Der *JAVA – COACH*

— Modellieren mit UML, Programmieren mit JDK, Dokumentieren mit HTML —

Hinrich E. G. Bonin[1]

10. November 1998

[1]Prof. Dr. rer. publ. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. Bonin, Fachhochschule Nordostniedersachsen, Volgershall 1, D-21339 Lüneburg, http://as.fh-lueneburg.de/.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Vorwort

Unter den Fehlleistungen der Programmierung wird ständig und überall gelitten: Zu kompliziert, zu viele Mängel, nicht übertragbar, zu teuer, zu spät und so weiter. Nun soll es Java richten (*The Java Factor*[1]). Unser Hoffnungsträger basiert auf einer konsequent objekt-orientierten Programmierung. Sie soll die gewünschte Qualität ermöglichen. Dabei wird Qualität durch die Leistungsfähigkeit, Zuverlässigkeit, Durchschaubarkeit & Wartbarkeit, Portabilität & Anpaßbarkeit, Ergonomie & Benutzerfreundlichkeit und Effizienz beschrieben. Der $\mathcal{JAVA}\Leftrightarrow\mathcal{COACH}$ schwärmt wohl vom Glanz der „Java-Philosophie“, ist aber nicht euphorisch eingestimmt. Es wäre schon viel erreicht, wenn Sie, liebe Programmiererin, lieber Programmierer, nach dem Arbeiten mit diesem Buch Java als ein Akronym für *Just a valid application* betrachten können.

1.1 Java ↪ *Just a valid application*

Der $\mathcal{JAVA}\Leftrightarrow\mathcal{COACH}$ wendet sich an alle, die auf einer fundierten Theoriebasis Schritt für Schritt anhand von praxisnahen Beispielen Java gründlich verstehen wollen. Im Mittelpunkt steht das objekt-orientierte Programmieren. Dazu muß der Beginner viele Konstrukte erlernen und bewährte Konstruktionen nachbauen. Im Rahmen der Modellierung wird die „Benutzermaschine“ mit Hilfe der *Unified Modeling Language* spezifiziert. Die „Basismaschine“ wird natürlich in Java notiert. Die Dokumentation der Software erfolgt als Hypertext (HTML 4.0).

Wizard

Ebensowenig wie zum Beispiel Autofahren allein aus Büchern erlernrbar ist, wird niemand zum „Java-Wizard“ (deutsch: Hexenmeister) durch Nachlesen von erläuterten Beispielen. Das intensive Durchdenken der Beispiele im Dialog mit einem lauffähigen *Java Development Kit* (JDK) vermittelt jedoch, um im Bild zu bleiben, unstrittig die Kenntnis der Straßenverkehrsordnung und ermöglicht ein erfolgreiches Teilnehmen am Verkehrsgeschehen — auch im Großstadtverkehr. Für diesen Lernprozeß wünsche ich der „Arbeiterin“ und dem „Arbeiter“ viel Freude.

Der $\mathcal{JAVA}\Leftrightarrow\mathcal{COACH}$ ist konzipiert als ein Buch zum *Selbststudium*

[1] Titelüberschrift von *Communications of the ACM*, June 1998, Volume 41, Number 6.

Quelle: [ITS97], Seite 134.

Abbildung 1.1: Java — wunderschöne Insel Indonesiens

und für *Lehrveranstaltungen*. Mit den umfassenden Quellenangaben und vielen Vor- und Rückwärtsverweisen dient es auch als Nachschlagewerk und Informationslieferant für Spezialfragen.

Der $\mathcal{JAVA} \Leftrightarrow \mathcal{COACH}$ vermittelt mehr als nur einen Einstieg. Er befaßt sich eingehend mit *Multithreading, Inner Classes, Serialization, Reflection* und *Remote Method Invocation*. Erläutert wird die Integration eines objekt-orientierten Datenbanksystems am Beispiel POET Version 5.0. Der $\mathcal{JAVA} \Leftrightarrow \mathcal{COACH}$ folgt nicht den üblichen Einführungen in eine (neue) Programmiersprache. Diese beginnen meist unmittelbar mit dem obligatorischen Beispiel „Hello World“[2]. Zunächst werden eingehend die prägenden Ideen und Konzepte von Java erläutert. Damit Java-Programme überhaupt in der gewünschten Qualität erzeugbar sind, müssen die „richtigen“ Objekte und deren „richtigen“ Beziehungen aus der Anwendungswelt erkannt und präzise notiert werden. Deshalb erklärt der $\mathcal{JAVA} \Leftrightarrow \mathcal{COACH}$ vorab die „UML-Boxologie“ ehe das Buch von Java-Quellcode-Beispielen bestimmt wird.

[2] Dieses Beispiel befindet sich im $\mathcal{JAVA}-\mathcal{COACH}$ erst im Abschnitt 5.1.1 auf Seite 60.

1.2 Java überall: Von der *ChipCard* bis zum *Host*

Java ist eine wunderschöne Insel Indonesiens mit exotischer Ausstrahlung. Sie gehört zu den vulkanreichsten Ländern der Erde. Im Zeitalter des Massentourismus ist Java für Kosten erreichbar, die ungefähr für eine Lizenz einer Java-Entwicklungsumgebung aufzubringen sind. In der amerikanischen Umgangssprache ist Java auch ein Synonym für „starker Bohnenkaffee“. Das Projektteam um Bill Joy und James Gosling (Sun Microsystems, Inc. USA) war wohl kaffeesüchtig, als es seinen *Object Application Kernel* (OAK) urheberrechtlich bedingt einfach in Java umtaufte [Hist97]. Sun beschreibt Java, ursprünglich entwickelt zur Steuerung von Haushaltsgeräten, als eine Programmiersprache mit vielen guten Eigenschaften (→Tabelle 1.1 auf Seite 13).

Java ist eine:
• „einfache,
• objektorientierte,
• verteilte,
• interpretierte,
• robuste,
• sichere,
• architektur-unabhängige,
• portable,
• hochperformante,
• multithread-fähige und
• dynamische Sprache“ [Arnold/Gosling96, JavaSpec].

Tabelle 1.1: Java-Beschreibung von Sun Microsystems, Inc. USA

Java ist weit mehr als eine von `C++` abgeleitete, bessere objekt-orientierte Programmiersprache, auch wenn Java scherzhaft als „C plus-plus-minus-minus“ (Bill Joy zitiert nach [Orfali/Harkey97]) bezeichnet wird (→Tabelle 1.2 auf Seite 14).

JavaCard	Embedded Java	PersonalJava	Java
Java Platform APIs			
JavaCard Virtual Machine JavaCard Classe ISO Support Classes	Java Virtual Machine Required Java Classes Java IO	Java Virtual Machine Required Java Classes Java IO Java Beans Java Applet Networking Personal AWT	Java Virtual Machine Java Classes Java IO Java Beans Java Applet Networking AWT Security (Advanced) RMI JDBC
Memory & Processor Targets			
ROM = 16 KB RAM = 521 Bytes CPU: 8/16 Bit	ROM < 512 KB RAM = 256–512 KB CPU: 32 Bit,≈25 MHz	ROM < 2 MB RAM = 1 MB CPU: 32 Bit,≈ 50 MHz	ROM ≈4–8 MB RAM > 4 MB CPU: 32 Bit > 100 MHz
Target Application Environments			
SmartCards Rings	Industrial Controllers & Instrumentation Automation Low-end phones	Set-Top Boxes Screen Phones Mid-range mobile Phones Advanced automotive AV Systems	Desktop Operation Systems Enterprise Servers

Legende: Quelle [Sun98] Seite 16
`API` ≡ Application Programming Interface

Tabelle 1.2: Java Varianten

1.3 Notation

In diesem Buch wird erst gar nicht der Versuch unternommen, die weltweit übliche Informatik-Fachsprache Englisch zu übersetzen. Es ist daher teilweise „mischsprachig“: Deutsch und Englisch.

Aus Lesbarkeitsgründen sind nur die männlichen Formulierungen genannt; die Leserinnen seien implizit berücksichtigt. So steht das Wort „Programmierer“ hier für Programmiererin und Programmierer.

UML1.1

Für die Notation des („benutzernahen“) Modells einer Anwendung wird UML (**Unified Modeling Language** [Rational97]), Version 1.1 (September 1997), genutzt. UML ist eine Zusammenführung der Methoden von Grady Booch, James Rumbaugh und Ivar Jacobsen. UML ist eine Sprache für die Spezifikation, Visualisierung, Konstruktion und Dokumentation von „Artifakten“[3] eines Softwaresystems. UML wurde bei der OMG[4] zur Standardisierung eingereicht ([Oestereich97] S. 20).

Für die Notation von („maschinennahen“) Modellen beziehungsweise Algorithmen wird auch im Text Java verwendet. Beispielsweise wird zur Kennzeichnung einer Prozedur (Funktion) — also eines „aktivierbaren“ (Quellcode-)Teils — eine leere Liste an den Bezeichner angehängt, zum Beispiel `main()`.

HTML4.0

Zur Beschreibung der Strukturen in Dokumenten werden **HTML**[5]-Konstrukte verwendet. Die Layout-Spezifikation erfolgt mit Hilfe von *Cascading Style Sheets* (CSS).

Ein Programm (Quellcode) ist in der Schriftart `Typewriter` dargestellt. Die ausgewiesenen Zeilennummern in einer solchen Programmdarstellung sind kein Bestandteil des Quellcodes. Sie dienen zur Vereinfachung der Erläuterung.

Während der Arbeit am Buch lernt man erfreulicherweise stets dazu. Das hat jedoch auch den Nachteil, daß man laufend neue Unzulänglichkeiten am Manuskript erkennt. Ich bitte Sie daher im Voraus um Verständnis für Unzulänglichkeiten. Willkommen sind Ihre konstruktiven Vorschläge, um die Unzulänglichkeiten Schritt für Schritt weiter zu verringern.

PS: Das kleine *Smiley* hinter der email-Adresse am Ende einer Seite bedeutet *grin - basic smiley* und soll mit dazubeitragen, die Dinge im richtigen Licht zu sehen. Mehr zu dieser kleinen Tradition von *Smileys* aus der Zeit der ASCII-Zeichen und der Zweifingertipper siehe beispielsweise:
`http://www.infowork.be/smileys.htm`

[3] Als Artifakt wird das durch menschliches Können Geschaffene, das Kunsterzeugnis bezeichnet. Artifakt ist auch das Werkzeug aus vorgeschichtlicher Zeit, das menschliche Bearbeitung erkennen läßt.

[4] OMG ≡ Object Management Group

[5] HTML ≡ Hypertext Markup Language

Kapitel 2

Java: Eine Welt voller Objekte

Die objekt-orientierte Denkwelt von Java schöpft ihre Ideen aus schon lange bekannten Konzepten. Zum Beispiel aus dem Konzept des Polymorphismus (generische Funktion) und der daten- oder mustergesteuerten Programmierung. Stets geht es dabei um das Meistern von komplexen Systemen mit angemessenem wirtschaftlichen Aufwand. Die Objekt-Orientierung zielt auf zwei Aspekte:

1. Komplexe Software soll erfolgreich konstruierbar und betreibbar werden.
   ↪ Qualität, Validität
2. Die Erstellungs- und Wartungskosten von komplexer Software soll gesenkt werden.
   ↪ Wirtschaftlichkeit

Dies soll primär durch eine bessere Verstehbarkeit der Software erreicht werden. Transparenter wird die Software, weil sie unmittelbar die Objekte aus dem Anwendungsfeld in Objekte des Quellcodes abbildet.

Trainingsplan

Das Kapitel „Java: Eine Welt von Objekten“ erläutert:

- das Paradigma der Objekt-Orientierung,
  ↪ Seite 18 ...
- die Wurzelen der Objekt-Orientierung,
  ↪ Seite 20 ...
- die Ausrichtung von objekt-orientierten Programmiersprachen und
  ↪ Seite 22 ...

- skizziert prädestinierte Anwendungsfelder für die Entwicklung von objekt-orientierter Software.
↪ Seite 22 ...

2.1 Denkwelt der Objekt-Orientierung

Bei der Entwicklung einer Anwendung ist Software in einer ausreichenden Qualität zu erstellen. Dabei wird die Qualität von Software bestimmt durch ihre:

1. *Leistungsfähigkeit*,
   das heißt, das Programm erfüllt die gewünschten Anforderungen.

2. *Zuverlässigkeit*,
   das heißt, das Programm arbeitet auch bei ungewöhnlichen Bedienungsmaßnahmen und bei Ausfall gewisser Komponenten weiter und liefert aussagekräftige Fehlermeldungen (Robustheit),

3. *Durchschaubarkeit & Wartbarkeit*,
   das heißt, das Programm kann auch von anderen Programmierern als dem Autor verstanden, verbessert und auf geänderte Verhältnisse eingestellt werden,

4. *Portabilität & Anpaßbarkeit*,
   das heißt, das Programm kann ohne großen Aufwand an weitere Anforderungen angepaßt werden

5. *Ergonomie & Benutzerfreundlichkeit*,
   das heißt, das Programm ist leicht zu handhaben,

6. *Effizienz*,
   das heißt, das Programm benötigt möglichst wenig Ressourcen.

Aufgrund der langjährigen Fachdiskussion über die Innovation *Objekt--Orientierung*[1], wollen wir annehmen, daß dieses Paradigma[2] (≈ Denkmodell), etwas besseres ist, als das *was die Praktiker immer schon gewußt und gemacht haben*. Damit stellt sich die Kernfrage: Wenn das objekt-orientierte Paradigma die Lösung ist, was ist eigentlich das Problem? Des Pudels Kern ist offensichtlich das Unvermögen, komplexe Systeme mit angemessenem

Komplexität

[1] Das Koppelwort *Objekt-Orientierung* ist hier mit Bindestrich geschrieben. Einerseits erleichtert diese Schreibweise die Lesbarkeit, andererseits betont sie Präfix-Alternativen wie zum Beispiel Logik-, Regel- oder Muster-Orientierung.

[2] Ein Paradigma ist ein von der wissenschaftlichen Fachwelt als Grundlage der weiteren Arbeiten anerkanntes Erklärungsmodell, eine forschungsleitende Theorie. Es entsteht, weil es bei der Lösung von als dringlich erkannten Problemen erfolgreicher ist (zu sein scheint) als andere, bisher „geltende“ Ansätze (Kritik am klassischen Paradigma der Softwareentwicklung →[Bonin88]).

wirtschaftlichen Aufwand zu meistern. Objekt-Orientierung verspricht deshalb (→[Kim/Lochovsky89]),

- einerseits komplexere Systeme erfolgreich konstruieren und betreiben zu können und
- andererseits die Erstellungs- und Wartungskosten von heutigen Systemen zu senken.

Bewirken soll diesen Fortschritt primär eine wesentliche Steigerung der Durchschaubarkeit der Modelle in den Phasen: Anforderungsanalyse, Systemdesign, Programmierung und Wartung. Die Grundidee ist:

Ein „Objekt“ der realen (oder erdachten) Welt bleibt stets erhalten. Es ist über die verschiedenen Abstraktionsebenen leicht verfolgbar. Das gewachsene Verständnis über die Objekte der realen Welt verursacht eine größere Modelltransparenz.

Modell

Was ist ein Objekt im Rahmen der Erarbeitung eines objekt-orientierten Modells? Der Begriff „Modell“ ist mit divergierenden Inhalten belegt. In der mathematischen Logik ist „Modell“ das Besondere im Verhältnis zu einem Allgemeinen: Das Modell eines Axiomensystems ist eine konkrete Inkarnation (≈ Interpretation) dieses Systems. In der Informatik wird der Begriff in der Regel im umgekehrten Sinne verwendet: Das Modell ist das Allgemeine gegenüber einem Besonderen.

Wie können die Objekte in einem konkreten Anwendungsfall erkannt, benannt und beschrieben werden? Das objekt-orientierte Paradigma gibt darauf keine einheitliche, unstrittige Antwort. Es kennt verschiedene bis hin zu konkurrierenden Ausprägungen (zum Beispiel Klassen versus Prototypen). Die jeweiligen Ansätze spiegeln sich in charakteristischen Programmiersprachen wider. Zum Beispiel steht Smalltalk [Goldberg/Robson83, Goldberg83] für die Ausrichtung auf den Nachrichtenaustauch[3] oder CLOS [Bobrow/Moon88] für die inkrementelle Konstruktion von Operationen mittels generischer Funktionen (*Polymorphismus*).

Zur Modellbeschreibung nach einem beliebigen Paradigma dienen in der Praxis Bilder aus beschrifteten Kästchen, Kreisen und Pfeilen (≡ gerichteten Kanten eines Graphen). Die gemalten Kästchen (≡ Boxen) sind Schritt für Schritt zu präzisieren. Boxen sind durch neue Boxen zu verfeinern, das heißt es entstehen weitere Kästchen kombiniert mit Kreisen und Pfeilen. Überspitzt formuliert: Die Lehre von der Boxen-Zeichnerei ist wesentlicher Bestandteil der Modellierung.

Boxologie

Die Informatik kennt schon lange bewährte *Boxologien*[4] im Zusammenhang mit anderen Modellierungstechniken. Die aus dem griechischen stammende Nachsilbe ... *logie* bedeutet *Lehre, Kunde, Wissenschaft* wie sie zum

[3] engl. message passing

[4] Angemerkt sei, daß diese Boxologie nichts mit den Boxern im Sinne von Faustkämpfern oder im Sinne des chinesischen Geheimbundes um 1900 zu tun hat. Auch die Box als eine einfache Kamera oder als Pferdeunterstand sind hier keine hilfreiche Assoziation.

Beispiel beim Wort Ethnologie (≡ Völkerkunde) zum Ausdruck kommt. Boxologie ist einerseits spassig-provokativ gemeint. Andererseits soll damit die Wichtigkeit aussagekräftiger Zeichnungen hervorgehoben werden.

Im imperativen Paradigma umfaßt die Boxologie zum Beispiel Ablaufpläne oder Struktogramme. Geht es um nicht-prozedurale Sachverhalte (zum Beipiel Kausalitäten oder Prädikatenlogik), dann bietet die allgemeine Netztheorie vielfältige Darstellungsmöglichkeiten. So sind zum Beispiel mit (Petri-)Netzen nebenläufige Vorgänge gut visualisierbar.

Sollen die Vorteile einer objekt-orientierten Modellierung in der Praxis zum Tragen kommen, dann gilt es, auch ihre spezifische Boxologie im Hinblick auf Vor- und Nachteile zu analysieren, das heißt die Fragen zu klären: Welche Erkenntnis vermitteln ihre vielfältigen Diagramme, beispielsweise Diagramme vom Typ:

- Anwendungsfalldiagramm,
- Klassendiagramm,
- Verhaltensdiagramm und
- Implementationsdiagramm.

Die Vorteile der objekt-orientierten Modellierung sind nicht in allen Anwendungsfeldern gleich gut umsetzbar. Soll Objekt-Orientierung nicht nur eine Marketing-Worthülse sein, sondern der erfolgreiche Modellierungsansatz, dann bedarf es zunächst einer Konzentration auf dafür besonders prädestinierte Automationsaufgaben. Wir nennen daher Kriterien, um das Erkennen solcher bevorzugten OO-Aufgaben zu unterstützen (→Abschnitt 2.4 auf Seite 25). Als Kriterien dienen einerseits die prognostizierte Menge der Objekte und andererseits ihre Komplexität und Struktur.

2.2 Wurzeln der Objekt-Orientierung

Java, ein konsequent objekt-orientierter Ansatz, schöpft viele Ideen aus den drei klassischen Konzepten:

1. Polymorphismus (generische Funktion)
2. Daten-gesteuerte Programmierung (explizites „dispatching“)
3. Muster-gesteuerter Prozeduraufruf[5]

Stets geht es dabei um die Technik einer separat, inkrementell definierten Implementation und automatischen Wahl von Code.

[5] engl. pattern directed procedure invocation

2.2.1 Wurzel: Polymorphismus

Ein polymorpher Operator ist ein Operator, dessen Verhalten oder Implementation abhängt von der Beschreibung seiner Argumente.[6] Java unterstützt separate Definitionen von polymorphen Operationen (Methoden), wobei die einzelnen Operationen denselben Namen haben, aber Argumente mit verschiedenen Typen abarbeiten. Die Operationen selbst werden als unabhängig voneinander behandelt. Zum Beispiel kann man eine Methode `groesse()` konstruieren, die definiert ist für ein Argument vom Typ `String` (Zeichenkette) und eine weitere ebenfalls mit dem Namen `groesse()` für ein Argument vom Benutzer-definierten Typ `Jacke`. Das Java-System wählt dann die passende Implementation basierend auf den deklarierten oder abgeleiteten Typen der Argumente im aufrufenden Code.
[Hinweis: Die folgende Java-Notation mag für Java-Neulinge noch unverständlich sein. Sie kann von ihnen sorglos überlesen werden. Die „Java-Berührten“ verstehen mit ihr die polymorphen Operationen jedoch leichter.]

```
...
public int groesse(String s) {...}
...
public int groesse(Jacke j)  {...}
...
foo.groesse(myString);
...
foo.groesse(myJacke);
...
```

Zeitpunkt

Entscheidend für objekt-orientierte Ansätze ist die Frage, zu welchem *Zeitpunkt* die Auswertung der Typen der Argumente vollzogen wird. Prinzipiell kann es zur Zeit der Compilierung (*statisches Konzept*) oder zur Laufzeit (*dynamisches Konzept*) erfolgen. Java unterstützt den Compilierungszeit--Polymorphismus. Smalltalk ist beispielsweise ein Vertreter des Laufzeit--Polymorphismus.

2.2.2 Wurzel: Daten-gesteuerte Programmierung

Daten-gesteuerte Programmierung ist keine leere Worthülse oder kein Pleonasmus[7], weil offensichtlich Programme von Daten gesteuert werden. Zumindest aus der Sicht der Mikroprogramme eines Rechners sind ja alle Programme selbst Daten. Hier geht es um die Verknüpfung von passiven

[6] Schon die alte Programmiersprache FORTRAN kennt polymorphe arithmetische Operatoren. In FORTRAN hängt der Typ des Arguments vom ersten Buchstaben des Namens der Variablen ab, wobei die Buchstaben $I, \ldots, N$ auf Fixpunktzahlen, während die anderen auf Gleitkommazahlen verweisen. Zur Compilierungszeit wird die passende Operation anhand der Typen der Argumente ausgewählt. Während FORTRAN dem Programmierer nicht ermöglicht, solche polymorphen Operationen selbst zu definieren, ist dies in modernen objekt-orientierten Sprachen, wie zum Beispiel in Java oder `C++` (→[Stroustrup86, Stroustrup89]) möglich.

[7] Als Pleonasmus wird eine überflüssige Häufung sinngleicher oder sinnähnlicher Ausdrücke bezeichnet.

Systemkomponenten (≡ anwendungsspezifischen Daten) mit aktiven Systemkomponenten (≡ anwendungsspezifischen Operationen).

Insbesondere bei Anwendungen im Bereich der Symbolverarbeitung haben sich im Umfeld der Programmierung mit LISP (LISt Processing) verschiedene Ausprägungen der sogenannten *daten-gesteuerten Programmierung*[8] entwickelt (→[Abelson85]). Der Programmierer definiert selbst eine *dispatch*-Funktion[9], die den Zugang zu den datentypabhängigen Operationen regelt. Die Idee besteht darin, für jeden anwendungsspezifischen Datentyp ein selbstdefiniertes Symbol zu vergeben. Jedem dieser Datentyp-Symbole ist die jeweilige Operation als ein Attribut zugeordnet.[10] Quasi übernehmen damit die „Datenobjekte“ selbst die Programmablaufsteuerung. Erst zum Zeitpunkt der Evaluation („Auswertung“) eines Objektes wird die zugeordnete Operation ermittelt (→[Bonin91b]).

2.2.3 Wurzel: Muster-gesteuerter Prozeduraufruf

Ziel: Paßt!

Ist die Auswahl der Operation abhängig von mehreren Daten im Sinne eines strukturierten Datentyps, dann liegt es nahe, das Auffinden der Prozedur als einen Vergleich zwischen einem Muster und einem Prüfling mit dem *Ziel: Paßt!* zu konzipieren. Ein *allgemeingültiger Interpreter*, der dieses Passen feststellt, wird dann isoliert, das heißt aus dem individuellen Programmteil herausgezogen.

Da eine Datenbeschreibung (zum Beispiel eine aktuelle Menge von Argumenten) prinzipiell mehrere Muster und/oder diese auf mehr als einem Wege entsprechen könnte, ist eine Abarbeitungsfolge vorzugeben. Die *Kontrollstruktur* der Abarbeitung ist gestaltbar im Sinne eines Beweises, das heißt sie geht aus von einer Zielthese und weist deren Zutreffen nach, oder im Sinne einer Zielsuche, das heißt sie verfolgt einen Weg zum zunächst unbekannten Ziel. Anhand der Kontrollstruktur kann die Verknüpfung von Daten und Prozedur mittels der Technik des Mustervergleichs[11] als Vorwärtsverkettung[12] und/oder als Rückwärtsverkettung[13] erfolgen.

Aus der Tradition des muster-gesteuerten Prozeduraufrufs sind Schritt für Schritt leistungsfähige regel- und logik-orientierte Sprachen mit Rückwärtsverkettung[14], wie zum Beispiel PROLOG[15], entstanden.

2.3 Ausrichtung objekt-orientierter Sprachen

passiv

Bei einem groben, holzschnittartigen Bild von objekt-orientierten Sprachen ist die Zusammenfassung der passiven Komponente (≡ Daten) mit der aktiven Komponente (≡ zugeordnete Operationen) zum Objekt verknüpft mit

[8] engl. „data-directed programming“ oder auch „data-driven programming“
[9] engl. dispatch ≡ Abfertigung
[10] LISP ermöglicht diese Attributzuordnung über die Eigenschaftsliste, die LISP für jedes Symbol automatisch führt.
[11] engl. „pattern matching“
[12] engl. „forward chaining“
[13] engl. „backward chaining“
[14] engl. backward chaining rule languages
[15] *PROgramming in LOGic* →[Belli88, Clocksin/Mellish87]

Schwerpunkte der Objekt-Orientierung			
		I *Klassen-/ (Daten)Typ-Betonung*	II *Prototyp-/ Einzelobjekt-Betonung*
A	*Betonung der Operations-konstruktion*	CLOS (Java) (C++)	(Daten-gesteuerte Programmierung) PROLOG
B	*Betonung des Nachrichten-austausches*	Smalltalk (Java) (C++)	SELF (Actor-Sprachen)

Legende:
Actor-Sprachen →[Lieberman81]
$C++$ →[Stroustrup86, Stroustrup89]
CLOS (*C*ommon *L*isp *O*bject *S*ystem) →[Bobrow/Moon88]
Daten-gesteuerte Programmierung →[Abelson85]
Java →[Arnold/Gosling96]
SELF →[Ungar/Smith91]
Smalltalk →[Goldberg/Robson83, Goldberg83]
PROLOG (*PRO*gramming in *LO*gic) →[Belli88, Clocksin/Mellish87]

Tabelle 2.1: Ausrichtung von objekt-orientierten Ansätzen

aktiv

dem Konzept des Nachrichtenaustausches. Dabei enthält die Nachricht, die an ein Objekt, den Empfänger, gesendet wird, den Namen der Operation und die Argumente. Entsprechend der *daten-gesteuerten Programmierung* dient der Empfänger und der Name der Operation zur Auswahl der auszuführenden Operation, die üblicherweise Methode genannt wird. So gesehen ist die Objekt-Orientierung, wie folgt, definierbar:

Objekt-Orientierung	≡	Datentypen + Nachrichtenaustausch

Das Bild von Objekten, die miteinander Nachrichten austauschen, verdeutlicht nur einen Schwerpunkt der Objekt-Orientierung. Verfolgt man die Polymorphismuswurzel, dann rückt nicht der Transfer von Nachrichten, sondern die Konstruktion von generischen Funktionen in den Fokus. Kurz: Die Konstruktion der Operation wird betont.

Es scheint trivial zu sein, festzustellen, daß bei objekt-orientierten Sprachen neue Objekte aus existierenden Objekten konstruiert werden. Die Konzepte der „Ableitung“ der neuen, also der anwendungsspezifischen Objekte unterscheiden sich jedoch. Gemeinsame Eigenschaften einzelner Objekte können zu einer Klasse oder zu einem charakteristischen Einzelobjekt zusammengefasst werden. Oder umgekehrt: Ein konkretes Objekt kann sich aus *Klasseneigenschaften* ergeben oder es kann bezugnehmen auf einen *Prototyp* (ein anderes Einzelobjekt).

Klasse

Das Klassenkonzept unterstellt von Anfang an eine größere Zahl von Objekten, die gleiche Eigenschaften (Struktur und Verhalten) aufweisen. Als Beispiel nehmen wir mehrere Ausgabekonten an, wobei jedes einzelne Ausgabekonto einen SOLL-Wert aufweist. Die Klasse `Ausgabekonten` beschreibt das für alle Ausgabekonten gleiche Merkmal SOLL-Wert. Wird ein einzelnes Ausgabekonto erzeugt, so „erbt“ es aus der Klasse `Ausgabekonten` die Definition SOLL-Wert. Da die Klassendefinition selbst wiederum auf abstraktere Klassen zurückgreift, zum Beispiel auf eine Klasse `Haushaltskonten`, umfaßt das Klassenkonzept mehrere Abstraktionsebenen. Das Objekt entsteht aus einer Hierarchie von Beschreibungen auf verschiedenen Abstraktionsebenen.

Prototyp

Beim Konzept mit Prototypen unterstellt man ein charakteristisches konkretes Objekt mit möglichst umfassend vordefinierten Eigenschaften. Beim Definieren der einzelnen Konten nimmt man Bezug auf diese Beschreibung des charakteristischen Einzelobjekts. Ein übliches Ausgabekonto ist zunächst detailliert zu beschreiben. In unserem Kontenbeispiel wäre zunächst ein Ausgabekonto, vielleicht ein Konto 1203/52121, zu beschreiben und zwar einschließlich seiner Eigenschaft SOLL-Wert. Die einzelnen Ausgabekonten enthalten den Bezug auf und gegebenenfalls die Abweichung von diesem Prototypen 1203/52121. Dieses Prototypmodell unterstützt vergleichsweise weniger das Herausarbeiten von mehreren Abstraktionsebenen. Einerseits entfällt die Suche nach Eigenschaften und Namen der höheren Abstraktion für unser Kontensystem. Andererseits ist die Wiederverwendbarkeit geringer.

Die Tabelle 2.1 auf Seite 23 zeigt die vier skizzierten Schwerpunkte als die zwei Dimensionen: Klassen / Prototyp und Operation / Nachrichtenversand (ähnlich [Gabriel91]). Eine schlagwortartige Übersicht (→Tabelle 2.2 auf Seite 26) vergleicht Java mit , Smalltalk und CLOS. Ausgewählt wurden diese OO-Sprachen aufgrund ihrer unterschiedlichen OO-Konzepte.

Die Option eines Benutzer-definierten Polymorphismus zur Compilierungszeit ist in Zukunft sicherlich auch in weit verbreitete Programmiersprachen wie COBOL, BASIC, Pascal oder FORTRAN einbaubar. Damit können jedoch solche ursprünglich imperativ-geprägten Sprachen mit dieser Form aufgepfropfter Objekt-Orientierung nicht die volle Flexibilität ausschöpfen, wie sie zum Beispiel von CLOS geboten wird mit *Multiargument Polymorphismus zur Laufzeit*.

CLOS

Objekt-orientierte Modelle sind geprägt durch eine (Un)Menge von Definitionen, die Abstraktionen (≡ Klassen) von „realen“ Einheiten (≡ Objekten) beschreiben. Da Objektklassen von anderen Klassen Eigenschaften (interne Variablen, Methoden) *erben* können, lassen sich aus allgemeinen Objekten spezielle anwendungsspezifische konstruieren. Vererbungsgraphen, gezeichnet mit beschrifteten Kästchen, Kreisen und Pfeilen, bilden dabei das Modell ab (→Abbildung 3.11 auf Seite 45). Die Erkenntnisgewinnung und -vermittlung geschieht primär über Bilder, die aus vielen verknüpften „Boxen“ bestehen.

2.4 Prädestinierte Java-Anwendungsfelder

Entscheidungsträger in der Praxis fordern Handlungsempfehlungen. Welche Anwendungsfelder sind für die Objekt-Orientierung in der Art und Weise von Java heute besonders erfolgversprechend? Offensichtlich ist eine Antwort darauf risikoreich. Die Einflußfaktoren sind vielfältig, und die Prioritäten bei den Zielen divergieren.

Wir begrenzen unsere Antwort auf den Fall, daß die Objekt-Orientierung den gesamten Softwarelebenszyklus umfaßt. Es soll kein Paradigmenwechsel zwischen Analyse-, Entwurfs- und Implementationsphase geben. Eine vielleicht erfolgversprechende Kombination zwischen objekt-orientiertem Entwurf und imperativer Programmierung bleibt hier unberücksichtigt.

Effizienz

Da auch der Programmcode objekt-orientiert ist, ist die erreichbare (Laufzeit-)Effizienz ebenfalls ein Entscheidungskriterium. Müssen wir mangels heute schon bewährter objekt-orientierter Datenbank-Managementsysteme erst die Objekte aus den Daten bei jedem Datenbankzugriff erzeugen und beim Zurückschreiben wieder in Tabellen (Daten) konvertieren, dann entstehen zumindest bei komplex stukturierten Objekten Laufzeitprobleme. Die Objektmenge und die Komplexität der Objektstruktur sind relevante Kriterien, um sich für die Objekt-Orientierung entscheiden zu können (Tabelle 2.3 auf Seite 27).

2.4.1 Kriterium: Menge

Die Laufzeit-Effizienz betrifft zwei Aspekte:

1. den Nachrichtenaustausch zwischen den Objekten und

Terminologie (Leistungen) bei der Objekt-Orientierung				
	Aspekte[1]	I Java	II **Smalltalk**	III **CLOS**
1	Abstraktion	class		
2	Objekt	member of class	instance	
3	Ver-erbung	superclass		direct superclasses
		subclass		
4	Objekt-struktur	member data elements	instance variables	slots
5	Struktur-beschrei-bung (*Slot*)	name		
		type initial value forms		initial value forms, accessor functions, initi-alizer keywords
6	Opera-tionsaufruf	calling a method	sending a message	calling a generic function
7	Opera-tionsimple-mentation	method		
8	Verknüpfung Klasse mit Operation	defined in class scope	through class browser	specializers in parameter list
9	Multiargument „Dispatch“	chained dispatch		multi-methods
10	Referenz zum Objekt	this	self	name in parameter list
11	Auruf der verdeckten Operation	call method with qualified name	message to super	call-next-method
12	Direktzu-griff auf interen Objektzustand	by name within class scope	by name within method scope	slot-value
13	Allgemeiner Zugriff auf internen Objektzustand	user methods		
		public declaration		accessor functions
14	Operations-kombination	Nein		standard method combination

Legende:

[1] ähnlich [Kczales91] S. 253.

Tabelle 2.2: Java im Vergleich mit Smalltalk und CLOS

Kriterien für die Wahl besonders geeigneter OO-Anwendungsfelder				
Objekt-**Struktur**			Objekt-**Menge** (Datenvolumen)	
		gleich-artige Fälle	I *relativ beschränkt* (ladbar in Arbeits-speicher)	II *sehr groß* (DBMS erforder-lich)
einfach	A_1	wenige	[0] OO → Effi-zienz-ver-lust	[-] Objekt ⇔ Datensatz Konver-tierung
	A_2	viele		
komplex	B_1	wenige	[+] OO plus XPS	[–] ohne OO-DBMS nicht mach-bar
	B_2	viele	[++] **OO**	

Legende:

Bewertungsskala:

[–]	ungeeignet
[-]	kaum geeignet
[0]	machbar
[+]	geeignet
[++]	sehr gut geeignet

DBMS	≡	Datenbank-Managementsystem
OO	≡	Objekt-Orientierung
XPS	≡	Expertensystem-Techniken (z.B. Regel-Orientierung)

Tabelle 2.3: Besonders geeignete Anwendungsfelder für die Objekt-Orientierung

2. das Propagieren von Modifikationen der Objekt-Eigenschaften, des Vererbungsgraphen und gegebenenfalls der Vererbungsstrategie.

Beide Aspekte sind nur bei Objektmengen einigermaßen effizient beherrschbar, bei denen alle betroffenen Objekte sich im Arbeitsspeicher des Rechners befinden. Ist die Menge der Objekte jedoch so groß, daß Objekte auf externen Speichern zu führen sind, dann ist die *Konvertierungszeit* von Objekten in Daten(relation)en und umgekehrt ein erheblicher Zeitfaktor. Erfreulicherweise kann bei manchen Anwendungen diese Konvertierung im Zusammenhang mit der Start- und Beendigungsprozedur erfolgen. Solche terminierbaren Rüstzeiten können häufig akzeptiert werden.

Rüstzeit

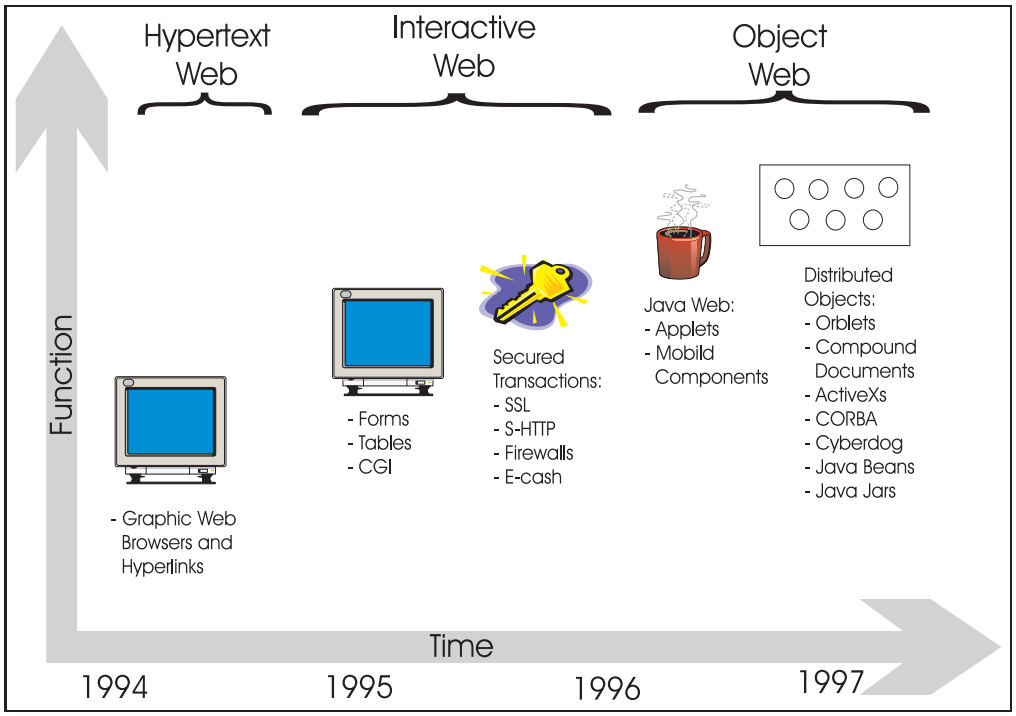
Bei wirklich großen Objektmengen ist auch das Propagieren der Modifikationen auf alle betroffenen Objekte zeitkonsumptiv, insbesondere wenn der Zugriff auf Objekte erst die Konvertierung von Daten in Objekte umfaßt. Daher ist zum Beispiel ein objekt-orientiertes Einwohnerwesen für eine Großstadt nur bedingt zu empfehlen, solange der Nachrichtenaustausch und das Propagieren von Modifikationen mit Konvertierungsvorgängen belastet sind. In diesem Kontext reicht es nicht aus, wenn wir die Laufzeit-Effizienz durch eine Typprüfung zur Compilierungszeit steigern und zum Beispiel auf einen Laufzeit-Polymorphismus (→Abschnitt 2.2.1 auf Seite 21) ganz verzichten.

