international PHP2007 conference

PHP Design Patterns

Frank Kleine & Stephan Schmidt
1&1 Internet AG

Agenda

- OOP-Crashkurs
- Software-Design
- Erzeugungsmuster
- Strukturmuster
- Verhaltensmuster
- Enterprise-Patterns
  - Datenschicht
  - Business-Logik-Schicht
  - Präsentationsschicht

Frank Kleine

- Senior Web-Developer bei der 1&1
- PHP-Entwickler seit 2000
- Lead Developer Stubbles
- Lead Developer XJConf for PHP
- Co-Autor "Exploring PHP"

Stephan Schmidt

- Leiter Web-Development bei der 1&1
- PHP-Entwickler seit 1999
- Entwickler in PEAR, PECL, pat, Stubbles
- Autor für verschiedene Fachmagazine
- Speaker auf internationalen Konferenzen
- Autor von PHP Design Patterns, erschienen im O'Reilly Verlag
- Leider heute krank

OOP Crashkurs

- Klassen und Objekte
- Vererbung kontrollieren
- Sichtbarkeiten
- Interfaces
- Fehlerbehandlung mit Exceptions
- Interzeptoren in PHP
- Was kommt in PHP 5.3

Klassen – Baupläne für Objekte

- Sammlung aller Eigenschaften und Operationen eines Objektes

```
class Car {
  public $manufacturer;
  public $milage;
  public function __construct($manufacturer) {
    $this->manufacturer = $manufacturer;
  }
  public function moveForward($miles) {
    $this->milage = $this->milage + $miles;
  }
}

$bmw = new Car('BMW');
$bmw->moveForward(50);
```

Vererbung

- Kopieren von Eigenschaften und Methoden aus der abgeleiteten Klasse

```
class Convertible extends Car {
  public $roofOpen = false;
  public function openRoof() {
    $this->roofOpen = true;
  }
}

$bmw = new Convertible('BMW');
$bmw->openRoof();
$bmw->moveForward(50);
```

Sichtbarkeiten

- Erlaubt den Zugriff auf Eigenschaften und Methoden zu beschränken

```
class Car {
  private $manufacturer;
  protected $milage;
  public function getManufacturer() {
    return $this->manufacturer;
  }
  public function setManufacturer($manufacturer) {
    $this->manufacturer = $manufacturer;
  }
}
```

- private: nur diese Klasse
- protected: diese und abgeleitete Klassen

Finale Methoden und Klassen

- Verbietet des Überschreiben von Methoden

```
class Car {
  …
  public final function moveForward($miles) {
    $this->milage = $this->milage + $miles;
  }
}
```

- Verbietet das Ableiten von Klassen

```
final class Convertible extends Car {
}
```

Abstrakte Klassen

- Fordert das Implementieren von Methoden

```
abstract class Convertible extends Car {
  …
  public abstract function openRoof();
}
```

- Abstrakte Methoden haben keinen Rumpf
- Klassen mit einer oder mehr abstrakten Methoden sind auch abstrakt

Interfaces

- Summe aller öffentlichen Methoden, (Eigenschaften) und Konstanten

```
interface Vehicle {
  public function startEngine();
  public function moveForward($miles);
  public function stopEngine();
}

class Car implements Vehicle {…}
class Bike implements Vehicle {…}
```

- Erhöht den Grad der Kapselung
- Vererbung analog zu Klassen

Type-Hints

- Verlangen einen bestimmten Typ für Parameter

```
function move(Vehicle $v, $miles) {
  $v->startEngine();
  $v->moveForward($miles);
  $v->stopEngine();
}
$bmw = new Car('BMW');
move($bmw, 50);
```

- Typ kann Interface oder Klasse sein
- Vereinfacht Checks
- Seit 5.1 auch für Arrays

Exceptions

- Ermöglichen einfache Fehlerbehandlung

```
class Car {
  public function moveForward($miles) {
    if (!$this->engineStarted) {
      throw new IllegalStateException();
    }
  }
}
$bmw = new Car('BMW');
try {
  $bmw->moveForward(50);
} catch (Exception $e) {
  // Fehlerbehandlung
}
```

Exceptions

- Sind normale Objekte
- Müssen von Exception abgeleitet werden
- Unterschiedliche Fehlertypen durch unterschiedliche Exceptionklassen
- Fehlertypen werden durch Klassenhierarchie feiner granuliert

Exceptions

```
class Car {
  public function moveForward($miles) {
    if (!$this->engineStarted) {
      throw new IllegalStateException();
    }
    if (0 > $miles) {
      throw new IllegalArgumentException();
    }
  }
}
$bmw = new Car('BMW');
try {
  $bmw->moveForward(50);
} catch (IllegalStateException $e) {
  // Fehlerbehandlung
} catch (IllegalArgumentException $e) {
  // andere Fehlerbehandlung
}
```

Interzeptoren in PHP

- Greifen, bevor ein Fehler auftritt

```
class Car {
  public function __call($method, $args) {
    echo "Methode {$method} aufgerufen\n";
  }
}
$bmw = new Car('BMW');
$bmw->fly();
```

- **__call()** für Methoden
- **__get(), __set(), __isset()** für Eigenschaften
- **__autoload()** für Klassen

