

# Software Architektur

# Wer ich bin

Peter Reinhardt  
Enterprise Architect @ BASF

[peter.reinhardt@basf.com](mailto:peter.reinhardt@basf.com)  
0621/60-46587



# Einleitung

- Moderne Softwaresysteme sind extrem komplex

Die Rolle des Softwarearchitekten ist herausfordernd. Diese Vorlesung gibt eine Überblick über Software Architektur, über die Aufgaben und Verantwortlichkeiten eines Softwarearchitekten.

Softwarearchitekten benötigen technische und nicht-technische Skills, teilweise breit angelegtes Wissen, teilweise Wissen bis in die Details.

Die Vorlesung beginnt bei Anforderungen aus dem Business, der Arbeit in Organisationen und der Erfassung von Anforderungen.

Wir werden uns dann technischen Themen wie Software Qualität, Software Design, Best Practices, Patterns, Performance und Security widmen.

Nach der Vorlesung sollten Sie einen Überblick über die komplexe Welt der Softwarearchitektur besitzen.



# Grundlagen + benötigte Kenntnisse

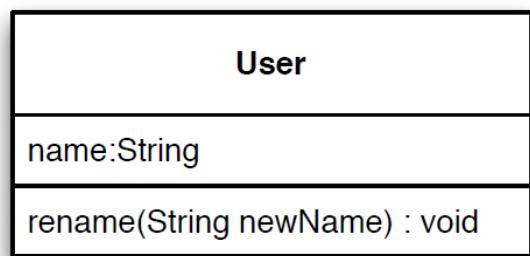
- Basiswissen in Objektorientierung

- Klassen
- Objekte
- Attribute
- Methoden
- Vererbung
- Polymorphismus

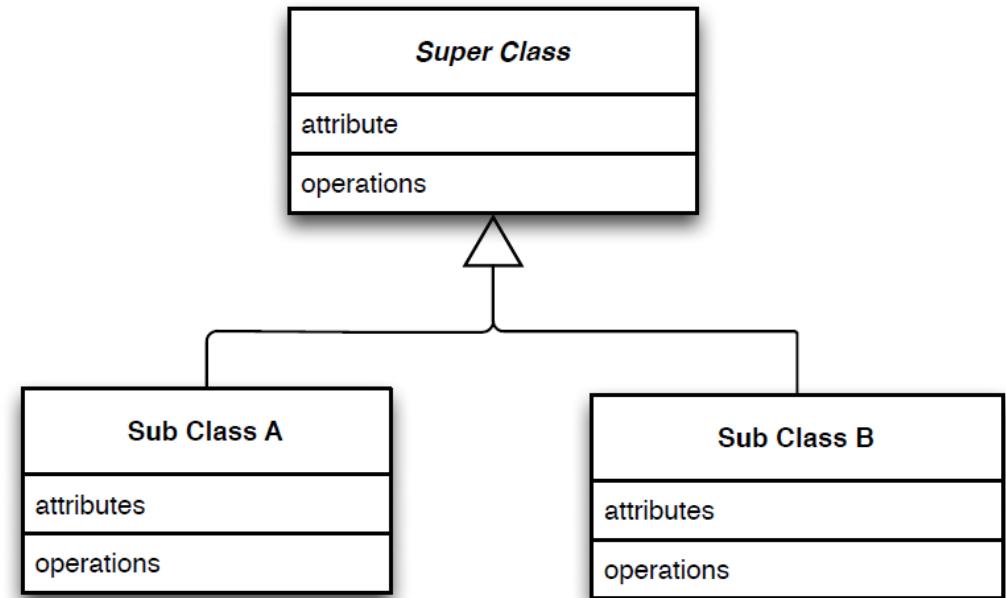
- Vielleicht schon gehört ?

- Encapsulation and information hiding
- Open-Closed principle (OCP)
- Single responsibility principle (SRP)
- Liskov substitution principle (LSP)
- Don't repeat yourself (DRY)
- Law of Demeter (LoD)
- Dependency Injection or Inversion of Control (IoC)

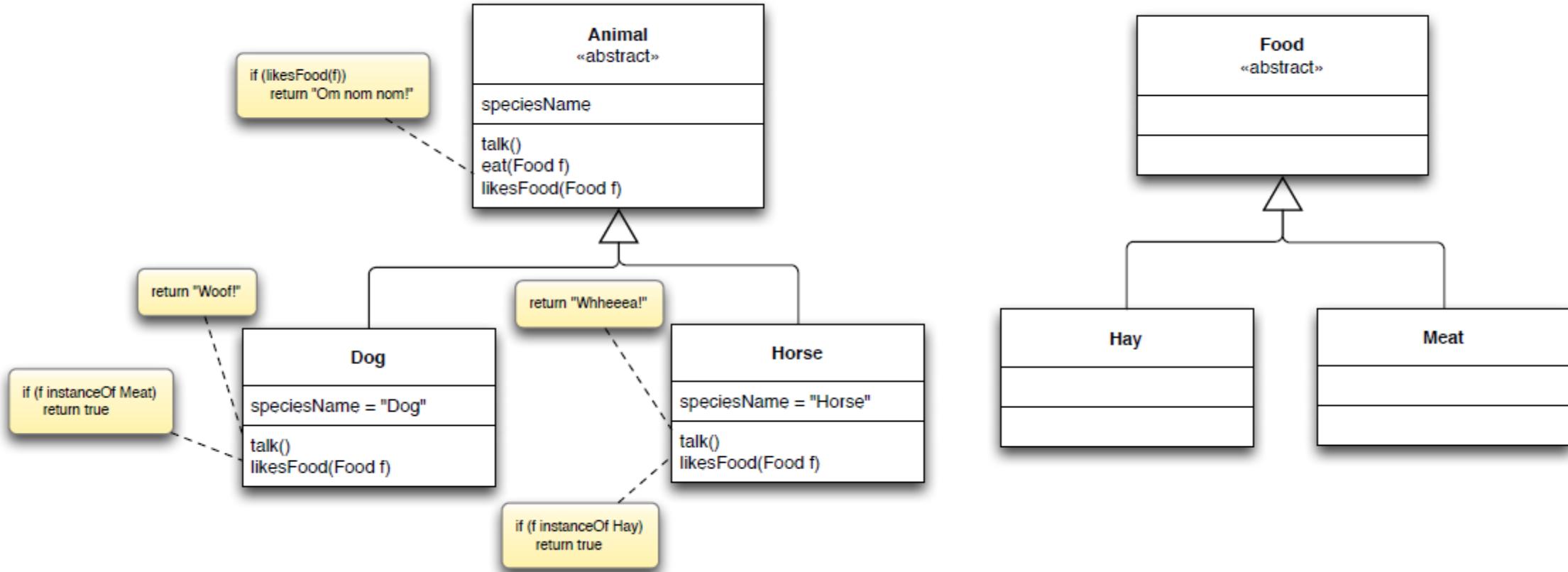
# Grundlagen + benötigte Kenntnisse



PersonA : User



# Grundlagen + benötigte Kenntnisse



# Aufgabe (Teamarbeit)

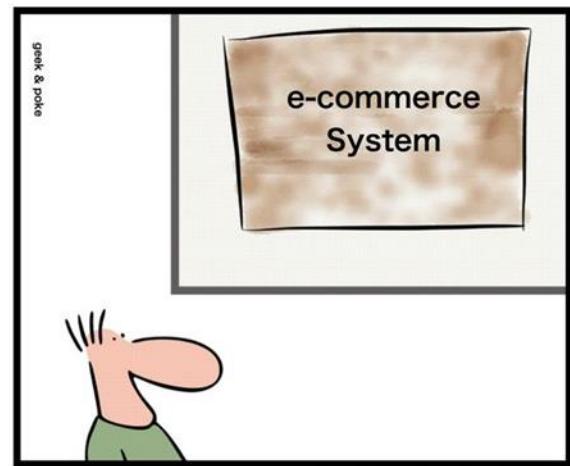
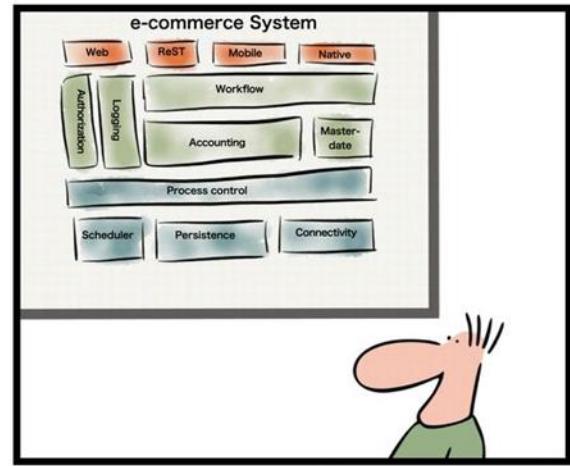
Was ist Polymorphismus ?

- Definition / Erklärung
- 1 Beispiel (lauffähig)

# Inhaltsverzeichnis

- 1) Die Bedeutung von Softwarearchitektur
- 2) Softwarearchitektur in Unternehmen
- 3) Die Domäne
- 4) Software Qualitätsattribute
- 5) Softwarearchitektur Design
- 6) Software Entwicklung Prinzipien und Praktiken
- 7) Softwarearchitektur Patterns
- 8) Moderne Architekturen
- 9) Performance
- 10) Security
- 11) Dokumentation
- 12) DevOps und Softwarearchitektur
- 13) Die Skills eines Softwarearchitekten
- 14) Evolutionäre Architekturen
- 15) Wie werde ich ein guter Softwarearchitekt
- 16) Architektur und Legacy Applications

HOW TO DRAW THE ARCHITECTURE OF YOUR SYSTEM



# 1) Die Bedeutung von Software Architektur

- Alles beginnt mit einer Definition
- Warum spielt Softwarearchitektur eine wichtige Rolle in der Softwareentwicklung ?
- Warum hat es Vorteile ein gutes Architekturdesign zu haben ?

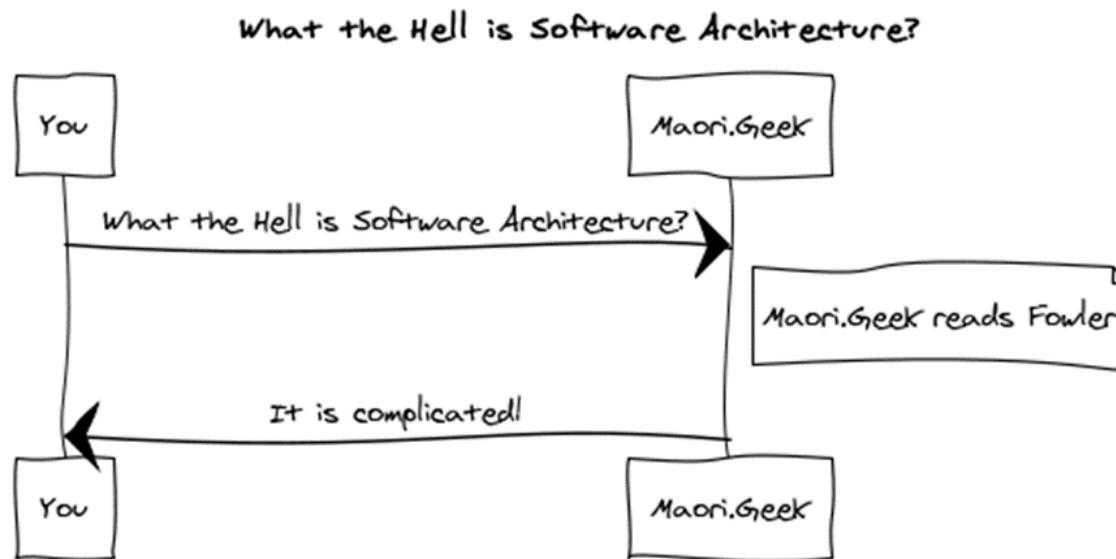
Wir werden folgendes betrachten

- a) Was ist Softwarearchitektur ?
- b) Warum ist Softwarearchitektur so wichtig ?
- c) Wer benötigt Softwarearchitektur ?
- d) Was ist die Rolle des Softwarearchitekten ?

# 1a) Was ist Softwarearchitektur ?

- Software Architektur ist die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehung zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.

IEEE Standard 1471



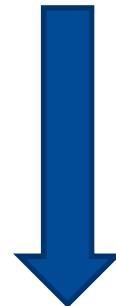
# 1a) Was ist Softwarearchitektur ?

- Der IEEE Standard beleuchtet die folgenden Hauptkriterien
  - ▶ Die grundsätzliche Organisation eines Systems
  - ▶ Ein System befindet sich in einer Umgebung
  - ▶ Eine Architekturdokumentation beschreibt die Architektur und kommuniziert an sog. Stakeholder wie die Architektur die Anforderungen des Systems abdeckt
  - ▶ Architektursichten werden aus der Beschreibung erzeugt und jede Sicht beschreibt ein Anliegen der Stakeholder
  - ▶ Eine Architektur ist eine Abstraktion
  - ▶ Eine Architektur beschreibt Strukturen von Systemen

# 1a) Was ist Softwarearchitektur ?

- Martin Fowler erklärt das so (<https://www.youtube.com/watch?v=DngAZyWMGR0>):

- The set of design decisions that must be made early
- The decisions that you wish you could get right early
- The decisions that are hard to change



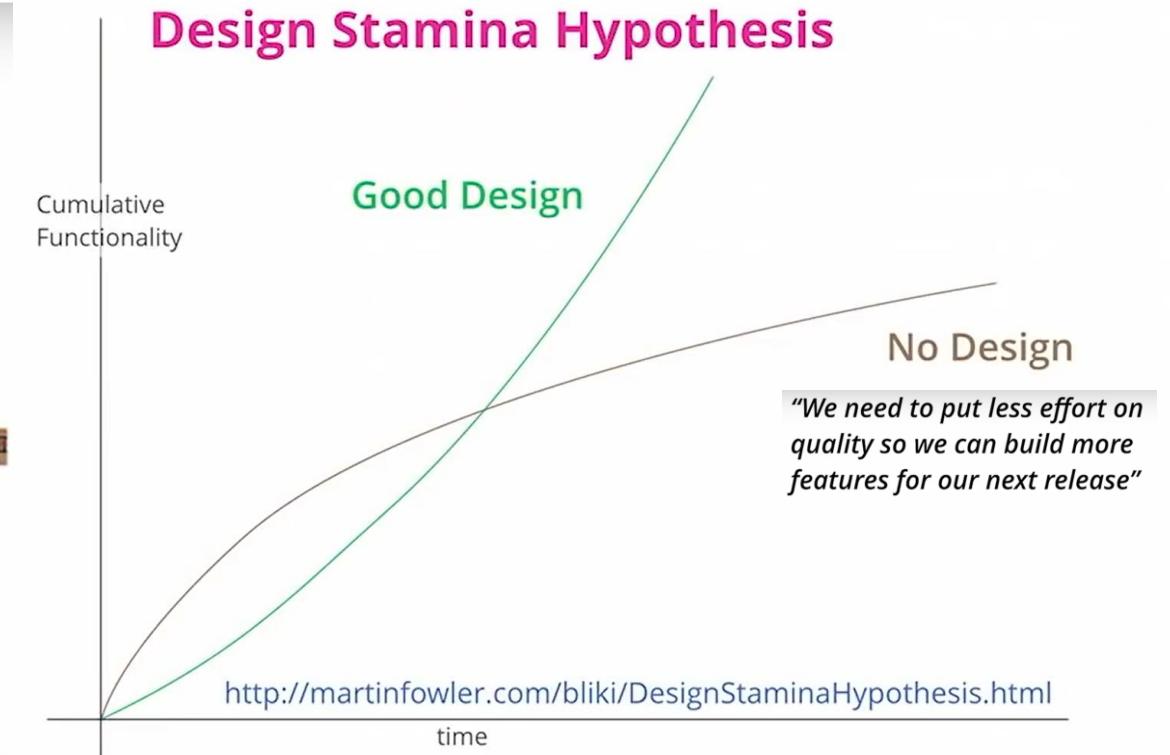
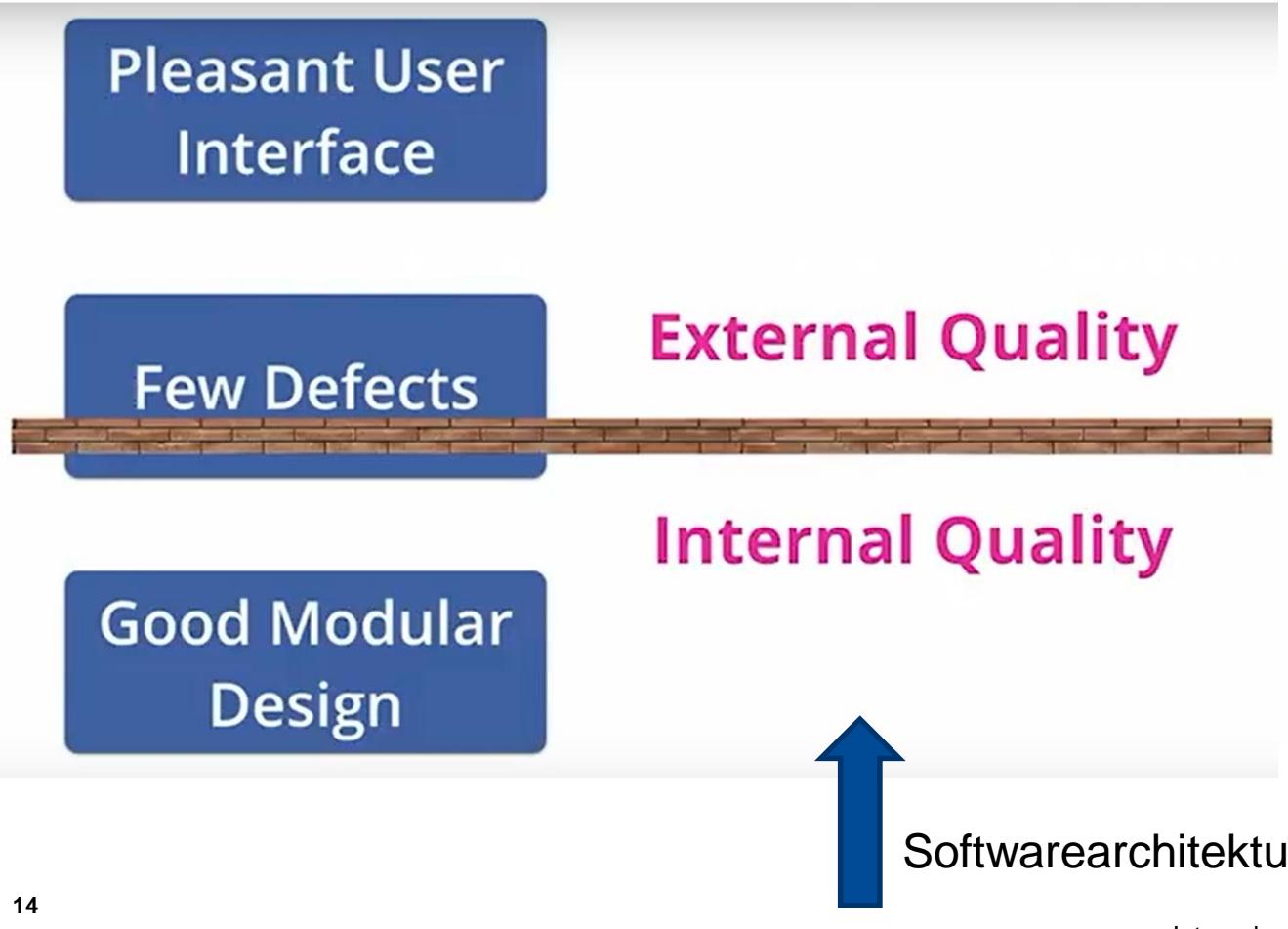
„Architecture is about the important stuff. Whatever that is.“

Ralph Johnson (Autor von *Design Patterns: Elements of Reusable Object-Oriented Software*)

# 1b) Warum ist Softwarearchitektur so wichtig ?

- Softwarearchitektur ist die Basis eines Software Systems
  - ▶ Hat direkte Auswirkung auf die Qualität eines Systems, auf die Entwicklung und die Wartung
  - ▶ Softwarearchitektur ist eine Reihe von Entscheidungen – frühere Entscheidungen beeinflussen dabei spätere Entscheidungen
  - ▶ Alle Software Systeme haben eine Architektur, auch wenn sie weder explizit getan noch dokumentiert wurde
  - ▶ Je größer und komplexer Software Systeme werden/sind, um so wichtiger ist eine ausgereifte und gut überlegte Architektur
  - ▶ Softwarearchitektur ist die Grundlage für ein erfolgreiches Software System

# 1b) Warum ist Softwarearchitektur so wichtig ?



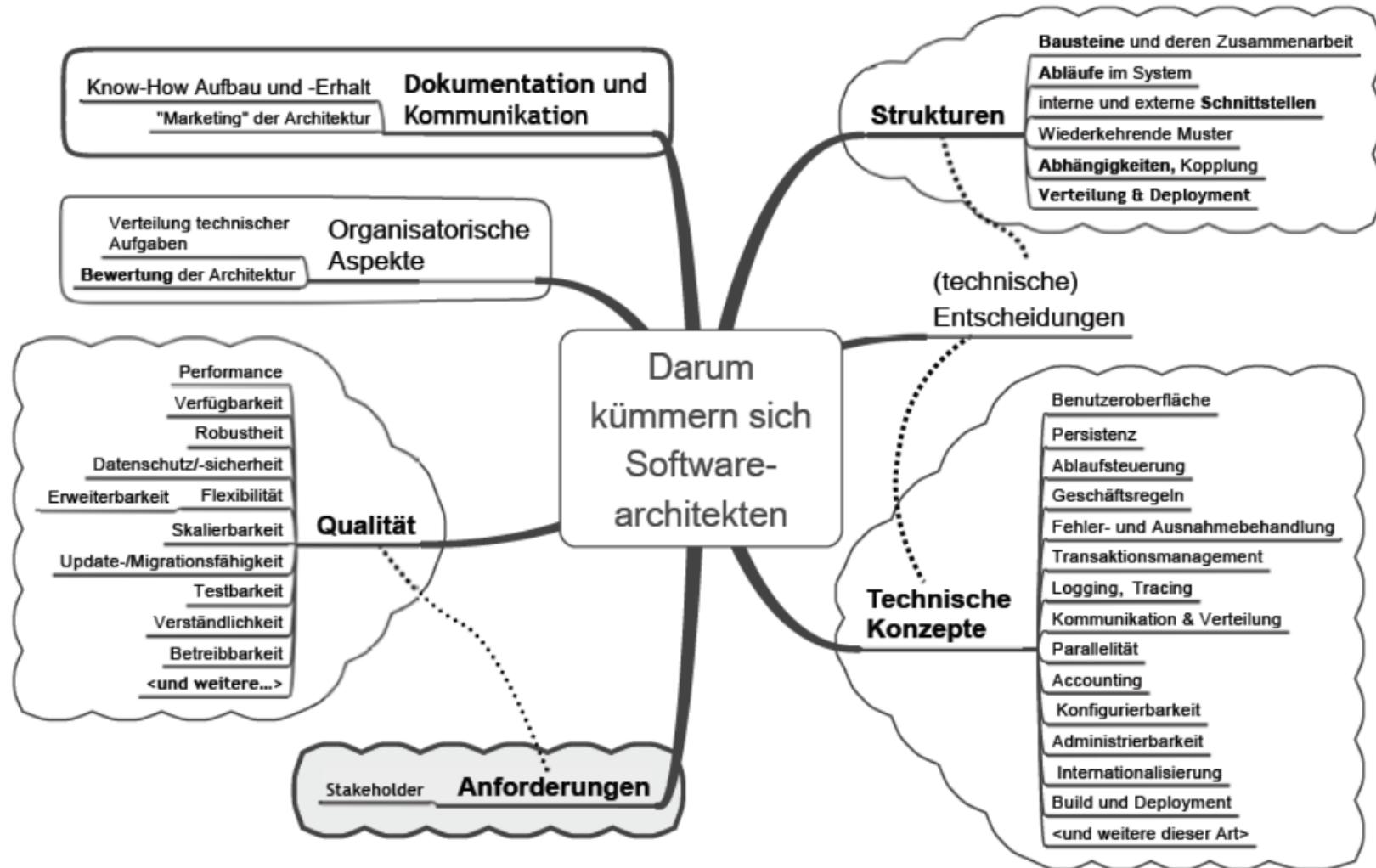
# 1b) Warum ist Softwarearchitektur so wichtig ?

- Software versucht alle funktionalen, nicht-funktionalen, technischen und betriebsbedingten Anforderungen zu erfüllen
- Anforderungen werden in der Arbeit mit Stakeholdern (Domain Experten, Business Analysten, Produkt Owners, End user) erfasst
- Eine Softwarearchitektur definiert eine Lösung, die diese Anforderungen erfüllt
- Software Systeme ohne Architektur erfüllen deshalb oft nicht die Anforderungen
- Software Systeme mit schlechter Architektur verfehlten die Qualitätsziele und sind typischerweise schwerer zu warten, zu verteilen und zu betreiben
- Softwarearchitektur liefert ein wiederverwendbares Modell (Entscheidungen, Code) und kann daher zu Kosten- und Resourceneinsparungen beitragen
- Gute Softwarearchitektur wird oft als Trainingsmodell verwendet

# 1c) Wer benötigt Softwarearchitektur ?

- Folgende Stakeholder benötigen Softwarearchitektur
  - ▶ End user
  - ▶ Business Analysten
  - ▶ Domain Experten
  - ▶ Qualitätskontrolle
  - ▶ Manager
  - ▶ Systemintegratoren
  - ▶ Softwareentwickler
  - ▶ Betriebseinheiten

# 1d) Was ist die Rolle des Softwarearchitekten ?



# 1d) Was ist die Rolle des Softwarearchitekten ?



- Nicht-technische Aufgaben
  - ▶ Führung
  - ▶ Begleitung in Projekten, incl. Kosten- und Aufwandsschätzung
  - ▶ Mentor für Kollegen
  - ▶ Hilfe bei der Auswahl von Teammitgliedern
  - ▶ Verständnis für Business Domänen
  - ▶ Unterstützung bei der Erfassung und Analyse von Anforderungen
  - ▶ Kommunikation mit einer Reihe von technischen und nicht-technischen Stakeholdern
  - ▶ Vision für zukünftige Produkte

# 1d) Was ist die Rolle des Softwarearchitekten ?



## ■ Technische Aufgaben

- ▶ Nicht-funktionale Anforderungen und Qualitätsattribute verstehen
- ▶ Das Design von Softwarearchitekturen beherrschen
- ▶ Patterns und Best Practices der Softwareentwicklung beherrschen
- ▶ Tiefes Wissen über Softwarearchitektur Patterns, Vor- und Nachteile und das Wissen welches Pattern wann am besten geeignet ist
- ▶ Erfahrung beim Bearbeiten von Querschnittsthemen
- ▶ Sicherstellen, daß Performance und Security Anforderungen abgedeckt sind
- ▶ Softwarearchitekturen dokumentieren und prüfen können
- ▶ Ein Verständnis für DevOps und Deployment Prozesse besitzen
- ▶ Wissen für die Integration und das Arbeiten mit Legacy Applikationen besitzen
- ▶ Adaptive Softwarearchitekturen beherrschen

## 2) Software Architektur in einer Organisation

- Software Systeme erfüllen Ziele des Business
- Softwarearchitekten sind Teil einer Organisation
- Organisationen beeinflussen Softwarearchitekten an vielen Stellen
- Welche typischen Ausprägungen findet man in Organisationen

Wir werden folgendes betrachten

- a) Typen von Softwarearchitekten
- b) Softwareentwicklungsmethoden
- c) Projektmanagement
- d) Politik
- e) Software Risikomanagement

## 2a) Typen von Softwarearchitekten

- Business Architect
- Enterprise Architect
- Solution Architect
- Application Architect
- Data/Information Architect
- Infrastructure Architect
- Information Security Architect
- Cloud Architect

## 2a) Typen von Softwarearchitekten

### Enterprise Architect

- The organizing logic for business processes and IT infrastructure reflecting the integration and standardization requirements of the firm's operating model.  
[Source: MIT Center for Information Systems Research]
- A conceptual blueprint that defines the structure and operation of an organization. The intent of an enterprise architecture is to determine how an organization can most effectively achieve its current and future objectives.  
[Source: SearchCIO.com]
- Enterprise architecture (EA) is a discipline for proactively and holistically leading enterprise responses to disruptive forces by identifying and analyzing the execution of change toward desired business vision and outcomes. EA delivers value by presenting business and IT leaders with signature-ready recommendations for adjusting policies and projects to achieve target business outcomes that capitalize on relevant business disruptions.  
[Source: Gartner.com]

## 2a) Typen von Softwarearchitekten

### Enterprise Architect

- **Vergleich mit einer Stadt**

- Straßen
- Brücken
- Versorgungsleitungen
  - Strom
  - Wasser/Gas
  - Kommunikation
- Abwasserleitungen
- Baugebiete
  - Wohngebiete
  - Industriegebiete
  - Einkaufszentren
- Autobahnabbindungen
- Bahnhöfe
- Flughäfen
- Schiffshäfen
- Kläranlagen
- Deponien

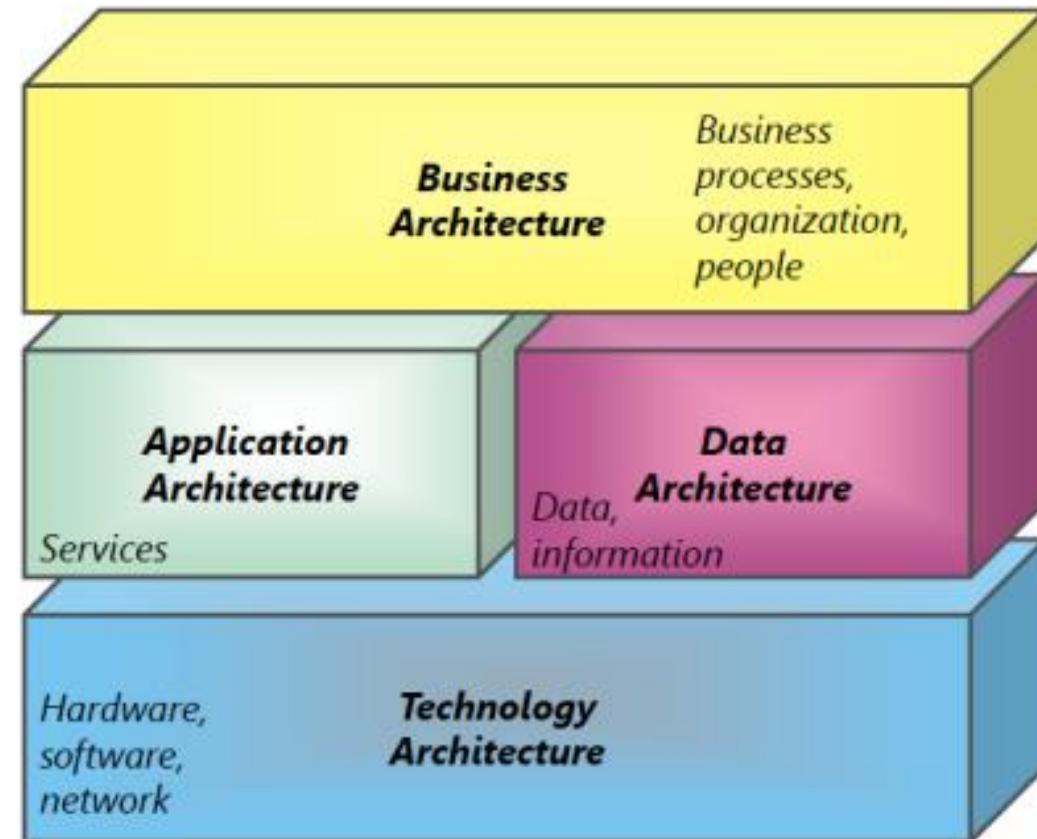


## 2a) Typen von Softwarearchitekten

### Enterprise Architect

- **4 betrachtete Domänen**

- Business Architektur
- Anwendungsarchitektur
- Data-/Informationsarchitektur
- Technologiearchitektur

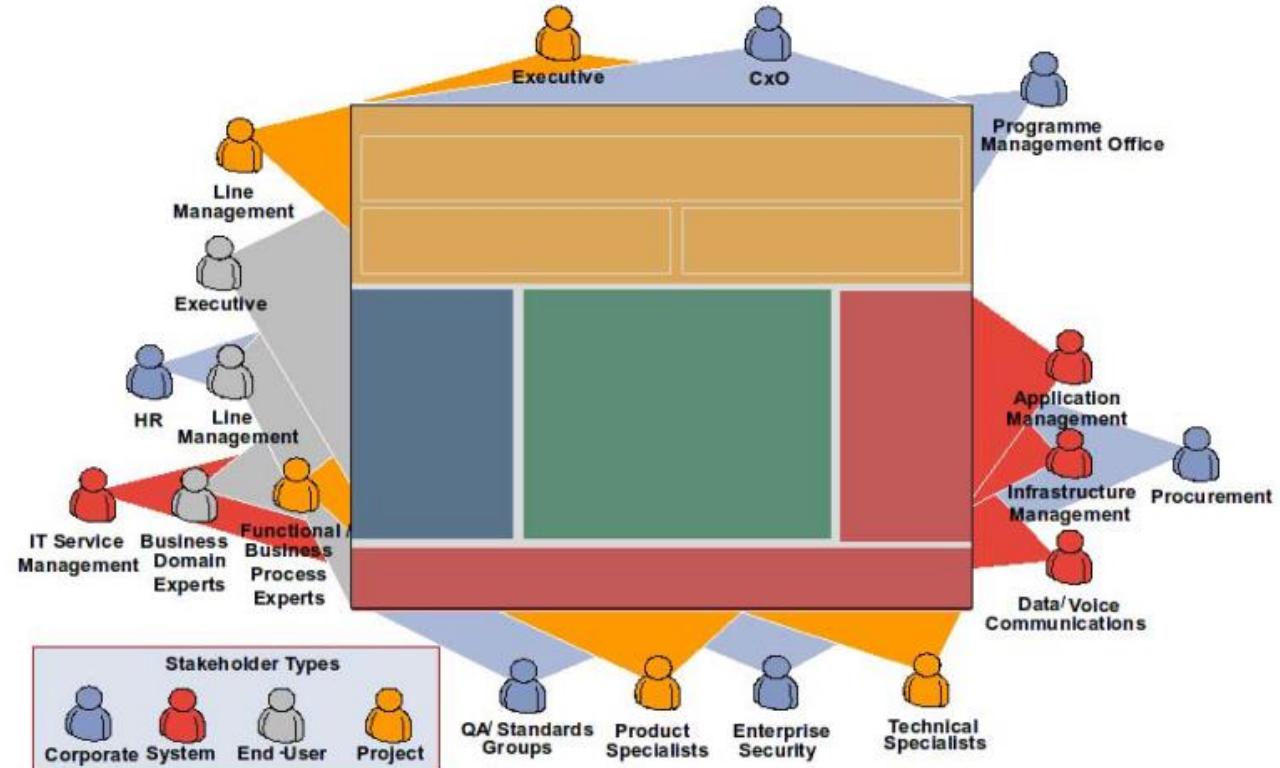


## 2a) Typen von Softwarearchitekten

### Enterprise Architect

- **Warum Enterprise Architektur**

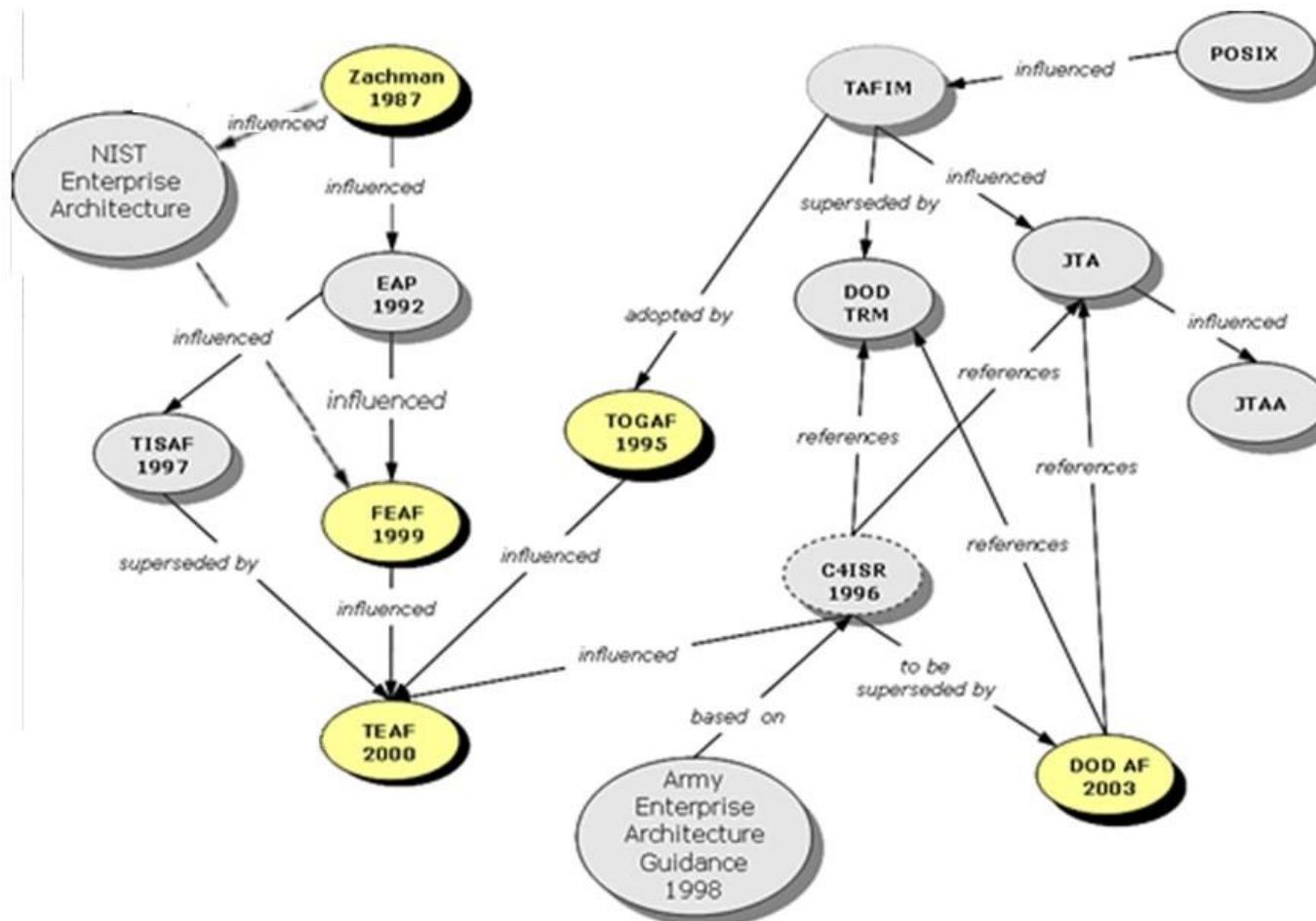
- Agilität und Ausrichtung
  - Strategie
  - Motive
  - Treiber
- Einflüsse von außen (Digitalisierung)
  - Schnelle Reaktion
  - Schnelle Anpassungen
  - Neue Geschäftsmodelle
- Effizientes IT Management
  - Blueprint der IT Landschaft
  - IT Kostenoptimierung
- Bessere Planung und Kommunikation
  - Prioritäten
  - Projekte
  - Prozesse
  - Gemeinsame Sprache



## 2a) Typen von Softwarearchitekten

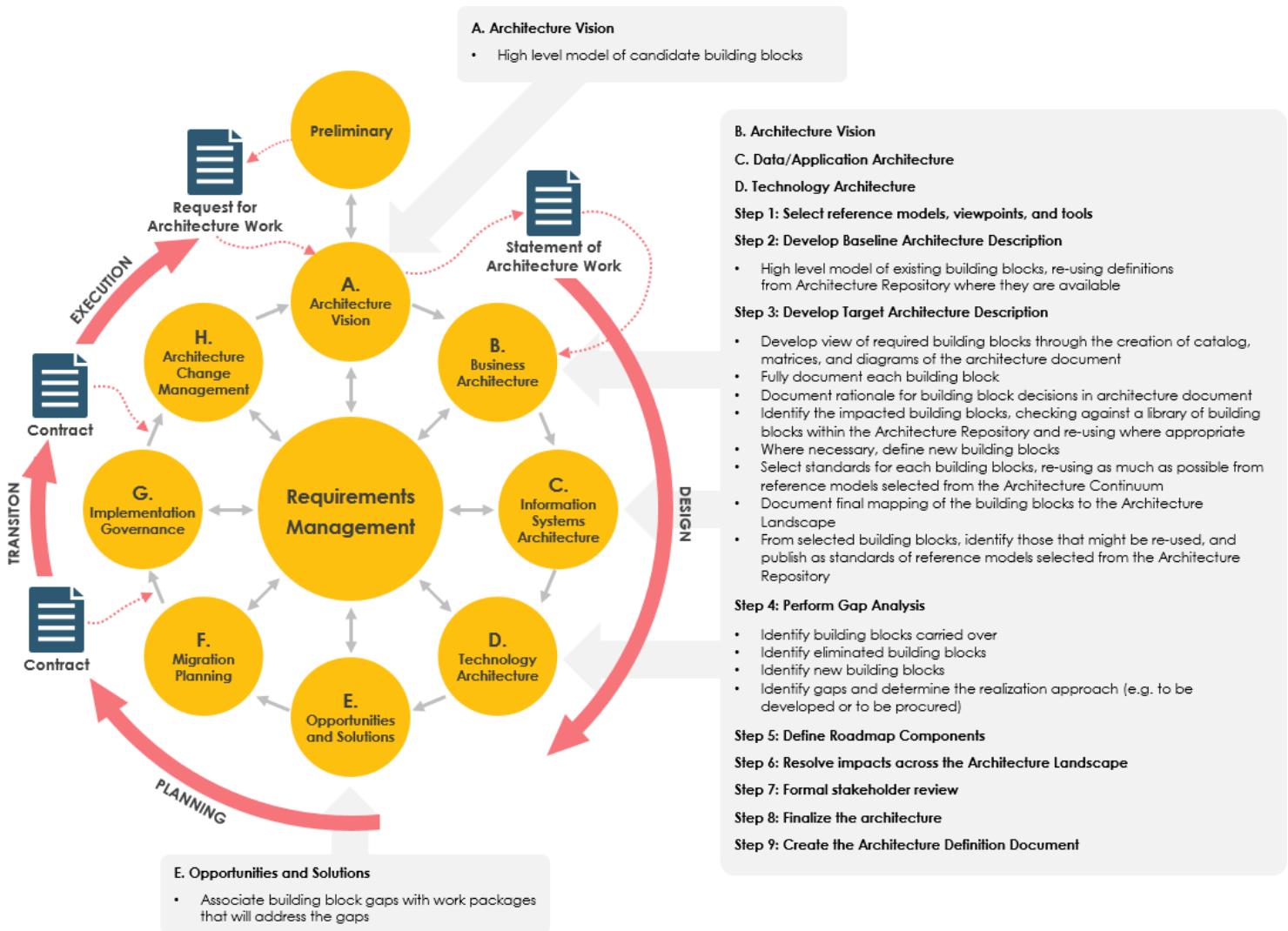
### Enterprise Architect

- Frameworks

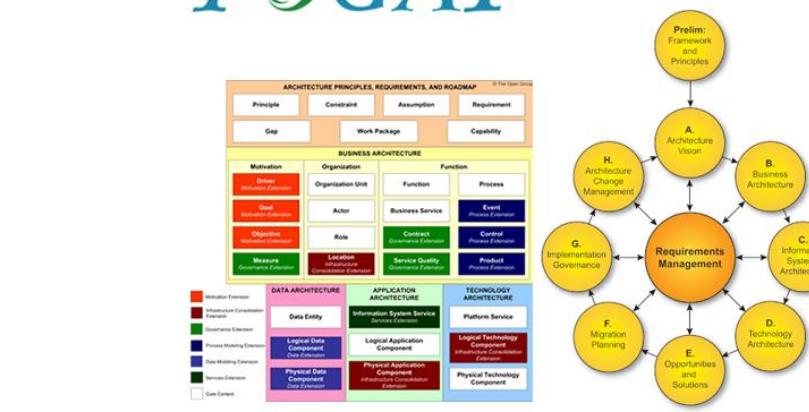
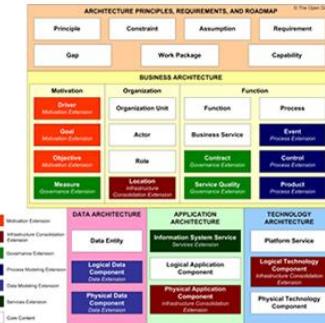


# 2a) Typen von Softwarearchitekten

## Enterprise Architect



# TOGAF®



## 2a) Typen von Softwarearchitekten

### Solution Architect

- Überträgt Anforderungen in eine Lösungsarchitektur
  - ▶ Arbeiten dafür mit Business Analysten und Product Owners zusammen
- Wählen geeignete Technologien für die Problemlösung
  - ▶ Arbeiten mit Enterprise Architekten zusammen
- Bilden die Brücke zwischen Enterprise Architekten und Application Architekten

## 2a) Typen von Softwarearchitekten

### Application Architect

- Auf Anwendungsarchitektur von Anwendungen fokussiert
  - ▶ Das Design der Anwendung muss den Anforderungen genügen
- Sind das Bindeglied zwischen den IT Techikern und den Anwendern
- Sind in alle Phasen der Softwareentwicklung involviert
- Sie schlagen Lösungen oder Technologien vor und evaluieren Alternativen
- Stellen sicher, daß Entwicklungsteams „Best practices“ und Standards folgen
- Sie geben Orientierung, führen und beraten Entwicklerteams
- Sie sind in Code Reviews involviert
- Arbeiten mit Enterprise Architekten zusammen, um die strategische Ausrichtung zu garantieren

## 2a) Typen von Softwarearchitekten

### Data/Information Architect

- Auf Datenarchitekturen von Organisationen fokussiert
  - ▶ Sorgen für die Bereitstellen von Daten für entsprechende Konsumenten
- Sind für alle Datenquellen einer Organisation zuständig
- Sorgen für die Erfüllung strategischer Anforderungen an Daten
- Sie erstellen Designs und Modelle
- Sie definieren die Speicherung, die Verarbeitung und die Integration von Daten in den zahlreichen Anwendungen einer Organisation
- Sie sorgen für Datensicherheit und Zugriffsschutz von Daten
  - ▶ Backup Prozesse, Archivierungsszenarien, Datenbank Recovery Szenarien
- Unterstützen Entwickler bei Datenbankentwicklung
- Helfen bei Problemen mit Performance und Störungen im Betrieb
- Information Architects haben den Fokus mehr auf Benutzern
  - ▶ User Experience, Usability

## 2a) Typen von Softwarearchitekten

### Infrastructure Architect

- Sie sind verantwortlich für die Infrastruktur
  - ▶ Server
  - ▶ Netzwerk
  - ▶ Speichersysteme
  - ▶ Einrichtungen (Gebäude, Räume, Schränke, Stromversorgung, Kühlung, physikal. Sicherheit, ...)
- Sie überwachen die Infrastruktur
  - ▶ Workload, Durchsatz, Latency, Kapazitäten, Redundanz
  - ▶ Sie nutzen Infrastruktur Management Tools

## 2a) Typen von Softwarearchitekten

### Information Security Architect

- Sie sind verantwortlich für die Computer und Netzwerk Sicherheit
  - ▶ Dafür müssen sie die Infrastruktur in allen Komponenten gut kennen und verstehen
- Sie führen Security Assessments durch und testen die Infrastruktur auf Angreifbarkeit
- Sie erkennen Sicherheitslücken und helfen bei der Behebung
- Sie kennen sich mit Security Standards, Best Practices und Technologien sehr gut aus
- Sie helfen bei der Behebung von Security Incidents und bei der Abwehr von Angriffen
- Sie helfen bei Security Awareness Programmen und bei der Erstellung von Security Policies und Prozeduren

## 2a) Typen von Softwarearchitekten

### Cloud Architect

- Sie sind verantwortlich für die Cloud Strategie/Initiativen eines Unternehmens
- Sie sorgen für eine erfolgreiche Einführung und Nutzung von Cloud Plattformen
- Sie wählen Cloud Anbieter und entsprechende Nutzungsmodelle aus (IaaS, SaaS, PaaS)
- Sie erstellen Cloud Migrationspläne für bestehende Anwendungen
- Sie unterstützen bei der Entwicklung neuer Cloud-native Anwendungen
- Sie erstellen Regelwerke, überwachen und managen Cloud Deployments
- Sie benötigen ein tiefes Verständnis für Security Fragen
  - ▶ Sie arbeiten eng mit Security Architekten zusammen
- Sie helfen beim Umdenken von lokalen Anwendungen hin zu Anwendungen in der Cloud (cultural change)

## 2b) Softwareentwicklungsmethoden

- Typischerweise hat jede Organisation ihre eigenen Softwareentwicklungsmethoden
- Softwarearchitekten spielen oft eine Rolle bei der Auswahl geeigneter Methoden
- Ein guter Überblick über gängige Methoden ist von Vorteil
  - ▶ Es gibt unterschiedlichste Methoden – jede von ihnen hat Stärken und Schwächen
  - ▶ Heute werden in den meisten Fällen Agile Methoden benutzt
  - ▶ Es gibt nicht die eine agile Methode, sondern unzählige Variationen
  - ▶ Die Auswahl einer geeigneten Methode bestimmt den Erfolg eines Softwareprojekts
- Wir betrachten die 2 am meisten genutzten Methoden
  - ▶ Wasserfall
  - ▶ Agile

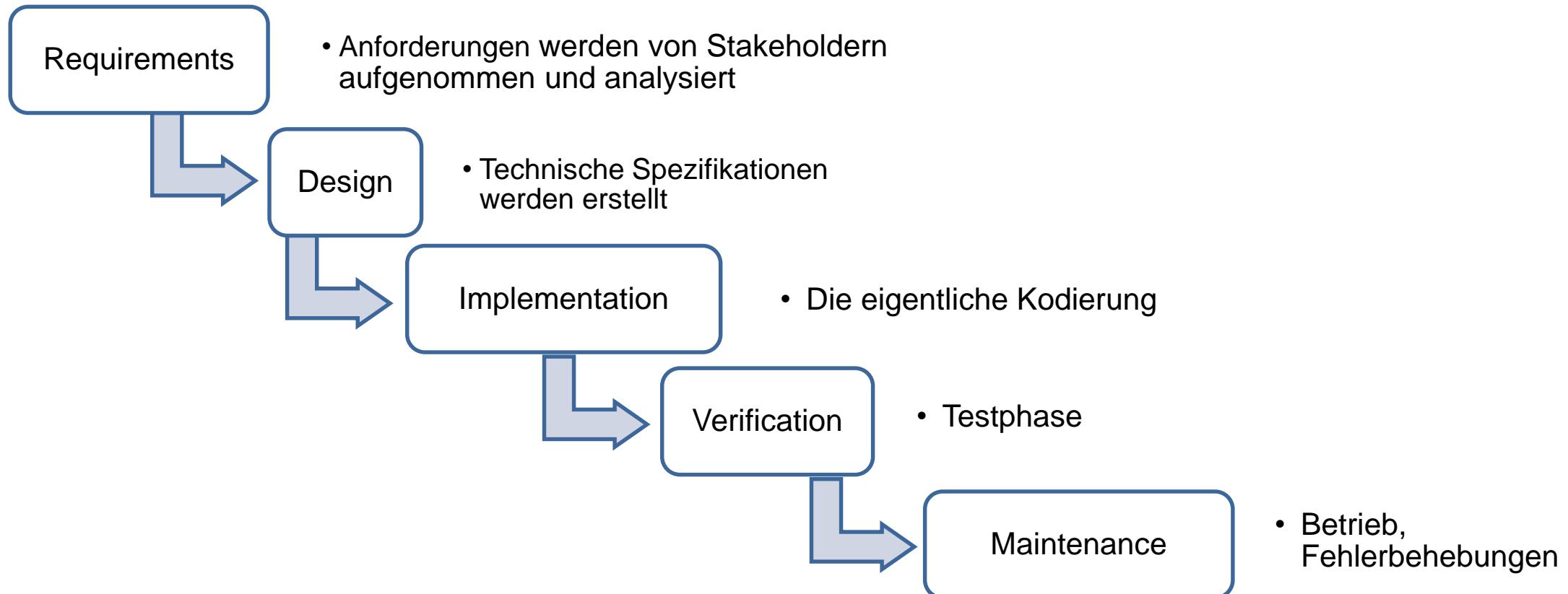
## 2b) Softwareentwicklungsmethoden

### Das Wasserfall Modell

- Das Wasserfall Modell ist ein sequentielles Modell
- Es besitzt einen Lebens Zyklus mit mehreren Stufen, die jeweils komplett abgearbeitet sein müssen, bevor zur nächsten Stufe weiter gegangen werden kann
- Vorteile vom Wasserfall Modell
  - ▶ Einfach anwendbar und verständlich
  - ▶ Stakeholder sehen zu jeder Zeit, was zeitlich, funktional und kostenmäßig erwartet werden kann
  - ▶ Artifakte, die in jeder Phase erzeugt werden müssen, sorgen für eine gute Dokumentation der einzelnen Teile des Gesamtsystems  
Diese Dokumentation unterstützt den späteren Betrieb des Systems  
Diese Dokumentation sorgt auch dafür, daß zu jedem Zeitpunkt neue Mitarbeiter ins Projekt geholt werden können (ohne große Einarbeitung)

