webmasters press

Jan Teriete

Objektorientiertes PHP

Band 1: Grundlagen der OOP

- Vorteile der OOP verstehen
- Praktischer Einstieg in die Objektorientierung
- Saubere Strukturierung von Web-Anwendungen
- Mit vielen Übungen und Code-Beispielen

Autorisiertes Curriculum für das **Webmasters Europe** Ausbildungs- und Zertifizierungsprogramm

we

Jan Teriete

Objektorientiertes PHP

Band 1: Grundlagen der OOP

Ein Webmasters Press Lernbuch

Version 8.0.0 vom 18.1.2017

Autorisiertes Curriculum für das Webmasters Europe Ausbildungs- und Zertifizierungsprogramm.

www.webmasters-europe.org

www.webmasters-press.de

Webmasters Akademie Nürnberg GmbH
Neumeyerstr. 22–26
90411 Nürnberg
Germany
www.webmasters-akademie.de

Printed books made with Prince

Art.-Nr. 1289ae27409a
Version 8.0.0 vom 18.1.2017

Inhaltsverzeichnis

1 Einführung

In dieser Lektion lernen Sie
• welche Ziele dieses Lernbuch hat. • was Entwurfsmuster sind. • was Sie an Wissen mitbringen sollten, um erfolgreich mit diesem Lernbuch arbeiten zu können.

1.1 Einleitung

Dieses Buch versucht, Ihnen eine neue Sichtweise auf die PHP-Programmierung zu vermitteln. Bisher haben Sie wahrscheinlich vorwiegend kleinere Projekte mit PHP umgesetzt. Je größer ein Projekt allerdings wird, desto wichtiger, aber auch schwieriger ist es, die Übersicht zu behalten. Ihnen die Fähigkeiten dazu zu vermitteln, ist das Ziel dieses Lernbuchs.

Natürlich werden Sie auch neue PHP-Techniken erlernen. Sie werden in die Syntax der objektorientierten Programmierung eingeführt und lernen abschließend MVC kennen.

Im wichtigsten Teil dieses Buches geht es darum, wie Sie die neuen und auch die bereits bekannten Techniken einsetzen können, um sauberen, wartbaren und vor allem lesbaren Code zu schreiben. Wenn Sie dieses Buch durchgearbeitet haben, werden Sie nicht nur die Grundlagen der Objektorientierung beherrschen, Sie werden Ihre OOP-Kenntnisse (objektorientierte Programmierung) einsetzen können, um Code zu schreiben,

- den Sie auch nach Wochen und Monaten sofort wieder verstehen.
- der sich fast ein wenig so liest, wie Sie als Mensch sprechen würden.
- in dem Sie sofort, ohne groß zu suchen, die fragliche Codestelle finden.
- der Sie erfreulicherweise immer seltener überraschen wird (»Warum zum …«).

1.2 Was sind Entwurfsmuster?

Ich versuche Ihnen, wie gesagt, in diesem Buch zu vermitteln, wie Sie immer größer werdende PHP-Projekte wartbar und übersichtlich halten. Zu diesem Zweck werde ich auch auf ***Entwurfsmuster*** zu sprechen kommen. Entwurfsmuster oder englisch ***design patterns*** sind, kurz gesagt, eine praktische Möglichkeit, von der Intelligenz und Erfahrung anderer Programmierer zu profitieren (siehe auch https://de.wikipedia.org/wiki/Entwurfsmuster).

Sie können sich sicher sein, dass jedes Programmierproblem, an dem Sie gerade knobeln, schon mindestens 100 Leute vor Ihnen gelöst haben. Viele dieser Probleme treten sogar so häufig auf, dass es sich gelohnt hat, eine standardisierte Lösung zu entwerfen. So existieren inzwischen zu den gängigen Problemen der PHP-Programmierung fertige Vorgehensweisen, wie diese zu lösen sind. Diese Lösungen bestehen nicht aus fertigem Code, sondern versuchen nur den besten Weg zu zeigen, ein Problem anzugehen.

Daher nennt man sie Entwurfsmuster. Sie bieten eine Art Bauplan, nach dem Sie Ihren Code schreiben können. Glauben Sie mir, es spart wirklich eine Menge Zeit und Nerven, wenn man Lösungsansätze kennt, die garantiert funktionieren, weil sie ständig verwendet werden.

1.3 Vorkenntnisse

Für die Arbeit mit diesem Lernbuch sollten Sie folgende Fähigkeiten mitbringen:

- Sie sollten die grundlegenden PHP-Techniken beherrschen (Variablen, Konstanten, Zuweisungen, Blöcke usw.).
- Sie sollten mit den Datentypen Integer, Float, String und Array umgehen können.
- Sie sollten die gängigen Kontrollstrukturen wie Schleifen (`for`, `while`, `foreach`) und Verzweigungen (`if-else`, `switch`) beherrschen.
- Sie sollten Funktionen schreiben können, die Parameter erwarten und

Rückgabewerte erzeugen.

- Sie sollten gängige PHP-Funktionen beherrschen und sich bei Bedarf selbstständig neue aneignen können (http://php.net/).
- Sie sollten möglichst etwas Erfahrung in der Entwicklung kleiner oder mittlerer Webapplikationen mitbringen.
- Sie sollten auf jeden Fall wissen, wo Sie PHP-Dateien auf Ihrem Entwicklungsrechner ablegen müssen und wie Sie diese im Browser aufrufen. Dies geht meist über eine URL in der Form `http://meinserver/php_datei.php`. Auf einem lokal installierten Webserver entspricht `meinserver` normalerweise `localhost` (ohne Portangabe) und bei externen Webservern dem FQDN (siehe https://de.wikipedia.org/wiki/Domain).

Um das Wichtigste nicht zu vergessen: Sie sollten Spaß daran haben, an Ihrem Code herumzuknobeln und neue Dinge auszuprobieren.

1.4 Aufbau der Lektionen

Zu Beginn jeder Lektion gibt es einen Abschnitt, der Ihnen die Lernziele vermittelt. Meistens werden Sie vor ein konkretes Problem gestellt oder es wird Ihnen zumindest gezeigt, an welchen Stellen Sie mit Ihren momentanen Kenntnissen Schwierigkeiten haben. Im Rest der Lektion erarbeiten Sie sich dann Schritt für Schritt die für die Lösung notwendigen Kenntnisse und Fähigkeiten.

1.4.1 Aufgaben im Fließtext

Sie werden immer wieder mitten im Text auf Aufgaben stoßen. Diese erfordern normalerweise nur einen recht geringen Zeitaufwand. Sie dienen entweder als Vorbereitung auf kommende Inhalte oder sollen Ihnen die soeben erklärten Konzepte noch einmal verdeutlichen. Erläuterungen finden Sie oftmals im nachfolgenden Text.

1.4.2 »Testen Sie Ihr Wissen«

Gegen Ende einer Lektion finden Sie meist einen Abschnitt mit Fragen. Hier können Sie testen, wie viel Sie bereits von dem Lernstoff verstanden und behalten haben. Gehen Sie die Fragen gewissenhaft durch und beginnen Sie erst mit der nächsten Lektion, wenn Sie alle Fragen korrekt beantworten können.

Am Ende des Lernbuchs gibt es zu allen Fragen aller Lektionen die Antworten. Machen Sie sich aber bitte zuerst Gedanken über die Fragen, ehe Sie die Antworten nachschlagen.

1.4.3 »Aufgaben zur Selbstkontrolle«

Mit den »Aufgaben zur Selbstkontrolle« können Sie prüfen, ob Sie den gelernten Stoff wirklich anwenden können. Bearbeiten Sie diese Aufgaben erst, wenn Sie alle Aufgaben im Fließtext bereits erledigt haben. Sie sollten erst dann mit einer neuen Lektion beginnen, wenn Sie alle Aufgaben gelöst und auch verstanden haben.

Die Lösungen zu den Übungsaufgaben jeder Lektion finden Sie im Begleitmaterial.

1.4.4 Optionale Aufgaben

Die optionalen Aufgaben testen ebenfalls den Stoff der jeweiligen Lektion, aber die Aufgaben sind umfassender und meistens auch schwieriger. Die Aufgaben verbinden auch oft den Stoff der aktuellen Lektion mit den Techniken vorheriger Lektionen. Wenn Sie bisher wenig Erfahrung in der Programmierung haben, werden Sie eventuell nicht alle optionalen Aufgaben sofort selbstständig lösen können.

Ist das der Fall, machen Sie sich keine Sorgen und gehen Sie zur nächsten Lektion über. Die Pflichtübungen prüfen alle wichtigen Grundlagen ab. Kehren Sie, nachdem Sie das ganze Buch durchgearbeitet haben, noch einmal zu den optionalen Aufgaben zurück. Sie werden sehen, dass Sie diese nun lösen

können.

Es gibt einige Lektionen, in denen die Aufgaben aufeinander aufbauen. Dies gilt jeweils für die Pflichtaufgaben und für die optionalen Aufgaben; wenn Sie also in einer vorangegangenen Lektion nicht alle optionalen Aufgaben gelöst haben, werden Sie in der weiterführenden Lektion zwar alle Pflichtübungen durchführen können, aber bevor Sie zu den optionalen Aufgaben übergehen, sollten Sie die der vorhergehenden Lektion lösen.

1.5 Anforderungen an PHP

Der zweite Band dieses OOP-Lernbuchs verwendet *Doctrine 2*, um Daten in einer MySQL-Datenbank abzulegen und dort wieder auszulesen. Doctrine 2.5 setzt PHP ab Version 5.4 voraus. Sollten Sie eine ältere PHP-Version verwenden, so ist eine Aktualisierung zwingend notwendig. Da jedoch die EOL-Phase von PHP 5.5 am 21.07.2016 endete, gehe ich davon aus, dass Ihnen mindestens eine Entwicklungsumgebung mit **PHP 5.6** zur Verfügung steht. Ich selbst nutze übrigens seit September 2016 eine XAMPP-Installation mit PHP 7.0.

Die Beispiele und Aufgaben wurden ursprünglich auf einem System mit der PHP-Version **5.3.14** entwickelt und getestet. Auf die Behandlung neuer Features aus PHP 5.4, 5.5 oder 5.6 habe ich an vielen Stellen bewusst verzichtet, da die Angabe von zwei oder gar drei Varianten bei den Code-Beispielen den Seitenumfang dieses Lernbuchs gesprengt hätte. Der Code ist aber natürlich trotzdem so ausgelegt, dass er mit neueren PHP-Versionen lauffähig ist.

Ich gehe in allen Beispielen und Musterlösungen mit Datums- bzw. Zeitangaben davon aus, dass in der *php.ini* des verwendeten Webservers bereits die **korrekte Zeitzone** (z. B. `'Europe/Berlin'`) hinterlegt wurde.

2 Strukturierung von PHP-Webprojekten

In dieser Lektion lernen Sie
• wie Sie durch eine einfache Strukturierung des Codes Ihre PHP-Anwendung übersichtlicher gestalten.

2.1Das Problem

Einer der Gründe, die PHP so beliebt und erfolgreich machen, ist die einfache Handhabung. Sie können direkt in HTML-Dateien, die natürlich die Endung `.php` haben sollten, PHP-Tags öffnen und beginnen, Code zu schreiben. Leider erweist sich gerade das, was PHP so einsteigerfreundlich macht, für größere Projekte als Problem.

Vielleicht haben Sie einige der folgenden Situationen auch schon einmal erlebt:

- Sie haben Mühe, sich im PHP-Code eines Kollegen zurechtzufinden.
- Sie haben Mühe, sich in PHP-Code zurechtzufinden, den Sie selbst erst kürzlich geschrieben haben.
- Sie mussten im Code eines Kollegen über mehrere eingebundene Dateien (`include` bzw. `require`) hinweg suchen, wo genau eine Variable geändert wurde.
- Sie haben versehentlich dieselben (globalen) Variablen-Namen an mehreren Stellen verwendet und somit plötzlich Fehler an Stellen Ihres Programms, die Sie gar nicht geändert haben.
- Sie arbeiten mit einem Web-Designer zusammen, der kein PHP kann. Dieser beschwert sich ständig über die seltsamen Tags in den Dateien, die er nicht versteht.
- Derselbe Designer hat auch schon mehrmals bei seinen Layout-Arbeiten versehentlich Fehler in Ihren PHP-Code eingeschleust.

Gerade weil man in PHP »einfach drauflos« programmieren kann, ist der Code ab einem gewissen Umfang oft nur noch schwer zu erweitern. Die Fähigkeit von PHP, HTML und PHP zu vermischen, bewirkt regelmäßig Frust bei den Web-Designern, die ihre HTML-Tags mitten im PHP-Code suchen müssen.

Sie können dieser Liste sicher noch etliche weitere Beispiele hinzufügen.

2.2 Strukturierung von PHP-Code

Bedeutet dies, dass PHP eine schlechte Programmiersprache ist? Ist PHP etwa für alle Aufgaben außer für kleine Seiten mit einigen dynamischen Elementen ungeeignet?

Es stimmt, dass PHP schlechten Programmier-Stil zwar nicht aktiv fördert, aber doch zumindest zulässt. Es ist aber ebenso möglich, sehr lesbaren, sauberen und erweiterbaren Code zu schreiben und damit auch in größeren Projekten die Übersicht zu behalten. Dabei soll Ihnen diese Lektion helfen.

2.2.1 PHP und HTML trennen

Als Sie mit PHP angefangen haben, waren Sie wahrscheinlich begeistert davon, wie einfach Sie PHP in Ihre Webseiten einbetten konnten: einfach einen PHP-Tag öffnen und losprogrammieren. Je größer Ihre Dateien aber wurden, desto mehr Zeit haben Sie wahrscheinlich mit der Suche nach dem PHP-Fragment verbracht, das Sie gerade anpassen wollten. Ihr PHP-Code war auf viele einzelne, kurze Blöcke aufgesplittet, die sich auf das ganze HTML-Dokument verteilten.

Als Sie dann für das Layout einen Web-Designer mit ins Boot geholt haben, musste sich dieser umgekehrt die HTML-Fragmente in Ihrem PHP-Code zusammensuchen.

Beispiel

```
1 <!DOCTYPE html>
2 <html>
```

```
3
4  <head>
5      <meta charset="utf-8" />
6      <title>News anzeigen</title>
7  </head>
8
9  <body>
10     <h1>Die aktuellen Nachrichten</h1>
11     <?php
12
13     $nachrichten = unserialize(file_get_contents('nachrichte
14     foreach ($nachrichten as $n):
15         echo "<h2>" . $n['titel'] . "</h2>";
16         echo "<p>" . $n['inhalt'] . "</p>";
17         echo "<p>geschrieben von " . $n['autor'] . " am ";
18         echo $n['datum'] . "</p>";
19     endforeach;
20
21     ?>
22 </body>
23
24 </html>
```

news_anzeigen.php (Version 1)

***Codebeispiel 1** news_anzeigen.php (Version 1)*

Auf den ersten Blick ein sauber strukturiertes PHP-Programm, welches sogar zur besseren Lesbarkeit die alternative Syntax von Kontrollstrukturen verwendet. Ich werde diese Syntax übrigens immer dann verwenden, wenn sich Kontrollstrukturen innerhalb der Ausgabe befinden, welche durch den Doctype eingeleitet wird. Dennoch ist diese Datei aus Sicht eines Designers ohne PHP-Kenntnisse schon ein Problem. Er muss mitten in Ihren PHP-Tags Anpassungen vornehmen und kann dabei unbeabsichtigt Fehler in den PHP-Code einbauen. Das ist gar nicht so unwahrscheinlich, denn wenn der Designer kein PHP kann, versteht er nicht, welche Bedeutungen die einzelnen Zeichen haben.

Beispiel

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8" />
6      <title>News anzeigen</title>
7      <link href="shared/styles.css" rel="stylesheet" type="te
8  </head>
```

```
 9
10 <body>
11     <h1>Die aktuellen Nachrichten</h1>
12     <?php
13
14     $nachrichten = unserialize(file_get_contents('nachrichte
15     foreach ($nachrichten as $n):
16         echo "<h2>" . $n['titel'] . "</h2>";
17         echo "<p class="inhalt">" . $n['inhalt'] . "</p>";
18         echo "<p class="autor">geschrieben von " . $n['autor
19         echo $n['datum'] . "</p>";
20     endforeach;
21
22     ?>
23 </body>
24
25 </html>
```

news_anzeigen.php - Fehler (Version 2)

***Codebeispiel 2** news_anzeigen.php - Fehler (Version 2)*

Betrachten Sie die Änderungen, die unser hypothetischer Designer hier vorgenommen hat. Wie es modernes Webdesign oftmals verlangt, hat er die beiden p-Tags mit `class`-Attributen ausgestattet, um sie mittels CSS zu formatieren. Außerdem hat er ein externes Stylesheet eingebunden.

Übung 1:

1. Versuchen Sie die Fehler zu finden, die der Designer versehentlich eingebaut hat.
2. Denken Sie darüber nach, wie **Sie als Programmierer** hätten verhindern können, dass dieser Fehler passiert.

Wahrscheinlich haben Sie die beiden Probleme schnell entdeckt. Mit den doppelten und einfachen Anführungszeichen hatten Sie schließlich als Anfänger genug zu kämpfen. In den Zeilen 17 und 18 hat der Designer zwei CSS-Klassen in die PHP-Tags eingebaut und, wie es in HTML üblich ist, die Klassen-Namen in **doppelte** Anführungszeichen gesetzt. Diese haben Sie aber schon als umschließende Zeichen für die `echo`-Anweisung verwendet, also haben wir an diesen Stellen Syntax-Fehler, da PHP annimmt, dort würde der String enden.

Beispiel

```php
<?php

$nachrichten = unserialize(file_get_contents('nachrichten.tx
foreach ($nachrichten as $n):
    echo '<h2>' . $n['titel'] . '</h2>';
    echo '<p class="inhalt">' . $n['inhalt'] . '</p>';
    echo '<p class="autor">geschrieben von ' . $n['autor'] .
    echo $n['datum'] . '</p>';
endforeach;

?>
```

news_anzeigen.php - Ausschnitt (Version 2b)

Codebeispiel 3 *news_anzeigen.php - Ausschnitt (Version 2b)*

Nun könnten Sie das Problem umgehen, indem Sie nur einfache Anführungszeichen in PHP verwenden. Auf diese Weise kann der Designer nicht versehentlich einen String beenden. Doch das löst nicht das eigentliche Problem: Der Designer muss sein HTML immer noch in Ihrem PHP-Code suchen und ändern. Es können immer weitere Fehler auftreten, die ein Nicht-Programmierer weder erkennen noch beheben kann.

Es wäre doch besser, wenn der Designer von unserem PHP-Code gar nichts sehen würde. Ganz können wir ihm diesen Wunsch nicht erfüllen, aber wir können diesem Ziel zumindest näher kommen.

Beispiel

```php
<?php

$nachrichten = unserialize(file_get_contents('nachrichten.tx

?>
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>News anzeigen</title>
    <link href="shared/styles.css" rel="stylesheet" type="te
</head>

<body>
```

```
16      <h1>Die aktuellen Nachrichten</h1>
17
18      <?php foreach ($nachrichten as $n): ?>
19          <h2><?php echo $n['titel']; ?></h2>
20
21          <p class="inhalt"><?php echo $n['inhalt']; ?></p>
22          <p class="autor">
23              geschrieben von <?php echo $n['autor']; ?>
24              am <?php echo $n['datum']; ?>
25          </p>
26      <?php endforeach; ?>
27  </body>
28
29  </html>
```

news_anzeigen.php (Version 3)

Codebeispiel 4 *news_anzeigen.php (Version 3)*

Betrachten Sie die neueste Version unserer Datei. Was hat sich verändert? Es steht noch immer viel PHP-Code in dieser Datei.

Der erste Unterschied ist nur eine Kleinigkeit. Jeglicher PHP-Code, der nicht nur Text ausgibt, wurde an den Anfang der Datei verschoben. Nach diesem PHP-Block stehen außer dem HTML nur noch `echo`-Anweisungen und das `foreach`. Auf diese Weise finden Sie alles, was in Ihrem PHP-Code wirklich **etwas tut**, an einer Stelle. Sie müssen nicht suchen.

> Versuchen Sie die komplette Logik Ihres Programms, also den Teil, der wirklich arbeiten muss, in einem Block am Anfang der Datei zusammenzufassen. Im HTML-Bereich sollte nur noch Text ausgegeben werden.

Der zweite Unterschied ist, dass **kein HTML-Element mehr innerhalb von PHP-Tags** steht. Die `echo`-Anweisungen geben nur noch die wirklich dynamischen Inhalte aus, nicht mehr das statische HTML. Nun müssen Sie Ihrem Designer nur noch sagen, dass er auf keinen Fall etwas verändern darf, das zwischen dem **öffnenden** und dem **schließenden PHP-Tag** steht. Den kompletten Rest des Dokuments darf er nach Belieben anpassen.

Mit dieser Art, PHP und HTML zu mischen, haben Sie zwar unter Umständen mehr PHP-Tags, aber diese haben einen konsistenten Aufbau - was gut ist.

Personen, die kein PHP beherrschen, müssen sich nur von diesen Tags fernhalten und können so nicht versehentlich etwas an Ihrem Code zerstören.

Ein weiterer Vorteil ist, dass man im Vergleich zu vorher den HTML-Teil direkt lesen kann. Er ist nicht in `echo`-Anweisungen versteckt, sondern steht direkt in der Datei, so wie es ein Web-Designer gewohnt ist. Ein Bonus ist, dass nun auch das ***Syntax-Highlighting*** Ihres Editors den HTML-Teil wieder korrekt anzeigt.

> Achten Sie darauf, dass keine HTML-Elemente innerhalb von PHP-Tags stehen.

2.2.2 Lesbaren Code schreiben

Im letzten Abschnitt haben Sie gelernt, wie Sie Ihren Web-Designer glücklich machen können. Die hierbei vermittelten Regeln kommen jedoch natürlich auch Ihnen als Programmierer zu Gute. Doch auch direkt im PHP-Code können Sie dafür sorgen, dass Sie sich leichter zurechtfinden.

Eine der einfachsten und wichtigsten Regeln ist:

> Schreiben Sie immer Code, den Sie auch noch nach mehreren Wochen Pause sofort wieder verstehen.

Das klingt einleuchtend, aber was genau bedeutet es? Sie haben bestimmt schon öfter ein schwieriges Problem in PHP gelöst. Sie haben stundenlang an ein paar Zeilen Code herumgeschrieben, bis diese endlich das getan haben, was sie tun sollten. Dann haben Sie erschöpft, aber glücklich erst mal Pause oder sogar Feierabend gemacht. Einige Zeit später mussten Sie noch einmal an demselben Code-Abschnitt Änderungen vornehmen - und Sie hatten keine Ahnung mehr, was genau der Code an dieser Stelle tut.

Statt sofort Feierabend zu machen, hätten Sie besser noch etwas Zeit investiert, um Ihren Code lesbar zu gestalten. Gerade wenn man viel herumexperimentiert hat, ist der Code oft von schlechter Qualität:

- Es sind noch Variablen definiert, die gar nicht mehr verwendet werden.
- Ganze Funktionen werden gar nicht mehr verwendet.
- Es werden gar keine Funktionen verwendet.
- Die Namen der Variablen und Funktionen sind nicht sehr aussagekräftig.
- Der Code ist nicht korrekt eingerückt.
- Sie haben an schwierigen Stellen keine Kommentare geschrieben.
- HTML wird mit `echo`-Anweisungen ausgegeben (siehe Abschnitt 2.2.1).
- Ganze Blöcke, die nichts mehr tun, stehen noch auskommentiert im Code.

Alle diese Punkte machen es auch erfahrenen Programmierern schwer, sich in diesen Abschnitt wieder einzuarbeiten und Änderungen durchzuführen. Sie selbst werden nach einiger Zeit Schwierigkeiten damit haben, den Sinn des Abschnitts wieder zu verstehen. Nehmen Sie sich noch ein paar Minuten Zeit und räumen Sie hinter sich auf - Sie werden es sich selbst danken.

2.2.3 Kapselung in Funktionen

Zu Beginn Ihrer Programmierer-Karriere haben Sie sicherlich öfter über den Sinn von Funktionen nachgedacht. Wahrscheinlich wurde Ihnen gesagt, sie dienten der Wiederverwendung von Code. Das stimmt auch, aber es kann sich unter Umständen lohnen, selbst dann eine PHP-Funktion zu schreiben, wenn diese nur ein einziges Mal verwendet wird. Denn Funktionen dienen auch dazu, komplexen Code vor Ihnen zu verstecken.

Sie als Programmierer sind in der Lage, einen Code-Abschnitt einer gewissen Komplexität und Größe zu verstehen. Je nach Erfahrung kann dieser Abschnitt größer oder kleiner sein, aber selbst die besten und erfahrensten Programmierer stoßen irgendwann an ihre Grenzen. Das Ziel ist es also nicht, möglichst komplizierten Code verstehen zu können, sondern, ihn zu vermeiden oder zumindest zu verstecken.

Beispiel

```
1 <?php
```

```php

$datei = 'nachrichten.txt';

if (file_exists($datei)) {
    $nachrichten = unserialize(file_get_contents($datei));
}

if (!isset($nachrichten) || !is_array($nachrichten)) {
    $nachrichten = array();
}

var_dump($nachrichten);
```

test.php (Version 1)

Codebeispiel 5 *test.php (Version 1)*

Beispiel

```php
<?php

function holeDaten($datei)
{
    if (file_exists($datei)) {
        $nachrichten = unserialize(file_get_contents($datei)
    }

    // Wenn die Datei nicht existiert oder das Unserialize fe
    if (!isset($nachrichten) || !is_array($nachrichten)) {
        $nachrichten = array();
    }

    return $nachrichten;
}
```

datei.inc.php

Codebeispiel 6 *datei.inc.php*

```php
<?php

require_once 'datei.inc.php';

$nachrichten = holeDaten('nachrichten.txt');
var_dump($nachrichten);
```

test.php (Version 2)

Codebeispiel 7 *test.php (Version 2)*

Vergleichen Sie die beiden Beispiele, wobei das zweite aus zwei Dateien besteht. Beide Beispiele bewirken das Gleiche. Sie prüfen, ob eine Datei

existiert. Sofern sie existiert, werden serialisierte Daten aus ihr ausgelesen und wieder in ein Array umgewandelt. Existiert die Datei nicht oder schlägt das Umwandeln fehl, so wird stattdessen ein leeres Array erzeugt. Durch diese doppelte Absicherung mittels `if`-Abfragen können Sie im nachfolgenden Code immer von einem Array in `$nachrichten` ausgehen und brauchen dies dort nicht mehr zu kontrollieren. Wenn Sie nach einiger Zeit wieder über das Beispiel in Version 1 stolpern, oder Sie es zum ersten Mal sehen, da es ein Kollege geschrieben hat, werden Sie nicht wissen, was genau hier passiert. Zumindest hätten Sie sich über einen Kommentar zum besseren Verständnis gefreut.

In Version 2 wird eine Funktion namens `holeDaten()` aufgerufen, die exakt das Gleiche tut wie der Block in Version 1. Zudem erläutert ein Kommentar die schwierigste Stelle. Nur müssen Sie nicht mehr rätseln: Aufgrund des Namens wissen Sie, dass diese Funktion Daten holt. Ihr Code in der Datei `test.php` ist sehr einfach und dazu noch kürzer. Sie werden auch Wochen später auf den ersten Blick sehen, was hier passiert. An dieser Stelle ist nämlich nur relevant, dass Daten geholt werden, und nicht, wie. Erst wenn Sie nicht die erwarteten Daten erhalten, wird das »wie« und somit der Inhalt der Funktion relevant.

> Wenn Sie ein kompliziertes Problem gelöst haben, verstecken Sie den Code in einer Funktion mit einem aussagekräftigen Namen. Selbst wenn Sie diese Funktion an keiner weiteren Stelle benötigen, haben Sie für sich selbst dokumentiert, was dort passiert.

2.2.4 Funktionen, die atomare Probleme lösen

Atome sind die kleinsten Teilchen, oder waren es zumindest einmal. Ein atomares Problem in der Programmierung ist eines, das sich nicht mehr weiter zerlegen lässt. Anders ausgedrückt bedeutet das also:

> Eine Funktion hat genau eine Verantwortlichkeit bzw. Aufgabe, d.h. sie löst genau ein Problem (**Single Responsibility Principle**, siehe https://de.wikipedia.org/wiki/Single_Responsibility_Principle). Dieser Ausdruck wurde von *Robert C. Martin* eingeführt und bezeichnet eine Design-Richtlinie in der Architektur von Software. SRP kann man für

Klassen, Methoden und Funktionen gleichermaßen anwenden, wobei Sie vermutlich bisher nur Funktionen kennen. Wenn eine Funktion mehrere Aufgaben hat, sollte sie in mehrere Funktionen aufgeteilt werden.

Was genau ist aber eine **einzelne Aufgabe**? Lassen Sie es mich anhand eines Beispiels verdeutlichen:

Sie haben eine Textdatei *mitarbeiter.txt*, in der Ihre Mitarbeiter mit einigen Daten gespeichert sind. Sie wollen zwei Funktionen haben: eine, die alle Mitarbeiter, und eine, die nur alle Frauen zurückgibt.

Beispiel

```php
<?php

function holeAlleMitarbeiter()
{
    $mitarbeiter = unserialize(file_get_contents('mitarbeite

    return $mitarbeiter;
}

function holeWeiblicheMitarbeiter()
{
    $mitarbeiter = unserialize(file_get_contents('mitarbeite

