Teil I

Grundlagen

In diesem Teil:

- **Kapitel 1**
  Objektorientierte Programmierung – diesmal richtig 29
- **Kapitel 2**
  Automatisierte Tests 43
- **Kapitel 3**
  Entwicklungsprinzipien der objektorientierten Programmierung 69
- **Kapitel 4**
  Entwurfsmuster der objektorientierten Programmierung 81
- **Kapitel 5**
  Refactoring 111
- **Kapitel 6**
  Fehlerbehandlung 123

Objektorientierte Programmierung – diesmal richtig

Da es im Zusammenhang mit der objektorientierten Programmierung immer wieder zu Missverständnissen kommt, möchte ich die Grundlagen dieser Technik kurz beschreiben. Erfahrungsgemäß trägt ein historischer Rückblick zum Verständnis dieser Methode bei, weshalb wir am Anfang beginnen wollen – nicht ganz am Anfang, aber dennoch früh genug.

1.1 Die prozedurale Programmierung

Die erste Programmiersprache, die die prozedurale Programmierung unterstützte, war FORTRAN (FORmula TRANslator), eine Programmiersprache, die in den fünfziger Jahren des vorigen Jahrhunderts zur Lösung mathematischer Probleme bei IBM entwickelt wurde. 1985 wurden mit FORTRAN II Sprachmittel zur prozeduralen Programmierung wie Unterprogramme (SUBROUTINE, FUNCTION) und Schleifen (DO) eingeführt. Das war ein großer Fortschritt gegenüber der Verwendung von absoluten und relativen Sprüngen (GOTO, JUMP). Bei der prozeduralen Programmierung wird ein Programm in Unterprogramme (Prozeduren, Funktionen) aufgeteilt. Daneben kennt ein prozedurales Programm lokale und globale Variablen. Erstere sind nur innerhalb eines Unterprogramms gültig, auf globale Variablen hingegen können alle Unterprogramme zugreifen.

In den darauf folgenden Jahrzehnten entstanden zahlreiche weitere prozedurale Programmiersprachen wie Algol, PASCAL, MODULA und C. Kein anderes Paradigma (Musterbeispiel) hat eine so weite Verbreitung unter Programmierern gefunden wie die prozedurale Programmierung.

Hier ein Beispiel für ein prozedurales Programm. Um Duplikate zu vermeiden, wird der Mechanismus zum Erzeugen der Tierlaute in der Funktion `letSpeak` zusammengefasst:

```
public class AnimalSounds {
    static void letSpeak(String animal) {
        if (animal.equals("dog")) {
            System.out.println("bark");
        } else if (animal.equals("cat")) {
```

```
            System.out.println("miaow");
        } else {
            System.err.println("No idea what " + animal
                    + " speaks");
        }
    }

    public static void main(String[] args) {
        letSpeak("dog");
        letSpeak("cat");
    }
}
```

Listing 1.1: Beispiel für prozedurale Programmierung

Dem Vorteil der Einfachheit dieses Paradigmas standen aber schon bald seine Nachteile entgegen, die sich besonders in großen Programmen auswirkten. Da der Zugriff auf globale Variablen von jeder Stelle des Programms aus möglich war, verlor man leicht den Überblick darüber, welches Unterprogramm auf welche Variablen zugriff. Darüber hinaus konnten Unterprogramme einander beliebig gegenseitig aufrufen, was zu komplexen Abhängigkeiten zwischen den Unterprogrammen führte. Die geschilderten Nachteile der prozeduralen Programmierung und der Sprachen, die sie unterstützte, verhinderten eine Modularisierung von Programmen in größerem Umfang. Umfangreiche Programme konnten dadurch nur mit großer Mühe erweitert werden. Mit der mangelnden Übersicht wurde natürlich auch das Hinzuziehen und Schulen neuer Programmierer immer aufwändiger, sobald die Programme eine gewisse Größe erreicht hatten. Allerorten war der gefürchtete »Spaghetti-Code« die Folge. Zur Lösung dieser Probleme wurden unterschiedliche Spracherweiterungen wie das Zusammenfassen von Funktionen in Module vorgeschlagen und umgesetzt. Geholfen hat das nur bedingt, weshalb sich Programmierer bald nach anderen Paradigmen umsahen.