Sonstiges

- Automatisches Konvertieren in Strings

```
class Car {
  public function __toString() {
    return "Auto vom Typ {$this->manufacturer}\n";
  }
}
$bmw = new Car('BMW');
echo $bmw;
```

- Statische Methoden

```
class Car {
  public static function doSomething() { ... }
}
Car::doSomething();
```

Neu in PHP 5.3

- Namespaces

```
car.php:
namespace MyApp;
class Car {
  ...
}

bmw.php:
import MyApp::Car as MyCar;
$bmw = new MyCar('BMW');
echo $bmw;
```

- Verhindern Namenskonflikte mit anderen Applikationen

Neu in PHP 5.3

- **`__callStatic()`** analog zu **`__call()`**

- Late Static Binding:
  In statischen Methoden kann die Klasse zur Laufzeit referenziert werden **(`$this`** für statische Aufrufe)

Neu in PHP 5.3

```
class A {
  public static function who() {
    echo __CLASS__;
  }
  public static function test() {
    self::who();
  }
}
class B extends A {
  public static function who() {
    echo __CLASS__;
  }
}
PHP <  5.3: B::test(); // A
PHP >= 5.3: B::test(); // B
```

Software-Design

- Namenswahl
- Kapselung von Daten
- Regeln guten Software-Designs
- Dependency Injection

Namenswahl

- Sinnvolle Klassen- und Methodennamen wählen
- Code sollte wie ein Satz lesbar sein

```
if ($user->hasPermission(PERM::Write) &&
    $file->isWriteable()) {
  $user->write('Foo')->to($file);
}
```

- Fluent-Interfaces machen Code lesbarer
- Getter mit booleschen Rückgabewerten mit *is* oder *has* beginnen

Kapselung von Daten

- Eigenschaften sollten immer **`private`** oder **`protected`** sein
- Getter- und Setter-Methoden zum Modifizieren/Auslesen anbieten
- Alternativ **`__set()`** und **`__get()`** verwenden und virtuelle Properties einsetzen

Regeln des Software-Design

1. Wiederverwendbarkeit von Code ist besser als Duplizierung.
2. Kapseln Sie den Zugriff auf Daten immer innerhalb einer Klasse.
3. Kapseln Sie auch Algorithmen in einer Klasse.
4. Programmieren Sie immer gegen eine Schnittstelle.
5. Entwickeln Sie erweiterbare Schnittstellen.

Regeln des Software-Design (2)

6. Vermeiden Sie monolithische Strukturen. Ersetzen Sie komplexe if/else Blöcke durch einzelne Klassen.
7. Verwenden Sie Objektkomposition statt Vererbung.
8. Vermeiden Sie feste Abhängigkeiten zwischen Klassen, sondern koppeln diese lose miteinander.

Dependency Injection

"*Rufen Sie uns nicht an, wir rufen Sie an*"

- Objekte erstellen abhängige Objekte nicht selbst
- Abhängige Objekte werden von außen über Konstruktor oder Methoden injiziert

Dedizierte Session, Mittwoch 8:30 Uhr
Raum „Tempelhof“

Design Patterns

- Lösungsmuster für häufig auftretende Entwurfsaufgaben in der Software-Entwicklung
- Keine Code-Bibliothek
- Organisiert in Pattern-Katalogen (z.B. Gang-of-Four Buch)
- verschiedene Kategorien: Erzeugungsmuster, Strukturmuster, Verhaltensmuster, Enterprise-Patterns

Erzeugungsmuster

Erzeugungsmuster werden verwendet, um Objekte zu konstruieren.

- Singleton-Pattern
- Factory-Method-Pattern
- Abstract-Factory-Pattern

Singleton

- Problem: es darf nicht mehr als eine Instanz einer Klasse geben, z.B. um Speicher zu sparen
- Debugger, Logger
- zentraler Zugriffspunkt benötigt, aber globaler Namensraum soll nicht durch eine Variable mit der Instanz verschmutzt werden

Singleton

- stellt einen zentralen Zugriffspunkt bereit
- sichert ab, dass von einer Klasse nur eine Instanz existiert
- Implementierung:
  - statische Klasseneigenschaft speichert Instanz
  - statische Methode gibt Instanz zurück bzw. erstellt sie bei Nichtexistenz
  - Nutzung von __construct() und __clone von Außerhalb unterbinden

Singleton

```
class Debugger {
  protected static $instance;
  public static function getInstance() {
    if (null === self::$instance) {
      self::$instance = new self();
    }
    return self::$instance;
  }
  protected function __construct() { }
  private function __clone() { }
}

$debugger = Debugger::getInstance();
```

Singleton

- globaler Namensraum nicht verschmutzt
- genaue Kontrolle, wie auf Klasse zugegriffen wird
- immer nur eine Instanz der Klasse vorhanden
  - Variation: Instanz abhängig von Parameter der getInstance()-Methode
- Vorsicht: Einfachheit führt zu übermäßiger Nutzung in unangebrachten Fällen

Factory Method

- Problem: Objekte werden überall im Code mit **new** erzeugt.
- Änderungen sind nicht einfach möglich
  - Ändern der Konstruktor-Parameter
  - Austauschen der konkreten Implementierung
- Widerspricht Regel 4 (Programmierung gegen Schnittstellen)

