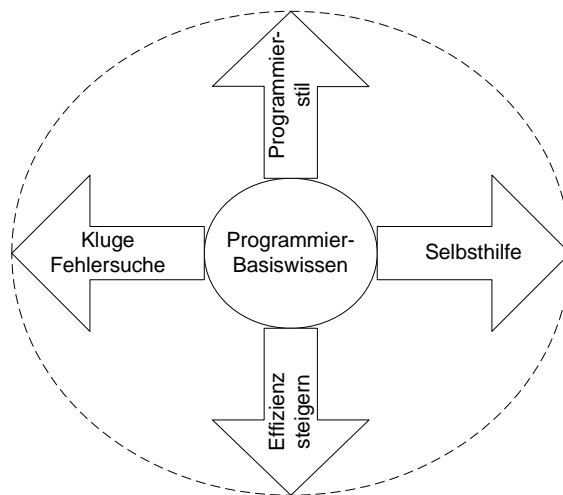


## 4 Richtiges & falsches Perl

### 4.1 Worum geht's?

Von der ersten Seite bis hierher haben wir die Grundlagen von Perl und die der Modellierung gelernt. Damit können wir Programmieren. Oder? Ja und Nein. Zweifellos wissen Sie nun, wie man eine Problemlösung modelliert, wie man Daten einsetzt und wie man Abläufe in korrekte Algorithmen umsetzt. Was noch fehlt, ist eine Richtung zu zeigen, in die Sie Ihren *Programmierstil* entwickeln sollten: Man kann sehr gut Programmieren können und dennoch einen sehr schlechten Stil haben. Das sollten Sie vermeiden; aber nicht, weil Stil ein Wert an sich ist (was nach unserer Ansicht durchaus stimmt), sondern weil es Ihnen handfeste Vorteile bringt: Wer verständlich programmiert, macht weniger inhaltliche Fehler, diese Fehler sind leichter zu finden und die Programme sind üblicherweise effizienter als »gehackte« Lösungen.

*Programmierstil*



**Abbildung 4.1**  
*Programmier-Basiswissen intelligent erweitern*

Abbildung 4.1 illustriert die Ziele dieses Kapitels. Durch kluge Erweiterung des bisher erworbenen Basiswissens wollen wir den Mehrwert für Ihr Können als ProgrammiererIn maximieren. Im Detail wer-

*Überblick über das Folgende*

den wir uns mit den folgenden Problembereichen beschäftigen: In Abschnitt 4.2 werden wir uns mit gutem Programmierstil an sich beschäftigen. Wir werden uns damit auseinandersetzen, wie man Programme stilvoll gliedert, welche Befehle man verwendet (und welche nicht) und wie Muster in Programmen eingesetzt werden. Abschnitt 4.3 diskutiert die Hauptbeschäftigung beim Programmieren: Fehlersuche. Neben effizienten Verfahren zur Fehlereingrenzung und typischen Perl-Fehlern werden wir ausdrücklich auf *Verfahren zur Selbsthilfe* eingehen. Wenn man gelernt hat, sich selbst von Problemen zu befreien, ist in der Tat alles nur noch halb so schlimm! Zudem basiert wirkungsvolle Selbsthilfe auf nur wenigen, einfachen Regeln. Abschnitt 4.4 schließlich zeigt, wie man Programme schneller machen kann, ohne den Anspruch an guten Stil zu verlieren.

## 4.2 Programme richtig formulieren

### 4.2.1 Beispiele für richtig und falsch

Sehen wir uns zunächst an, wie man es nicht machen sollte. Es gibt eine Menge Möglichkeiten, um einen schlechten Algorithmus zu programmieren: Man kann ihn zum Beispiel über und über verschachteln oder gar nicht gliedern, jede Funktion viel zu lang machen und mehrere Aufgaben in eine Funktion packen. Dafür kann man dann dieselbe Aufgabe in mehrere Funktionen stecken. Man kann einfache Bedingungen unglaublich komplex formulieren, die Namen und Datentypen von Variablen schlecht wählen und vieles andere mehr. Ein Beispiel: Der folgende Algorithmus gibt alle Zeichenketten einer Liste aus. Dabei wandelt er den Anfangsbuchstaben aller Zeichenketten, die mit `b` beginnen, in ein `B` um:

*Ein unschöner  
Algorithmus*

```
my @hh = ("test", "knödel", "bratsche", "buchsbaum", "tofu");
my $string;
my $x=0;
do {
    $string = $hh[$x++];
    $string =~ /^b/ && $string =~ s/^(.)/\U$1\E/;
    print $string, "\n";
} while($x<5);
```

Unglaublich hässlich! `@hh` ist kein Name; `@wortListe` wäre schon besser. Der gewählte Schleifentyp ist sehr unglücklich: Was, wenn die Liste nicht fünf Elemente enthält? Außerdem ist der Zähler unnötig. `foreach` ist bei Schleifen über eine Liste variabler oder unbekannter Länge fast

immer die ideale Wahl. Drittens ist es eine sinnlose Unart, das Inkrement des Zählers in die Zuweisung zu packen. Zudem ist der Ausdruck zum Umwandeln des Anfangsbuchstabens zwar technisch korrekt, aber ebenso unnötig kompliziert wie sinnlos: `if` durch eine Kette von »und«-Operatoren zu ersetzen ist zwar möglich (der zweite Ausdruck wird nur ausgeführt, wenn der erste zu »wahr« evaluiert), bringt außer der Befriedigung persönlicher Eitelkeit aber keinen Nutzen. Der reguläre Ausdruck (*Regex*) zum Ersetzen des Anfangsbuchstabens ist ebenfalls eine Sache für sich. Sehen wir uns folgende Lösung an:

```
my @wortListe=("test", "knödel", "bratsche", "buchsbaum", "tofu");

foreach my $wort (@wortListe) {
    $wort =~ s/^b/B/;
    print "$wort\n";
}
```

*Schöner und schneller*

Sie erfüllt denselben Zweck, ist aber kürzer und viel einfacher. Da wir uns ohnehin nur für mit `b` beginnende Wörter interessieren, ersetzen wir stur jedes Auftreten durch `B`. Der im ersten Algorithmus gewählte Ansatz wäre nur vorzuziehen, wenn wir auch andere Anfangsbuchstaben ersetzen möchten.

Sehen wir uns noch ein Beispiel an; diesmal zum Thema *Gliederung*. Der nachfolgende Algorithmus gibt paarweise Namen und Orte aus (zwei Listen). Dabei werden jeweils die Anfangsbuchstaben auf Großbuchstaben konvertiert.

