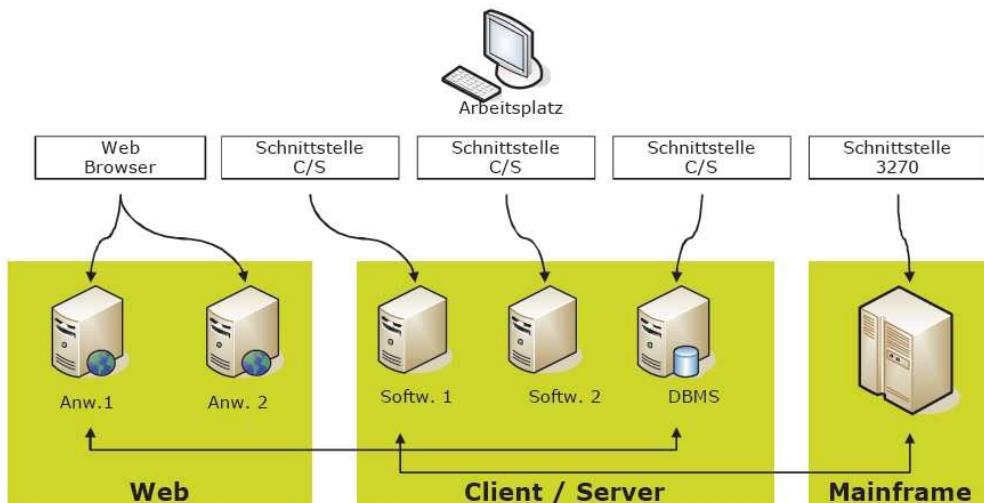


# Softwarekomponenten

## Software-Architektur

### Entropie

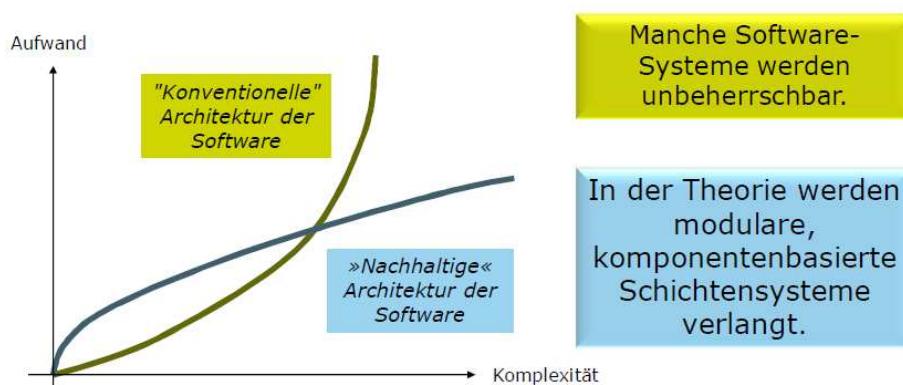


- Softwaresysteme werden immer grösser
- Moderne Softwaresysteme sind „verteilt“
- Geschäfts- und lebenskritische Prozesse werden zunehmend von Softwaresystemen geregelt
- Von Softwaresystemen wird erwartet, dass sie mit anderen Systemen interagieren
- Von Softwaresystemen wird verlangt, dass sie immer sicherer und zuverlässiger werden
- -> Softwaresysteme werden immer komplexer!!

### Anforderungen an das heutige Informationssystem

- Neue Zugänge auf Dienste eines Unternehmens
- Unterstützung von automatisierten Prozessen mit unterschiedlichen Akteuren
- Unterstützung von Unternehmensüberschreitenden Prozessen
- Praktisch andauernde Verfügbarkeit

### Komplexe System



## Notwendigkeit der Abstraktion

Um solche komplexe Systeme entwerfen, bauen, unterhalten und der Geschäftsentwicklung anpassen zu können, müssen diese formell dargestellt werden können. Da die Wirklichkeit oft kompliziert und durcheinander ist, werden Abstraktionen benötigt, um etwas Ordnung ins Chaos zu bringen. Ziel ist letztlich ein Informationssystem, das die Geschäftstätigkeit unterstützt.

Abstraktionen sind aber auch für weitere Zwecke nützlich:

- Definition, wie das Geschäft eigentlich läuft, z.B. für die Qualitätszertifizierung (ISO)
- Identifikation der Geschäftsprozesse, so dass diese überwacht, gemessen und verbessert werden können

Für die Modellierung des Geschäfts und deren Abbildung in IT-Systemen wurden unterschiedlichste Modelle vorgeschlagen, die jedoch kaum übereinstimmen. Es gibt keinen einheitlichen Industriestandard. Wir werden drei Arten von Abstraktionen besprechen:

- Frameworks
- Architekturen
- Modelle

## Frameworks

Das Framework gibt einen Rahmen, wie das Informationssystem einer Organisation beschrieben werden kann. „System“ meint hier nicht nur den automatisierten Teil, sondern schliesst auch die Menschen ein. Das Framework ist ein virtuelles Baugerüst, das Raum bietet, weniger abstrakte Notationen zu platzieren. Es kann kein physisches Beispiel davon erstellt werden, es unterstützt jedoch die Strukturierung der Entwicklung. Es gibt:

- Eine Checkliste der Dimensionen, die bei der Modellierung berücksichtigt werden
- Einen Kontext für das Verständnis der Zusammenhänge unterschiedlicher Eigenschaften des Informationssystems
- Ein konsistentes Grundvokabular in der Organisation

## Das Zachman Framework

Das Zachman Framework (1987) ist in einer zweidimensionalen Matrix angeordnet. Die **Zeilen** repräsentieren unterschiedliche Gesichtspunkte, oder Perspektiven, die eingenommen werden können, wenn auf das System geschaut wird:

Zeile	Perspektive	typisch für
1	Scope / Objectives (contextual)	Planer
2	Enterprise Model (conceptual)	Owner
3	System Model (logical)	Architect
4	Technology model (physical)	Designer
5	Detailed Representations	Builder
6	Functioning Enterprise	-

Die **Spalten** stehen für die unterschiedlichen Aspekte des Systems; den Fragen, die über das System gestellt werden können:

Spalte	Aspekt	Beantwortet die Frage
1	Data	What
2	Function	How
3	Network	Where
4	People	Who
5	Time	When
6	Motivation	Why

Das Zachman Framework:

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organizational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organizational Unit & Role Rel. Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location details	Event Details

## Architektur

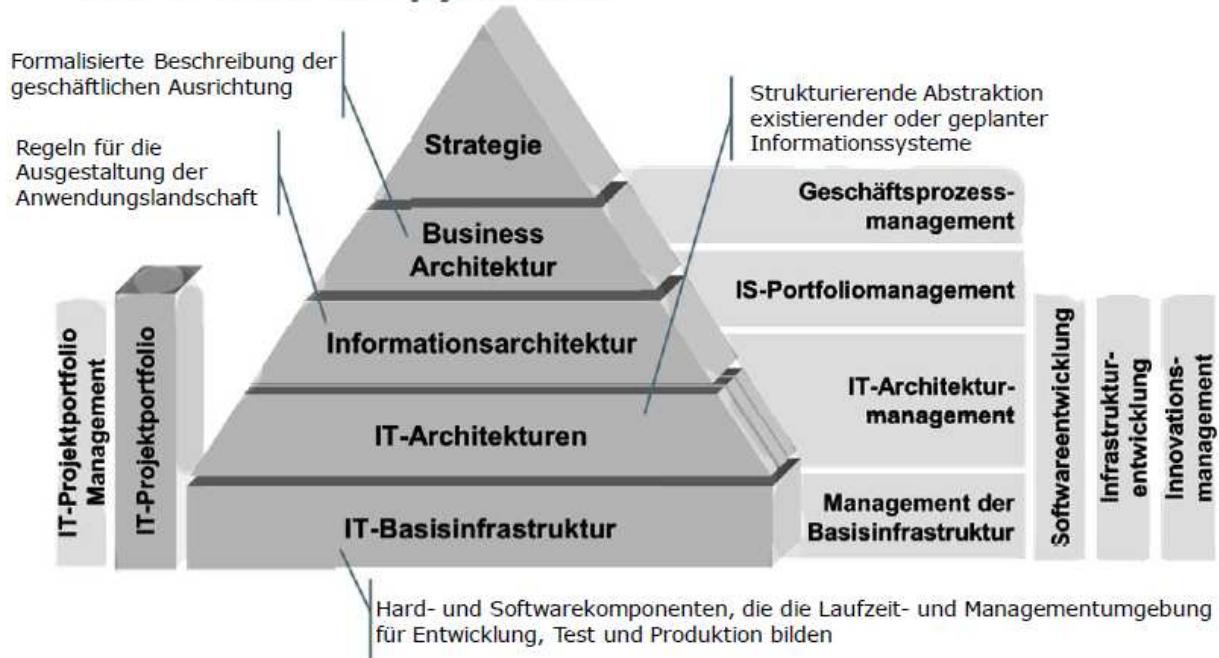
In der ITK können folgende 2 Punkte von einer Architektur erwartet werden:

- Definitionen der unterschiedlichen Arten von verfügbaren Bausteinen
- Regeln, wie diese zu komplexeren Gebilde zusammengesetzt werden

Gemeinsamkeiten	
Gebäude-Architektur	IT-Architektur
<ul style="list-style-type: none"> <li>- grosse Bauwerke mit langer Lebensdauer (Jahrhunderte)</li> <li>- grosse Auswirkungen auf viele Beteiligte</li> <li>- schwierige Koordination; komplexer, arbeitsteiliger Prozesse (Leistungsstufen)</li> <li>- Bedeutung von Planung und Modellen</li> </ul>	<ul style="list-style-type: none"> <li>- grosse Systeme mit langer Lebensdauer (Jahrzehnte)</li> <li>- grosse Auswirkungen auf viele Beteiligte</li> <li>- schwierige Koordination; komplexer, arbeitsteiliger Prozesse (Modelle)</li> <li>- Bedeutung von Planung und Modellen</li> </ul>

Unterschiede	
Gebäude-Architektur	IT-Architektur
<ul style="list-style-type: none"> <li>- Seit 10'000 Jahren</li> <li>- Wissenskanon, Werkzeuge</li> <li>- Mehr oder minder künstlerischer Anteil</li> <li>- Wenig Freiheitsgrade</li> </ul>	<ul style="list-style-type: none"> <li>- Seit 1968</li> <li>- Wenig Theorie, Werkzeuge</li> <li>- Sehr technisch, mathematisch ausgerichtet</li> <li>- Extrem viele Freiheitsgrade -&gt;abstrakt</li> </ul>

## Die Architekturpyramide



## Definition Informationssystem

Ein **Informationssystem** (Geschäftsanwendung, Geschäftssystem) ist eine Menge fachlicher und infrastruktureller Softwarebausteine, welche die Durchführung von Kern- oder Serviceprozessen unterstützen und die so zusammenspielen, dass die Aussensicht eines abgegrenzten, eigenständigen Systems in Form

wohldefinierter Schnittstellen (Interfaces) entsteht. Folgende Arten von Informationssystemen werden unterschieden:

- **Kernsysteme**  
Anwendungen, die ihren Nutzern betriebswirtschaftliche Funktionalität zur Verfügung stellen, um sie bei der Durchführung von Kerngeschäftsprozessen zu unterstützen.
- **Servicesysteme**  
Anwendungen, die darauf ausgerichtet sind, Kernsysteme durch die Bereitstellung zusätzlicher Dienste zu unterstützen.

Informationssysteme können durch die Definition von Subsystemen und Komponenten verfeinert werden. Diese bilden logische abgeschlossene Einheiten, die über sauber abgegrenzte Schnittstellen verfügen und die z. B. zur Festlegung von Verantwortlichkeiten genutzt werden

## Definition IT-Architektur

Eine **IT-Architektur** ist die strukturierende Abstraktion existierender oder geplanter Informationssysteme. Die Abstraktion schafft die gemeinsame Kommunikationsplattform aller an der Gestaltung von Informationssystemen Beteiligten, um so die Planbarkeit und die Steuerbarkeit der Gestaltung realer, miteinander in Wechselwirkung stehender Entitäten der IT eines Unternehmens zu erhöhen.

Architekturen können in Form von Referenzarchitekturen die übergreifende Struktur mehrerer IT-Architekturen und damit Gruppen von Informationssystemen prägen. Oder sie beziehen sich konkret auf ein einzelnes Informationssystem, Subsystem oder eine wesentliche Komponente wie den Produktserver einer Versicherung.

## Business-Architektur

Die **Business-Architektur** ist die formalisierte Beschreibung der geschäftlichen Ausrichtung eines Unternehmens oder Geschäftsfeldes.

**Business-Treiber** beschreiben die fundamentalen geschäftlichen Prinzipien eines Unternehmens und seiner Geschäftsfelder. In ihnen manifestiert sich die geschäftliche Ausrichtung eines Unternehmens und seiner Geschäftsfelder.

Die **Prozessarchitektur** eines Unternehmens definiert und strukturiert die Geschäftsprozesse eines Unternehmens. Sie umfasst die Aufteilung in Kernprozesse und Service Prozesse, die Darstellung der Kommunikationsbeziehungen nach innen und aussen sowie die Beschreibung der zu ihrer Durchführung notwendigen

**Funktionen** (Business-Funktionen) einschliesslich ihres **Informationsbedarfes**.

In der **Organisationsarchitektur** wird die Aufbau- und Ablauforganisation so gestaltet, dass die im Sinne der Business Treiber optimale Durchführung der Geschäftsprozesse Gewähr leistet ist.

## Informationsarchitektur

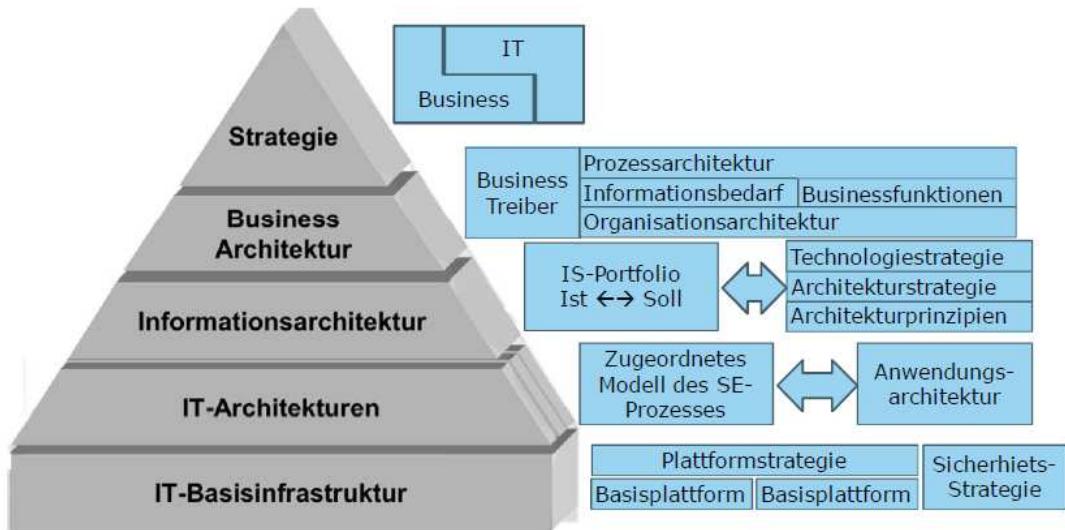
Die Informationsarchitektur eines Unternehmens definiert die strukturierenden Prinzipien und Regeln, die für die Gestaltung der Informationssystemlandschaft wegweisend sind. Die Informationsarchitektur umfasst folgende Elemente:

- IS-Portfolio: systematisch definierte Aufstellung der Informationssysteme eines Unternehmens einschliesslich ihres Zusammenwirkens.
- Technologiestrategie: legt die Leitlinien für Entwicklung der IT-Basisinfrastruktur fest.
- Architekturstrategie: beschreibt in einer kurzen und prägnanten Form den Zielzustand, der durch das Management der IT-Architekturen eines Unternehmens erreicht werden soll einschliesslich des Weges zu seiner Erreichung.
- Architekturprinzipien: alle architekturbbezogenen Grundsätze und übergreifenden Standardisierungen verstanden, die für die Weiterentwicklung des IS-Portfolios eines Unternehmens und der dazu notwendigen IT-Architekturen gelten

