Programmierkonzepte in der Physikalischen Chemie

27. Dependency-Inversion-Prinzip

UNI FREIBURG

Albert-Ludwigs-Universität Freiburg

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2016/17

Zentrale Aspekte

UNI FREIBURG

- Anwendungen bestehen aus klar getrennten Schichten, die Services über definierte Schnittstellen bereitstellen.
- Die Kernaspekte einer Anwendung sollten nicht von ihrer Peripherie abhängen – sondern beide von Abstraktionen.
- Die zugrundeliegenden Abstraktionen zu finden und zu implementieren ist das Ziel der Anwendungsentwicklung.
- Die Anwendungslogik steht im Zentrum.
  Daten und Nutzerschnittstellen sind peripher.
- Umkehr der Abhängigkeiten sorgt für intrinsische Testbarkeit der einzelnen Schichten.

Übersicht

UNI FREIBURG

Das Dependency-Inversion-Prinzip

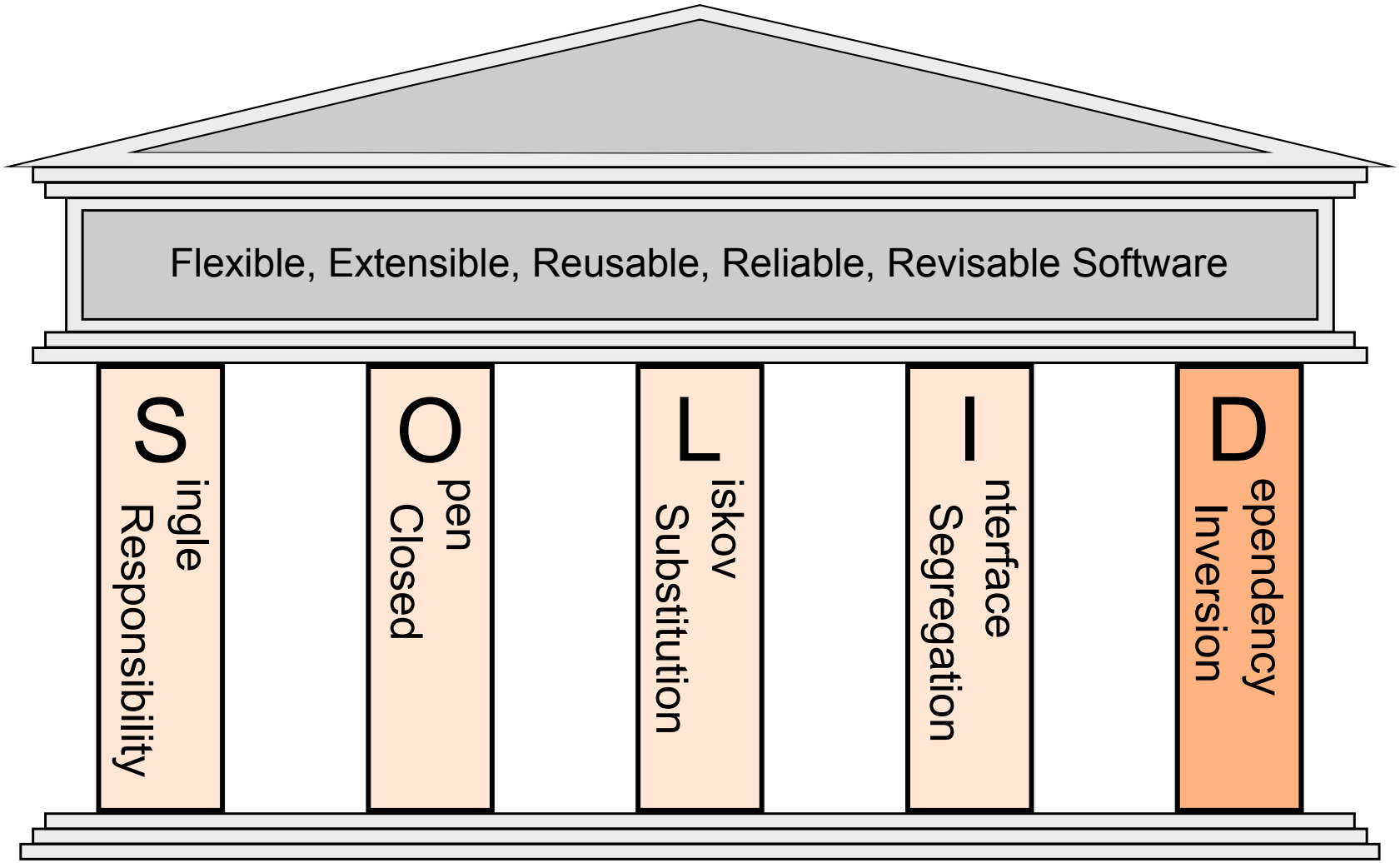
Symptome, die für seinen Einsatz sprechen