*Gute Gliederung*

```
my @namen = ("frida", "thor", "helga", "zottel");
my @orte = ("walhallein", "walhallali", "walter", "wald");

# 1. Große Anfangsbuchstaben
my @grosseOrte = kleinAufGross(@orte);
my @grosseNamen = kleinAufGross(@namen);

# 2. Paare von Namen und Orten sortiert ausgeben
foreach my $name (sort @grosseNamen) {
    # 2.1 Position des Ortes in der Liste suchen
    my $pos=0;
    while($grosseNamen[$pos] ne $name) {
        $pos++;
    }

    # 2.2 Paare ausgeben
    print $name, ", ", $grosseOrte[$pos], "\n";
}
```

```

sub kleinAufGross {
    my $wort;
    my @wortListe=();

    foreach $wort (@_) {
        $wort =~ s/^(.)/\U$1\E/;
        push(@wortListe, $wort);
    }

    @wortListe;
}

```

An sich ein schönes Programm. Das Problem besteht darin, dass durch das Sortieren der Namen die Bindung von Namen und Orten (über den Listenindex) verloren geht. Deshalb muss die Indexexposition in Block 2.1 wiedergefunden werden. Folgender Algorithmus löst dasselbe Problem eleganter:

```

my @namen = ("frida", "thor", "helga", "zottel");
my @orte = ("walhallein", "walhallali", "walhter", "wald");

my @paare = ();
for(my $index=0; $index <= $#namen; $index++) {
    push(@paare, $namen[$index] . ", " . $orte[$index]);
}

foreach my $paar (sort @paare) {
    print kleinAufGross($paar), "\n";
}

sub kleinAufGross {
    my $wort = shift;

    $wort =~ s/((^\s)/\U$1\E/g;

    $wort;
}

```

Aus dem ersten Algorithmus haben wir gelernt, dass die Hauptarbeit im Finden von Name-Ort-Paaren bestand. Deshalb ist das auch das Erste, was wir im verbesserten Algorithmus tun (gespeichert in der Liste @paare). Der Rest des Algorithmus ergibt sich zwangsläufig: Paare lassen sich leicht sortieren und die Konvertierfunktion muss dazu in der Lage sein, mehrere Konvertierungen in einer Zeichenkette durchzuführen

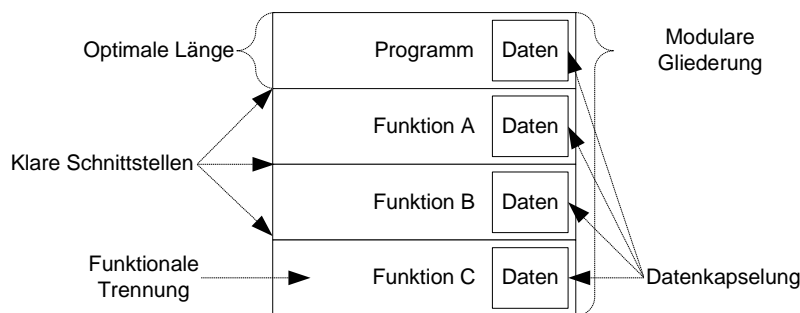
(nach dem Beginn und nach Leerzeichen). Wichtig ist: *Verbessern Sie Ihre Algorithmen iterativ (das nennt man auch Refactoring) und gliedern Sie immer zuerst den Teil eines Algorithmus neu, der die höchste Komplexität hat!*

Außerdem sollten Sie folgende Lehren aus diesem Abschnitt ziehen: Vermeiden Sie sinnlose Komplexität und schwer verständliche Abläufe. Vermeiden Sie insbesondere implizite Anweisungen (wie Funktionen in Bedingungen, Inkrement in Zuweisungen et cetera)! Wichtig ist auch, Variablen sprechend zu benennen und stets zu versuchen, Variablen durch Umstellung der Anweisungsfolge zu vermeiden. Im nächsten Abschnitt werden wir uns mit Heuristiken (Daumenregeln) beschäftigen, die zu klug gegliederten Algorithmen führen.

*Schlussfolgerungen*

### 4.2.2 Möglichkeiten der Gliederung und Modularisierung

Gibt es Regeln, wie man gute Blöcke definiert? (Wir sprechen bewusst von Blöcken und nicht von Funktionen. »Block« ist im Folgenden jeder inhaltlich zusammengehörende Bereich eines Programms.) Gute Blöcke zu definieren, ist auch wichtig für das Testen. Wir werden in späteren Abschnitten sehen, dass Testen im Wesentlichen aus dem Eingrenzen von Fehlern besteht. Eingrenzen erfordert, dass zusammengehörende Abläufe (textuell betrachtet) nahe beieinander stehen.



**Abbildung 4.2**  
*Regeln für gute Programm-Modellierung*

Abbildung 4.2 zeigt, welche Eigenschaften ein gut strukturiertes Programm haben sollte. Als Faustregel gilt: *Ein guter Block enthält alles, was inhaltlich dazu gehört, nichts anderes und ist dabei nicht zu lang.* Geht das überhaupt immer? Nach unserer Erfahrung ja (meistens), wenn man ein paar einfache Regeln beherzigt (lernt und bewusst umsetzt):

Modulare Gliederung eines Algorithmus erreicht man primär durch funktionale Gliederung. Das heißt, die Teile der Lösung, die im Rahmen

*Modularer Aufbau*

der Analyse identifiziert wurden, werden in getrennten Funktionen implementiert. Jede Funktion wird nur *einmal* implementiert. Ausnahmen sind nur dort zulässig, wo Funktionen durch große Datenmengen miteinander verbunden sind (zum Beispiel, wenn mehrdimensionale Listen zwischen Funktionen ausgetauscht werden müssten). Es empfiehlt sich wieder einmal eine iterative Vorgangsweise: *Gliedern Sie Ihre Programme zunächst nach rein analytischen Gesichtspunkten. Suchen Sie dann im zweiten Schritt nach komplexen Bereichen (verschachtelte Schleifen, große Datenmengen als Parameter et cetera) und versuchen Sie, diese Komplexitäten durch Umstellung der Funktionen und Blöcke zu beseitigen.* Wiederholen Sie das Umstellen, bis Sie das Gefühl haben, dass keine weitere Verbesserung mehr erreichbar ist. Auch hier ist Erfahrung wichtig: Je öfter Sie üben, umso besser werden Ihre Gliederungen!

#### Datenkapselung

Zur modularen Gliederung gehören zudem *Datenkapselung* und *klare Schnittstellen*. Diese beiden Punkte gehen Hand in Hand. Als Schnittstelle (*Signatur*) einer Funktion bezeichnet man die Summe ihrer Parameter und Rückgabewerte. Datenkapselung bedeutet in Perl die sture Verwendung von `my`. Widerstehen Sie der Verlockung, in Funktionen auf globale Variablen direkt zuzugreifen. Übergeben Sie sie stattdessen als Parameter an geschützte Variablen. Ebenso sollten Sie mit den Rückgabewerten verfahren. Der Gewinn lohnt den Aufwand: Schwer zu findende Fehler, wie sie durch unerlaubtes Überschreiben von Variablen verursacht werden können, werden durch dieses Verhalten unmöglich gemacht.

#### Länge von Blöcken

Das letzte Kriterium für gute Modularisierung ist die richtige Länge von Blöcken. Ein Block sollte zumindest aus drei Anweisungen bestehen und nicht länger als etwa dreißig Zeilen sein. Freilich ist es nicht immer leicht, diese Regel einzuhalten. (Manchmal ist es auch kontraproduktiv.) Zumeist ist es aber möglich, in einer zu lang geratenen Funktion wiederum inhaltlich zusammengehörende Blöcke zu identifizieren und die Funktion durch eine Reihe von Unterfunktionen zu ersetzen. Der dadurch gewonnene Vorteil der Übersichtlichkeit ist während des Testens Gold wert.

