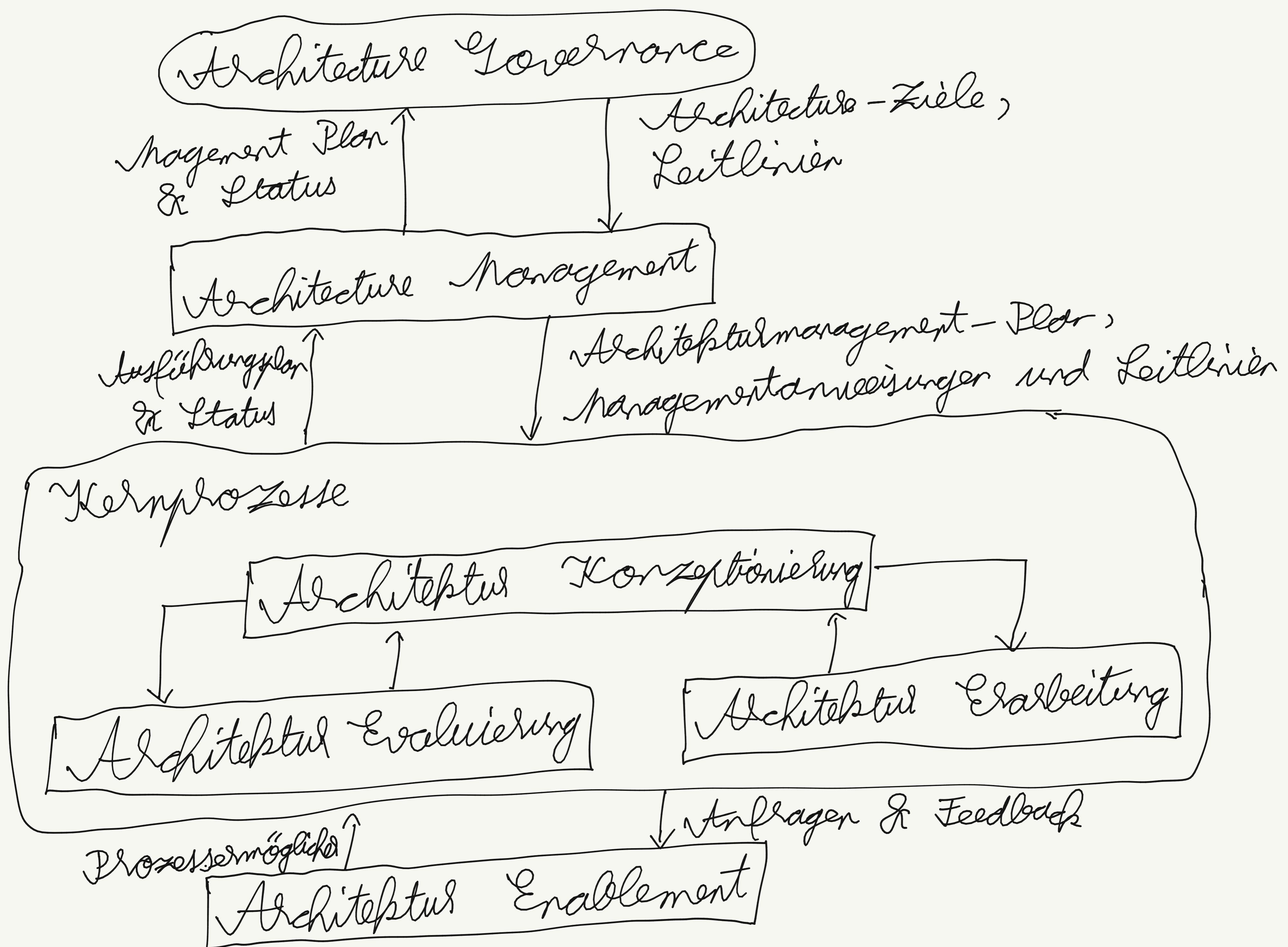


# Vorlesung 3

## Entwurf von Softwarearchitekturen

Architekturprozesse und deren Interaktionen  
nach ISO/IEC/IEEE 42020; 2019



## Architectural Governance:

- Festlegung von Richtlinien und Standards für die Architekturentwicklung
- Einrichten von Governance-Strukturen wie z.B. Architekturboards
- Überwachung der Einhaltung von Architekturstandards und -richtlinien

## Architectural Management:

- Planung und Priorisierung von Architekturaktivitäten und -ressourcen
- Risiko- und Forderungsmanagement für die Architektur
- Kommunikation und Koordination mit Stakeholdern

## Architectural Enablement:

- Schulung und Weiterbildung von Entwicklungsteams und Stakeholdern
- Bereitstellung von Tools und Ressourcen zur Unterstützung der Architekturentwicklung
- Aufbau und Pflege eines Wissenmanagements für Architekturwissen
- Förderung einer Kultur der kontinuierlichen Verbesserung und Innovation

- ## Konzeptionierung
- Sammlung und Analyse von Anforderungen und Randbedingungen
  - Dokumentation der Architekturkonzepte und -entscheidungen
  - Erstellung von Prototypen und Modellen zur Veranschaulichung der Architektur

## Entwicklung

- Detaillierung der Architekturkonzepte
- Unterstützung des Entwicklungsteams bei der Umsetzung der Architektur
- Erstellung von Implementierungsrichtlinien und -vorgaben

## Evaluierung

- Durchführung von Architekturenvestigungen und -analysen
- Einkholung von Feedback von Stakeholdern und Experten
- Identifikation und Bewertung von Risiken und Schwachstellen

