HOCHSCHULE
DER MEDIEN

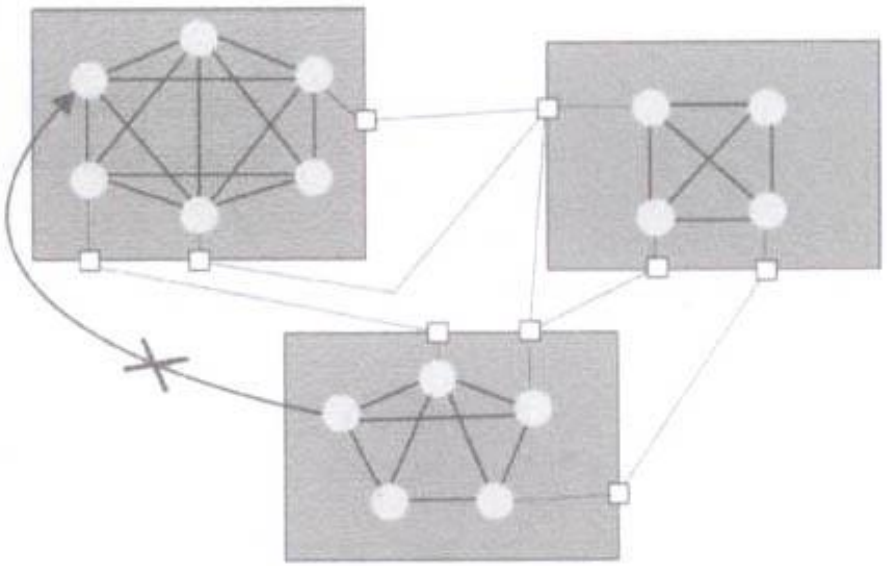
# Softwarearchitektur und -design

# Modularisierung

HOCHSCHULE DER MEDIEN

Ziele:

- Beherrschbarkeit
- Flexibilität
- Strukturierung



Um das zu erreichen:

- Wenige Abhängigkeiten zwischen den Modulen
- Module mit geringer Wechselwirkung
- Schmale Modul-Schnittstellen
- Benutzungs-Abstraktion
- Information-Hiding

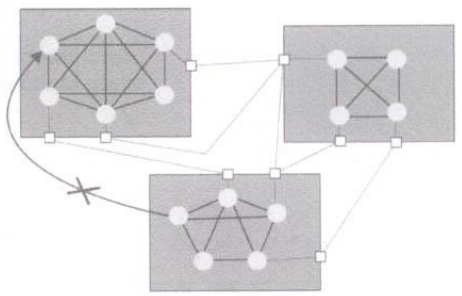
Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Modularisierung

HOCHSCHULE DER MEDIEN

Module sollten folgende Eigenschaften besitzen:

- Benutzungsabstraktion
  - Benutzen und Testen ohne Implementierung zu kennen
- Gegenseitige Nichtbeeinflussung
  - Verborgene Implementierung
  - Keine Annahmen, die nicht in der Schnittstellenbeschreibung dokumentiert sind
- Kontextunabhängigkeit
  - Wiederverwendbarkeit auch in anderer Umgebung

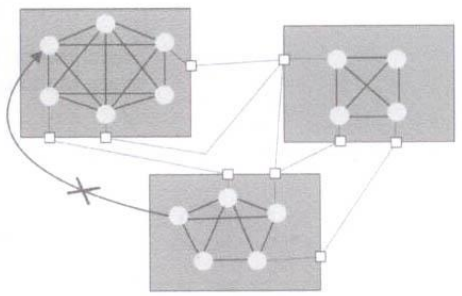


Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Modularisierung

Module sollten folgende Eigenschaften besitzen:

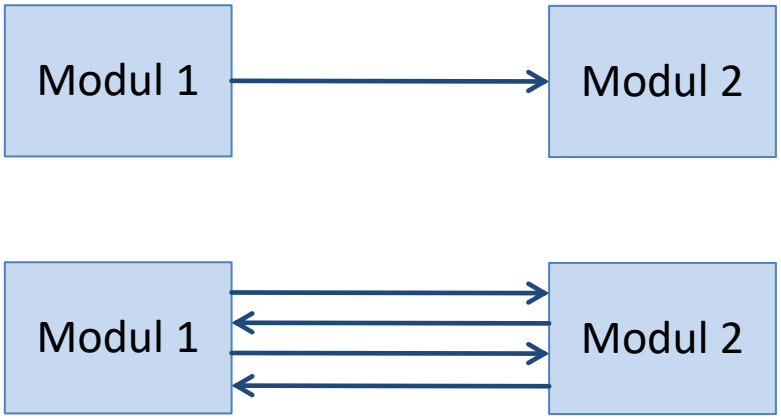
- Getrennt entwickelbar, testbar, änderbar
- Wartbarkeit
  - Leicht an geänderte Umgebung anpassbar
- Lokalitätsprinzip
  - Alle Information an einer Stelle



Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

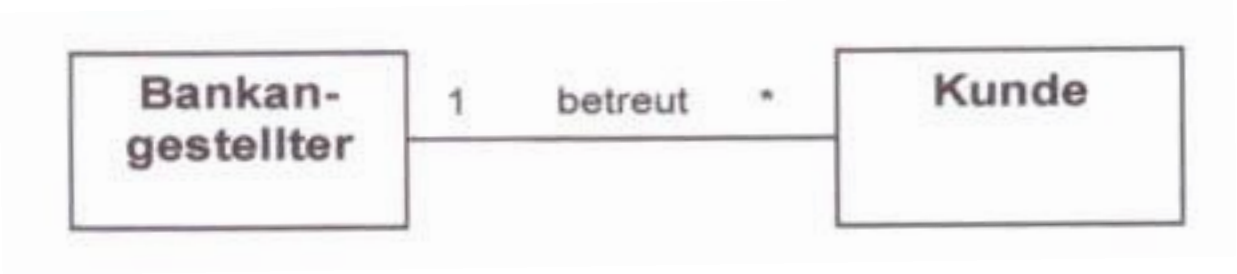
- Entstehen, wenn ein Modul ein anderes benutzt
- Lassen sich nicht vermeiden, aber minimieren



# Abhängigkeiten

Assoziation

- Beide Objekte brauchen sich gegenseitig




Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Komposition

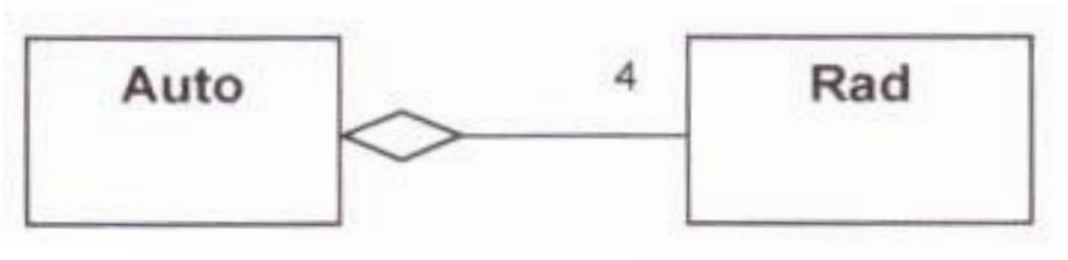
- Ein Objekt ist Teil des anderen und
- Das Teil Objekt lebt genau so lang wie das Ganze


Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Aggregation

- Ein Objekt referenziert ein anderes Objekt
- Objekt kann von mehreren anderen referenziert werden
- Lebensdauer kann unterschiedlich sein

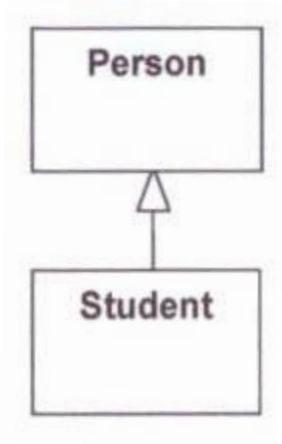



Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Generalisierung und Spezialisierung

- Vererbung
- Generelle Eigenschaften in Basisklasse
- Spezielle Eigenschaften in abgeleiteter Klasse
- „is a“-Beziehung

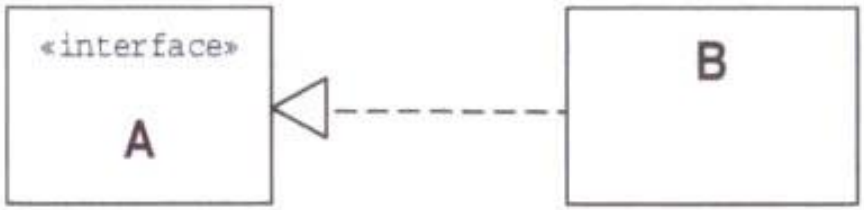



Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Realisierung

- Gegebener Vertag/Schnittstelle wird konkretisiert
- Realisierung ist stark von der Definition anhängig




Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Allgemein

- A ist unabhängig
- B ist von A abhängig
- Eine Änderung von A, kann eine Änderung von B erfordern


Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Abhängigkeiten

Logische Abhängigkeiten

- Werden häufig erst bemerkt, wenn etwas schief läuft
- Treten auf, wenn Annahmen gemacht werden die nicht i. A. gelten

➔sollten **vermieden** oder wenigstens **dokumentiert** werden

Beispiel:

- Aktuelles Datum als String gegeben
- Annahmen über das Format (30.10.2020 vs. 10/30/2020)

