

# Wissenschaftliche Softwareentwicklung

## 25. Dependency-Inversion-Prinzip

Till Biskup

Physikalische Chemie

Universität Rostock

05.01.2024





- Die Kernaspekte einer Anwendung sollten nicht von ihrer Peripherie abhängen – sondern beide von Abstraktionen.
- Anwendungsentwicklung zielt auf die zugrundeliegenden Abstraktionen und ihre stabilen, abstrakten Schnittstellen.
- Anwendungen bestehen aus klar getrennten Schichten, die Dienste über definierte Schnittstellen bereitstellen.
- Die Anwendungslogik steht im Zentrum.  
Datenquellen und Nutzerschnittstellen sind peripher.
- Umkehr der Abhängigkeiten sorgt für Unabhängigkeit und intrinsische Testbarkeit der einzelnen Schichten.

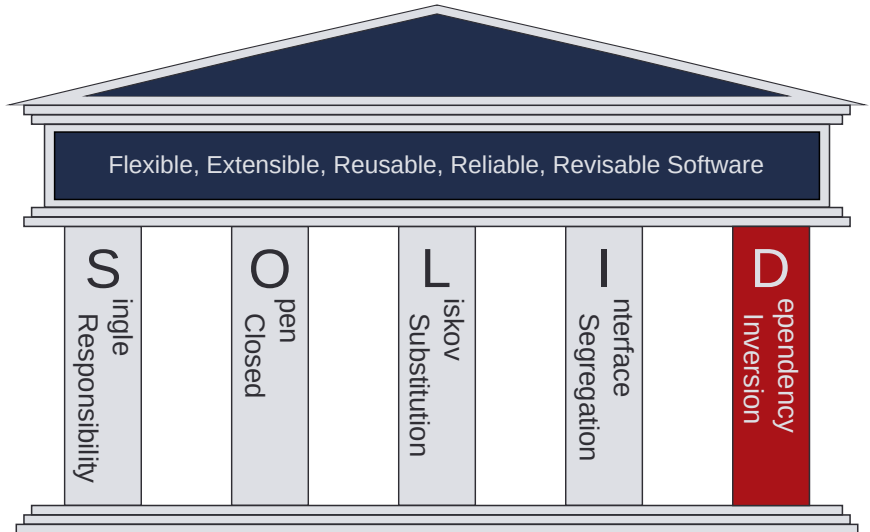
Das Dependency-Inversion-Prinzip

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

# Das Dependency-Inversion-Prinzip

Übersicht über die fünf Prinzipien



- “ a. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
- b. *Abstractions should not depend on details. Details should depend on abstractions.*

– Robert C. Martin

- Abstraktion ist der Kern jeder Softwareentwicklung.
  - Formalisierung von Zusammenhängen und Abläufen
- Abstraktionen bilden den Kern jedes guten Programms.
  - Geschäftsregeln (*business rules*), Modell der Realität
  - sollten unabhängig von Details der Implementierung sein
  - vgl. die Grundidee des „*Domain-Driven Design*“

- Das DIP ist keine sklavisch befolgbare Regel.
    - Abhängigkeiten von konkreten Klassen sind unvermeidlich.
    - Stabile konkrete Klassen sind nicht das Problem.
    - Bsp.: Basistypen einer Sprache („stabiler Hintergrund“)
  - Softwareentwicklung bedeutet ständige Veränderung.
    - Abhängigkeiten von unbeständigen Elementen vermeiden
    - Entkopplung durch abstrakte Schnittstellen
  - Schnittstellen sind stabiler als ihre Implementierungen.
    - Änderungen der Schnittstellen erzwingen Änderungen aufseiten der Nutzer der Schnittstellen.
    - Änderungen der Implementierung einer Schnittstelle haben meist keinen Einfluss auf die Schnittstelle selbst.
- ☛ Gute Architektur nutzt stabile, abstrakte Schnittstellen.