Beispiele für seinen Einsatz

Bedeutung im Gesamtkontext der Software-Architektur

Das Dependency-Inversion-Prinzip

Übersicht über die fünf Prinzipien



Das Dependency-Inversion-Prinzip

Umkehr der Abhängigkeiten sorgt für Modularität

UNI FREIBURG

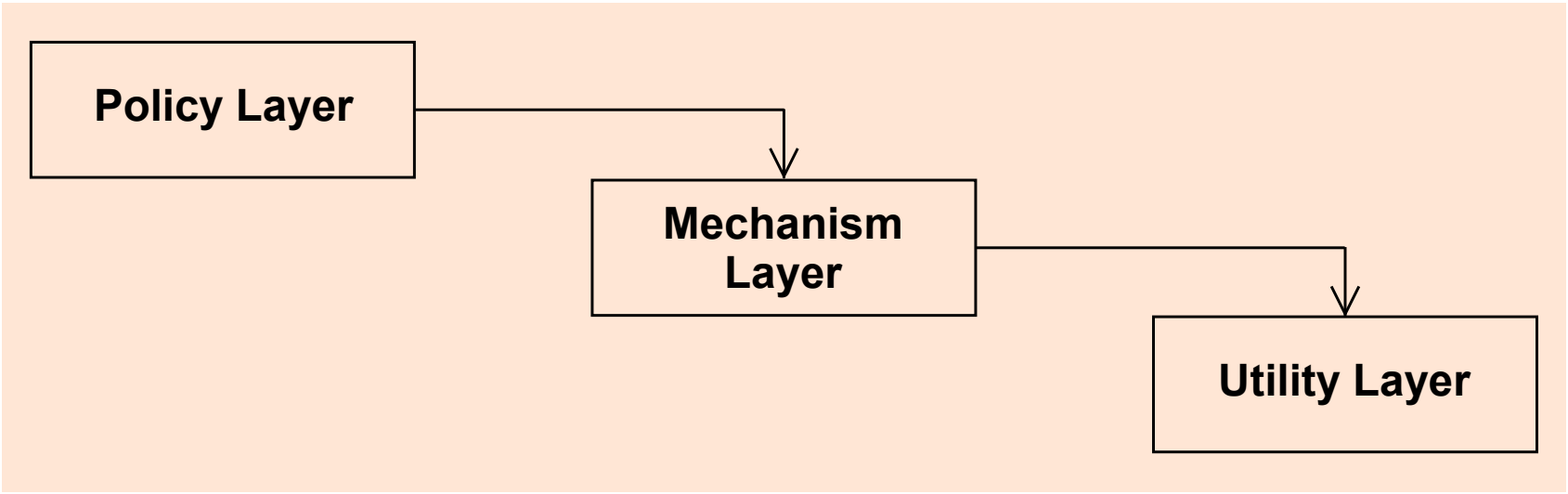
> *a. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> *b. Abstractions should not depend on details. Details should depend on abstractions.*
>
> – Robert C. Martin

- Komplexere Programme sind in Schichten unterteilt.
  - Entscheidend ist die Richtung der Abhängigkeit zwischen den Schichten.
- Abhängigkeit und Kontrollfluss lassen sich trennen.
  - Die Abhängigkeitsbeziehung lässt sich invertieren.

