Kapitel 2: Prinzipien des objektorientierten Designs (SOLID)

1. Motivation und Code-Qualität

In der objektorientierten Programmierung (OOP) führt unstrukturierter Code schnell zu "Software-Rot" (Code Smells), hoher Kopplung und mangelnder Testbarkeit. Die fünf **SOLID-Prinzipien** bieten einen Leitfaden, um Architekturen flexibel, erweiterbar und langfristig wartbar zu gestalten.

2. S - Single Responsibility Principle (SRP)

Das SRP besagt, dass eine Klasse oder ein Modul exakt eine einzige Verantwortung besitzen darf. Oder treffender formuliert: *"Es darf für eine Klasse nur genau einen einzigen Grund geben, sich zu ändern."*

Wenn eine Klasse beispielsweise sowohl die mathematische Berechnung eines Preises durchführt als auch die Formatierung dieses Preises als HTML-Bericht übernimmt, verletzt sie das SRP. Ändert sich das Rechnungs-Layout, muss die Rechenklasse modifiziert werden. Dies erhöht das Risiko, ungewollt Fehler in die Berechnungslogik einzubauen. Die Lösung ist die Separation in zwei getrennte Klassen (z. B. `PriceCalculator` und `InvoiceFormatter`).

3. O - Open-Closed Principle (OCP)

Das OCP fordert: *"Software-Entitäten sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein."*

Das bedeutet, dass das Verhalten einer bestehenden Softwarekomponente erweitert werden kann, ohne dass deren bereits getesteter Quellcode direkt editiert werden muss. Erreicht wird dies konsequent durch den Einsatz von Abstraktionen (Interfaces oder abstrakten Klassen) und Polymorphie.

Betrachten wir folgendes konzeptionelles Negativbeispiel:

```
// Verstoß gegen das OCP
if (form == "Kreis") { zeichneKreis(); }
else if (form == "Rechteck") { zeichneRechteck(); }
```

Kommt eine neue Form (z. B. "Dreieck") hinzu, muss diese bestehende Methode modifiziert werden. Gemäß OCP definiert man stattdessen ein Interface `Shape` mit einer abstrakten Methode `draw()`. Jede konkrete Form implementiert dieses Interface selbstständig. Neue Formen können als neue Klassen hinzugefügt werden, ohne eine Zeile alten Codes anzufassen.

4. Die weiteren Prinzipien im kompakten Überblick

- **L - Liskov Substitution Principle (LSP):** Subtypen müssen sich so verhalten, dass sie jederzeit durch ihre Basistypen ersetzt werden können, ohne das Programmverhalten zu korrumpieren.

- **I - Interface Segregation Principle (ISP):** Clients sollten nicht gezwungen werden, von Interfaces abzuhängen, deren Methoden sie überhaupt nicht nutzen. Lieber viele spezifische Schnittstellen als eine fette "Eierlegende Wollmilchsau"-Schnittstelle.
- **D - Dependency Inversion Principle (DIP):** Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Zudem sollten Abstraktionen nicht von Details abhängen, sondern Details von Abstraktionen.