- nicht auf unbeständige konkrete Klassen verweisen
  - stattdessen abstrakte Schnittstellen verwenden
  - erzwingt meist die Nutzung von *abstract factories*
- nicht von unbeständigen konkreten Klassen erben
  - Vererbung erzeugt direkte Abhängigkeiten im Quellcode.
- konkrete Funktionen nicht außer Kraft setzen
  - Lösung: Abstraktion und Polymorphie (vgl. OCP)
- Namen konkreter, unbeständiger Elemente nicht nennen
- ☛ Objekterzeugung führt zu Quellcode-Abhängigkeiten.
- ☛ Konkrete, unbeständige Objekte bedürfen spezieller Mechanismen, um Abhängigkeiten zu invertieren.

Das Dependency-Inversion-Prinzip

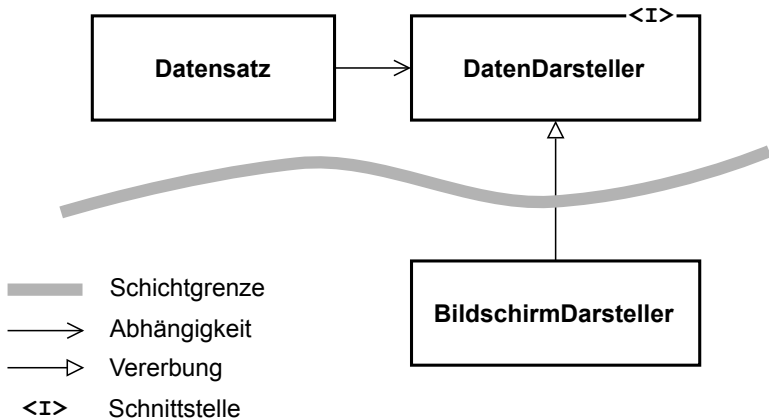
Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur



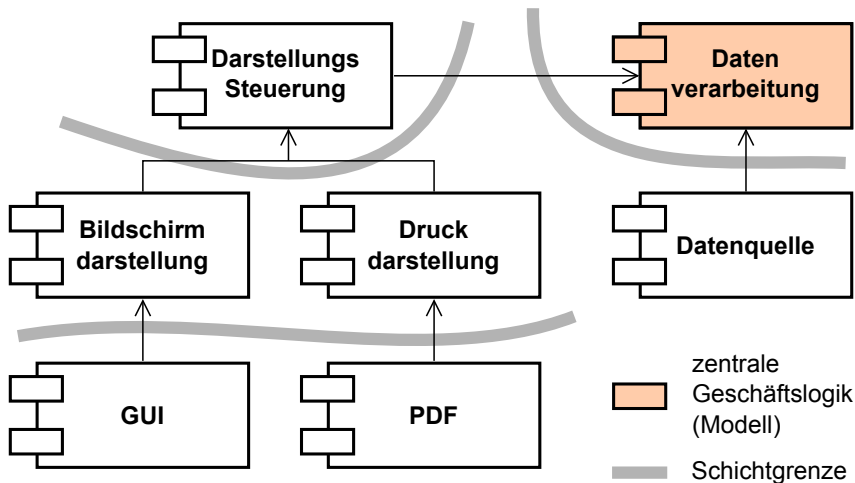
# Beispiele für seinen Einsatz

Die im Rahmen des OCP eingeführte abstrakte Schnittstelle



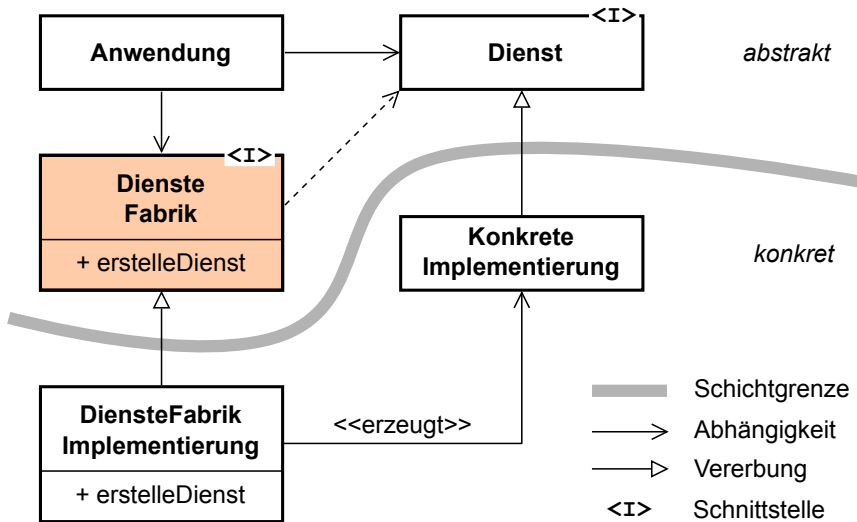
# Beispiele für seinen Einsatz

Eine komplexere Architektur mit Abhängigkeiten in *eine* Richtung



# Beispiele für seinen Einsatz

Der Kern der Umkehr von Abhängigkeiten: die *Abstract Factory*



### Abstrakte Fabrik (*abstract factory*)

Schnittstelle zur Erzeugung einer Familie von Objekten.  
Die konkreten Klassen der zu erzeugenden Objekte werden nicht näher festgelegt und sind dem Nutzer unbekannt.

- Erzeugung von Objekten wird ausgelagert.
  - Objekterzeugung führt zu Quellcode-Abhängigkeiten.
  - Nutzer einer abstrakten Fabrik kennt nur abstrakte Objekte
- konkrete Fabriken erben von abstrakter Fabrik
  - erzeugen konkrete Objekte
  - Liskov-Substitutionsprinzip sorgt für Kompatibilität

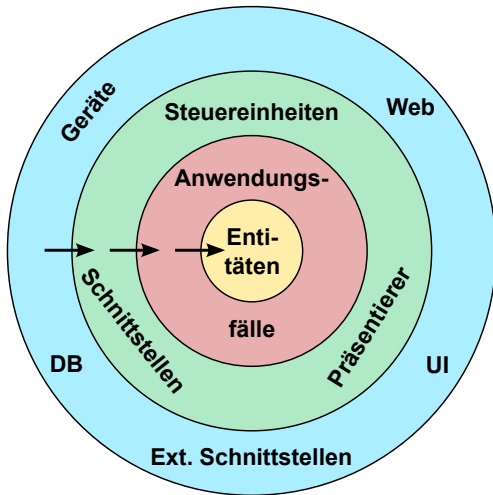
Das Dependency-Inversion-Prinzip

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

# Bedeutung im Gesamtkontext

Entscheidend für flexible, modulare, wiederverwendbare Architektur



Unternehmens-  
Geschäftsregeln

Anwendungs-  
geschäftsregeln

Schnittstellen-  
Adapter

Frameworks  
& Treiber

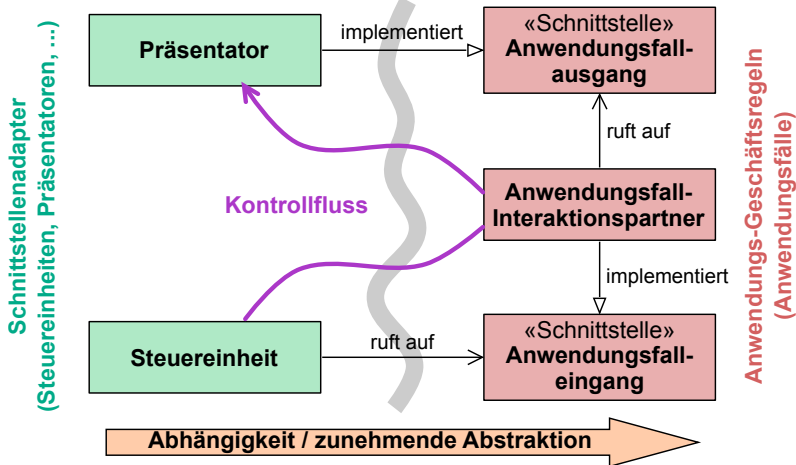
→ Richtung der  
Abhängigkeiten

**"saubere Architektur"**

verändert nach Robert C. Martin: Clean Architecture. Prentice Hall, Boston 2018, S. 203

