

No & Low Code

C1TT Continental Institut
für Technologie
und Transformation

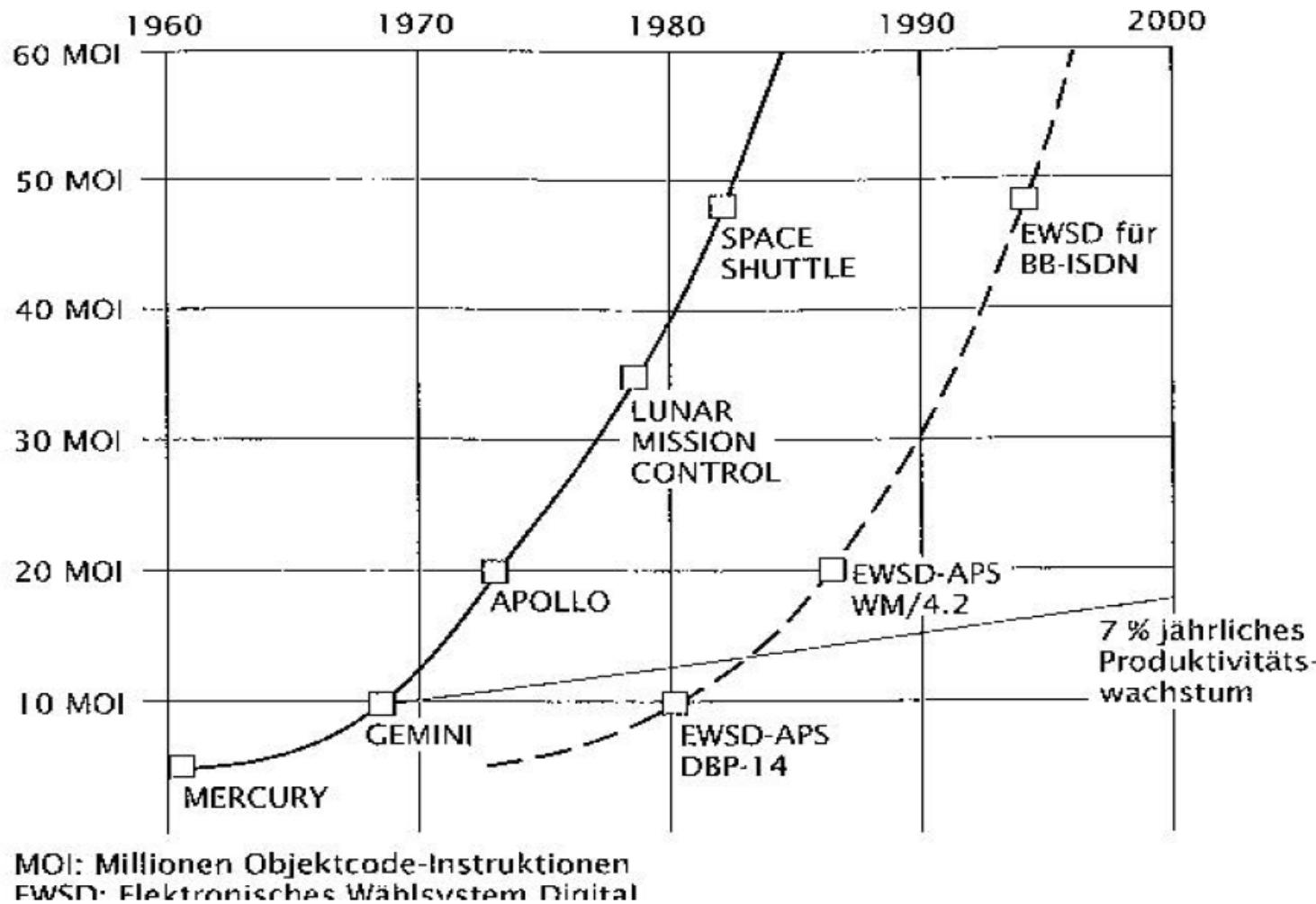
Software-Architektur - Methoden und Techniken der Entwicklung

- Motivation für Software Architektur
- Grundbegriffe und Bedeutung
- Entwicklung und Entwurf von Software Architekturen
- Dokumentation von Software Architektur
- Microservices Best Practice
- Technische Schulden in Software Architekturen abbauen

Funktionalität ist nicht alles!



Software wird immer komplexer



Quelle: Siemens EWSD™ V8.1: 12,5 Millionen LOC, ca. 190.000 S. Dokumentation

Finanzielle Katastrophen

- 1992: Integration des Reservierungssystems SABRE mit anderen Reservierungssystemen abgebrochen: 165 Mio US-\$
- 1997: Entwicklung des Informationssystems SACSS für den Staat Kalifornien abgebrochen: 300 Mio US-\$

Terminkatastrophen

- 1994: Eröffnung des Denver International Airport um 9 Monate verzögert wegen Softwareproblemen im Gepäcktransport-System
- 2004: Toll-Collect – LKW Mautsystem in Deutschland

Technikkatastrophen

- 1999: Fehlstart einer Titan/Centaur-Rakete wegen falscher Software-Version
- 1999: Verlust der Sonde „Mars Climate Orbiter“ wegen falscher Einheitenumrechnung

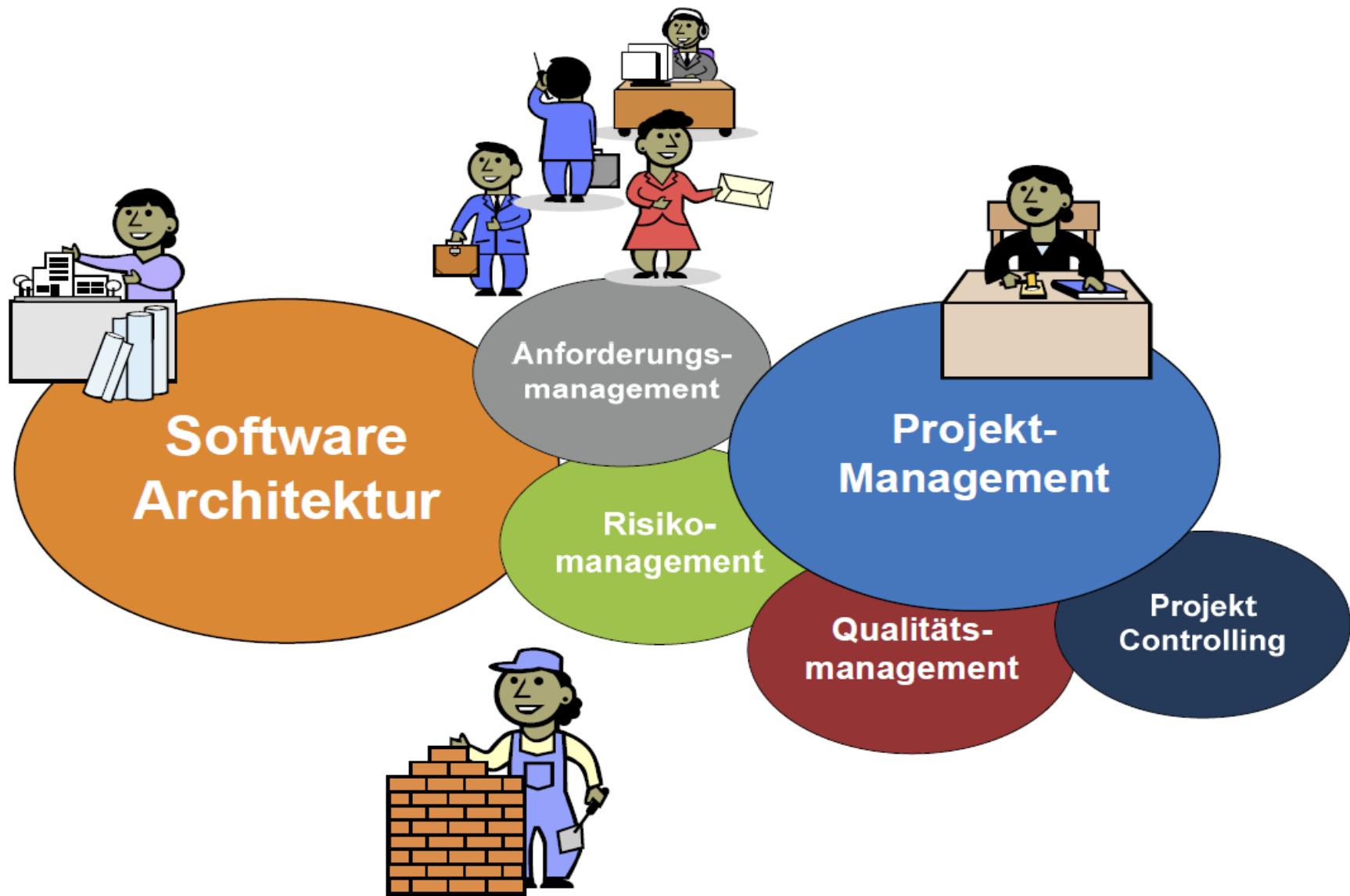
Allgemeine Fehlerursachen beim Scheitern von SW-Entwicklungsprojekten

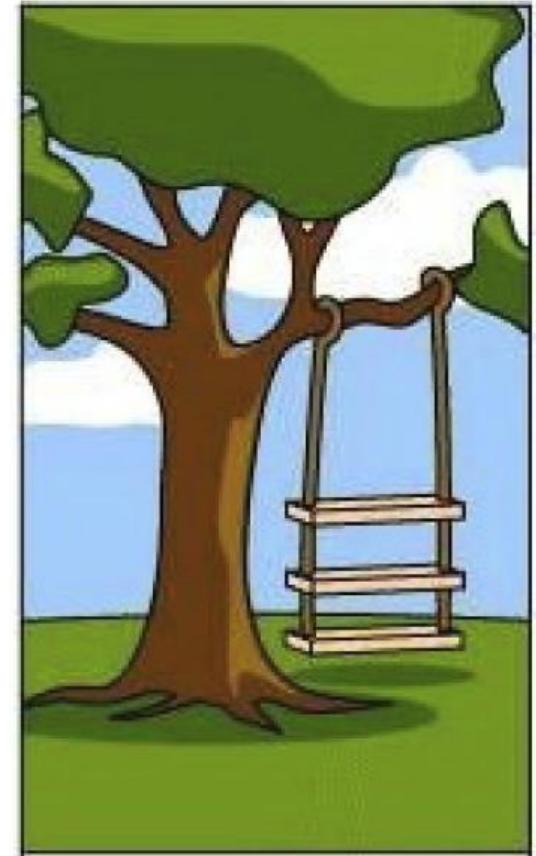
- Fehlende Projektplanung und -steuerung
- Fehlerhafte Aufwandsabschätzungen
- Vernachlässigung der Entwurfsphase
- Ad-hoc Lösungen, fehlende Systematik
- Mangelnde Erfassung der Benutzerbedürfnisse und der Anwendungsumgebung
- Fehlende Dokumentation
- Uneingespielte Entwicklungsteams
- zu spätes Reagieren auf Entwicklungsrisiken technischer oder organisatorischer Natur

1. Lösung: detailbasierende Methoden



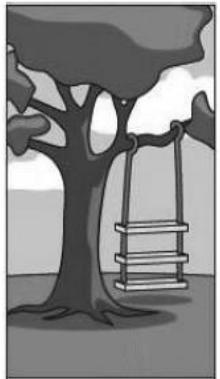
2. Lösung: Integrierender Überblick



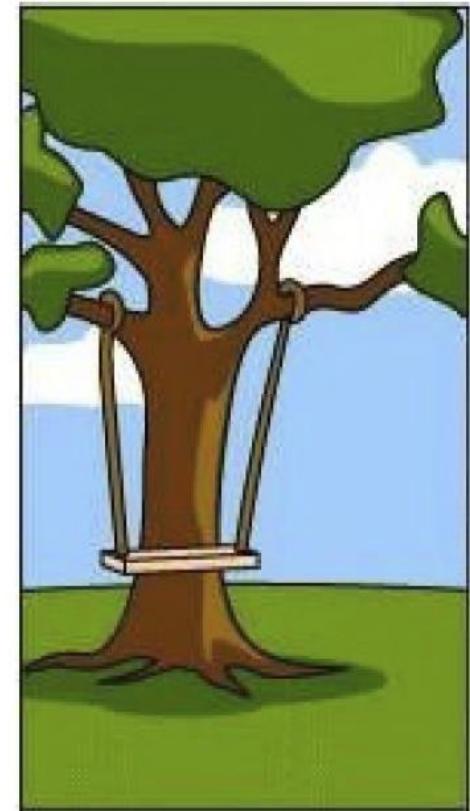


**Was der Kunde
beschrieb**

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb

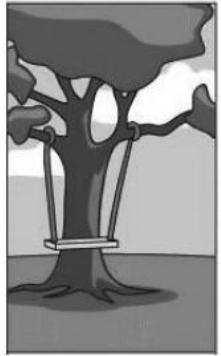


**Verständnis des
Projektleiters**

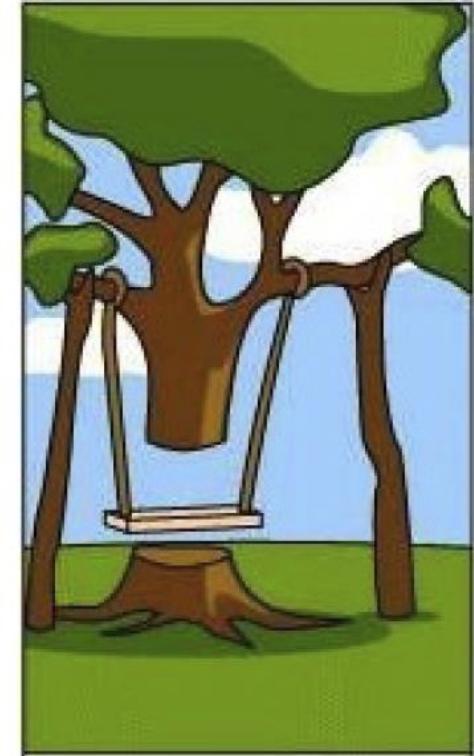
Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb



Verständnis
des Projektleiters



**Bewertung
im Controlling**

Vom Nutzen verständlicher (An-)Sichten



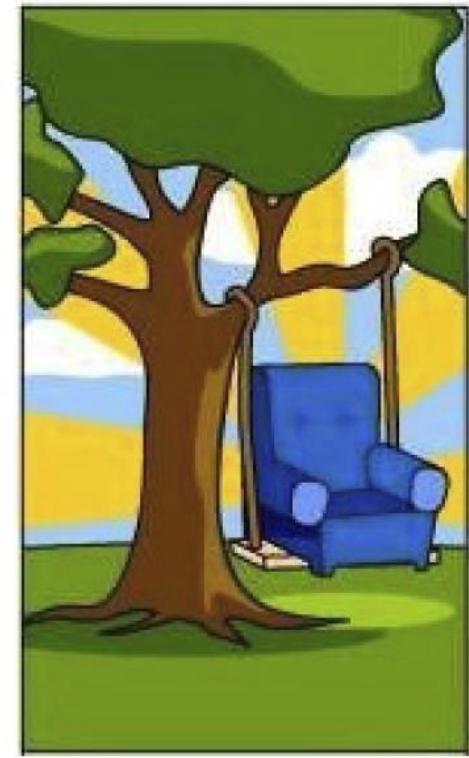
Was der Kunde
beschrieb



Verständnis
des Projektleiters

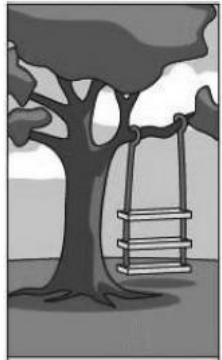


Bewertung
im Controlling



**Festlegung des
Anforderungs-
management**

Vom Nutzen verständlicher (An-)Sichten

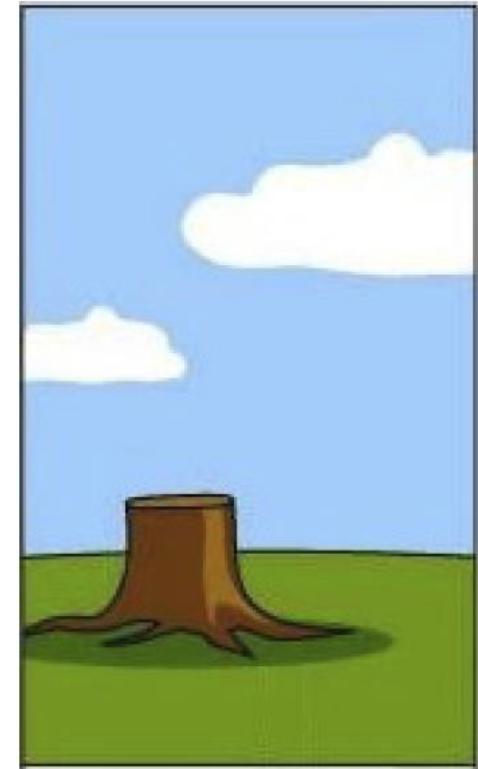


Was der Kunde
beschrieb

Verständnis
des Projektleiters

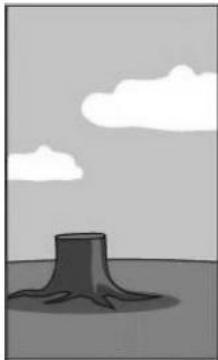
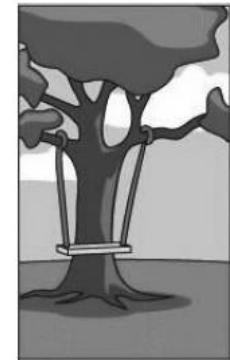
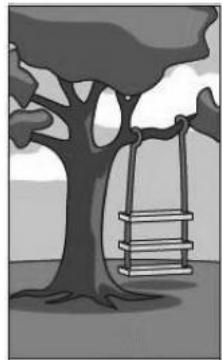
Bewertung
im Controlling

Festlegung des
Anforderungs-
Management



**Bewertung
der Risiken**

Vom Nutzen verständlicher (An-)Sichten



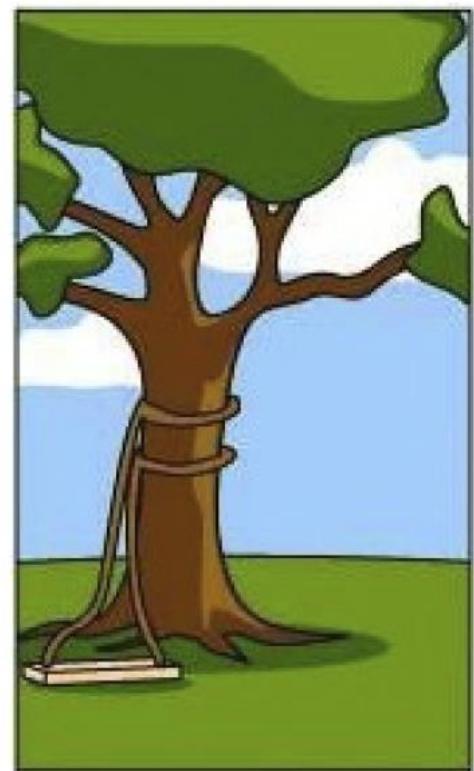
Was der Kunde
beschrieb

Verständnis
des Projektleiters

Bewertung
im Controlling

Festlegung des
Anforderungs-
Management

Bewertung
der Risiken



**Umsetzung in
der Entwicklung**

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb



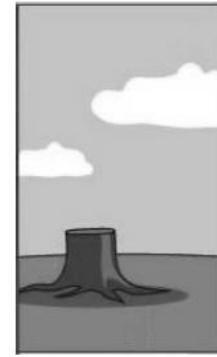
Verständnis
des Projektleiters



Bewertung
im Controlling



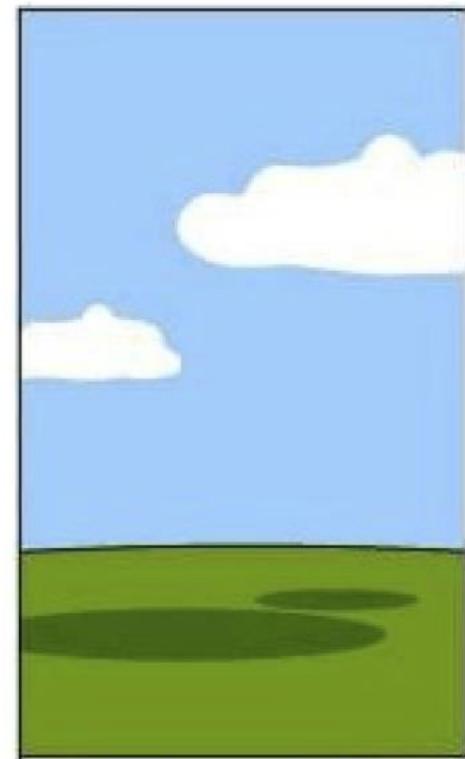
Festlegung des
Anforderungs-
Management



Bewertung
der Risiken

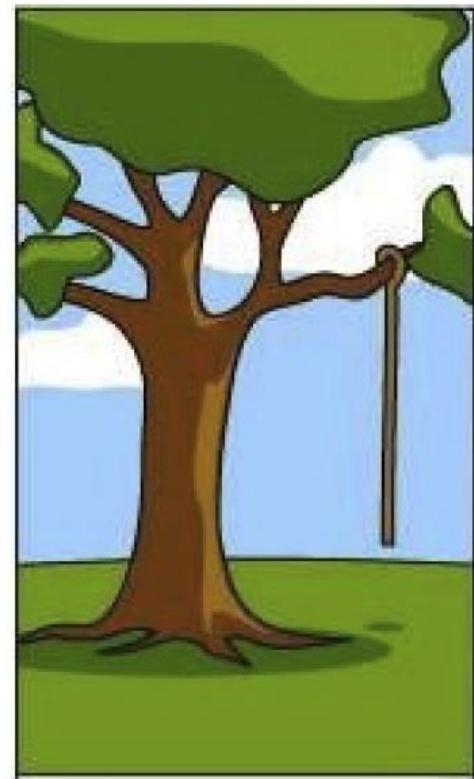
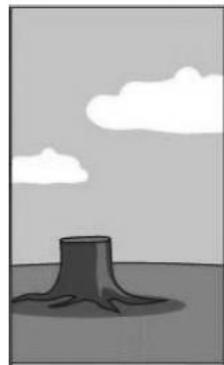
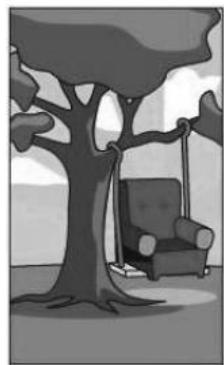
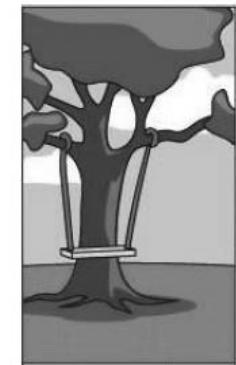
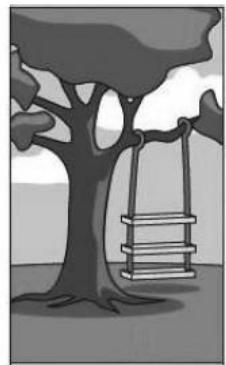


Umsetzung in
der Entwicklung



Dokumentation

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb

Verständnis
des Projektleiters

Bewertung
im Controlling

Festlegung des
Anforderungs-
Management

Bewertung
der Risiken

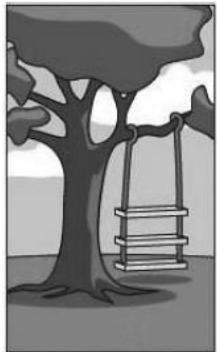


Umsetzung
der Entwicklung

Dokumentation

**Was ausgeliefert
wurde**

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb



Verständnis
des Projektleiters



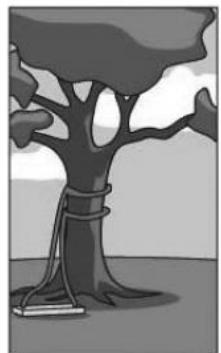
Bewertung
im Controlling



Festlegung des
Anforderungs-
Management



Bewertung
der Risiken



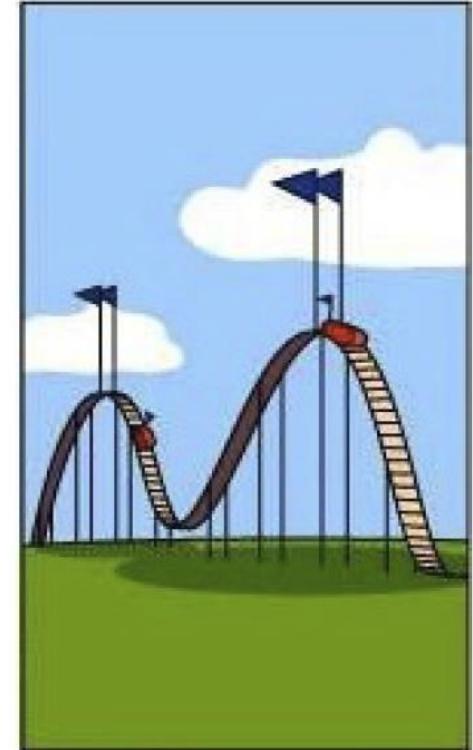
Umsetzung
der Entwicklung



Dokumentation



Was ausgeliefert
wurde



**Was in Rechnung
gestellt wurde**

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde
beschrieb



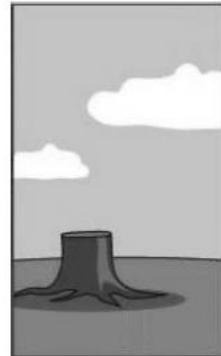
Verständnis
des Projektleiters



Bewertung
im Controlling



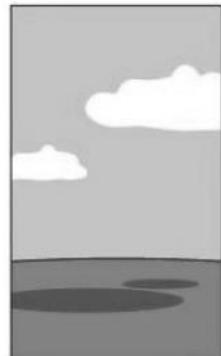
Festlegung des
Anforderungs-
Management



Bewertung
der Risiken



Umsetzung
der Entwicklung



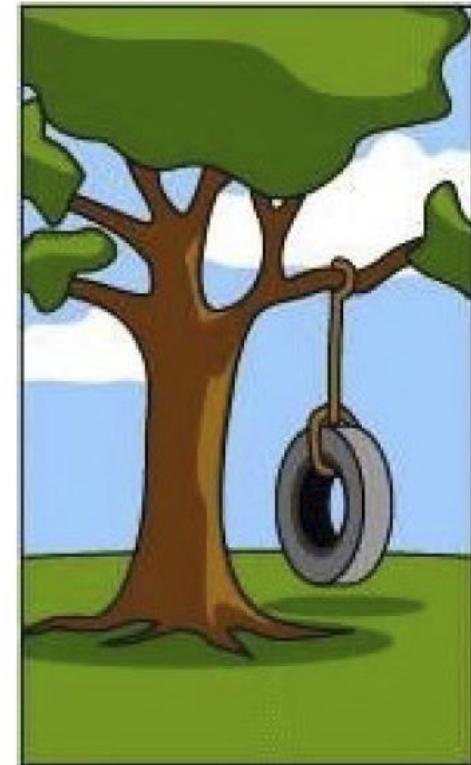
Dokumentation



Was ausgeliefert
wurde



Was in Rechnung
gestellt wurde



**Was der Kunde
gebraucht hätte**

Vom Nutzen verständlicher (An-)Sichten



Was der Kunde beschrieb



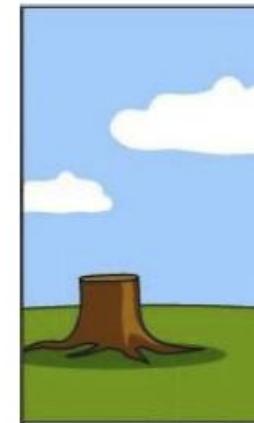
Verständnis des Projektleiters



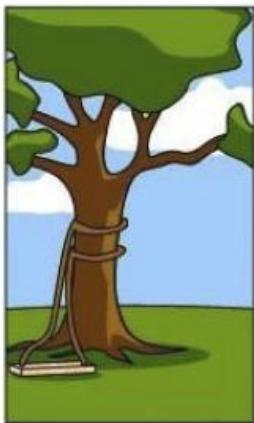
Bewertung im Controlling



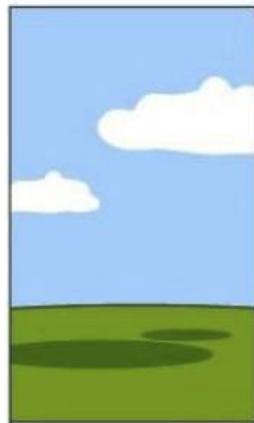
Festlegung des Anforderungs-Management



Bewertung der Risiken



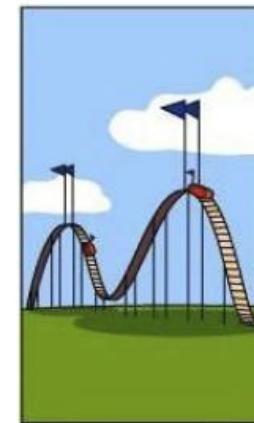
Umsetzung der Entwicklung



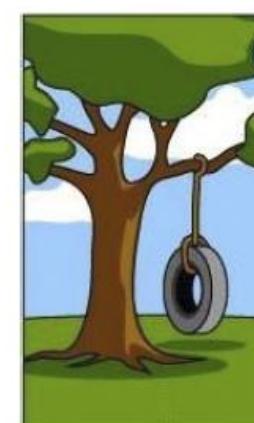
Dokumentation



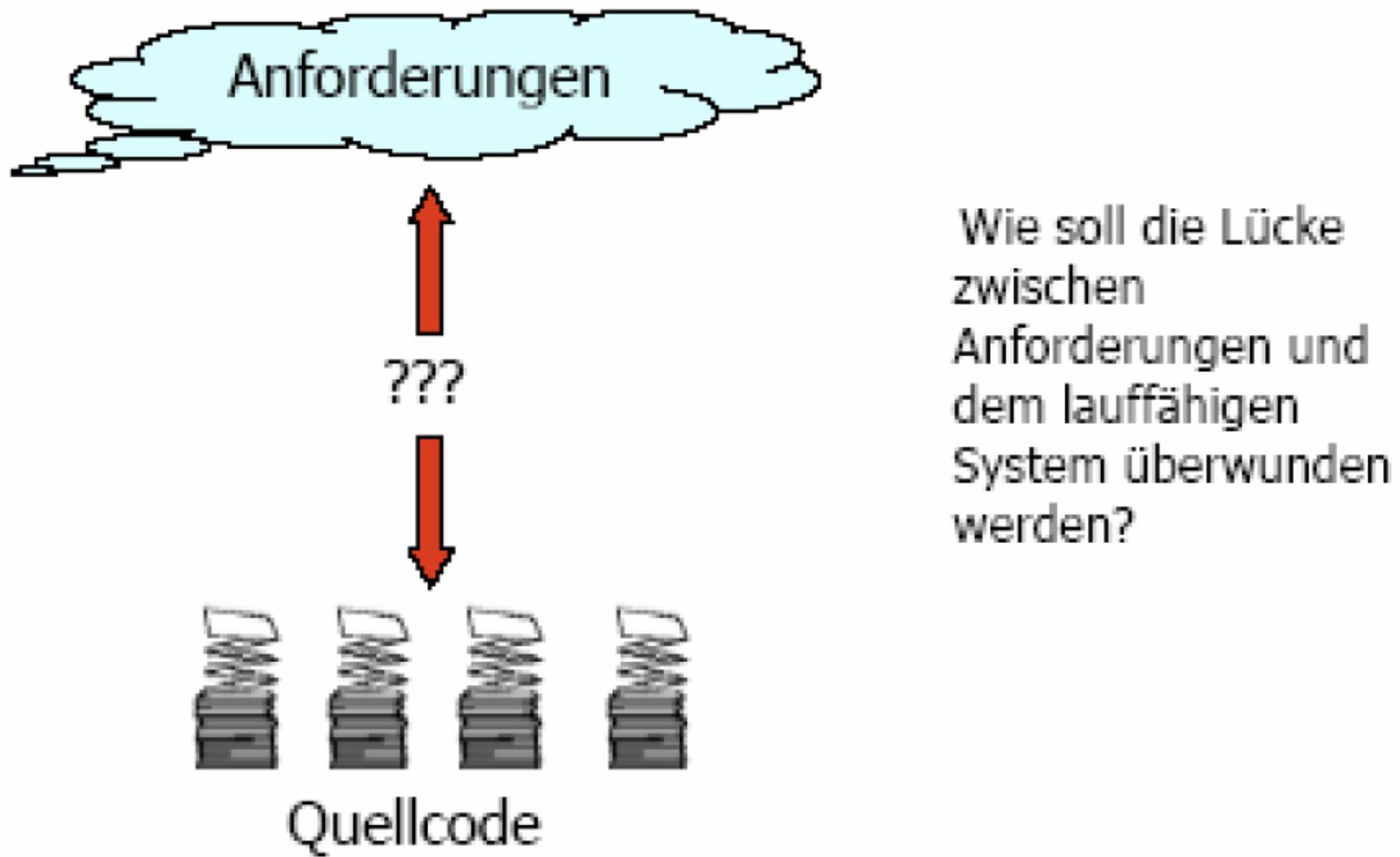
Was ausgeliefert wurde



Was in Rechnung gestellt wurde

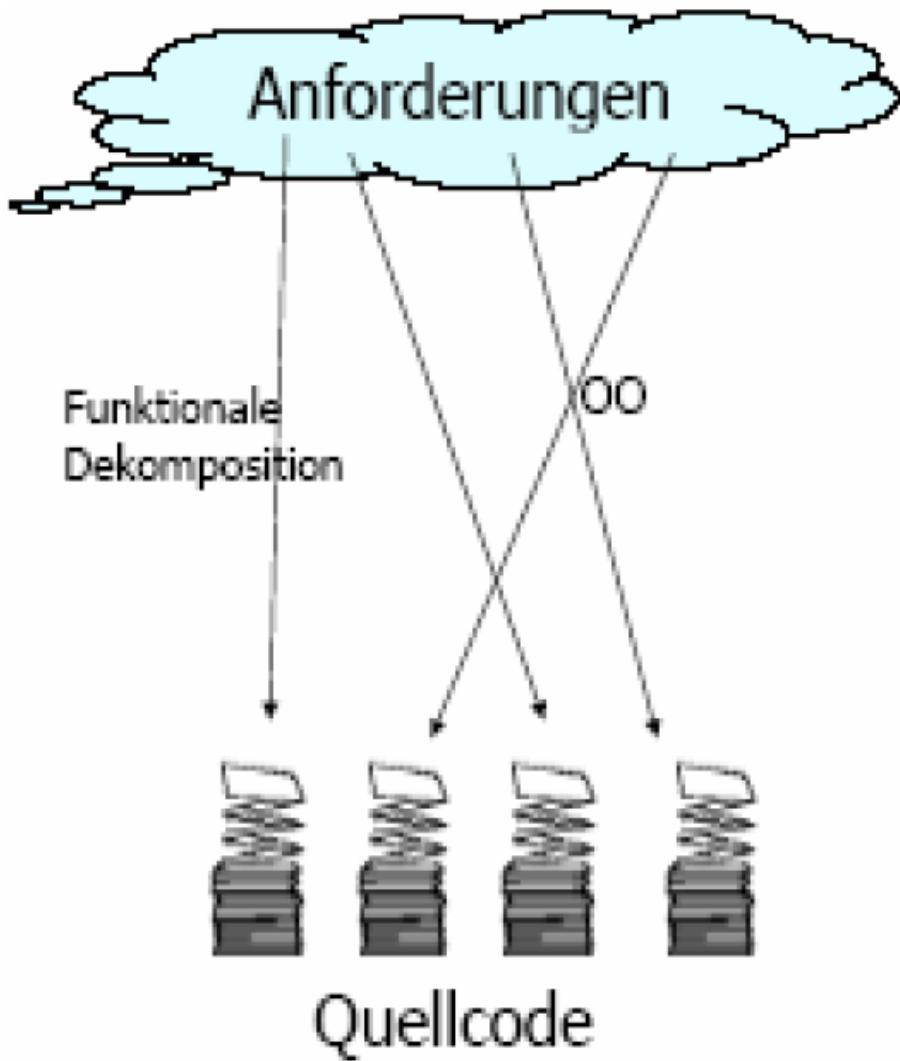


Was der Kunde Gebraucht hätte





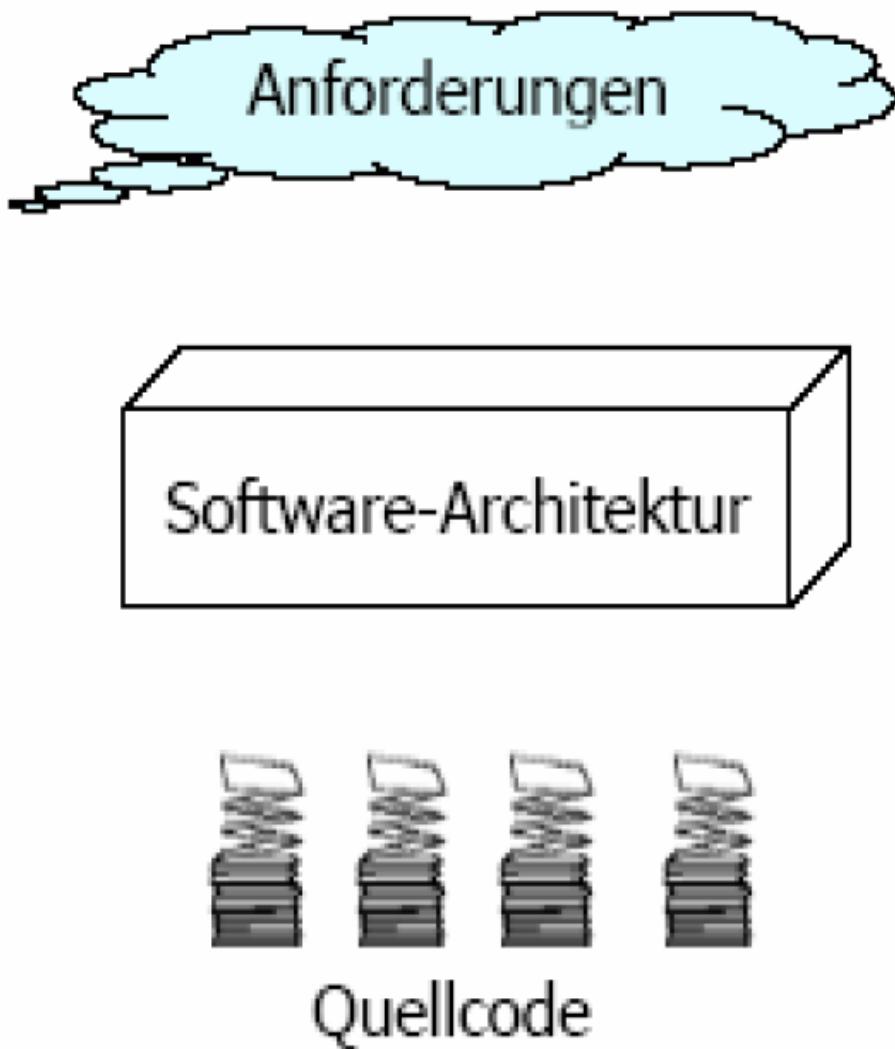
- Unvorhersehbar
- Ungeplant, nicht wiederholbar
- Benötigt „Künstler“
- Teuer



- systematisch
- besser vorhersehbar

ABER

- Benötigt viel Erfahrung
- Für jedes Produkt vollst. neu auszuführen
- Erst anhand des Resultats analysierbar



- Grobes Systemkonzept
- Systemweite Abstraktion
- Struktur der Strukturen des Systems
- Wiederverwendung abstrakter Ideen

Vorteile

- Langfristige Planung!
- Verstehbar!
- Analysierbar!
- Wiederverwendbar!

Bücher

- Starke, Gernot: Effektive Software-Architekturen. Ein praktischer Leitfaden. Hanser, 2005.
- Andresen, Andreas: Komponentenorientierte Softwareentwicklung, Hansa-Verlag, 2004.
- Siedersleben, Johannes: Moderne Softwarearchitektur. dpunkt Verlag (Heidelberg), 2004.
- Oestereich, Bernd: Objektorientierte Softwareentwicklung, Analyse und Design mit Unified Modeling Language

Relevante Webseiten

- Software-Engineering Institute (SEI) der Carnegie-Mellon University
<http://www.sei.cmu.edu/architecture/definitions.html>
- Ressourcen für SW-Architekten <http://www.arc42.de>
- UML 2.1.1 Specification <http://www.uml.org>
- UML 2.0 Notationsübersicht <http://www.oose.de/downloads/uml-2-Notationsuebersichtoose.de.pdf>
- SYSML Notation http://www.oose.de/downloads/sysml_notationsuebersicht.pdf
- Volere Requirements Specification Template Edition 11 – 2006/02
<http://www.systemsguild.com/GuildSite/Robs/Template.html>

Grundbegriffe und Bedeutung

von Software Architekturen

- **Definitionen von Softwarearchitektur**
- Nutzen und Ziele von Softwarearchitektur
- Softwarearchitektur in Software-Lebenszyklus
- Aufgaben und Verantwortung von Softwarearchitekten

- Es gibt keine allgemein gültige Definition von Software Architektur
- Literatur und Wissenschaft bieten eine Menge an unterschiedlichen Definitionen
- Der Begriff ist schwer zu greifen
- Die Aufgaben sind vielschichtig und oftmals unterschiedlich definiert
- Wir erkennen jedoch schlechte Architektur

Definitionsversuch von Softwarearchitektur

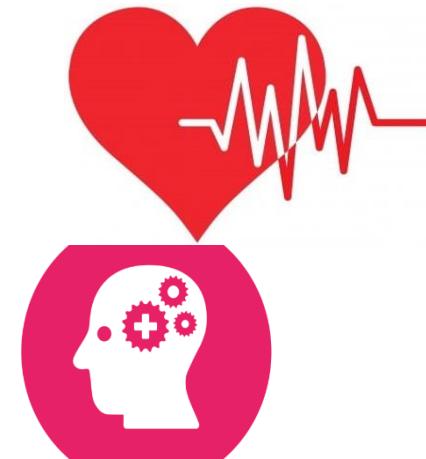
Die Architektur eines Softwaresystems definiert dessen Komponenten und deren Zusammenwirken über Schnittstellen. Sie beschreibt die Struktur von Komponenten. Architektur betrachtet sowohl statische als auch dynamische Aspekte und zeigt damit sowohl den Bauplan als auch den Ablaufplan für Software auf.

Architektur macht Komplexität erst beherrschbar. Wichtigstes Hilfsmittel ist also die Analyse. Eine Aufgabe wird zerlegt und die Komponenten und Schnittstellen den Anforderungen entsprechend angeordnet.

Matthias Geirhos

- Womit beschäftigt sich Softwarearchitektur?
 - die Struktur und Komponenten eines Softwaresystems zu identifizieren,
 - diese in Beziehung zueinander zu bringen und die Art dieser Beziehungen zu erkennen und zu beschreiben,
 - die Konfiguration und deren Eigenschaften zu bestimmen,
 - die Infrastruktur um diese Komponenten herum richtig auszuwählen und richtig anzuwenden, und
 - die einzelnen Softwaresysteme so miteinander zu verbinden, dass ein funktionierendes Ganzes entsteht.
- *Abstrakt definiert, geht es um **Komponenten, Schnittstellen und deren Beziehungen***

- Der Neurologe, der Orthopäde, der Dermatologe haben unterschiedliche Sichten auf den menschlichen Körper
- Der Kardiologe konzentriert sich auf:
- Die Kinesiologie und die Psychiatrie sehen sich unterschiedliche Aspekte des kompletten Verhaltens an
- All diese Sichten haben unterschiedliche Darstellungen mit unterschiedlichen Eigenschaften. Jedoch sind diese überlappend und miteinander verbunden
- Alle zusammen beschreiben den menschlichen Körper
- Und so sollte es auch mit Software sein



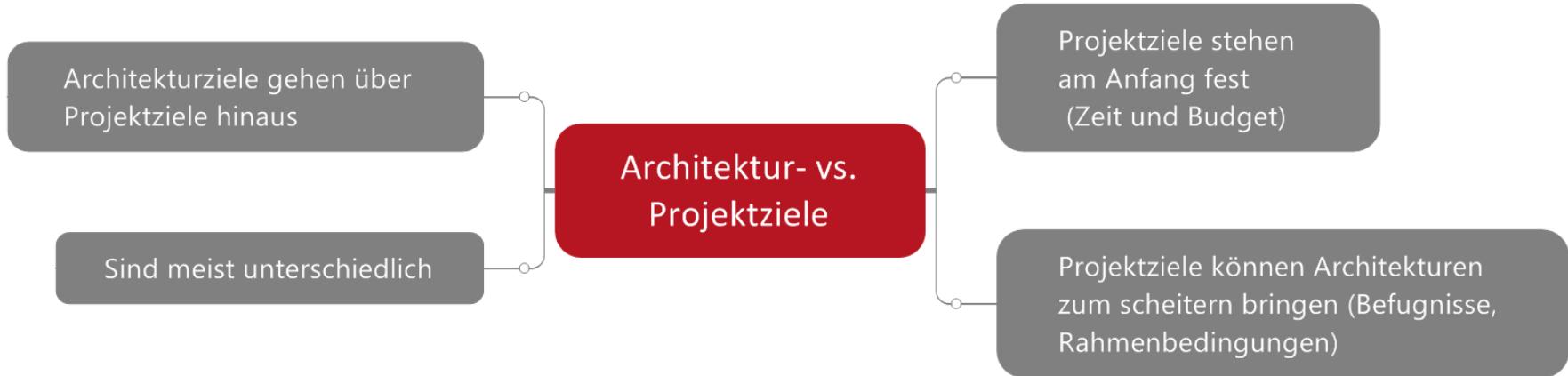
- Definitionen von Softwarearchitektur
- **Nutzen und Ziele von Softwarearchitektur**
- Softwarearchitektur in Software-Lebenszyklus
- Aufgaben und Verantwortung von Softwarearchitekten

Ziele einer Architektur

- Orientiert sich an funktionalen und qualitativen (nicht-funktionale) Anforderungen
- Macht Implementierung planbar und kontrollierbar
- Stellt die Anpassbarkeit und Erweiterbarkeit sicher
- Orientiert sich an Einflussfaktoren des Unternehmens

- Die Architektur sollte das Produkt einer kleinen Gruppe mit einem identifizierten technischen Lead sein
- Das Architektur-Team erstellt die Architektur auf Basis einer Prioritätenliste von gut definierten Qualitätsanforderungen
- Die Architektur sollte mittels Sichten dokumentiert werden. Die Sichten müssen die Bedürfnisse und Sorgen der jeweiligen Stakeholder adressieren
- Die Architektur sollte inkrementell entwickelt werden

- Die Architekturmodule sollten gut strukturiert sein, deren funktionale Implementierungen sollten auf den Prinzipien von Information Hiding und Separation of Concerns beruhen
- Es muss beschrieben sein, welche Architekturmuster und Entscheidungen des Architekten, welche Qualitätsanforderungen unterstützen
- Die Architektur sollte nie auf eine bestimmte Version eines kommerziellen Produkts oder Werkzeug beruhen



- Systeme werden geschaffen um Projekt-/Geschäftsziele eines oder mehrerer Unternehmen zu unterstützen.
 - Entwicklungsunternehmen möchten Gewinne erzielen, Marktanteile erhöhen, im Geschäft bleiben, ihre Kunden bei der Arbeit unterstützen, ihre Mitarbeiter gewinnbringend einsetzen, ihre Aktionäre glücklich machen oder von all dem etwas
 - Kunden haben ihre eigenen Ziele oder Vorstellungen beim Erwerb eines Systems. Normalerweise sind dies Aspekte wie, das Leben leichter oder produktiver zu gestalten. Andere im Projekt-Life-Cycle beteiligten (Sub)Unternehmen oder Organisationen oder Behörden haben ihre eigenen Ziele im Umgang mit den Systemen
- Architekten müssen die beteiligten Unternehmen/Organisationen und deren Ziele verstehen, da viele dieser Ziele einen wesentlichen Einfluss auf die Architektur haben können

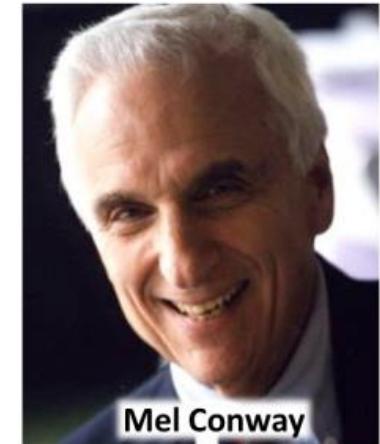
- Jedes Qualitätsmerkmal - wie sichtbare Rückmeldungen für den Benutzer, Plattformunabhängigkeit, hohe Softwaresicherheit oder viele andere Anforderungen sollten sich aus einem übergeordneten Zweck ableiten
 - “Warum wollen wir wirklich schnelle Antwortzeiten?”
 - Das hebt das Produkt gegenüber den Wettbewerbern hervor und kann zu zusätzlichen Marktanteilen führen.
- Manche Projekt-/Geschäftsziele bzw. -zwecke tauchen in den Anforderungen nicht auf
- Andere Projekt-/ Geschäftsziele haben wiederum keine Auswirkung auf die Architektur.
 - Die Vorgabe die Kosten zu minimieren können auch durch Arbeit im Home-Office, besseres Management der Raumtemperatur oder durch Reduzierung der ausgedruckten Dokumente erreicht werden
- **Architekturziele sind meist langfristiger als Projektziele**

- Definitionen von Softwarearchitektur
- Nutzen und Ziele von Softwarearchitektur
- **Softwarearchitektur und Software-Lebenszyklus**
 - Aufgaben und Verantwortung von Softwarearchitekten

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

Oder

Any piece of software reflects the organizational structure that produced it.



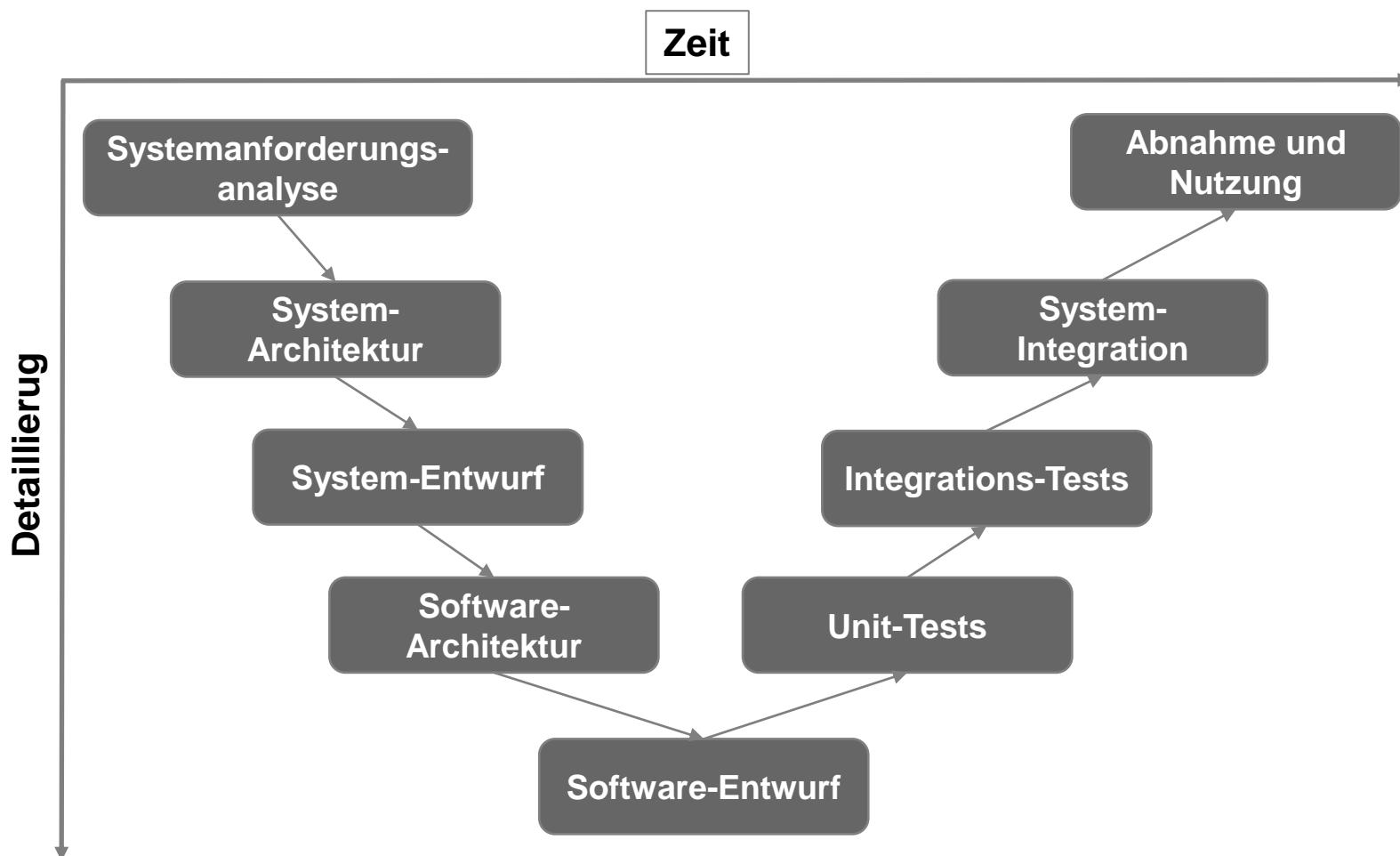
Mel Conway

Das Gesetz von Conway basiert auf der Überlegung, dass für die Definition der Schnittstellen zwischen getrennten Softwaremodulen zwischenmenschliche Kommunikation notwendig ist.

Daher haben die Kommunikationsstrukturen der Organisationen einen großen Einfluss auf die Strukturen dieser Schnittstellen.

