5 Objektorientierte Programmierung

In diesem Kapitel werden wir lernen, dass gute objektorientierte Programmierung (kurz »OOP«) auch in Perl kein Problem ist.

Bei der prozeduralen Programmierung steht die Programmlogik im Vordergrund. Sie definiert Funktionen für das Verarbeiten von Daten. Bei komplexen Daten wird diese Art der Programmierung aber schnell unübersichtlich, und die Skripts sind im Nachhinein nicht einfach zu ändern.

Im Gegensatz zur prozeduralen Programmierung stehen bei der objektorientierten Vorgehensweise die Daten selbst im Mittelpunkt. Sie sind nun nicht mehr einfache Parameter von Funktionsaufrufen, die als Kopie oder Referenz übergeben werden, sondern treten selbst in Aktion und sind sozusagen »lebendig«. Objekte besitzen Eigenschaften (so genannte Attribute, englisch »Attributes«) und bieten Methoden an, mit denen diese Eigenschaften gelesen oder verändert werden können. Auch Aktionen, die sich auf das gesamte Objekt beziehen, werden von den Objekten selbst in Form von Methoden zur Verfügung gestellt.

Der Vorteil objektorientierter Programmierung liegt unter anderem darin, dass bei einer Änderung der internen Verarbeitung der Daten die Programmierschnittstelle nach außen unverändert bleibt. Dies nennt man »Kapselung«. Auch bei Änderungen im Programmcode hat OOP die Nase gegenüber herkömmlicher Programmierung vorn, weil man eine Änderung an zentraler Stelle durchführen kann, ohne dass sich diese Änderung wie ein Rattenschwanz durch alle Programme zieht.

Ein weiterer Begriff, den wir weiter unten noch ausführlich besprechen werden, spielt bei OOP ebenso eine Rolle, das ist die »Vererbung«. Damit kann man zunächst mit einer relativ einfachen Implementierung beginnen und den bereits erstellten Code an weitere, spezialisierte Module weitervererben. Durch die Vererbung von Attributen und Methoden spart man sich oft eine Menge Schreibarbeit.

Hier ein Beispiel für Objekte mit Vererbung:

Aus der Abbildung wird deutlich, dass der Wagen von Frau Huber ein Auto (KFZ) ist, den die Behörden als »PKW« klassifizieren. Die Marke des PKW heißt »Lamborghini«, und das Modell ist »Typ 1«.

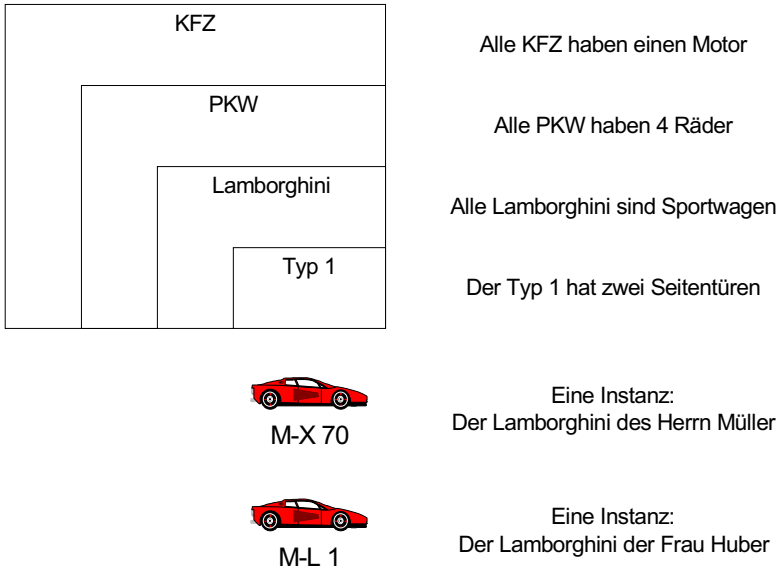

Abbildung 5.1: Autos als Objekte

Herr Müller besitzt ebenfalls einen Lamborghini aus der Modellreihe »Typ 1«. Beide Wagen unterscheiden sich zunächst rein äußerlich durch die verschiedenen Kennzeichen. Natürlich sind auch Fahrgestellnummer, Ausstattung, Motor etc. nicht identisch. Diese individuellen Eigenschaften werden beim Zusammenbauen des Autos in der Fabrik festgelegt, bis am Ende eine physische Ausprägung des Objekts »Modell Typ 1« vom Fließband läuft, die man »Instanz« nennt. Das Zusammenbauen einer solchen Instanz wird als »Instanzierung« bezeichnet.

Sehen wir uns noch einmal das Bild an: Jede Objektinstanz ist einer so genannten »Klasse« zugeordnet. Sowohl das Auto von Frau Huber als auch das des Herrn Müller gehören zur Klasse »Typ 1«. In dieser Klasse werden diejenigen Eigenschaften festgelegt, die für alle Instanzen der Klasse »Typ 1« gleichermaßen gelten (z.B. hat jede Instanz zwei Seitentüren). Eine weitere Eigenschaft, nämlich die Gattung »Sportwagen«, ist kein Merkmal, das nur für dieses eine Modell gilt, sondern für alle Wagen der Marke »Lamborghini«. Man kann also sagen, die Klasse »Typ 1« ist eine Unterklasse von »Lamborghini« und erbt das Attribut »Sportwagen«.

Gehen wir noch einen Schritt weiter: Jeder Lamborghini, ebenso alle Autos der anderen Marken, werden als »PKW« bezeichnet und haben (meist) 4 Räder. Die Klasse »Lamborghini« kann also auch als Unterklasse von »PKW« angesehen werden und erbt von dieser das Merkmal »4 Räder«. Eine weitere Untergliederung findet man, wenn man zum Beispiel alle PKW und LKW als Unterklassen von »KFZ« definiert, denen das Merkmal »Sie besitzen einen Motor« gemeinsam ist.

Nun sollten wir mit den Begriffen »Klasse«, »Instanz« und »Vererbung« keine größeren Probleme mehr haben.

In diesem Kapitel will ich mit Ihnen die Vorteile der objektorientierten Programmierung anhand eines einfachen Beispiels erarbeiten. Schreiben wir ein Modul, mit dessen Hilfe Userdaten verarbeitet werden können. Im einfachsten Fall benötigt man hierfür nur den Benutzernamen (englisch »login«) und das Kennwort (englisch »password«, wird meist mit »pwd« abgekürzt).

In der folgenden Abbildung möchte ich kurz skizzieren, was auf uns zukommt:

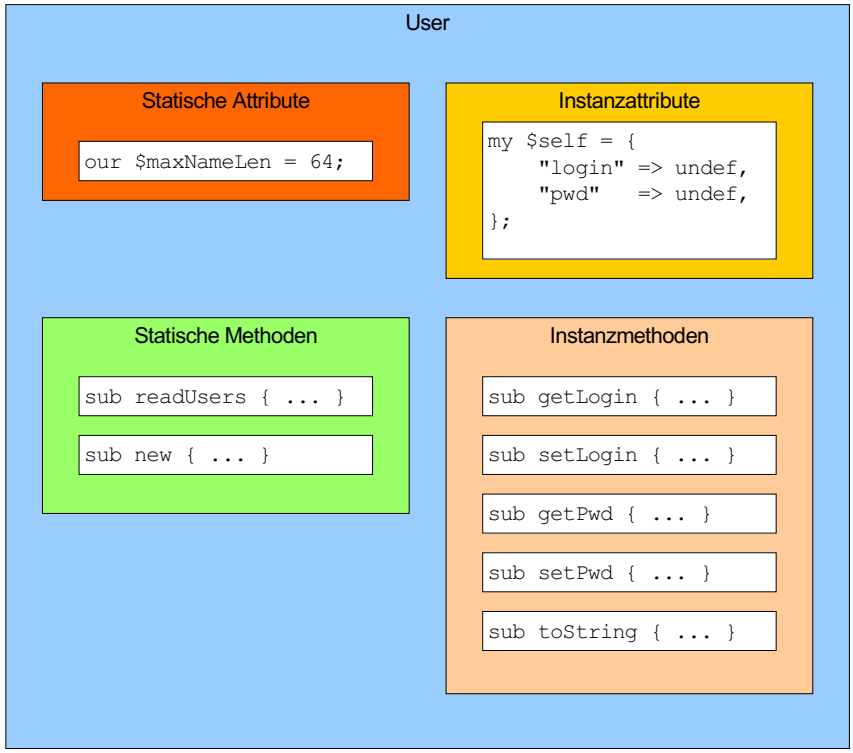

Abbildung 5.2: Beispielklasse »User«

Das Modul für unsere Beispielklasse enthält sowohl statische Attribute und Methoden als auch solche, die individuellen Instanzen der Klasse zugeordnet werden. Im Folgenden werden wir auf die einzelnen Bestandteile genauer eingehen.

5.1 Klassen

Nicht nur in Perl, sondern auch in allen anderen objektorientierten Programmiersprachen werden Objekte durch so genannte Klassen dargestellt.

Eine Klasse definiert Eigenschaften (Attribute) und Funktionalitäten (Methoden) für ein bestimmtes Objekt.

In Perl ist eine Klasse identisch mit einem Package. Der Programmcode steht in einem Perl-Modul, dessen Dateiname die Endung »`.pm`« besitzt und als Prefix den Packagenamen hat.

Wenn wir also die Klasse »User« implementieren, dann schreiben wir den Programmcode in die Datei »`User.pm`«.

> Es hat sich eingebürgert, die Dateinamen von Modulen mit einem Großbuchstaben zu beginnen. Halten wir uns also daran.

Beispiel für den grundsätzlichen Aufbau einer Klasse in Form eines Packages:

```
# Datei User.pm
package User;

use strict;
...
1;
```

Wie wir deutlich sehen, sieht die Struktur genauso aus wie die eines ganz normalen Perl-Moduls. Am Beginn des Moduls steht die Direktive `package`, danach folgt die für jeden ordentlichen Programmierer zwingend vorgeschriebene Direktive `use strict;`, und am Ende unsere obligatorische Zeile `1;`. Dazwischen werden wir noch Funktionsdefinitionen implementieren.

Eine Klasse definiert zwei Arten von Attributen und Methoden: solche, die für alle Objekte einer Klasse gleichermaßen gelten, und solche, die für jedes einzelne Objekt einer Klasse unterschiedlich sind.

5.1.1 Klassenattribute und Klassenmethoden

Diese Art von Attributen und Methoden nennt man auch »statisch«. Klassenattribute sind Eigenschaften der Klasse selbst, die für alle Objekte (Instanzen) der Klasse gleichermaßen gelten. So ist zum Beispiel die maximale Länge des Benutzernamens (nennen wir die Variable `$maxNameLen`) eines Users für alle Objekte gültig.

Die Variable, in welcher die Länge gespeichert ist, muss also nur einmal als Klassenvariable definiert werden:

```
package User;
...
our $maxNameLen = 64;
...
```

Die Variable `$maxNameLen` wird im Package-Scope (englisch für »Geltungsbereich innerhalb eines Moduls«) mit dem Bareword `our` definiert, sie ist also im gesamten Modul gültig und somit allen Objekten der Klasse gemeinsam.

Klassenmethoden bieten Funktionalitäten an, die nicht auf ein einzelnes Objekt der Klasse bezogen sind. Man benötigt also keine individuelle Instanz des Objekts, wenn man eine statische Methode aufruft. Eine Klassenmethode wird wie eine normale Package-Funktion aufgerufen, zum Beispiel `User::readUsers()`.

Die Definition einer Klassenmethode erfolgt genauso wie in normalen prozeduralen, nicht objektorientierten Modulen:

```
sub readUsers {
    # Code für das Lesen aller Benutzer
    # Es wird eine Liste der gelesenen Benutzer
    # an den Aufrufer der Funktion zurückgegeben.
    # Die einzelnen Elemente der Liste können
    # dann Instanzen der Klasse "User" sein.
}
```

5.1.2 Konstruktor

Den Begriff »Konstruktor« kann man auch mit dem nicht sehr schönen, dafür aber deutschen Wort »Zusammenbauer« übersetzen, denn es sagt genau aus, was damit gemeint ist. Er baut im Hauptspeicher eine Instanz der Klasse zusammen. Erst nach dem Aufruf des Konstruktors kann man auf die individuellen Attribute und Methoden des Objekts zugreifen.