## 2b) Softwareentwicklungsmethoden

### Das Wasserfall Modell



## 2b) Softwareentwicklungsmethoden

### Das Wasserfall Modell

- Das Wasserfall Modell ist auch heute noch ein Modell der Wahl, wenn
  - ▶ Das Entwicklungsteam die Business Domäne, die Business Regeln und die Funktionalitäten sehr gut kennt
  - ▶ Die Anforderungen klar verstanden wurden und es sicher ist, dass sie sich nicht mehr ändern
  - ▶ Der Umfang/die Zielsetzung (Scope) des Projektes gesetzt ist und stabil bleibt
  - ▶ Die benutzten Technologien und die Architektur beherrscht werden und sich nicht mehr ändern
  - ▶ Es ist ein überschaubares Projekt mit geringer Komplexität
  - ▶ Es ist akzeptierbar, dass ein Ergebnis erst zu einer späten Phase zur Verfügung steht
  - ▶ Das Projekt hat eine bereits beschlossene Dauer und/oder Deadline

## 2b) Softwareentwicklungsmethoden

### Das Wasserfall Modell

- Das Wasserfall Modell hat auch Nachteile
  - ▶ Testen beginnt erst nach der gesamten Implementierung
  - ▶ Eine lauffähige Version der Software ist erst spät verfügbar
  - ▶ Oft gibt es nach der Fertigstellung Diskussionen um die Funktionalität
  - ▶ Es kommt oft vor, daß Anforderungen erst spät bekannt werden
  - ▶ Bei zeitlichen Engpässen leidet vor allem die Testphase (Qualitätsprobleme)
  - ▶ Sich ändernde Anforderungen während des Projektes können kaum abgebildet werden
  - ▶ Die rigide Vorgehensweise führt oft zu Spannungen im Entwicklungsteam
  - ▶ Entwickler und Tester sind oft die gleichen Personen
  - ▶ Viele parallele Wasserfall Projekte streiten sich um Ressourcen

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

- Agile Softwareentwicklungsmodelle sind aus der Erfahrung mit Einschränkungen entstanden
- Sie sind heute die populärsten Modelle
- Es gibt verschiedene Modelle wie
  - ▶ Kanban
  - ▶ Scrum
  - ▶ Extreme Programming (XP)
  - ▶ Crystal
- Alle sind verschieden voneinander, aber haben eines gemeinsam
  - ▶ Die Anpassungsfähigkeit an Änderungen

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

- Agile Werte und Prinzipien
  - ▶ Alle agilen Modelle haben Werte und verfolgen Prinzipien
  - ▶ Diese sind dokumentiert in
    - Den 4 Werten und
    - Den 12 Prinzipien des
  - ▶ „Agile Manifesto“ (<http://agilemanifesto.org/>)

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

- Die 4 Werte
  - ▶ Individuals and Interactions Over Processes and Tools  
Individuen verstehen das Business und treiben Entwicklungen (auch von Software)
  - ▶ Working Software Over Comprehensive Documentation  
nur soviel Dokumentation wie nötig
  - ▶ Customer Collaboration Over Contract Negotiation  
der Kunde wird ein entscheidender Teil des Entwicklungsprozesses
  - ▶ Responding to Change Over Following a Plan  
kurze Iterationen in der Entwicklung lassen Änderungen jederzeit zu

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

#### ■ Die 12 Prinzipien

- ▶ Customer satisfaction through early and continuous software delivery  
lauffähige Software in regulären Intervallen
- ▶ Accommodate changing requirements throughout the development process  
keine Verzögerungen bei Änderungsanforderungen
- ▶ Frequent delivery of working software  
Nutzung von Sprints oder Iterationen
- ▶ Collaboration between the business stakeholders and developers throughout the project  
Wenn Kunde und technisches Team zusammen arbeiten
- ▶ Support, trust, and motivate the people involved  
motivierte Teams sind die halbe Miete

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

#### ■ Die 12 Prinzipien

- ▶ Enable face-to-face interactions  
Zusammen sitzen und arbeiten ist wichtig
- ▶ Working software is the primary measure of progress  
das einzige Maß ist lauffähige Software
- ▶ Agile processes to support a consistent development pace  
Die Geschwindigkeit der Entwicklung muss wiederholbar und beherrschbar sein
- ▶ Attention to technical detail and design enhances agility  
die richtigen Skills müssen vorhanden sein
- ▶ Simplicity  
Einfach, einfach, einfach

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

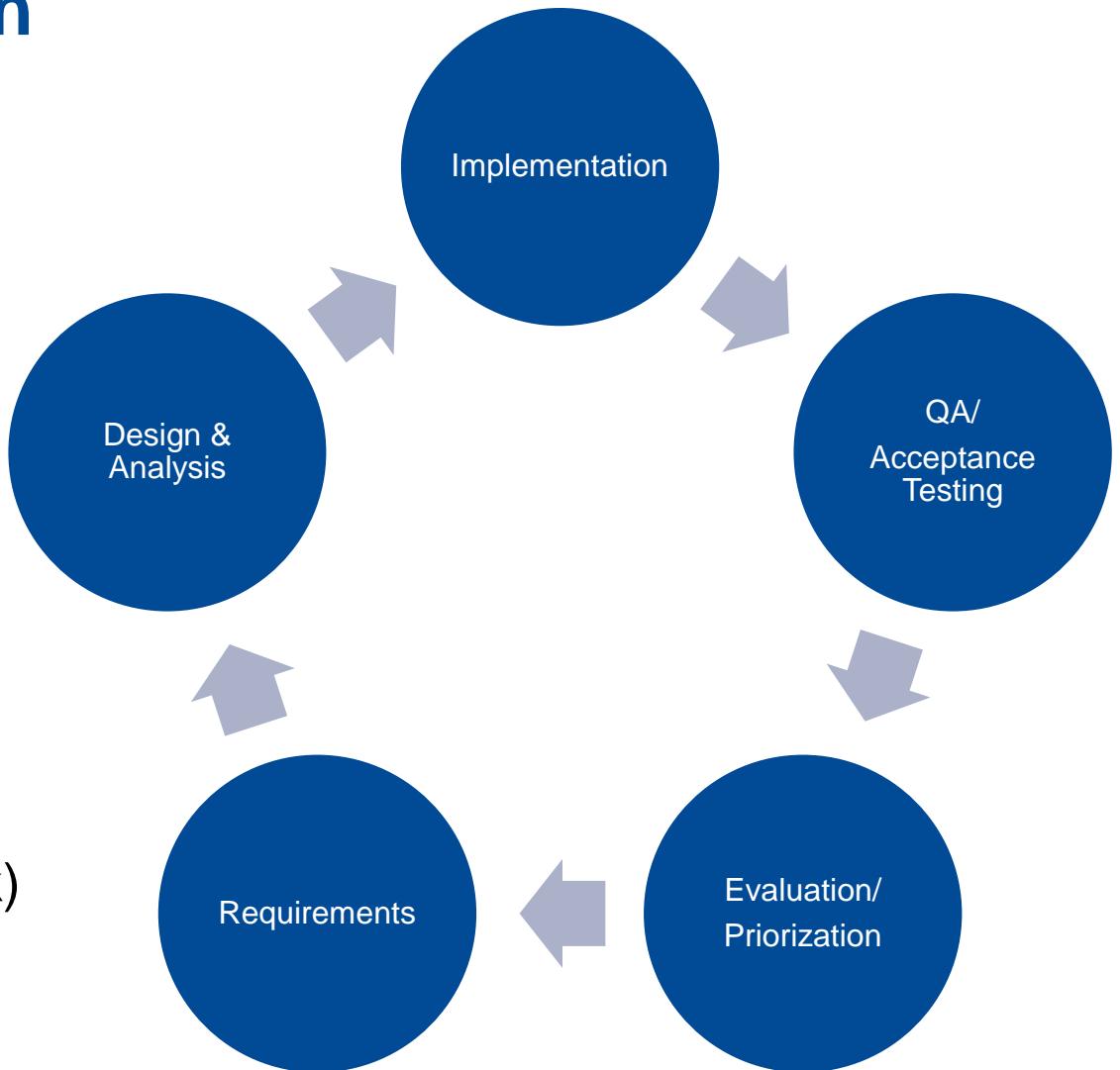
#### ■ Die 12 Prinzipien

- ▶ Self-organizing teams encourage great architectures, requirements, and designs  
Entscheidungen werden im Team getroffen, das Team übernimmt Verantwortung  
Entscheidungen und Ideen werden kommuniziert
- ▶ Regular reflections on how to become more effective  
Weiterbildung, Selbstverbesserung, Prozessverbesserung, Teamspirit

## 2b) Softwareentwicklungsmethoden

### Agile Modelle

- Eine iterative Methode
  - ▶ Software wird inkrementell entwickelt
  - ▶ Jede Iteration entspricht einem Teil der Anforderungen
  - ▶ Nach jeder Iteration ist eine lauffähige Version der Software verfügbar
  - ▶ Getestet wird in der gleichen Iteration
  - ▶ Dauernde Rückmeldung (continuous feedback) sorgt für Transparenz



## 2b) Softwareentwicklungsmethoden

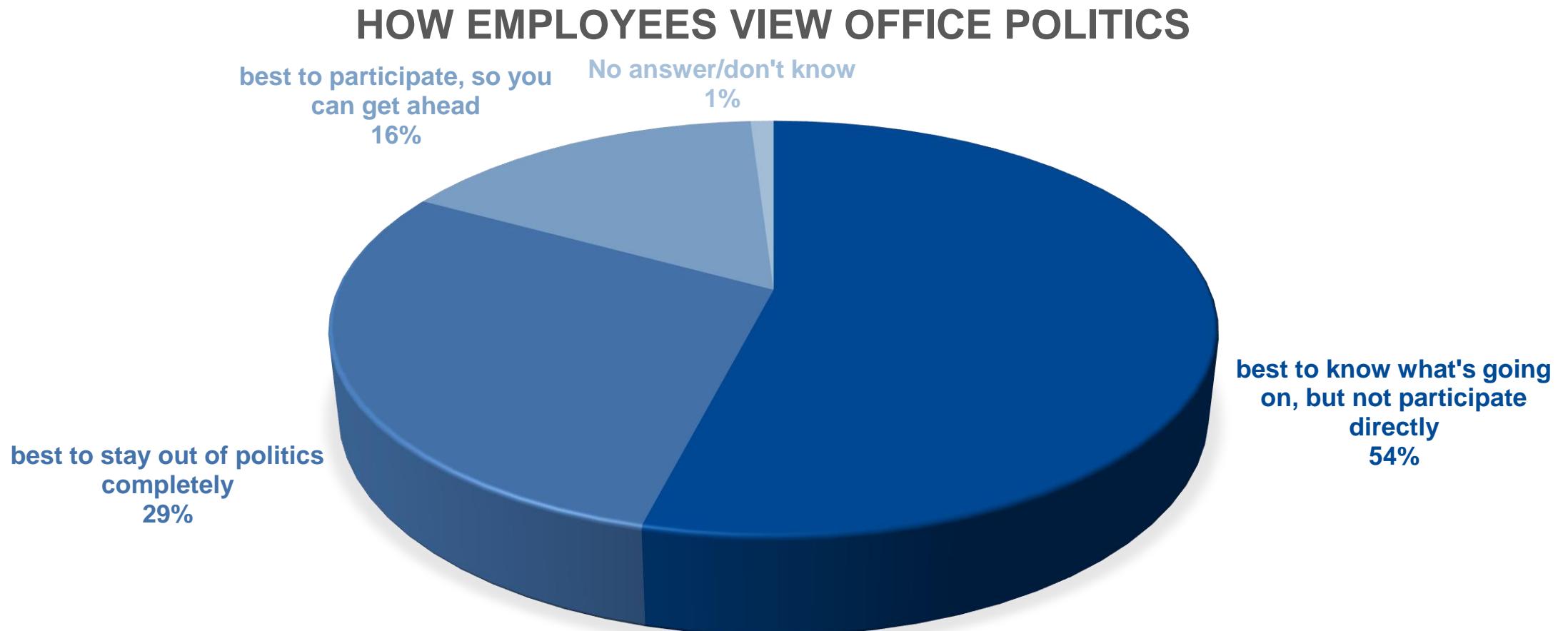
### Agile Modelle

- Anpassungsfähigkeit statt Vorhersehbarkeit
  - ▶ Agile Modelle sind auf die Anpassungsfähigkeit für Änderungen ausgelegt
  - ▶ Feedback unterstützt die Planung der nächsten Iteration
- Daily stand-up meetings
  - ▶ Gebräuchliche Praxis in agilen Teams
  - ▶ Ein sog. Facilitator leitet die Meetings
  - ▶ Was wird besprochen (was habe ich gestern getan, was mache ich heute, gibt es Hindernisse)
- Vorteile
  - ▶ Wissenstransfer
  - ▶ Hindernisse werden benannt, Hilfe wird möglich
  - ▶ Das Teamverständnis wird gefördert

## 2c) Projektmanagement

- Software Projektschätzungen sind wichtig  
Schätzungen und Planungen sind wichtige Erfolgsfaktoren
- Arbeit in Schätzungen zu stecken lohnt sich  
Proof of Concept (POC) hilft bei Einschätzungen
- Sei ein Realist (oder vielleicht sogar ein Pessimist)
- Beziehe das Team und begleitende Umstände mit ein
- Der Projektplan wird sich ändern und darauf muss reagiert werden
  - ▶ Mehrarbeit
  - ▶ Reduzierung des Umfangs
  - ▶ Hinzufügen von Ressourcen
  - ▶ Neuverteilung von Ressourcen
  - ▶ Problemfelder identifizieren
  - ▶ Reagiere zeitnah

## 2d) Politik



## 2d) Politik

- Software Architekten können sich nicht der Firmenpolitik entziehen
  - ▶ Kontakt und Kommunikation mit unterschiedlichsten Stakeholdern notwendig
  - ▶ In Organisation treffen unterschiedlichste Individuen mit unterschiedlichen Wissensständen, Hintergründen, Ansichten und Zielen aufeinander
  - ▶ Unterschiedlichste Faktoren können die Motivation von Personen beeinflussen
  - ▶ Das führt in den meisten Fällen zu Firmenpolitik
  - ▶ Um etwas zu erreichen, kann es notwendig werden „politisches Kapital“ auzugeben, zu leihen oder zu bekommen
  - ▶ Ein Softwarearchitekt will Lösungen schaffen und keine Politik betreiben, politisches Können auf diesem Parkett ist aber oft hilfreich
  - ▶ Im folgenden ein paar Punkte zur Beachtung

## 2d) Politik

- Die Ziele der Organisation verstehen
  - ▶ Die strategischen Ziele und Richtungen sollten verstanden sein
  - ▶ Die eigenen Ziele an die Ziele der Organisation anpassen, um Konflikte zu vermeiden
  - ▶ Zu wissen, wie die Organisation Geld macht und wie sie Geld investiert, ist hilfreich
    - Projektressourcen, Hard- und Software oder Lizenzen sind Investitionen
    - ROI (Return on Invest) Berechnungen sind notwendig (wie passt die Investition zu den Zielen)
- Die Bedenken Anderer adressieren
  - ▶ Viele Stakeholder werden Bedenken zu Technologien oder Prozessen artikulieren
  - ▶ Es ist notwendig, solche Bedenken aufzunehmen und das Gefühl zu geben, die Bedenken verstanden zu haben und sie in die Betrachtungen mit einzubeziehen
  - ▶ Wird das nicht getan, können oft auch unwichtige Themen unnötig „eskaliert“ werden

## 2d) Politik

- Andere bei Ihren Zielen unterstützen
  - ▶ Wenn immer möglich, andere bei Ihren Vorhaben unterstützen
  - ▶ Wenn Sie sich im Rahmen der Ziele und Werte des Unternehmens bewegen
  - ▶ Auch ohne Erwartung auf Gegenleistung
  - ▶ Hilfe von Anderen ist so leichter zu bekommen
- Bereit sein Kompromisse einzugehen
  - ▶ Nicht immer lässt sich alles einfach und ohne Kompromisse erreichen
  - ▶ Den geeigneten Zeitpunkt für einen Kompromiss finden, ist eine Kunst
  - ▶ Ein Kompromiss in einem Punkt hilft vielleicht ein anderes Ziel zu erreichen
  - ▶ Sich über mögliche Kompromisse im voraus Gedanken gemacht zu haben, hilft in Diskussionen
- Kulturelle Unterschiede beachten
  - ▶ Verschiedene Kulturen haben unterschiedliche Vorgehensweisen und Handlungen
  - ▶ Darauf einzugehen, ist von Vorteil – Lernen hilfreich

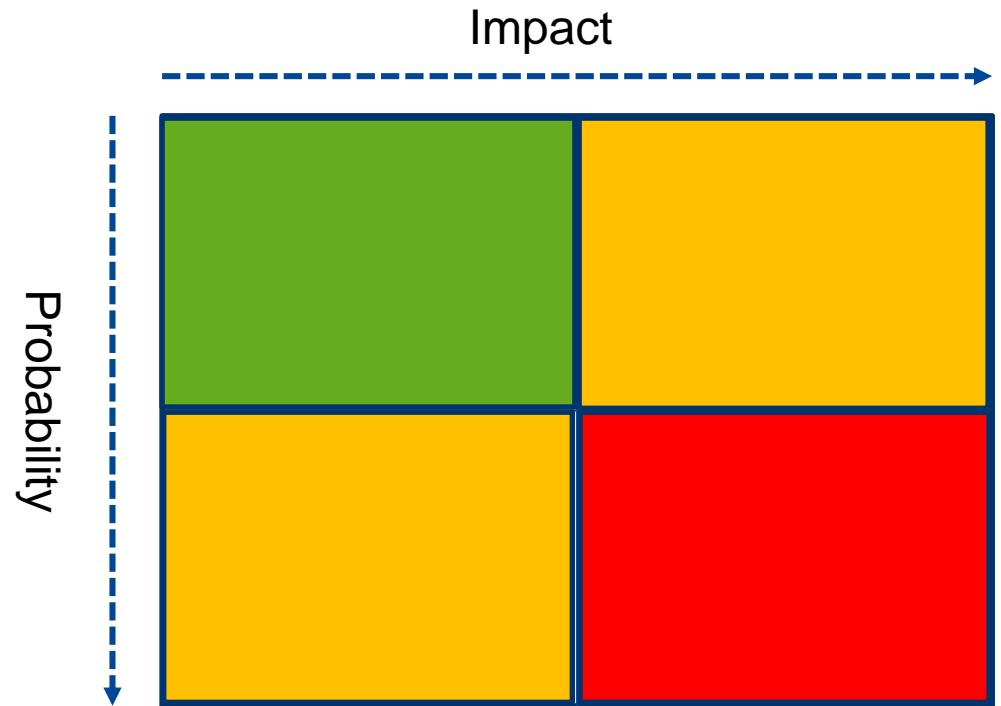
## 2e) Software Risiko Management

### ■ Potentielle Typen von Risiken

- ▶ Funktionale Risiken
  - Falsche oder ungenaue Anforderungen, wenig Einbeziehung/Bereitschaft von Endbenutzern und Fachspezialisten, widersprüchliche Business Ziele
- ▶ Technische Risiken
  - Außmaß an Komplexität, Projektgröße, neue für Team unbekannte Tools / Sprachen / Frameworks, Abhängigkeiten von Zulieferern von außerhalb des Unternehmens
- ▶ Perönliche Risiken
  - Team fehlt Wissen/Erfahrung, Resourcenengpässe, Produktivitätsengpässe
- ▶ Finanzielle Risiken
  - Budget nicht ausreichend, ROI schwer erreichbar
- ▶ Rechtliche Risiken
  - Vorgaben vom Staat, Rechtliche Vorgaben, Vertragseinschränkungen
- ▶ Führungsrisiken
  - Erfahrung und Wissen fehlen, falsche Planung, fehlende Kommunikation, Probleme in der Organisation

## 2e) Software Risiko Management

- Techniken der Risikobehandlung
  - ▶ Risikovermeidung
    - Mit in die Planung einbeziehen
  - ▶ Weitergabe des Risikos an einen Dritten
    - An einen Subkontraktor
  - ▶ Risikominimierung
    - Risiko wird nicht behoben, aber vermindert
  - ▶ Risiko Akzeptanz
    - Risiken können nicht alle vermieden werden, einen Teil davon muss man akzeptieren können
- Ein gutes Risikomanagement führt zu einem Satz von Risiken, die von allen akzeptiert werden können



### 3) Die Domäne

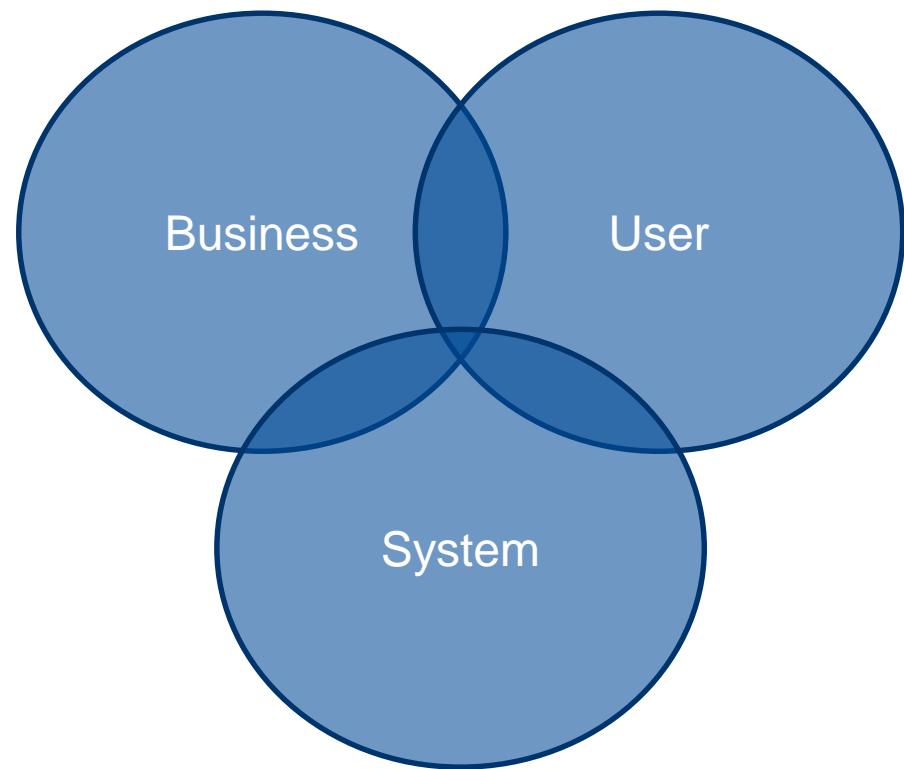
- Es gibt Business Domänen und technische Domänen (Wirkungsraum)
- Ein Architekt muss beide beherrschen
- Das Domänenmodell hilft dabei

Wir werden folgendes betrachten

- a) Einen Sinn fürs Geschäft entwickeln (business acumen)
- b) Domain Driven Design (DDD)
- c) Anforderungs (Requirements) Engineering
- d) Anforderungs Erhebung

### 3a) Einen Sinn fürs Geschäft entwickeln

- Ein Software Architekt muss die Anforderungen/Ziele/Einschränkungen zusammenbringen



Internal

### 3a) Einen Sinn fürs Geschäft entwickeln

- Ein Verständnis fürs Geschäft und der Fachsprache macht den Architekten komplett
- Architekten profitieren mehr als z.B. Softwareentwickler von diesem Verständnis
- Softwarearchitekten stehen dauernd im Austausch mit dem Business und ein gemeinsames Verständnis und eine gemeinsame Sprache ist deshalb mehr als hilfreich
- Den Hut des Benutzers aufsetzen zu können, hilft bei der Umsetzung von Zielen und Anforderungen
- In größeren Unternehmen gibt es oft Architekten für Business-Bereiche wie Finanzen, Personalwesen, Produktion, Logistik u.a.
- Ein Verständnis für Entscheidungen basierend auf ROI (Return on Invest) Konzepten bringt Mehrwert in entsprechende Diskussionen
- Erreicht werden kann dieses Wissen entweder durch Mitarbeit im Business (Jobrotation), Selbststudium, Literatur oder Recherchen

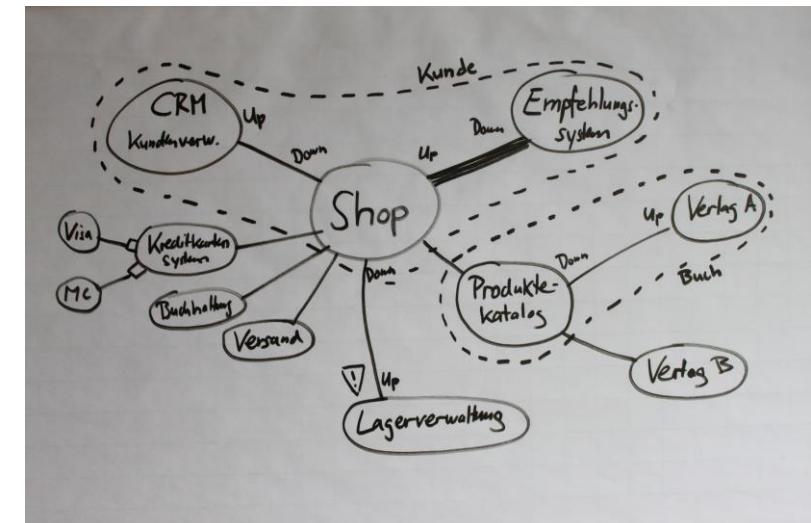
## 3a) Einen Sinn fürs Geschäft entwickeln

### ■ Schwerpunkte

- ▶ Produkte und Dienstleistungen
- ▶ Womit verdient mein Unternehmen Geld ?
- ▶ Verständnis der Business Prozesse
- ▶ Marktumfeld
  - Konkurrenten
    - Unterschiede / Gemeinsamkeiten
    - Stärken / Schwächen
  - Kunden (wichtigster Aspekt überhaupt)
    - Business des Kunden
    - Warum nutzen die Kunden die Produkte / Dienste
  - Märkte
    - Was zeichnet die Märkte aus ?

## 3b) Domain Driven Design

- Der Versuch Anforderungen in logische Bereiche und Unterbereiche aufzuteilen
- Methoden wie Bounded Context, Context Map, Context Mapping Patterns helfen dabei
  - Bounded context = zusammengehöriger Teilbereich im Domänenmodell
- Eine sog. „ubiquitous (universal, common) language“ dient zu Beschreibung von Domänen und Zusammenhängen/Beziehungen
  - Man beginnt am besten mit sog. Business Capabilities
  - Man definiert sog. Entitäten, Wertobjekte und Aggregate
- Man zeigt Informationsflüsse zwischen und in Bounded Contexts
- Architekturprinzipien und –muster wie „Lose Kopplung, Single Responsibility, Event Sourcing oder CQRS kommen zum Ansatz



### 3c) Requirements Engineering

- Notwendig als Grundlage zur Domänenmodellierung und Architektur
- Umfasst alle Aufgaben wie Aufnehmen, Analysieren, Dokumentieren, Validieren und Pflegen der Anforderungen (und Einschränkungen) der Stakeholder an ein Softwaresystem
- Als Software Architekt ist man in alle diese Aufgaben involviert, deshalb macht es Sinn sie zu kennen
  
- Typen von Software Anforderungen (Requirements)
  - ▶ Business requirements
  - ▶ Functional requirements
  - ▶ Non-functional requirements
  - ▶ Constraints (Einschränkungen)

### 3c) Requirements Engineering

- Business requirements
  - ▶ Sind die Ziele des Business (hohes Level), die mit dem Software System erreicht werden sollen
  - ▶ Die Probleme, die gelöst werden sollen oder die Möglichkeiten, die das Software System für das Business erreichen soll
  - ▶ Business requirements können Anforderungen, die vom Markt kommen, beinhalten
  - ▶ Unterscheidungsmerkmale oder Ähnlichkeiten zu Lösungen von Wettbewerbern finden sich hier
  - ▶ Business requirements haben meistens den Fokus auf Qualitätsmerkmalen

### 3c) Requirements Engineering

- Functional requirements
  - ▶ Beschreiben die Funktionalität des Software Systems
  - ▶ Sie beschreiben die Fähigkeiten in Bezug auf das Verhalten des Systems
  - ▶ Funktionalitäten und Fähigkeiten ermöglichen Stakeholdern ihre Aufgaben zu erfüllen
  - ▶ Beinhaltet die Interaktion des System mit der Umgebung (Input, Output, Services, Interfaces)
  - ▶ Können von unterschiedlichen Quellen kommen wie
    - Organisation (Regeln und Grundsätze des Unternehmens)
    - Gesetzgebung (Recht und Rechtsprechung)
    - Ethik (Sicherheit und Schutz)
    - Auslieferung und Verwendung
    - Standards (die befolgt werden müssen)
    - Extern (z.B. Integration mit einem externen System)

## 3c) Requirements Engineering

- Non-functional requirements
  - ▶ Bedingungen, die zur Effektivität der Lösung beitragen
  - ▶ Einschränkungen, die bedacht werden müssen
  - ▶ Non-functional requirements werden oft nicht berücksichtigt, sind aber ein wichtiger Erfolgsfaktor
  - ▶ Können signifikante Auswirkungen auf das Design und die Architektur haben
  - ▶ Software Architekten müssen eine aktive Rolle bei der Erfassung spielen
  - ▶ Qualitätsattribute wie Wartbarkeit, Benutzbarkeit, Testbarkeit und Kompatibilität sind wichtige Themen

## 3c) Requirements Engineering

- Constraints (Einschränkungen)
  - ▶ Technisch oder nicht-technisch
  - ▶ Funktional oder nicht-funktional
  - ▶ Entscheidungen, die schon gefallen sind und befolgt werden sollen
  - ▶ Ein Software Architekt muss sie beachten, er kann sie nicht ändern (jedenfalls nicht so einfach)
  - ▶ Beispiele
    - Abkommen mit Lieferanten
    - Bereits ausgewähltes Tool
    - Die Software muss sich im gesetzlichen Rahmen bewegen
    - Es gibt bereits festgelegte Termine oder Meilensteine
    - Ressourcen sind fest zugeordnet
    - Ein Entwicklerteam kennt hauptsächlich (nur) eine bestimmte Programmiersprache
  - ▶ Constraints sollten wie Anforderungen behandelt werden

## 3c) Requirements Engineering

- Die Wichtigkeit kann nicht oft genug betont werden
- Wichtig für ein erfolgreiches Projekt, für eine gute Architektur, für zufriedene Kunden
- Vorteile
  - ▶ Reduzierte Nacharbeit
  - ▶ Weniger unnötige Eigenschaften
  - ▶ Geringere Erweiterungskosten
  - ▶ Schnellere Entwicklung
  - ▶ Geringere Entwicklungskosten
  - ▶ Bessere Kommunikation
  - ▶ Genauere Schätzungen beim Testen
  - ▶ Höhere Kundenzufriedenheit

### 3c) Requirements Engineering

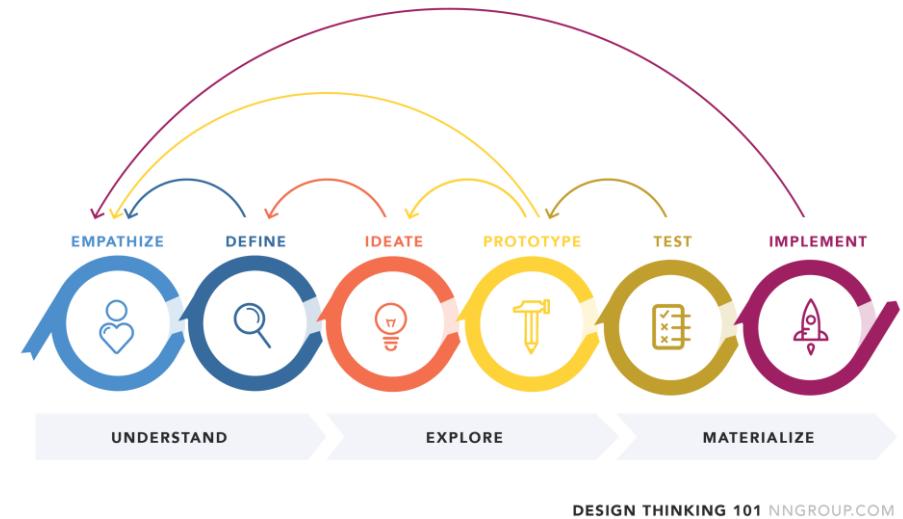
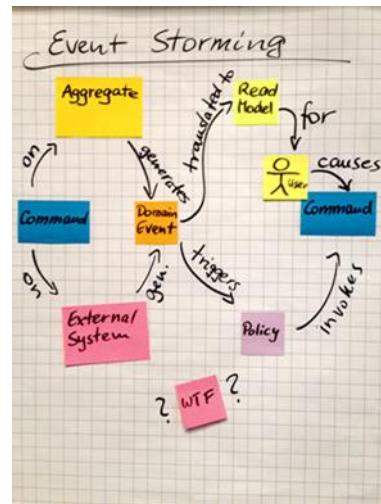
- Anforderungen müssen
  - ▶ meßbar und testfähig sein
    - Webpage ist in 2 sec bedienbar und nicht Webpage ist schnell bedienbar
  - ▶ Eindeutig und konsistent
  - ▶ Klar und nicht widersprechend
  - ▶ Verifizierbar (nachprüfbar)
- Software Architekten weisen auf Unstimmigkeiten und Unzulänglichkeiten hin
- Software Architekten sorgen für eine klare und stimmige Dokumentation ohne Mißverständnisse
- Anforderungen können (müssen nicht) die Architektur massiv beeinflussen
- Anforderungen an die Qualität beeinflussen meistens die Architektur

### 3c) Anforderungs Erhebung

wissen was man weiß,  
wissen was man nicht weiß,  
nicht wissen was man weiß,  
nicht wissen was man nicht weiß

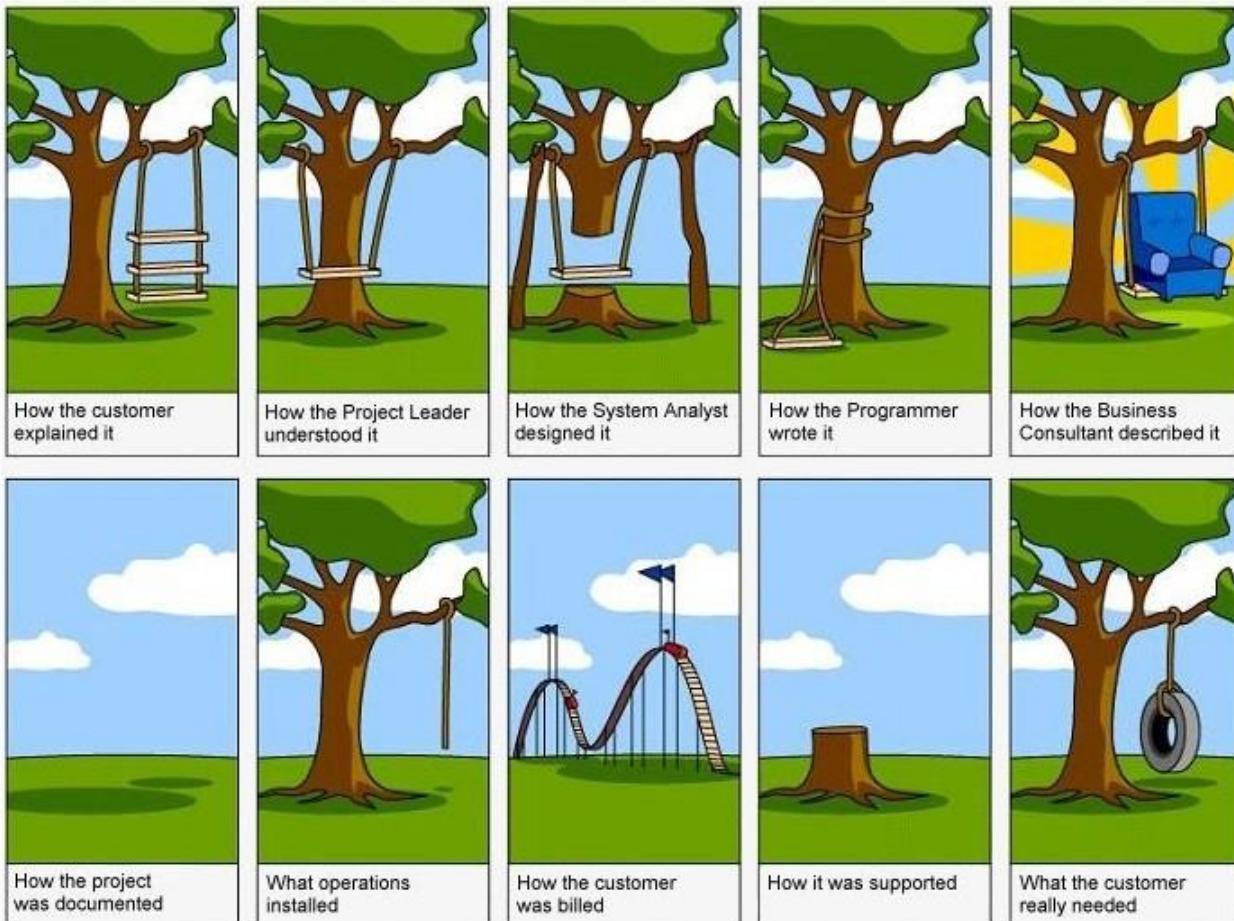
#### ■ Es gibt verschiedenste Erhebungsmethoden

- ▶ Interviews
- ▶ Workshops
  - Event storming
  - Design Thinking
  - Brainstorming
- ▶ Beobachtungen
- ▶ Umfragen
- ▶ Analyse von vorhandener Dokumentation
- ▶ Prototyping
- ▶ Reverse Engineering



### 3c) Anforderungs Erhebung

Wissen was man weiß,  
wissen was man nicht weiß,  
nicht wissen was man weiß,  
nicht wissen was man nicht weiß



## 4) Software Qualitätsattribute

- Sind für Architekten enorm wichtig, da sie Architekturentscheidungen beeinflussen
- In diesem Kapitel betrachten wir Softwareattribute und deren Bedeutung im Softwareentwicklungszyklus
- Manche Attribute lassen sich nicht einfach feststellen

Wir werden folgendes betrachten

- a) Was sind Qualitätsattribute
- b) Maintainability
- c) Usability
- d) Availability ( + Ausflug in moderne Softwarearchitektur)
- e) Portability
- f) Interoperability
- g) Testability

## 4a) Qualitätsattribute

- Sind Eigenschaften eines Softwaresystems und eine Untermenge der non-functional requirements
- Wie alle Anforderungen sollten sie meßbar und testbar sein
- Sie sind ein Benchmark für die Qualität eines Software Systems und messen dessen Fitness
- Ein Software System besteht aus einer Kombination von Qualitätsattributen
- Je besser die Qualitätsattribute erfüllt werden, um so besser ist die Qualität der Software
  - ▶ Sie haben entscheidenden Einfluss auf
    - Das Design
    - Die Wartbarkeit
    - Das Laufzeitverhalten
    - Die User Experience
- Einige Attribute beeinflussen sich gegenseitig oder können sich (wenigstens teilweise) gegenseitig ausschließen (z.B. Performance mit Scalability oder Security und Usability)
- Es ist wichtig, sie zu kennen, zu verstehen und benutzen zu können
- Eine vernünftige Balance kann akzeptierbare Lösungen hervorbringen

## 4a) Qualitätsattribute

- Können intern oder extern sein
- Interne Attribute gehören zum Softwaresystem, sind meßbar und für Entwickler sichtbar
- Beispiele
  - ▶ LOC (lines of Code)
  - ▶ Maß der Kohäsion
  - ▶ Lesbarkeit des Codes
  - ▶ Grad der Kopplung zwischen Modulen
- Sie zeigen die Komplexität eines Software Systems
- Sie beeinflussen externe Attribute, obwohl sie nicht direkt von Endanwendern gesehen werden
- Ein höherer Level von interner Qualität hat in den meisten Fällen eine höhere äußere Qualität zur Folge

## 4a) Qualitätsattribute

- Externe Qualitätsattribute sind Eigenschaften die extern sichtbar sind
- Sie sind für Endbenutzer sichtbar/bemerkbar
- Beispiele
  - ▶ Performance
  - ▶ Reliability (Ausfallsicherheit)
  - ▶ Availability (Verfügbarkeit)
  - ▶ Usability (Benutzbarkeit)
- Qualitätsattribute spielen eine wichtige Rolle im gesamten SDLC (Software Development Life Cycle)
  - ▶ Vom Design bis zum Testen

## 4a) Qualitätsattribute

- Müssen immer wieder überprüft werden
  - ▶ Manuell (z.B. Usability Test)
  - ▶ Durch Tools nach definierten Benchmarks
  - ▶ Durch Code Reviews
  - ▶ Durch automatisierte Unit Tests
- Jede Testmethode hat seine Stärken und Schwächen
- Oft reicht ein Test alleine nicht aus, um ein Attribut zu bestimmen
- Manchmal sind umfangreiche Tests notwendig
- Eine sinnvolle Balance zwischen Testaufwand und benötigter Zeit ist wichtig
- Automatisierung von Test, wo immer möglich, ist notwendig
- Automatisierte Test sind oft Teil eines Continuous Delivery Prozesses

## 4b) Maintainability

- Maß für die Wartbarkeit eines Software Systems
- Wartung muss auch nach der Inbetriebnahme einfach möglich sein
- Sichert den Wert des Software Systems über die Zeit
- Änderungen an Software Systemen sind vorprogrammiert – am besten schon vorgeplant
- Früher lag der Hauptkostenblock eines Systems in der Entwicklung
- Heute hat sich das in Richtung Wartung verschoben
  - ▶ Früher am Markt, dafür eingeschränkte Funktionalität
- Beeinflusst daher stark die Gesamtkosten
- Ein guter Entwickler entwickelt gut wartbaren Code (auch für andere Entwickler)
- Gute Wartbarkeit unterstützt die Migration von Systemen auf neue Technologien
- Gute Wartbarkeit unterstützt Reverse-Engineering

## 4b) Maintainability

- Typen von Wartbarkeit
  - ▶ Corrective
    - Korrektur von Fehlern (Bugs), gefunden im Betrieb oder von Benutzern
    - Führt zu Incidents (Prio = Reihenfolge der Behebung, Severity = Wirkweite)
    - Wartbarkeit kann an der Zeit für Suche, Finden und Behebung gemessen werden
  - ▶ Perfective
    - Bei neuer oder erweiterter Funktionalität
    - Eine gute Architektur kann Perfective Maintainability unterstützen

## 4b) Maintainability

- Typen von Wartbarkeit
  - ▶ Adaptive
    - Um Softwaresysteme an Änderungen der Umgebung anzupassen
      - Neues Betriebssystem oder neues Datenbanksystem
  - ▶ Preventive
    - Um Fehler, Ausfälle oder Probleme in der Zukunft zu vermeiden
  - ▶ Predictive
    - Der Versuch, mit Big Data Methoden Fehler, Probleme oder Ausfälle vorherzusagen

## 4b) Maintainability

- Modifiability (Veränderbarkeit)
  - ▶ Die Möglichkeit Software Systeme einfach zu ändern ohne die Wartbarkeit zu verschlechtern
  - ▶ Wichtig für moderne Software in agilen Umgebungen
  - ▶ Iterationen in der Entwicklung ändern Systeme Schritt für Schritt
- Erweiterbarkeit und Flexibilität
  - ▶ Wird auch Adaptivität und Resilienz genannt
  - ▶ Werden wir in Teil 17 behandeln

## 4b) Maintainability

- Spezielles Design zur Unterstützung von Wartbarkeit ist notwendig
  - ▶ Reduzierung der Komplexität
  - ▶ Klingt einfach – kann aber mehr Zeit und/oder Geld kosten
- Techniken zur Reduzierung der Komplexität
  - ▶ Reduzierung der Größe
  - ▶ Erhöhung der Kohesion (Zusammenhalt)
  - ▶ Reduzierung von Koppelung

Siehe Teil 6 Prinzipien und Praktiken

## 4b) Maintainability

### ■ Das Messen der Wartbarkeit

- ▶ Lines of Code (LOC)
  - Vergleich zwischen 2 Systemen nur möglich bei unterschiedlichen Größenordnungen
  - Kann von Entwicklungsumgebungen gezählt werden
- ▶ Cyclomatische Komplexität
  - Ist eine quantitative Metrik
  - Entwickelt von Thomas J. McCabe
  - Maßzahl von linear unabhängigen Pfaden durch ein Modul oder Element
  - Formel :  $CC = E - N + 2P$
  - Darstellung in Graphen

E = Anzahl der Kanten im Graphen

N = Anzahl der Knoten im Graphen

P = Anzahl von Pfaden im Graphen

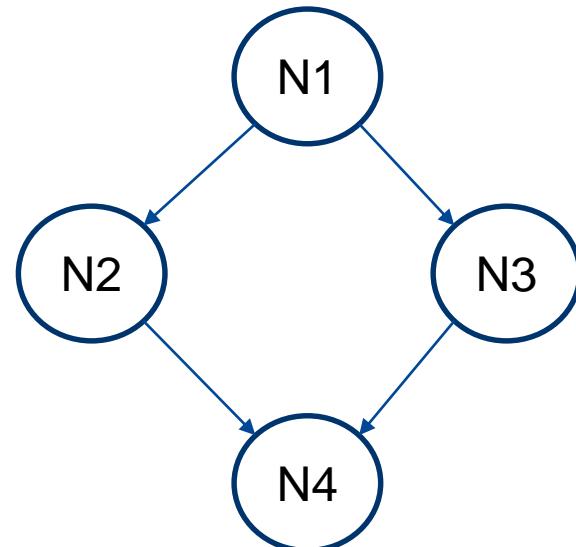
## 4b) Maintainability

- Cyclomatische Komplexität
  - ▶ Beispiel

Pseudocode

```
If (N1)  
  then N2  
  else N3  
End if  
N4
```

Graph



$$\begin{aligned} E &= 4 \\ N &= 4 \\ P &= 1 \end{aligned}$$

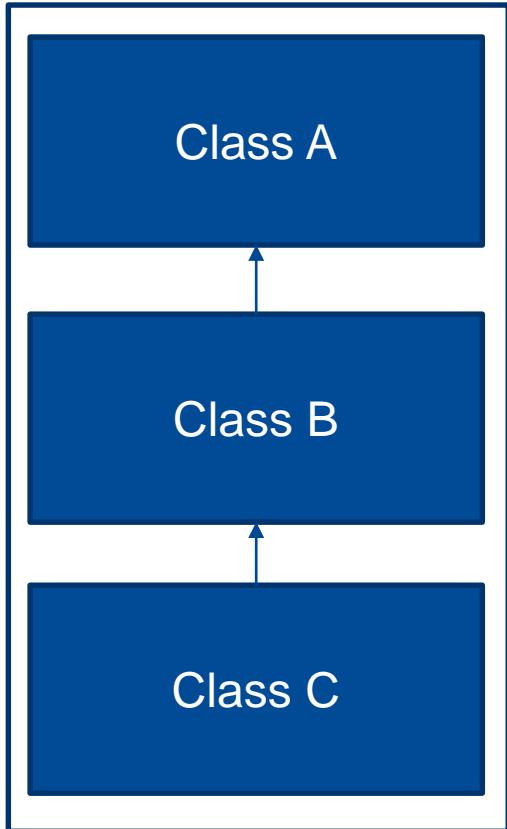
$$CC = 4 - 4 + 2 * 1 = 2$$

Geringe Komplexität

Ab CC > 10 spricht man von erhöhter Komplexität

## 4b) Maintainability

- Das Messen der Wartbarkeit
  - ▶ Depth of Inheritance tree (DIT)



DIT = 0

Je größer DIT je höher die Komplexität

DIT = 1

DIT > 5 deutet auf Optimierungspotential hin

DIT = 2

## 4c) Usability

- Wichtiger Punkt für gute Architektur
- Beschreibt wie einfach es für Enduser ist ihre Aufgaben mit dem System zu erledigen (useful)
- Ist für Enduser hauptsächlich im Fokus
- Hat Einfluss auf den Gesamteindruck des Systems
- Kann oft am einfachsten verbessert werden
- Ist enorm wichtig, da das die Produktivität der Enduser maßgeblich beeinflusst
  - ▶ Gute Usability erhöht die Produktivität
- Kann im schlechten Fall zur totalen Ablehnung oder zur Suche nach Alternativen führen
- Usability untergliedert sich in
  - ▶ Learnability
    - Die Geschwindigkeit mit der neue User das System zu bedienen lernen (ISO/IEC 25010)
  - ▶ Accessibility
    - Wie kann auf das System zugegriffen werden (Technologien, Navigation, Farben, ...)
  - ▶ Feedback
    - Nachrichten, Tooltips, lang laufende Tasks

## 4d) Availability

- Beschreibt ein prozentuales Maß über eine definierte Zeit, an der ein System verfügbar ist
- Ist die Wahrscheinlichkeit der Verfügbarkeit eines Systems zu einem bestimmten Zeitpunkt
- Man sieht oft Maßzahlen wie 99,9%, 99,99% oder 99,999%
- Ist definiert als

$$\text{Verfügbarkeit} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

MTBF = meantime between failures (manchmal auch MTTF = meantime to failure)

Die mittlere Zeit zwischen 2 Ausfällen

MTTR = meantime to recovery/repair

Die mittlere Zeit die zum Wiederanlaufen/zur Reparatur benötigt wird

Wichtig : es muss ein Bezugszeitraum angegeben sein  
99,999% in Bezug auf 1 Jahr bedeutet 5 min und 15 sec

## 4d) Availability

- Wichtig : es muss ein Bezugszeitraum angegeben sein
  - ▶ Typische Verfügbarkeiten

Verfügbarkeiten	Downtime / Jahr	Downtime / Monat	Downtime / Woche
99,0 %	3,65 Tage	7,2 Stunden	1,68 Stunden
99,9 %	8,76 Stunden	43,2 Minuten	10,1 Minuten
99,99 %	52,6 Minuten	4,32 Minuten	60,5 Sekunden
99,999 %	5,26 Minuten	25,9 Sekunden	6,05 Sekunden

- ▶ Verfügbarkeiten abhängiger Systeme werden multipliziert

## 4d) Availability

- Fehlererkennung
  - ▶ Ping
    - Von außen anpingen (TCP/IP Ping)
  - ▶ Heartbeat
    - Periodisches Senden von Nachrichten nach außen
  - ▶ Timestamp
    - Zur Erkennung von Fehlern in Reihenfolgen
  - ▶ Voting
    - Triple modular redundancy (TMR)
    - Drei Module machen die gleiche Berechnung
    - Die Mehrheit gleicher Ergebnisse bestimmt das Gesamtergebnis
  - ▶ Sanity checking
    - Oft eine speziell implementierte Funktion (von außen aufrufbar)
  - ▶ Condition monitoring und Selbsttests

## 4d) Availability

- Fehlerbehebung
  - ▶ Exception handling
    - geplant gesteuerte Ausnahmebehandlung
  - ▶ Retry strategien
    - Wiederholungen (in Intervallen, incrementell, sofort, zufällig)
  - ▶ Redundanz
    - Failover Mechanismen
    - Active/passive/cold Spare
  - ▶ Rollback
    - Zurückrollen zu einem definierten Punkt
  - ▶ Graceful degradation
    - Gezieltes Abschalten von Funktionalität
  - ▶ Ignorieren

## 4d) Availability

- Fehlervermeidung
  - ▶ Removal from service (Stilllegen)
  - ▶ Transaktionen
    - Siehe Datenbanken
  - ▶ Competence Sets
    - Erhöhung der Fähigkeit eines Systems mit Fehlern umzugehen
    - Ausnahmen werden Bestandteil der Logik
  - ▶ Exception prevention
    - Prüfen von Grenzen (Arrays)
    - Überprüfen von Argumenten auf zulässige Werte

## 4d) Ausflug in moderne Softwarearchitektur Resilienz

- In heutigen komplexen, verteilten und hochgradig vernetzten Systemlandschaften ist es unmöglich, die vielfältigen möglichen Fehlerquellen zu antizipieren und durch vorbeugende Maßnahmen auszuschließen.
- Der immer höher werdende Grad an Verteilung, sei es wegen Cloud, Mobile oder Internet of Things sowie die immer schwerer vorhersagbaren Lastmuster werden dafür sorgen, dass Resilient Software Design in Zukunft noch viel wichtiger wird als es jetzt bereits ist.

