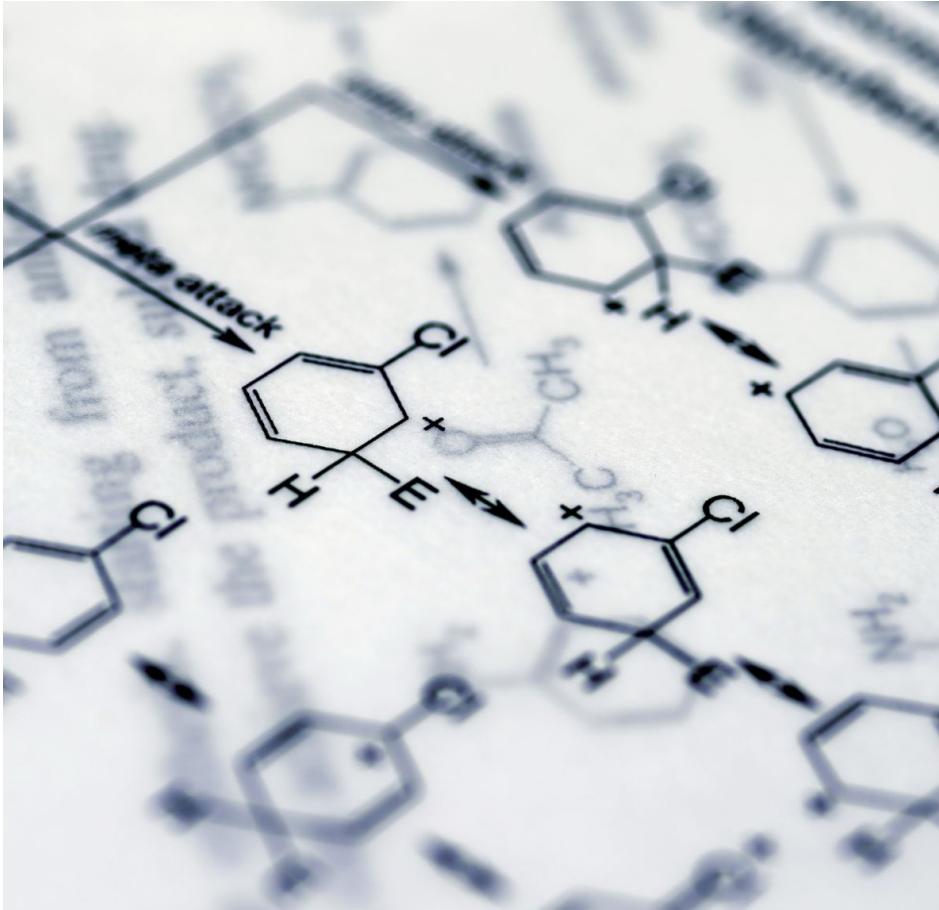


Themenbereiche der Vorlesung



Themenbereiche

A – Grundlegende Konzepte

B – Dokumentation und Kommunikation

C – Entwurf von Softwarearchitekturen

D – Architekturmuster

E – Qualität von Softwarearchitekturen

Entwurf von Softwarearchitekturen

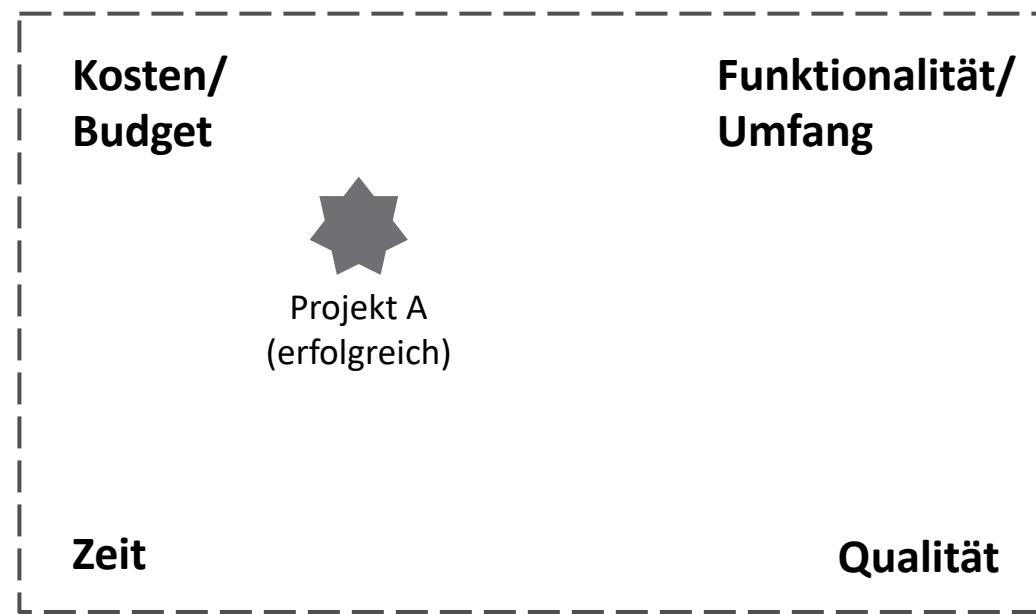
Themenbereich C

Inhalte

- Architekturprozesse und deren Interaktionen nach ISO/IEC/IEEE 42020:2019
- Zusammenspiel zwischen Architekturentwurf und Requirements Engineering
- Top-Down & Bottom-Up Architekturentwurf
- Design by Contract
- Conway's Law
- Kopplung & Kohäsion
- Domain-Driven Design (DDD)
- Weitere Designprinzipien

Ziele und Aufgaben des Architekturentwurfs

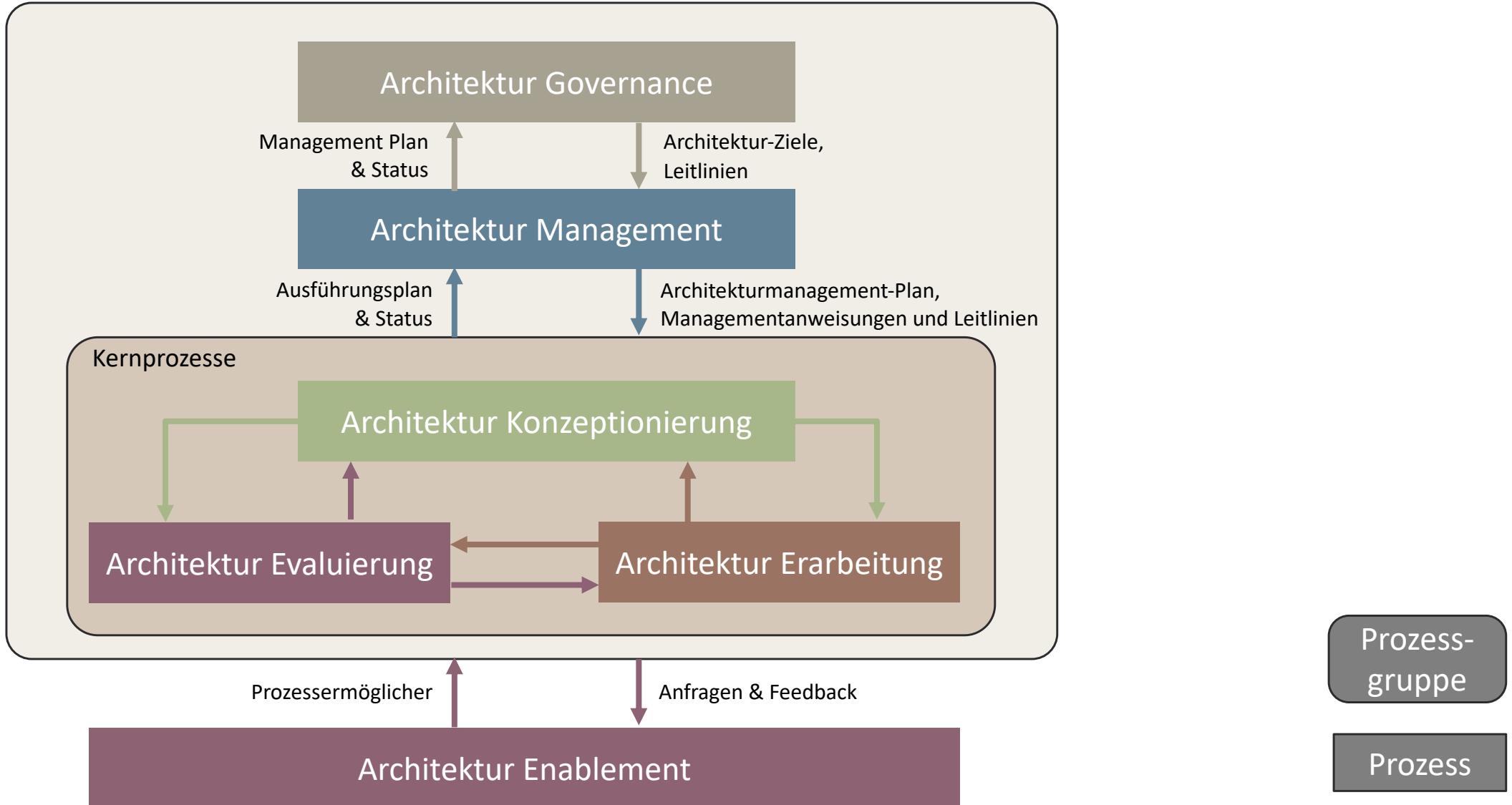
- Finden eines Konstruktionswegs, mit dem die funktionalen und nicht-funktionalen Anforderungen aus dem Requirements Engineering umgesetzt werden können
- Unterstützung der Projektziele
 - siehe magisches Viereck erfolgreicher Softwareprojekte