In Perl ist der Konstruktor nichts anderes als eine Funktion wie alle anderen auch und sieht deshalb etwa so aus:

```
sub new {
    # Programmcode
}
```

Der Konstruktor hat in allen Programmiersprachen den Namen `new`, deshalb heißt der Funktionsname in Perl üblicherweise ebenfalls `new`. Im Prinzip könnte man sich jeden beliebigen Namen dafür einfallen lassen, bleiben wir aber beim Standard.

Der Konstruktor liefert dem Aufrufer eine skalare Referenz auf das individuelle Objekt (Instanz) zurück, die ich im Weiteren »Objektreferenz« nenne. Alle weiteren Aktionen werden dann über Instanzmethoden durchgeführt. In Perl sind dies ebenfalls Funktionen, wie wir weiter unten noch sehen werden.

Die vom Konstruktor zurückgelieferte Objektreferenz auf die neu erschaffene Instanz der Klasse ist eine Zwittervariable, denn zum einen ist sie eine Referenzvariable auf das Hash, in dem die Instanzattribute gespeichert sind, zum anderen verhält sie sich wie ein Modul, über das Funktionen aufgerufen werden können.

Damit Sie gleich Bescheid wissen: Die Zwittervariable wird fast ausnahmslos `$self` genannt (obwohl sie jeden beliebigen Namen haben könnte).

Bleiben wir bei unserem Beispiel der Klasse `User` und sehen uns eine Implementierung des Konstruktors für die Klasse an:

```
01 # Modul User.pm
02 package User;
03
04 # Statische Variablen, die allen Objekten (Instanzen)
05 # der Klasse gemeinsam sind
06 our $maxNameLength = 64;
07
08 # Konstruktor
09 sub new {
10     my $proto = shift( @_ );
11     my $class = ref( $proto ) || $proto;
12
13     my $self = {
14         "login"         => undef,
15         "pwd"           => undef,
16     };
17
18     # Überprüfung der Argumente
19     unless ( @_ and $_[ 0 ] and $_[ 1 ] ) {
20         return undef;
21     }
22
23     if ( length( $_[ 0 ] ) > $maxNameLength ) {
24         return undef;
25     }
26
27     # Setzen der Instanzattribute "login" und "pwd"
28     $self->{ "login" } = shift( @_ );
29     $self->{ "pwd" } = shift( @_ );
30
31     # Umwandeln der Hash-Referenzvariable zu einer
```

```
32     # Objektreferenz
33     bless( $self, $class );
34
35     return $self;
36 }
```

Bevor ich näher auf den Programmcode eingehe, möchte ich Ihnen kurz zeigen, wie man den Konstruktor in einem Skript oder in einem anderen Modul benutzt:

```
...
use User;

my $user = new User( "Hugo", "secret" );
...
```

Der Aufruf des Konstruktors ist bei Perl in vielen Varianten möglich. Häufig werden Sie in Programmen zum Beispiel folgende Version davon sehen:

```
my $user = User->new( "Hugo", "secret" );
```

Ich persönlich verwende die erste Variante, weil diese Syntax in anderen objektorientierten Programmiersprachen Standard ist.

Im Beispiel sehen wir eine Mischung aus statischen Elementen, die allen Instanzen gemeinsam sind, und solchen, die nur für eine einzelne Objektinstanz gelten. Die Variable `$maxNameLength` ist eine innerhalb des gesamten Moduls gültige Variable und somit ein statisches Attribut, das für alle Objekt Instanzen gleichermaßen gilt, während der Loginname und das Kennwort, die beim Aufruf des Konstruktors übergeben werden, nur für diese eine Objektinstanz gelten.

Die Zeilen 10 und 11

```
my $proto = shift( @_ );
my $class = ref( $proto ) || $proto;
```

dienen dazu, den Klassennamen für die Instanz zu erhalten, der beim Aufruf der Konstruktorfunktion `new()` implizit vom Perl-Interpreter als erstes Argument übergeben wird. Diese Eigenart werden wir bei allen Instanzmethoden bei Perl wiedertreffen. Von außen rufen wir den Konstruktor `new()` so auf:

```
my $user = new User( "Hugo", "secret" );
```

Der Interpreter fügt aber unsichtbar den Klassennamen als weiteren Parameter an den Anfang der Argumentliste hinzu, als hätten wir den Konstruktor wie folgt aufgerufen:

```
my $user = User::new( "User", "Hugo", "secret" );
```

In der zweiten Zeile wird geprüft, ob das vom Interpreter unsichtbar eingefügte Argument ein einfacher String ist, der den Klassennamen enthält, oder seinerseits eine Referenzvariable auf ein Objekt. Wir hätten nämlich den Konstruktor auch über eine bereits existierende Instanz der Klasse aufrufen können:

```
# Normale Instanzierung einer Objektinstanz
my $user = new User( "Hugo", "secret" );

# Instanzierung eines neuen Objekts über eine bereits
# existierende Objektreferenz
my $user1 = new $user( "Egon", "verysecret" );
# Oder auch so:
my $user2 = $user->new( "Willy", "auchsecret" );
```

Die Variable `$class` muss aber in jedem Fall den Klassennamen enthalten, da wir diesen weiter unten im Code noch benötigen. Der Klassenname ist in unserem Beispiel der String »`User`«.

Um den Unterschied zu verdeutlichen: Beim Aufruf

```
my $user = new User( "Hugo", "secret" );
```

enthält die Variable `$proto` den String `User` und damit bereits den benötigten Klassennamen, während der Aufruf von `ref( $proto )` in diesem Fall einen leeren String zurückliefert und somit einen `FALSE`-Wert ergibt.

Ruft man den Konstruktor aber über eine bereits existierende Instanz der Klasse auf:

```
my $user1 = new $user( "Egon", "verysecret" );
```

dann liefert der Aufruf `ref( $proto )` den Klassennamen `User` zurück, während die Variable »`$proto`« selbst eine Objektreferenz ist.

Wenn Sie diesen Mechanismus als zu schwierig empfinden, dann empfehle ich Ihnen, gar nicht weiter darüber zu grübeln und einfach immer diese beiden Zeilen zu verwenden, denn damit funktioniert der Code immer. Nicht nachdenken, einfach hinschreiben, heißt hier die Devise.

Die Zeilen 13 bis 16

```
# Alle Attribute einer Objektinstanz sind Elemente
# einer Hash-Referenzvariable.
# Die Namen der Attribute sind die Hash-Keys,
# die Attributwerte sind die Hash-Values.
my $self = {
    "login"             => undef,
    "pwd"               => undef,
};
```

sollten eigentlich keine Probleme bereiten. Sie definieren zunächst nur ein anonymes Hash, das die Objektattribute enthält und nur über die Referenzvariable `$self` zugänglich ist. Im Prinzip könnte man den Namen der Referenzvariable beliebig wählen, es hat sich bei Perl jedoch eingebürgert, ihn `$self` zu nennen. Wer andere objektorientierte Programmiersprachen kennt, kann sich unter dem Namen `this` dasselbe vorstellen.

Wir werden weiter unten sehen, dass diese Variable ein »zweites Ich« entwickelt.

In den Zeilen 19 bis 25

```
unless ( @_ and $_[ 0 ] and $_[ 1 ] ) {
    return undef;
}

if ( length( $_[ 0 ] ) > $maxNameLength ) {
    return undef;
}
```

findet ein Check der Argumente des Konstruktors statt. Es wird überprüft, ob deren Anzahl stimmt, ob die Argumente definiert und nicht leer sind und ob die Länge des Benutzernamens in Ordnung ist. Hier sehen wir eine Mischung aus Instanzcode und statischem Code, denn die Variable `$maxNameLength` ist statisch und für alle Objektinstanzen gleich.

Die Zeilen 28 und 29

```
$self->{ "login" } = shift( @_ );
$self->{ "pwd" } = shift( @_ );
```

füllen die Instanzattribute »`login`« sowie »`pwd`« mit den Argumenten, die beim Aufruf des Konstruktors übergeben wurden. Wie wir weiter unten sehen werden, setzt man Instanzattribute ausschließlich über so genannte »Setter«-Methoden und nicht direkt, wie ich es hier gemacht habe. Der Einfachheit halber wollen wir aber eine Ausnahme von der Regel machen.

Eine sehr wichtige Programmzeile ist die Zeile 33

```
bless( $self, $class );
```

denn hier wird die Hash-Referenzvariable `$self` zu einer Objektreferenz gemacht und führt von hier an ein Zwitterdasein, da sie mit dem Aufruf von `bless()` sowohl eine Referenz auf das Objekt der Klasse `User` als auch eine Hash-Referenz auf die Attribute der Instanz wird. Erst nachdem man die Referenzvariable an die Klasse gebunden hat, entsteht eine Objektinstanz, die in der nächsten Zeile des Programmcodes an den Aufrufer zurückgegeben wird.

Versuchen Sie nicht, in einem Wörterbuch die deutsche Bedeutung des Worts `bless` zu finden, ich möchte nicht, dass irgendjemand verzweifelt versucht, die Übersetzung »`segnen`« oder »`glückselig machen`« mit dem in Einklang zu bringen, was die Funktion `bless()` wirklich tut: Sie bindet die Hash-Referenz an die Klasse und macht sie zu einer Objektreferenz.

Eine Grundregel von OOP lautet: Der Konstruktor sollte so kurz wie möglich sein. Um dieser Regel gerecht zu werden, können wir ein paar Zeilen (19 bis 25) des Konstruktors in eine separate Funktion auslagern, die wir `_init()` nennen wollen (ich glaube, der Name ist selbsterklärend).

```
sub _init {
    my ( $login, $pwd ) = @_;

    unless (
        $login and $pwd and
        ( length( $login ) <= $maxNameLen )
    ) {
        return undef;
    }

    return 1;
}
```

Die neu erstellte Funktion dient der internen Verarbeitung von Daten für die Initialisierung beim Erzeugen der Instanz. Sie ist also gewissermaßen »privat«, da sie nur von Funktionen (hier: vom Konstruktor) innerhalb des Moduls *User.pm* benutzt wird. In der Praxis hat es sich bewährt, den Namen solcher privaten Methoden mit einem Unterstrich zu beginnen. Wir werden weiter unten noch sehen, welche Alternative Perl für private Methoden bietet.

Unser Konstruktor kann nun etwas übersichtlicher gestaltet werden:

```
01 # Modul User.pm
02 package User;
03
04 # Statische Variablen, die allen Objekten (Instanzen)
05 # der Klasse gemeinsam sind
06 our $maxNameLength = 64;
07
08 # Konstruktor
09 sub new {
10     my $proto = shift( @_ );
11     my $class = ref( $proto ) || $proto;
12
13     my $self = {
14         "login"         => undef,
```

```
15          "pwd"           => undef,
16      };
17
18      # Überprüfung der Argumente
19      unless ( _init( @_ ) ) {
20          return undef;
21      }
22
23      # Setzen der Instanzattribute "login" und "pwd"
24      $self->{ "login" } = shift( @_ );
25      $self->{ "pwd" } = shift( @_ );
26
27      # Umwandeln der Hash-Referenzvariable zu einer
28      # Objektreferenz
29      bless( $self, $class );
30
31      return $self;
32  }
```

5.1.3 Instanzattribute und Instanzmethoden

Im Gegensatz zu statischen Attributen, die für alle Objekte (Instanzen) der Klasse gleichermaßen gelten, repräsentieren Instanzattribute individuelle Eigenschaften einer Objektinstanz und müssen deshalb für jede neue Instanz der Klasse separat in einer Hash-Referenzvariablen gespeichert werden, die beim Anlegen eines neuen Objekts nur in diesem definiert ist.

