

Programmierkonzepte in der Physikalischen Chemie

24. Open-Closed-Prinzip

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2018/19



- 🔑 Softwareeinheiten sollten offen für Erweiterungen, aber verschlossen gegenüber Veränderungen sein.
- 🔑 Schlüssel des Prinzips sind zwei Kernaspekte der OOP: Vererbung und Polymorphie (zusammen mit Abstraktion).
- 🔑 Entwickelt werden sollte immer nur gegen (abstrakte) Schnittstellen.
- 🔑 Verschlossenheit gegenüber Veränderung ist nie vollständig.
- 🔑 Kern objektorientierten Entwurfs:
Führt zu Flexibilität, Wiederverwendbarkeit, Wartbarkeit.

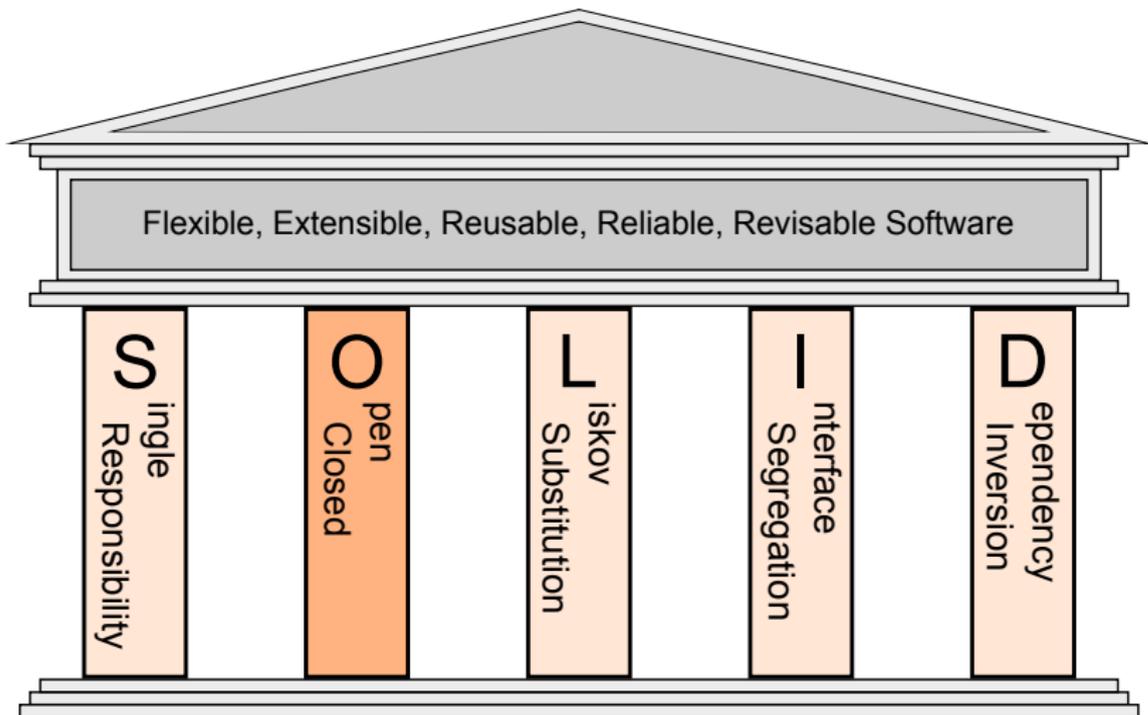
Das Open-Closed-Prinzip

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

Das Open-Closed-Prinzip

Übersicht über die fünf Prinzipien



“ *A software artifact should be open for extension,
but closed for modification.*

– Robert C. Martin

- ▶ Klingt zunächst wie ein Widerspruch.
 - Normalerweise bedeutet Erweiterung auch Änderung.
 - Idee: Erweiterung ohne Modifikation der Schnittstelle
 - mit nicht-objektorientierten Techniken nicht realisierbar
- ▶ Lösung: Vererbung und Polymorphie in Verbindung mit Abstraktion
 - Eine Klasse hängt nur von abstrakten Klassen ab.
 - Objekte dieser Klasse nutzen Objekte konkreter Klassen, die von der abstrakten Klasse erben.

Robert C. Martin: Clean Architecture. Prentice Hall, Boston 2018, S. 70

Original: Bertrand Meyer: Object-Oriented Software Construction, Prentice Hall, Upper Saddle River 1997, S. 57

Vererbung (*inheritance*)

Eine Klasse kann von einer (abstrakten) Klasse abgeleitet werden und erbt deren Attribute und Methoden.

Polymorphie (*polymorphism*)

Ähnliche Objekte können auf die gleiche Botschaft (den Aufruf einer gleichnamigen Methode) in unterschiedlicher Weise reagieren („Vielgestaltigkeit“).

-  Das dritte Grundprinzip der OOP (Kapselung) wird noch beim Interface-Segregation-Prinzip wichtig werden.