# Zusammenspiel zwischen Architekturentwurf und Requirements Engineering

Softwarearchitektur  $\rightleftarrows$  Anforderungsspezifikation

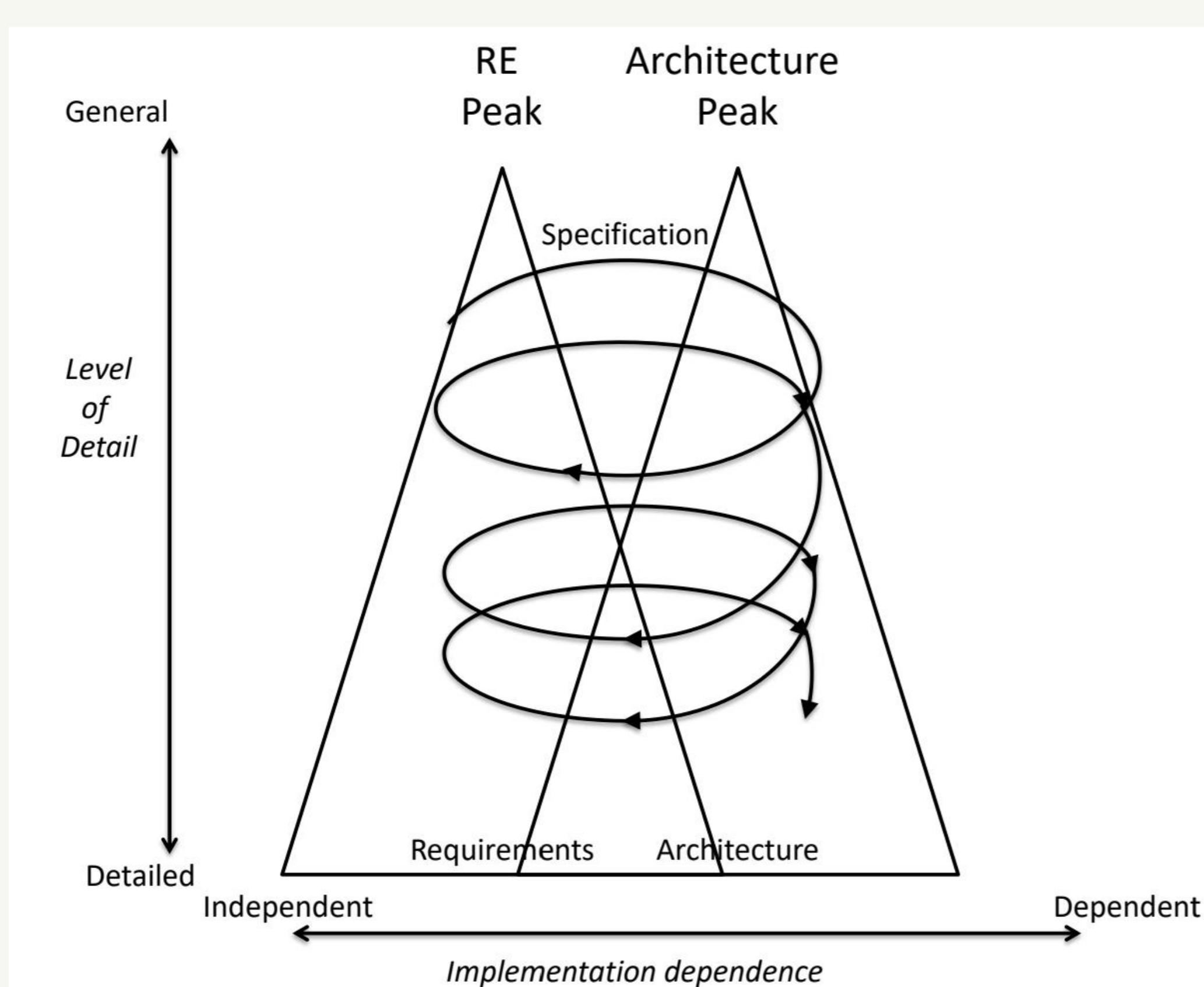
Einfluss von Softwarearchitektur auf die Anforderungsspezifikation:

- Verfeinerung
  - Ergänzung
  - Validierung
  - Priorisierung
  - Dokumentation
- }  $\rightarrow$  der Anforderung

Einfluss von Anforderungen auf die Softwarearchitektur:

$\rightarrow$  Funktional und nicht-funktional  
Anforderungen (geforderte Qualitätseigenschaften)

## Two Peaks Model



Grundgedanke:

RE & Architekturentwurf dürfen nicht isoliert voneinander betrachtet werden

Wesentliche Konzepte:

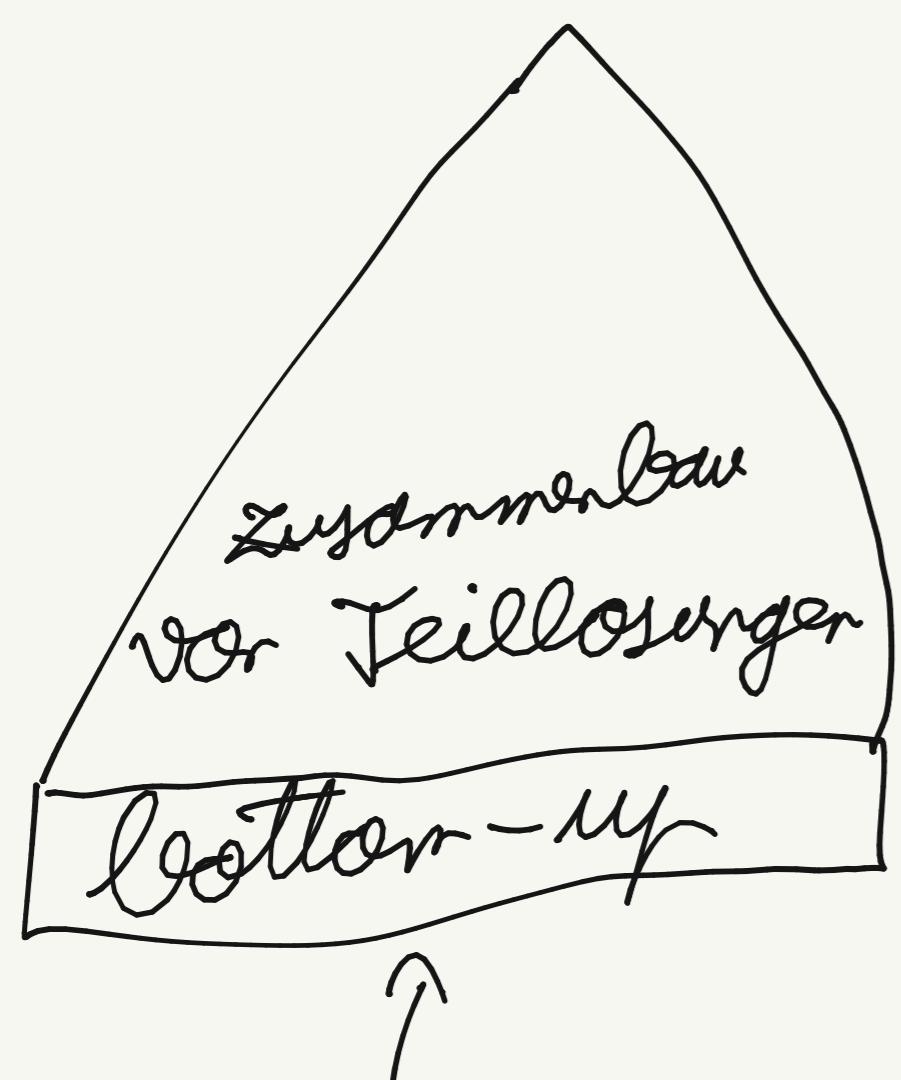
- Parallel Entwicklung
- Iterativer Ansatz
- Eng verzahnte Aktivitäten
- Berücksichtigung von Risiken und Unsicherheiten

RE Peak  
· repräsentiert Aktivitäten zur Erhebung und Analyse von Anforderungen

Architecture Peak  
· repräsentiert Aktivitäten zur Entwicklung und Verfeinerung der Architektur

# Top-Down & Bottom-Up Architekturentwurf

## Bottom-Up Architekturentwurf

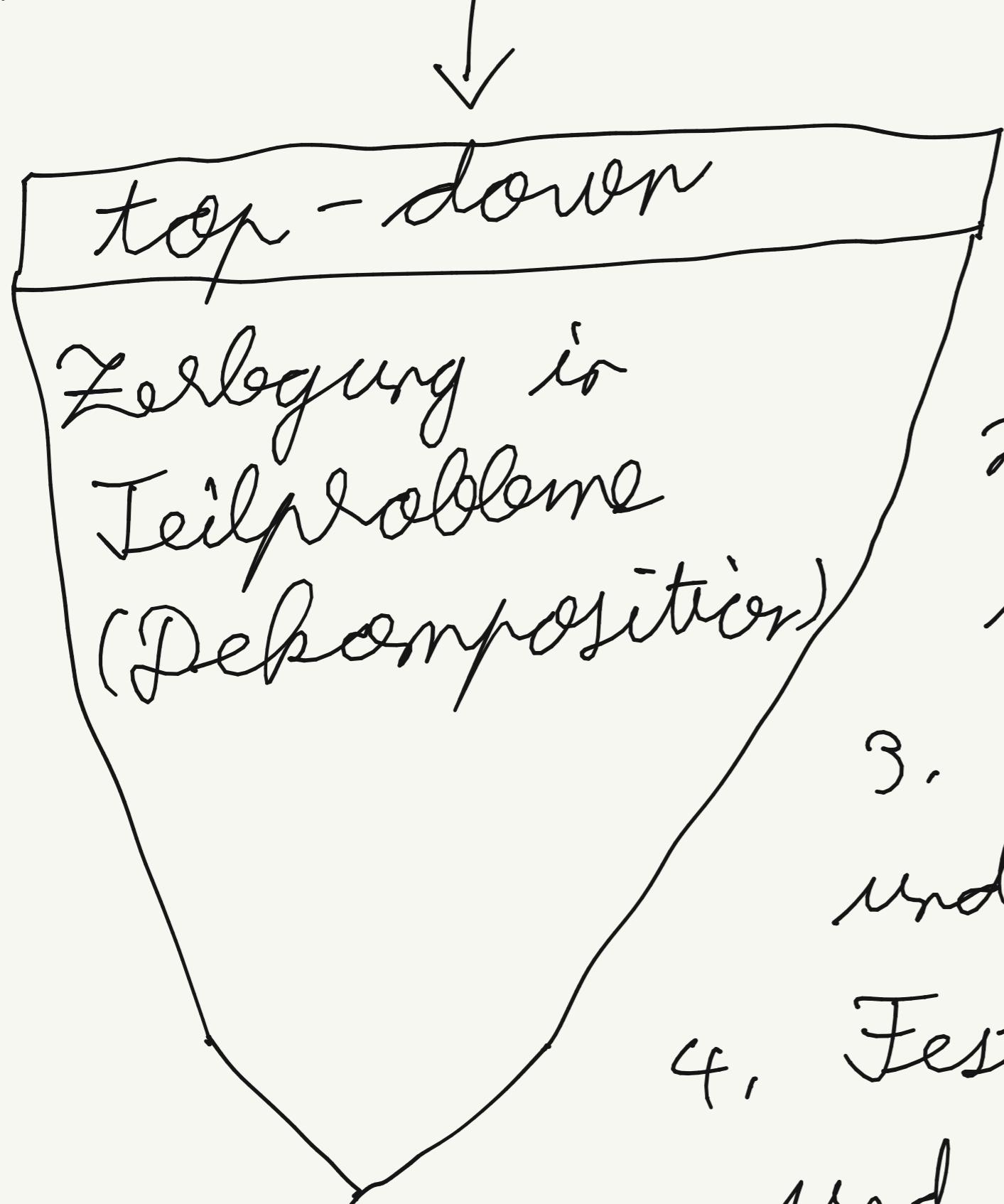


5. Optimierung und Verbesserung
4. Test und Validierung
3. Integration von Komponenten
2. Entwicklung von Komponenten
1. Identifikation von Bausteinen