Beispiel für ein Instanzattribut ist das Kennwort der Klasse `User`, das als Hash-Element mit dem Key »`pwd`« abgespeichert wird, denn es kann für jedes instanzierte Objekt der Klasse unterschiedlich sein. Zur Rekapitulierung: Bei Perl werden alle Instanzattribute in einer Hash-Referenz abgelegt, wie wir bereits beim Konstruktor gesehen haben:

```
# Alle Attribute einer Objektinstanz sind Elemente
# einer Hash-Referenzvariable.
# Die Namen der Attribute sind die Hash-Keys,
# die Attributwerte sind die Hash-Values.
my $self = {
    "login"         => undef,
    "pwd"           => undef,
};
```

Nun wollen wir uns die Methoden näher ansehen, mit denen wir zum einen die Attribute des Objekts lesen oder verändern, zum anderen auch Aktionen wie »Schreibe die Daten des gesamten Objekts in eine Datei« usw. durchführen können.

Aufruf von Instanzmethoden

Weiter oben haben wir bereits gelernt, wie man statische Funktionen von Perl-Modulen aufruft:

```
# Wir laden ein anderes Perl-Modul
use OtherPackage;

# Aufruf von Modulfunktionen ohne OOP
OtherPackage::func( "a" );
# In der Funktion "func" des Packages "OtherPackage"
# lautet die Argumentliste: ( "a" )

# Aufruf mit OOP
# (Klassenname bzw. Packagename wird vom
# Interpreter heimlich eingefügt)
OtherPackage->func( "a" )
# In der Funktion "func" des Packages "OtherPackage"
# lautet die Argumentliste: ( "OtherPackage", "a" )
```

Um es kurz zu machen: Der Aufruf von Instanzmethoden ist fast identisch mit der zweiten Variante, nur schreibt man anstelle des Packagenamens die Objektreferenz:

```
# Wir laden ein anderes Perl-Modul
use OtherPackage;

# Neue Instanz der Klasse durch Aufruf der
# Konstruktorfunktion "new()" erzeugen und in $obj
# speichern
my $obj = new OtherPackage();

# Aufruf der Instanzmethode "getVersion()" über die
# Objektreferenz
my $version = $obj->getVersion();
```

Nach der Variable `$obj` folgt der Dereferenzierungsoperator ->, den wir schon im Abschnitt über Referenzvariablen kennen gelernt haben. Anschließend gibt man den Namen der aufzurufenden Methode und die Parameterliste an. In unserem Beispiel heißt die Instanzmethode `getVersion()`, und die Parameterliste ist leer.

Die aufgerufene Methode erhält dennoch einen Parameter, den der Interpreter heimlich einfügt, und zwar ist es die Objektreferenz selbst. Somit kann die Funktion `getVersion()` auf das Hash `$self` zugreifen, das eigentlich als private Variable in der Konstruktorfunktion `new()` definiert ist (und normalerweise nur dort gültig wäre).

Soweit die allgemeinen Spezialitäten der Instanzmethoden. Die Riege der OOP-Designer hat sich aber noch eine Reihe von Feinheiten einfallen lassen, mit denen ein gewisser Standard erreicht (und bitteschön von allen Programmierern auch eingehalten) werden soll.

Übrigens: Bei der Namenskonvention von Instanzmethoden gilt wie immer: Funktionsnamen beginnen mit einem Kleinbuchstaben. Ausnahmen sind (wenn auch selten) erlaubt.

So werden Instanzmethoden zum Beispiel je nach Einsatzgebiet in verschiedene Kategorien eingeteilt. Diese wollen wir nun näher betrachten.

Getter-Methoden

In der objektorientierten Programmierung werden alle Methoden, die nur für das Lesen von Attributen verwendet werden, mit dem Begriff »Getter«-Methoden bezeichnet. Das Wort kommt vom englischen Verb »to get«, das man hier mit »holen« oder »lesen« übersetzen kann.

Bei der Namenskonvention für solche Methoden hat sich eingebürgert (und man sollte diesen Standard unbedingt einhalten), dass der Funktionsname einer Getter-Methode immer mit »`get`« beginnt, woran der Name des Attributs mit einem großen Anfangsbuchstaben angehängt wird. Der Name des Attributs wiederum ist identisch mit dem Hash-Key des entsprechenden Hash-Elements in `$self`.

Lassen Sie uns doch gleich für beide Attribute unseres User-Objekts die Getter-Methoden implementieren:

```
# Getter für das Attribut "login":
sub getLogin {
    my $self = shift( @_ );
    return $self->{ "login" };
}

sub getPwd {
    my $self = shift( @_ );
    return $self->{ "pwd" };
}
```

Dem aufmerksamen Leser wird mit Sicherheit die Zeile

```
my $self = shift( @_ );
```

ins Auge springen. Was soll diese Zeile? Die Methode soll doch nur den Wert des Attributs zurückliefern und erwartet somit gar keinen Übergabeparameter.

Das ist eine der seltsamen Eigenarten, denen man manchmal bei Perl begegnet. Weiter oben hatten wir bereits einen ähnlichen Fall, als wir den Konstruktor einer Klasse kennen gelernt haben.

Beim Aufruf einer Instanzmethode (diese müssen zwingend immer über die Objekt-Referenzvariable aufgerufen werden) fügt der Interpreter »klammheimlich« einen zusätzlichen Parameter am Anfang der Argumentliste ein. Wir können uns denken,

was da auf mystische Art und Weise hinzugekommen ist, wenn wir den Variablennamen lesen. Es ist die Objektreferenz, die ja, wie wir nun wissen, gleichzeitig auch als Hash-Referenz herhalten muss.

Andere Programmiersprachen wie zum Beispiel Java bieten in ihrem Sprachwortschatz dasselbe unter einem anderen Namen an. Dort heißt die Objektreferenz für das aktuelle Objekt `this` und ist in allen Instanzmethoden automatisch definiert.

Da aber in Perl `$self` eine ganz normale Variable ist, die in der Konstruktorfunktion `new()` mit dem Scope `my` definiert wird, wäre sie normalerweise in allen anderen Funktionen nicht gültig, auch dann nicht, wenn diese Funktionen in derselben Datei definiert werden wie der Konstruktor. Also macht Perl hier einen Salto und fügt die dringend benötigte Variable eben als unsichtbaren Parameter in die Argumentliste der Instanzmethoden ein.

Somit ist es möglich, dass wir in der Methode `getLogin()` auf die Hash-Referenz zugreifen können, die eigentlich nur in der Konstruktorfunktion `new()` definiert ist. Ebenso möglich ist übrigens der Aufruf einer weiteren Instanzmethode über die Objektreferenz, da `$self` aufgrund ihres Zwitterdaseins gleichzeitig auch eine Objektreferenz ist.

Setter Methoden

Diese Art von Methoden dient dem schreibenden Zugriff auf Instanzattribute. Der Begriff »Setter« kommt vom englischen Verb »to set«, was im Deutschen »setzen« oder auch »mit einem Wert belegen« bedeutet.

Auch hier gilt als Konvention für die Namen der Methoden: Er beginnt mit `set` gefolgt vom Namen des Attributs, das man ändern möchte (wobei dessen erster Buchstabe großgeschrieben wird). Dieser ist identisch mit dem Hash-Key des entsprechenden Elements von `$self`.

Nun wieder an die Arbeit: Wir implementieren die Setter-Methoden für unsere beiden Attribute:

```
sub setLogin {
    my $self = shift( @_ );

    my ( $arg ) = @_;

    unless ( $arg ) {
        return undef;
    }

    $self->{ "login" } = $arg;

    return 1;
}
```

```
sub setPwd {
    my $self = shift( @_ );

    my ( $arg ) = @_;

    unless ( $arg ) {
        return undef;
    }

    $self->{ "pwd" } = $arg;

    return 1;
}
```

Die Zeile

```
    my $self = shift( @_ );
```

sollte von den Getter-Methoden her noch bekannt sein.

Beide Methoden erwarten genau ein Argument, nämlich den Wert für das Attribut, dessen Wert geändert werden soll. Wie wir sehen, prüfen die Methoden, ob der Wert leer ist. In diesem Fall geben sie den Status `undef` an den Aufrufer zurück, um ihm mitzuteilen, dass dies ein Fehler ist. Ansonsten wird der Value des Hash-Elements in `$self` neu gesetzt.

Die Prüfung von Argumenten in Setter-Methoden ist kein Muss, es hängt von der Programmlogik ab, ob ein leerer oder gar ein `undef`-Wert erlaubt ist oder nicht. Grundsätzlich aber sollten Setter-Methoden immer einen definierten Status an den Aufrufer zurückliefern. Wie bei allen Funktionen gilt: Ein `TRUE`-Status bedeutet OK, der `FALSE`-Status (meist) einen Fehler, der `undef`-Status immer einen Fehler.

Getter- und Setter Methoden für boolesche Attribute

Werden Getter-Methoden für Attribute implementiert, die einen booleschen Wert darstellen, dann ist die Namenskonvention für die Funktionsnamen etwas anders:

Getter-Methoden für boolesche Attribute beginnen entweder mit `is` oder mit `has`, gefolgt vom Namen der logischen Aktion, die durch das Attribut beeinflusst wird. Der Name der Aktion beginnt auch hier mit einem Großbuchstaben.

Die Namenskonvention bei Setter-Methoden hat dieselben Regeln, allerdings werden häufg gar keine Setter-Methoden für boolesche Attribute verwendet.

Damit Sie sich darunter auch etwas vorstellen können, hier ein Beispiel:

Angenommen, die Klasse `User` hätte ein boolesches Attribut namens `enabled`. Wie der Name schon vermuten lässt, handelt es sich dabei um ein Flag, das angibt, ob der User freigeschaltet (`enabled=TRUE`) oder gesperrt (`enabled=FALSE`) ist.

Jetzt kann man natürlich wie üblich eine Getter- und eine Setter-Methode implementieren. Deren Namen wären `getEnabled()` und `setEnabled()`. In aller Regel macht man hier aber eine Ausnahme und nennt die Getter-Methode `isEnabled()`. Die Methode zum Freischalten eines Users ist keine richtige Setter-Methode mehr, sondern eine normale Aktionsmethode (die aber nur das Attribut `enabled` ändert) und heißt `enable()`. Dementsprechend gibt es meist auch Methoden für die umgekehrte Aktion: `isDisabled()` gibt `TRUE` zurück, wenn der User gesperrt ist, und `disable()` sperrt den User.

Der Vollständigkeit halber noch eine Erklärung für den Begriff »Aktionsmethode«: Während Getter- und Setter-Methoden meist nur den Wert des Attributs im Hauptspeicher ändern und eine eigene spezielle Namenskonvention haben, führen Aktionsmethoden in der Regel eine »echte« Verarbeitung der Daten z.B. in der Datenbank durch. Der Name von Aktionsmethoden beginnt auch nicht mit `get` oder `set`.

Meist verwendet man für den Status von Objekten jedoch kein boolesches Attribut, sondern nimmt dafür eine Zahl oder einen String. Wir werden das im folgenden Beispiel kurz demonstrieren.

Hier ein Beispiel für boolesche Attribute:

```
...
# Konstruktor
sub new {
    ...
    my $self = {
        ...
        "status"     => "e",
        ...
    };
    ...
}
...
# Getter-Methode für den Status eines Users
# Sie liefert TRUE, wenn der User freigeschaltet ist
sub isEnabled {
    my $self = shift( @_ );

    return ( $self->{ "status" } eq "e" ) ? 1 : 0;
}

# Dasselbe in entgegengesetzter Richtung:
# Sie liefert TRUE, wenn der User gesperrt ist
sub isDisabled {
    my $self = shift( @_ );

    return ! $self->isEnabled();
}
```

Wie wir im Beispielcode sehen, heißt das Attribut (und damit der Hash-Key) für den Status des Users nicht `enabled`, sondern `status`. Dies hat den Vorteil, dass man nun mehr als nur zwei Zustände darin abspeichern kann. Ich habe für den Zustand »freigeschaltet« den Wert `e` gewählt. Dementsprechend wäre ein `d` gleichbedeutend mit »gesperrt«, was im Englischen `disabled` heißt. Die Methode `isEnabled()` gibt nur dann `TRUE` zurück, wenn das Attribut `status` den Wert `e` hat, ansonsten liefert sie `FALSE` zurück.

