

Clean Code Kochbuch für PHP-Entwickler

51 Regeln, Richtlinien und
Prinzipien für sauberen PHP Code

Jan Brinkmann
PHPGeek.org

Inhalt

1 Intro	4
1.1 Sauberer Code ist die Kombination aus vielen Details	4
1.2 Erfolg ist die Summe richtiger Entscheidungen?	4
2 Allgemeiner Stil	6
2.1 Schreibe in camelCase	6
2.2 Werte in Konstanten, die gesucht werden können	6
2.3 Instanzvariablen, use und Konstanten sortieren	7
2.4 Englische Bezeichnungen	7
2.5 Kein global verwenden	8
2.6 Verwende keine zu Bezeichnungen	8
2.7 Verwende ausreichende Abstraktion	9
2.8 Einheitliche Formatierung	9
3 Kontrollfluss	10
3.1 Keine kreativen Namen für Schleifenzähler	10
3.2 Logik der switch-Statement Fälle auslagern	10
3.3 Bedingungen nicht unnötig verschachteln (Early Exit)	10
3.4 Exceptions sind für Fehlerbehandlung besser	11
3.5 Den Code aus try/catch Blöcken auslagern	12
3.6 Setze auf positive if-Bedingungen	12
4 Variablen	13
4.1 Namen von Variablen werden immer klein geschrieben	13
4.2 Namen beschreiben den Inhalt der Variable	13
4.3 Aussprechbare Namen verwenden	13
5 Klassen	14
5.1 Klassennamen sind Nomen	14
5.2 Keine generischen Klassennamen	14
5.3 Keine Verben in Klassennamen	14
5.4 Klassennamen immer groß	14
5.5 Instanzvariablen immer deklarieren	15
5.6 Instanzvariablen immer über den Konstruktor	15
5.7 Instanzvariablen im richtigen Kontext definieren	15

5.8 Namen nicht unnötig aufblähen	16
5.9 Verzichte auf das Singleton Pattern	17
5.10 Kapselung aller Daten	17
5.11 Schreibe sich aufrufende Methoden untereinander	18

6 Kommentare

6.1 Nutze Kommentare nicht, um Code zu erklären.	19
6.2 Keine Kommentare um Konstante Werte zu erklären	19
6.3 PHPDoc: Parameter Rückgabewerte dokumentieren	20
6.4 Einen erklärenden Satz zu jeder Methode	20
6.5 Schreib noch offene Aufgaben in TODO Kommentare	20
6.6 Verzichte auf redundante Kommentare	21
6.7 Auskommentierter Code	21
6.8 Zu lange Kommentare	21

7 Methoden

7.1 Jede Methode hat nur eine Aufgabe	21
7.2 Mehr als 2 Parameter verwenden	22
7.3 Parameter unterscheidbar benennen	22
7.4 Keine flag-Argumente verwenden	22
7.5 Keine Methoden mit mehreren Bereichen	23
7.6 Verb im Namen einer Methode	24
7.7 Konsistente Verben im gesamten Projekt	24
7.8 Keine Seiteneffekte produzieren	25

8 Prinzipien

8.1 DRY - Don't Repeat Yourself	27
8.2 Gesetz von Demeter	27
8.3 SOLID	28

9 Fazit und Anwendung der Prinzipien

1 Intro

Entwickler sind intelligent. Du bist in der Lage komplexe Zusammenhänge im Kopf zu jonglieren und zu verstehen.

Selbst wenn Du gerade am Anfang stehst und Dir das Schwierigkeiten macht: mit etwas Übung wird das besser.

Oft ist das notwendig, um Strukturen zu entwerfen. Ein Blick aus der Vogelperspektive. Die benötigten Klassen und Methoden wie ein Puzzle zusammensetzen.

Was aber nicht sein muss: Dir oder anderen Steine in den Weg legen. Sauberer Code ist gut lesbar. Du musst nicht ständig überlegen was genau passiert, was in einer Variable steht.

Wird Deine Arbeit zu unübersichtlich, fängst Du sogar oft mehrfach oben in der Methode an um zu verstehen, was genau eigentlich gerade passiert.

Die Aufgabe dieser Richtlinien hier ist es genau das zu verhindern.

1.1 Sauberer Code ist die Kombination aus vielen Details

Du musst nicht die eine Sache beachten und Dein Projekt wird übersichtlich. Es sind neben der groben Struktur viele kleine Details.

Es ist wie bei anderen Kunstformen auch. Du bemerkst beim Betrachten nicht, was es so toll macht. Aber es wirkt einfach gut.

Viele sind in der Lage sofort zu beurteilen ob ihnen eine Zeichnung oder ein Gemälde gefällt. Aber deswegen sind sie noch lange nicht in der Lage selbst eines zu produzieren.

Mit den Richtlinien in diesem eBook lernst Du, die kleinen Details zu einem Meisterwerk zu kombinieren.

1.2 Erfolg ist die Summe richtiger Entscheidungen?

Es gibt dieses Sprichwort: "Erfolg ist die Summe richtiger Entscheidungen". Mathematisch betrachtet also:

$$\textit{Detail A} + \textit{Detail B} + \textit{Details C} + \dots = \textit{Erfolg}$$

Das möchte ich für die Richtlinien umformulieren: "Sauberer Code ist das Produkt richtiger Entscheidungen":

$$\textit{Detail A} \times \textit{Detail B} \times \textit{Details C} \times \dots = \textit{Erfolg}$$

Diese ganzen Richtlinien verstärken sich gegenseitig. Sie greifen im ganzen Projekt. Wenn Du quer durch Deinen Code gute Namen wählst, macht es jede Logik besser lesbar.

Schreibst Du grundsätzlich kurze Methoden, wird der Code noch besser lesbar.