    $frauen = array();
    foreach ($mitarbeiter as $m) {
        if ($m['geschlecht'] === 'w') {
            $frauen[] = $m;
        }
    }

    return $frauen;
}
```

mitarbeiter.inc.php

Codebeispiel 8 *mitarbeiter.inc.php*

Beide Funktionen sollten ihren Zweck erfüllen, auch wenn ich der Einfachheit halber auf die abgesicherte Vorgehensweise aus dem vorherigen Kapitel verzichtet habe.

Übung 2:

1. Überlegen Sie, welche der beiden Funktionen zwei Probleme löst.
2. Überlegen Sie nun, wie man den Code dieser Funktion lesbarer gestalten könnte. Sie müssen diese Idee jedoch nicht ausprogrammieren.

Leider hält sich unsere Funktion `holeWeiblicheMitarbeiter()` nicht an das Prinzip, dass nur atomare Probleme gelöst werden sollen. Denn sie holt erst alle Mitarbeiter aus der Datei und dann filtert sie alle Mitarbeiter nach den Frauen. Das sind zwei **verschiedene Aufgaben**. Hinzu kommt, dass für die erste Aufgabe bereits eine Funktion existiert.

Beispiel

```php
<?php

function holeAlleMitarbeiter()
{
    $mitarbeiter = unserialize(file_get_contents('mitarbeite

    return $mitarbeiter;
}

function filtereWeiblicheMitarbeiter($mitarbeiter)
{
    $frauen = array();
    foreach ($mitarbeiter as $m) {
        if ($m['geschlecht'] === 'w') {
            $frauen[] = $m;
        }
    }

    return $frauen;
}
```

funktionen.inc.php

Codebeispiel 9 *funktionen.inc.php*

```php
<?php

require_once 'funktionen.inc.php';

```

```php
$mitarbeiter = holeAlleMitarbeiter();
$frauen = filtereWeiblicheMitarbeiter($mitarbeiter);

?>
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Weibliche Mitarbeiter</title>
</head>

<body>
    <ul>
        <?php foreach ($frauen as $f): ?>
            <li>
                <?php echo $f['vorname']; ?>
                <?php echo $f['name']; ?>
            </li>
        <?php endforeach; ?>
    </ul>
</body>

</html>
```

zeige_weibliche_mitarbeiter.php

Codebeispiel 10 *zeige_weibliche_mitarbeiter.php*

Jetzt löst die verbesserte zweite Funktion nur noch ein Problem, nämlich das Filtern von Mitarbeitern. Beachten Sie auch, dass die Funktion jetzt anders heißt. Sie holt jetzt keine weiblichen Mitarbeiter mehr, sie filtert sie nur noch.

Als Bonus können Sie die Funktion nun auch auf Mitarbeiter-Arrays anwenden, die nicht aus der Datei *mitarbeiter.txt* kommen. Diese Daten könnten aus einer Datenbank oder einer Benutzereingabe stammen - es macht keinen Unterschied. Ihr Code ist also auch flexibler geworden.

Ein deutliches Zeichen dafür, dass eine Funktion keine atomaren Probleme löst, ist ihre Länge. Es ist extrem selten, dass eine Funktion länger als eine halbe Bildschirm-Seite ist und wirklich nur kleinste Probleme bearbeitet. Es ist natürlich auch möglich, dass Sie einfach zu kompliziert denken und somit das **KISS-Prinzip** (engl. »Keep it simple and straightforward«, dt. »Gestalte es einfach und überschaubar«, siehe https://de.wikipedia.org/wiki/KISS-Prinzip) verletzen.

2.3 Zusammenfassung

Sie haben verschiedene Gründe für schlechte Code-Qualität kennengelernt und wissen, wie Sie durch Code-Kapselung in Funktionen und Beachtung des Single Responsibility Principle besseren Code erhalten.

3 Einführung in die objektorientierte Programmierung

In dieser Lektion lernen Sie
• welche Vorteile objektorientierte Programmierung gegenüber prozeduraler bietet. • was Objekte und was Klassen sind. • wie Sie in PHP Instanzen von Klassen erzeugen. • wie Sie Attribute und Methoden einsetzen.

3.1Einleitung

Je größer PHP-Projekte werden, desto wichtiger ist es, sie sauber zu strukturieren. In Lektion 2 haben Sie bereits einige Techniken kennengelernt, um sich das Leben in dieser Hinsicht etwas leichter zu machen. Mit der objektorientierten Programmierung möchte ich Ihnen ein weiteres Hilfsmittel vorstellen, mit dem Sie klar strukturierten, besser wiederverwendbaren und besser wartbaren Code erzeugen können.

3.2 Was sind Objekte?

Dreh- und Angelpunkt der objektorientierten Programmierung (kurz OOP) bilden selbstverständlich die sogenannten Objekte. Doch was sind Objekte? In der realen Welt bezeichnet man oftmals greifbare Gegenstände (keine Lebewesen, obwohl Tiere rechtlich gesehen wie Sachen behandelt werden) als ***Objekt***. Jeder von uns hat tagtäglich mit solchen Objekten zu tun, beispielsweise mit **dem neuen Smartphone** nach dem Aufstehen, **dem verbogenen Schlüssel** beim Verlassen der Wohnung und **der Monatskarte des Fahrgasts gegenüber** bei der Kontrolle in der U-Bahn.

Es ist möglich, sich über diese Objekte zu unterhalten, ohne konkrete Details

nennen zu müssen. Man spricht dann nicht über bestimmte Objekte, sondern über Objektgruppen, wie beispielsweise Smartphones, Schlüssel oder Fahrkarten. Für jedes Objekt lässt sich immer mindestens eine Gruppe finden, in die man es einsortieren kann.

Jedes Objekt verfügt zudem über eine bestimmte Anzahl von Eigenschaften, die bei allen Objekten einer Gruppe vorhanden sind. Mein Smartphone hat beispielsweise eine **Displaygröße** von 5 Zoll, die **Farbe** schwarz und ein **Herstellungsdatum** aus dem letzten Monat. Der besagte Schlüssel wurde aus einem metallischen **Material** hergestellt, ist vom **Typ** her vermutlich ein Bohrmuldenschlüssel und sein **Zustand** ist ziemlich verbogen. Die Fahrkarte hat eine **Gültigkeitsdauer** von einem Monat, wurde für einen bestimmten **Fahrgast** ausgestellt und hat dummerweise den **Status** abgelaufen, da sie vom Vormonat ist.

Auch wenn zwei Objekte zur gleichen Objektgruppe gehören, so sind sie doch nur ähnlich und nie 100%ig identisch. Nehmen wir als Beispiel das neue Smartphone und einen Bekannten, der sich das gleiche Modell aus der gleichen Charge gekauft hat. Die Eigenschaften Displaygröße, Farbe und Herstellungsdatum sind bei beiden Smartphones vorhanden, jedoch nicht zwingend mit 100%ig übereinstimmenden Werten. Es lässt sich zudem bei einem realen Objekt immer mindestens eine Eigenschaft finden, deren Wert auf jeden Fall eine Unterscheidung von zwei Objekten ermöglicht. Sei es beispielsweise der Besitzer, die IMEI (International Mobile Equipment Identity), die Seriennummer oder die augenblickliche Position.

Fassen wir noch einmal kurz unsere Erkenntnisse zusammen. Ein Objekt in der realen Welt:

- ist meist ein greifbarer Gegenstand, sofern er nicht gerade entwendet wurde.
- gehört zu mindestens einer Gruppe, die eine Kategorisierung des Objekts zulässt.
- hat mindestens eine Eigenschaft, für die man einen Wert angeben kann.
- ist immer von einem anderen Objekt der gleichen Gruppe unterscheidbar (beispielsweise mittels seiner Position), auch wenn dies in der Praxis

durchaus schwer fallen mag.

Das Beispiel eines Sparbuch-Objektes

Kommen wir nun zu einem etwas detaillierteren Beispiel eines Gegenstandes und versuchen diesen mit PHP zu beschreiben. Beim Aufräumen fiel mir neulich ein altes Sparbuch in die Hände. Es stammte aus der Zeit der Euroumstellung und war tatsächlich noch aus Papier. Betrachten wir die Eigenschaften (z. B. das Guthaben) dieses Objektes einmal genauer, wobei ich auf Eigenschaften wie Farbe und Material bewusst verzichtet habe. Was würden Sie ohne Objektorientierung machen, um die Eigenschaften und ihre Werte mittels Code abzubilden?

Beispiel

```php
<?php

$sparbuch = array();
$sparbuch['glaeubiger'] = 'Jan Teriete';
$sparbuch['kontonummer'] = 'geheim';
$sparbuch['guthaben'] = 100;
$sparbuch['waehrung'] = 'Euro';

var_dump($sparbuch);
```

sparbuch.php

Codebeispiel 11 *sparbuch.php*

Richtig, wie in PHP üblich würden Sie ein assoziatives Array nutzen, bei dem die Eigenschaften als Schlüssel verwendet werden. Was tun Sie aber nun im PHP-Code, wenn Sie 98,19 Euro »einzahlen«?

Beispiel

```php
$sparbuch['guthaben'] += 98.19;
```

Einzahlung vornehmen (Version 1)

Codebeispiel 12 *Einzahlung vornehmen (Version 1)*

Richtig, Sie berechnen die Summe aus dem bisherigen Guthaben und der aktuellen Einzahlung. Merken Sie, wie künstlich dieser Vorgang ist? Um eine

Einzahlung vorzunehmen, müssen Sie einen kombinierten Summen-Operator auf ein Element eines Arrays anwenden. Dieser Transfer von der realen Welt hin zu PHP erfordert von unserem Gehirn Arbeit, die nicht sein müsste.

Wäre es nicht besser, dem Sparbuch direkt zu »sagen«, dass Sie eine Einzahlung vornehmen möchten?

Beispiel

```
$sparbuch->macheEinzahlung(98.19);
```

Einzahlung vornehmen (Version 2)

Codebeispiel 13 *Einzahlung vornehmen (Version 2)*

Die objektorientierte Programmierung versucht, die Sprache der Programmierung so weit wie möglich an unsere menschliche Sprache und damit Denkweise anzunähern. Der PHP-Code wird hierdurch intuitiver. Natürlich haben Sie aber immer noch PHP-Code vor sich und dürfen keine Wunder erwarten. Im Hintergrund würde deshalb bei der Einzahlung immer noch die Summe berechnet. Der Code wäre jedoch lesbarer, da diese Berechnung vor uns »versteckt« ist.

3.2.1 Objekte in PHP

Was sind nun aber Objekte in PHP? Aus Sicht der Sprache sind sie erst einmal nur ein weiterer Datentyp, wie Strings oder Arrays. Wenn Sie auf einem Objekt die Funktion `gettype()` aufrufen, erhalten Sie den Datentyp `object` (dt. Objekt) zurück.

Beispiel

```
1 <?php
2
3 $sparbuch = new stdClass();
4 echo gettype($sparbuch); //Diese Zeile gibt "object" zurueck
```

erstes_objekt.php (Version 1)

Codebeispiel 14 *erstes_objekt.php (Version 1)*

Was ist hier passiert? In Zeile 3 wurde ein neues Objekt erzeugt und der Variablen `$sparbuch` zugewiesen. Auch diese Variable `$sparbuch` selbst kann übrigens als Objekt bezeichnet werden, da sie ein Objekt als Wert enthält. Genau wie Strings und Arrays liegen auch Objekte in Variablen, nur die Art, wie sie erzeugt werden, unterscheidet sich.

Beispiel

```
1  <?php
2
3  $text = 'Hallo Welt';
4  var_dump($text);
5
6  $nummern = array(1, 2, 3, 4, 5);
7  var_dump($nummern);
8
9  $sparbuch = new stdClass();
10 var_dump($sparbuch);
```

erstes_objekt.php (Version 2)

***Codebeispiel 15** erstes_objekt.php (Version 2)*

Ein Objekt erzeugen Sie im Gegensatz zu allen bisherigen Datentypen mit dem Schlüsselwort `new` und der gewünschten Objektgruppe. Man bezeichnet diesen Vorgang auch als »***Instanz*** einer Klasse erzeugen« (zum Begriff der Klasse kommen wir gleich). Die Gruppe war in diesem Fall `stdClass`, also zu Deutsch `StandardKlasse`. Wir haben aus dieser Gruppe ein Standardobjekt erzeugt, wie wir in den Zeilen zuvor einen String und ein Array erzeugt haben.

Da dieses Objekt in einer Variablen namens `$sparbuch` abgelegt wurde, wird es sich wohl weiterhin um mein altes Sparbuch handeln, doch Genaueres können wir noch nicht sagen, da es wirklich nur ein PHP-Objekt ist. Es hat keine Eigenschaften und tut auch noch nichts.

Beispiel

```
1 <?php
2
3 $sparbuch1 = new stdClass();
4 var_dump($sparbuch1);
5
6 $sparbuch2 = new stdClass();
```

```
7 var_dump($sparbuch2);
```

erstes_objekt.php (Version 3)

Codebeispiel 16 *erstes_objekt.php (Version 3)*

Erzeugt man zwei Objekte der gleichen Gruppe und vergleicht die Ausgabe von `var_dump()`, so fällt einem sofort die unterschiedliche Angabe hinter der Raute auf. Für das Objekt `$sparbuch1` wird `object(stdClass)#1` und für das Objekt `$sparbuch2` wird `object(stdClass)#2` ausgegeben. Es findet also eine Art interne Nummerierung der Objekte durch PHP statt.

Beispiel

```
1  <?php
2
3  $sparbuch1 = new stdClass();
4  $sparbuch2 = new stdClass();
5
6  $sparbuch3 = $sparbuch2;
7
8  ?>
9  <p>1/2 Gleich? <?php echo $sparbuch1 == $sparbuch2; //ja ?><
10 <p>1/2 Identisch? <?php echo $sparbuch1 === $sparbuch2; //ne
11 <p>2/3 Identisch? <?php echo $sparbuch2 === $sparbuch3; //ja
```

erstes_objekt.php (Version 4)

Codebeispiel 17 *erstes_objekt.php (Version 4)*

Wenn Sie die beiden Objekte `$sparbuch1` und `$sparbuch2` mittels Gleichheitsoperator (==) vergleichen, so sind diese gleich. Sie gehören zur gleichen Gruppe und unterscheiden sich auch nicht bei den Werten der Eigenschaften. Um genau zu sein, gibt es ja noch gar keine Eigenschaften. Zu diesen kommen wir jedoch bald.

Vergleichen Sie hingegen `$sparbuch1` und `$sparbuch2` mittels Identitätsoperator (===), so erfahren Sie, dass diese Objekte nicht identisch sind, was die unterschiedliche Nummerierung im vorherigen Beispiel ja bereits suggeriert hat.

Befüllen Sie jedoch wie in Zeile 6 eine Variable mit dem Wert einer anderen Variablen, die ein Objekt enthält, so sind die Objekte in beiden Variablen identisch. Bei der Nutzung des Zuweisungsoperators (=) behält ein Objekt also seine Identität (engl. ***Identity***).[1]

1 Dies ist seit PHP 5 für Objekte der Fall und gilt ebenfalls bei der Übergabe als Parameter.

Fassen wir noch einmal zusammen. Objekte in PHP:

- haben den Datentyp `object`.
- werden mit dem Schlüsselwort `new` erzeugt.
- können aus der allgemeinen Objektgruppe `stdClass` instanziiert werden.
- sind gleich, sofern sich Objektgruppe und Eigenschaften (bzw. genauer deren Werte) nicht unterscheiden.
- sind identisch, sofern sie die gleiche Identität haben.

3.2.2 Klassen

Sehen Sie sich doch bitte das folgende Stück Code an:

Beispiel

```php
<?php

$sparbuch = new stdClass();
var_dump($sparbuch);

$tagesgeldKonto = new stdClass();
var_dump($tagesgeldKonto);

$giroKonto = new stdClass();
var_dump($giroKonto);
```

konto_objekte.php (Version 1)

Codebeispiel 18 *konto_objekte.php (Version 1)*

In allen drei Fällen handelt es sich um Objekte im Sinne von PHP, obwohl man lediglich das Sparbuch tatsächlich in den Händen halten kann. Auch die ursprünglich für das Sparbuch festgelegten Eigenschaften Gläubiger, Kontonummer, Guthaben und Währung sind überall anwendbar. Die Objekte gehören also alle zur gleichen Gruppe **Konto**. Das Spar(kassen)buch weist die Geldbewegungen (Einzahlungen, Auszahlungen, Zinsen usw.) eines Sparkontos aus und ist somit ein greifbares Objekt, welches das virtuelle Objekt Sparkonto in der Realität repräsentiert.

Umgangssprachlich könnte man sagen: »`$sparbuch` ist ein Konto.« Genau dieses »ist ein« ist es, was man unter einer ***Klasse*** versteht (also die Gruppe oder die Art eines Objektes). Im Deutschen würde man zwar nicht als Erstes auf den Begriff Klasse kommen, aber er leitet sich vom englischen ***class*** ab und hat sich auch im Deutschen in dieser Bedeutung durchgesetzt.

Klassen sind also dazu da, um Objekte in bestimmte Gruppen einzusortieren. Alternativ könnte man auch sagen, dass Klassen als Baupläne für die Erzeugung von Objekten der gleichen Art dienen. Doch dazu gleich mehr.

Eigene Klassen werden in PHP mit dem Schlüsselwort `class` definiert und unverändert mit dem Schlüsselwort `new` instanziiert.

Beispiel

```php
<?php

class Konto
{
}

$sparbuch = new Konto();
var_dump($sparbuch);

$tagesgeldKonto = new Konto();
var_dump($tagesgeldKonto);

$giroKonto = new Konto();
var_dump($giroKonto);
```

konto_objekte.php (Version 2)

Codebeispiel 19 *konto_objekte.php (Version 2)*

Klassennamen werden immer in der Einzahl geschrieben, da sie die Art eines Objektes wiedergeben. Sie würden ja auch sagen »Das da ist ein Auto« und nicht »Das da ist Autos«. In PHP und den meisten anderen Programmiersprachen hat es sich eingebürgert, den ersten Buchstaben eines (eigenen) Klassennamens großzuschreiben.

Eine einfache Methode, eine PHP-Datei kürzer und damit übersichtlicher zu bekommen, ist, einen Teil des Codes in eine andere Datei auszulagern. Sie haben sicherlich schon oft Funktionen in eine andere Datei geschrieben und per

`include(_once)` oder `require(_once)` eingebunden. Für jede Klasse legen wir deswegen eine eigene Datei an.

Beispiel

```php
<?php

class Konto
{
}
```

konto.php - Klassendefinition

Codebeispiel 20 *konto.php - Klassendefinition*

```php
<?php

require_once 'konto.php';

$sparbuch = new Konto();
var_dump($sparbuch);

$tagesgeldKonto = new Konto();
var_dump($tagesgeldKonto);

$giroKonto = new Konto();
var_dump($giroKonto);
```

konto_objekte.php (Version 3)

Codebeispiel 21 *konto_objekte.php (Version 3)*

Nun können wir direkt aus dieser eigenen Klasse `Konto` unsere Objekte instanziieren, ohne auf `stdClass` ausweichen zu müssen, und haben durch die Auslagerung eine Möglichkeit zur Wiederverwendung geschaffen.

Fassen wir die Erkenntnisse dieses Abschnitts noch einmal zusammen:

- Objekte in PHP beschreiben nicht zwingend einen greifbaren Gegenstand der realen Welt.
- Klasse ist der Fachbegriff für die Gruppe bzw. den Bauplan eines Objektes.
- Eigene Klassen werden mit dem Schlüsselwort `class` definiert.
- Klassennamen werden immer in der Einzahl geschrieben und es hat sich eingebürgert, den ersten Buchstaben eines Klassennamens

großzuschreiben.

- Es macht Sinn, die Definition jeder einzelnen Klasse in eine eigene Datei auszulagern, um eine optimale Wiederverwendbarkeit zu ermöglichen.

Im Folgenden wird in einigen Beispielen der Code einer Klasse (die sogenannte **Klassendefinition**) und deren Verwendung (z. B. die Instanziierung) in einer Datei zusammengefasst und nicht ausgelagert. Dies geschieht jedoch nur, um die Beispiele kurz zu halten. Ab Lektion 9 gehen wir endgültig zur besseren Strukturierung mit Auslagerung über.

Übung 3:

1. Versuchen Sie für folgende Objekte geeignete Klassennamen zu finden: München, rot, Stuttgart, Ingwer, Hamburg, grün, Pfeffer, Katze, Berlin, gelb, Salami, Hund, Schinken, Kümmel.
2. Erzeugen Sie für eine dieser Klassen eine neue PHP-Datei und programmieren Sie die Klasse aus.
3. Schreiben Sie ein PHP-Skript *index.php*, welches eine Instanz der Klasse erzeugt. Kontrollieren Sie das Objekt mittels `var_dump()`.

3.3 Attribute

3.3.1 Attribute verändern

Sie fragen sich bestimmt langsam, was man mit diesen Objekten und Klassen anfangen kann. Nun: Zum Beispiel können Sie ihnen endlich Eigenschaften zuweisen. Mein Sparbuch hat beispielsweise ein Guthaben von 100 (Euro) und den Gläubiger »Jan Teriete«:

Beispiel

```
1 <?php
```

```
2
3 require_once 'konto.php';
4
5 $sparbuch = new Konto();
6 $sparbuch->glaeubiger = 'Jan Teriete';
7 $sparbuch->guthaben = 100;
8
9 var_dump($sparbuch);
```

konto_attribute.php (Version 1)

***Codebeispiel 22** konto_attribute.php (Version 1)*

Mit dem Pfeil-Operator können Sie auf Eigenschaften eines Objektes Einfluss nehmen. Er trennt den Objekt-Namen bzw. genauer gesagt die Variable, in der sich unser Objekt befindet, von der Eigenschaft. In diesem Fall haben wir also beim Objekt `$sparbuch` die Eigenschaft `glaeubiger` auf `'Jan Teriete'` und die Eigenschaft `guthaben` auf `100` gesetzt. Ansonsten können Sie Eigenschaften weitestgehend wie Variablen verwenden. Wenn Sie einer Eigenschaft einen Wert zuweisen wollen, geschieht das mit dem Zuweisungsoperator `=`.

Da sich der Fachbegriff für Eigenschaften auch in diesem Fall von einem englischen Wort (nämlich ***attribute***) ableitet, lautet die offizielle Bezeichnung auch im Deutschen ***Attribut***.

3.3.2 Attribute auslesen

Ein Attribut eines Objektes wieder auszulesen, funktioniert wie bei einer Variablen:

Beispiel

```
1 <?php
2
3 require_once 'konto.php';
4
5 $sparbuch = new Konto();
6 $sparbuch->glaeubiger = 'Jan Teriete';
7 $sparbuch->guthaben = 100;
8
9 echo $sparbuch->guthaben; //gibt 100 aus
10
11 $kontostand = $sparbuch->guthaben; //auch Zuweisungen funkti
```

```
12 echo $kontostand;
```
konto_attribute.php (Version 2)

Codebeispiel 23 *konto_attribute.php (Version 2)*

Allerdings müssen Sie auch in diesem Fall das Objekt `$sparbuch` und den Pfeil-Operator vor dem Namen des Attributs angeben.

> Die Art, wie wir hier die Werte von Attributen auslesen und verändern, wird in Lektion 4 durch eine bessere Methode ersetzt.

3.3.3 Attribute in Klassen definieren

Wie Sie gesehen haben, ist es zwar möglich, einem Objekt nach Belieben neue Attribute hinzuzufügen, aber damit ignorieren Sie einen der größten Vorteile der OOP, nämlich Daten sauber und übersichtlich zu strukturieren. Oberflächlich betrachtet gibt es derzeit nämlich (noch) keine gravierenden Unterschiede zwischen einem Objekt und einem assoziativen Array, doch dies wird sich gleich ändern.

Um eine solche saubere Strukturierung zu erreichen, sollten Sie schon in der Klassendefinition angeben, welche Attribute ein Objekt bei der Instanziierung haben soll. Das Schlüsselwort, das ein Attribut hierbei kennzeichnet, ist `public`:

Beispiel

```
 1 <?php
 2
 3 class Konto
 4 {
 5     public $glaeubiger;
 6     public $kontoNummer;
 7     public $guthaben;
 8     public $waehrung;
 9 }
10
11 $sparbuch = new Konto();
12 var_dump($sparbuch);
```
konto.php (Version 1)

Codebeispiel 24 *konto.php (Version 1)*

Auf diese Weise definieren Sie alle Eigenschaften zentral im Bauplan einer Objektgruppe und können außerdem bereits vor der Instanziierung Standardwerte festlegen, die für alle Objekte dieser Klasse gelten:

Beispiel

```php
<?php

class Konto
{
    public $glaeubiger;
    public $kontoNummer;
    public $guthaben;
    public $waehrung = 'Euro';
}

$sparbuch = new Konto();
echo $sparbuch->waehrung; //gibt 'Euro' aus
```

konto.php (Version 2)

Codebeispiel 25 *konto.php (Version 2)*

Der Standardwert kann natürlich später in einem Objekt jederzeit geändert werden. Er dient einfach dazu, einen sinnvollen Startwert für ein Attribut festzulegen.

Beispiel

```php
<?php

class Konto
{
    public $glaeubiger = '';
    public $kontoNummer = '';
    public $guthaben = 0;
    public $waehrung = 'Euro';
}

$sparbuch = new Konto();

echo gettype($sparbuch->guthaben); //gibt 'integer' aus

echo $sparbuch->kontoNummer; //gibt '' aus

$sparbuch->glaeubiger = 'Jan Teriete';
echo $sparbuch->glaeubiger; //gibt 'Jan Teriete' aus
```

konto.php (Version 3)

Codebeispiel 26 *konto.php (Version 3)*

Es gilt als guter Stil, auch Attribute ohne Standardwert zumindest mit dem korrekten Datentyp zu initialisieren. So sieht jeder, der den Code überarbeitet, welcher Datentyp hier erwartet wird. Diese empfehlenswerte Vorgehensweise bezeichnet man übrigens als »sich selbst dokumentierenden Code«.

3.4 Methoden

3.4.1 Grundlagen

Bis jetzt sind unsere Objekte reine Datenbehälter. Sie »tun« noch nichts, was wir nun ändern werden. In der objektorientierten Programmierung können Sie nämlich eine Funktion in eine Klasse einbauen. Jedes Objekt, das aus der Klasse erzeugt wird, hat diese Funktion.

Beispiel

```php
<?php

class Konto
{
    public $glaeubiger = '';
    public $kontoNummer = '';
    public $guthaben = 0;
    public $waehrung = 'Euro';

    function holeHallo()
    {
        return 'Hallo Welt';
    }
}

$sparbuch = new Konto();

echo holeHallo(); //Fehler, da die Funktion nicht gefunden wi
```

konto.php - Fehler (Version 4)

Codebeispiel 27 *konto.php - Fehler (Version 4)*

Wir haben in der Klasse `Konto` die Funktion `holeHallo()` angelegt. Da `$sparbuch` eine Instanz der Klasse ist, verfügt auch das Objekt über diese Funktion. **Trotzdem scheitert der Aufruf der Funktion.**

Beispiel

```php
<?php

class Konto
{
    public $glaeubiger = '';
    public $kontoNummer = '';
    public $guthaben = 0;
    public $waehrung = 'Euro';

    function holeHallo()
    {
        return 'Hallo Welt';
    }
}

$sparbuch = new Konto();

echo $sparbuch->holeHallo(); //gibt 'Hallo Welt' aus.
```

konto.php (Version 4b)

***Codebeispiel 28** konto.php (Version 4b)*

Die neue Funktion ist fest an Instanzen der Klasse `Konto` gebunden, Sie können sie nicht ohne ein `Konto`-Objekt aufrufen. Auch hier müssen wir hinter dem Objekt `$sparbuch` (genau wie beim Zugriff auf Attribute) den Pfeil-Operator verwenden, um die Funktion anzusprechen.

Jetzt werden Sie sich langsam fragen, warum hier ständig von Funktionen die Rede ist, obwohl das Kapitel ***Methoden*** heißt. Ganz einfach: Dieser Begriff ist nur ein anderes Wort für eine Funktion, die an eine Klasse gebunden ist.

> Methoden sind Funktionen, die an Klassen gebunden sind. Methoden können auf Objekten aufgerufen werden, die aus einer Klasse erzeugt wurden, die diese Methode enthält.

3.4.2 Vorteil von Methoden

Welchen Vorteil hat es, Methoden anstatt normaler Funktionen zu verwenden? Ein Hauptargument für die Verwendung von Methoden kennen Sie bereits: Mit der Hilfe von Methoden können Sie Ihren Code besser strukturieren und übersichtlicher halten.

Sie haben sicherlich gelernt, dass man möglichst viel mit Funktionen arbeiten sollte. Wenn Sie sich daran halten und Ihre Projekte eine bestimmte Größe überschreiten, haben Sie eine ziemliche Menge an Funktionen, was einige Probleme mit sich bringt:

- Ihre Standard-Datei für Funktionen, z. B. *funktionen.inc.php*, wird in diesem Fall so groß, dass Sie sich nicht mehr auskennen. Zunächst beginnen Sie vielleicht damit, Funktionen in der Reihenfolge zu verschieben, um sie thematisch zu gruppieren. Wenn dies nicht mehr hilft, legen Sie mehrere Dateien mit Funktionen an. Dafür fragen Sie sich jetzt, in welcher Datei Sie eine bestimmte Funktion abgelegt haben.
- Sie beginnen damit, sich kreative Namen für Ihre Funktionen auszudenken, da Sie die kurzen, prägnanten Namen alle schon verwendet haben.
- Die Anzahl der Parameter, die Sie an Funktionen übergeben müssen, steigt mit der Größe Ihres Projektes an. Einige Ihrer Funktionen haben bereits mehr als fünf Parameter.

Zugegeben, ohne Funktionen wäre der Code wahrscheinlich gar nicht mehr wartbar, aber auch so verbringen Sie unnötig viel Zeit damit, Ihren Code übersichtlich zu halten. Auch in diesem Fall gilt: Die objektorientierte Programmierung ist kein Allheilmittel. Aber sie bietet Ihnen einige elegante Möglichkeiten, wie Sie Ihren Code wartbar halten:

- Da Methoden an Klassen gebunden werden, sind sie automatisch gruppiert. Sie wissen in den meisten Fällen sofort, wo Sie eine bestimmte Methode definiert haben.
- Sie können dieselben Methodennamen mehrfach verwenden, solange die Methoden in verschiedenen Klassen definiert werden.
- Methoden können direkt auf Attribute von Objekten zugreifen. Sie müssen

nicht wie bei Funktionen jeden Wert als Parameter übergeben.

Lassen Sie uns ein Beispiel für den zweiten Punkt ansehen. Sie haben einen Online-Shop, der Bücher und CDs verkauft. Für beide Produktarten benötigen Sie ähnliche Funktionen, z. B. für die Berechnung des Preises (inklusive Rabatte, verschiedene Mehrwertsteuersätze usw.). Sie werden also wahrscheinlich zwei Funktionen anlegen, die Sie `berechnePreisBuch()` und `berechnePreisCd()` nennen. Da Sie später auch noch DVDs und Kajaks verkaufen, kommen auch noch `berechnePreisDvd()` und `berechnePreisKajak()` hinzu.

Wenn Sie objektorientiert arbeiten, haben Sie für alle Produktgruppen Klassen, also `Buch`, `CD`, `DVD` und `Kajak`. Wenn jede dieser Klassen eine Methode zum Berechnen des Preises haben soll, dann kann diese einfach `berechnePreis()` heißen. Da die Methode zu einer Klasse gehört, müssen Sie gar nicht mehr sagen, welche Art von Preis Sie berechnen. Die Methode `berechnePreis()` der Klasse `Buch` berechnet den Preis von Büchern. Das ist offensichtlich, ohne dass die Methode `berechnePreisBuch()` heißt.

Jedes Objekt einer Klasse übernimmt bei der Instanziierung sämtliche Attribute und Methoden dieser Klasse. Die Klasse ist somit eine Art Prototyp bzw. ein Bauplan, in dem Sie alles definieren, was später benötigt wird. Mit Hilfe dieses Bauplans können dann individualisierbare Varianten (die sogenannten Objekte) instanziiert werden.

Die Attribute, etwa der Preis, können und werden sich zukünftig von Instanz zu Instanz unterscheiden. Beachten Sie, dass der Begriff Attribut nicht nur die Eigenschaft bezeichnen kann, sondern auch den hierin abgelegten Wert. Hier meine ich den Wert.

Außerdem können zwei verschiedene Klassen identische Bezeichner für Attribute und Methoden verwenden.Sie können zwei Methoden, die das Gleiche tun, also gleich benennen, sofern diese in unterschiedlichen Klassen definiert werden. Durch ihre Bindung an die Klasse bleiben sie trotzdem eindeutig.

3.4.3 Die Variable $this

Ich hatte im letzten Abschnitt behauptet, Sie könnten in Methoden direkt auf Attribute eines Objektes zugreifen. Schauen wir uns als Beispiel ein `Benutzer`-Objekt an. In PHP können also auch Lebewesen Objekte sein. Allgemein ist in PHP der Begriff ziemlich weit gefächert; sofern man eine Klasse dafür definieren kann, kann man es auch instanziieren und somit ist es ein Objekt.

Beispiel