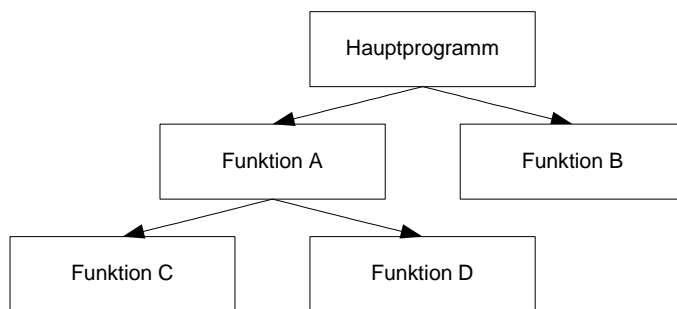
#### Funktionsgraphen

Zur Visualisierung modularer Strukturen gibt es – neben den bereits erwähnten Flussdiagrammen – eine sehr alte, einfache und intuitive Methode: Funktionsgraphen. Ausgehend vom Hauptprogramm zeigt ein Funktionsgraph alle funktionalen Abhängigkeiten: Hauptprogramm und Funktionen werden als Kästchen dargestellt, die durch Pfeile verbunden sind. Folgendes Programm lässt sich zum Beispiel durch den Funktionsgraphen in Abbildung 4.3 darstellen.

```
# Hauptprogramm
&funktionA;
&kfunktionB;

# Funktionen
sub funktionA {
    &kfunktionC;
    &kfunktionD;
}

sub funktionB {}
sub funktionC {}
sub funktionD {}
```

**Abbildung 4.3**

Funktionsgraph für das Beispiel

Schließlich ist es wichtig, Blöcke auch textuell zu visualisieren. Rücken Sie jeden Block (alles, was innerhalb geschwungener Klammern steht) zumindest um drei Leerzeichen ein und fügen Sie vor inhaltlichen Übergängen im Programm zumindest eine Leer- und eine Kommentarzeile ein. Trennen Sie das Hauptprogramm und die Funktionen durch geeignete Kommentare.

### 4.2.3 Besondere Anweisungen

Moderne Programmiersprachen enthalten keinen redundanzfreien Satz von Anweisungen, sondern bieten oft mehrere Wege zur Lösung eines Problems. Das hat im Wesentlichen kommerzielle Gründe: Natürlich muss C# alle Sprachkonstrukte von Java enthalten, ebenso jene der früheren Microsoft-Hauptsprachen (Visual C++, Visual Basic). Java enthält wiederum eine Menge Konzepte aus C++, Smalltalk und anderen Sprachen. Bietet eine Sprache eine Obermenge der Möglichkeiten anderer Sprachen, so ist sie diesen überlegen – so die Milchmädchenrechnung. In der Praxis heißt das *Featureritis*. Damit bezeichnet man

das Überladen eines Systems mit »das sollte es auch noch können«-Fähigkeiten. Featureritis bringt ein wesentliches Problem mit sich: Das befallene System wird unübersichtlich und schwer zu handhaben.

*Featureritis in Perl?*

Perl leidet nicht unter sehr starker Featureritis. Die Entwickler hatten eine klare Vorstellung von ihrem Ziel. Dennoch bietet auch Perl mehrere Lösungswege für viele Probleme. Ein paar von ihnen werden wir uns jetzt ansehen. Generell empfehlen wir folgenden Ansatz: *Beschränken Sie sich bei Ihren Programmen auf ein elementares Set von Anweisungen!* Was Sie bisher gelernt haben, reicht leicht zum Lösen der meisten Programmierprobleme aus. Durch freiwillige Selbstbeschränkung im Algorithmus erreichen Sie Übersichtlichkeit und erhöhen die Wiederverwendbarkeit. Indem Sie *Ihren* Satz an Anweisungen besser kennen, werden Sie auch sicherer und schneller programmieren. Wir nennen diesen Ansatz *Elementary Programming*.

*Elementary  
Programming*

Sehen wir uns ein Beispiel an, in dem wir Anweisungen verwenden, die wir bisher noch nicht kennen gelernt haben. Dieser Algorithmus dient dazu, Listen von Wörtern, die per Telegraph übertragen wurden, in Sätze umzuwandeln (getrennt durch stop) und auszugeben. Initialisiert wird das Ausgabeprogramm über das Kommando `init`:

*Einige besondere  
Schlüsselwörter im  
Einsatz*

```
my @nachricht = ("init", "lotto", "gewonnen", "stop",
    "millionärin", "stop", "mit", "scheidung", "einverstanden",
    "stop", "gruss", "an", "geliebte", "stop");
my @saetze = ();
my $index = 0;

# 1. Wörter in Sätze umwandeln
foreach my $wort (@nachricht) {
    if ($wort =~ /^init$/) {
        @saetze = ();
        $index = 0;
        $saetze[$index] = "";
        next;
    }

    if ($wort cmp "stop") {
        $saetze[$index] .= $saetze[$index] =~ /^$/ ? $wort : " $wort";
    } else {
        $saetze[$index] .= ".";
        $index++;
        $saetze[$index] = "";
    }
}
$saetze[$index] = "Ende der Nachricht.";
```

```
# 2. Ausgeben
$index = 0;
while(1) {
  unless($saetze[$index] =~ /^Ende der Nachricht/) {
    print $saetze[$index], "\n";
    $index++;
  } else {
    last;
  }
}
```

Zunächst wird die Nachricht (in @nachricht) in Sätze umgewandelt. Der erste Block fängt das `init`-Kommando ab und initialisiert die Satzliste und einen Zähler. Danach wird mit `next` zur nächsten Iteration der Schleife (dem nächsten Wort) gesprungen. Das heißt, der Rest des Blocks wird nicht ausgeführt. Für alle übrigen Wörter wird mit `cmp` festgestellt, ob sie sich vom Wort `stop` unterscheiden. Falls ja, wird dem Satz ein neues Wort hinzugefügt. `<bedingung> ? <wert1> : <wert2>;` nennt man den tertiären Operator. Wenn die Bedingung »wahr« ist, wird der erste Wert, sonst der zweite in der Zuweisung verwendet. In unserem Beispiel werden so die Leerzeichen zwischen den Wörtern eingefügt. Am Ende der Nachricht wird schließlich noch ein gleichlautender Satz eingefügt.

*Tertiärer Operator*

Im zweiten Teil des Programms werden alle Sätze außer dem Ende der Nachricht ausgegeben. `unless` überprüft, ob die angegebene Bedingung »nicht wahr« ist. Falls dem so ist, wird ein Satz ausgegeben. Falls die Bedingung aber »wahr« ist, wird die `while`-Endlosschleife mit `last` abgebrochen.

Alle besprochenen Anweisungen lassen sich leicht ersetzen: `next` und `last` durch Umstellungen im Algorithmus, `unless` durch `if` und den Negationsoperator, der tertiäre Operator durch `if` und `cmp` durch einen passenden regulären Ausdruck. Wir raten nochmals von der Verwendung dieser Anweisungen ab: Sie sind überflüssig. Dennoch sollten Sie sie kennen, um die Programme anderer verstehen zu können.