1.2 Die objektorientierte Programmierung

Die erste Programmiersprache, die die objektorientierte Programmierung unterstützte, war SIMULA I, die in den Jahren 1962-65 am Norwegian Computing Center in Oslo von Ole-Johan Dahl und Kristen Nygaard entwickelt wurde. Sie wurde (wie der Name schon vermuten lässt) vornehmlich zur Durchführung von Simulationen entwickelt und genutzt. Obwohl sie in einigen Nischen bis heute verwendet wird, war ihr kein großer Erfolg beschieden. Erst mit der ungeheuren Popularität der Programmiersprache C++ in den späten achtziger Jahren und erst recht mit der der Sprache Java in den neunziger Jahren des vorigen Jahrhunderts wurde die objektorientierte Programmierung zum dominierenden Paradigma. Die Auftei-

lung von objektorientierten Programmen war vollkommen anders als die der prozeduralen. Daten und Verhalten wurden in *Objekten* gekapselt, die über *Methoden* (objektgebundene Unterprogramme) Nachrichten austauschen konnten, weshalb sie auch als *Akteure* (Schauspieler) bezeichnet wurden. Das ganze entsprach dem Modellierungsansatz, den man bei Simulationen wählt: Komplexe Systeme werden in Untersysteme (*Objekte*) aufgeteilt und die Kommunikation dieser Untersysteme miteinander untersucht. Die Kapselung von Daten in Objekten bewirkte auch, dass Objekte nur über Methodenaufrufe auf die Daten anderer Objekte zugreifen konnten. Zusätzlich wurde die Technik der *Vererbung* eingeführt. Mit ihr konnten spezialisierte Objekte von allgemeineren abgeleitet werden. Kindobjekte konnten damit die Funktionalitäten der Elternobjekte um spezifische eigene Funktionalitäten erweitern.

Zur objektorientierten Variante des vorigen Beispiels: Die einzelnen Tierobjekte werden in der Klasse `Animal` zusammengefasst (abstrahiert). Der Laut, den das jeweilige Tier erzeugt, wird erst im jeweiligen Tierobjekt definiert. Die Methode `main` erzeugt jeweils ein solches Tierobjekt und weist es an, Laut zu geben (`speak`).

```
public class AnimalSounds {
   static interface Animal {
      void speak();
   }

   static class Dog implements Animal {
      public void speak() {
         System.out.println("bark");
      }
   }

   static class Cat implements Animal {
      public void speak() {
         System.out.println("miaow");
      }
   }

   public static void main(String[] args) {
      Dog bello = new Dog();
      bello.speak();
      Cat tom = new Cat();
      tom.speak();
   }
}
```

Listing 1.2: Beispiel für objektorientierte Programmierung

Hinweis

In diesem einfachen Beispiel ergibt sich kein offensichtlicher Vorteil gegenüber der prozeduralen Version. Das ändert sich mit zunehmender Komplexität allerdings rasch.

Die objektorientierte Programmierung wird durch folgende Paradigmen gekennzeichnet:

1.2.1 Abstraktion

Jedes Objekt im System kann als *Akteur* betrachtet werden, der andere Objekte erzeugen und Nachrichten an andere Objekte senden sowie von ihnen empfangen kann. Das Objekt *kann* dabei anderen Objekten seinen eigenen inneren *Zustand* durch Methodenaufrufe mitteilen. Die konkrete Implementierung eines Objekts ist unterdessen allen anderen Objekten unbekannt. Eine Abstraktion dieser Art kann entweder als *Klasse* oder direkt durch Erzeugen eines Objekts implementiert werden. In Java (und auch in den meisten anderen objektorientierten Programmiersprachen) finden Klassen Verwendung. Es handelt sich hierbei um Datentypen, aus denen zur Laufzeit Objekte generiert werden. Das Verhalten eines Objekts wird in *Methoden* definiert. Der Zustand des Objekts wird in *Attributen* (*Feldern* bzw. *Membern*) gespeichert.