# 4d) Ausflug in moderne Softwarearchitektur

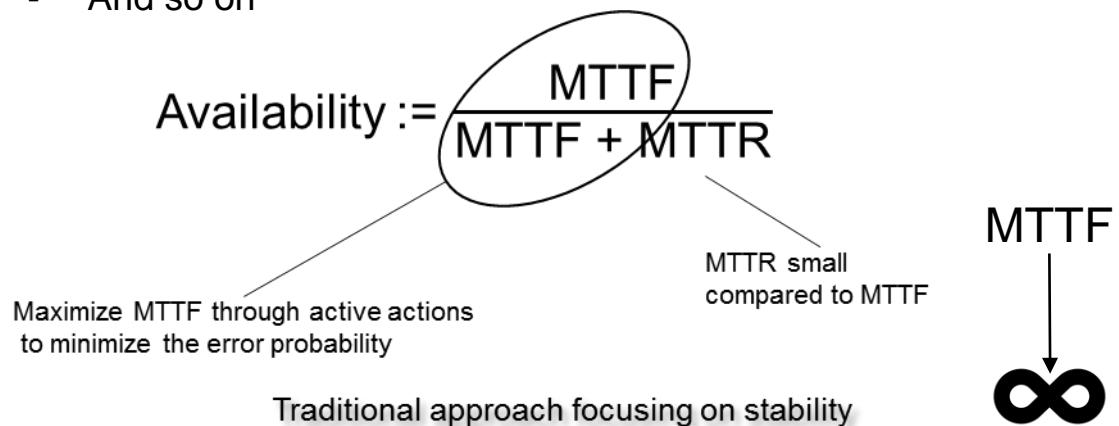
## Resilienz

### Traditional approach

The traditional approach is to postpone the incidence of an error as long as possible, that means to make MTTF extremely large to make MTTR nonsignificant for availability.

To reach that goal, high investments in infrastructure is needed :

- redundant Hardware (Cluster) and Mirroring
- redundant networks
- Shadow databases
- Second (remote) data center
- Businesscopies within storage subsystems
- And so on

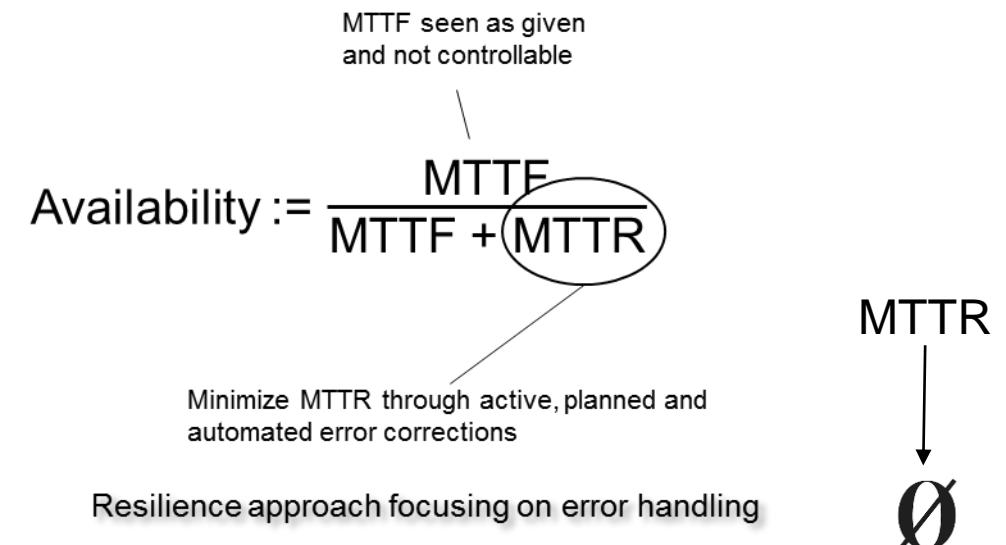


Goal : Availability = 1 (= 100 %)

### Resilient approach

Resilient Software Design tries to accept errors, because they cannot neither be avoided nor foreseen, they just happen.

Resiliency is the capability of a system to handle unexpected situations, without disturbing the end user (best case) or (worst case) with a defined and planned „graceful degradation of service“.



# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

**Welches Design, welche Architektur ist für resiliente Software notwendig ?**

Gibt es Resilienzprinzipien und Muster ?

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Isolation

Das erste Grundprinzip von Resilienz ist Isolation.

Auf diesem Prinzip bauen fast alle anderen Resilienz-Muster auf.

Isolation bedeutet, dass ein System niemals als Ganzes kaputtgehen darf.

Um das zu vermeiden, teilt man das System in möglichst unabhängige Einheiten auf und isoliert diese gegeneinander.

Diese unabhängigen Einheiten werden Bulkheads, Failure Units oder Units of Mitigation genannt.

Die Einheiten isolieren sich gegen Fehler anderer Einheiten, um kaskadierende, d. h. sich über mehrere Einheiten fortpflanzende Fehler zu vermeiden.



Elementare Grundprinzipien im Resilient Software Design

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Redundanz

#### Redundanz ist ein weiteres zentrales Resilienz-Prinzip.

Redundante Failure Units werden zur Korrektur oder Abschwächung von Fehlern genutzt.

Redundanz ist geeignet mit allen Arten von Fehlern umzugehen  
(nicht nur mit Absturzfehlern).

Was möchte man mit Redundanz adressieren ?

- Failover (komplett transparentes Umschalten auf eine andere Einheit)
- Geringe Latenz (Reduktion der Wahrscheinlichkeit zu langsamer Antworten)
- Erkennen von Antwortfehlern (durch Auswertung von Antworten mehrerer redundanter und unabhängiger Einheiten)
- Lastverteilung (zu zahlreiche Anfragen auf mehrere Einheiten verteilen)
- ... usw.

Abhängig vom gewählten Szenario sind weitere Aspekte zu berücksichtigen :

- Routingstrategien (Load Balancer, Round Robin)
- Master-Slave-Ansätze für Failoverszenarien
- Fan out & quickest one wins-Ansatz für geringe Latenz
- Automatisierung (Menschen unter Stress machen Fehler) mit Eingreifmöglichkeiten für Administratoren
- Verfolgung von Systemzuständen. Welche Einheit ist Master, welche ist Slave?  
Wie viele Einheiten laufen? Wo laufen sie? usw.

Redundanz ist ein mächtiges Prinzip, das eine Menge Architekturarbeit benötigt.



Elementare Grundprinzipien im Resilient Software Design

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Lose Kopplung

Ist ein Grundprinzip zur Vermeidung kaskadierender Fehler und unterstützt die Isolation von Failure Units.

Ein Muster zur Umsetzung von loser Kopplung ist die Verwendung asynchroner Event- oder Nachrichten-basierter Kommunikation. Dadurch wird eine maximale Entkopplung der einzelnen Einheiten erreicht und die Wahrscheinlichkeit sich fortpflanzender Fehler wird minimiert.

Asynchrone Kommunikation benötigt zusätzliche Visualisierungs- und Überwachungsmöglichkeiten für asynchrone Nachrichtennetzwerke, um den Überblick zur Laufzeit nicht zu verlieren.

Synchrone Kommunikation muß Timeout- und Latenzüberwachung betreiben, um Antwortzeitfehler erkennen und behandeln zu können. Antworten, die nicht rechtzeitig ankommen oder synchrone Handshakes machen synchrone Kommunikation ähnlich kompliziert wie asynchrone Kommunikation.

Für synchrone Kommunikation (manchmal gibt es auch dafür gute Gründe) sollte man auf jeden Fall Muster wie Timeout und Circuit Breaker einsetzen, um eine gute Latenzüberwachung zu implementieren (s.a. Bibliotheken wie Hystrix von Netflix).

Ein hilfreiches Muster bei asynchroner Kommunikation ist die sog. Idempotenz.



# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Lose Kopplung → Idempotenz

- Mathematische Definition

Ein Element  $a$  einer Menge  $X$  heißt idempotent bezüglich einer  $n$ -stelligen Verknüpfung  $v : X^n \rightarrow X$ ,  $n \in \mathbb{N}$  und  $n > 1$  falls gilt:

$$v(a, \dots, a) = a$$

Falls  $n = 2$  ist und die Verknüpfung (wie etwa bei der Multiplikation üblich) in Potenzschreibweise notiert wird, schreibt sich die Bedingung als  $a^*a = a^2 = a$ , woraus unmittelbar  $a^n = a$  für alle  $n \in \mathbb{N}$  folgt (da es für  $n = 1$  und  $n = 2$  gilt), was die Bezeichnung Idempotenz (lat. für gleiche Potenz) erklärt.

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Lose Kopplung → Idempotenz

- Ein (verändernder) Aufruf ist idempotent, wenn wiederholte Aufrufe keine zusätzlichen Seiteneffekte haben.
  - Beispiel: Der Aufruf „Addiere 1 auf Wert“ ist nicht idempotent, weil der Wert sich mit jedem Aufruf verändert. Der Aufruf „Setze Wert auf 5“ hingegen ist idempotent. Egal, wie oft ich diesen Aufruf auf einen Wert anwende, das Ergebnis wird immer gleich sein, d. h. es entstehen keine zusätzlichen Seiteneffekte.
- Bei synchronen Aufrufen mit Latenzüberwachung gibt es das Problem, dass im Fall einer zu langsam Antwort in der Regel nicht unterschieden werden kann, ob der Aufruf gar nicht beim Empfänger angekommen ist oder ob der Empfänger nur zu langsam geantwortet hat.
  - ▶ Es ist also unklar, ob die Anfrage verarbeitet wurde oder nicht.
- Verwendet man nicht idempotente Aufrufe, steht man bei zu langsamen Antworten vor dem Problem, dass man nicht oder nur mit sehr großem Aufwand entscheiden kann, ob man den Aufruf noch einmal senden darf oder nicht.
- Verwendet man idempotente Aufrufe, stellt sich dieses Problem nicht
  - ▶ Man ruft den Empfänger einfach so oft auf (mit sinnvollen Pausen und Fehlerbehebungsmaßnahmen zwischen den Aufrufen), bis man die Rückmeldung erhält, dass der Aufruf erfolgreich verarbeitet worden ist.
  - ▶ Idempotente Aufrufe ermöglichen es, von einer Exactly Once-Kommunikation auf eine At Least Once-Kommunikation zu wechseln.
  - ▶ Diese Art der Kommunikation ist wesentlich leichter umsetzbar und hat deutlich geringere Anforderungen an die Kommunikationsinfrastruktur.

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Lose Kopplung → Idempotenz

- Wie kann man Aufrufe/Nachrichten idempotent bekommen ?
  - ▶ Beispiel :  
Erhöhe ab 25.01.2019 Anzahl Kinder bei Mitarbeiterin Maria um 1  
Mitarbeiterin Maria hat am 25.01.2019 ein Kind bekommen
  - ▶ Was ist der Unterschied ?  
  
Erhöhe Anzahl Kinder bei Mitarbeiterin Maria um 1 transportiert die Daten (25.01.2019, 1)  
und die notwendige Verarbeitungslogik (erhöhe = +) in der Nachricht  
  
Mitarbeiterin Maria hat am 25.01.2019 ein Kind bekommen transportiert die Information – die Verarbeitungslogik liegt beim Empfänger  
Diese Information kann ich beliebig oft schicken (bis der Empfänger mir mitteilt, daß er verstanden hat und ich damit aufhören kann)
  - ▶ **Der Austausch von Informationen statt Daten ist eine Möglichkeit idempotente Aufrufe/Nachrichten zu erzeugen.**

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Fallback

Es ist notwendig eine klare Strategie zu besitzen mit Fehlern umzugehen (beheben oder abschwächen).

Diese Strategie sollte sowohl die Bedürfnisse der Nutzer als auch der Administratoren so gut wie möglich unterstützen.

Die Benutzer sollten im Idealfall überhaupt nicht bemerken, dass ein Fehler aufgetreten ist. Die Administratoren sollten einfach Fehlerbehebung durchführen können.

Das Mittel der Wahl ist eine vollständig automatisierte Fehlerbehandlung.

Beispiel : einfache Webanwendung

In einer Webanwendung führt eine Anfrage an die Datenbank nach einer Sekunde in einen (erkannten) Timeout. Alle weiteren Anfragen ebenfalls.

Wie könnte die Anwendung reagieren :

- Fehlerseite mit Hinweis auf das Problem (bitte später noch einmal versuchen)
- Nutzerdaten in Caches und Leseanfragen aus den Caches bedienen - Schreibanfragen mit Hinweis
- Schreibanfragen in eine Queue, die asynchron abgearbeitet wird, sobald die Verbindung zur Datenbank wieder verfügbar ist, mit Hinweis, dass der Auftrag angenommen wurde und später verarbeitet wird
- Oder ... ?

Das sind alles Varianten einer **Graceful Degradation Of Service**.

96



Elementare Grundprinzipien im Resilient Software Design

# 4d) Ausflug in moderne Softwarearchitektur

## Resilienz

### Graceful Degradation Of Service

Ist die kontrollierte Herabsetzung der Servicequalität.

Jede der vorher genannten Reaktionen kann je nach Anwendungsfall valide sein.

Eine Variante ist nie valide → der Anwender sieht die Sanduhr, bis sein Browser nach 5 Minuten einen Netzwerktimeout meldet.

Was genau gewählt wird, muss geplant sein und ist keine Entscheidung, die ein Entwickler „on the fly“ treffen kann, während er gerade die entsprechende Fehlerbehandlung implementiert.

Fachanforderungen bestimmen die gewählte Variante, deshalb müssen diese Fragen wie die User Stories und die Basisarchitektur geklärt sein, bevor ein Entwickler den zugehörigen Code schreibt.

In einem agilen Umfeld klärt und beantwortet das ein Product Owner. Gute Architekten entwickeln mit allen Produkt Ownern gemeinsam einheitliche Strategien über alle Produkte.

Ist die grundsätzliche Fallback-Strategie festgelegt, kann man die Strategie bei der Umsetzung noch durch zusätzliche Muster wie Escalation Strategy oder Error Handler, für das Implementieren einer automatischen Fehlerbehebung, unterstützen (s.a. Patterns for Fault Tolerant Software Oktober 2007 von Robert Hanmer).

Graceful Degradation of Service ist also immer **eine geplante Strategie** - aus Fachanforderungen entwickelt.



## 4e) Portability

- Ist ein Maß der Effizienz und Effektivität ein Software System von einer Umgebung in eine andere zu migrieren. Die Faktoren, die Portability beeinflussen sind
  - ▶ Adaptibility (Adaptivität, Anpassbarkeit)
    - Code, der anpassbar ist
    - SOLID Prinzipien unterstützen Adaptivität
      - Single responsibility principle
      - Open/closed principle
      - Liskov substitution principle
      - Interface segregation
      - Dependency inversion

## 4e) Portability

- Ist ein Maß der Effizienz und Effektivität ein Software System von einer Umgebung in eine andere zu migrieren. Die Faktoren, die Portability beeinflussen sind
  - ▶ Installability
    - Grad der einfachen Installierbarkeit/Deinstallierbarkeit eines Systems
    - Der Installationsprozess sollte einfach bedienbar und verständlich sein
    - Der Installationsprozess sollte Updates verarbeiten können (Apple als Vorbild)
  - ▶ Replaceability
    - Maß der Fähigkeit, daß ein System ein anderes ersetzen kann
  - ▶ Internationalisierung und Lokalisierung
    - Maß der Anpassung an verschiedene Sprachen und kulturelle Unterschiede
- Wenn ein Software System auf Portability getrimmt ist, sollte dieser Zustand auch nach nachfolgenden Änderungen bewahrt werden

## 4f) Interoperability

- Das Maß der Fähigkeit Informationen mit anderen Software Systemen austauschen zu können
- Um Informationen austauschen zu können, müssen Systeme kommunizieren können
  - ▶ Syntaktische Interoperabilität
- Um Informationen austauschen zu können, müssen Systeme die Informationen verstehen können
  - ▶ Semantische Interoperabilität
- Es gibt Interoperability Standards wie JSON oder HTTP(S)

## 4g) Testability

- Das Maß der Testfähigkeit eines Systems
- Ein nicht unerheblicher Teil der Gesamtkosten eines Software Systems werden durch das Testen verursacht
- Gute Software Architektur sorgt für gute Testfähigkeit
- Ein hohes Maß an Testfähigkeit hilft auch bei der Fehlersuche
- Schnelles Auffinden von Fehlern erhöht die Gesamtqualität eines Systems

## 5) Softwarearchitektur Design

- Ist einer der wichtigsten Schritte erfolgreiche Software Systeme zu entwickeln
- Es gibt 2 wichtige Ansätze für ein Software Design
  - Top-down
  - Bottom-up
- Software Design ist ein komplexes Thema, dafür wurden Design Prinzipien und Lösungen zur Unterstützung entwickelt
- Design Prozesse leiten Architekten bei Erstellen von Architekturen, die den Anforderungen, Qualitätsattributen und Einschränkungen genügen

# 5) Softwarearchitektur Design

Wir werden folgendes betrachten

- a) Software Architektur Design
- b) Typen von Software Architektur Design
- c) Attribute Driven Design (ADD)
- d) Microsofts Technik für Architektur und Design
- e) Architecture Centric Design Method (ACDM)
- f) Architecture Development Method (ADM)
- g) Den Fortschritt einer Software Architektur verfolgen

## 5a) Software Architektur Design

- Ist die notwendige Grundlage für Entscheidungen zu Software Systemen
- Basiert auf functional und non-functional requirements, Qualitätsattributen und Einschränkungen
- Definiert die Strukturen, die Elemente und die Beziehungen zwischen Elementen eines Software Systems und dokumentiert sie
  - ▶ Liefert die Beschreibung der Funktionalitäten und Interaktionen zwischen Elementen
- Ist signifikant für die Qualität und den langfristigen Erfolg eines Software Systems
- Ist die technische Hilfestellung für die Entwicklung
  - ▶ Kann sich während der Entwicklung ändern,  
wenn sich Anforderungen oder Qualitätsattribute ändern
- Ist ein kreativer Prozess
  - ▶ Teilweise hoher Spaßfaktor
- Ist ein kollaborativer Prozess
  - ▶ Je mehr Wissensträger dazu beitragen, um so besser
- „Perfect is the enemy of good“

## 5b) Software Architektur Design

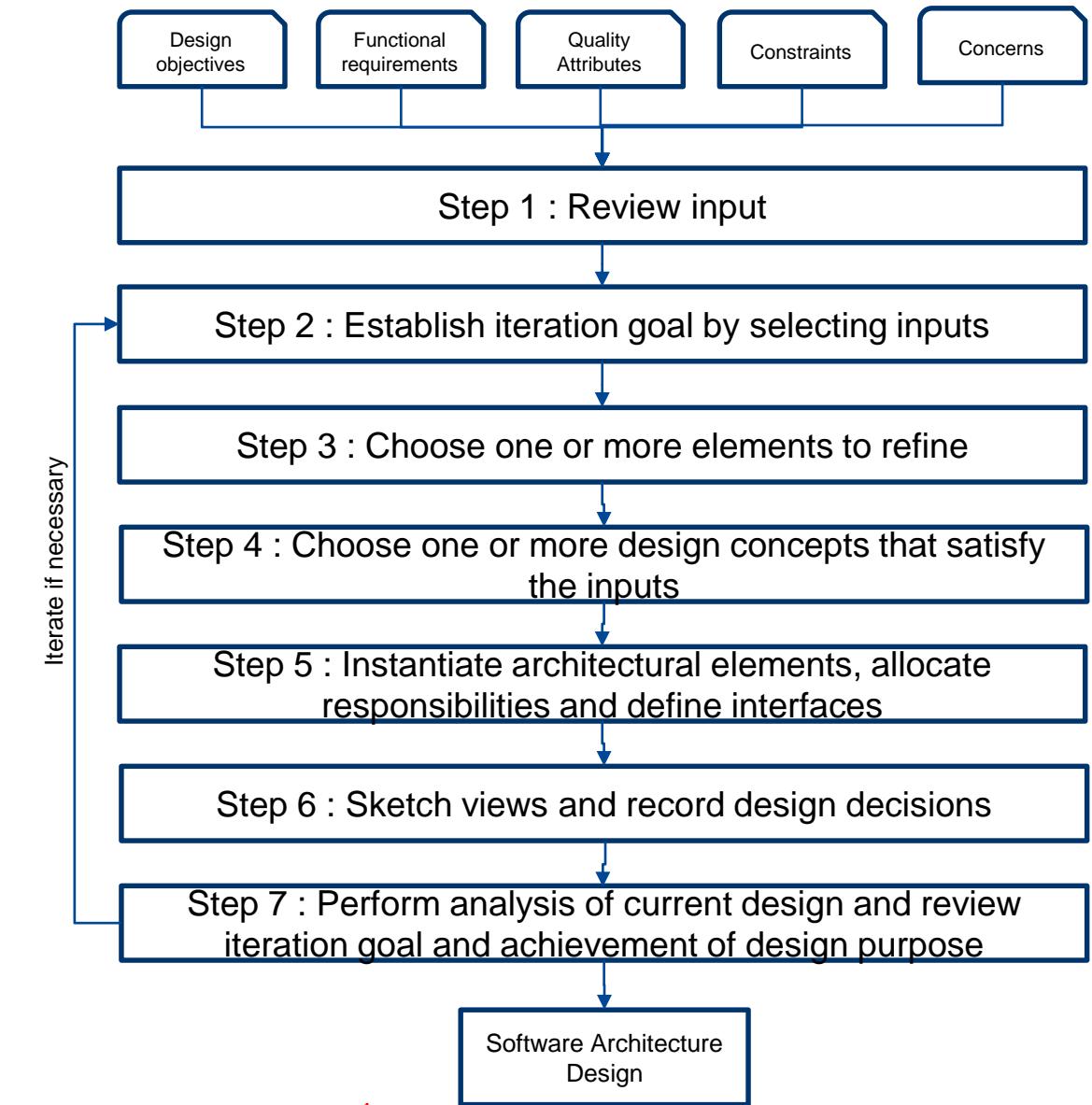
### ■ Typen

- ▶ Top-down
  - Bei großen und komplexen Projekten, bei großen Teams, bei Enterprise Software, die Business Domäne ist bekannt
- ▶ Bottom-up
  - Punkte von top-down negiert
- ▶ Greenfield
  - Grüne Wiese Ansatz
- ▶ Brownfield
  - Aufbauend auf einem bereits bestehenden System

## 5c) Attribute Driven Design

### Schritt 1 Review Input

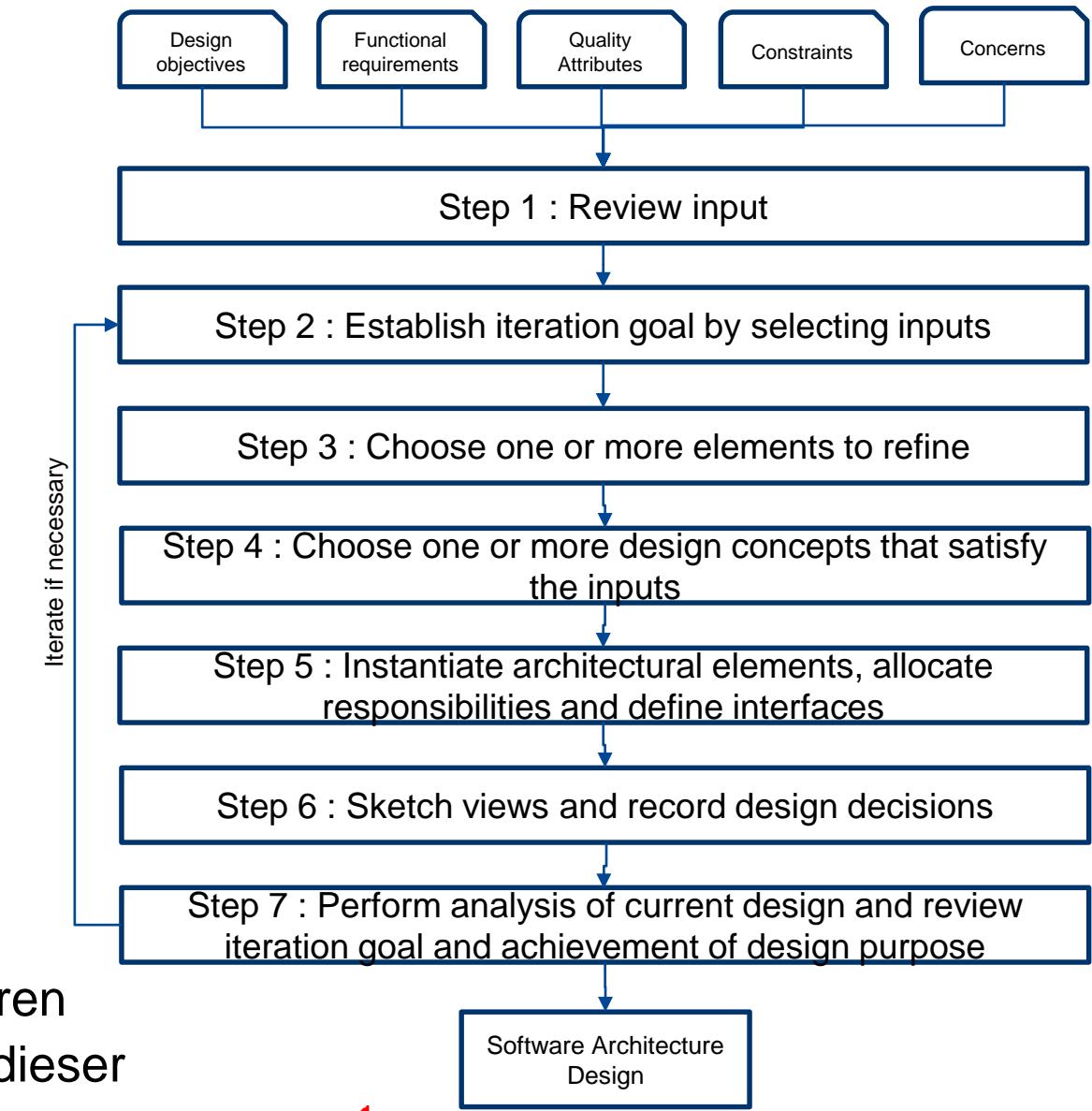
- ▶ Im Schritt 1 wird überprüft, ob alle notwendigen Unterlagen/Eingaben vorhanden und vollständig sind
- ▶ Der Input sind die folgenden Treiber
  - Design Ziele/Vorgaben
  - Functional Requirements
  - Qualitätsattribute
  - Einschränkungen
  - Belange
- ▶ Oft liegt auch schon eine Architektur vor, oder zumindest Teile davon



## 5c) Attribute Driven Design

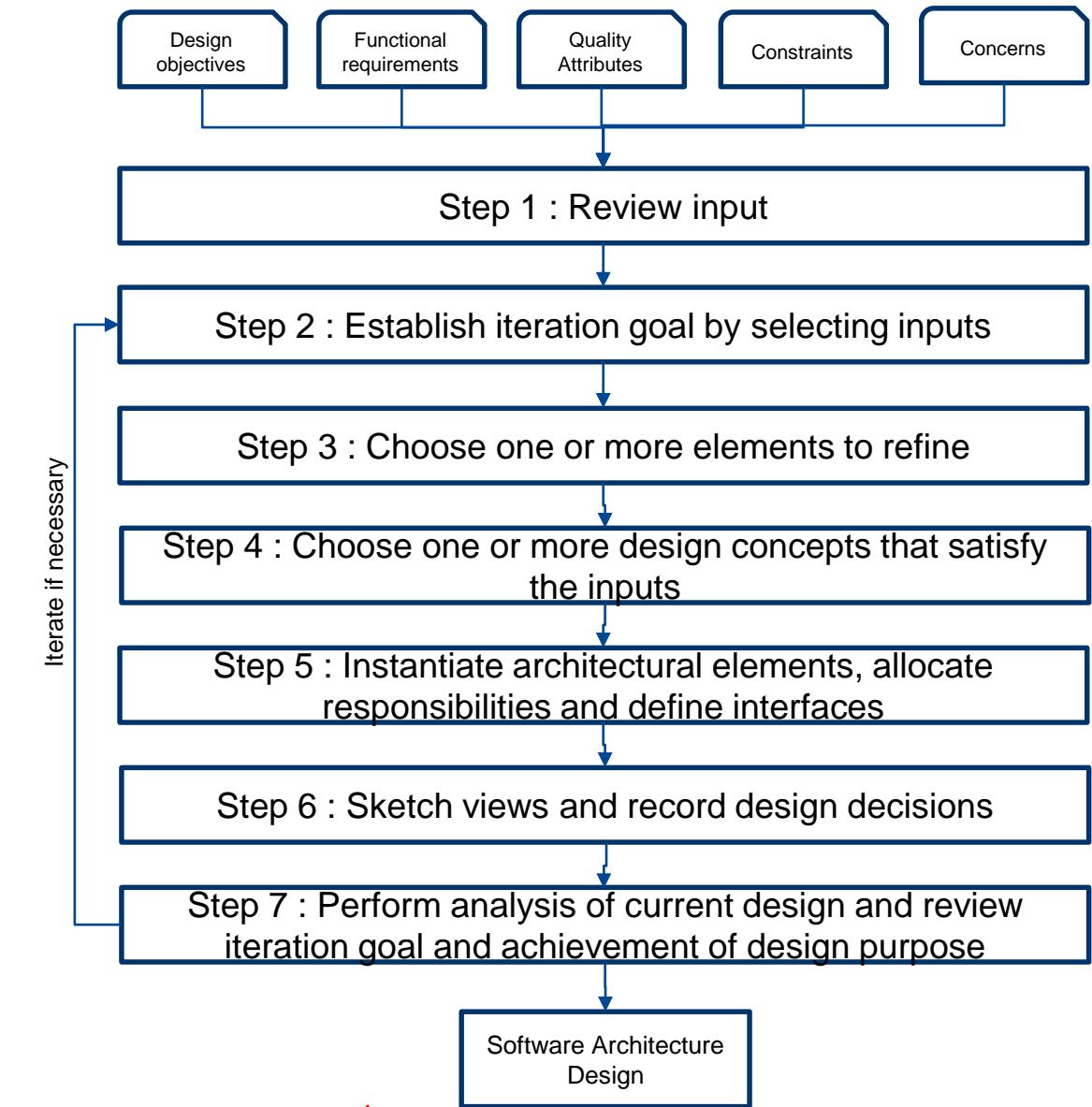
### ■ Schritt 2 Establish iteration goals

- ▶ Nach Schritt 1 werden 1 – N Iterationen durchgeführt werden, die immer mit Schritt 2 beginnen
- ▶ Mit einer agilen Entwicklungsmethode werden immer mehrere Iterationen durchgeführt (Sprints)
- ▶ Das Designziel jeder Iteration wird zu Beginn der Iteration festgelegt
- ▶ Wichtige Frage : welches Designziel wollen wir in dieser Iteration lösen
- ▶ Jedes Designziel muss mit einem oder mehreren Eingaben in Beziehung stehen und stehen in dieser Iteration im Fokus



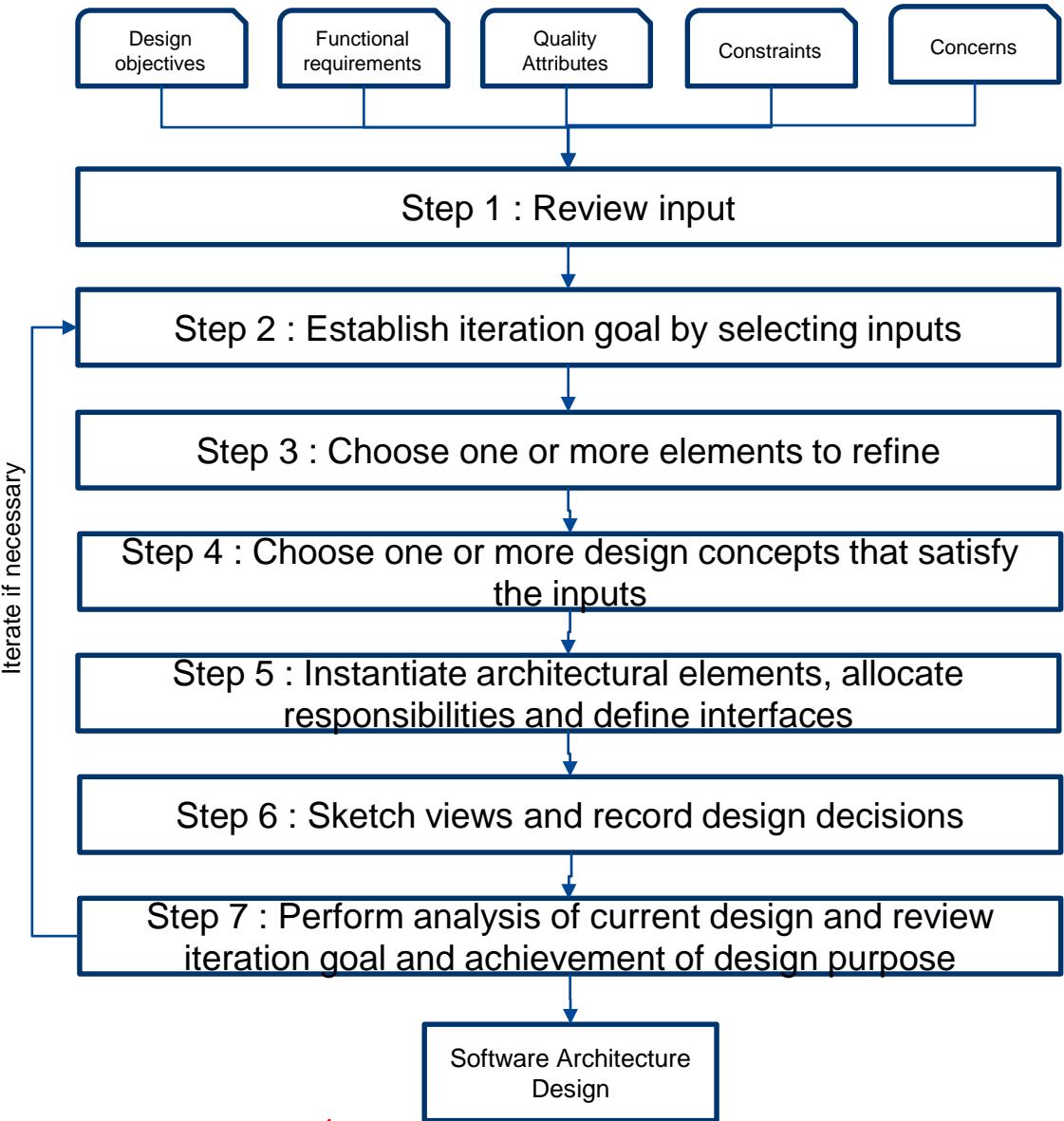
## 5c) Attribute Driven Design

- Schritt 3 Choose one or more elements
  - Die Teilelemente des Gesamtsystems, auf die in dieser Iteration der Fokus liegen soll, werden ausgewählt
  - Diese Elemente werden detaillierter betrachtet



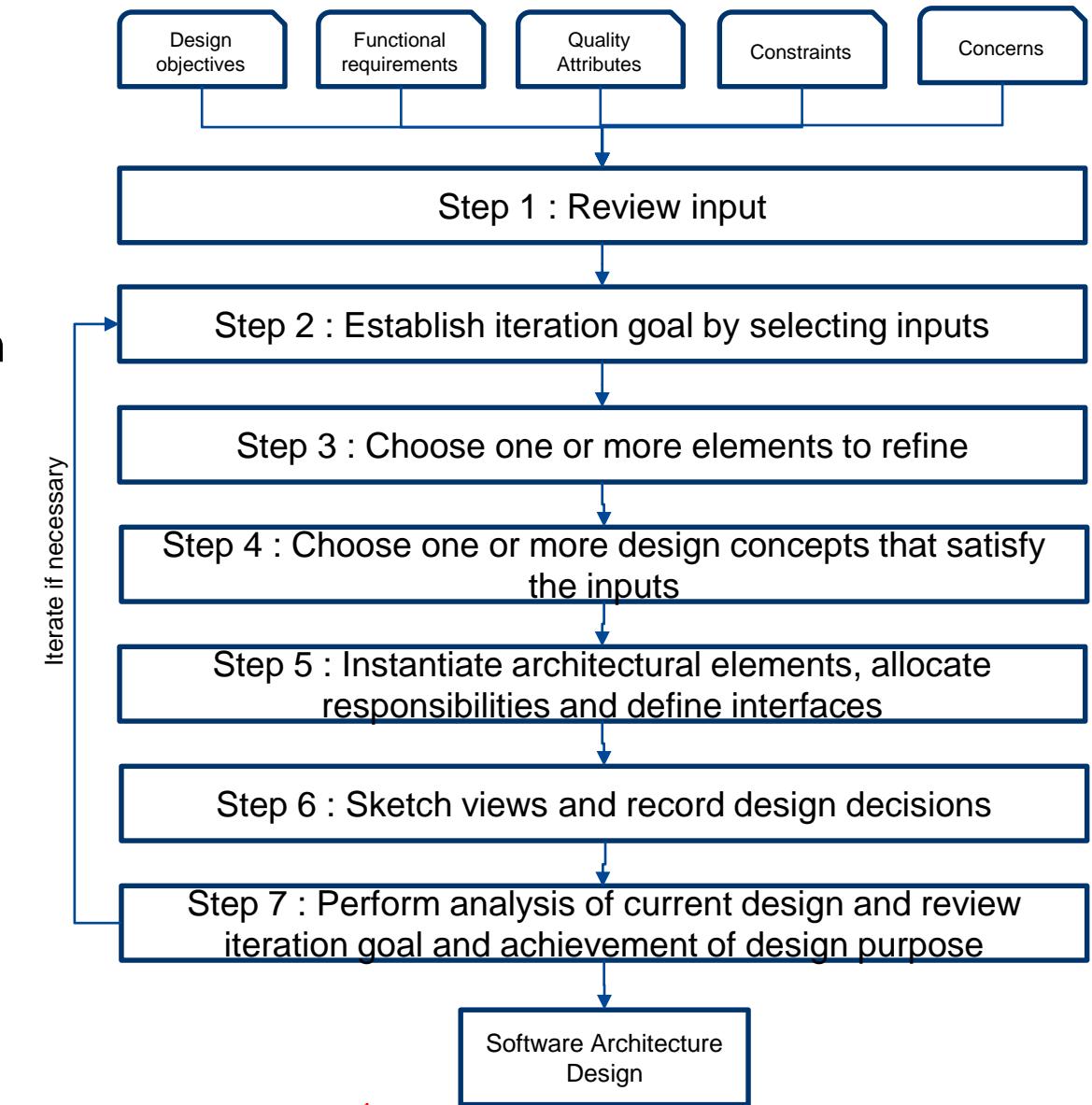
## 5c) Attribute Driven Design

- Schritt 4 Choose one or more elements to refine Design Konzepte zum Erreichen des Iterationsziels werden auf der Basis, der in Schritt 3 gewählten Elementen und der Eingaben, bestimmt
- Design Konzepte sind Design Prinzipien, Architektur Pattern, Referenzarchitekturen, taktische Maßnahmen und extern entwickelte (zugekaufte) Software/Lösungen



## 5c) Attribute Driven Design

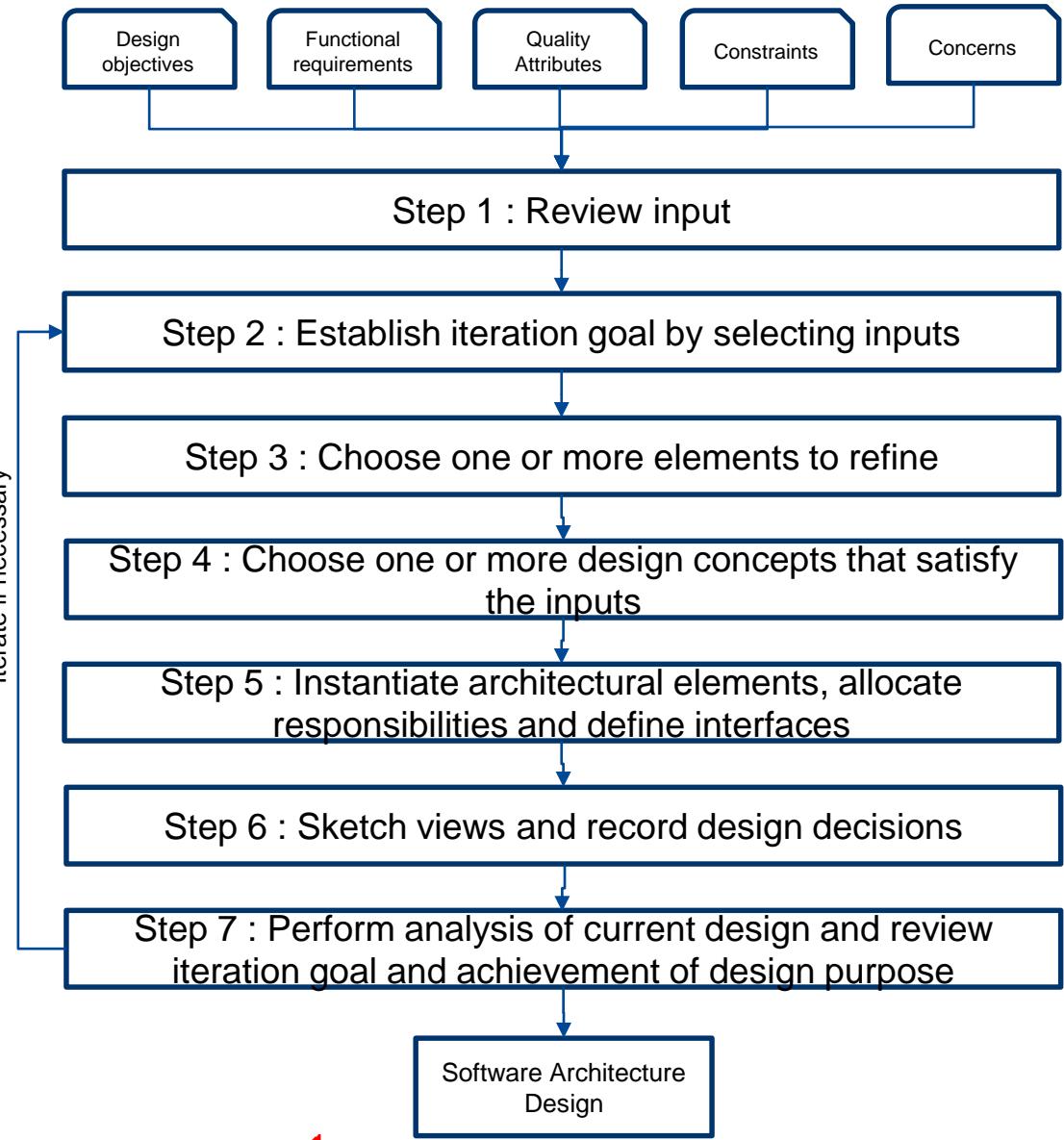
- Schritt 5 Instantiating architectural elements
  - Die Verantwortlichkeiten und Schnittstellen der Elemente werden auf Basis der gewählten Designkonzepte analysiert



## 5c) Attribute Driven Design

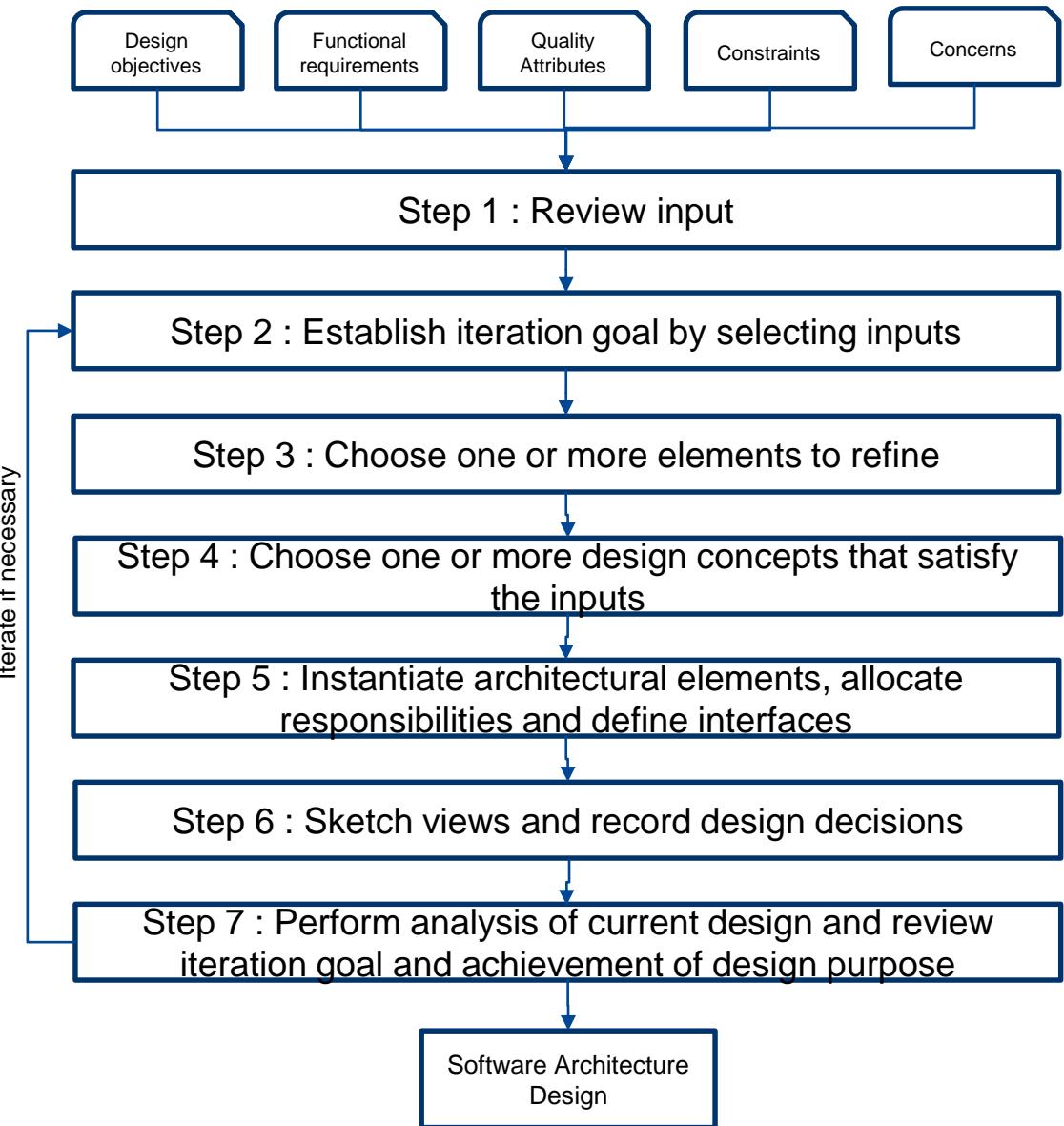
### ■ Schritt 6 Sketch views and record design decisions

- ▶ Sichten auf die Lösung müssen für die verschiedenen Stakeholder skizziert und dokumentiert werden
- ▶ Die Begründung des gewählten Designs wird auch dokumentiert
- ▶ Die erstellten Skizzen beinhalten alle Design Entscheidungen
- ▶ Die Skizzen sollten nicht zu formal und detailliert sein



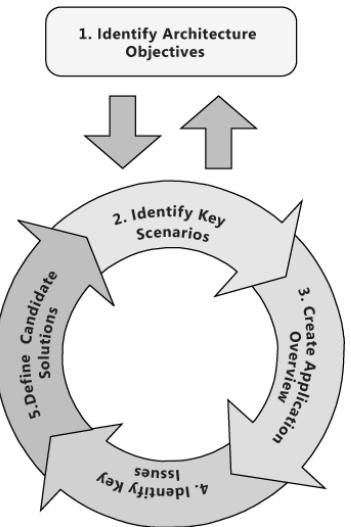
## 5c) Attribute Driven Design

- Schritt 7 Perform analysis of current design
  - ▶ Der Architekt und evtl. weitere Teammitglieder analysieren das Design
  - ▶ Sie prüfen auf Korrektheit und Übereinstimmung mit den Zielen der Iteration
  - ▶ In diesem Schritt wird überprüft und festgelegt, ob weitere Iterationen notwendig sind
  - ▶ Sind keine Iterationen mehr notwendig, ist das Software Architektur Design vollständig



## 5d) Microsofts Technik für Architektur und Design

- Microsofts Agile Architecture Methode ist ein iterativer und incrementeller Ansatz für ein Software Architektur Design
- Zusammengefasst es ist eine Technik
  - ▶ Setzt den Rahmen und fokussiert die Architekturarbeit
  - ▶ Benutzt Szenarien um das Design zu lenken und potentielle Lösungen zu beurteilen
  - ▶ Hilft bei der Auswahl von Anwendungstypen, Plattformen, Architekturstile und Technologien
  - ▶ Hilft beim schnellen Betrachten möglicher Lösungen
  - ▶ Hilft beim Aussuchen von möglichen Pattern



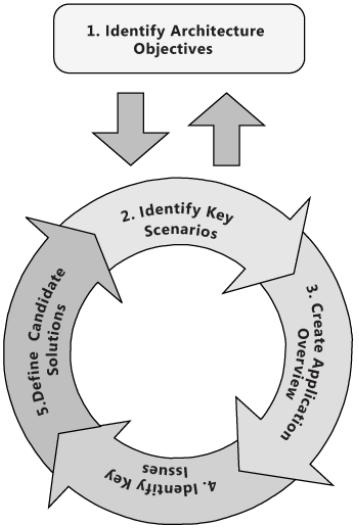
## 5d) Microsofts Technik für Architektur und Design

### ■ Wichtigste Eingaben (Input) in den Prozess

- ▶ Use cases and usage scenarios
- ▶ Functional requirements
- ▶ Non-functional requirements  
(quality attributes such as performance, security, and reliability)
- ▶ Technological requirements
- ▶ Target deployment environment
- ▶ Constraints

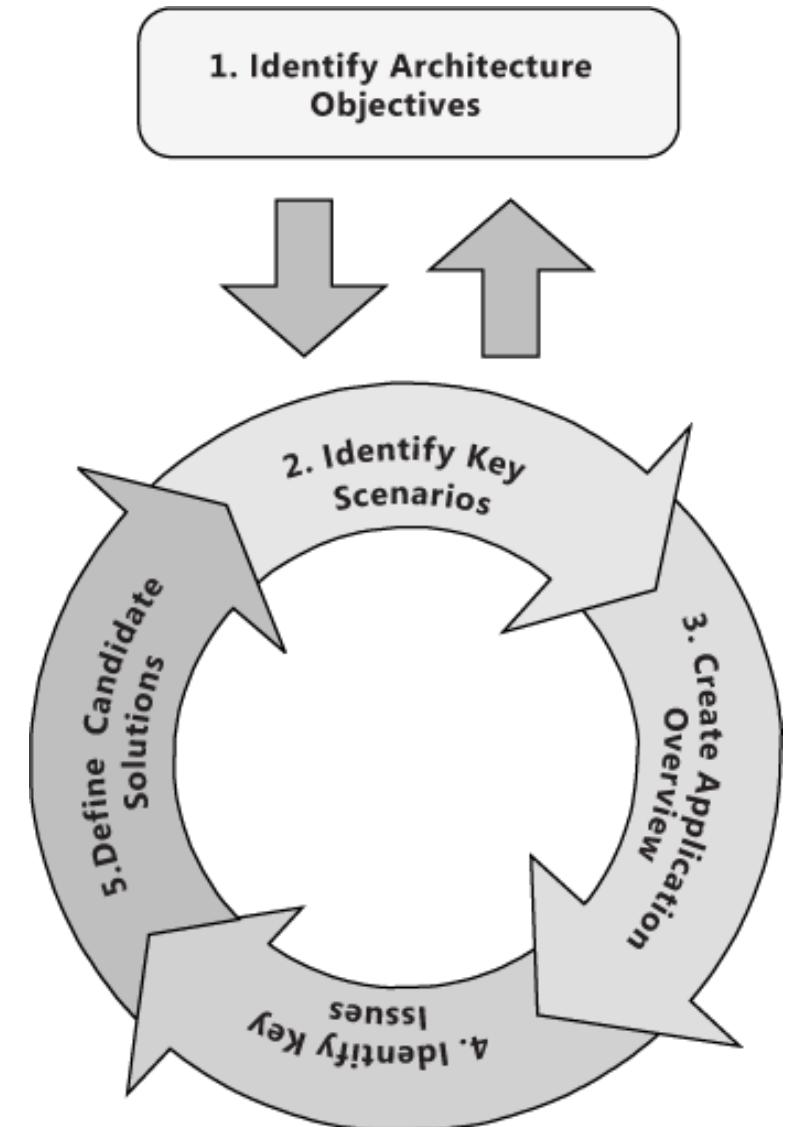
### ■ Ergebnis des Prozesses

- ▶ Architecturally significant use cases
- ▶ Architecture hot spots
- ▶ Candidate architectures
- ▶ Architectural spikes



## 5d) Microsofts Technik für Architektur und Design

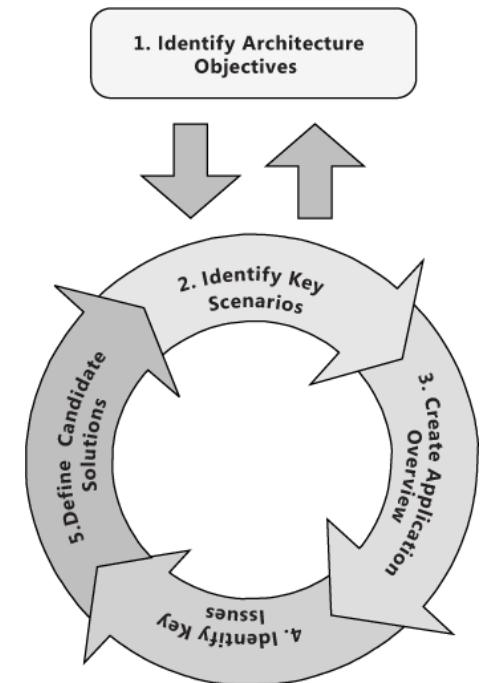
- Zusammenfassung der Schritte
  - ▶ Step 1. Identify Architecture Objectives.
  - ▶ Step 2. Identify Key Scenarios.
  - ▶ Step 3. Create an Application Overview.
  - ▶ Step 4. Analyze Key Hot Spots.
  - ▶ Step 5. Create Candidate Solutions.