Eine Besonderheit findet sich in der umgekehrten Methode `isDisabled()`. Normalerweise würde man auch dort direkt den Wert des Hash-Elements abfragen. Dies hätte aber den Nachteil, dass man bei einer Änderung des Designs den Programmcode an zwei verschiedenen Stellen umschreiben müsste. Wesentlich eleganter ist die hier vorgestellte Variante. Die Methode `isDisabled()` ruft ihrerseits wieder die Methode `isEnabled()` auf und invertiert mit dem Operator »`!`« deren Rückgabewert.

So viel zu Setter- und Getter-Methoden. Alle weiteren Instanzmethoden sind normale Aktionsmethoden, die mit den Objektdaten der Instanz irgendetwas tun, zum Beispiel den Datensatz aus einer Datenbank oder einer Datei lesen bzw. dort neu anlegen oder ändern.

Es gibt noch eine weitere Art von Methoden, die sowohl statisch der Klasse (die Methoden greifen nur auf Klassenvariablen zu) als auch einer Instanz (die Methdoen greifen sowohl auf Klassenvariablen als auch auf Instanzattribute zu) zugeordnet sein können. Dies sind die privaten Methoden (obwohl diese Methoden natürlich auch wieder in Getter-, Setter- und Aktionsmethoden aufgeteilt werden können).

Private Methoden zeichnen sich dadurch aus, dass sie nur von Funktionen desselben Moduls aufgerufen werden können, nicht aber von anderem Programmcode, der das Modul lädt. Wie wir sehen werden, ist das in Perl gar nicht so einfach, denn grundsätzlich sind Funktionen nach außen bekannt und können durch die Angabe des absoluten Namespaces aufgerufen werden.

Eine Methode der »Privatisierung« von Methoden haben wir bei der `_init()`-Methode gesehen, die ich im Abschnitt »Konstruktor« beschrieben habe. Dort wird vor den eigentlichen Funktionsnamen ein Unterstrich gestellt, um sie als private Methode zu deklarieren. Aber trotzdem könnte man die Funktion von außen aufrufen, denn der Unterstrich ist nur ein Hilfsmittel für die Kennzeichnung, verhindert aber nichts.

Mit Hilfe von Referenzvariablen, die auf einen anonymen Codeblock zeigen, lässt sich aber gänzlich verhindern, dass eine Funktion von außen sichtbar und damit aufrufbar ist.

Greifen wir noch einmal unser Beispiel auf und schreiben die Methode `_init()` so um, dass sie wirklich privat und damit unsichtbar für andere Module wird:

```
# Beispiel für eine wirklich private Klassenfunktion
my $_init = sub {
    my ( $login, $pwd ) = @_;
```

```
    unless (
        $login and $pwd and
        ( length( $login ) <= $maxNameLen )
    ) {
        return undef;
    }

    return 1;
};
```

Der eigentliche Funktionscode hat sich überhaupt nicht geändert. Einzig die Definition sieht anders aus. Aus der Funktionsdefinition mit dem Bareword `sub` ist nun eine Zuweisung von anonymem Code an eine Variable geworden, die nur innerhalb desselben Moduls gültig und somit nach außen unsichtbar ist.

> Da es sich um eine Zuweisung handelt, muss nach der schließenden geschweiften Klammer ein Semikolon stehen, sonst meldet der Interpreter einen Syntaxfehler.

Wie wird nun diese seltsame Funktion aufgerufen? Sehen wir den Code im Konstruktor an:

```
# Code ohne Referenzvariable
    # Überprüfung der Argumente
    unless ( _init( @_ ) ) {
        return undef;
    }

# Code mit Referenzvariable
    # Überprüfung der Argumente
    unless ( &{ $_init }( @_ ) ) {
        return undef;
    }
```

Durch die Verwendung einer Referenzvariable muss man vor das Typkennzeichen der Variable ein Kaufmännisches Und »&« als Typkennzeichen für einen Funktionsaufruf stellen (und am besten die Variable in geschweifte Klammern setzen, das kann nie schaden). Damit teilt man dem Interpreter mit: »Hier möchte ich nicht auf den Wert der Variable zugreifen, sondern den Code ausführen, der in der Variable abgespeichert ist«.

Die Methode »toString()«

Grundsätzlich sollte jedes Objekt eine Methode bereitstellen, die dem Aufrufer alle Instanzattribute als String zurückliefert. Zum einen hilft diese Methode dem Programmierer in der Entwicklungsphase, seine Fehler zu finden, zum anderen ist es ein Quasi-Standard, dass jedes Objekt die Methode `toString()` implementiert. Also halten wir uns an den Standard und schreiben für unsere Klasse `User` die `toString()`-Methode:

```
sub toString {
    my $self = shift;

    my $res = "Attribute von User:" .
        "\n\tlogin ='" .
        ( defined( $self->getLogin() ) ?
            $self->getLogin() : "undef" ) .
        "'\n\tpwd = '" .
        ( defined( $self->getPwd() ) ?
            $self->getPwd() : "undef" ) .
        "'\n";

    return $res;
}
```

Unsere `toString()`-Methode gibt einen String zurück, der alle Instanzattribute mit ihrem Namen und ihrem Wert enthält. Die Attribute werden hier im Beispiel durch ein Zeilenende-Zeichen voneinander getrennt und mit einem TAB-Zeichen eingerückt.

Der aktuelle Wert eines Attributs wird jeweils auf `undef` überprüft. In diesem Fall wird statt des Werts der String `undef` eingesetzt. Das ist notwendig, weil der Interpreter andernfalls eine Fehlermeldung liefern würde, wenn man versucht, den String auszugeben.

Bei komplexeren Objekten, wo Attribute wiederum Objekte oder Hashes oder Arrays sein können, muss die `toString()`-Methode natürlich entsprechend angepasst werden.

Nehmen wir wieder unser bisheriges Beispiel der Klasse `User` und sehen uns an, was die Methode `toString()` zurückliefert:

```
# Hauptprogramm
#!D:/Perl/bin/perl.exe -w
use strict;

use User;

my $user = new User( "Egon", "Wahr" );
unless ( $user ) {
    print( STDERR "Fehler beim Anlegen des Objekts\n" );
    exit( 1 );
}

print( $user->toString(), "\n" );

exit( 0 );
```

Wenn wir das Skript ausführen, erhalten wir folgende Ausgabe:

```
login = 'Egon'
pwd = 'Wahr'
```

5.1.4 Fehlermeldungen von Klassen

Bisher haben unsere Klassenmethoden einfach nur den Pseudowert `undef` zurückgeliefert, wenn Fehler aufgetreten sind. Das ist speziell in der Entwicklungsphase von Modulen nicht besonders hilfreich, da man ja auch wissen möchte, was passiert ist. Grundsätzlich verbietet sich die direkte Ausgabe von Fehlermeldungen nach `STDOUT` oder `STDERR` in Modulen, da man nicht wissen kann, in welcher Umgebung das Modul benutzt wird.

Speziell im CGI-Umfeld können Fehlermeldungen nicht einfach ausgegeben werden, weil der Anwender im Browser dann meist einen »Server Error 500« erhalten würde (der mit Sicherheit unbeliebteste Fehler bei Programmierern).

Dasselbe Problem hat man bei Debug-Informationen, die man im Modul zusätzlich zu den Fehlermeldungen zur Verfügung stellen möchte. Irgendwie muss diese Information zum Programmcode gelangen, der das Package benutzt.

Die einfachste Lösung ist eine statische Klassenvariable, die die Fehlermeldungen aufnimmt. Dazu implementiert man noch eine ebenfalls statische Methode, mit deren Hilfe man die Fehlermeldungen von außen lesen kann.

> Wird eine Klasse in Multi-Threading-Umgebung benutzt, heißt es aufpassen, da hier eine Klasse normalerweise nur ein einziges Mal geladen wird. Dies ist z.B. in Java die Regel. Bis dato wird Perl noch nicht »multi-threaded« benutzt, deswegen ergeben sich hier meist noch keine Probleme bezüglich der Synchronisation von Daten im Hauptspeicher.
>
> In einer Multi-Threading-Umgebung hat man grundsätzlich das Problem, dass die gemeinsamen Daten einer Klasse synchronisiert werden müssen, d.h., es ist darauf zu achten, dass die unterschiedlichen Threads die Klassendaten nicht gegenseitig überschreiben.
>
> Ein Ausweg wäre, die Fehlerbehandlung nicht statisch in der Klasse zu verarbeiten, sondern separat für jede Objektinstanz. Auf Deutsch heißt das nichts anderes, als dass man für die Verarbeitung von Fehlern Instanzattribute und Instanzmethoden benutzt. Das geht aber nur, wenn vom Konstruktor eine gültige Objektreferenz zurückgeliefert wird.

Sehen wir uns eine Beispiel-Implementierung für Fehlermeldungen an:

```
package User;

use strict;

# Alle Fehlermeldungen der Klasse landen in einer
# Array-Variable, auf die man von außen nicht direkt,
# sondern nur über eine Getter-Methode Zugriff hat.
my @errors = ();
```

```
# Getter-Methode zum Auslesen der Fehlermeldungen
# Sie gibt alle Fehlermeldungen als Elemente einer
# Liste zurück.
sub getErrors {
    return @errors;
}

# Methode zum Löschen der Fehlermeldungen
sub clearErrors {
    @errors = ();
}

# Abfragemethode, mit der man von außen feststellen
# kann, ob ein Fehler aufgetreten ist
# Sie liefert dann TRUE zurück
sub hasErrors {
    return @errors ? 1 : 0;
}
```

Natürlich benötigen wir nun auch noch eine private Klassenmethode, mit deren Hilfe andere Funktionen unserer Klasse Fehlermeldungen erzeugen können:

```
sub _err {
    my $str = "";

    foreach my $arg ( @_ ) {
        $str .= defined( $arg ) ? $arg : "undef";
    }

    push( @errors, $str );
}
```

Die Methode ist so geschrieben, dass mehrere Argumente im Funktionsaufruf angegeben werden können. Um Fehlermeldungen vom Interpreter vorzubeugen, falls eines der Argumente `undef` ist, werden in einer Schleife alle Parameter auf diesen Wert hin überprüft. Es wird dann stattdessen der String `undef` angehängt.

Nun wollen wir uns ansehen, wie die anderen Funktionen des Moduls geändert werden müssen, um Fehlermeldungen zu erzeugen. Als Beispiel nehmen wir einmal die Funktion `_init()`:

```
sub _init {
    my ( $login, $pwd ) = @_;

    # Präfix, das vor die eigentliche Fehlermeldung
    # gesetzt wird, damit man weiß, welche Funktion
    # die Meldung abgesetzt hatte.
    my $prefix = "_init():";
```

```
        # Fehlermeldung, falls das Argument für den
        # Benutzernamen undef ist.
        unless ( defined( $login ) ) {
            _err( "$prefix undefined login" );
            return undef;
        }

        # Fehlermeldung, falls das Argument für den
        # Benutzernamen FALSE ist.
        unless ( $login ) {
            _err( "$prefix empty login" );
            return undef;
        }

        # Fehlermeldung, falls das Argument für das
        # Kennwort undef ist.
        unless ( defined( $pwd ) ) {
            _err( "$prefix undefined pwd" );
            return undef;
        }

        # Fehlermeldung, falls das Argument für das
        # Kennwort FALSE ist.
        unless ( $pwd ) {
            _err( "$prefix empty pwd" );
            return undef;
        }

        return 1;
    }
```

Wie wir sehen, wird der Programmcode um so länger, je detaillierter die Meldungen sind. Aber das ist leider ein Naturgesetz, dem man machtlos gegenübersteht, also Kopf hoch und durch!

Übrigens: Wenn eine Funktion des Moduls wiederum eine andere Funktion aufruft usw., dann erhalten wir im Fehlerfall eine geschachtelte Fehlermeldungsliste, die man landläufig auch als »Error Stack« bezeichnet. Nehmen wir als Beispiel den Konstruktor, der seinerseits die Funktion `_init()` aufruft. Wenn dort ein Fehler auftritt, dann erzeugt sie die erste Fehlermeldung.

Der Konstruktor prüft den Rückgabewert von `_init()` und schreibt im Fehlerfall ebenfalls eine Meldung in das Array `@errors`. Das Ganze ergibt einen so genannten »Stack Trace«. Man kann also genau verfolgen, wo welcher Fehler aufgetreten ist, und wer wen wann aufgerufen hat.