1.2.2 Datenkapselung

Als *Datenkapselung* bezeichnet man das Verbergen der internen Daten des Objekts. Eine Änderung des Objektzustands eines Objekts erfolgt ausschließlich über dessen Methoden. Dadurch hat das Objekt jederzeit die vollständige Kontrolle über seinen internen Zustand. Im Gegensatz zur reinen Lehre der objektorientierten Programmierung kann man in Java Daten öffentlich zugänglich machen, indem man öffentliche Felder definiert und damit die Datenkapselung bricht. Davon sollte man aber generell absehen, es sei denn, das Feld ist konstant (`public static final`) oder es kann nur im Konstruktor gesetzt werden (`public final`).

1.2.3 Vererbung

Der Begriff *Vererbung* wird hier etwas anders definiert als im herkömmlichen Sprachgebrauch. Der Sinn der Vererbung ist es nicht, vor dem Ableben eines Objekts irgendetwas an ein anderes weiterzugeben, vielmehr ist mit Vererbung in diesem Fall *Spezialisierung* gemeint und das geschieht auf Klassenebene (sofern die Programmiersprache Klassen besitzt, wovon wir künftig ausgehen). Bei der Vererbung übernimmt (erbt) das Kindobjekt die Daten und das Verhalten des

Elternobjekts und erweitert es gegebenenfalls um neue Daten und neues Verhalten. Dabei kann das Verhalten der Elternklasse teilweise oder ganz vom Verhalten der Kindklasse überlagert werden. In Java (und vielen anderen objektorientierten Programmiersprachen) kann die Elternklasse festlegen, auf welche Felder die Kindklasse zugreifen darf und welche Methoden es überlagern darf. Der Begriff *Klasse* wird hier als Überbegriff für die Java-Sprachkonstrukte *Interface*, *abstrakte Klasse* und *konkrete Klasse* verwendet.

1.2.4 Polymorphie

In Java kommt der Begriff *Polymorphie* in Zusammenhang mit Objekten und Methoden vor. Wir betrachten hier erst einmal die Polymorphie für Objekte. Eigentlich ist der Begriff Polymorphie (Vielgestaltigkeit) in diesem Fall irreführend, da sich die Gestalt, also die äußere Schnittstelle nicht ändert, sondern lediglich die Implementierung. Wie wir bereits besprochen haben, können Objekte Methoden anderer Objekte (die Zielobjekte) aufrufen und dabei wiederum Objekte als Parameter übergeben (Parameterobjekte). Ein Objekt muss sich dabei allerdings nicht an die Vorgaben der Methode des Zielobjekts halten. Es kann auch ein Kindobjekt des geforderten Datentyps übergeben. Die Methode des Zielobjekts verwendet dann die Methodensignatur (Schnittstelle) des Parameterobjekts, obwohl intern die Methoden des Kindobjekts dieses Parameterobjekts verwendet werden. Das Kindobjekt kommt also in »Verkleidung« des Elternobjekts daher. Die Methode des Zielobjekts braucht dazu in keinster Weise verändert zu werden: Die Programmiersprache kümmert sich automatisch darum, dass die richtige Methode (die des Kindobjekts oder die des Elternobjekts, abhängig davon, ob das Kindobjekt besagte Methode überlädt oder nicht) aufgerufen wird. Das Ganze klingt viel komplizierter, als es in Wahrheit ist. Vielmehr ist es das Verhalten, das man ohnehin erwarten würde und ohne das die ganze Vererbung eher nutzlos wäre.

Im folgenden Beispiel ruft die Methode `letSpeak` die Methode `speak` des übergebenen Tieres (Implementierung von `Animal`) auf:

```
public class AnimalSounds {
    static interface Animal {
        void speak();
    }

    static class Dog implements Animal {
        public void speak() {
            System.out.println("bark");
        }
    }
```

```
    static class Cat implements Animal {
        public void speak() {
            System.out.println("miaow");
        }
    }

    static void letSpeak(Animal animal) {
        animal.speak();
    }

    public static void main(String[] args) {
        Dog bello = new Dog();
        letSpeak(bello);
        Cat tom = new Cat();
        letSpeak(tom);
    }
}
```

Listing 1.3: Beispiel für Polymorphie

Je nachdem, ob ein Hund oder eine Katze übergeben wird, werden unterschiedliche Laute ausgegeben. Die Ausgabe des Programms ist somit:

```
bark
miaow
```

Listing 1.4: Beispiel für Polymorphie – Ergebnis

Das Ganze funktioniert auch mit abstrakten und konkreten Klassen, wie die folgende Erweiterung unseres Beispiels verdeutlicht.