#### 4.2.4 Muster und Nicht-Muster

*Muster* sind eine andere, höhere Form der freiwilligen Selbstbeschränkung beim Programmieren. »Muster« sind Modelle, die sich in der Softwareentwicklung bewährt haben und in abstrakter Form zur Verwendung empfohlen werden. Die Verwendung von Mustern bietet zwei wesentliche Vorteile:

Vorteile der  
Verwendung von  
Mustern

1. Das in der Analyse angestrebte Verstehen findet bereits auf einer höheren Ebene statt. Kennt man viele Programmiermuster, ist man eher dazu in der Lage, ein Problem in bekannte Teilprobleme zu zerlegen. Die Implementierung der Muster muss nicht mehr selbst gefunden werden, sondern liegt bereits (in Form von Datenbanken, Büchern et cetera) vor.
2. Muster sind eine Form der Kommunikation (ein Code) zwischen Entwicklern. Basiert eine Problemlösung auf einem gängigen Muster, so ist es nicht mehr notwendig, den genauen Aufbau der Lösung mitzuteilen. Es genügt, einer interessierten Person den Namen des verwendeten Musters mitzuteilen. Dadurch wird die Kommunikation (Dokumentation et cetera) effizienter, öfter eindeutig und besser verständlich.

Maßgeschneiderte  
Anwendung

Wir können zwei Arten von Mustern unterscheiden: Positiv-Muster und Negativ-Muster. Negativ-Muster beschreiben, wie man etwas nicht lösen sollte. Auf einer sehr hohen Ebene sind die Beispiele, die wir in den beiden Abschnitten oben gezeigt haben, Negativ-Muster für die Programmierung. Die überwiegende Zahl der Muster sind aber Positiv-Muster. Die Autoren von [13] haben einen umfangreichen Katalog an Ablauf- und Datenmustern für die objektorientierte Programmierung vorgelegt. [25] und [35] bieten Muster für das Sortieren von Daten und Index-Datenstrukturen an (das sind *die* Klassiker der Muster). Einen Katalog von Perl-Mustern finden Sie unter anderem unter [47]. Im Anhang A.3 haben wir einen dokumentierten Katalog praktischer Perl-Muster erstellt, die wir immer wieder verwenden. Beim Einsatz von Mustern ist ein Punkt wesentlich: *Jedes Muster muss erst in seine Umgebung (den Problemraum) eingepasst werden!* Scheuen Sie sich nicht, den Algorithmus des Musters zu verändern, damit er für Ihr Problem passt. Das ist wesentlich klüger als das Muster unverändert zu lassen und den Rest des Programms an das Muster anzupassen (Schnittstellen et cetera). Wie bei allen anderen Dingen, die die Programmierung betreffen, ist auch der Einsatz von Mustern eine Frage der Übung. Je besser Sie den Sinn eines Musters verstehen, desto besser können Sie es einsetzen. Beschäftigen Sie sich daher eingehend mit den angegebenen Musterdatenbanken!

#### 4.2.5 Zusammengefasste Lehren

Folgendes sollten Sie aus diesem Abschnitt mitnehmen:

- Guter Stil ist die erste Grundlage erfolgreichen Programmierens. Kommentare und Leerzeichen erhöhen die Lesbarkeit des Co-

des. Vermeiden Sie sinnlose Komplexität. Seien Sie misstrauisch, wenn etwas allzu »cool« wirkt.

- ❑ Betreiben Sie elementares Programmieren. Viele Elemente des Sprachumfangs einer Programmiersprache können einfach ersetzt werden. Wenn das geht, tun Sie es. Weniger Anweisungen, die man öfter verwendet, sind vertrauter und können effizienter eingesetzt werden.
- ❑ Modellierung ist zu weiten Teilen Modularisierung. Überprüfen Sie immer wieder Ihr Modell; suchen Sie ausdrücklich nach komplexen Stellen. Beseitigen Sie Komplexitäten durch Teilen des Algorithmus.
- ❑ Verwenden Sie Muster, um Blöcke zu füllen. Beschäftigen Sie sich mit der zu Mustern existierenden Literatur. Arbeiten Sie die Muster durch, die wir im Anhang aufgelistet haben. Setzen Sie sie als Standardmodelle für Teilprobleme von Lösungen ein.

Im nächsten Abschnitt beschäftigen wir uns mit dem Jagdaspekt der Programmierung: der ewigen Suche nach der Wanze im Algorithmus.

## 4.3 Kluge Fehlersuche

### 4.3.1 Beispiel: Falsch testen, richtig testen

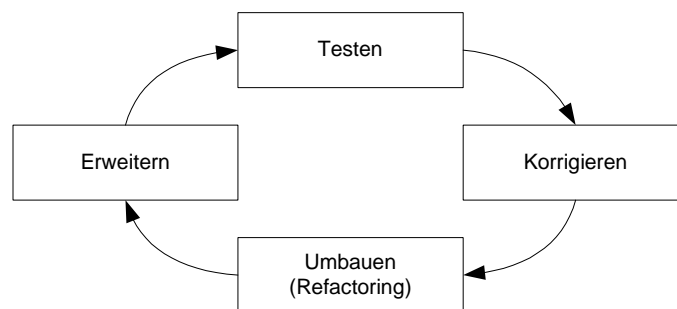
Testen dient dazu, Fehler im Programm (Modell) zu finden. Grundsätzlich gibt es zwei Arten von Fehlern: Syntaxfehler (Verletzungen der *Grammatik* der Programmiersprache) und inhaltliche (semantische) Fehler. Erstere sind leicht zu finden. Der Perl-Interpreter gibt sie mit (fast immer richtiger) Zeilennummer an. Semantische Fehler zu finden ist dagegen eine Kunst, die zu erlernen der Übung bedarf. Glücklicherweise bekommt man beim Programmieren ausreichend Gelegenheit dazu.

Zunächst stellt sich die Frage, wie ein Testprozess aussehen sollte. Um Nachhaltigkeit zu garantieren, das heißt die Wahrscheinlichkeit der Fehlerfreiheit eines Programms zu maximieren, muss der Testprozess – wie das Vorgehensmodell, zu dem er gehört – iterativ sein. Nur indem man wieder und wieder prüft, kann man einigermaßen Gewissheit erlangen. (Es gibt ProgrammiererInnen, die die Auffassung vertreten, dass kein Programm je ohne Fehler ist; ja, dass Programme als Einschränkung der Wirklichkeit an sich schon Fehler sind.) Abbildung 4.4 illustriert einen optimalen Testprozess. Auf die Evaluierung des Programms (Modells) folgt die Korrektur der gefundenen Fehler. Damit ergeben sich meist Stilbrüche und Komplexitäten im Algorithmus (zum Beispiel komplexe Bedingungen und lange Blöcke). Diese werden in einem

*Testablauf*

Testen ist 80% der Programmierarbeit!