Factory Method

- Definiert eine Schnittstelle zur Erzeugung von Objekten
- Verlagert die eigentliche Instanziierung in Unterklassen
- Läßt Unterklassen entscheiden, welche konkrete Implementierung verwendet wird

Factory Method

```
abstract class AbstractManufacturer {
  protected $name;

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

  public function sellVehicle() {
    $vehicle = $this->manufactureVehicle();
    // weitere Operationen möglich
    return $vehicle;
  }

  public abstract function manufactureVehicle();
}
```

Factory Method

```
class CarManufacturer extends AbstractManufacturer {
  public function manufactureVehicle() {
    return new Car($this->name);
  }
}

$bmwManufacturer = new CarManufacturer('BMW');
$bmw = $bmwManufacturer->sellVehicle();
```

Factory Method

- Ermöglicht, dass Framework-Code nur gegen Schnittstellen arbeitet
- Konkrete Implementierungen werden durch anwendungsspezifische Fabriken integriert.
- Verwandt mit der statischen Fabrikmethode

```
$con = MDB2::factory($dsn, $options);
```

Strukturmuster

- Composite-Pattern
- Adapter-Pattern
- Decorator-Pattern
- Proxy-Pattern
- Facade-Pattern

Composite

- Problem: mehrere Objekte gleicher Art müssen kombiniert werden
- Diese Struktur soll jedoch wie ein einziges Objekt behandelt werden: Ignorieren der Unterschiede zwischen Komposition von Objekten und Einzelobjekten
- Lösung: Composite
  - fügt mehrere Objekte gleicher Art in einer Baumstruktur zusammen
  - Clients können die gesamte Struktur wie ein einziges Objekt behandeln

Composite

- eine Klasse hält alle Instanzen des gleichen Typs in einer Liste
- Klasse kann neue Instanzen dieses Typs zur Liste hinzufügen
- Methodenaufrufe werden an alle Instanzen in der Liste weiter geleitet
- Klasse implementiert das gleiche Interface wie die Instanzen
- beliebig tief in einer Baumstruktur schachtelbar

Composite

```
class DebuggerLog implements Debugger { ... }
class DebuggerMail implements Debugger { ... }
class DebuggerComposite implements Debugger {
  protected $debuggers = array();
  public function addDebugger(Debugger $debugger) {
    $this->debuggers[] = $debugger;
  }
  public function debug($message) {
    foreach ($this-debugger as $debugger) {
      $debugger->debug($message);
    }
  }
}
$debuggerComposite = new DebuggerComposite();
$debuggerComposite->addDebugger(new DebuggerLog());
$debuggerComposite->addDebugger(new DebuggerMail());
```

Composite

- leicht erweiterbar: neue Klassen müssen lediglich das Interface implementieren und sind sofort hinzufügbar
- beliebig tief schachtelbar: einer Composite-Instanz können andere Composite-Instanzen hinzugefügt werden
- Vorsicht: Entwurf nicht zu allgemein werden lassen, da sonst Typen-Checks zur Laufzeit notwendig werden

Decorator

- Problem: Bildung von Unterklassen ist nicht (einfach) zur Laufzeit möglich
- Funktionalität kann nur zur Kompilierungszeit hinzugefügt werden
- Vererbung ist sehr statisch
- Selten genutzte Funktionalität muss trotzdem in jedem Objekt zur Verfügung stehen

Decorator

- Erweitert ein Objekt zur Laufzeit um neue Funktionalitäten
- Erweitert oder verändert die vorhandenen Methoden
- Flexible Alternative zur Bildung von Unterklassen

Decorator

```
interface String {
  public function getValue();
  public function setValue();
}

class StringImpl implements String {
  protected $value;

  public function _construct($value) {
    $this->value = $value;
  }
  // Setter und Setter
}

$s = new StringImpl('Hello!');
echo $s->getValue();
```

Decorator

```
class StringDecorator implements String {
  protected $target;

  public function _construct(String $target) {
    $this->target = $target;
  }

  public function getValue() {
    return $this->target->getValue();
  }

 public function setValue($value) {
    return $this->target->setValue($value);
  }
}
```

Decorator

```
class ReverseDecorator extends StringDecorator {
  public function reverse() {
    $v = $this->target->getValue();
    $this->target->setValue(strrev($v));
  }
}

$s = new StringImpl();
$decorated = new ReverseDecorator($s);
$decorated->reverse();
echo $decorated->getValue();

if ($decorated instanceof String) {
  echo "verhält sich auch wie ein String";
}
```

Decorator

- Schafft größere Flexibilität als Vererbung
- Hält einzelne Klassen schlank, da nur wichtige Logik implementiert wird

- Verlangt, dass es zu Ihren Klassen auch Interfaces gibt
- Viele Decorator machen Code unübersichtlich
- Jeder Decorator macht den Code langsamer

Proxy

- Problem: eine Klasse soll nicht in der Form verfügbar gemacht werden wie ist:
  - Einschränkung der Zugriffe
  - zu teuer, um sie immer zu instantiieren
  - konkretes Objekt liegt auf einem anderen Server

