

Bernhard Steppan

Einstieg in Java 6

Auf einen Blick

Vorwort	19
Teil 1 Basiswissen	
1 Digitale Informationsverarbeitung	27
2 Programmiersprachen	43
3 Objektorientierte Programmierung	63
Teil 2 Java im Detail	
4 Sprache	89
5 Entwicklungsprozesse	147
6 Plattform	175
7 Gesetzmäßigkeiten	195
8 Klassenbibliotheken	227
9 Algorithmen	281
Teil 3 Größere Java-Projekte	
10 Konsolenprogramme	297
11 Erste Schritte mit grafischen Oberflächen	313
12 Einfache Oberflächen mit AWT und Swing	347
13 Komplexe Oberflächen mit Swing.....	377
14 Weboberflächen mit Servlets	403
15 Datenbankprogrammierung	429
16 Datenbankanwendungen	449
17 Dynamische Websites	469
Teil 4 Lösungen	
18 Lösungen zu Teil I	485
19 Lösungen zu Teil II	493
20 Lösungen zu Teil III	509
Teil 5 Anhang	
21 Werkzeuge	525
22 Computerhardware	561
23 Glossar	569
24 Literatur	579

Inhalt

Vorwort

19

Teil 1 Basiswissen

1 Digitale Informationsverarbeitung

27

1.1	Einleitung	27
1.2	Zahlensysteme	27
1.2.1	Dezimalsystem	27
1.2.2	Binärsystem	28
1.2.3	Hexadezimalsystem	30
1.3	Informationseinheiten	32
1.3.1	Bit	32
1.3.2	Byte	33
1.3.3	Wort	33
1.4	Kodierung von Zeichen	33
1.4.1	ASCII-Code	33
1.4.2	ANSI-Code	35
1.4.3	Unicode	35
1.5	Kodierung logischer Informationen	36
1.5.1	Und-Funktion	37
1.5.2	Oder-Funktion	38
1.5.3	Nicht-Funktion	39
1.6	Zusammenfassung	39
1.7	Aufgaben	40
1.7.1	Zahlensysteme	40
1.7.2	Informationseinheiten	40
1.7.3	Zeichenkodierung	40
1.7.4	Kodierung logischer Informationen	40

2 Programmiersprachen

43

2.1	Einleitung	43
2.1.1	Verständigungsschwierigkeiten	43
2.1.2	Definition	43
2.1.3	Klassifizierung	44

2.1.4	Geschichte	45
2.2	Programmiersprachen der ersten Generation	46
2.2.1	Programmaufbau	46
2.2.2	Portabilität	47
2.2.3	Ausführungsgeschwindigkeit	48
2.2.4	Einsatzbereich	48
2.3	Programmiersprachen der zweiten Generation	48
2.3.1	Programmaufbau	48
2.3.2	Portabilität	50
2.3.3	Ausführungsgeschwindigkeit	50
2.3.4	Einsatzbereich	50
2.4	Programmiersprachen der dritten Generation	51
2.4.1	Programmaufbau	51
2.4.2	Portabilität	52
2.4.3	Ausführungsgeschwindigkeit	53
2.4.4	Einsatzbereich	53
2.5	Programmiersprachen der vierten Generation	53
2.5.1	Programmaufbau	53
2.5.2	Portabilität	54
2.5.3	Ausführungsgeschwindigkeit	54
2.5.4	Einsatzbereich	54
2.6	Programmiersprachen der fünften Generation	55
2.6.1	Programmaufbau	55
2.6.2	Portabilität	56
2.6.3	Ausführungsgeschwindigkeit	56
2.6.4	Einsatzbereich	56
2.7	Programmiersprachen der sechsten Generation	56
2.7.1	Programmaufbau	56
2.7.2	Portabilität	58
2.7.3	Ausführungsgeschwindigkeit	58
2.7.4	Einsatzbereich	58
2.8	Zusammenfassung	58
2.9	Aufgaben	59
2.9.1	Programmiersprachen der ersten Generation	59
2.9.2	Programmiersprachen der zweiten Generation	59
2.9.3	Programmiersprachen der dritten Generation	59

3 Objektorientierte Programmierung 63

3.1	Einleitung	63
3.1.1	Grundbegriffe	63
3.1.2	Prinzipien	64
3.2	Objekte	64
3.3	Klassen	65

3.3.1	Attribute	65
3.3.2	Methoden	67
3.4	Abstraktion	69
3.5	Vererbung	70
3.5.1	Basisklassen	72
3.5.2	Abgeleitete Klassen	72
3.5.3	Mehrfachvererbung	73
3.6	Kapselung	73
3.7	Beziehungen	74
3.7.1	Beziehungen, die nicht auf Vererbung beruhen	75
3.7.2	Vererbungsbeziehungen	76
3.8	Designfehler	78
3.9	Umstrukturierung	78
3.10	Modellierung	78
3.11	Persistenz	78
3.12	Polymorphie	79
3.12.1	Statische Polymorphie	80
3.12.2	Dynamische Polymorphie	80
3.13	Designregeln	80
3.14	Zusammenfassung	81
3.15	Aufgaben	81
3.15.1	Fragen	81
3.15.2	Übungen	82

Teil 2 Java im Detail

4	Sprache	89
4.1	Einleitung	89
4.1.1	Geschichte	89
4.1.2	Beschreibung mittels Text	89
4.1.3	Überblick über die Sprachelemente	90
4.2	Schlüsselwörter	92
4.3	Einfache Datentypen	93
4.3.1	Grundlagen	93
4.3.2	Festkommazahlen	97
4.3.3	Gleitkommazahlen	100
4.3.4	Wahrheitswerte	101
4.3.5	Zeichen	102
4.4	Erweiterte Datentypen	102
4.4.1	Arrays	102

4.4.2	Aufzählungstyp	105
4.5	Benutzerdefinierte Datentypen	106
4.5.1	Konkrete Klassen	106
4.5.2	Abstrakte Klassen	109
4.5.3	Interfaces	110
4.5.4	Generische Klassen	111
4.6	Variablen	112
4.7	Konstanten	112
4.8	Methoden	112
4.8.1	Methodenarten	113
4.8.2	Konstruktoren	115
4.8.3	Destruktoren	116
4.8.4	Akzessoren	116
4.8.5	Mutatoren	117
4.8.6	Funktionen	117
4.9	Operatoren	118
4.9.1	Arithmetische Operatoren	118
4.9.2	Vergleichende Operatoren	123
4.9.3	Logische Operatoren	126
4.9.4	Bitweise Operatoren	128
4.9.5	Zuweisungsoperatoren	128
4.9.6	Fragezeichenoperator	129
4.9.7	New-Operator	130
4.9.8	Cast-Operator	130
4.10	Ausdrücke	131
4.10.1	Zuweisungen	131
4.10.2	Elementare Anweisungen	133
4.10.3	Verzweigungen	134
4.10.4	Schleifen	135
4.11	Module	138
4.11.1	Klassenimport	138
4.11.2	Namensräume	140
4.12	Dokumentation	141
4.12.1	Zeilenbezogene Kommentare	141
4.12.2	Abschnittsbezogene Kommentare	141
4.12.3	Dokumentationskommentare	141
4.13	Zusammenfassung	142
4.14	Aufgaben	142
4.14.1	Fragen	142
4.14.2	Übungen	143

5 Entwicklungsprozesse 147

5.1	Einleitung	147
5.1.1	Phasen	147
5.1.2	Aktivitäten	148
5.1.3	Werkzeuge	149
5.2	Planungsphase	150
5.2.1	Missverständnisse	150
5.2.2	Anforderungen aufnehmen	150
5.3	Konstruktionsphase	151
5.3.1	Objektorientierte Analyse	151
5.3.2	Objektorientiertes Design	151
5.3.3	Implementierung in Java	152
5.3.4	Test	160
5.4	Betriebsphase	170
5.4.1	Verteilung	170
5.4.2	Pflege	170
5.5	Zusammenfassung	171
5.6	Aufgaben	171
5.6.1	Fragen	171
5.6.2	Übungen	171

6 Plattform 175

6.1	Einleitung	175
6.2	Bytecode	175
6.3	Java Runtime Environment	177
6.3.1	Virtuelle Maschine	178
6.3.2	Garbage Collector	182
6.3.3	Bibliotheken	183
6.3.4	Ressourcen und Property-Dateien	183
6.4	Native Java-Programme	183
6.5	Portabilität eines Java-Programms	185
6.5.1	Binärkompatibler Bytecode	185
6.5.2	Voraussetzungen	187
6.6	Starten eines Java-Programms	188
6.6.1	Application	188
6.6.2	Applet	190
6.6.3	Servlets und JavaServer Pages	191
6.7	Zusammenfassung	191
6.8	Aufgaben	192
6.8.1	Fragen	192
6.8.2	Übungen	192

7 Gesetzmäßigkeiten 195

7.1	Einleitung	195
7.2	Sichtbarkeit	195
7.2.1	Klassenkapselung	195
7.2.2	Gültigkeitsbereich von Variablen	203
7.3	Auswertungsreihenfolge	205
7.3.1	Punkt vor Strich	205
7.3.2	Punkt vor Punkt	206
7.4	Typkonvertierung	208
7.4.1	Implizite Konvertierung	209
7.4.2	Explizite Konvertierung	211
7.5	Polymorphie	213
7.5.1	Überladen von Methoden	213
7.5.2	Überschreiben von Methoden	215
7.6	Programmierkonventionen	218
7.6.1	Vorschriften zur Schreibweise	218
7.6.2	Empfehlungen zur Schreibweise	219
7.7	Zusammenfassung	221
7.7.1	Sichtbarkeit	221
7.7.2	Auswertungsreihenfolge	221
7.7.3	Typkonvertierung	221
7.7.4	Polymorphie	222
7.7.5	Programmierkonventionen	222
7.8	Aufgaben	222
7.8.1	Fragen	222
7.8.2	Übungen	223

8 Klassenbibliotheken 227

8.1	Einleitung	227
8.1.1	Von der Klasse zur Bibliothek	227
8.1.2	Von der Bibliothek zum Universum	228
8.1.3	Vom Universum zum eigenen Programm	228
8.1.4	Bibliotheken und Bücher	228
8.1.5	Bibliotheken erweitern die Sprache	229
8.1.6	Bibliotheken steigern die Produktivität	229
8.1.7	Kommerzielle Klassenbibliotheken	230
8.1.8	Open-Source-Bibliotheken	230
8.1.9	Bibliotheken von Sun Microsystems	230
8.2	Java 2 Standard Edition	230
8.2.1	Java-Language-Bibliothek	231
8.2.2	Klasse System	237
8.2.3	Stream-Bibliotheken	247

8.2.4	Hilfsklassen	249
8.2.5	Abstract Windowing Toolkit	250
8.2.6	Swing	260
8.2.7	JavaBeans	264
8.2.8	Applets	265
8.2.9	Applications	267
8.2.10	Java Database Connectivity (JDBC)	267
8.2.11	Java Native Interface	269
8.2.12	Remote Method Invocation	269
8.3	Java 2 Enterprise Edition	270
8.3.1	Servlets	271
8.3.2	JavaServer Pages	272
8.3.3	CORBA	273
8.3.4	Enterprise JavaBeans	274
8.4	Java 2 Micro Edition	275
8.5	Zusammenfassung	277
8.6	Aufgaben	277
8.6.1	Fragen	277
8.6.2	Übungen	278

9 Algorithmen 281

9.1	Einleitung	281
9.2	Algorithmen entwickeln	281
9.3	Algorithmenarten	282
9.3.1	Sortieren	283
9.3.2	Diagramme	283
9.4	Algorithmen anwenden	288
9.4.1	Sortieren	289
9.4.2	Suchen	290
9.5	Aufgaben	291
9.5.1	Fragen	291
9.5.2	Übungen	291

Teil 3 Größere Java-Projekte

10 Konsolenprogramme 297

10.1	Einleitung	297
10.2	Projekt »Transfer«	297
10.2.1	Anforderungen	297
10.2.2	Analyse und Design	298

10.2.3	Implementierung der Klasse »TransferApp«	300
10.2.4	Implementierung der Klasse »CopyThread«	303
10.2.5	Implementierung der Properties-Datei	307
10.2.6	Test	308
10.2.7	Verteilung	308
10.3	Aufgaben	309
10.3.1	Fragen	309
10.3.2	Übungen	309

11 Erste Schritte mit grafischen Oberflächen 313

11.1	Einleitung	313
11.2	Projekt »Memory«	313
11.2.1	Anforderungen	313
11.2.2	Analyse und Design	315
11.2.3	Implementierung der Klasse »Card«	318
11.2.4	Implementierung der Klasse »CardEvent«	325
11.2.5	Implementierung des Interfaces »CardListener«	326
11.2.6	Implementierung der Klasse »CardBeanInfo«	326
11.2.7	Implementierung des Testtreibers	328
11.2.8	Implementierung der Klasse »GameBoard«	331
11.2.9	Implementierung des Hauptfensters	334
11.2.10	Implementierung der Klasse »AboutDlg«	337
11.2.11	Test	341
11.2.12	Verteilung	342
11.3	Zusammenfassung	342
11.4	Aufgaben	343
11.4.1	Fragen	343
11.4.2	Übungen	343

12 Einfache Oberflächen mit AWT und Swing 347

12.1	Einleitung	347
12.2	Projekt »Abakus«	347
12.2.1	Anforderungen	347
12.2.2	Analyse und Design	349
12.2.3	Implementierung der Applikationsklasse	353
12.2.4	Implementierung des Hauptfensters	354
12.2.5	Implementierung der Klasse »AboutDlg«	370
12.2.6	Zeichen als Unicode codieren	370
12.2.7	Dialog zentriert sich selbst	370
12.3	Zusammenfassung	371
12.4	Aufgaben	372
12.4.1	Fragen	372
12.4.2	Übungen	373

13 Komplexe Oberflächen mit Swing 377

13.1	Einleitung	377
13.2	Projekt »Nestor« – die Oberfläche	377
13.2.1	Anforderungen	377
13.2.2	Analyse und Design	379
13.2.3	Implementierung der Datenbankfassade	383
13.2.4	Implementierung der Applikationsklasse	384
13.2.5	Aufbau des Hauptfensters	385
13.2.6	Implementierung der Adresskomponente	386
13.2.7	Implementierung des Hauptfensters	389
13.2.8	Implementierung des Dialogs »Einstellungen«	395
13.2.9	Test	395
13.2.10	Verteilung	397
13.3	Zusammenfassung	397
13.4	Aufgaben	398
13.4.1	Fragen	398
13.4.2	Übungen	398

14 Weboberflächen mit Servlets 403

14.1	Einleitung	403
14.1.1	Hypertext Markup Language	403
14.1.2	Hypertext-Transfer-Protokoll	406
14.1.3	Common Gateway Interface	408
14.1.4	Servlets	408
14.2	Projekt »Xenia« – die Oberfläche	409
14.2.1	Anforderungen	409
14.2.2	Analyse und Design	411
14.2.3	Implementierung der HTML-Vorlagen	412
14.2.4	Implementierung der Klasse »GuestList«	414
14.2.5	Implementierung der Klasse »NewGuest«	419
14.2.6	Verteilung	425
14.3	Zusammenfassung	425
14.4	Aufgaben	426
14.4.1	Fragen	426
14.4.2	Übungen	426

15 Datenbankprogrammierung 429

15.1	Einleitung	429
15.1.1	Vom Modell zum Datenmodell	429
15.1.2	Vom Datenmodell zur Datenbank	429
15.1.3	Von der Datenbank zu den Daten	430

15.1.4	Von den Daten zum Programm	430
15.2	Projekt »Hades«	431
15.2.1	Anforderungen	431
15.2.2	Analyse & Design	431
15.2.3	Implementierung	432
15.2.4	Test	433
15.3	Projekt »Charon«	433
15.3.1	Anforderungen	434
15.3.2	Implementierung der Klasse »HadesDb«	435
15.3.3	Implementierung der Klasse »Charon«	438
15.3.4	Implementierung der Klasse »HadesTest«	441
15.3.5	Implementierung der Klasse »CharonTest«	443
15.3.6	Implementierung der Datei »Db.properties«	444
15.3.7	Test	445
15.3.8	Verteilung	446
15.4	Zusammenfassung	446
15.5	Aufgaben	446
15.5.1	Fragen	446
15.5.2	Übungen	446