Haben die Methoden auch noch gute Namen, dokumentiert sich der Code auch noch selbst.

Jeder der folgenden Tipps ist also nicht einfach in kleines Detail. Mit jedem gut umgesetzten Punkt wird die Qualität um ein vielfaches angehoben.

Jetzt aber erstmal viel Spaß und viel Erfolg bei der Umsetzung.

2 Allgemeiner Stil

2.1 Schreibe in camelCase

Namen von Variablen, Methoden und Klassen werden bei JBCommerce in *camelCase* geschrieben:

```
// Methode
public function getProduct() {}

// Variable
$productName = 'xyz';

// Klasse
class UserRepository {}
```

Du musst nur die Wörter zusammensetzen. Der erste Buchstabe wird bei der Verbindung immer groß geschrieben.

2.1.1 Halte Dich an die Vorgabe der Frameworks

Python allgemein bevorzugt `snake_case`. So eine Vorgabe gibt es bei PHP nicht.

Best Practice ist es, Dich an die Vorgabe Deines Frameworks zu halten. In Symfony wird `camelCase` bevorzugt (siehe [hier](#)).

Zum Beispiel WordPress setzt auf `snake_case` (siehe [hier](#)). Aber ich würde WordPress ohnehin ignorieren, wenn es um Stil beim Programmieren geht.

2.2 Werte in Konstanten, die gesucht werden können

In Deinem Code hast Du immer wieder Konstanten. Etwas das nicht durch Konfiguration & Co. gesetzt werden muss. Feste Größen zum Beispiel:

```
class RateCalculator {
    public function getRatePerWeek($dailyRate) {
        return $dailyRate * 5;
    }
}
```

Es sind 5 Werktage pro Woche. Mach den Code lieber gleich sprechend:

```
class RateCalculator {
    const WORK_DAYS_PER_WEEK = 5;

    public function getRatePerWeek($dailyRate) {
        return $dailyRate * self::WORK_DAYS_PER_WEEK;
    }
}
```

Du kannst nach solchen Bezeichnungen suchen. Die Ziffer 5 wirst Du später über Volltextsuche nicht so einfach finden.

Bedenke immer: die Beispiele hier sind schlicht. Wenn Du es tausende Zeilen und hunderte Klassen gibt, geht Übersicht verloren.

2.3 Instanzvariablen, use und Konstanten sortieren

Bei der Deklaration von Instanzvariablen ist die alphabetische Sortierung nach Namen ein echtes Sahnehäubchen:

```
class User {
    private $email;
    private $id;
    private $name;
}
```

Das funktioniert auch bei use-Statements oben in der Klasse:

```
<?php

namespace My\App;

use My\App\Db\Adapter;
use My\App\Session\Helper;
use My\App\Model\UserFactory;

// ...
```

Auch Konstanten werden so übersichtlicher:

```
class Cache {
    const CACHE_DEFAULT_PREFIX = 'MyApp';
    const EXPIRE_BY_DEFAULT = true;
    const EXPIRE_IN_DAYS = 7;
}
```

2.4 Englische Bezeichnungen

Nutze englische Bezeichnungen. Die Chance in heutigen Teams Entwickler zu finden die international arbeiten, ist hoch.

Bei JBCommerce sind wir komplett Remote aufgestellt. Das Konzept macht vor Ländergrenzen keinen halt.

Gleichzeitig führen englische Bezeichner zu Code der sich dokumentiert. Der Satzbau macht die Aktion besser lesbar:

```
//
$db->benutzerErstellen($id);
$db->benutzerLoeschen($id);
$db->benutzerAktualisieren($id);

//
$db->createUser($id);
$db->deleteUser($id);
$db->updateUser($id);
```

Das Verb steht im englischen vor dem Objekt. Das geht in Deutsch in Kurzform nicht mit sinnvollen Aussagen.

2.5 Kein global verwenden

Es ist hoffentlich nur eine Formalität. Aber globale Variablen werden nicht verwendet. Punkt.

Sie waren schon immer eine schlechte Idee. Vielleicht auch einer der Gründe warum es Vorurteile über PHP gibt.

Fakt: Sie öffnen Tür und Tor für Seiteneffekte. Niemand weiß, wo gerade eine Veränderung stattfindet.

2.6 Verwende keine zu Bezeichnungen

Robert Martin empfiehlt in Clean Code zwischen Bezeichnungen aus der Problem Domain (der Umgebung der Aufgabenstellung) und Bezeichnungen aus der Solution Domain (Umgebung der Problemlösung) zu unterscheiden.

Reales Beispiel: Dein Auftraggeber ist eine Druckerei. Begriffe aus der dortige Fachsprache sind Begriffe aus der "Problem Domain".

Du musst Text vor einem Druck prüfen. Werden bestimmte Buchstaben verwendet, muss die Schrift zwei Pixel kleiner sein.

Die Buchstaben d, b und h haben einen aufsteigenden Teil. Der reicht über den Buchstaben hinaus. Der englische Begriff ist "Ascender".

Eine Methode um das zu prüfen:

```
// schlecht, da zu spezifisch
class PrintValidator
{
    public function containsAscender($text)
    {
        // ...
    }
}
```

Ein außenstehender wird das nicht nachvollziehen können. Du kannst mit einem Kommentar erklären, aber besser wäre, die Methode anders zu nennen:

```
class PrintValidator
{
    public function containsLargeCharacter($text)
    {
        // ...
    }
}
```

Die Methode dokumentiert sich wieder selbst, auch wenn Deine Urlaubsvertretung in den Code schaut. Du bist Teil der Lösung, nicht des Problems. Gern geschehen.

2.7 Verwende ausreichende Abstraktion

Versuche Deine Logik und Daten angemessen zu abstrahieren. Nehmen wir einen Import von Produkten.

