Inhaltsverzeichnis

1. Objektorientierte Programmierung mit PHP

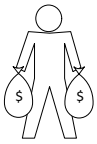
1.1 Einführung in OOP

Bei der Entwicklung von PHP-Anwendungen sollten Sie bereits mit der Entwicklung von Datenmodellen z. B. mittels dem Entity-Relationship-Modell Erfahrungen gesammelt haben.

Bei der Erstellung von Datenmodellen geht es ausschließlich um die im System verwendeten Daten. In der objektorientierten Programmierung werden nicht nur die Daten betrachtet, sondern auch die dazugehörigen Funktionen und Prozesse.

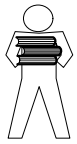
Objekte sind dabei Abbildungen der realen Welt, für die ein System programmiert werden sollen.

Fritz Feuerstein
Geboren 29.08.1960
Personalleiter
Braune Haare



Gustav Geier
Geboren 03.05.1970
Buchhalter
Hobby: Kitesurfen

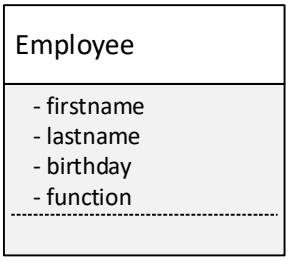
Sabine Schmidt
12.01.1980
Vertriebsleiterin
Lackierte Fingernägel



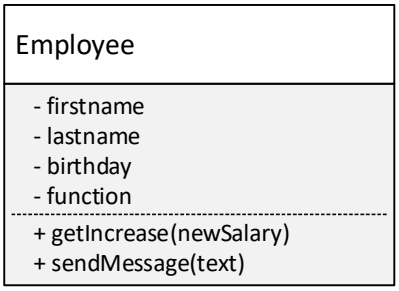
Willi Wunderlich
Geboren 30.10.1990
Bürobote
Fährt einen grünen Golf

Die oben angegebenen Mitarbeiter und Mitarbeiterinnen einer Firma haben alle unterschiedliche Attribute (Name, Geburtstag, Funktion, Einstellungsdatum, Gehalt usw.). Innerhalb einer Software für eine Mitarbeiterverwaltung müssen die relevanten Daten herausgefunden und die nicht relevanten Daten (Haarfarbe, Hobbys usw.) nicht beachtet werden.

Dafür erstellen wir einen Bauplan für Mitarbeiter und Mitarbeiterinnen, der die Klasse **Employee** beschreibt.



Attribute, bzw. Eigenschaften sind **firstname**, **lastname**, **birthday** und **function**. Es fehlen noch die notwendigen Aktivitäten. Der Mitarbeiter Gustav Geier könnte eine Gehaltserhöhung erhalten **getIncrease(newSalary)** oder Willi Wunderlich könnte eine Nachricht an Sabine Schmidt schicken **sendMessage(text)**. Dies sind Aktivitäten, die in Methoden einer Klasse abgebildet werden.



Wir haben jetzt eine Klasse mit Eigenschaften und Methoden, die einen "Bauplan" für einen Mitarbeiter darstellen.

In der objektorientierten Programmierung unterscheidet wir zwischen Klassen und Objekten. Eine Klasse repräsentiert wie schon erwähnt einen "Bauplan" für ein Objekt. So hat das Objekt `feuerstein` andere Werte als das Objekt `schmidt`. Beide haben jedoch den selben "Bauplan" Klasse `Employee`.

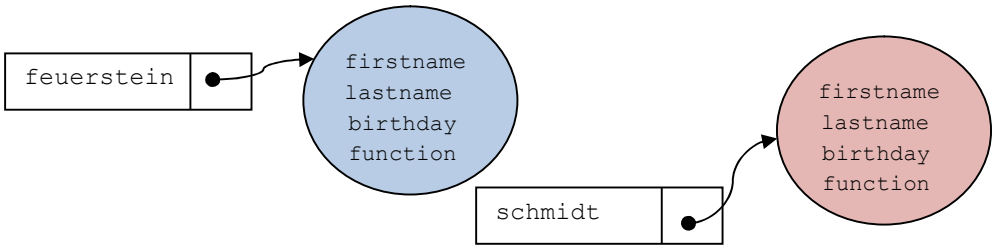


Abb. 1: Klassen und Objekte im Arbeitsspeicher

Im Allgemeinen ist das Objekt eine Variable, die im Arbeitsspeicher vorhanden ist, während eine Klasse eine Datei ist, die auf der Festplatte unter einem eigenen Namen abgespeichert wird. Diese Variable verweist dabei auf einen Speicherplatz im Arbeitsspeicher, in welchem die eigentlichen Werte (firstname, lastname, usw.) liegen.

Alle Klassen, die für eine Anwendung benötigt werden, können in einem **UML-Klassendiagramm** abgebildet werden. (siehe Kap. 1.3)

1.2 Aufgabenstellung

Am Beispiel einer Webanwendung, bei der es möglich ist online eine Art "Schwarzes Brett" innerhalb einer Firma zur Verfügung zu stellen, betrachten wir die objektorientierte Form.

A. Zielsetzung und Ausgangssituation

A.1 Ausgangssituation

In einer Firma soll es den MitarbeiterInnen ermöglicht werden private Anzeigen im Intranet zu schalten, bzw. zu lesen. Das Angebot soll ohne Registrierung oder Anmeldung zur Verfügung stehen. Die Interessenten treten mit den Inserenten über E-Mail in Kontakt.

Wenn ein Angebot eingestellt wird, muss ein Nickname und eine E-Mailadresse eingetragen werden. Ein Nickname und die dazugehörige E-Mailadresse können beliebig oft für das Einstellen von Anzeigen verwendet werden. Allerdings darf weder der Nickname noch die E-Mailadresse mit anderen E-Mailadressen, bzw. Nicknames kombiniert werden.

A.2 Zielsetzung

i. Musskriterien

- Aufgeben von Anzeigen
- Ausgabe aller Anzeigen der jeweiligen Rubrik

ii. Wunschkriterien

- Die Verwaltung der Anzeigen und der Inserenten soll von einem Online-Redakteur erledigt werden.
- Der Online Redakteur muss sich anmelden, um die neuen Funktionen benutzen zu können.
- Nach der Anmeldung sieht der Online Redakteur das Menü des Backends
- Anzeigen freigeben und löschen
- Inserenten anlegen, ändern und löschen
- Anzeigenrubriken anlegen, ändern und löschen.
- Nachdem der Online-Redakteur eine Anzeige freigegeben hat, soll automatisch eine E-Mail an den Inserenten gesendet werden.
- Täglich sollen Anzeigen, die älter als 14 Tage alt sind, automatisch gelöscht werden.
- Das Frontend für alle Mitarbeiter bleibt unverändert

iii. Abgrenzungskriterien

- Responsive Webdesign

B. Produktübersicht

Die Webanwendung soll die drei beschriebenen Grundfunktionen "Anzeige aufgeben", "alle Anzeigen einer Rubrik anzeigen" und "alle Rubriken anzeigen" beinhalten. Dabei soll aus dem Anzeigen aller Rubriken die Möglichkeit geschaffen werden alle Anzeigen absteigend nach Datum anzuzeigen.

Wenn eine Rubrik ausgewählt wurde, wird der Anwendungsfall "Anzeigen einer Rubrik lesen" ausgeführt. Sonst nicht.

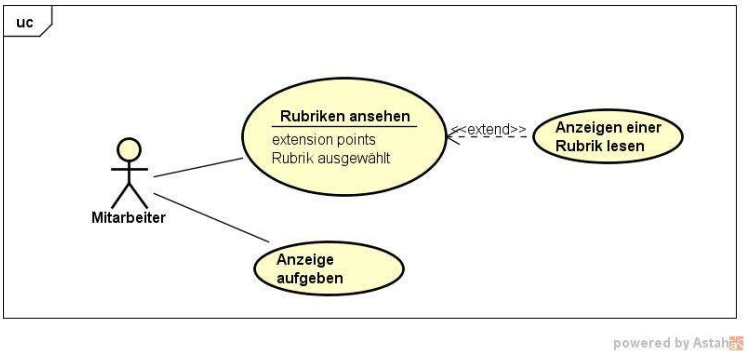


Abb. 2: UseCase Webbrett

1.3 UML-Klassendiagramm

Ein **Objekt** ist dabei eine konkrete Anzeige mit einer Anzeigenummer und einem Anzeigentext, die es beschreiben. In diesem Beispiel ist ein Objekt die Anzeige mit der Anzeigennummer 503 und dem Anzeigentext "Kinderfahrrad, blau zu verkaufen". Ein Objekt wird im Allgemeinen im Arbeitsspeicher abgelegt.

Eine **Klasse** ist eine allgemeine Beschreibung aller gleichartigen Objekte. Für diese Objekte wird eine Art Bauplan festgelegt, für alle Attribute und Funktionalitäten gleich sind. Wir sprechen von **Eigenschaften** eines Objekts, bzw. einer Klasse, die das konkrete Objekt beschreiben**.**

Nur Variable, die ein Objekt näher beschreiben, sollten Eigenschaften der Klasse werden!

Jede Klasse sollte auch über Funktionen verfügen, um etwas zu speichern, eine Aktion auszulösen oder eine Nachricht an das konkrete Objekt verschicken zu können. Zum Beispiel, wenn ein Inserent eine Anzeige aufgeben möchte: `anzeigeEintragen()`. Die Funktionen werden **Methoden** genannt.

Als Klassen identifizieren wir zuerst einmal die Anzeigen, die Inserenten und die Rubriken.

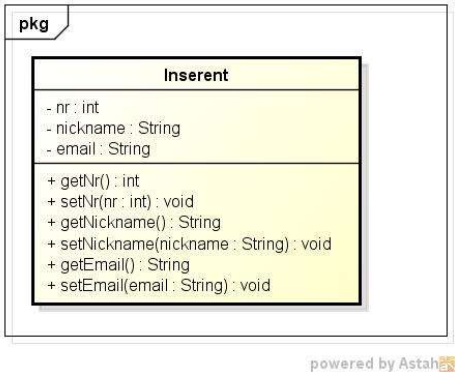
Die Werte der Eigenschaften können verschieden sein und sollten möglichst valide sein. Das bedeutet, es dürfen keine falschen Werte (negative Anzeigennummern, ungültige E-Mailadressen) eingetragen werden. Aus diesem Grund werden die Eigenschaften geschützt. Werte dürfen nur über Funktionen, bzw. Methoden gesetzt werden. Das nennen wir **Kapselung**. So kann eine E-Mailadressen nur über die Methode `setEmail`() verändert und über `getEmail()` gelesen werden.

Wir erhalten das UML-Klassendiagramm Abbildung 2 im Pflichtenheft.

Dabei werden die Sichtbarkeit, bzw. Zugriffsmöglichkeiten auf Eigenschaften und Methoden durch +, - oder #-Operator dargestellt.

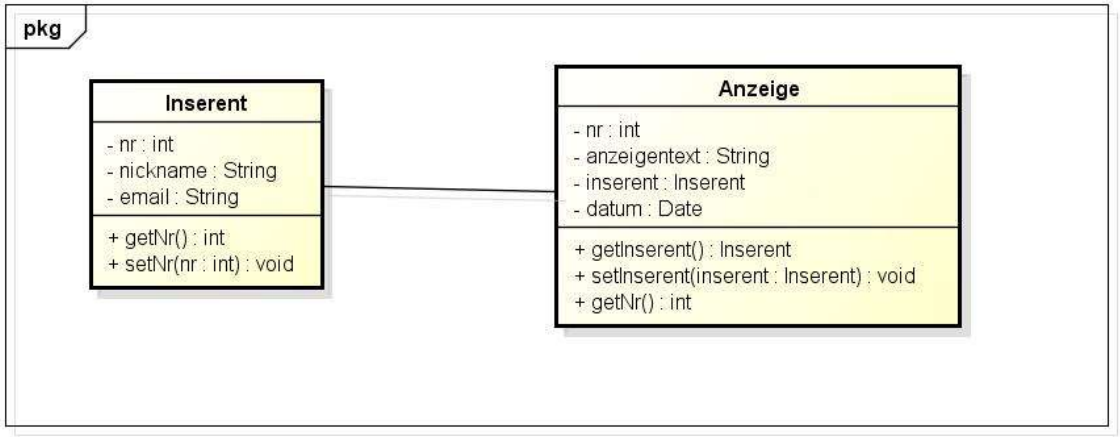
+	public	Jede andere Klasse hat uneingeschränkten Zugriff
#	protected	Nur innerhalb der Klasse oder in abgeleiteten Klassen sichtbar und Zugriff möglich.
-	private	Nur innerhalb der Klasse sichtbar und Zugriff möglich.

Für Eigenschaften und Methoden wird im UML-Klassendiagramm immer **name : typ** angegeben, auch wenn PHP die Datentypen dynamisch bei der Verwendung festlegt. Das Schlüsselwort **void** sagt aus, dass kein Wert zurückgegeben wird. Dies darf nicht mit **null** verwechselt werden. Mit **null** wird ausgesagt, dass ein Objekt, bzw. eine Variable keinen Wert besitzt.

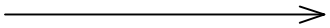


Assoziationen

Innerhalb eines Klassenmodells wird es immer mehrere Klassen geben, die miteinander in Interaktion stehen. Die Beziehungen werden als **Assoziationen** bezeichnet.