16 Datenbankanwendungen 449

16.1	Einleitung	449
16.2	Projekt »Perseus«	449
16.2.1	Anforderungen	449
16.2.2	Analyse und Design	450
16.2.3	Implementierung der Klasse »BasisWnd«	453
16.2.4	Implementierung der Klasse »Alignment«	454
16.2.5	Implementierung der Klasse »SplashWnd«	455
16.2.6	Implementierung der Klasse »BasicDlg«	457
16.3	Projekt »Charon«	460
16.3.1	Anforderungen	460
16.3.2	Analyse und Design	460
16.3.3	Implementierung von »HadesDb«	461
16.3.4	Implementierung von »Charon«	461
16.3.5	Test	461
16.3.6	Verteilung	462
16.4	Projekt »Nestor«	462
16.4.1	Integration der Klasse »SplashWnd«	462
16.4.2	Integration der Klasse »SplashWnd«	463
16.4.3	Implementierung der Methode »showSplashScreen«	463
16.4.4	Integration der Klasse »BasicDlg«	464
16.4.5	Integration der Klasse »Charon«	465
16.4.6	Verteilung	465
16.5	Zusammenfassung	466

16.6	Aufgaben	466
16.6.1	Fragen	466
16.6.2	Übungen	466

17 Dynamische Websites 469

17.1	Einleitung	469
17.2	Projekt »Charon«	469
17.2.1	Anforderungen	469
17.2.2	Analyse und Design	469
17.2.3	Implementierung der Klasse »HadesDb«	470
17.2.4	Implementierung der Klasse »Charon«	472
17.3	Projekt »Xenia«	473
17.3.1	Anforderungen	473
17.3.2	Analyse und Design	473
17.3.3	Implementierung der Klasse »NewGuest«	474
17.3.4	Implementierung der Klasse »GuestList«	475
17.3.5	Änderungen am Projektverzeichnis	476
17.3.6	Test	477
17.3.7	Verteilung	479
17.4	Zusammenfassung	479
17.5	Aufgaben	480
17.5.1	Fragen	480
17.5.2	Übungen	480

Teil 4 Lösungen

18 Lösungen zu Teil I 485

18.1	Digitale Informationsverarbeitung	485
18.1.1	Zahlensysteme	485
18.1.2	Informationseinheiten	485
18.1.3	Zeichenkodierung	486
18.1.4	Kodierung logischer Informationen	486
18.2	Programmiersprachen	486
18.2.1	Programmiersprachen der ersten Generation	486
18.2.2	Programmiersprachen der zweiten Generation	487
18.2.3	Programmiersprachen der dritten Generation	487
18.3	Objektorientierte Programmierung	487
18.3.1	Fragen	487
18.3.2	Übungen	488

19 Lösungen zu Teil II 493

19.1	Sprache	493
19.1.1	Fragen	493
19.1.2	Übungen	495
19.2	Entwicklungsprozesse	496
19.2.1	Fragen	496
19.2.2	Übungen	497
19.3	Plattform	498
19.3.1	Fragen	498
19.3.2	Übungen	499
19.4	Gesetzmäßigkeiten	500
19.4.1	Fragen	500
19.4.2	Übungen	501
19.5	Klassenbibliotheken	501
19.5.1	Fragen	501
19.5.2	Übungen	502
19.6	Algorithmen	503
19.6.1	Fragen	503
19.6.2	Übungen	504

20 Lösungen zu Teil III 509

20.1	Konsolenprogramme	509
20.1.1	Fragen	509
20.1.2	Übungen	509
20.2	Erste Schritte mit grafischen Oberflächen	510
20.2.1	Fragen	510
20.2.2	Übungen	511
20.3	Einfache Oberflächen mit AWT und Swing	511
20.3.1	Fragen	511
20.3.2	Übungen	512
20.4	Komplexe Oberflächen mit Swing	513
20.4.1	Fragen	513
20.4.2	Übungen	514
20.5	Weboberflächen mit Servlets	515
20.5.1	Fragen	515
20.5.2	Übungen	515
20.6	Datenbankprogrammierung	516
20.6.1	Fragen	516
20.6.2	Übungen	516
20.7	Datenbankanwendungen	517

20.7.1	Fragen	517
20.7.2	Übungen	517
20.8	Dynamische Websites	518
20.8.1	Fragen	518
20.8.2	Übungen	518

Teil 5 Anhang

21	Werkzeuge	525
21.1	Einleitung	525
21.1.1	Einzelwerkzeuge versus Werkzeugsuiten	525
21.1.2	Zielgruppen	526
21.2	Kriterien zur Werkzeugauswahl	527
21.2.1	Allgemeine Kriterien	528
21.2.2	Projektverwaltung	531
21.2.3	Modellierungswerkzeuge	531
21.2.4	Texteditor	533
21.2.5	Java-Compiler	534
21.2.6	Java-Decompiler	535
21.2.7	GUI-Builder	535
21.2.8	Laufzeitumgebung	536
21.2.9	Java-Debugger	537
21.2.10	Werkzeuge zur Verteilung	538
21.2.11	Wizards	539
21.3	Einzelwerkzeuge	539
21.3.1	Modellierungswerkzeuge	539
21.3.2	Texteditor	540
21.3.3	Java-Compiler	541
21.3.4	Java-Decompiler	542
21.3.5	GUI-Builder	542
21.3.6	Laufzeitumgebungen	543
21.3.7	Java-Debugger	544
21.3.8	Versionskontrollwerkzeuge	544
21.3.9	Werkzeuge zur Verteilung	545
21.4	Werkzeugsuiten	545
21.4.1	Eclipse	546
21.4.2	JBuiler	548
21.4.3	Java Development Kit	549
21.4.4	NetBeans	555
21.4.5	Rational XDE	556
21.4.6	Sun One Studio	556
21.4.7	Together	556
21.4.8	VisualAge Java	557

21.4.9	WebSphere Studio	558
--------	------------------------	-----

22 Computerhardware 561

22.1	Einleitung	561
22.2	Aufbau eines Computers	561
22.3	Bussystem	561
22.4	Prozessoren	562
22.4.1	Central Processing Unit	562
22.4.2	Grafikprozessor	563
22.5	Speichermedien	563
22.5.1	Hauptspeicher	563
22.5.2	Festplattenspeicher	564
22.6	Ein- und Ausgabesteuerung	564
22.7	Taktgeber	565
22.8	Zusammenfassung	565

23 Glossar 569

24 Literatur 579

24.1	Basiswissen	579
24.2	Java im Detail	579
24.3	Größere Java-Projekte	580
24.4	Anhang	581

Index 583

2 Programmiersprachen

*»In keiner Sprache kann man sich so schwer verständigen
wie in der Sprache.«
(Karl Kraus)*

2.1 Einleitung

Dieses Kapitel gibt Ihnen einen Überblick über die babylonische Vielfalt der Programmiersprachen. Es hilft Ihnen, die Programmiersprache Java in den nachfolgenden Kapiteln besser einzuordnen, die Entwicklung der Sprache besser nachzuvollziehen und ihre Konzepte besser zu verstehen. Ab diesem Kapitel sollten Sie eine Java-Entwicklungsumgebung (zum Beispiel Eclipse) installiert haben, um das erste Beispiel gleich nachzuvollziehen zu können. Hilfe bei der Auswahl und Installation von Java-Entwicklungsumgebungen finden Sie in Kapitel 21, »Werkzeuge«.

2.1.1 Verständigungsschwierigkeiten

In Kapitel 1, »Digitale Informationsverarbeitung« haben Sie erfahren, dass ein Digitalcomputer Informationen auf sehr primitive Art darstellt. Vielleicht haben Sie sich gefragt, wie eine so dumme Maschine in der Lage ist, vom Menschen entwickelte intelligente Programme auszuführen. Das ist in der Tat nicht einfach.

Zwischen dem Menschen und dem Computer gibt es enorme Verständigungsschwierigkeiten, da sich die menschliche Sprache und die Maschinensprache des Computers stark unterscheiden. Es hat einige Jahrzehnte gedauert, die Verständigungsschwierigkeiten halbwegs aus dem Weg zu räumen. Der Schlüssel dazu liegt in der Entwicklung geeigneter Programmiersprachen.

2.1.2 Definition

Programmiersprachen sind Sprachen, mit deren Hilfe ein Softwareentwickler Befehle (Rechenvorschriften) für den Computer formuliert. Eine bestimmte Ansammlung von Befehlen ergibt ein Computerprogramm. Die Befehle dieser Programmiersprachen sind nicht so leicht verständlich, wie es die natürliche Sprache für uns ist. Diese Sprachen können aber vom Menschen viel besser verstanden werden als der Binärcode des Computers. Programmiersprachen vermitteln also

zwischen beiden Welten im Kreislauf zwischen Mensch und Maschine (Abbildung 2.1).

Damit Sie eine Programmiersprache wie Java verstehen, müssen Sie diese Sprache wie jede Fremdsprache erlernen. Der Computer hat es besser: Er muss die Fremdsprache Java nicht erlernen. Für ihn haben findige Softwareentwickler eine Art Dolmetscher (Interpreter, Compiler) erfunden. Dieser *Babelfisch*¹ übersetzt die Java-Sprache in die Muttersprache des Computers (Kapitel 5, »Entwicklungsprozesse«, und 6, »Plattform«, stellen den Compiler ausführlich vor).

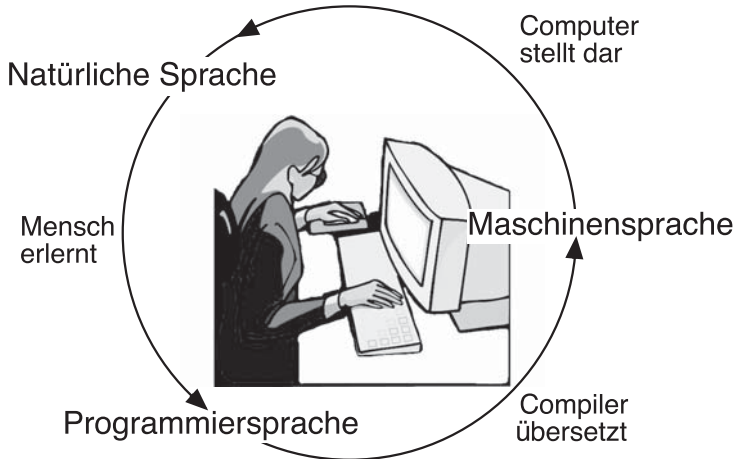


Abbildung 2.1 Kreislauf zwischen Mensch und Maschine

2.1.3 Klassifizierung

Es gibt verschiedene Möglichkeiten, Programmiersprachen einzuordnen: entweder nach Sprachmerkmalen (Abbildung 2.2 auf der nächsten Seite) oder nach ihrer Abstammung (Abbildung 2.3 auf Seite 46) oder chronologisch (Abschnitt 2.1.4, »Geschichte«).

Peter Rechenberg sagt, ein einziges Klassifikationsschema sei niemals ausreichend, und schlägt stattdessen gleich zwei verschiedene Schemata vor (Abbildung 2.2 auf der nächsten Seite). Dieses Kapitel gruppiert die Programmiersprachen chronologisch und beginnt daher mit ihrer Geschichte.

1 Aus Douglas Adams, »Per Anhalter durch die Galaxis«: Ein Babelfisch ist ein Fisch, den man sich ins Ohr steckt und der per Gedankenübertragung alle Sprachen übersetzt.

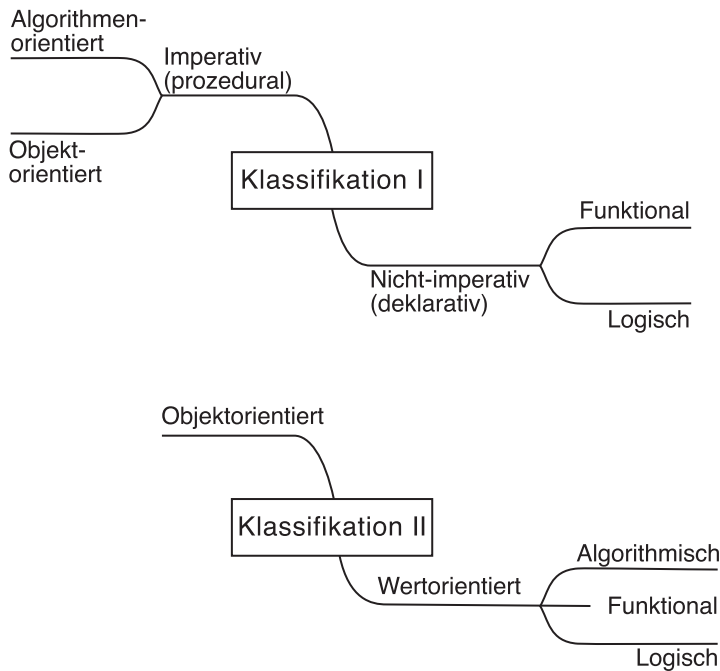


Abbildung 2.2 Klassifikation nach Rechenberg

2.1.4 Geschichte

Programmiersprachen unterliegen einem steten Wandel. Ständig kommen neue Sprachen hinzu, alte verschwinden wieder. Der Grund für diese hektische Betriebsamkeit ist die Suche der Softwareentwickler nach *der* optimalen Programmiersprache.

Die Idealsprache ist extrem leicht zu erlernen, für jeden Einsatzbereich geeignet und befähigt die Entwicklung hochwertiger, extrem schneller Software, die auf jedem Computersystem ausgeführt werden kann – kurz: diese Sprache gibt es (noch) nicht.

Auch wenn die optimale Programmiersprache noch nicht existiert, ist der bisher erzielte Fortschritt bei der Entwicklung neuer Programmiersprachen beachtlich. Ausgangspunkt dieser Entwicklung war die »Muttersprache« der Computer, die so genannte Maschinensprache (Abbildung 2.3).

Von der maschinennahen Programmierung hat man sich jedoch im Laufe der Zeit immer weiter entfernt. Ordnet man die Programmiersprachen chronologisch, so kommt man heute je nach Zählweise auf bis zu sechs Generationen von Programmiersprachen, die ich Ihnen vorstellen möchte.

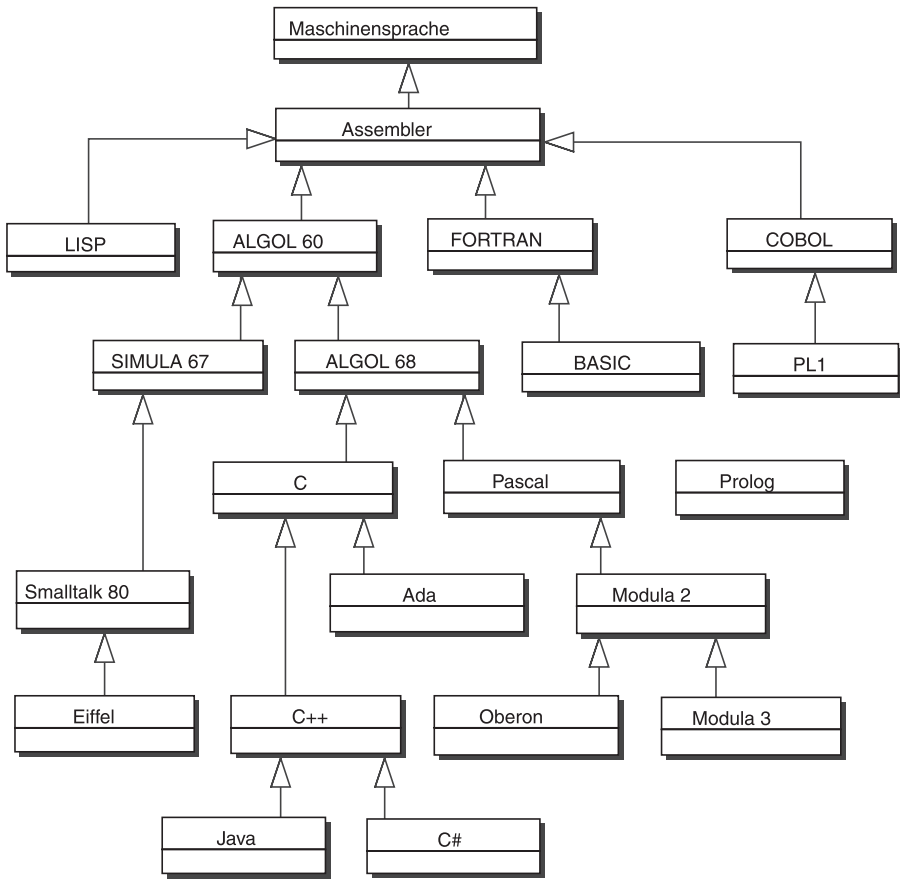


Abbildung 2.3 Stammbaum der wichtigsten Programmiersprachen

2.2 Programmiersprachen der ersten Generation

Als die ersten Computer entwickelt wurden, programmierte man sie direkt in Maschinensprache. Die »Muttersprache« des Computers nennt sich Maschinensprache, weil sie vom Computer (der Maschine) direkt und ohne Übersetzung ausgeführt werden kann.

