

Software Engineering 1

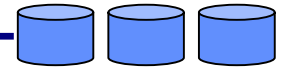
Vorlesungsblock 3: Entwurf

Kristof Böhmer
Fakultät für Informatik
Universität Wien

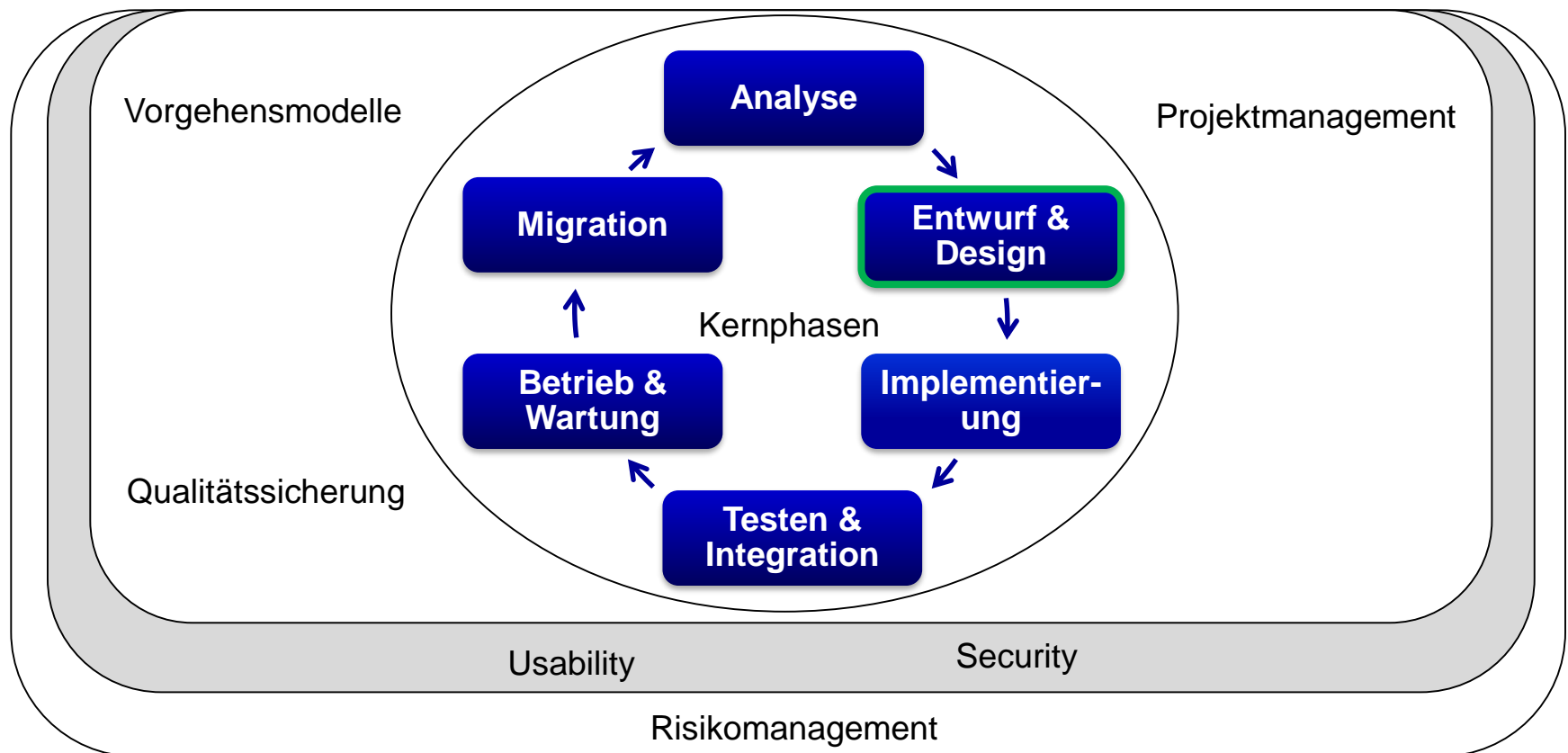
Universität Wien

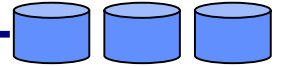
29

Fakultät für Informatik
Institut für Software Engineering und
Formale Verifikation



- ❑ **Projektphasen:** Zuordnung des dritten Vorlesungsblocks – Entwurf
 - Diese Phase beschäftigt sich mit der Architektur des Softwareprojektes.





3.1 Motivation

3.2 Grundlagen

3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

3.5 Serviceorientierte Architektur

3.6 Zusammenfassung

3.1 Motivation: Entwurf



- ❑ **Fachliche Sicht trifft auf technische Realisierung.**
- ❑ **Abbildung von funktionalen und nichtfunktionalen Anforderungen.**
- ❑ **Das Pflichtenheft stellt zumeist den Ausgangspunkt dar.**
- ❑ **Ziel ist es einen „Bauplan“ für das Gesamt-System zu erstellen.**
- ❑ **Bauplan: Spezifikation von Struktur und Verhalten der Software und der enthaltenen oder eingebundenen Komponenten.**

Block 3: Die nachfolgenden Folien basieren (sofern nicht extra angegeben) auf

- T. Grechenig, M. Bernhart, R. Breiteneder, K. Kappel: **Softwaretechnik. Pearson Studium 2010**
- A. Schatten, M. Demolsky, D. Winkler, S. Biffel, E. Gostischa-Franta, T. Östreicher: **Best Practice Software-Engineering. Spektrum 2010**

3.1 Motivation: Lernziele



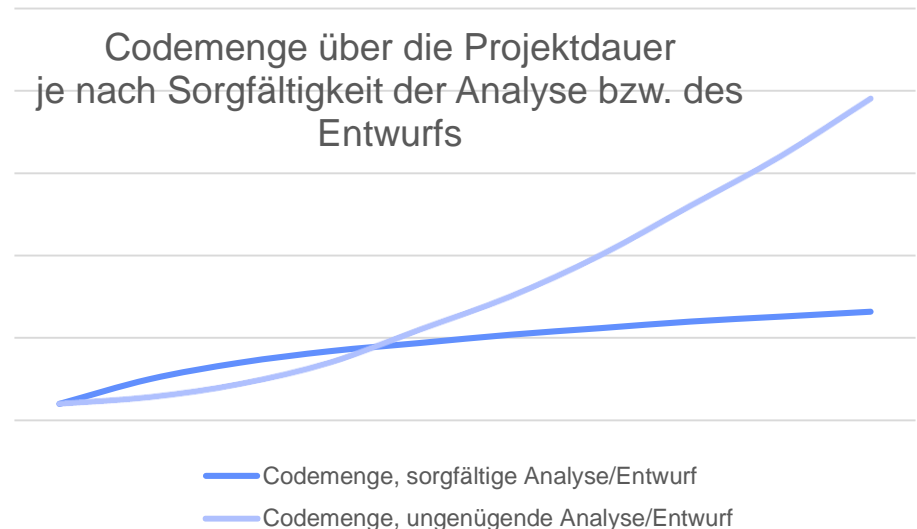
□ Mit diesem Foliensatz möchten wir folgende Lernziele vermitteln:

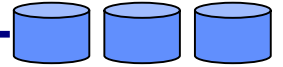
- Die Unterschiede zwischen Makro- und Mikroarchitektur.
 - ◆ Was sind die jeweils wichtigsten Architekturkonzepte?
- Die Transformation von Anforderungen in eine Architektur.
 - ◆ Wie kommen wir von den erhobenen und dokumentierten Anforderungen zu einer Softwarearchitektur?
- Historische Aspekte, zur Kontextbildung.
 - ◆ Wie haben sich Softwarearchitekturen im Laufe der Zeit verändert?
- Objektorientierter Entwurf und serviceorientierter Entwurf.
 - ◆ Was sind die Unterschiede und wie sind entsprechende Entwürfe aufgebaut?
- Einen Überblick über Pattern (Design- und Architekturpattern).
 - ◆ Welche Arten von Pattern gibt es und wie können diese in Projekten eingesetzt werden?
- Detaillierte Darstellung einer Architektur am Beispiel einer serviceorientierten Architektur (SOA).
 - ◆ Wie lässt sich eine derartige Architektur realisieren und was gilt es zu beachten?

3.1 Motivation: Warum ist der Entwurf einer Architektur wichtig?



- ❑ **Ungenügende Sorgfalt während Analyse bzw. Entwurf führt zu:**
 - **Mehraufwand:** Für jede während der Analyse oder dem Entwurf eingesparte Stunde entsteht durchschnittlich ein Mehraufwand von 100 Stunden während der Implementierung und Wartung.
 - **Höhere Codemenge:** Nichtfunktionale Anforderungen werden oft nur ungenügend berücksichtigt. Dies wird zumeist erst spät erkannt und führt meist dazu, dass große Codeteile neu geschrieben und ausgetauscht werden müssen.
 - **Schlechte Erweiterbarkeit:** Neue Funktionen lassen sich nur mit viel Aufwand oder gar nur durch die Neuentwicklung größerer Codeteile integrieren.





3.1 Motivation

3.2 Grundlagen

3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

3.5 Serviceorientierte Architektur

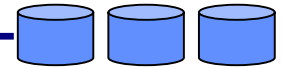
3.6 Zusammenfassung

3.2 Grundlagen: Definition Entwurf und Design



- ❑ **Entwurfs- und Designphase sind eng aneinander gekoppelt**
⇒ Beide werden nur selten in unterschiedliche Phasen unterteilt.
- ❑ **Nach dem IEEE 610.12-90 ist Entwurf/Design folgendermaßen definiert:**
 - „*Software Design is the process of defining the architecture, components, interfaces, and other characteristics of a system or component and the results of that process.*“
- ❑ **Nach der Definition der Anforderungen in der Analysephase wird nun (während der Entwurfsphase) spezifiziert wie diese Anforderungen im Detail umgesetzt werden sollen.**
 - Basierend auf der Anforderungsdokumentation werden nun alle Artefakte (Klassen, Module, Interaktionen, etc.) geplant, welche für die konkrete Realisierung des Projektes notwendig sind.
⇒ Dies bildet den Grundstein für die konkrete Implementierung.

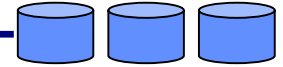
3.2 Grundlagen: Ziel der Entwurfsphase



- ❑ **Ziel der Entwurfsphase:** Festlegen der konkreten Umsetzung des geplanten Projektes.
 - Definition der internen Strukturen, der Architektur, der Datenflüsse, der Algorithmen und der Komponenten bzw. Schnittstellen zwischen den Komponenten sowie der Benutzerschnittstellen.
 - Wichtig: Jede Designentscheidung muss dokumentiert werden.



3.2 Grundlagen: Zusammenhang Entwurf und Architektur



❑ **Strategisches Design bzw. strategischer Entwurf**

- Architektur im „Großen“ (Zusammenspiel verschiedener Systeme und Services).
- Entwurf auf Systemebene (Makroarchitektur).
- Entscheidungen über Technologie und Gesamtaufbau des Systems.

❑ **Taktisches Design bzw. taktischer Entwurf (Entwurf im klassischen Sinne)**

- Architektur im „Kleinen“ (Mikroarchitektur).
- Entwurf im klassischen Software Engineering Sinne (auf Bausteinebene, nicht auf Systemebene, der Fokus liegt auf: Klassen, Modulen, Methoden, Algorithmen).
- Entscheidungen über Aufbau der einzelnen Bausteine (z.B. Entwurf der Klassen).
- Hat ein niedrigeres Abstraktionslevel als die Architektur (z.B. Methoden statt Services).

❑ **Beide Aspekte werden vor der Umsetzung modelliert und müssen entsprechend dokumentiert werden (beispielsweise mit UML Komponenten- und Klassendiagrammen).**



❑ Definition der Architektur nach Len Bass

- *“Structure or structures of the system, which comprise software elements and the externally visible properties of those elements and the relationship among them.”*

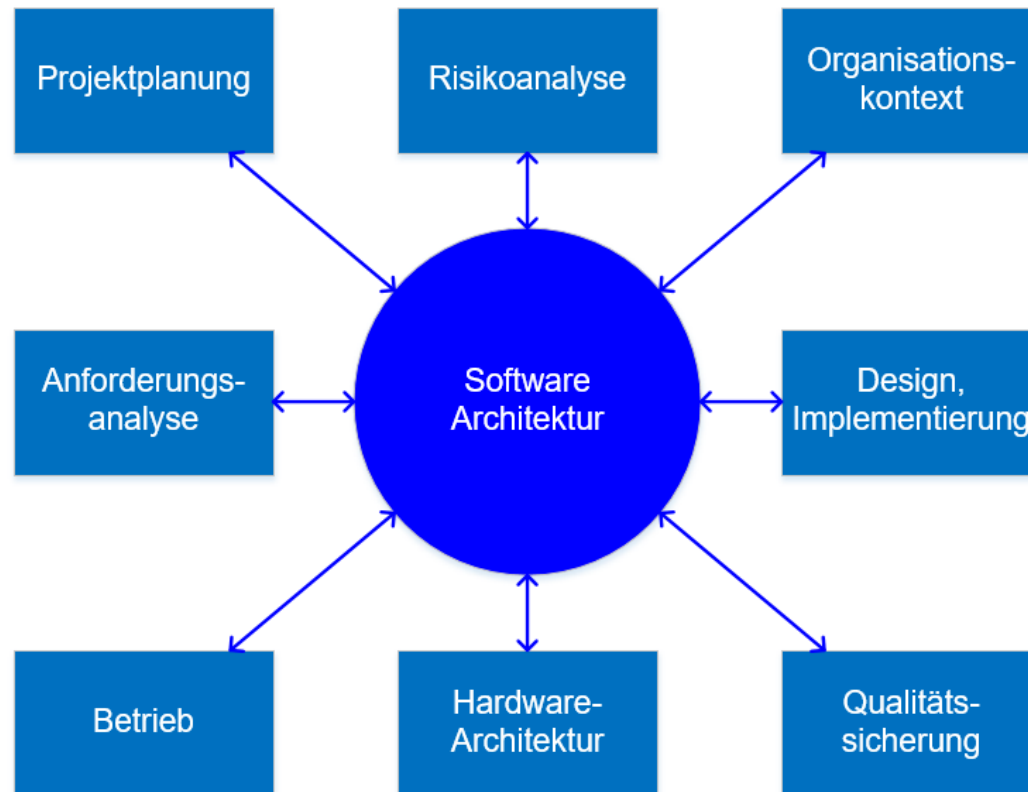
❑ Eine Software Architektur identifiziert Komponenten (z.B. Klassen, Module, Services) und regelt deren Zusammenspiel.

- Ziel: Unterstützen von funktionalen und nichtfunktionalen Anforderungen.
- Vor allem die nichtfunktionalen Anforderungen haben einen großen Einfluss auf die Architektur.
 - ◆ Beispielsweise ist je nach gewünschter Leistungsfähigkeit (z.B., 100 vs. 1000 parallele Anwender) eine andere Architektur notwendig.

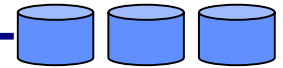
3.2 Grundlagen: Einfluss der Architektur



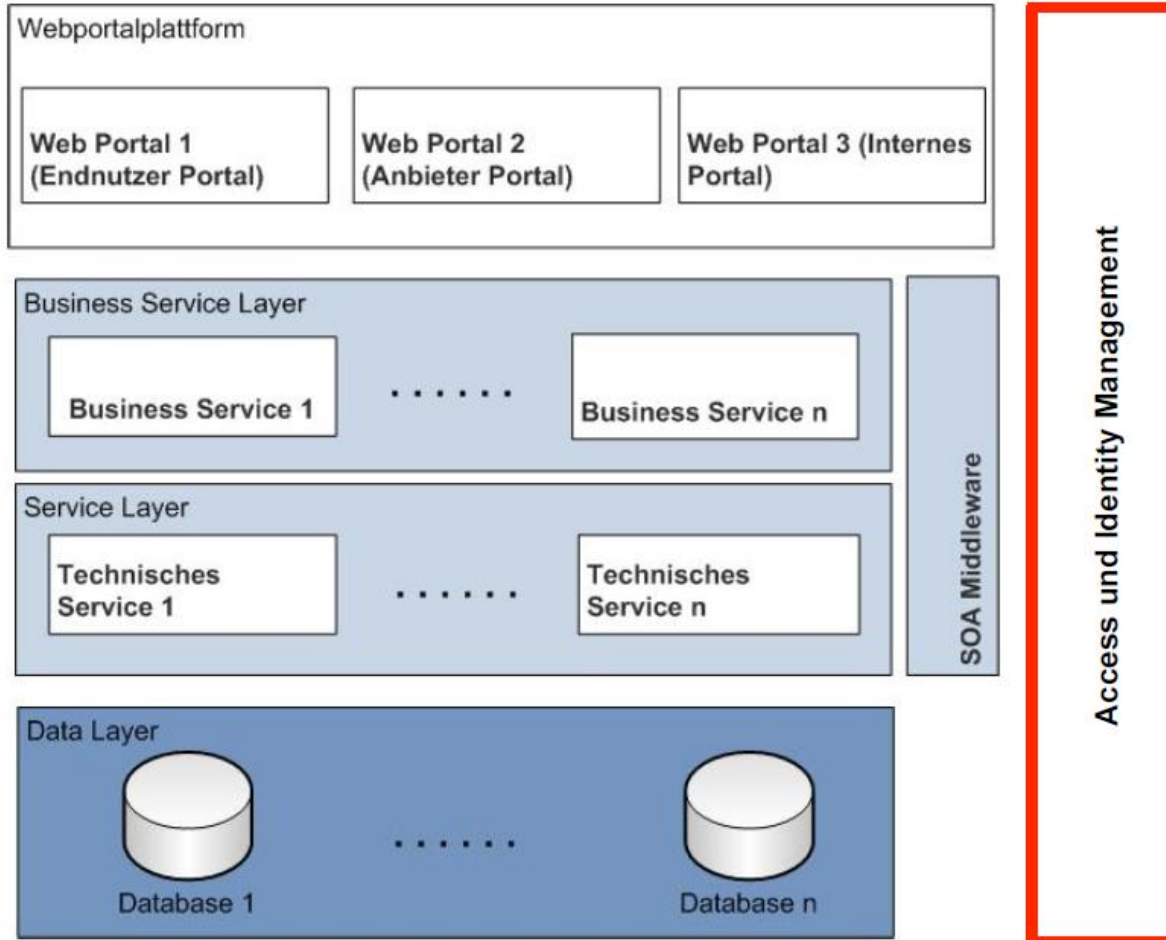
- **Eine Architektur beeinflusst alle Bestandteile eines Projektes und wird im Gegenzug auch von allen beeinflusst.**
 - Die hierdurch entstehenden Wechselwirkungen sind im Folgenden sichtbar:



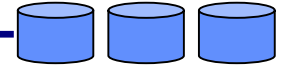
3.2 Grundlagen: Beispiel für eine Architektur 1



□ Beispiel für die Architektur einer Software

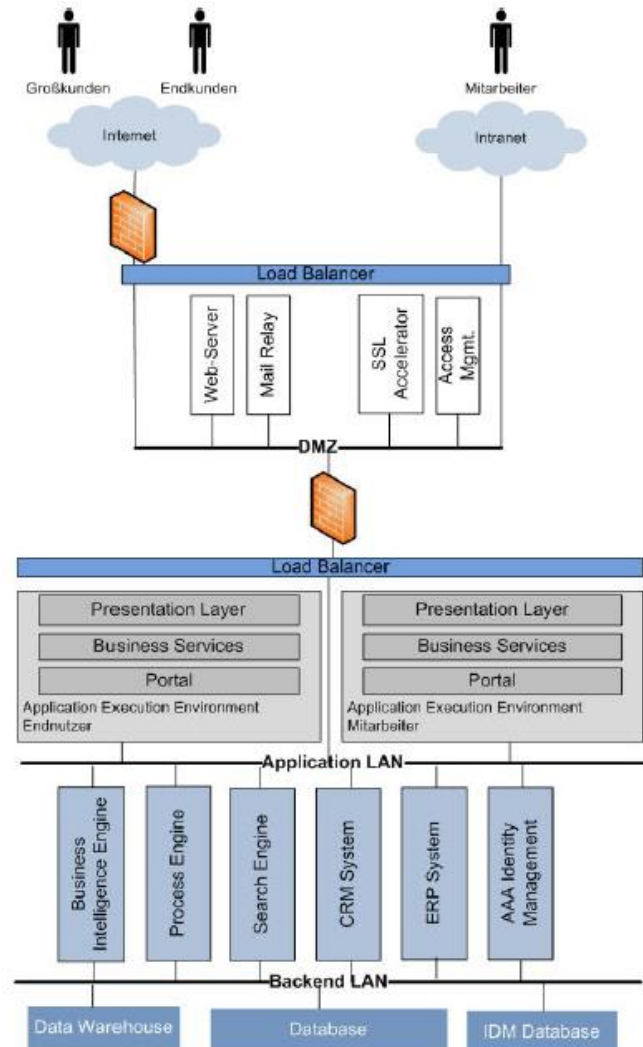


3.2 Grundlagen: Beispiel für eine Architektur 2



□ Architektur im „Großen“

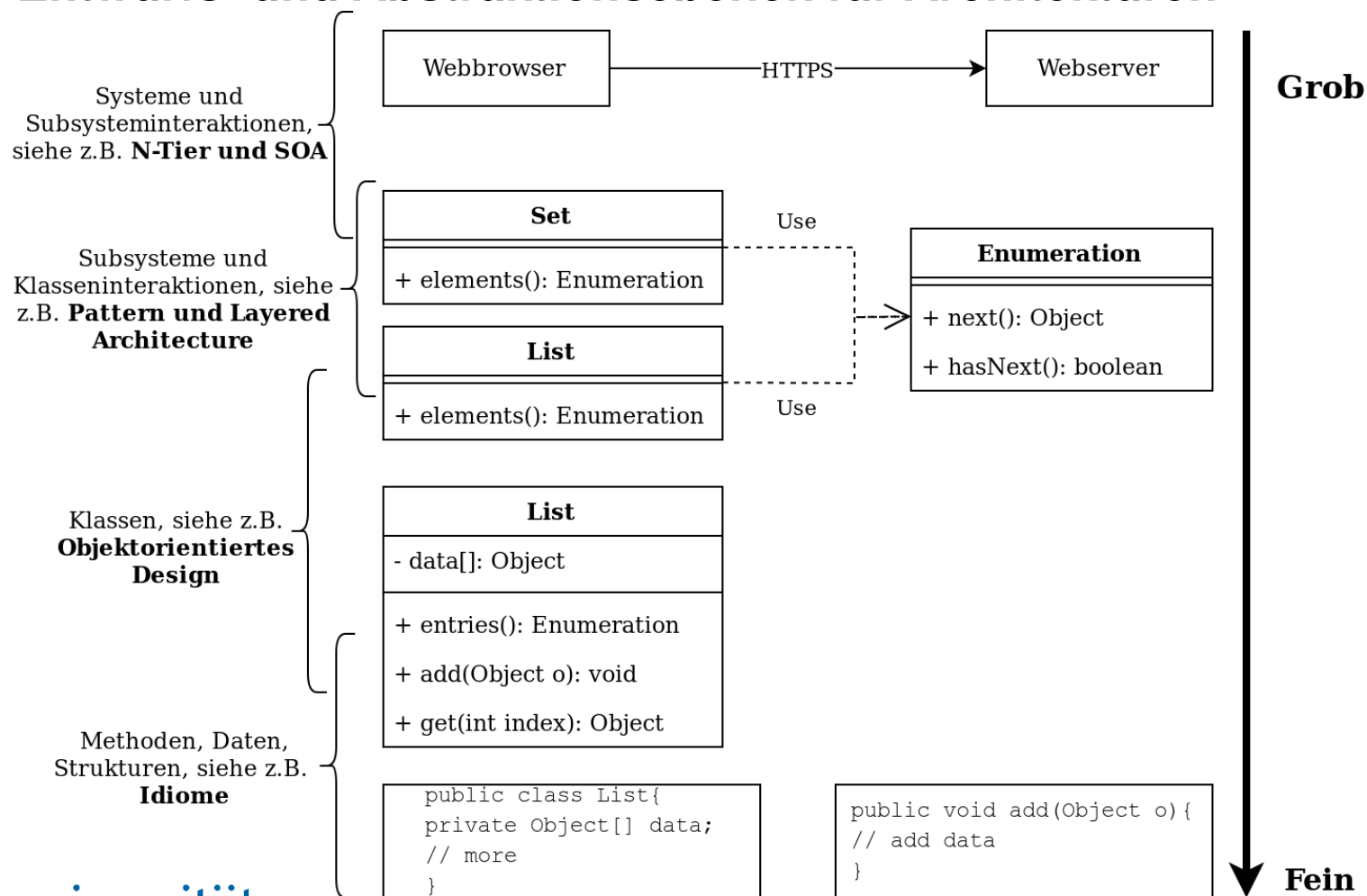
- Zeigt grob wie mehrere einzelne Komponenten zusammenarbeiten.
- Zumeist werden mehrere Sichtweisen (auf unterschiedlichen Abstraktionsebenen) kombiniert. Auf der folgenden Seite wird hierzu ein Beispiel gegeben.
- Jede einzelne dieser Sichtweisen/Ebenen wird aufeinander abgestimmt bzw. baut aufeinander auf – siehe die folgende Seite.