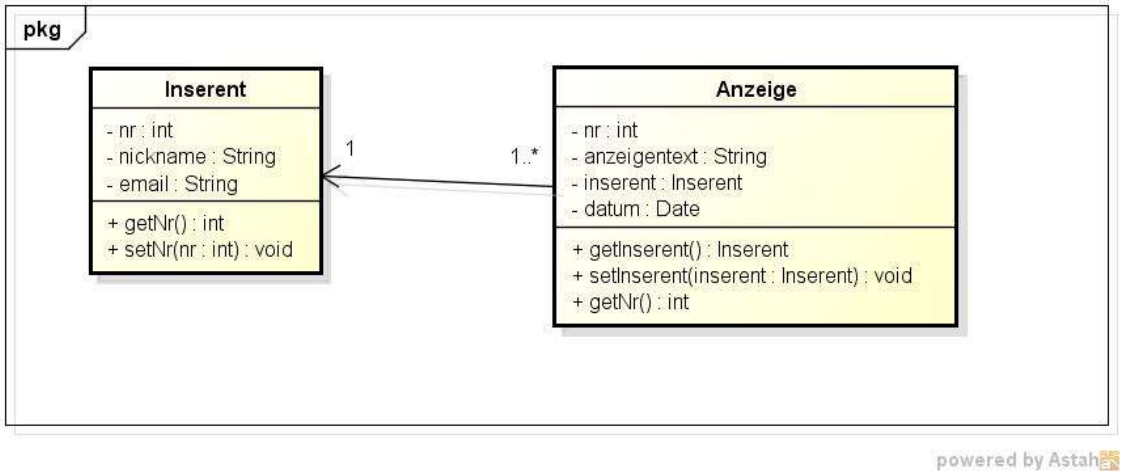


Diese Art der Verbindung zwischen Inserent und Anzeige sagt noch wenig aus. Eine Assoziation beschreibt ähnlich wie im Entity-Relationship-Diagramm die Beziehungen zwischen den Klassen. Eine gerichtete Assoziation sagt aus, dass die Klasse eine andere Klasse kennt, bzw. auf sie zugreifen kann.



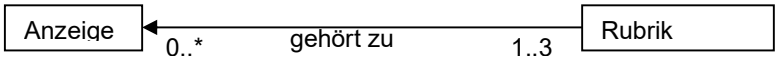
Gerichtete Assoziation

Verändertes Beispiel:



Wir sehen, dass die Anzeige ihren Inserenten kennt, weil der Pfeil von Anzeige auf Inserent zeigt. Dafür besitzt die Klasse Anzeige eine neue Eigenschaft inserent, die ein Objekt vom Typ Inserent enthält. Damit kann innerhalb der Klasse Anzeige auf die Methoden des Objekts inserent zugegriffen werden.

Die Klasse Rubrik kennt die Klasse der Anzeigen.



Die Zahlen an den Klassen drücken die **Multiplizität** aus, das ist die untere und obere Grenze der möglichen Anzahl. Eine Anzeige kann zu 1 oder bis zu 3 Rubriken gehören. Eine Rubrik hat 0 oder viele (*) Anzeigen. Dabei treten zwangläufig doppelte Anzeigen auf. Wenn eine Anzeige "Verkaufe Buch 'PHP ist auch eine Insel' " zu zwei verschiedenen Rubriken ("Computer" und "Bücher") gehört, dann enthält jedes Rubrik-Objekt dasselbe Anzeigen-Objekt.

Denkbar sind im Klassenmodell auch bidirektionale Beziehungen zwischen Klassen. So könnten die Anzeige die Rubriken "kennen" und die Rubriken die Anzeigen.

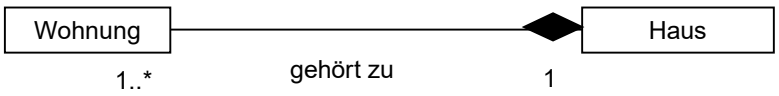
Aggregation

Genauso wie eine Anzeige ein Objekt Inserent enthält und nicht wie im relationalen DB-Modell eine Inserentennummer, so sollte die Liste aus Anzeigen, die der betreffenden Rubrik zugeordnet wurden, kein Array aus lauter Anzeigennummern sein, sondern ein **Array aus Objekten des Typs Anzeige**. Wir erhalten eine Anzeigenliste, über die deutlich wird, welche Anzeigen zur Rubrik gehören.

Dafür gibt es eine genauere Darstellungsart, um anzugeben, dass eine Rubrik besitzt/ enthält/ besteht aus mehrere/n Anzeigen. Man spricht dann von **Aggregation**. Die Raute muss dabei in der Nähe der Klasse stehen, die die anderen Klassen enthalten.

Komposition

Eine besondere Art wäre dann die **Komposition**, bei der der Aufbau der Aggregation ähnelt, aber die Teile können ohne das Ganze nicht existieren. Beispiele: Ein Haus enthält Wohnungen, eine Rechnung enthält Rechnungspositionen usw.



Um oben beschriebene Anwendungsfälle umsetzen zu können, wird ein Klassenmodel der benötigten Klassen erstellt. Hierbei sind nur die Klassen `Rubrik`, `Anzeige` und `Inserent` abgebildet, die die Geschäftslogik darstellen. Es fehlen die Anwendungsfälle.

Die Klasse `Anzeige` soll eine Klasse `Inserent` als Eigenschaft enthalten, damit wird angegeben, dass die `Anzeige` den `Inserenten` kennt. Sobald es eine `Anzeige` gibt, muss es auch einen `Inserenten` geben, da sonst die Anzeige sinnlos wäre. Ein `Inserent` kann viele `Anzeigen` aufgeben.

Eine `Rubrik` enthält keine oder mehrere `Anzeigen`. Eine `Anzeige` gehört mindestens zu einer `Rubrik`, aber maximal zu drei Rubriken. Dabei navigieren wir von der `Rubrik` zur `Anzeige`, die `Rubrik` kennt ihre `anzeigenList`.

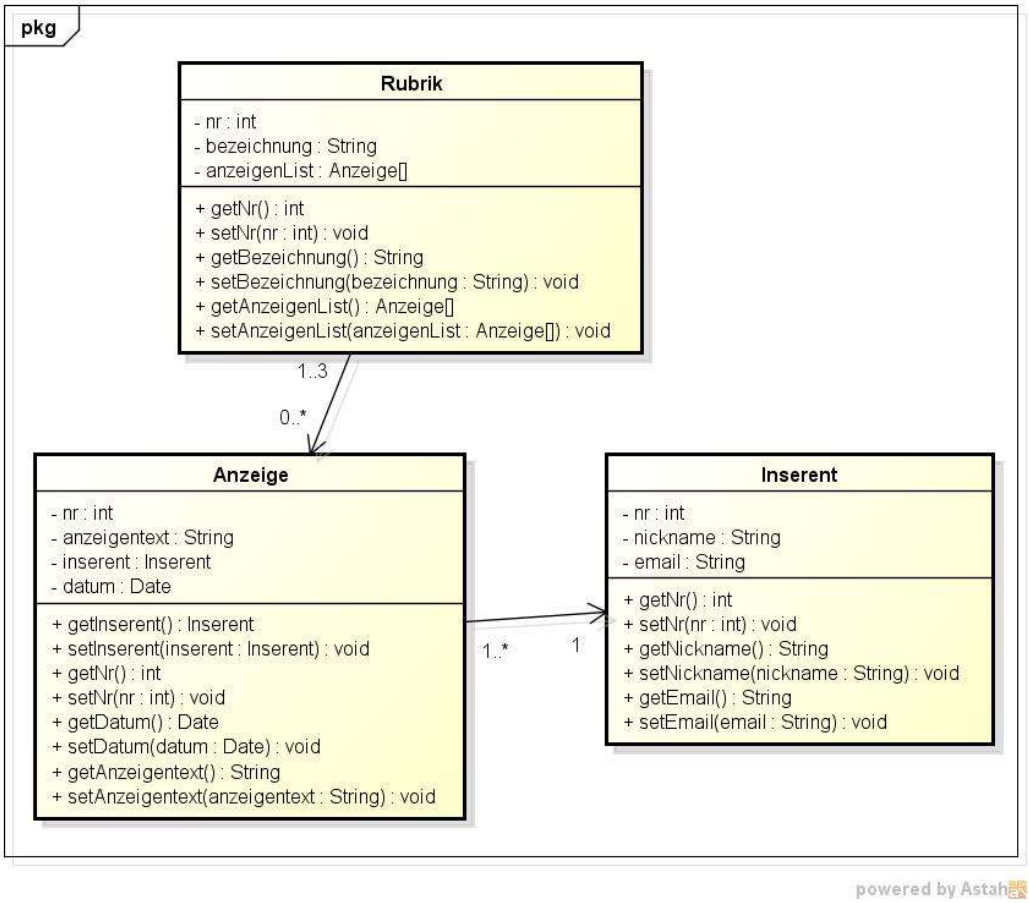


Abb. 3: Das vollständige Klassendiagramm Webbrett

1.4 Klassen und Objekte in PHP

Eine Klasse wird in PHP immer mit dem Schlüsselwort `class` begonnen und sollte in einer eigenen Datei abgespeichert werden. Außerdem sollte der Klassenname immer mit einem Großbuchstaben beginnen.

Anzeige.php enthält:

```
class Anzeige{
   . . .
}
```

Innerhalb der Klasse werden zuerst die Eigenschaften mit Zugriffsidentifikatoren (hier `private`) und dann die Methoden mit dem Schlüsselwort **function** angegeben.

```
class Anzeige{
   private $nummer;
   private $text
   . . .

   public function setNumer( $nummer ){
. . .
   }

. . .

   public function getDatum(){
      . . .
   }

}
```

Von einer Klasse kann man eine beliebige Anzahl von Objekten erzeugen. Es wird dann auch vom Instanziieren oder vom Erzeugen einen Instanz gesprochen. In PHP heißt das, dass außerhalb der Klasse ein konkretes Objekt mit dem Schlüsselwort **new** erzeugt wird:

```
$anzeige = new Anzeige(); //   Objekt <-> Klasse
```

Oder auch:

```
$anzeige1 = new Anzeige();
$anzeige2 = new Anzeige();
$anzeige3 = new Anzeige();
```

Erst danach kann auf die Methoden der Klasse zugegriffen werden, das bedeutet der Code der Methode ausgeführt werden.

```
$anzeige2->setNummer(78);
```

Ausgaben oder Berechnungen können dann ebenfalls über die Methode eines Objekts realisiert werden:

```
echo "Anzeige mit dem Datum " . $anzeige3->getDatum();
```

Wie eben dargestellt kann auf public-Methoden über die Syntax `$objektname->…` zugegriffen werden. Wird innerhalb der Klasse auf eine Eigenschaft oder eine Methode zugegriffen, wird das Schlüsselwort **this** verwendet.

```
  public function setNummer( $nummer ){
    $this->nummer = $nummer;
  }
```

Auch in PHP gibt es **Konstruktoren**, die bei der Erzeugung (Instanziieren) eines Objekts aufgerufen werden. Innerhalb dieser Konstruktoren können Eigenschaften des Objekts initialisiert werden, um einen definierten Zustand für ein Objekt zu erhalten oder dieses aus einer Datenbankabfrage heraus mit Werten zu füllen.

Wichtig: In PHP darf pro Klasse nur ein Konstruktor definiert werden.

```
class Kunde{
  private $nr;
  . . .

  public function __construct($nr, $nachname = "", $vorname = ""){
    $this->nr = $nr;
    $this->nachname = $nachname;
    $this->vorname = $vorname;
    . . .
  }
}
```

Der Konstruktor gehört zu den sogenannten **magischen Methoden** in PHP, die mit zwei Unterstrichen beginnen, festgelegte Namen haben und nicht explizit sondern automatisch vom System aufgerufen werden.

Neben dem Konstruktor gibt es noch weitere magische Methoden im Zusammenhang mit Klassen und Objekten. `__destruct()` wird aufgerufen, wenn das Objekt zerstört wird, bzw. am Ende des Skripts und `__toString()`, um eine Ausgabe der Klasse, bzw. der Objekteigenschaften als String zu implementieren.

Um die Kapselung realisieren zu können, werden alle Eigenschaften der Klasse mit private angegeben und für jede Eigenschaft eine get- und eine set-Methode geschrieben. Diese sogenannten **Getter** und **Setter** werden immer nach dem gleichen Prinzip erstellt. Vor den Namen der Eigenschaft wird ein "get", bzw. ein "set" geschrieben und dann mit großem Buchstaben weiter geschrieben.

```
public function setNickname($nickname){
  $this->nickname = $nickname;
}

public function getNickname(){
  return $this->nickname;
}
```

Auch alle anderen Methoden beginnen immer mit einem Verb und einem kleinen Buchstaben.

```
public function addAnzeige(Anzeige $anzeige){
  . . .
```

Hier darf ausnahmsweise der erwartete Datentyp (Anzeige) bei den Parametern einer Methode angegeben werden. So wird gewährleistet, dass nur ein Objekt vom Typ Anzeige übergeben werden kann.