**Abbildung 4.4**  
Iterativer Testablauf



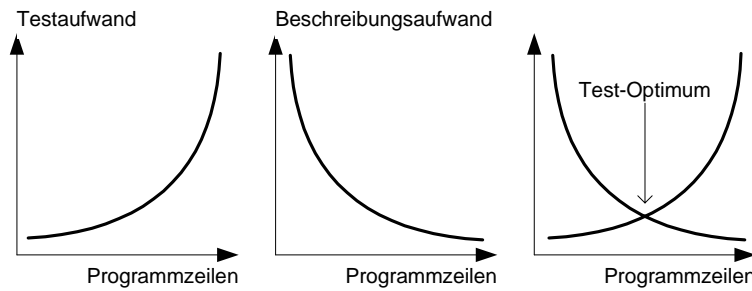
Wann testen?

Als nächste Frage stellt sich: Wann soll man testen? Der klassische Ansatz ist, jeweils eine Funktion zu implementieren und dann vollständig zu testen (»auszutesten«). In den letzten Jahren setzen sich aber immer mehr die so genannten *test-getriebenen* Ansätze (*test driven approaches*) durch. Ein solcher ist zum Beispiel das *Extreme Programming*-Vorgehensmodell [29]. Test-getriebenen Ansätzen folgend implementiert man erst die Testfunktionen und dann erst die zu testenden Funktionen. Was zunächst sonderbar erscheint, hat aber Methode: Beim Erstellen der Testfunktionen setzt man den Analyse-/Modellierungsprozess fort (man denkt zum Beispiel im Detail über die Anforderungen der Funktion nach). Das Ergebnis sind Programme, die besser den Anforderungen entsprechen.

Schildts Regel

Egal ob vorher oder nachher getestet wird, in beiden Fällen ergibt sich die Frage, nach *wie vielen* Änderungen man testen soll. Eine Funktion kann als Einheit bereits zu groß sein. Der Autor von [36] (einem Klassiker über die Programmierung in C) gibt als Faustregel für die Dauer eines Testvorgangs folgende Gleichung an:  $testZeit = (anzahlZeilen + X)^2$ . Das bedeutet, der Testaufwand steigt quadratisch mit der Anzahl zu testender Zeilen (der linke Graph in Abbildung 4.5). Die Anzahl zu testender Zeilen sollte also minimal sein, um den Testaufwand zu minimieren.

Dem gegenüber steht der *Beschreibungsaufwand*. Darunter verstehen wir, wie schwierig es ist, den zu testenden Modellteil in einer Programmiersprache zu beschreiben. Im mittleren Graph von Abbildung 4.5 nehmen wir an, dass der Beschreibungsaufwand quadratisch mit der Anzahl der erlaubten Programmzeilen abnimmt. Das heißt, mit zu wenigen Zeilen ist es praktisch unmöglich, das Modell adäquat zu beschreiben, und ab einer gewissen Zahl von Zeilen kann die Beschrei-



**Abbildung 4.5**  
Bestimmung der optimalen Zahl von Programmzeilen für das Testen

bung nicht mehr sinnvoll verbessert werden. Die optimale Zahl von Programmzeilen liegt im Schnittpunkt dieser beiden Kurven. Das sagt zwar immer noch nicht, nach wie vielen neuen/geänderten Programmzeilen man wieder testen soll; das Modell vermittelt aber Verständnis für das Spannungsfeld, in dem dieses Problem zu lösen ist. *Im Allgemeinen wird man immer dann testen, wenn man eine substantielle Änderung am Code vorgenommen hat.*

Damit kommen wir zum wesentlichen Punkt beim Testen: der richtigen Vorgangsweise. Für richtiges Testen gibt es – neben regelmäßig und iterativ testen – nur zwei einfache Regeln: *Fehler schrittweise eingrenzen* und *nie zwei Dinge auf einmal ändern*. Die zweite Regel ist klar: Nur so kann man einen Zusammenhang zwischen *Ursache und Wirkung* erkennen. Eingrenzen bedeutet: Wenn man einen Fehler bemerkt hat, schrittweise den Ort des Fehlers finden. Das funktioniert, indem man sich alle Variablen in der Umgebung des Fehlers immer wieder ansieht (zum Beispiel durch Ausgaben mit `print`) und indem man bereits überprüfte Teile des Algorithmus ausschaltet (zum Beispiel durch Auskommentieren). Ein Beispiel:

Fehler eingrenzen

```
my $farben = "gelb, rot, grün, orange, blau";
my @farbenListe = erstelleFarbenListe($farben);
print $farbenListe[0], "\n";

sub erstelleFarbenListe {
    my $farbenListe = sort split(/, /, $_[0]);
    return(@farbenListe);
}
```

Dieses Programm sollte eigentlich die Farben sortieren, als Liste speichern und die erste Farbe ausgeben. Wenn wir es ausführen, wird aber nichts ausgegeben. Wo ist der Fehler? Zunächst überprüfen wir das Hauptprogramm: nichts zu finden. Also sehen wir uns die Funktion an. Parameterübergabe und Rückgabe funktionieren einwandfrei. Das

können wir mit `print`-Anweisungen an den richtigen Stellen feststellen. Also muss der Fehler im Zerlegen und Sortieren liegen. Und damit haben wir ihn: `$farbenListe` ist als Skalar definiert und nicht als Liste! Daher wird eine leere Liste an das Hauptprogramm zurückgegeben. Durch schrittweises Eingrenzen und Überprüfen der verwendeten Daten haben wir den Fehler gefunden.

*Debugger*

Abschließend noch ein Hinweis: Moderne Entwicklungsumgebungen (aber auch die Perl-Laufzeitumgebung selbst) bieten zum Testen so genannte *Debugger* an. Ein Debugger ist ein Interpreter, der es erlaubt, ein Programm schrittweise auszuführen, die Werte von Variablen zu beobachten (durch *Watches*) und die Programmausführung an beliebigen Stellen zu unterbrechen (*Breakpoints*). Ob Sie einen Debugger verwenden oder »von Hand« mit `print` und Kommentaren testen ist letztendlich Geschmackssache. In [42] finden Sie eine detaillierte Einführung in das Testen und Debuggen.

### 4.3.2 Typische Perl-Fehler

Wir sind bereits in früheren Kapiteln auf typische Perl-Fehler eingegangen. Hier noch einmal eine Liste der wichtigsten und häufigsten Fehlermöglichkeiten, auf die Sie achten sollten:

- ❑ Fehler um 1 in Listen und Schleifen. Jeder Listenindex in Perl beginnt mit 0. Daher sollten Sie auch jeden Zähler in einer Schleife mit 0 beginnen lassen und bis zum Wert `length(<variable>)-1` laufen lassen.
- ❑ Achten Sie darauf, dass Variablen immer mit dem richtigen Typ definiert sind: `$` für Skalare, `@` für Listen, `%` für Hashes. Das obige Beispiel zeigt, zu welchen kryptischen Fehlern falsche Deklarationen führen können.
- ❑ Die Verwendung von `@`, `%` und `$` in Zeichenketten funktioniert nur richtig, wenn die Funktion dieser Zeichen durch einen Backslash maskiert ist. Sonst wird das nächste Wort immer als Variablenname interpretiert.
- ❑ Schließen von Anführungszeichen und Klammern. Besonders fehlende `}` können unerklärliche Fehler auslösen. Alle anderen fehlenden Zeichen sind relativ leicht zu finden.
- ❑ Tippfehler in Variablennamen. Perl erlaubt die Verwendung beliebiger Variablen (auch nicht initialisierter). Daher gibt es bei falsch geschriebenen Variablennamen (zum Beispiel in Zuweisungen) auch keine Fehlermeldungen. Fast alle Perl-Interpreter (etwa der Unix-Perl-Interpreter mit Option `-w`) geben bei der Verwendung nicht initialisierter Variablen aber Warnungen aus.

