Software Engineering und Projektmanagement 2.0 VO

Projektphase Entwurf (Wie wird gebaut?)

„ [When talking architecture] ... we're talking something that's important" – Fowler (2003)

www.inso.tuwien.ac.at

Motivation

- **Fachliche Sicht trifft auf technische Realisierung**
- **Liefert „Bauplan“ für Gesamt-System**
- **Spezifiziert Struktur und Verhalten der Software und ihrer Komponenten**
- **"*Prozess der Definition der Architektur, Komponenten, Schnittstellen und anderer Merkmale eines Software(teil)systems*.“ [IEEE Standard Glossary, 1990]**

Übersicht

- **Was ist Softwareentwurf und Architektur**
- **Entwurfsparadigmen**
- **Erläuterung der unterschiedlichen Aspekte von Architekturen**
  - Welche Sichtweisen von Architekturen gibt es?
- **Architektur im agilen Kontext**
- **Entwicklung der Architektur**
  - Wie hat sich die Softwarearchitektur mit der Zeit verändert?
- **Architektur Beispiele**
- **Übersicht über Patterns**
  - Was ist die Motivation hinter Patterns, wo könnten Patterns in Projekten eingesetzt werden?

Was ist Softwareentwurf und Architektur

- **Ziele und Aufgabe des Entwurfs**
  - Entwurf eines Systems, das die Anforderungen des Kunden optimal abdeckt
  - Sorgt für Umsetzung der nicht-funktionalen und funktionalen Anforderungen
  - Beschreibung des Systems auf technischer Ebene
  - Ausgangspunkt für Implementierung
  - Festlegung auf Architektur des Systems
    - Auf Bausteinebene
    - Auf Gesamtsystemebene
  - Aufteilung des Systems in funktional getrennte Blöcke
    - Notwendig um Anforderungen an Skalierbarkeit und Wartbarkeit zu erfüllen

Was ist Softwareentwurf und Architektur

- **Definition Architektur**
  - Eine Architektur ist eine Beschreibung von System-Strukturen (Datenströme, Modulare Strukturen, Prozess Strukturen usw.) [Bass et al. (1998)]
  - Eine Architektur ist das erste Artefakt zur Analyse der Einhaltung von Qualitätseigenschaften [Bass et al. (1998)]
  - Eine Architektur ist eine Beschreibung der Beziehung von Komponenten und Verbindungen [Bass et al. (1998)]
  - **Eine Architektur ist die fundamentale Organisation eines Systems, verkörpert durch seine Komponenten, deren Beziehungen und deren Umwelt, sowie die Prinzipien hinter dem Design und der Evolution des Systems [IEEE Standard 1471 (2000)]**

Was ist Softwareentwurf und Architektur

- **Was ist Architektur**
  - Die Architektur eines Softwaresystems besteht aus
    - seinen Strukturen
    - der Zerlegung in Komponenten
    - deren Schnittstellen
    - und deren Beziehungen untereinander
  - Architektur beschreibt eine Lösung und umfasst Strukturen und Konzepte eines Systems.
  - Architektur basiert auf Entwurfsentscheidungen
    - einige bilden die Grundlage für den Entwurf der Komponenten,
    - andere bestimmen über die Auswahl der eingesetzten Technologie.

Was ist Softwareentwurf und Architektur

- **Von der Anforderung zur Architektur**
  - Architektur ist der Hauptträger der NFAs eines Systems – diese sind z.B.:
    - Performanz
    - Veränderbarkeit
    - Sicherheit
  - Ziel des Entwurfs: Abbildung eines Systems, das Anforderungen optimal beschreibt
  - Zwei Arten von Anforderungen (funktional und nicht-funktional)
    - Nicht-funktionale Anforderungen (z.B. Zuverlässigkeit, Benutzbarkeit, etc.) sind Faktoren hinter strategischem Entwurf
    - Funktionale Anforderungen (z.B. Verarbeitung von Eingaben, Erstellen von Ausgaben, etc.) sind Faktoren hinter taktischem Entwurf
  - Voraussetzungen um Anforderungen in Architektur umzusetzen
    - Erfahrung des Softwarearchitekten
    - Unterstützung durch Auftraggeber