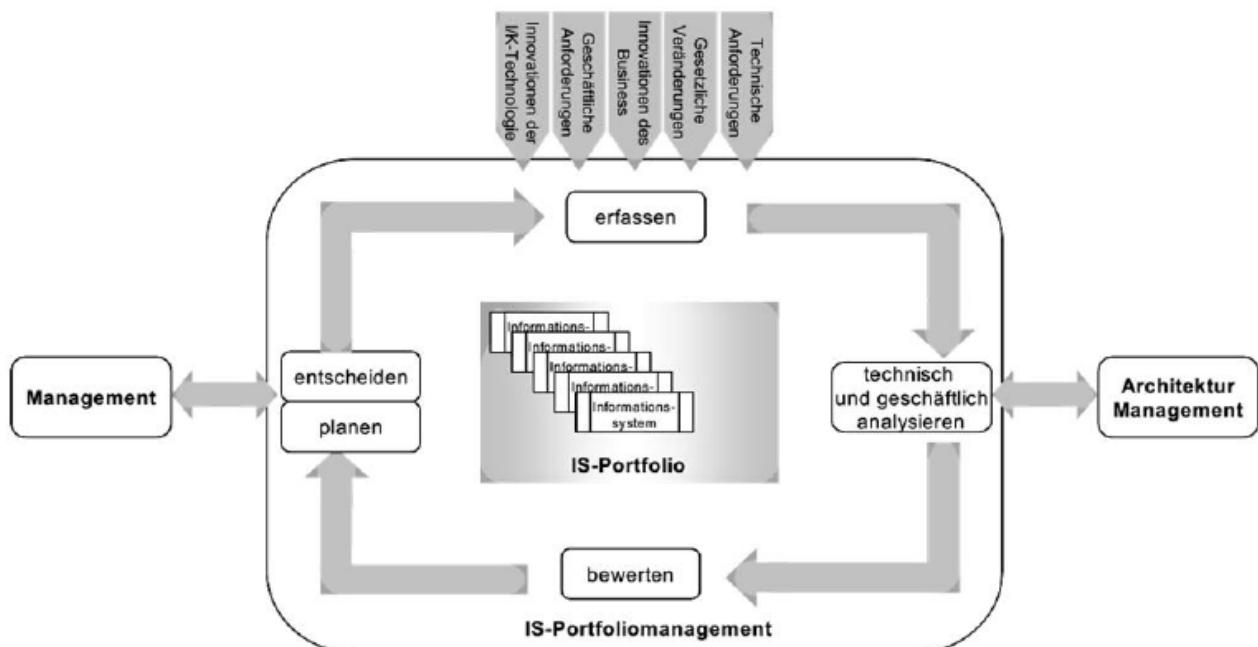
## IT-Basisarchitektur

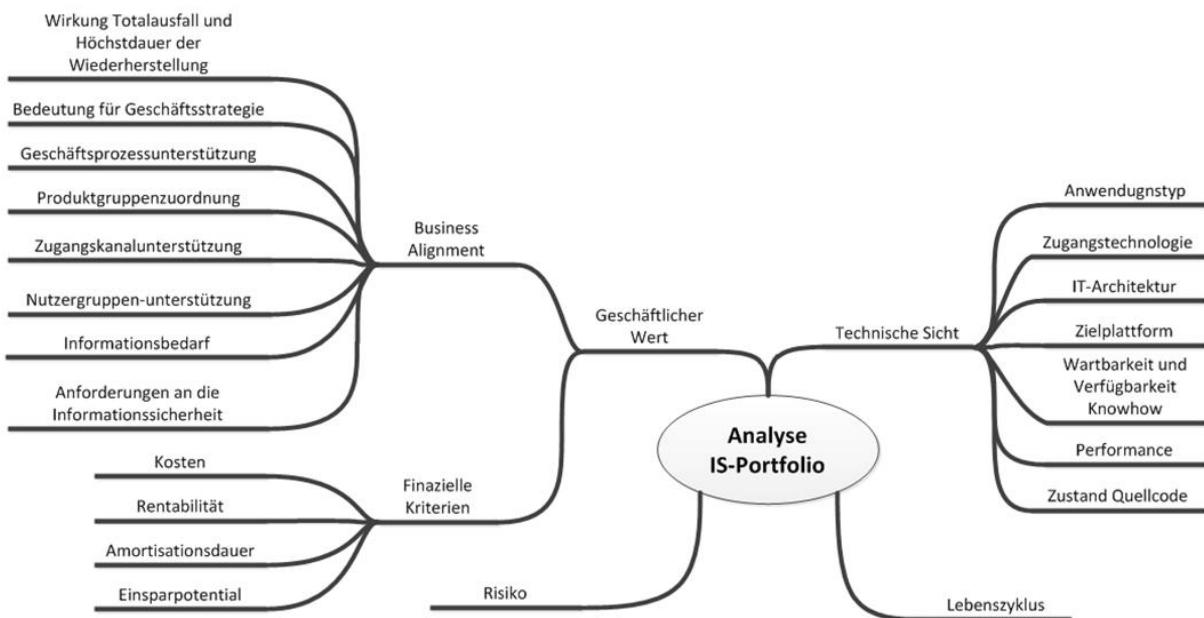
Als **IT-Basisinfrastruktur** wird die Menge aller Hardware- und aller systemnahen Softwarekomponenten verstanden, die die Laufzeit- und Managementumgebung für Entwicklung, Test und Produktion von Informationssystemen bilden. Diese Komponenten werden zu Basisplattformen gruppiert und bilden die Zielplattform von Informationssystemen. Die IT-Basisinfrastruktur umfasst die **Plattform-** und die **Security-**Strategie.

## Verfeinerung der Architekturpyramide

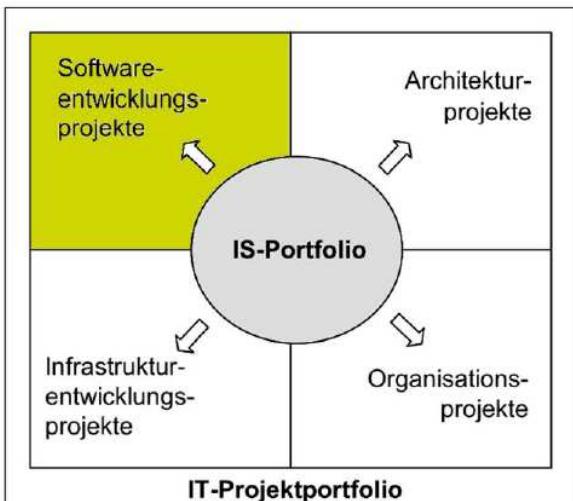


## IS-Portfoliomangement





### Vom IS- zum IT-Projektportfolio



### Bestandteile einer Software-Architektur

- Beschreibung der Struktur grosser Softwarehersteller
- Abstrakte Sicht mit Verzicht auf Implementations-, Algorithmen- und Datenrepräsentationsdetails
- Konzentration auf das Verhalten und die Interaktion von „Black-Box“-Elementen
- Erster Entwurfsschritt in Richtung eines Systems mit der gewünschten Eigenschaften

## Ziele der SW-Architektur



## Inputs und Outputs der SW-Entwicklung

Sehr allgemein basiert die Entwicklung eines Informationssystems aus 2 Inputs:

1. Einer strukturierten Beschreibung dessen, was das Geschäft tun will
2. Einer Definition der Technologie, die gebraucht wird, dies zu erreichen

und aus 2 Outputs:

1. Einer lesbaren Beschreibung der Geschäftsdefinitionen
2. Spezifischen Softwarekomponenten, die zusammen ein zweckmässiges Informationssystem bilden

## Komponenten Architektur

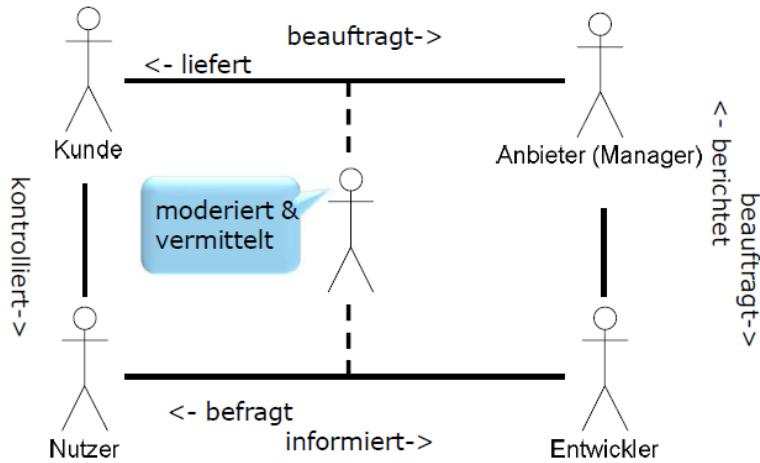
Beschreibt den technologischen Output des Entwicklungsprozesses, die Grundlagen der modularen Einheiten, die bereitgestellt werden, um die Informationsbedürfnisse für ein Unternehmen zu befriedigen. Sie beschreibt, wie die unabhängig freigegebenen Elemente zusammen verwaltet und zusammengesetzt werden, damit ein funktionierendes System entsteht.

## Abhängigkeiten der Architekturen

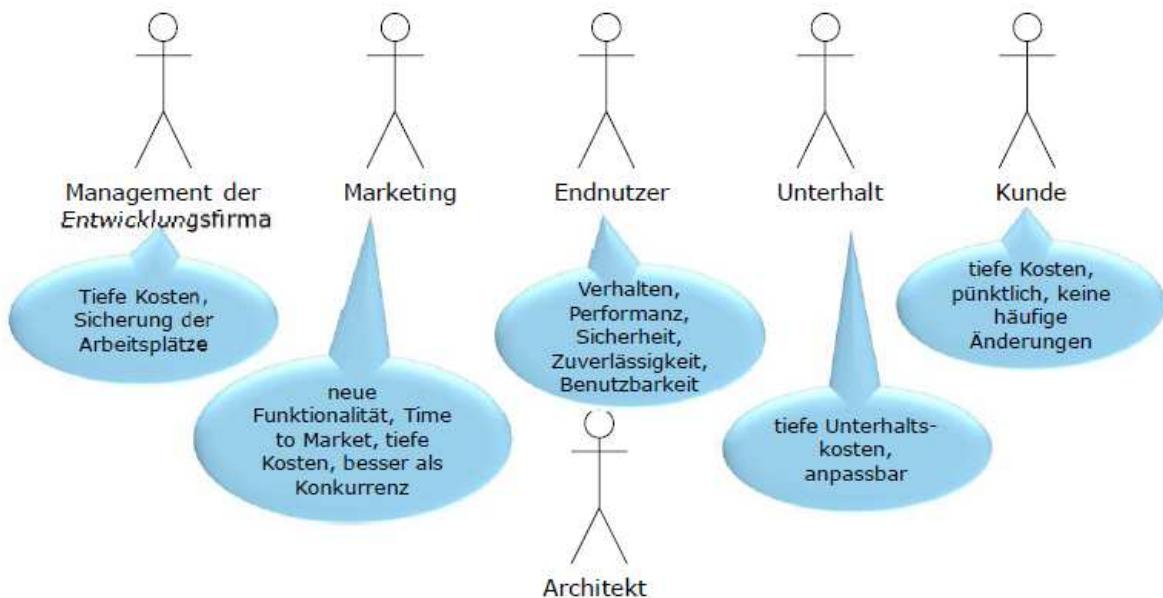
Jede Architektur beschreibt etwas anderes. Die Architekturen decken Zeilen 2,3 und 4 vom Zachman-Framework ab. Für die übrigen Zeilen brauchen wir keine Architekturen:

- Zeile 1 besteht aus unstrukturierten Beschreibungen, was im Scope des Systems liegt
- Zeile 5 enthält detaillierte, technologische Artefakte mit gegebenen Namen
- Zeile 6 ist das funktionierende Unternehmen selbst

## Rolle des Software-Architekten



## Stakeholders in der Architektur



## Aktivitäten zur Erstellung einer Software-Architektur

- Geschäftsfall erstellen (was will ich? Ziel?)
- Anforderungen verstehen
- Architektur auswählen/erstellen/analysieren/evaluieren/dokumentieren/veröffentlichen
- System auf Architektur basierend implementieren
- Übereinstimmung von Architektur und System sicherstellen

## Einige Begriffe

- **Komponenten:** Architektur-Einheiten, also als Gegenstand von Entwurf, Wiederverwendung, Auftragsvergabe, Versionskontrolle, ...
- **Port:** Schnittstelle einer Komponente
- **Rolle:** beschreibt Signatur und Verhalten
- **Protokoll:** Kollaboration von Rollen

## Weshalb Modelle?

Die Motivation für die Geschäftsmodellierung ist eng mit der Anforderungsanalyse verknüpft.

- Ziel ist die Definition, **was** getan werden soll
- Gute Anforderungen vermeiden das **wie**, sagen nichts über die Technologie wie Windows oder Java aus; Gründe:
  - Agilität
  - Technologische Freiheit, kein "lock in" des Business aufgrund eines Technologieentscheids
  - Ein Modell ist keine Realität, immer eine Vereinfachung
  - Ein Modell ist ein hervorragendes Mittel, um den **Scope** eines Vorhabens zu klären
  - In einigen Fällen braucht es ein klares Bild des aktuellen Geschäfts
  - Ist-Modell
  - In den meisten Fällen sollte eine Sicht auf zukünftige Geschäftstätigkeiten erstellt werden
  - Soll-Modelle

## Strukturen und Notationen für Modelle

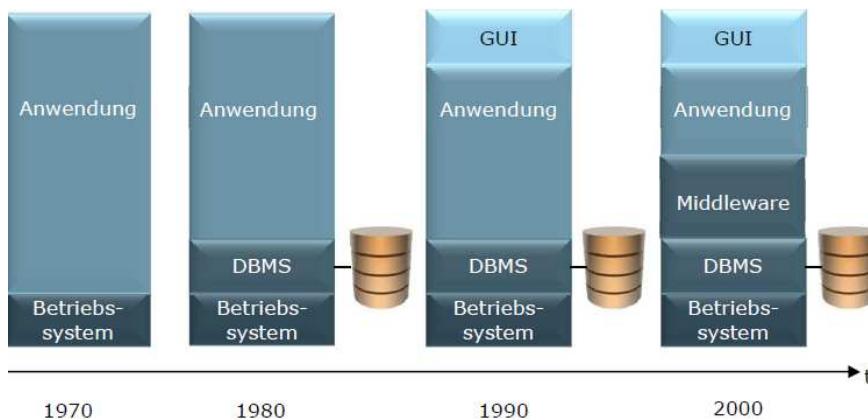
Für unsere Zwecke ist es nützlich, dass Modelle in menschenlesbaren Notationen erstellt werden.

Was immer wir dafür benutzen, muss zwei Probleme lösen:

- Wir können mit einer einzigen Beschreibung nicht alles lösen
- Die Beschreibung muss genügend ausdruckstark sein, um den Zweck zu erreichen, jedoch so einfach, dass sie von unterschiedlichen Ansprechpartnern verstanden wird

Dazu wurden viele Notationen entwickelt. In der Softwareentwicklung hat sich jedoch in der letzten Zeit die **Unified Modeling Language (UML)** als Standard durchgesetzt.

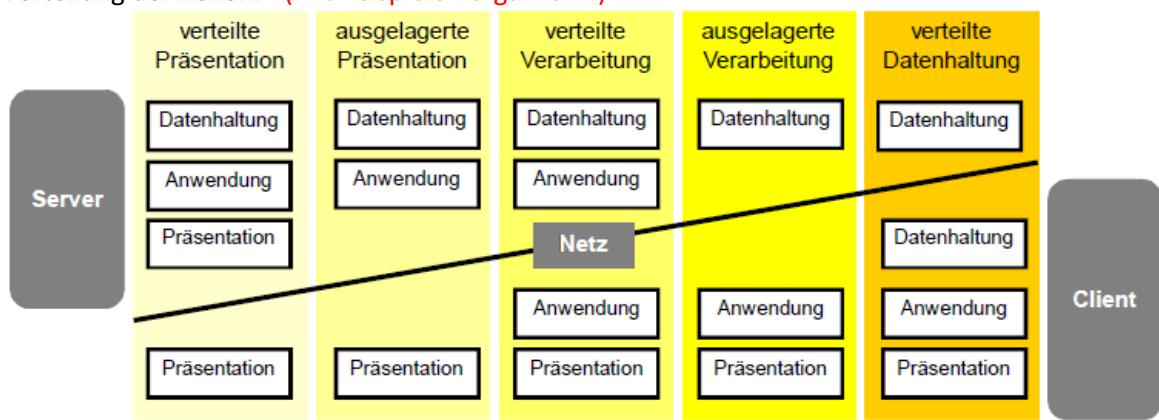
## Trends in der IT-Geschichte



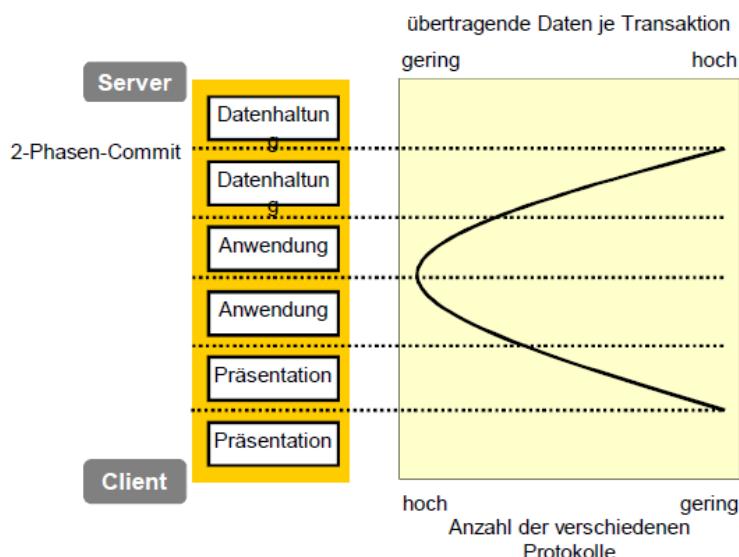
## Client – Server Architektur

- **Client:** aktive Kommunikationspartner
- **Server:** passiver Kommunikationspartner
- **Datentransfer:** meist TCP/IP + weiteres Protokoll wie beispielsweise http

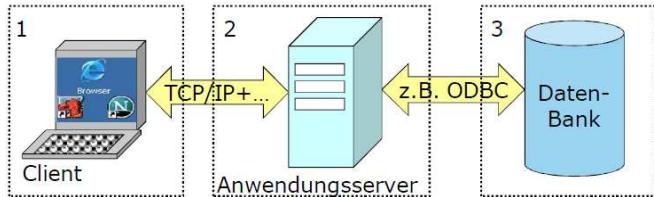
Verteilung der Rollen: (mit Beispielen ergänzen!!)



