# Was sind Prinzipien?

Prinzipien sind keine festen Regeln. Entwickler werden dafür bezahlt Probleme zu lösen und nicht dafür Prinzipien einzuhalten. Prinzipien entstehen aber aus Erfahrungswerten, gegen die man nicht leichtfertig verstoßen sollte.

Wenn die Einhaltung von Prinzipien die Erfüllung von Anforderungen verhindern, sollte man sich gegen dieses Prinzip entscheiden. Begrenzte Ressourcen bei Hardware naher Programmierung oder Zwang zu hoher Performance kann ein gutes Argument sein, bewusst gegen eines oder mehrerer Prinzipien zu Verstoßen. Aber dies sollte man dann bewusst dokumentieren. Es gibt auch Prinzipien, die sich widersprechen.

Häufig ist es aber Bequemlichkeit, mangelndes Qualitätsbewusstsein oder eine unterschiedliche Auffassung was Qualität eigentlich ausmacht der Hauptgrund gegen diese Prinzipien zu Verstoßen. In der Software Entwicklung ist es leider meist so, dass die schnellen kostengünstigen Lösungen am Ende zu schwer wartbarem Code und Dokumente führt.

# Grundlegende Entwurfsprinzipien

## Einfachheit und Verständlichkeit – Das KISS Prinzip

KISS steht für Keep it Small and Simple. Schlichtheit und Einfachheit erleichtern allen Personen, die Ihre Entwürfe später lesen, das Verständnis. Sie vermeiden, dass potenzielle Probleme durch überflüssige Komplexität verdeckt werden. Einfachheit hilft, Probleme bereits im Vorfeld zu erkennen.

Einfache Lösungen sind leichter wartbarer. Sie können einfache Lösungen leichter implementieren

und der Quellcode wird kurzer und verständlicher sein.

Quelle Effektive Softwarearchitekturen 8. Auflage

## Vermeide Redundanz DRY (Don’t repeat yourself)

Vermeiden Sie Redundanz, halten Sie alle Informationen in oder über Ihr System möglichst, an einer einzigen Stelle. Dieses Prinzip ist auch bekannt als „*Single Source of Truth*“.

DRY fordert Sie zur Wiederverwendung auf: Faktorisieren Sie mehrfach benötigte Codestellen

heraus, definieren oder beschreiben Sie Dinge an nur einer Stelle.

Wenn Sie dieses Prinzip durchgängig anwenden, müssen Sie bei Änderungen oder Fehlerkorrekturen

nur an dieser einen Stelle andern – theoretisch zumindest. In der Realität stößt DRY schnell an Grenzen – wenn Sie beispielsweise in Ihrer Dokumentation eine Bausteinsicht beschreiben – aber später die Bezeichner der Bausteine im Quellcode andern. Sofern Sie nicht Ihren Code aus dem Modell heraus generieren, müssen Sie in solchen Fällen an mehreren Stellen Änderungen vornehmen.

Vorsicht: Ein klein wenig Redundanz kann den Komfort oder die Verständlichkeit erhöhen.

Sie mochten als Leser einer (Architektur-)Dokumentation lieber einen in sich geschlossenen Text lesen, statt dauernd in referenzierten Dokumenten nachblättern zu müssen.

Quelle Effektive Softwarearchitekturen 8. Auflage

## Abstraktion, Kapselung und das Geheimnisprinzip

Trennen Sie „Was“ vom „Wie“ bei der Implementierung von Bausteinen. Verbergen (kapseln) Sie das Innenleben Ihrer Bausteine vor der Außenwelt. Definieren Sie für Ihre Bausteine eine öffentliche Schnittstelle, die beschreibt „was“ der Baustein leistet.

Interne Entwurfs- oder Implementierungsentscheidungen, das „Wie“, halten Sie vor äußerem

Zugriff verborgen. Wenn Sie die Kapselung oder das Geheimnisprinzip missachten,

entstehen ungewollte und schwierige Abhängigkeiten zwischen den einzelnen Systemteilen (Subsystemen, Komponenten), die insgesamt zu einer höheren Komplexität fuhren. Ohne Kapselung erschwert eine Zerlegung des Systems das Problem, statt es zu vereinfachen (denn: was bekannt ist, wird auch ausgenutzt!).

Quelle Effektive Softwarearchitekturen 8. Auflage

## Konsistenz und konzeptionelle Integrität

Der Begriff *Konsistenz* bedeutet in unserem Kontext Stimmigkeit, Geschlossenheit oder

Zusammenhalt.

Konzeptionelle Integrität (conceptual integrity7) ist eine wesentliche Eigenschaft guter Entwurfe:

Ähnliche Probleme sollten ähnlich gelöst werden, ein System soll eine kleine Menge von Entwurfs- oder Lösungsansätzen konsistent verfolgen. Entwurfsentscheidungen sollten

innerhalb des Systems *stimmig* getroffen werden – zueinander passend.