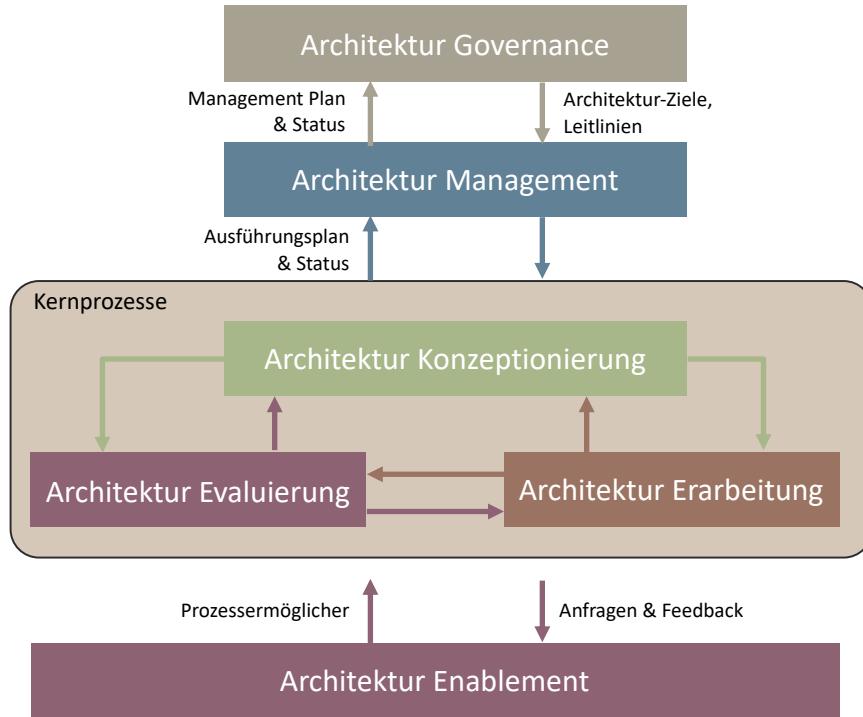
Architekturprozesse und deren Interaktionen nach ISO/IEC/IEEE 42020:2019

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

Angelehnt an Abbildung 1 aus: ISO/IEC/IEEE 42020:2019(E):
 ISO/IEC/IEEE International Standard - Software, Systems and
 Enterprise -- Architecture Processes.



Unterstützungsprozesse



- **Architektur Governance**

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

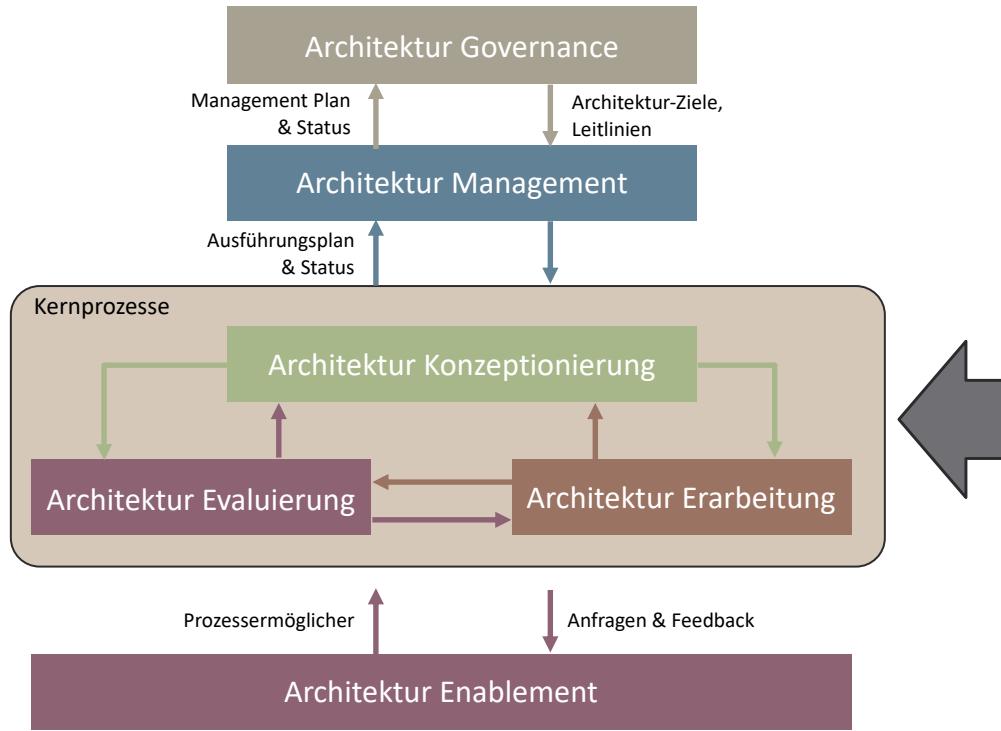
- **Architektur Management**

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

- **Architektur Enablement**

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

Kernprozesse



- **Konzeptionierung**

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

- **Erarbeitung**

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

- **Evaluierung**

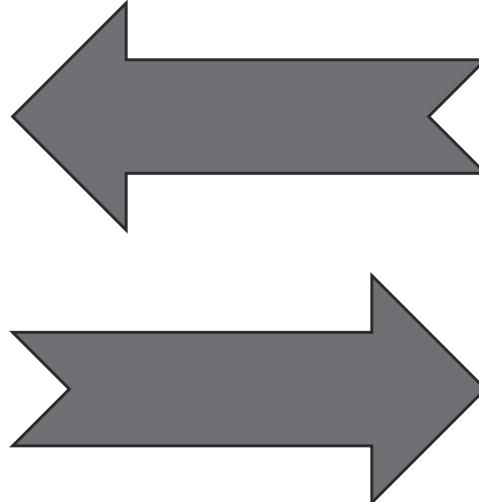
- Durchführung von Architekturbewertungen und –analysen
- Einholung von Feedback von Stakeholdern und Experten
- Identifikation und Bewertung von Risiken und Schwachstellen

Zusammenspiel zwischen Architekturentwurf und Requirements Engineering

Zusammenspiel Architekturentwurf und Anforderungsmanagement



Softwarearchitektur



Anforderungs-
spezifikation

Einfluss von Softwarearchitektur auf die Anforderungsspezifikation



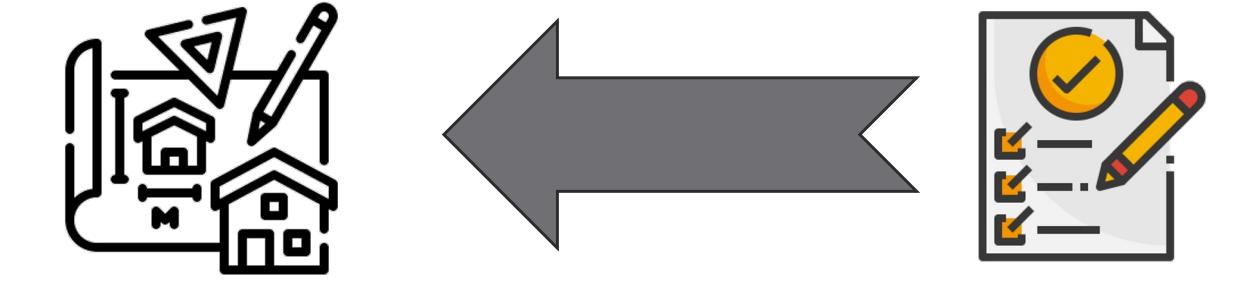
Softwarearchitektur



Anforderungs-
spezifikation

- Verfeinerung...
 - Ergänzung...
 - Validierung...
 - Priorisierung...
 - Dokumentation...
- ... der Anforderungen

Einfluss von Anforderungen auf die Softwarearchitektur

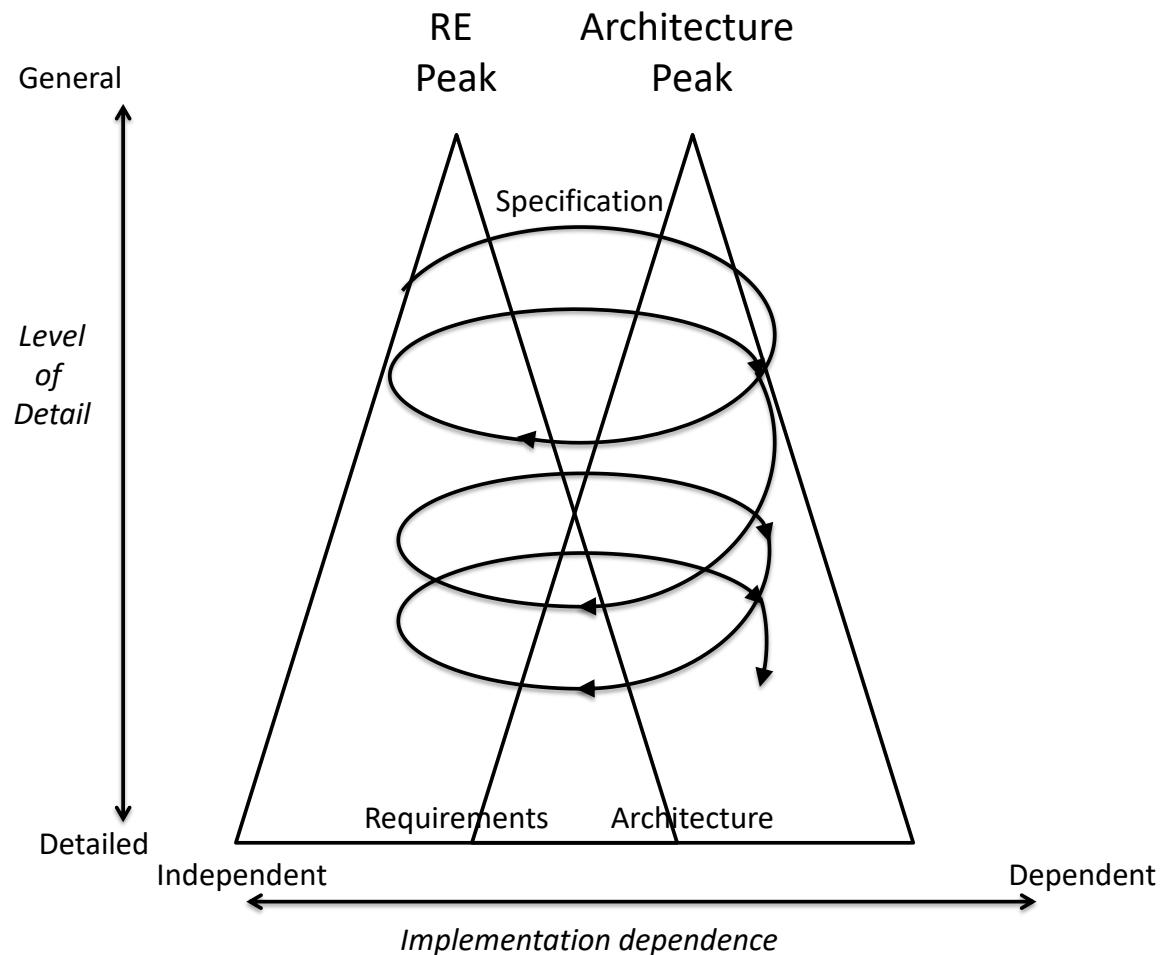


Softwarearchitektur

**Anforderungs-
spezifikation**

Funktionale & nicht-funktionale
Anforderungen (geforderte
Qualitätseigenschaften)

