

# **Objektorientierte Programmierung**

**Theorie – sprachenspezifische Konzeption – Anwendung**

---

Fachbereichsarbeit im Wahlpflichtfach Informatik

**Doron Edlinger**

Betreuung:

**Mag. Carl Metnitz**

Schuljahr 2005/2006

# 1 Inhalt

1	Inhalt .....	b
2	Einleitung .....	1
3	Theoretische Grundlagen.....	2
3.1	Was ist ein Objekt? Was ist eine Klasse? .....	2
3.1.1	Besondere Klassen.....	3
3.2	Eigenschaften und Methoden.....	4
3.2.1	Eigenschaften .....	4
3.2.2	Methoden.....	4
3.3	Klassenbeziehungen .....	6
3.3.1	Vererbung .....	7
3.3.2	Aggregation.....	8
3.3.3	Assoziation.....	8
3.3.4	Using-Beziehungen.....	8
4	Sprachenspezifische Konzeption .....	10
4.1	Java.....	10
4.1.1	Statische Eigenschaften und Methoden.....	13
4.1.2	Kapselung.....	14
4.2	Object Pascal .....	14
4.2.1	Deklaration einer Klasse .....	15
4.2.2	Vererbung .....	17
4.2.3	Typecasting.....	17
4.3	Perl.....	18
4.3.1	Referenzen .....	18
4.3.2	Module und Packages .....	20
4.3.3	Klassendefinition.....	20
4.3.4	Verwendung von Objekten.....	22
4.3.5	Vererbung .....	22
5	Anwendung am Beispiel eines Online-Shops .....	24
5.1	Verwendung einer Datenbank .....	24
5.2	Die shop-Klasse .....	26
6	Schlussbemerkungen .....	28
7	Anhang.....	i
7.1	Literaturverzeichnis .....	i
7.2	Kurzübersicht über die verwendeten Programmiersprachen.....	ii
7.2.1	Java .....	ii
7.2.2	Delphi/Object Pascal.....	ii
7.2.3	Perl .....	iii
7.3	Danksagung .....	v

## 2 Einleitung

In den 1960er Jahren stieg durch die komplexeren Anforderungen an die Software das Verlangen nach einem den neuen Aufgaben gewachsenen Programmierkonzept. Die bisherigen Programmiersprachen wie *Fortran* und *LISP* kannten gerade erst Funktionen, also die Einteilung in mehrfach verwendete Unterprogramme; ein System, das zwar eine grobe Strukturierung erlaubt, den steigenden Ansprüchen aber nicht entsprach.

Das neue System sollte vor allem die Realität widerspiegeln können. Sieht man die Welt als Sammlung unterschiedlicher Objekte, die teils gleichen Ursprungs sind, wird der Grundgedanke der *objektorientierten Programmierung*, abgekürzt als *OOP*, ersichtlich. Häufig wird dazu Platons Ideenlehre zitiert, die besagt, dass es eine Welt der Ideen und eine Welt deren Abbilder gibt.

Was diese Begriffe umgelegt auf die Programmierung bedeuten, ist Thema des ersten Teils dieser Arbeit. Allgemein wird auch darauf eingegangen, wie man mit Objekten umgeht und wie man sie zueinander in Beziehung setzt. Der zweite Teil zeigt, wie diese Prinzipien in völlig unterschiedlichen Sprachen umgesetzt werden. Im dritten Teil wird anhand des Beispiels eines Online-Shops erläutert, wie vorgegangen wird, um ein objektorientiertes Programm<sup>1</sup> sinnvoll zu planen und zu programmieren.

In fast allen modernen Programmiersprachen der 4. Generation wurde die objektorientierte Programmierung eingeführt; die erste objektorientierte Sprache war jedoch *Smalltalk*, das 1980 veröffentlicht<sup>2</sup> wurde. *Smalltalk* beeinflusste nicht nur die Programmierung nachhaltig, sondern fand dank der ausgeklügelten Entwicklungsumgebung auch Niederschlag in den grafischen Benutzeroberflächen der modernen benutzerorientierten Betriebssysteme<sup>3</sup>.

---

<sup>1</sup> Da die Skriptsprache Perl verwendet wurde, handelt es sich eigentlich um ein objektbasiertes Skript.

<sup>2</sup> ihr Vorgänger war die Sprache *Simula* von 1960, vgl. Wikipedia, „Simula“

<sup>3</sup> vgl. Wikipedia, „Smalltalk (Programmiersprache)“

## 3 Theoretische Grundlagen

### 3.1 Was ist ein Objekt? Was ist eine Klasse?

Betrachtet man ein Objekt in der Natur, erkennt man, dass es sich durch zwei grundlegende Dinge auszeichnet, einerseits durch seine äußeren und inneren Eigenschaften, die den Zustand des Objekts widerspiegeln, andererseits durch die Funktion, die es erfüllt. Als Beispiel sei das Objekt Auto genannt<sup>4</sup>: neben den äußeren Eigenschaften, wie Form, Farbe und derzeitige Geschwindigkeit, und den inneren, wie die maximal erreichbare Geschwindigkeit, kann es verschiedene Aktionen ausführen, von denen die offensichtlichsten Beschleunigen und Bremsen sind. Ein technischerer Ansatz zur Beschreibung von Objekten besagt, „*ein Objekt beinhaltet Datenfelder und Funktionen, die auf diese Datenfelder zugreifen können*“<sup>5</sup>.

Betrachtet man nun eine Reihe von Autos eines Modells, wird man erkennen, dass sich zwar die einzelnen Autos voneinander unterscheiden, im Grunde aber nach einem Bauplan konstruiert wurden.

Bei der objektorientierten Programmierung verhält es sich nicht anders. Sieht man den einfachsten Variablentypen, eine *Boolean-Variable*, die ein Bit, also den Zustand 0 oder 1 repräsentiert, als Objekt<sup>6</sup>, ist dieses entsprechend simpel, denn es besitzt eine einzige *Eigenschaft* und minimalen Funktionsumfang (beispielsweise könnte man eine Funktion, genauer gesagt eine *Methode*, definieren, die den Zustand umkehrt). Das erzeugte Objekt ist eine *Instanz* eines Bauplans, der als *Klasse* bezeichnet wird. Zu anderen Objekten der Klasse unterscheidet es sich nicht nur durch einen möglicherweise anderen Zustand, sondern auch durch eine eindeutige *Identität*. Zur Veranschaulichung sei der folgende Code-Ausschnitt einer Variablendefinition in der Programmiersprache Pascal gegeben:

```
var
  b : Boolean;
  i : Integer;
```

---

<sup>4</sup> vgl. Schili, 1998, S. 153ff

<sup>5</sup> Erlenkötter, Reher, 1997, S. 43

<sup>6</sup> Die in der Praxis verwendeten Objekte sind natürlich komplexer, lassen sich aber letztendlich durch wenige grundlegende Objekte, wie beispielsweise ein String-Objekt oder ein Integer-Objekt, zusammensetzen.

Übertragen auf die objektorientierte Programmierung würde man jeweils die linke Seite als die Identität der Instanz/des Objekts bezeichnen und die rechte als Klasse. In einer konsequent umgesetzten objektorientierten Programmiersprache sind alle grundlegenden Variablentypen als Objekte zu betrachten.

Der Begriff „Objekt“ wirkt so gewissermaßen abstrakt, letztendlich trifft man aber beim Gebrauch von *GUIs*<sup>7</sup> täglich auf Objekte – so sind Fenster an sich und alle darin befindlichen Elemente, wie Schaltflächen, Menüs, Textfelder, programmiertechnisch gesehen Objekte.

### 3.1.1 Besondere Klassen

#### 3.1.1.1 Container-Klassen

Wie im vorigen Kapitel angesprochen, müssen auch *Arrays* als Objekt angesehen werden. Im herkömmlichen Sinn versteht man darunter eine Liste von *Skalaren*<sup>8</sup> (oder weiteren *Arrays*), auf die durch einen *Index* zugegriffen werden kann. In der objektorientierten Programmierung sind jedoch Skalare selbst Objekte. Ein Objekt, das zur Vereinigung und Verwaltung einer Menge von Objekten dient, bezeichnet man als *Container-Objekt*, und deren Bauplan als *Container-Klasse*.

#### 3.1.1.2 Basisklassen

Als *Basisklasse* oder *abstrakte Klasse* bezeichnet man eine Klasse, auf deren Grundlage keine Instanzen erzeugt werden (können). Das Analogon in der Natur ist beispielsweise der Begriff „Fahrzeug“. Fahrzeug selbst ist ein allgemeiner Begriff, auf ein Fahrzeug an sich trifft man in der Natur nicht, sondern auf Unterarten von Fahrzeugen. Darauf wird im Kapitel Vererbung näher eingegangen.