Hinweis

Die Definitionen aus dem vorigen Beispiel werden unverändert übernommen, das deuten die Punkte `...` an. Alles, was neu hinzugekommen ist, ist fett gedruckt.

```
public class AnimalSounds {
    ...

    static class SleepingCat extends Cat {
        public void speak() {
            System.out.println("zzz");
        }
```

```
    }

    static void letSpeak(Animal animal) {
        animal.speak();
    }

    public static void main(String[] args) {
        Dog bello = new Dog();
        letSpeak(bello);
        Cat tom = new Cat();
        letSpeak(tom);
        Cat garfield = new SleepingCat();
        letSpeak(garfield);
    }
}
```

Listing 1.5: Erweitertes Beispiel für Polymorphie

Die Ausgabe des Programms ist jetzt natürlich:

```
bark
miaow
zzz
```

Listing 1.6: Erweitertes Beispiel für Polymorphie – Ergebnis

Die Polymorphie funktioniert auch für Rückgabeparameter, so habe ich `garfield` gleich als `Cat` und nicht als `SleepingCat` definieren können. Java sorgt dafür, dass immer die richtige Methode in der Vererbungshierarchie aufgerufen wird.

Wir werden in den folgenden Kapiteln noch viele Beispiele für die Anwendung dieses Paradigmas sehen. So werden wir beispielsweise bestehendem Code »gefälschte« Objekte unterjubeln, um diesen Code in einem Test ausführen zu können. In Java gibt es den Begriff Polymorphie noch im Zusammenhang mit Methoden. Es können ja Methoden mit demselben Namen in Java mehrfach definiert werden, sofern sie nur unterschiedliche Parametersignaturen haben (die Anzahl und/oder die Datentypen müssen unterschiedlich sein, damit man Methoden gleichen Namens eindeutig unterscheiden kann). Das wird ebenfalls als Polymorphie, nämlich als *Methodenpolymorphie* bezeichnet und dieses Mal stimmt der Vergleich mit der unterschiedlichen Gestalt auch.

Ein Beispiel für Polymorphie im Zusammenhang mit Methoden:

```
public class MethodPolymorphism {
    static int sum(int a, int b) {
        return a + b;
```

```
    }

    static int sum(int a, int b, int c) {
        return a + b + c;
    }

    static double sum(double a, double b) {
        return a + b;
    }

    public static void main(String[] args) {
        System.out.println(sum(1, 2));
        System.out.println(sum(1, 2, 3));
        System.out.println(sum(1.5, 2.5));
    }
}
```

Listing 1.7: Methodenpolymorphie

Obwohl `sum` dreimal definiert wird, findet Java anhand der Parametersignatur die richtige Methode. Die Ausgabe lautet richtigerweise:

```
3
6
4.0
```

Listing 1.8: Methodenpolymorphie – Ergebnis

1.3 Vorteile der objektorientierten Programmierung

Es sei hier ausdrücklich betont, dass man mit einer Universalsprache (alle gängigen Programmiersprachen sind Universalsprachen) keine Funktionalität implementieren kann, die man nicht auch mit einer anderen implementieren könnte, egal, welchem Paradigma die eine oder die andere folgt. Das lässt sich leicht beweisen: Man könnte ja jederzeit einen Interpreter der einen Sprache in einer anderen schreiben, wodurch die Sprache, in der der Interpreter implementiert ist, alle Fähigkeiten der anderen Sprache übernimmt. Der Vorteil der objektorientierten Programmierung liegt eindeutig in der hohen *Flexibilität*, die objektorientierte Programme erreichen können. Wie wir noch sehen werden, können objektorientierte Programme leichter automatisierten Tests unterzogen werden. Auch reduzieren sich die Auswirkungen von Änderungen auf üblicherweise wenige Klassen, wohingegen bei prozeduralen Programmen schon kleine Erweiterungen umfangreiche Änderungen an vielen Stellen im Quelltext zur Folge haben.