Robert C. Martin: Agile Software Development. Prentice Hall, Upper Saddle River 2003, S. 127

Das Dependency-Inversion-Prinzip

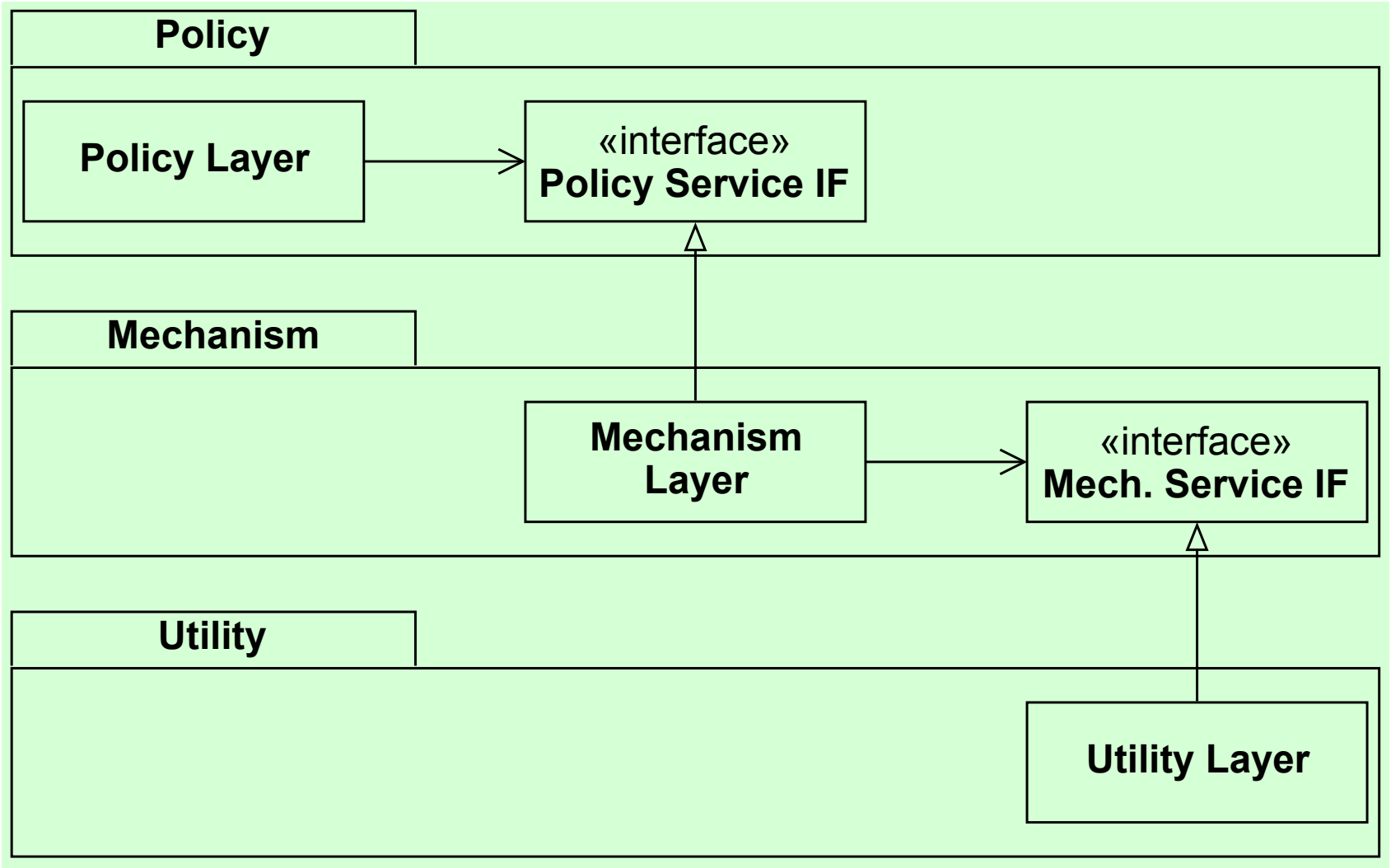
Schichten eines Programms – der naive Ansatz