# Abhängigkeiten

HOCHSCHULE DER MEDIEN

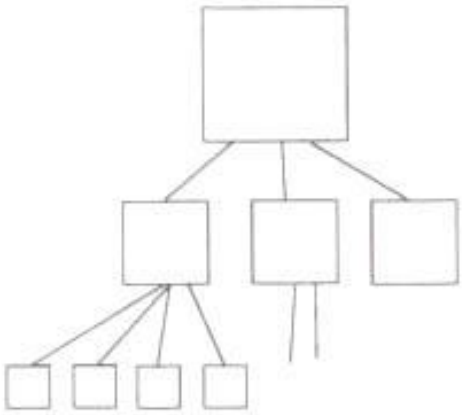
Ziel

- Reduzierung von Abhängigkeiten
- Vermeidung i.d.R. nicht möglich

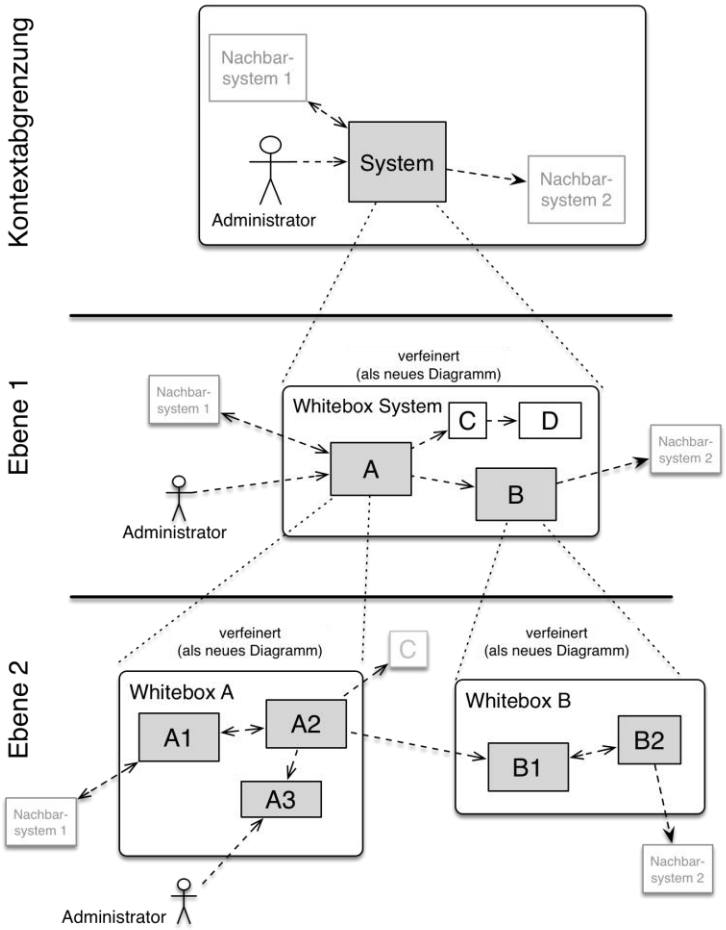
➔ Verwendung von Entwurfs- und Design-Prinzipien

# Entwurfsprinzipen für Modulare Struktur

- Teile und Herrsche (Top-Down)
  - Schrittweise Zerlegung in Teilprobleme
- Bottom-Up
  - Aufbauen von Infrastruktur
- Design-to-Test
  - unabhängig testbare Komponenten



Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer




Bildquelle: arc42 Projetdokumentations-Template

# Entwurfsprinzipien zur Komplexitätsreduktion

HOCHSCHULE DER MEDIEN

- Keep it Simple, stupid (KISS)
  - Einfache Systeme sind leicht zu verstehen, aber oft schwer zu designen
  - Komplexe Systemarchitekturen sind oft leichter zu finden

Vorteile:

- Einfachere zu bauen, teste, ändern
- Weniger fehleranfällig
- Weniger Abhängigkeiten
- Konzentration auf das Wesentliche

# Entwurfsprinzipien zur Komplexitätsreduktion

HOCHSCHULE DER MEDIEN

- You aren´t gonna need it (YAGNI)
  - Nichts implementieren, was nicht tatsächlich gebraucht wird
  - Keine spekulativen Generalisierungen
  - Vermeidung von Overengineering

Vorteile:

- Weniger Aufwand
- Fördert änderbaren und erweiterbaren Code
- Genaue Spezifikation
- Vermeidung von überflüssigen Tests

Aber:

- Nicht in jedem Fall zu empfehlen
- gute Basis ist notwendig
- Weiterentwickelbarkeit nicht vernachlässigen !!!

# Entwurfsprinzipien zur Komplexitätsreduktion

HOCHSCHULE DER MEDIEN

- Don´t repeat yourself (DRY)
  - Nichts darf dupliziert werde
  - Betrifft jede Art von Information (Code, Skripte, Dokumentation, ….)

Nicht immer umsetzbar:

- Fehlertolerante Systeme erfordern prinzipiell Redundanzen
- Manchmal sind Redundanzen aus Performance-Gründen nötig

➔ **Verletzung nur in begründeten Ausnahmefällen !!!**

# Entwurfsprinzipien zur Komplexitätsreduktion

HOCHSCHULE DER MEDIEN

- Single Level of Abstraction
  - Jedes Programm besitzt Elemente unterschiedlicher Abstraktionsstufen
  - Verwende in einer Methode nur Anweisungen derselben Abstraktionsstufe

Beispiel:

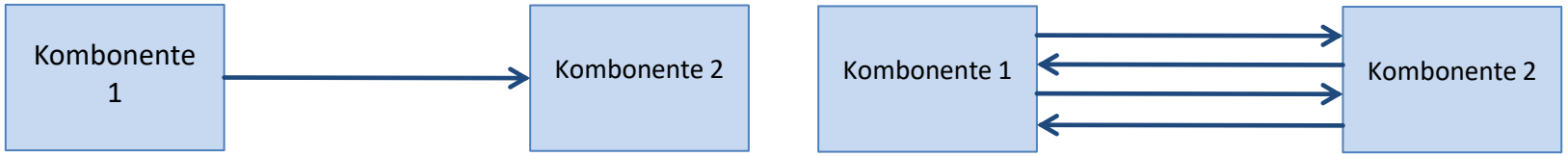
```
loadData();
processData();
storeResult();
```

# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Loose Coupling and Strong Cohesion
  - Schmale Schnittstellen zwischen den Komponenten
  - Abhängigkeiten nur von den Schnittstellen, nicht von der Implementierung

