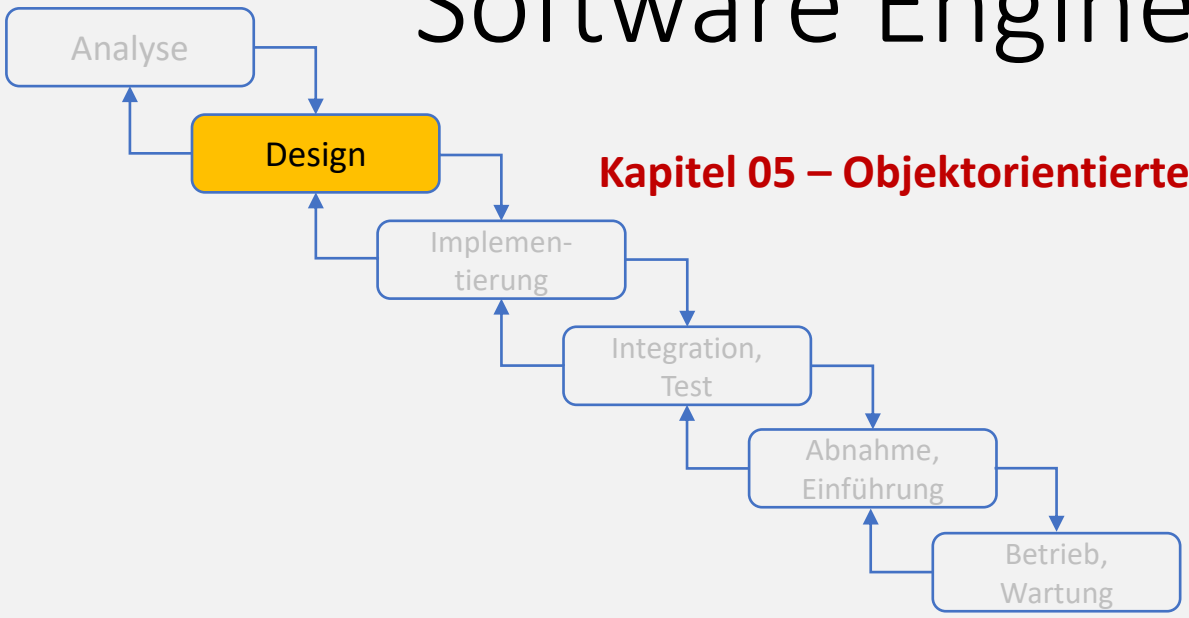
# Software Engineering



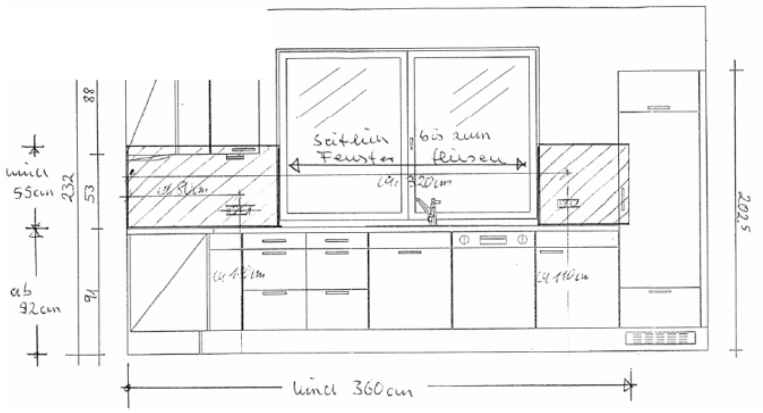

## Kapitel 05 – Objektorientiertes Design (OOD)

Architekturprinzipien

OSTBAYERISCHE TECHNISCHE HOCHSCHULE REGENSBURG
INFORMATIK UND MATHEMATIK

# Gliederung

1. Motivation
2. Softwarearchitektur
3. Architektursichten
   - nach Kruchten
   - nach Starke
4. Architekturprinzipien (Kopplung, Kohäsion, DRY, OCP, …)
5. Architekturmuster (Schichten, Pipes&Filters, Blackboard, …)




# Architektur

# Einordnung in den Gesamtkontext

- Requirements (WAS soll umgesetzt werden)
  - Grobe Beschreibung
  - Sprache des Anwenders
  - System als Black-Box
- OOA (WAS soll umgesetzt werden)
  - Detaillierte Beschreibung
  - Verfeinerung von fachlichen Begriffen/Ideen (konzeptuelle Klassen) und deren Beziehungen → Domänenmodell
  - Ablauf auf Systemebene → dynamisches Verhaltensmodell
  - System als Black-Box
- OOD (WIE soll Umsetzung erfolgen)
  - Erarbeiten und Verfeinern der Lösung
  - System wird Stück für Stück zur White-Box
  - Lösungselemente sind Objekte
  - Ziel: Lösung so weit beschreiben, dass Entwickler keine ad-hoc-Entscheidungen treffen muss

# Wozu der Aufwand?

**Merkmale schlechter Software:**

- steif (rigid):
  - Selbst kleine Änderungen sind schwierig/aufwendig/riskant
  - State-of-the-art-Funktionen bleiben unimplementiert
  - Symptome: nicht-kritische Fehler bleiben unbeseitigt
- zerbrechlich (fragile):
  - Kleine Fehler lösen Fehler in entfernten Softwarebereichen aus
  - Symptome: Großteil der Entwicklungsarbeit fließt in Bugfixing

# Wozu der Aufwand?

**Merkmale schlechter Software:**

- unbeweglich (immobile):
  - Viele Querverbindungen/Abhängigkeiten
  - Software nicht einmal in Teilen wiederverwendbar
  - Symptome : parallele Produktlinien mit ähnlicher Funktionalität und unabhängiger Codebasis
- zäh (viscous):
  - Änderungen gegen den Entwurfsgedanken sind leichter umsetzbar als Änderungen im Sinne des Entwurfsgedankens
  - Symptome: nicht genutzter und redundanter Code, irrelevante Dokumentation

**Alle diese negativen Entwurfseigenschaften sind problemlos in OO-Sprachen umsetzbar!**

# Wie kann man die schlechten Eigenschaften vermeiden?

**Was macht einen guten OO-Entwurf aus:**

- Einhaltung von Prinzipien:
  - SOLID: Single Responsibility, Open/Closed Principle, LSP, Interface Segregation, Dependency Inversion
  - GRASP: General Responsibility Assignment Patterns (Menge von Entwurfsmustern)
  - …
- Die Prinzipien führen auf eine Struktur der Lösung
  - Softwarearchitektur
  - Softwaredesign

# Weitere Kriterien für “guten” Entwurf

- Korrektheit
  - Erfüllung der Anforderungen
  - Wiedergabe aller Funktionen des Systemmodells
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlichkeit & Präzision
  - Gute Dokumentation
- Anpassbarkeit
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung

→ Diese Kriterien gelten auf allen Ebenen des Entwurfs (Architektur, Subsysteme, Komponenten).

# Kohäsion

**Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente**

- Hohe Kohäsion einer Komponente erleichtert:
  - Verständnis, Wartung und Anpassung
- Hohe Kohäsion wird erreicht durch:
  - Prinzipien der Objektorientierung (Datenkapselung)
  - Einhaltung von Regeln zur Paketbildung
  - Verwendung geeigneter Muster zu Kopplung und Entkopplung

# Kopplung

**Kopplung ist ein Maß für die Abhängigkeiten zwischen Komponenten**

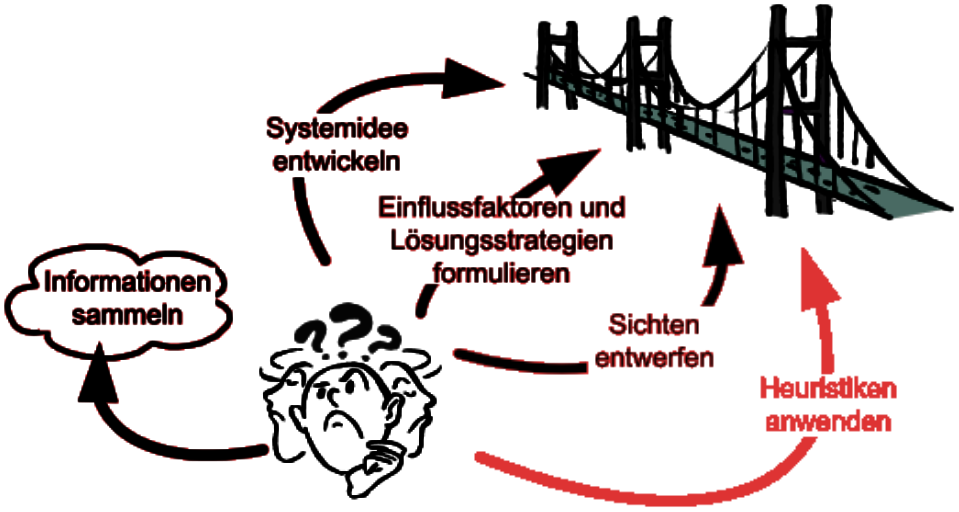
- Geringe Kopplung erleichtert die Wartbarkeit und macht Systeme stabiler
- Arten der Kopplung:
  - Datenkopplung (gemeinsame Daten)
  - Schnittstellenkopplung (gegenseitiger Aufruf)
  - Strukturkopplung (gemeinsame Strukturelemente)
- Reduktion der Kopplung:
  - Kopplung kann nie auf Null reduziert werden!
  - Schnittstellenkopplung ist akzeptabel, da höhere Flexibilität
  - Datenkopplung vermeiden!
    - aber durch Objektorientierung meist gegeben
  - Strukturkopplung vermeiden!
    - z.B. keine Vererbung über Paketgrenzen hinweg
- Entkopplungsbeispiel: getter/setter-Methoden statt direkter Attributzugriff

# Interne Wiederverwendung

**Interne Wiederverwendung (reuse) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten**

- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - Im objektorientierten Entwurf: Vererbung, Parametrisierung
  - Im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
- **Aber: Wiederverwendung kann die Kopplung erhöhen:**
  - Schnittstellenkopplung und Strukturkopplung

# Vorgehen beim Architekturentwurf




- Typisch beim Architekturentwurf: heuristisches Vorgehen

**Heuristik:**
(altgr. εὑρίσκω *heurísko* „ich finde“; von εὑρίσκειν *heuriskein* ‚auffinden‘, ‚entdecken‘) bezeichnet die Kunst, mit begrenztem Wissen (unvollständigen Informationen) und wenig Zeit zu guten Lösungen zu kommen.