1.5 Einbinden von Klassen

1.5.1 Per Include

In PHP gibt es verschiedene Methoden, um die unterschiedlichen Klassen innerhalb eines Projektes benutzen zu können. Man könnte über include die Klassen-Dateien in das benutzende PHP-Script eingebunden werden. Dies kann allerdings schnell unübersichtlich und fehleranfällig werden.

Zum Beispiel in der `index.php`:

```
<!DOCTYPE HTML>
<html>
<head>
  <title>OOP</title>
</head>
<body>
  <?php
    include_once "Anzeige.php";
    $kunde = new Anzeige(1, "Mustermann", "mustermann@irgendwo");
    echo $kunde->getNickname();
  ?>
</body>
</html>
```

1.5.2 Per magischer Methode __autoload()

Es könnte über die magische Methode__autoload() ein Mechanismus zum Einbinden der Klassen-Dateien programmiert werden.

```
function __autoload($class){
  include_once("./". $class .".php");
}
```

Diese Funktion muss am Anfang des Projekts einmal angelegt werden und importiert dann automatisch alle benötigten Klassen des Projektes. Der Nachteil ist, dass die Klassen nicht in verschiedenen (Unter-)Verzeichnissen sein dürfen und keine unterschiedlichen Endungen haben dürfen.

Komfortabler wird es aber, wenn mit `spl_autoload_register()` der dort implementierte Autoload-Mechanismus aktiviert wird. Dann werden alle Klassen, die sich im aktuellen Verzeichnis befinden und die passende Extension besitzen, eingebunden. Normalerweise sucht PHP im aktuellen Verzeichnis und im Verzeichnis `"\xampp\php\PEAR"` nach Klassen.

Es können auch eigene Autoload-Mechanismen dort registriert werden. Allerdings wird dann unter Umständen der Standardmechanismus außer Kraft gesetzt. Weitere Endungen (Extension) können über `spl_autoload_extensions()` bekannt gemacht werden. Diese Funktionen gehören zur SPL (Standard PHP Library).

```
function myAutoload ( $class ) {
  include 'classes/' . $class . '.php';
}
spl_autoload_register('myAutoload');

$extension = spl_autoload_extensions();
spl_autoload_extensions( $extension . ",.class.php" );
```

1.5.3 Per Namespace

Benutzt man Namespaces können auch Unterverzeichnisse verwendet werden, ohne dass verschiedene Autoloader registriert werden müssen.

Für jede Klasse wird der Pfad hinter dem Schlüsselwort **`namespace`** angegeben. Dafür werden Backslash benutzt. Der Pfad wird ausgehend von der index.php angegeben. Der jeweilige Projektordner (hier: `webbrett`) ist dabei unwichtig und wird nicht angegeben.

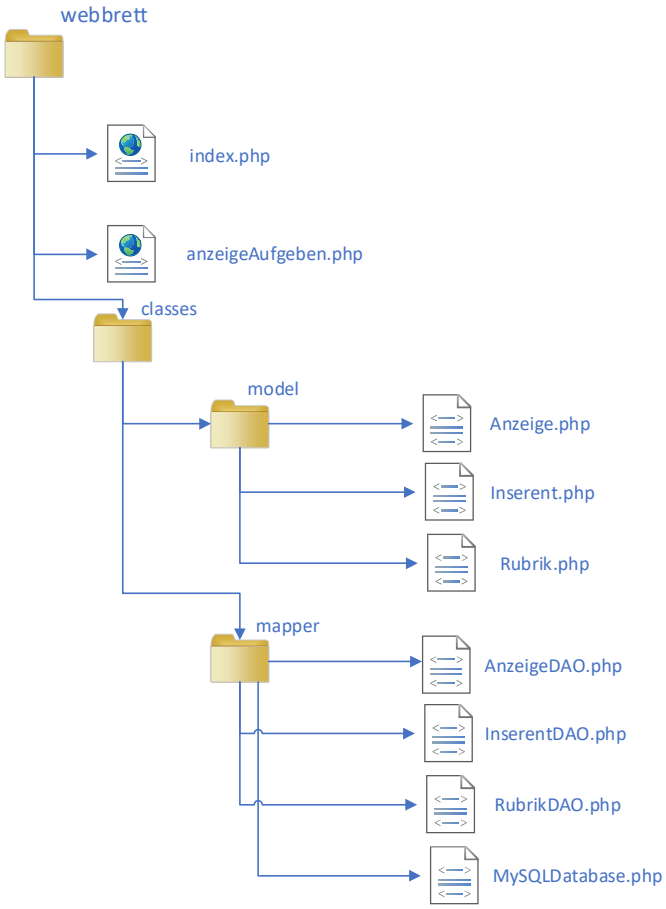
Wenn eine Klasse in einem PHP-Skript oder einer anderen Klasse benutzt wird, muss mit dem Schlüsselwort **`use`** der Pfad zur Klasse angegeben werden.

Für webbrett/classes/mapper/AnzeigeDAO.php:

```
<?php
   namespace classes\mapper;

   use classes\model\Anzeige;

   class AnzeigeDAO{
      . . .
   }
?>
```



Klassen, die sich im gleichen Verzeichnis befinden, müssen nicht über das Schlüsselwort `use` bekanntgemacht werden. So muss in der Klasse `Anzeige` nicht mittels `use` die `Inserenten`-Klasse angegeben werden.

Für webbrett/classes/model/Anzeige.php:

```
<?php
namespace classes\model;

class Anzeige{

   . . .

   public function setInserent(Inserent $inserent){
      $this->inserent  = $inserent;
   }
}
```

Für PHP-Skriptdateien, die keine Klassen sind, wie die `index.php` oder `anzeigeAufgeben.php` entfällt die `namespace`-Angabe.

Für webbrett/index.php:

```
<?php
   use classes\model\Inserent;
   use classes\model\Anzeige;
   use classes\model\Rubrik;

   include_once("./autoload.php");
   spl_autoload_register('autoloader');
?>

<!DOCTYPE html>
. . .
<?php
   $inserent1 = new Inserent(101, "mickey", "mickeymouse@entenhausen.com");
```

Eine wichtige Rolle spielt dabei die Standard PHP Library (SPL). Diese ist eine Erweiterung für PHP 5 und ist standardmäßig aktiviert. Das automatische Laden von Klassen-Dateien wurde schon mit `__autoload()` erklärt. Die Funktion spl_`autoload_register()` kann der interne Autoload-Mechanismus über die magische Funktion `__autoload()` deaktiviert werden und der Mechanismus der SPL angewendet werden. Dieser Mechanismus bewirkt ohne Angabe eines Namens das Anhängen von `.php` und .inc an den Klassennamen und das Einbinden der entsprechenden Klassendatei. Mit Angabe eines Namens als Parameter, kann jede beliebige Funktion angeben werden, die einen eigenen Mechanismus implementiert. Das ist z. B. bei UNIX-kompatiblen PHP-Skripten erforderlich.

Auf jeden Fall muss einmal am Anfang die Funktion `spl_autoload_register('autoloader')` aufgerufen werden, um die Funktion `autoloader()` einzutragen, die sich in der Datei `autoload.php` befindet.

Außerdem ist es nötig, Standardklassen mit `\klassenname` aufzurufen.

```
new \mysqli(DB_SERVER, DB_USER, DB_PASSWORD, DB_DATABASE);
```

2. PHP und MySQL

2.1 Einführung

Innerhalb von PHP gibt es verschiedene Implementierungen, um MySQL-Datenbankzugriffe und –manipulationen zu realisieren. Hier werden drei APIs (Application Programming Interface) vorgestellt und miteinander verglichen: mysql-, mysqli- und PDO Erweiterung.

Die MySQL-API ist die älteste und sollte nicht mehr verwendet werden. Sie wird nicht mehr weiterentwickelt und wird in neueren PHP-Versionen nicht mehr zur Verfügung gestellt.

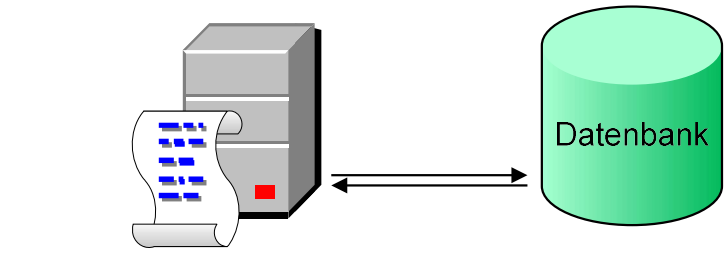
Die MySQLi-API stellt sowohl die Möglichkeit strukturiert und objektorientiert auf die Datenbankfunktionalitäten zugreifen zu können.

Mit der PDO-API ist nur eine objektorientierte Schnittstelle geschaffen worden. Sie hat aber den Vorteil, dass hier verschiedene Datenbankserver unterstützt werden. So ist es möglich PHP-Webanwendungen auch auf andere Datenbanksysteme zu übertragen.

	ext/mysqli	**PDO_MySQL**	**ext/mysql**
PHP version introduced	5.0	5.1	2.0
Included with PHP 5.x	Yes	Yes	Yes
Development status	Active	Active	Maintenance only
Lifecycle	Active	Active	Deprecated
Recommended for new projects	Yes	Yes	No
OOP Interface	Yes	Yes	No
Procedural Interface	Yes	No	Yes
API supports non-blocking, asynchronous queries with mysqlnd	Yes	No	No
Persistent Connections	Yes	Yes	Yes
API supports server-side Prepared Statements	Yes	Yes	No
Supports all MySQL 5.1+ functionality	Yes	Most	No

Quelle: https://php.net/mysqlinfo.api.choosing

Im Folgenden werden wir uns mit der MySQLi-Extension beschäftigen und die Zugriffe auf die MySQL-Datenbank realisieren



Webserver mit PHP-Interpreter

2.2 MySQLi-Extension

Die MySQL improved Extension steht für eine Erweiterung, die ab PHP 4.1.3 verwendet werden kann. Hier gibt es einen prozeduralen Ansatz und einen objektorientierten.

Beispiel:

- Prozedural:

```
mixed mysqli_query ( mysqli $link , string $query
[, int $resultmode = MYSQLI_STORE_RESULT ] )
```

- Objektorientiert

```
mixed mysqli::query ( string $query
[, int $resultmode = MYSQLI_STORE_RESULT ] )
```

Wir werden den objektorientierten Ansatz verfolgen, weil es der einzige sinnvolle Ansatz im Zusammenhang mit der objektorientierten Programmierung ist.

Unter MySQLi gibt es verschiedene Klassen, die unterschiedliche Aufgaben haben. Mysqli-Klasse ist für die Verbindung zum MySQL-Datenbankserver verantwortlich. Hier sind Methoden und Eigenschaften angesiedelt, die den Status des Servers überprüfen, eine Abfrage absenden oder Fehler abfragen.

Die MySQLI-Statementklasse ermöglicht zum Beispiel eine vorbereitete Abfrage. Eine vorbereitete Abfrage (prepared statement) ermöglicht die typsichere Abfrage oder Datenmanipulation und beugt auf diese Art SQL-Injection vor.

Die MySQLi-Ergebnisklasse beinhaltet nur wenige Methoden, die die Darstellung der Ergebnisse einer Abfrage realisieren.

Die MySQLi_driver-Klasse liefert Informationen über den benutzten Treiber. Die Fehlerbehandlungs-Klasse MySQLi-Exception hat von der Klasse RuntimeException geerbt und dient zur Exceptionbehandlung in eine try-catch-Block.

3. MySQLi

3.1 Aufbau einer Datenbank-Verbindung

Wie auch über die Eingabeaufforderung oder die Shell muss zuerst eine Verbindung zum Datenbankserver mittels Angabe des Servers, des Benutzernamens und des Kennworts aufgebaut werden. Danach würden wir mit dem Befehl `use <datenbankname>` die Datenbank, mit der wir arbeiten wollen, setzen. Diese beiden Vorgänge werden über einen Befehl realisiert.

Beispiel:

```
<?php
   $dbConnection = new mysqli('localhost', 'phpuser', 'geheim',
                                    'webbrett');

   if ($dbConnection->connect_error) {
      die('Verbindungsfehler (' . $dbConnection->connect_errno . ') ');
   }
   // Verarbeitung kann erfolgen

   $dbConnection->close();
?>
```

Besserer Stil ist es die Anmeldedaten in Variablen zu schreiben und sogar in eine externe Datei auszulagern. Der Vorteil ist hier, dass bei Verschieben einer Anwendung auf ein anderes System, nur noch an einer Stelle – der externen Datei verbindung.inc.php – die Daten angepasst werden müssen.

Damit diese Datei besser gefunden wird und später auch durch Verzeichniszugriffsrechte abgesichert werden kann, sollte sie unbedingt in einem Unterordner des Projekts angelegt werden.

Beispiel:

Datei verbindung.inc.php:

```
<?php
   $dbserver = "localhost";
   $dbuser = "phpuser";
   $dbpassword = "geheim";
   $dbname = "webbrett";
?>
```

Datei index.php:

```
<?php
   include("./include/verbindung.inc.php");
   $dbConnection = new mysqli($dbserver, $dbuser, $dbpassword, $dbname);
```

oder als Konstanten:

```
   <?php
      //definiert die Datenbankverbindungsparameter<.
      define ('DB_SERVER', "localhost");
      define ('DB_USER', "phpuser");
      define ('DB_PASSWORD', "geheim");
      define ('DB_DATABASE', "webbrett");
   ?>
```

3.2 Datenbankabfrage

Nachdem wie oben beschrieben eine Datenbankverbindung erfolgreich aufgebaut worden ist, werden die Daten abgefragt. Eine Möglichkeit ist die Methode mysqli_query, bzw. mysql::query. Hier wird der SQL-String übergeben und ein Verweis auf die Ergebnismenge zurückgeliefert.