---

<sup>7</sup> *Graphical User Interfaces*, grafische Benutzeroberflächen

<sup>8</sup> als *Skalar* bezeichnet man einen allgemeinen Variablentypen, der einen einfachen Wert wie einen Text (*String*) oder eine Zahl (*Integer* oder *Float*) repräsentiert.

## 3.2 Eigenschaften und Methoden

### 3.2.1 Eigenschaften

Als *Eigenschaften* bezeichnet man, wie oben erwähnt, eine Menge von Variablen bzw. Objekten, die den inneren Zustand des Objektes wiedergeben. Welche Eigenschaften welchen Typs ein Objekt hat, wird bei der Definition der Klasse festgelegt.

Eine wichtige Eigenheit von Objekten ist die *Kapselung*. Dies bedeutet, dass der Zustand des Objekts nicht von außen manipuliert werden darf, es sei denn, dass dafür von der Klasse Möglichkeiten oder Schnittstellen zur Verfügung gestellt werden<sup>9</sup>. Dieses Konzept wird meistens nicht vollständig umgesetzt, oder es wird dem/der Programmierer/in überlassen, inwieweit dieser Schutzmechanismus verwendet wird<sup>10</sup> - letztendlich wird damit der Programmierer/die Programmiererin dazu gezwungen, einen sauberen Programmierstil zu verwenden, was diese/r aufgrund der Erfahrung ohnehin tun wird, denn der Vorteil der Kapselung liegt auf der Hand: Bei der Verwendung der Klasse muss nicht auf deren Implementierung Rücksicht genommen werden, da der Zugriff ausschließlich über Schnittstellen erfolgt. Möchte man mit einem Auto fahren, ist es unerheblich, *wodurch* es fährt, für den Benutzer/die Benutzerin ist nur wichtig, *dass* es fährt, wenn es durch die Schnittstellen Gaspedal, Kupplung, Bremse oder Lenkrad angesteuert wird.

### 3.2.2 Methoden

*Methoden* sind Funktionen und Prozeduren ähnlich und werden ebenfalls in der Klassendefinition festgelegt.

#### 3.2.2.1 Parameter

Wie beim gewöhnlichen Aufruf einer Funktion können, sofern dies in der *Definition* der Funktion festgelegt wurde, *Parameter* übergeben werden, auf die im Ablauf der Funktion als Variablen zugegriffen werden kann. Parameter können in drei Formen übergeben werden:

- als beliebig manipulierbare Kopie der übergebenen Wertes (x),

---

<sup>9</sup> Vgl. Kapitel Methoden auf Seite 4

<sup>10</sup> Vgl. Kapitel Sprachenspezifische Konzeption auf Seite 10

- als Verweis auf die Originalvariable (y),
- als Verweis auf die Originalvariable, ohne diese verändern zu dürfen (z).

Nachfolgender Programmcode soll dies verdeutlichen<sup>11</sup>.

```
program Parameterbeispiel ;

uses
    SysUtils;

//          Parameterdefinition
procedure Test(x: Integer; var y: Integer; const z: Integer);
begin
    x := x * 10;
    y := y * 10;
    // Dies würde einen Fehler auslösen:
    // z := z * 10

    // Gibt 330 aus
    WriteLn(IntToStr(x + y + z));
end;

// Programmablauf
var
    a, b, c : Integer;
begin
    a := 10;
    b := 20;
    c := 30;

    // Gibt 60 aus
    WriteLn(IntToStr(a + b + c));
    // Funktionsaufruf von Testfunktion mit
    // x = a, y = b und z = c
    Testfunktion(a, b, c);
    // Gibt 240 aus.
    WriteLn(IntToStr(a + b + c));

end.
```

---

<sup>11</sup> Eine Erklärung der verwendeten Funktionen und Operatoren findet sich im Anhang auf Seite ii.

Im Geltungsbereich einer Methode kann zusätzlich auf die Eigenschaften des Objekts zugegriffen und deren Manipulation vorgenommen werden.

### 3.2.2.2 Methodenarten

Man unterscheidet mehrere Methodenarten:

- **Konstruktoren**

Mit einem *Konstruktor* wird eine Instanz erschaffen. Das bedeutet u.a., dass Speicher reserviert wird und grundlegende Eigenschaften, oft auch durch Übergabe von Parametern, festgelegt werden.

- **Destruktoren**

Ein *Destruktor* ist je nach Programmiersprache und Klasse nicht zwingend notwendig. Bei einer Klasse, die eine Datei repräsentiert, könnte damit etwa das Schreiben der Daten auf die Festplatte übernommen werden.

- **Mutatoren**

Mit einem *Mutator* wird der Zustand des Objekts verändert. Das auf Seite 2 genannte Beispiel zur Umkehrung des Zustandes des Boolean-Objekts ist ein Mutator.

- **Accessoren**

Ein *Accessor* bezeichnet eine Schnittstelle zur Abfrage des Zustandes oder einer Eigenschaft eines Objekts. Sofern das Konzept der Kapselung vollständig umgesetzt wurde, werden meist bestimmte Werte in der Klassendefinition explizit zur Abfrage freigeben, wodurch das Programmieren von Accessoren unnötig wird, aber dennoch als sauberer gilt, da so die Implementierung des Objekts geändert werden kann, ohne die Implementierung des eigentlichen Programms zu beeinflussen.

- **Funktionen**

*Funktionen* führen Berechnungen aufgrund des Zustandes des Objekts durch, ohne diesen zu verändern.

## 3.3 Klassenbeziehungen

Bei der Verwendung der OOP sind *Klassenbeziehungen* etwa so alltäglich wie Operatoren. Dennoch werden sie, abgesehen von der Vererbung, meist selbstverständ-

lich verwendet, ohne sich darüber Gedanken zu machen, um welche Klassenbeziehungsart es sich handelt. Man unterscheidet zwischen vier Arten.

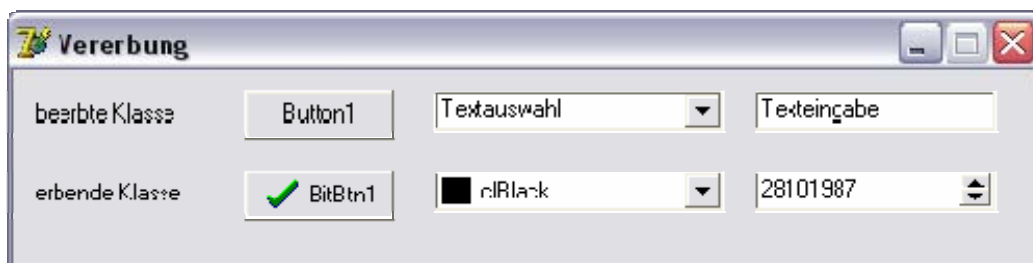
### 3.3.1 Vererbung

Ziel der *Vererbung* ist, ähnliche Klassen zu vereinheitlichen, um damit einerseits eine einheitlichere Funktionsweise und damit sauberere Struktur zu erreichen, andererseits Programmieraufwand einzusparen.

Um das Beispiel des Autos wieder aufzunehmen: Ein Auto lässt sich auf den Begriff Fahrzeug reduzieren, umgekehrt gibt es mehrere Unterarten von Autos, etwa Kleinwagen oder Limousinen. Eine Eigenschaft *Mini bar* hätte bei der Klasse *Kleinwagen* wenig Sinn, jedoch unterstützen sowohl die Klasse *Limousine* als auch die Klasse *Kleinwagen* die Methoden *Vorderlicht\_an*, die wiederum die Klasse *Scooter* nicht unterstützt. Allen drei Klassen sind jedoch die Methoden *Beschleunigen* und *Bremsen* sowie die Eigenschaft *Geschwindigkeit* gemein. Bei der Vererbung werden also gemeinsame Eigenschaften und Methoden von einer Elternklasse, die, wie in Kapitel Basisklassen auf Seite 3 erwähnt, auch eine Basisklasse sein kann, übernommen.

Geerbte Methoden können bei Bedarf auch überschrieben werden. Stammt beispielsweise von einer Klasse *Kleinwagen* die Klasse *WasserstoffKleinwagen* ab, so müsste die bisher auf Benzin oder Diesel basierende Methode *Antrieb* ergänzt oder durch eine neue ersetzt werden.

Das nachfolgende Beispiel zeigt drei Standardkomponenten, und deren mögliche „Nachfahren“. In der Regel<sup>12</sup> beruhen jedoch beide gezeigten Komponenten auf einer gemeinsamen Basisklasse, die sich nicht darstellen lässt.