Aktuell lädt der Kunde CSV Dateien hoch. Du liest sie ein und arbeitest, quer durch die Applikation, mit dem zurückgegebenen Array.

Was, wenn sich nun die Datenquelle ändert? Der Kunde stellt auf eine andere Warenwirtschaft um. Die Daten kommen nun per XML.

Wenn Du vom direkten Rückgabewert abhängig gewesen bist, hast Du nicht genug abstrahiert. Besser wäre es, eine Klasse für den Datentransport zu definieren (DataTransferObject).

Bei der Umstellung musst Du vielleicht nun viele Stellen ändern. Quer durch das ganze Projekt.

Man spricht dabei auch von "Shotgun Refactoring". Das führt immer zu weitreichenden Fehlern und umfassenden Tests. Das ist sehr unangenehm, Kostet Zeit und bleibt dauerhaft fehleranfällig.

2.8 Einheitliche Formatierung

Zur Formatierung, also wie weit eingerückt wird, welche Leerzeichen bleiben, gibt es keine festgelegte Richtlinie.

Jeder hat seine eigenen Vorlieben

Du solltest im gesamten Team den gleichen Stil zu verwenden. Und gleichzeitig sollte eine IDE automatisch für die Formatierung sorgen.

Es macht das Leben um Längen einfacher.

3 Kontrollfluss

3.1 Keine kreativen Namen für Schleifenzähler

Die einzigen Variablen die \$i, \$j oder \$k heißen dürfen, sind Schleifenzähler. Diese werden per Konvention so genannt. Du findest sie in allen Sprachen wieder.

Es ist also nicht nur akzeptiert. Jeder Entwickler weiß etwas damit anzufangen. Versuch also nicht mit kreativen eigenen Namen zu arbeiten.

Und fang immer bei \$i an. Du benötigst \$j nur in einer verschachtelten Schleifen. Wenn es darin noch eine Schleife gibt, wäre auch noch \$k möglich. Eine vierte Ebene sollte es nicht mehr geben.

3.2 Logik der switch-Statement Fälle auslagern

Ein switch-Statement sollte ohnehin nicht lang werden. Aber auch innerhalb der Fälle sollte der Code nicht direkt hinterlegt werden. Lagere die Logik lieber in Funktionen aus.

```
class AuthAdapter
{
    public function processLogin(User $user)
    {
        switch ($user->getStatus()) {
            case User::STATUS_ENABLED:
                $this->executeLogin();
                break;
            case User::STATUS_DISABLED:
                $this->handleDisabledUser();
                break;
            case User::STATUS_EMAIL_PENDING:
                $this->handleEmailVerficiation();
                break;
        }
    }
    //...
}
```

Direkt innerhalb der verschiedenen case-Definitionen Logik zu implementieren, ist immer unübersichtlich.

3.3 Bedingungen nicht unnötig verschachteln (Early Exit)

Die Gefahr mehrfacher Verschachtelung ist groß. Aber das ist nicht nur formal unsauber. Es ist einfach wirklich maximal unübersichtlich. Im folgenden Beispiel sind es nur wenige Zeilen. In der Praxis ufert das schnell aus.

```
// so nicht
class AuthAdapter
{
    public function login(User $user)
    {
        if ($user && $user->isEnabled()) {
            if ($this->loginHandler->authenticate($user)) {
```

```

        // ...
        return true;
    } else {
        // ...
    }
} else {
    // ...
}
}
}

```

Die Fehlerprüfung ist richtig. Aber die Verschachtelung lässt sich auflösen:

```

class AuthAdapter
{
    public function login(User $user)
    {
        if ($user && $user->isEnabled()) {
            return $this->processLogin($user);
        }
    }

    public function processLogin(User $user)
    {
        if ($this->loginHandler->authenticate($user)) {
            // ...
            return true;
        } else {
            return false;
        }
    }
}

```

Die zweite Variante ist schon ohne weitere Logik viel übersichtlicher.

3.4 Exceptions sind für Fehlerbehandlung besser

Müssen Fehler verarbeitet werden, sind Exceptions das Mittel der Wahl. Rückgabewerte müssen bei der Fehlerverarbeitung durchgereicht werden. Das kann sich über mehrere Ebenen ziehen:

getProductDescriptionForId(\$id)

- ruft getProductById(\$id) auf
- darin wird z.B. ein Repository oder anderes geladen
- das Repository greift auf ein ResourceModel zu
- die Datenbankabfrage erfolgt

Wenn das das Produkt nicht gibt oder andere Fehler in der Kette auftreten, ist es umständlich die Rückgabewerte weiterzugeben.

Exceptions können irgendwo in dieser Reihe verarbeitet werden. Unter anderem genau da, wo Du die Methoden aufrufen willst.

Du kannst den Fehler mit try/catch behandeln und Rückmeldung an den Benutzer geben.

3.5 Den Code aus try/catch Blöcken auslagern

Durch try/catch ist Dein Code automatisch verschachtelt. Er bleibt übersichtlich, wenn Du die Logik darin auslagerst.

```
// statt
try {
    $this->db->deleteUser($id);
    $this->updateSession();
    $this->clearCache();
} catch (Exception $e) {
    $this->logger->log($e->getMessage());
}
```

Lieber ...

```
public function deleteUser($id)
{
    try {
        $this->deleteFromUserDb($id);
    } catch (Exception $e) {
        $this->logError($e);
    }
}

private function deleteFromUserDb($id)
{
    $this->db->deleteUser($id);
    $this->updateSession();
    $this->clearCache();
}

public function logError(Exception $e) {
    $this->logError($e);
}
```

3.6 Setze auf positive if-Bedingungen

Für Menschen sind positive Bedingungen unmittelbar zu verstehen. Keine Interpretation notwendig. Statt die negative Situation zu prüfen, versuch das positive Szenario testen:

```
if ($this->isEnabled()) {
    $this->executeImport();
}
```

Statt...

```
if (!$this->isEnabled()) {
    return;
}
```

4 Variablen

4.1 Namen von Variablen werden immer klein geschrieben

Werden Variablen mal groß und mal klein geschrieben, ist das unglaublich unübersichtlich. Nur über das \$ lässt sich erkennen worum es geht.

```
// so nicht:  
$User = $session>getUser();  
  
// immer so:  
$user = $session>getUser();
```

Großer Anfangsbuchstabe steht immer für eine Klasse. Aber dazu später mehr.