- Problem: Abhängigkeiten sind transitiv.
  - Die Strategieschicht (*Policy Layer*) hängt hier (auch) von der Werkzeugschicht (*Utility Layer*) ab.
  - Bsp.: Die Prozessierung hängt von der Datenbank ab.
- Die Schichten sollten separiert werden.
  - Abhängigkeiten invertieren, Kontrollfluss beibehalten

Das Dependency-Inversion-Prinzip

Schichten eines Programms mit umgekehrter Abhängigkeit



Symptome, die für seinen Einsatz sprechen

Geringe Flexibilität und Testbarkeit, Unübersichtlichkeit

UNI FREIBURG

- mangelnde Flexibilität
  - System schwer in einzelne Komponenten aufteilbar
  - Austausch peripherer Komponenten (Nutzerschnittstelle, Persistenz) erzwingt Änderungen der Anwendungslogik.
- mangelnde Testbarkeit
  - Die einzelnen Schichten sind nicht unabhängig testbar.
  - Unittests sind entsprechend nicht durchführbar.
- Abhängigkeit von (unwichtigen) Details
  - Die Anwendungslogik hängt von peripheren Schichten ab.
  - Abstrakte(re) hängen von konkrete(re)n Komponenten ab.
- Unübersichtlichkeit
  - Abstraktionsebenen sind bunt gemischt.
  - eigentliche Aufgabe der Anwendung schwer erkennbar

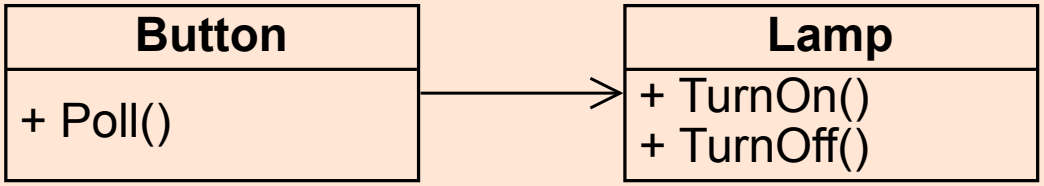
Beispiele für seinen Einsatz

Wann immer zwei Klassen/Objekte Nachrichten austauschen

- Knopf
  - nimmt seine Umgebung wahr
  - kann feststellen, ob er gedrückt wurde
- Lampe
  - beeinflusst die externe Umgebung
  - schaltet je nach Nachricht ein Licht an oder aus

Beispiele für seinen Einsatz

Knopf und Lampe: eine erste naive Implementierung



Listing 1: Beispielimplementierung der Button-Klasse (in Java)

```
public class Button
{
  private Lamp itsLamp;
  public void poll()
  {
    if (/*some condition*/)
      itsLamp.turnOn();
  }
}
```

- `Button` hängt direkt von `Lamp` ab.
- `Button` kann nicht unabhängig wiederverwendet werden.