[Quelle: Wikipedia]

# Heuristiken effektiver Softwarearchitektur

- Erfolg wird vom Kunden definiert, nicht vom Architekten.
- Mit den schwierigsten Teilen beginnen (enthalten das höchste Risiko)
  - Prototypen für diese Teile des Systems erstellen
- Ursprüngliche Anforderungen nicht notwendigerweise die richtigen:
  - Endgültigen Anforderungen sind ein Ergebnis der Architektur
  - Anforderungen hinterfragen und auf Konsequenzen und Alternativen hinweisen
  - Auftraggebern und Projektleitung dabei helfen, Aufwand und Nutzen der Anforderungen abzuwägen
- Ein Modell ist keine Realität:
  - Ihre Modelle möglichst gemeinsam mit Kunden und Auftraggebern verifizieren
- Ihr Entwurf ist perfekt? Dann haben Sie ihn noch niemandem gezeigt:
  - Frühzeitig mit anderen Projektbeteiligten über Alternativen und Konsequenzen Ihrer Entwurfsentscheidungen diskutieren
- Entwerfen (und behalten) Sie Alternativen und Optionen so lange wie möglich in Ihren Entwürfen. Sie werden sie brauchen.

[Quelle: Effektive Software Architekturen, G. Starke]

# Von der Idee zur Struktur: Wie gehe ich vor?

**Probleme zerlegen:**

- „In Scheiben schneiden“:
  - Zerlegen Sie ein System in (horizontale) Schichten
  - Jede Schicht stellt einige klar definierte Schnittstellen zur Verfügung und
  - Nutzt Dienste von darunter liegenden Schichten
- „In Stücke schneiden“:
  - Zerlegen Sie ein System in (vertikale) Teile
  - Jeder Teil übernimmt eine bestimmte fachliche oder technische Funktion
- Kapselung:
  - Teile (Komponenten) als Black-Box betrachten
  - Mit der Umgebung über klar definierte Schnittstellen kommunizieren

# Zerlegen: aber wie?

- Verwenden Sie etablierte und erprobte Strukturen wieder. Erfinden Sie möglichst wenig neu, sondern halten Sie sich an Bewährtes.
- Entwerfen Sie in Iterationen. Überprüfen Sie die Stärken und Schwächen eines Entwurfes anhand von Prototypen oder technischen Durchsichten.
- Bewerten Sie diese Versuche explizit und überarbeiten daraufhin Ihre Strukturen – wenden Sie es beim Entwurf von Strukturen intensiv und dauernd an.
- Dokumentieren Sie die Entscheidungen, die zu einer bestimmten Struktur führen. Andere Projektbeteiligte werden sie künftig verstehen müssen.

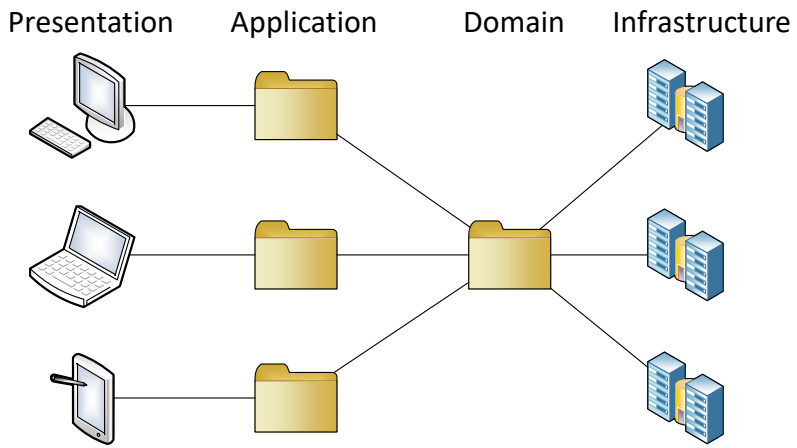
# Fachmodelle als Basis der Entwürfe

- Beginnen Sie den Entwurf mit der Strukturierung der Fachdomäne.
- Modellieren Sie in der Sprache der Fachdomäne
- Isolieren Sie die Fachdomäne in der Architektur

# Isolierung der Fachdomäne nach Evans

| | |
|---|---|
| User Interface<br>(Presentation Layer) | • Darstellung und Informationsanzeige<br>• Nimmt Eingaben und Kommandos von Nutzern entgegen |
| Application Layer | • Beschreibt oder koordiniert Geschäftsprozesse,<br>• Delegiert an den Domain Layer oder den Infrastructure Layer |
| Domain Layer<br>(Fachmodell-Schicht) | • Kern von DDD (Domain Driven Design)! Schicht repräsentiert Fachdomäne<br>• Hier "lebt" das Modell mit seinem aktuellen Zustand<br>• Persistenz seiner Entitäten delegiert diese Schicht an Infrastructure Layer |
| Infrastructure Layer | • Allgemeine technische Services (z.B. Persistenz, Kommunikation mit anderen Systemen) |

[Quelle: Domain Driven Design, Eric Evans]



# Verwaltung der Domänenobjekte nach Evans

- **Aggregate:**

  Sie kapseln vernetzte (d.h. miteinander assoziierte) Domänenobjekte. Ein Aggregat hat grundsätzlich eine einzige Entität als Wurzelobjekt. Diese Wurzel ist der einzige Einstiegspunkt in das Aggregat, sämtliche mit der Wurzel verbundene Domänenobjekte sind lokal. Objekte von außen dürfen nur Referenzen auf die Wurzelentität halten.

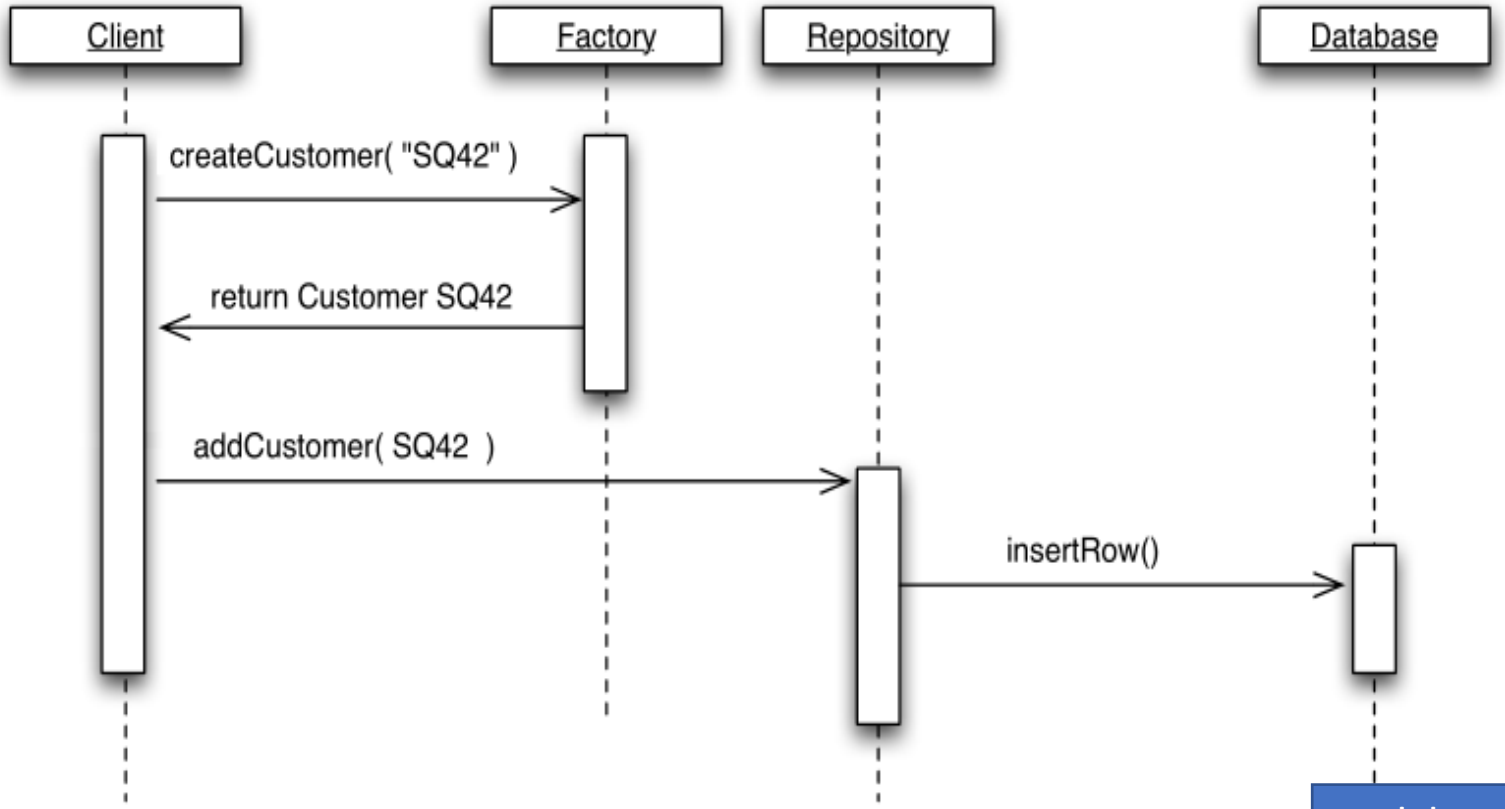
- **Factories:**

  Entities und insbesondere Aggregate können komplexe Strukturen vernetzter Objekte bilden, die Sie nicht über triviale Konstruktoraufrufe erzeugen können oder wollen. Verwenden Sie Factories, um die Erzeugung von Aggregaten und Entitäten zu kapseln. Factory-Objekte arbeiten ausschließlich innerhalb der Domäne und haben keinen Zugriff auf den Infrastruktur-Layer.