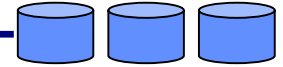
3.2 Grundlagen: Beispiel für eine Architektur 3



□ Entwurfs- und Abstraktionsebenen für Architekturen

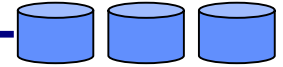


3.2 Grundlagen: Von der Anforderung zur Architektur 1



- ❑ **Die Architektur beeinflusst und wird beeinflusst von nichtfunktionalen Anforderungen, wie:**
 - Performance
 - Modifiability
 - Security
- ❑ **Das Hauptziel des Softwareentwurfs ist der Entwurf eines Systems, das die Anforderungen des Kunden optimal abdeckt.**
 - Dies erfordert jedoch, dass die erhobenen Anforderungen folgende Eigenschaften aufweisen (siehe Anforderungvalidierung):
 - ◆ Jede Anforderung muss eindeutig identifizierbar sein,
 - ◆ eine abgeschlossene mess- und prüfbar Aufgabenstellung behandeln und
 - ◆ die Schnittstellen zu anderen Systemen, Services und Komponenten exakt definieren.
 - Außerdem sind noch folgende Voraussetzungen notwendig:
 - ◆ Erfahrene Softwarearchitekten und
 - ◆ der Auftraggeber muss sich mit einbringen (beispielsweise um Unklarheiten zu klären).

3.2 Grundlagen: Von der Anforderung zur Architektur 2

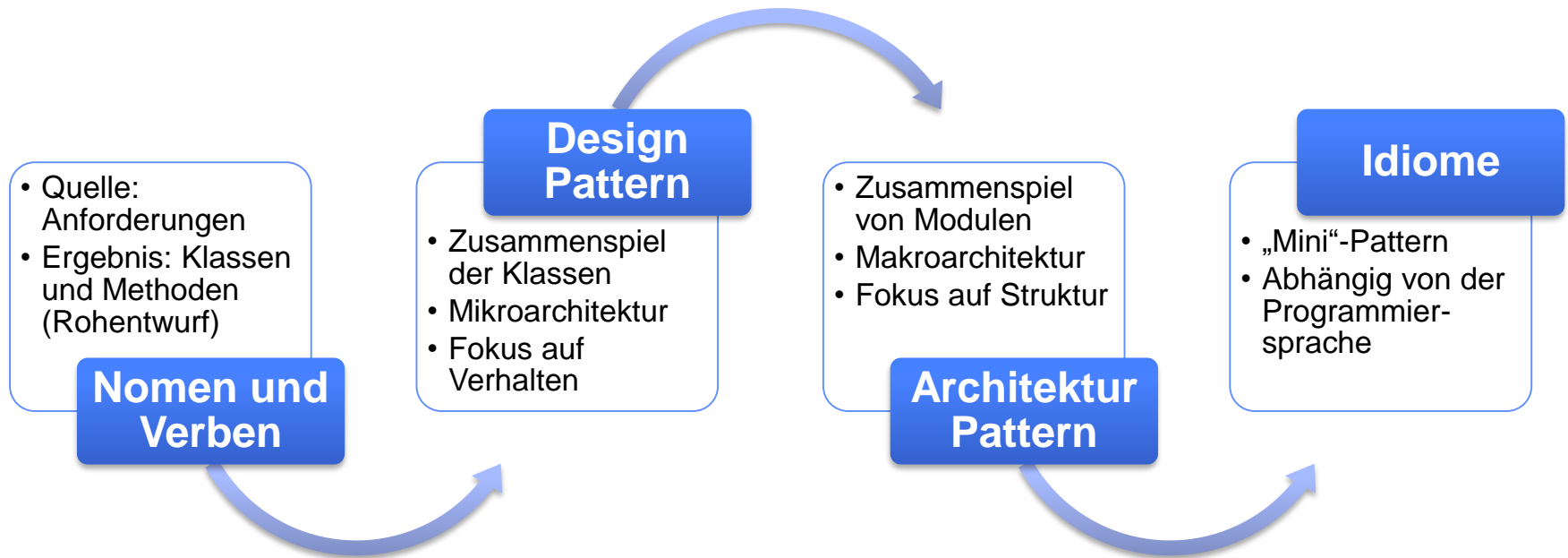


- ❑ **Ein Softwareprojekt ist die Summe seiner funktionalen und nichtfunktionalen Anforderungen.**
- ❑ **Nichtfunktionale Anforderungen (NFA):** Beschreiben Qualitätskriterien (z.B. Zuverlässigkeit, Benutzbarkeit).
 - Beeinflussen maßgeblich die Architektur im „Großen“ (Makroarchitektur: Komponenten, Systeme, Services, Protokolle, etc.).
 - Wirken sich, unter anderem, auf die Zielplattform, der Art der Anwendung (z.B. Webapplikation, Rich-Client-Anwendung) und die verwendeten Technologien aus.
- ❑ **Funktionale Anforderungen:** Benötigte Funktionalität eines Systems (z.B. Verarbeitung von Eingaben, Erstellen von Ausgaben).
 - Beeinflussen maßgeblich die Architektur im „Kleinen“ (Mikroarchitektur: Klassen, Methoden, Algorithmen, etc.).

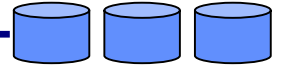
3.2 Grundlagen: Von der Anforderung zur Architektur 3



□ Grundlegendes Vorgehensmodell um Architekturen zu entwerfen:



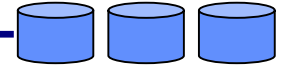
□ Besuchen Sie das Architektur-Tutorial für zusätzliche Details und Tipps (auch zu obigen Vorgehensmodell).



Grundlagen

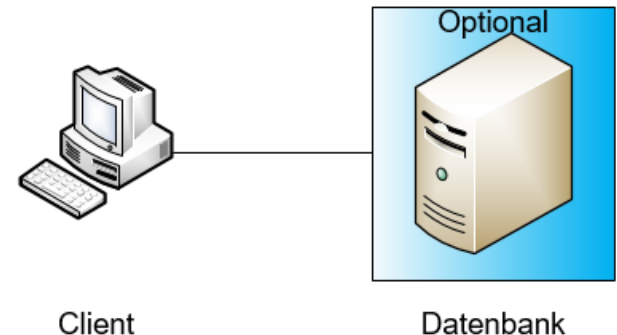
SOFTWAREARCHITEKTUR IM WANDEL DER ZEIT

3.2 Grundlagen: Softwarearchitektur im Wandel der Zeit 1

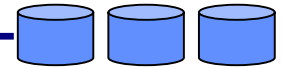


□ Seit den 1960ern (und auch heute noch) kam/kommt es jede Dekade zu disruptiven Umbrüchen

- **60er und 70er Jahre:** Prozedurale Programmiersprachen wie z.B. Basic, C und Pascal.
 - ◆ Entwicklung erfolgt hauptsächlich für teure Großcomputer.
 - ◆ Erst gegen Ende der 70er Jahre: Erweiterung auf Heimcomputer und Spielkonsolen.
- **80er Jahre:** Objektorientierte Programmiersprachen wie z.B. C++, Java und C#
 - ◆ Objektorientierte Sprachen erleichtern die Entwicklung von modularen Systemen.
- **90er Jahre:** Entwicklung von UML und dadurch die Möglichkeit komplexe Systeme grafisch zu beschreiben.
- **Bis frühe 2000er Jahre:** Die Entwicklung von Rich-Client Applikationen dominiert den Software-Markt.
 - ◆ Komplette Geschäftslogik und alle Funktionalität der Anwendung wird in die Desktop-Applikation verlagert.
 - ◆ Backendsysteme wie z.B. ein Datenbank-Server sind meistens optional.
 - ◆ Oft wird eine klassische Client/Server Architektur verwendet.



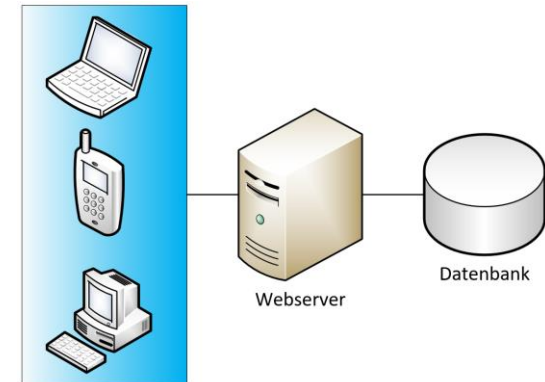
3.2 Grundlagen: Softwarearchitektur im Wandel der Zeit 2



□ Seit den 2000er Jahren verwendete Architekturen:

- **Webapplikationen:** Die Einführung von PHP, Java Server Pages und der Siegeszug des Internets ermöglichte Applikationen mit ähnlicher Funktionalität wie Rich-Client-Anwendungen komplett in einem Webbrowser ablaufen zu lassen.

- ◆ Vorteile: Keine Installation mehr notwendig. Zugriff auf Anwendungen ist von „überall“ möglich.



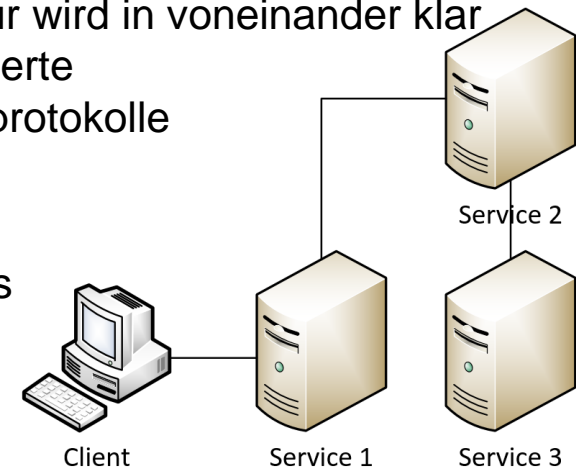
- **Service Oriented Architecture (SOA):** Die Architektur wird in voneinander klar getrennte Komponenten aufgebrochen, die über definierte Schnittstellen mithilfe verschiedener Kommunikationsprotokolle miteinander kommunizieren.

- ◆ Wird vor allem bei Großprojekten eingesetzt.

- **Model Driven Architecture (MDA):** Software wird aus strukturierten Modellen (z.B. UML) erzeugt.

- ◆ Vor allem im akademischen Forschungsbereich von Interesse.

- ◆ Idee: Aus Modellen entsteht direkt und automatisch ausführbare Software ohne Code händisch zu erstellen.

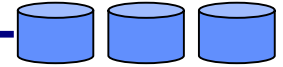




Grundlagen

ARCHITEKTURBAUSTEINE UND DEREN KOMMUNIKATION

3.2 Grundlagen: Architekturbausteine

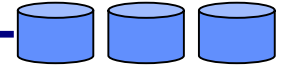


- ❑ **Architekturbausteine:** Auf der Makroebene besteht jede Architektur aus Bausteinen (building blocks) – Module und Services
 - Sind für unterschiedliche Aufgaben verantwortlich und kommunizieren miteinander.
 - „Unnötige“ Details, z.B., Infrastrukturdienste (Firewalls, etc.), werden auf dieser Ebene weggelassen. Das Hauptaugenmerk liegt auf der Kommunikation und den Schnittstellen zwischen Modulen und Services.
- ❑ **Arten:** Es existieren generische und spezifische Bausteine.

Gängigste generische serverbasierte Bausteine	
Webserver	Portalserver
Applikationsserver	Datenbankserver
Nachrichtenserver(Message Queue)	Dokumenten-Management-System (DMS)
Transaktionsserver	Verzeichnisserver (Directory)
Stapelverarbeitung (Batch Processing)	Public Key Infrastructure (PKI)

- **Spezifische Bausteine:** Sind in der jeweiligen Domäne relevant z.B. Clearing & Billing Server.

3.2 Grundlagen: Kommunikation 1



- ❑ **Kommunikation:** Die Kommunikation (Art, Ausprägung, etc.) zwischen Client und Server, Servern, Services und Modulen spielt für die Architektur eine entscheidende Rolle.
- ❑ **Kurzübersicht: Kommunikationscharakteristika**
 - **Direktionalität:** Die Kommunikation zwischen zwei Komponenten kann entweder unidirektional oder bidirektional erfolgen.
 - **Zustandsbasiertheit:** Die Kommunikation kann entweder zustandsbasiert (stateful) oder zustandslos (stateless) erfolgen.
 - ◆ Zustandsbasiert: Nachrichten bauen aufeinander auf. Die Kommunikationspartner speichern Information über die Kommunikation während der gesamten Kommunikationsdauer (z.B. FTP).
 - ◆ Zustandslos: Nachrichten werden von einander unabhängig behandelt (z.B. HTTP).
 - **Zugriffsart:** Wie erhalten die einzelnen Komponenten neue Informationen?
 - ◆ Request/Response-Muster (wichtigste Zugriffsart): Anfragen werden mit relevanten Informationen beantwortet und lösen zumeist auch Aktionen aus.
 - ◆ Push-Kommunikation: Informationen werden von der Quelle an Interessenten verteilt.
 - ◆ Pull-Kommunikation: Interessenten fordern selbst Informationen von der Quelle ein.

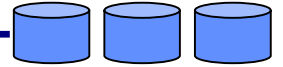


□ **Kurzübersicht: Kommunikationscharakteristika – Fortsetzung**

- **Synchronität:** Muss der Sender auf eine Antwort warten?
 - ◆ **Synchron:** Nach dem Senden einer Anfrage wartet der Sender (blockierend) auf die Antwort (oft einfacher umzusetzen/einzubinden).
 - ◆ **Asynchron:** Nach dem Senden einer Anfrage setzt der Sender die Programmausführung fort und verarbeitet die Antwort ereignisgesteuert sobald diese eintrifft (höhere Performance, allerdings aufwändiger umzusetzen).
- **Standardkonformität:** Kommunikation erfolgt konform zu standardisierten offenen oder proprietären Protokollen (siehe z.B. RFCs).
- **Absicherung:** Gesicherte Kommunikation zwischen zwei Architekturbausteinen.
 - ◆ Sicherung ist Teil des Protokolls, oder
 - ◆ Protokoll wird zur Sicherung in ein entsprechendes Trägerprotokoll gekapselt (z.B.: https).
- **Beispiel:** Vergleichen Sie z.B. HTTP/1.1 mit HTTP/2

□ **Grundregel, um Kommunikation zu vereinfachen:**

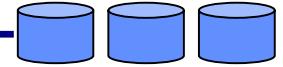
- **„Be strict in what you send and be liberal in what you accept“**
- **Ziel:** Kommunikation möglichst einfach an neue Herausforderungen anpassen zu können ohne bestehenden Software ändern zu müssen.



Grundlagen

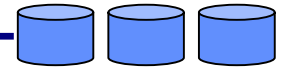
ARCHITEKTUR-FRAMEWORKS

3.2 Grundlagen: Architektur-Frameworks

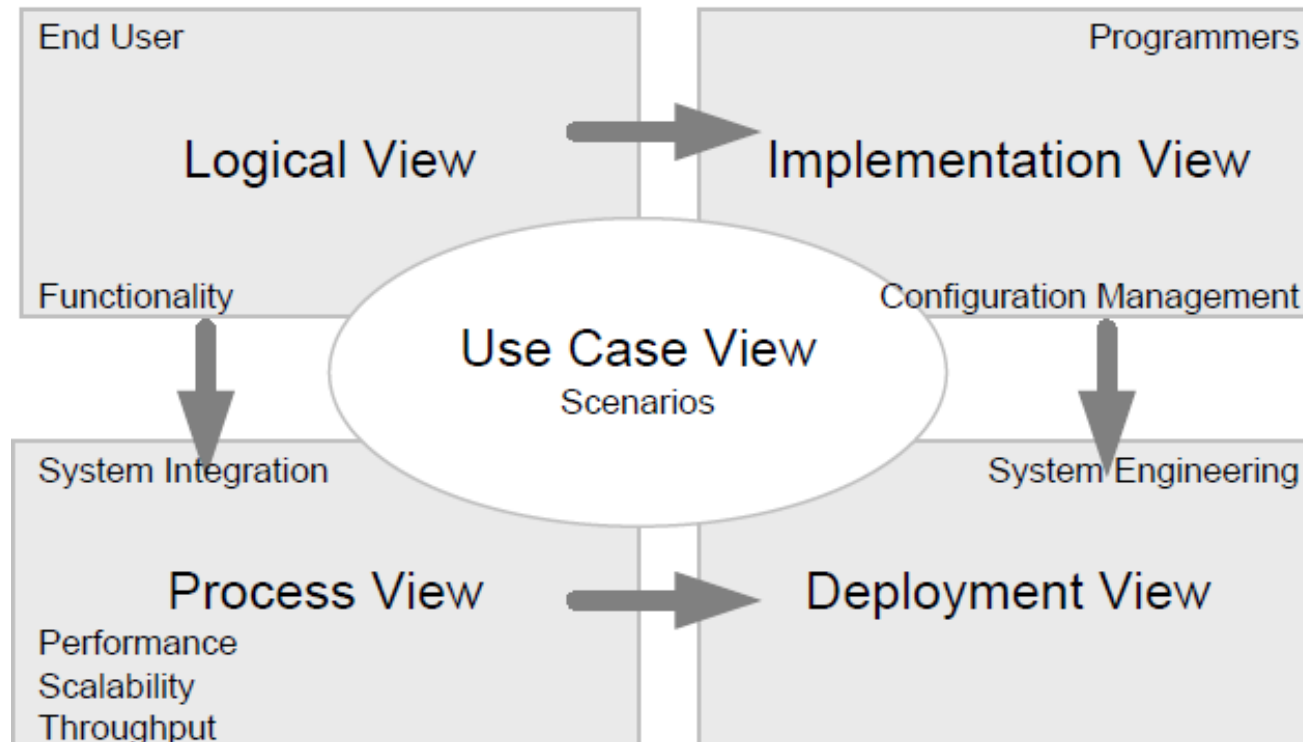


- ❑ **Architektur-Frameworks:** „Fertigteil-Bausätze“ um Softwarearchitekturen zu entwerfen und zu dokumentieren
 - **Ziel:** Unterteilen einer Softwarearchitektur in verschiedene Sichten.
 - **Vorteile:** Liefern Hinweise auf allgemeine Problemstellungen und verringern die Wahrscheinlichkeit, dass wichtige Bestandteile vergessen werden.
 - Unterschiedliche Stakeholder, die im Rahmen der Entwicklung eines Software-Projekts involviert sind, haben unterschiedliche Sichtweisen auf die konkrete Umsetzung. Die Architektur (und das zugehörige Architektur-Framework) ist dabei das (standardisierte) Kommunikationsmedium zwischen den Stakeholdern.
- ❑ **Bekannte Architektur-Frameworks**
 - *4+1 Sichtenmodell (4+1 View Model)*
 - Zachman-Framework
 - DODAF (Department of Defense Architecture Framework)
 - MODAF (British Ministry of Defence Architecture Framework)
 - TOGAF (The Open Group Architecture Framework)
 - RM-ODP (Reference Model of Open Distributed Processing)

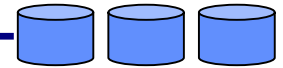
3.2 Grundlagen: Architektur-Frameworks – 4+1 Sichtenmodell 1



- ❑ **4+1 Sichtenmodell:** Jede Sicht beschreibt einen Teilaspekt der Architektur, des Designs und die spezifischen Eigenschaften eines Systems. Sind die Grundlage für die Entwicklung einer Software.
 - Übersichtliche Darstellung mittels Diagrammen aus der UML-Diagramm-Familie.

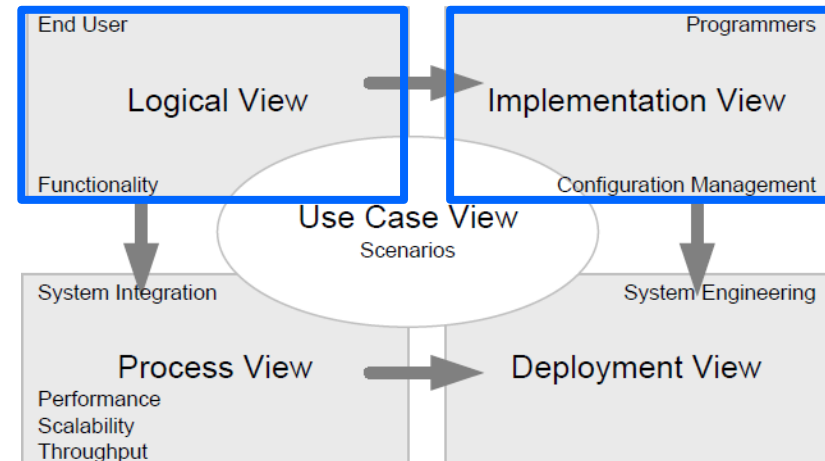


3.2 Grundlagen: Architektur-Frameworks – 4+1 Sichtenmodell 2



❑ **Logische Sicht (Logical View) oder Kontextsicht:** Beschreibt funktionale Anforderungen der Endbenutzer.

- Verwendung von Klassendiagrammen, Zustandsautomaten oder Package-Diagrammen
- Beispiele: Design Packages, Subsysteme, Klassen



❑ **Implementierungssicht (Implementation View):** Beschreibt den statischen Zusammenhang der Module.

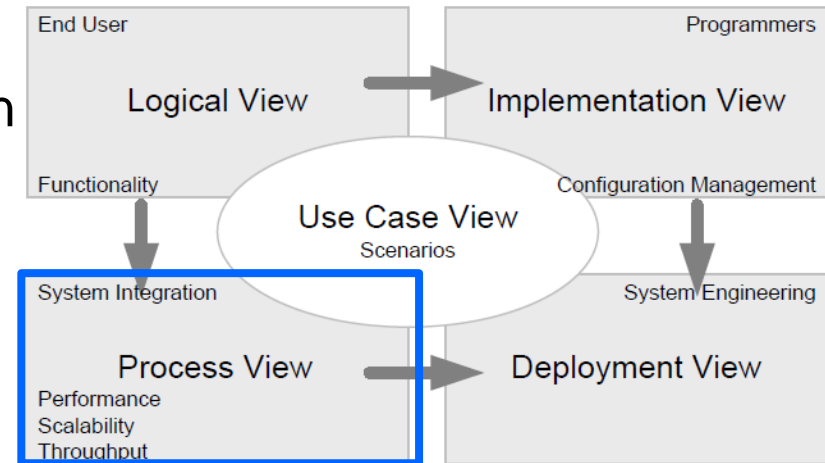
- Sicht des Programmierers ⇨ Kommt dem Sourcecode am nächsten.
- Verwendung von Komponentendiagrammen und Klassendiagrammen.
- Beispiele: Configuration Management, Source Code

3.2 Grundlagen: Architektur-Frameworks – 4+1 Sichtenmodell 3



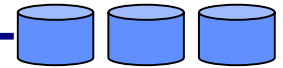
❑ **Prozesssicht (Process View):** Der Fokus liegt auf dem Laufzeitverhalten im Rahmen der Systemintegration.