*Mehrfachvererbung* bedeutet, dass eine Klasse von mehreren Klassen erbt und diese in einer Synthese vereint. Zur Veranschaulichung sei die Klasse *Amphibi enfahr-*

---

<sup>12</sup> Im hier gezeigten Beispiel stammt nur die mit einer Grafik versehene Schaltfläche tatsächlich von der reinen Textschaltfläche ab, die Verwandtschaft der anderen Komponenten ist dennoch ersichtlich.

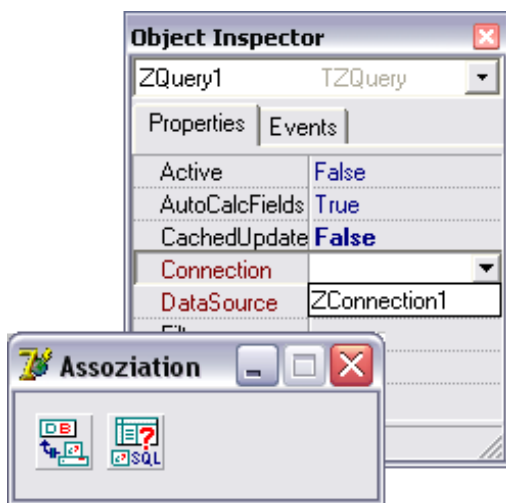
zeug genannt: diese erbt sowohl von der Klasse `Auto`, als auch von der Klasse `Boot`. Diese Möglichkeit wird von wenigen Programmiersprachen unterstützt.

### 3.3.2 Aggregation

*Aggregation* bedeutet, dass ein Objekt ein anderes Objekt „besitzt“, das zweite Objekt ein Teil – eine Eigenschaft - des ersten ist. Im abgebildeten Beispiel ist eine Schaltfläche Teil eines Formulars.



### 3.3.3 Assoziation



Die *Assoziation* ist der *Aggregation* sehr ähnlich und wird in manchen Programmiersprachen, wie beispielsweise Perl, im Quellcode nicht ersichtlich. Der Unterschied liegt darin, dass bei der *Assoziation* zwei Objekte nebeneinander als eigenständige Objekte existieren und nicht das eine Objekt Eigenschaft des anderen ist, sondern das erste Objekt vom zweiten Objekt „weiß“. Eine vergleichbare Erscheinung in der Natur ist ein Auto-Objekt, welches in einem Garage-Objekt steht.

Das Beispiel zeigt zwei Objekte: das linke hat die Identität `ZConnection1` und stellt eine Datenbankverbindung her, während das rechte, `ZQuery1`, zum Abfragen einer Datenbank mittels *SQL*<sup>13</sup> dient. Zu diesem Zweck muss zugewiesen werden, über welche Verbindung die Abfrage getätigt werden kann – dies wird in *Borland Delphi 7* im *Object Inspector* eingestellt. Wären mehrere Datenbankabfragen verwendet worden, würden im Regelfall alle dieselbe Verbindung nutzen, ohne, dass von ihrem Objekt eine Kopie erstellt werden würde, wie das bei einer *Aggregation* der Fall wäre.

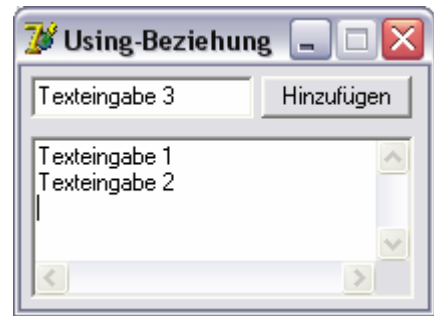
### 3.3.4 Using-Beziehungen

In einer *Using-Beziehung* manipuliert ein Objekt den Zustand eines anderen Objekts, dessen Identität als Parameter übergeben wird. So könnte durch den Aufruf der Methode `Steuern` eines Fahrer-Objekts je nach dem eigenen Zustand entweder die Me-

<sup>13</sup> *Structured Query Language*, einfache Sprache zur Datenabfrage von diversen Datenbanksystemen.

thode Beschleunigen oder die Methode Bremsen eines Auto-Objekts aufgerufen werden und somit die Geschwindigkeit verändert werden.

Das abgebildete, im Grunde alltägliche Beispiel zeigt eine Schaltfläche, über eine deren Methoden der Zustand - in diesem Fall der Text - eines Textfeldes verändert wird.



## 4 Sprachenspezifische Konzeption

Die Umsetzung der objektorientierten Programmierung in den unterschiedlichen Programmiersprachen variiert beträchtlich: zum einen gibt es Sprachen, die sich konsequent an das Konzept halten und zu dessen Verwendung zwingen – ein klassisches Beispiel dafür ist die Programmiersprache *Java*. Weiters gibt es Sprachen, die dem Entwickler/der Entwicklerin die Möglichkeit geben, sich an die objektorientierte Programmierung zu halten, dies aber nicht verlangen. Dieser Gruppe gehören die meisten Sprachen, darunter auch *Object Pascal*, an. Eine besondere Rolle spielen jene Sprachen, die ursprünglich keine Möglichkeit zur objektorientierten Programmierung boten, diese aber nachträglich integrierten; dazu zählt etwa die Skriptsprache<sup>14</sup> *Perl*, in dessen Syntax gut ersichtlich ist, wie einfach, aber dennoch fortschrittlich die objektorientierte Programmierung ist. Diese Sprachen lassen meist ebenfalls die Möglichkeit zur Wahl des Programmierkonzepts, sollen aber im Folgenden gesondert behandelt werden.

Allgemein ist zu sagen, dass die Objekte, wie im theoretischen Teil auf Seite 2 erwähnt, durch eine Identität repräsentiert werden, im Programmcode also so aussehen, wie gewöhnliche Variablen. Um auf die Methoden und Eigenschaften eines Objekts zuzugreifen, schreibt man den Namen des Objekts gefolgt von einem Punkt (.) oder – je nach Programmiersprache - einem anderen Trennzeichen und dem Namen der Methode oder der Eigenschaft.

### 4.1 Java

Wie zuvor angesprochen, wurde in Java das Prinzip der Objektorientierung fast<sup>15</sup> durchgängig umgesetzt. So ist jedes Programm an sich eine Klasse, die über eine Methode namens `main` verfügen muss, die den Programmhauptablauf bestimmt. Die Klassendefinition wird durch das Schlüsselwort `class` eingeleitet und von der Klassenbezeichnung und einem Anweisungsblock, der in `{` und `}` eingeschlossen wird, gefolgt.

```
class Testprogramm {
```

---

<sup>14</sup> *Scriptsprachen* sind jene Sprachen, die durch einen *Interpreter* ausgeführt werden und nicht zu selbstständig lauffähigen Dateien kompiliert werden. Dies ermöglicht i.d.R. Portierbarkeit unter den unterschiedlichen Betriebssystemen.

<sup>15</sup> Die Grunddatentypen – Boolean/Integer/Float - sind in Java keine Klassen.

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

Jede weitere programmierte Funktion ist somit eine Methode. Die Stichwörter `public`, `static` und `void` bedeuten, dass die Methode generell zugänglich ist, sie zur Klasse und nicht zu einer Instanz gehört<sup>16</sup> und keinen Rückgabewert besitzt. In der Klammer wird, wie bei dem Beispiel auf Seite 4, festgelegt, welche Parameter übergeben werden müssen<sup>17</sup> - der Methode `main` werden jene Parameter übergeben, die beim Kommandozeilenaufruf angegeben werden.

Die Deklaration von Eigenschaften erfolgt im Hauptblock – also zwischen der ersten und der letzten geschwungenen Klammer - der Klasse. Dabei wird zunächst angegeben, um welchen Variablentypen oder welche Klasse es sich handelt. Anschließend folgt eine durch Kommata (,) getrennte Liste der zu deklarierenden Eigenschaften. Getrennt durch ein Ist-gleich-Zeichen (=) kann ein Startwert zugewiesen werden. Der Zugriff innerhalb von Methoden erfolgt wie der Zugriff auf gewöhnliche Variablen.

```
class Beispielklasse {  
    int Eigenschaft1 = 0, Eigenschaft2 = 1;  
    double Mittelw;  
  
    public void BerechneMittelwert() {  
        Mittelw = (Eigenschaft1 + Eigenschaft2) / 2;  
    }  
}
```

Die Vererbung erfolgt durch das Stichwort `extends` nach der Klassendeklaration. Eine Mehrfachvererbung<sup>18</sup> ist dabei nicht möglich, um Namenskonflikte zu vermeiden<sup>19</sup>. Methoden der Elternklasse können überschrieben werden, indem in der erbenden Klasse eine namensgleiche Methode mit derselben Parameterdefinition<sup>20</sup> deklariert wird. Alle Klassen stammen von der Basisklasse `Object` ab, die über Methoden zur Speicherverwaltung verfügt<sup>21</sup>.