Entwurfsparadigmen

- **Grundsätze des Entwurfs** - *Grundsätze sind allgemein akzeptierte und abstrakt formulierte Regeln:*

  - Einfachheit gewinnt (Keep It Small and Simple, KISS)
  - Entscheide spezifisch, es gibt kein Patentrezept zur Lösung beliebiger Entwurfsprobleme
  - Explizit statt implizit: Voraussetzungen, Annahmen und auch Entscheidungen möglichst explizit definieren
  - Rechne mit Änderungen und Fehlern
  - Qualitätsanforderungen immer einhalten

Entwurfsparadigmen

- **Prinzipien des Entwurfs (1)** - *Prinzipien sind konkrete und etablierte Regeln des methodischen Softwareengineering:*

  - Geringe Kopplung (Maß für die zwischen Bausteinen bestehenden Abhängigkeiten)
  - Hohe Kohäsion (Maß für den inneren Zusammenhalt von Bausteinen und Funktionen)
  - Separation of Concern: Bausteinen im Systeme haben jeweils spezifische Verantwortlichkeiten, Zuständigkeiten oder Aufgaben
  - Zerlege Systeme in Module: Jedes Modul sollte bestimmte Verantwortlichkeiten kapseln, die über wohldefinierte Schnittstellen zugänglich sind

Entwurfsparadigmen

- **Prinzipien des Entwurfs (2)** - *Prinzipien sind konkrete und etablierte Regeln des methodischen Softwareengineering:*
  - Abstraktion, Kapselung und Geheimnisprinzip:
    - Verbergen (kapseln) des Innenleben der Bausteine
    - Definieren einer öffentliche Schnittstelle, die beschreibt was der Baustein leistet
    - Interne Entwurfs- oder Implementierungsentscheidungen, vor äußerem Zugriff verbergen
  - Konsistenz: Entwurfsentscheidungen sollten innerhalb des Systems stimmig getroffen werden
  - Vermeide Redundanz (Don’t Repeat Yourself, DRY)
  - Vermeide zyklische Abhängigkeiten

Entwurfsparadigmen

- **Objektorientiertes Design - Allgemein**
  - Objektorientierte Entwicklung immer noch eines der wesentlichsten Programmierparadigmen
  - Modelle aus der objektorientierten Analyse
  - Möglichst genaue Abbildung des später zu implementierenden Systems
  - Verwendet folgende Diagramme:
    - Klassendiagramme – Festlegung der Struktur
    - Zustands- und Sequenzdiagramme – Verhalten des Systems
    - Komponentendiagramme – Abbildung der Systemkomponenten und deren Zusammenhang
  - Klassendiagramm im Mittelpunkt
  - Als Notation wird oft UML verwendet

Entwurfsparadigmen

- **Objektorientiertes Design – SOLID Prinzipien (Robert Martin)**
  - **S**RP (Single Responsibility Principle): Bereits vorher als Separation of Concerns definiert. Eine Klasse soll nur eine Aufgabe haben
  - **O**CP (Open-Close Principle): Klassen sollten so beschaffen sein, dass künftige Erweiterungen ohne Änderungen von bestehendem Quellcode möglich sind
  - **L**SP (Liskov Substitution Principle): Unterklassen müssen grundsätzlich für ihre Oberklassen eingesetzt werden können (Vererbung)
  - **I**SP (Interface Segregation Principle): Verwenden Sie feingranulare, client-spezifische Schnittstellen statt ein großes Universalinterface
  - **D**IP (Dependency Inversion Principle): Erlaube Abhängigkeiten von Abstraktionen, nicht von konkreten Implementierungen