4.2 Namen beschreiben den Inhalt der Variable

Jede Variable muss deutlich machen was in ihr gespeichert ist. Kryptische Abkürzungen wirken manchmal clever, sind aber nicht hilfreich:

```
// schwer zu lesen:  
$d = date('Y-m-d');  
  
// einfacher:  
$currentDate = date('Y-m-d');
```

Bei der ersten Variante musst Du im Zweifel die Stelle finden, an der sie initialisiert wurde. Das ist bei Objektorientierung mit mehreren Abstraktionsebenen nicht immer einfach. Und es kostet Zeit.

4.3 Aussprechbare Namen verwenden

Das menschliche Gehirn ist gut im Umgang mit Wörter und Geschichten. Nutze das als Entwickler.

Eine Variable soll den Inhalt beschreiben. Wenn der Name leicht auszusprechen ist, machst Du Dir den Großteil unseres Gehirns zu nutze.

```
// schlecht:  
$date_ymd = date('Y-m-d');  
  
// auch besser:  
$currentDate = date('Y-m-d');
```

Die erste Variable zeigt zwar den Inhalt, aber ist trotzdem nicht lesbar.

5 Klassen

5.1 Klassennamen sind Nomen

Der Klassenname sollte immer ein Nomen sein. Wichtig ist die Einzahl zu beachten, da das Objekt am Ende für eine Instanz steht.

- User
- Customer
- Product
- Session

Wenn es sich um eine Liste handelt, ist Mehrzahl nicht eindeutig genug:

```
// schlecht
class Users
{
    public function getAllEntries() { ... }
}

// besser
class UserList
{
    public function getAllEntries() { ... }
}
```

5.2 Keine generischen Klassennamen

Generische Klassennamen führen schnell zu Verwechslungen. Es ist nicht klar was genau die Aufgabe ist:

- Manager
- Handler
- Processor
- Data

Eingebettet in Namespaces ist es leichter lesbar. Einfach für sich freistehend wären diese Klassen ohne einen Blick in den Code nicht zu verstehen.

5.3 Keine Verben in Klassennamen

Verben sind für Methoden reserviert.

5.4 Klassennamen immer groß

Die Namen von Klassen werden immer groß geschrieben. Die Abgrenzung zu Variablen ist so eindeutiger:

```
class User {
    // ...
}

$user = new User();
```

Es hilft auf den ersten Blick zu unterscheiden. Beim Zugriff auf Konstanten und statische Methoden ist es ebenfalls hilfreich:

```
// Singleton
Session::getInstance();

// Konstante
UserConfig::MAX_CLIENTS
```

5.5 Instanzvariablen immer deklarieren

Es ist möglich Eigenschaften einer Klasse zu setzen, ohne die Variable zu deklarieren. Das ist umfangreichen Klassen unübersichtlich.

```
// falsch:
class User {
    public function __construct($id, $name) {
        $this->id = $id;
        $this->name = $name;
    }
}

// richtig: immer explizite Eigenschaften anlegen

// immer so:
class User {
    private $id
    private $name;

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

Die Eigenschaften sind ohne Deklaration automatisch public. Entwickler können so die Kapselung und getX/setX Methoden umgehen.

5.6 Instanzvariablen immer über den Konstruktor

Die Eigenschaften einer Klasse stehen immer oben über dem Konstruktor, der Methode `__construct()`.

Wenn sie später gesucht werden, ist sofort klar, wo sie gefunden werden:

```
// immer so:
class User {
    private $id
    private $name;

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

5.7 Instanzvariablen im richtigen Kontext definieren

Es geht dann auch fast schon in Richtung SOLID-Prinzipien. Aber es gibt Grenzfälle wie zum Bei-

spiel bei einem Benutzer und seinen Adressinformationen.

Nimm an es gibt die Klasse User. Enthalten sind Informationen (auf Englisch):

- zipcode: Postleitzahl
- street: Straße
- city: Stadt
- state: Bundesland

Der Aufruf von außen erfolgt auf die get-Methode für State:

```
$state = $user->getState();
```

Es ist nicht eindeutig, ob hier vielleicht auch ein Status gemeint ist. Du könntest einen Prefix verwenden:

- \$addrZipcode
- \$addrStreet
- \$addrCity
- \$addrState

Der noch bessere Weg wäre eine Klasse für die Adresse. Der Benutzer hat dann nur noch die als Eigenschaft.

5.8 Namen nicht unnötig aufblähen

Wenn Du in einem bestimmten Kontext arbeitest, müssen Eigenschaften und Methoden den Kontext nicht noch einmal enthalten. Es hilft nicht bei der Übersicht.

```
// überflüssig
class Product
{
    private $productId;
    private $productDescription;
    private $productName;
}

// sinnvoller
class Product
{
    private $id;
    private $description;
    private $name;
}
```

Das gilt natürlich auch für die Methoden:

```
// überflüssig
class Product
{
    public function getProductName()
    {
        // ...
    }
}
```

Der Kontext definiert bereits, was für ein Name es ist:

```
// besser
class Product
{
    public function getName()
    {
        // ...
    }
}
```

5.9 Verzichte auf das Singleton Pattern

Ich habe das Singleton Pattern selbst viel genutzt. Es hat mich bei Magento 1 Jahre begleitet.