Die Ergebnismenge muss über entsprechende Methoden aufbereitet werden. Wir verwenden die Methode mysqli_fetch_array, bzw. mysqli_result::fetch_array , die uns ein Array der abgefragten Spalten zurückliefert.

Beispiel:

```
<?php
   ...
   $abfrage = "SELECT rubriknummer, rubrikbezeichnung
          FROM rubrik order by rubrikbezeichnung";
   $ergebnis = $dbConnection->query($abfrage);

   $i = 0;
   while($zeile = $ergebnis ->fetch_array(MYSQLI_NUM)){
      $rubriknummer[$i] = $zeile[0];
      $rubrikbezeichnung[$i] = $zeile[1];
      $i++;
   }
   $ergebnis->free();
   ...
?>
```

Die Spalten sind im unten angegebenen Beispiel rubriknummer und rubrikbezeichnung. Eine Zeile enthält dann jeweils die Nummer und die Bezeichnung:

```
   $zeile[0]
   $zeile[1]
```

$zeile[0] enthält im ersten Schleifendurchgang 501 und $zeile[1] die Zeichenkette "Autos". Das bedeutet $zeile[0] steht für die Rubriknummer und $zeile[1] für die Rubrikbezeichnung.

Die Konstante MYSQLI_NUM steht dabei für die numerische Variante das Array $zeile auszulesen. Entsprechend dazu gibt es die Konstante MYSQLI_ASSOC, die die Angabe der Spaltenbezeichnung erforderlich macht:

```
   $zeile['rubriknummer']
   $zeile['rubrikbezeichnung']
```

Die Methode mysqli_query, bzw. mysql::query gibt bei der Abfrage von Daten mittels SELECT-Kommando false zurück, wenn die Abfrage fehlerhaft war.

Wenn die Abfrage erfolgreich war, wird die Ergebnismenge vom Typ mysqli_result zurückgegeben und die für diese Klasse vorhandenen Methoden können benutzt werden.

Auf diese Art kann auch überprüft werden, ob ein SQL-Statement erfolgreich war. Allerdings muss beachtet werden, dass hier eine Überprüfung mit Typsicherheit notwendig ist

```
<?php
   ...
   $abfrage = "CREATE TABLE temp LIKE anzeige";
   if ( $dbConnection->query($abfrage) === true) {
     echo "Anlegen der Tabelle temp war erfolgreich!";
   }
   ...
?>
```

Eine andere Möglichkeit, die etwas ressourcensparender ist, ist

```
mysqli_result::fetch_all ([ int $resulttype = MYSQLI_NUM ] ) : mixed
```

Hierbei wird ein zweidimensionales Array mit einer Anweisung ausgelesen und kann danach verarbeitet werden.

Beispiel:

```
$resultArray = $resultData->fetch_all();

for($i = 0; $i < count($resultArray); $i++){
  $rubrikList[] = new Rubrik($resultArray[$i][0], $resultArray[$i][1]);
}

$resultData->free();
```

Auch hier gibt es auch wieder eine assoziative Variante:

Beispiel:

```
$resultArray = $resultData->fetch_all(MYSQLI_ASSOC);

for($i = 0; $i < count($resultArray); $i++){
  $rubrikList[] = new Rubrik($resultArray[$i]['rubriknummer'],
                      $resultArray[$i]['rubrikbezeichnung']);
}

$resultData->free();
```

Wenn die Abfrage erfolgreich war, kann zum Beispiel die Anzahl der Datensätze mit

```
int $mysqli_result::num_rows;
```

abgefragt werden.

Beispiel:

```
<?php
  ...
  if ($ergebnis = $dbConnection->query($abfrage)) {
    $anzahlZeilen = $ergebnis->num_rows;
  }
  ...
?>
```

3.3 Datenbankmanipulation

3.3.1 Einfügen

Das Einfügen von neuen Datensätzen mittels INSERT ist ähnlich wie das Abfragen von Daten zu realisieren. Auch hier muss vorher eine Datenbankverbindung aufgebaut werden. Dann kann das SQL-Kommando angegeben werden. Unten ist eine andere Art über die Funktion `sprintf` angegeben. Hier muss jeder Parameter, der mit Platzhalter %s für Zeichenkette, %d für Integer und %f für Float eingetragen wurde, der Funktion mit Komma getrennt übergeben werden.

```
echo sprintf ( %s %d %f , "Hallo" , 1 , 99.45) ;

// Ausgabe: Hallo 1 99.45
```

Dabei realisiert `sprint` keine Ausgabe wie `echo` oder `printf`, sondern formatiert nur den übergegebene Zeichenkette und gibt sie zurück.

Beispiel:

```
<?php
   ...
   $nickname = "hugo";
   $email = "hm@t-online.de";
   $anzeigentext = "PHP 5.3 + MySQL 5.1, Dr. Florence Maurice";

   $ abfrage = sprintf("INSERT INTO inserent
               set nickname='%s', email='%s'" ,
               $nickname, $email);
   $dbConnection ->query($abfrage);
   $inserentennummer = $dbConnection->insert_id;
   ...
?>
```

Mit der Methode, bzw. Eigenschaft mysqli_insert_id, bzw. mysqli::$insert_id kann der Primärschlüssel des zuletzt eingefügten Datensatzes ausgelesen werden. Diese Funktionalität wird benutzt, wenn ein neuer Inserent eine Anzeige aufgegeben hat. Zuerst muss dann der Inserent eingelesen werden und die neue Inserentennummer abgefragt werden, um danach erst die Anzeige in der Tabelle anzeige mit dem Fremdschlüssel inserentennummer einzutragen.

3.3.2 Löschen und Ändern

Ähnlich werden die Aktionen löschen oder ändern eines Datensatzes mit DELETE, bzw. UPDATE-Befehl realisiert. Dabei kann mit mysqli::affected_rows, bzw. mysqli_affected_rows die Anzahl der Datensätze abgefragt werden, die gelöscht oder geändert wurden.

Beispiel:

```
<?php
   ...
   $abfrage = "DELETE FROM anzeige where YEAR(anzeigendatum) = 1999"
   $dbConnection->query($abfrage);
   echo "Betroffene Datensätze: " . $dbConnection->affected_rows;
   ...
?>
```

3.4 Prepared Statement

3.4.1 Überblick

Wie schon in Kapitel 3.2 beschrieben beinhalten Prepared Statements den Vorteil, dass die übergebenen Werte typsicher sind. Das heißt, dass ein String an der entsprechenden Stelle auch als String-Wert eingesetzt wird und nicht als Teil des SQL-Kommandos.

Bei der Verarbeitung eines Prepared Statements übernimmt der Datenbankserver das SQL-Kommando und schreibt es vorübersetzt in eine Art Puffer. Danach werden die Platzhalter nur noch eingefügt und schadhafter Code kann nicht mehr ausgeführt werden. So werden SQL-Injections verhindert und zudem kann die Performance erhöht werden, wenn ein SQL-Befehl mehrmals mit verschiedenen Werten für die Platzhalter ausgeführt wird.

Beispiel:

```
<?php
   ...
   $inserent = "sissi";
   $abfrage = "SELECT email FROM inserent WHERE nickname=?";

   if($preparedStatement = $dbConnection->prepare($abfrage)){
      $preparedStatement->bind_param("s", $inserent);
      $preparedStatement->execute();
      $preparedStatement->bind_result($emailadresse);

      if($preparedStatement->fetch() ) {
         printf("%s hat die E-Mailadresse %s",
                $inserent,
                $emailadresse);
      }

      $preparedStatement->close();
   }
?>
```

Hier wird der SQL-Befehl vorbereitet und mit der Methode `mysqli_stmt_bind_param`, bzw. `mysqli_stmt::bind_param` wird eine Bindung der Variablen zu dem Platzhalter (?) im SQL-Befehl hergestellt. Wichtig ist hierbei, dass der Typ korrekt angegeben wird. Werden mehrere Platzhalter in einem SQL-Befehl verwendet, muss jeder Platzhalter durch einen Buchstaben und ein Attribut vertreten sein.

```
//s .. String, i .. Integer, d .. Double
$preparedStatement->bind_param("is", $inserentennummer, $inserent);
```

Über die Methode mysqli_stmt_execute, bzw. mysql_stmt::execute wird der Platzhalter in den SQL-Befehl eingesetzt und verarbeitet. Das kann auch mehrfach mit unterschiedlichen Werten erfolgen.

Wichtig:
Für Abfragen, die eine Ergebnismenge zurückliefern, muss unter gewissen Umständen das Ergebnis zwischengepuffert werden. Dafür gibt es die Methode, bzw. Funktion `mysqli_stmt::store_result`, bzw. `mysqli_stmt_store_result`.

So muss zum Beispiel eine Ergebnismenge zwischengespeichert werden, wenn die Anzahl der Datensätze abgefragt werden soll.

Beispiel:

```
<?php
   ...
$abfrage = "select anzeigentext, anzeigendatum from anzeige where
day(anzeigendatum) = day(now())";

if ($preparedStatement = $dbConnection->prepare($abfrage)) {
    $preparedStatement->execute();

// Puffern der Abfrageergebnisse:
    $preparedStatement->store_result();
    echo("Anzahl der Zeilen " . $preparedStatement->num_rows);

    $preparedStatement->free_result();
    $preparedStatement->close();
 . . .
?>
```

Auch wenn zwei Abfragen parallel aufgerufen werden soll, ist eine Pufferung notwendig.

3.4.2 Datenmanipulationen mit Prepared Statements

Wie schon im vorherigen Kapitel erwähnt kann mittels Prepared Statements auch ein SQL-Befehl mit verschiedenen Werten mehrmals ausgeführt werden. Dies ist praktisch bei INSERT-Befehlen.

Wir wollen im folgenden Beispiel verschiedene Rubriken zum Testen anlegen und uns die mühevolle Tipparbeit sparen. Dabei wird die `rubriknummer` in einer Schleife generiert und die `rubrikbezeichnung` einfach auf "Rubrik" gesetzt. Wichtig ist hier, dass die Parameter in der richtigen Reihenfolge angegeben werden:

`"is", $id, $bez` -> `i` steht für `$id` und `s` für `$bez`

Beispiel:

```
<?php
   ...
   $abfrage = "INSERT INTO rubrik(rubriknummer, rubrikbezeichnung)
          VALUES (?,?)";

   if($preparedStatement = $dbConnection->prepare($abfrage)){

      $id = 1;
      $bez = "Rubrik";

      if(!$preparedStatement->bind_param("is", $id, $bez)){
         echo $dbConnection->error . "<br>";
      }
      else{
         if(!$preparedStatement->execute()){
            echo $dbConnection->error . "<br>";
         }
         else{
            for ($id = 2; $id < 5; $id++) {
               if(!$preparedStatement->execute())
                  echo $dbConnection->error . "<br>";
            }
         }
      }
      $preparedStatement->close();
   } ...
?>
```

4. Aufbau von Anwendungen ohne Entwurfsmuster

Es hat sich aus verschiedenen Gründen nicht bewährt, eine Vielzahl von PHP-Skripten mit verschiedenen Aufgaben zu entwickeln. Ein Grund ist, dass die Übersichtlichkeit mit jeder Datei abnimmt und auch die Namensgebung für die Skriptdateien viel Kreativität erfordert (siehe unten).

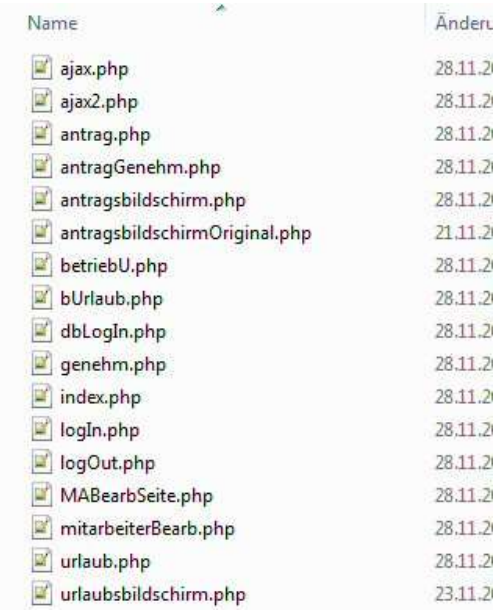


Abb. 4 Negativbeispiel für Verwendung von PHP-Skriptdateien

Man sollte die Anwendungsfälle/ Aufgaben der Software so aufteilen, dass ein Skript genau eine Aufgabe hat. Außerdem sollte die Verarbeitung einer Anfrage, das Auslesen von Datenbanken und das Anzeigen jeweils auf verschiedene Arten von Klassen oder Skripte aufgeteilt werden. Es bietet sich hier ein vereinfachtes Model-View-Controller Pattern an. Das MVC-Muster teilt dabei die Aufgaben in den Teil der Geschäftslogik (model), den Teil der Anzeige (view) und den Verarbeitungsteil von Anfragen (controller) ein.