Bibliotheken und Teilfunktionen

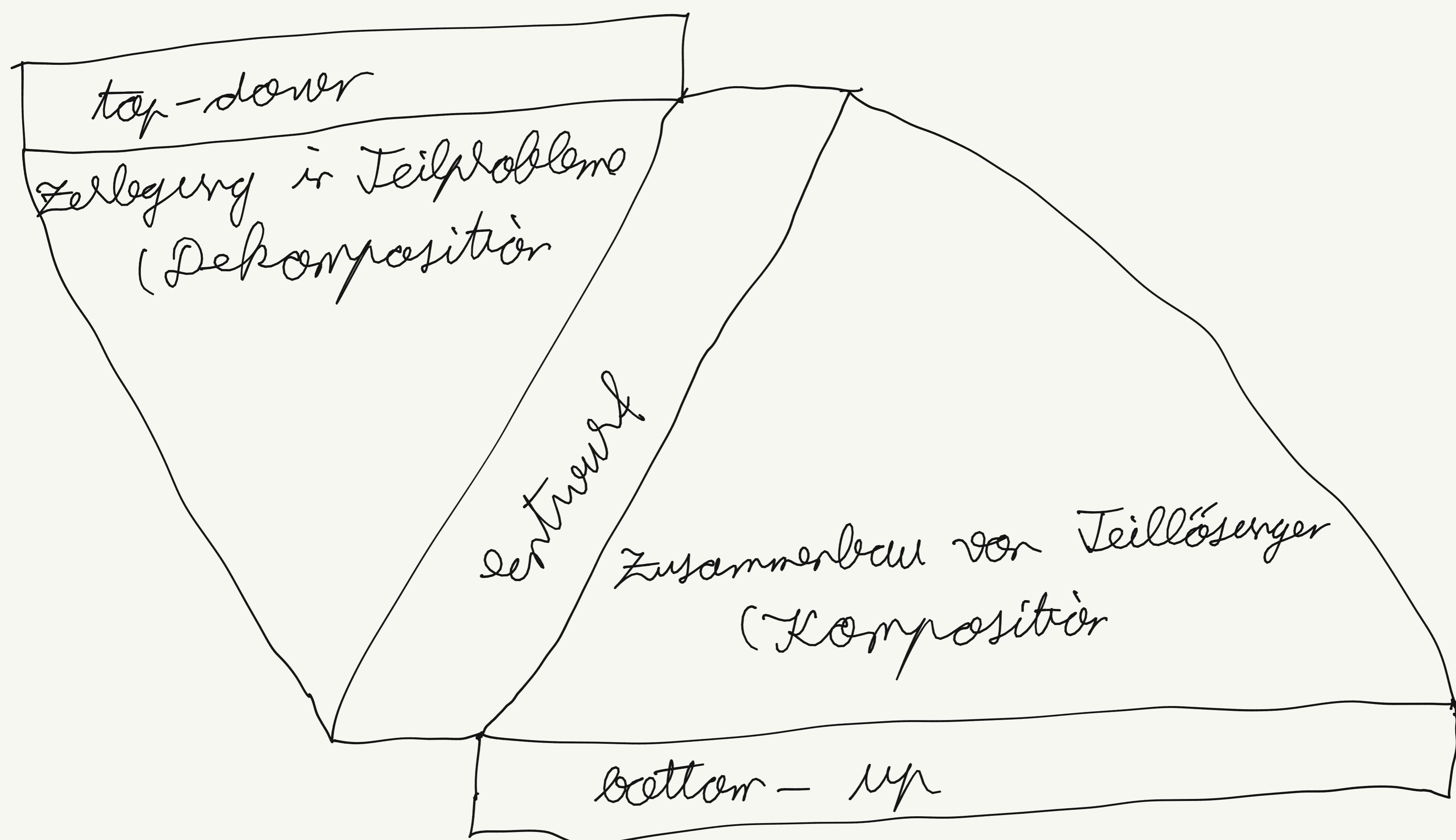
## Top-Down Architekturentwurf

Vision vom Gesamtsystem



1. Identifikation der Anforderungen
2. Entwicklung der übergeordneten Architektur
3. Aufteilung in Subsysteme und Komponenten
4. Festlegung von Schnittstellen und Interaktionen
5. Detaillierte Spezifikation und Implementierung

# Middle-Out / Hybrid-Ansatz



- Top-Down für die Grobarchitektur,  
Bottom-Up für Details
- Top-Down für langfristige Vision,  
Bottom-Up für Prototypen und Iterationen
- Iteratives Vorgehen mit Feedbackschleife

## Design by Contract

- enthält eine Reihe von Prinzipien zur komponentenbasierter Softwareentwicklung
- Analogie mit Verträge im Geschäftsleben
  - Die Komponenten schließen untereinander Verträge ab
  - Ein Vertrag beinhaltet das Versprechen einer Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind
  - Letzte Programm so aus Komponenten zusammen, dass es richtig funktioniert, wenn alle nur ihre Verträge einhalten

## Vertrag für eine Methode

Ein Vertrag für eine Methode kann aus folgender Elementer bestehen:

### • Vorbedingung:

- Bedingungen, die erfüllt sein müssen, bevor eine Methode aufgerufen werden kann
- Beispiel: Methode zur Berechnung einer Wurzel darf nur aufgerufen werden, wenn der Parameter eine nicht-negative Zahl ist

### • Nachbedingung:

- Welche Leistung erbringt die Methode, d.h. welche Eigenschaft gilt nach ihrer Ausführung
- Beispiel: Methode stellt sich, dass das Quadrat des Ergebnisses gleich der Eingabe ist

### • Klasseninvariante:

- Zu jedem Zeitpunkt erfüllt der Zustand der Klasse eine bestimmte Eigenschaft (die Invariante)
- Beispiel: Eine Klasse, die eine Uhrzeit repräsentiert, dass die Stunden jederzeit zwischen 0-23 liegen, und die Minuten jederzeit bei 0-59.