- Ist vor allem für die Umsetzung von nichtfunktionalen Anforderungen relevant.
- Verwendet Komponenten aus der Implementierungssicht.
- Beschreibt wie diese Komponenten kommunizieren bzw. wie deren Zustände in bestimmten Arbeitsabläufen sind.
- Verwendung von Sequenz-, Aktivitäts-, oder Kommunikationsdiagrammen.
- Beispiele: Laufzeitbedingungen wie Concurrency, Lastverteilung und Fehlertoleranz



❑ **Im Folgenden werden wir uns vor allem dieser und den vorangegangenen beiden Sichten (Logische-Sicht und Implementierungssicht) widmen.**

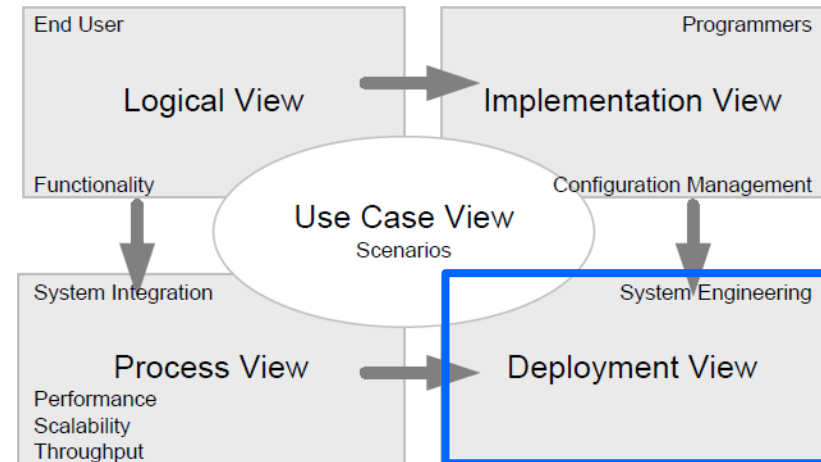
3.2 Grundlagen: Architektur-Frameworks – 4+1 Sichtenmodell 4



❑ **Verteilungssicht**

(Deployment View): Beschreibt die ausführbare Applikation unter Berücksichtigung der jeweiligen Plattform und der für die Anwendung notwendigen Ressourcen.

- Protokolle für die Kommunikation der einzelnen Bestandteile werden ebenfalls angeführt (wie etwa HTTP, JMS, TCP, etc.)
- Fokus auf System Engineers.
- Verwendet UML Deployment-Diagramme.
- Beispiele: Deployment, Installation, Performance
- Siehe, Block 6: Inbetriebnahme, Rollout und Wartung

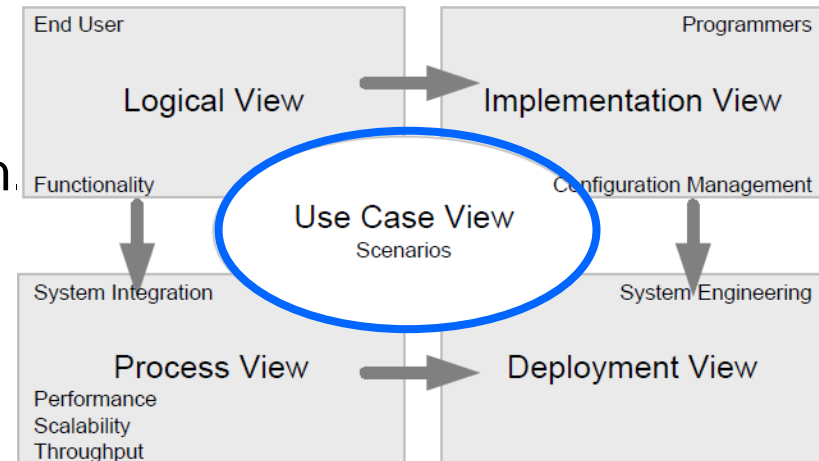


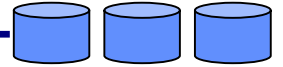
3.2 Grundlagen: Architektur-Frameworks – 4+1 Sichtenmodell 5



□ **Anwendungsfallsicht (Use-Case View):** Dient als Ergänzung zu den bisherigen Sichten.

- Gemeinsamer Nenner, in dem die Anwendungsfälle und Aktivitäten beispielsweise als Szenarien abgebildet werden.
- Fokus auf Systemanalyse sowie Entwurf und Design.
 - ◆ Schnittstellenfunktion zwischen den andern Sichten aus der Architektursicht.
 - ◆ Beinhaltet Schlüsselszenarien der Applikation aus Geschäftsprozesssicht.
- Fokus auf Implementierung und „Transition“.
 - ◆ Kann zu Verifikations- und Validierungszwecken verwendet werden.
- Verwendet Anwendungsfalldiagramme und entsprechende Beschreibungen sowie Aktivitätsdiagramme.
- Siehe, Block 2: Analyse und Anforderungserhebung





3.1 Motivation

3.2 Grundlagen

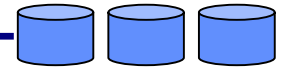
3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

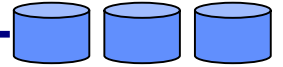
3.5 Serviceorientierte Architektur

3.6 Zusammenfassung

3.3 Entwurfsparadigmen



- ❑ **Es existieren mehrere Ansätze für den Entwurf einer Applikation.**
- ❑ **Abhängig von den Anforderungen kann es vorteilhafter sein auf eine objektorientierte oder eine serviceorientierte Architektur zu setzen. Die jeweiligen Vorteile werden im Folgenden diskutiert.**
- ❑ **Vergleich: Serviceorientiert vs. objektorientiert**
 - **Serviceorientierter Entwurf/Entwicklung:** Service- und Prozessbeschreibung steht beim Entwurf im Mittelpunkt.
 - ◆ Basierend auf den Geschäftsprozessen des Kunden wird die Software in Services unterteilt und diese getrennt von einander entwickelt und installiert.
 - ◆ Siehe: Serviceorientiertes Design und SOA
 - **Objektorientierter Entwurf/Entwicklung:** Klassen und deren Zusammenarbeit (vgl., Klassen- und Sequenzdiagramme) stehen beim Entwurf im Mittelpunkt.
 - ◆ Die fachlichen Komponente fließt in Form eines Domänenmodells und der Anwendungsfälle mit ein.
 - Zumeist werden beide Ansätze kombiniert, beispielsweise werden zum Entwurf von Services Methoden des objektorientierten Entwurfs eingesetzt.



Entwurfsparadigmen

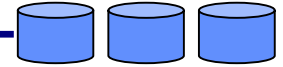
OBJEKTORIENTIERTES DESIGN

3.3 Entwurfsparadigmen: Objektorientiertes Design 1



- ❑ **Objektorientiertes Design (objektorientierter Entwurf):** Übernimmt und erweitert Diagramme und Modelle der objektorientierten Analyse.
 - **Ziel:** Erstellung einer möglichst genauen Abbildung des später zu implementierenden Systems.
 - **Relevante Diagramme:**
 - ◆ Klassendiagramme: Legen die Struktur (Klassen) fest.
 - ◆ Zustands- und Sequenzdiagramme: Geben Verhalten des Systems und Abläufe vor.
 - ◆ Komponentendiagramme: Bilden Systemkomponenten und ihren Zusammenhang ab.
 - **Notation:** Unified Modeling Language (UML).
 - **Vorgehensweise (Tasks innerhalb der Mikroarchitektur):**
 - ◆ Identifizieren der Klassen und Objekte auf jeder Abstraktionsebene.
 - ◆ Festlegen der Semantiken zwischen den Klassen und Objekten.
 - ◆ Identifizieren der Beziehungen zwischen Klassen und Objekten.
 - ◆ Spezifizieren der Schnittstellen und Implementationen der Klassen.
 - Tasks finden während der Analyse aber auch im Entwurf statt und liefern abhängig davon ob sie während der Analyse oder im Entwurf stattfinden andere Ergebnisse.

3.3 Entwurfsparadigmen: Objektorientiertes Design 2



❑ **Objektorientiertes Design (objektorientierter Entwurf):** Fortsetzung

- Spielt sich vor allem auf zwei Ebenen ab:
 - ◆ Mikroarchitektur: Klassen und Methoden und wie diese kommunizieren.
 - ◆ Makroarchitektur: Gruppen von Klassen (Module) und wie diese kommunizieren.
- Für beide Ebenen gibt es ähnliche Ziele, beispielsweise:
 - ◆ Kommunikation: Nur über wenige definierte Schnittstellen, dies erleichtert es Änderungen durchzuführen. Negativbeispiel: Jede Klasse (Modul) kommuniziert mit jeder anderen Klasse (Modul), jede Änderung beeinflusst auch alle anderen Klassen (Module).
 - ◆ Funktionen: Jede(s) Modul/Klasse sollte nur Aspekte genau einer Funktion abdecken. Beispiel: Alle Methoden einer Klasse sollen die gleichen globalen Variablen bearbeiten.

- ### ❑ **Designprinzipien:** Sind über Jahre gereifte Tipps, Tricks und Empfehlungen um Entwicklungsaufwand zu reduzieren und die Wartung/Erweiterung von Software zu erleichtern.

Designprinzipien für den objektorientierten Entwurf	
Abstraktion	Kapselung (encapsulation)
Kopplung und Kohäsion	Kontrolle
Dekomposition und Modularisierung	

3.3 Entwurfsparadigmen: Objektorientiertes Design – Designprinzipien 1



- ❑ **Abstraktion:** Ausblenden irrelevanter Details und Zusammenfassen von Gemeinsamkeiten.

- Beispielsweise abstrakte Datentypen.

Designprinzipien für den objektorientierten Entwurf

Abstraktion

Kapselung
(encapsulation)

Kopplung und Kohäsion

Kontrolle

Dekomposition und Modularisierung

- ❑ **Kapselung (encapsulation) und Information Hiding:** Möglichst viele Details nach außen verstecken.

- Nur die notwendige(n) Funktionalität und Eigenschaften bereitstellen.
 - Private Methoden ⇔ Öffentliche Methoden
 - Kommunikation erfolgt über Schnittstellen.
 - Zentrale Rolle bei der komponentenorientierten Softwareentwicklung: Trennung von Schnittstelle (Sicht von außen) und der inneren Realisierung (Implementierung).
 - ◆ Ziel: Die interne Implementierung kann geändert werden ohne die tatsächliche Verwendung (von außen) zu beeinflussen.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Designprinzipien 2



□ Kopplung und Kohäsion

- **Kopplung (coupling):** Beschreibt die Abhängigkeit zwischen den einzelnen Komponenten
z.B.: Anzahl der gegenseitigen Methodenaufrufe.

- ◆ Hohe Kopplung bedeutet eine hohe Abhängigkeit zu anderen Klassen (Modulen): Eine hohe Anzahl an gegenseitigen Methodenaufrufen ⇒ Ziel ist aber eine niedere Kopplung.
- ◆ Vorteile einer niederen Kopplung: Änderungen an einer Komponente wirken sich nur auf eine beschränkte (minimale) Anzahl von anderen Komponenten aus.
⇒ Komponenten können besser wiederverwendet, getestet und gewartet werden.

- **Kohäsion (cohesion):** Maß für den inneren Zusammenhalt einer Komponente.
 - ◆ Zusammengehörige Funktionalität sollte in die gleiche Komponente (z.B., Klasse) gepackt werden ⇒ hohe Kohäsion aber auch höhere Komplexität.
- Wünschenswert ist ein Gleichgewicht zwischen Kopplung und Kohäsion.
- Sequenzdiagramme ermöglichen es Kopplung und Kohäsion zu visualisieren.
 - ◆ Komplexe Diagramme: Hohe Kopplung und niedrige Kohäsion.
 - ◆ Die Qualität dieses Aspektes lässt sich berechnen mit LCOM4.

Designprinzipien für den objektorientierten Entwurf

Abstraktion

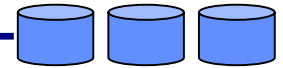
Kapselung
(encapsulation)

Kopplung und Kohäsion

Kontrolle

Dekomposition und Modularisierung

3.3 Entwurfsparadigmen: Objektorientiertes Design – Designprinzipien 3



□ Kontrolle am Beispiel Stair vs. Fork

○ „Fork“-Ansatz: Zentrale Kontrolle

- ◆ Zentrales Objekt übernimmt die Kontrolle ⇒ Der gesamte Ablauf wird in diesem Objekt gesteuert.
- ◆ Vorteile: Änderungen und Anpassungen müssen nur an einem Objekt oder sehr wenigen Objekten vorgenommen werden ⇒ Wartbarkeit wird erhöht.
- ◆ Nachteile: Zentrales Element fällt aus ⇒ Gesamtes System kommt zum Stillstand außerdem Vereinigung des gesamten Systemwissens in einer Komponente ⇒ Große komplexe Kontrollobjekte .

○ „Stair“-Ansatz: Verteilt und dezentral, also größtenteils lokale Kontrolle.

- ◆ Kontrolle wird auf die beteiligten Objekte aufgeteilt ⇒ Jeder Schritt des Ablaufs wird von einem eigenen Objekt gesteuert.
- ◆ Vorteile: Teile des Systems fallen aus ⇒ Restliche Teile des Gesamtsystems funktionieren, Systemwissen wird auf unterschiedliche Komponenten verteilt und Wiederverwendbarkeit wird erhöht. Letzteres, da viel Funktionalität in den jeweiligen Teilen gekapselt wird und so ganze Lösungsblöcke übernommen werden können.
- ◆ Nachteile: Für eine einzelne Änderungen muss man oft mehrere Stellen im Code anfassen ⇒ dieses Vorgehen ist fehleranfälliger, es leidet auch die Wartbarkeit.

Designprinzipien für den objektorientierten Entwurf

Abstraktion

Kapselung
(encapsulation)

Kopplung und Kohäsion

Kontrolle

Dekomposition und Modularisierung

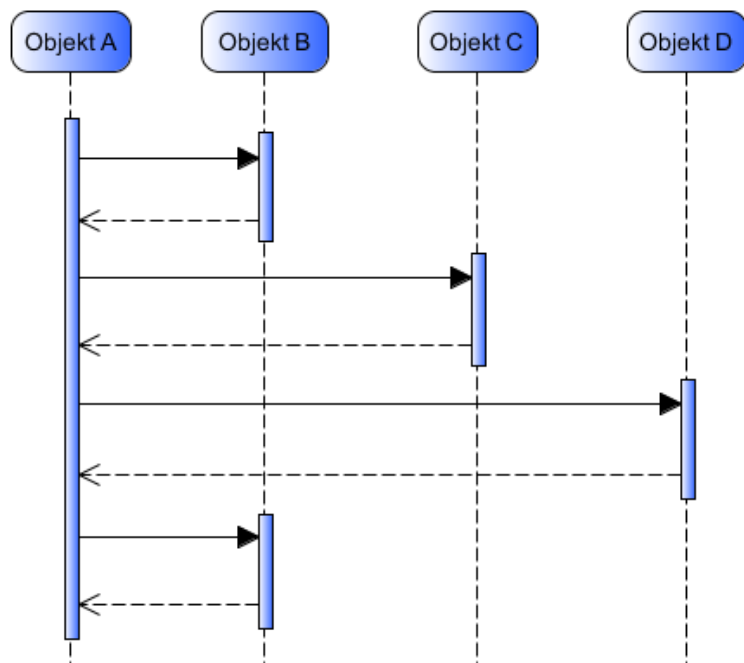
3.3 Entwurfsparadigmen: Objektorientiertes Design – Designprinzipien 4



□ Kontrolle am Beispiel Stair vs. Fork

- Grafische Darstellung

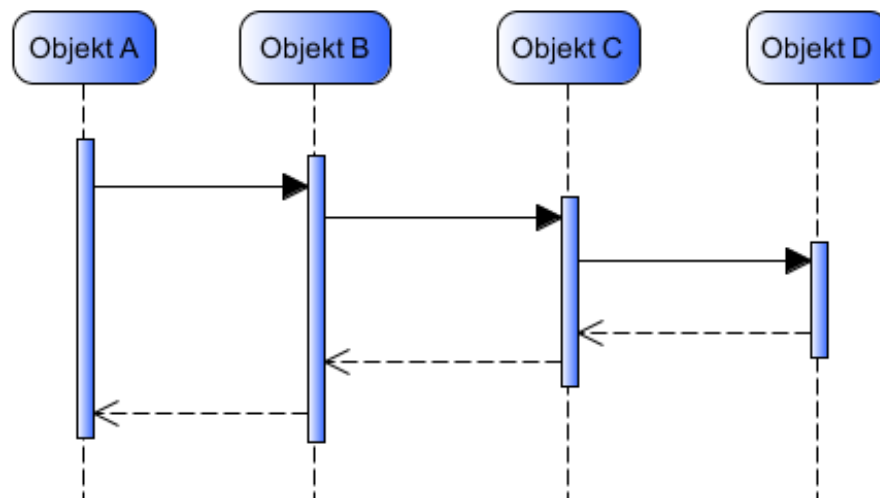
Sequenzdiagramm: Fork



Designprinzipien für den objektorientierten Entwurf

Abstraktion	Kapselung (encapsulation)
Kopplung und Kohäsion	Kontrolle
Dekomposition und Modularisierung	

Sequenzdiagramm: Stair



3.3 Entwurfsparadigmen: Objektorientiertes Design – Designprinzipien 5



❑ **Dekomposition und Modularisierung:** Komplexität beherrschbar machen.

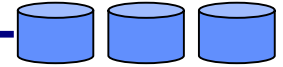
- Zerlegen von großen komplexen Lösungsteilen in kleine unabhängige, einfache Teile (Teilprobleme) mit zusammengehöriger Funktionalität (*divide and conquer*).
- Vorteile:
 - ◆ Jedes Teilproblem hat eine relativ geringer Komplexität und ist einfach umzusetzen.
 - ◆ Erleichtert das wiederverwenden von bereits fertigen Teillösungen ⇒ Beschleunigt die Entwicklung von Software (Praktisches Beispiel: SOA).
 - ◆ Unterstützung der Entwickler bei der Entwicklung und Änderung von Komponenten sowie bei der Wartung (z.B., jedes Teilproblem lässt sich einzeln und unabhängig testen).
- Zu beachten ist jedoch:
 - ◆ Geeignete Kontrollmechanismen (zentrale oder verteilte Kontrolle) sind notwendig.
 - ◆ Zusammenhalt der Komponenten (Kohäsion).
 - ◆ Abhängigkeit zu anderen Komponenten (Kopplung).
 - ◆ Betrifft beide Ebenen: Makroarchitektur (Klassen/Module), Mikroarchitektur (Methoden).

Designprinzipien für den objektorientierten Entwurf

Abstraktion	Kapselung (encapsulation)
Kopplung und Kohäsion	Kontrolle

Dekomposition und Modularisierung

3.3 Entwurfsparadigmen: Objektorientiertes Design – UML 1



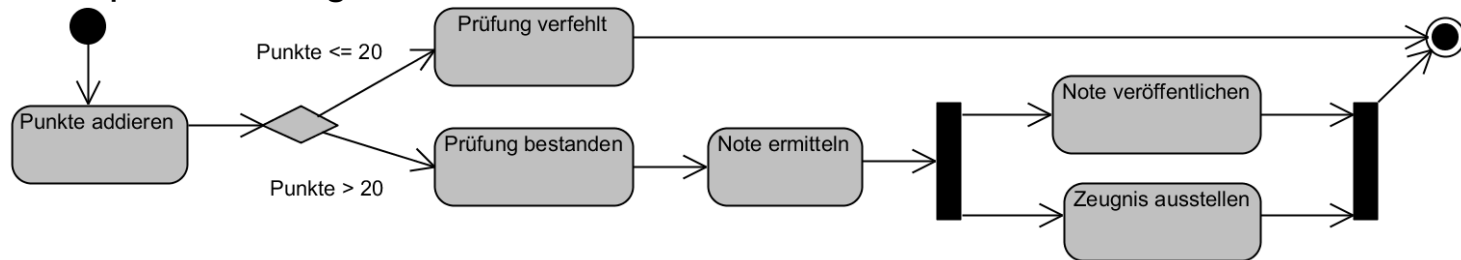
- ❑ **UML:** Unterstützung des objektorientierten Entwurfs durch die Unified Modeling Language (UML)
 - Sammlung von Modellierungssprachen, entwickelt und standardisiert durch die OMG.
 - Ausgelegt auf komplexe und große Systeme.
 - Fokussiert sich auf objekt-orientierte Entwicklung und Design.
 - Ermöglicht/Erleichtert die Dokumentation und Kommunikation während aller Softwareentwicklungsphasen.
- ❑ **Ausgewählte Modelltypen gegliedert nach zwei Modellarten**
 - **Verhaltensmodelle:** Beschreiben dynamisches Verhalten.
 - ◆ Aktivitätsdiagramme, Zustandsdiagramme, Sequenzdiagramme
 - **Strukturmodelle:** Beschreiben statische Teile des Systems.
 - ◆ Klassendiagramm, Komponentendiagramm, Verteilungsdiagramm

3.3 Entwurfsparadigmen: Objektorientiertes Design – UML 2



❑ Beispiel für ein Verhaltensmodell: Zustandsdiagramm

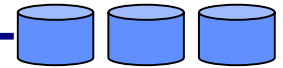
- Zur Modellierung von dynamischen Verhalten.
- Beschreibt mögliche Systemzustände und bildet Zustandsänderungen sowie auslösende Ereignisse ab.
- Abbildung in Form eines endlichen Zustandsautomaten.
- Beispiel: Prüfung bewerten



❑ Beispiel für ein Strukturmodell: Klassendiagramm

- Zur Modellierung von statischen Zusammenhängen.
- Beschreibt die Struktur eines Systems mittels Klassen und Methoden.
- Klasse: Zusammenfassung gleicher Objekte in Bezug auf deren Eigenschaften und Fähigkeiten.

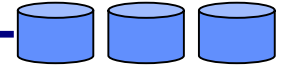
3.3 Entwurfsparadigmen: Objektorientiertes Design – UML 3



- ❑ **Literaturempfehlung zu Unified Modeling Language (UML):**
Kecher, Christoph. *UML 2.5: Das umfassende Handbuch*; [UML lernen und effektiv in Projekten einsetzen; alle Diagramme und Notationselemente im Überblick; mit zahlreichen Praxisbeispielen in Java und C#; inkl. Poster mit allen Diagrammtypen]. Rheinwerk Verlag, 2015.
- ❑ Zeigt gut die Relation zwischen UML Modellen und Source Code.



3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 1



- ❑ **Beispiel:** Modellierung eines vereinfachten Modells der Universität Wien entsprechend der Anforderungen mittels UML Klassendiagramm.