- Software Architektur sollte so früh wie möglich im Entwicklungsprozess entstehen
- Softwareentwicklungsprozesse sind Standardvorgehensweisen beim entwickeln von Softwaresystemen
- Es gibt vier vorherrschende Softwareentwicklungsprozesse :
 - Wasserfall
 - Iterativ
 - Agile
 - Modellgetriebene Entwicklung

Software-Entwicklungsprozess als Beispiel: V-Modell



Architekturen sollten iterativ entstehen

Wegen Unsicherheiten und offenen Entscheidungen

Entwurfsprobleme werden dabei frühzeitig entdeckt

Höhere Chancen für Wiederverwendung, Testbarkeit, Flexibilität, Wartbarkeit, ...

Stakeholder können Anforderungen jederzeit ändern

Architektur Evaluierung:

- Funktioniert Architekturevaluierung als Teil eines Agilen Prozesses?
→ Absolut
- Erfüllung der wichtigen Anliegen von Stakeholdern ist ein Grundpfeiler der Agilen Philosophie
- Es ist einfach eine leichtgewichtige Architekturevaluierung zu erstellen
- Solche Evaluierungen können einen wertvollen Beitrag zum Refaktorisieren und für das Re-Design leisten

- Wenn Sie ein großes komplexes System mit relativ stabilen und gut verstandenen Anforderungen erstellen, zahlt es sich aus viel Architekturarbeit im Vorfeld zu erledigen
- Bei größeren Projekten mit *unstabilen* Anforderungen, starten Sie mit der schnellen Erstellung einer Architektur, selbst wenn viele Details noch nicht berücksichtigt werden
 - Bereiten Sie sich darauf vor, diese Architektur zu ändern, je nachdem wie sich die funktionelle Anforderungen und Qualitätsanforderungen ändern
- Bei kleineren Projekten mit unsicheren Anforderungen, versuchen Sie, die Eckpfeiler der Anwendung abzustimmen. Verbringen Sie nicht zu viel Zeit auf Architektur-Design, Dokumentation oder Analyse im Vorfeld
- Es gibt einen “sweet spot” wo sich Architekturplanung im Vorfeld lohnt



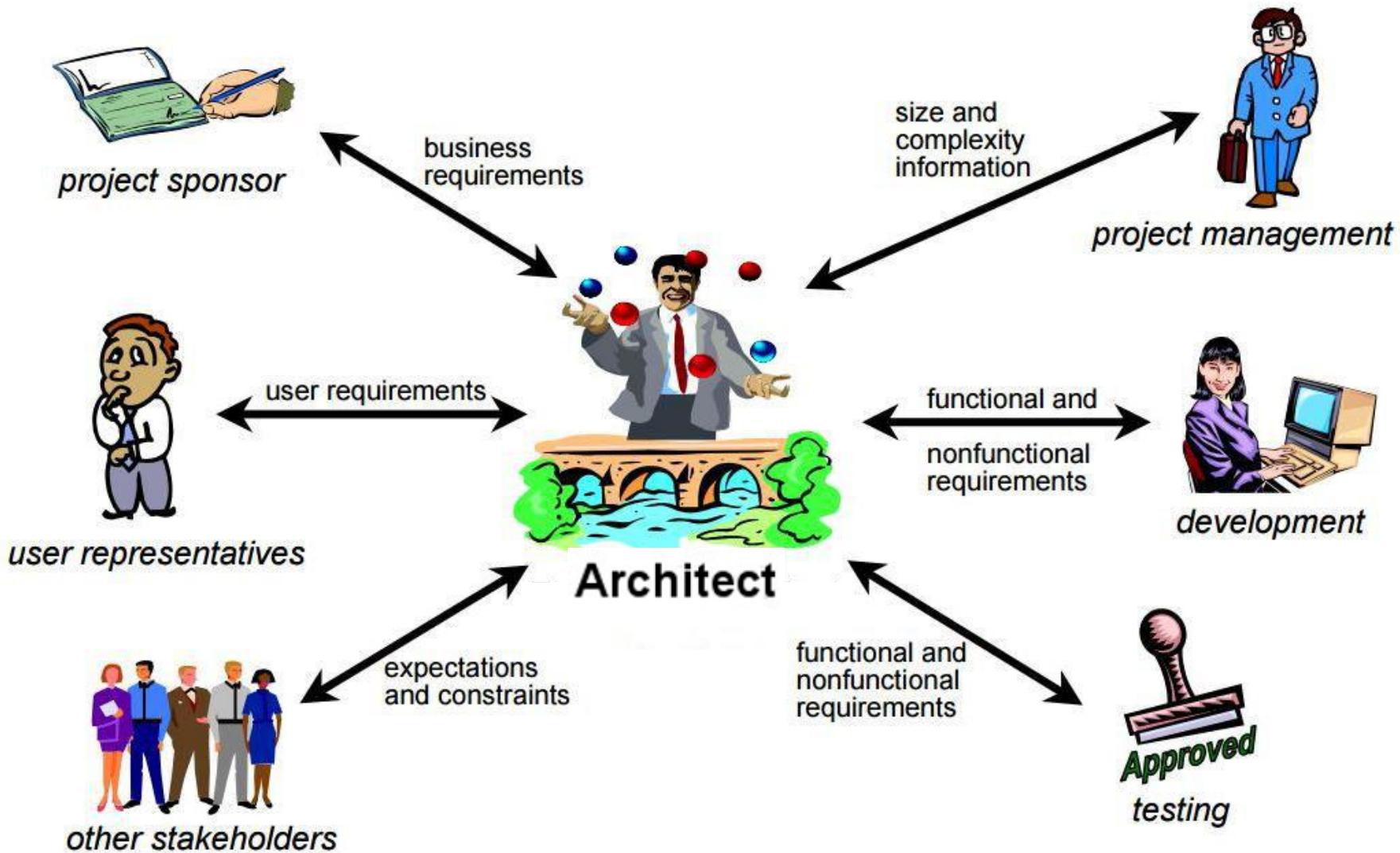
Zusätzliche Aufgaben und Verantwortung von Software Architekten

- Sie werden auch mit vielen Arbeiten abseits der direkten Architekturentwicklung befasst sein
 - Sie werden im Support Management ebenso eingebunden sein, wie in die direkten Kommunikation mit Kunden
- Architekten benötigen daher mehr als technische Fähigkeiten
 - Architekten müssen den Stakeholdern erklären können, warum sie diese oder jene Prioritäten gesetzt haben bzw. warum die eine oder andere Vorstellung des/der Stakeholder nicht oder nicht im gewünschten Umfang umgesetzt werden können
 - Architekten benötigen diplomatische, verhandlungstechnische und kommunikative Fähigkeiten
 - Sie müssen eine kompetente Führungskraft sein. Für das Entwicklerteam ebenso, wie in den Augen des Managements
- Architekten brauchen aktuelles Wissen
 - Sie müssen **technisches** Know-how haben (z.B. zu: Patterns, Datenbankplattformen oder Web-Services Standards)
 - Sie benötigen auch betriebswirtschaftliches Wissen

- Denkt analytisch, kann ein Problem zerteilen
- Kennt die eingesetzten Technologien, deren Stärken und Limitationen
- Hat selbst schon viel programmiert und besitzt dementsprechend viel Erfahrung
- Versteht den Anwender und entwirft die Architektur so, dass die entstehende Software für und nicht gegen ihn arbeitet
- Kann sich auf verschiedenen Abstraktionsebenen bewegen
- Passt die Architektur an die Aufgabenstellung an
- Kennt die gängigen Architekturstile und kann diese situationsgerecht anwenden
- Bezieht alle beteiligten Stakeholder in den Prozess mit ein

- Architektur des Systems definieren
- Integrität der Architektur sicherstellen
- Technische Risiken erkennen
- Risikomanagement-Strategien entwickeln
- An der Projektplanung teilnehmen
- Reihenfolge und Inhalt von Iterationen vorschlagen
- Consulting für Design-, Implementierungs- und Integrations-Teams
- Produktmarketing mit neuen Visionen unterstützen

Das Stakeholder – „Problem“



- Anforderungsmanagement
 - Stellt die Machbarkeit sicher
 - Klärt funktionale und nicht-funktionale Anforderungen
- Implementierungsteam
 - Berät das Team
 - Sorgt für die Kontrolle (Code Reviews)
- Projektleitung
 - Gibt Vorschläge/Vorgaben für Organisation(Iterationsplanung)
 - Sorgt für die Umsetzung mit angemessenen Kosten
- Qualitätssicherung
 - Klärt Kritikalität
 - Klärt Testreihenfolge
 - Informiert sich über Fehlermeldungen (Relevanz)

- IT-Betrieb
 - Kennt betriebliche Rahmenbedingungen
 - Infrastruktur
 - Konfigurationen
- Security
 - Liefert Sicherheitsvorgaben
 - Der Architekt berücksichtigt die Vorgaben in der Umsetzung
- Kunde
 - Der Architekt sorgt für die Umsetzbarkeit der Lösung
 - Der Architekt begleitet die Umsetzung
- Weitere Organisationseinheiten
 - Informiert
 - Motiviert

Geschäftsprozess-Architektur oder Business-Architektur

Unternehmens-IT-Architektur oder Enterprise-IT-Architektur

Softwarearchitektur

Infrastruktur-Architektur

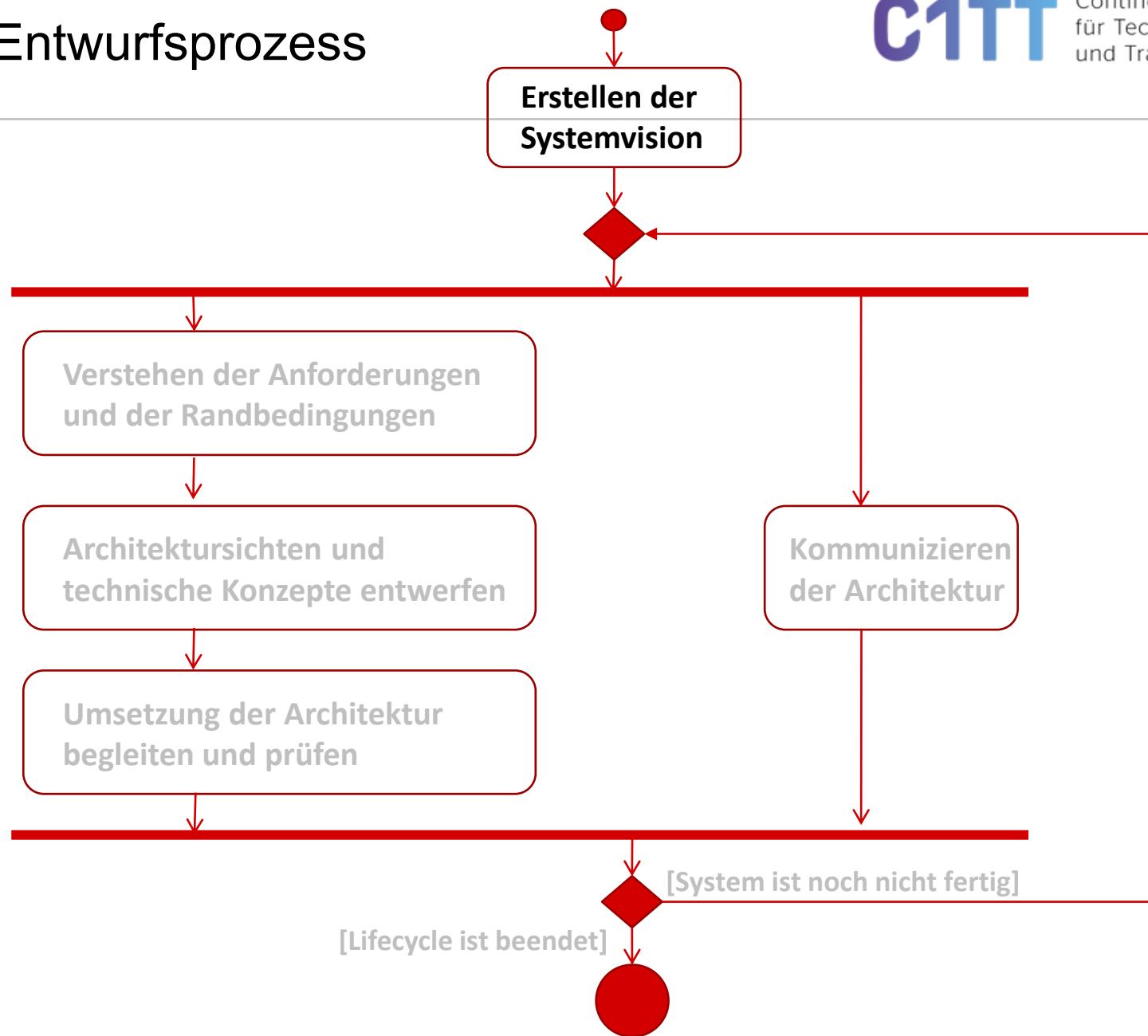
Hardware-Architektur

- [BWSW2014] M.Gharbi, A.Koschel, A.Rausch, G.Starke
Basiswissen für Softwarearchitekten (2. Auflage) ISBN: 978-3-86490-165-2
- [SWP2012] L.Bass, P.Clements, R.Kazman
Software Architecture in Practice (3rd Edition) ISBN: 978-0321815736
- [FRAU2012] R.Carbon
Architecture-Centric Software Producibility Analysis ISBN: 978-3-8396-0372-7
- [STARKE2014] G.Starke
Effektive Software-Architekturen (6.Auflage) ISBN: 978-3-446-43614-5
- [PPROG2011] M.Geirhos
Professionell entwickeln (1.Auflage) ISBN: 978-3-8362-1474-2
- [ISO42010] ISO/IEC/IEEE Systems and software engineering --
Architecture description ISBN: 978-0-7381-7142-5
- [BUND2015] V-Modell-XT
<http://www.cio.bund.de/Web/DE/Architekturen-und-Standards/V-Modell-XT/>

Entwicklung und Entwurf von Software Architekturen

- **Vorbereitung zur Architekturentwicklung**
- Der Entwurfsprozess und Vorgehensweise
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen

Der Entwurfsprozess



- für das Projekt benötigtes Domänenwissen und technisches Hintergrundwissen erarbeiten
- in der Organisation vorhandene Systeme ermitteln und auf Wiederverwendbarkeit prüfen
- von Dritten angebotene Systeme ermitteln, die eine ähnliche Aufgabe - oder wenigstens Teile davon - erfüllen wie das zu entwickelnde System
- passende technische Literatur für benötigte Lösungsansätze und Vorgehensmuster sichten

Hierzu sollten die Hauptaufgabe und die Verantwortlichkeit des Systems in wenigen Sätzen beschrieben werden mit Bezug auf die wichtigsten Begriffe und Aspekte der Fachdomäne.

- **Ein erster Rahmen für die Architektur**
- **System bereits in eine der Systemkategorien^{*)} zuordnbar**

^{*)} 2042_K02_Grundbegriffe_2

Eine Systemvision lotet zum einen die ***betriebswirtschaftliche Zweckmäßigkeit*** einer Initiative aus.

- Beispielsweise wird eine Systemvision zur Ablösung eines über Jahrzehnte gewachsenen Host-Systems durch ein Microservice-basiertes System klare Aussagen über die angestrebte und erwartete Rentabilität beinhalten.

Zum anderen definiert eine Systemvision jedoch auch die ***essenziellen Anforderungen*** an das zukünftige System.

In der Verantwortung eines Architekten liegt es, diese Anforderungen

- hinsichtlich ihrer architektonischen Machbarkeit kritisch zu hinterfragen,
- im gesamten bzw. globalen IT-Kontext einer Organisation zu sehen,
- auf sich widersprechende Anforderungen hinzuweisen und
- Alternativen aufzuzeigen.

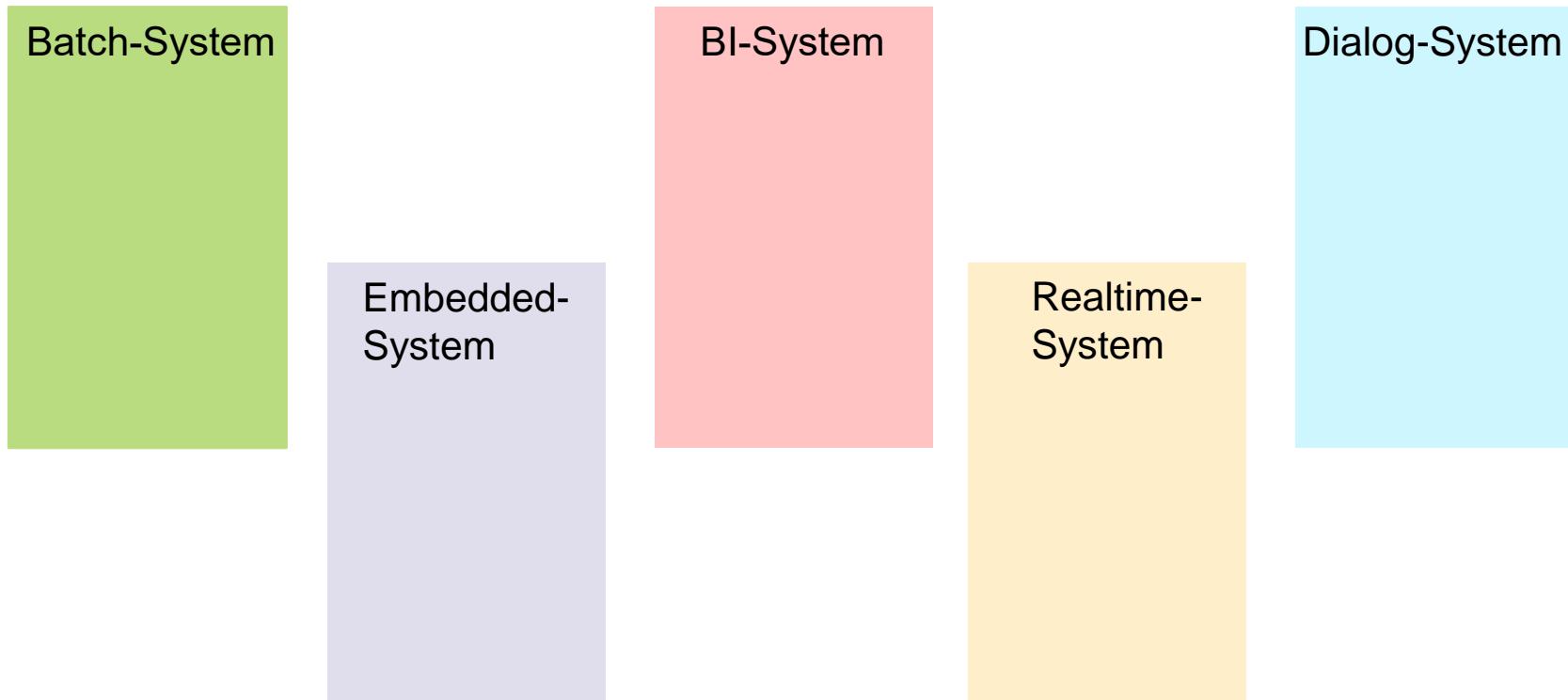
Daher ist es von eminenter Bedeutung, dass bereits zu diesem frühen Zeitpunkt ein Architekt an der Erstellung der Systemvision mitwirkt.

Sonst besteht die Gefahr, dass eine Systemvision aus architektonischer Sicht unrealistisch ist und die Realisierung eines Systems bereits unter schlechten Vorzeichen beginnt.

Der Architekt

- beschreibt die Kernaufgaben des Systems
- legt die Art der Steuerung fest
 - Z.B. Hierarchisch / Prozedural
 - Event-Driven
 - Parallel
 - Regelbasiert
- beschreibt die Nutzungsart des Systems

Dies soll in einer sehr frühen Phase bereits Ordnung in die Gedanken bringen, so dass auch andere Architekten sofort verstehen was für eine Art System hier erstellt werden soll.

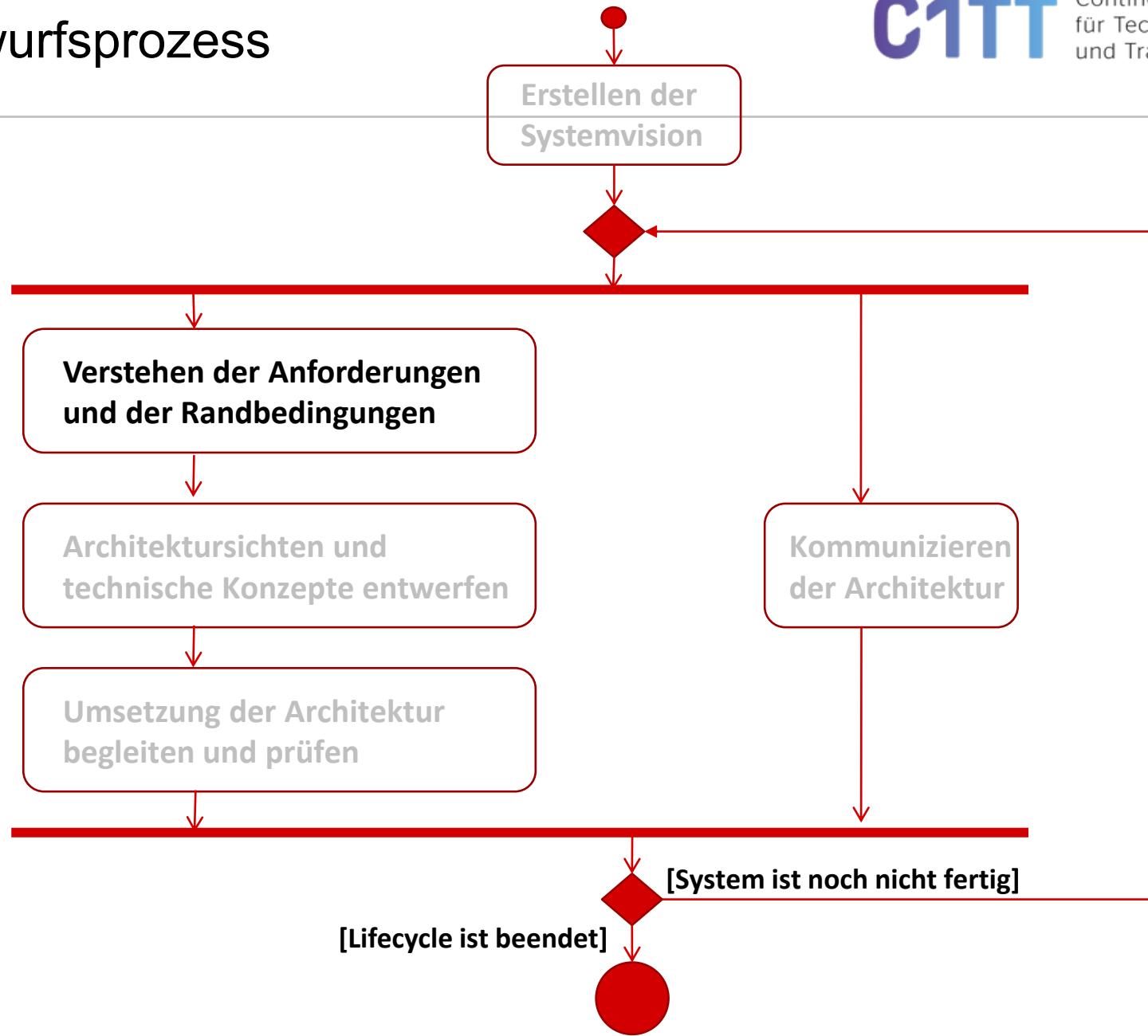


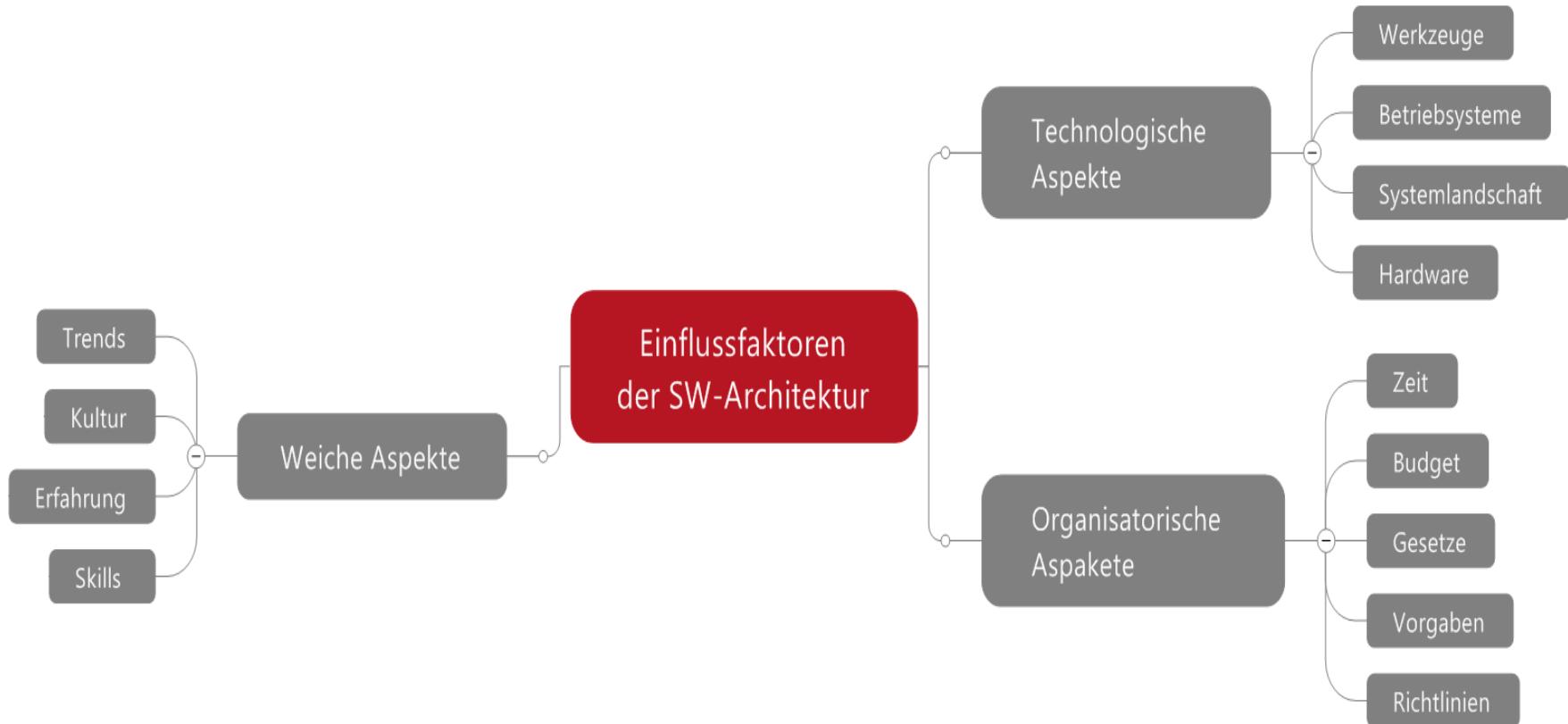
In der Praxis werden diese Arten vermischt, sind daher eher als Anhaltspunkte geeignet

- Informationssysteme / Online Systeme
 - Kernversicherungsverwaltungssystem einer Versicherung, SAP-Systeme, CAD-Systeme, komplexe Simulationssysteme für Wettervorhersagen
- Eingebettete Systeme
 - Werkzeugmaschinen oder Produktionslinien in der Fertigungsindustrie, Funkzellen von Handy-Netzen, Airbag-Steuerungen
- Mobile Systeme
 - Smartphones, (semi-)autonome Transportroboter
- Real Time Systeme
 - Die Anforderung an das Systeme, innerhalb vorgegebener Zeitgrenzen zu reagieren
Z.B.: Online Transaction Processing

- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen

Entwurfsprozess





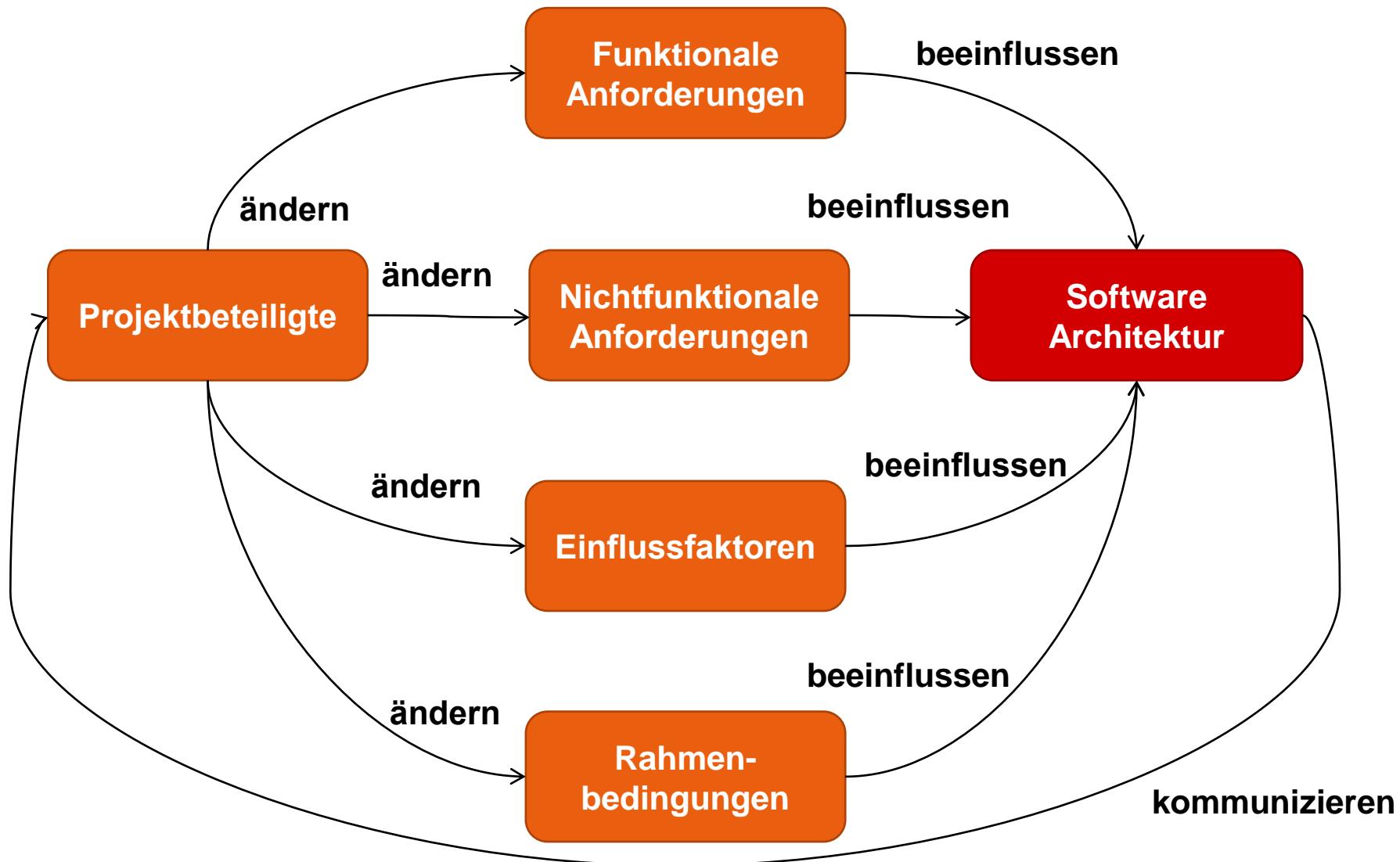
- Zu den organisatorischen Einflussfaktoren gehören alle Vorgaben, Richtlinien oder Gesetze,
 - die von außen (Staat, Institut, etc.) und
 - von innen (Lastenheft, Fachkonzept, Pflichtenheft, Projektleiter, etc.)
 - dem Projekt vorgeschrieben werden
- Diese Faktoren sind als statisch anzusehen und können in 90% der Fälle nicht geändert werden. Besonders staatliche Vorgaben und Gesetze sind an Gültigkeitstermine gekoppelt und daher „Pflichtprogramm“

- Zu wenig Zeit
 - Die Zeit bis zum Endtermin ist zu knapp bemessen, um die Anforderungen mit dem zugehörigen Projektteam zu erfüllen
- Zu wenig Budget
 - Es mangelt dem Projekt an Geld. Aus Projektsicht notwendige Investitionen in Hardware, Software oder Wissen (in Form von Schulungen oder weiteren Mitarbeitern) unterbleiben
- Zu wenig Wissen und Erfahrung
 - Es mangelt dem Projektteam an Wissen und Erfahrung in einigen Bereichen der Entwicklung. Beispielsweise wird ein neues, dem Team unbekanntes Entwicklungswerkzeug eingesetzt

- Die technischen Einflussfaktoren umfassen
 - bereits vorhandene Strukturen
 - eingesetzte Hardware
 - verwendetet Technologien
- Insbesondere in vorhandenen und produktiven Systemlandschaften spielen diese Faktoren eine große Rolle, da Schnittstellen verwendet oder Zugriffsrechte beachtet werden müssen.
- Bei Projekten, die auf der „grünen Wiese“ beginnen, sind die technischen Aspekte sekundär zu bewerten

- Subjektive, also nicht messbare Faktoren, zählen zu den „weichen“ Einflussfaktoren
- Trends wie:
 - Containerbasierte Entwicklung
 - Microservices
 - Cloud Computing
 - Middlewarenehmen starken Einfluss auf die neueren Softwarearchitekturen.
- Niemand würde heutzutage eine Fat Client-Anwendung mit Datenbankzugriff für mehr als 300 Personen entwickeln
 - Der Updateprozess und der Koordinationsaufwand wäre einfach zu groß
- Des weiteren gehören die Erfahrungen und der Skill der Mitarbeiter, sowie die vorherrschende Unternehmenskultur (Trendsetter oder safety first) mit dazu.
- Nicht vorhandene Skills können durch externe Berater ausgeglichen werden

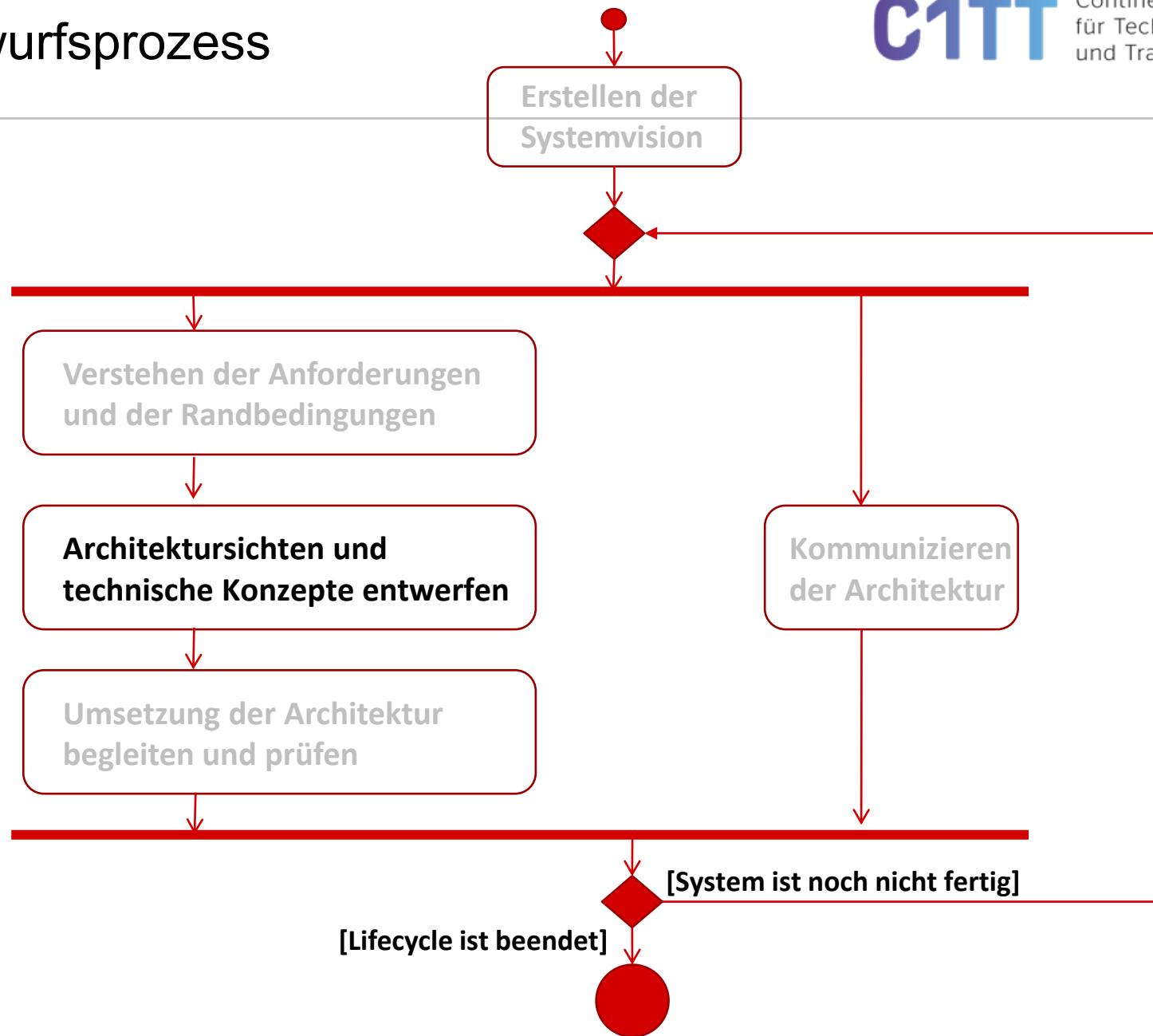
- Teilen sich auf in funktionale und nicht-funktionale Anforderungen
- **Funktionale Anforderungen**
 - Beschreiben gewünschte Funktionalitäten (was soll das System tun/können) eines Systems bzw. Produkts, dessen Daten oder Verhalten
- **Qualitative (Nicht-funktionale) Anforderungen**
 - Anforderungen an die Qualität der geforderten Funktionalitäten



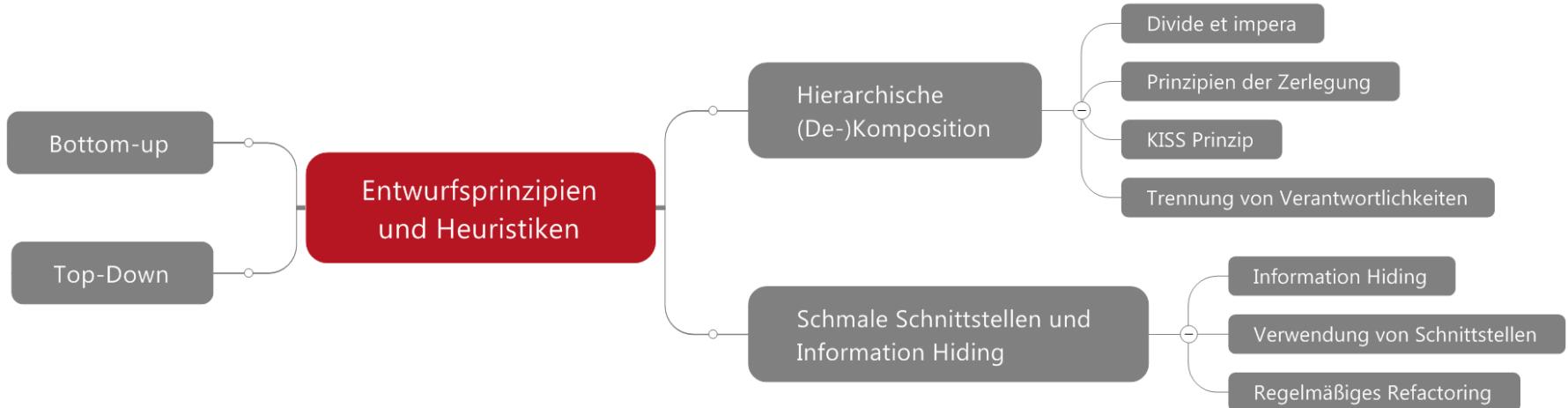
Funktionalität	Leistungseffizienz	Kompatibilität	Benutzerfreundlichkeit
<ul style="list-style-type: none">• Vollständigkeit• Korrektheit• Angemessenheit	<ul style="list-style-type: none">• Zeitverhalten• Ressourcenverbrauch• Kapazität	<ul style="list-style-type: none">• Ko-Existenz• Interoperabilität	<ul style="list-style-type: none">• Erlernbarkeit,• Bedienbarkeit,• Schutz vor Fehlern des Benutzers• Barrierefreiheit
Portierbarkeit	Wartbarkeit	Sicherheit	Zuverlässigkeit
<ul style="list-style-type: none">• Anpassbarkeit• Installierbarkeit• Austauschbarkeit	<ul style="list-style-type: none">• Modularität• Wiederverwendbarkeit• Analysierbarkeit• Änderbarkeit• Testbarkeit	<ul style="list-style-type: none">• Vertraulichkeit• Integrität• Nachweisbarkeit• Ordnungsmäßigkeit• Authentizität	<ul style="list-style-type: none">• Reife,• Verfügbarkeit• Fehlertoleranz• Wiederherstellbarkeit

- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - **Entwurfsprinzipien**
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen

Entwurfsprozess



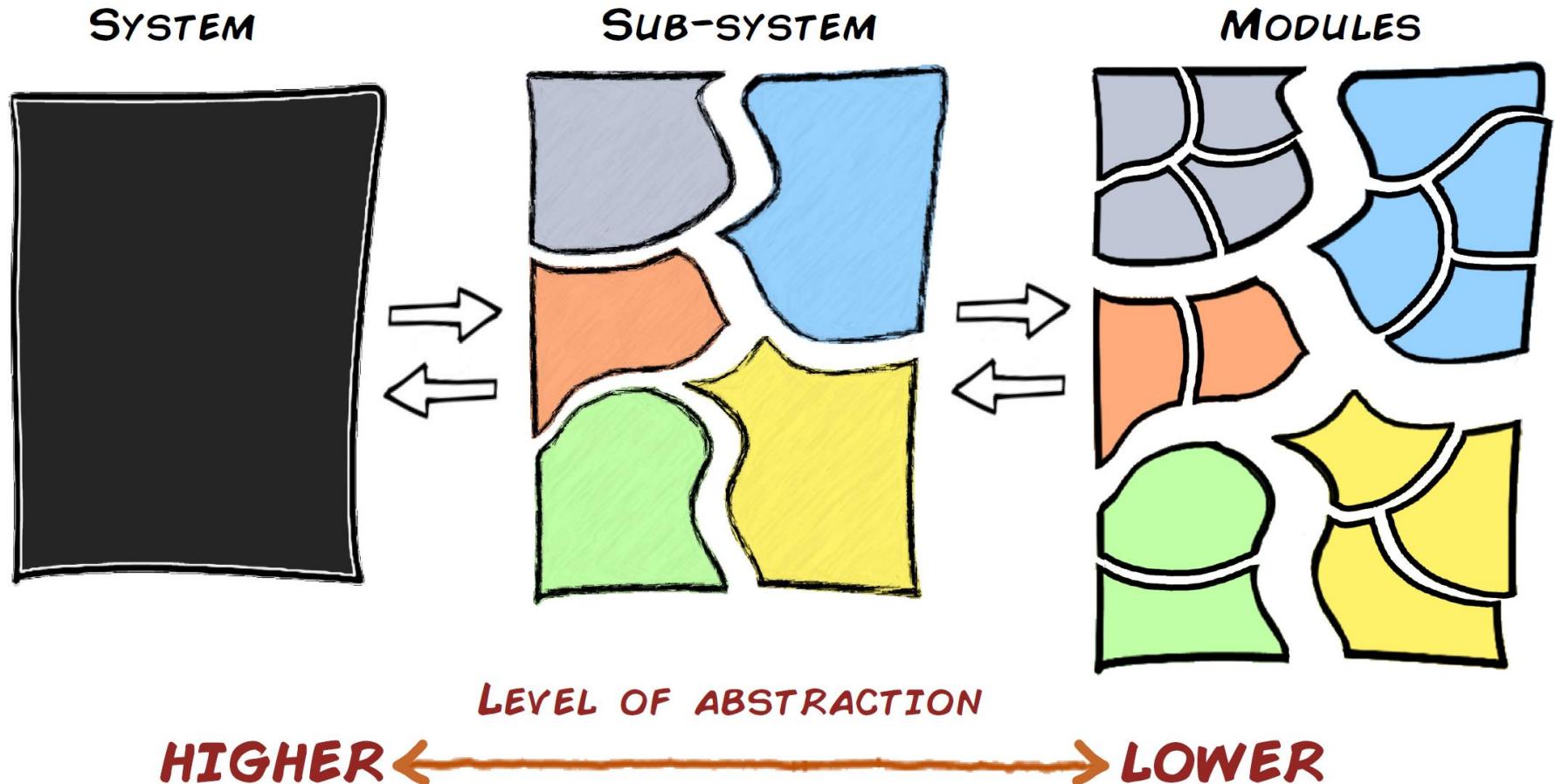
- Helfen dem Softwarearchitekt beim Entwurf
- Repräsentieren bewährte Grundsätze
- Behandeln meist folgende Hauptprobleme:
 - Reduktion der Komplexität
 - Erhöhung der Flexibilität
 - Änderbarkeit einer Softwarearchitektur



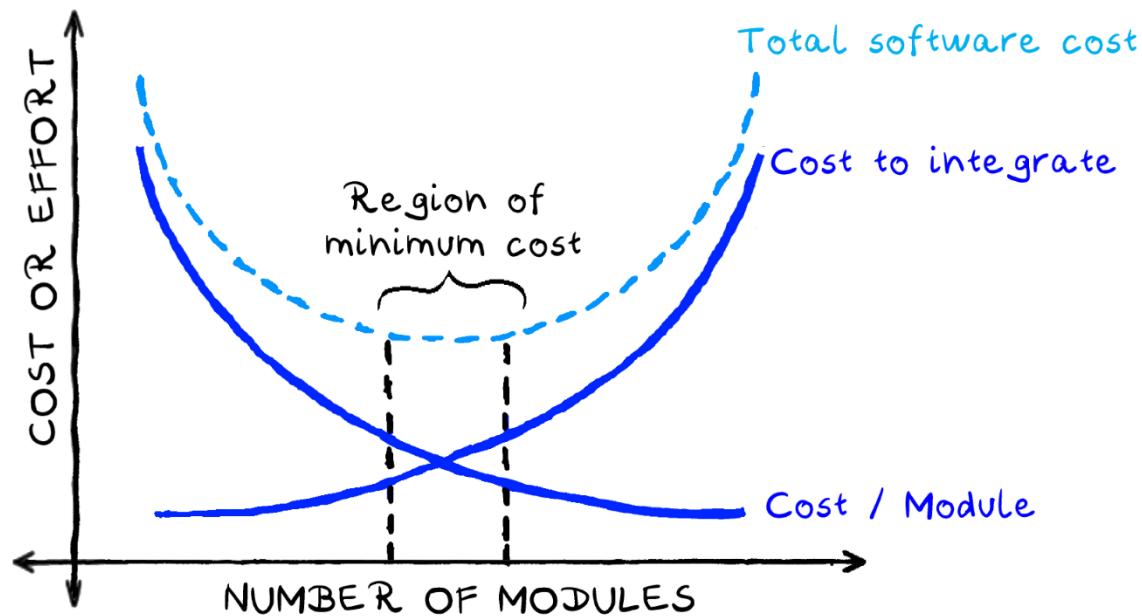
- Der Begriff „Trennung von Verantwortlichkeiten“ (engl. separation of concerns) wurde 1974 von E. W. Dijkstra in seiner Arbeit „On the role of scientific thought“ geprägt
- Die einzige Möglichkeit um die Komplexität in einem Projekt zu beherrschen ist, diese in Verantwortlichkeiten zu trennen
- Verantwortlichkeiten
 - Zeit
 - Qualität
 - Sichten
 - Teile
 - Wissen



Edsger Wybe
Dijkstra

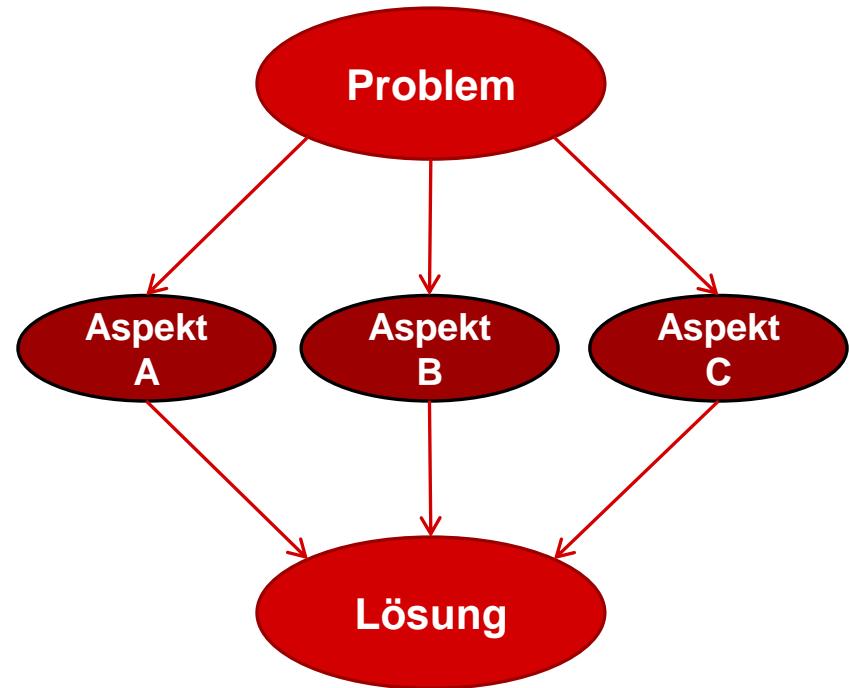


- Modularität ist die **Anwendung von separation of concerns**
- Behandle ein Modul zu einem Zeitpunkt, ignoriere die Details der anderen Modulen
- Aber Vorsicht !!!

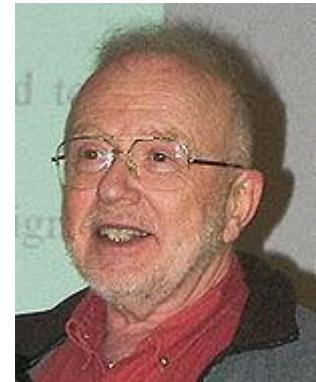


- Bekannt auch als „teile und herrsche“ (divide and conquer)
- The “divide your enemy so you can reign” approach is attributed to Julius Cesar—he successfully applied it to conquer Gaul twenty-two centuries ago (no typo).