Die Maschinensprache ist sehr verschieden von den natürlichen Sprachen, mit denen sich Menschen verständigen (Humansprachen). Sie besteht aus kleinen Codeeinheiten im *Binärformat*, den eingangs erwähnten Befehlen.

2.2.1 Programmaufbau

Wenn ein Maschinenprogramm abläuft, liest der Computer diese binären Zahlen und interpretiert sie als Befehle. Das Programm befindet sich während seiner Ausführung in einem bestimmten Bereich des Hauptspeichers (Kapitel 22, »Com-

puterhardware«). Somit kann jedem Befehl eine eindeutige Adresse zugeordnet werden. Das Listing 2.1 zeigt auf der linken Seite die Adresse im Hauptspeicher in hexadezimaler Notation und auf der rechten Seite das eigentliche Maschinenprogramm in hexadezimaler Notation.

Listing 2.1 Ausschnitt aus einem Maschinenprogramm (Intel-80x86-CPU)

```
XXXX:0100  B9  01  00
XXXX:0103  B8  00  00
XXXX:0106  01  C8
XXXX:0108  41
XXXX:0109  83  F9  05
XXXX:010C  76  F8
```

Verständlichkeit

Durch die hexadezimale Notation des Maschinenprogramms ist es einem Experten schon viel besser möglich, das Programm zu verstehen – besonders gut lesbar ist die Ansammlung mehr oder weniger verständlicher Anweisungen jedoch nicht.

Mikrobefehle

Das Maschinenprogramm besteht aus einer Reihe von Mikrobefehlen, auf die ich an dieser Stelle bewusst nicht weiter eingehen möchte. Die Erläuterung der Befehle folgt im nächsten Abschnitt, wo das gleiche Programm in der Assembler-Sprache vorgestellt wird.

Binärcode

Nur so viel an dieser Stelle: Was Sie im Listing 2.1 sehen, ist der so genannte *Binärcode* eines Computerprogramms. Ein binäres Computerprogramm besteht aus Befehlen in der nativen (eigentlichen) Sprache des Computers. Wenn man ein Computerprogramm direkt in dieser Maschinensprache schreibt, muss es nicht mehr in die Sprache des Computers übersetzt werden.

2.2.2 Portabilität

Es ist wichtig zu wissen, dass sich Computer unterschiedlicher Bauart mehr oder weniger stark in ihrem Binärcode unterscheiden. Ein Maschinenprogramm für einen Apple Macintosh G5 von einem Maschinenprogramm für einen Intel-PC oder einem IBM-Großrechner. Durch diese Tatsache kann ein Maschinenprogramm, das für einen Intel-PC entwickelt wurde, nicht direkt auf einem anderen Computersystem wie einem IBM-Großrechner ausgeführt werden.

Um das zu erreichen, muss das Computerprogramm von einer Maschinensprache in die andere übertragen (portiert) werden (engl. portable: übertragbar). Wenn Sie nochmals einen Blick auf Listing 2.1 werfen, können Sie sich vorstellen, was es bedeutet, Tausende von derart simplen Instruktionen zu übertragen. Ein Entwickler, der diese Arbeit manuell durchführt, muss – neben unendlicher Geduld – über sehr gute Kenntnisse in der Hardware *beider* Computersysteme verfügen.

In Maschinensprache geschriebene Computerprogramme lassen sich ab einer bestimmten Komplexität praktisch nicht mehr auf andere Computersysteme übertragen. Dies ist neben der schlechten Verständlichkeit einer der Hauptnachteile der Maschinensprache und einer der Gründe, warum man Hochsprachen entwickelt hat.

2.2.3 Ausführungsgeschwindigkeit

Entwickler von Maschinenprogrammen besitzen in der Regel sehr gute Hardwarekenntnisse und können daher den Programmcode und Speicherplatzbedarf der Programme stark optimieren. Daher sind direkt in Maschinensprache entwickelte Programme meistens sehr effizient programmiert. Sie laufen im Vergleich zu Programmen, die in Hochsprachen (Pascal, Java) programmiert sind, oftmals viel schneller, benötigen nur wenig Hauptspeicher und Festplattenkapazität.

2.2.4 Einsatzbereich

Direkt in Maschinensprache wird trotz ihres Geschwindigkeitsvorteils aufgrund ihrer Hardwareabhängigkeit, ihrer schlechten Portabilität und ihrer extrem schlechten Verständlichkeit heute kaum mehr programmiert. Wenn man überhaupt maschinennah programmiert, dann in Assembler, der Programmiersprache der zweiten Generation.

2.3 Programmiersprachen der zweiten Generation

Um den Computer nicht in der für Menschen schlecht verständlichen Maschinensprache programmieren zu müssen, hat man die Assembler-Sprache erfunden. In Assembler geschriebene Programme bestehen aus einzelnen symbolischen Anweisungen, die sich der Programmierer besser merken kann.

2.3.1 Programmaufbau

Beachten Sie bitte nur den rechten Teil des Programmbeispiels in Listing 2.2, der mit *MOV* beginnt. Der gesamte linke Bereich ist das entsprechende Maschinenprogramm aus Listing 2.1, das ich zum besseren Vergleich nochmals eingefügt habe.

Listing 2.2 Ausschnitt aus einem Assembler-Programm (Intel-80x86-CPU)

```
XXXX:0100  B9 01 00  MOV  CX,  1
XXXX:0103  B8 00 00  MOV  AX,  0
XXXX:0106  01 C8      ADD  AX,  CX
XXXX:0108  41        INC  CX
XXXX:0109  83 F9 05  CMP  CX,  05
XXXX:010C  76 F8      JBE  106
```

Verständlichkeit

Das Assembler-Programm besteht im Gegensatz zu den kryptischen Zahlencodes des Maschinenprogramms aus symbolischen Befehlen. Ein Computer verfügt über mindestens einen Hauptprozessor, die so genannte Central Processing Unit (CPU). Dieser Prozessor besitzt einen typspezifischen Befehlssatz und mehrere Register (Kapitel 22, »Computerhardware«).

Mikrobefehle

Die Register dienen beim Ausführen des Programms als kurzfristiger Zwischenspeicher für Zahlenwerte, mit denen der Hauptprozessor beschäftigt ist: Sie besitzen daher die extrem wichtige Funktion eines Kurzzeitgedächtnisses für den Prozessor.

Zu Anfang des Programms (Listing 2.2) lädt der Prozessor den Wert 1 in das Register *CX*. Der entsprechende Assembler-Befehl ist ein sogenannter Datentransferbefehl. Er lautet *MOV* und ist eine Abkürzung von »to move« (bewegen). Jeder dieser Mikrobefehle ist ein solches Kürzel, das man sich leicht merken kann und das deshalb auch Mnemonik (Stütze fürs Gedächtnis) heißt.

Die zweite Anweisung *MOV AX, 0* initialisiert das Akkumulatorregister (*AX*) mit dem Wert 0, um die nachfolgende Berechnung bei 0 zu beginnen. Danach addiert der Prozessor den Wert im Zählerregister *CX* (counter register) zum Register *AX*. Im Anschluss daran erhöht der Befehl *INC CX* den Anfangswert um 1. Das Mnemonik lautet *INC* und bedeutet »increment« (Zunahme).

Nachfolgend vergleicht der Prozessor den Wert des Registers *CX* mit dem Wert 5 und springt zur Adresse 106, wenn der Wert kleiner oder gleich 5 ist. Die beiden Befehle *CMP* und *JBE* bilden demnach eine Einheit. *CMP* bedeutet »to compare« (vergleichen) und *JBE* »jump below or equal« (springe, wenn kleiner oder gleich).

Binärcode

Was Sie in Listing 2.2 sehen, ist der so genannte *Assembler-Code* eines Computerprogramms. Damit der Computer diesen Programmtext (*ASCII-Code*) verstehen kann, muss er in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. Dazu verwendet der Softwareentwickler ein spezielles Entwicklungswerkzeug, den sogenannten Assembler. Der Assembler *fügt* das Programm *zusammen* (engl. to assemble: zusammenfügen, montieren). Von diesem Werkzeug bekam die Programmiersprache ihren Namen.

2.3.2 Portabilität

Ein Assembler-Programm von einem Computersystem auf ein anderes zu übertragen, ist ähnlich schwer wie die Portierung eines Maschinenprogramms. Meist ist es sinnvoller, sich die Dokumentation durchzulesen und das gesamte Programm neu zu schreiben.

Bedenken Sie, was die Hardwareabhängigkeit von Software bedeutet: Nicht nur, um ein Computerprogramm von einem Computertyp auf einen anderen zu übertragen, muss die Software verändert werden. Sie müsste eigentlich auch dann verändert werden, wenn ein neueres Modell des gleichen Computertyps erscheint, sobald dessen Maschinensprache umfangreicher geworden ist. Wenn Sie ein in Assembler geschriebenes Programm ausliefern, müssten Sie unterschiedliche Versionen für unterschiedliche Computertypen und -modelle produzieren.

Aus diesen genannten Gründen ist der Anteil der Assembler-Programmierung bei komplexen Projekten inzwischen unbedeutend. Es ist einfach unwirtschaftlich, in Assembler zu programmieren.

2.3.3 Ausführungsgeschwindigkeit

Aber egal, wie man zur hardwarenahen Programmierung steht: Da die Assembler-Sprache mit der Maschinensprache sehr verwandt ist, kann ein Assembler-Programm extrem leicht in effizienten Binärcode umgesetzt werden. Es ist kompakt, benötigt also sehr wenig Festplattenspeicherplatz, beansprucht normalerweise wenig Hauptspeicher und kann bei geschickter Programmierung deutlich schneller ausgeführt werden als vergleichbare Hochsprachenprogramme.

2.3.4 Einsatzbereich

Sinnvolle Anwendungsbereiche der Assembler-Sprachen sind dort, wo extreme Anforderungen an die Ausführungsgeschwindigkeit und Kompaktheit des Codes auftreten, zum Beispiel bei Computerspielen, bei Gerätetreibern oder bei geschwindigkeitskritischen Betriebssystemteilen.

2.4 Programmiersprachen der dritten Generation

Da heute aufgrund der genannten Nachteile niemand mehr seinen Computer ausschließlich in Maschinen- oder Assembler-Sprache programmieren möchte, hat man eine Reihe von so genannten höheren Programmiersprachen entwickelt. Deren wichtigste Vertreter sind FORTRAN, COBOL, Algol, Pascal, BASIC, SIMULA, C, C++, Java und C#.

Die Programmiersprachen der dritten Generation stehen zwischen der unverständlichen, aber extrem effizienten Maschinensprache und der für den Menschen optimal verständlichen, aber aus Maschinensicht ineffizienten und unpräzisen natürlichen Sprache.

Der Übergang von der Assembler-Sprache zu den Programmiersprachen der dritten Generation kommt einem Quantensprung gleich. Die neue Generation unterstützt die Umsetzung von Algorithmen (Kapitel 9, »Algorithmen«) viel besser als die Assembler-Sprachen und besitzt nicht deren extreme Hardwareabhängigkeit.

Obwohl es heute Sprachen der fünften Generation gibt, dominieren die Programmiersprachen der dritten Generation die Welt der Softwareentwicklung. Sie bieten einen guten Kompromiss zwischen der Flexibilität der Assembler-Sprache und der Mächtigkeit der Sprachen der fünften Generation.

2.4.1 Programmaufbau

Programme, die in einer höheren Programmiersprache geschrieben wurden, gleichen sich prinzipiell im Aufbau. Sie verfügen über eine Deklaration von Datenstrukturen, über Funktionen und Kontrollstrukturen. Ein Beispiel zeigt das Java-Programm in Listing 2.3.

Listing 2.3 Ein Java-Programm

```
// CD/examples/ch02/ex03
class Addition {
    public static void main(String[] arguments) {
        // Anfangswert setzen:
        int i = 0;
        // Schleife:
        while (i <= 5) {
            i++; // Zaehler erhoehen
        }
        // Summe ausgeben:
        System.out.println("Summe = " + i);
    }
}
```

Verständlichkeit

Das kleine Programm leistet praktisch das Gleiche wie das Assembler-Programm zuvor, ist aber sicher selbst von jedem Informatik-Laien weit besser zu verstehen. Das liegt zum Teil daran, dass sich die Java-Programmiersprache sehr an die Bezeichnungen der Mathematik anlehnt und natürliche Begriffe als Schlüsselwörter (*class*, *main*, *while*) verwendet.

Makrobefehle

Wenn Sie die Assembler-Programme mit Java-Programmen vergleichen, stellen Sie fest, dass ein Java-Befehl in der Regel weit mächtiger ist als ein Assembler-Befehl. Mit anderen Worten: Sie müssen nicht so viel schreiben und kommen bei der Programmierung schneller zum Ziel.

Binärcode

Damit der Computer den *Java-Code* (ASCII-Text) ausführen kann, muss dieser in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. An dieser Stelle soll als Erklärung genügen, dass Sie hierfür ein Programm namens Compiler benötigen. Der Compiler *stellt* übersetzt das Java-Programm in ein Binärprogramm *zusammen* (lat. *compilare*: zusammenraffen, plündern).

Die meisten Entwicklungsumgebungen für Sprachen der dritten Generation verwenden Compiler. Statt eines Compilers lässt sich aber auch ein Interpreter einsetzen, um das Programm Schritt für Schritt in die Maschinsprache zu übertragen. Java kombiniert beide Verfahren. Wenn Sie das Programm ausführen wollen, wechseln Sie in das Verzeichnis, wo sich die Beispiele dieses Buchs befinden. Danach suchen Sie in das Unterverzeichnis `ch02/ex03/bin` und geben auf der Kommandozeile ein:

```
java Addition
```

Das Programm wird nun ausgeführt und sollte folgendes Ergebnis ausgeben:

```
Summe = 6
```

Ich würde an dieser Stelle zu weit führen, das ganze Verfahren der Herstellung und Ausführung eines Java-Programms genau zu erklären. Dafür ist das Kapitel 6, »Plattform«, reserviert.

2.4.2 Portabilität

Neben der besseren Verständlichkeit und höheren Produktivität besitzt das vorliegende Java-Programm gegenüber dem vorangehenden Assembler-Beispiel einen weiteren entscheidenden Vorteil: Es ist nicht abhängig von einer bestimmte

Hardware, sondern sehr leicht von einem Computersystem auf ein anderes zu übertragen.

Das Merkmal der leichten Portabilität trifft auf alle Programmiersprachen der dritten Generation zu – jedoch im unterschiedlichen Ausmaß. Es gibt zum Beispiel zwischen Sprachen wie C++ und Java einige deutliche Unterschiede: Bei C++ ist nur der Quelltext weitestgehend portabel, bei Java hingegen auch der Binärcode.

Java gehört zu den Programmiersprachen, deren Programme am leichtesten portierbar sind. Der vom Java-Compiler erzeugte Binärcode (Bytecode: Abschnitt 6.2, »Bytecode«, 175) kann bei Einhaltung von bestimmten Programmierregeln praktisch unverändert sowohl auf einem Macintosh als auch auf einem IBM-Großrechner ausgeführt werden.

2.4.3 Ausführungsgeschwindigkeit

Es gibt heute sehr leistungsfähige Compiler, die aus einem Hochsprachenprogramm ein schnelles Maschinenprogramm erzeugen. Trotzdem ist es im Regelfall so, dass ein optimales, in Assembler geschriebenes Programm schneller ausgeführt wird als ein optimales Hochsprachenprogramm. Dieser Unterschied rechtfertigt heute jedoch in den meisten Fällen nicht mehr den Einsatz der Assembler-Sprache.

2.4.4 Einsatzbereich

Programmiersprachen der dritten Generation sind Allzweckprogrammiersprachen, die auf allen Gebieten der Softwareentwicklung verwendet werden. Mittlerweile verdrängen sie die Assembler-Sprache sogar auf dem Gebiet der Treiberprogrammierung.