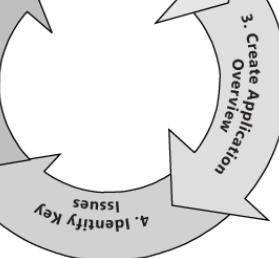
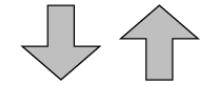


## 5d) Microsofts Technik für Architektur und Design

### ■ Schritt 1 Identify Architecture Objectives

- ▶ In diesem Schritt werden die Ziele, die mit der Architektur erreicht werden sollen, festgelegt
- ▶ Die Festlegung der Ziele sichert die Fokussierung auf die richtigen Problemstellungen
- ▶ Mögliche Ziele könnten sein
  - Prototyperstellung
  - Identifizierung von wichtigen technischen Risiken
  - Testen von möglichen Abläufen
  - Modell- und Verständnisabgleich
- ▶ Die notwendige Zeit und die benötigten Ressourcen werden hier berechnet

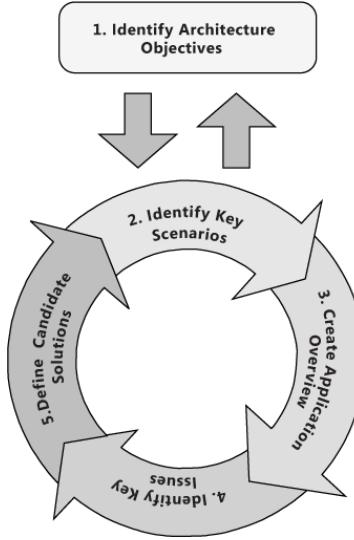




## 5d) Microsofts Technik für Architektur und Design

### ■ Schritt 2 Identify Key Scenarios

- ▶ Identifikation relevanter Szenarien
  - um das Design auf die wichtigsten Punkte zu konzentrieren
  - um geeignete Lösungen zu bestimmen
- ▶ Identifikation für die Architektur signifikanter Use Cases, die folgenden Kriterien genügen
  - Sie sind wichtig für den Erfolg und die Akzeptanz der zu erstellenden Lösung
  - Sie wenden an und prüfen genug vom Design, um den Nutzen für die Architektur feststellen zu können
- ▶ Schlüsselszenarien von User Stories, Business Stories und System Stories werden aufgezeichnet

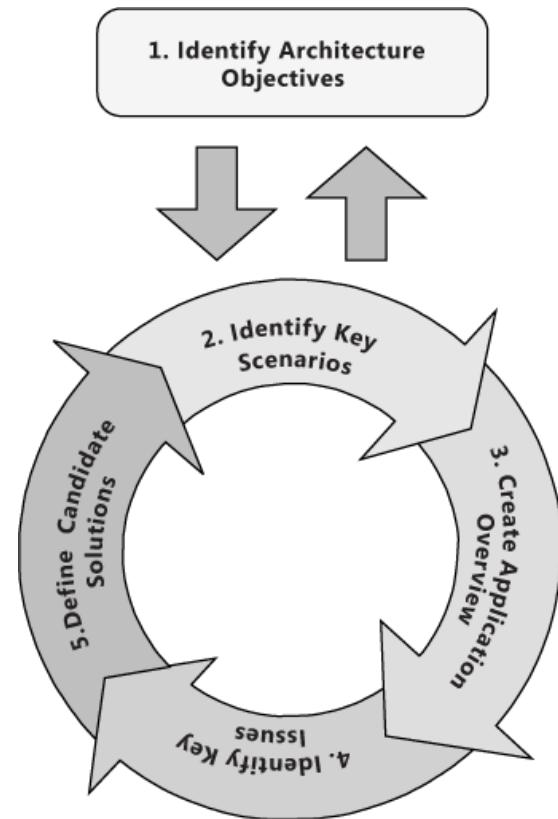
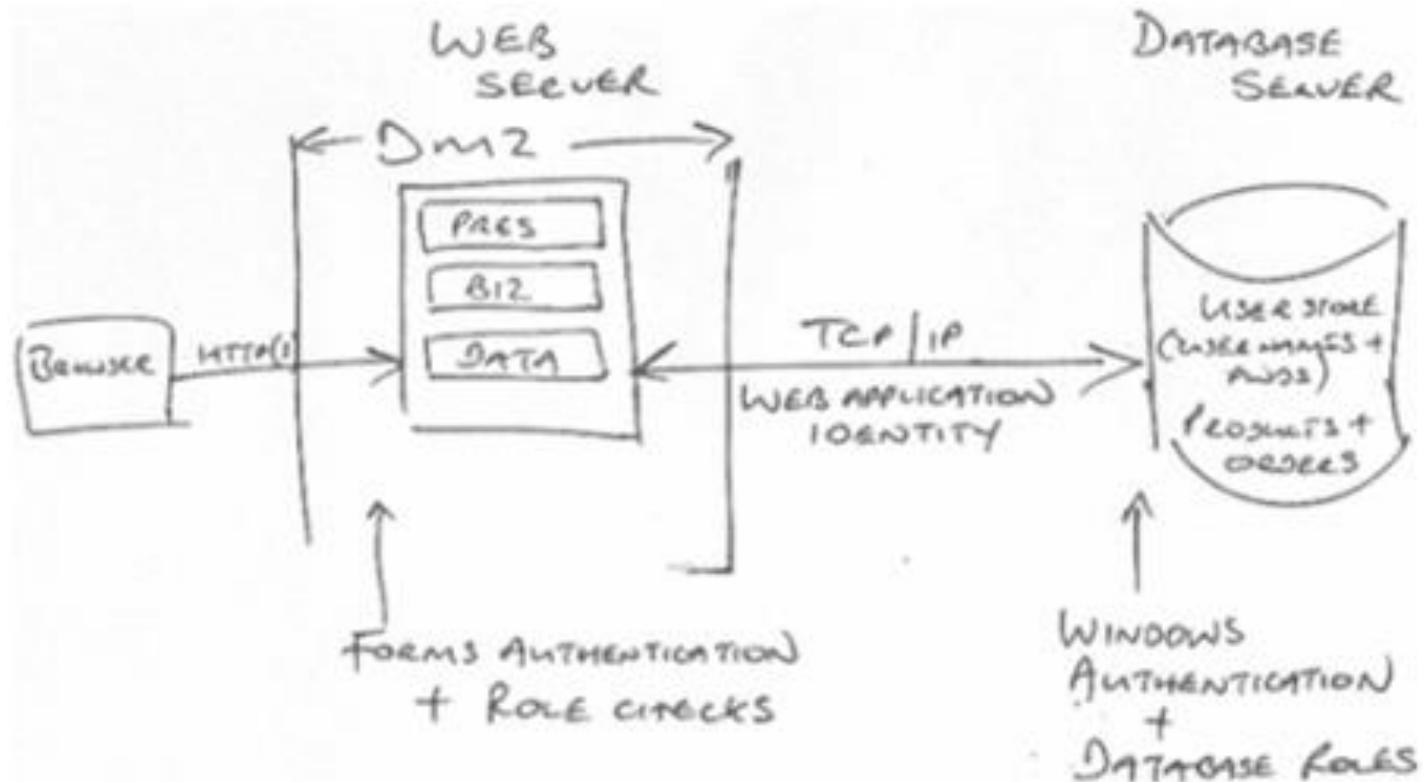


## 5d) Microsofts Technik für Architektur und Design

- Schritt 3 Create an Application Overview
- Die Übersicht über die Anwendung macht eine Architektur mehr real, indem man sie mit echten Einschränkungen und Entscheidungen verbindet
- Eine Anwendungsübersicht wird wie folgt erstellt
  - ▶ Bestimme den Anwendungstyp  
(Mobile, Rich Client, Internet application, Service, Web application, oder Kombinationen davon)
  - ▶ Verstehe die Einschränkungen beim Ausrollen
  - ▶ Verstehe die Zielumgebung und erkenne den Einfluss auf die Architektur
  - ▶ Identifiziere wichtige Architekturstile (SOA, client/server, layered, message bus, Kombinationen)
  - ▶ Bestimme relevante Technologien basierend auf dem Anwendungstyp, dem Architekturstil und den Einschränkungen
- Ein guter Test für das Verständnis und einen Überblick liefert ein sog. Whiteboard

# 5d) Microsofts Technik für Architektur und Design

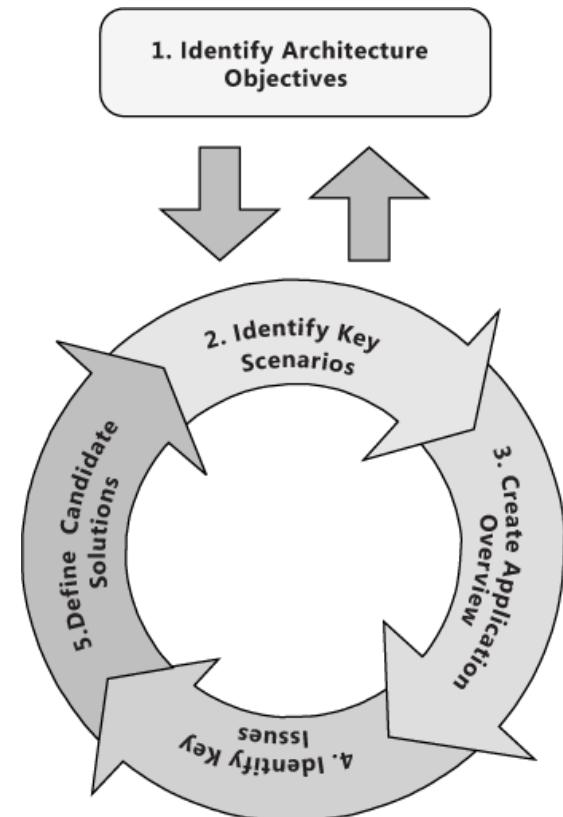
- Schritt 3 Create an Application Overview
  - Whiteboard



## 5d) Microsofts Technik für Architektur und Design

### ■ Schritt 4 Analyze Key Issues

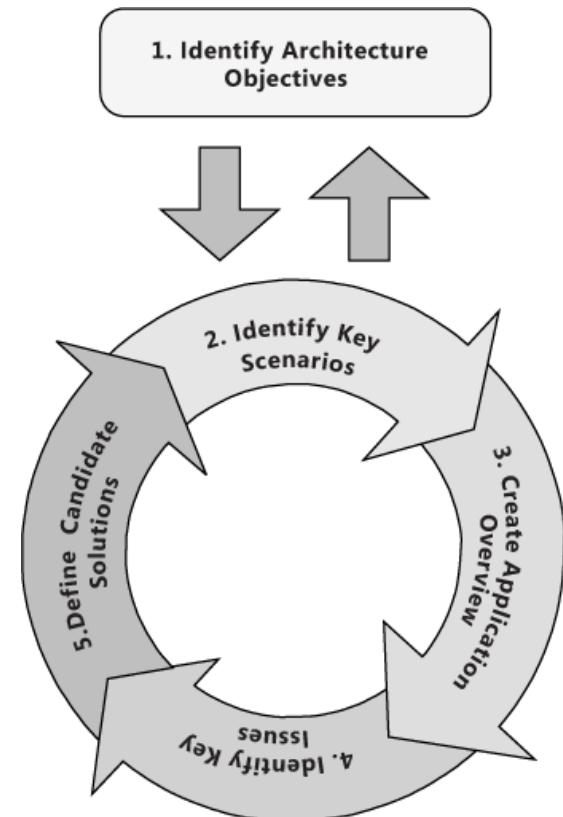
- ▶ Identifiziere die wichtigsten Hindernisse der (in der) Architektur
- ▶ Hindernisse lassen sich typischerweise auf Qualitätsattribute abbilden
- ▶ Gibt Hinweise, wo typische Fehler bei der Anwendungsarchitektur gemacht werden können



## 5d) Microsofts Technik für Architektur und Design

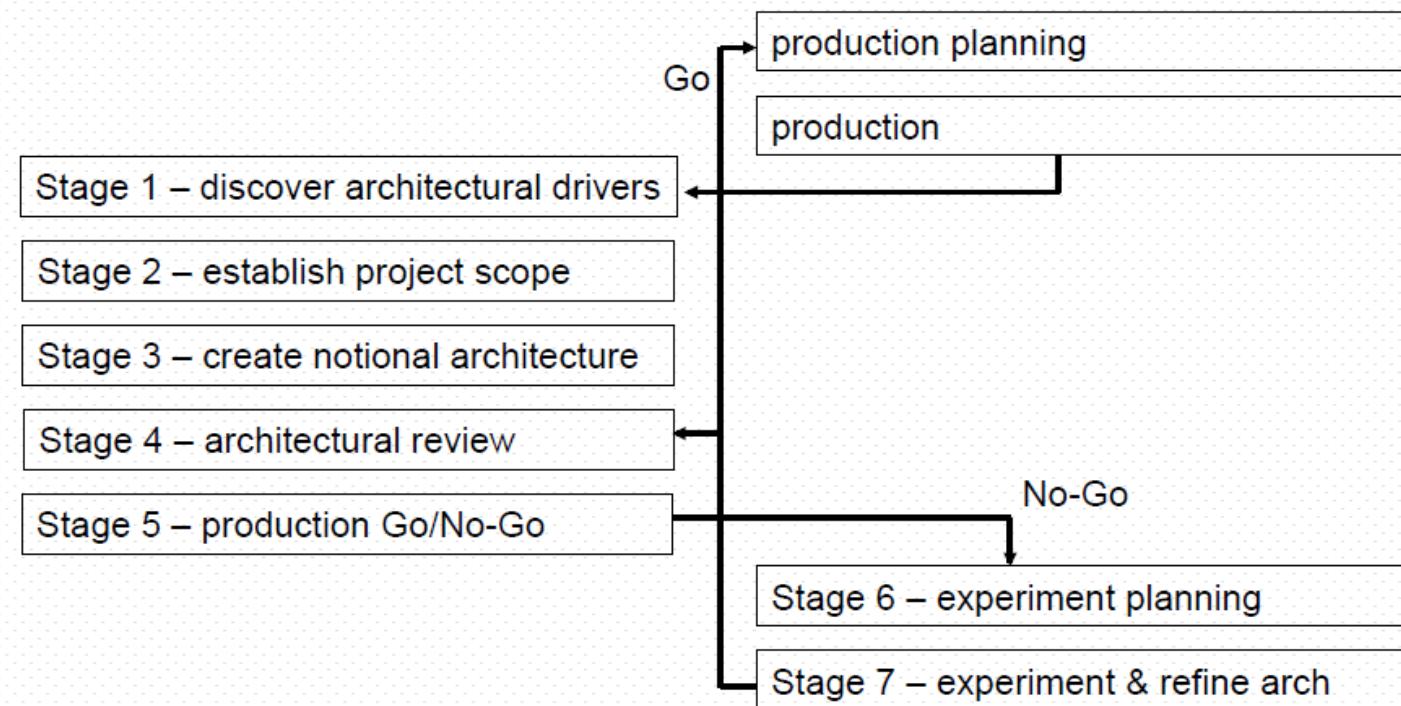
### ■ Schritt 5 Define Candidate Solutions

- ▶ Lösungskandidaten werden definiert
- ▶ In der ersten Iteration entsteht eine neue Architektur in weiteren Iteration wird die Architektur angepasst/erweitert/verbessert
- ▶ Nachdem geeignete Kandidaten in die Gesamtarchitektur aufgenommen wurden, kann die Architektur geprüft und ausgewertet werden
- ▶ Wird in diesem Schritt festgestellt, daß weitere Architekturarbeit notwendig ist, geht es in die nächste Iteration (mit Schritt 2)



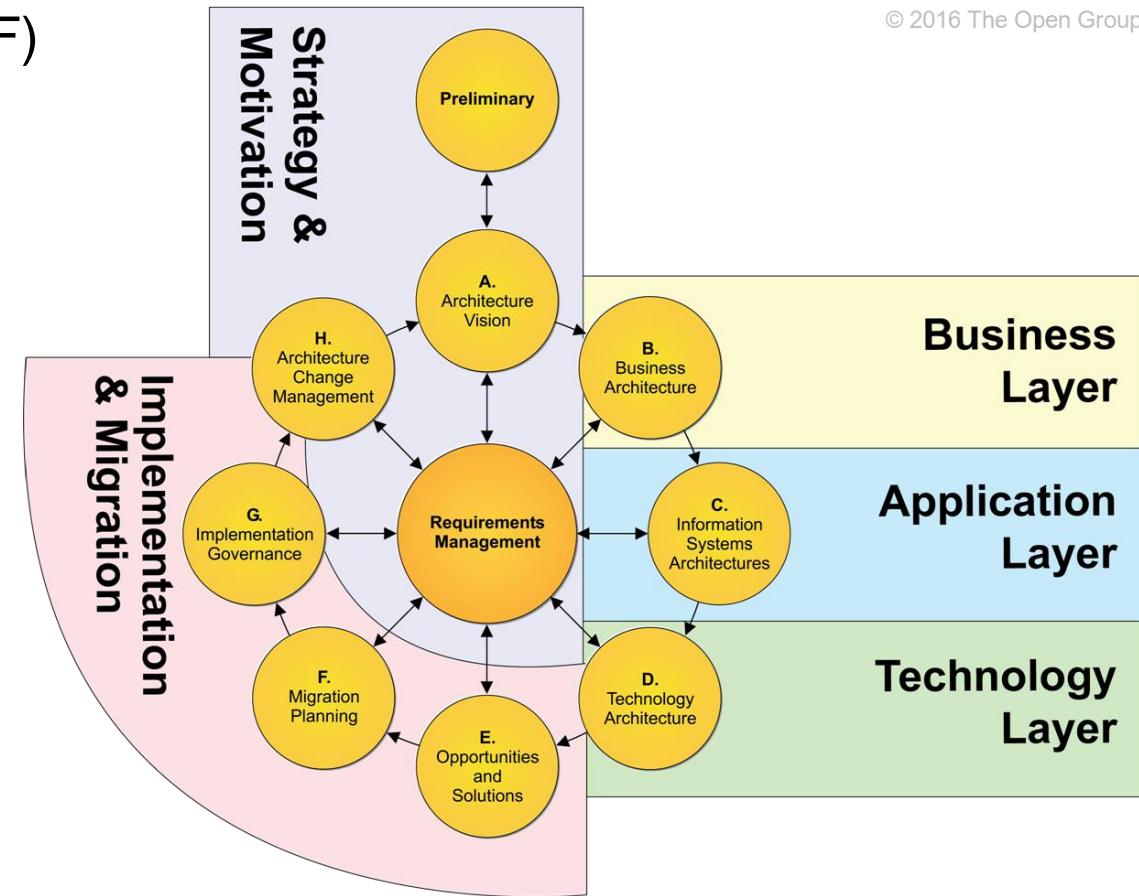
## 5e) Architecture Centric Design Method

- Methode mit Fokus auf das Produkt und den Betrieb
- Die Architektur kann in Schritt 4 bei jeder Iteration geändert werden und ist deshalb in der ersten Iteration i.a. nicht vollständig



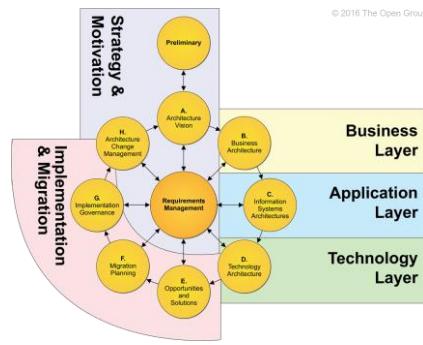
# 5f) Architecture Development Method

- ADM ist Teil eines größeren Frameworks
  - The Open Group Architecture Framework (TOGAF)
  - Framework für Enterprise Architektur
  - ADM besteht aus acht Phasen A-H und einer sog. Preliminary
  - In der Preliminary bereitet sich eine Organisation auf die Einführung von ADM vor (Regeln, Frameworks u.ä.)
  - Jede Phase betrachtet kontinuierlich die Anforderungen



© 2016 The Open Group

## 5f) Architecture Development Method



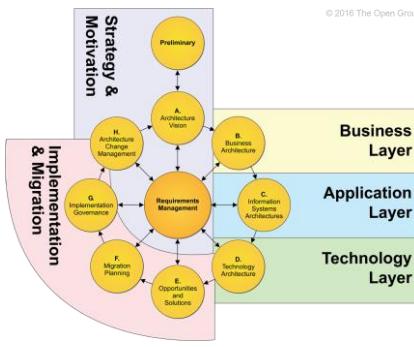
### ■ Phase A Architecture Vision

- ▶ Hier wird die Vision der Architektur mit ihren Fähigkeiten und ihrem Wert entwickelt
- ▶ Teil dieser Phase ist das Festlegen von Umfang, Ziele des Business, Treibern, Einschränkungen, Anforderungen, Rollen, Verantwortlichkeiten und Zeitplänen
- ▶ Alles wird in einem sog. Statement of Architecture Work (SAW) festgehalten
- ▶ Dieses Dokument enthält typischerweise
  - Projektantrag mit Hintergrundinformationen
  - Projektbeschreibung mit Projektumfang (Scope)
  - Übersicht der Architekturvision
  - Rollen, Verantwortlichkeiten, Lieferungen
  - Projekt- und Zeitplan

## 5f) Architecture Development Method

### ■ Phase B Business Architecture

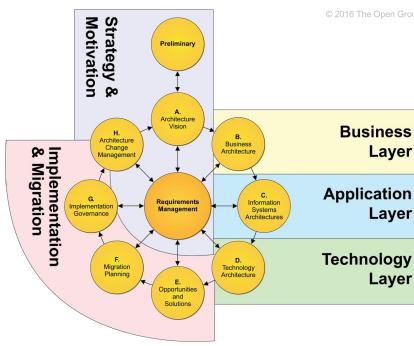
- ▶ Eine der TOGAF Domänen
  - Daten, Anwendungen, Business und Technologie sind die 4 maßgeblichen Domänen
- ▶ Hier wird die Business/Service Strategy der Organisation beschrieben
- ▶ Hier wird das Business und das zugehörige Umfeld beschrieben
- ▶ Hier wird die Ziel-Businessarchitektur festgelegt
- ▶ Um das Ziel zu erreichen und eine Roadmap dahin zu erstellen gibt es die Schritte
  - Die aktuelle (Business)Architektur verstehen
  - Die zukünftige (Business)Architektur aufstellen
  - Die Lücken zwischen beiden Architekturen bestimmen
  - Eine Roadmap für eine Überführung erstellen



# 5f) Architecture Development Method

## ■ Phase C Information Systems Architecture

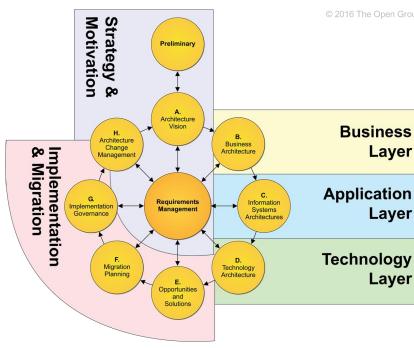
- ▶ Eine der TOGAF Domänen
- ▶ Die Datenarchitektur beschreibt die Daten einer Organisation und wie sie verarbeitet werden
- ▶ Phase A und Phase B bestimmen mögliche Änderungen in Datenmodellen
- ▶ Auch in dieser Phase wird der Unterschied voher/nachher ausgearbeitet



## 5f) Architecture Development Method

### ■ Phase D Technology Architecture

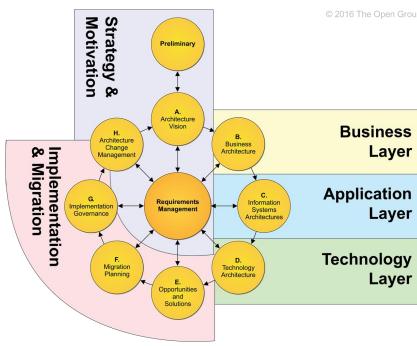
- ▶ Eine der TOGAF Domänen
- ▶ Bezieht die Infrastrukturkomponenten in die Architekturbetrachtung mit ein
  - Hardware und Software
- ▶ Auch hier wird eine Gap Betrachtung gemacht
- ▶ Eventuelle Auswirkungen oder notwendige Änderungen auf die Infrastruktur wird in eigenen Projekten berücksichtigt



# 5f) Architecture Development Method

## ■ Phase E Opportunities and solutions

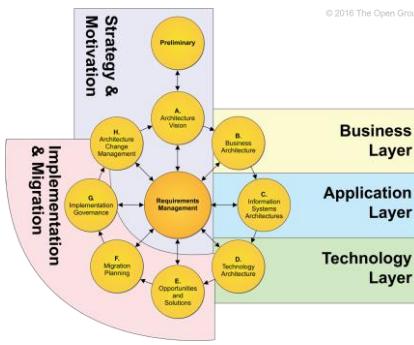
- ▶ Der Output von Phase A,B,C,D wird in eine gemeinsame Roadmap konsolidiert
- ▶ Mögliche Komponenten werden identifiziert
- ▶ Lösungen werden konzipiert
- ▶ Inkrementelle Ansätze können definiert werden



# 5f) Architecture Development Method

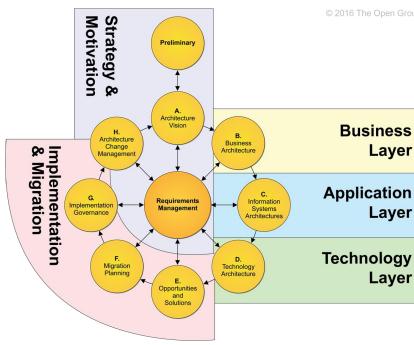
## ■ Phase F Migration Planning

- ▶ Die Implementierung wird geplant
- ▶ Projekte werden definiert
- ▶ Change Management und Projekt Management werden eingebunden



## 5f) Architecture Development Method

- Phase G Implementation Governance
  - ▶ Architekten begleiten und steuern den Entwicklungsprozess
  
- Phase H Architecture Change Management
  - ▶ Architekten begleiten und steuern den Change Management Prozess im Unternehmen



## 5g) Den Fortschritt einer Software Architektur verfolgen

- Während eines Entwicklungsprozesses will man zu jeder Zeit über den Fortschritt informiert sein
- Dazu gibt es seit der Entwicklung agiler Entwicklungsmethoden folgende Hilfsmittel
  - ▶ (Product)Backlog
    - Liste aller Bugs und Features eines Softwareprodukts
    - Während der Entwicklungszyklen (Sprints) ist diese Liste die Grundlage aller Entwicklungen
    - Dazu können einzelnen Einträge von den agilen Teams priorisiert werden
      - DIVE Kriterien
        - Dependencies
        - Insure against risks
        - Business Value
        - Estimated effort
  - ▶ Backlogs verändern sich dynamisch

## 6) Software Entwicklung Prinzipien und Praktiken

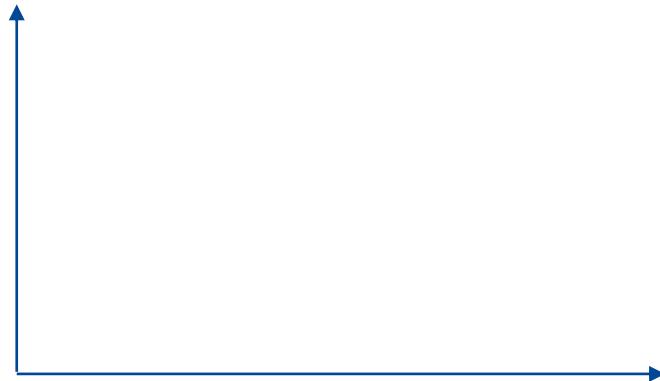
- Ein Software Architekt will Lösungen mit hoher Qualität liefern
- Dafür gibt es erprobte Design Prinzipien und Best Practices
- Software Architekten können diese Prinzipien und Praktiken auswählen und als Vorgabe an die Entwicklung geben
- Diese Prinzipien und Praktiken wurden entwickelt um die Qualität zu erhöhen, den Betrieb zu vereinfachen, die Wiederverwendbarkeit zu erhöhen, Fehler zu erkennen und Softwaresysteme einfacher testen zu können

## 6) Software Entwicklung Prinzipien und Praktiken

- Wir betrachten folgende Punkte
  - a) Lose Kopplung und hohe Kohäsion
  - b) Reduzierung von Komplexität
  - c) Die SOLID Design Prinzipien
  - d) Inversion of Control
  - e) Dependency Injection

## 6a) Lose Kopplung und hohe Kohäsion

- In der Geometrie sind 2 euklidische Vektoren orthogonal, wenn sie im rechten Winkel aufeinander stehen, sie stehen in nur einen Punkt aufeinander und schneiden sich sonst nicht



- Software nennt man orthogonal, wenn ihre einzelnen Module voneinander unabhängig sind
- Änderungen an einem Modul benötigen keine Änderungen an anderen Modulen
- Orthogonale Systeme sind lose gekoppelt und in hohem Maße geschlossen

## 6a) Lose Kopplung und hohe Kohäsion

- Lose Kopplung ist das Ausmaß der Abhängigkeit von Software Modulen zueinander
- Der Grad der Kopplung geht von lose (loos) bis eng (tight)
- Die Qualität eines Software Systems ist direct abhängig vom Grad der Kopplung
- Software Module, die eng gekoppelt sind, verringern die Wartbarkeit und verhindern einfache Änderungen am Code (weil Änderungen an einen Modul, Änderungen an anderen Modulen verursachen)
- Bei loser Kopplung ist die getrennt Entwicklung von Modulen einfacher
- Enge Kopplung schränkt die Wiederverwendbarkeit ein

## 6a) Lose Kopplung und hohe Kohäsion

- Es gibt verschiedene Arten von Kopplung
  - ▶ Content coupling auch pathological coupling genannt (die schärfste Form der Kopplung)
    - Wenn ein Modul interne oder private Informationen eines anderen Moduls benutzt oder abhängig vom internen Verhalten eines anderen Moduls ist
    - Z.B. Änderung von privaten Attributen, Aufruf von geschützten Funktionen
    - **Sollte auf keinen Fall benutzt werden**
  - ▶ Common coupling auch global coupling genannt
    - Wenn Module globale Daten (Variablen) gemeinsam nutzen
    - Wird oft bei Konfigurationen (Daten) benutzt – auf spezielle Lösungen ausweichen
    - Globale Konstanten statt Variablen benutzen
    - **Sollte auf keinen Fall benutzt werden**

## 6a) Lose Kopplung und hohe Kohäsion

- Es gibt verschiedene Arten von Kopplung
  - ▶ External coupling
    - Wenn mehrere Module Teile aus der gemeinsamen Umgebung nutzen
    - Beispiele : Datenformate, Interfaces, Formate, Tools oder Devices
    - **Manchmal nicht vermeidbar, Anzahl der Module mit dieser Kopplung gering halten**
  - ▶ Control coupling
    - Moderate Kopplung
    - Ein Modul nutzt Kontrollmechanismen über andere Module (Flag)
    - **Manchmal akzeptierbar, Anzahl gering halten**
  - ▶ Stamp coupling (data-structured coupling)
    - Wenn Module eine gemeinsame zusammengesetzte Datenstruktur nutzen
    - Nicht alle Inhalte werden von allen Modulen genutzt
    - **akzeptierbar**

## 6a) Lose Kopplung und hohe Kohäsion

- Es gibt verschiedene Arten von Kopplung
  - ▶ Data coupling
    - Austausch von Parametern
    - Alle Datenfelder werden benutzt
    - akzeptierbar
  - ▶ Message coupling
    - Module senden Nachrichten ohne Parameter
    - Nur der Name der Nachricht oder der aufgerufenen Methode ist bekannt
    - Niedrigster Level von Kopplung
    - akzeptierbar
  - ▶ No coupling
    - Keine Abhängigkeiten zwischen Modulen
    - Ideal und immer anzustreben

## 6a) Lose Kopplung und hohe Kohäsion

- The Law of Demeter (LoD) / principle of least knowledge
  - ▶ Um lose Kopplung zu erreichen, kann man dieses Prinzip anwenden
  - ▶ Dieses Prinzip folgt dem “only talk to your friends” Idiom
  - ▶ Limitiert die Kommunikation unter Modulen (nur die Freunde reden miteinander)
  - ▶ Idealerweise sollte ein Objekt in einem Modul nur Methoden von sich selbst, von Objekten, die ihm übergeben wurden, von Unterobjekten, von Objekten die es erzeugt hat oder von globalen Objekten aufrufen
  - ▶ Ein Modul sollte so wenig wie möglich von anderen Modulen wissen
  - ▶ Das fördert lose Kopplung
  - ▶ Information hiding hilft dabei
- Lose Kopplung ist eines der wichtigsten Designprinzipien

## 6a) Lose Kopplung und hohe Kohäsion

- Hohe Kohäsion (High cohesion)
  - ▶ Ist ein Maß für den Zusammenhalt von Elementen innerhalb eines Moduls
  - ▶ Kohäsion ist ein qualitatives Maß an Konsistenz innerhalb eines Moduls
  - ▶ Je höher das Maß der Kohäsion, je besser
  - ▶ Softwaremodule mit geringer Kohäsion sind schwerer wartbar
    - Änderungen in Teilbereichen führen oft zu Änderungen in anderen Bereichen
    - Sie sind meistens schwerer verständlich und deshalb schwerer änderbar/wartbar
  - ▶ Softwaremodule mit geringer Kohäsion sind schlechter wiederverwendbar
  - ▶ Softwaremodule mit hoher Kohäsion wirken wie eine zusammengehörige Einheit

## 6a) Lose Kopplung und hohe Kohäsion

### ■ Die Formen der Kohäsion

#### ▶ Coincidental cohesion

- Elemente in einem Modul sind willkürlich gruppiert
- Es gibt keine Beziehungen oder Gemeinsamkeiten
- Sieht man manchmal in Utilities oder Helper Klassen
- Ist die niedrigste Form der Kohäsion und sollte unbedingt vermieden werden
- Module mit dieser Kohäsion sollten neu entwickelt werden (Trennung)

#### ▶ Logical cohesion

- Elemente in einem Modul sind nach einer logischen Beziehung gruppiert
- Beispiel : ein Modul, das alle möglichen IO Operationen behandelt (Seriell, TCP/IP, ...)
  - Es könnte besser sein und würde die Kohäsion verbessern, wenn es eigene Module pro IO Operation geben würde
- Kohäsion ist niedrig und sollte verbessert werden

# 6a) Lose Kopplung und hohe Kohäsion

## ■ Die Formen der Kohäsion

### ▶ Temporal cohesion

- Elemente innerhalb eines Moduls sind nach zeitlicher Zusammengehörigkeit gruppiert
  - Zusammengehörigkeit bei einer Abarbeitung
  - Gleichzeitigkeit der Ausführung
- Beispiel : Elemente gruppiert nach System Startup/Shutdown
- Elemente sollten in verschiedene Module aufgetrennt werden (mit jeweils eigenen Zweck)

### ▶ Procedural cohesion

- Elemente innerhalb eines Moduls sind nach einer bestimmten Verarbeitungssequenz gruppiert
- Beispiel : Zahlungsverarbeitung nach einer Bestellung
  - Erfassen der Kontoinformationen → Validierung der Zahlungsmethode → Konto gedeckt → Speichern des Auftrags → Prüfen der Lager → ...
- Die einzelnen Punkte gehören in der Reihenfolge zusammen, aber nicht logisch
- Moderate Kohäsion – kann akzeptiert werden

## 6a) Lose Kopplung und hohe Kohäsion

- Die Formen der Kohäsion
  - ▶ Communicational cohesion
    - Elemente sind nach gemeinsamen Inputs und/oder Outputs gruppiert (z.B. verändern die gleiche Datenstruktur)
    - Beispiel : Datenstruktur, die einen Shopping Cart eines Kunden darstellt und alle Elemente, die ihn lesen oder verändern sind zusammen in einem Modul gruppiert
    - Moderate Kohäsion – akzeptierbar
  - ▶ Sequential cohesion
    - Elemente sind in einem Modul gruppiert, da der Output des einen Elements als Input für das nächste Element dient
    - Findet man oft bei Formatierungs- oder Validierungsroutinen
    - Moderate Kohäsion - akzeptierbar

## 6a) Lose Kopplung und hohe Kohäsion

- Die Formen der Kohäsion
  - ▶ Functional cohesion
    - Alle Elemente in einem Modul dienen dem gleichen, eindeutigen und klar definierten Zweck
    - Höchste Ebene der Kohäsion und immer anzustreben
    - Beispiel : Modul, das Elemente besitzt, die Dateien lesen – Modul, das die Transportkosten berechnet
- Architekten sollten immer Module entwerfen, die eine hohe Kohäsion besitzen
  - ▶ Jedes Modul sollte eine eindeutige, klar definierte Aufgabe besitzen
  - ▶ Alle Elemente eines Moduls sollten gemeinsam auf diese Aufgabe hinarbeiten
  - ▶ Alle anderen Elemente sind Kandidaten, sie in andere Module zu bringen oder in neue Module aufzunehmen
- Kohäsion und Kopplung sind voneinander abhängig
  - ▶ Hohe Kohäsion unterstützt lose Kopplung
  - ▶ Schwache Kohäsion erzeugt enge Kopplung

## 6b) Reduzierung von Komplexität

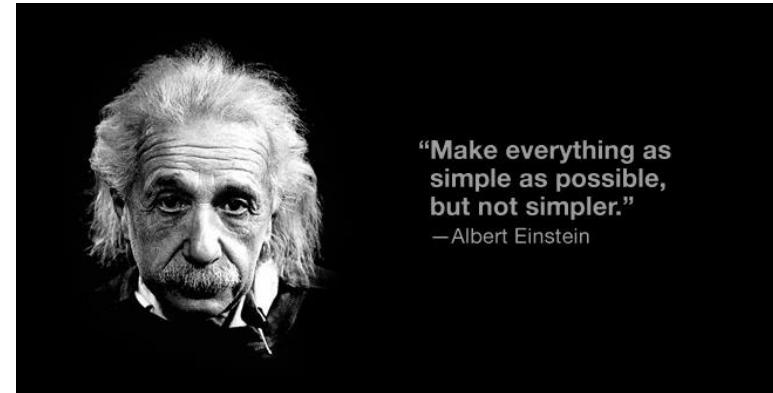
- Software bleibt immer komplex und Komplexität erzeugt eine Reihe von Problemen
  - ▶ Zeitverzögerungen bei der Auslieferung
  - ▶ Überschreiten von geplanten Kosten
  - ▶ Unvorhergesehenes oder ungeplantes Verhalten
  - ▶ Sicherheitslücken oder Probleme beim Auffinden/Schließen solcher Lücken
  - ▶ Geringere Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit

## 6b) Reduzierung von Komplexität

### ■ Prinzipien zur Reduzierung von Komplexität

#### ▶ KISS

- Keep it simple, stupid
- Allgemeingültiges Prinzip : Einfachheit gewinnt
- Urheber : Kelly Johnson, Lockheed Corporation, Flugzeugbau und –reparatur
- Ganz wichtig in der Softwareentwicklung und in der Software Architektur
- Hohe Komplexität beeinflusst direct und indirect andere Qualitätsattribute
- Beispiele
  - Vermeidung von Duplizierung (copy & paste)
  - Vermeidung unnötiger Eigenschaften
  - Verstecken von Komplexität und Design Entscheidungen
  - Folgen von Standards und Vermeiden von Abweichungen und Überraschungen
  - Kann übertrieben werden



"Make everything as simple as possible,  
but not simpler."  
—Albert Einstein

## 6b) Reduzierung von Komplexität

### ■ Prinzipien zur Reduzierung von Komplexität

- ▶ Keine Redundanz (Don't repeat yourself DRY)
  - Definitionen und Beschreibungen an einer Stelle
  - Copy & Paste Programmierung
  - Magic strings im Code : string result = cache.get("FilePathCacheKey");
  - Wiederverwendung anstreben
  - Abstraktion anwenden
  - Kann auch übertrieben werden

## 6b) Reduzierung von Komplexität

### ■ Prinzipien zur Reduzierung von Komplexität

#### ▶ Information hiding

- Module verstecken Implementierungsdetails
- Entkoppelt die interne Verarbeitung eines Moduls vom Modulbenutzer
- Modulbenutzer sehen nur die öffentliche Schnittstelle eines Moduls
- Wichtig in allen Phasen des Designs
- Trennt das Was vom Wie

#### ▶ YAGNI

- You aren't gonna need it
- Kommt von der Softwareentwicklungsmethode XP (extreme Programming)
- Keine Funktionalität implementieren, die nicht benötigt wird
  - Sog. Over-engineering
- Funktionalität, die nicht implementiert ist, hat Zeit gespart

## 6b) Reduzierung von Komplexität

### ■ Prinzipien zur Reduzierung von Komplexität

#### ▶ Separation of Concern

- Verantwortlichkeiten trennen
- Bausteinen haben jeweils spezifische Verantwortlichkeiten, Zuständigkeiten oder Aufgaben  
Verantwortlichkeiten gibt es für „Wissen“ (Verantwortung für Informationen) und „Handeln“ (Verantwortung für Aktivitäten)
- Geben Sie den Bausteinen Ihrer Systeme jeweils spezifische Verantwortlichkeiten, Zuständigkeiten oder Aufgaben. Nennen Sie diese Verantwortlichkeiten beim Namen!  
Beschreiben Sie sie in kurzen, präzisen Sätzen. Kommen darin mehrere „und“ vor, könnten Sie versehentlich mehrere Belange oder Verantwortlichkeiten vermischt haben
- MVC (Model-View-Controller) ist ein Beispiel dafür
- Ein Beispiel aus der Webprogrammierung
  - HTML (Inhalt), CSS (Aussehen), Javascript (Verhalten)

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

- S = Single Responsibility
- O = Open-Close Principle
- L = Liskov-Substitution Principle
- I = Interface Segregation Principle
- D = Dependency Inversion Principle
  
- Software Architekten sollten diese Prinzipien genau kennen
- Sind Anleitung, kein Muß – aber sehr hilfreich

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

### ■ S = Single Responsibility

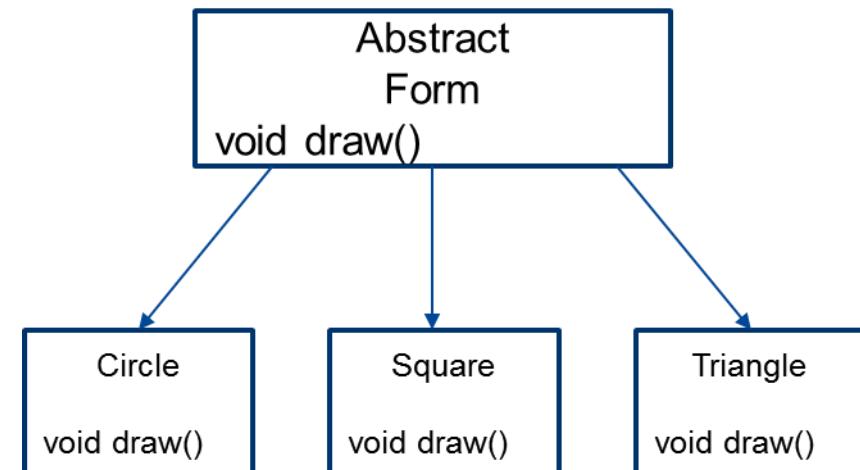
- ▶ Jede Klasse sollte genau eine Verantwortlichkeit besitzen
- ▶ Hat eine Klasse mehr als einen Grund geändert zu werden, hat sie mehr als eine Verantwortlichkeit
- ▶ Klassen mit mehr als einer Verantwortlichkeit sollten in mehrere kleinere Klassen mit jeweils einer einzigen Verantwortlichkeit aufgeteilt werden
- ▶ Vorgehensweise
  - Aufräumen
  - Abstraktion
  - Interfaces
  - Einsatz von zusätzlichen Patterns

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

### ■ O = Open-Close Principle

- ▶ Klassen können ohne bestehenden Code zu ändern, erweitert werden
- ▶ Minimiert Änderungsrisiken
  - Beispiel :

```
Void Draw(Form f)
{
    if (f.type == circle)
        drawCircle(f);
    else if (f.type == square)
        drawSquare(f);
    ....
}
```



Polymorphismus

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

- O = Open-Close Principle
  - ▶ Bertrand Meyer 1988 : „Software entities should be open for extension, but closed for modification“
- Robert C. Martin 2003
  - ▶ „Open for extension“ This means that the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.

„Closed for modification“ Extending the behavior of a module does not result in changes to the source or binary code of the module. The binary executable version of the module whether in a linkable library, a DLL, or a Java .jar, remains untouched.

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

- O = Open-Close Principle
- Erlaubte Ausnahmen
  - ▶ Bug fixes
  - ▶ Änderungen am Code, die keinerlei Auswirkung auf den Nutzer dieses Codes hat (lose Koppelung)
- Änderungen in einer Klasse dürfen keine Änderungen in einer anderen Klasse auslösen

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

### ■ L = Liskov-Substitution Principle

- ▶ Unterklassen müssen ihre Oberklassen ersetzen können
- ▶ Beispiel Kreis als Unterklasse von Ellipse
- ▶ Unterklassen sollen nicht mehr erwarten und nicht weniger liefern als ihre Oberklassen (Robert Martin)
- ▶ Änderungen in Unterklassen dürfen das Verhalten von Oberklassen nicht ändern
- ▶ Methoden und Attribute einer Oberklasse sollten sinnvoll definiert sein und auch für alle Unterklassen Sinn machen

# 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

## ■ L = Liskov-Substitution Principle

### ► Beispiel

```
public class Rectangle
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }

    public int CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    private int _width;
    private int _height;

    public override int Width
    {
        get { return _width; }
        set
        {
            _width = value;
            _height = value;
        }
    }
}
```



```
Rectangle rect = new Rectangle { Width = 5; Height = 4; }
? = rect.CalculateArea();
```

```
Square sqr = new Square { Width = 4; }
? = sqr.CalculateArea();
```

```
Rectangle sqrforcorrect = new Square { Width = 3; Height = 2; }
? = sqrforcorrect.CalculateArea();
```

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

### ■ I = Interface Segregation Principle

- ▶ Interfaces beschreiben Methoden und Attribute, implementieren aber nicht
- ▶ Klassen, die ein Interface implementieren, liefern die Implementierung der Methoden und Attribute
- ▶ Das Interface Segregation Principle sagt aus, daß Implementierungen nicht von Methoden und/oder Attributen abhängig sein dürfen, die sie nicht benutzen
- ▶ Oder : Interfaces sollten so klein und überschaubar wie möglich sein und Methoden und Attribute nicht voneinander abhängig sein

# 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

## ■ I = Interface Segregation Principle

### ► Beispiel

```
public interface iProduct
{
    int ProductId { get; set; }
    string Title { get; set; }
    int AuthorId { get; set; }
    decimal Price { get; set; }
}

public class Book : iProduct
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public decimal Price { get; set; }
}

public class Movie : iProduct
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId
    {
        get => throw new NotSupportedException();
        set => throw new NotSupportedException();
    }
    public decimal Price { get; set; }
    public int RunningTime { get; set; }
}
```



```
public interface iProduct
{
    int ProductId { get; set; }
    string Title { get; set; }
    decimal Price { get; set; }
}

public class iBook : iProduct
{
    int AuthorId { get; set; }
}

public interface iMovie : iProduct
{
    int RunningTime { get; set; }
}
```

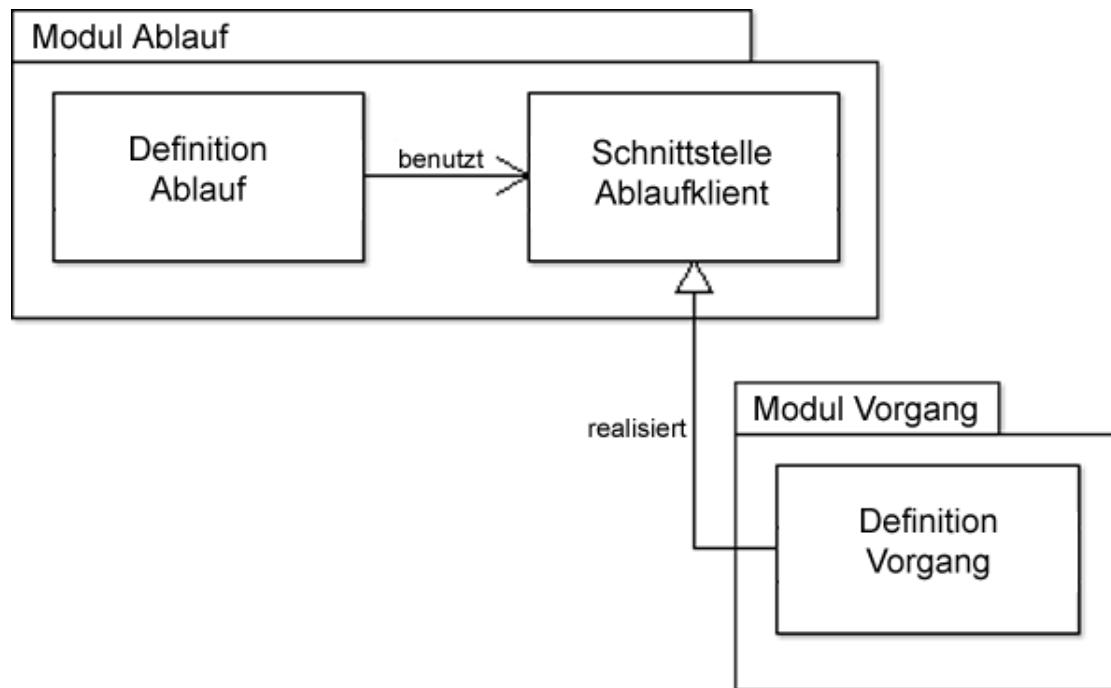
```
public class Book : iBook
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public int AuthorId { get; set; }
    public decimal Price { get; set; }
}

public class Movie : iMovie
{
    public int ProductId { get; set; }
    public string Title { get; set; }
    public decimal Price { get; set; }
    public int RunningTime { get; set; }
}
```

## 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

### ■ D = 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.



# 6c) Die SOLID Prinzipien des objektorientierten Entwurfs

## ■ D = Dependency Inversion Principle

### ► Beispiel

```
public class Lampe {  
    private boolean leuchtet;  
  
    public void anschalten() {  
        leuchtet = true;  
    }  
  
    public void ausschalten() {  
        leuchtet = false;  
    }  
  
    public class Schalter {  
        private Lampe lampe;  
        private boolean gedrueckt;  
  
        public Schalter(Lampe lampe) {  
            this.lampe = lampe;  
        }  
  
        public void drueckeSchalter() {  
            gedrueckt = !gedrueckt;  
            if(gedrueckt) {  
                lampe.anSchalten();  
            } else {  
                lampe.ausschalten();  
            }  
        }  
    }  
}
```



```
public interface SchalterClient {  
    public void anschalten();  
    public void ausschalten();  
}  
  
public class Lampe implements SchalterClient {  
    private boolean leuchtet;  
  
    public void anschalten() {  
        leuchtet = true;  
    }  
  
    public void ausschalten() {  
        leuchtet = false;  
    }  
}
```

```
public class Schalter {  
    private SchalterClient client;  
    private boolean gedrueckt;  
  
    public Schalter(SchalterClient client) {  
        this.client = client;  
    }  
  
    public void drueckeSchalter() {  
        gedrueckt = !gedrueckt;  
        if(gedrueckt) {  
            client.anSchalten();  
        } else {  
            client.ausschalten();  
        }  
    }  
}
```

## 6d) Inversion of Control

- Der Begriff Inversion of Control (IoC, deutsch Umkehrung der Steuerung oder Steuerungsumkehr) bezeichnet ein Umsetzungsparadigma, das u. a. in der objektorientierten Programmierung Anwendung findet.
- Dieses Paradigma beschreibt die Arbeitsweise von Frameworks: eine Funktion eines Anwendungsprogramms wird bei einer Standardbibliothek registriert und von dieser zu einem späteren Zeitpunkt aufgerufen. Statt dass die Anwendung den Kontrollfluss steuert und lediglich Standardfunktionen benutzt, wird die Steuerung der Ausführung bestimmter Unterprogramme an das Framework abgegeben.
- Ein einfaches Beispiel einer solchen Umkehrung sind Listener (gemäß dem Beobachter-Muster), aber auch Java-Applets und Servlets folgen diesem Entwurfsmuster. Plug-ins und Rückruffunktionen (callback) sind weitere Beispiele dafür, die Steuerung einzelner Programmteile einem (Framework-)Objekt zu überlassen.