- Kernaspekte
    - Abstraktion nimmt nach innen zu.
    - Abhängigkeiten zeigen immer nur nach innen.
    - Anwendungsfälle stehen *nicht* im Zentrum.
    - Schnittstellen nach außen sind peripher.
    - Jede Schicht hat ein konsistentes Abstraktionsniveau.
    - Schichten sind unabhängig voneinander testbar.
  - praktische Hinweise
    - Anwendungen von innen nach außen entwickeln
    - Die Anzahl der Schichten ist flexibel.
    - Schnittstellen nach außen über Frameworks abbildbar
- ☞ Der Kontrollfluss über die Schichtgrenzen wird über das Dependency-Inversion-Prinzip realisiert.

### Grenzen überschreiten in einer mehrschichtigen Architektur





### ■ Kernaspekte

- Anwendungsfälle gegen Schnittstellen implementieren
- Anwendungsfälle und zugehörige Schnittstellen befinden sich in der gleichen Schicht und Abstraktionsebene.
- Abhängigkeiten zeigen nie nach außen.
- Die äußere Schicht nutzt (innen liegende) Schnittstellen, um temporär den Kontrollfluss nach innen abzugeben.

### ■ praktische Hinweise

- Daten auf die im jeweiligen Kontext natürlichste Art abbilden
- Datenformate nicht von außen nach innen durchreichen
- Schnittstellen werden von der inneren Schicht diktiert.

☞ Die innere Schicht weiß nichts von der äußeren Schicht – und soll auch gar nichts von ihr wissen.

- entkoppelt die einzelnen Schichten eines Programms
    - garantiert Wiederverwendbarkeit abstrakter Module
    - ermöglicht die (unabhängige) Testbarkeit der Schichten
  - erlaubt Fokussierung auf das eigentlich Wesentliche
    - Zentral ist die Strategieschicht, alles andere ist peripher.
    - Peripherie oft über Frameworks implementierbar.
  - zentral für die Entwicklung von Frameworks
    - ermöglicht die notwendige Abstraktion und Modularität
  - grundlegender Mechanismus für die OOP
    - ermöglicht Flexibilität, Wiederverwendbarkeit, Wartbarkeit
- 👉 Verwendung des DIP entscheidet über objektorientierten oder strukturierten Entwurf einer Anwendung

“ *Depend in the direction of stability.*

– Robert C. Martin

- Stable-Dependencies-Prinzip (SDP)

- Veränderbarkeit ist der *raison d'être* von Software.
- Niemand sollte von unbeständigen Elementen abhängen.

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

– Robert C. Martin

- Stable-Abstractions-Prinzip (SAP)

- Abstraktion ermöglicht Erweiterung ohne Veränderung.

“ [D]ependencies run in the direction of abstraction.

– Robert C. Martin

- Zusammenfassung von SDP und SAP
    - Stabilität und Abstraktion hängen oft zusammen.
    - Gilt *nicht* für den „stabilen Hintergrund“ (Basistypen, ...)
  - DIP auf Komponenten-Ebene
    - DIP gilt für Klassen: sind entweder abstrakt oder konkret
    - Komponenten können teilweise abstrakt und konkret sein.
- ☛ Abstraktion nimmt zum Kern einer Anwendung hin zu.
- ☛ Gute Abstraktionen ermöglichen Wiederverwendbarkeit.



- Die Kernaspekte einer Anwendung sollten nicht von ihrer Peripherie abhängen – sondern beide von Abstraktionen.
- Anwendungsentwicklung zielt auf die zugrundeliegenden Abstraktionen und ihre stabilen, abstrakten Schnittstellen.
- Anwendungen bestehen aus klar getrennten Schichten, die Dienste über definierte Schnittstellen bereitstellen.
- Die Anwendungslogik steht im Zentrum.  
Datenquellen und Nutzerschnittstellen sind peripher.
- Umkehr der Abhängigkeiten sorgt für Unabhängigkeit und intrinsische Testbarkeit der einzelnen Schichten.