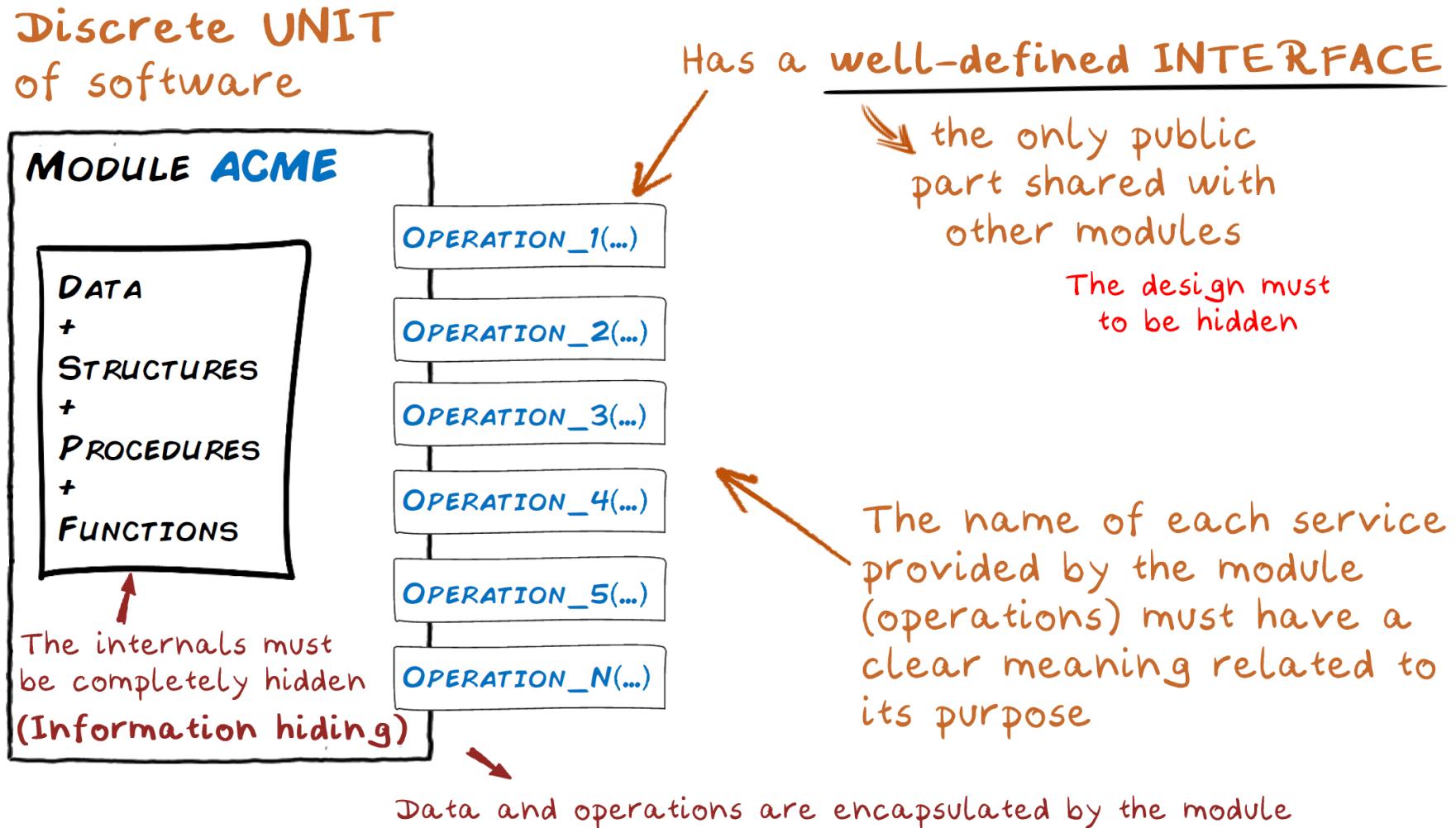
*„Identifying different aspects of a problem we can concentrate on each one individually“
Edsger Wybe Dijkstra 1974*



- Der Begriff „information hiding“ wurde 1972 von D. L. Parnas in seiner Arbeit „On the Criteria to be Used in Decomposing Software into Modules“ beschrieben
- Die einzige Möglichkeit um die Komplexität in einem Projekt zu beherrschen ist, diese in Verantwortlichkeiten zu trennen



David Lorge
Parnas



Schnittstelle



Implementierung

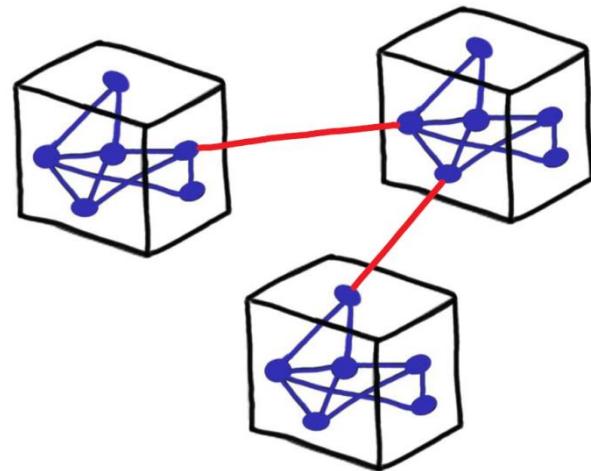
- Kapselung und Information Hiding
- Client Komponenten hängen nur von Schnittstellen ab

- Das verbessert:
 - Weiterentwicklung: Kleine Änderungen haben lokale Auswirkungen
 - Wiederverwendbarkeit und Änderbarkeit
 - Schutz: Fehlerisolierung

Meyer, 1988 - 1997

- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - **Abhängigkeiten und Kopplung von Bausteinen**
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen

- Kopplung ist das externe Ma von Modulabhangigkeit
- Ziel 1: Jedes Modul kommuniziert mit so wenig Modulen, wie moglich
- Ziel 2: Loose Kopplung
 - Zwei Komponenten arbeiten zusammen, wissen aber sehr wenig von einander
- Kopplung zu reduzieren bewirkt:
 - Verstandlichkeit
 - Testbarkeit
 - Anderbarkeit
 - Wiederverwendbarkeit



- **Aufruf**

Eine Kopplung liegt vor, wenn eine Klasse eine andere Klasse direkt benutzt, indem sie eine Methode der Klasse aufruft

- **Erzeugung**

Eine andere Art der Kopplung besteht, wenn ein Baustein einen anderen Baustein erzeugt

- **Daten**

Eine weniger starke Kopplung liegt vor, wenn die Klassen über eine globale Datenstruktur oder nur über Methodenparameter kommunizieren

- **Ausführungsart**

Eine Kopplung über Hardware besteht, wenn Bausteine in der gleichen Laufzeitumgebung oder der gleichen virtuellen Maschine ablaufen müssen

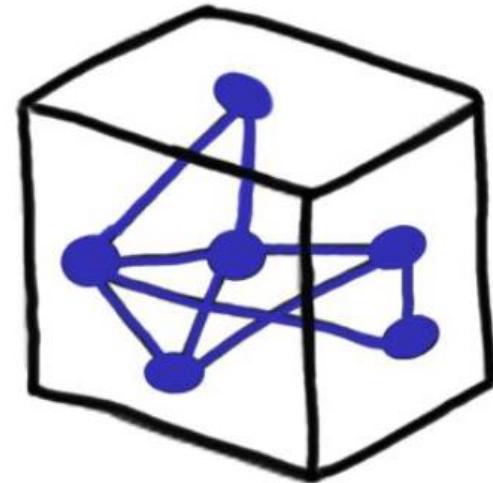
- **Zeit**

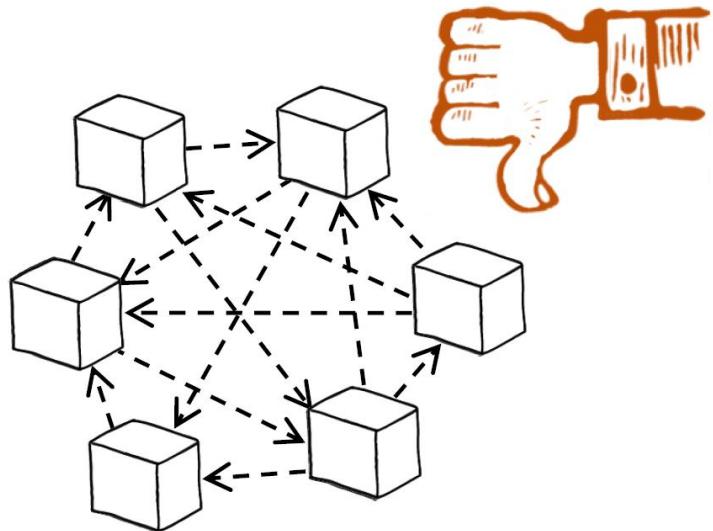
Wenn die zeitliche Abfolge von Bausteinaktivitäten eine Rolle spielt, liegt eine Kopplung über die Zeit vor

- **Vererbung**

In der Objektorientierung ist eine Unterklasse bereits durch das Erben von Eigenschaften mit ihrer Oberklasse gekoppelt

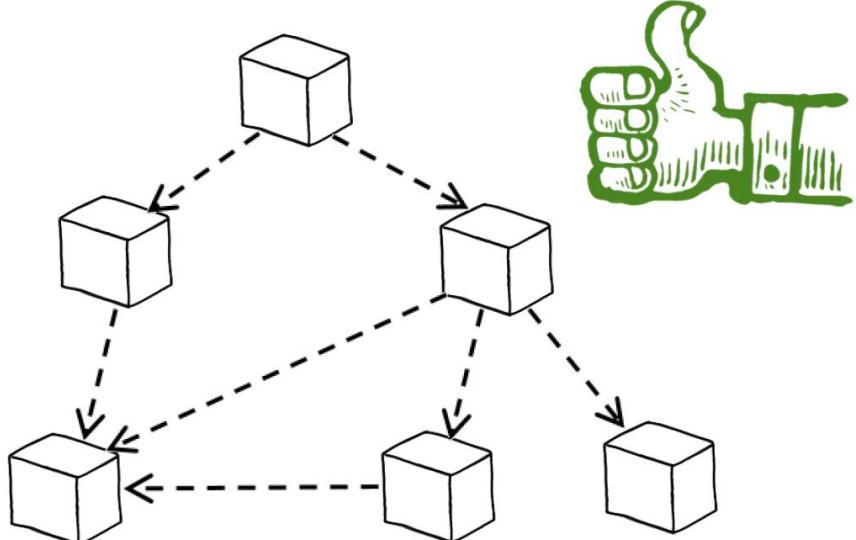
- Alle Elemente eines Moduls müssen zusammen gehören
- Alle Elemente teilen den selben Zweck
- Das erhöht die Verständlichkeit eines Modules
- Ziel: Hohe Kohäsion



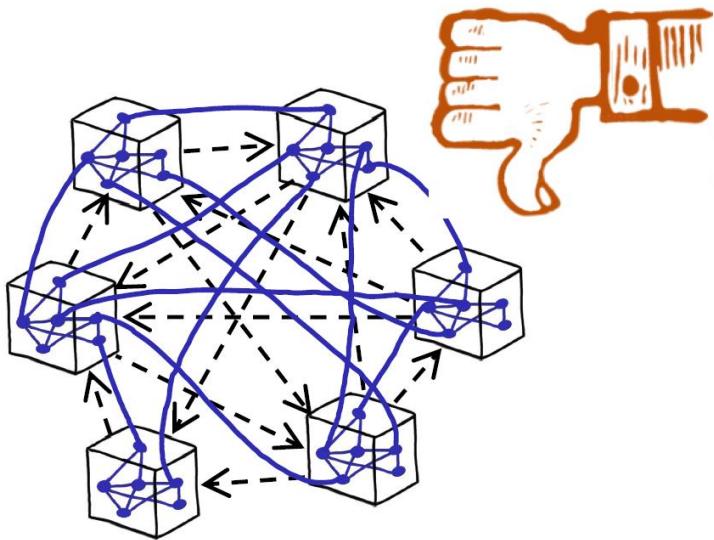


**NON-HIERARCHICAL
ORGANIZATION**
HIGH COUPLING

DEPENDENCY ----->
(USES, CALL)

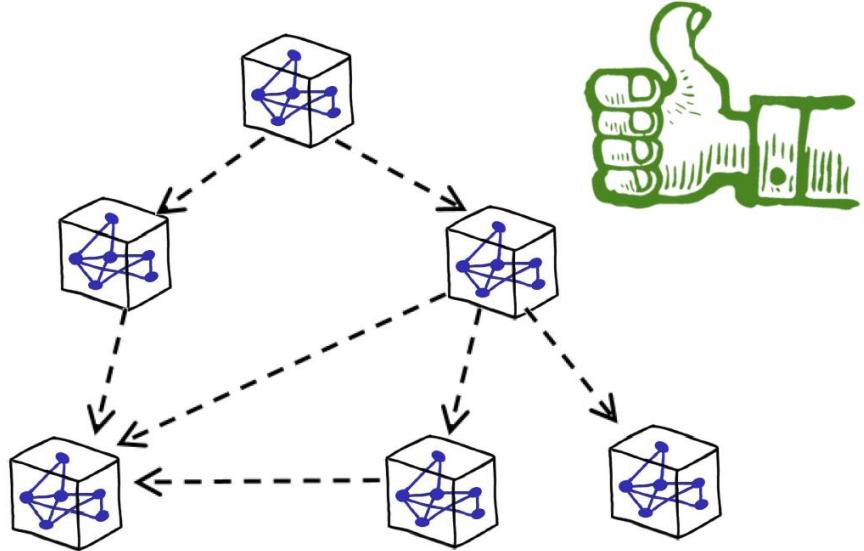


**HIERARCHICAL
ORGANIZATION**
LOW COUPLING



**NON-HIERARCHICAL
ORGANIZATION**
HIGH COUPLING

DEPENDENCY ----->
(USES, CALL)

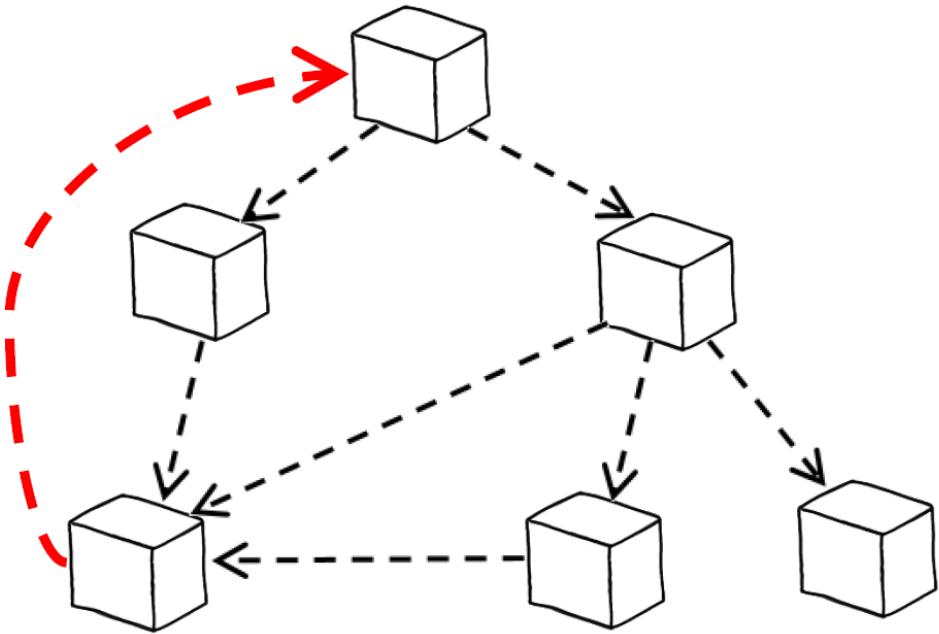


**HIERARCHICAL
ORGANIZATION**
LOW COUPLING

GOT IT?

Vermeiden Sie das:

Avoid
this, OK?



DEPENDENCY ----->
(USES, CALL)

Wissen Sie warum?

- Single responsibility principle (SRP)
- Open closed principle (OCP)
- Liskov substitution principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle (DIP)



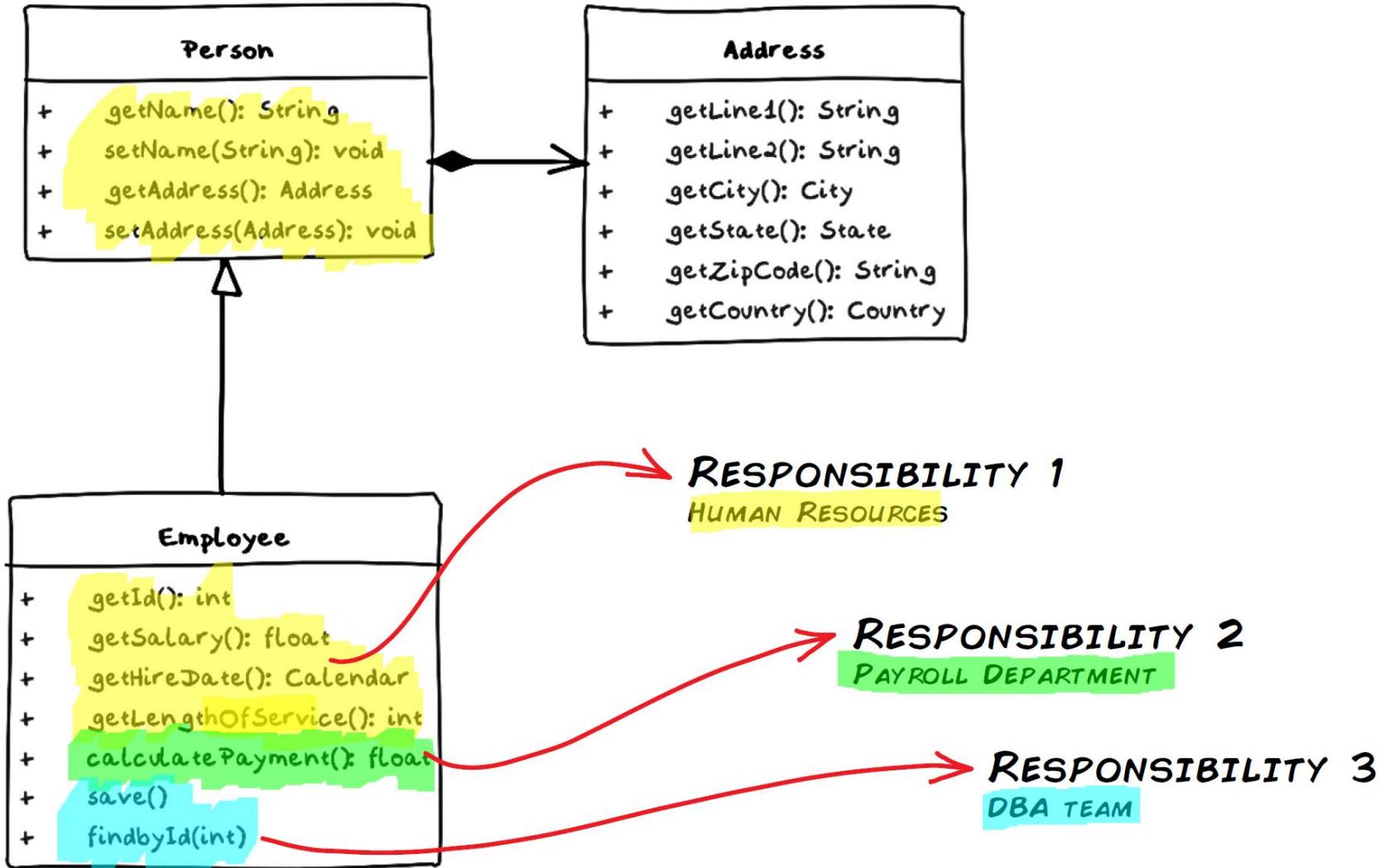
Robert C. Martin (Uncle Bob)

- **Eine Klasse darf sich nur aus einem Grund ändern**

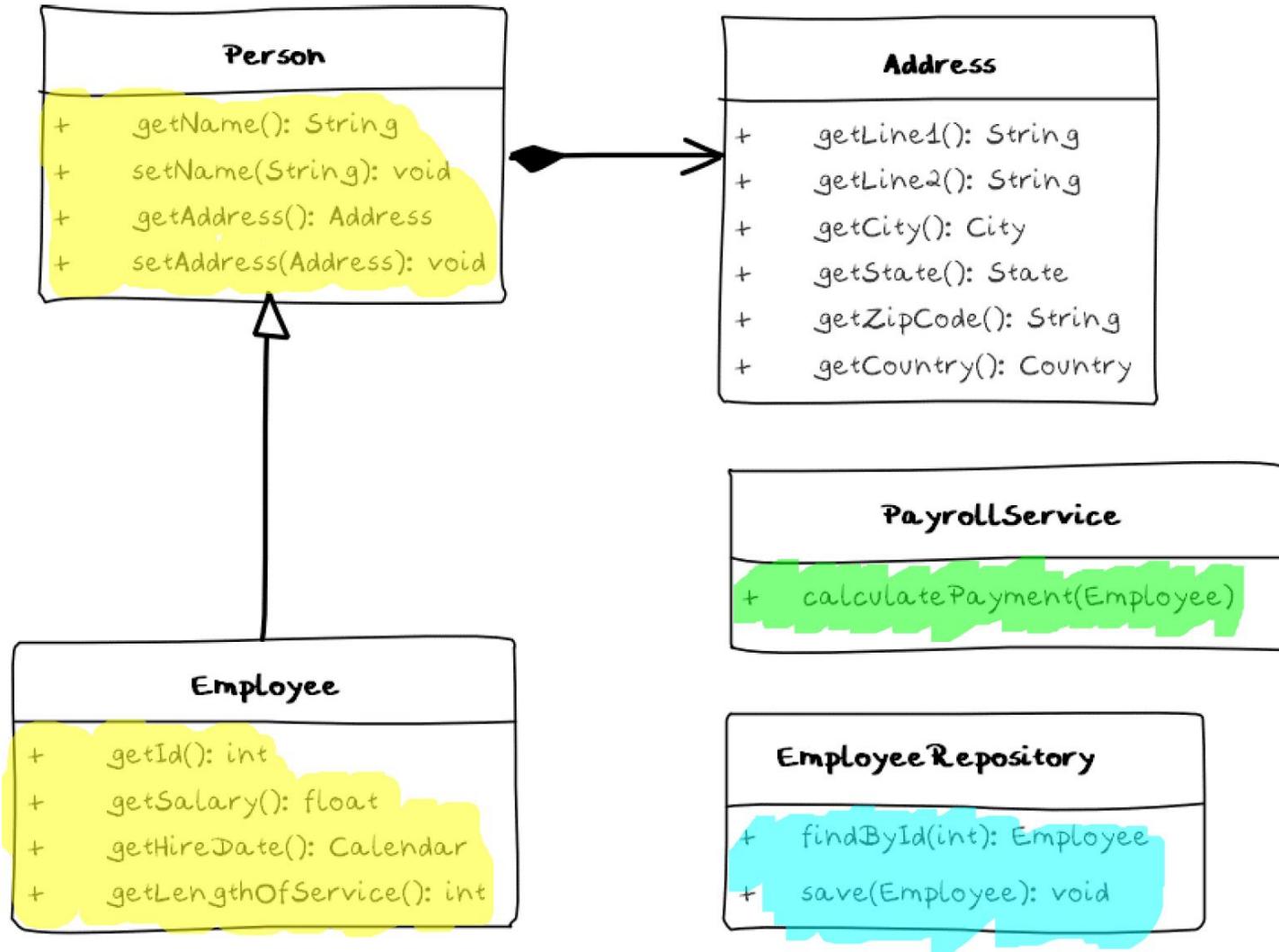
- **Zuständigkeit**
 - Grund der Änderung
 - Einzige Teil der Funktionalität
 - Kapselung -> hohe Kohäsion
 - Voraussehen wie sich die Klasse entwickelt



Das Single-Responsibility Prinzip



Das Single-Responsibility Prinzip



Offen Geschlossen Prinzip Open-Closed Principle

- Module müssen offen und geschlossen sein
 - Offen für Erweiterungen
 - Geschlossen für Änderungen
- Neues Verhalten zum Erfüllen einer Anforderung
 - Offen für Erweiterung
- Jedoch ohne Änderung des Sources der vorhandenen Implementierung
 - Geschlossen für Änderungen

„Sei $q(x)$ eine Eigenschaft des Objektes x vom Typ T , dann sollte $q(y)$ für alle Objekte y des Typs S gelten, wo S ein Subtyp von T ist.“

1993 Barbara Liskov und Jeannette Wing

oder ...



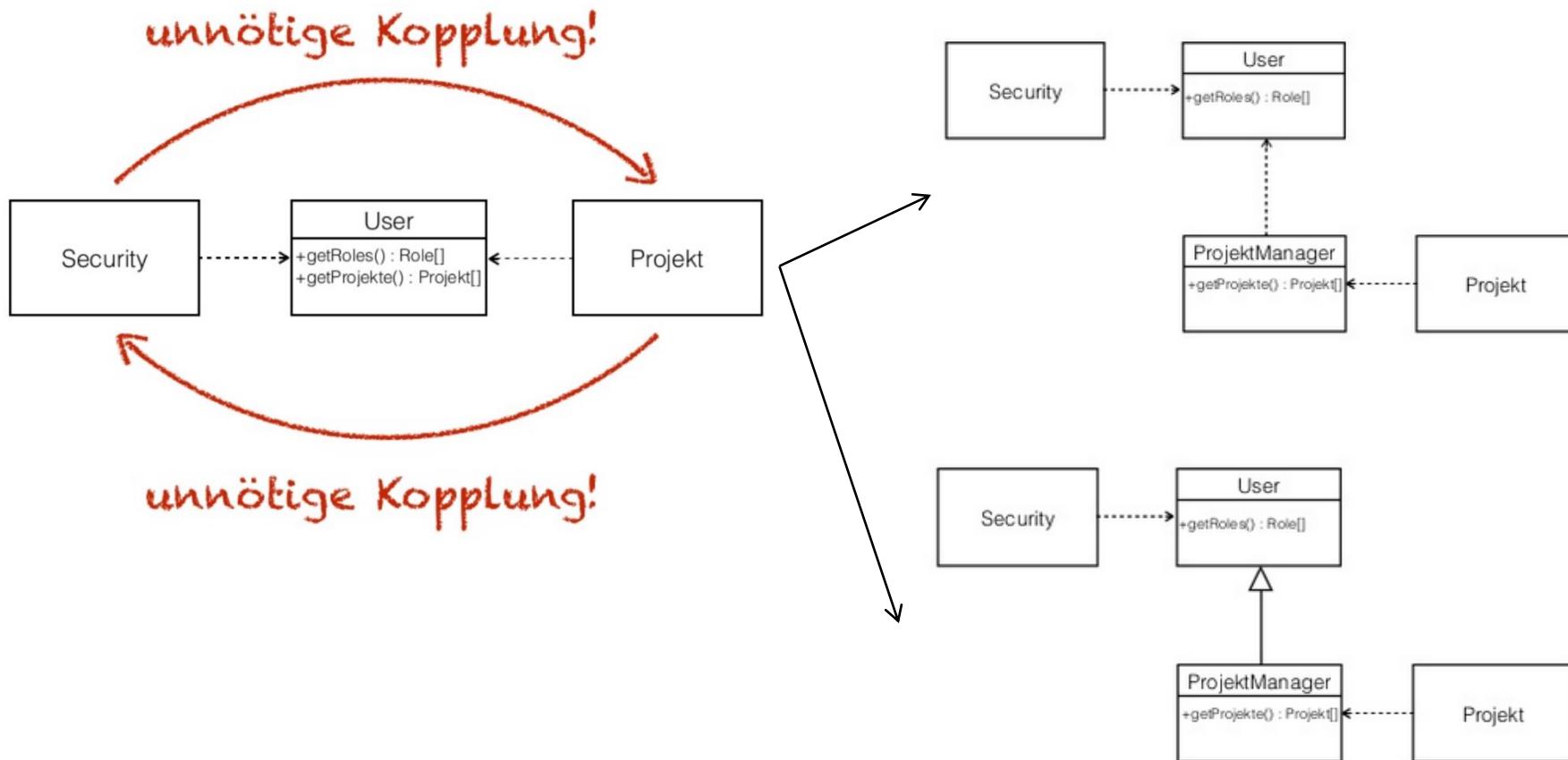
Barbara Liskov

Klassen sollen in jedem Fall durch ihre Unterklassen ersetztbar sein und sich auch gleich verhalten. (Starke 2011)

Häufig werden mathematische Beispiele wie Ellipse – Kreis oder Rechteck – Quadrat zur Demonstration verwendet.

Interface Segregation Principle

„Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden.“

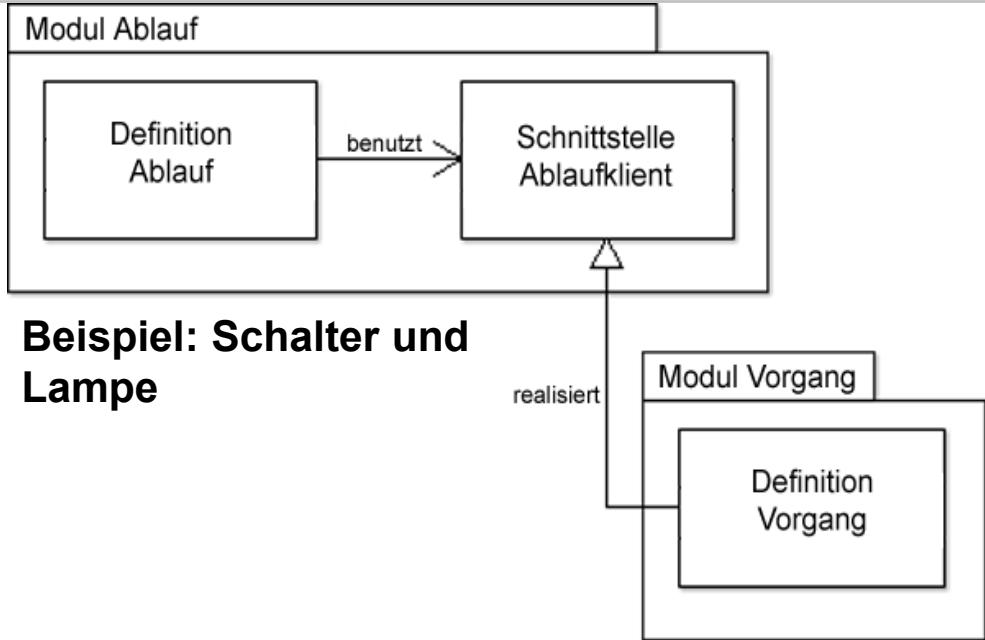


- Bei vielfacher Nutzung einer umfangreichen Schnittstelle ist es sinnvoll, diese in mehrere spezifische Schnittstellen zu zerlegen, z.B.:
 - Zerlegung nach semantischem Zusammenhang
 - Zerlegung nach Verantwortungsbereich
- Eine solche Zerlegung reduziert die Anzahl der abhängigen Benutzer und damit auch mögliche Folgeänderungen.
- Außerdem sind kleine, fokussierte Schnittstellen leichter implementierbar und wartbar.

Dependency Inversion Principle

Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen

Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen



- Beim Dependency Inversion Principle geht es um die Umkehrung von einer direkten Implementierung zu einer Abstraktion.
- Beispielweise eine Desktopanwendung nutzt direkt die Windows-Komponenten.
- Würde die Desktopanwendung eine “Abstraktion” nutzen, so könnte man die Desktopanwendung z.B. auch auf einem MacOS laufen lassen.

- Läuft auch unter dem “Hollywood”-Prinzip und ist ein Prinzip bei “Frameworks”.
 - „Don’t call us, we call you.“
- Über definierte Schnittstellen (Callbacks) vom Framework, implementieren Clients die Spezialisierung, die dann vom Framework genutzt werden.
- DI ist eine spezielle Form vom IoC.
 - Beispiel vom Spring-Framework: ein IoC-Container, welcher DI implementiert
 - Bei DI geht es um das Managen der Abhängigkeiten: statt eines direkten Managements der Abhängigkeiten durch den Client, werden die Abhängigkeiten extern “injiziert”.
 - Extern ist dem Sinne der IoC-Container oder auch Assembler genannt.

- Bei DI geht es um das **wie** (wie komme ich an die Abhängigkeit)
- bei IOC um das **wer** (wer ruft wen auf)
- Daher ist auch DI eine spezielle Variante von IOC
- Bei DIP geht es um die Klärung, **an wen** richtet sich der Aufruf

- Zusammen spielen alle 3 ihre Mächtigkeit aus. DIP kann auch für sich allein stehen und ist ein genereller Ansatz als IOC und damit DI
- IOC/DI können demnach auch ohne DIP eingesetzt werden

- Verwirrung total?

- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - **Wichtige Architekturmuster und Architekturstile**
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen

Architekturmuster / Architekturstile

- Beschreiben die globale Strukturierung eines Softwaresystems
- Unterteilen das System in Subsysteme mit deren Verantwortlichkeiten
- Helfen, die Vor- und Nachteile von Architekturen zu erkennen und abzuwägen
- Ziehen sich durch die ganze Anwendung

Entwurfsmuster / Designpattern

- Grundlegende sind von der GoF definiert worden
- Unterteilt in
 - Strukturmuster
 - Verhaltensmuster
 - Erzeugungsmuster
- Beschreiben eher die Lösung eines lokalen Problems
- Werden meist durch Zufall gefunden

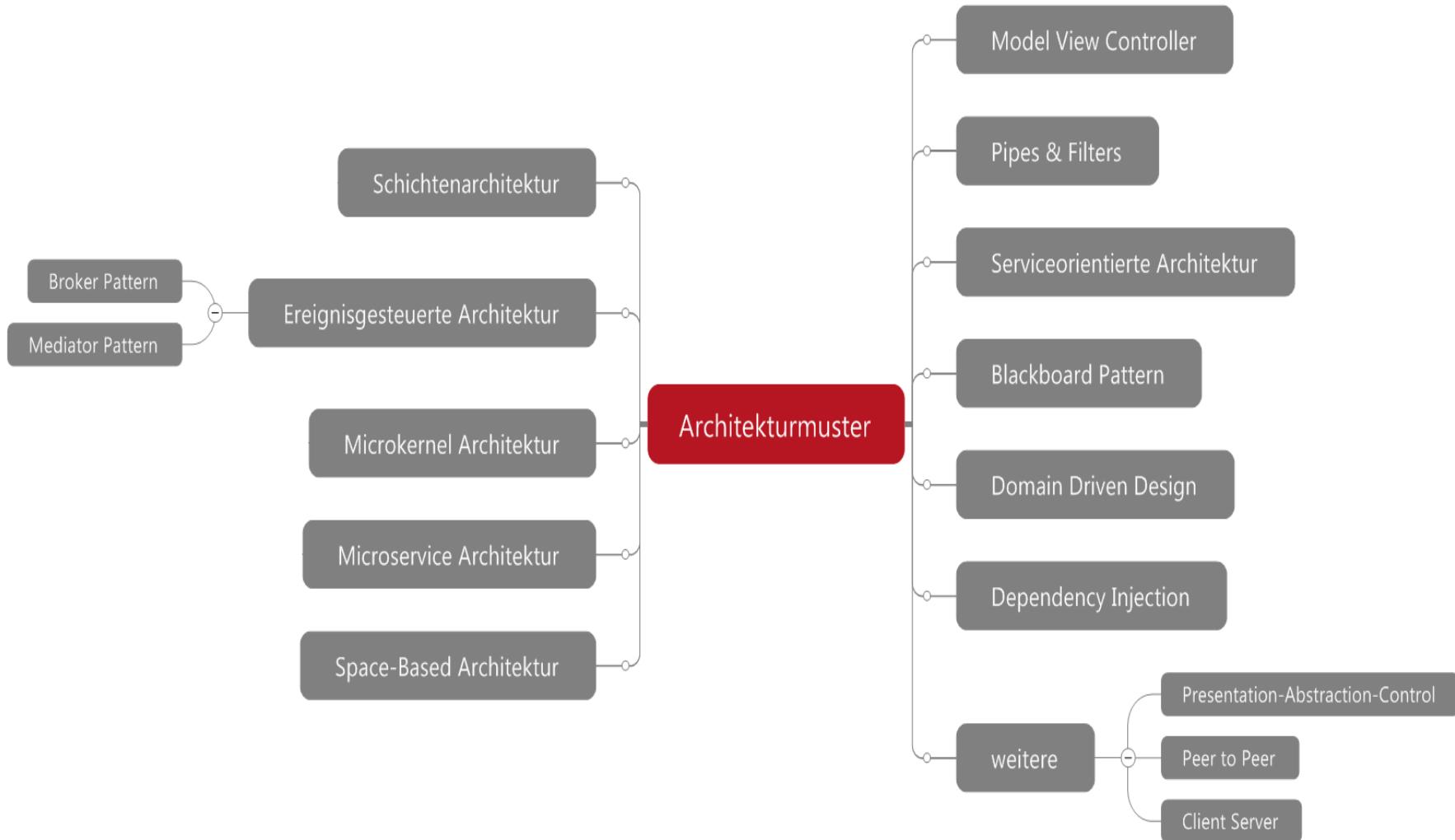
Architekturmuster

grob

Entwurfsmuster

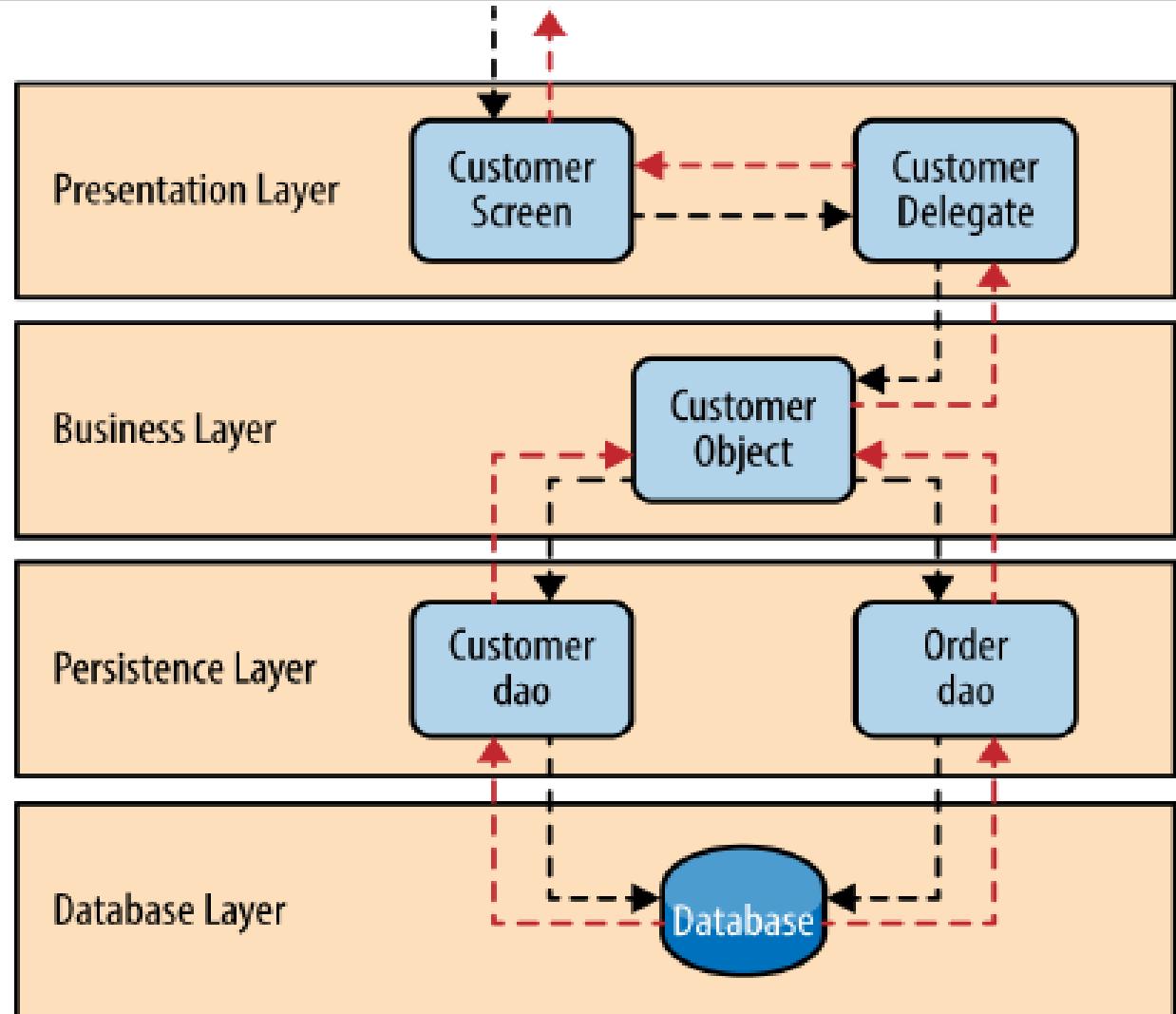
fein

Granularität



- **Warum:**
 - In allen komplexen Systemen ergibt sich die Notwendigkeit Teile des Systems unabhängig vom System zu entwickeln.
 - Aus diesem Grund benötigen die Entwickler des Systems eine klare und gut dokumentierte Gliederung der Vorgaben, so dass Module des Systems unabhängig entwickelt und gepflegt werden können
- **Herausforderung:**
 - Die Software muss derart segmentiert werden, dass das Modul mit wenig Interaktion zwischen den Teilen entwickelt werden kann, dabei aber Portabilität, Modifizierbarkeit und Wiederverwendbarkeit erhalten bleibt
- **Lösung:**
 - Um diese Trennung der Vorgaben zu erreichen, teilt die Schichtenarchitektur die Software in Einheiten, die „Schichten“ genannt werden.
 - Jede Schicht ist eine Gruppierung von Modulen, die eine zusammenhängende Reihe von Services bieten.
 - Der Einsatz muss unidirektional sein.
 - Layer teilen vollständig ein Set der Software, und jeder Teil besitzt eine öffentliche Schnittstelle.

- Das Layer pattern definiert Gruppe/n von Modulen die einer 'allowed-to-use' Einschränkung unterliegen
- Das Design soll definieren, welche Layer-Benutzungsregeln bzw. welche erlaubten Ausnahmen es gibt



Vorteile:

- Leicht zu verstehendes und zu entwickelndes Pattern
- Gute Testbarkeit von Komponenten aufgrund der Schichtenzuordnung
- Schichten können unabhängig voneinander entwickelt werden

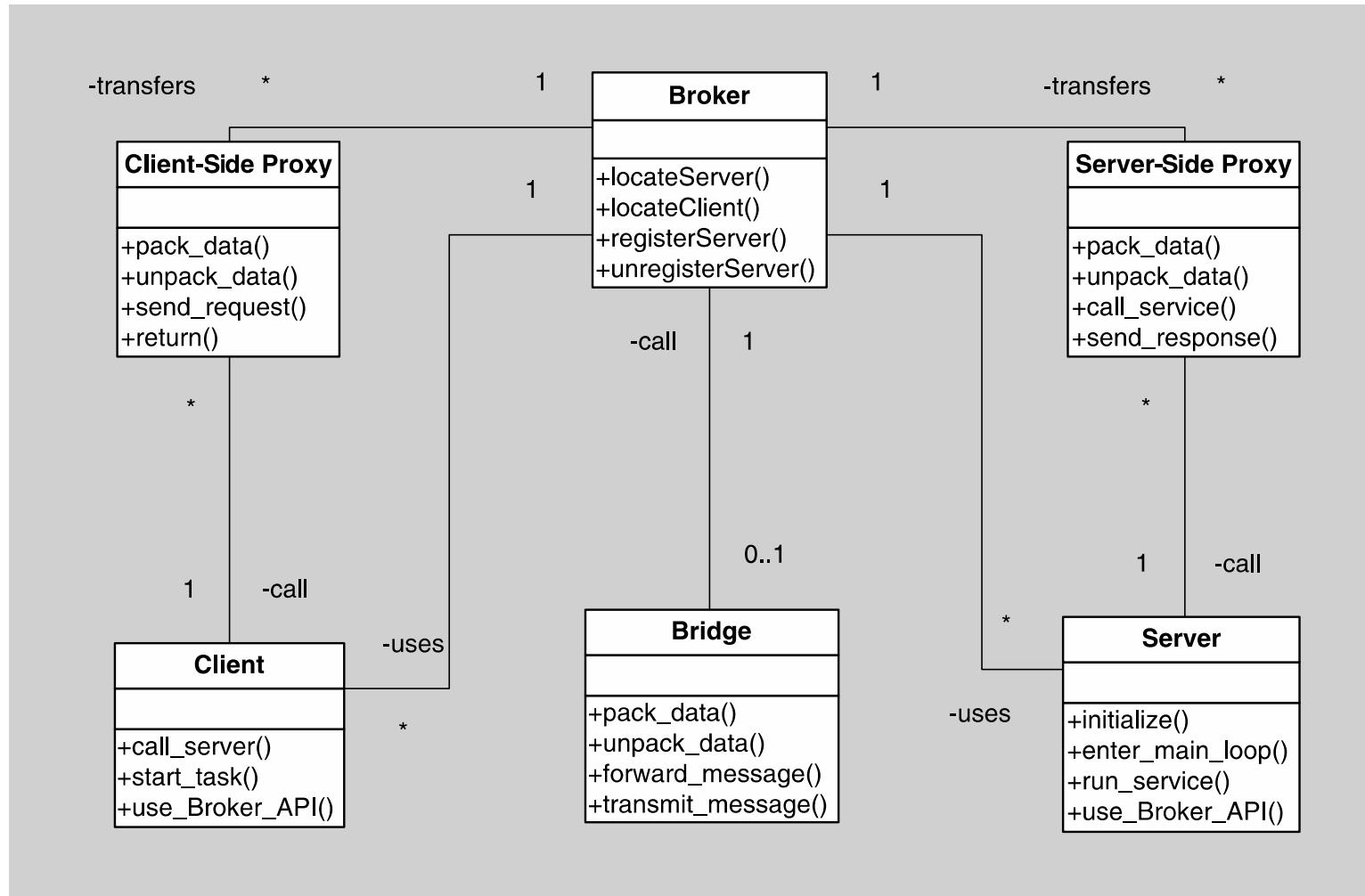
Nachteile:

- Manche Änderungen haben große Auswirkungen (Entitäten)
- Größere Applikationen werden leicht zu Monolithen
- In der Regel schlechtere Performance aufgrund der zusätzlichen Schnittstellen zwischen den Schichten
- Skalierung auf Softwareebene schwer möglich

- Jeder Softwareteil ist exakt einer Schicht zugeordnet
- Es gibt mindesten zwei Schichten (normalerweise sind es drei oder mehrere)
- Die ‘allowed-to-use’ Relationen dürfen nicht zirkular sein (z.B., eine niedrigere Schicht darf höhere Schichten nicht benutzen)
- Das überspringen einer Schicht ist nur in Ausnamefällen erlaubt. (allowed-to-use)
- Layer (logische Schicht) sind KEINE Tiers (physikalische Schicht)

- **Warum:**
 - Mehrere Systeme bestehen aus einer Sammlung von Diensten und sind über mehrere Server verteilt.
 - Die Umsetzung dieser Systeme ist komplex, weil man beachten muss, wie die Systeme interagieren und wie sie miteinander in Verbindung treten.
- **Herausforderung:**
 - Wie strukturieren wir eine verteilte Software, so dass
 - der Servicebenutzer den Aufbau und den Ort des Serviceproviders nicht kennen muss, und
 - Bindungen zwischen Benutzern und Providern einfach und dynamisch geändert werden können?
- **Lösung:**
 - Das Brokerpattern separiert Benutzer des Services (Clients) von Serviceanbietern (Servers) durch Einfügen eines Vermittlers (Broker).
 - Wenn ein Client ein Service benötigt, fragt er via Serviceinterface beim Broker an.
 - Der Broker leitet die Serviceanfrage des Clients weiter an einen Server (den der Client nicht kennen muss)
 - Der Server bearbeitet dann die Anfrage.

Broker Pattern Beispiel

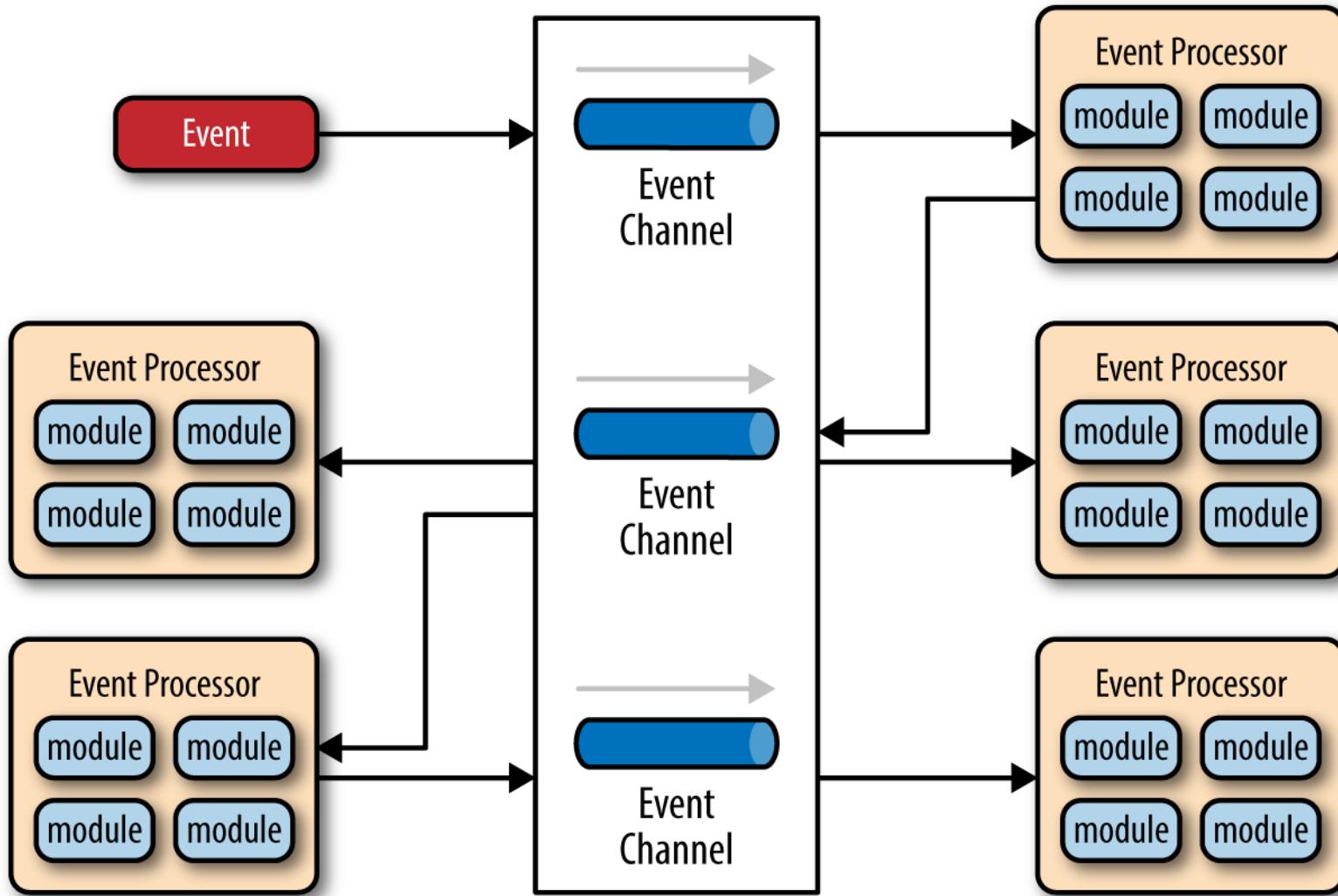


Vorteile:

- Standortunabhängigkeit der Client und Server
- Unabhängigkeit des Clients gegenüber Implementierungsänderungen des Servers
- leichte Portierbarkeit auf neue Systeme, da Client und Server frei von systemspezifischer Funktionalität sind
- neue Services können leicht implementiert werden, da sie Basisservices des Broker Systems nutzen können

Nachteile:

- Schlechte Testbarkeit durch Vielzahl von Events
- Relativ schwer zu entwickeln wegen der Asynchronität
- Der Broker kann ein 'Single Point of Failure' sein
- Niedrige Fehlertoleranz
- Eingeschränkte Effizienz, hohe Netzwerklast.



Mediator Topologie

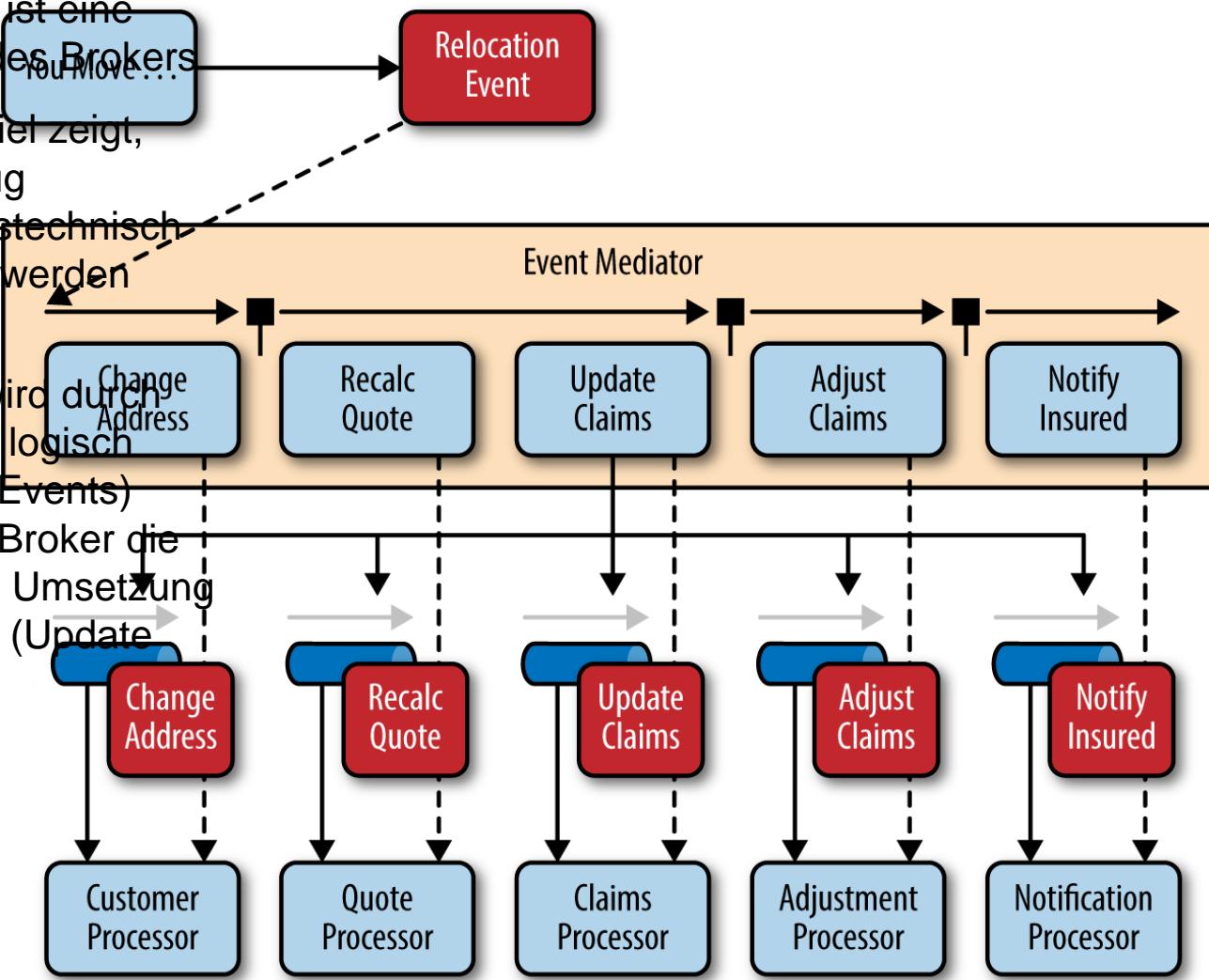
Der Mediator ist eine
Sonderform des Brokers

HIER VON...

Dieses Beispiel zeigt,
wie ein Umzug
versicherungstechnisch
durchgeführt werden
kann.