## Vorteile:

- Klare Spezifikation
- Frühe Fehlererkennung
- Verbesserte Wartbarkeit
- Erleichtertes Testen

## Nachteile:

- Erhöhter Aufwand
- Laufzeitüberprüfungen  
→ Swift bietet hier zum Beispiel:
  - > assert(...)
  - > precondition(...)
- Komplexität

## Conway's Law

"Organisations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organisations"

- Beobachtung durch Conway:  
Häufig korrelieren die Module einer Software mit den Organisationseinheiten, welche die Software entwickeln
- Heißt: Die Organisationsstruktur nimmt Einfluss auf die Softwarearchitektur
- Dies kann auch die explizite Beeinflussung für eine bestimmte Architektur dienen:
  - Umverse Conway Monarchie
  - z.B.: Aufteilung der Teams nach Komponenten

# Kopplung und Kohäsion

## Kopplung

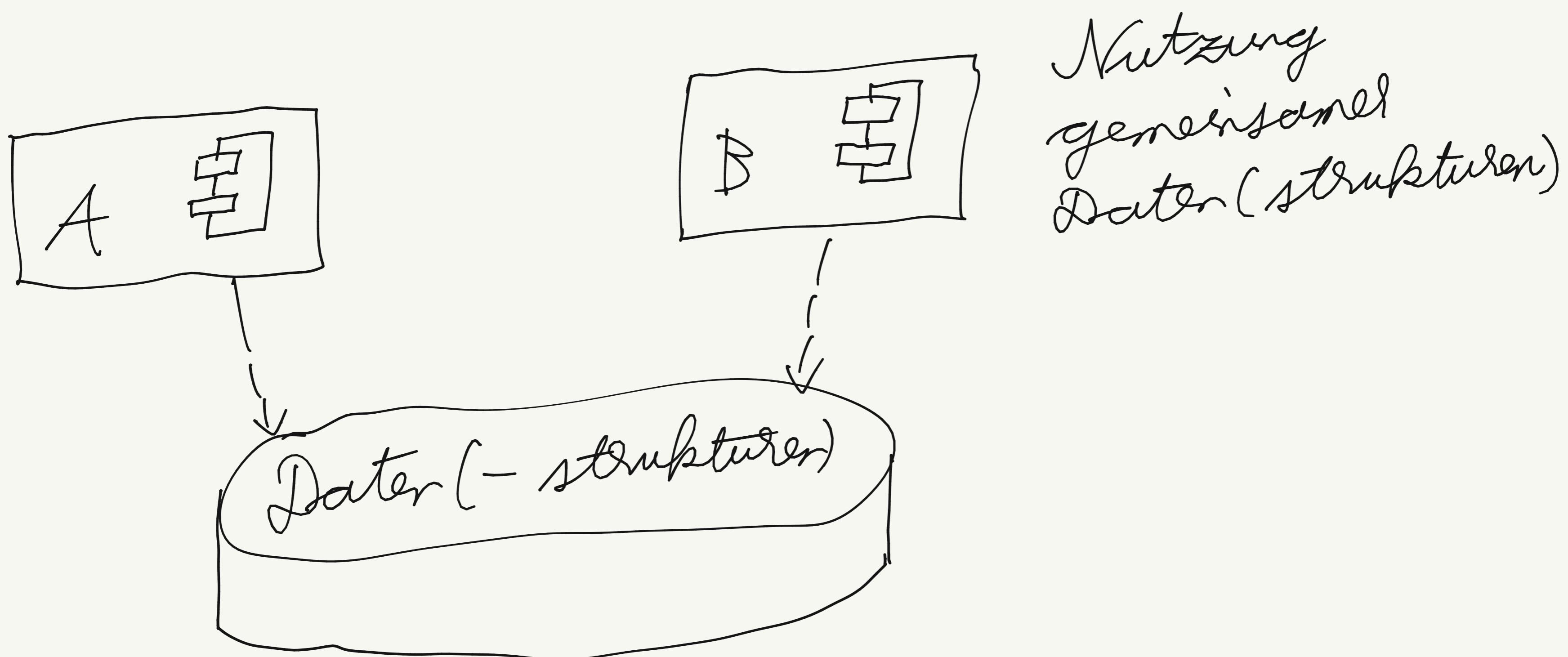
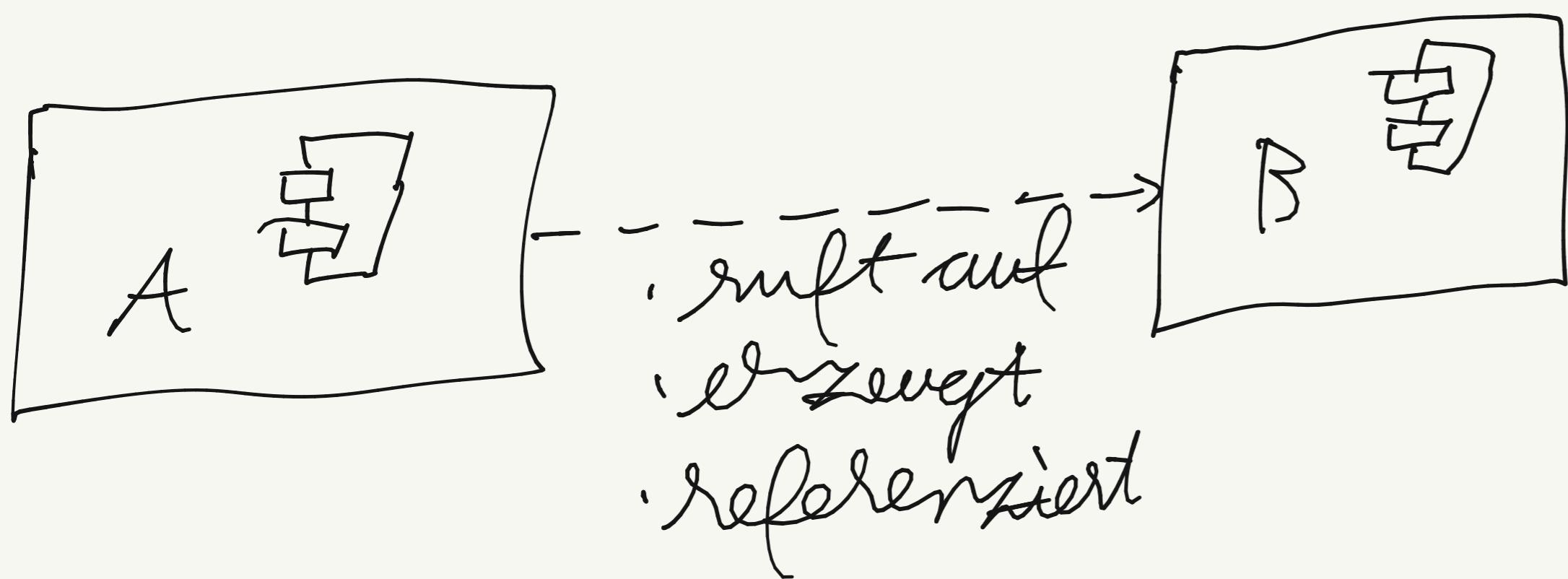
- Maß der Abhängigkeit zwischen verschiedenen Modulen oder Komponenten eines Softwaresystems
- Gibt Aufschluss darüber, wie stark Änderungen an Komponenten andere beeinflussen