```php
<?php

class Benutzer
{
    public $name = '';
    public $passwort = '';
    public $gruppen = array('gaeste');

    function holeGruppen()
    {
        return $gruppen;
    }
}

$remolt = new Benutzer();
var_dump($remolt->holeGruppen());
```

benutzer.php - Fehler (Version 1)

Codebeispiel 29 *benutzer.php - Fehler (Version 1)*

Übung 4:

1. Überlegen Sie, was der Code aus dem letzten Beispiel als Ausgabe erzeugt.
2. Probieren Sie den Code aus. Lagen Sie richtig?
3. Überlegen Sie, warum es zu dieser Ausgabe gekommen ist.

Waren Sie überrascht? Eigentlich ist es klar, warum keine Gruppen angezeigt wurden, wenn Sie sich erinnern, was Sie über Funktionen wissen. Variablen außerhalb einer Funktion sind innerhalb der Funktion nicht sichtbar (und umgekehrt), sofern man von der Verwendung von `global` absieht. Das ist

allerdings ein PHP-Feature, das Sie besser nicht verwenden sollten. Da Methoden eigentlich auch nur Funktionen sind, verhalten sie sich an dieser Stelle genauso. Als in Zeile 11 in der Methode `holeGruppen()` auf die Variable `$gruppen` zugegriffen wurde, war es eine lokale Variable innerhalb der Funktion und nicht das Attribut `$gruppen`, das Sie eigentlich wollten.

Beispiel

```
 1 <?php
 2
 3 class Benutzer
 4 {
 5     public $name = '';
 6     public $passwort = '';
 7     public $gruppen = array('gaeste');
 8
 9     function holeGruppen()
10     {
11         //return $gruppen;
12     }
13 }
14
15 $remolt = new Benutzer();
16 var_dump($remolt->gruppen);
```

benutzer.php (Version 2)

***Codebeispiel 30** benutzer.php (Version 2)*

Wenn Sie keine lokale Variable meinen, sondern ein Attribut, müssen Sie das ausdrücklich sagen. Außerhalb von Klassen würden Sie das wie in Zeile 16 dargestellt beispielsweise mit `$remolt->gruppen` ausdrücken.

Wie aber sagen Sie in einer Klasse, dass Sie eine Instanz dieser Klasse meinen? Sie wissen ja vorher nicht einmal, in welcher Variable das Objekt abgelegt sein wird.

Beispiel

```
1 <?php
2
3 class Benutzer
4 {
5     public $name = '';
```

```
 6      public $passwort = '';
 7      public $gruppen = array('gaeste');
 8
 9      function holeGruppen()
10      {
11          return $remolt->gruppen;
12      }
13  }
14
15  $remolt = new Benutzer();
16  var_dump($remolt->holeGruppen());
```

benutzer.php - Fehler (Version 3)

Codebeispiel 31 *benutzer.php - Fehler (Version 3)*

In Zeile 11 versuchen Sie, auf ein Objekt `$remolt` zuzugreifen. Es gibt gleich mehrere Gründe, warum das nicht funktioniert:

- `$remolt` ist an dieser Stelle eine lokale Variable.
- `$remolt` wird erst viel später überhaupt erzeugt.
- Was wäre, wenn Sie das `Benutzer`-Objekt nicht in `$remolt`, sondern in `$schneider` abgelegt hätten?

Aus diesen Gründen gibt es in PHP eine spezielle Variable, die immer das Objekt enthält, in dem Sie sich gerade befinden, nämlich ***$this***. Wann immer Sie in einer Klasse Code schreiben, der auf das später erzeugte Objekt verweist, verwenden Sie einfach `$this` und PHP weiß, was Sie meinen. Das bedeutet übrigens auch, **Sie dürfen niemals selbst eine Variable mit diesem Namen erzeugen**.

Beispiel

```
 1  <?php
 2
 3  class Benutzer
 4  {
 5      public $name = '';
 6      public $passwort = '';
 7      public $gruppen = array('gaeste');
 8
 9      function holeGruppen()
10      {
11          return $this->gruppen;
```

```
12      }
13  }
14
15  $remolt = new Benutzer();
16  var_dump($remolt->holeGruppen());
```

benutzer.php (Version 4)

Codebeispiel 32 *benutzer.php (Version 4)*

Jetzt sagen Sie in Zeile 11, dass Sie auf das Attribut `$gruppen` in diesem (engl. this) Objekt zugreifen wollen.

> Wann immer Sie in einer Klasse auf ein Attribut oder eine Methode eines aus dieser Klasse instanziierten Objektes zugreifen wollen, müssen Sie die spezielle Variable `$this` verwenden. Also müssen Sie `$this->attribut` oder `$this->methode()` schreiben.

3.5 Namenskonventionen

Sie haben wahrscheinlich in den Beispielen schon gemerkt, dass in PHP einige Programmierrichtlinien (siehe auch Lektion 11) existieren, wie Namen geschrieben werden sollten. Sie müssen sich natürlich wie üblich nicht an diese Konventionen halten, aber es macht Sinn, wenn Ihr Code so aussieht wie der der anderen PHP-Programmierer.

3.5.1 Klassen

Klassen-Namen verwenden den ***UpperCamelCase***, beginnen also mit einem Großbuchstaben. Der Rest des Namens wird kleingeschrieben. Setzt sich der Name aus mehreren Wörtern zusammen, wird der erste Buchstabe jedes Wortes großgeschrieben.

Beispiel

- `class Person`

- `class Auto`
- `class AutoWerkstatt`

3.5.2 Attribute

Attribute verwenden den ***lowerCamelCase***, werden also kleingeschrieben. Setzt sich der Name aus mehreren Wörtern zusammen, wird der erste Buchstabe jedes Wortes ab dem zweiten großgeschrieben.

Beispiel

- `$name`
- `$passwort`
- `$firmenName`

3.5.3 Methoden

Für Methoden gelten dieselben Regeln wie für Attribute, allerdings sollte der Name immer mit einem Verb beginnen.

Beispiel

- `function speichere()`
- `function holeVorname()`
- `function testeLogin()`

3.6 Testen Sie Ihr Wissen

1. Was ist in der Objektorientierung ein Objekt?
2. Was ist eine Klasse?

3. Was sind Attribute?
4. Wie können Sie in PHP aus einer Klasse ein Objekt instanziieren?
5. Auf was bezieht sich die Variable `$this`?
6. Was unterscheidet Methoden von Funktionen?
7. Wie können Sie eine Methode eines Objektes aufrufen?

3.7 Aufgaben zur Selbstkontrolle

Übung 5:

Sehen Sie sich in Ihrer unmittelbaren Umgebung um und suchen Sie sich für die folgenden Aufgaben ein beliebiges Objekt der realen Welt aus. Es ist dabei tatsächlich vollkommen unwichtig, was Sie sich aussuchen, sei es eine andere Person, ein Computer, ein Tisch oder der Vogel auf der Fensterbank.

Übung 6:

Überlegen Sie sich, welche Eigenschaften Ihr Objekt besitzt. Wählen Sie allerdings nicht mehr als fünf aus, um sich später nicht zu viel Tipparbeit zuzumuten.

Übung 7:

Überlegen Sie, zu welcher Klasse Ihr Objekt gehören könnte.

Übung 8:

Programmieren Sie jetzt die Klasse als Datei aus. Vergeben Sie für alle Attribute sinnvolle Standardwerte.

Übung 9:

Schreiben Sie ein PHP-Skript *index.php*, welches eine Instanz der eben erstellten Klasse erzeugt und die Attribute anzeigt.

4 Getter- und Setter-Methoden

In dieser Lektion lernen Sie
• wie Sie Attribute von Objekten in Methoden kapseln können. • warum der direkte Zugriff auf Attribute gefährlich ist. • wie Sie Getter- und Setter-Methoden schreiben. • was das Schlüsselwort `protected` tut.

4.1Das Problem

Sie haben zwar bereits jetzt einige Vorteile der OOP kennengelernt, aber im Hinblick auf die Attribute verhält sich ein Objekt immer noch sehr wie ein Array. Sie können beliebig neue Attribute anlegen, ohne Kontrolle Werte und sogar Datentypen ändern und die einzelnen Attribute wieder löschen. Niemand hindert Sie zum Beispiel daran, in einem Attribut `$vorname` die Zahl `5` zu speichern.

So haben die Attribute alle Nachteile von Arrays ohne deren Vorteile, wie zum Beispiel die Verwendung der `foreach`-Schleife. Objekte erscheinen momentan also als armselige Arrays mit eingebauten Methoden (genau das waren sie auch tatsächlich in PHP 4).

4.2 Kapselung

Ein grundlegendes Prinzip der objektorientierten Programmierung ist die ***Kapselung***. Kapselung bedeutet, die tatsächliche Funktionalität eines Codeblocks in Funktionen oder in Methoden einer Klasse zu verbergen.

Der Vorteil von Kapselung ist, dass Sie in den Methoden den Code beliebig ändern können, ohne dass dies Auswirkungen auf den restlichen Code hat. So lange sich am Methodenaufruf und am Rückgabewert nichts ändert, können Sie den Inhalt der Methode jederzeit gegen besseren oder schnelleren Code

austauschen.

Von außen sieht die Klasse also weiterhin gleich aus, nur im Inneren ändert sich die Funktionalität. Man spricht hier auch vom ***Interface*** einer Klasse, was nichts anderes als die Sicht von außen meint. Solange das Interface stabil (also unverändert) bleibt, müssen Sie bei Änderungen an einer Methode den aufrufenden Code nicht ändern.

> Versuchen Sie immer, das Interface einer Klasse stabil zu halten. Jede Änderung an Methodenaufrufen zieht immer mehr Arbeit nach sich, je größer die Projekte werden.

4.3 Getter-Methoden

Bei der sogenannten ***Datenkapselung*** werden die Attribute nach außen hin versteckt und der Zugriff auf sie über Methoden ermöglicht. Beginnen wir hierbei mit einer Methode, die das Attribut `$vorname` zurückliefert.

Beispiel

```php
<?php

class Person
{
    public $vorname = '';
    public $nachname = '';

    function getVorname()
    {
        return $this->vorname;
    }
}

$schad = new Person();
$schad->vorname = 'Frank';
$schad->nachname = 'Schad';

echo $schad->vorname; // gibt 'Frank' aus
echo $schad->getVorname(); // gibt ebenfalls 'Frank' aus
```

person.php (Version 1)
Codebeispiel 33 *person.php (Version 1)*

Beide Varianten liefern dasselbe Ergebnis zurück, nämlich den Wert des Attributs `$vorname`.

Der Fall, dass Zugriffs-Methoden für Attribute geschrieben werden, kommt so häufig vor, dass es in der OOP sogar einen eigenen Namen dafür gibt. Diese Methoden nennt man ***Getter-Methoden*** oder einfach nur ***Getter***, da sie das Attribut bzw. dessen Wert »holen« (engl. get). Diese Methoden können eigentlich heißen, wie Sie es als Programmierer für richtig halten, doch es hat sich über die Jahre eine Namenskonvention eingebürgert.

> Der Bezeichner einer Methode, die den Wert eines Attributs zurückgibt, beginnt mit `get`, gefolgt vom Namen des Attributs. Es wird die übliche Namenskonvention für Methoden verwendet.

Beispiel

Wenn Ihre Klasse ein Attribut namens `$vorname` hat, so heißt die passende Getter-Methode `getVorname()`.

4.3.1 Vorteil von Getter-Methoden

Warum sollten Sie nun einen Getter verwenden, anstatt direkt auf das Attribut zuzugreifen? Schließlich führen beide Wege zum gleichen Ergebnis, nur ist mit Gettern der Zugriff auf das Attribut in einer Methode gekapselt.

Der Vorteil liegt darin, dass Sie in einer Methode noch eingreifen können, bevor das Attribut ausgeliefert wird. Sie könnten den Zugriff auf bestimmte Attribute generell sperren, oder das Attribut auch zuerst ein wenig umformatieren, bevor es zurückgegeben wird.

Beispiel

```
1 <?php
```

```
 2
 3 class Benutzer
 4 {
 5     public $name = '';
 6     public $passwort = '';
 7     public $gruppen = array();
 8
 9     function getPasswort()
10     {
11         return $this->passwort;
12     }
13 }
14
15 $administrator = new Benutzer();
16 $administrator->passwort = 'geheim';
17
18 echo $administrator->getPasswort(); //gibt 'geheim' aus
```

benutzer.php (Version 1)

Codebeispiel 34 *benutzer.php (Version 1)*

Es gilt als guter Stil, für jedes Attribut eine Getter-Methode zu schreiben, auch wenn diese Methode nur das Attribut zurückgibt. Der Vorteil ist: Wenn Sie später doch Umformatierungen benötigen, haben Sie bereits eine Methode, die überall verwendet wird. Hätten Sie im restlichen Code direkt auf das Attribut zugegriffen, müssten Sie alle Stellen suchen und durch den Methoden-Aufruf ersetzen. Falls das viele Stellen sind, haben Sie sich eine Menge unnötige Arbeit gemacht.

Sie könnten sogar intern den Namen des Attributs ändern, und so lange der Getter existiert, verändert sich der Zugriff auf die Objekte dieser Klasse nicht. Dieses Prinzip, dass Änderungen an einer Stelle des Codes möglichst wenige Änderungen an anderen Stellen verursachen, sollten Sie sich so weit wie möglich zu eigen machen. Wenn Sie regelmäßig nach kleinen Korrekturen Ihren kompletten Code nach Fehlern durchforsten müssen, sollten Sie die Struktur Ihres Codes grundsätzlich überdenken.

4.3.2 Ein Attribut »protected« machen

So wie die Klasse `Benutzer` momentan aussieht, kann jeder, der die Seite aufruft, das Passwort eines Benutzers auslesen. Wir wollen aber, dass dies für niemanden möglich ist.

Beispiel

```php
<?php

class Benutzer
{
    public $name = '';
    public $passwort = '';
    public $gruppen = array();

    function getPasswort()
    {
        return '';
    }
}

$administrator = new Benutzer();
$administrator->passwort = 'geheim';

echo $administrator->getPasswort(); //gibt '' aus
```

benutzer.php (Version 2)

Codebeispiel 35 *benutzer.php (Version 2)*

Unser Getter gibt nun einen leeren String zurück und verhindert so die Ausgabe des Kennworts. Der Code hat jedoch einen Nachteil. Sie haben zwar einen Getter angelegt, aber wer zwingt Sie (oder einen Kollegen), diesen auch zu verwenden? Niemand hindert Sie daran, trotzdem direkt das Attribut aufzurufen und den Getter zu umgehen.

Beispiel

```php
<?php

class Benutzer
{
    public $name = '';
    public $passwort = '';
    public $gruppen = array();

    function getPasswort()
    {
        return '';
    }
}

$administrator = new Benutzer();
```

```
16 $administrator->passwort = 'geheim';
17
18 echo $administrator->passwort; //gibt weiterhin 'geheim' aus
```

benutzer.php (Version 3)

Codebeispiel 36 *benutzer.php (Version 3)*

Auf diese Weise entstehen unschöne Sicherheitslücken in Ihrem Code. Sie dachten, Sie hätten sich schon um ein Problem gekümmert, und es taucht trotzdem wieder auf. Wäre es nicht besser, den Zugriff auf Attribute von außen verbieten zu können? So erhalten Sie, falls Sie an einer Stelle versehentlich doch direkt auf das Attribut zugegriffen haben, eine Fehlermeldung und können das Problem sofort korrigieren.

Exakt diese Möglichkeit bietet Ihnen PHP, wenn Sie ein Attribut als **geschützt** (engl. protected) kennzeichnen. Innerhalb des Objektes können Sie weiter direkt mit dem Attribut arbeiten, aber von außen ist das Attribut nur indirekt über Methoden erreichbar. Das erreichen Sie, indem Sie bei der Definition des Attributs `public` durch `protected` ersetzen.

Beispiel

```
 1 <?php
 2
 3 class Benutzer
 4 {
 5     public $name = '';
 6     protected $passwort = 'geheim';
 7     public $gruppen = array();
 8
 9     function getPasswort()
10     {
11         return '';
12     }
13 }
14
15 $administrator = new Benutzer();
16
17 echo $administrator->passwort; //erzeugt einen Fehler
18 echo $administrator->getPasswort(); //das funktioniert
```

benutzer.php (Version 4)

Codebeispiel 37 *benutzer.php (Version 4)*

In Zeile 17 wird das PHP-Programm einen Fehler melden, da Sie versucht

haben, direkt auf das geschützte Attribut `$passwort` zuzugreifen.

Durch die Verwendung der Schlüsselwörter `public` und `protected` können Sie also entscheiden, ob von außen auf ein Attribut zugegriffen werden kann. Das gilt übrigens auch für den schreibenden Zugriff. Sie können nun nicht mehr von außen direkt das Attribut verändern. Doch dazu gleich mehr.

> Sie sollten jedes Attribut auf `protected` setzen, für das Sie eine Getter-Methode geschrieben haben, um den doppelten Zugriff zu verhindern. Da Sie gelernt haben, dass es guter Stil ist, für jedes Attribut einen Getter zu haben, sollten Sie also jedes Attribut `protected` machen.

4.4 Setter-Methoden

4.4.1 Nachteile des direkten Änderns eines Attributs

Ähnlich wie beim lesenden Zugriff auf Attribute kann es zu verschiedenen Problemen führen, wenn Sie auf ein Attribut direkt schreibend zugreifen. Die Folgen sind im Allgemeinen noch unangenehmer, da nun auch Daten unkontrolliert verändert werden können:

- Wenn das Attribut nicht existiert, wird es automatisch angelegt. Wenn Sie sich also verschreiben und `$password` statt `$passwort` schreiben, haben Sie plötzlich zwei Attribute und erhalten von PHP noch nicht einmal eine Warnung.
- Wenn Sie ein Attribut als `protected` kennzeichnen, was Sie tun sollten, können Sie gar nicht direkt darauf zugreifen.
- Sie haben keine Möglichkeit, zu überprüfen, ob die Daten, die in das Attribut geschrieben werden, an dieser Stelle überhaupt passen. So können Sie ein Array ablegen, wo eigentlich ein String hingehört, oder den Benutzernamen unter `$passwort` abspeichern.
- Sie haben auch keine Möglichkeit, die Daten noch einmal anzupassen. So

möchten Sie evtl. Leerzeichen trimmen oder ein Passwort verschlüsseln, bevor es im Attribut abgelegt wird.

- Sie haben auch keine Möglichkeit, für bestimmte Attribute den Schreibzugriff zu sperren.

4.4.2 Methoden zum Ändern von Attributen

Warum also verwenden wir nicht Methoden, um Attribute zu ändern? Im einfachsten Fall sieht das folgendermaßen aus:

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    function getVorname()
    {
        return $this->vorname;
    }

    function getNachname()
    {
        return $this->nachname;
    }

    function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    function setNachname($nachname)
    {
        $this->nachname = $nachname;
    }
}
```

person.php (Version 2)

Codebeispiel 38 *person.php (Version 2)*

Wenn Sie nun der Meinung sind, dass das eine Menge Code für eine so simple

Klasse ist, dann haben Sie recht. In der OOP werden Sie in den meisten Fällen mehr Zeilen Code schreiben als in der rein prozeduralen Programmierung. Dafür ist dieser Code aber wesentlich übersichtlicher und flexibler.

So können Sie beispielsweise durch eine Bereinigung der Daten verhindern, dass ungültige oder unerwünschte Werte in Ihre Datenbank gelangen. Natürlich können Sie nicht verhindern, dass ein Benutzer einen falschen Namen angibt. Aber Sie können beispielsweise eine Telefonnummer durch Ersetzung der Trennzeichen bereinigen oder unerwünschte Zeichen entfernen.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';

    function getVorname()
    {
        return $this->vorname;
    }

    function setVorname($vorname)
    {
        $this->vorname = trim($vorname);
    }

}

$person = new Person();
$person->setVorname(' Hans ');

echo $person->getVorname(); //gibt 'Hans' aus
```

person.php (Version 3)

Codebeispiel 39 *person.php (Version 3)*

Mit der Zeit werden Sie sich komplexere Setter ausdenken als diese simple Entfernung unnötiger Leerzeichen. So könnten Sie beispielsweise auch die Groß-/Kleinschreibung anpassen. Das Wichtige ist jedoch: Von außen ändert sich nichts. Sie rufen immer noch dieselbe Methode auf, egal was im Inneren passiert. Hier sehen Sie ein sehr wichtiges Prinzip, das Sie sich zu eigen machen sollten:

> Versuchen Sie – wo immer möglich – nur im Inneren von Klassen etwas zu ändern. Auf diese Weise müssen Sie den Code, der mit den Objekten arbeitet, nicht anpassen. Sie sparen viel Zeit und Ärger.

Da das Ändern von Attributen durch Methoden sehr verbreitet ist, hat sich auch hier ein eigener Begriff eingebürgert. Methoden, die Attribute ändern, nennt man ***Setter-Methoden*** oder kurz ***Setter***.

Wie bei den Gettern existiert auch hier eine Namens-Konvention. Ein Setter beginnt mit `set`, gefolgt vom Namen des Attributs.

Beispiel

- `function setVorname($vorname)`
- `function setPasswort($passwort)`
- `function setLieferAdresse($lieferAdresse)`

4.5 Öffentliche und geschützte Methoden

Sie können eine Methode, die nur innerhalb eines Objektes verwendet werden soll, als `protected` markieren. Diese Art Methoden sind häufig Helfer für die von außen sichtbaren Methoden, bei denen es unnötig oder sogar gefährlich wäre, den Zugriff von außerhalb eines Objekts zu gestatten.

Beispiel

```
1  <?php
2
3  class GeschuetzteMethoden
4  {
5      public function ichBinOeffentlich()
6      {
7          return 'Auf mich kann man von außen zugreifen.';
8      }
9
10     protected function ichBinGeschuetzt()
```

```
11      {
12          return 'Ich bin die geschuetzte Methode.';
13      }
14  }
15
16  $test = new GeschuetzteMethoden();
17
18  echo $test->ichBinOeffentlich();
19  echo $test->ichBinGeschuetzt(); // Hier wird von PHP ein Erro
```

geschuetzte_methoden.php

Codebeispiel 40 *geschuetzte_methoden.php*

In Zeile 5 wird die Methode `ichBinOeffentlich()` wie ein Attribut mit dem Schlüsselwort `public` definiert. Da eine Methode standardmäßig öffentlich ist, können Sie theoretisch `public` an dieser Stelle weglassen. Wir werden die Sichtbarkeit jedoch immer angeben.

In Zeile 10 wird die zweite Methode als `protected` markiert. Dementsprechend scheitert auch der Methoden-Aufruf in Zeile 19.

4.6 Testen Sie Ihr Wissen

1. Was unterscheidet Getter- und Setter-Methoden von normalen Methoden?
2. Warum ist Ihr Code besser wartbar, wenn Sie Attribute über Methoden verändern statt direkt?
3. Wie können Sie auf ein Attribut nur lesenden Zugriff erlauben?

4.7 Aufgaben zur Selbstkontrolle

Übung 10:

In Lektion 3 haben Sie eine Klasse für ein Objekt in Ihrer Umgebung geschaffen. Erweitern Sie Ihre Klasse. Alle Attribute sollen nun `protected` sein.

Übung 11:

Schreiben Sie für jedes Attribut Getter-Methoden. Benutzen Sie diese, um die Werte der Attribute in der *index.php* anzuzeigen.

Übung 12:

Schreiben Sie für jedes Attribut Setter-Methoden. Verwenden Sie diese in der *index.php* zur Anpassung von einigen Werten in den Attributen.

5 Arbeiten mit Objekten

In dieser Lektion lernen Sie
• wie Objekte miteinander kommunizieren und arbeiten. • dass Sie Objekte in der Session speichern können.

5.1Das Problem

Sie wissen zwar inzwischen, was Objekte, Klassen, Methoden und Attribute sind, haben aber noch keine wirkliche Vorstellung, wie Sie mit ihnen arbeiten sollen. Um die menschliche Sprache als Metapher zu verwenden: Sie haben zwar die Vokabeln gelernt, können aber noch keine kompletten Sätze bilden. Sie werden in dieser Lektion kaum neue Funktionalität von PHP kennenlernen. Vielmehr werden Sie lernen, mit der objektorientierten Programmierung eleganten Code zu schreiben.

5.2 Methoden, die andere Methoden aufrufen

Sie haben gelernt, wie Sie Methoden definieren und wie Methoden in Objekten aufgerufen werden. Was aber tun Sie, wenn Sie in einer Methode eine andere Methode desselben Objekts aufrufen wollen?

Beispiel

```
1  <?php
2
3  class Person
4  {
5      protected $vorname = '';
6      protected $nachname = '';
7
8      public function getVorname()
9      {
10         return $this->vorname;
```

```
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function beschreibe()
    {
        $text = 'Die Person heisst' . $this->vorname . ' ';
        $text .= $this->nachname . '!';
        return $text;
    }
}
```

person.php (Version 1)

Codebeispiel 41 *person.php (Version 1)*

Sie sehen im Beispiel eine Klasse `Person` mit den Attributen `$vorname` und `$nachname` und den entsprechenden Gettern. Zusätzlich hat `Person` noch eine Methode `beschreibe()`, die das jeweilige `Person`-Objekt in Worten beschreibt.

Diese Methode `beschreibe()` greift direkt auf die geschützten Attribute zu, was möglich ist, weil `protected` nur den Zugriff von außen unterbindet. Trotzdem, wir haben Getter, also sollten wir sie überall verwenden.

Wenn Sie in einer Methode eine andere Methode aufrufen wollen, so können Sie das mit dem Ausdruck `$this->methodenName()` erreichen. Die spezielle Variable `$this` ist notwendig, um zu zeigen, dass wir eine Methode dieses Objektes aufrufen, keine globale Funktion. Folgendes Beispiel ist also **falsch**:

Beispiel

```
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function getVorname()
    {
        return $this->vorname;
    }

```

```
13      public function getNachname()
14      {
15          return $this->nachname;
16      }
17
18      public function beschreibe()
19      {
20          $text = 'Die Person heisst' . getVorname() . ' '; //
21          $text .= getNachname() . '!'; // noch ein Fehler
22          return $text;
23      }
24  }
```

person.php - Fehler (Version 2)

Codebeispiel 42 *person.php - Fehler (Version 2)*

Die Methode `beschreibe()` versucht, globale Funktionen namens `getVorname()` und `getNachname()` zu finden und wird einen PHP-Fehler erzeugen, da sie diese nicht finden kann. **Richtig** ist folgende Variante:

Beispiel