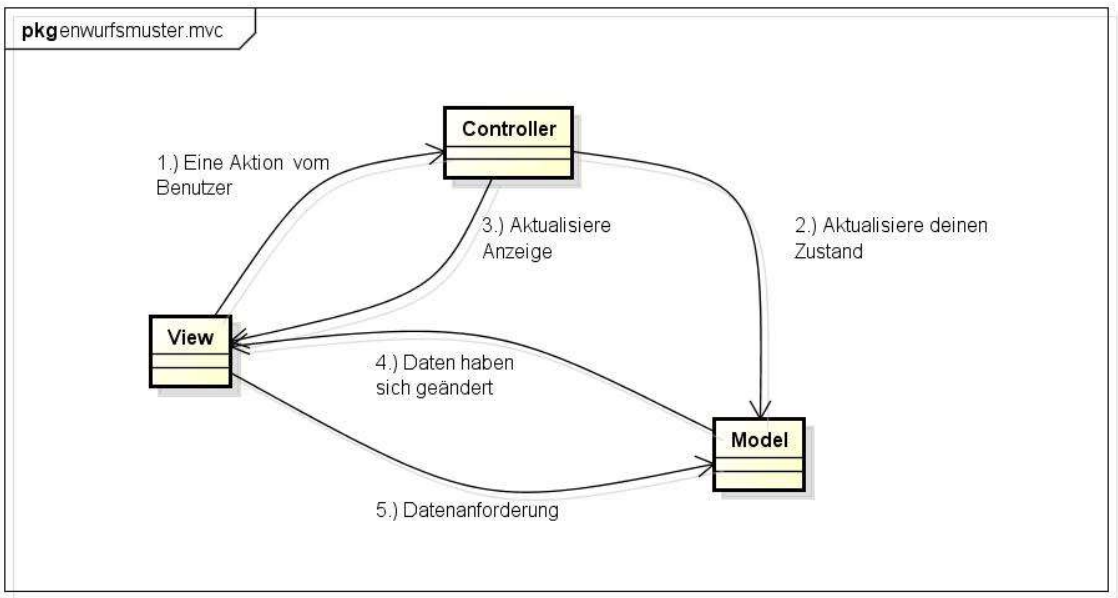


Abb. 5 Negativbeispiel für Verwendung von PHP-Skriptdateien

Auch für jeden einzelnen Bereich (model, view, controller) gibt es dann eine klare Aufgabenverteilung. Ein View zum Anzeigen aller Rubriken, ein View zum Anzeigen aller Anzeigen einer Rubrik und ein View zum Eingeben einer Anzeige.

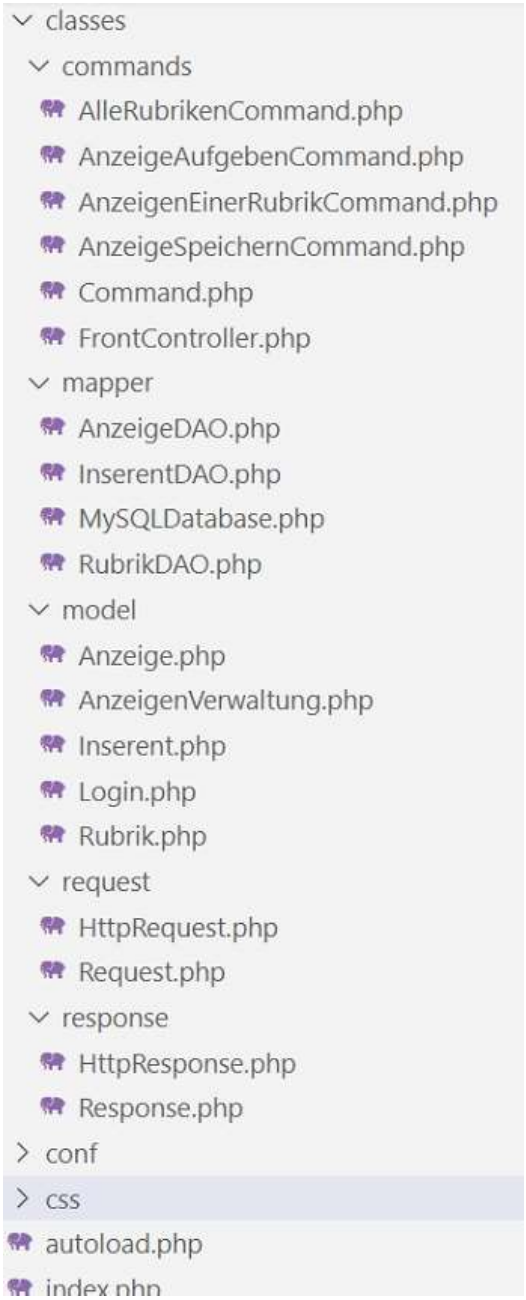


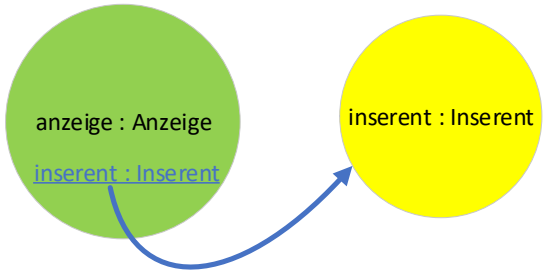
Abb. 6 Verwendete Ordnerstruktur ohne Views mit Model (inkl. Mappern)

5. Entwurfsmuster

5.1 Data-Mapper

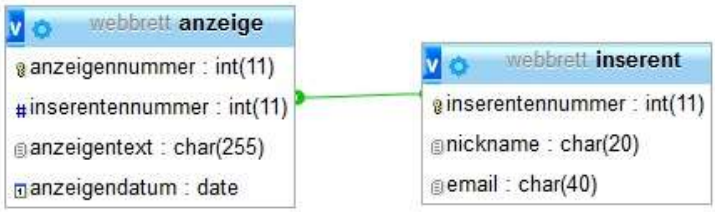
Ein Problem, bei der objektorientierten Programmierung ist der Unterschied der Datenhaltung im Vergleich zum relationalen Datenbanksystem. Bei der Objektorientierung besteht die Verbindung zwischen zwei Klassen aus der Eigenschaft, die ein Objekt enthält.

Beispiel: Jede Anzeige enthält einen Inserenten (siehe Kap. 1.1) als Eigenschaft



Im relationalen Datenbanksystem würden hier zwei Tabellen angelegt werden, die über eine Fremdschlüsselbeziehung die Adresse zum Inserenten zuordnet.

Beispiel: In der Tabelle `anzeige` ist ein Fremdschlüssel mit der `inserentennummer` enthalten



Um diese andere Art der Datenhaltung zu berücksichtigen, gibt es die Technik des *object-relational Mapping*, *ORM,* welches es ermöglich Objekte in relationalen Datenbank-Tabellen abzulegen. Hier gibt es diverse Frameworks, die das erledigen.

Das ist Ziel ist die Datenverwendung von der Datenhaltung zu trennen. Dafür gibt es unter anderem das Data Mapper-Entwurfsmuster, bei welchem es für jede Klasse eine Mapperklasse gibt, die für die Persistierung der Objekte zuständig ist. Die Klassen der Geschäftslogik werden dabei vollkommen frei von SQL-Code gehalten. So kann jederzeit die Persistenzschicht verändert werden, in dem einfach eine andere Mapperschicht verwendet wird.

Die Mapper-Klassen enthalten die CRUD-Operationen create (erzeugt einen Datensatz), read (liest einen oder mehrere Datensätze), update (ändert einen Datensatz) und delete (löscht einen Datensatz). In manchen Implementationen wird hier mit Interfaces gearbeitet, um sicherzustellen, dass jede Persistenzschicht die gleichen Methoden zur Verfügung stellt. Dieses entspricht dann dem Factory Design Pattern (vergleiche http://www.oracle.com/technetwork/java/dataaccessobject-138824.html).

Für die Rubrik-Klasse gibt es eine RubrikDAO-Klasse (DAO data access control), die die CRUD-Operationen (create, read, update und delete) beinhaltet.

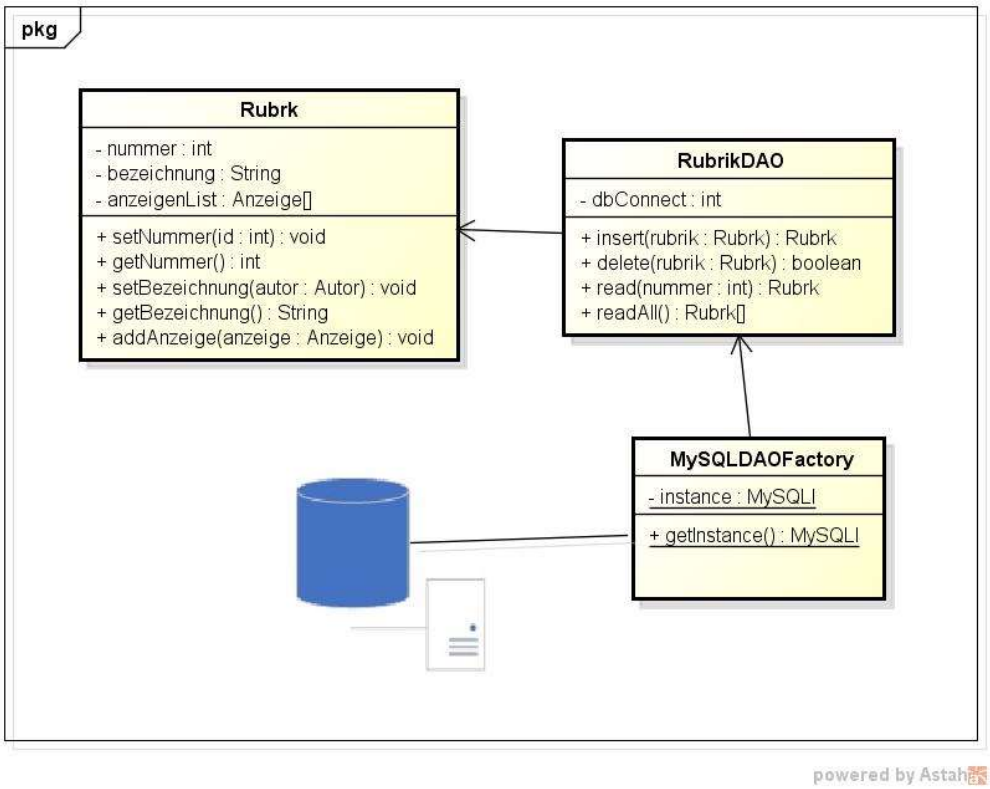


Abb. 7 Data Mapper-Klassendiagramm

Das Data Mapper-Entwurfsmuster kann gut mit dem Singleton-Entwurfsmuster für die Datenbankverbindung ergänzt werden. Das Singleton-Entwurfsmuster sichert ab, dass von einer Klasse nur ein Objekt existiert. Es gibt eine Eigenschaft, die statisch ist, und die Klasse stellt einen zentralen Zugriffspunkt zur Verfügung.

Hier eine stark vereinfachte Version des Singleton-Patterns:

```
class MySQLDatabase{
  private static $instance;

  public static function getInstance(){
    if(!self::$instance){
      try{
        self::$instance  =
            new \mysqli(DB_SERVER, DB_USER, DB_PASSWORD, DB_DATABASE);
      }
      catch(Exception $e){
        echo self::$instance->connect_error;
      }
    }
    return self::$instance;
  }
}
```

Dabei stellt `$instance` die Datenbankverbindung dar, auf die nur über `getInstance()` zugegriffen werden kann. Entweder ist eine gültige Datenbankverbindung vorhanden oder es wird eine erzeugt.

Wichtig: Statische Eigenschaften werden nicht über `this->`, sondern über `self::` innerhalb der Klasse angesprochen. Außerhalb der Klasse wird über den Klassennamen und die Methode zugegriffen:

```
MySQLDatabase::getInstance()
```

Um die eigentliche Datenzugriffsklasse zu entwickeln, muss man als Eigenschaft eine Verbindung zur Datenbank aufbauen und die erforderliche CRUD-Funktionalität umsetzen.

Für die Klasse Rubrik benötigen wir das Lesen aus der Datenbank von allen Rubriken und das Auslesen einer Rubrik anhand einer übergebenen Nummer.

```
namespace classes\mapper;
use classes\model\Rubrik;

class RubrikDAO{
  private $dbConnect;

  public function __construct (){
    $this->dbConnect = SQLDAOFactory::getInstance();
  }

  public function readAll(){
    $sql = "SELECT rubriknummer, rubrikbezeichnung FROM rubrik";
    $rubrikList = array();

    $resultData = $this->dbConnect->query($sql);

    while($row = $resultData->fetch_array(MYSQLI_ASSOC)){
      $rubrikList[] = new Rubrik($row["rubriknummer"],
                                 $row["rubrikbezeichnung"]);
    }

    $resultData->free();

    return $rubrikList;
  }

  public function read($nummer){
    $sql = "SELECT rubriknummer, rubrikbezeichnung " .
           "FROM rubrik " .
           "WHERE rubriknummer=?";
    $rubrik = null;

    $preStmt = $this->dbConnect->prepare($sql);
    $preStmt->bind_param("i", $nummer);
    $preStmt->execute();
    $preStmt->bind_result($nummer, $name);

    if($preStmt->fetch()){
      $rubrik = new Rubrik($nummer, $name);
    }
    $preStmt->free_result();
    $preStmt->close();

    return $rubrik;
  }
}
```

Quellcode: RubrikDAO.php

Eingebunden werden, können diese Abfrage dann entsprechend:

```
  $rubrikDAO = new RubrikDAO();
  $rubrikList = $rubrikDAO->readAll();

  $rubrik = $rubrikDAO->read($rubriknr);
```

5.2 Front-Controller-Pattern

Das Front-Controller-Pattern basiert auf der Idee, dass es für alle Geschäftsvorfälle einen einzigen Einstiegspunkt gibt und danach in Abhängigkeit vom angehängten Wert auf andere Controller, bzw. Commands weitergeleitet (geroutet) wird. Auch sollen alle Routinen, die Ein- und Ausgaben realisieren, an einem Punkt verarbeitet werden.