Der Umzug wird durch
den Mediator logisch
abgewickelt (Events)

während der Broker die
physische Umsetzung
gewährleistet (Update
Claims)

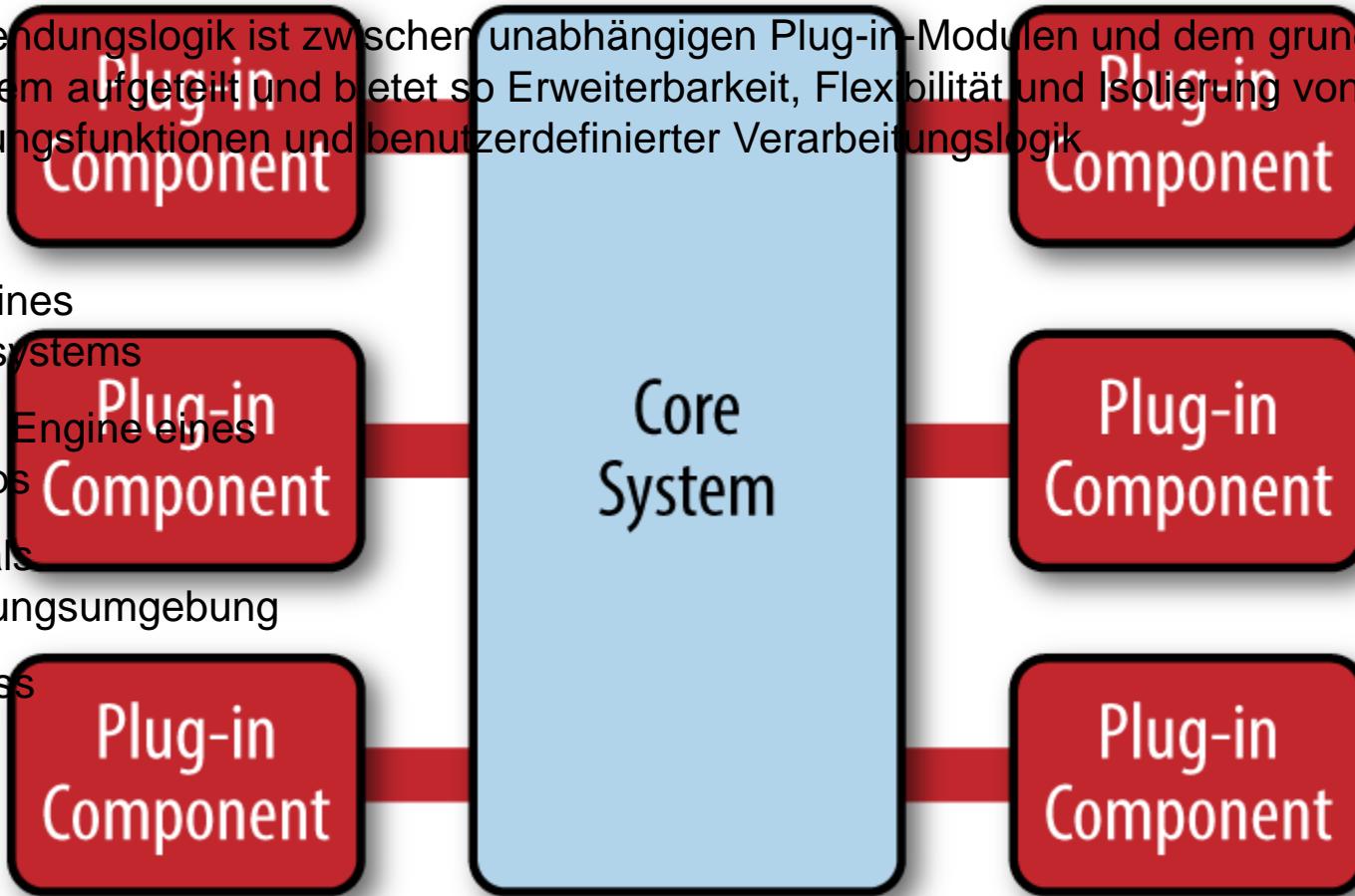


- **Warum:**
 - Mehrere Applikationen benutzen ähnliche Programmier-Schnittstellen, welche auf derselben Kernfunktionalität aufsetzen.
 - Dennoch sollen diese Applikationen erweitert werden können und auch Funktionstrennung und -isolierung bieten.
- **Herausforderung:**
 - Die Applikationsplattform muss immer die neuste Hardware- und Software-Generation unterstützen
 - Die Applikationsplattform muss portabel, erweiterbar und einfach in neue Systeme integrierbar sein
- **Lösung:**
 - Der funktionale Kern mit den wichtigsten Services der Applikationsplattform sollte in eine Komponente mit minimalem Speicherverbrauch (Microkernel) und weiteren Performance schonenden Services (Servers) aufgeteilt werden. Beispiele: Eclipse und generell Browser-Architekturen

- Das Microkernel-Architekturmuster besteht aus zwei Arten von Architekturkomponenten: einem Kernsystem und Plug-in-Modulen.
- Die Anwendungslogik ist zwischen unabhängigen Plug-in-Modulen und dem grundlegenden Kernsystem aufgeteilt und bietet so Erweiterbarkeit, Flexibilität und Isolierung von Anwendungsfunktionen und benutzerdefinierter Verarbeitungslogik.

Beispiele:

- Prinzip eines Betriebssystems
- Payment Engine eines Webshops
- Eclipse als Entwicklungsumgebung
- Wordpress



Vorteile:

- Lässt sich gut mit anderen Pattern verbinden (Schichten)
- Änderungen können schnell durch Plug-Ins realisiert werden
- Gute Testbarkeit
- Hohe Performance

Nachteile:

- Schlechte Skalierbarkeit aufgrund der meist eingeschränkten Produktgröße
- Man braucht im Vorfeld ein gutes Design, da die Umsetzung komplex werden kann
- In der Praxis kommt es häufig zu ungewollten Abhängigkeiten der Plug-Ins

- **Warum:**

- Monolithische Anwendungen sind schwer aufzutrennen um einzelne Komponenten wieder zu verwenden.
- Microservices sollen klein, autonom und miteinander kombinierbar sein.
- Die Zerlegung der Anwendung in einzelne Komponenten erfolgt dabei vertikal entlang der Fachlichkeit.

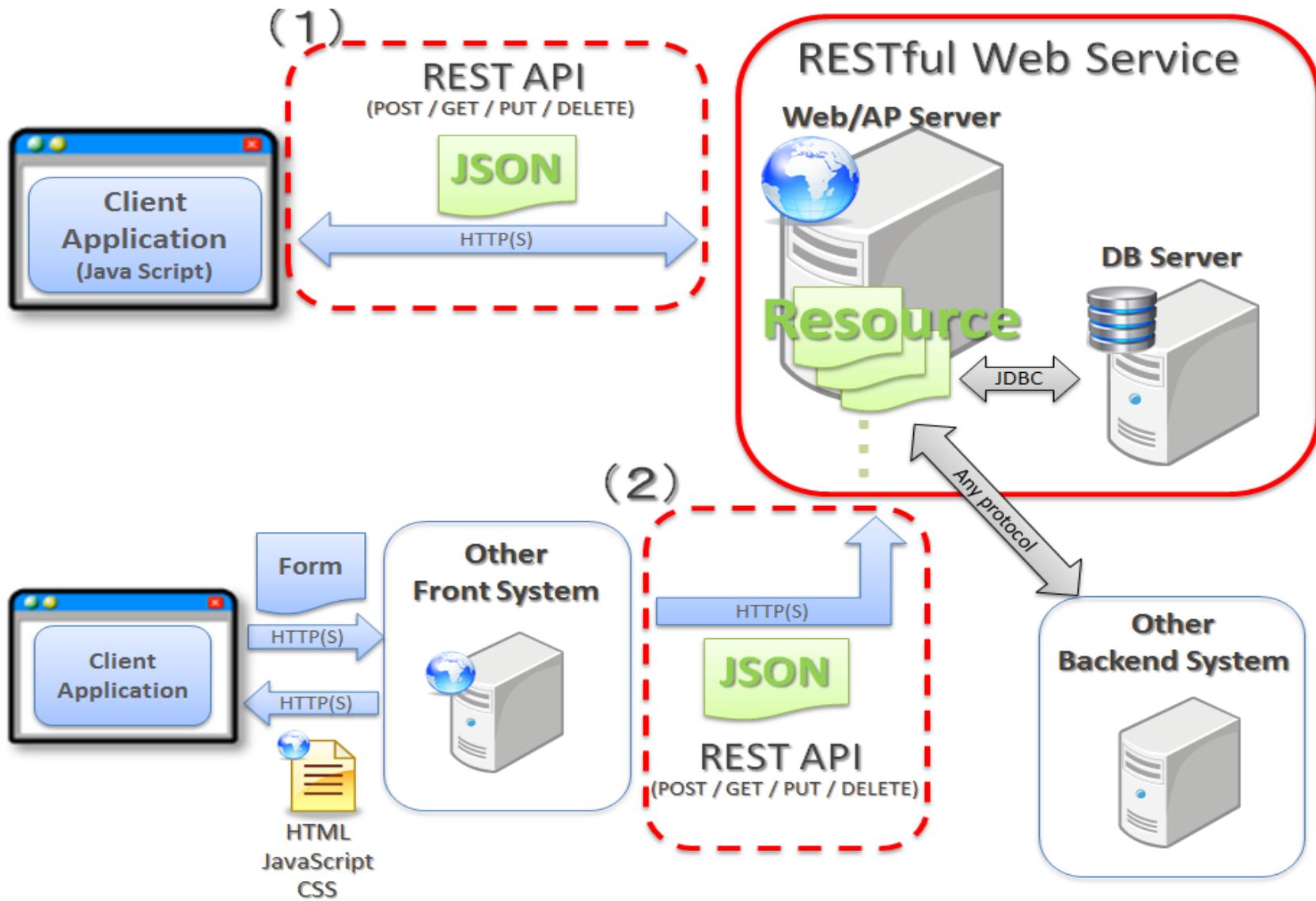
- **Herausforderung:**

- Nach welchen Kriterien erfolgt die Zerlegung der Anwendung, um die Anforderungen, wie klein und autonom zu erfüllen und die Vorteile von Microservices nicht zu verlieren?

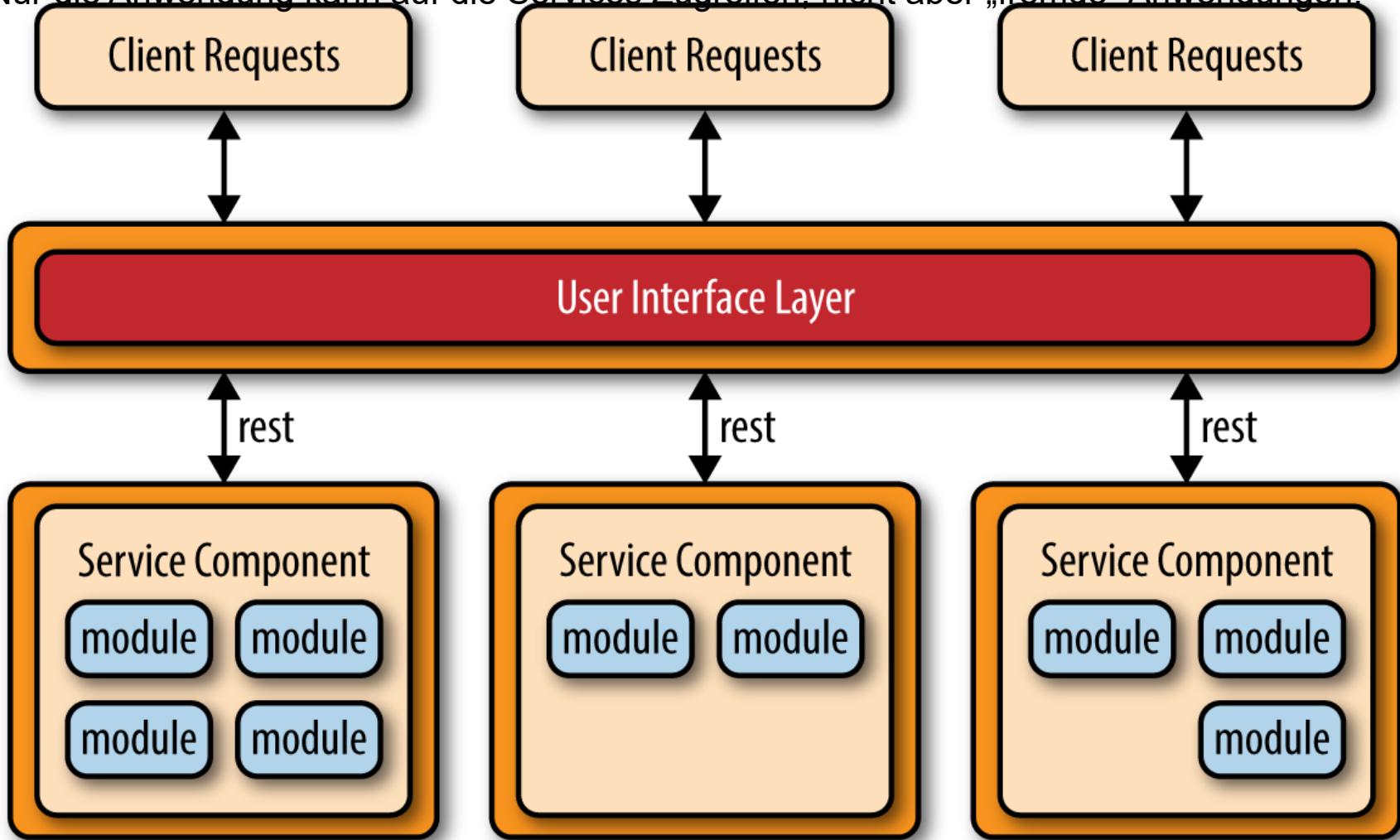
- **Lösung:**

- Die Microservice Architektur teilt große Anwendungsteile in kleinere auf.
- Diese Teile erfüllen folgende Anforderungen:
 - Einzeln installierbar
 - Robustheit
 - Skalierbarkeit
 - Erweiterbarkeit
 - Austauschbarkeit

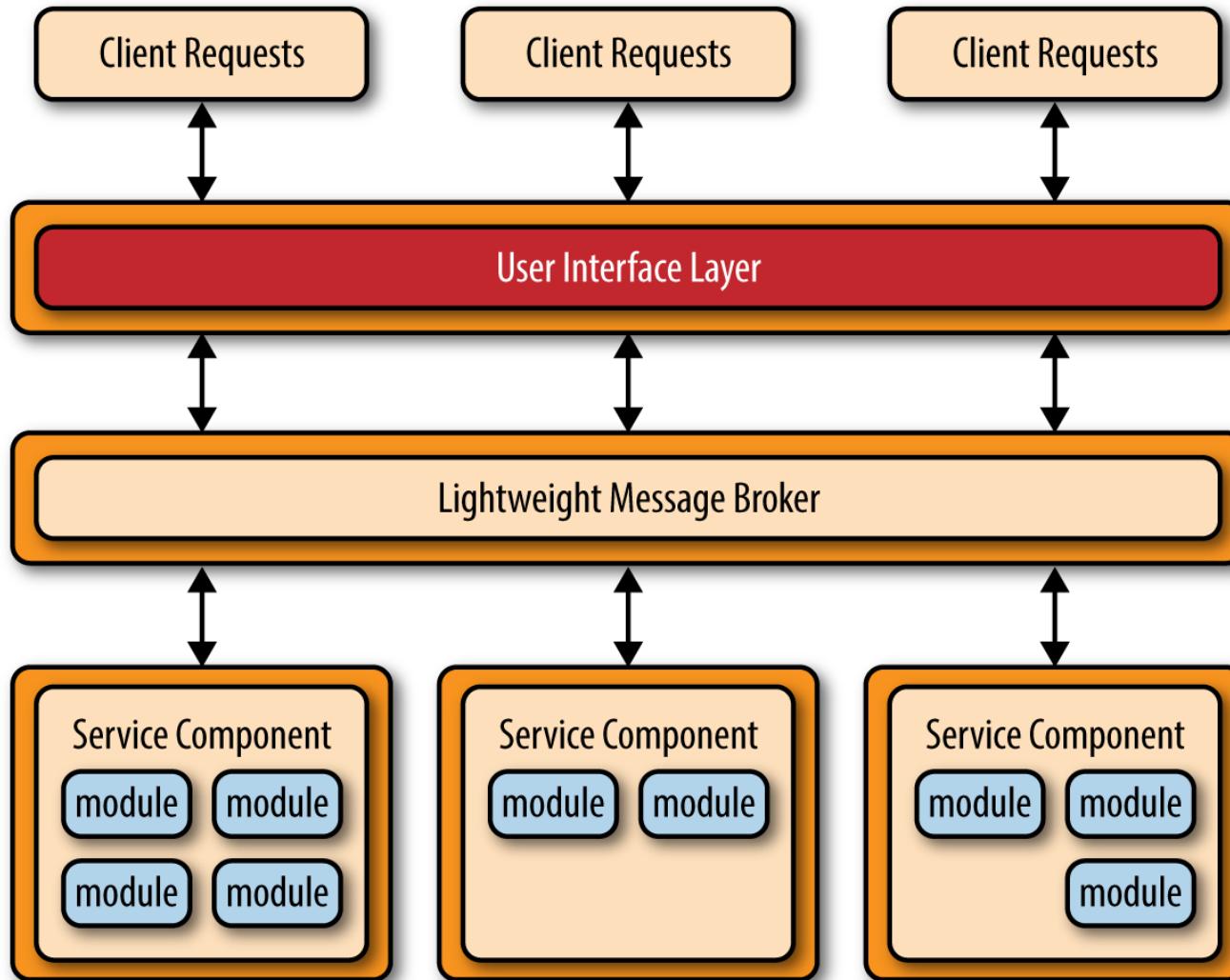
- Der Ansatz basiert ursprünglich auf dem SOA Pattern
- SOA Ansatz ist jedoch ein Overkill für viele Anwendungen
- Microservices sollen Services vereinfachen, Orchestration unnötig machen und den Zugriff auf Services erleichtern
- Es gibt 1000 Wege um eine solche Architektur zu implementieren und diese wurden auch beschrieben
- Die am häufigsten benutzten sind jedoch:
 - API REST-based
 - Application REST-based
 - Centralized messaging



Nur die Anwendung kann auf die Services Zugreifen, nicht aber „fremde“ Anwendungen:



Centralized messaging topology



Vorteile:

- Microservices sind klein und übersichtlich.
- Können leicht durch eine Neuimplementierung ersetzt werden.
- Microservices können unabhängig voneinander entwickelt, verteilt und skaliert werden.
- Das Gesamtsystem ist robust, da Microservices-Systeme gegen den Ausfall anderer Services abgesichert werden.
- Erlauben langfristig eine produktive Entwicklung des Systems, da sie wartbar bleiben und auch die Architektur des Gesamtsystems erhalten bleibt.

Nachteile:

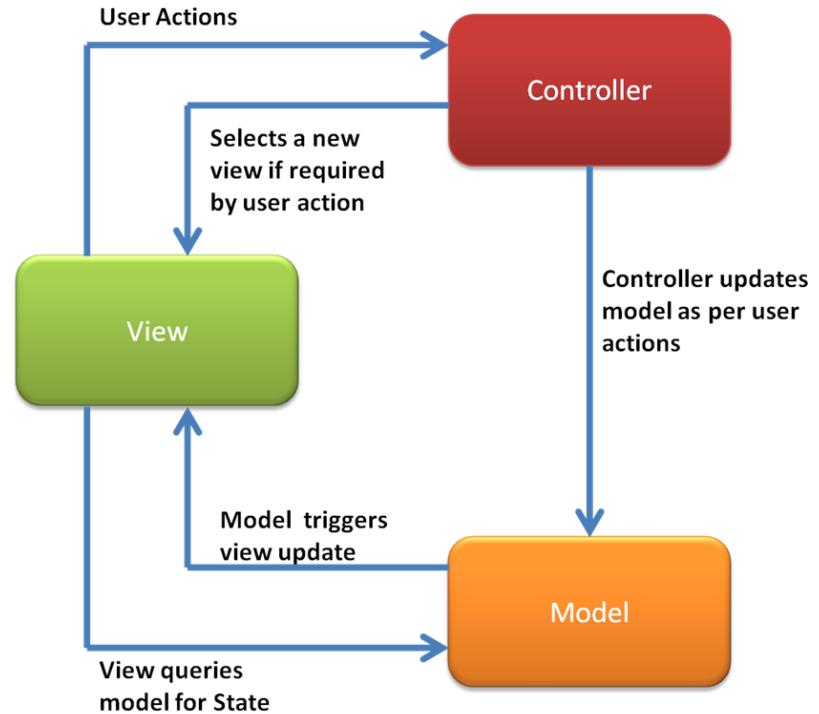
- Softwareverteilung und Testen sind komplexer
- Migrationsaufwand bestehender Systeme ist hoch und bedeutet meist eine Anpassung der Kommunikations-kultur in den beteiligten Organisationen
- Ausfallwahrscheinlichkeit von Komponenten steigt mit der Anzahl der Systeme.
- Die verteilte Architektur erzeugt zusätzliche Komplexität, vor allem durch Lastverteilung, Netzwerk-Latenzen oder Fehlertoleranz.
- Das Logging und Monitoring wird komplexer, da mehrere Systeme involviert sind, welche ggf. unterschiedliche Logging- und Monitoringtechnologien einsetzen

- **Warum:**
 - Benutzerschnittstellensoftware ist typischerweise der am häufigsten modifizierte Teil einer interaktiven Anwendung.
 - Benutzer möchten oftmals Daten aus verschiedenen Perspektiven betrachten. Diese Darstellungen sollten alle stets auf dem aktuellen Stand der Daten sein.
- **Herausforderung:**
 - Wie kann die Schnittstellenfunktionalität von der Applikationsfunktionalität getrennt werden? Und wie können multiple Ansichten des User Interface erstellt und gewartet werden, wie werden die Ansichten verwaltet, wenn sich darunterliegende Daten ändern?
- **Lösung:**
 - Das Modell-View-Kontroller (MVC) Pattern teilt Applikationsfunktionalität in drei Typen von Komponenten:
 - Ein Modell, das die Applikationsdaten enthält
 - Eine Sicht, die Teile der darunterliegenden Daten enthält und mit dem Benutzer interagiert
 - Ein Kontroller, der zwischen dem Modell und der Sicht vermittelt und die Benachrichtigungen über Statusänderungen verwaltet

Model-View-Controller Pattern (MVC)

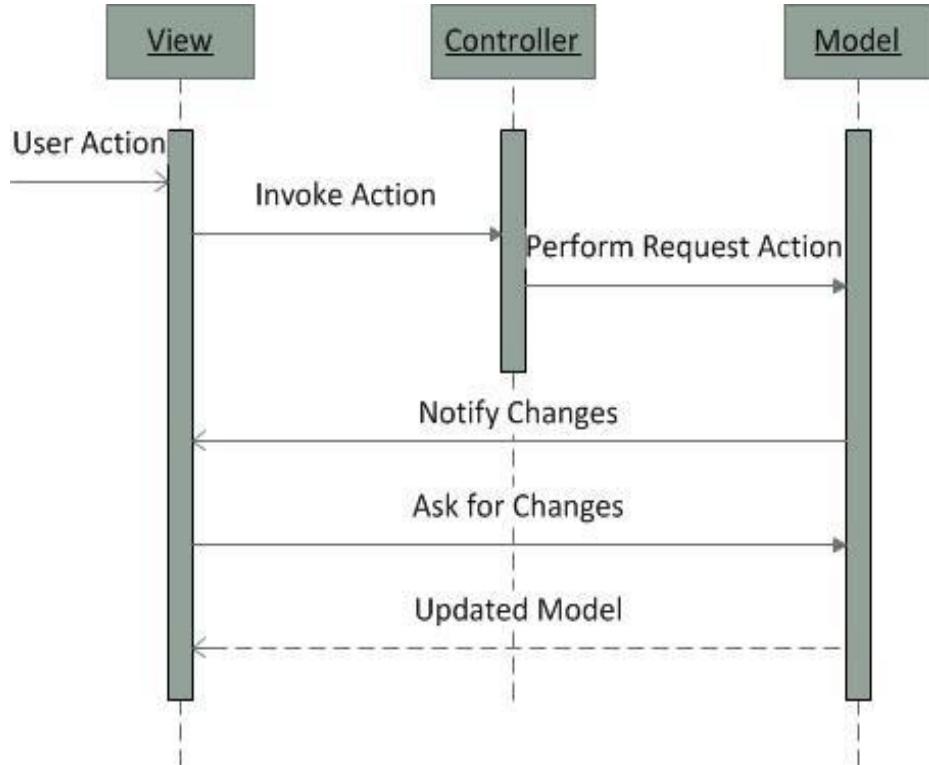
- View – Controller ist eine 1:1 Verbindung.
- Jedes Model kann n View-Controller Paare haben

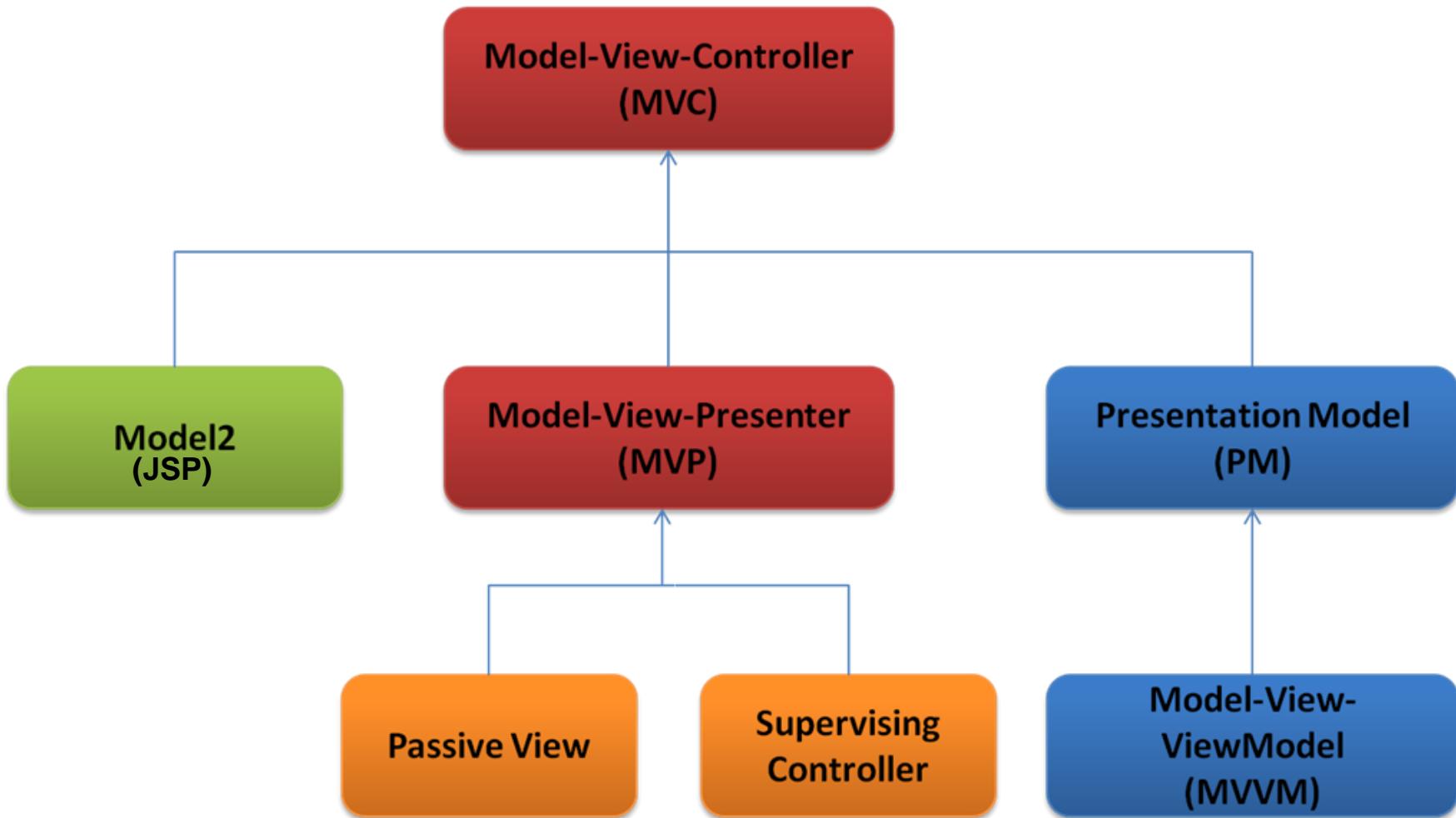
- Model:
 - Daten, die erforderlich sind, um in der Ansicht angezeigt zu werden.
- View:
 - Ansicht, welche Methoden den Controllers ruft. Überwacht Model auf Statusänderungen (Observer Pattern)
- Controller
 - Eingebunden von der View und interagiert mit dem Model. Er hat keine Kenntnis über die Änderungen



Model-View-Controller Pattern (MVC)

- Das größte Missverständnis tritt beim Controller auf
- Er vermittelt NICHT zwischen View und Model und ist NICHT zuständig für das Ändern der View
- Er nimmt die Useraktion auf und ändert das Model
- Es ist die Aufgabe der View den Status und Änderungen vom Model zu rendern
- Das klassische MVC Pattern ist stark an das Observer Pattern gebunden und findet heute nur mehr selten Einsatz.





Vorteile

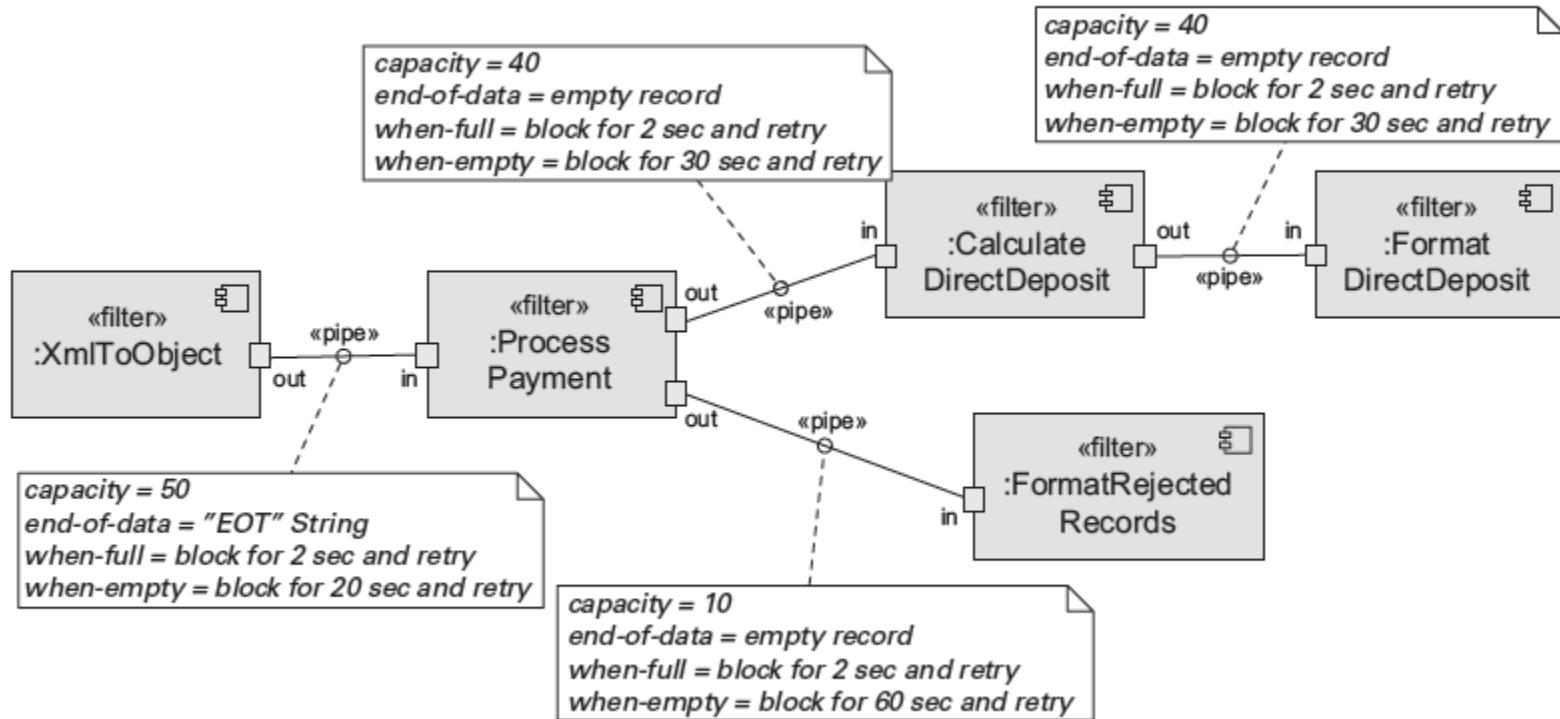
- Mehrere Ansichten auf das
selbe Modell.
- Automatische Synchronisation
aller Ansichten.
- Austauschbarkeit von Ansicht
und Controller.
- Gute Trennung von Modell und
VC.
- Potential für vorgefertigte
Frameworks.

Nachteile

- Größere Komplexität
- Potential für unnötige häufige
Aktualisierungen
- Enge Verbindung zwischen
View und Controller
- Enge Kopplung von VC an das
Modell.
- Häufig ineffizienter Datenzugriff
auf das Modell.
- View und Controller sind schwer
zu portieren.

- **Warum:**
 - In vielen Systemen ist es erforderlich Datenelemente vom Eingang zum Ausgang zu transformieren.
 - Viele Arten von Transformationen treten in der Praxis wiederholt auf
- **Herausforderung:**
 - Solche Systeme müssen in wiederverwendbaren, lose gekoppelten, mit einfachen Interaktionsmechanismen ausgestattete Komponenten aufgeteilt werden.
 - Die lose gekoppelten Komponenten sind einfach wieder zu verwenden.
 - Diese unabhängigen Komponenten können parallel ausgeführt werden
- **Lösung:**
 - Das Pipes und Filter Pattern ist durch aufeinanderfolgende Transformationen von Datenströmen charakterisiert.
 - Daten gelangen zu einem/mehreren Eingangsanschluss des Filters, werden umgewandelt, und dann über den/die Ausgangs-Port(s) zum nächsten Filter ‚gepiped‘.
 - Ein einzelner Filter kann Daten von einem oder mehreren Ports erhalten oder erzeugen

Pipes and Filter Pattern



Vorteile

- Flexibilität durch Austausch und Hinzufügen neuen von Filtern
- Flexibilität durch Neuanordnung
- Wiederverwendung einzelner Filter
- Rapid Prototyping von Pipeline Prototypen

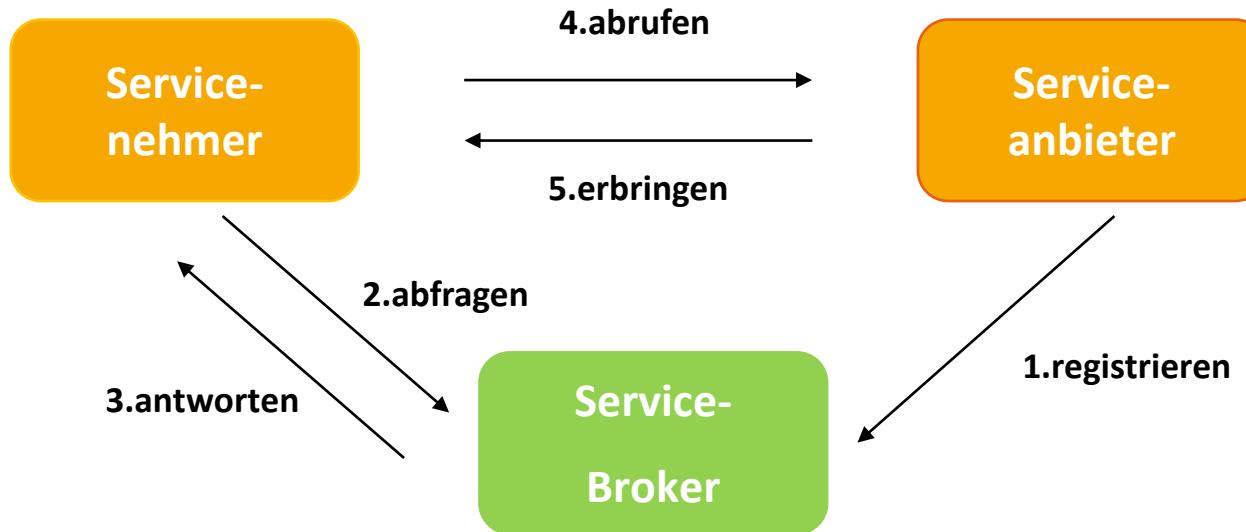
Nachteile

- Die Kosten der Datenübertragung zwischen den Filtern können je nach Pipe sehr hoch sein
- Häufig überflüssige Datentransformationen zwischen den einzelnen Filterstufen
- Fehlerbehandlung über Filterstufen hinweg ist teilweise schwierig

Verbindlichkeiten

- **Steuerung über zentrale Instanz oder über Pipes**
- **Fehlermeldungen müssen mit den Daten übertragen werden**

- **Warum:**
 - Eine Anzahl an Diensten werden von Service-Providern angeboten (und beschrieben) und von Service-Verbraucher angefordert/benötigt.
 - Service-Verbraucher müssen in der Lage sein, diese Dienste ohne detaillierte Kenntnisse über deren Umsetzung zu verstehen und zu verwenden
- **Herausforderung:**
 - Wie können wir die Interoperabilität von verteilten Komponenten, die
 - auf verschiedenen Plattformen laufen,
 - in verschiedenen Implementierungssprachen umgesetzt sind,
 - von verschiedenen Organisationen zur Verfügung gestellt und
 - über das Internet verteilt sind,unterstützen?
- **Lösung:**
 - Das serviceorientierte Architektur (SOA) Pattern beschreibt eine Sammlung von verteilten Komponenten, die Services anbieten/anfordern.



- Die Zuordnung von Diensten zu deren Spezifikationen wird über ein Dienstverzeichnis (Service Registry) organisiert.
- Die tatsächliche Bindung zwischen den Softwareelementen Dienstnutzer (Service Consumer) und Dienstanbieter (Service Provider) erfolgt dynamisch zur Ausführungszeit

Vorteile

- Weniger Redundanzen. Kernfunktionen werden nur einmal erstellt und wieder verwendet.
- Höhere Flexibilität. Wenn sich Geschäftsprozesse ändern kann die Software leichter angepasst oder ergänzt werden.
- Verteilte Anwendungen können auch über das Internet genutzt werden. Damit können externe Partner auf Kunden und Lieferseite besser angebunden werden
- Die Zusammenarbeit über Unternehmensgrenzen wird einfacher.
- Für das zugrunde liegende XML Format gibt es zahlreiche Tools.

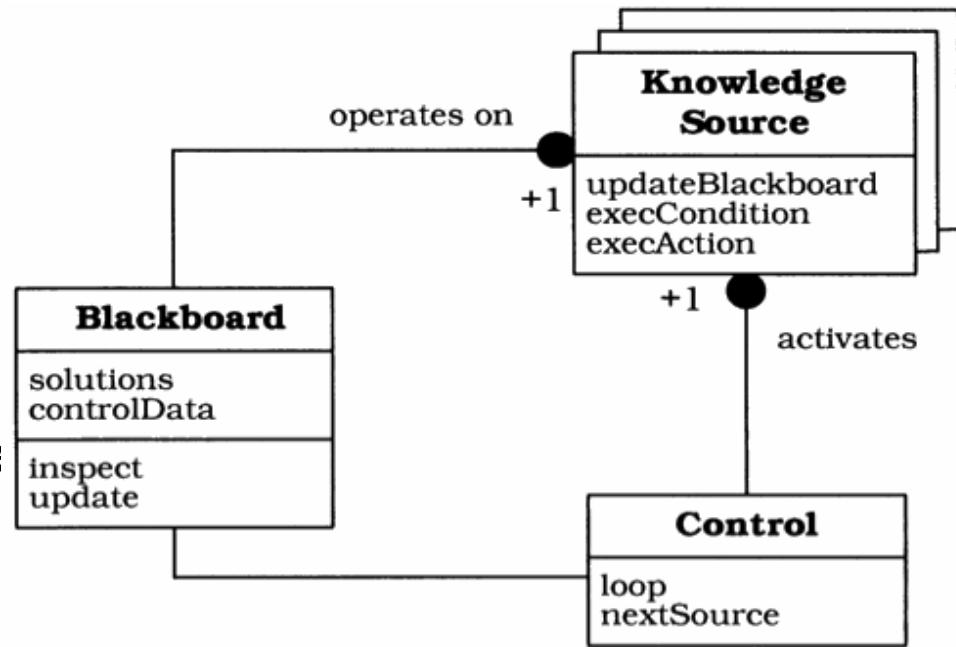
Nachteile

- Die lose Kopplung erhöht die Verarbeitungszeit einzelnen Aufgaben.
- Die Komplexität bei der Wartung und Versionierung ist höher
- Für die Integration der Services gibt es unterschiedliche Technologien wie z.B. der Enterprise Service Bus (ESB)
- Die Einhaltung fester Antwortzeiten ist bei der losen Kopplung schwieriger. Wichtig ist daher ein internes und externes Service Level Agreement (SLA).
- Unter Umständen kann höhere Rechenleistung durch die Übertragung der Daten zwischen den Services benötigt werden.

- **Warum:**
 - Es gibt keine deterministische Lösung für ein bestimmtes Problem.
 - Eine komplett Absuche des Lösungsraumes ist wegen der Komplexität nicht möglich
- **Herausforderung:**
 - Das Endresultat ist nicht eindeutig.
 - Unsichere Zwischenresultate sind involviert
 - Verschiedene Teilgebiete können nicht sequentiell bearbeitet werden.
 - Es besteht nicht genügend Wissen um das Problem rein statistisch zu lösen
- **Lösung:**
 - Eine Sammlung von unabhängigen Programmen (Knowledge Sources / Experts) arbeiten zusammen an einer gemeinsamen Datenstruktur (Blackboard).
 - Jedes Programm ist dabei spezialisiert auf sein Teilgebiet.
 - Die einzelnen Programme rufen sich weder gegenseitig auf, noch ist eine bestimmte Reihenfolge der Einsätze der Programme festgelegt.

Wir kennen zwei Versionen des Blackboard Pattern: Aktiv und Passiv

- Aktives Blackboard verwaltet die gemeinsamen Daten und stellt sie der Control Component und den Knowledge Sources zur Verfügung
- Knowledge Sources (Experts)
 - condition part: Evaluiert den aktuellen Zustand des Lösungsfortschrittes
 - action part: Produziert Resultate anhand seiner Informationen und schreibt diese auf das Blackboard
- Control Component
 - Überwacht das Blackboard
 - Koordiniert, wählt und aktiviert Knowledge Sources



Vorteile

- Flexibilität bei Problemen bei welchen das komplette Absuchen des Lösungsraumes nicht möglich ist.
- Gut erweiterbar und wartbar durch strikte Separierung der Zuständigkeiten
- Wiederverwendbarkeit der Knowledge sourcen
- Fehlertoleranz und Robustheit

Nachteile

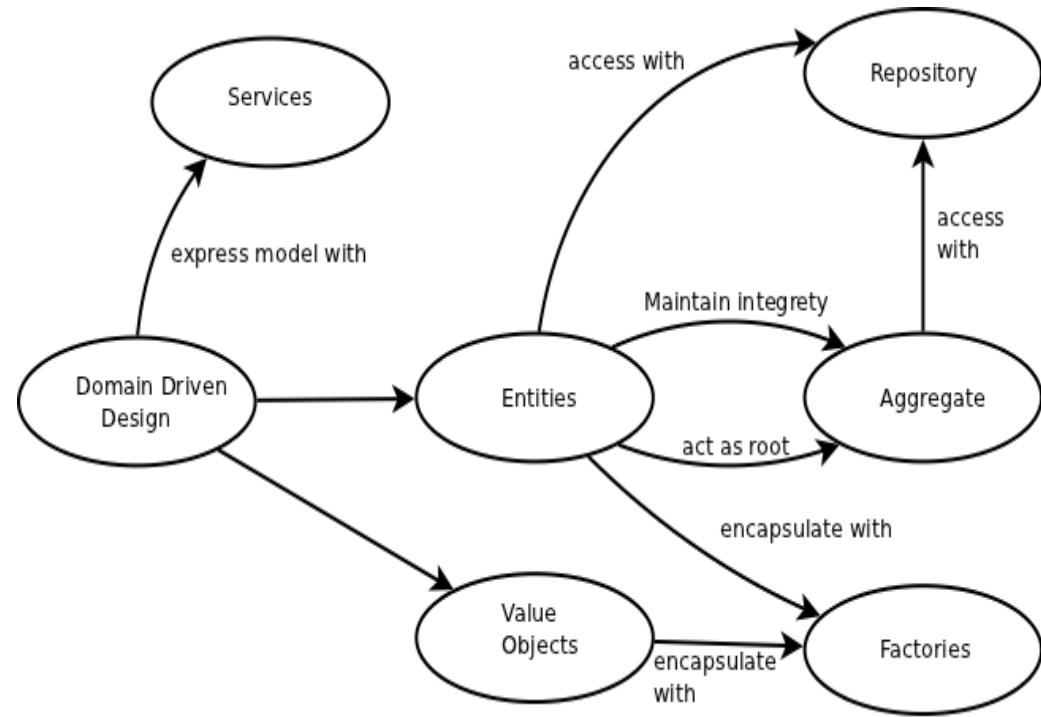
- Testen gestaltet sich schwierig
- Keine brauchbare Lösung garantiert
- Schwierig eine gute Kontrollstrategie zu entwickeln (erfordert eine experimentelle Vorgehensweise)
- Effizienz eher tief
- Hoher Entwicklungsaufwand
- Synchronisierung von parallelen Vorgängen kann komplex werden

Domain Driven Design basiert auf zwei Annahmen

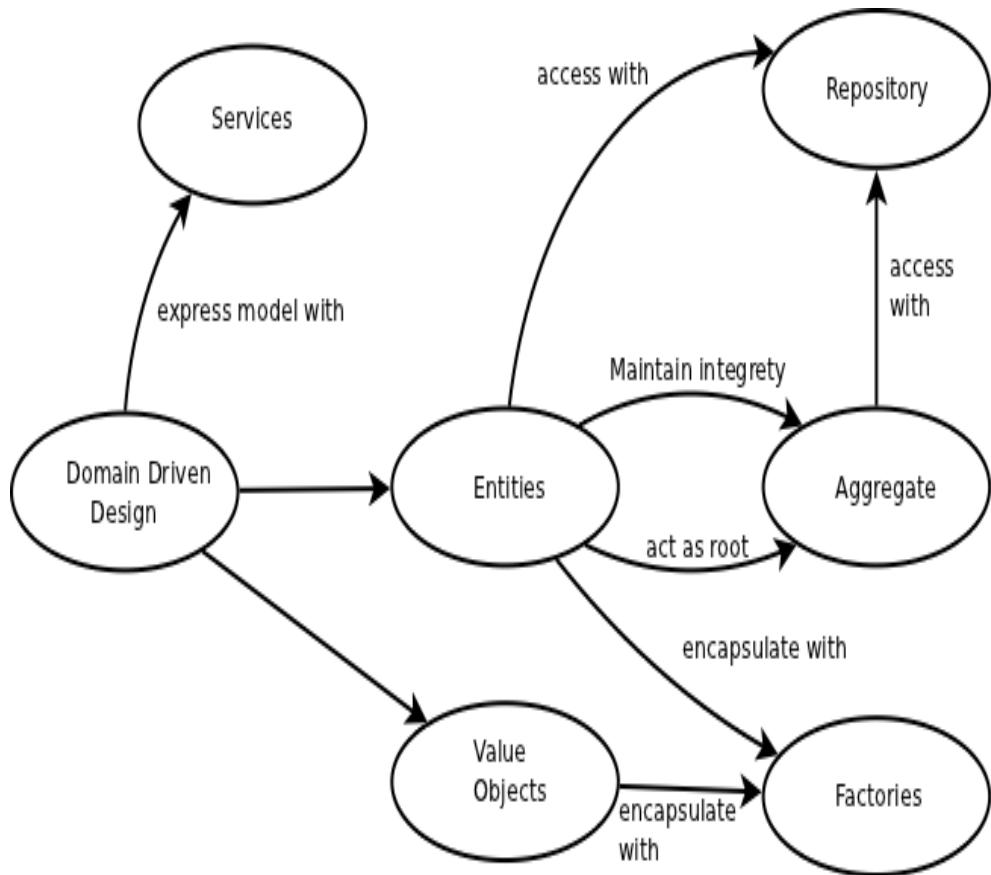
- Der Schwerpunkt des Softwaredesigns liegt auf der Fachlichkeit und der Fachlogik.
- Der Entwurf komplexer fachlicher Zusammenhänge sollte auf einem Fachmodell basieren.

-> Explizit machen von impliziten Zusammenhängen

- **Serviceobjekte (services)**
Funktionalitäten, welche ein wichtiges Konzept der Fachlichkeit darstellen und konzeptionell zu mehreren Objekten des Domänenmodells gehören
- **Entitäten (Entities)**
Objekte des Modelles, welche nicht durch ihre Eigenschaften, sondern durch ihre Identität definiert werden
- **Wertobjekte (value objects)**
Objekte des Modelles, welche keine konzeptionelle Identität haben oder benötigen und somit allein durch ihre Eigenschaften definiert werden



- **Aggregate** sind Zusammenfassungen von Entitäten und Wertobjekten und deren Assoziationen untereinander zu einer gemeinsamen transaktionalen Einheit. Aggregate definieren genau eine Entität als einzigen Zugriff auf das gesamte Aggregat
- **Fabriken** dienen dazu, die Erzeugung von Fachobjekten in spezielle Fabrik-Objekte auszulagern
- **Repositories** abstrahieren die Persistierung und Suche von Fachobjekten. Mittels Repositories werden die technische Infrastruktur sowie alle Zugriffsmechanismen auf diese von der Geschäftslogikschicht getrennt



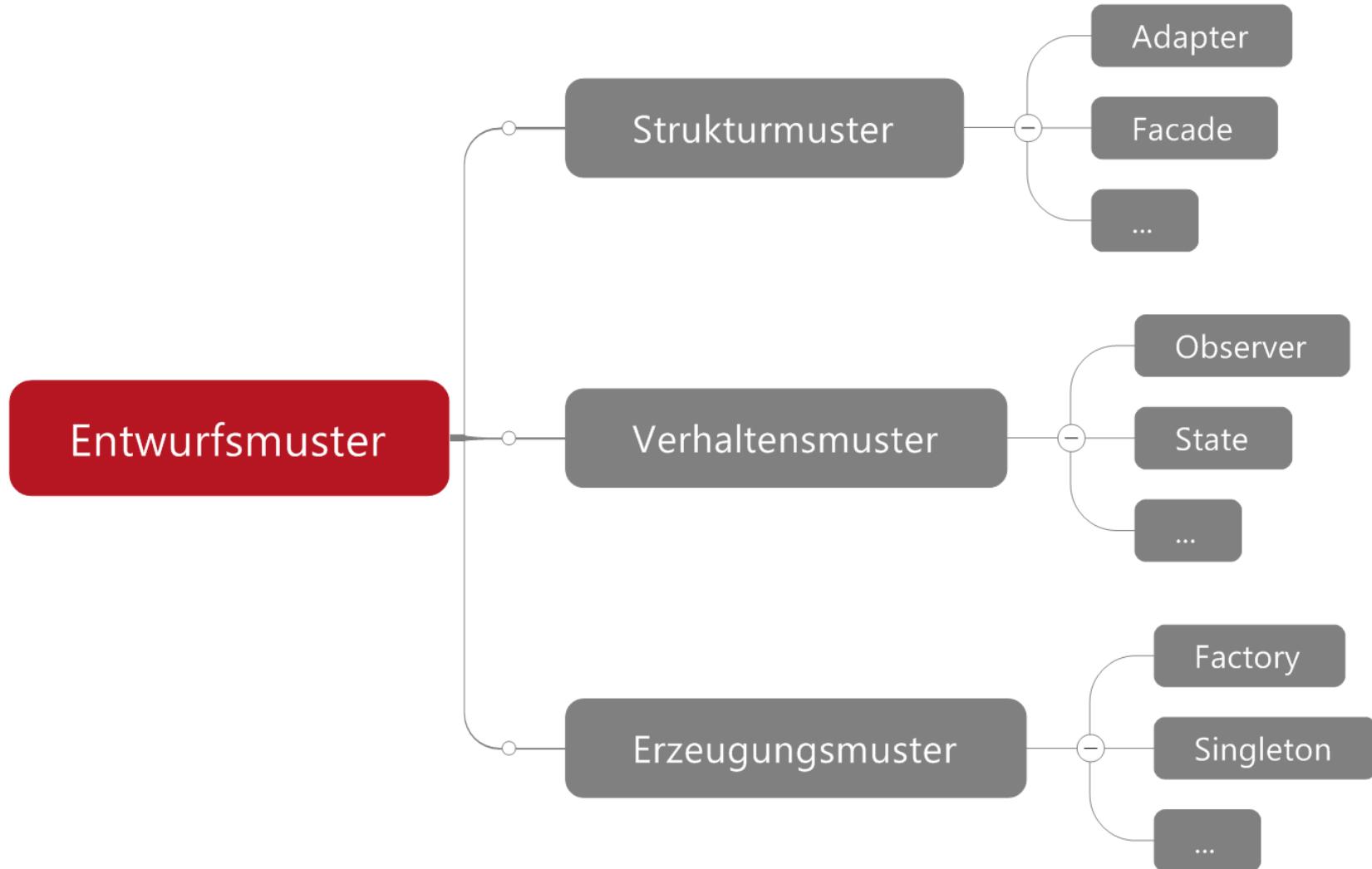
Vorteile

- Besseres Verständnis der Domäne
- Saubere Strukturierung des Codes
- Zuständigkeiten klar getrennt
- Leicht zu erweitern
- schnelleres Time-to-market

Nachteile

- Der Initialaufwand zur Entwicklung
- Es wird oft nur ein Rahmen erzeugt, welcher noch von Hand um die tatsächliche Funktion ergänzt werden muss
- Projektaufwand wird unterschätzt
- Eine Unterstützung zur Fehlersuche auf Modellebene ist oft nicht oder nur unvollständig vorhanden

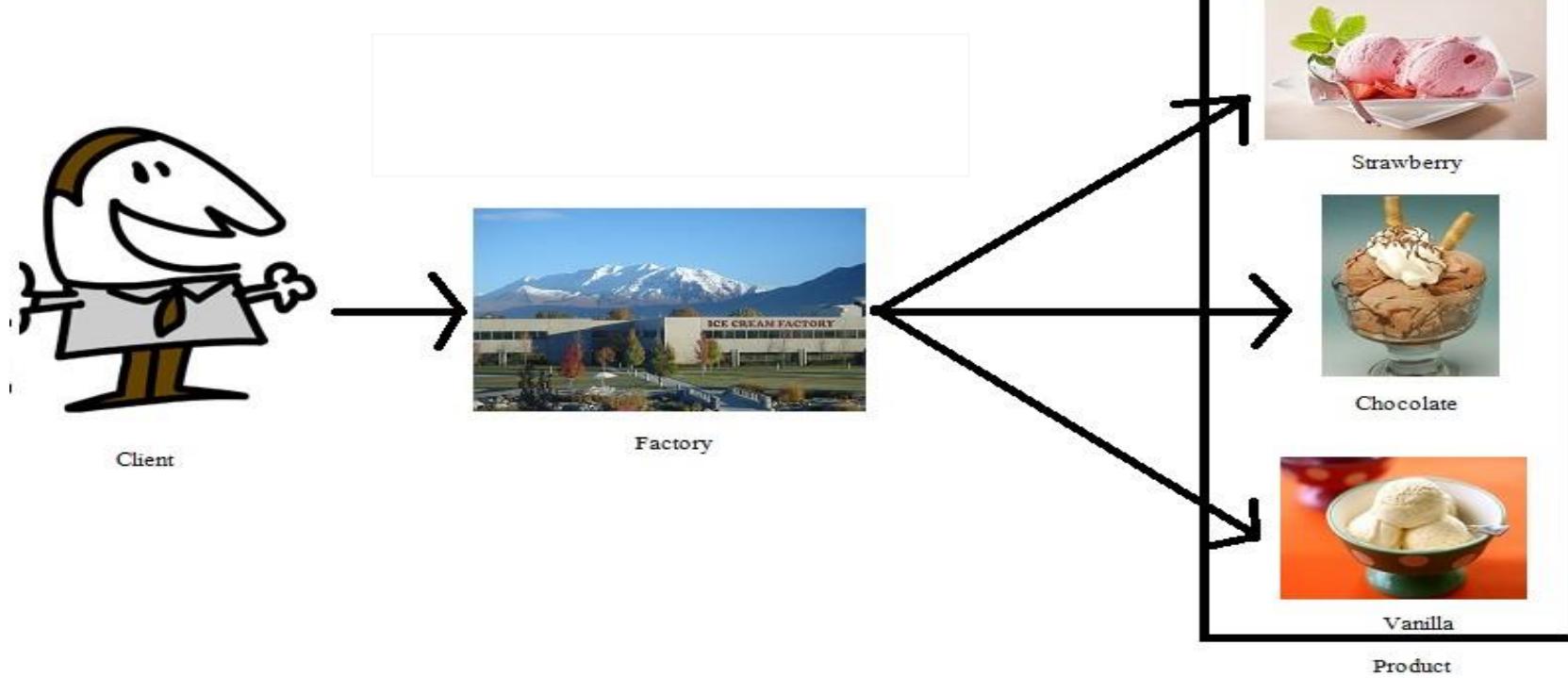
- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - **Architekturrelevante Entwurfsmuster**
 - Übergreifende technische Konzepte
 - Schnittstellen entwerfen



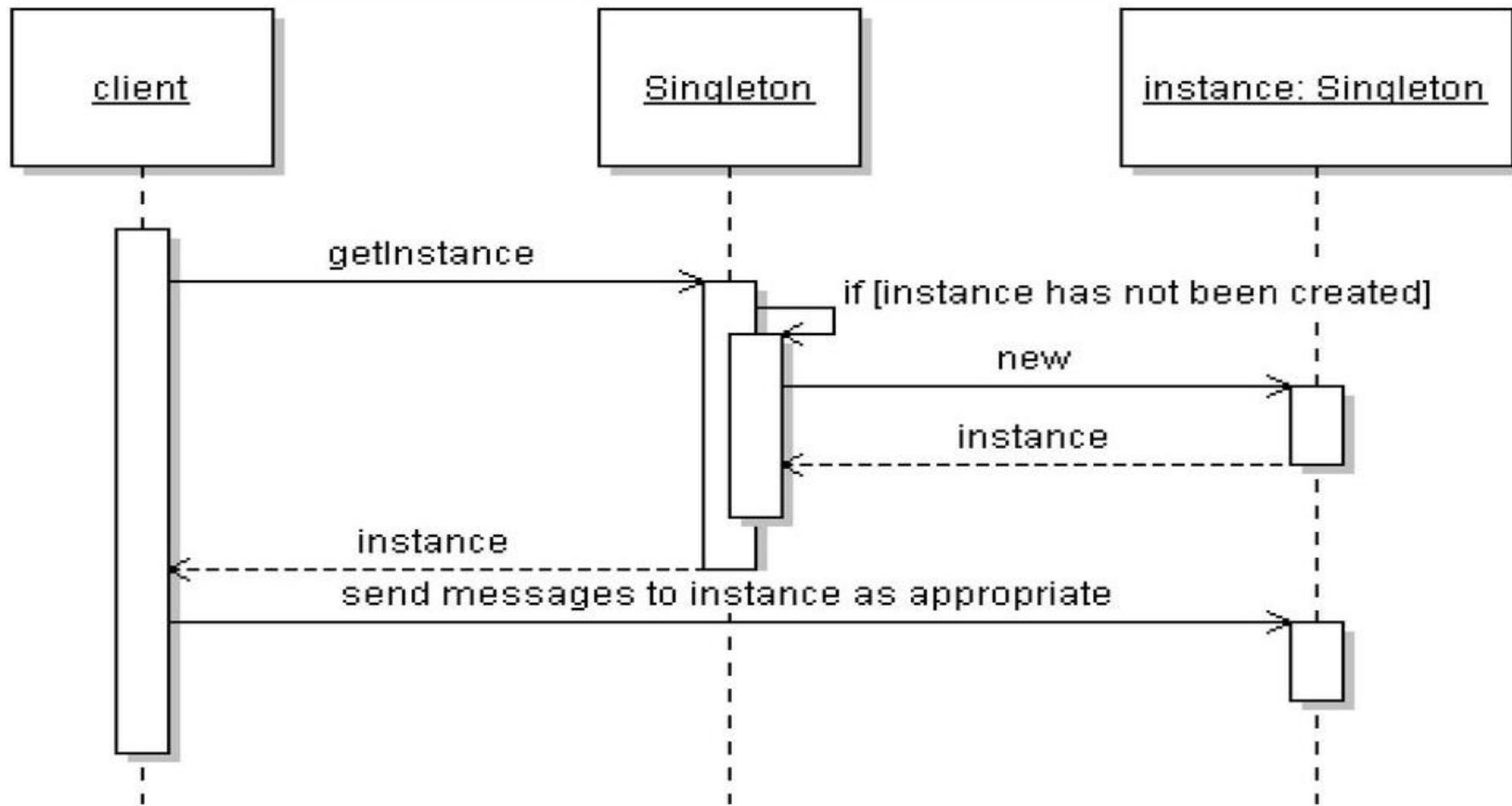
Factory Method (Fabrikmethode)

- "Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren."