Proxy

- kontrolliert den Zugriff auf ein Objekt
- ist ein „Stellvertreter“ oder „Ersatz“ anstelle des eigentlichen Objekts
- verschiedene Ausprägungen:
  - Schutz-Proxy kontrolliert, ob Zugriffe auf das echte Objekt gestattet sind
  - Virtueller Proxy ist ein Ersatz für ein teures Objekt und erzeugt dieses erst wenn es nicht mehr ohne geht
  - Remote Proxy: das eigentliche Objekt befindet sich auf einem anderen Server

Proxy

```
class ImageReal implements Image {
  public function __construct($imagePath) {
    // teuer: Bild in Speicher laden
  }
  public function draw() { ... }
}
class ImageProxy implements Image {
  protected $imgPath; protected $realImage;
  public function __construct($imagePath) {
    $this->imgPath = $imagePath;
  }
  public function draw() {
    if (null === $this->realImage) {
      $this->realImage = new ImageReal($this->imgPath);
    }
    $this->realImage->draw();
}
```

Decorator vs. Proxy vs. Adapter

Nicht verwechseln:

- Proxy kontrolliert Zugriff
- Decorator fügt zusätzliches Verhalten hinzu
- Adapter ändert das Interface der gewrappten Klasse, Decorator und Proxy implementieren das Interface der gewrappten Klasse
- Proxy instantiiert das eigentliche Objekt erst bei Bedarf
- Decorator und Adapter wrappen ein bereits existierendes Objekt

Facade

- Problem: Applikation besteht aus komplexen Objektkompositionen
- Subsysteme sollten nutzbar sein, ohne dass Wissen über Interna der Anwendung erlernt werden muss

Facade

- Bietet eine vereinheitlichte Schnittstelle für einen Satz von Schnittstellen
- Bietet eine hochstufigere Schnittstelle, die die Verwendung des Basissystems vereinfacht

Facade

```
// ohne Facade
$tagParser = new DefinitionParser();
$defs = $tagParser->parse($definitionFile);

$conf = new XmlParser();
try {
  $conf->setTagDefinitions($defs);
  $conf->parse($configFile);
catch (Exception $e) {
}

// Mit Facade
$facade = new XJConfFacade();
$facade->addDefinitions($definitionFile);
$facade->parse($configFile);
```

Facade

- Vereinfacht bestehende Schnittstellen
- Mehrere Fassaden auf ein Subsystem möglich und oft auch sinnvoll
- Fördert das Prinzip der Verschwiegenheit

"*Halten Sie die Anzahl der Klassen, mit denen Ihre Objekte interagieren, möglichst gering.*"

Verhaltensmuster

- Subject/Observer-Pattern
- Template-Method-Pattern
- Command-Pattern
- Visitor-Pattern
- Iterator-Pattern

Template Method

- Problem: feste Abfolge von Schritten, aber unterschiedliche Implementierungen einzelner Schritte möglich

```
class Recipe {
  public function prepare() {
    $this->collectIngredients();
    $this->prepareIngedients();
    $this->cook();
    $this->serve();
  }
  ...
}
```

Template Method

- definiert die Schritte eines Algorithmus in einer Methode
- Implementierung der einzelnen Schritte bleibt Unterklassen vorbehalten

Template Method

```
abstract class Recipe {
  public final function prepare() {
    $this->collectIngredients();
    $this->prepareIngedients();
    $this->cook();
    $this->serve();
  }
  protected abstract function collectIngredients();
  protected abstract function prepareIngredients();
  protected abstract function cook();
  protected abstract function serve();
}
class CheeseBurgerRecipe extends Recipe {
  protected function collectIngredients() {...}
  ...
}
```

Template Method

- erhöht Wiederverwendbarkeit
- gemeinsames Verhalten muss nur einmal implementiert werden: Änderungen am Algorithmus nur an einer Stelle notwendig
- neue Unterklassen müssen nur die konkreten Schritte implementieren
- Achtung: Anzahl abstrakter Schritte nicht zu groß werden lassen, evtl. einige optional machen und Basisimplementation anbieten

Command

- Problem: Sie möchten verschiedene Aktion nacheinander ausführen, die beliebig miteinander kombiniert werden können.
- Beispiel: Waschprogramm einer Waschstrasse besteht aus einzelnen "Waschbefehlen", die miteinander kombiniert werden können.
- Erster Lösungsansatz - eine Klasse pro Waschprogramm? Besser nicht.

Command

- Das Command Pattern kapselt einen Auftrag als Objekt.
- Aufträge (Objekte) sind parametrisierbar
- Aufträge können in einer Queue nacheinander abgearbeitet werden
- Aufträge können rückgängig gemacht werden.