## 6e) Dependency Injection

- Dependency Injection ist eine Technik wo ein Objekt die Beziehungen eines anderen Objektes liefert. Eine Beziehung ist ein Objekt, das genutzt werden kann, z.B. als Service. Nicht der Klient spezifiziert, welche Services er nutzt, sondern jemand anderes. Die "Injection" bezieht sich auf die Übergabe der Beziehung (Service) an das Objekt (Klient), der ihn nutzt.  
Die Übergabe des Service an den Klient, anstatt ihn das selbst bestimmen zu lassen, ist die Grundanforderung dieses Prinzips.
- Der Sinn hinter Dependency Injection ist das Erreichen von Separation of Concerns der Erzeugung und Benutzung von Objekten. Das erhöht die Lesbarkeit und Wiederverwendbarkeit von Code.
- Dependency Injection ist eine Form der Technik "Inversion of Control". Der Klient über gibt die Verantwortung Beziehungen zu liefern an externen Code (Injector). Der Klient darf den Injector Code nicht direct aufrufen. Der "einspeisende" Code konstruiert die Services und ruft den Klient, um sie einzufügen. Der Klient kennt den "einspeisenden" Code, die Services und wie sie konstruiert werden, nicht. Er weiß auch nicht, welchen Service er gerade nutzt.
- Das trennt die Verantwortlichkeiten der Erstellung von der Nutzung.

## 6e) Dependency Injection

- Typen von Dependency Injection
  - ▶ Constructor Injection
    - Die Beziehungen werden durch den Konstruktor des Klienten geliefert
  - ▶ Setter/Property Injection
    - Der Klient stellt eine setter-Methode für die Injection zur Verfügung
  - ▶ Interface/Method Injection
    - Das Interface der Beziehung stellt eine Injectionsmethode zur Verfügung
    - Klienten müssen ein Interface implementieren, das eine setter-Methode implementiert

# 6e) Dependency Injection

## ■ Beispiel

```
// Constructor  
Client(Service service)  
{  
    // Save the reference to the passed-in service inside this client  
    this.service = service;  
}  
  
// Setter method  
public void setService(Service service)  
{  
    // Save the reference to the passed-in service inside this client.  
    this.service = service;  
}  
  
// Service setter interface.  
public interface ServiceSetter  
{  
    public void setService(Service service);  
}  
  
// Client class  
public class Client implements ServiceSetter  
{  
    // Internal reference to the service used by this client.  
    private Service service;  
  
    // Set the service that this client is to use.  
    @Override  
    public void setService(Service service)  
    {  
        this.service = service;  
    }  
}
```

## 7) Software Architektur Pattern

- Pattern spielen in der Softwarearchitektur eine große Rolle
- Viele Probleme im Software Design haben bereits erprobte Lösungen
- Erfahrene Architekten erkennen Einsatzmöglichkeiten von verfügbaren Patterns
- Wir betrachten
  - a) Was ist ein Software Architektur Pattern
  - b) Verschiedene Architekturen und Pattern

## 7a) Was ist ein Software Architektur Pattern

- Ein Software Architektur Pattern ist eine Lösung zu einem wiederkehrenden Problem, das in einem bestimmten Kontext, gut verstanden ist
- Jedes Pattern besteht aus einem Kontext, einem Problem und einer Lösung
- Das Problem könnte eine Herausforderung, eine vorteilhafte Gelegenheit oder die Antwort auf die Anforderungen von Qualitätsattributen sein
- Pattern kodieren Wissen und Erfahrung in Lösungen, die wir wiederverwenden können
- Pattern vereinfachen Design und Architektur durch die Nutzung erprobter Lösungen
- Architekturen sind übergreifender, Designpattern spezieller, detaillierter
- Jedes Pattern hat Ausprägungen, Stärken und Schwächen
- Pattern helfen die Komplexität zu verringern
- Pattern können kombiniert werden
- Pattern können übermäßig benutzt werden
- Architektur Stile und Pattern werden manchmal getrennt aufgelistet und unterschieden (hier nicht)
- Architekten sind mit Pattern vertraut

## 7b) Verschiedene Architekturen und Pattern

- Architekturen oder Architekturmuster
  - ▶ Architectural patterns
  - ▶ Hoher Abstraktionsgrad
  - ▶ Architektur eines Systems
    - Layers, verteilte Systeme, MVC
- Entwurfsmuster oder Pattern
  - ▶ Design patterns (GoF = Gang of Four / Gama, Helm, Johnson, Vlissides)
  - ▶ Niedrigerer Anstraktionsgrad
  - ▶ Problemstellung innerhalb eines Subsystems / Konstruktionsprinzipien im Kleinen (Mikroarchitekturen)
  - ▶ Beeinflusst i.a. nicht die Architektur eines Systems
- Idiome
  - ▶ Muster in einer bestimmten Programmiersprache

## 7b) Verschiedene Architekturen und Pattern

### ■ Schema zur Darstellung

- ▶ Name
- ▶ Problembeschreibung
- ▶ Lösungsbeschreibung
  - Teilnehmer (Rollenbeschreibungen)
  - Diagramme (Klassen, Beziehungen)
  - Verhalten (Sequenzdiagramm)
  - Beispiel (Code)
- ▶ Bewertung
  - Vorteile
  - Nachteile
- ▶ Einsatzgebiete
- ▶ Ähnliche Muster (Gemeinsamkeiten und Unterschiede)

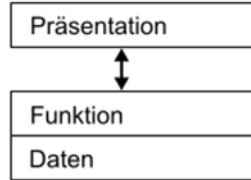
# 7b) Verschiedene Architekturen und Pattern

## ■ Architekturmuster

### ▶ Schichtenarchitektur

#### ● Horizontale Schichten

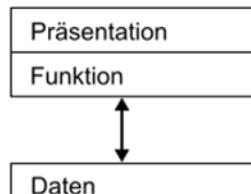
- unterschiedliche Aufteilung



THIN-CLIENT, AKTIVER SERVER

- + zentrale Administration & Wartung
- + Kostenersparnis
- + Sicherheit (nur 1 Server muss geschützt werden)
- + Flexibilität

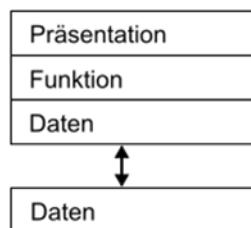
- hohe Belastung für Server



DATEN-SERVER

- Server liefert nur benötigte Daten
- + jeder Client kann selber rechnen → höhere Performance
- + zentrale Datenhaltung bleibt bestehen → Sicherheit

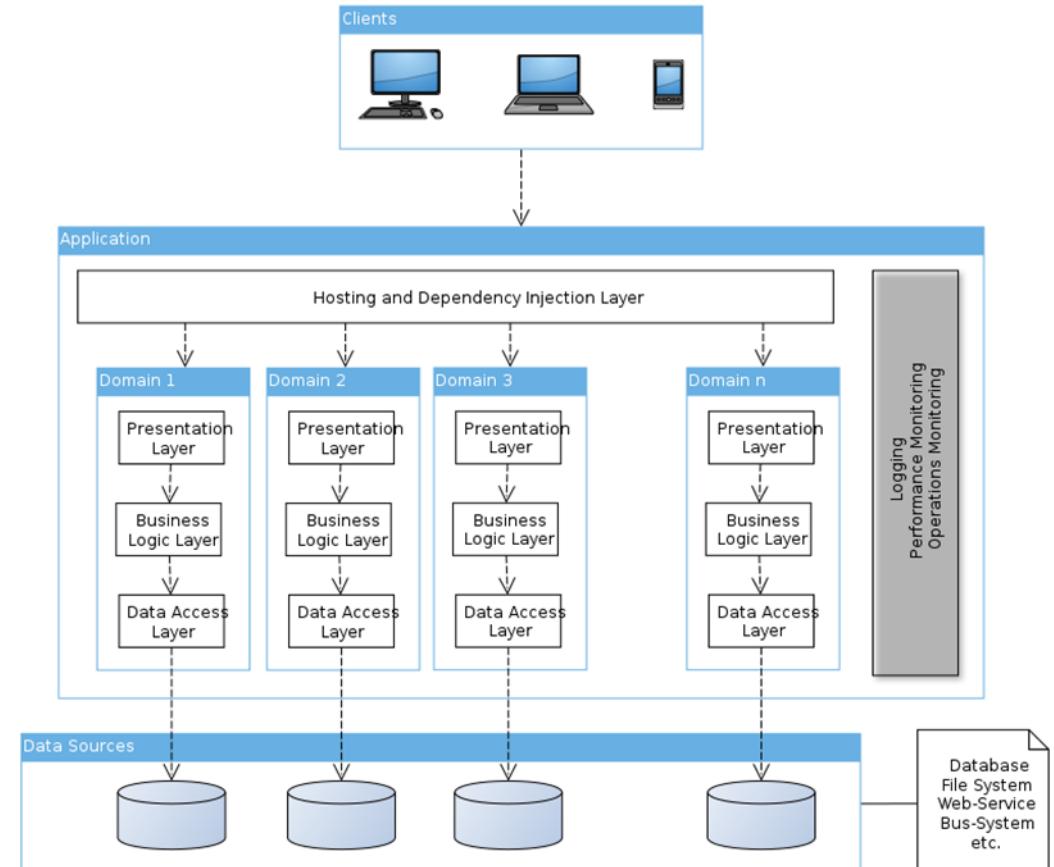
- hoher Installationsaufwand, da jeder Client alle Applikationen braucht



FAT-CLIENT

- + Entlastung Server & Datenleitung durch Plug-Ins
- + Weiterarbeiten möglich, wenn Kommunikation zum Server gestört

- lange Dauer, bis alle Clients Updates haben  
- hohe Wartungskosten



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Schichtenarchitektur

- Es gibt offenen und geschlossene Schichten
  - Geschlossene Layerarchitekturen verhindern das Umgehen (Bypass) von Schichten
- Tiers und Layers
  - Layers sind die logische, Tiers die physikalische Aufteilung in Schichten
- Vorteile
  - Separation of concern wird direkt unterstützt oder lässt sich einfach implementieren
  - Abhängigkeiten zwischen Layern lassen sich reduzieren
  - Wiederverwendbarkeit wird unterstützt
  - Skalierbarkeit wird unterstützt (jeder Layer auf eigener Hardware)
  - Security kann höher sein

## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Schichtenarchitektur
    - Nachteile
      - Änderungen (ein neues Feld) an einem Layer verursachen oft Änderungen in anderen Layern
      - Die Interfaces zwischen den Layern verursachen mehr Code und damit mehr Komplexität
      - Oft landet Code im falschen Layer (Business Logik im Presentation Layer oder Datenzugriffslogik im Business Layer)
      - Stark verteilte Anwendungen mit hoher Anforderung an Performance und Verfügbarkeit bringen solche Architekturen an ihre Grenzen
        - Bottlenecks
        - Laufzeiten durch die Layer
        - Überlast von Ressourcen

## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Event-driven Architektur

- Ist eine verteiltes, asynchrones Architektur Pattern, das Anwendungen und Komponenten durch das Erzeugen und Verwalten von Events verbindet
- Ist eine lose gekoppelte Architektur
  - Der Erzeuger eines Events hat keine Ahnung vom Empfänger/Konsumenten des Events
- Kann eine sehr komplexe Architektur sein (mit Mediatoren und Brokern)
- Kann sehr mächtig sein (wenn richtig geplant und implementiert)

## 7b) Verschiedene Architekturen und Pattern

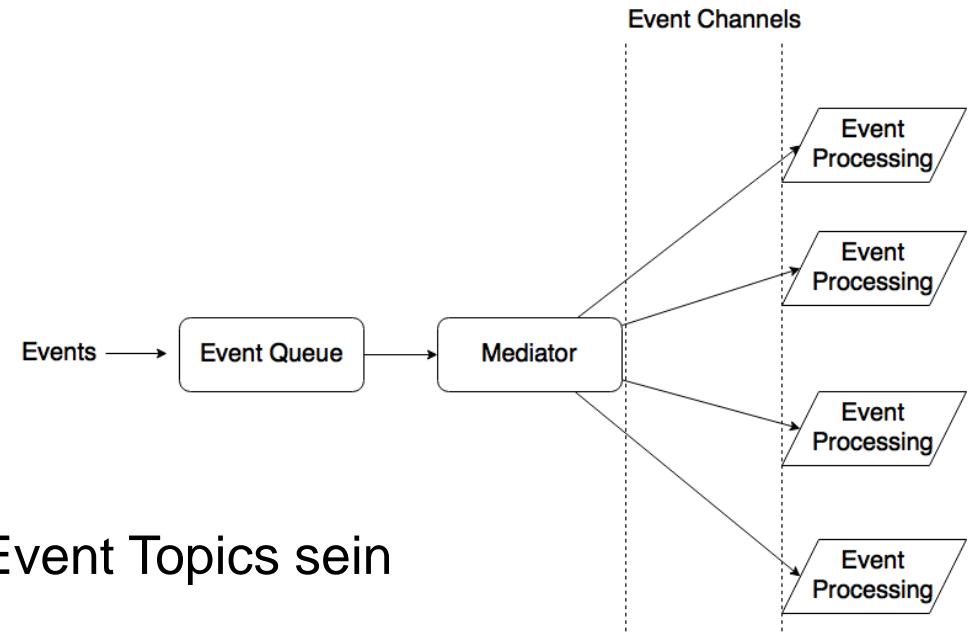
- Architekturmuster
  - ▶ Event-driven Architektur
    - Die beiden wichtigsten Event Topologien benutzen
      - Event channels (= Ströme von Event Nachrichten) zwischen Event Prozessoren (Verarbeitern)
      - Event channels sind typischerweise als Message Queues implementiert, die
        - Point-to-point Verbindungen (genau ein Empfänger)
        - Message Topics (mehrere Empfänger)
        - Publish-subscribe Pattern (Broadcast an Interessierte, Empfänger melden Interesse an) implementieren

## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Event-driven Architektur

- Die Mediator Topologie
  - Alle Event gehen durch einen sog. Mediator
  - Der Mediator orchestriert die Events, generiert einen oder mehrere Events zur folgenden Eventverarbeitung
  - Event Channels können Event Queues oder Event Topics sein
  - Event Prozessoren warten an einem Channel auf einen Event zur Verarbeitung
  - Implementierungen dieser Art findet man oft in Business Prozess oder Workflow Umgebungen

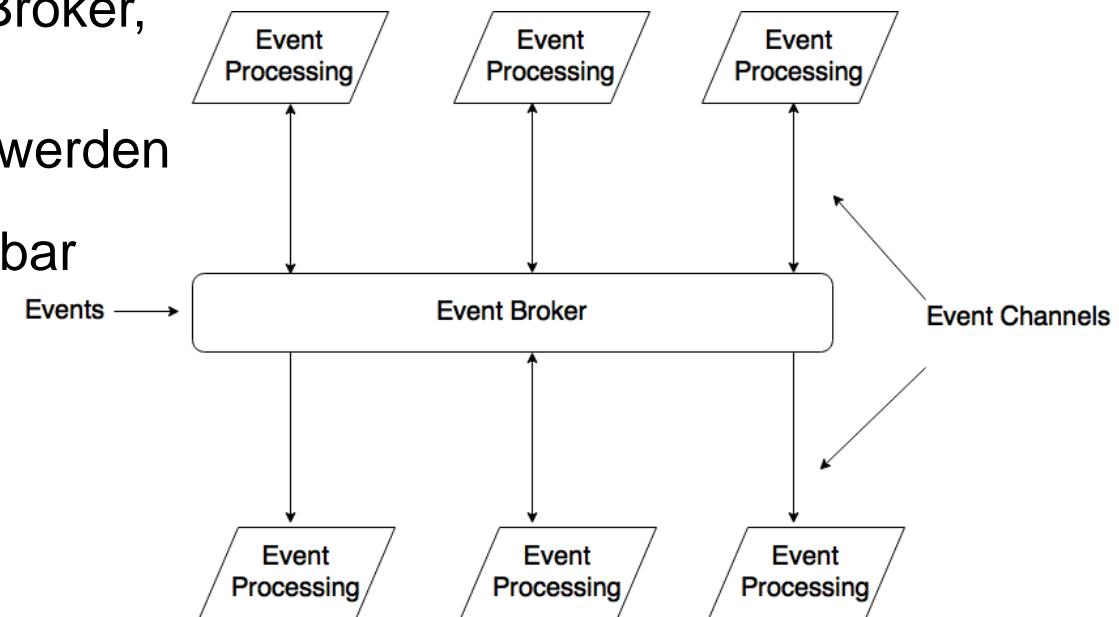


## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Event-driven Architektur

- Die Broker Topologie
  - Ohne Event Queue
  - Event Prozessoren holen sich Events vom Broker, verarbeiten sie und erzeugen nach Bedarf neue Events, die an den Broker übergeben werden
  - Bei einfachen Event Flüssen besser einsetzbar



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

- ▶ Event-driven Architektur
  - Event Verarbeitung
    - Simple Event processing
      - Events werden zur Verarbeitung sofort weitergeleitet
      - Kann für Real-Time Workflows verwendet werden
    - Event Stream processing
      - Analysiert Event Streams und steuert danach die weitere Verarbeitung
      - Beispiel : Kauf von Aktien, die unter eine definierte Schwelle gefallen sind
    - Complex Event processing
      - Analysiert Events Streams auf wiederkehrende Muster
      - Läuft i.a. über längere Zeiträume
      - Beispiel : Kreditlimit

## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

- ▶ Event-driven Architektur
  - Event Verarbeitung
    - Simple Event processing
      - Events werden zur Verarbeitung sofort weitergeleitet
      - Kann für Real-Time Workflows verwendet werden
    - Event Stream processing
      - Analysiert Event Streams und steuert danach die weitere Verarbeitung
      - Beispiel : Kauf von Aktien, die unter eine definierte Schwelle gefallen sind
    - Complex Event processing
      - Analysiert Events Streams auf wiederkehrende Muster
      - Läuft i.a. über längere Zeiträume
      - Beispiel : Kreditlimit

## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Event-driven Architektur
    - Typen von Event-driven Funktonalität
      - Event notification
        - Eine Nachricht (Event) wird generiert/gesendet, wenn ein Ereignis auftritt
        - Mediator und Broker verarbeiten/steuern i.a. genau solche Events
        - Es existiert eine lose Kopplung zwischen Sender und Empfänger/Konsument
        - Event Prozessoren haben eine genau definierte Aufgabe
        - Die lose Kopplung macht die Event Verfolgung (Fehlerverfolgung) schwieriger

## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Event-driven Architektur
    - Typen von Event-driven Funktionalität
      - Event-carried state transfer
        - Eine Variante von Event notification
        - Nach dem Empfang eines Events könnte der Empfänger weitere Informationen benötigen/anfordern, um den Event verarbeiten zu können
        - Statusinformationen erhöhen die Nutzbarkeit von Events und werden deshalb mitgeliefert
        - es werden mehr Daten übermittelt und Datenkonsistenz gerät in Gefahr

## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Event-driven Architektur

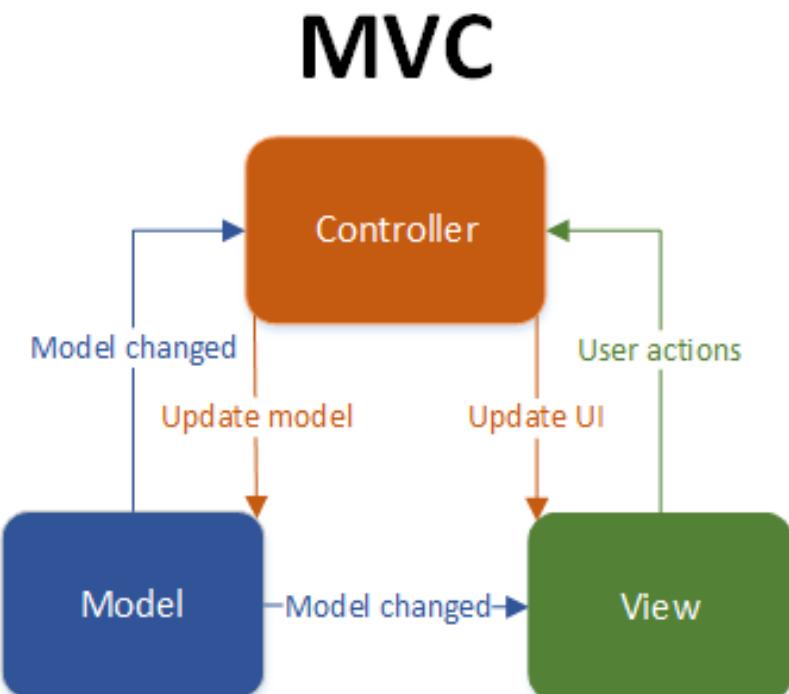
- Typen von Event-driven Funktionalität
  - Event sourcing

- Events werden in einem Event Store persistiert, um Änderungen (z.B. am Status) nachvollziehen zu können
- Events können aus dem Store “gespielt” werden
- Roll back Szenarien sind möglich
- ähnlich einem Transaction Log bei Datenbanken
- bringt zusätzliche Komplexität ins Gesamtsystem
- wenn sich Event Processing ändert, müssen evtl. ältere Events angepasst werden
- es macht Sinn, externe Systeme über Events anzubinden  
(um sie speichern zu können)

## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

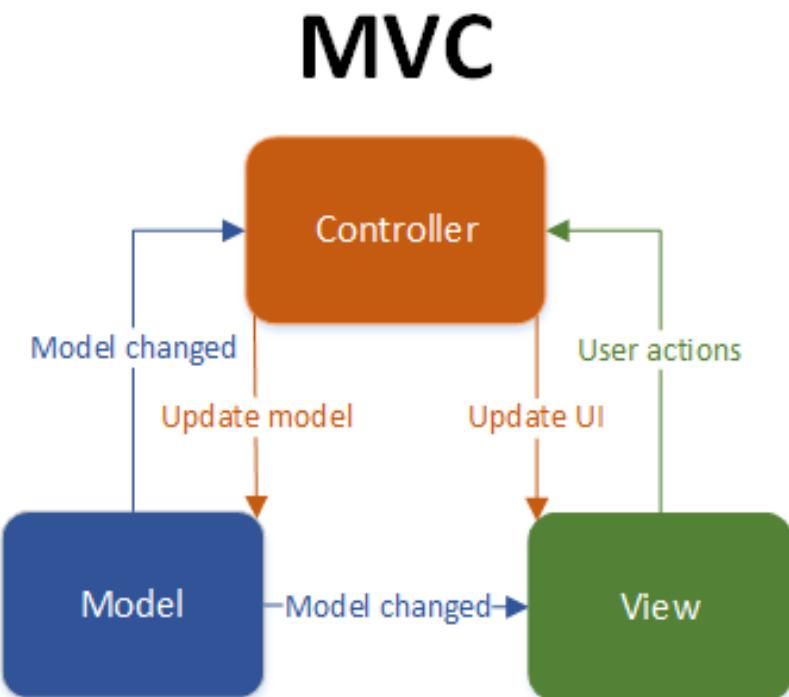
- ▶ Das Model-View-Controller Pattern (MVC)
  - weit verbreitet (IOS Apps)
  - trennt Verantwortlichkeiten der Schnittstellen zum Nutzer
  - Model
    - ist verantwortlich für Daten und Status
    - auch für die Verarbeitung von Daten von und zu einer Datenbank
    - unabhängig von Controller und View
    - empfängt Anforderungen vom Controller
    - hat eine eher passive Rolle



## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Das Model-View-Controller Pattern (MVC)

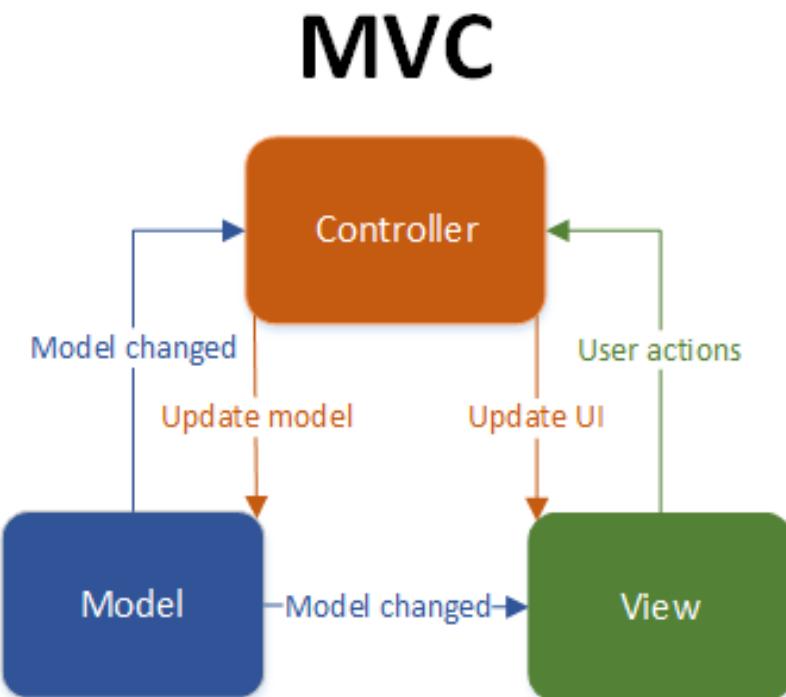
- View
  - ist verantwortlich für die Anzeige der Anwendung
  - ist für den Benutzer sichtbar
  - zeigt Daten an und nimmt Eingaben vom Benutzer auf
  - Gibt Eingaben an den Controller weiter



## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Das Model-View-Controller Pattern (MVC)

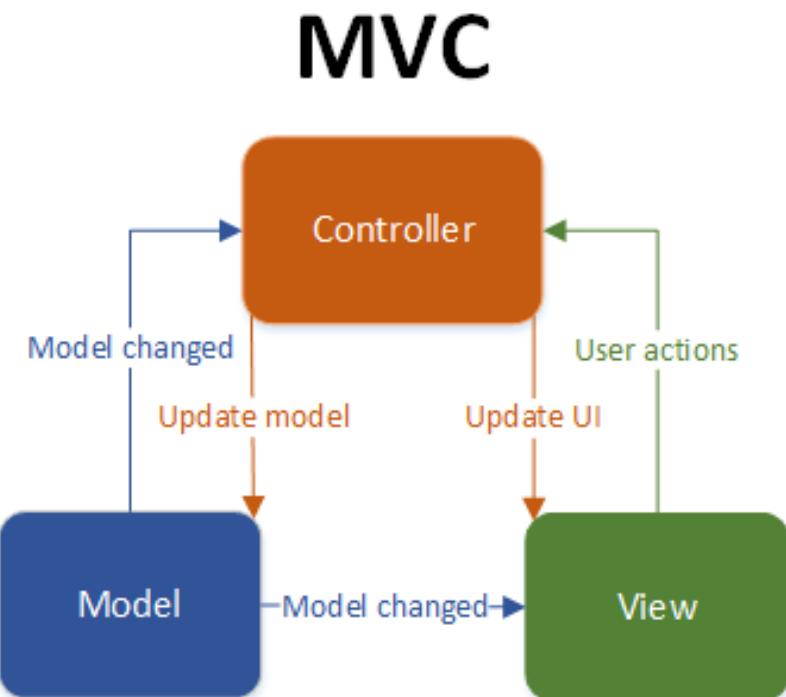
- Controller
  - ist verantwortlich für die Logik der Anwendung
  - ist Mittler zwischen Daten (Model) und Anzeige (View)
  - gibt Anfragen/Updates an das Model und empfängt von dort Daten
  - wählt geeignete Views zur Anzeige aus



## 7b) Verschiedene Architekturen und Pattern

- Architekturmuster
  - ▶ Das Model-View-Controller Pattern (MVC)

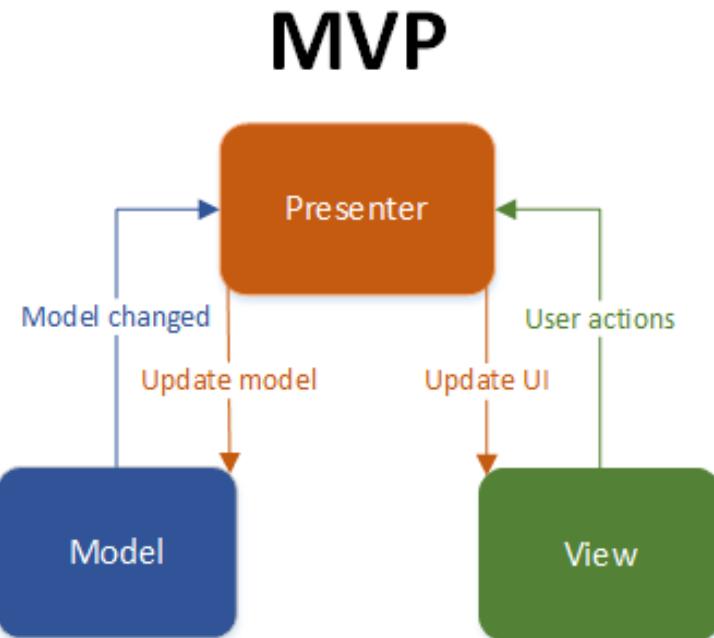
- Vorteile
  - direkte Unterstützung von “Separation of concerns”
  - einfacheres Testen der Einzelteile
  - keine ganz lose Kopplung erreichbar (neues Feld)
  - dient hauptsächlich zum Abtrennen von Benutzeroberflächen (dynam. Ändern zur Laufzeit)



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

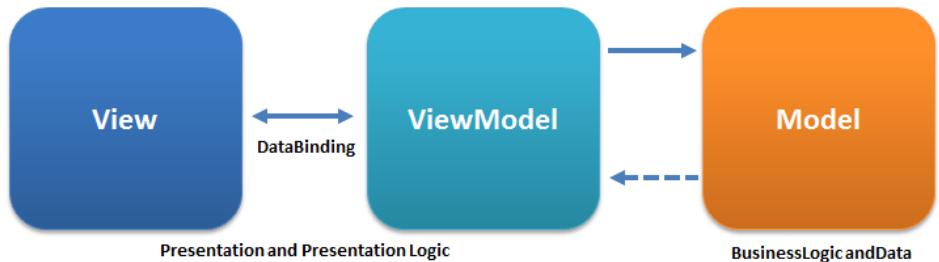
- ▶ Das Model-View-Presenter Pattern (MVP)
  - eine Variante des MVC Pattern
  - der Presenter ersetzt den Controller
  - jede View hat ein spezielles Interface (view interface), der Presenter ist an die View interfaces gekoppelt und steuert diese
  - MVP hat eine stärkere lose Kopplung als MVC (speziell zwischen View und Model)
  - im Gegensatz zum MVC verwaltet ein Presenter eine View
  - es kann mehrere Presenter geben



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

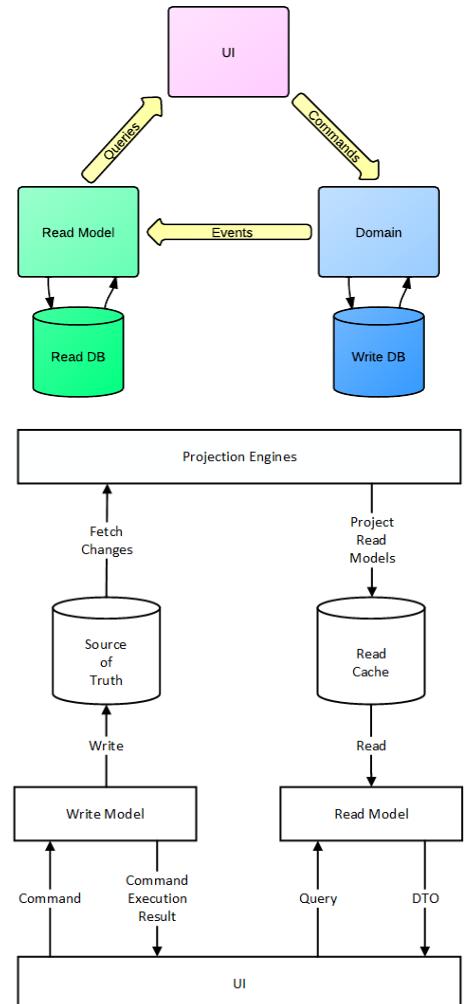
- ▶ Das Model-View-ViewModel Pattern
  - eine Mischung von MVC und MVP
  - Windows Presentation Foundation nutzt dieses Modell
  - Model besitzt die Logik und die Anbindung der Daten
  - View ist das für den Anwender sichtbare Interface
  - ViewModel spielt die Rolle des Controllers oder Presenters
  - ViewModel besitzt die Logik der Navigation
  - klares Schichtenmodell, gut geeignet für Rich Clients



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

- ▶ Das Command Query Responsibility Segregation Pattern (CQRS)
  - Schreibkanal getrennt vom Lesekanal
  - unterschiedliche Workloads zwischen Lese- und Schreiboperationen können ideal abgebildet werden
  - Sperren von Datensätzen wird vermieden
  - Lesedaten können für die Anzeige optimiert abgelegt sein (Performance)
  - Optimierte und unterschiedliche Datenpools können benutzt werden
  - Updates der Datenpools über Events (z.B. Event sourcing)
  - Manche NoSQL Datenbanken unterstützen CQRS out-of-the-box
  - Das Trennen von Schreib- und Lesemodell kann die Security erhöhen
  - CQRS erhöht die Komplexität des Gesamtsystems
  - Lesepool “hinkt” hinterher



## 7b) Verschiedene Architekturen und Pattern

### ■ Architekturmuster

#### ▶ Service Oriented Architecture (SOA)

- lose gekoppelte voneinander unabhängige Services, die zusammen die Anwendung bilden
- teilt ein Software System in kleinere Einheiten auf, die jeweils eine andere Aufgabe erfüllen
- kann auch Business Logik in kleinere Einheiten aufteilen
- ein Service kann aus mehreren anderen Services bestehen oder sie nutzen
- verbindet die Logik mit der Technik
- unterstützt die Nutzung verschiedener Serviceanbieter
- unterstützt stark agile Entwicklungsmethoden
- benötigt gute Architekten
- bedingt gute Architektur von Services und eine vollständige Beschreibung
- besitzt oft sog. Service Register als Verzeichnis der vorhandenen Services

## 8) Moderne Architekturen

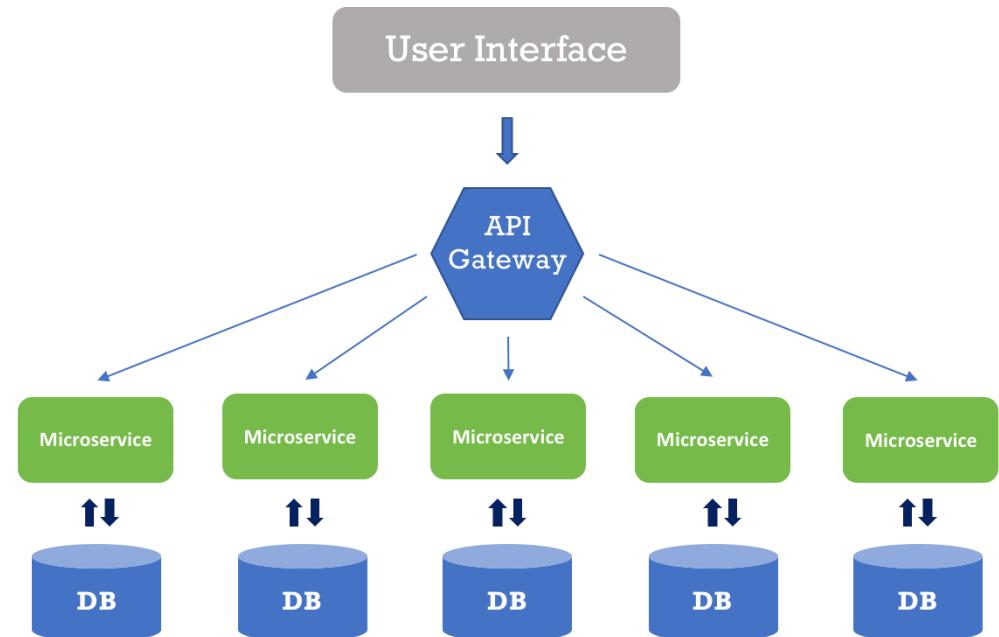
- Moderne Anwendungen in der Cloud haben ganz andere Erwartungen und Anforderungen an Architekturen im Vergleich zu früher
- Um diesem zu genügen, wurden neue Software Architektur Patterns entwickelt
- Wir betrachten
  - a) Monolithic Architecture
  - b) Microservice Architecture
  - c) Serverless Architecture
  - d) Cloud-native Architecture
  - e) Microsoft Cloud Architekturstile

## 8a) Monolithic Architecture

- einzelne sich selbst beinhaltende Anwendung
- sehr weit verbreitet
- hohe interne Koppelung der Module
- keine Trennung von Logik und Anzeige oder Logik und Datenhaltung
- Vorteile
  - ▶ bessere Performance (speziell bei kleinen Anwendungen)
  - ▶ einfacher zu verteilen und installieren
  - ▶ einfacher zu testen und debuggen
- Nachteile
  - ▶ skaliert schlechter (speziell in verteilten Umgebungen)
  - ▶ niedrige Wartbarkeit
  - ▶ wenig/schlecht erweiterbar
  - ▶ unterstützt agile Entwicklungsmethoden und Teamstrukturen schlecht

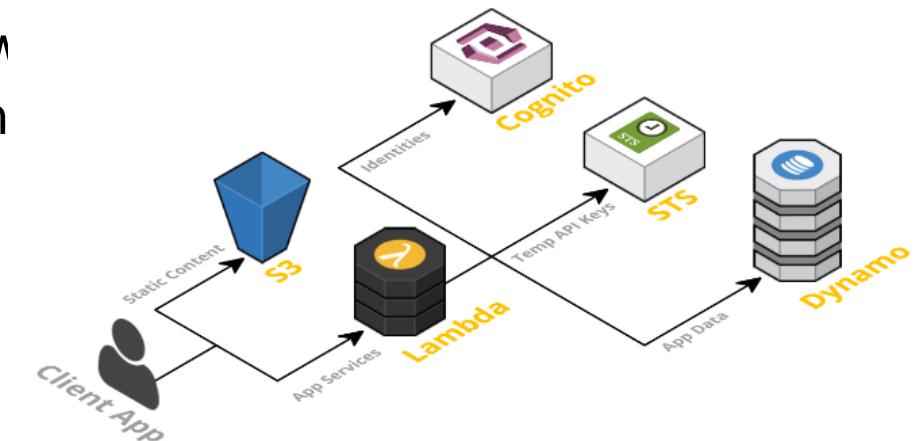
## 8b) Microservice Architecture

- kleine, autonome, unabhängig versionierte Services
- genau definiertes Interface zur Kommunikation
- eingehende Requests gehen über ein sog. API Gateway
- eine Variante von SOA (SOA done right)
- Charakteristiken
  - ▶ kleine, fokussierte Services
  - ▶ genau spezifizierte Interfaces
  - ▶ autonom und unabhängig ausrollbar
  - ▶ unabhängiger Datenspeicher
  - ▶ einfache Kommunikationsprotokolle  
(JSON, Advanced Message Queueing Protocol (AMQP))
  - ▶ sehr gute Isolation
  - ▶ Bulkhead fähig
  - ▶ Recovery fähig
  - ▶ Polyglot



## 8c) Serverless Architecture

- Services ohne Berücksichtigung von Hardware (Server)
- Code wird auf einer Plattform dann ausgeführt, wenn benötigt
- ist oft in einem Container implementiert, der zur Laufzeit hochgefahren und wieder beendet wird
- besteht oft aus statischen (Speicher) und dynamischen (Funktionen) Anteilen
- benutzt FaaS (function as a service) oder BaaS (Backend as a service) Modelle
- Jede Funktion folgt dem Single Responsibility Principle und dient genau einem definierten Zweck
- Funktionen sollten idempotent sein
- Funktionen können synchron oder asynchron ablaufen
- ein wichtiger Teil einer Serverless Architektur ist das API Gateway
- Beispiele Amazon Lambda, Azure Services, Google Cloud Functions



## 8c) Serverless Architecture

### ■ Vorteile

#### ▶ Kosten

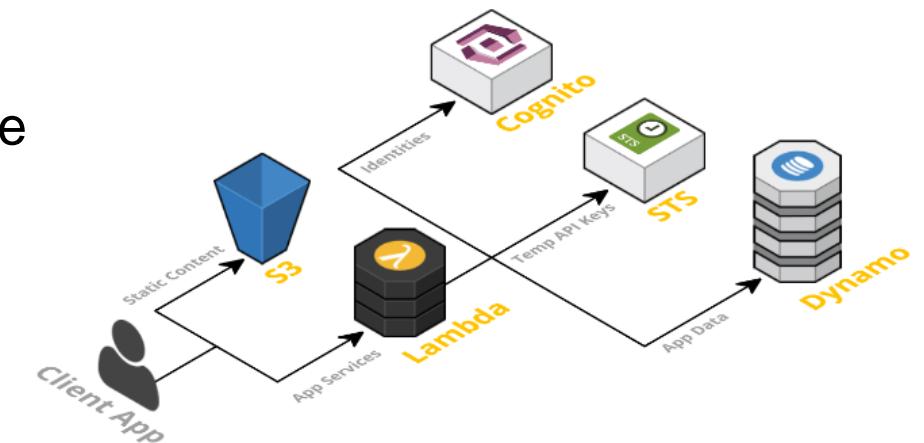
- Code läuft nur, wenn benötigt
- abgerechnet werden benutzte Ressourcen (pay per usage)
- keine Hardwarekosten (Bestellung, Installation, Betrieb, Austausch)
- keine Betriebsmannschaften

#### ▶ Skalierbar und flexible

- es steht immer genügend Kapazität zur Verfügung
- keine Verschwendungen von Kapazitäten, keine Freistände

#### ▶ Fokussiert

- auf die Lösung des Problems
- keine Infrastruktur notwendig



## 8c) Serverless Architecture

### ■ Vorteile

- ▶ Polyglot
  - best-of-breed Programmiersprachen einsetzbar

### ▶ Cloud Plattformen beliebig mischbar

### ■ Nachteile

### ▶ aufwändigeres Monitoring und Debugging

- verteilt, Event getrieben, stateless

### ▶ wenige verfügbare Pattern

### ▶ verfügbare Pattern noch nicht lange im Einsatz

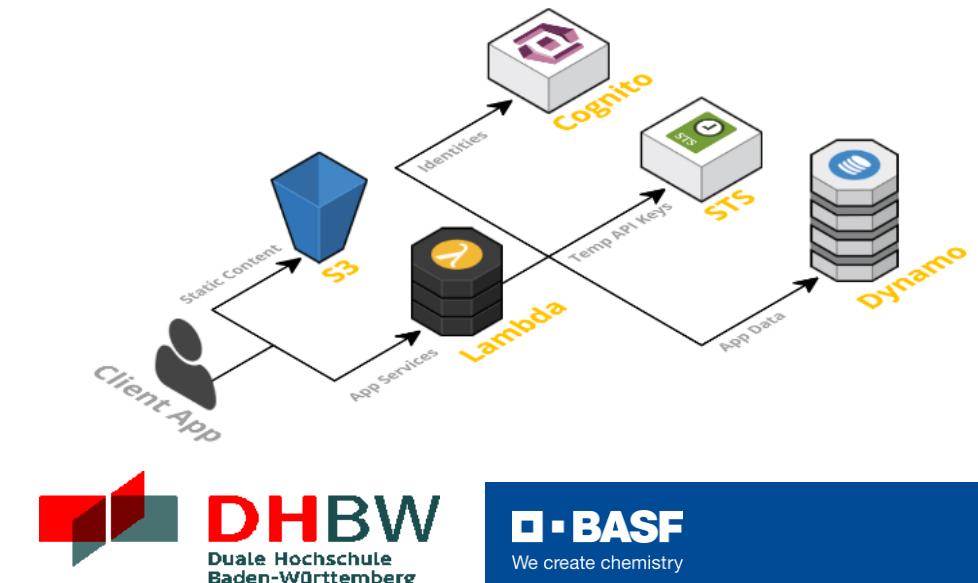
- geringerer Reifegrad

### ▶ Maintenance (Performance & Security)

### ▶ Vendor lock-in

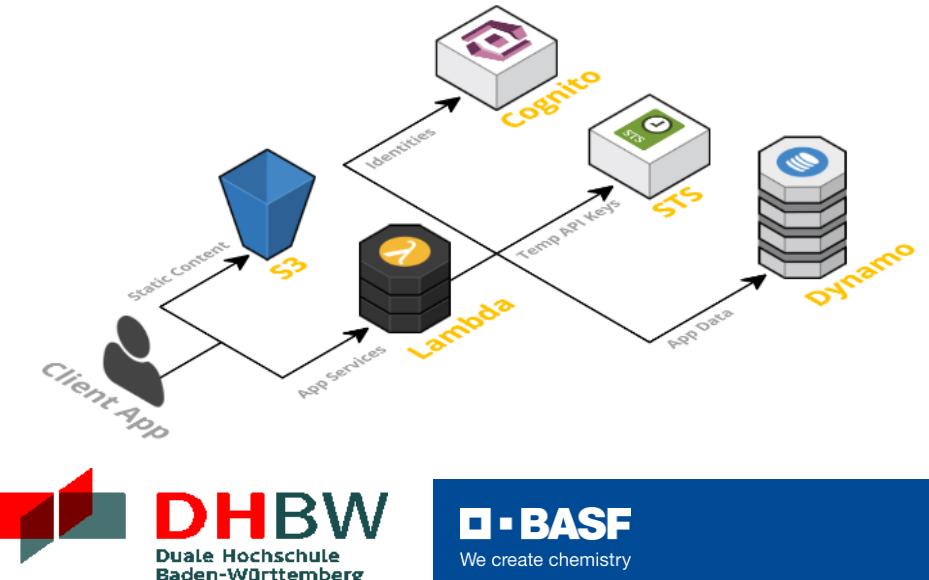
### ▶ Komplexität

### ▶ Optimierungsmöglichkeiten eingeschränkt



## 8c) Serverless Architecture

- Hybride Ansätze möglich
  - ▶ teilweise on-premise, teilweise in der Cloud
- Aufrufmethoden von Funktionen
  - ▶ Synchron
  - ▶ asynchron
  - ▶ Message stream
  - ▶ Batch job



## 8d) Cloud-native Architecture

- Was bedeutet Cloud Native ?
  - ▶ Cloud Native (cloud-native) beinhaltet folgende Konzepte
    - „Wegwerf“-Infrastruktur als Triebfeder
    - Einsatz von funktional eingeschränkten, isolierten Komponenten
    - Globale Skalierung
    - Eine Architektur, die sich dauernd verändert
    - Wirksamer Einsatz von wertbringenden Cloud Services
    - Vielsprachigkeit (Programmiersprachen)
    - Unabhängige, voll selbstständige Teamstrukturen
    - Kulturwechsel in Organisationen

## 8d) Cloud-native Architecture

- „Wegwerf“-Infrastruktur als Triebfeder
  - ▶ Erzeugen und Vernichten von Ressourcen zu jeder Zeit
  - ▶ Automatisierung
  - ▶ Geschwindigkeit, Schnelligkeit
  - ▶ Infrastructure as Code
  - ▶ Sicherheit
  - ▶ Optimierung von Teamgröße und Effizienz

## 8d) Cloud-native Architecture

- Die Cloud ist die Datenbank
- Reactive Architectur ([www.reactivemanifesto.org](http://www.reactivemanifesto.org))
  - ▶ Responsiv, Resilient, Elastic, Message driven
- Die Datenbank nach außen stülpen (inside out)
  - ▶ Weg vom Monolithen zur Replikation
- Bulkheads
- Event streaming
- Vielschichtige Persistenz
- Cloud Native Datenbanken
- Cloud Native Patterns
- Bounded Isolated components



## 8d) Cloud-native Architecture

- Cloud Native Patterns
  - ▶ Foundation Patterns
  - ▶ Boundary patterns
  - ▶ Control patterns

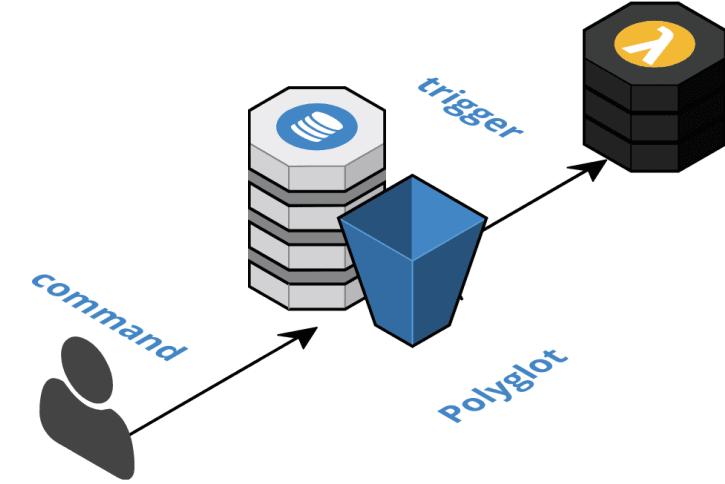
## 8d) Cloud-native Architecture

- Cloud Native Foundation Patterns
  - ▶ Cloud Native Database per Component
  - ▶ Event Streaming
  - ▶ Event Sourcing
  - ▶ Data Lake
  - ▶ Stream Circuit Breaker
  - ▶ Trilateral API

## 8d) Cloud-native Architecture

### ■ Cloud Native Database per Component

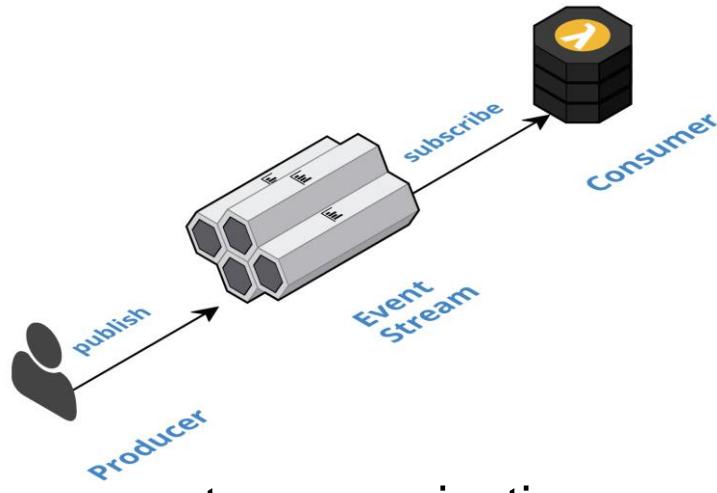
- ▶ Leverage one or more fully managed cloud-native databases that are not shared across components and react to emitted events to trigger intra-component processing logic.
- ▶ Leverage your cloud provider's fully managed cloud-native databases services. Employ multiple database types within a component, as needed, to match the component's workload characteristics.
- ▶ Choose database types like blob store, document store
- ▶ Each database is dedicated to a specific component and not shared across components



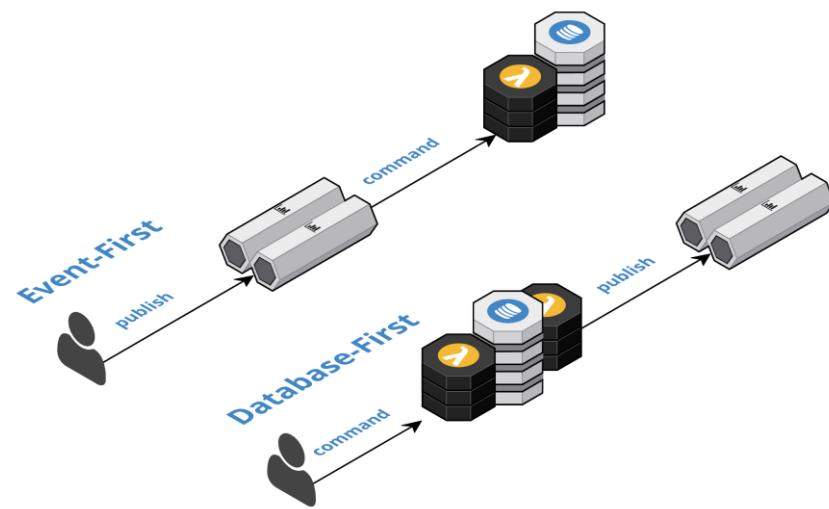
## 8d) Cloud-native Architecture

### ■ Event streaming

- ▶ Leverage a fully managed streaming service to implement all inter-component communication asynchronously, whereby upstream components delegate processing to downstream components by publishing domain events that are consumed downstream.
- ▶ Consumers will subscribe to one or more streams as appropriate.
- ▶ Define a standard event envelope format that all events extend to ensure event handling in a consistent manner.