Der FrontController verarbeitet alle HTTP-Requests (Anforderungen) über den angehängten Parameter `cmd`. Das heißt, der Aufruf erfolgt über `index.php?cmd=`*`CommandName`*. Für jede Aufgabe gibt es eine Command-Klassen, die die Anfrage verarbeitet. Der FrontController holt sich alle übergebenen Name-Werte-Paare über die Klasse `HttpRequest` und alle Ausgaben werden über die Klasse `HttpResponse` an den Webbrowser zurückgeschickt.

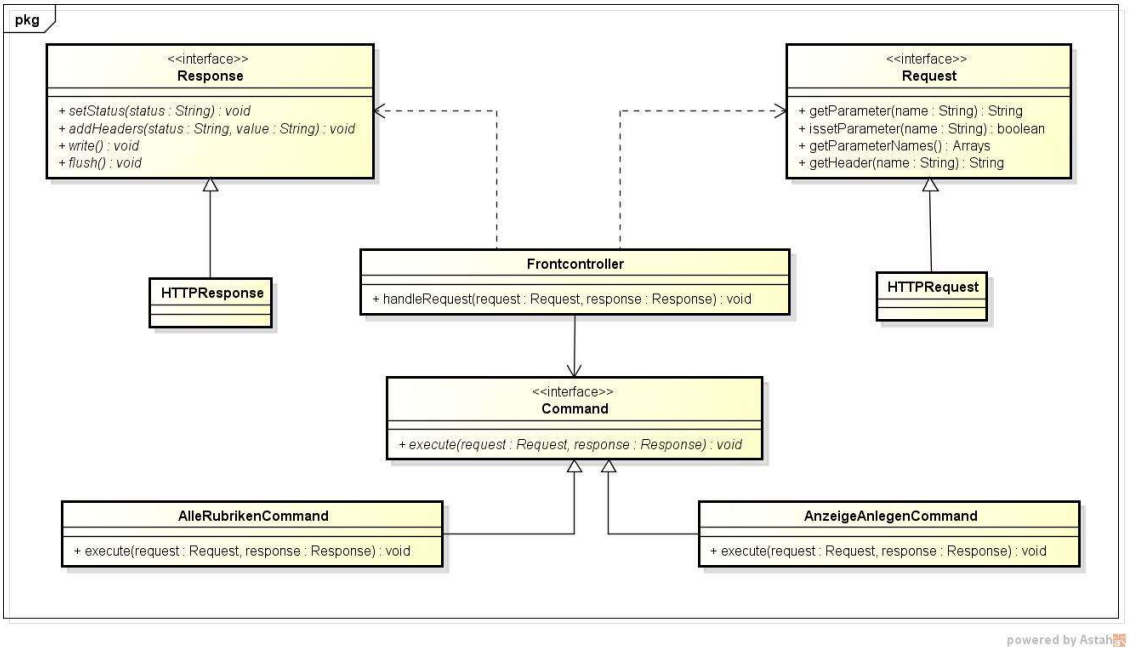


Abb. 8 Front-Controller-Klassendiagramm

Ausschnitt aus der index.php:

```
$request = new HttpRequest();
$response = new HttpResponse();
$controller = new FrontController('classes\commands', 'AlleRubriken');

$controller->handleRequest($request, $response);
```

Die Argumente, die beim Erzeugen des FrontController-Objekts übergeben werden, ist der Pfad, in dem sich alle konkreten Command-Klassen befinden und der Name der Default-Command-Klasse, die aufgerufen wird, wenn die index.php ohne cmd-Parameter aufgerufen wird.

Dabei wird gegen Schnittstellen (Interface) programmiert, um unabhängig von der konkreten Implementierung zu sein.

Ein **Interface** ist eine Art Klasse, die nur Methodenköpfe ohne Code enthält. Wenn eine konkrete Klasse ein Interface implementiert, dann müssen die Methodenköpfe als konkrete Methoden realisiert worden sein.

Wichtig: Es gibt ein Interface `Request` und eine Klasse `HttpRequest`, die das Interface implementiert hat. So ist es möglich, noch weitere Klasse zu entwerfen, die aber alle die gleichen vorgegebenen Methoden wie das Interface haben und somit sind die konkreten Klassen austauschbar.

Beispiel:

```
interface Request{
   public function getParameterNames();
   public function issetParameter($name);
   public function getParameter($name);
   public function getHeader($name);
}
```

Quellcode: Request.php

```
class HttpRequest implements Request{
   private $parameters;

   public function __construct(){
      $this->parameters = $_REQUEST;
   }

   public function getParameterNames(){
      return array_keys($this->parameters);
   }

   public function issetParameter($name){
      return isset($this->parameters[$name]);
   }

   public function getParameter($name){
      if (isset($this->parameters[$name])) {
         return $this->parameters[$name];
      }
      return null;
   }

   public function getHeader($name){
      $name = "HTTP_" . strtoupper(str_replace("-", "_", $name));
      if (isset($_SERVER[$name])) {
         return $_SERVER[$name];
      }
      return null;
   }
}
```

Quellcode: HttpRequest.php

Egal, ob über die Methoden post oder get eine Anforderung an den Webserver gesendet wird, speichert der PHP-Interpreter die Name-Werte-Paare auch im Array `$_REQUEST`. Dieses wird in die Eigenschaft `parameters` unserer `HTTPRequest`-Klasse gespeichert.

`getParameterNames()` – gibt ein Array mit allen Namen (Schlüsseln) von $_REQUEST zurück.
`issetParameter()` – überprüft, ob ein bestimmter Name (Schlüssel) vorhanden ist
`getParameter()` – gibt den Wert eines Namens (Schlüssels) zurück
`getHeader()` – liefert den Wert eines bestimmten HTTP-Headers zurück.

Für die konkreten Command-Klassen wird der gleiche Ansatz verfolgt. Da es beliebig viele konkrete Command-Klassen geben kann, wird ein Interface `Command` erstellt, welches die Methode `execute` als Methodenkopf enthält. Das heißt, alle Klassen, die das Command-Interface implementieren, müssen mindestens die Methode `execute` enthalten.

```
interface Command{
  public function execute(Request $request, Response $response);
}
```

Quellcode: Command.php

```
class AlleRubrikenCommand implements Command{
  public function execute(Request $request, Response $response){
    . . .
  }
}
```

Quellcode: AlleRubrikenCommand.php

Innerhalb der Methode `execute()` steht der Code, der für die konkrete Command-Klasse notwendig ist. So können übergebene Name-Werte-Paare ausgelesen werden und Ausgaben über die Methode `write()` an das Objekt `$response` übergeben werden.

```
  $rubriknr = $request->getParameter("nr");
  $anzeigenList = $anzeigeDAO->readByRubrik($rubriknr);
  $rubrik = $rubrikDAO->read($rubriknr);
  $response->write("<h3>" . rubrik->getBezeichnung() . "</h3>");

  foreach($anzeigenList as $anzeige){
    $response->write("Anzeigendatum: " . $anzeige->getDatum() . "</br>");
    if($anzeige->getInserent() !== null){
      $response->write($anzeige->getInserent()->getNickname() . "</br>");
      $response->write($anzeige->getInserent()->getEmail() . "</br>");
    }
    $response->write($anzeige->getText() . "</br> ");
  }
```

Ausschnitt aus einer möglichen Command-Implementierung

Die für die Ausgabe gedachte Klasse `HttpResponse` implementiert auch ein Interface, da es auch hier möglich sein sollte eine andere Verarbeitung zu wählen und eine weitere Implementierung von Response zu entwickeln. So könnte es sein, dass alle Ausgaben in eine Pdf-Datei geschrieben werden, die den Webbrowser als Antwort gesendet wird. Dafür muss aber auch wieder gewährleistet sein, dass der FrontController eine Methode `addHeaders()` oder `flush()` aufrufen kann.

```
interface Response{
  public function setStatus($status);
  public function addHeaders($name, $value);
  public function write($data);
  public function flush();
}
```

Quellcode: Response.php

```
class HttpResponse implements Response{
  private $status;
  private $headers;
  private $body;

  public function __construct(){
    $this->status = "200 OK";
    $this->headers = array();
    $this->body = null;
  }

  public function setStatus($status){
    $this->status = $status;
  }
  public function addHeaders($name, $value){
    $this->headers[$name] = $value;
  }
  public function write($data){
    $this->body .= $data;
  }
  public function flush(){
    header("HTTP/1.0 " . $this->status);
    foreach ($this->headers as $name => $value){
      header($name . ":" . $value);
    }
    print $this->body;
    $this->headers = array();
    $this->data = null;
  }
}
```

Quellcode: HttpResponse.php

`setStatus()` – Setzen des HTTP-Status
`addHeader()` – Hinzufügen eines Headers, bzw. eines Anteil eines Headers
`write()` – baut einen String auf, der alle Ausgaben, die an den Client/ Webbrowser geschickt werden sollen, enthält
`flush()` – schickt alle Daten an den Client/ Webbrowser und setzt die Eigenschaften zurück

Das Herzstück der Anwendung ist der FrontController, der die Ein- und Ausgaben instanziert und die geforderte konkrete Command-Klasse heraussucht und aufruft.

```
class FrontController {
  private $path;
  private $defaultCommand;

  public function __construct($path, $defaultCommand){
    $this->path = $path;
    $this->defaultCommand = $defaultCommand;
  }

  public function handleRequest(Request $request, Response $response){
    $command = $this->getCommand($request);
    $command->execute($request, $response);
    $response->flush();
  }

  public function getCommand(Request $request){
    if ($request->issetParameter("cmd")) {
      $cmdName = $request->getParameter("cmd");
      $command = $this->loadCommand($cmdName);
    }

    else{
      $command = $this->loadCommand($this->defaultCommand);
    }

    return $command;
  }

  public function loadCommand($cmdName){
    $class = $this->path . "\\" . $cmdName . "Command";
    $file = $this->path . "/" . $cmdName . "Command.php";

    if (!file_exists($file)) {
      return false;
    }

    $command = new $class();
    return $command;
  }
}
```

Quellcode: FrontController.php

`handleRequest()` – ruft die `execute()`-Methode der konkreten Command-Klasse auf, die vorher ermittelt wird und schickt alle über die `flush()`-Mthode der Response-Klasse alles an den Client/ Webbrowser
`getCommand()` – hier wird überprüft, ob der cmd-Parameter in der URI enthalten ist, ansonsten wird die Standard-Command-Klasse ausgewählt.
`loadCommand()` – es wird die in `getCommand()` ermittelte konkrete Command-Klasse geladen

5.3 Das Template-View-Pattern

Da es sehr umständlich ist, vollständigen HTML-Code über `$response->write()` zu schreiben und auch viel zu umständlich zu gestalten wäre, bietet sich das Template-View-Pattern an, um die Ausgaben in vollständige HTML-Seiten einzubetten. Dafür wird das TemplateView-Interface angelegt, welches die Methodenköpfe `assign()` und `render()` enthält.

Das Pattern hat den Zweck über die Methode `assign()` Platzhalter zu definieren, die HTML-Seite mit den Platzhaltern einzubinden und das Ganze über die `write()`-Methode an den Webbrowser zurückzuliefern.