2.4.2 Kriterium: Struktur

Nicht nur der Konvertierungsaufwand ist offensichtlich abhängig von der Komplexität der Objektstruktur, sondern auch die Konstruktion und Abarbeitung eines Vererbungsgraphens.

Stück-Kosten

Ist ein umfangreicher, tiefgeschachtelter Graph mit relativ wenigen „Blättern“ (Instanzen) pro Knoten zu erwarten, dann ist der Konstruktions- und Abarbeitungsaufwand pro Instanz hoch (Stückkosten!). Alternativen mit geringerem Abbildungsaufwand kommen in Betracht. Neben dem Prototyp-Ansatz ist der Übergang auf die Strukturierung im Sinne der Expertensystem-Technik (XPS) eine zweckmäßige Alternative. Die Modularisierung erfolgt primär in handhabbaren „Wissensbrocken“, die ein allgemeingültiger „Interpreter“ auswertet. Verkürzt formuliert: In Konkurrenz zur Welt aus kommunizierenden Objekten treten XPS mit Regel-Orientierung.



Bildidee: [Orfali/Harkey97]

Abbildung 2.1: Die Entwicklung der WWW-Technologie

Kapitel 3

Modellieren mit UML

Die Qualität des Java-Programms hängt ab von der Qualität der Modelle, die die Objekte der Anwendungswelt sowie die (Benutzer-)Anforderungen abbilden. Erst die „richtigen“ Objekte mit den „richtigen“ Beziehungen führen zum gelungenen Java-Programm. Kurz: Fehler im Modell gleich Fehler im Programm! Für den Transfer der „richtigen“ Objekte und Beziehungen in den Java-Quellcode werden die erforderlichen Modelle in *Unified Modeling Language* (UML) notiert.

UML ist eine Sprache zum Spezifizieren, Visualisieren, Konstruieren und Dokumentieren von „Artifakten eines Softwaresystems“. Mit UML können Geschäftsvorgänge gut modelliert werden. Zusätzlich stellt UML eine Sammlung von besten Ingenieurpraktiken dar, die sich erfolgreich beim Modellieren großer, komplexer Systeme bewährt haben.

Trainingsplan

Das Kapitel „Modellieren mit UML“ erläutert:

- die verschiedenen Arten von UML Diagrammen,
  ↪ Seite 32 ...
- die verschiedenen Typen von Klassen mit ihren Varaibalen und Methoden,
  ↪ Seite 33 ...
- die Verbindung zwischen Klassen in Form der Assoziation,
  ↪ Seite 37 ...
- die Beziehungen zwischen dem Ganzen und seinen (Einzel-)Teilen,
  ↪ Seite 41 ...
- die Vererbung von Eigenschaften (Variablen und Methoden) und
  ↪ Seite 43 ...
- gibt Empfehlungen für die Namensvergabe für UML-Elemente.
  ↪ Seite 46 ...

3.1 UML-Boxologie

Für die objekt-orientierte Modellierung, also für Analyse und Design, wurden zu Beginn der 90-Jahre eine Menge von Methoden mit ihren speziellen Notationen (Symbolen) bekannt. Exemplarisch sind beispielsweise zu erwähnen:

- G. Booch;
  Objekt-oriented Analysis and Design with Applications [Booch94]
- D. W. Embley u. a.;
  Object-Oriented Systems Analysis – A Model-Driven Approach [Embley92]
- I. Jacobsen u. a.;
  Object-oriented Software Engineering, A Use Case Driver Approach [Jacobsen92]
- W. Kim u. a.;
  Object-Oriented Concepts, Databases, and Applications [Kim/Lochovsky89]
- J. Rumbaugh u. a.;
  Object-oriented Modelling and Design [Rumbaugh91]

Rational

Die *Unified Modeling Language* (UML) wurde von der *Rational Software Corporation* (→[Rational97]) aus solchen Methoden und Notationen entwickelt. UML ist eine Sprache zum

1. Spezifizieren,
2. Visualisieren,
3. Konstruieren und
4. Dokumentieren

von Artifakten eines Softwaresystems und auch für die Modellierung von Geschäftsvorgängen (*business modeling*) und anderen Nicht-Software-Systemen. Darüberhinaus stellt UML eine Sammlung von besten Ingenieurpraktiken dar, die sich erfolgreich beim Modellieren großer, komplexer Systeme bewährt haben. UML umfaßt folgende graphische Diagramme (*Originalnamen*):

1. Anwendungsfalldiagramm (*use case diagram*)
2. Klassendiagramm (*class diagram*)
3. Verhaltensdiagramme (*behavoir diagrams*):
   (a) Zustandsdiagramm (*statechart diagram*)

   (b) Aktivitätsdiagramm (*activity diagram*)

   (c) Interaktionsdiagramme (*interaction diagrams*):

      i. Sequenzdiagramm (*sequence diagram*)

      ii. Kollaborationsdiagramm (*collaboration diagram*)

4. Implementierungsdiagramme (*implementation diagrams*):

   (a) Komponentendiagramm (*component diagram*)

   (b) Einsatzdiagramm (Knoten[1]) (*deployment diagram*)

Diese Diagramme ermöglichen verschiedene Sichten auf das objekt-orientierte System und zwar besonders aus den Perspektiven der Analyse und des Designs. Aufgrund der konsequenten Objekt-Orientierung unterstützt UML beispielsweise keine Datenflußdiagramme, weil nicht Daten und deren Fluß durch das Programm sondern Objekte und deren Kommunikation zur objekt-orientierte Denkwelt gehören.

3.2 Basiselement: Klasse

3.2.1 Beschreibung: Klasse

Eine Klasse definiert die Eigenschaften ihrer Objekte mit Hilfe von Variablen (Attributen) und Methoden (Operationen). Darüberhinaus kann diese Definition auch Zusicherungen, Merkmale und Stereotypen umfassen. Tabelle 3.1 auf Seite 34 zeigt die UML-Notation einer Klasse.

[Hinweis: Ein verwandter Begriffe für die Klasse ist der Begriff (Daten-)-Typ. In Java ist ein einfacher Datentypen wie zum Beispiel `float` von einem „zusammengesetzten“ Datentype (*Reference Type*) wie zum Beispiel `FahrzeugProg` (→Abschnitt 5.1.3 auf Seite 64) zu unterscheiden.]

Beschreibung: Abstrakte Klasse

Eine abstrakte Klasse ist eine Klasse, die die Basis für weitere Unterklassen bildet. Eine abstrakte Klasse hat keine Mitglieder (*member*), also keine Instanzen (Objektexemplare). Sie wird als eine (normale) Klasse mit dem Merkmal {`abstract`} notiert[2].

Beschreibung: Metaklasse

Eine Metaklasse dient zum Erzeugen von Klassen. Sie wird wie eine (normale) Klasse notiert und erhält den Stereotyp ≪metaclass≫[3].

[1] Ein Knoten ist ein Computer, also ein zur Laufzeit vorhandenes Gerät (Objekt), daß über Speicher und Prozessorleistung verfügt.

[2] Alternativ zu dieser Kennzeichnung kann der Klassenname auch *kursiv* dargestellt werden.

[3] Näheres zur Programmierung mit Metaobjekten →[Kczales91].

Klassensymbol:

Klassensymbol mit Variablen und Methoden:

≪stereotyp≫ **paket**::**Klasse** {merkmal}
variable : Typ=initialwert {merkmal} {zusicherung}
methode(param : Typ=initialwert) : Rückgabetyp {merkmal} {zusicherung}

Beispiel: `Fahrzeug` (→Abschnitt 5.1.3 auf Seite 64)

≪fachklasse≫ **de.FHNON.Fahrzeug**::**Fahrzeug** {public, Version=1.0}
geschwindigkeit : double=0 {geschwindigkeit ≥ 0} fahrtrichtung : long=0 {fahrtrichtung ≥ 0} eigentuemer : String=" " anzahl : long=0 {anzahl ≥ 0}
≪getMethode≫ getAnzahl() : long getGeschwindigkeit() : double getFahrtrichtung() : long getEigentuemer() : String ≪konstuktor≫ Fahrzeug() Fahrzeug(eigentuemer: String) Fahrzeug(geschwindigkeit: double, fahrtrichtung : long, eigentuemer: String) ≪bewegung≫ wohin(g : double, f : long) : Fahrt wohin(fahrzeugFahrt : Fahrt) beschleunigt(g : double) : double

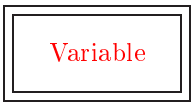
Tabelle 3.1: UML-Basiselement: Klasse

UML-Notation	Erläuterung
+publicVariable	allgemein zugreifbare Variable
⇔privateVariable	private, nicht allgemein zugreifbare Variable
#protectedVariable	geschützte, bedingt zugreifbare Variable
/abgeleitetesVariable	hat einen Wert aus anderen Variablen
klassenVariable	Variable einer Klasse (`static`)

Hinweis: Sichtbarkeit (Zugriffsrechte) im Java-Kontext →Tabelle 5.6 auf Seite 86.

Tabelle 3.2: Kennzeichnung einer Variablen in UML

Beschreibung: Variable (Attribut)



Ein Variable benennt einen Speicherplatz in einer Instanz (≡ Instanzvariable) oder in der Klasse selbst (≡ Klassenvariable).

```
variable : Typ=initialwert {merkmal} {zusicherung}
```

[Hinweis: Verwandte Begriffe für die Variable sind die Begriffe Attribut, *Member*, *Slot* und Datenelement.]

Mit Hilfe eines der Sonderzeichen „+“, „⇔“ und „#“, das dem Namen der Variablen vorangestellt wird, kann eine Variable in Bezug auf ihre Sichtbarkeit besonders gekennzeichnet werden. Ein vorangestellter Schrägstrich (*slash*) gibt an, daß der Wert der Variable von anderen Variablen abgeleitet ist. Handelt es sich um eine Klassenvariable, dann wird der Name unterstrichen. Tabelle 3.2 auf Seite 35 zeigt diese Möglichkeiten einer Kennzeichnung.

Beschreibung: Methode

Methoden sind der „aktiv(ierbar)e“ Teil (Algorithmus) der Klasse. Eine Methode wird durch eine Nachricht an eine Instanz aktiviert. Eine Methode kann sich auch auf die Klasse selbst beziehen (≡ Klassenmethode (`static`)). Dann wird sie durch eine Nachricht an die Klasse aktiviert.

```
methode(parameter : ParameterTyp=standardWert) :
                    RückgabeTyp {merkmal} {zusicherung}
```

Beispiel:

```
setPosition(x : int=10, y : int=300) :
  boolean {abstract} {(x ≥ 10) ∧ (y ≥ 300)}
```

[Hinweis: Verwandte Begriffe für Methode sind die Begriffe Funktion, Prozedur und Operation.]

Mit Hilfe eines der Sonderzeichen „+“, „⇔“ und „#“, das dem Namen der Methode vorangestellt wird, kann eine Methode in Bezug auf ihre Sichtbarkeit besonders gekennzeichnet werden. Handelt es sich um eine Klassen-

UML-Notation	Erläuterung
+publicMethode()	allgemein zugreifbare Methode
⇔privateMethode()	private, nicht allgemein zugreifbare Methode
#protectedMethode()	geschützte, bedingt zugreifbare Methode
klassenMethode()	Methode einer Klasse (`static`)

Hinweis:
Sichtbarkeit (Zugriffsrechte) im Java-Kontext →Tabelle 5.6 auf Seite 86.

Tabelle 3.3: Kennzeichnung einer Methode in UML

methode, dann wird der Name unterstrichen. Tabelle 3.3 auf Seite 36 zeigt diese Möglichkeiten einer Kennzeichnung.

Merkmal

Beschreibung: Merkmal und Zusicherung

Ein Merkmal ist ein Schlüsselwort aus einer in der Regel vorgegebenen Menge, das eine charakteristische Eigenschaft benennt. Ein Merkmal steuert häufig die Quellcodegenerierung.
Beispiele: {`abstract`}, {`readOnly`} oder {`old`}

Eine Zusicherung definiert eine Integritätsregel (Bedingung). Häufig beschreibt sie die zulässige Wertmenge, eine Vor- oder Nachbedingung für eine Methode, eine strukturelle Eigenschaft oder eine zeitliche Bedingung.
Beispiel: `Rechnung.kunde = Rechnung.vertrag.kunde`

Die Angaben von {`zusicherung`} und {`merkmal`} überlappen sich. So kann jedes Merkmal auch als eine Zusicherung angegeben werden.
Merkmal als Zusicherung angegeben — Beispiele:
{`abstract=true`}, {`readOnly=true`} oder {`old=true`}

Stereotyp

Beschreibung: Stereotyp

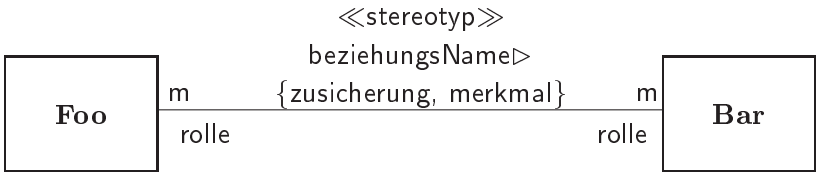
Ein Stereotyp ist eine Möglichkeit zur Kennzeichnung einer Gliederung auf projekt- oder unternehmensweiter Ebene. Ein Stereotyp gibt in der Regel den Verwendungskontext einer Klasse, Schnittstelle, Beziehung oder eines Paketes an.
Beispiele: {`fachklasse`}, {`präsentation`} oder {`vorgang`}

[Hinweis: Verwandte Begriffe für den Stereotyp sind die Begriffe Verwendungskontext und Zusicherung.]

3.2.2 Beschreibung: Paket

Paket

Ein Paket beinhaltet eine Ansammlung von Modellelementen beliebigen Typs. Mit Hilfe von Paketen wird ein (komplexes) Gesamtmodell in überschaubarere Einheiten gegliedert. Jedes Modellelement gehört genau zu einem Paket. Ein Paket wird mit dem Symbol eines Aktenregisters darge-



Legende:

beziehungsName	Name der Assoziation
{zusicherung}	→Abschnitt 3.2.1 auf Seite 36
{merkmal}	→Abschnitt 3.2.1 auf Seite 36
m	Multiplizität (→Tabelle 3.4 auf Seite 39)
rolle	Sichweise durch das gegenüberliegende Objekt
▷	(Lese)-Richtung für die Beziehung; hier: **Foo**beziehungsName**Bar**

Abbildung 3.1: UML-Beziehungselement: Assoziation

stellt. Innerhalb des Symbols steht der Name des Paketes. Werden innerhalb des Symbols Aktenregisters Modellelemente genannt, dann steht der Paketname auf der Aktenregisterlasche.
Beispiel: `de.FHNON.Fahrzeug`

[Hinweis: Verwandte Begriffe für das Paket sind die Begriffe Klassenkategorie, Subsystem und *Package*.]

3.3 Beziehungselement: Assoziation

3.3.1 Beschreibung: Assoziation

Eine Assoziation beschreibt eine Verbindung zwischen Klassen (→Abbildung 3.1 auf Seite 37). Die Beziehung zwischen einer Instanz der einen Klasse mit einer Instanz der „anderen“ Klasse wird Objektverbindung (englisch: *link*) genannt. Links lassen sich daher als Instanzen einer Assoziation auffassen. Häufig ist eine Assoziation eine Beziehung zwischen zwei verschiedenen Klassen (→Abbildung 3.2 auf Seite 38). Jedoch kann eine Assoziation auch von rekursiver Art sein, das heißt beispielsweise als Beziehung zwischen zwei Instanzen derselben Klasse formuliert werden (Beispiel →Abbildung 3.3 auf Seite 39) oder eine Assoziation zwischen mehreren Klassen sein. Spezielle Formen der Assoziation sind die Aggregation (→Abschnitt 3.4.1 auf Seite 41) und die Komposition (→Abschnitt 3.4.2 auf Seite 42).

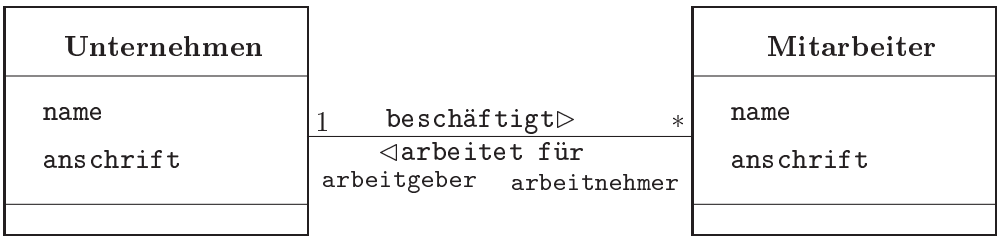
link

[Hinweis:Verwandte Begriffe für die Assoziation sind die Begriffe Relation, Aggregation, Komposition, Link und Objektverbindung.]

3.3.2 Multiplizität

Die Multiplizität m gibt an mit wievielen Instanzen der gegenüberliegende Klasse **Bar** eine Instanz der Klasse **Foo** assoziiert ist[4]. Dabei kann ei-

min:max



Legende:
→Abbildung 3.1 auf Seite 37

Abbildung 3.2: Beispiel einer Assoziation: Ein Unternehmen beschäftigt viele Mitarbeiter

ne Bandbreite durch den Mini- und den Maximumwert angegeben werden (→Tabelle 3.4 auf Seite 39).

Ein Assoziation wird in der Regel so implementiert, daß die beteiligten Klassen zusätzlich entsprechende Referenzvariablen bekommen. Im Beispiel „Ein Unternehmen beschäftigt viele Mitarbeiter“ (→Abbildung 3.2 auf Seite 38) erhält die Klasse **Unternehmen** die Variable `arbeitnehmer` und die Klasse **Mitarbeiter** die Variable `arbeitgeber`. Aufgrund der angebenen Multiplizität $*$ („viele Mitarbeiter“) muß die Variable `arbeitnehmer` vom Typ einer Behälterklasse sein, damit sie viele Objekte aufnehmen kann. Ein *Set* (Menge ohne Duplizität) oder ein *Bag* (Menge mit Duplizität) sind beispielsweise übliche Behälterklassen.

[Hinweis: Moderne Modellierungswerkzeuge verwenden den Rollennamen für die Referenzvariable und generieren entsprechend der Multiplizität die Variable mit ihrem Typ automatisch.]

Eine Assoziation kann selbst Vaiablen haben. Im Beispiel „Ein Unternehmen beschäftigt viele Mitarbeiter“ (→Abbildung 3.2 auf Seite 38) kann dies beispielsweise die Historie der Beschäftigungsverhälnisse für einen Mitarbeiter sein, das heißt die `von`- und `bis`-Daten der Assoziation `beschäftigt`. Eine solche Beziehung bezeichnet man als degenerierte Assoziationsklasse. Das Beiwort „degeneriert“ verdeutlicht, daß die Assoziationsklasse keine Instanzen beschreibt und daher keinen eigenen Namen benötigt. In den späteren Phasen der Modellierung wird eine solche degenerierte Klasse in eine vollständige Assoziationsklasse, die dann einen Namen hat und Instanzen aufweisen kann, umgeformt (→Abbildung 3.4 auf Seite 40).

3.3.3 Referentielle Integrität

Integrität

Soll eine Assoziation eine Bedingung erfüllen, dann ist diese in Form der {zusicherung} neben der Assoziationslinie zu notieren. Eine {zusicherung} kann auch die referenzielle Integrität beschreiben. Hierzu werden beim Löschen beispielsweise angebenen:

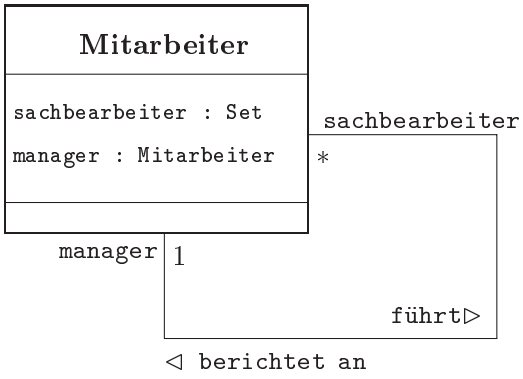
[4] ... beziehungsweise assoziiert sein kann.

Multi-plizität	Erläuterung
	keine Angabe entspricht *
*	null oder größer
0..*	null oder größer
1..*	eins oder größer
1	genau eins
0,1	null oder eins
0..3	null oder eins oder zwei oder drei
7,9	sieben oder neun
0..2,5,7	(zwischen null und zwei) oder fünf oder sieben

Legende:

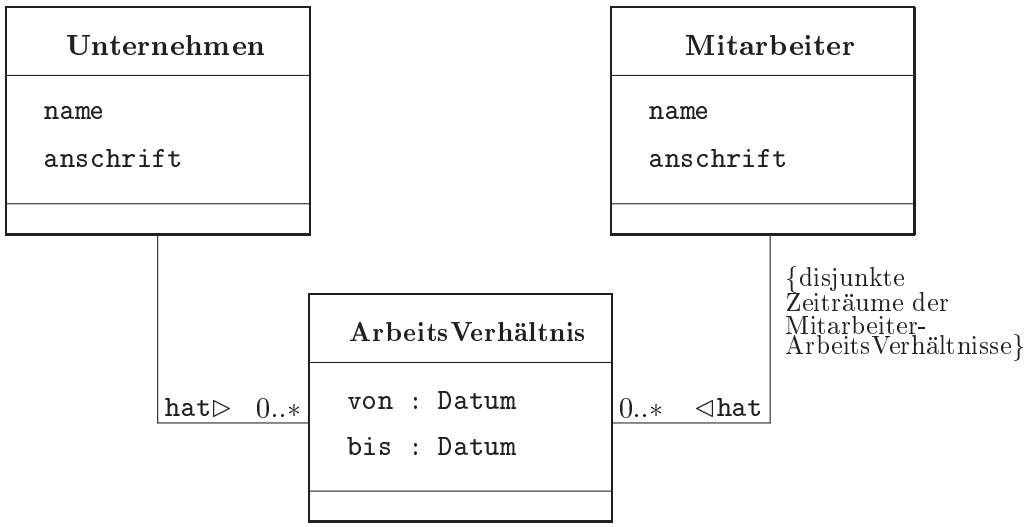
,	Komma ist Trennzeichen für die Aufzählung
*	beliebig viele
0	optional

Tabelle 3.4: Angabe der Multiplizität



Legende:
Assoziation →Abbildung 3.1 auf Seite 37

Abbildung 3.3: Beispiel einer direkten rekursiven Assoziation



Legende:
→Abbildung 3.2 auf Seite 38

Abbildung 3.4: Beispiel einer Assoziationsklasse: `ArbeitsVerhältnis`

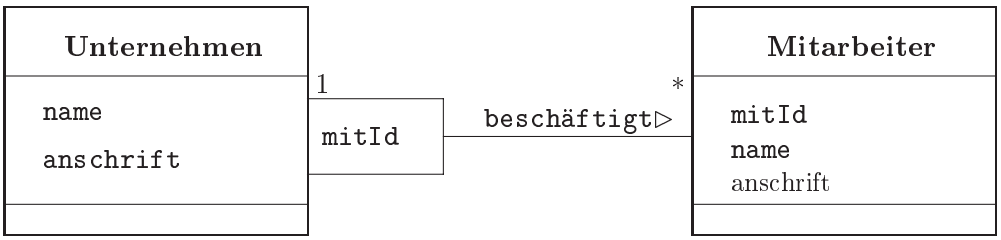
- {`prohibit deletion`}
  Das Löschen eines Objektes ist nur erlaubt, wenn keine Beziehung zu einem anderen Objekt besteht.

- {`delete link`}
  Wenn ein Objekt gelöscht wird, dann wird nur die Beziehung zwischen den Objekten gelöscht.

- {`delete related object`}
  Wenn ein Objekt gelöscht wird, dann wird das assoziierte („gegenüberliegende“) Objekt ebenfalls gelöscht.

3.3.4 Schlüsselangabe bei einer Assoziation

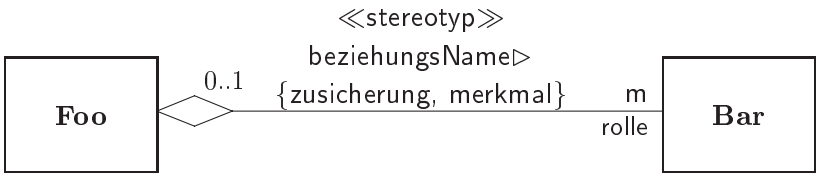
Beziehungen mit der Multiplizität $m = *$ werden in der Regel mittels einer Behälterklasse implementiert (→Abschnitt 3.3.2 auf Seite 37). Dabei kann es sinnvoll sein schon im Klassenmodell mit den Assoziationen den Zugriffsschlüssel darzustellen. Ein solcher Schlüssel, auch als qualifizierendes Attribut der Assoziation bezeichnet, wird als Rechteck an der Seite der Klasse notiert, die über diesen Schlüssel auf das Zielobjekt zugreift. Ist ein solcher Schlüssel genannt, dann ist er Bestandteil der Assoziation. Die Navigation erfolgt dann ausschließlich über diesen Schlüssel. Ein Beispiel zeigt Abbildung 3.5 auf Seite 41.



Legende:
→Abbildung 3.2 auf Seite 38

Unternehmen		Mitarbeiter		
name	anschrift	mitId	name	anschrift
Otto KG	Nürnberg	Boe	Boesewicht	Bonn
Otto KG	Nürnberg	Gu	Gutknecht	Lüneburg
Emma AG	Berlin	Fr	Freund	Lüneburg

Abbildung 3.5: Beispiel einer qualifizierenden Assoziation: Schlüssel=`mitId`



Legende:
→Abbildung 3.1 auf Seite 37

Abbildung 3.6: UML-Beziehungselement: Aggregation

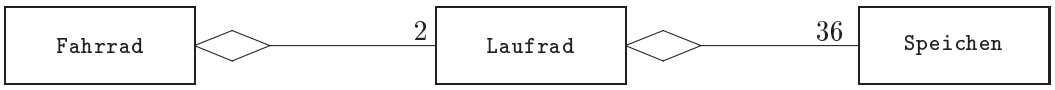
3.4 Beziehungselemente: Ganzes ⇔ Teile

3.4.1 Beschreibung: Aggregation

Eine Aggregation beschreibt eine „Ganzes⇔Teile“-Assoziation. (→Abbildung 3.6 auf Seite 41) Das Ganze nimmt dabei Aufgaben stellvertretend für seine Teile wahr. Im Unterschied zur normalen Assoziation haben die beteiligten Klassen keine gleichberechtigten Beziehungen. Die Aggregationsklasse hat eine hervorgehobene Rolle und übernimmt die „Koordination“ ihrer Teileklassen. Zur Unterscheidung zwischen Aggregationsklasse und Teileklasse(n) wird die Beziehungslinie durch eine Raute auf der Seite der Aggregationsklasse ergänzt. Die Raute symbolisiert das Behälterobjekt, das die Teile aufnimmt.

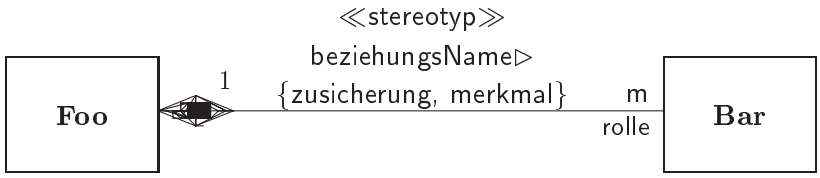
[Hinweis:Verwandte Begriffe für die Aggregation sind die Begriffe Ganzes-Teile-Beziehung und Assoziation.]

Die Abbildung 3.7 auf Seite 42 zeigt den Fall: „Ein Fahrrad hat zwei Laufräder mit jeweils 36 Speichern“. Dieses Beispiel verdeutlich, daß ein Teil (Laufrad) selbst wieder eine Aggregation sein kann.



Legende:
→Abbildung 3.6 auf Seite 41

Abbildung 3.7: Beispiel einer Aggregation: Ein Fahrrad hat zwei Laufräder mit jeweils 36 Speichen



Alternative Notation:

Legende:
→Abbildungen 3.6 auf Seite 41 und 3.1 auf Seite 37

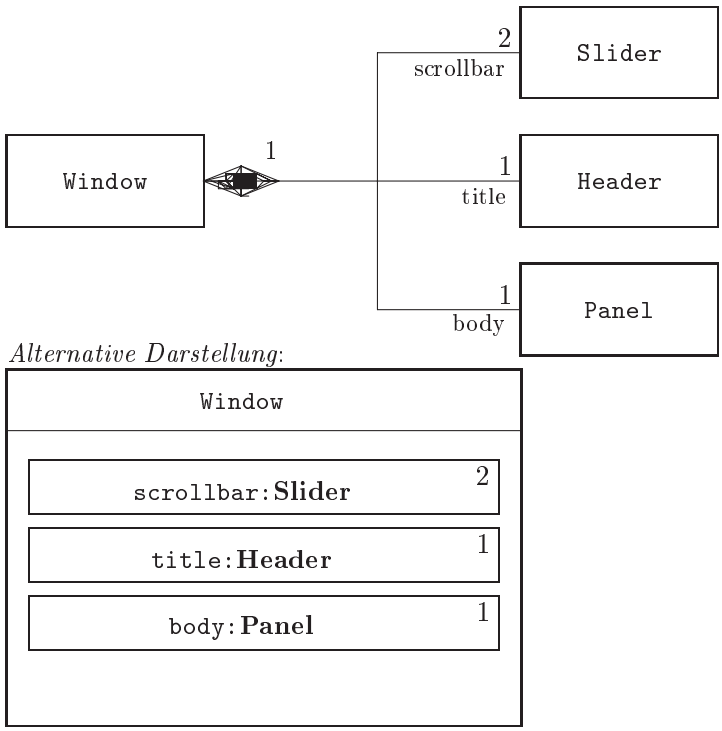
Abbildung 3.8: UML-Beziehungselement: Komposition

3.4.2 Beschreibung: Komposition

Eine Komposition ist eine spezielle Form der Aggregation und damit auch eine spezielle Form der Assoziation. Bei dieser „Ganzes⇔Teile"-Assoziation sind die Teile existenzabhängig vom Ganzen (→Abbildung 3.8 auf Seite 42). Die Lebenszeit eines Teils ist abhängig von der Lebenszeit des Ganzen, das heißt, ein Teil wird zusammen mit dem Ganzen (oder im Anschluß daran) erzeugt und wird vor dem Ende des Ganzen (oder gleichzeitig damit) „vernichtet"[5]. Die Kompositionsklasse hat eine hervorgehobene Rolle und übernimmt die „Koordination" ihrer Teileklassen wie bei der (normalen) Aggregation. Zur Unterscheidung zwischen Kompositionsklasse und Teileklasse(n) wird die Beziehungslinie durch eine ausgefüllte Raute auf der Seite der Kompositionsklasse ergänzt. Auch hier symbolisiert die Raute das Behälterobjekt, das die Teile aufnimmt. Neben der Kennzeichnung durch die ausgefüllte Raute können die Teileklassen auch direkt in den Kasten der Kompositionsklasse geschrieben werden.

[Hinweis:Verwandte Begriffe für die Komposition sind die Begriffe Ganzes-

[5] In Java also nicht mehr referenzierbar.



Legende:
→Abbildung 3.6 auf Seite 41 (Beispielidee [Rational97])

Abbildung 3.9: Beispiel einer Komposition: Ein Window besteht aus zwei Slider, einem Header und einem Panel

Teile-Beziehung und Assoziation.]

Ein Beispiel aus den Bereich *Graphical User Interface* verdeutlicht, daß ein *Window* aus zwei *Slider*, einem *Header* und einem *Panel* besteht. Diese Teile bestehen nicht unabhängig vom Ganzen, dem *Window* (→Abbildung 3.9 auf Seite 43).

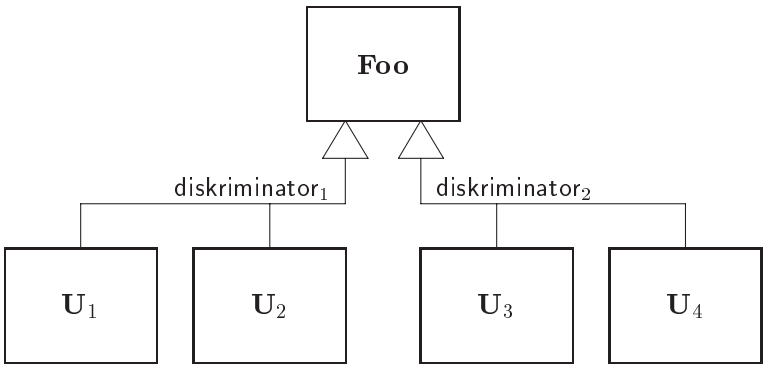
3.5 Beziehungselement: Vererbung

3.5.1 Beschreibung: Vererbung

Vererbung

Als Vererbung bezeichnet man einen Mechanismus, der die Eigenschaften (Variablen und Methoden) einer Klasse (≡ Oberklasse) für eine andere Klasse (≡ Unterklasse) zugänglich macht. Aus der Sicht einer Unterklasse sind die Eigenschaften der Oberklasse eine Generalisierung, das heißt, sie sind für die Unterklasse allgemeine (abstrakte) Eigenschaften. Umgekehrt ist die Unterklasse aus der Sicht der Oberklasse eine Spezialisierung ihrer Oberklasseneigenschaften. Mit der Vererbung wird eine Klassenhierarchie modelliert (→Abbildung 3.10 auf Seite 44). Welche gemeinsamen Eigenschaften von Unterklassen zu einer Oberklasse zusammengefaßt, also generalisiert werden, und umgekehrt, welche Eigenschaften der Oberklasse in

Diskriminator



Legende:

$\triangle$	$\equiv$	Kennzeichnung der Oberklasse Vererbungsrichtung
dikriminator$_j$	$\equiv$	charakteristisches Gliederungsmerkmal für die Gereralisierungs⇔Spezialisierungs-Beziehung
Foo	$\equiv$	Oberklasse von $\mathbf{U}_1$,$\mathbf{U}_2$,$\mathbf{U}_3$ und $\mathbf{U}_4$ Eigenschaften von **Foo** sind in $\mathbf{U}_i$ zugreifbar
$\mathbf{U}_i$	$\equiv$	$\mathbf{U}_i$ ist Unterklasse von **Foo**

Abbildung 3.10: UML-Beziehungselement: Vererbung

Unterklassen genauer beschrieben, also spezialisiert werden, ist abhängig vom jeweiligen charakteristischen Unterscheidungsmerkmal der einzelnen Unterklassen. Ein solches Merkmal wird „Diskriminator“genannt.

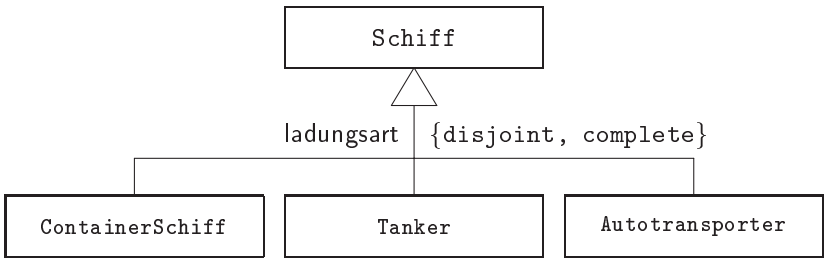
Beispielsweise kann eine Oberklasse `Fahrrad` untergliedert werden nach dem Diskriminator `Verwendungszweck` und zwar in die Unterklassen `Rennrad`, `Tourenrad` und `Stadtrad`. Genau so wäre ein Diskiminator `Schaltungsart` möglich. Dieser ergäbe beispielsweise die Unterklassen `KettenschaltungsFahrrad` und `NabenschaltungsFahrrad`. Welcher Diskriminator zu wählen und wie dieser zu bezeichnen ist, hängt von der gewollten Semantik der Generalisierung⇔Spezialisierung-Relation ab.

Bei der Modellierung einer Vererbung ist es zweckmäßig den Diskrimitator explizit anzugeben. Dabei ist es möglich, daß eine Oberklasse auf der Basis von mehreren Diskriminatoren Unterklassen hat.
[Hinweis:Verwandte Begriffe für die Vererbung sind die Begriffe Generalisierung, Sepzialisierung und *Inheritance*.]

3.5.2 Randbedingungen (*Constraints*)

Bei Modellierung einer Vererbung können für die Unterklassen Randbedingungen (*Constraints*) notiert werden. Vordefiniert sind in UML die Randbedingungen:

- {`overlapping`}
  Ein Objekt einer Unterklasse kann gleichzeitig auch ein Objekt einer anderen Unterklasse sein.



Legende:
→Abbildung 3.10 auf Seite 44
Eine Oberklasse `Schiff` vererbt an die Unterklassen `ContainerSchiff`, `Tanker` und `Autotransporter`.

Abbildung 3.11: Beispiel einer Vererbung

In dem Beispiel „Schiffe“ (→Abbildung 3.11 auf Seite 45) könnte ein Objekt `Emma-II` sowohl ein Objekt der Unterklasse `Tanker` wie auch der Unterklasse `ContainerSchiff` sein, wenn {`overlapping`} angegeben wäre.

- {`disjoint`}
  Ein Objekt einer Unterklasse kann nicht gleichzeitig ein Objekt einer anderen Unterklasse sein.

  In unserem Schiffsbeispiel könnte das Objekt `Emma-II` nur ein Objekt der Unterklasse `Tanker` sein und nicht auch eines der Unterklassen `ContainerSchiff` und `Autotransporter`, weil {`disjoint`} angegeben ist.

- {`complete`}
  Alle Unterklassen der Oberklasse sind spezifiziert. Es gibt keine weiteren Unterklassen. Dabei ist unerheblich ob in dem Diagramm auch alle Unterklassen dargestellt sind.

  In unserem Schffsbeispiel könnte also keine Unterklasse `Kreuzfahrt-Schiff` auftauchen, weil {`complete`} angegeben ist.

- {`incomplete`}
  Weitere Unterklassen der Oberklasse sind noch zu spezifizieren. Das Modell ist noch nicht vollständig. Die Aussage bezieht sich auf die Modellierung und nicht auf die Darstellung. Es wird daher nicht {`incomplete`} angebenen, wenn nur aus zeichnerischen Gründen eine Unterklasse fehlt.

  In unserem Schiffsbeispiel könnte also eine weitere Unterklasse `Kreuz-fahrtSchiff` später modelliert werden, wenn {`incomplete`} angegeben wäre.