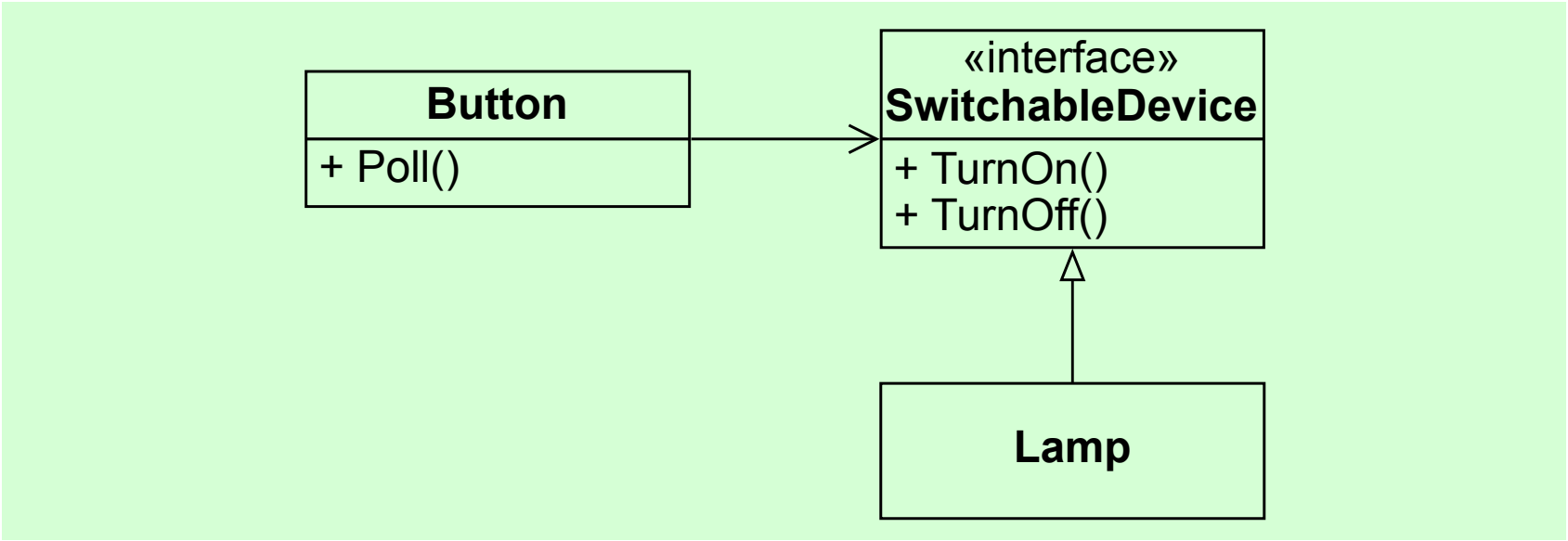
Gesucht: die zugrunde liegende Abstraktion

- grundlegende Strategie
  - der Anwendung zugrunde liegende Abstraktion
  - ändert sich nicht, wenn sich die Details ändern
  - letztlich die Metapher
- konkreter Fall: Knopf und Lampe
  - Schaltvorgang („an“ bzw. „aus“) wahrnehmen und an ein Zielobjekt weitergeben
- unwichtige Details
  - Welcher Mechanismus wird verwendet, um den Schaltvorgang wahrzunehmen?
  - Was ist das Zielobjekt?

☛ Abstraktion sollte unabhängig und wiederverwendbar sein.