- **Repositories:**

  Alle Arten von Objekten (sowohl aus dem Domain Layer wie auch dem Application Layer) benötigen eine Möglichkeit, die Objektreferenzen anderer Objekte zu erhalten. Repositories kapseln die technischen Details der Infrastrukturschicht gegenüber den Domänenobjekten. Dadurch bleibt das Domänenmodell auch in dieser Hinsicht „technologiefrei“. Repositories beschaffen beispielsweise Objektreferenzen von Entitäten, die aus Datenbanken gelesen werden

# Zusammenspiel: Factory, Repository, Infrastruktur




Welche Konvention wird verletzt?

# Architekturprinzipien

# Principles vs Patterns (Gebot vs Rezept)

- **Principle (Prinzip):** "Was Du nicht willst, das man Dir tut, …"
  - Abstrakte Handlungsanweisung (z.B.: DRY)
  - Auf einen „höheren" Zweck gerichtet (Gebot)
  - Kann auch ein (nicht-konstruktives) Bewertungskriterium sein
- **Pattern (Muster):** „Wenn Dir einer eine Ohrfeige gibt, dann halte auch…"
  - Lösungsschablone (erprobte Lösung für ein verbreitetes Problem)
  - Auf einen konkreten Zweck gerichtet (Rezept: „Wenn DIES, tue DAS")
  - Ist konstruktiv

# Architekturprinzipien: Beispiele

- Lose Kopplung
- Hohe Kohäsion

Hauptprinzipien

- Don’t Repeat Yourself
- Open Closed Principle
- Information Hiding
- Separation of Concerns
- Abhängigkeit nur von Abstraktionen
- Dependency Injection
- Liskovsches Substitutionsprinzip
- Abtrennung von Schnittstellen
- Keine zyklischen Abhängigkeiten
- …

Grundprinzipien

# Hauptprinzipien des Architekturentwurfs

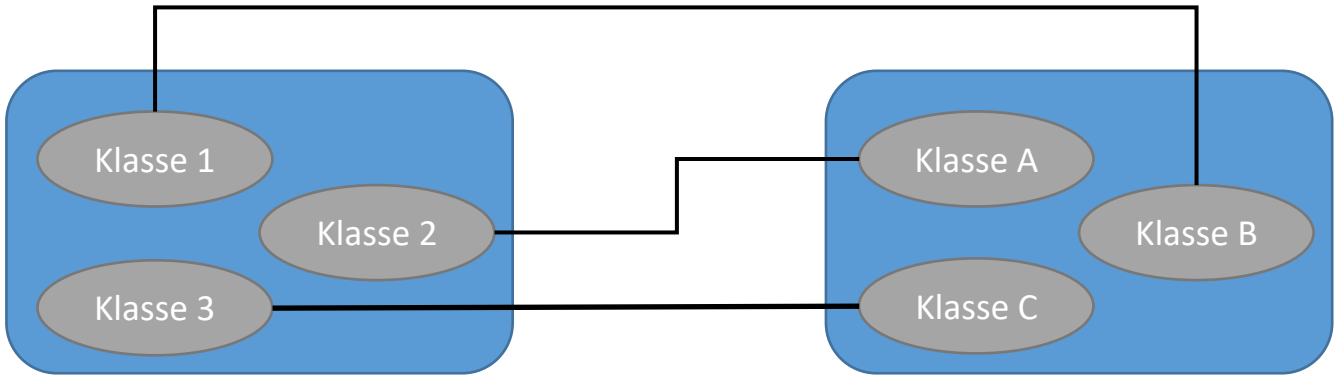
**Eine gute Software-Architektur verfügt über:**

1. Lose/Geringe Kopplung (leichter Austausch von Komponenten)
   - Datenkopplung (gemeinsame Daten)
   - Schnittstellenkopplung (gegenseitiger Aufruf)
   - Strukturkopplung (gemeinsame Strukturelemente)

2. Hohe Kohäsion ("Unteilbarkeit" der Komponenten)

# Hauptprinzip des Architekturentwurfs: Ziel: Lose Kopplung (1)

**Gegenteil: Hohe/Enge Kopplung**



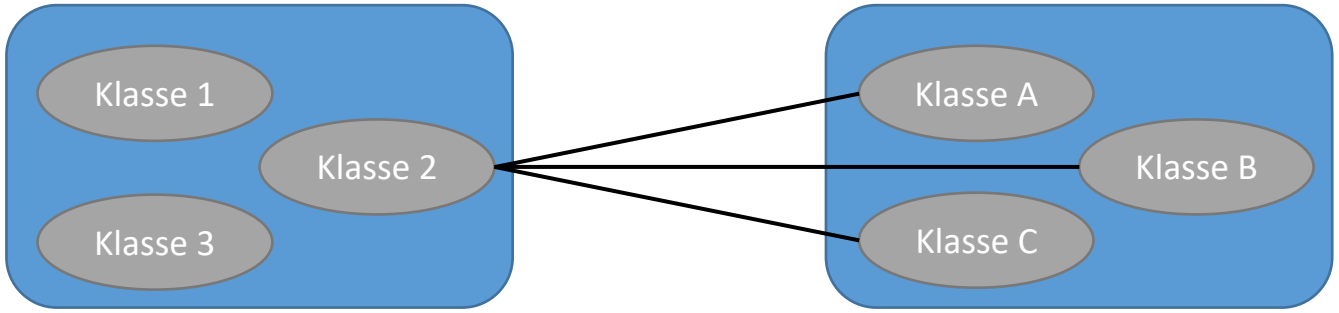

**Hier:**

- Hohe Abhängigkeit zwischen Modulen
- Änderungen an einem Modul haben (unerwünschte) Auswirkungen auf andere Module

# Hauptprinzip des Architekturentwurfs: Ziel: Lose Kopplung (2)

**Ziel: Geringe/Lose Kopplung:**

- Erleichtert Wartbarkeit
- Macht Systeme stabiler

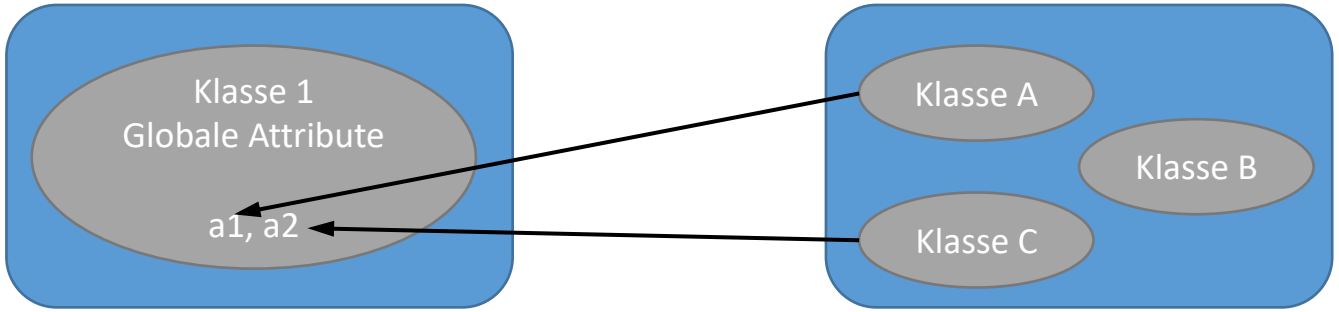



**Hier: Geringe Kopplung**

- Abhängigkeit zu wenigen Elementen des gekoppelten Moduls
- Änderungen an einem Modul haben geringe Auswirkungen auf andere Module

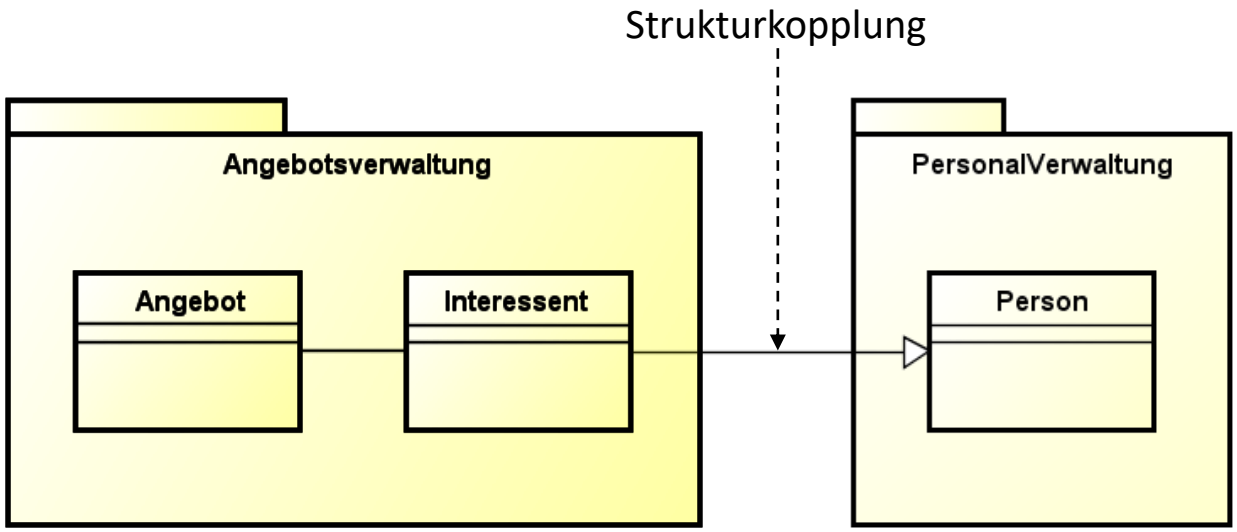
# Datenkopplung

**Negativbeispiel:**




- Zugriff auf interne Datenstrukturen einer Komponente
- Problematisch bei Veränderungen
- Allgemein: Einhaltung von Bedingungen, die immer erfüllt sein müssen, sind kritisch