1.4 Die mathematische Theorie der abstrakten Datentypen

Abstrakte Datentypen (kurz ADTs) sind eine mathematische Theorie, die die zuvor beschriebenen Prinzipien mathematisch beschreibt. Man kann sich einen abstrakten Datentyp als mathematisches Pendant zu einem Typ eines Objekts – also einer Klasse – vorstellen. Für jeden abstrakten Datentyp werden Funktionen definiert, die an den ADT gebunden sind. Diese Funktionen entsprechen den Methoden in der objektorientierten Programmierung. Im Gegensatz zur objektorientierten Programmierung wird das Verhalten und der Zustand nicht direkt durch die Implementierung von Zustand und Methoden definiert, sondern durch Anfangsbedingungen und Axiome – sozusagen »von außen« – festgelegt. Anfangsbedingungen legen dabei den anfänglichen Zustand des ADTs fest. Dies geschieht dadurch, dass man das Ergebnis von Abfragefunktionen (das sind Funktionen, die Informationen über den internen Zustand des ADTs zurückgeben, den Zustand selbst jedoch unverändert lassen) festlegt, noch bevor der interne Zustand des ADTs verändert wurde. Axiome legen das Verhalten des ADTs durch unterschiedliche Kombinationen von Aufrufen der einzelnen Funktionen und das Vorschreiben der Ergebnisse dieser Aufrufkette fest. Es werden genauso viele Axiome definiert, wie für die vollständige Beschreibung des Verhaltens des ADTs nötig sind. Da der innere Zustand eines ADTs gar nicht explizit definiert wird, ist die Eigenschaft der Kapselung automatisch gegeben. Die Kenntnis der Theorie der abstrakten Datentypen ist für die objektorientierte Programmierung allerdings nicht wirklich nötig. Es ist jedoch bemerkenswert, dass man das Verhalten von Datentypen allein aus der Kommunikation mit ihrer Umwelt vollständig definieren kann, ohne sich über die Implementierung Gedanken machen zu müssen. In der objektorientierten Programmierung geht man allerdings nicht so weit. Hier wird die Funktion eines Objekts durch seine Programmierung definiert.

1.5 Die objektorientierte Modellierung

Gestärkt von der Popularität der objektorientierten Programmierung übernahmen Experten auf dem Gebiet der Datenmodellierung das Prinzip der Vererbung, mischten es mit Aspekten der relationalen Modellierung (Modellierung der Beziehungen der Objekte zueinander) und nannten das ganze *objektorientierte Modellierung*. Es ist äußerst wichtig, hier zwischen der objektorientierten Programmierung und der objektorientierten Modellierung zu unterscheiden, denn bei Letzterer wird ein Objekt im Sinne der Mengenlehre als Summe seiner *Eigenschaften* und Beziehungen zu anderen Objekten verstanden. Dieser Ansatz eignet sich jedoch nicht wirklich für die Entwicklung von objektorientierten Programmen, da die Frage der richtigen Aufteilung der Logik unter den Objekten mit diesem Ansatz nicht gelöst werden kann, ganz im Gegensatz zum kommunikationsbasierten Ansatz in der objektorientierten Programmierung.

Hinweis

Ein Übungsbeispiel für Hartgesottene: Erstellen Sie ein einfaches Klassendiagramm für ein kleines Programm mit einem CASE-Tool Ihrer Wahl (zum Beispiel IBM Rational Clear Case oder Borland Together) und lassen Sie es den Code dazu generieren. Programmieren Sie daraufhin das Programm fertig, ohne dass Sie Klassen, Felder oder Methoden hinzufügen, abändern oder entfernen. Wenn Sie das schaffen und das Programm kein vollkommenes Chaos ist, gehören Sie zu den wenigen Auserwählten, die das können. Ich für meinen Teil und die Mehrzahl der weniger begabten Entwickler werden das Modell schon bald verwerfen und im Wesentlichen neu, diesmal aber ohne CASE-Tool entwickeln.