Command

```
interface CarWashCommand {
  public function execute(Car $car);
}

class CarSimpleWashCommand implements CarWashCommand
{
  public function execute(Car $car) {
    echo "Das Auto wird gewaschen";
  }
}

class CarDryingCommand implements CarWashCommand {
  public function execute(Car $car) {
    echo "Das Auto wird getrocknet";
  }
}
```

Command

```
class CarWash {
  protected $programmes = array();
  public function addProgramme($name,
                               array $commands) {
    $this->programmes[$name] = $commands;
  }
  public function wash($prog, Car $car) {
    foreach ($this->programmes[$prog] as $command) {
      $command->execute($car)
    }
  }
}
```

Command

```
$wash = new CarWash();
$wash->addProgramme('standard',
          array(
            new CarSimpleWashCommand(),
            new CarDryingCommand()
          ));
$wash->addProgramme('komfort',
          array(
            new CarSimpleWashCommand(),
            new CarEngineWashCommand(),
            new CarDryingCommand(),
            new CarWaxingCommand()
          ));

$wash->wash('standard', $bmw);
```

Command

- Kapselt Aufträge als Objekte
- Erleichtert das Einfügen neuer Operationen
- Sehr beliebt für Installer
- Sehr gut mit Composite-Pattern kombinierbar
- Durch Hinzufügen einer undo() Methode können Aufträge auch wieder rückgängog gemacht werden

Iterator

- Problem: Sie möchten verschiedene Listen-Typen durchlaufen, ohne für jeden Typ anderen Schleifen-Code zu verwenden

```php
$arr = array(...);
foreach ($arr as $val) {
  ...
}

$d = dir('path/to/dir');
while (false !== ($entry = $d->read())) {
  ...
}
```

Iterator

- Ermöglicht , auf die Elemente eines zusammengesetzten Objekts sequentiell zuzugreifen, ohne die zu Grunde liegende Struktur zu offenbaren
- PHP bringt in der SPL bereits Interface und konkrete Implementierungen mit

```
interface Iterator {
  public function current();
  public function key();
  public function next();
  public function rewind();
  public function valid();
}
```

Iterator

```
$arr = array(...);
$it  = new ArrayIterator($arr);
foreach ($it as $value) {
  ...
}

$it  = new DirectoryIterator('path/to/dir');
foreach ($it as $value) {
  ...
}
```

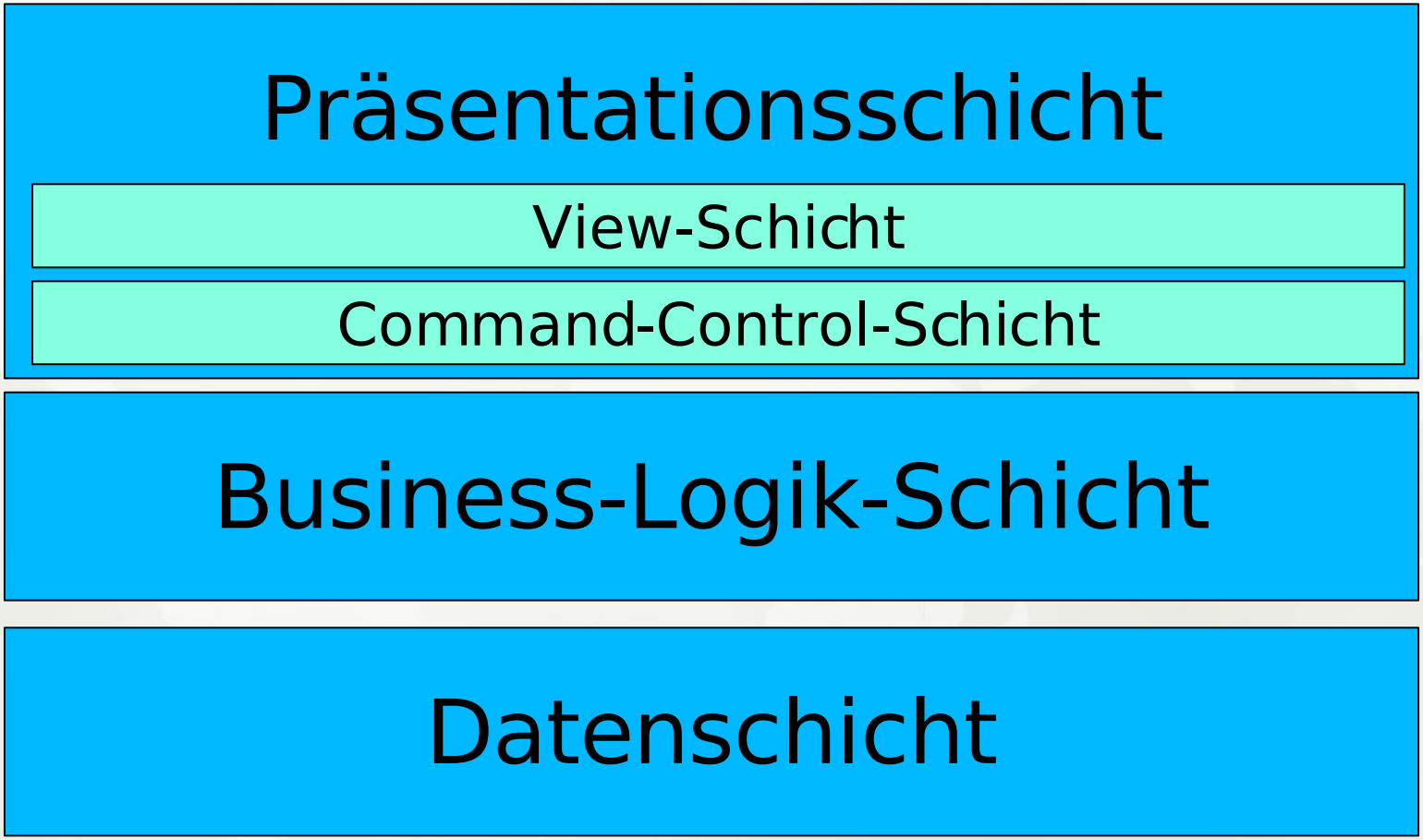
Iterator

- Sehr komplexes Pattern, in PHP 5 sehr einfach zu nutzen
- Gute Integration durch **`foreach`**
- SPL stellt auch **`IteratorAggregate`** zur Verfügung. Dies ermöglicht externe Iteration (mehrfache, parallele Iteration über das selbe Objekt möglich)
- **`FilterIterator`** überspringt manche Elemente (z.B. nur über XML-Dateien iterieren)