- ❑ Verwendung des Punktes in regulären Ausdrücken (zum Beispiel in `/.+xxx$/`). Der Punkt steht für ein beliebiges Zeichen. In Kombination mit `+` und `*` kann er zum »Auffressen« nachfolgender Muster (des `xxx` im Beispiel) führen. Überlegen Sie lieber genau, welche Zeichenklassen wirklich vorkommen können.
- ❑ Austauschen von Dateien zwischen Unix- und Windows-Systemen. Microsoft-Windows-Betriebssysteme hängen hinter jeden Zeilenwechsel ein Zeilenvorschub-Zeichen. Unix-Betriebssysteme tun das nicht und reagieren verwirrt auf dieses Sonderzeichen. Wenn sich ein Windows-Perl-Programm unter Unix nicht ausführen lässt, liegt das meist an diesem Unterschied. Die Lösung ist ein einfaches Perl-Skript:

```
while(<>) {      # @ARGV==(windows-perl-programm)
  chop;
  print;
}
```

Sind Sie mit einem neuartigen Fehler konfrontiert und ratlos, dann sollten Sie zunächst immer nach einem dieser Fehler Ausschau halten. Insbesondere nicht geschlossene Blöcke können zu unverständlichen Fehlermeldungen führen. Auch hier gilt wieder: Nur planmäßiges, schrittweises Eingrenzen des Fehlers führt zum Erfolg.

### 4.3.3 Anleitung zur Selbsthilfe

*Selbsthilfe ist Verstehen.* Alles beginnt mit einem Problem: Man versteht einen Zusammenhang nicht oder ist vollkommen ratlos, wie etwas funktionieren soll. Oder – am frustrierendsten – man versteht nicht, warum etwas *nicht* funktioniert. In jedem Fall Anlass, zur *planmäßigen* Selbsthilfe zu schreiten. Wie das geht, skizziert dieser Abschnitt. Wir haben ihn eingefügt, weil es unsere Erfahrung ist, dass die meisten Menschen in ihren Programmierbemühungen an mangelnder Fähigkeit zur Selbsthilfe scheitern. Wichtig ist: *Gehen Sie systematisch vor!* Nur so besiegen Sie die Verzweiflung. Wann immer Sie ein Gefühl der Computer-bedingten Hoffnungslosigkeit in sich aufsteigen spüren, versuchen Sie den Empfehlungen dieses Abschnittes zu folgen.

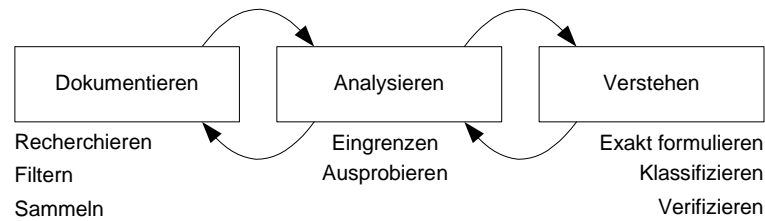
Selbsthilfe ist ein Prozess, der aus drei Elementen besteht (siehe Abbildung 4.6). Diese drei Phasen sind *Dokumentation*, *Analyse* und *Verstehen*. Wie alle guten (das heißt, menschen- und wirklichkeitsgerechten) Prozesse ist auch Selbsthilfe ein iterativer, rückgekoppelter Prozess. Das heißt, wenn Sie in einem Schritt etwas lernen, das Ihr Bild von den früheren Schritten verändert, dann kehren Sie zurück und versuchen

*Systematisches  
Arbeiten*

*Dokumentation,  
Analyse, Verstehen*

Sie, die geänderte Sichtweise sowie die sich daraus ergebenden Folgerungen für die späteren Schritte zu verstehen.

**Abbildung 4.6**  
Schritte der Selbsthilfe



»Dokumentation« bedeutet, das Problem zu beschreiben. Zunächst ist es wichtig, den eigenen Unmut zu überwinden und sich zu zwingen, die Phänomene exakt zu beobachten, anhand derer man das Problem erkannt hat. Da die Beschäftigung mit Problemen zumeist ein seelisches Ungleichgewicht mit einschließt, ist es von Vorteil, ein neutrales Medium zur Dokumentation zu verwenden: also Papier oder eine Textdatei anstatt des eigenen Gedächtnisses. Die Verwendung von Papier zur Dokumentation hat darüber hinaus die Vorteile unmittelbarer Visualisierung des Wissens und des wahlfreien Schreib- und Malzugriffes. Nach unserer Ansicht ist Papier das ideale Medium, um Probleme zu lösen.

Objektivierung durch  
Aufschreiben

Hat man eine Beschreibung des Problems, die alles enthält, was man beobachtet hat (auch alle Vermutungen dazu), ist der nächste Schritt zu recherchieren. Versuchen Sie herauszufinden, ob auch schon jemand anderer dieselben Phänomene wahrgenommen hat. Der Königsweg dazu ist heutzutage das World Wide Web in Gestalt der Suchmaschine *Google* [22]. Formulieren Sie Ihre Erkenntnisse in ganzen Sätzen, geben Sie wichtige, zusammenhängende Formulierungen (etwa Fehlermeldungen) zwischen doppelte Anführungszeichen (Phrasensuche), eliminieren Sie Stoppwörter (Artikel, Fürwörter et cetera), formulieren Sie möglichst in englischer Sprache und führen Sie Ihre Abfragen für alle möglichen Fälle und Verbformen wiederholt durch. Darüber hinaus bietet Ihnen Google mit dem Attribut `filetype:` die Möglichkeit, nur nach ganz bestimmten Medien zu suchen. Nach Webseiten mit Musterdatenbanken für Perl könnte man unter anderem folgendermaßen suchen:

Suchen im Google

```
introduction "design patterns"
"design patterns" perl
patterns perl filetype:html
```

Wichtig ist auch, richtig in den Ergebnissen zu suchen. Hat Ihre Suche zu mehr als hundert Treffern geführt, ist sie wertlos. Führen Sie sie erneut durch mit zusätzlichen einschränkenden Suchbegriffen. Haben Sie

eine ausreichend kleine Treffermenge, dann sehen Sie sich wirklich alle Ergebnisse an. Google sortiert Ergebnisse zwar sehr schlau, ist aber dennoch nur eine Maschine. Oft ist das beste Ergebnis erst das hundertste. Unter Unix stehen darüber hinaus noch zwei äußerst wertvolle Systemkommandos zur Verfügung: man zum Lesen von Dokumentation und apropos zum Suchen von Dokumentation [26].