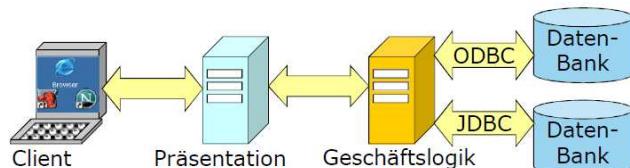
Bewertung unterschiedlicher Verteilungsarten:



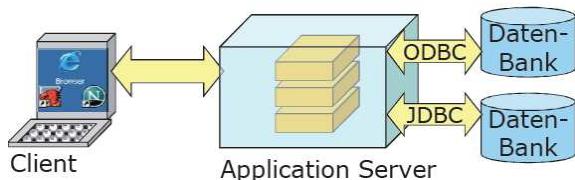
### 3 Schichten Architektur



### 4 Schichten Architektur



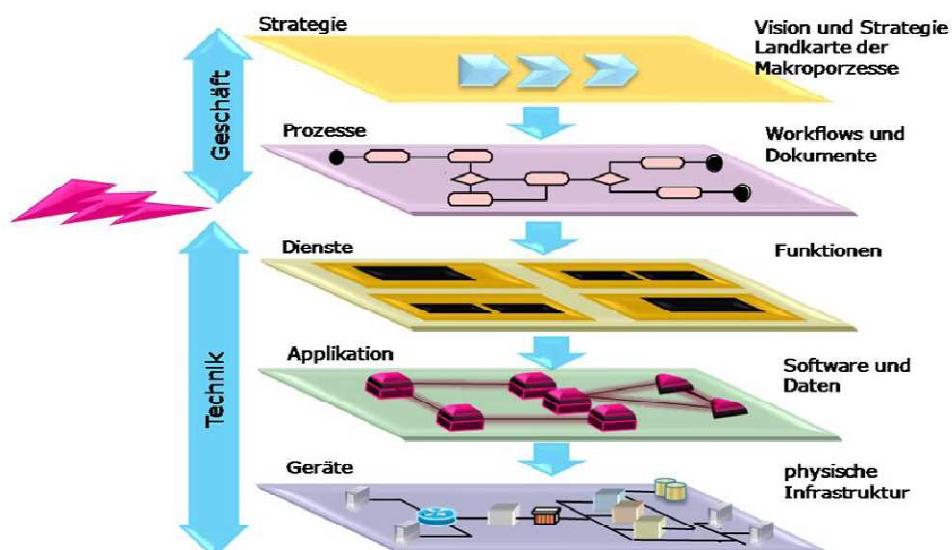
### Application Server



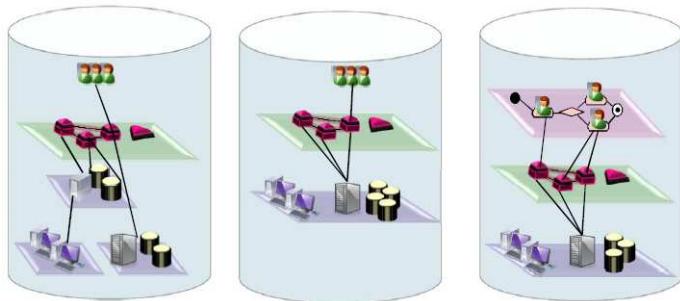
- Mehrere Anwendungen integrieren
- Automatische Zuordnung von Anwender und Anwendung
- Einheitliche Softwareschnittstelle

### Urbanisation

Strukturiertes Vorgehen zur Erstellung einer Beherrschbaren IT-Landschaft (Inspiriert vom Städtebau). Abstraktion der Gesamt-IT in unterschiedliche, definierte Layer mit klaren Schnittstellen.  
Referenzmodell:

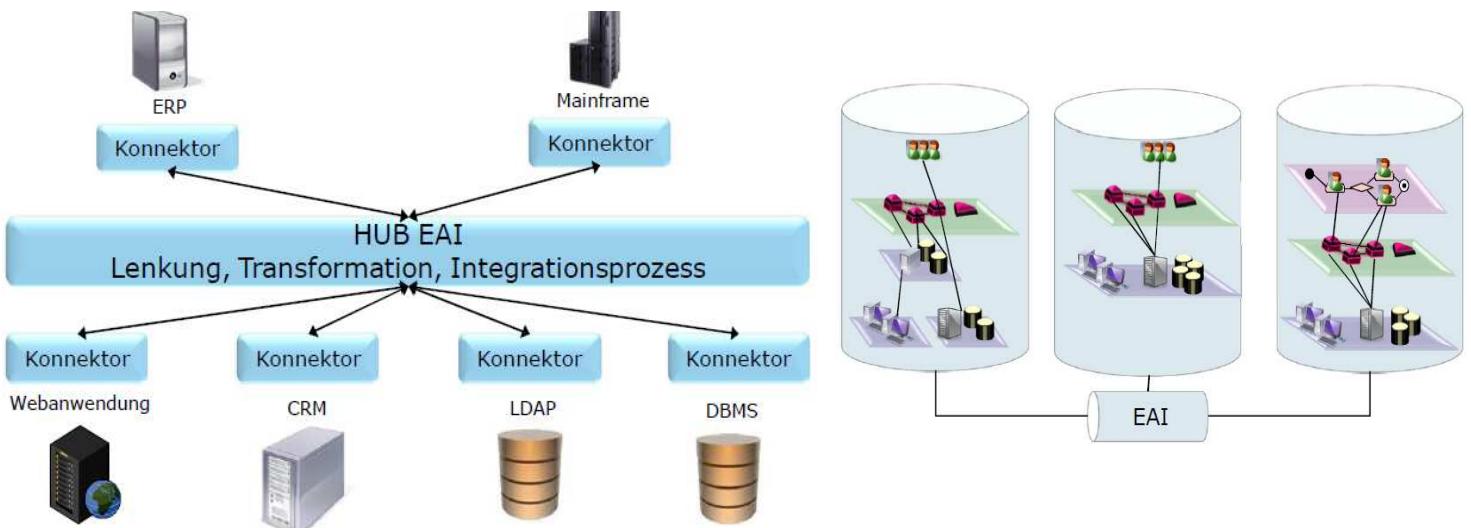


## Informatiksilos als Stolpersteine



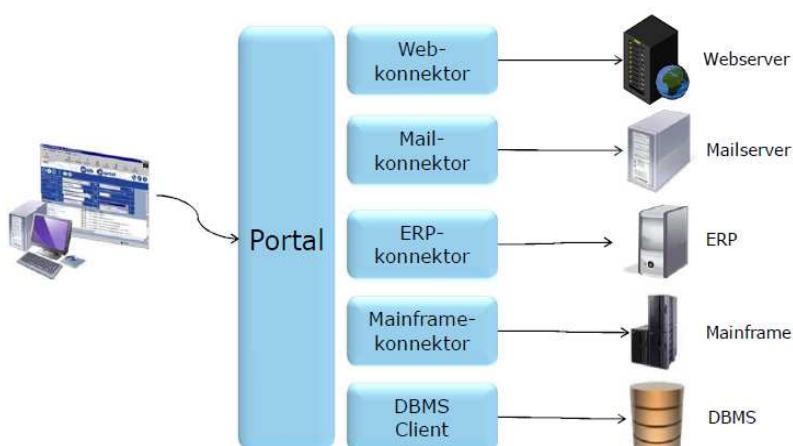
Die Inseln einzeln miteinander zu Integrieren resultiert in einer rigiden, fest «verdrahteten» Informatiklandschaft.

### Lösungsversuch 1 (Enterprise Application Integration):



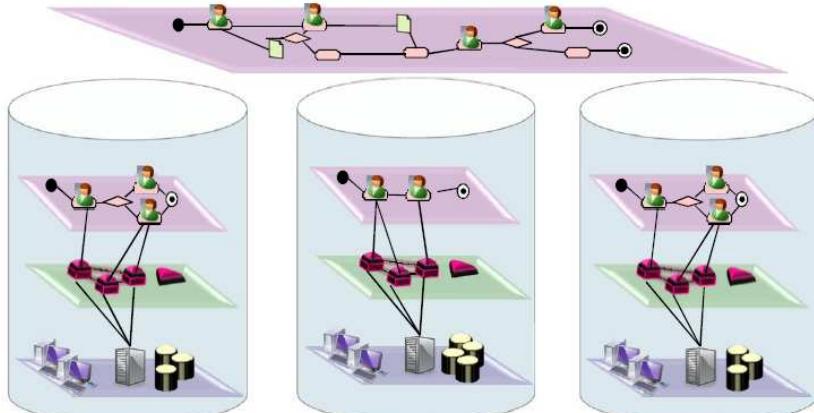
→EAI ist ein "Palleatifmittel", löst das Problem jedoch nicht...

### Lösungsversuch 2 (Integration über Webportale):



→Auch Webportale der ersten Generation fördern Silos!!!

Lösungsversuch 3 (Workflowwerkzeuge):



- traditionelle Workflowwerkzeuge sind meist Anwendungs proprietär fördern die IT-Silos

### Technische Anforderungen an eine IT-Landschaft

- Unterstützt eine **rasche** Umsetzung von Geschäftsprozessen
- Das Geschäft benötigt eine **Echtzeitsicht** auf das Business
- Informatikinvestitionen in Phasen **parallel** zu den Geschäftsbedürfnissen
- **Investitionsschutz** und Wiederverwendbarkeit
- Unterstützt unterschiedliche Informationsaustauschmechanismen:
  - Synchron, Asynchron
  - Publish and Subscribe
  - Ereignisgetrieben
- Sichert den Austausch von Informationen zwischen Anwendungen
- Stellt die Informationszustellung sicher
- Verwaltet verteilte **Transaktionen**

## Softwarekartographie

Anwendungslandschaften in Unternehmen sind langlebige hoch-komplexe Strukturen bestehend aus hunderten bis tausenden von miteinander vernetzten betrieblichen Informationssystemen, die von Personen mit sehr unterschiedlichen Interessen und Erfahrungshintergrund konzipiert, erstellt, modifiziert, betrieben, genutzt und finanziert werden.

Die Softwarekartographie zielt darauf ab, die Kommunikation zwischen diesen Personen durch zielgruppenspezifische verständliche graphische Visualisierungen zu unterstützen, die Geschäfts- und Informatik-Aspekte gleichermaßen berücksichtigen, und die speziell für langfristige und strategische Management-Betrachtungen geeignet sind.

### Management von Anwendungslandschaften

Geschäft und Management beklagen die geringe Kosten- und Nutzentransparenz der Anwendungslandschaft. Umgekehrt beklagen Mitarbeiter der IT das Desinteresse bei Geschäft und Management, sich mit zentralen Fragen der Gestaltung der Anwendungslandschaft auseinander zu setzen. Weiterhin fehlt aus Sicht der IT-Verantwortlichen häufig eine ausreichende Konkretisierung unternehmensweit gesetzter strategischer Geschäftsziele, um die von der IT verlangte strategische „Ausrichtung am Geschäft“ zu leisten (**Business – IT Alignment**).

Die Verantwortlichkeiten sind oft unklar: Es gibt keine nachhaltige Dokumentation der Eigentümer für Prozesse, Anwendungen, Schnittstellen und Dienste, oder es fehlen daraus abgeleitete verbindliche Rechte und Pflichten für IT und Geschäft (**IT Governance**).

Wesentliche technologische und konzeptuelle Fortschritte durch SOA, modellgetriebene Entwicklung, domänenpezifische Sprachen und Architekturen, Middleware-Produkte und Server-Virtualisierung versprechen, die Anpassbarkeit der Anwendungslandschaft an vorab nicht genau spezifizierte Anforderungen zu verbessern (**IT Agility**).

### Kernkonzepte der Softwarekartographie

Eine **Clusterkarte** fasst die zu verortenden Elemente (z.B. Anwendungssysteme, Services oder Datenbanken) in logischen Domänen zusammen. Die Domänen ergeben sich aus Funktionsbereichen, Organisationseinheiten oder Standorten.

Eine **kartesische Karte** verwendet zwei Achsen (X/Y). Jede Achse ist in Intervalle aufgeteilt, für die eventuell eine Ordnungsbeziehung oder weitergehend eine metrische Abstandsfunktion existiert. Diese definiert dann die Reihenfolge und Breite der Intervalle. Ein Element wird auf einer solchen Karte platziert, indem für das Element ein oder ggf. auch mehrere (dann meist benachbarte) zugehörige X- und Y-Intervalle bestimmt werden und das Element innerhalb der Schnittfläche(n) der Intervalle platziert wird.

Eine **Prozessunterstützungskarte** wählt als X-Achse die Wertschöpfungskette des Unternehmens. Zur Verortung auf der Y-Achse können bei einer Prozessunterstützungskarte verschiedene Merkmale zum Einsatz kommen (Organisationseinheiten, Standorte, Produkte oder Kategorisierung in dispositive, operative und administrative Systeme).

Prozessunterstützungskarten sind besonders hilfreich, um Potenziale für vertikale und horizontale Integration in gewachsenen Anwendungslandschaften zu identifizieren. Ziel einer vertikalen Integration ist es, an mehreren Standorten oder für mehrere Produkte einheitliche Systeme zu etablieren, um Kostenvorteile zu erzielen.

Bei einer **Zeitintervallkarte** wird als X-Achse die Zeit als intervallskaliertes Merkmal gewählt. Auf der Y-Achse bieten sich bei diesem Kartentyp wiederum verschiedene Elemente an, die mit einem Zeitbezug dargestellt werden sollen. Z.B. die Versionen eines Anwendungssystems stehen in einer Vorgänger-/Nachfolgerbeziehung. Für jede Anwendungssystemversion wird ihr Status mit den entsprechenden Zeitintervallen dargestellt.

Bei einer **Graphlayout-Karte** besitzt die Position eines Elements keine Bedeutung (vgl. UML Informatik Diagramme, E/R-Diagramme, Petrinetze).

### Gestaltungsregeln und Schichtenprinzip

Die Softwarekartographie verwendet die Konzepte der Kartographie, speziell die der Thematischen Karten: Gestaltungsmittel sind die graphischen Grundelemente (Punkt, Linie und Fläche) sowie die zusammengesetzten Zeichen, wie Signatur, Diagramm, Halbton oder Schrift.

Gestaltungsmittelinstanzen sind Instanzen eines Gestaltungsmittels und verhalten sich zu ihnen wie Objekte zu Klassen. Gestaltungsvariablen beeinflussen die Erscheinung einer Instanz eines Gestaltungsmittels auf einer konkreten Karte, indem die Variable die Instanz hinsichtlich Größe, Form, Füllung, Tonwert, Richtung und Farbe verändert.

Darüber hinausgehend definiert die Softwarekartographie Gestaltungsregeln, dies sind Darstellungsbeschränkungen (verpflichtende Bedingungen) oder Darstellungswünsche (wünschenswerte Bedingungen).

Softwarekarten als graphische Repräsentationen von Anwendungslandschaften sollen Anwendungssysteme sowie relevante Merkmale und Beziehungen zwischen Anwendungssystemen visualisieren. Merkmale und Beziehungen können fachlich gruppiert und jeder Gruppe kann eine Schicht zugewiesen werden. Durch Ein-/Ausblenden von Schichten und durch Vergrößern/Verkleinern können die angezeigten Informationen gefiltert und die Informationsdichte variiert werden, um für einen bestimmten Anwendungsfall die gewünschte Softwarekarte zu erhalten.

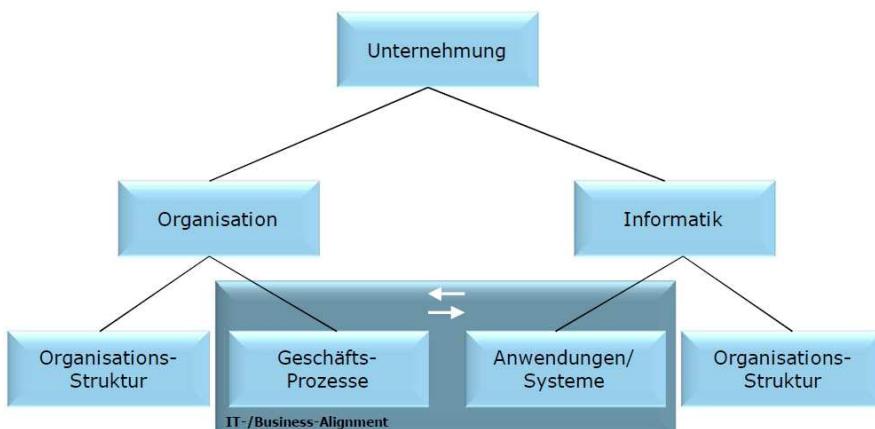
### Versionisierung von Karten und Modellen

Wie bei der Stadtplanung, so haben auch beim Management von Anwendungslandschaften praktisch alle Modellinformationen (Objekte, Beziehungen, Attribute) einen Zeitbezug. Es hat sich bewährt, diskrete Planzyklen (1–2 mal pro Jahr) und entsprechende „Schnappschüsse“ der Anwendungslandschaft einzuführen.

Die Softwarekartographie berücksichtigt diese Anforderungen, indem jede Karte neben dem eigentlichen Kartenfeld weitere Meta-Informationen besitzt: Kartentitel, Kartentyp, Autoren, Status, Erstellungsdatum, Zeitbezug (Ist, Plan, Soll) und Legende, die verwendete Gestaltungsmittel, Gestaltungsvariablen und Gestaltungsregeln erläutert.