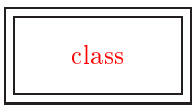
3.6 Pragmatische UML-Namenskonventionen

Setzt sich der Namen aus mehreren ganzen oder abgekürzten Wörtern zusammen, dann werden diese ohne Zwischenzeichen (zum Beispiel ohne „⇔“ oder „_“) direkt hintereinander geschrieben. Durch Wechsel der Groß-/Kleinschreibung bleiben die Wortgrenzen erkennbar.

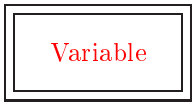
Beispiele zum Wechsel der Groß/Kleinschreibung:
`FahrzeugProg`
`getGeschwindigkeitFahrrad()`
`setFahrtrichtung()`

Die Groß/Kleinschreibung des ersten Buchstabens eines Namens richtet sich (möglichst) nach folgenden Regeln:

- Paket: Der Name beginnt mit einem kleinen Buchstaben.
  Beispiel: `java.lang`



- Klasse oder Schnittstelle: Der Name beginnt mit einem Großbuchstaben.
  Beispiel: `Fahrzeug`
- (Daten-)Typ (zum Beispiel Rückgabetyp): Der Name beginnt in der Regel mit einem Großbuchstaben, weil eine (Daten-)Typangabe eine Klasse benennt. Bei einfachen Java-Datentypen (*PrimitiveType* →Tabelle 5.4 auf Seite 84) beginnen die Namen mit einem kleinen Buchstaben um der Java-Konvention zu entsprechen.
  Beispiele: `int`, `double`, `MyNet`, `HelloWorld`



- Variable oder Methode: Der Name beginnt mit einem kleinen Buchstaben. Ausnahme bildet eine Konstruktor-Methode. Der Konstruktor muß in Java exakt den Namen der Klasse haben. Deshalb beginnt ein Konstruktor stets mit einem Großbuchstaben.
  Beispiel: `Fahrzeug()`, `beschleunigt()`
- Merkmal oder Zusicherung: Der Name beginnt mit einem kleinen Buchstaben.
  Beispiel: {`public`}
- Stereotyp: Der Name beginnt mit einem kleinen Buchstaben.
  Beispiel: {`metaclass`}

3.7 Übung: Modellierung einer Stückliste

Das Unternehmen *RadVertriebsExperten GmbH* (RVE) verkauft im Geschäftsjahr ≈ 5000 Fahrräder, die bei ≈ 7 Herstellern eingekauft werden. RVE beauftragt das Softwarehaus *InterSystems AG* (IAG) ein objekt-orientiertes Warenwirtschaftssystem grob zu planen. Als erstes Diskussionspapier soll die IAG zunächst nur ein Klassendiagramm für die Produkte aus der **Montagesicht** aufstellen.

Im Rahmen dieses Auftrages stellt die IAG bei ihren Recherchen folgende Punkte fest:

1. Alle Produkte sind Fahrräder.
2. Ein Fahrrad ist entweder ein Einrad oder ein Zweirad. Diese beiden Radtypen unterscheiden sich durch die Anzahl ihrer Laufräder.
3. Ein Laufrad wird durch seinen Durchmesser beschrieben.
4. Jedes Fahrrad hat eine Rahmennummer.
5. Schutzbleche und Gepäckträger sind Anbauteile.
6. Jedes Anbauteil hat eine Teilenummer.
7. Ein Schutzblech wird durch die Materialart und die Breite beschrieben.
8. Ein Gepäckträger wird durch die Tragkraft beschrieben.
9. Nur an ein Zweirad können Anbauteile montiert werden.
10. Es werden stets zwei Schutzbleche montiert.
11. Es werden maximal zwei Gepäckträger montiert.
12. Zu jedem Anbauteil gibt es eine Montageanleitung.
13. Die Montageanleitung nennt die durchschnittliche Montagedauer und hat einen Text, der die Vorgehensschritte beschreibt.
14. Jedem Fahrrad ist anzusehen, ob es probegefahren wurde.

3.7.1 Klassendiagramm für die Montagesicht

Entwerfen Sie ein Klassendiagramm für die RVE. Notieren Sie Ihr Klassendiagramm in UML. Es sollte möglichst viele der obigen Punkte abbilden.

3.7.2 Diagrammerweiterung um den Montageplatz

In der ersten Diskussionsrunde mit der RVE möchte Herr Abteilungsleiter Dr. Moritz Krause unbedingt den Montageplatz noch aufgenommen haben. Ein Montageplatz ist ausgestattet nach der Vorgabe G (≡ Grundausstattung) oder S (≡ Sonderausstattung). Skizzieren Sie die notwendige Ergänzung in Ihrem Klassendiagramm.

Kapitel 4

Java ≈ mobiles Code-System

Write Once, Run Everywhere.
Java-Slogan der Sun Microsystems, Inc. USA

Klassen mit Variablen und Methoden, Assoziationen, Aggregationen, Kompositionen und Vererbung sind (nun bekannte) Begriffe der Objekt-Orientierung. Java ist jedoch mehr als eine objekt-orientierte Programmiersprache. Java ist (fast) ein *mobiles Code-System*.

Java ermöglicht es, Code einschließlich Daten über Netzknoten, also über Computer in den Rollen eines Clients und eines Servers, problemlos zu verteilen und auszuführen. Ein Stück mobiler Java-Code (*Applet*) wird dynamisch geladen und von einem eigenständigen („standalone“) Programm ausgeführt. Ein solches Programm kann ein WWW-Browser, Appletviewer oder WWW-Server sein.

Trainingsplan

Das Kapitel „Java ≈ mobiles Code-System“ erläutert:

- das Zusammenspiel von Java und dem *World Wide Web*,
  ↪ Seite 50 ...
- die Portabilität aufgrund des Bytecodes,
  ↪ Seite 52 ...
- das Sicherheitskonzept und
  ↪ Seite 54 ...
- skizziert den Weg der Umstellung auf eine Softwareentwicklung mit Java.
  ↪ Seite 56 ...

4.1 Java im Netz

Java ist auch ein System, um im Internet ausführbaren Code auszutauschen. Eine Anwendung im *World Wide Web* (WWW) kann zusätzlich zum gewohnten Laden von Texten, Graphiken, Sounds und Videos den Java Bytecode laden und diesen direkt ausführen. Über einen vorgegebenen Fensterausschnitt des Browsers kann dann das Applet mit dem Benutzer kommunizieren.

Java ist daher auch ein *mobiles Code-System*.[1] Ein solches System ermöglicht es, Code einschließlich Daten über Netzknoten, also über Computer in den Rollen eines Clients und eines Servers, zu verteilen. Ein mobiles Objekt, in der Java-Welt als *Applet* bezeichnet, ist selbst ein Stück ausführbarer Code. Ebenso wie traditionelle Software enthält auch ein mobiles Objekt eine Sequenz von ausführbaren Instruktionen. Anders jedoch als bei traditioneller Software wird ein mobiles Objekt, also ein Applet, dynamisch geladen und von einem eigenständigen („standalone") Programm ausgeführt. Ein solches Programm kann ein WWW-Browser, Appletviewer oder WWW-Server sein (→Abbildung 4.1 auf Seite 51). Das WWW-Szenario der Client⇔Server-Interaktionen läßt sich mit folgenden Schritten skizzieren:

Request

1. Anfordern des Applets (*request*-Schritt)
   Ein Java-fähiger WWW-Browser (Client) fordert das Applet vom Server an, wenn er im empfangenen HTML-Dokument ein `<APPLET>`-Konstrukt feststellt. Das Attribut `CODE` der `<APPLET>`-Marke hat als Wert den Namen des Applets, also den Dateinamen der Java-Bytecodedatei mit dem Suffix `class`. Typischerweise befindet sich diese Java-Bytecodedatei auf demselben Server wie das angefragte HTML-Dokument.

Download

2. Empfangen des Applets (*download*-Schritt)
   Der Browser initiiert eine eigene TCP/IP-Session um das Applet vom Server herunterzuladen (*download*). Der Browser behandelt dabei das Applet wie andere HTML-Objekte, zum Beispiel wie eine Video- oder Sounddatei.

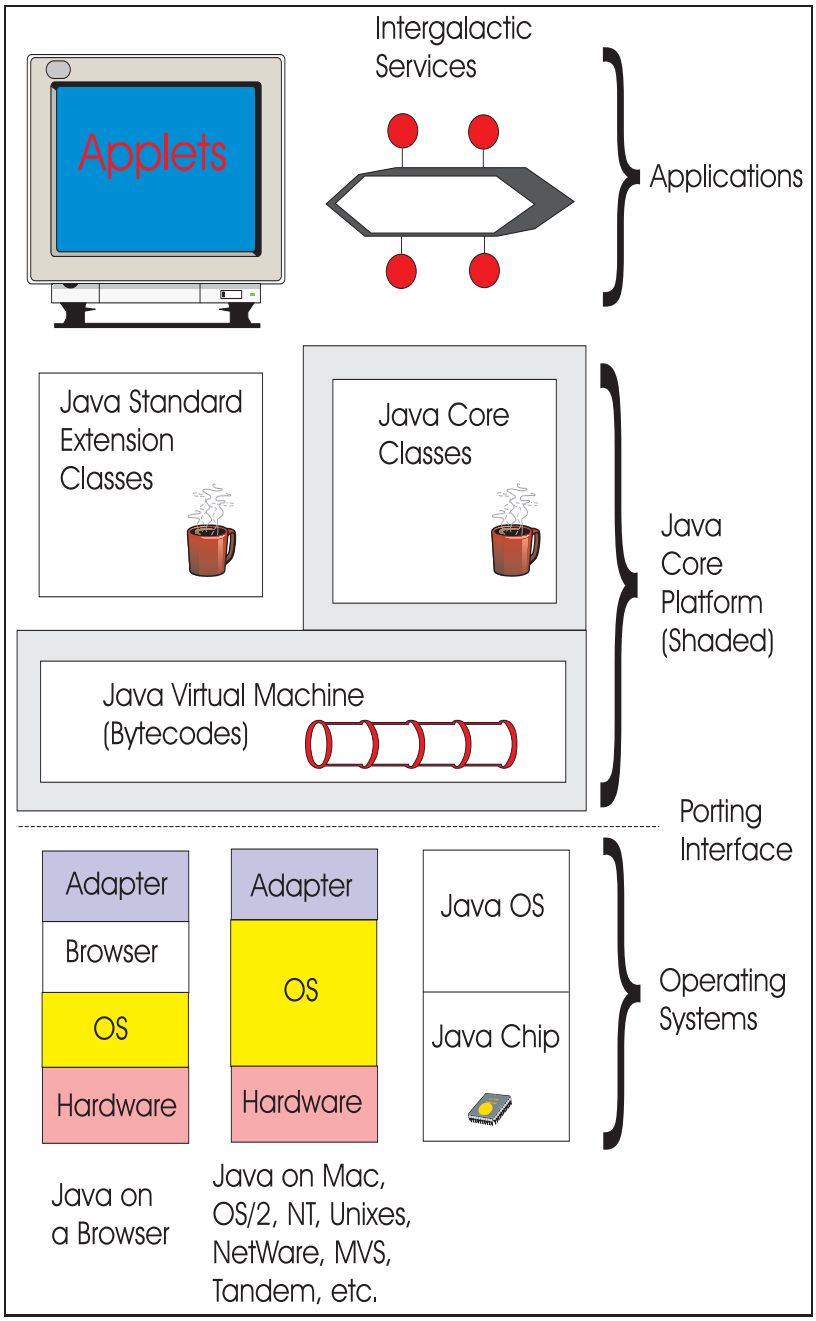
Execute

3. Laden und ausführen des Applets (*execute*-Schritt)
   Der Browser läd das Applet in den Arbeitsspeicher des Client und stößt seine Ausführung an. Typischerweise kreiert ein Applet graphische Ausgaben und reagiert auf Eingaben (Keyboard und Maus). Dies geschieht alles in einer festgelegten Bildschirmfläche der angezeigten HTML-Seite. Die Größe dieser Fläche wird durch die Werte der Attribute `WIDTH` und `HEIGHT` bestimmt. Die Applet-Fläche kann mit anderen Applet-Flächen „kommunizieren", aber nicht mit dem Rest der HTML-Seite, das heißt, sie ist isoliert davon.

Delete

4. Stoppen und löschen des Applets (*delete*-Schritt)
   Der Browser stoppt die Ausführung des Applet und gibt den Arbeits-

[1] Ein anderes Beispiel für ein mobiles Code-System ist *Safe-Tcl* →[Orfali/Harkey97].



Bildidee: [Orfali/Harkey97]

Abbildung 4.1: Die Java-Plattformen

speicher des Client wieder frei. Dies geschieht beim „Verlasssen“ des HTML-Dokumentes.

Jedes mobile Code-System, also auch Java, sollte die beiden Kernforderungen *Portabilität* und *Sicherheit* möglichst gut erfüllen. Dafür muß es (mindestens) folgende Aspekte abdecken:

1. Portabilität
   (a) Eine Plattformunabhängigkeit der gesamten Leistungen

   Ein mobiles Code-System muß ein plattform-übergreifendes Management des Arbeitsspeichers bereitstellen. Parallel ablaufende Prozesse (*threads*) und ihre Kommunikation inclusive ihrer Synchronisation sind unabhängig vom jeweiligen Betriebssystem der Plattform zu realisieren. Die gleiche Plattformunabhängigkeit wird auch für die graphische Benutzungsschnittstelle (GUI) erwartet.

   (b) Ein Kontrollsystem für den ganzen Lebenszyklus

   Ein mobiles Code-System muß die Laufzeitumgebung für das Laden, Ausführen und das „Entladen“ des Codes bereitstellen.

2. Sicherheit
   (a) Eine kontrollierbare Ausführungsumgebung für den mobilen Code (*safe environment*)

   Bei einem mobilen Code-System muß der Anwender in der Lage sein, die Ausführungsumgebung des Codes (Applets) präzise zu steuern, das heißt, den Zugriff auf den Arbeitsspeicher und auf das Dateisystem, den Aufruf von Systemroutinen und das Nachladen von Servern muß kontrollierbar sein.

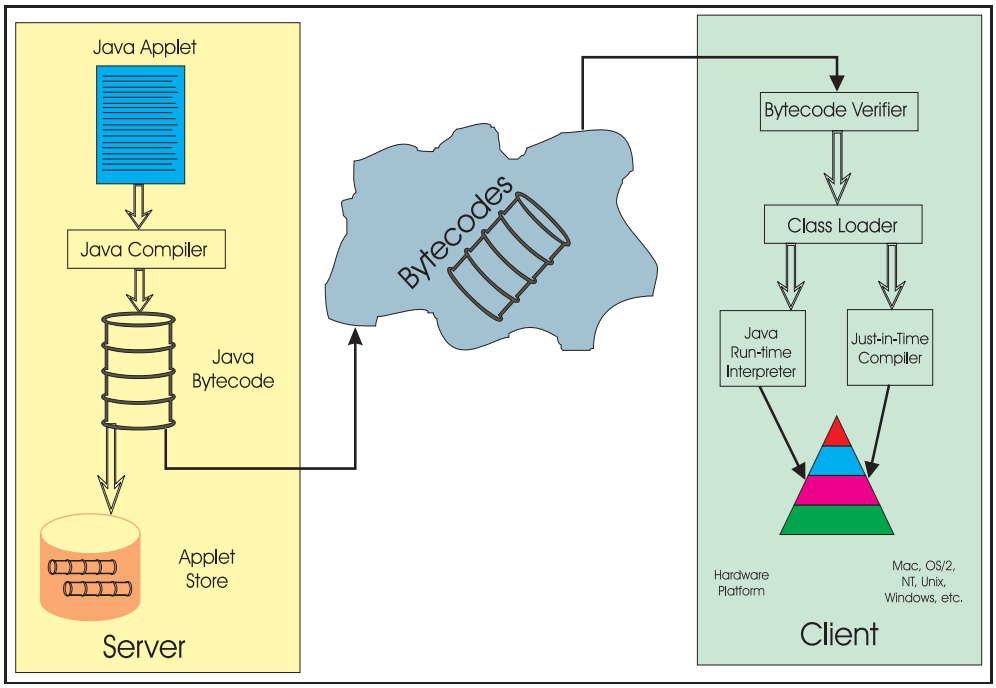
   (b) Eine sichere Code-Verteilung über das Netz

   Ein mobiles Code-System muß den Transfer des Codes (Applets) über das Netz sicher, also unverfälscht, gestalten. Dazu ist die Authentifikation sowohl auf Client- wie auf Server-Seite erforderlich. Es ist zu gewährleisten, daß der Client beziehungsweise der Server wirklich derjenige ist, der er vorgibt zu sein. Zusätzlich ist der Code zu zertifizieren. Pointiert formuliert: Es ist alles zu tun, damit der Code nicht von Viren infiziert werden kann.

4.2 Bytecode: Portabilität ⇔ Effizienz

Bytecode

Java realisiert die Portabilität indem der Java-Quellcode übersetzt wird in primitive Instruktionen eines virtuellen Prozessors. Diese maschinennäheren, primitiven Instruktionen nennt man Bytecode. Das Compilieren bezieht sich bei Java nicht auf den Befehlssatz eines bestimmten, marktüblichen



Bildidee: [Orfali/Harkey97]

Abbildung 4.2: Von der Bytecode-Produktion bis zur Ausführung

Prozessors, sondern auf die sogenannte *Java Virtual Machine*. Der Bytecode bildet eine möglichst maschinennahe Code-Ebene ab, ohne jedoch, daß seine einzelnen Instruktionen wirklich maschinenabhängig sind. Darüberhinaus legt Java die Größe seiner einfachen Datentypen (*PrimitiveType*→Tabelle 5.4 auf Seite 84) und das Verhalten seiner arithmetischen Operatoren präzise fest. Daher sind Rechenergebnisse stets gleich, also unabhängig davon, ob die jeweilige Plattform 16-, 32- oder 64-Bit-basiert ist.

Der Bytecode macht Java zu einer sogenannten „partiell-compilierten" Sprache (→Tabelle 4.2 auf Seite 53). Um den Bytecode aus dem Java-Quellcode zu erzeugen, ist ungefähr 80% des gesamten Compilationsaufwandes notwendig; die restlichen 20% entfallen auf Arbeiten, die das Java-Laufzeitsystem übernimmt. So kann man sich Java als 80% compiliert und 20% interpretiert vorstellen. Dieser 80/20-Mix führt zu einer exzellenten Code-Portabilität bei gleichzeitig relativ guter Effizienz, da der Java-Bytecode eine gelungene, recht maschinennahe Abstraktion über viele Plattformen darstellt. Trotz alledem ist der Java-Bytecode beim Interpretieren über 15mal[2] langsamer als maschinenspezifisch compilierter Code (≡ *native code*). Um diesen Nachteil an Effizienz auszuräumen gibt es auch für Java heute *Just-In-Time*-Compiler[3] (JIT) und reguläre, maschinenabhängige

80/20-Mix

JIT

[2] →[Orfali/Harkey97] page 32.

[3] Ein JIT-Compiler konvertiert Java's Stack-basierte Zwischenrepräsentation in den benötigten (*native*) Maschinencode und zwar unmittelbar vor der Ausführung. Die Bezeichnung „Just-In-Time" vermittelt den Eindruck einer rechtzeitigen (schnellen) Programmausführung. Aber der JIT-Compiler erledigt seine Arbeit erst nachdem man der

Compiler.

Bei genauer Betrachtung läuft jedes Java-Programm nur einen sehr geringen Prozentsatz seiner Zeit wirklich in Java. Java schafft nur den Eindruck über jede Plattform-Architektur Alles exakt zu kennen. Wie soll Java bespielsweise wissen wie eine Linie auf dem Bildschirm für jede möglich Plattform gezogen wird. Jedes Betriebssystem in dem Java heute üblicherweise läuft nutzt dafür Routinen, geschrieben in anderen Sprachen, zum Beispiel in C oder C++. Egal ob nun Etwas auf dem Bildschirm auszugeben ist oder ein *Thread*[4] oder eine TCP/IP-Verbindung zu meistern sind, das was Java tun kann, ist das jeweilige Betriebssystem zu beauftragen diese Dinge zu tun. So wird letztlich eine Java-Anwendung, die überall läuft, hauptsächlich über die Abarbeitung von Routinen in *C*-Code realisiert.

4.3 Sicherheit

4.3.1 Prüfung des Bytecodes (*Bytecode Verifier*)

Verifier

Für die Sicherheit ist in den Bytecode-Zyklus von der Erzeugung über das Laden bis hin zur Ausführung ein Schritt der Code-Verifizierung eingebaut. Zunächst wird der Java-Quellcode zu Bytecode compiliert. Danach wird dieser Bytecode üblicherweise über das Netz zum nachfragenden Client transferiert. Bevor der Bytecode dort ausgeführt wird, durchläuft er den Bytecode-Verifizierer. Dieser prüft den Bytecode in vielerlei Hinsicht, beispielsweise auf nachgemachte Zeiger, Zugriffsverletzungen, nicht passende Parametertypen und auf Stack-Überlauf. Man kann sich den Java-Verifizierer als einen Türkontrolleur vorstellen, der aufpaßt, daß kein unsicherer Code von außerhalb oder auch von der lokalen Maschine Eintritt zur Ausführung hat. Erst nach seinem OK wird das Laden der Klassen aktiviert. Dieses übernimmt der Klassenlader (*class loader*). Er übergibt den Bytecode an den Interpreter. Dieser ist das Laufzeitelement, das die Bytecode-Instruktionen auf der Arbeitsmaschine in die dortigen Maschinenbefehle umsetzt und zur Ausführung bringt.

4.3.2 Java traut niemandem

Java's Sicherheitsphilosophie ist geprägt von der Annahme, daß niemandem zu trauen ist. Dieses Mißtrauen hat zu einem Konzept der Rundumverteidigung geführt. Diese beschränkt sich nicht nur auf den Bytecode-Verifizierer, sondern setzt bei der Sprache selbst an und bezieht selbst den Browser mit ein. Im folgenden sind einige Aspekte dieser Rundumverteidigung skizziert:

1. Sicherheit durch das *Memory Layout* zur Laufzeit
   Ein wichtiger Sicherheitsaspekt liegt in der Entscheidung über die

Anwendung gesagt hat: „`run`“. Die Zeit zwischen diesem Startkommando und dem Zeitpunkt, wenn das übersetzte Programm wirklich beginnt das Gewünschte zu tun, ist Wartezeit für den Anwedner. Ein mehr passender Name wäre daher „*Wait to the Last Minute Holding Everybody Up Compiler*“ [Tyma98] p. 42.

[4] Näheres dazu →Abschnitt 6.1 auf Seite 96.

Bindung von Arbeitsspeicher (*Memory*). Im Gegensatz zu den Sprachen `C` und `C++` wird vom Java-Compiler nicht das Memory-Layout entschieden. Es wird erst abgeleitet zur Laufzeit. Dieser Mechanismus einer späten Bindung verhindert es, aus der Deklaration einer Klasse auf ihr physikalisches Memory-Layout zu schließen. Eine solche Kenntnis war stets ein Tor für „Einbrüche“.

2. Sicherheit durch Verzicht auf Zeiger
   Java verzichtet auf Zeiger (*Pointer*) in der Art wie sie in den Sprachen `C` und `C++` vorkommen und dort auch häufig im Sinne schwerdurchschaubarer Codezeilen genutzt werden. Java kennt keine Speicherzellen, die ihrerseits wieder Adressen zu anderen Zellen speichern. Java referenziert Arbeitsspeicher nur über symbolische „Namen“, deren Auflösung in konkrete Speicheradressen erst zur Laufzeit durch den Java-Interpreter erfolgt. Es gibt daher keine Gelegenheit, Zeiger zu „verbiegen“, um hinterrücks etwas zu erledigen.

   keine Zeiger

3. Sicherheit durch eigene Namensräume
   Der Java-Klassenlader unterteilt die Menge der Klassen in unterschiedliche Namensräume. Eine Klasse kann nur auf Objekte innerhalb ihres Namensraumes zugreifen. Java kreiert einen Namensraum für alle Klassen, die vom lokalen Dateisystem kommen, und jeweils einen unterschiedlichen Namensraum für jede einzelne Netzquelle. Wird eine Klasse über das Netz importiert, dann wird sie in einen eigenen Namensraum plaziert, der mit ihrer Quelle (WWW-Server) assoziiert ist. Wenn eine Klasse `Foo` die Klasse `Bar` referenziert, dann durchsucht Java zuerst den Namensraum des lokalen Dateisystems (*built-in classes*) und danach den Namensraum der Klasse `Foo`.

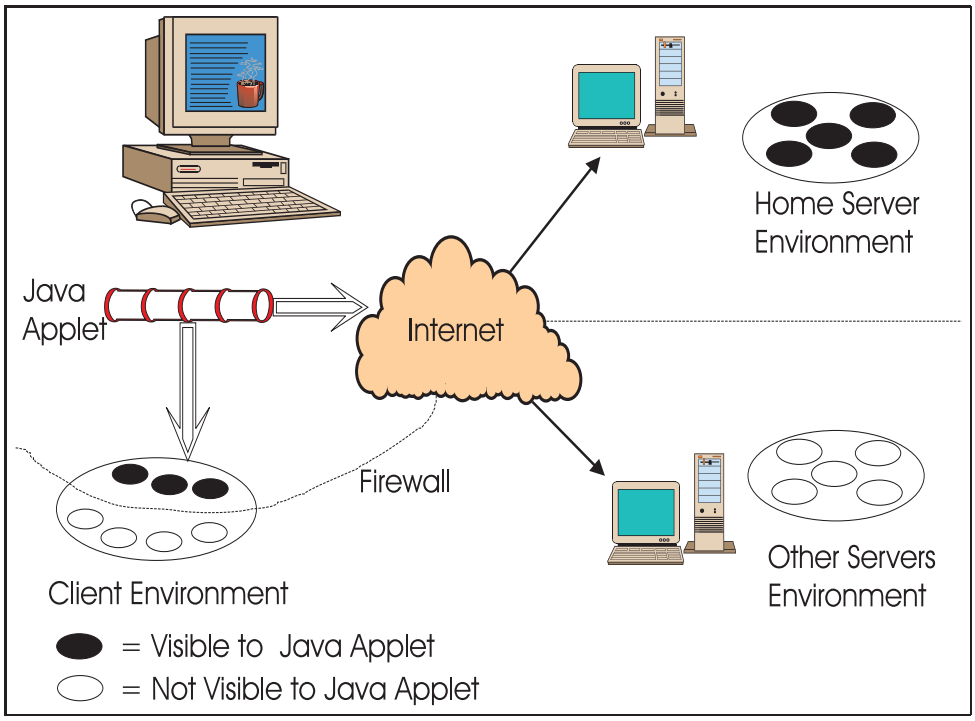
   Namen

4. Sicherheit durch Zugriffs-Kontroll-Listen
   Die Dateizugriffskonstrukte implementieren die sogenannten Kontroll-Listen. Damit lassen sich Lese- und Schreib-Zugriffe zu Dateien, die vom importierten Code ausgehen oder von ihm veranlaßt wurden, benutzungsspezifisch kontrollieren. Die Standardwerte (*defaults*) für diese Zugriffs-Kontroll-Listen sind äußerst restriktiv.

5. Sicherheit durch Browser-Restriktionen
   Moderne Browser unterscheiden verschiedene Sicherheitsstufen. So lassen sich Netzzugriffe eines Applets unterbinden oder auf den Bereich einer Sicherheitszone (*Firewall-Bereich*) begrenzen (→Tabelle 4.3 auf Seite 56).

6. Sicherheit durch zusätzliche Applet-Zertifizierung
   Mit Hilfe der Kryptologie läßt sich die Sicherheit wesentlich steigern. So kann beispielsweise über das Verfahren *Pretty Good Privacy* (PGP) ein Applet unterschrieben (signiert) werden. Veränderungen am Bytecode auf dem Transport werden sofort erkannt. Der liefernde Server und der nachfragende Client können einer Authentifikation unterzogen werden, das heißt, sie müssen sich ordungsgemäß ausweisen.

   PGP



Bildidee: [Orfali/Harkey97]

Abbildung 4.3: Applet-Erreichbarkeit mit *Host-Mode*-Restriktion

4.4 The Road To Java

Umstellung

Sun Microsystems formliert unter dem Motto „The Road to Java“ (→[Sun97] S. 4) fünf Meilensteine, die den Weg hin zu einer Java Computing Architektur markieren:

1. Erhebung (*Investigate*):
   Sammlung von Informationen über Java und über die Geschäftsauswirkungen von Java Computing.

2. Bewertung (*Evaluate*):
   Bewertung von Technologien und Geschäftsauswirkungen im Rahmen des jeweiligen Unternehmens.

3. Gestaltung (*Architect*):
   Entwicklung einer um Java Computing erweiterten Architektur der bisherigen Informationstechnologie.

4. Pilotierung (*Pilot*):
   Initiierung von Pilotprojekten, um Erfahrungen zu sammeln.

5. Betrieb (*Implement*):
   Implementierung und unternehmensweite Umsetzung der Java Computing Architektur.

SNiFF+J (Release 2.3.2) unterstützt die Entwicklungsaufgaben:

- Visualisierung großer Datenmengen (*code comprehension and browsing*),
- Verteilen von Änderungen (*development*),
- Hypertextdokumentation (*documentation building*),
- Teamkoordination (*project and code management for teams*),
- Zugriffsverwaltung (*version and configuration management*),
- Versionsverwaltung (*build management*),
- Fehlersuche (*debugging support*) und
- Schnittstellen zu Werkzeugen (*tool and control integration*).

Quelle: TakeFive Software [TakeFive97]

Tabelle 4.1: SNiFF+J: Entwicklungsumgebung für große Java-Projekte

Übliche Enwicklungswerkzeuge für Java (z.B.: Java Workshop 2.0 von Sun Microsystems, Inc. oder Visual Age for Java von IBM) unterstellen einen einzelnen Systementwickler statt ein Team von Konstrukteuren. Sie enthalten beispielsweise Versionsmanagement oder Test- und Freigabeunterstützung nur in Ansätzen. Erst langsam integrieren Entwicklungsumgebungen, die sehr große Projekte („Millionen Zeilen Quellcode“) und mehrere Entwicklungsteams unterstützen können, neben C++ auch Java (→Tabelle 4.1 auf Seite 57).

Enwicklungsumgebung für die Beispiel im $\mathcal{JAVA} \Leftrightarrow \mathcal{COACH}$
Die Beispiele wurden auf den beiden folgenden Plattformen entwickelt:

- *AIX-Plattform*:
  IBM[5] RISC[6]-Workstation RS[7]/6000, Typ 250 und Typ 43p, Betriebssystem AIX[8] 4.1

AIX

- *NT-Plattform*:
  Intel PC, Betriebssystem Microsoft Windows NT[9]

NT

Im Rahmen dieses Buches wird die Anwendungsentwicklung auf der Basis einer minimalen Java-Entwicklungsumgebung erläutert. Diese Basisentwicklungsumgebung besteht aus:

1. **Java Development Kit** (JDK), Version $\geq$ 1.1.4

JDK

[5] IBM ≡ International Business Machines Corporation
[6] RISC ≡ Reduced Instruction Set Computer
[7] RS ≡ RISC System
[8] AIX ≡ Advanced Interactive Executive — IBM's Implementation eines *UNIX-Operating System*
[9] NT ≡ New Technology, 32-Bit-Betriebssystem mit Multithreading und Multitasking

(a) `appletviewer` — Java-Appletviewer

(b) `java` — Java-Interpreter

(c) `javac` — Java-Compiler

(d) `javadoc` — Java-Dokumentations-Generator

(e) `javah` — native Methoden, C-Dateigenerator

(f) `javap` — Java-Klassen-Disassembler

(g) `jdb` — Java-Debugger

Emacs

2. **Editor** und **Shell**:
   Emacs[10], verschiedene Versionen zum Beispiel `xemacs`, GNU[11] emacs oder Micro Emacs.

3. **Browser**:
   Netscape Communicator Version $\geq$ *4.03*

4. **Datenaustausch** im Netz:
   File Transfer Program (FTP) auf Basis von TCP/IP[12]

5. **Prozeß- und Produkt-Dokumentation**:
   HTML-Dateien auf einem WWW-Server, erstellt mittels Emacs (→Punkt 2)

Darüber hinaus werden die Entwicklungsumgebung `SNiFF+J` von TakeFive [TakeFive97] und das CASE-Tool Innovator von MID eingesetzt. Der Vorteil solcher Werkzeuge liegt in der Unterstützung von mehreren Programmieren, also von Teamarbeit.

[10] Emacs ≡ Editing Macros — gigantic, functionally rich editor
[11] GNU ≡ Free Software Foundation: GNU stands for „GNU's Not Unix“
[12] TCP/IP ≡ Transmission Control Protocol / Internet Protocol — communications protocol in UNIX environment

Kapitel 5

Java-Konstrukte (Bausteine zum Programmieren)

In der Java-Welt werden Applikationen (≡ „eigenständige“ Programme) von Applets (≡ Programm eingebettet in eine HTML-Seite) unterschieden. Beide benutzen die Bausteine (*Konstrukte*[1]) aus dem *Java Development Kit* (JDK). Praxisrelevante Konstrukte werden anhand von Beispielen eingehend erläutert.

Trainingsplan

Das Kapitel „Java-Konstrukte“ erläutert:

- Klassen, Variablen, Methoden, Parameter, Konstruktoren, Internetzugriff und GUI-Bausteine anhand von ersten Kostproben,
  ↪ Seite 60 ...
- den Unterschied zwischen Applet und Applikation,
  ↪ Seite 75 ...
- das Einbinden eines Applets in ein HTML-Dokument,
  ↪ Seite 75 ...
- die Syntax, die Semantik und die Pragmatik.
  ↪ Seite 82 ...

[1] Der lateinische Begriff Konstrukt (Construct, Constructum) bezeichnet eine Arbeitshypothese für die Beschreibung von Phänomenen, die der direkten Beobachtung nicht zugänglich sind, sondern nur aus anderen beobachteten Daten erschlossen werden können. In der Linguistik ist zum Beispiel Kompetenz ein solches Konstrukt. Im JAVA–COACH wird jeder verwendbare „Baustein“, der bestimmte Eigenschaften hat, als Konstrukt bezeichnet.

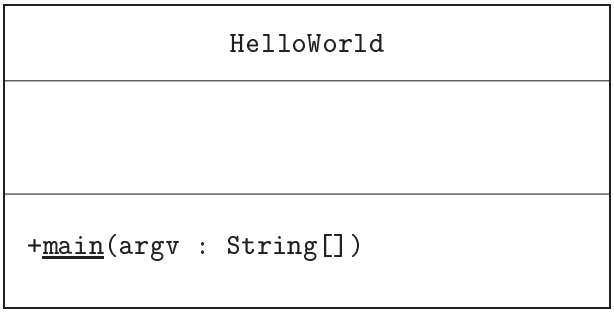


Abbildung 5.1: Klassendiagramm für `HelloWorld.java`

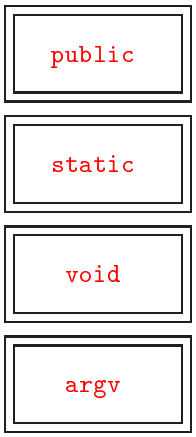
5.1 Einige Java-Kostproben

Die folgenden Kostproben enthalten zum Teil Java-Konstrukte, die erst später eingehender erläutert werden. Der Quellcode der Beispiele dient primär zum schnellen Lernen der Syntax, Semantik und Pragmatik von Java. Er ist nicht im Hinblick auf Effizienz optimiert oder entsprechend eines einheitlichen (Firmen-)Standards formuliert.
[Hinweis: Die Zeilennummerierung ist kein Quellcodebestandteil.]

5.1.1 Kostprobe `HelloWorld.java`

Jeder Einstieg in eine formale (Programmier-)Sprache beginnt mit der Ausgabe der Meldung „Hello World“ auf dem Bildschirm. Dies entspricht einer „alten“ Informatik-Gepflogenheit. Die Abbildung 5.1 auf Seite 60 zeigt das Klassendiagramm für die Java-Applikation `HelloWorld.java`.