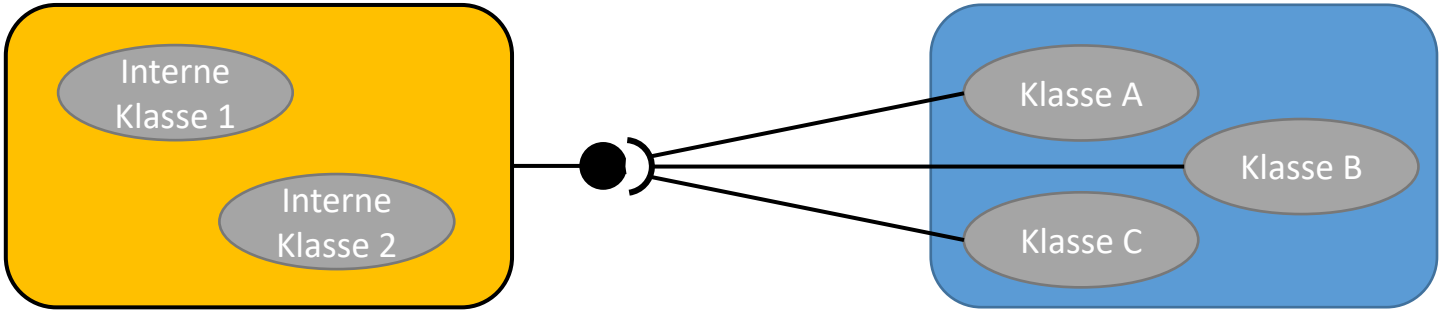
# Strukturkopplung




**Diskussion:**

- Modul ist abhängig von Strukturelementen eines anderen Moduls
- Subsysteme sollen unabhängig voneinander austauschbar/erstellbar sein
- Strukturkopplung hat hohe Abhängigkeit zur Folge
  (→ Nachteilig für Stabilität des Systems)

# Schnittstellenkopplung




**Diskussion:**

- Zugriff nur über wohldefinierte Schnittstellen (Verbergen von Impl.details)
- Diese Kopplung ist akzeptabel (interne Änderungen haben keine Auswirkungen auf andere Module)
- Problematisch: Schnittstellenveränderungen

# Fazit: Kopplung

## Ziel: Reduktion von Kopplung

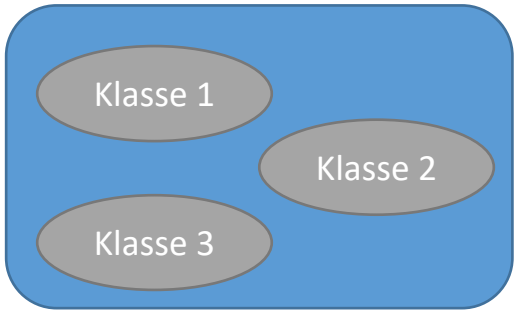
- Kopplung kann nie auf Null reduziert werden
- Schnittstellenkopplung ist akzeptabel
- Datenkopplung vermeiden
- Strukturkopplung vermeiden (z.B. keine Vererbung über Paketgrenzen)

## Generelles Ziel:

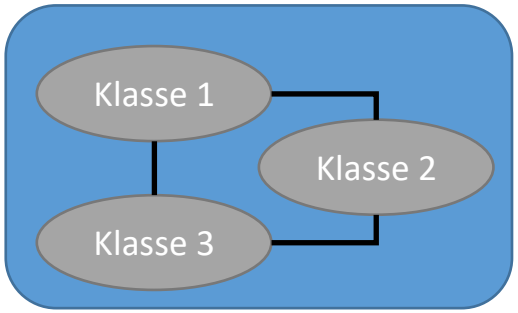
- Reduktion der Kopplung zwischen verschiedenen Modulen

# Hauptprinzip des Architekturentwurfs: Ziel: Hohe Kohäsion

- **Kohäsion = Zusammenhalt**
- Dinge in Struktureinheiten zusammenfassen, die inhaltlich zusammen gehören
- Kohäsion ist ein Maß für die Zusammengehörigkeit der Bestandteile einer Komponente
- Hohe Kohäsion einer Komponente erleichtert Verständnis, Wartung und Anpassung




- Geringe Kohäsion
- Geringe Abhängigkeiten innerhalb der Komponente




- Hohe Kohäsion
- Klassen erfüllen eine gemeinsame Aufgabe

# Hauptprinzip des Architekturentwurfs: Ziel: Wie erreiche ich hohe Kohäsion?

**Ziel: Erreichung hoher Kohäsion durch:**

- Prinzipien der Objektorientierung (Datenkapselung)
- Einhaltung von Regeln zur Paketbildung
- Verwendung geeigneter Design-Patterns zu Kopplung und Entkopplung (s. Kap. 06b)

**Generelles Ziel:**

- Hohe Kohäsion innerhalb eines Moduls oder Subsystems

# Grundprinzipien “guter” Architektur: das DRY-Prinzip

**DRY Principle: Don‘t Repeat Yourself**

- **Wiederholungen:**
  - Machen Arbeit
  - Werden irgendwann inkonsistent
  - Erschweren das Verständnis
  - Verschlechtern Performance beim Kompilieren und zur Laufzeit

→ **Vermeide Wiederholungen** auf allen Ebenen der Softwareerstellung:
  - Code
  - Dokumentation
  - Tests
  - Datenbankschemata
  - …

# Grundprinzipien “guter” Architektur: Beispiel: Wiederholung im Code (1)

```
Bruch add(Bruch b1, Bruch b2)
{
    Bruch ergebnis = new Bruch();

    if (b1.nenner == b2.nenner) {
        ergebnis.zaehler = b1.zaehler + b2.zaehler;
        ergebnis.nenner = b1.nenner;
    }
    else
    {
        ergebnis.zaehler = b1.zaehler * b2.nenner + b2.zaehler * b1.nenner;
        ergebnis.nenner = b1.nenner * b2.nenner;
    }

    return ergebnis;
}
```

# Grundprinzipien “guter” Architektur: Beispiel: Wiederholung im Code (2)

```
Bruch sub(Bruch b1, Bruch b2)
{
    Bruch ergebnis = new Bruch();

    if (b1.nenner == b2.nenner) {
        ergebnis.zaehler = b1.zaehler - b2.zaehler;
        ergebnis.nenner = b1.nenner;
    }
    else
    {
        ergebnis.zaehler = b1.zaehler * b2.nenner - b2.zaehler * b1.nenner;
        ergebnis.nenner = b1.nenner * b2.nenner;
    }

    return ergebnis;
}
```

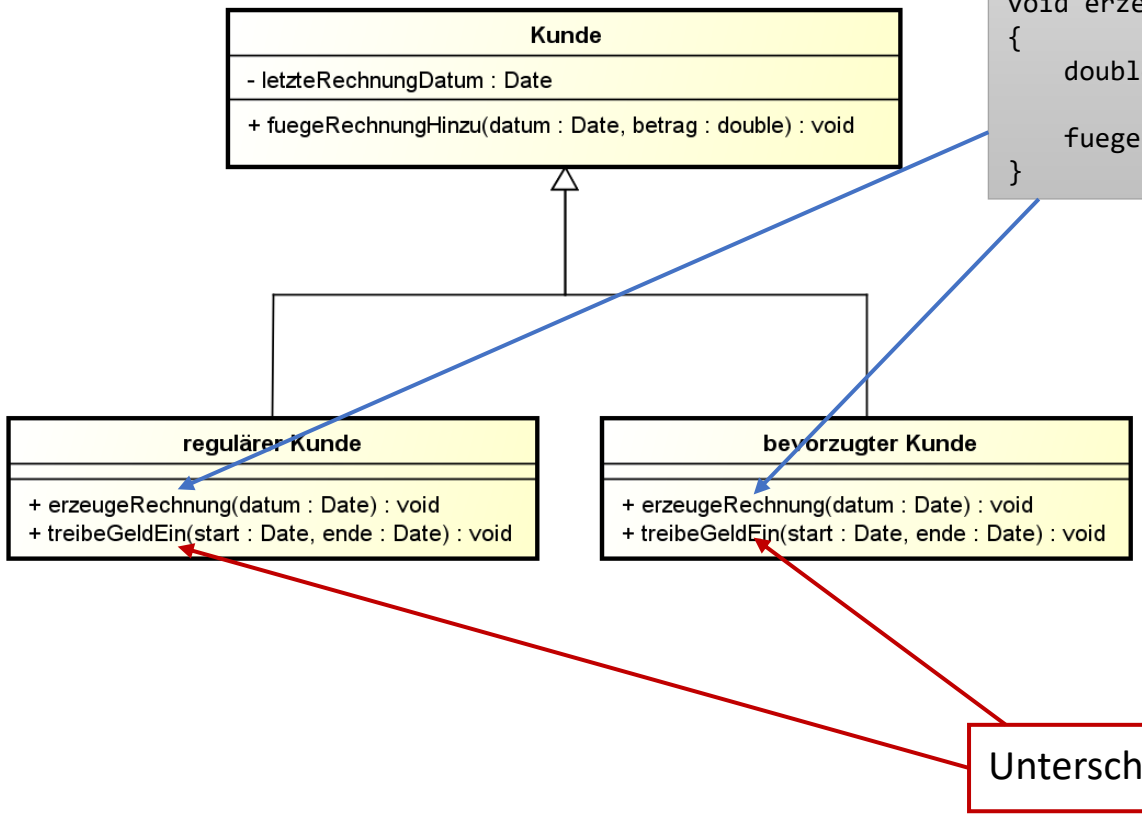
Unterschiede zu add(…)

Lösung?

# Beispiel: Wiederholung bei Vererbungsbeziehungen

Gleicher Code:

```
void erzeugeRechnung(Date datum)
{
    double kosten = treibeGeldEin(letzteRechnungDatum,
                                  datum);
    fuegeRechnungHinzu(datum, kosten);
}
```




Lösung?