Twin Peaks Model



Quelle: "Weaving the Software Development Process Between Requirements and Architecture", Proceedings of ICSE-2001 International Workshop: From Software Requirements to Architectures (STRAW-01) Toronto, Canada

Grundgedanke:

RE & Architekturentwurf dürfen nicht isoliert voneinander betrachtet werden.

Wesentliche Konzepte

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

RE Peak

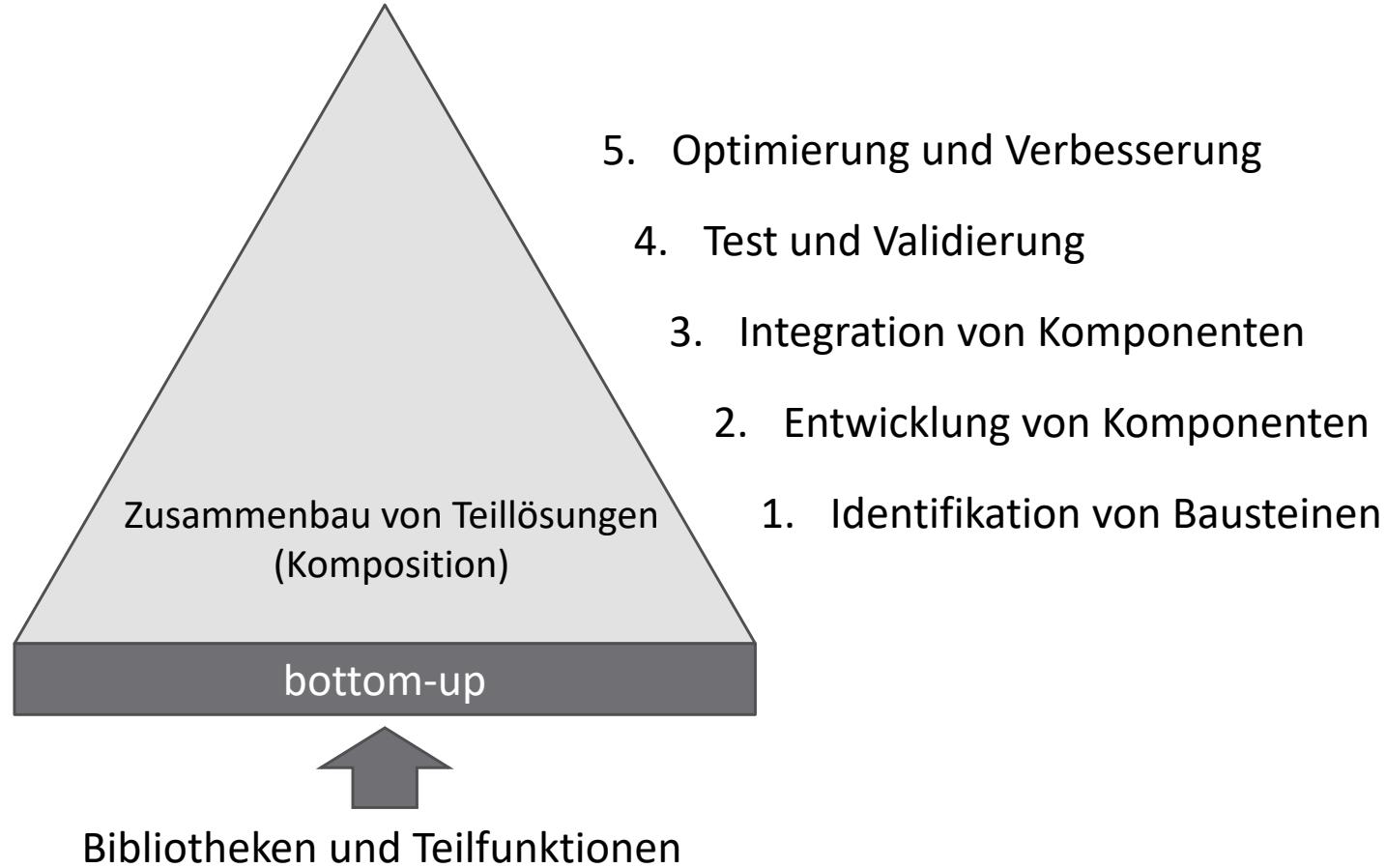
- Repräsentiert Aktivitäten zur Erhebung und Analyse von Anforderungen

Architecture Peak

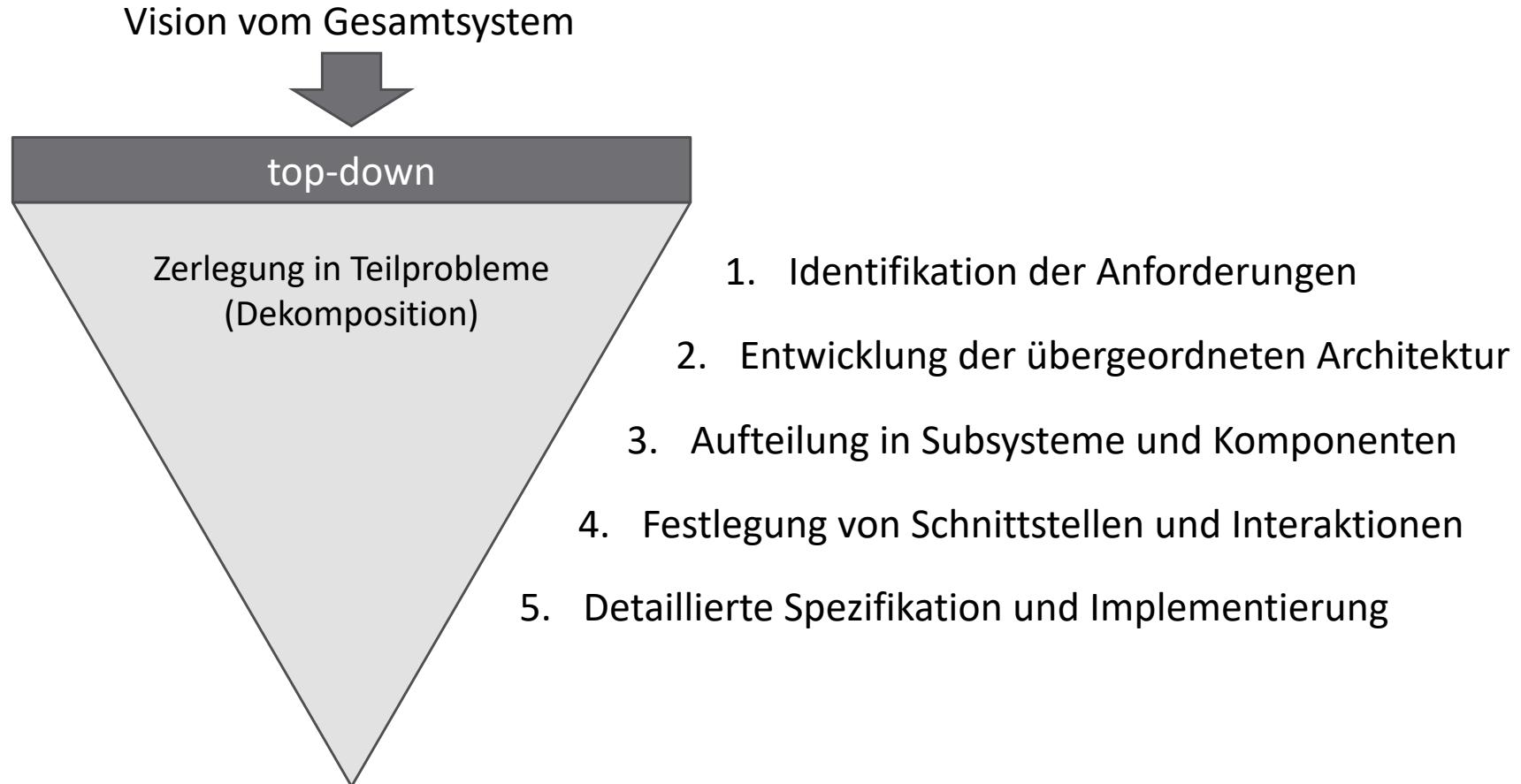
- Repräsentiert Aktivitäten zur Entwicklung und Verfeinerung der Architektur

Top-Down & Bottom-Up Architekturentwurf

Bottom-Up Architekturentwurf



Top-Down Architekturentwurf



Middle-Out / Hybrid-Ansatz

Vision vom Gesamtsystem



top-down

Zerlegung in Teilprobleme
(Dekomposition)