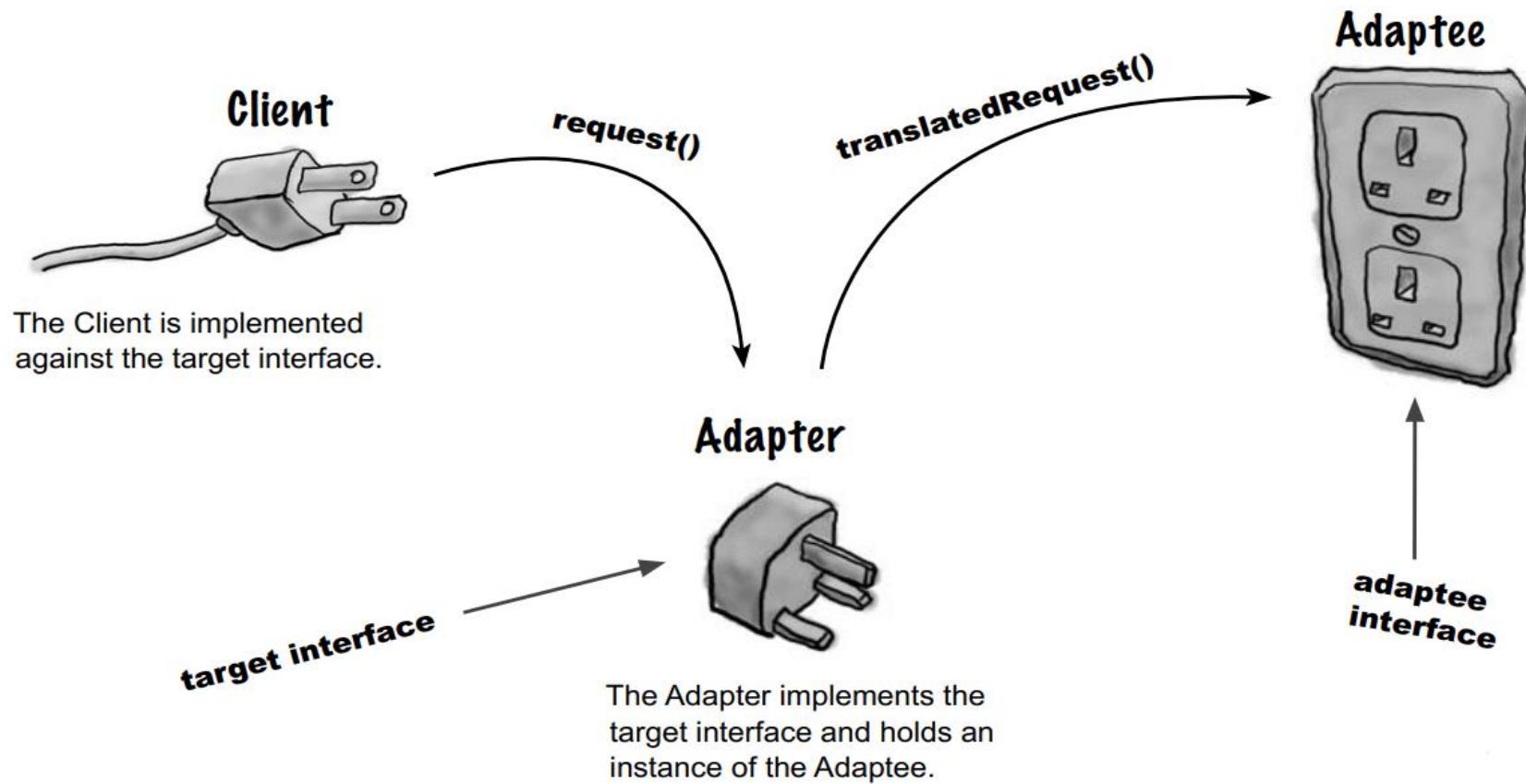
Erich Gamma



- "Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit."

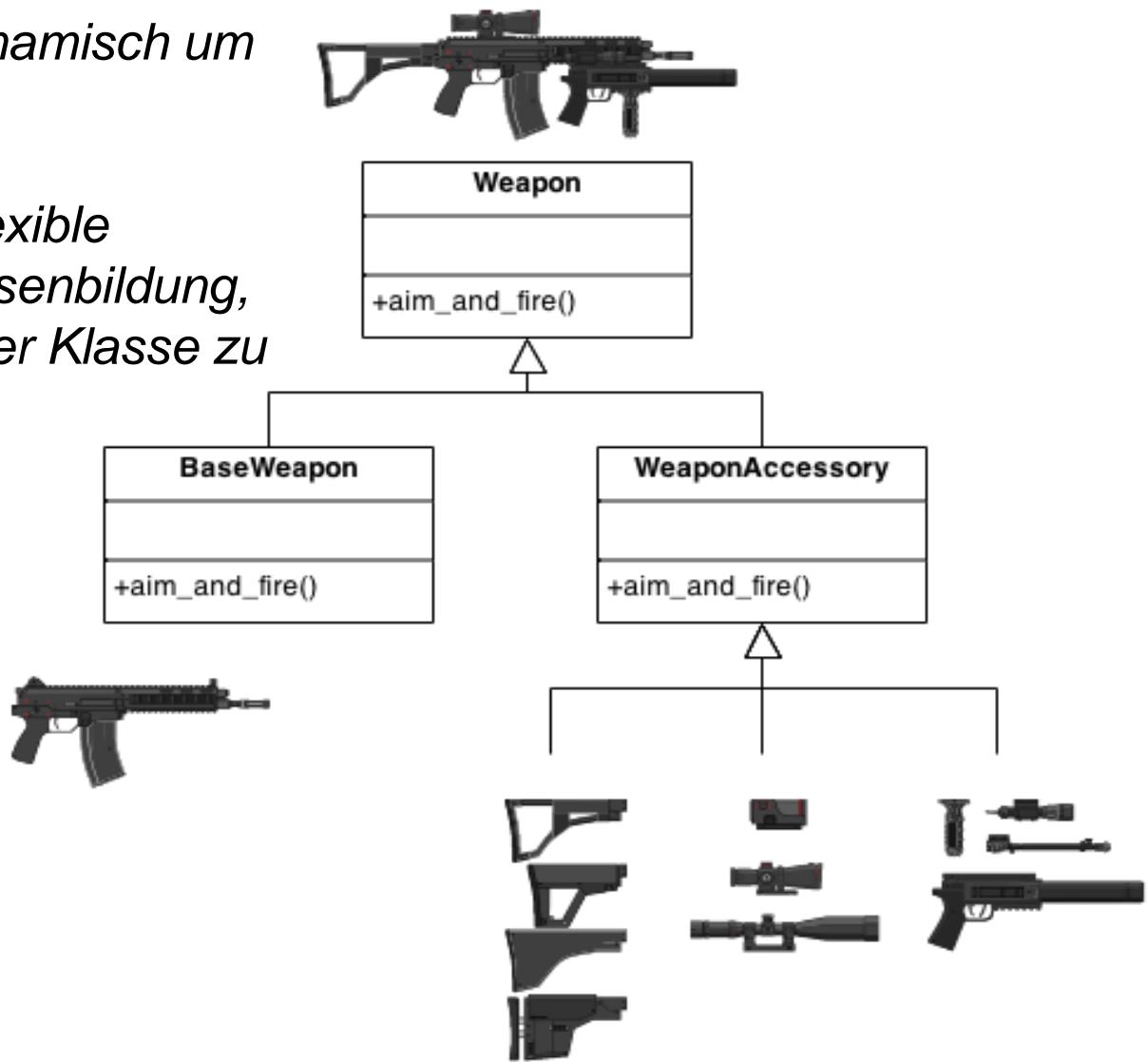


"Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster lässt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten nicht zusammenarbeiten könnten."

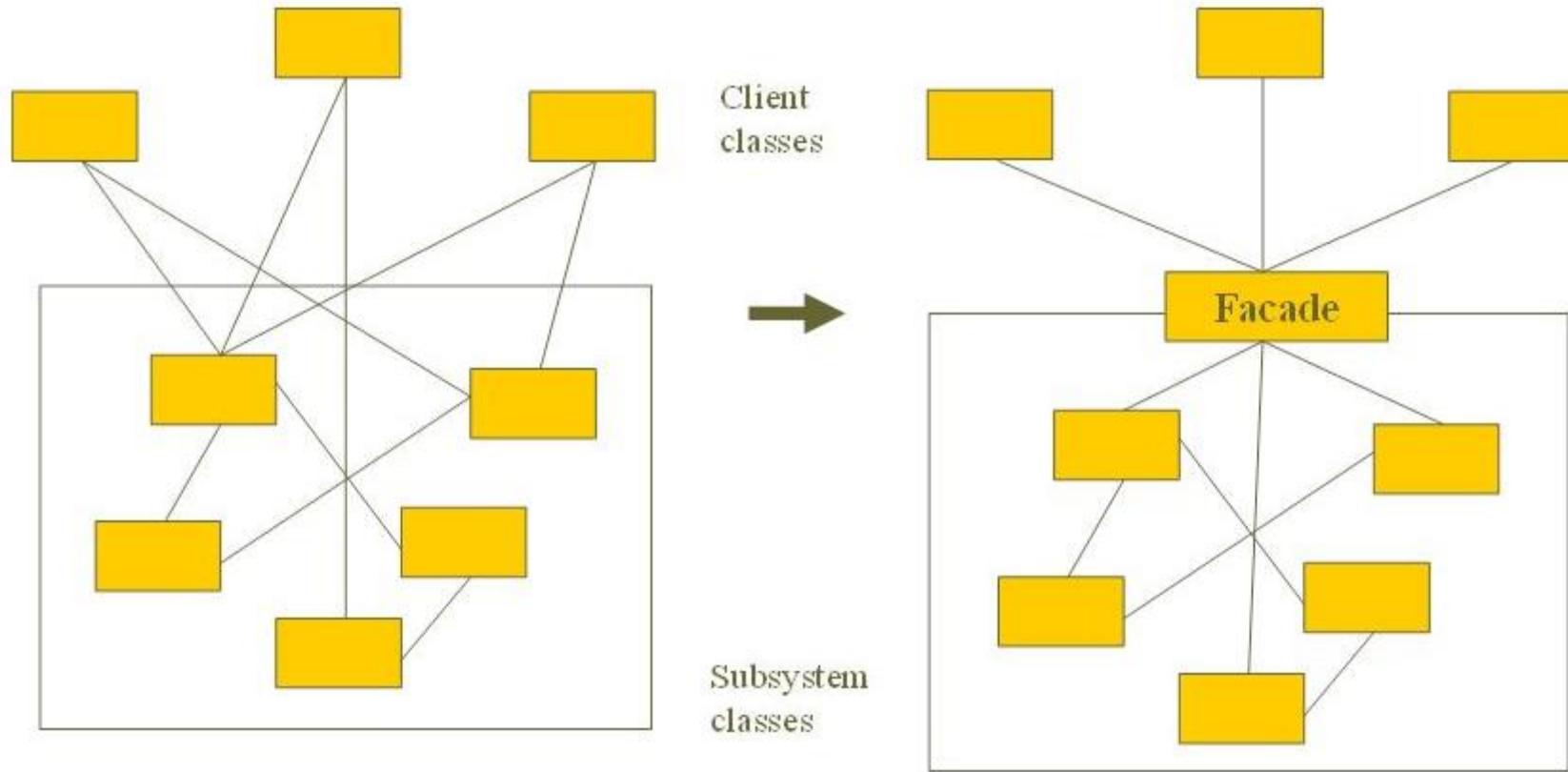


"Erweitere ein Objekt dynamisch um Zuständigkeiten."

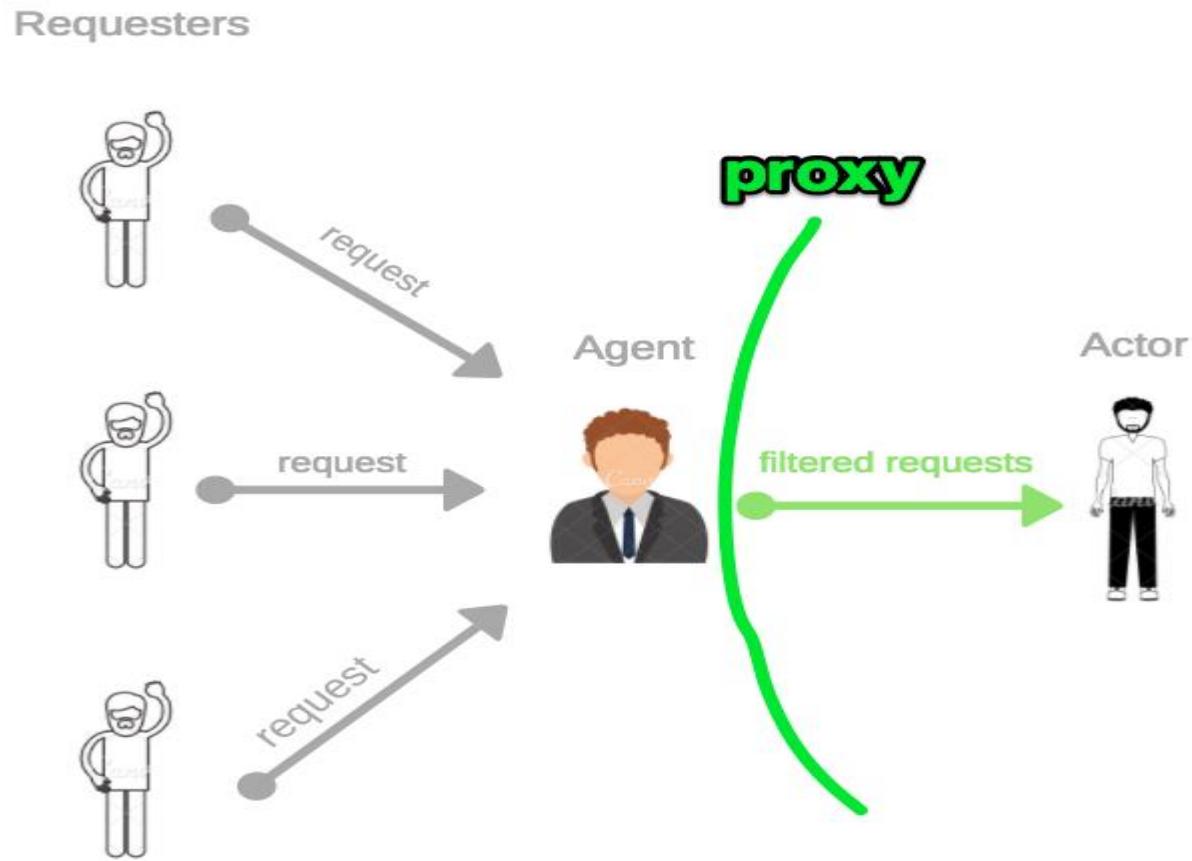
Dekorierer bieten eine flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern."



"Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.“

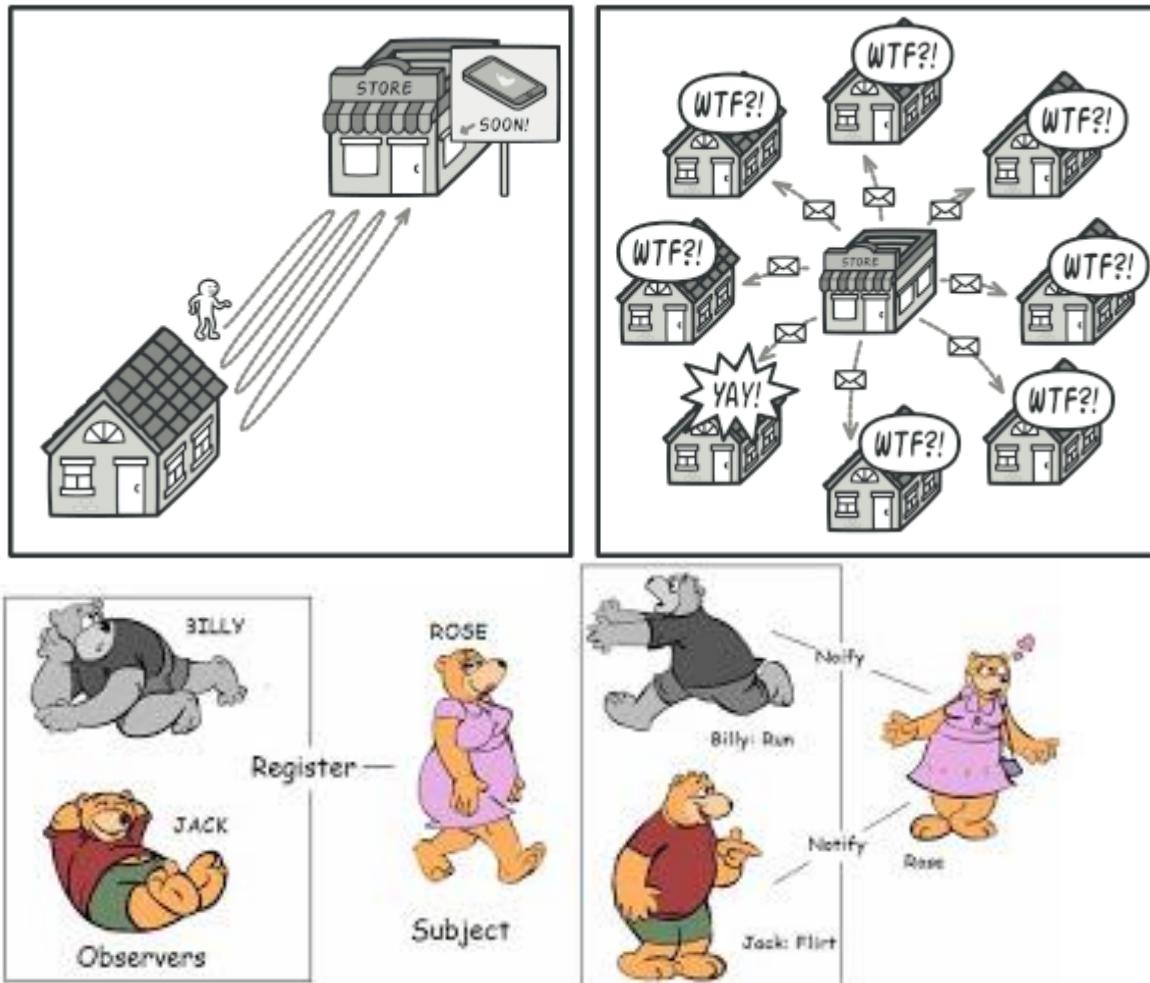


"Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts."

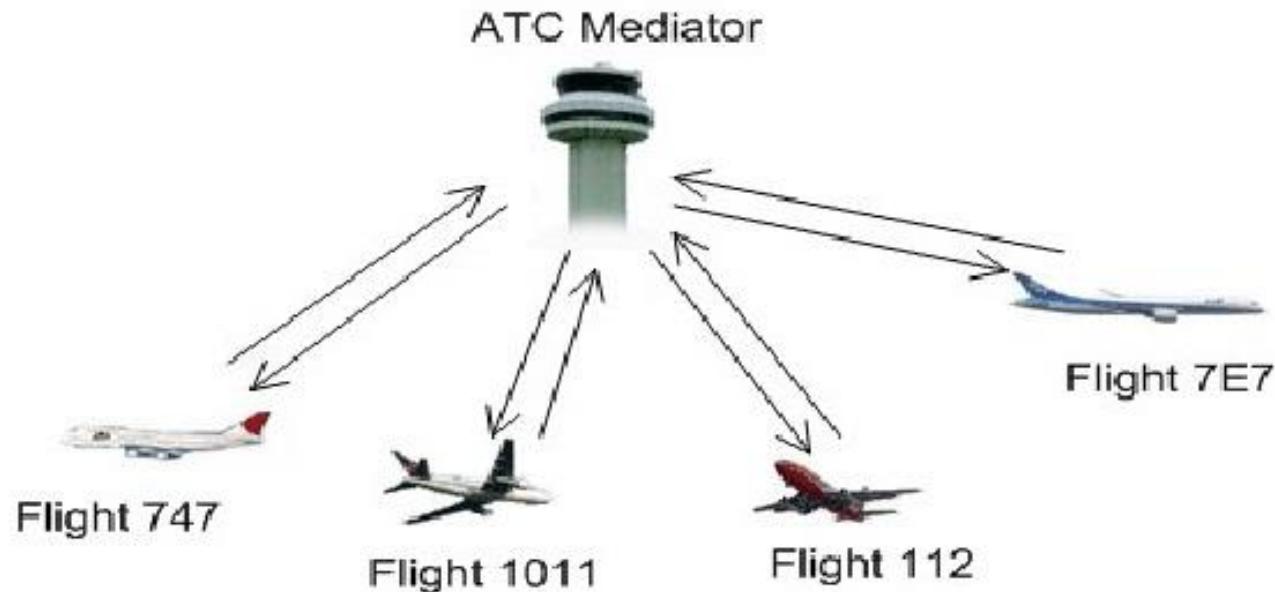


Observer (Beobachter)

"Definiere eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden."



- "Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es Ihnen, das Zusammenspiel der Objekte von ihnen unabhängig zu variieren."



"Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hätte."



States:

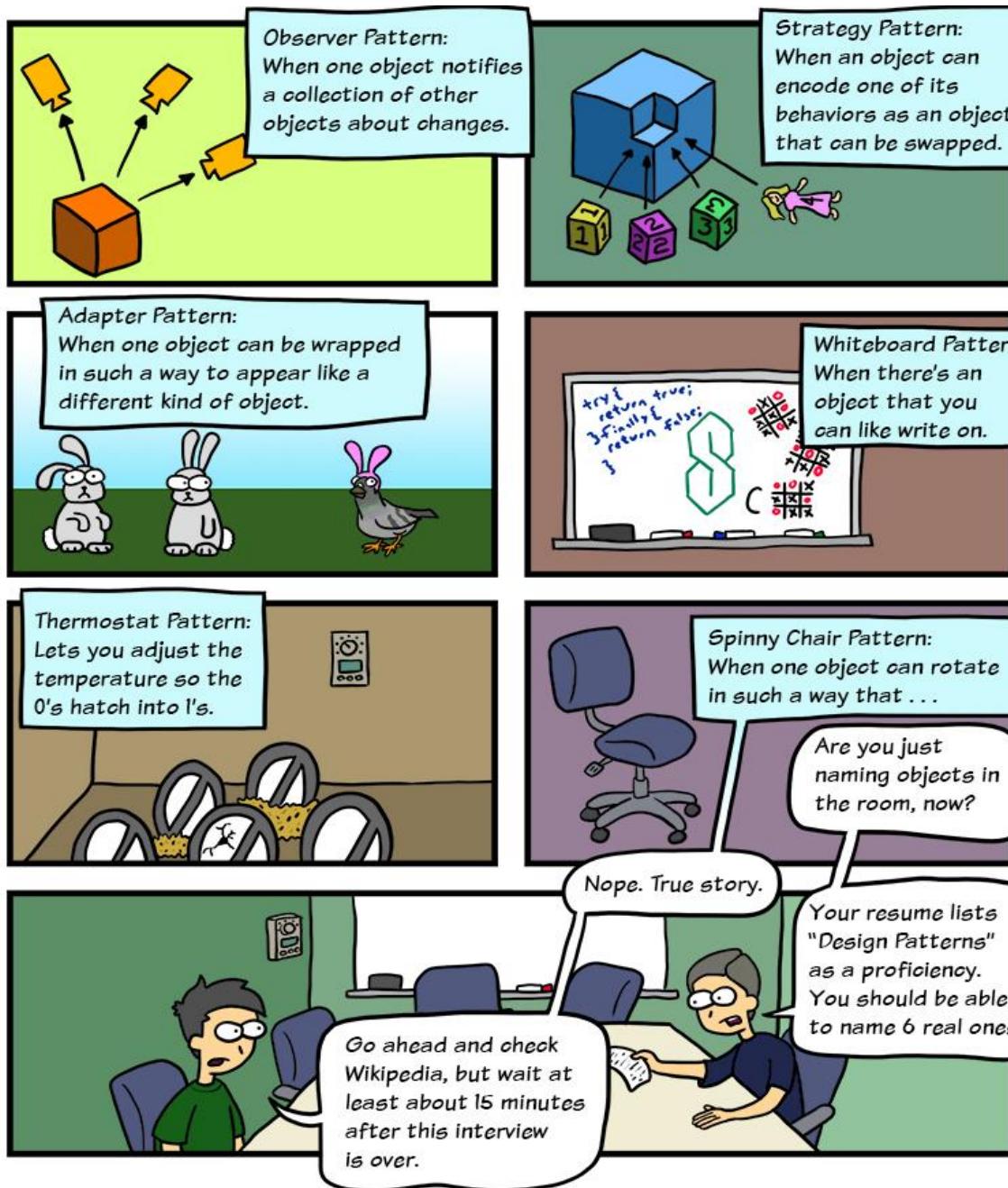
Money Not Inserted and Product Not Selected

Behaviors

1. You can select the product and Insert the Money
2. You cannot get the Product From the Vending Machine

Money Inserted and Product Selected

1. You can get the Product from the Vending Machine
2. You can get the balance Amount



- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - **Übergreifende technische Konzepte**
 - Schnittstellen entwerfen

Die Umsetzung kann auf einem einzelnen Baustein, oder verteilt auf mehrere Bausteine erfolgen



- Der Architekt muss die Testbarkeit des zu implementierenden Systems immer im Auge behalten
- Unterstützung für einfache (und möglichst automatische) Tests
- Diese Eigenschaft bildet die Grundlage für das wichtige Erfolgsmuster "Continuous Integration"
- In Projekten sollte mindestens täglich der gesamte Stand der Entwicklung gebaut und (automatisch) getestet werden - daher spielt Testbarkeit eine wichtige Rolle
- Wichtige Stichworte hierzu sind Unit- Tests und Mock-Objekte

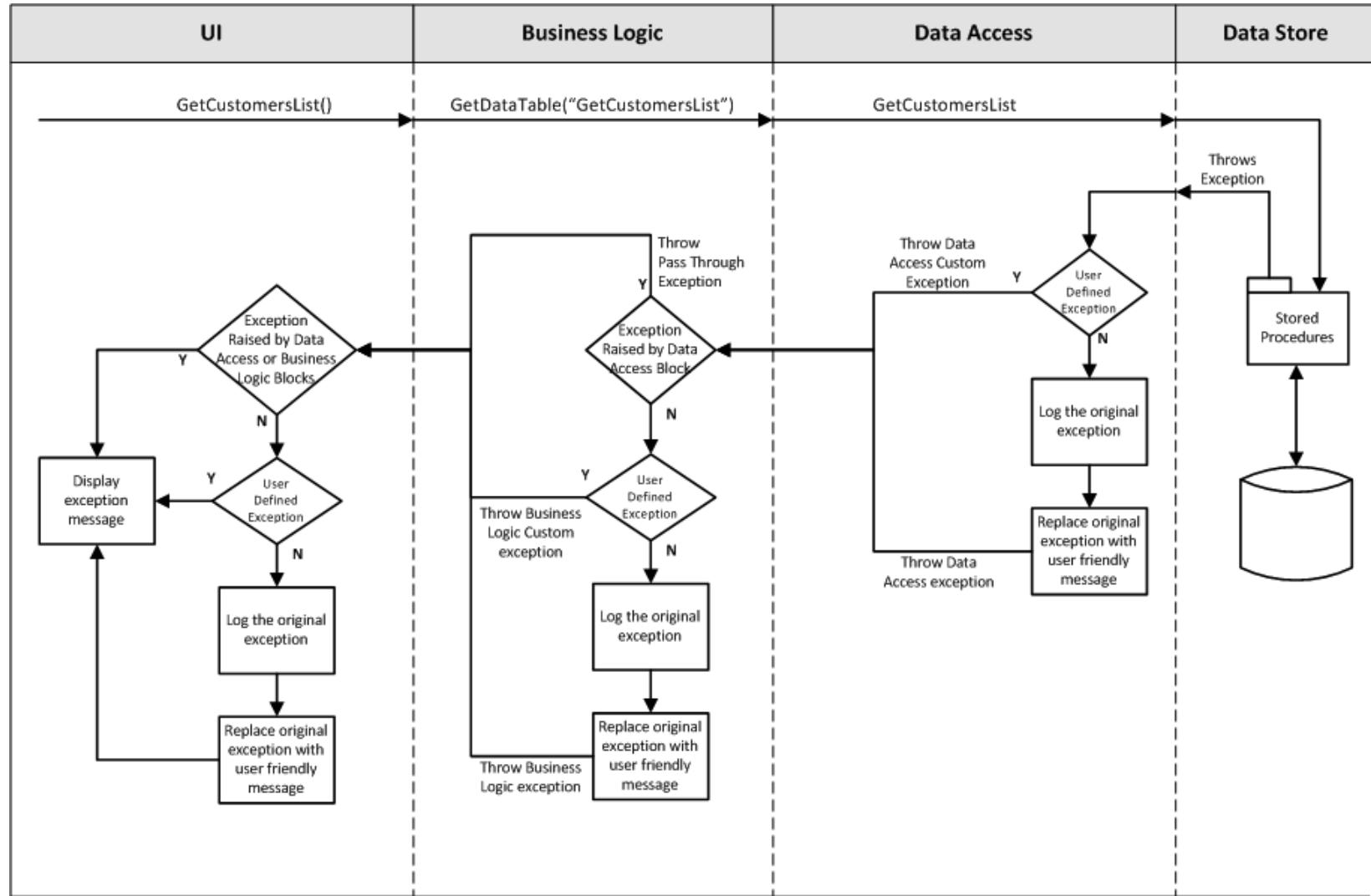
- Programme können in parallelen Prozessen oder Threads ablaufen - was die Notwendigkeit von Synchronisationspunkten mit sich bringt
- Für die Architektur und Implementierung nebenläufiger Systeme sind viele technische Detailaspekte zu berücksichtigen
 - Adressräume,
 - Arten von Synchronisationsmechanismen (Guards, Wächter, Semaphore),
 - Prozesse und Threads,
 - Parallelität im Betriebssystem,
 - Parallelität in virtuellen Maschinen und anderen
- Parallelisierung ist nicht immer durch ein Framework abgedeckt

- Unterstützung für den Einsatz von Systemen in unterschiedlichen Ländern, Anpassung der Systeme an länderspezifische Merkmale.
- Bei der Internationalisierung (aufgrund der 18 Buchstaben zwischen „I“ und „n“ des englischen Begriffs „Internationalisation“ auch i18n genannt) geht es neben der Übersetzung von Aus- oder Eingabetexten auch um:
 - verwendete Zeichensätze,
 - Orientierung von Schriften,
 - Anwendungscode,
 - Technische Einheiten,
 - Fehlerbehandlung
 - Logging
 - ...

Es gibt zwei Ausprägungen der Protokollierung, das *Logging* und das *Tracing*.

- **Logging** kann fachliche oder technische Protokollierung sein, oder eine beliebige Kombination von beidem
 - Fachliche Protokolle werden gewöhnlich anwenderspezifisch aufbereitet und übersetzt.
 - Sie dienen Endbenutzern, Administratoren oder Betreibern von Softwaresystemen und liefern Informationen über die vom Programm abgewickelten Geschäftsprozesse
 - Technische Protokolle sind Informationen für Betreiber oder Entwickler. Sie dienen der Fehlersuche sowie der Systemoptimierung
- **Tracing** soll Debugging -Information für Entwickler oder Supportmitarbeiter liefern.
 - Es dient primär zur Fehlersuche und -analyse

Fehlerbehandlung



- Persistenz (Dauerhaftigkeit, Beständigkeit) bedeutet, Daten aus dem (flüchtigen) Hauptspeicher auf ein beständiges Medium (und wieder zurück) zu bringen
- Persistenz ist primär eher ein technisch bedingtes Thema, trägt aber in vielen Fällen zum fachlichen Nutzen eines Systems bei
- Deshalb sollten Sie sich als Architekt mit dem Thema auseinander setzen, denn ein erheblicher Teil der meisten Software-Systeme benötigt einen effizienten Zugriff auf persistent gespeicherte Daten
- Hierzu gehören praktisch sämtliche kommerzielle aber auch viele technischen Systeme

- IT-Systeme, die von (menschlichen) Benutzern interaktiv genutzt werden, benötigen eine Benutzungsoberfläche. Das können sowohl grafische als auch textuelle Oberflächen sein
- Hier kommt schnell das Thema UX Design zum tragen
- User Experience spielt eine große Rolle bei der Produkt- und Serviceentwicklung.
- In vielen Bereichen wird Vitruvius als erster Architekt und Designer gesehen, der mit den Begriffen Firmitas (Festigkeit), Utilitas (Nützlichkeit, Usability) und Venustas (Schönheit) die Kriterien für das Nutzerlebnis definiert hat, wenn es auch damals noch eher auf Gebäude ausgerichtet war
- Dabei können eine Reihe von NFA abgeleitet werden.
 - Festigkeit bzw. Stabilität
 - Nützlichkeit
 - Schönheit

- Ablaufsteuerung von IT-Systemen bezieht sich sowohl auf die an der (grafischen) Oberfläche sichtbaren Abläufe als auch auf die Steuerung der Hintergrundaktivitäten
- Zur Ablaufsteuerung gehört daher unter anderem die Steuerung der Benutzeroberfläche als auch die Workflow-Steuerung
- Beispiele
 - Workflow Systeme
 - jBPM
 - Activiti
 - Business Rule Engines
 - Drools

- Transaktionen sind Arbeitsschritte oder Abläufe, die entweder alle gemeinsam oder gar nicht durchgeführt werden. Der Begriff ist bekannt durch viele Relationale Datenbanken
- Wichtiges Stichwort hier sind „ACID“
- **Atomar**
 - Eine Transaktion wird entweder ganz oder gar nicht ausgeführt.
- **Consistent**
 - Nach Ausführung der Transaktion muss der Datenbestand in einer konsistenten Form sein
- **Isolated**
 - Bei gleichzeitiger Ausführung mehrerer Transaktionen dürfen sich diese nicht gegenseitig beeinflussen
- **Durable**
 - Die Auswirkungen einer Transaktion müssen im Datenbestand dauerhaft bestehen bleiben.

- Die Sicherheit von IT-Systemen befasst sich mit Mechanismen zur Gewährleistung von Datensicherheit und Datenschutz sowie Verhinderung von Datenmissbrauch
- Typische Fragestellungen sind:
 - Wie können Daten auf dem Transportweg (beispielsweise über offene Netze wie das Internet) vor Missbrauch geschützt werden?
 - Wie können Kommunikationspartner sich gegenseitig vertrauen?
 - Wie können sich Kommunikationspartner eindeutig erkennen und vor falschen Kommunikationspartner schützen?
 - Wie können Kommunikationspartner die Herkunft von Daten für sich beanspruchen (oder die Echtheit von Daten bestätigen)?
- Das Thema IT-Sicherheit hat häufig Berührung zu juristischen Aspekten, teilweise sogar zu internationalem Recht

- Vorbereitung zur Architekturentwicklung
- **Entwurfsprozess und Vorgehensweise**
 - Einflussfaktoren
 - Entwurfsprinzipien
 - Abhängigkeiten und Kopplung von Bausteinen
 - Wichtige Architekturmuster und Architekturstile
 - Architekturrelevante Entwurfsmuster
 - Übergreifende technische Konzepte
 - **Schnittstellen entwerfen**

- Schnittstellen sind eine der zentralen Komponenten in Architekturen
- Schnittstellen entkoppeln Systembausteine voneinander

- Wünschenswerte Eigenschaften von Schnittstellen sind
 - Einfach zu erlernen, einfach zu benutzen, einfach zu erweitern
 - Schwer zu missbrauchen
 - Funktional vollständig aus Sicht der Nutzer oder nutzenden Bausteine

- Schnittstellen müssen gut dokumentiert sein
 - Nutzung der API einer Library
 - Nutzung von SOAP Services
 - Nutzung von REST Services

- Beispiel: Google vs. Ebay Services

- **Service-/Funktions-orientiert**

Der Schwerpunkt liegt analog zur Programmierung auf aufzurufenden Operationen und zu übergebenen typgesicherten Parameter-Objekten (RPC, Remote Procedure Call).

- **Nachrichtenorientiert**

Der Schwerpunkt liegt auf auszutauschenden Nachrichten/Dokumenten. Die Kommunikation erfolgt bevorzugt asynchron, zum Beispiel indirekt über vermittelnde MOM-Systeme.

- **Ressourcenorientiert**

Besser bekannt unter REST (Representational State Transfer). Der Schwerpunkt liegt analog dem Web und seinem HTTP-Protokoll auf per URI (z.B. URL) identifizierbaren Ressourcen.

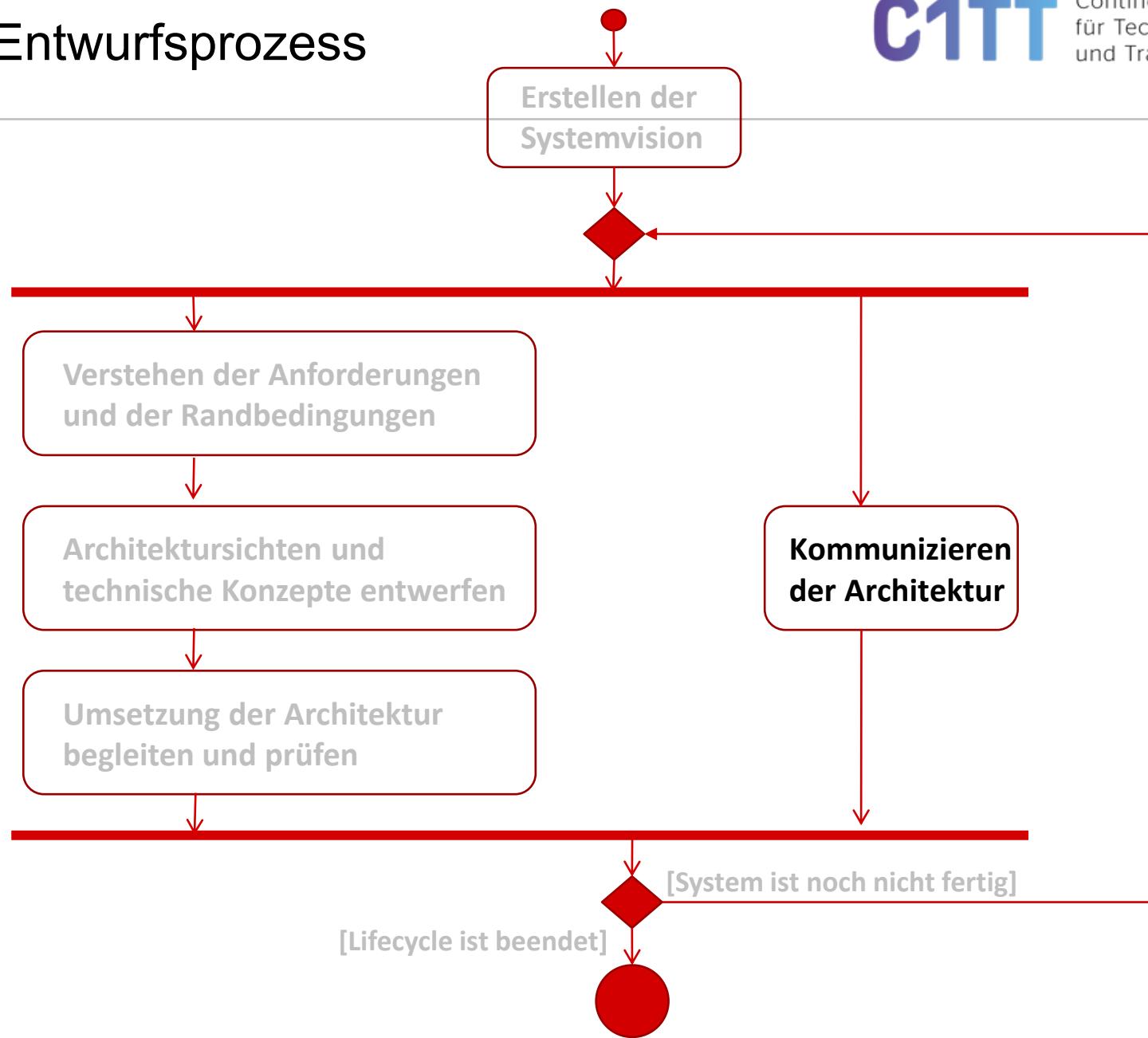
- Paper "On the role of scientific thought" transcribed by Richard Walker
<http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html>
- Program Development by Stepwise Refinement
http://oberoncore.ru/_media/library/wirth_program_development_by_stepwise_refinement2.pdf
- On the Criteria to be Used in Decomposing Software into Modules
<https://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf>
- [BWSW2014] M.Gharbi, A.Koschel, A.Rausch, G.Starke
Basiswissen für Softwarearchitekten (2. Auflage) ISBN: 978-3-86490-165-2
- [SWP2012] L.Bass, P.Clements, R.Kazman
Software Architecture in Practice (3rd Edition) ISBN: 978-0321815736
- [SAP2015] Mark Richards
Software Architecture Patterns, ISBN 978-1-491-92424-2
- (Ghezzi, Jazayeri & Mandrioli, 2003) Fundamentals of Software Engineering

- [TOH2014] Vorgehensmuster für Softwarearchitektur
ISBN: 978-3-446-43615-2
- [Martin, 2008] Clean Code
ISBN: 978-0132350884
- [Mayer, 1997] Object-Oriented Software Construction
- [Sommerville, 2015] Software Engineering
ISBN: 978-0133943030
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-orientierte Softwarearchitektur.*
- [Martin, Robert C. (2003)]. Agile Software Development, Principles, Patterns, and Practices. Prentice Hall. pp. 127–131. ISBN 978-0135974445.

Dokumentation

von Software Architekturen

Der Entwurfsprozess



- **Qualitätsmerkmale technischer Dokumentation**
- Softwarearchitekturen Stakeholder-gerecht beschreiben
- Architektursichten erläutern
- Schnittstellen beschreiben
- Architekturentscheidungen erläutern
- Weitere Hilfsmittel und Werkzeuge

- Zielorientiert
 - Die wenigsten Benutzer interessieren sich dafür, wie Ihr Produkt funktioniert – auch wenn Sie als Entwickler zu Recht stolz darauf sind. Benutzer wollen ganz einfach schnell und unkompliziert zum Ziel kommen und lesen Benutzerhandbücher und Online-Hilfen nur dann, wenn Sie ein konkretes Problem vor Augen haben. Dieses Problem müssen sie in der Dokumentation wiederfinden.
 - Entspricht der Aufbau der Dokumentation der technischen Architektur Ihres Produkts (= schlecht) oder spiegeln die Themen die Aufgaben und Ziele der Benutzer wider (= gut)?
 - Enthält die Dokumentation das, was Sie sagen möchten (= schlecht) oder das, was die Benutzer wissen wollen (= gut)?
 - Wird beschrieben, was der Benutzer ohnehin am Produkt sieht (= schlecht), oder das, was er nicht sieht (= gut)? Wird das Produkt beschrieben (= schlecht) oder seine Anwendung (= gut)?

- Übersichtlichkeit
 - Ist die Gliederung flach, einfach und übersichtlich? Oder gibt es Kapitel, Unterkapitel, Unterunterkapitel, Unterunterunterkapitel, ...?
 - Entspricht die Reihenfolge der Kapitel der Häufigkeit, in der die Kapitel benötigt werden? Steht das Wichtigere vor dem Unwichtigeren, das Allgemeine vor dem Speziellen?
- Gezielter Zugang
 - Ist der Text klar durch Zwischenüberschriften untergliedert? Oder gibt es lange unübersichtliche „Textwüsten“?
 - Wird in jedem Absatz nur ein Thema behandelt oder sind die Inhalte wild vermischt?
 - Sind Handlungsanleitungen nummeriert und klar als solche erkennbar?
 - Können Benutzer gezielt das lesen, was sie in einer bestimmten Situation interessiert: Einführung, schrittweise Anleitung oder Detailinformation? Oder sind die Informationstypen bunt gemischt, so dass immer alles gelesen werden muss?

- Mediengerechte Nutzung
 - Wie eng und mediengerecht ist die Anbindung einer Online-Dokumentation an die Software? Lässt sich lediglich eine PDF-Datei aufrufen? Oder ist die Online-Hilfe kontextsensitiv oder sogar direkt in die Software-Oberfläche integriert („embedded“)?
 - Ist eine Online-Hilfe lediglich ein „Handbuch am Bildschirm“, oder nutzt sie tatsächlich die Möglichkeiten des Online-Mediums, wie Interaktion, Animation, Personalisierung?
- Benutzbarkeit
 - Enthält jedes Topic einer Online-Hilfe sinnvolle Informationen, oder gibt es weitgehend inhaltsleere Topics, die die Benutzer lediglich weiter verweisen und „von Pontius zu Pilatus schicken“?
 - Existiert ein redaktionell bearbeiteter Index oder nur ein automatisch generierter Index?

- Ansprache und Eindeutigkeit
 - Vermittelt der Text Vertrauen, Zuversicht und Sicherheit, oder wirkt er eher einschüchternd?
 - Werden die Leser direkt angesprochen (Beispiel: „Klicken Sie ...“) oder gibt es unpersönliche Formulierungen, bei denen unklar bleibt, wer handelt (Beispiel: „Es muss ... durchgeführt werden“)?
 - Gibt es unpräzise Formulierungen, aus denen nicht eindeutig hervorgeht, ob zwingend gehandelt werden muss, oder ob optional gehandelt werden kann („eventuell“, „sollte“)?
 - Sind alle Überschriften eindeutig formuliert, so dass sich bereits im Vorfeld der Inhalt eines Themas eindeutig erkennen lässt? Oder muss der Benutzer jedes Thema zunächst einmal anlesen, um zu sehen, ob der Inhalt für ihn überhaupt relevant ist? (Beispiel: Ein Thema heißt lediglich „Überblick“. Überblick über was?)
 - Sind alle Links so eindeutig formuliert, dass die Benutzer bereits vor dem Anklicken erkennen können, ob es sich lohnt, ihnen zu folgen?

- Strukturiertheit
 - Machen Struktur und Formulierung den Lesern das Finden und Verstehen der gebotenen Informationen leicht? Ist es möglich, auf einzelne Informationen auch punktuell zuzugreifen?
 - Steht die wichtigste Information am Anfang?
 - Steht die Information, die alle Benutzer brauchen vor der Information, die nur wenige Benutzer interessiert?
 - Stehen häufig benötigte Informationen vor seltener benötigten Informationen?
 - Steht inhaltlich Zusammengehöriges auch räumlich zusammen?
 - Stehen die Schlüsselbegriffe im Satz möglichst weit vorne?
 - Sind die Sätze kurz und einfach? Können auch Nicht-Muttersprachler den Text verstehen?

- Änderungsfreundlichkeit, Übersetzungsfreundlichkeit
 - Wird die Dokumentation mit einem geeigneten Autorensystem erstellt oder „von Hand zusammengestrickt“? Kann die Dokumentation mit wenig Aufwand überarbeitet und neu produziert werden?
 - Sind Autorensystem und Format zukunftssicher?
 - Wird beim Schreiben auf Lokalisierbarkeit und Übersetzungsfreundlichkeit geachtet?
 - Unterstützen Format, Autorensystem und Dokumentaufbau Single Source Publishing, also die zukünftige Ausgabe in weiteren Ausgabemedien, Formaten und Versionen?

- Qualitätsmerkmale technischer Dokumentation
- **Softwarearchitekturen Stakeholder-gerecht beschreiben**
- Architektursichten erläutern
- Schnittstellen beschreiben
- Architekturentscheidungen erläutern
- Weitere Hilfsmittel und Werkzeuge

- Selbst die beste Architektur ist nutzlos wenn die Personen die sie brauchen
 - Nicht wissen worum es sich dabei handelt
 - Nicht ausreichend verstehen, wie die Architektur eingesetzt oder modifiziert werden kann
 - Die Architektur missverstehen und inkorrekt anwenden
- Der gesamte Aufwand für Analyse, Ihre harte Arbeit und das brillante Design welches das Architekturteam geleistet und erstellt hat wird verschwendet.