Ziel ist eine geringe Kopplung zwischen Komponenten

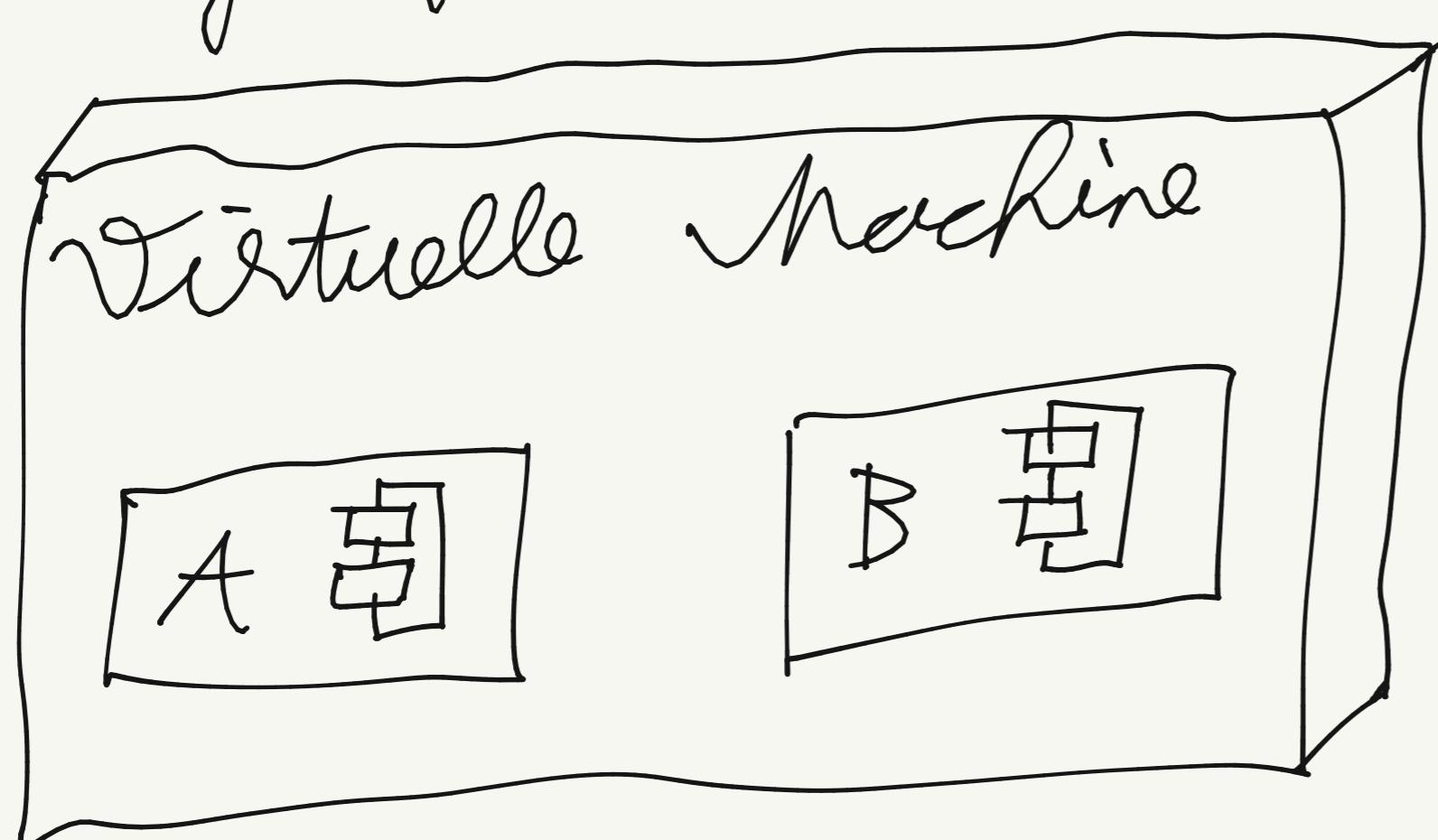
eine geringe Kopplung verbessert die:

- Wartbarkeit
- Flexibilität
- Testbarkeit

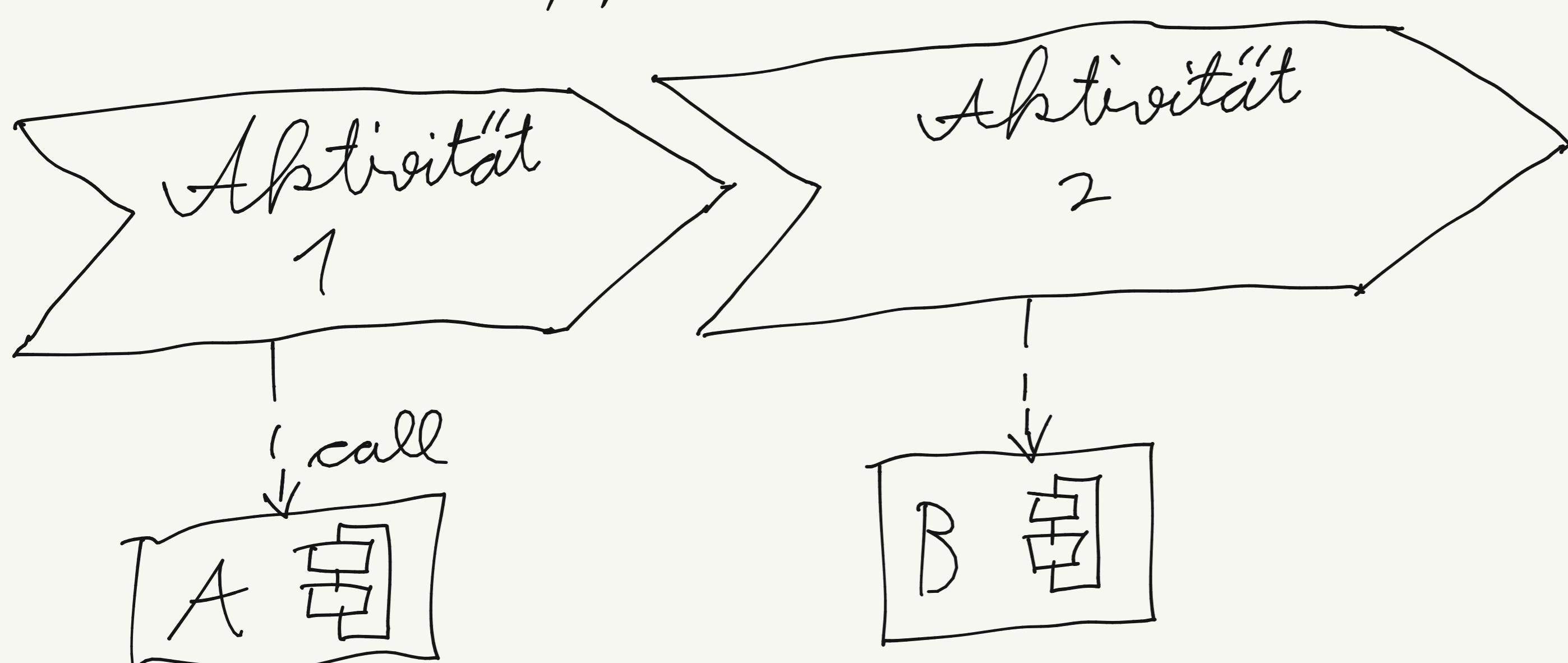
## Arten von Kopplung



Nutzung gemeinsamer Ressourcen



## Zeitliche Kopplung



## Efferente Kopplung / Efferent Coupling (Ce)

- Anzahl vor Komponenten, die eine bestimmte Komponente referenzieren
- Ein hoher Ce-Wert deutet darauf hin, dass die Komponente viele andere Komponenten nutzt  
⇒ Indiz für starke Kopplung

## Afferente Kopplung / Afferent Coupling (Ca)

- Anzahl der anderen Komponenten, die auf eine bestimmte Komponente direkt zugreifen
- Ca zählt, wie viele andere Komponenten direkt von der betrachteten Komponente abhängen
- Ein höherer Ca deutet darauf hin, dass die betrachtete Komponente von vielen anderen Komponenten genutzt wird:
  - Indiz für gute Wiederverwendbarkeit
  - Ggf. Single-point-of-Failure
  - Ggf. hohe Kopplung

## Instability (I) / Instabilitätswert

- Verhältnis von efferenter Kopplung zu der Summe aus efferenter und afferenter Kopplung

$$I = \frac{Ce}{Ce + Ca}$$

- Instabilitätswert nahe 1 ⇒ Komponente ist sehr instabil, da stark von anderen abhängig
- Instabilitätswert nahe 0 ⇒ Komponente ist stabil, da viele Komponenten von ihr abhängig sind

## Kohäsion

- Bezieht sich auf das Maß, in dem die Elemente oder Funktionen innerhalb einer Komponente zusammengehören oder gemeinsam ein Blaues und zusammenhängendes Ziel verfolgen
- Hohe Kohäsion  
=> Die in einer Komponente enthaltenen Elemente oder Funktionen erfüllen gemeinsam eine spezifische Aufgabe
- Niedrige Kohäsion  
=> Die in einer Komponente enthaltenen Elemente oder Funktionen sind weniger zusammengehörig und erfüllen möglicherweise unterschiedliche Aufgaben oder haben unterschiedliche Verantwortlichkeiten

Ziel ist eine hohe Kohäsion innerhalb einer Komponente.  
Eine hohe Kohäsion verbessert die:

- Wartbarkeit
- Flexibilität
- Testbarkeit

## Messung von Kohäsion

### LCOM - Lack of Cohesion Metric

Die LCOM-Metrik versucht, den Grad des Zusammenhalts innerhalb einer Klasse zu quantifizieren.

$$LCOM = \frac{\left( \frac{1}{a} \sum_{j=1}^a n(A_j) \right) - m}{1 - m}$$

Je höher der LCOM-Wert, desto geringer die Kohäsion:

- A: Attribut der Klasse
- a: Anzahl Attribute in einer Klasse
- m: Anzahl der Methoden in einer Klasse
- $n(A_j)$ : Anzahl der Methoden einer Klasse, die Attribut  $A_j$  nutzen

# Domain-Driven Design (DDD)

Verstehen sich alle Beteiligten wirklich?  
The information model is a fundamental tool for  
managers in the  
digital change  
Auch Domänenmodell genannt

## Grundlagen

- Domain-Driven Design (DDD) ist eine Vorgehensweise an die Modellierung komplexer Software
- „Der entscheidende Punkt, um den es mir geht, ist auf der Grund, warum das Konzept „Domain - driven Design“ heißt:
  - Wenn wir Software entwickeln, sollte unser Fokus nicht primär auf der Technologie liegen, die wir verwenden
  - Stattdessen sollte sich unser Hauptgermany auf die geschäftlich Leute richten, also auf der fachlichen Bereich, der wir mit unserer Software unterstützt werden - kurz auf die Domäne
- Das ist es, was ich mit Domain - driven Design meine“

## Ubiquitous Language

- DDD betont die Verwendung einer allgegenwärtiger Sprache, die vor allen Teammitgliedern - Entwicklern, Domänenexperten und anderen Stakeholdern - verwendet wird
- Diese gemeinsame Sprache wird im Code, in der Dokumentation und in den Gesprächen verwendet
- Die Verwendung einer ubiquitären Sprache reduziert Missverständnisse und Kommunikationsprobleme zwischen verschiedenen Teilen des Systems und zwischen Entwicklern und Domänenexperten
- Darstellung unter anderem mittels:
  - Glossar
  - UML-Klassendiagramm

## Weitere Konzepte von DDD

- Kontextbegrenzer (Bounded Context)
  - zentraler Begriff im Domain-Driven Design
  - Domäne wird in klar abgegrenzte Bereiche unterteilt
  - Jeder Bounded Context definiert eine Grenze, innerhalb derer eine spezifische Modellierung der Domäne gilt und die Regeln, Begriffe und Konzepte konsistent sind
- Kontextübersicht (Context Map)
  - visualisiert die Beziehungen und Interaktionen zwischen verschiedenen Bounded Contexts innerhalb einer Domäne
  - hilft ein besseres Verständnis der Gesamtdomäne zu entwickeln, indem sie zeigt, wie verschiedene Teile eines Systems miteinander verbunden sind und wie sie miteinander kommunizieren
- Antikorruptionsschicht (Anticorruption Layer)
  - Ein Anti-Corruption Layer (ACL) ist eine Architekturkonzept im Domain-Driven Design (DDD)
  - Schafft eine schützende Schicht zwischen zwei unterschiedlichen Bounded Contexts oder Systemen
  - Diese Schicht stellt sicher, dass der eigene Kontext vor den Einflüssen und der möglich unpassender Modelle oder Konzepten eines anderen Kontexts geschützt wird

## Event Storming

Workshop-basierte Methode um schnell herauszufinden was in einer Domain "passiert"   
 >> The business process is "stormed out" as a series of domain events which are denoted as stickies.   
 Methode, um Fachexperten und Softwareentwickler zusammenzubringen

### Ablauf:

1. Vorbereitung:
  - Stakeholder einladen und Moderator bestimmen
  - groÙe Arbeitsfläche / Wand bereitlegen,
  - groÙe Haftnotizzettel und Marker
  - ebenso viele bunte Haftnotizzettel
2. Identifikation von Domänenereignissen
  - Teilnehmer schreibt wichtige Ereignisse auf
  - längere Haftnotizzettel
  - Ein Ereignis ist etwas, das passiert und für das Geschäft von Bedeutung ist
  - Ereignisse in chronologischer Reihenfolge an die Wand kleben
3. Erweitern und Vertiefen
  - Kommandos definieren (blaue Haftnotizzettel), die Ereignisse auflösen
  - Ein Kommando ist eine Benutzaktion oder eine Entscheidung im System

## Weitere Designprinzipien

Postel's Law:

"Be conservative in what you do, be liberal in what you accept"

- Sei stetig bei dem, was du tust, und offen bei dem, was du vor anderen akzeptierst
- Aus Sicht eines Schnittsteller-Konsumenten:
  - Rechne mit Service-Ausfällen,
  - mit fehlerhaftem Verhalten, ...
- Aus Sicht eines Schnittsteller-Implementierers:
  - Halte dich genau an die Schnittstellen-Spezifikation
  - Biete nicht nur die aktuelle API an, sondern auch mindestens eine vergangene Version. Das gibt der Verwender Gelegenheit, auf die neue Version umzusteigen

## Interface Segregation Principle

- Clients sollte nicht dazu gezwungen werden von Schnittstelle (und deren Methoden) abzuhängen, die sie gar nicht brauchen
- Damit würden sich Änderungen an nicht benötigter Schnittstelle (und deren Methoden) unnötigerweise auf Verwendung auswirken.
- Schnittstelle sollte fachlich strukturiert sein.  
Deshalb:
  - diese nicht ausschließlich an den Bedürfnissen aller Clients anpassen
  - diese nicht ausschließlich an den Bedürfnissen aller Implementierer ausrichten
- Im Vordergrund sollte semantischer und fachlicher Zusammenhang der Methoden stehen
- Kleine und fokussierte Schnittstellen sind leichter implementierbar und wartbar

## Offen - Geschlossen - Prinzip

- Ein Modul soll offen für Erweiterung und geschlossen für Änderungen sein

## So - einfach - wie - möglich - Prinzip

- Albert Einstein: "Mache Dinge so einfach wie möglich - aber nicht einfacher"

## Information Hiding (Geheimnisprinzip)

- Kapselung von Komplexität in Komponenten
- Verwenden von Schnittstellen
- Ließe Kerneigenschaften einer Komponente
- Siehe Kernbereich

## Regelmäßiges Refactoring und Redesign

## Nutzung von Architekturmustern

- siehe nächster Themenbereich