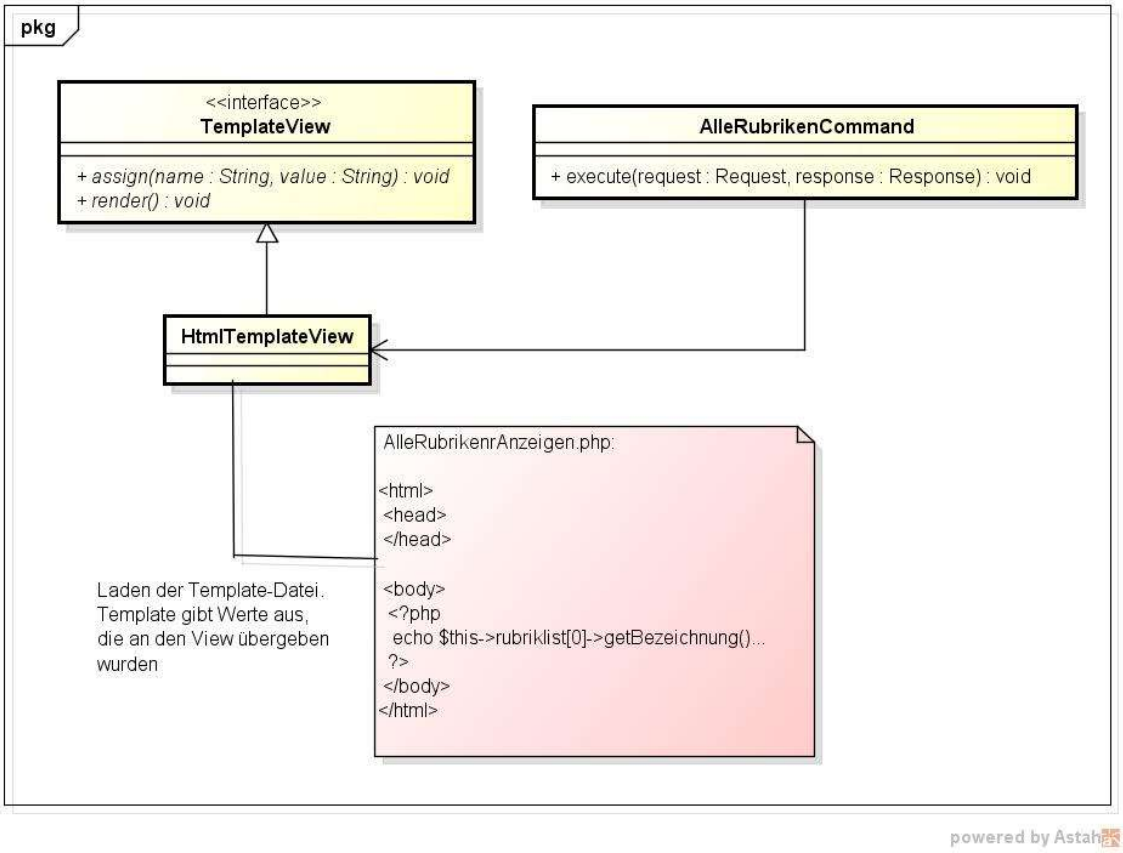


Abb. 9 Template-View-Klassendiagramm

```
interface TemplateView{
  public function assign($name, $value);
  public function render(Request $request, Response $response);
}
```

Quellcode: TemplateView.php

```
class HtmlTemplateView implements TemplateView{
  private $template;
  private $vars = array();

  public function __construct($template){
    $this->template = $template;
  }

  public function __get($property){
    if(isset($this->vars[$property])){
      return $this->vars[$property];
    }
    return null;
  }

  public function assign($name, $value){
    $this->vars[$name] = $value;
  }

  public function render(Request $request, Response $response){
    ob_start();
    $filename = "views/". $this->template .".php";
    require_once $filename;
    $data = ob_get_clean();
    $response->write($data);
  }
}
```

Quellcode: HtmlTemplateView.php

`assign()` – Der Name eines Platzhalter wird definiert und mit einem Wert verknüpft
`render()` – die entsprechende View-Datei wird geladen und alle Ausgabedaten mit der `write()`-Methoden der `HttpResponse`-Klasse an den Webbrowser/ Client übertragen.
`__get()` – ist eine sogenannte magische oder Interzeptor-Methode, die automatisch aufgerufen wird, wenn eine undefinierte Eigenschaft der Klasse angesprochen wird. Da die Platzhalter sich alle im Array `$vars` befinden, würde ein Aufruf `$this->platzhalter` diese Methode automatisch aufrufen und den entsprechenden Wert, der an dieser Stelle abgelegt wurde, übergeben.

Wichtig: `$this->meldung` ruft `__get('meldung')` auf und gibt den Wert von `$vars['meldung']` zurück, der mit `assign('meldung', "Fehler")` gespeichert wurde.

`ob_start()` – aktiviert die Ausgabenpufferung, das heißt, dass Ausgaben innerhalb eines Skriptes (z. B. echo, print) nicht mehr direkt an den Webbrowser übertragen werden, sondern erst mal zwischengepuffert.

`ob_get_clean()` – die Daten des Ausgabepuffers werden zurückgegeben und der Ausgabepuffer wird geleert

Die konkreten Command-Klassen wie zum Beispiel AlleRubrikenCommand laden eventuelle Request-Daten, holen aus der Datenbank notwendige Daten, erzeugen ein Objekt der Klasse HtmlTemplateView und geben Ergebnisse an das Objekt von HtmlTemplateView.

```
class AlleRubrikenCommand implements Command{
  public function execute(Request $request, Response $response){
    $rubrikDAO = new RubrikDAO();
    $rubrikList = $rubrikDAO->readAll();
    $view = 'alleRubrikenAnzeigen';

    $template = new HtmlTemplateView($view);

    $meldung = "";
    $link = "<a href='index.php?cmd=AnzeigeEintragen'>
                                          Anzeige aufgeben</a>";

    $template->assign('link', $link);
    $template->assign('rubrikList', $rubrikList);
    $template->assign('meldung', $meldung);
    $template->render( $request, $response);
  }
}
```

Quellcode: AlleRubrikenCommand.php

Der dazugehörige View:

```
<!DOCTYPE html>
<html lang="de">
<head>
  <meta charset="utf-8">
  <link href="css/bildschirm.css" rel="stylesheet" media="all">
</head>
<body>
  <header role="banner">
    <h1>Das Webbrett</h1>
  </header>

  <nav>
    <?= $this->link ?>
  </nav>

  <div class="wrapper">
    <?php
      foreach( $this->rubrikList as $rubrik){
        echo("<a href='index.php?cmd=AnzeigenEinerRubrik&nr=".
                    $rubrik->getNummer() ."' > " .
                    $rubrik->getBezeichnung() . "</a> <br>");
      }
  ?>
    <br>
    <?= $this->meldung ?>
  </div> <!-- end of wrapper -->

 . . .
  </body>
</html>
```

Quellcode: alleRubrikenAnzeigen.php

6. Loginsystem mit Sessions

6.1 Sessions

Um das Backend wie gefordert umsetzen zu können, ergibt sich ein Problem. Das von Webseiten verwendete Übertragungsprotokoll HTTP arbeitet in der aktuellen Version so, dass eine Anforderung vom Web-Client gesendet wird und der Webserver antwortet. Nach dem Senden verliert der Webserver alle Informationen über den Kommunikationspartner. Es ist ein sogenanntes zustandsloses Protokoll. Für Situationen, in denen sich zum Beispiel der Benutzer bei einem Content Management System angemeldet hat oder in einem Webshop Artikel in den Warenkorb gelegt hat, ist das ungünstig. Hier muss der Webserver die Informationen über den Kommunikationspartner für eine bestimmte Zeit behalten. Der Webserver muss den Web-Client wiedererkennen.

Damit der Webserver den Web-Client über mehrere Anforderungen hinweg wiedererkennen kann, muss eine Sitzung eingerichtet werden. Dafür werden Sessions verwendet. Nachdem eine Session eingerichtet wurde, können Werte in sogenannten Sessionvariablen gespeichert werden.

Sessions folgen dabei einem einfachen Ansatz. Wird eine Session gestartet, wird überprüft, ob eine Session-ID vom Client (Webbrowser) an den Server übertragen worden ist. Falls nicht wird eine neue Session-ID erzeugt und bei der Antwort an den Client (Webbrowser) mit übertragen.

Nachdem die Session gestartet wurde, füllt PHP die `$_SESSION`-Superglobale mit allen Sessiondaten. Am Ende des Skripts nimmt PHP automatisch den Inhalt der `$_SESSION`-Superglobalen, serialisiert ihn und verwendet Session-Speicherfunktion um ihn zu speichern.

Das Starten einer Session erfolgt in PHP mit session_start().Erst danach kann die $_SESSION-Superglobale benutzt werden.

```
<?php
   session_start();

   $_SESSION['user'] = $benutzername;
```

Oder überprüft werden, ob eine Variable vorhanden ist:

```
<?php
   session_start();

   if(isset($_SESSION['redakteur']){
      . . .
```

Ähnlich wird verfahren, wenn eine Session wieder zerstört werden soll:

```
. . .

   unset($_SESSION['user'] );
   session_destroy();
```

6.2 Login realisieren

Für die zu realisierenden Wunschkriterien (siehe Kap. 1.2 Aufgabenstellung Webbrett) muss ein Login für einen Redakteur realisiert werden. Zuerst wird ein entsprechendes Formular erstellt.


```
class LoginCommand implements Command{
   public function execute(Request $request, Response $response){
      //Anwendungsfall: Anmelden:
      $view = "login";
      $link = "<a href='index.php'>Startseite</a> <br><br>";
      $meldung = "Bitte geben Sie den Redakteursnamen und das Kennwort ein";

      $template = new HtmlTemplateView($view);
      $template->assign('link', $link);
      $template->assign('meldung', $meldung);
      $template->render( $request, $response);
   }
}
```

Quellcode: loginCommand.php

Der Controller `loginCommand.php` bindet den View `login.php` ein

```
 . .
<form name="login" action="index.php" method="post">
   <input type="hidden" name="cmd" value="checkLogin">
   <p>
      <label for="redakteur">Redakteur:</label>
      <input name="redakteur" type="text" size="30">
   </p>
   <p>
      <label for="kennwort">Kennwort:</label>
      <input name="kennwort" type="password" size="30">
   </p>
   <p>
      <input name="anmelden" value="anmelden" type="submit" size="30">
      <input name="abbrechen" value="abbrechen" type="reset" size="30">
   </p>
</form>
 . .
```

Quellcode: login.php

Das versteckte Eingabefeld `name="cmd"` garantiert uns, dass der richtige Controller `CheckLoginCommand` aufgerufen wird, um die Verarbeitung der Eingaben zu realisieren. Dort wird der eingetragene Redakteur und das Kennwort mit der Datenbank abgeglichen. Dafür gibt es innerhalb der `RedakteurDAO` die Methode `checkLogin`.

Nachdem die Anmeldedaten eingegeben worden sind, wird der Button `anmelden` angeklickt und das Skript auf den Controller `checkLoginCommand.php` auf dem Server angefordert (siehe `hidden`-Feld). Das Skript prüft die Eingaben und wenn der Redakteur-Anmeldename und das Kennwort korrekt sind, wird die Session gestartet und der Anmeldename des Redakteurs einer Session-Variablen zugewiesen. Als View-Template wird das `redakteurMenu.php` eingebunden.

Sind die Eingaben nicht korrekt, wird der View login.php erneut eingebunden.

```
<?php
namespace classes\commands;

use classes\request\Request;
use classes\response\Response;
use classes\template\HtmlTemplateView;
use classes\mapper\RedakteurDAO;

class CheckLoginCommand implements Command{

   public function execute(Request $request, Response $response){
      $link = "<a href='index.php'>Startseite</a> <br> <br>";

      if($request->issetParameter('anmelden') &&
         !empty($request->getParameter('redakteur')) &&
         !empty($request->getParameter('kennwort'))){

         $anmeldename = $request->getParameter('redakteur');
         $kennwort = $request->getParameter('kennwort');

         $redakteurDAO = new RedakteurDAO();
         $redakteur = $redakteurDAO->checkLogin($anmeldename, $kennwort);

         if($redakteur != null){
            session_start();
            $_SESSION['redakteur'] = $redakteur;
            $link = "<a href='index.php?cmd=Logout'>[logout]</a> <br> <br>";
            $meldung="Angemeldet:" . $_SESSION['redakteur']->getAnmeldename();
            $view = 'redakteurMenu';
         }
         else{
            $meldung = "Fehler! Falscher Benutzername oder Kennwort!<br>";
            $view = 'login';
         }
      }
      else{
         $meldung = "Fehler! Nicht alle Felder sind ausgefüllt.<br>";
         $view = 'login';
      }
      $template = new HtmlTemplateView($view);
      $template->assign('link', $link);
      $template->assign('meldung', $meldung);
      $template->render( $request, $response);
   }

}
?>
```

Quellcode: loginCommand.php

Wie oben schon beschrieben, wird in der Klasse `RedakteurDAO` in der Methode `checkLogin` nach einem gültigen Datensatz in der Datenbanktabelle `redakteur` nach einem `redakteur` mit dem eingegebenen Namen und dem eingegebenen Kennwort unter `kennwort` gesucht. Ansonsten wird `null` zurückgegeben.

```
<?php
namespace classes\mapper;
use classes\model\Redakteur;

class RedakteurDAO{
  private $dbConnect;

  public function __construct (){
    $this->dbConnect = MySQLDatabase::getInstance();
  }

 . . .
  public function checkLogin($anmeldename, $kennwort){
    $sql = "SELECT nr, anmeldename, rechte " .
           "FROM redakteur " .
           "WHERE anmeldename=? and kennwort=?";
    $redakteur = null;

    if(!$preStmt = $this->dbConnect->prepare($sql)){
      echo "Fehler bei SQL-Vorber. ". $this->dbConnect->error ."<br>";
    }
    else{
      if(!$preStmt->bind_param("ss", $anmeldename, $kennwort)){
        echo "Fehler beim Binding ". $this->dbConnect->error ."<br>";
      }
      else{
        if(!$preStmt->execute()){
          echo "Fehler beim Ausführen ". $this->dbConnect->error ."<br>";
        }
        else{
          if(!$preStmt->bind_result($nr, $anmelde, $rechte)){
             echo "Fehler beim Binding ". $this->dbConnect->error
."<br>";
          }
          else{
             if($preStmt->fetch()){
                 $redakteur = new redakteur($nr, $anmelde, "", $rechte);
             }
             $preStmt->free_result();
          }
        }
      }
      $preStmt->close();
    }
    return $redakteur;
  }
}
```

Quellcode-Ausschnitt: RedakteurDAO.php

Auf allen Seiten, die ein Benutzer nur eindeutig authentifiziert öffnen darf, muss die Session gestartet werden und eine Überprüfung der Session-Variable erfolgen. Das heißt in jeder Redaktionsseite muss die Funktion `session_start()` eingebunden werden. Außerdem sollte überprüft werden, ob die Session-Variable `$_SESSION['redakteur']` vorhanden und gefüllt ist.

```
<?php
session_start();
if(isset($_SESSION['redakteur']) && !empty($_SESSION['redakteur'])){
   $link = "<a href='index.php?cmd=logout' > [Logout] </a>
     <a href='index.php?cmd=anzeigeFreigebenUebersicht' > Anzeigen . . </a>
     <a href='index.php?cmd=loeschAnzeigen' > Anzeigen löschen </a> ";

   $meldung = "Sie sind angemeldet als " . $_SESSION['redakteur'];

   //Einbinden des Redaktionsmenus
   include("views/redakteurMenu.php");
}
else{
   session_destroy();
   $link = "<a href='index.php?'>Startseite</a> <br> <br>";
   $meldung = " Sie haben keine Berechtigung.<br>";
   include("views/login.php");
}
?>
```

Quellcode: redakteurMenuCommand.php

Sollte die Session-Variable `$_SESSION['redakteur']` nicht vorhanden oder vorhanden aber leer sein, wird die Session sofort wieder zerstört und die Anmeldeseite eingebunden.