Die Startmethode einer Java-Applikation ist `main()`. Sie muß folgende Eigenschaften haben:



- Sie muß von außen zugreifbar sein ↪Kennwort `public`.
- Da ihr Klassenname beim Aufruf des Java-Interpreters angegeben wird (— und nicht eine Instanz ihrer Klasse —) muß `main()` selbst eine Klassenmethode sein ↪Kennwort `static`.
- Da keine Rückgabe eines Wertes organisiert wird (— Wohin damit? —), muß `main()` ohne Rückgabewert definiert werden ↪Kennwort `void`).
- Die Methode `main()` weist nur einen Parameter auf. Dieser wird üblicherweise `argv` (*argument value*) oder `args` (*arguments*) genannt. Er ist als ein Feld von Zeichenketten (`String`) deklariert.

```
1  /**
2   Java-Kostprobe "Hello World"
3   @author Hinrich Bonin
4   @version 1.0
```

```
 5     22-Feb-1997
 6    Update: 27-Oct-1997, 13-Jul-1998
 7
 8  */
 9  public class HelloWorld {
10    // Hauptprogram, Klassenmethode main()
11    public static void main(String argv[]) {
12
13    // \n im String bedeutet neue Zeile (newline)
14    // + hier zum Verknüpfen von Strings
15    System.out.println("\n" +
16      "*** Hello (wonderful) world! *** \n" +
17      "================================\n\n");
18
19    System.out.println(
20      "Erste (Integer-)Divisionsaufgabe: "
21      + "Teile 100 durch 30" );
22
23    System.out.println(
24      "Ergebins: " +  100 / 30 + " plus Rest: " +
25      100 % 30 );
26
27    System.out.println("Oder IEEE-Rest: " +
28      Math.IEEEremainder(100, 30) + "\n");
29
30    System.out.println(
31      "Zweite (Gleitkomma-)Divisionsaufgabe: " +
32      "Teile 100.0 durch 30.0" );
33
34    System.out.println("Ergebins: " +
35      100.0 / 30.0 + "\n" +
36      "  mit Rundungsüberprüfung " +
37      "(100.0  - (( 100.0 / 30.0 ) * 30.0)) = " +
38      ( 100.0  - (( 100.0 / 30.0 ) * 30.0))  +
39      "\n");
40
41    System.out.println(
42      "Erste (Gleitkomma-)Multiplikationsaufgabe: \n" +
43      "  Multipliziere 3.33333 mit 30.0" );
44
45    System.out.println("Ergebins: " +
46      ( 3.33333 * 30.0 ) + "\n");
47
48    System.out.println(
49      "Werte von mathematischen Konstanten: \n" +
50      "Wert e = " + Math.E + "\n" +
51      "Wert pi= " + Math.PI + "\n");
52
53    System.out.println(
54      "Wert von Wurzel aus 3 zum Quadart, d.h. \n" +
55      "Math.pow(Math.sqrt(3), 2) = " +
56      Math.pow(Math.sqrt(3), 2));
57
58    System.out.println("\n" + "Bonin (c) Copyright 1997 \n");
```

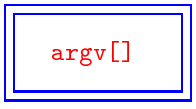
```
59    }
60  }
61  // End of File cl3:/u/bonin/myjava/HelloWorld.java
```

Compilation und Ausführung von `HelloWorld`:

```
cl3:/home/bonin/myjava:>java -version
java version "1.1.2"
cl3:/home/bonin/myjava:>javac HelloWorld.java
cl3:/home/bonin/myjava:>java HelloWorld

*** Hello (wonderful) world! ***
================================


Erste (Integer-)Divisionsaufgabe: Teile 100 durch 30
Ergebins: 3 plus Rest: 10
Oder IEEE-Rest: 10.0

Zweite (Gleitkomma-)Divisionsaufgabe: Teile 100.0 durch 30.0
Ergebins: 3.3333333333333335
  mit Rundungsüberprüfung (100.0  - (( 100.0 / 30.0 ) * 30.0)) = 0.0

Erste (Gleitkomma-)Multiplikationsaufgabe:
  Multipliziere 3.33333 mit 30.0
Ergebins: 99.9999

Werte von mathematischen Konstanten:
Wert e = 2.718281828459045
Wert pi= 3.141592653589793

Wert von Wurzel aus 3 zum Quadart, d.h.
Math.pow(Math.sqrt(3), 2) = 2.9999999999999996

Bonin (c) Copyright 1997

cl3:/home/bonin/myjava:>
```

5.1.2 Kostprobe `Foo.java` — Parameterübergabe der Applikation

Das Beispiel `Foo.java` zeigt wie Argumente beim Aufruf der Java-Applikation übergeben werden. Die Werte der Argumente werden als Zeichenkette (Datentyp `String`) an den einen Parameter der Methode `main()` gebunden. Dieser Parameter ist vom Datentyp `Array` und umfaßt soviele Felder wie es Argumente gibt. Die Adressierung dieser Felder beginnt mit dem Wert 0; das heißt, der Wert des erste Arguments „steht“ im nullten Feld (*zero based*). Die Abbildung 5.2 auf Seite 63 zeigt das Klassendiagramm für die Java-Applikation `Foo.java`.




```
public static void main(String argv[]) { ... }
```

```
1  /**
```

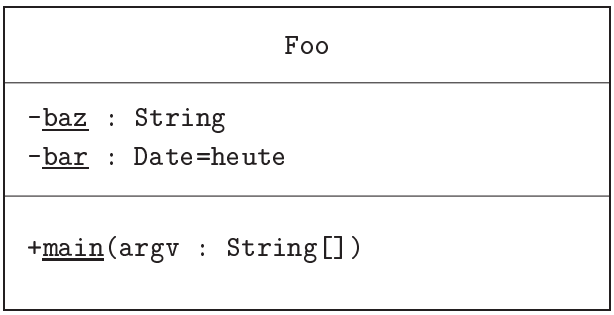


Abbildung 5.2: Klassendiagramm für `Foo.java`

```
 2   Kleiner Java Spaß: Demonstration der Bindung
 3   des Parameters an die Argumente ("call by value")
 4   Bonin 14-Oct-1997
 5     Update 21-Oct-1997; 24-Oct-1997;
 6            29-Oct-1997; 03-Nov-1997; 20-Apr-1998
 7  */
 8  import java.util.*;
 9
10  public class Foo {
11
12    private static String baz;
13    private static Date bar = new Date();
14
15    public static void main(String argv[]) {
16
17      for (int i =0; i < argv.length; i = i + 1)
18        {
19        System.out.println("Eingabeteil:" + i);
20        System.out.println("+" + argv[i] + "+");
21        };
22      if (argv.length != 0)
23        {
24         argv[argv.length -1] = "Neuer Wert: ";
25         baz = argv[argv.length -1]; }
26        else
27         baz  = "Kein Argument: ";
28
29    System.out.println(
30      baz  + "Java ist interessant! " + bar.toString());
31    }
32  }
33  // End of file: /u/bonin/myjava/Foo.java
```

Nach dem Aufruf von:

```
>java Foo is my $WWW_HOME
```

ist der Wert des Arguments `WWW_HOME` nicht verändert, weil die Shell des Betriebssystems, in UNIX-Welt zum Beispiel die Korn-Shell, die Variable

`WWW_HOME` als Wert übergibt.

Compilation und Ausführung von `Foo`:

```
cl3:/u/bonin:>java -fullversion
java full version "JDK1.1.2 IBM build a112-19971017"
cl3:/u/bonin:>export CLASSPATH=/home/bonin/myjava:$CLASSPATH
cl3:/u/bonin:>javac myjava/Foo.java
cl3:/u/bonin:>ls -l Foo.class
Foo.class not found
cl3:/u/bonin:>java Foo
Kein Argument: Java ist interessant! Fri Oct 24 11:14:48 GMT+01:00 1997
cl3:/u/bonin:>echo $WWW_HOME
http://www.fh-lueneburg.de/
cl3:/u/bonin:>java Foo is my $WWW_HOME
Eingabeteil:0
+is+
Eingabeteil:1
+my+
Eingabeteil:2
+http://www.fh-lueneburg.de/+
Neuer Wert: Java ist interessant! Fri Oct 24 11:15:18 GMT+01:00 1997
cl3:/u/bonin:>echo $WWW_HOME
http://www.fh-lueneburg.de/
cl3:/u/bonin:>
```

5.1.3 Kostprobe `FahrzeugProg.java` — Konstruktor

In diesem Beispiel einer Java-Applikation sind drei Klassen definiert, um die zwei Fahrzeuge `myVolo` und `myBianchi` zu konstruieren:

- `class Fahrzeug`
  Sie ist die eigentliche „fachliche“ Klasse und beschreibt ein Fahrzeug durch die drei Attribute (→Abschnitt 3.1 auf Seite 34):
  - Geschwindigkeit
  - Fahrtrichtung
  - Eigentümer
- `class FahrzeugProg`
  Sie enthält die Methode `main()`. Diese Klasse entspricht dem „Steuerungsblock“ eines (üblichen) imperativen Programmes.
- `class Fahrt`
  Sie dient zum Erzeugen eines „Hilfsobjektes“. Ein solches Objekt wird einerseits als Argument und andererseits als Rückgabewert der Methode `wohin()` genutzt. Damit wird gezeigt wie mehrere Einzelwerte zusammegefaßt von einer Methode zurück gegeben werden können.

Diese drei Klassen befinden sich in einer Quellcodedatei. Diese Datei muß den Namen `FahrzeugProg.java` haben, da die Klasse `FahrzeugProg` die

`main`-Methode[2] enthält. Beim Compilieren der Quellcodedatei `FahrzeugProg.java` entstehen drei Dateien:

- `Fahrt.class`
- `Fahrzeug.class`
- `FahrzeugProg.class`

Um eine Ordnung in die vielen Klassen zu bekommen, werden mehrere Klassen zu einem Paket (*package*) zusammengefaßt. Hier wurde als Paketname:

`de.FHNON.Fahrzeug`

Paket

gewählt. Die drei Klassendateien sind relativ zum Pfad, der in `CLASSPATH` angegeben ist, zu speichern. Die Abbildung 5.3 auf Seite 66 zeigt das Klassendiagramm für die Java-Applikation `FahrzeugProg.java`.
Der Aufruf der Klasse `FahrzeugProg` erfolgt mit dem vorangestellten Paketnamen, das heißt in unserem Beispiel auf einer AIX-Plattform:

```
cl3:/home/bonin:>echo $CLASSPATH %$
/u/bonin/myjava:/usr/lpp/J1.1.6/lib/classes.zip:/usr/lpp/J1.1.6/lib:.
cl3:/home/bonin:>javac ./myjava/de/FHNON/Fahrzeug/FahrzeugProg.java
cl3:/home/bonin:>java de.FHNON.Fahrzeug.FahrzeugProg
```

[Hinweis: Zu beachten ist, daß die Namensteile der Angabe für den Java-Interpreter mit einem Punkt (`.`) getrennt werden, während die Angabe für den Java-Compiler die übliche Pfadtrennzeichen aufweist, also mit Slash (`/`) in der UNIX-Welt bwz. Backlslash (\) in der Windows-Welt.]

```
 1  /**
 2    Beispiel für Objekterzeugung (Konstruktor)
 3      und destruktive Methoden ("call by reference")
 4    Grundidee: [RRZN97] S. 40ff (jedoch stark modifiziert und korrigiert)
 5    Bonin 26-Oct-1997
 6      Update 27-Oct-1997; 13-Jul-1998
 7  */
 8  /*
 9    Package:  Fahrt.class, Fahrzeug.class, FahrzeugProg.class
10    Die Klassen werden im Unterverzeichnis
11    relativ zum CLASSPATH erzeugt.
12    Achtung:
13    Weltweite Eindeutigkeit des Paketnamens ist sicherzustellen!
14  */
15  package de.FHNON.Fahrzeug;
16  /*
17    Die Klasse Fahrt als "Hilfsobjekt".
18    Sie ist definiert, um für eine Methode den
19    Rückgabewert einer solchen Instanz zu haben.
20  */
```

[2] [Hinweis: Jede Klasse kann — zum Beispiel zum Testen — eine Methode `main()` enthalten. Entscheidend ist die Methode `main()` der Klasse, die vom Java-Interpreter aufgerufen wird.]

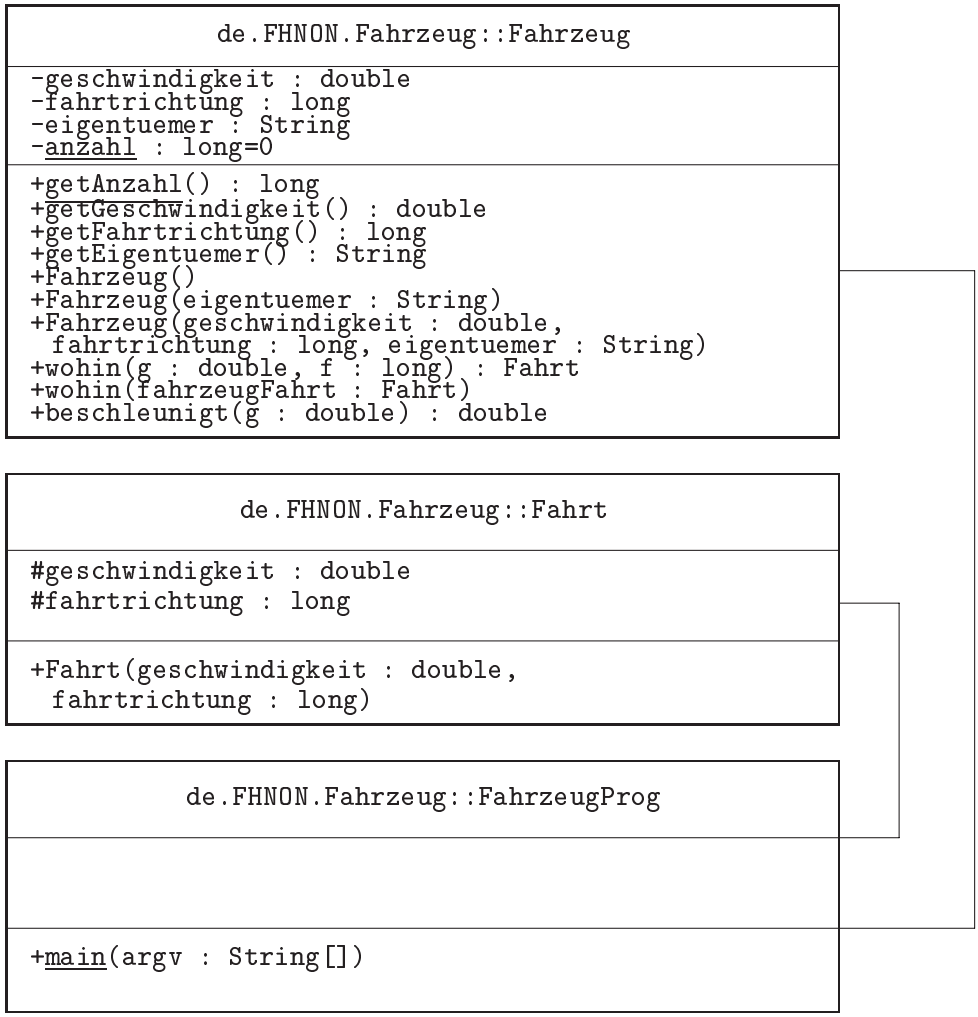


Abbildung 5.3: Klassendiagramm für `FahrzeugProg.java`

```
21  class Fahrt {
22    protected double geschwindigkeit;
23    protected long   fahrtrichtung;
24    // Konstruktor eines Objektes Fahrt mit zwei Parametern,
25    // die den gleichen Namen wie die Datenkomponenten (Slots) haben.
26    // this-Referenzierung daher erforderlich
27    public Fahrt(double geschwindigkeit, long fahrtrichtung) {
28      this.geschwindigkeit = geschwindigkeit;
29      this.fahrtrichtung   = fahrtrichtung;
30    }
31  }
32  /*
33
34    Klasse Fahrzeug als "fachliches Objekt"
35
36  */
37  class Fahrzeug {
38    private double geschwindigkeit;
39    private long   fahrtrichtung;
40    private String eigentuemer;
41
42    // Klassenvariable
43    private static long anzahl;
44    // Static Initialization Block zum
45    // setzten der Anfangszuweisungen (kein Rückgabewert!)
46    static {
47      anzahl = 0;
48    }
49    //Selektor (Zugriffsmethode auf den Klassen-Slot)
50    public static long getAnzahl() {
51      return anzahl;
52    }
53  /*
54    Datenkapselung in Klasse Fahrzeug, daher
55    Selektoren als Methoden definiert.
56  */
57    public double getGeschwindigkeit() {
58      return geschwindigkeit;
59    }
60    public long getFahrtrichtung() {
61      return fahrtrichtung;
62    }
63    public String getEigentuemer() {
64      return eigentuemer;
65    }
66  /*
67   Konstruktoren für "fachliche Objekte"
68  */
69    // Standard-Konstruktor (ohne Parameter)
70    public Fahrzeug() {
71      anzahl = anzahl + 1;
72    }
73    // Konstruktor mit einem Parameter eigentuemer nutzt
74    // Standard-Konstruktor um Instanz zu gründen.
```

```
 75    public Fahrzeug(String eigentuemer) {
 76      this();
 77      this.eigentuemer = eigentuemer;
 78    }
 79    // Konstruktor nutzt Konstruktor-Hierarchie
 80    // Fahrzeug(eigentuemer) --> Fahrzeug()
 81    public Fahrzeug(double geschwindigkeit,
 82      long fahrtrichtung, String eigentuemer) {
 83      this(eigentuemer);
 84      this.geschwindigkeit = geschwindigkeit;
 85      this.fahrtrichtung   = fahrtrichtung;
 86    }
 87  /*
 88   Beispielmethoden
 89  */
 90    // Gibt neue Instanz von Fahrt mit gewünschter
 91    // Geschwindigkeit  und Fahrtrichtung zurück
 92    // Bei einem einfachen Datentyp wird der Wert übergeben.
 93    // Entspricht "call by value".
 94    public Fahrt wohin(double g, long f) {
 95      Fahrt fahrzeugFahrt = new Fahrt(g, f);
 96      return fahrzeugFahrt;
 97    }
 98    // Modifiziert eine übergebene Instanz mit
 99    // den Werten des Objektes auf das die
100    // Methode angewendet wurde.
101    // Ein Objekt und ein Array werden als Referenz übergeben.
102    // Entspricht "call by reference"
103    public void wohin(Fahrt fahrzeugFahrt) {
104      fahrzeugFahrt.geschwindigkeit = geschwindigkeit;
105      fahrzeugFahrt.fahrtrichtung   = fahrtrichtung;
106    }
107    // Erhöht die Geschwindigkeit um einen festen Wert
108    public double beschleunigt(double g) {
109      geschwindigkeit = geschwindigkeit + g;
110      return geschwindigkeit;
111    }
112  }
113  /*
114
115    Klasse FahrzeugProg
116    beschreibt primär  die Kommuninkation (Steuerblock)
117    (entspricht dem Hauptprogramm
118    in der imperativen Programmierung)
119
120  */
121  public class FahrzeugProg {
122    public static void main(String argv[]) {
123      Fahrzeug myVolvo = new Fahrzeug(193.5, 360, "Emma");
124      Fahrt himmelfahrt = new Fahrt(77, 90);
125      myVolvo.wohin(himmelfahrt);
126      myVolvo.beschleunigt(6.4);
127
128      Fahrzeug myBianchi = new Fahrzeug("Gustav");
```

```
129       int g = 44;
130       int f = 180;
131       Fahrt osterfahrt = myBianchi.wohin(g, f);
132       myBianchi.beschleunigt(39);
133
134       System.out.println("Fahrzeuganzahl: " + Fahrzeug.getAnzahl());
135       System.out.println("Richtung der Himmelfahrt: " +
136         himmelfahrt.fahrtrichtung);
137       System.out.println("Richtung der Osterfahrt: " +
138         osterfahrt.fahrtrichtung);
139       System.out.println("myVolvo: " +
140         myVolvo.getGeschwindigkeit() + " | " +
141         myVolvo.getFahrtrichtung() + " | " +
142         myVolvo.getEigentuemer());
143       System.out.println("myBianchi: " +
144         myBianchi.getGeschwindigkeit() + " | " +
145         myBianchi.getFahrtrichtung() + " | " +
146         myBianchi.getEigentuemer());
147
148       // Referenz auf ein Objekt freigaben
149       myVolvo = null;
150       // Garbage Collector aufrufen
151       System.gc();
152     }
153   }
154   // End of File cl3:/u/bonin/myjava/DE/FHNON/Fahrzeug/FahrzeugProg.java
```

Im obigen Beispiel `FahrzeugProg` wird das Fahrzeug `MyVolvo` mit:

- der `geschwindigkeit` = 193.5,
- der `fahrtrichtung` = 360 und
- dem `eigentuemer` = `"Emma"`

angelegt. Außerdem wird die Fahrt `Himmelfahrt` mit:

- der `geschwindigkeit` = 77 und
- der `fahrtrichtung` = 90

angelegt. Durch die Anwendung der „destruktiven“ Methode:

```
wohin(himmelfahrt)
```

auf das Objekt `myVolvo` werden die Werte des Objektes `himmelfahrt` geändert, obwohl `himmelfahrt` nur als Argument übergeben wurde. Da `himmelfahrt` ein *Reference Type* (→Tabelle 5.4 auf Seite 84) ist, wird das Objekt als Referenz und nicht als Wert übergeben. Zur Erzeugung des Objektes `osterfahrt` wird
`myBianchi.wohin(g,f)`
ausgeführt. Bei dieser Methode sind die Parameter vom Typ `double` und `long`, das heißt einfache Datentypen (*Primitive Type*). Die Argumente werden daher als Werte und nicht als Referenzen übergeben.

In der Klasse `Fahrzeug` sind zwei namensgleiche Methoden `wohin()` definiert. Die Entscheidung der jeweils anzuwendenden Methode `wohin()` erfolgt über den Vergleich der Anzahl und des Typs der Parameter mit den jeweils angegebenen Argumenten.

[Hinweis: Einfache Datentypen (→Tabelle 5.5 auf Seite 85) werden stets durch ihren Wert übergenen. Bei einem „zusammengesetzten“ Objekt und einem Array (*ReferenceType* →Tabelle 5.4 auf Seite 84) wird die Referenz auf das Objekt übergeben.]

Compilation und Ausführung von `FahrzeugProg`:

```
cl3:/home/bonin:>java -fullversion
java full version "JDK 1.1.6 IBM build a116-19980529" (JIT: jitc)
cl3:/home/bonin:>echo $CLASSPATH
/u/bonin/myjava:/usr/lpp/J1.1.6/lib/classes.zip:/usr/lpp/J1.1.6/lib:.
cl3:/home/bonin:>javac ./myjava/de/FHNON/Fahrzeug/FahrzeugProg.java
cl3:/home/bonin:>java de.FHNON.Fahrzeug.FahrzeugProg
Fahrzeuganzahl: 2
Richtung der Himmelfahrt: 360
Richtung der Osterfahrt: 180
myVolvo: 199.9 | 360 | Emma
myBianchi: 39.0 | 0 | Gustav
cl3:/home/bonin:>ls -l *.class
*.class not found
cl3:/home/bonin:>cd ./myjava/de/FHNON/Fahrzeug
cl3:/home/bonin/myjava/de/FHNON/Fahrzeug:>ls -l *.class
-rw-r--r--   1 bonin    staff        431 Jul 13 09:54 Fahrt.class
-rw-r--r--   1 bonin    staff       1315 Jul 13 09:54 Fahrzeug.class
-rw-r--r--   1 bonin    staff       1683 Jul 13 09:54 FahrzeugProg.class
cl3:/home/bonin/myjava/de/FHNON/Fahrzeug:>
```

5.1.4 Kostprobe `MyNetProg.java` — Internetzugriff

In diesem Beispiel wird ein dynamisches[3] Dokument vom WWW-Server:

URL

`cl3.fbw.fh-luneburg.de`

mit der Portnummer `6667` gelesen. Verwendet wird dabei die Klasse `URL`[4] des Standardpaketes:

java.net

`java.net`.

Mit dem Konstruktor wird das Objekt `homePage` an einen konkreten URL gebunden. Die Verbindung zum WWW-Server wird im Objekt `homePageConnection` vom Typ `URLConnection` abgebildet. Mit der Methode `openConnection()` wird die Verbindung aktiviert und mit der Mehtode `get.InputStream` wird

[3] „Dynamisches“ Dokument über CGI-Skript angestoßen (CGI ≡ Common Gateway Interface)

[4] URL ≡ Uniform Resource Locator

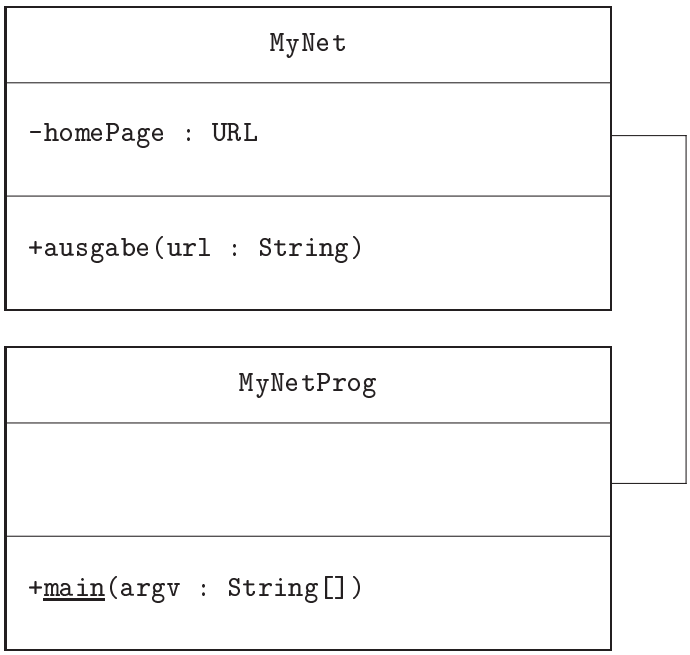


Abbildung 5.4: Klassendiagramm für `MyNetProg.java`

der HTTP[5]-Datenstrom gelesen. Die Abbildung 5.4 auf Seite 71 zeigt das Klassendiagramm der Applikation `MyNetProg.java`.

```
 1  /**
 2    Beispiel für HTTP-Kommunikation im Internet
 3    Zugriff auf den WWW-Server
 4     "cl3.fbw.fh-lueneburg.de"
 5       mit der Portnummer 6667 und
 6       Aktivierung eines CGI-Skriptes.
 7    Grundidee: [RRZN97] S. 111ff
 8    Bonin 28-Oct-1997
 9      Update 13-Jul-1998
10  */
11  package de.FHNON.MyNet;
12  import java.net.*;
13  import java.io.*;
14  class MyNet {
15    private URL homePage;
16    public void ausgabe(String url) {
17      try {
18        URL homePage = new URL(url);
19        System.out.println("URL: " + homePage + "\n" +
20          "WWW-Server: " + homePage.getHost());
21
22        // Verbindung zum Dokument
23        URLConnection homePageConnection = homePage.openConnection();
24
25        // gelieferten HTTP-Datenstrom in ein DataInputStream wandeln
```

[5] HTTP ≡ HyperText Transfer Protocol

```
26        DataInputStream in =
27          new DataInputStream(homePageConnection.getInputStream());
28
29        // Ausgeben zeilenweise
30        for (int i = 0; true; i++) {
31          String line = in.readLine();
32          if (line == null)
33            break;
34          System.out.println(i + ": " + line);
35        }
36      }
37      catch(IOException e1) { // ... hier nicht abgefangen
38      }
39    }
40  }
41  public class MyNetProg {
42    public static void main(String argv[]) {
43      MyNet netObject = new MyNet();
44      // fest verdrahtete URL-Angabe
45      netObject.ausgabe(
46        "http://cl3.fbw.fh-lueneburg.de:6667/cgi-bin/spass.ksh"
47        );
48    }
49  }
50  // End of File cl3:/u/bonin/myjava/DE/FHNON/MyNet/MyNetProg.java
```

Compilation und Ausführung von `MyNetProg`:

Hier wird der Java-Compiler mit der Option `deprecation` („Mißbilligung“) aufgerufen. Damit wird der Text der Warnungen ausgegeben. Die Methode `readLine()` der Klasse `DataInputStream` liest Zeichen aus einem *Stream* bis sie auf ein *Newline*-Zeichen, ein *Carriage Return* oder auf beide hintereinander trifft.

```
cl3:/home/bonin:>javac -deprecation ./myjava/de/FHNON/MyNet/MyNetProg.java
./myjava/de/FHNON/MyNet/MyNetProg.java:38:
Note: The method java.lang.String readLine() in class
java.io.DataInputStream has been deprecated.
        String line = in.readLine();
                                  ^
Note: ./myjava/de/FHNON/MyNet/MyNetProg.java uses a deprecated API.
Please consult the documentation for a better alternative.
2 warnings
cl3:/home/bonin:>java de.FHNON.MyNet.MyNetProg
URL: http://cl3.fbw.fh-lueneburg.de:6667/cgi-bin/spass.ksh
WWW-Server: cl3.fbw.fh-lueneburg.de
0: <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN//">
1: <HTML>
2: <HEAD><TITLE>fac(n)</TITLE> </HEAD>
3: <BODY>
4: <P>Das Datum meines Rechners IBM RS/6000
5: (cl3, IP=193.174.33.106) ist: <B>
6: Tue Oct 28 13:52:17 NFT 1997
```

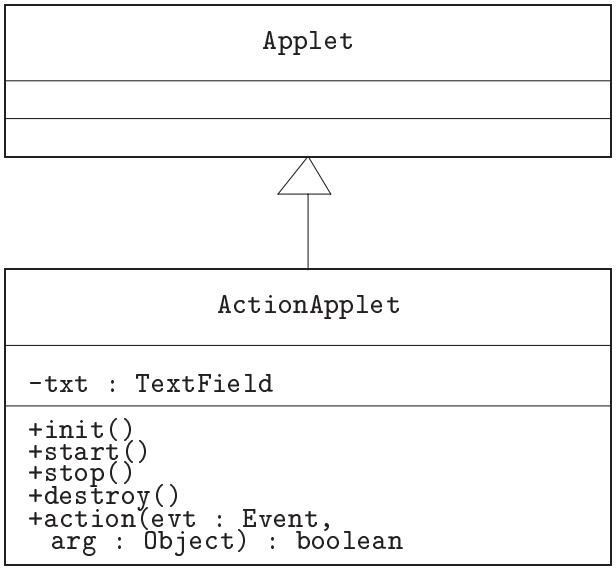


Abbildung 5.5: Klassendiagramm für `ActionApplet.java`

```
7: </B></P>
8: <P> Berechnung der Fakult&auml;t von n=100</P>
9: <P><IMG SRC="/bonin/facbild.gif"
10: ALT="[Mathematische Notation fac(n)]"
11: ALIGN="BOTTOM"></P>
12: <FONT SIZE=+1><PRE>
  .
  .
  .
95: <HR>
96: Datenautobahn
97: <A HREF="/bonin/autobahn.html">
98: <IMG SRC="/bonin/links.gif"
99: ALT="[Finger nach links]" ALIGN="MIDDLE"></A>
100: </BODY>
101: </HTML>
cl3:/home/bonin:>
```

5.1.5 Kostprobe `ActionApplet.java` — GUI

Die Abbildung 5.5 auf Seite 73 zeigt das Klassendiagramm für `ActionApplet.java`.

```
1  /**
2    ActionApplet.class zeichnet geschachtelte Panels
3    und akzeptiert in der Mitte einen Text, der
4    in der Statuszeile des Browsers und auf
5    der Console (System.out) ausgegebnen wird.
6    Grundidee: [HSS96]
7
8    Bonin 22-Jan-1997
9      Update 13-Jul-1998
```

```
10  */
11
12  import java.awt.*;
13  import java.applet.*;
14
15  public class ActionApplet extends Applet {
16    // Textfeld zur Erfassung eines Textes, der
17    // dann in der Statuszeile des Browsers angezeigt wird.
18    private TextField txt;
19    public void init() {
20      // Initialisierung mit Font Helvetica, Bold, 24
21      Font pFont = new Font("Helvetica",Font.BOLD,24);
22      setLayout(new BorderLayout());
23      setBackground(Color.white);
24      setForeground(Color.green);
25
26      add("North", new Button("Norden"));
27      add("South", new Button("Süden"));
28
29      // Erzeugt ein Panel p0 mit Struktur im Zentrum
30      Panel p0 = new Panel();
31      p0.setBackground(Color.red);
32      p0.setForeground(Color.white);
33      p0.setLayout(new BorderLayout());
34      add("Center",p0);
35      p0.add("North", new Button("Oben"));
36      p0.add("South", new Button("Unten"));
37
38      // Erzeugt ein Panel p1 mit Struktur im Zentrum von p0
39      Panel p1 = new Panel();
40      p1.setBackground(Color.blue);
41      p1.setForeground(Color.yellow);
42      p1.setLayout(new BorderLayout());
43      add("Center",p1);
44      p1.add("North", new Button("Hamburg"));
45      p1.add("South", new Button("Hannover"));
46
47      // Setzt das Textfeld in die Mitte des inneren Panels
48      txt = new TextField(10);
49      txt.setFont(pFont);
50      p1.add("Center", txt);
51
52      p1.add("East", new Button("Lüneburg"));
53      p1.add("West", new Button("Salzhausen"));
54
55      p0.add("West", new Button("Links"));
56      p0.add("East", new Button("Rechts"));
57
58      add("West", new Button("Westen"));
59      add("East", new Button("Osten"));
60     }
61    public void start() {
62    }
63    public void stop() {
```

```
64    }
65    public void destroy() {
66    }
67    public boolean action(Event evt, Object arg) {
68      System.out.println(
69        ((Button)evt.target).getLabel()
70        + ": " + txt.getText());
71      showStatus(
72        ((Button)evt.target).getLabel()
73        + ": " + txt.getText());
74        return true;
75    }
76  }
77  // End of File cl3:/u/bonin/mywww/ActionApplet/ActionApplet.java
```

Laden und Ausführen des Applets `ActionApplet`:
Das obige Beispiel wird vom `appletviewer` des *Java Development Kit* auf einer NT-Plattform angezeigt. Dieses Applet ist in der HTML-Datei `PruefeApplet.html` (→Abschnitt 5.2 auf Seite 75) eingebunden. Diese HTML-Datei wird über den WWW-Server wie folgt erreicht:

```
appletviewer http://cl3.fbw.fh-lueneburg.de:6667/bonin/PrufeApplet.html
```

Die Abbildung 5.6 auf Seite 76 zeigt einen Ausschnitt nach diesem Aufruf. Jedes Applet, das in dieser HTML-Datei genannt ist, wird in einem eigenen Fenster dargestellt.

5.2 Applet-Einbindung in ein HTML-Dokument

5.2.1 Applet ⇔ Applikaton

Java unterscheidet zwei Ausführungstypen:[6]

- Applikation
  Als Applikation bezeichnet man ein „eigenständiges“ Java-Programm, das als „Startmethode“ `main()` enthält und **direkt**, also nicht über einen WWW-Browser oder den `appletviewer`, sondern mittels Aufruf von `java` ausgeführt wird, dem *Java-Byte-Code-Interpreter* des *Java Development Kit* (JDK).

Applikation

- Applet
  Als Applet bezeichnet man ein Java-Programm, das über eine HTML-Seite aufgerufen wird und über diese auch die Aufrufargumente erhält. Das Laden und die Ausführung des Applets steuert ein Java-fähiger World-Wide-Web-Browser wie zum Beispiel `Netscape Communicator` oder der `appletviewer` aus dem JDK. Ein Applet, ursprünglich als kleines Java-Programm gedacht, kann durchaus sehr umfangreich sein. Ein Applet ist eine Unterklasse der Klasse `java.applet.Applet`.

Applet

[6] auch als Programm(formen) bezeichnet

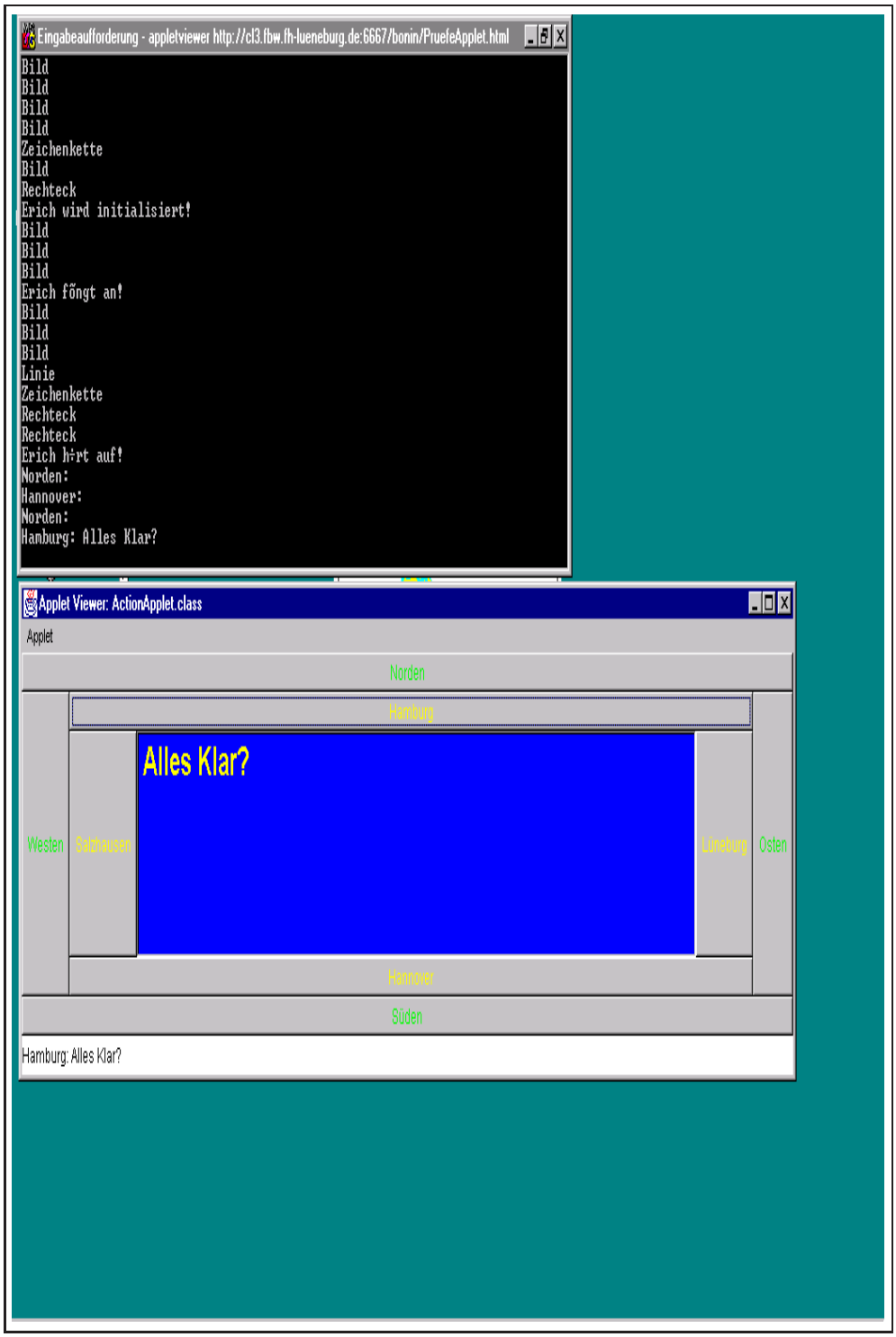


Abbildung 5.6: Ausführung des Applets `ActionApplet.class`

Syntax eines HTML-Konstruktes

$$\overbrace{\underbrace{< bezeichner \ \{attribut[= value]\} >}_{Anfangsmarke} \ [text] \mid [Konstrukt] \ \underbrace{[< /bezeichner >]}_{Endmarke}}^{Konstrukt}$$

zwingend erforderlich — manchmal optional

Legende:
Notation gemäß Backus-Naur-Form (BNF)

`[...]`	≡ das Eingeklammerte kann entfallen (optional)
{...}	≡ das Eingeklammerte kann einmal, mehrmals oder gar nicht vorkommen
a | *b*	≡ Alternative, entweder *a* oder it
bezeichner	≡ aus Buchstabe(n), manchmal gefolgt von Integerzahl
attribut	≡ aus ASCII-Zeichen
value	≡ aus ASCII-Zeichen
text	≡ aus ASCII-Zeichen

Beispiel: Geschachtelte Konstrukte „`<TITLE>` in `<HEAD>`“
```
<HEAD><TITLE>Softwarekonstruktion</TITLE></HEAD>
```

Beispiel: Sequenz der Konstrukte: *Link*, Neue Zeile, Strich („`<A><BR><HR>`“)
```
<A HREF="/w3/media.html">Multimedia</A><BR><HR>
```
Näheres →[Bonin96]

Tabelle 5.1: Syntax eines HTML-Konstruktes

5.2.2 HTML-Marken: <OBJECT> und <APPLET>

Ein HTML-Konstrukt ist definiert:

- durch eine (Anfangs-)Marke, notiert als „`<bezeichner>`“ und gegebenenfalls
- durch eine Endemarke, notiert als „`</bezeichner>`“.

Einige Konstrukte haben keine Endemarke oder diese Marke kann entfallen. Zusätzlich zum Bezeichner können Marken Attribute (Argumente) haben, denen über ein Gleichheitszeichen ein Wert zugewiesen werden kann. Der Wert ist in doppelte Hochkommata einzuschließen.[7] Bei der Angabe eines Wertes wird Groß/Klein-Schreibung unterschieden. Die Syntax für ein Konstrukt in HTML verdeutlicht Tabelle 5.1 auf Seite 77. Sie ist dort rekursiv notiert, da Konstrukte geschachtelt werden können. Ein Konstrukt kann eine Sequenz von weiteren Konstrukten einschließen.

Ein Applet wird in HTML 4.0 mit Hilfe des `<OBJECT>`-Konstruktes eingebunden. In vorhergehenden HTML-Versionen dient dazu das `<APPLET>`-Konstrukt. Das `<OBJECT>`-Konstrukt ermöglicht außer Applets, die auch in

[7] Viele *Browser* benötigen jedoch die doppelten Hochkommata nicht (mehr).

anderen Sprachen als Java geschrieben sein können, quasi beliebige Multimedia-Objekte wie zum Beispiel Videos und Bilder syntaktisch einheitlich einzubauen. Der `OBJECT`-Begriff beschreibt hier alle Dinge, die man in ein HTML-Dokument plazieren möchte. Andere Bezeichnungen in diesem Zusammenhang sind *Media Handlers* und *Plug Ins*.