- **Anforderungen – Beschreibung:**

- ◆ Die Universität Wien besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter geleitet.
- ◆ Die Gesamtanzahl der Mitarbeiter ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.
- ◆ Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.
- ◆ Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt.
- ◆ Weiters können wissenschaftliche Mitarbeiter an Projekten beteiligt sein, von welchen ein Name und Anfang- sowie Enddatum bekannt sind.
- ◆ Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet. Lehrveranstaltungen haben eine ID, einen Namen und eine Stundenzahl.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 2



- ❑ **Beispiel:** Ermittlung potentieller Klassen anhand der Anforderungen, Fokus hierbei auf Nomen legen. Ergebnis mehrfach durchdenken.
 - **Anforderungen – Identifikation von Klassen:**
 - ◆ Die Universität Wien besteht aus mehreren **Fakultäten**, die sich wiederum aus verschiedenen **Instituten** zusammensetzen. Jede Fakultät und jedes Institut besitzt eine Bezeichnung. Für jedes Institut ist eine Adresse bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter geleitet.
 - ◆ Die Gesamtanzahl der **Mitarbeiter** ist bekannt. Mitarbeiter haben eine Sozialversicherungsnummer, einen Namen und eine E-Mail-Adresse. Es wird zwischen **wissenschaftlichem** und **nicht-wissenschaftlichem Personal** unterschieden.
 - ◆ Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.
 - ◆ Für jeden wissenschaftlichen Mitarbeiter ist seine Fachrichtung bekannt.
 - ◆ Weiters können wissenschaftliche Mitarbeiter an **Projekten** beteiligt sein, von welchen ein Name und Anfang- sowie Enddatum bekannt sind.
 - ◆ Manche wissenschaftliche Mitarbeiter führen **Lehrveranstaltungen** durch – diese werden als **Vortragende** bezeichnet. Lehrveranstaltungen haben eine ID, einen Namen und eine Stundenzahl.
 - **Anmerkung:** Potentielle Klassen sind blau und fett hervorgehoben.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 3

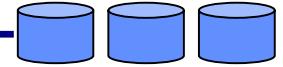


- ❑ **Beispiel:** Ermittlung der finalen Klassen anhand der Anforderungen, Fokus hierbei auf Nomen legen.
 - **Anforderungen – Identifikation von Klassen:**



- **Anmerkung:** UML Diagramm der identifizierten Klassen.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 4

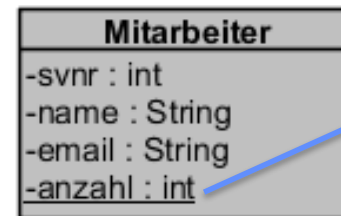
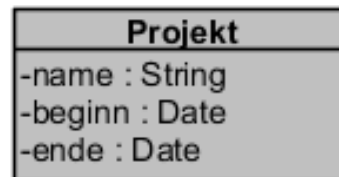
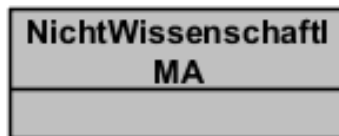
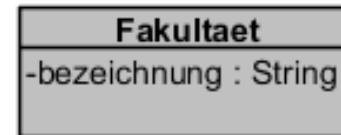
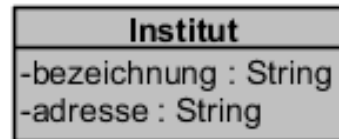


- ❑ **Beispiel:** Potentielle Attribute, Variablen und Methoden identifizieren, hierzu auf Verben fokussieren. Ergebnis mehrfach durchdenken.
 - **Anforderungen – Identifikation von zu den Klassen gehörenden Attributen & Variablen:**
 - ◆ Die Universität Wien besteht aus mehreren Fakultäten, die sich wiederum aus verschiedenen Instituten zusammensetzen. Jede Fakultät und jedes Institut besitzt eine **Bezeichnung**. Für jedes Institut ist eine **Adresse** bekannt. Jede Fakultät wird von ihrem Dekan, einem Mitarbeiter geleitet.
 - ◆ Die **Gesamtanzahl** der Mitarbeiter ist bekannt. Mitarbeiter haben eine **Sozialversicherungsnummer**, einen **Namen** und eine **E-Mail-Adresse**. Es wird zwischen wissenschaftlichem und nicht-wissenschaftlichem Personal unterschieden.
 - ◆ Wissenschaftliche Mitarbeiter sind zumindest einem Institut zugeordnet.
 - ◆ Für jeden wissenschaftlichen Mitarbeiter ist seine **Fachrichtung** bekannt.
 - ◆ Weiters können wissenschaftliche Mitarbeiter an Projekten beteiligt sein, von welchen ein **Name** und **Anfang- sowie Enddatum** bekannt sind.
 - ◆ Manche wissenschaftliche Mitarbeiter führen Lehrveranstaltungen durch – diese werden als Vortragende bezeichnet. Lehrveranstaltungen haben eine **ID**, einen **Namen** und eine **Stundenzahl**.
 - **Anmerkung:** Potentielle Attribute & Variablen sind blau und fett hervorgehoben.

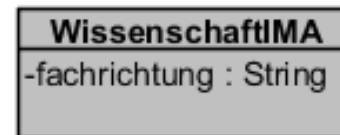
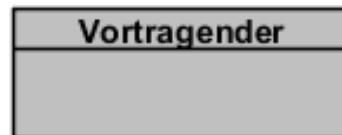
3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 5



- ❑ **Beispiel:** Erweiterung der zuvor ermittelten Klassen anhand der identifizierten Attribute & Variablen.
 - **Anforderungen – Identifikation von zu den Klassen gehörenden Attributen & Variablen:**

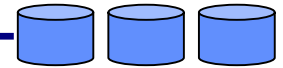


Anzahl wird automatisch mittels einer static Variable erhoben.

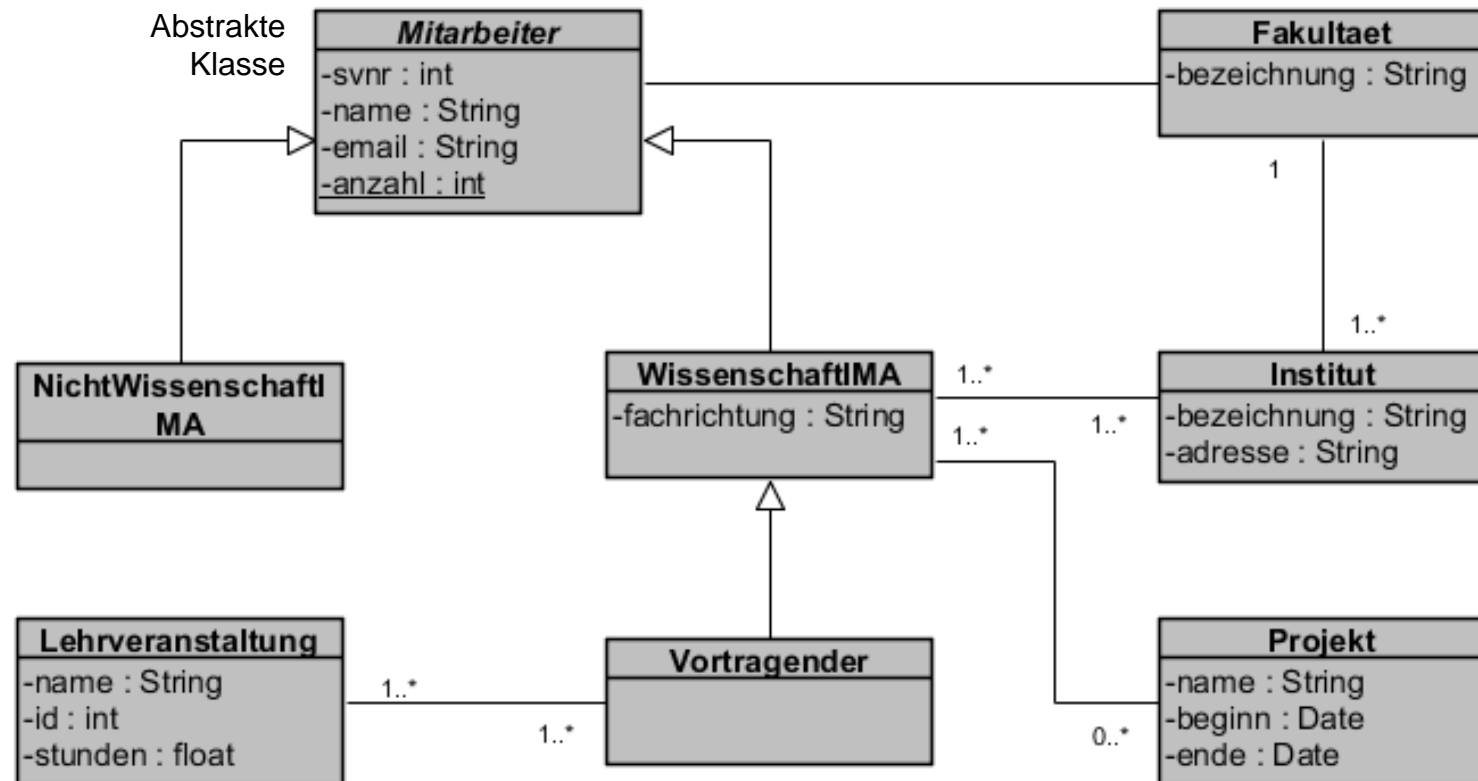


- **Anmerkung:** UML Diagramm der Klassen inklusive Attribute & Variablen.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 6



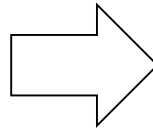
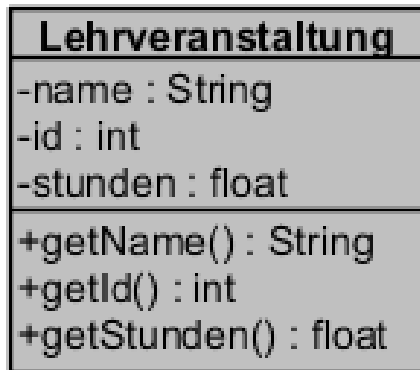
- ❑ **Beispiel:** Entwurf des Klassendiagramms anhand der zuvor ermittelten Klassen und der identifizierten Attribute & Variablen.
 - **Klassendiagramm, inkl. hinzugefügten Relationen & Attributen:**



3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 7



□ **Beispiel:** Übersetzung nach Java – Klassen.

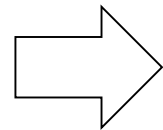
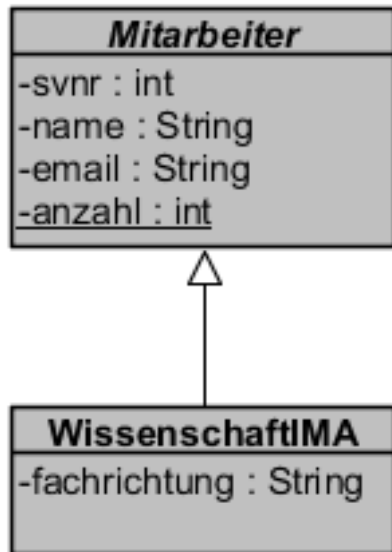


```
public class Lehrveranstaltung {  
    private String name;  
    private int id;  
    private float stunden;  
  
    public Lehrveranstaltung(String name, int id,  
        float stunden) {  
        this.name = name;  
        this.id = id;  
        this.stunden = stunden;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getId() {  
        return id;  
    }  
  
    public float getStunden() {  
        return stunden;  
    }  
}
```

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 8



- ❑ **Beispiel:** Übersetzung nach Java – Abstrakte Klasse, Attribut, Generalisierung und Klassenvariablen.



```
public abstract class Mitarbeiter {
    private int svnr;
    private String name;
    private String email;
    private static int anzahl = 0;

    public Mitarbeiter (int svnr, String name,
        String email) {
        this.svnr = svnr;
        this.name = name;
        this.email = email;
        anzahl++;
    }
}

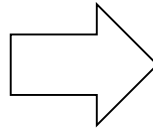
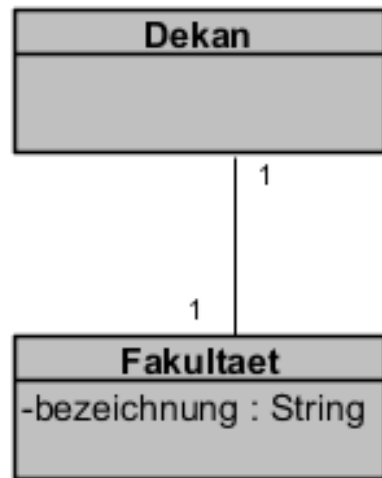
public class WissenschaftlMA extends Mitarbeiter {
    private String fachrichtung;

    public WissenschaftlMA(int svnr, String name,
        String email, String fachrichtung) {
        super(svnr, name, email);
        this.fachrichtung = fachrichtung;
    }
}
```

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 9



- ❑ **Beispiel:** Übersetzung nach Java – 1:1-Assoziation mittels Klassenvariable.



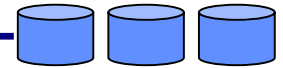
```
public class Dekan {
    private Fakultaet fakultaet;

    public Fakultaet getFakultaet() {
        return fakultaet;
    }
}

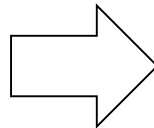
public class Fakultaet {
    private String bezeichnung;
    private Dekan dekan;

    public Dekan getDekan() {
        return dekan;
    }
}
```

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 10



- **Beispiel:** Übersetzung nach Java – 1:*-Assoziation mittels Liste.



```
import java.util.List;
import java.util.ArrayList;

public class Vortragender {
    private List<Lehrveranstaltung> lvas;

    public Vortragender {
        lvas = new ArrayList<>();
    }
}
```

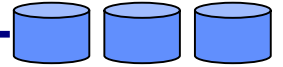
Die Klasse Lehrveranstaltung bleibt unverändert.

3.3 Entwurfsparadigmen: Objektorientiertes Design – Klassendiagramm 11



□ **Beispiel:** Zusammenfassung

- UML Klassen werden in Java-Klassen übersetzt.
- Klassenvariablen und Operationen werden in Java mittels Variablen und Methoden abgebildet.
- Klassenattribute und Klassenoperationen (unterstrichen im Klassendiagramm) werden mit dem Schlüsselwort `static` versehen.
- Assoziationen werden mit Hilfe von Variablen ausgedrückt.
 - ◆ **1:1-Beziehungen:** Eine Variable vom Typ der assoziierten Klasse.
 - ◆ **1:n-Beziehungen:** Benötigt beispielsweise eine `List` wie `ArrayList` mit generischen Typ der verbundenen Klasse. Alternativen wie `Set` oder `Map` bei Bedarf überlegt wählen je nach den benötigten Eigenschaften.
- Einfachvererbung wird in Java direkt unterstützt (`extends`).
 - ◆ Best Practice: Wenn möglich Interfaces und Komposition bevorzugen.
- **Anmerkung:** Getter und Setter wurden zur Erhöhung der Übersichtlichkeit in den vorangegangenen Beispielen nicht abgebildet.



Entwurfsparadigmen

SERVICEORIENTIERTES DESIGN

3.3 Entwurfsparadigmen: Serviceorientiertes Design 1



□ Serviceorientiertes Design (serviceorientierter Entwurf)

- Stülpt dem objektorientierten Entwurf noch zwei weitere Abstraktionsebenen über. Insgesamt entstehen folgende Layer (Ebenen):
 - ◆ Service Layer (Fasst Komponenten zu Services zusammen)
 - ◆ Component Layer (Fasst Klassen zu Komponenten zusammen)
 - ◆ Class Layer (Klassen und Methoden)
- Je nach Abstraktionsebene werden unterschiedliche Notationen werden. Der Wechsel von der fachlichen Realisierung (Service Layer) zu technischer Realisierung (Class Layer) ist fließend.
 - ◆ BPMN: Service Layer (Fokussiert sich auf die Fachexperten, nicht technische Stakeholder)
 - ◆ Komponentendiagramme: Component Layer
 - ◆ UML: Class Layer
- Erstellen der Teil-Anwendungen (Services) mithilfe des objektorientierten Entwurfs.
- Verknüpfung der so erstellten Teile zu einer „großen Anwendung“.
 - ◆ **Services:** Kombination der während des Entwurfs entstanden Klassen/Komponenten.
 - ◆ **Business Services:** „Große Anwendung“.
- Zentrale Grundlage: Geschäftsprozesse nicht Klassen

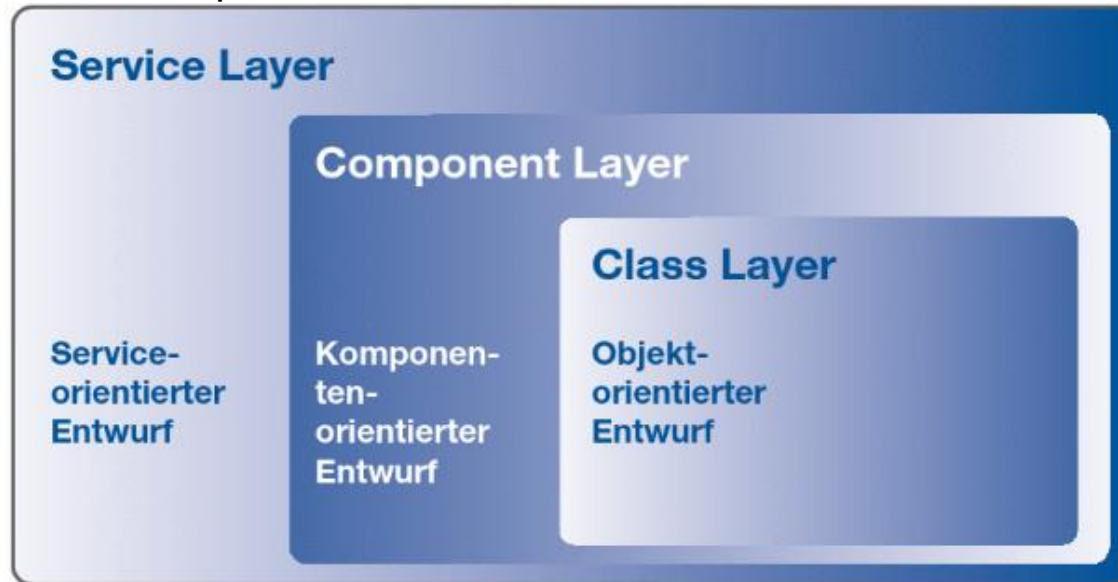
3.3 Entwurfsparadigmen: Serviceorientiertes Design 2



❑ Serviceorientiertes Design (serviceorientierter Entwurf)

○ Grafische Darstellung der Bestandteile

- ◆ Sichtbar wird wie feingranulare Bestandteile (z.B. Klassen) zu größeren Funktionsblöcken (wie Komponenten und Services) kombiniert werden um deren Komplexität zu verstecken und diese größeren Blöcke leichter untereinander integrieren und austauschen zu können. Vergleichbar zu dem von der LV-Leitung bereitgestellten Serverimplementierung welche dessen interne Komplexität hinter einer einfachen REST API „versteckt“.



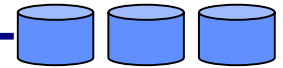
3.3 Entwurfsparadigmen: Serviceorientiertes Design – BPM 1



□ **Service Layer und Business Process Management (BPM)**

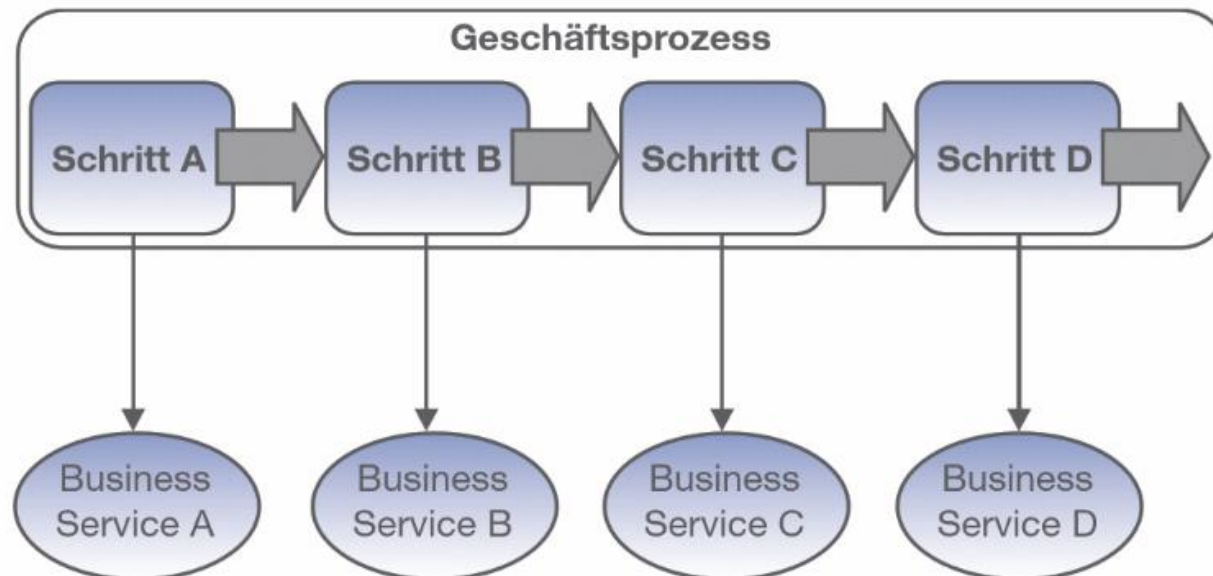
- **Business Services:** Umfassen mehrere prozesstechnische Services, die wiederum durch Orchestrierung aus mehreren atomaren Services zusammengesetzt wurden.
 - ◆ Ein Service, welcher mehrere andere Services kombiniert.
- Identifizieren der Service-Kandidaten anhand der bereits vom Kunden eingesetzten Geschäftsprozesse und der darin abgebildeten Aktivitäten ⇒ Service Layer enthält die technische Realisierung der Business Services.
- Service Litmus-Test aus der IBM Service-Oriented Modeling and Architecture (SOMA): Wesentliche Gesichtspunkte zur Identifikation von Service Kandidaten.
 - ◆ Business-aligned: Rückführbarkeit jedes Service auf entweder ein Business Requirement (z.B. Key Performance Indicator (KPI) oder Geschäftsziel) oder eine Aktivität innerhalb eines Geschäftsprozesses.
 - ◆ Reusable: Die Services sollten generisch genug sein um diese über den ursprünglichen Anwendungskontext hinaus einsetzen zu können: Wiederverwendung/Neukombination der Services in verschiedenen Lösungen bzw. für verschiedene Problemstellungen.

- ### □ **Business Process Management und SOA sind eng miteinander verknüpft:** Anwendungen werden hierbei maßgeblich von der Geschäftsprozess-Modellierung beeinflusst.

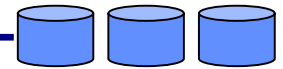


❑ Beispiel für den Zusammenhang von Business Process Management (BPM) und SOA

- Prozessschritte (Aktivitäten) setzen entsprechende Service-Calls an die Business Services ab. Die Services erledigen hierbei die eigentliche Arbeit.
- Hohe Abstraktion: Lässt sich auch an nicht technische Stakeholder kommunizieren.
- Prozessmodelle (wie das Beispiel unten) definieren welche Services in welchen Reihenfolgen aufgerufen werden.