Das C2Wiki8 nennt einige interessante Beispiele konzeptioneller Integrität:

* Unix: Das gesamte System basiert durchgängig auf der zentralen Abstraktion „File“:

Directorys, Devices, Sockets repräsentieren so etwas wie Files.

* SQL: Alle Daten liegen in Tabellen – mit Schlusselbeziehungen untereinander.
* Lisp: Alles ist eine Liste.

Für mich zahlen zu diesem Prinzip kleine wie große Entscheidungen. Unter anderem finde ich

konsistenten Programmierstil relevant, aber auch der konsistente Einsatz von Bibliotheken,

die konsistente Anwendung von Entwurfsmustern oder Losungskonzepten.

# Kohäsion und Kopplung

Eine gute Softwarearchitektur zeichnet sich durch eine lose Kopplung zwischen den Modulen (Paketen) und eine hohe Kohäsion innerhalb der Module (Pakete) aus.

## Verschiedene Arten von Kopplung

* durch Aufruf
* durch Erzeugung
* durch Daten (globale Datenstrukturen oder Methodenparameter)
* Ausführungsort (kommunizierende Bausteine müssen sich im selben Adressraum befinden)
* durch die Zeit. Reihenfolge von Aktivitäten in verschiedenen Bausteinen spielt eine Rolle
* durch Vererbung. Unterklasse erbt Eigenschaften von der Oberklasse.

## Vorteile loser Kopplung

Durch die lose Kopplung soll die Komplexität von Strukturen geringgehalten werden. Je geringer die Kopplung zwischen mehreren Bausteinen, desto leichter ist es, einen Baustein zu verstehen, ohne sich dabei viele andere Bausteine anschauen zu müssen. Ein weiterer Punkt ist die Änderbarkeit. Je geringer die Kopplung, desto einfacher können lokale Änderungen an einzelnen Bausteinen durchgeführt werden, ohne dabei andere Bausteine betrachten zu müssen.

Quelle Effektive Softwarearchitekturen 8. Auflage Kapitel 3.5.2

## Hohe Kohäsion

Die Kohäsion wird auch Zusammenhangskraft genannt. Der Begriff stammt vom lateinischen cohaerere ab, was so viel wie »zusammenhängen« bedeutet.

Das Prinzip der losen Kopplung führt oft zum Prinzip der hohen Kohäsion. Denn hält man die Kopplung zwischen den Bausteinen gering, führt dies oft dazu, dass die Bausteine im Inneren stärker zusammenhängend entworfen werden.

Eine kohärente Klasse löst ein einziges Problem und besitzt eine spezifische Menge an stark zusammenhängenden Funktionen. Je höher die Kohäsion, desto stärker zusammenhängend ist die Zuständigkeit einer Klasse in der Anwendung.

Auch hier geht es wieder um die lokale Änderbarkeit und Verständlichkeit von

Systembausteinen. Wenn ein Systembaustein alle Eigenschaften, die zum Verstehen

und Ändern benötigt werden, vereint, kann man ihn dementsprechend auch leichter

ändern, ohne andere Systembausteine in Anspruch nehmen zu müssen.

In Paketen sollten Sie nicht alle Klassen desselben Typs gruppieren (z.B. alle

Filter, alle Entitäten), sondern stattdessen nach Systemen und Subsystemen gruppieren.

Kohärente Pakete beherbergen Klassen eines zusammenhängenden Funktionskomplexes.

Quelle Effektive Softwarearchitekturen 8. Auflage Kapitel 3.5.3

# Solid Prinzipien

Ich habe in meiner privaten Lernplattform [www.cocodil.org](http://www.cocodil.org) ein längeres Lernvideo über die Solid Prinzipien veröffentlicht (shttp://meucht.de/klaus/cocodil/Solid/solid.html). Aber da der Zweck des Dokuments ist, das Wesentliche kurz zusammenzufassen, möchte

SOLID steht für 5 Prinzipien

* Single Responsibility Principle (SRP)
* Open Closed Principle (OCP)
* Listkow Substitution Principle (LSP)
* Interface Segregation Principle (ISP)
* Dependency Inversion Principle (DIP)

## Single Responsibility Principle (SRP)

### Definition

Das SRP besagt, dass es nur einen einzigen Grund geben darf, eine Klasse zu ändern. Es führt i.a. zu sehr kleinen Klassen, mit einem hohen Grad an Wiederverwendbarkeit.

### Unterschied zu Separation of Concerns

Robert Cecile Martin hat ein Beispiel gebracht, das Ich als sehr grenzwertig empfand. Er hat eine Verdoppelung des Codes für die Berechnung der geleisteten Arbeitsstunden zugelassen, abhängig davon ob die Personalabteilung oder die Lohnbuchhaltung diese verwendet.

Ist es von den Anforderungen richtig, dass verschiedene Stakeholder unterschiedliche Sichtweisen auf dieselbe Funktionalität haben – dann ist eine Verdoppelung des Codes sicherlich richtig. Der Wunsch nach einer Veränderung der Berechnung in der Lohnbuchhaltung, darf nicht zu einer Veränderung im Personalwesen führen.