Entwurf

Zusammenbau von Teillösungen
(Komposition)

bottom-up



Bibliotheken und Teilfunktionen

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

Design by Contract

Design by Contract

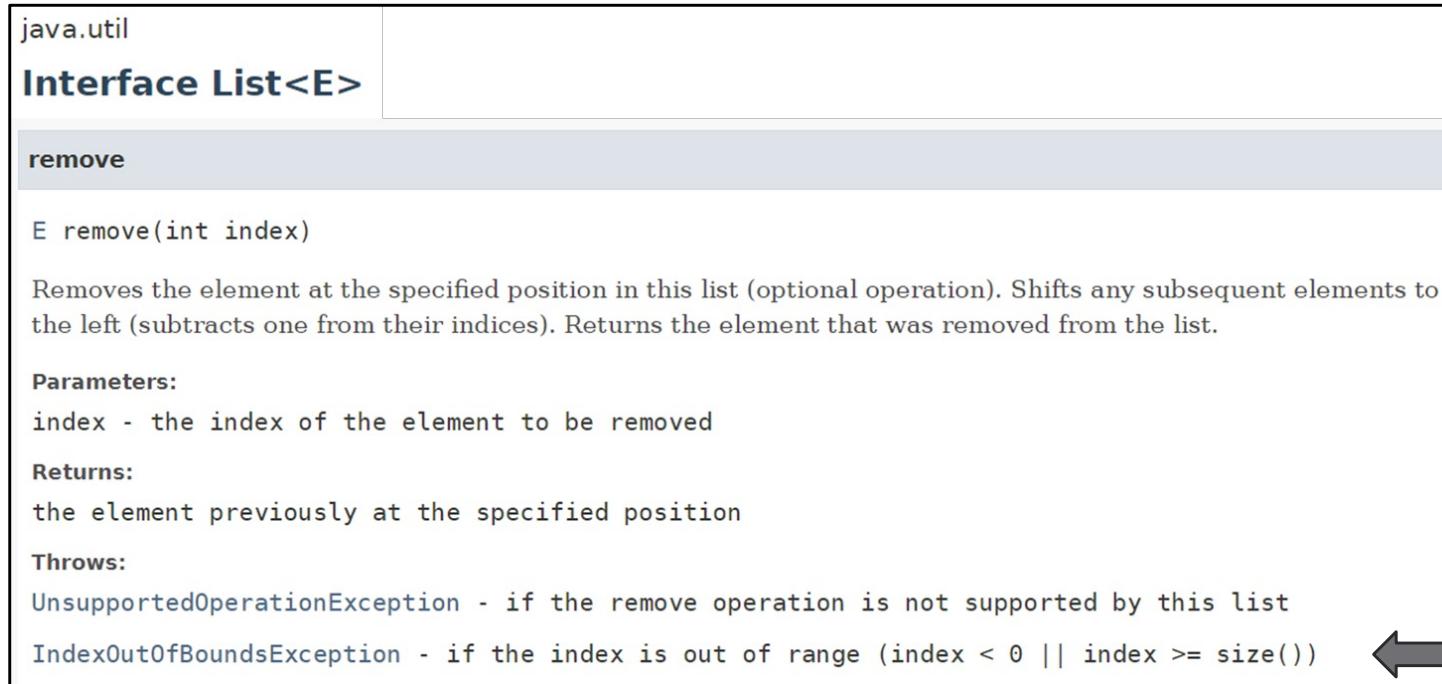
- Design by Contract enthält eine Reihe von Prinzipien zur komponentenbasierten Softwareentwicklung
- Analogie mit Verträgen im Geschäftsleben
 - Die Komponenten schließen untereinander Verträge ab
 - Ein Vertrag beinhaltet das Versprechen einer Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind.
 - Setze 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 zum Beispiel aus folgenden Elementen 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 sicher, 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.

Beispiel – Interface List

Beispiel:



The screenshot shows the JavaDoc for the `remove` method of the `List` interface. The method signature is `E remove(int index)`. The description states: "Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list." Below the description, there are sections for **Parameters**, **Returns**, and **Throws**. The **Parameters** section contains `index - the index of the element to be removed`. The **Returns** section contains `the element previously at the specified position`. The **Throws** section contains `UnsupportedOperationException - if the remove operation is not supported by this list` and `IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())`.

Nachbedingung

Vorbedingung

Quelle: <https://docs.oracle.com/javase/8/docs/api/java/util/List.html#remove-int->

Invarianten

- Die Liste darf nie eine Lücke enthalten.
- Die Länge der Liste (`size()`) muss immer die Anzahl der enthaltenen Elemente widerspiegeln.

Beispiel – Interface List



implementiert

Kennt nur die Schnittstelle,
nicht den Anwendungskontext
oder den Verwender.

Vertrag / Schnittstelle

java.util
Interface List<E>

remove

```
E remove(int index)
```

Removes the element at the specified position in this list (optional operation). Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the list.

Parameters:
index - the index of the element to be removed

Returns:
the element previously at the specified position

Throws:
UnsupportedOperationException - if the remove operation is not supported by this list
IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

verwendet



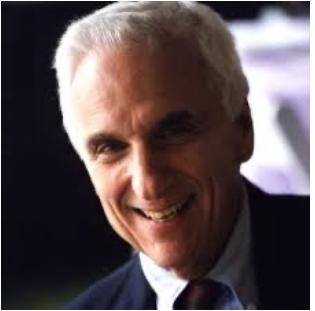
Kennt nur die Schnittstelle,
nicht die Implementierung und
nicht den Implementierer.

Vor- und Nachteile

- **Vorteile**
 - Klare Spezifikation
 - Frühe Fehlererkennung
 - Verbesserte Wartbarkeit
 - Erleichtertes Testen
- **Nachteile**
 - Erhöhter Aufwand
 - Laufzeitüberprüfungen
 - Die Programmiersprache Swift bietet hier zum Beispiel:
 - `assert(...)`: Überprüft eine Bedingung zur Laufzeit; Kann in einem Release-Build deaktiviert werden
 - `precondition(...)`: Überprüft eine Bedingung und beendet das Programm mit einer Fehlermeldung. Wird immer auch im Release-Build geprüft.
 - Komplexität

Conway's Law

Conway's Law



“Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.”

[Conway, 1967]

Melvin Edward Conway

- 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 der expliziten Beeinflussung hin zu einer bestimmten Architektur dienen:
 - Inverse Conway Maneuver
 - Z.B. Aufteilung der Teams nach Komponenten

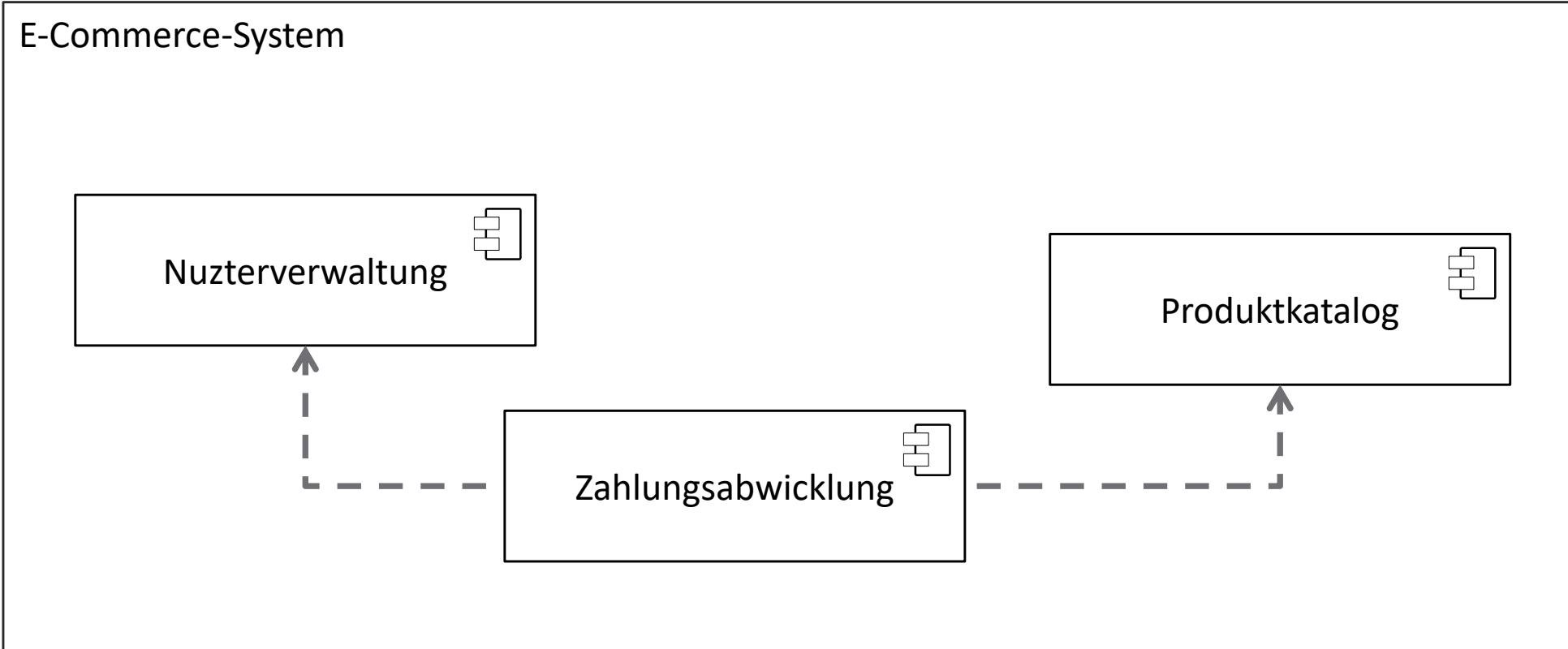
Beispiel – E-Commerce-System



Mögliche (positive) Auswirkungen auf die Architektur:

- Modulare Architektur
- Kommunikation zwischen Komponenten über dokumentierte Schnittstellen