```
 1  <?php
 2
 3  class Person
 4  {
 5      protected $vorname = '';
 6      protected $nachname = '';
 7
 8      public function getVorname()
 9      {
10          return $this->vorname;
11      }
12
13      public function getNachname()
14      {
15          return $this->nachname;
16      }
17
18      public function beschreibe()
19      {
20          $text = 'Die Person heisst' . $this->getVorname() .
21          $text .= $this->getNachname() . '!';
22          return $text;
23      }
24  }
```

person.php

Codebeispiel 43 *person.php*

Jetzt weiß PHP durch `$this`, dass eine Methode dieses Objekts gemeint ist.

5.3 Methoden in anderen Objekten aufrufen

5.3.1 Grundlagen

Sie können natürlich auch Methoden in anderen Objekten aufrufen. Die Syntax ist die, die Sie schon kennen. Sie rufen die Methode mit dem Pfeil-Operator auf der Variablen auf, die das Objekt enthält. Die Kunst ist nur, das Objekt der anderen Methode bekannt zu machen. Das können Sie auf mehrere Arten erreichen:

Objekte als lokale Variablen

Sie erzeugen das zweite Objekt innerhalb der Methode. Damit ist das Objekt eine lokale Variable und kann wie jede andere lokale Variable verwendet werden.

Beispiel

```
 1 <?php
 2
 3 require_once 'person.php';
 4
 5 class Test
 6 {
 7     public function beschreibePerson()
 8     {
 9         $person = new Person();
10         return $person->beschreibe(); //gibt 'Die Person hei
11     }
12 }
13
14 $test = new Test();
15 echo $test->beschreibePerson();
```

test.php (Version 1)

Codebeispiel 44 *test.php (Version 1)*

Objekte als Parameter übergeben

Sie können auch ein bereits bestehendes Objekt als Parameter an eine Methode übergeben. Da sich Objekte wie jeder andere Datentyp verhalten, macht es keinen Unterschied, ob Sie einen String, ein Array oder eben ein Objekt übergeben.

Beispiel

```php
<?php

require_once 'person.php';

class Test
{
    public function beschreibePerson($person)
    {
        return $person->beschreibe(); //gibt 'Die Person hei
    }
}

$teriete = new Person();
$test = new Test();
echo $test->beschreibePerson($teriete); //Objekt wird ueberge
```

test.php (Version 2)

Codebeispiel 45 *test.php (Version 2)*

Objekte in superglobalen Variablen

Auch die dritte Variante kennen Sie im Prinzip schon. Wenn ein Objekt in einer global verfügbaren Variablen liegt, können Sie direkt darauf zugreifen. Das erreichen Sie zum Beispiel über die Verwendung des superglobalen Arrays `$_SESSION`. Damit haben Sie gleich noch eine andere Sache gelernt:

> Genau wie Strings, Integers, Arrays usw. können Sie auch Objekte in der Session ablegen. Allerdings müssen die Klassendefinitionen vor dem Start der Session bekannt sein (siehe http://phpperformance.de/web-dev/php/objekte-in-sessions-speichern). Außerdem müssen alle Attribute serialisierbare Datentypen enthalten, womit der Datentyp Resource (siehe http://php.net/de/language.types.resource) nicht erlaubt ist.

Beispiel

```php
<?php

require_once 'person.php';
session_start();

// Objekt in der Session ablegen
$_SESSION['teriete'] = new Person();
```

test.php (Version 3) - Session befüllen

Codebeispiel 46 *test.php (Version 3) - Session befüllen*

```php
<?php

require_once 'person.php';

class Test
{
    public function beschreibePerson($person)
    {
        return $person->beschreibe(); //gibt 'Die Person hei
    }
}

session_start();
$test = new Test();

// Objekt aus der Session auslesen und als Parameter uebergel
echo $test->beschreibePerson($_SESSION['teriete']);
```

test.php (Version 3b) - Session auslesen

Codebeispiel 47 *test.php (Version 3b) - Session auslesen*

Die Syntax ist - zugegeben - hier etwas gewöhnungsbedürftig. Wenn ein Array-Element ein Objekt ist, wie es hier bei `$_SESSION['teriete']` der Fall ist, können Sie dieses auch problemlos als Parameter übergeben. Theoretisch ist auch innerhalb einer Methode ein direkter Zugriff auf superglobale Variablen möglich, ich würde dies wegen der schlechteren Lesbarkeit des Codes aber möglichst vermeiden.

Objekte in Attributen ablegen

Sie können auch ein bereits bestehendes Objekt als Parameter an einen Setter übergeben und so in einem Attribut ablegen. Es macht keinen Unterschied, ob Sie einen String, ein Array oder eben ein Objekt in einem Attribut ablegen.

Beispiel

```php
<?php

require_once 'person.php';

class Test
{
    protected $person;

    public function setPerson($person)
    {
        $this->person = $person;
    }

    public function getPerson()
    {
        return $this->person;
    }

    public function beschreibePerson()
    {
        $person = $this->getPerson();
        return $person->beschreibe(); //gibt 'Die Person hei
    }
}

$teriete = new Person();
$test = new Test();
$test->setPerson($teriete); //Objekt wird uebergeben
echo $test->beschreibePerson();
```

test.php (Version 4)

Codebeispiel 48 *test.php (Version 4)*

5.3.2 Der instanceof-Operator

Wenn Sie ein Objekt als Parameter übergeben oder auf eine globale Variable zugreifen, können Sie nicht sicher sein, ob auch tatsächlich ein passendes Objekt enthalten ist. Die Variable könnte leer sein oder z. B. einen String beinhalten. Das ist insbesondere dann problematisch, wenn Sie versuchen, eine Methode auf der Variablen aufzurufen.

Beispiel

```php
<?php

require_once 'person.php';
$test2 = new Person();

//hier passieren viele Dinge ...
$test2 = 'Das ist ein Test';
//hier passieren noch mehr Dinge ...

echo $test2->beschreibe(); //erzeugt einen Fehler
```

test2.php (Version 1)

Codebeispiel 49 *test2.php (Version 1)*

Hier hat sich in Zeile 7 ein böser Fehler eingeschlichen. Wir haben versehentlich das `Person`-Objekt `$test2` mit einem String überschrieben. Wenn wir in Zeile 10 nun versuchen, auf diesem String die Methode aufzurufen, erhalten wir eine Fehlermeldung.

Zugegeben, das Beispiel ist etwas künstlich, aber gerade wenn Sie Objekte als Parameter übergeben, sollten Sie mit Problemen rechnen. So kann es beispielsweise durchaus sein, dass Sie nur unter bestimmten Umständen das erwartete Objekt erhalten. Stellen Sie dies also sicher, bevor Sie mit Methoden des Objekts arbeiten, was beispielsweise mit dem Operator ***instanceof*** möglich ist. Er testet, ob die fragliche Variable ein Objekt enthält, das aus einer bestimmten Klasse instanziiert wurde, und gibt dementsprechend entweder `true` oder `false` zurück.

Beispiel

```php
<?php

require_once 'person.php';
$test2 = new Person();

//hier passieren viele Dinge ...
$test2 = 'Das ist ein Test';
//hier passieren noch mehr Dinge ...

if ($test2 instanceof Person) {
    echo $test2->beschreibe();
}
```

test2.php (Version 2)

Codebeispiel 50 *test2.php (Version 2)*

In Zeile 10 wird getestet, ob `$test2` ein Objekt enthält, das aus der Klasse `Person` erzeugt wurde. Da Objekte auch als Instanzen von Klassen bezeichnet werden, macht auch der Name des Operators Sinn. Wann immer Sie nicht absolut sicher sein können, dass Sie an einer Stelle ein passendes Objekt haben, sollten Sie vorher testen und entsprechende Gegenmaßnahmen ergreifen. Das könnte z. B. bedeuten, einen Fehler zu erzeugen, oder auch, das fragliche Objekt einfach anzulegen.

Beispiel

```
1  <?php
2
3  require_once 'person.php';
4  $test2 = new Person();
5
6  //hier passieren viele Dinge ...
7  $test2 = 'Das ist ein Test';
8  //hier passieren noch mehr Dinge ...
9
10 if (!($test2 instanceof Person)) {
11     //wenn $test2 keine Person ist, machen wir eine draus
12     $test2 = new Person();
13 }
14
15 echo $test2->beschreibe();
```

test2.php (Version 3)

Codebeispiel 51 *test2.php (Version 3)*

5.3.3 Type-Hinting

Hinter dem seltsamen Begriff ***Type-Hinting*** verbirgt sich eine weitere Möglichkeit, die Klasse eines Objekts zu überprüfen. Sie können bei der Definition einer Methode jedem Parameter den Namen einer Klasse mitgeben, die in dem Parameter abgelegt sein muss.

Beispiel

```
1  <?php
2
3  class Adresse
4  {
```

```
 5 }
 6
 7 class Person
 8 {
 9     public function setAdresse(Adresse $adresse)
10     {
11         //tuewas mit dem Adress-Objekt
12     }
13 }
14
15 $hans = new Person();
16 $adresse = new Adresse();
17 $hans->setAdresse($adresse);
18 $hans->setAdresse('Hinter dem Mond'); //erzeugt Fehler
```

type_hint.php

Codebeispiel 52 *type_hint.php*

Der Code in Zeile 17 wird problemlos ausgeführt, da, wie verlangt, ein `Adresse`-Objekt übergeben wurde. In Zeile 18 hingegen wird von PHP ein Fehler gemeldet, da ein String übergeben wurde.

> Wann immer Sie einer Methode ein Objekt als Parameter übergeben, sollten Sie den Typ prüfen. Ihr Code wird auf diese Weise robuster gegen versehentlich falsch übergebene Parameter. Diese Technik funktioniert seit PHP 5.1 ebenfalls bei Arrays (Type-Hint `array`). Mit PHP 7 kamen dann auch noch Type Hints für die skalaren PHP-Datentypen (`string`, `int`, `float` und `bool`) hinzu. Letztere führen allerdings standardmäßig zu einer automatischen Typwandlung. Ein strikter Typcheck ohne Wandlung ist nur auf Dateiebene aktivierbar, was viele Entwickler als wenig hilfreich erachten. Ich persönlich habe mir hierzu noch keine endgültige Meinung gebildet.

5.4 Testen Sie Ihr Wissen

1. Wie können Sie eine Methode desselben Objektes von einer anderen Methode aus aufrufen?

2. Können Sie Objekte in der Session speichern?

3. Was testet der Operator `instanceof`?

5.5 Aufgaben zur Selbstkontrolle

Übung 13:

Legen Sie eine Klasse `Fussball` an, die die Attribute `$farbe` und `$durchmesser` besitzt.

Übung 14:

Schreiben Sie Getter- und Setter-Methoden für `Fussball`, die die Attribute auslesen und ändern. Vergeben Sie für die beiden Attribute Standardwerte, damit Sie die Setter gleich besser testen können.

Übung 15:

Schreiben Sie in `Fussball` eine Methode `beschreibeFussball()`, die etwa folgenden Text zurückgibt: `Dieser Fussball ist schwarz und hat einen Durchmesser von 30 cm.` Die Methode soll sich die Werte von den entsprechenden Gettern von `Fussball` holen.

Übung 16:

Schreiben Sie ein PHP-Skript *index.php*, das eine Instanz von `Fussball` in der Variablen `$ball` anlegt, diese mittels Setter mit den zwei Werten befüllt und dann deren Methode `beschreibeFussball()` aufruft.

Übung 17:

Erweitern Sie die *index.php*, so dass diese prüft, ob in `$ball` wirklich ein

Objekt der Klasse `Fussball` abgelegt ist. Wenn ja, soll die Methode `beschreibeFussball()` aufgerufen werden, ansonsten wird die Meldung `In $ball ist kein Fussball` ausgegeben.

6 Virtuelle Attribute

In dieser Lektion lernen Sie
• wie flexibel Getter und Setter sind und welche Vorteile das hat. • wie Sie mit virtuellen Attributen arbeiten.

6.1Das Problem

In vielen Programmier-Projekten gibt es zwei Anforderungen an die Daten.

- Sie sollen möglichst effizient und nicht redundant abgelegt werden.
- Es soll möglichst einfach sein, mit ihnen zu arbeiten.

Leider schließen sich die beiden Bedingungen meistens aus:

Junior-Programmierer	Senior-Programmierer
Können wir nicht ein Attribut Name anlegen, das den Vornamen und den Nachnamen enthält? Das brauchen wir ständig und es ist aufwendig, das jedes Mal zusammenzubauen.	Wir dürfen auf keinen Fall Daten an mehreren Stellen abspeichern. Das macht uns später garantiert Ärger.
Wir brauchen dieselben Texte einmal unformatiert und einmal mit HTML-Tags. Können wir nicht beides abspeichern?	Dieselben Informationen in mehreren Formaten abzuspeichern erzeugt garantiert Inkonsistenzen. Sucht euch ein Format aus und konvertiert die Daten bei Bedarf!
Mit den Daten, die ihr liefert, kann man unmöglich vernünftig arbeiten!	Würden wir die Daten so ablegen, wie ihr es wollt, gäbe es ein heilloses Durcheinander!

Tabelle 6.1 *Konflikte*

6.2 Virtuelle Attribute

6.2.1 Konzept

Es gibt also einen Konflikt, wie Daten abzulegen sind. Einerseits möchten wir sie möglichst gut strukturiert und wiederverwendbar abspeichern, andererseits sollte man auch gut mit ihnen arbeiten können. Die objektorientierte Programmierung hat für dieses Problem eine praktische Lösung parat: Sie lügt!

Das ist weniger schlimm, als es sich anhört. Wir gaukeln einfach die Existenz von Attributen vor, die es in Wirklichkeit gar nicht gibt. Dies erreichen wir, indem wir Getter-Methoden schreiben, die sich die Daten aus anderen Attributen holen und aufbereiten.

> Diese Vorgehensweise ist als **Uniform Access Principle** (siehe https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs#Uniform_Ac bekannt. Das Prinzip besagt, dass für den Zugriff von außen eine gleichartige Notation (Schreibweise) verwendet werden soll. Diese Notation gibt jedoch nicht preis, ob sie über direkte Datenzugriffe oder »Berechnungen« umgesetzt wurde. Die Notwendigkeit einer solchen Vorgehensweise wurde übrigens zuerst von *Bertrand Meyer* erkannt und beschrieben.

Beispiel

```php
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0; // Nettopreis

    public function getTitel()
    {
        return $this->titel;
    }

    public function getPreis()
    {
```

```
15          return $this->preis;
16      }
17
18      public function getBruttoPreis()
19      {
20          $bruttoPreis = $this->getPreis() * 1.07;
21          return $bruttoPreis;
22      }
23  }
```

buch.php (Version 1)

Codebeispiel 53 *buch.php (Version 1)*

Im Beispiel sehen Sie eine Klasse `Buch`, die über zwei echte Attribute verfügt. Auf Setter habe ich hier bewusst verzichtet, um das Beispiel kurz zu halten. Es existieren aber drei Getter, wobei einer, nämlich `getBruttoPreis()`, kein reales Attribut ausgibt. Diese Methode nimmt das Attribut `$preis` und rechnet sieben Prozent Mehrwertsteuer dazu. Den errechneten Wert gibt sie dann zurück. Von außen scheint es also, als gäbe es ein Attribut `$bruttoPreis`, das durch diesen Getter verwaltet wird. Dieses Vorgaukeln von Attributen bezeichnen wir als ***virtuelle Attribute***.

6.2.2 Setter für virtuelle Attribute

Natürlich ist es genauso möglich, virtuelle Attribute mit Settern auszustatten. Sie müssen in diesem Fall allerdings beachten, dass Sie nun unter Umständen mehrere Setter für ein reales Attribut haben. Diese dürfen sich nie in die Quere kommen. Davon abgesehen müssen Sie nur aufpassen, dass Sie die korrekten Werte erzeugen.

Beispiel

```
 1  <?php
 2
 3  class Buch
 4  {
 5      protected $titel = '';
 6      protected $preis = 0; // Nettopreis
 7
 8      public function getTitel()
 9      {
10          return $this->titel;
```

```
11      }
12
13      public function setTitel($titel)
14      {
15          $this->titel = $titel;
16      }
17
18      public function getPreis()
19      {
20          return $this->preis;
21      }
22
23      public function setPreis($preis)
24      {
25          $this->preis = $preis;
26      }
27
28      public function getBruttoPreis()
29      {
30          $bruttoPreis = $this->getPreis() * 1.07;
31          return $bruttoPreis;
32      }
33
34      public function setBruttoPreis($bruttoPreis)
35      {
36          $this->setPreis($bruttoPreis / 1.07);
37      }
38  }
```

buch.php (Version 2)

Codebeispiel 54 *buch.php (Version 2)*

Beim Setter des virtuellen Attributs `$bruttoPreis` müssen Sie dieses Mal durch 1,07 teilen, um den Nettopreis zu erhalten.

6.3 Testen Sie Ihr Wissen

1. Was unterscheidet virtuelle Attribute von echten Attributen?
2. Unter welchen Umständen sind virtuelle Attribute nützlich?
3. Was ist bei Settern von virtuellen Attributen zu beachten?

6.4 Aufgaben zur Selbstkontrolle

Übung 18:

Schreiben Sie eine Klasse `Person` mit den Attributen `$vorname`, `$nachname` und `$geburtTimestamp` mit den entsprechenden Gettern und Settern. Nutzen Sie eine extra Datei *index.php*, um ein Objekt zu erstellen, dieses zu befüllen und den kompletten Namen auszugeben. Zum Testen können Sie der Einfachheit halber den heutigen Timestamp verwenden.

Übung 19:

`Person` soll das virtuelle Attribut `$name` mit dem Getter `getName()` enthalten, das den Vor- und Nachnamen enthält. Ändern Sie die Getter-Aufrufe in der *index.php* entsprechend ab.

6.5 Optionale Aufgaben

Übung 20:

`Person` soll das virtuelle Attribut `$geburtstag` mit Getter enthalten, das den Geburtstag im Format `tt.mm.` zurückgibt. Ergänzen Sie die Ausgabe des Geburtstags in Ihrer *index.php*.

Übung 21:

Schreiben Sie die Methode `setName()`, der Sie den Vor- und Nachnamen als einen String übergeben können. Die Methode soll den String in den Vor- und den Nachnamen zerlegen und den entsprechenden echten Attributen zuweisen. Verwenden Sie hierfür die PHP-Funktion `explode()`. Nehmen Sie der Einfachheit halber an, dass der Name nur ein Leerzeichen enthalten kann. Testen Sie den neuen Setter.

Übung 22:

Schreiben Sie eine Methode `setGeburtstag()`, der Sie ein Datum im Format `tt.mm.jjjj` übergeben können. Diese erzeugt daraus den ***Timestamp*** und speichert ihn in `$geburtTimestamp`. Verwenden Sie hierfür die PHP-Funktionen `strtotime()`. Ziehen Sie für die genaue Syntax http://php.net/ zu Rate. Testen Sie auch diesen Setter.

7 Magische Methoden

In dieser Lektion lernen Sie
• was eine magische Methode ist. • wann die magischen Methoden `__toString()` und `__construct()` aktiviert werden. • was beim Aufruf von `new` eigentlich passiert. • wie Konstruktoren Ihnen das Erzeugen von Objekten erleichtern.

7.1Das Problem

Je häufiger Sie mit Objekten arbeiten, desto mehr werden Sie feststellen, dass Sie bestimmte Dinge mit jedem Objekt einer Klasse anstellen. Jedes Mal, wenn Sie ein Person-Objekt erzeugen, rufen Sie danach die Methoden `setVorname()` und `setNachname()` auf.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function getVorname()
    {
        return $this->vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
```

```
21      }
22
23      public function setNachname($nachname)
24      {
25          $this->nachname = $nachname;
26      }
27  }
```

person.php

Codebeispiel 55 *person.php*

```
1  <?php
2
3  require_once 'person.php';
4
5  $remolt = new Person();
6  $remolt->setVorname('Marc');
7  $remolt->setNachname('Remolt');
8
9  var_dump($remolt);
```

neue_person.php (Version 1)

Codebeispiel 56 *neue_person.php (Version 1)*

Gegen diese Schreibweise ist nichts einzuwenden, nur wäre es sinnvoll, wenn Dinge, die jedes Mal beim Erzeugen eines Objekts gemacht werden sollen, auch automatisch gemacht werden. Es ist einfach unnötig, drei Zeilen Code zu schreiben, wo theoretisch nur eine notwendig ist.

7.2 Magische Methoden

7.2.1 Konzept

Keine Sorge, wir driften jetzt nicht in die Esoterik ab. Eine ***magische Methode*** (engl. magic method) ist eine Methode, die Sie nicht aufrufen müssen, sondern die selbst weiß, wann sie gebraucht wird. Daher auch magisch - plötzlich tut sie etwas, ohne dass Sie es sagen mussten.

Diese Methoden haben einen festgelegten Namen und einen Auslöser. Wann immer dieser Auslöser aktiviert wird, sieht PHP nach, ob Sie die passende magische Methode definiert haben. Wenn ja, wird sie ausgeführt. Einige mögliche Auslöser sind:

- Sie versuchen, ein Objekt mit `echo` auszugeben.
- Ein neues Objekt einer bestimmten Klasse wird angelegt.
- Sie versuchen, eine nicht existierende Methode aufzurufen.
- Sie versuchen, lesend auf ein nicht existierendes Attribut zuzugreifen.

> Für eine komplette Liste aller magischen Methoden und ihrer Auslöser in PHP möchte ich Sie auf die PHP-Referenz verweisen: http://php.net/de/language.oop5.magic.

7.2.2 Die Methode __toString()

Bevor wir uns an die Konstruktoren heranwagen, lassen Sie uns erst ein einfaches Beispiel für eine magische Methode betrachten, nämlich `__toString()`. Was als Erstes auffällt, ist der seltsame Name, der mit zwei Unterstrichen (engl. Underscore) beginnt. Das ist ein Kennzeichen für eine magische Methode: Jede beginnt mit zwei Unterstrichen __.

Diese Methode wird immer dann aktiv, wenn Sie versuchen, ein Objekt mit `echo` auszugeben, was normalerweise fehlschlägt.

Beispiel

```
1 <?php
2
3 require_once 'person.php';
4
5 $remolt = new Person();
6 $remolt->setVorname('Marc');
7 $remolt->setNachname('Remolt');
8
9 echo $remolt; //Erzeugt einen Fehler
```

neue_person.php - Fehler (Version 2)

Codebeispiel 57 *neue_person.php - Fehler (Version 2)*

In Zeile 9 passiert genau das, was wir erwarten, es wird ein Fehler erzeugt. Es ist auch ziemlich abwegig, ein Objekt als Text ausgeben zu wollen. Wenn wir allerdings die magische Methode `__toString()` in der Klasse definieren, sieht

die Sache ganz anders aus.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __toString()
    {
        return $this->getVorname() . ' ' . $this->getNachname
    }

    public function getVorname()
    {
        return $this->vorname;
    }

    public function getNachname()
    {
        return $this->nachname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }

    public function setNachname($nachname)
    {
        $this->nachname = $nachname;
    }
}
```

person_to_string.php

Codebeispiel 58 *person_to_string.php*

```php
<?php

require_once 'person_to_string.php';

$remolt = new Person();
$remolt->setVorname('Marc');
$remolt->setNachname('Remolt');

echo $remolt; //Gibt "Marc Remolt" aus
```

neue_person.php (Version 2b)
Codebeispiel 59 *neue_person.php (Version 2b)*

Sobald das Objekt `$remolt` im Zusammenhang mit `echo` aufgerufen wird, ruft PHP die Methode `__toString()` auf und verwendet den Rückgabewert an der Stelle im Code als Text.

Sie können die Methode `__toString()` also verwenden, um eine für Menschen lesbare Repräsentation des Objekts zu erhalten. Meistens nutzt man ohnehin eine derartige Methode, und so erhält man noch den Magie-Bonus.

> Sie müssen allerdings beachten, dass `__toString()` immer einen String zurückgeben muss. Im Grunde ist das zwar logisch, es wird aber leider immer wieder vergessen.

7.3 Konstruktoren

7.3.1 Konzept

Als ***Konstruktor*** bezeichnet man eine spezielle Methode, die beim Erzeugen eines Objekts aufgerufen wird. In ihr sollten Sie alles abarbeiten, was zur Verwendung des Objekts notwendig ist. Möglichkeiten gibt es hier viele, daher nur ein paar Beispiele:

- Es werden Standardwerte für die Attribute gesetzt.
- Die Attribute werden auf Basis von Formulardaten befüllt.
- Das Objekt benötigt andere Objekte zum Betrieb. Diese werden im Konstruktor erzeugt.

7.3.2 Die Methode __construct()

Der Konstruktor könnte im Prinzip eine beliebige Methode sein, die Sie sofort aufrufen, nachdem Sie ein Objekt mit `new` erzeugt haben. Dieser Weg hat

allerdings zwei Nachteile:

- Sie benötigen zwei Zeilen Code.
- Sie könnten den Methoden-Aufruf vergessen.

Es ist besser, sich die Arbeit von PHP abnehmen zu lassen. Wenn Sie Ihren Konstruktor `__construct()` nennen, wird diese Methode automatisch beim Aufruf von `new` ausgeführt.

Beispiel

```php
<?php

class TestKlasse
{
    public $text;

    public function __construct()
    {
        $this->text = 'Hallo Welt';
    }
}

$test = new TestKlasse();
echo $test->text; //Gibt "Hallo Welt" aus
```

test.php (Version 1)

Codebeispiel 60 *test.php (Version 1)*

Wann immer Sie also ein neues Objekt aus der Klasse `TestKlasse` erzeugen, wird das Attribut `$text` mit einem festen String befüllt. Das mag kein sonderlich sinnvolles Beispiel sein, aber es zeigt das Konzept.

7.3.3 Parameter an __construct() übergeben

Wie aber können Sie schon beim Erzeugen des Objekts Werte übergeben? Sie könnten einer Person schon beim Erzeugen einen Namen geben oder einen variablen Text in unserem vorherigen Beispiel nutzen.

Das zu erreichen ist nicht schwer, und Sie erfahren auch endlich, warum Sie an den Klassen-Namen beim Aufruf von `new` immer Klammern anhängen, als ob Sie

eine Methode aufrufen würden. Die Antwort müssten Sie inzwischen schon kennen: Zwar rufen nicht Sie eine Methode auf, aber PHP tut es, nämlich `__construct()`.

Alles, was Sie als Parameter in die Klammern hinter dem Klassen-Namen schreiben, landet direkt als Parameter in der magischen Methode.

Beispiel

```
1  <?php
2
3  class TestKlasse
4  {
5      public $text;
6
7      public function __construct()
8      {
9          $this->text = 'Hallo Welt';
10     }
11 }
12
13 $test = new TestKlasse('Hallo Leute!');
14 echo $test->text; //Gibt immer noch "Hallo Welt" aus
```

test.php (Version 2)

Codebeispiel 61 *test.php (Version 2)*

Der Code in Zeile 13 führt dazu, dass in dem Objekt `$test` die Methode `__construct()` mit dem String `'Hallo Leute!'` als Parameter aufgerufen wird, also genauso, als hätten Sie `$test->__construct('Hallo Leute!')` von Hand aufgerufen. Letzteres ist technisch gesehen übrigens durchaus möglich, aber normalerweise durch den an einen Auslöser gebundenen automatisierten Aufruf unnötig. Durch den Aufruf ändert sich im Moment jedoch noch nichts im Objekt, da die Methode noch keine Parameter erwartet.

Beispiel

```
1  <?php
2
3  class TestKlasse
4  {
5      protected $text;
6
```

```php
    public function __construct($eingabe)
    {
        $this->setText($eingabe);
    }

    public function setText($text)
    {
        $this->text = $text;
    }

    public function getText()
    {
        return $this->text;
    }
}

$test = new TestKlasse('Hallo Welt');
echo $test->getText(); //Gibt "Hallo Welt" aus

$test = new TestKlasse('Wie geht es euch denn so?');
echo $test->getText(); //Gibt "Wie geht es euch denn so?" aus
```

test.php (Version 2b)

Codebeispiel 62 *test.php (Version 2b)*

Jetzt wird der String in der Methode `__construct()` im Attribut `$text` abgelegt und kann wie gewohnt weiterverarbeitet werden. Nebenbei haben wir auch noch den Code überarbeitet, indem wir einen passenden Getter und Setter geschrieben haben.

Lassen Sie uns nun das Problem vom Beginn dieser Lektion erneut aufgreifen. Der Aufruf könnte nun folgendermaßen aussehen:

Beispiel

```php
<?php

require_once 'person_konstruktor.php';

$remolt = new Person('Marc', 'Remolt');

echo $remolt; //Gibt "Marc Remolt" aus
```

neue_person.php (Version 3)

Codebeispiel 63 *neue_person.php (Version 3)*

Natürlich müssen Sie noch die Klasse `Person` um einen Konstruktor erweitern,

der den Vornamen und den Nachnamen als Parameter akzeptiert. Es ist also nicht wirklich eine Zeile, aber zumindest sehen Sie nur noch eine.

7.3.4 Ein assoziatives Array an den Konstruktor übergeben

Wenn Sie dem Konstruktor viele Werte übergeben müssen, bietet es sich an, dies als assoziatives Array zu tun. So müssen Sie nicht auf die Reihenfolge der Parameter achten oder können das Array auch vorbefüllen und dem Konstruktor als eine Variable übergeben.

Im Konstruktor weisen Sie dann die einzelnen Werte aus dem Array den Attributen des Objekts zu, indem Sie deren Setter verwenden.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __construct(array $daten = array())
    {
        if ($daten) {
            $this->setVorname($daten['vorname']);
            $this->setNachname($daten['nachname']);
        }
    }

    public function __toString()
    {
        return $this->getVorname() . ' ' . $this->getNachname
    }

    public function getVorname()
    {
        return $this->vorname;
    }

    public function setVorname($vorname)
    {
        $this->vorname = $vorname;
    }
```

```
30
31      public function getNachname()
32      {
33          return $this->nachname;
34      }
35
36      public function setNachname($nachname)
37      {
38          $this->nachname = $nachname;
39      }
40  }
```

person_konstruktor_array.php

Codebeispiel 64 *person_konstruktor_array.php*

```
 1  <?php
 2
 3  require_once 'person_konstruktor_array.php';
 4
 5  $person = new Person(
 6      array(
 7          'vorname' => 'Arthur',
 8          'nachname' => 'Dent',
 9      )
10  );
11
12  echo $person;
```

neue_person.php (Version 4)

Codebeispiel 65 *neue_person.php (Version 4)*

Der Konstruktor erhält als optionalen Parameter ein Array `$daten` und prüft über den TypeHint, ob er auch tatsächlich ein Array erhält. Die Methode selbst prüft, ob das Array Daten enthält. Wenn ja, werden die Setter der Attribute mit den passenden Schlüsseln des Arrays beliefert.