Diese unterschiedlichen Definitionen des Wortes »Objektorientierung« führen leider bis heute zu Missverständnissen. Um den Bedürfnissen der objektorientierten Programmierung hinsichtlich der Kommunikation nachzukommen, wurden Interaktionsdiagramme im Nachhinein in die Unified Modelling Language (UML) aufgenommen, die zuvor nur Diagramme zur objektorientierten Modellierung (Klassendiagramme etc.) kannte. Verstehen Sie mich nicht falsch: Klassendiagramme sind auch im Zusammenhang mit der objektorientierten Programmierung durchaus nützlich, sie sollten aber besser nicht als Ausgangsbasis für ein objektorientiertes Programm verwendet werden. Man kann damit nämlich keine Interaktion zwischen Objekten modellieren, doch diese Interaktion ist ja eigentlich die Essenz der objektorientierten Programmierung.

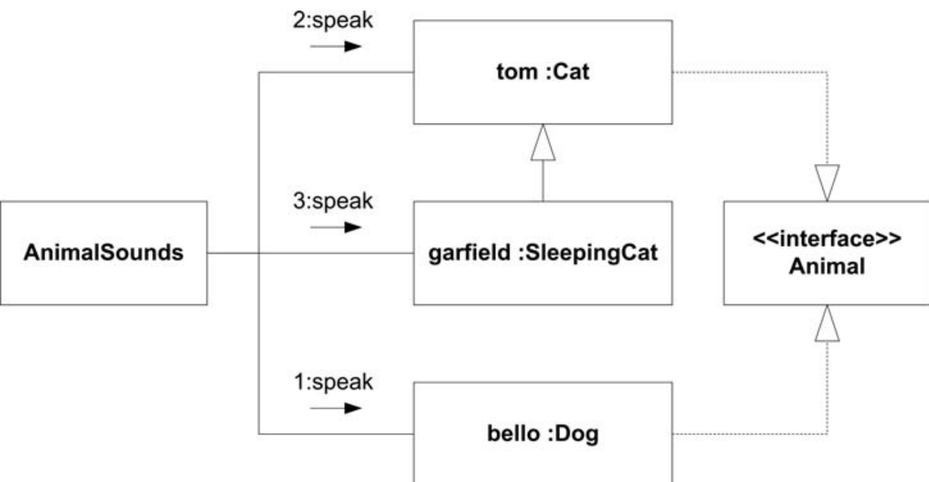


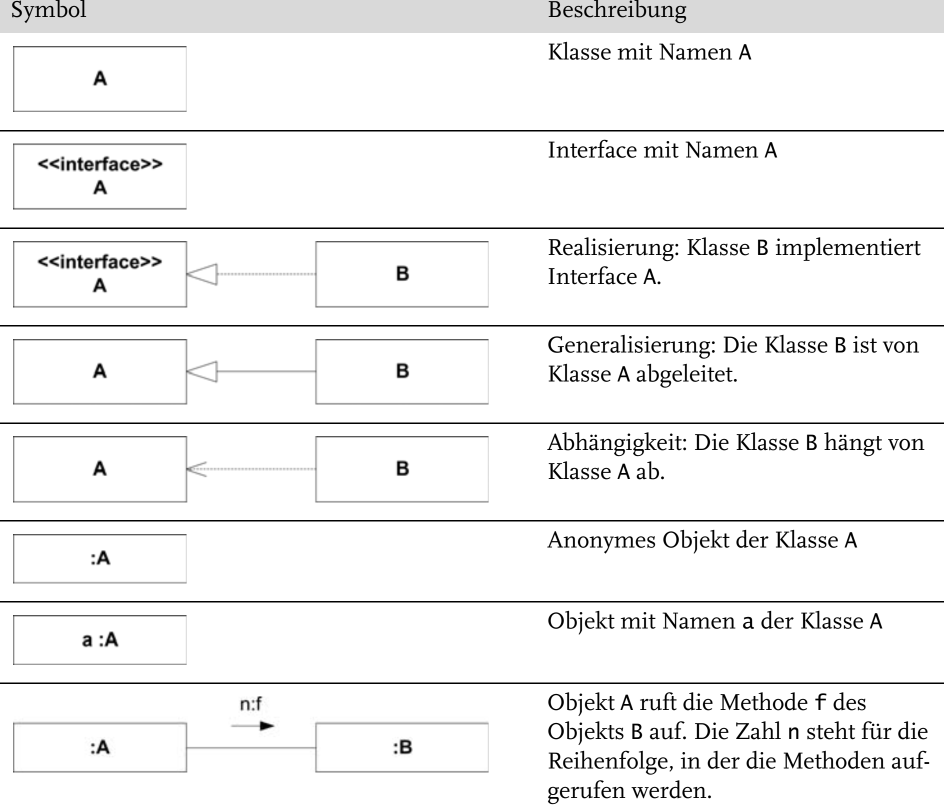
Abb. 1.1: Beispiel für ein Kommunikationsdiagramm