- Architekturdokumentation muss ...
 - ausreichend transparent und zugänglich sein, um von neuen Mitarbeitern schnell verstanden zu werden
 - hinreichend konkrete Anweisungen enthalten, um als Blueprint für die Erstellung zu dienen
 - genügend Information enthalten, um sie als Basis für Analysen zu verwenden
- Architekturdokumentation ist vorschreibend und beschreibend
 - Für die einen schreibt es vor was wie sein SOLL, und einschränkend welche weiteren Entscheidungen möglich sind
 - Für andere wiederum beschreibt es was IST, 'erzählend' welche Entscheidungen bereits im Systemdesign getroffen wurden
- Das Verstehen WIE Stakeholder die Dokumentation verwenden ist extrem wichtig
- Anhand dieser Benutzungen wird bestimmt welche Informationen in der Dokumentation aufgenommen werden

Ausbildung

- Neue Teammitglieder
- Externe Analysten
- Neue Architekten

Primärer Kanal für Kommunikation zwischen Stakeholder

- Architekt mit Entwickler
- Architekt mit künftigen Architekten
- Architekt mit Security

Basis zur Systemanalyse und Aufbau

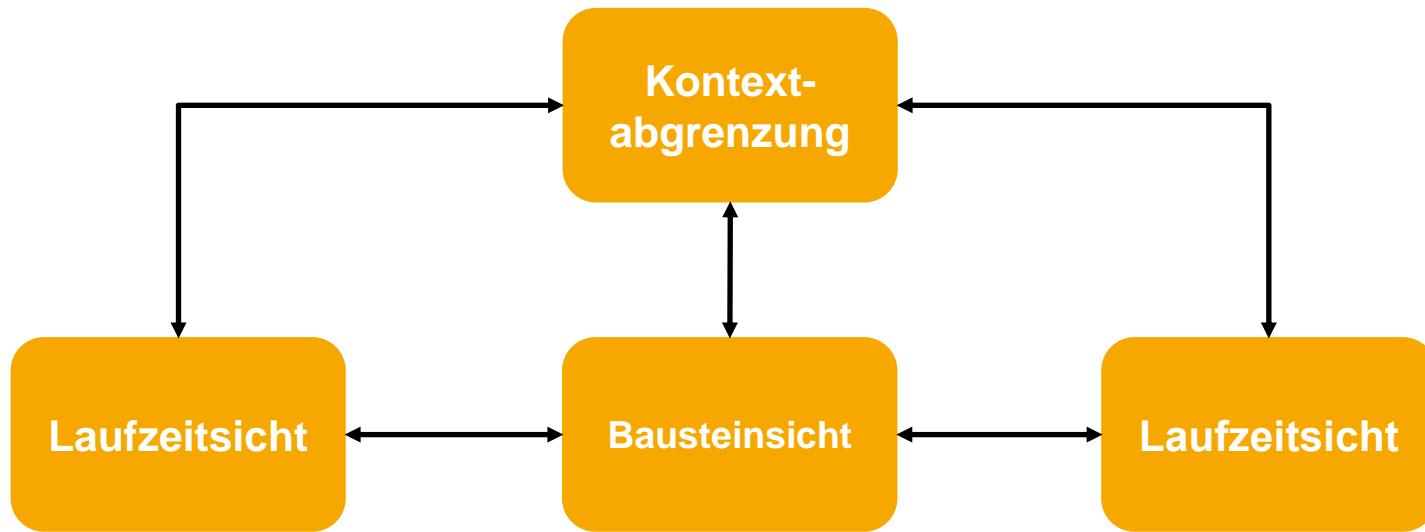
- Architektur sagt Entwicklern was zu entwickeln ist.
- Jedes Modul hat Schnittstellen die bereitgestellt werden müssen und benutzt Schnittstellen von anderen Modulen.
- Die Dokumentation kann als Kanal für die Erfassung und Kommunikation zu ungelösten Problemen dienen.
- Architektur Dokumentation dient als Basis für die Evaluierung der Architektur.

- Kompromisse (Trade-Off)
 - Normalerweise sind eher formale Darstellungen zeitintensiver und aufwändiger in der Erstellung und im Verstehen, haben aber weniger Unklarheiten und bieten mehr Möglichkeiten für die Analyse. (ADL Architecture Description Language)
 - Im Gegenzug sind eher informelle Schreibweisen leichter zu erstellen, bieten aber weniger Funktionssicherheit. (UML)
- Unterschiedliche Schreibweisen sind besser um verschiedene Arten der Information zu transportieren.
 - UML Klassendiagramme werden Ihnen bei der Planbarkeit nicht helfen und Laufzeitdiagramme liefern kaum Informationen zum Roll-out.
 - Wählen sie die Darstellungen nach den jeweiligen Erfordernissen die Sie erfassen und erklären müssen.

- Qualitätsmerkmale technischer Dokumentation
- Softwarearchitekturen Stakeholder-gerecht beschreiben
- **Architektursichten erläutern**
- Schnittstellen beschreiben
- Architekturentscheidungen erläutern
- Weitere Hilfsmittel und Werkzeuge

- Mittels Sichten können Sie einer Softwarearchitektur die verschiedene Komplexitäten für unterschiedliche Stakeholder darstellen.
- Prinzipien der Architekturdokumentation:
 - *Bei der Dokumentierung einer Architektur geht es um die Dokumentierung der relevantesten Sichten und anschließend um die Dokumentation weiterer Sichten.*
- Dafür gibt es Sichtenmodelle
 - 4+1 Sichtenmodell nach Philippe Kruchten
 - 4 Sichtenmodell nach Gernot Starke
 - Siemens Sichtenmodell nach Hofmeister

4 Sichtenmodell nach Gernot Starke



- Kontextabgrenzung
 - Einbettung des Systems in seine Umgebung (Nachbarsysteme, Stakeholder, Infrastruktur)
 - System als Blackbox
 - Sehr abstrahiert
- Bausteinsicht
 - Aufbau des Systems aus Subsystemen, Komponenten, Teilkästen, Frameworks, Konfigurationen, ...
 - Zusammenwirken der Bausteine (Schnittstellen)
 - Top-Down mit Blackboxen und Whiteboxen
- Laufzeitsicht
 - Dynamische Struktur
 - Interaktion von Laufzeitinstanzen
- Verteilungssicht
 - Technische Ablaufumgebung
 - Hardwarekomponenten und ihr Zusammenspiel
 - Deployment-Einheiten

Diagramm	Sicht
Komponentendiagramm	Kontextabgrenzung
Klassendiagramm, Paketdiagramm, Komponentendiagramm	Bausteinsicht
Aktivitätsdiagramm, Zustandsdiagramm	Laufzeitsicht – interne Steuerung
Sequenzdiagramm	Laufzeitsicht – externe Steuerung
Verteilungsdiagramm	Verteilungssicht
Kommunikationsdiagramm, Zeitverhaltensdiagramm	Meist in der embedded Entwicklung im Einsatz

Die Kontextabgrenzung zeigt

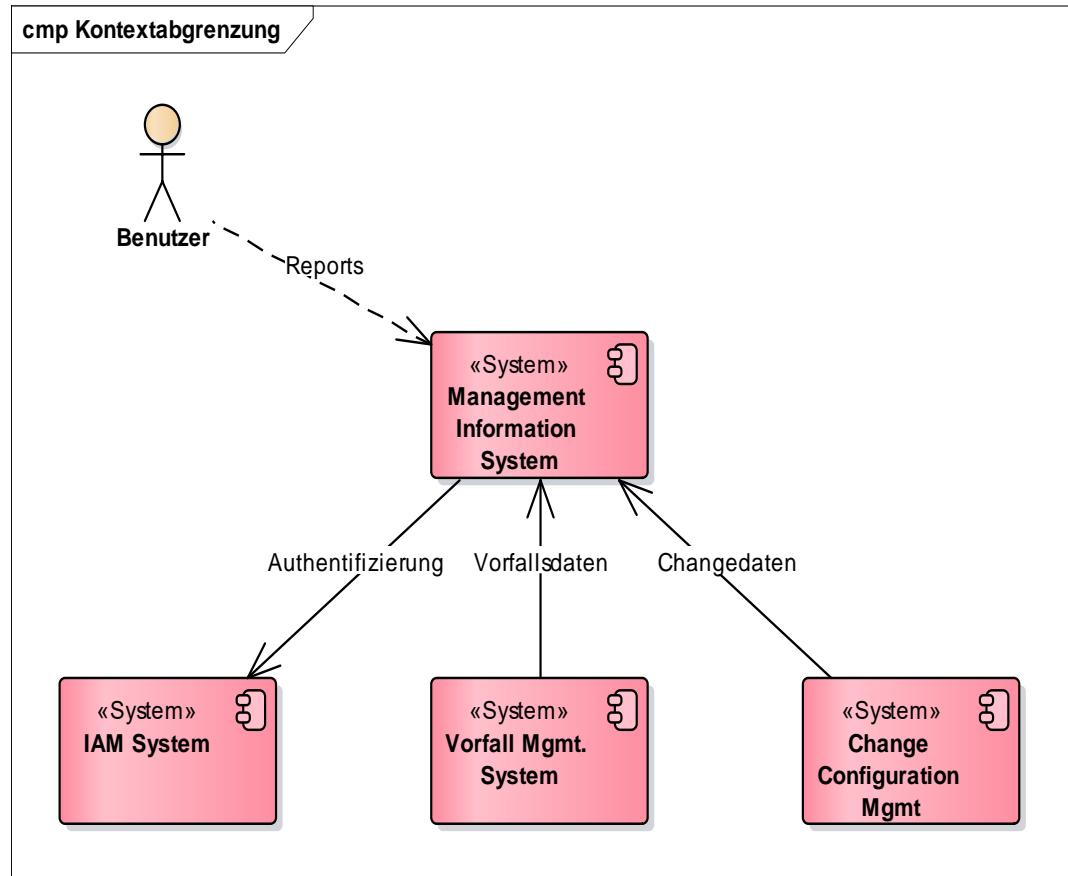


- das System als Black-Box, d.h. in einer Sicht von außen
- die Schnittstellen zur Außenwelt, zu Anwendern, Betreibern und Fremdsystemen, inklusive der über diese Schnittstellen transportierten Daten oder Ressourcen
- die wichtigsten Anwendungsfälle (Use Cases) des gesamten Systems
- die technische Systemumgebung, Prozessoren, Kommunikationskanäle

Die Kontextabgrenzung stellt also eine Abstraktion der übrigen Sichten dar und dient allen als Einstiegspunkt in das System

Kontextabgrenzungen zur Kommunikation

- Erläutern Sie Auftraggebern oder Kunden, wie das System die Anforderungen erfüllen soll
- Weisen Sie auf mögliche Risikofaktoren hin
- Fragen Sie Entwickler nach ihrer Meinung
- Sprechen Sie mit den zukünftigen Betreibern
- Lassen Sie die Ergebnisse dieser Gespräche in die Entwürfe einfließen.



Was zeigt die Bausteinsicht?

- Die Bausteinsicht bildet die Aufgaben des Systems auf Software-Bausteine oder -Komponenten ab
- Diese Sicht macht Struktur und Zusammenhänge zwischen den Bausteinen der Architektur explizit
- Bausteinsichten zeigen statische Aspekte von Systemen
- In der Bausteinsicht sollten Sie Funktionalitäten und nichtfunktionale Anforderungen auf Architekturbausteine abbilden



Die Bausteinsicht beantwortet folgende Fragen:

- Aus welchen Komponenten, Paketen, Klassen, Subsystemen oder Partitionen besteht das System?
- Welche Abhängigkeiten bestehen zwischen diesen Bausteinen?
- Welche Bausteine müssen Sie implementieren, konfigurieren oder kaufen, um die gewünschten Anforderungen zu erfüllen?

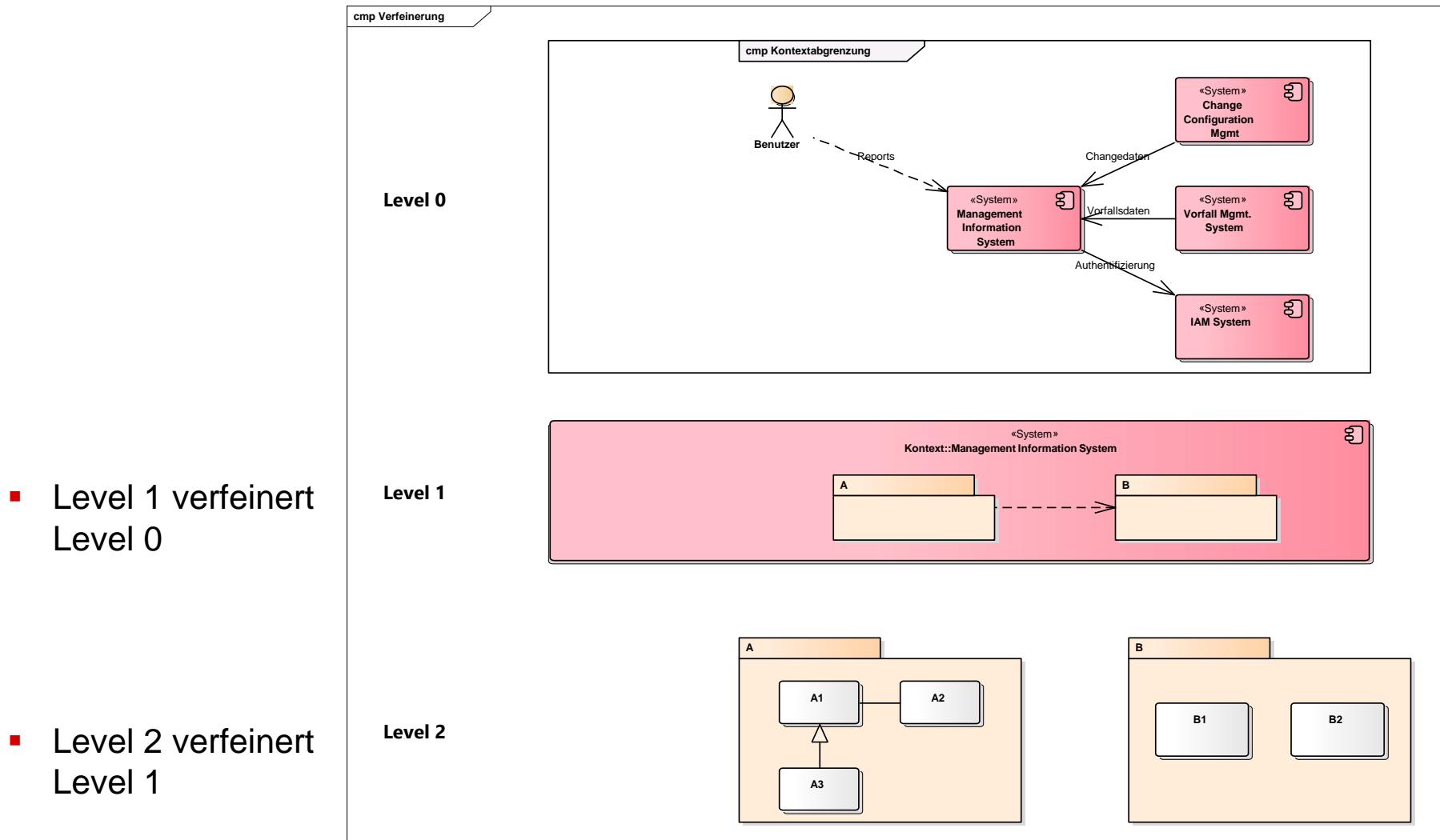
Bausteinsichten zeigen zwei verschiedene Arten von Abstraktionen, nämlich Black- und Whiteboxes

- Blackboxes sind ausschließlich durch ihre externen Schnittstellen und ihre Funktionalität beschrieben. Sie folgen dem Geheimnisprinzip
- Whiteboxes sind geöffnete Blackboxes: sie zeigen deren innere Struktur und Arbeitsweise. Whiteboxes bestehen ihrerseits wiederum aus Blackboxes

Darstellung

- Sie können Blackboxes weiter verfeinern, indem Sie ihren „Inhalt“ als Whitebox zeigen
- Hier stellen Sie Bausteine in Hierarchien oder Architekturebenen („Levels“) dar, wobei Sie den Detaillierungsgrad der Sichten schrittweise verfeinern

Bausteinsicht – Verfeinerung



- **Klassensymbole**

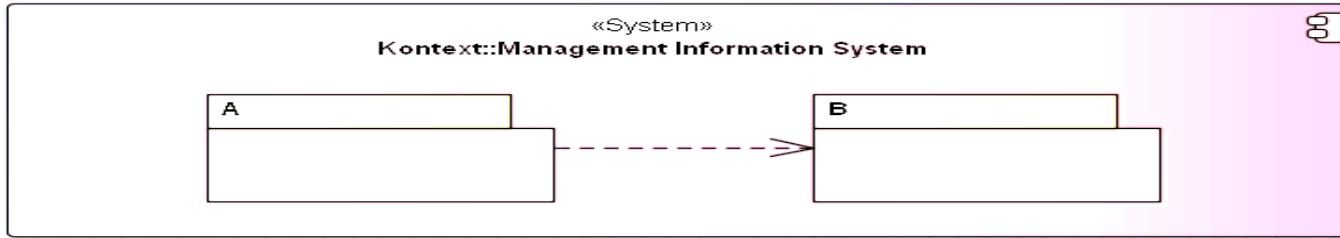
bezeichnen einzelne Bausteine. Das können Klassen einer objektorientierten Programmiersprache sein, aber auch alle anderen ausführbaren Software-Einheiten (Funktionen, Prozeduren, Programme)

- **Komponenten**

bezeichnen ebenfalls einzelne Bausteine, bieten jedoch die Möglichkeit, die ein- und ausgehenden Schnittstellen genau zu beschreiben

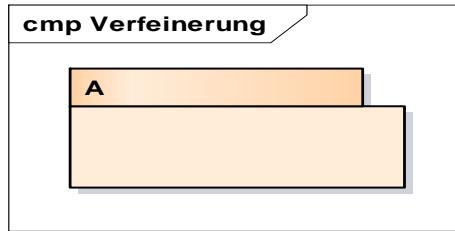
- **Pakete**

stehen für Gruppen, Mengen oder Strukturen von Bausteinen. Diese Symbole zeigen, dass es sich um Abstraktionen handelt (die in der Architektur oder im detaillierten Entwurf weiter verfeinert werden)



Überschrift	Inhalt
Übersichtsdiagramm	Ein Diagramm (UML Paket- oder Klassendiagramm), das die innere Struktur dieser Whitebox zeigt
Lokale Bausteine	Tabelle oder Liste der lokalen Blackbox-Bausteine
Lokale Beziehungen	Tabelle oder Liste der Abhängigkeiten und Beziehungen zwischen den lokalen Bausteinen
Entwurfsentscheidungen	Gründe, die zu dieser Struktur geführt haben

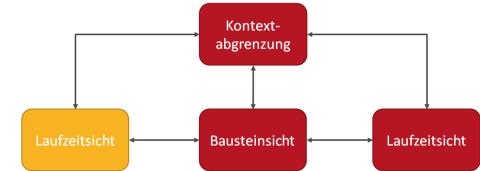
Beschreibung von Blackboxes



Überschrift	Inhalt
Zweck / Verantwortlichkeit	In knappen Worten beschrieben
Schnittstellen	Die eingehenden und ausgehenden Schnittstellen
Erfüllte Anforderungen	Die damit verbundenen funkt. und nicht funkt Anforderungen
Variabilität	Mögliche Variabilität, erwartete Änderungen oder auch Konfigurationsmöglichkeiten
Leistungsmerkmale	Verfügbarkeit
Ablageort / Datei	Weitere Informationen
Sonstige Verwaltungsinformation	Autor, Datum, letzte Änderung
Offene Punkte	

- Bevor Sie sich an den Entwurf der Bausteinsicht begeben, prüfen Sie, ob Sie Teile dieses Systems aus anderen Quellen wieder verwenden können
- Suchen Sie dabei auch an den Stellen, die nicht unbedingt zum Kern Ihres konkreten Systems gehören
- Falls Sie eine bestimmte benötigte Semantik mit den Symbolen der UML nicht direkt ausdrücken können, benutzen Sie Notizen
- Beschreiben Sie kritische Schnittstellen, etwa zu Nachbarsystemen, möglichst früh im Projekt
- Berücksichtigen Einflussfaktoren und Randbedingungen
 - die technische Infrastruktur
 - die technischen Einflussfaktoren
 - die organisatorischen Einflussfaktoren
 - die Qualitätsanforderungen

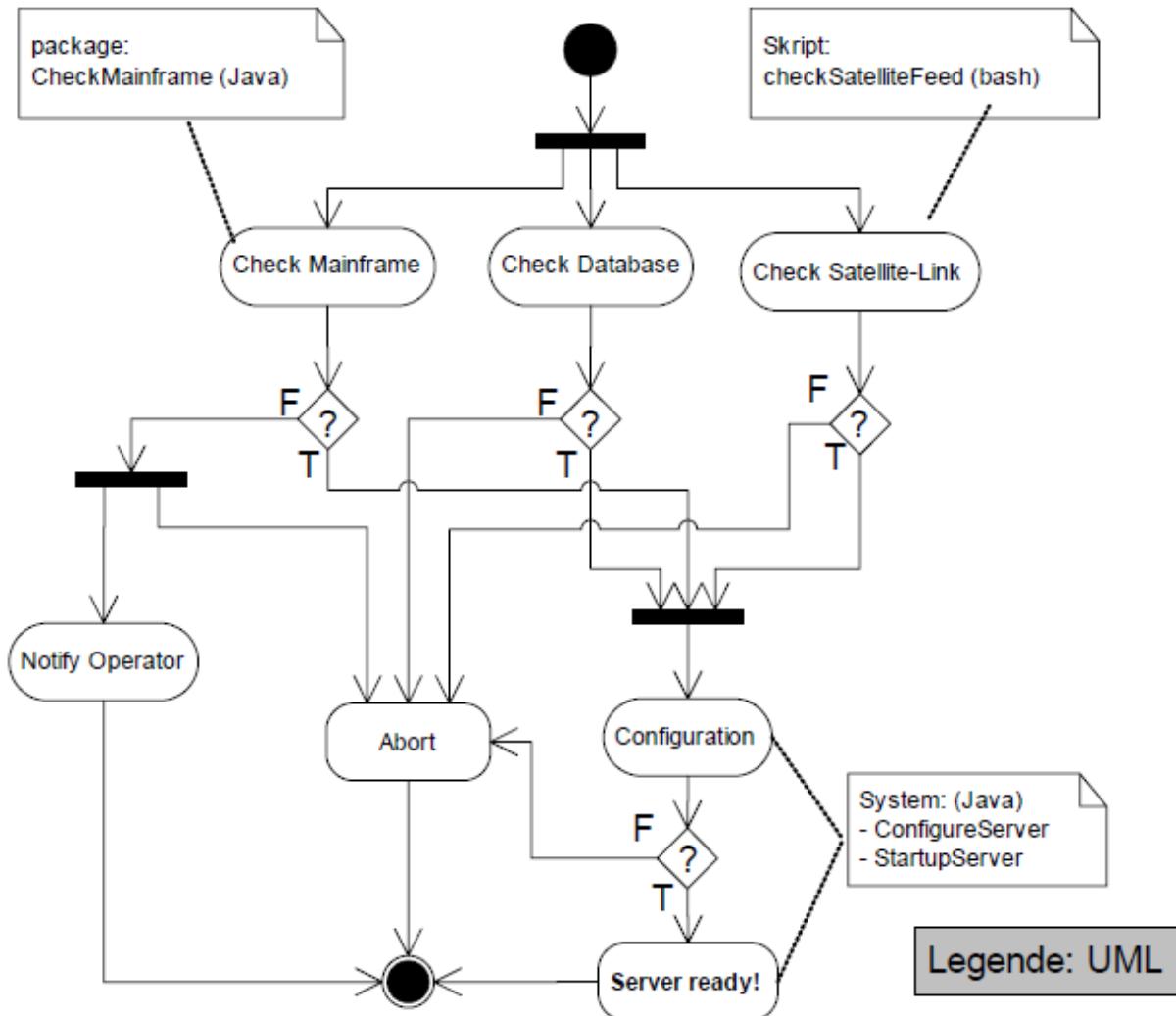
- Die Laufzeitsicht beschreibt, welche Bestandteile des Systems zur Laufzeit existieren und wie diese zusammenwirken
- Darüber hinaus dokumentiert die Laufzeitsicht, wie Laufzeitkomponenten sich aus Instanzen von Implementierungsbausteinen ergeben
- Für eingebettete Systeme oder Echtzeitsysteme ist es darüber hinaus wichtig, auch die Abbildung auf Prozesse, Tasks oder Threads zu beschreiben

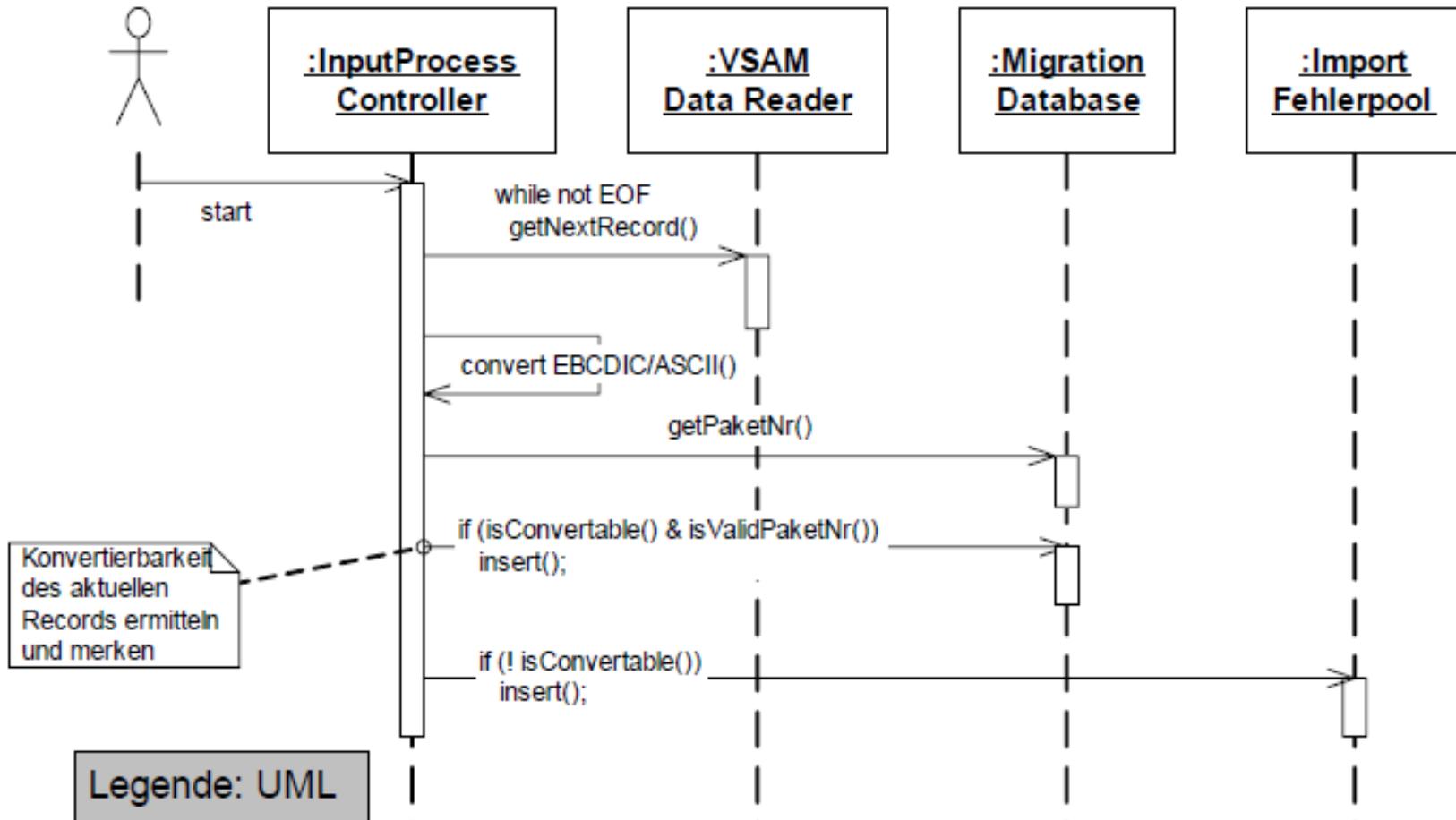


Die Laufzeitsicht beantwortet folgende Fragen:

- Wie arbeiten die Systemkomponenten zur Laufzeit zusammen?
- Wie werden die wichtigsten Use-Cases durch die Architekturbausteine bearbeitet?
- Welche Instanzen von Architekturbausteinen gibt es zur Laufzeit, und wie werden diese gestartet, überwacht und beendet?
- Wie arbeiten Systemkomponenten mit externen und vorhandenen Komponenten zusammen?
- Wie startet das System?

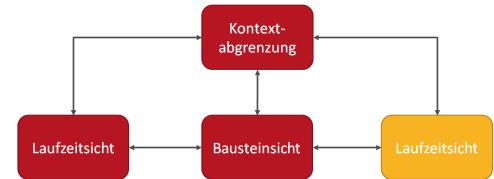
Laufzeitsicht – Notation: Aktivitätsdiagramm





- Zur Beschreibung von Abläufen können Sie auch nummerierte Listen oder Quellcode verwenden, sofern er für Ihre Leser gut verständlich ist und
- Beachten Sie bei der Dokumentation der Laufzeitsicht auch die Informationsbedürfnisse der Betreiber und Administratoren des Systems, nicht nur die von Managern und Entwicklern

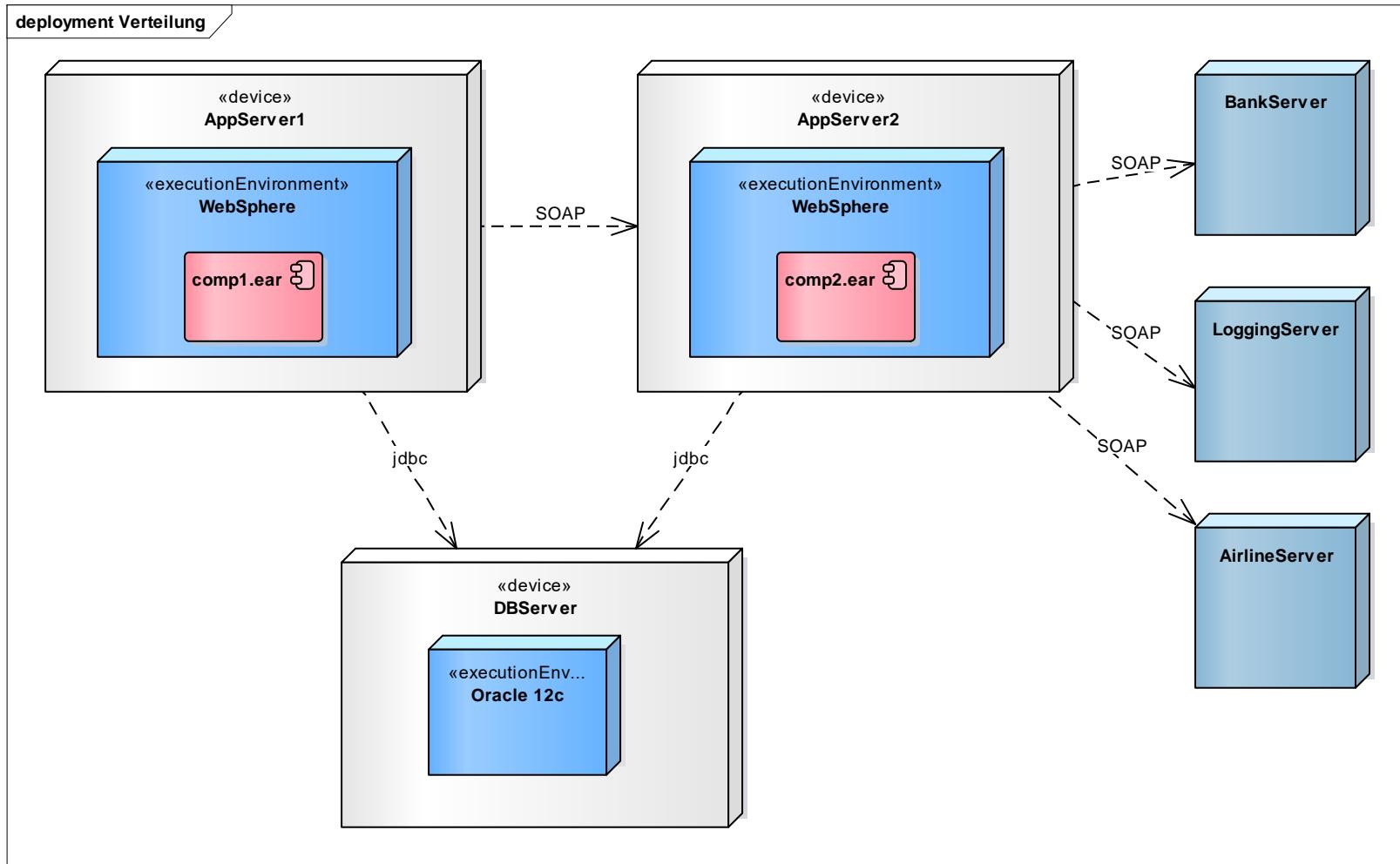
- In Verteilungssichten (oder Infrastruktursichten) beschreiben Sie die Ablaufumgebung des Systems in Form von Hardwarekomponenten mit den beteiligten Protokollen



Elemente der Verteilungssicht

- Bestandteile der technischen Infrastruktur, so genannte „Knoten“
- Laufzeitelemente, auch „Laufzeitartefakte“ genannt
- Kanäle sind Verbindungen zwischen Knoten (physische Kanäle) beziehungsweise zwischen Laufzeitelementen (logische Kanäle)

Verteilungssicht - Notation



- Ein Modell der Infrastruktursicht sollte eine Landkarte der beteiligten Hardware und der externen Systeme sein
- Gleichen Sie die Infrastruktursicht mit den Mengengerüsten der beteiligten Daten ab
- Sie können Mengengerüste und Datenvolumina mit der Infrastruktursicht überlagern?
- Falls Ihre Systeme in verteilten oder heterogenen Umgebungen ablaufen, sollten Sie vorhandene Kommunikationsmechanismen, Protokolle und Middleware in die Infrastruktursicht aufnehmen
- In heterogenen und verteilten Systemlandschaften sollten Sie eine detaillierte Sicht der technischen Infrastruktur erstellen
- Berücksichtigen Sie die realen Verfügbarkeiten der technischen Komponenten
- Oftmals ist die „Verpackung“ von Implementierungsbausteinen in Deployment-Artefakte eine aufwändige oder komplexe Aufgabe, die hoffentlich Ihr Build-Prozess automatisiert. Verweisen auf Ihre Build-Skripte

- Es können weitere Sichten nach eigenen Bedürfnissen erstellt werden
- z.B.: Eigene Bausteinaspkte als Sichttypen (Datensicht)

- Bedenken Sie jedoch:
 - den Erstellungs- und Wartungsaufwand
 - die eventuell zusätzliche, detaillierte Erläuterung der gesonderten Symbolik
 - vier der vorgestellte Sichtarten meist ausreichend

- Qualitätsmerkmale technischer Dokumentation
- Softwarearchitekturen Stakeholder-gerecht beschreiben
- Architektursichten erläutern
- **Schnittstellen beschreiben**
- Architekturentscheidungen erläutern
- Weitere Hilfsmittel und Werkzeuge

- Schnittstellen besitzen als Abgrenzungen zwischen den verschiedenen Bausteinen von Systemen besondere Bedeutung:
Über Schnittstellen arbeiten diese zusammen und erzeugen letztlich den Mehrwert des Gesamtsystems
- Manche Schnittstellen basieren auf Standard Konstrukten der eingesetzten Programmiersprachen
- Viele Schnittstellen werden bereits als Bestandteile von Sichten dokumentiert, insbesondere in den Bausteinsichten
- Häufig besteht seitens einiger Projektbeteiligter der Bedarf nach einer eigenständigen Schnittstellendokumentation

Überschrift	Inhalt
Identifikation	Genaue Bezeichnung und Version der Schnittstelle
Bereitgestellte Ressourcen	Welche Ressourcen stellt dieses Element für Ihre Akteure (Benutzern, Aufrufern) bereit? Syntax der Ressource Semantik der Ressource Restriktionen bei der Benutzung
Fehlerszenarien	Beschreiben Sie sowohl mögliche Fehlersituationen als auch deren Behandlung.
Variabilitäten und Konfigurierbarkeit	Kann das Verhalten der Schnittstelle oder der Ressourcen verändert oder konfiguriert werden?
Qualitätseigenschaften	Welche Qualitätseigenschaften wie z.B. Verfügbarkeit, Performance, Sicherheit sind für diese Schnittstelle gültig?
Entwurfsentscheidungen	Welche Gründe haben zum Entwurf dieser Schnittstelle geführt? Welche Alternativen gibt es, und warum wurden diese verworfen?
Benutzungshinweise	Hinweise oder Beispiele zur Benutzung dieser Schnittstelle

- Qualitätsmerkmale technischer Dokumentation
- Softwarearchitekturen Stakeholder-gerecht beschreiben
- Architektursichten erläutern
- Schnittstellen beschreiben
- **Architekturentscheidungen erläutern**
- Weitere Hilfsmittel und Werkzeuge

Dokumentieren Sie Entwurfsentscheidungen

- Dokumentieren Sie hier alle wesentlichen Entwurfsentscheidungen und deren Gründe!

Motivation

- Es ist wünschenswert, alle wichtigen Entwurfsentscheidungen geschlossen nachlesen zu können. Wägen Sie ab, inwiefern Entwurfsentscheidungen hier zentral dokumentiert werden sollen oder wo eine lokale Beschreibung (z.B in der Whitebox-Sicht von Bausteinen) sinnvoller ist. Vermeiden Sie aber redundante Texte.

Form

- informelle Liste, möglichst nach Wichtigkeit und Tragweite der Entscheidungen für den Leser aufgebaut.

Szenarien beschreiben, was beim Eintreffen eines Stimulus auf ein System in bestimmten Situationen geschieht.

- Nutzungsszenarien (auch genannt *Anwendungs- oder Anwendungsfallszenarien*) beschreiben, wie das System zur Laufzeit auf einen bestimmten Auslöser reagieren soll. (Performance)
- Änderungsszenarien beschreiben eine Modifikation des Systems oder seiner unmittelbarer Umgebung. (zusätzliche Funktionalität)
- Grenz- oder Stress-Szenarien beschreiben, wie das System auf Extremsituationen reagiert. (z.B. vollständiger Stromausfall)

- **Zentrale Architekturbeschreibung:**
ist (sinnvollerweise) das Kerdokument für eine Softwarearchitektur. Sie enthält, soweit möglich, alle architekturrelevanten Informationen (Referenz ARC42)
- **Architekturüberblick:**
dient als schnell lesbare Kurzfassung (möglichst nicht mehr als 30 Seiten) der zentralen Architekturbeschreibung.
- **Dokumentationsübersicht:**
Das Verzeichnis ist als Index für alle architekturrelevanten Dokumente zu sehen und nennt auch deren Abhängigkeiten.
- **Übersichtspräsentation:**
Foliensatz, anhand dessen die Architektur in maximal einer Stunde (technisch) präsentiert werden kann.

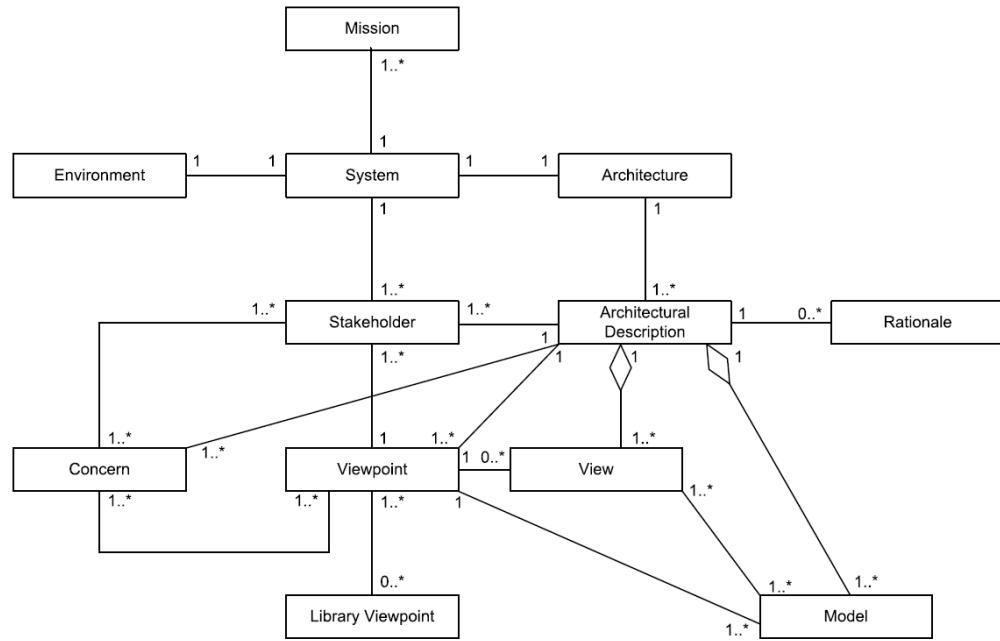
- **Architekturtapete:**
Mit einer »Architekturtapete« sollen viele Architekturaspekte in einem Gesamtüberblick dargestellt werden.
- **Handbuch zur Architekturdokumentation:**
erläutert die Funktionsweise und die Struktur der gesamten Dokumentation.
- **Technische Informationen:**
ein oder mehrere Dokumente mit wichtigen technischen Informationen
- **Dokumentation von externen Schnittstellen:**
Diese sind für das Zusammenwirken des Gesamtsystems in seinem Kontext von zentraler Bedeutung.

- Qualitätsmerkmale technischer Dokumentation
- Softwarearchitekturen Stakeholder-gerecht beschreiben
- Architektursichten erläutern
- Schnittstellen beschreiben
- Architekturentscheidungen erläutern
- **Weitere Hilfsmittel und Werkzeuge**

Frameworks für Architektur Dokumentation

ISO 42010

- Hier werden Architekturen mit Hilfe von Modellen beschrieben. Diese Modelle sind durch Abstraktion gebildet und zeigen das Grundgerüst einer Systemstruktur.
- Mit dem ISO–Standard wird ein formaler und konzeptueller Rahmen für Architekturbeschreibungen zur Verfügung gestellt.
- Zudem werden damit Grundprinzipien der Architekturgestaltung definiert.



Introduction

- What is ISO/IEC/IEEE 42010?
- An international standard, entitled *Systems and software engineering—Architecture description*
- Published December 2011
- Based on IEEE Std 1471:2000
- Specifies *best practices* for documenting enterprise, system and software architectures



Arc42

- Kostenfreies Template von Gernot Starke und Peter Hruschka
- Basiert auf praktischer Erfahrung
- Gliedert die Architektur in eine flexible Informationsstruktur
- Sie finden darin eine vollständige, strukturierte und vernetzte Sammlung architektureller Themen

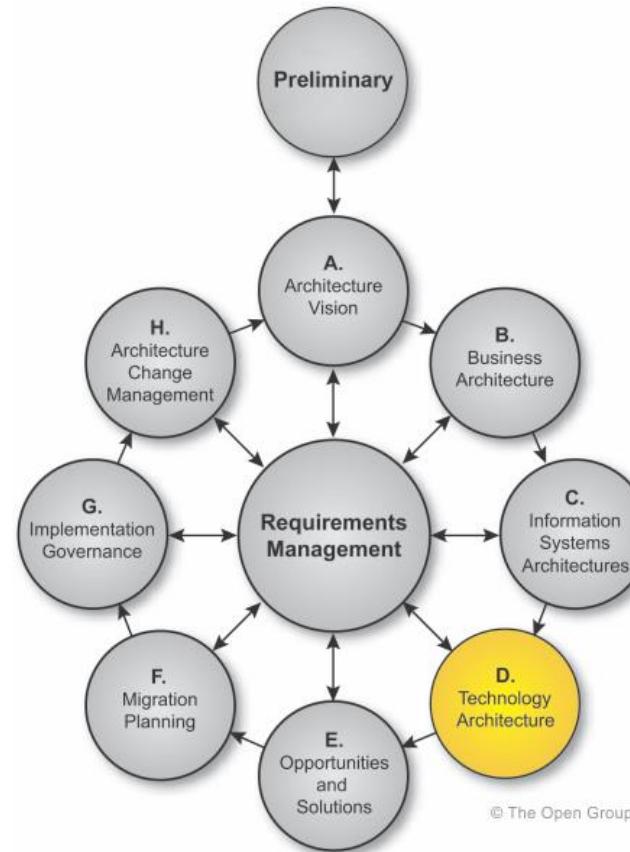
1. Einführung und Ziele Aufgabenstellung, Qualitätsziele, eine Kurzfassung der architekturellen Anforderungen (insb. die nichtfunktionalen), Stakeholder.
2. Randbedingungen Welche Leitplanken schränken die Entwurfsentscheidungen ein?
3. Kontextabgrenzung In welchem fachlichen und/oder technischen Umfeld arbeitet das System?
4. Lösungsstrategie Wie funktioniert die Lösung? Was sind die fundamentalen Lösungsansätze?
5. Bausteinsicht Die statische Struktur des Systems, der Aufbau aus Implementierungsteilen.
6. Laufzeitsicht Zusammenwirken der Bausteine zur Laufzeit, gezeigt an exemplarischen Abläufen ("Szenarien")
7. Verteilungssicht Deployment: Auf welcher Hardware werden die Bausteine betrieben?
8. Querschnittliche Konzepte und Muster Wiederkehrende Muster und Strukturen. Fachliche Strukturen. Querschnittliche, übergreifende Konzepte, Nutzungs- oder Einsatzanleitungen für Technologien. Oftmals projekt-/systemübergreifend verwendbar!
9. Entwurfsentscheidungen Zentrale, prägende und wichtige Entscheidungen.
10. Qualitätsszenarien Qualitätsbaum sowie dessen Konkretisierung durch Szenarien
11. Risiken
12. Glossar Wichtige Begriffe.

Legende:

anforderungsbezogene Informationen	Strukturen der Lösung (Sichten)	übergreifende (technische) Informationen	besonders wichtige Entscheidungen
------------------------------------	---------------------------------	--	-----------------------------------

TOGAF

- Das The Open Group Architecture Framework bietet einen Ansatz für Entwurf, Planung, Implementierung und Wartung von Unternehmensarchitekturen
- Es beschreibt ebenfalls Ansätze zur Architekturdokumentation
- Die Dokumentation ist in den Prinzipien von TOGAF aufgenommen worden



- [STARKE2014] G.Starke
Effektive Software-Architekturen (6.Auflage) ISBN: 978-3-446-43614-5
- [BWSW2014] M.Gharbi, A.Koschel, A.Rausch, G.Starke
Basiswissen für Softwarearchitekten (2. Auflage) ISBN: 978-3-86490-165-2
- [ZOERN2012] Stefan Zörner, Softwarearchitekturen dokumentieren und kommunizieren ISBN: 978-3-44642-924-6
- [SWP2012] L.Bass, P.Clements, R.Kazman
Software Architecture in Practice (3rd Edition) ISBN: 978-0321815736

Microservices

Best Practices

- Altsystem soll technisch modernisiert werden
- Unsere Neugier fiel dabei auf das Architekturmuster „Microservices“
- Experten wurden zusammengestellt (Entwickler, Fachabteilung)
- ... und folgende nicht funktionalen Anforderungen wurden gestellt
 - Leichte Erweiterbarkeit des Systems
 - Testbarkeit
 - Hoher Grad an Wiederverwendungsmöglichkeiten
 - Skalierbarkeit
- Mein erster Gedanke war: aha wir sollen also die „eierlegende Wollmilchsau“ entwickeln.

- Funktional sehr gute Softwaresysteme
 - ... Jedoch nicht in jedem Fall optimal betrieben, gewartet und genutzt
 - ... und technische Schulden wurden aufgebaut (feat., feat., feat.)
- Diese *nicht funktionalen* Anforderungen führten zur Suche nach neuen Architekturansätzen



Fragen:

- Wie definieren sich Microservices?
- Wann sollte die Microservices Architektur verwendet werden?
- Wie entwickle ich konkret Microservices?
- Wie betreibe ich Microservices?
- Erreichen wir mit Microservices die *qualitativen* Anforderungen besser?

Ziele:

- Einen noch höheren Grad der fachlichen Nachnutzung einzelner Services
- Eine einheitliche Integrationsstrategie
- Eine Infrastrukturplattform die Microservices bzw. die fachlichen Anwendungen ausfallsicherer, höher verfügbar, skalierbarer und flexibler macht
- Schnellere Realisierung von fachlichen Anforderungen

Microservices - auch bekannt als Microservice-Architektur - ist ein Architekturstil, der eine Anwendung als eine Sammlung von Diensten strukturiert, die

- Hochgradig wartbar und testbar sind
- Lose gekoppelt
- Unabhängig implementierbar
- Um Geschäftsfunktionen herum organisiert sind
- Von einem kleinen Team verwaltet werden

Die Microservice-Architektur ermöglicht die schnelle, häufige und zuverlässige Bereitstellung von großen, komplexen Anwendungen.

Sie ermöglicht es einem Unternehmen außerdem, seinen Technologie-Stack weiterzuentwickeln.

Quelle: <https://microservices.io/>

- **Granularität**

Durch eine zentrierte Modellierung werden verschiedene Variationen der Granularität erlaubt

- **Hohe Kohäsion**

Unter dem Gesichtspunkt der fachlichen / technischen Trennung respektieren die hochgradig kohäsiven Microservices das Prinzip der „single responsibility“

- **Geringe Kopplung**

Geringe Kopplung, als ein Kernprinzip von MS, ist in den Mindestqualitätsmerkmalen enthalten

- **Skalierbarkeit**

Die Fähigkeit eines Microservices, unabhängig von Änderungen der Größe (Menge der Ressourcen) korrekt (wie entworfen) zu funktionieren, ohne Leistungseinbußen in Kauf nehmen zu müssen

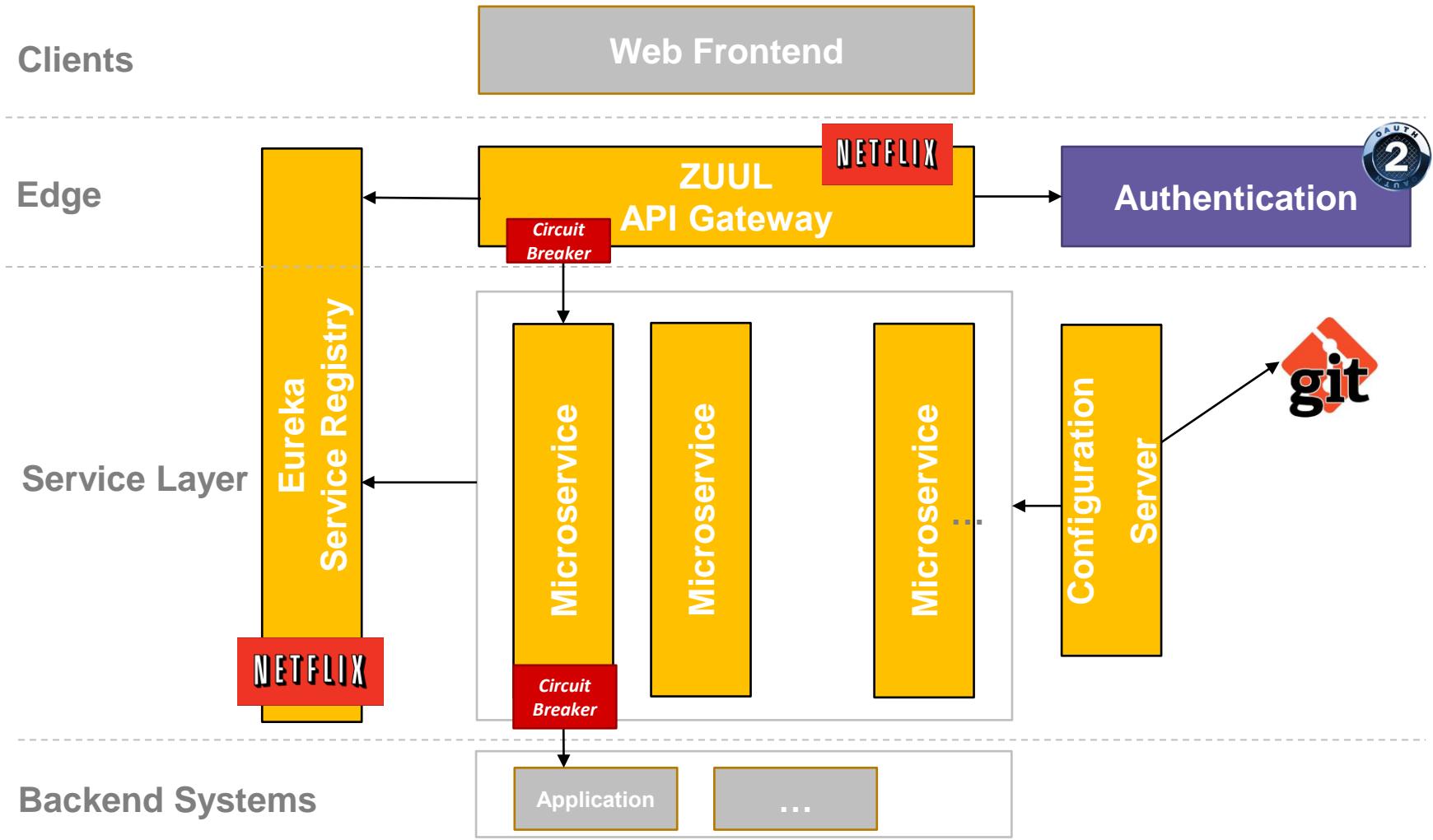
Quelle: Vrije Universiteit Amsterdam
<https://alexandru-uta.github.io/pubs/attributes.pdf>

Vorteile

- Die unabhängige Arbeit der Teams
- Isolierung technischer Entscheidungen
- Die enorme Skalierbarkeit „out of the box“

Nachteile

- Die Komplexität der Komponenten
- Viel mehr Einheiten zu deployen
- Der Betrieb der einzelnen Komponenten



- Service Registry
- Load Balancing
- Widerstandsfähigkeit
- Integration
- Umsetzung

- Stellt in gewisser Weise eine Art Telefonzentrale dar
- Eine Clientanwendung, ein DMS, eine Webanwendung fragt einen Service an (/search/CustomerRessource)
- Die Antwort beinhaltet eine vollständige URL und dient als Einstiegspunkt des Services

Vorteile

- Zentrale Übersicht aller Service und deren Status
- Möglichkeit eines Load balancing
- Keine clientseitige Anpassung bei Service und Infrastruktur Änderungen

- Rechtfertigen diese Vorteile den Aufwand und den Betrieb einer Service Registry?

