sind.

4.3.5 GUI–Framework

Dieser Abschnitt behandelt das GUI–Framework. Er beginnt mit einer Beschreibung, wie ein GUI–Framework verwendet wird, gibt einen Überblick über die GUI–Frameworks in IBM Smalltalk und Java und betrachtet anschließend die verschiedenen Teile des Frameworks. Das GUI–Framework ist selbst wieder unterteilt in Subsysteme. Diese werden in Abbildung 4.12 dargestellt.

Diese Betrachtung beschränkt sich auf „handgeschriebene“ Oberflächen. Für IBM Smalltalk gibt es einen integrierten GUI–Builder, der auf einem eigenen Komponentenmodell beruht. Für Java existieren eine Vielzahl an Einzellösungen, die jedoch alle sehr unterschiedlich sind. Eine solche Betrachtung würde den Rahmen dieser Arbeit sprengen.

Verwenden des GUI–Framework

Das GUI–Framework ist für die Architektur einer Applikation mit graphischer Benutzerschnittstelle von zentraler Bedeutung. Die Applikation registriert Fenster beim System, welche die Schnittstelle zum Benutzer darstellen und stellt die Applikationslogik zur Verfügungen. An dieser Stelle tritt die Inversion des Kontrollflusses auf: Die Applikation gibt die Kontrolle des Programmflusses zugunsten

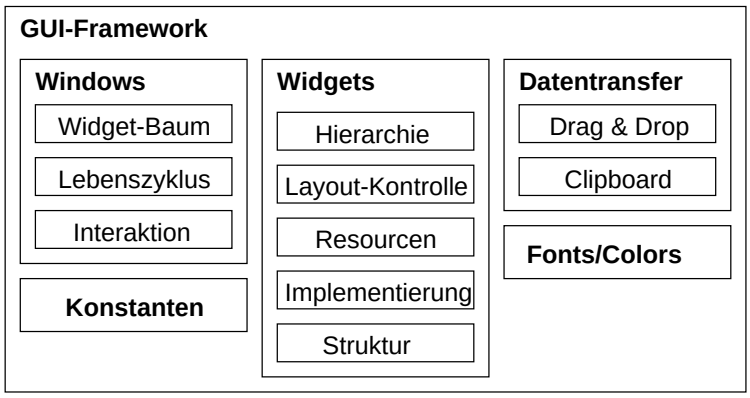


Abbildung 4.12: Subsysteme des GUI–Frameworks

der GUI auf. Die GUI informiert die Applikation in Form von Ereignissen über die Aktivitäten des Benutzers um auf deren Funktionalität zurückzugreifen. Erst wenn der Benutzer die Aktivität der GUI beendet, erhält die Applikation die volle Kontrolle über den Programmfluß zurück.

Das Verwenden eines GUI–Framework besteht also zum einen aus einem Verwenden der vorgegebenen Elemente (Widgets) zum Aufbau einer graphischen Oberfläche. Zum anderen besteht es aus einem Ausfüllen der „hot spots“ des Frameworks [Pre95]. Dieses Ausfüllen bedeutet, daß die spezifische Applikation das Framework als das „Skelett einer Applikation“ betrachtet und nur noch den applikationsspezifischen Teil zur Verfügung stellen muß [Joh97]. Sie zieht damit Nutzen aus dem wiederverwendbaren Design des Frameworks, welches in Form von wiederverwendbare Klassen verfügbar ist.

IBM Smalltalk

Das GUI–Framework von IBM Smalltalk teilt sich in mehrere Subsysteme auf. Der wichtigste Teil ist das *Common Widgets Subsystem*. Es stellt die grundlegenden Mechanismen zum GUI–Aufbau und eine grundlegende Menge an Widgets zur Verfügung. Das *Extended Widget Subsystem* ist eine Erweiterung, welche neue Widgets definiert. Um Daten zwischen verschiedenen Applikationen transferieren zu können, enthält das Framework das *Drag&Drop Subsystem*.

Java

Das GUI–Framework basiert auf dem *Abstract Windowing Toolkit* (AWT), welches das zeitlich erste GUI–Framework von Java war und welches die grundlegenden Elemente zum Aufbau einer GUI enthält. Swing ist der Nachfolger von AWT und zeichnet sich dadurch aus, daß es über *pluggable Look&Feel* verfügt und komplett in Java geschrieben ist. Dadurch ist es auf allen Rechnern, die über eine JVM verfügen lauffähig. AWT benötigte dagegen auf jeder Rechnerplattform eine spezifische Implementierung. Im Rahmen dieser Untersuchung konzentrieren wir uns auf Swing.

Widget–Baum

Auf beiden Plattformen besteht die graphische Oberfläche aus einer Aggregation von GUI–Elementen (widget tree). Jedes Widget außer dem obersten hat ein Eltern–Widget. Hierdurch wird eine Eltern–Kind–Relation definiert. Abbildung 4.13 zeigt einen typischen Widget-Baum für IBM Smalltalk [IBM97, S.134].

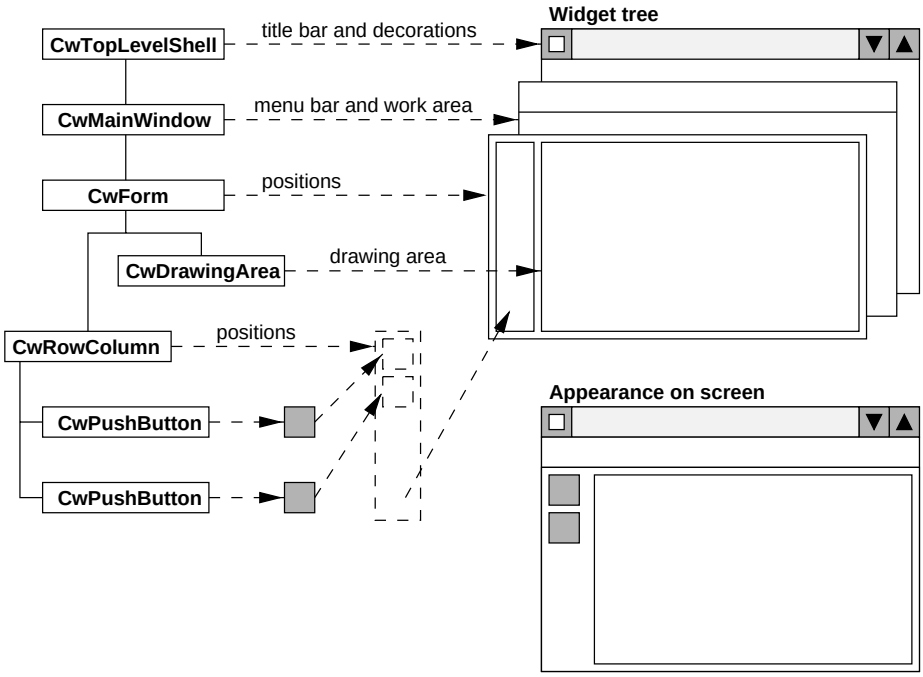


Abbildung 4.13: Widget–Baum in IBM Smalltalk

In IBM Smalltalk ist ein Shell–Widget die Basis für jedes Applikations–Fenster. Es dient der Interaktion mit dem Fenstermanager und ist die Wurzel des Widget–Baums. In der Darstellung auf dem Bildschirm liegen die verschiedenen Widgets aufeinander. Je weiter oben ein Widget im Widget–Baum angesiedelt ist, desto mehr seiner Fläche wird durch Kind–Widgets überdeckt. Im Beispiel stellt das Main–window die Menüleiste und den Arbeitsbereich der Applikation bereit. Das Form–Widget ist für das Layout seiner zugeordneten Widgets zuständig. Die Buttons sind primitive Widgets, mit denen der Benutzer Aktionen anstößt.

In Java sind die *Top–Level–Container* wie `JFrame`, `JApplet`, `JWindow` und `JDialog` für den Anschluß an die Infrastruktur wie die Menüleisten oder den Fenstermanager zuständig. Abbildung 4.14 zeigt einen typischen Widget–Baum.

Zu Top–Level–Containern können nicht direkt Kind–Widgets hinzugefügt werden, obwohl sie von `Container` abgeleitet sind. Eine Applikation muß zunächst das content pane des Top–Level–Containers erfragen, welches die Kind–Widgets speichern kann. Das direkte Kind eines Top–Level–Containers ist ein `JRootPane`. Dieses enthält ein `JLayeredPane` welches die Menüleiste und den content pane verwaltet. Zusätzlich verwaltet `JRootPane` ein glass pane. Das glass pane liegt auf dem Bildschirm vor sämtlichen anderen Widgets des Fensters und dient dazu, Mausbewegungen abzufangen oder Drag&Drop–Operationen zu initiieren. Um das Layout der Kind–Widgets zu steuern, registriert eine Applikation einen `LayoutManager` beim content pane. Insgesamt ist der Aufbau eines Fensters in Java um einiges komplizierter als in Smalltalk.

Für die Migration ist wichtig, daß Kind–Widgets nicht direkt dem Top–Level–Container zugeordnet werden, sondern dessen content pane. Die zum Layout von Kind–Widgets in Smalltalk benötigten Form–Widgets entfallen, stattdessen muß beim jeweiligen content pane ein Layout–Manager registriert werden.

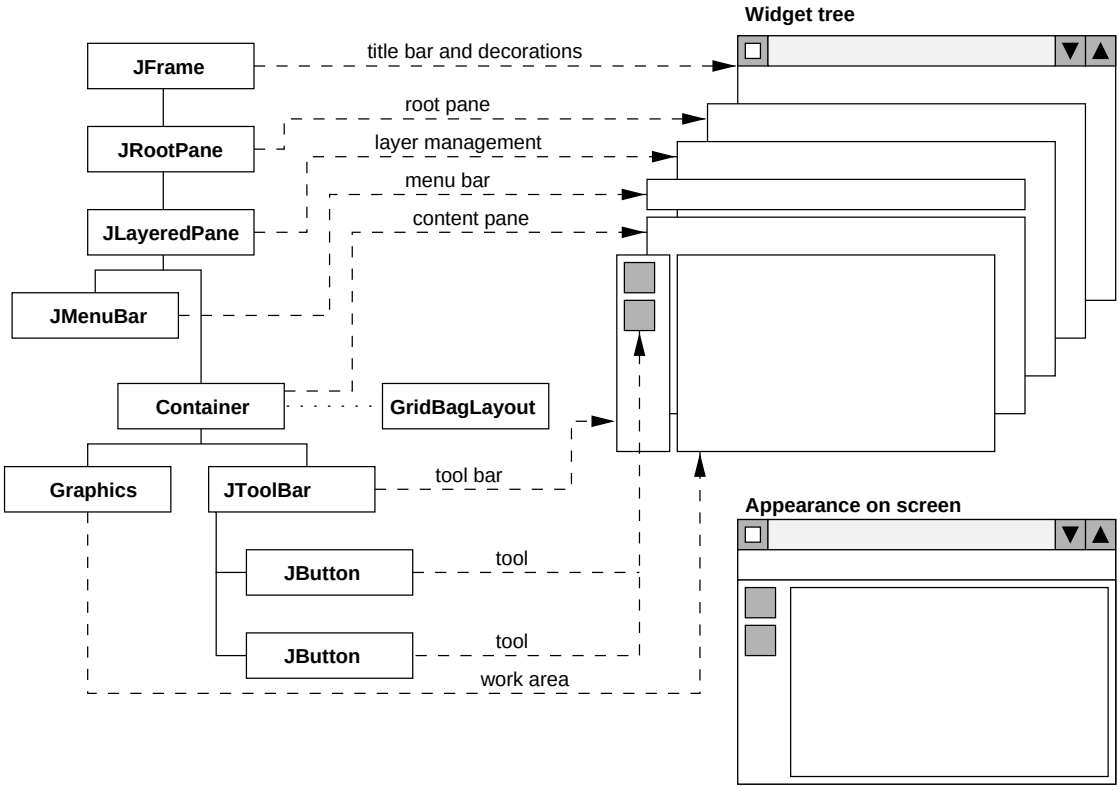


Abbildung 4.14: Widget–Tree in Java

Lebenszyklus

Ein Widget muß mehrere Schritte durchlaufen, bis es auf dem Bildschirm sichtbar wird. Die verschiedenen Schritte unterscheiden sich zwischen Java und Smalltalk.

Die Widget–Erzeugung legt in IBM Smalltalk die Eltern–Kind–Relation fest, indem ein Kind über *creation–convenience*–Methoden durch das Eltern–Widget erzeugt wird. Das **Managing** bezieht das Widget in die Berechnung der graphischen Anordnung im Eltern–Widget mit ein. Der Platz für seine Darstellung wird bereitgehalten. Das **Mapping** gibt an, daß ein Widget auf dem Bildschirm dargestellt werden soll, sobald es realisiert ist. Meist wird ein Widget durch den Managing–Schritt gemappt. **Realisieren** eines Widget ist das Sichtbarmachen auf dem Bildschirm. Wenn ein Widget sichtbar gemacht wird, werden rekursiv alle Kinder ebenfalls sichtbar gemacht. Ein Widget wird entweder durch den Benutzer oder unter Programmkontrolle durch `destroyWidget` freigegeben. Alle Kinder im Widget–Baum eines freizugebenden Widgets werden ebenfalls freigegeben.

In Java legt die Widget–Erzeugung die Eltern–Kind–Beziehung noch nicht fest. Eine Applikation muß eine neue Komponente explizit durch `add()` einem Eltern–Widget zuordnen. *Creation–convenience*–Methoden wie in Smalltalk existieren kaum (Ausnahme: `createDialog` in `JOptionPane`). Die Aufteilung des Lebenszyklusses eines Widgets erfordert weniger Schritte als in Smalltalk. Nach dem Hinzufügen zu einem Eltern–Widget und dessen Sichtbarmachen wird ein Widget sofort auf dem Bildschirm dargestellt. Die Größe eines Kind–Widgets kann anhand der Größe des enthaltenden Fensters durch den Layout–Manager bestimmt werden. Durch die Methode `pack()`

erhalten die Widgets die Chance, ihre bevorzugte Größe mitzuteilen.

Für die Migration ist entscheidend, daß Widgets in Smalltalk durch creation–convenience–Methoden erstellt werden. Die Eltern–Kind–Beziehung ist direkt zu sehen und muß in Java durch `add()` als zusätzlicher Schritt erfolgen. Da die Schritte von der Erzeugung bis zum Sichtbarwerden eines Widgets in Smalltalk normalerweise von der GUI übernommen werden, ist in diesen Fällen die Migration unkritisch. Falls die Applikation allerdings in diesen Prozeß eingreift, stellt sich die Migration nach Java schwierig dar, weil Java nur einen Schritt kennt.

Interaktion mit Applikation

Eine graphische Oberfläche muß die Möglichkeit haben, die Applikation über Benutzerereignisse zu unterrichten. Hierfür existieren mehrere Möglichkeiten.

IBM Smalltalk verwendet zur Interaktion mit der Applikationslogik events und callbacks (siehe 4.3.3). Events zeigen die primitiven Operationen der Oberfläche an wie ein Mausklick oder das Bewegen des Mauszeigers. Callbacks fassen evtl. mehere primitive Aktionen zusammen und zeigen das Kommando des Benutzers an wie das Aktivieren eines Buttons. Die events sind in allen Widgets identisch. Die Applikation registriert sich für ein event oder einen callback mit einem Selektor, welcher auf eine Methode im registrierten Objekt verweist. Diese Methode wird bei Auftreten des Ereignisses aufgerufen.

In Java verwendet die GUI das Ereignismodell zur Interaktion mit der Applikationslogik. Das GUI–Framework faßt mehrere Ereignisse in Listener–Interfaces zusammen. Jedes Widget zeigt an, für welche Listener es Ereignisse erzeugen kann. Die Applikation muß dieses Interface implementieren, um sich für ein Ereignis registrieren zu lassen. Bei Auftreten eines Ereignisses, wird die über das Listener–Interface definierte Methode aufgerufen.

Bei der Migration muß für jeden callback und für jeden event des passende Listener–Interface gefunden werden. Tabelle B.12 enthält diese Abbildung für callbacks, Tabelle B.9 für events (siehe Anhang B).

Die Migration kann einige Umstrukturierungen der Applikationsklassen zur Folge haben. In Smalltalk ist der Ereignisbehandler eines spezifischen Ereignisses eine beliebige Methode. In Java muß dieser Behandler über das Listener–Interface definiert sein, welches mehere Ereignisse zusammenfaßt. D.h. evtl. müssen zwei ereignisbehandelnde Methoden in zwei verschiedenen Smalltalk–Klassen in einer Java–Klasse zusammengefaßt werden. Falls eine Klassenmethode die Ereignisbehandlung in Smalltalk übernimmt, läßt sich das in Java nicht darstellen. Durch die Verwendung von inneren Klassen wird es schwieriger, einen Ereignisbehandler „von Hand“ aufzurufen, weil zunächst die Instanz der inneren Klasse referenziert werden muß. Außerdem sind die Zuständigkeiten der definierten callbacks und Listener nicht vollständig identisch.

Insgesamt kann also eine Migration einige Änderungen der Applikationsarchitektur notwendig machen.

Struktur eines Widgets

Die Strukturierung eines Widget kann von Plattform zu Plattform unterschiedlich sein. Die Struktur muß drei Faktoren berücksichtigen: wie werden die Daten des Widget gespeichert, wie werden sie auf der Oberfläche präsentiert und wie werden Benutzereingaben verarbeitet. Ein oft verwendetes Architekturmuster in diesem Bereich ist MVC (siehe 5.3.3).

Die innere Struktur eines Widgets in Smalltalk ist nicht sonderlich kompliziert. Die Widget–Klasse selbst ist im wesentlichen ein Proxy für das native Betriebssystemwidget (`osWidget`). Eine Strukturierung der Widgets nach dem MVC–Konzept wie in VisualWorks ist in IBM Smalltalk folglich nicht zu finden. Das Look&Feel eines Widget ist somit nicht änderbar.

Java teilt ein Widget in Modell und Delegate auf. Das Modell enthält den Wert des Widget, während das Delegate für die Darstellung auf dem Bildschirm einerseits und die Verarbeitung von Benutzeraktionen andererseits zuständig ist. Die Standard–Widgets wie `JButton`, `JTextField`, etc. sind eine Kombination aus einem Delegate und einem entsprechenden Modell, um als vollständiges GUI–Element arbeiten zu können. Abbildung 4.15 zeigt die Aufteilung eines `JButton`.

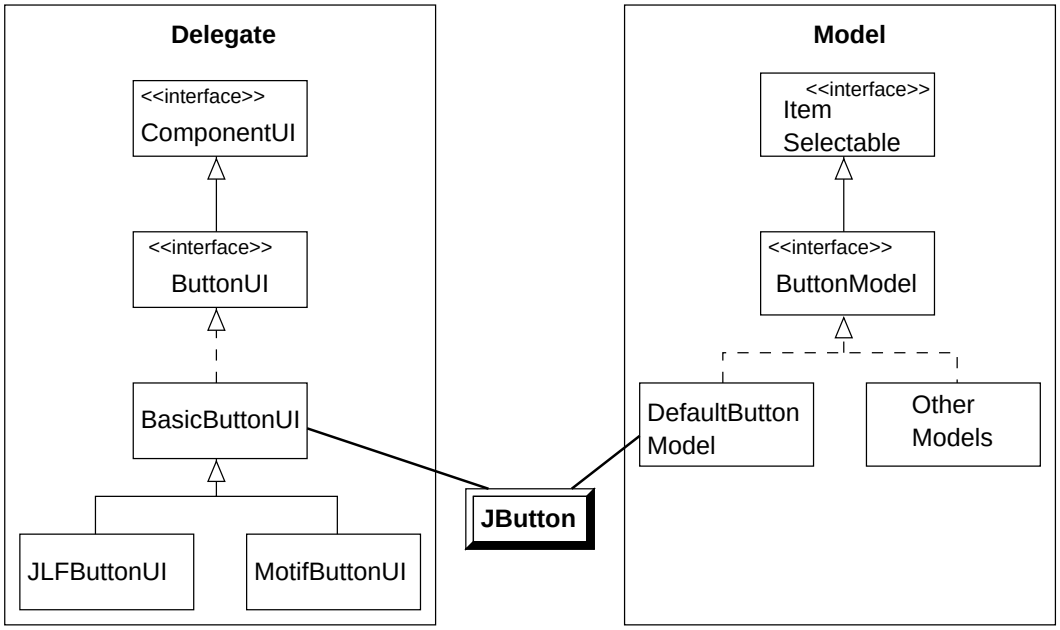


Abbildung 4.15: Struktur des `JButton`

Diese Struktur bietet folgende Vorteile: Das Look&Feel der Widgets einer Applikation kann zur Laufzeit umgeschaltet werden und die dargestellten Interfaces stellen ein Framework zur Erstellung eigener portabler Widgets dar.

Für die Migration ist diese Aufteilung ohne Bedeutung. In Smalltalk kann durch die Verwendung der nativen Betriebssystem–GUI–Elemente das Delegate nicht verändert werden. Die Aufteilung in Modell und Delegate wird von Applikationen nicht genutzt. Aus Applikationssicht speichert das Widget seinen Wert selbst, bietet aber Methoden an, die den Wert abfragen, bzw. setzen können (siehe Abschnitt Ressourcen und Funktionen). Aus Java–Sicht erfüllen die Standard–Implementierungen der Widgets genau diese Funktion.

Klassen–Hierarchie

Das GUI–Framework ordnet die Widgets in einer Klassenhierarchie an. Diese Hierarchie bildet die statischen Beziehung zwischen den Widget–Klassen. Der Widget–Baum stellt die dynamischen Beziehungen der Instanzen der Widget–Klassen dar und ist von dieser Klassenhierarchie unabhängig.

Smalltalk unterscheidet folgende Widget–Kategorien:

Primitive Widgets sind die einfachsten Elemente einer graphischen Benutzeroberfläche wie `CwLabel` oder `CwList`. Sie können selbst keine Kinder mehr haben.

Zusammengesetzte Widgets können andere Widgets enthalten. Beispiele sind `CwFrame` und `CwMainWindow`. Sie bieten verschieden Möglichkeiten an, wie deren Kinder angeordnet werden sollen.

Shell–Widgets haben genau ein Kind. Sie bilden die Wurzel des Widget–Baumes und sind für die Kommunikation mit dem Fenster–Manager zuständig. Beispiele sind `CwTopLevelShell` und `CwDialogShell`.

Darüber hinaus existiert ein Framework zum Erstellen eigener Widgets. Das Extended Widget Subsystem ist eine Sammlung von Widgets, die mit Hilfe dieses Frameworks implementiert sind. Prompter sind Dialogboxen des zugrundeliegenden Betriebssystems, welche aus der sonstigen Widget–Hierarchie herausfallen. Sie benötigen kein eigenes Shell–Widget und erfragen beim Benutzer Information, die die Applikation zur Weiterarbeit benötigt. Dialoge sind aus gewöhnlichen Widgets zusammengesetzt und für eine bestimmte Interaktion mit dem Benutzer vorkonfiguriert. Sie sind jeweils Kinder einer `CwDialogShell`, um sie unabhängig vom Hauptfenster zu machen.

Java definiert zwei Arten an GUI–Komponenten. *Top–Level–Container* wie `JFrame`, `JApplet`, `JWindow` und `JDialog` bieten zum einen den Bereich, in dem sich die Elemente der Widget–Hierarchie zeichnen können. Zum anderen sorgen sie für den Anschluß an die Infrastruktur wie die Menüleisten oder den Fenstermanager. Jedes Applikationsfenster sollte von `JFrame` erben und jedes Applet sollte von `JApplet` erben. Die GUI–Elemente wie `JButton`, `JPanel`, etc. sind Widget–Implementierungen des Lightweight UI Framework. Da jedes Widget von `Container` erbt, kann es eigene Komponenten enthalten.

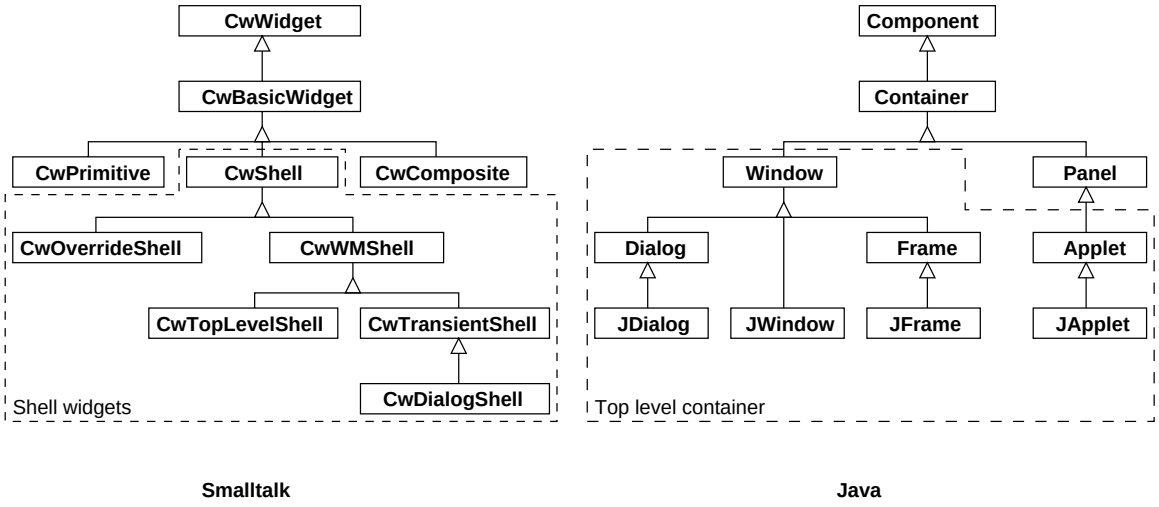


Abbildung 4.16: Shell–Widget–Hierarchien

Die Migration hat damit zu kämpfen, daß die in IBM Smalltalk und in Java angebotenen Widget–Klassen zum Teil nicht die identische Funktionalität bieten. Layout–Kontrolle der Kind–Widgets geschieht nicht über eine eigene Widget–Klasse, sondern über Layout–Manager.

Abbildung 4.16 stellt die Hierarchien der Shell–Widgets von Smalltalk den Top–Level–Containern von Java einander gegenüber. Anhang B enthält für jede der in Smalltalk beschriebenen Widget–Kategorien und Dialoge eine Tabelle, die ein Smalltalk–Widget auf das am ehesten passende Java–Widget abbildet.

Java kennt die in Smalltalk vorhandenen Prompter nicht. Sie müssen mit gewöhnlichen Mitteln emuliert werden. Tabelle B.10 bildet die vorhandenen Prompter–Klassen auf mögliche Java–Widgets ab. Prompter werden nicht über creation convenience Methoden erstellt, weshalb deren Erzeugung sehr ähnlich zu der in Java ist.

Aus Migrationssicht sind die Elemente des *Extended Widget–Subsystem* genau gleich zu behandeln wie gewöhnliche Widgets. Tabelle B.6 (siehe Anhang) enthält eine Abbildung der vorhandenen Elemente auf Java–Widgets. Eine Schwierigkeit besteht darin, daß die Elemente in Swing nicht automatisch verschiebbar sind. Um das zu erreichen, muß jeweils ein `JScrollPane` dazwischengeschaltet werden. Dies ist in Smalltalk nicht nötig, weil Smalltalk automatisch die Notwendigkeit erkennt, ein Widget verschiebbar zu gestalten. Dies ist für die Migration insofern schwierig, weil erst herausgefunden werden muß, welche GUI–Elemente verschiebbar gestaltet werden müssen.

Insgesamt betrachtet kann die Smalltalk–Funktionalität nahezu vollständig in Java dargestellt werden. In der Implementierung existieren jedoch einige Detailunterschiede.

Layout–Kontrolle

Die Layout–Kontrolle dient dazu, Widgets so anzuordnen, daß keine festen Positionen vergeben werden müssen. Dieser Mechanismus ermöglicht zum einen die Skalierbarkeit eines Fensters und zum anderen eine Darstellung der Widgets, die unabhängig von zugrundeliegenden Betriebssystem ist (Die Pixelgrößen der GUI–Elemente variieren von Betriebssystem zu Betriebssystem).

In IBM Smalltalk dienen die Klassen `CwForm` und `CwRowColumn` der Layout–Kontrolle. Im ersten Fall gibt eine Applikation die Positionen der Widgets relativ zueinander an, indem Nachbarschaftsbeziehungen zueinander spezifiziert werden. Dies ist in IBM Smalltalk der allgemeinste Mechanismus, um Widgets anzuordnen. Im zweiten Fall regelt das Layout–Widget die Positionierung der enthaltenen Kind–Widgets, die in Spalten und Zeilen angeordnet werden. Hiermit können beispielsweise Listen oder Werkzeugleisten gestaltet werden.