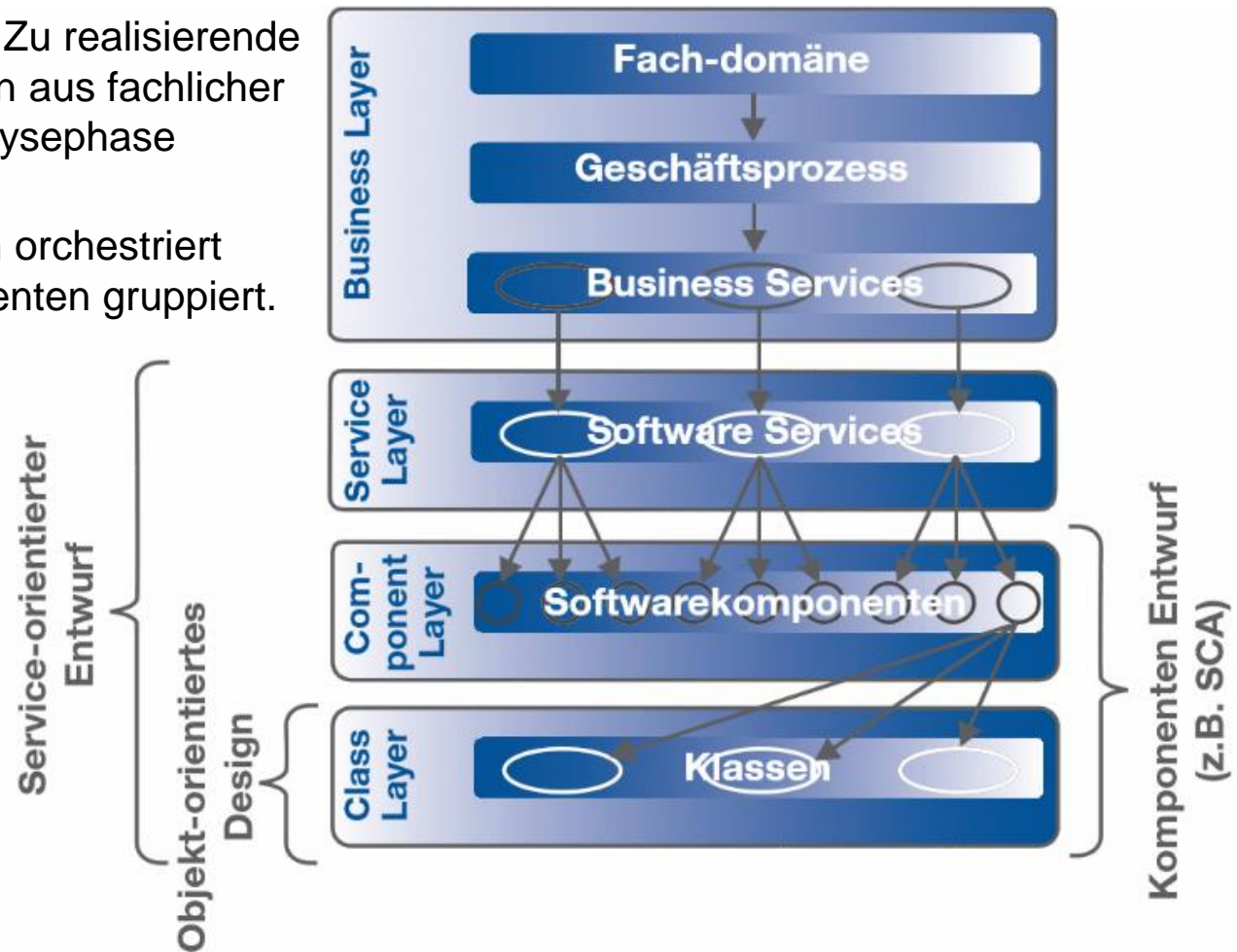


3.3 Entwurfsparadigmen: Serviceorientiertes Design – Zusammenspiel



□ Zusammenspiel des Service orientierten Entwurfs

- Business Layer: Zu realisierende Prozesse werden aus fachlicher Sicht in der Analysephase modelliert.
- Services werden orchestriert und zu Komponenten gruppiert.



3.3 Entwurfsparadigmen: Serviceorientiertes Design – Komponenten



- ❑ **Komponenten, welche zusätzlich zu den eigentlichen Services benötigt werden:**
 - **Enterprise Service Bus (ESB):** Typbezeichnung für eine bestimmte Art von Software zur Integration von Services, bietet vor allem Funktionen für die Transformation, Routing und Orchestrierung von Serviceaufrufen (Kommunikation).
 - **Process Engine (Service Orchestration):** Führt Prozessmodelle aus.
 - ◆ Jeder Prozess beinhaltet eine Abfolge von genau definierten Aktivitäten, jede dieser Aktivitäten kann dabei durch einen individuellen Service (bzw. Serviceaufruf) übernommen/realisiert werden.
 - ◆ Eine Engine koordiniert nun anhand des Prozessmodells wie und in welcher Reihenfolge diese Serviceaufrufe durchgeführt werden.
 - **Service Repository:** Enthält Informationen über verfügbare Services.
 - **Service Monitoring:** Überwacht den Status der einzelnen Services und die Kommunikation zwischen den Services.
 - **Rules Engine:** Verwaltung von Geschäftsregeln, z.B. dass gewisse Kombinationen von Services verboten sind.



3.1 Motivation

3.2 Grundlagen

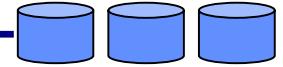
3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

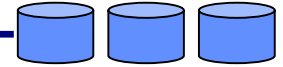
3.5 Serviceorientierte Architektur

3.6 Zusammenfassung

3.4 Muster: Motivation

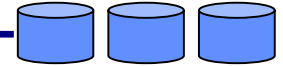


- ❑ **Muster (Patterns):** Stellen eine gute Lösungsstrategie für wiederkehrende Probleme dar.
- ❑ **Was sind die Vorteile von Mustern? Warum werden diese eingesetzt bzw. hier vermittelt?**
 - Vermeiden, dass in jedem Softwareprojekt „das Rad neu erfunden werden muss“.
 - Erleichtern und beschleunigen den Entwurf einer Software.
 - Ermöglichen es Ihnen die Prinzipien der objektorientierten Entwicklung (z.B. Kopplung und Kohäsion) praxisnah kennenzulernen.
 - Erleichtern die Kommunikation z.B. Sourcecode-Kommentare verweisen teilweise direkt darauf, dass ein bestimmtes Muster an dieser Stelle umgesetzt wurde.
 - Verbesserte Wart- und Erweiterbarkeit durch erhöhte Abstraktion/Flexibilität.
- ❑ **Es ist jedoch Vorsicht geboten, daher:**
 - Nicht jedes Problem zwanghaft mit Mustern „erschlagen“, sonst leidet die Übersichtlichkeit und Vorteile, wie beispielsweise die erleichterte Wartbarkeit, gehen wieder verloren.
 - Die zusätzliche Abstraktion die Muster zumeist absichtlich mit sich bringen bzw. ermöglichen kosten teilweise empfindlich Performance.



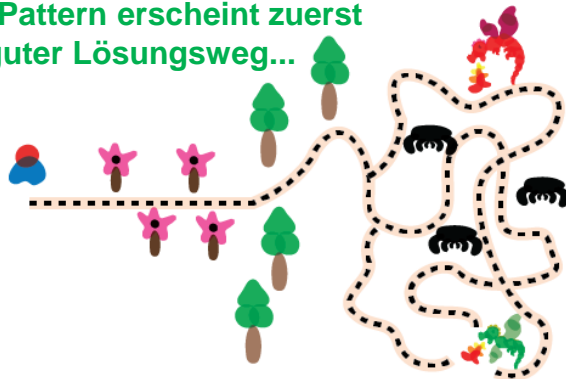
- ❑ **Beschreibung von Mustern:** Musterbeschreibungen umfassen zumeist folgende drei Bestandteile:
 - Kontext: Beschreibt die Problemdomäne.
 - Problem: Beschreibt die Aufgabenstellung.
 - Lösung: Beschreibt die Idee hinter dem Pattern sowie das Pattern an sich.
 - Erweiterung möglich durch:
 - ◆ Recurrence: Wiederauftreten des Musters in einer neuen Situation außerhalb der zuvor genannten Problemdomäne.
 - ◆ Teaching: Idee hinter dem Pattern und Vorschläge zur Umsetzung.
 - ◆ Name: Möglichst eindeutige Bezeichnung des Patterns.
- ❑ **Je nach Abstraktionsebene (Mikro vs. Makro) werden verschiedene Muster eingesetzt.**
 - Architekturmuster (Architektur-Pattern): Architektonische Ebene (Makroarchitektur)
 - Entwurfsmuster (Design-Pattern): Klassenebene (Mikroarchitektur)

3.4 Muster: Grundlagen



- ❑ **Pattern vs. Anti-Pattern:** Ähnlich wie Pattern, jedoch beschreiben diese eine Lösung welche nur scheinbar vorteilhaft ist:
 - Einige Beispiele für Anti-Pattern sind:
 - ◆ Angst vor Klassen/Methoden: Die Hoffnung Übersichtlichkeit zu erlangen indem nur einige wenige Klassen/Methoden erstellt werden. Diese werden im Endeffekt aber sehr groß und deshalb erst recht komplex sowie schwer zu verstehen, warten und testen.
 - ◆ Vorrauseilende Abstraktion: Klassen welche Abstraktionsebenen zwischen Klassen einziehen um für zukünftige Erweiterungen gerüstet zu sein, zum jetzigen Zeitpunkt jedoch keine Funktionalität hinzufügen. Statt die Verständlichkeit des Codes zu verbessern gibt es jetzt nur "mehr" zu lesen. Beispiel hierfür:

Ein Anti-Pattern erscheint zuerst wie ein guter Lösungsweg...



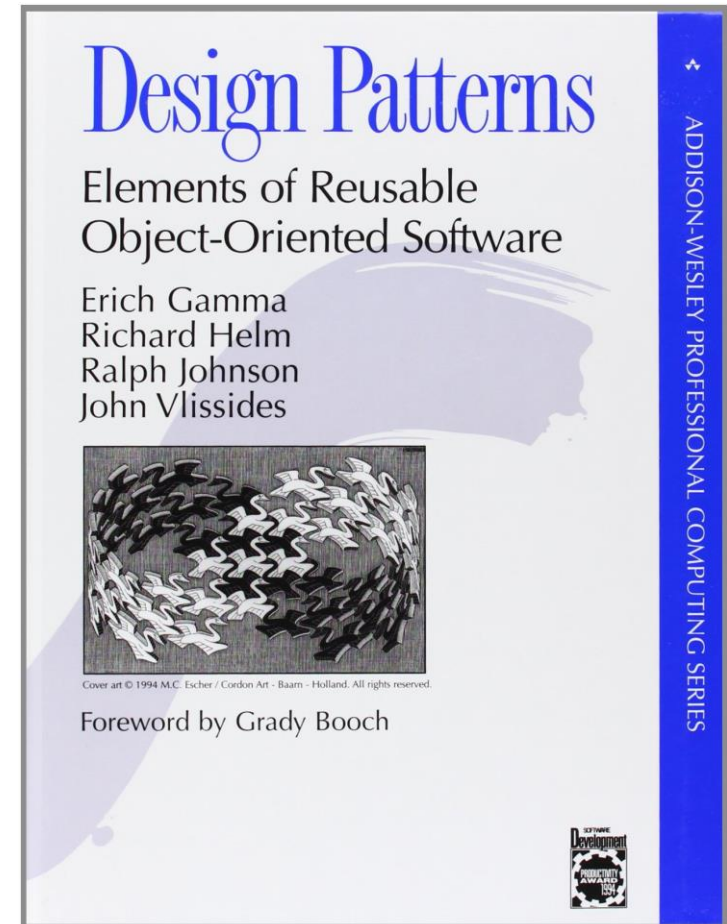
...führt jedoch in ein Labyrinth gefüllt mit Fallen/Monstern und langfristig zu vielen Problemen.

```
public class TetraStack<T> {  
    private LinkedList<T> list;  
    public TetraStack() {  
        list = new LinkedList<T>();  
    }  
    public boolean empty() {  
        return list.isEmpty();  
    }  
    public T peek() throws  
        EmptyStackException {  
        if (list.isEmpty()) {  
            throw new EmptyStackException();  
        }  
        return list.peek();  
    }  
}
```

3.4 Muster: Literatur für Entwurfsmuster



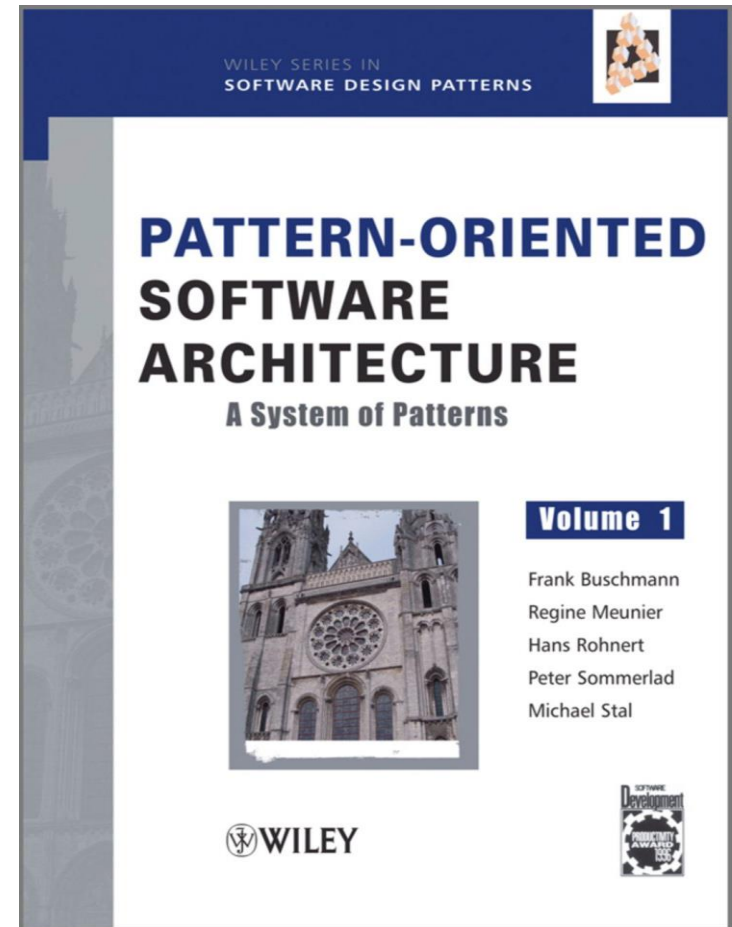
- ❑ **Literaturempfehlung zu Entwurfsmustern:**
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
Design Patterns. Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995
- ❑ Standardwerk, zeigt den Einsatz und die Vorteile vieler Pattern anhand eines zusammenhängen Beispielprojektes.
- ❑ Empfohlen, für alle die mehr über Pattern erfahren möchten.

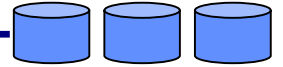


3.4 Muster: Literatur für Architekturmuster



- ❑ **Literaturempfehlung zu Architekturmustern:**
Frank Buschmann, Regine Meunier, Hans Rohnert, et al.:
Pattern-Oriented Software Architecture
Wiley, 1996
- ❑ Buchreihe (POSA 1 – POSA 5), jeweils mit unterschiedlichen Fokussen. Von klassischen Architekturpattern bis hin zu Beschreibungssprachen für Pattern.
- ❑ Empfehlung POSA 1 (gut als Einstieg), POSA 2 und POSA 4.

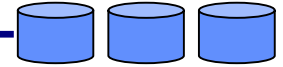




Muster

ARCHITEKTURMUSTER

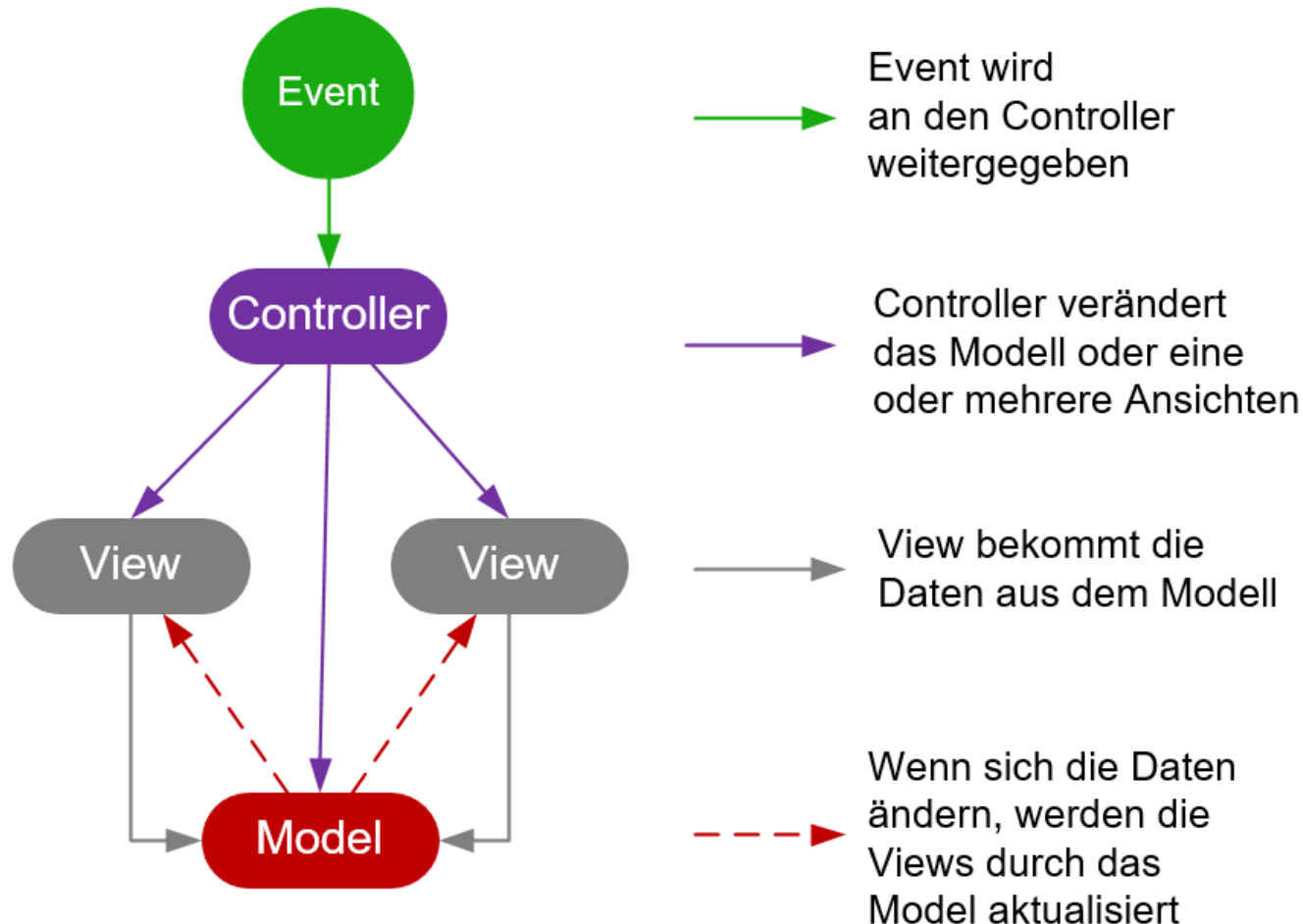
3.4 Architekturmuster: Definition



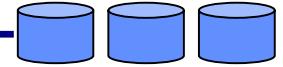
- ❑ **Nach Buschmann werden Architekturmuster bzw. Architektur-Pattern wie folgt definiert:**
 - *“An Architectural pattern expresses a fundamental structural organization schema for software systems.”*
 - *“It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationship between them.”*
- ❑ **Alternative Bezeichnung für Architekturmuster:**
 - Styles innerhalb der „Component and Connector View-Type“.
- ❑ **Die wichtigsten Architekturmuster sind:**
 - Model-View-Controller
 - Schichtenarchitektur (Layered Architecture)



□ Model-View-Controller: Grafische Darstellung/Überblick



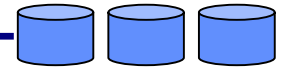
3.4 Architekturmuster: Model-View-Controller – Beschreibung



□ Model-View-Controller: Beschreibung

- **Kontext:** Interaktive Anwendung mit einer (oder mehreren) möglichst flexiblen Schnittstelle(n) zwischen Mensch und Computer.
- **Problem:** User Interfaces werden oft verändert oder erweitert.
- **Lösung:** Aufteilung der Anwendung in drei Gebiete (Verarbeitung, Output, Input).
- **Model:** Kapselt Daten und Funktionalität – unabhängig von der Darstellung des Outputs und des Input-Verhaltens.
 - ◆ Hält Daten für die View vor, erhält aber auch Daten von der View (über den Controller).
- **View:** Darstellung der entsprechenden Informationen für den User.
 - ◆ Wichtig: Es sind mehrere Views für ein einzelnes Model möglich.
- **Controller:** Empfängt Inputs in Form von Events und gibt diese in Form von Service Requests an das Model oder die View weiter.
 - ◆ Jede View hat einen Controller.
 - ◆ Interaktion mit dem User kann nur über den Controller stattfinden.

3.4 Architekturmuster: Model-View-Controller – Beispiel

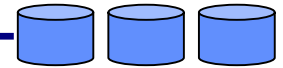


□ Model-View-Controller Beispiel

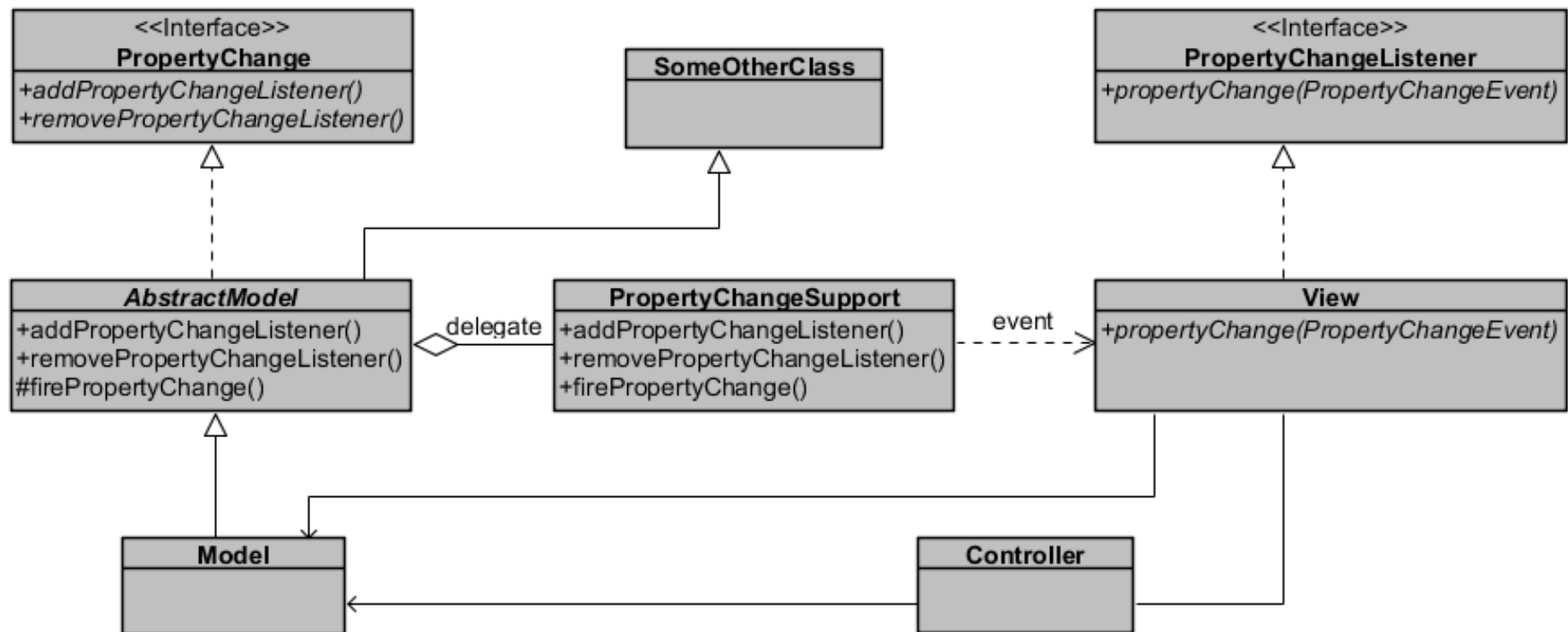
- Smartphone – mehrere Views für Anrufe:
 - ◆ Bildschirm
 - ◆ Lautsprecher
 - ◆ Benachrichtigungsleuchte
 - ◆ Vibrationsmotor



