OOP und Grundgedanken Architektur

PHP OOP - eigentlich ganz einfach

```php
<?php

declare(strict_types=1);

namespace Wickb\Webentwicklung;

class Router extends AbstractRouter implements RegistrationRouter
{
    /**
     * @var RequestInterface $request
     */
    protected RequestInterface $request;

    /**
     * Router constructor.
     * @param RequestInterface $request
     */
    public function __construct(RequestInterface $request)
    {
        $this->request = $request;
    }

    /**
     * @param string $requestUri
     */
    public function route(string $requestUri): void
    {
        // ...
    }
}
```

Erwartungen an Architektur

- Webprojekte sind (oft) nicht trivial
  - Wichtig für unternehmerischen Erfolg der Kunden
  - Langjährige Wartung mit hohen Budgets

- Muss klare Struktur vorgeben
- Soll Wartbarkeit von Software fördern
- Soll Erweiterbarkeit erleichtern
- Sollte zukunftsfähig sein
  - Stabile Basis statt Proof of Concept

Separation of Concerns

- Trennung von Belangen
- Prinzip hinter Modularität von Software
- Ein Modul ist für eine Sache zuständig und erlaubt Zugriff durch Schnittstellen
- Im “Großen” als auch “Kleinen” anwendbar
  - CSS, HTML, Javascript als getrennte Sprachen
  - Pakete die sich über Composer installieren lassen
  - Einzelne Klassen mit klaren Zuständigkeiten z.B. FileLogger, DatabaseLogger, ...

Architektur - SOLID

- Sammlung grundlegender Prinzipien der objektorientierten Software-Architektur
- Soll Wartbarkeit und Lebensdauer von Software erhöhen
- Begründet (auch) von Robert C. Martin, Bertrand Meyer und Barbara Liskov
- Basis vieler anderer Methoden, Pattern, Prinzipien, ...
- SOLID (ca. 2000) steht für:
  - **S**ingle Responsibility
  - **O**pen-Closed Prinzip
  - **L**iskovsches Substitutionsprinzip
  - **I**nterface Segregation
  - **D**ependency Inversion

SOLID - Single Responsibility

- **Klassen (und andere Konstrukte) sollten nur wenig, klar definierte, Aufgaben haben**
- Fokussiert auf hohe Kohäsion/Eindeutigkeit der Aufgabe
- Verringert Wartungsaufwand an einer Klasse/Methode
- Grundlage vieler ander Pattern
- Verhindert “Gott-Klassen”
- Vergleiche **Separation of Concerns**
- Erlaubt strukturierte Architektur

```
class FileSystemLogger implements LoggerInterface
{
    public function log($message, $level)
    {
        // ...
    }

    protected function prepareLogFile()
    {
        // ...
    }
}
```

SOLID - Open-Closed Prinzip

- **Software Konstrukte (Module, Klassen, Methoden) sollen:**
  - **Offen sein für Erweiterung**
  - **Geschlossen für Modifikation**
- Garantiert Erweiterbarkeit
- Beispiel auf Klassenebene: Vererbung
  - Erlaubt Erweiterung durch Kindklassen, Modifikation aber nur im Scope des Kindes
  - Keine Veränderung der Struktur und Aufgabe
- Beispiel auf Architekturebene: Zerlegung von Klassen und Interfaces als Typehint
  - Eierlegende Wollmilchsau-Klassen zerlegen in spezialisierte Einzelklassen
  - Bieten klare Schnittstellen für zusätzliche Implementierungen
  - Absicherung gegen anderes Verhalten

SOLID - Liskovsches Substitutionsprinzip

- Auch Ersetzbarkeitsprinzip
- **Instanzen einer Klasse müssen durch Instanzen ihrer Kindklasse austauschbar sein, ohne unerwartetes Verhalten der Instanz**
  - Kindklassen dürfen nur erweitern, nicht modifizieren oder entfernen!

```php
class LiskovParent
{
    public function log($message, $level)
    {
        // ...
    }
}
```

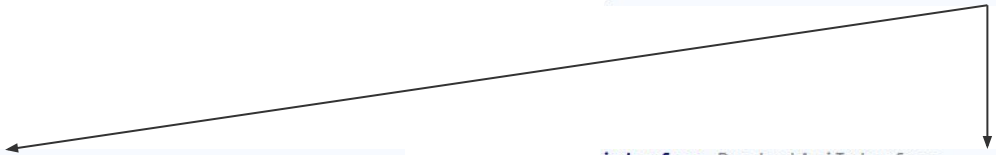
```php
class LiskovNiceChild extends LiskovParent
{
    public function log($message, $level, $foo= 'bar')
    {
        // ...
    }
}
```

```php
class LiskovBadChild extends LiskovParent
{
    private function log($message, $level)
    {
        // ...
    }
}
```