In Java ist die Layout–Kontrolle sehr viel flexibler. Jeder Container besitzt einen `LayoutManager` der für die Positionierung der enthaltenen Elemente zuständig ist — ebenfalls unabhängig vom Betriebssystem oder der Fenstergröße. Ein großer Vorteil davon ist, daß sich die Applikation nicht schon beim Aufbau des Widget–Baums auf eine bestimmtes Layout festlegen muß — wie im Falle von Smalltalk. Das Layout der Elemente eines Container kann dynamisch an die jeweiligen Bedürfnisse angepaßt werden. Java bietet über dieses Interface die Möglichkeit, eigene Layoutmanager zu schreiben, bietet aber schon eine große Zahl von vorgefertigten Layoutmanagern an.

In Bezug auf Migration sind vor allem `GridLayout` und `GridBagLayout` von Interesse. Mit ihnen kann das Verhalten von `CwRowColumn` bzw. `CwForm` nachempfunden werden. Die Positionierungsangaben der Elemente in `CwForm` erfährt ein `GridBagLayout` durch `GridBagConstraints`. Diese können ebenso wie in Smalltalk für jede Komponente einzeln angegeben werden. Das Verhalten der Layout–Kontrolle kann also in Java nachgebildet werden.

Die Migration der Layout–Kontrolle führt zu einem anderen Aufbau des Widget–Baums. Sämtlicher Code, der im Zusammenhang mit der Layoutkontrolle steht, muß neu geschrieben werden. Eine Migration dieses Teils ist aufgrund der unterschiedlichen Mechanismen aufwendig.

Ressourcen und Funktionen

Ressourcen definieren das Verhalten und Aussehen eines Widgets. Funktionen veranlassen ein Widget zu einer bestimmten Aktion.

IBM Smalltalk realisiert Ressourcen und Funktionen als gewöhnliche Methoden in der Widget–Instanz. Ressourcen steuern beispielsweise die Geometrie eines Widget oder enthalten den Wert eines Textfeldes. Funktionen stoßen komplexere Aktionen eines Widget an wie beispielsweise das *managing* eines Kind–Widgets. Sämtliche Methoden eines Widgets, die nicht Getter– oder Setter–Methoden für Ressourcen sind, sind Funktionen.

Java macht keine Unterscheidung zwischen Ressourcen und Methoden. Widgets definieren eine API, über die mit Hilfe von Methoden das Verhalten und Aussehen gesteuert werden kann.

Für die Migration spielt die Unterscheidung der API eines Widgets in Ressourcen und Funktionen zunächst keine Rolle, da beide von der Applikation mit Hilfe von Methoden angesprochen werden. Hingegen auf Plattformen, die das X Window System verwenden, lassen sich die Widget–Ressourcen über die bekannten Ressource–Dateien des X Window System setzen. Eine solche Konfiguration muß in Java explizit ausprogrammiert werden.

Implementierung

Eine Sprachplattform hat prinzipell zwei Möglichkeiten, die Darstellung ihrer Widgets zu realisieren.

Die erste Möglichkeit verwendet die GUI–Elemente des jeweiligen Betriebssystems und legt darüber eine eigene betriebssystemunabhängige Schicht. Für jedes Widget des Framework existiert also ein Peer–Widget, welches den Zugriff auf das Widget der Plattform regelt. Der Vorteil dieses Ansatzes ist, daß eine Applikation weitgehend das Look&Feel des jeweiligen Betriebssystems annimmt. Der Nachteil ist, daß über verschiedene Betriebssysteme hinweg lediglich der kleinste gemeinsame Nenner an Widget–Elementen verwendet werden kann — außer das GUI–Framework emuliert nicht vorhandene Widgets selbst. IBM Smalltalk verfolgt diesen Ansatz [IBM97, S.1].

Die zweite Möglichkeit ist der Aufbau eines komplett unabhängigen Widget–Systems, welches lediglich die fundamentalste Fenster–Funktionalität des zugrundeliegenden Betriebssystems verwendet. Sämtliche Widgets müssen selbst gezeichnet werden, sind aber erheblich schlanker, weil nicht für jedes Framework–Widget ein Peer–Widget und ein Widget des Betriebssystems mitverwaltet werden muß. Der Nachteil ist, daß eine Applikation nicht unbedingt dem Look&Feel des zugrundeliegenden Betriebssystems entspricht, wodurch sie evtl. aus der Menge der sonstigen Anwendungen herausfällt. Der große Vorteil ist jedoch die Plattformunabhängigkeit. Eine Applikation die ein solches GUI–Framework als Basis verwendet, ist über sämtliche Betriebssysteme hinweg durchgängig portabel. Sie ist nicht mehr abhängig vom kleinsten gemeinsamen Nenner der betrachteten Betriebssysteme. Das *Lightweight UI Framework* in Java realisiert genau diese zweite Möglichkeit.

Für die Migration spielen diese Überlegungen keine Rolle. Eine Applikation verwendet die von der Sprachplattform bereitgestellten GUI–Elemente, ohne sich Gedanken über deren Realisierung machen zu müssen.

Konstanten

Konstanten werden in den GUI–Frameworks von Java und Smalltalk zur Konfiguration von Ressourcen eingesetzt.

In Smalltalk sind die Konstanten des GUI–Frameworks in den beiden Variablenpools `CwConstants` und `EwConstants` zusammengefaßt. Eine Applikation, die Konstanten aus diesen Pools benötigt, muß diese Pools folglich in der Klassendefinition angeben.

Java verwendet zwei Mechanismen, um den Zugriff auf Konstanten zu ermöglichen. `WindowConstants`, `ScrollPaneConstants` und `SwingConstants` fassen mehrere Konstanten in einem Interface zusammen. Manche Klassen definieren ihre eigenen Konstanten (Bsp: `GridBagLayout`) als Klassenvariablen.

Da die verwendeten Konstaten in Smalltalk und Java sehr unterschiedlich sind, ist bei der Migration von Fall zu Fall zu entscheiden, welche Konstanten in Java benötigt werden. Die größten Änderungen sind bei der Verwendungvon Layout–Managern als Ersatz für Form–Widgets zu erwarten.

Datentransfer

Beide Sprachplattformen bieten Mechanismen, damit verschiedene Applikationen innerhalb des selben Rechners miteinander Daten austauschen können.

Clipboard Die Clipboard–Variante nutzt einen Betriebssystem–Mechanismus, welcher kurzfristig Daten speichern kann, die später wieder gelesen werden.

Die Arbeit mit dem Clipboard ist in Smalltalk an ein beliebiges Window gebunden. Eine Applikation beginnt eine Transaktion, schreibt die Daten samt Formatangabe in das Clipboard und beendet die Transaktion. Zum Lesen fragt die Applikation erst auf vorhandene Daten im Clipboard ab und liest sie gemäß Formatangabe ein. Smalltalk unterstützt die plattformunabhängigen Datenformate `STRING` und `PIXMAP`. Zusätzlich kann eine Applikation plattform–spezifische bzw. proprietäre Formate registrieren.

Eine Java–Applikation holt sich das System–Clipboard durch das `Toolkit`–Objekt, das mit einem beliebigen Fenster verbunden ist. Zur Kommunikation mit dem Clipboard erzeugt die Applikation ein `Transferable`–Objekt, welches die zu übertragenden Daten repräsentiert. Dieses Objekt wird dann in das Clipboard geschrieben. Beim Lesen erzeugt das Clipboard ein `Transferable`–Objekt, welches die Applikation auslesen kann. Ein `DataFlavor` besteht in Java aus einem Namen und einer Typspezifikation die ein Java–Objekt oder einen MIME–Typ darstellt. Applikationsübergreifen kann Java bislang aber lediglich die Datenflavors `stringFlavor` und `textFlavor` verarbeiten [Fla97, S.163], jedoch keine PixMaps.

Im Ansatz sind die Java–Datenformate vielfältiger als in Smalltalk, deren Implementierung bereitet aber noch Probleme. Ansonsten sind die Clipboard–Mechanismen in Smalltalk und Java sehr ähnlich.

Drag & Drop Der Datentransfer durch Drag&Drop erfolgt darüber, daß der Benutzer Icons von einem Applikationsfenster in das andere schiebt. Die beteiligten Applikationen tauschen die Daten direkt aus, ohne den Umweg über das Clipboard zu nehmen. Allerdings bedeutet diese Form des Datentransfers im Vergleich zum Clipboard–Mechanismus mehr Aufwand bei der Gestaltung der GUI–Ansteuerung. Der Beginn eines Drag&Drop–Vorganges muß erkannt werden. Ebenso muß die Applikation dem Benutzer graphisch mitteilen, wo das Icon platziert werden darf.

Smalltalk bietet zum einen das Drag&Drop–Framework des Common Widget Subsystem. Zum anderen enthält es ein komplettes Drag&Drop–Subsystem, welches unabhängig von erstgenanntem ist.

Java hat die Drag&Drop–Funktionalität zwar im bisherigen Framework–Design vorgesehen und es existiert eine Draft–Version der API, aber zum Zeitpunkt der Erstellung dieser Arbeit existiert noch keine vollständige Implementierung.

Daher erübrigt sich zu diesem Zeitpunkt die Frage nach einer Migration. Auf Drag&Drop muß in Java zunächst verzichtet werden.

Farben und Zeichensätze

Die Auswahl von Farben und Zeichensätzen ist zwar für eine Applikation sehr wichtig, hat aber keinen Einfluß auf ihre Architektur. Daher sind diese beiden Aspekte des GUI–Framework kein Untersuchungsgegenstand dieser Arbeit.

4.3.6 Zusammenfassung

Der Frameworkteil weist die größten Unterschiede zwischen Smalltalk und Java auf. Im einzelnen werden die Unterschiede an folgenden Punkten deutlich:

- Die Ereignisbehandlung verwendet in Java Instanzen innerer Klassen anstatt callbacks. Dieser Mechanismus macht die Ereignisbehandlung komplizierter und kann zu einigen Umstrukturierungen führen. Aber prinzipiell steht der Nachbildung der Funktionalität in Java nichts im Weg.
- Eine Klasse die abhängige Objekte über Änderungen benachrichtigen soll, muß von `Observable` erben. Daher ist zur Verwendung des Abhängigkeitsmechanismus evtl. eine Umstrukturierung der Klassenhierarchie notwendig.
- Die Ausnahmebehandlung weist einige Unterschiede auf. Die Mechanismen sind aber doch so nahe verwandt, daß einer Migration nichts im Weg steht.
- Das GUI–Framework weist viele Gemeinsamkeiten auf. Insbesondere die graphischen Darstellungsmöglichkeiten durch Widgets sind in Bezug auf Migration sehr gut miteinander zu vergleichen. Probleme bereitet zum einen die Layout–Kontrolle. Diese ist bei der Migration mit einer Umstrukturierung des Widget–Baums verbunden. Außerdem ist die GUI–Interaktion mit der Applikation stark zu überarbeiten.

5

Migration von Mustern

Unter der Architektur einer Applikation verstehen wir im Rahmen dieser Untersuchung die Kombination von Mustern auf den verschiedenen Abstraktionsebenen. Jedes Muster trägt als „Mikroarchitektur“ zur Gesamtarchitektur bei. Die Gesamtarchitektur der Applikation ergibt sich aus der Summe der eingesetzten bzw. verwendeten Muster.

In diesem Kapitel untersuchen wir die Migration einer Reihe von Mustern, wie sie in der Klassenbibliothek oder in Applikationen zu finden sind. Durch die getrennte Betrachtung der Migration von Teilen der Gesamtarchitektur in Form von Mustern erhalten wir eine Reihe von Einzelproblemen. Bei der Migration einer gesamten Applikation sind wir mit der Summe all dieser Einzelprobleme konfrontiert. Eine Zusammenfassung dieser Einzelprobleme erfolgt im nachfolgenden Kapitel.

Der Aufbau dieses Kapitels orientiert sich an dem in Abschnitt 2.4 beschriebenen Muster-System. Danach verteilen sich Muster auf verschiedene Abstraktionsebenen — von der architektonischen Framework-Ebene über die Entwurfsmuster-Ebene bis zu den Idiomen auf der Sprachebene. Auch wenn Muster prinzipiell sprachunabhängig sind, so ist doch die konkrete Ausprägung auf den verschiedenen Plattformen mehr oder weniger spezialisiert. Manche Muster sind auf der einen Plattform notwendig, während sie auf der anderen Plattform überflüssig sind (Bsp: Double Dispatch). Manchmal ermöglicht eine Plattform eine besonders effiziente Realisierung eines Musters, während das gleiche Muster auf der anderen Plattform nicht so einfach realisiert werden kann (Bsp: Proxy). Insbesondere diese Unterschiede bzw. Schwierigkeiten, die bei der Übertragung eines Musters von der Smalltalk– auf die Java–Plattform auftreten, sind für die Untersuchung von Interesse.

Die in diesem Kapitel beschriebenen Muster treten bei der Migration auf zweierlei Art in Erscheinung. Als wiederverwendbares Design werden sie bei der Applikationsentwicklung dazu eingesetzt, die Architektur der Applikation zu gestalten [BJ94]. Die Implementierung dieser Muster verwendet die Elemente der Plattform. Bei der Migration nach Java müssen also die Muster im Smalltalk–Code erkannt und mit den Mitteln von Java implementiert werden. Beispielsweise muß das Proxy–Muster von einer Applikation meist selbst realisiert werden. Die zweite Art, auf die Muster in Erscheinung treten, ist die Benutzung der Klassenbibliothek. Die Frameworks und Toolkits enthalten Muster, um die Verwendung ihrer Elemente durch eine Applikation zu strukturieren [Joh92]. Für die Migration bedeutet das, daß die Verwendung der Plattformelemente in Form von Mustern erkannt werden muß. In Java muß das entsprechende Plattformelement über dessen spezifisches Muster angesprochen werden. Beispielsweise bieten Smalltalk wie Java Iteratoren an, die bei der Applikationsentwicklung meist unverändert verwendet werden.

Format der Beschreibung

Musterbeschreibungen sind durch ein festes Format gekennzeichnet. Die Beschreibung der Migration von Mustern in diesem Kapitel folgt ebenfalls einem festen Format. Dieses Format soll helfen, die Darstellung der Migration klarer zu machen und eine schnelle Orientierung in dem Katalog ermöglichen.

Name
Struktur
Unterschiede
Migration
Probleme
Abhängigkeiten

Der **Name** eines Musters ist essentiell für dessen Beschreibung. Die in dieser Arbeit verwendeten Namen entstammen direkt der Originalliteratur. Zunächst wird die **Struktur** des Musters dargestellt. Dies kann allerdings nur ein kurzer Abriß sein. Für eine detaillierte Darstellung der Struktur der behandelten Muster muß auf die Originalliteratur verwiesen werden. Jede Sprachplattform bietet spezifische Möglichkeiten, ein Muster zu implementieren bzw. realisiert das Muster auf spezifische Weise. Daher sollen an dieser Stelle die **Unterschiede** in der Implementierung bzw. Realisierung der verschiedenen Muster dargelegt werden. Ein Hauptaugenmerk dieser Arbeit liegt auf **Migration**. Daher wird hier untersucht, wie die Unterschiede der Implementierungen der Muster überwunden werden können. Um zu entscheiden, ob die Migration einer Applikation gegenüber einer kompletten Neuentwicklung sinnvoll erscheint, ist es wichtig zu wissen, wo die **Probleme** liegen, die eine Migration mit sich bringt bzw. die die Migration schwierig machen. Die Implementierung eines Musters verwendet einige Elemente der Sprachplattform. Diese **Abhängigkeiten** werden hier aufgelistet.

5.1 Migration von Idiomen

Idiome treten bei der Migration in Form von konkreten Implementierungen im Programmtext auf. Sie werden hier betrachtet, weil sie die typische Verwendung der Elemente der Sprachplattform darstellen. Die verwendeten Elemente werden bei der Migration durch die Elemente der Java–Plattform ersetzt. Diese müssen nach der Migration in einer für Java typischen Weise verwendet werden.

5.1.1 Constructor Method

Struktur Eine Konstruktormethode definiert die Parameter, die zur Erzeugung einer wohlgeformten Instanz einer Klasse benötigt werden. [Bec97, S.23]

Unterschiede In Smalltalk ist kein Sprachkonstrukt zum Erzeugen einer Instanz definiert. Das Meta–Objekt–Protokoll (4.1.7) definiert für jede Klasse eine Klassenmethode zum Erzeugen einer Instanz. Um eine parametrisierte Instanziierung zu ermöglichen, muß die Klasse eine eigene Klassenmethode zu diesem Zweck implementieren. Diese ersetzt die übliche Instanzerzeugungsmethode `new` und definiert die Parameter, die zum Anlegen einer wohlgeformten Instanz benötigt werden. Ein instanzerzeugender Aufruf muß die hierin definierten Parameter übergeben. Dieses Idiom ist lediglich eine Konvention.

Konstruktoren sind in Java definierte Sprachkonstrukte. Jede instanziierbare Klasse benötigt einen Konstruktor. Der Konstruktor definiert die Parameter, die er zur Erzeugung einer wohlgeformten Instanz benötigt. Konstruktoren in Java und Konstruktormethoden in Smalltalk verhalten sich nicht vollständig identisch. Ein Konstruktor in Java ruft implizit zuerst den Konstruktor seiner Superklasse auf (constructor chaining). In Smalltalk entscheidet hierüber der Programmierer. Parametrisierte Konstruktoren werden in Java nicht vererbt, während die Konstruktormethoden von Smalltalk an die Subklassen vererbt werden.

Migration Die Existenz von Konstruktormethoden in Smalltalk deutet normalerweise auf die Notwendigkeit hin, einer neuen Instanz Argumente übergeben zu müssen. Diese Argumente müssen in Java folglich einem Konstruktor übergeben werden, der die Initialisierung übernimmt. Falls in Smalltalk eine parametrisierte Konstruktormethode in einer Subklasse aufgerufen wird, muß in Java ein solcher Konstruktor in der Subklasse explizit implementiert werden.

Ein Java–Konstruktor ruft aufgrund der impliziten Konstruktorverkettung immer zuerst den Konstruktor der Superklasse auf. In Smalltalk muß dieser Aufruf explizit erfolgen. Daher kann der Code, der in der Konstruktormethode vor dem Aufruf des Konstruktors der Superklasse steht, zu Problemen führen. Falls dies umfangreiche Berechnungen sind, deren Werte zum Aufruf des Superklassen–Konstruktors benötigt werden, muß diese Berechnung entweder schon im instanziierenden Klienten erfolgen oder die Superklasse muß nachträglich nochmals aufgerufen werden.

Probleme

- Konstruktormethoden sind in Smalltalk lediglich Konvention, während sie in Java in der Sprache definiert sind.
- Java verwendet implizite Konstruktorverkettung, während sie in Smalltalk explizit gemacht werden muß.

Abhängigkeiten Instanzerzeugung (4.1.2), Klassenmethoden (4.1.1), Initialisierung (4.1.2)

5.1.2 Constructor Parameter Method

Struktur Der Konstruktor eines neu anzulegenden Objekts benötigt oft Parameter, um die Instanzvariablen so zu setzen, daß ein gültiges Objekt entsteht. Am einfachsten setzt der Konstruktor die Instanzvariablen direkt, jedoch kann das Setzen der Instanzvariablen eines neu angelegten Objekts komplex werden. Daher wird oft eine private Instanzmethode definiert, die das Setzen der Instanzvariablen übernimmt. [Bec97, S.25]

Unterschiede Diese Methode wird von einer Konstruktormethode (5.1.1) aufgerufen, um die neu angelegte Instanz zu initialisieren. Der Grund für dieses Idiom ist die Form der Kapselung in Smalltalk. Weil die Konstruktormethode eine Klassenmethode ist, kann sie auf die Instanzvariablen der neuen Instanz nicht zugreifen. Daher muß sie die Setter–Methoden verwenden, um die Variablen zu setzen. Diese verhalten sich aber evtl. anders, als bei der Initalisierung gewünscht.

Ein Java–Programm verwendet zur Instanzerzeugung die in die Sprache eingebauten Konstruktoren. Ein Java–Konstruktor lebt im gleichen Sichtbarkeitsbereich wie die Instanzvariablen und kann daher direkt zugreifen. Probleme bei der Initialisierung von Instanzvariablen löst Java mit instance initializers. Das von Beck erwähnte Problem, wie die Typen der Instanzvariablen dokumentiert werden, existiert in Java aufgrund des Typsystems nicht. Dieses Idiom wird also in Java nur sehr selten eingesetzt werden.

Migration Die Migration macht eine solche Methode meist überflüssig. Der Methodenrumpf wird dem Konstruktor oder einem der Initialisatoren zugeordnet.

Probleme

- Konstruktormethoden sind in Smalltalk lediglich Konvention, während sie in Java in der Sprache definiert sind.
- Java verwendet implizite Konstruktorverkettung, während sie in Smalltalk explizit gemacht werden muß.
- Instanzvariablen sind privat.
- Smalltalk definiert keinen Mechanismus zum Initialisieren einer Variablen.

Abhängigkeiten Instanzerzeugung (4.1.2), Instanz–Initialisierung (4.1.2), Sichtbarkeit von Variablen (4.1.1), Dynamisches Typsystem (4.1.1)

5.1.3 Shortcut Constructor Method

Struktur Manche Objekte werden an sehr vielen Stellen in einer Applikation erzeugt. Gesucht ist nun also ein Verfahren, wie diese Objekte angelegt werden können, ohne viel Schreibarbeit verrichten zu müssen. Die Erzeugung eines Objekts kann durch das Versenden einer Nachricht an einen der Parameter erfolgen, die für die Erzeugung des neuen Objekts benötigt werden. [Bec97, S.26]

Unterschiede In diesem Idiom wird die Konstruktormethode des zu erzeugenden Objekts nicht direkt aufgerufen, sondern von einem der Parameter. Beispielsweise definiert `Number` den Operator `@` zum Erzeugen eines Punktes aus einem Nummernpaar. Normalerweise werden durch dieses Idiom die Smalltalk–Klassen erweitert, die den primitiven Typen in Java entsprechen. Dazu werden die entsprechenden Klassen der Klassenbibliothek verändert.

In Java ist es weder möglich primitive Typen um neue Operatoren zu erweitern, noch bestehende Klassen um neue Methoden zu erweitern. Selbstgeschriebene Klassen können Methoden definieren, die genau die Aufgabe dieses Idioms erfüllen. Da sich der Programmierer hierdurch aber nicht sehr viel Schreibarbeit spart, wird eine solche Methode in Java meist über einen Konstruktoraufruf der zu instanziierenden Klasse ersetzt.

Migration Dieses Idiom funktioniert in Java nur für selbstgeschriebene Klassen. Daher müssen sämtliche Shortcut–Konstruktormethoden, die Basisklassen erweitern, ersatzlos gestrichen werden. Dies hat Einfluß auf die Klassen die diese Methoden verwenden. In ihnen muß der Aufruf der Shortcut Constructor Method durch den direkten Aufruf des Konstruktors ersetzt werden. Für selbstgeschriebene Klassen wird eine Shortcut–Konstruktormethode durch eine gewöhnliche Methode dargestellt, die den passenden Konstruktor aufruft und das erzeugt Objekt zurückgibt.

Probleme

- Die Basisklassen lassen sich in Smalltalk verändern, in Java dagegen nicht.
- Java unterscheidet primitive und Referenztypen.
- In Java können keine neuen Operatoren definiert werden.

Abhängigkeiten Base Class Extension 4.1.1, Objekterzeugung 4.1.2

5.1.4 Converter Method

Struktur Eine Konverter–Methode liefert ein Objekt, das das gleiche Protokoll versteht wie das Ausgangobjekt, aber zu einer anderen Klasse gehört (beispielsweise `Collection>>asSet`). [Bec97, S.28]

Unterschiede In der Smalltalk–Implementierung besitzen die beiden Klassen normalerweise eine gemeinsame Superklasse, die das gemeinsame Protokoll definiert. Der Empfänger legt eine neue Instanz der gewünschten Klasse an und initialisiert sie mit dem eigenen Status. Manchmal werden bestehende Klassen um eine solche Konverter–Methode erweitert.

In Java wird das gemeinsame Protokoll entweder von einer gemeinsamen Superklasse oder von einem Interface definiert. Falls beide Klassen das gleiche Interface implementieren, müssen sie nicht mehr derselben Subklassenhierarchie angehören. In Java legt die Konverter–Methode eine Instanz der gewünschten Klasse durch Aufruf des Konstruktors an. Diesem wird das Empfängerobjekt übergeben. Eine Erweiterung von bestehenden Klassen der Klassenbibliothek ist nicht möglich.

Migration Durch die Migration muß der Aufruf der Konstruktor–Methode in Smalltalk durch den Aufruf des Konstruktors in Java ersetzt werden (siehe 5.1.1). Die Klasse des gewünschten Objekts sollte einen Konstruktor definieren, dem das Ausgangsobjekt übergeben wird. Dieser füllt die neu angelegte Instanz.

In Smalltalk werden Basisklassen oft um Konverter–Methoden erweitert. In Java ist diese Erweiterung nicht möglich, weshalb solche Methoden bei der Migration ersatzlos wegfallen. Klienten einer solchen Methode müssen die gewünschte Instanz in Java also selbst anlegen. Falls die gewünschte Klasse keine passende Konverter–Konstruktor–Methode anbietet (siehe 5.1.5), muß die Konversion im Klienten erfolgen.

Probleme

- Basisklassen können in Java nicht erweitert werden.
- Smalltalk definiert keine Interfaces.

Abhängigkeiten Protokoll (4.1.1), Interfaces (4.1.1), Base Class Extension (4.1.1), Konstruktormethode (4.1.2)

5.1.5 Converter Constructor Method

Struktur Im Gegensatz zu Converter Method liefert dieses Idiom ein Objekt, das über ein anderes Protokoll angesprochen werden soll. Dazu wird die Klasse instanziiert, deren Protokoll das neue Objekt gehorchen soll. Das bisherige Objekt ist Argument für die Instanziierung. [Bec97, S.29]

Unterschiede In Smalltalk ist eine Konverter–Konstruktor–Methode als Konstruktor–Methode der Zielklasse ausgelegt, welche das Ausgangobjekt als Argument erhält. Die neue Instanz verwendet des Zustand des Ausgangobjekts, um den eigenen Zustand zu initialisieren. Der Name der Konverter–Konstruktor–Methode enthält den Typ des Ausgangsobjekts (Bsp: `Date>>fromString`).

Eine Java–Klasse stellt eine Konverter–Konstruktor–Methode durch einen Konstruktor dar. Der Typ des Ausgangsobjekts wird durch den formalen Parameter des Konstruktors festgelegt; der Name des Konstruktors ist durch den Klassennamen festgelegt. Bei komplexen Objekten ist es in Java möglich, ein Interface zu erzeugen, welches das gewünschte Protokoll definiert. Das Ausgangsobjekt müßte dann dieses Interface implementieren. Der Zugriff auf dieses Objekt über das zweite Protokoll geschieht durch eine Typumwandlung im Klienten.

Migration Die Migration besteht aus einer Übersetzung der Konstruktor–Methode in einen Konstruktor (siehe 5.1.1). Falls in Java die Lösung über ein zusätzliches Interface gewählt wird, bedeutet das für die Klienten eine explizite Typumwandlung anstatt des Aufrufs der Konverter–Konstruktor–Methode. Außerdem ist dann zu beachten, daß der Klient lediglich mit einem Objekt arbeitet. Änderungen am Objekt, die über das eine Protokoll erfolgen, sind also auch über das andere Protokoll zu sehen.

In Smalltalk werden Basisklassen oft um Konverter–Methoden erweitert. In Java ist diese Erweiterung nicht möglich, weshalb solche Methoden bei der Migration ersatzlos wegfallen. Klienten einer solchen Methode müssen die gewünschte Konversion in Java also selbst vornehmen.

Probleme

- Basisklassen können in Java nicht erweitert werden.

Abhängigkeiten Base Class Extension 4.1.1, Konstruktormethode 4.1.2, Zweites Protokoll 4.1.1

5.1.6 Comparing Method

Struktur Um Objekte miteinander vergleichen zu können, muß die Klasse eine Vergleichsoperation implementieren. Die Objekte gehorchen dann einer „natürlichen“ Ordnung. [Bec97, S.32]

Unterschiede In Smalltalk definiert die Klasse `Object` die Methode `<=`. Die Vergleichsmethode erspart in manchen Fällen die Angabe eines Sortierblockes (beispielsweise bei `SortedCollection`). Die Vergleichsmethode muß zunächst testen, ob das zu vergleichende Objekt den gleichen Typ hat wie der Empfänger. Anschließend kann der Status verglichen werden.

In Java definiert eine Applikation die Ordnung der Instanzen einer Klasse durch implementieren des Interface `Comparable`. Dieses enthält die Methode `compareTo(Object o)`. Der Test auf Typverträglichkeit des Arguments wie in Smalltalk findet sich auch im Java–Code wieder, weil das zu vergleichende Objekt als Instanz von `Object` übergeben wird. Dann erfolgt ein *Downcast*, um auf den Status des Arguments zugreifen zu können.