## SOA – Software Oriented Architecture

**Serviceorientierte Architektur (SOA)**, auch **dienstorientierte Architektur**, ist ein Architekturmuster der Informationstechnik aus dem Bereich der verteilten Systeme, um Dienste von IT-Systemen zu strukturieren und zu nutzen. Vereinfacht kann SOA als Methode angesehen werden, die vorhandenen EDV-Komponenten wie Datenbanken, Server und Websites so in Dienste zu kapseln und dann zu koordinieren („Orchestrierung“), dass ihre Leistungen zu höheren Diensten zusammengefasst und anderen Organisationsabteilungen oder Kunden zur Verfügung gestellt werden können. Maßgeblich sind also nicht technische Einzelaufgaben wie Datenbankabfragen, Berechnungen und Datenaufbereitungen, sondern die Zusammenführung dieser IT-Leistungen zu „höheren Zwecken“ – wie Ausführen einer Bestellung oder Prüfen der Rentabilität einer Abteilung usw. –, die eine Organisationsabteilung anbietet.



### Was ist ein Dienst?

Ein Dienst im Sinne einer SOA stellt den Akteuren (Menschen oder Anwendungen) innerhalb von Geschäftsprozessen den Zugriff auf eine oder mehrere Geschäftsfunktionen zur Verfügung. Der Dienst realisiert die Verbindung zwischen der Geschäftsschicht (Rolle des Abnehmers) und der Implementierung im IS (Rolle des Lieferanten), indem er für den Austauschkontrakt verantwortlich zeichnet (Rolle des Vermittlers). Die Zusammenstellung der Funktionen müssen **aus Geschäftssicht sinnvoll sein**; der Abnehmer des Dienstes hat sich nicht mit der Art der Implementierung und deren Technik zu kümmern...

### Anforderungen an einen Dienst

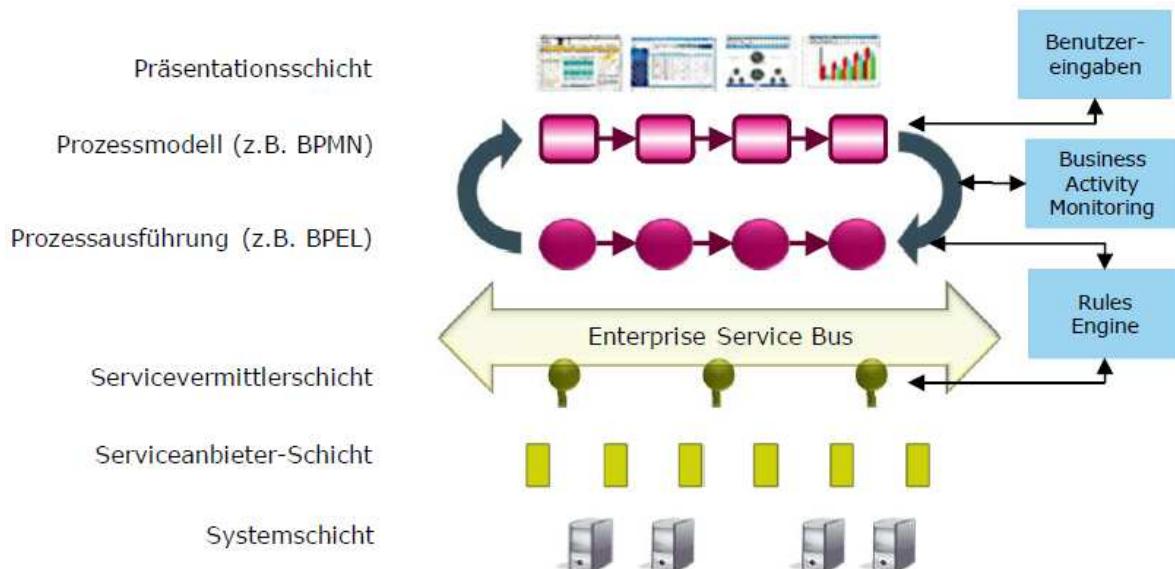
Ein SOA-Dienst muss interoperabel sein; er kommuniziert mit den Abnehmern auf eine **standardisierte** Art, sowohl auf technischer als auch auf geschäftlicher Ebene. Aus Sicht des Abnehmers muss der Dienst einen **Mehrwert** bieten, sowie eine Qualitätsgarantie. Aus Sicht des Vermittlers muss der Dienst **ständig verbessert** werden, um den Mehrwert zu bieten:

- einfache Handhabung (Homogenität)
- Performant
- Bietet eine messbare Qualität

## Weitere Elemente einer SOA

- ESB : Enterprise Service Bus
- Registry und Repository
- BRE: Business Rules Engine
- BAM: Business Activity Monitoring
- Benutzerinteraktion

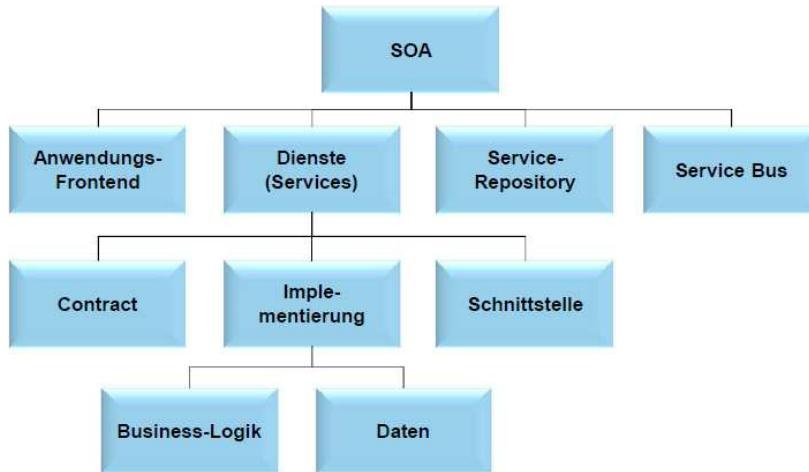
## Beispiel einer möglichen Architektur



## Einige grössere SOA-Produkte

Anbieter/Prod.	Kurzbeschreibung
IBM WebSphere	SOA-Lösung, deren Basis der WebSphere Enterprise ESB bildet. Weitere Bestandteile sind der Application Server, das WebSphere Portal, die Business Services Fabric, der Business Modeler, der Business Monitor, der Process Server und das Produkt Service Registry and Repository.
Microsoft Biz Talk Server 2009	SOA-Lösung, die in Kombination mit dem Windows Server 2008 mit .NET-Framework, dem SharePoint Portal Server 2007, eingesetzt wird.
Oracle SOA Suite	Paketlösung, die mit Produkten für Portalserver, Application Server und Adapter für Standardanwendungen kombiniert werden.
SAP® Enterprise Service Architecture	SOA-Lösung auf Basis der Plattform SAP NetWeaver in Kombination mit einem Service Repository und einer Auswahl an Services
Tibco ActiveMatrix BusinessWorks	Business Works als Enterprise Service Bus in Kombination mit weiteren Tibco-Produkten, wie z.B. dem Portal Builder und der Staffware Process Suite
RedHat JBoss Enterprise Middleware Suite	OpenSource Enterprise Middleware Suite von JBoss umfasst ein komplett Umgebung zur Realisierung von SOA mit analogen Elementen wie diese z.B. bei IBM gefunden werden können.
Apache Geronimo	Application Server Framework mit OpenSource ESB ServiceMix und diversen weiteren Komponenten.
IONA FUSE	IONA bietet mit FUSE eine Open Source-Distribution an, die den ESB und das Messaging System von Apache mit dem Application Server WebSphere AS Community Edition von IBM kombiniert

## Bestandteile einer SOA



## Klassen von Diensten

- Anwendungs-Frontend
  - Anwendungs Frontend** sind keine eigentliche Dienste. Es sind aktive Elemente einer SOA. Sie initiieren die Geschäftsprozesse und erhalten die Resultate. Typische Beispiele sind GUIs oder Batch-Prozesse
- Basis-Dienste
  - Basis-Dienste** sind die Grundlage einer SOA. Sie repräsentieren die Grundelemente einer vertikalen Domäne. Basisdienste können sowohl Daten- als auch Prozesszentriert sein.
- Zwischen-Dienste
  - Zwischen-Dienste** setzen sich zusammen aus Technologie-Gateways, Adapters, Facades und funktionsanreichernde Dienste. Ähnlich der prozesszentrierten Dienste sind sie sowohl Client als auch Server in einer SOA. Ungleich prozesszentrierten Diensten sind sie jedoch zustandslos.
- Prozess-zentrierte Dienste
  - Prozesszentrierte Dienste** enthalten das Wissen über die Geschäftsprozesse. Sie sind typischerweise sowohl Client als auch Server einer SOA und unterhalten den Prozesszustand.
- Öffentliche Unternehmensdienste
  - Öffentliche Unternehmensdienste** bieten die Schnittstellen für eine unternehmensübergreifende Integration an. Sie sind deshalb meist grobkörniger und müssen geeignete Mechanismen für Unabhängigkeit, Robustheit, Sicherheit und Verrechenbarkeit anbieten

## Rolle von XML in einer SOA

Die technische Standardisierung benötigt ein Vokabular, das das Format und die Struktur der ausgetauschten Daten spezifiziert. Die Metasprache XML ist zwar im strikten Sinne nicht Voraussetzung für eine SOA, hat sich aber de facto als «lingua franca» durchgesetzt.

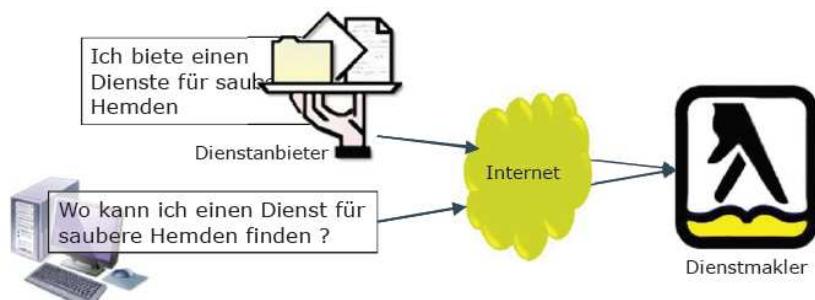
Die geschäftliche Standardisierung benötigt zwingend die Verbreitung einer Sprache, die die Semantik der ausgetauschten Daten beschreibt. Einige Anwendung in XML (RDF, OWL, RulesML, ...) gehen in diese Richtung.

## Standards für Web-Services

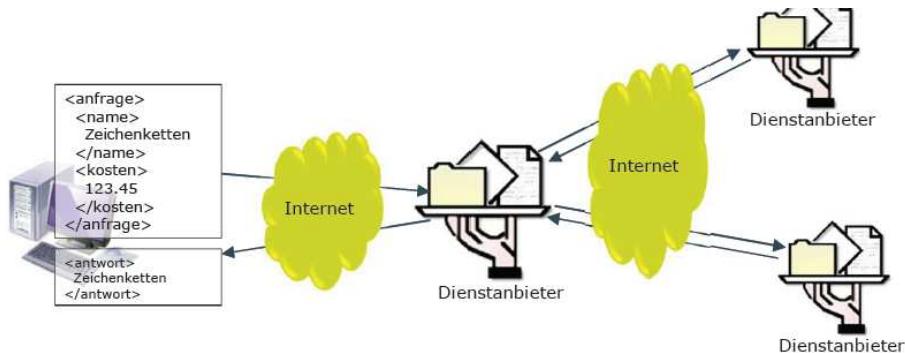
### Was sind Web-Services?

Ein Web Service ist ein durch einen URI eindeutige identifizierte Softwareanwendung, deren Schnittstellen als XML-Artefakte definiert, beschrieben und gefunden werden können. Ein Web Service unterstützt die direkte Interaktion mit anderen Softwareagenten durch XML-basierte Nachrichten, die über Internetprotokolle ausgetauscht werden."

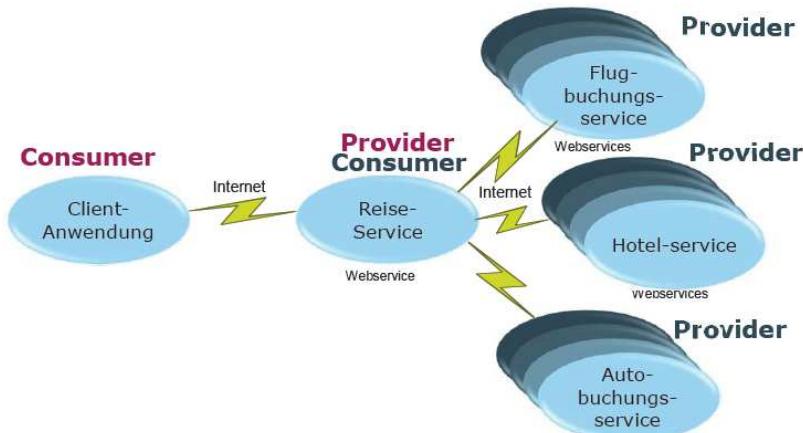
- Web Services bieten eine auf Standards basierende Technologie, um SOAs zu realisieren
- Web Services (WS) ermöglichen Web-basierte Remote Procedure Calls
- WS sind einer der Hypes der letzten Jahre
- Sind eine relative junge Technologie (seit ca. 2000 im Zusammenhang mit Microsoft .Net)
- Werden heute in umfangreichen Masse eingesetzt
- Die Weiterentwicklung der Standards rund um Web Services wird (hauptsächlich) vom W3C (World Wide Web Consortium) vorangetrieben
- Web Services sind **plattform-** und **implementierungsunabhängige** Softwarekomponenten:
  - heisst, der Client muss nicht wissen, was für eine Sprache, Betriebssystem oder Computertyp der Server verwendet.
  - heisst, dass (von einigen Ausnahmen abgesehen) binäre Daten weder gesendet noch empfangen werden.
- mit einer Web Service Description Language beschrieben (WSDL):
  - heisst, dass ein Web Service selber **beschreibt**, welche Anfragen gemacht werden können, was die Argumente sind und welches Transportmedium verwendet wird.
- auf einem "registry of services" registriert,
  - heisst, ein Web Service kann einem "registry service" sagen, wo er gefunden werden kann ("Gelbe Seiten")
- durch einen standardisierten Mechanismus gefunden,
  - heisst, ein potentieller Kunde kann den Service auf dem "registry service" finden



- mit anderen Services verbunden, heisst das ein Service gleichzeitig auch ein Client sein kann



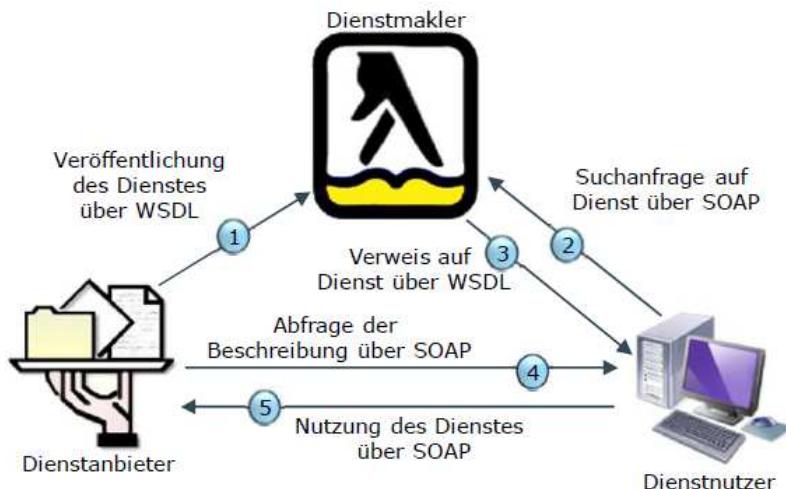
## Web-Services: Die Vision



## Netzwerkprotokolle

- WebServices sind plattform- und implementierungsunabhängige Softwarekomponenten, die
  - mit einer Service Description Language beschrieben,
  - auf einem "registry of services" registriert und durch einen standardisierten Mechanismus gefunden,
  - durch ein deklariertes API über das Netz aufgerufen,
  - mit anderen Services verbunden werden können.
- Fast alle diese Protokolle basieren auf XML (Ausnahme UDDI)
  - **SOAP**
  - **WSDL**
  - **UDDI**
  - **SOAP**
  - **BPEL**
- XML (eXtended Markup Language)  
Erweiterbares Textformat für den Austausch von strukturierten Daten.
- **SOAP** (Simple Object Access Protocol)  
Realisiert Entfernten Prozeduraufruf (RPC). Transportiert XML Nachrichten (z.B. über HTTP).
- **WSDL** (Webservice Description Language)  
XML Notation zur Beschreibung von Webservices.
- **UDDI** (Universal Description, Discovery and Integration)  
Verzeichnisdienst zur Veröffentlichung von Webservices.
- **BPEL** (Business Process Execution Language)  
Choreographierung und Orchestrierung von Web Services

## Übersicht zu Web Service Standards im Rollenmodell der SOA



### SOAP (Simple Object Access Protocol)

- Spezifisches Kommunikationsprotokoll, um XML Dokumente über das Internet zu schicken, unter Verwendung von:
  - http
  - SMTP
  - andere Protokolle
- vorgegebene Serealisierung
- Plattform unabhängig

Motivation:

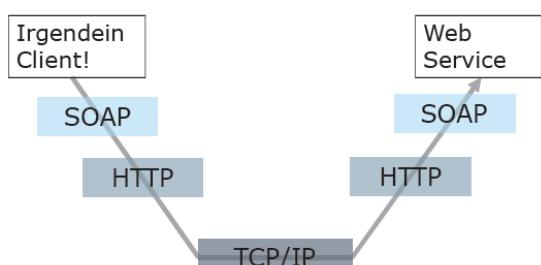
Viele verteilte Anwendungen kommunizieren über RPC (Remote Procedure Calls) über Objekte miteinander. Beispiele:

- DCOM
- CORBA

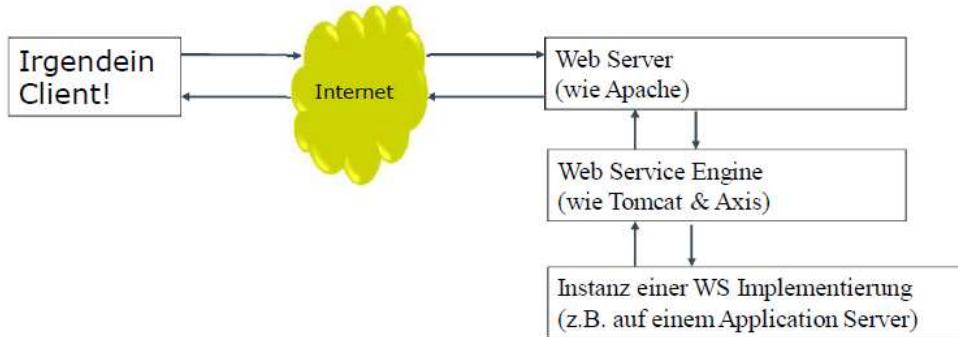
HTTP ist für solche Objekte nicht geeignet. RPC Aufrufe können nicht einfach für das Internet adoptiert werden. Es bestehen Sicherheitsprobleme für diese RPC-Aufrufe. Die meisten Firewalls und Proxy-Server blockieren solche Aufrufe. HTTP wird von allen Browsern und Servern unterstützt. SOAP bietet ein brauchbares Protokoll für RPC.