Enterprise-Patterns

- Schichten einer Applikation
- Model-View-Controller
- Patterns der Datenschicht
  - Row-Data-Gateway-Pattern
  - Active-Record-Pattern
  - Data-Mapper-Pattern
- Registry-Pattern
- Patterns der Business-Logik-Schicht

Schichten einer Applikation



Model-View-Controller

- Buzzword #1 der letzten Jahre
- Set von Design Patterns, das eingesetzt wird, um die Schichten voneinander zu trennen
- *View* stellt die Daten des Models dar
- *Model* speichert die Daten der Applikation
- *Controller* nimmt die Aktion des Benutzers an und fordert Model oder View auf, den Zustand zu ändern

Datenschicht

- Kümmert sich um die Verwaltung der Daten
- Persistierung der Daten in einem Datenspeicher
- Selektieren der Daten aus diesem Datenspeicher
- Häufig durch RDBMS abgebildet, aber nicht darauf beschränkt

Row-Data-Gateway

- Problem: Sie möchten Daten der Autos in einer Datenbank persistieren, aber nicht darauf verzichten, diese als Objekte zu verwenden
- Jedes Auto soll in einer Zeile einer Tabelle gespeichert werden
- Mögliche Operationen sollen INSERT, UPDATE, SELECT und DELETE sein

Row-Data-Gateway

- Row-Data-Gateway dient als Repräsentation einer Zeile in einer Datenbanktabelle
- Eine Instanz pro Zeile, über die die Spalten der Zeile verändert werden
- Pro Tabelle wird eine Klasse benötigt
- Je Eigenschaft eine Getter- und Setter-Methode
- Zusätzliche Methoden zum Speichern der Daten in der Klasse

Row-Data-Gateway

- SQL Funktionalität wird meist durch abstrakte Basis-Klasse bereit gestellt
- Die Klassen sind reine Datencontainer, bieten keine zusätzliche Funktionalität
- Zusätzliche "Finder" Klasse zum Selektieren der Daten
- Propel ist bekannte Implementierung in PHP, generiert PHP Klassen für DB-Tabellen

Row-Data-Gateway

```
$bmw = new Car('BMW');
$bmw->setColor('Blau');
$bmw->save();

$bmw = CarPeer::retrieveByPk(1);
echo $bmw->getColor();

$c = new Criteria();
$c->add(CarPeer::MANUFACTURER, 'BMW');
$bmws = CarPeer::doSelect($c);

foreach ($bmws as $bmw) {
  echo $bmw->getColor();
}
```

Row-Data-Gateway

- Kein SQL mehr in der Business-Logik
- Unabhängig von der gewählten Datenbank
- Führt zu Performance-Einbußen und Verzicht auf DB-spezifische Features

Active-Record

- Ähnelt Row-Data-Gateway
- Fügt den Klassen jedoch Domänenlogik hinzu
- Auch in Propel realisiert, da Unterklassen erstellt werden, die vom Entwickler modifiziert werden können

Data Mapper

- Row-Data-Gateway und Active Record nicht der Weisheit letzter Schluss
- Datenklassen sollten Modell wiederspiegeln, nicht wie sie selbst gespeichert werden
- Daten sollen unterschiedlich persistiert werden können: Datenbank, Datei, Application Server, ...

Data Mapper

- Lösung: gesonderte Schicht, welche die Applikations-Objekte auf den Datenspeicher mappt
- Objekte und Mapper sind unabhängig voneinander
- Im Bestfall: Mapper kann jede Art von Objekt persistieren (soweit sinnvoll)

Data Mapper

- XML-Beschreibungen:
  - Mapping wird in XML-Dateien definiert
  - Mapper liest XML-Datei und behandelt Objekt entsprechend dieser Konfiguration
- Besser: Annotations
  - Mapping steht als Annotation an/in der Klasse, ist aber kein Programmcode
  - weniger Definitionsoverhead als mit XML
  - Annotations an Methoden/Eigenschaften werden in Klassenhierarchie mit vererbt

Data Mapper

```
/**
 * @Entity
 */
class Person {
  protected $name, $age;
  public function __construct($name, $age) { ... }
  /**
   * @DBColumn(name='name')
   */
  public function getName() {
    return $this->name;
  }
  public function getAge() {
    return $this->age;
  }
}
$entityManager->save(new Person('mikey', 28));
```