Vorteil der Java–Lösung ist, daß durch die Angabe des Interfaces im Kopf der Klassendefinition schon durch Studium der Klassendefinition klar ist, ob ihre Instanzen vergleichbar sind. In einer Smalltalk–Klasse muß die Vergleichsmethode gesucht werden, um darüber Klarheit zu haben.

Migration Damit eine Klasse in Java vergleichbar ist, muß sie das `Comparable`–Interface implementieren. In Java muß vor dem Zugriff auf den Status des Arguments ein Downcast eingefügt werden.

Probleme

- Smalltalk kennt keine Interfaces.
- Statische Typisierung in Smalltalk im Gegensatz zu dynamischer Typisierung in Java.

Abhängigkeiten `Object`–Framework 4.3.1

5.1.7 Execute Around Method

Struktur Vor und nach einer Operation müssen in machen Fällen bestimmte Aktionen ausgeführt werden. [Bec97, S.37]

Unterschiede In Smalltalk erhält die Execute Around–Methode vom Klienten einen Codeblock, der von Aktionen umrahmt werden soll. Die Execute Around–Methode führt zunächst die erste Aktionen aus, wertet den erhaltenen Block aus und schließt die Methode mit der zweiten Aktion ab.

In Java sind Codeblöcke keine Objekte erster Klasse, weshalb sie nicht als Argumente verwendet werden können. Daher muß der Execute Around–Methode ein Objekt übergeben werden, welches den auszuführenden Code kapselt. Hierzu muß ein Interface definiert sein, welches vom übergebenen Objekt implementiert wird. Nach der ersten Aktion innerhalb der Methode ruft sie dieses Objekt über das im Interface definierte Protokoll auf. Anschließend erfolgt die abschließende Aktion.

Migration Zunächst muß ein Interface angelegt werden. Die Execute Around–Methode und das erhaltene Objekt kommunizieren über das in diesem Interface definierte Protokoll. Falls das Objekt nicht nur über die externe Schnittstelle auf den Klienten zugreift, muß es eine Instanz einer inneren Klasse sein. Eine Instanz einer inneren Klasse hat Zugriff auf die Internas des Objekts, welches die innere Klasse definiert. Näheres zur Migration von Blöcken ist in 4.1.3 nachzulesen.

Probleme
- In Java sind Blöcke keine Objekte erster Klasse.

Abhängigkeiten Blöcke (4.1.3), Innere Klassen (4.1.1), Protokoll (4.1.1), Interfaces (4.1.1)

5.1.8 Debug Printing Method

Struktur Zur Verwendung in Listen, Tabellen, Text–Widgets, Entwicklungswerkzeugen, etc. muß ein Objekt eine textuelle Beschreibung von sich selbst erzeugen können. [Bec97, S.39]

Unterschiede In Smalltalk definiert hierzu die Klasse `Object` die Methode `printOn:`, welche von einer interessierten Klasse überschrieben werden kann.

Die Klasse `Object` definiert in Java die Methode `toString()`, welche genau die gleiche Funktion hat wie `printOn:` in Smalltalk.

Migration Die Smalltalk–Methode `printOn:` erhält im Gegensatz zur Java–Methode `toString()` einen Stream als Argument. Sie muß ihre eigene textuelle Repräsentation an diesen Stream anhängen. In Java muß die Methode lediglich die textuelle Repräsentation zurückgeben.

Probleme
- Keine.

Abhängigkeiten String 4.2.1, Objekt–Framework 4.3.1, Ströme 4.2.3

5.1.9 Double Dispatch

Struktur Dieses Idiom löst das Problem, wie das Verhalten eines Methodenaufrufs vom Typ des Parameters abhängig gemacht werden kann. [Bec97, S.55]

Unterschiede Die Empfängermethode delegiert den Aufruf an den Parameter, wobei der Typ des ersten Emfängers im Namen der Nachricht an den zweiten Empfänger kodiert ist. Beispielsweise arbeiten auf diese Art `Integer` und `Float` zusammen. Die Methode `Integer>>add:` ist so implementiert, daß sie dem Argument die Nachricht `addInteger:` schickt. Ziel dieses Idioms ist es, eine Fallunterscheidung in der Empfängermethode zu verhindern.

Dieses Idiom ist in Java nicht notwendig, weil überschriebene Methoden in Java anhand der Typen der Argumente ausgewählt werden können. Dies wird durch das statische Typsystem in Java ermöglicht.

Migration Die Migration dieses Idioms führt zu einer Reihe gleichnamiger Empfängermethoden, deren Signaturen sich im Typ des Parameters unterscheiden. Die Kodierung des Typs im Namen der Methode wie in Smalltalk fällt weg.

Probleme
- Smalltalk ist dynamisch getypt im Gegensatz zur statischen Typisierung in Java.
- Eine Methode kann in Smalltalk nicht überladen werden.

Abhängigkeiten Nachrichten (4.1.1), Dynamische Typisierung (4.1.1)

5.1.10 Mediating Protocol

Struktur Zwei Objekte interagieren über bestimmte Methoden. Das Mediating Protocol ist die Gesamtheit dieser Methoden, die zur Interaktion zwischen zwei Objekten verwendet wird. [Bec97, S.57]

Unterschiede Die Definition eines Mediating Protocol ist in Smalltalk informell gegeben durch die spezifizierten Methoden, über die zwei Objekte miteinander sprechen. In der Entwicklungsumgebung kann eine Methode in eine bestimmte Protokollklasse eingeordnet werden (siehe 4.1.1). Diese Zuordnung ist aber nicht zwingend. Sie hat lediglich Kommentarcharakter und wird normalerweise zur Strukturierung der Methoden *eines* Objekts verwendet. Um das Protokoll zwischen *zwei* Objekten herauszustellen, existiert in Smalltalk kein Mechanismus.

In Java eignen sich Interfaces zur formalen Definition eines Mediating Protocol. Alle darin deklarierten Methoden müssen in einer implementierenden Klasse vorkommen. Dadurch kann ein Objekt unter dem Typ des Interfaces erscheinen. Damit ist die Dokumentation eines Protokolls eindeutig und die Überprüfbarkeit durch den Compiler ist gegeben. Ein anderes Objekt, das über dieses Protokoll mit dem Empfänger kommuniziert, ruft die Methoden über dieses Interface auf.

Migration Zunächst müssen genau die Methoden erkannt werden, die das Mediating Protocol definieren. Diese werden in zwei Interfaces zusammengefaßt. Sämtliche Nachrichten zwischen den kommunizierenden Objekten werden in Java nur über diese Interfaces verschickt.

Probleme

- In Smalltalk können Protokolle zwischen Objekten nicht formal definiert werden wie durch Interfaces in Java.

Abhängigkeiten Protokolle (4.1.1), Interfaces (4.1.1)

5.1.11 Explicit Initialization

Struktur Dieses Idiom setzt den Default–Wert einer Menge von Variablen auf einmal. Dazu wird der Initialisierungscode an einer Stelle zusammengefaßt und zu Beginn ausgeführt. [Bec97, S.83]

Unterschiede Eine Smalltalk–Implementierung verwendet zum Setzen der Default–Werte von Instanzvariablen eine Initialisierungsmethode. Diese muß beim Anlegen einer neuen Instanz explizit aufgerufen werden. Normalerweise wird hierzu die Konstruktormethode überschrieben (siehe 5.1.1). Klassenvariablen müssen in Smalltalk normalerweise nicht initialisiert werden, weil das Klassenobjekt durch seine Existenz im Image das Verlassen des Programms bis zum Neustart überdauert. Falls eine Initialisierung nach einem Neustart dennoch notwendig ist, kann dies durch Überschreiben der `startup`–Klassenmethode geschehen, die beim Neustart des Systems aufgerufen wird.

In Java werden Instanzvariablen entweder im Konstruktor oder in einem *instance initializer* initialisiert. Zur Initialisierung von Klassenvariablen existieren *static initializers*. Diese werden beim Laden der Klasse einmal ausgeführt. Daneben kann jede Variable bei ihrer Deklaration mit einem literalen Wert initialisiert werden.

Migration Der Code zur Initialisierung von Klassenvariablen wird in Java in einem *static initializer* untergebracht. Der Aufruf der Initialisierungsmethode von `startup` aus muß entfernt werden.

Der Code zur Initialisierung von Instanzvariablen wird in Java in einem *instance initializer* untergebracht. Der Aufruf der Initialisierungsmethode in der Konstruktormethode muß entfernt werden.

Probleme

- Die zeitliche Persistenz eines Klassenobjekts über einen Neustart des Systems hinweg.
- Smalltalk definiert keinen Mechanismus zum Initialisieren einer Variablen.
- Smalltalk definiert kein Sprachkonstrukt zum Erzeugen einer Instanz.

Abhängigkeiten Instance–Erzeugung (4.1.2), Meta–Objekt–Protokoll (4.1.7), Initialisierung (4.1.1)

5.1.12 Lazy Initialization

Struktur Dieses Idiom setzt den Default–Wert einer einzelnen Variablen. Bei einem Zugriff auf die Variable wird festgestellt, ob sie noch uninitialisiert ist. Der erste Zugriff setzt die Variable auf ihren Default–Wert. [Bec97, S.85]

Unterschiede Eine Smalltalk–Klasse implementiert sehr häufig eine Getter–Methode, um eine Instanzvariable von außerhalb der Klasse zugänglich zu machen. Dieses Idiom fügt in diese Methode einen Test ein, der zunächst feststellt, ob die Variable schon initialisiert wurde. Die Initialisierung geschieht in dieser Methode.

In Java ist dieses Idiom ebenfalls möglich. In Java sind Getter–Methoden nicht nötig, weil die Sichtbarkeit von Instanzvariablen durch Modifikatoren auf externe Klassen ausgeweitet werden kann. Falls die Applikation nicht darauf angewiesen ist, daß die Variable erst zum Zeitpunkt des ersten Zugriffs initialisiert wird, kann diese Form der Initialisierung in Java durch eine lokale Initialisierung ersetzt werden. Dabei erhält die Variable ihren Default–Wert schon bei ihrer Deklaration bzw. bei Ausführen des *instance initializers*.

Migration Bei der Migration kann entweder die Getter–Methode nach Java übertragen werden oder die Initialisierung durch eine Sprachkonstrukt erfolgen. Bei Übertragung der Getter–Methode sollte die Sichtbarkeit der Variablen auf `private` gesetzt werden, um den Zugriff auf die Variable über die Getter–Methode zu erzwingen. Falls die Initialisierung der Variablen durch ein Sprachkonstrukt erfolgt, kann die Getter–Methode weggelassen werden. Die Sichtbarkeit der Variablen muß dann angepaßt werden.

Probleme

- Smalltalk definiert kein Sprachkonstrukt zur Initialisierung von Variablen.
- Smalltalk definiert kein Sprachkonstrukt zum Erzeugen einer Instanz.

Abhängigkeiten Sichtbarkeit von Variablen (4.1.1), Initialisierung von Variablen (4.1.1), Instanzmethoden (4.1.1)

5.1.13 Enumeration Method

Struktur Manche Objekte enthalten eine Collection zum Speichern von Elementen. Solche Objekte benötigen eine Möglichkeit, einem Klienten Zugang zu diesen Elementen zu verschafffen, ohne die Collection komplett nach außen sichtbar zu machen. [Bec97, S.99]

Unterschiede In Smalltalk implementiert ein solches Objekt die Methode `do:` welche einen internen Iterator darstellt. Dieser Iterator erhält den auf den Elementen auszuführenden Code in Form eines Codeblocks. Die Ausführung dieser Nachricht delegiert sie an die verwendete Collection.

Die Klassenbibliothek von Java definiert keine internen Iteratoren. Ein Klient kann nur über einen externen Iterator Zugriff auf die Elemente einer Collection erhalten. Iteratoren implementieren entweder das Interface `Enumeration` oder das Interface `Iterator`. Die Anforderung eines Iterators delegiert das Objekt an die verwendete Collection und gibt der erhaltenen externen Iterator an den Klienten zurück.

Migration Die Delegation der Nachricht ist leicht umzuschreiben. Die Tatsache, daß Java nur externe Iteratoren kennt, macht allerdings mehr Arbeit. Das bedeutet für den Klient, daß er nicht mehr einfach einen Codeblock angeben, der auf allen Elementen der Collection arbeiten soll. Der Klient muß selbst eine Schleife programmieren, um über alle Elementen zu iterieren und den Code im Codeblock lokal auszuführen.

Probleme

- Smalltalk kennt interne und externe Iteratoren für Collections während Java nur externe Iteratoren definiert.
- In Java sind Codeblöcke keine Objekte erster Klasse.

Abhängigkeiten Externe Iteratoren 4.2.3, Collections 4.2.1, Interfaces 4.1.1

5.1.14 Duplicate Removing Set

Struktur Eine häufig gestellte Aufgabe ist das Entfernen von doppelten Elementen aus einer Collection. Die Idee ist die Speicherung der Elemente in einer Menge unter Ausnutzung der Mengen–Eigenschaft, daß eine Menge kein Element doppelt enthalten kann. [Bec97, S.154]

Unterschiede Eine Smalltalk–Applikation erzeugt hierfür eine Instanz von `Set` und initialisiert sie mit den Elementen der vorhandenen Collection. Eine Instanz von `Set` versteht das gleiche Protokoll wie die ursprüngliche Collection, weshalb die Applikation unverändert auf sie zugreifen kann.

Das Collection–Toolkit definiert auch in Java eine Menge. Die Applikation muß also eine Implementierung hiervon instanziieren und mit der vorhandenen Collection initialisieren. Hierfür stellen die Collections–Klassen Konvertiermethoden bereit. Da `Set` ein Subinterface von `Collection` ist, versteht es ebenfalls das Collection–Protokoll.

Migration Bei der Migration ist ein Problem zu beachten, das bei jeder Verwendung einer Collections auftreten kann: Instanzen primitiver Typen können in Collections nicht gespeichert werden. Sie müssen erst mit dem passenden Wrapper–Objekt gekapselt werden. Da die natürliche Art in Java zur Speicherung von primitven Types ein Array wäre, ist das Beseitigen von doppelten Elementen nicht so einfach. Ein Array ist nicht Teil des Collection–Toolkits und kann deshalb nicht auf einfache Art in eine Instanz von `Set` umgewandelt werden. Ob sich der Aufwand lohnt, jedes Elemente eines Arrays in einen Wrapper zu packen, dann in einer Collection zu speichern und als Menge zu verwenden, ist im Einzelfall zu entscheiden. In diesem Fall ist evtl. eine algorithmische Lösung angebracht.

Probleme

- Java unterscheidet primitive Typen von Referenztypen
- Die Klassen des Collection–Toolkits können in Java nur Objekte speichern — keine primitiven Elemente.
- Das Collection–Toolkit definiert keinen einfachen Mechanismus, um ein Array in eine Collection zu verwandeln.

Abhängigkeiten Interface (4.1.1), Collections (4.2.1), Converter Method (5.1.4), Datentypen (4.1.1)

5.1.15 Temporarily Sorted Collection

Struktur Um eine Collection zu sortieren, kann sie temporär in eine sich selbst sortierende Collection konvertiert werden, wenn der Aufwand zu groß wäre, dauerhaft eine sich selbst sortierende Collection zu verwenden. [Bec97, S.155]

Unterschiede Eine Smalltalk–Applikation erzeugt jedesmal, wenn die Collection sortiert werden muß eine neue `SortedCollection`. Diese sortiert sich selbst anhand der natürlichen Ordnung der Elemente oder anhand eines von der Applikation übergebenen Sortierblockes.

Java definiert zwei getrennte Mechanismen zur Sortierung von Collections und Arrays. Die Methode `java.util.Collections.sort()` dient dem Sortieren von Collections. Bei Bedarf kann ein Sortierblock angegeben werden. Ein Array speichert primitve Elemente. Die Deklaration eines Arrays legt fest, welchen Typ die gespeicherten Elemente haben müssen. Die mehrfach überladene Methode `java.util.Arrays.sort()` dient dem Sortieren der verschiedenen möglichen Arrays.

Migration Die Migration muß grundsätzlich unterscheiden zwischen den Typen der gespeicherten Elemente. Falls die Elemente in Java Referenztypen sind, erfolgt die Speicherung in einer Collection. Diese kann bei Bedarf mit obiger Methode sortiert werden. Eine Speicherung in einer sich selbst sortierenden Collection ist somit nicht notwendig. Die Verwendung eines Sortierblockes stößt auf die Schwierigkeit, daß Java keine Codeblöcke kennt. Daher muß der Methode ein Objekt übergeben werden, welches das `Comparator`–Interface implementiert.

Falls die Elemente der Collection in Java primitive Typen sind, kann die Collection entweder durch ein Array dargestellt werden oder die Elemente werden mit Wrapper–Objekten gekapselt, um sie in einer Collection zu speichern. Die Sortierung der zweiten Möglichkeit ist genau wie oben beschrieben zu behandeln. Die Sortierung der ersten Möglichkeit verwendet die entsprechende Sortiermethode für Arrays.

Probleme

- Java unterscheidet zwischen primitiven und Referenztypen.
- Java kennt keine Codeblöcke.
- Collections können in Java keine primitiven Elemente speichern.

Abhängigkeiten Codeblöcke 4.1.3, Vergleich von Objekten 4.3.1 Interface (4.1.1), Collections (4.2.1), Datentypen (4.1.1)

5.1.16 Cascade

Struktur Eine Kaskade definiert in Smalltalk die Möglichkeit, mehrere Nachrichten an einen Empfänger zu schicken, wobei der Empfänger nur einmal angegeben werden muß. [Bec97, S.183]

Unterschiede Die Möglichkeit zum Aufbau einer Kaskade reduziert in vielen Fällen die Notwendigkeit, eine temporäre Variable anzulegen.

Java biete keine Möglichkeit zum Aufbau einer Kaskade. Daher muß das zu Beginn einer Kaskade stehende Objekt zunächst in einer temporären Variablen gespeichert werden. Weitere Nachrichten an dieses Objekt werden durch Zugriff auf diese Variable verschickt.

Migration Die Migration besteht aus der Erzeugung einer temporären Variablen und der Expandierung der Methodenaufrufe. In manchen Fällen ist die letzte Nachricht einer Kaskade die Nachricht `yourself`. Dieser Fall wird in 5.1.17 behandelt.

Probleme

- Java definiert keine Kaskaden

Abhängigkeiten Nachrichten und Kaskadierung 4.1.1 Temporäre Variablen (4.1.1),

5.1.17 Yourself

Struktur Manchmal ist es nötig, am Ende einer Nachrichtenkaskade den Empfänger zu erfahren. [Bec97, S.186]

Unterschiede Falls die letzte Nachricht der Kaskade in Smalltalk nicht den Empfänger zurückgibt, muß `yourself` angehängt werden, um den Empfänger der Nachrichten in der Kaskade zu erfahren. Diese Nachricht macht nichts anderes, als den Empfänger dieser Nachricht zurückzugeben. Nach Beck ist dieses Idiom der Spitzenreiter unter den verwirrenden Nachrichten von Smalltalk.

Da in Java keine Nachrichtenkaskaden erlaubt sind, muß der Empfänger jedesmal explizit angegeben werden (siehe 5.1.16). Damit wird `yourself` überflüssig.

Migration Durch die Migration fällt diese Nachricht ersatzlos weg.

Probleme

- Java definiert keine Kaskaden.

Abhängigkeiten Nachrichten und Kaskadierung (4.1.1), Temporäre Variablen (4.1.1), Rückgabewert einer Methode (4.1.1)

5.1.18 Interesting Return Value

Struktur Eine Methode sollte immer klar machen, ob der zurückgegebene Wert weiterverwendet werden kann. [Bec97, S.188]

Unterschiede Falls eine Methode in Smalltalk nicht explizit einen Rückgabewert definiert, liefert sie den Empfänger der Nachricht. Um klar zu machen, wann der Rückgabewert einer Methode zur Weiterverwendung gedacht ist, sollte in genau diesen Fällen ein explizites `return` erfolgen — auch wenn dadurch der Empfänger selbst zurückgegeben wird. Das Zurückgeben des Emfängers erleichtert das Verwenden eines Methodenaufrufs in einer Nachrichtenkette.

Aufgrund der statischen Typisierung muß eine Java–Methode schon im Methodenkopf angeben, welche Typ der Rückgabewert hat. Dadurch ist klar, ob die Methode ein weiterzuverwendendes Objekt zurückgibt. Die Return–Anweisung muß nicht konsultiert werden.

Migration Dieses Idiom macht es der Migration besonders einfach zu entscheiden, welchen Rückgabewert eine Methode spezifiziert. Allerdings ist dieses Idiom nur eine Konvention. Die Migration kann sich nicht darauf verlassen daß jedesmal, wenn der Empfänger für den Klienten von Interesse ist, eine Return–Anweisung dies explizit tut. Durch die Migration wird der Typ des zurückgegebenen Wertes im Methodenkopf angegeben.

Probleme

- Smalltalk zwingt eine Methode nicht dazu, eine explizite Aussage über den Typ des Rückgabewertes zu machen.
- Smalltalk verwendet dynamische Typisierung im Gegensatz zu statischer Typisierung in Java.

Abhängigkeiten Nachrichtenverkettung (4.1.1), Rückgabewert (4.1.1), Dynamische Typisierung (4.1.1)

5.2 Migration von Entwurfsmustern

Entwurfsmuster spielen bei der Applikationsentwicklung in zweierlei Hinsicht eine wichtige Rolle. Als wiederverwendbares Design dienen sie dem Applikationsentwickler als Hilfsmittel bei der Strukturierung der Applikation. Für ihre Implementierung werden Plattformelemente verwendet. Bei der Migration müssen diese Elemente durch die Elemente von Java ersetzt werden. In der Klassenbibliothek treten Entwurfsmuster ebenfalls in Erscheinung. Dort geben sie vor, wie die Elemente der Klassenbibliothek verwendet werden. Damit beeinflussen sie die Architektur einer Applikation, die diese Plattformelemente verwendet. Ein Entwurfsmuster wird in dieser Arbeit ausschließlich dann behandelt, wenn dessen Smalltalk–Implementierung einen signifikanten Unterschied zur Java–Implementierung aufweist bzw. die Smalltalk–Implementierung eine Eigenschaft der Smalltalk–Plattform verwendet, die in Java nicht vorhanden ist.

Der Schwerpunkt liegt in der Darstellung, welche Elemente der Sprachplattformen zur Implementierung der Entwurfsmuster verwendet werden. Die Darstellung der Struktur der untersuchten Entwurfsmuster kann hier nur ein Abriß sein. Für eine vollständige Darstellung muß auf die Originalliteratur verwiesen werden.

5.2.1 Abstract Factory

Struktur Dieses Muster dient dazu, Produkte einer Produktfamilie mit Hilfe einer Factory anzulegen, ohne deren konkrete Klassen wissen zu müssen. Ein *Klient* legt die *konkreten Produkte* einer Produkt–Familie nicht direkt an, sondern kennt nur die *concrete factory* der Familie, die er betrachtet. Der Klient spricht die concrete factory über das Interface der *abstract factory* an.

Um ein konkretes Produkt zu erzeugen, verwendet der Klient eine Methode der abstract factory, die für alle Objekte eines bestimmten Typs über alle Familien hinweg gleich ist. Der Klient referenziert ein konkretes Produkt über ein *abstraktes Produkt*–Interface. [ABW98, S.31]

Unterschiede In Smalltalk sind drei verschiedene Implementierungsvarianten dieses Musters möglich. Die *erste Variante* definiert in der abstract factory–Klasse die Methoden zum Erzeugen der Produkte. Jede concrete factory–Klasse muß diese Methoden implementieren. Diese Implementierungen erzeugen die konkreten Produkte. Die *zweite Variante* verwendet einen Teilekatalog, der für jede concrete factory die spezifischen Produkt–Klassen enthält. Dadurch ist nicht mehr für jeden Produkttyp eine eigene Methode zu implementieren. Die *dritte Variante* verwendet die reflexiven Eigenschaften von Smalltalk. Mit Hilfe dieser Eigenschaften wird der Name der zu instanziierenden Produkt–Klasse dynamisch erzeugt.

In Java sind prinzipiell alle drei Implementierungsvarianten möglich. Bei der ersten Variante muß ebenfalls für jeden Produkttyp eine eigene Erzeugungsmethode in allen concrete factory–Klassen implementiert werden. Der Teilekatalog der zweiten Variante wird in Java durch einen `Vector` oder ein Element der Collection–Klassen dargestellt. Der Katalog speichert die Klassenobjekte der zu instanziierenden Produkte. Die dritte Variante erzeugt den Namen der zu erzeugenden Klasse zur Laufzeit. Auch in Java läßt sich der Name einer Klasse zur Laufzeit zusammenbauen. Über `Class.forName()` kann das Klassenobjekt ermittelt werden, über welches das konkrete Produkt instanziiert werden kann.

Migration Vom Standpunkt der Überprüfbarkeit der Typsicherheit durch den Compiler aus gesehen ist in Java die erste Variante vorzuziehen. Das würde allerdings ein Ausprogrammieren sämtlicher factory–Methoden bedeuten. Auf der anderen Seite fiele dadurch der Code zum Füllen der Teilekataloge weg.

Bei der Migration der zweiten und dritten Variante wird das konkrete Produkt nicht durch das Sprachkonstrukt `new` erzeugt, sondern durch das Klassenobjekt seiner Klasse. Dadurch ist

ein Downcast auf die konkrete Klasse nach dem Erzeugen der Instanz erforderlich. In Smalltalk ist dieser Downcast aufgrund des dynamischen Typsystems nicht notwendig. Durch das dynamische Ermitteln des Klassenobjekts in der dritten Variante wird die Überprüfbarkeit noch weiter verringert.

Probleme

- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.

Abhängigkeiten Abstrakte Klassen (4.1.1), Instanzerzeugung (4.1.2), Collection–Protokoll (4.2.1), Symbole (4.1.1), Reflexion (4.1.7)

5.2.2 Factory Method

Struktur Durch dieses Entwurfsmuster wird ein Protokoll zur Erzeugung eines Produkts definiert. Die Entscheidung, welche konkrete Klasse instanziiert wird, fällt in einer Unterklasse.

Ein Klient erzeugt ein *concrete product* durch Aufruf einer *creator*–Klasse. Ein *concrete creator* ist als Subklasse von creator für die konkrete Instanziierung zuständig. Dieser concrete creator implementiert die *factory method*, die in der creator–Klasse als abstrakte Methode definiert ist.

Dieses Entwurfsmuster wird typischerweise gebraucht um ein Framework zu realisieren. Oft müssen in einem Framework parallele Klassenhierarchien instanziiert werden, wobei das Framework die abstrakten Klassen dazu liefert. In der abstrakten Klasse wird eine factory method definiert, die von der konkreten Klasse implementiert werden muß. Klienten sehen diese factory method nicht direkt, sondern arbeiten über die Schnittstelle, die diese Methode verwendet. [ABW98, S.63]

Unterschiede Die verschiedenen Implementierungsmöglichkeiten folgen dem Muster „Abstract Factory“. Die einzige Implementierungsvariante dieses Musters, die spezifische Elemente von Smalltalk verwendet, setzt eine Klasseninstanzvariable ein. Hierbei definiert die creator–Klasse eine Klasseninstanzvariable, die die Klasse des concrete product enthält. Die factory method kann dadurch in der abstrakten creator–Klasse nicht nur definiert, sondern auch implementiert werden. Sie verwendet die Klasseninstanzvariable, um die Klasse des zu instanziierenden concrete produkt zu erfragen. Damit fällt die Notwendigkeit weg, in jeder Subklasse von creator eine factory method zu implementieren. Bei der Initialisierung der concrete creator–Klassen wird die Klasseninstanzvariable mit der Klasse des concrete product gefüllt. Durch die Einführung dieser Klasseninstanzvariablen fällt die abstrakte factory–Methode weg.

In Java sind keine Klasseninstanzvariablen definiert. Eine Lösung dieses Problems ist, das Entwurfsmuster in seiner ursprünglichen Form zu implementieren. Dadurch wird die abstrakte factory–Methode wieder eingeführt. Die Implementierungen dieser factory–Methode in den concrete creator–Klassen liefern die Klasse des concrete product zurück. Diese Klasse kann dann im creator instanziiert werden.

Migration Durch die Migration wird die Klasseninstanzvariable durch die abstrakte factory–Methode ersetzt. Weil dies lediglich intern die Implementierung des Musters modifiziert, ändert sich für Klienten der creator–Klasse nichts.

Probleme

- In Java sind keine Klasseninstanzvariablen definiert.