Regel: Je stärker Modul in sich zusammenhängen, desto schwächer die Abhängigkeiten nach außen



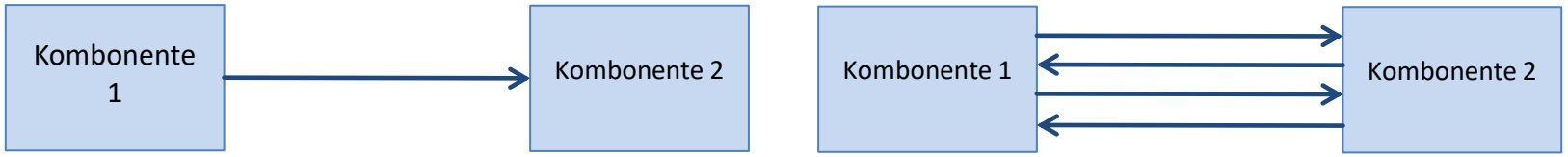
# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Loose Coupling and Strong Cohesion

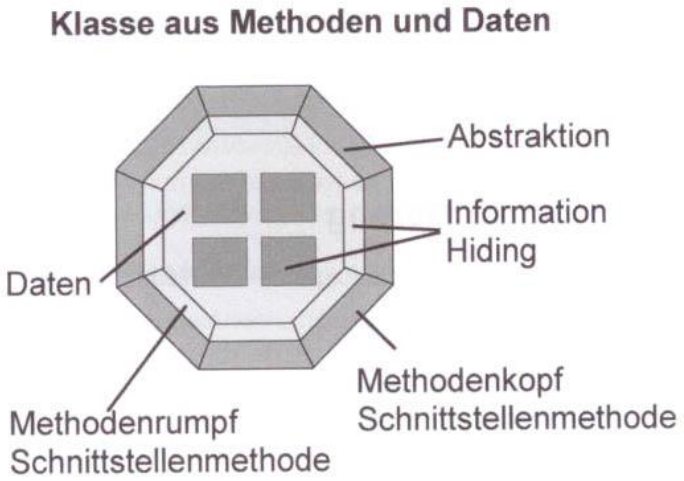
Vorteile:

- Geringe Abhängigkeiten zwischen den Komponenten
- Komponenten isoliert testbar
- Komponenten isoliert wiederverwendbar
- Leichtere Wartbarkeit und Erweiterbarkeit
- Arbeitsteilige Entwicklung
- Load-on-the-fly: in Komponenten sollte möglichst nichts sein, was nicht benötigt wird



# Prinzipien und Konzepte für schwach gekoppelte Systeme

- Information Hiding (Black Box)
  - Zugriff nur über schmale Schnittstelle
  - Implementierung austauschbar
  - Leichter testbar
  - Gilt für Module wie für Klassen




Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

Von Bin im Garten - Eigenes Werk, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=11452111

# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Separation of Concerns (SoC)
  - Eine Komponente kümmert sich nur um ein einzigen Belang
  - Fokussierung
  - Arbeitsteilige Entwicklung
  - Einzeln testbar

Umsetzung:

- Klassen in OO
- Funktionen, Methoden und Prozeduren
- Schichten in Schichtenmodell (Zwiebelschalenmodell)

# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Law of Demeter
  - Jeder darf nur mit direkten Nachbarn kommunizieren
  - Ziel: Abhängigkeiten zwischen Komponenten reduzieren

➔ Eine Methode eines Objekt darf nur Methoden aufrufen:

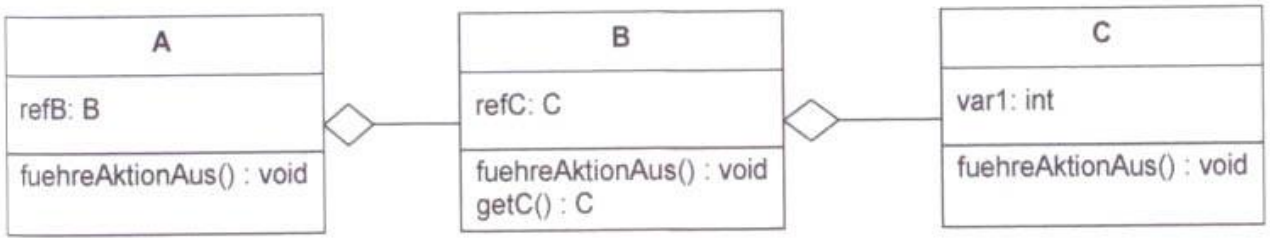
- Des Objektes selbst
- Der übergebenen Objekte
- Von lokal erzeugten Objekten
- Eines static Objektes der eigenen Klasse

➔ „Schüchterner Code“

# Prinzipien und Konzepte für schwach gekoppelte Systeme

- Law of Demeter

Beipiel:




in einer Methode von A wäre verboten:

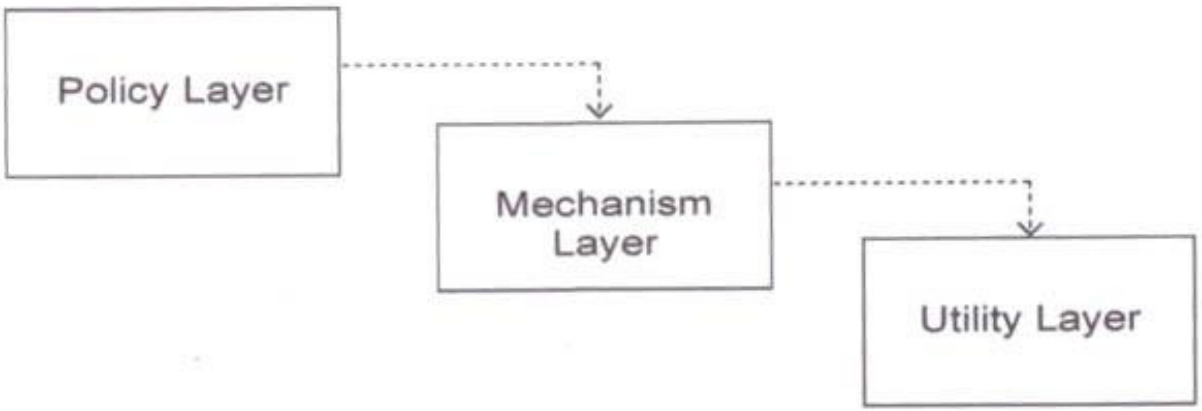
```
refB.getC().fuehreAktionAus();
```

Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Dependency-Inversion-Principle
  - Problem: hierarchisch höher liegende Komponenten hängen beim klassischen Entwurf von tieferen ab



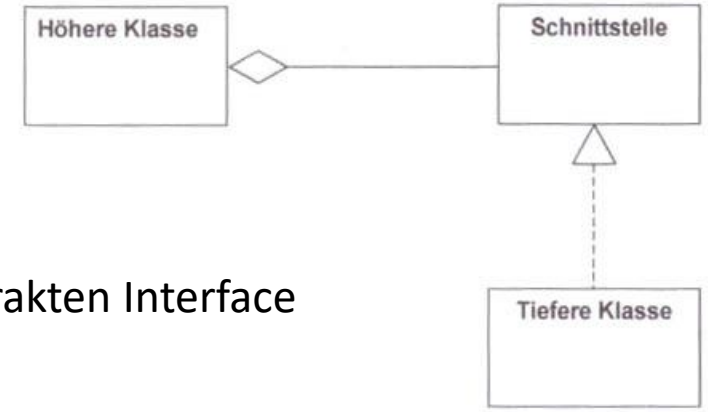

Abhängigkeit von einer konkreten Implementierung

Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Prinzipien und Konzepte für schwach gekoppelte Systeme

- Dependency-Inversion-Principle
  - Lösung: höhere Klasse legt den Vertrag fest, die niedrigere implementiert ihn

➔ Interfaces




Abhängigkeit von einem abstrakten Interface

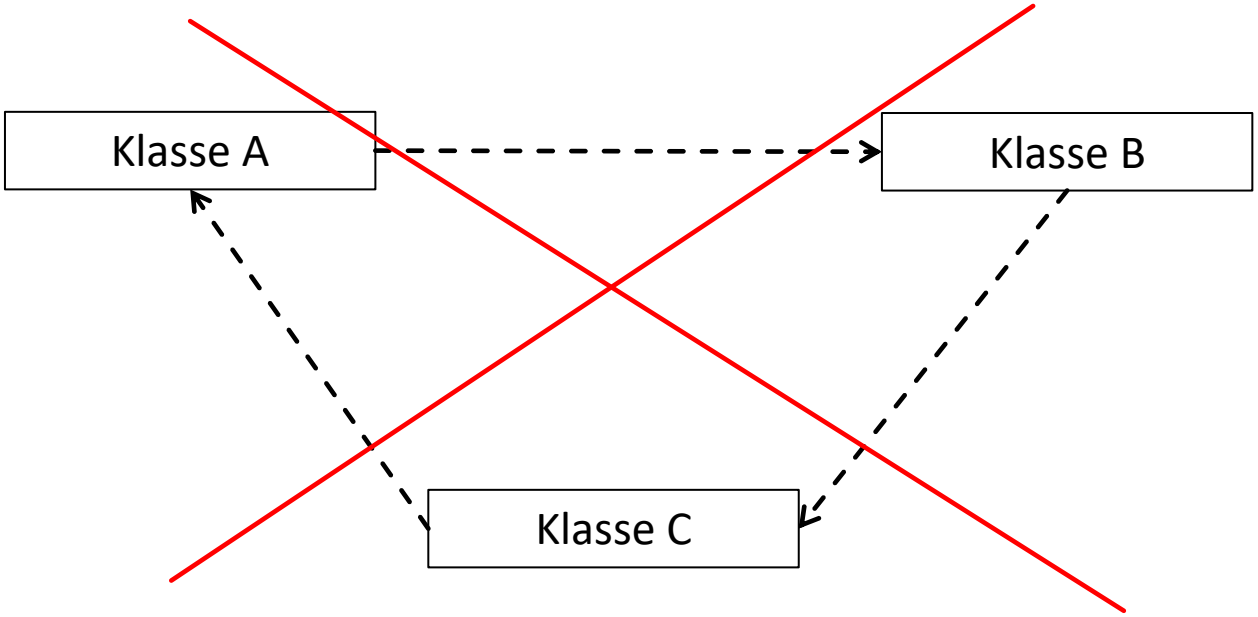
Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Prinzipien und Konzepte für schwach gekoppelte Systeme