## 8d) Cloud-native Architecture



- Event sourcing
  - ▶ Communicate and persist the change in state of domain entities as a series of atomically produced immutable domain events, using Event-First or Database-First techniques, to drive asynchronous inter-component communication and facilitate event processing logic.
  - ▶ Strictly adhere to a policy that a command must perform one and only one atomic write operation against a single resource, either a stream or a cloud-native database, and then rely on the asynchronous mechanism of the resource to chain commands together.
  - ▶ Event-First : wrapped data in a domain event published to one single stream
  - ▶ Database-First : data written to a single cloud-native database, that triggers a domain event published to a single stream

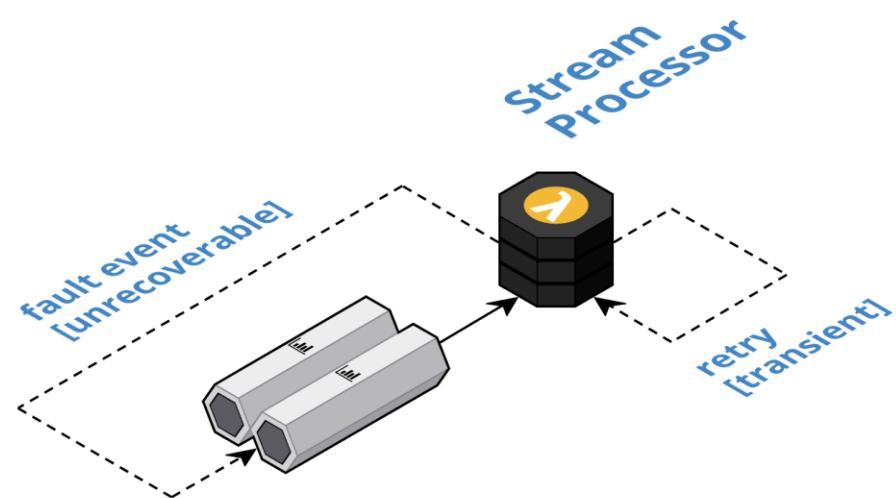
## 8d) Cloud-native Architecture



### ■ Data Lake

- ▶ Collect, store, and index all events in their raw format in perpetuity with complete fidelity and high durability to support auditing, replay, and analytics.
- ▶ Implement one or more consumers that in aggregate consume all events from all streams and store the unaltered, raw events in highly durable blob storage.
- ▶ Optionally, implement consumers to store events in a search engine, such as Elasticsearch, for indexing and time series analytics.

## 8d) Cloud-native Architecture



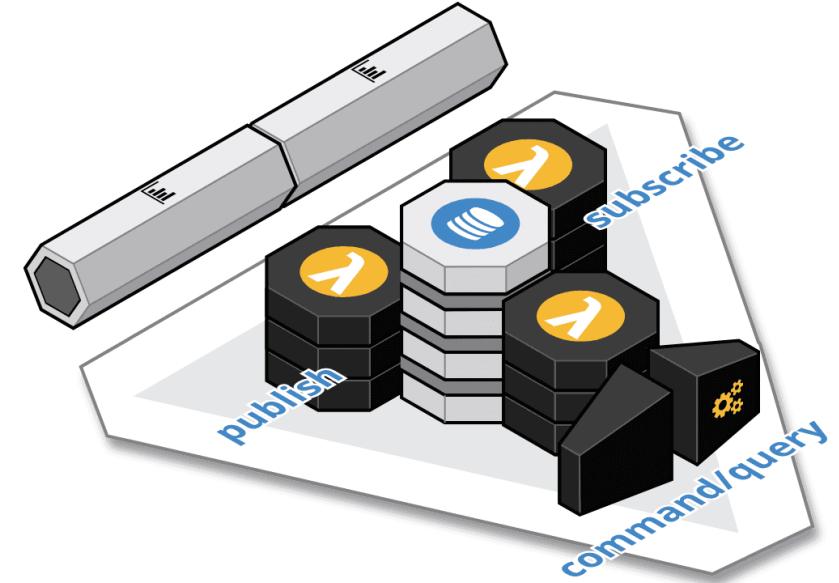
### ■ Stream Circuit Breaker

- ▶ Control the flow of events in stream processors so that failures do not inappropriately disrupt throughput, by delegating the handling of unrecoverable errors through fault events.
- ▶ Delegate events with unrecoverable errors to another component for handling so that they do not block legitimate events from processing.
- ▶ Publish these errors as fault events, along with the effected events and the stream processor information, so that the events can be resubmitted when appropriate.
- ▶ Monitor and alert on these fault events so that the team can react in a timely manner.
- ▶ Provide the necessary utilities to resubmit the effected events from the data lake to the component that emitted the fault event.

## 8d) Cloud-native Architecture

### ■ Trilateral API

- ▶ Publish multiple interfaces for each component :
  - A synchronous API for processing commands and queries
  - An asynchronous API for publishing events as the state of the components changes
  - An asynchronous API for consuming the events emitted by other components



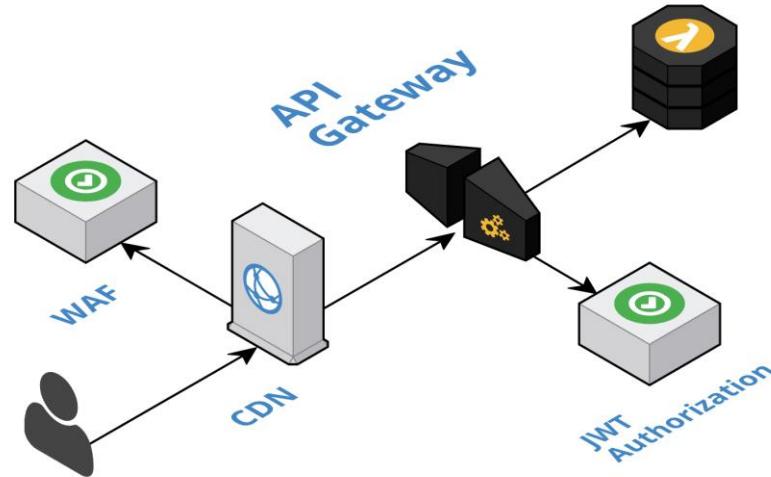
## 8d) Cloud-native Architecture

- Cloud Native Boundary Patterns
  - ▶ API Gateway
  - ▶ Command Query Responsibility Segregation (CQRS)
  - ▶ Offline-First Database
  - ▶ Backend For Frontend
  - ▶ External Service Gateway

## 8d) Cloud-native Architecture

### ■ API Gateway

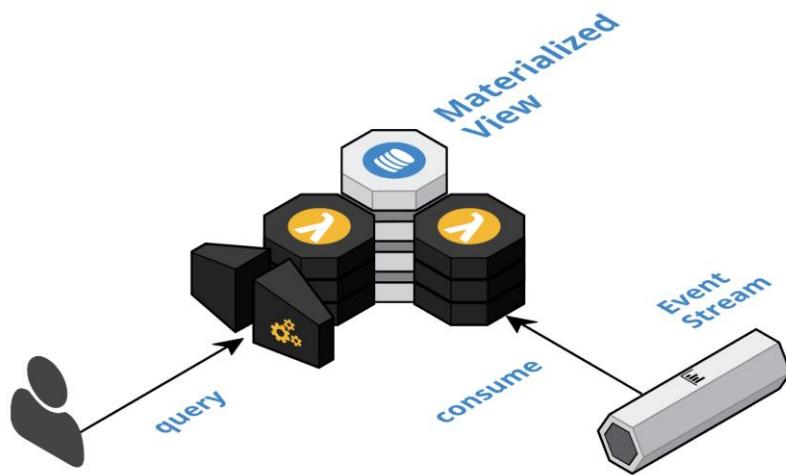
- ▶ Leverage a fully managed API gateway to create a barrier at the boundaries of a cloud-native system by pushing cross-cutting concerns, such as security and caching, to the edge of the cloud where some load is absorbed before entering the interior of the system.
  - CDN = Content Delivery Network
  - WAF = Web Application Firewall
- ▶ The perimeter acts as a bulkhead to protect the internals of the system from attacks at the boundaries of the system.



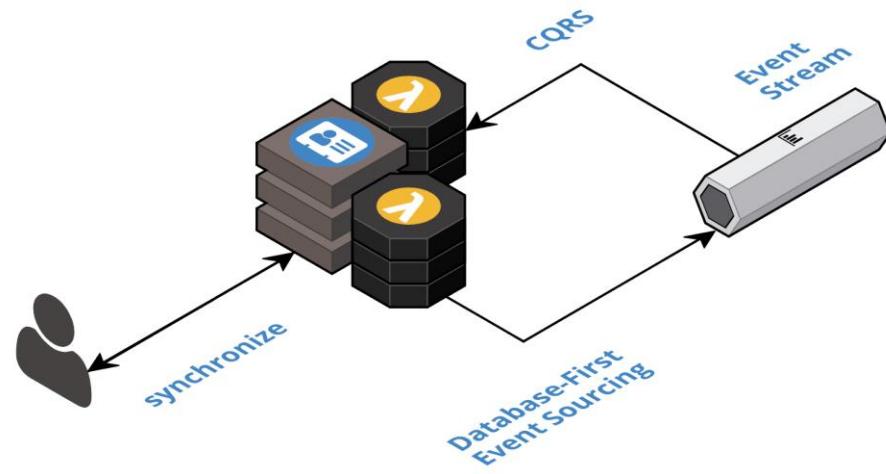
## 8d) Cloud-native Architecture

### ■ Command Query Responsibility Segregation (CQRS)

- ▶ Consume state change events from upstream components and maintain materialized views that support queries used within a component.
- ▶ The benefit is that it provides proper bulkheads between components to make downstream components resilient to failures in upstream components.
- ▶ Upstream unavailabilities will have zero impact on downstream components. Queries will continue to return results from the materialized views.
- ▶ A materialized view acts as a local cache that is always hot and always up to date with the last known values. It is a local resource that is owned by the component and thus no unnecessary layers must be traversed to retrieve the data.



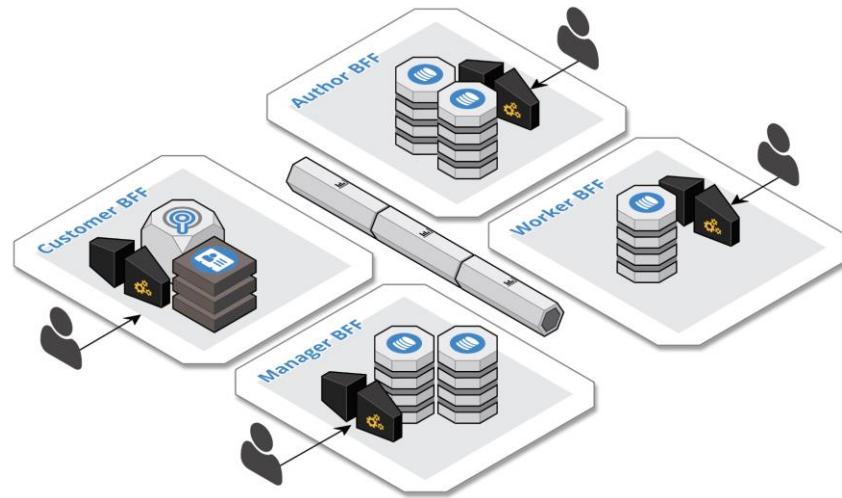
## 8d) Cloud-native Architecture



### ■ Offline-First Database

- ▶ Persist user data in local storage and synchronize with the cloud when connected so that client-side changes are published as events and cloud-side changes are retrieved from materialized views.
- ▶ The primary benefit of this solution is that it provides extremely high availability and responsiveness for important user interaction. Users perceive the system as responsive and available even when connectivity is intermittent.
- ▶ When connectivity is restored, the data is synchronized with the cloud storage and ultimately across devices, creating a resilient solution.

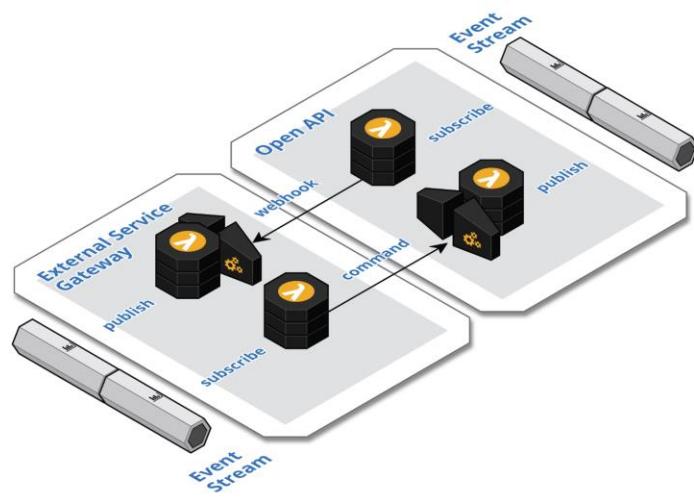
## 8d) Cloud-native Architecture



### ■ Backend For Frontend

- ▶ Create dedicated and self-sufficient backend components to support the features of user focused, frontend applications.
- ▶ Architecture the user experience as a set of many independant experiences, each supporting a distinct user base and a distinct and cohesive set of features.
- ▶ The benefit of this solution is that it heeds the Conway's Law that „organizations are constrained to produce application designs which are copies of their communication structures“.
- ▶ The user experiences are decoupled, each managed by a single development team.

## 8d) Cloud-native Architecture



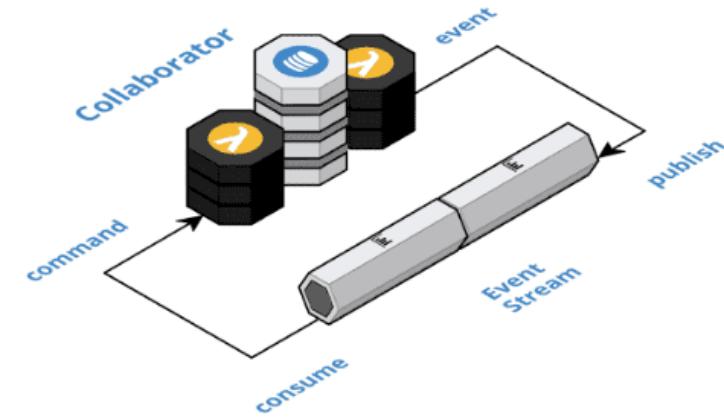
### ■ External Service Gateway

- ▶ Integrate with external systems by encapsulating the inbound and outbound inter-system communication within a bounded isolated component to provide an anti-corruption layer that acts as a bridge to exchange events between the systems.
- ▶ Treat external systems like any other bounded isolated component. Create a component for each external system that acts as a bidirectional bridge for transmitting events between the systems.
- ▶ Invoking commands on the external system follows the Event-First variant of the Event Sourcing pattern with the external system representing the database.
- ▶ Consuming events from the external system follows the Database-First variant of the Event-Sourcing pattern with the external system representing a event stream. Events are re-published to the internal event stream to delegate processing to downstream components.

## 8d) Cloud-native Architecture

- Cloud Native Control Patterns
  - ▶ Event Collaboration
  - ▶ Event Orchestration
  - ▶ Saga

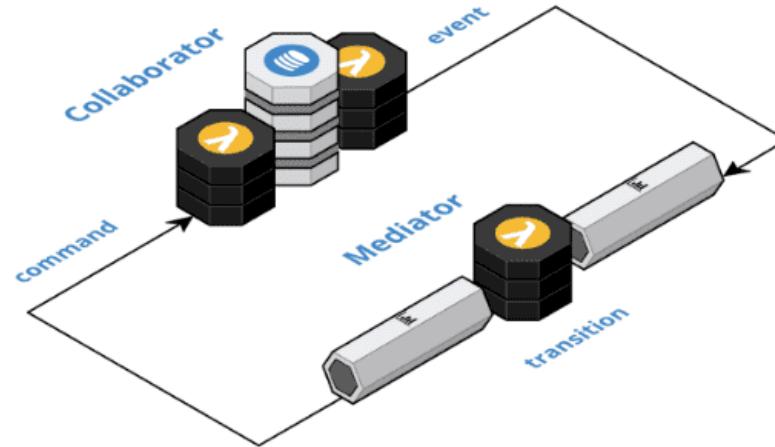
## 8d) Cloud-native Architecture



### ■ Event Collaboration

- ▶ Publish domain events to trigger downstream commands and create a reactive chain of collaboration across multiple components.
- ▶ Replace synchronous inter-component communication with asynchronous inter-component communication by using the event stream to publish events which trigger downstream commands.

## 8d) Cloud-native Architecture



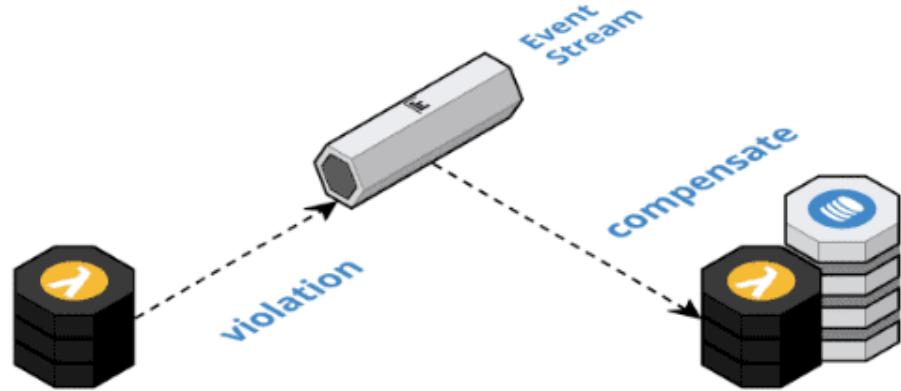
### ■ Event Orchestration

- ▶ Leverage a mediator component to orchestrate collaboration between components without event type coupling.
- ▶ Create a component for each business process to act as a mediator between the collaborator components and orchestrate the collaboration flow across those components.
- ▶ The mediator maps and translates the published events of upstream components to the consumed events of downstream components. These mappings and translations are encapsulated in the mediator as a set of rules.
- ▶ The collaborators are completely decoupled from each other and the mediator.
- ▶ The mediator defines the business process, thus only the mediator needs to change when the process evolves.

## 8d) Cloud-native Architecture

### Saga

- ▶ Trigger compensating transactions to undo changes in a multi-step flow when business rules are violated downstreams.
- ▶ Use compensating transactions to undo changes in a multi-step business process.
- ▶ Business violations are signaled as additional events and the compensations are the reactions to these specific events.
- ▶ Compensation is not equal to a rollback. The intent is not to erase the existence of the execution of a previous step. We want to have an audit trail that the previous step was completed and later reversed.
- ▶ In some domains, such as public accounting, this is a requirement.



## 8e) Microsoft Cloud Architekturstile

Architekturstil	Abhängigkeitsmanagement	Domäentyp
<b>N-Schichten</b>	Horizontale Ebenen, unterteilt durch Subnetz	Traditionelle Business-Domäne. Die Frequenz der Aktualisierungen ist niedrig
<b>Web-Queue-Worker</b>	Front- und Back-End-Aufgaben, entkoppelt durch asynchrones Messaging	Relativ einfache Domäne mit einigen ressourcenintensiven Aufgaben
<b>Mikroservices</b>	Vertikale (funktional) zerlegte Services, die sich gegenseitig über APIs aufrufen	Komplizierte Domäne. Häufige Updates
<b>Command and Query Responsibility Segregation (CQRS)</b>	Lesen/Schreiben Aufgabentrennung. Schema und Maßstab werden separat optimiert	Kollaborative Domäne, in der viele Benutzer auf dieselben Daten zugreifen
<b>Ereignisgesteuerte Architektur</b>	Produzent/Konsument. Unabhängige Ansicht pro Subsystem	IoT- und Echtzeitsysteme
<b>Big Data</b>	Aufteilen eines großen Datensatzes in kleine Blöcke. Parallel Verarbeitung in lokalen Datensätzen	Datenanalyse im Stapel und in Echtzeit. Prädiktive Analyse mit ML
<b>Big Compute</b>	Datenzuordnung an Tausende von Kernen	Berechnungsintensive Domänen wie Simulationen

Quelle : Microsoft

## 8e) Microsoft Cloud Architekturstile

### ■ N-Schichten (N-tier)

ist eine traditionelle Architektur für Unternehmensanwendungen.

Abhängigkeiten werden verwaltet, indem die Anwendung in Ebenen unterteilt wird, die logische Funktionen wie Präsentation, Geschäftslogik und Datenzugriff ausführen. Eine Ebene kann nur Ebenen aufrufen, die darunter liegen.

Diese horizontale Schichtung kann jedoch ein Problem darstellen.

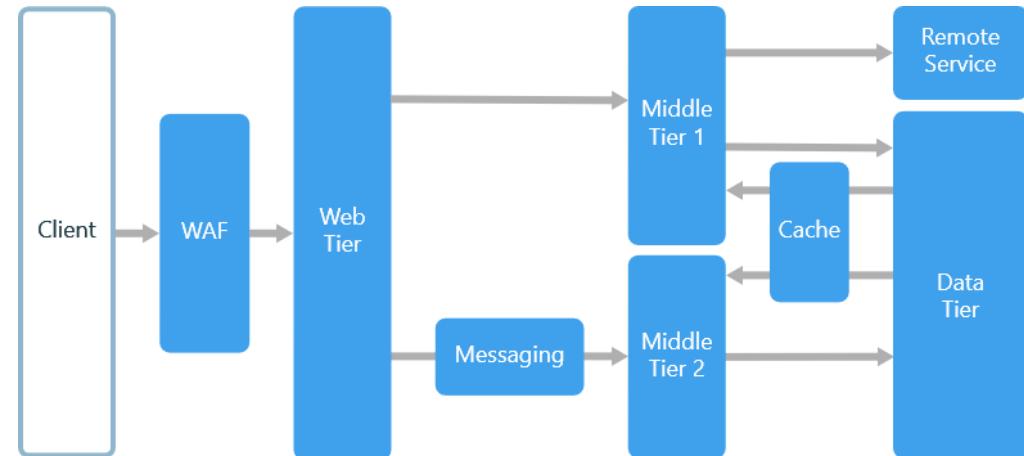
Es kann schwierig sein, Änderungen in einem Teil der Anwendung einzufügen, ohne den Rest der Anwendung zu berühren.

Das macht häufige Aktualisierungen zu einer Herausforderung und begrenzt die Geschwindigkeit, mit der neue Funktionen hinzugefügt werden können.

N-Schichten eignet sich hervorragend für die Migration vorhandener Anwendungen, die bereits eine geschichtete Architektur aufweisen.

Aus diesem Grund wird N-Schichten am häufigsten in IaaS-Lösungen (Infrastructure as a Service) verwendet oder in Anwendungen, die eine Mischung aus IaaS und verwaltete Services sind.

WAF = Web Application Firewall



## 8e) Microsoft Cloud Architekturstile

### ■ Web-Queue-Worker

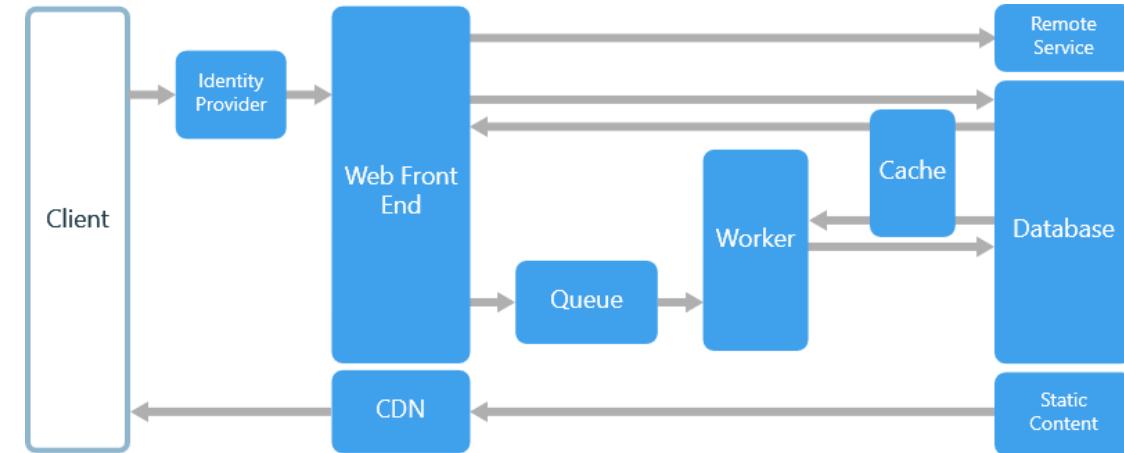
Erwägen Sie für eine reine PaaS-Lösung eine Web-Queue-Worker-Architektur. In diesem Stil verfügt die Anwendung über ein Web-Front-End, das HTTP-Anforderungen verarbeitet, und einen Back-End-Worker, der CPU-intensive Aufgaben oder lang andauernde Vorgänge ausführt. Das Front-End kommuniziert über eine asynchrone Nachrichtenwarteschlange mit dem Auftragnehmer.

Web-Queue-Worker eignet sich für relativ einfache Domänen mit einigen ressourcenintensiven Aufgaben.

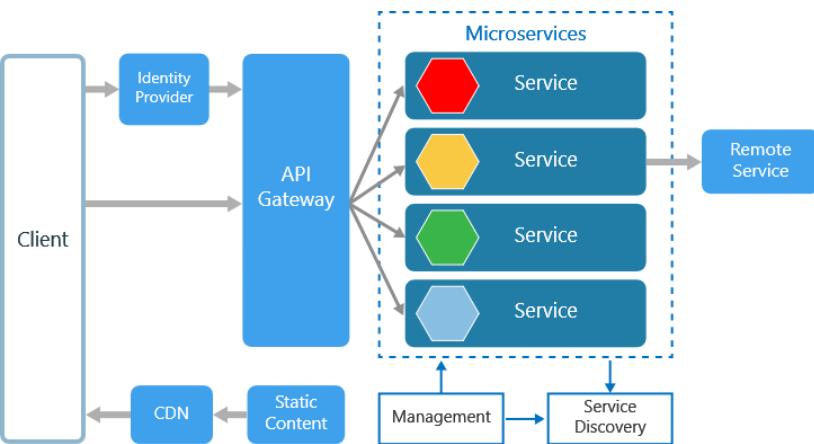
Wie bei N-Schichten ist die Architektur leicht zu verstehen. Die Verwendung von verwalteten Services vereinfacht die Bereitstellung und den Betrieb. Bei komplexen Domänen kann es jedoch schwierig sein, die Abhängigkeiten zu verwalten.

Das Front-End und der Worker können leicht zu großen, monolithischen Komponenten werden, die schwierig zu warten und zu aktualisieren sind.

Wie bei N-Schichten kann dies die Häufigkeit von Aktualisierungen reduzieren und die Innovation einschränken.



## 8e) Microsoft Cloud Architekturstile



### ■ Mikro-Services

Wenn Ihre Anwendung eine komplexere Domäne hat, überlegen Sie einen Wechsel auf eine Microservices-Architektur.

Eine Microservices-Anwendung besteht aus vielen kleinen, unabhängigen Services.

Jeder Dienst implementiert eine einzelne Business-Fähigkeit. Die Services sind lose gekoppelt und kommunizieren über API-Verträge.

Jeder Service kann von einem kleinen, fokussierten Entwicklerteam erstellt werden.

Einzelne Services können ohne große Koordinierung zwischen den Teams bereitgestellt werden, was die Möglichkeit häufiger Aktualisierungen fördert. Eine Microservice-Architektur ist komplexer zu entwickeln und als N-Schichten oder Web-Queue-Worker schwerer zu verwalten.

Das erfordert eine ausgereifte Entwicklung und DevOps-Kultur. Aber richtig gemacht, kann dieser Stil zu höherer

Veröffentlichungsgeschwindigkeit, schnellerer Innovation und einer widerstandsfähigeren Architektur führen.

## 8e) Microsoft Cloud Architekturstile

### CQRS

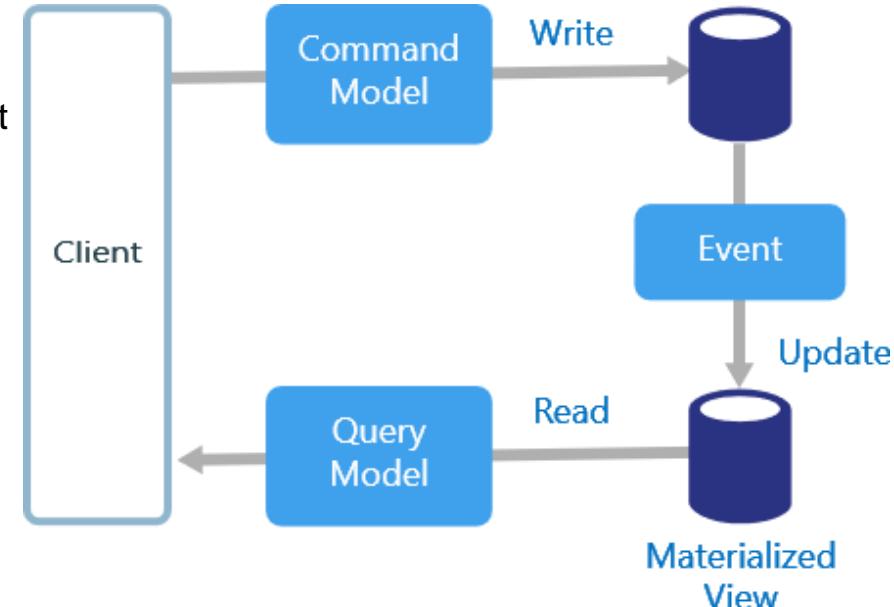
Der CQRS-Stil – Command and Query Responsibility Segregation (CQRS) Pattern – trennt Lese- und Schreibvorgänge in separate Modelle. Dies isoliert die Teile des Systems, die Daten von den Teilen aktualisieren, von denen die Daten lesen.

Darüber hinaus können Lesevorgänge für eine materialisierte Ansicht ausgeführt werden, die physisch von der Schreibdatenbank getrennt ist.

Dadurch können Sie die Workloads für das Lesen und Schreiben unabhängig skalieren und die materialisierte Ansicht für Abfragen optimieren. CQRS ergibt vor allem dann Sinn, wenn es auf ein Subsystem einer größeren Architektur angewendet wird.

Sie sollten es nicht für eine gesamte Anwendung nutzen, da dies nur unnötige Komplexität verursacht.

Verwenden Sie es für kollaborative Domänen, in denen viele Benutzer auf dieselben Daten zugreifen.



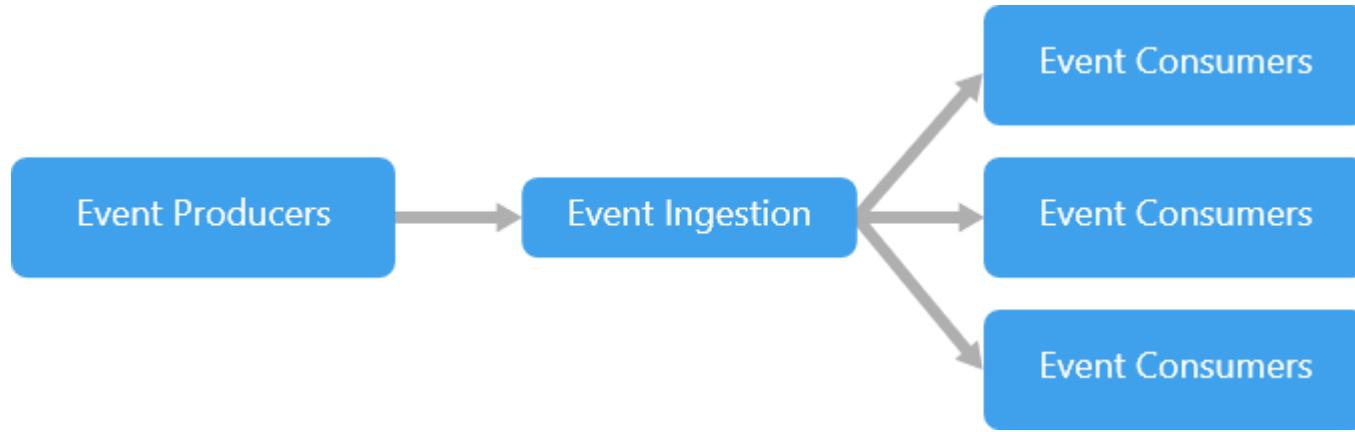
## 8e) Microsoft Cloud Architekturstile

### ■ Ereignisgesteuerte Architektur

Ereignisgesteuerte Architekturen verwenden ein Publish-Subscribe-Modell (Pub-Sub-Modell), bei dem die Produzenten Ereignisse veröffentlichen und Kunden sie abonnieren.

Die Produzenten sind unabhängig von den Konsumenten und die Konsumenten sind voneinander unabhängig.

Verwenden Sie eine ereignisgesteuerte Architektur für Anwendungen, die eine große Datenmenge mit sehr geringer Latenz verarbeiten, z. B. IoT-Lösungen. Der Stil ist auch nützlich, wenn verschiedene Subsysteme unterschiedliche Verarbeitungstypen für dieselben Ereignisdaten ausführen müssen.



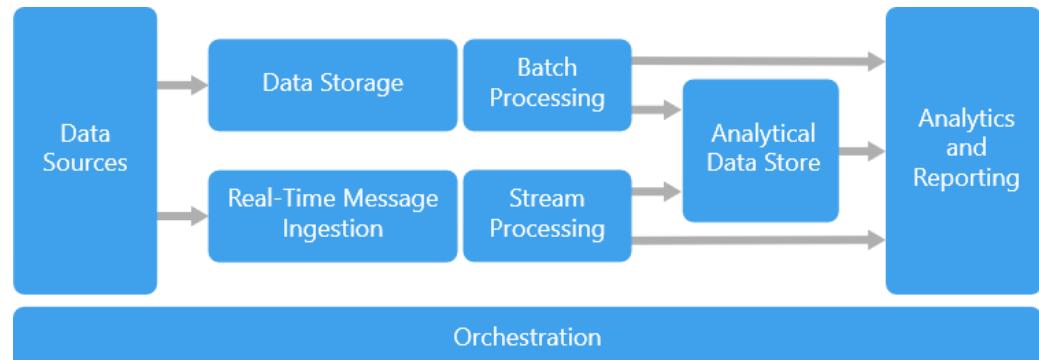
Internal

## 8e) Microsoft Cloud Architekturstile

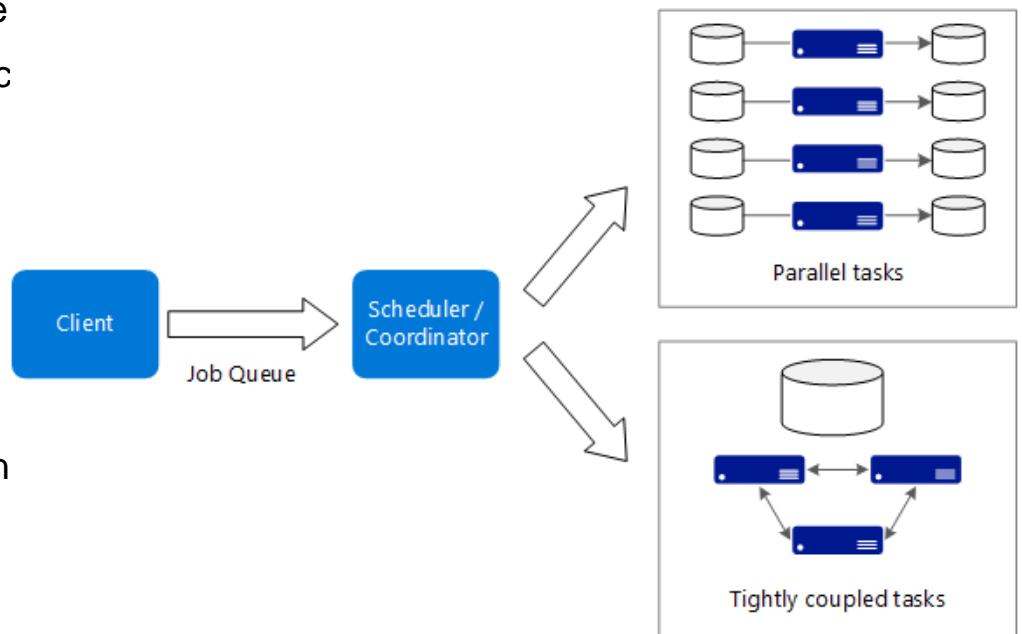
### ■ Big Data und Big Compute

Big Data und Big Compute sind spezialisierte Architekturstile für Workloads, die bestimmten spezifischen Profilen entsprechen.

Big Data unterteilt sehr große Daten-Sets in Chunks und führt die parallele Verarbeitung über das gesamte Set hinweg für Analysen und Berichte durc



Big Compute, auch High-Performance Computing (HPC) genannt, führt parallele Berechnungen über eine große Anzahl von (tausenden) Kernen durch. Zu den Domänen gehören Simulationen, Modellierungen und 3D-Renderin



## 9) Performance

- Benutzer besitzen i.a. hohe Anforderungen an die Performance von Anwendungen
- Performance Anforderungen finden sich in den Qualitätsattributen und sind Teile einer guten Architektur
- Architekten und Software Entwickler müssen sich mit Performance Themen beschäftigen und die Verantwortung dafür übernehmen
- Wir betrachten
  - a) die Bedeutung von Performance
  - b) Performance Terminologien
  - c) Systematische Vorgehensweisen
  - d) Verschiedenste Strategien zur Verbesserung der Performance

## 9a) Die Bedeutung von Performance

- Die Performance einer Anwendung ist ein Maß der Reaktionsfähigkeit ihrer Tätigkeiten
- Benutzer haben heute höhere Anforderungen als früher
- Sie erwarten gute bis sehr gute Antwortzeiten unabhängig vom Ort und des benutzten Devices
- Sollte eine Anwendung allerdings den Anforderungen nach Benutzbarkeit und funktionalen Anforderungen nicht Gerecht werden, dann spielt Performance auch keine Rolle mehr
- Erfüllt hingegen eine Anwendung alle Anforderungen, dann ist gute Performance um so wichtiger
  - ▶ in Webanwendungen sind z.B. die Ladezeiten von Seiten enorm wichtig
- Gute Performance erhöht die Produktivität der Anwender
- Gute Performance von Webseiten erhöht das Ranking in Suchmaschinen
- Performance ist ein Requirement und deshalb in den Qualitätsattributen

## 9b) Performance Terminologien

### ■ Bounce rate

- ▶ manchmal auch Exit rate genannt
- ▶ ein bounce eines Benutzer ist der Besuch und das Verlassen einer Seite ohne andere Seiten einer Anwendung besucht zu haben

$$\text{Bounce rate} = \frac{\text{total number of bounces}}{\text{total entries to a page}}$$

- ▶ wenn eine Seite langsam lädt, erhöht sich zwangsläufig die Bounce rate
- ▶ bounces sind jegliche Art von Verlassen einer Seite  
(exit, back, neue Url, Browserfenster schließen, ...)

## 9b) Performance Terminologien

### ■ Conversion rate

- ▶ ist der Prozentsatz von Benutzern, die eine Seite besuchen und die gewünschte Aktion durchführen
  - eine Bestellung machen, Registrierung als Mitglied, Download einer Datei, Bestellung eines Newsletters

$$\text{Conversion rate} = \frac{\text{number of goal achievements}}{\text{number of visitors}}$$

- ▶ Anwendungen mit schlechter Performance haben i.a. eine niedrige Conversion rate

## 9b) Performance Terminologien

### ■ Latency

- ▶ die Zeit (oder die Wartezeit), die benötigt wird, um Informationen von einer Quelle zu einem Ziel zu schicken
- ▶ manchmal auch beschrieben als die Zeit im Kabel oder im Netz
- ▶ Normalerweise in Millisekunden gemessen
- ▶ Einflußfaktoren sind Hardware, Netzwerkkomponenten wie Router u.ä., der Verbindungstyp, die Entfernung und der Netzwerkverkehr
- ▶ Oft besteht ein Großteil der gesamten Latency aus der Latency zwischen dem Endbenutzer und dem Internet Service Provider (ISP) – dann redet man von last-mile latency

## 9b) Performance Terminologien

- Throughput (Durchsatz)
  - ▶ eine Menge von Arbeitseinheiten pro Zeiteinheit
  - ▶ im Kontext eines Netzwerks ist das die Menge der übertragenen Daten pro Zeiteinheit
  - ▶ Typische Maßeinheiten sind
    - bits per second (bps)
    - megabits per second (Mbps)
    - gigabits per second (Gbps)
  - ▶ Im Kontext einer Anwendung ist Durchsatz die Anzahl von durchgeföhrten Aktionen pro Zeiteinheit
    - z.B. die Anzahl der Transaktionen pro Sekunde

## 9b) Performance Terminologien

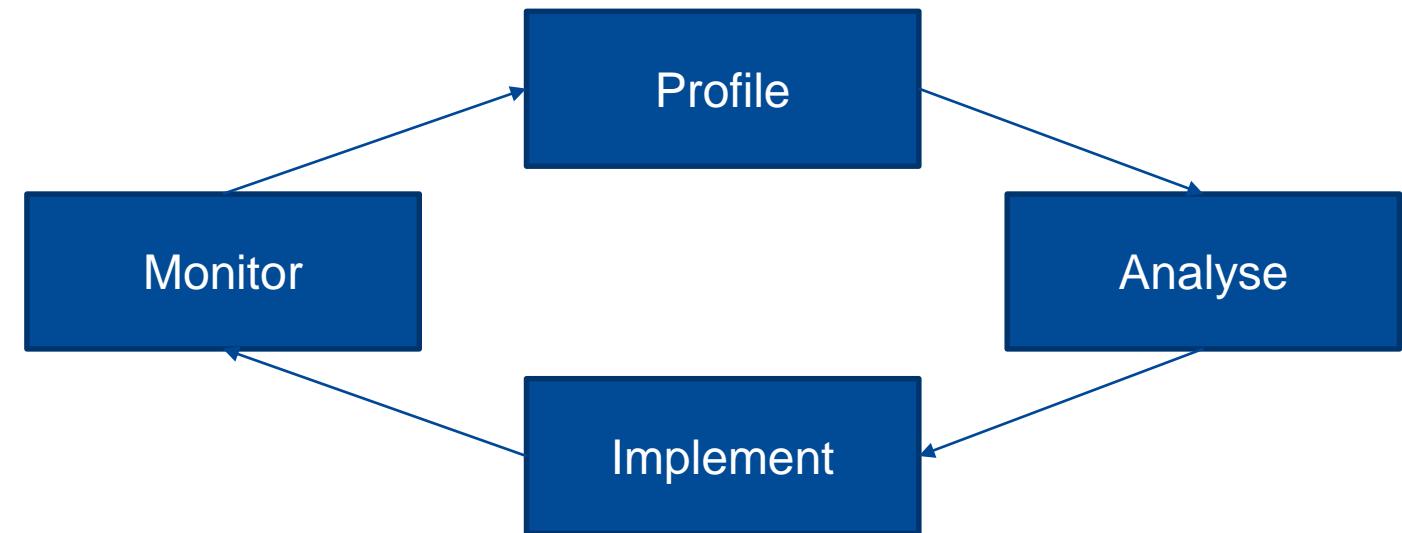
- Bandwidth (Bandbreite)
  - ▶ ist der maximale Durchsatz eines logischen oder physikalischen Kommunikationsweges
- Processing time
  - ▶ die Zeitdauer, die eine Anwendung benötigt, um eine Anforderung abzuarbeiten
  - ▶ Latency wird dabei nicht berücksichtigt
  - ▶ manchmal wird zwischen Server Processing time und Client Processing time unterschieden
  - ▶ es gibt einige Faktoren, die die Processing time beeinflussen können  
(Hardware, Software, Architektur)
- Response time (Antwortzeit)
  - ▶ die Zeitdauer zwischen der Anforderung eines Benutzers und der dazugehörigen Antwort des Systems
  - ▶ Latency wird dabei nicht berücksichtigt (kann aber oft vom Benutzer nicht getrennt werden)

## 9b) Performance Terminologien

- Workload (Last)
  - ▶ die Menge von Verarbeitungsschritten, die ein System zu einer Zeit zu bewältigen hat
  - ▶ Verarbeitungsschritte benutzen Anteile von Prozessorkapazitäten und lassen damit weniger für andere Verarbeitungen übrig
  - ▶ Typische Workloads sind CPU-, Memory-, I/O- und Datenbank-Workloads
  - ▶ Regelmäßige Messungen von Workloads helfen Spitzen Belastungen vorherzusagen und die Performance einer Anwendung bei unterschiedlichen Belastungen zu messen und zu optimieren
- Utilization (Auslastung)
  - ▶ die Prozentzahl der Zeit, in der eine Resource benutzt wird, verglichen mit der Gesamtverfügbarkeit einer Resource
  - ▶ Beispiel : eine CPU ist innerhalb einer Minute 45 Sekunden mit einer Transaktion beschäftigt dann ist die Utilization 75%
  - ▶ Utilization von CPU, Memory und Platten spielen eine wichtige Rolle bei der Performancebetrachtung von Anwendungen

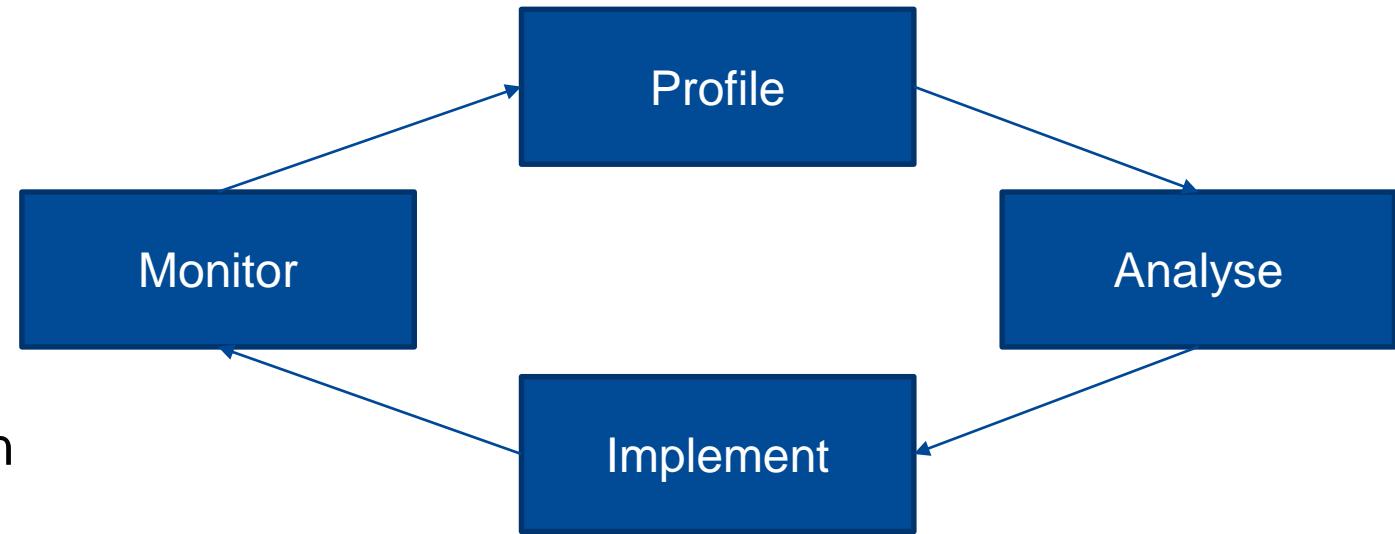
## 9c) Systematische Vorgehensweisen

- Bei der Performancebetrachtung muss das Entwicklungsteam involviert sein
- Das Entwicklungsteam muss die Verantwortung für die Performance einer Anwendung übernehmen
- Um die Performance zu verbessern, ist es sinnvoll systematisch vorzugehen
- Eine beispielhafte Vorgehensweise benutzt einen iterativen Ansatz
  - ▶ eine Anwendung messen (Profiling)
  - ▶ die Ergebnisse auswerten
  - ▶ Änderungen implementieren
  - ▶ Änderungen überwachen (Monitoring)



## 9c) Systematische Vorgehensweisen

- eine Anwendung messen (Profiling)
  - ▶ ist eine Analyse einer Anwendung, die die Ausführung einer Anwendung misst
  - ▶ überall in einer Anwendung werden dazu Meßpunkte/Meßstellen implementiert
  - ▶ sie erfassen Daten wie
    - Laufzeit von Modulen
    - Anzahl Aufrufe von Modulen
    - Durchsatz
    - Antwortzeiten
    - Auslastung
- Es gibt dazu Profiling Tools am Markt
- Es gibt 2 Arten wie Profiler Daten sammeln
  - ▶ Instrumentation
  - ▶ Statistical

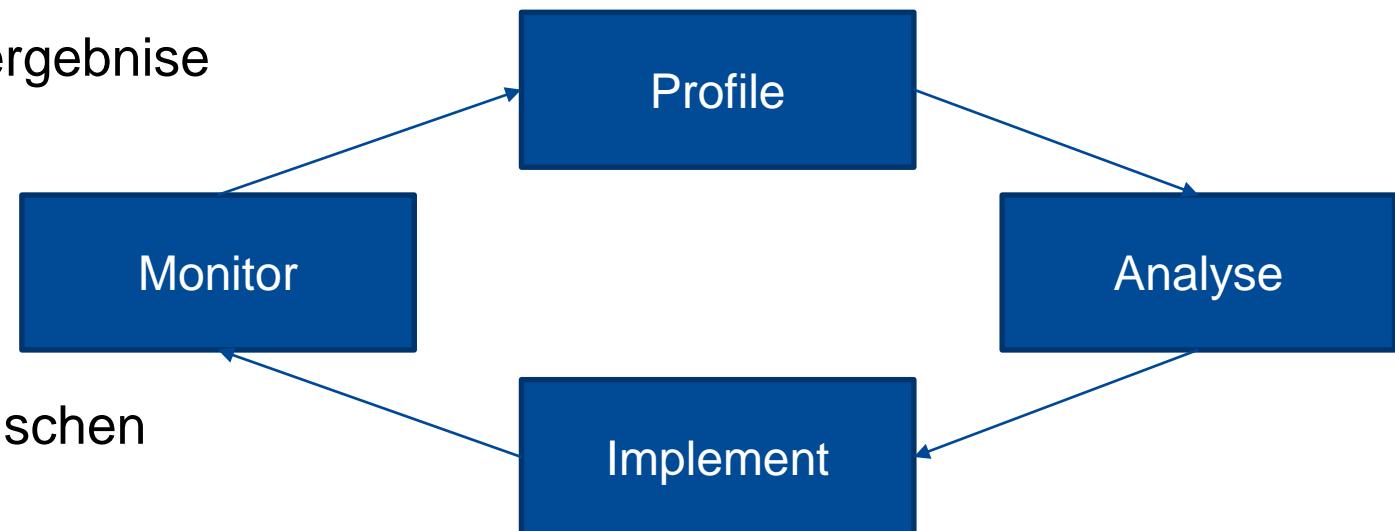


## 9c) Systematische Vorgehensweisen

### ■ Arten von Profilern

#### ▶ Instrumentation

- Code wird zur Anwendung hinzugefügt
- ein Call in den Profiler am Anfang und Ende jeder Funktion
- es gibt Profiler, die Source Code modifizieren können
- der eingefügte Code kann die Meßergebnisse beeinflussen
- gute Profiler berücksichtigen das
- eingefügter Code kann aber auch die Optimierungsstrategie der CPU beeinflussen und damit Werte verfälschen