2.5 Programmiersprachen der vierten Generation

Mit den Programmiersprachen der vierten Generation versuchten die Entwickler Probleme wie den Datenbankzugriff in einer abstrakteren Art und Weise zu lösen als mit den Sprachen der dritten Generation. Ein Beispiel für eine Programmiersprache dieser Generation ist Natural.

2.5.1 Programmaufbau

Natural-Programme bestehen wie Programme, die mit Sprachen der dritten Generation geschrieben wurden, aus Datendeklarationen, Kontrollstrukturen und Funktionen. Ein Beispiel sehen Sie in Listing 2.4.

Listing 2.4 Ein Natural-Programmbeispiel

```
PGM-ID: Addition
DEFINE DATA
  LOCAL
    01 #i
END-DEFINE
* -----
FOR #i 1 TO 5 STEP 1
END-FOR
```

Verständlichkeit

Wenn Sie dieses Beispiel mit dem eingangs gezeigten Assembler-Listing vergleichen, erkennen Sie ebenfalls, dass es erheblich besser zu verstehen ist.

Makrobefehle

Natural-Programme verfügen über Befehle, die im Vergleich zu Sprachen der dritten Generation in der Regel weit mächtiger sind. Die Sprache ist im Vergleich zu C++ oder Java weit weniger flexibel, aber in bestimmten Einsatzbereichen sicher annähernd so produktiv.

Binärcode

Um *Natural-Code* in ein binäres Computerprogramm (*Binärcode*) zu übersetzen, kommt entweder ein Compiler oder ein Interpreter zum Einsatz.

2.5.2 Portabilität

Wie das Java-Beispiel ist auch das Natural-Programm im Vergleich zu Assembler-Programmen leicht portierbar, da es keine direkten Abhängigkeiten zu der zugrunde liegenden Hardware besitzt.

2.5.3 Ausführungsgeschwindigkeit

Mir sind hier keine vergleichenden Studien bekannt, die auf Unterschiede in der Ausführungsgeschwindigkeit zwischen Natural- und Assembler-Programmen hinweisen. Da Programmiersprachen wie Natural vor allem in Zusammenhang mit der Datenbankprogrammierung eingesetzt werden, kann man davon ausgehen, dass die Ausführungsgeschwindigkeit zufriedenstellend ist.

2.5.4 Einsatzbereich

Programmiersprachen wie Natural haben ihren Haupteinsatzbereich in der Datenbankprogrammierung.

2.6 Programmiersprachen der fünften Generation

Noch weiter entfernt von der Maschinensprache als Natural sind Programmiersprachen der fünften Generation. Sie wurden konzipiert, um Expertensysteme zu entwickeln. Programmiersprachen dieser Generation nennt man auch logische Programmiersprachen. Prominentester Vertreter dieser Gattung ist neben Datalog die Sprache Prolog (**P**rogramming in **l**ogic).

2.6.1 Programmaufbau

Ein Prolog-Programm besteht aus einer Reihe von Funktionen, deren Reihenfolge egal ist. Die Funktionen bestehen aus Sätzen (*clauses*). Bei diesen ist die Reihenfolge sehr wichtig. Es gibt zwei Typen von clauses: Tatsachen (*facts*) und Regeln (*rules*). Ein Beispiel sehen Sie in Listing 2.5.

Listing 2.5 Ein Prolog-Programmbeispiel

```
% Wir legen fest, dass Peter ein Mann ist:
man (peter).
%
% Wir legen weiter fest, dass Peter ein Elternteil
% von Paul ist. Das geschieht nach dem Muster:
% parent(Eltern, Kind)
parent (peter, paul).
%
% Wir legen zudem fest, dass jeder Elternteil, der
% ein Mann ist, zugleich auch ein Vater ist:
father (FA, CH):-man (FA), parent (FA, CH).
%
% Nach dem die Randbedingungen klar sind,
% kann man dem System die Frage stellen:
% Wer ist Pauls Vater?
?-father(X, paul).
```

Verständlichkeit

Prolog wirkt auf Programmierer von mathematisch geprägten Sprachen wie Java sehr ungewohnt. Man kann jedoch erkennen, dass die Sprache ideal sein könnte, um Software zu entwickeln, die logische Probleme lösen soll (sogenannte Expertensysteme).

Makrobefehle

Die Befehle von Prolog-Programmen sind sehr mächtig. Wie Natural ist die Sprache nicht sehr flexibel, aber in einem bestimmten Nischenbereich ist der Entwickler damit sehr produktiv.

Binärcode

Auch *Prolog-Code* muss wieder in ein binäres Computerprogramm (*Binärcode*) übersetzt werden. Dazu verwendet man entweder einen Compiler oder einen Interpreter.

2.6.2 Portabilität

Die Sprache Prolog ist wegen ihres starken Abstraktionsgrades von der Hardware wie andere höhere Programmiersprachen prinzipiell leicht zu portieren. Es gibt eine Vielzahl von Compilern für die unterschiedlichsten Computertypen.

2.6.3 Ausführungsgeschwindigkeit

Mir sind keine vergleichenden Studien zwischen in Prolog geschriebenen Programmen und Assembler-Programmen bekannt. Es soll jedoch optimierende Compiler geben, die hocheffizienten Binärcode erzeugen können.

2.6.4 Einsatzbereich

Logische Programmiersprachen wie Prolog werden vorzugsweise zur Programmierung von Expertensystemen eingesetzt. Expertensysteme sind Programme, die auf künstlicher Intelligenz (KI) aufbauen und logische Schlussfolgerungen ziehen können.

2.7 Programmiersprachen der sechsten Generation

Ähnlich wie der Übergang von der hardwarenahen Assembler-Sprache zu den Hochsprachen soll auch der Übergang zu den Sprachen der sechsten Generation eine Zäsur darstellen: Diese Sprachen beschreiben ein Computerprogramm nicht wie gewohnt durch Text, sondern in Form von Grafiken (zum Beispiel Ablaufdiagrammen). Mit der Version 2.0 der *Unified Modeling Language* (UML) in Verbindung mit der *Model Driven Architecture* (MDA) entwickelt sich zurzeit eine Programmiersprache der sechsten Generation.

2.7.1 Programmaufbau

Von der grafischen Programmiersprache UML kann ich Ihnen kein Listing präsentieren, da die Programme nur grafisch beschrieben werden. Ein Beispiel für einen Ausschnitt eines Programms zeigt Abbildung 2.4.

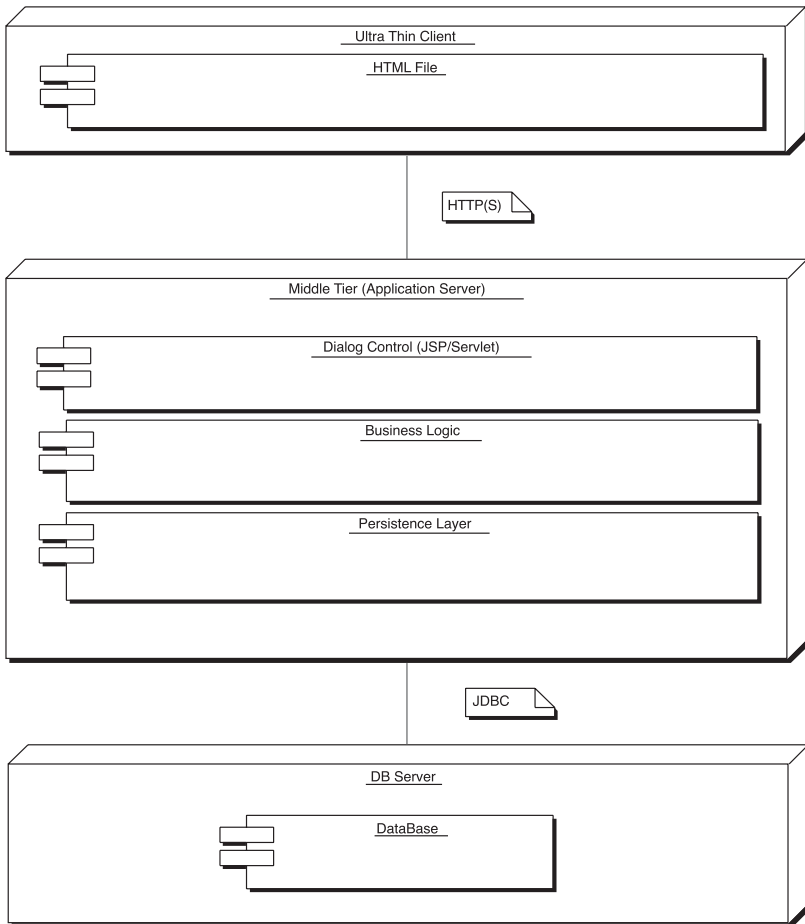


Abbildung 2.4 Eine UML-Beispielgrafik (Verteilungsdiagramm)

Verständlichkeit

Wie Sie aus der Abbildung entnehmen können, ist hier der prinzipielle Aufbau einer Internet-Anwendung skizziert. Die Abbildung zeigt allerdings nicht das vollständige Programm, sondern nur einen Teil davon. Um ein Programm mit der UML vollständig zu beschreiben, ist eine Vielzahl von solchen Grafiken notwendig. Diese sind jedoch sehr gut verständlich.

Grafiken

In Sprachen der sechsten Generation soll es keine Befehle mehr geben, sondern nur eine Vielzahl von Grafiken, die das Modell der Software bilden. Die momentan verfügbaren MDA-Werkzeuge generieren (erzeugen) aus diesen Modellen herkömmlichen Programmcode. Dazu sind in den Werkzeugen Generatoren in-

tegiert. Der von ihnen erzeugte Programmcode ist in einer Sprache der dritten Generation geschrieben.

Binärcode

Der erzeugte *Programmcode* muss letztendlich wieder – direkt oder indirekt – in ein Binärprogramm (*Binärcode*) übersetzt werden. Dazu wird man wahrscheinlich wieder einen Compiler verwenden.

2.7.2 Portabilität

Ein Programm, das nach der Model Driven Architecture entwickelt wurde, soll unabhängig von der Hardware sein und daher leicht von einem Computertyp auf einen anderen übertragen werden können.

2.7.3 Ausführungsgeschwindigkeit

Da aus den Modellen beispielsweise C#- oder Java-Code erzeugt wird, verhält es sich mit der Ausführungsgeschwindigkeit wie bei Programmiersprachen der dritten Generation. Sie ist abhängig vom Geschick des Entwicklers, von der Qualität des Generators und von der Qualität des Compilers oder Interpreters.

2.7.4 Einsatzbereich

Programmiersprachen der sechsten Generation sind gerade in der Entstehung. Ihr Einsatz ist bis auf weiteres Zukunftsmusik, auch wenn es inzwischen einige MDA-Werkzeuge gibt.

2.8 Zusammenfassung

Programmiersprachen haben sich aus der nativen Sprache der Computer (Machinesprache) entwickelt. Sie entfernen sich von der hardwarenahen Programmierung (erste und zweite Generation) und entwickeln sich in Richtung der natürlichen Sprache (Humansprache).

Programmiersprachen der ersten und zweiten Generation eignen sich für hardwarenahe Programme, während Programmiersprachen der vierten und fünften Generation für spezielle Anwendungsfälle, wie Datenbankprogrammierung und Expertensysteme, geeignet sind. Programmiersprachen der dritten Generation beherrschen heute die Programmentwicklung. Zu ihnen gehört auch die Programmiersprache Java.

2.9 Aufgaben

Versuchen Sie, folgende Aufgaben zu lösen:

2.9.1 Programmiersprachen der ersten Generation

1. Wie nennen sich Programmiersprachen der ersten Generation?
2. Woher stammt ihr Name?
3. Weshalb programmiert man heute nicht mehr mit Sprachen der ersten Generation?

2.9.2 Programmiersprachen der zweiten Generation

1. Nennen Sie die drei wichtigsten Vorteile der Assembler-Sprache gegenüber den Hochsprachen.
2. Für welche Software setzt man heute noch die Assembler-Sprache ein?
3. Was sind die drei wesentlichen Vorteile von Hochsprachen gegenüber der Assembler-Sprache?

2.9.3 Programmiersprachen der dritten Generation

1. Was versteht man unter portablen Computerprogrammen?
2. Nennen Sie drei Programmiersprachen der dritten Generation.

Die Lösungen zu den Aufgaben finden Sie in Kapitel 18 ab Seite 486.

3 Objektorientierte Programmierung

*»Ich sehe ein Pferd, dann sehe ich noch ein Pferd – dann noch eins. Die Pferde sind nicht ganz gleich, aber es gibt etwas, das allen Pferden gemeinsam ist, und das, was allen Pferden gemeinsam ist, ist die Form des Pferds. Was unterschiedlich oder individuell ist, gehört zum Stoff des Pferds.«
(Jostein Gaarder)*

3.1 Einleitung

Mitte der 60er-Jahre des letzten Jahrhunderts kam es zu einer Softwarekrise. Die Anforderungen an Programme stiegen, und die Software wurde dadurch komplexer sowie fehlerhafter. Auf Kongressen diskutierten Experten die Ursachen der Krise und die Gründe für die gestiegene Fehlerrate. Ein Teil der Softwareexperten kam zu dem Schluss, dass die Softwarekrise nicht mit den herkömmlichen Programmiersprachen zu bewältigen sei. Sie begannen deshalb, eine Generation von neuen Programmiersprachen zu entwickeln.

Die Entwickler dieser Sprachen kritisierten an den herkömmlichen Programmiersprachen vor allem, dass sich die natürliche Welt bisher nur unzureichend abbilden lasse. Um dem zu entgehen, gingen sie von natürlichen Begriffen aus, wie sie die Formenlehre der klassischen griechischen Philosophie geprägt hat, und wandelten sie für die Programmierung ab. Da sich alles um den Begriff des Objekts dreht, nannten sie die neue Generation von Sprachen »objektorientiert«.

3.1.1 Grundbegriffe

Alan Kay, einer der Erfinder der Programmiersprache *Smalltalk*, hat die Grundbegriffe der objektorientierten Programmierung folgendermaßen zusammengefasst:

- ▶ Alles ist ein Objekt.
- ▶ Objekte kommunizieren durch Nachrichtenaustausch.
- ▶ Objekte haben ihren eigenen Speicher.
- ▶ Jedes Objekt ist ein Exemplar einer Klasse.
- ▶ Die Klasse modelliert das gemeinsame Verhalten ihrer Objekte.
- ▶ Ein Programm wird ausgeführt, indem dem ersten Objekt die Kontrolle übergeben und der Rest als dessen Nachricht behandelt wird.

3.1.2 Prinzipien

Neben diesen Grundbegriffen sind folgende Prinzipien wichtig:

- ▶ Abstraktion
- ▶ Vererbung
- ▶ Kapselung
- ▶ Beziehungen
- ▶ Persistenz
- ▶ Polymorphie

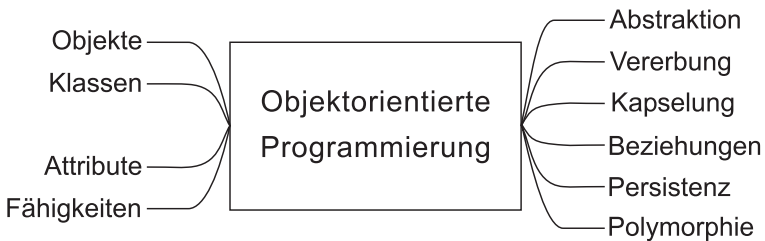


Abbildung 3.1 Hauptbegriffe der objektorientierten Programmierung

3.2 Objekte

Objekte sind für ein Java-Programm das, was Zellen für einen Organismus bedeuten: Aus diesen kleinsten Einheiten setzt sich eine Anwendung zusammen. Objekte haben eine bestimmte Gestalt (Aussehen, Attribute, Kennzeichen) und bestimmte Fähigkeiten (Methoden, Funktionen). Gestalt und Fähigkeiten eines Objekts werden durch seine Erbinformationen bestimmt. Diese Erbinformationen sind der Bauplan, nach dem das Objekt erzeugt wird. In der objektorientierten Programmierung ist der Bauplan eines Objekts seine *Klasse*.

Wenn Sie eine Reihe von Pferden betrachten, fällt Ihnen auf, dass ihnen die prinzipielle *Gestalt* gemeinsam ist. Ihre Unterschiede sind die *Attribute*, die das einzelne Pferd kennzeichnen. Ein Objekt der Klasse *Pferd* ist beispielsweise ein Pferd mit dem Namen *Xanthos*, ein anderes Pferd heißt *Balios*. Beide *Exemplare*¹ haben einen ähnlichen Körperbau (Gestalt) und ähnliche Fähigkeiten. Sie können beispielsweise beide laufen und wiehern.