SOLID - Interface Segregation

- **Zu große Interfaces sollen in mehrere kleine aufgeteilt werden**
  - Niemand nutzt unnötige Definitionen
- Erhöht Entkopplung/Vermindert Abhängigkeiten
- Erleichtert Bau von spezifischen Schnittstellen
  - Erleichtert Erweiterbarkeit und Nutzung durch Dritte
  - Verhindert Redundanz und Code Bloat

```php
interface UniversalApiInterface
{
    public function getCustomerData(Criteria $criteria) : Customer ;
    public function updateCustomerData(Customer $customer) : bool ;
    public function getProductData(Criteria $criteria) : Product ;
    public function updateProductData(Product $product) : bool ;
}
```

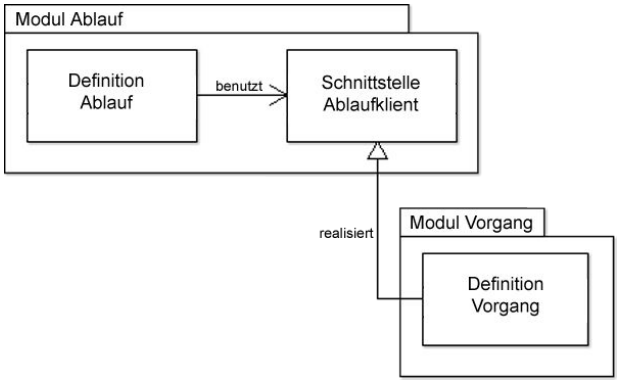


```php
interface CustomerApiInterface
{
    public function getCustomerData(Criteria $criteria) : Customer ;
    public function updateCustomerData(Customer $customer) : bool ;
}
```

```php
interface ProductApiInterface
{
    public function getProductData(Criteria $criteria) : Product ;
    public function updateProductData(Product $product) : bool ;
}
```

SOLID - Dependency Inversion

- **Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen.**
- **Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.**
- Verhindert zyklische Abhängigkeit und enge Kopplung
  - Verhindert “Änderungskaskaden” bei der Wartung
  - Erlaubt Erweiterung über zusätzliche Implementierungen



SOLID - Dependency Inversion

```php
class Lamp
{
    private bool $isGlowing;

    public function switchOn() { /* ... */}
    public function switchOff() { /* ... */}
}
```

```php
class LightSwitch
{
    private Lamp $lamp;
    private bool $isActive;

    public function __construct()
    {
        $this->lamp = new Lamp();
    }

    public function press()
    {
        $this->isActive = !$this->isActive;
        if ($this->isActive) {
            $this->lamp->switchOn();
        } else {
            $this->lamp->switchOff();
        }
    }
}
```



```php
class Lamp implements SwitchableInterface
{
    private bool $isGlowing;

    public function switchOn() { /* ... */}
    public function switchOff() { /* ... */}
}
```

```php
interface SwitchableInterface
{
    public function switchOn();
    public function switchOff();
}
```

```php
class ArbitrarySwitch
{
    private SwitchableInterface $switchable;
    private bool $isActive;

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

    public function press()
    {
        // ...
    }
}
```