Ältere HTML-Versionen (< 4.0): `<APPLET>...</APPLET>`

```
<BODY>
<P>
<APPLET codebase="myPath"
  code="myApplet.class"
  width="300" height="500"
  alt="Mein Logo als tolles Applet myApplet">
  Java myApplet.class: Mein Logo
</APPLET>
</P>
<BODY>
```

HTML-Version 4.0: `<OBJECT>...</OBJECT>`

```
<BODY>
<P>
<OBJECT codetype="application/java"
  codebase="myPath"
  classid="java:myApplet.class"
  width="300" height="500"
  alt="Mein Logo als tolles Applet myApplet">
  Java myApplet.class: Mein Logo
</OBJECT>
</P>
</BODY>
```

Syntaktisch gleichartig ist zum Beispiel der Einbau eines Bildes:

```
<BODY>
<P>Hier ist mein tolles Hundefoto:
<OBJECT data="http://www.irgendwo.de/Foo/Edi.png"
  type="image/png">
  Mein tolles Hundefoto.
</OBJECT>
</P>
</BODY>
```

Die Tabelle 5.2 auf Seite 79 beschreibt Attribute des `<OBJECT>`-Konstruktes.[8] Das Attribut `align` sollte jedoch entsprechend dem CSS-Konzept (→Abschnitt 8.2

[8] Umfassende, vollständige Beschreibung des `<OBJECT>`-Konstruktes siehe HTML4.0-Spezifikation, zum Beispiel:
`http://www.w3.org/TR/REC-html40`

Attribute im <OBJECT>-Konstrukt	
classid=uri	Ort der Objekt-Implementation als URI-Angabe.
codebase=uri	Basispfad für die Auflösung der relativen URI-Angaben in `classid`, `data` und `archive`. Bei keiner Angabe wird als Ersatzwert die URI-Basis des aktuellen Dokumentes verwendet.
codetype=content-t.	Gibt den erwarteten Datentyp an, wenn ein Objekt, das durch `classid` spezifiziert wurde, geladen wird. Es ist ein optionales Attribut, das ein Laden eines nicht unterstützten Datentyps vermeiden soll.
data=uri	Gibt den Ort der Objektdaten an, zum Beispiel für Bilddaten.
type=content-t.	Spezifiziert den Datentyp für `data`. Es ist ein optionales Attribut, das ein Laden eines nicht unterstützten Datentyps vermeiden soll.
archive=uri list	Eine URI-Liste für die Angabe von Archiven, die relevante Quellen für das Objekt enthalten (Trennzeichen in der Liste ist das Leerzeichen.)
standby=text	Text, der angezeigt wird während das Objekt geladen wird.
width=length	Angabe für den *User Agent* seinen *Default*-Wert mit der angegebenen Länge (in Pixel) zu überschreiben.
height=length	analog zu `width`
align=position	Gibt die Objekt-Position bezogen auf den (Kon)Text an. • `left` Linke Rand ist Objekt-Position • `right` Rechte Rand ist Objekt-Position • `bottom` Unterkante fluchtet mit aktueller Basisline • `middle` Mitte fluchtet mit aktueller Basisline • `top` Oberkante fluchtet mit aktueller Basisline

Legende:
uri ≡ *Universal Resource Identifier* (≈ allgemeine Dokumentenadresses)

Tabelle 5.2: Applet-Einbindung: Einige Attribute des <OBJECT>-Konstruktes

auf Seite 196) verwendet werden, das heißt nicht direkt im <OBJECT>-Konstrukt sondern im <STYLE>-Konstrukt.
Hinweis: Nicht jeder marktübliche Browser unterstützt alle Attribute (korrekt).

5.2.3 Beispiel PruefeApplet.html

Das Dokument PruefeApplet.html umfaßt zwei Applets. Das Applet ActionApplet.class ist über das <OBJECT>-Konstrukt eingebunden. Für das Applet ImageLoopItem.class wird das „überholte“[9] <APPLET>-Konstrukt genutzt. Das <STYLE>-Konstrukt spezifiziert nur das Layout, das heißt hier Farben, Fonts und die Textausrichtung. Seine Wirkungweise wird später eingehend erläutert (→Abschnitt 8.2 auf Seite 196).

Das Beispieldokument PruefeApplet.html weist die übliche Grundstruktur zum Einbinden eines Applets auf:

```
<!DOCTYPE ...>
<HTML>
<HEAD>
<TITLE>...</TITLE>
</HEAD>
<BODY>
...
<OBJECT ...>
  ...
</OBJECT>
...
</BODY>
</HTML>
```

Die zusätzlichen Angaben wie zum Beispiel die <META>-Konstrukte beschreiben das HTML-Dokument als Ganzes und betreffen hier nicht direkt das <OBJECT>-Konstrukt.

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
 2    "http://www.w3c.org/TR/REC-html40/strict.dtd">
 3  <!-- Testbett fuer Applets                    -->
 4  <!-- Bonin  13-Jan-1997                       -->
 5  <!-- Update 13-Jul-1998                       -->
 6  <HTML>
 7  <HEAD>
 8  <BASE href="http://cl3.fbw.fh-lueneburg.de:6667/bonin/">
 9  <TITLE>JAVA-COACH's Applet-Testbett</TITLE>
10  <META http-equiv="Last-Modified"
11        content="13-Jul-98 13:00:00 GMT">
12  <META http-equiv="Expires"
13        content="01-Jan-99 00:00:00 GMT">
14  <META name="KEYWORDS"
```

[9] Das <APPLET>-Konstrukt wird im HTML4.0-Standard mißbillig (≡ *a deprecated element*).

```
15        content="Applet-Beispiele, Arbeiten von H. Bonin">
16  <META name="DESCRIPTION"
17        content="Bonin, JAVA-COACH">
18  <LINK rev=owns
19        title="Hinrich E.G. Bonin"
20        href="mailto:hinrich-bonin@fbw.fh-lueneburg.de">
21  <STYLE type="text/css">
22    P.links {
23      text-align: left;
24    }
25    EM {
26      font-style: italic;
27      color: #FFFFFF;
28      background-color: #000000;
29    }
30    BR.freiLassen {
31      clear: left;
32    }
33    BODY {
34      color: black;
35      background-color: #00FFFF
36    }
37  </STYLE>
38  </HEAD>
39  <BODY>
40  <H1>JAVA-COACH's Applet-Testbett
41  </H1>
42  <P class="links">
43  <OBJECT
44    codetype="application/java"
45    codebase="ActionApplet"
46    classid="java:ActionApplet.class"
47    width="350" height="125"
48    standby="Hier kommt gleich was zum Clicken!">
49    Java Applet ActionApplet.class
50  </OBJECT>
51  Das nebenstehende Beispiel ist mit
52  <EM>ActionApplet.class</EM> konstruiert
53  (<A HREF="/bonin/ActionApplet/ActionApplet.java">
54  Java Ouellcode</A>)
55  <BR class="freiLassen"></P>
56  <P class="links">
57  <APPLET code="ImageLoopItem.class"
58    width="179" height="175" align="left"
59    alt="Der schnelle Powerman!">
60    <PARAM name="NIMGS" value="5">
61    <PARAM name="IMG"   value="IrinaRad">
62    <PARAM name="PAUSE" value="0">
63    Java Applet ImageLoopItem:
64    Schneller Powerman auf der Radstrecke!
65  </APPLET>
66  Das nebenstehende Bewegtbild ist mit
67  <EM>ImageLoopItem.class</EM> konstruiert.
68  <BR>
```

```
69  Ein schneller Powerman auf der gef&uuml;rchteten
70  Strecke in Zofingen (Schweiz)! Man beachte das
71  Dreispeichenvorderrad zur Verbesserung der Aerodynamik.
72  Der Duathlet h&auml;tte nat&uuml;rlich  viel flacher
73  liegen m&uuml;ssen, um einen minimalen Luftwiderstand
74  zu erreichen. Jedoch h&auml;tte er wohl dann kaum
75  gut durchatmen k&ouml;nnen.
76  <BR class="freiLassen"></P>
77  <P>
78  Copyright Bonin 16-Jan-97 all rights reserved
79  </P>
80  <ADDRESS>
81  <A HREF="mailto:hinrich-bonin@fbw.fh-lueneburg.de"
82    >hinrich-bonin@fbw.fh-lueneburg.de</A>
83  </ADDRESS>
84  </BODY>
85  <!-- Ende der Datei /u/bonin/mywww/PruefeApplet.html -->
86  </HTML>
```

5.3 Syntax & Semantik & Pragmatik

Die Tabelle 5.3 auf Seite 83 nennt die in Java reservierten Wörter. Diese können nicht als Namen für eine eigene Klasse, Schnittstelle, Variable oder Methode verwendet werden. Darüber hinaus sollten die folgenden Methodennamen aus der `Object`-Klasse nicht benutzt werden, es sei denn man möchte die `Object`-Methode überschreiben.

Reservierte Methodennamen:
`clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString` und `wait`.

In Java werden „zusammengesetzte“ Objekte (*ReferenzType*) von einfachen Datentypen (*PrimitiveType*) unterschieden. Die Tabelle 5.4 auf Seite 84 zeigt anhand einer rekursiven Beschreibung die unterschiedlichen Typen.

5.3.1 Attribute für Klasse, Schnittstelle, Variable und Methode

Bei der Deklaration einer Klasse, Schnittstelle, Variable oder Methode können Attribute, sogenannte Modifikatoren, angegeben werden. Neben den Modifikatoren für die Zugriffsrechte[10] (Sichtbarkeit) sind es folgende Attribute:

- `static`
  - Variable: Die Variable ist eine Klassenvariable (hat „Speicherplatz“ nur in der Klasse) und wird durch den Klassennamen angesprochen.

[10] ⇒ Tabelle 5.6 auf Seite 86

Wort	Stichworthafte Erläuterung	Wort	Stichworthafte Erläuterung
`abstract`	Deklaration von Klasse / Meth.	`boolean`	einf. Datentyp: `true` / `false`
`break`	Kontrollkonstrukt; terminiert	`byte`	einfacher Datentyp (8 Bit Zahl)
`byvalue`	— nicht genutzt —	`case`	Kontrollkonstrukt; mit `switch`
`cast`	— nicht genutzt —	`catch`	Kontrollkonstrukt; mit `try`
`char`	einfacher Datentyp	`class`	Deklariert eine Klasse
`const`	— nicht genutzt —	`continue`	Kontrollkonstrukt
`default`	Kontrollkonstrukt; mit `switch`	`do`	Kontrollkonstrukt; mit `while`
`double`	einf. Datentyp (64 Bit Fließkom.)	`else`	Kontrollkonstrukt; mit `if`
`extends`	Superklassenangabe	`false`	`boolean`-Wert
`final`	Keine Subklasse; unübersch. M.	`finally`	Kontrollkon.; mit `try/catch`
`float`	einf. Datentyp (32 Bit Fließkom.)	`for`	Kontrollkonstrukt; Iteration
`future`	— nicht genutzt —	`generic`	— nicht genutzt —
`goto`	— nicht genutzt —	`if`	Kontrollkonstrukt; Alternative
`implements`	Implementiert Schnittstelle	`import`	Namensabkürzungen
`inner`	— nicht genutzt —	`instanceof`	Prüft Instanz einer Klasse
`int`	einfacher Datentyp (32 Bit Zahl)	`interface`	Deklariert Schnittstelle
`long`	einfacher Datentyp (64 Bit Zahl)	`native`	„Andere“(C-)Implemtation
`new`	Erzeugt neues Objekt / Array	`null`	„kein Objekt“-Referenze
`operator`	— nicht genutzt —	`outer`	— nicht genutzt —
`package`	Paket; erste Anweisung	`private`	Zugriffsrecht
`protected`	Zugriffsrecht	`public`	Zugriffsrecht
`rest`	— nicht genutzt —	`return`	Kontrollkonstruk; Rückgabe
`short`	einfacher Datenyp (16 Bit Zahl)	`static`	Klassen-Variable / -Methode
`super`	Zugriff auf Super-Klasse	`switch`	Kontrollk.; Fallunterscheidung
`synchronized`	Sperrmechanismus	`this`	Verweist auf „dieses Objekt“
`throw`	Erzeugt Ausnahmeobjekt	`throws`	Deklariert Ausnahmezustände
`transient`	Kein persistenter Objektteil	`true`	`boolean`-Wert
`try`	Kontrollk.; mit `catch/finally`	`var`	— nicht genutzt —
`void`	Kein Rückgabewert	`volatile`	Asynchrone Wertänderung
`while`	Kontrollkonstrukt		

Tabelle 5.3: Reservierte Java-Bezeichner (Wörter)

  - Methode: Die Methode ist eine Klassenmethode und ist implizit `final`. Sie wird durch den Klassennamen aufgerufen.

- `abstract`
  - Klasse: Die Klasse kann keine Instanz haben. Sie kann nicht-implementierte Methode beinhalten.
  - Schnittstelle: Eine Schnittstelle ist stets `abstract`. Die Angabe kann daher entfallen.
  - Methode: Die einschließende Klasse muß ebenfalls `abstract` sein. Es wird kein Methodenkörper angegeben. Dieser wird von der Subklasse bereitgestellt. Der Methodenangabe (Signatur[11]) folgt ein Semikolon.

- `final`
  - Klasse: Von der Klasse kann keine Subklasse gebildet werden.
  - Methode: Die Methode ist nicht überschreibbar. Der Compiler kann daher effizienteren Code erzeugen.
  - Variable: Der Wert der Variablen ist nicht änderbar. Schon der Compiler kann Ausdrücke auswerten und muß dies nicht dem Interpreter überlassen.

- `native`
  - Methode: Die Methode ist in einer maschinennahen, plattformabhängigen Art implementiert, zum Beispiel in C. Es wird kein Methodenbody angegeben und die Nennung wird mit einem Semikolon abgeschlossen.

[11] Signatur besteht aus Namen, Anzahl und Typ der Parameter

Type ≡
    PrimitiveType | *ReferenceType*

PrimitiveType ≡
    NumericType |
    `boolean`

NumericType ≡
    IntegralType | *FloatingPointType*

IntegralType ≡
    `byte` | `short` | `int` | `long` | `char`

FloatingPointType ≡
    `float` | `double`

ReferenceType ≡
    ClassOrInterfaceType | *ArrayType*

ClassOrInterfaceType ≡
    Name

ArrayType≡
    PrimitiveType[] | *Name*[] | *ArrayType*[]

Legende:

`terminal`	≡ definiert als
nonterminal	≡ ist noch weiter zu definieren
a | b	≡ a oder b

Weitere Definitionen →[Arnold/Gosling96]

Tabelle 5.4: Datentypbeschreibung anhand von Produktionsregeln

Typ	Enthält	Standard	Größe in Bit	Werte Minimal	Maximal
`boolean`	`true` oder `false`	`false`	1	—	—
`char`	Unicode Zeichen	`\u0000`	16	`\u0000`	`\uFFFF`
`byte`	Integer mit Vorzeichen	0	8	-128	127
`short`	Integer mit Vorzeichen	0	16	-32768	32767
`int`	Integer mit Vorzeichen	0	32	-2147483648	2147483647
`long`	Integer mit Vorzeichen	0	64	-9223372036854775808	9223372036854775807
`float`	Fließkomma IEEE 754	0.0	32	±3.40282347E+38	±1.40239846E-45
`double`	Fließkomma IEEE 754	0.0	32	±1.79769313486231570E+308	±4.94065645841246544E-324

Quelle: [Flanagan96] Seite 183
Hinweis: Die Größe in Bit ist nicht implementationsabhängig!

Tabelle 5.5: Javas einfache Datentypen

- `synchronized`
  - Methode: Die Methode nimmt eine oder mehrere Veränderungen der Klasse oder einer Instanz vor, wobei sichergestellt ist, das zwei Threads eine Änderung nicht gleichzeitig vornehmen können. Dazu wird die Instanz für den Mehrfachzugriff gesperrt. Bei einer `static`-Methode wird die Klasse gesperrt.
- `transient`
  - Variable: Damit wird gekennzeichnet, daß die Variable kein Teil des persistenten Zustandes des Objektes ist.
- `volatile`
  - Variable: Die Variable ändert sich asynchron (zum Beispiel kann sie ein Hardwareregister eines Peripheriegerätes sein). Ihr Wert sollte daher vom Compiler nicht in Registern gespeichert werden.

5.3.2 Erreichbarkeit bei Klasse, Schnittstelle, Variable und Methode

Zur Einschränkung oder Erweiterung des Zugriffs gegenüber keiner Angabe (*default*) dienen die foldenden Modifikatoren:

`public`, `private` und `protected`

Die Tabelle 5.6 auf Seite 86 beschreibt ihre Anwendungsmöglichkeiten. Einige Zugriffssituationen zeigt Tabelle 5.7 auf Seite 87.

Lfd. Nr.	**Modifikator**	**Erläuterung der Erreichbarkeit**
1	keine Angabe (default)	Wenn kein Zugriffsrecht (Modifikator) angegeben wird, ist eine Klasse, Schnittstelle, Variable oder Methode nur innerhalb des gleichen Paketes zugreifbar.
2	`public`	Eine Klasse oder Schnittstelle ist überall zugreifbar. Eine Methode oder Variable ist überall in der Klasse zugreifbar.
3	`private`	Eine Variable oder Methode ist nur in ihrer eigenen Klasse zugreifbar. • Eine Klasse kann nicht als `private` gekennzeichnet werden.
4	`protected`	Eine Variable oder Methode ist im gesamten Paket ihrer Klasse zugreifbar und in allen Subklassen der Klasse. Eine Subklasse in einem anderen Paket als ihre Superklasse kann auf die `protected`-Einträge zugreifen, die ihre Instanzen geerbt haben, aber sie kann nicht die Einträge in den (direkten) Instanzen der Superklasse erreichen. • Eine Klasse kann nicht als `protected` gekennzeichnet werden.
5	`private protected`	Eine Variable oder Methode ist nur innerhalb ihrer eigenen Klasse und den zugehörigen Subklassen zugreifbar. Eine Subklasse kann auf alle „`private protected`“-Einträge zugreifen, die ihre Instanzen geerbt haben, aber sie kann nicht die Einträge in den (direkten) Instanzen der Superklasse erreichen. • Eine Klasse kann nicht als `private protected` gekennzeichnet werden.

Hinweis: Die Standardzugreifbarkeit (keine Angabe) ist **strikter** als die `protected`-Angabe! (Quelle [Flanagan96] Seite 188)

Tabelle 5.6: Java-Zugriffsrechte für Klasse, Schnittstelle, Variable und Methode

Lfd. Nr.	Situation	Gekennzeichnet mit:
1	Erreichbar für Subklassen eines **anderen** Paketes!	`public`

2	Erreichbar für Subklassen des **gleichen** Paketes!	— (default)
		`public`
		`protected`

3	Erreichbar für **Nicht**-Subklassen eines **anderen** Paketes!	`public`

4	Erreichbar für **Nicht**-Subklassen des **gleichen** Paketes!	— (default)
		`public`
		`protected`

5	Geerbt von Subklassen in einem **anderen** Paket!	`public`
		`protected`
		`private protected`

6	Geerbt von Subklassen im **gleichen** Paket!	— (default)
		`public`
		`protected`
		`private protected`

Tabelle 5.7: Zugriffssituationen

5.4 Übungen

5.4.1 Shell-Kommando „`echo`“ programmieren

Abbildung als Java-Applikation

Schreiben Sie eine Java-Applikation, die das übliche `echo`-Kommando einer UNIX- und/oder MS-DOS-Shell abbildet. (Idee entnommen [Flanagan96])

Unterschiede der Java-Lösung zum Shell-Kommando

Nennen Sie Unterschiede Ihrer Lösung zum `echo`-Kommando einer üblichen Shell.

5.4.2 Applikation `Kontrolle.java`

```
 1  /**
 2   Kleine Kostprobe fuer:
 3   data types and control structures
 4   @author Bonin 02-Apr-1998
 5   @version 1.0
 6  */
 7  public class Kontrolle {
 8    public static void main(String argv[])
 9      {
10       int i = argv.length, j, k;
11       double m = 0.3141592653589791;
12       int n = (int) m;                        // Casting
13
14       String  p  = "java";
15
```

```
16       boolean  wichtig,  vielleicht = true, klar;
17
18       i += p.length();
19       j = i++;
20       i--;
21       k = ++i;
22       --i;
23
24       wichtig = (i == j && vielleicht);
25       wichtig = (wichtig != vielleicht);
26       klar = (i <= k ) || (  wichtig == true);
27
28       System.out.println("Werte: " +
29                          "\ni = " + i +
30                          "\nj = " + j +
31                          "\nk = " + k +
32                          "\nm = " + m +
33                          "\nn = " + n +
34                          "\np = " + p +
35                          "\nvielleicht = " + !vielleicht +
36                          "\nwichtig = " + !wichtig +
37                          "\nklar = " + klar );
38     }
39  }
40  //--- cl3:/u/bonin/myjava/Kontrolle.java
```

Geben Sie bei dem folgenden Aufruf das Ergebnis an:

```
>java Kontrolle ist besser!
```

5.4.3 Applikation `Iteration.java`

```
 1  /**
 2   Kleine Kostprobe fuer:
 3   Iterationen
 4   @author Bonin 02-Apr-1998
 5   @version 1.0
 6  */
 7  public class Iteration {
 8    public static void main(String argv[])  {
 9      boolean in = true;
10      int zaehler = 0, index;
11      String spruchTabelle[]  = new String[argv.length];
12
13      String meinSpruch = "";
14      String wortZumSuchen = "C++", wortZumErsetzen = "Java";
15
16      spruchTabelle[0] = "Maximum";
17      spruchTabelle[1] = "UML";
18      spruchTabelle[2] = "&";
19      spruchTabelle[3] = "C++";
20      spruchTabelle[4] = "in der";
21      spruchTabelle[5] = "Anwendungsentwicklung";
22
```

```
23      int  anzahlPositionen = spruchTabelle.length;
24
25      while (zaehler  <  anzahlPositionen)   {
26        if (spruchTabelle[zaehler].equals(wortZumSuchen ))  {
27          spruchTabelle[zaehler] = wortZumErsetzen;
28          break;
29        }
30        zaehler++;
31      }
32
33      zaehler = -1;
34      do {
35        zaehler++;
36        meinSpruch += spruchTabelle[zaehler];
37        meinSpruch += " ";
38       }
39       while (zaehler <  (anzahlPositionen - 1));
40       System.out.println(meinSpruch +
41         "\nDies sind " + meinSpruch.length() +
42         " Zeichen!");
43      }
44  }
45  // cl3:/u/bonin/myjava/Iteration.java
```

Geben Sie bei den folgenden Aufrufen das Ergebnisse an:

```
>java Iteration 1 2 3 4 5 6 7
>java Iteration 1 2
```

5.4.4 Applikation `LinieProg.java`

Die Abbildung 5.7 auf Seite 90 zeigt das Klassendiagramm der Applikation `LinieProg.java`.

```
 1  /**
 2   Kleine Kostprobe fuer:
 3   this und super-Konstruktor
 4   @author Bonin 06-Apr-1998
 5     update: 15-Jul-1998
 6   @version 1.0
 7  */
 8  class Linie {
 9    private int startX, startY, endeX, endeY;
10
11    public int getStartX() {return startX;}
12    public int getStartY() {return startY;}
13    public int getEndeX()  {return endeX;}
14    public int getEndeY()  {return startY;}
15
16    public Linie(int startX, int startY,
17                 int endeX,  int endeY) {
18      super();
19      this.startX = startX;
20      this.startY = startY;
```

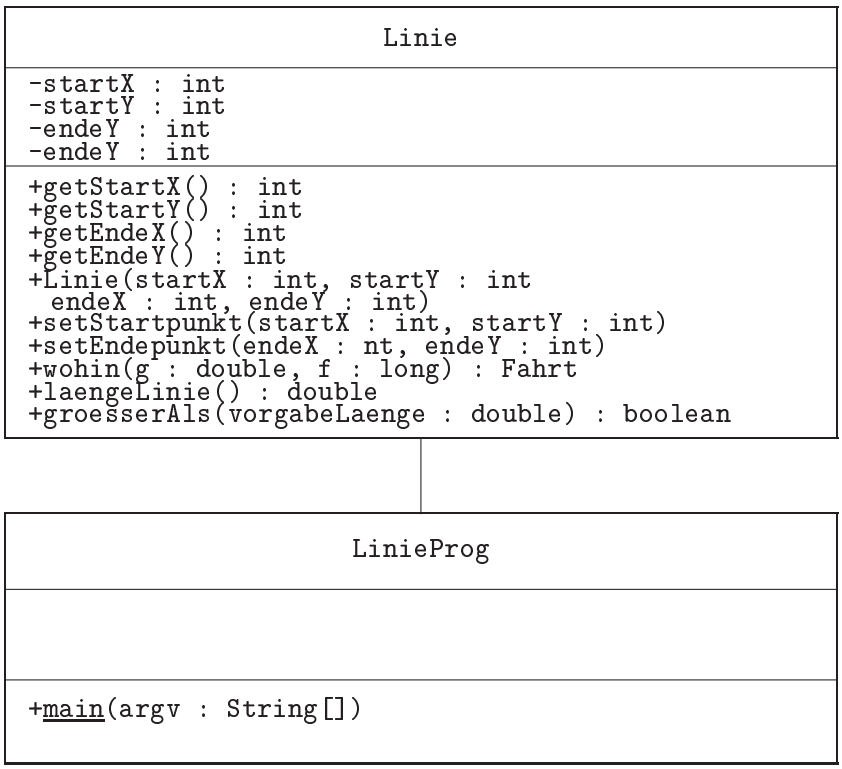


Abbildung 5.7: Klassendiagramm für `LinieProg.java`

```
21      this.endeX  = endeX;
22      this.endeY  = endeY;
23    }
24
25    public void setStartpunkt(int startX, int startY) {
26      this.startX = startX;
27      this.startY = startY;
28    }
29    public void setEndepunkt(int endeX, int endeY) {
30      this.endeX = endeX;
31      this.endeY = endeY;
32    }
33
34    public double laengeLinie() {
35      return (Math.sqrt(Math.pow((double) endeX - startX, 2.0) +
36                        Math.pow((double) endeY - startY, 2.0)));
37    }
38
39    public boolean groesserAls(double vorgabeLaenge) {
40      return (this.laengeLinie() > vorgabeLaenge);
41    }
42  }
43
44  public class LinieProg {
45
46    public static void main(String argv[]) {
```

```
47      Linie l1 = new Linie(10,10,13,14);
48
49      Linie l2 = l1;
50      l2.setStartpunkt(0,0);
51      l2.setEndepunkt(3,4);
52
53      double vorgabeLaenge = 0.25000e2;
54
55      System.out.println("" +
56        l1.getStartX() + l1.getStartY() +
57        l1.getEndeX()  + l1.getEndeY() + "\n" +
58        l1.laengeLinie() + "\n" +
59        l1.groesserAls(vorgabeLaenge)
60        );
61      }
62  }
63  // cl3:/u/bonin/myjava/LinieProg.java
```

Geben Sie bei dem folgenden Aufruf das Ergebnisse an:

```
>java LinieProg
```

5.4.5 Applikation `Inheritance.java`

Die Abbildung 5.8 auf Seite 92 zeigt das Klassendiagramm der Applikation `Inheritance.java`.

```
 1  /**
 2   Kleine Kostprobe fuer:
 3   Vererbung --- abstract class
 4   @author Bonin 05-Apr-1998, 15-Jul-1998
 5   @version 1.1
 6  */
 7  abstract class Foo {
 8    protected int n = 3;
 9    protected static int k = 3;
10
11    abstract int eins();
12    int getN() {return n;}
13    static int getK() {return k;}
14  }
15
16  abstract class Bar extends Foo {
17
18    protected int n = super.n + 7;
19    private static StringBuffer myName;
20    abstract int zwei();
21
22    int getN() {return n;}
23    int getFooN() {return super.getN();}
24    static int getK() {return  2 * k ;}
25    static void setMyName(StringBuffer name) {
26      myName = name;
27    }
```

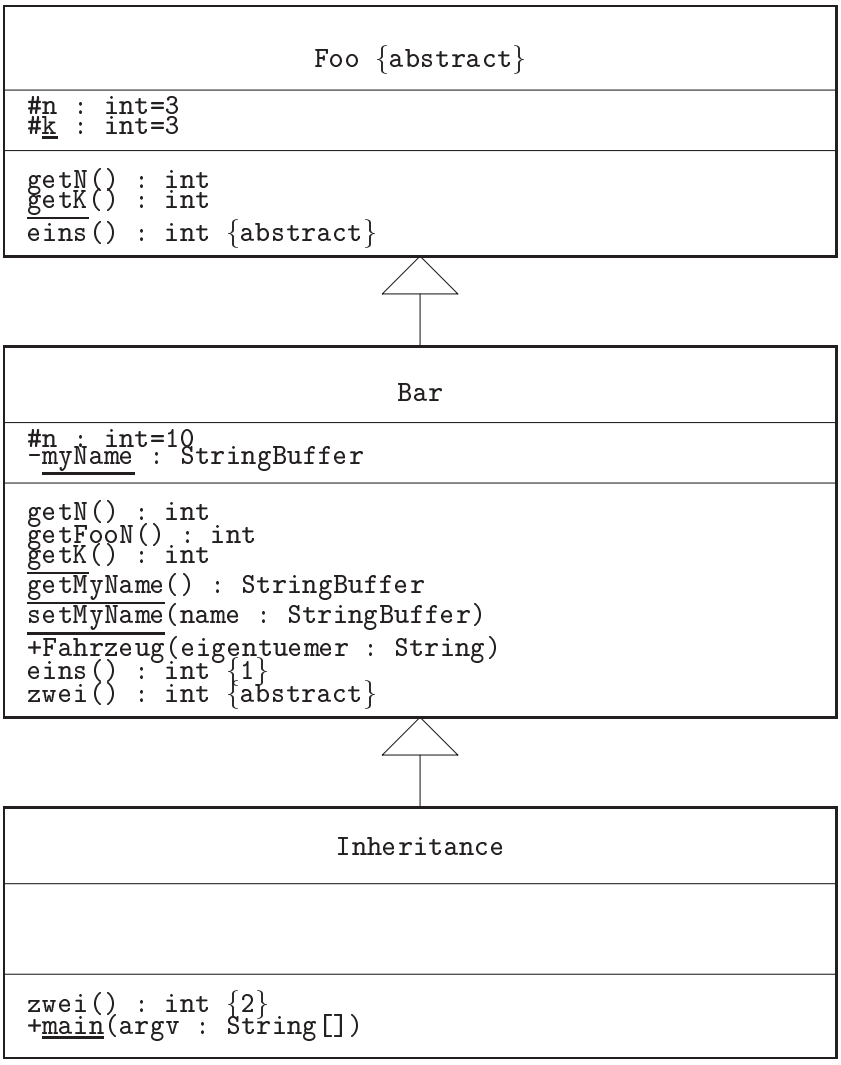


Abbildung 5.8: Klassendiagramm für `Inheritance.java`

```
28    static StringBuffer getMyName() {
29      return myName;
30    }
31    int  eins() {return 1;}
32  }
33
34  public class Inheritance extends Bar {
35    int zwei() {return 2;}
36
37    public static void main(String argv[]) {
38      Inheritance m = new Inheritance();
39
40      Bar.setMyName(new StringBuffer("Otto AG"));
41      Bar.getMyName().setCharAt(3,'i');
42      System.out.println(Bar.getMyName());
43
44      System.out.println("Fall I  : " +
45        (m.eins() + m.zwei() + m.getFooN() + Foo.getK()));
46      System.out.println("Fall II : " +
47         m.eins() + m.zwei() + m.getN() + Bar.getK());
48      }
49  }
50  // cl3:/u/bonin/myjava/Inheritance.java
```

Geben Sie bei dem folgenden Aufruf das Ergebnisse an:

```
>java Inheritance
```

5.4.6 Applikation `TableProg.java`

Die Abbildung 5.9 auf Seite 94 zeigt das Klassendiagramm der Applikation `TableProg.java`.

```
 1  /**
 2   Kleine Kostprobe fuer
 3   einen Fehler der nicht von javac erkannt wird
 4   Idee entnommen aus Adam Freeman / Darrel Ince;
 5    activeJava, 1996, p.105. (Quellcode stark modifiziert.)
 6   @author Bonin 13-Apr-1998
 7   @version 1.0
 8  */
 9  final class LookupTable {
10    private int size;
11    private String holder[];
12
13    LookupTable() {
14      this(100);
15    }
16    LookupTable(int size) {
17      this();
18      this.size = size;
19      holder = new String[size];
20    }
21    public int getSize() {return size;}
```

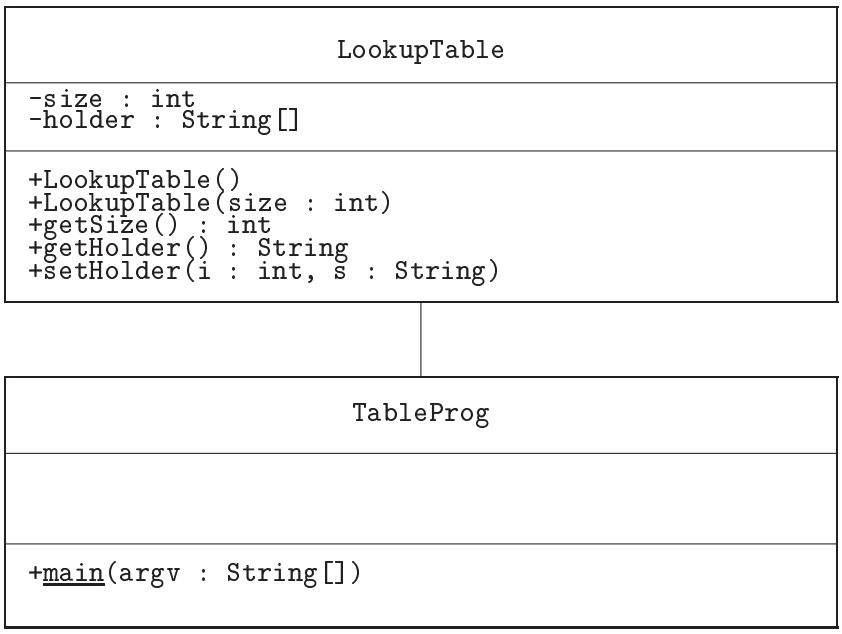


Abbildung 5.9: Klassendiagramm für `TableProg.java`

```
22    public String getHolder(int i) {
23      return holder[i];
24    }
25    public void setHolder(int i, String s) {
26      holder[i] = s;
27    }
28  }
29  public class TableProg {
30    public static void main(String argv[]) {
31    LookupTable myTable = new LookupTable();
32    myTable.setHolder(99, "Alles richtig, oder was?");
33    System.out.println("Tabelle mit " +
34      myTable.getSize() + " erzeugt!");
35    System.out.println(myTable.getHolder(99));
36   }
37  }
38  // cl3:/u/bonin/myjava/TableProg.java
```

Geben Sie bei dem folgenden Aufruf das Ergebnisse an:

```
>java TableProg
```

Kapitel 6

Java-Konstruktionen (Analyse und Synthese)

Primitive Bausteine werden zusammengefügt zu komplexeren Konstruktionen. Dabei dreht sich alles um das Wechselspiel zwischen Aufteilen in mehrere kleinere Objekte und dem Zusammenfassen in größere Objekte. Das Konstruieren ist ein verwobener Prozeß von Analyse- und Synthese-Aktionen. Im Mittelpunkt stehen die konstruktiven („handwerklichen“) Aspekte des Programmierens und weniger die ästhetischen, wie sie etwa im Leitmotto „The Art of Programming“ zum Ausdruck kommen.

Trainingsplan

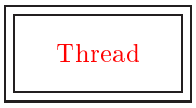
Das Kapitel „Java-Konstruktionen“ erläutert:

- die nebenläufige Ausführung von Teilprozessen (*Multithreading*),
  ↪ Seite 96 ...
- die Behandlung von Ereignissen (Delegationsmodell),
  ↪ Seite 101 ...
- die Realisierung von persistenten Objekten,
  ↪ Seite 110 ...
- das Schachteln von Klassen ineinander (*Inner Classes*),
  ↪ Seite 116 ...
- die Möglichkeit auf die innere Struktur einer Klasse zuzugreifen (*Reflection*),
  ↪ Seite 132 ...
- das Konzept Java-Bausteine zu verteilen und an neue Bedingungen anzupassen (*Java Beans*),
  ↪ Seite 139 ...

- das Arbeiten mit einem objekt-orientierten Datenbanksystem am Beispiel von POET und
↪ Seite 142 ...
- das Zusammenarbeiten verteilter Objekte mit Hilfe von *Remote Method Invocation* (RMI).
↪ Seite 154 ...

6.1 Nebenläufigkeit (*Multithreading*)

Ein *Thread* („Faden“) ist ein ablauffähiges Codestück, daß nebenläufig zu anderen Codestücken ausgeführt wird. Dabei bedeutet Nebenläufigkeit ein unabhängiges, quasi zeitgleiches (paralleles) Ablaufen der einzelnen *Threads*. Ein solches *Multithreading* basiert auf den folgenden beiden Objekten:



- Objekt der `class Thread`
  Ein Objekt dieser Klasse dient zur Steuerung, also beispielsweise zum Starten und Beenden des *Thread*.

- Objekt einer Klasse vom `interface Runnable`
  Ein Objekt einer Klasse, die das Interface `Runnable` implementiert, bildet das nebenläufig abarbeitbare Objekt. Ein solches Objekt `myThread` wird folgendermaßen erzeugt und gestartet:

```
Thread myThread = new Thread(myRunnable).start;
```

  Die Klasse der Instanz `myRunnable` muß die Methode `run()` implementieren, beispielsweise wie folgt:

```
public class MyBesteIdee implements Runnable {
  public void run() {
    // tue was
  }
}
...
MyBesteIdee myRunnable = new MyBesteIdee();
...
```

Die Methode `stop()` könnte innerhalb und außerhalb des `Runnable`-Objektes (hier `myRunnable`) appliziert werden. Startbar ist der Thread natürlich nur außerhalb, das heißt, die Methode `start()` kann nur außerhalb des `Runnable`-Objektes appliziert werden. Innerhalb des `Runnable`-Objektes kann das zugehörige `Thread`-Objekt mit Hilfe der Klassenmethode `currentTread()` festgestellt werden.

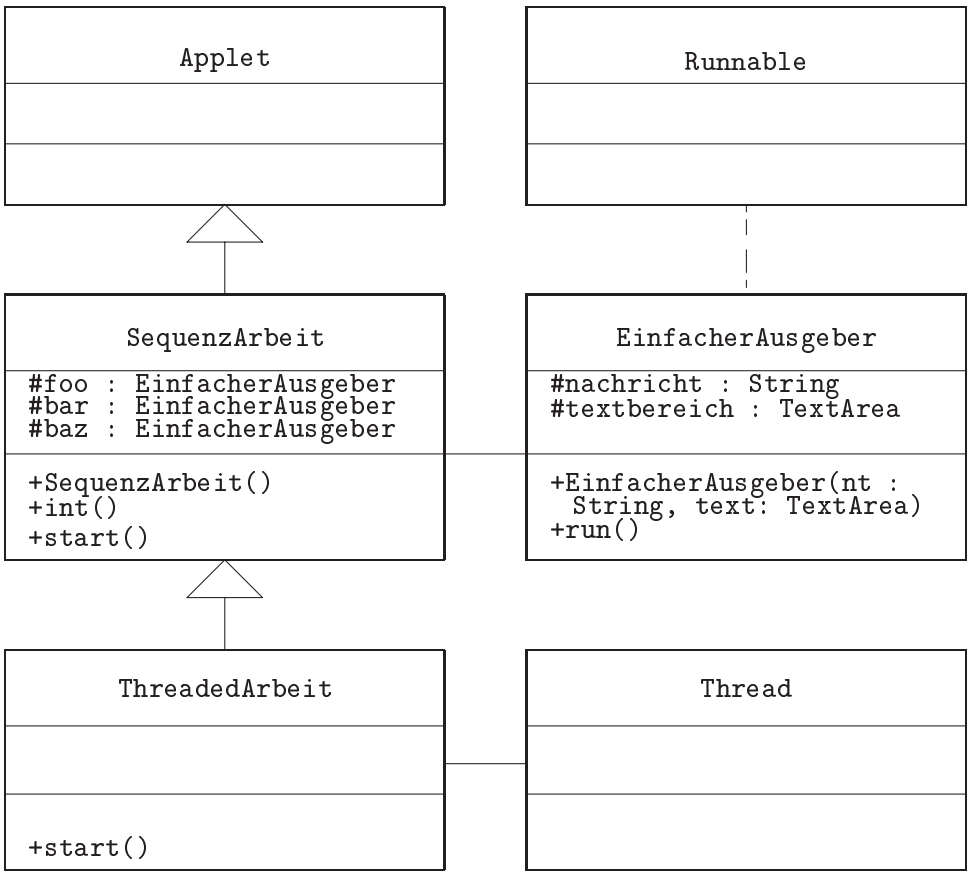


Abbildung 6.1: Klassendiagramm für das Multithreading-Beispiel „Textausgabe“

Das folgende Beispielapplet zeigt ein Textfeld in das drei Zeilen geschrieben werden. Zunächst werden diese drei Zeilen hintereinander erstellt und dann der Reihe nach, also sequentiell, ausgegeben (→Seite 98). Danach erfolgt die Ausgabe mit Hilfe von drei *Threads*, für jede Zeile ein eigener *Thread* (→Seite 99). In beiden Fällen wird dazu die Klasse `EinfacherAusgeber` benutzt (→Seite 98). Dieses einfache Beispiel verdeutlich, daß eine sequentiell konzipierte Lösung durchaus noch in eine nebenläufige verwandelt werden kann. Die Abbildung 6.1 auf Seite 97 zeigt das Klassendiagramm für diese Multithreading-Beispiel.

Teil des HTML-Dokumentes für das Applet `SequenzArbeit.class`

```
1  <OBJECT
2     codetype="application/java"
3     classid="java:SequenzArbeit.class"
4     name="SeqArbeit"
5     width="500"
6     height="600"
7     alt="Kleiner Scherz &uuml;ber SAP.">
```

```
 8     Java SequenzArbeit.class
 9  </OBJECT>
```

SequenzArbeit.java

```
 1  import java.applet.*;
 2  import java.awt.*;
 3  /**
 4    Sequentielles Abarbeiten in einem Applet!
 5    @author Hinrich E. G. Bonin
 6    @version 1.0 13-Jan-1998
 7          update 16-Jul-1998
 8    Idee/Code ähnlich: Doug Lea;
 9     Concurrent Programming in Java, ISBN 0-201-63455-4
10  */
11  public class SequenzArbeit extends Applet {
12    protected TextArea text;
13    protected EinfacherAusgeber foo;
14    protected EinfacherAusgeber bar;
15    protected EinfacherAusgeber baz;
16
17    public SequenzArbeit() {
18      // Textbereich von 5 Zeile mit 20 Spalten
19      text = new TextArea(5,20);
20      foo  = new EinfacherAusgeber("SAP ist ...\n", text);
21      bar  = new EinfacherAusgeber("SAP go go ...\n", text);
22      baz  = new EinfacherAusgeber("Tschüß ...\n", text);
23    }
24    public void init() {
25      add(text);
26    }
27    public void start() {
28      foo.run();
29      bar.run();
30      baz.run();
31    }
32  }
33  // End of File: cl3:/u/bonin/mywww/JavaSAP/SequenzArbeit.java
```

EinfacherAusgeber.java

```
 1  import java.awt.*;
 2  /**
 3    Einfache Ausgabe in einem Fenster!
 4    @author Hinrich E. G. Bonin
 5    @version 1.0 13-Jan-98
 6    Idee/Code ähnlich: Doug Lea;
 7      Concurrent Programming in Java, ISBN 0-201-63455-4
 8  */
 9  public class EinfacherAusgeber implements Runnable {
10    // Variable für die auszugebende Nachricht
```

```
11    protected String nachricht;
12    // Variable für den Textbereich in den ausgegeben wird
13    protected TextArea textbereich;
14
15    public EinfacherAusgeber(String nt, TextArea text) {
16      nachricht    = nt;
17      textbereich  = text;
18    }
19    public void run() {
20      textbereich.appendText(nachricht);
21    }
22  }
23  // End of File: cl3:/u/bonin/mywww/JavaSAP/EinfacherAusgeber.java
```

Teil des HTML-Dokumentes für das Applet `ThreadedArbeit.class`

```
1  <OBJECT
2     codetype="application/java"
3     classid="java:ThreadedArbeit.class"
4     name="ThreadedArbeit"
5     width="500"
6     height="600"
7     alt="Kleiner Scherz &uuml;ber SAP.">
8     Java ThreadedArbeit.class
9  </OBJECT>
```

`ThreadedArbeit.java`

```
1  import java.applet.*;
2  import java.awt.*;
3  /**
4    Nebenläufiges Abarbeiten in einem Applet!
5    @author Hinrich E. G. Bonin
6    @version 1.0 13-Jan-98
7    Idee/Code ähnlich: Doug Lea;
8      Concurrent Programming in Java ISBN 0-201-63455-4
9  */
10 public class ThreadedArbeit extends SequenzArbeit {
11    public void start() {
12     new Thread(foo).start();
13     new Thread(bar).start();
14     new Thread(baz).start();
15   }
16 }
17 // End of File: cl3:/u/bonin/mywww/JavaSAP/ThreadedArbeit.java
```

6.1.1 Unterbrechung (`sleep`)

Bei mehreren *Threads* spielt die Aufteilung der Rechenzeit ein große Rolle. Durch das direkte Setzten von Prioritäten mit der Methode `setPriority()` besteht eine Möglichkeit der Beeinflussung. Trotzdem kann es passieren, daß

ein *Thread* die gesamte Kapazität in Anspruch nimmt und die anderen nicht zum Zuge kommen. Um eine solche Situation zu vermeiden, sollte jeder *Thread* mal unterbrochen werden, damit die anderen eine Chance bekommen. Dazu dient die Klassenmethode `sleep(long zeitMilliSekunden)` der Klasse `Thread`. Beim Aufruf dieser Methode muß die Fehlersituation `InterruptedException` abgefangen werden. Unbedingt eingebaut werden sollte die `sleep`-Methode bei Iterationen und besonders bei Endlosschleifen. Dazu bietet sich beispielsweise folgende Konstruktion an:

```
public class MyBesteIdee implements Runnable {
  public void run() {
    while (true) {
      // tue was
      try {
        Thread.sleep(500);
      }
      catch (InterruptedException e) {
        // Fehler behandeln
      }
    }
  }
}
```

6.1.2 Synchronisation (`wait()`, `notify()`, `synchronized`, `join`)

Das `Thread`-Konzept ist in Java konsequent für alle Objekte verwirklicht. So kennt jedes Objekt, jede Subklasse von `Object`, folgende Steuerungsmethoden:

- `wait()` und `wait(long timeout)`
  Die Methode zum Abwarten, bis der gestartete *Thread* ein `notify()` für dieses Objekt sendet. Dabei gibt ein Argument die maximale Wartezeit in Millisekunden an.

- `notify()` und `notifyAll()`
  Die Methode dient zum Beenden des Wartezustandes. Wenn ein oder mehrere *Threads* mit `wait()` warten, wird danach der Wartezustand beendet. Wartet kein *Thread* ist diese Methode ohne Bedeutung.

Diese Steuerung der *Threads* kann zu Blockaden („*Deadlocks*“) führen, wenn beispielsweise der *Thread* X wartet, das der *Thread* Y ein `notify()` sendet, und der *Thread* Y wartet, daß der *Thread* X ein `notify()` sendet.

Wenn ein `Thread`-Objekt die Ergebnisse, die ein anderer *Thread* X produziert, benötigt, dann muß gewartet werden, bis X fertig ist. Dazu dient die Methode `join()`. Diese Methode aus der Klasse `Thread` sorgt für das Abwarten, bis der gestartete *Thread* X fertig ist. Wenn mehrere laufende *Threads* dasselbe Objekt ändern, kommt es zu Problemen, wenn nur halb ausgeführte Zwischenzustände eines *Threads* vom anderen schon geändert werden (*concurrent update problem*). Um dies zu verhindern, sind

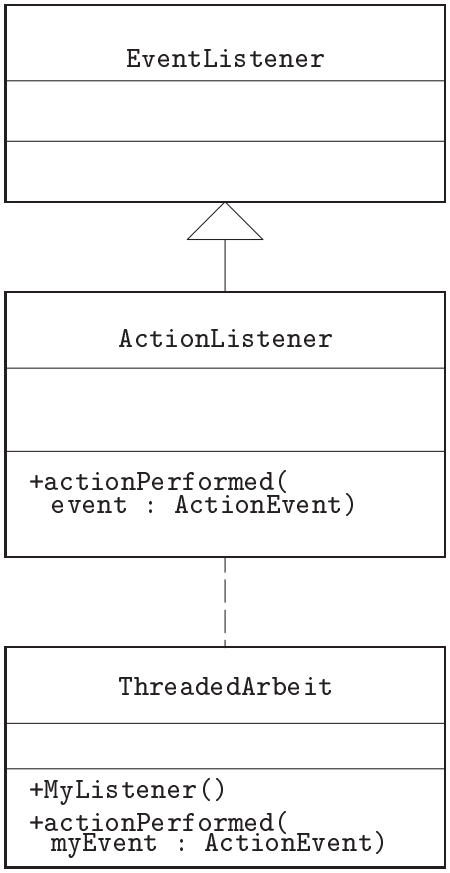


Abbildung 6.2: Java AWT: Konstruktion des Delegationsmodells

alle gefährdeten Aktionen in einem Block zusammenzufassen. Dazu dient das `synchronized`-Konstrukt.

6.2 Ereignisbehandlung (Delegationsmodell)

Ein GUI-System (*Graphical User Interface*)[1] muß Interaktionen mit dem Benutzer steuern. Dazu nutzt es eine Ereigniskontrollschleife (*event loop*) in der festgestellt wird, ob eine betreffende Benutzeraktion eingetreten ist. Im JDK 1.1 wurde diese Ereignisbehandlung neu konzipiert und von Sun als Delegationsmodell bezeichnet. Einem GUI-Objekt kann jetzt ein *event handler* direkt zugeordnet werden. Dieser überwacht das Eintreten eines Ereignisses. Er „horcht“ permanent und appliziert eine fest vorgegebene Methode, wenn das Ereignis eintritt. Er wird daher als Listener bezeichnet. Erfolgt beispielsweise ein Mausclick auf einen Button, dann führt sein zugeordneter `ActionListener` die Methode `actionPerformed()` aus. Der Listener selbst ist ein Interface und spezialisiert die Klasse `EventListener`. Die Abbildung 6.2 auf Seite 101 skizziert als Klassendiagramm dieses Delegationsmodell.

Listener

[1] Andere Beispiele sind Microsoft's Windows und Unix's Motif.

```
public interface ActionListener extends EventListener {
  public abstract void actionPerformed(ActionEvent event}
}
```

Das Delegationsmodell des JDK 1.1 für einen aktionskontrollierten Auslöseknopf (Button) fußt dann beispielsweise auf folgender Konstruktion. Der eigene Listener `MyListener` implementiert das Interface `ActionListener` wie folgt:

```
public class MyListener implements ActionListener {
  public MyListener(...){
  // Konstruktormethode
  // Die drei Punkte stehen fuer das Objekt welches beim
  // Ereigniseintritt modifiziert werden soll. So wird
  // es fuer den Listener erreichbar.
  }
  public void actionPerformed(ActionEvent myEvent) {
    // irgend eine Aktion wie modifiziere ...
    // und/oder zum Beispiel
    System.out.println{Ereignis eingetreten};
  }
}
```

Einer Instanz von `MyButton` wird dann eine Instanz von `MyListener` mit der Methode `addActionLister()` wie folgt zugeordnet:

```
public class MyButton extends Button {
  Button anmelden;
  public MyButton()
    // irgendeinen Button definieren, zum Beipiel
    anmelden = new Button("Anmelden");
  }
}
public class MyApplication extends Frame {
  public MyApplication() {
    MyListener mL = new MyListener(...);
    MyButton   mB = new MyButton();
    mB.addActionListener(mL);
    ...
  }
  public static void main(String argv[]) {
    new MyApplication().show();
  }
}
```

Die Zuordnung einer Instanz von `MyListener` zu einer Instanz von `MyButton` kann auch im Konstruktor `MyButton()` geschehen und zwar wie folgt:

```
public MyButton extends Button implements ActionListener {
  Button anmelden;
```

```
  public MyButton() {
    anmelden = new Button("Anmelden");
    this.addActionListener(this);
  }
  public void actionPerformed(ActionEvent myEvent) {
    System.out.println{Ereignis eingetreten};
  }
}
```

6.2.1 `ActionListener` — Beispiel `SetFarbe.java`

Mit einem Beispiel wird im folgenden das Delegationsmodell verdeutlicht. Es wird angenommen, daß ein Applet-Benutzer die Farbe eines Ausgabetextes direkt verändern möchte. Dabei werden zwei Lösungen angeboten: Im ersten Fall geschieht die Farbwahl durch Eintragung des gewünschten Farbnamens in ein Textfeld (→Seite 105). Im zweiten Fall kann durch einen Doppelklick auf eine Liste mit Farbnamen die gewünschte Farbe ausgewählt werden (→Seite 105). Für beide Fälle wird nur eine Klasse `SetFarbe`, die das Interface `ActionListener` implementiert, benötigt (→Seite 103), denn die zu kontrollierende Benutzeraktion besteht jeweils aus dem Setzen einer Farbe. In der Methode `actionPerformed()` wird der Farbname als ein `String` über die Methode `getActionCommand()` gewonnen. In beiden Fällen liefert diese Methode einen `String`, unabhängig davon, ob die Benutzeraktion durch einen Doppelklick oder über die Entertaste nach Eingabe des Textes erfolgte. Die Abbildung 6.3 auf Seite 104 zeigt das Klassendiagramm für den Fall der textlichen Eingabe der gewünschten Farbe.

`SetFarbe.java`

```
 1  /**
 2   Kleines Beispiel fuer die
 3   ActionListener
 4   Idee aus Glenn Vanderburg, et al.; MAXIMUM JAVA 1.1, pp.230--234
 5   Quellcode modifiziert.
 6
 7   Teil I: Listener SetFarbe
 8   @author Bonin 27-Apr-1998
 9           update 16-Jul-May-1998
10   @version 1.0
11  */
12  import java.awt.event.ActionListener;
13  import java.awt.event.ActionEvent;
14  import java.awt.* ;
15
16  public class SetFarbe implements ActionListener {
17    final String farbname[] = {"rot", "gruen", "blau"};
18    final Color farbe[]      = {Color.red, Color.green, Color.blue};
19    private Label bezeichnung;
```

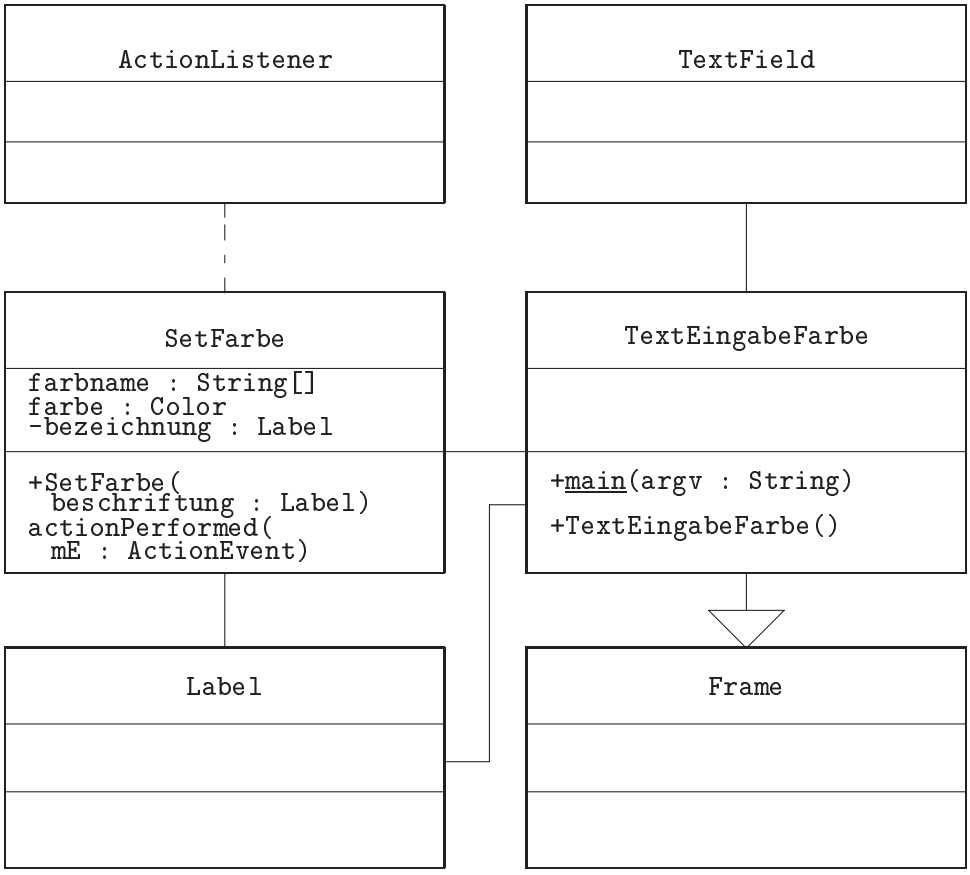


Abbildung 6.3: Klassendiagramm für `TextEingabeFarbe.java`

```
20
21    public SetFarbe(Label beschriftung) {
22      bezeichnung = beschriftung;
23    }
24    public void actionPerformed(ActionEvent myE) {
25      String name = myE.getActionCommand();
26      for (int n = 0; n < farbname.length; n++) {
27        if (farbname[n].equals(name)) {
28          bezeichnung.setForeground(farbe[n]);
29          return;
30        }
31      }
32      System.out.println("Unbekannter Farbwunsch: " + name);
33    }
34  }
35  //End of file cl3:/u/bonin/myjava/SetFarbe.java
36
```

TextEingabeFarbe.java

```
 1  /**
 2   Kleines Beispiel fuer ActionListener
 3   Teil IIa: Farbwahl per Texteingabe
 4   @author Bonin 27-Apr-1998
 5           Update 16-Jul-1998
 6   @version 1.0
 7  */
 8  import java.awt.* ;
 9
10  public class TextEingabeFarbe extends Frame {
11    public static void main (String argv[]) {
12      new TextEingabeFarbe().show();
13    }
14    public TextEingabeFarbe() {
15      Label myL = new Label("Hello World");
16      TextField text = new TextField(20);
17
18      this.add("North", text);
19      this.add("Center", myL);
20      this.pack();
21
22      SetFarbe sf = new SetFarbe(myL);
23      // Aktion ist Enter-Taste druecken
24      text.addActionListener(sf);
25    }
26  }
27  //End of file cl3:/u/bonin/myjava/TextEingabeFarbe.java
28
```

ListWahlFarbe.java

Abbildung 6.4: Ergebnis: `java TextEingabeFarbe`

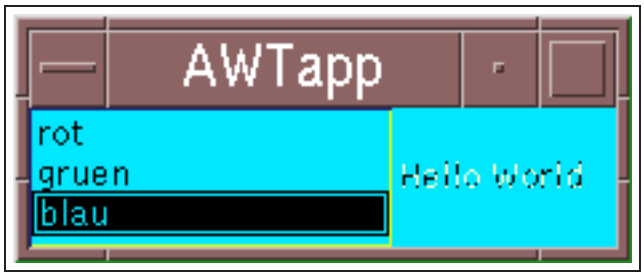


Abbildung 6.5: Ergebnis: `java ListWahlFarbe`

```
1  /**
2   Kleines Beispiel fuer ActionListener
3   Teil IIb: Farbliste
4   @author Bonin 27-Apr-1998
5          Update 16-Jul-1998
6   @version 1.0
7  */
8  import java.awt.* ;
9
10 public class ListWahlFarbe extends Frame {
11   public static void main (String argv[]) {
12     new ListWahlFarbe().show();
13   }
14   public ListWahlFarbe() {
15     Label myL = new Label("Hello World");
16     List myFarbList = new List(3);
17     myFarbList.add("rot");
18     myFarbList.add("gruen");
19     myFarbList.add("blau");
20
21     this.add("West", myFarbList);
22     this.add("Center", myL);
23     this.pack();
24
25     SetFarbe sf = new SetFarbe(myL);
26     // Aktion besteht im Doppelklick auf List-Eintragung
27     myFarbList.addActionListener(sf);
28   }
29 }
30 //End of file cl3:/u/bonin/myjava/ListWahlFarbe.java
31
```

event	*listener*	*method*
`ActionEvent`	`ActionListener`	`actionPerformed()`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged()`

`ComponentEvent`	`ComponentListener`	`componentResized()`
		`componentMoved()`
		`componentShown()`
		`componentHidden()`

`FocusEvent`	`FocusListener`	`focusGained()` `focusLost()`
`ItemEvent`	`ItemListener`	`itemStateChanged()`

`KeyEvent`	`KeyListener`	`keyTyped()`
		`keyPressed()`
		`keyReleased()`

`MouseEvent`	`MouseListener`
	
	`MouseMotionListener`
		`mouseClicked()`
		`mouseEntered()`
		`mouseExited()`
		`mousePressed()`
		`mouseReleased()`
		`mouseDragged()`
		`mouseMoved()`

`WindowEvent`	`WindowListener`	`windowClosed()`
		`windowClosing()`
		`windowDeiconified()`
		`windowIconified()`
		`windowOpened()`

Legende:
Aus Effizienzgründe gibt es zwei *Listener* für ein `MouseEvent`.

Tabelle 6.1: Listener-Typen: event→listener→method

6.2.2 Event→Listener→Method

Für verschiedene Ereignisse gibt es unterschiedliche *Listener* mit fest vorgegebenen Methoden (→Tabelle 6.1 auf Seite 107). Das Beispiel `ZeigeTastenWert.java` nutzt den `KeyListener` (→Seite 107). Verständlicherweise kann nicht jedes Objekt jedem *Listener* zugeordnet werden; beispielsweise setzt der `WindowListener` ein Objekt der Klasse `Frame` voraus. Tabelle 6.2 auf Seite 108 zeigt die Zuordnung. Die Auslösung des Ereignisses ist ebenfalls abhängig vom jeweiligen Objekt. Zum Beispiel ist es ein Mausklick beim `Button` und ein Mausdoppelclick beim `List`-Objekt. Tabelle 6.3 auf Seite 108 zeigt die jeweiligen Aktionen. Darüber hinaus ist bedeutsam, welcher Wert zur Identifizierung des jeweiligen Objekts von `getActionCommand()` zurückgegeben wird. Tabelle 6.4 auf Seite 108 nennt die Werte.

6.2.3 `KeyListener` — Beispiel `ZeigeTastenWert.java`

Die Abbildung 6.6 auf Seite 109 zeigt das Klassendiagramm.

```
1  /**
2   Kleines Beispiel fuer die
```

GUI *object*	*listener*
`Button`	`ActionListener`
`Choice`	`ItemListener`
`Checkbox`	`ItemListener`

`Component`	`ComponentListener`
	`FocusListener`
	`KeyListener`
	`MouseListener`
	`MouseMotionListener`

`Dialog`	`WindowListener`
`Frame`	`WindowListener`
`List`	`ActionListener` `ItemListener`
`MenuItem`	`ActionListener`
`Scrollbar`	`AdjustmentListener`
`TextField`	`ActionListener`

Legende:
listener →Tabelle 6.1 auf Seite 107

Tabelle 6.2: Object→listener

GUI *object*	Benutzeraktion
`Button`	Click auf den `Button`
`List`	Doppelclick auf das gewählte `Item`
`MenuItem`	Loslassen auf dem `MenuItem`
`TextField`	Auslösen der Entertaste

Tabelle 6.3: Benutzeraktion auf das GUI *object*

GUI *object*	Rückgabewert von `getActionCommand()`
`Button`	Text von `Button`
`List`	Text vom gewählten `Item`
`MenuItem`	Text vom gewählten `MenuItem`
`TextField`	Gesamte Text des Feldes

Tabelle 6.4: Rückgabewert von `getActionCommand()`

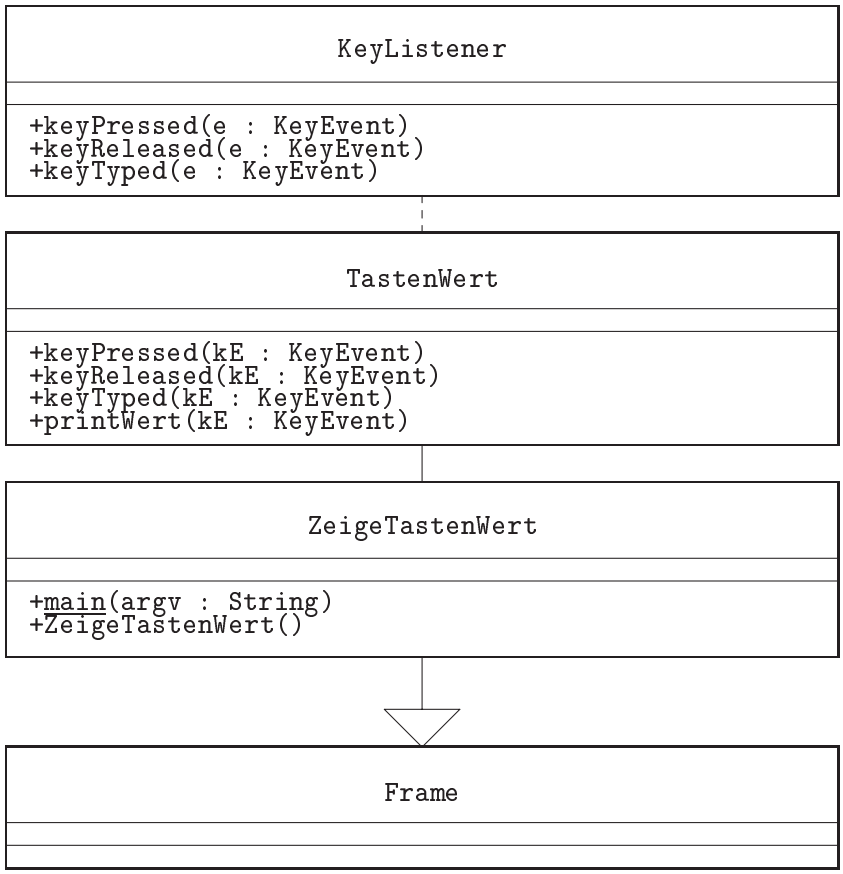


Abbildung 6.6: Klassendiagramm für `ZeigeTastenWert.java`

```
 3   Tasten-Rueckgabewerte der Tastatur
 4   Idee aus Glenn Vanderburg, et al.; MAXIMUM JAVA 1.1, pp.239
 5   Quellcode  modifiziert.
 6   @author Bonin 27-Apr-1998
 7           update 16-Jul-1998
 8   @version 1.0
 9 */
10 import java.awt.event.KeyListener;
11 import java.awt.event.KeyEvent;
12 import java.awt.* ;
13
14 public class ZeigeTastenWert extends Frame {
15   public static void main(String argv[]) {
16     new ZeigeTastenWert().show();
17   }
18   public ZeigeTastenWert() {
19     Label l = new Label("Hello World");
20     this.add(l);
21     this.pack();
22
23     TastenWert tW = new TastenWert();
24     l.addKeyListener(tW);
25   }
26 }
27 class TastenWert implements KeyListener {
28   public void keyPressed(KeyEvent kE) {
29     if (!Character.isLetter(kE.getKeyChar())) {
30       Toolkit.getDefaultToolkit().beep();
31       kE.consume();
32      }
33     printWert(kE);
34   }
35   public void keyReleased(KeyEvent kE) {
36     printWert(kE);
37   }
38   public void keyTyped(KeyEvent kE) {
39      printWert(kE);
40   }
41   public void printWert(KeyEvent kE) {
42     System.out.println(kE.toString());
43   }
44 }
45 //End of file cl3:/u/bonin/myjava/ZeigeTastenWert.java
46
```

6.3 Persistente Objekte

Der Begriff Persistenz bedeutet üblicherweise[2] das Beibehaltung eines Zustandes über einen längeren Zeitraum. Im Zusammenhang mit der Objekt-

[2] besonders in der Biologie und der Medizin

orientierung beschreibt Persistenz die Existenz eines Objektes unabhängig vom Ort und der Lebensdauer seines erzeugenden Programmes. Ein persistentes Objekt kann in eine Datei geschrieben werden und später benutzt oder an einen anderen Rechner übertragen werden. Um Objekt-Persistenz mit dem JDK zu erreichen, sind folgende Schritte erforderlich:

Serialization

1. Konvertierung der Repräsentation des Objektes im Arbeitsspeicher (≡ memory layout) in einen sequentiellen Bytestrom, der geeignet ist für eine Speicherung in einer Datei oder für eine Netzübertragung.

Casting

2. (Re-)Konstruktion eines Objektes aus dem sequentiellen Bytestrom in der ursprünglichen Form mit dem „identischen Verhalten“.

Da die Persistenz über einen Bytestrom erreicht wird, bleibt die Objektidentität selbst nicht erhalten. Persistent ist nur das Objektverhalten, da alle Methoden und Variablen mit ihren Werten aus dem Bytestrom und dem Lesen der jeweiligen Klassen wieder rekonstruiert werden.[3]

Bei der Serialization werden nur die Variablen mit ihren Werten und die Klassendeklaration codiert und in den Bytestrom geschrieben. Der *Java Virtual Maschine Byte Code*, der die Methoden des Objektes abbildet, wird dabei nicht gespeichert. Wenn ein Objekt rekonstruiert wird, dann wird die Klassendeklaration gelesen und der normale Klassenladungsmechanismus, das heißt, Suchen entlang dem `CLASSPATH`, wird ausgeführt. Auf diese Art wird der *Java Byte Code* der Methoden verfügbar. Wird die Klasse nicht gefunden, kommt es zu einer Ausnahme, genauer formuliert:

```
readObject() throws ClassNotFoundException
```

!Agent

Diese JDK-Form der Persistenz ist daher nicht hinreichend für Objekte, die sich wie Agenten völlig frei im Netz bewegen sollen.

Ein besonderes Problem der *Serialization* liegt in der Behandlung der Referenzen auf andere Objekte. Von allen Objekten die vom persistenten Objekt referenziert werden, muß eine „Objektkopie“ mit in den Bytestrom gespeichert werden. Referenziert ein referenziertes Objekt wieder ein anderes Objekt, dann muß auch dessen Kopie in den Bytestrom kommen und so weiter. Der Bytestrom wird daher häufig sehr viele Bytes umfassen, obwohl das zu serialisierende Objekt selbst recht klein ist. Die Referenzen können ein Objekt mehrfach betreffen oder auch zirkulär sein. Um dieses Problem zu lösen wird nur jeweils einmal der „Inhalt eines Objektes“ gespeichert und die Referenzen dann extra. (Näheres dazu beispielsweise →[Vanderburg97] pp. 554–559)

Im folgenden werden einige Konstrukte, die ein Objekt persistent machen, anhand eines Beispiels dargestellt. Als Beispielobjekt dient ein Button, der beim Drücken eine Nachricht auf die Java-Console schreibt. Dieser Button wird in der Datei `PersButton.java` beschrieben (→Seite 113). Seine Klasse `PersButton` ist eine Unterklasse von `Button` und implementiert

[3] Damit ist Objekt-Persistenz zu unterscheiden von einem einfachen Speichern einer Zeichenkette (string) wie es beispielsweise durch die Methoden `save()` und `load` der Klasse `java.util.Properties` erfolgt. Dort wird nur der Inhalt der Zeichenkette gespeichert und die zugehörenden Methoden der Klasse `String` werden nicht berücksichtigt.

das Interface `ActionListener` damit über die Methode `actionPerformed()` das Drücken als Ereignis die Systemausgabe veranlaßt. Weil eine mit dem Konstruktor `PersButton()` erzeugte Instanz serialisiert werden soll, implementiert die Klasse `PersButton` auch das Interface `Serializable`. Ohne eine weitere Angabe als `implements Serializable` wird das *default Java runtime serialization format* benutzt.

Mit der Klasse `PersButtonProg` in der Datei `PersButtonProg.java` wird in deren Methode `main()` der Beispielbutton `foo` erzeugt (→Seite 114). Das Schreiben in die Datei `PButton.ser` erfolgt in einem `try-catch`-Konstrukt um Fehler beim Plattenzugriff abzufangen. Die eigentliche Serialization und das Schreiben von `foo` erfolgt durch:

```
out.writeObject(foo)
```

Dabei ist `out` eine Instanz der Klasse `ObjectOutputStream`. Bei der Erzeugung von `out` wird dem Konstruktor eine Instanz der Klasse `FileOutputStream` übergeben. Diese übergebene Instanz selbst wird erzeugt mit ihrem Konstruktor, dem der Dateiname als Zeichenkette übergeben wird. Durch die Klasse `java.util.zip.GZIPOutputStream` wird dafür gesorgt, daß die Daten komprimiert werden, bevor sie in die Datei geschrieben werden. Die Serialization fußt hier auf der folgenden Konstruktion:

→file

```
FileOutputStream  fout = new FileOutputStream("PButton.ser");
GZIPOutputStream gzout = new GZIPOutputStream(fout);
ObjectOutputStream out = new ObjectOutputStream(gzout);
out.writeObject(foo);
out.close();
```

In der Datei `UseButton.java` steht die Klasse `UseButton` mit ihrer Methode `doUserInterface()` (→Seite 114). In dieser Methode wird der Bytestrom gelesen und unser Buttonobjekt wieder rekonstruiert. Dazu dient die Methode `readObjekct()`. Diese erzeugt eine Instanz der Klasse `Object` und nicht automatisch eine Instanz der Klasse `PersButton`. Es ist daher ein *Casting* erforderlich, das heißt, es bedarf einer Konvertierung von `Object`→`PersButton`. Die Rekonstruktion fußt hier auf der folgenden Konstruktion:

←file

```
FileInputStream  fin = new FileInputStream("PButton.ser");
GZIPInputStream gzin = new GZIPInputStream(fin);
ObjectInputStream in = new ObjectInputStream(gzin);
PersButton bar = (PersButton) in.readObject();
in.close();
```

Der ursprüngliche Name des Beispielsbuttons `foo` geht verloren. Der rekonstruierte Button aus dem Bytestrom der Datei `PButton.ser` wird jetzt mit `bar` referenziert.

Um mehr Kontrolle über die *Serialization* zu gewinnen, gibt es das Interface `Externalizable`, eine Spezialisierung des Interfaces `Serializable`.

Dabei kann man dann beispielsweise selbst entscheiden, welche Superklassen mit in den Bytestrom kommen.

Beim Speichern unseres Beispielbuttons `foo` wird eine `serialVersionUID` der Klasse `PersButton` mit in den Bytestrom geschrieben. Der Wert von `serialVersionUID` wird aus einem Hashcode über die Variablen, Methoden und Interfaces der Klasse berechnet. Beim Rekonstruieren des Beispielbuttons in `UseButton` wird aus der (nun über den `CLASSPATH`) geladenen Klasse `PersButton` wieder mittels Hashcode der Wert berechnet. Gibt es eine Abweichung, dann stimmt die Version der Klasse zum Zeitpunkt der Rekonstruktion nicht mit der Version der Klasse zum Zeitpunkt des Schreibens in den Bytestrom überein. Um „alte Objekte“ trotz einer Veränderung ihrer Klasse noch verfügbar zu machen, gibt es eine Unterstützung des Versionsmanagements, das heißt, es kann in die Versionskontrolle eingegriffen werden (Stichwort: Investitionsschutz für alte, nützliche Objekte).

transient

Nicht jedes Objekt ist *serializable*, zum Beispiel wäre eine Instanz `baz` der Klasse `java.lang.Thread` so nicht behandelbar. Um die Serialization zu verhindern, ist bei der Deklaration das Kennwort `transient` anzugeben.