Hat man Information gefunden, besteht die nächste Hürde darin, sie *richtig* zu lesen. Falsch ist: Dokumente elektronisch speichern und von A bis Z durcharbeiten. Richtig ist: In der Dokumentation nach *interessanten Stellen* suchen (querlesen, an zufälligen Stellen aufschlagen et cetera) und interessante Seiten ausdrucken. Speichern Sie sich außerdem einen Link auf die Quelle in einer Textdatei. Sammeln Sie Ihre Ausdrücke zusammen mit der textuellen Problembeschreibung. Wenn Sie das Gefühl haben, dass eine Quelle früher Gefundenes widerlegt, dann ist es sehr wichtig, dieses veraltete Wissen *wegzuwerfen*. Sammeln und Filtern sind die ultimativen Schritte zur Problembewältigung. Und nochmals: nie ein Dokument als Ganzes lesen (außer dieses Buch natürlich)!

*Richtiges Lesen*

Die Provokation im letzten Absatz hat einen Sinn: Zweck der Selbsthilfe ist nicht nur, für den Moment ein Problem zu lösen, sondern vielmehr, langfristig zu lernen. Über das Problemgebiet; vor allem aber über Selbsthilfe. Wissen ist eine feine Sache im Leben. Sich selbst helfen zu können ist aber eine bessere Sache. Außerdem lehrt die Erfahrung, dass das Schicksal jedes Detailwissens *Vergessen* ist. Es ist daher ökonomisch nicht sinnvoll, allzu großes Detailwissen zu erlernen. Techniken sind besser. Dennoch ist es praktisch, die zu einem Problem erstellte Dokumentation in einer Kartei zu sammeln, um beim nächsten Auftreten Zeit zu sparen.

Auf die Dokumentation folgt die Analyse. Wie beim Testen heißt das wieder: *Ausprobieren* und *Eingrenzen*. Ist unser Problem ein Programmierproblem, so gehen wir wieder wie oben beschrieben vor. Ist es etwas anderes, versuchen wir durch Analyse eine Problemlösung zu modellieren (zum Beispiel als *Problembaum*, der alle Elemente der Lösung – wie ein Funktionsgraph Funktionen – enthält). Durch Ausprobieren (etwa im Geist die Lösung durchspielen, den so genannten Schreibtischtest) können Lücken, Doppeldeutigkeiten und sonstige Mängel der Lösung gefunden werden. Eine wesentliche Frage ist dabei stets: *Welches Wissen fehlt mir noch zur Problemlösung? Was verstehe ich nicht?* Das zu beantworten, führt uns wieder zur Dokumentation zurück. Der entscheidende Punkt, um sich durch Analyse über ein Problem zu erheben, ist jedoch: *Spielen Sie mit dem Problem. Machen Sie bewusst Fehler und beobachten Sie, was passiert. Lachen Sie sich krumm über die Dummheit Ihres Computers. Lernen Sie aus den beobachteten Phänomenen.*

*Ausprobieren und  
Eingrenzen*

So gewinnen Sie ein Gefühl für Probleme und Lösungen. Außerdem ist ein wichtiger Schritt der Selbsthilfe, sich dem Problem nicht unterzuordnen, sondern es zu persiflieren.

*Verstehen basiert auf  
genauem Denken*

Die Analyse führt uns direkt zum Verstehen. Verstehen bedeutet, dass wir dazu in der Lage sind, was wir zunächst nur anhand von Phänomenen (beobachteter und gedachter) beschreiben konnten, in Form von Fragen zu formulieren. Verstehen ist *Exaktheit*. Ein Problem exakt formulieren können, bedeutet, ein Modell formulieren können. Damit sind wir wieder am selben Punkt wie zu Beginn dieses Buches: Problem und Lösung unterscheiden sich nur im Detaillierungsgrad. *Selbsthilfe ist die Suche nach dem Detail*. Wer lernt, exakt zu denken, lernt, sich selbst zu helfen. Praktisch versteht man, indem man mögliche Lösungen (Klassen von Lösungen) aus den Analyseergebnissen ableitet und ihre Wahrscheinlichkeit bewertet. Durch Ausprobieren (Verifizieren) kann man dann eine passende Lösung auswählen. Beachten Sie dabei, dass jedes Problem (jede Lösung) eine eigene Vorstellungswelt darstellt. Dass daher die Lösung innerhalb dieser Welt plausibel sein muss. Und dass etwas nicht dadurch plausibel wird, indem es unserer Voreingenommenheit entspricht (die in Vorstellungswelten als etwas Geistigem besonders stark ist!). Versuchen Sie vielmehr, objektive Kriterien zu finden, anhand derer Sie die Lösungen bewerten. Wenn Sie finden, dass zur Lösung noch etwas fehlt, so versuchen Sie diese Empfindung zu formulieren und kehren Sie zur Analyse zurück.

*Zusammenfassung*

Das ist Selbsthilfe. Im Grunde dasselbe wie Analysieren, Modellieren und Testen - Programmieren also. Kein Wunder: Da wie dort geht es darum, ein Problem in eine Lösung umzuwandeln. Hinter dem Programmieren steht also ein allgemeines Prinzip, das sich auf alle Lebensbereiche anwenden lässt. Wir möchten sogar so weit gehen zu behaupten, dass, wer den Geist des Programmierens verinnerlicht hat, auch in vielen anderen Bereichen des Lebens erfolgreicher sein wird. Mit diesem Zuckerl möchten wir Sie motivieren, die Übungsaufgaben am Ende des Kapitels fleißig durchzuarbeiten. Denn: *Auch Selbsthilfe ist Üben!*

#### 4.3.4 Destilliertes Testwissen

Damit haben wir über das Testen – als dem integralen Bestandteil erfolgreicher Softwareentwicklung – Folgendes gelernt:

- Selbsthilfe ist ein iterativer Prozess, bestehend aus Dokumentation, Analyse und Verstehen. Wesentlich ist, sich über das Problem zu erheben und es spielerisch zu lösen. Und: Was nicht zur Lösung beiträgt, wegwerfen!

- ❑ Richtiges Testen besteht aus Ausprobieren, Korrigieren, Umbauen und Erweitern. Fehler findet man, indem man sie durch Beobachten von Variablen und Auskommentieren von Blöcken eingrenzt. Testen ist iterativ und eng verzahnt mit Analyse und Modellierung.
- ❑ Die bei weitem häufigsten Fehler in Perl-Programmen sind falsch gesetzte } zum Schließen von Blöcken. Die daraus resultierenden Fehlermeldungen weisen meist direkt ins Nirvana. Denken Sie also an diese Möglichkeit, wenn Sie einen Fehler nicht verstehen.

Wir haben in diesem Kapitel gelernt, wie man *richtig* programmiert und Fehler findet. Im letzten Abschnitt sehen wir uns an, wie man bereits funktionierende Programme *schneller* macht.

## 4.4 Programme schneller machen

Ein Programm muss eine gute Performance haben. Was ist »Performance«? Darunter versteht man, dass ein Algorithmus ein Problem, abhängig von der Komplexität des Problems, möglichst schnell lösen soll. Grundlage des Performance-Begriffes ist also *algorithmische Komplexität*. Wichtig ist, sich stets bewusst zu halten, dass Performance trotzdem nie so wichtig ist wie leserlich zu schreiben und klug zu gliedern.