# Grundprinzipien “guter” Architektur: Open Closed Principle (OCP) (1)

**OCP: Module (= Klassen, Funktionen, Packages, … )**

- Sollen für Erweiterungen offen und
- Für Veränderungen/Modifikationen geschlossen sein

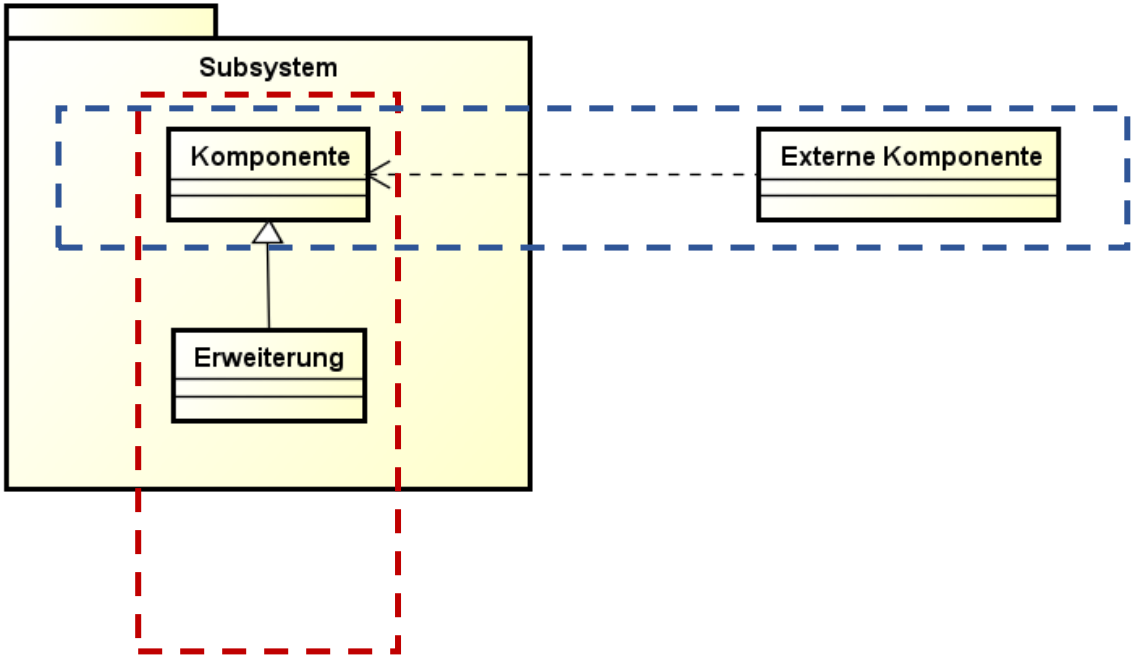
→ Eine **Erweiterung** in diesem Sinne:
  - Verändert das vorhandene Verhalten der Einheit nicht,
  - Vielmehr erweitert es die Einheit um zusätzliche Funktionen oder Daten

→ Eine **Modifikation** hingegen würde das bisherige Verhalten der Einheit ändern

**Ziele:**

- Neue Eigenschaften hinzufügen, ohne bestehenden Code modifizieren zu müssen
- Sicherstellen, dass künftige Erweiterungen ohne Nebenwirkungen funktionieren

# Grundprinzipien “guter” Architektur: Open Closed Principle (OCP) (2)




“Closed” aus Sicht einer nutzenden Komponente

Erweiterbar ohne die “closed”-Eigenschaft zu verletzen

# Grundprinzipien “guter” Architektur: Beispiel: Open Closed Principle (OCP) (1)

- **Ziel:** mit `draw(…)` soll eine geometrische Figur gezeichnet werden:
- **Beispiel:**

```
…
void draw (Form f)
{
  if (f.type == circle)
    drawCircle(f);
  else if (f.type == square)
    drawSquare(f);
  …
}
```

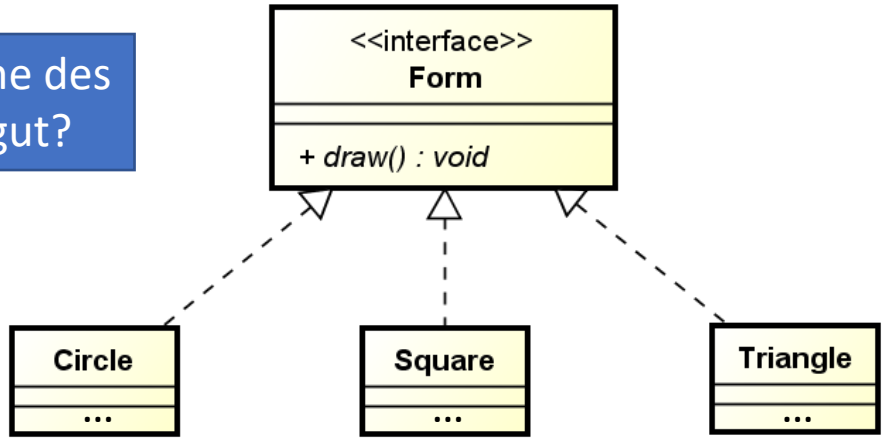
→ Schlechtes Code-Fragment! Warum?

→ Was passiert, wenn eine neue Klasse zu zeichnender Figuren hinzukommt?

# Grundprinzipien “guter” Architektur: Beispiel: Open Closed Principle (OCP) (2)

- **Ziel:** mit `draw(…)` soll eine geometrische Figur gezeichnet werden:
- **Besser:**

Ist der Name des Interfaces gut?




```
// in Klasse Circle
void draw ()
{
    // draw Circle
    …
}
// in Klasse Square
void draw ()
{
    // draw Square
    …
}
```

→ Warum ist die Lösung besser?

- Zeichnen von Objekten in Schnittstelle modelliert, in Klassen implementiert
- Aufrufendes System ist nur von Schnittstelle abhängig

→ Was passiert, wenn eine neue Klasse zu zeichnender Figuren hinzukommt?

# Grundprinzipien “guter” Architektur: Open Closed Principle OCP

**Generelle Anmerkungen zu OCP:**

- Schlüssel zu OCP ist Polymorphie (und dynamische Bindung)
- Klassen können durch Ableitung zusätzliches Verhalten bereitstellen, ohne die Basisklasse zu verändern
- OO-Sprachen machen Umsetzung des OCP daher leicht

- Was tut man bei nicht-OO-Sprachen?

→ OCP im Konzept verwenden

# Grundprinzipien “guter” Architektur: Interne Wiederverwendung

**Interne Wiederverwendung (reuse) ist ein Maß für die Ausnutzung von Gemeinsamkeiten zwischen Komponenten**

- Reduktion der Redundanz
- Erhöhung der Stabilität und Ergonomie
- Hilfsmittel für Wiederverwendung:
  - Im objektorientierten Entwurf: Vererbung, Parametrisierung
  - Im modularen und objektorientierten Entwurf: Module/Objekte mit allgemeinen Schnittstellen (Interfaces)
- Aber: Wiederverwendung kann die Kopplung erhöhen:
  - Schnittstellenkopplung und Strukturkopplung

**Generelles Ziel:**

- Hohe Wiederverwendbarkeit (aber nicht um jeden Preis! )

# Grundprinzipien “guter” Architektur: Information Hiding (Geheimhaltungsprinzip)

**Information Hiding (D. Parnas):**

- Verbergen der Funktionsweise eines Systemteils im Inneren des Moduls
- Nach außen nur Informationen bekannt, die über Schnittstelle explizit zur Verfügung gestellt werden
- Schnittstelle gibt so wenig wie möglich nach außen bekannt

- Umsetzung mithilfe der Objektorientierung:
  - Kapselung ≠ Information Hiding
    - Kapselung: Attribute und Methoden sinnvoll zusammengefügt
    - Information Hiding: Innere Struktur geht Clients nichts an und Änderungen daran betreffen sie nicht
  - Zugriff auf Objektzustand nur über wohldefinierte Methoden
  - Objekt selbst ist verantwortlich und kann über die Zulässigkeit entscheiden

# Kapselung vs. Information Hiding
# Beispiel: “GPS-Koordinaten” (1)

**Beispiel: “GPS-Koordinaten”**

```
public class Position
{
  public double latitude;
  public double longitude;
}


public class PositionUtility
{
  public static double distance( Position position1, Position position2 )
  {
    // Calculate and return the distance between the specified positions.
  }
  public static double heading( Position position1, Position position2 )
  {
    // Calculate and return the heading from position1 to position2.
  }
}
```

Klasse für Koordinaten

Klasse mit statischen Methoden
für Abstands- und Richtungsberechnungen

[Quelle: www.javaworld.com]

# Beispiel: “GPS-Koordinaten” (2) Mangelnde Kapselung im Client-Code

**Client-Code (Variante 1):**

```
// Create a Position representing my house
Position myHouse = new Position();
myHouse.latitude =    36.538611;
myHouse.longitude = -121.797500;
// Create a Position representing a local coffee shop
Position coffeeShop = new Position();
coffeeShop.latitude =    36.539722;
coffeeShop.longitude = -121.907222;
// Use a PositionUtility to calculate distance and heading from my house to the local
// coffee shop.
double distance = PositionUtility.distance( myHouse, coffeeShop );
double heading  = PositionUtility.heading(  myHouse, coffeeShop );
// Print results
System.out.println ( "From my house at (" + myHouse.latitude + ", " + myHouse.longitude +
    ") to the coffee shop at (" + coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance + " at a heading of " + heading + " degrees." );
```

# Beispiel: “GPS-Koordinaten” (3) Kapselung

**Bessere Variante (aber immer noch nicht gut!):**

```
public class Position
{
  public double distance( Position position )
  {
    // Calculate and return the distance from this object to the specified position.
  }
  public double heading( Position position )
  {
    // Calculate and return the heading from this object to the specified position.
  }
  public double latitude;
  public double longitude;
}
```

# Beispiel: “GPS-Koordinaten” (4)

**Client-Code (Variante 2):**

```
// Create a Position representing my house
Position myHouse = new Position();
myHouse.latitude =    36.538611;
myHouse.longitude = -121.797500;
// Create a Position representing a local coffee shop
Position coffeeShop = new Position();
coffeeShop.latitude =    36.539722;
coffeeShop.longitude = -121.907222;
// Use a PositionUtility to calculate distance and heading from my house to the local
// coffee shop.
double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );
// Print results
System.out.println ( "From my house at (" + myHouse.latitude + ", " + myHouse.longitude +
    ") to the coffee shop at (" + coffeeShop.latitude + ", " + coffeeShop.longitude +
    ") is a distance of " + distance + " at a heading of " + heading + " degrees." );
```

# Beispiel: “GPS-Koordinaten” (5) Zwischenfazit

**Wdh. Kapselung:** **Technik, die Daten mit zugehörigen Operationen bündelt**

- **Problem:**
  - Nicht garantiert: Schutz der Daten
  - Nicht garantiert: Information Hiding
- **Ergebnis:**
  - Jeder Client der Klasse `position` kann die Informationen `latitude`, `longitude` ändern
  - Aktuelles Design ist nicht ausreichend!