Das Singleton Pattern erschafft Abhängigkeiten. Diese Abhängigkeiten lassen sich von außen nicht sehen.

Tauscht Du die Singleton Klasse aus, müssen alle Stellen mit Aufrufen geändert werden.

Die bessere Alternative ist Dependency Inversion (siehe SOLID). Hier kannst Du über Interfaces Deinen Code viel einfacher erweiterbar halten.

Gleichzeitig ist von außen sichtbar, welche anderen Klassen in einer Klasse verwendet werden.

5.10 Kapsle alle Daten

Der direkte Zugriff auf Eigenschaften der Klassen ist theoretisch möglich. Der Zugriff über get/set-Methoden ist aber deutlich sinnvoller. Du kannst beim Abruf (get) Veränderungen vornehmen und neue Werte kontrollieren (set).

```
// schlecht
class User
{
    public $name;

    // ...
}

$user = new User();
$user->name = "Jan";

// besser:
class User
{
    private $name;

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

    public function setName($name)
    {
        if (strlen($name) === 0) {
```

```
        throw new Exception("Name can't be empty");
    }

    $this->name = $name;
}
}
```

Selbst wenn das in 95% der Fälle nicht passiert. Manchmal kommt es vor, häufig erst später. Wenn dann der direkte Zugriff auf Eigenschaften verboten, führt das zu weitreichenden Änderungen.

Nicht jeder Code wird nur intern verändert. Gerade wenn es um Plugins, Extensions & Co. geht, arbeiten andere Entwickler mit Deinen Klassen. Grobe Veränderungen führen dort zu Fehlern und notwendigen Nacharbeiten. Das kannst Du verhindern. Kapsle Deine Daten direkt.

5.11 Schreibe sich aufrufende Methoden untereinander

Dein Code lässt sich am besten von oben nach unten lesen. Es entsteht ein natürlicher Lesefluss, wenn der Aufrufer über der aufgerufenen Methode definiert ist.

```
class ImportCommand {
    // ...

    public function executeImport()
    {
        $products = $this->getProducts();
        $this->saveProducts($products);
    }

    public function getProducts()
    {
        $importData = $this->importSource->getData();
        // do stuff

        return $entries;
    }

    // ...
}
```

Das kannst Du bis unten durchhalten.

6 Kommentare

6.1 Nutze Kommentare nicht, um Code zu erklären.

```
// check if user is admin or has required ROLE
if ($user->getStatus() == 'admin' || $user->hasRole('view_orders') {
    // ...
}
```

Viel besser:

```
if ($user->canViewOrder()) {
    // ...
}
```

6.2 Keine Kommentare um Konstante Werte zu erklären

Manchmal gibt es Konstante Werte oder etwas wie eine Typ-ID. Statt mit einem Kommentar den Wert zu erklären:

```
// check wether product is a grouped product
if ($product->getTypeId() == 4) {
    // ...
}
```

Lieber explizit:

```
if ($product->isGroupedProduct()) {
    // ...
}
```

Oder wenigstens mit einer Konstante arbeiten:

```
if ($product->getTypeId() == self::TYPEID_GROUPED_PRODUCT) {
    // ...
}
```

6.3 PHPDoc: Parameter Rückgabewerte dokumentieren

Jede Methode sollte einen PHPDoc Kommentar erhalten. So dokumentierst Du Parameter, Rückgabewerte und ähnliche Verhaltensweisen:

```
/**
 * @param $ordernumber
 * @param int $count
 * @return \Doctrine\DBAL\Driver\Statement|int
 */
public function reduceStock($ordernumber, $count = 1)
{
    $queryBuilder = $this->connection->createQueryBuilder();
    $queryBuilder->update('s_articles_details', 'd')
        ->set('d.instock', 'd.instock - :amount')
        ->setParameter(':amount', $count)
        ->where('d.ordernumber = :ordernumber')
        ->setParameter(':ordernumber', $ordernumber)
        ->execute();
}
```

6.4 Einen erklärenden Satz zu jeder Methode

```
/**
 * Reduce available stock in the store by specified quantity
 *
 * @param $ordernumber
 * @param int $count
 * @return \Doctrine\DBAL\Driver\Statement|int
 */
public function reduceStock($ordernumber, $count = 1)
{
    $queryBuilder = $this->connection->createQueryBuilder();
    $queryBuilder->update('s_articles_details', 'd')
        ->set('d.instock', 'd.instock - :amount')
        ->setParameter(':amount', $count)
        ->where('d.ordernumber = :ordernumber')
        ->setParameter(':ordernumber', $ordernumber)
        ->execute();
}
```

Im Beispiel hier wird nun auch der Hintergrund der Methode erklärt.

6.5 Schreib noch offene Aufgaben in TODO Kommentare

Es gibt manchmal kleine Aufgaben oder Verbesserungen, die noch zu erledigen sind. Nicht immer klappt alles. Oder es fehlen für den letzten Schritt Informationen.

Statt Deine bisherige Aufgabe zu Arbeit zu beenden, schreib direkt im Code noch einen TODO Kommentar. Eine gute IDE wie PhpStorm kann diese Stellen finden.

Außerdem weißt Du am nächsten Tag genau, was Du noch ändern wolltest:

```

class RedisCache {
    public function setValueForKey($key, $value) {
        // ...
    }

    public function getValueForKey($key) {
        // ...
    }

    // TODO: add method to reset certain keys
}

```

6.6 Verzichte auf redundante Kommentare

Redundant meint hier überflüssig oder doppelt:

```

public function updateStock($ordernumber, $count = 1)
{
    // get query builder instance
    $queryBuilder = $this->connection->createQueryBuilder();
}

```

Zum einen ist der Name schon fast Dokumentation genug.