Abhängigkeiten Abstrakte Klasse 4.1.1, Klasseninstanzvariable 4.1.1, Reflexion 4.1.7, Klassenmethoden 4.1.1

5.2.3 Prototype

Struktur Dieses Entwurfsmuser beschreibt eine Möglichkeit, Arten von Objekten durch eine prototypische Instanz zu spezifizieren. Ein neues Objekt wird durch Kopieren des *Prototyps* erzeugt. [ABW98, S.77]

Unterschiede Die Erzeugung einer neuen Instanz besteht aus einem Kopieren der prototypischen Instanz und anschließendem Anpassen. Um ein Objekt zu kopieren, existieren in Smalltalk die beiden Methoden `shallowCopy` und `deepCopy`. Die Klonierungsmethode des Prototypen verwendet eine dieser Methoden, um eine Kopie von sich selbst anzulegen.

In Java dagegen existiert nur die eine Methode `clone()`. Standardmäßig erzeugt sie eine flache Kopie wie bei `shallowCopy`. Falls auch Objekte kopiert werden sollen, die im Prototypen enthalten sind, muß die `clone()`–Methode überschrieben werden.

Migration Falls die Smalltalk–Version nur `shallowCopy` verwendet, reicht in Java die Verwendung der unveränderten `clone()`–Methode. Falls die Smalltalk–Applikation allerdings die Kopiermethode überschreibt oder sogar `deepCopy` verwendet, muß dies in der Java–Applikation nachgezogen werden. In einem solchen Fall sind alle Klassen, die direkt oder indirekt als Element des Prototypen vorkommen können, zu untersuchen. Dieses Überschreiben von `clone()` kann Einfluß auf andere Klienten der Klasse des Prototypen haben, falls für sie die Unterscheidung von `shallowCopy` und `deepCopy` in Smalltalk wichtig ist.

Probleme

- Smalltalk unterschiedet zwischen `shallowCopy` und `deepCopy`, während Java nur die Methode `clone()` definiert.

Abhängigkeiten Klassenmethoden 4.1.1, Objektinitialsierung 4.1.2, Kopieren von Objekten 4.3.1

5.2.4 Singleton

Struktur Das Singleton–Muster stellt sicher, daß lediglich eine einzige Instanz der behandelten Klasse existiert. Außerdem stellt das Muster nur einen einzigen Zugang zu der singulären Instanz zur Verfügung. [ABW98, S.91]

Unterschiede Weil in Smalltalk die Sichtbarkeit von Methoden nicht eingeschränkt werden kann, muß die Smalltalk–Implementierung eine Laufzeit–Lösung verwenden, um ein beliebiges Anlegen einer Instanz der *Singleton*–Klasse zu verhindern. Die Methoden `new` und bei Bedarf `new:` werden überschrieben, um einen Fehler zu erzeugen. Die Klassenmethode zum Erfragen der singulären Instanz erzeugt die Instanz erst beim ersten Aufruf und speichert sie in einer Klassenvariablen. Falls jede Subklasse einer Hierarchie eine eigene Instanz besitzen soll, verwendet Smalltalk eine Klasseninstanzvariable. Um ein Klonieren der singulären Instanz zu verhindern, müssen die Methoden `shallowCopy` und `deepCopy` überschrieben werden.

In Java wird eine Instanz durch Aufruf des Konstruktors erzeugt. Dessen Sichtbarkeit kann eingeschränkt werden. Der Sichbarkeits–Modifikator `private` des Konstruktors verhindert, daß Instanzen der Singleton–Klasse beliebig angelegt werden können. Um ein Klonieren der singulären Instanz zu verhindern, existieren zwei Verfahren: Die Sichtbarkeit von `clone` wird mit `private` überschrieben oder mit `final` markiert (falls keine Subklassen von Singleton benötigt werden). Dadurch kann keine Subklasse der Singleton–Klasse angelegt werden. Dies reicht aus, um das Klonieren der Klasse zu verhindern, da die Klasse `Object` die Sichtbarkeit von `clone` als `protected`

definiert [Eck98, S.718]. In Java sind keine Klasseninstanzvariablen definiert. Falls jede Subklasse eine eigene singuläre Instanz benötigt, muß die Superklasse abstrakte Getter– und Setter–Methoden zum Zugriff auf die singuläre Instanz definieren. Die Subklassen müssen diese abstrakten Methoden implementieren.

Migration Die Migration muß berücksichtigen, daß in Smalltalk das beliebige Anlegen einer neuen Instanz nur zur Laufzeit verhindert werden kann. Der Konstruktor wird in Java als `private` deklariert. Der Code in den Methoden `new` und `new:`, der in Smalltalk das Anlegen einer Instanz verhindern soll, wird somit nicht mehr benötigt. Falls das Klonieren einer singulären Instanz verhindert werden muß, so sind die oben beschriebenen Maßnahmen zu treffen.

Die Definition einer eigenen Instanz in jeder Subklasse bedeutet das Entfernen der Klasseninstanzvariablen und das Erzeugen von abstrakten Getter– und Setter–Methoden in Java. Auf Klienten der Singleton–Klasse hat das keinen Einfluß, weil diese beiden Methoden nur innerhalb der Klasse verwendet werden.

Probleme

- Eine Smalltalk–Klasse kann die Sichtbarkeit ihrer Methoden nicht einschränken.
- In Smalltalk ist kein Sprachkonstrukt zum Anlegen von Instanzen definiert.
- In Java sind keine Klasseninstanzvariabeln definiert.

Abhängigkeiten Klassenvariablen und Klasseninstanzvariablen 4.1.1, Sichtbarkeit von Methoden (4.1.1), Klassenmethoden (4.1.1), Instanzerzeugung (4.1.2), Klassenobjekt (4.1.7), Globale Variable (4.1.1), Fehlermeldungen des Objekt–Framework (4.3.1), Kopieren von Objekten (4.3.1)

5.2.5 Adapter

Struktur Ein *adapter* paßt das Interface eines Nachrichten–Senders (*client*) an das Interface eines Nachrichten–Empfängers (*adaptee*) an. Dadurch können zwei Klassen zusammenarbeiten, die ansonsten eine inkompatible Schnittstelle besitzen. [ABW98, S.105]

Unterschiede In Smalltalk sind drei Varianten dieses Entwurfsmusters gebräuchlich: tailored adapter, message based adapter und block based adapter. *Tailored adapters* werden für jeden Einsatz speziell erstellt. Die Anpassung an die Schnittstelle des Empfängers erfolgt durch feste Kodierung der aufzurufenden Methoden des Empfängers im Adapter. Ein tailored adapter kann in einem anderen Kontext nicht wiederverwendet werden. *Message based adapters* können mit den aufzurufenden Methoden des Empfängers bei ihrer Instanziierung konfiguriert werden. Hierzu verwendet der Adapter einen dynamischen Methodenaufruf. *Block based adapters* werden verwendet, falls das Anpassen der Schnittstellen mehr als ein Umsetzen der Selektoren verlangt. In diesem Fall konfiguriert der Klient den Adapter mit Codeblöcken, die bei Bedarf ausgewertet werden.

In Java sind alle drei Varianten möglich. Die Funktionalität, die durch den block based adapter in Smalltalk abgedeckt wird, ist allerdings die von der Java–Klassenbibliothek am häufigsten eingesetzte Variante dieses Entwurfsmusters. Codeblöcke sind in Java keine Objekte erster Klasse und können daher nicht als Argument für den Adapter verwendet werden. Ein *block based adapter* wird in Java durch ein Adapterobjekt ersetzt, welches ein spezielles Interface implementiert. Dieses Interface spezifiziert das Protokoll zwischen dem Sender und dem Adapterobjekt. Der Sender ruft das Adapterobjekt über die im Interface definierten Methoden auf. Diese Methoden des Adapterobjekts wiederum führen die gewünschten Operationen auf dem Empfänger aus. Falls das Adapterobjekt Zugriff auf nicht–öffentliche Variablen oder Methoden des Empfängers benötigt, muß es als innere Klasse des Empfängers definiert sein (siehe 4.1.3). Die Variante *message based*

adapter muß in Java aufgrund des statischen Typsystems den Typ des Empfängers kennen. Da die Empfängerklasse erst zur Laufzeit bekannt ist, muß an dieser Stelle ein dynamischer Downcast erfolgen. Die aufzurufende Methode wird durch die reflexiven Eigenschaften von Java ermittelt. Ein *tailored adapter* besitzt in Java genau die gleiche Klassenstruktur wie in Smalltalk.

Migration Da ein tailored adapter die aufzurufenden Methoden fest im Programmcode kodiert, ändert sich durch die Migration nichts.

Die Migration eines block based adapters bedeutet die Definition eines Interface und die Kapselung des Codes des Blocks in einer Klasse, die dieses Interface implementiert. Außerdem wird der Code nicht mehr einfach ausgewertet, sondern muß über das Interface aufgerufen werden.

Flanagan definiert einen `UniversalActionListener`, der message based adapters in Java mit Hilfe der reflexiven Eigenschaften realisiert [Fla97, S.222]. Dieser ersetzt nach der Migration den message based adapter von Smalltalk.

Probleme

- Dynamische Methodenaufrufe sind in Java aufwendig.
- Java kennt keine Codeblöcke.
- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.

Abhängigkeiten Abstrakte Klassen (4.1.1), Blöcke (4.1.3), Selektor (4.1.1), Symbol (4.1.1), Dynamischer Methodenaufruf (4.1.1), Ereignisbehandlung (4.3.3)

5.2.6 Bridge

Struktur Dieses Muster trennt den Typ eines Objekts von seiner Implementierung. Dadurch können sich beide unabhängig voneinander verändern. Die *abstraction* bietet lediglich die externe Schnittstelle an. Der *implementor* realisiert die Funktionalität, tritt nach außen aber nur über die abstraction in Erscheinung. [ABW98, S.121]

Unterschiede In Smalltalk werden zur Implementierung dieses Musters zwei Klassen verwendet. Die eine Klasse definiert die abstraction. Sie delegiert sämtliche Methodenaufrufe an die Instanz der implementor–Klasse.

In Java kann dieses Entwurfsmuster auf die gleiche Art implementiert werden. Daneben existiert aber die Möglichkeit, die beiden Klassen miteinander zu verschmelzen. Dazu wird ein Interface definiert, welches die Rolle der abstraction übernimmt. Die implementor–Klasse implementiert dieses Interface. Dadurch fällt die Delegation der Methoden wie in der Smalltalk–Lösung weg. Die implementor–Objekte können nach wie vor gegeneinander ausgetauscht werden. Allerdings ist es in manchen Fällen störend, daß hierdurch die implementor–Klasse permanent an die abstraction gebunden ist [Mar91].

Migration Durch die Migration können beide für Java beschriebene Implementierungsvariante erstellt werden.

Die erste Migrations–Möglichkeit führt zu einer identischen Klassenstruktur wie in Smalltalk. In Java werden die abstrakten Superklassen zusätzlich mit `abstract` markiert, um dem Compiler die Überprüfung zu erleichtern.

Die zweite Migrations–Möglichkeit wandelt die abstraction–Klasse in ein Java–Interface um. Sämtlicher Delegationscode wird entfernt. Die implementor–Klassen müssen sämtliche Methoden implementieren, die in dem neu erstellten Interface definiert sind. Durch dieses Interface fällt ebenfalls die abstrakte implementor–Klasse weg.

Probleme

- Smalltalk kennt keine Interfaces.
- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.

Abhängigkeiten Abstrakte Superklasse (4.1.1), Dynamische Typisierung (4.1.1), Interface (4.1.1), Vererbung (4.1.1)

5.2.7 Composite

Struktur Dieses Muster dient dazu, eine Baumstruktur aufzubauen. Durch die abstrakte *component*–Klasse kann auf die Blätter (*leaf*) und Knoten (*composite*) über die selbe Schnittstelle zugegriffen werden. [ABW98, S.137]

Unterschiede In Smalltalk definiert die component–Klasse lediglich ein Rumpf–Protokoll. Dieses enthält ausschließlich solche Methoden, die von allen Elementen der component–Hierarchie verstanden werden. Das Protokoll zum Zugriff auf die Knoten ist ausschließlich in der composite–Klasse definiert. Einem Klienten wird der Zugriff auf dieses zusätzliche Protokoll durch die dynamische Typisierung ermöglicht.

In statisch getypten Programmiersprachen müßten aufgrund der Typsicherheit auch die Methoden des composite–Protokolls in der abstrakten Superklasse component definiert sein. Dies ist der Fall bei der Java–Implementierung dieses Musters. Daneben bietet Java eine weitere Möglichkeit zur Implementierung einer Baumstruktur. Das in der Klassenbibliothek vorhandene `TreeNode`–Interface definiert alle Methoden, die für einen Knoten in einer Baumstruktur benötigt werden. Damit müssen Blätter und Knoten, die dieses Interface implementieren, nicht mehr von der gleichen Superklasse erben. Die Klassenbibliothek von Java definiert die Klasse `DefaultTreeNode` als Standard–Implementierung des `TreeNode`–Interfaces. Durch Vererbung können beliebige Baumstrukturen aufgebaut werden.

Migration Das Ergebnis der Migration kann eine der beiden möglichen Java–Varianten sein.

Bei der ersten Möglichkeit werden die Klassen des Composite–Musters fast unverändert verwendet. Lediglich die Methoden zur Realisierung des composite–Protokolls müssen in die component–Klasse verschoben werden. Dieses Verschieben stellt die Überprüfbarkeit der Typsicherheit durch den Compiler wieder her. Ein Downcast in den Klienten des Composite–Musters wird somit verhindert.

Bei der zweiten Möglichkeit wird das oben beschriebene `TreeNode`–Interface verwendet. Die leaf– und die composite–Klassen müssen dieses Interface implementieren. Dadurch fällt die Notwendigkeit einer gemeinsamen component–Klasse weg. Der Wegfall dieser Einschränkung ermöglicht größere Flexibilität. Insbesondere wenn die Baumstruktur mit dem `JTree`–Widget zusammenarbeiten können soll, ist die Implementierung des `TreeNode`–Interface durch die Knoten und Blätter verlangt.

Probleme

- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.
- In Smalltalk sind keine Interfaces definiert.

Abhängigkeiten Abstrakte Superklasse (4.1.1), Dynamische Typisierung (4.1.1), `TreeNode`–Interface (4.3.5), Interface (4.1.1)

5.2.8 Proxy

Struktur Ein *Proxy* ist ein Ersatz oder ein Platzhalter für ein anderes Objekt (*real subject*). Dieses Entwurfsmuster ermöglicht das Überwachen des Zugriffs auf das real subject. [ABW98, S.213]

Unterschiede Die klassische Implementierung dieses Musters besteht in der Definition einer gemeinsamen Superklasse für die Proxy– und die RealSubject–Klassen. Diese Superklasse definiert das gemeinsame Protokoll, über das ein Klient mit dem Proxy oder dem real subject kommuniziert. Ein Klient kann somit nicht unterscheiden, ob er mit dem Proxy oder dem RealSubject kommuniziert. Die Mechanismen des dynamischen Methodenaufrufs und der dynamischen Typisierung eröffnen in Smalltalk eine weitere Möglichkeit zur Implementierung dieses Musters. Hierzu wird eine generische Proxy-Klasse definiert, die als Proxy für jede beliebige RealSubject-Klasse dienen kann, ohne daß der Proxy und das real subject eine gemeinsame Superklasse haben müssen. Ein generischer Proxy überschreibt die Methode `doesNotUnderstand`, um über alle Nachrichten informiert zu werden, die er nicht versteht (Diese Fehlermethode wird immer dann aufgerufen, wenn ein Objekt eine erhaltene Nachricht nicht durch eine Methode implementiert). Die nicht verstandene Nachricht leitet er dann an das real subject weiter. Zur Vereinfachung der Implementierung erbt ein Proxy–Objekt oft von `nil` (Da ein solches Objekt nicht von der Klasse `Object` abgeleitet ist, erbt es keine Methoden). Um den Proxy durch das real subject zu ersetzen, kann die Methode `become:` verwendet werden (Objekt–Mutation). Dadurch werden alle Referenzen zum Proxy auf das real subject umgesetzt.

Die Java–Lösung orientiert sich an der klassischen Implementierung. Jede RealSubject–Klasse erhält eine eigene Proxy–Klasse, wobei über eine gemeinsame Superklasse das gemeinsame Protokoll definiert wird. In Java bleibt das Proxy–Objekt auch nach dem Zugriff auf das real subject erhalten — eine Objekt–Mutation findet nicht statt, weil sie in Java nicht definiert ist.

Migration Die Migration der klassischen Implementierungsvariante ist einfach. Die Klassenstruktur ist in Java identisch. Da die Proxy–Klasse und die RealSubject–Klasse eine gemeinsame Superklasse haben, sind sie über das gleiche Protokoll ansprechbar.

In Java kann aufgrund des statischen Typsystems kein generischer Proxy implementiert werden. Daher muß jedes Verwenden der generischen Proxy–Klasse durch eine klassisch implementierte Proxy–Klasse ersetzt werden. Jede RealSubject–Klasse benötigt eine abstrakte Subject–Klasse, die das gemeinsame Protokoll der Proxy–Klasse und der RealSubject–Klasse definiert. Dadurch nimmt die Migration einen großen Einfluß auf die Klassenstruktur und auf die Instanziierung des Proxy. Die Instanziierung der generischen Proxy–Klasse und anschließende Parametrisierung mit dem real subject wird durch ein direktes Instanziieren der spezialisierten Proxy–Klasse ersetzt.

Der Aufruf zur Objekt–Mutation `become:` muß ersatzlos entfallen. Dies kann dazu führen, daß sich die verwendenden Klassen darüber bewußt sein müssen, daß es sich „nur“ um ein Proxy–Objekt handelt.

Probleme

- Dynamisches Typsystem in Smalltalk im Gegensatz zum statischen Typsystem in Java.
- In Java ist keine Objekt–Mutation definiert.
- Ein Objekt kann in Java nicht von `null` erben.

Abhängigkeiten Objekt–Mutation (4.1.2), Dynamische Typisierung (4.1.1), Fehlermethoden von `Object` (4.1.2), Null–Objekt (4.1.2)

5.2.9 Command

Struktur Dieses Muster dient dem Kapseln von Operationen als Objekt. Dadurch können Klienten mit verschiedenen Command–Objekten parametrisiert werden und ein Mechanismus zum Widerrufen von Operationen wird ermöglicht. Typischerweise wird dieses Muster in der graphischen Benutzeroberfläche verwendet, um Aktionen des Benutzers als *Command* zu kapseln. [ABW98, S.245]

Unterschiede Bei Verwendung in einer graphischen Oberfläche registriert die Applikation ein Command–Objekt bei einem Widget. Dieses Widget ruft dieses Command–Objekt auf, sobald es vom Benutzer aktiviert wird. An dieser Stelle beschreiben wir drei mögliche Implementierungsvarianten dieses Musters. Die *erste Variante* verwendet eine abstrakte Command–Klasse, die das gemeinsame Protokoll sämtlicher Commands definiert. Ein konkretes Command erbt von dieser Superklasse. Die konkrete Command–Klasse enthält den hart kodierten Methodenaufruf, um die Methode aufzurufen, die die gekapselte Operation realisiert. Bei dieser Variante muß für jede gekapselte Operation eine eigene Klasse angelegt werden. Die *zweite Variante* verwendet eine generische Command–Klasse. Diese wird bei ihrer Instanziierung mit der aufzurufenden Methode parametrisiert. IBM Smalltalk definiert die Klassen `Message` und `DirectedMessage`, die Implementierungen dieses Entwurfsmusters sind. Die *dritte Variante* ist keine direkte Implementierung des Command–Musters, weil auf eine Command–Instanz verzichtet wird. Das Ereignismodell erlaubt es einem Klienten, direkt bei einer Ereignisquelle eine Behandlermethode für ein bestimmtes Ereignis zu registrieren. Bei Eintreten des Ereignisses ruft das Widget diese Behandlermethode direkt auf, ohne ein Command–Objekt zu verwenden. Auf diese Art können bei einem Widget mehrere Behandler registriert werden im Gegensatz zu dem einen Command–Objekt der beiden ersten Varianten.

Die Klassenbibliothek von Java kennt keine direkte Implementierung des Command–Musters, aber dennoch sind alle drei Varianten in Java ebenfalls möglich. Die *erste Variante* benötigt auch in Java eine abstrakte Superklasse, um das gemeinsame Protokoll zu definieren. Der Methodenaufruf der von der Operation aufzurufenden Methode ist wieder hart in der konkreten Command–Klasse kodiert. Die *zweite Variante* führt einen dynamischen Methodenaufruf der Methode aus, die bei der Parametrisierung gesetzt wurde. Dies wird durch die reflexiven Eigenschaften von Java ermöglicht. Diese Variante muß allerdings selbst implementiert werden, weil die Klassenbibliothek von Java keine Standardimplementierung hierfür anbietet. Die *dritte Variante* verwendet das Ereignismodell von Java. Hiermit kann eine Applikation ein Behandlerobjekt bei einer Ereignisquelle registrieren. Dazu muß das Behandlerobjekt ein Listener–Interface implementieren über welches die Ereignisquelle mit dem Behandlerobjekt kommuniziert. In Java kann, wie in Smalltalk, eine beliebige Anzahl an Behandlern bei einer Ereignisquelle registriert werden. Der große Vorteil der Java–Implementierung ist, daß die Behandler selbst wieder Objekte sind (nicht „nur“ Methoden) und damit als vollwertige Command–Objekte verwendet werden können (beispielsweise in einer Undo–Redo–Liste).

Migration Durch die Migration der *ersten Variante* ändert sich die Klassenstruktur nicht. Ein Klient eines Command–Objekts verwendet das in der Superklasse definierte Protokoll.

Bei der *zweiten Variante* müsste eine generische Command–Klasse definiert werden. Instanzen dieser Klasse müssen einen generischen Methodenaufruf ausführen. Das dazu benötigte Methodenobjekte der Methode, die durch die Operation aufgerufen werden soll, muß durch Reflexion ermittelt werden. Dadurch geht allerdings die Überprüfbarkeit der Typisierung durch den Compiler verloren.

Bei der dritten Variante wird die Verwendung des jeweiligen Ereignismodells von Smalltalk nach Java migriert. Dies wird in 4.3.3 und 5.2.12 behandelt. In Java ist diese Implementierung

vorzuziehen, weil sie die Vorteile von Command–Objekten damit verbindet, mehrere Behandler bei einer einzigen Ereignisquelle registrieren zu können.

Probleme

- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.
- Dynamischer Methodenaufruf ist in Java aufwendig.
- In der Java–Klassenbibliothek existiert kein Gegenstück zu `Message` oder `Directed Message`.

Abhängigkeiten Dynamischer Methodenaufruf (4.1.1), Ereignismodell (4.3.3), Widgets (4.3.5), Dynamische Typisierung (4.1.1)

5.2.10 Iterator

Struktur Dieses Entwurfsmuster beschreibt einen Mechanismus, mit dem auf die Elemente einer Aggregation in sequentieller Weise zugegriffen werden kann, ohne die zugrundeliegende Datenstruktur offenlegen zu müssen. Ein `Iterator` verfügt über ein Protokoll zu den Elementen einer Aggregation; allerdings kann immer nur auf ein einziges Element zugegriffen werden. Der Aufbau der Aggregation bleibt den Klienten verborgen. [ABW98, S.273]

Unterschiede Die Klassenbibliothek von Smalltalk kennt zwei Arten von Iteratoren. Die Collection–Klassen definieren interne Iteratoren, die jedes Element einer Collection ansprechen. Der interne Iterator wird durch den Aufruf von `do:` angestoßen, dem ein Codeblock mitgegeben wird. Dieser Codeblock wird über allen Elementen der Aggregation ausgeführt. Zusätzlich definieren die Collection–Klassen Filter–Iteratoren. Diese filtern gewisse Elemente aus der Aggregation heraus. Sie sind ebenfalls als interne Iteratoren gestaltet. Innerhalb der Iteration kann kein Element in die Collection eingefügt oder gelöscht werden. Mit den `Stream`–Klassen definiert Smalltalk externe Iteratoren. Diese bieten mehr Flexibilität, weil der Ablauf der Iteration genauer kontrolliert werden und die Stream–Instanz an verschiedenen Stellen im Klienten verwendet werden kann. Streams erlauben das Einfügen und Löschen von Elementen während der Iteration.

Die Java–Klassenbibliothek definiert nur die externen Iteratoren `Enumeration`, `Iterator` und `ListIterator`. Eine Aggregation erzeugt einen Iterator für ihre Elemente durch einen Methodenaufruf. Die drei verschiedenen Iterator–Interfaces unterscheiden sich hinsichtlich der Behälter–Klassen über deren Elemente sie iterieren können und hinsichtlich der Änderbarkeit der Elemente. Interne Iteratoren oder Filter–Iteratoren sind nicht vorhanden. Neben den Iteratoren existiert in Java ein anderer Mechanismus zum Zugriff auf die Elemente eines Array. Primitive Elemente können nur in einem Array gespeichert werden. Auf diese Elemente des Arrays wird über den numerischen Index zugegriffen. Dieser Mechanismus ist in der Sprache definiert und wird nicht durch die Klassenbibliothek unterstützt.

Migration Interne Iteratoren werden in Java durch externe Iteratoren ersetzt. Dies bedeutet für die Klienten, daß sie nun selbst eine Schleife programmieren müssen, die über alle Elemente iteriert. Der Schleifenrumpf besteht aus dem Code der in Smalltalk noch in Form eines Codeblockes an den internen Iterator übergeben wurde. Das Verhalten von Filteriteratoren muß in Java durch den Klienten nachgebildet werden. In Smalltalk ist das Idiom „Enumeration Method“ üblich (siehe 5.1.13). Dabei erhält die Iteratormethode den Codeblock nicht direkt vom Klienten, sondern durch Delegation derjenigen Klasse, die die Collection zur Speicherung ihrer Elemente verwendet. Diese Delegation muß bei der Migration über sämtliche Delegationsschritte zurück bis zum ersten Aufruf verfolgt werden. Dieser erste Aufruf ist auf oben beschriebene Weise zu migrieren.

Externe Iteratoren können lesend oder schreibend auf eine Collection zugreifen bzw. absolut positioniert werden (falls die Collection dies zuläßt). Die Migration von externen Iteratoren kann unter Umständen einiges an Umstrukturierung im Klienten bedeuten, weil in Java lediglich ein `ListIterator` beliebig lesend und schreibend zugreifen bzw. beliebig positioniert werden kann.

Iteratoren können in Java nur mit Instanzen von `Object` arbeiten. D.h. bevor ein von einem Iterator erhaltenes Object verwendet werden kann, muß erst ein Downcast auf den gewünschten Typ erfolgen. Dies geschieht im Schleifenrumpf des Klienten.

Falls eine Collection nur solche Elemente enthält, deren Entsprechungen in Java primitive Typen haben, können die Elemente in einem Array gespeichert werden. Auf die Elemente eines Arrays wird über den Index zugegriffen. Dadurch ist auch in Java ein beliebiges Lesen, Schreiben und Positionieren möglich.

Probleme

- Java kennt keine internen Iteratoren.
- Durch die freiere Positionierbarkeit von Stream–Instanzen kann der Umfang der Umstrukturierung von Klientencode nicht abgeschätzt werden.
- Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.
- Java unterscheidet primitive Typen von Referenztypen.

Abhängigkeiten Collections (4.2.1), Ströme (4.2.3), Dynamische Typisierung (4.1.1), Primitive Typen und Referenztypen (4.1.1)

5.2.11 Memento

Struktur Ziel dieses Musters ist es, den internen Zustand eines Objekts zu sichern, um es später in diesen Zustand zurückversetzen zu können. Ein *Memento*–Objekt speichert den internen Zustand des *Originator*. Der *Caretaker* verwaltet die verschiedenen Mementos und veranlaßt gegebenenfalls den Originator, den Zustand des gespeicherten Mementos anzunehmen. [ABW98, S.297]

Unterschiede In Smalltalk kann die Sichtbarkeit einer Methode nicht eingeschränkt werden. Daher ist die Beschränkung des Zugriffs auf das Memento ausschließlich durch den Originator schwierig. In Smalltalk sind die Originator– und die Memento–Klassen daher oft die selben. Der Originator kopiert sich selbst und liefert diese Kopie als Memento. Unter Umständen muß aus diesem Grund in den Kopierprozeß eingegriffen werden (`copy` oder `deepCopy`).