¹ Exemplar und Objekt sind gleichbedeutend. Im Gegensatz dazu ist der Begriff »Instanz« eine Fehlübersetzung (engl. instance: Exemplar) und taucht in diesem Buch deshalb nicht auf.

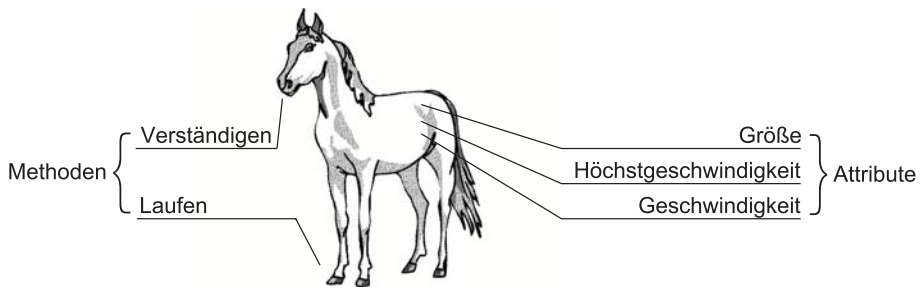


Abbildung 3.2 Jedes Objekt besitzt sein bestimmtes Verhalten und Aussehen.

Die beiden Objekte weisen aber auch einige deutliche Unterschiede auf: *Xanthos* ist weiß und *Balios* braun, und *Xanthos* kann schneller laufen als *Balios*. Obwohl *Xanthos* und *Balios* zur gleichen Klasse gehören, sind nur ihre *prinzipiellen* Fähigkeiten identisch, nicht aber ihre *individuellen* Attribute. Was das bedeutet, wird im nächsten Abschnitt deutlich.

3.3 Klassen

Xanthos und *Balios* gehören zu der Klasse *Pferd*. Die Klasse ist es, die die prinzipielle Gestalt und die Fähigkeiten der beiden Pferde-Objekte festlegt. Man bezeichnet Klassen daher als

- ▶ *Bauplan* für Objekte *oder* als
- ▶ *Oberbegriff* für verschiedene Objekte (Klassifizierung) *oder* als
- ▶ *Schablone* für verschiedene Objekte.

3.3.1 Attribute

Die eingangs erwähnte Klasse *Pferd* soll die Attribute *Größe*, *Höchstgeschwindigkeit* und *Geschwindigkeit* besitzen. Wenn aus dieser Klasse neue Objekte (neue Exemplare) entstehen, besitzen *alle* eine bestimmte *Größe*, eine bestimmte *Höchstgeschwindigkeit* und eine bestimmte *Geschwindigkeit* – aber welche Werte haben diese Attribute? Sie werden erst beim Entstehen (Erzeugen) des Objekts mit den individuellen Werten belegt.

Konstanten

Beispielsweise soll *Xanthos* über folgende Attribute verfügen: *Größe* (Stockmaß) 1,90 m, *Höchstgeschwindigkeit* 65 km/h, *Geschwindigkeit* 0 km/h. *Balios* hingegen soll 1,85 m groß sein, maximal 60 km/h schnell laufen können und momentan gerade 5 km/h laufen.

Obwohl beide Pferde nach dem gleichen Bauplan erzeugt worden sind, sind zwei deutlich unterschiedliche Objekte entstanden: Beide sind unterschiedlich groß, können laufen, aber unterschiedlich schnell, beide besitzen eine Geschwindigkeit, aber ein Pferd steht und das andere bewegt sich langsam.

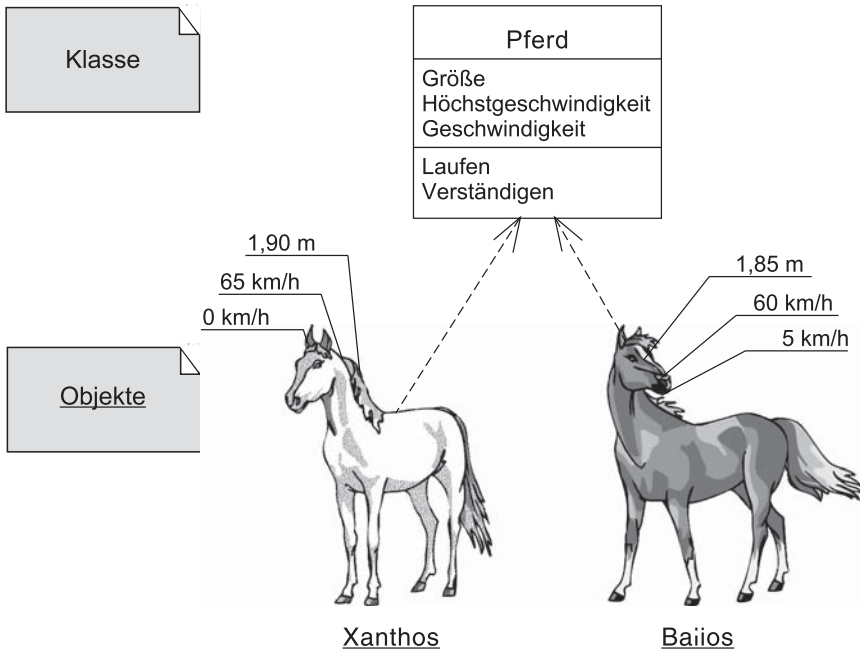


Abbildung 3.3 Die Klasse »Pferd« liefert den Bauplan für Pferde-Objekte.

Zustände

Es ist Ihnen vielleicht aufgefallen, dass bei den bisherigen Attributen der beiden Pferde einige mit festen Werten belegt waren, andere hingegen mit veränderlichen Werten. Die flexiblen Attribute beschreiben den *Zustand* des Objekts. Zum Beispiel beschreibt die *Geschwindigkeit*, wie schnell sich Xanthos gerade bewegt. Der Zustand eines Objekts kann sich im Laufe der Zeit ändern.

Kennungen

Was würde passieren, wenn man Xanthos und Baios so erzeugen würde, dass sie die gleiche *Größe*, die gleiche *Höchstgeschwindigkeit* und die gleiche momentane *Geschwindigkeit* besitzen? Wie könnte man sie dann unterscheiden? In diesem Fall haben beide Objekte zwar individuelle Werte für ihre Attribute bekommen, aber diese sind zufällig gleich. Damit gleichen sich auch die Objekte in einem Programm wie eineiige Zwillinge.

Um die Pferde zu unterscheiden, benötigt man so etwas wie einen genetischen Fingerabdruck. In der Programmierung vergibt der Entwickler eine so genannte Kennung. Diese Kennung ist ein zusätzliches Attribut, bei dem darauf geachtet wird, dass es *eindeutig* ist. Objekte der *gleichen Klasse* besitzen also die gleichen Attribute, aber mit individuellen Werten. Erst die Kennung eines Objekts sorgt dafür, dass das Programm unterschiedliche Exemplare auch dann unterscheiden kann, wenn ihre Attribute zufällig die gleichen Werte besitzen.

3.3.2 Methoden

Angenommen, Sie wollen dem Objekt *Balios* mitteilen, dass es nun springen soll. Im wirklichen Leben geben Sie ihm dazu ein Zeichen. In der objektorientierten Programmierung müssen Sie stattdessen eine Methode des Objekts *Balios* aufrufen. Statt »Methode« werden Sie auch öfter auf die Begriffe »Botschaft« (Smalltalk), »Nachricht« oder »Operation« stoßen, die das Gleiche bedeuten sollen.

Springen



Abbildung 3.4 Objekte verständigen sich durch den Austausch von Nachrichten.

Egal, wie der Begriff nun bei den verschiedenen Programmiersprachen und in der Literatur genannt wird, eines ist gleich: Verhaltensweisen wie *Laufen* und *Verständigen* bestimmen die Fähigkeit eines Objekts zu kommunizieren und Aufgaben zu erledigen. Objekte verständigen sich also über Methoden.

Es existiert nicht nur eine Art von Methoden, sondern es gibt folgende fünf Grundtypen: *Konstruktoren* (»Erbauer«), *Destruktoren* (»Zerstörer«), *Mutatoren* (»Veränderer«), *Akzessoren* (»Zugriffsmethoden«) und *Funktionen* (»Tätigkeiten«).

Konstruktoren

Die wichtigsten Methoden sind die, die ein Objekt erzeugen. Sie werden demzufolge auch Konstruktoren genannt, denn sie konstruieren, das heißt erschaffen ein Objekt.

Destruktoren

Methoden, die ein Objekt zerstören, nennen sich in der objektorientierten Programmierung Destruktoren. In Programmiersprachen wie C++ können Sie diese Destruktoren auch aufrufen und damit unmittelbar ein Objekt zerstören. In Java hat man hingegen aus Sicherheitsgründen darauf verzichtet, Destruktoren direkt aufzurufen. Hier wird ein Objekt automatisch zerstört, wenn es nicht mehr benötigt wird.

Mutatoren

Methoden, die den Wert eines Attributs verändern, nennen sich Mutatoren. Sie verändern den Zustand des Objekts. Mit einer solchen Methode kann ein Reiter namens *Achilles* die Geschwindigkeit des Pferds *Xanthos* ändern (Abbildung 3.5). Die entsprechende Methode nennt sich *Laufen* und verfügt über einen so genannten Parameter, der den neuen Zustand, die *Geschwindigkeit* des Pferds, vorgibt.

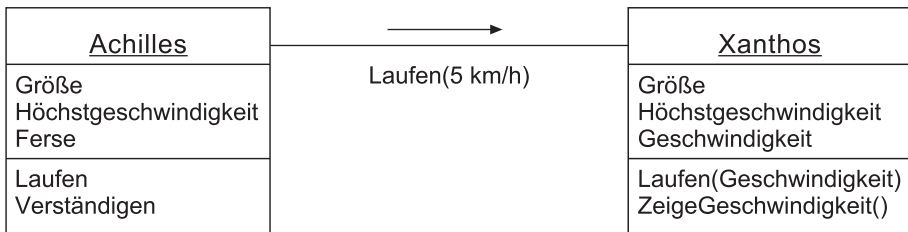


Abbildung 3.5 Die Methode »Laufen« verändert den Zustand von Xanthos.

Akzessoren

Akzessoren sind Zugriffsmethoden, die nur ein bestimmtes Attribut abfragen, ohne etwas am Zustand des Objekts zu ändern. Eine solche Methode wäre zum Beispiel die Abfrage der momentanen Geschwindigkeit des Pferds *Xanthos* (Abbildung 3.6 auf der nächsten Seite). Diese Methode besitzt einen sogenannten Rückgabewert: die aktuelle Geschwindigkeit, mit der sich *Xanthos* fortbewegt.

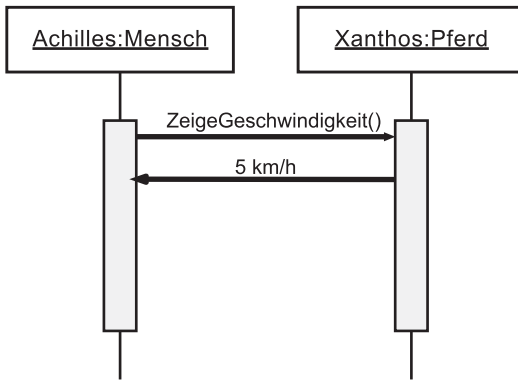


Abbildung 3.6 Die Methode »ZeigeGeschwindigkeit« gibt den Zustand (die momentane Geschwindigkeit) von Xanthos zurück.

Funktionen

Methoden, die zum Beispiel nur eine Rechenoperation durchführen, werden häufig auch in der objektorientierten Programmierung als Funktionen bezeichnet. Sie dürfen trotzdem nicht mit den Funktionen der klassischen Programmiersprachen verwechselt werden, denn es besteht zumindest ein erheblicher Unterschied: sie werden wie andere Methoden auch von Klasse zu Klasse weitervererbt (Abschnitt 3.5, »Vererbung«).

3.4 Abstraktion

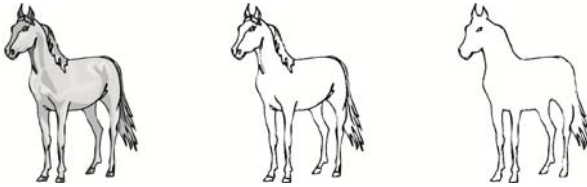
Vielleicht werden Sie jetzt sagen: »Das ist doch alles Unsinn. Die Fähigkeiten und Attribute eines Pferds sind viel komplexer und können nicht auf Größe und Farbe, auf Laufen und Wiehern reduziert werden.« Das ist in der natürlichen Welt richtig, aber in der künstlichen Welt der Softwareentwicklung in der Regel völlig falsch.

Richtig wäre es nur dann, wenn man die Natur in einem Programm vollständig abbilden müsste. Aber für so eine übertriebene Genauigkeit gibt es bei der Programmierung selten einen Grund. Die objektorientierte Programmierung erleichtert eine möglichst natürliche Abbildung der realen Welt und fördert damit gutes Softwaredesign.

Sie)) verführt damit auch zu übertriebenen Konstruktionen. Die Kunst besteht darin, dem entgegen zu steuern und die Wirklichkeit so genau wie nötig, aber so einfach wie möglich abzubilden. Wie Sie später bei größeren Beispielpogrammen sehen werden, bereitet gerade die Analyse der für das Programm wesentlichen und richtigen Bestandteile unter Umständen große Probleme.

Wenn man innerhalb eines Programms nur die für die Funktionalität wesentlichen Teile programmiert, dann hat das praktische Gründe: Das Programm lässt sich schneller entwickeln, es wird billiger und schlanker. Somit benötigt es weniger Speicherplatz, und es wird in der Regel schneller ablaufen als ein Programm, das mit unnötigen Informationen überfrachtet ist.

Um diese Kompaktheit zu erreichen, ist es notwendig, die meist extrem komplizierten natürlichen Objekte und deren Beziehungen so weit es geht zu abstrahieren, also zu vereinfachen. Der Fachbegriff für diese Technik nennt sich demzufolge auch *Abstraktion* (Abbildung 3.7).



—————▶ **Abstraktion**

Abbildung 3.7 Durch Abstraktion erhält man das Wesentliche einer Klasse.

3.5 Vererbung

Nach der Einführung von Klassen, Objekten, Methoden und Attributen ist es an der Zeit, diese neuen Begriffe in den Zusammenhang mit dem Begriff der *Vererbung* zu stellen. Vererbung gestattet es, Verhalten zwischen Klassen und damit auch zwischen Objekten mit Hilfe des Bauplans zu übertragen.

Ein Beispiel: Pferd und Zebra sind eng verwandt (Abbildung 3.8), in mancherlei Hinsicht aber doch sehr verschieden. Diese Unterschiede sind von anderer Güte als die Unterschiede zwischen zwei Pferden: Pferde und Zebras haben eine deutlich unterschiedliche Gestalt.

Dass Zebras im Sinne der Formenlehre eine andere Gestalt besitzen als Pferde, wird deutlich, wenn Sie überlegen, welche Farbe man einem Zebra zuordnen müsste: Schwarz oder Weiß? Zebras haben alle verschiedene Muster. Die Muster unterscheiden sich wie Fingerabdrücke beim Menschen. Das Muster des Fells ist eines der Merkmale, die ein Zebra von einem Pferd unterscheiden (es gibt noch andere).

Es ist also in den Fällen, in denen es auf die Unterschiede zwischen Farbe und Muster ankommt, immer besser, einem Zebra die Eigenschaft *Muster* zu geben und es einer anderen Klasse zuzuordnen (Abbildung 3.9). In allen anderen Fällen genügt eine gemeinsame Klasse.