Beispiele für seinen Einsatz

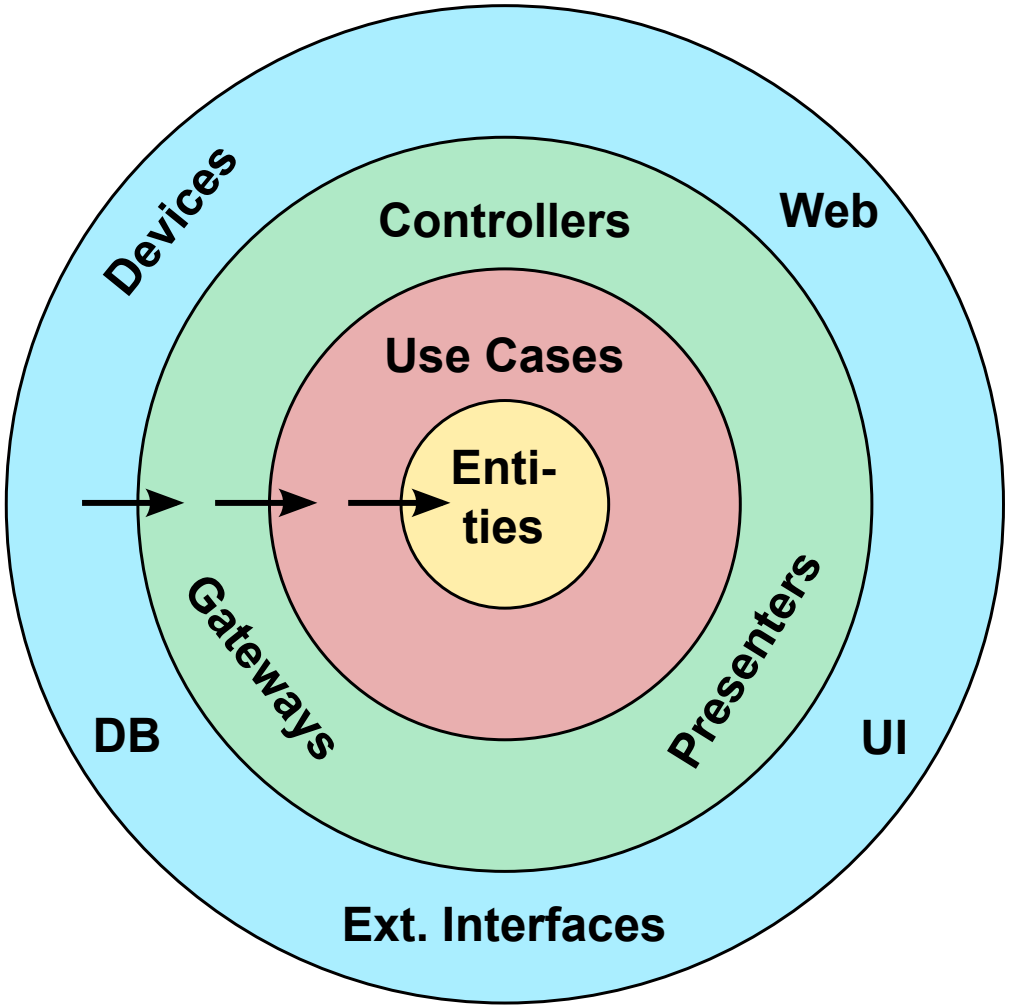
Knopf und Lampe: eine DIP-konforme Implementierung



- `Button` kann beliebige Objekte steuern.
  - Voraussetzung: Implementierung der Schnittstelle
  - gilt auch für Objekte, die noch gar nicht erfunden wurden
- Umbenennung der Schnittstelle kann hilfreich sein.
  - `ButtonServer` ⇒ `SwitchableDevice`