---

<sup>16</sup> Vgl. Abschnitt Statische Eigenschaften und Methoden auf Seite 13

<sup>17</sup> Zur Variablendeklaration vgl. Anhang auf Seite ii

<sup>18</sup> Vgl. Abschnitt Vererbung auf Seite 7

<sup>19</sup> Vgl. Erlenkötter, Reher, 1997, S. 86

<sup>20</sup> Java erlaubt die Definition mehrerer Methoden mit der selben Bezeichnung, sofern diese durch die benötigten Parameter unterschieden werden. Vgl. ebenda, S. 91

<sup>21</sup> Vgl. ebenda, S. 103

```

class erbendeKlasse extends Elternklasse {
    // ...
}

```

Sollte ein Konstruktor zur Initialisierung der Eigenschaften notwendig sein, wird dieser als Methode ohne Rückgabewert<sup>22</sup> deklariert, welche den gleichen Namen wie die Klasse trägt. Wie bei einer gewöhnlichen Methode können Parameter übergeben werden.

```

class Beispielklasse {
    Klasse() {
        // ...
    }
}

```

Die Instanzierung<sup>23</sup> einer Klasse erfolgt in Form von Adressverweisen, so genannten Referenzen, auf ein von Java intern gespeichertes Objekt.

```

int a;
Klassenbezeichnung Objektbezeichnung;
a = 1;
Objektbezeichnung = new Klassenbezeichnung;

```

Zunächst wurde eine Variable vom Typ *Referenz auf Objekt des Typs* Klassenbezeichnung erstellt (analog dazu ist im Beispiel die Deklaration einer Integer-Variable dargestellt). Dieser Variable wird anhand des Stichwortes `new` eine Referenz auf ein neu angelegtes Objekt der Klasse Klassenbezeichnung zugewiesen. Der Zugriff auf Methoden und Eigenschaften erfolgt dann folgendermaßen:

```

Objekt.Eigenschaft1 = "Eigenschaftswert";
Objekt.Methode1("Parameter1");

```

Zur Vermeidung von Namenskonflikten stellt Java die Möglichkeit zur Deklaration von *Packages* zur Verfügung, die „eine Sammlung von Klassen“<sup>24</sup> darstellen. Diese können über das Schlüsselwort `import` in ein anderes Paket eingebunden werden.

```

package testpackage.testklassen;

class test1 {
    // ...
}

```

<sup>22</sup> Das Schlüsselwort `void` ist hierbei nicht notwendig.

<sup>23</sup> Instanzieren bedeutet, eine Instanz (ein Objekt) einer Klasse zu bilden.

<sup>24</sup> Erlenkötter, Reher, 1997, S. 101.

```
class test2 {
    // ...
}
```

```
// Einbinden einer Klasse
import testpackage.testklassen.test1;
// Einbinden aller Klassen
import testpackage.testklassen.*;
```

#### 4.1.1 Statische Eigenschaften und Methoden

Statische<sup>25</sup> Eigenschaften sind Eigenschaften, die innerhalb einer Klasse nur ein Mal vorliegen und sich nicht von Objekt zu Objekt unterscheiden. Diese Besonderheit wird durch das Schlüsselwort `static` vor der Deklaration bestimmt. Wird diese Eigenschaft durch ein Objekt manipuliert, betrifft dies folglich alle Objekte dieser Klasse, wie das folgende Beispiel<sup>26</sup> zeigt. Hierbei wird jedem erzeugten Objekt eine eindeutige Identifikationsnummer zugeteilt, die automatisch im Konstruktor erhöht wird.

```
class Nummernklasse {
    public int id;
    static protected27 naechste_id = 1;

    Nummernklasse {
        id = naechste_id;
        naechste_id++; // um 1 erhöhen
    }
}
```

Analog zu statischen Eigenschaften können statische Methoden deklariert werden, anhand derer die statischen Eigenschaften manipuliert werden können. Ihr Aufruf erfolgt nicht anhand der Identität des Objekts, sondern der Klasse:

```
class Nummernklasse2 extends Nummernklasse {
    static void Nummer_ueberspringen {
        naechste_id++;
    }
}
```

<sup>25</sup> Der Begriff statisch ist hier nicht als Synonym von konstant zu sehen.

<sup>26</sup> Vgl. ebenda, S. 91

<sup>27</sup> Das Schlüsselwort `protected` wird im Abschnitt Kapselung auf Seite 14 erklärt.

```

class Nummerntest {
    public void NeuesNummernobjekt {
        Nummernklasse2. Nummer_ueberspringen();
        Nummernklasse2 Nummer = new Nummernklasse2;

        // Das Objekt Nummer hat nun die id 2
    }
}

```

Statische Methoden kommen insbesondere bei so genannten *Hüllenklassen* zum Einsatz. Hüllenklassen sind von Java zur Verfügung gestellte Klassen, welche Methoden zur Manipulation und Umwandlung der Grunddatentypen, die, wie auf Seite 10 beschrieben, keine Objekte sind, besitzen. Ihre Bezeichnung ist jeweils die ihres Grunddatentyps mit dem Unterschied, dass das erste Zeichen groß geschrieben wird. Nachfolgendes Beispiel zeigt die Umwandlung eines Boolean-Wertes zu einem String anhand der Hüllenklasse `String`.

```

boolean Testbool = true;
System.out.println(String.valueOf(Testbool));

```

#### 4.1.2 Kapselung

Java unterscheidet zwischen drei Möglichkeiten zur Kapselung für Methoden und Eigenschaften:

<code>public</code>	Für andere Klassen ist der Zugriff erlaubt
<code>protected</code>	Für erbende Klassen ist der Zugriff erlaubt
<code>private</code>	Der Zugriff ist nur innerhalb der Klasse erlaubt

#### 4.2 Object Pascal

Ähnlich wie in Java, sind die in der Entwicklungsumgebung Delphi erzeugten Fenster, in denen der Programmablauf stattfindet, Objekte. Durch einen Hauptprogrammteil werden diese Objekte erzeugt. Die Klassen werden in der Regel in eigenen Units, die als Dateien mit Programmcode zu verstehen sind, abgespeichert. Eine solche Unit ist wie folgt aufgebaut:

```

unit Uni tbezeichnung;

interface

```

```
// eventuelle Einbindung anderer Units:  
uses  
    Uni t1, Uni t2;  
  
// Deklaration der Klassen, Variablen und Funktionen  
  
implementation  
  
// Ausprogrammierung der Methoden, und Funktionen  
  
end.
```

Auffallend ist, dass die Deklaration der Klassen getrennt von dem eigentlichen Programmcode wurde. Dies bewirkt ein hohes Maß an Übersichtlichkeit. Weiters können in einer Unit mehrere Klassen mit „gewöhnlichen“ Variablen und Funktionen gemischt werden. Auf diese kann innerhalb der Methoden der Klassen zugegriffen werden.

#### 4.2.1 Deklaration einer Klasse

Die Deklaration einer Klasse steht zwischen den Schlüsselwörtern `type` und `end`. Die Bezeichnung der Klasse steht am Anfang und beginnt in der Regel mit einem führenden `T` zur besseren Unterscheidung zwischen Klassen und Objekten. Ihr folgend steht das Schlüsselwort `class`, dem in Klammern die Bezeichnung der zu beerbenden Klasse folgen kann (ist keine angegeben, erbt die Klasse, wie in Java, von der Klasse `TObject`). Das Schlüsselwort `class` ist auf Grund der Tatsache notwendig, dass auch die Möglichkeit zur Deklaration von so genannten *Records*, die gewissermaßen Klassen ohne Methoden darstellen, besteht. Anschließend werden die Methoden und Eigenschaften, gruppiert nach Kapselungsmodus, festgelegt.

Die Kapselungsmodi sind identisch mit jenen, die in Java verwendet werden<sup>28</sup>. Jedoch gibt es zusätzlich den Modus `published`, der grundsätzlich das gleiche bewirkt, wie der Modus `public`, jedoch „werden diese Eigenschaften im Objektinspektor angezeigt und können somit zur Entwurfszeit gesetzt werden“<sup>29</sup>. Dies ist irrelevant bei Objekten, die nicht optisch dargestellt werden (wie das zum Beispiel bei einer Schaltfläche der Fall ist).