Jetzt fehlt noch der entsprechende Code im Skript, der die Fehlermeldung entgegennimmt:

```
...
my $user = new User( "Willy", "secret" );
if ( ( ! $user ) or User::hasErrors() ) {
    # es ist ein Fehler aufgetreten, es werden alle
    # Fehlermeldungen nach STDOUT ausgegeben
    my @errors = User::getErrors();
    print( join( "\n ", @errors ), "\n" );
    exit( 1 );
}
```

Nun wollen wir uns einem Thema widmen, das wohl eines der wichtigsten bei der objektorientierten Programmierung darstellt: der Vererbung.

5.2 Vererbung

Die objektorientierte Programmierung stellt mit der Möglichkeit, Attribute und Methoden von Objekten weiter zu vererben, eines der leistungsfähigsten Programmiermittel zur Verfügung. Im Englischen wird dafür der Begriff »Inheritance« verwendet.

Das Vererbungsprinzip ist das Folgende:

Zunächst entwickelt man eine möglichst allgemeine Klasse mit wenigen Attributen und Methoden. Dann implementiert man weitere Klassen, welche diese allgemeine Klasse erweitern, indem sie neue Attribute und Methoden hinzufügen. Alle bereits von der allgemeinen Klasse entwickelten Attribute und Methoden werden den »Kind«-Klassen, die aus der allgemeinen Klasse hervorgehen, weitervererbt, d.h., sie können diese benutzen, als seien sie neu implementiert worden. Man muss also den bereits geschriebenen Programmcode nicht noch einmal schreiben. Der Begriff »Kind« heißt im Englischen übrigens »child«, die Mehrzahl ist »children«.

Dieses Spiel kann man nun beliebig weiterspielen, mit jeder weiteren »Ableitung«, sprich Vererbung, entsteht eine Klasse, die spezifischer ist als die »Eltern«-Klasse, aus welcher sie hervorgeht. Kurz zum Begriff »Eltern«: Dazu sagt man im Englischen »parent«.

Ein Beispiel für eine sehr allgemeine Klasse haben wir bereits in Form unseres Moduls *User.pm* kennen gelernt. Ausgehend von dieser Basisklasse wollen wir nun die Funktionalität erweitern, indem wir eine davon abgeleitete Klasse implementieren, die zusätzlich zu den bereits bekannten Attributen `login` und `pwd` noch das Attribut `email` anbietet. Ich möchte die Klasse `AnonymousUser` nennen, weil sich hinter dem Benutzer

keine Person verbirgt, sondern irgendjemand, der nur durch seine E-Mail-Adresse bekannt und damit sozusagen »anonym« ist. Den Programmcode speichern wir in der Datei *AnonymousUser.pm* ab.

Die erste Frage, die der aufmerksame Leser auf der Zunge haben wird, ist: »Wie leitet man eine Klasse ab?«

Die Antwort auf diese Frage ist gar nicht so schwer, wie man meinen könnte. Das einzige, was man tun muss, ist die Definition einer neuen Variable und eine geringfügige Änderung im Programmcode des Konstruktors.

5.2.1 Die Variable @ISA

Damit eine Methode aus der übergeordneten Klasse (zu deutsch »Eltern-Klasse«, englisch »parent class« oder auch »super class«) benutzt werden kann, muss im Modul der daraus abgeleiteten Klasse (zu deutsch »Kind-Klasse«, englisch »child class« oder auch »derived class«) die Variable `@ISA` definiert und darin alle Packagenamen der übergeordneten Klassen als Elemente des Arrays angegeben sein.

Damit Sie ob der neuen Begriffe nicht völlig im Regen stehen, wollen wir uns das Ganze in einem Beispiel ansehen. Den gezeigten Programmcode speichern wir, wie gesagt, in der Datei mit dem Namen *AnonymousUser.pm* ab, das Package heißt also `AnonymousUser`.

```
# Modul AnonymousUser.pm
package AnonymousUser;

use strict;

# Laden des Moduls für die Eltern-Klasse
use User;

# Übernehmen der Methoden aus der Eltern-Klasse in die
# Kind-Klasse mit Hilfe der Variable @ISA
our @ISA = qw( User );
...
```

Was bewirkt die Variable »`@ISA`«?

Die Variable `@ISA` veranlasst den Interpreter, Funktionen nicht nur in dem Package zu suchen, wo der Aufruf über die Objektreferenz steht, sondern auch in den Packages, die als Liste in dem Array `@ISA` stehen. Erst mit dieser Variable wird Vererbung überhaupt möglich. Lassen Sie mich die Zusammenhänge der Vererbung mit `@ISA` anhand eines Schaubildes erläutern:

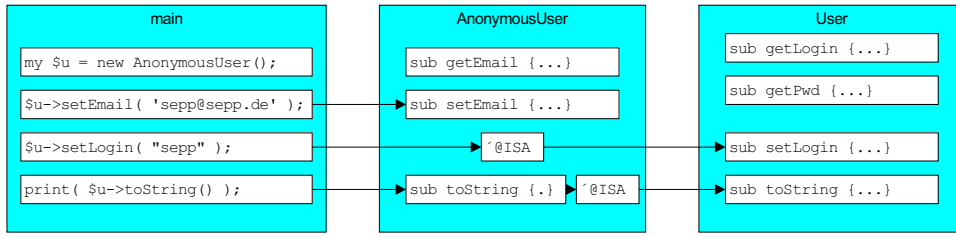

Abbildung 5.3: Vererbung mit @ISA

Im Hauptprogramm (Package `main`) wird eine Instanz der Klasse `AnonymousUser` erzeugt. Das erfolgt mit der Anweisung:

```
my $u = new AnonymousUser();
```

Kleiner Hinweis: Der Einfachheit halber habe ich die Parameter weggelassen. Wir werden weiter unten noch sehen, was im Konstruktor von `AnonymousUser` zu tun ist.

Mit dem nächsten Statement

```
$u->setEmail( 'sepp@sepp.de' );
```

setzt man das Attribut `email`. Dieses sowie die notwendige Instanzmethode `setEmail()` werden im Package `AnonymousUser` definiert.

Nun wird im Hauptprogramm mit

```
$u->setLogin( "sepp" );
```

das Attribut `login` gesetzt.

Schauen Sie noch einmal genau auf den Anfang der Zeile: Es wird die Funktion `setLogin()` über die Objektreferenz `$u` aufgerufen. `$u` ist aber eine Referenz auf ein Objekt der Klasse `AnonymousUser`. In dieser Klasse aber gibt es gar keine Funktion `setLogin()`, vielmehr ist die Methode in der Klasse `User` implementiert.

Genau das versteht man unter Vererbung: Dank der Variable `@ISA` scheint es so, als seien alle Attribute und Methoden von `User` auch in `AnonymousUser` vorhanden, obwohl wir in diesem Modul keine einzige Codezeile dafür schreiben müssen. Aus der Sicht des Hauptprogramms existiert gar keine Klasse `User`. Dort laden wir nur die Klasse `AnonymousUser`:

```
use AnonymousUser;
```

Hinweis: `@ISA` sollte grundsätzlich nur den Namen der Eltern-Klasse enthalten, auch wenn man grundsätzlich beliebig viele Klassen in das Array packen kann. Der Interpreter geht bei der Suche nach Funktionen die Liste von links nach rechts vor. Dieser

Mechanismus wird auch »mehrfache Vererbung« genannt und bereitet seit seiner Einführung in C++ allen Programmierern heftige Kopfschmerzen. Also: Lassen Sie die Finger von der »mehrfachen Vererbung«!

Jetzt zum zweiten Teil der Antwort auf die Frage »Wie leitet man eine Klasse ab?«:

Man schreibt in den Konstruktor der Kind-Klasse einen Aufruf des Konstruktors der Eltern-Klasse und erweitert anschließend das Hash, in dem die Attribute gespeichert sind. In unserem Beispiel ist `AnonymousUser` die Kind-Klasse und `User` die Eltern-Klasse. Sehen wir uns nun den Konstruktor der Kind-Klasse an:

```
package AnonymousUser;

use strict;

# Mit der folgenden Direktive wird die Eltern-Klasse
# geladen, von der wir die Attribute und Methoden
# vererbt bekommen.
use User;

# Vererbung heißt @ISA
our @ISA = qw( User );

# Konstruktor
sub new {
    my $proto = shift( @_ );
    my $class = ref( $proto ) || $proto;

    # Nun müssen wir als Erstes den Konstruktor
    # unserer Eltern-Klasse aufrufen, dem wir
    # unsere eigenen Aufrufparameter mitgeben.
    # $self wird hier nicht als Hash-Referenz
    # definiert, sondern erhält den Rückgabewert
    # des Konstruktors der übergeordneten Klasse.
    my $self = new User( @_ );

    # Wir müssen noch überprüfen, ob etwas
    # schief gegangen ist (der Einfachheit halber
    # ist hier die Behandlung von Fehlermeldungen
    # nicht implementiert; wir liefern einfach nur
    # "undef" zurück, falls ein Fehler aufgetreten ist).
    unless ( $self ) {
        return undef;
    }

    # $self ist momentan noch eine Objektreferenz
    # auf eine Instanz der Eltern-Klasse.
    # Mit der folgenden Anweisung ändern wir die
    # Referenz, so dass $self zu einer Referenz
    # unserer eigenen Instanz wird.
    bless( $self, $class );
```

```
    # Jetzt fügen wir das neue Attribut "email"
    # hinzu. Den Wert dieses Attributs erhalten wir
    # aus dem dritten Parameter von @_
    # Die ersten beiden sind "login" und "pwd".
    # Auch hier habe ich der Einfachheit halber
    # die Fehlerbehandlung weggelassen.
    $self->setEmail( $_[ 2 ] );

    # Wir sind fertig, also geben wir die
    # Objektreferenz zurück.
    return $self;
}
```

Aus dem Hauptprogramm heraus wird jetzt nur der Konstruktor der abgeleiteten Klasse aufgerufen:

```
# Hauptprogramm
...
use AnonymousUser;

my $user = new AnonymousUser(
    "sepp", "secret", 'sepp@sepp.de'
);
```

Wie wir sehen, taucht das Modul *User.pm* im Hauptprogramm überhaupt nicht auf. Vielmehr wird es vom Package `AnonymousUser` geladen.

Da in der E-Mail-Adresse das Zeichen »@« vorkommt, muss man den String entweder in einfache Quotes setzen oder die Sonderbedeutung von »@« durch einen vorangestellten Backslash entwerten. Ich habe mich für die erste Variante entschieden.

Mit der Instanz, die uns der Konstruktor von `AnonymousUser` zurückliefert, können wir nun sowohl auf Methoden von `AnonymousUser` als auch von `User` zugreifen:

```
# Wir rufen die Methode "getEmail()" auf, die in
# "AnonymousUser" implementiert ist.
my $email = $user->getEmail();

# Jetzt rufen wir "getLogin()" auf. Diese Methode
# ist in "User" implementiert.
# Dem Hauptprogramm kann es aber egal sein,
# wo welche Methode definiert wurde, alle
# Funktionsaufrufe erfolgen über unsere
# Objektreferenz "$user".
my $login = $user->getLogin();
```

Der Vollständigkeit halber implementieren wir nun noch die Getter- und Setter-Methoden für das neue Attribut `email` im Modul *AnonymousUser.pm*:

```
# Getter
sub getEmail {
    my $self = shift( @_ );
    return $self->{ "email" };
}

# Setter
sub setEmail {
    my $self = shift( @_ );

    my $arg = shift( @_ );

    # Hier könnte noch eine Überprüfung der email
    # stehen
    $self->{ "email" } = $arg;

    return 1;
}
```

Damit haben wir die Klasse `User` abgeleitet und die Kind-Klasse `AnonymousUser` implementiert, die alle Attribute und Methoden von `User` erbt.