```
transient Thread baz;
```

Um nach der Rekonstruktion wieder über das mit `transient` gekennzeichnete Objekt zu verfügen, kann in die zu serialisierende Klasse eine Methode `public void readObject()` aufgenommen werden. Wird die serialisierte Klasse rekonstruiert, dann sucht Java nach dieser Methode und wendet sie für das Rekonstruieren der Klasse an.

6.3.1 Serialization — Beispiel `PersButton.java`

```
 1  /**
 2   Kleines Beispiel fuer die
 3   ,,Serializing a button'',  Grundidee ,,Button'' aus:
 4     Glenn Vanderburg, et al.; MAXIMUM JAVA 1.1, pp.543
 5     jedoch eigene Quellcodestruktur.
 6   Teil I: Button-Beschreibng
 7   @author Bonin 30-Apr-1998
 8   @version 1.0
 9  */
10  import java.io.*;
11  import java.awt.event.*;
12  import java.awt.* ;
13
14  public class PersButton extends Button
15    implements ActionListener, Serializable {
16
17    Button myButton;
18
19    public PersButton() {
20      myButton = new Button("Anmelden");
21      System.out.println("Button erzeugt");
22      this.myButton.addActionListener(this);
23    }
```

```
24
25    public void actionPerformed(ActionEvent e) {
26      System.out.println("Button gedrueckt");
27    }
28  }
29  //End of file cl3:/u/bonin/myjava/PersButton.java
```

```
 1  /**
 2   Kleines Beispiel fuer die
 3   ,,Serializing a button''
 4   Teil II: Schreiben eines Button in PButton.dat
 5   @author Bonin 30-Apr-1998
 6           Update 05-May-1998
 7   @version 1.0
 8  */
 9  import java.io.*;
10  import java.util.zip.*;
11
12  public class PersButtonProg {
13
14    public static void main(String args[]) {
15      PersButton foo = new PersButton();
16      try {
17        FileOutputStream fout = new FileOutputStream("PButton.ser");
18        GZIPOutputStream gzout = new GZIPOutputStream(fout);
19        ObjectOutputStream out = new ObjectOutputStream(gzout);
20        out.writeObject(foo);
21        out.close();
22        System.exit(0);
23      }
24      catch (Exception e) {
25        e.printStackTrace(System.out);
26      }
27    }
28  }
29  //End of file cl3:/u/bonin/myjava/PersButtonProg.java
30
```

6.3.2 Rekonstruktion — Beispiel `UseButton.java`

```
 1  /**
 2   Kleines Beispiel fuer die
 3   ,,Use a persistent button object''
 4   @author Bonin 29-Apr-1998
 5           Update 05-May-1998
 6   @version 1.0
 7  */
 8  import java.io.*;
 9  import java.awt.event.*;
10  import java.awt.* ;
11  import java.util.zip.*;
```

```
12
13  public class UseButton extends Frame {
14
15    public static void main(String args[]) {
16      Frame myFrame = new Frame("Willi will ...?");
17      Panel myP = new Panel();
18      UseButton myUseButton = new UseButton();
19      myUseButton.doUserInterface(myFrame, myP);
20      myFrame.pack();
21      myFrame.show();
22    }
23
24    public void doUserInterface(Frame frame, Panel panel) {
25      try {
26        FileInputStream fin = new FileInputStream("PButton.ser");
27        GZIPInputStream gzin = new GZIPInputStream(fin);
28        ObjectInputStream in = new ObjectInputStream(gzin);
29
30        PersButton bar = (PersButton) in.readObject();
31
32        in.close();
33        frame.setLayout(new BorderLayout());
34        panel.add(bar.myButton);
35        frame.add("Center", panel);
36      }
37      catch (Exception e) {
38        e.printStackTrace(System.out);
39      }
40    }
41  }
42  //End of file cl3:/u/bonin/myjava/UseButton.java
43
```

```
1  cl3:/u/bonin/myjava:>java -fullversion
2  java full version "JDK1.1.4 IBM build a114-19971209" (JIT on)
3  cl3:/u/bonin/myjava:>javac PersButton.java
4  cl3:/u/bonin/myjava:>javac PersButtonProg.java
5  cl3:/u/bonin/myjava:>javac UseButton.java
6  cl3:/u/bonin/myjava:>java PersButtonProg
7  Button erzeugt
8  cl3:/u/bonin/myjava:>java UseButton
9  Button gedrueckt
10 Button gedrueckt
11 Button gedrueckt
12 cl3:/u/bonin/myjava:>
```

6.3.3 `jar` (*Java Archiv*)

Da ein serialisiertes Objekt zu seiner Rekonstruktion seine Klasse benötigt, bietet es sich an beide in einem gemeinsamen Archiv zu verwalten. Dazu

JAR

gibt es im JDK das Werkzeug JAR, das *Java Archiv*. . Üblicherweise werden in einer solchen Archivdatei mit dem Suffix `.jar` folgende Dateitypen zusammengefaßt:

- `<filename>.ser`
  Dateien der serialisierten Objekte
- `<filename>.class`
  Dateien der dazugehörenden Klassen
- Sound- und Image-Dateien

Die `jar`-Kommandos werden mit dem Aufruf ohne Kommandos angezeigt:

```
cl3:/u/bonin/myjava:>jar
Usage: jar {ctx}[vfm0M] [jar-file] [manifest-file] files ...
Options:
  -c  create new archive
  -t  list table of contents for archive
  -x  extract named (or all) files from archive
  -v  generate verbose output on standard error
  -f  specify archive file name
  -m  include manifest information from specified manifest file
  -0  store only; use no ZIP compression
  -M  Do not create a manifest file for the entries

If any file is a directory then it is processed recursively.
Example: to archive two class files into an archive called classes.jar:
  jar cvf classes.jar Foo.class Bar.class
Note: use the '0' option to create a jar file that
can be put in your CLASSPATH
cl3:/u/bonin/myjava:>
```

In unserem Beispiel bietet sich folgendes `jar`-Kommando an:

```
jar cf Button.jar PButton.ser PersButton.class UseButton.class
```

Dabei bedeuten die Parameter `cf`, daß ein neues Archiv mit dem Dateinamen des ersten Argumentes erzeugt wird.

Das *Java Archiv* ist besonders nützlich für Applets, da so mehrere Dateien mit einem HTTP-Request übertragen werden. (Näheres dazu →[Vanderburg97] pp. 559–564). Das *Java Archiv* bildet die Grundlage für *Java Beans* (→Abschnitt 6.7 auf Seite 139).

6.4 Geschachtelte Klassen (*Inner Classes*)

In Java können Klassen innerhalb von Klassen definiert werden. Um diesen Mechanismus der sogenannten *Inner Classes* zu erläutern, werden einige Konstruktionsalternativen anhand eines sehr einfachen Beispiel gezeigt.

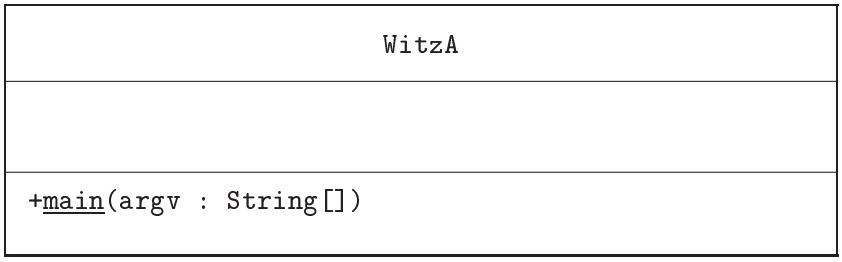


Abbildung 6.7: Klassendiagramm für `WitzA.java`

Dieses Beispiel gibt den Witztext[4] `Piep, piep ... lieb!` als String auf der Systemconsole aus.

Witztext als lokale Variable Zunächst wird eine einfache Klassendefinition `WitzA.java` (→Seite 117) mit einer lokalen Variable betrachtet. In `WitzA.java` ist innerhalb der Klassenmethode `main()` die lokale Variable `text` initialisiert und anschließend wird sie ausgegeben.

Beispiel `WitzA.java` Die Abbildung 6.7 auf Seite 117 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als lokale Variable
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
 7  */
 8  public class WitzA {
 9    public static void main(String[] argv) {
10      final String text = "Piep, piep ... lieb!";
11      System.out.println(text);
12    }
13  }
14  // End of file cl3:/u/bonin/myjava/Aussen/WitzA.java
```

Witztext als Instanzvariable In `WitzB.java` (→Seite 117) wird statt einer lokalen Variablen eine Instanzvariable `text` definiert. Um diese Instanzvariable ausgeben zu können, ist vorher eine Instanz dieser Klasse zu erzeugen. Dazu wird der Konstruktur der Klasse, also `WitzB()`, angewendet.

[4] Schlagerkurztext von Guildo Horn, Mai 1998

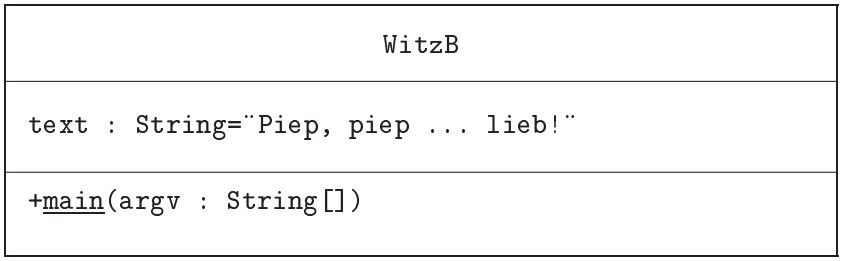


Abbildung 6.8: Klassendiagramm für `WitzB.java`

Beispiel `WitzB.java` Die Abbildung 6.8 auf Seite 118 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als Instanzvariable
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
 7  */
 8  public class WitzB {
 9    final String text = "Piep, piep ... lieb!";
10    public static void main(String[] argv) {
11      WitzB foo = new WitzB();
12      System.out.println(foo.text);
13    }
14  }
15  // End of file cl3:/u/bonin/myjava/Aussen/WitzB.java
```

Witztext als Instanzvariablen einer anderen Klasse In `WitzC.java` (→Seite 118) wird die Instanzvariable `text` in einer eigenen Klasse `WitzText` definiert. Erzeugt wird sie daher mit dem Konstruktor `WitzText()`. Beide Klassen `WitzC` und `WitzText` stehen in derselben Datei[5] `WitzC.java`. Sie sind im gemeinsamen Paket.

Beispiel `WitzC.java` Die Abbildung 6.9 auf Seite 119 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als Instanzvariable einer anderen Klasse
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
 7  */
```

[5] Sollte die Klasse `WitzText` allgemein zugreifbar sein, also mit `public` definiert werden, dann muß sie in einer eigenen Datei mit dem Namen `WitzText.java` stehen.

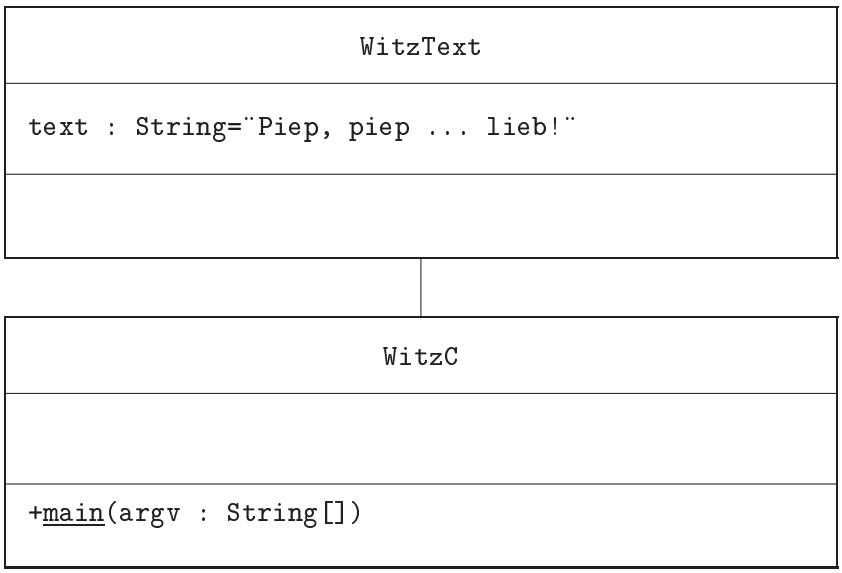


Abbildung 6.9: Klassendiagramm für `WitzC.java`

```
 8  public class WitzC {
 9    public static void main(String[] argv) {
10      WitzText foo = new WitzText();
11      System.out.println(foo.text);
12    }
13  }
14
15  class WitzText {
16    final String text = "Piep, piep ... lieb!";
17  }
18  // End of file cl3:/u/bonin/myjava/Aussen/WitzC.java
```

Witztext als Member Class Wird nun die Klasse `WitzText` nicht außerhalb der Klasse `WitzC` definiert, sondern innerhalb der Definition der Klasse `WitzC`, dann ändert sich auch die Art und Weise des Zugriffs auf die Instanzvariable `text`. Das folgende Beispiel `WitzD.java` (→Seite 119) zeigt diesen Mechanismus der *Inner classes*. Zur Hervorhebung dieser Schachtelung wird die „innere Klasse “ in `WitzTextInnen` umgetauft.

Beispiel `WitzD.java` Die Abbildung 6.10 auf Seite 120 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als Member Class
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
```

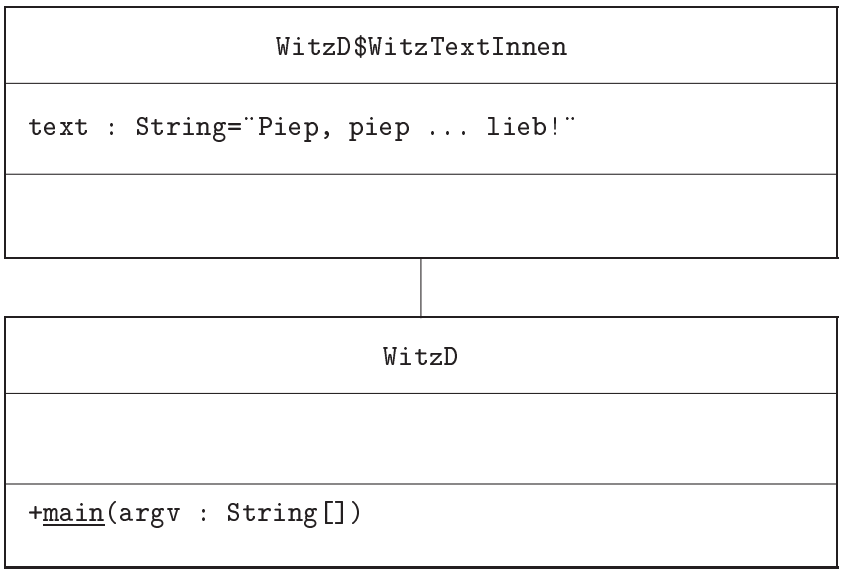


Abbildung 6.10: Klassendiagramm für `WitzD.java`

```
 7  */
 8  public class WitzD {
 9    public static void main(String[] argv) {
10      WitzD foo = new WitzD();
11      WitzD.WitzTextInnen myText = foo.new WitzTextInnen();
12      System.out.println(myText.text);
13    }
14    class WitzTextInnen {
15      final String text = "Piep, piep ... lieb!";
16    }
17  }
18  // End of file cl3:/u/bonin/myjava/Aussen/WitzD.java
```

Witztext als Instanzvariable einer Superklasse Für die Objektorientierung ist die Vererbung ein charakteristisches Merkmal. Daher kann die Instanzvariable `text` der Klasse `WitzText` auch darüber zugänglich gemacht werden (→Seite 120). Der Konstruktor der Subklasse `WitzE()` erzeugt eine Instanz, die auch die Variable `text` enthält.

Beispiel `WitzE.java` Die Abbildung 6.11 auf Seite 121 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als Instanzvariable einer Superklasse
 5   @author Bonin 18-Mai-1998
 6   @version 1.0
```

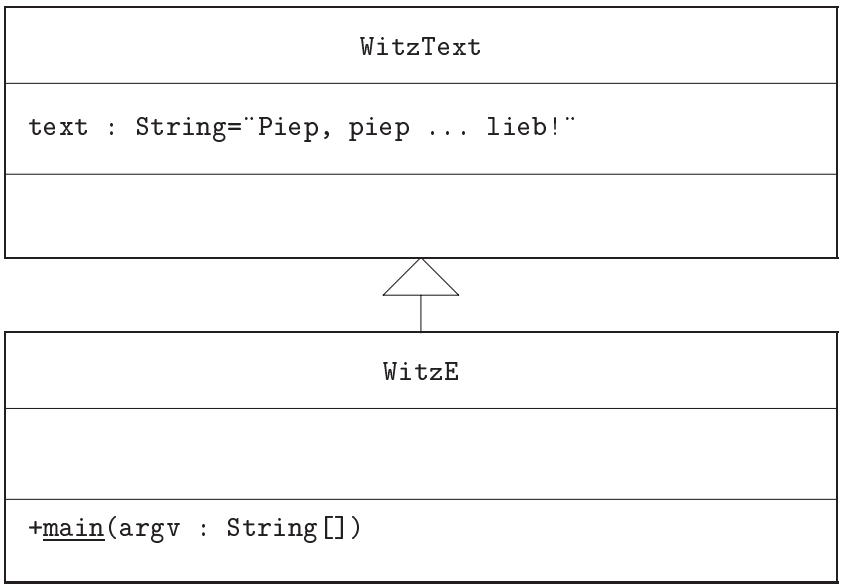


Abbildung 6.11: Klassendiagramm für `WitzE.java`

```
 7  */
 8  public class WitzE extends WitzText {
 9    public static void main(String[] argv) {
10      WitzE foo = new WitzE();
11      System.out.println(foo.text);
12    }
13  }
14  class WitzText {
15    final String text = "Piep, piep ... lieb!";
16  }
17  // End of file cl3:/u/bonin/myjava/Aussen/WitzE.java
```

Witztext als lokale Klasse Eine lokale Klasse wird in einem Block oder einer Methode deklariert. Sie ist daher ähnlich wie eine *Member Class* zu betrachten. Innerhalb der Methode (oder Block), welche die lokale Klasse deklariert, kann direkt mit ihren Konstruktor eine Instanz erzeugt werden (→Seite 121). Außerhalb des Blockes oder der Methode ist keine Instanz erzeugbar.

Beispiel `WitzF.java` Die Abbildung 6.12 auf Seite 122 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als lokale Klasse
 5   @author Bonin 18-Mai-1998
 6   @version 1.0
```

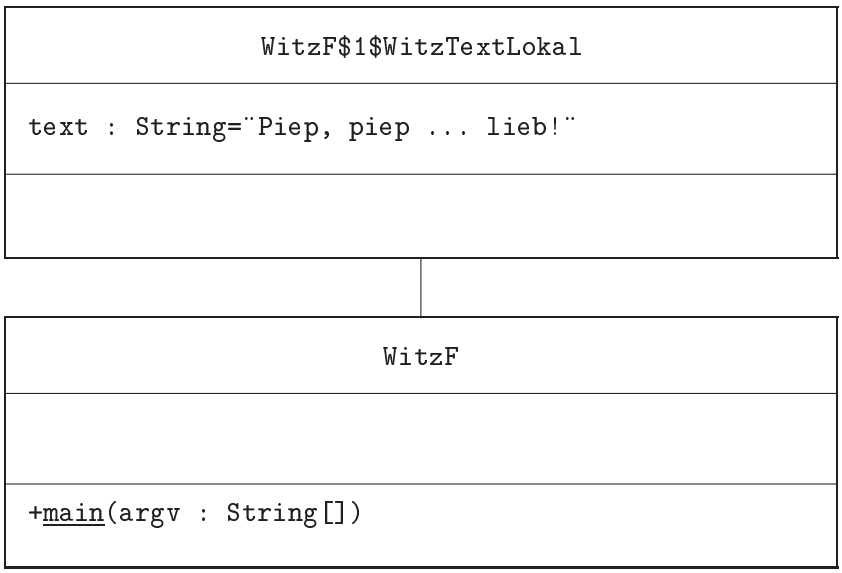


Abbildung 6.12: Klassendiagramm für `WitzF.java`

```
 7  */
 8  public class WitzF {
 9    public static void main(String[] argv) {
10      class WitzTextLokal {
11        final String text = "Piep, piep ... lieb!";
12      }
13      WitzTextLokal foo = new WitzTextLokal();
14      System.out.println(foo.text);
15    }
16  }
17  // End of file cl3:/u/bonin/myjava/Aussen/WitzF.java
```

Witztext als anonyme Klasse Wenn der Name einer lokalen Klasse unbedeutend ist und eher die Durchschaubarkeit verschlechtert als erhöht, dann kann auch eine anonyme Klasse konstruiert werden (→Seite 122). Mit der Ausführung von:

```
WitzG foo = new WitzG(){ .. }
```

wird eine anonyme Klasse erzeugt und instanziert, die die Klasse `WitzG` spezialisiert, also hier die Methoden `bar()` der Klasse `WitzG` überschreibt. Die Instanz `foo` ist als eine Instanz dieser anonymen Klasse zu betrachten. Daher muß die Signatur der Methode `bar()` in beiden Fällen gleich sein. Um dies zu verdeutlichen, ist zusätzlich die alternative Konstruktion `WitzGa.java` hier angeführt (→Seite 123).

Beispiel `WitzG.java` Die Abbildung 6.13 auf Seite 123 zeigt das Klassendiagramm.

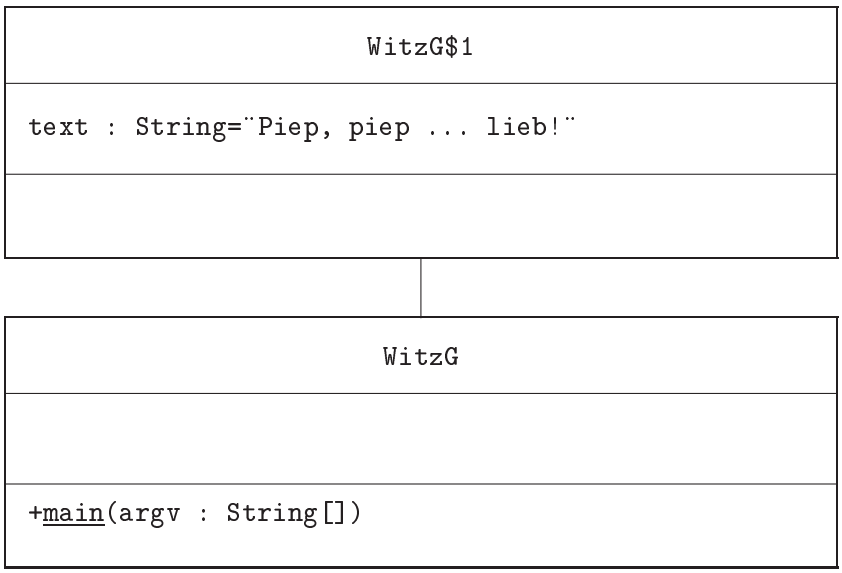


Abbildung 6.13: Klassendiagramm für `WitzG.java`

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als anonyme Klasse
 5   @author Bonin 19-Mai-1998
 6   @version 1.0
 7  */
 8
 9  public class WitzG {
10    public void bar(){};
11    public static void main(String[] argv) {
12      WitzG foo = new WitzG(){
13        public void bar () {final String text = "Piep, piep ... lieb!";
14        System.out.println(text);}};
15      foo.bar();
16    }
17  }
18  // End of file cl3:/u/bonin/myjava/Aussen/WitzG.java
```

Beispiel `WitzGa.java` Die Abbildung 6.14 auf Seite 124 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als anonyme Klasse - Alternative
 5   @author Bonin 19-Mai-1998
 6   @version 1.0
```

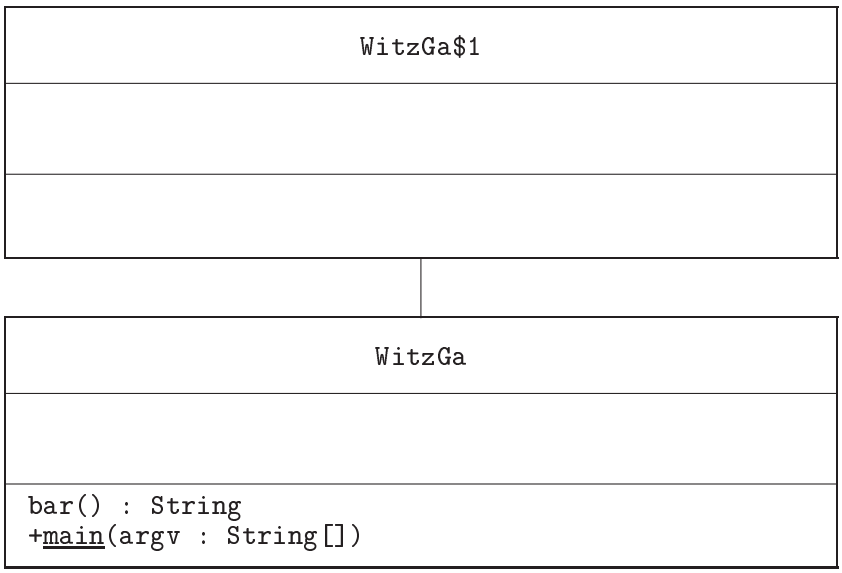


Abbildung 6.14: Klassendiagramm für `WitzGa.java`

```
 7  */
 8  public class WitzGa {
 9    public String bar(){
10      final String text = "Piep, piep ... lieb!";
11      return text;
12    }
13    public static void main(String[] argv) {
14      WitzGa foo = new WitzGa(){
15        public String bar () {
16          System.out.println(super.bar());
17          return "";
18       }
19      };
20
21      foo.bar();
22    }
23  }
24  // End of file cl3:/u/bonin/myjava/Aussen/WitzGa.java
```

Witztext als Klassenvariable derselben Klasse Eine einfache Konstruktion definiert den Witztext als eine Klassenvariable `text` der eigenen Klasse (→Seite 124).

Beispiel `WitzH.java` Die Abbildung 6.15 auf Seite 125 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
```

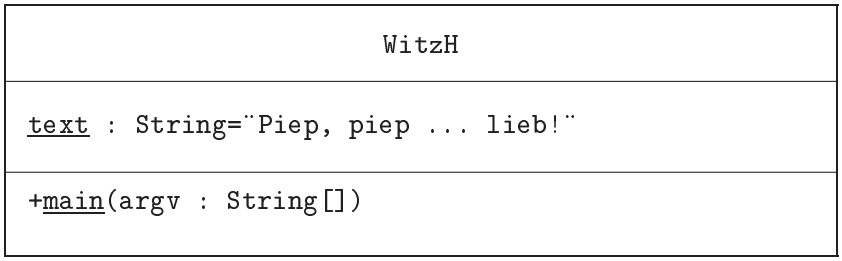


Abbildung 6.15: Klassendiagramm für `WitzH.java`

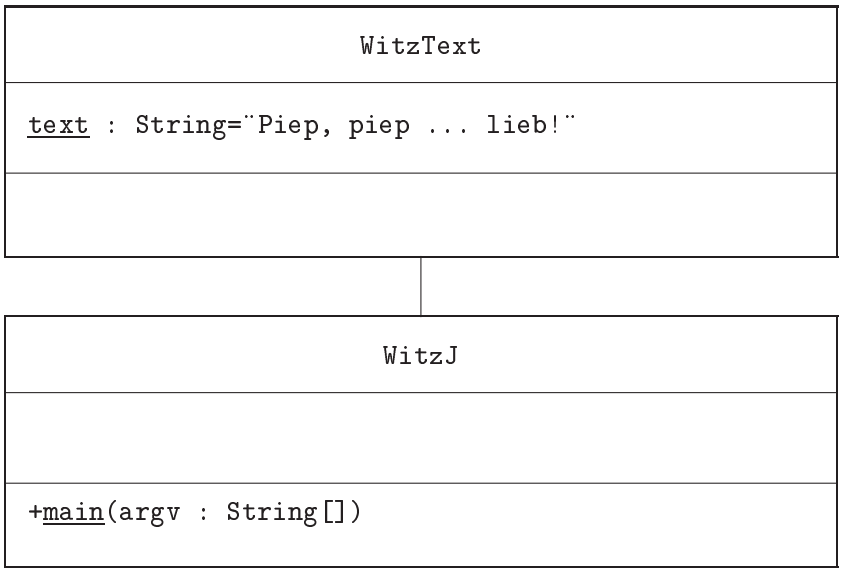


Abbildung 6.16: Klassendiagramm für `WitzJ.java`

```
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als Klassenvariable derselben Klasse
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
 7  */
 8  public class WitzH {
 9    final static String text = "Piep, piep ... lieb!";
10    public static void main(String[] argv) {
11      System.out.println(WitzH.text);
12    }
13  }
14  // End of file cl3:/u/bonin/myjava/Aussen/WitzH.java
```

Witztext als eigene Klasse mit Klassenvariable Etwas aufwendiger ist eine Klassenvariable `text` in einer eigenen Klasse `WitzText` (→Seite 125).

Beispiel `WitzJ.java` Die Abbildung 6.16 auf Seite 125 zeigt das Klassendiagramm.

```
 1  /**
 2   Kleines Beispiel fuer
 3   ,,Konstruktionsalternativen''
 4   hier: Witztext als eigene Klasse mit Klassenvariable
 5   @author Bonin 13-Mai-1998
 6   @version 1.0
 7  */
 8  public class WitzJ {
 9    public static void main(String[] argv) {
10          System.out.println(WitzText.text);
11    }
12  }
13  class WitzText {
14    final static String text = "Piep, piep ... lieb!";
15  }
16  // End of file cl3:/u/bonin/myjava/Aussen/WitzJ.java
```

6.4.1 Beispiel `Aussen.java`

Nachdem die verschiedene Konstruktionen über die Ausgabe eines Witztextes den Mechanismus der *Inner Classes* im Gesamtzusammenhang verdeutlicht haben, wird nun anhand eines neuen Beispiels die Schachtelungstiefe erhöht. Die Idee zum folgenden Beispiel für „*inner classes*“ stammt von [Flanagan97] S. 109–110. Zu beachten ist dabei, daß im `new`-Konstrukt die Klasse relativ zur Instanz, die diese enthält, angegeben wird, das heißt zum Beispiel:

```
Aussen.Innen.GanzInnen g = i.new GanzInnen();
```

und nicht:

```
Aussen.Innen.GanzInnen g = new Aussen.Innen.GanzInnen();
```

```
 1  /**
 2    Kleines Beispiel fuer
 3    ,,inner classes''
 4
 5    @author Bonin 12-Mai-1998
 6    @version 1.0
 7  */
 8
 9  public class Aussen {
10    private String name = "Aussen";
11    class Innen {
12      public String name = "Innen";
13      class GanzInnen {
14        public String name = "GanzInnen";
15        public void printSituation() {
```

```
16          System.out.println(name);
17          System.out.println(this.name);
18          System.out.println(GanzInnen.this.name);
19          System.out.println(Innen.this.name);
20          System.out.println(Aussen.this.name);
21        }
22      }
23    }
24
25     public static void main(String[] argv) {
26      // Erzeugen einer Instanz von Aussen
27      Aussen a = new Aussen();
28
29      // Erzeugen einer Instanz von Innen innerhalb der Instanz a
30      Aussen.Innen i = a.new Innen();
31
32      // Erzeugen einer Instanz von GanzInnen innerhalb der Instanz b
33      Aussen.Innen.GanzInnen g = i.new GanzInnen();
34
35      // Aufrufen der Methode der Instanz g
36      g.printSituation();
37     }
38  }
39  // End of file cl3:/u/bonin/myjava/Aussen/Aussen.java
```

Die inneren Klassen werden als eigene Dateien erzeugt. Der Dateiname besteht aus den Namen der „äußeren Klassen“ jeweils getrennt durch ein Dollarzeichen, dem Klassennamen und dem Suffix `.class`.

```
 1  cl3:/u/bonin/myjava/Aussen:>java -fullversion
 2  java full version "JDK1.1.4 IBM build a114-19971209" (JIT on)
 3  cl3:/u/bonin/myjava/Aussen:>javac Aussen.java
 4  cl3:/u/bonin/myjava/Aussen:>ls Aussen*
 5  Aussen$Innen$GanzInnen.class   Aussen.java
 6  Aussen$Innen.class             Aussen.log
 7  Aussen.class
 8  cl3:/u/bonin/myjava/Aussen:>java Aussen
 9  GanzInnen
10  GanzInnen
11  GanzInnen
12  Innen
13  Aussen
14  cl3:/u/bonin/myjava/Aussen:>
```

6.4.2 Beispiel `BlinkLicht.java`

Das eine *Inner Class* nützlich sein kann, soll das folgende Applet `BlinkLicht.java` verdeutlichen. Um es zu verstehen, sei an dieser Stelle kurz (nochmals) erwähnt, daß die Methode `paint()` auf zwei Weisen appliziert wird:

1. expliziter Aufruf
   durch Angabe von `paint()`, `repaint()` oder `setVisible(true)` und

2. automatischer („impliziter“) Aufruf
   immer dann, wenn sich die Sichtbarkeit des Fensters am Bildschirm ändert, zum Beispiel durch Verschieben, Verkleinern, oder indem Verdecktes wieder Hervorkommen soll.

Die Methode `drawImage()` hat hier vier Argumenten. Im Quellcode steht folgendes Statement:

```
g.drawImage(bild,0,0,this);
```

Die Argumente haben folgende Bedeutung:

- Das erste Argument ist eine Referenz auf das Bildobjekt.
- Das zweite und dritte Argument bilden die x,y-Koordinaten ab, an deren Stelle das Bild dargestellt werden soll. Dabei wird durch diese Angabe die linke, obere Ecke des Bildes bestimmt.
- Das vierte Argument ist eine Referenz auf ein Objekt von `ImageObserver`.

`ImageObserver` ist ein Objekt „auf welchem das Bild gezeigt“ wird. Hier ist es durch `this` angegeben, also durch das Applet selbst. Ein `ImageObserver-`-Objekt kann jedes Objekt sein, daß das Interface `ImageObserver` implementiert. Dieses Interface wird von der Klasse `Component` implementiert. Da `Applet` eine Unterklasse von `Panel` ist und `Panel` eine Unterklasse von `Container` und `Container` eine Unterklasse von `Component`, ist in einem Applet das Interface `ImageObserver` implementiert.

```
 1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN"
 2    "http://www.w3c.org/TR/REC-html40/strict.dtd">
 3  <!-- Testbett fuer Applet BlinkLicht.class      -->
 4  <!-- Bonin  12-May-1998                         -->
 5  <!-- Update 17-Jul-1998                         -->
 6  <HTML>
 7  <HEAD>
 8  <TITLE>Blinklicht</TITLE>
 9  <STYLE type="text/css">
10    P.links {
11      text-align: left;
12    }
13    EM {
14      font-size: 28pt;
15      font-style: italic;
16      color: #FFFFFF;
17      background-color: #000000;
18    }
19    BODY {
20      color: white;
21      background-color: #000000
22    }
23  </STYLE>
24  </HEAD>
25  <BODY>
26  <H1>Blinklicht</H1>
```

```
27  <P class="links">
28  <OBJECT
29    codetype="application/java"
30    classid="java:BlinkLicht.class"
31    name="Blinker"
32    width="80"
33    height="275"
34    standby="Hier kommt gleich ein Blinklicht!"
35    alt="Java Applet BlinkLicht.class">
36    Java Applet BlinkLicht.class
37  </OBJECT>
38  </P>
39  <P><EM>Gelbes Blinklicht mittels Thread</P>
40  <P> Copyright Prof. Bonin 13-May-1998 all rights reserved</P>
41  <ADDRESS>
42  <A HREF="mailto:hinich-bonin@fbw.fh-lueneburg.de"
43    >hinich-bonin@fbw.fh-lueneburg.de</A>
44  </ADDRES>
45  </BODY>
46  <!-- File cl3: /u/bonin/mywww/BlinkLicht/BlinkLicht.html -->
47  </HTML>
```

```
 1  /**
 2   Kleines Beispiel fuer einen
 3   ,,Thread'',  Idee aus:
 4   Hubert Partl; Java-Einfuehrung, Version April 1998,
 5   Mutsterlösungen, S. 33
 6   http://www.boku.ac.at/javaeinf/
 7   Quellcode leicht modifiziert
 8
 9   @author Bonin 12-Mai-1998
10   @version 1.0
11  */
12  import java.applet.*;
13  import java.awt.* ;
14
15  public class BlinkLicht extends Applet
16    implements Runnable {
17
18    Graphics grafik;
19    Image bild;
20
21    MyCanvas theCanvas;
22
23    Thread myT = null;
24
25    public void init() {
26      setLayout (new FlowLayout());
27      theCanvas = new MyCanvas();
28      theCanvas.setSize(100,260);
29      add(theCanvas);
30      setVisible(true);
31
```

```
32      Dimension d = theCanvas.getSize();
33      bild   = theCanvas.createImage(d.width,d.height);
34      grafik = bild.getGraphics();
35    }
36
37    public void start() {
38      if (myT == null) {
39        myT = new Thread(this);
40        myT.start();
41      }
42      System.out.println("start() appliziert");
43    }
44
45    public void stop() {
46      if (myT != null) {
47        myT.stop();
48        myT = null;
49      }
50      System.out.println("stop() appliziert");
51    }
52
53    public void run() {
54      boolean onOff = false;
55      while (true) {
56        grafik.setColor(Color.black);
57        grafik.fillRect(10,10,80,240);
58        if (onOff) {
59          grafik.setColor(Color.yellow);
60          grafik.fillOval(20,100,60,60);
61        }
62        onOff = !onOff;
63        theCanvas.repaint();
64        try {Thread.sleep(1000);}
65        catch (InterruptedException e) {}
66      }
67    }
68
69
70    private class MyCanvas extends Canvas { // Inner class !!!!
71
72      public Dimension getMinimumSize() {
73        return new Dimension(100,260);
74      }
75
76      public Dimension getPreferredSize() {
77        return getMinimumSize();
78      }
79
80      public void paint (Graphics g) {
81        update(g);
82      }
83
84      public void update (Graphics g) {
85        if (bild != null) g.drawImage(bild,0,0,this);
```

```
86      }
87   } // End of inner class MyCanvas
88  }
89  //End of file cl3:/u/bonin/mywww/BlinkLicht/BlinkLicht.java
```

```
 1  cl3:/home/bonin/mywww/BlinkLicht:>java -fullversion
 2  java full version "JDK1.1.4 IBM build a114-19971209" (JIT on)
 3  cl3:/home/bonin/mywww/BlinkLicht:>javac BlinkLicht.java
 4  cl3:/u/bonin/mywww/BlinkLicht:>ls Bl*
 5  BlinkLicht$MyCanvas.class   BlinkLicht.java
 6  BlinkLicht.class            BlinkLicht.log
 7  BlinkLicht.html
 8  cl3:/home/bonin/mywww/BlinkLicht:>appletviewer
 9    http://as.fh-lueneburg.de/mywww/BlinkLicht/BlinkLicht.html
10  start() appliziert
11  stop() appliziert
12  cl3:/home/bonin/mywww/BlinkLicht:>
```

6.5 Interna einer Klasse (*Reflection*)

Das Paket `java.lang.reflect` ermöglicht zusammen mit der Klasse `java.lang.Class` auf die Interna[6] einer Klasse oder eines Interfaces zuzugreifen. Einige Möglichkeiten dieser sogenannten *Reflection* zeigt das folgende Beispiel `ZeigeKlasse.java` (→Seite 133).

Ausgangspunkt ist die Möglichkeit eine Klasse dynamisch zu laden, indem man der Methode `forName()` von `Class` als Argument den voll qualifizierten Klassennamen (Paketname plus Klassennamen) übergibt. Die Methode `forName(String className)` lädt die Klasse in den Java Interpreter, falls sie nicht schon geladen ist. Rückgabewert ist ein Objekt vom Datentyp `Class`. Mit dem Beispielprogramm `ZeigeKlasse.java` soll beispielsweise die existierende Klasse `OttoPlan` reflektiert werden und zwar mit folgendem Aufruf:

```
C:\myjava>java ZeigeKlasse OttoPLan
```

Die Klasse `OttoPlan` erhält man innerhalb von `ZeigeKlasse` als eine Klasse (≡ Instanz von `Class`) mit folgendem Statement:

```
Class c = Class.forName(argv[0]);
```

Das Paket `java.lang.reflect` hat die Klassen `Field`, `Constructor` und `Method` für die Abbildung von Feldern (= Variablen), Konstruktoren und Methoden (hier von `OttoPlan`). Objekte von ihnen werden zurückgegeben von den Methoden `getDeclared...()` der Klasse `Class`.

```
            Field[] myFields = c.getDeclaredFields();
Constructor[] myConstructors = c.getDeclaredConstructors();
          Method[] myMethods = c.getDeclaredMethods();
```

Auch lassen sich auch die implementierten Interfaces mit einer Methode `getInterfaces()` verfügbar machen.

```
        Class[] myInterfaces = c.getInterfaces();
```

Die Klasse `java.lang.reflect.Modifier` definiert einige Konstanten und Klassenmethoden, um die Integerzahlen, die von der Methode `getModifiers()` zurückgegeben werden, zu interpretieren. Mit `Modifier.toString(c.getModifiers())` erhält man daher die Modifikatoren in Form der reservierten Wörter.