Ein weiterer Vorteil dieses Konstruktors ist übrigens, dass Sie ihn sehr gut zusammen mit Formularen verwenden können. Formulardaten liegen normalerweise in `$_POST` in Form eines assoziativen Arrays. Na, klingelt es schon? Mit einem derartig aufgebauten Konstruktor können Sie folgenden Code schreiben:

```
$person = new Person($_POST);
```

Das ist doch mal praktisch, oder?

Wir können die ganze Sache sogar noch ein wenig weiter treiben. Oftmals existiert ein namentlicher Zusammenhang zwischen den Array-Schlüsseln, den Attributen der Klasse und den Settern. So gehören zum Array-Schlüssel `vorname` das Attribut `$vorname` und der Setter `setVorname`.

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __construct(array $daten = array())
    {
        // wenn $daten nicht leer ist, rufe die passenden Se
        if ($daten) {
            foreach ($daten as $k => $v) {
                $setterName = 'set' . ucfirst($k);
                // pruefe ob ein passender Setter existiert
                if (method_exists($this, $setterName)) {
                    $this->$setterName($v); // Setteraufruf
                }
            }
        }
    }

    // gekuerztes Beispiel ohne __toString und Getter/Setter
}
```

person_konstruktor_schleife.php

Codebeispiel 66 *person_konstruktor_schleife.php*

Das Beispiel nutzt genau diesen namentlichen Zusammenhang, indem das Array `$daten` in einer `foreach`-Schleife durchlaufen wird. Von jedem Schlüssel `$k` wird der Anfangsbuchstabe in einen Großbuchstaben umgewandelt (`ucfirst`) und davor der String `set` ergänzt. Aus dem Schlüssel `vorname` wird so also der Methodenname `setVorname`.

Mit der Funktion `method_exists()` können Sie prüfen, ob eine Methode mit einem bestimmten Namen in einem Objekt existiert. In unserem Beispiel wird in Zeile 15 also bestätigt, dass es die Methode `setVorname` in dem aktuellen Objekt gibt.

Handelt es sich um eine gültige Methode, so rufen wir diese in Zeile 16 auf und übergeben den Array-Wert `$v` als Parameter (`$this->$setterName($v)`). Hierfür nutzen wir das Konzept der ***Variablenfunktionen***: Wenn Sie an das Ende einer Variablen Klammern hängen, versucht PHP eine Funktion aufzurufen, deren Name der aktuelle Wert der Variablen ist. Dies gilt natürlich auch für die Methoden eines Objektes, wenn wir noch `$this` und den Pfeiloperator davorschreiben.

Die Konstruktor-Methode durchläuft also das Array `$daten`, und wenn es einen Setter passend zum Schlüssel gibt, wird dieser aufgerufen. Schlüssel, zu denen es kein Attribut im Objekt gibt, werden einfach ignoriert.

Allerdings löst unser schöner neuer Konstruktor nicht alle Probleme. Häufig muss man ein existierendes Objekt aktualisieren.

Beispiel

```php
<?php

require_once 'person_konstruktor_array.php';

$person = new Person(
    array(
        'vorname' => 'Marc',
        'nachname' => 'Remolt',
    )
);

//hier passieren viele Dinge ...

$formularDaten = array(
    'vorname' => 'Jan',
    'nachname' => 'Teriete',
);

$person->setVorname($formularDaten['vorname']);
$person->setNachname($formularDaten['nachname']);

echo $person;
```

neue_person.php (Version 5)

Codebeispiel 67 *neue_person.php (Version 5)*

Wie im Beispiel veranschaulicht, würde man für diese Aktualisierung weiterhin

jeden Setter einzeln aufrufen. Nicht wirklich schön, oder?

Übung 23:

1. Sie haben in dieser Lektion gelernt, wie Sie alle Setter aufrufen können, ohne dass Sie jeden Aufruf einzeln notieren müssen. Überlegen Sie, wie das ging.
2. Probieren Sie es nun aus. Ist der entstandene Code lesbar?

Wenn Sie sich an den Abschnitt 7.3.3 zurückerinnern, so kennen Sie eine Schreibweise für den manuellen Aufruf einer magischen Methode. In diesem Fall benötigen Sie den Konstruktor, der dann automatisch jeden Setter aufruft, zu dem ein Array-Schlüssel existiert.

Allerdings wäre Ihr Code nicht besonders gut lesbar, wenn Sie für eine Aktualisierung von Objekt-Daten jedesmal den Konstruktor aufrufen würden. Zudem besteht die Möglichkeit, dass der Konstruktor weitere Aufgaben neben dem Aufruf der Setter wahrnimmt. Doch wie lässt sich dieses Problem lösen?

Beispiel

```php
<?php

class Person
{
    protected $vorname = '';
    protected $nachname = '';

    public function __construct(array $daten = array())
    {
        $this->setDaten($daten);
    }

    public function setDaten(array $daten)
    {
        // wenn $daten nicht leer ist, rufe die passenden Se
        if ($daten) {
            foreach ($daten as $k => $v) {
                $setterName = 'set' . ucfirst($k);
                // pruefe ob ein passender Setter existiert
                if (method_exists($this, $setterName)) {
                    $this->$setterName($v); // Setteraufruf
```

```
22                      }
23                  }
24              }
25      }
26
27      public function getVorname()
28      {
29          return $this->vorname;
30      }
31
32      public function setVorname($vorname)
33      {
34          $this->vorname = $vorname;
35      }
36
37      public function getNachname()
38      {
39          return $this->nachname;
40      }
41
42      public function setNachname($nachname)
43      {
44          $this->nachname = $nachname;
45      }
46  }
```

person_set_daten.php

***Codebeispiel 68** person_set_daten.php*

Code, der eine klar definierte Aufgabe hat (z. B. Aufruf der Setter auf Basis des assoziativen Arrays), lässt sich normalerweise auslagern (Stichwort ***Single Responsibility Principle***). In diesem Fall soll diese Funktionalität fest an die Objekte der `Person`-Klasse gebunden sein, weswegen wir den Code in die Methode `setDaten()` auslagern und diese im Konstruktor in Zeile 10 aufrufen. Wann immer Sie die Daten eines bestehenden Objekts aktualisieren wollen, können Sie fortan diese Methode nutzen.

7.4 Testen Sie Ihr Wissen

1. Was versteht man unter einer ***magic method***?
2. Wann wird die Methode `__toString()` eines Objektes automatisch aufgerufen?

3. Wann wird die Methode `__construct()` eines Objektes automatisch aufgerufen?

4. Benötigt jede Klasse eine Konstruktor-Methode?

5. Sie haben ein `Person`-Objekt in der Variable `$person` und möchten dessen Inhalt als Text ausgeben. Die in dieser Lektion vorgestellten magischen Methoden sind vorhanden. Was können Sie aufrufen, wenn Sie keine Getter verwenden dürfen?

7.5 Aufgaben zur Selbstkontrolle

Übung 24:

Erweitern Sie die Klasse `Fussball` aus den Aufgaben von Lektion 5, so dass sie über einen Konstruktor verfügt, dem die Attribute des Fussballs direkt übergeben werden können. Verwenden Sie mehrere Parameter. Erzeugen Sie mehrere Bälle und geben Sie deren Beschreibung mittels `beschreibeFussball()` aus.

Übung 25:

Ersetzen Sie die Methode `beschreibeFussball()` durch die magische Methode `__toString()`. Passen Sie auch die Ausgabe in der *index.php* an.

7.6 Optionale Aufgaben

Übung 26:

Ändern Sie die Klasse `Fussball` und verwenden Sie nun einen einzelnen Parameter als Array. Erweitern Sie hierfür die Klasse um die Methode `setDaten()`.

8 Beziehungen zwischen Objekten

In dieser Lektion lernen Sie
• dass Sie als Methoden-Parameter ganze Objekte benutzen sollten. • wie Sie auf Objekte in anderen Objekten elegant zugreifen können.

8.1Das Problem

Sie wissen zwar inzwischen, dass Sie Objekte in anderen Objekten ablegen können, kennen aber noch keine elegante Lösung, um auf diese Objekte wieder zuzugreifen. Eine Möglichkeit ist natürlich, das Objekt wieder herauszuholen und dann auf die einzelnen Attribute zuzugreifen. Das ist aber recht viel Arbeit, wenn Sie nur ein einzelnes Attribut benötigen.

8.2 Objekte in anderen Objekten verstecken

Eine einfache Lösung, um an die Attribute von derartigen Objekten heranzukommen, haben Sie in ähnlicher Form in Lektion 6 bereits kennengelernt. Wir gaukeln den Außenstehenden vor, die fraglichen Attribute gehörten zur Klasse des Haupt-Objekts, indem wir für die Attribute Getter und Setter schreiben. Von außen sieht es also aus, als gäbe es nur ein einziges Objekt mit vielen Attributen.

Es handelt sich hierbei um eine sogenannte ***Delegation***. Der Begriff besagt, dass die Implementierung von Methoden eines Objekts durch Aufruf äquivalenter Methoden eines anderen Objekts umgesetzt wurde, die eigentliche Erfüllung einer Aufgabe also an ein anderes Objekt delegiert wird (siehe https://de.wikipedia.org/wiki/Delegation_%28Softwareentwicklung%29). In der ersten Klasse werden lediglich die Methoden eines Objekts einer anderen Klasse aufgerufen, eine Kenntnis über die genaue Implemetierung der eigentlichen Funktionalität in der zweiten Klasse besteht jedoch nicht.

Beispiel

```php
<?php

class Autor
{
    protected $name = '';

    public function getName()
    {
        return $this->name;
    }

    public function setName($name)
    {
        $this->name = $name;
    }
}

class Buch
{
    protected $titel = '';
    protected $autor; // kein Standardwert

    public function getTitel()
    {
        return $this->titel;
    }

    public function setTitel($titel)
    {
        $this->titel = $titel;
    }

    public function getAutorName()
    {
        return $this->autor->getName();
    }

    public function setAutorName($name)
    {
        $this->autor->setName($name);
    }
}

$buch = new Buch();
$buch->setTitel('Der Dativ ist dem Genetiv sein Tod');
$buch->setAutorName('Bastian Sick'); // Erzeugt eine Fehlerme
```

buch.php - Fehler (Version 1)

Codebeispiel 69 *buch.php - Fehler (Version 1)*

Sehen Sie sich das Beispiel etwas genauer an. Es werden zwei Klassen definiert: `Buch` und `Autor`. Aus der realen Welt wissen wir, dass ein Buch einen Autor hat. Genauso bilden wir das in PHP ab. Ein `Autor`-Objekt wird als Attribut `$autor` in einem `Buch`-Objekt abgelegt. Dieses Attribut erhält jedoch keinen Standardwert. Diesen vergeben wir nur bei skalaren Datentypen oder Arrays.

Wollen wir auf den Namen des Autors zugreifen, geschieht das über ein virtuelles Attribut in `Buch`, dessen Getter und Setter die entsprechenden Methoden in `$autor` aufrufen. In Zeile 35 und 40 sehen Sie etwas Neues. In den Ausdrücken befinden sich zwei Pfeil-Operatoren. Wenn Sie ein wenig darüber nachdenken, macht das allerdings Sinn.

Der Pfeil trennt das Objekt vom Attribut oder von der Methode. In Zeile 35 ist `$this` das Objekt und `$autor` das Attribut. Da aber `$autor` wiederum ein Objekt ist bzw. enthält, können Sie mit einem weiteren Pfeil auf die Methode `getName()` in `$autor` zugreifen. Falls Sie sich das gerade fragen: Ja, technisch gesehen können Sie Objekte beliebig tief verschachteln und mehr als zwei Pfeile verwenden.

Ein Prinzip namens ***Law of Demeter*** besagt, dass Objekte nur mit ihren unmittelbaren Nachbarn kommunizieren, also an möglichst wenigen Stellen auf weiter entfernte Nachbarn zugreifen sollten. Je mehr Verbindungen es zwischen Klassen gibt, desto größer wird nämlich der Aufwand bei Änderungen. LoD ist eine sogenannte Entwurfs-Richtlinie in der objektorientierten Softwareentwicklung und wurde 1987 an der Northeastern University in Boston vorgeschlagen.

Soweit ist eigentlich alles in Ordnung, aber in Zeile 46 erzeugt PHP einen hässlichen Fehler.

Übung 27:

1. Schreiben Sie das PHP-Skript *buch.php* aus dem letzten Beispiel ab.
2. Interpretieren Sie die Fehlermeldung, die das Skript erzeugt.

3. Überlegen Sie, wodurch das Problem verursacht wird und wie man es umgehen könnte. Sie müssen diese Idee jedoch nicht ausprogrammieren.

Haben Sie herausgefunden, wo der Fehler liegt? Wann genau haben wir eigentlich in das Attribut `$autor` ein `Autor`-Objekt gelegt? Richtig, gar nicht. Wenn Sie also in Zeile 46 versuchen, eine Methode auf `$autor` aufzurufen, schlägt das fehl, weil sich dort noch kein `Autor` befindet. Das Problem lösen Sie am besten, indem Sie schon im Konstruktor von `Buch` dafür sorgen, dass ein `Autor`-Objekt angelegt wird. Auf diese Weise können Sie es nicht vergessen.

Beispiel

```
 1 <?php
 2
 3 require_once 'autor.php'; //Autor wurde ausgelagert
 4
 5 class Buch
 6 {
 7     protected $titel = '';
 8     protected $autor; // kein Standardwert
 9
10     public function __construct()
11     {
12         $this->autor = new Autor();
13     }
14
15     public function getTitel()
16     {
17         return $this->titel;
18     }
19
20     public function setTitel($titel)
21     {
22         $this->titel = $titel;
23     }
24
25     public function getAutorName()
26     {
27         return $this->autor->getName();
28     }
29
30     public function setAutorName($name)
31     {
```

```
32          $this->autor->setName($name);
33      }
34  }
35
36  $buch = new Buch();
37  $buch->setTitel('Der Dativ ist dem Genetiv sein Tod');
38  $buch->setAutorName('Bastian Sick');
39
40  var_dump($buch);
```

buch.php (Version 2)

Codebeispiel 70 *buch.php (Version 2)*

Nun sollte der Code ohne Fehler funktionieren.

8.3 Ganze Objekte als Parameter übergeben

Dass Sie Objekte als Parameter in Methoden-Aufrufen übergeben können, wissen Sie bereits. Daher möchte ich Ihnen an dieser Stelle nur noch einen guten Rat geben: So lange Sie keinen triftigen Grund haben, es anders zu handhaben, übergeben Sie immer ganze Objekte als Parameter, auch wenn Sie in einer Methode nur ein einzelnes Attribut benötigen.

Auch hier gilt wieder das Prinzip der Kapselung, also Funktionalität in den Klassen zu verstecken. Von außen muss nicht sichtbar sein, welches Attribut die Methode benötigt. Sie müssen nur wissen, welches Objekt benötigt wird. Stellen Sie sich vor, Sie schreiben die Methode später um und nun benötigt sie zwei Attribute. Haben Sie nur das erste Attribut übergeben, müssen Sie jede Stelle, wo die Methode aufgerufen wird, ändern. Haben Sie das ganze Objekt übergeben, muss nur der Code in der Methode geändert werden.

Ein weiterer Vorteil dieser Vorgehensweise ist, dass Sie bei Objekten als Parameter Type-Hints (siehe Abschnitt 5.3.3) verwenden können, was Ihren Code noch weniger anfällig für Fehler macht.

> Schreiben Sie Ihren Code immer möglichst wartungsfreundlich. Das bedeutet, dass Änderungen an einer Stelle möglichst wenige zusätzliche Änderungen nach sich ziehen sollten. Je stärker Sie Ihren Code kapseln, desto robuster ist er gegen Änderungen.

8.4 Testen Sie Ihr Wissen

1. Wie können Sie ein Objekt in einem anderen Objekt ablegen?
2. Wie können Sie auf die Attribute des innen abgelegten Objekts zugreifen?
3. Warum sollten Sie bevorzugt ganze Objekte als Parameter an Methoden übergeben?

8.5 Aufgaben zur Selbstkontrolle

Übung 28:

Schreiben Sie zwei Klassen `Person` und `Adresse`. Die Klasse `Person` hat die Attribute `$name`, `$email` und `$adresse`. Die Klasse `Adresse` hat die Attribute `$strasse`, `$plz` und `$ort`. Vergessen Sie nicht die entsprechenden Getter und Setter. Ergänzen Sie eine *index.php*, in welcher Sie Instanzen beider Klassen erzeugen, diese mit Ausnahme des Attributs `$adresse` über die Setter befüllen und dann die fünf befüllten Attribute mittels Getter ausgeben. Da derzeit noch keine Beziehung zwischen den beiden Objekten besteht, müssen Sie für zwei Attribute auf das `Person`-Objekt zugreifen und für die restlichen drei Attribute auf das `Adresse`-Objekt.

Übung 29:

Modifizieren Sie die Klasse `Person` so, dass sie bei `setAdresse()` ein `Adresse`-Objekt erwartet. Befüllen Sie nun auch dieses Attribut in der *index.php*.

Übung 30:

Erweitern Sie `Person` um die Methoden `getStrasse()`, `getPlz()` und `getOrt()`, die ihre Informationen aus dem `Adresse`-Objekt erhalten. Verwenden Sie nun bei der Ausgabe in der *index.php* diese neuen Getter und somit nur noch das `Person`-Objekt.

8.6 Optionale Aufgaben

Übung 31:

Erweitern Sie `Person` um einen Konstruktor. Sorgen Sie in diesem dafür, dass immer ein leeres `Adresse`-Objekt für eine neue Person existiert.

Übung 32:

Erweitern Sie `Adresse` um das Attribut `$hausnummer`. Schreiben Sie hierfür passende Getter und Setter. Vergessen Sie nicht, eine Delegator-Methode `getHausnummer()` in `Person` anzulegen. Befüllen Sie das neue Attribut mittels Setter und geben Sie den Inhalt in der *index.php* aus.

Übung 33:

Erweitern Sie `Person` um die Methoden `setStrasse()`, `setHausnummer()`, `setPlz()` und `setOrt()`, die in dem `Adresse`-Objekt die passenden Setter aufrufen. Verwenden Sie in der *index.php* nun nur noch Getter und Setter der Klasse `Person`.

9 MVC

In dieser Lektion lernen Sie
• was sich hinter der Abkürzung MVC verbirgt. • wie Sie mit MVC den PHP-Code sauber vom HTML-Code trennen.

9.1Einleitung

Die Abkürzung ***MVC*** steht für ***Model-View-Controller*** und ist ein Muster zur Strukturierung von Software. 1979 wurde dieses Muster erstmals durch *Trygve Reenskaug* für Benutzeroberflächen in *Smalltalk* beschrieben. Viele Autoren stufen es als Entwurfsmuster ein, etliche andere jedoch als Architekturmuster (siehe auch https://de.wikipedia.org/wiki/Architekturmuster). Die genaue Einstufung ist jedoch meiner Meinung nach egal, es ist halt ein Muster, mit dem wir von der Erfahrung anderer Entwickler profitieren können.

9.1.1 MVC als Konzept

Model-View-Controller ist ein Konzept, welches für eine klare Trennung von Zuständigkeiten (engl. ***Separation Of Concerns***) sorgen soll. Wir als Entwickler sollen also Teile unserer Anwendungen separieren. Manche Teile beschäftigen sich beispielsweise mit der Präsentation, d.h. sie zeigen dem Benutzer bestimmte Dinge an. Einige Teile sorgen für eine Datenhaltung und wiederum andere Teile dienen der Benutzerinteraktion, d.h. sie werten die Eingaben eines Benutzers aus und sorgen für eine Weiterverarbeitung. Nebenbei gibt es aber noch eine ganze Reihe weiterer Dinge, die nicht direkt zu den gerade genannten Teilen passen (z. B. die Validierung von Benutzereingaben).

Das MVC-Muster teilt diese drei Zuständigkeiten in die Anwendungsebenen (engl. Layer) ***View***, ***Model*** und ***Controller*** auf. Es beschreibt hierbei vor allem die Interaktionen mit und zwischen diesen drei Ebenen sehr konkret. Allerdings

ist dieses Muster für unseren Code nicht 1:1 übertragbar, da es für Desktop-Applikationen entworfen wurde und wir mit PHP serverseitige Webanwendungen entwickeln, bei denen ein Browser mittels HTTP mit einem Webserver kommuniziert. Die Ausgabe wird hierbei auf dem Server in Form von HTML-Code vorbereitet und erst durch unseren Browser tatsächlich gerendert. Sobald der Webserver den HTML-Code ausgeliefert hat, kann man diesen mit PHP jedoch nicht mehr beeinflussen, da jede Antwort (HTTP-Response) an den Browser auch eine Anfrage des Browsers (HTTP-Request) voraussetzt (sogenannter ***Request-Response-Cycle***). Das MVC-Muster basiert jedoch genau auf der Annahme, dass eine nachträgliche Aktualisierung dieser Präsentation möglich ist und vom ***View***-Layer selbst initiiert werden kann.

Wann immer eine PHP-Anwendung oder ein PHP-Framework behauptet, ***Model-View-Controller*** zu verwenden, kann also nicht das ursprüngliche MVC-Muster gemeint sein. Stattdessen konzentrieren sich diese Umsetzungen auf eine klare Trennung der Zuständigkeiten.[2] **Es gibt hierbei jedoch nicht das eine MVC-Konzept für PHP** und dies ist auch das Problem, weswegen heutzutage jeder etwas anderes unter dem Begriff MVC versteht. Alle Implementierungen eines webbasierten MVC-Konzepts mit PHP unterscheiden sich in den Details, also was wo hingehört und wie die Anwendungsebenen interagieren. Ziel jedes MVC-Konzepts ist jedoch immer ein Codeaufbau, der die **Wartbarkeit** und **Anpassbarkeit** des Codes verbessert und eine **Wiederverwendbarkeit** einzelner Komponenten ermöglicht.

---

2 Ich möchte mich an dieser Stelle herzlichst bei *Anthony Ferrara* bedanken, der mir mit seinem Blog-Artikel http://blog.ircmaxell.com/2014/11/a-beginners-guide-to-mvc-for-web.html genau diesen Umstand verdeutlicht hat.

9.1.2 Unser MVC-Konzept

Das hier vorgestellte Konzept orientiert sich an dem PHP-Framework *Symfony 2* (siehe auch http://symfony.com/doc/current/book/from_flat_php_to_symfony2.html). Wichtig war mir eine klare und einfach umsetzbare Trennung der Zuständigkeiten, wie es auch bei Symfony der Fall ist (siehe http://fabien.potencier.org/what-is-symfony2.html). Bevor wir uns jedoch genauer mit der Implementierung beschäftigen, möchte ich eine Differenzierung der drei Anwendungsebenen

Model, ***View*** und ***Controller*** für webbasierte PHP-Anwendungen nach dem hier verwendeten Konzept vornehmen.

Model

Der ***Model***-Layer wird von einer Reihe von Klassen gebildet, deren Aufgabengebiete miteinander verzahnt sind. Zunächst einmal finden wir hier unsere bisherigen Datenklassen, welche primär der Datenhaltung und sekundär der Vorbereitung einer Persistierung dienen. Eine solche Speicherung kann beispielsweise in einer Datei oder in einer Datenbank erfolgen. Außerdem finden sich in diesem Layer unter anderem Klassen aus dem Gebiet der Validierung (Validator-Klassen), welche also der Überprüfung von Benutzereingaben vor der eigentlichen Persistierung dienen.

Der ***Model***-Layer enthält somit vor allem die darzustellenden Daten. Bei einer Buchverwaltung wären dies beispielsweise mehrere Datensätze mit Büchern in Form von Objekten. Der Layer müsste also zumindest eine Klasse für die Bücher enthalten.

Der ***Model***-Layer ist alleine nicht funktionsfähig.

View

Der ***View***-Layer ist für die Präsentation der Daten zuständig. Er legt also fest, was ein Besucher zu sehen bekommt, und generiert beispielsweise eine HTML-Seite mit dem Titel und Preis jedes Buches und Buttons zum Editieren der Datensätze. Das eigentliche Rendering übernimmt jedoch der Browser des Benutzers.

Der ***View***-Layer ist alleine nicht funktionsfähig bzw. sollte er nicht isoliert eingesetzt werden.

Controller

Der ***Controller***-Layer nimmt die Benutzeraktionen (z. B. Formulareingaben) entgegen, wertet die Requests aus und reagiert entsprechend auf diese. Er übernimmt somit die eigentliche Steuerung einer Anwendung. Hierbei koordiniert

der Layer die Datenhaltung, die Datenverarbeitung und die Datenverwaltung und übergibt die benötigten Daten (z. B. ein oder mehrere Objekte mit Büchern) zur Response-Generierung an den ***View***-Layer. Er bildet also eine Art Bindeglied zwischen ***Model*** und ***View***.

Wo und wie wird interagiert?

Benutzer interagieren per Browser-Request mit Controllern, Controller aktualisieren Modelle und übergeben diese an Views, welche dann eine Ausgabe vorbereiten, die der Browser des Benutzers als Response vom Server erhält und dem Benutzer präsentiert.

9.2 Das Beispielprojekt »hallo«

Nachfolgend werden wir uns die drei Layer unseres MVC-Konzepts etwas näher ansehen und zu diesem Zweck einige Übungen durchführen.

Übung 34:

Erstellen Sie in Ihrem Webserver-Verzeichnis einen neuen Ordner namens *hallo*. Oder legen Sie, falls Sie eine IDE wie beispielsweise PhpStorm, Netbeans oder Eclipse verwenden, ein neues Projekt namens *hallo* an.

Um Ihnen dabei zu helfen, die Übersicht bei den Aufgaben dieser Lektion zu behalten, habe ich hier eine Verzeichnisansicht des Beispielprojekts *hallo* für Sie:

```
hallo
-- src
---- Entities
------ Adresse.php
------ Person.php
-- templates
---- zeige_person.tpl.php
---- person.tpl.php
zeige_person.php
```

Verzeichnisansicht

***Codebeispiel 71** Verzeichnisansicht*

Beachten Sie bitte, dass dieses Beispielprojekt nur in den Aufgaben im Fließtext verwendet wird. In den Listings der Lektion wird ein anderes Beispiel behandelt. Teile dieses zweiten Beispiels kommen dann bei den Aufgaben am Ende der Lektion zum Einsatz.

9.3 Der Model-Layer

Grundsätzlich gilt, dass es sich beim ***Model***-Layer um klassenbasierten PHP-Code handelt, der nicht direkt im Browser aufgerufen wird bzw. werden kann. Dies sind zunächst einmal primär unsere bisherigen Datenklassen. Betrachten wir als Beispiel die Datenklasse eines Buches:

Beispiel

```php
<?php

class Buch
{
    protected $titel = '';
    protected $preis = 0; // Nettopreis

    public function __construct(array $daten = array())
    {
        $this->setDaten($daten);
    }

    public function setDaten(array $daten)
    {
        // wenn $daten nicht leer ist, rufe die passenden Se
        if ($daten) {
            foreach ($daten as $k => $v) {
                $setterName = 'set' . ucfirst($k);
                // pruefe ob ein passender Setter existiert
                if (method_exists($this, $setterName)) {
                    $this->$setterName($v); // Setteraufruf
                }
            }
        }
    }

```

```
27      public function getTitel()
28      {
29          return $this->titel;
30      }
31
32      public function setTitel($titel)
33      {
34          $this->titel = $titel;
35      }
36
37      public function getPreis()
38      {
39          return $this->preis;
40      }
41
42      public function setPreis($preis)
43      {
44          $this->preis = $preis;
45      }
46
47      public function getBruttoPreis()
48      {
49          $bruttoPreis = $this->getPreis() * 1.07;
50
51          return $bruttoPreis;
52      }
53
54      public function setBruttoPreis($bruttoPreis)
55      {
56          $this->setPreis($bruttoPreis / 1.07);
57      }
58  }
```

src/Entities/Buch.php

Codebeispiel 72 *src/Entities/Buch.php*

An dieser Klassendefinition sollte eigentlich nichts Besonderes mehr sein. Allerdings wird diese (im Gegensatz zu unserer bisherigen Vorgehensweise) in einem eigenen Verzeichnis abgelegt, um sie von den PHP-Dateien abzugrenzen, die direkt aufgerufen werden. In realen PHP-Projekten werden hierfür Verzeichnisse mit Namen wie *lib*, *inc*, *includes* oder auch *src* benutzt. Da wir zukünftig Klassen mit verschiedenen Aufgabenbereichen haben werden, benutzen wir für jeden Aufgabenbereich einen eigenen Unterordner.

Der Model-Layer wird immer in Form von ***Klassen*** realisiert, nicht jede Klasse ist jedoch ein Modell. Unsere bisherigen Datenklassen gehören zu

diesem Layer und werden ab jetzt im Verzeichnis *src/Entities* abgelegt. Funktions-Bibliotheken (z. B. *funktionen.inc.php*) gehören nicht zum Model-Layer. Um genau zu sein, sind sie ja nicht einmal ***Klassen***. Sie werden ab jetzt separat im Verzeichnis *inc* abgelegt.

Doch nicht nur der Ordner ist neu, wir werden die Dateinamen von Klassendefinitionen fortan genau so schreiben, wie die Klasse benannt ist. Hierbei muss auch die genaue Schreibweise, d.h. Groß- und Kleinbuchstaben, berücksichtigt werden. Somit liegt unsere Beispielklasse `Buch` in einer Datei *Buch.php*.

Beispiel

```php
<?php

// Emuliert ein "SELECT * FROM buecher"

function holeBuecher()
{
    $buecher = array();
    $buecher[] = new Buch(
        array(
            'titel' => 'Der Graf von Monte Christo',
            'preis' => 9.95,
        )
    );
    $buecher[] = new Buch(
        array(
            'titel' => 'Per Anhalter durch die Galaxis',
            'preis' => 7.95,
        )
    );

    return $buecher;
}

// Emuliert ein "SELECT * FROM buecher WHERE id=:id"

function holeBuch($id)
{
    $buecher = holeBuecher();

    $buch = null;
    if (isset($buecher[$id])) {
```

```
32          $buch = $buecher[$id];
33      }
34
35      return $buch;
36  }
```

inc/emulator.inc.php

Codebeispiel 73 *inc/emulator.inc.php*

Da Sie noch keine Möglichkeit zum Erzeugen von Objekten aus Datenbankinhalten kennengelernt haben, emulieren wir mit den beiden dargestellten Funktionen die zwei gängigsten Varianten (**hole alle** Bücher-Datensätze und **hole einen** Datensatz anhand seiner ID). Dies ist aber wirklich nur eine **vorübergehende Behelfsmaßnahme**.

Wenn man **alle** Datensätze holt, so geschieht dies normalerweise in Form eines Arrays, dessen Werte Objekte sind. Wir erhalten also von der Funktion `holeBuecher` ein Array mit `Buch`-Objekten als Rückgabewert.

Hinweis: Mit der Persistierung in einer Datei werden wir uns am Ende dieser Lektion beschäftigen. In Band 2 dieses OOP-Lernbuchs werden wir die Datenklassen mittels sogenannter Annotationen zu Entity-Klassen erweitern und die objektorientierte Umsetzung einer Persistierung in einer Datenbank am Beispiel der Bibliothek *Doctrine 2* behandeln. Außerdem lernen Sie in Band 2 auch, wie man die Benutzereingaben zuvor validiert.

Übung 35:

1. Legen Sie im Projekt-Verzeichnis *hallo* den Ordner *src/Entities* für die Datenklassen an.
2. Kopieren Sie die Dateien *adresse.php* und *person.php*, die Sie in den Aufgaben zu Lektion 8 erstellt haben, in diesen Ordner. Die Umsetzung der optionalen Aufgaben ist hierfür nicht zwingend nötig. Berücksichtigen Sie jedoch die neuen Vorgaben bei der Benennung dieser beiden Dateien.

9.4 Templates

Wenn wir bisher Daten mittels PHP im Browser anzeigen wollten, war unsere Vorgehensweise wie folgt:

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/emulator.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 $buch = holeBuch(0);
 7
 8 ?>
 9 <!DOCTYPE html>
10 <html>
11
12 <head>
13     <meta charset="utf-8" />
14     <title>Ein klasse Buch</title>
15 </head>
16
17 <body>
18     <h2 class="titel"><?php echo $buch->getTitel(); ?></h2>
19
20     <p class="preis"><?php echo $buch->getPreis(); ?></p>
21 </body>
22
23 </html>
```

zeige_buch.php (Version 1)

***Codebeispiel 74** zeige_buch.php (Version 1)*

Das einzig Besondere an diesem Code ist Zeile 6, wo wir den anzuzeigenden Datensatz aus dem Array `$buecher` auswählen. Dies geschieht auf Basis des Array-Schlüssels (hier `0`).