- **Zwischenfazit:** Kapselung allein ist nicht ausreichend

# Beispiel: “GPS-Koordinaten” (6)

**Erster Lösungsansatz:**

- Membervariablen von Klasse `Position` verstecken
- Über Konstruktor setzen lassen

```
// Create a Position representing my house
Position myHouse = new Position( 36.538611, -121.797500 );
// Create a Position representing a local coffee shop
Position coffeeShop = new Position( 36.539722, -121.907222 );
// Use a PositionUtility to calculate distance and heading from my house to the local
// coffee shop.
double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );
// Print results
System.out.println ( "From my house at (" + myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" + coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance + " at a heading of " + heading + " degrees." );
```

# Beispiel: “GPS-Koordinaten” (7)

## Innere Daten schützen, Wertebereiche beachten

```
public class Position
{
  public Position( double latitude, double longitude ) {
    setLatitude( latitude );
    setLongitude( longitude );
  }
  public void setLatitude( double latitude ) {
    // Ensure -90 <= latitude <= 90 using modulo arithmetic. (Code not shown)
    this.latitude = latitude;
  }
  public void setLongitude( double longitude ) {
    // Ensure -180 < longitude <= 180 using modulo arithmetic. (Code not shown)
    this.longitude = longitude;
  }
  public double getLatitude() {
    return latitude;
  }
  public double getLongitude() {
    return longitude;
  }
  public double distance( Position position ) { ... }
  public double heading ( Position position ) { ... }

  private double latitude;
  private double longitude;
}
```

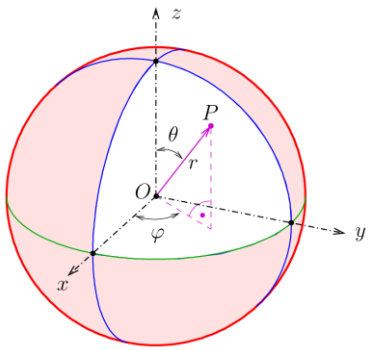
# Potentielle Änderungen isolieren

**Änderungen an der internen Struktur einer Klasse sollen keine Client-Klassen betreffen**

Welche neuen Anforderungen könnten eintreffen?

- **Beispiel:**
  - In Positions-Klasse keinerlei Informationen von Einheiten (m, km, miles, etc.)
  - Geometrie der Erde wurde nicht berücksichtigt (lokal relativ irrelevant, global nicht)
- **Lösung?**
  - Einführen von privaten Membern der Interfaces Units, Geometry
  - Units: mit Implementierungen für z.B. `Kilometer`, `Miles`, `NauticalMiles`, …
  - Geometry: mit Implementierungen für z.B. `PlaneGeometry`, `SphericalGeometry`, …
  - Austausch von `latitude` und `longitude` durch Kugelkoordinaten $r$, $\varphi$, $\theta$

    (Aufwendig, falls Clients auf lat., long. zugreifen)



# Beispiel: “GPS-Koordinaten” (8)
# Klasse mit Information Hiding

```
public class Position
{
  private double phi;
  private double theta;
  private Geometry geometry;
  private Units units;

  public Position( double latitude, double longitude ) {
    setLatitude( latitude );
    setLongitude( longitude );
    // Default to plane geometry and kilometers
    geometry = new PlaneGeometry();
    units = new Kilometers();
  }
  public void setLatitude( double latitude ) {
    setPhi( Math.toRadians( latitude ) );
  }
  public void setLongitude( double longitude ) {
    setTheta( Math.toRadians( longitude ) );
  }
  public void setPhi( double phi ) {
    // Ensure -pi/2 <= phi <= pi/2 using modulo arithmetic. (Code not shown)
    this.phi = phi;
  }
  public void setTheta( double theta ) {
    // Ensure -pi < theta <= pi using modulo arithmetic. (Code not shown)
    this.theta = theta;
  }
```

# Beispiel: “GPS-Koordinaten” (9) Klasse mit Information Hiding

- **Wichtig:**
  - Methoden nach Außen unverändert (trotz internem $\varphi$, $\theta$)
  - Client-Code nicht betroffen

Wie hieß dieses Prinzip noch einmal?

```
  // Setters for geometry and units not shown
  public double getLatitude() {
    return( Math.toDegrees( phi ) );
  }
  public double getLongitude() {
    return( Math.toDegrees( theta ) );
  }
  // Getters for geometry and units not shown
  public double distance( Position position ) {
    // Calculate and return the distance from this object to the specified
    // position using the current geometry and units.
  }
  public double heading( Position position ) {
    // Calculate and return the heading from this object to the specified
    // position using the current geometry and units.
  }
}
```

# Beispiel: “GPS-Koordinaten” (10)
# Klasse mit Information Hiding

**Client-Code (Variante 3):**

```
Position myHouse = new Position( 36.538611, -121.797500 );
Position coffeeShop = new Position( 36.539722, -121.907222 );
double distance = myHouse.distance( coffeeShop );
double heading = myHouse.heading( coffeeShop );
System.out.println ( "Using " + myHouse.getGeometry() + " geometry, " +
    "from my house at (" + myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" + coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance + " " + myHouse.getUnits() +
    " at a heading of " + heading + " degrees." );
```

```
myHouse.setGeometry( Geometry.SPHERICAL );
myHouse.setUnits( Units.STATUTE_MILES );
distance = myHouse.distance( coffeeShop );
heading  = myHouse.heading(  coffeeShop );
System.out.println ( "Using " + myHouse.getGeometry() + " geometry, " +
    "from my house at (" + myHouse.getLatitude() + ", " + myHouse.getLongitude() +
    ") to the coffee shop at (" + coffeeShop.getLatitude() + ", " + coffeeShop.getLongitude() +
    ") is a distance of " + distance + " " + myHouse.getUnits() +
    " at a heading of " + heading + " degrees." );
```

# Beispiel: “GPS-Koordinaten” (10)
# Klasse mit Information Hiding

**Client-Code-Output (Variante 3):**

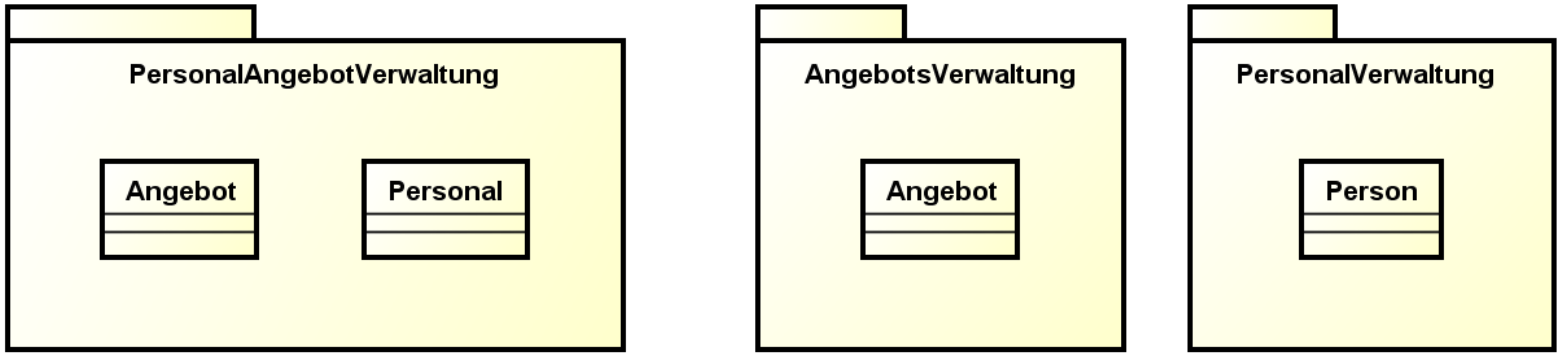
```
Using Plane geometry, from my house at (36.538611, -121.7975)
to the coffee shop at (36.539722, -121.907222) is a distance of
9.79675938972254 Kilometers at a heading of 270.58013375337254
degrees.
Using Spherical geometry, from my house at (36.538611, -121.7975)
to the coffee shop at (36.539722, -121.907222) is a distance of
6.0873776351893385 Statute Miles at a heading of 270.7547022304523
degrees.
```

- **Ergebnis:**
  - Kapselung: Attribute und zugehörige Methoden sinnvoll gebündelt
  - Information Hiding:
    - Innere Struktur der Lieferanten-Klasse für Client unbekannt
    - Änderungen daran betreffen Client nicht (OCP erfüllt)

# Grundprinzipien “guter” Architektur: Separation of Concerns (SoC) (1)

- **Concern:**
  - Menge von Informationen/Wissensbereiche, die den Programmcode beeinflussen
  - Concerns sollten nach Möglichkeit orthogonal zueinander sein
- **Separation of Concerns:**
  - Concerns in spezialisierte Funktionseinheiten (Systemteile) ausgliedern
  - jede Komponente einer Architektur ist nur für eine einzige Aufgabe zuständig
- **Beispiel:**
  - Logging, Transaktionalität, Caching
- **Warum?**
  - Wenn Codeeinheit keine klare Aufgabe hat, ist es schwer sie zu verstehen, zu korrigieren oder zu erweitern
- **Ziel:**
  - Entwicklung und Wartung von Programmen vereinfachen