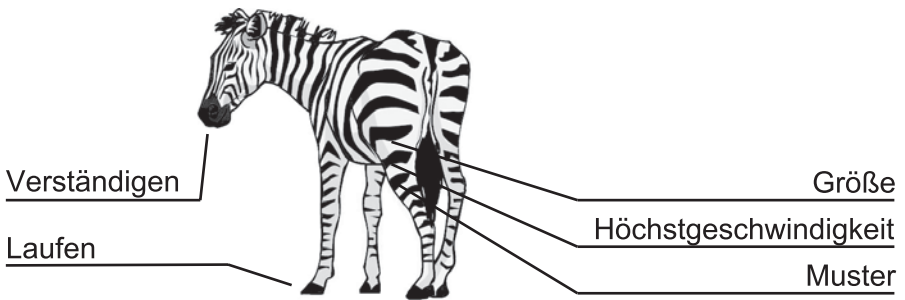
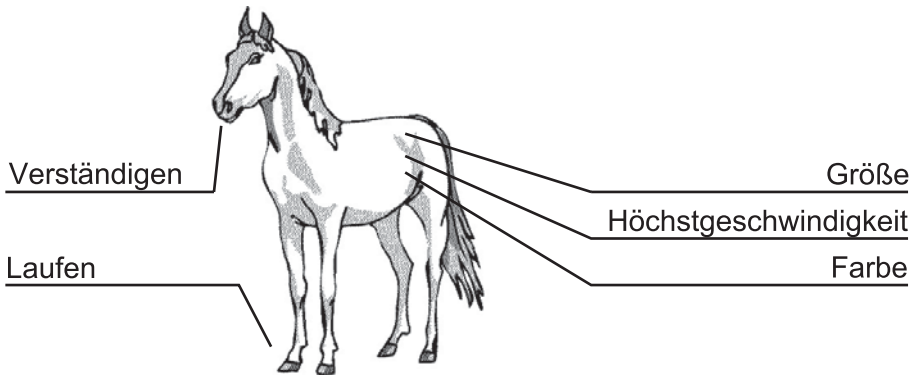


Abbildung 3.8 Objekte verschiedener Klassen unterscheiden sich in ihrer Form.

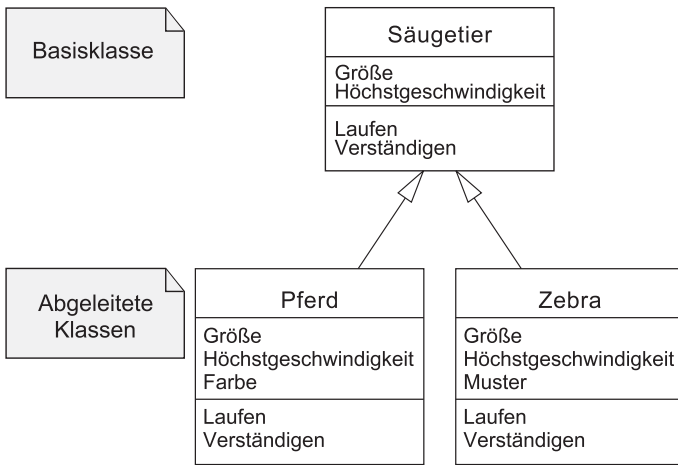


Abbildung 3.9 Die Basisklasse überträgt Basis-Eigenschaften und -Verhalten.

3.5.1 Basisklassen

Was ist Pferd und Zebra gemeinsam? Jedes Zebra und jedes Pferd haben eine bestimmte Größe, ein bestimmtes Gewicht, sie können laufen und sich verständigen. Die genannten Eigenschaften teilen sie mit einer Vielzahl von Tieren. Biologisch gesehen, gehören Pferd und Zebra zu den Säugetieren – was liegt also näher, sie auch dieser Klasse zuzuordnen?

Damit man sich nicht für jede der Klassen *Pferd* und *Zebra* das Verhalten *Laufen* neu ausdenken muss, bietet es sich an, dieses Verhalten und die Attribute *Größe* und *Gewicht* in eine Basisklasse zu verlagern. Wie sieht es mit der Verständigung aus? Wieherzt ein Zebra? – Wohl kaum, die Methode sollte deshalb besser allgemein *Verständigen* genannt werden.

Die neue Basisklasse erleichtert die Erschaffung neuer Säugetier-Klassen, da wesentliche Attribute und ein Teil der Methoden schon fertig vorliegen. An die neue Basisklasse werden aber auch große Ansprüche gestellt, denn Fehler in dieser Klasse rächen sich bei den Klassen, die man ableitet, wie Sie gleich sehen werden.

3.5.2 Abgeleitete Klassen

Angenommen, Sie möchten gern eine neue Klasse namens *Muli* auf Basis der Klasse *Säugetier* erzeugen. In der objektorientierten Programmierung spricht man davon, von *Säugetier* eine neue Klasse namens *Muli* abzuleiten. Die neue Klasse *Muli* erbt wie schon zuvor Pferd und Zebra das Verhalten und die Attribute *Größe* und *Gewicht* der Basisklasse *Säugetier*. Sie stammt von *Säugetier* ab.

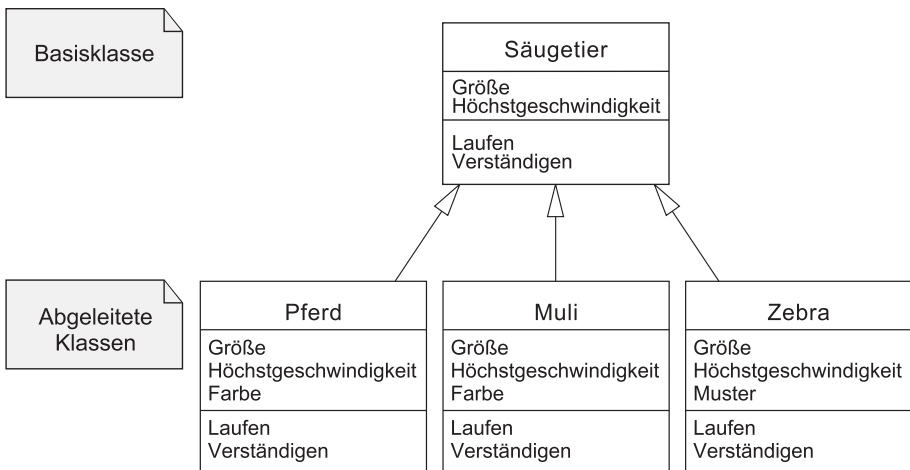


Abbildung 3.10 Die neue Klasse »Muli« ist eine von »Säugetier« abgeleitete Klasse.

3.5.3 Mehrfachvererbung

In der Natur ist sie üblich, in den Programmiersprachen Java und Smalltalk jedoch nicht erlaubt: die Mehrfachvererbung. Sie wäre dann praktisch, wenn Sie zwei Klassen verschmelzen wollten, zum Beispiel die Klasse *Pferd* mit der Klasse *Esel*. Die neue Kreuzung *Muli* würde Attribute und Verhalten beider Basisklassen erben (Abbildung 3.11). Aber welche Attribute und welches Verhalten? Sollen sich Mulis verständigen und laufen wie Pferde oder wie Esel?

Bei derartigen Szenarien kommt die Softwareentwicklung an die Grenze des technisch Sinnvollen. Es ist nicht sinnvoll, Erbinformationen nach dem Zufallsprinzip zu übertragen, um die Natur zu imitieren. Der Anwender wünscht sich im Regelfall Programme, die über definierte Eigenschaften verfügen und deren Verhalten vorhersehbar ist.

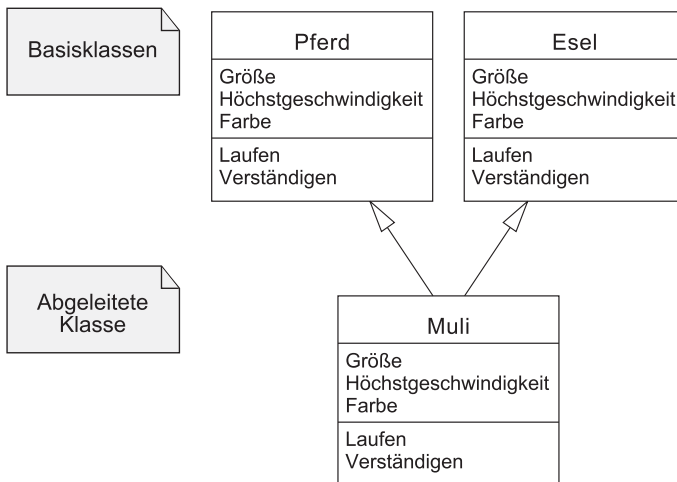


Abbildung 3.11 Mehrfachvererbung am Beispiel einer Kreuzung

Aus den genannten Gründen haben sich die Entwickler der Programmiersprache Java bewusst gegen die konventionelle Mehrfachvererbung entschieden, wie sie in C++ realisierbar ist. Wie Sie trotzdem mehrere Basisklassen ohne Nebenwirkungen miteinander verbinden können, erfahren Sie in Abschnitt 4.5.3.

3.6 Kapselung

Eines der wichtigsten Merkmale objektorientierter Sprachen ist der Schutz von Klassen und Attributen vor unerwünschtem Zugriff. Jedes Objekt besitzt eine Kapsel, die die Daten und Methoden des Objekts schützt (Abbildung 3.12 auf der nächsten Seite). Die Kapsel versteckt die Teile des Objekts, die von außen nicht

oder nur durch bestimmte andere Objekte erreichbar sein sollen. Die Stellen, an denen die Kapsel durchlässig ist, nennt man *Schnittstellen*.

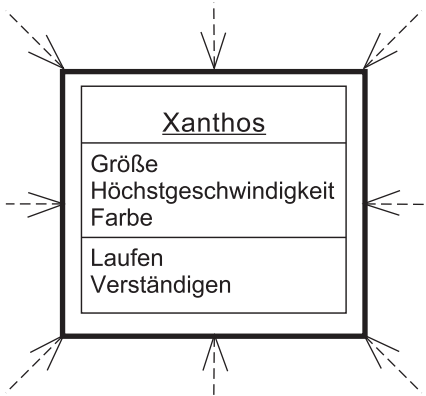


Abbildung 3.12 Die Kapsel schützt das Objekt vor unerwünschten Zugriffen.

Die wichtigste Schnittstelle der Klasse *Pferd* ist sein Konstruktor. Über diese spezielle Methode lässt sich ein Objekt der Klasse *Pferd* erzeugen. Ein anderes Beispiel für eine solche Schnittstelle ist die Methode *Laufen* der Klasse *Pferd*. Das Objekt *Xanthos* besitzt eine solche Methode *Laufen*, und *Achilles*, ein Objekt der Klasse *Mensch*, kann diese Methode verwenden. Er kommuniziert mit *Xanthos* über diese Schnittstelle (Abbildung 3.13) und teilt darüber *Xanthos* mit, dass er laufen soll. Das Objekt *Achilles* darf nicht alle Daten von *Xanthos* verändern. Zum Beispiel soll es ihm selbstverständlich nicht erlaubt sein, die Größe des Pferds zu ändern. Gäbe es eine öffentlich zugängliche Methode wie zum Beispiel *Wachsen*, so könnte er *Xanthos* damit verändern.

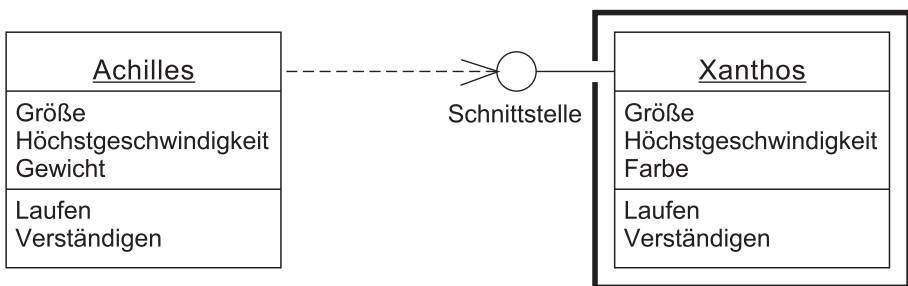


Abbildung 3.13 Objekte kommunizieren nur über Schnittstellen.

3.7 Beziehungen

Klassen und deren Objekte unterhalten in einem Programm die unterschiedlichsten Beziehungen untereinander. In den vergangenen Abschnitten haben Sie

bereits mehrere Formen der Beziehungen kennengelernt: den Aufruf von Methoden und Vererbungen.

An dieser Stelle möchte ich Ihren Blick für die zwei grundlegend verschiedene Arten von Beziehungen zwischen Klassen und Objekten schärfen: Beziehungen, die nicht auf Vererbung beruhen, und Vererbungsbeziehungen.

3.7.1 Beziehungen, die nicht auf Vererbung beruhen

Man unterscheidet bei dieser Form von Beziehungen drei verschiedene Unterarten: *Assoziationen* (Verknüpfungen), *Aggregationen* (Zusammenlagerungen) sowie *Kompositionen* (Zusammensetzungen).

Assoziation

Assoziation ist die einfachste Form einer Beziehung zwischen Klassen und Objekten. Die Abhängigkeiten sind bei dieser Beziehungsart im Vergleich zur Vererbung gering. Man sagt auch, die Objekte sind lose gekoppelt.

Eine Assoziation besteht zum Beispiel, wenn ein Reiter-Objekt namens *Achilles* einem Pferde-Objekt namens *Xanthos* die Botschaft *Springen* sendet (Abbildung 3.14). Die beiden Objekte *Achilles* und *Xanthos* existieren getrennt und erben nichts voneinander.



Abbildung 3.14 Eine einfache Assoziation zwischen Mensch und Pferd

Aggregation

Eine Steigerung der Assoziation ist die Aggregation. Eine solche Beziehung besteht dann, wenn ein Objekt aus anderen Objekten besteht. Zum Beispiel soll Pferdefutter aus einer nicht näher bestimmten Anzahl von Karotten bestehen (Abbildung 3.15). Das bedeutet zum Beispiel, dass Pferdefutter eine »Besteht-aus-Beziehung« zur Karotte unterhält.

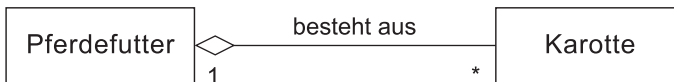


Abbildung 3.15 Aggregation zwischen Pferdefutter und Karotten

Diese Beziehung ist aber von einer völlig anderen Qualität als im vorhergehenden Beispiel zwischen einem Mensch und einem Pferd. Während Pferd und Mensch allein und unabhängig voneinander existieren können, setzt sich das Pferdefutter (unter anderem) aus Karotten zusammen. Wichtig ist hierbei wieder, dass beide Objekte nichts voneinander erben und jedes Karotten-Objekt auch allein lebensfähig ist, was dieses Beispiel von der strengeren Komposition unterscheidet.

Komposition

Die stärkste Form der Beziehungen, die nicht auf Vererbung beruhen, stellt die *Komposition* dar. Wie bei der Aggregation liegt wieder eine »Besteht-aus-Beziehung« vor, sie ist aber im Gegensatz zur Aggregation abermals verschärft. Die Abhängigkeiten sind nochmals stärker.

Ein Beispiel für eine Komposition ist das Verhältnis zwischen einem Pferd und seinen vier Beinen. Hier besteht eine sehr enge Beziehung, denn ein Bein ist – im Gegensatz zur Karotte – als selbstständiges Objekt vollkommen sinnlos. Bei der Erzeugung eines Pferde-Objekts bekommt dieses automatisch vier Beine, die im Zusammenhang mit anderen Klassen nicht verwendet werden können.

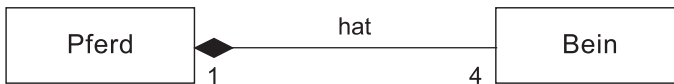


Abbildung 3.16 Ein Pferd und seine vier Beine als Komposition

Pferdebeine sind also ohne ein geeignetes Objekt der Klasse *Pferd* nicht lebensfähig. Wenn ein Pferde-Objekt stirbt, so sterben auch seine Pferdebeine.

3.7.2 Vererbungsbeziehungen

Vererbungsbeziehungen nennen sich auch Generalisierung (Verallgemeinerung) oder Spezialisierung (Verfeinerung). Dies sind nicht etwa Unterarten der Vererbung, sondern alternative Begriffe für Vererbungsbeziehungen. Welchen der zwei alternativen Begriffe man verwenden möchte, hängt vom Blickwinkel ab, aus dem man die Vererbungsbeziehung betrachtet.

Generalisierung

Wenn Sie die Basisklasse aus dem Blickwinkel der abgeleiteten Klasse betrachten wollen, ist Generalisierung der passende Begriff dazu. Zum Beispiel ist die Klasse *Säugetier* eine Generalisierung der Klassen *Pferd* oder *Zebra*. Mit anderen Worten: Die Klasse *Säugetier* ist der allgemeine Begriff (Oberbegriff) für die Klassen *Pferd* und *Zebra* (Klassifizierung).