Beispiel – E-Commerce-System



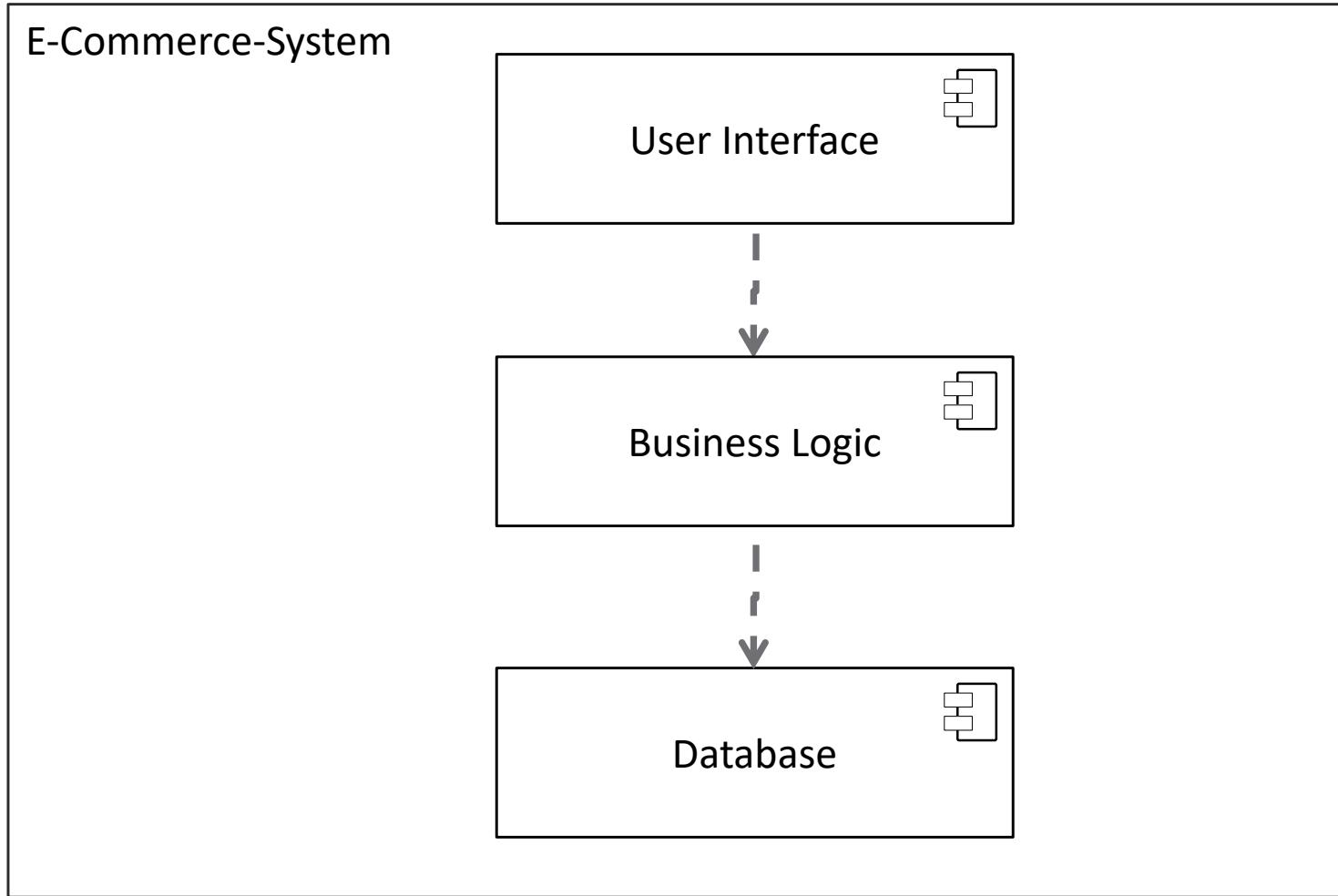
Beispiel – E-Commerce-System



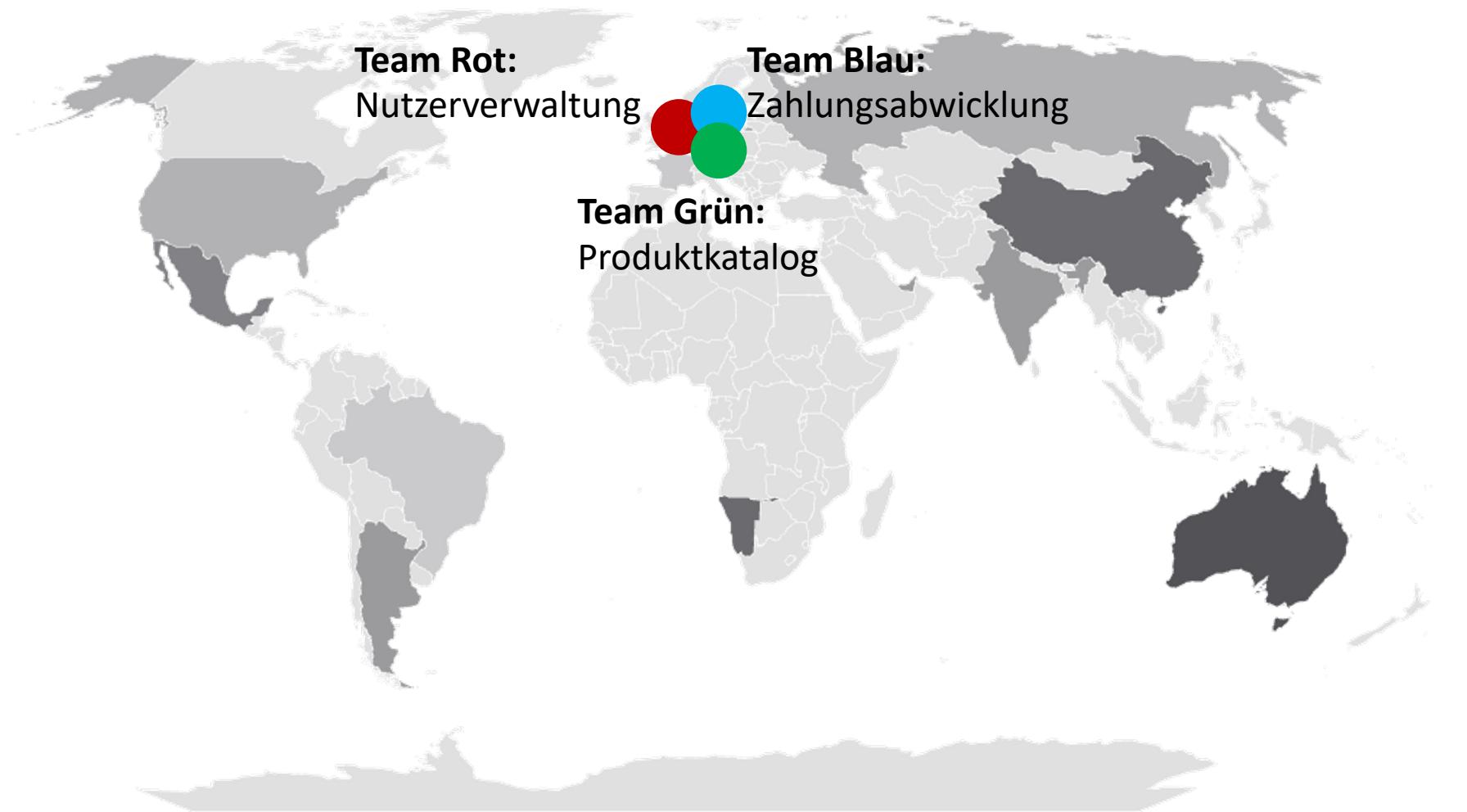
Mögliche (positive) Auswirkungen auf die Architektur:

- Modulare Architektur
- Kommunikation zwischen Komponenten über dokumentierte Schnittstellen

Beispiel – E-Commerce-System



Beispiel – E-Commerce-System



Beispiel – E-Commerce-System



Mögliche (negative) Auswirkungen auf die Architektur:

- Monolithische Architektur
- Schlechte Modularisierung

Kopplung & Kohäsion

Kopplung

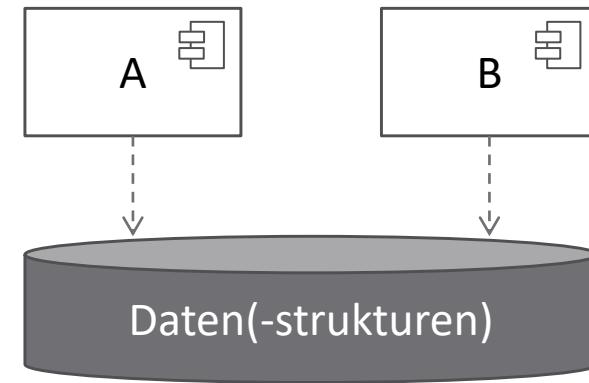
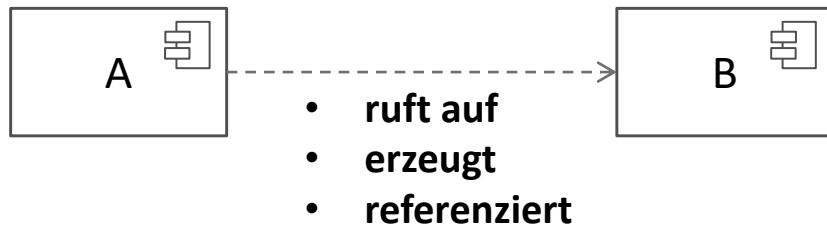
- Maß der Abhängigkeit zwischen verschiedenen Modulen oder Komponenten eines Softwaresystems
- Gibt Aufschluss darüber, wie stark Änderungen an Komponenten andere beeinflussen