# Grundprinzipien “guter” Architektur: Separation of Concerns (SoC) (2)




Schlechte Struktur:

- Concerns `Angebot` und `Personal` in einem Subsystem vermischt

Gute Struktur:

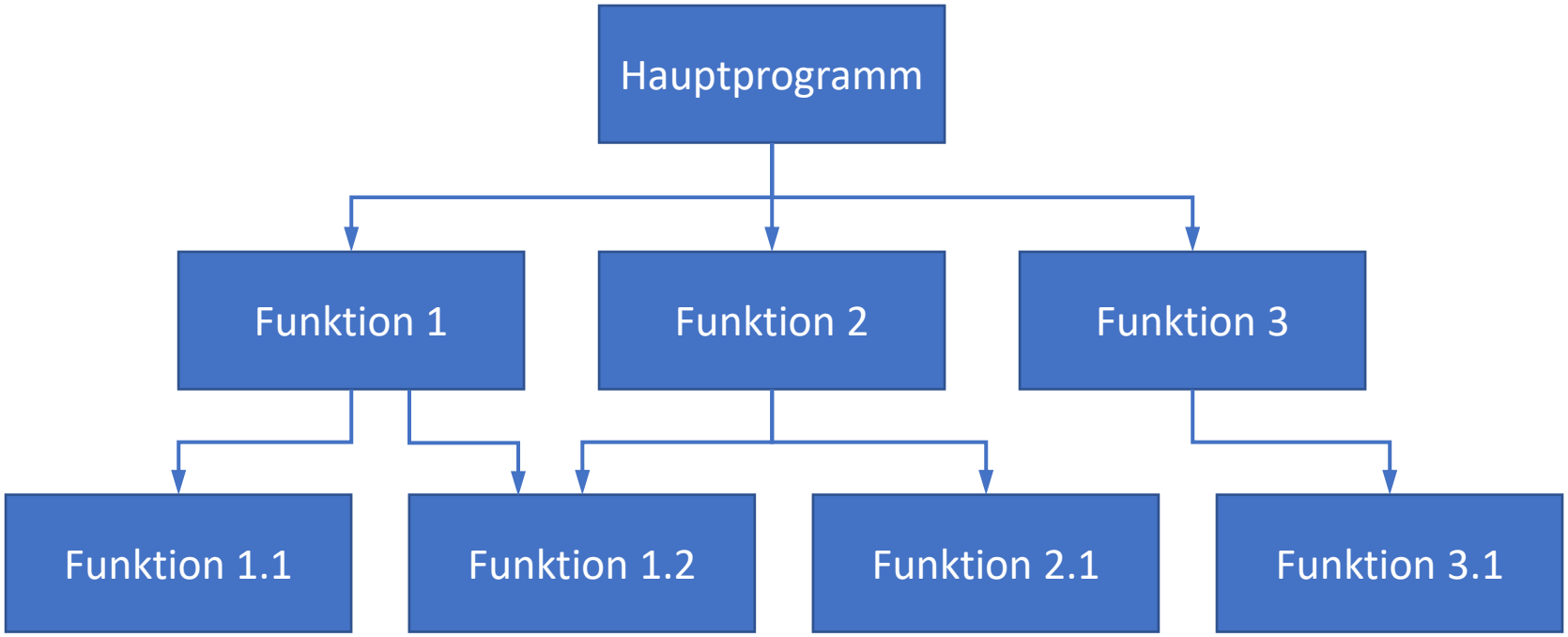
- Concerns `Angebot` und `Personal` sind separiert
- Voneinander unabhängige Entwicklung, Korrektur, Verständnis

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (1)

- **engl.: Dependency Inversion Principle (DIP):**
  - Abhängigkeiten nur von Abstraktionen erlauben, nicht von konkreten Implementierungen
- **Warum?**
  - Hängen Module voneinander ab oder sogar Abstraktionen von Implementierungen, so besteht die Gefahr von:
    - erhöhter Kopplung und
    - zyklischen Abhängigkeiten
- **Ziel:**
  - Invertierung der Abhängigkeiten
  - Module auf höheren Ebenen definieren Schnittstelle, niedrigere realisieren sie
  - Beide hängen von Abstraktionen ab

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (2)

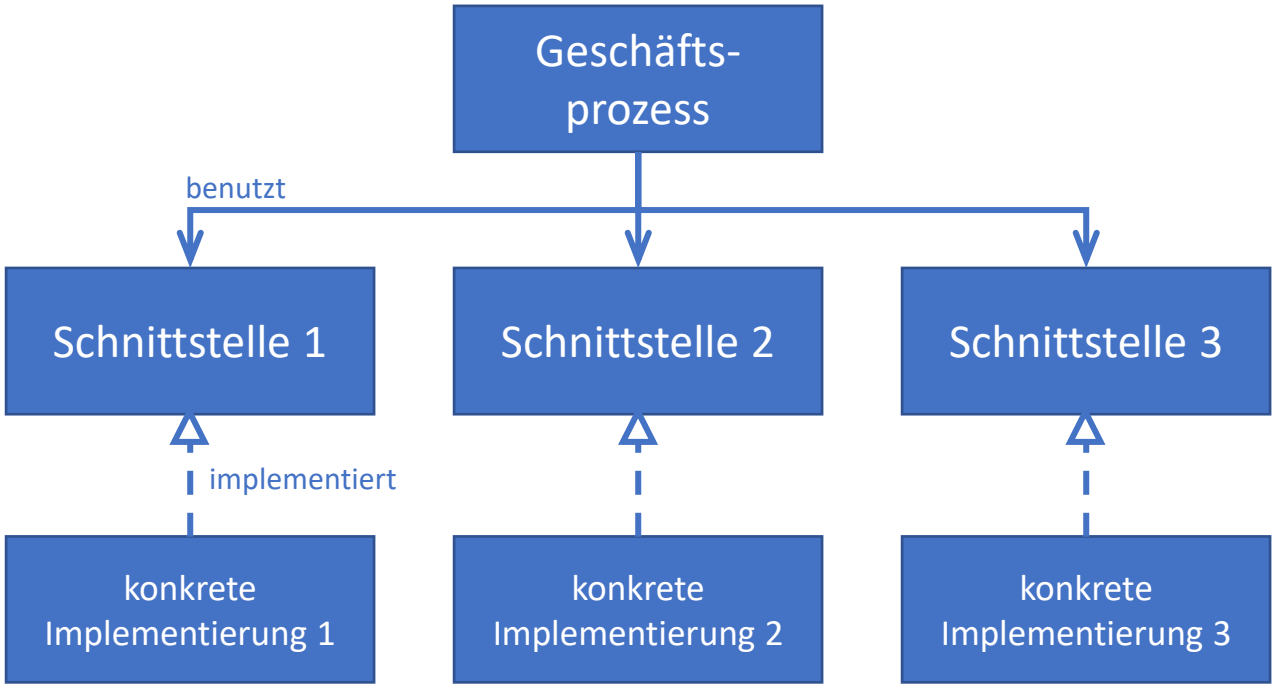
**Typische Abhängigkeiten prozeduraler Softwaresysteme:**




- Fazit: Schlechte Abhängigkeitsstruktur!

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (3)

**Keine Abhängigkeiten von konkreten Implementierungen:**

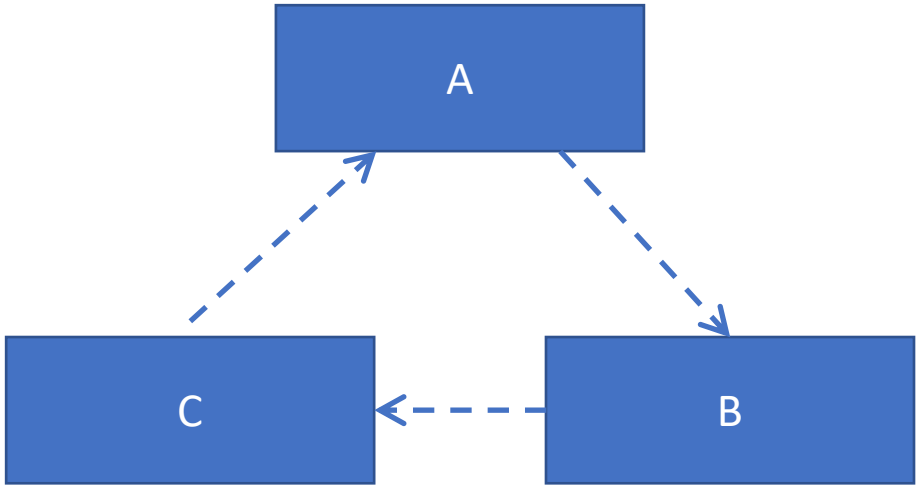



- Fazit: Bessere Abhängigkeitsstruktur! Warum?

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (4)

**Vermeidung zyklischer Abhängigkeiten:**

Waum?




- Problem:
  - Wenn A geändert wird:
    - Müssen B, C evtl. angepasst werden
    - Müssen A, B, C getestet werden
  - Kompilieren kann bereits problematisch werden

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (5)

```
public class Lampe {
   private boolean leuchtet = false;

   public void anschalten() {
      leuchtet = true;
   }

   public void ausschalten() {
      leuchtet = false;
   }
}

public class Schalter {
   private Lampe lampe;
   private boolean gedrueckt;

   public Schalter(Lampe lampe) {
      this.lampe = lampe;
   }

   public void drueckeSchalter() {
      gedrueckt = !gedrueckt;
      if(gedrueckt) {
         lampe.anschalten();
      } else {
         lampe.ausschalten();
      }
   }
}
```

Klassendiagramm?