Der eigentliche View `redakteurMenu.php` enthält ein Menu. Das könnte auch in eine HTML-Datei ausgelagert werden.

```
   <header role="banner">
     <h1>Das Redaktionsmenu</h1>
   </header>

<nav>
   <?= $this->link ?>
   <br> <br>
   <a href='index.php?cmd=Anzeigen'>Anzeigen bearbeiten</a> <br> <br>
   <a href='index.php?cmd=Inserenten'>Inserenten bearbeiten</a> <br> <br>
   <a href='index.php?cmd=Rubriken'>Rubriken bearbeiten</a> <br> <br>
   </nav>
   <div class="wrapper">
     <br>
     <?= $meldung ?>
     <br>
   </div> <!-- end of wrapper -->
```

Ausschnitt: redakteurMenu.php

Bei vollständig objektorientierter Realisierung bietet sich ansonsten das **Intercepting Filter** Design Pattern an (siehe Kap. 7).

7. Intercepting-Filter-Design-Pattern

Mit diesem Entwurfsmuster ist es möglich in eine bestehende Anwendung zusätzliche Logik einzubinden. So kann die zuvor beschriebene Authentifizierung über einen Filter realisiert werden. Ein Intercepting-Filter filtert Anfragen an Applikationen oder Antworten von Applikationen. Alle Filter werden in einer Filterkette zusammengefasst.

In Falle eines Präprocessing (Vorabverarbeitung) filtert eine Anfrage bevor sie von der eigentlichen Anwendung verarbeitet wird. Genauso wäre es möglich eine Antwort zu modifizieren, nach dem die Webanwendung diese erstellt hat, aber bevor diese an den Browser ausgeliefert wird. Hierbei handelt es sich um ein Post Processing (Nachbearbeitung).

Das Intercepting Filter Pattern arbeitet mit Filter-Klassen, die alle ein und dasselbe Interface implementieren. Es gibt das Interface Filter und mit einer Methode `execute()`. Dieses Interface wird z. B. mit `ConcreteFilter1` und `ConcreteFilter2` realisiert und damit auch die Methode `execute()` programmiert. Diese beiden konkreten Filter sind Beispiele für ein oder mehrere mögliche Implementierung von `Filter`.

Die Filter-Objekte der konkreten Filterklassen werden eine Filterketten-Klasse `FilterChain` hinzugefügt, die für das Aufrufen zuständig ist. Das heißt, innerhalb der Filterkette werden alle Objekte in einem Array abgelegt und beim Aufruf der Methode `processFilter()` wird nacheinander für alle Objekte, die im Array enthalten sind, die `execute()`-Methode aufgerufen. Das Hinzufügen eines konkreten Filterobjekts wird über die Methode `addFilters()` realisiert.

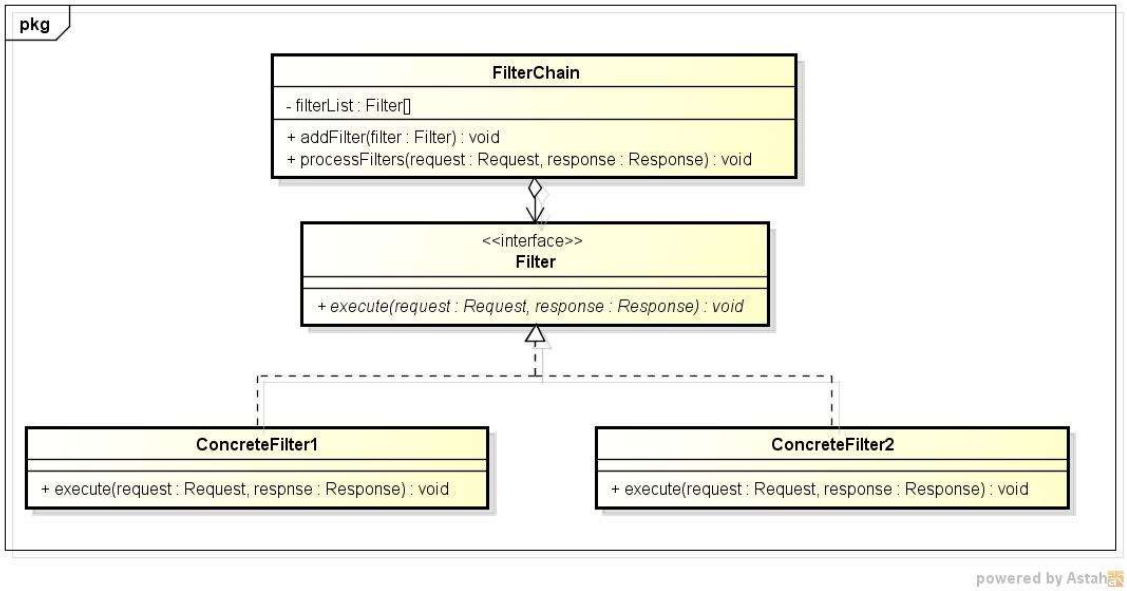


Abb. 10: Klassendiagramm des Intercepting-Filter ohne FilterManager

Wir definieren das Interface `Filter` und eine Klasse `SessionAuthFilter`.

```
<?php
namespace classes\filter;

use classes\request\Request;
use classes\response\Response;

interface Filter{
  public function execute(Request $request, Response $response);
}
?>
```

Quellcode: Filter.php

Auch der Filter-Methode `execute()` werden als Parameter die Objekte $request und $response übergeben. So kann auf übergebene Name-/ Werte-Paare zugegriffen werden oder Ausgaben realisiert werden.

In unserem Beispiel gibt es nur einen konkreten Filter, der für die Überprüfung, ob jemand angemeldet ist, übernimmt. Diese konkrete Klasse heißt hier `SessionAuthFilter`. Hier wird überprüft, ob es eine Sessionvariable `$_SESSION['redakteur']` vorhanden und nicht leer ist. Ausgaben sind hier eigentlich nicht gewollt, weil ein Filter seine Arbeit "unsichtbar" verrichten soll. Es sollte lediglich zum Login umgeleitet werden, wenn man auf eine Redakteurs-Site wechseln möchte, ohne angemeldet zu sein. Die Ausgaben unten dienen zur Veranschaulichung der Arbeitsweise des Filters.

```
<?php
namespace classes\filter;

use classes\request\Request;
use classes\response\Response;

class SessionAuthFilter implements Filter{
  public function __construct(){

  }

  public function execute(Request $request, Response $response){

    //Die Webseiten ohne Anmeldung heraushalten
    if($request->getParameter("cmd") == "RedakteurMenu"
      || $request->getParameter("cmd") == "Rubriken"
      || $request->getParameter("cmd") == "Anzeigen"
      || $request->getParameter("cmd") == "Inserenten"
      || $request->getParameter("cmd") == "Logout"){

      //ACHTUNG: Session wird immer gestartet!
      session_start();
      if(isset($_SESSION['redakteur']) && !empty($_SESSION['redakteur'])
        ){
        $response->write("$_SESSION['redakteur'] . " ist angemeldet<br>");
      }
      else{
        $request->setParameter("cmd", "Login");
      }
    }
  }
}
?>
```

Quellcode: SessionAuthFilter.php

Die Klasse FilterChain enthält ein Array für die Objekte der konkreten Filterklassen – in unserem Beispiel die Klasse `SessionAuthFilter`.

Wichtig: Es wird damit gerechnet, dass es mehrere Filter gibt, auch wenn es tatsächlich nur einen Filter in unserem Beispiel gibt.

```
<?php
namespace classes\filter;

use classes\request\Request;
use classes\response\Response;

class FilterChain{
   private $filters = array();

   public function addFilter(Filter $filter){
      $this->filters[] = $filter;
   }

   public function processFilters(Request $request, Response $response){
      foreach($this->filters as $filter){
         $filter->execute($request, $response);
      }
   }
}
?>
```

Quellcode: FilterChain.php

Die Methode `processFilters()` durchläuft das Array aus Filter-Objekten und ruft für jedes Objekt die `execute()`-Methode auf.

Der Frontcontroller agiert hier als Filtermanager, der die Filterketten enthält. Es kann Filter geben, die vor der eigentlichen Verarbeitung abgearbeitet werden (`preFilters`) und Filter, die nach der eigentlichen Verarbeitung ausgeführt werden (`postFilters`).

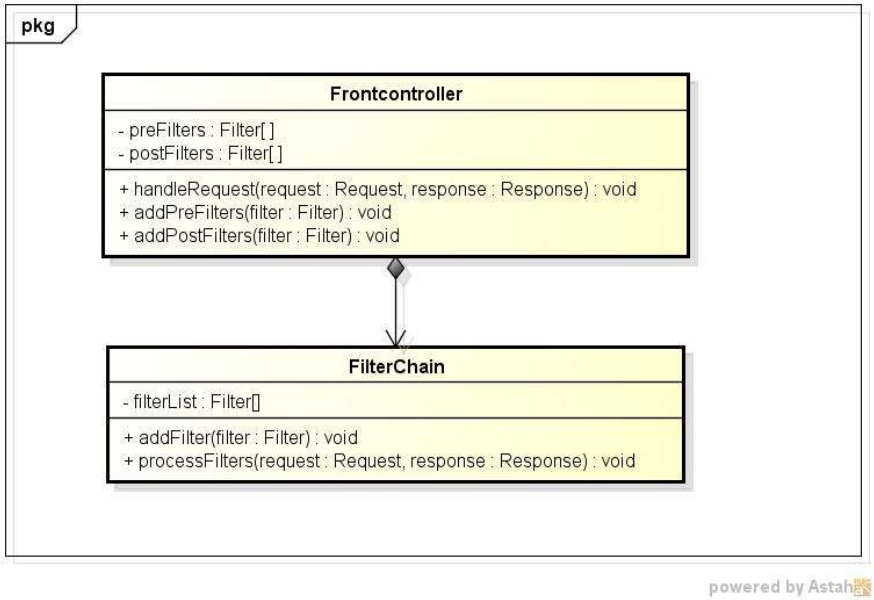


Abb. 11: Ausschnitt aus dem Klassenmodell für den FilterManager

Die eigentliche Verarbeitung wäre dann in der Methode `handleRequest()` durch den Aufruf der Methode `execute()` der jeweiligen Controller (`XYZCommand`) realisiert.

```
class FrontController {
  private $path;
  private $defaultCommand;
  private $preFilters;
  private $postFilters;

  public function __construct($path, $defaultCommand){
    $this->path = $path;
    $this->defaultCommand = $defaultCommand;
    $this->preFilters = new FilterChain();
    $this->postFilters = new FilterChain();
  }

  public function handleRequest(Request $request, Response $response){
    $this->preFilters->processFilters($request,  $response);

    $command = $this->getCommand($request);
    $command->execute($request, $response);

    $this->postFilters->processFilters($request,  $response);

    $response->flush();
  }
 . . .
```

Quellcode-Ausschnitt: FrontController.php

In der `index.php` wird das Filterobjekt für die Klasse `SessionAuthFilter` erzeugt und dem `FrontController`-Objekt `$controller` hinzugefügt. Das Aufrufen der `execute()`-Methode des Objekts erfolgt dann indirekt über die `handleRequest`-Methode.

```
  $request = new HttpRequest();
  $response = new HttpResponse();
  $controller = new FrontController('classes\commands', 'AlleRubriken');

  //neu für die Filter
  $authFilter = new SessionAuthFilter();
  $controller->addPreFilter($authFilter);

  $controller->handleRequest($request, $response);
```

Quellcode-Ausschnitt: index.php