Ziel ist eine **geringe** Kopplung zwischen Komponenten

Eine geringe Kopplung verbessert die

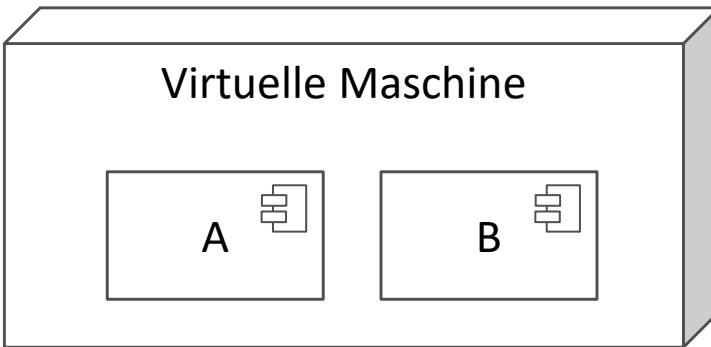
- Wartbarkeit
- Flexibilität
- Testbarkeit

Arten von Kopplung

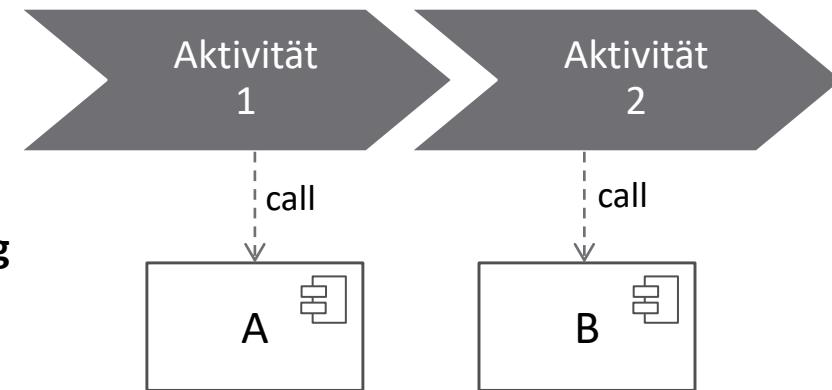


Nutzung gemeinsamer Daten(strukturen)

Nutzung gemeinsamer Ressourcen



Zeitliche Kopplung



Beispiele

• Hohe Kopplung

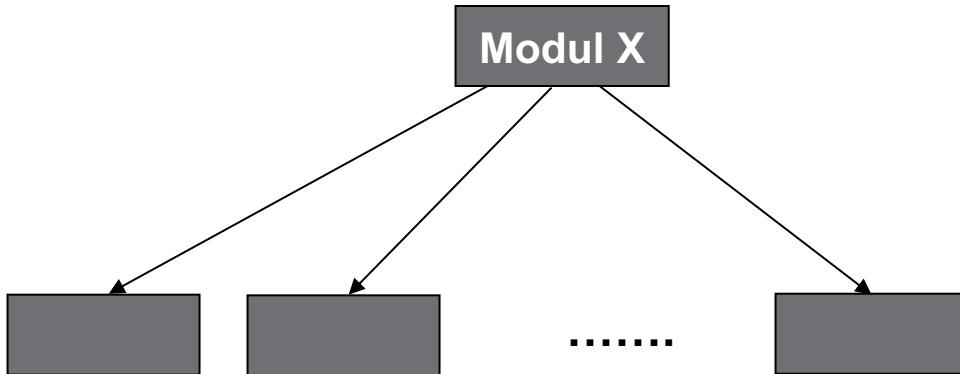
- Ein Modul ändert den Zustand eines anderen Moduls direkt, indem es dessen interne Datenstrukturen modifiziert. Das führt dazu, dass jede Änderung in einem Modul zwangsläufig Änderungen in anderen Modulen nach sich zieht.

• Niedrige Kopplung

- Ein Modul kommuniziert mit einem anderen Modul ausschließlich über eine klar definierte API, ohne interne Details preiszugeben. Änderungen an der internen Implementierung eines Moduls haben somit keine Auswirkungen auf andere Module, solange die API unverändert bleibt.

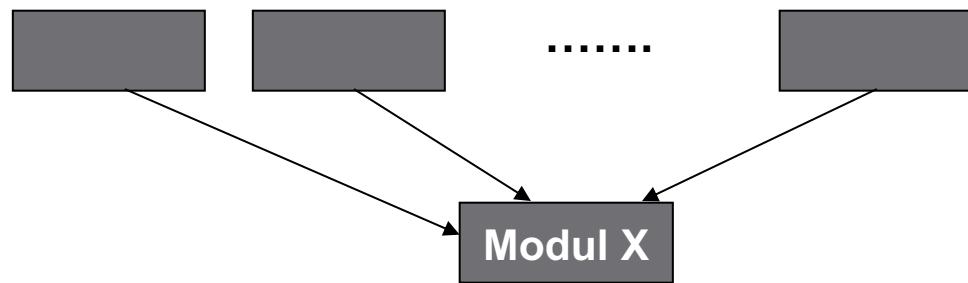
Efferente Kopplung/Efferent Coupling (Ce)

- Anzahl von Komponenten, die eine bestimmte Komponente referenziert
- 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 Wiederverwendung
 - 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 klares 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

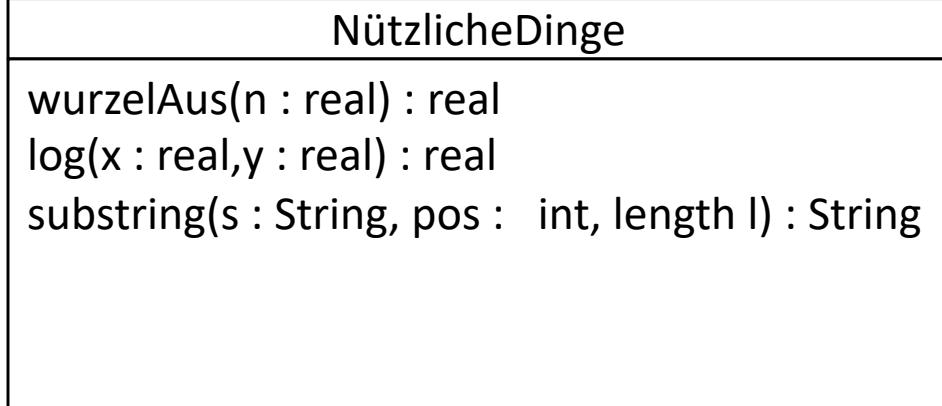
Ziel ist eine **hohe** Kohäsion innerhalb einer Komponente.

Eine hohe Kohäsion verbessert die

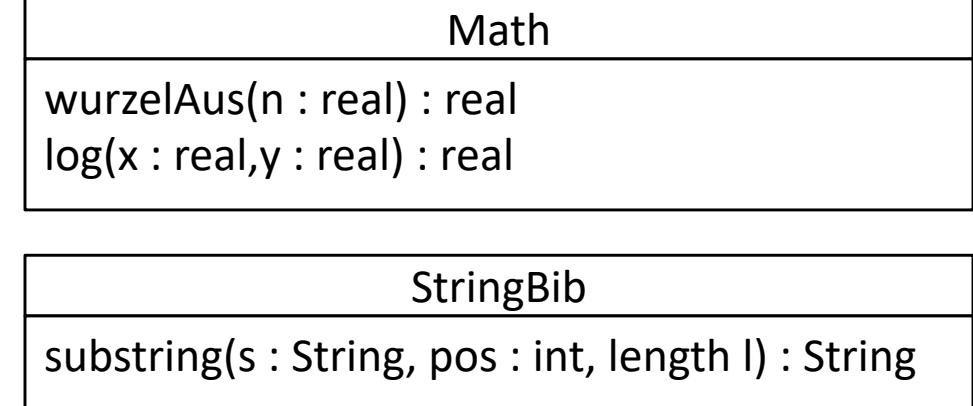
- Wartbarkeit
- Flexibilität
- Testbarkeit

Kohäsion

- Bezieht sich auf das Maß, in dem die Elemente oder Funktionen innerhalb einer Komponente zusammengehören oder gemeinsam ein klares und zusammenhängendes Ziel verfolgen



Schlechte Kohäsion!



Besser!

Messung von Kohäsion

LCOM - Lack of Cohesion Metric

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

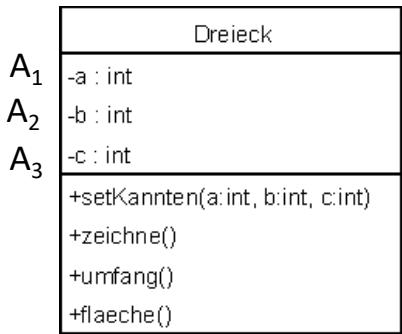
$$LCOM = \frac{\left(\frac{1}{a} \sum_{j=1}^a n(A_j) \right) - m}{1 - m}$$

Je höher der LCOM-Wert, desto geringer die Kohäsion

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

Messung von Kohäsion LCOM - Beispiel

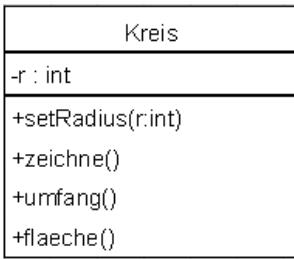
$$LCOM = \frac{\left(\frac{1}{a} \sum_{j=1}^a n(A_j) \right) - m}{1 - m}$$



Dreieck

- a = 3
- m = 4
- n(A₁) = 4
- n(A₂) = 4
- n(A₃) = 4

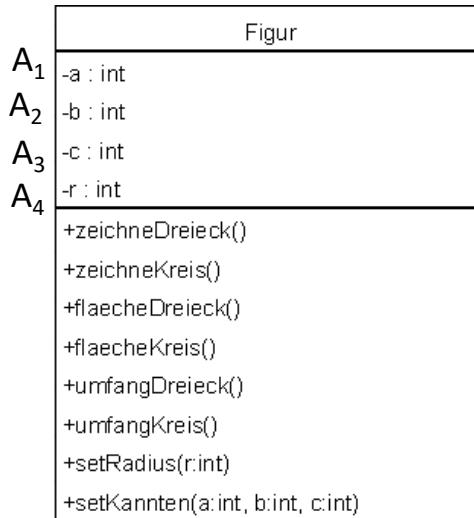
=> LCOM = 0



Kreis

- a = 1
- m = 4
- n(A₁) = 4

=> LCOM = 0



Figur

- a = 4
- m = 8
- n(A₁) = 4
- n(A₂) = 4
- n(A₃) = 4
- n(A₄) = 4

=> LCOM = 4/7

Jedes Attribut wird in jeder Methode verwendet

Domain-Driven Design (DDD)

Verstehen sich alle Beteiligten wirklich?