---

<sup>28</sup> Vgl. Abschnitt Kapselung im Kapitel Java auf Seite 14

<sup>29</sup> Ebner, 2000, S. 114

```

type
  TTestklasse = class(TObject)
  private
    PrivateEigenschaft : Integer;
    function PrivateMethode(Param1 : String): Double;
  protected
    GeschuetzteEigenschaft : String;
  published
    VeroeffentlichteEigenschaft : Boolean;
    procedure VeroeffentlichteMethode;
end;

```

Die Deklaration der Methoden erfolgt entweder durch das Stichwort `function` (sofern ein Rückgabewert vorhanden ist) oder `procedure`. Die Deklaration eines Konstruktors und Destruktors erfolgt wie die einer Methode und wird anhand der Stichworte `constructor` oder `destructor` eingeleitet.

Die Funktionsweise wird, wie im obigen Überblick gezeigt, im `implementation`-Teil der Unit festgelegt. Die Methoden werden durch eine Wiederholung der Deklaration einschließlich der gebrauchten Parameter eingeleitet. Da die Methoden aller in der Unit deklarierten Klassen gemischt werden, ist es notwendig, zusätzlich vor dem Namen der Methode getrennt durch einen Punkt (.) die Klassenbezeichnung hinzuzufügen. Innerhalb der Methode kann über die Variable `self` auf das Objekt zugegriffen werden.

```

function TTestklasse.PrivateMethode(Param1 : String): Double;
var
  // Variablendeklaration
begin
  // Code der Methode
  self.GeschuetzteEigenschaft := Param1;
end;

```

Die Deklaration einer Instanz erfolgt wie die Deklaration einer gewöhnlichen Variablen. Da jedoch, wie in Java, die deklarierte Variable nur eine Referenz auf das Objekt darstellt, muss zunächst das Objekt auch tatsächlich erzeugt werden. Dies geschieht durch die Methode `Create`.

```

procedure Test;
var
  Obj : TObject;
begin
  Obj := TObject.Create;

```

```
end;
```

### 4.2.2 Vererbung

Wie im Abschnitt Deklaration einer Klasse auf Seite 15 gezeigt, erfolgt die Vererbung durch die Angabe der Elternklasse nach dem Stichwort `class`. Das Überschreiben von Methoden erfolgt wie in Java durch die Deklaration einer namensgleichen Methode mit derselben Parameterdefinition. Zusätzlich kann Object Pascal das Stichwort `inherited`. Innerhalb einer überschreibenden Methode bewirkt es, dass die namensgleiche Methode der Elternklasse aufgerufen wird. Dies ist von besonderer Bedeutung, wenn der Konstruktor der Basisklasse `TObject` überschrieben wird, um eine einwandfreie Speicherbelegung zu gewährleisten<sup>30</sup> - `inherited` muss im neuen Konstruktor unbedingt angegeben werden. Weiters ist es möglich, durch das Hinzufügen des Stichwortes `overload` Methoden mit der gleichen Bezeichnung, aber unterschiedlichen Parameterdefinitionen festzulegen.

### 4.2.3 Typecasting

Variablen vom Typ einer Elternklasse kann ein Objekt der Kindklasse zugewiesen werden. Dieser Vorgang wird als *Typecasting* bezeichnet und wird dann verwendet, wenn beispielsweise einer Funktion entweder ein Objekt der Klasse A oder der Klasse B als Parameter übergeben werden muss<sup>31</sup> – die Funktion könnte dann so definiert werden, dass sie als Parameter ein Objekt der Klasse `TObject` entgegennimmt. Um festzustellen, um was für ein Objekt es sich handelt, existiert der Operator `is`. Um dann auf den Funktionsumfang der tatsächlich übergebenen Klasse zugreifen zu können, muss ein erneuter Typecast durchgeführt werden. Die entsprechende Syntax wird im folgenden Beispiel gezeigt.

```
function Beschri ften(ziel : TObject);  
const  
    text : String = 'Hello World';  
begin  
    if (ziel is TButton) then  
        // 1. Mögl i chkei t  
        (ziel as TButton).Caption := text  
    else
```

---

<sup>30</sup> Vgl. Geisler, Geisler, 2002, S. 156

<sup>31</sup> Vgl. Ebner, 2000, S. 120

```
if (ziel is TImage) then
    // 2. Möglichkeit
    TImage(ziel).Canvas.TextOut(0, 0, text);
end;
```

## 4.3 Perl

Perl spielt in vieler Hinsicht eine Sonderrolle. So auch in der objektorientierten Programmierung, die in Perl mit der 1997 erschienenen Version 5 Einzug gefunden hat. Da Perl ein hohes Potential an Dynamik bietet, fehlen Mechanismen wie die Kapselung. Dennoch werden die grundlegenden Möglichkeiten der Objektorientierung geboten.

### 4.3.1 Referenzen

Die Basis der objektorientierten Programmierung in Perl liegt im Verständnis des Variablenkonzepts<sup>32</sup>. Auf die drei Variablentypen und Subroutinen können so genannte *Referenzen* gelegt werden, durch die mit einer veränderten Syntax auf die Variable zugegriffen werden kann, wie nachfolgendes Beispiel zeigt.

```
sub manipulation {
    my $textreferenz = shift
    ${$textreferenz} = 'b';
}
sub ausgabe {
    my $text = 'a';
    print "$text\n";
    manipulation(\$text);
    print "$text\n";
}
ausgabe();
```

Das Skript gibt zunächst a und – nach der Manipulation – b aus. Nachfolgend einige Anmerkungen zum Verständnis des Programmcodes.

- Das Schlüsselwort `my` sorgt dafür, dass die nachfolgend deklarierte Variable in dem Block, in dem es steht, und in allen untergeordneten Blocks verfügbar ist. In der Subroutine `manipulation` kann also nicht auf den Skalar `$text` zugegriffen werden.

---

<sup>32</sup> Vgl. Anhang auf Seite iii

- Mit dem verkehrten Schrägstrich (\) vor dem Skalar `$text` wird eine Referenz erzeugt und diese der Subroutine `manipulation` als Parameter übergeben<sup>33</sup>.
- Diese wird mit der Funktion `shift` in den Skalar `$textreferenz` übernommen. Gibt man `$textreferenz` direkt aus, erhält man die Adresse, auf die `$textreferenz` verweist.
- Auf den Inhalt von `$text` kann nun mittels `$$textreferenz` zugegriffen werden. Würde es sich um ein Array handeln, würde die Syntax `@{$arrayreferenz}` lauten, wobei der Zugriff auf die einzelnen Arrayelemente mittels `$arrayreferenz->[$index]` zu erfolgen hat. Bei einem assoziativen Array, das in Perl meist als Hash bezeichnet wird, wäre dies `#{ $hashreferenz }` oder `$hashreferenz->{ $index }`.

Perl lässt auch Referenzen auf so genannte *anonyme Variablen* zu. Darunter sind Variablen zu verstehen, die keine Identität haben und auf die daher nicht zugegriffen werden kann, wenn keine Referenz auf sie vorliegt oder sie direkt im Kontext verwendet werden.

```
# Erstellen einer Referenz auf einen benannten Hash
my %hash = (abc0 => 'xyz0',
            abc1 => 'xyz1');
my $hashref1 = \%hash;

# Erstellen einer Referenz auf einen anonymen Hash
my $hashref2 = {abc0 => 'xyz0',
                abc1 => 'xyz1'};

if ($hashref1->{abc0} eq $hashref2->{abc0}) {
    # ...
}
```

`$hashref1` und `$hashref2` verfügen nun über die gleichen Daten, die gleichartig manipuliert und verwendet werden können. Ein anonymer Hash wird durch `{` und `}` erzeugt, ein anonymes Array durch `[` und `]`.

Perl verfolgt mit der Verwendung von Referenzen ein speicherschonendes Konzept: bis Version 5 mussten alle Daten kopiert und wieder zurückkopiert werden.