9.4.1 Vollständige Templates

Was Sie wahrscheinlich noch nicht gemacht haben, ist, den HTML-Teil Ihrer PHP-Datei auszulagern. Dies sollte jedoch rein technisch keinen Unterschied zum Auslagern anderer Code-Fragmente darstellen.

Beispiel

```php
<?php

require_once 'inc/emulator.inc.php';
require_once 'src/Entities/Buch.php';

$buch = holeBuch(0);

require_once 'templates/zeige_buch.tpl.php';
```

zeige_buch.php (Version 2)

Codebeispiel 75 *zeige_buch.php (Version 2)*

```php
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Ein klasse Buch</title>
</head>

<body>
    <h2 class="titel"><?php echo $buch->getTitel(); ?></h2>

    <p class="preis"><?php echo $buch->getPreis(); ?></p>
</body>

</html>
```

templates/zeige_buch.tpl.php

Codebeispiel 76 *templates/zeige_buch.tpl.php*

Wir haben nun den kompletten HTML-Code ausgelagert. Sie sehen, dass in der Datei mit den HTML-Elementen nur noch `echo`-Anweisungen stehen. Ansonsten handelt es sich um eine normale HTML-Datei.

Diese Art von HTML-Datei bezeichnet man auch als ***Template*** (dt. Vorlage). Sie arbeitet ähnlich wie eine Vorlage, wie Sie sie eventuell aus Textverarbeitungen kennen. Der größte Teil ist statischer Text, nur an einigen Stellen gibt es Platzhalter, wo der veränderliche Teil stehen wird, z. B. bei einem Serienbrief der Name des Empfängers.

> Ein Template ist eine Datei, die statischen Text und dazu ein paar Platzhalter enthält. Diese Platzhalter werden im fertigen Dokument durch den eigentlichen Text ersetzt.

Die Templates sollten Sie in einen Unterordner namens *templates* auslagern, um Ihr Projekt übersichtlicher zu machen. Beachten Sie zudem die Datei-Endung `.tpl.php`. Wird eine Datei in PHP per `include(_once)` oder `require(_once)` eingebunden, so ist es egal, welche Endung sie hat. Nur eine PHP-Datei, die im Browser aufgerufen wird, muss technisch gesehen auf `.php` enden (oder eine andere Datei-Endung aufweisen, die der Webserver als PHP-Datei ansieht). Theoretisch wäre somit auch die Datei-Endung `.tpl.html` oder `.xyz` denkbar. Hier gäbe es in einigen Editoren jedoch Probleme mit dem Syntax-Highlighting. Durch die Endung `.tpl.php` sehen Sie (und Ihr Designer) auf den ersten Blick, dass es sich um ein HTML-**T**em**pl**ate handelt. Sie können Ihrem Designer sogar sagen, dass er sich nur noch um Dateien mit dieser Endung zu kümmern braucht.

9.4.2 Teil-Templates

Stellen Sie sich vor, dass wir zusätzlich noch eine Liste von Büchern anzeigen wollen.

Beispiel

```php
<?php

require_once 'inc/emulator.inc.php';
require_once 'src/Entities/Buch.php';

$buecher = holeBuecher();

require_once 'templates/zeige_buecher.tpl.php';
```

zeige_buecher.php

***Codebeispiel 77** zeige_buecher.php*

```html
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Buecherliste</title>
</head>

```

```
 9 <body>
10     <?php foreach ($buecher as $buch): ?>
11         <h2 class="titel"><?php echo $buch->getTitel(); ?></h
12         <p class="preis"><?php echo $buch->getPreis(); ?></p
13     <?php endforeach; ?>
14 </body>
15
16 </html>
```

templates/zeige_buecher.tpl.php (Version 1)

Codebeispiel 78 *templates/zeige_buecher.tpl.php (Version 1)*

Bei diesem Beispiel haben wir zwar in der *zeige_buecher.php* eine saubere **Dreiteilung des Codes** (als PHP-Anfänger haben Sie dies vermutlich gelernt: Zuerst werden immer die Funktionen definiert, dann folgt die Programmlogik und zuletzt wird die Ausgabe erzeugt), und im Template *zeige_buecher.tpl.php* stehen auch alle HTML-Elemente sauber außerhalb von PHP-Tags; trotzdem gibt es eine unschöne Stelle im Code. Der Code-Block innerhalb der `foreach`-Schleife ist nämlich identisch zu einem Codefragment in der *zeige_buch.tpl.php*. In diesem Beispiel handelt es sich zwar nur um zwei Zeilen Code, dies ist in der Realität aber eher selten der Fall (Autor, Beschreibung, ISBN, Kundenmeinungen usw.).

Beispiel

```
 1 <!DOCTYPE html>
 2 <html>
 3
 4 <head>
 5     <meta charset="utf-8" />
 6     <title>Buecherliste</title>
 7 </head>
 8
 9 <body>
10     <?php foreach ($buecher as $buch): ?>
11         <?php require 'buch.tpl.php'; ?>
12     <?php endforeach; ?>
13 </body>
14
15 </html>
```

templates/zeige_buecher.tpl.php (Version 2)

Codebeispiel 79 *templates/zeige_buecher.tpl.php (Version 2)*

```
1 <h2 class="titel"><?php echo $buch->getTitel(); ?></h2>
2 <p class="preis"><?php echo $buch->getPreis(); ?></p>
```

templates/buch.tpl.php
Codebeispiel 80 *templates/buch.tpl.php*

Zur Verbesserung haben wir den Zweizeiler in ein eigenes ***Template*** ausgelagert. Da es sich bei diesem Template nur um ein kleines Fragment einer HTML-Seite handelt, bezeichnet man es auch als Teil-Template (engl. ***partial***).

Welchen Vorteil bringt uns diese zusätzliche Auslagerung? Zum einen hat der Designer nun eine sehr kleine und übersichtliche Datei für die Anzeige eines Buches, zum anderen können wir diese Datei nun überall wiederverwenden und beispielsweise auch die Stelle in der *zeige_buch.tpl.php* durch ein `require` ersetzen.

> Achten Sie beim Einsatz von Teil-Templates möglichst darauf, immer ein `require` und kein `require_once` zu benutzen. In letzterem Fall würde nämlich das Teil-Template innerhalb einer Schleife nur beim ersten Durchlauf eingebunden. Ein konsequenter Einsatz von `require` ist deswegen meiner Meinung nach bei Partials sinnvoll, um dieses Schleifenproblem direkt im Ansatz zu vermeiden.

9.5 Der View-Layer

Der ***View***-Layer regelt, was ein Besucher zu sehen bekommt. Hierbei kommt eines oder mehrere Templates zum Einsatz. Die Präsentation der Daten muss jedoch nicht zwingend als HTML-Seite erfolgen, einfacher Text oder JSON (siehe https://de.wikipedia.org/wiki/JavaScript_Object_Notation) ist beispielsweise auch möglich.

In den verwendeten ***Templates*** sollte möglichst wenig PHP-Code zu finden sein. Im Idealfall wären das nur `echo`-Anweisungen, allerdings sind auch einfache Schleifen und Verzweigungen in Ordnung, wenn es die Anwendung erfordert. Sollten Sie sich unsicher sein, ob ein Code-Fragment in ein ***Template*** gehört, fragen Sie sich, ob es wirklich nur zur Ausgabe von Daten dient. Sollte es etwas anderes tun, wie beispielsweise einzelne Datensätze mit einer `if`-Abfrage ausfiltern, so gehört dieser Code sehr wahrscheinlich eher in den

Controller-Layer. Eine solche Code-Verschiebung kann jedoch durchaus einen komplett anderen Ansatz erfordern. So ist beispielsweise eine Datensatz-Filterung direkt auf MySQL-Ebene auf jeden Fall einer späteren Filterung mit `if`-Abfragen im Template vorzuziehen.

In diesem Lernbuch wird fortan der Ordner *templates* und die Dateiendung `.tpl.php` für alle Templates verwendet.

Übung 36:

1. Erstellen Sie im Projekt-Verzeichnis *hallo* den neuen Ordner *templates*.
2. Legen Sie in diesem Ordner zwei HTML-Templates namens *zeige_person.tpl.php* und *person.tpl.php* an.
3. Das Template *zeige_person.tpl.php* soll das Teil-Template *person.tpl.php* einbinden.
4. Das Teil-Template *person.tpl.php* soll derzeit noch nichts ausgeben.

9.6 Der Controller-Layer

Auch der ***Controller***-Layer ist an sich nichts Besonderes. Er wird primär von den PHP-Dateien, die direkt im Browser aufgerufen werden, gebildet. Diese Controller-Dateien liegen deswegen meist direkt im Hauptordner des Projekts. Sie beschaffen beispielsweise alle Daten, die zur Anzeige einer HTML-Seite notwendig sind, und binden danach ein oder mehrere Templates ein, um die Daten auch tatsächlich auszugeben. Es kann jedoch beispielsweise im Falle einer Header-Umleitung auch Controller ohne jegliche Template-Einbindung geben. Somit haben wir alleine in dieser Lektion bereits zwei Controller, nämlich *zeige_buch.php* und *zeige_buecher.php*, kennengelernt. Wenn Sie sauber gearbeitet und möglichst viel Code ausgelagert haben, so sollten diese Controller relativ kurz und gut lesbar sein.

Übung 37:

1. Erstellen Sie im Projekt-Verzeichnis *hallo* den Controller *zeige_person.php*.
2. Der Controller soll ein Objekt `$person` instanziieren, dieses mit **allen** Daten (inklusive einer Adresse) befüllen und dann das Template *zeige_person.tpl.php* einbinden. Falls Sie den Stand der `Person`-Klasse verwenden, die noch keine Setter für die virtuellen Attribute verwendet, müssen Sie natürlich zusätzlich ein `Adresse`-Objekt im Controller instanziieren.
3. Sorgen Sie nun im Teil-Template *person.tpl.php* für die Ausgabe dieser Daten.
4. Rufen Sie abschließend den Controller im Browser auf und testen Sie, ob alles funktioniert. Dies geht normalerweise über eine URL in der Form `http://meinserver/hallo/zeige_person.php`. Wichtig ist hierbei vor allem die Angabe des korrekten Hosts (lokal meist `localhost`) und des korrekten Unterordners (hier *hallo*).

9.6.1 Controller mit Aktionen

Wie gesagt sind viele Controller-Dateien eher kurz, und ein erheblicher Teil des Codes ist auch noch in jedem Controller gleich:

- Zunächst werden benötigte Klassen und Funktions-Bibliotheken eingebunden.
- Die Session wird gestartet (sofern diese tatsächlich genutzt wird).
- Eine Datenbankverbindung wird aufgebaut (machen wir erst bei Doctrine 2).
- Am Ende wird sofern nötig ein Template zur HTML-Ausgabe eingebunden.

Unser Code ist zwar schon erheblich besser als vorher, aber viele kleine Dateien mit sehr ähnlichem Code sind immer ein Hinweis auf mögliche Verbesserungen. Versuchen wir also, mehrere Controller zu einer Datei zusammenzufassen:

- Es wird eine neue Controller-Datei erstellt. Diese nennen wir *index.php*.

- Im Kopfbereich des Controllers werden wie üblich alle Dateien, die später benötigt werden, per `require_once` eingebunden. Neu ist, dass wirklich alle Dateien eingebunden werden müssen, d.h. jede von mindestens einem der ehemaligen Controller benötigte Datei.

Als Nächstes wäre die Datenverarbeitung an der Reihe. Doch wie soll der Controller wissen, welche persistierten Daten er beschaffen bzw. welche Daten er überhaupt erst persistieren und welches Template er danach einbinden soll?

An dieser Stelle muss aus mehreren Möglichkeiten eine Aktion ausgewählt werden, also müssen wir hier eine Verzweigung programmieren. Eine `if-elseif`-Anweisung wäre möglich, aber da es eine ganze Reihe von Aktionen geben kann, macht an dieser Stelle eine `switch`-Anweisung mehr Sinn.

Um dem Controller die Möglichkeit zu geben, zwischen den verschiedenen Aktionen auszuwählen, werden wir beim Seitenaufruf einen URL-Parameter namens `action` übergeben. Den Wert dieses Parameters können wir somit mittels `$_GET['action']` ermitteln und er teilt dem Controller zukünftig mit, welche Aktion ausgeführt werden soll.

Beispiel

```php
<?php

require_once 'inc/emulator.inc.php';
require_once 'src/Entities/Buch.php';

$action = isset($_GET['action']) ? $_GET['action'] : null;

switch ($action) {
    case 'zeige_buecher':
        $buecher = holeBuecher();
        require_once 'templates/zeige_buecher.tpl.php';
        break;
    case 'zeige_buch':
        $buch = holeBuch(0);
        require_once 'templates/zeige_buch.tpl.php';
        break;
}
```

index.php (Version 1)

Codebeispiel 81 *index.php (Version 1)*

In Zeile 6 befüllen wir die Variable `$action` auf Basis des URL-Parameters und berücksichtigen dabei auch den Fall, dass dieser nicht angegeben wurde. Hierfür verwenden wir den sogenannten Trinitäts-Operator (engl. ternary operator, siehe http://php.net/de/language.operators.comparison).

Wenn Sie nun die URL `http://meinserver/index.php?action=zeige_buecher` aufrufen, springt die `switch`-Anweisung an die entsprechende Stelle und zeigt das richtige Template an. Dasselbe passiert bei der URL `http://meinserver/index.php?action=zeige_buch`. Geben Sie auch hierbei den Namen des korrekten Unterordners an, sofern sich die *index.php* nicht direkt im Document Root (meist *htdocs*) Ihres Webservers befinden sollte.

Doch wir können noch Verbesserungen vornehmen. Es ist kein Zufall, dass die Aktion genau so benannt ist wie das Template. Das hat erstens den Vorteil, dass Sie sofort sehen, welches Template Sie für welche Aktion bearbeiten müssen, und zweitens können Sie sich mit einem kleinen Trick noch ein paar Zeilen Code sparen.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/emulator.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 $action = isset($_GET['action']) ? $_GET['action'] : null;
 7
 8 switch ($action) {
 9     case 'zeige_buecher':
10         $buecher = holeBuecher();
11         break;
12     case 'zeige_buch':
13         $buch = holeBuch(0);
14         break;
15 }
16
17 require_once 'templates/' . $action . '.tpl.php';
```

index.php (Version 2)

Codebeispiel 82 *index.php (Version 2)*

Wenn das Template immer den Namen der Aktion hat, können Sie nach dem

`switch` einfach ein Template einbinden, das so heißt wie der Inhalt der Variable `$action` mit dem Anhang `.tpl.php`.

9.6.2 Die Standard-Aktion

So wie der Controller *index.php* momentan aussieht, hat er noch ein Problem.

Übung 38:

1. Überlegen Sie, was passiert, wenn der Controller ohne Parameter aufgerufen wird, also beispielsweise nur mit `http://meinserver/index.php`.
2. Wie können Sie dieses Problem umgehen? Sie müssen Ihre Idee nicht umsetzen. Sollten Sie keine Idee haben, so lesen Sie einfach weiter.
3. Überlegen Sie, was passiert, wenn der Controller mit einem ungültigen Parameter aufgerufen wird, also z. B. `http://meinserver/index.php?action=ergrhrhtzhj`.
4. Kann Ihre Lösung aus 2. auch dieses Problem lösen?

Wenn die Controller-Datei ohne oder mit einer falschen Aktion aufgerufen wird, wird momentan nichts Sinnvolles angezeigt. Genau genommen wird sogar eine Fehlermeldung ausgegeben, dass eine Datei (das Template) mit diesem Namen nicht gefunden wurde. Das ist normalerweise nicht das, was Sie wollen. Zum Glück lässt sich das Problem recht einfach und elegant lösen. Die `switch`-Anweisung hat einen speziellen Fall `default` (die Standard-Aktion), der immer dann ausgeführt wird, wenn keiner der normalen Fälle zutrifft.

Beispiel

```php
<?php

require_once 'inc/emulator.inc.php';
require_once 'src/Entities/Buch.php';

$action = isset($_GET['action']) ? $_GET['action'] : null;
$template = $action;
```

```
 8
 9 switch ($action) {
10     case 'zeige_buecher':
11         $buecher = holeBuecher();
12         break;
13     case 'zeige_buch':
14         $buch = holeBuch(0);
15         break;
16     default:
17         $buecher = holeBuecher();
18         $template = 'zeige_buecher';
19         break;
20 }
21
22 require_once 'templates/' . $template . '.tpl.php';
```

index.php (Version 3 mit Standard-Aktion)

Codebeispiel 83 *index.php (Version 3 mit Standard-Aktion)*

Beachten Sie, dass in Zeile 7 eine neue Variable `$template` eingeführt wird. Mit dieser haben Sie die Möglichkeit, manuell ein Template festzulegen. Trotzdem haben Sie noch den Original-Wert der Aktion in `$action` zur Verfügung (z. B. für Fallunterscheidungen im Menü der Anwendung).

Im aktuellen Beispiel wird nun immer dann das Template `zeige_buecher` angezeigt, wenn keine oder eine ungültige Aktion übergeben wurde. Sie müssen nur die erforderlichen Daten beschaffen und in Zeile 18 die Variable `$template` auf das gewünschte Template (ohne Dateiendung) setzen.

Da nun für die Standard-Aktion und für die Aktion `zeige_buecher` die gleiche HTML-Seite angezeigt wird und hierfür sogar Code dupliziert werden musste, können Sie auf die ursprüngliche Aktion auch komplett verzichten.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/emulator.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 $action = isset($_GET['action']) ? $_GET['action'] : null;
 7 $template = $action;
 8
 9 switch ($action) {
10     case 'zeige_buch':
```

```
11          $buch = holeBuch(0);
12          break;
13      default:
14          $buecher = holeBuecher();
15          $template = 'zeige_buecher';
16          break;
17  }
18
19  require_once 'templates/' . $template . '.tpl.php';
```

index.php (Version 4 mit Standard-Aktion)

Codebeispiel 84 *index.php (Version 4 mit Standard-Aktion)*

Wenn Sie nun die URL `http://meinserver/index.php?action=zeige_buecher` aufrufen, wird Dank der Standard-Aktion immer noch die Liste angezeigt, obwohl es sich bei `zeige_buecher` nun um eine ungültige (da als Fall nicht mehr vorhandene) Aktion handelt. Um sich Schreibarbeit zu sparen, sollten Sie fortan nur noch die URL `http://meinserver/index.php` zur Anzeige dieser Bücherliste nutzen.

9.7 Zusammenfassung

MVC führt eine Aufteilung des PHP-Codes in drei Layer ein. Es gibt je einen Layer für die Modelle (z. B. Datenklassen und Validator-Klassen), für den oder die Controller (koordinieren die Reaktionen auf einen Request) und für die Views (bei uns erfolgt die Präsentation der angeforderten Daten mittels HTML-Templates). Durch diese Teilung werden Sie gerade in größeren Projekten weit besser den Überblick behalten und Ihren Code besser erweitern und warten können.

9.8 Testen Sie Ihr Wissen

1. Auf welche Sprachelemente sollte sich der PHP-Code in Templates beschränken?

2. Warum sollten Sie so wenig PHP wie möglich in Templates verwenden?

3. Welche Aufgabe hat ein Controller?
4. Wie kann der Controller entscheiden, welche Aktion er ausführen soll?
5. Warum sollte es eine Standard-Aktion geben?

9.9 Aufgaben zur Selbstkontrolle

Beispiel

```php
<?php

function holeBuecher()
{
    $buecher = array();
    $buecher[0] = new Buch(
        array(
            'titel' => 'Der Graf von Monte Christo',
            'preis' => 9.95,
        )
    );
    $buecher[1] = new Buch(
        array(
            'titel' => 'Per Anhalter durch die Galaxis',
            'preis' => 7.95,
        )
    );

    return $buecher;
}

function holeBuch($id)
{
    $buecher = holeBuecher();

    $buch = null;
    if (isset($buecher[$id])) {
        $buch = $buecher[$id];
    }

    return $buch;
}

function speichereBuch($daten)
{
```

```
36      // Das tut noch nichts.
37  }
```

inc/funktionen.inc.php

Codebeispiel 85 *inc/funktionen.inc.php*

Die Funktion `speichereBuch()` tut momentan noch nichts. Das ist so geplant.

```
 1  <!DOCTYPE html>
 2  <html>
 3
 4  <head>
 5      <meta charset="utf-8" />
 6      <title>Eintrag anlegen</title>
 7  </head>
 8
 9  <body>
10      <ul id="navi">
11          <li><a href="liste.php">Startseite</a></li>
12          <li><a href="neu.php">Buch anlegen</a></li>
13      </ul>
14
15      <form action="neu.php" method="post">
16          <input type="text" name="titel" id="titel" placeholde
17          <input type="text" name="preis" id="preis" placeholde
18          <input type="submit" value="speichern" />
19      </form>
20  </body>
21
22  </html>
```

neu.php

Codebeispiel 86 *neu.php*

Die Datei `neu.php` zeigt ein Formular an, in dem man die Daten eines Buches eingeben kann.

```
 1  <?php
 2
 3  require_once 'inc/funktionen.inc.php';
 4  require_once 'src/Entities/Buch.php';
 5
 6  $buecher = holeBuecher();
 7
 8  ?>
 9  <!DOCTYPE html>
10  <html>
11
12  <head>
13      <meta charset="utf-8" />
```

```
    <title>Buch anzeigen</title>
</head>

<body>
    <ul id="navi">
        <li><a href="liste.php">Startseite</a></li>
        <li><a href="neu.php">Buch anlegen</a></li>
    </ul>

    <ul>
        <?php foreach ($buecher as $id => $buch): ?>
            <li>
                <a
                    href="zeige.php?id=<?php echo $id; ?>"
                ><?php echo $buch->getTitel(); ?></a>
            </li>
        <?php endforeach; ?>
    </ul>
</body>

</html>
```

liste.php

Codebeispiel 87 *liste.php*

Die Datei `liste.php` dient zur Anzeige aller Bücher.

```
<?php

require_once 'inc/funktionen.inc.php';
require_once 'src/Entities/Buch.php';

$buch = holeBuch($_GET['id']);

?>
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Buch anzeigen</title>
</head>
<body>
    <ul id="navi">
        <li><a href="liste.php">Startseite</a></li>
        <li><a href="neu.php">Buch anlegen</a></li>
    </ul>

    <p>
        Titel: <?php echo $buch->getTitel(); ?><br/>
        Preis: <?php echo $buch->getPreis(); ?>
```

```
24      </p>
25  </body>
26
27  </html>
```

zeige.php

***Codebeispiel 88** zeige.php*

Die Datei `zeige.php` dient der Anzeige eines bestimmten Buches.

Übung 39:

Gegeben seien die Dateien aus dem Beispiel und die Datenklasse `Buch` aus der aktuellen Lektion. Legen Sie zunächst eine Projekt-Struktur mit den Ordnern *inc*, *src/Entities* und *templates* an. Dateien, die HTML-Code enthalten, gehören in den Ordner *templates*. Schreiben Sie den Code dieser Templates so um, dass er dem MVC-Pattern und unseren Vorgaben zur Dateibenennung entspricht. Die Navigation soll hierbei in ein Teil-Template ausgelagert werden.

Übung 40:

Schreiben Sie nun den restlichen Code um. Es soll einen Controller namens *index.php* geben. Die Aktionen heißen wie die Dateien vorher, also `neu`, `zeige` und `liste`. Die Aktion `liste` soll die Standard-Aktion werden. Modifizieren Sie Ihren Code entsprechend.

Übung 41:

Ergänzen Sie in der Aktion `neu` eine `if`-Abfrage. Sofern Formulardaten existieren, soll die Funktion `speichereBuch()` aufgerufen und dieser `$_POST` als Parameter übergeben werden. Nehmen Sie direkt nach diesem Aufruf eine Header-Umleitung auf die Standard-Aktion des Controllers vor. Vergessen Sie auch nicht, dass das Formular nun an diese Controller-Aktion verschickt werden muss.

Übung 42:

Setzen Sie die Funktion `speichereBuch()` um. Sie soll nun den Eintrag tatsächlich in einer Datei *buecher.txt* im Ordner *daten* speichern. **Alle Bücher liegen als Objekte in einem numerischen Array, das serialisiert in der Datei gespeichert wird.** Ermitteln Sie zunächst den bisherigen Stand des Arrays mit der Funktion `holeBuecher()` (wird in der nächsten Übung überarbeitet). Ergänzen Sie dann das neue Buch im Array und speichern danach das Array in der Datei. Benutzen Sie hierfür die Funktion `file_put_contents()`.

Hinweis: Beachten Sie bei der Umsetzung (sofern Sie kein Windows-Betriebssystem verwenden) bitte, dass PHP unter Umständen zusätzliche Schreibberechtigungen für die Datei mit den Daten benötigt.

Übung 43:

Überarbeiten Sie die Funktion `holeBuecher()`. Sie soll nun aus der Datei *buecher.txt* alle gespeicherten Bücher holen. Benutzen Sie hierfür die Funktion `file_get_contents()`.

Tipp: Falls Ihre Funktion eine Warnung erzeugt, könnte das daran liegen, dass die Datei *buecher.txt* noch nicht existiert. Sie haben in Lektion 2 gelernt, wie Sie Ihren Code mit einer entsprechenden Prüfung versehen.

10 Klassenbasierte Controller

In dieser Lektion lernen Sie
• welche Vorteile klassenbasierte Controller haben. • welche Änderungen an unserer Projektstruktur für ihren Einsatz nötig sind.

10.1Einleitung

In der letzten Lektion habe ich behauptet, dass der Einsatz von MVC einen besseren Überblick über den Code eines Projektes ermöglicht. Dies ist auch grundsätzlich korrekt. Unser bisheriger Controller *index.php* hat jedoch leider eine gravierende Schwachstelle. Mit steigendem Featureumfang eines Projektes wird dieser nämlich extrem lang. Zudem müssen Sie ständig Code in Funktionen auslagern, um diesen wiederverwenden zu können. Hierbei stoßen Sie, wie schon vor der Einführung der Methoden, auf das Problem, dass Sie diese wiederfinden und somit irgendwie sortieren müssen. Wir benötigen also einen Ersatz für die `switch`-Anweisung. Diese überarbeitete Lösung soll stattdessen Klassen und Methoden verwenden. Doch wie könnte eine solche Lösung umgesetzt werden?

10.2 Eine klassenbasierte Fallunterscheidung

10.2.1 Schritt 1

Wir werden im Folgenden das Beispielprojekt mit den Büchern aus den Übungen der letzten Lektion schrittweise weiterentwickeln. Streichen wir zunächst einmal den Switch aus unserem Controller und überlegen dann weiter.