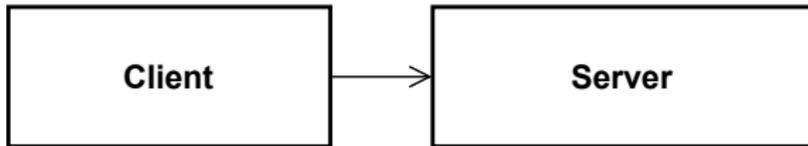
abstrakte Klasse (*abstract class*)

Klasse, die zunächst einmal nur eine Schnittstelle liefert und nur (abstrakte) Methoden ohne Implementierung enthält.

- ▶ konkrete, abgeleitete Klassen erben von abstrakter Klasse
 - implementieren die zunächst abstrakten Methoden
- ▶ Zahl der abgeleiteten Klassen prinzipiell unbegrenzt
 - ermöglicht die beliebige *Erweiterung* der Elternklasse, ohne ihr Verhalten zu verändern
- ☞ Dieser Einsatz von Vererbung wird von manchen Autoren als ihr einzig richtiger Einsatz bezeichnet.

Beispiele für seinen Einsatz

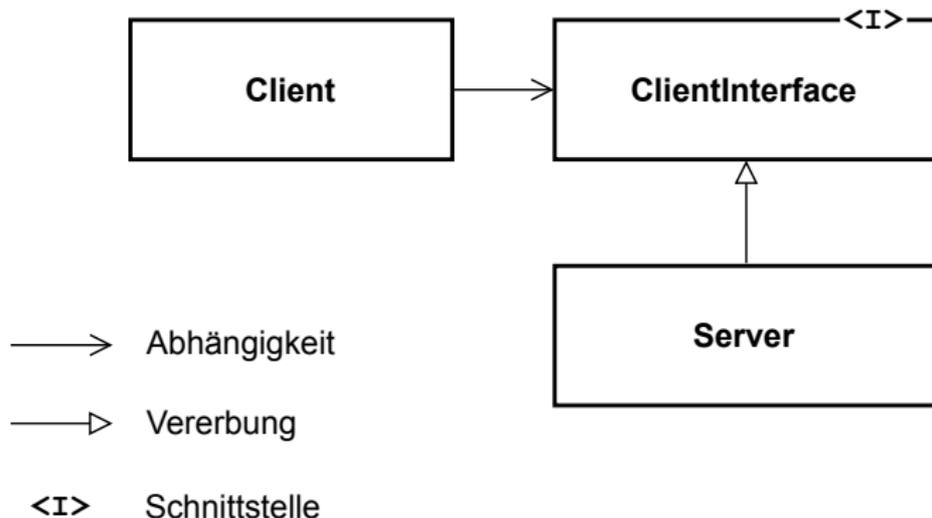
Das abstrakteste Beispiel: Zusammenspiel von Server und Client



Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 100.

Beispiele für seinen Einsatz

Das abstrakteste Beispiel: Zusammenspiel von Server und Client

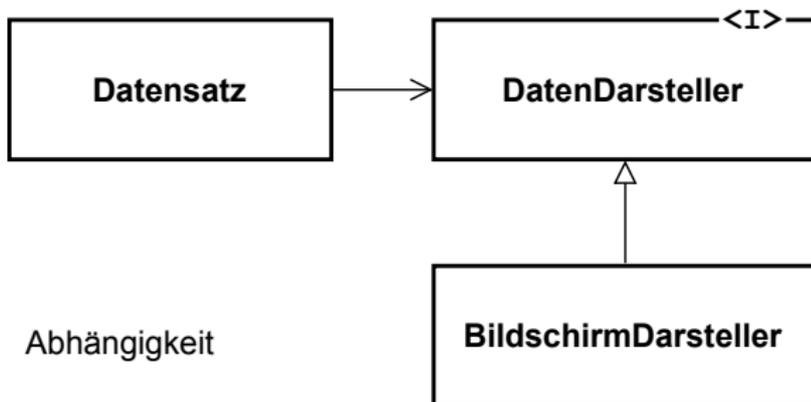


Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 101.

- ▶ naive Implementierung: Verstoß gegen das Prinzip
 - Client nutzt direkt Server.
 - Soll Server ausgetauscht werden, muss Client geändert werden.
 - Kontrollfluss erzwingt Abhängigkeit in gleicher Richtung.
- ▶ Lösung im Einklang mit dem Prinzip
 - Client und Server sind entkoppelt.
 - Server kann ausgetauscht werden, ohne dass Client etwas davon mitbekommt.
 - Client und ClientInterface befinden sich in derselben Komponente, Server in einer anderen.
- ☞ Grundregel: Immer gegen (abstrakte) Schnittstellen, nicht gegen (konkrete) Implementierungen programmieren.