Entwurfsparadigmen

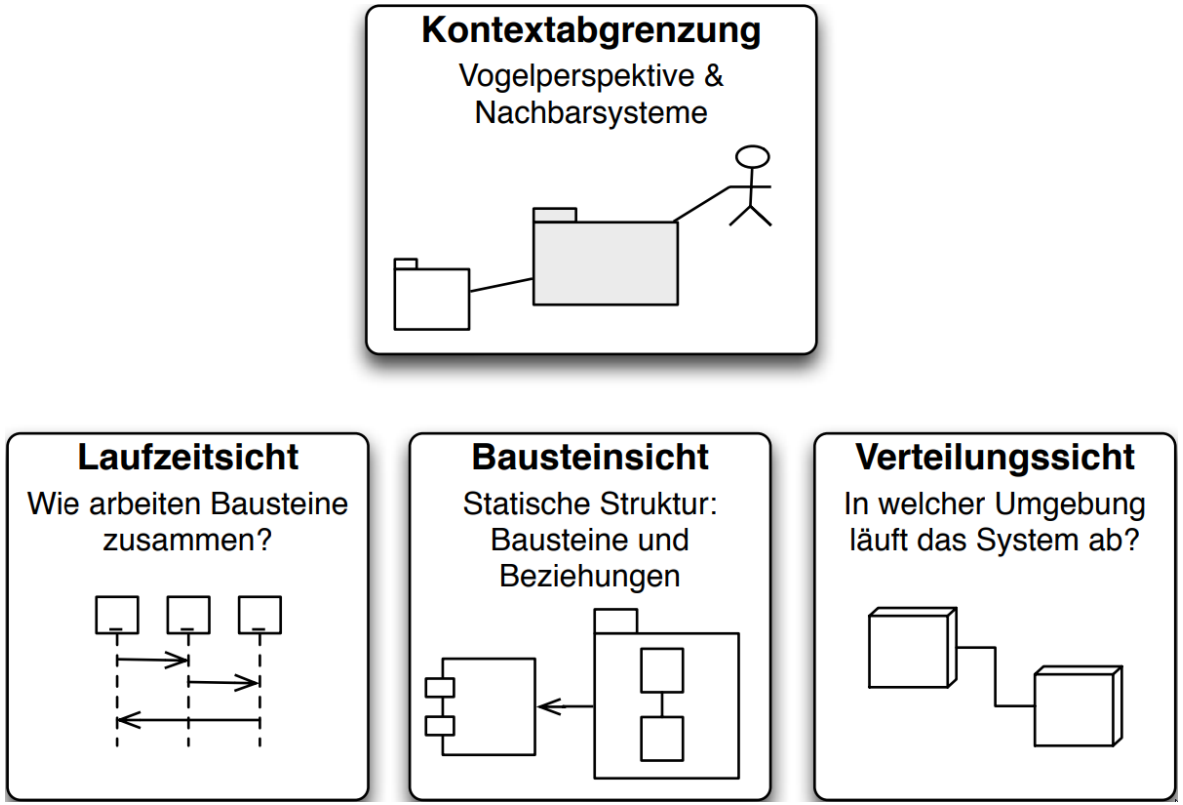
- **Domain-Driven Design (Eric Evans) (1)**
  - Beginn des Entwurfs mit der Strukturierung der jeweiligen Fachdomäne
  - Modellieren der Sprache der Fachdomäne
    - Basis jeder guten Software ist ein profundes Verständnis der jeweiligen Fachdomäne
    - Modellieren einer Abstraktion der Domäne und diskutieren der Abstraktion mit Fachexperten, aber auch mit Softwareentwicklern
    - Freihalten des Domänenmodell von technischen Aspekten
  - Clustern im Großen („Strategisches Design“)
    - Beschreibt Fachgebiete oder Themenbereiche, die inhaltlich eng zusammengehören, genannt Bounded Context (BC)
    - Innerhalb eines BCs besitzen Begriffe eine konsistente Bedeutung
    - Jeder BC kann sein eigenes und eigenständiges Modell haben
    - Modellierung mehrere BCs in einer Context Map
    - Bsp. für BCs eines Telefonanbieters: Netzbetrieb oder Billing

