# SOLID - Design Prinzipien

## Single-Responsibility-Prinzip

*„Es sollte nie mehr als einen Grund dafür geben, eine Klasse zu ändern.“ Robert C. Martin*

- Dieses Prinzip besagt, dass jede **Komponente nur für eine Aufgabe verantwortlich** sein und auch die gesamte Funktionalität einer Aufgabe abdecken sollte.
- Werden **Änderungen** an der Software vorgenommen, sollte es einen **Grund** dafür geben.
- In der Praxis hat eine Komponente dann die richtige Größe, wenn sich die **Änderungen** aufgrund einer User Story **nur auf eine Komponente auswirken**.
- Die **Kernaussage** des Prinzips ist, dass **jede Klasse nur genau eine fest definierte Aufgabe** zu **erfüllen** hat. Wenn eine Klasse mehrere Verantwortungen zu tragen hat, führt das zu Schwierigkeiten bei zukünftigen Änderungen und das Fehlerrisiko steigt.
- Eine **hohe Kohäsion** – alle Methoden innerhalb einer Klassen haben einen starken gemeinsamen Bezug – **sollte angestrebt werden**.

## Open-Closed-Prinzip

*„Module sollten sowohl offen (für Erweiterungen) als auch geschlossen (für Modifikationen) sein.“ Bertrand Meyer*

- Dieses Prinzip besagt, dass **Klassen**, **Methoden**, Module et cetera so entwickelt werden sollen, dass sie **einfach** zu **erweitern** sind – **ohne** ihr **Verhalten zu ändern**.
- Ein Beispiel ist die **Vererbung**. Das Verhalten einer Klasse wird nicht verändert, erhöht die Funktionalität der Software. Das Überschreiben von Methoden verändert nicht das Verhalten der Basisklasse, ausschließlich die Methoden der abgeleiteten Klasse.

## Liskovsches Substitutionsprinzip

*„Sei q(x) eine Eigenschaft des Objektes x vom Typ T, dann sollte q(y) für alle Objekte y des Typs S gelten, wobei S ein Subtyp von T ist.“ Barbara Liskov*

- **Abgeleitete Klasse muss im Kontext ihrer Basisklasse eingesetzt werden können**. Sie darf das **Verhalten der Basisklasse nur erweitern**, aber **nicht einschränken**.
- Das Liskovsches Substitutionsprinzip besagt, dass jede **Basisklasse immer durch ihre abgeleiteten Klassen (Unterklassen) ersetzbar** sein soll.
- Eine Methode, die ein Objekt vom Typ der Basisklasse erwartet, soll auch korrekt funktionieren, wenn ein Objekt der Unterklasse übergeben wird.
- Eine **Subklasse** soll **alle Eigenschaften der Superklasse erfüllen** und **als Objekt der Superklasse verwendbar** sein.
- Eine Subklasse darf Erweiterungen enthalten, aber keine grundlegenden Änderungen.

## Interface-Segregation-Prinzip

*„Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden.“ Robert C. Martin*

- Der **Umfang** eines **Interface**s wird **durch** die **Anforderungen** des **Client bestimmt** und nicht umgekehrt.
- **Vermeidung** umfangreicher **Universalschnittstellen**.
- Modellierung der **Verantwortungsbereiche** einzelner **Komponenten in** jeweils einer **eigene**n **Schnittstelle**.
- Ein Client darf nicht gezwungen werden, Funktionalität zu implementieren, die gar nicht benötigt wird. Damit wird der Zusammenhalt von Modulen gestärkt, deren Kopplung jedoch reduziert.

## Dependency-Inversion-Prinzip

*„A. Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen. B. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.“ Robert C. Martin*

- In einer geschichteten Architektur existieren Klassen und **Module auf unterschiedlichen Abstraktionsebenen**.
- Dependency-Inversion-Prinzip besagt, dass **Module auf höheren Abstraktionsebenen nicht von Modulen niedrigerer Abstraktionsebenen abhängen**, sondern sich jeweils auf Interfaces beziehen sollten. Interfaces hingegen sollten nicht von Details abhängen, sondern die Details müssen eine Abhängigkeit von den jeweiligen Interfaces aufweisen.
- **Abhängigkeit nur von Abstraktionen und nicht von konkreten Implementierungen**.
- Die Abhängigkeit von Abstraktionen ist eine effektive Methode zur **Steigerung** der **Flexibilität** und **Erweiterbarkeit**.