3.4 Architekturmuster: Model-View-Controller – Struktur 1



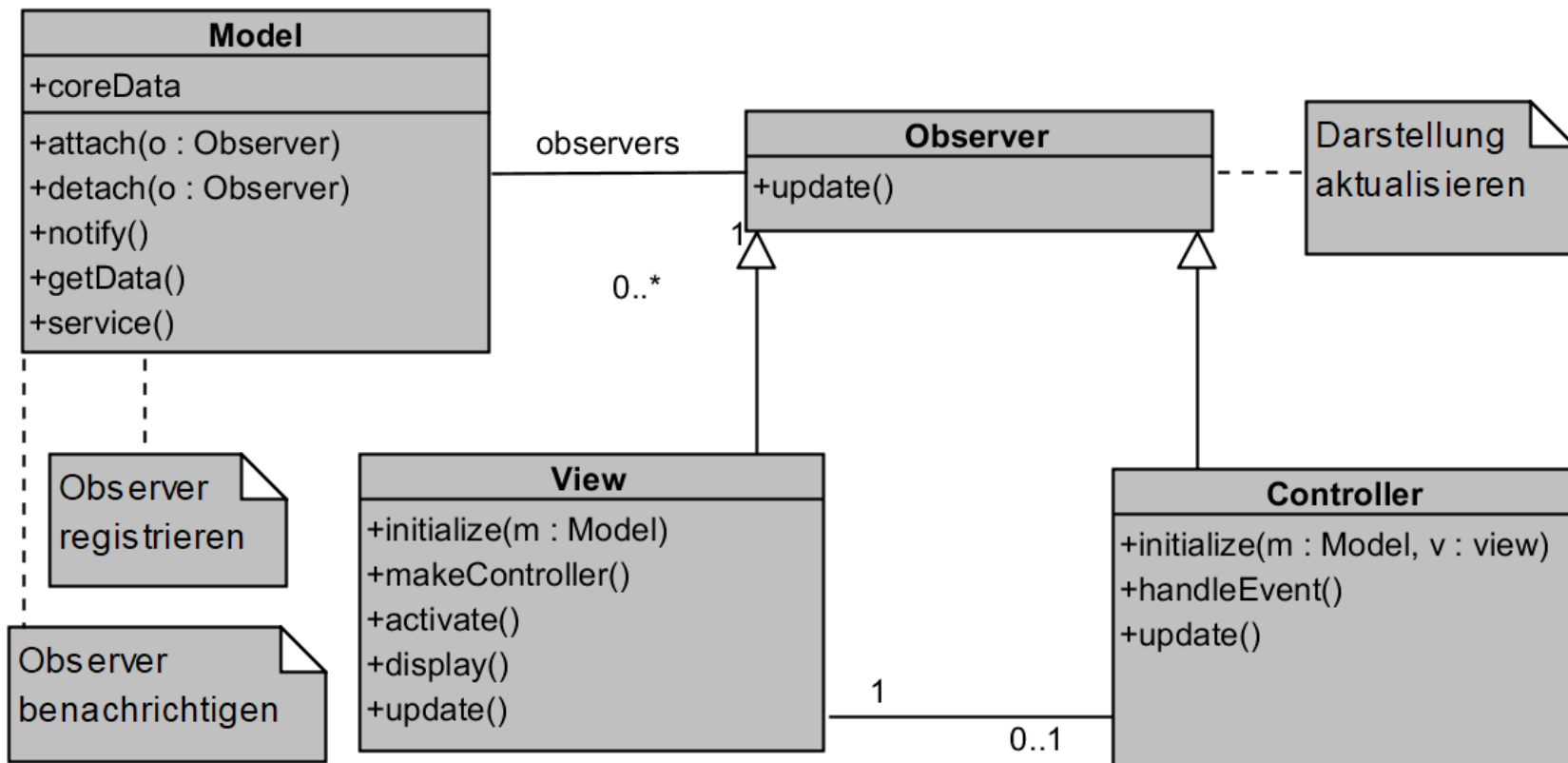
□ Model-View-Controller: Klassenmodell in Java mit PropertyChangedListener



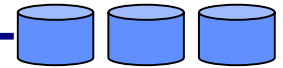
3.4 Architekturmuster: Model-View-Controller – Struktur 2



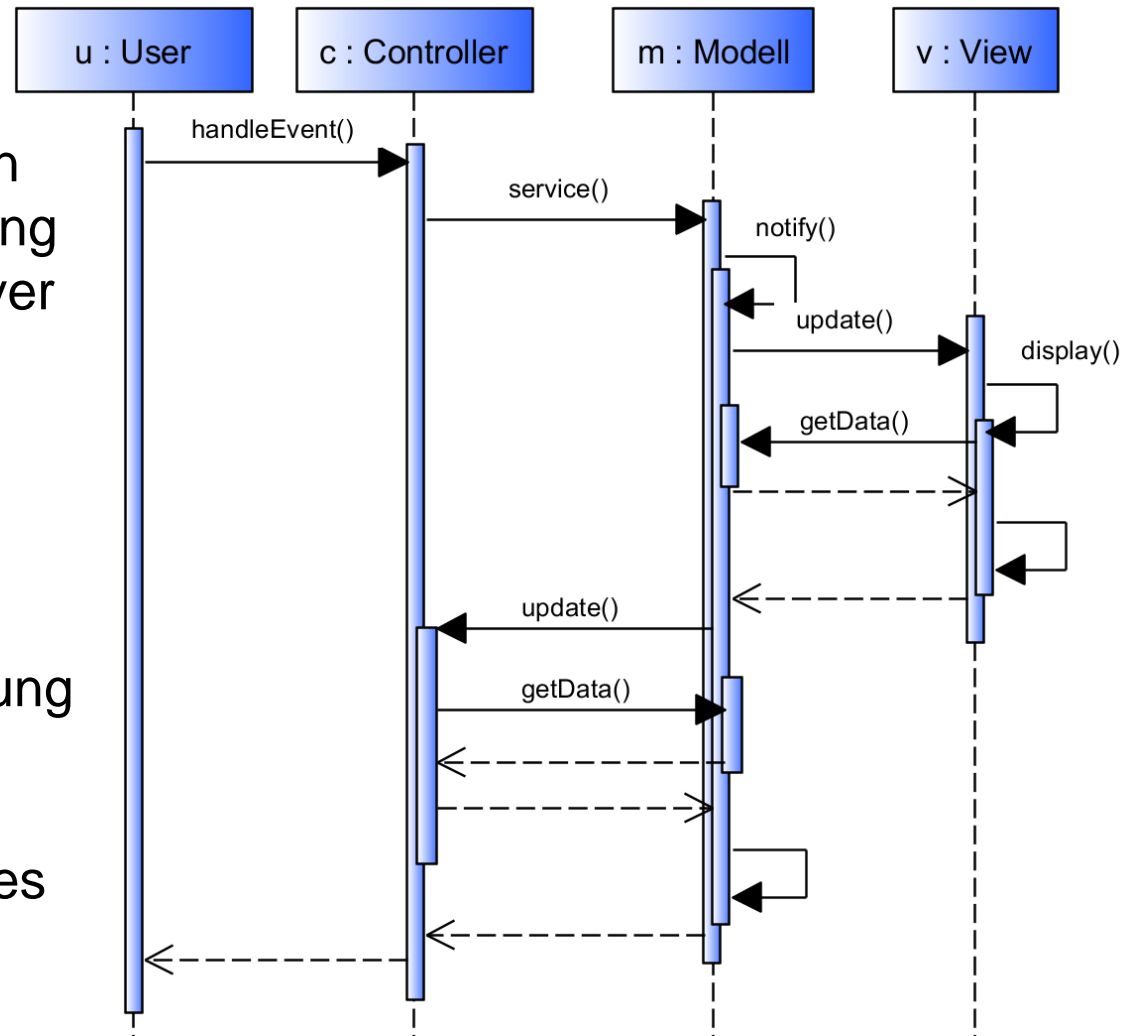
❑ Model-View-Controller: Klassenmodell in Java mit Observer Pattern



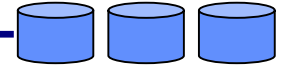
3.4 Architekturmuster: Model-View-Controller – Dynamisches Verhalten



- ❑ **Model-View-Controller:** Dynamisches Verhalten bei einer neuen Useraktion (z.B. Betätigung eines Knopfes) mit Observer Pattern.
- ❑ Observer/Prop. Changed sind aus Sicht des MVC Patterns gleichwertig. Es zählt bei Pattern weniger eine spezifische Umsetzung als, dass die Ziele (z.B., das flexible dynamische hinzufügen von Views) des Patterns erreicht werden.



3.4 Architekturmuster: Model-View-Controller – Vorteile und Nachteile



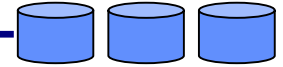
□ Vorteile des Model-View-Controller Pattern

- Mehrere Views, basierend auf dem selben Modell.
- Synchrone Views, daher verschiedene Anzeigen sind zueinander konsistent.
- „Ansteckbare“ Views und Controller (einfach zu erweitern).
- Wird während der Entwicklung der meisten grafischen Oberflächen und den zugehörigen Frameworks in unterschiedlichen Ausprägungen eingesetzt.

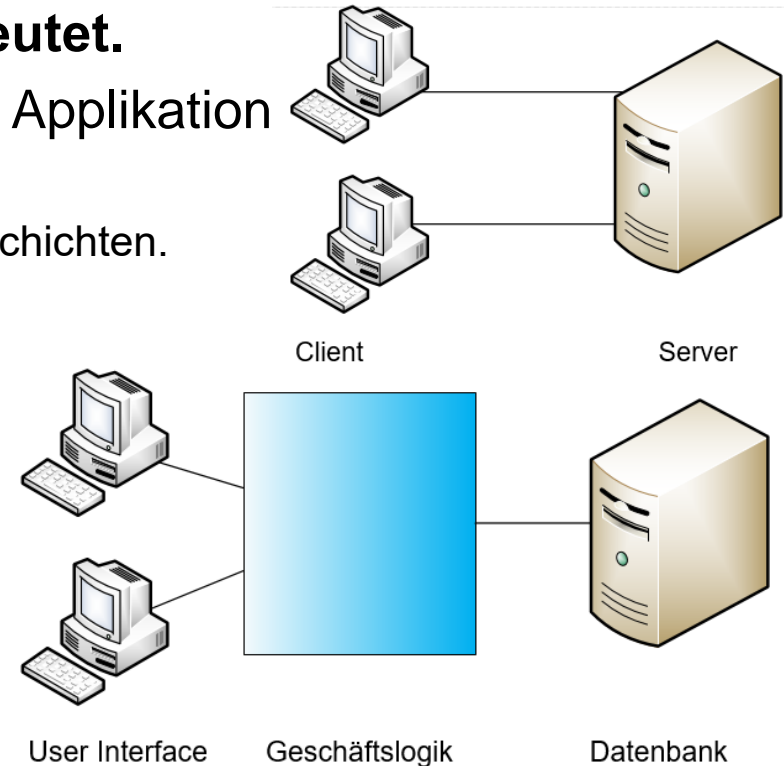
□ Nachteile des Model-View-Controller Pattern

- Erhöhte Komplexität
- Kopplung zwischen Modell und View ⇒ Zusätzlicher Aufwand notwendig um:
 - ◆ Modeländerungen gebündelt und asynchron zu verarbeiten um die Darstellungsgeschwindigkeit zu erhöhen (sonst wird bei jeder einzelnen Datenänderung das UI neu gezeichnet).
 - ◆ Daten und deren Darstellung zu entkoppeln bzw. zu automatisieren.
- Starke Kopplung zwischen Modell und Controller
 - ◆ Lösung dieser Einschränkung mittels Command-Pattern möglich.
 - ◆ Dieses Pattern ermöglicht es komplexes Verhalten im Controller weiter zu unterteilen und Funktionalität leichter hinzuzufügen (erschwert aber das Verständnis).

3.4 Architekturmuster: Unterschied N-Tier- und Schichtenarchitektur



- ❑ **Grundkonzept des Softwareentwurfs sind N-Tier-Architekturen, wobei „Tier“ übersetzt Schicht bedeutet.**
- ❑ **N-Tier-Architekturen:** Unterteilen der Applikation in funktional abgegrenzte Schichten.
 - „N“ steht für die Anzahl der eingesetzten Schichten.
 - Im praktischen Einsatz finden sich:
 - ◆ 2-Tier-Architektur bzw. Client/Server Architektur
 - ◆ 3-Tier-Architektur
- ❑ **Unterschied N-Tier Architektur und Layered Architecture.**
 - N-Tier-Architekturen werden zumeist auf verschiedenen Computersystemen ausgeführt (Verteilte Systeme).
 - Schichtenarchitektur unterteilt ein Programm in verschiedene Layer, das heißt ein Teil einer N-Tier-Architektur kann wiederum aus mehreren Layer aufgebaut sein.

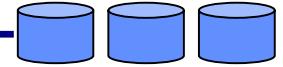


3.4 Architekturmuster: Schichtenarchitektur 1



- ❑ **Schichtenarchitektur (Layered Architecture):** Jede Schicht (Layer) übernimmt eine klar definierte Rolle und bietet darüber liegenden Schichten eine Menge an Diensten an.
 - Einer der beliebtesten Architektur-Stile im Rahmen der Softwareentwicklung.
- ❑ **Welche Eigenschaften sollte eine Schichtenarchitektur aufweisen**
 - Jede Schicht verbirgt sowohl die darunterliegende Schicht als auch die interne Komplexität.
 - ◆ Untere Schichten konzentrieren sich auf die Technik.
 - ◆ Obere Schichten konzentrieren sich auf die Benutzerschnittstelle.
 - Ähnlicher Abstraktionsgrad von Komponenten innerhalb einer Schicht.
 - Die Kommunikation zwischen den Schichten erfolgt durch klar definierte Schnittstellen (Interfaces) und Protokolle.
 - Eine Schicht kommuniziert nur mit der Schicht direkt darunter.
 - Eine Schicht kommuniziert nicht mit den darüber liegenden Schichten.
 - Eine Schicht hat keine Abhängigkeiten zu einer der darüber liegenden Ebenen.
 - Auftretende Exceptions müssen in der werfenden Schicht verarbeitet werden.

3.4 Architekturmuster: Schichtenarchitektur 2



❑ Grund für den Einsatz der Schichtenarchitektur

- Jede Schicht kann als eigene Programmkomponente gesehen werden, dadurch kann jede Schicht unabhängig von den anderen entwickelt und getestet werden.

❑ Kommunikationsarten

○ Top-down-Kommunikation

- ◆ Benutzereingabe wird von der obersten Schicht entgegengenommen
- ◆ Anschließend werden diese bis in die untersten Schichten weitergeleitet.
- ◆ Ergebnisse der unteren Schichten werden gegebenenfalls wieder nach oben weitergeleitet, bis das Ergebnis an den Benutzer zurückgegeben wird.

○ Bottom-Up-Kommunikation

- ◆ Unterste Schicht empfängt ein Signal.
- ◆ Signal wird die Schichten nach oben weitergeleitet bis der Benutzer benachrichtigt wird.

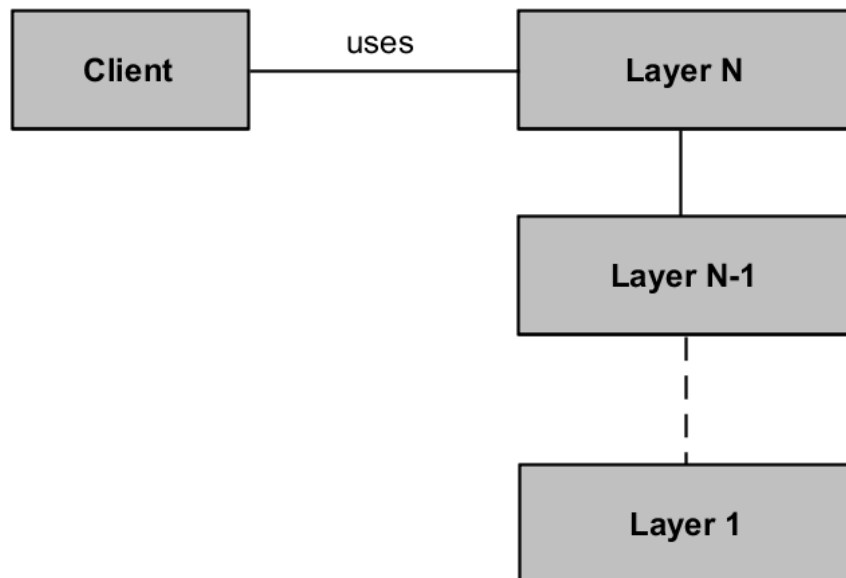
❑ Art der Schichtenbildung

- **Horizontale Schichtenbildung:** Schichten werden horizontal gestapelt und jede horizontale Schicht hat eine Aufgabe.
- **Vertikale Schichtenbildung:** Unterteilung einer Schicht in mehrere Partitionen. Jede Partition hat vollen Zugriff auf alle angrenzenden horizontalen Schichten.

3.4 Architekturmuster: Schichtenarchitektur 3



□ Grafische Darstellung der Schichtenarchitektur



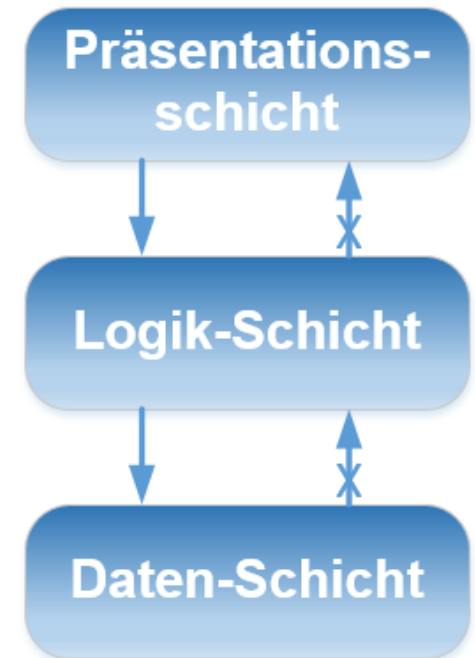
□ Mögliche Ausprägungen der Schichtenarchitektur

- 2-Schichtenarchitektur (in verteilten Systemen auch Client/Server Architektur)
 - ◆ Client: Beinhaltet GUI und die gesamte Applikationslogik.
 - ◆ Server: Besteht meist aus einer relationalen Datenbank.
- 3- und 5-Schichtenarchitektur

3.4 Architekturmuster: Schichtenarchitektur – 3-Schichtenarchitektur



- ❑ **3-Schichtenarchitektur:** Es wird eine zusätzliche Logik- oder Business-Schicht eingeführt um Code der Geschäftslogik vom Client zu trennen.
 - **Präsentationsschicht:** Über diese Schicht erfolgt jegliche Interaktion mit dem Benutzer.
 - ◆ Enthält Logik für die Verarbeitung von Ereignissen, diese Logik sollte sich allerdings nur auf GUI Komponenten (Anzeige) beziehen, das heißt keine Implementierung der Geschäftslogik.
 - **Logik-Schicht:** Beinhaltet die Kernfunktionalität des Systems (Geschäftslogik).
 - ◆ Aufbauend auf die Daten-Schicht.
 - ◆ Typische Funktionen: Verarbeitung und Aufbereitung von Daten für die Präsentationsschicht.
 - **Daten-Schicht:** Stellt eine Menge an Datenquellen zur Verfügung.
 - ◆ Stellt die unterste Ebene dar.
 - ◆ Typische Objekte: Relationale oder objektorientierte Datenbanken

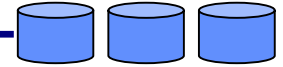


3.4 Architekturmuster: Schichtenarchitektur – 5-Schichtenarchitektur



- ❑ **4-Schichtenarchitektur:** Weitere Unterteilung der Logik-Schicht in:
 - **Datenzugriffs-Schicht (Data Access Layer)**
 - ◆ Abstrahieren den Zugriff auf die darunterliegende Schicht.
 - ◆ Enthält eine Menge von Datenzugriffsobjekten (Data Access Objects, DAOs).
 - **Service-Schicht**
 - ◆ Servicekomponenten laden und speichern Daten unter Verwendung der Data Access Objects.
- ❑ **5-Schichtenarchitektur:** Zusätzliche Prozess-Schicht wird eingeführt um die Geschäftslogik so atomar wie möglich zu implementieren.
 - Prozess-Schicht liegt zwischen der Logik-Schicht und der Präsentationsschicht.
 - Kümmt sich um die Abläufe (Prozesse) höherer Granularität und wird häufig unter Verwendung einer Prozess-Engine (z.B. jBPM oder Node-Red) umgesetzt.
 - Atomare Services aus der Logik-Schicht werden zu einem Prozess kombiniert der in einer eigenen Prozess-Sprache (Prozessmodell) vorliegt.
 - Ablauflogik befindet sich nicht mehr im Code sondern in Prozessmodellen, dadurch können Abläufe leichter verändert und an neue Bedürfnisse angepasst werden. Da dies oft mit „einfachen“ grafischen Beschreibungssprachen möglich ist erlangt der Kunde bzw. das Management die Möglichkeit selbst Verhalten zu definieren.

3.4 Architekturmuster: Schichtenarchitektur – Vorteile und Nachteile



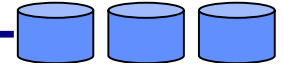
□ Vorteile der Schichtenarchitektur

- Einfaches, verständliches Architekturmuster.
- Saubere Trennung der einzelnen Schichten – ermöglicht verteiltes Arbeiten
- Austauschbare Schichten, Betrieb kann auf verschiedenen Infrastrukturen erfolgen.
- Eine minimale Abhängigkeit zwischen den Schichten wird durch die klare Definition der Kommunikationswege zwischen Schichten durch Schnittstellen erreicht.
- Klare Trennung der Schichten: Vereinfachung der Wartbarkeit und Erweiterbarkeit.

□ Nachteile der Schichtenarchitektur

- Strikte Einhaltung der Top-down-Kommunikation kann dazu führen, dass eine triviale Operation in einer Schicht durch alle Schichten „durchgereicht“ werden muss und dadurch kann eine einfache Operation zu einer komplexen Funktion werden.
- Mehr Schichten: Mehr Komplexität und Implementierungsaufwand.
- Weniger Schichten: Stärkere Kopplung und weniger Flexibilität.
- Änderung an den Schnittstellen der unteren Schichten können zu Änderungen in den oberen Schichten führen und erhöht dadurch den Anpassungsaufwand.