Entwurfsparadigmen

- **Domain-Driven Design (Eric Evans) (2)**
  - Strukturieren im Kleinen („Taktisches Design“)
    - Modellieren feingranulare Basisbausteine innerhalb eines einzelnen BCs, genannt Internal Building Blocks oder Tactical Patterns:
      - Entities: stellen die Kernobjekte einer Fachdomäne dar und sind unveränderliche Identität innerhalb der Domäne
      - Value-Objects: besitzen keine eigene Identität und beschreiben den Zustand anderer Objekte
      - Domain-Events: repräsentieren (vergangene) Ereignisse innerhalb der Domäne
      - Services stellen Abläufe oder Prozesse der Domäne dar, die nicht von Entities wahrgenommen werden
  - Isolieren der Domäne in der Architektur
    - Struktur und Konzepte der Fachdomäne sollen langfristig in der Architektur repräsentiert werden
    - Daher Implementierung der Domäne (etwa Aggregate, Entitäten, Services und Value-Objekte) in eine eigene Schicht der Architektur

Aspekte und Sichtweisen von Architekturen

- **Vier Sichten auf Architektur**



Aspekte und Sichtweisen von Architekturen

- **Vier Sichten auf Architektur**
  - Kontextabgrenzung - Wie ist das System in seine Umgebung eingebettet?
    - Schnittstellen zu Nachbarsystemen
    - Interaktionen mit wichtigen Stakeholdern
    - wesentlichen Teile der umgebenden Infrastruktur
  - Bausteinsicht – Wie ist das System intern aufgebaut?
    - statischen Strukturen der Architekturbausteine des
      - Systems
      - Subsysteme
      - Komponenten
      - und deren Schnittstellen
    - Bausteinsichten werden meist top-down beschrieben
    - Bausteinsichten unterstützen Projektleiter und Auftraggeber bei der Projektüberwachung und dienen der Zuteilung von Arbeitspaketen

Aspekte und Sichtweisen von Architekturen

- **Vier Sichten auf Architektur**
  - Laufzeitsicht – Wie läuft das System ab?
    - welche Bausteine des Systems existieren zur Laufzeit und wie wirken sie zusammen
    - beschreibt dynamische Strukturen
  - Verteilungssichten (Infrastruktursichten) – In welcher Umgebung läuft das System ab?
    - beschreiben die Hardwarekomponenten, auf denen das System läuft
    - Dokumentation der physischen Systemumgebung
    - zeigt das System aus Betreibersicht

Architektur im agilen Kontext

- Ca. 10 bis 25 % der Anforderungen an Software ändern sich pro Jahr
- Management und Architekten von Softwareprojekten müssen sich durch flexible und bedarfsgerechte Vorgehensweisen darauf einstellen
- Softwarearchitekturen müssen stabile Grundgerüste bereitstellen, die die Umsetzung neuer und geänderter Anforderungen ermöglichen
- Änderungen müssen schnell und effektiv erfolgen oder sie sind wirkungslos
- Agile Methoden basieren auf Feedback und Iterationen

Architektur im agilen Kontext

- **Emergent Design – ohne bewusste Steuerung gestaltetes Design**
  - Basiert auf Refactoring von erstellten, lauffähigen Komponenten
  - Zuerst Deliverables (Best Practices!) → Dann entsteht Design
  - Beispiele für notwendige Best Practices:
    - Code Conventions
    - Code Reviews
  - High Quality Design ergibt sich aus 4 einfachen Regeln:
  - Test everything; eliminate duplication; express all ideas; minimize entities
- **Rahmenbedingungen trotzdem notwendig**
  - Benachbarte Systeme, existierende Systemlandschaft
  - → strategischer Rahmen muss definiert sein