Spezialisierung

Wenn Sie die abgeleitete Klasse aus dem Blickwinkel der Basisklasse betrachten wollen, ist Spezialisierung der passende Begriff dazu. Zum Beispiel sind die Klassen *Pferd* oder *Zebra* eine Spezialisierung der Klasse *Säugetier*. Mit anderen Worten: Die Klassen *Pferd* und *Zebra* stellen eine Verfeinerung der Klasse *Säugetier* dar.

Probleme mit der Vererbung

Vererbungsbeziehungen stellen eine sehr starke Kopplung zwischen Klassen und damit auch zwischen Objekten her. Eine solch starke Kopplung hat nicht nur Vorteile, sondern auch gravierende Nachteile, wie das folgende Beispiel zeigt:

Eine Klasse namens *Wal* soll aus der Klasse *Säugetier* erzeugt werden (Abbildung 3.17). Die neue Klasse erbt wie die Klassen *Pferd* und *Zebra* die Attribute *Größe* und *Höchstgeschwindigkeit* sowie die Methoden *Laufen* und *Verständigen* – Moment mal: *Laufen*? Fast alle Säugetiere können laufen, Wale jedoch nicht.

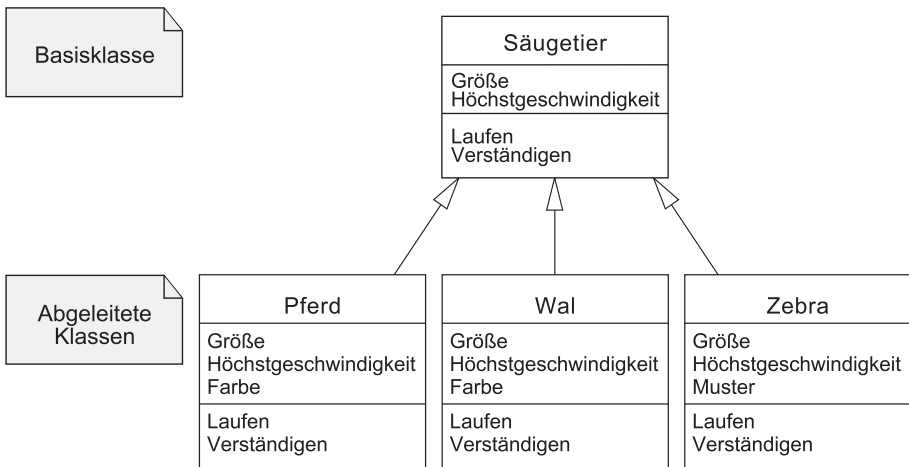


Abbildung 3.17 Durch Vererbung vererben sich auch Designfehler.

Hier ist genau das passiert, was tagtäglich zu den Problemen der objektorientierten Programmierung gehört: Die Funktionalität der Basisklasse ist nicht ausreichend analysiert worden. Vereinfacht gesagt: Hier liegt ein Designfehler vor, den man dadurch beheben muss, dass man die Methode *Laufen* durch die Methode *Fortbewegen* ersetzt.

3.8 Designfehler

Sie können sich vielleicht vorstellen, dass es sehr unangenehm ist, wenn die Basisklasse aufgrund eines Designfehlers geändert werden muss. Durch die starke Beziehung zwischen Basisklasse und abgeleiteter Klasse pflanzen sich etwaige Änderungen lawinenartig in alle Programmteile fort, in denen Objekte des Typs *Pferd* und *Zebra* mit der Methode *Laufen* verwendet wurden. An allen Stellen des Programms, wo die Methode *Laufen* der Klasse *Säugetier* verwendet wurde, muss sie durch die Methode *Fortbewegen* ersetzt werden.

Im Fall von Designfehlern stellt sich die Technik der Vererbung als großer Nachteil heraus. Vererbung hat neben diesem Manko auch den Nachteil, dass sich nicht nur Designfehler, sondern alle anderen vorzüglich gestalteten, aber unerwünschten Teile der Basisklasse in die abgeleiteten Klassen in Form von Ballast übertragen: Die Nachkommen solcher übergewichtiger Klassen werden immer fatter und fatter. Daher sollten Sie Vererbung stets kritisch betrachten, sparsam einsetzen und wirklich nur dort verwenden, wo sie sinnvoll ist.

3.9 Umstrukturierung

Aber zurück zu den Designfehlern. Wie geht man mit Fehlern dieser Art um? Sie sind trotz der Vererbung heute kein so großes Problem mehr wie noch vor ein paar Jahren. Es gibt mittlerweile moderne Softwareentwicklungswerkzeuge (Kapitel 21, »Werkzeuge«), mit denen es relativ leicht ist, die notwendige Umstrukturierung (Refactoring) vorzunehmen. Allerdings sollten Sie Software möglichst nur während der Analyse- und Designphase der Software (Kapitel 5, »Entwicklungsprozesse«) umstrukturieren. Als Regel gilt: Je später Änderungen vorgenommen werden, desto höher ist der damit verbundene Aufwand.

3.10 Modellierung

Um solche Designfehler und damit kostspielige Umstrukturierungen zu vermeiden, ist es bei größeren Projekten sinnvoll, ein Modell der Software zu entwerfen. Genauso wie man im Automobilbau vor jedem neu zu konstruierenden Automobil ein Modell entwickelt, ist es auch in der Softwareentwicklung sinnvoll, ein Modell zu konstruieren, bevor man mit der eigentlichen Umsetzung des Projekts beginnt. Ein Modell, das eine getreue Nachbildung eines kompletten Ausschnitts der Software darstellt, nennt sich Prototyp (Muster, Vorläufer).

3.11 Persistenz

Ein Programm erzeugt Objekte, die an ihrem Lebensende wieder zerstört werden. Diese Objekte bezeichnet man als transient, also flüchtig. Manchmal ist aber

ein »Leben nach dem Tod« auch für Objekte erstrebenswert. Sie sollen auch dann wieder zum Leben erweckt werden, wenn das Programm beendet ist und der Anwender des Programms nach Hause geht. Am nächsten Tag startet der Anwender das Programm erneut und möchte mit dem gleichen Objekt weiterarbeiten.

Solche »unsterblichen« Objekte bezeichnet man als persistent (dauerhaft). Das bedeutet nichts anderes, als dass sie in geeigneter Form gespeichert werden. Sie befinden sich dann in einer Art Tiefschlaf in einer Datei auf einer Festplatte oder im Verbund mit anderen Objekten in einer Datenbank.

3.12 Polymorphie

Der Name Polymorphie kommt aus dem Griechischen und bedeutet so viel wie Vielgestaltigkeit, Verschiedengestaltigkeit. Der Begriff klingt mehr nach Mineralienkunde als nach Informatik, und so wundert es Sie vielleicht auch nicht, dass der Chemiker Mitscherlich die Polymorphie bei Mineralien Anfang des 19. Jahrhunderts entdeckte. Er stellte fest, dass manche Mineralien wie Kalziumcarbonat (CaCO_3) unterschiedliche Kristallformen annehmen können, ohne ihre chemische Zusammensetzung zu ändern. Das bedeutet, sie können je nach Druck und Temperatur eine verschiedene Gestalt annehmen.

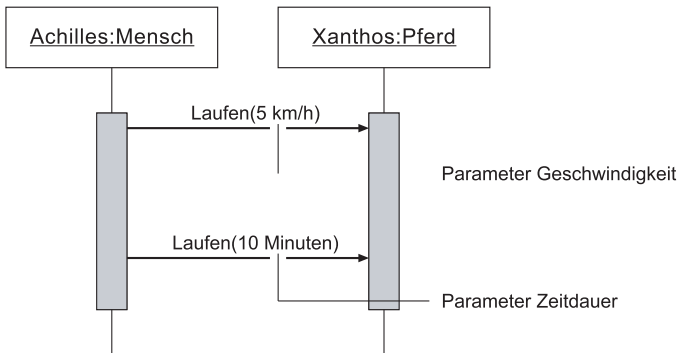


Abbildung 3.18 Xanthos verfügt über zwei verschieden gestaltete Methoden namens »Laufen«. Sie unterscheiden sich durch ihre Parameter.

Alles sehr schön bis jetzt, aber was hat das mit objektorientierter Programmierung zu tun? – Das bedeutet auf keinen Fall, dass ein Objekt wie *Xanthos* so radikal seine Form verändern kann wie ein Mineral. Es bedeutet, dass *Xanthos* bei geschickter »Programmierung« situationsbedingt verschieden reagieren kann. Klingt wie Zauberei, ist es aber nicht.

3.12.1 Statische Polymorphie

Stellen Sie sich vor, das Objekt *Achilles* teilt dem Objekt *Xanthos* mit, dass es laufen soll, und zwar mit der Geschwindigkeit 5 km/h. Was wird passieren? – *Xanthos* wird sich mit dieser Geschwindigkeit fortbewegen. Offensichtlich ist die Richtung ebenso egal wie die Dauer. Was würde passieren, wenn *Achilles* abermals *Xanthos* mitteilt, er solle laufen, und zwar 10 Minuten? *Xanthos* würde 10 Minuten lang mit 5 km/h laufen und danach stehen bleiben.

Damit *Xanthos* den etwas wirr klingenden Anweisungen seines Reiters Folge leisten kann, benötigt er Methoden »unterschiedlicher Gestalt«. Er benötigt eine Methode, die auf den Parameter Geschwindigkeit reagiert, und eine Methode, die auf den Parameter Zeitdauer reagiert. Obwohl die Methoden den gleichen Namen tragen, führen sie zu einer unterschiedlichen Verarbeitung durch das Objekt *Xanthos*. Der Fachausdruck für diese Technik heißt *Überladen*.

3.12.2 Dynamische Polymorphie

Anders als bei der Mehrfachvererbung sieht es aus, wenn man Eigenschaften der Basisklasse bei der Vererbung bewusst umgehen möchte. Dazu möchte ich nochmals auf das Beispiel der Basisklasse *Säugetier* zurückgreifen. Angenommen, Sie möchten in der abgeleiteten Klasse *Pferd* bestimmen, auf welche Weise sich Pferde-Objekte verständigen. Dazu *überschreiben* Sie die Methode *Verständigen* und legen die Art und Weise des Wieherns in der Klasse *Pferd* für die abgeleiteten Objekte fest.

Das Überschreiben von Methoden ist ein sehr mächtiges Mittel der objektorientierten Programmierung. Es erlaubt Ihnen, unerwünschte Erbinformationen teilweise oder ganz zu unterdrücken und damit eventuelle Designfehler – in Grenzen – auszugleichen beziehungsweise Lücken in der Basisklasse zu füllen. Dabei ist die Technik extrem simpel. Es reicht aus, eine identische Methode in der abgeleiteten Klasse *Pferd* zu beschreiben, damit sich Objekte wie *Xanthos* »plötzlich« anders verhalten.

3.13 Designregeln

Auch wenn die objektorientierte Softwareentwicklung im Vergleich zur konventionellen Programmierung gutes Softwaredesign besser unterstützt, ist sie auf keinen Fall eine Garantie für sauber strukturierte und logisch aufgebaute Programme. Die objektorientierte Programmierung erleichtert zwar gutes Softwaredesign, sie erzwingt es jedoch nicht. Da man trotz Objektorientierung schlechte Programme entwickeln kann, sollten Sie einige Grundregeln beachten:

- ▶ Vermeiden Sie Vererbung.
- ▶ Reduzieren Sie die Anforderungen auf das Wesentliche.
- ▶ Kapseln Sie alle Attribute und Methoden, die nicht sichtbar sein müssen.
- ▶ Arbeiten Sie bei großen Projekten mit einem Modell.
- ▶ Verwenden Sie einen Prototyp.

3.14 Zusammenfassung

Die objektorientierte Programmierung war eine Antwort auf die Softwarekrise in der Mitte der 60er-Jahre des letzten Jahrhunderts. Durch Objektorientierung lässt sich die natürliche Welt leichter in Computerprogrammen umsetzen. Diese objektorientierten Computerprogramme bestehen aus einer Sammlung von einem oder mehreren Objekten.

Ein Objekt lässt sich mit einem natürlichen Lebewesen vergleichen und verfügt über eine Gestalt und Fähigkeiten. Die Gestalt prägen Attribute, während die Fähigkeiten von Methoden bestimmt sind. Beide Bestandteile eines Objekts sind in der Klasse festgelegt, von der ein Objekt abstammt. Sie liefert den Bauplan für gleichartige Objekte.

Objektorientierte Programmierung ist kein Allheilmittel. Sie unterstützt gutes Design, ohne es zu erzwingen. Es ist deshalb notwendig, auf sauberes Design zu achten, wenn man mit objektorientierter Programmierung erfolgreich sein will.

3.15 Aufgaben

Versuchen Sie, folgende Aufgaben zu lösen:

3.15.1 Fragen

1. Worin unterscheiden sich Klassen von Objekten?
2. Wie unterscheiden sich Objekte der gleichen Klasse voneinander?
3. Was bedeutet der Begriff »Basisklasse«?
4. Was bedeutet der Begriff »abgeleitete Klasse«?
5. Wie verständigen sich Objekte untereinander?
6. Welche Arten von Beziehungen gibt es, und wie unterscheiden sie sich?
7. Worin liegt die Gefahr bei Vererbungsbeziehungen?

3.15.2 Übungen

1. Zeichnen Sie zur Abbildung 3.19 eine Klasse mit Klassennamen, Attributen und Methoden.

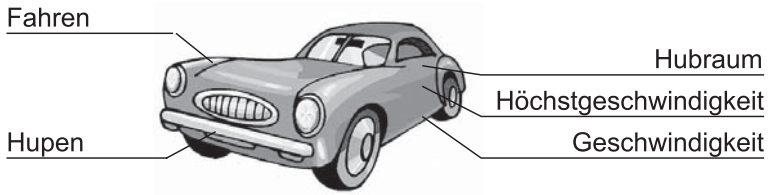


Abbildung 3.19 Ein Objekt mit verschiedenen Merkmalen und Fähigkeiten

2. Zeichnen Sie zur Abbildung 3.20 eine gemeinsame Basisklasse und aus den zwei Objekten zwei abgeleitete Klassen mit Klassennamen, Attributen und Methoden.

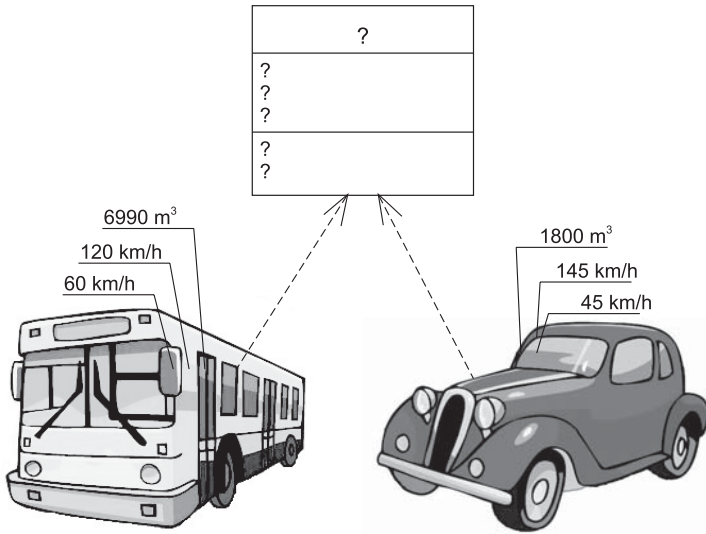


Abbildung 3.20 Zwei verschiedene Objekte

3. Zeichnen Sie zur Abbildung 3.21 ein Klassendiagramm mit einer Basisklasse und drei abgeleiteten Klassen, die in Beziehung zur Basisklasse stehen.

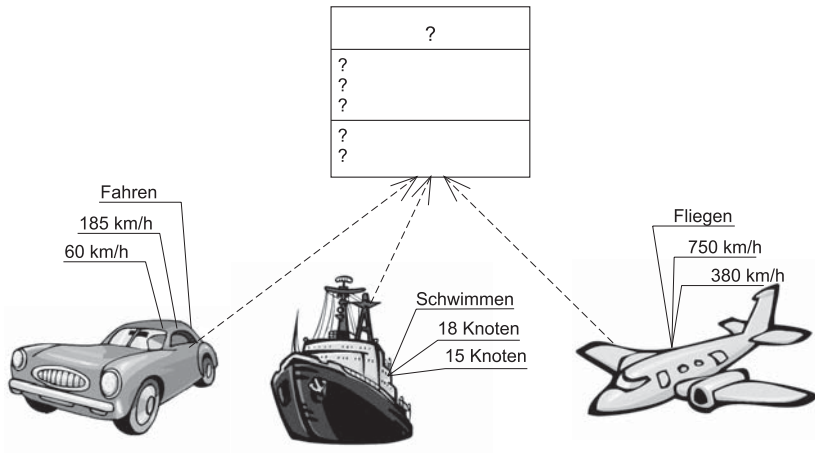


Abbildung 3.21 Drei verschiedene Objekte

Die Lösungen zu den Aufgaben finden Sie in Kapitel 18 ab Seite 487.

