Wie halte ich mein Software-Design mithilfe der SOLID-Methode offen?

RENÉ GLASOW

Inhaltsverzeichnis

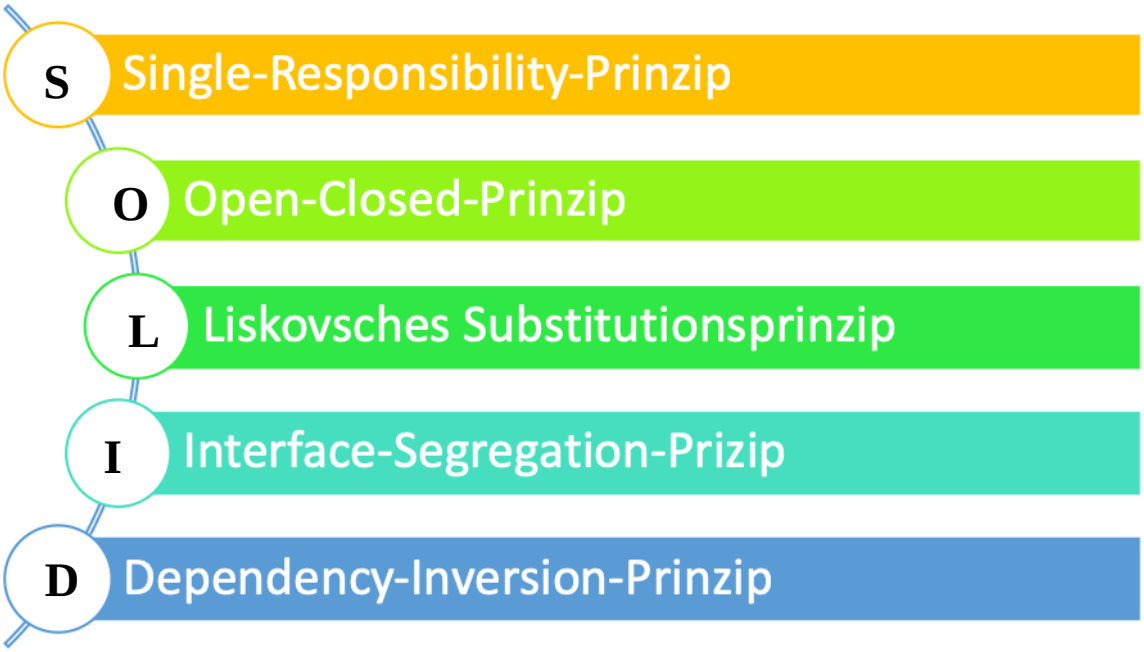
Einleitung

In diesem Leitfaden möchte ich die Möglichkeiten aufzeigen, wie man als Entwickler das Design für Veränderungen offenhalten kann, insbesondere um ohne größere Probleme auf Veränderung reagieren zu können.

Auf Themen wie „zentrale Config-Dateien“ vs. „hardcodierte Zeilen im Code“ möchte ich in diesem Leitfaden nicht weiter eingehen da dies allen bekannt sein sollte.

Stattdessen möchte ich zum Thema offenes Software-Design die „SOLID“-Methode näher erläutern.

SOLID ist ein Akronym, das fünf Prinzipien in sich vereint, die für bessere Codequalität im Allgemeinen sorgen:



Single-Responsible-Prinzip (SRP)

„There should never be more than one reason for a class to change."
– Robert C. Martin – Urvater des objektorientierten Designs

Das SRP (Single-Responsibility-Prinzip) sagt aus, dass eine Klasse nicht mehr als eine „Aufgabe" (Verantwortlichkeit) besitzen soll. (s. Kernaussage von R. C. Martin)

Das bedeutet im Umkehrschluss, dass bei bevorstehenden Änderungen nur die Klasse umcodiert werden muss, was wiederum bedeutet, dass die Fehlerquote deutlich sinkt, da nicht der gesamte Code geändert wird.

Open-Closed-Prinzip (OCP)

„Modules should be both open (for extension) and closed (for modification)."
- Bertrand Meyer -

Das OCP (Open-Closed-Prinzip) sagt aus, dass Klassen, Methoden, Module usw. so geschrieben werden sollen, dass es einfach ist, diese zu erweitern und zu entwickeln, ohne ihr Verhalten zu verändern.

Das OCP kann vornehmlich über zwei Wege erreicht werden:

1. Vererbung
2. Einsatz von Interfaces

Liskovsches Substitutionsprinzip

"Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) shpuld be true for objects of Type S where S is a subtype of T."
- Barbara Liskov –

Klingt kryptisch, ist es auch.

Einfacher gesagt sollen alle Objekte einer übergeordneten Klasse mit Objekten seiner Subklassen ersetzbar sein, ohne die Applikation zu zerstören. Gleiches gilt für die der Klasse innewohnenden Methoden und deren Return-Types.

Um es an einem noch einfacheren Beispiel zu verdeutlichen:

Gegeben sind zwei Klassen:

Bird und *Ostrich* (Vogel-Strauss)

Die Klasse *Bird* verfügt über die Methode "*fly*".

```
public class Bird {
        public function fly()
}
```

Ein *Ostrich* ist zwar ein *Bird*, kann aber nicht fliegen. Dementsprechend würden wir das LSP brechen, indem wir Folgendes tun:

```
public class Ostrich extends Bird {}
```

Oder noch einfach gesagt:

Interface Segregation Prinzip

„Clients should not be forced to implement interfaces they do not use.“
- Bertrand Meyer –

Auch dieses Prinzip lässt sich an einem einfachen Beispiel am besten verdeutlichen.