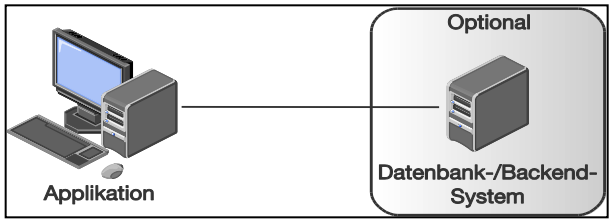
Entwicklung der Architektur

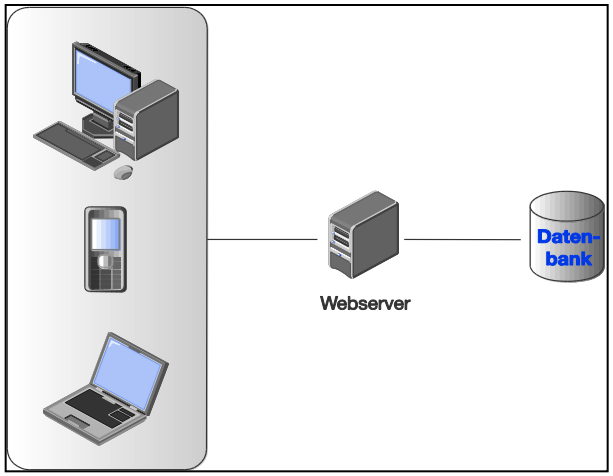
- **Faktoren zur Weiterentwicklung von Softwarearchitektur**
  - Prozedural zu objektorientierten Programmiersprachen
  - Vom Großcomputer zum Heimcomputer
  - Durch objektorientierte Programmiersprachen Trend von monolithischen Applikationen zu modularen Systemen
  - In den 80er, 90er und frühen 2000er Jahren → hauptsächlich *Rich-Client-Applikationen*
  - Trend zu Webapplikationen, Self Containt Systems (SCS, Microservices) und Service-orientierte Architektur (SOA)
- **Historische Softwarearchitekturen**
  - 2-Tier Architekturen
  - 3-Tier Architekturen
  - Service-orientierte Architekturen
  - Microservice Architekturen