Ein wichtiger Punkt ist aber noch offen: Die Methode `toString()`. Bisher ist die Funktion nur in »`User`« implementiert, nicht aber in `AnonymousUser`. Für den Aufruf der Methode vom Hauptprogramm aus macht das nichts aus, allerdings ist das neu hinzugekommene Attribut `email` im Package `User` unbekannt und wird deshalb nicht berücksichtigt. Die Folge davon ist, dass der Programmcode

```
...
my $user = new AnonymousUser(
    "Sepp", "secret", 'sepp@sepp.de'
);
print( $user->toString() );
```

nicht alle Attribute ausgibt:

```
Attribute von User:
        login = 'Sepp'
        pwd = 'secret'
```

Es wird ja die Methode `toString()` von `User` aufgerufen, und dort ist das Attribut `email` unbekannt.

Als Lösung für dieses Problem kommt ein wichtiger Aspekt der objektorientierten Programmierung ins Spiel, den man »Overloading« nennt, was im Deutschen so viel wie »Überladen« bedeutet.

5.2.2 Overloading

Unter dem Begriff »Overloading« versteht man das Aufrufen einer namensgleichen Methode der übergeordneten Eltern-Klasse aus der Kind-Klasse heraus. Um Ihnen gleich ein Anschauungsbeispiel zu liefern:

Im letzten Abschnitt hatten wir das Problem, dass für die `toString()`-Methode von `User` das Attribut `email` unbekannt ist, denn es wurde ja erst von der abgeleiteten Kind-Klasse `AnonymousUser` hinzugefügt. Um jetzt wirklich alle Attribute in einen String umzuwandeln, muss also auch in der Kind-Klasse eine Methode `toString()` implementiert werden, die alle Attribute der Klasse `AnonymousUser` umwandelt. Zusätzlich aber muss diese Funktion die `toString()`-Methode der Eltern-Klasse aufrufen, damit auch deren Attribute erfasst werden. Die Frage ist nur, wie?

Perl bietet hierfür eine Reihe von Möglichkeiten an, von denen ich aber nur die sauberste präsentieren möchte. Das Schlüsselwort heißt `SUPER`. Mit diesem Schlüsselwort, hinter dem sich eine Pseudoklasse verbirgt, wird der Interpreter dazu veranlasst, in denjenigen Klassen nach der Funktion zu suchen, die in der Liste von `@ISA` stehen. Zur Demonstration möchte ich die Methode `toString()` der Kind-Klasse `AnonymousUser` gleich implementieren:

```
package AnonymousUser;
...
sub toString {
    my $self = shift( @_ );

    my $res = $self->SUPER::toString();

    $res .= "\nAttribute von AnonymousUser:" .
        "\n\temail = '" .
        ( defined( $self->getEmail() ) ?
            $self->getEmail() : "undef" ) .
        "'\n";

    return $res;
}
```

Die wichtige Zeile ist

```
my $res = $self->SUPER::toString();
```

denn hier wird die Methode der Eltern-Klasse aufgerufen.

Man hätte natürlich auch den Klassennamen statt SUPER hinschreiben können, aber in diesem Fall wäre der Aufruf im Programmcode»hart verdrahtet«. SUPER ist die elegantere Alternative, weil der Code bei einer Änderung von Klassennamen gleich bleiben kann. Das Statement

```
$self->SUPER::toString();
```

besagt einfach: Lieber Interpreter, rufe bitte die Methode toString() von meiner Eltern-Klasse auf, egal, wie diese heißt. Dieser Mechanismus funktioniert bei mehr als nur einer abgeleiteten Kind-Klasse. Wenn wir zum Beispiel eine neue Klasse PersUser implementieren, die wiederum von AnonymousUser abgeleitet ist, und der neuen Klasse keine toString()-Methode spendieren, dann ruft der Interpreter die Funktion aus der Klasse AnonymousUser auf.

Bevor wir zum nächsten neuen Begriff kommen, möchte ich Sie noch auf eine weitere Eigenart von Perl aufmerksam machen. Weiter oben, als wir die Vererbung besprochen hatten, erzählte ich Ihnen, dass die Eltern-Klasse keine Attribute der Kind-Klasse kennt. Das ist auch richtig, zumindest, was den Namen des Attributs angeht. Jedoch werden alle Attribute, auch die der abgeleiteten Klassen, in einem gemeinsamen Hash gespeichert, und dieses kennt die Eltern-Klasse sehr wohl. Am besten, ich zeige Ihnen die Folgen anhand eines Beispiels.

Lassen Sie uns die toString()-Methode der Klasse User ein wenig verändern:

```
package User;

# Neue Implementierung der toString()-Methode von "User"
sub toString {
    my $self = shift( @_ );
    my $res = "Attribute von User:\n";
    foreach my $key ( sort( keys( %{ $self } ) ) ) {
        my $val = defined( $self->{ $key } ) ?
            $self->{ $key } : "undef";

        $res .= "\t$key = '$val'\n";
    }

    return $res;
}
```

Die neue Variante von toString() greift jetzt nicht mehr über die Namen der Attribute auf die Hash-Elemente zu, sondern anonym, indem sie sich eine Liste der Hash-Keys holt. Die Funktion gibt jetzt also einfach alle im Hash gespeicherten Elemente aus.

Wenn wir jetzt nach dieser Änderung die `toString()`-Methode von `AnonymousUser` aufrufen:

```
# Hauptprogramm

use AnonymousUser;

my $user = new AnonymousUser(
    "Sepp", "secret", 'sepp@sepp.de'
);

...

print( $user->toString() );
```

dann erhalten wir folgende Ausgabe:

```
Attribute von User:
        email = 'sepp@sepp.de'
        login = 'Sepp'
        pwd = 'secret'

Attribute von AnonymousUser:
        email = 'sepp@sepp.de'
```

Ich habe nur die Funktion der Klasse `User` geändert, nicht die aus der Klasse `AnonymousUser`.

Wie wir deutlich sehen, sind alle Attribute, sowohl die der Eltern- als auch die der Kind-Klasse, in einem einzigen Hash abgelegt.

Das bedeutet im Klartext, dass Sie in keinem Fall neue Attribute in Kind-Klassen hinzufügen dürfen, die denselben Namen haben wie bereits in Eltern-Klassen existierende Attribute.

Nachdem wir den Begriff »Overloading« alle perfekt buchstabieren können, will ich Ihnen gleich noch einen weiteren vorsetzen: »Overriding«.

5.2.3 Overriding

Nachdem Sie wissen, was »Overloading« bedeutet, ist der neue Begriff »Overriding« ein Kinderspiel. Er bedeutet nämlich im Prinzip dasselbe, nur fehlt der Aufruf der gleichnamigen Funktion aus der übergeordneten Eltern-Klasse:

```
package AnonymousUser;
...
sub toString {
    my $self = shift( @_ );
```

```
    my $res = "Attribute von AnonymousUser:" .
        "\n\temail = '" .
        ( defined( $self->getEmail() ) ?
            $self->getEmail() : "undef" ) .
        "'\n";

    return $res;
}
```

Der Unterschied zwischen dieser Variante von `toString()` und der vorher gezeigten besteht wirklich nur darin, dass jetzt der Funktionsaufruf der Eltern Klasse fehlt.

Bevor wir »Overriding« in Aktion sehen, möchte ich Ihnen ein paar Aktionsmethoden zeigen, mit denen wir die Datensätze der Benutzer aus einer Datei lesen oder in eine Datei schreiben können, denn ohne solche Aktionen wären unsere Klassen ein wenig langweilig. Im Zuge der Neuimplementierungen werden wir den bisher präsentierten Demonstrationscode so ändern, dass er für die Praxis tauglich wird.

Beginnen wir bei der Klasse `User`. Bisher haben wir alle benötigten Attribute als Parameterliste im Konstruktor angegeben:

```
my $user = new User( "Sepp", "secret" );
```

Dieses Programmdesign ist in der Praxis meist untauglich, weil damit die Positionen der Argumente festgelegt sind und später nicht geändert werden können, ohne dass sich damit auch das API des Moduls ändert. Jeder, der Ihr Modul benutzt, muss seinen Programmcode dann also ebenfalls umschreiben.

Es gibt zwei Alternativen, die flexibler sind: Man erlaubt einen Konstruktor ohne Argumente und setzt die Attribute über Setter-Methoden (ein leerer Konstruktor wird auch »Default-Konstruktor« bzw. »Standard-Konstruktor« genannt):

```
my $user = new User();

$user->setLogin( "Sepp" );
$user->setPwd( "secret" );
```

Der Konstruktor wird dann sehr kurz:

```
sub new {
    my $proto = shift( @_ );
    my $class = ref( $proto ) || $proto;

    my $self = {
        "login"         => undef,
        "pwd"           => undef,
    };
```

```
    # Umwandeln der Hash-Referenzvariable in eine
    # Objektreferenz
    bless( $self, $class );

    return $self;
}
```

Natürlich muss man in diesem Fall in den Setter-Methoden die Parameter überprüfen, damit von außen kein Müll in das Objekt fließen kann. Der große Vorteil dieser Variante ist, dass man keine Abhängigkeiten in Parametern hat, aus dem einfachen Grund, weil es keine Parameter im Konstruktor gibt.

Der leere Konstruktor hat den weiteren Vorteil, dass er in jedem Fall eine gültige Objektreferenz zurückliefert. Damit kann man zum Beispiel Fehlermeldungen in der Instanz selbst statt in einer statischen Klassenvariablen speichern (dies kann speziell bei Multi-Threading-Umgebungen wichtig werden).

Allerdings hat diese Implementierung auch einen Nachteil, wenn man zum Beispiel Pflichtattribute in der Klasse vorsieht, die in jedem Fall mit einem gültigen Wert vorhanden sein müssen, bevor eine bestimmte Aktion ausgeführt werden darf. So ergibt es keinen Sinn, nach einem Benutzer zu suchen, wenn dessen Login `undef` ist. Das ist aber durch den Default Konstruktor (auch »Standard-Konstruktor« genannt) ohne Parameter möglich. Also muss die Methode, mit der nach Benutzern gesucht werden soll, prüfen, ob alle Pflichtattribute gültig sind.

Die zweite Alternative bietet sich in Form eines Hashs an, das dem Konstruktor als Referenz übergeben wird:

```
my %args = (
    "login" => "Sepp",
    "pwd" => "secret",
);

my $user = new User( \%args );
```

Damit hat man eine große Freizügigkeit, wenn im Laufe der Weiterentwicklung des Moduls neue Attribute hinzukommen, denn die Positionen der Elemente sind bei einem Hash ja irrelevant. Mit dieser Variante sind Pflichtattribute kein Problem. Allerdings müssen die Namen (Keys) der Hash-Elemente im API des Moduls dokumentiert werden. So muss in unserem Fall der Benutzername durch den Hash-Key `login` angegeben sein.

Wir wollen hier für unsere Beispiele die Variante eines Konstruktors ohne Argumente verwenden.

Implementieren wir zunächst die Methode `write()`, mit der wir den Datensatz der Instanz in eine Datei schreiben. Bevor mit dem Kodieren begonnen werden kann, müssen wir uns die Struktur der Datei für die Datensätze überlegen. Wie wäre es mit folgendem Design?

Jeder Datensatz steht in einer einzelnen Zeile der Datei, die Attribute eines Datensatzes werden durch ein TAB-Zeichen getrennt. Die Reihenfolge der Attribute lautet: `login, pwd, status.`

Ja, Sie sehen richtig, in unserem Code wird auch das Attribut `status` unterstützt, über das wir weiter oben bereits gesprochen hatten.