---

<sup>33</sup> Vgl. Abschnitt Parameter auf Seite 4

### 4.3.2 Module und Packages

Als *Modul* bezeichnet man das, was in anderen Sprachen als *Units* oder *Bibliotheken* bezeichnet wird. Es handelt sich hierbei um ausgegliederte in separaten Dateien stehende Programmteile, die wieder verwendet werden können. Für Perl steht eine große Sammlung an kostenlosen Modulen im Internet zur Verfügung, was als ein Grund für den Erfolg von Perl angesehen werden kann. Diese Module werden mittels des Befehls `use` eingebunden.

```
use CGI ;
use Net::FTP;
```

Alle Perl-Skripts werden in so genannte *packages* eingeteilt, die einen eigenen Namensraum für Variablen und Subroutinen zur Verfügung stellen. Das Package, in dem der Hauptteil des Programms abläuft, heißt *main* und muss nicht explizit eingeleitet werden.

```
# package main;
my $x = 'a';
print "$x\n";

package Testpackage;
my $x = 'b';
```

### 4.3.3 Klassendefinition

Eine Klasse ist in Perl ein einfaches Package, das aus mehreren Subroutinen und eventuell mehreren „globalen“ Variablen besteht, an welches eine Referenz gebunden wird. Eine Instanz ist also eine einfache Referenz. Die Erstellung und das Zuweisen dieser Referenz erfolgt in einer speziellen Subroutine, die analog zu anderen Sprachen als Konstruktor<sup>34</sup> zu bezeichnen ist. Sie wird meist als `new` definiert.

```
package Auto;

sub new {
    my $class = shift;
    my $farbe = shift;
    my $marke = shift;

    my $objekt_referenz = {farbe => $farbe,
```

---

<sup>34</sup> Vgl. Abschnitt Methodenarten auf Seite 6

```

        marke => $marke,
        geschwindigkeit => 0};
    bless $objekt_referenz, $class;
    return $objekt_referenz;
}

```

Die hier definierte Klasse heißt Auto. Der Konstruktor nimmt mehrere Parameter entgegen: der erste ist `$class` – er enthält die Bezeichnung der Klasse (Auto), die beim Aufruf automatisch übergeben wird. Die anderen beiden Parameter - `$farbe` und `$marke` – repräsentieren Eigenschaften, die bei der Initialisierung gesetzt werden sollen. Anhand dieser Eigenschaften und einer weiteren Eigenschaften (`geschwindigkeit`) wird eine Referenz auf einen anonymen Hash in dem Skalar `$objekt_referenz` abgelegt. Ein Objekt muss in Perl nicht unbedingt ein Hash sein, sondern kann auf jedem beliebigen Variablentypen basieren; jedoch wird meist ein assoziatives Array verwendet, da dieses am besten das Verhältnis Eigenschaft zu Wert darstellt und somit dem „klassischen“ Modell am ehesten entspricht.

`$objekt_referenz` enthält nun die Adresse des anonymen Hashes und wird mittels der Funktion `bless` (segnen) an das Package, dessen Bezeichnung in `$class` steht, gebunden. Das bedeutet, dass alle Subroutinen, die im eigentlichen Programmablauf in Zusammenhang programmiert werden – also als Methoden verwendet werden –, im Package `Auto` gesucht werden. Diese „gebundene“ Referenz und damit der Verweis auf den anonymen Hash wird nun durch die Funktion `return` von der Subroutine `new` als Rückgabewert verwendet.

Die Definition weiterer Methoden erfolgt wie die Definition gewöhnlicher Subroutinen im Package der Klasse, wobei auf die Besonderheit zu achten ist, dass als erster Parameter die Referenz auf das Objekt übergeben wird.

```

sub lackieren {
    my $self = shift;
    my $farbe = shift;

    $self->{farbe} = $farbe;
}

```

Diese Referenz wird, wie im Beispiel gezeigt, meist in der Variablen `$self` empfangen und kann dann wie eine gewöhnliche Referenz behandelt werden.

### 4.3.4 Verwendung von Objekten

Zur Instanziierung eines Objekts muss zunächst der Konstruktor eines objektorientierten Packages aufgerufen werden. Um Perl mitzuteilen, auf welches Package zugegriffen werden soll, wird die folgende Syntax verwendet:

```
my $mein_auto = Auto->new('gelb', 'Ferrari');
```

Die Referenz `$mein_auto` kann, ohne dadurch beeinflusst zu werden, dass sie ein Objekt darstellt, als Hash-Referenz verwendet werden. So ließen sich etwa alle Eigenschaften des Objekts folgendermaßen bequem auslesen:

```
foreach my $key (keys %{$mein_auto}) {  
    print "Die Eigenschaft $key ist $mein_auto->{$key}\n";  
}
```

Hier wird ersichtlich, dass Perl über keine Möglichkeit zur Kapselung verfügt. Somit wird dem Programmier/der Programmiererin selbst überlassen, inwieweit „sauber“ programmiert wird. Ein „schlechter Stil“ ist insofern verlockend, als die Programmierung von Schnittstellen, also Methoden zum Zugriff auf die Eigenschaften, gespart wird. Doch dieser Vorteil wird bei komplexeren Skripts, an denen womöglich mehrere Personen arbeiten, schnell eingebüßt.

Der Zugriff auf die Methoden erfolgt ähnlich wie der Zugriff auf den Konstruktor: statt der Klassenbezeichnung wird nun die Objektreferenz verwendet - das Trennzeichen `->` bleibt gleich:

```
$mein_auto->lackiere('hellgrün');
```

Der Wert vor dem Trennzeichen wird, wie beim Konstruktor auch, der Methode übergeben, sodass innerhalb der Methode auf das Objekt zugegriffen werden kann.

### 4.3.5 Vererbung

Auch die Vererbung gliedert sich in das einfach gehaltene Umsetzungskonzept Perls. So wird innerhalb des Packages ein Array mit der Bezeichnung `@ISA` (engl. „is a“ – „ist ein“) deklariert, in dem alle Klassen angegeben werden, die beerbt werden sollen.

```
package Auto;  
  
@ISA = ('Fahrzeug', 'Gegenstand');
```

Erfolgt über ein Auto-Objekt ein Zugriff auf eine Methode, die im Package `Auto` nicht definiert wurde, wird nach der Reihenfolge ihrer Eintragung im Array `@ISA` nach der Methode gesucht. Erfolgt also beispielsweise der Aufruf

```
$mein_auto->fahre('SSW', 35);
```

würde der Interpreter im Package `Fahrzeug` fündig werden und das Package `Gegenstand` nicht durchsuchen. Wäre der Aufruf allerdings

```
my $masse = $mein_auto->masse();
```

müssten die Packages `Auto` und `Fahrzeug` durchsucht werden, um schließlich den in der Klasse `Gegenstand` definierten `Accessor`<sup>35</sup> aufzurufen.

`@ISA` ist eine einfache Variable, die folglich auch zur Laufzeit manipuliert werden kann. Dies ermöglicht komplexe Konstrukte, anhand derer ein hohes Maß an Dynamik und in bestimmten Fällen auch an Geschwindigkeit erreicht werden kann<sup>36</sup>. Jedoch widerspricht dies grundlegend der Zielsetzung der objektorientierten Programmierung, die besagt, dass die Programme überschaubarer und der Natur ähnlich verfasst werden sollen. Allerdings muss bei hochperformanten Programmen häufig auf Übersicht verzichtet werden, was beispielsweise auch bei der Einbindung von maschinennahen `Assembler`<sup>37</sup>-Programmteilen passieren kann.

---

<sup>35</sup> Vgl. Abschnitt Methodenarten auf Seite 6.

<sup>36</sup> So könnte zur Laufzeit entschieden werden, welche Programmteile geladen werden müssen, wodurch unnötige Prüfung und Übersetzung von Quellcode erspart werden kann.

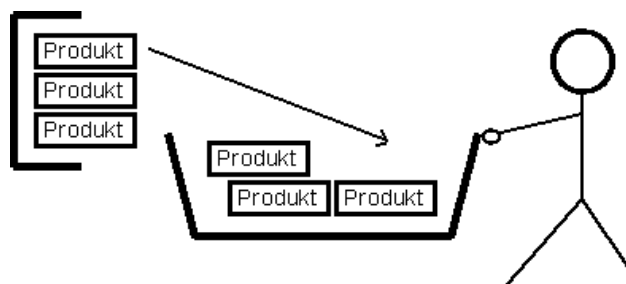
<sup>37</sup> `Assembler` ist eine Programmiersprache mit minimalem Befehlsumfang, mit dem der Prozessor fast direkt angesteuert wird.

## 5 Anwendung am Beispiel eines Online-Shops

Bei der Entwicklung eines Programms mittels objektorientierter Programmierung ist es nützlich, ähnliche Vorgangsweisen in der Natur zu suchen und die Frage zu stellen: „Was ist das und woraus besteht es?“. Beim Beispiel des Online-Shops<sup>38</sup>, der in Perl programmiert wurde, wäre dies das Szenario eines Supermarkts. Man teilt diesen in Regale ein, die Produkte enthalten, des Weiteren gibt es Kund/inn/en, Einkaufskörbe und eine Kassa.

Der nächste Schritt ist, diese Klassen auf das notwendige Mindestmaß zu reduzieren, also zu *abstrahieren* – für die Funktion des Online-Shops ist es irrelevant, dass Kund/inn/en weit mehr können, als einkaufen<sup>39</sup>. Weiters gruppiert man ähnliche Funktionsweisen und versucht somit, eine Elternklasse zu erkennen. Diese ist im vorgestellten Beispiel nicht offensichtlich, denn es handelt sich um den Datenbankzugriff.