Registry

- Konfiguration soll nur einmal eingelesen werden
- Konfigurationsobjekt wird jedoch in verschiedenen Schichten benötigt
- Durchreichen durch die Schichten führt zu unnötig langen Methodensignaturen, Objekte werden oftmals einfach nur durchgereicht
- Anwendung des Singleton-Pattern nicht sinnvoll, da mehrere verschiedene Instanzen benötigt
- globale Variablen sind böse :-)

Registry

- Registry speichert Objekte oder Informationen zur gemeinsamen Nutzung durch andere Objekte
- bietet zentralen Zugriffspunkt auf Objekte und Informationen
- Registry ist selbst ein Singleton (Alternative: komplett statische Implementierung)

Registry

```
class Registry {
  protected $data = array();
  // Singleton-Implementierung ausgelassen
  public function get($key) {
    if (isset($this->data[$key])) {
      return $this->data[$key];
    }
    return null;
  }
  public function set($key, $value) {
    $this->data[$key] = $value;
  }
}
Registry::getInstance()->set('foo' => new Bar('baz'));
Registry::getInstance()->get('foo');
```

Registry

```
class Registry {
  protected static $data = array;
  public static function get($key) {
    if (isset(self::$data[$key])) {
      return self::$data[$key];
    }
    return null;
  }
  public function set($key, $value) {
    self::$data[$key] = $value;
  }
}
Registry::set('foo' => new Bar('baz'));
Registry::get('foo');
```

Business-Logik-Schicht

- Enthält die eigentliche Anwendungslogik
- Kaum spezielle Patterns
  - Domain-Model-Pattern
  - Value-Object-Pattern
- Alle klassischen Patterns werden in dieser Schicht eingesetzt

Enterprise-Patterns (2)

- Patterns der Command-Control-Schicht
  - Front-Controller-Pattern
  - Intercepting-Filter-Pattern
- Event-Dispatcher-Pattern
- Patterns der View-Schicht
  - Template-View-Pattern
  - View-Helper-Pattern

Command-Control-Schicht

- Kümmert sich im Interaktion mit dem Nutzer
- Nimmt die HTTP-Anfragen in Web-Anwendungen entgegen
- Analysiert Anfragen und steuert Business-Logik-Schicht an
- Geschäftslogik wird dadurch unabhängig vom Web-Umfeld

Front Controller

- Entkopplung der Applikation vom Webumfeld, um Teile auch in einem anderem Umfeld wiederverwenden zu können
- Hinzufügen neuer Seiten ermöglichen, ohne zentrale Logik zu duplizieren
- einfachere Sicherheitsüberprüfungen durch zentralen Einstiegspunkt ermöglichen

Front Controller

- nimmt alle Anfragen an die Webanwendung entgegen
- führt alle zentralen Arbeiten durch
- Weiterleitung der Anfrage an zuständige Objekte, welche die Antwort erzeugen
- erfordert gesonderte Klassen zur Kapselung der Daten von Anfrage (Request) und Antwort (Response)

Front Controller

- Viele verschiedene Implementierungen verfügbar:
  - Zend Framework
  - Symfony
  - WACT
  - Stubbles
  - ...

Intercepting Filter

- Problem: Sie möchten Logik hinzufügen, die bei jedem Request ausgeführt werden soll.
- Authentifizierung, Logging, URL-Rewriting, etc.
- Front-Controller soll jedoch nicht dafür modifiziert werden müssen

Intercepting Filter

- Intercepting Filter filtert Anfragen an eine Applikation.
- Kann als Prä- oder Post-Prozessor eingesetzt werden
- Kann damit sowohl Anfrage als auch Antwort filtern
- Kann in eine Filterkette integriert werden

Intercepting Filter

```
interface Filter {
  public function execute(Request $req,
                          Response $resp);
}

class AuthFilter implements Filter {
  public function execute(Request $req,
                          Response $resp) {
    $authData = $req->getHttpAuth();
    if (!$this->authDataMatches($authData)) {
      $request->cancel();
      $response->sendError();
    }
  }
}
```

Intercepting Filter

- Intercepting Filter ist eine Spezialisierung des Command-Patterns
- Filterkette kann einfach als Composite oder Decorator implementiert werden
- Controller sollte zwei Filter-Ketten haben, eine vor und eine nach der Verarbeitung
- Fügt dem Controller eine einfache Plugin-Schnittstelle hinzu
- Filter sind unabhängig voneinander, kann manchmal Probleme bereiten

Event Dispatcher

- Subject/Observer für konkreten Anwendungsfall nicht geeignet: Subject besitzt keinen Zustand
- Zustandsänderung oder weitere Verarbeitung soll abhängig von Observer sein
- Subject sollte nicht alle Observer kennen und selbst benachrichtigen müssen

Event Dispatcher

- Vermittler für Nachrichten von einem Absender zu beliebig vielen Adressaten
- Objekte registrieren sich beim Dispatcher für Nachrichten eines bestimmten Typs
- Dispatcher informiert Objekte automatisch über neue Nachrichten
- Absender kann abfragen, ob das Ereignis abgebrochen wurde
- Nachrichten können Kontextinformationen beinhalten (z.B. das Objekt, welches das Ereignis ausgelöst hat)