**Nachteil:** relativ schwerfällig, als alternative bietet sich **RESTful** an

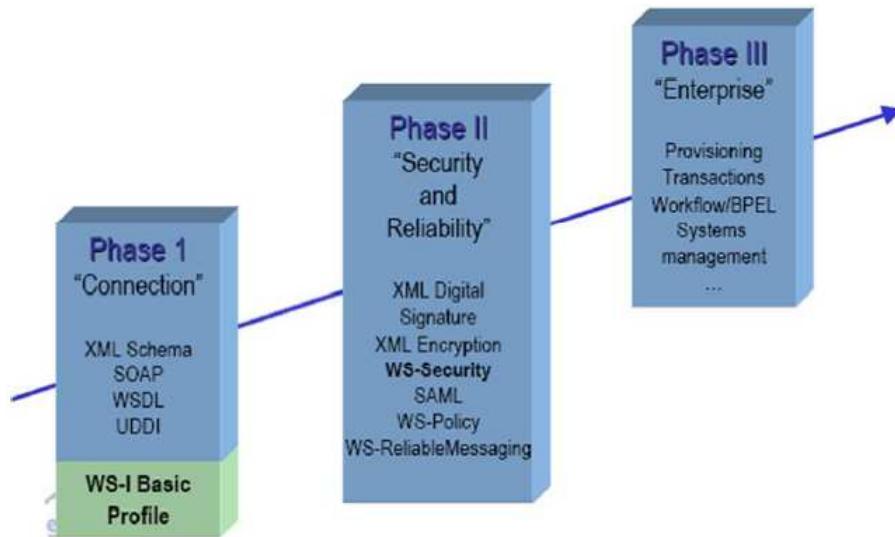
Verwendung:



## Beispiel der Verarbeitung eines Web-Services



## Entwicklung der WS Standards



## Einführung in die Unified Modeling Language

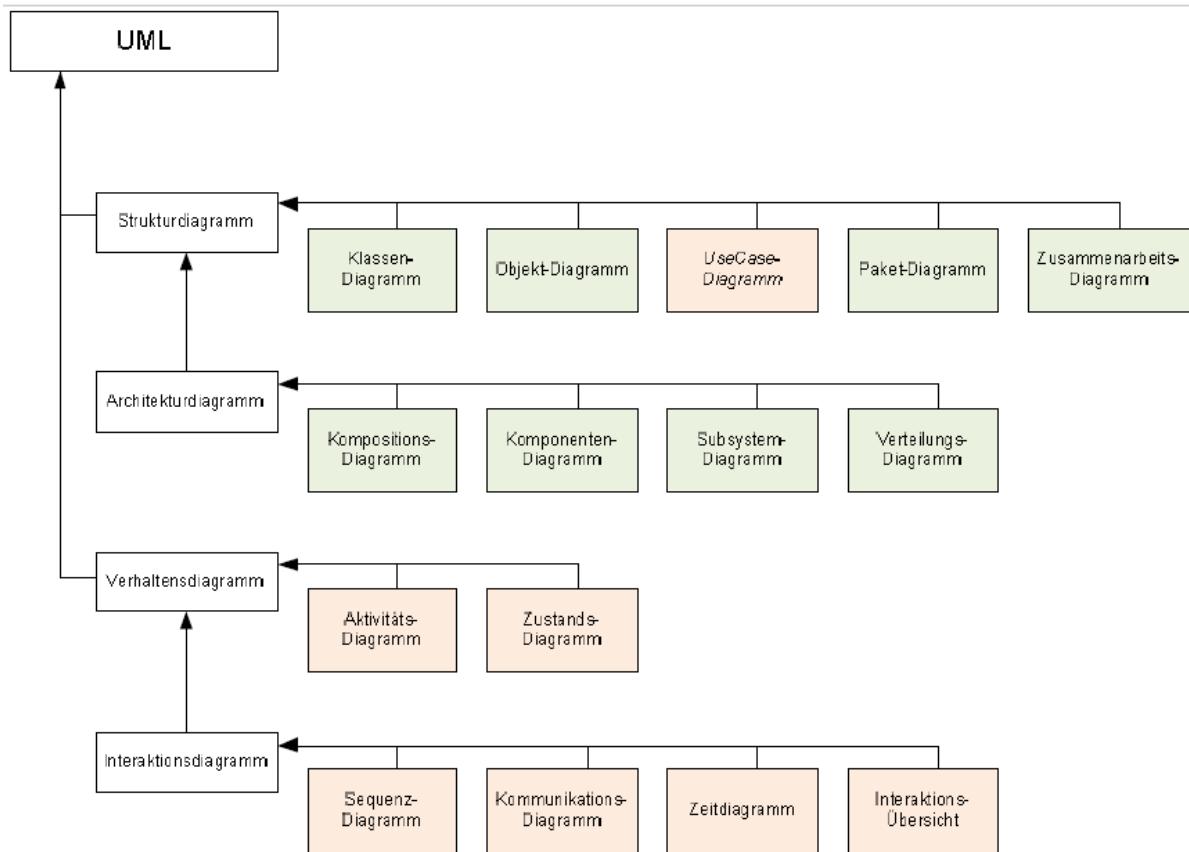
Unter UML versteht man eine einheitliche Sprache und Notation zur Modellierung, Dokumentation und Spezifizierung und Visualisierung komplexer Softwaresysteme unabhängig von deren Fach- und Realisierungsgebiet. Sie stellt Notationselemente sowohl für statische als auch für dynamische Modelle zur Verfügung und unterstützt insbesondere die objektorientierte Vorgehensweise.

UML kann auch bei der Modellierung der Datenbanken eingesetzt werden und soll eine einheitliche Sprache werden, die von möglichst vielen SW-Entwicklern benutzt wird.

### Unterstützte Sichten von UML

Statisch	Dynamisch	Implementierung
Anwendungsfalldiagramme Klassendiagramme Objektdiagramm Paketdiagramm	Sequenzdiagramme Kommunikationsdiagramme Zustandsdiagramme Aktivitätendiagramme	Komponentendiagramme Verteilungsdiagramme (Deployment-Diagramme) Subsystemdiagramme

## UML Modulübersicht



## Klassendiagramm

### Klasse

Eine **Klasse** beschreibt eine Menge von Objekten, die dieselben Merkmale, Einschränkungen (semantische Integritätsbedingungen) und dieselbe Semantik besitzen. [...]

Ziel einer Klasse ist die Klassifikation von Objekten und die Festlegung der Merkmale, die die Struktur und das Verhalten dieser Objekte festlegt.

Eine Klasse ist eine Spezialisierung eines *Classifiers* (Informationsträger). Ein Classifier in UML ist eine Klassifizierung von Instanzen, die folgende Eigenschaften teilen:

- Namensraum
- Überschreibbare Elemente
- Typ

Es handelt sich um eine abstrakte Metaklasse

## Klassendiagramm

Ein **Klassendiagramm** ist eine statisch-deklarative Darstellung der Beziehungen zwischen den Klassen des Systems.

Optional können die **Attribute** und **Operationen** jeder Klasse hinzugefügt werden, also die jeweilige Klasse selbst beschrieben werden.

Eine Klasse kann unter zwei Gesichtspunkten betrachtet werden:

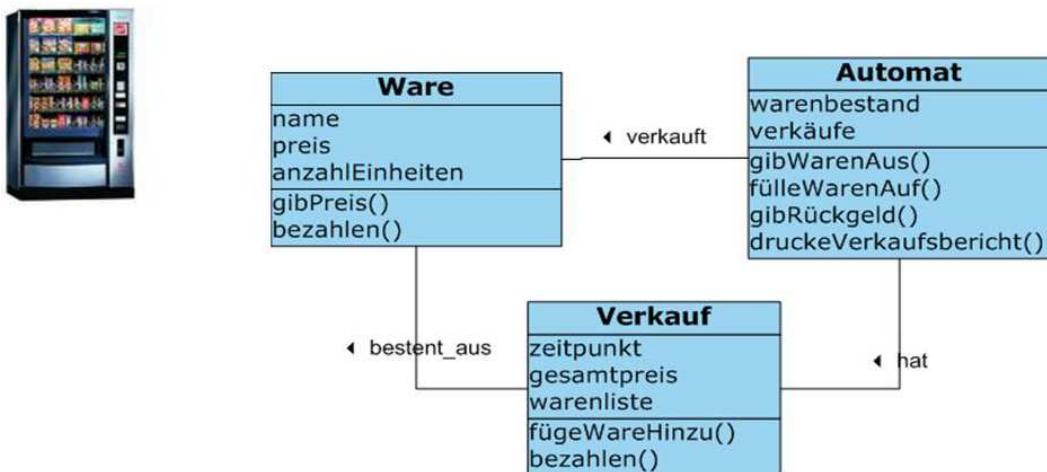
1. **Intension** (Intent, Bedeutung):

Eine Klasse beschreibt die verallgemeinerte Eigenschaften aller Objekte

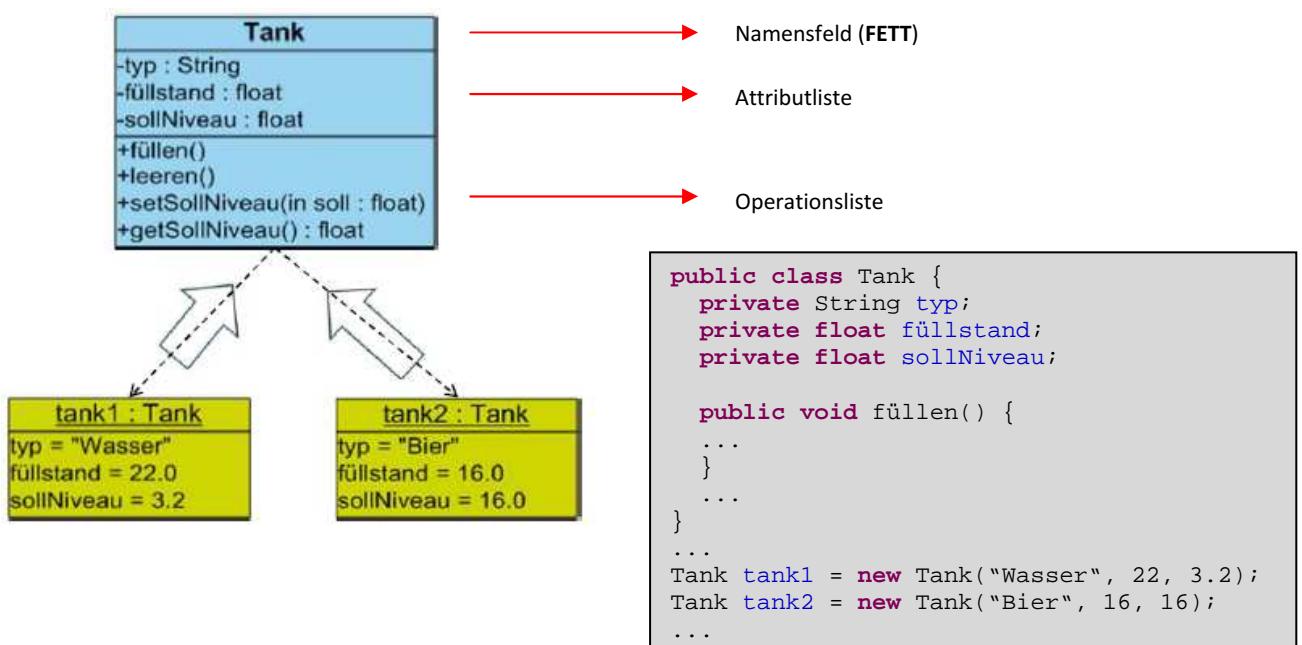
2. **Extension** (Extent, Umfang):

Eine Klasse bezeichnet auch die Menge aller zu ihr gehörenden Objekte. Somit kann eine Klasse auch Operationen haben, die auf diese Menge angewandt werden kann.

Beispiel eines Klassendiagramms (Warenautomat):



## Klassennotation in UML



Klassendiagramme beinhalten Klassen und weitere Symbole (z.B. Assoziationen, Vererbung). Es handelt sich um ein statisches Modell des Systems. Bei grossen Systemen braucht es mehrere Klassendiagramme. Erweiterungen der Klassennotation in der UML:

- **Stereotyp (stereotype)**  
Übergreifender Bezeichner, klassifiziert Elemente des Modells. Beispiel «interface», «GUI»
- **Merkmal (property)**  
Beschreibt Eigenschaften (z.B. Einschränkungen) für ein Modellelement  
Beispiel: füllstand {0 <= füllstand <= maxNiveau}

## Verantwortlichkeit der Klasse

Jede Klasse soll für genau einen Aspekt des Gesamtsystems verantwortlich sein. Die in diesem Verantwortlichkeitsbereich liegenden Eigenschaften sollen in einer einzelnen Klasse zusammengefasst sein und nicht auf verschiedene Klassen aufgeteilt werden. Eine Klasse soll keine Eigenschaften enthalten, die nicht zu diesem Verantwortlichkeitsbereich gehören!

Kunde	Anschrift	Bankverbindung
<ul style="list-style-type: none"> <li>• Verwaltet alle personenbezogene Daten eines Kunden</li> <li>• Verwaltet Anschrift</li> <li>• Telekommunikationsverbindungen und Bankverbindungen</li> </ul>	<ul style="list-style-type: none"> <li>• Verwaltet und repräsentiert eine postalische Anschrift</li> <li>• Prüft soweit möglich und sinnvoll die Anschrift gegen vorhandene PLZ- und Strassenverzeichnisse</li> </ul>	<ul style="list-style-type: none"> <li>• Verwaltet und repräsentiert ein Konto bei einem Geldinstitut</li> <li>• Prüft den IBAN gegen einen vorhandenes IBAN-Dienst</li> </ul>

## Sichtbarkeit in UML (und Java)

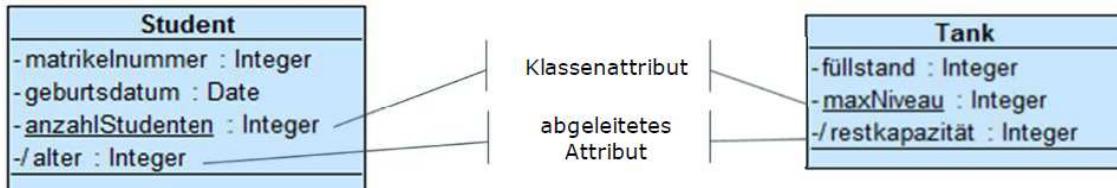
Name	UML	Java
public	+	public
private	-	private
protected	#	protected
package	~	/* default */

## Klassenattribut und abgeleitetes Attribut

- **Klassenattribut**
  - Gehört nicht zu einem einzelnen Objekt, sondern ist Attribut einer Klasse
  - Es existiert nur ein Attributwert für alle Objekte einer Klasse
  - Änderung des Attributwertes gilt für alle Objekte dieser Klasse
  - Klassenattribute existieren auch, wenn es zu einer Klasse (noch) keine Objekte gibt
  - Verwendung: z.B. Zähler für Objektverwaltung
  - Kennzeichnung durch Unterstrichen Beispiel: anzahlPakete
  - In Java wird ein solches Attribut mit dem Modifikator **static** ausgezeichnet.  
Beispiel : **static** int anzahlPakete;

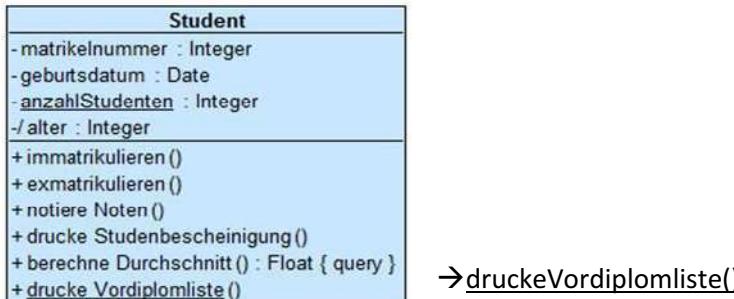
- Abgeleitetes Attribut

- Der Wert wird automatisch aus anderen Attributwerten berechnet
- darf nicht direkt geändert werden
- Kennzeichnung mit dem Präfix "/"
- Angabe eines Anfangswertes entfällt
- Berechnungsvorschrift kann in Form einer Restriktion angegeben werden