Tipp

Sie können einen erfahrenen von einem weniger erfahrenen objektorientierten Programmierer unter anderem dadurch unterscheiden, welches Diagramm er als Erstes zeichnet. Beginnt er mit einem *Kommunikations-* oder *Sequenzdiagramm*, hat er die objektorientierte Programmierung verstanden. Beginnt er mit einem *Klassendiagramm*, gibt es wahrscheinlich noch Schulungsbedarf.

1.6 UML

Ich möchte hier auf die ausführliche Beschreibung von UML verzichten, da es zahlreiche exzellente Bücher zu diesem Thema gibt. In der folgenden Tabelle finden Sie nur jene UML-Symbole, die für das Verständnis der UML-Diagramme in diesem Buch nötig sind.



Symbol	Beschreibung
A	Klasse mit Namen A
<<interface>> A	Interface mit Namen A
<<interface>> A ◁┈┈ B	Realisierung: Klasse B implementiert Interface A.
A ◁── B	Generalisierung: Die Klasse B ist von Klasse A abgeleitet.
A ←┈┈ B	Abhängigkeit: Die Klasse B hängt von Klasse A ab.
:A	Anonymes Objekt der Klasse A
a :A	Objekt mit Namen a der Klasse A
:A ── :B (n:f →)	Objekt A ruft die Methode f des Objekts B auf. Die Zahl n steht für die Reihenfolge, in der die Methoden aufgerufen werden.

Tabelle 1.1: Auswahl der UML-Symbole

Symbol	Beschreibung
A (abgegrenzter Bereich)	Abgegrenzter Bereich mit Namen A
A (Zustand)	Zustand A
●→	Anfangs-Pseudoszustand
A —b→ B	Zustand A geht über in den Zustand B, wenn Bedingung b erfüllt ist.

Tabelle 1.1: Auswahl der UML-Symbole (Forts.)

1.7 Das Problem mit dem Umlernen

Obwohl die objektorientierte Programmierung bereits ein Methusalem in punkto Maßstab der schnelllebigen Informationstechnologie ist, haben viele Programmierer Probleme, sich mit der objektorientierten Programmierung anzufreunden. Ein Grund dafür ist häufig die langjährige Erfahrung der Programmierer mit der prozeduralen Programmierung, die sie dazu verleitet, jedes Problem in dieser Richtung zu abstrahieren. In der prozeduralen Programmierung ist nämlich die grundsätzliche Herangehensweise eine völlig andere als in der objektorientierten. Die erste Frage, die sich ein prozeduraler Programmierer stellt, ist: »Was ist zu tun?« Diese Frage beantwortet er mit einer Reihe von Aktivitäten, die ausgeführt werden müssen, um zum Ziel, nämlich der Lösung des Problems zu gelangen. Damit wird das Programm eine Art TODO-Liste. Man kann sich so ein Programm auch wie ein Kochrezept für eine Maschine vorstellen. Da eine Maschine natürlich keine Intelligenz im menschlichen Sinne besitzt, müssen die Anweisungen extrem detailreich sein und dürfen keinerlei Vorbildung oder Mitdenken voraussetzen.