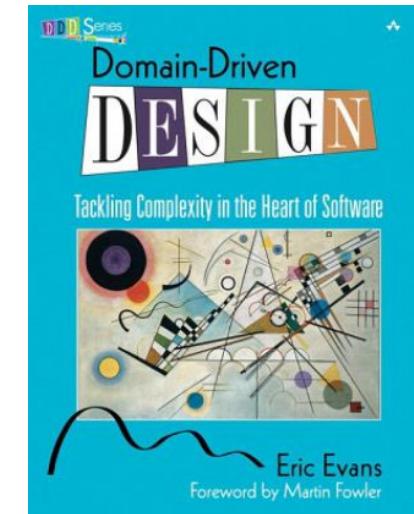
“The information model is a fundamental tool for managers in the digital change.” Stefan Berner, July 2019

Auch Domänenmodell genannt

Grundlagen

- Domain Driven Design (DDD) ist eine Herangehensweise 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 den Technologien liegen, die wir verwenden.
 - Stattdessen sollte sich unser Hauptaugenmerk auf die geschäftliche Seite richten, also auf den fachlichen Bereich, den wir mit unserer Software unterstützen wollen – kurz: auf die Domäne.
- Das ist es, was ich mit Domain-driven Design meinte“

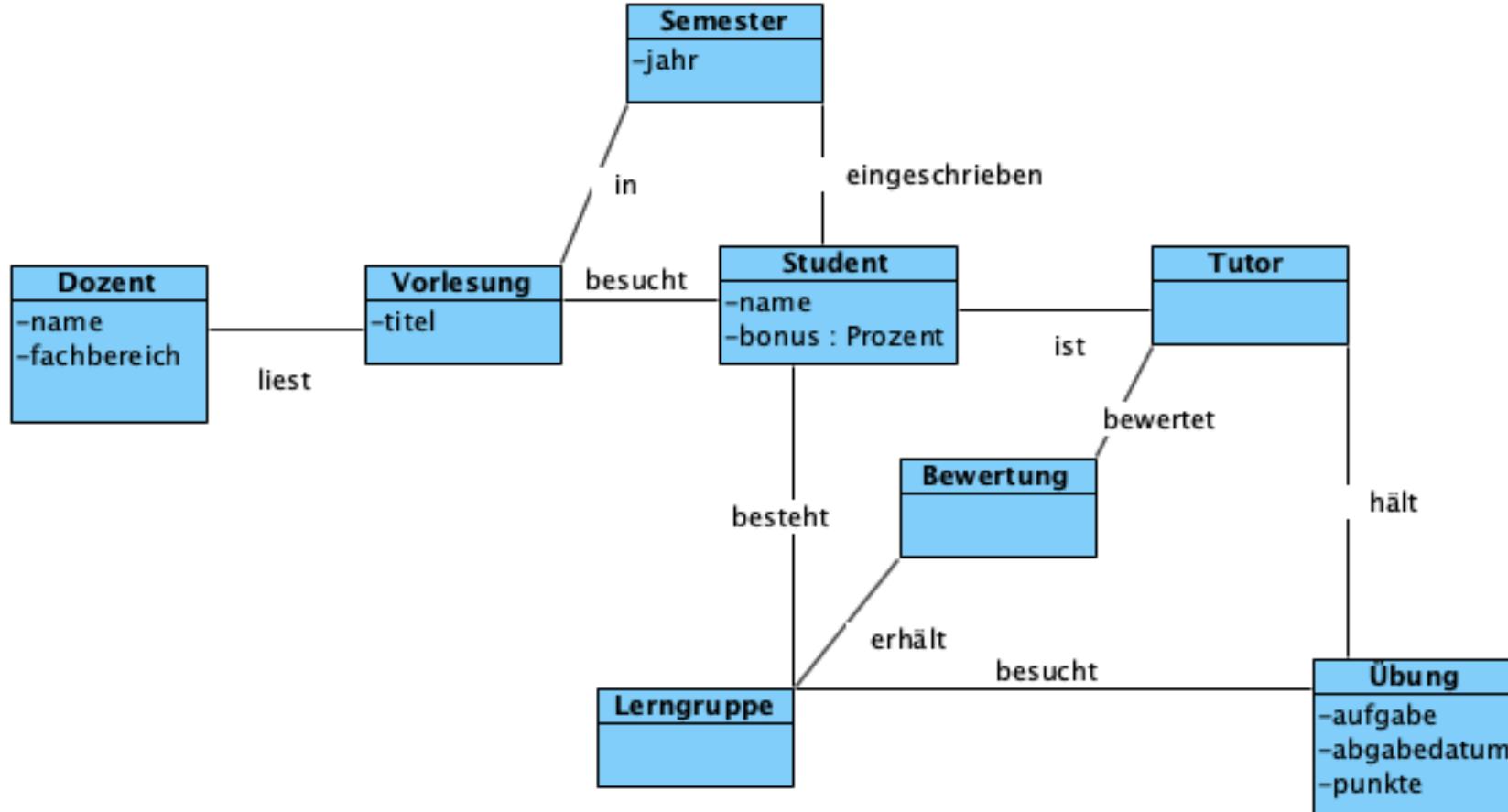
Auszug aus einer Rede vom Erfinder von DDD, Eric Evans.



Ubiquitous Language

- DDD betont die Verwendung einer **allgegenwärtigen Sprache** (Ubiquitous Language), die von 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

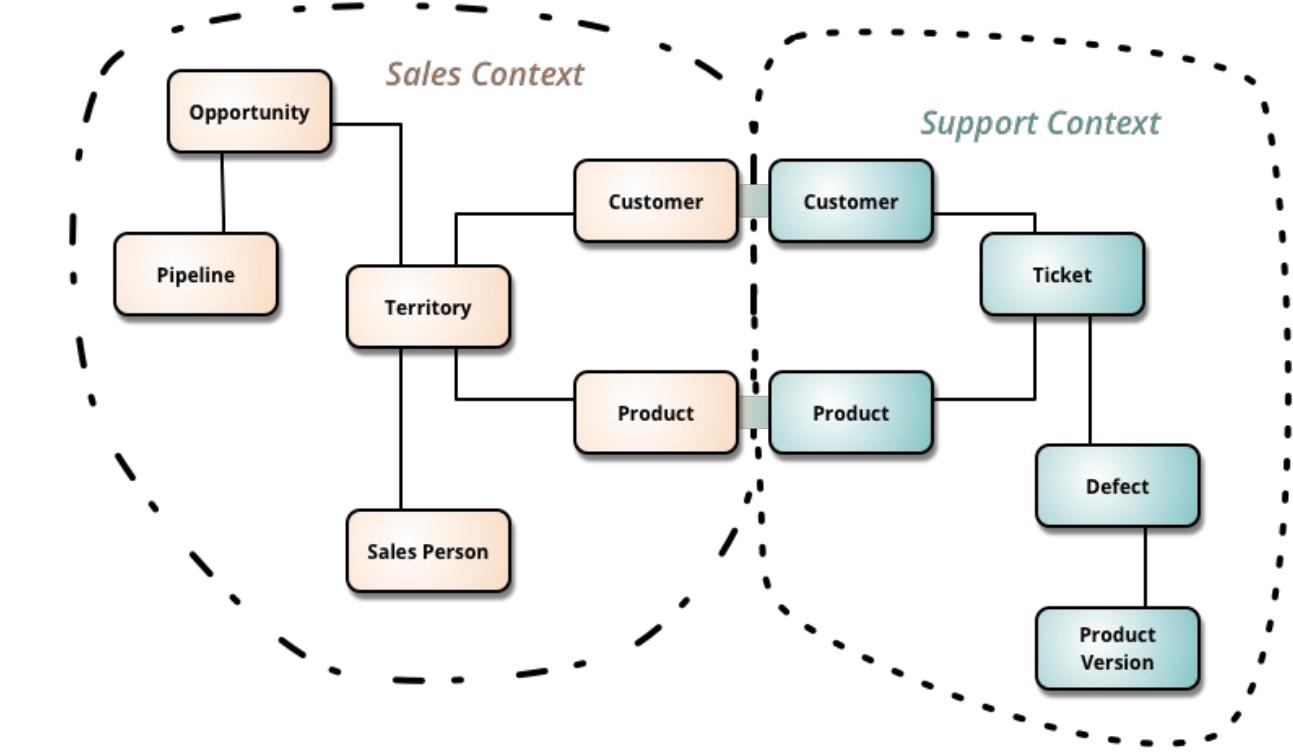
Beispiel Domänenmodell



Weitere Konzepte von DDD

- Kontextgrenzen (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)
 - Eine Anti-Corruption Layer (ACL) ist ein 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 den möglicherweise unpassenden Modellen oder Konzepten eines anderen Kontexts geschützt wird.

Beispiel – Bounded Contexts



- DDD erkennt an, dass „eine vollständige Vereinheitlichung des Domänenmodells für ein großes System weder machbar noch kosteneffizient ist“
- Stattdessen unterteilt DDD ein großes System in Bounded Contexts, von denen jeder ein vereinheitlichtes Modell haben kann

- „Customer“ im Kontext des **Supports**
 - Eine Person oder eine Organisation sein, die Unterstützung, Beratung oder Lösungen von einem Supportteam oder einer Supportabteilung benötigt.
- „Customer“ im Kontext des **Sales**
 - Eine Person oder Organisation, die ein Produkt oder eine Dienstleistung von einem Unternehmen erwirbt oder potenziell erwerben könnte.
 - Die Zielgruppe, auf die das Vertriebsteam abzielt, um Produkte oder Dienstleistungen zu verkaufen.