3.4 Architekturmuster: Schichtenarchitektur – Beispiel



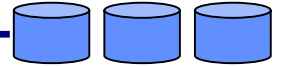
- ❑ **ISO/OSI Referenzmodell:**
Teilt Netzwerk-Protokolle in 7 Schichten, jede Schicht ist für eine bestimmte „Aufgabe“ zuständig

Anwendungsschichten
(Application Layer)

Transportschichten
(Transport Layer)

- ❑ **Weiteres Beispiel:**
JDBC Schnittstelle

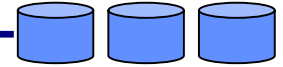




Muster

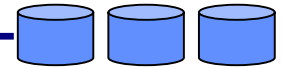
ENTWURFSMUSTER

3.4 Entwurfsmuster: Definition



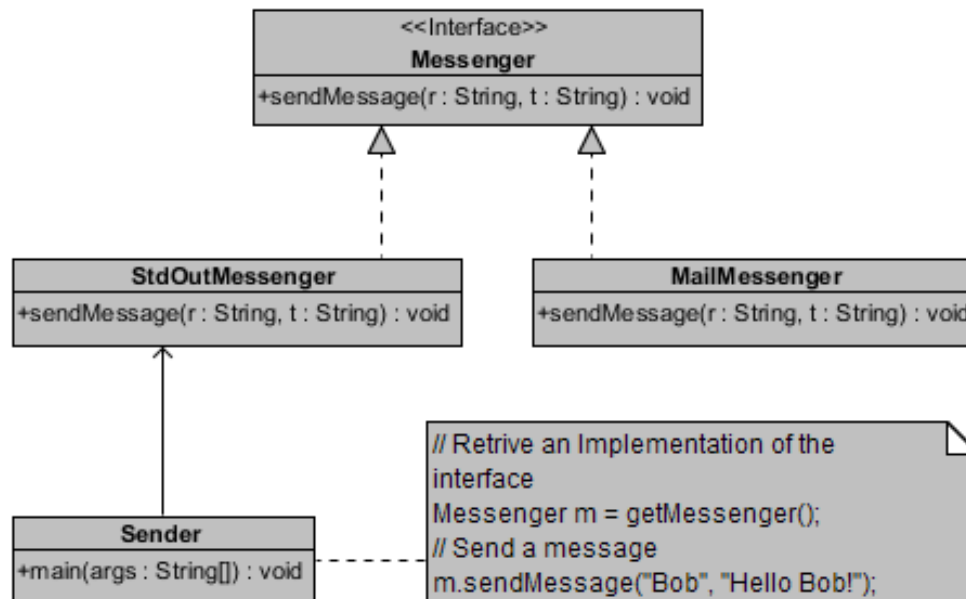
- ❑ **Definition Entwurfsmuster bzw. Design-Pattern (Gamma et al.):**
 - „A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationship between them.“
 - „It describes a commonly-recurring structure of communication components that solves a general design problem within a particular context.“
- ❑ **Entwurfsmuster befinden sich nach dem Abstraktionsgrad unterhalb der Architekturmuster**
- ❑ **Einteilung und Beispiele der Entwurfsmuster**
 - **Grundlegende Muster:** *Interface, Strategy, Immutable*
 - **Erzeugungsmuster (Creational patterns):** Klassen und Objekten initialisieren und konfigurieren z.B. *Factory, Singleton*
 - **Strukturmuster (Structural patterns):** Decoupling von Interface und Implementation von Klassen und Objekten z.B. *Fassade, Iterator, Adapter, Data Access Object*
 - **Verhaltensmuster (Behavioral patterns):** Dynamische Interaktion zwischen Objekten z.B. *Observer, Decorator*

3.4 Entwurfsmuster: Grundlegende Muster 1

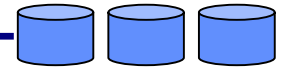


❑ Grundlegende Muster: Schnittstelle (Interface)

- Einfachstes und fundamentalstes Muster zur Trennung von Schnittstelle und Implementierung („*Develop against an Interface not a concrete Implementation*“).
- Anwendung: Grundlage für Design by Contract (Zusammenspiel einzelner Komponenten über formal definierte Schnittstellen) oder falls es für eine bestimmte Aufgabe mehrere unterschiedliche Implementationen geben wird/soll.
- Beispiel: Anwendung mit welcher Nachrichten verschickt werden können



3.4 Entwurfsmuster: Grundlegende Muster 2

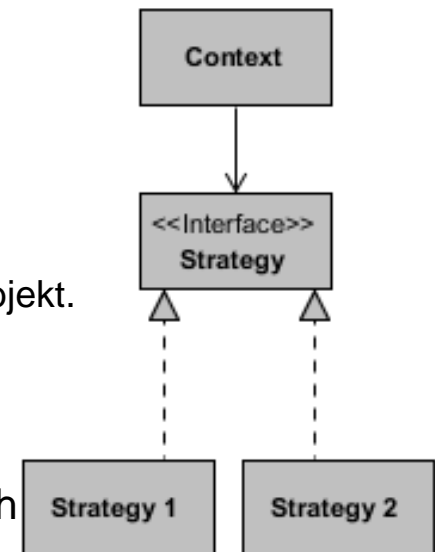


❑ Grundlegende Muster: Nicht veränderbares Objekt (Immutable)

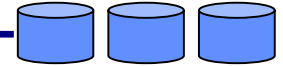
- Verhindert, dass der Zustand eines Objekts nach der Erstellung verändert wird. Einmal festgelegte Daten können nicht mehr verändert werden.
- Häufige Anwendung: Mehrere Threads arbeiten mit einem Objekt
 - ◆ Sicherstellen, dass Daten nur gelesen, aber nicht geschrieben werden können.
 - ◆ Alternativ: Objekte die Konfigurationseinstellungen speichern die nur einmal bei Programmstart eingelesen werden.
- Lösung: Werte der Klasse im Konstruktor setzen. Auslesen der Werte über Getter-Methoden aber keine Setter-Methoden anbieten.

❑ Grundlegende Muster: Strategie (Strategy)

- Entkopplung von konkreten Verhalten
- Grundstruktur:
 - ◆ Context-Objekt verwendet entweder das Strategy 1 oder 2 Objekt.
 - ◆ Zugriff erfolgt aber über das Strategy-Interface.
 - ◆ Dadurch kann das Context Objekt selbst zur Laufzeit einfach unterschiedliche Strategy Objekte einbinden.
- Anwendung: Wann immer Verhalten zu Laufzeit dynamisch ausgetauscht werden muss (z.B. Loginverfahren)



3.4 Entwurfsmuster: Grundlegende Muster – Beispiel Strategiepattern 1



□ Einsatz des Strategiepattern um zwischen zwei Loginverfahren dynamisch zu wechseln.

- Im Folgenden werden beiden Strategien definiert. Auf der nächsten Seite wird deren Verwendung gezeigt.

```
//Interface um beiden Strategien die gleiche Schnittstelle zu geben
public interface LoginStrategy {
    public String login(String user, String password) throw LoginFailed;
}
```

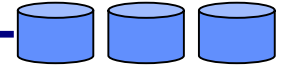
```
//Erste Login Strategie
```

```
public class OAuthLogin implements LoginStrategy {
    public String login(String user, String password) throw LoginFailed {
        // Implementierung, funktioniert nur wenn Server OAuth unterstützt
    }
}
```

```
//Zweite Login Strategie
```

```
public class DigestLogin implements LoginStrategy {
    public String login(String user, String password) throw LoginFailed {
        // Implementierung, funktioniert nur wenn Server Digest unterstützt
    }
}
```

3.4 Entwurfsmuster: Grundlegende Muster – Beispiel Strategiemuster 2



□ Einsatz des Strategiemusters um zwischen zwei Loginverfahren dynamisch zu wechseln.

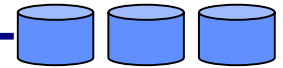
- Der folgenden Code stellt den Anwendungskontext der beiden Strategien dar.

```
public class HttpConnection {
    private URL url;
    public HttpConnection(URL url) {
        this.url = url;
    }
    public String getText(LoginStrategy authStrategy) throw LoginFailed {
        return url.downloadWithAuth(authStrategy);
    }
}
```

```
//Anwendung der Strategien, zuerst wird OAuth versucht, schlägt
//dies fehl gibt es einen Fallback auf eine andere Loginstrategie
```

```
HttpConnection connection = new HttpConnection(new URL("test"));
String text = null;
try {
    text = connection.getText(new OAuthLogin("a","b"));
}
catch(LoginFailed) {
    //fallback
    text = connection.getText(new DigestLogin("a","b"));
}
```

3.4 Entwurfsmuster: Erzeugungsmuster



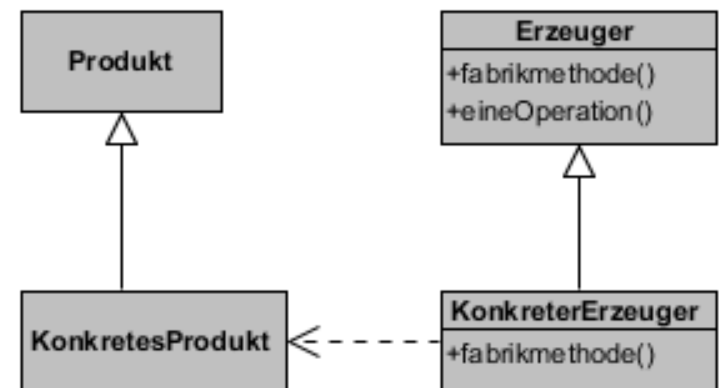
❑ Erzeugungsmuster: Singleton

- Von einer Klasse darf (soll) es nur eine einzige Instanz geben. Leicht mit `static` umzusetzen.
- Anwendung: Kommunikation mit realer Hardware, einem Logging Service, Konfigurationen einlesen, ...
- Vorteil: Kein laufendes Erzeugen und Verwerfen von Instanzen. Ermöglicht es Zugriffe und Daten durch ein gemeinsames Nadelöhr zu schleusen um z.B. gezielt Wartezeiten zwischen Zugriffen sicherzustellen.
- Best Practice: Nur verwenden wenn sich Zugriffe auf ein Singleton *nicht* gegenseitig beeinflussen können. Sonst können Singletons zu schwer debugbaren Fehlern führen, insbesondere wenn eine Implementierung mehrere Threads verwendet.

Singleton
- <u>singleton : Singleton</u>
- Singleton()
+ <u>getInstance() : Singleton</u>

❑ Erzeugungsmuster: Fabrik (Factory)

- Erzeugen von Instanzen unter komplexen Rahmenbedingungen.
- Factory kümmert sich um die Abarbeitung der korrekten Initialisierungsschritte.
- Hilft Objekte voneinander zu entkoppeln.

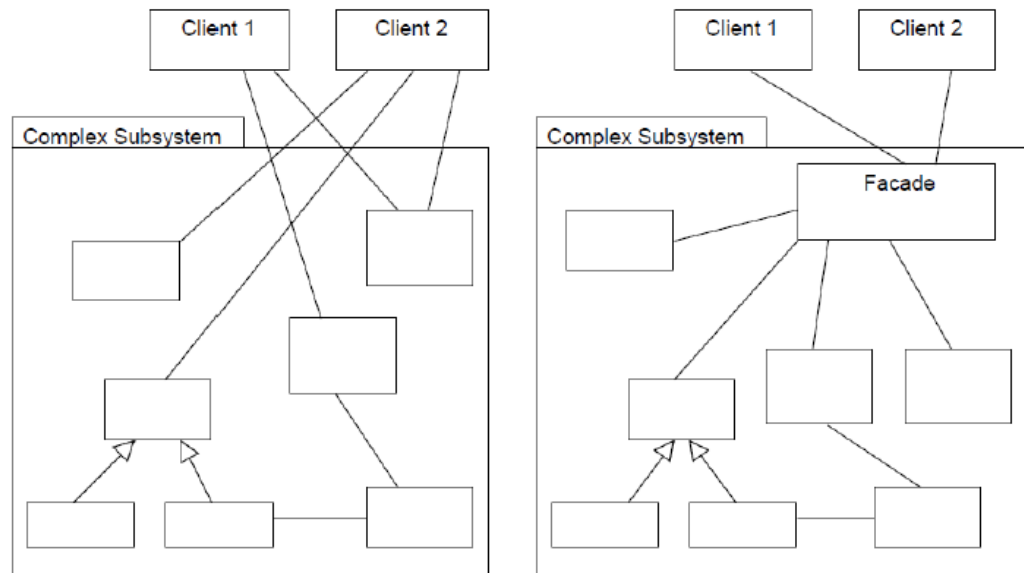


3.4 Entwurfsmuster: Strukturmuster



□ Strukturmuster: Fassade (Facade)

- Vereinfachter Zugriff auch komplexe APIs.
 - ◆ Erleichtert die Interaktion mit komplexen Bibliotheken oder Frameworks.
 - ◆ Beispiel: Benachrichtigungen per E-Mail versenden – eine Möglichkeit wäre es die Sun JavaMail Bibliothek zu verwenden, welche in Version 1.4 über mehr als 120 Klassen und Interfaces verfügt mit denen man sich auseinandersetzen müsste.
 - ◆ Lösung: Eine oder mehrere „Komfort-Methoden“ schreiben, die man sich als Zwischenschicht zwischen dem eigenen Code und der JavaMail API vorstellen kann.



3.4 Entwurfsmuster: Verhaltensmuster 1



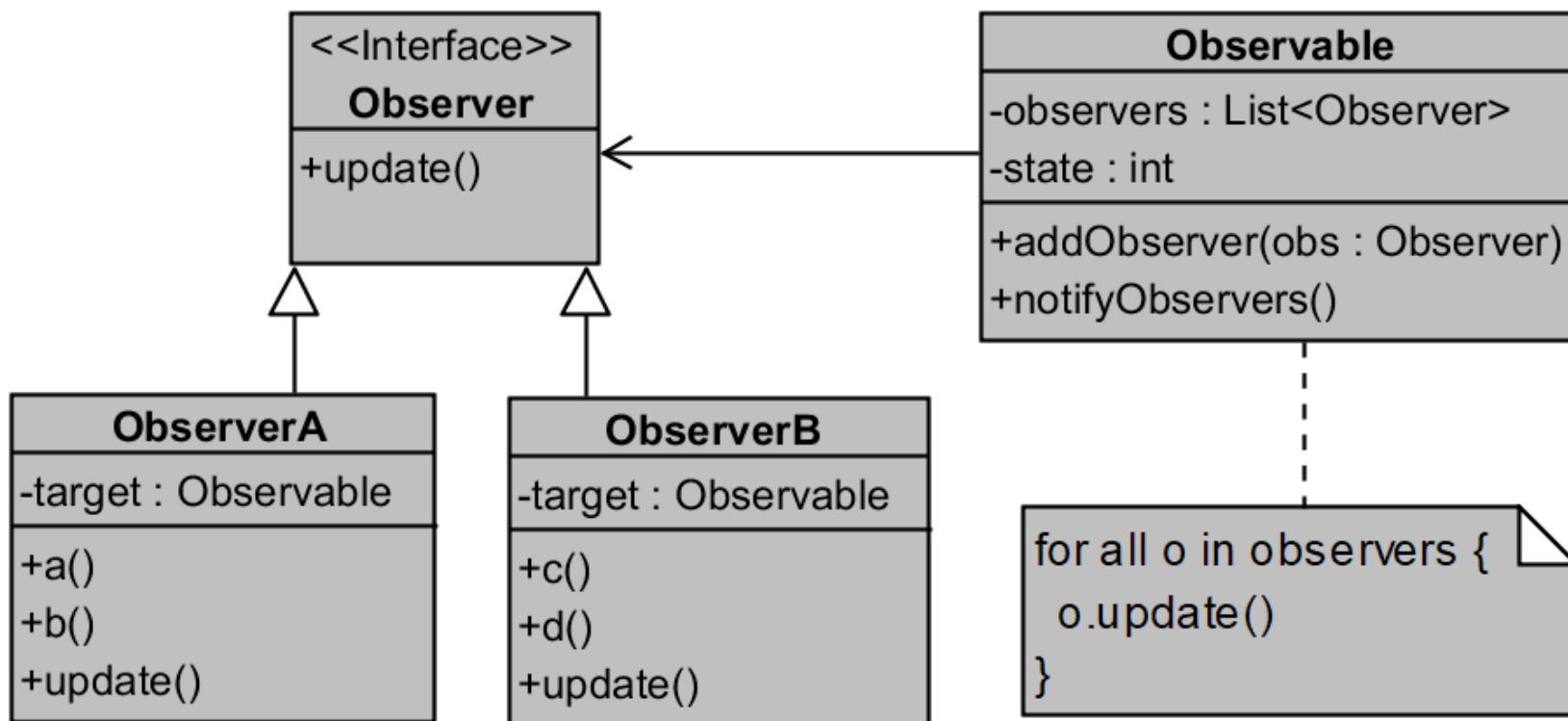
❑ Verhaltensmuster: Observer

- Im Fall von Zustandsänderungen werden interessierte Objekte benachrichtigt.
 - ◆ Oft hilfreich bei der Entwicklung von Benutzerschnittstellen, siehe MVC Pattern.
- Funktionsweise:
 - ◆ Observable: Klasse ist „unter Beobachtung“, das heißt alle Zustandsänderungen dieser Klasse werden an registrierte (interessierte) Objekte weitergeleitet.
 - ◆ Observer Interface: Definiert die Methode update().
 - ◆ Klassen, welche von Zustandsänderungen des Observable benachrichtigt werden wollen: Implementieren das Observer Interface und Registrieren sich beim Observable mithilfe der Methode addObserver().
 - ◆ Alle registrierten Objekte werden durch den Aufruf der Methode update() über Änderungen informiert.
- Vorteile
 - ◆ Registrierung bzw. die Entfernung der Registrierung einer Klasse kann zur Laufzeit erfolgen. Dies verringert die Kopplung zwischen den Klassen.
 - ◆ Ermöglicht es Zustandsänderungen auszutauschen ohne Klassen zu ändern.
- Java: Sowohl das Observer Interface als auch die Klasse Observable sind im Java Framework im Paket java.util implementiert, dadurch ist in diesem Fall das Pattern besonders leicht umzusetzen

3.4 Entwurfsmuster: Verhaltensmuster 2



❑ Verhaltensmuster: Observer – grafische Darstellung





3.1 Motivation

3.2 Grundlagen

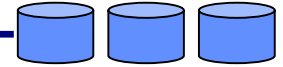
3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

3.5 Serviceorientierte Architektur

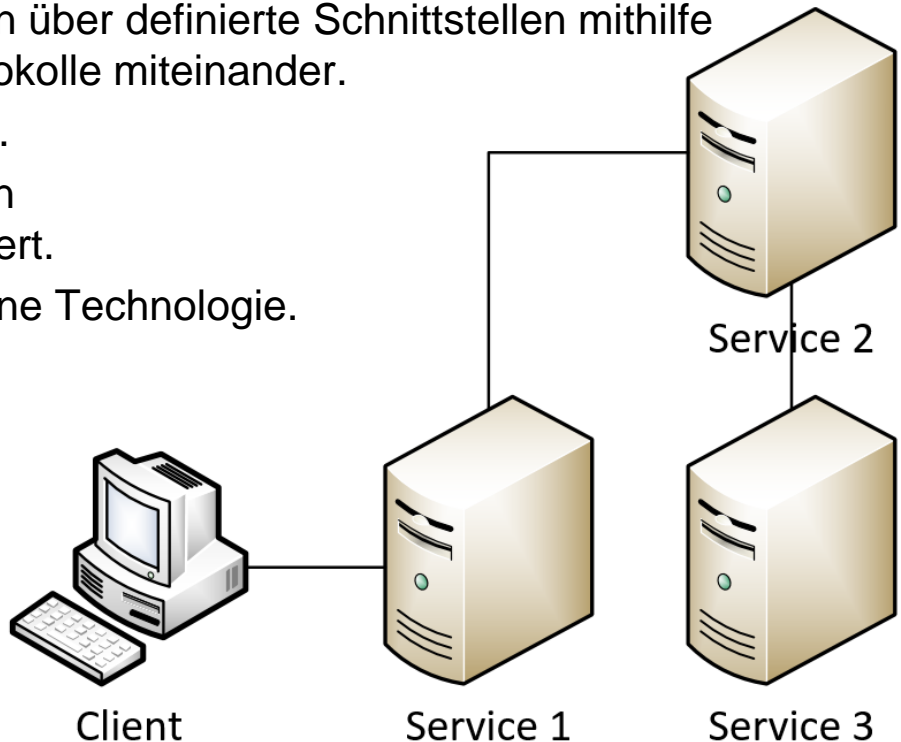
3.6 Zusammenfassung

3.5 Serviceorientierte Architektur: Grundlagen

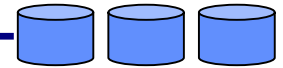


□ Serviceorientierte Architektur oder Service Oriented Architecture (SOA)

- Das Gesamtsystem wird in voneinander klar getrennte Komponenten aufgebrochen. Diese kommunizieren über definierte Schnittstellen mithilfe verschiedener Kommunikationsprotokolle miteinander.
- SOA ist Geschäftsprozess orientiert.
- Die einzelnen Anwendungen werden durch mehrere Services implementiert.
- SOA beschreibt eine Architektur keine Technologie.
- Serviceorientiertes Design ist der zugehörige Modellierungsansatz zu diesem Paradigma.

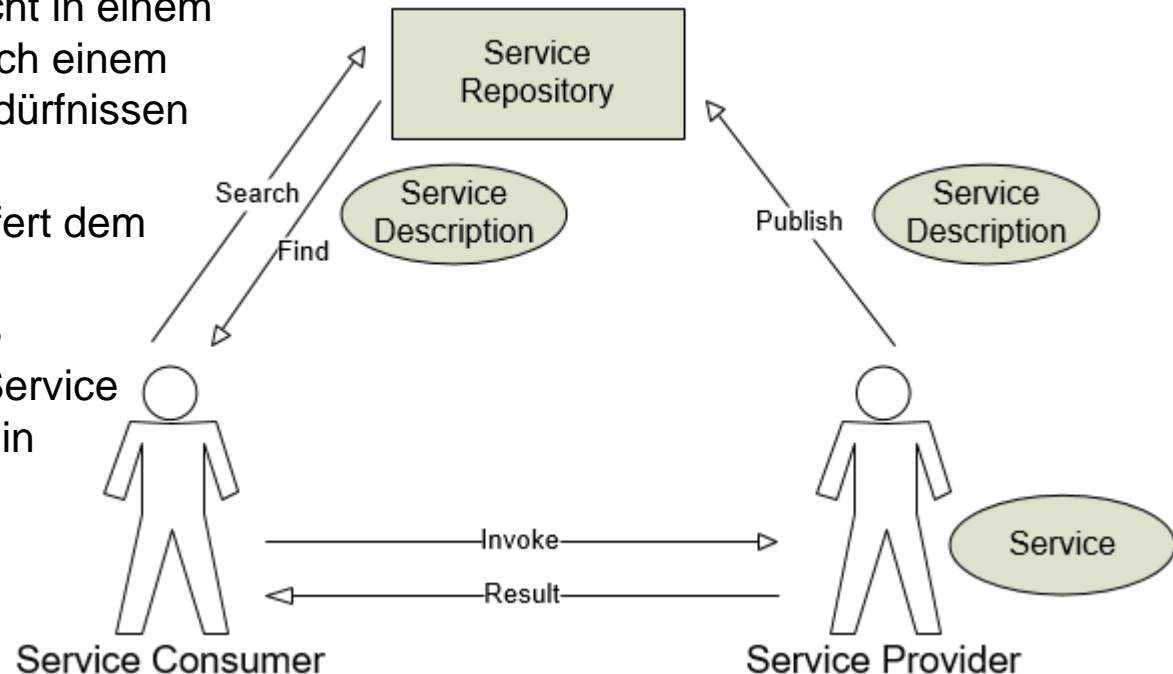


3.5 Serviceorientierte Architektur: Grundgedanke



□ Serviceorientierte Architektur Grundmodell

- Service Provider bietet einen Service an.
- Jeder Service verfügt über eine neutrale Beschreibung die in einem Service-Repository abgelegt wird.
- Service Consumer sucht in einem Service-Repository nach einem Service der seinen Bedürfnissen entspricht.
- Service-Repository liefert dem Service Consumer die Service-Beschreibung.
- Zur Laufzeit wird der Service des Service Providers in Anspruch genommen.

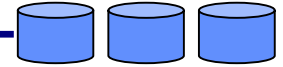


3.5 Serviceorientierte Architektur: Plattformunabhängigkeit



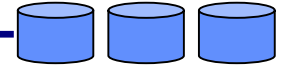
- ❑ **Bedeutung der Plattformunabhängigkeit in SOA:** Daten sowie Services über Systemgrenzen hinweg nutzen zu können.
 - ❑ **Das Service selbst sowie Service-Beschreibungen und Transportprotokolle sind in einer plattformunabhängigen Art und Weise beschrieben.**
 - Oft erfolgt die Verwendung von Standards auf XML-Basis wie
 - ◆ SOAP für Webservices
 - ◆ WSDL als Service-Beschreibung
 - ◆ http als Transportprotokoll
- ⇒ **Die Implementierung der Service Provider sowie der Service Consumer kann in einer beliebigen Technologie erfolgen.**

3.5 Serviceorientierte Architektur: Granularität 1



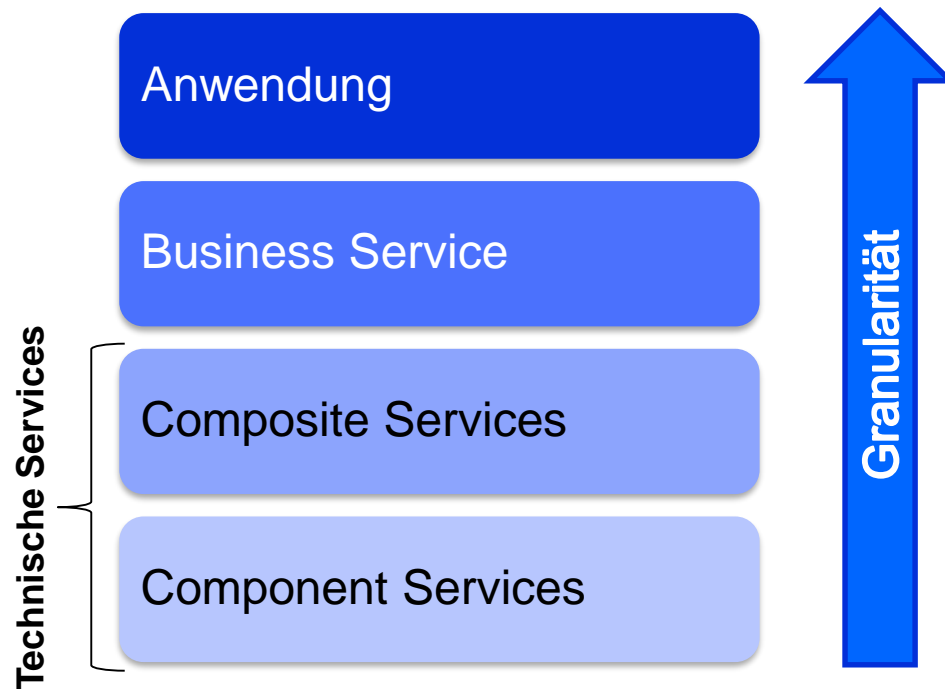
- ❑ **Granularität:** Einer der wichtigsten Faktoren im Entwurf einer SOA.
 - Maßgeblich für die Wiederverwendung (Reuse) und für die Wartung des Systems.
 - Bestimmt die Unterteilung des Systems genauso wie die Bündelung von Zugriffen auf externe oder veraltete (legacy) Anwendungen.
 - Granularität beginnt im Service orientierten Entwurf bei der Einteilung der Business Services, es stellen sich folgende Fragen:
 - ◆ Wie viele Funktionen werden in einem Business Service gekapselt?
 - ◆ Wurden zu viele Schritte im Geschäftsprozess in eine Aktivität zusammengefasst?
 - Technische Services müssen ebenfalls entsprechend (fein-)granular sein.
 - Service-Aufrufe sollten niemals über die eigene Abstraktionsebene hinausgehen – Business Services rufen immer nur Business Services auf niemals außerhalb liegende Component Services.
- ❑ **Einschub: Technische Services (Component bzw. Composite)**
 - **Component Service:** Component – Besteht aus einer konfigurierten Instanz einer Implementierung (z.B., ein einzelner grundlegender Webservice).
 - **Composite Service:** Composite – Anwendung die durch die Verknüpfung mehrerer Komponenten (Component Services) entsteht (Business Services).