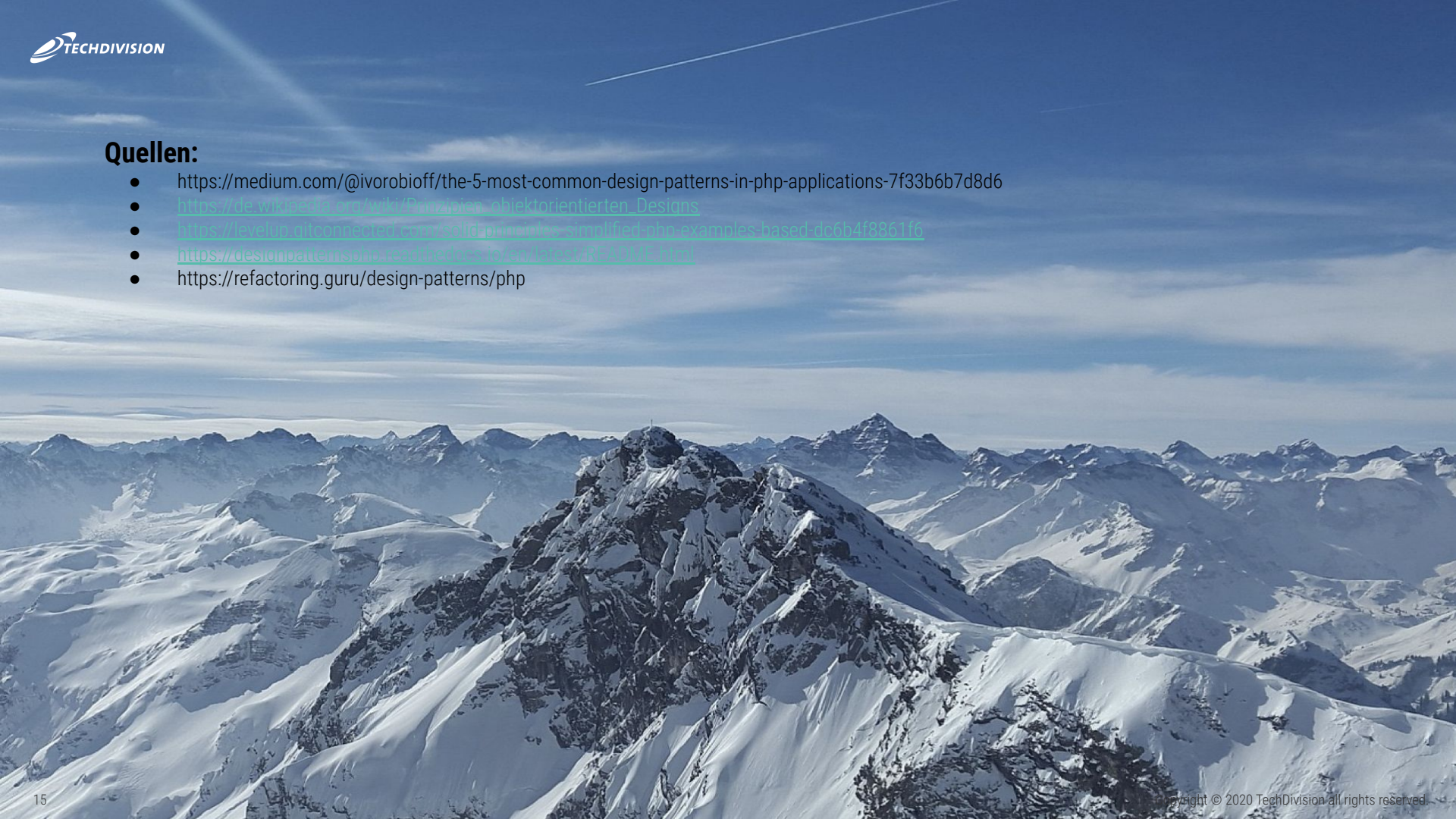
SOLID - Zusammenfassung

- Interfaces als Schnittstellen nutzen (z.B. in Typehints)
  - Häufig dort nutzen wo Erweiterung durch Dritte möglich ist
- Das *new* Keyword vermeiden und Abhängigkeiten von außen erhalten
  - Vergleiche Dependency Injection
- Zuständigkeiten von Klassen müssen klar und eng begrenzt sein
- Kleine und spezifische Interfaces definieren
- Vererbung sparsam und nur hierarchisch einsetzen
  - Logische Beziehung hinterfragen

Design Patterns

- Sammlung an Mustern zur Code Organisation
- Direkter Benefit durch die Erfahrung anderer
- Einige sind überall (z.B. Factory) andere sind sehr speziell (z.B. Memento)
- Begegnen einem in jeder Software (nicht nur im Web)
- Wichtig zu wissen: **Wann** nutze ich **welches** Pattern **wofür**?

- Gute Quelle: https://refactoring.guru/design-patterns/php



Quellen:

- https://medium.com/@ivorobioff/the-5-most-common-design-patterns-in-php-applications-7f33b6b7d8d6
- https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs
- https://levelup.gitconnected.com/solid-principles-simplified-php-examples-based-dc6b4f8861f6
- https://designpatternsphp.readthedocs.io/en/latest/README.html
- https://refactoring.guru/design-patterns/php

Bildquellen:

- Dependency Inversion Beispiel vermutlich von Sevenclev, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=1886669
- Chain of Responsibility Beispiel https://refactoring.guru/design-patterns/chain-of-responsibility