Im weiteren Vorgehen werden die Objekte in Verbindung gesetzt: Der Kunde/die Kundin *hat* einen Einkaufskorb, der mehrere Produkte *haben kann*. Ebendies gilt für die Regale. Der Kunde/die Kundin kann in beliebiger Anzahl Produkte *in den Einkaufskorb legen*. Zwischen den Regalen gibt es Informationsschilder. All diese Objekte liegen innerhalb des Supermarkt-Objekts, das die Vorgänge kontrolliert.



### 5.1 Verwendung einer Datenbank

Da die Daten eines Online-Shops schnell und häufig geändert werden müssen, ist es sinnvoll, ein Datenbanksystem zu verwenden. Die Datenbankschnittstelle Perls heißt *DBI* und stellt eine Klasse zur Verfügung, die meist als *\$dbh* (Data base handler) in-

---

<sup>38</sup> Der vollständige, unvereinfachte Programmcode ist auf dem beigelegten Datenträger zu finden.

<sup>39</sup> In der Praxis der Supermärkte gilt dies ebenso.

stanziert wird. Über sie können SQL-Befehle an den Datenbankserver gesendet werden.

Die verwendeten Klassen basieren größtenteils auf Daten einer Datenbank. Daher ist es sinnvoll, eine Basisklasse zu erstellen, durch welche die grundlegenden Zugriffsformen - Abfragen, Einfügen, Verändern und Löschen von Daten – implementiert werden. Im vorliegenden Beispiel heißt diese Klasse `dbentry` und ist wie folgt aufgebaut:

- Der Konstruktor `new` nimmt als Parameter entweder ein Hash mit allen Daten oder eine Identifikationsnummer (*Primary key*), über die der Datenbankeintrag eindeutig identifiziert werden kann, entgegen. Weiters wird eine Referenz auf die Datenbankverbindung (`$dbh`) übergeben und als Eigenschaft gespeichert. Hierbei handelt es sich, wie auf Seite 8 beschrieben, um eine Assoziation, da `$dbh` mehrfach verwendet wird.
- Sollten die Daten nicht übergeben worden sein, werden sie geladen. Um den entsprechenden SQL-Befehl zu erstellen, verwendet die Klasse die Methoden `FIELDS` und `TABLE`, die in der Basisklasse selbst nicht definiert wurden und daher in den abgeleiteten Klassen implementiert werden müssen<sup>40</sup>. Eigentlich sind die genannten Methoden konstante Eigenschaften, jedoch stellt Perl keine Möglichkeit zur Deklaration von Konstanten zur Verfügung, weshalb Subroutinen, die einen konstanten Wert zurückliefern, verwendet werden müssen.
- Nach dem Laden werden die Daten in einem assoziativen Array gespeichert und können beliebig über die Schnittstellenmethoden `get_param` und `set_param` manipuliert werden.
- Um unnötige Datenbankzugriffe zu vermeiden, muss zum Speichern der veränderten Daten die Methode `save` verwendet werden – die Speicherung erfolgt nicht automatisch im Destruktor des Objekts.

Von dieser Basisklasse leiten sich folgende Klassen ab:

- `product`  
Die Klasse `product` erweitert die Klasse `dbentry` um eine Methode namens

---

<sup>40</sup> Diese Konstruktion ist nur in Scriptsprachen möglich, da diese ein großes Maß an Dynamik bieten.

file, mittels derer das Produkt über eine Verbindungstabelle<sup>41</sup> einer Kategorie zugeordnet werden kann.

- **category**  
category verfügt über eine Methode `products`, die alle zugeordneten Produkte in Form eines Arrays mit Referenzen auf `product`-Objekte zurückgibt. In der oben gezeigten Skizze entspräche diese Klasse den Regalen.
- **order**  
Das von dieser Klasse instanziierte Objekt speichert diverse Daten, die den Bestellvorgang an sich betreffen (beispielsweise den Bestellstatus, die Versandart etc.). Zusätzlich muss es eine Methode geben, die der Methode `products` der Klasse `category` sehr ähnlich ist, jedoch zusätzlich eine Anzahl verwalten kann. Weiters wird eine Methode `add` definiert, über die ein Produkt in einer bestimmten Menge der Bestellung hinzugefügt werden kann. Im Supermarkt entspricht diese Klasse dem Einkaufswagen.
- **user**  
Das `user`-Objekt speichert Daten wie den Namen oder die E-Mail-Adresse des Benutzers/der Benutzerin.

## 5.2 Die shop-Klasse

Das `shop`-Objekt muss in Abhängigkeit von der Aufgabe die oben beschriebenen Klassen instanzieren und steuern. Zusätzlich muss es eine Liste aller Kategorien erzeugen können.

In jedem Fall muss ein `order`- und ein `user`-Objekt erzeugt werden, um allgemeine Daten (wie die Gesamtsumme der bestellten Produkte) anzeigen zu können. Alles Weitere hängt von der gewählten Aufgabe ab:

- **Produkte anzeigen**  
Ein `category`-Objekt wird anhand der übergebenen Id erzeugt. Die von diesem Objekt erzeugten `product`-Objekte werden mit Hilfe einer Vorlage zu einer HTML-Seite verarbeitet.

---

<sup>41</sup> Produkte und Kategorien sind in separaten Tabellen gespeichert. In einer dritten Tabelle werden jeweils die Id einer Kategorie sowie die Id eines Produkts miteinander verknüpft, sodass anhand einer SQL-Abfrage Produkte eindeutig einer Kategorie zugeordnet werden können.

- Ein Produkt in den Warenkorb legen

Die Methode `add` des `order`-Objekts wird aufgerufen und erhält als Parameter die `Id` und `Anzahl` des gewählten Produkts. Anschließend wird derselbe Vorgang wie bei der Produktanzeige ausgelöst.

- Bestellung versenden

Zunächst muss der Warenkorb anhand der Methode `products` des `order`-Objekts angezeigt werden. Anschließend müssen durch ein Formular die Benutzer/innen/daten im `user`-Objekt abgelegt und gespeichert werden (sofern dies nicht bei einer früheren Bestellung geschehen ist). Nach einer weiteren Kontrollanzeige aller Daten kann der Status des `order`-Objekts geändert und in der Datenbank abgelegt werden.

## 6 Schlussbemerkungen

Selbstverständlich wäre die Entwicklung des oben vorgestellten Online-Shops auch ohne objektorientierte Programmierung möglich gewesen. Die Arbeit und besonders die Fehlersuche wurden jedoch erheblich erleichtert, da jede Klasse isoliert in einer einfachen Testumgebung geprüft werden konnte. Zusätzlich kann und wird die zum Datenbankzugriff verfasste Klasse veröffentlicht und weiterverwendet werden.

Gerade heute, wo oft unzählige Programmierer/innen an einem Programm arbeiten müssen, um Innovationen zu erschaffen, gewinnt die sinnvolle Strukturierung, die durch die objektorientierte Programmierung gewährleistet ist, an Bedeutung.

„*'Alles ist ein Objekt!' – Diese Phrase mag schon keiner mehr hören.*“<sup>42</sup>, schreibt Michael Schilli. An dieser Betrachtung liegt sicherlich etwas Wahres, zumal die objektorientierte Programmierung ideologisiert wurde: ihrem Ziel kommt man nicht näher, indem Integer-Variablen zum Objekt gemacht werden. Sinnvoller ist es beispielsweise bei der Objektdatenbank ZOPE, die bei der Entwicklung von Content Management Systemen im Internet zum Einsatz kommt: die generierten HTML-Seiten entstammen einer Zusammenarbeit von mehreren einfachen Objekten, die mit ihren Eigenschaften in einer besonderen Datenbank abgelegt wurden.

In der nächsten Sprachengeneration wird *„die Problembeschreibung zugleich das Programm. Man muss nicht mehr den 2. Schritt gehen, das Wie der Problemlösung zu erstellen.“*<sup>43</sup>. Auch in diesem Ansatz werden Objekte eine Rolle spielen, wenn auch in einem anderen Sinn, nämlich zur Beschreibung von zu lösenden Vorgängen – denn: alles ist ein Objekt.

---

<sup>42</sup> Schilli, 1998, S. 138

<sup>43</sup> Cleve, Lämmel, 2001, S. 135

## 7 Anhang

### 7.1 Literaturverzeichnis

**Baufeld; Friedrich; Mäuers; Mühle; Müller; Wabnitz:**

Java – Das Fundament professioneller Java-Programmierung. - Düsseldorf: Data Becker, 1999.