Aufwand ist gering:

- Eureka (weit verbreitet)
- Consul (weit verbreitet)
- Zookeeper
- Etcd

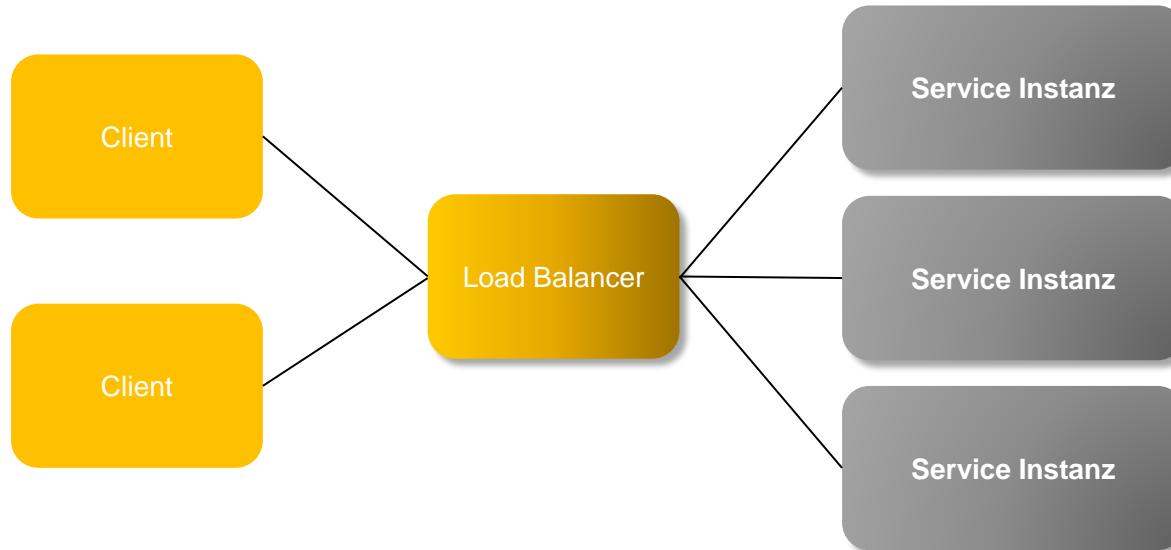
In unserem Projekt haben wir hauptsächlich auf den Netflix OSS gesetzt und daher auf Eureka.

- Service Registry darf nicht zum SPoF werden
 - Idee von Eberhardt Wolff: Spring Boot mit aktiver Registry
- Service Registry auch ohne Microservices hat unschätzbaren Wert
 - Anpassungsaufwand entfällt
- Wir konnten für Produkte, welche auf http-Kommunikation basieren, Extensions schreiben ohne Produktanpassung

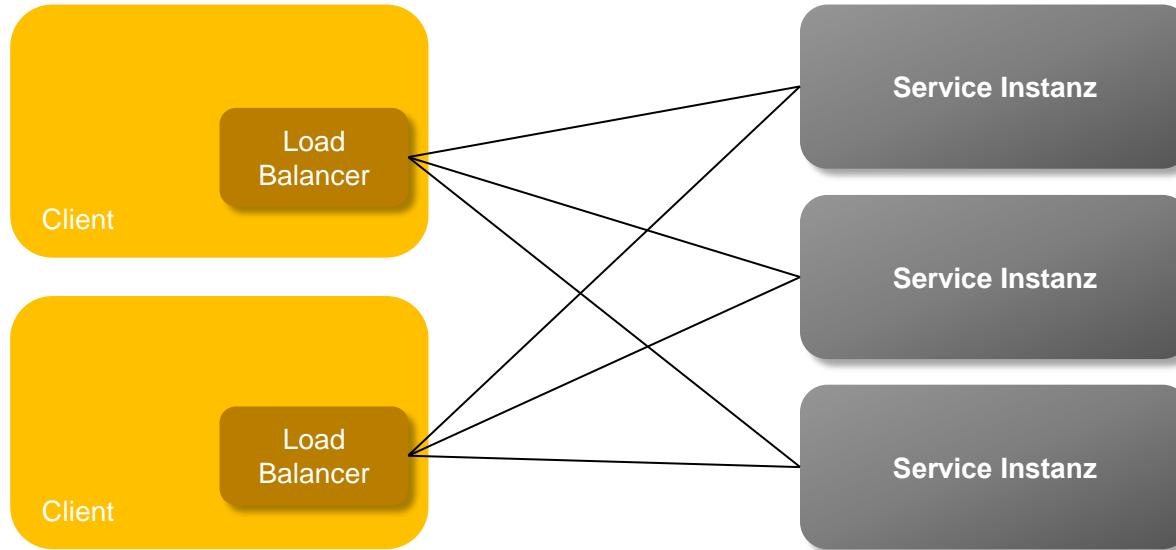
Ziele:

- Skalierbarkeit war ein sehr wichtiges Thema in unserem Projekt
- Zentrale Dienste wie Solr und Elasticsearch
- Dynamisches Deployment von Services auf zusätzlichen Servern
 - Automatisiert ohne Administrations-, Konfigurations- bzw. Entwicklungsaufwand

Zentrales Load Balancing:



- Dezentrales Load Balancing



- Netflix OSS bietet mit Ribbon dezentrales, client-seitiges Load Balancing
- Elegante Verwendung von REST Templates

- Test eines Ausfalls von einem der Services während des Uploads von 1000 XML Requests
- Mit Einsatz von Eureka und Ribbon
 - Anwender bemerkt nichts
- Load Balancing mit dezentralen Services empfehlenswert
- Aufwand der Integration in eine Java Anwendung ist gering
- Gut skalierbar und sehr robust

- Widerstandsfähigkeit bzgl. Serviceausfälle oder Netzwerklatenz war ein wichtiger Aspekt
- Client Anwendung soll nicht blockieren
- Client Anwendung wartet solange bis entweder die Antwort eintrifft oder aber ein Timeout eintritt
- Als Lösung für solche Situationen bietet sich Sentinel (Alibaba Open Source), als sogenannte „Circuit Breaker“ Bibliothek an.

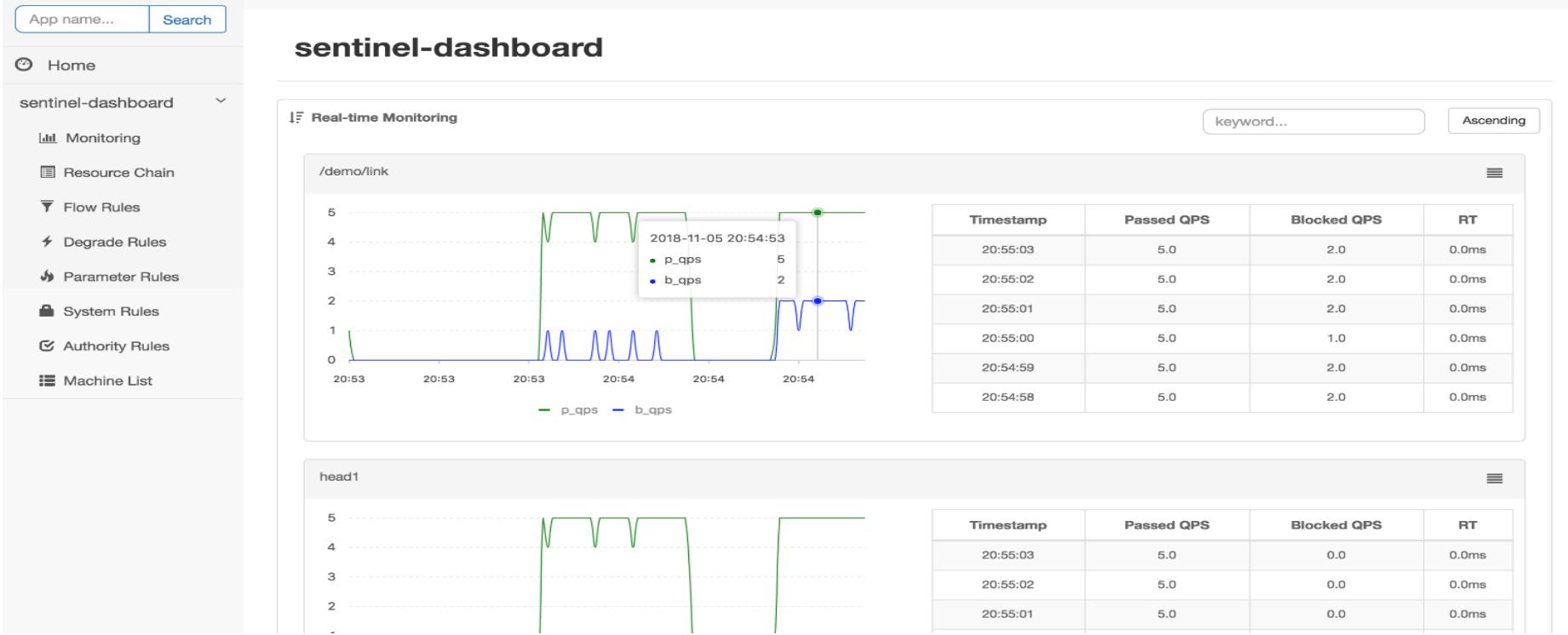


Funktionalitäten

- kapselt ein Kommando und bietet synchrone und asynchrone Ausführung,
- löst nach definiertem Timeout die Sicherung aus,
- führt jedes Kommando in einem definierten Thread Pool aus,
- Auslösen der Sicherung bei zu hoher Fehlerrate eines Kommandos,
- ermöglicht die Ausführung eines Fallbacks im Fehlerfall,
- erzeugt Performance Metriken,
- ermöglicht die Nutzung eines Caches.

Circuit Breaker – Beispiel Sentinel

Sentinel Dashboard



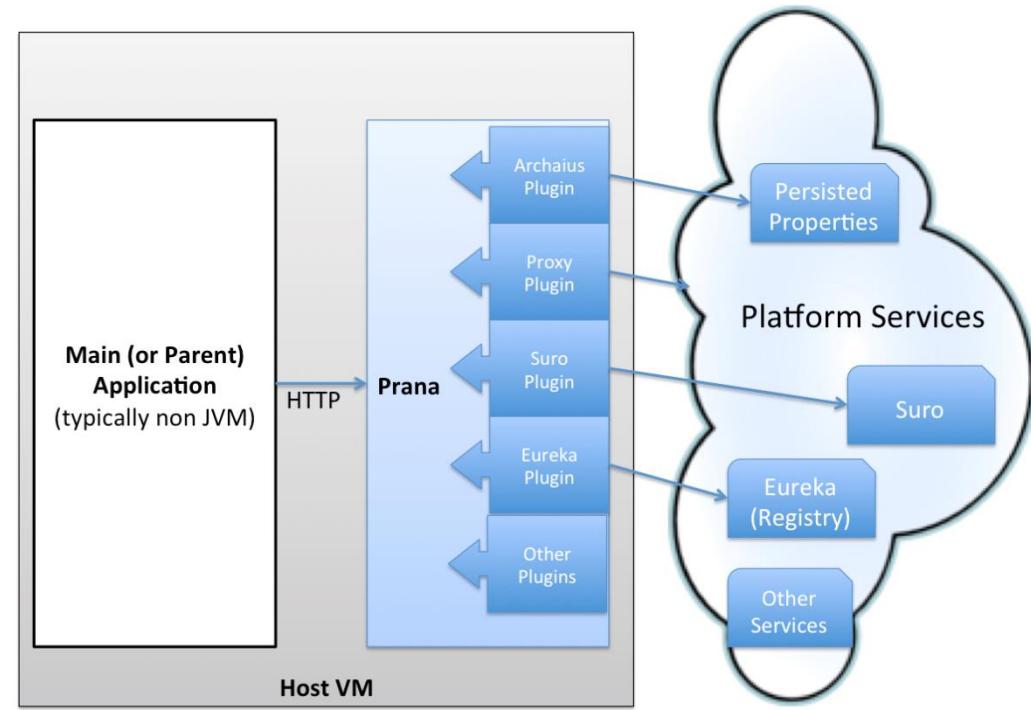
Sentinel unterstützt Circuit Breaking nach

- durchschnittlicher Antwortzeit,
- Ausnahmequote und
- Ausnahmeanzahl

Prana - Einfache Integration mit NetflixOSS-Diensten.

Prana stellt Java-basierte Client-Bibliotheken von verschiedenen Diensten wie Eureka, Ribbon, Archaius über HTTP zur Verfügung.

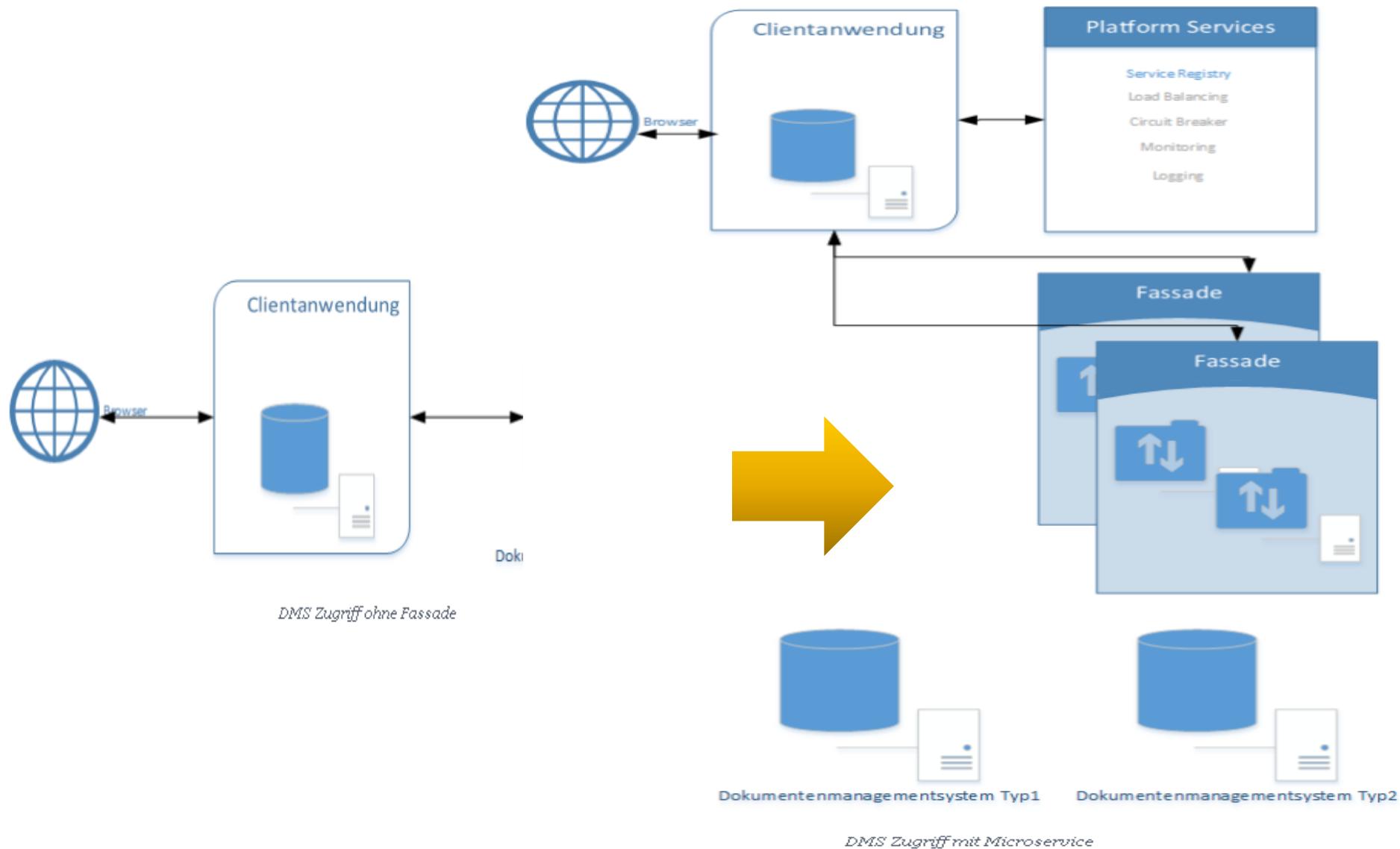
Prana macht es Anwendungen - insbesondere solchen, die in Nicht-JVM-Sprachen geschrieben sind - leicht, im NetflixOSS-Ökosystem zu bestehen.



- Wir haben Test Services geschrieben
- Ohne diese Test Services anpassen zu müssen, lassen sich die „Platform Services“ von Netflix integrieren
- Die Test Services sind damit in der Service Registry sichtbar und abrufbar
- Ein Aufruf eines Test Service über die Sidecar URL wird als Circuit Breaker Command ausgeführt
- Es werden Sentinel Metriken generiert welche im Dashboard sichtbar sind
- Es kann Load Balancing verwendet werden, sobald ein Test Service mehrfach vorhanden ist
- Es kann eine zentrale Configuration Management Instanz verwendet werden

- Integration einer Softwarekomponente, welche nicht den Netflix Stack via Bibliotheken Integration nutzen kann, ist sehr einfach
- Gilt für Komponenten die bereits über REST API angesprochen werden können.
- Für Softwarekomponenten ohne REST API ist die Integration ebenfalls möglich aber etwas aufwendiger.

Umsetzung

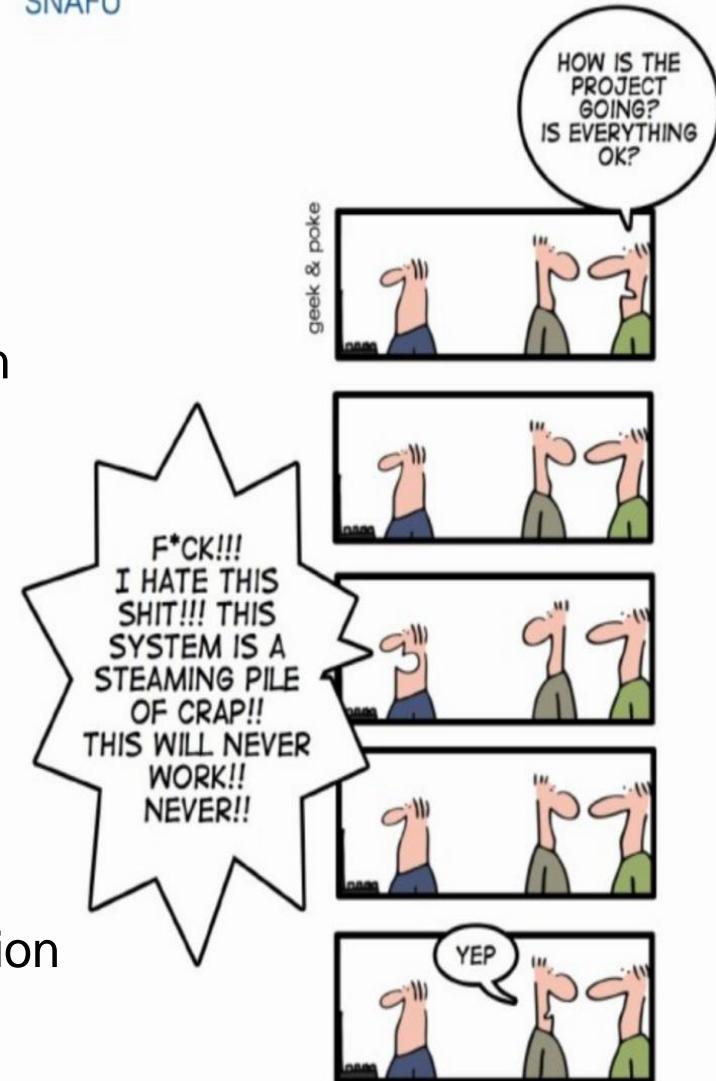


Als Microservice

- kann die Fassade sehr einfach skaliert und somit vielen Anwendungen zur Verfügung gestellt werden
- ist ein Zugriff auf das System gegen Probleme mit schlechten Netzwerklatenzen gesichert
- kann die Fassade unabhängig von den Clientanwendungen weiterentwickelt werden
- kann eine Migration bzw. ein Wechsel des Systems ohne Aufwand im Bereich der Clientanwendung durchgeführt werden.
- können alle Clientanwendungen von der Weiterentwicklung der Fassade profitieren.

- In Anbetracht der Vorteile fällt unser Feedback positiv für diesen Anwendungsfall aus
- Mit sorgfältiger Planung lassen sich die Herausforderungen dieser Architektur meistern
- Der initiale Mehraufwand ist unserer Meinung nach wegen des geringeren Gesamtaufwands zu rechtfertigen
- Der produktive Betrieb einer Microservice-Architektur gehört sorgfältig geplant
- Dauer der Umsetzung von Start bis in Produktion 6 Monate (10 Beteiligte)

SNAFU



Verwendeter Stack

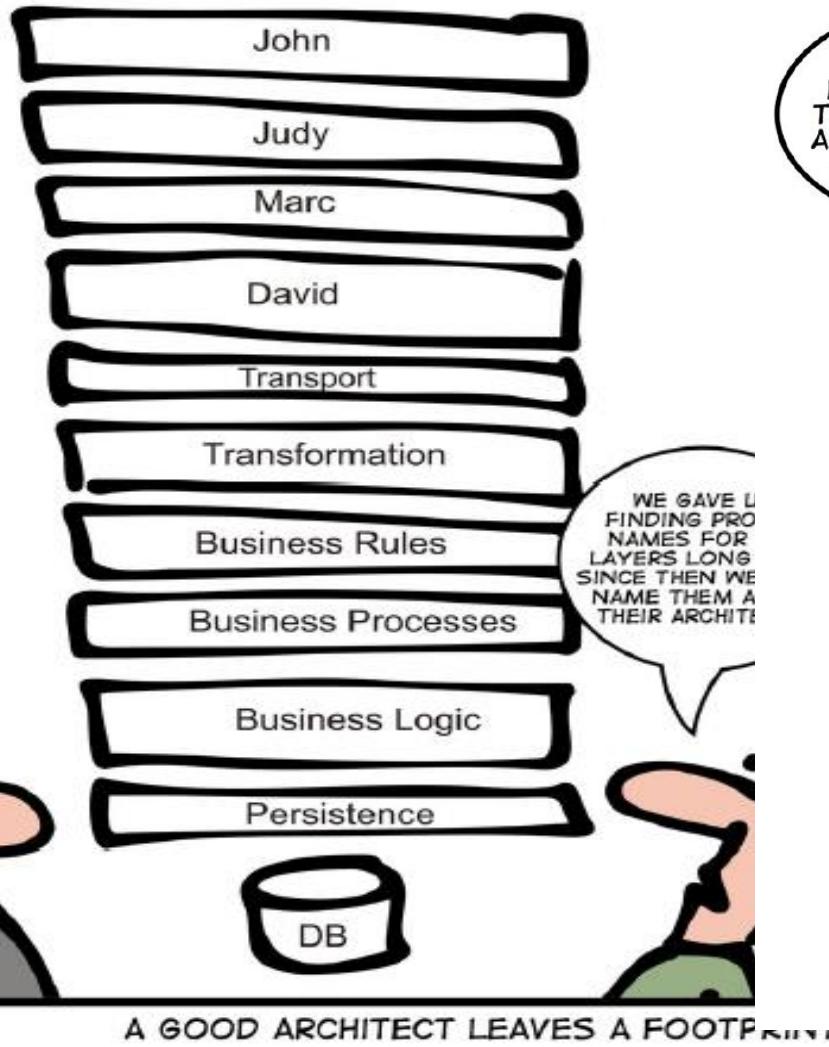
Komponente	Library
Service Provider	Wildfly Swarm
Service Discovery	Netflix Eureka
Load Balancing	Netflix Ribbon
Circuit Breaker	Sentinel
Monitor	Sentinel
OAuth 2	Spring Security
Zentrales Logging	*beat, Elasticsearch, Grafana

Technische Schulden

in Architekturen abbauen

- **Einführung - Motivation**
- Architektur Muster
- Architektur Modularität
- Architektur Hierarchien

Sometimes Reality



Architekturziel 1: **Wartbarkeit**

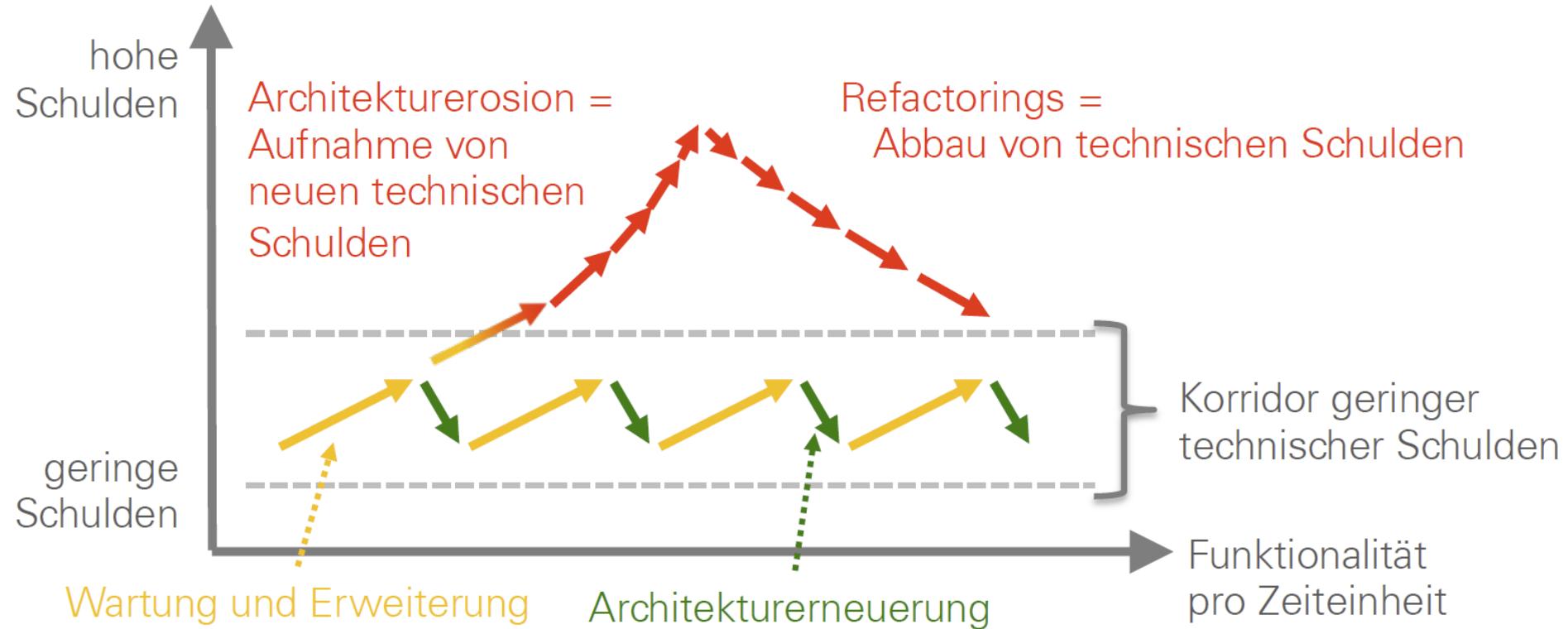
- schnelle Fehleranalyse
- schnelle Anpassungen
- Analysierbarkeit und Verständlichkeit
- Reduktion von Komplexität

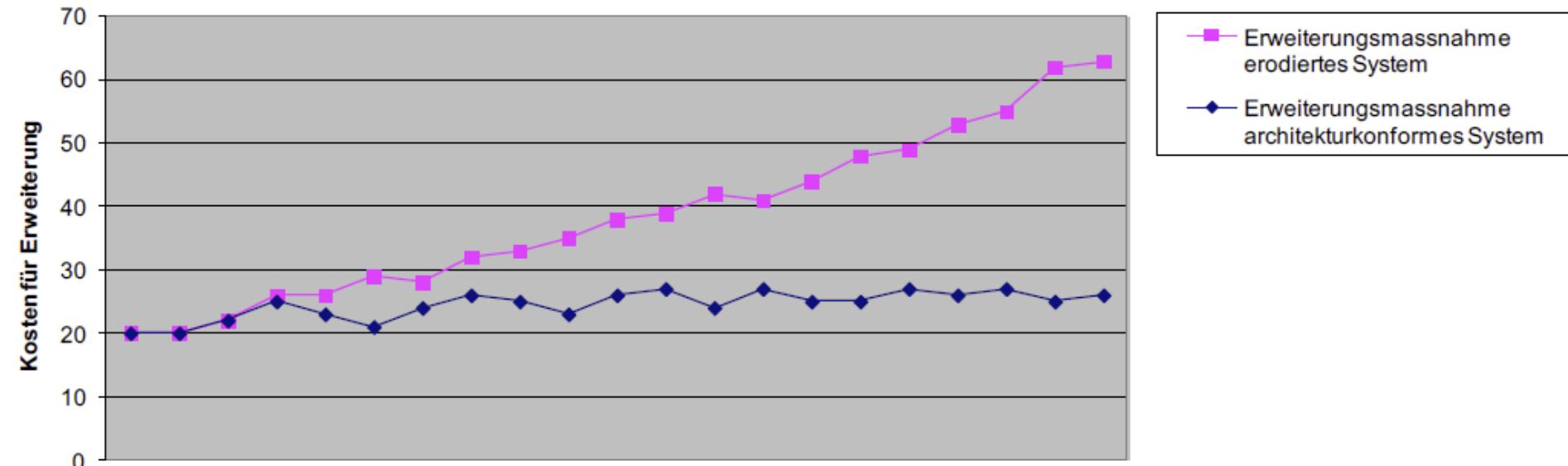
Architekturziel 2: **Flexibilität**

- Varianten von Geschäftsprozessen
- Geänderte Anforderungen
- Microservices und Skalierbarkeit
- Baukastenprinzip

Technischen Schulden = Architektur-Erosion

Ausmaß an
technischen Schulden





Erosion hat zur Folge:

- Software ist schwerer zu verstehen
- Erweiterungen verursachen viele Seiteneffekte
 - Zeit und Kosten steigen
 - Neue Fehler werden produziert
- Erfüllung der Qualitätsmerkmale sinkt

Ursachen dieses Erosions-Prozesses:

- Unbemerkt steigender Kopplungsgrad und Komplexität
- Hacks aus Zeitdruck durch Kunde und Projektleitung
- Entwicklung eines zweiten/dritten Produkts auf Basis eines Ersten
- Keine Zeit für Architektur-Diskussionen + Architektur-Erneuerung
- Fehlendes Architekturverständnis im Entwicklungsteam

Zwei Herausforderungen:

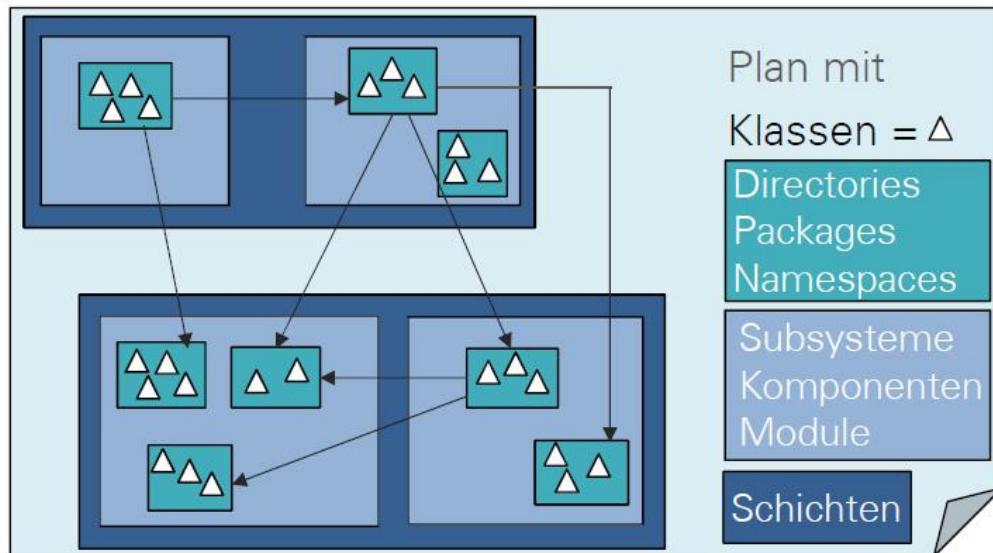
- Qualität muss früh bedacht werden:
 - Jede „kleinere“ architektonische Anpassung zieht bei größeren Anwendungssystemen große Refactorings nach sich
 - Das Vertrauen des Kunden sinkt mit jeder Auslieferung, die das Qualitätsmerkmal nicht ausreichend erfüllt
 - Qualität muss über die Lebensdauer der Anwendung sicher gestellt werden:
 - Kosten für Anpassungen steigen durch Architektur-Erosion
 - Erfüllung der Qualitätsmerkmale sinkt ebenfalls
- Beides bedingt erhöhte Anpassungskosten, sinkende Erfüllung der Qualitätsmerkmale und dadurch sinkendes Kundenvertrauen

- Festlegen von verbindlichen Architekturzielen
- Durchgängige Architekturprinzipien und Architekturstile
- Automatisches Testen und Refactoring
- Weiterbildung der Architekturen und Entwickler
- Regelmäßige Architekturanalyse und -Erneuerung

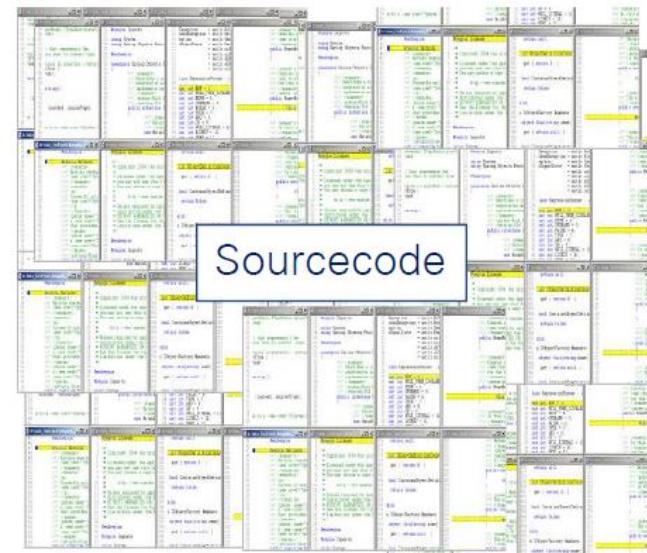
Architekturanalyse: Was ist das?

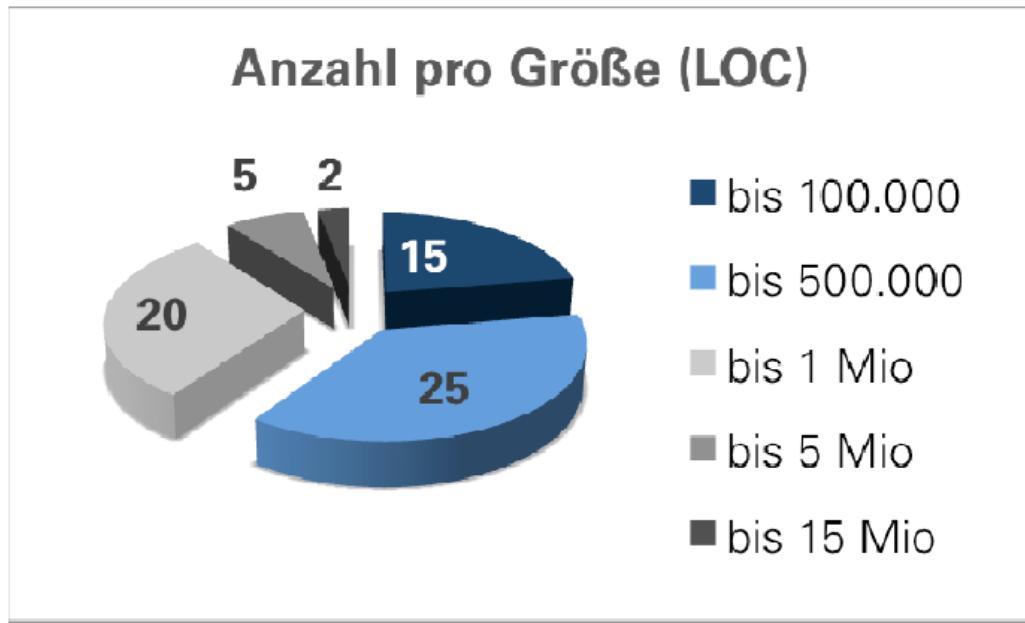
Findet sich die geplante Architektur (Soll-Architektur) in der Strukturen der implementierten Software (Ist-Architektur) wieder?

Soll-Architektur



Ist-Architektur



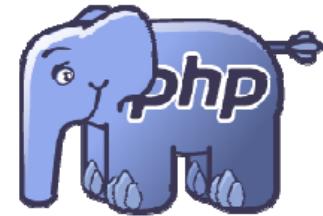


Java

Microsoft
C#.net

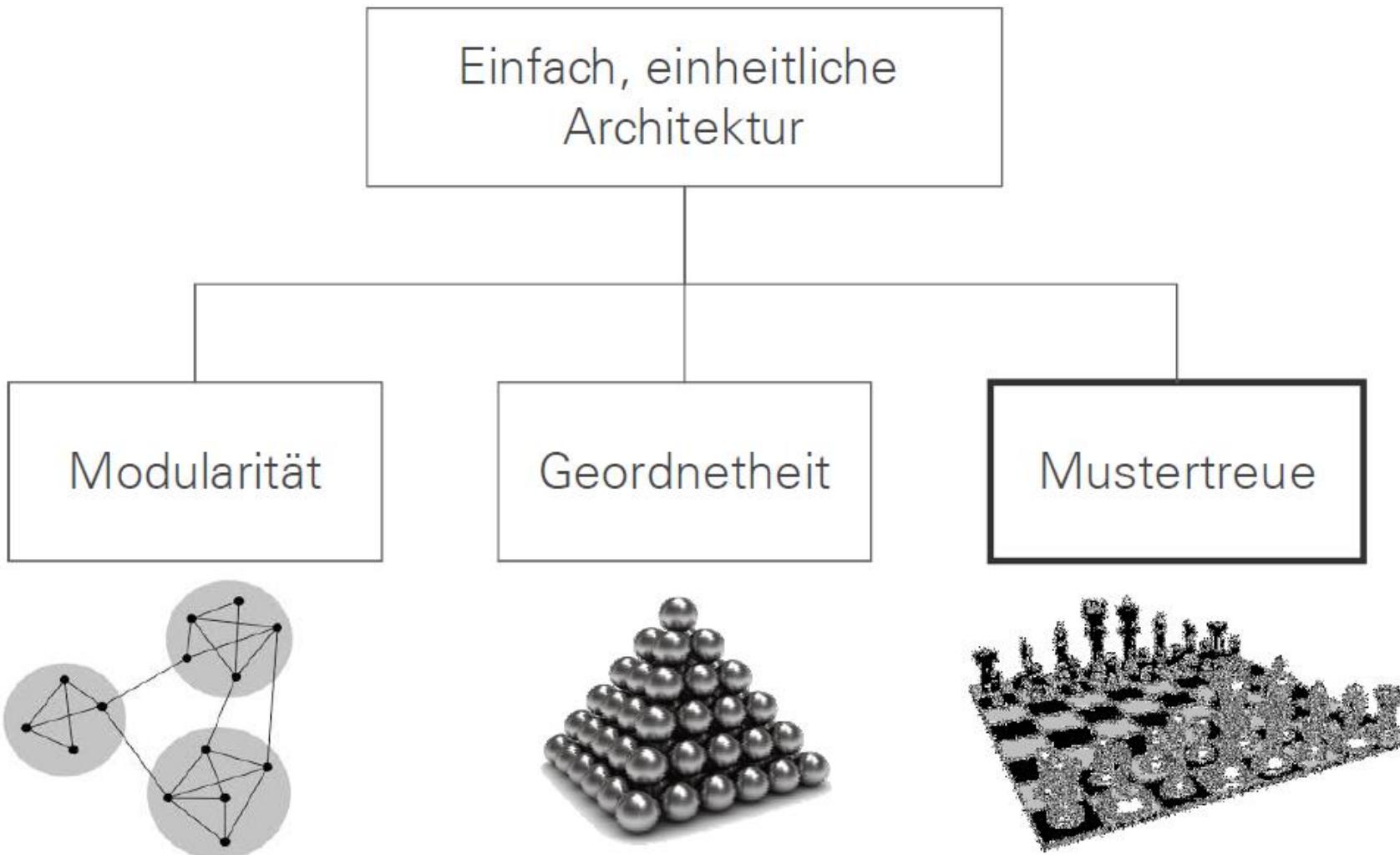
C++

SAP ABAP



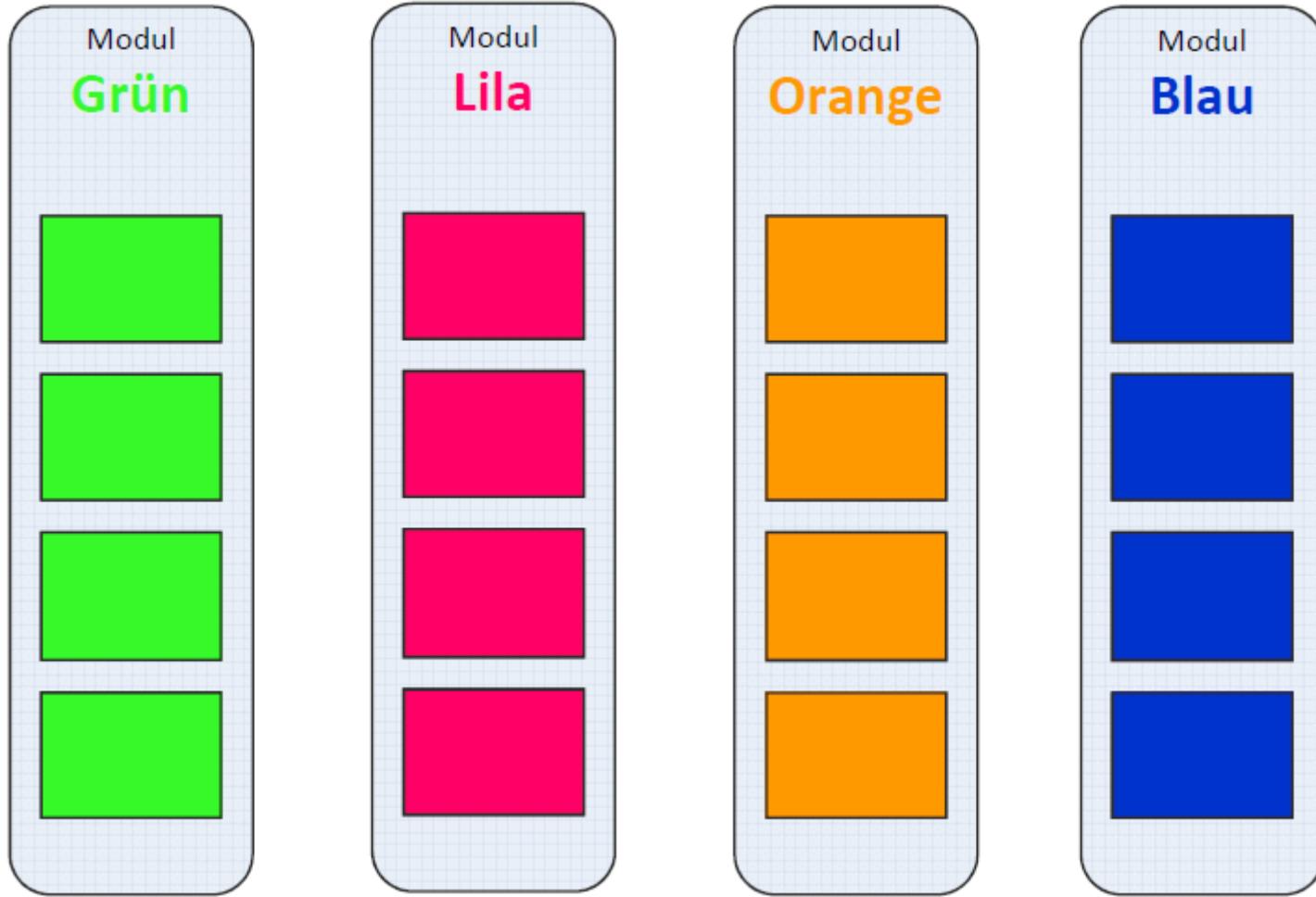
Erkenntnisse

- Typische Eigenschaften der Architektur nach Größen und Sprache
- Strukturelle Einfachheit und Einheitlichkeit ist der Schlüssel zum Erfolg
- Ohne Architektur-Erneuerung sammeln sich technische Schulden an

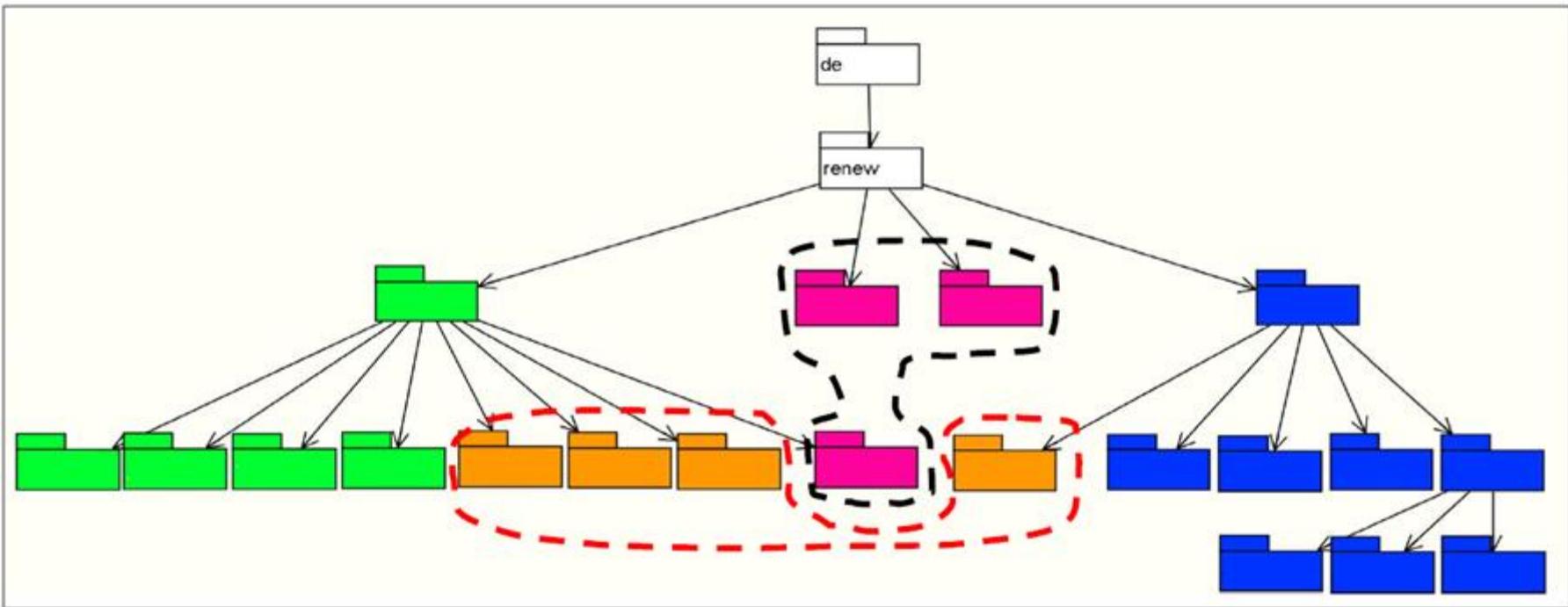


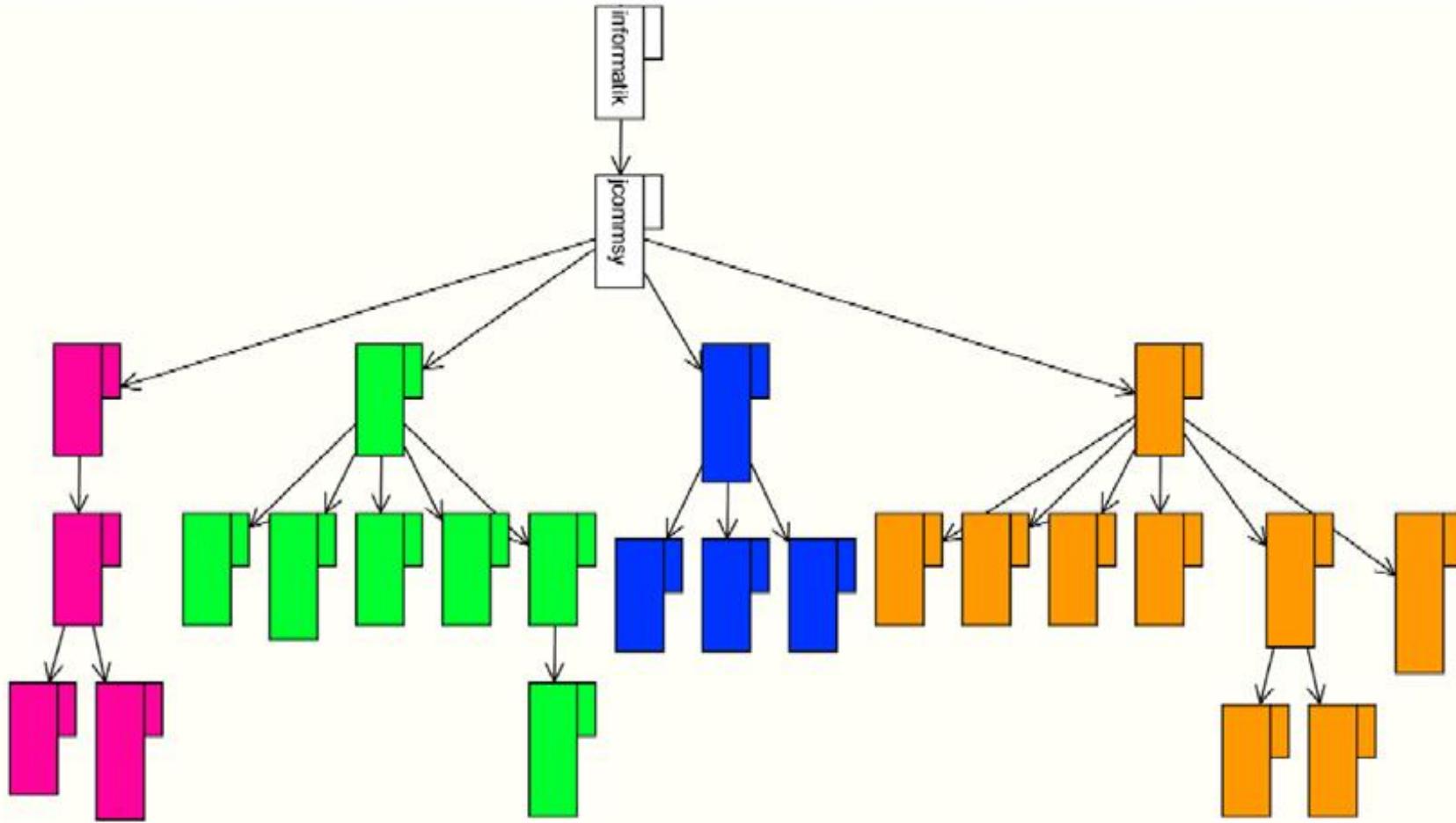
- Einführung - Motivation
- **Architektur Muster**
- Architektur Modularität
- Architektur Hierarchien

Muster auf Architekturebene: Vier Module



- Ist die Abbildung der Architektur in der Struktur des Codes zu erkennen?