Java erlaubt zwei Arten der Implementierung des Memento–Musters. Sichtbarkeitsregeln wie `friend` in C++ erreicht Java durch das Package–Konzept. Damit ist eine Implementierung möglich, wie sie von Gamma et al. vorgeschlagen wird [GHJV95, S.283]. Das Memento kann auf den internen Zustand des Originator zugreifen, während das öffentliche Interface diesen Zugriff nicht erlaubt. Zwei Klassen im gleichen Package haben eine zu `friend` vergleichbare Sichtbarkeit. Das Konzept der inneren Klassen erlaubt in Java eine noch viel schlüssigere Implementierung des Memento–Musters. Das Memento wird als innere Klasse des Originators gestaltet. Ein Objekt dieser Klasse kann also auf den vollständigen inneren Zustand des Originator zugreifen, ohne daß dieser Zugriff von außen möglich sein muß. Der innere Zustand des Memento–Objekts kann vor dem Zugriff von außen geschätzt werden. Um dieses innere Objekt dem Caretaker gegenüber eindeutig zu identifizieren, kann es mit einem Marker–Interface ausgestattet sein. Ein solches Interface definiert keine Methoden, sondern dient lediglich der Markierung von Klassen, die dieses Interface implementieren (Bsp.: `Runnable` in `java.lang`).

Migration Die Smalltalk–Implementierung geht in den meisten Fällen von einer Kombination des Originators und des Memento aus. Die Migration muß diese beiden voneinander trennen. Da in Java die zweite oben beschriebene Alternative bevorzugt wird, muß der Memento–Anteil des Originator als innere Klasse gestaltet werden. Das Memento–Protokoll muß dann eine Instanz dieser inneren Klasse erzeugen bzw. zum Setzen des internen Zustandes akzeptieren. Die Definition eines Memento–Marker–Interface ist nur dann nötig, wenn das Memento von außen als solches erkannt werden muß. Dies hängt vom Caretaker ab. Die Modifikation des Kopierprozesses des Originators ist nochmals zu überprüfen, da die Smalltalk–Lösung hier evtl. eingegriffen hat.

Probleme

- Die Sichtbarkeit von Methoden kann in Smalltalk nicht eingeschränkt werden.

Abhängigkeiten Sichtbarkeit von Variablen (4.1.1), Sichtbarkeit von Methoden (4.1.1), Kopieren von Objekten (4.3.1), Interfaces (4.1.1)

5.2.12 Observer

Struktur Dieses Entwurfsmuster definiert eine eins–zu–n–Abhängigkeitsbeziehung. Ein *Observer* kann sich bei einem *Subjekt* registrieren, um über alle Änderungen im Subjekt automatisch unterrichtet zu werden. Bei Eintreten der Änderung informiert das Subjekt alle registrierten Observer über diese Änderung. [ABW98, S.305]

Unterschiede Dieses Muster tritt in Smalltalk in zwei Varianten auf: als Abhängigkeits–Mechanismus (4.3.2) und als Ereignismodell (4.3.3). Der Abhängigkeitsmechanismus existierte schon in Smalltalk–80. Ein Objekt kann sich bei einem anderen als Abhängigem eintragen (`addDependent:`). Sobald sich der Zustand des beobachteten Objekts ändert (`changed`), werden alle registrierten Objekte benachrichtigt (`update`). Der Nachteil ist, daß bei *jedem* Ändern des Zustandes *alle* Observer benachrichtigt werden. Das Ereignismodell erlaubt eine genauere Kontrolle über die Benachrichtigung von beobachtenden Objekten über ein Ereignis. Es klassifiziert die möglichen Änderungen als Ereignisse. Ein Observer registriert sich für ein bestimmtes Ereignis und wird durch Aufruf einer zu konfigurierenden Methode über das Ereignis benachrichtigt. Ein Observer wird nur über die Ereignisse informiert, für die er sich registriert hat.

In Java sind beide Mechanismen definiert. Ein Observer, der den Abhängigkeitsmechanismus nutzen will, muß aufgrund des statischen Typsystems in Java das Interface `Observer` implementieren. Dann kann er sich bei einem Subjekt registrieren. Ein Subjekt muß von der Klasse `Observable` erben, die die Verwaltung der Observer besorgt. Das Ereignismodell von Java klassifiziert die Änderungen ebenfalls nach Ereignissen. Ein Klient kann ein Behandler–Objekt für ein Ereignis registrieren. Dieses Behandlerobjekt muß das spezifische Listener–Interface dieses Ereignisses implementieren. Die Ereignisquelle informiert alle Behandlerobjekte über die in diesem Interface definierten Methoden über das Eintreten des Ereignisses. Der Methodenname kann im Gegensatz zu Smalltalk nicht frei gewählt werden, sondern ist im Listener–Interface definiert.

Migration Bei der Migration muß zwischen beiden Mechanismen unterschieden werden.

Der Abhängigkeitsmechanismus kann unter Verwendung der entsprechenden Java–Klassen direkt nachgebildet werden. Schwierig wird die Migration, falls das Subjekt nicht von `Object` erbt, weil in Java das Subjekt eine Subklassen von `Observable` sein muß. Im einfachsten Fall kann die `Observable`–Klassen in die Vererbungskette eingehängt werden. Ansonsten ist eine Umstrukturierung der Klassen nötig, die aber vom Einzelfall abhängt.

Die Migration einer Verwendung des Ereignismodells kann eine größere Änderung bedeuten. In Smalltalk kann eine beliebige Methode eines beliebigen Objekts als Ereignisbehandler registriert werden. In Java dagegen muß das zu registrierende Objekt das passende Listener–Interface implementieren. Ein Listener–Interface faßt mehrere Methoden zusammen, die das Behandler–Objekt über das Eintreten des Ereignisses informieren. Die Behandlermethoden eines Interfaces werden in Java gerne in einem eigenen Objekt zusammengefaßt. Falls die Behandlermethoden eines Listener–Interfaces in Smalltalk über mehrere Objekte verteilt sind, müssen sie in Java in einem Objekt zusammengefaßt werden. Dies kann umfangreiche Verschiebungen von Methoden zur Folge haben, was sich in der Folge auf die Gestaltung der externen Schnittstellen der beteiligten Klassen auswirken kann. Häufig muß ein Behandlerobjekt auf den internen Zustand des registrierenden Objekts zugreifen können. Dieses Recht kann das Behandlerobjekt dadurch erlangen, daß es als Instanz einer inneren Klasse definiert wird. Die Instanziierung eines solchen Objekts bedeutet für den Klienten, daß er bei Registrieren zunächst eine solche innere Klasse definieren muß. Zudem sind durch die Festlegung der Namen im Interface normalerweise Umbenennungen der Methoden notwendig. Diese Umbenennung muß auch in allen anderen Klienten der Methode erfolgen. Diese sind aufgrund des dynamischen Typsystems und aufgrund von dynamischen Methodenaufrufen nicht immer leicht festzustellen.

Probleme

- Dynamischer Methodenaufruf ist in Java aufwendig.
- Der Abhängigkeits–Mechanismus ist nicht im `Object`–Framework enthalten. Daher muß ein Subjekt von der Klasse `Observable` erben.
- Dynamisches Typsystem in Smalltalk im Gegensatz zum statischen Typsystem in Java.
- Listener–Interface schreibt Namen der Behandlermethoden in Java vor.

Abhängigkeiten Abhängigkeits–Mechanismus (4.3.2), Ereignismodell (4.3.3), Dynamischer Methodenaufruf (4.1.1), Innere Klassen (4.1.1), Dynamische Typisierung (4.1.1)

5.2.13 Template Method

Struktur Dieses Muster beschreibt, wie lediglich das Skelett eines Algorithmus definiert werden kann. Es bleibt den konkreten Subklassen vorbehalten, die noch fehlenden Schritte auszufüllen. Dieses Ausfüllen erlaubt Subklassen eine Modifikation des Algorithmus, ohne seine Struktur zu ändern.

Das Template–Muster ist der Schlüssel zur sauberen Verwendung der Vererbung. Wiederverwendbarer Code kommt in die Superklasse, anwendungsspezifischer in die Subklasse. [ABW98, S.355]

Unterschiede Die Implementierung unterscheidet zwischen Template–Methoden, konkreten Methoden, abstrakten Methoden und Hooks. Template–Methoden sind die strukturellen Bestandteile des Algorithmus. Sie sind in der abstrakten Superklasse implementiert und verwenden die anderen drei Methodenarten zur Realisierung des Algorithmus. Normalerweise übernehmen Subklassen die Template–Methoden unverändert, können sie aber erweitern. Konkrete Methoden realisieren die Bestandteile des Algorithmus, die unveränderlich sind. Daher sind sie in der Superklasse implementiert und dürfen in einer Subklasse nicht überschrieben werden. Abstrakte Methoden sind in der Superklasse lediglich definiert und müssen in einer konkreten Subklasse implementiert werden. Hooks sind in der Superklasse definierte und implementierte Methoden, die bei Bedarf überschrieben werden können. Die Standardimplementierung ist meist sehr einfach. Solche Methoden dienen der Erweiterung des Algorithmus.

Die Unterschiede zwischen der Smalltalk– und der Java–Implementierung sind nicht struktureller Art. Die verwendeten Klassen und die Beziehungen zwischen den Klassen sind identisch. Allerdings können in Java die verschiedenen Methodenarten klarer voneinander getrennt werden. Abstrakte Methoden können mit `abstract` gekennzeichnet werden. Damit ist automatisch klar, daß die Superklasse ebenfalls abstrakt ist. Konkrete Methoden in der Superklasse sind nicht zum Überschreiben vorgesehen, weshalb sie in Java mit `final` markiert werden. Zudem können durch Sichtbarkeits–Modifikatoren die Methoden, die nicht zur externen Schnittstelle des Algorithmus gehören, mit `private` bzw. `protected` deklariert werden. Diese Modifikatoren ermöglichen dem Compiler, die korrekte Verwendung der Methoden in den Subklassen zu überprüfen.

Migration Die Migration besteht aus dem Einfügen oben genannter Modifikatoren. Dadurch wird in bestehenden Subklassen evtl. eine inkorrekte Verwendung des Protokolls festgestellt. Falls diese falsche Verwendung auf strukturelle Mängel in der Superklasse zurückzuführen ist, muß diese refaktoriert werden.

Probleme

- In Smalltalk sind die Modifikatoren `abstract` und `final` und die Sichtbarkeitsmodifikatoren nicht definiert.
- Dynamische Typisierung in Smalltalk im Gegensatz zur statischen in Java.

Abhängigkeiten Vererbung (4.1.1), Objekt–Framework (4.3.1), Methoden–Sichtbarkeit (4.1.1), Abstrakte Klassen (4.1.1)

5.2.14 Visitor

Struktur Dieses Entwurfsmuster erlaubt es, die Operationen auf den Elementen einer Objektstruktur getrennt von diesen Elementen zu definieren. Durch dieses Muster kann eine neue Operation eingefügt werden, ohne daß die Elementklassen dazu geändert werden müssen.

Ein *Visitor* sucht jedes *Element* einer Ojektstruktur auf (`acceptVisitor:`). Das Element wiederrum ruft eine auf das Element spezialisierte Methode im Visitor auf (`visitConcreteElement:`), die die Operation auf dem Element ausführt. [ABW98, S.371]

Unterschiede Durch dieses Muster wird in jedem Element der Objektstruktur die Methode aufgerufen, die den Visitor akzeptiert. Diese Methode enthält in der klassischen Implementierungsvariante dieses Musters den hart kodierten Methodenaufruf des Visitors, um dieses Element zu verarbeiten. Durch die Möglichkeit, einen aufzurufenden Selektorennamen dynamisch zu erzeugen, eröffnet sich in Smalltalk eine weitere Möglichkeit der Implementierung. Nicht mehr jede konkrete Elementklasse der Objektstruktur enthält die Methode zum Akzeptieren des Visitor, sondern nur noch deren abstrakte Superklasse. Diese setzt den Selektor der im Visitor aufzurufenden Methode dynamisch aus der Klasse des besuchten Elements zusammen und ruft dann den Visitor auf. Das erspart den Elementenklasen, eine `acceptVisitor`–Methode zu implementieren.

Die Java–Version orientiert sich eher an der klassischen Variante. Ein dynamisches Zusammensetzen des Methodennamens ist zwar über die reflexiven Eigenschaften von Java möglich, aber in zweierlei Hinsicht abzulehnen: Das Verwenden der Reflection–API ist aufwendig und durch dessen Verwendung wird die statische Typüberprüfung unterlaufen.

Migration Die Migration ist davon abhängig, welche Implementierungsvariante in Smalltalk vorliegt. Die klassische Variante ist eins–zu–eins zu migrieren. In Java kann der Compiler dabei unterstützt werden, die Vollständigkeit der `acceptVisitor`–Methoden festzustellen, indem sie in der

Superklasse als abstrakt markiert werden. Der Compiler moniert dann das Fehlen einer Implementierung.

Die Migration der zweiten Variante bedeutet entweder ein Nachbilden des dynamischen Methodenaufrufs in Java über die reflexiven Eigenschaften von Java oder ein Umschreiben der Implementierung auf die klassische Variante. Die klassische Variante hat den Vorteil der einfacheren Implementierung, der größeren Performanz und der Überprüfbarkeit der Typsicherheit zur Übersetzungszeit. In diesem Fall entfällt die Standardimplementierung von `acceptVisitor` in der Superklasse. Sie wird stattdessen als abstrakt markiert und jede Elementklasse muß eine Implementierung dafür angeben. Das Nachbilden des dynamischen Methodenaufrufs beinhaltet ein Ermitteln der passenden `Method`–Instanz der aufzurufenden Methode über das Klassenobjekt des Visitors mit Hilfe von `getMessage()`. Hierzu muß dynamisch die Klasse des besuchten Elements ermittelt werden. Diese wird im Namen der zu ermittelnden Methode und in deren Argumenten kodiert.

Probleme

- Dynamischer Methodenaufruf ist in Java aufwendig.
- Statische Typisierung in Smalltalk im Gegensatz zu dynamischer in Java.

Abhängigkeiten Dynamischer Methodenaufruf (4.1.1), Dynamische Typisierung (4.1.1), Abstrakte Klassen (4.1.1)

5.3 Migration von Architekturmustern

Architekturmuster dienen der Strukturierung der Applikationsarchitektur. Sie üben ihren Einfluß durch ihre Realisierung in der Klassenbibliothek aus. In realen Systemen kommen Architekturmuster meist als Kombination verschiedener Architekturmuster vor.

> While it is important to understand the individual nature of each of these styles, most systems typically involve some combination of several styles. [SG96, S.32]

Auch die in dieser Arbeit betrachteten Plattformen vermischen die untersuchten Architekturmuster miteinander.

5.3.1 Virtuelle Maschine, Interpreter

Struktur Eine virtuelle Maschine simuliert auf einem Rechner eine nicht als Hardware vorhandene Maschine. Diese interpretiert die Programme, die die simulierte Maschine als Ausführungsplattform verwenden.

In Smalltalk und Java werden Programme in einen Zwischencode übersetzt. Die virtuelle Maschine beider Sprachen interpretiert diesen Zwischencode und ruft die entsprechende Funktionalität des darunterliegenden Betriebssystems auf. [SG96, S.27]

Unterschiede In Smalltalk ist die virtuelle Maschine die Basis für die gesamte Entwicklungsumgebung. Browser, Debugger, Editoren usw. sind alle in Smalltalk geschrieben. Die virtuelle Maschine ist die Ausführungsbasis für alle Betriebssysteme, auf denen eine Smalltalk–Applikation ausgeführt werden soll. Sie kapselt die hardwareabhängigen Eigenschaften und ermöglicht daher die Portierung von Smalltalk–Applikationen. Normalerweise enthält eine Smalltalk–Applikation die virtuelle Maschine, die zu ihrem Ablauf notwendig ist. Die Applikation wird somit unabhängig von der Installation auf der Zielmaschine. Um externe Funktionen aufzurufen, die direkt auf der Hardware arbeiten, ermöglicht die VM von IBM Smalltalk die Definition einer neuen Primitive bzw. den Aufruf von Funktionen in einer dynamisch ladbaren Unterprogrammbibliothek (beispielsweise eine dynamic link library oder shared library). Der Grund für externe Funktionen kann die Notwendigkeit zum Zugriff auf die Hardware, die größere Performanz von direkt ausführbarem Code oder die Nutzung eines vorhandenen Programms sein. Allerdings geht damit die Portabilität der Applikation verloren.

In Java hat die virtuelle Maschine (JVM) die gleiche Funktion: Durch Kapseln der hardwareabhängigen Eigenschaften soll die Portabilität von Java–Anwendungen gewährleistet werden. Allerdings ist im Gegensatz zu Smalltalk die JVM nur in Ausnahmefällen die Basis für die Entwicklungsumgebung. Normalerweise läuft nur die fertig übersetzte Applikation auf der JVM. Ein Java–Programm verläßt sich normalerweise auf die Installation einer virtuellen Maschine auf dem Zielrechner. Das fertige Programm enthält ausschließlich den übersetzten Bytecode. Die Erweiterung um externe Funktionen erfolgt bei der JVM etwas anders als in Smalltalk. Eine Methode kann im Java–Source–Code mit `native` markiert werden. Beim Aufruf dieser Methode wird die entsprechende Funktion in einer dynamisch ladbaren Bibliothek gesucht, die vorher bei der JVM registriert werden muß. Auch in Java geht hierdurch die Portierbarkeit verloren.

Migration Auf die Migration hat die virtuelle Maschine nur einen geringen Einfluß. Die Schwierigkeiten betreffen vor allem die Version der installierten virtuellen Maschine für Java–Programme und die Verwendung von externen Funktionen.

Da ein kompiliertes Java–Programm von der installierten virtuellen Maschine abhängig ist, kann aufgrund von häufigen Versionswechseln eine veraltete JVM auf dem Zielrechner vorhanden

sein. Insbesondere die Ausführbarkeit von Applets innerhalb eines Webbrowsers ist extrem von der Version des verwendeten Webbrowsers abhängig.

Um externen Code migrieren zu können, muß in Java ein Wrapper in C geschrieben werden, der die Methoden in der dynamisch ladbaren Bibliothek aufruft. Um applikationsspezifische Primitive verwenden zu können, muß der entsprechende Programmcode mit Hilfe des Java Native Interface–API (JNI) komplett neu geschrieben werden, weil sich der Zugriff auf die Laufzeitobjekte von C aus in Smalltalk und Java sehr unterscheidet.

Probleme

- Unterschiedlicher Zugriff auf Laufzeitobjekte aus einer C–Funktion heraus.
- Smalltalk liefert die virtuelle Maschine mit der Applikation aus, während sich ein Java–Programm auf die Installation verläßt.

Abhängigkeiten Die virtuelle Maschine ist die Basis für die Interpretation des erzeugten Zwischencodes. Sie verwendet die Elemente der darunterliegenden Ausführungsplattform.

5.3.2 Reflexion

Struktur Die reflexiven Eigenschaften einer Sprache ermöglichen einer Applikation den Zugriff auf Informationen über sich selbst. Smalltalk wie Java haben reflexive Eigenschaften, die sich jedoch deutlich unterscheiden. [BMR+96, S.123]

Unterschiede Für jede definierte Klasse in Smalltalk existiert eine Metaklasse. Die Klasse selbst ist die einzige Instanz dieser Metaklasse. Durch Zugriff auf die Metaklasse durch das Meta–Objekt–Protokoll können die verfügbaren Klassen, sowie deren Methoden und Variablen ermittelt werden. Darüber hinaus können alle diese Eigenschaften zur Laufzeit *verändert* werden. Die Werkzeuge des Entwicklungssystems verwenden genau diese Eigenschaften von Smalltalk um neue Klassen zu definieren. Durch das Meta–Objekt–Protokoll kann sogar das Verhalten der Sprache selbst beeinflußt werden.

Die Reflection–API in Java dient lediglich der *Abfrage* von Metainformationen. Mit Hilfe des Klassenobjekts einer Klasse können die vorhandenen Konstruktoren, Methoden und Variablen abgefragt werden. Ebenso ist der dynamische Aufruf von Methoden möglich. Ein Ändern des Systemverhaltens oder die Definition von neuen Klassen, Methoden oder Variablen ist über das Reflection–API nicht möglich. Bei Bedarf kann eine neue Klasse vom Java–Compiler erzeugt werden, um sie anschließend mit Hilfe des `ClassLoader` bei der virtuellen Maschine zu registrieren.

Die Methoden zum Abfragen der Meta–Information sind über mehrere Metaklassen verteilt, die selbst wieder eine Hierarchie bilden, die parallel zur gewöhnlichen Klassenhierarchie verläuft. Die Abfrage von Informationen über eine Klasse geschieht in Java über das Klassenobjekt, welches die in `java.lang.reflect` definierten Klassen verwendet.

Migration Die Verwendung des Meta–Objekt–Protokolls kann enormen Einfluß auf die Migrierbarkeit einer Smalltalk–Anwendung haben. Sobald nicht nur Information über die vorhandenen Klassen abgefragt, sondern die Struktur oder das Verhalten des Laufzeitsystems verändert wird, ist eine Neuentwicklung in Java nicht zu vermeiden.

Probleme

- Sehr mächtiges Meta-Objekt-Protokoll in Smalltalk im Vergleich zur Reflection–API, die lediglich der Abfrage von Informationen dient.

Abhängigkeiten Das Meta–Objekt–Protokoll ist die Basis für das Klassensystem von Smalltalk. Jedoch tritt es bei der Applikationsentwicklung kaum in Erscheinung. Vorwiegend Entwicklungswerkzeuge greifen über das Meta–Objekt–Protokoll auf das Laufzeitsystem zu.

5.3.3 Interaktive Anwendung, MVC

Struktur Eine der wichtigsten Anforderungen an eine interaktive Anwendung ist es, die Kernfunktionalität der Anwendung unabhängig von der Benutzeroberfläche zu halten und das Aussehen und Gestalten der Benutzeroberfläche möglichst flexibel zu gestalten. [BMR+96, S.123]

Ein Standard–Konzept hierfür ist die Aufteilung in Modell, View und Controller (MVC). Die Elemente von MVC sind:

- Das Modell, welches die Kernfunktionalität enthält und unabhängig von der Darstellung gestaltet wird,
- die View, welche für die Bildschirmdarstellung verantworlich ist und die durch einen Benachrichtigungs–Mechanismus über Änderungen im Modell informiert wird und
- der Controller, der Benutzereingaben verarbeitet und Ereignisse erzeugt.

Die beiden Komponenten View und Controller arbeiten sehr eng zusammen, wohingegen das Modell keine direkte Kenntnis von den beiden anderen Komponenten hat. Dadurch wird seine Unabhängigkeit von der Bildschirmdarstellung erreicht. [KP88]

Unterschiede MVC dient gleichfalls der Strukturierung von einzelnen Widgets und der Strukturierung der gesamten Applikation.

Zur Realisierung der Widgets verwendet IBM Smalltalk die Oberflächenelemente des Betriebssystems. Diese fassen die drei Komponenten in einem Objekt zusammen. Eine Applikation ist in IBM Smalltalk nach dem MVC–Konzept aufgeteilt. Die Ereignisbehandlung dient der Benachrichtigung des Modells durch die Controller–Elemente. Die View wird vom Modell durch einen expliziten Methodenaufruf über Änderungen unterrichtet. Über den Abhängigkeitsmechanismus ist es möglich, diese Benachrichtigung automatisch vorzunehmen.

In Java ist die MVC–Aufteilung auch bei den Widgets vorhanden. Allerdings fassen die Widgets in Java die View und den Controller zusammen zum sogenannten Delegate (siehe Abbildung 5.1).

Die Standardimplementierungen (`JButton` usw.) erzeugen eine Kombination aus einem Delegate und einem passenden Modell, womit das resultierende Widget nach außen wie in Smalltalk erscheint. Die Applikation ist in Java ebenfalls nach dem MVC–Konzept aufgeteilt. Die Ereignisbehandlung dient der Benachrichtigung des Modells durch die Controller–Elemente. Die View kann vom Modell durch einen expliziten Methodenaufruf über Änderungen unterrichtet werden. Eine Java–Applikation kann aber auch eigene Adapter erstellen, die Änderung automatisch an die View–Elemente weiterreichen.

Migration Der grundsätzliche Aufbau dieses Architekturmusters ist in Smalltalk und Java so ähnlich, daß daraus keine großen Architekturänderungen der Applikation zu erwarten sind. Im Detail unterscheiden sich die einzelnen Mechanismen, die zur Realisierung dieses Musters eingesetzt werden soweit, daß sie in begrenztem Umfang Einfluß auf die Architektur ausüben. Eine ausführliche Behandlung des GUI–Frameworks erfolgt im Abschnitt 4.3.5.

Probleme

- Abgesehen von Detailunterschieden der verwendeten Mechanismen ist mit keinen grundsätzlichen Problemen zu rechnen.

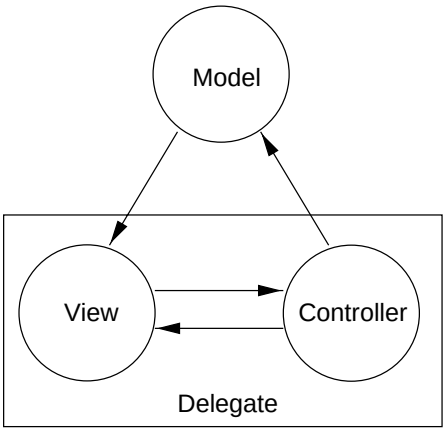


Abbildung 5.1: Delegate–Variante von MVC

Abhängigkeiten Dieses Architekturmuster verläßt sich auf das Ereignismodell zur Benachrichtigung des Modells und den Abhängigkeitsmechanismus, um die Views automatisch aktualisieren zu können. Dynamischer Methodenaufruf bzw. callbacks dienen dem Anschluß der View an das Modell über Adapter.

6

Vergleich der Applikationsarchitekturen

Dieses Kapitel zeigt, wie sich die Architektur einer Applikation bei einer Migration von Smalltalk nach Java ändert bzw. ändern kann.

Hierzu werden die Erkenntnisse der beiden letzten Kapitel zueinander in Beziehung gesetzt. Kapitel 4 führte die Elemente der Smalltalk–Plattform auf und untersuchte sie daraufhin, wie deren Funktionalität in Java realisiert wird. Kapitel 5 betrachtete die Migration von Mustern auf verschiedenen Abstraktionsebenen.

In diesem Kapitel werden Architekturänderungen auf zwei Wegen untersucht. Zunächst wird der Einfluß der Problemklassen, die bei der Migration der Muster gefunden wurden, auf die Änderung der Applikationsarchitektur dargestellt. Anschließend werden die verschiedenen Plattformelemente daraufhin untersucht, wie groß die Änderung der Applikationsarchitektur ist, die ihre Migration mit sich bringt.

6.1 Einfluß der Problemklassen auf Architekturänderung

Dieser Abschnitt klassifiziert die bei der Migration auftretenden Probleme und stellt dar, welche Architekturänderungen durch diese Probleme bei einer Migration hervorgerufen werden. Die Probleme wurden durch die Betrachtung der Migration der Idiome, Entwurfsmuster und Architekturmuster im letzten Kapitel gefunden. Dieser Abschnitt stellt deren Einfluß zusammenfassend dar. Anhang C enthält eine Kurzübersicht der behandelten Probleme.

Problem 1

Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.

Hintergrund In Smalltalk führt der Compiler lediglich eine Syntaxüberprüfung durch und übersetzt das Programm in den Zwischencode der virtuellen Maschine. Eine Typüberprüfung wird erst zur Laufzeit vorgenommen. Sie stellt fest, ob ein Objekt die empfangene Nachricht tatsächlich versteht.

In Java führt der Compiler diese Typüberprüfung statisch aus. Somit kann in den meisten Fällen schon zur Übersetzungszeit die falsche Verwendung eines Objekts festgestellt werden. Allerdings kann auch in Java nicht immer die korrekte Verwendung eines Objekts garantiert werden (4.1.1).

Architekturänderungen Um einem Objekt in Smalltalk eine Nachricht schicken zu können, muß ein Klient dessen Typ nicht kennen. In Java muß der Typ bekannt sein, damit der Compiler statisch überprüfen kann, ob der Empfänger die Nachricht versteht. Für die Migration kann das folgendes bedeuten:

- eine Verschiebung der in einer Subklasse definierten Methoden in die Superklasse, damit ein Klient auf das Protokoll der Subklasse über eine Referenz der Superklasse zugreifen kann (5.2.8);
- eine explizite Typumwandlung auf die konkrete Klasse eines Objekts (Downcast) vor dessen Verwendung — beispielsweise bei Speicherung in einer Collection (5.1.6);
- das Implementieren eines Java–Interfaces, über das ein Klient auf ein Objekt zugreifen kann. Dieses Interface wird vom Klienten definiert und ermöglicht ihm die Kommunikation mit einem beliebigen Objekt, das zu einem späteren Zeitpunkt als er selbst erstellt wird.

In Smalltalk kann ein Symbol dazu verwendet werden, einen dynamischen Methodenaufruf auszuführen, d.h. die aufgerufene Methode ist erst zur Laufzeit bekannt. In Java bedeutet das ein Ermitteln des Methodenobjekts mit Hilfe der reflexiven Eigenschaften von Java. Zudem erzwingt der Compiler die Kapselung des Aufrufs in einer Ausnahmebehandlung (5.2.5, 5.2.8, 5.2.9, 5.2.12).