Beispiele für seinen Einsatz

Ein praxisnäheres Beispiel: Darstellung der Daten eines Datensatzes



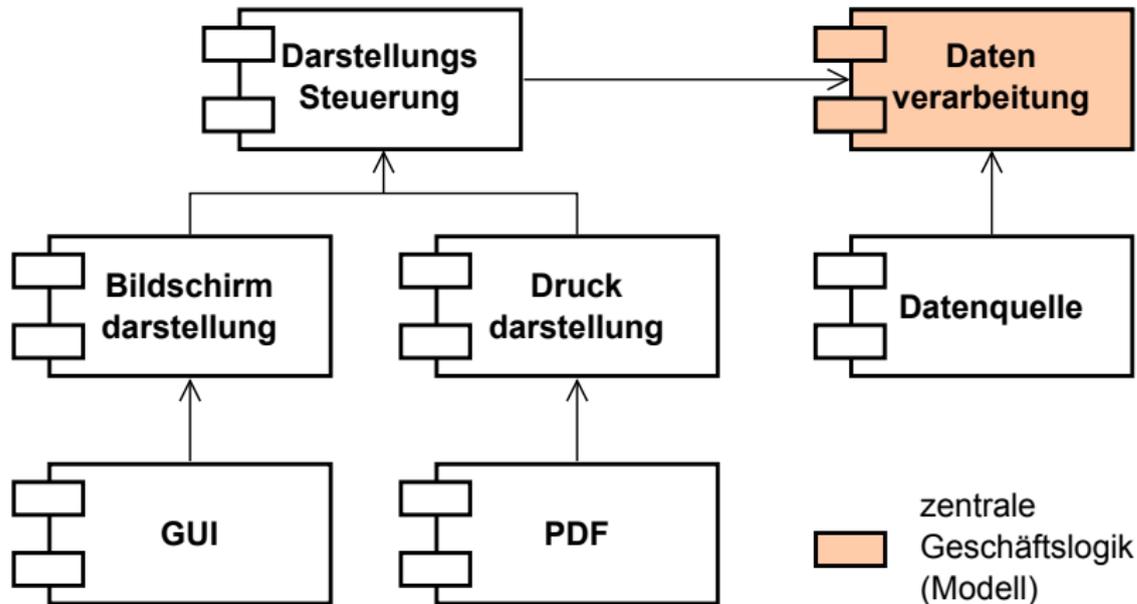
—> Abhängigkeit

—▷ Vererbung

<I> Schnittstelle

Beispiele für seinen Einsatz

Darf's auch etwas komplizierter sein? OCP auf Komponenten-Ebene



- ▶ zentral für gute objektorientierte Software-Architektur
 - Schlüssel zum Erreichen der Versprechen objektorientierter Programmierung
 - führt zu Flexibilität, Wiederverwendbarkeit, Wartbarkeit
- ▶ Es reicht nicht, objektorientiert zu programmieren.
 - Der richtige und bewusste Einsatz der zur Verfügung stehenden Werkzeuge ist entscheidend.
- ▶ Übermäßiger Einsatz ist kontraproduktiv.
 - Voreiliger Abstraktion zu widerstehen ist genauso wichtig wie Abstraktion selbst.
 - Anwendung nur bei solchen Teilen des Programms, die sich häufig ändern.

Zusammenspiel mit den anderen Prinzipien

Verbindung zu allen vier anderen SOLID-Prinzipien

- ▶ SRP
 - Aufteilung in Komponenten nach Verantwortlichkeiten und damit nach Gründen, sich zu ändern
- ▶ LSP
 - Kriterien für Vererbung, die Polymorphie und damit die Austauschbarkeit abgeleiteter Klassen ermöglichen
- ▶ ISP
 - Kapselung verhindert transitive Abhängigkeiten über Architekturgrenzen hinweg
- ▶ DIP
 - Trennung von Kontrollfluss und Quellcode-Abhängigkeiten, Umkehr der Quellcode-Abhängigkeit

“ *A component should be as abstract as it is stable.*

– Robert C. Martin

- ▶ Abstraktion ermöglicht Erweiterung ohne Veränderung.
 - Schlüsselerkenntnis, die im OCP formuliert ist.
 - Der Grad der Abstraktion einer Systemkomponente und ihre Stabilität stehen in einem klaren Verhältnis.
- ▶ Metriken für Abstraktion
 - erlauben die Quantifizierung und Analyse von Systemen
 - Details u.a. in Robert C. Martin: Clean Architecture (a.a.u.)



- ❏ Softwareeinheiten sollten offen für Erweiterungen, aber verschlossen gegenüber Veränderungen sein.
- ❏ Schlüssel des Prinzips sind zwei Kernaspekte der OOP: Vererbung und Polymorphie (zusammen mit Abstraktion).
- ❏ Entwickelt werden sollte immer nur gegen (abstrakte) Schnittstellen.
- ❏ Verschlossenheit gegenüber Veränderung ist nie vollständig.
- ❏ Kern objektorientierten Entwurfs:
Führt zu Flexibilität, Wiederverwendbarkeit, Wartbarkeit.