Es ist ein Unterschied ob eine Funktionalität nur zufällig gleich ist, und durchaus auseinanderlaufen darf – oder ob Anforderungen unterschiedlicher Stakeholder abgestimmt werden müssen.

Im ersteren Fall ist eine Verdoppelung des Codes und eine Verletzung des DRY Prinzips (Don’t repeat yourself) sinnvoll. Das ist aber dann etwas, was zwingend in der Architekturdokumentation erläutert werden sollte. Es ist ein typisches Beispiel dafür, wie wichtig es ist, dass Entwickler und Stakeholder eine gemeinsame Sichtweise auf das Produkt haben.

## Das Open Closed Principle

### Definition

Das Open Closed Prinzip besagt, dass ein Modul geschlossen gegenüber Änderungen, aber offen für Erweiterungen sein sollte. I.a. wird die Vererbung genommen, um neue Funktionalität hinzuzufügen.

### Kritik

Ich teile die Meinung der Kritiker, dass dieses Prinzip mehr Schaden als Nutzen bringt. Das Open Closed Prinzip stammt aus Zeiten in denen Mehrfachvererbung die Regel waren.

Der Wesentliche Vorteil der Objektorientiertheit ist die Kapselung. Vererbung ist zwar ein typisches Merkmal der Objektorientiertheit – wirkt aber der Kapselung entgegen. Vererbungsbeziehungen sind zur Laufzeit nicht änderbar. Deshalb verzichten fast alle neuen objektorientierten Sprachen auf die Möglichkeit der Mehrfachvererbung.

Ich halte das Prinzip gefährlich, weil man die Änderung von Software als Niederlage empfindet. Das Wesentliche an Software ist aber die Änderbarkeit. Ändern sich Anforderungen oder Rahmenbedingungen sollte man auch den Mut haben die Software zu ändern. Softwareentwicklung ist ein Lernprozess. Bei dem enormen technologischen Wandel und dem Einsatz von Software, ändern sich die Sichtweise wie Software einzusetzen ist.

## Das Listkov Substitution Prinzip

### Definition

Das Liskovsche Substitutionsprinzip (LSP) oder *Ersetzbarkeitsprinzip* fordert, dass eine Instanz einer [**abgeleiteten Klasse**](https://de.wikipedia.org/wiki/Abgeleitete_Klasse) sich so verhalten muss, dass jemand, der meint, ein Objekt der [**Basisklasse**](https://de.wikipedia.org/wiki/Basisklasse)vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird, wenn es sich dabei tatsächlich um ein Objekt eines Subtyps handelt.

„Sei **q(x)**  eine Eigenschaft des Objektes**x** {\displaystyle x}x xvom Typ **T** {\displaystyle T}T, dann sollte **q(y)** {\displaystyle q(y)}q(y) qqqqfür alle Objekte y {\displaystyle y}qqqqqqqqq des Typs **{\displaystyle S} T** gelten, wobei **S** {\displaystyle S}ein Subtyp von **T**{\displaystyle T}T TTT  ist.“

Eine Methode einer Klasse, sollte auch für die Instanzen all seiner Unterklassen gelten.

### Beispiel

Probleme gibt es immer falls innerhalb einer Vererbung die Unterklasse eine Eigenschaft wegnimmt, statt eine Eigenschaft nur hinzufügt. Ein typisches Beispiel ist die Vererbung zwischen Quadrat als Unterklasse von Rechteck (Die zweite Seite fällt weg), oder ein Elektroauto als Unterklasse von Auto (Ein Elektroauto hat kein Getriebe).

## Interface Segregation Principle (ISP)

### Definition

Demnach sollen zu großen Schnittstellen in mehrere Schnittstellen aufgeteilt werden, falls implementierende Klassen unnötige Methoden haben müssen. Nach erfolgreicher Anwendung dieses Entwurf Prinzips müsste ein Modul, das eine Schnittstelle benutzt, nur diejenigen Methoden implementieren, die es auch wirklich braucht.

### Erklärung

Abbildung Beispiel problematischer Schnittstellen

Abbildung Passgenaue Schnittstellen

## Dependency Inversion Principle (DIP)

### Definition

Das Prinzip der Umkehr der Abhängigkeiten (engl. Dependency Inversion) besagt,

dass Sie keine direkten Abhängigkeiten, sondern nur Abhängigkeiten von Abstraktionen

erlauben sollten. Das führt schließlich dazu, dass die Austauschbarkeit

von Bausteinen erleichtert wird. Direkte Abhängigkeiten zwischen Klassen sollten.

# Umkehr von Abhängigkeiten

Das folgende Bild zeigt wie zyklische Abhängigkeiten aufgelöst werden können. Ich habe ein entsprechendes Bild aus dem Buch „Effektive Softwarearchitekten“ 8. Auflage orientiert, aber die Auflösung etwas deutlicher dargestellt.

Abbildung Zyklische Abhängigkeiten auflösen.

Nach dem DIP sollten Klassen nie konkret miteinander kommunizieren, sondern nur über Interfaces oder abstrakte Klassen.