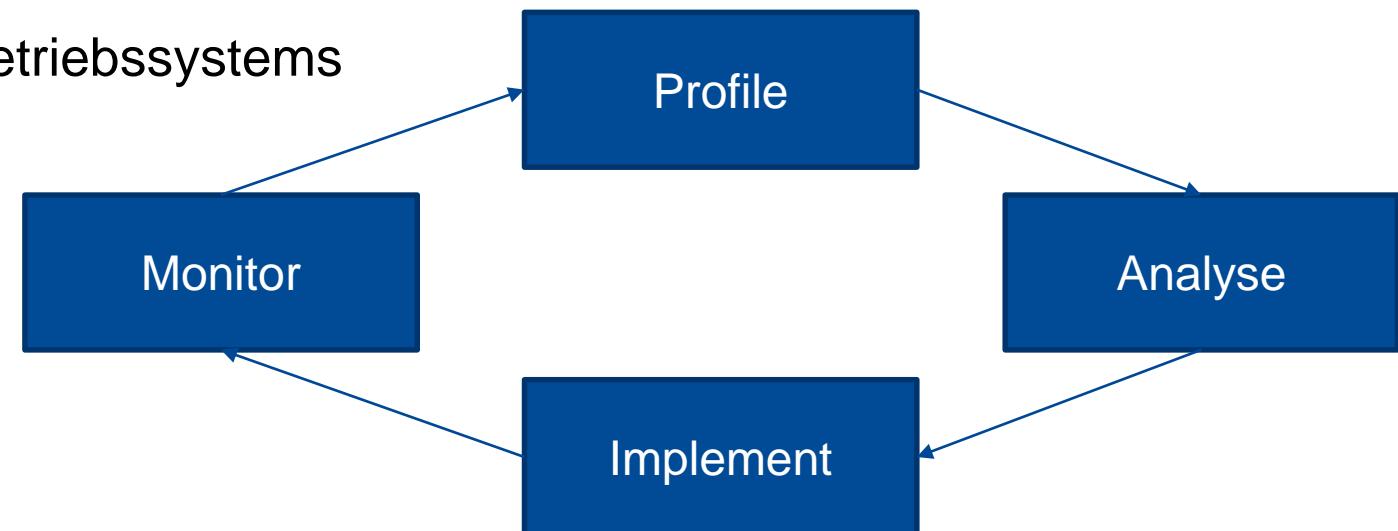


## 9c) Systematische Vorgehensweisen

### ■ Arten von Profilern

#### ▶ Statistical

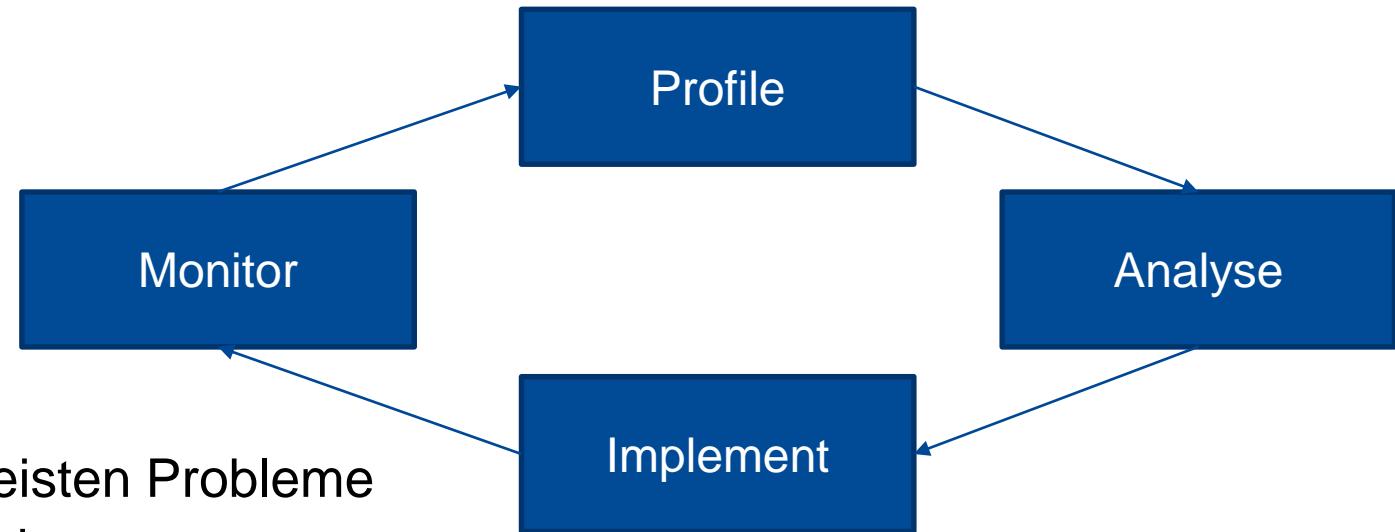
- sammelt Messwerte außerhalb einer Anwendung
- ohne Code einzufügen
- benutztes Verfahren : Sampling
- Sampling nutzt die Interrupts des Betriebssystems zum Sammeln von Messwerten
- Messergebnisse sind deshalb oft nur Näherungswerte
- Instrumentation liefert genauere Werte



## 9c) Systematische Vorgehensweisen

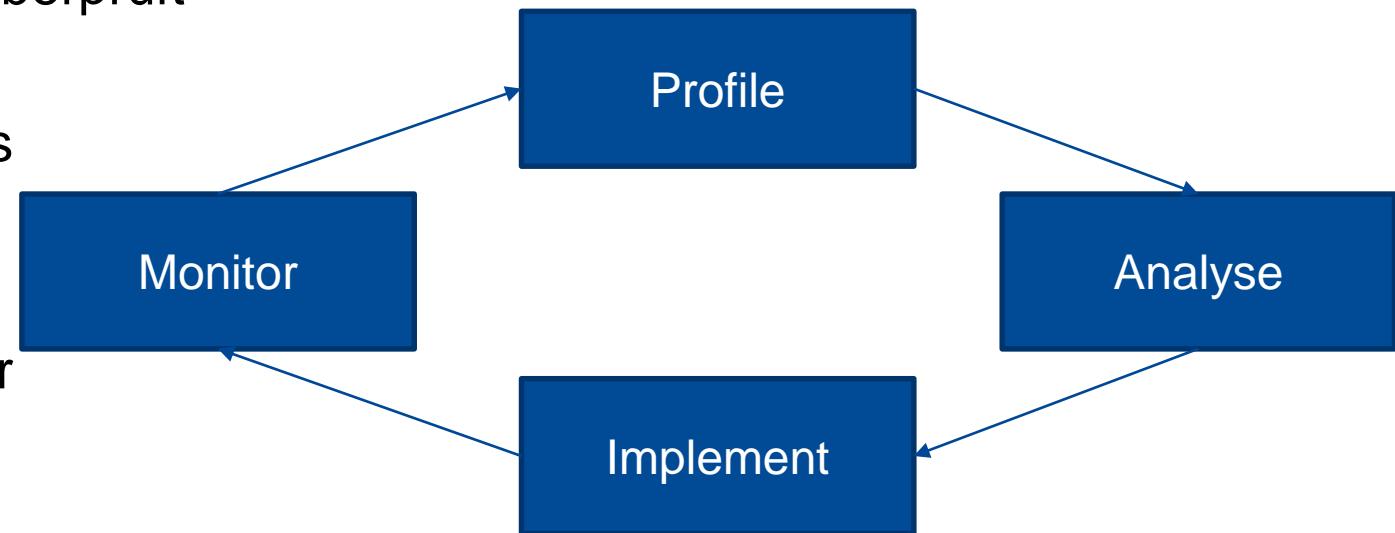
### ■ Analyse der Daten

- ▶ zum Aufspüren von Bottlenecks
- ▶ Bottlenecks sind die Teile einer Anwendung, die die Performance negativ beeinflussen
- ▶ der Fokus liegt also auf der Optimierung/Behebung der Bottlenecks
- ▶ Bottlenecks könnten sein
  - CPU
  - Memory
  - Netzwerk
  - Datenbanken
  - I/O
- ▶ die (meine) Erfahrung zeigt, daß die meisten Probleme in der Anwendung selbst verborgen sind
- ▶ Entwickler wollen das i.a. nicht wahrhaben



## 9c) Systematische Vorgehensweisen

- Implementieren der Änderungen
  - ▶ basierend auf den Messergebnissen
- Überwachen der Änderungen
  - ▶ die Änderungen müssen auf Wirkung überprüft und überwacht werden
  - ▶ es kann sein, daß sich jetzt Bottlenecks in andere Bereiche der Anwendung verschoben haben
  - ▶ Monitoring von Anwendungen ist immer angebracht
  - ▶ der Zyklus kann von vorne beginnen



## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Caching

- ▶ Caching ist ein gängiges Hilsmittel zur Erhöhung der Performance
- ▶ benötigt genaue Planung und muss in die Architektur aufgenommen werden
- ▶ Caching kann den Zugriff auf langsame Systeme/Medien optimieren
- ▶ Caching bedeutet immer das Anlegen von Kopien auf anderen (schnelleren) Medien
- ▶ der Cache sollte nah bei der Anwendung liegen, um Latency zu vermeiden
- ▶ Caching erzeugt zusätzlichen Verwaltungsaufwand

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Caching

- ▶ In verteilten Anwendungen findet man
  - privater (lokaler) Cache auf dem Rechner des Endusers
    - schneller Cache im Memory
    - Einstellungen, Credentials
  - gemeinsam genutzter Cache am Server
    - Cache Service oder spezielle Implementierung
    - genutzt von mehreren Instanzen oder Cluster der Anwendung
    - kann in Storage systemen eingebaut sein
    - kann in spezieller Hardware/Software/Systemen im Netz verteilt sein

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Caching

- ▶ Priming
  - vorfüllen eines Caches beim Starten einer Anwendung
  - wegschreiben des Caches beim Beenden einer Anwendung
- ▶ Nicht mehr benötigte Cache-Inhalte (Invalidating) löschen oder ersetzen
  - kann extrem kompliziert sein
  - es gibt 2 gebräuchliche Strategien
    - Expiring
    - Evicting

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

- Caching
  - ▶ Cache Strategien
    - Expiring
      - nach Datum oder Zeitspanne (zum 31.12.2019, 1 Tag)
    - Evicting
      - Least recently used (die am wenigsten benutzten Daten werden gelöscht)
      - most recently used (gerade benutzt, in absehbarer Zeit nicht nochmal benutzt)
      - First-in First-out (zuerst eingestellte (älteste) Daten werden gelöscht)
      - Last-in First-out (selten benutzt)
      - explicitly evicting (z.B. aus dem Cache gelöscht, wenn aus der Datenbank gelöscht)

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Caching

#### ▶ Cache Pattern

- es gibt für Anwendungen 2 Wege Caches zu nutzen
  - die Anwendung verwaltet den Cache selbst neben dem eigentlichen Datenpool
    - cache aside
  - die Anwendung sieht den Cache als den Datenpool, der dann unabhängig von der Anwendung den dahinterliegenden echten Datenpool verwaltet
    - read through (Cache im Lesepfad)
    - write through (Cache im Schreibpfad)
    - write behind (Puffern von Schreibanweisungen, verzögertes Schreiben)

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
  - ▶ Vermeidung von Datentransfer über das Netzwerk
  - ▶ Browercaching von Daten (z.B. große Bilddateien)
  - ▶ Sharing von CSS oder Javascript Dateien über mehrere Webseiten
  - ▶ Benutzung von entsprechenden Direktiven in den HTTP Headern
    - Validation token (ETag header)
    - Cache-control directives

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ HTTP Caching (Webanwendungen)

#### ▶ Benutzung von entsprechenden Direktiven in den HTTP Headern

##### ● Validation token (**ETag** header)

- der ETag HTTP response header ist ein Identifier für eine bestimmte Version einer Resource. Er trägt zur Effizienzsteigerung von Caches bei und spart Bandwidth, weil ein Webserver nur die Teile schicken muss, die sich geändert haben
- Zusätzlich helfen Etags zu verhindern, daß sich simultane Updates auf Rersourcen gegenseitig überschreiben ("mid-air collisions")
- ETags sind Fingerabdrücken ähnlich und können in Vergleichen verwendet werden
- Beispiel :
  - ETag: "33a64df551425fcc55e4d42a148795d9f25f89d4" ist der Hash einer Webseite
  - Wenn Änderungen dieser Seite gespeichert werden sollen, enthält der POST request einen If-Match header mit dem Inhalt des ETag zur Überprüfung auf Änderungen (If-Match: "33a64df551425fcc55e4d42a148795d9f25f89d4")
  - Sind die Hashes ungleich, wurde die Seite inzwischen geändert und ein 412 Precondition Failed error wird erzeugt

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ HTTP Caching (Webanwendungen)

- ▶ Benutzung von entsprechenden Direktiven in den HTTP Headern
  - Cache Control directives
    - steuern das Cachen von Responses (Antworten), die Bedingungen zum Cachen und die Zeitspannen
    - für das Verhindern der Ablage von sensiblen Daten in Caches zu verhindern, wird die Direktive **no-store** verwendet (gilt für den Browercache und alle Caches auf dem Weg zum Browser)
    - die **private** Direktive erlaubt das Cachen im Browser des Benutzers, verhindert aber die Ablage in Caches auf dem Weg
    - die **non-cache** Direktive sorgt im Zusammenspiel mit der If-match Direktive, ob aus dem Cache gelesen wird oder nicht
    - die **max-age** Direktive gibt an wie lange (in Sekunden und relativ zur Zeit der Antwort) eine Antwort aus dem Cache gelesen werden kann
    - Caches können vom Benutzer im Browser gelöscht werden
    - Eine oft benutzte Variante zum gesteuerten Nachladen von Ressourcen ist die Veränderung der URL, was ein Nachladen auslöst
    - Bei URL's auf Dateien kann ein Fingerprint eingefügt werden, was eine Art der Versionierung direkt unterstützt kann

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ HTTP Caching (Webanwendungen)

- ▶ Kompression
  - eine wichtige Technik zur Performancverbesserung
  - Nutzung eines Algorithmus zur Verkleinerung von Ressourcen
  - Verbessert Transferzeiten und Bandbreiten Nutzung
  - Webserver und Browser haben entsprechende Funktionen bereits implementiert
  - Webserver und Browser müssen sich über den benutzten Algorithmus einig sein
  - die 2 Haupttypen von Kompression sind
    - file compression
    - content-encoding (end-to-end) compression

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ HTTP Caching (Webanwendungen)

#### ▶ Kompression

##### — file compression

- Images, Videos und Audiodateien haben eine hohe Rate von Redundanz
- Images machen in modernen Webseiten den Großteil der Bytes, die geladen werden, aus
- diese Ressourcen sollten unbedingt komprimiert werden
- dafür gibt es verschiedene Tools und Algorithmen
- man unterscheidet 2 Kompressionsalgorithmen
  - lossless compression (verlustlose Komprimierung)
    - alle Bytes einer Resource können 1:1 wiederhergestellt werden
    - GIF (Graphics Interchange File) und PNG (Portable Network Graphics) sind Dateiformate, die lossless compression beinhalten
    - Images hoher Qualität und der Anforderung nach verlustfreiem Transfer werden deshalb in PNG gespeichert
  - lossy compression
    - nicht verlustfrei bei der Komprimierung
    - funktioniert gut bei Images, Video und Audio Dateien, bei denen es nicht auf hohe Qualität ankommt
    - es gibt verschiedene Stufen der Komprimierung
    - JPEG (Joint Graphic Expert Group) ist ein Beispiel

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)
  - ▶ Kompression
    - Content-encoding(end-to-end) compression
      - signifikante Performancesteigerungen möglich
      - der komplette Body-Teil einer Nachricht wird komprimiert
      - nur der Klient (Browser) dekomprimiert die Nachricht, keine Stellen dazwischen
      - Server und Klient müssen den Algorithmus via ***content negotiation*** abstimmen
      - es gibt die gebräuchlichen Algorithmen ***content-encoding gzip*** und ***content-encoding br*** (Brotli)
      - bereits komprimierte Ressourcen werden bei nochmaliger Komprimierung nicht kleiner, eher größer

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

- HTTP Caching (Webanwendungen)

- ▶ Minifying Resourcen
    - Minification nennt man das Entfernen aller unnötigen Zeichen aus Dateien
      - Leerzeichen aus Javascript, HTML und CSS
  - ▶ Bundling Resourcen
    - das Zusammenführen mehrerer Files mit gleicher Aufgabe in ein File (foo.css und bar.css in styles.css)

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ HTTP/2

- ▶ rückwärtskompatibel zu HTTP/1.1
- ▶ ist ein binäres Übertragungsprotokoll (HTTP/1.1 ist textorientiert)
- ▶ kompakter, einfacher zu parsen und weniger fehleranfällig
- ▶ kann Multiplexing
  - paralleles und asynchrones Senden und Empfangen von Nachrichten über eine TCP Verbindung
  - macht Bundling überflüssig und die Nutzung von Caches werden optimiert
- ▶ kann Server Push
  - proaktives Senden von benötigten Ressourcen vom Server an den Client
  - vorsichtig einzusetzen, kann zu zuviel Übertragungen führen
- ▶ Header Kompression
  - HPACK Kompressionsformat (Huffman lossless)

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Content Delivery Networks

- ▶ Benutzer von Ressourcen können weltweit verteilt sein
- ▶ um Latency zu vermindern gibt es sog. Content Delivery Networks (CDN)
  - geografisch verteilte Servergruppen, um Content local Benutzern zur Verfügung zu stellen
- ▶ CDN unterstützen zusätzlich Load Balancing, Caching, Minification und Dateikomprimierung

### ■ Optimierte Webfonts

- ▶ spezielle Fonts werden in den Browser geladen (wenn unterstützt)
- ▶ durch geschickten Einsatz von Webfonts kann man Ladezeiten von Webpages verringern
- ▶ Leider gibt es noch keinen einheitlichen Standard

### ■ Optimierung des kritischen Rendering Paths (critical rendering path CRP)

- ▶ Wenn man die Art und Weise, wie ein Browser eine Site rendernd, kennt, dann kann man die Seiten entsprechend aufbauen, daß die interne Erzeugung des Document Object Model (DOM) und des CSS Object Model schnell abläuft

## 9c) Verschiedenste Strategien zur Verbesserung der Performance

### ■ Datenbank Performance

- ▶ Effizientes Datenbankschema
  - Normalisierung
  - Manchmal muß aus Performancegründen (teilweise) denormalisiert werden
  - Nutzung von Schlüsseln (Primary, Foreign)
  - Auswahl von geeigneten Datentypen (Konvertierung in andere Typen kostet Zeit)
  - Nutzung von Indizes
  - Skalierung von Datenbankservern
  - Nutzung von Transaktionen
  - CAP Theorem
    - Consistency, Availability, Partition intolerance
    - nur 2 der 3 Anforderungen können in verteilten Systemen erfüllt werden
    - manche NoSql Datenbanken stellen Performance über Consistency

# 10) Security

- Software Systeme zu designen und entwickeln, die eine hohe Sicherheit besitzen, wird im wichtiger
- Die Anzahl von Attacken auf Firmennetzwerke und Anwendungen nimmt dramatisch zu
- Der Schaden durch solche Attacken kann für Firmen existenzbedrohend sein
- Wir betrachten
  - a) Zustände von Informationen
  - b) das CIA Dreieck
  - c) Bedrohungsmodellierung
  - d) Secure by Design
  - e) Kryptografie
  - f) Identity und Access Management (mit Authentifizierung und Authorisierung)
  - g) Einige Risiken von Webanwendungen

# 10) Security

- Security ist die Fähigkeit einer Software Anwendung heimtückische Angriffe zu vermeiden und abzuwehren und die nicht genehmigte Benutzung der Anwendung und deren Daten zu verhindern
- sie schützt das wichtigste Kapital einer Firma : die Informationen
- Software Architekten müssen Security Themen in ihre Arbeit integrieren und von Anfang an beachten
- Security ist eines der Qualitätsattribute und wird aus entsprechenden Anforderungen abgeleitet
- Security ist ein Teil der Architektur und muss bei der Erfassung der Anforderungen, beim Design, in der Entwicklungsphase und beim Testen berücksichtigt werden

## 10a) Zustände von Informationen

- Informationen, die abgesichert werden sollen, können in 3 Zuständen vorliegen
  - ▶ im Ruhezustand (at rest)
  - ▶ in Gebrauch (in use)
  - ▶ auf dem Transportweg (in transit)
- in allen 3 Zuständen müssen Informationen abgesichert werden
  - ▶ sichere Ablage
  - ▶ sichere Verarbeitung
  - ▶ sichere Übertragung

# 10b) Das CIA Dreieck

## ■ CIA Triad

- ▶ Confidentiality (Vertraulichkeit)
  - unberechtigter Zugriff auf Informationen vermeiden
- ▶ Integrity (Integrität)
  - unberechtigtes Verändern oder Löschen von Informationen vermeiden
- ▶ Availability (Verfügbarkeit)
  - Informationen Berechtigten zeitnah zur Verfügung stellen



## 10c) Bedrohungsmodellierung

- Threat modelling
  - ▶ strukturierte Vorgehensweise, um die Sicherheit von Anwendungen zu analysieren
  - ▶ kommt aus der Sicht des Angreifers, greift die Anwendung von außen an
  - ▶ identifiziert und priorisiert potentielle Sicherheitsbedrohungen, damit Entwicklerteams erkennen können, wo ihre Software am meisten angreifbar ist
  - ▶ nach der Analyse hilft ein daraus resultierender Plan die Sicherheitslücken zu verkleinern oder zu schließen
  - ▶ betrachtet die Anwendung in Teilen (decomposing)
    - wo ist das wertvollste Gut
    - was könnten mögliche Ziele von Attacken sein
  - ▶ betrachtet die Angreifer
    - interne und externe Angreifer
  - ▶ betrachtet alle Schnittstellen zu anderen Systemen und Benutzern

# 10c) Bedrohungsmodellierung

## ■ Potentielle Bedrohungen

### ▶ Threat Kategorisierungsmodell STRIDE

- Spoofing Identity (Verschleierung)
  - sich als jemanden anderen ausgeben
- Tampering with data (Verfälschung)
  - modifizieren, verfälschen von Daten
- Repudiation (Zurückweisung)
  - zurückweisen von Verantwortlichkeit (z.B. da nicht protokolliert)
- Information disclosure (Offenlegung)
  - fehlende Absicherung von Informationen gegen unberechtigten Zugriff
- Denial-of-service (Abweisen, Stören)
  - stören und/oder ausschalten von Services
- Elevation of privilege (Erhöhung)
  - unberechtigtes Erlangen höherer Berechtigungen

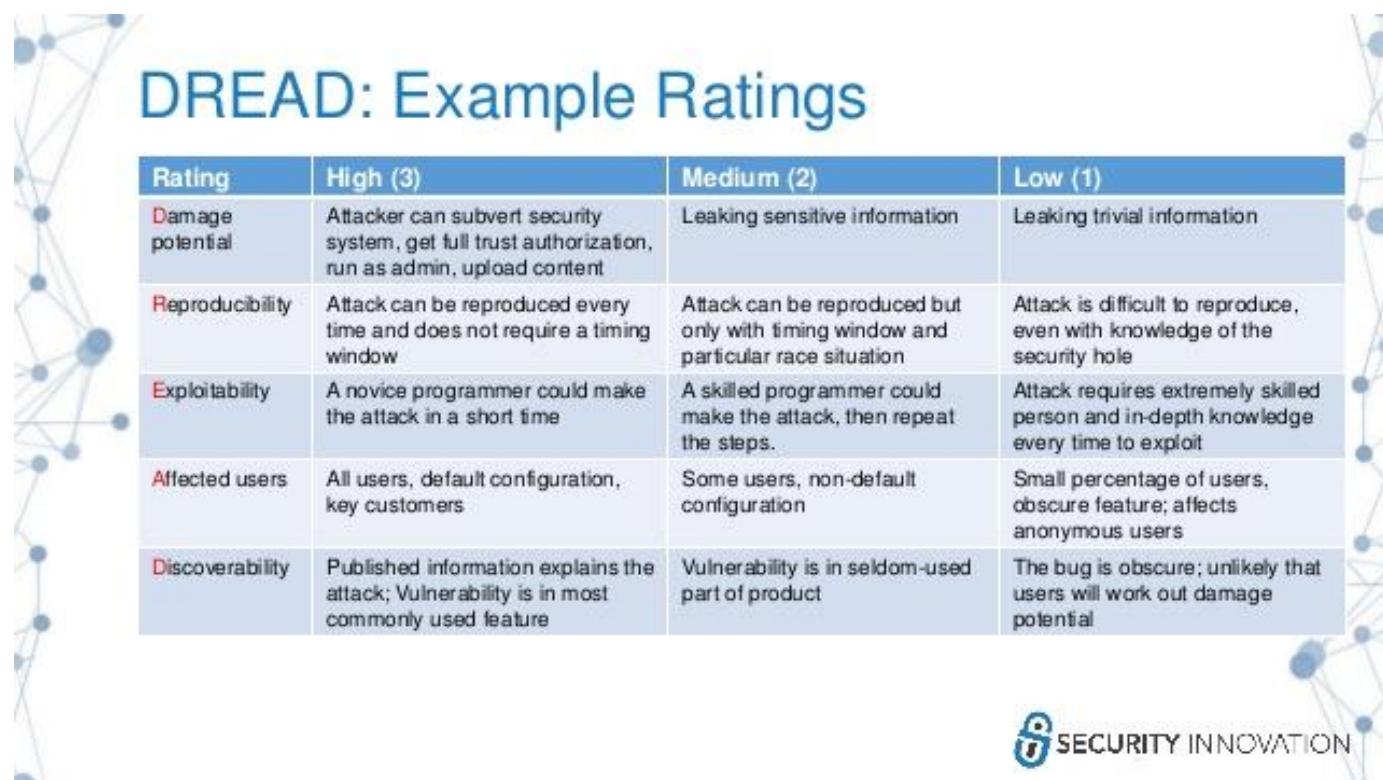
Threat	Definition
Spoofing	An attacker tries to be something or someone he/she isn't
Tampering	An attacker attempts to modify data that's exchanged between your application and a legitimate user
Repudiation	An attacker or actor can perform an action with your application that is not attributable
Information Disclosure	An attacker can read the private data that your application is transmitting or storing
Denial of Service	An attacker can prevent your legitimate users from accessing your application or service
Elevation of Privilege	An attacker is able to gain elevated access rights through unauthorized means

# 10c) Bedrohungsmodellierung

## ■ Potentielle Bedrohungen

### ▶ Risk assessment model DREAD (Priorisierungsmodell)

- Damage potential
  - potentieller Schaden
- Reproducibility
  - Reproduzierbarkeit einer Attacke
- Exploitability
  - Nachvollziehbarkeit
- Affected users
  - Anzahl betroffener Benutzer
- Discoverability
  - Grad der Erkennbarkeit



DREAD: Example Ratings

Rating	High (3)	Medium (2)	Low (1)
Damage potential	Attacker can subvert security system, get full trust authorization, run as admin, upload content	Leaking sensitive information	Leaking trivial information
Reproducibility	Attack can be reproduced every time and does not require a timing window	Attack can be reproduced but only with timing window and particular race situation	Attack is difficult to reproduce, even with knowledge of the security hole
Exploitability	A novice programmer could make the attack in a short time	A skilled programmer could make the attack, then repeat the steps.	Attack requires extremely skilled person and in-depth knowledge every time to exploit
Affected users	All users, default configuration, key customers	Some users, non-default configuration	Small percentage of users, obscure feature; affects anonymous users
Discoverability	Published information explains the attack; Vulnerability is in most commonly used feature	Vulnerability is in seldom-used part of product	The bug is obscure; unlikely that users will work out damage potential

 SECURITY INNOVATION

# 10d) Secure by Design

- Prinzipien und Praktiken Software zu erstellen, die sicher ist
  - ▶ Minimierung der Oberfläche für Attacken (attack surface)
    - alle Punkte, die attackiert werden können
  - ▶ Defense in Depth
    - mehrere Hindernisse hintereinander
  - ▶ Principle of least privilege (PoLP)
    - gerade soviel Berechtigung wie nötig
    - ist oft schwierig zu bestimmen, Definition von Rollen helfen beim Design
  - ▶ Simplicity (KISS)
    - einfachere Systeme sind einfacher zu sichern
  - ▶ Secure by default
    - alle Security Mechanismen sind default auf „an“
  - ▶ Default deny
    - Zugriffe nur auf ... anstatt auf alles Zugriff außer ...

# 10d) Secure by Design

- Prinzipien und Praktiken Software zu erstellen, die sicher ist
  - ▶ Validating input
    - alle Eingaben prüfen, egal woher sie kommen (User, Schnittstelle, Kommunikation)
  - ▶ Secure the weakest link
    - eine Kette ist nur so stark wie das schwächste Glied in der Kette
    - Angreifer suchen zuerst nach solchen Schwachstellen
  - ▶ Security must be usable
    - dürfen die Benutzbarkeit nicht herabsetzen, sonst gehen Benutzer andere (unsichere) Wege
  - ▶ Fail securely
    - hohe Sicherheit auch im Fehlerfall (und Fehler passieren)

# 10e) Kryptografie

- Verschlüsselung
  - ▶ symetrisch (geheimer Schlüssel)
    - benutzt einen Schlüssel zum Verschlüsseln
  - ▶ asymetrisch (öffentlicher Schlüssel)
    - benutzt 2 Schlüssel zum Verschlüsseln (1 x geheim, 1 x öffentlich)
  - ▶ cryptographic hash functions
    - ein fixer Wert, der einen beliebigen Input repräsentiert
    - es gibt verschiedenste Algorithmen
      - MD5, SHA-256, SHA-512
    - kann zum Vergleich von 2 Dateien, Texten, Datenblöcken benutzt werden
    - werden genutzt für
      - digitale Signaturen
      - HTTPS Zertifikate
      - SSL/TLS oder SSH Protokolle

# 10f) Identity und Access Management

## ■ Identity und Access Management (IAM)

- ▶ 2 der fundamentalen Konzepte von IAM sind
  - Authentication (Authentifizierung)
    - ist jemand der, den er vorgibt zu sein
    - Validierung der Identität
    - Multi-factor authentication (MFA)
      - zusätzlicher Level von Sicherheit
      - Sicherheit wird über mehrere sog. Faktoren sichergestellt
      - eine Variante ist 2FA (two-factor identification)
      - Typen von Faktoren
        - Knowledge factor (etwas, was der Benutzer wissen sollte oder weiß)
        - Possession factor (etwas, was die Person hat, wie ein Mobile Phone, das einen Code empfangen kann)
        - Inherence factor (Nutzung von Finger- oder Augen-Scannern oder anderer biometrischer Daten)

# 10f) Identity und Access Management

- Identity und Access Management (IAM)
  - ▶ 2 der fundamentalen Konzepte von IAM sind
    - Authorization
      - was einem Benutzer erlaubt ist zu tun
      - Zugriffssteuerung auf Systeme oder Teile von Systemen
      - die Granularität von Zgriffsrechten ist ein wichtiger Aspekt in Software Architekturen
        - zu klein = zu grobe Rechtevergabe
        - zu groß = zu komplizierte Vergabe und Anwendung
  - ▶ Passwörter
    - im Klartext (unbedingt vermeiden !)
    - verschlüsselt (Verschlüsselungsalgorithmus muss geschützt sein)
    - hashed (dictionary attacks versuchen Verzeichnisse zu hacken, rainbow tables dienen als Vergleich)

# 10f) Identity und Access Management

## ■ Identity und Access Management (IAM)

- ▶ Domain authentication
  - Benutzung von sog. Domain Controllern zusammen mit einem Directory Service wie Active Directory (AD)
- ▶ zentralisierter Identity Provider (IdP)
  - manche Anwendungen müssen mit API's interagieren, die nicht in der gleichen Domäne sind, manchmal sogar weltweit öffentlich sein müssen
  - Domain Controller können das nicht leisten
  - Zugriffsrechte über Domänengrenzen hinweg können mit IdP's realisiert werden
  - IdP's können auch die Verantwortung der Authentifizierung von der Anwendung nehmen und selbst verwalten
  - all Funktionen wie Benutzerregistrierung, Passwort Policies, Passwort Änderungen und die Behandlung von ausgesperrten Benutzern kann von IdP's übernommen werden
  - Einmal implementiert, kann die Funktionalität von allen Anwendung genutzt werden
  - Erniedrigt die Komplexität des Gesamtsystems

# 10f) Identity und Access Management

- Identity und Access Management (IAM)
  - ▶ OAuth 2/OpenID Connect (OIDC)
    - offener Standard für Authentifizierung
    - unterstützt eine Anwendung beim Zugriff auf Resourcen anderer Anwendungen und gewährt Zugriff auf eigene Resourcen für andere Anwendungen
    - OIDC ist eine Identity Layer oberhalb von OAuth 2 und wird zur Verifizierung von Benutzeridentitäten benutzt
    - OAuth 2 definiert 4 Rollen
      - Resource owner : Person oder Anwendung, der die Resourcen gehören, auf die Zugriff benötigt wird
      - Resource server : der Server, der die Resourcen hosted
      - Client : die Anwendung, die die Resource nutzen will
      - Authorization server : Server, der den Client für die Nutzung der Resource autorisiert
    - Authorization Server und Resource Server können der gleiche Server sein

# 10g) Einige Risiken von Webanwendungen

- Das Open Web Application Security Project (OWASP) stellt auf Ihrer Homepage (<https://www.owasp.org>) jede Menge nützlicher Informationen zur Verfügung – unter anderem eine Liste der Top Sicherheitsrisiken von Webanwendungen
- Risiken von Webanwendungen
  - ▶ Injection
    - nicht vertrauenswürdige Daten werden einem Interpreter geschickt und unvorhergesehene Befehle werden ausgeführt
    - bekanntestes Beispiel : SQL injection (SQLi)
      - SQL Befehle in Benutzerdaten versteckt, die dann aus Versehen am Server ausgeführt werden
      - ein Web Applications Firewall (WAF) kann das anhand von Signaturen größtenteils (nicht alle) erkennen und ausfiltern
      - Methoden wie Validating input, SQL parameter und das Prinzip “least privilege” kann die Gefahr reduzieren

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Broken authentication
    - eine Schwachstelle im Authentifizierungs- oder Sessionmanagement kann die Security eines Software Systems gefährden
    - Angreifer versuchen zuerst manuell Schwachstellen zu finden, um sie anschließend mit Toolunterstützung auszunutzen
    - hashing, multi-factor Authetifizierung, Secure by default und Installationen ohne Nutzung von Defaultrechten helfen das Risiko dieser Art von Attacken zu minimieren
    - Passwort Policies und das zyklische Erneuern von Passwörter erhöhen die Sicherheit
    - Anwendung sollten gezielte Log-outs unterstützen, um den Einfall von Hackern bei Timeouts von Sessions noch nicht abgemeldeter Benutzer zu verhindern
    - Session ID's sollten nicht in der URL erkennbar sein und nicht zwischen Sessions rotieren
    - Informationen wie Passwörter, Tokens, Session ID's und andere Berechtigungen sollten immer über sichere Verbindungen ausgetauscht werden

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Sensitive data exposure
    - nicht richtig abgesicherte vertrauliche Informationen wie Sozialversicherungs- oder Kreditkarten-Nummer, Berechtigungen u.ä.
    - Sensitive Daten nur speichern, wenn notwendig und sobald als möglich entfernen
    - Sensitive Daten, die längerfristig gespeichert sind, sollten verschlüsselt sein (auch alle Backups)
    - Sensitive Daten, die übertragen werden, müssen während der Übertragung verschlüsselt sein
    - Starke und moderne Algorithmen sollten zusammen mit einem entsprechenden Schlüsselmanagement benutzt werden
    - Browser Direktiven und Header können die Sicherheit erhöhen
      - vermeiden von Cachen von sensitiven Daten

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ XML external entity (XXE) attack
    - greift Parser, die XML parsen an
    - wenn XML Input Referenzen auf externe Ressourcen enthält und der Parser nicht entsprechend konfiguriert, kann dies zu Sicherheitslücken führen
    - Denial of Service (DoS) und Server-Side Request Forgery (SSRF) (Verfälschung) können mit einer XXE Attacke ausgelöst werden
    - eine der effektivsten Methoden XXE Attacken zu vermeiden, ist die Wahl anderer Datenformate wie JSON
    - komplette Deaktivierung von document type definitions (DTD) ist auch effektiv
    - ist das nicht möglich, müssen die Nutzung externer Entities oder Entity expansion verhindert werden
    - die letzten gültigen Security patches und fixes sollten eingespielt sein

# 10g) Einige Risiken von Webanwendungen

## ■ Risiken von Webanwendungen

- ▶ XML external entity (XXE) attack, Beispiel (billion laughs attack oder XML bomb)

```
<?xml version=„1.0“?>
<!DOCTYPE lolz [
  <!ENTITY lol „lol“>
  <!ELEMENT lolz (#PCDATA)>
  <!ENTITY lol1 „&lol;&lol; &lol;&lol; &lol;&lol; &lol;&lol; &lol;&lol;“>
  <!ENTITY lol2 „&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;“>
  <!ENTITY lol3 „&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;“>
  <!ENTITY lol4 „&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;“>
  <!ENTITY lol5 „&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;“>
  <!ENTITY lol6 „&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;“>
  <!ENTITY lol7 „&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;“>
  <!ENTITY lol8 „&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;“>
  <!ENTITY lol9 „&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;“>
]>
<lolz&lol9;</lolz>
```

- ▶ wird zu einer Milliarde Entitäten aufgebläht und sprengt alle Puffer und erzeugt damit einen DoS

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Broken access control
    - eine weit verbreitete Sicherheitsbedrohung
    - kann manuell oder mit Tools entdeckt werden und kann zur Benutzung von nicht autorisierten Benutzungsrechten führen
    - auch wenn die Benutzeroberfläche in allen Punkten sicher ist, können Angreifer versuchen die Server zu attackieren, indem sie die URL, den Anwendungsstatus oder Tokens ändern oder Requests gezielt verändern, um Zugriff auf nicht autorisierte Funktionen zu bekommen
    - Auf Client- und Serverseite zu prüfen
    - Zugriffstokens so schnell wie möglich ungültig machen
    - Deny by Default Ansatz anwenden
    - Access Control Fehler protokollieren und bei erhöhter Anzahl unverzüglich und automatisch melden
    - Softwaretests sollten solche Attacken beinhalten

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Security misconfiguration
    - oft der Fall
    - je größer und komplexer die Anwendung, um so höher die Wahrscheinlichkeit
    - die Anwendung muss vor der Installation auf Security geprüft sein
    - Default Werte vermeiden
    - Teile von Anwendungen, die nicht benötigt oder nicht benutzt werden, sollten nicht installiert sein
      - Beispiele : Accounts, Berechtigungen, Ports, Services
    - Alle Passwörter von Default Accounts sollten geändert oder die Accounts gelöscht sein
    - Viele Anwendungen nutzen eine Vielzahl von Frameworks und Tools
      - alle diese Hilfsmittel müssen beherrscht werden und richtig konfiguriert sein
    - Die Fehlerbehandlung sollte nicht zu detailliert sein und vor allem keine Informationen ausgeben, die zu einem Angriff genutzt werden können (z.B. detaillierter Stacktrace)
    - Aktuelle Patches und Fixes sind eingespielt
      - oft sind Probleme in Patches gelöst, aber nicht installiert

# 10g) Einige Risiken von Webanwendungen

## ■ Risiken von Webanwendungen

- ▶ Cross-site scripting (XSS)
  - erlaubt Angreifern das ausführen von Scripts im Browser
  - Scripts können User Sessions übernehmen, Content ersetzen oder Benutzer umlenken (redirect)
  - sehr geläufige Methode
  - 3 Haupttypen
    - Reflected XSS
      - eine Anwendung übernimmt nicht vertrauenswürdige Daten und schickt sie ohne Überprüfung an den Browser
    - Stored XSS
      - nicht überprüfte Daten werden zur späteren Anzeige gespeichert
    - DOM XSS
      - Daten oder Code kontrolliert durch einen Angreifer wird dynamisch durch ein Script eingefügt
  - Überprüfungen vermeiden nicht validierte und unsichere Eingaben
  - Server erkennen (untrusted HTTP requests) und beenden entsprechende Attacken

# 10g) Einige Risiken von Webanwendungen

## ■ Risiken von Webanwendungen

- ▶ Unsecure deserialization
  - Attacken gegen serialized Objekte
  - kann zur Ausführung von nicht gewolltem Code führen
  - Datenstrukturen mit Zugriffsrechten können verändert werden
  - Serialized Objekte nur von vertrauenswürdigen Quellen akzeptieren
  - alle Exceptions loggen

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Komponenten nutzen, die bekannt unsicher sind
    - externe Bibliotheken oder Systeme
    - auf Wartung und Support achten
  - ▶ nicht ausreichendes Logging und Monitoring
    - ein wichtiges Risiko bei der Security Betrachtung
    - Angreifer wollen solange wie möglich unentdeckt bleiben
    - gutes Logging und Monitoring verhindert das
    - typische Fragestellung : wer/was/wann
    - kontrolliertes Log-Management notwendig
      - Auswertungen
      - Information an entsprechende Verantwortliche
    - auch Logs müssen gesichert sein
      - Angreifer könnten Spuren verwischen oder Informationen abgreifen

# 10g) Einige Risiken von Webanwendungen

- Risiken von Webanwendungen
  - ▶ Unvalidated redirects and forwards
    - Weiterleitungen können zum Weiterleiten auf unsichere Webseiten benutzt werden
    - Redirects und Forwards nicht nutzen
    - Alle Redirects sollten über eine Seite gehen, die den Benutzer darüber informiert – er muss die Weiterleitung bestätigen

# 11) Dokumentation

1. Einführung und Ziele
2. Randbedingungen
3. Kontextabgrenzung
4. Lösungsstrategie
5. Bausteinsicht
6. Laufzeitsicht
7. Verteilungssicht
8. Querschnittliche Konzepte
9. Entwurfsentscheidungen
10. Qualitätsszenarien
11. Risiken und technische Schulden
12. Glossar



# 11) Dokumentation

## Einführung und Ziele

- Beschreibt die wesentliche Anforderungen und treibenden Kräfte, die Softwarearchitekten und Entwicklungsteams berücksichtigen müssen.  
Dazu gehören die
  - zugrunde liegenden Geschäftsziele, wesentliche Aufgabenstellung und essenzielle fachliche Anforderungen an das System
  - Qualitätsziele für die Architektur
  - relevante Stakeholder und deren Erwartungshaltung



# 11) Dokumentation

## Einführung und Ziele

Geschäftsziele

Aufgabenstellungen

Fachl.  
Anforderungen



- Inhalt
  - Kurzbeschreibung der Geschäftsziele und der fachlichen Aufgabenstellung, treibenden Kräfte, Extrakt (oder Abstract) der Anforderungen. Verweis auf vielleicht vorhandene Anforderungsdokumente (mit Versionsbezeichnungen und Ablageorten).
- Motivation
  - Aus Sicht der späteren Nutzer ist die Unterstützung einer fachlichen Aufgabe oder Verbesserung der Qualität der eigentliche Beweggrund, ein neues System zu schaffen oder ein bestehendes zu modifizieren.
- Form
  - Kurze textuelle Beschreibung, eventuell in tabellarischer Use-Case Form. Sofern vorhanden sollte die Aufgabenstellung Verweise auf die entsprechenden Anforderungsdokumente enthalten.
  - Halten Sie diese Auszüge so knapp wie möglich und wägen Sie Lesbarkeit und Redundanzfreiheit gegeneinander ab.

# 11) Dokumentation

## Einführung und Ziele

### Qualitätsziele



- Inhalt
  - Die Top-3 bis Top-5 der Qualitätsziele für die Architektur, deren Erfüllung oder Einhaltung den maßgeblichen Stakeholdern besonders wichtig sind. Gemeint sind hier wirklich Qualitätsziele, die nicht unbedingt mit den Zielen des Projekts übereinstimmen. Beachten Sie den Unterschied.
- Motivation
  - Weil Qualitätsziele grundlegende Architekturentscheidungen oft maßgeblich beeinflussen, sollten Sie die für Ihre Stakeholder relevanten Qualitätsziele kennen, möglichst konkret und operationalisierbar.
  - Wenn Sie als Architekt nicht wissen, woran Ihre Arbeit gemessen wird, ....
- Form
  - Tabellarische Darstellung der Qualitätsziele mit möglichst konkreten Szenarien, geordnet nach Prioritäten.

# 11) Dokumentation

## Einführung und Ziele

### Stakeholder



- Inhalt
  - Expliziter Überblick über die Stakeholder des Systems, d.h. über alle Personen, Rollen oder Organisationen, die die Architektur kennen sollten oder von der Architektur überzeugt werden müssen, mit Architektur oder Code arbeiten (z.B. Schnittstellen nutzen), Dokumentation der Architektur für ihre eigene Arbeit benötigen, Entscheidungen über das System und dessen Entwicklung treffen.
- Motivation
  - Sie sollten die Projektbeteiligten und -betroffenen kennen, sonst erleben Sie später im Entwicklungsprozess Überraschungen. Diese Stakeholder bestimmen unter anderem Umfang und Detaillierungsgrad der von Ihnen zu leistenden Arbeit und Ergebnisse.
- Form
  - Tabelle mit Rollen- oder Personennamen, sowie deren Erwartungshaltung bezüglich der Architektur und deren Dokumentation.

Rolle	Kontakt	Erwartungshaltung
<Rolle-1>	<Kontakt-1>	<Erwartung-1>
<Rolle-2>	<Kontakt-2>	<Erwartung-2>



# 11) Dokumentation

## Randbedingungen



- Inhalt
  - Fesseln und Vorgaben, die ihre Freiheiten bezüglich Entwurf, Implementierung oder Ihres Entwicklungsprozesses einschränken. Diese Randbedingungen gelten manchmal organisations- oder firmenweit über die Grenzen einzelner Systeme hinweg.
- Motivation
  - Als Architekt sollten Sie explizit wissen, wo Ihre Freiheitsgrade bezüglich Entwurfsentscheidungen liegen und wo Sie Randbedingungen beachten müssen. Sie können Randbedingungen vielleicht noch verhandeln, zunächst sind sie aber da.
- Form
  - Einfache Tabellen der Randbedingungen mit Erläuterungen. Bei Bedarf unterscheiden Sie technische, organisatorische und politische Randbedingungen oder übergreifende Konventionen (beispielsweise Programmier- oder Versionierungsrichtlinien, Dokumentation- oder Namenskonvention).

# 11) Dokumentation

## Kontextabgrenzung



- Inhalt
  - Die Kontextabgrenzung grenzt das System von allen Kommunikationspartnern (Nachbarsystemen und Benutzerrollen) ab. Sie legt damit die externen Schnittstellen fest.
  - Differenzieren Sie fachlichen Kontext (fachliche Ein- und Ausgaben) und technischen Kontext (Kanäle, Protokolle, Hardware), falls nötig.
- Motivation
  - Die fachlichen und technischen Schnittstellen zu Kommunikationspartnern gehören zu den kritischsten Aspekten eines Systems. Stellen Sie sicher, dass Sie diese komplett verstanden haben.
- Form
  - Verschiedene Optionen:
    - Diverse Kontextdiagramme
    - Listen von Kommunikationspartnern mit deren Schnittstellen

# 11) Dokumentation

## Kontextabgrenzung fachlich



- Inhalt
  - Festlegung aller Kommunikationspartner (Nutzer, IT-Systeme, ...) mit Erklärung der fachlichen Ein- und Ausgabedaten oder Schnittstellen. Zusätzlich bei Bedarf fachliche Datenformate oder Protokolle der Kommunikation mit den Nachbarsystemen.
- Motivation
  - Alle Beteiligten müssen verstehen, welche fachlichen Informationen mit der Umgebung ausgetauscht werden.
- Form
  - Alle Diagrammarten, die das System als Black Box darstellen und die fachlichen Schnittstellen zu den Nachbarn beschreiben.
  - Alternativ oder ergänzend können Sie eine Tabelle verwenden. Der Titel gibt den Namen Ihres Systems wieder; die drei Spalten sind: Kommunikationspartner, Eingabe, Ausgabe.  
*<Diagramm und/oder Tabelle>*  
*<optional: Erläuterung der externen fachlichen Schnittstellen>*

# 11) Dokumentation

## Kontextabgrenzung technisch



- Inhalt
  - Technische Schnittstellen (Kanäle, Übertragungsmedien) zwischen dem System und seiner Umwelt. Zusätzlich eine Erklärung (mapping), welche fachlichen Ein- und Ausgaben über welche technischen Kanäle fließen.
- Motivation
  - Viele Stakeholder treffen Architekturentscheidungen auf Basis der technischen Schnittstellen des Systems zu seinem Kontext.
  - Insbesondere Infrastruktur- oder Hardwareentwickler entscheiden auch über diese technischen Schnittstellen.
- Form
  - Beispielsweise UML Deployment-Diagramme mit den Kanälen zu Nachbarsystemen, begleitet von einer Tabelle, die Kanäle auf Ein-/Ausgaben abbildet.  
*<Diagramm oder Tabelle>*  
*<optional: Erläuterung der externen technischen Schnittstellen>*  
*<Mapping fachliche auf technische Schnittstellen>*

# 11) Dokumentation

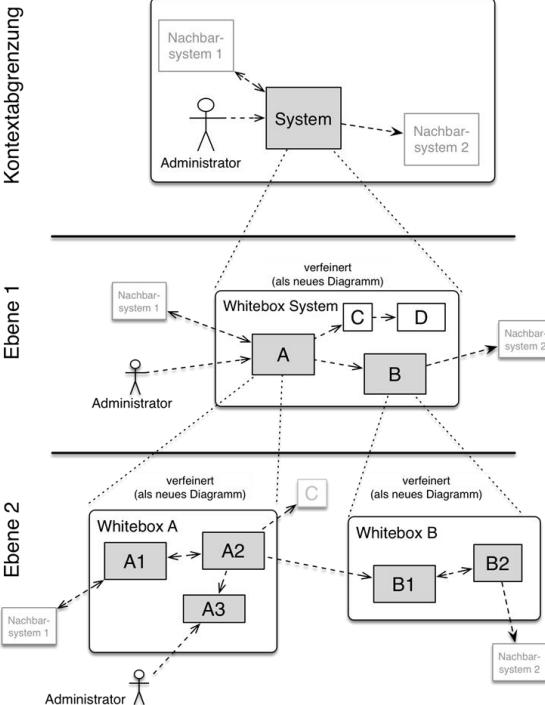
## Lösungsstrategie



- Inhalt
  - Kurzer Überblick über die grundlegenden Entscheidungen und Lösungsansätze, die Entwurf und Implementierung des Systems prägen. Hierzu gehören:
    - Technologieentscheidungen
    - Entscheidungen über die Top-Level-Zerlegung des Systems, beispielsweise die Verwendung prägender Entwurfs- oder Architekturmuster
    - Entscheidungen zur Erreichung der wichtigsten Qualitätsanforderungen
    - relevante organisatorische Entscheidungen (Entwicklungsprozesse oder Delegation bestimmter Aufgaben an andere Stakeholder).
- Motivation
  - Diese Entscheidungen bilden wesentliche „Eckpfeiler“ der Architektur. Von ihnen hängen meistens viele weitere Entscheidungen oder Implementierungsregeln ab.
- Form
  - Fassen Sie die zentralen Entwurfsentscheidungen kurz zusammen. Motivieren Sie ausgehend von Aufgabenstellung, Qualitätszielen und Randbedingungen, was Sie entschieden haben und warum Sie so entschieden haben.