Nehmen wir an wir hätten ein Interface Animal, die die Methoden eat, sleep und walk enthält.

Auch wenn dieses Interface für unseren Code jetzt sinnvoll sein mag, kann es sein, dass wir später noch Tiere implementieren möchten, die nicht gehen (walk), sondern vielleicht schwimmen.

Damit wäre unser Interface nicht mehr geeignet, um es zum Beispiel von unseren zukünftigen Shark Klassen implementieren zu lassen.

Dependency Inversion Prinzip

„A. High level modules should not depend upon low level modules. Both should depend upon abstractions.
B. Abstractions should not depend upon details. Details should depend upon abstractions.“
- Robert C. Martin –

Nehmen wir an wir hätten eine Applikation, die Benutzer über verschiedene externe Services authentifiziert, z.B. GoogleAuthService, FacebookAuthService usw.

Möchten wir diese Services nun nutzen, könnten wir Code schreiben, der jede dieses Services direkt in unserem Code adaptiert. Dies könnte zu Probelem führen, wenn wir in Zukunft weitere Services hinzufügen möchte und wäre ein Verstoß gegen das Open-Close Prinzip.

Besser wäre es eine AuthentificationInterface zu schreiben und diesen die Methodensignatur authenticate(AuthService $authService) mitzugeben.

Hier könnten wir dann bei Bedarf den entsprechenden Service übergeben: authenticate(new GoogleAuthService)

Beispiele konkreter Umsetzung von offenem Design

Interfaces

Nicht jede Klasse muss zwangsweise Interfaces implementieren aber oft bietet es sich an, wenn es darum geht Code offen für Veränderungen zu halten. Interfaces sind Verträge, die mit Klassen geschlossen werden.

Zur Erinnerung: Interfaces enthalten nur die Methodensignatur, geben aber nicht vor wie die Klasse diese zu implementieren hat.

Klassenkonstanten sind auch in Interfaces erlaubt und machen so eventuell eine Klasse, die das Interface implementiert übersichtlicher.

Die Klasse, die das Interface implementiert, muss eine Methodensignatur verwenden, die gemäß LSP (Liskovsches Substitutionsprinzip) kompatibel ist. Andernfalls führt das zu einem fatalen Fehler.

DBAL/FSAL

Abstraktionschichten wie z.B. das Database Abstraction Layer oder das File System Abstraction Layer bieten Möglichkeiten den Code offen für Zugriffe durch andere als zunächst geplant zu halten.

Als Beispiel in unserem aktuellen Code können wir hier den Query Builder vom Doctrine DBAL anführen. Dieser kann Queries unabhängig vom dahinterstehenden SQL-Dialekt ausführen.

Dependency Injection

Dependency Injection, kurz DI, bezeichnet eine Entwurfsmuster, das wir bereits ständig einsetzen. Es regelt die Abhängigkeiten eines Objektes zur **Laufzeit.**

Mit Dependency Injection ist es möglich – entsprechend dem SRP – die Verantwortlichkeit für den Aufbau des Abhängigkeitsnetzes zwischen den Objekten eines Programmes aus den einzelnen Klassen in eine zentrale Komponente zu überführen.
Das Injecten selbst kann auf verschiedene Arten erreicht werden. Die Übergabe der Abhängigkeit mittels Konstruktor (Constructor Injection), mittels Interface (Interface Injection) oder mittels Setter (Setter Injection).

Symfony greift uns hier noch als Framework unter die Arme und auch Custom Injections z.B. über Factories sind möglich.

Eventlistener & -subscriber

Eventlistener und –subscriber ermöglichen es uns ähnlich wie schon aus JS bekannt auf bestimmt Events, hier vornehmlich Kernel-Events zu horchen und Code ausführen zu lassen.

Wir benutzen diese bereits nachdem das Logging in unserer API umgestellt wurde. Das Logging wird nun nicht mehr dort aufgerufen, wo auch eine Exception geworfen wird, sondern horcht nur noch auf bestimmte Events (onKernelException) und führt dann den Code für das Logging aus. Dadurch ist es besser wartbar und weitaus übersichtlicher und kann zudem noch bequem an weiteren Stellen eingefügt werden.

```php
class ExceptionSubscriber implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        // return the subscribed events, their methods and priorities
        return [
            KernelEvents::EXCEPTION => [
                ['processException', 10],
                ['logException', 0],
                ['notifyException', -10],
            ]
        ];
    }

    public function processException(GetResponseForExceptionEvent $event)
    {
        // ...
    }

    public function logException(GetResponseForExceptionEvent $event)
    {
```

Eventsubscriber (s.o.) sind Klassen, die eine oder mehrere Methoden enthalten und die auf eine oder auch mehrere Events horchen. Innerhalb der Methoden kann die Reihefolge der Ausführung der Methoden entsprechend priorisiert werden.

Weiterführende Quellen

https://symfony.com/doc/current/event_dispatcher.html
https://stackify.com/solid-design-liskov-substitution-principle/
https://de.wikipedia.org/wiki/Dependency_Injection
https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design