## Klassenoperationen

Klassenoperationen sind der Klasse zugeordnet und nicht dem einzelnen Objekt!



```

public class Student {
    ...
    void notiereNoten() {
    ...
    }
    float berechneDurchschnitt() {
    ...
    }
    static void druckeVordiplomliste()
    {
    ...
    }
}
  
```

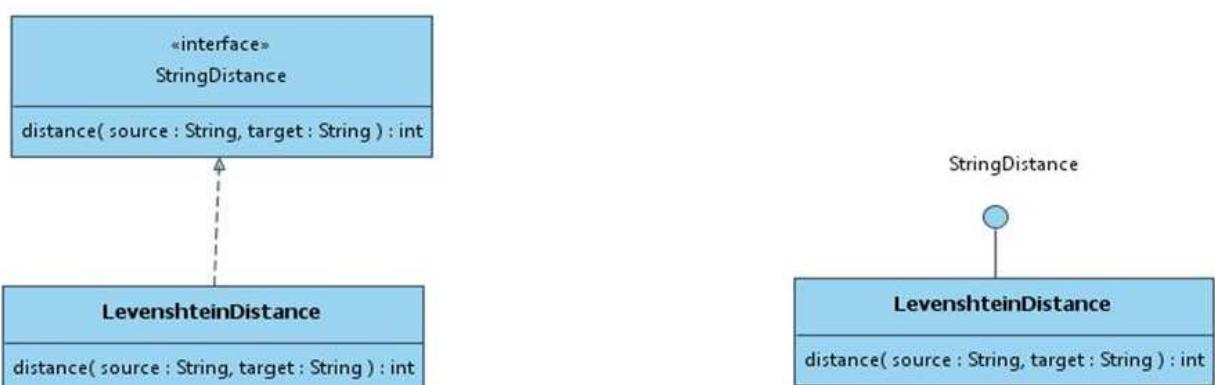
- für Aufgaben, die unabhängig von einem ausgewählten Objekt sind (nicht auf Objektattribute zugreifen)
- für Operationen, die sich auf alle Objekte einer Klasse beziehen

## Interface

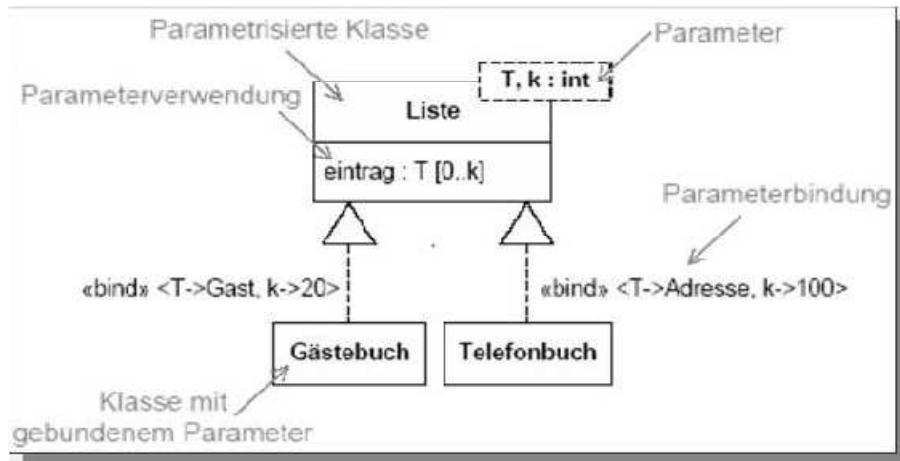
„Interface“ ist eine besondere Form einer Klasse, die die gewünschte Funktionalität *beschreibt*

- Trennung von Beschreibung (Spezifikation) und Implementierung
- hat ausschliesslich abstrakte **Methoden** und **Konstanten**
  - Methoden sind öffentlich (public)
  - Definition von Konstruktoren nicht erlaubt

Beispiel:



## Parametrisierte Klasse

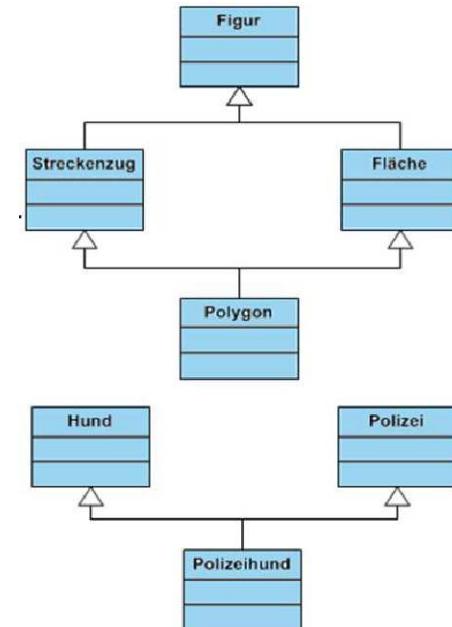


Eine parametrisierte Klasse ist eine Vorlage (Template), die eine Klasse mit einem oder mehreren ungebundenen, formalen Parametern beschreibt. Sie definiert eine Familie von Klassen, von denen jede durch Binden der Parameter an tatsächliche Werte angegeben wird. Die formalen Parameter in gestrichelten Rechteck rechts oben besitzen die Syntax Name : Typ, wobei Name einen Bezeichner und Typ eine Zeichenfolge darstellt.

## Generalisierung

Die Darstellung der Generalisierung (oder in Gegenrichtung Spezialisierung) wird mit einem geschlossenen, leeren Pfeil modelliert. In der UML können alle Classifier angehängt werden.

Von der Logik der Vererbung muss jeder Vorgänger mindestens zwei Nachfolger haben, bei denen zusätzliche Attribute hinzutreten. In seltenen Fällen unterscheiden sie sich nur in den (überschriebenen) Operationen. Dass in den Diagrammen der UML dies nicht immer der Fall ist, bedeutet nur, dass der Vorgänger in mindestens einem weiteren Diagramm als Elternelement auftritt.



## Assoziation

Eine **Assoziation** modelliert Verbindungen zwischen Objekten einer oder mehreren Klassen

Assoziationen modelliert Verbindung zwischen Objekten, nicht zwischen Klassen! Beispiel:

Objekte der Klasse Student (z.B. Paul, Susi, Peter, ...) haben eine Verbindung zu Objekten der Klasse Lehrveranstaltung (z.B. Einführung in die Programmierung).



Weiteres Beispiel (Assoziation zwischen Tank und Ventil):

Klassendiagramm:



Zeigt Klassen und ihre Beziehungen (statische Modellelemente)

Objektdiagramm:



Zeigt Objekte und ihre konkreten Beziehungen zu einem Zeitpunkt

Die Menge aller Verbindungen wird als Assoziation zwischen den Objekten der Klasse Tank und Ventil bezeichnet!

### Eigenschaften von Assoziationen

- Es gibt binäre (zwischen zwei Objekten) und höherwertige Assoziationen
- Eine reflexive Assoziation besteht zwischen Objekten derselben Klasse
- Eine Assoziation hat eine Richtung (Navigierbarkeit)
  - "Welches Objekt ist über die Beziehung informiert?"
  - Unidirektional
  - Bidirektional
- Assoziationen sind in der Systemanalyse bidirektional, Objekte "kennen" sich gegenseitig erst im Design wird entschieden, ob sie unidirektional sein sollen.
- 3 Arten von Assoziation
  - einfache Assoziation
  - Aggregation
  - Komposition

### UML Notation einer Assoziation

- Linie zwischen einer oder zwei Klassen
- Assoziationsname
- An jedem Ende der Linie steht die Wertigkeit bzw. Kardinalität (multiplicity)
- An jedem Ende kann ein Rollename stehen



### Name der Assoziation

- Beschreibt Semantik (Bedeutung) der Assoziation
- Beschreibt meistens nur eine Richtung der Assoziation
- Ein schwarzes Dreieck kann die Leserichtung angeben
- Name darf fehlen, wenn die Bedeutung der Assoziation offensichtlich ist

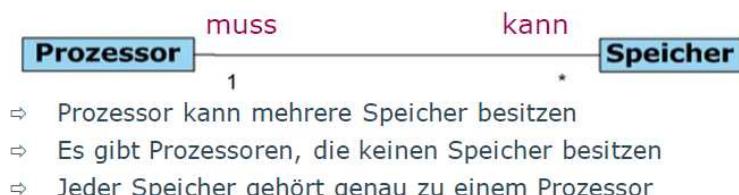


### UML Notation der Kardinalität

1	genau 1
0 .. 1	0 bis 1
*	0 bis viele
3 .. *	3 bis viele
0..2	0 bis 2
2	genau 2
2, 4, 6	2, 4 oder 6
1..5, 8, 10..*	nicht 6, 7 oder 9

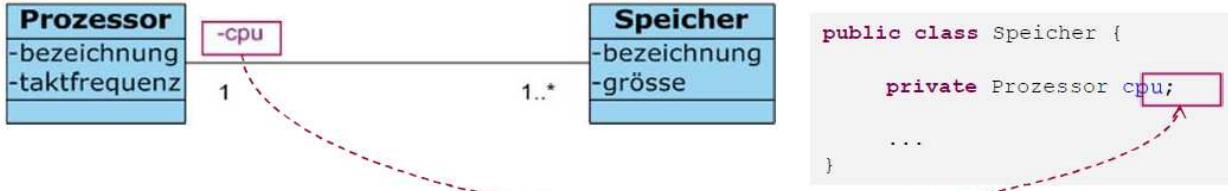
### Bedeutung der Kardinalität

- Kann-Assoziation  
Untergrenze: Kardinalität 0
- Muss-Assoziation  
Untergrenze: Kardinalität 1 oder grösser



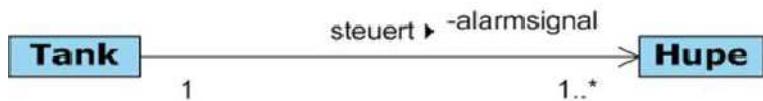
### Rollenname

Der Rollenname beschreibt die Bedeutung eines Objekts in einer Assoziation. Binäre Assoziationen besitzen maximal zwei Rollen. Der Rollenname wird an das Ende der Assoziation bei der Klasse geschrieben, deren Bedeutung in der Assoziation die Rolle beschreibt.

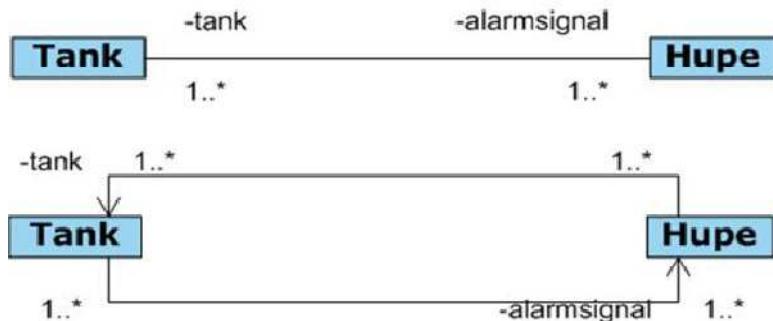


### Gerichtete Assoziation

- Nur ein Objekt ist über Beziehung informiert (unidirektionale Navigierbarkeit)
- Nur in die Navigationsrichtung können Methoden aufgerufen werden
- Darstellung durch geöffnete Pfeilspitze



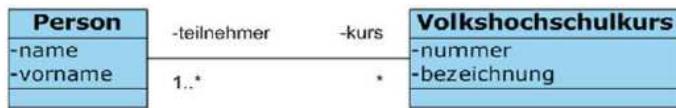
- Jede bidirektionale Assoziation kann durch zwei unidirektionale Assoziationen ausgedrückt werden



- In der Analyse wird die Navigierbarkeit normalerweise noch nicht festgelegt.

## Aufgabe: Assoziationsbeschreibung

- Beschreiben Sie mit eigenen Worten, welche Art von Beziehung zwischen den zwei Klassen in dem nachfolgend abgebildeten Modell besteht:



- Java (gekürzt):

```

public class Person {
    private String name;
    private String vorname;

    private List<Volkshochschulkurs> kurs;
    ...
}

public class Volkshochschulkurs {
    private int nummer;
    private String bezeichnung;

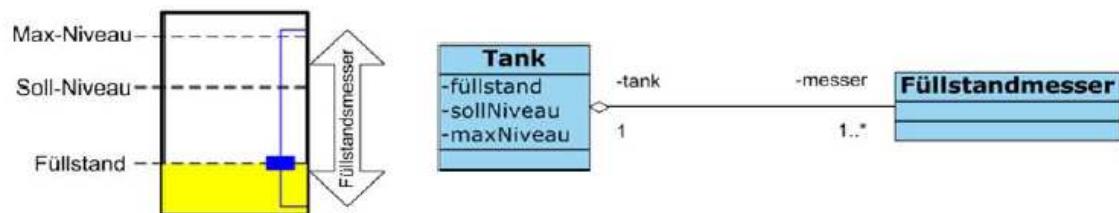
    private List<Person> teilnehmer;
    ...
}
  
```

## Aggregation

Eine **Aggregation** bezeichnet eine "Teil-Ganzes"- (whole-part) oder "ist-Teil-von"-Beziehung zwischen Objekten

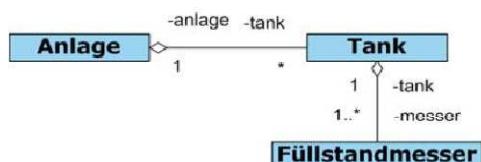


- Aggregation ist eine spezielle Form der Assoziation
- Lässt sich durch ist Teil von bzw. besteht aus beschreiben (Ganze –Teil)
- leere Raute kennzeichnet das Ganze

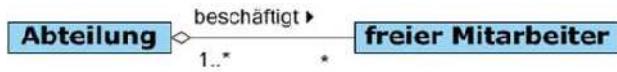


## Eigenschaften:

- wenn B Teil von A ist, dann darf A nicht Teil von B sein (ist asymmetrisch)
- wenn A Teil von B und B teil von C ist, dann ist auch A Teil von C (ist transitiv)



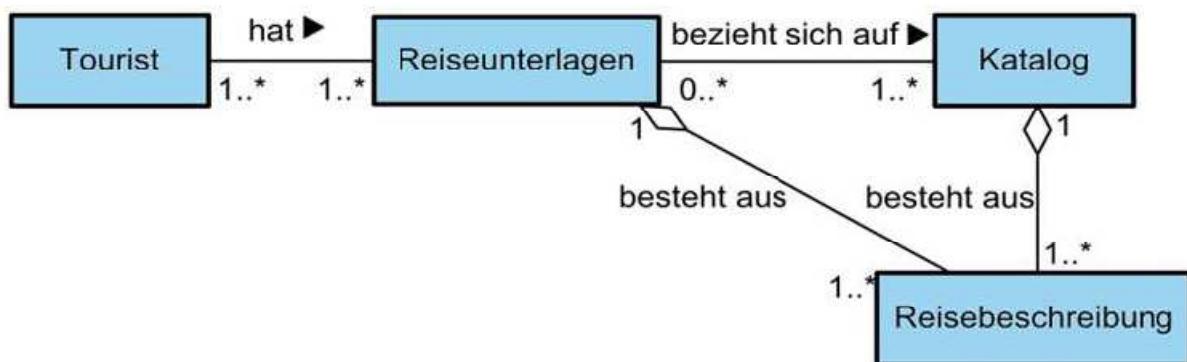
- muss nicht exklusiv sein
- B darf gleichzeitig Teil von A und Teil von C sein



- Das Ganze übernimmt Aufgaben stellvertretend für seine Teile

#### Beispiel Reiseunternehmen:

Ein Reiseunternehmen erarbeitet ein Klassenmodell für die Verwaltung seiner Reisekataloge, seiner Reisebeschreibungen und der Reiseunterlagen. Selbstverständlich beschreibt eine Klasse auch den Touristen, der die Reisen bucht. Ein Reisekatalog enthält eine oder mehrere Reisebeschreibungen. Die Reiseunterlagen für den Touristen werden aus einzelnen Reisebeschreibungen, eventuell aus verschiedenen Katalogen, zusammengestellt. Mehrere Touristen können auch gemeinsame Reiseunterlagen erhalten.



#### Wann liegt eine Aggregation vor?

Eine Aggregation liegt vor, wenn die folgenden Fragen mit ja beantwortet werden können:

- Ist die Beschreibung "Teil von" zutreffend?
- Werden manche Operationen auf das "Ganze" automatisch auch auf die "Teile" angewandt?
- Pflanzen sich manche Attribute vom "Ganzen" auf alle oder einige "Teile" fort?
- Ist die Verbindung durch eine Asymmetrie gekennzeichnet, bei der die "Teile" dem "Ganzen" untergeordnet sind?

#### Beispiele für Aggregationsbeziehungen:

- Das Ganze und seine Teile  
Bsp.: Firma (Kollektion) und Angestellte (Mitglieder)
- Der Behälter und sein Inhalt  
Bsp.: Pkw (Ganzes) und Motor (Teil)
- Die Kollektion und ihre Mitglieder  
Bsp.: Kaffeemaschine (Behälter) und Kaffeepulver (Inhalt)

## Komposition

Eine Aggregation mit starker Bindung (Existenzabhängigkeit) wird als **Komposition** bezeichnet.