Die Datei könnte also z.B. so aussehen:

```
Sepp\tsecret\te
egon\tmypwd\td
```

Die Datei hat zwei Datensätze, der erste Benutzer ist freigeschaltet (`status=e`), der zweite gesperrt (`status=d`)

Nun können wir die Methode `write()` kodieren:

```
01 sub write {
02     my $self = shift( @_ );
03
04     my $login = $self->getLogin();
05     my $pwd = $self->getPwd();
06     my $st = $self->getStatus();
07
08     unless ( $login and $pwd ) {
09         return undef;
10     }
11
12     use FileHandle;
13
14     my $fh = new FileHandle( $dataPath, "r+" );
15     unless ( $fh ) {
16         return undef;
17     }
18
19     my @users = ();
20     my $written = undef;
21
22     while ( defined( my $line = <$fh> ) ) {
23         chomp( $line );
24         if ( $line =~ /^([^\t]+)\t[^\t]+\t.$/ ) {
25             if ( $1 eq $login ) {
26                 push( @users, "$login\t$pwd\t$st" );
27                 $written = 1;
28             }
```

```
29              else {
30                  push( @users, $line );
31              }
32          }
33      }
34
35      unless ( $written ) {
36          push( @users,  "$login\t$pwd\t$st" );
37      }
38
39      unless ( seek( $fh, 0, 0 ) ) {
40          $fh->close();
41          return undef;
42      }
43
44      unless ( truncate( $fh, 0 ) ) {
45          $fh->close();
46          return undef;
47      }
48
49      foreach my $line ( @users ) {
50          print( $fh "$line\n" );
51      }
52
53      $fh->close();
54
55      return 1;
56  }
```

Nachdem Sie sich den Code eine Weile angesehen haben, könnte es durchaus sein, dass Sie ein paar Fragen haben.

Bis zur Zeile 08 sollte alles klar sein. Wir lesen die Attribute, die wir in die Datei schreiben wollen, mit Getter-Methoden aus dem Objekt.

Aber schon in Zeile 08

```
08      unless ( $login and $pwd ) {
```

könnten Sie sich vielleicht wundern, warum ich zwar die Attribute `login` und `pwd` überprüfe, nicht aber den Status, der im Attribut `status` gespeichert wird.

Die Werte für den Benutzernamen und das Kennwort kommen normalerweise von außen, d.h. von Perl-Skripts oder anderen Modulen, die wir nicht kennen. Da wir weiter oben gesagt haben, wir erlauben im Konstruktor eine leere Parameterliste, könnte jemand diesen Konstruktor verwenden und sofort danach die `write()`-Methode aufrufen, ohne dass er vorher die Setter-Methoden für die Attribute benutzt. Die Folge wären `undef`-Werte und vermutlich eine Fehlermeldung des Interpreters. Das können wir natürlich nicht zulassen, deshalb die Überprüfung.

Den Status jedoch müssen wir nicht prüfen, denn dieser wird ausschließlich durch unseren eigenen Programmcode in *User.pm* versorgt. Wir müssen einfach nur sicherstellen, dass er niemals `undef` sein kann. Das geschieht bereits im Konstruktor:

```
...
my $self = {
    "login"    => undef,
    "pwd"      => undef,
    "status"   => "e",
};
...
```

Wie Sie sehen, wird der Status im Konstruktor immer mit einem Defaultwert belegt und kann somit gar nicht `undef` sein. Von außen kann er anschließend mit den Methoden `disable()` und `enable()` nur noch auf `d` oder `e` geändert, niemals aber `undef` gemacht werden, also erübrigt sich eine Prüfung.

Alle Objektattribute, die von »außen« gesetzt werden können, müssen genau daraufhin geprüft werden, ob sie gültig sind. Werden aber Attribute durch Methoden gesetzt, die im Klassenmodul definiert sind, kann diese Prüfung entfallen (wenn man sauber programmiert). Das führt häufig zu Methoden, die zwar dasselbe tun, aber unterschiedliche Prüfungen durchführen.

Ein Beispiel dafür ist die Methode zum Lesen eines Datensatzes aus einer Datei oder einer Datenbank. Hier muss kein einziges Attribut überprüft werden, da diese ja bereits vorher beim Anlegen des Datensatzes einem Check unterlagen. Wir werden also häufig Setter-Methoden in zweifacher Ausführung sehen: eine öffentliche Methode (im Englischen auch »public method« genannt), bei der eine Prüfung der Argumente stattfindet, und eine private Methode, die keine Prüfung durchführt.

Die Überlegung, die ich Ihnen gerade erläutert habe, führt an sehr vielen Stellen im Programmcode dazu, dass man unnötigen Code weglassen kann, wenn man vorher genaue Überlegungen anstellt. Das ist gerade bei umfangreicheren Programmen wichtig, wenn es auf Hochgeschwindigkeit ankommt. Überlassen Sie bitte nicht alles dem Compiler. Gerade die jüngeren Entwickler bekommen in Schulen und Universitäten oft den falschen Eindruck, dass heutige Compiler Alleskönner sind und einem den Verstand abnehmen.

Die nächste Frage dürften Sie vielleicht in Zeile 12 haben:

```
12     use FileHandle;
```

Nein, ich habe keinen Fehler gemacht. Das Laden von Modulen kann an beliebiger Stelle im Programmcode erfolgen, auch in Funktionen. Der Vorteil davon ist derselbe, den man bei der Definition und Benutzung von Variablen hat. Deklarieren Sie erst dann, wenn es unbedingt sein muss, nicht alles am Beginn des Programms.

Hier der Vorteil: Sollten Sie sich nach einiger Zeit entschließen, den Code auf Datenbankzugriff umzustellen, dann ändern Sie sicherlich die Methode `write()` (und `read()`, aber dazu später). Hätten Sie aber die `use`-Direktive an den Anfang des Moduls gestellt, dann wäre die Gefahr hoch, dass sie nicht gelöscht wird und weiterhin als »Leiche« im Programmcode steht.

Auch Zeile 14 ist erwähnenswert:

```
14      my $fh = new FileHandle( $dataPath, "r+" );
```

Wo kommt die Variable `$dataPath` her?

Die Frage ist berechtigt. Um die Wahrheit zu sagen: Ich habe `$dataPath` eben erst als globale Packagevariable erfunden. Das bedeutet natürlich, dass wir in unser Modul *User.pm* noch die Variablendefinition mit aufnehmen müssen:

```
our $dataPath = "C:/temp/users.data";
```

Der tatsächliche Pfad für die Datei ist zweitrangig und hängt auch vom verwendeten Betriebssystem ab. Wir werden weiter unten noch sehen, wie man mit Hilfe von Propertiesdateien feste Pfadnamen in Programmen vermeidet, für unser Beispiel jedoch soll dies genügen.

Interessant ist auch der Dateimodus. Die Datei wird sowohl zum Lesen als auch zum Schreiben geöffnet. Vorsicht ist allerdings geboten, denn die Datei muss bereits existieren, sonst straft uns der Interpreter mit einer Fehlermeldung (bzw. wir selbst, da wir den Fehler ja in den folgenden Zeilen abfangen). Es reicht übrigens aus, wenn die Datei leer vorhanden ist.

Jedoch kann man den Code verbessern, so dass er auch dann funktioniert, wenn die Datei noch nicht existiert:

```
my $fh = undef;

unless ( -f $dataPath ) {
    $fh = new FileHandle(
        $dataPath, O_RDWR | O_CREAT
    );
}
else {
    $fh = new FileHandle( $dataPath, "r+" );
}
```

Natürlich kann man gleich die erste Variante für das Öffnen der Datei verwenden, bei welcher die Datei angelegt wird, falls sie vorher noch nicht existiert:

```
$fh = new FileHandle ($dataPath, O_RDWR | O_CREAT );
```

Ich wollte Ihnen aber diesen kleinen Unterschied im Konstruktor von »FileHandle« nicht verschweigen.

Der nächste Programmteil, zu dem ein paar Worte angebracht sind, steht in den Zeilen 19 bis 37:

```
19      my @users = ();
20      my $written = undef;
21
22      while ( defined( my $line = <$fh> ) ) {
23          chomp( $line );
24          if ( $line =~ /^([^\t]+)\t[^\t]+\t.$/ ) {
25              if ( $1 eq $login ) {
26                  push( @users, "$login\t$pwd\t$st" );
27                  $written = 1;
28              }
29              else {
30                  push( @users, $line );
31              }
32          }
33      }
34
35      unless ( $written ) {
36          push( @users,  "$login\t$pwd\t$st" );
37      }
```

Es ist schon seltsam, dass man die Datei lesen muss, wenn man doch einen Datensatz hineinschreiben möchte. Einfacher wäre es natürlich, wenn man die Daten ans Ende der Datei anhängen würde. Das verbietet sich aber deshalb, weil die Methode `write()` nicht nur neue Datensätze in die Datei schreiben, sondern auch bestehende Daten ändern soll. Hinge man also die Zeile mit den Daten ans Ende der Datei, dann würde in diesem Fall der Benutzer zweimal vorkommen.

Nun zu der Frage: »Was macht der Code? «

Wir lesen den gesamten Datei-Inhalt Zeile für Zeile und stellen die Zeilen in ein Array. Bei jedem Datensatz überprüfen wir, ob der Benutzername mit dem aktuellen Namen unserer Instanz identisch ist. Falls ja, dann kommt nicht der Datensatz aus der Datei ins Array, sondern unser aktueller Daten-Satz der Instanz.

Wir benötigen ein Flag, das wir in diesem Fall auf `TRUE` setzen, denn es kann ja sein, dass es den Benutzer unserer aktuellen Instanz noch gar nicht gibt. In diesem Fall muss der Datensatz am Ende hinzugefügt werden (Zeilen 35 bis 37).

Mit dem Pattern Matching

```
24          if ( $line =~ /^([^\t]+)\t[^\t]+\t.$/ ) {
```

werden nur solche Zeilen der Datei berücksichtigt, die unserem vorgegebenen Format entsprechen. Falls Sie Probleme beim Verständnis des regulären Ausdrucks haben, empfehle ich Ihnen das Kapitel »Pattern Matching«.

Der nächste Programmteil, der Schwierigkeiten bereiten könnte, ist:

```
39      unless ( seek( $fh, 0, 0 ) ) {
40          $fh->close();
41          return undef;
42      }
43
44      unless ( truncate( $fh, 0 ) ) {
45          $fh->close();
46          return undef;
47      }
```

Nachdem wir den Datei-Inhalt gelesen und gegebenenfalls einen Datensatz verändert haben, müssen wir den Inhalt der Datei neu schreiben. Dazu muss der Dateizeiger zuerst (mit `seek()`) auf den Beginn der Datei gesetzt und anschließend die Datei geleert werden (mit `truncate()`). Der alleinige Aufruf von `truncate()` reicht nicht aus, da er sich ab der aktuellen Position des Dateizeigers auswirkt, die nach dem Lesen ja am Ende der Datei ist.

Um es deutlich zu sagen: In der Praxis wäre diese Implementierung nicht ausreichend. Man müsste eigentlich eine Dateisperre mit Hilfe der Perl-Funktion `flock()` verwenden. Aber ich will das Kapitel OOP nicht zu sehr strapazieren. Im Anhang finden Sie bei der Beschreibung der Perl-Funktion `flock()` auch ein Beispiel.

Weiter geht es mit einer `read()`-Methode, denn wir wollen ja schließlich nicht nur neue Benutzer anlegen oder bestehende Datensätze ändern, sondern auch Benutzerdaten einlesen.

Bevor ein Benutzer eingelesen werden kann, muss das Attribut `login` mit dem Benutzernamen gesetzt werden, dieses Attribut ist also ein Pflichtfeld. Sehen wir uns doch gleich den Programmcode an:

```
01 sub read {
02      my $self = shift( @_ );
03
04      my $login = $self->getLogin();
05      unless ( $login ) {
06          return undef;
07      }
08
09      use FileHandle;
10
11      my $fh = new FileHandle( $dataPath, "r" );
12      unless ( $fh ) {
```

```
13          return undef;
14      }
15
16      my $found = 0;
17
18      while ( defined( my $line = $fh->getline() ) ) {
19          chomp( $line );
20
21          if ( $line =~ /^([^\t]+)\t([^\t]+)\t(.)$/ ) {
22              if ( $1 eq $login ) {
23                  $self->setPwd( $2 );
24                  $self->setStatus( $3 );
25                  last;
26              }
27          }
28      }
29
30      $fh->close();
31
32      return $found;
33  }
```

Bis Zeile 16 sollte alles klar sein. Der Code ist ähnlich wie vorher bei der `write()`-Methode, nur dass wir diesmal die Datei nur zum Lesen öffnen.

In Zeile 16 definieren wir ein Flag, das uns auch als Rückgabewert dient und als Merkmal dafür verwendet wird, ob der gesuchte Datensatz gefunden wurde oder nicht. Hier sei angemerkt, dass die Funktion drei verschiedene Rückgabewerte hat: `undef` bei einem Fehler, `FALSE`, wenn kein Datensatz gefunden wurde, und `TRUE` bei einem Treffer.