HOCHSCHULE DER MEDIEN

- Acyclic-Dependencies-Prinzip

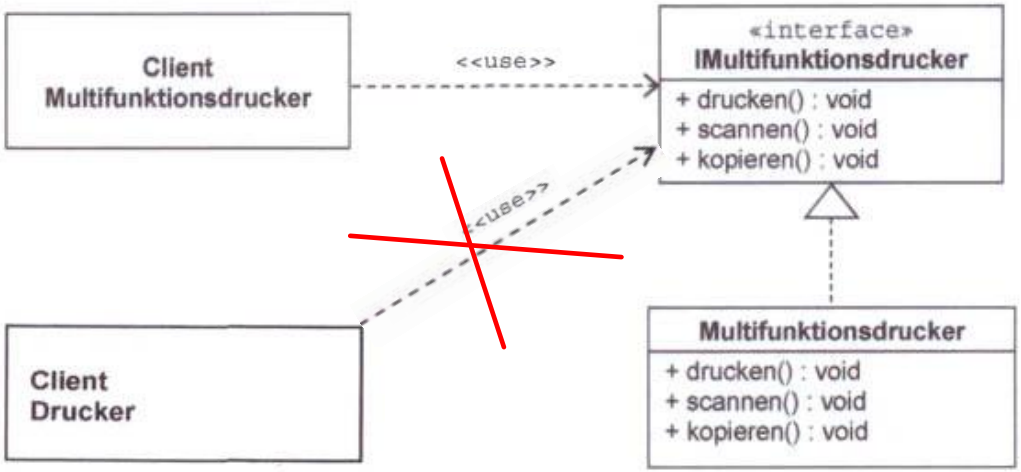
➔ Lasse keine zyklischen Abgängigkeiten zu
z.B: Dependency-Inversion-Principle verwenden



# Interface Segregation Principle (ISP)

- Schnittstellen sollen nur benötigte Methoden enthalten
- Keine Abhängigkeiten von Methoden, die nicht verwendet werden

➔ Loose Coupling and Strong Cohesion




Bildquelle J. Goll: Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik, Springer

# Single Responsibility Principle

HOCHSCHULE DER MEDIEN

- Jedes Modul/jede Klasse hat eine einzige Verantwortlichkeit

  „Gather together the things that change for the same reason“

  „separate those things that change for separate reasons“

➔ Strong Cohesion

# Dependency Injection

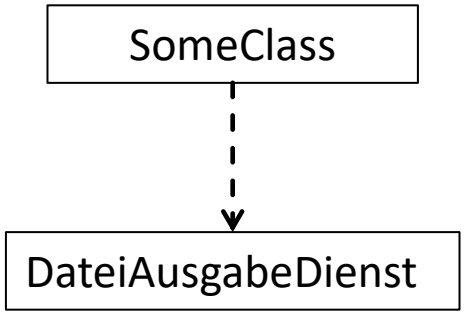
HOCHSCHULE DER MEDIEN

- Vermeiden, dass eine Klasse eine Objekt einer anderen Klasse erzeugt

Beispiel für Abhängigkeit:

```
public class SomeClass
// hängt von der Klasse DateiAusgabeDienst ab
{
   private AusgabeDienst dienst;

   public SomeClass ()
   {
      dienst = new DateiAusgabeDienst(…);
   }
}
```



# Dependency Injection

HOCHSCHULE DER MEDIEN

- Stattdessen: Objekt  wird außerhalb erzeugt und „injiziert“

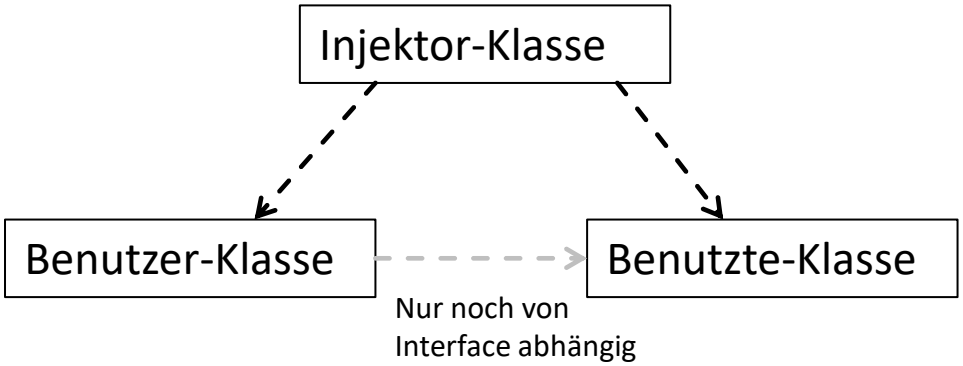
```
Interface IAusgabeDienst …

public class SomeClass
{
   private AusgabeDienst dienst;

   public SomeClass (IAusgabeDienst ausgabeDienst)
   {
      dienst = ausgabeDienst;
   }
}
```

# Dependency Injection

- Abhängigkeiten werden nicht beseitigt, sondern nur reduziert
- Man benötigt „Injector“-Klasse, die dann von beiden Klassen abhängig
- Möglichkeiten der Injektion
  - Über Konstruktor (wie im Beispiel der vorigen Folie)
  - Über eine Setter-Methode
  - Über ein Interface