In dem Beispiel ist jedem Symfony oder Shopware Entwickler klar was passiert. Ein kompletter Neueinsteiger lernt die Bedeutung sehr schnell.

Solche Dinge sollten nicht kommentiert werden. In diesem Beispiel geht es um 4-5 Zeilen. In der Praxis ist aber eine Methode häufig länger. Wenn Du bei 20 Zeilen 2 sparen konntest, hast Du die Methode um 10% gekürzt!

6.7 Auskommentierter Code

Wenn Code nicht mehr benötigt wird, muss er entfernt werden. Für einen Test können lokal Zeilen deaktiviert werden. Die bleiben aber nicht im Code - falls man sie noch benötigt. Für die Historie gibt es schließlich Git.

6.8 Zu lange Kommentare

Merkmal von einem guten Kommentar: kurz und präzise. Wenn eine Funktion mehr als 1 - 2 Sätze benötigt, geschieht vermutlich zu viel.

Umfangreiche Dokumentationen sollten ohnehin separat erfasst werden. Es gibt wesentlich bessere Formate als Quellcode Kommentare.

7 Methoden

7.1 Jede Methode hat nur eine Aufgabe

In jeder Methode gibt es nur eine Aufgabe! Das können ganz unterschiedliche Dinge sein:

- Daten aus der Datenbank abrufen
- aus Daten im Array Objekte erstellen
- die Objekte über ein SDK an eine API senden

Häufigster Fehler: diese 3 Schritte in einer Methode hinterlegen. Viel sauberer ist es alle Aufgaben zu trennen. In einer übergeordneten Methode werden sie aufgerufen und dann mit dem Rückgabewert gearbeitet.

7.2 Mehr als 2 Parameter verwenden

Es gilt als guter Stil im besten Fall keine oder nur einen Parameter zu übergeben.

```
public function getXyz() { ... }  
  
public function setName($newName) { ... }
```

Es gibt aber sinnvolle Anwendung für 2 Parameter:

```
public function multiply($x, $z) {  
    return $x * $z;  
}
```

Drei oder mehr? Bis auf wenige Ausnahmen, generell vermeiden. Oft lässt sich eine andere Struktur finden.

7.3 Parameter unterscheidbar benennen

Parameter sind Variablen. Es gelten die gleichen Konzepte. Lesbare Namen, die man aussprechen kann.

Bei einem Parameter sollte auch klar sein, was er bedeutet.

```
// schlecht  
class FileHandler  
{  
    public function copy($file1, $file2) { ... }  
}  
  
// besser  
class FileHandler  
{  
    public function copy($source, $destination) { ... }  
}
```

Nur diese kleine Änderung macht den Code doppelt so gut. In einer modernen IDE wie PhpStorm, siehst Du in welcher Reihenfolge die Parameter anzugeben sind. Im ersten Beispiel musst Du vermutlich erst die Implementierung anschauen.

7.4 Keine flag-Argumente verwenden

Einen booleschen Wert als flag-Argument zu verwenden ist unsauber. Jede Methode soll nur eine Aufgabe haben. Richtig?

Mit einem flag-Argument erweiterst Du die Funktion allerdings. Und noch schlimmer: Du musst das Flag meist mit durch mehrere Methoden tragen.

Verzichte auf so etwas:

```
class ProductFactory
```

```

{
    public function createProduct(bool $isGroupedProduct)
    {
        if ($isGroupedProduct) {
            //...
            return $xyz;
        } else {
            // ...
            return $abc;
        }
    }
}

```

Besser wäre eine saubere Trennung. Entweder zwei Methoden in der gleichen Klasse, oder noch besser zwei eigene Factory Klassen für unterschiedliche Produktarten.

7.5 Keine Methoden mit mehreren Bereichen

Methoden mit Kommentaren in Bereiche einteilen ist Bad Practice. Gemeint ist:

```

class AddressBook {
    // ...

    public function getAllEmails() {
        // get users from db
        $db = $this->db->getConnection();
        $sql = "SELECT * FROM contacts";
        $users = $db->fetchAll($sql);

        // get email from users
        $emails = [];
        foreach ($users as $user) {
            $emails[] = $user->getEmail();
        }

        return $emails;
    }
}

```

Sauber wäre die einzelnen Schritte in eigene Methoden auszulagern:

```

class AddressBook {
    // ...

    public function getAllEmails() {
        $users = $this->getUsersFromDb();
        return $this->getEmailFromUsers();
    }

    private function getEmailFromUsers()
    {
        $emails = [];
        foreach ($users as $user) {
            $emails[] = $user->getEmail();
        }
    }
}

```

```

        return $emails;
    }

    private function getUsersFromDb()
    {
        $db = $this->db->getConnection();
        $sql = "SELECT * FROM contacts";
        return $db->fetchAll($sql);
    }
}

```

Du benötigst auch keine Kommentare mehr. Die Namen der Funktionen dokumentieren was passiert.

7.6 Verb im Namen einer Methode

In der Objektorientierung stehen die Methoden für die Fähigkeiten. Ein Mensch kann gehen oder springen. Ein Auto kann beschleunigen oder bremsen.

In Methoden sollte der Name immer mit einem Verb beginnen. Es zeigt, was hier passiert. Bei englischer Schreibweise entsteht so fast ein Satz.

Beliebt und häufig genutzt sind Bezeichnungen wie:

- set... / get...
- has...
- fetch...
- is...

Es können auch nur einzelne Wörter sein. Die sollten aber auch beschreiben was passiert:

```

// zu generischer Name
class SMTPAdapter
{
    // schlecht
    public function handle(Mail $mail) {
        // ...
    }
}

// besser explizit: send
class SMTPAdapter
{
    // schlecht
    public function send(Mail $mail) {
        // ...
    }
}

```

Du musst ohne den Code der Methode anzuschauen verstehen, was sie tut.