Das Lesen aus der Datei in Zeile 18 ist übrigens mit Absicht etwas unterschiedlich kodiert, damit Sie die Variationen von Perl kennen lernen.

Die anschließende Leseschleife ist ähnlich aufgebaut wie bei der `write()`-Methode. Wenn der Benutzername mit dem Suchbegriff im Attribut `login` übereinstimmt, werden die restlichen Attribute der Instanz versorgt und die Schleife beendet.

Jetzt können wir Datensätze der Klasse `User` erzeugen, ändern und lesen. Nun wollen wir das Ganze noch für die abgeleitete Klasse `AnonymousUser` implementieren. Die beiden gerade gezeigten Methoden können wir nicht verwenden, weil sie das in der Klasse `AnonymousUser` hinzugekommene Attribut `email` nicht kennen. Also müssen wir in dieser Klasse zwei neue Methoden `write()` und `read()` entwickeln.

Die gleichnamigen Methoden der abgeleiteten Klasse überschreiben sozusagen die der Eltern-Klasse (Overriding). Wie wir sehen werden, ist der Unterschied nicht so groß.

Zunächst überlegen wir uns wieder ein Format für die Datensätze in der Datei. Damit wir abwärtskompatibel sind, übernehmen wir das bisherige Format und hängen das neue Attribut am Ende an:

```
login\tpwd\tstatus\email
```

Fangen wir mit der Methode `write()` an:

```
01 sub write {
02     my $self = shift( @_ );
03
04     my $login = $self->getLogin();
05     my $pwd = $self->getPwd();
06     my $st = $self->getStatus();
07     my $em = $self->getEmail();
08
09     unless ( $login and $pwd and $em ) {
10         return undef;
11     }
12
13     use FileHandle;
14
15     my $fh = new FileHandle( $dataPath, "r+" );
16     unless ( $fh ) {
17         return undef;
18     }
19
20     my @users = ();
21     my $written = undef;
22
23     while ( defined( my $line = $fh->getline() ) ) {
24         chomp( $line );
25         my $pattern = '^([^\t]+)' . '\t[^\t]+' x 2 .
26                '\t.$';
27         if ( $line =~ /$pattern/ ) {
28             if ( $1 eq $login ) {
29                 push(
30                       @users,
31                       "$login\t$pwd\t$st\tem"
32                   );
33
34                 $written = 1;
35             }
36             else {
37                 push( @users, $line );
38             }
39         }
40     }
41
42     unless ( $written ) {
```

```
43          push( @users,  "$login\t$pwd\t$st\t$em" );
44      }
45
46      unless ( seek( $fh, 0, 0 ) ) {
47          $fh->close();
48          return undef;
49      }
50
51      unless ( truncate( $fh, 0 ) ) {
52          $fh->close();
53          return undef;
54      }
55
56      foreach my $line ( @users ) {
57          print( $fh "$line\n" );
58      }
59
60      $fh->close();
61
62      return 1;
63  }
```

Der gezeigte Code ist bis auf wenige Ausnahmen identisch mit dem in der Klasse `User` gezeigten. Eine Ausnahme ist natürlich die zusätzliche Unterstützung des neuen Attributs.

Aber dennoch verdienen die Zeilen 25 bis 27 besondere Aufmerksamkeit:

```
25          my $pattern = '^([^\t]+)' . '\t[^\t]+' x 2 .
26                  '\t.$';
27          if ( $line =~ /$pattern/ ) {
```

Hier sind zwei Besonderheiten festzustellen: Wir verwenden eine Variable als Searchpattern. Damit nicht genug, außerdem habe ich den Vervielfältigungsoperator für das Pattern benutzt. Der Ausdruck

```
'^([^\t]+)' . '\t[^\t]+' x 2 . '\t.$'
```

ist derselbe wie

```
'^([^\t]+)\t[^\t]+\t[^\t]+\t.$'
```

Weitere Beispiele hierzu gibt es bei der Beschreibung des Vervielfältigungsoperators.

Ich glaube, nun haben wir das Wichtigste von OOP besprochen. Echte Praxisbeispiele finden Sie in den weiteren Kapiteln. Wem das nicht genügt, der kann in der Perl-Online-Dokumentation unter den Themen »perlboot«, »perlobj«, »perltoot« und »perltootc« schmökern.

Einen wichtigen Aspekt von OOP haben wir bis jetzt noch nicht besprochen: Factories.

5.3 Factories

Bisher haben wir nur Klassen aus der Sicht eines Objekts betrachtet (Instanzierung eines Objekts, Attribute eines Objekts, Schreiben und Lesen eines Objekts).

Nun aber wollen wir unseren Horizont erweitern. Häufig benötigt man nicht eine einzelne Instanz einer Klasse, sondern eine Liste von mehreren Objekten. Dies ist zum Beispiel in administrativen Programmen der Fall, wenn ein Verwalter Daten von Benutzern ändern, Benutzer sperren oder löschen, neue Benutzer hinzufügen muss etc.

Solche übergeordneten Funktionalitäten packt man heutzutage gerne in so genannte »Factories«, zu Deutsch »Fabriken«. Hinter diesem Ausdruck verbirgt sich eigentlich nichts anderes als ein ganz normales Package wie `User` oder `AnonymousUser` auch, es ist also nichts Mystisches daran. Das wichtigste Unterscheidungsmerkmal zu normalen Klassenmodulen ist, dass in Factories alle Funktionen definiert sind, die über einer oder mehreren Klassen angesiedelt sind.

Nehmen wir doch gleich ein typisches Beispiel für eine Factory: Ein Administrator möchte den Datensatz eines Benutzers ändern. Dafür benötigt er eine Liste von Benutzern, aus der er einen für die Änderung auswählt. Diese Liste wird üblicherweise in einer Factory-Funktion erzeugt.

Bei der Erzeugung der Liste von Datensätzen kommt in manchen Fällen die Geschwindigkeit vor der Forderung der sauberen Programmierung, die besagt: »Jede Operation für die Klasse User muss auch in dieser Klasse implementiert werden.« Nehmen wir unsere Klasse `AnonymousUser` als Beispiel:

In einem System mit 100.000 Benutzern kann es ziemlich lange dauern, bis man eine Liste von instanzierten Objekten mit allen Attributen zusammengestellt hat. Doch was braucht man in der Liste wirklich? Meist doch nur den Benutzernamen, der anschließend ausgewählt wird. Es bedeutet also, mit Kanonen auf Spatzen zu schießen, wenn wir 100.000 Instanzen mit allen Attributen im Hauptspeicher erzeugen, obwohl das Speichern des Benutzernamens völlig ausreichend ist.

Wenn wir zu den Themen »CGI« und »DBI« kommen, werden Sie sehen, dass man nicht einmal einen Namen, sondern nur die Datenbank-ID in der Liste braucht.

Vor allem Hochschulabgänger der heutigen Generation unterliegen immer wieder dem Trugschluss, dass Interpreter bzw. Compiler einem die ganze Arbeit abnehmen und man so tun könnte, als hätte man einen unendlich großen Hauptspeicher und eine unendlich schnelle CPU. Mit ein wenig Hirnzellengymnastik aber kann man Applikationen erstellen, die zum einen so wenig Hauptspeicher wie möglich verwenden, zum anderen so schnell wie möglich ablaufen. Glauben Sie mir, ich weiß es aus Erfahrung: 1 GB Hauptspeicher ist schnell gefüllt, wenn man seinen Verstand nicht richtig einsetzt.

Ich habe schon Entwickler gesehen, die drei Wochen lang Fehlersuche in ihrem Programm betrieben haben, das eine unbegrenzte Anzahl von Threads gestartet hat. Der Grund, warum das Programm auf Rechner A lief, auf Rechner B aber nicht, lag einfach daran, dass auf Rechner B die Anzahl der geöffneten FileHandles begrenzt war.

Ich will Sie damit nicht auf eine Zeitreise ins Jahr 1985 schicken, als der Hauptspeicher noch in KBytes und die Festplattenkapazität in MBytes angegeben wurde. Aber gesunder Menschenverstand ist nie fehl am Platz!

Nun zurück zu unserem eigentlichen Thema, den Factories.

Lassen Sie uns eine (statische) Funktion implementieren, mit der man eine Liste aller Benutzer erhält. Wir wollen sie `readUserLogins()` nennen und im Modul *UserFactory.pm* abspeichern:

```
sub readUserLogins {
    my @logins = ();

    use FileHandle;

    my $fh = new FileHandle( $dataPath, "r" );
    unless ( $fh ) {
        return( undef, @logins );
    }

    while ( defined( my $line = $fh->getline() ) ) {
        if ( $line =~ /^([^\t]+)/ ) {
            push( @logins, $1 );
        }
    }

    $fh->close();

    return ( 1, @logins );
}
```

Die Funktion benutzt, wie die vorherigen auch, die globale Variable `$dataPath`, in welcher der Pfad der Datendatei gespeichert ist.

Von jedem Datensatz wird nur der Benutzername gelesen, die anderen Attribute sind uninteressant. Es soll ja nur eine Liste aller Benutzer zurückgegeben werden. Die Sache wäre etwas anders, wenn man Filter einsetzt, wie zum Beispiel: »Gib mir eine Liste aller Benutzer, die gesperrt sind.« In diesem Fall müsste man zusätzlich das Attribut `status` lesen.

Die zurückgegebene Liste der Benutzer enthält also nur die Werte des Attributs `login`. Das sieht sehr effizient aus. Ist es auch, muss ich hinzufügen. Kurz und prägnant.

Man muss sich immer vor Augen halten, was mit dieser Liste passiert. In den meisten Fällen wird vom Anwender der Applikation ein einziges Element der Liste ausgewählt, und erst dann ist es notwendig, alle Attribute des selektierten Benutzers zu lesen. Für die Ausgabe der Liste reicht der Benutzername.

Ich möchte Ihnen als Kontrast noch die (etwas langsamere) Lang-Version der Funktion zeigen, bei der jeder einzelne Datensatz als komplette Klasseninstanz angelegt wird (und dementsprechend viel CPU-Zeit und Hauptspeicher in Anspruch nimmt):

```
sub readUsers {
    my @users = ();

    use FileHandle;

    my $fh = new FileHandle( $dataPath, "r" );
    unless ( $fh ) {
        return( undef );
    }

    while ( defined( my $line = $fh->getline() ) ) {
        if ( $line =~ /^([^\t]+)\t([^\t]+)(.)$/ ) {
            my $user = new User();
            $user->setLogin( $1 );
            $user->setPwd( $2 );
            $user->setStatus( $3 );
            push( @logins, $user );
        }
    }

    $fh->close();

    return ( 1, @users );
}
```

Übrigens, hier sehen wir ein Beispiel, wie man in einer Funktion mehrere Rückgabewerte an den Aufrufer zurückliefert. Der erste Wert ist der Rückgabestatus, der zweite beinhaltet die Liste der Benutzer. Im Falle eines Fehlers ist der Rückgabestatus `undef`, und es ist in diesem Fall auch das einzige Argument, das zurückgegeben wird. Denken Sie bitte daran, dass die Reihenfolge der Rückgabewerte nicht vertauschbar ist. Wenn wir nämlich mit

```
return ( @users, 1 );
```

zuerst das Array und am Schluss den Status zurückgeben, dann haben wir ein Problem, weil für den Aufrufer der Funktion der Status als letztes Element der Liste aufgenommen wird und kein eigenständiger Wert mehr ist. Der Aufruf

```
my ( @users, $status ) = UserFactory::readUsers();
```

würde die Variable `$status` nicht versorgen, sie wäre `undef`.

Es soll übrigens Entwickler geben, die in der Schleife Folgendes programmieren:

```
while ( defined( my $line = $fh->getline() ) ) {
    if ( $line =~ /^([^\t]+)\t([^\t]+)(.)$/ ) {
        my $user = new User();
        $user->setLogin( $1 );
        $user->read();
        push( @logins, $user );
    }
}
```

Der Code wäre tödlich, weil beim Aufruf

```
$user->read();
```

die Datei noch einmal geöffnet und gelesen wird, ohne dass man es direkt sieht.