Entwicklung der Architektur

- **Entwicklung von Softwarearchitekturen:**

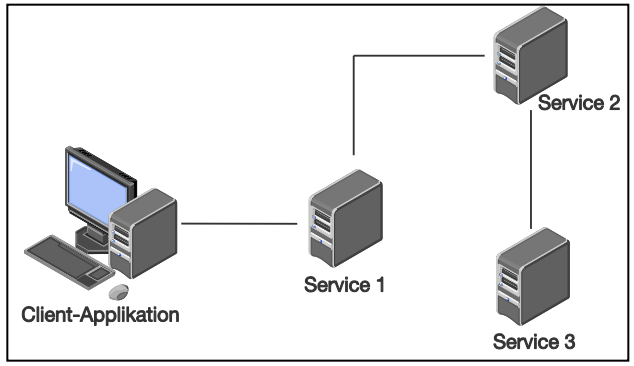
2-Tier Architektur

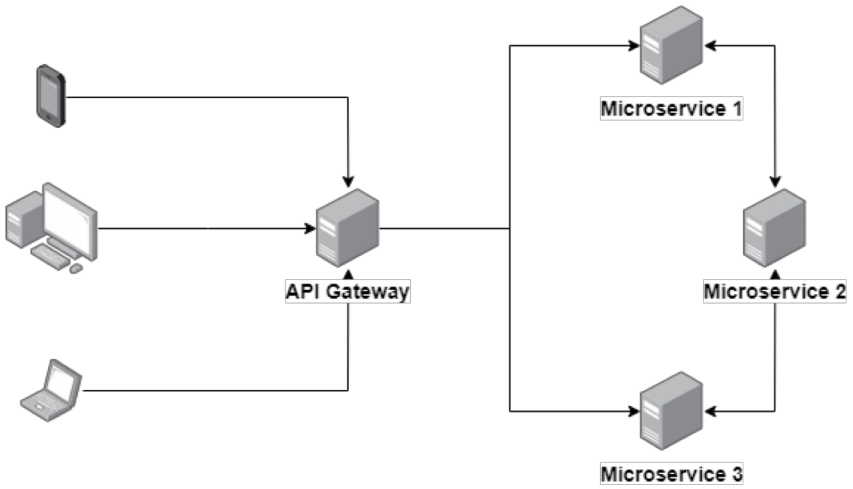




3-Tier Architektur

Service-orientierte Architektur

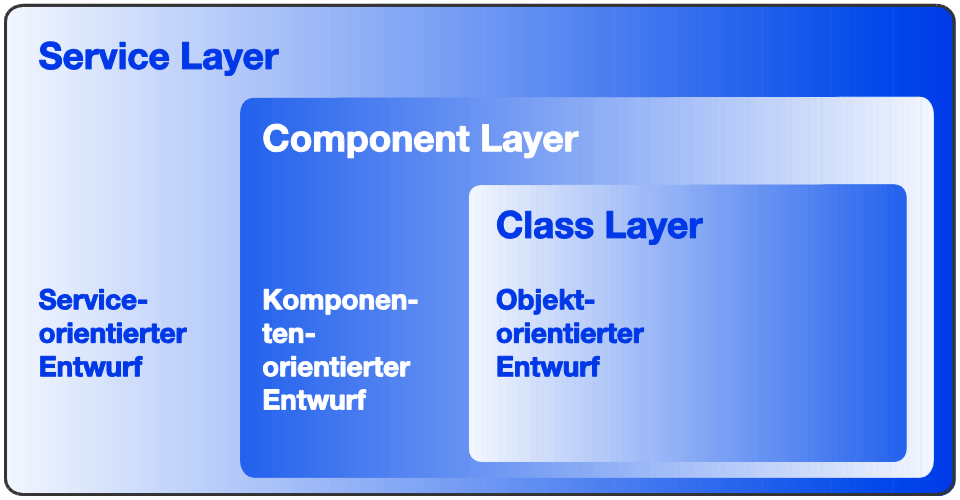




Microservice Architektur

Architektur Beispiele - SOA

- **Service-orientiertes Design**
  - Umspannt mehrere Abstraktionsebenen
    - Service Layer
    - Component Layer
    - Class Layer
  - Unterschiedliche Notationen je Layer
    - Z.B. Webservice (Service Layer), Komponentendiagramme (Component Layer), UML (Class Layer)



Architektur Beispiele - SOA

- **Service-orientierte Architektur**
  - Geschäftsprozess-orientiert
  - Anwendungen werden durch mehrere Services gelöst
  - Services in einer Anwendung sollen auch aus anderen Anwendungen heraus aufgerufen werden können
    - Reuse
    - Modularisierung
  - Zusammenstellung von Services in bestimmten Abläufen zur Umsetzung von Geschäftsprozessen (Orchestrierung)
  - Services existieren auf mehreren Abstraktionsebenen (Business Service, Component Service, technisches Service, usw.)
  - Beschreibt eine Architektur, keine Technologie