7.7 Konsistente Verben im gesamten Projekt

Wenn Du Methoden schreibst, gibt es verschiedene Aktionen häufiger. Für das gleiche Prinzip

solltest Du nur einen Präfix verwenden.

Einträge erzeugen, etwas erstellen:

- create...
- new...

Daten abrufen:

- get...
- fetch...
- retrieve...

Speichern:

- save...
- persist...
- store...

Es gibt kein "bestes Wort" dafür. Du solltest aber nur eines verwenden, und dies dann überall.

7.8 Keine Seiteneffekte produzieren

Seiteneffekte sind unerwartete Nebeneffekte, auf die der Name der Methode nicht schließen lässt. Du arbeitest an Stelle A. An der Stelle B entsteht dadurch ein Problem.

Einfaches Beispiel. Du verarbeitest einen Login via Formular. Der Benutzer wird eingeloggt, wenn das Passwort korrekt ist.

```
class LoginValidator
{
    private $encryptMethod;
    private $userDb;
    private $userSession;

    // ...

    public function checkPassword($email, $password)
    {
        // ...
        $encryptedPassword = $this->encryptMethod->getEncryptedPass-
word($password);
        if ($this->userDb->validatePassword($email, $encryptedPassword))
        {
            $this->userSession->init();
            $this->userSession->setEmail($email);
            return true;
        } else {
            return false;
        }
    }
}
```

Solche Fehler sind nicht ungewöhnlich. Problematisch ist die Session. Bei dem Vergleich von Passwörtern darf die nicht initialisiert werden. Wenn die später an anderer Stelle verwendet wird, leert das eventuell die gesamte (bereits laufende) Session.

8 Prinzipien

8.1 DRY - Don't Repeat Yourself

Es kommt immer wieder vor: die gleiche Logik wird mehrfach implementiert.

Natürlich können mal Methoden mehrfach entstehen. Das passiert gerade im Team schnell. Wichtig, das beim Code Review Duplikate entfernt werden.

Aber viel unbemerkter: manchmal wird in mehreren Methoden das Entfernen, das Error-Logging oder ähnliches in gleicher Form geprüft.

Das gilt es in einer Methode an geeigneter Stelle nur einmal zu hinterlegen um dann die Methode mehrfach aufzurufen.

8.2 Gesetz von Demeter

Das Gesetz von Demeter verhindert zu starke Kopplung zwischen Klassen.

Das Gegenteil von Kopplung ist eine Trennung. Bedeutet für Dich als Entwickler: die Klassen sind übersichtlicher, besser wartbar (Du kannst sie leichter ohne Seiteneffekte verändern) und wiederverwenden.

Das Gesetz kannst Du Dir als "Sprich nur mit Deinen direkten Verwandten merken". Es bezieht sich darauf, welche Methoden aufgerufen werden können.

Eine Methode xyz in der Klasse Example, darf nur Methoden folgendes aufrufen:

- alle Methoden von Example (über \$this)
- alle Methoden von Instanzvariablen, die Objekte sind
- Methoden von Objekten, die in xyz erst entstehen
- Methoden von Objekten, die als Argument übergeben werden

Ein vollständiges Beispiel zeigt, was erlaubt ist. Zunächst eine Klasse Example:

```
class Example
{
    private $customer;

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

    public function badCall()
    {
        return $this->customer->getAddress()->getDefaultBilling();
    }

    public function cleanCall()
    {
        return $this->customer>getDefaultBilling();
    }
}
```

Ganz einfacher Aufbau. Es wird ein Kunde hinterlegt. Hier die Klasse dazu:

```

class Customer
{
    private $address;

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

    public function getAddress()
    {
        return $this->address;
    }

    public function getDefaultBilling()
    {
        return $this->address->getDefaultBilling();
    }
}

```

Zu jedem Kunden gibt es eine Adresse, die hier zugewiesen wird:

```

class Address
{
    public function getDefaultBilling()
    {
        return ['...'];
    }
}

```

Der Verstoß gegen das Gesetz von Demeter ist in der Methode `badCall()` zu sehen. Das Problem. In der Methode wird eine Methode aufgerufen, die nicht zu einem "direkten Nachbarn" gehört.

Die Lösung ist in diesem Fall, eine Methode im Customer zu hinterlegen. Darin wird die Methode aufgerufen. Hier ist es kein Verstoß gegen das Prinzip. Die Adresse ist ja eine Eigenschaft der Kundenklasse. In der Methode `cleanCall()` ist der saubere zu sehen.

8.3 SOLID

Die SOLID-Prinzipien sind ein wesentlicher Bestandteil bei der Arbeit in unserer Agentur. Speziell Single-Responsibility, Open-Closed und Dependency Inversion sind für klare Trennung und pflegbaren Code unverzichtbar.

8.3.1 Single-Responsibility

Eine Klasse sollte nur eine Aufgabe haben. Offiziell lautet die Beschreibung "nur einen Grund" um sich zu ändern.

Ein Beispiel:

```

class User {

    private $id;
    private $name;

    // Getter and setter...
}

```

```

    public function save()
    {
        // ...
    }
}

```

Diese Klasse hätte nun jetzt zwei Aufgaben:

- Verwaltung von Benutzerinformationen (Data Transfer Object)
- Speichern der Daten

Ähnliches gilt Methoden wie `printHtml`, `getAsHtml` oder ähnliches. Solche Darstellungen müssen über eigene Klassen realisiert werden.