Beispiel

```
1 <?php
```

```
 2
 3 require_once 'inc/funktionen.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 $action = isset($_GET['action']) ? $_GET['action'] : null;
 7 $template = $action;
 8
 9 // TODO: Klassenbasierter Ersatz fuer den Switch
10
11 require_once 'templates/' . $template . '.tpl.php';
```

index.php (Version 1)

Codebeispiel 89 *index.php (Version 1)*

Als Nächstes benötigen wir eine Klasse, in welche wir den weggefallenen Code auslagern können. Hierbei soll jede Aktion eine eigene Methode werden. Der Name jeder dieser drei Methoden soll mit `Action` enden und wird natürlich im lowerCamelCase notiert. Auf das normalerweise verpflichtende Verb im Namen verzichten wir übrigens in diesem speziellen Fall.

Beispiel

```
 1 <?php
 2
 3 class IndexController
 4 {
 5     public function indexAction()
 6     {
 7         $buecher = holeBuecher();
 8         $template = 'liste';
 9     }
10
11     public function zeigeAction()
12     {
13         $buch = holeBuch($_GET['id']);
14     }
15
16     public function neuAction()
17     {
18         if ($_POST) {
19             speichereBuch($_POST);
20             header('Location: index.php');
21             exit;
22         }
23     }
24 }
```

src/Controllers/IndexController.php (Version 1)

Codebeispiel 90 *src/Controllers/IndexController.php (Version 1)*

Im Falle von `'liste'` benutzen wir den allgemeineren Namen `indexAction`, welchen wir fortan immer für die Standard-Aktion verwenden werden. Der aktuelle Code ist natürlich so noch **nicht wirklich lauffähig** und muss erst noch weiter umgebaut werden.

10.2.2 Schritt 2

Nun benötigen wir in unserer *index.php* eine Möglichkeit, um abhängig vom Wert von `$action` die korrekte Methode unserer Controller-Klasse aufzurufen. Hierzu instanziieren wir zunächst einmal die Klasse.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/funktionen.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 require_once 'src/Controllers/IndexController.php';
 7 $requestController = new IndexController();
 8
 9 $action = isset($_GET['action']) ? $_GET['action'] : 'index'
10 $template = $action;
11 $methodName = $action . 'Action';
12
13 if (method_exists($requestController, $methodName)) {
14     $requestController->$methodName();
15 }
16
17 require_once 'templates/' . $template . '.tpl.php';
```

index.php (Version 2)

Codebeispiel 91 *index.php (Version 2)*

In Zeile 6 binden wir die neue Klasse per `require_once` ein, damit wir in Zeile 7 überhaupt erst die Möglichkeit haben, eine Instanz in der Variablen `$requestController` abzulegen. Danach verketten wir in Zeile 11 den Wert von `$action` mit dem String `'Action'`. Hierdurch erhalten wir den gewünschten Methodennamen, ohne den Inhalt unserer Templates anpassen zu müssen. Natürlich könnten wir stattdessen auch den `action`-Parameter in den Templates

anpassen. Die von mir verwendete Vorgehensweise hat jedoch den Vorteil, dass wir gleichzeitig eine Art Zugriffsschutz für unsere Controller-Methoden erhalten. Nur Methoden, die auf `Action` enden, können nämlich über die *index.php* aufgerufen werden.

Da wir ohne `switch` keine Standard-Aktion mehr nutzen können, müssen wir uns für den fehlenden URL-Parameter etwas Neues überlegen. So neu ist der benötigte Ansatz jedoch gar nicht, wir müssen lediglich in Zeile 9 die dritte Angabe des Trinitäts-Operators anpassen und dort den String `'index'` angeben. Wird also fortan kein URL-Parameter angegeben, so wird immer die Methode `indexAction` aufgerufen. Im Gegensatz zur bisherigen Lösung geschieht dies jedoch nicht bei Angabe einer falschen Aktion!

Der restliche Code sollte Ihnen bereits durch die Methode `setDaten()` bekannt vorkommen. Wir prüfen hier lediglich, ob die Methode in unserem Objekt existiert. Sofern eine gültige Methode existiert, rufen wir sie danach auch auf.

10.2.3 Schritt 3

Als Nächstes müssen wir gleich zwei Probleme lösen:

1. Wir benötigen für die Einbindung des Templates einen Wert für `$template`, den wir in den Controller-Methoden ändern können. Rufen wir derzeit die Startseite unserer Anwendung auf, so wird kein Template gefunden, da wir in der Methode lediglich eine lokale Variable `$template` befüllen. Diese Variable verfällt jedoch nach dem Abarbeiten der Methode `indexAction` ungenutzt.
2. Zwei unserer Templates benötigen Daten, die in unseren Controller-Methoden ermittelt werden. Auf diese fehlt jedoch aktuell noch der Zugriff.

Beschäftigen wir uns zunächst mit dem Aufruf des korrekten Templates. Dies könnte man genauso wie das zweite Problem theoretisch mit einem Rückgabewert in den drei Methoden lösen. Ich möchte jedoch möglichst viel Code in die Controller-Klasse verschieben und habe mich deswegen für einen anderen Ansatz entschieden.

Beispiel

```php
<?php

require_once 'inc/funktionen.inc.php';
require_once 'src/Entities/Buch.php';

require_once 'src/Controllers/IndexController.php';
$requestController = new IndexController();

$action = isset($_GET['action']) ? $_GET['action'] : 'index'
$methodName = $action . 'Action';

if (method_exists($requestController, $methodName)) {
    $requestController->$methodName();
}
```

index.php (Version 3)

Codebeispiel 92 *index.php (Version 3)*

Wie Sie sehen, habe ich die beiden Zeilen, in denen `$template` vorkam, komplett gelöscht. Ansonsten hat sich in der *index.php* nichts geändert.

Beispiel

```php
<?php

class IndexController
{
    protected $template = '';

    public function indexAction()
    {
        $buecher = holeBuecher();

        $this->setTemplate('liste');
        $this->render();
    }

    public function zeigeAction()
    {
        $buch = holeBuch($_GET['id']);

        $this->setTemplate('zeige');
        $this->render();
    }

    public function neuAction()
```

```
24      {
25          if ($_POST) {
26              speichereBuch($_POST);
27              header('Location: index.php');
28              exit;
29          }
30
31          $this->setTemplate('neu');
32          $this->render();
33      }
34
35      protected function setTemplate($template)
36      {
37          $this->template = $template . '.tpl.php';
38      }
39
40      protected function getTemplate()
41      {
42          return $this->template;
43      }
44
45      protected function render()
46      {
47          require_once 'templates/' . $this->getTemplate();
48      }
49  }
```

src/Controllers/IndexController.php (Version 2)

Codebeispiel 93 *src/Controllers/IndexController.php (Version 2)*

Die wichtigste Änderung an unserer Klasse ist das neue Attribut `$template` und die neue Methode `render()`. Außerdem wurde ein Setter und ein Getter für das neue Attribut ergänzt. Der Setter und die Methode `render()` wird am Ende jeder unserer bisherigen Methoden aufgerufen. Als Parameter bekommt der Setter dann mitgeteilt, welches Template eingebunden werden soll. Die doppelte Dateiendung wird hierbei automatisch ergänzt, da sie sich nie ändert. Diese Information legt der Setter dann im neuen Attribut ab. Von dort besorgt sich danach die Methode `render()` diese Information wieder und bindet die entsprechende Datei ein. Damit ist unser erstes Problem gelöst, auch wenn wir derzeit eigentlich einen Rückschritt in der Evolution unseres Codes vorgenommen haben, da wir das Template wieder manuell festlegen müssen. Doch darum kümmern wir uns später.

Nun wollen wir das zweite Problem lösen. Doch überlegen wir zunächst einmal, was wir hierfür erreichen müssen. Fest steht, dass wir den Inhalt unserer

Templates möglichst nicht ändern möchten. Wir benötigen also für das Template *liste.tpl.php* eine korrekt befüllte Variable `$buecher` und für das Template *zeige.tpl.php* eine Variable `$buch`. Da beide Templates in der Methode `render()` eingebunden werden, müssten die beiden Variablen in dieser Methode als lokale Variablen existieren.

Wir müssen also abhängig von der gewählten Aktion in der Methode `render()` unterschiedliche lokale Variablen definieren. Würden wir das `require_once` für die Template-Einbindung einfach in den Methoden unserer Aktionen notieren, so wäre dies unproblematisch. Dies ist jedoch nicht der Fall, da der Code zur besseren Wartbarkeit ausgelagert wurde. Wir stehen also vor dem Problem, dass wir eine lokale Variable über die Grenze zwischen zwei Methoden transferieren müssen.

Diese Anforderung hört sich zunächst ziemlich trivial an, da man die Information ja einfach in einem weiteren Parameter übergeben könnte. Dann müssten wir jedoch eine Art Fallunterscheidung in `render()` implementieren, welche die für die Aktion benötigten lokalen Variablen auf Basis dieses Parameters befüllt. Wir benötigen ja derzeit zwei unterschiedliche Variablen und es könnten zukünftig noch viel mehr werden. Zudem kann es sein, dass für ein Template mehrere Variablen benötigt werden. Dies ginge alles relativ einfach, wenn der neue Parameter immer einen Wert vom Datentyp Array hätte. Die Schlüssel in diesem Array müssten den Namen der gewünschten lokalen Variablen (ohne das `$`) entsprechen. Wenn dies der Fall wäre, so könnten wir die Funktion `extract()` (siehe http://php.net/de/function.extract) verwenden, um die benötigten Variablen aus dem Array zu erzeugen. Für Letzteres habe ich mich tatsächlich entschieden. Anstatt des Parameters nutze ich jedoch ein weiteres Attribut. Dies hat den Vorteil, dass ich das Array stückchenweise und sogar innerhalb unterschiedlicher Methoden befüllen kann.

Beispiel

```php
<?php

class IndexController
{
    protected $context = array();
    protected $template = '';
```

```php

    public function indexAction()
    {
        $this->addContext('buecher', holeBuecher());

        $this->setTemplate('liste');
        $this->render();
    }

    public function zeigeAction()
    {
        $this->addContext('buch', holeBuch($_GET['id']));

        $this->setTemplate('zeige');
        $this->render();
    }

    public function neuAction()
    {
        if ($_POST) {
            speichereBuch($_POST);
            header('Location: index.php');
            exit;
        }

        $this->setTemplate('neu');
        $this->render();
    }

    protected function setTemplate($template)
    {
        $this->template = $template . '.tpl.php';
    }

    protected function getTemplate()
    {
        return $this->template;
    }

    protected function addContext($key, $value)
    {
        $this->context[$key] = $value;
    }

    protected function render()
    {
        extract($this->context);

        require_once 'templates/' . $this->getTemplate();
    }
```

```
57 }
```

src/Controllers/IndexController.php (Version 3)

Codebeispiel 94 *src/Controllers/IndexController.php (Version 3)*

Die erste Neuerung finden Sie in Zeile 5, wo das neue Attribut `$context` definiert wird. Als Standardwert verwenden wir ein leeres Array. In den Zeile 46 bis 49 finden Sie dann die neue Methode `addContext()`. Diese macht eigentlich nichts anderes, als einen Wert zu einem assoziativen Array hinzuzufügen. Der erste Parameter bestimmt den Array-Schlüssel und somit später den Namen der lokalen Variable. Der zweite Parameter dient zur Festlegung des entsprechenden Wertes im Array und somit des späteren Variableninhalts. Diese Methode rufen wir dann in den Zeilen 10 und 18 auf. Die bisherige Angabe vor dem Zuweisungsoperator landet als String im ersten Parameter und die Anweisung hinter dem Zuweisungsoperator verwenden wir als Wert für den zweiten Parameter. Zur Erzeugung der lokalen Variablen aus dem Array haben wir außerdem in Zeile 53 den Aufruf der Funktion `extract()` ergänzt.

10.2.4 Schritt 4

Als Nächstes kümmern wir uns um den fehlenden Automatismus für die Template-Auswahl. Wir möchten zukünftig ein Template nur noch dann festlegen, wenn wir das einer anderen Aktion wiederverwenden wollen.

Beispiel

```
 1 <?php
 2
 3 class IndexController
 4 {
 5     protected $context = array();
 6     protected $template = '';
 7
 8     public function run($action)
 9     {
10         $this->addContext('action', $action);
11
12         $methodName = $action . 'Action';
13         $this->setTemplate($methodName);
14
15         if (method_exists($this, $methodName)) {
```

```php
            $this->$methodName();
        }

        $this->render();
    }

    protected function indexAction()
    {
        $this->addContext('buecher', holeBuecher());
    }

    protected function zeigeAction()
    {
        $this->addContext('buch', holeBuch($_GET['id']));
    }

    protected function neuAction()
    {
        if ($_POST) {
            speichereBuch($_POST);
            header('Location: index.php');
            exit;
        }
    }

    protected function setTemplate($template)
    {
        $this->template = $template . '.tpl.php';
    }

    protected function getTemplate()
    {
        return $this->template;
    }

    protected function addContext($key, $value)
    {
        $this->context[$key] = $value;
    }

    protected function render()
    {
        extract($this->context);

        require_once 'templates/' . $this->getTemplate();
    }
}
```

src/Controllers/IndexController.php (Version 4)

Codebeispiel 95 *src/Controllers/IndexController.php (Version 4)*

Bei der genaueren Betrachtung der Änderungen sollte auf jeden Fall die neue Methode `run()` auffallen. Diese bekommt als Parameter mitgeteilt, welche Aktion und somit welche weitere Methode von ihr intern aufgerufen werden soll. Der entsprechende Code entspricht weitestgehend unserer bisherigen Vorgehensweise in der *index.php*. Es gibt jedoch ein paar Extras, welche Sie in den Zeilen 10, 13 und 19 finden. In Zeile 10 sorgen wir dafür, dass in unseren Templates wieder die Variable `$action` verfügbar ist, damit wir in den Templates die Möglichkeit zu Fallunterscheidungen haben. Diese Möglichkeit benötigen wir nämlich relativ häufig, wenn ein Template von zwei Aktionen verwendet wird. In Zeile 13 finden Sie dann unseren neuen Automatismus. Nachdem der Methodenname ermittelt wurde, wird dieser als Name für das Template benutzt. Dies bedeutet, dass Sie alle Templates umbenennen müssen. Aus *liste.tpl.php* wird *indexAction.tpl.php*, aus *neu.tpl.php* wird *neuAction.tpl.php* und aus *zeige.tpl.php* wird *zeigeAction.tpl.php*. Der Dateiname muss somit nun 1:1 dem Namen der Methode entsprechen, d.h. lowerCamelCase und das Suffix `Action`. An der doppelten Dateiendung ändert sich hingegen nichts. Außerdem rufen wir in Zeile 19 die Methode `render()` auf, womit dies in den Methoden unserer drei Aktionen überflüssig ist. Auch das manuelle Festlegen eines Templates entfällt nun natürlich in den drei Methoden, da wir derzeit noch keines wiederverwenden wollen.

Wenn Sie den aktuellen Beispielcode mit der vorherigen Version vergleichen, so sollte Ihnen aber noch eine weitere Änderung auffallen. Oder haben Sie diese bereits entdeckt? Die Sichtbarkeit der drei Aktions-Methoden hat sich geändert, da von außen nur noch die Methode `run()` erreichbar sein muss.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/funktionen.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 require_once 'src/Controllers/IndexController.php';
 7 $requestController = new IndexController();
 8
 9 $action = isset($_GET['action']) ? $_GET['action'] : 'index'
10 $requestController->run($action);
```

index.php (Version 4)

Codebeispiel 96 *index.php (Version 4)*

In der *index.php* sind alle Zeilen nach der Festlegung des Wertes für `$action` weggefallen, da sich dieser Code nun in der neuen Methode befindet. Hinzugekommen ist dafür lediglich der Aufruf von `run()`, wobei wir den Wert von `$action` an diese Methode übergeben.

10.2.5 Schritt 5

Noch haben wir das in der Einleitung festgelegte Ziel nicht erreicht. Wir benutzen nämlich derzeit genau eine Klasse und keine **Klassen**. Es fehlt also eine Möglichkeit, zwischen verschiedenen Controller-Klassen wechseln. Hierfür führen wir einen neuen optionalen URL-Parameter `controller` ein.

Beispiel

```
 1 <?php
 2
 3 require_once 'inc/funktionen.inc.php';
 4 require_once 'src/Entities/Buch.php';
 5
 6 $controller = isset($_GET['controller']) ? $_GET['controller
 7 $action = isset($_GET['action']) ? $_GET['action'] : 'index'
 8
 9 $controllerName = ucfirst($controller) . 'Controller';
10 $controllerFile = 'src/Controllers/' . $controllerName . '.pl
11
12 if (file_exists($controllerFile)) {
13     require_once $controllerFile;
14
15     $requestController = new $controllerName();
16     $requestController->run($action);
17 }
```

index.php (Version 5)

Codebeispiel 97 *index.php (Version 5)*

Dieser neue URL-Parameter wird in Zeile 6 ausgelesen. Wurde er nicht angegeben, so benutzen wir den String `'index'` als Wert. Auf Basis dieser Information ermitteln wir dann in Zeile 9 den Klassennamen, wobei wir das erste Zeichen in einen Großbuchstaben umwandeln und das Suffix `Controller` anhängen. Mit dieser Information können wir dann in Zeile 10 auch den

Dateinamen der einzubindenden Controller-Klasse festlegen. Sofern die Datei tatsächlich existiert, binden wir sie im folgenden Code ein, instanziieren die Klasse und rufen abschließend, wie schon zuvor, die Methode `run()` auf.

Nun sind wir theoretisch schon in der Lage, mehrere Controller-Klassen zu verwenden. Leider scheitert dies in der Praxis daran, dass Methodennamen nicht mehr eindeutig sein müssen (was ja eigentlich ein ganz wesentlicher Vorteil bei der Verwendung von Klassen ist). Da die Methodennamen durch unseren Automatismus auch als Dateinamen der Templates Verwendung finden, stehen wir vor einem kleinen Problem. Wir müssen in einem solchen Fall entweder wieder das Template manuell bestimmen, oder wir passen einfach unseren Automatismus an.

Die Anpassung ist auch nicht weiter schwer, da wir die Templates einfach für jede Controller-Klasse separat in einem Unterordner ablegen können. Wir könnten hierfür der Methode `run()` einen weiteren Parameter verpassen, um so den Wert von `$controller` zu erfahren und diesen als Ordnernamen zu verwenden. Allerdings wäre dies etwas unschön, da wir diese Information ja schon durch die Instanziierung der korrekten Controller-Klasse in der *index.php* festgelegt haben. Wir haben diese Information also eigentlich schon. Anstatt den Wert also erneut an die Methode zu übergeben, fragen wir einfach das Objekt danach. Dies geht mit der Funktion `get_class()`.

Beispiel

```
 1 <?php
 2
 3 class IndexController
 4 {
 5     protected $context = array();
 6     protected $template = '';
 7
 8     // gekuerztes Beispiel
 9
10     protected function setTemplate($template, $controller = n
11     {
12         if (empty($controller)) {
13             $controller = get_class($this);
14         }
15
16         $this->template = $controller . '/' . $template . '.
```

```
17      }
18
19      protected function getTemplate()
20      {
21          return $this->template;
22      }
23
24      protected function addContext($key, $value)
25      {
26          $this->context[$key] = $value;
27      }
28
29      protected function render()
30      {
31          extract($this->context);
32
33          require_once 'templates/' . $this->getTemplate();
34      }
35  }
```

src/Controllers/IndexController.php (Version 5)

Codebeispiel 98 *src/Controllers/IndexController.php (Version 5)*

Es ändert sich nur die Methode `setTemplate()`. Hier gibt es nun einen zweiten optionalen Parameter, mit dem wir den Namen einer Controller-Klasse übergeben können. Dies ermöglicht uns zukünftig eine Wiederverwendung von Templates anderer Controller. Wird kein Name übergeben, so ermittelt die Methode intern den Klassennamen des aktuellen Objekts. In beiden Fällen wird dieser Name in Zeile 16 als Ordner vor dem Dateinamen des Templates angegeben. Sie müssen somit alle bisherigen Templates in den Ordner *templates/IndexController* verschieben.

Das Ziel von klassenbasierten Controllern haben wir nun erreicht. Ganz nebenbei haben wir bei der Umsetzung ein unter dem Namen ***Front-Controller*** bekanntes Entwurfsmuster verwendet.

> Ein Front-Controller dient als Einstiegspunkt in eine Webanwendung. Er erweitert üblicherweise das Model-View-Controller-Konzept. Alle Anfragen an die Webanwendung werden vom Front-Controller empfangen und an einen bestimmten Controller delegiert.

Bei uns ist die *index.php* dieser Front-Controller. Der Front-Controller

übernimmt jedoch nicht nur das Delegieren, sondern kann auch Aufgaben übernehmen, die für jede Controller-Klasse ausgeführt werden müssen. Zu diesen Aufgaben gehört z. B. das Starten der Session (darüber mehr in Band 2 dieses OOP-Lernbuchs), welche wir unter anderem für die Zwischenspeicherung von Informationen benötigen werden.

10.3 Vererbung

Eine Besonderheit von Klassen (wenn nicht sogar **die** Besonderheit) ist die sogenannte ***Vererbung***. Mittels Vererbung können Klassen aufeinander aufbauen. Die Klassen erben auf diesem Weg alle Methoden und Attribute der ***Elternklasse*** (auch Vater- oder Mutterklasse genannt). Genau diese Möglichkeit benötigen wir in unseren Controller-Klassen, da wir sonst in jeder Klasse Methoden, wie beispielsweise `run()` und `render()`, duplizieren müssten.

Beispiel

```php
<?php

abstract class AbstractBase
{
    protected $context = array();
    protected $template = '';

    public function run($action)
    {
        $this->addContext('action', $action);

        $methodName = $action . 'Action';
        $this->setTemplate($methodName);

        if (method_exists($this, $methodName)) {
            $this->$methodName();
        }

        $this->render();
    }

    protected function setTemplate($template, $controller = n
    {
        if (empty($controller)) {
```

```
25              $controller = get_class($this);
26          }
27
28          $this->template = $controller . '/' . $template . '.
29      }
30
31      protected function getTemplate()
32      {
33          return $this->template;
34      }
35
36      protected function addContext($key, $value)
37      {
38          $this->context[$key] = $value;
39      }
40
41      protected function render()
42      {
43          extract($this->context);
44
45          require_once 'templates/' . $this->getTemplate();
46      }
47  }
```

src/Controllers/AbstractBase.php (Version 1)

Codebeispiel 99 *src/Controllers/AbstractBase.php (Version 1)*

Um die besagte Code-Duplizierung zu vermeiden, habe ich eine neue Elternklasse `AbstractBase` erstellt. Diese enthält alle Attribute und Methoden, die wir ab Schritt 2 ergänzt hatten. In dieser neuen Klasse gibt es eigentlich nur zwei Besonderheiten und beide finden Sie in Zeile 3. Zunächst einmal fällt im Namen der Klasse auf, dass es ein Präfix `Abstract` gibt und unser übliches Suffix `Controller` fehlt. Außerdem findet sich ein weiteres `abstract` vor dem Schlüsselwort `class`. Beginnen wir am Anfang der Zeile. Das komplett kleingeschriebene `abstract` legt fest, dass eine Instanziierung dieser Klasse **nicht** möglich ist. Das fehlende Suffix versteckt die Elternklasse vor unserem Front-Controller. An dem Präfix erkennen wir zukünftig bereits am Klassennamen die fehlende Möglichkeit zur Instanziierung. Dies ist aber nur eine Namenskonvention (siehe auch Lektion 11).

Beispiel

```
1  <?php
2
```

```
3  class IndexController extends AbstractBase
4  {
5      protected function indexAction()
6      {
7          $this->addContext('buecher', holeBuecher());
8      }
9
10     protected function zeigeAction()
11     {
12         $this->addContext('buch', holeBuch($_GET['id']));
13     }
14
15     protected function neuAction()
16     {
17         if ($_POST) {
18             speichereBuch($_POST);
19             header('Location: index.php');
20             exit;
21         }
22     }
23 }
```

src/Controllers/IndexController.php (Version 6)

Codebeispiel 100 *src/Controllers/IndexController.php (Version 6)*

An der Klasse `IndexController` hat sich wenig geändert. Natürlich sind nur noch die drei ursprünglichen Methoden enthalten, der Rest befindet sich ja nun in der Elternklasse. Wesentlich interessanter ist jedoch Zeile 3. Hier finden Sie ein neues Schlüsselwort, nämlich `extends`. Dieses Schlüsselwort existiert bereits seit PHP 4 und ermöglicht in PHP eine Vererbung. Dahinter muss man lediglich noch die gewünschte Elternklasse angeben.

Beispiel

```
1  <?php
2
3  require_once 'inc/funktionen.inc.php';
4  require_once 'src/Entities/Buch.php';
5  require_once 'src/Controllers/AbstractBase.php';
6
7  $controller = isset($_GET['controller']) ? $_GET['controller
8  $action = isset($_GET['action']) ? $_GET['action'] : 'index'
9
10 $controllerName = ucfirst($controller) . 'Controller';
11 $controllerFile = 'src/Controllers/' . $controllerName . '.p
12
```

```php
if (file_exists($controllerFile)) {
    require_once $controllerFile;

    $requestController = new $controllerName();
    $requestController->run($action);
}
```

index.php (Version 6)

Codebeispiel 101 *index.php (Version 6)*

Abschließend fehlt nur noch eine Kleinigkeit, nämlich die Einbindung der Elternklasse in Zeile 5 unseres Front-Controllers.

10.4 Two-Step-Rendering

Eine Web-Anwendung hat im Normalfall ein einheitliches Layout und Design, d.h. auf allen Seiten wird beispielsweise ein identischer Kopf- und Fußbereich angezeigt. Um den Aufwand von Gestaltungsänderungen zu minimieren, könnte man wie in den Aufgaben der vorherigen Lektion das Menü in ein Teil-Template auslagern. Man könnte jedoch noch viel weiter gehen und direkt den kompletten Kopfbereich auslagern. Das Gleiche gilt natürlich auch für den Fußbereich. In einem solchen Fall gibt es jedoch häufig Probleme mit dem Syntax-Highlighting, da in den Teil-Templates öffnende oder schließende Tags fehlen können. Doch wie kann man dieses Problem umgehen?

Beispiel

```php
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Buchverwaltung</title>
</head>

<body>
    <ul id="navi">
        <li><a href="index.php">Startseite</a></li>
        <li><a href="index.php?action=neu">Buch anlegen</a></
    </ul>

    <?php require $template; ?>
```

```
16 </body>
17
18 </html>
```

templates/layout.tpl.php

Codebeispiel 102 *templates/layout.tpl.php*

Das Problem lässt sich lösen, indem man das Template-Rendering in zwei Schritten (engl. Steps) vornimmt. Für alle Aktionen wird in Schritt 1 zunächst ein gemeinsames Template *layout.tpl.php* eingebunden. In Schritt 2 bindet dieses dann wiederum das Template der aktuellen Aktion ein. Kopf- und Fußbereich befinden sich nun also in einer gemeinsamen Datei, lediglich der Inhaltsbereich ist ausgelagert. Es spricht jedoch nichts dagegen, Teile der *layout.tpl.php* in Partials auszulagern. Dies erleichtert beispielsweise bei einem langen Menü oftmals die Lesbarkeit.

Beispiel

```php
<?php

abstract class AbstractBase
{
    protected $context = array();
    protected $template = '';

    public function run($action)
    {
        $this->addContext('action', $action);

        $methodName = $action . 'Action';
        $this->setTemplate($methodName);

        if (method_exists($this, $methodName)) {
            $this->$methodName();
        }

        $this->render();
    }

    protected function setTemplate($template, $controller = n
    {
        if (empty($controller)) {
            $controller = get_class($this);
        }

        $this->template = $controller . '/' . $template . '.
```

```
29      }
30
31      protected function getTemplate()
32      {
33          return $this->template;
34      }
35
36      protected function addContext($key, $value)
37      {
38          $this->context[$key] = $value;
39      }
40
41      protected function render()
42      {
43          extract($this->context);
44
45          $template = $this->getTemplate();
46
47          require_once 'templates/layout.tpl.php';
48      }
49  }
```

src/Controllers/AbstractBase.php (Version 2)

Codebeispiel 103 *src/Controllers/AbstractBase.php (Version 2)*

Wir müssen zur Nutzung dieses gemeinsamen Templates lediglich die Methode `render()` überarbeiten. In Zeile 45 befüllen wir die lokale Variable `$template`, welche in Zeile 15 der *layout.tpl.php* benötigt wird. In Zeile 47 binden wir dann das neue Layout-Template ein und starten so die Ausführung von Schritt 1.

10.5 Zusammenfassung

Glückwunsch! Damit haben Sie die Grundlagen der objektorientierten Programmierung hinter sich gebracht. In Band 2 werden wir uns dann mit fortgeschritteneren PHP-Aspekten wie Composer, dem OR-Mapper Doctrine und den Themen Validierung und Sicherheit beschäftigen. Vor allem den letzten Aspekt haben wir nämlich in diesem Lernbuch sträflich vernachlässigt. Um den derzeitigen Stand unseres Codes produktiv zu verwenden, sollte man beispielsweise auf jeden Fall eine Maskierung aller Benutzereingaben ergänzen!

10.6 Aufgaben zur Selbstkontrolle

Übung 44:

Vollziehen Sie die Änderungen im Buch-Projekt nach und implementieren Sie den Endstand mit klassenbasiertem Controller, Front-Controller, Vererbung und Two-Step-Rendering. Testen Sie die Umsetzung im Browser. Was passiert, wenn Sie als URL-Parameter den Controller `index` angeben?

Übung 45:

Was passiert, wenn Sie als URL-Parameter einen nicht existierenden Controller wie beispielsweise `test` angeben? Überlegen Sie, wie eine Lösung für dieses Problem aussehen könnte und implementieren Sie diese testweise.