In Java müssen sämtliche Variablen, Parameter und Methoden eindeutig getypt sein. D.h. bei der Migration muß diese Typisierung in den Programmtext eingebaut werden (5.1.18).

Polymorphie beschränkt sich in Smalltalk darauf, daß der Klient die konkrete Klasse des Empfängers nicht kennt. In Java ist die Auswahl polymorpher Methoden zusätzlich von den Typen der Parameter abhängig. Dadurch fallen evtl. Smalltalk–Idiome weg, die auf eine solche polymorphe Methodenauswahl in Smalltalk nachbilden (5.1.9).

Betroffene Muster Comparing Method (5.1.6), Double Dispatch (5.1.9), Interesting Return Value (5.1.18), Abstract Factory (5.2.1), Adapter (5.2.5), Bridge (5.2.6), Composite (5.2.7), Proxy (5.2.8), Command (5.2.9), Iterator (5.2.10), Observer (5.2.12), Template Method (5.2.13), Visitor (5.2.14)

Problem 2

Java unterscheidet primitive Typen und Referenztypen.

Hintergrund In Smalltalk sind ausschließlich Referenztypen definiert. In Java wird zwischen primitiven Typen und Referenztypen unterschieden (4.1.1).

Architekturänderungen Die Unterscheidung in primitive Typen und Referenztypen ist für die Migration insbesondere dann wichtig, wenn Elemente, die in Java primitiv sind, in Smalltalk als Objekte verwendet werden. Für die Migration eröffnen sich hier zwei Möglichkeiten.

Wenn die Objekteigenschaft für die Operation auf dem primitiven Element notwendig ist (etwa zur Speicherung in einer Collection), muß es in ein Wrapper–Objekt gepackt werden[1] (5.1.14, 5.1.15).

Oft verwenden die Smalltalk–Operationen ein Element so, wie es der Verwendung eines primitiven Elements in Java entspräche. Dann muß u.U. eine Smalltalk–Collection in ein primitives Array in Java umgewandelt werden. Diese Umwandlung hat Einfluß auf die Iteratoren zum Durchlaufen einer Collection (5.2.10). Zudem greifen Klienten dadurch auf andere Art auf die Elemente des Arrays zu (4.1.1).

Betroffene Muster Shortcut Constructor Method (5.1.3), Duplicate Removing Set (5.1.14), Temporarily Sorted Collection (5.1.15), Iterator (5.2.10)

[1] Dieser Vorgang wird *embedding* genannt.

Problem 3

In Java sind Blöcke keine Objekte erster Klasse wie in Smalltalk.

Hintergrund In Smalltalk ist *jedes* Sprachkonstrukt ein Objekt. Auch Codeblöcke können wie gewöhnliche Objekte in Variablen gespeichert werden oder einer Nachricht als Argumente mitgegeben werden. Sie werden durch einen Methodenaufruf im Kontext der definierenden Methode ausgewertet.
In Java dienen Codeblöcke lediglich der Strukturierung des Quelltextes.

Architekturänderungen Blöcke werden in Smalltalk an vielen Stellen auf viele verschiedene Arten verwendet (4.1.3). Jede Art der Verwendung hat bei der Migration ihren speziellen Einfluß auf die Applikationsarchitektur. Wir führen einige Verwendungen von Codeblöcken auf, sofern dabei deren Eigenschaft wichtig ist, als Objekte erster Klasse zu gelten:

- Durch Blöcke realisierte Kontrollstrukturen werden in Java auf Sprachkonstrukte abgebildet.
- Interne Iteratoren werden in Java durch externe Iteratoren dargestellt. Der jeweilige Codeblock wird in den Schleifenrumpf verschoben (5.2.10).
- Der Einsatz von Blöcken in Smalltalk als parallele Ablaufeinheiten führt in Java zur Definition neuer Klassen, die das `Thread`–Interface implementieren (4.1.5).
- Lokale Ausnahmebehandlung muß vom Klienten selbst übernommen werden, was den Code umständlicher macht (4.3.4).

Insgesamt sind die Änderungen, die durch die unterschiedlichen Eigenschaften von Codeblocks in Java und Smalltalk hervorgerufen werden, vielfältiger Natur. In den meisten Fällen hat die Migration jedoch keinen großen Einfluß auf die Architektur der Applikation.

Betroffene Muster Execute Around Method (5.1.7), Enumeration Method (5.1.13), Temporarily Sorted Collection (5.1.15), Adapter (5.2.5)

Problem 4

Sehr mächtiges Meta-Objekt-Protokoll in Smalltalk im Vergleich zur Reflection–API in Java.

Hintergrund Das Meta–Objekt–Protokoll definiert eine Schnittstelle, über die Metainformation abgefragt und die Klassenstruktur und das Verhalten des Smalltalk–Systems verändert werden kann. Über das Meta–Objekt–Protokoll hat man direkten Zugriff auf das Klassensystem von Smalltalk.

Das Reflection–API dient in Java lediglich der Abfrage von Metainformationen.

Architekturänderungen Falls in Smalltalk lediglich Metainformationen abgefragt werden, ist die Änderung durch die Migration minimal. Falls dagegen die Klassenstruktur oder das Verhalten des Systems geändert wird, kann die Änderung der Applikationsarchitektur so groß sein, daß eine Neuentwicklung in weiten Teilen notwendig ist.

Betroffene Muster Reflexion (5.3.2), Constructor Method (5.1.1), Constructor Parameter Method (5.1.2)

Problem 5

Instanzvariablen sind in Smalltalk privat.

Hintergrund Instanzvariablen sind in Smalltalk nur für die Instanz selbst sichtbar. Falls eine Instanzvariable von außen zugänglich sein muß, wird eine Getter– bzw. Setter–Methode definiert. In Java kann die Sichtbarkeit von Variablen durch Sichtbarkeitsmodifikatoren genau eingestellt werden.

Architekturänderungen Im einfachsten Fall besteht die Migration aus der Angabe des Modifikators, der die Sichtbarkeit passend einstellt (`private` für nach wie vor private Variablen und `public` für öffentlich zugängliche Variablen).

Falls die Getter– und Setter–Methoden entfernt werden sollen, müssen sämtliche Klienten dieser Methoden ebenfalls angepaßt werden. Dies ist aufgrund des dynamischen Typsystems nicht immer einfach.

Ein Entfernen der Getter–Methoden birgt noch ein anderes Problem. Die Initialisierung einzelner Variablen geschieht oft durch das Idiom Lazy Initialization. Dieses Idiom ist auf die Getter–Methode angewiesen (5.1.12). Jedoch kann dieses Idiom meist durch lokale Initialisierung ersetzt werden.

Betroffene Muster Constructor Parameter Method (5.1.2)

Problem 6

Die Sichtbarkeit von Methoden kann in Smalltalk nicht eingeschränkt werden.

Hintergrund In Smalltalk sind sämtliche Methoden öffentlich zugänglich. Ihre Sichtbarkeit kann nicht eingeschränkt werden. In Java dagegen ist mit Hilfe von Sichtbarkeitsmodifikatoren eine sehr genaue Einstellung möglich.

Architekturänderungen Um eine Methode privat zu machen, benötigt Smalltalk eine Laufzeit–Lösung. Dabei wird festgestellt, ob der Sender der Empfänger selbst ist (`self`). In Java kann dieser Code durch die Markierung der Methode mit `private` entfallen (5.2.4).

Beim Memento–Muster führt das beschriebene Problem dazu, daß das Objekt selbst als Memento verwendet wird. Durch den Sichtbarkeitsmechanismus besteht hierzu kein Anlaß mehr. Jedoch muß dann eine eigene Memento–Klasse angelegt werden. Meist wird auch noch ein neues Interface definiert, um die Memento–Klasse als solche markieren zu können (5.2.11).

In anderen Fällen kann der `private`–Modifikator eine falsche Verwendung einer Methode aufdecken, die vom Entwurf als privat angelegt ist (5.2.13).

Betroffene Muster Singleton (5.2.4), Memento (5.2.11), Template Method (5.2.13)

Problem 7

In Smalltalk ist kein Sprachkonstrukt zum Erzeugen von Instanzen definiert.

Hintergrund In Smalltalk wird eine Instanz nicht durch ein explizites Sprachkonstrukt angelegt, sondern von einer Klassenmethode (Konstruktormethode 5.1.1). Der Mechanismus zum Erzeugen neuer Klassen wird von der Metaklassenhierarchie realisiert, die parallel zur gewöhnlichen Klassenhierarchie verläuft (4.1.7).

Architekturänderungen Der Code zum Anlegen einer wohlgeformten Instanz muß von der Klassenmethode in einen Java–Konstruktor verschoben werden. In Smalltalk sorgt der Vererbungsmechanismus der Klassenmethoden dafür, daß eine Konstruktormethode in allen Subklassen sichtbar ist. Konstruktoren in Java werden dagegen nicht vererbt (außer parameterlose Konstruktoren). D.h. falls der Konstruktor der Superklasse einen oder mehrere Parameter definiert, ist er in den Subklassen nicht vorhanden. Diese müssen einen solchen Konstruktor selbst definieren und darin den Konstruktor der Superklasse explizit aufrufen.

In Java ist die Art der Konstruktorenverkettung in der Sprache definiert. In Smalltalk dagegen definiert die Implementierung der Konstruktormethode, wie die Konstruktoren in diesem Fall miteinander verkettet werden (auf Konstruktorenverkettung kann sogar ganz verzichtet werden). Die benutzergestaltete Konstruktorverkettung muß in Java in den von der Sprache vorgegebenen Mechanismus eingepaßt werden. Der Teil der Konstruktormethode in Smalltalk *vor* dem Aufruf des Konstruktors der Superklasse kann nicht direkt nach Java übernommen werden. Im einfachsten Fall hilft ein einfaches Umschreiben des Konstruktors. Allerdings kann u.U. ein Refaktorieren der Klienten notwendig sein (4.1.2).

Die Idiome zur Erzeugung einer wohlgeformten Instanz werden auch dazu genutzt, die Initialisierung von Instanzvariablen vorzunehmen. Die Initialisierung von Variablen wird in Java meist von einem dedizierten Sprachkonstrukt übernommen (5.1.11).

Betroffene Muster Constructor Method (5.1.1), Constructor Parameter Method (5.1.2), Converter Method (5.1.4), Converter Constructor Method (5.1.5), Explicit Initialization (5.1.11), Lazy Initialization (5.1.12), Singleton (5.2.4)

Problem 8

In Smalltalk ist kein Sprachkonstrukt zum Initialisieren von Variablen definiert.

Hintergrund In Smalltalk fehlen die Sprachkonstrukte zum Initialisieren von Variablen. Daher haben sich Idiome eingebürgert bzw. die Initialisierung wird beim Anlegen der Instanz vorgenommen.

Architekturänderungen Die Initialisierung einer Variablen geschieht in Java in einem static initializer, einem instance initializer oder direkt bei der Variablendeklaration.

Das Konfigurieren einer Instanz bzw. die Initialisierung wird in Smalltalk oft durch eine constructor parameter method erledigt (5.1.2). Die Migration bedeutet also ein Verschieben des entsprechenden Codes in das passende Sprachkonstrukt.

Die Initialisierung einer Reihe von Variablen wird in Smalltalk oft in einer Initialisierungsmethode zusammengefaßt (5.1.11). Dieser Code wird in Java in einen instance initializer verschoben. Dadurch entfällt der Aufruf der Initialisierungsmethode im Konstruktor.

Zur Initialisierung einer einzelnen Variablen wird oft eine Getter–Methode verwendet (5.1.12). Diese kann in Java meist entfallen, indem die Initalisierung ebenfalls im instance initializer vorgenommen wird bzw. direkt bei der Deklaration der betroffenen Variablen.

Betroffene Muster Explicit Initialization (5.1.11), Lazy Initialization (5.1.12), Constructor Parameter Method (5.1.2)

Problem 9

Basisklassen können in Smalltalk geändert werden, in Java dagegen nicht.

Hintergrund Üblicherweise wird eine Klasse dadurch erweitert, daß eine neue Subklasse angelegt wird. Durch den Mechanismus der base class extension kann in Smalltalk eine beliebige Klasse der bestehenden Klassenbibliothek erweitert werden. D.h. eine Applikation kann neue Methoden für bestehende Basisklassen definieren. Diese Erweiterbarkeit schließt auch Operatoren mit ein, weil Operatoren in Smalltalk als Methoden realisiert sind.

In Java können weder die Basisklassen erweitert werden, noch können neue Operatoren definiert oder bestehende überladen werden.

Architekturänderungen Die Erweiterung von Basisklassen bedeutet für die Migration auf jeden Fall ein Refaktorieren, weil sämtlicher Code, der sich auf die Basisklassenerweiterung verläßt, umgeschrieben werden muß. Dies bedeutet in vielen Fällen das direkte Aufrufen eines Konstruktors der gewünschten Klasse (5.1.4) oder die Verwendung einer komplizierteren Methode der Klassenbibliothek (5.1.3). Große Architekturänderungen ergeben sich dadurch allerdings nicht.

Betroffene Muster Shortcut Constructor Method (5.1.3), Converter Method (5.1.4), Converter Constructor Method (5.1.5)

Problem 10

In Smalltalk sind keine Interfaces definiert.

Hintergrund Insbesondere die Klassen der Klassenbibliothek müssen oft auf Objekte zugreifen können, deren Klassen erst später definiert werden. In Smalltalk ist das aufgrund der dynamischen Typisierung kein Problem. Um in Java trotzdem die Typsicherheit gewährleisten zu können, sind in Java Interfaces definiert. Sie sind eine Art Rumpfversion der Mehrfachvererbung (4.1.1).

Architekturänderungen Ein Interface muß überall dort eingesetzt werden, wo eine früher übersetzte Klasse auf eine Instanz einer erst später zu definierende Klasse zugreifen muß. Dieses Interface muß im Kopf der Klassendefinition der später definierten Klasse angegeben werden.

Durch Interfaces kann die Implementierung einer Klasse von ihrer Schnittstelle getrennt werden. In Smalltalk werden hierzu manchmal getrennte Klassen verwendet, die über Delegation miteinander verbunden sind. In Java werden solche Klassen also manchmal zusammengefaßt. Dadurch fallen meist gemeinsame Superklassen weg (5.2.6).

Interfaces können zu einem Abbau komplizierter Klassenhierarchien führen, falls die Superklasse in Smalltalk nur dazu dient, eine gemeinsame externe Schnittstelle zu definieren (5.2.7). Die Schnittstelle wird in ein Interface gesteckt und die Klassenhierarchie zum Teil aufgelöst werden.

Betroffene Muster Converter Method (5.1.4), Comparing Method (5.1.6), Mediating Protocol (5.1.10), Bridge (5.2.6), Composite (5.2.7)

Problem 11

In Java sind keine Klasseninstanzvariabeln definiert.

Hintergrund Klasseninstanzvariablen sind eine Besonderheit von Smalltalk. Sie gehen darauf zurück, daß jede Klasse in Smalltalk durch ihr Klassenobjekt repräsentiert wird. Eine Klasseninstanzvariable ist eine Variable, die direkt diesem Klassenobjekt zugeordnet ist. Daher hat jede Unterklasse eine eigene Klasseninstanzvariable, obwohl sie nur einmal in der Superklasse deklariert ist.

Architekturänderungen Klasseninstanzvariablen müssen in Java entfallen. Es gibt kein direktes Sprachkonstrukt, das die gleiche Funktionalität erfüllt. Sie werden durch abstrakte Getter– und Setter–Methoden dargestellt, die in der Superklasse definiert sind und von jeder Subklasse implementiert werden müssen. Jede Verwendung dieser Variablen muß durch den entsprechenden Methodenaufruf ersetzt werden. Die Änderung der Architektur ist jedoch minimal.

Betroffene Muster Factory Method (5.2.2), Singleton (5.2.4)

Problem 12

Der Abhängigkeits–Mechanismus ist in Java nicht im Objekt–Framework enthalten.

Hintergrund In Smalltalk kann sich ein Observer bei einem beliebigen Objekt (Subjekt) für den Erhalt von Änderungsmeldungen registrieren. In Java dagegen muß das Subjekt von `Observable` erben, um Observer verwalten zu können.

Architekturänderungen Die direkte Migration des Abhängigkeitsmechanismus bedeutet, daß die Klassenstruktur so umgebaut werden muß, daß das Subjekt von `Observable` erbt. Das ist nicht immer möglich. Außerdem funktioniert dieses Verfahren nur für selbstgeschriebene Klassen, da in die Vererbungsstruktur bestehender Klassen nicht eingegriffen werden kann.

Meist wird der Abhängigkeitsmechanismus in Java durch das Ereignismodell ersetzt. Das Ereignismodell ist jedoch auch mit einer Architekturänderung verbunden. Der Observer muß ein Listener–Objekt definieren, welches ein passendes Listener–Interface implementiert. Der Observer kann sich außerdem nicht mehr für jede Änderung im Subjekt registrieren, sondern muß sich auf eine Ereignisklasse festlegen. Bei diesem Verfahren wird also eine Klasse, eine Interface und evtl. noch eine Hilfsklasse zusätzlich in die Architektur eingebracht.

In beiden Fällen kann es nötig sein, die Klassenstruktur zu ändern.

Betroffene Muster Observer (5.2.12), Interaktive Anwendung, MVC (5.3.3)

Problem 13

Listener–Interface schreibt Namen der Behandlermethoden in Java vor.

Hintergrund Listener–Klassen definieren das Protokoll, über welches die ereigniserzeugenden Objekte mit registrierten Behandlerobjekten kommunizieren. Dieses Protokoll legt fest, welche Methode im Behandlerobjekt zur Behandlung welchen Ereignisses aufgerufen wird. In Smalltalk kann eine beliebige Methode zum Behandlen eines Ereignisses registriert werden.

Architekturänderungen Dieses Problem kann in zweierlei Hinsicht zu einer Architekturänderung führen.

Die Behandlermethoden, die zu einer Ereignisklasse gehören, können in Smalltalk über mehrere Objekte verteilt sein. In Java müssen sie in einem Behandlerobjekt zusammengefaßt sein. Falls die Behandlermethoden in ihren ursprünglichen Objekten auf den internen Zustand zugreifen, muß diese Möglichkeit auch nach der Migration bestehen. Hierzu sind evtl. Adapter in Form von inneren Klassen nötig (5.2.5).

Weiterhin hat die Festlegung der Namen im Interface zur Folge, daß die Behandlermethoden umbenannt werden müssen. Folglich muß der Aufruf in allen Klienten geändert werden, die diese Methoden aufrufen. Die Klienten sind aufgrund des dynamischen Typsystems bzw. aufgrund von dynamischen Methodenaufrufen nicht immer leicht festzustellen.

Betroffene Muster Observer (5.2.12)

Problem 14

Klassenobjekte bleiben in Smalltalk über einen Neustart des Systems hinweg erhalten.

Hintergrund Eine Klasse wird in Smalltalk durch ein Klassenobjekt repräsentiert. Dieses Klassenobjekt wird beim Anlegen der Klasse erzeugt. Anschließend können die Klassenvariablen und Klasseninstanzvariablen vom Entwickler initialisiert werden. Vor dem Verlassen des Smalltalk–Systems wird das Klassenobjekt im Image abgespeichert und beim Neustart wiederhergestellt. Die Initialisierung geht nicht verloren und somit wird in machen Fällen kein expliziter Mechanismus zur Initialisierung von Klassenvariablen und Klasseninstanzvariablen benötigt. In Java wird das Klassenobjekt bei jedem Laden der Klasse neu angelegt, weshalb die Klassenvariablen jedesmal wieder neu initialisiert werden müssen.

Architekturänderungen In Java ist ein Sprachkonstrukt zum Initialisieren von Klassenvariablen definiert (static initializer). Die Schwierigkeit liegt in diesem Fall darin, die Startwerte der Variablen zu ermitteln. Eine Möglichkeit ist, den aktuellen Wert der Variablen zu nehmen. Daneben existiert noch die Möglichkeit einer Initialisierung durch das Smalltalk–System. Manche registrierten Klassen werden vom Smalltalk–System über den Neustart unterrichtet (`startUp`). Diese können nun ihrerseits Initialisierungen ausführen bzw. weitere Initialisierungsmethoden aufrufen. Durch die Migration nach Java kann ein Großteil dieser Initialisierungen im static initializer geschehen, weshalb einige Methoden wegfallen können.

Betroffene Muster Explicit Initialization (5.1.11), Lazy Initialization (5.1.12)

Problem 15

In Java sind keine internen Iteratoren definiert.

Hintergrund Interne Iteratoren werden in Smalltalk von den Collection–Klassen definiert. Sie dienen dem Iterieren über den Elementen einer Collection. Dem Aufruf eines internen Iterators wird ein Codeblock mitgegeben, welcher die auszuführenden Aktionen auf dem Element kapselt. In Java sind ausschließlich externe Iteratoren definiert.

Architekturänderungen Bei der Migration müssen interne Iteratoren folglich zu externen Iteratoren umgeschrieben werden. Das bedeutet, in Java muß die Schleife zum Iterieren über den

Elementen explizit kodiert werden. Die verschiedenen Arten, Elemente beim Iterieren auszufiltern, müssen von Hand nachgebildet werden (5.2.10).

Die Transformation zu externen Iteratoren kann sich ebenfalls auf Objekte auswirken, die eine Collection zum Speichern von Elementen verwenden. Um einem Klienten Zugriff auf diese Elemente zu geben, wird in Smalltalk der Iteratorblock durch Delegation an die Collection weitergegeben. In Java liefert dagegen die Collection einen externen Iterator. Dieser Iterator wird an den Klienten weitergegeben (5.1.13).

Betroffene Muster Enumeration Method (5.1.13), Iterator (5.2.10)

Problem 16

Smalltalk unterscheidet zwischen `shallowCopy` *und* `deepCopy`*, während Java nur die Methode* `clone()` *definiert.*

Hintergrund In Smalltalk sind zwei Mechanismen zum Kopieren eines Objekts vordefiniert. D.h. ein Klient kann sich beim Verwenden eines Objekts entscheiden, welcher Mechanismus in diesem Fall verwendet werden soll. In Java fällt die Entscheidung über die Art des Kopierens bei der Definition der Klasse. In der Klassendefinition kann der vordefinierte Mechanismus überschrieben werden.

Architekturänderungen Schwierigkeiten sind zu erwarten, falls in Smalltalk beide Kopiermechanismen einer Klasse gemischt verwendet werden. In Java muß dann der zweite Mechanismus durch eine eigene Methode nachgebildet werden. Diese Methode muß im Extremfall in allen Klassen eingefügt werden, deren Instanzen als Elemente des zu kopierenden Objekts vorkommen können. Das führt potentiell zu einer Erweiterung des Protokolls bei sehr vielen Klassen.

Dieses Nachbilden in Java funktioniert nur, wenn der Code sämtlicher beteiligter Klassen selbst verändert werden kann. Falls dies nicht der Fall ist, muß sich die Migration auf einen der beiden Kopiermechanismen beschränken. Diese Beschränkung kann dazu führen, daß die Klienten der zu kopierenden Klasse selbst in den Kopierprozeß eingreifen müssen.

Betroffene Muster Prototype (5.2.3)

Problem 17

In Java ist keine Objekt–Mutation definiert.

Hintergrund In Smalltalk können zwei Objekte gegeneinander ausgetauscht werden. Das bedeutet ein Vertauschen sämtlicher Verweise auf die beiden Objekte.

Architekturänderungen Die Objektmutation ist ein sehr mächtiges Hilfsmittel in Smalltalk. Es ist in Java nicht vorhanden. Um die Objekt–Mutation zu ersetzen kann es nötig sein, mehrere zusammenarbeitende Klassen zu erstellen (5.2.8). Alternativ könnte das Vertauschen sämtlicher Verweise von der Applikation selbst implementiert werden. Auf jeden Fall ist die Verwendung dieses Plattformelements mit großen Architekturänderungen verbunden. Glücklicherweise wird dieser Mechanismus von Smalltalk sehr selten eingesetzt.

Betroffene Muster Proxy (5.2.8)

Problem 18
Ein Objekt kann in Java nicht von `null` *erben.*

Hintergrund Eine Subklasse erbt das Protokoll ihrer Superklasse. Durch das Erben von `null` kann in Smalltalk eine Klasse definiert werden, die keinerlei Protokoll besitzt.

Architekturänderungen In Java existiert dieser Mechanismus nicht. Eine Klasse muß zumindest von `Object` erben. Durch die Migration müssen entweder sämtliche Methoden entsprechend überschrieben werden, oder auf das Objekt wird über ein Interface zugegriffen, welches keine Methoden definiert. Auf die Architektur der Applikation wirkt sich diese Änderung kaum aus. Zudem wird dieser Mechanismus in Smalltalk selten eingesetzt.

Betroffene Muster Proxy (5.2.8)

Problem 19
In Smalltalk sind Instanzen der Stream–Klassen freier positionierbar als in Java.

Hintergrund Die Stream–Klassen dienen dem Iterieren über den Elementen einer Collection. Sie können frei auf der zugrundeliegenden Collection positioniert werden. In Java können externe Iteratoren lediglich sequentiell auf die Elemente zugreifen.

Architekturänderungen Die freie Positionierbarkeit kann es in Java notwendig machen, auf die Elemente der Collection direkt zugreifen zu müssen. Dadurch wird die Kapselung der Collection durch den Iterator aufgehoben und der Code im Klienten zum Arbeiten mit den Elementen wird komplizierter. Die Änderung in der Architektur drückt sich durch Ändern des Protokolls der beteiligten Klassen und Umschreiben des Zugriffscodes auf die Elemente im Klienten aus.

Betroffene Muster Iterator (5.2.10)

6.2 Bewertung der Plattformelemente

Jede Verwendung eines Plattformelements führt bei der Migration zu einer mehr oder weniger starken Änderung der Applikationsarchitektur. Der Grad dieser Änderung wird für die verschiedenen Plattformelemente diskutiert und der Einfluß auf die Architektur bewertet.

Hierzu definieren wir zunächst die beiden Bewertungskriterien Einfluß auf Applikationsarchitektur und Unterschied zwischen Smalltalk und Java. Durch die Kombination dieser beiden Kriterien können wir eine Aussage über den Grad der Architekturänderung bei der Migration treffen, die sich aus der Verwendung des jeweiligen Plattformelements ergibt.

Einfluß auf Applikationsarchitektur
Um über typische Architekturmerkmale sprechen zu können, betrachten wir Idiome, Entwurfsmuster und Architekturmuster. Zur Implementierung dieser Muster werden die Elemente der Plattformen verwendet bzw. die Elemente werden anhand dieser Muster strukturiert. Um den Grad des Einflusses eines Plattformelements auf die Applikationsarchitektur zu bewerten, betrachten wir auf welcher Ebene der Musterhierarchie sich das durch das Plattformelement realisierte Muster befindet. Der Einfluß auf die Applikationsarchitektur ist eine qualitative Größe.

Beispielsweise ist die Verwendung der Methode `yourself` lediglich als Idiom zu finden. Daher ist der Einfluß dieser Methode auf die Applikationsarchitektur gering. Das GUI–Framework dagegen implementiert ein Architekturmuster. Daher ist sein Einfluß auf die Applikationsarchitektur sehr groß.

Unterschied zwischen Smalltalk und Java
Dieses Kriterium drückt aus, wieviel Änderungsaufwand im Programmcode durch die Migration notwendig wird. Das kann ein Umstellen in der Klassenhierarchie bedeuten oder lediglich das Umformatieren von Programmcode. Hierbei interessiert uns nicht, ob diese konkreten Änderungen beispielsweise durch einen Texteditor automatisch vorgenommen werden können oder vom Entwickler noch von Hand durchgeführt werden müssen (Bsp. Umstellungen der Klassenstruktur). Durch das Aufkommen von Werkzeugen zur Refaktorierung kann in Zukunft auch in diesem Bereich mehr automatisiert werden [Opd97]. Der Unterschied zwischen Smalltalk und Java ist eine quantitative Größe.

Beispielsweise ist die Verwendung von Zufallszahlen auf beiden Plattformen nahezu identisch, weshalb der Grad der Änderung gering ist. Dagegen bringt die Verwendung von Ausnahmen große Veränderungen mit sich.

Sprache (4.1.1)

Einfluß auf Architektur Die Sprache beschreibt, wie ein Programm aufgeschrieben wird und wie die einzelnen Sprachkonstrukte zusammenarbeiten. Dadurch ist der gesamte Quellcode von den Eigenschaften der Sprache abhängig. Allerdings hat die reine Quellcodeebene kaum einen Einfluß auf die Architektur (Bsp. Operatoren, Ausdrücke, Literale, Namensräume). Die Sprache gibt aber auch Möglichkeiten zur Gestaltung der Applikationsarchitektur vor (Bsp. Klassenbeziehungen, Programmdefinition). Der Bereich dazwischen hebt sich von der reinen Quellcodeebene ab, hat aber nur in Einzelfällen einen größeren Einfluß auf die Architektur (Datentypen, Variablen, Nachrichten und Methoden). Daneben existieren Verwaltungsmechanismen, die aber eher der Quellcodeorganisation dienen (Protokolle, Kategorien).