Index

A

Abakus 347
Abgeleitete Klasse 72
Ableiten 108
Abschnittsbezogene Kommentare 141
abstract 92
Abstract Windowing Toolkit 250, 569
Abstrakte Klasse 106, 109, 569
Abstrakte Methode 569
Abstraktion 64, 69
Acceleratoren 569
Active Server Pages 272
Aggregation 75
Aktivitäten 148
Akzessor 68, 116
Algol 51
Algorithmen 281
 anwenden 288
Algorithmen entwickeln 281
Algorithmenarten 282
American National Standards Institute 569
American Standard Code for Information Interchange 569
Analyse 148
Anforderungsaufnahme 148
Anonyme Klasse 108
ANSI 569
ANSI-Code 35
Anweisungen 131
Anwendungsarchitektur 569
Anwendungsfall 569
API 569
Applets 265
appletviewer 553
Application 267
Application Objects 274
Application Programming Interface 569
Application Server 543

Applikationsschicht 270
Architektur 569, 571, 574
ArgoUML 540
Argument 114
Argumente 114, 117
Argumente übergeben 117
Arithmetische Operatoren 118
Arrays 102
ASCII 569
ASCII-Code 33
ASP 272
Assembler-Sprache 48
assert 92
Assoziation 75
Attribut 64, 65
Aufbau eines Computers 561
Aufzählungstyp 105
Ausdruck 131
Ausnahmebehandlung 240
AWT 250, 569

B

BASIC 51
Basisklasse 72, 569
Behälterklasse 569
Betriebsphase 147
Bezeichner 95
Beziehung 74, 75
Beziehungen 64
Bildlauffeld 569
Bildlauffeiste 569
Binärcode 47, 50
Binärformat 46
Binärprogramm 28
Binärsystem 28
Binärzahlen 28, 31
Bit 32
Bitweise Operatoren 128
Block 133
BMP 274
boolean 92, 94, 101
Border-Layout 255
break 92

Bussystem 561
Button 569
BX for Java 542
Byte 33
byte 92, 94, 98, 104

C

C 51, 118, 134, 141, 269
C++ 51, 116, 118, 134, 269
C# 51
CardListener 326
case 92
Case-Verzweigung 134
Cast-Operator 130, 208
catch 92
Central Processing Unit 562
CGI 271, 403, 408, 570
CGI-Programm 570
char 92, 94, 102, 104
Charon 433, 469
class 92
Clipboard 575
CMP 274
COBOL 51
Combo Box 570
Common Facilities 274
Common Gateway Interface 271, 408, 570
Common Object Services 274
Compiler 154
Compilieren 154
Computer Aided Software Engineering 570
Computerhardware 561
const 92
Container 250
Container Managed Persistence 274
Container-Klasse 570
continue 92
Coprozessor 96
CORBA 273, 537, 570

CPU 562
 CVS 545

D

Datenbank 429
 Datenbank-Management-system 570
 Datenbank-anwendungen 449
 Datenbanken 537
 Datenbank-programmierung 429
 Datenmodell 429
 Datentyp 94
 DBMS 570
 Debugger 537
 default 92, 195
 Deklaration 94
 Deployment 545
 Derby 543
 Design 148
 Designfehler 78
 Designregel 80
 Destruktor 68, 116
 Dezimaldarstellung 27
 Dezimalsystem 27
 Dezimalzahl 29
 Dialog 570
 Dialogfeld 570
 Dialogfenster 570
 Differenz 120
 Digitalcomputer 28
 Digitalsystem 28
 Digitalzahlen 28
 do 92
 Do-Schleife 136
 doGet 414
 Dokumentation 141
 Dokumentations-kommentare 141
 Doppelwort 32
 double 92, 94, 101
 Dreamweaver 542
 Dualsystem 28
 Dünner Treiber 268
 Dynamische Polymorphie 80
 Dynamische Websites 469

E

Eclipse 546
 Editieren 153
 Ein- und Ausgabesteuerung 564
 Einfache Klassentypen 235
 Einfacher Datentyp 93
 Einfachvererbung 570
 Einzelwerkzeuge 539
 EJB 274, 537
 Elementare Anweisungen 133
 else 92
 Enterprise JavaBeans 270, 274, 275
 Entity Beans 275
 Entwicklungsprozess 147
 Entwurfsmuster 570
 enum 92, 93
 Enumeration 570
 ER-Modell 431, 570
 Ereignis 570
 Ereignisbehandlung 252
 Ereignissteuerung 252, 264
 Erweiterter Datentyp 102
 Event-Handling 252
 Excelsior 541
 Exception Handling 240, 247
 Exemplar 570
 extends 92, 108, 109
 Extensible Markup Language 570
 Extranet 570

F

Fachliche Architektur 571
 Fachliches Klassenmodell 571
 false 92
 Fehlerbehandlung 240, 241
 Festkommazahl 97
 Festplattenspeicher 564
 FileReader 247
 FileWriter 248
 final 92

finalize 116
 finally 92
 Firebird 544
 Firewalls 266
 Flash 403
 float 92, 94, 100
 Floating Window 571
 Fokus 571
 for 92
 For-Schleife 136
 FORTRAN 51
 Fragezeichenoperator 129
 FTP 571
 Funktion 69, 117

G

Ganzzahl 96
 Garbage Collector 182
 GByte 32
 Genauigkeit 95
 Generalisierung 76, 571
 Generics 106, 111
 Generische Klasse 106, 111
 Gleitkommazahl 96
 goto 92
 Grafikprozessor 563
 Graphics 284
 GridBag-Layout 257
 Group Box 571
 Gruppenfeld 571
 GUI 571
 GUI-Builder 535, 542

H

Hades 431
 HadesTest 441
 Handler 340, 394
 Hauptspeicher 563
 Heap 563
 Hexadezimalsystem 30
 Hilfsklasse 249
 Höhere Datentypen 106
 Home Interface 275
 Hot Swap 538
 HotJava 89
 hsqldb 544

HTML 403, 571
HTTP 271, 571
Hypertext Markup
Language 403
Hypertext Transfer
Protocol 406
Hypertext Transport
Protocol 271, 571

I

IBM 574
if 92
if-Anweisung 134
If-Verzweigung 134
IIOP 266, 571
Implementierung 569, 574
implements 92, 111
Import 138
import 92
Importanweisung 138
InstallAnywhere 545
Installation 545
Installationsprogramm 545
instanceof 92
Instantiierung 571
Instanz 64, 106, 571
Instanzen 64, 106, 569
Instanzieren 106
Instanziierung 571
int 92, 94, 99, 104
Interface 106, 569, 571,
575
interface 92
Interfaces 110
Internet 571, 575
Internet Inter Orb
Protocol 266
Intranet 571

J

J2EE 230
J2ME 230
J2SE 230
JAD 542
JAR 555
Java 51, 100–102, 106,
116, 230, 231, 240, 260,
265, 270, 273–275, 569

Java 2 Enterprise
Edition 230, 270
Java 2 Micro Edition 230,
275
Java 2 Standard
Edition 230
Java Database Connec-
tivity 267
Java Development Kit 549
Java GUI Builder 543
Java Native Interface 269
Java Remote Method
Protocol 571
Java Runtime Environ-
ment 231
Java-Compiler 534, 541
Java-Datenbank Derby 543
Java-Datenbank
hsqldb 544
Java-Debugger 537
Java-Decompiler 535, 542
Java-Language-
Bibliothek 231
JavaBean 264, 273, 326
JavaBeans 264
javac 551
javap 554
JavaServer Pages 270, 272
JBuilder 548
jdb 553
JDBC 267, 430
JDBC-ODBC-Bridge 268
JDBC-Treiber 267
JDK 230, 549
JDK-Switching 531
jEdit 540
JexePack 545
Jikes 541
JMenu 315
JMenuBar 315
JNI 269
JRE 231
JRMP 572
JSP 272
JSwat 544
jvider 542

K

Kapselung 64, 73, 110,
274
Kardinalität 572
KByte 32
Kennung 66
Klasse 63, 64, 102, 106,
108–111, 115, 138, 195,
196, 214, 215, 240, 266,
569, 571, 572, 574
Klassenattribut 572
Klassenbibliothek 227, 572
Klassenimport 138
Klassenmethode 572
Klassenoperation 572
Klassenvariable 112, 572
Kommentar 141
Kompilieren 154
Komposition 76
Konkrete Klasse 106, 572
Konsolenprogramme 297
Konstante 65
Konstanten 112
Konstruktionsphase 147
Konstruktor 68, 115, 572
Kontrollfeld 572
Kriterien zur Werkzeugaus-
wahl 527

L

Laufzeitumgebung 536,
543
Layout-Manager 254
Lebensdauer 573
Linux 575
Logische Operatoren 36,
126
Lokale Klasse 108
long 92, 94, 99

M

Makrobefehle 52
Maschinenprogramm 46,
47
Maschinensprache 46
MByte 32

- Mehrfachvererbung 73, 110, 572
 - Menüleiste 315, 380
 - Menüs 348, 377
 - Methode 67, 113, 214, 215, 569, 572, 574
 - Methoden 112
 - Mikrobefehle 47, 49
 - Mnemonics 572
 - Modales Dialogfenster 572
 - Model View Controller 572
 - Modell 78
 - Modellierung 78
 - Modellierungswerkzeuge 531
 - Modul 138
 - Mutator 68, 117
 - MVC 572
 - MySQL 544
- N**
- Namensraum 140
 - Nativ 572
 - native 92
 - Native Java-Programme 183
 - Native-API-Treiber 268
 - Native-Code-Compiler 183
 - Natural 53
 - Negation 126
 - Nestor 377, 462
 - Net-Treiber 268
 - NetBeans 555
 - Netscape 575
 - Netzwerk 573
 - new 92
 - New-Operator 130
 - Nibble 32
 - Nicht-Funktion 39
 - Nichtmodale Dialogfenster 573
 - null 92
- O**
- Oak 89
 - Oberklasse 573
 - Object 231
 - Object Management Group 273, 570, 573
 - Object Request Broker 274, 570, 573
 - Objekt 63–65, 102, 118, 269, 274, 573
 - Objekte 572, 573
 - Objekte erzeugen 106
 - Objektidentität 573
 - Objektmethode 113
 - Objektorientierte Programmiersprache 63, 573
 - Objektorientierung 573
 - Objektvariable 112, 573
 - Oder-Funktion 38
 - Oder-Verknüpfung 127
 - OMG 273, 573
 - OO 573
 - OOA/OOD 78
 - Operator 118
 - Optionsfeld 573
 - ORB 573
- P**
- Package 138
 - package 92, 140
 - Paket 90, 138
 - Parameter 114, 117
 - Parameter übergeben 117
 - Parameterübergabe 117
 - Parameterliste 114
 - Pascal 51
 - Perforce 545
 - Perl-Skripte 272
 - Perseus 449
 - Persistentes Objekt 573
 - Persistenz 64, 78
 - Phase 147
 - PHP-Skripte 272
 - Planungsphase 147
 - Polymorphie 64, 79, 213, 573
 - Polymorphismus 79, 573
 - Portabilität 47, 50, 52, 54, 56
 - Poseidon 540
 - Postdekrement 122
 - Postinkrement 122
 - Prädekrement 122
 - Präinkrement 121
 - private 92, 195
 - Produkt 120
 - Programmierkonventionen 218
 - Prolog 55
 - Properties 249
 - Properties-Datei 307
 - protected 92, 195
 - Proxy-Schicht 270
 - Prozessoren 562
 - public 92, 195
- Q**
- Query 574
 - Quotient 120
- R**
- Radio Button 573
 - Radioschalter 573
 - Rapid Prototyping 573
 - Rational XDE 556
 - Rechenwerk 562
 - Rechnerunendlich 96
 - Refactoring 78
 - Remote Interface 275
 - Remote Method Invocation 269, 573
 - Remote-Debugging 538
 - Remote-Schicht 270
 - return 92
 - Reverse-Engineering 573
 - RMI 269, 573, 574
 - Roundtrip-Engineering 574
 - Rumpf einer Methode 115
 - Runtime 244
- S**
- Schaltfläche 574
 - Schlüsselwort 92
 - Schleifen 135
 - Schnittstelle 110
 - Schriftkonventionen 21
 - Scroll Bar 569
 - Sedezimalsystem 30

- Servlets 271, 403, 537
 - Session Beans 275
 - Shell 574
 - Shellskript 574
 - short 92, 94, 98, 104
 - Short-Cuts 569
 - Shortcuts 574
 - Sicherheitseinstellungen 265
 - Sichtbarkeit einer Methode 113
 - Signatur 114, 574
 - SIMULA 51
 - Smalltalk 63
 - Solaris 575
 - Sortieren 283, 289
 - SourceAgain 542
 - Speicher freigeben 182
 - Speichermedien 563
 - Spin Button 574
 - SplashWnd 450
 - SQL 574
 - Stack 563
 - Standardkonstruktor 115
 - Stateful Session Beans 275
 - Stateless Session Beans 275
 - static 92, 574
 - Statische Polymorphie 80
 - Steuerwerk 563
 - strictfp 92
 - String 233
 - StringBuffer 237
 - Subklasse 574
 - Summe 119
 - Sun Microsystems 89
 - Sun One Studio 556
 - super 92, 574
 - Superklasse 574
 - Superklasse Object 231
 - Swing 251, 313, 347, 377
 - switch 92
 - switch-Anweisung 134
 - Symbolleiste 378, 574
 - synchronized 92
 - System 237
 - Systemarchitektur 574
- T**
- Tag 403
 - Tags 574
 - Taktgeber 565
 - Tastaturkombinationen 569, 574
 - Tastenkombinationen 569
 - TByte 32
 - Technische Architektur 574
 - Test 148
 - Texteditor 533, 540
 - this 92, 574
 - Thread 303
 - Threads 245
 - throw 92
 - throws 92
 - Together 540, 556
 - Tomcat 543
 - Tool Bar 574
 - Transfer 297
 - transient 92
 - Transientes Objekt 574
 - Transportschicht 270
 - true 92
 - try 92
 - Typ des Rückgabewertes 114
 - Typkonvertierung 130, 208
- U**
- Überladen von Methoden 213
 - Überschreiben verhindern 218
 - Überschreiben von Methoden 215
 - Übersetzen 154
 - UI 574
 - UltraEdit 541
 - UML 56, 574
 - UML-konform 532
 - Umstrukturierung 78
 - Und-Funktion 37
 - Und-Verknüpfung 127
 - Unicode 35
 - Uniform Resource Locator 575
 - Uniplexed Information and Computing System 575
 - UNIX 575
 - Unterklasse 575
 - URL 575
- V**
- vererben 108
 - Vererbung 64, 70, 108, 111, 215, 571, 572, 575
 - Vergleich auf Gleichheit 123
 - Vergleich auf größer 125
 - Vergleich auf größer oder gleich 126
 - Vergleich auf kleiner 124
 - Vergleich auf kleiner oder gleich 125
 - Vergleich auf Ungleichheit 124
 - Vergleichender Operator 123
 - Verifizierung 575
 - Versionskontrolle 531
 - Versionskontrollwerkzeuge 544
 - Verteilung 545, 569
 - Verzweigungen 134
 - Vielgestaltigkeit 573
 - VisualAge Java 557
 - VM 178
 - void 92, 117
 - volatile 92
 - Vorzeichen 95, 119
- W**
- Wahrheitswert 101
 - Wahrheitswerte 36, 126
 - Webbrowser 575
 - Webseite 575
 - Webserver 575
 - Website 575
 - WebSphere Studio 558
 - Werkzeug 149, 525
 - Werkzeuge zur Verteilung 538
 - Werkzeugsuiten 525, 545
 - Wertebereich 95

while 92
While-Schleife 135
World Wide Web 575
Wort 33
Wrapper-Klasse 237
WWW 575
WYSIWYG 575

X
XML 272, 570, 575

Z
Zahlensysteme 27
Zeichen 102
Zeilenbezogene Kommen-
tare 141

Zustand 66
Zuweisung 131
Zuweisungsoperator 128
Zuweisungsoperatoren 128
Zwischenablage 575