- ohne Kompositionsklasse ist die Teilklassse nicht existenzfähig
- die gefüllte Raute kennzeichnet das Ganze



### Eigenschaften:

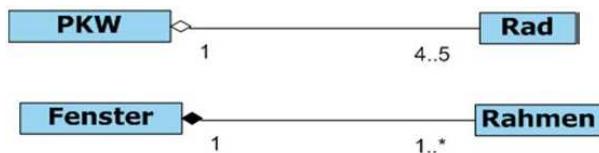
Zusätzlich zur Aggregation gilt:

- Jedes Objekt der Teilklassse kann zu einem bestimmten Zeitpunkt nur Komponente eines einzigen Objekts der Aggregatklassse sein
- Kardinalität der Aggregatklassse  $\leq 1$
- Ein Teil darf evtl. auch anderem Ganzen zugeordnet werden (aber nicht gleichzeitig)
- Dynamische Semantik des Ganzen gilt auch für seine Teile (propagation semantics)

Wird das Ganze kopiert, werden auch seine Teile kopiert



## Aggregation VS Komposition



- Aggregation: "PKW hat Räder":
  - Räder gehören notwendigerweise zu einem Auto (Aggregation)
  - Räder können aber eigenständig und zwischen PKWs austauschbar betrachtet werden (keine Komposition)

Komposition: "Fenster hat Rahmen":

- Wird das Fenster gelöscht, werden auch alle existenzabhängigen Einzelteile mitgelöscht

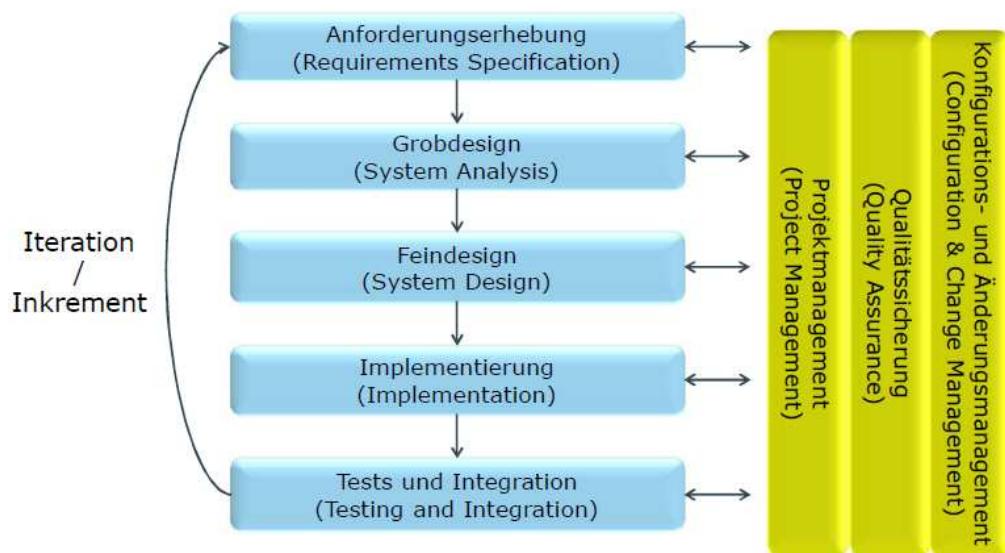
Komposition

## Analyse

### Phasen der Softwareentwicklung



### Die Disziplinen der Softwareentwicklung



## UML – Use Case Diagram

Anwendungsfälle (use cases) beschreiben, was ein System leisten soll. Da diese Leistung immer mit den Nutzern des Systems zu tun hat, erläutern sie damit das Zusammenwirken von Anwendern mit dem System. Die Aufgabe der Anwendungsfälle ist, die durch ein System benötigte Funktionalität zu erfassen. Es besteht in der Grundstruktur aus:

- **Anwendungsfällen** (die Funktionalität/Verhalten benennen)
- Strichmenschen/**Akteure** (die die Nutzer des gesamten Anwendungsfalls darstellen)
- Und verbindenden Linien (**Beziehungen**)

### Anwendungsfälle

Jeder Anwendungsfall repräsentiert ein Verhalten das ein System leisten kann in Zusammenarbeit mit einem oder mehreren Akteuren.

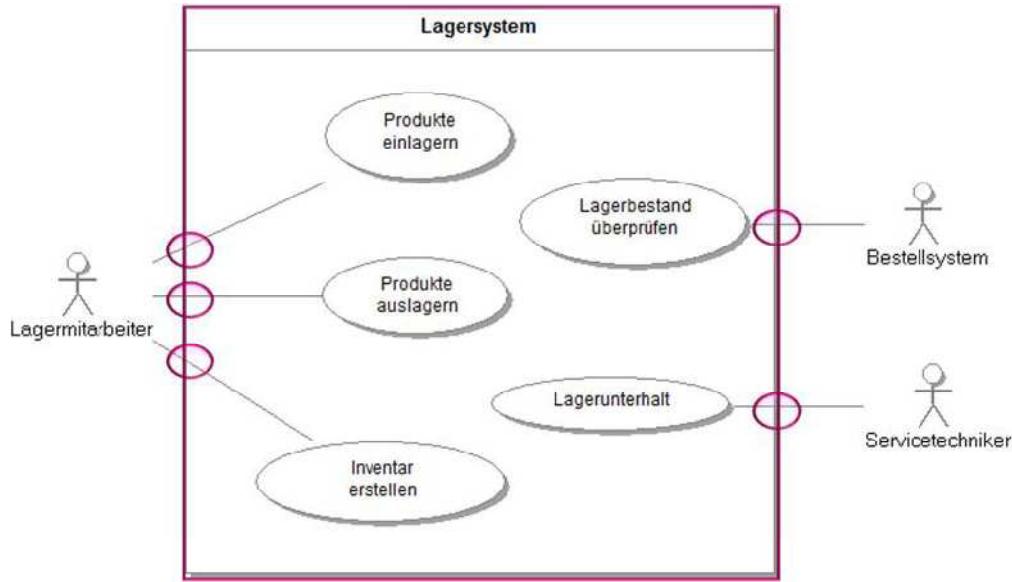
„Ein Anwendungsfall beschreibt einen typischen Arbeitsablauf.“ [ebenda]

Anwendungsfälle identifizieren und beschreiben Abläufe bzw. Aktivitäten, die isoliert betrachtet werden können und werden von einem Akteur angestossen.

## Akteure

Als Akteure werden die Nutzer eines Systems bezeichnet. Dies sind menschliche Nutzer oder auch andere Systeme. Wichtig ist, dass beide extern, d.h. ausserhalb des (Teil-) Systems, angesiedelt sind. Die Interaktion eines Akteurs mit dem System kann z.B. darin bestehen, Signale oder Daten auszutauschen. Dies können die Eingaben eines Menschen am Geldautomaten sein.

### Beispiel:



### Beschreibung

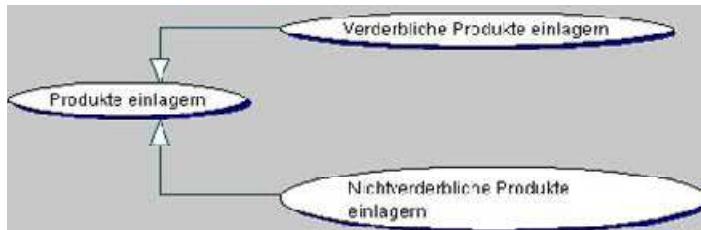
Ein Anwendungsfall muss möglichst genau beschrieben werden, um die Missverständnisse von vornherein zu vermeiden und dem Entwickler alle für die Implementierung benötigten Informationen zur Verfügung zu stellen.

ID	L1UC6			
Name	Geld am Automat auszahlen			
Beschreibung	Einem Verfügungsberchtigten wird durch einen Geldautomaten ein vom Verfügungsberchtigten geforderter Geldbetrag ausgezahlt und vom zugehörigen Konto abgebucht.			
Akteur	Verfügungsberchtigter, Kontoführungssystem			
Vorbedingung	Der Geldautomat ist bereit, eine Karte aufzunehmen.			
Auslösendes Ereignis	Der Verfügungsberchtigte steckt eine Karte in den Geldautomat			
essentieller Ablauf (happy path)	<ul style="list-style-type: none"> <li>- Verfügungsberchtigten identifizieren</li> <li>- Angeforderten Geldbetrag bestimmen</li> <li>- Auszahlungsmöglichkeit prüfen</li> <li>- Auszahlung auf Konto buchen</li> <li>- Geldbetrag übertragen</li> <li>- Karte auswerfen</li> </ul>			
Alternativer Ablauf	<ul style="list-style-type: none"> <li>- ...</li> </ul>			
Nachbedingung	Der Geldautomat ist bereit, eine Karte aufzunehmen.			
Offene Punkte	Keine			
Änderungshistorie	wann	wer	neuer Status	was
	01.01.2010	M. Muster	in Arbeit	Erstellung des AF

## Beziehungen

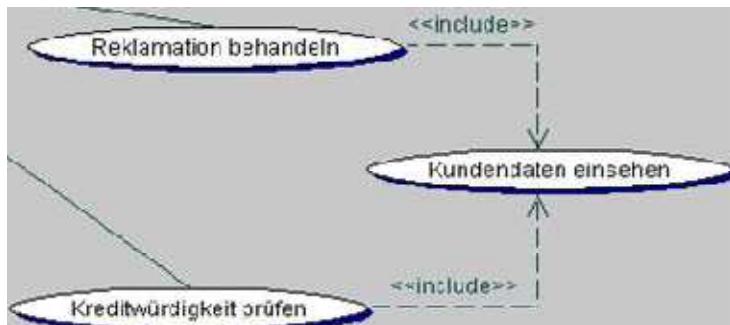
### 1. Generalisierung

wird verwendet, wenn ein allgemeingültiges Verhalten in einem Basis-Anwendungsfall abgebildet werden kann und in spezialisierten Anwendungsfällen vervollständigt bzw. überschrieben werden kann.



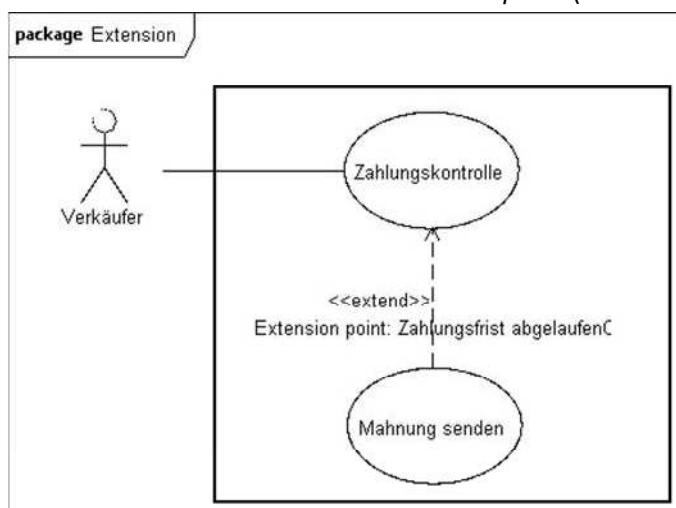
### 2. Die *include*-Beziehung

wird verwendet, wenn ein Verhaltensanteil in verschiedenen Anwendungsfällen gleich ist. Dann wird dieser aus allen, in denen er vorkommt, ausgelagert in einen eigenen Anwendungsfall. Durch die Verwendung der *include*-Beziehung kann das mehrfache Implementieren eines Verhaltens verhindert werden. Ein importiertes Use Case (*include*) wird **immer** ausgeführt.

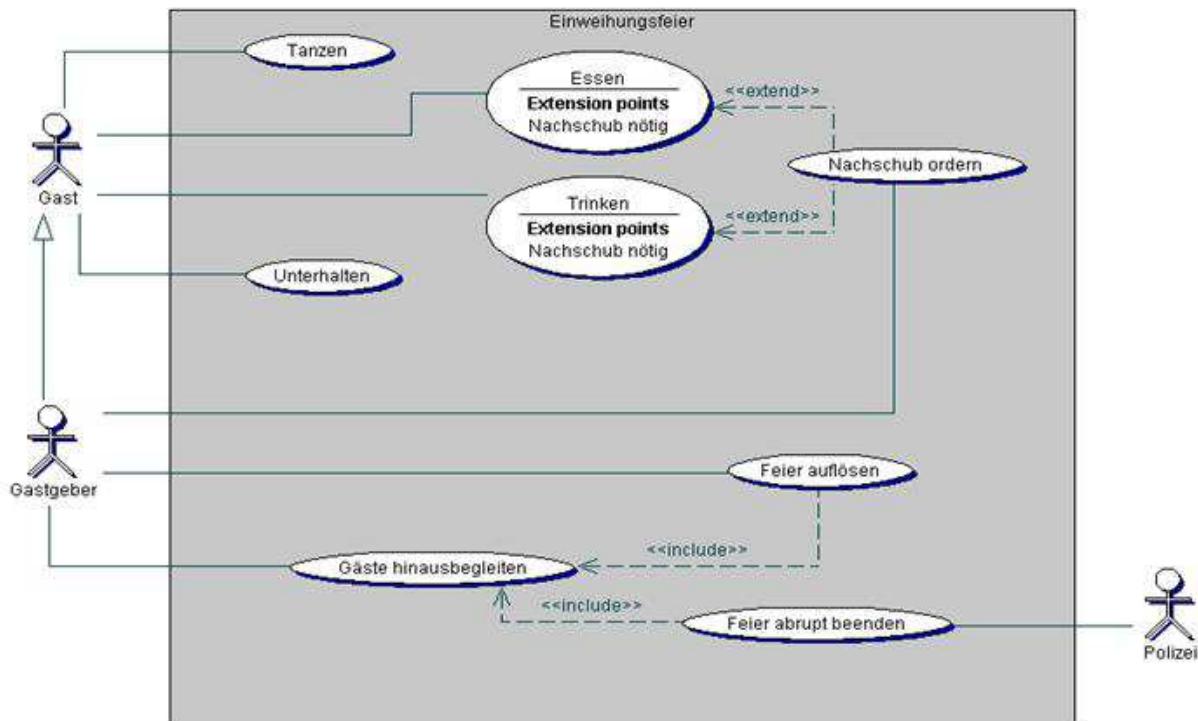


### 3. Die *extends*-Beziehung

liegt dann vor, wenn das Verhalten eines Use Cases durch einen anderen Use Case erweitert werden **kann**, aber **nicht muss**. Den Zeitpunkt, an dem ein Verhalten eines Use Case erweitert werden kann wird als *extension point* (Erweiterungspunkt) bezeichnet.



Beispiel:



- Man soll
  - Use Cases aus Sicht des Benutzers modellieren
  - Use Cases gut beschreiben
  - Use Cases durch eigene Notation sinnvoll ergänzen
  - Benennung von Use Cases und Akteure einheitlich machen
- Man soll nicht
  - Use Cases für Beschreibung von Systemfunktionen verwenden
  - Use Cases aus Sicht des Entwicklers erstellen
  - Zu viele Use Cases modellieren und zu viele Beziehungen zwischen Use Cases angeben

## UML MODELLE FÜR DIE ANALYSE DISZIPLIN

### Ziele der Analysedisziplin

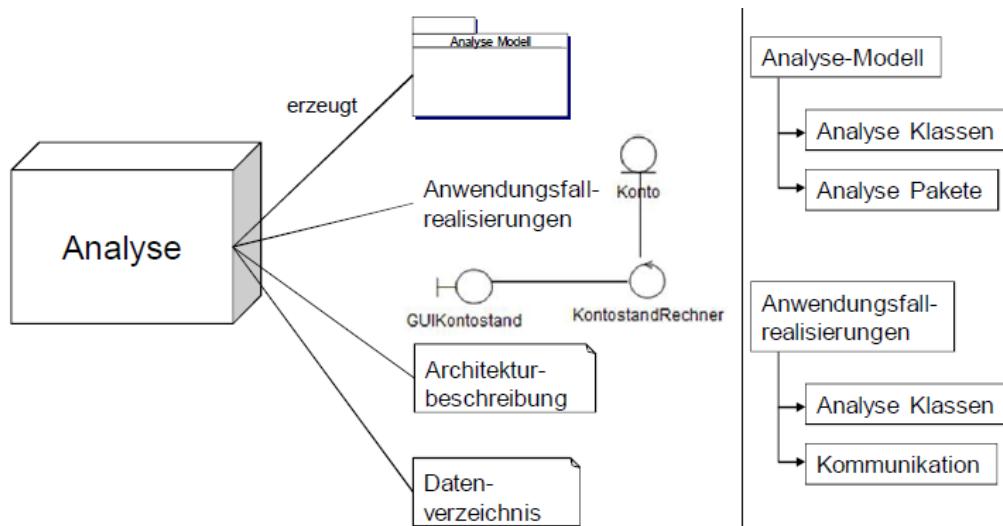
- Analyse der Anforderungen und Strukturierung mit systeminternen Elementen.
- Transformation der externen Sicht der Anwendungsfälle zu einer internen Sicht, die zeigt, **was** das System tun muss, um diese zu realisieren.
- Zeigt **nicht wie** das System umgesetzt wird.
- Überführung der Anforderungen in die Sprache der Entwickler
- Identifizierung der Schlüsselemente, -konzepte, -entitäten.
- Identifizierung der wichtigen Entitäten, die das Grundgerüst der Architektur bilden. Können später im Design Klassen, Sammlungen von Klassen oder gar Subsysteme werden.

- Das Analysemodell enthält häufig **Analyseklassen**, **Pakete**, **Kommunikationsdiagramm**, **Zustandsdiagramme** oder **Ablaufdiagramme**.
- Die Analyse entwickelt darüber hinaus **Anwendungsfallrealisierungen** (Use Case Realizations), die zeigen, wie die Anwendungsfälle im Analysemodell abgebildet sind.