In der objektorientierten Programmierung lautet die erste Frage hingegen: »Wer tut was?« Die Arbeit wird an hochspezialisierte Objekte delegiert, die miteinander kommunizieren. Wir könnten die Frage auch anders als »Wer tut was mit wem?« formulieren, um den Kommunikationscharakter zu verdeutlichen. Die Anweisungen sind immer als Kommunikation zwischen Objekten zu verstehen. Es gibt ein Objekt, das Anweisungen ausgibt und einen Adressaten dieser Anweisung, der dasselbe oder ein anderes Objekt ist. Das Kochrezept wird also nicht als eine Reihe von Anweisungen für einen Koch verstanden, sondern als Anweisungen für einzelne, hochspezialisierte Küchenhelfer. Jeder dieser Helfer bekommt einen Teil

des Rezeptes, in dem nur der für ihn wichtige Teil dokumentiert ist. (Der Spruch »viele Köche verderben den Brei« gilt hier nur unter der Voraussetzung, dass die einzelnen Tätigkeiten der Köche nicht koordiniert sind oder sich Zuständigkeiten überlappen. Wir widmen uns diesem Problem im Zusammenhang mit dem Einzelzuständigkeitsprinzip im Kapitel »Entwurfsprinzipien der objektorientierten Programmierung«). Ein objektorientiertes Programm kann man sich auch als eine Art Theaterstück für mehrere Schauspieler vorstellen – daher der Begriff »Akteur« – während ein prozedurales eher einem Ein-Personen-Stück gleicht. Es sei hier natürlich darauf hingewiesen, dass jede Analogie ihre Grenzen hat. Analogien können jedoch dabei helfen, individuelle Strategien zu finden, um abstrakte Sachverhalte einfacher zu verstehen.

1.8 Das Problem mit der Zweckentfremdung von Programmiersprachen

Viele Programmierer behalten die grundsätzlichen Denkmuster bei, wenn sie von einer Programmiersprache auf eine andere umlernen. So versuchen viele Programmierer, die objektorientierte Programmierung erst gar nicht zu verstehen und programmieren so weiter, wie sie es in prozeduralen Sprachen gemacht haben. Leider stößt das bei manchen objektorientierten Programmiersprachen auf Grenzen. Inwiefern man in einer objektorientierten Programmiersprache sinnvoll prozedural programmieren kann, hängt nämlich davon ab, inwieweit die Programmiersprache die prozedurale Programmierung durch geeignete Sprachmittel unterstützt. So ist es hilfreich, wenn man Prozeduren und Funktionen außerhalb von Klassen definieren kann. Darüber hinaus sollte man eigene Datentypen definieren können, die keine Klassen oder Interfaces sind. Aus all dem ergibt sich die Einsicht, dass sich Java schlecht für die prozedurale Programmierung eignet. Das geht noch eher mit C++, C# oder Object-Pascal (das sich auch in Delphi oder Free-Pascal findet).

Das Fehlen von prozeduralen Sprachmitteln in Java ist allerdings nur dann ein Problem, wenn man unbedingt an der prozeduralen Programmierung festhalten möchte. Dafür gibt es jedoch keinen zwingenden Grund, da es keinesfalls schwieriger ist, objektorientiert zu programmieren als prozedural. Im Gegenteil: In größeren Programmen, die regelmäßig modifiziert und erweitert werden müssen, fällt die objektorientierte Programmierung viel leichter als die prozedurale. Um noch einmal auf das Kochrezept zurückzukommen: Wollen Sie das prozedurale Kochrezept ändern, müssen Sie dem Koch ein neues Exemplar zur Verfügung stellen. Da das objektorientierte Kochrezept aufgeteilt ist, genügt es meist, wenn Sie nur an einige Küchenhelfer neue Rezepte austeilen. Die anderen können die alten weiterverwenden. Änderungen bleiben dadurch auf einige Objekte (Küchenhelfer) begrenzt, dadurch skaliert der objektorientierte Ansatz besser. Hier stößt die Ana-

logie allerdings an ihre Grenzen, da Programme um Größenordnungen komplexer sind als Kochrezepte. Wir haben es schon in kleineren Programmen mit einer ganzen Armee von Helfern zu tun – und hier ist es ganz offensichtlich von Vorteil, wenn Änderungen auf möglichst wenige Helfer (also Objekte) begrenzt bleiben.