Unterschied zwischen Smalltalk und Java Die größten Unterschiede sind dort zu finden, wo der Einfluß auf die Applikationsarchitektur nicht sonderlich groß ist. Dagegen sind die Möglichkeiten der Gestaltung der Beziehungen zwischen den Klassen eng beieinander. Die Programmdefinition hebt sich hiervon ab. Die Trennungslinie zwischen Applikation und System ist nicht immer leicht zu ziehen. Allerdings bieten die Entwicklungswerkzeuge hierfür meist eine Unterstützung an, womit Applikationen getrennt verwaltet werden können. Dadurch bleiben Änderungen in der Architektur begrenzt.

Bewertung Insgesamt wirkt sich die Migration auf Sprachebene nicht so stark auf die Applikationsarchitektur aus wie zunächst erwartet. Auch wenn sich Smalltalk und Java zum Teil stark unterscheiden, so ist doch die Umsetzung in ein passendes Java–Konstrukt meist sehr naheliegend und schlüssig zu bewerkstelligen.

Fundamental (4.1.2)

Einfluß auf Architektur Die Elemente der Fundamental–Protokollklasse unterscheiden sich hinsichtlich ihres Einflusses auf die Applikationsarchitektur. Teilweise ist der Einfluß zu vernachlässigen (Zeichen, Boolesche Werte) oder nur in Details zu sehen (Pseudovariablen). Den weitreichendsten Einfluß hat die Klasse `Object`. Sie definiert das grundlegende Verhalten aller Objekte und durchdringt damit die gesamte Applikation. Falls Objekt–Mutation auftritt, ist ihr Einfluß äußerst groß, allerdings begrenzt auf diese Stelle.

Unterschied zwischen Smalltalk und Java Die Arbeit mit Zeichen und Booleschen Werten unterscheidet sich nicht grundlegend. Die Objekt–Erzeugung und Initialisierung sind zwar auf andere

Art realisiert, aber mit überschaubarem Aufwand zu migrieren. Ein Mechanismus zum Freigeben von Objekten existiert in Smalltalk nur begrenzt. Die Smalltalk–spezifischen Fehlermethoden werden in Java meist nicht benötigt. Objekt–Mutation existiert in Java gar nicht.

Bewertung Objekt–Mutation ist der kritischste Aspekt von Fundamental. Die Änderungen können im Einzelfall ziemlich weitreichend sein. Allerdings wird Objekt–Mutation selten verwendet. Der Umgang mit Objekten unterscheidet sich im Detail, kann aber so in Java abgebildet werden, daß sich die Änderung der Architektur in Grenzen hält.

Valuable (4.1.3)

Einfluß auf Architektur Valuable ist laut ANSI–Standard für Smalltalk der Überbegriff für alle Objekte, die durch Aufruf einer `value`–Methode ausgewertet werden. In dieser Arbeit betrachten wir hiervon ausschließlich Codeblöcke, die in Smalltalk als Objekte erster Klasse behandelt werden. Der Einfluß von Codeblöcken macht sich in einer Applikation vorwiegend im Detail bemerkbar. Sie ermöglichen in Smalltalk die Realisierung von Kontrollstrukturen und die elegante Formulierung von internen Iteratoren und einer lokalen Ausnahmebehandlung, wo sie nur einen geringen Einfluß auf die Gesamtarchitektur entwickeln. Weiterhin sind sie die Ablaufeinheiten der Parallelverarbeitung von Smalltalk und die von der Ausnahmebehandlung zu schützenden Programmstücke, wobei der Einfluß dieser beiden Aspekte auf die Applikationsarchitektur deutlich zu spüren ist.

Unterschied zwischen Smalltalk und Java Codeblöcke sind in Java auch vorhanden, können aber nicht als Objekte erster Klasse verwendet werden, sondern dienen lediglich der Strukturierung von Quellcode. Die Funktion der Codeblöcke wird in Java in den meisten Fällen von anderen Sprachkonstrukten oder Plattformelementen übernommen.

Bewertung Die Unterschiede zwischen Java und Smalltalk sind nur dort deutlich ausgeprägt, wo kein großer Einfluß auf die Applikationsarchitektur besteht. Daher sind die Änderungen in der Architektur meist zu vernachlässigen.

Der größte Unterschied liegt in der Parallelverarbeitung. Allerdings ist die Migration der Parallelverarbeitung aufgrund der sehr eingeschränkten Fähigkeiten von Smalltalk in diesem Bereich kein Problem.

Reflexion (4.1.7)

Einfluß auf Architektur Die reflexiven Eigenschaften von Smalltalk sind Teil des Meta–Objekt–Protokolls. Das Meta–Objekt–Protokoll definiert die Schnittstelle zum Klassensystem. Über dieses Protokoll können Metainformationen abgefragt, aber auch die bestehende Klassenstruktur verändert und das Verhalten des Systems beeinflußt werden. Die Abfrage und Verwendung von Metainformationen kann aufgrund der Dynamik, die dadurch ermöglicht wird, einen mittleren Einfluß auf die Applikationsarchitektur haben. Eine weitergehende Verwendung des Meta–Objekt–Protokolls, wie Änderung der Klassenstruktur oder des Systemverhaltens, hat massiven Einfluß auf die Applikationsarchitektur.

Unterschied zwischen Smalltalk und Java In Java sind lediglich solche reflexive Eigenschaften definiert, die der *Abfrage* von Metainformationen dienen. Eingriffe in die Klassenstruktur bzw. Veränderungen des Systemverhaltens sind nicht möglich.

Bewertung Die intensive Verwendung des Meta–Objekt–Protokolls kann derart bestimmend für die Applikationsarchitektur sein, daß oft nur eine Neuentwicklung möglich ist. Die Architektur der gleichen Applikation kann in Java vollständig anders aussehen, insbesondere weil viele Mechanismen des Meta–Objekt–Protokolls nicht nachgebildet werden können.

Ausnahmebehandlung (4.3.3)

Einfluß auf Architektur Eine Ausnahme bedeutet ein Abweichen vom ursprünglichen Kontrollfluß aufgrund einer unerwarteten Situation. Ein solches Abweichen kann sich auf dem Laufzeitstapel über mehrere Aktivierungsblöcke, sprich über mehrere Methodenaufrufe in möglicherweise unterschiedlichen Klassen hinziehen. Für den Entwurf der Ausnahmebehandlung bedeutet das, daß ein solches dynamisches Abweichen vom üblichen Kontrollfluß über mehrere Klassen hinweg bedacht werden muß — im Einzelfall über die gesamte Applikation hinweg.

Unterschied zwischen Smalltalk und Java In beiden Sprachen sind Blöcke die Behandlungseinheiten der Ausnahmebehandlung. Eine Applikation kann für einen Block einen Ausnahmebehandler registrieren.

In Smalltalk kann im Gegensatz zu Java mit Hilfe von Ausnahme–Collections ein Behandler für eine ganz Reihe von Ausnahmen registriert werden. Zudem sind in Smalltalk Default–Behandler definiert, die aufgerufen werden, falls entlang des Aufrufstapels kein Behandler gefunden werden kann. In Java propagiert die Ausnahmebehandlung in einem solchen Fall bis zur `main()`–Methode. Weil eine Applikation in Java Zugriff auf diese Methode hat, kann sie dort die Default–Behandler installieren und damit die Smalltalk–Funktionalität sehr gut nachbilden.

Bewertung Die Verwendung der Ausnahmebehandlung führt bei der Migration in der Regel zu keinen großen Änderungen der Architektur, wenn auch die Implementierungsdetails sehr unterschiedlich sein können.

Widgets (4.3.5)

Einfluß auf Architektur Die vorhandenen Widget–Klassen bestimmen, aus welchen Elementen sich eine graphische Benutzeroberfläche zusammensetzen kann und wie die Elemente zusammenarbeiten. Die Gestaltung der Oberfläche ist idealerweise von der Kernfunktionalität der Applikation getrennt. Folglich beschränkt sich der Einfluß der Widgets auf die Architektur der Präsentationsschicht.

Unterschied zwischen Smalltalk und Java Sämtliche relevanten Widget–Klassen können in Java ebenfalls dargestellt werden, auch wenn im Detail große Unterschiede vorliegen (z.B. Layout–Management).

Bewertung Aufgrund des unterschiedlichen Aufbaus muß die graphische Oberfläche normalerweise neugeschrieben werden. Die Gesamtarchitektur der Applikation ist aber mehr vom Interaktionsmechanismus zwischen der Präsentationsschicht und dem Applikationsmodell abhängig.

Ereignisbehandlung (4.3.3)

Einfluß auf Architektur Die Ereignisbehandlung dient dazu, einen Klienten über das Eintreten eines Ereignisses zu informieren (beispielsweise Aktion des Benutzers). Diese Ereignisse treten asynchron auf. Durch die Ereignisbehandlung wird der Kontrollfluß umgedreht (Hollywood–Prinzip).

Dieser Mechanismus hat einen großen Einfluß auf die Applikationsarchitektur, weil nicht mehr die Applikation den Kontrollfluß bestimmt, sondern sich in den globalen Kontrollfluß einklinken muß.

Unterschied zwischen Smalltalk und Java Die Ereignisbehandlung wird in Smalltalk über einen dynamischen Methodenaufruf realisiert. Eine Applikation kann für jedes definierte Ereignis einen eigenen Behandler registrieren. In Java werden Behandlerobjekte registriert, deren Protokoll über ein Interface angesprochen wird. Zudem müssen die Behandlermethoden einer Ereignisklasse zusammengefaßt werden. Da durch die Migration einige Umstrukturierungen notwendig sind, hält sich der Unterschied im mittleren Rahmen.

Bewertung Die Verwendung der Ausnahmebehandlung kann bei der Migration zu mittleren Änderungen in der Architektur führen.

Windows (4.3.5)

Einfluß auf Architektur Die Strukturierung der Fenster ist durch das GUI–Framework vorgegeben. Es bestimmt den Lebenszyklus der verwendeten Fenster, sorgt für den Anschluß an die Applikationslogik und strukturiert den Aufbau der Fenster. Insbesondere aufgrund des verwendeten Ereignismodells hat der Anschluß an die Applikationslogik einen großen Einfluß auf die Gesamtarchitektur der Applikation.

Unterschied zwischen Smalltalk und Java Der Aufbau der Fenster unterscheidet sich zwischen Smalltalk und Java stark. Der Lebenszyklus ist in Smalltalk in feinere Schritte unterteilt und damit spezifischer zu beeinflussen als in Java. Die Unterschiede in der Interaktion mit dem Applikationsmodell beruhen auf dem Ereignismodell.

Bewertung Insbesondere die Art des Anschlusses der Oberfläche an die Applikationslogik kann zu Veränderungen in der Gesamtarchitektur führen. Das ist u.a. auf das Typsystem bzw. das Ereignismodell zurückzuführen. Der unterschiedliche Lebenszyklus und vor allem der unterschiedliche Aufbau des Widget–Baums führen zu Änderungen in der Präsentationsschicht, die aber auf diese Schicht beschränkt bleiben.

Datentransfer in GUI (4.3.5)

Einfluß auf Architektur Zum Datentransfer in der graphischen Benutzeroberfläche durch den Benutzer existieren zwei Mechanismen. Zum Verwenden der Clipboard–Funktionalität ruft die Applikation lediglich Methoden auf, die von bestehenden Objekte angeboten werden. Der Drag&Drop–Mechanismus hat einen größeren Einfluß. Die Dialogsteuerung muß über Aktionen des Benutzers informiert werden, wozu eine Ereignisbehandlung vorgesehen werden muß. Diese tritt jedoch sehr konzentriert in der Dialogsteuerung und selten in Erscheinung, weshalb der Einfluß auf die Gesamtarchitektur vergleichsweise gering ist.

Unterschied zwischen Smalltalk und Java Der Clipboard–Mechanismus wird auf ähnliche Art realisiert. Der Drag&Drop-Mechanismus ist für Java zwar vorgesehen, im Moment allerdings noch nicht implementiert. Darauf muß bei der Migration also im Moment noch verzichtet werden.

Bewertung Durch die Migration der Verwendung des Datentransfermechanismus werden vorwiegend Änderungen in der Präsentationsschicht verursacht. Die Änderungen in der Gesamtarchitektur fallen gering aus.

Prozesse (4.1.5)

Einfluß auf Architektur Die Mechanismen zur Parallelverarbeitung können großen Einfluß auf die Applikationsarchitektur haben, da die Zusammenarbeit und Synchronisation paralleler Prozesse bedacht werden muß und sich der Kontrollfluß nicht mehr linear nachvollziehen läßt.

Unterschied zwischen Smalltalk und Java Smalltalk bietet nur minimale Unterstützung für Parallelverarbeitung. Dagegen ist Parallelverarbeitung eine der Stärken von Java, weshalb Java hier vielfältige Möglichkeiten bietet. Zudem sind die Ablaufeinheiten in Smalltalk Codeblöcke, während sie in Java vollständige Objekte sind.

Bewertung Die Migration ist unproblematisch, kann aber einige architekturelle Änderungen nach sich ziehen — insbesondere aufgrund der unterschiedlichen Ablaufeinheiten.

Objekt–Framework (4.3.1)

Einfluß auf Architektur Jedes Objekt erbt von der Klasse `Object`. Daher wirkt sich die Gestaltung dieses Protokolls auf die gesamte Applikation aus.

Unterschied zwischen Smalltalk und Java Die Klasse `Object` weist in Smalltalk und Java nur geringe Unterschiede auf. Java fehlen einige Details (Bsp. `addDependent`), besitzt hingegen Eigenschaften, die in Smalltalk fehlen (Bsp. `notify()`).

Bewertung In einigen Fällen müssen in Java statt des Objekt–Protokolls andere Klassen verwendet werden. Jedoch sind aufgrund von `Object` keine großen Änderungen in der Architektur zu erwarten.

Collections (4.2.1)

Einfluß auf Architektur Collections sind in Smalltalk überall zu finden. Sie implementieren grundlegende Datentypen (Strings, Symbole) und stellen eine Behälterfunktionalität zur Verfügung. Die Behälterfunktionalität dient unter anderem der Gestaltung von Aggregationen.

Die vorhandenen Datentypen beeinflussen vorwiegend die Implementierung, nicht die Architektur einer Applikation. Die Eigenschaften der Behälterfunktionalität bestimmt, wie dynamische Klassenbeziehungen gestaltet werden können und nehmen dadurch Einfluß auf die Gestaltungsmöglichkeiten der Architektur.

Unterschied zwischen Smalltalk und Java Im wesentlichen decken die Java–Collections die Behälter–Funktionalität von Smalltalk ab. Die Smalltalk–Collections sind jedoch noch etwas komfortabler und flexibler.

Die Datentyp–Funktionalität wird in Java durch Klassen dargestellt, die nicht zu den Collections gehören.

Bewertung Obwohl Collections in Smalltalk und Java viele Unterschiede aufweisen und deren Einfluß auf die Architektur gegeben ist, hält sich die Änderung aufgrund der unterschiedlichen Gewichtungen der einzelnen Aspekte in engen bis mittleren Grenzen.

Zufallszahlen (4.2.5)

Einfluß auf Architektur Der Einfluß auf die Architektur ist vernachlässigbar.

Unterschied zwischen Smalltalk und Java Java kann hier mehr, aber ein großer Unterschied ist nicht zu entdecken.

Bewertung Dieses Plattformelement hat gute Chancen dasjenige zu sein, dessen Migration die geringsten Veränderungen in der Applikationsarchitektur hervorruft.

Ströme (4.2.3)

Einfluß auf Architektur Ströme dienen dazu, über die Elemente einer `Collection` zu iterieren. Aufgrund der Existenz von internen Iteratoren in Smalltalk kommen Ströme nur selten zum Einsatz. Falls Ströme eingesetzt werden, bestimmen sie die Art, wie auf die darunterliegende Collection zugeriffen werden kann. Durch Ströme der Zugriff auf eine einzige Collection von vielen Stellen in der Applikation aus möglich.

Unterschied zwischen Smalltalk und Java Im wesentlichen besitzen Smalltalk–Ströme und externe Iteratoren in Java die gleichen Eigenschaften. Allerdings können viele Ströme in Smalltalk absolut positioniert werden. Dies geht in Java nicht.

Bewertung Meist hat die Migration von Strömen keine großen Änderungen der Applikationsarchitektur zur Folge. Falls die Eigenschaft der Positionierbarkeit eingesetzt wird, muß in Java der Zugriff direkt auf die Collection erfolgen. Dabei sind alle beteiligten Klassen zu ändern, wenn der Zugriff auf den Strom in Smalltalk an mehreren Stellen erfolgt.

Dateiströme (4.2.4)

Einfluß auf Architektur Dateiströme dienen dem Zugriff auf Dateien. Dieser Zugriffsmechanismus hat eine eigene Architektur, die jedoch nur lokalen Einfluß hat.

Unterschied zwischen Smalltalk und Java Der Zugriff auf Dateien ist in Smalltalk wie Java strombasiert. Allerdings weist die konkrete Gestaltung des Zugriffs große Unterschiede auf. Java ist hierbei um einiges flexibler.

Bewertung Auch wenn der Zugriff auf Dateien viele Änderungen bedeutet, so ist doch die Änderung in der Gesamtarchitektur nur punktuell zu sehen.

Zahlen (4.1.4)

Einfluß auf Architektur Der Einfluß der in einer Sprache vorhandenen Zahlen beschränkt sich darauf, welche Rechnungen mit Zahlen problemlos möglich sind und wieviel in der Applikation selbst berücksichtigt werden muß beim Umgang mit Zahlen. Dieser Einfluß tritt aber nur sehr lokalisiert auf.

Unterschied zwischen Smalltalk und Java Durch die automatische Umwandlung in den benötigten Zahlentyp in Smalltalk muß in der Applikation dafür nur in den seltensten Fällen Vorsorge getroffen werden. In Java muß jede Typumwandlung explizit erfolgen. Zudem decken sich die vorhandenen Zahlentypen nicht vollständig.

Bewertung Ein paar Detailunterschiede aber ohne Änderung an der Applikationsarchitektur.

Datum und Zeit (4.2.2)

Einfluß auf Architektur Der Einfluß ist äußerst gering.

Unterschied zwischen Smalltalk und Java Diese Elemente weisen in Smalltalk und Java wieder einige Detailunterschiede auf, haben aber prinzipiell die gleiche Funktionalität.

Bewertung Die Verwendung dieser Plattformelemente führt zu keiner nennenswerten Änderung in der Applikationsarchitektur.

Wie oben gesehen haben die verschiedenen Plattformelemente einen unterschiedlich großen Einfluß auf die Architektur einer Applikation. Beispielsweise beeinflußt das GUI–Framework die Applikationsarchitektur mehr, als die Klassen zur Datumsberechnung.

Gleichzeitig unterscheiden sich die verschiedenen Plattformelemente nach der Größe des Unterschiedes zwischen Java und Smalltalk. Insbesondere auf der Sprachebene sind sehr große Unterschiede vorhanden, während auf Frameworkebene viele Gemeinsamkeiten zu finden sind.

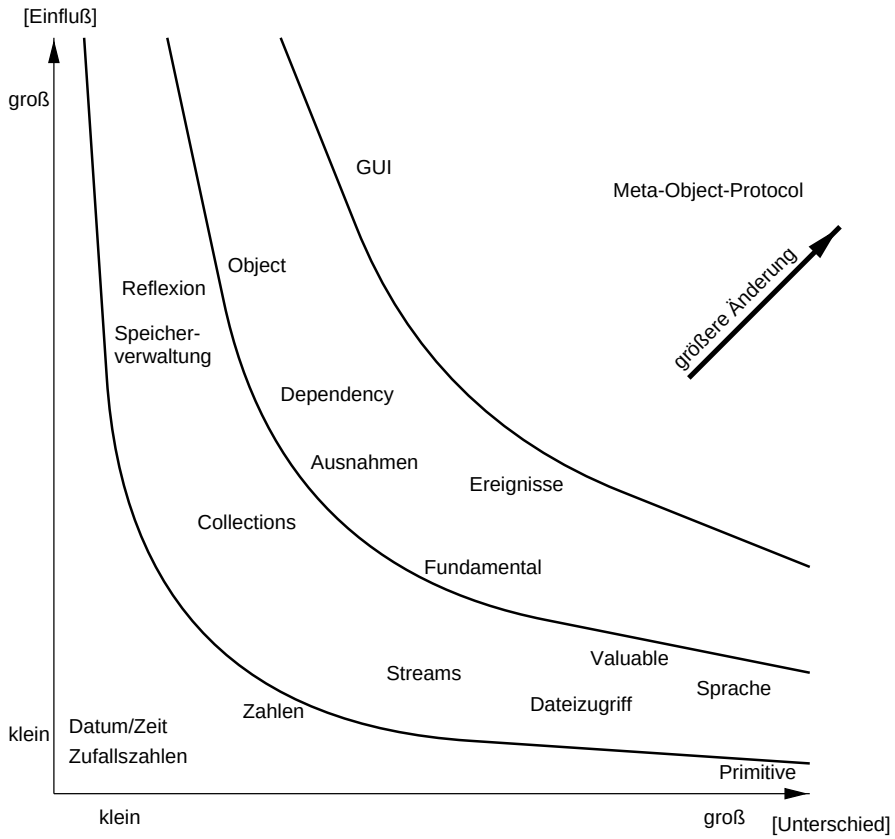


Abbildung 6.1: Architekturänderung durch Migration der Verwendung von Plattformelementen

Die Änderung der Applikationsarchitektur durch die Migration eines Plattformelements ergibt sich aus der Kombination der beiden Bewertungskriterien (siehe Abbildung 6.1). Je größer der Einfluß eines Plattformelements auf die Applikationsarchitektur und je größer die Unterschiede zwischen Smalltalk und Java, desto größer ist die Änderung der Architektur einer Applikation. Zwei Plattformelemente, die auf derselben Kurve liegen, führen zu einer vergleichbar großen Änderung der Architektur einer Applikation bei der Migration. Die in der Abbildung dargestellten Hyperbeln teilen die Plattformelemente in Äquivalenzklassen modulo „Grad der Änderung der Applikationsarchitektur“ ein. Die Positionierung der Elemente darf nicht absolut aufgefaßt werden. Sie soll vielmehr ein Ausdruck der Verhältnisse der Plattformelemente zueinander sein.

6.3 Zusammenfassung

Dieses Kapitel nährte sich dem Problem der Änderung der Applikationsarchitektur bei einer Migration auf zwei Wegen:

Zunächst wurden die Problemklassen, die bei der Migration der Muster gefunden wurden dahingehend untersucht, auf welche Art sie Einfluß auf die Änderung der Applikationsarchitektur bei der Migration nehmen. Anschließend wurde dargestellt, wie groß die Änderung der Applikationsarchitektur ist, die die Verwendung der verschiedenen Plattformelemente mit sich bringt.

Die Aufgabenstellung der Arbeit war, den Einfluß der Sprachplattform einerseits und der verwendeten bzw. induzierten Muster andererseits auf die Änderung der Applikationsarchitektur bei einer Migration darzustellen. Dies geschah in den beiden vorausgehenden Kapiteln. Dieses Kapitel trägt die daraus gewonnenen Erkenntnisse zusammen und stellt sie in konzentrierter Form dar. Wir haben damit eine im Rahmen der Aufgabenstellung erschöpfende Antwort gegeben.

7

Zusammenfassung

7.1 Gang der Untersuchung

Diese Arbeit vergleicht die Applikationsarchitekturen in Smalltalk und Java im Hinblick auf Migration. Das bedeutet, daß die Architektur einer Applikation nur insoweit betrachtet wird, wie sie für die Migration eine Rolle spielt. Den Begriff Applikationsarchitektur fassen wir als Kombination von Mustern auf verschiedenen Abstraktionsebenen auf. Die Gesamtarchitektur einer Applikation ergibt sich aus der Summe dieser Muster. Die Implementierungen dieser Muster sind abhängig von den Elementen der jeweiligen Sprachplattform. Die Sprachplattform nimmt also auch Einfluß auf die Architektur einer Applikation.

Speziell untersucht diese Arbeit, wie sich die Architektur einer Applikation durch die Migration ändert bzw. ändern muß, damit eine Applikation im Java–Stil entsteht.

Die angestellte Untersuchung teilt sich in mehrere Schritte auf:

1. Zunächst wurden die Elemente beider Plattformen miteinander verglichen. Der Vergleich verwendete eine möglichst herstellerunabhängige Strukturierung der Smalltalk–Plattform. Der Schwerpunkt bei diesem Vergleich lag darauf, wie die Funktionalität, die durch die Verwendung eines Plattformelements genutzt wird, in Java angeboten und im Stil von Java genutzt wird. Jedes Plattformelement wurde kurz vorgestellt, soweit es für die Darstellung der Migration nötig war. Das Element der Java–Plattform, welches die gleiche Funktionalität wie das betrachtete Element der Smalltalk–Plattform anbietet, wurde ebenfalls kurz erläutert. Anschließend erfolgte eine Beschreibung, wie die Verwendung eines Elements der Smalltalk–Plattform nach Java zu portieren ist. Falls mehrere gangbare Alternativen vorlagen, wurden diese aufgezeigt.

2. Danach wurde die Migration auf der Ebene von Mustern beschrieben. Muster sind Beschreibungen von wiederverwendbarem Design. Sie sind in einem Mustersystem angeordnet, welches die Muster anhand unterschiedlicher Abstraktionsgrade unterscheidet. Sehr eng mit der Sprachebene sind Idiome verknüpft. Sie stellen Problemlösungen dar, die sich im täglichen Umgang mit der Programmiersprache ergeben. Idiome sind für diese Arbeit interessant, weil sie die typische Verwendung der Sprachplattformen aufzeigen. Entwurfsmuster abstrahieren mehr von der Implementierung. Sie sind eine Form von Mikroarchitektur. Zum einen werden Entwurfsmuster von Applikationsentwicklern verwendet, um die Architektur einer Applikation zu gestalten. Zum anderen existieren sie in der vorhandenen Klassenbibliothek. Durch die Art ihrer Verwendung prägen sie einer Applikation eine Architektur auf. Architekturmuster schließlich dienen der Strukturierung der Applikation insgesamt. Die in einer Sprachplatt-

form vorhandenen Architekturmuster müssen verstanden werden, um die Elemente im Stil der Plattform zu verwenden.

Die Untersuchung einer Reihe von Mustern zeigte deren spezifische Implementierungen bzw. Realisierungen auf. Die Verwendung der Elemente einer Sprachplattform ermöglicht zum Teil eine sehr spezialisierte Implementierung eines Musters. Jede Musterimplementierung wurde dann daraufhin untersucht, wie eine Migration nach Java erfolgen könnte bzw. wie die Verwendung dieser Musterimplementierung migriert werden könnte. Diese Betrachtung der Mustermigration kommt einer Betrachtung der Migration eines Teils der Architektur nach Java gleich. Wichtig für die Darstellung war die klare Auflistung der Unterschiede, die eine Migration problematisch machen.

3. Anschließend wurden die Ergebnisse der beiden vorangehenden Schritte zusammengetragen. Es erfolgte zunächst eine Diskussion der Probleme, die bei der Mustermigration gefunden wurden und einer Darstellung des Einflusses des jeweiligen Problems auf die Applikationsarchitektur. Jede Verwendung eines Plattformelements führt bei der Migration zu einer mehr oder weniger starken Änderung der Applikationsarchitektur. Der Grad dieser Änderung wurde für die verschiedenen Plattformelemente bewertet und der Einfluß auf die Architektur diskutiert.

7.2 Beitrag neuen Wissens

Durch die vorliegende Arbeit wurden einige wichtige Erkenntnisse gewonnen hinsichtlich der Architekturänderungen, denen eine Applikation bei der Migration von Smalltalk nach Java unterliegt.

1. Diese Arbeit präsentiert eine Auflistung von konkreten Unterschieden zwischen Smalltalk und Java und diskutiert für jeden dieser Unterschiede, welchen Änderungen die Architektur einer Applikation bei der Migration nach Java dadurch unterliegt.

2. Diese Arbeit führt eine Reihe von Musterimplementierungen auf verschiedenen Abstraktionsebenen in beiden Sprachen auf, stellt sie einander gegenüber und diskutiert die Migration.

3. Diese Arbeit bewertet die verschiedenen Plattformelemente hinsichtlich des Änderungsbedarfs in der Applikationsarchitektur, der sich aus der Verwendung des jeweiligen Plattformelements ergibt.

4. Diese Arbeit gibt eine mögliche Strukturierung der Smalltalk–Plattform an, um über die Elemente der Sprache und der Klassenbibliothek sprechen zu können.