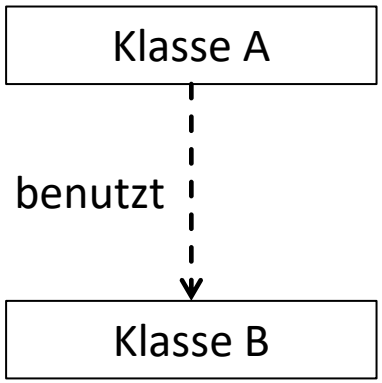
# Entwurfsprinzipien und -konzepte

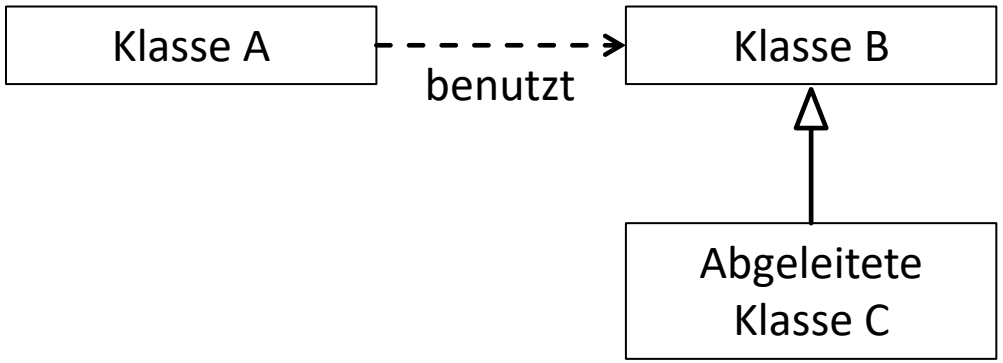
HOCHSCHULE DER MEDIEN

- Design by Contract
- Liskovsches Substitutionsprinzip
- Principle of Least Astonishment

# Design by Contrct (DbC)

- Sicherstellung dass Bedingungen von Methodenaufrufen auch tatsächlich eingehalten werden
- Sicherstellung dass abgeleitete Klassen die Bedingungen der Basisklassen auch erfüllen
- Vereinbarung von „Verträgen“, deren Einhalten zur Laufzeit überprüft werden.



HOCHSCHULE DER MEDIEN

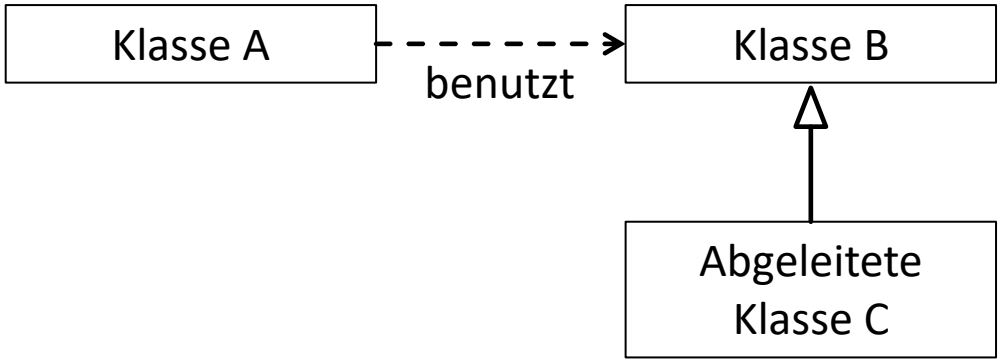
## Vertrag umfasst:

- Vorbedingungen für Methodenaufrufe
  - Verantwortlich: Aufrufer
    Häufig Überprüfung in der Methode -> Exception bei Verletzung
  - Nutzen: Aufgerufener
- Nachbedingungen für Methodenaufrufe
  - Verantwortlich: Aufgerufener
  - Nutzen: Aufrufender
- Invarianten der Klasse
  - Verantwortlich: Klasse selber
  - Gelten während der gesamten Objekt-Lebensdauer
  - Können innerhalb einer Methode temporär verletzt sein
  - beim Beenden müssen sie erfüllt sein

# Design by Contrct (DbC)

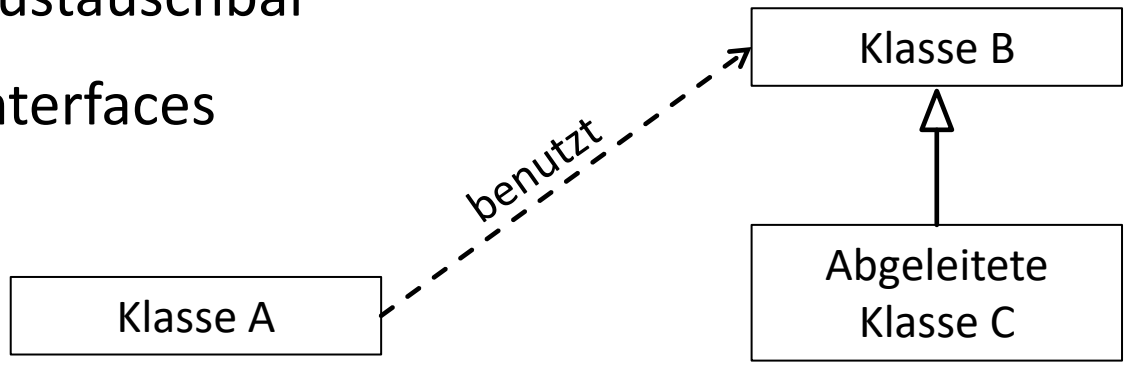
Überschreiben von Methoden:

- Vorbedingungen dürfen abgeschwächt aber nicht verschärft werden
- Nachbedingungen dürfen verschärft aber nicht abgeschwächt werden
- Invarianten dürfen verschärft aber nicht abgeschwächt werden



# Liskovsches Substitutionsprinzip (LSP)

- Referenzen auf Objekte einer Basisklasse müssen auch Objekte abgeleiteter Klassen referenzieren können
  - ➔ Durch Klassenhierarchie entsteht eine Typ-Hierarchie
  - ➔ Beim Überschreiben von Methoden müssen die Verträge der Basisklasse eingehalten werden
  - ➔ Objekte sind austauschbar
- Dasselbe gilt für Interfaces



# Principle of Least Astonishment (PLA)

HOCHSCHULE DER MEDIEN

- Verhalten eines Programms darf den Benutzer nicht überraschen
- Funktionen von UI-Elementen solle auf den ersten Blick erkennbar sein

  ➔Erwartungen der Benutzer müssen bekannt sein
- Programmierschnittstellen (Methoden, Interfaces,…) sollen genau das bewirken, was Namen und Parameter vermuten lassen

# Prinzipien für Stabilität und Erweiterbarkeit

HOCHSCHULE DER MEDIEN

Prinzipien tragen zu Stabilität bei, wenn sie für

- eine schwache Wechselwirkung zwischen den Modulen sorgen
- eine starke Kohäsion innerhalb eines Moduls sorgen

- Open-Closed Principle
- Objektkomposition vor Vererbung
- Programmiere gegen Schnittstellen, nicht gegen Implementierungen

# Open-Closed Principle (OCP)

HOCHSCHULE DER MEDIEN

- Betrifft Programme und Spezifikationen
- Module sollen offen für Erweiterungen aber geschlossen für Modifikationen sein
- Änderungen der Anforderungen dürfen nicht durch Ändern vorhandenen Codes erfüllt werden, sondern durch Wiederverwenden stabiler bereits getesteter Programmteile

  ➔ Vermeidung von Fehlern in bestehender Funktionalität beim Einbau zusätzlicher Funktionen

# Open-Closed Principle (OCP)

HOCHSCHULE DER MEDIEN

- Erlaubt für Erweiterungen:
  - Ableitung weiterer Klassen
  - Zusätzliche Objekt-Kompositionen
  - Zusätzliche Methoden
- Vorteil:
  - Stabilität  bei Erweiterungen
  - Erhöhung der Wiederverwendbarkeit
- Nachteil:
  - Höherer Abstraktionsgrad erforderlich

# SOLID

Formulierung von Robert C. Martin: Clean Code, Prentice Hall:

**S**ingle Responsibility Principle
Do one thing and do it well

**O**pen Closed Principle
Objects should be open for extension and closed for midification

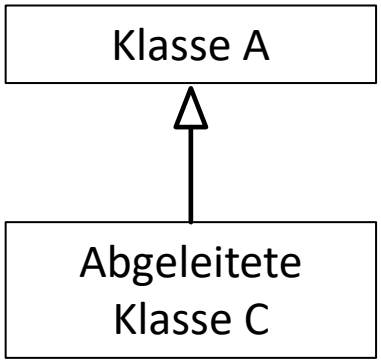
**L**iskovsches Substitutionsprinzips
Subclass Object “is a” Baseclass Object

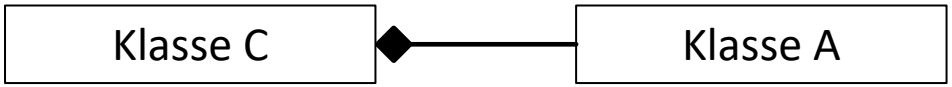
**I**nterface Segregation Principle
A client should never be forced to implement an part of an interface that it does not use

**D**ependency-Inversion-Principle
Entities must depend on abstractions not on concretions

# Favour Composition Over Inheritance (FCOI)

- Objektkomposition vor Vererbung
- Führt zu Verringerung von Abhängigkeiten
- Abgeleitet Klasse hängt stark von der Basisklasse ab
- Bei FCOI ist Inhalt der Klasse nicht sichtbar. Abhängigkeit nur von der Schnittstelle



# Favour Composition Over Inheritance (FCOI)

HOCHSCHULE DER MEDIEN

- Vorteile:
  - Keine komplexen Klassenhierarchien
  - Simulation von Mehrfachvererbung
  - Leichteres Testen (Verwendung von Mock-Objekten)
  - Flexibilität (Austausch zur Laufzeit)
  - Dependency Injection ist einfacher
- Nachteile:
  - Bei gleichzeitiger Anwendung des „Interface Segregation Principle“ oft große Anzahl an Interfaces notwendig
  - Keine direkte Verwendbarkeit von Methoden der Basisklasse. Methoden müssen weitergereicht werden
- Vererbung ist sinnvoll bei „is a“-Beziehung

# Programmiere gegen Schnittstellen, nicht gegen Implementierungen

HOCHSCHULE DER MEDIEN

- Verwendung konkreter anderer Klassen erzeugt Abhängigkeiten

➔ Verwendung abstrakter Datentypen

  - Abstrakte Klassen
  - Interfaces

- Erfordert
  - Einführung einer zusätzlichen Schicht
  - Einhaltung des Liskovsches Substitutionsprinzips