# 11) Dokumentation

## Bausteinsicht



- **Inhalt**

- Diese Sicht zeigt die statische Zerlegung des Systems in Bausteine (Module, Komponenten, Subsysteme, Klassen, Interfaces, Pakete, Bibliotheken, Frameworks, Schichten, Partitionen, Tiers, Funktionen, Makros, Operationen, Datenstrukturen...) sowie deren Beziehungen.
- Diese Sicht sollte in jeder Architekturdokumentation vorhanden sein. In der Analogie zum Hausbau bildet die Bausteinsicht den Grundrissplan.

- **Motivation**

- Behalten Sie den Überblick über den Quellcode, indem Sie die statische Struktur des Systems durch Abstraktion verständlich machen.
- Damit ermöglichen Sie Kommunikation auf abstrakterer Ebene, ohne zu viele Implementierungsdetails offenlegen zu müssen.

- **Form**

- Die Bausteinsicht ist eine hierarchische Sammlung von Blackboxen und Whiteboxen (siehe Abbildung unten) und deren Beschreibungen.

# 11) Dokumentation

## Laufzeitsicht



- Inhalt
  - Diese Sicht erklärt konkrete Abläufe und Beziehungen zwischen Bausteinen in Form von Szenarien aus folgenden Bereichen:
    - Wichtige Abläufe oder Features
    - Interaktionen an kritischen externen Schnittstellen
    - Betrieb und Administration: Inbetriebnahme, Start, Stop.
    - Fehler- und Ausnahmeszenarien
- Motivation
  - Verständnis der Aufgabenerfüllung und Kommunikation von Bausteinen des Systems
  - Bessere Kommunikation mit Stakeholdern (verständlicher als z.B. Bausteinsicht, Verteilungssicht)
- Form
  - Folgende Ausdrucksmöglichkeiten
    - Nummerierte Schrittfolgen oder Aufzählungen in Umgangssprache
    - Aktivitäts- oder Flussdiagramme
    - Sequenzdiagramme
    - BPMN oder EPKs (Ereignis-Prozessketten)
    - Zustandsautomaten
    - ...

Anmerkung: Kriterium für die Auswahl der möglichen Szenarien (d.h. Abläufe) des Systems ist deren Architekturrelevanz. Es geht nicht darum, möglichst viele Abläufe darzustellen, sondern eine angemessene Auswahl zu dokumentieren.

# 11) Dokumentation

## Verteilungssicht



- Inhalt
  - die technische Infrastruktur, auf der Ihr System ausgeführt (Standorte, Umgebungen, Rechnern, Prozessoren, Kanälen und Netztoplogien, ...)
  - die Abbildung von (Software-)Bausteinen auf diese Infrastruktur
  - Darstellung von relevanten Entwicklung-/Test-/Produktionsumgebungen
  - Verteilung darstellen (Rechner, Prozessor, Server oder Container)
  - Schwerpunkt auf Softwaresicht
- Motivation
  - Die Infrastruktur sollte allen bekannt sein
  - Kombination von „Black-box“ mit weiteren Detailsichten
- Form
  - Diagramme
  - Sichten

# 11) Dokumentation

## Querschnittliche Konzepte



- Inhalt
  - übergreifende, prinzipielle Regelungen und Lösungsansätze
  - Themen
    - fachliche Modelle,
    - eingesetzte Architektur- oder Entwurfsmuster,
    - Regeln für den konkreten Einsatz von Technologien,
    - prinzipielle, meist technische, Festlegungen übergreifender Art,
    - Implementierungsregeln



### Motivation

- Grundlage für konzeptionelle Integrität (Konsistenz, Homogenität) der Architektur
- wesentliche Grundlage für die innere Qualität Ihrer Systeme
- Platz zum behandeln spezieller Themen (wie z.B. "Sicherheit")
- Form
  - Konzeptpapiere mit beliebiger Gliederung, Modelle/Szenarien
  - Verweise auf Nutzung von Standardframeworks (z.B. Hibernate als Object/Relational Mapper)

# 11) Dokumentation

## Entwurfsentscheidungen



- Inhalt
  - Wichtige Architektur- oder Entwurfsentscheidungen mit Begründungen
  - Auswahl einer von mehreren Alternativen unter vorgegebenen Kriterien
  - Vermeiden Sie Redundanz
- Motivation
  - Stakeholder des Systems sollten wichtige Entscheidungen verstehen und nachvollziehen können.
- Form.
  - Liste oder Tabelle, nach Wichtigkeit und Tragweite der Entscheidungen geordnet
  - einzelne Unterkapitel je Entscheidung
  - ADR (Architecture Decision Record) für jede wichtige Entscheidung

# 11) Dokumentation

## Qualitätsszenarien



- Inhalt
  - Konkretisierung der Qualitätsanforderungen durch (Qualitäts-)Szenarien
  - Diese Szenarien beschreiben, was beim Eintreffen eines Ereignisses auf ein System in bestimmten Situationen geschieht
    - Nutzungsszenarien oder auch Anwendungs- oder Anwendungsfall-szenarien (Reaktionen zur Laufzeit, Effizienz oder Performance)
    - Änderungsszenarien (Modifikation des Systems oder seiner unmittelbaren Umgebung)
- Motivation
  - Szenarien operationalisieren Qualitätsanforderungen und machen deren Erfüllung mess- oder entscheidbar
  - Qualitätsszenarien müssen diskutierbar und nachprüfbar sein
- Form.
  - Entweder tabellarisch oder als Freitext

# 11) Dokumentation

## Risiken und technische Schulden



- Inhalt
  - Eine nach Prioritäten geordnete Liste der erkannten Architekturrisiken und/oder technischen Schulden
- Motivation
  - "Risikomanagement ist Projektmanagement für Erwachsene" (Tim Lister, Atlantic Systems Guild.)
  - Unter diesem Motto sollten Sie Architekturrisiken und/oder technische Schulden gezielt ermitteln, bewerten und Ihren Management-Stakeholdern (z.B. Projektleitung, Product-Owner) transparent machen
- Form
  - Liste oder Tabelle von Risiko und/oder technischen Schulden, eventuell mit vorgeschlagenen Maßnahmen zur Risikovermeidung, Risikominimierung oder dem Abbau der technischen Schulden.

# 11) Dokumentation

## Glossar



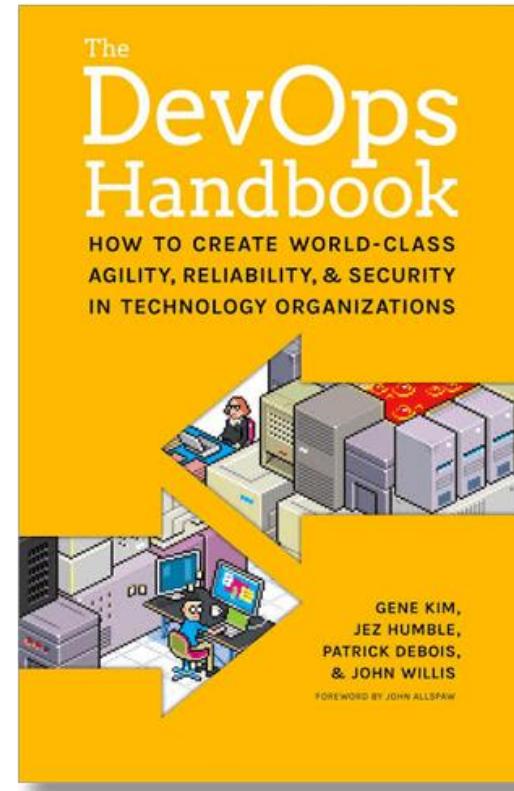
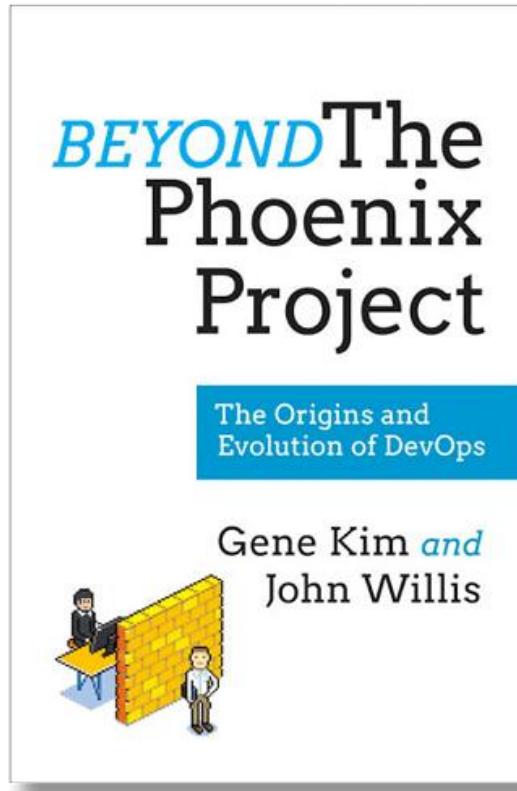
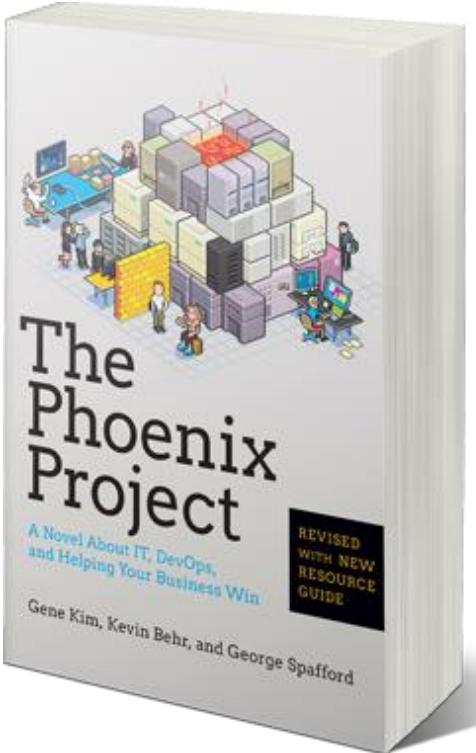
- Inhalt
  - Die wesentlichen fachlichen und technischen Begriffe, die Stakeholder im Zusammenhang mit dem System verwenden
  - Nutzen Sie das Glossar ebenfalls als Übersetzungsreferenz, falls Sie in mehrsprachigen Teams arbeiten
- Motivation
  - Sie sollten relevante Begriffe klar definieren, so dass alle Beteiligten
    - diese Begriffe identisch verstehen, und
    - vermeiden, mehrere Begriffe für die gleiche Sache zu haben
- Form
  - Zweispaltige Tabelle mit <Begriff> und <Definition>
  - Eventuell weitere Spalten mit Übersetzungen, falls notwendig

## 12) DevOps und Softwarearchitektur

- DevOps ist die Kombination von kulturellen Werten, Praktiken und Tools für eine schnelle Softwareentwicklung
- Software Architekten sollten damit vertraut sein
- DevOps Praktiken kann Architekturen beeinflussen
- Wir betrachten
  - a) DevOps
  - b) DevOps Toolkette
  - c) DevOps Praktiken
  - d) DevOps und Architektur
  - e) DevOps und Cloud

## 12a) DevOps

- DevOps ist ein Satz von Tools, Praktiken und zugehöriger Kultur, um Softwareentwicklerteams mit Betriebsteams während des ganzen Lebenszyklus eines Software Systems zusammen zu bringen
- Weiterführende Literatur



## 12a) DevOps

- DevOps verbindet die Collaboration zwischen Teams mit der Automatisierung von Prozessen, um das gemeinsame Ziel, zusätzlichen Mehrwert für die User Experience zu erzeugen, zu erreichen indem Software schnell und mit hoher Qualität erstellt und geliefert wird
- DevOps setzt eine kulturelle Veränderung innerhalb von Organisationen voraus, im Zusammenspiel mit der Nutzung neuer Technologien
- *Shift left* ist ein bekannter Ausdruck bei DevOps und meint die Idee Aufgaben früher im Lebenszyklus durchzuführen oder sie nach links auf der Zeitschiene zu schieben
- DevOps hat sich die *shift left* Mentalität zu eigen gemacht und versucht in jeder Phase Aktivitäten nach links zu schieben
- z.B. sollte die Betriebsmannschaft *shift left* von Anfang an in alle Entwicklungsschritte eingebunden sein
- Weitere Beispiele für *shift left* sind Continuous Integration, bei dem die Integration und das Erstellen von Changes nach links geschoben wird und Continuous Delivery, bei dem das Ausrollen von Änderungen in Produktion viel schneller beim Endkunden ankommt

# 12a) DevOps

## ■ CALMS

- ▶ repräsentiert die Kernwerte von DevOps
  - Culture
  - Automation
  - Lean
  - Measurement
  - Sharing

# 12a) DevOps

## ■ CALMS

### ▶ Culture

- im Inneren ist DevOps Kultur und Philosophie
- die Änderung der Kultur, das Aufbrechen von Barrieren zwischen Teams in einer Organisation ist die Grundlage
- DevOps führt sog. cross-funktionale Teams ein und bringt Mitarbeiter mit unterschiedlichen Skills zusammen
- die DevOps Kultur schätzt Lernen (auch aus Fehlern) und jede Art von Wissen für Verbesserungen zu nutzen
- Verantwortlichkeit ist auch Teil der DevOps Kultur. Wenn Fehler gemacht werden, übernehmen Teams die Verantwortung und arbeiten an einer Lösung, ohne zuerst Schuldige zu suchen
- Qualität steht immer im Fokus von DevOps und deshalb Teil der Kultur
- Mitarbeiter werden zu eigenen Entscheidungen befähigt und ermächtigt und in Entscheidungen einbezogen

# 12a) DevOps

## ■ CALMS

### ▶ Automation

- Automatisierung bei DevOps bedeutet, daß alle manuellen Schritte automatisiert werden und damit nachvollziehbar immer wieder gleich ablaufen können (wiederholbar)
- automatische Prozesse garantieren höhere Konsistenz und Genauigkeit, im Vergleich zu manuellen Prozessen
- automatische Prozesse können zu jeder Zeit ablaufen (auch zu Zeiten an denen niemand anwesend ist)
  - kann für eine bessere Auslastung von Ressourcen sorgen
- automatische Prozesse laufen i.a. schneller ab und sind weniger fehleranfällig
- automatische Prozesse liefern schneller Feedback über Probleme oder notwendige Änderungen

# 12a) DevOps

## ■ CALMS

### ▶ Lean

- Lean Software Development kommt vom Lean Manufacturing und hat einige “best practices” von dort übernommen
- das Ziel ist die Optimierung von Prozessen und die Minimierung von unnötiger Arbeit
- die 7 Lean Prinzipien
  - eliminate waste
  - build quality in
  - create knowledge
  - defer commitment
  - deliver fast
  - respect people
  - optimize the whole
- passen gut mit den Zielen von DevOps zusammen

# 12a) DevOps

## ■ CALMS

### ▶ Measurement

- Meßbarkeit ist wichtig, denn ohne Meßbarkeit ist kein Fortschritt nachweisbar
- Architekten sind von Maßzahlen abhängig, ansonsten hätten sie nur ihr Bauchgefühl übrig ☺
- Maßzahlen oder gemessene Werte müssen transparent für alle Beteiligten verfügbar sein

### ▶ Sharing

- alle Ressourcen wie Tools oder auch Methoden, Ideen, Wissen und Erfahrungen müssen mit allen anderen geteilt werden
- nur so kann man von Fehlern oder von anderen lernen
- meistens der erste Schritt zur Änderung der Unternehmenskultur

# 12a) DevOps

## ■ Warum DevOps ?

- ▶ Als Software Architekt will man immer Software schneller, öfter und mit wenigen Fehler ausliefern
- ▶ Andere Ansätze als DevOps haben nicht so den Fokus auf Continuous Delivery und benötigen deshalb länger bis der Wert der erstellten Lösungen beim Kunden ankommt
- ▶ Wenn der Betrieb nicht in die Entwicklung integriert wird, dann gibt es oft zusätzliche Wartezeiten bei der Bestellung und Installation der notwendigen Umgebungen
  - die Software ist schon fertig, aber die Umgebung steht noch nicht
  - die Bereitstellung der Umgebung ist nicht automatisiert
- ▶ oft werden Hardware Ressourcen zu groß bestellt, weil das Entwicklungsteam nur einmal den hinderlichen Bestellprozess durchlaufen will (ein späteres Upgrade will man sich sparen)
- ▶ um heute wettbewerbsfähig zu bleiben, müssen Lösungen schnell am Markt verfügbar sein, das gilt auch (oder ganz speziell) für Softwarelösungen
- ▶ das geht nur, wenn man bereit ist, dem Kunden Teillösungen, die später erweitert werden, anzubieten
- ▶ in das Design der Erweiterungen wird der Kunde eingebunden

# 12a) DevOps

## ■ Warum DevOps ?

- ▶ die Einbindung des Kunden in den Entwicklungsprozess sorgt für besseres Verständnis des Anforderungen und für eine höhere Zufriedenheit beim Kunden
- ▶ Kunden haben heute weniger Verständnis für Ausfälle als früher
- ▶ Deshalb wird es immer wichtiger, daß Fehler schnell entdeckt und behoben und neue Versionen oder Releases sofort automatisiert ausgerollt werden können
- ▶ Softwarelösungen werden immer größer und komplexer. DevOps, zusammen mit entsprechenden Prinzipien und Pattern, kann für eine Reduzierung der Komplexität sorgen
- ▶ DevOps sorgt für einen Wandel in der Firmenkultur und sorgt für die Zusammenarbeit unterschiedlichster Teams (Entwicklung, Betrieb, Security) bei der Erstellung von Kundenlösungen
- ▶ Wissen, Erfahrung und Begeisterung wird ausgetauscht – Informationen sind transparent verfügbar

# 12b) DevOps Toolchain

## ■ Der DevOps Toolchain

- ▶ Toolchains sind ein Satz von Tools, die in Kombination zur Erstellung von Lösungen verwendet werden
- ▶ Ein DevOps Toolchain ist auf die Entwicklung und das Ausrollen von Softwarelösungen fokussiert
- ▶ Software Architekten sorgen für die Auswahl und den konsistenten Einsatz von entsprechenden Tools
  - es gibt immer noch die Microsoft und Java Fraktionen mit unterschiedlichen Glaubensrichtungen
- ▶ der Betrieb einer Lösung ist in den Toolchain eingebunden
- ▶ Automatisierungsskripte oder Software sollte in den Entwicklungszyklus eingebunden sein (Sourcecontrol u.ä.), ist also Teil der Entwicklung (Infrastructure as Code)

# 12c) DevOps Praktiken

- 3 wichtige DevOps Praktiken
  - ▶ Continuous Integration (CI)
    - Source Control
      - Entwickler legen alle Entwicklungen und Änderungen so oft wie möglich in einem gemeinsamen Source Control System ab
      - Ein Art der Versionierung ist dabei unbedingt notwendig
        - Semantic Versioning : Major.Minor.Patch = größere Änderungen.Erweiterungen.Bugfixes
      - Entwickler sollten in bestimmten Zyklen ihre Änderungen bestätigen (committen)
      - dies erlaubt Konflikte früh zu erkennen
    - automatisierte Builds
      - alle Commits werden automatisch zur benutzbaren Gesamtlösung zusammengefügt und erzeugt
      - Schritte dahin könnten sein : Prüfen von Abhängigkeiten, Kompilieren von Sources, Paketierung, Erzeugen von Installern, Erzeugen/Update von Datenbank Schemas, Aufruf von Skripts zur Konfiguration, automatisierte Tests
      - die Laufzeit von Builds sollte 20 min nicht überschreiten
      - Architekten sollten für eine CI Umgebung sorgen
      - automatisierte Tests sorgen für das Erkennen von Problemen während der Builds

## 12c) DevOps Praktiken

- 3 wichtige DevOps Praktiken
  - ▶ Continuous Delivery (CD)
    - ist die Fähigkeit, Änderungen schnell, wiederholbar und nachvollziehbar auszurollen
    - Releasezyklen sind kurz und reduzieren Kosten, Risiken und Aufwände
    - ist die Fähigkeit, eine Lösung jederzeit in einem produktionsfähigen Status verteilen zu können
    - fasst CI, automatisches Testen (technisch und user acceptance) und das Ausrollen zusammen (manuell)
  - ▶ Continuous Deployment
    - CD mit zusätzlichem automatiserten Ausrollen der Lösung
    - verschiedene Staging Areas wie Test, Consolidation, Production werden unterstützt

## 12d) DevOps und Architektur

- Software Architekten müssen DevOps in ihrer Architektur berücksichtigen
  - ▶ DevOps beeinflusst die Qualitätsattribute
    - Testability
      - ganz wichtig, da hoch automatisiert
    - Maintainability
      - um Zwischenstände schnell ausrollen zu können, muß das Softwaresystem einfach wartbar sein
      - Reduzierung von Komplexität erfordert eine hohe Aufmerksamkeit
      - geringer Komplexität erlaubt kürzere Zyklen
    - Deployability
      - die verschiedenen Stages sollten sehr ähnlich aufgebaut oder gleich sein
      - Konfigurationsdaten sollten von der Anwendung getrennt vorliegen

# 12d) DevOps und Architektur

- Architektur Pattern, die DevOps erleichtern
  - ▶ Microservices
    - schmale kleine Services mit definiertem Interface
    - jeder Microservice sollte einzeln austauschbar und ausrollbar sein
    - jeder Microservice hat seinen eigenen Datenpool
    - Microservices unterstützen hohe Fehlerisolierung

## 12e) DevOps und Cloud

- viele der Vorteile von DevOps werden auf Cloudplattformen direkt unterstützt
- Cloudtypen können sein
  - ▶ Public Cloud (AWS, MS Azure)
    - hohe Verfügbarkeit und unbegrenzte Skalierbarkeit
    - Datensicherheit muss beachtet werden (Sensitive Daten)
  - ▶ Private Cloud
    - ausschließlich von einer Organisation benutzt
    - kann bei einem Provider oder im eigenen Rechenzentrum sein
    - höhere Kosten und eingeschränkte Skalierbarkeit
    - höhere Datensicherheit realisierbar

## 12e) DevOps und Cloud

- Cloudtypen können sein
  - ▶ Hybrid Cloud
    - Kombination von Public und Private Cloud
    - Nutzung der jeweiligen Vorteile
    - Übergangsphasen (Transitions) sind einfacher
    - Notfallszenarien können geplant werden

# 12a) DevOps und Cloud

- Cloudmodelle
  - ▶ Infrastructure as a Service (IaaS)
    - Betrieb von Hardware in der Cloud (gängiger Start oder Eintritt in Cloudplattformen)
  - ▶ Containers as a Service (CaaS)
    - Betrieb von virtuellen Umgebungen (VM oder Container) in der Cloud
    - Kubernetes, Docker Swarm, Apache Mesos
  - ▶ Platform as a Service (PaaS)
    - komplette Plattform zur Entwicklung und zum Ausrollen von Lösungen
    - beliebige Programmierumgebungen und –sprachen werden unterstützt
    - Kontrolle über das Betriebssystem geht verloren

# 12e) DevOps und Cloud

## ■ Cloudmodelle

- ▶ Serverless Function as a Service (FaaS)
  - ähnlich PaaS
  - Verrechnungsmodell verschieden (nach Ausführung nicht nach Hosting)
- ▶ Software as a Service (SaaS)
  - Cloud Software über das Internet (MS Email, MS Office, Salesforce)

# 13) Die Skills eines Softwarearchitekten

- Neben technischen Skills sollte jeder Architekt auch eine Reihe von Softskills besitzen
- Softskills sind für den Erfolg eines Architekten wichtiger als technische Skills
- Softskills machen den Unterschied zwischen Architekten und besonderen Architekten aus
- Wir betrachten
  - a) was sind Softskills
  - b) Kommunikation
  - c) Führung
  - d) Verhandlungsgeschick

## 13a) Was sind Softskills

- HardSkills sind konkret und können definiert und gemessen werden
- sie sind typischerweise aufgaben- oder job-spezifisch
- sie können mit Training (intern oder extern) und durch Lernen und Lesen verbessert werden
- bei Software Architekten kann das die Beherrschung einer Programmiersprache oder den geeigneten Einsatz von Frameworks beinhalten
- SoftSkills sind schwerer zu fassen, schlecht zu definieren und kaum zu messen
- SoftSkills beziehen sich mehr auf Beziehungen zu anderen, wie Führung, Kommunikation, Zuhören, Empathie, Gesprächsführung und Geduld
- Auch wenn man SoftSkills trainieren kann, sind sie doch mehr angeboren oder in der Person verwurzelt
- Software Architekten profitieren vom Besitz, der Anwendung und dem Auf- und Ausbau von SoftSkills
- SoftSkills spielen eine große Rolle bei der Arbeit von Architekten
- die wichtigsten Softskills werden wir betrachten

## 13b) Kommunikation

- Kommunikationsfähigkeit ist wahrscheinlich eine der wichtigsten Fähigkeiten eines Architekten
- Stakeholdern die Architektur oder Architekturvorschläge vermitteln zu können, ist überlebensnotwendig
- um etwas kommunizieren zu können, muß man die verstehen können und das Zielpublikum genau kennen
- dies hilft bei der Auswahl der geeigneten Hilfsmittel und Methoden zur Weitergabe von Informationen
- meistens ist die Art, wie man etwas sagt, wichtiger als was man sagt
- manchmal geschieht die Kommunikation auf rein technischer Ebene, manchmal auf fachlicher Ebene
- ein guter Architekt richtet seine Kommunikation am Zielpublikum aus
- Architekturen müssen an die Entwickler, die Projektteilnehmer und an die Nutzer kommuniziert werden
  - ▶ technische Aspekte, Programmervorgaben, Designentscheidungen u.ä. and die Programmierer
  - ▶ Qualitätsattribute wie Performance, Usability, Security, Entscheidungen u.ä. and die Anwender
  - ▶ Resourceplanung, Zeitplanung u.ä. an das Projekt
- zusätzlich wird evtl. das Management dauernd über den Fortschritt informiert werden
- gute Kommunikation zwischen allen Beteiligten erhöht die Erfolgsaussichten und verhindert Überraschungen

# 13b) Kommunikation

## ■ die 7 C's der Kommunikation (Tips für effective Kommunikation)

- ▶ Klarheit (Clarity)
  - Klarheit sorgt für Verständnis und effective Kommunikation
  - Einbeziehung der Zuhörer und die Wahl der Sprache unterstützen Klarheit
- ▶ Prägnanz (Conciseness)
  - Weniger ist oft mehr, Zuhörer lieben Prägnanz
  - zu viele Worte beeinträchtigen Klarheit und den Inhalt
  - Prägnanz leitet auf den eigentlichen Inhalt, indem unnötige Dinge weggelassen werden
  - Prägnanz spart Zeit und Geld
- ▶ Konkretheit (Concreteness)
  - spezifische statt vage Aussagen
  - bringt Klarheit in Aussagen und verhindert Mißverständnisse
  - macht Aussagen für Zuhörer interessanter
  - lebhafte, active Sprache fesselt Zuhörer

# 13b) Kommunikation

## ■ die 7 C's der Kommunikation (Tips für effective Kommunikation)

- ▶ Höflichkeit (Courteousness)
  - zeigt Respekt gegenüber den Zuhörern
  - unterstützt bei der Auf-/Annahme der Nachricht durch andere
  - verstärkt Beziehungen
  - berücksichtigt kulturelle, religiöse und ethnische Unterschiede
- ▶ Rücksicht (Consideration)
  - hat die Empfänger der Nachricht im Sinn
  - legt den Schwerpunkt auf "Du" nicht "Wir" und "Ich"
  - berücksichtigt die Blickpunkte der Empfänger
- ▶ Richtigkeit (Correctness)
  - Aussagen müssen richtig sein (keine fake news)
  - die Wortwahl muss das unterstützen
  - unterstützt das Vertrauen der Zuhörer in die Nachricht

# 13b) Kommunikation

## ■ die 7 C's der Kommunikation (Tips für effective Kommunikation)

- ▶ Höflichkeit (Courteousness)
  - zeigt Respekt gegenüber den Zuhörern
  - unterstützt bei der Auf-/Annahme der Nachricht durch andere
  - verstärkt Beziehungen
  - berücksichtigt kulturelle, religiöse und ethnische Unterschiede
- ▶ Rücksicht (Consideration)
  - hat die Empfänger der Nachricht im Sinn
  - legt den Schwerpunkt auf "Du" nicht "Wir" und "Ich"
  - berücksichtigt die Blickpunkte der Empfänger
- ▶ Richtigkeit (Correctness)
  - Aussagen müssen richtig sein (keine fake news)
  - die Wortwahl muss das unterstützen
  - unterstützt das Vertrauen der Zuhörer in die Nachricht

# 13b) Kommunikation

## ■ die 7 C's der Kommunikation (Tips für effective Kommunikation)

### ▶ Vollständigkeit (Completeness)

- alle notwendigen Informationen sind enthalten
- nichts wird verschwiegen
- Hilfreich dabei die 5 W-Fragen (Who, What, Where, When, Why)
- in der Nachricht sind alle relevanten Antworten zu den Fragen
- Vollständigkeit ist die Grundlage für gute Entscheidungen
- spart Zeit und Kosten

# 13b) Kommunikation

## ■ Zuhören

- ▶ richtig zuhören zu können ist enorm wichtig und die Grundlage für Verstehen
- ▶ Kommunikation geht immer in 2 Richtungen
- ▶ Nachrichten können verloren gehen, wenn nicht richtig zugehört wird
- ▶ Hören ist nicht Zuhören, Aufmerksamkeit ist notwendig
- ▶ Einfühlungsvermögen in die Person gegenüber unterstützt das Verstehen
- ▶ Mehr zuhören, weniger reden und Verständnisfragen stellen, zeigen Einfühlungsvermögen
- ▶ die Methode “Effective Listening” kann geübt werden

## ■ Präsentationen halten

- ▶ diese Fähigkeit muss stark ausgeprägt sein und immer wieder geübt werden

# 13b) Kommunikation

## ■ Präsentationen halten

- ▶ diese Fähigkeit muss stark ausgeprägt sein und immer wieder geübt werden
- ▶ die 4 P's von Präsentationen
  - Plan
    - gut geplant ist halb gewonnen
    - Ziele, Zuhörer, Inhalt, Art, Dauer
  - Prepare
    - Aufteilung, Humor, Inhalt, Art, Aufteilung, Design, Methodik
  - Practice
    - Üben, Test
  - Present
    - Dress, erster Eindruck, Augenkontakt, Begeisterung, Lautstärke, keine Panik, positive Aussage am Schluß

## 13c) Führung

- Andere mitreißen, inspirieren oder positiv beeinflussen können
- Respekt, Vertrauen und Glaubwürdigkeit erlangen
- immer hilfsbereit sein
- mit Herausforderungen umgehen können
- technisches Vorbild sein können
- Visionen haben und vermitteln können
- Innovationen vorantreiben
- Verantwortung übernehmen und andere mit einbeziehen
- andere unterstützen und voranbringen
- delegieren können
- Änderungen nicht fürchten sondern vorantreiben
- Experimentierfreude zeigen
- richtig kommunizieren
- ander anleiten (Mentoring)
- Walk the talk
- mit gutem Beispiel vorangehen
- andere einbeziehen

## 13d) Verhandlungsgeschick

- bei Verhandlungen aktiv unterstützen
- Meinungsverschiedenheiten ausräumen
- Einigungen anstreben und erarbeiten
- kommt oft mit der Erfahrung
- Alternativen kennen und betrachten
- auch auf andere Alternativen vorbereitet sein

## 14) Evolutionäre Architekturen

- Viele Softwareanwendungen stehen unter dauerndem Druck geändert und angepasst zu werden
- Software Architekten müssen Anwendungen planen, die diesen Anforderungen genügen
- Lange Planungszeiten vor der Realisierung von Anwendungen gehören der Vergangenheit an
- Moderne (evolutionäre) Anwendungen müssen für die später (sicher) kommenden Änderungen vorbereitet sein
- Wir betrachten
  - a) Änderungen sind unausweichlich
  - b) Lehmann's Gesetze der Software Evolution
  - c) Evolutionäre Architekturen entwerfen

## 14a) Änderungen sind unausweichlich

- mit (Ver)Änderungen muss von Anfang an geplant werden
- Softwaresysteme sind keine statischen Systeme mehr und können nicht geplant kontrolliert werden
- Änderungen können kaum vorhergesehen werden
- moderne Softwaresysteme richten sich an realen Bedingungen aus und sind damit deren Veränderungen unterworfen
- Änderungsanforderungen können aus technischen Gründen entstehen oder aus Anforderungen, die vom Business kommen
- die Vielzahl an aktuellen Technologien lassen mehr Möglichkeiten zu und können deshalb den Wunsch nach Änderung zusätzlich treiben
- Änderungen im Business wie Zukäufe, Verkäufe, neue Märkte, neue Konkurrenten, neue Produkte geschehen in immer kürzeren Zeiträumen
- Erwartungen/Anforderungen von Benutzern werden größer und ändern sich öfter
- Architekturen müssen anpassbare und jederzeit änderbare Lösungen zum Ziel haben

## 14b) Lehmann's Gesetze der Software Evolution

- In seinem Papier “Programs, Life Cycles, and Laws of Software Evolution” beschreibt Lehmann 3 verschiedene Typen von Systemen

- ▶ **S-type** Systeme

- unterscheidbar mit genauer **Spezifikation**
    - kann formal beschrieben werden und die Lösung ist einfach verständlich
    - vollkommen zutreffende Lösungen können bereitgestellt werden
    - Anforderungen ändern sich selten und entwickeln sich nicht weiter
    - Lehmann's Gesetz ist für S-Type Systeme nicht anwendbar

- ▶ **P-type** Systeme

- **Probleme** sind genau festgesetzt
    - das Resultat ist bekannt und eine Spezifikation ist erstellbar
    - im Gegensatz zu S-type Systemen ist aber die Lösung nicht verstanden oder unpraktisch umzusetzen
      - die Komplexität der Logik ist so hoch, daß eine Umsetzung keinen Sinn macht
    - Lehmann's Gesetz ist für P-Type Systeme nicht anwendbar

## 14b) Lehmann's Gesetze der Software Evolution

- In seinem Papier "Programs, Life Cycles, and Laws of Software Evolution" beschreibt Lehmann 3 verschiedene Typen von Systemen

- ▶ E-type Systeme

- System ist realitätsnah (Prozesse und Menschen) modelliert
    - die meisten Softwaresysteme sind E-Type Systeme
    - E-type Systeme beeinflussen ihre Umgebung, in die sie eingebettet sind
    - die Umgebung löst Veränderungsdruck auf E-Type Systeme aus
    - E-type Systeme müssen sich verändern, um benutzbar zu bleiben
    - Lehmann's Gesetz ist für E-Type Systeme anwendbar

# 14b) Lehmann's Gesetze der Software Evolution

- Gesetz I : Continuing change
  - ▶ Softwaresysteme müssen durch einen andauernden Veränderungsprozess gehen, sonst werden sie unbrauchbar
  - ▶ die Zufriedenheit von Endusers wird abnehmen, wenn das System die sich verändernden Anforderungen nicht mehr abdecken kann
- Gesetz II : Increasing Complexity
  - ▶ mit der Zeit erhöht sich durch die vorgenommenen Änderungen die Komplexität des Systems, obwohl Anstrengungen zur Reduzierung der Komplexität unternommen wurden
  - ▶ Software Entropy (abgeleitet aus dem 2. Gesetz der Thermodynamik) sorgt für Unordnung durch die Erhöhung von Modifikationen in einem System
- Gesetz III : Self-regulation
  - ▶ die Entwicklung eines Systems ist selbst regulierend
  - ▶ es gibt strukturelle und orgab'nisatorische Faktoren, die für eine Regulierung sorgen
    - je größer und komplexer eine System ist, um so schwieriger können Änderungen umgesetzt werden
    - komplizierte Abstimmungen in Organisationen behindern zusätzlich

# 14b) Lehmann's Gesetze der Software Evolution

- Gesetz IV : Conservation of organizational stability
  - ▶ über den Lebenszyklus eines Systems ist die Weiterentwicklungsrate nahezu constant und unabhängig von den eingesetzten Ressourcen
- Gesetz V : Conservation of familiarity
  - ▶ über den Lebenszyklus eines Systems muss immer der gleiche Level an Know-How vorhanden sein
  - ▶ Wissen über technische und funktionale Aspekte muss erhalten werden
- Gesetz VI : Continuing growth
  - ▶ wenn sich ein Softwaresystem weiter entwickelt, wird es wachsen
  - ▶ die Anzahl der Funktionen wird wachsen und damit die technischen Implementierungen
  - ▶ steht in Beziehung zu Gesetz II, da Größe in Beziehung zu Komplexität steht
- Gesetz VII : Declining quality
  - ▶ wenn sich ein Softwaresystem weiterentwickelt, wird seine Qualität abnehmen wenn nicht zusätzliche Aufwände in die Qualitätsverbesserung investiert werden
  - ▶ mehr Code erhöht die Wahrscheinlichkeit von Defekten und Ausfällen

# 14b) Lehmann's Gesetze der Software Evolution

## ■ Gesetz VIII : Feedback system

- ▶ Software Weiterentwicklung ist ein komplexer Prozess und benötigt Rückmeldungen von verschiedenen Stakeholdern, um sicher zu stellen, daß die Weiterentwicklung in die richtige Richtung geht und Mehrwert liefert
- ▶ Änderungen werden aus Rückmeldungen abgeleitet

## 14c) Evolutionäre Architekturen entwerfen

- Wie können Software Architekten ihre Architekturen so entwickeln, daß sie auf Änderungen vorbereitet sind
- Evolutionäre Architekturen lassen sich selbst einfach ändern und erweitern
- Evolutionäre Architekturen unterstützen Modifikationen am System ohne die Architektur verändern zu müssen

“An evolutionary architecture supports guided incremental change across multiple dimensions.”

- aus dem Buch : Building Evolutionary Architectures (Rebecca Parsons, Neal Ford, Patrick Kua)

## 14c) Evolutionäre Architekturen entwerfen

- Software Architekten sollten alle Änderungen an Software Systemen führen, damit die Charakteristiken der Architektur erhalten bleiben
  - ▶ Alle Design Entscheidungen und alle Qualitätsattribute
- Wenn sich eine Architektur, ausgelöst durch technische Änderungen oder Änderungen im Business, anpassen muss, dann müssen die Konsequenzen dieser Anpassungen berücksichtigt werden und in eine neue, angepasste Architektur einfließen
- Architekturen müssen vielleicht in einem Maß verändert werden, das nicht verhersehbar war
- Architekten müssen deshalb aktiv durch diesen Änderungsprozess führen
- Architekten nutzen sog. Fitness functions um die Auswirkungen von Modifikationen an Software Systemen zu bewerten

# 14c) Evolutionäre Architekturen entwerfen

## ■ Fitness functions

- ▶ eine Funktion, um festzustellen, wie nah sich eine Lösung am gewünschten Ziel befindet
- ▶ sie bestimmen die Fitness einer Lösung
- ▶ Fitness functions können auch zur Überprüfung der Designcharakteristik von Software Architekturen verwendet werden
- ▶ Fitness functions helfen die Fitness einer Lösung vor und nach der Realisierung zu messen und zu bestimmen
- ▶ Sie können auch bei der Auswahl geeigneter Lösungen helfen

# 14c) Evolutionäre Architekturen entwerfen

## ■ Kategorien von Fitness functions

- ▶ atomare (fokussiert auf eine Architekturcharakteristik)
- ▶ ganzheitliche (auf mehrere Charakteristika fokussiert)
- ▶ ausgelöste (ausgelöst durch einen Event)
- ▶ kontinuierliche (dauernd laufende)
- ▶ statische (der Wert einer Bedingung bleibt constant)
- ▶ dynamische (sich verändernde Bedingungen)
- ▶ automatische (automatisch ausgelöst, z.B. in einem automatischen Build Prozess)
- ▶ manuelle (manuell ausgelöst)
- ▶ zeitlich begrenzte
- ▶ bewusste (schon früh bekannte)
- ▶ plötzlich auftretende (z.B. während der Entwicklung)
- ▶ domänen-spezifische (in Beziehung zu einer Business Domäne, z.B. Regeln oder Sicherheit)

# 14c) Evolutionäre Architekturen entwerfen

- Beispiele von Fitness functions
  - ▶ Fitness functions können in Form von Tests entstehen, aber nicht alle Tests sind Fitness functions, sondern nur diejenigen, die Architekturcharakteristika betrachten
  - ▶ andere Fitness functions können z.B. die Wartbarkeit betrachten
  - ▶ Security Test können die Fitness eines Systems in Bezug auf Sicherheit prüfen
  - ▶ Andere Beispiele für Fitness functions kommen aus dem Chaos Engineering (Chaos Monkey)
    - um systematisch die Stärke (oder Schwäche) eines System zu messen
    - ein Tool, das von Netflix entwickelt wurde, um die Auswirkung von Ausfällen auf das Gesamtsystem zu messen

## 14c) Evolutionäre Architekturen entwerfen

- Evolutionäre Architekturen unterstützen Änderungen über mehrere Dimensionen
  - ▶ Programmiersprachen
  - ▶ Frameworks
  - ▶ Datenbanken
  - ▶ Security
  - ▶ Performance
- lose Kopplung als wichtiger Pfeiler für evolutionäre Architekturen
- das Design erweiterbarer API's ist ein weiterer Aspekt evolutionärer Architekturen
  - ▶ Postels Gesetz (das Prinzip der Robustheit)
  - ▶ konservativ, was man selbst tut und liberal, was man von anderen akzeptiert
- in evolutionären Architekturen liegt ein Schwerpunkt auf der Nutzung von Standards
- Verlagern von Entscheidungen auf den letztmöglichen Zeitpunkt (last responsible moment)
  - ▶ schwierig zu bestimmen
  - ▶ Kosten der Verschiebung dürfen Kosten der Entscheidung nicht übertreffen

## 15) Wie werde ich ein guter Softwarearchitekt

- Softwareentwicklung und Software Architektur ist ein sich dauernd änderndes Tätigkeitsfeld
- Software Architekten müssen immer auf der Höhe der Zeit sein, aktuelle Trends kenn und sich entsprechend weiterbilden
- **Wir betrachten**
  - a) Continuous learning
  - b) Teilnahme an Open Source Projekten
  - c) den eigenen Blog betreiben
  - d) sich Zeit nehmen, andere zu schulen
  - e) neue Technologien ausprobieren
  - f) selbst entwickeln
  - g) an User Groups und Konferenzen teilnehmen

# 15) Wie werde ich ein guter Softwarearchitekt

## ■ Continuous learning

- ▶ Ein guter Architekt erweitert dauernd sein Wissensportfolio
- ▶ Architekten müssen sich wie Softwaresysteme an Veränderungen anpassen
- ▶ Architekten erhalten ihren Wert, indem sie Ihr Wissen immer auf dem aktuellen Stand halten
  - sie sind unzufrieden mit ihrem Wissen und begierig darauf, neues zu lernen
  - sie vertiefen ihr Wissen in Bereichen, in denen sie noch nicht so gut sind
- ▶ die Technik, Methoden und Konzepte ändern sich – ein guter Architekt kennt sich aus
- ▶ gute Architekten lösen sich von Detailwissen hin zu mehr Überblick und Wissen in der Breite
- ▶ wenn Architekten zu sehr auf einzelne Technologien fokussiert sind, besteht die Gefahr, daß diese Technologien zu sehr in den Fokus rücken und immer wieder empfohlen werden oder Grundlage von Architekturen sind
  - Law of the instrument oder law of the hammer oder Maslow's hammer
  - *“wenn das einzige Werkzeug, das Du hast, ein Hammer ist, sieht alles aus wie ein Nagel”*
- ▶ gute Architekten finden die Zeit und nehmen sich die Zeit, um zu lernen
- ▶ Bücher sind immer eine gute Quelle

# 15) Wie werde ich ein guter Softwarearchitekt

- Open Source Projekt
  - ▶ die Teilnahme an Open Source Projekten kann der Firma und dem Architekten weiter helfen
    - Wissen und Ansehen (Marktwert)
- den eigenen Blog betreiben
  - ▶ über technische Themen in Blogs zu schreiben, kann das Wissen und die Fähigkeit, komplexe Themen darzustellen, erweitern
  - ▶ Blogs helfen dabei, die "Sichtbarkeit" nach außen zu erhöhen (Marktwert)
- sich Zeit nehmen, andere zu schulen
  - ▶ interne Kurse anbieten
  - ▶ Vorlesungen halten
  - ▶ Mentor für andere sein
  - ▶ was man schulen kann, hat man selbst auch verstanden
- neue Technologien ausprobieren
  - ▶ Vorsicht – nicht dem Spieltrieb fröhnen

# 15) Wie werde ich ein guter Softwarearchitekt

- selbst entwickeln
  - ▶ ein guter Softwarearchitekt kann auch selbst entwickeln und kennt die gängigen Methoden und Programmiersprachen
  - ▶ er kann mit Programmierern mitreden und kennt die gängigen Fachbegriffe
  - ▶ manchmal macht es Sinn in einem Projekt eine Teilaufgabe als Entwickler zu übernehmen
  - ▶ Code von anderen Programmierern lesssen und verstehen lernen
- an Usergroups und Konferenzen teilnehmen
  - ▶ aktive Teilnahme
    - Vorträge halten
    - an Arbeitsgruppen teilnehmen
  - ▶ sich mit anderen austauschen
  - ▶ neue Leute kennenlernen (Marktwert)

# 16) Architektur und Legacy Applications

- Auch wenn es viele neue Plattformen, Umgebungen und Technologien gibt und jeden Tag neue dazu kommen, wird sich jeder Architekt mindestens einmal im Leben mit einer Legacy Application beschäftigen müssen
- Software Architekten müssen Legacy Applications deshalb beherrschen können
- **Wir betrachten**
  - a) Was ist eine Legacy Application
  - b) Ändern von Legacy Applications
  - c) Migration in die Cloud
  - d) Anwenden von agilen Methoden
  - e) Build und Deployment modernisieren
  - f) Legacy Applications integrieren (in andere Lösungen)

# 16) Architektur und Legacy Applications

- Was ist eine Legacy Application
  - ▶ eine Legacy Application ist eine Anwendung, die noch in Benutzung aber schwer zu warten ist
  - ▶ auch wenn es mehr Spaß macht, sich mit neuen Technologien zu beschäftigen und neue Systeme auf der grünen Wiese zu planen (Greenfield Ansatz), wird die eine oder andere bereits existierende (Legacy) Anwendung irgendwann auf dem Tisch des Architekten landen
  - ▶ die neuen Anwendungen von heute sind die Legacy Anwendungen von morgen
  - ▶ Probleme mit Legacy Anwendungen
    - keine Architektur, nicht mehr zeitgemäß, schlecht zu warten, keine Dokumentation vorhanden, Programmierer in Rente, Designentscheidungen nicht bekannt, Anforderungen unbekannt
    - Software Entropie ist weit fortgeschritten, Spaghetti code, alte (nicht mehr unterstützte) Versionen von Software und Betriebssystem
    - fachliche Anforderungen haben sich verändert
    - einzige noch lauffähige Anwendung, die ein wichtiges Problem löst
    - Anwendung, die andere Anwendungen mit Daten versorgt oder Grundlage für andere Anwendungen ist

# 16) Architektur und Legacy Applications

- Ändern von Legacy Applications
  - ▶ klassisches Buch : *Refactoring: Improving the Design of Existing Code* von Martin Fowler
  - ▶ “*....the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves ist internal structure*”
  - ▶ die Business Logik und die Funktionalität bleiben erhalten
  - ▶ folgende Schritte sind notwendig
    - den Legacy Code testbar machen
      - unit Test einführen
    - redundanten Code entfernen
      - unerreichbar, tot, auskommentiert, doppelt
    - Tools benutzen
      - IDE kann oft dabei helfen
    - kleine, inkrementelle Änderungen planen und vornehmen
    - Monolithen in Microservices umarbeiten

# 16) Architektur und Legacy Applications

## ■ Migration in die Cloud

- ▶ um Legacy Applications zu modernisieren, könnte auch die Migration in die Cloud ein geeignetes Mittel sein
- ▶ Es gibt Gründe, warum eine Migration Sinn machen könnte wie z.B.
  - Kosteneinsparung bei HW/SW, Betrieb, Patches
  - höhere Verfügbarkeit und Skalierbarkeit
- ▶ die 6 R der Migration
  - Remove (oder Retire)
    - bei jeder Migration fallen immer Anwendungen an, die eingestellt werden können
  - Retain
    - es gibt immer Anwendungen, die in ihrer Umgebung bleiben müssen  
(techn., organisator., finanziell, politisch)
  - Replatform
    - manche Anwendungen werden zu IaaS Providern migriert
  - Rehost
    - ein externer Provider hostet die Anwendung

# 16) Architektur und Legacy Applications

## ■ Migration in die Cloud

- ▶ die 6 R der Migration
  - Repurchase
    - manchmal macht es Sinn die Anwendung durch eine neue Standardanwendung zu ersetzen
  - Refactor (Rearchitect)
    - das ist die anspruchsvollste Variante, hier können aber alle Vorteile der neuen Plattform genutzt werden
      - Verfügbarkeit, Resilienz, Skalierbarkeit, Performance, weltweite Verteilung, Kosteneinsparungen, neue Verrechnungsmodelle (pay per usage)
      - in den meisten Fällen ist eine neue Architektur notwendig

# 16) Architektur und Legacy Applications

- Anwenden von agilen Methoden
  - ▶ ein Teil der Modernisierung einer Legacy Application könnte auch der Einsatz agiler Entwicklungsmethoden sein
  - ▶ agile Methoden können immer dann von Vorteil sein, wenn der Schwerpunkt bei der Anwendung mehr auf Marktorientierung und schnelle/dauernde Änderungen liegt, statt auf Optimierung von längerfristig benutzten Funktionalitäten
  - ▶ der Einsatz von agilen Methoden bedingt allerdings ein Umdenken in den Entwicklungsteams
  - ▶ der Einsatz von Microservices unterstützt den Ansatz agiler Methoden

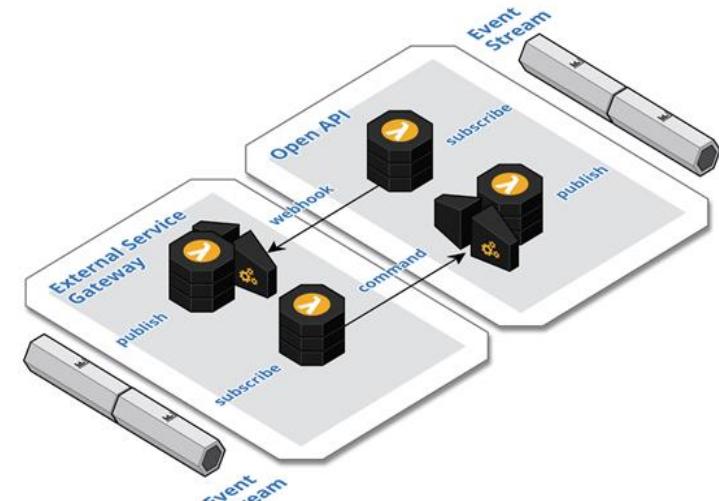
# 16) Architektur und Legacy Applications

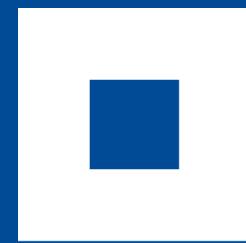
- Build und Deployment modernisieren
  - ▶ oft werden Legacy Applications mit veralteten und manuell gesteuerten Build- und Deployment Prozessen betrieben
  - ▶ manchmal steuern Skripte diesen Prozess, die entweder plattformspezifisch oder unübersichtlich und ohne Dokumentation sind
  - ▶ die Umstellung auf moderne Prozesse und Tools bringt immer Vorteile
    - eine Schwerpunkt sollte auf die Automatisierung dieser Prozesse gelegt werden (s. DevOps Kapitel)
  - ▶ die schrittweise Einführung von Continuous Integration und Continuous Delivery haben weiteres Optimierungspotential

# 16) Architektur und Legacy Applications

## ■ Legacy Applications integrieren

- ▶ Anwendungen haben oft verschiedenste Integrationen mit anderen Anwendungen
- ▶ können Legacy Applications nicht komplett neu aufgebaut werden (z.B. Refactor), kommen erprobte Patterns für die Integration von Anwendungen zum Einsatz
  - wir hatten in Kapitel 8 das Pattern “External Service Gateway”
    - Integration mit externen Systemen über die Kapselung der ein- und ausgehenden Kommunikation zu anderen Systemen über eine isolierte Komponente als Brücke zum Austausch von Events
    - Das erlaubt die Einbeziehung von Legacy Systemen in cloud-basierte, asynchrone, message-orientierte Umgebungen
- ▶ weitere Integrationen könnten über gemeinsam genutzte Daten und/oder Funktionen realisiert werden





**BASF**

We create chemistry