Hinweis: Meine Musterlösung geht von der Annahme aus, dass die Klasse `IndexController` immer existiert.

Übung 46:

Was passiert, wenn Sie als URL-Parameter eine nicht existierende Aktion wie beispielsweise `honk` angeben? Überlegen Sie, wie eine Lösung für dieses Problem aussehen könnte und implementieren Sie auch diese. Können Sie hierfür Teile der Controller-Lösung aus der vorherigen Aufgabe wiederverwerten?

10.7 Optionale Aufgaben

Übung 47:

Bauen Sie das Beispielprojekt *hallo* aus Lektion 9 um. Es soll nun ebenfalls

einen klassenbasierten Controller, einen Front-Controller *index.php* und eine Vererbung von der Elternklasse `AbstractBase` verwenden. Sie können gerne auch ein Two-Step-Rendering verwenden. Dieses ist jedoch bei einer einzigen Aktion (noch) nicht zwingend nötig.

11 Anhang: Programmierrichtlinien

In dieser Lektion lernen Sie
• wieso man Programmierrichtlinien einhalten sollte. • was die PHP FIG ist und wieso sie Richtlinien veröffentlicht. • was es mit PSR-1 und PSR-2 auf sich hat. • welche weiteren Richtlinien sinnvoll sind.

11.1Einleitung

In diesem Anhang finden Sie eine Zusammenstellung der in diesem Lernbuch verwendeten Programmierrichtlinien (auch Styleguides genannt). Diese Richtlinien erheben nicht den Anspruch, vollständig oder gar allgemeingültig zu sein. Sie sollen Ihnen aber helfen, besser lesbaren Code zu schreiben. Es ist übrigens nicht entscheidend, wie die Richtlinien genau lauten, sondern nur deren konsequente Einhaltung!

11.2 Richtlinien? Wieso? Weshalb? Warum?

Programmierrichtlinien regeln, wie unser Code in formaler und struktureller Hinsicht gestaltet sein soll (unabhängig vom eigentlichen Ziel des Codes). Dies bezieht sich insbesondere auf seine **Lesbarkeit** und **Wartbarkeit**. Solche Regeln sind natürlich nicht lebensnotwendig für die Aufgaben dieses Lernbuchs, aber unverzichtbar, wenn man in einem größeren Team arbeitet. Der Zweck solcher Richtlinien ist es, allen Teammitgliedern die Arbeit zu erleichtern.

Damit unser Code verständlich bleibt, kann eine Richtlinie die Verwendung von theoretisch erlaubten (aber praktisch unsauberen und schlecht lesbaren) Programmkonstrukten einschränken (sogenannter ***Code-Smell***, dt. übelriechender Code, siehe https://de.wikipedia.org/wiki/Smell_%28Programmierung%29). Auch gibt es

Richtlinien, die bei zwei gleichwertigen Varianten eine Variante vorgeben, damit der Code unabhängig vom jeweiligen Programmierer ein einheitliches Erscheinungsbild aufweist.

Früher oder später wird jeder Entwickler mit solchen Regeln konfrontiert. Dies kann beispielsweise der Fall sein, wenn Sie als neuer Entwickler in einem Team anfangen, als Freelancer bei einem Projekt mitarbeiten oder sich an einem Open-Source-Projekt beteiligen. Ich würde Ihnen deswegen raten, sich frühzeitig an die Einhaltung solcher Regeln zu gewöhnen. Wenn Sie möchten, können Sie beispielsweise überprüfen, ob Ihre bisherigen Lösungen die nachfolgenden Empfehlungen einhalten. Außerdem ist es sinnvoll, wenn Sie zukünftig bei Ihrem Code immer Programmierrichtlinien berücksichtigen. Dies müssen natürlich nicht zwingend die hier vermittelten Richtlinien sein, in Ihrem Team oder Projekt können im Einzelfall sogar genau gegensätzliche Regeln gelten.[3]

---

3 Das bekannteste Beispiel dürfte wohl die Entscheidung zwischen Tabulatoren und Spaces für Code-Einrückungen sein.

Der Punkt der Wartbarkeit wird übrigens umso wichtiger, je länger die Lebensdauer eines Softwareprodukts ist. In der Vergangenheit hat sich gezeigt, dass bei Softwareprodukten oftmals weit über 50% des Gesamtaufwandes auf die Wartung entfallen. Nur selten wird der Code jedoch vom ursprünglichen Programmierer oder Team gewartet. Umso wichtiger ist es, dass bereits vom ersten Augenblick an ein guter Programmierstil verwendet wird. Nur so erleichtern wir unseren Nachfolgern das Beheben von Fehlern und die Umsetzung neuer Anforderungen. Bedenken Sie hierbei immer, dass Sie auch einmal selbst ein solcher Nachfolger sein könnten.

11.3 Hintergrund

Andere Programmiersprachen haben oftmals ihre Richtlinien direkt dabei.[4] Bei PHP hat es jedoch nie einen konkreten Standard direkt vom Vendor-Team gegeben. Es wurde also nie offiziell festgelegt, wie man PHP-Code eigentlich schreibt – also wie man ihn so schreibt, dass er lesbar ist und bei jedem gleich aussieht. Im Laufe der Zeit gab es jedoch einige Versuche (z. B. *PEAR* siehe

http://pear.php.net/manual/en/standards.php und *Zend* siehe http://framework.zend.com/manual/1.12/en/coding-standard.html), diese Lücke zu füllen. Aber es fehlte immer die breite Akzeptanz.

---

4 In Python gibt es beispielsweise PEP 8. Siehe auch: http://legacy.python.org/dev/peps/pep-0008/

2009 trafen sich deshalb ein paar Entwickler, die verschiedene Frameworks der PHP-Welt repräsentierten. Ein Agendapunkt waren einheitliche Standards, die eine bessere Wiederverwendbarkeit von Framework-Komponenten ermöglichen sollten. So entstand die *PHP Standards Group* (siehe http://blog.echolibre.com/2009/06/recommended-php-standards-group/).

Diese Gruppe wollte Richtlinien entwickeln, die zukünftig für alle Mitgliedsprojekte gelten sollen. Gelten **sollen**, nicht müssen; ein Mitgliedsprojekt muss somit nicht zwingend jeden verabschiedeten Standard einhalten. Die Zielgruppe dieser Standards sind jedoch nicht wir, die Community der PHP-Entwickler als Gesamtes, sondern die Mitglieder der Gruppe selbst. Dies hindert uns aber natürlich nicht daran, die Standards auch bei unseren eigenen Projekten zu verwenden.

Seit der Gründung sind etliche neue stimmberechtigte Mitglieder[5] hinzugekommen, womit sich natürlich auch der Querschnitt der beteiligten Projekte und somit die Akzeptanz erweitert hat. Hinzugekommen sind u.a. Anwendungen wie Foren und Content Management Systeme (u.a. *Contao*, *Drupal* und *TYPO3* mit dem Flow Framework). Beteiligt sind aber auch Projekte wie *Composer* und *Doctrine*, die wir in Band 2 näher beleuchten werden.

---

5 Jeder (inklusive dir!) kann übrigens ein **nicht** stimmberechtigtes Community-Mitglied der Gruppe werden, indem er sich bei der Mailingliste anmeldet. Siehe auch: http://groups.google.com/group/php-fig/

Inzwischen hat sich die Gruppe in *PHP Framework Interoperability Group* (kurz FIG, siehe http://www.php-fig.org/) umgetauft und ein paar sehr gute Standards veröffentlicht, von denen ich die wichtigsten Regeln der Standards PSR-1 und PSR-2 im Folgenden kurz vorstellen möchte. PSR ist hierbei die Abkürzung für ***PHP Standards Recommendation*** (dt. Empfehlung).

11.4 Die Empfehlungen

Die Inhalte von Programmierrichtlinien können von Fall zu Fall unterschiedlich sein. Die Bandbreite reicht von einfachen Vorgaben zur Code-Strukturierung bis hin zur genauen Festlegung von Implementierungsdetails. In den meisten Richtlinien findet man jedoch Vorgaben zur Code-Einrückung (Strukturierung) und die Festlegung von Namenskonventionen (wie Bezeichner zu wählen sind).

Doch kommen wir nun zu den Regeln, die in diesem Lernbuch verwendet wurden. Wir beginnen mit PSR-1 und PSR-2 und kommen danach zu einigen Ergänzungen. Die Schlüsselwörter MÜSSEN/MÜSSEN NICHT/DÜRFEN NUR/DÜRFEN NICHT und SOLLTEN/SOLLTEN NICHT definieren innerhalb der beiden PSRs, wie verpflichtend eine Regel laut PHP FIG einzuhalten ist. Wie weitgehend Sie diese Empfehlungen selbst einhalten, ist Ihre Entscheidung. Beide PSRs werden jedoch bereits von vielen Projekten umgesetzt und scheinen sich zu einer Art Quasistandard zu entwickeln.

11.4.1 PSR-1: Framework-Interoperabilität

PSR-1 definiert ein paar Basisregeln bezüglich Namensgebung und Dateiinhalten (siehe auch https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-1-basic-coding-standard.md). Dies sind:

1. Dateien DÜRFEN NUR die PHP-Tags `<?php` und `<?=` verwenden.[6]
2. Dateien MÜSSEN als Zeichenkodierung UTF-8 ohne BOM für PHP-Code verwenden.
3. Dateien SOLLTEN entweder Symbole (wie Klassen, Funktionen, Konstanten, etc.) definieren oder eine Auswirkung haben (z. B. eine Ausgabe mit `echo` generieren). Sie SOLLTEN NICHT beides tun.
4. Namespaces und Klassen MÜSSEN einen Autoloading-Standard befolgen.[7]
5. Klassen-Namen MÜSSEN den UpperCamelCase verwenden.
6. Klassen-Konstanten MÜSSEN Großbuchstaben verwenden. Einzelne Wörter innerhalb dieser Bezeichner werden mittels Underscore getrennt.
7. Methoden-Namen MÜSSEN den lowerCamelCase verwenden.

11.4.2 PSR-2: Stilistische Programmierrichtlinien

PSR-2 ist eine Erweiterung von PSR-1 um stilistische Feinheiten (siehe auch https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-2-coding-style-guide.md).

Struktur

1. Programmiercode MUSS vier Leerzeichen als Einrückung verwenden (keine Tabulatoren bzw. Tabs).
2. Es DARF KEIN hartes (verpflichtendes) Zeichenlimit für einzelne Zeilen geben; das softe Limit MÜSSEN 120 Zeichen pro Zeile sein; Zeilen SOLLTEN 80 Zeichen oder weniger enthalten.[8]
3. Eine Zeile DARF NUR eine Anweisung enthalten, d.h. nach einem Semikolon erfolgt immer ein Zeilenumbruch.[9]
4. Nach der Deklaration des Namespaces MUSS eine Leerzeile folgen und es MUSS eine weitere Leerzeile nach dem Block mit den USE-Deklarationen geben.[10]
5. Die öffnenden und schließenden geschweiften Klammern einer Klasse MÜSSEN einzeln in einer eigenen Zeile stehen.
6. Die öffnenden und schließenden geschweiften Klammern einer Methode MÜSSEN einzeln in einer eigenen Zeile stehen.
7. Die Sichtbarkeit von Attributen und Methoden MUSS immer angegeben werden; `abstract` oder `final` MUSS vor der Sichtbarkeit angegeben werden; `static` MUSS nach der Sichtbarkeit angegeben werden.
8. Kontrollstrukturen MÜSSEN ein Leerzeichen zwischen dem jeweiligen Schlüsselwort und der öffnenden runden Klammer aufweisen; Funktions- und Methodenaufrufe DÜRFEN KEIN Leerzeichen vor der öffnenden runden Klammer haben.
9. Die öffnenden geschweiften Klammern von Kontrollstrukturen DÜRFEN KEINE eigene neue Zeile einnehmen, die schließenden geschweiften Klammern MÜSSEN hingegen einzeln in einer eigenen Zeile stehen.
10. Nach den öffnenden und vor den schließenden runden Klammern von

Kontrollstrukturen DARF KEIN Leerzeichen stehen.

Namenskonventionen

1. PHP-Schlüsselwörter (z. B. `if` und `for`) und PHP-Konstanten (`true`, `false` und `null`) MÜSSEN in Kleinbuchstaben geschrieben werden.
2. Die Bezeichner von Attributen und Methoden DÜRFEN NICHT mit einem Underscore beginnen.

Dateien

1. Alle PHP-Dateien MÜSSEN den Unix-Zeilenumbruch (LF) verwenden.
2. Das schließende PHP-Tag am Dateiende MUSS weggelassen werden, sofern die Datei nur PHP-Code enthält (sogenannte pure PHP-Dateien).
3. Alle (puren) PHP-Dateien MÜSSEN mit einer einzelnen Leerzeile enden.

Die letzten drei Regeln dürften die einen oder anderen Fragezeichen über Ihrem Kopf hervorgerufen haben. Ich möchte nun kurz auf die hiermit verbundenen Fragen eingehen.

Zeilenumbruch: Das Steuerzeichen zur Erzeugung eines Zeilenumbruchs unterscheidet sich je nach Betriebssystem (in Unix/Linux/Mac OS X ist es `\n` bzw. LF, unter Windows `\r\n` bzw. CR LF und unter Mac OS bis Version 9 war es `\r` bzw. CR. In den meisten modernen Windows-Editoren und IDEs kann man jedoch problemlos angeben, welcher Zeilenumbruch für neue Dateien verwendet werden soll, und hier hat sich als Standard LF eingebürgert.

Schließendes PHP-Tag: Das schließende Tag `?>` ist am Dateiende optional. Wird es jedoch in einer Datei angegeben und dahinter befinden sich noch ein paar (unsichtbare) Leerzeichen, so kann dies unter Umständen zu Problemen bei der Verwendung der `header`-Funktion führen. Wenn man das Tag in einer reinen PHP-Datei weglässt, vermeidet man also eine potenzielle Fehlerquelle. Templates sind jedoch keine solchen Dateien, da sie HTML-Code enthalten. Sie sind also von der Regel ausgenommen und ich würde das schließende Tag immer notieren.

Leerzeilen-Ende: Diese Empfehlung wirft meiner Meinung nach mehr Fragen auf, als sie Probleme löst. Dies zeigen auch die diversen Diskussionen (siehe https://groups.google.com/forum/#!topic/php-fig/KUP2sChmDJM und https://groups.google.com/forum/#!topic/php-fig/PSzmcFVQst0/discussion) zur Umsetzung dieser Empfehlung. Ich würde die Empfehlung so interpretieren, dass pure PHP-Dateien grundsätzlich mit einem LF enden sollen.[11] Dies wird in den meisten Editoren und IDEs dann als Leerzeile angezeigt. Die Leerzeile ist somit praktisch der Ersatz für das fehlende schließende PHP-Tag. Aus technischen Gründen sind LFs in den Beispielen dieses Lernbuchs jedoch leider **nicht darstellbar**.

---

11 Templates und PHP-Dateien, die mit HTML-Code enden, sind somit von der Regel ausgenommen.

11.4.3 Ergänzungen aus den Symfony-Standards

PSR-1 und PSR-2 regeln schon einiges, aber mir persönlich fehlten noch zu viele Empfehlungen für spezifische Details (z. B. Setzung von Leerzeichen). Aus diesem Grunde habe ich mich entschieden, zusätzlich einige Punkte aus den Symfony-Standards zu übernehmen, welche die genannten PSRs meiner Meinung nach optimal ergänzen (siehe auch http://symfony.com/doc/current/contributing/code/standards.html).

Struktur

1. Nach jedem Komma-Separator MUSS genau ein Leerzeichen folgen.[12]
2. Vor und hinter binären Operatoren (z. B. bei `==` und `&&`) MUSS genau ein Leerzeichen gesetzt werden.
3. Zwischen unären Operatoren (z. B. `!` und `--`) und dem Operanden DARF KEIN Leerzeichen gesetzt werden.
4. Hinter jedem Element in einem mehrzeiligen Array **(auch und vor allem hinter dem letzten Element)** MUSS ein Komma folgen.
5. Vor `return`-Anweisungen MUSS genau eine Leerzeile ergänzt werden, außer das Return steht als einzige Anweisung innerhalb geschweifter Klammern (z. B. bei einer `if`-Anweisung).

6. Unabhängig von der Anzahl der Anweisungen MÜSSEN immer geschweifte Klammern benutzt werden, um den Rumpf von Kontrollstrukturen zu kennzeichnen (z. B. bei `if`-Anweisungen oder Schleifen).[13]
7. Pro Datei DARF NUR genau eine Klassen-Definition notiert werden.[14]
8. Alle Klassen-Attribute MÜSSEN über den Methoden deklariert werden.
9. Methoden MÜSSEN bei der Deklaration nach ihrer Sichtbarkeit sortiert werden, zuerst kommt `public`, dann `protected` und zuletzt `private`.[15] Laut Symfony-Standards bildet der Konstruktor eine Ausnahme, der zur besseren Lesbarkeit unabhängig von seiner Sichtbarkeit immer direkt nach den Attributen kommen sollte. Ich persönlich würde dies auf alle magischen Methoden ausweiten und den Konstruktur als Erstes angeben.
10. Bei der Instanziierung einer Klasse MÜSSEN die runden Klammern nach dem Klassen-Namen immer angegeben werden.[16]

Namenskonventionen

1. Für die Bezeichner von Variablen, Funktionen und Parametern MUSS immer der lowerCamelCase verwendet werden.[17]
2. Für die Schlüssel von Options-Arrays und bei den Namen von Formularfeldern MÜSSEN Kleinbuchstaben verwendet werden. Einzelne Wörter werden durch Underscores getrennt.
3. Für alle Klassen MÜSSEN Namespaces verwendet werden.[18]
4. Bei abstrakten Klassen MUSS das Präfix `Abstract` im Bezeichner verwendet werden.
5. In Dateinamen DÜRFEN NUR alphanumerische Zeichen (lateinische Buchstaben und arabische Ziffern)[19] und der Underscore verwendet werden.[20]

11.4.4 Eigene Ergänzungen

Aus meiner Erfahrung als PHP-Tutor resultieren noch ein paar weitere Vorschläge, deren Einhaltung uns allen das Leben leichter macht.

Struktur

1. Für Kontrollstrukturen SOLLTE die C-Syntax mit geschweiften Klammern verwendet werden. Lediglich in Templates SOLLTE zur besseren Lesbarkeit die alternative Syntax (siehe http://php.net/de/control-structures.alternative-syntax) verwendet werden, wobei der Doppelpunkt direkt nach der schließenden runden Klammer kommt.
2. Die öffnenden und schließenden geschweiften Klammern von Funktionen SOLLTEN genauso wie bei den Methoden einzeln in einer eigenen Zeile stehen.
3. Vor und hinter dem Pfeiloperator `->` DÜRFEN KEINE Leerzeichen stehen.

Namenskonventionen

1. Klassen-Namen SOLLTEN im Singular formuliert werden.
2. Tabellennamen in Datenbanken SOLLTEN im Plural formuliert werden.[21]
3. In den Namen von Datenbanken, Tabellen und Tabellenspalten DÜRFEN NUR Kleinbuchstaben verwendet werden. Einzelne Wörter werden durch Underscores getrennt.

Außerdem gilt natürlich weiterhin alles, was Sie in Lektion 2 zum Thema Strukturierung von PHP-Code gelernt haben.

12 Anhang: Weiterführende Informationen

In dieser Lektion lernen Sie
• wo Sie interessante, weiterführende Quellen zum Thema PHP finden.

12.1Einführung

Nach dem langen Weg durch dieses Lernbuch muss ich Ihnen leider sagen, dass das noch lange nicht alles war, was es zum Thema PHP oder OOP zu wissen gibt. Sonst wäre dieses Buch mehrere tausend Seiten dick und das Wort »alles« wäre immer noch gelogen. Deshalb habe ich Ihnen in der letzten Lektion einige interessante Webseiten zusammengestellt, über die Sie weiterführende Informationen rund um PHP erhalten. Außerdem möchte ich Ihnen auch Band 2 dieses Lernbuchs ans Herz legen, in welchem weitere OOP-Aspekte am Beispiel von *Doctrine 2* behandelt werden.

12.2 Weblinks

12.2.1 www.php.net

http://php.net/manual/de/

Die offizielle Seite der PHP-Entwickler bietet nicht nur eine Übersicht über alle PHP-Funktionen, sondern auch ein Handbuch in deutscher Sprache. Dieses Handbuch versteht sich selbst eher als Referenz denn als Lernheft. Folglich sind auch die Erklärungen und Beispiele effizient, aber eher knapp gehalten. Die Seite eignet sich sehr gut, um in Ihrem PHP-Wissen gezielt Lücken zu füllen. Als entspannte Abend-Lektüre würde ich sie weniger empfehlen.

12.2.2 www.phpdeveloper.org

http://www.phpdeveloper.org/

<?PHPDeveloper.org ist eine englischsprachige Kommunikationsplattform rund um PHP. Sie finden dort eine Übersicht von Neuigkeiten aus der PHP-Welt, Informationen über Konferenzen, Jobangebote, aktuelle Diskussionen und eine Übersicht der interessantesten Fachartikel der letzten Zeit.

12.2.3 devzone.zend.com

http://devzone.zend.com/

In der englischsprachigen Entwickler-Community von ***ZEND***, den Entwicklern von PHP, finden Sie interessante und oft sehr aktuelle, aber doch recht fortgeschrittene Artikel (engl. Tutorials) zum Thema PHP. Die Seite wird für Sie umso wichtiger werden, je mehr Sie über PHP wissen.

Lösungen der Wissensfragen

Lektion 3: Einführung in die objektorientierte Programmierung

1. **Was ist in der Objektorientierung ein Objekt?**

   Ein Objekt in der Programmierung (auch Instanz genannt) ist vergleichbar mit den greifbaren Objekten bzw. Gegenständen der realen Welt. Jedes Objekt gehört zu einer bestimmten Objektgruppe und verfügt über eine bestimmte Anzahl von Eigenschaften, die bei allen Objekten einer Gruppe vorhanden sind. Im Gegensatz zur realen Welt liegt jedoch keine Betonung auf greifbar, d.h. auch die Gruppen Flüssigkeit, Gas und Lebewesen sind nicht ausgeschlossen.

2. **Was ist eine Klasse?**

   Bei dem Begriff Klasse handelt es sich um den aus dem englischen abgeleiteten Fachbegriff für eine Objektgruppe. Sie stellt eine Art Bauplan bei der Instanziierung von Objekten dar.

3. **Was sind Attribute?**

   Attribute sind die Eigenschaften, die ein Objekt hat bzw. haben kann. Sie werden im Normalfall in der Klasse definiert und mit einem Standardwert befüllt.

4. **Wie können Sie in PHP aus einer Klasse ein Objekt instanziieren?**

   Dies geht beispielsweise, indem man das Schlüsselwort `new` und den Namen der Klasse verwendet und das Ergebnis einer Variablen zuweist: `$sparbuch = new Konto();`

5. **Auf was bezieht sich die Variable `$this`?**

   Auf das aktuelle Objekt, in dem man sich gerade befindet. Wann immer man in einer Klasse auf ein Attribut oder eine Methode einer Instanz dieser Klasse zugreifen will, muss man die spezielle Variable `$this` verwenden.

6. **Was unterscheidet Methoden von Funktionen?**

   Methoden werden in Klassen definiert und sind an deren Instanzen gebunden. Methoden können über das Schlüsselwort `$this` auf die Attribute und andere Methoden einer Instanz zugreifen. Ansonsten verhalten sie sich wie Funktionen.

7. **Wie können Sie eine Methode eines Objektes aufrufen?**

   Dies geht mit der Schreibweise `$objekt->nameDerMethode()`, wobei `$objekt` eine Variable sein muss, in der eine Instanz der gewünschten Klasse abgelegt ist.

Lektion 4: Getter- und Setter-Methoden

1. **Was unterscheidet Getter- und Setter-Methoden von normalen Methoden?**

   Technisch gesehen sind es ganz normale Methoden, also nichts.

2. **Warum ist Ihr Code besser wartbar, wenn Sie Attribute über Methoden verändern statt direkt?**

   Weil man auf diese Weise in das Ändern des Attributs eingreifen kann, um

den Wert zu prüfen oder zu ändern, ohne dass man davon außen etwas bemerkt.

3. **Wie können Sie auf ein Attribut nur lesenden Zugriff erlauben?**

   Man markiert das Attribut als `protected` und schreibt nur einen Getter.

Lektion 5: Arbeiten mit Objekten

1. **Wie können Sie eine Methode desselben Objektes von einer anderen Methode aus aufrufen?**

   ```
   $this->methodenName()
   ```

2. **Können Sie Objekte in der Session speichern?**

   Ja!

3. **Was testet der Operator `instanceof`?**

   Er testet, ob eine Variable ein Objekt enthält, das aus einer bestimmten Klasse erzeugt wurde. Der Code `$test instanceof TestKlasse` überprüft beispielsweise, ob in `$test` eine Instanz von `TestKlasse` abgelegt ist.

Lektion 6: Virtuelle Attribute

1. **Was unterscheidet virtuelle Attribute von echten Attributen?**

   Virtuelle Attribute sind nicht wirklich in der Klasse definiert. Sie verfügen aber über Getter und Setter, die deren Existenz vorgaukeln.

2. **Unter welchen Umständen sind virtuelle Attribute nützlich?**

   Wenn man mehrere Attribute zur Verfügung stellen will, die sich aber aus einer einzigen Datenquelle ableiten lassen.

3. **Was ist bei Settern von virtuellen Attributen zu beachten?**

   Diese Setter verändern nicht die virtuellen, sondern echte Attribute, die selbst ebenfalls Setter haben. Man muss dafür sorgen, dass sich diese Setter nicht in die Quere kommen.

Lektion 7: Magische Methoden

1. **Was versteht man unter einer *magic method*?**

   Magische Methoden sind Methoden, die von PHP automatisch aufgerufen werden, wenn ein vordefiniertes Ereignis eintritt. Sie haben genau festgelegte Namen.

2. **Wann wird die Methode `__toString()` eines Objektes automatisch aufgerufen?**

   Sobald man ein Objekt als String behandelt, also z. B. versucht, es mit `echo` auszugeben.

3. **Wann wird die Methode `__construct()` eines Objektes automatisch aufgerufen?**

   Jedes Mal, wenn ein neues Objekt erzeugt wird.

4. **Benötigt jede Klasse eine Konstruktor-Methode?**

   Es ist keine zwingende Voraussetzung. Wenn es nichts für einen Konstruktor zu tun gibt, kann man ihn selbstverständlich weglassen.

5. **Sie haben ein `Person`-Objekt in der Variable `$person` und möchten dessen Inhalt als Text ausgeben. Die in dieser Lektion vorgestellten magischen Methoden sind vorhanden. Was können Sie aufrufen, wenn Sie keine Getter verwenden dürfen?**

   `echo $person` oder `echo $person->__toString()`

Lektion 8: Beziehungen zwischen Objekten

1. **Wie können Sie ein Objekt in einem anderen Objekt ablegen?**

   Indem das innere Objekt als Attribut des anderen Objekts abgelegt wird.

2. **Wie können Sie auf die Attribute des innen abgelegten Objekts zugreifen?**

   Dies geht bespielsweise über virtuelle Attribute, deren Getter/Setter auf die Attribute des inneren Objekts zugreifen.

3. **Warum sollten Sie bevorzugt ganze Objekte als Parameter an Methoden übergeben?**

   Es macht den Code robuster gegen Änderungen, da man weniger Informationen über die eigenen Klassen nach außen dringen lässt.

Lektion 9: MVC

1. **Auf welche Sprachelemente sollte sich der PHP-Code in Templates beschränken?**

   Nach Möglichkeit auf `echo`, Schleifen und `if-else`-Anweisungen.

2. **Warum sollten Sie so wenig PHP wie möglich in Templates verwenden?**

   Um so weit wie möglich HTML- von PHP-Code zu trennen. Auf diese Weise ist der PHP-Teil übersichtlicher und der HTML-Teil kann leichter von Web-Designern ohne PHP-Kenntnisse angepasst werden.

3. **Welche Aufgabe hat ein Controller?**

   Ein Controller entscheidet auf Basis des aktuellen Requests, welche Aktion ausgeführt (z. B. welche Modelle hierfür benötigt oder verändert werden müssen) und welches Template zur Anzeige verwendet werden soll.

4. **Wie kann der Controller entscheiden, welche Aktion er ausführen soll?**

   Über einen Parameter, der beispielsweise beim Aufruf einer Seite in der URL übergeben und mit PHP ausgelesen bzw. ausgewertet wird (bei uns mit `$_GET['action']`).

5. **Warum sollte es eine Standard-Aktion geben?**

   Falls kein Parameter übergeben wurde, sollte trotzdem eine Aktion ausgeführt werden.

Über den Autor

Jan Teriete ist seit 1997 in der IT tätig. Vom EDV-Support erfolgte relativ schnell der erste Schritt in die Weiten des World Wide Web. In den Jahren 2004/2005 absolvierte er eine Vollzeit-Ausbildung zum Webmaster, womit auch seine Liebe zu PHP begann.

Seit 2005 ist Jan Teriete als Freelancer unterwegs und verdient seinen Lebensunterhalt als Web-Entwickler, vorwiegend mit CMS-Projekten auf Basis von Drupal und WordPress. Nebenbei ist er auch bei der Webmasters Akademie in Nürnberg als freiberuflicher Tutor, Dozent und Autor für den Fachbereich PHP/OOP und Doctrine 2 tätig.

Außerdem ist er bei Drupal- und WordPress-Meetups im Raum Nürnberg anzutreffen, wo er regelmäßig Vorträge zu CMS-Themen hält.

Jetzt Buch registrieren und Begleitmaterial herunterladen!

Registrieren Sie Ihr Buch auf webmasters-press.de und laden Sie sich die Übungsdateien und Lösungen herunter:

www.webmasters-press.de/register
Registrierungscode: 1289ae27409a

Table of Contents