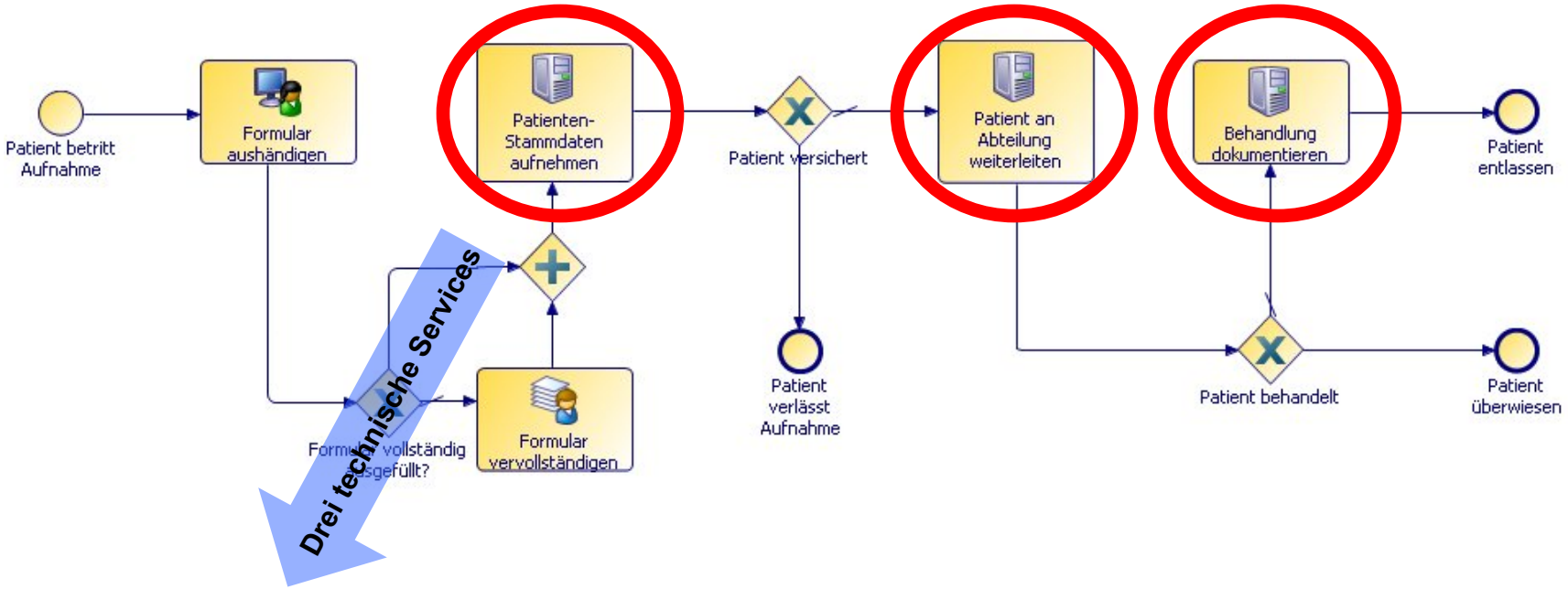
Architektur Beispiele - SOA

- **Paradigmen der Service-orientierten Architektur**
  - Lose Kopplung
    - Niedrige Kopplung der Services ermöglicht hohen Reuse und Flexibilität
  - Abstraktion
    - Reduktion von Neuentwicklungen – Spannungsfeld Abstraktion vs. Technische Anforderungen (z.B. KÖNNEN manche technische Services einfach nicht abstrakt gelöst werden)
  - Standardisierung
    - Standardisierung erlaubt die Zusammenarbeit von Services
  - Composable
    - Standardisierte Services können zu Komponenten zusammengefasst werden

Business Process Modelling (BPM) und SOA

- Im Service-orientierten Entwurf wird ein Schritt in einem Geschäftsprozess extrahiert und gekapselt → dabei handelt es sich um ein sog. Business Service
- Service-orientierte Architekturen sind *business driven*, d.h. sie sind an Geschäftsprozesse angelehnt
- BPM und SOA sind eng miteinander verknüpft – Anwendungen werden maßgeblich von der Geschäftsprozess-Modellierung beeinflusst

Beispiel zu BPM und SOA



- **eingabePatientInKIS:** Eingabe der Patientendaten in das Krankenhaus Informationssystem (KIS)
- **abfrageVersichertenStatus:** Abfrage des Versichertenstatus – Aufruf an externes Service
- **anbindungAnZentralesRegister:** Abruf der Behandlungshistorie aus zentralem Register – Verknüpfung mit Patientendaten (Ausgabe: ELAK)

Architektur Beispiele - Microservices

- **Microservices bezeichnet einen Architekturstil, der die Aufteilung größerer Systeme in kleinere, einzeln entwickel-, deploy- und betreibbare Einheiten vorsieht, mit folgenden Zielen:**
  - Erhöhung der Wandelbarkeit und Flexibilität von Software
  - Verbesserung von Time-to-Market, d. h. neue Features sowie Fixes sollen schneller in Produktion gehen
  - Schnellere Entwicklung durch kleinere Einheiten
  - Mehr Innovationsfähigkeit sowie Flexibilität bezüglich Technologieauswahl, d. h. Entwicklungsteams treffen Technologieentscheidungen
  - Jedes Microservice läuft in einer eigenen Ablaufumgebung, ist also unabhängig von anderen Microservices

Architektur Beispiele - Microservices

- **Charakterisierung von Microservices**
  - Modularisierung anhand von fachlichen Funktionen/Aufgaben und über Services, die über leichtgewichtige Protokolle wie HTTP interagieren
  - Jedes Service läuft in einer eigenen Ablaufumgebung, Deployment und Skalierung geschehen unabhängig
  - Selbstständig agierende Endpunkte und simple Kommunikationskanäle
  - Management von Microservices geschieht dezentral, durch voneinander unabhängige Teams
  - Dezentrale Datenverwaltung: Microservices enthalten und verwalten ihre eigenen Daten
  - Die gesamte Infrastruktur von Microservices (Entwicklung, Inbetriebnahme, Deployment/ Delivery) ist automatisiert
  - Kommunikation zwischen Microservices findet in der Regel „remote“ statt. Das benötigt Werkzeuge zum gegenseitigen Auffinden von Service-Instanzen inklusive Failover („Service Discovery“).
  - Evolutionärer Entwurf: Microservices als kleine Entwicklungseinheiten erlauben kürzere Release-Zyklen und risikoarme Änderungen