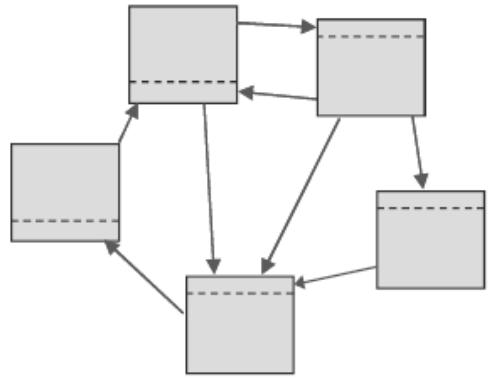




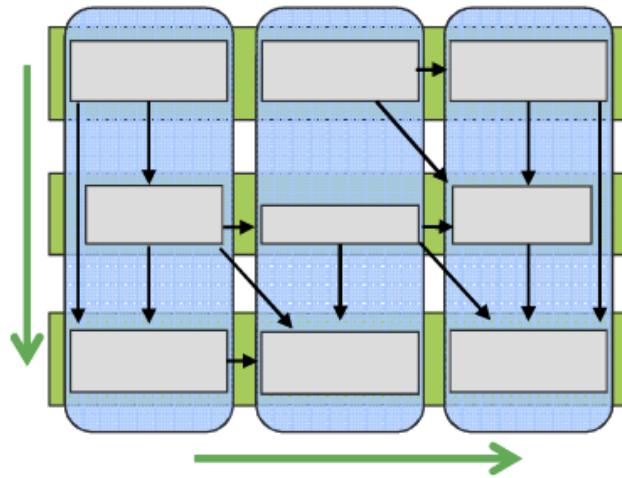
„Ein Architekturstil ist eine **prinzipielle Lösungsstruktur**, die für ein Softwaresystem **durchgängig** und unter weitgehendem **Verzicht** auf **Ausnahmen** angewandt werden sollte.“

[Reussner et al. 2006]

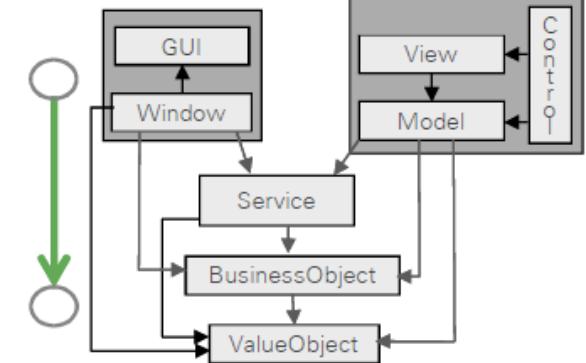
Komponentenarchitektur

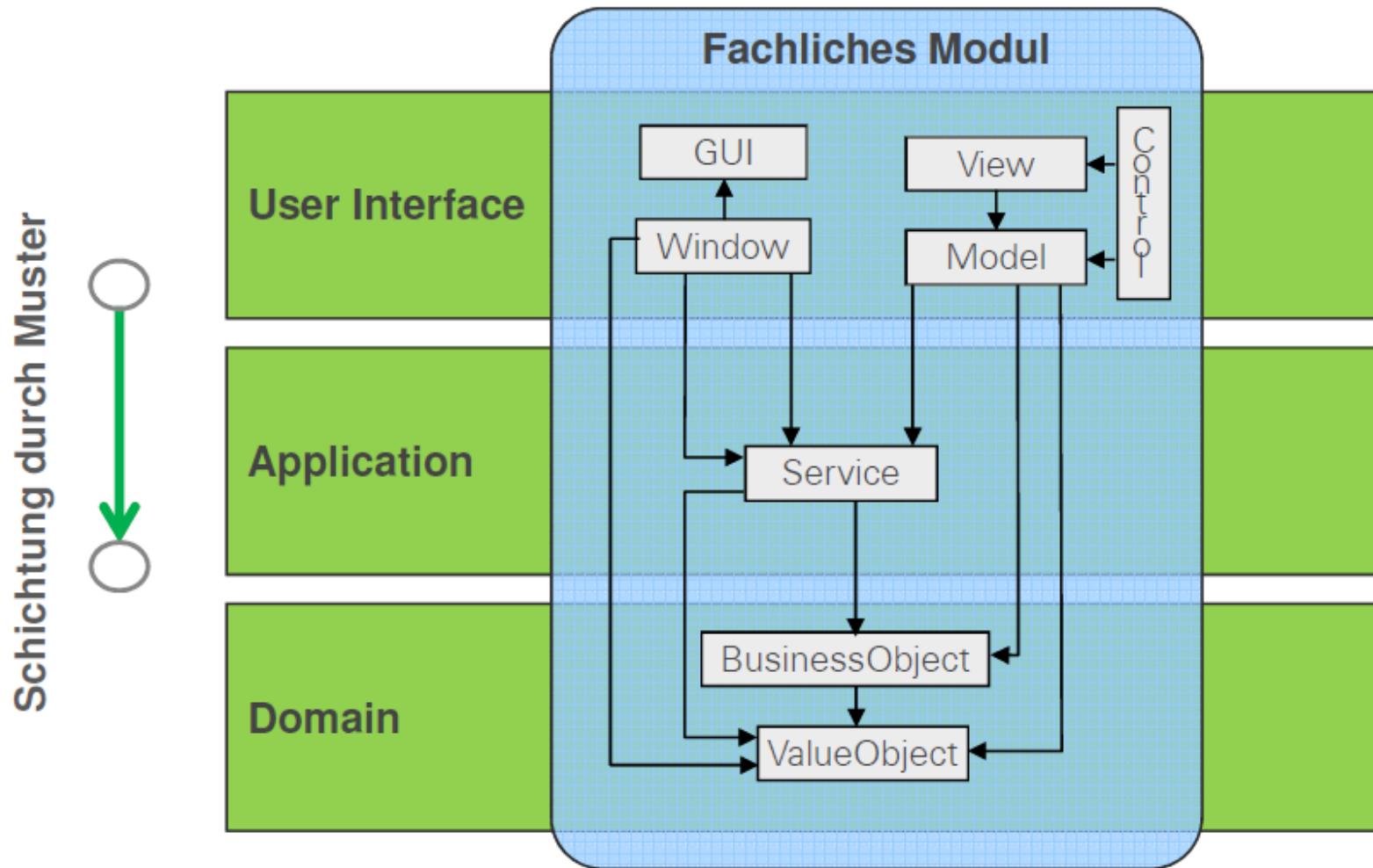


Schichtenarchitektur

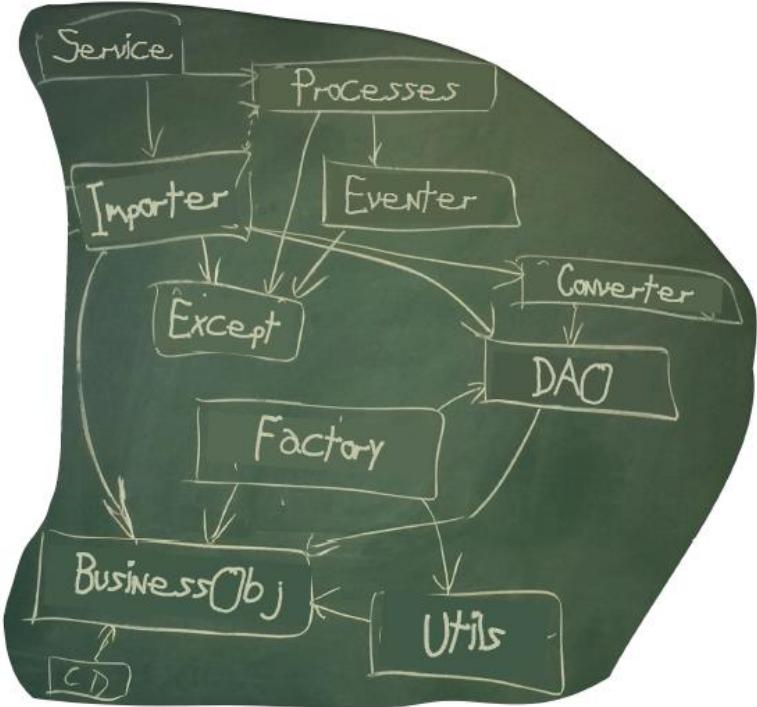


Mustersprache

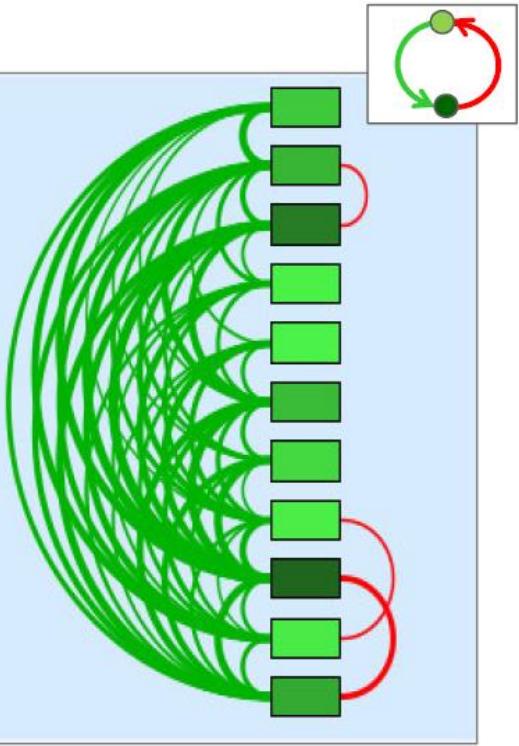




Gute umgesetzte Mustersprache

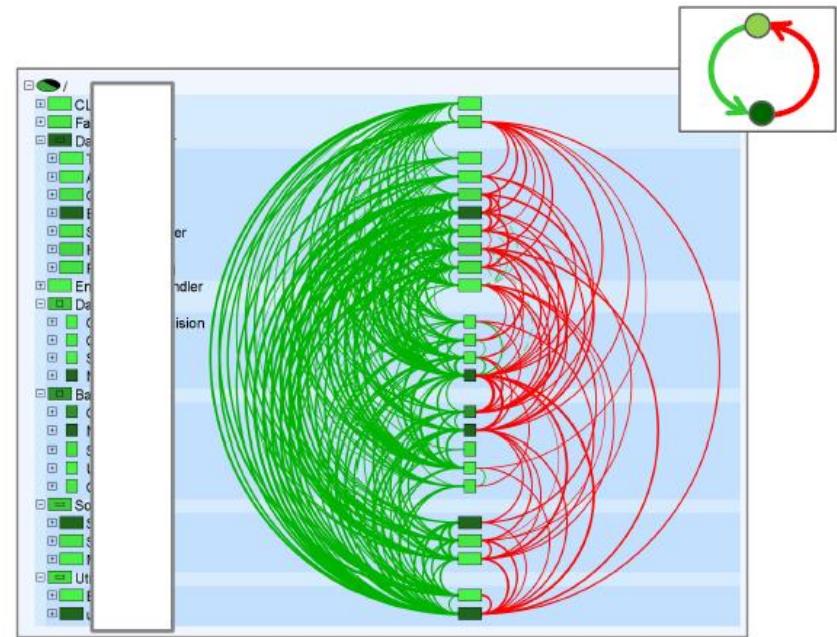
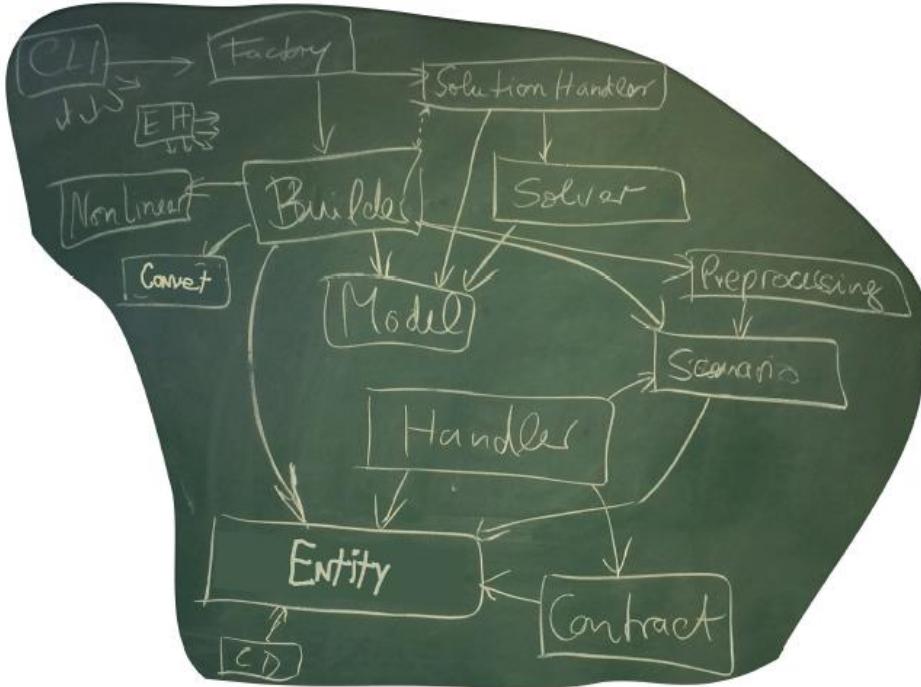


⊕	Services
⊕	Processes
⊕	Importer
⊕	Eventer
⊕	Converter
⊕	DAO
⊕	Factories
⊕	Utils
⊕	BusinessObjects
⊕	Exceptions
⊕	IDs



- ☺ 90% des Sourcecodes lässt sich den Mustern zuordnen
- ☺ 0,1% Verletzungen in den Mustern

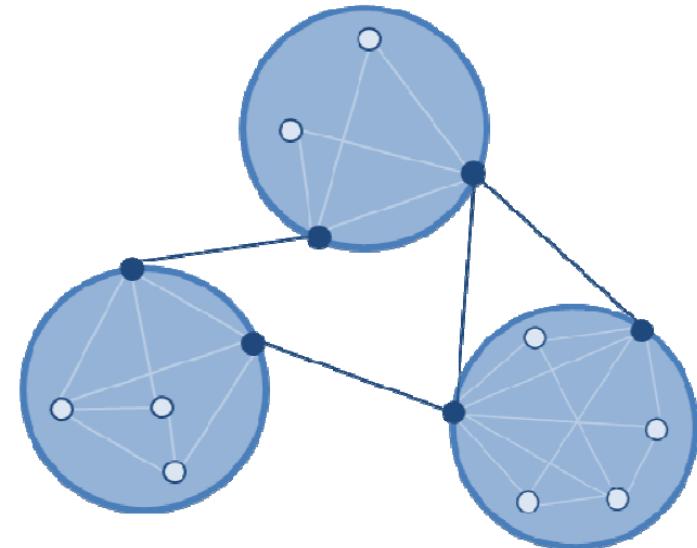
Entdeckung einer Mustersprache



- ☺ 80% des Sourcecodes lässt sich den 23 Mustern zuordnen
- ☺ 4% Verletzungen in den Mustern

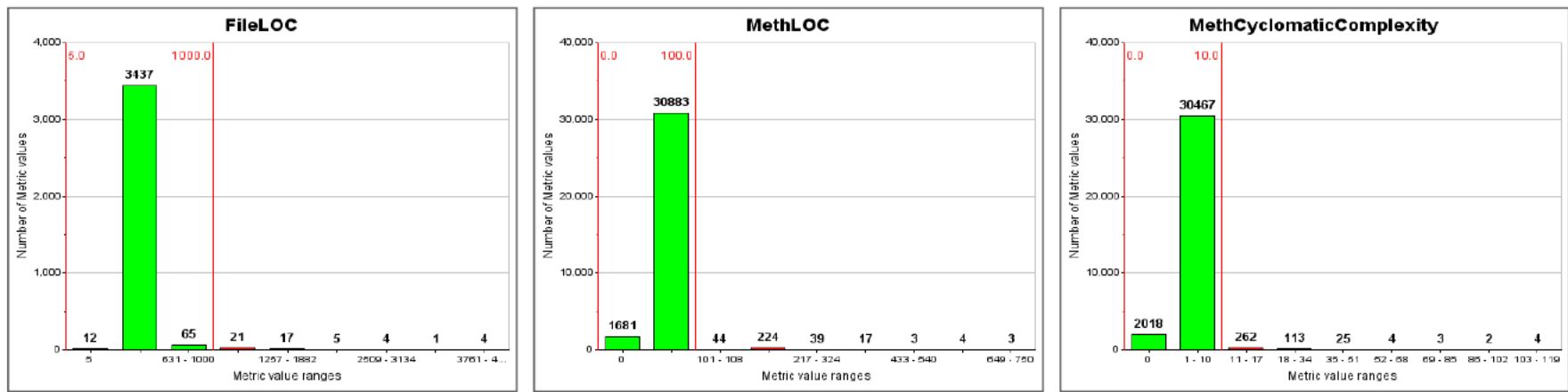
- Einführung - Motivation
- Architektur Muster
- **Architektur Modularität**
- Architektur Hierarchien

- Hohe Kohäsion und lose Kopplung
- Responsibility Driven Design
- Separation of Concern
- Single Responsibility Principle



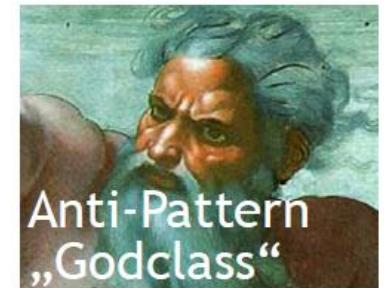
Modularität: Ausgewogene Größenverhältnisse

- Ist das System auf den verschiedenen Ebenen ausgewogen?
- Welche Code-Abschnitte fallen durch ihre Größe besonders auf?



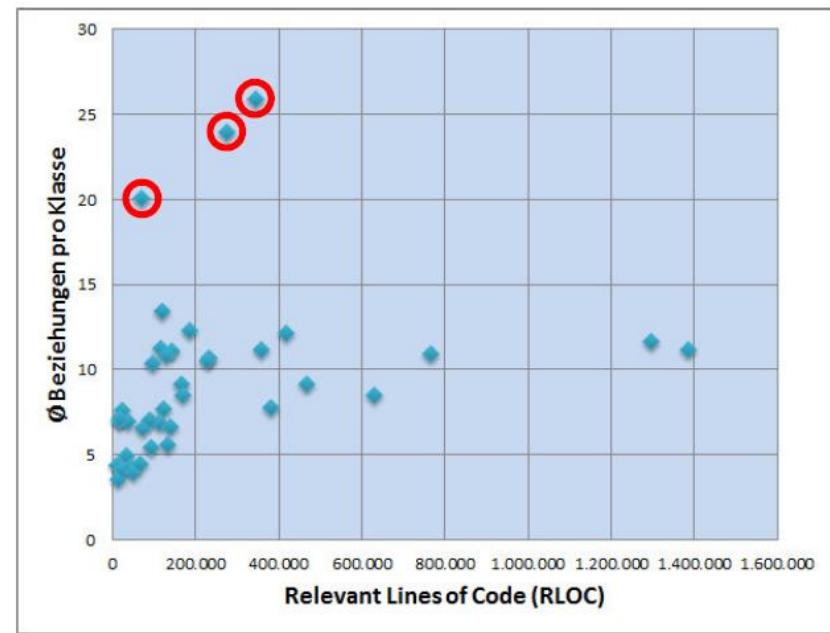
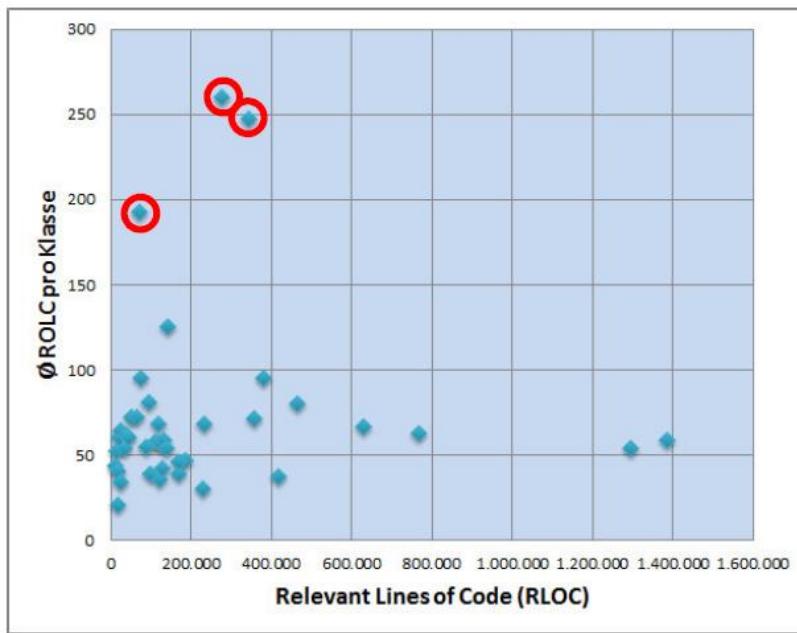
Typische Metriken:

- LOC pro Methode, Klasse, Package, Komponenten
- Duplizierter Code
- Zyklomatische Komplexität



Beispiel: Größenverhältnis und Kopplungsgrad

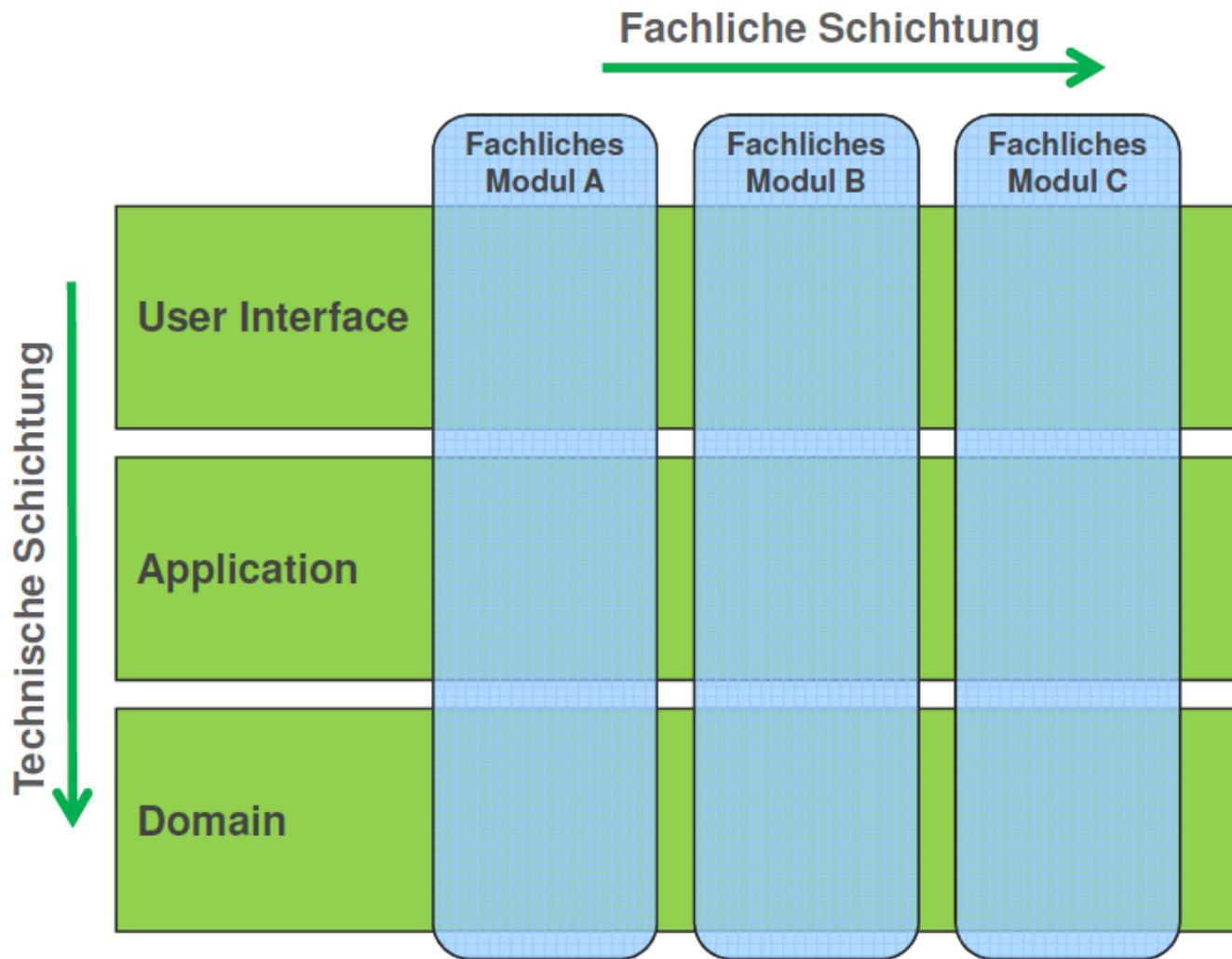
- Große Steuerungsklassen benutzen bis zu 100 – 500 andere Klassen



- Ausgewogene Größenverhältnisse führen zu geringerer Kopplung

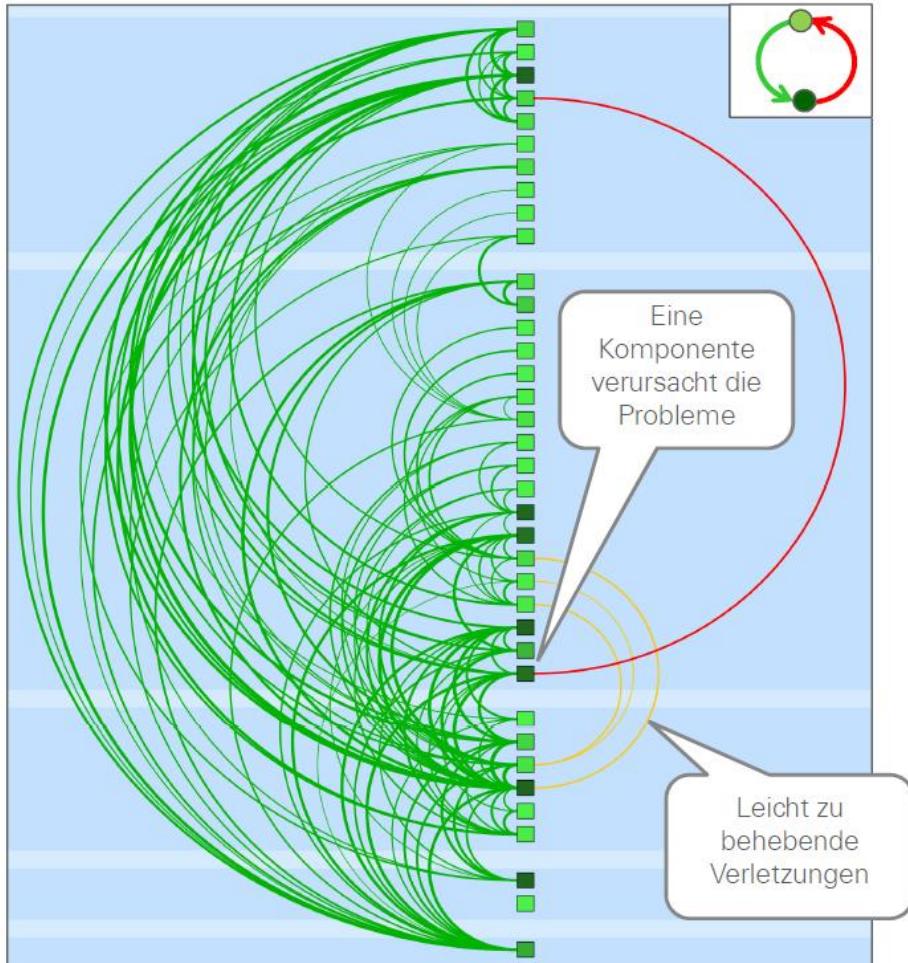
- Einführung - Motivation
- Architektur Muster
- Architektur Modularität
- **Architektur Hierarchien**

Hierarchien in Architekturebene: Schichten und Module

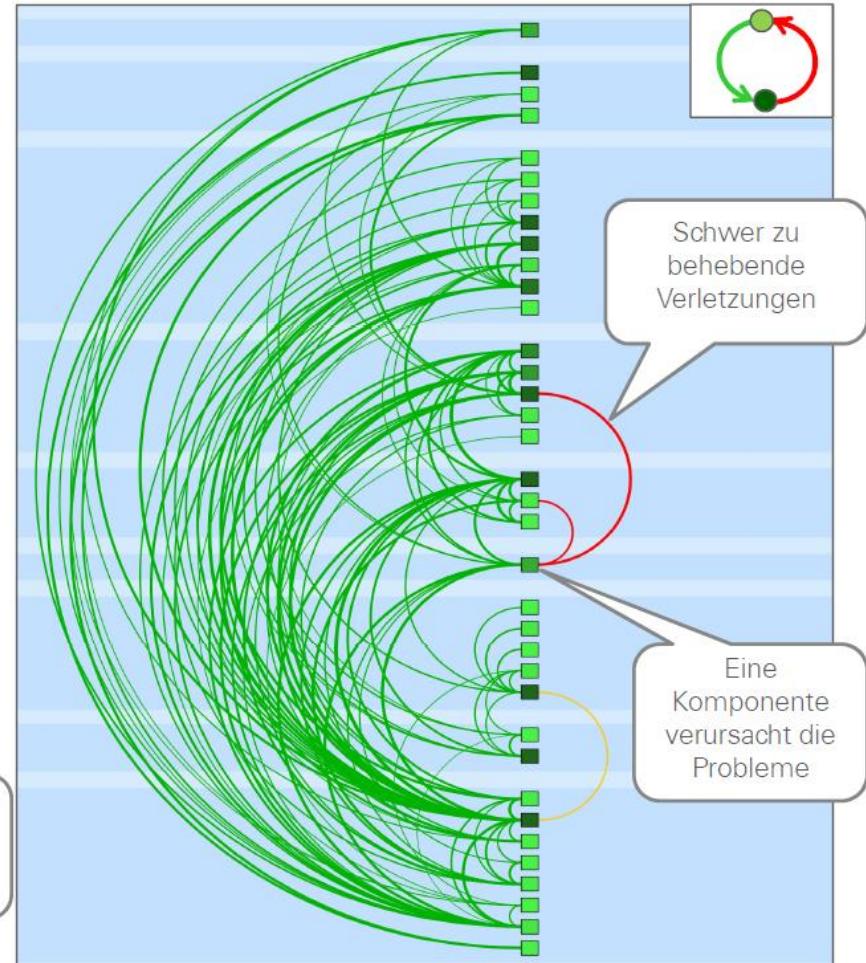


Zwei Dimensionen einer Architektur

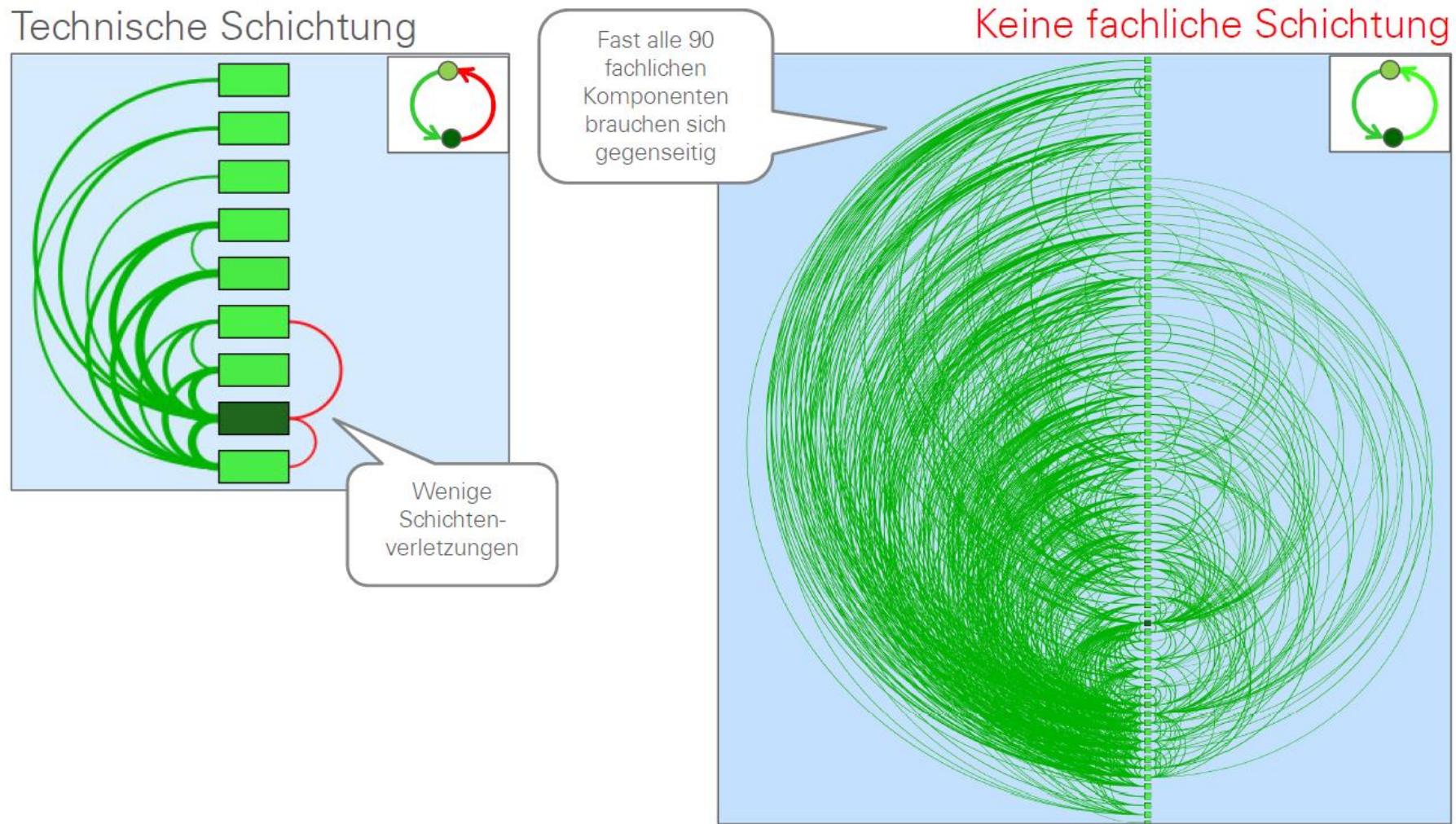
Technische Schichtung



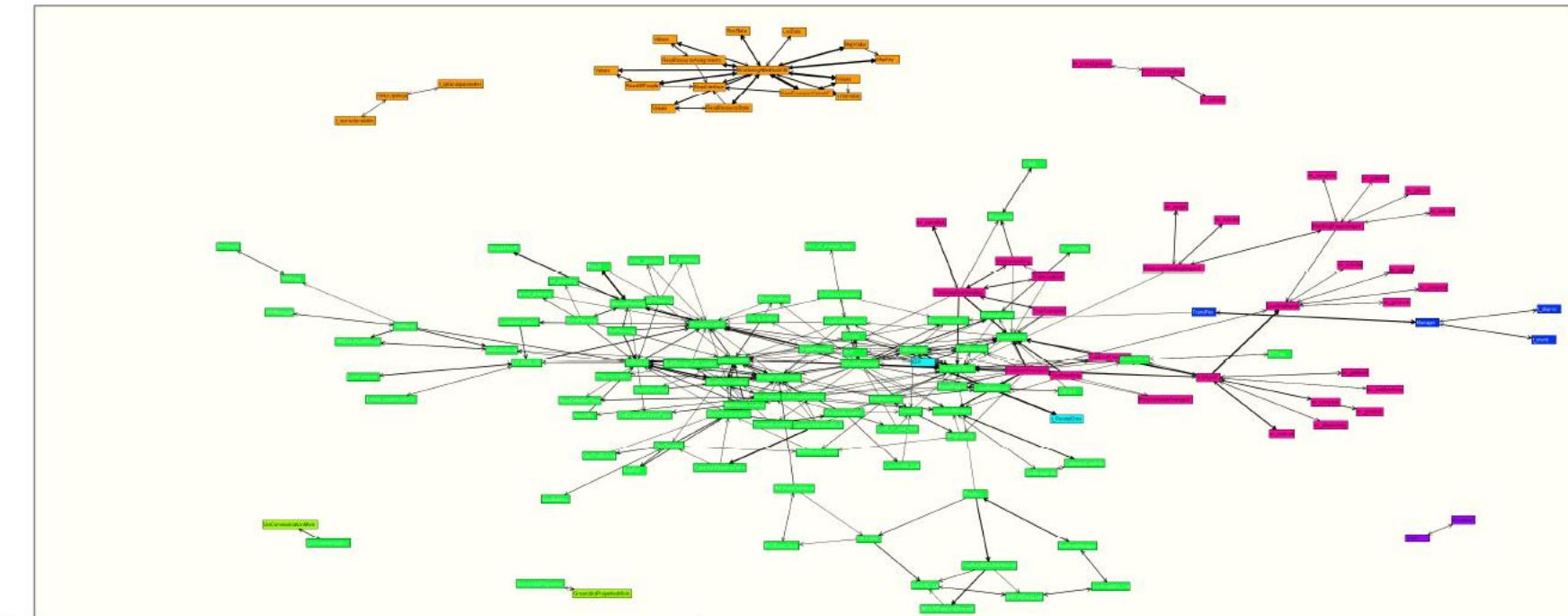
Fachliche Schichtung



Fachliche Schichtung misslungen

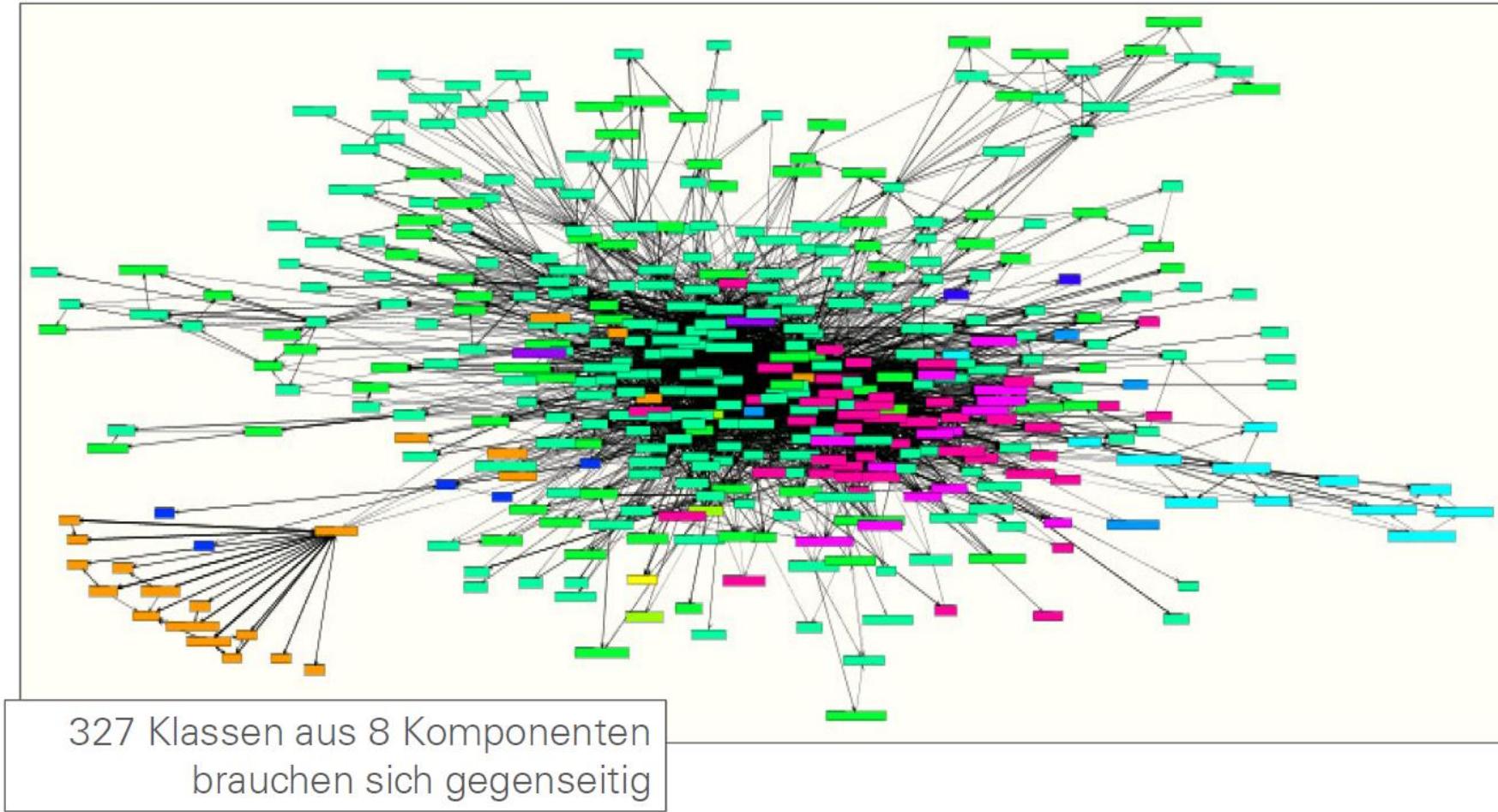


Zyklische Strukturen sichtbar machen, bevor

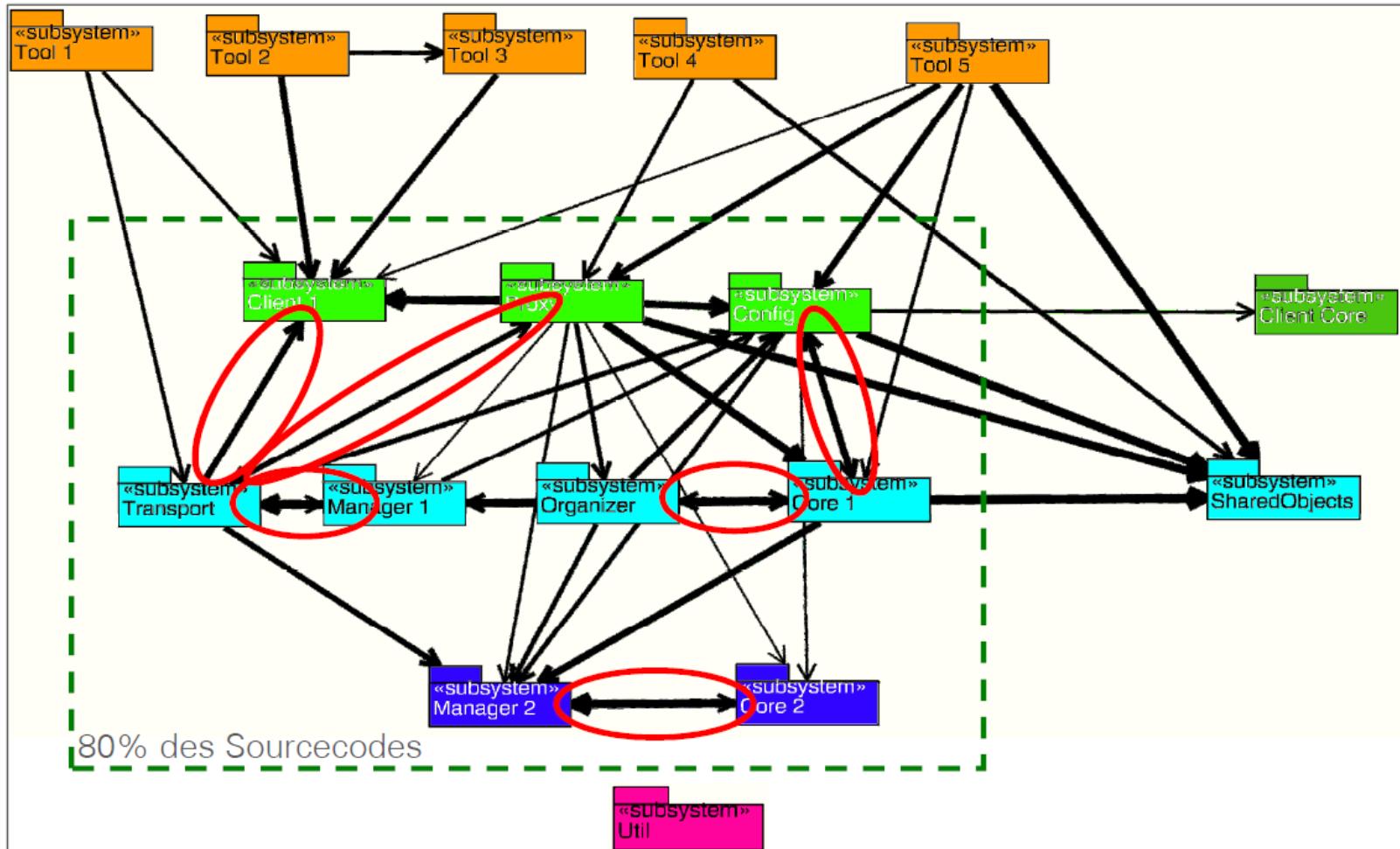


119 Klassen aus 4 Komponenten
+ 28 weitere Klassen

sie immer weiter verklumpen!



Der Zwang zur Zyklenfreiheit



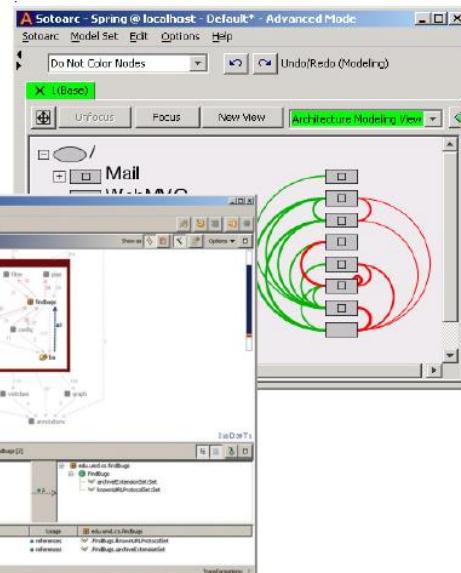
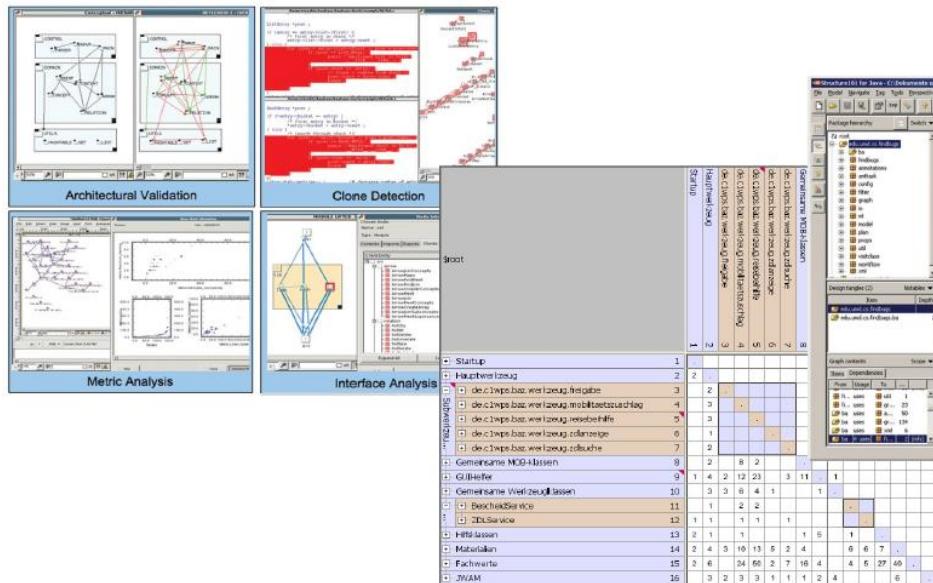
9 Komponenten = 17 Subsysteme

- Einfach, einheitliche Architektur

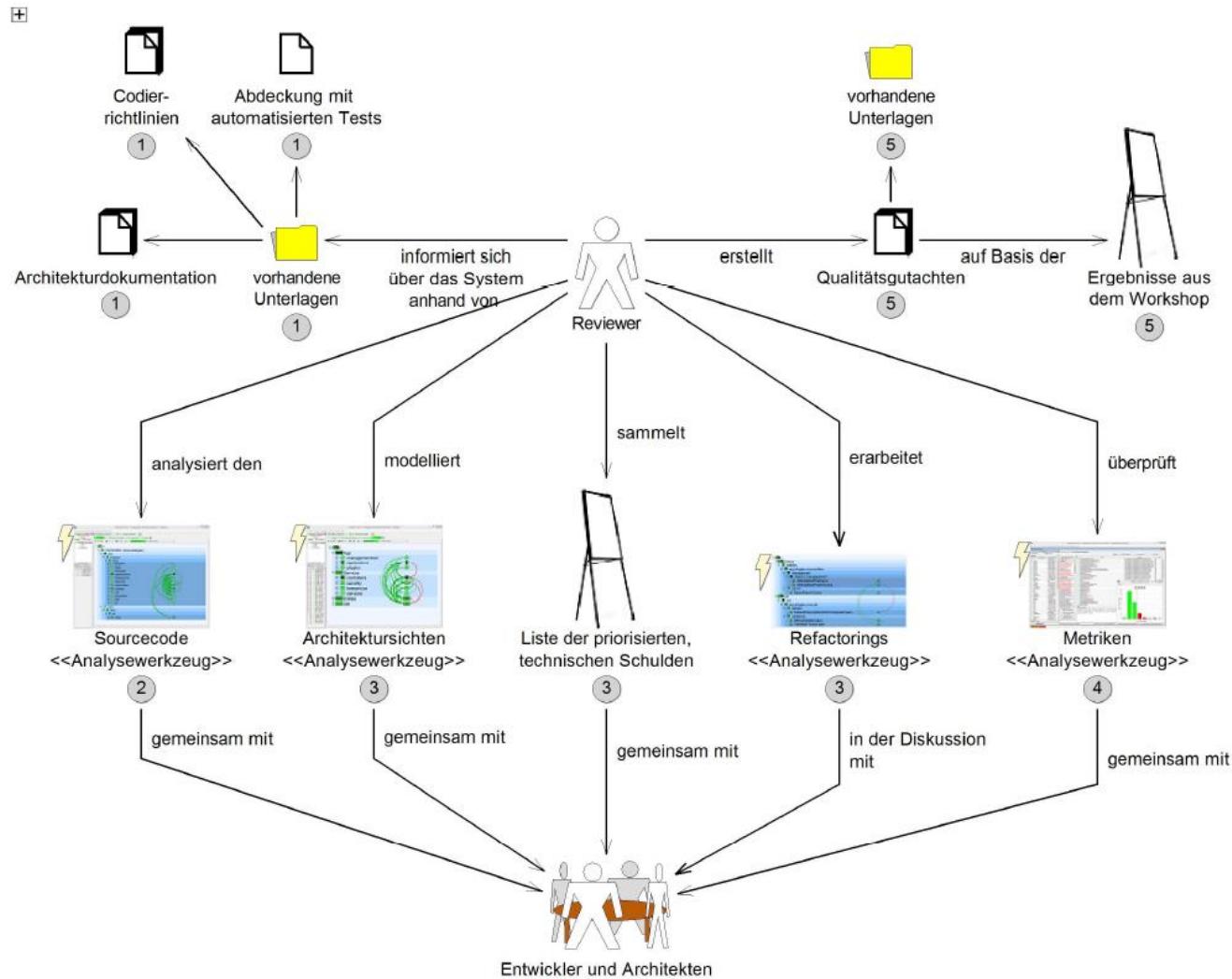
- **Musterkonsistenz**
 - Einheitlich und durchgängig
- **Modularität**
 - Zuständigkeit
 - Kopplung
 - Größenverhältnisse
 - Schnittstellen
- **Hierarchie**
 - Zyklendifferenz auf allen Ebenen

- SonarQube:
 - Leitstand für Qualitätsmetriken
 - Plattform für vielfältige Plugins
- VisualJArchitect
 - wenige Metriken
 - einfache Abhängigkeitsanalyse
- Ndepend/CDepend:
 - Metriken
 - Abhängigkeitsanalyse
- XRadar:
 - Analyse von Java-Projekten via maven
 - Reports bezüglich Komplexität und Architekturverletzungen

- Axivion Bauhaus: Java, .Net, C/C++, Ada, VB und Cobol
- Lattix: Java, .Net, C/C++, Ada, Delphi und DB-Systeme
- Structure101: Java, C++, Ada
- Sonargraph: Java, .Net, C/C++, ABAP, PHP



Vorgehen bei der Architekturanalyse und Verbesserung



Schrittweise Weiterentwicklung der Architektur



Phase 1: Aufräumen

Phase 1: **Aufräumen**
Abgleich Soll-/Ist-Architektur
fehlende Architekturkonzepte ergänzen

Phase 2: Verbessern

Phase 2: **Verbessern**
Architektur diskutieren und verbessern
Architekturregeln festlegen

Phase 3: Erhalten

Phase 3: **Erhalten**
Im Architekturkorridor bleiben
Langlebigkeit fördern

Initialer
Workshop

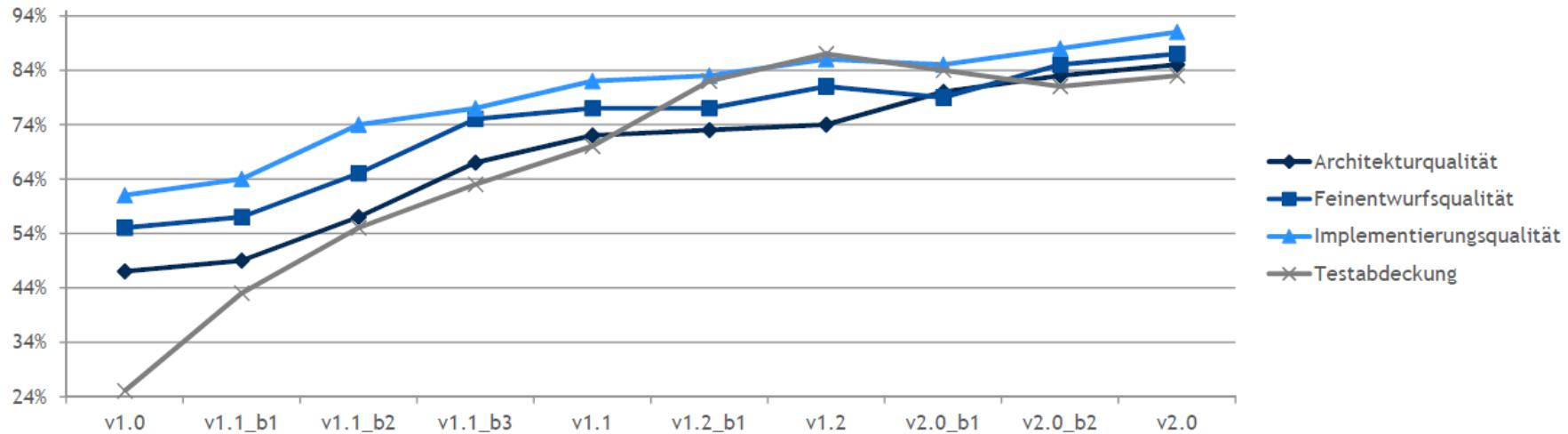
Verletzungen beheben
Strukturen einziehen

Analyse-
Workshop

Anpassungen an
neue Architektur-
Regeln

**Nach-
sorge**

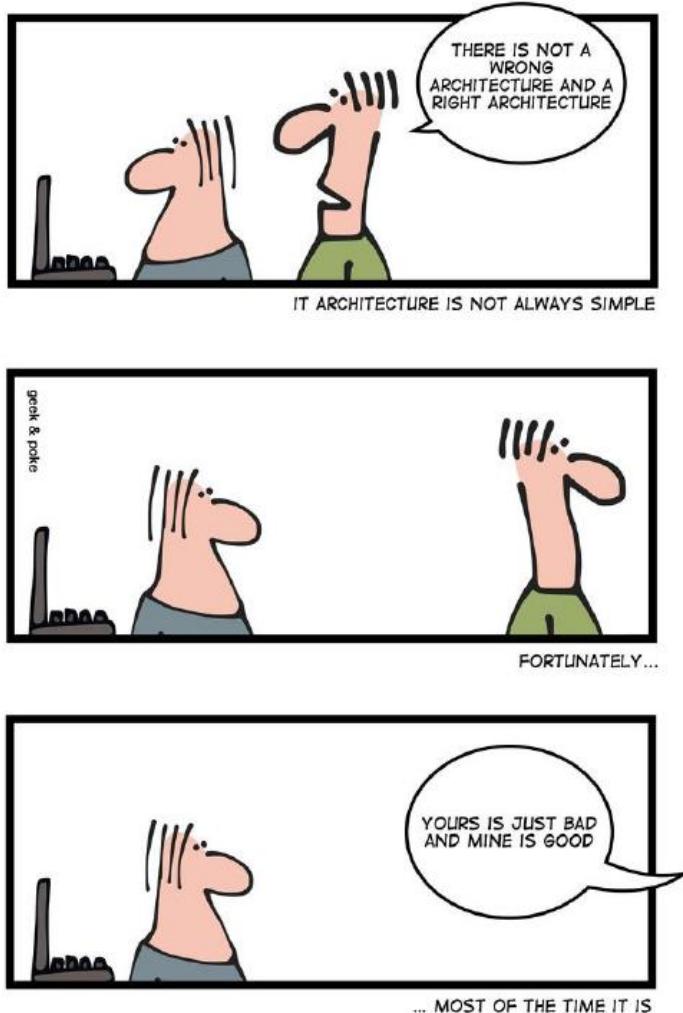
kleinere
Reparaturen



Ergebnis

- Die Architekturziele sind im ganzen Team präsent und werden verfolgt.
- Softwarewartung und -Änderung ist einfacher und kostengünstig.
- Die Software ist stabil, flexibel und langlebig.
- Neue Mitarbeiter können nach kurzer Zeit produktiv mitentwickeln.

Vielen Dank für Ihre Aufmerksamkeit!



© Integrata Cegos GmbH

Integrata Cegos GmbH

Zettachring 4
70567 Stuttgart

Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.