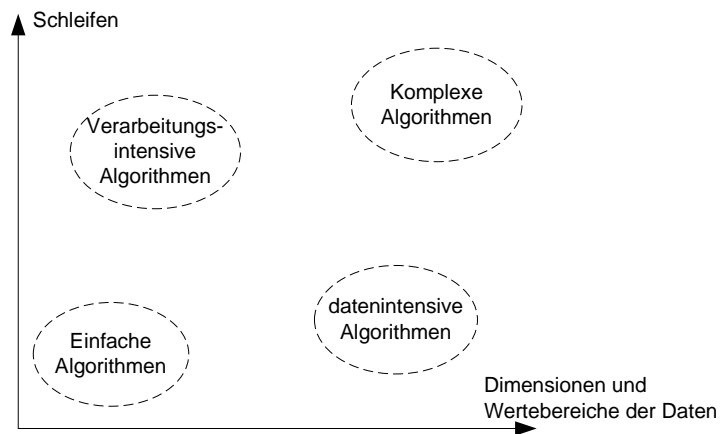
In der theoretischen Informatik (ein Feld der Wissenschaft, das von so großen Namen wie Alan Turing, John von Neumann und Noam Chomsky geprägt wurde) gibt es ausgefeilte Methoden, um algorithmische Komplexität zu messen. Für uns reicht aber eine einfachere *Operationalisierung* (Übersetzung in die Verwendbarkeit; auch ein Wort für Programmieren). Abbildung 4.7 zeigt die beiden wesentlichen Faktoren, die Auswirkung auf die algorithmische Komplexität haben: die *Dimensionalität der verarbeiteten Daten* und die *Anzahl der Schleifen im Algorithmus*. Diese beiden Faktoren gehen oft Hand in Hand.

Eine vernünftige Vorgangsweise beim Schnellermachen von Algorithmen beginnt mit einem weitgehend ausgetesteten, funktionierenden Programm. Zunächst versucht man, Stellen im Algorithmus zu finden, die komplex sind: die also verschachtelte Schleifen und Operationen auf mehrdimensionale Daten enthalten. Das sind die lohnendsten Ansatzpunkte. Hat man eine solche Stelle gefunden, versucht man, ihre Komplexität zu verringern. Dabei verwendet man wieder dieselben Methoden wie bei der Modularisierung: Zergliedern von Datenfeldern, Trennen von verschachtelten Schleifen in mehrere Funktionen et cetera. Oft helfen auch Muster (Sortieralgorithmen et cetera), um Programme schneller zu machen. Natürlich ist das nicht immer möglich. Das ändert

*Komplexität von Algorithmen*

*Faustregeln*

**Abbildung 4.7**  
Faktoren, die Einfluss  
auf die Komplexität  
eines Algorithmus  
haben



aber nichts an der prinzipiellen Richtigkeit des Ansatzes. Sehen wir uns zum Abschluss ein Beispiel an:

```
my @personenName = ("Gunnar", "Annegret", "Sven");
my @orte = ([ "oslo", "stockholm", "ostende" ],
            [ "göteborg", "bergen", "malmö" ],
            [ "stockholm", "malmö", "oslo" ] );
my $namensIndex=0;
foreach my $name (@personenName) {
    my $andererNamensIndex=0;
    foreach my $andererName (@personenName) {
        if ($name ne $andererName) {
            for(my $ortsIndex=0; $ortsIndex < 3; $ortsIndex++) {

                my @andereOrte = @orte[$andererNamensIndex];

                for(my $andererOrtsIndex=0; $andererOrtsIndex < 3;
                    $andererOrtsIndex++) {
                    if ($orte[$namensIndex][$ortsIndex] eq
                        $orte[$andererNamensIndex][$andererOrtsIndex]) {
                        print "$name - $andererName: $ortsIndex ",
                            "$andererOrtsIndex ",
                            $orte[$namensIndex][$ortsIndex], "\n";
                    }
                }
            }
        }
        $andererNamensIndex++;
    }
    $namensIndex++;
}
```

Dieser Algorithmus überprüft, ob zwei Personen einen gemeinsamen Wohnort haben, und gibt im Erfolgsfall ihre beiden Namen und den Ort aus. Durch Anwendung der oben angesprochenen Verfahren lässt sich dieser Algorithmus in eine viel weniger komplexe Form bringen. Hauptproblem sind derzeit die vier (!) ineinander verschachtelten Schleifen. Durch Umorganisation der Daten und Verwenden regulärer Ausdrücke lässt sich eine Schleife eliminieren. Das Ergebnis ist folgender Algorithmus:

```
my @personenName = ("Gunnar", "Annegret", "Sven");
my @orte = ([ "oslo", "stockholm", "ostende" ],
            [ "göteborg", "bergen", "malmö" ],
            [ "stockholm", "malmö", "oslo" ] );
my @wohnOrte;

# 1. Daten umorganisieren
for(my $index=0; $index < 3; $index++) {
    $wohnOrte[$index] = "";
    for(my $i=0; $i < 3; $i++) {
        $wohnOrte[$index]..= $orte[$index][$i] . " ";
    }
}

# 2. Vergleichen
for(my $namensIndex=0; $namensIndex < 2; $namensIndex++) {
    for(my $andererNamensIndex=$namensIndex+1;
        $andererNamensIndex < 3; $andererNamensIndex++) {
        for(my $i=0; $i < 3; $i++) {
            if ($wohnOrte[$andererNamensIndex] =~
                $orte[$namensIndex][$i]) {
                print $personenName[$namensIndex], " - ",
                    $personenName[$andererNamensIndex], ": ",
                    $orte[$namensIndex][$i], "\n";
            }
        }
    }
}
```

Immerhin eine Ebene weniger. Weitere Performance-Gewinne sind nur über Zusatzwissen über die Daten (Anzahl der Zeilen, Spalten) zu erzielen. Fazit: Performance steigern ist eine Kunst.

*Programmieren ist Üben!* Beschäftigen Sie sich eingehend mit den Übungsaufgaben im nächsten Abschnitt und denken Sie sich bessere aus. Helfen Sie mit, dieses Buch zu verbessern, indem Sie sie an

*Feedback an perl-  
buch@ims.tuwien.ac.at*

perlbuch@ims.tuwien.ac.at mailen. Damit würden Sie uns eine große Freude machen!

## 4.5 Übungsaufgaben

1. Sehen Sie sich die Algorithmen, die Sie bisher modelliert haben, nochmals an. Versuchen Sie, Komplexitäten aufzuspüren und die Gliederung der Algorithmen zu verbessern. Versuchen Sie außerdem, Anweisungen zu ersetzen und Muster zu verwenden.
2. Üben Sie die Fehlersuche. Implementieren Sie Ihre Modelle in Perl und implementieren Sie Testfunktionen. Versuchen Sie, die Korrektheit Ihrer Algorithmen durch Eingrenzen zu verifizieren.
3. Versuchen Sie, die Performance Ihrer Algorithmen mit den genannten Methoden zu verbessern. Versuchen Sie außerdem, komplexe Teile Ihrer Algorithmen zu finden und ihre praktische Auswirkung abzuschätzen.
4. Suchen Sie im Internet nach Musterdatenbanken für Perl. Erstellen Sie sich einen Katalog an nützlichen Mustern und vergleichen Sie diese Muster mit den Mustern im Anhang.