Bedeutung im Gesamtkontext

Entscheidend für flexible, modulare, wiederverwendbare Architektur



Enterprise Business Rules

Application Business Rules

Interface Adapters

Frameworks & Drivers

The Clean Architecture

Bedeutung im Gesamtkontext

Anmerkungen zur „Clean Architecture“

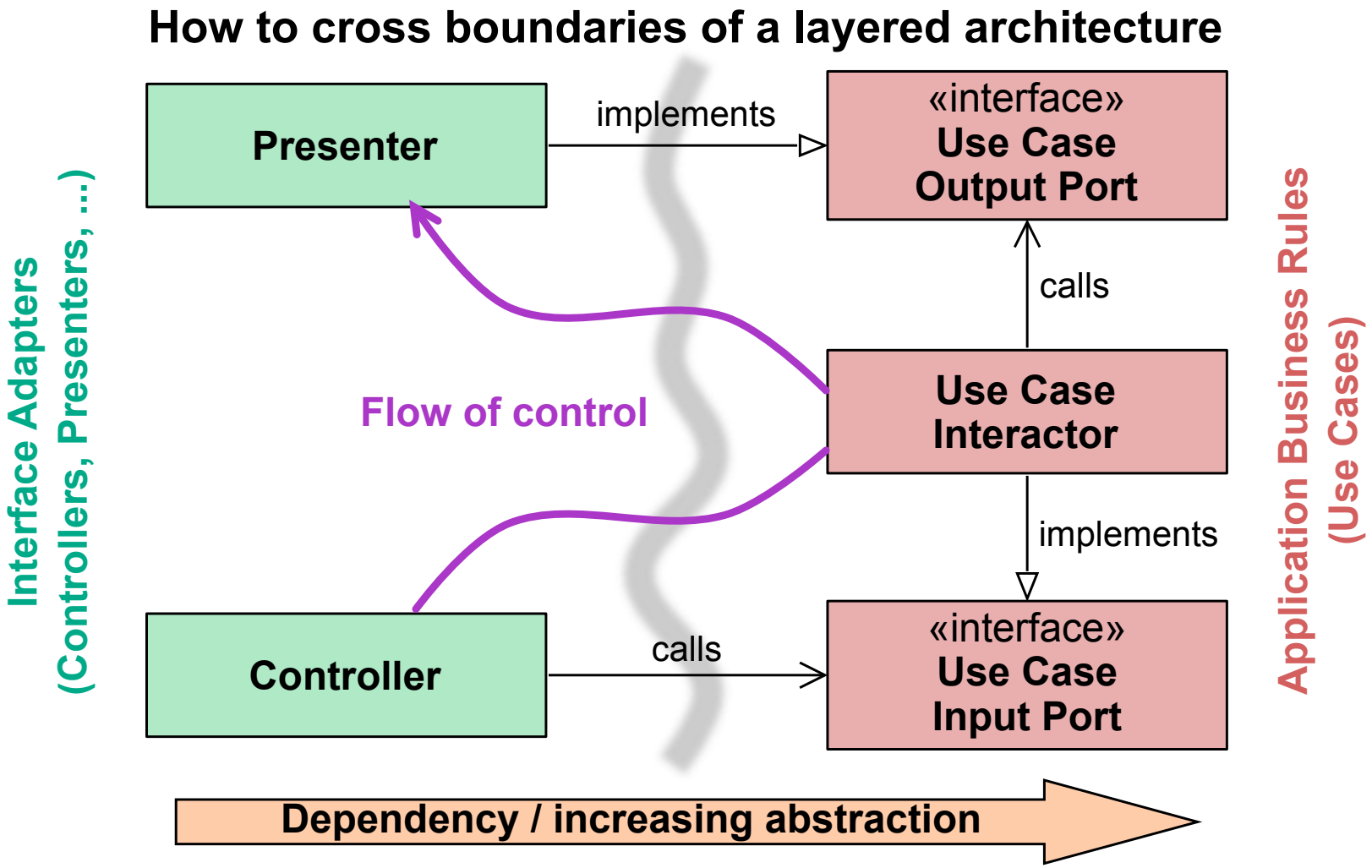
UNI FREIBURG

- 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.

Bedeutung im Gesamtkontext

Abhängigkeiten und Kontrollfluss über Schichtgrenzen



Bedeutung im Gesamtkontext

Anmerkungen zum Kontrollfluss über Schichtgrenzen

UNI FREIBURG

- zentrale Aspekte
  - 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.

Bedeutung im Gesamtkontext

Essentiell für objektorientierte Programmierung

UNI FREIBURG

- 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

Zentrale Aspekte

UNI FREIBURG

- Anwendungen bestehen aus klar getrennten Schichten, die Services über definierte Schnittstellen bereitstellen.
- Die Kernaspekte einer Anwendung sollten nicht von ihrer Peripherie abhängen – sondern beide von Abstraktionen.
- Die zugrundeliegenden Abstraktionen zu finden und zu implementieren ist das Ziel der Anwendungsentwicklung.
- Die Anwendungslogik steht im Zentrum.
  Daten und Nutzerschnittstellen sind peripher.
- Umkehr der Abhängigkeiten sorgt für intrinsische Testbarkeit der einzelnen Schichten.