Das Interface `java.lang.reflect.Member` wird von den Klassen `Field`, `Method` und `Constructor` implementiert. Daher kann man die folgende Methode:

```
public static void printMethodOrConstructor(Member member)
```

einmal mit einem Objekt der Klasse `Method` und das andere Mal mit einem Objekt der Klasse `Constructor` aufrufen. Die Typerkennung erfolgt

[6] Interna ≈ nur die inneren, eigenen Verhältnisse angehenden Angelegenheiten; vorbehaltenes eigenes Gebiet

dann mit dem Operator `instanceof` und einem *Casting*, beispielsweise in der folgenden Form:

```
Method m = (Method) member;
```

In der Ausgabe von `ZeigeKlasse.java` sind die Methoden mit ihrem Namen und dem Typ ihrer Parameter angegeben. Die Parameternamen selbst fehlen, denn diese werden nicht in der `class`-Datei gespeichert und sind daher auch nicht über das *Reflection Application Programming Interface* (API) verfügbar.

Beispiel `ZeigeKlasse.java`

```
1  /**
2   Kleines Beispiel fuer die
3   ,,Reflection API''-Moeglichkeiten
4   Idee von David Flanagan; Java Examples in a Nutshell, 1997, p. 257
5   Quellcode modifiziert.
6   @author Bonin 23-Mai-1998
7   @version 1.0
8  */
9  import java.lang.reflect.* ;
10
11 public class ZeigeKlasse {
12
13   String[] myWitz = new String[]{"Piep" , "piep", "...", "lieb!"};
14
15   public static void main(String argv[])
16     throws ClassNotFoundException {
17
18     Class c = Class.forName(argv[0]);
19     printClass(c);
20     }
21
22   /** Gibt von der Klasse die Modifkatoren, den Namen,
23    die Superklasse und das Interface aus.*/
24   public static void printClass(Class c) {
25     if (c.isInterface()) {
26       // Modifikatoren enthalten das Wort "interface"
27       System.out.print(Modifier.toString(c.getModifiers())
28         + c.getName());
29     }
30     // es gibt kein c.isClass() daher else
31     else if (c.getModifiers() != 0)
32       System.out.print(Modifier.toString(c.getModifiers())
33         + " class " +  c.getName() + " extends "
34         + c.getSuperclass().getName());
35     else
36       // Modifier.toString(0) gibt "" zurueck
37       System.out.print("class " + c.getName());
38
39    // Interfaces oder Super-Interfaces
40    // einer Klasse oder eines Interface
41    Class[] myInterfaces = c.getInterfaces();
```

```
42     if ((myInterfaces != null) && (myInterfaces.length > 0)) {
43       if (c.isInterface())
44         System.out.println(" extends ");
45       else
46         System.out.print(" implements ");
47       for (int i = 0; i < myInterfaces.length; i++) {
48         if (i > 0) System.out.print(", ");
49         System.out.print(myInterfaces[i].getName());
50       }
51     }
52     // Beginnklammer fuer Klassenbody
53     System.out.println(" {");
54
55     /** Ausgabe der Felder  */
56     System.out.println("  // Feld(er)");
57     Field[] myFields = c.getDeclaredFields();
58     for (int i = 0; i < myFields.length; i++)
59     printField(myFields[i]);
60
61     /** Ausgabe der Konstruktoren  */
62     System.out.println("  // Konstruktor(en)");
63     Constructor[] myConstructors = c.getDeclaredConstructors();
64     for (int i = 0; i < myConstructors.length; i++)
65       printMethodOrConstructor(myConstructors[i]);
66
67     /** Ausgabe der Methoden */
68     System.out.println("  // Methode(n)");
69     Method[] myMethods = c.getDeclaredMethods();
70     for (int i = 0; i < myMethods.length; i++)
71       printMethodOrConstructor(myMethods[i]);
72
73     // Endeklammer fuer Klassenbody
74     System.out.println("}");
75   }
76
77   /** Drucken Methoden und Konstruktoren */
78   public static void printMethodOrConstructor(Member member) {
79     Class returnType = null, myParameters[], myExceptions[];
80     if (member instanceof Method) {
81       Method m = (Method) member;
82       returnType = m.getReturnType();
83       myParameters = m.getParameterTypes();
84       myExceptions = m.getExceptionTypes();
85     } else {
86       Constructor c = (Constructor) member;
87       myParameters  = c.getParameterTypes();
88       myExceptions  = c.getExceptionTypes();
89     }
90     System.out.print("  " +
91       modifiersSpaces(member.getModifiers()) +
92       ((returnType != null)? typeName(returnType) + " " : "") +
93        member.getName()+ "(");
94     for (int i = 0; i < myParameters.length; i++) {
95       if (i > 0) System.out.print(", ");
```

```
96        System.out.print(typeName(myParameters[i]));
97      }
98      System.out.print(")");
99      if (myExceptions.length > 0)
100       System.out.print(" throws ");
101     for (int i = 0; i < myExceptions.length; i++) {
102       if (i > 0) System.out.print(", ");
103       System.out.print(typeName(myExceptions[i]));
104     }
105     System.out.println(";");
106   }
107
108   /** Aufbereitung der Modifiers mit Zwischenraeumen */
109   public static String modifiersSpaces(int m) {
110     if (m == 0) return "";
111     else return Modifier.toString(m) + " ";
112   }
113
114   /** Feld-Ausgabe mit Modifiers und Type */
115   public static void printField(Field f) {
116     System.out.println("  " +
117       modifiersSpaces(f.getModifiers()) +
118       typeName(f.getType()) + " " + f.getName() + ";");
119   }
120
121   /** Aufbereitung des Namens mit Array-Klammern */
122   public static String typeName(Class t) {
123     String myBrackets = "";
124     while(t.isArray()) {
125       myBrackets += "[]";
126       t = t.getComponentType();
127     }
128     return t.getName() + myBrackets;
129   }
130 }
131 //End of file cl3:/u/bonin/myjava/ZeigeKlasse.java
132
```

```
1  cl3:/u/bonin/myjava:>java -fullversion
2  java full version "JDK1.1.4 IBM build a114-19971209" (JIT on)
3  cl3:/u/bonin/myjava:>javac ZeigeKlasse.java
4  cl3:/u/bonin/myjava:>java ZeigeKlasse ZeigeKlasse
5  public synchronized class ZeigeKlasse extends java.lang.Object {
6    // Feld(er)
7    java.lang.String[] myWitz;
8    // Konstruktor(en)
9    public ZeigeKlasse();
10   // Methode(n)
11   public static void main(java.lang.String[]) throws java.lang.ClassNotFoundException;
12   public static void printClass(java.lang.Class);
13   public static void printMethodOrConstructor(java.lang.reflect.Member);
14   public static java.lang.String modifiersSpaces(int);
15   public static void printField(java.lang.reflect.Field);
16   public static java.lang.String typeName(java.lang.Class);
17 }
18 cl3:/u/bonin/myjava:>
```

6.6 Referenzen & *Cloning*

Wenn man eine „Kopie“ eines Objektes mittels einer Zuweisung anlegt, dann verweist die Referenz der Kopie auf dasselbe Orginalobjekt. Zum Beispiel sollen zwei fast gleiche Kunden erzeugt werden. Dann bietet es sich an, eine Kopie vom zunächst erzeugten Kunden für den zweiten Kunden als Ausgangsbasis zu nutzen.

```
Kunde orignal = new Kunde("Emma AG", "Hamburg", "4-Nov-98");
Kunde copie = orginal;
copie.setName("Otto AG"); // Peng! Emma AG vernichtet
```

Wenn man eine Kopie als ein neues Objekt benötigt, dann muß das Objekt geklont werden. Dazu dient die Methode `clone()`.

```
Kunde orignal = new Kunde("Emma AG", "Hamburg", "4-Nov-98");
Kunde copie = (Kunde) kunde1.clone();
copie.setName("Otto AG"); // OK! orginal bleibt unveraendert
```

Bit→Bit

Die Dinge mit der Methode `clone()` sind aber nicht ganz so einfach. Man stelle sich einmal vor, wie die Methode `clone()` der Klasse `Object` arbeiten kann. Sie hat keine Information über die Struktrur der Klasse `Kunde`. Daher kann sie nur eine *Bit-für-Bit*-Kopie fertigen. Bei nicht primitiven Objekten stellen diese Bits zum Teil Referenzen auf andere Objekt dar. Da nun diese Bits genauso in der Kopie enthalten sind, verweist die Kopie auf dieselben Objekte wie das Orginal. Es gilt daher drei Fälle beim *Cloning* zu unterscheiden:

Cloning

1. Problemloses *Cloning*
   Die Default-Methode `clone()` reicht aus, weil das Orginal nur aus primivtiven Objekten besteht oder die referenzierten Objekte werden später nicht modifiziert.

2. Mühsames *Cloning*
   Für das Objekt kann eine geeignete Methode `clone()` definiert werden. Für jede Instanzvariable wird die Default-Methode `clone()` mit Zusammenhang mit einer `Cast`-Operation explizit angewendete.

3. Hoffnungsloses *Cloning*
   Häufiger Fall — man muß auf das *Cloning* verzichten.

Für den Zugriff auf die Methode `clone()` hat die Klasse das Interface `Cloneable` zu implementieren. Darüberhinaus ist `clone()` der Klasse `Object` zu redefinierten und zwar mit dem Zugriffrecht `public`. Dabei verhält sich das Interface `Cloneable` anders als ein übliches Interface. Man kann es sich mehr als einen Erinnerungsposten für den Programmier vorstellen. Er verweist darauf, daß *Cloning* nicht ohne Kenntnis des Kopierprozesses angewendte werden kann.

Das folgende Beispiel `Person.java` enthält darüberhinaus ein paar Konstruktionsaspekte, die zum Nachdenken anregen sollen. Einerseits zeigt es eine rekursive Beschreibung einer Person durch den Ehegatten, der selbst

wieder eine Person ist. Andererseits nutzt es die Klasse `Vector` in der Ausprägung das JDK1.2.[7]

```
 1  /**
 2  Beispiel einer Rekursion innerhalb der Klasse:
 3   Der Ehegatte und die Kinder sind wieder eine Person
 4  @author Bonin 27-Oct-1998, 4-Nov-98
 5  @version 1.0
 6  */
 7  import java.util.*;
 8
 9  class Person implements Cloneable {
10      private String name = "";
11      private Person ehegatte;
12      private Vector kinder = new Vector();
13
14      public Person(String name) {
15  this.name = name;
16  System.out.println(name + " lebt!");
17      }
18
19      public String getName() {
20  return name;
21      }
22
23      public Person getEhegatte() {
24  return ehegatte;
25      }
26
27      public Person getKind(int i) {
28  return  (Person) kinder.get(i - 1);
29      }
30
31      public Person setName(String name) {
32  this.name = name;
33  return this;
34      }
35
36      public void setEhegatte(Person ehegatte) {
37  this.ehegatte = ehegatte;
38      }
39
40      public void setKind(Person kind, int i) {
41  kinder.add(i - 1, (Object) kind);
42      }
43
44      public Object clone() {
45  try {
46      return super.clone();
47  }
```

[7] Im JDK1.1.n gibt es nicht die Methode `add(int index, Object element)` in der Klasse `Vector`.

```
48  catch (CloneNotSupportedException e) {
49      System.out.println("Objekt nicht geklont");
50      return null;
51  }
52      }
53
54      public static void main(String[] argv) {
55  Person oE = new Person("Otto Eiderente");
56  // Beispiel set-Methode mit Retrun-Wert
57  Person eE = (new Person("Musterperson")).setName("Emma Eiderente");
58
59  oE.setEhegatte(eE);
60  eE.setEhegatte(oE);
61
62  Person madel = new Person("Emmchen Eiderente");
63  eE.setKind(madel,1);
64  oE.setKind(eE.getKind(1), 1);
65
66  Person junge = new Person("Ottochen Eiderente");
67  junge.setEhegatte(madel);
68  eE.setKind(junge,2);
69  oE.setKind(eE.getKind(2), 2);
70
71  System.out.println("Person eE: Ehegatte von Ehegatte ist " +
72     eE.getEhegatte().getEhegatte().getName());
73  System.out.println("Ehegatte vom zweiten Kind vom Ehegatten ist " +
74     eE.getEhegatte().getKind(2).getEhegatte().getName());
75
76  /* Simple Loesung fuer das Problem der mehrfachen Applikation der
77     Methode getEhegatte() auf das jeweilige Resultatobjekt.
78  */
79  // Das Clonen (= Bitstromkopierung) sichert hier nur
80  // die urspruengliche Referenz fuer eE
81  Person eEclone = (Person) eE.clone();
82  int i;
83  for(i = 1; (i < 4); i++) {
84      eEclone = eEclone.getEhegatte();
85  }
86  System.out.println("Von " +  eE.getName() +
87     " ausgehend immer wieder Ehegatte von Ehegatte ist " +
88     eEclone.getName());
89      }
90  }
91  /*
92  Java-Systemkonsole:
93  C:\bonin\myJava>C:\jdk1.2beta4\bin\java -fullversion
94  java full version "JDK-1.2beta4-K"
95
96  C:\bonin\myJava>C:\jdk1.2beta4\bin\javac Person.java
97
98  C:\bonin\myJava>C:\jdk1.2beta4\bin\java Person
99  Otto Eiderente lebt!
100 Musterperson lebt!
101 Emmchen Eiderente lebt!
```

```
102  Ottochen Eiderente lebt!
103  Person eE: Ehegatte von Ehegatte ist Emma Eiderente
104  Ehegatte vom zweiten Kind vom Ehegatten ist Emmchen Eiderente
105  Von Emma Eiderente ausgehend immer wieder Ehegatte von Ehegatte ist Otto Eiderente
106  */
107  // End of file: 193.174.33.99 C:\bonin\myJava\Person.java
108
109
110
111
112
113
114
115
116
```

6.7 Architektur von *Java Beans*

Der eigentliche Traum, den das objekt-orientierte Paradigma vermittelt, ist die Entwicklung von problemlos, überall wiederverwendbaren Softwarekomponenten. Mit einer anwendungsfeldspezifischen Bibliothek von solchen Softwarekomponenten soll sich das Entwickeln der gewünschten Anwendung auf das „Zusammenstecken von Bauteilen“ reduzieren. Programmieren im engeren Sinne ist dann nur noch ein „*plugging in*-Prozeß von genormten Bausteinen„ in den eigenen Rest-Quellcode. Java's Lösung für die *Plug-In*-Bausteine heißt *Java Beans*.

> „*A Java Bean is a reusable software component that can be manipulated visually in a builder tool.*“ ([Vanderburg97] p. 578 oder [Flanagan97] p. 233).

Wenn man zur Manipulation kein kommerzielles *Builder*-Werkzeug[8] verfügbar hat, kann man das *Beans Development Kit*[9] (BDK) von Sun mit seiner Bean-Testbox benutzen.

Ein *Java Bean* ist ein normales Java Objekt[10], das Eigenschaften, Ereignisse und Methoden exportiert und zwar nach vorgegebenen Konstruktionsmustern und (Namens-)Regeln. Diese Vorgaben umfassen primär folgende Punkte:

1. Eine Eigenschaft (*property*) des Objektes ist ein Teil des inneren Zustandes eines *Bean*. Sie wird über öffentliche Zugriffsmethoden verfügbar. Diese `get`- und `set`-Methoden, salopp auch als „Getter“ und „Setter“ bezeichnet, haben eine fest vorgegebene Signatur. Für eine Eigenschaft mit dem Namen `Foo` sind es folgende Methoden:

   - `public FooType getFoo(){...}`

Getter

[8] Liste der Hersteller: `http://splash.javasoft.com/beans/tools.html` (Zugriff: 24-Mai-1998)

[9] BDK Quelle: `http://splash.javasoft.com/beans/bdk_download.html` (Zugriff: 24-Mai-1998)

[10] Auf unterstem Level können beispielsweise alle AWT-Komponenten als Beans bezeichnet werden.

Setter

- `public boolean isFoo(){...}`
- `public void setFoo(FooType wert){...}`
- `public void setFoo(boolean wert){...}`

Zusätzlich gibt es für Eigenschaften auch einen indizierten Zugriff. Dieser Zugriff läßt sich für ein Element mit dem Namen PropertyName und dem Typ PropertyElement wie folgt beschreiben:

- `public PropertyElement getPropertyName(int index){...}`
- `public void setPropertyName(int index, PropertyElement wert){...}`

Listener

2. Die Ereignisbehandlung basiert auf dem Delegationsmodell (*listener classes* für *events*, →Abschnitt 6.2 auf Seite 101). Dazu ist folgendes Paar von Methoden zu definieren:

   - `public void addEventListenerType (EventListenerType l){...}`
   - `public void removeEventListenerType (EventListenerType l){...}`

   Dabei muß EventListenerType abgeleitet sein von `java.util.EventListener`. Sein Name muß mit dem Wort `Listener` enden, also zum Beispiel:

   ```
   addFooListener(FooListener l);
   ```

3. Persistente Objekte basieren auf dem Interface `java.io.Serializable` (→Abschnitt 6.3 auf Seite 110).

Manifest

4. Verfügbar wird ein *Java Bean* als ein *Java Archiv* (JAR) mit einem sogenannten Manifest (JAR-Parameter `-m`, →Seite 115). Eine solche Manifest-Datei hat folgende Eintragungen:

   ```
   Name: className
   Java-Bean: trueOrFalse
   Name: nextClassName
   Java-Bean: trueOrFalse
   ⋮
   ```

   Beispielsweise hat `MyBean` dann folgendes Manifest[11]:

   ```
   Name: myjava/AllBeans/MyBean.class
   Java-Bean: true
   ```

Das folgende *Java-Bean*-Beispiel skizziert grob eine übliche Konstruktion. Für weiter Informationen zum Schreiben von eigenen *Java Beans* siehe zum Beispiel [Vanderburg97].

[11] Hinweis: Auch auf Windows-Plattformen gilt hier der Schrägstrich (*slash*) und nicht der *Backslash*.

Beispiel `SimpleBean.java` Dieses Bean-Beispiel hat eine Eigenschaft `Witz` mit dem Getter `getWitz()` und dem Setter `setWitz()`. Es informiert automatisch „interessierte Parteien“ bevor sich diese Eigenschaft ändert. Man spricht daher von einer *bound*-Eigenschaft. Dazu dient die Klasse `java.beans.PropertyChangeSupport`.

bound

Sie stellt die Methode `firePropertyChange()` bereit, die ein Objekt von `PropertyChangeEvent` an alle registrierten Listener sendet.

Mit der Klasse `java.beans.VetoableChangeSupport` und deren Methoden wird die Eigenschaftsänderung von einer Bedingung abhängig gemacht. Man spricht daher von einer *constrained*-Eigenschaft. Bevor eine Eigenschaftsänderung erfolgt, werden alle Listener angefragt, indem ein `PropertyChangeEvent`-Objekt gesendet wird. Wenn ein Listener ein Veto sendet, dann schickt die Methode `fireVetoableChange()` wieder an alle Listener ein `PropertyChangeEvent`-Objekt um mitzuteilen, daß die Eigenschaft wieder den ursprünglichen Wert hat.

constrained

Selbst wenn man nicht beabsichtigt, eine eigene Klasse als *Java Bean* zu verbreiten, so ist es doch sinnvoll die Konstruktionsmuster und (Namens-)Regeln direkt zu übernehmen.

```
 1  /**
 2   Grundstruktur fuer ein ,,Java Bean''
 3   mit  Kontrollmechanismus fuer eine Aenderung:
 4   ,,bound'' und ,,constrained''
 5   Idee aus Glenn Vanderburg, MAXIMUM Java 1.1, 1997, p. 597
 6   @author Bonin 23-Mai-1998
 7     update 29-May-1998
 8   @version 1.0
 9  */
10  package de.FHNON.beans;
11  import java.beans.*;
12  import java.awt.*;
13
14  public class SimpleBean extends Canvas {
15    String myWitz = "Piep, piep ... lieb";
16    // bound-Eigenschaft
17    // Automatisches Informieren ueber eine Aenderung
18    private PropertyChangeSupport
19      changes = new PropertyChangeSupport(this);
20    // constrained-Eigenschaft
21    // Vetomechanismus fuer eine Aenderung
22    private VetoableChangeSupport
23      vetos = new VetoableChangeSupport(this);
24    // Konstruktor ohne Argument
25    public SimpleBean() {
26      setBackground(Color.green);
27    }
28
29    // Getter
30    public String getWitz() {
31      return myWitz;
32    }
```

```
33
34    // Setter
35    public void setWitz(String neuerWitz)
36      throws PropertyVetoException {
37        String alterWitz = myWitz;
38        vetos.fireVetoableChange("Witz", alterWitz, neuerWitz);
39        // Kein Veto fuer die Aenderung
40        myWitz = neuerWitz;
41        // Nachtraegliche Information ueber Aenderung
42        changes.firePropertyChange("Witz", alterWitz, neuerWitz);
43    }
44
45    // Vetomechanismus fuer Aenderung mit VetoableChangeListener
46    public void addVetoableChangeListener(VetoableChangeListener l)
47    {
48      vetos.addVetoableChangeListener(l);
49    }
50    public void removeVetoableChangeListener(VetoableChangeListener l)
51    {
52      vetos.removeVetoableChangeListener(l);
53    }
54
55    // Aenderungsinformation mit PropertyChangeListener
56    public void addPropertyChangeListener(PropertyChangeListener l)
57    {
58      changes.addPropertyChangeListener(l);
59    }
60    public void removePropertyChangeListener(PropertyChangeListener l)
61    {
62      changes.removePropertyChangeListener(l);
63    }
64
65    // Sonstige exportierte Methoden
66    public Dimension getMinimuSize() {
67      return new Dimension(100, 150);
68    }
69  }
70  // End of file cl3:/u/bonin/myjava/de/FHNON/beans/SimpleBean.java
```

6.8 Integration eines ODBMS

POET

ODMG

Als charakteristisches Beispielprodukt für ein objekt-orientiertes DatenBankManagementSystem dient im Folgenden das Produkt `POET`[12] der Firma POET Software GmbH, Hamburg[13]. Das Modell der ODMG[14] (*Object Data Management Group*) für persistente Objekte in einer Datenbank spezifiziert das Erzeugen eines solchen Objektes im Rahmen einer Transaktion.

[12] POET 5.0 — Java SDK, Version 1.0

[13] Beziehungsweise: POET Software Corporation, San Mateo, USA

[14] ODMG ≡ vormals: Object Database Management Group,
→`http://www.odmg.org/frrbottom.htm` (Zugriff: 27-May-98)

6.8.1 Transaktions-Modell

Man kreiert ein solches „Transaktionsobjekt“ zwischen den Methode `begin()` und `commit()`.

```
import COM.POET.odmg.*;
... // Deklaration von Klassen
Transaction txn = new Transaction();
txn.begin();
... // Datenbankobjekte werden immer
... // innerhalb der Transaktion erzeugt.
txn.commit();
```

Das Erzeugen eines Objektes für die Datenbank und die Zuweisung von Werten geschieht auf die übliche Art und Weise, jedoch innerhalb der Transaktion. Ein Beispiel sei hier die Instanz `myP` einer Klasse `Person`.

```
class Person {
  private String name;
  private Person ehegatte;
   ...
  public static void main(String[] argv) {
    Person myP = new Person();
    myP.setName("Emma Musterfrau");
    Person myE = new Person();
    myE.setName("Otto Mustermann");
    myP.setEhegatte(myE);
  }
}
```

6.8.2 Speichern von Objekten mittels Namen

Noch ist das Objekt `myP` nicht in einer Datenbank gespeichert. Dazu muß zunächst ein Objekt `database` erzeugt werden. Mit Hilfe der Methode `bind()` geschieht dann die Zuordnung zwischen dem Objekt `myP` und der Datenbank `database`. Um das Objekt `myP` später wieder aus der Datenbank `database` zu selektieren, wird als zweites Argument von `bind()` ein String als kennzeichnender Name übergeben:

bind()

```
database.bind(myP,"EMusterF");
```

Die Methode `bind()` speichert auch noch nicht endgültig das Objekt `myP` in die Datenbank. Dies geschieht erst zum Zeitpunkt des Aufrufs der Methode `commit()`. Wird beispielsweise die Transaktion mit der Methode `abort()` abgebrochen, dann ist `myP` nicht gespeichert. Hier sei nochmals kurz zusammengefaßt der Quellcode dargestellt um das Objekt `myP` zu erzeugen und persistent in der Datenbank `database` zu speichern.

```
import COM.POET.odmg.*;
... // Deklaration von Klassen
```

```
Transaction txn = new Transaction();
txn.begin();
  Person myP = new Person();
  myP.setName("Emma Musterfrau");
  database.bind(myP,"EMusterF");
txn.commit();
```

lookup()

Aus der Datenbank wird das Objekt mit der Methode `lookup()` wiedergewonnen und anschließend mittels einer *Casting*-Operation rekonstruiert.

```
Person p = (Person) database.lookup("EMusterF");
System.out.println(p.name);
```

Man kann auch ein Objekt ohne Namen in der Datenbank ablegen. Dazu wird die Methode `bind()` mit dem zweiten Argument `null` aufgerufen.

```
database.bind(myP,null);
```

Solche Datenbankobjekte ohne Namen werden häufig mit Hilfe von `Extent` selektiert (→Abschnitt 6.8.5 auf Seite 145).

6.8.3 Referenzierung & Persistenz

In Java werden Beziehungen zwischen Objekten durch Referenzen abgebildet. Hat beispielsweise eine Person einen Ehegatten, dann wird auf das Objekt Ehegatte über eine Referenz (zum Beispiel über einen Instanzvariablennamen) zugegriffen. Damit sich das aus der Datenbank rekonstruierte Objekt genauso verhält wie das ursprüngliche, müssen alle referenzierten Objekte ebenfalls gespeichert werden. Diese Notwendigkeit wird als *persistence-by-reachability* charakterisiert. In unserem Beispiel „Ehegatte“ muß ein Objekt Ehegatte mit gespeichert werden, und zwar zum Zeitpunkt, wenn die Person selbst gespeichert wird.

```
Transaction txn = new Transaction();
txn.begin();
  Person myP = new Person();
  myP.setName("Emma Musterfrau");
  Person myE = new Person();
  myE.setName("Otto Musterfrau");
  myP.setEhegatte(myE);
  database.bind(myP,"EMusterF");
txn.commit();
```

Nachdem das Objekt `myP` aus der Datenbank wieder selektiert und rekonstruiert ist, ist auch das Objekt „Ehegatte“ wieder verfügbar.

```
Person p = (Person) database.lookup("EMusterF");
// Gibt "Otto Musterfrau" aus.
System.out.println(p.getEhegatte().getName());
```

Im Regelfall ist beim Speichern eines Objektes ein umfangreiches Netzwerk von referenzierten Objekten betroffen, damit das Orginalverhalten des Objektes nach dem Selektieren und Rekonstruieren wieder herstellbar ist.

ODMG-Collections		
Typ	**Sortierung**	**Duplikate**
`Bag`	vom System bestimmt	Ja
`List`	selbstgewählt	Ja
`Set`	vom System bestimmt	Nein
`VArray`	selbstgewählt	Ja

Tabelle 6.5: POET's Java Binding Collection Interfaces

6.8.4 Collections

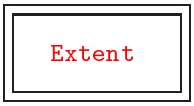
Eine *Collection* ermöglicht einem Objekt, eine Sammlung mit mehreren Objekten[15] zu enthalten. Beim Beispiel „Ehegatte" könnten so die Kinder des Ehepaares abgebildet werden. Der folgende Quellcode gibt daher die Kinder von `Emma Musterfrau` aus.

```
Person p = (Person) database.lookup("EMusterF");
Enumeration e = p.getKinder().createIterator();
while (e.hasMoreElements()) {
  Person k = (Person) e.nextElement();
  // Gibt den Namen des Kindes aus.
  System.out.println(k.getName());
}
```

Entsprechend der ODMG-Spezifikation unterstützt *POET's Java Binding* die *Collections* `Bag`, `List` `Set` und `VArray`. Die Tabelle 6.5 auf Seite 145 zeigt die Unterschiede in Bezug auf die Sortierung der Objekte und auf das mehrfache Vorkommen des gleichen Objektes.

6.8.5 Extent

Objekte können gespeichert werden ohne spezifizierten Namen oder indirekt weil sie referenziert werden über ein Objekt mit Namen. Zusätzlich ist es häufig notwendig auf alle Objekte mit „bestimmten Eigenschaften" in der Datenbank zugreifen zu können und zwar nicht nur über den Weg der einzelnen Objektnamen. Um einen solchen Zugriff auf eine größere Menge von Datenbankobjekten zu ermöglichen, gibt es die Klasse `Extent`[16]. Diese wird benutzt, um alle Objekte einer Klasse in der Datenbank zu selektieren. Immer wenn ein Objekt in die Datenbank gespeichert wird, dann wird eine Referenz für den späteren Zugriff über `Extent` zusätzlich in der Datenbank gespeichert. Der Konstrukor `Extent()` hat daher zwei Parameter:



[15] oder eine Sammlung von Referenzen auf mehrere Objekte

[16] Derzeit definiert weder Java noch das *ODMG Java Binding* ein Konstrukt `Extent`. Es handelt sich dabei (noch?) um eine spezifische POET-Leistung.

1. die gewählte Datenbank und 2. den Klassennamen, der zu selektierenden Objekte. In dem obigen Beispiel wäre folgende Konstruktion erforderlich:

```
Extent allePersonen = new Extent(database, "Person");
```

Das `Extent`-Objekt wird dann benutzt um alle einzelnen persistenten Objekte zu rekonstruieren.

```
while (allePersonen.hasMoreElements()) {
  Person p = (Person) allePersonen.nextElement();
  // Gibt den Namen der Person aus.
  System.out.println(p.getName());
}
```

6.8.6 Transientes Objekt & Constraints

In Java gehört ein Objekt, das mit dem Modifikator `transient` gekennzeichnet ist, nicht zu den persistenten Objekten. Es wird konsequenterweise auch nicht in der Datenbank gespeichert. Ein solches transientes Objekt existiert nur zur Laufzeit des Programms im Arbeitsspeicher. Im Folgenden sei eine Instanzvariable `alter` ein solches transientes Objekt.

```
import java.util.*;
class Person {
  private String name;
  private Date geburtstag;
  private transient int alter;
}
```

Wenn eine Instanz `myP` der Klasse `Person` aus der Datenbank selektiert wird, dann muß beim Rekonstruieren von `myP` auch der Wert von `alter` erzeugt werden. Dazu dient die Methode `postRead()`. Sie wird automatisch vom DBMS nach dem Laden von `myP` aus der Datenbank appliziert. Für die Sicherung der Datenintegrität hält POET drei automatisch applizierte Methoden bereit. Diese werden im Interface `Constraints` vorgegeben.

```
import java.util.*;
class Person implements Constraints {
  private String name;
  private Date geburtstag;
  private transient int alter;

  // Methode zum Initialisieren
  public void postRead() {
    // Nur als Beispiel --- es geht genauer!
    Date heute = new Date();
    alter = heute.getYear() - geburtstag.getYear();
    // ...
  }
```

```
  // Methoden zum Clean-up
  public void preWrite() {
    // ...
  }
  public void preDelete() {
    // ...
  }
}
```

6.8.7 Objekt Resolution

Ein referenziertes Objekt wird vom DBMS erst geladen wenn es tatsächlich benötigt wird. Eine Variable ist daher als spezielles *POET Java reference object* implementiert. Das Laden des referenzierten Objektes in den Arbeitsspeicher wird als *Resolving the Reference* bezeichnet. Anhand eines Beispiels wird dieses *Resolving the Reference* deutlich.

```
class Buch {
  private String titel;
  private Person autor;
  private int jahr;

  public String getTitel() {
    return titel;
  }

  public Person getAutor() {
    return autor;
  }

  public int getJahr() {
    return jahr;
  }

  public Buch(String titel, String autor, int jahr) {
    this.titel = titel;
    this.autor = new Person(autor);
    this.jahr = jahr;
  }
}
```

Zunächst wird ein Objekt `myBuch` der Klasse `Buch` erzeugt und in der lokalen POET-Datenbank `BuchBase` gespeichert.

```
...
Database myDB = Database.open(
  "poet://LOCAL/BuchBase", Database.openReadWrite);
Transaction myT = new Transaction(myDB);
myT.begin();
```

```
try {
  Buch myBuch = new Buch(
    "Softwarekonstruktion mit LISP","Bonin",1991);
  myDB.bind(myBuch,"PKS1");
}
catch (ObjectNameNotUniqueException) {
  System.out.println("PKS1 gibt es schon!");
}
myT.commit();
myDB.close();
```

Mit einem anderen Programm wird das Buch „`Softwarekonstruktion mit LISP`“ wieder selektiert.

```
// Datenbank oeffnen und Transaktion starten
// ...
Buch b = (Buch) myDB.lookup("PKS1");
// ...
```

Wenn man jetzt mit Hilfe von `ObjectServices` abfragt, ob das Objekt `b` *resolved* ist, dann erhält man als Wert `false`.

```
// ...
System.out.println("Buch b  resolved =  " +
  ObjectServices.isResolved(b));
```

Zu diesem Zeitpunkt ist es für POET nur notwendig eine Referenz zum entsprechenden Buchobjekt `b` in der Datenbank zu erzeugen. Erst wenn man mit diesem Objekt `b` arbeitet, geschieht das *Resolving*.

```
// ...
int buchErscheinungsjahr = b.getJahr();
```

Danach ist der Rückgabewert von `ObjectServices.isResolved(b)` gleich `true`. Auch die Referenz auf den Autor, ein Objekt der Klasse `Person` wird erst aufgelöst, wenn der Wert tatsächlich benötigt wird. Erst nach einer „Benutzung“ der Variablen `autor`, zum Beispiel in der Form:

```
// ...
String inhaltVerantwortlich = b.getAutor();
```

hat `ObjectServices.isResolved(b.getAutor())` den Wert `true`. Das *Resolving*-Konzept beim Ladens eines Objektes aus der Datenbank in den Arbeitsspeicher kann man daher auch als *ondemand*-Laden bezeichnen. Dabei ermöglich POET neben dem automatischen (impliziten) *Resolving* auch ein explizites[17]. Dazu dienen die Methoden `resolve()` und `resolveALL()`.

ondemand

6.8.8 Abfragesprache (*OQL*)

OQL

Für das Arbeiten mit einer objektorientierte Datenbanken hat die ODMG als Abfragesprache OQL (*Object Query Language*) standardisiert. OQL basiert wie SQL[18] auf dem Konstrukt

[17] Ein *explizites Resolving* benötigt eine entsprechende Eintragung in der Konfigurationsdatei.

[18] Standart Query Language für eine relationale Datenbank.

```
SELECT ... FROM ... WHERE ...
```

POET's Java Binding ermöglicht mit Hilfe der Klasse `OQLQuery` Abfragen nach diesem Standard. Die Abfrage selbst wird als `String`-Objekt spezifiziert und dem Konstruktor `OQLQuery(String query)` übergeben. Für das obige Beispiel käme daher folgender Quellcode in Betracht:

```
String abfrageString  = "define extent allePersonen for Person;" +
  "select p from p in allePersonen" +
    "where p.getName() = \"Emma Musterfrau\";";
OQLQuery abfrage = new OQLQuery(abfrageString);
Object result abfrage.execute();
```

Wenn die Abfrage eine *Collection* von Objekten ergibt, dann sind die einzelnen Objekte mit Hilfe von `Enumeration` zugreifbar.

```
Enumeration e  = ((CollectionOfObject) result).createIterator();
while (e.hasMoreElements()){ ... };
```

6.8.9 POET's Java Precompiler `ptjavac`

Die Firma POET hat ihr *Java ODMG Binding*[19] über ihren Precompiler `ptjavac` realisiert. Dieser Precompiler interpretiert die POET-spezifischen Konstrukte und ruft danach `javac` auf.[20] Zur Steuerung von `ptjavac` dient eine eigene Konfigurationsdatei. Sie wird über den Parameter `-conf` beim Aufruf zugeordnet.

```
ptjavac -conf myConfigurationFile MyClass.java
```

Der Parameter `-conf` ist optional. Der *Default*-Wert ist `ptjavac.opt`. Die Konfigurationsdatei enthält die Information, welche Klasse persistent ist. Für eine persistente Klasse `Foo` ist eine Eintragung in folgender Form notwendig:

```
[classes\Foo]
persistent = true
```

Die Konfigurationsdatei enthält auch den Namen der Datenbank und den Namen des Klassenlexikons. Dabei wird das Klassenlexikon als Schema bezeichnet. Beide Namen führen zu entsprechenden Dateien im Filesystem des Betriebssystems. Um die Zugriffsgeschwindigkeit zu verbessern, kann die Datenbank als Indexdatei plus Inhaltsdatei im Filesystem abgebildet werden. Gleiches gilt auch für das Schema Diese Aufsplittung geschieht bei der Eintragung `oneFile = false`.

```
[schemata\myDict]
oneFile = true

[databases\myBase]
oneFile = true
```

[19] ODMG Standard Release 2.0

[20] Parameter für den Compiler `javac` werden vom Precompiler `ptjavac` durchgereicht, das heißt, sie können in der Kommandozeile von `ptjavac` angegeben werden.

6.8.10 POET-Beispiel: `MyClass.java`, `Bind.java`, `Lookup.java` und `Delete.java`

Das Einführungsbeispiel[21] verdeutlich die Unterscheidung in

- *persistence capable class* und
  ≡ Klasse, die persistenzfähig ist. Sie hat einen Schema-Eintrag in der Konfigurationsdatei ↪hier: `MyClass`. Ihre Objekte werden in der Datenbank gespeichert.

- *persistence aware class*.
  ≡ Klasse, die Kenntnis von der Persistenz hat. Sie nutzt persistente Objekte. Sie ist nicht in der Konfigurationsdatei vermerkt ↪hier: `Bind`,`Lookup` und `Delete`.

Das hier genutzte POET-System läuft auf einer NT-Plattform (Rechner: 193.174.33.100). Die Umgebungsvariable `CLASSPATH` ist vorab um den Ort der POET-Klassen zu ergänzen.

```
1  set CLASSPATH=%CLASSPATH%;C:\Programme\POET50\Lib\POETClasses.zip;.
2  ptjavac *.java
3  java Bind poet://LOCAL/my_base bonin
4  java Lookup poet://LOCAL/my_base bonin
5  java Delete poet://LOCAL/my_base bonin
```

Als Konfigurationsdateiname wird der *Default*-Name `ptjavac.opt` verwendet. Die Konfigurationsdatei hat folgenden Inhalt:

```
1  /**
2     ptjavac.opt
3   */
4
5  [schemata\my_dict]
6  oneFile = true
7
8  [databases\my_base]
9  oneFile = true
10
11 [classes\MyClass]
12 persistent = true
```

Beispiel `MyClass.java`

```
1  /**
2    Persistence capable class MyClass
3    */
4  import COM.POET.odmg.*;
5  import COM.POET.odmg.collection.*;
```

[21] Quelle: Inhalt des POET-Paketes 5.0
`/POET50/Examples/JavaODMG/First/`
— jedoch leicht modifiziert.