**Christiansen, Tom:**

Tom's object-oriented tutorial for perl. <http://perldoc.perl.org/perltoot.html>, 01.11.2005

**Cleve, Jürgen; Lämmel, Uwe:**

Künstliche Intelligenz. – München, Wien: Carl Hanser. 2001

**Ebner, Michael:**

Delphi 5 nachschlagen und verstehen. - München: Addison Wesley Longman, 2000.

**Erlenkötter, Helmut; Reher, Volker; Moos, Ludwig (Hrsg.):**

Java – HTML, Scripts, Applets und Anwendungen. - Hamburg: Rowohlt Taschenbuch, 1997.

**Dominus, Mark-Jason:**

Mark's very short tutorial about references. <http://perldoc.perl.org/perlreftut.html>, 01.11.2005

**Geisler, Frank; Geisler, Sandra:**

Delphi GE-PACKT. – Bonn: mitp, 2002.

**Schilli, Michael:**

GoTo Perl 5. - Berlin: Addison Wesley Longman, 1998.

**Steppan, Bernhard:**

Objektorientierte Programmierung.

<http://www.galileocomputing.de/artikel/gp/artikelID-215>, 01.11.2005

## 7.2 Kurzübersicht über die verwendeten Programmiersprachen

### 7.2.1 Java

Java unterscheidet zwischen Groß- und Kleinschreibung. Kommentare stehen entweder zwischen `//` und dem Zeilenende, `/*` und `*/` oder – zur Generierung automatischer Dokumentationen – `/**` und `*/`. Blöcke stehen zwischen `{` und `}`.

#### 7.2.1.1 Operatoren

<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>	Mathematische Grundoperatoren (+ ist auch auf Strings anwendbar und nimmt eine automatische Typkonvertierung vor)
<code>=</code>	Zuweisung
<code>==</code>	Vergleich

### 7.2.2 Delphi/Object Pascal

Pascal unterscheidet nicht zwischen Groß- und Kleinschreibung. Kommentare stehen entweder zwischen `//` und dem Zeilenende, `{` und `}` oder `/*` und `*/`. Blöcke stehen zwischen den Schlüsselwörtern `begin` und `end`. Die bekannteste Entwicklungsumgebung für einen Object Pascal-Dialekt<sup>44</sup>, die gleichzeitig als Compiler<sup>45</sup> fungiert, heißt *Borland Delphi* und wird häufig als Synonym für Object Pascal verwendet.

#### 7.2.2.1 Operatoren

<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code>	Mathematische Grundoperatoren (+ ist auch auf Strings anwendbar)
<code>:=</code>	Zuweisung, Bsp.: <code>a := 2</code>
<code>=</code> , <code>&gt;</code> , <code>&lt;</code> , <code>=&gt;</code> , <code>&lt;=</code>	Vergleich: gleich, größer als, kleiner als, größer gleich als, kleiner gleich als
<code>is</code>	Typenprüfung

#### 7.2.2.2 Variablentypen

Integer	Zahlenwert
Double, Real, Extended	Reelle Zahl
String	Zeichenkette
Array [x .. y] of z	Zusammenfassung mehrerer Variablen des Typs z, die über einen Index zwischen x und y aufgerufen werden können, z.B. <code>array[2]</code>

<sup>44</sup> Ein *Dialekt* einer Programmiersprache bezeichnet eine leicht veränderte Version.

<sup>45</sup> Als *Compiler* wird ein Programm bezeichnet, welches aus einem Programmcode ein selbstständig lauffähiges Programm erstellt.

### 7.2.2.3 Funktionen

`IntToStr`  
Umwandlung einer Zahl (Integer) zu  
Text (String)

`WriteLn`  
Ausgabe einer Textzeile

`Sleep`  
Programmablauf für eine gegebene  
Zeitspanne unterbrechen

### 7.2.3 Perl

*Perl (Practical Extracting and Report Language* – Praxisnahe Extraktions- und Berichtssprache) ist eine zwischen Groß- und Kleinschreibung unterscheidende Scriptsprache, die 1987 von Larry Wall veröffentlicht wurde. Ziel war, möglichst einfach Texte automatisch bearbeiten und auswerten zu können. Inzwischen ist Perl so weit entwickelt, dass es eine universell einsetzbare Sprache wurde. Dennoch liegt ihr Haupteinsatzgebiet in der Entwicklung von CGI-Scripts<sup>46</sup> und in der Systemadministration.

Kommentare stehen zwischen `#` und dem Zeilenende, Blöcke zwischen `{` und `}`.

#### 7.2.3.1 Variablen

Die optionale, aber zu einem guten Programmierstil gehörende Variablendeklaration erfolgt mit dem Stichwort `my` für den aktuellen und alle untergeordnete Blöcke und mit dem Stichwort `our` für das gesamte Package. Bei Verwendung des Pragmas<sup>47</sup> `strict` führen nicht deklarierte Variablen zu einer Fehlermeldung.

In der Verwendung unterscheidet Perl nicht zwischen Boolean-, Integer-, Float- und String-Werten. Diese werden als *Skalare* zusammengefasst und sind am Dollarzeichen (\$) vor der Variablenbezeichnung zu erkennen. Die Typkonvertierung erfolgt automatisch, jedoch kann durch unterschiedliche Operatoren, wie auf Seite iv beschrieben, eine Typunterscheidung vorgenommen werden.

```
my $x = 25;
print $x, "\n";
```

Des Weiteren kennzeichnet Perl Arrays, also eine Reihe von über einen Index zugreifbarer Skalaren, durch ein vorgestelltes `@`. Da die einzelnen Elemente Skalare

<sup>46</sup> Scripts, die über die CGI-Schnittstelle via HTTP im Internet zum Einsatz kommen und dynamisch generierten Inhalt bieten können.

<sup>47</sup> Als *Pragma* werden in Perl jene Module bezeichnet, die das Verhalten des Interpreters beeinflussen.

sind, ist beim Zugriff auf ein einzelnes Element \$ zu verwenden, während beim Zugriff auf eine Liste oder bei der Deklaration und Referenzerzeugung<sup>48</sup> @ genutzt wird.

```
my @array = ('Eintrag1', 'Eintrag2', 'Eintrag3');
# auf das erste Element wird mit 0 zugegriffen
print $array[0], "\n";
print @array[1, 2];
my $arrayreferenz = \@array;
```

Die Größe von Arrays ist dynamisch. So kann dem Array über die Funktion push ein neuer Wert zugefügt werden. Des Weiteren ist es möglich, einen beliebigen noch nicht definierten Index zu verwenden.

```
my @array = ();
push @array, 'Wert1';
$array[10] = 'Wert11';
```

Im Gegensatz zu Arrays verfügen *Hashes* bzw. *assoziative Arrays* über einen String als Index und verlangen % als Bezeichner. Bei der Deklaration werden *Keys* - also dem späteren Index - *Values* zugewiesen, indem sie voneinander durch das Zeichen => getrennt werden: Der Zugriff erfolgt statt der eckigen Klammern ([*\$index*]) durch geschwungene ({*\$index*}).

```
my %hash = (Key1 => 'Value1',
            Key2 => 'Value2');
print $hash{key1}, "\n";
# Zugriff auf mehrere Elemente
print @hash{'key1', 'key2'};
my $hashreferenz = \%hash;
```

### 7.2.3.2 Operatoren

#### Numerisch

==, !=, <, >, <=, >=

=

+

\*

/, -

#### String

eq, ne, lt, gt, le, ge

=

.

x

Nicht vorhanden

#### Bedeutung

Vergleich: gleich, ungleich, kleiner als, größer als, kleiner gleich, größer gleich

Zuweisung

Addition (bei Strings Verbindung)

Multiplikation

Mathematische Grundoperatoren

<sup>48</sup> Vgl. Kapitel Referenzen auf Seite 18

### 7.3 Danksagung

Ich möchte mich bei allen bedanken, die mich indirekt oder direkt bei dem Verfassen der Arbeit unterstützt haben.

- Der Gemeinde von [spotlight.de](http://spotlight.de),  
die mich gelehrt hat, vernünftig zu programmieren,
- meinem Hund Yoda,  
der mich gezwungen hat, meine Arbeit zu unterbrechen,
- meinem Betreuer Carl Metnitz,  
der mich gezwungen hat, die Arbeit wieder fortzusetzen,
- meinem Chef Karl Egger,  
der zeitlich auf mich Rücksicht nahm und mich mit großzügigen Schokoladen-  
spenden versorgte,
- den Macher/inne/n von [dnb-sets.de](http://dnb-sets.de),  
die mich unermüdlich und kostenlos mit Musik versorgt haben,
- und den Mitwirkenden von Wikipedia,  
ohne die Recherche etwas Aufwendiges wäre.