- Schalter steuert Ablauf
- Benutzt dazu Lampe
- Schalter gehört also zu höherer Ebene
- DIP-Prinzip verletzt, da Schalter abhängig von Lampe!
- Wird Methode von Lampe umbenannt, muss auch Schalter angepasst werden

- Grundlegendes Problem:
  - Schalter arbeitet direkt mit Lampe

- Lösung:
  - Schalter sollte selbst definieren, wie Objekt aussehen sollte, mit dem es arbeitet

# Grundprinzipien “guter” Architektur: Abhängigkeiten nur von Abstraktionen (6)

```
public interface SchalterKlient {
    public void geheAn();
    public void geheAus();
}

public class Schalter {
    private SchalterKlient klient;
    private boolean gedrueckt;

    public Schalter(SchalterKlient klient) {
        this.klient= klient;
    }

    public void drueckeSchalter() {
        gedrueckt = !gedrueckt;
        if(gedrueckt) {
            klient.geheAn();
        } else {
            klient.geheAus();
        }
    }
}
```

Klassendiagramm?

Wie sieht Klasse LampeDIP aus?

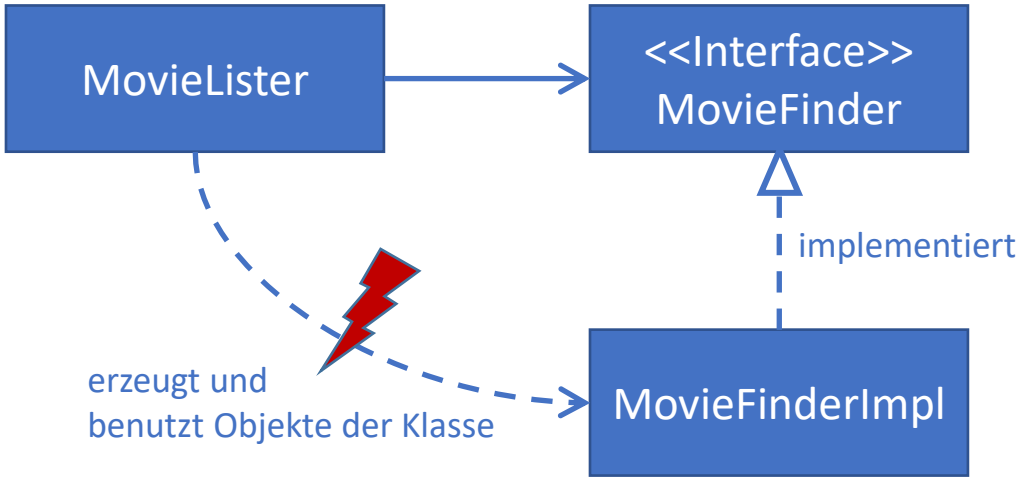
```
public class LampeDIP implements SchalterKlient{
    private boolean leuchtet = false;

    public void geheAn() {
        leuchtet = true;
    }

    public void geheAus() {
        leuchtet = false;
    }
}
```

Fazit:

- SchalterKlient gehört zu selbem Modul wie Schalter (Kohäsion, SoC)
- Spezifischere Module müssen SchalterKlient implementieren
- Umkehrung der Abhängigkeiten wurde erreicht (DIP-Prinzip erfüllt)

# Grundprinzipien “guter” Architektur: Dependency Injection (1)

- **Frage:** Wie bekommt man zur Laufzeit Objekte des benutzten Interfaces? (Informationsbereitstellung zur Laufzeit)
- Wenn MovieLister das selbst regelt, erhält man wieder eine Kopplung der konkreten Klassen

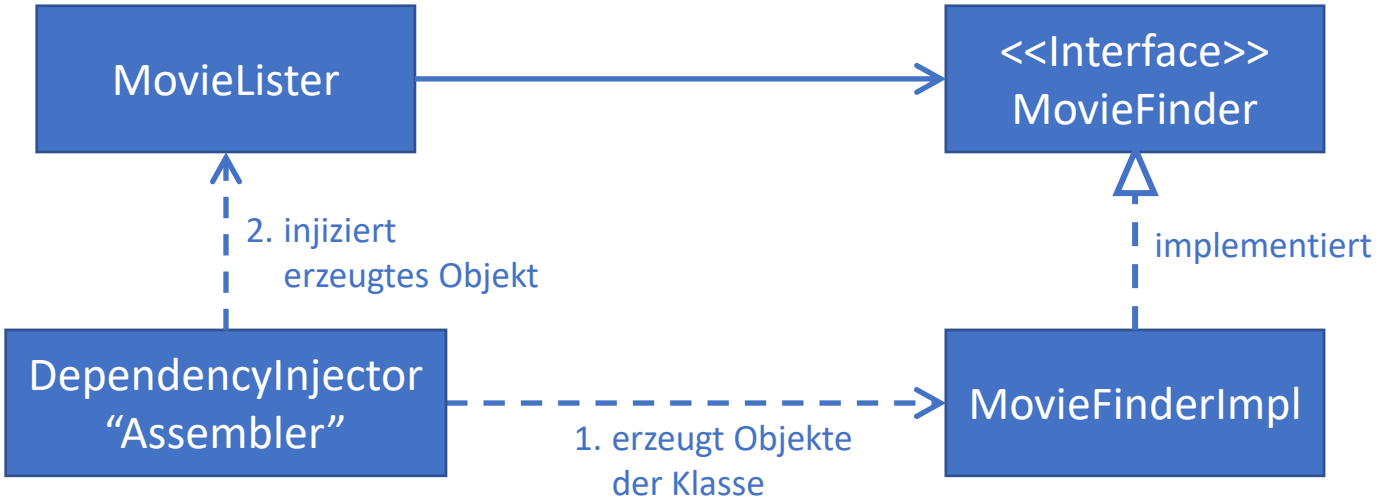



- **Idee:** Objekte zur Laufzeit erzeugen und dem MovieLister übergeben (*inversion of control*)

# Grundprinzipien “guter” Architektur: Dependency Injection (2)

**Lösung:**

- DependencyInjector (Assembler) als eigener Baustein




- Formen von Dependency Injection:
  - Constructor Injection
  - Interface Injection
  - Setter Injection

# Grundprinzipien “guter” Architektur: Dependency Injection (3)

**Naive Implementierung und Nutzung eines Loggers:**

```
public class ConsoleLogger {
    public void append(String message) {
        System.out.println(message);
    }
}
public class AnyClass {
    private ConsoleLogger logger = new ConsoleLogger();
    public void doSomething() {
        logger.append("I do something");
    }
}
public class Main {
    public static void main(String[] args) {
        AnyClass any = new AnyClass();
        any.doSomething();
    }
}
```

- Was ist schlecht an der Lösung?

Wie kann man die Lösung verbessern?

# Grundprinzipien “guter” Architektur: Dependency Injection (4)

```
public interface ILogger {
    void append(String message);
}
public class ConsoleLogger implements ILogger {
    public void append(String message) {
        System.out.println(message);
    }
}
public class AnyClass {
    private ILogger logger;
    public void setLogger(ILogger logger) {
        this.logger = logger;
    }
    public void doSomething() {
        logger.append("I do something");
    }
}
public class DependencyInjector {
    public static void main(String[] args) {
        ILogger logger = new ConsoleLogger();
        AnyClass any = new AnyClass();
        any.setLogger(logger);
        any.doSomething();
    }
}
```

Worin besteht die Verbesserung?

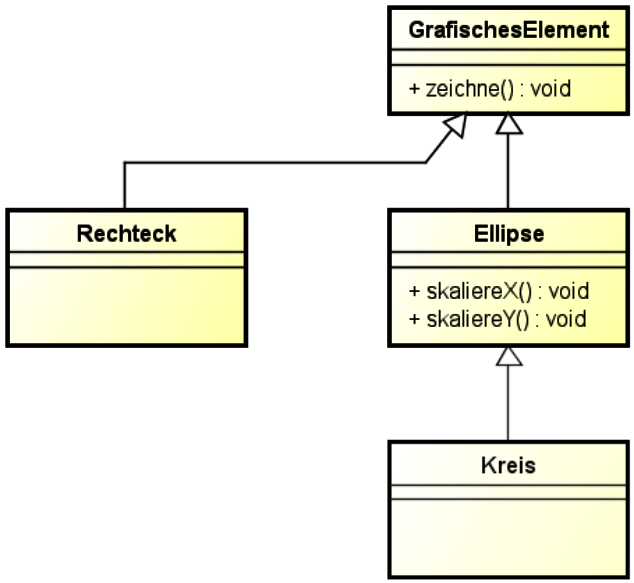
- Zunächst wird ein Interface `ILogger` erstellt, das die Schnittstelle für alle möglichen Logger festlegt
- `ConsoleLogger` implementiert Logger-Interface `ILogger`
- `AnyClass` soll Loggerfunktionalität nutzen (erzeugt Logger aber nicht)
- `AnyClass` stellt Methode zur Verfügung, um Logger zu setzen
- Ein `DependencyInjector` setzt den Logger von Klasse `AnyClass`

- Fazit:
  - Austausch des Loggers nur an einer zentralen Stelle (Dep.Injector)
- Frage: Injection-Typ?

# Grundprinzipien “guter” Architektur: Liskovsches Substitutionsprinzip (LSP) (1)

**Forderung:**

- Programm das Objekte einer Basisklasse T (hier: Ellipse) verwendet, muss auch mit Objekten einer abgeleiteten Klasse S (hier: Kreis) korrekt funktionieren, ohne dabei das Programm zu verändern
- Beispiel: Kreis-Ellipse-Problem (Verletzung des LSP)




- Verletzung des LSP, da Kreis die Methoden `skaliereX(…)` und `skaliereY(…)` nicht haben darf (er besitzt nur einen Radius)
- Damit darf hier `Kreis` nach dem LSP keine Unterklasse von `Ellipse` sein
- Beachte: die Entscheidung ist jeweils abhängig vom konkreten Fall:
  - Würde Ellipse hier die beiden Memberfunktionen nicht aufweisen, dürfte Kreis Unterklasse von Ellipse sein