7.3 Offene Punkte

Der angestellte Vergleich geschah auf der Basis der — zum Zeitpunkt der Erstellung — aktuellen Versionen von IBM Smalltalk und Java. Insbesondere die rasche Evolution der Java-Plattform mit Hinzunahme weiterer Elemente (z.B. parametrische Typen) wird eine Fortführung dieser Arbeit erfordern.

Die in dieser Arbeit gegebenen Migrationshinweise wenden sich an einen menschlichen Leser, der die Migration „von Hand“ vornimmt. Im Zuge einer vollständigen Lösung müßte die Möglichkeit einer Werkzeugunterstützung dieser Migration weiter untersucht werden.

Auch in der nächsten Zukunft wird Smalltalk als Implementierungsplattform verwendet werden. Um eine mögliche spätere Migration nach Java zu erleichtern, könnte ein Architekturstil–Handbuch

erstellt werden. Dieses würde Richtlinien dafür geben, wie Smalltalk–Programme so geschrieben werden können, daß sie möglichst leicht nach Java zu migrieren sind.

Ein aktuelles Thema in der Forschung zur Software–Architektur sind Architekturbeschreibungssprachen. Es bleibt zu untersuchen, inwieweit diese Sprachen heutzutage schon dazu geeignet sind, die Unterschiede der Applikationsarchitekturen im Hinblick auf Migration zu beschreiben. Generatives Programmieren ist ebenfalls ein aktuelles Thema. Hier bleibt zu untersuchen, inwieweit die plattformabhängigen Elemente einer Applikation von Generatoren automatisch erzeugt werden können.

Schließlich behandelt diese Arbeit nur einen Teil der Migrationsproblematik. Eine reine Übersetzung von Smalltalk nach Java kann nicht alleine stehen, sondern muß in ein Gesamtkonzept zur Migration bzw. zum Reengineering eingebettet sein. Die vorliegende Arbeit stellt nur einen Teilaspekt einer kompletten Systemmigration dar.

7.4 Fazit

Auch wenn die Java– und Smalltalk–Plattformen viele Detailunterschiede aufweisen, so ist doch die Verwandtschaft zwischen beiden so eng, daß sich die Applikationsarchitekturen auf beiden Plattformen sehr ähnlich sind.

Durch die Migration einer Applikation ändert sich ihre Architektur. Das Ausmaß dieser Änderung ist abhängig von den verwendeten Elementen der Smalltalk–Plattform und die Art der Realisierung ihrer Funktionalität in Java. Insbesondere im Vergleich zu anderen objektorientierten Programmiersprachen erscheinen die Änderungen bei einer Migration von Smalltalk nach Java jedoch als eher gering. Allerdings ist die Java–Lösung eines Problems oft um einiges uneleganter als die Lösung des gleichen Problems in Smalltalk, auch wenn die Lösung direkt in Java erfolgt.

Allerdings wird es nicht möglich sein, jede *beliebige* in Smalltalk geschriebene Applikation automatisch zu konvertieren, sondern die Migrierbarkeit ist an einige Bedingungen geknüpft. Die intensive Verwendung von Elementen der Smalltalk–Plattform, die in Bezug auf Migration sehr problembehaftet sind (Bsp. Meta–Objekt–Protokoll), steht einer Migration meist im Weg. Diese Elemente kommen in gewöhnlichen Applikationen jedoch eher selten vor. Außerdem verläßt sich der hier vorgestellte Ansatz darauf, daß das zu migrierende Smalltalk–Programm dem Stil von Smalltalk entspricht. Dieser Stil kommt in Form von Musterimplementierungen zum Ausdruck.

Insgesamt bedeutet das, daß eine automatische Migration von Smalltalk nach Java, bei der ein wartbares und dem Stil der Java–Plattform entsprechendes Programm entsteht, in vielen Fällen machbar erscheint.

A

Migrations–Index

Dieses Kapitel enthält mehrere Indices, die bei der Migration eines Smalltalkprogramms nach Java helfen sollen. Jeder Index führt die Konstrukte auf, die in Smalltalkprogrammen zu finden sind. Die Einträge verweisen auf die Stellen in der Arbeit, die das aufgeführte Konstrukt verwenden könnten und die evtl. hilfreich bei der Migration eines Programms sind. Dies kann direkt die Diskussion eines Plattformelements sein oder die Migrationsbeschreibung eines Musters. Hierzu ist dieses Kapitel aufgeteilt in einen Methodenindex, einen Klassenindex, einen Konzeptindex und einen Index der Variablenpools.

A.1 Methodenindex

A.2 Klassenindex

A.3 Konzeptindex

A.4 Index der Variablenpools und globalen Variablen

B

Tabellen

Methode	Klasse	Java
`createApplicationShell`	`CwTopLevelShell`	`JFrame` bei einer Applikation, `JApplet` bei einem Applet
`createPopupShell`	`CwOverrideShell`	`JWindow` bietet ein Fenster ohne jede Dekoration. Damit kann `CwOverrideShell` nachgebaut werden.

Tabelle B.1: Creation convenience Methoden der Shell–Klassen

Methode	Klasse	Java
`createMessageDialog`	`CwMessageBox`	`JOptionPane`
`createErrorDialog`	`CwMessageBox`	`JOptionPane`
`createInformationDialog`	`CwMessageBox`	`JOptionPane`
`createQuestionDialog`	`CwMessageBox`	`JOptionPane`
`createWarningDialog`	`CwMessageBox`	`JOptionPane`
`createWorkingDialog`	`CwMessageBox`	`JOptionPane`
`createSelectionDialog`	`CwSelectionBox`	Kein direktes Pendant. Mit `JDialog` darstellbar.
`createPromptDialog`	`CwSelectionBox`	Kein direktes Pendant. Mit `JDialog` darstellbar.
`createBulletinBoardDialog`	`CwBulletinBoard`	`JDialog`
`createFormDialog`	`CwForm`	`JDialog`

Tabelle B.2: Creation convenience Methoden der Dialog–Klassen

Methode	Klasse	Java
createDrawnList	EwDrawnList	JList mit einer Implementierung des Interface ListCellRenderer
createFlowedIconList	EwFlowedIconList	JList mit einer Implementierung des Interface ListCellRenderer und entsprechendem LayoutManager
createIconArea	EwIconArea	Nicht vorhanden in Java.
createIconList	EwIconList	JList mit einer Implementierung des Interface ListCellRenderer
createIconTree		JTree, wobei der BasicTreeCellRenderer eine vergleichbare Funktionalität anbietet.
createPMNotebook	EwPMNotebook	JTabbedPane mit Umstellung des L&F durch den UIManager. Problem ist, daß Java keine Unterscheidung von *minor tabs* und *major tabs* und keine Unterteilung einer Seite in Teilseiten kennt.
createProgressBar	EwProgressBar	JProgressBar
createScrolledDrawnList	EwDrawnList	
createScrolled-FlowedIconList	EwFlowedIconList	JList mit einer Implementierung des Interface ListCellRendererund entsprechendem LayoutManager. Zusätzlich ein JScrollPane.
createScrolledIconArea	EwIconArea	Nicht vorhanden in Java.
createScrolledIconList	EwIconList	JList mit einer Implementierung des Interface ListCellRenderer. Zusätzlich ein JScrollPane.
createScrolledIconTree		JTree, wobei der BasicTreeCellRenderer eine vergleichbare Funktionalität anbietet. Zusätzlich ein JScrollPane.
createScrolledTableList	EwTableList	swing.table bietet ein eigenes Package zur Gestaltung von Tabellen. Zusätzlich ein JScrollPane.
createScrolledTableTree	EwTableTree	Nicht vorhanden in Java.
createSlider	EwSlider	JSlider, allerdings werden keine Zahlen auf der Skala angezeigt.
createSpinButton	EwSpinButton	Nicht vorhanden in Java.
createSplitWindow	EwSplitWindow	JSplitPane, allerdings kann ein JSplitPane nur zwei Komponenten aufnehmen. Jedoch kann durch Schachtelung mehr dargestellt werden.
createTableList	EwTableList	swing.table bietet ein eigenes Package zur Gestaltung von Tabellen.
createTableTree	EwTableTree	Nicht vorhanden in Java.
createToolBar	EwToolBar	JToolBar. Standardmäßig ist eine Toolbar *floatable*, was jedoch ausgeschaltet werden kann.
createWINNotebook		JTabbedPane mit dem standardmäßig verwendeten Renderer.

Tabelle B.4: Creation convenience Methoden der erweiterten Widget–Klassen

Methode	Klasse	Java
createArrowButton	CwArrowButton	BasicArrowButton, MetalScrollButton oder OrganicScrollButton. Allerdings legt man sich damit auf ein L&F fest. Wird eigentlich nur indirekt verwendet (z.B. in einer JScrollBar).
createBulletinBoard	CwBulletinBoard	Wird kaum verwendet. Kein direktes Pendant in Swing. Nachzubilden mit JOptionPane.
createCascadeButton	CwCascadeButton	JMenu
createComboBox	CwComboBox	JComboBox
createDialogShell	CwDialogShell	
createDrawingArea	CwDrawingArea	
createDrawnButton	CwDrawnButton	Nicht direkt unterstützt. Realisierbar durch eigene Implementierung von ButtonUI.
createForm	CwForm	Eine Implementierung von LayoutManager muß als Layout–Manager beim Vater–Widget installiert werden.
createFrame	CwFrame	Mit swing.border steht ein komplette Package zum Gestalten von Rahmen zur Verfügung.
createLabel	CwLabel	JLabel
createList	CwList	JList
createMainWindow	CwMainWindow	JFrame bzw. JApplet faßt die CwMainWindow und CwTopLevelShell zusammen.
createMenuBar	CwRowColumn	JMenuBar
createOptionMenu	CwRowColumn	JMenu
createPopupMenu	CwRowColumn	JPopupMenu
createPulldownMenu	CwRowColumn	JMenu
createPushButton	CwPushButton	JButton
createRadioBox	CwRowColumn	ButtonGroup mit Instanzen von JRadioButton als Elementen.
createRowColumn	CwRowColumn	Eine Implementierung von LayoutManager muß als Layout–Manager beim Vater–Widget installiert werden.
createScale	CwScale	JSlider
createScrollBar	CwScrollBar	JScrollBar innerhalb eines JScrollPane
createScrolledList	CwList	JList innerhalb eines JScrollPane
createScrolledText	CwText	Mit swing.text steht ein komplette Package zur Behandlung von Texten zur Verfügung. Allerdings muß noch eine JScrollPane eingebaut werden.
createScrolledWindow	CwScrolledWindow	JScrollPane
createSeparator	CwSeparator	Kein eigenes Objekt. Stattdessen durch die Methode addSeparator zu erzeugen.

Tabelle B.6: Creation convenience Methoden der Widget–Klassen

Methode	Klasse	Java
createSimpleCheckBox	CwRowColumn	ButtonGroup
createSimpleMenuBar	CwRowColumn	JMenu
createSimpleOptionMenu	CwRowColumn	JMenu
createSimplePopupMenu	CwRowColumn	JPopupMenu
createSimplePulldownMenu	CwRowColumn	JMenu
createSimpleRadioBox	CwRowColumn	ButtonGroup
createText	CwText	Mit swing.text steht ein komplette Package zur Behandlung von Texten zur Verfügung.
createToggleButton	CwToggleButton	JToggleButton
createWorkArea		
createMessageBox	CwMessageBox	JDialog oder JOptionPane
createSelectionBox	CwSelectionBox	JDialog oder JOptionPane

Tabelle B.8: Creation convenience Methoden der Widget–Klassen (Forts.)

Smalltalk–Event	Java–Event	Listener–Methode
ButtonPress	MouseEvent	mousePressed()
ButtonRelease	MouseEvent	mouseReleased()
Expose	ComponentEvent	componentShown()
KeyPress	KeyEvent	keyPressed()
KeyRelease	KeyEvent	keyReleased()
MotionNotify	MouseEvent	mouseMoved()

Tabelle B.9: Abbildung von Smalltalk–Events auf Java

Smalltalk–Klasse	Java–Klasse
CwMessagePrompter	JOptionPane
CwTextPrompter	JOptionPane
CwFileSelectionPrompter	JDialog

Tabelle B.10: Gegenüberstellung der Prompter–Klassen

Smalltalk–Callback	Java–Event	Listener–Methode
activate	ActionEvent	actionPerformed()
apply	ActionEvent	actionPerformed()
arm	MouseEvent	mouseEntered()
browseSelection	ItemEvent	itemStateChanged()
cancel	WindowEvent	windowClosed()
cascading		
decrement	AdjustmentEvent	adjustmentValueChanged()
defaultAction	ActionEvent	actionPerformed()
destroy	WindowEvent	windowClosing()
disarm	MouseEvent	mouseExited()
drag	AdjustmentEvent	adjustmentValueChanged()
dragDetected	MouseEvent	mouseDragged()
expose	ComponentEvent	componentShow()
extendedSelection	ItemEvent	itemStateChanged()
focus	FocusEvent	focusGained()
help		
iconify	WindowEvent	windowIconified()
increment	AdjustmentEvent	adjustmentValueChanged()
input	KeyEvent	keyTyped()
	MouseEvent	mousePressed()
interceptExpose	ComponentEvent	componentShow()
losingFocus	FocusEvent	focusLost()
map	WindowEvent	windowOpened()
modifyVerify	TextEvent	textValueChanged()
multipleSelection	ItemEvent	itemStateChanged()
noMatch		
ok	ActionEvent	actionPerformed()
pageDecrement	AdjustmentEvent	adjustmentValueChanged()
pageIncrement	AdjustmentEvent	adjustmentValueChanged()
popdown		
popup		
resize	ComponentEvent	componentResized()
simple		
singleSelection	ItemEvent	itemStateChanged()
toBottom	AdjustmentEvent	adjustmentValueChanged()
toTop	AdjustmentEvent	adjustmentValueChanged()
unmap	WindowEvent	windowClosed()
valueChanged	ItemEvent	itemStateChanged()
windowClose	WindowEvent	windowClosed()

Tabelle B.12: Abbildung von Smalltalk–Callbacks auf Java

C

Problemübersicht

Die unten abgebildete Liste ist eine Zusammenfassung der in Kapitel 6 behandelten Probleme. Die hier aufgeführten Probleme wurden durch die Betrachtung der Migration von Mustern (Patterns) auf verschiedenen Abstraktionsebenen gefunden. Diese Liste erhebt nicht den Anspruch, jedes beliebige Migrationsproblem aufzuführen, sondern ist unter dem Aspekt der Auswahl der untersuchten Muster zu sehen. Wir haben jedoch versucht, diese Auswahl so breit wie möglich anzulegen.

Problem 1	Dynamische Typisierung in Smalltalk im Gegensatz zu statischer Typisierung in Java.
Problem 2	Java unterscheidet primitive Typen und Referenztypen.
Problem 3	In Java sind Blöcke keine Objekte erster Klasse wie in Smalltalk.
Problem 4	Sehr mächtiges Meta-Objekt-Protokoll in Smalltalk im Vergleich zur Reflection–API in Java.
Problem 5	Instanzvariablen sind in Smalltalk privat.
Problem 6	Die Sichtbarkeit von Methoden kann in Smalltalk nicht eingeschränkt werden.
Problem 7	In Smalltalk ist kein Sprachkonstrukt zum Erzeugen von Instanzen definiert.
Problem 8	In Smalltalk ist kein Sprachkonstrukt zum Initialisieren von Variablen definiert.
Problem 9	Basisklassen können in Smalltalk geändert werden, in Java dagegen nicht.
Problem 10	In Smalltalk sind keine Interfaces definiert.
Problem 11	In Java sind keine Klasseninstanzvariabeln definiert.
Problem 12	Der Abhängigkeits–Mechanismus ist in Java nicht im Objekt–Framework enthalten.
Problem 13	Listener–Interface schreibt Namen der Behandlermethoden in Java vor.
Problem 14	Klassenobjekte bleiben in Smalltalk über einen Neustart des Systems hinweg erhalten.
Problem 15	In Java sind keine internen Iteratoren definiert.
Problem 16	Smalltalk unterscheidet zwischen `shallowCopy`
Problem 17	In Java ist keine Objekt–Mutation definiert.
Problem 18	Ein Objekt kann in Java nicht von `null`
Problem 19	In Smalltalk sind Instanzen der Stream–Klassen freier positionierbar als in Java.

D

Glossar

Aggregation Die Zusammenfassung mehrerer Objekte durch ein einziges.

API (Application programming interface) Spezifikation einer Programmierschnittstelle.

Applikationsmodell Modelliert das Verhalten des Benutzers im Umgang mit den Domänenobjekten.

Bytecode Der vom Java–Compiler erzeugte Zwischencode.

Collection Eine Sammlung von Objekten. Eine Collection paßt ihre Kapazität gewöhnlich automatisch an.

Delegation Mechanismus zur Wiederverwendung von bestehenden Objekten. Damit wird die Verantwortung für die Ausführung eines Methodenaufrufs an ein anderes Objekt übertragen. Beispielsweise delegieren Klassen, die selbst Elemente enthalten, die darauf bezogenen Aufrufe meist an die verwendete Collection.

Domänenobjektmodell Modell der Objekte des Problembereichs. Wird oft verwendet, um die Spezifikation zusammen mit den Fachexperten zu erstellen.

Downcast Durch eine explizite Typumwandlung wird der Typ eines Objekts auf den einer Unterklasse spezialisiert. Das Objekt muß eine Instanz dieser Unterklasse sein, auch wenn zuvor nur über das Interface der Oberklasse zugegriffen wurde.

Embedding Um ein Element, welches kein Objekt ist, als Objekt verwenden zu können, wird es von einem Wrapper–Objekt gekapselt.

Entwurfsmuster (Design Pattern) Beschreibung eines ständig wiederkehrenden Designproblem samt einer Auflistung der Randbedingungen und einer Lösung.

Fixed class In Smalltalk eine Klasse, die ausschließlich solche Variablen enthält, die mit Namen versehen sind. Ggs. zu Variable class.

Framework Eine Technik, um Wiederverwendung in objekt-orientierten Programmiersprachen zu realisieren. Ein wiederverwendbares Design in Form abstrakter Klassen bzw. eine generische Applikation.

Getter-Methode Einen Methode, die einen lesenden Zugang zu einer Variablen von außerhalb der Instanz erlaubt. In Smalltalk wird damit das Problem behoben, daß Instanzvariablen immer privat sind.

Hollywood–Prinzip Umkehrung des Kontrollflusses in einem Framework. Die Applikation gibt die Kontrolle über den Kontrollfluß an das Framework ab und reagiert nur noch auf Ereignisse des Frameworks.

Hot Spot Die Stelle in einem Framework, die zur Erweiterung durch eine Applikation vorgesehen ist.

Image In Smalltalk eine Datei, die alle instanziierten Objekte enthält. Dadurch sind nach dem Neustart des Systems alle zuvor erzeugten Objekte wieder verfügbar ohne Zutun der Applikation.

Information hiding Ein Objekt wird als black-box betrachtet - interne Aspekte sind nach außen nicht sichtbar. Nach außen ist lediglich das öffentliche Interface zu sehen, während die interne Repräsentation verdeckt bleibt[WBWW90, S.18]. Siehe Kapselung.

Instanzvariable (instance variable) Jede Instanz einer Klasse erhält ein eigenes Exemplar einer Instanzvariable. Das Scope einer Instanzvariablen ist im Normalfall die Instanz.

Iterator Wird verwendet, um über den Elementen einer Sammlung zu iterieren. Interne Iteratoren bearbeiten in einem Durchlauf alle Elemente der Sammlung, ohne dem Benutzer eine Möglichkeit zum Eingreifen zu geben. Externe Iteratoren werden in einer externen Schleife verwendet. Schmalltalk kennt beide Arten, Java nur externe Iteratoren.

Just–in–time–Compiler Übersetzt vor einem Methodenaufruf den Zwischencode der Methode in Objektcode des Prozessors. Dient der Verbesserung der Performance.

JVM (Java Virtual Machine) Interpreter für den vom Java–Compiler erzeugten Zwischencode (Bytecode).

Kapselung (encapsulation) Das Zusammenfassen des Zustands und des Verhaltens im Objekt, um die Integrität der Daten garantieren zu können.

Klassenvariable (class variable) Eine Variable, die sichtbar ist für die Klasse selbst und jede Instanz dieser Klasse. Hiermit können Daten zwischen allen Instanzen der Klassen und ihrer Unterklassen ausgetauscht werden. (Vgl. [Lew95, S.29])

Klasseninstanzvariable (class instance variable) Variablen dieser Art sind nur direkt für die Klasse — nicht für ihre Instanzen — sichtbar. Jede Unterklasse bekommt eine eigene Kopie der Klasseninstanzvariablen. (Vgl. [Lew95, S.29])

Klassenobjekt Der Repräsentant einer Klasse. Wird in Smalltalk bei Klassendefinition automatisch angelegt. In Java muß das Klassenobjekt explizit erfragt werden.

Komponente Eine Technik zur Realisierung von Wiederverwendung. In objekt-orientierten Sprachen besteht eine Komponente meist aus einer Gruppe von Objekten, die über eine einheitliche Schnittstelle nach außen verfügen.

Late binding Erst beim Versenden einer Nachricht ermittelt das System die Methode, welche auf die Nachricht reagiert.

Lightweight UI Framework Ein Framework in Java, mit dem die Widgets von Swing implementiert sind. Darin übernimmt Java selbst das Zeichnen der Widgets.

Metainformationen Informationen über die Klassen und Klassenstruktur, die von einer Applikation erfragt werden können.

Model–View–Controller siehe MVC.

MVC Die Aufteilung einer Applikation bzw. der Oberflächenelemente in die Komponenten Model, View und Controller. Das Model enthält die Daten, die View sorgt für die Darstellung und der Controller verarbeitet Benutzereingaben.

Nachricht (message) Die Kommunikationseinheit, die an ein Objekt oder eine Klasse gesendet wird.

Objekt-Komposition Methode zur Wiederverwendung von Funktionalität eines anderen Objekts. Methodenaufrufe werden dabei an das wiederverwendete Objekt delegiert.

Package Sprachmechanismus in Java zum Strukturieren und Steuern der Sichtbarkeit der Klassen eines Systems.

Polymorphie (polymorphism) Polymorphie ist die Eigenschaft, daß ein Name Objekte vieler verschiedener Klassen bezeichnen kann.

Poolvariable siehe Variablenpool

Refaktorierung Verändern eines Programmes bei Erhalt der Funktionalität zum Verbessern der Lesbarkeit, Wartbarkeit, Erweiterbarkeit, etc.

Überschreiben (overriding) Durch Neudefinition in der Subklasse ist die Definition einer überschriebenen Methode nicht mehr implizit sichtbar, da sie von der lokalen Neudefinition verdeckt wird. Um auf die ursprüngliche Methode zuzugreifen, muß die Superklasse explizit referenziert werden.

Unicode ist eine universelle Zeichenkodierung, die vom *Unicode Consortium* definiert wurde und die als Nachfolger von ASCII propagiert wird.

Variable class In Smalltalk eine Klasse, deren Instanzvariablen über einen Index angesprochen werden. Ggs. zu Fixed class.

Variablenpool Ein Variablenpool ist in Smalltalk eine Gruppe an Variablen, die unter einem Namen anprechbar sind. Bei der Definition einer Klasse muß angegeben werden, welche Pools verwendet werden sollen. Der Pool `Smalltalk` wird standardmäßig immer verwendet. (Vgl. [GR83, S.47])

Vererbung Methode zur Wiederverwendung von Funktionalität, die die Superklasse anbietet. Diese Funktionalität kann in Form von Kode, Variablen oder einer Signatur erfolgen.

Wiederverwendung (reuse) Oberbegriff für alle Techniken die darauf abzielen, vorhandenen Code und bestehendes Design in neuen Programmen wiederzuverwenden. Manchmal auch als Verwendung bezeichnet.

Literaturverzeichnis

[ABW98] Alpert, Brown, and Woolf. *The Design Patterns Smalltalk Companion.* Addison-Wesley, Reading, MA, 1998.

[ANS97] ANSI. *Draft American National Standard for Information Systems – Programming Languages – Smalltalk.* 1997. Revision 1.9.

[Bec97] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[Ben98] Daimler Benz. Babylon – Vergleich von vier objektorientierten bzw. objektbasierten Programmiersprachen, 1998. Interne Projektdokumentation.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architectures. In *ECOOP Conference Proceedings*, pages 139–149. Springer-Verlag, 1994.

[BM95] Frank Buschmann and Regine Meunier. A system of patterns. In James O. Coplien and Douglas C. Schmidt, editors, *Patterns Languages of Program Design*, 1995.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns.* Wiley, West Sussex, England, 1996.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, second edition, 1994.

[Cop96] James O. Coplien. *Software Patterns.* SIGS Books & Multimedia, New York, 1996.

[Cox90] B. Cox. There is a silver bullet. *BYTE Magazine*, Oktober 1990.

[Deu89] L. Peter Deutsch. Design Reuse and Frameworks in the Smalltalk-80 Programming System. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume II. ACM Press, 1989.

[Eck98] Bruce Eckel. *Thinking in Java.* Prentice Hall PTR, Upper Saddle River, NJ, 1998.

[FE95] Donald G. Firesmith and Edward M. Eykholt. *Dictionary of Object Technology – The Definitive Desk Reference.* SIGS Books, New York, 1995.

[Fla97] David Flanagan. *Java in a Nutshell – A Desktop Quick Reference.* O'Reilly, Sebastopol, CA, second edition, 1997.

[Fow97] Martin Fowler. *UML Distilled: Applying the standard object modelling language.* Addison-Wesley, Reading, MA, 1997.

[FRS96] Peter Fingar, Dennis Read, and Jim Stikeleather. *Next Generation Computing — Distributed Objects for Business.* SIGS Books & Multimedia, New York, 1996.

[Gam97] Erich Gamma, 1997. Persönliche Kommunikation.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[GM95] James Gosling and Henry McGilton. The Java Language Environment – A White Paper, 1995. http://www.javasoft.com/docs/white/index.html.

[Gmb97] ObjectART Software GmbH. Smalltalk–to–Java–Converter, 1997. http://home.t-online.de/home/ObjectART#St2J.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Gra97] Mark Grand. *Java – Language Reference.* O'Reilly, Sebastopol, CA, second edition, 1997.

[IBM97] IBM. *IBM Smalltalk Programmers's Reference, Version 4.0.* 1997.

[JCJO92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2), Juni/July 1988.

[Joh92] Ralph Johnson. Documenting frameworks using patterns. In *OOPSLA Conference Proceedings*, 1992. In SIGPLAN Notices.

[Joh97] Ralph Johnson. Frameworks = components + patterns. *CACM*, 40(10), Oktober 1997.

[Kap95] Andreas Ch. Kapp. Vergleich der Designarchitektur zweier Smalltalksysteme – Konsequenzen der getroffenen Designentscheide. Master's thesis, Universität Basel, 1995.

[KP88] Krasner and Pope. A cookbook for using the MVC UI paradigm in Smalltalk-80. *JOOP*, August/September 1988.

[Kra97] Tom Krauß. GUI–Frameworks in Smalltalk und Java. In *STJA 97 Tagungsband*, 1997. TU Illmenau.

[Kra98] Timo Krauß. Internet und Java im Smalltalk–Kontext, 1998. http://www.gebit.de/GebitHomePage/forum/internet.htm.

[Lew95] Simon Lewis. *The Art and Science of Smalltalk – An Introduction to Object-Oriented Programming using Visual Works.* Prentice Hall, London, 1995.

[Mar91] Bruce Martin. The separation of interface and implementation in C++. In *Proceedings of the 1991 USENIX C++ Conference*, pages 51–63, Washington D.C., April 1991. USENIX Association.

[Obj97] ObjectShare. Classic Blend, 1997. http://www.arscorp.com/cgosi.htm.

[Opd97] Opdyke. *Refactoring object–oriented frameworks.* PhD thesis, University of Illinois at Urbana–Champaign, 1997. ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z.

[Ori97] Orisa. Smalltalk–nach–Java–Konverter, 1997. http://www.orisa.de/orisa/Projects/s2jfull.D.html.

[OW97] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly, Sebastopol, CA, 1997.

[Ple97] Jonathan Pletzke. *Advanced Smalltalk.* John Wiley & Sons Inc., New York, 1997.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, Wokingham, England, 1995.

[Sch97] Hans Albrecht Schmid. Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10), Oktober 1997.

[SG96] M. Shaw and D. Garlan. *Software Architecture: A Perspective on an Emerging Discipline.* Prentice Hall, 1996.

[Sof98] Java Soft. JDK 1.2 Beta 3 Documentation, 1998. http://www.javasoft.com.

[VC97] V.Srivivasan and Daniel Chang. Object persistence in object-oriented applications. *IBM Systems Journal*, 36(1), 1997.

[WBWW90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ, 1990.