3.5 Serviceorientierte Architektur: Granularität 2



□ **Granularität:** Grafische Darstellung

- Qualifizierte Service-Beschreibungen erlauben sowohl grob granulierte Services (Business Services) als auch fein granulierte Services (Component Services) zu modellieren. In beiden Fällen können ähnliche Technologien eingesetzt werden.



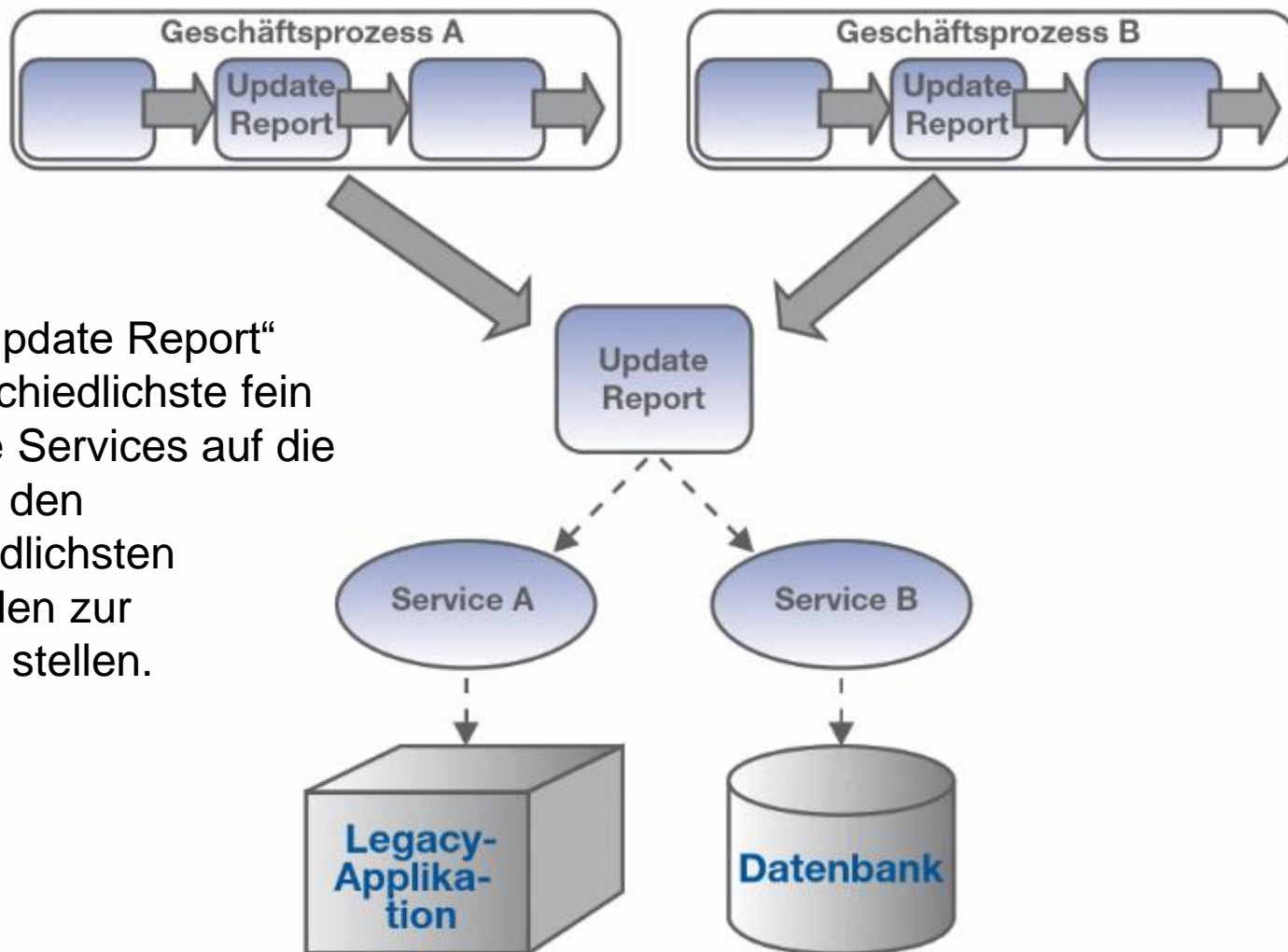
3.5 Serviceorientierte Architektur: Granularität 3



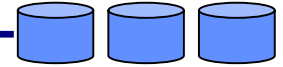
□ Granularität:

Beispiel
Business-
Service-
Granularität

- Service „Update Report“ ruft unterschiedlichste fein granulierte Services auf die Daten aus den unterschiedlichsten Datenquellen zur Verfügung stellen.



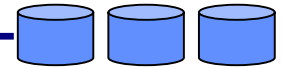
3.5 Serviceorientierte Architektur: Service-Design-Paradigmen 1



□ Paradigmen der Service-orientierten Architektur

- **Lose Kopplung (Loose Coupling):** Technik des Service-Designs, die auf größtmögliche Flexibilität ausgerichtet ist.
 - ◆ Niedrige Koppelung der Services ermöglicht eine Erhöhung der Wartbarkeit der Services, das Wissen über den Service wird reduziert und dadurch der Serviceaufruf vereinfacht.
- **Abstraktion:** Abstrakte Services können aus verschiedenen Kontexten heraus aufgerufen werden.
 - ◆ Neuentwicklungsbedarf wird reduziert und das Einsatzfeld der Services wird erhöht.
 - ◆ Spannungsfeld Abstraktion vs. technische Anforderungen: Manche Services müssen sehr konkret (spezialisiert) sein um ihren Zweck erfüllen zu können, in diesem Fall bedeutet Abstraktion das Entfernen (nahezu) jeglicher Funktionalität.
- **Standardisierung:** Zusammenarbeit und die Zusammenfügbarkeit von Services wird gewährleistet.
 - ◆ Vorteil: Erhöhte Reuseability und verbesserte Integrierbarkeit der Services.
 - ◆ Die Voraussetzung ist der Einsatz standardisierter Kommunikationsprotokolle und Datenmodelle (z.B. Ontologien können hierbei hilfreich sein).

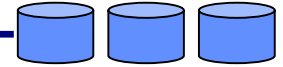
3.5 Serviceorientierte Architektur: Service-Design-Paradigmen 2



□ Paradigmen der Service-orientierten Architektur: Fortsetzung

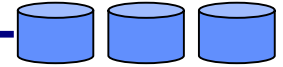
- **Composable:** Zusammenfügen von standardisierten Services im Rahmen von niedrigen Schichten sowie spezialisierten Services auf höheren Ebenen zu Hierarchien oder Komponenten. Hierdurch können z.B. Services von verschiedenen Anbietern mit einander integriert werden um Mehrwert zu erzeugen.
- **Virtualisierung (Virtualization) bzw. Virtualized:** Ermöglicht es Providern und Consumern unterschiedliche life cycles und upgrade cycles zu verfolgen.
 - ◆ Lose Kopplung, ermöglicht eine hohe Transparenz der zugrundeliegenden Ressourcen.
 - ◆ Service Provider und Service Consumer sind bezüglich der verwendeten Plattformen und verwendeten Designs unabhängig.
 - ◆ Vorteile: Dies ermöglicht es Multi Paradigma Implementierungen durchzuführen (unterschiedliche Sprachen und Frameworks werden je nach Service eingesetzt um optimale Lösungen zu erstellen).

3.5 Serviceorientierte Architektur: Service-Modellierung



- ❑ **Service-Modellierung:** In Form einer Service-Beschreibung ist ein wichtiger Faktor in der Planung einer SOA.
- ❑ **Ziel:** Möglichst viele Aspekte eines Services abzubilden.
- ❑ **Möglichen Implementierungs-Problemen** kann durch eine ausführliche Service-Beschreibung vorgegriffen werden.
- ❑ **Identifikation und Festlegung von Schnittstellen** vor der Implementierung vereinfacht eine spätere Systemintegration.
- ❑ **Wichtiges Mittel** um die Wiederverwendbarkeit eines Services zu erhöhen.
- ❑ **Liefert genaue Informationen** darüber was ein Service liefern soll und welche Abhängigkeiten zu anderen Services bestehen.

3.5 Serviceorientierte Architektur: Schichten

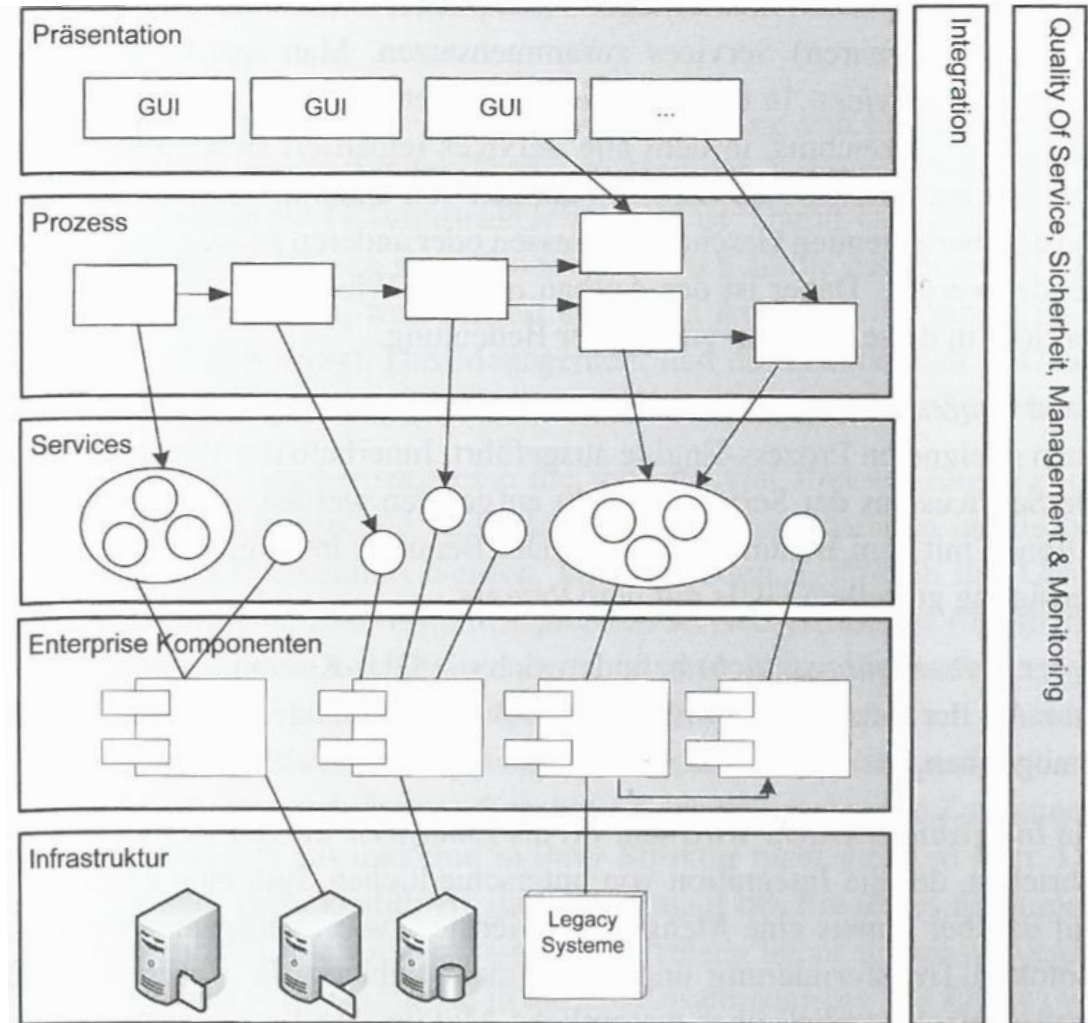


- ❑ **Schichten einer Serviceorientierten Architektur:** Schichten sind anders verteilt als in der herkömmlichen Schichtenarchitektur.
 - Schichten sind meistens auf viele unterschiedliche Rechner und sogar unterschiedliche Plattformen verteilt.
- ❑ **Verwendung von sowohl vertikalen als auch horizontalen Schichten.**
 - Horizontale Schichten können Dienste der vertikalen Schichten in Anspruch nehmen und umgekehrt.
- ❑ **Kommunikationsrichtung muss für jeden Anwendungsfall festgelegt werden.**
 - Definition erfolgt beim Entwurf der SOA.
 - Nachteil: Kommunikation wird schnell sehr komplex und dadurch schwer zu überwachen und zu debuggen.

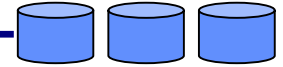
3.5 Serviceorientierte Architektur: Schichten – Grafisch



□ Schichten einer Serviceorientierten Architektur: Grafisch



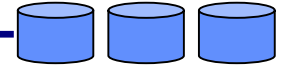
3.5 Serviceorientierte Architektur: Schichten – Beschreibung 1



□ **Schichten einer Serviceorientierten Architektur: Beschreibung**

- **Infrastruktur-Schicht:** Zusammenfassung aller bestehenden Systeme eines Unternehmens (Legacy-Systeme).
 - ◆ Legacy-System: Historisch gewachsenes System, das sich im Laufe der Zeit im Unternehmen etabliert hat. Müssen zumeist unverändert integriert werden.
- **Enterprise Komponenten:** Bestehen aus adaptierten Schnittstellen der Legacy-Systeme (ermöglichen es anderen Komponenten auf die Legacy-Systeme zuzugreifen) und neu entwickelten Systemen (komponentenorientiert entwickelt).
- **Service-Schicht:** Aus SOA Sicht die Kernschicht und beinhaltet die Services, welche von anderen Services als auch von darüberliegenden Geschäftsprozessen oder anderen Anwendungen verwendet werden können.
 - ◆ Atomare Services: Bieten Basisfunktionalitäten an.
 - ◆ Composite Services: Bieten komplexere Funktionen an und setzen sich aus mehreren atomaren Services zusammen.
 - ◆ Service Registry: Verzeichnis in dem alle Services registriert sind, befindet sich auch in dieser Schicht.
 - ◆ Fokus: Wiederverwendbarkeit der Services ist von großer Bedeutung.

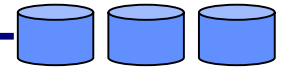
3.5 Serviceorientierte Architektur: Schichten – Beschreibung 2



□ Schichten einer Serviceorientierten Architektur: Beschreibung

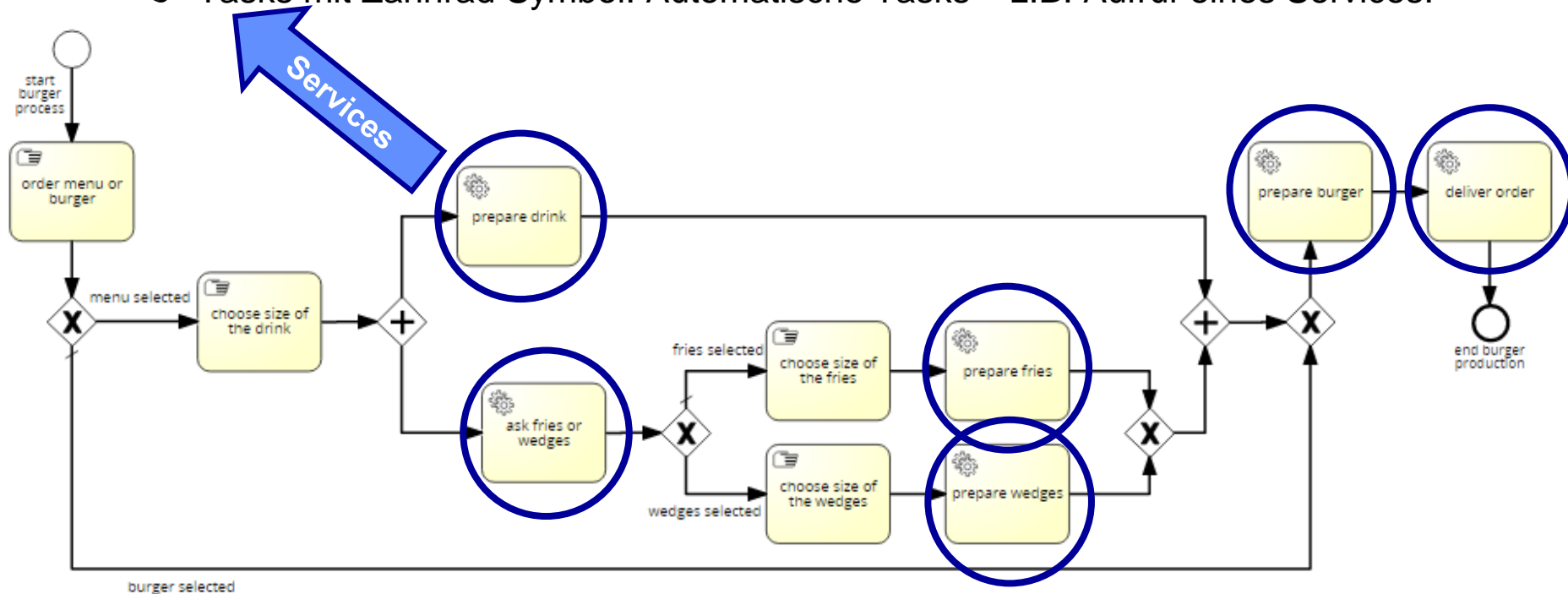
- **Prozess Schicht:** Abbilden der Geschäftsprozesse die in einer eigenen Prozess-Engine ausgeführt werden.
 - ◆ Prozesse können Services aus der Service-Schicht aufrufen.
 - ◆ Benutzer interagiert über die zur Verfügung gestellten GUIs mit dem Prozess.
- **Präsentationsschicht:** GUI-Komponenten, welche die Interaktion zwischen Benutzer und Backendsystem ermöglichen.
- **Integrationsschicht:** Ermöglicht die Integration von unterschiedlichen Systemen und bietet eine Menge von Diensten wie intelligentes Routing an.
 - ◆ Wird auch als Enterprise Service Bus (ESB) bezeichnet.
- **Quality-of-Service-Schicht:** Komponenten die Dienste für Security, Performance und Verfügbarkeit anbieten.
 - ◆ Es befinden sich auch Komponenten zur Steuerung und Überwachung der SOA-Plattform in dieser Schicht.

3.5 Serviceorientierte Architektur: Beispiel für die Geschäftsprozesssschicht

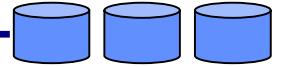


□ Beispiel für die Bestellung eines Burgers bzw. Menüs

- Tasks mit Hand Symbol: Müssen von einem Menschen durchgeführt werden.
- Tasks mit Zahnrad Symbol: Automatische Tasks – z.B. Aufruf eines Services.



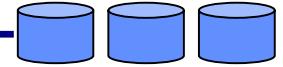
- Modelliert mit Signavio.



Serviceorientierte Architektur

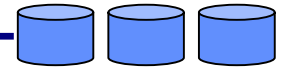
BUSINESS PROCESS MODEL AND NOTATION (BPMN)

3.5 Serviceorientierte Architektur: Exkurs – Vorteile von Prozessen






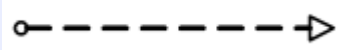
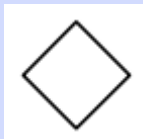
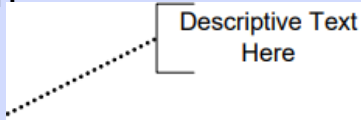
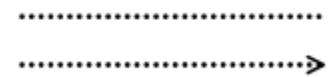


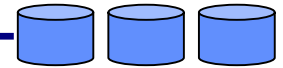
- ❑ **Prozess:** Sachlich logisch verbundene Folge von Aktivitäten die
 - Einen messbaren Nutzen bringen.
 - Einen Beitrag zur Erreichung der Unternehmensziele leisten.
 - Von allen Beteiligten nach bestimmten Regeln durchgeführt werden.
- ❑ **Vorteile**
 - Wartbarkeit wird erhöht da Logik stärker abstrahiert wird.
 - Lesbarkeit für Techniker aber auch Manager.
 - Parallelisierung wird vereinfacht da sich Abhängigkeiten einfacher beschreiben lassen.
- ❑ **Mögliche Visualisierung/Modellierung von Prozessen und im speziellen Geschäftsprozessen ⇒ Business Prozess Model and Notation (BPMN)**

3.5 Serviceorientierte Architektur: Exkurs – BPMN 1



- ❑ **Business Process Model and Notation (BPMN):** Grafische Modellierungssprache (Notation) zum standardisieren Abbilden von Geschäftsprozessen.
 - Entwickelt durch die Object Management Group (OMG).
 - Website: <http://www.bpmn.org/>
- ❑ **Elementkategorien (BPMN 2.0)** Quelle: <http://www.omg.org/spec/BPMN/2.0/>


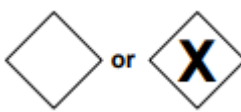




Flow Objects	Artefacts	Connecting Objects
Event 	Daten Objekte 	Sequence Flow 
Aktivitäten 	Group 	Message Flow 
Gateways 	Annotation 	Association 

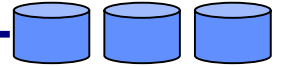


□ Business Process Model and Notation (BPMN): Fortsetzung

- Wichtigsten Events und Gateways

Quelle: <http://www.omg.org/spec/BPMN/2.0/>

Events		Gateways	
Start		Exclusive	
Intermediate		Inclusive	
End		Parallel	



3.1 Motivation

3.2 Grundlagen

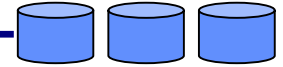
3.3 Entwurfsparadigmen

3.4 Muster (Architekturmuster und Entwurfsmuster)

3.5 Serviceorientierte Architektur

3.6 Zusammenfassung

3.6 Zusammenfassung



- ❑ **Historische Entwicklung von Softwarearchitekturen.**
 - Von einfachen 2-Tier und 3-Tier zu serviceorientierten Architekturen.
- ❑ **Im Entwurf werden Entscheidungen zur Mikro- und Makroarchitektur getroffen (taktisch vs. strategischer Entwurf).**
 - Makroarchitektur gibt den Rahmen für die Mikroarchitektur vor.
- ❑ **Mehrere Sichten und Abstraktionsebenen einer Architektur und deren Kommunikation.**
- ❑ **Entwurf: Transformieren von Anforderungen aus der Analysephase.**
- ❑ **Architektur- und Entwurfsmuster: Entwurf vereinfachen und bekannte Probleme effizient lösen.**