Architektur Beispiele - Microservices

- **Einführung von Microservices in einer Organisation funktioniert nur, wenn auch die Organisation dementsprechend angepasst wird**
  - Organisation (Firma, Abteilung, Bereich) sollte funktionsübergreifende Teams (cross-functional teams) unterstützen
  - Teams sollen sich von Anfang bis Ende, von Konzeption bis Produktion um „ihr Produkt“ kümmern. Amazon: „You build it, you run it“
  - Ein Team kann dabei mehrere Microservices verantworten – aber nie umgekehrt

Architektur Beispiele - Microservices

- **Für welche Systeme eignen sich Microservices?**
  - Internetbasierte Systeme, im weitesten Sinne eBusiness-Anwendungen, mit hohen Anforderungen an Flexibilität
  - Klassische Client/Server-Anwendungen können von Microservices profitieren, wenn die serverseitigen Anwendungen stärker modularisiert und voneinander entkoppelt werden
  - Wann immer unterschiedliche (fachliche) Funktionalitäten voneinander getrennt entwickelt und betrieben werden können, stellen Microservices eine Alternative zu herkömmlichen Monolithen dar

Architektur Beispiele - Microservices

- **Herausforderungen bei Microservices**
  - Entwickeln, testen, bauen und deployen – das sind schwierige Aufgaben, bei denen Teams nicht jedes Mal dieselben grundlegenden Probleme lösen möchten
  - Service-Discovery: vielen Services müssen zur Laufzeit zueinander finden
  - UI-Integration: Systeme sollen eine einheitliche, konsistente Benutzeroberfläche haben, wie kann das mit vielen eigenständigen Microservices funktionieren
  - Dezentralisierte Daten: Konsistenz von Daten sicherstellen, wenn jedes Service nur einen kleinen Teil der gesamten Daten verwaltet
  - Viele Laufzeitumgebungen müssen aufgesetzt, betrieben und überwacht werden

Patterns

- **Patterns stellen allgemeine Lösungsstrategien für wiederkehrende Probleme der Softwareentwicklung dar**
  - Abstrakt
  - Weitestgehend technologie-unabhängig
  - Wesentlicher Input in Entwurfsentscheidungen
- **Zwei Arten von Patterns nach Abstraktionsebene**
  - Design Patterns
    - Muster auf Klassen-Ebene (Mikroarchitektur)
  - Architektur Patterns
    - Muster auf architektonischer Ebene (Makroarchitektur)
- **Pattern Language**
  - Sammlung von Entwurfsmustern
  - Schaffung einer einheitlichen Sprache für Probleme und Lösungen

Patterns

- **Design Patterns**
  - Beschreiben wiederholt auftretende Strukturen von miteinander kommunizierenden Komponenten
  - Dienen dazu ein allgemein definiertes Problem innerhalb eines bestimmten Kontexts zu lösen
  - Design Patterns auf Abstraktionsebene unter Architekturpatterns
- **Ein Beispiel zu Design Patterns in Einheit Implementierung**

Patterns

- **Architektur Patterns**
  - Schablonen für konkrete Softwarearchitekturen
  - Spezifizieren systemweite, strukturelle Eigenschaften einer Anwendung
  - Großer Einfluss auf die Architektur der eigenen Subsysteme
  - Fundamentale Designentscheidung
  - Literaturhinweis: *Pattern-oriented Software Architecture – Buschmann et al. (1996)*
- **Bekannte Architektur Patterns**
  - Model-View-Controller
  - Pipes and Filters
  - Layered Architecture

Patterns

- **Architektur Pattern - Model View Controller**
  - Szenario: Interaktive Anwendungen mit einer flexiblen Schnittstelle zwischen Mensch und Computer
  - Problem: GUI wird oft geändert
  - Lösung: Aufteilung in Verarbeitung, Output und Input
- **Model kapselt Daten und Funktion**
- **View stellt Informationen dar**
- **Controller empfängt Events**