Event Dispatcher

```
class Auth {
  public function login($user, $password) {
    // authenticate user
    $this->userName = $userName;
    $this->isValid  = true;
    $event = EventDispatcher::getInstance()
             ->trigger('onLogin', $this);
    if ($event->isCancelled() == true) {
      $this->userName = null;
      $this->isValid  = false;
    }
}
```

Event Dispatcher

```
class UserLoginLogging implements EventListener {
  ..
  public function handleEvent(Event $event) {
    $this->log($event->getTime(), $event->getName(),
$event->getContext()->getUserName());
  }
}
class BlackList implements EventListener {
  ...
  public function handleEvent(Event $event) {
    $auth = $event->getContext();
    if ($this->isBlackListed($auth->getUserName()) ==
true) {
      $event->cancel();
      $this->log($event->getTime(), $auth-
>getUserName());
    }
  }
}
```

Event Dispatcher

```
$eventDispatcher = EventDispatcher::getInstance();
$eventDispatcher->register(new BlackList(), 'onLogin');
$eventDispatcher->register(new UserLoginLogging(),
'onLogin');
$auth = new Auth();
$auth->login('mikey');
// user mikey eingeloggt
$auth->login('schst');
// user schst geblockt

// im Logfile:
2007-01-30 11:35:45|onLogin|mikey
2007-01-30 12:11:57|cancelledLogin|schst
```

Event Dispatcher

- extrem lose Kopplung zwischen Klassen
- beliebig viele Nachrichten und damit Kopplungen möglich
- Aber: Anwendung wird schwerer durchschaubar, da nicht unbedingt klar ist welches Ereignis zu welchen Folgen führt bzw. wann ein Ereignis wo ausgelöst wird (Dokumentation!)
- Typsicherheit für Nachrichten-Kontext nicht möglich

View-Schicht

- Kümmert sich um die Darstellung der Daten aus der Datenschicht
- Entkoppelt die Commands aus der Command-Control-Schicht von der Darstellung der Daten
- Ermöglicht, dass Designer die Darstellung verändert, ohne mit Business-Logik in Berührung zu kommen

Template-View

- Problem: HTML Code soll nicht mit **echo** oder Methodenaufrufen auf dem Response-Objekt erzeugt werden.
- HTML-Code soll komplett von Business-Logik getrennt sein und von unabhängigen Personen gepflegt werden können

Template-View

- Ein Template-View trennt den HTML-Code für die Darstellung von der Logik. Dazu werden Platzhalter für die Daten im HTML-Code eingefügt.
- Platzhalter werden vom Template-Objekt durch die tatsächlichen Daten ersetzt

Template-View

- PHP selbst wurde als Template-View implementiert.
- Mittlerweile wird PHP für weitaus mehr genutzt und es gibt verschiedene Template-View-Implementierungen
  - patTemplate
  - Smarty
  - Zend_View
- Kein Code-Beispiel, um religiöse Diskussionen zu vermeiden

View-Helper

- Problem: Daten, die an den View übergeben werden, müssen vom View nochmal verändert werden können
  - Datumswerte müssen in das regional übliche Format konvertiert werden
  - Abschneiden langer Texte
- Diese Konvertierungen sollen nicht die Business-Logik "verschmutzen"

View-Helper

- View-Helper bieten Funktionalität, die in den Template-Views benötigt wird.
- Sie entkoppeln die Business-Logik von der Darstellung.

View-Helper

- Ursprünglich waren alle PHP-Funktionen als View-Helper gedacht:

```
<p>
  Heute ist der
  <?php echo date('d.m.Y', $date)?>
  .
</p>
```

- Jede Template-Engine bietet mittlerweile eine Implementierung für View-Helper

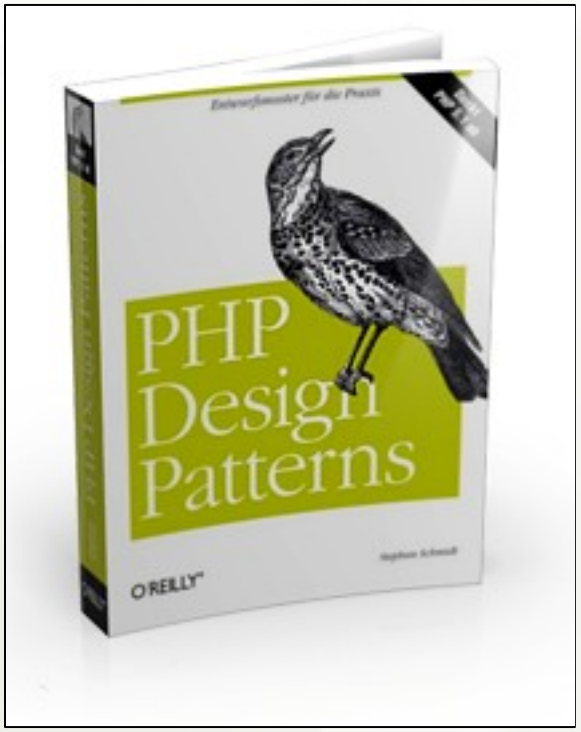
Ende

Vielen Dank für Ihre Aufmerksamkeit.

Noch Fragen?

php@frankkleine.de
me@schst.net

Zu schnell?



PHP Design Patterns
O'Reilly Verlag
http://www.phpdesignpatterns.de

Slides des Workshops
http://www.stubbles.net