Für das Speichern der Daten muss eine eigene Klasse geschaffen werden, zum Beispiel ein Resource Model oder ein Repository. Viele Frameworks lösen dies auf verschiedene Weise:

```

class UserRepository
{
    public function save(User $user)
    {
        // ...
    }
}

```

8.3.2 Open-Closed

Deine Projekte sollen offen für Erweiterung (Open for extension), aber geschlossen für Modifikation sein (Closed for modification).

Dein Code soll offen für neue Features sein, ohne dafür grundlegend verändert werden zu müssen.

Nimm einen Produktimport als Beispiel. Deine Hauptklasse ist der Einstiegspunkt. Sie bekommt eine Datenquelle übergeben. Statt eine konkrete Klasse zu verwenden, verlangst Du ein Interface:

```

interface ImportSourceInterface
{
    public function getImportData();
}

```

Das wird nun an die Hauptklasse übergeben:

```

class Import
{
    private $_importSource;

    public function __construct(
        ImportSourceInterface $importSource
    )
    {
        $this->_importSource = $importSource;
    }

    public function execute()
    {

```

```
        $products = $this->_importSource->getImportData();
    }
}
```

Jede Klasse, die das Interface `ImportSourceInterface` importiert, kann übergeben werden. Du brauchst jetzt nur noch eine konkrete Datenquelle implementieren. Folgendes Beispiel deutet CSV Dateien als Quelle an:

```
class CsvSource implements ImportSourceInterface
{
    public function getImportData()
    {
        return [...];
    }
}
```

Die Struktur erlaubt Daten aus neuen Quellen zu lesen (open for extension), ohne den Code umfassend zu verändern (closed for extension). Willst Du in Zukunft XML Dateien verarbeiten:

- eine neue Klasse `XMLSource` erstellen
- an die Importklasse übergeben

Mit Dependency Inversion kannst Du selbst das in eine Konfigurationsdatei auslagern.

8.3.3 Liskov substitution

Die Liskov Substitution ist ein wenig akademischer. Das Prinzip fordert, dass sich alle Subklassen, die von einer Basisklasse erben, sich bei Methodenaufrufen genau gleich verhalten.

Folgendes Beispiel zeigt Klassen für Kontobewegungen:

```
class Account
{
    private $currentBalance;

    // ...

    public function withdraw($amount)
    {
        $this->currentBalance -= $amount;
    }
}
```

Jetzt stellt Dir ein Festgeldkonto vor. Es hat die gleichen Eigenschaften, Kontostand. Aber Du kannst nicht frei über Dein Geld verfügen und etwas abbuchen:

```
class FixedMoneyAccount
{
    private $currentBalance;

    // ...

    public function withdraw($amount)
    {
    }
}
```

Die Methode `withdraw()` wurde überschrieben. Die Methode wird weiter angeboten, bucht aber

nichts vom Konto ab. Das wäre ein Verstoß gegen das Liskov Prinzip.

Die Auflösung könnte sein, diese beiden Klassen zu trennen. Ein Interface wie WithdrawableAccount könnte kennzeichnen, wenn von einem Konto abgebucht werden kann.

8.3.4 Interface segregation

Frei übersetzt bedeutet Interface segregation in etwa "Trennung von Schnittstellen". Und genau darum geht es.

Der Anwender von einem Interface sollte nicht gezwungen sein irrelevante Methoden zu implementieren. Es macht mehr Sinn, mehrere Interfaces zu schreiben.

Folgendes Beispiel zeigt wie es nicht gehen sollte:

```
interface Vehicle
{
    public function accelerate();
    public function slowDown();
    public function startEngine();
}
```

Das Problem ist die startEngine() Methode. Willst Du ein Fahrrad implementieren, wäre startEngine überflüssig.

Die Lösung könnte sein die Interfaces zu trennen:

```
interface MotorizedVehicle
{
    public function accelerate();
    public function slowDown();
    public function startEngine();
}
```

```
interface Vehicle
{
    public function accelerate();
    public function slowDown();
}
```

8.3.5 Dependency Inversion

Die Dependency Inversion besagt, dass Deine Klassen nicht die benötigten Instanzen einer Klasse (Objekte) selbst erstellen. Sie werden stattdessen übergeben, entweder über setXyz-Methoden (setter-Injection), oder über den Konstruktur:

```
class Import
{
    private $_importSource;

    public function __construct(
        ImportSourceInterface $importSource
    )
    {

        $this->_importSource = $importSource;
    }
}
```

```
public function execute()  
{  
    $products = $this->_importSource->getImportData();  
}  
}
```

Bei Dependency Injection musst Du als Entwickler nicht einmal darum kümmern, Objekte aktiv zu übergeben. Die werden vom Dependency Injection Container erstellt, verwaltet und dann wie angefordert übergeben.

9 Fazit und Anwendung der Prinzipien

Die gute Nachricht zuerst: Du musst nicht alle Prinzipien auswendig lernen und sofort anwenden.

Achte bei der Entwicklung auf alles, was Dir auffällt. Setz Dich aber nicht unter Druck. Vor Angst gegen Prinzipien zu verstoßen ständig alles zu hinterfragen, lähmt Dich. Du kommst nie in den Flow, wirst nichts fertig bekommen.

Habe Mut zur Lücke, stelle erstmal den “ugly first draft” fertig. Kritische jede Zeile hinterfragen? Das ist eine Aufgabe für Code Reviews. Aufräumen musst Du mit Refactoring.

Du wirst auf Dauer immer besser. Je öfter Du Deinen alten Code überarbeitest, desto mehr Prinzipien wendest Du unbewusst direkt an.

In diesem Sinne: lass dich nicht verunsichern. Diese Regeln sind alle hilfreich. Und ja, je mehr Du anwendest, desto besser der Code. Aber jeder fertige und nur halb perfekte Code ist besser, als die perfekt optimierte Klasse, die nicht ausgeliefert wird.

Vergiss nie:

“Good things come to those who ship”.