TECHNIKEN FÜR SAUBEREN CODE

CLEAN CODE

Developer Camp 2016 | Würzburg | Thomas Swonke | webfactor media GmbH

INHALTSVERZEICHNIS

EINFÜHRUNG

WAS BEDEUTET CLEAN CODE?

- Begriff geprägt von Robert C. Martin
- Stabilere, besser wartbarere und testbare Programme
- Schnellere Umsetzung von Erweiterungen bzw. Bugfixes
- Auswahl an konkreten Beispielen
  - Codestyling (Definieren!)

CODESTYLING

- Globale Standards, z. B. PSR-2 für PHP (s. http://www.php-fig.org/psr/psr-2/)
  - Indents, Whitespaces, Bracket Placement… etc…
  - IDEs unterstützen hier (z. B. PHPStorm)!
- Zumindest firmenweite Konvention!
  - Schreibung von Klassen, Methoden und Tabellen (camel case / snake case)
  - => jeder Developer findet sich schnell zurecht

WAS BEDEUTET CLEAN CODE?

- Begriff geprägt von Robert C. Martin
- Stabilere, besser wartbarere und testbare Programme
- Schnellere Umsetzung von Erweiterungen bzw. Bugfixes
- Auswahl an konkreten Beispielen
  - Codestyling (Definieren!)
  - Design Patterns (sollten bekannt sein)
  - convention over configuration (z. B. Model/Table, Frameworks!)

WAS BEDEUTET CLEAN CODE?

- => DISZIPLIN!

PRINZIPIEN (SOLID)

WAS BEDEUTET SOLID?

- Single-Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

SINGLE RESPONSIBILITY PRINCIPLE

- Jede Klasse/Methode/Funktion hat EINEN Aufgabenbereich
- Sobald Mischung => Extraktion vermutlich sinnvoll:
  - z. B. Authentifizierung, Datenbankabfrage und Ausgabeformatierung in einer Klasse
  - besser Auslagern und bei Ausgabe mit Interface arbeiten

OPEN-CLOSED PRINCIPLE

- offen für Erweiterung, aber geschlossen für Veränderung
- Lösung: „Separate extensible behavior behind an interface and flip the dependencies“
- Beispiele:
  - Flächenbezeichnung
  - Bezahlmöglichkeit

LISKOV SUBSTITUION PRINCIPLE

- Mathematische Definition lassen wir weg ;-)
- Abgeleitete Klassen müssen auch als Substitution für die Basisklasse verwendet werden können
- Rechts nicht der Fall!

```php
<?php

class VideoPlayer {
    public function play($file)
    {
        // play the video
    }
}

class AviVideoPlayer extends VideoPlayer {
    public function play($file)
    {
        if (pathinfo($file, PATHINFO_EXTENSION) !== 'avi')
        {
            throw new Exception; // violates the LSP
        }
    }
}
```

LISKOV SUBSTITUION PRINCIPLE

- Mathematische Definition lassen wir weg ;-)
- Abgeleitete Klassen müssen auch als Substitution für die Basisklasse verwendet werden können
- Rechts nicht der Fall!

```php
interface LessonRepositoryInterface {
    public function getAll();
}

class FileLessonRepository implements LessonRepositoryInterface {

    public function getAll()
    {
        // return through filesystem
        return [];
    }
}

class DbLessonRepository implements LessonRepositoryInterface {

    public function getAll()
    {
        return Lesson::all(); // violates the LSP
    }
}
```

INTERFACE SEGREGATION PRINCIPLE

- Ein Client sollte nicht gezwungen werden, ein Interface zu implementieren, dass er nicht benötigt

- Beispiel: Reminder für User
  - verlangt User-Model => weitere Abhängigkeit (z. B. Eloquent)
  - vieles wird nicht benötigt
  - besser: RemindableInterface => eine(!) Methode (akzeptiert aber User-Model, falls implementiert!)

DEPENDENCY INVERSION PRINCIPLE

- Dependency Inversion != Dependency Injection
- Depend on abstraction not on concretion

```php
<?php

class PasswordReminder {

    /**
     * @var MySQLConnection
     */
    private $dbConnection;

    public function __construct(MySQLConnection $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }
}
```

DEPENDENCY INVERSION PRINCIPLE

- Dependency Inversion != Dependency Injection
- Depend on abstraction not on concretion

```php
<?php

interface ConnectionInterface {
    public function connect();
}

class DbConnection implements ConnectionInterface {

}

class PasswordReminder {

    /**
     * @var MySQLConnection
     */
    private $dbConnection;

    public function __construct(ConnectionInterface $dbConnection)
    {
        $this->dbConnection = $dbConnection;
    }
```

- Es gibt noch weitere Prinzipien
- Es kann durchaus sein, dass man sie manchmal bricht (brechen muss?), aber man sollte sich dessen bewusst sein!
- In der Regel bricht man durch das Nicht-Einhalten einer Regel (z. B. Liskov Substitution Principle) an anderer Stelle weitere Regeln (z. B. Single Responsibility Principle aufgrund zusätzlicher if-Statements
- Alle Prinzipien hängen oft zusammen und letztlich geht es um:
  - decoupling code

WAS BEDEUTET CLEAN CODE?

- => CODE TO AN INTERFACE!

SMELLS / HEURISTIK

- Kommentare:
  - ungeeignete Informationen (Versionierung, Autoren)
  - alte Kommentare (überholt, vielleicht sogar falsch)
  - Redundanz (das Offensichtliche kommentieren)
  - Schlecht geschrieben, zu lang
  - Auskommentierter Code! (=> VCS!)

SMELLS / HEURISTIKEN

- Zu viele Argumente in Funktionen/Methoden (unübersichtlich)
- Tote Funktionen / allg. toter Code
- Mehrere Programmiersprachen pro Datei
- Verhalten irreführend / unklare Benennung
- Grenzverhalten!
- Duplizierung!

- Magische Zahlen
  - 86.400 vs. SECONDS_PER_DAY
- Bedingungen kapseln
  - if(hasExpired() && hasPermission()) vs. if(shouldBeDeleted())
- Negative Bedingungen
  - if(!shouldBeDeleted()) vs. if(schouldNotBeDeleted())
- Eine Aufgabe pro Methode / Funktion (SRP)
- Eine Ebene für Kontrollstrukturen

- Namen
  - Verantwortungsvoll / lesbar / beschreibend („was tut dieser Code (wirklich)?“)
  - Standards folgen
  - Eindeutig sein: doRename()?
  - Je größer der Geltungsbereich, desto länger der Name
    - $i als Inkrementvariable für ein paar Zeilen Code völlig ok

FAZIT

FAZIT

- Codestyling / Konventionen!
- Namen, Namen, Namen!
- SRP (Single Responsibility Principle) ist ein guter Anfang! :)

- Hilft Devs und Chefs! ;-)

- Tipp #1: Frameworks nutzen aber auch mal tiefer reinschauen (z. B. PHP: Laravel!)
- Tipp #2: Clean Coder, ebenfalls Robert C. Martin (Völkerverständigung!)

DANKE FÜR DIE AUFMERKSAMKEIT!

FRAGEN / DISKUSSION