Event Storming

Workshop-basierte Methode,
um schnell herauszufinden,
was in einer Domäne
„passiert“.

>>The business process is "stormed out" as a series of domain events which are denoted as stickies.<<

Methode, um Fachexperten und Softwareentwickler zusammenzubringen.



[Wikipedia]

Event Storming - Ablauf

1. Vorbereitung

- Stakeholder einladen (z.B. Entwickler, Domänenexperten, Produktmanager) und Moderator bestimmen
- Große Arbeitsfläche/Wand bereitstellen, ebenso viele bunte Haftnotizen und Marker

2. Identifikation von Domäneneignissen

- Teilnehmer schreiben wichtige Ereignisse auf orangene Haftnotizen.
- Ein Ereignis ist etwas, das passiert und für das Geschäft von Bedeutung ist
 - „Order Placed“, „Payment Processed“
- Ereignisse in chronologischer Reihenfolge an die Wand kleben

3. Erweitern und Vertiefen

- Kommandos definieren (blaue Haftnotizen), die Ereignisse auflösen
- Ein Kommando ist eine Benutzeraktion oder eine Entscheidung im System
 - „Place Order“, „Process Payment“

Weitere Designprinzipien

Postel's Law (Robustheitsprinzip)

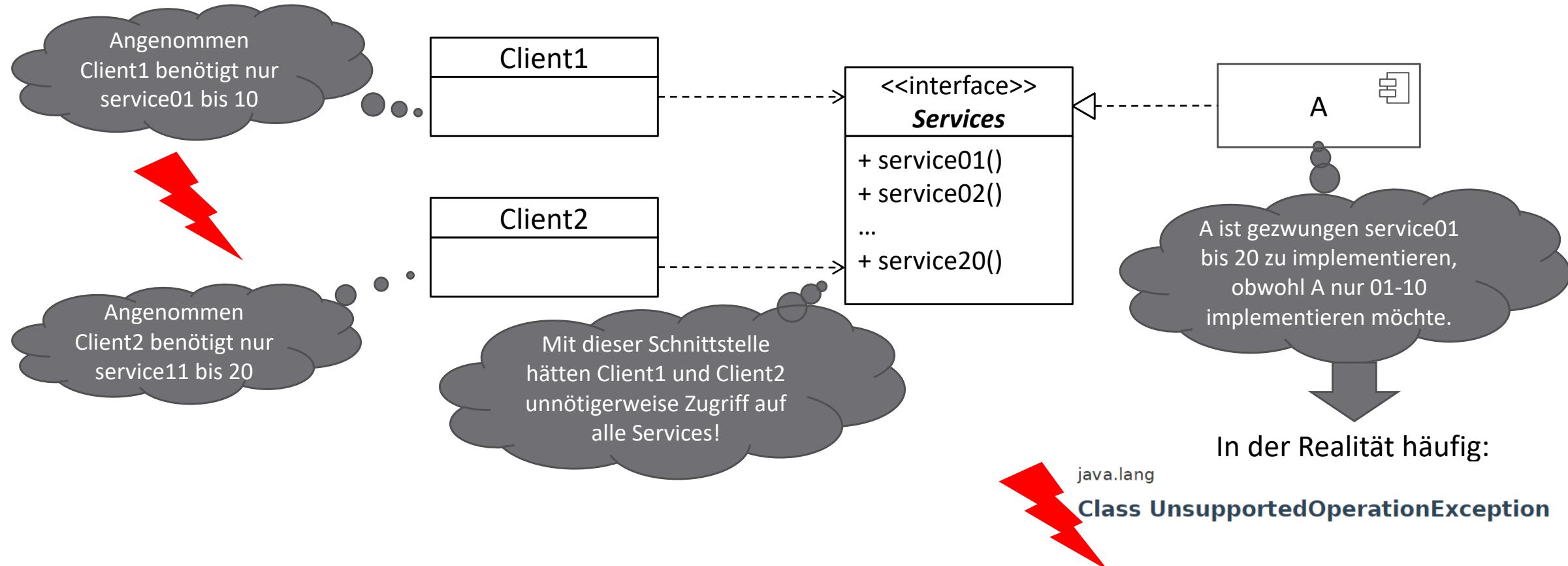
"Be conservative in what you do, be liberal in what you accept."

Jonathan Postel

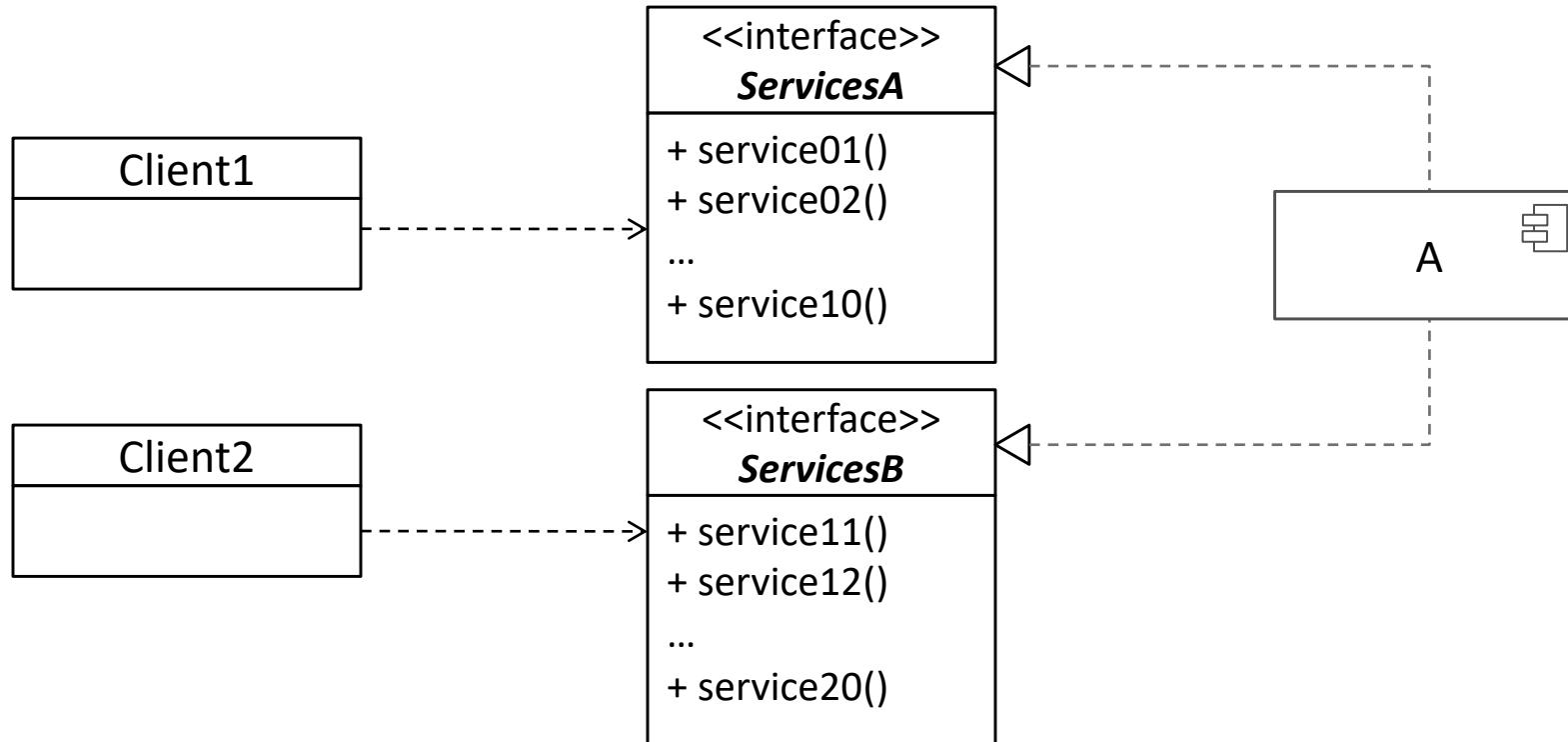
- Heißt: Sei streng bei dem, was du tust, und offen bei dem, was du von anderen akzeptierst.
- Aus Sicht eines Schnittstellen-Konsumenten:
 - Rechne mit Service-Ausfällen, mit fehlerhaftem Verhalten, ...
- Aus Sicht eines Schnittstellen-Implementierers:
 - Halte dich genau an die Schnittstellen-Spezifikation.
 - Biete nicht nur die aktuelle API an, sondern auch mind. eine vergangene Version. Das gibt dem Verwender Gelegenheit, auf die neue Version umzusteigen („Expand and Contract“).

Interface Segregation Principle

- Clients sollten nicht dazu gezwungen werden, von Schnittstellen (und deren Methoden) abzuhängen, die sie gar nicht brauchen.
- Damit würden sich Änderungen an nicht benötigten Schnittstellen (und deren Methoden) unnötigerweise auf Verwender auswirken.



Interface Segregation Principle



- Schnittstellen sollten 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

Weitere Designprinzipien

- Offen-Geschlossen-Prinzip
 - Ein Modul soll für Erweiterungen offen sein.
 - Ein Modul soll für Änderungen geschlossen sein.
- So-einfach-wie-möglich-Prinzip
 - Albert Einstein:
 - „Mache die Dinge so einfach wie möglich – aber nicht einfacher“
- Information Hiding (Geheimnisprinzip)
 - Kapselung von Komplexität in Komponenten
 - Verwenden von Schnittstellen
 - Siehe Kerneigenschaften einer Komponente
- Regelmäßiges Refactoring und Redesign
- Nutzung von Architekturmustern (siehe nächster Themenbereich)