## Vergleich zwischen Anforderungen und Grobentwurf

Anforderungen (Requirement Discipline)	Analyse-Disziplin
• Sprache des Kunden	• Sprache der Entwickler
• Externe Sicht auf die Funktionalität	• Interne Sicht des Systems auf die Funktionalität
• Strukturiert durch Anwendungsfälle	• Strukturiert durch stereotype Klassen und Pakete
• Dient zum verstehen, was das System tun soll	• Dient zum verstehen, was das System tun soll, um die Benutzeranforderungen zu unterstützen
• Kann Redundanzen und Inkonsistenzen enthalten	• Sollte keine Redundanzen und Inkonsistenzen enthalten
• Hält Funktionalitäten des Systems fest	• Hält die Kernkonzepte (Entitäten), welche für das System wichtig sind, fest
• Definiert Anwendungsfälle	• Definiert Anwendungsfallrealisierungen

## Bestandteile des Analysemodells



## Tätigkeiten der Analysedisziplin

- Analyse der Architektur
- Erzeugen der Analyse-Klassen
- Erzeugen der Analyse-Pakete
- Analyse der Anwendungsfälle und Erzeugen der Anwendungsfallrealisierungen
- Modellieren der Kommunikation der Analyseklassen und deren Zustände

Dieser Prozess ist nicht sequentiell, sondern iterativ!

## Das Analysemodell

- Kernelement der Analysedisziplin
- Wichtiger Bestandteil: Analyseklassen-Diagramme
  - Zeigen die statische Struktur mittels der wichtigsten Analyseklassen
  - Zeigen Zusammenhänge zwischen diesen Klassen.
- Informationen stammen von
  - Problembeschreibung
  - Domänenexperten
  - Allgemeinwissen
  - und im Speziellen von den Anwendungsfällen
- Reduziert die Menge der Klassen meist auf 3 Stereotypen:
  - Boundary
  - Control
  - Entity
- Ist eine Abstraktion des Systems und vermeidet Probleme zu adressieren, die im Feinentwurf gelöst werden.
- Ist einfacher zu verstehen als das Feindesign oder die komplette Systemarchitektur.

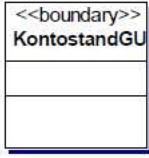
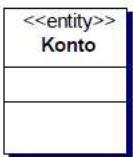
### Weshalb ein Analysemodell?

→ Identifiziert **Entitäten**, die für das System wichtig sind. Wahrscheinlich der wichtigste Aspekt der Analysetätigkeit.

## Klassen des Analysemodells

Analyseklassen sind Abstraktionen von ein oder mehreren Klassen oder Subsystemen des Feinentwurfs. Sie unterscheiden sich von Klassen im Feinentwurf:

- Sie haben Verantwortlichkeiten, keine Operationen. Diese werden textlich beschrieben.
- Sie haben grobkörnige Attribute der abgebildeten Domäne.
- Sie können komplexe Konzepte abbilden.
- Sie haben Beziehungen. Diese sind konzeptionell, nicht implementierungsabhängig.
- Sie sollten direkt in ein oder mehrere Anwendungsfallrealisierungen verwendet werden.
- Sie werden durch die Stereotypen **Boundary**, **Control** und **Entity** dargestellt.

<b>Boundary</b>	 <pre>&lt;&lt;boundary&gt;&gt; KontostandGU</pre>	 GUIKontostand	Interaktionen zwischen System und Akteuren, sollte deshalb mit mindestens einem Akteur verbunden sein.
<b>Control</b>	 <pre>&lt;&lt;control&gt;&gt; KontostandRechner</pre>	 KontostandRechner	Koordination oder Prozess, werden häufig verwendet um die Operationen des Anwendungsfalls zu beschreiben
<b>Entity</b>	 <pre>&lt;&lt;entity&gt;&gt; Konto</pre>	 Konto	Langlebige, persistente Daten (z.B. Bankkonto)

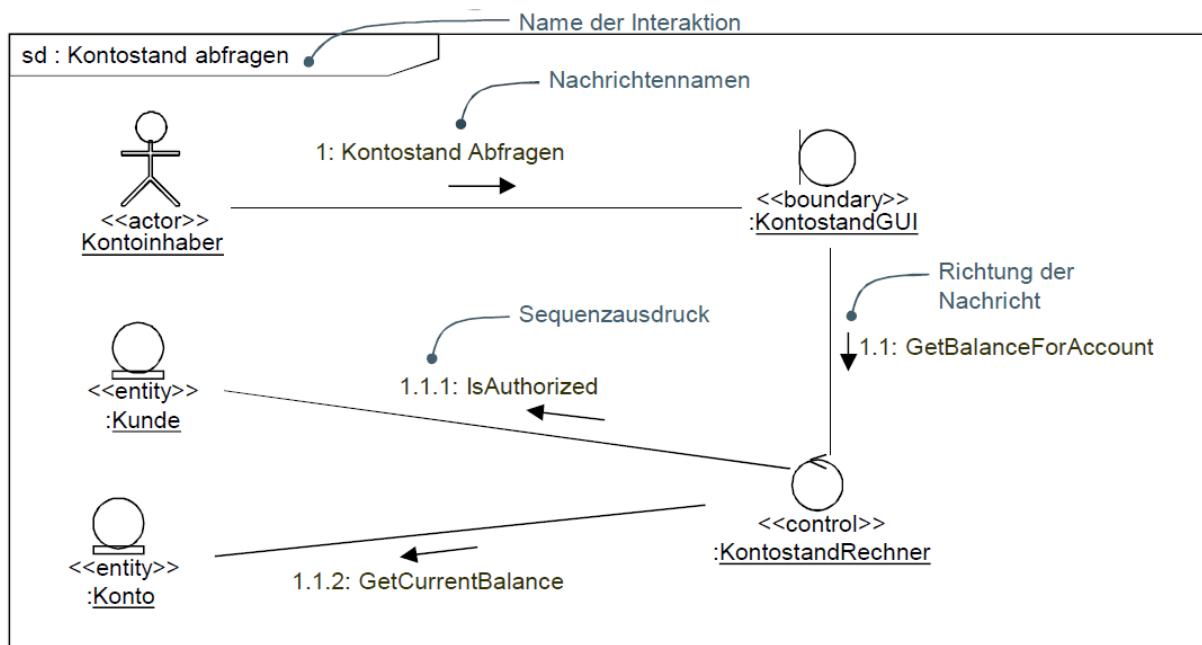
## Anwendungsfall-Realisierungen

Anwendungsfallrealisierungen verbinden die Anwendungsfälle der Anforderungsanalyse mit den Analyseklassen des Grobentwurfs. Sie zeigen die Kommunikation innerhalb der Analyseklassen im Analysemodell. Sie zeigen das Verhalten des Systems, nicht aber die aktuelle Implementierung. Elemente der Anwendungsfall-Realisierung sind:

- Klassen-Diagramme, die die Beziehungen zwischen den Klassen verdeutlichen
- Kommunikationsdiagramme
- Textliche Beschreibungen der Kommunikationsdiagramme

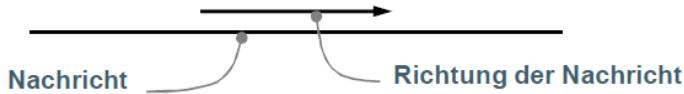
## Kommunikationsdiagramme

- Ist ein Interaktionsdiagramm (weitere Interaktionsdiagramme sind Sequenzdiagramm, Zeitdiagramm, Interaktionsübersichtdiagramm)
- Beantwortet die Frage: "Welche Teile einer komplexen Struktur arbeiten wie zusammen, um eine bestimmte Funktion zu erfüllen?"
- Verbindungen zwischen Objekten und deren Nachrichtenaustausch (Operation) in einer ganz bestimmten Situation werden dargestellt
- Der Nachrichtenaustausch wird als Pfeil bei der Objektverbindung dargestellt.
- Reihenfolge der Operationen wird durch die Nummerierung ausgedrückt.
- Objekte, die
  - neu erzeugt werden, sind mit {new}
  - während der Ausführung gelöscht werden, mit {destroyed}
  - sowohl erzeugt als auch gelöscht werden, mit {transient}
 gekennzeichnet. Dies ist optional.
- Analoge Bezeichnungen werden für Verbindungen verwendet

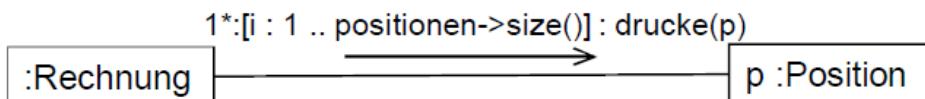


## Nachricht

Sequenzbezeichner : Attribut = NameDerNachricht (Parameter): Rückgabewert

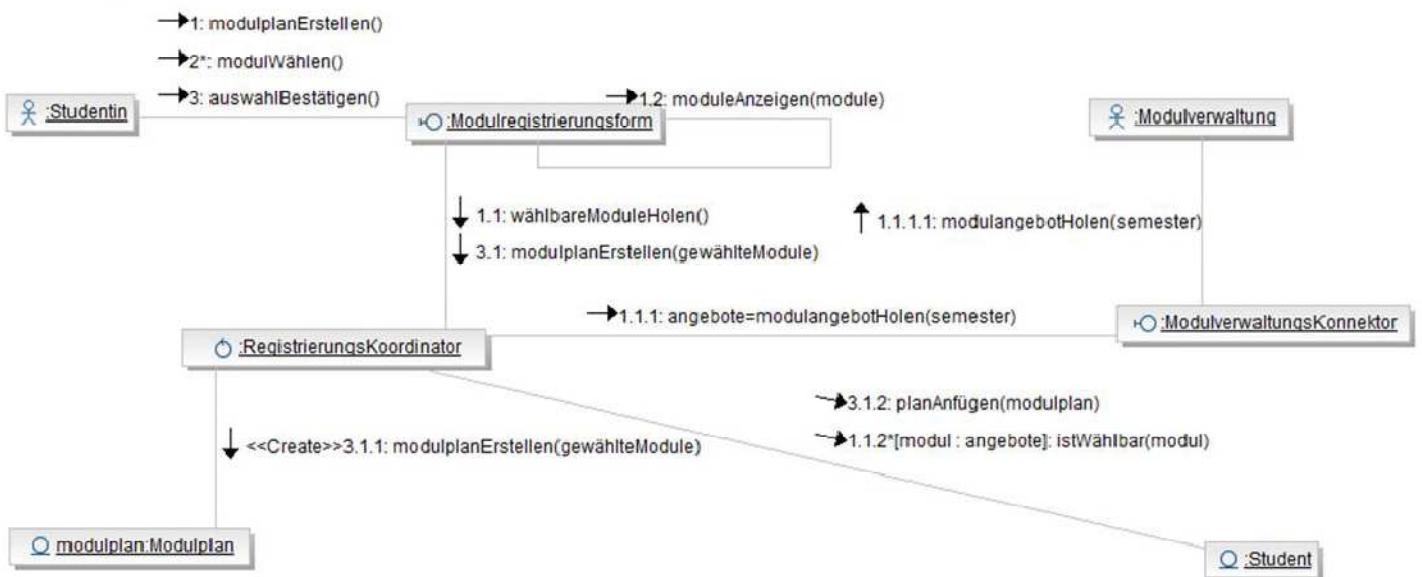


- Ein offener Pfeil steht für synchrone, ein geschlossener für asynchrone Nachrichten
- Mit dem Sequenzbezeichner können Hinweise auf den Kontrollfluss gegeben werden:
  - sequenzielle und geschachtelte Nachrichten : (geschachtelte) Nummerierung der Nachricht (z.B. 1, 2, 2.1, 2.2 etc)
  - nebenläufige Nachrichten : Kennzeichnung durch Kleinbuchstaben (z.B. 1.1a, 1.1b)
  - bedingte Nachrichten : Bedingung in eckigen Klammern (z.B. 1.1.2[isAuthorized] : GetCurrentBalance )
  - iterative Nachrichten : durch Kennzeichnung der Nummerierung mit einem Asterix (\*). Parallelle Nachrichten können durch einen Doppelstrich gekennzeichnet werden (\*||). Optional kann die Schleifenbedingung in eckigen Klammern stehen.



## Beispiel eines Kollaborationsdiagramm

### sd Moduleinschreibung für Studentin erstellen



## Kommunikationsdiagramm: Do & Don't

- Modellieren Sie Kommunikationsdiagramme aus der Vogelperspektive mit Blick auf die Kommunikationspartner und nicht aus Sicht der Kommunikationspartner

- Vermeiden Sie, Ablaufvarianten durch Bedingungen zu unterscheiden. Zeichnen Sie dafür ein eigenständiges Kommunikationsdiagramm
- Versuchen Sie, Ihr Kommunikationsdiagramm so zu gestalten, dass die Reihenfolge unwichtig wird
- Gehen Sie mit nebenläufigen und iterativen Nachrichten sparsam um. Zeichnen Sie dafür ein Sequenzdiagramm
- Vergewissern Sie Sich, dass für jede Nachricht die Richtung und die korrekte Sequenzbezeichnung stehen.

## Erstellen des Analysemodells

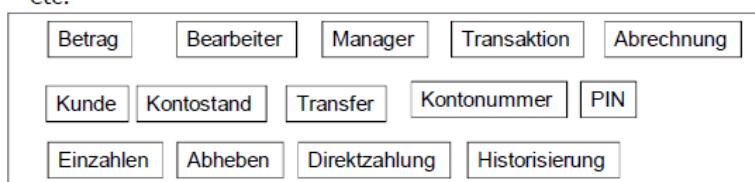
Folgende Schritte können bei der Erstellung des Analysemodells ausgeführt werden:

- Identifizieren von Objekten und Klassen
- Erzeugen von Anwendungsfallrealisierungen
- Vorbereiten eines Datenverzeichnis
- Identifizieren von Assoziationen zwischen Klassen
- Identifizieren von Attributen auf einem abstrakten Niveau (textlich)
- Organisieren und Vereinfachen von Klassen durch Vererbung
- Verfeinern des Modells (iterativ)
- Gruppierung der Klassen in Module (Pakete)

## Finden von Analyseklassen:

### Das Erzeugen von Objekten und Klassen

- Das Erzeugen von Boundary- und Control-Klassen in einem ersten Durchgang ist einfach:
  - Eine Boundary-Klasse für jede Akteur und Anwendungsfall-Beziehung
  - Eine Control-Klasse für jeden Anwendungsfall.
- Entity-Klassen sind schwieriger zu identifizieren. Dazu muss die Anwendung, der abgebildete Geschäftsbereich und die vorhergehenden Aktivitäten betrachtet werden:
  - Physische Entitäten wie Benzin, Maschinen etc.
  - Logische Entitäten wie Mitarbeiterunterlagen, Geschwindigkeit etc.
  - "Weiche" Entitäten wie Tokens, Ausdrücke, Datenströme, etc.
  - Konzeptionelle Entitäten wie Bedürfnisse, Bedingungen, Einschränkungen etc.



## Finden von Analyseklassen: Begründen der Klassen

- Unnötige Klassen können in einem weiteren Schritt eliminiert werden. Folgende Kriterien können überprüft werden:
  - Ist die Klasse kohäsiv, ist es ein einheitliches Konzept?
  - Gibt es redundante Klassen? (z.B. Historisierung und Abrechnung)
  - Sind Klassen irrelevant?
  - Sind Klassen ungenau?
  - Sind Klassen eher Attribute? (z.B. Kontostand, -nummer, PIN)
  - Sind Klassen eher Operationen? (z.B. Abheben, Einzahlen, Transfer)
  - Ist eine Klasse ein Implementierungskonstrukt?



## Erzeugen von Anwendungsfallrealisierungen

- Jeder Anwendungsfall wird untersucht. Hier werden die Boundary und Control-Klassen erzeugt. Weitere Entity-Klassen können gefunden werden.
- Starten mit dem Ereignis, das den Anwendungsfall auslöst. Dieses geht an eine Boundary-Klasse.
- Verfolgen der Sequenz von Aktionen, verbinden der benötigten Klassen.
- Eventuell neu gefundene Klassen werden im Datenverzeichnis festgehalten werden.
- Verteilen des Verhaltens auf die gefundenen Klassen.
- Der Hauptpfad resultiert in einem Kommunikationsdiagramm.
- Alternative Pfade können zu weiteren Kommunikationspfaden führen.
- Das Finden von Klassen und das Erzeugen von Anwendungsfallrealisierungen ist ein sehr iterativer Prozess.

## Identifizieren von Attributten

- Attribute sind oft Substantive eines "Besitzers", z.B. die Farbe des Autos, die Position des Kursors.
- Meist ist das Anfügen von neuen Attributen keine grosse Sache, deshalb ist es nicht schlimm, wenn nicht alle gleich gefunden werden.
- Zum Starten kann die Verantwortlichkeit des Objekts und die Geschäftsdomäne betrachtet werden.
- Entity-Attribute sollten mit Elementen der realen Welt übereinstimmen (z.B. Name und Adresse des Kunden)
- Boundary-Attribute repräsentieren Eigenschaften der Schnittstelle.
- Control-Attribute haben im Grobentwurf meist keine Attribute.
- Es sollten hier nur Attribute der Analyse, keine implementierungsabhängigen Designattribute aufgezeichnet werden.

## Vorbereiten des Datenverzeichnisses

Das Datenverzeichnis definiert jedes Element, das bei der Analyse festgehalten wird. Beispiel für den Online-Geldautomaten:

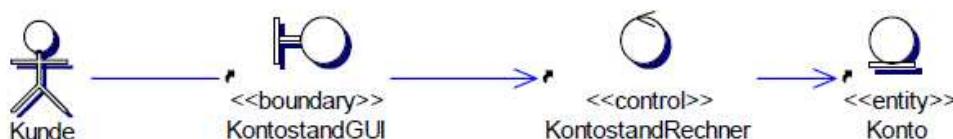
Name	Beschreibung	Annahmen	Attribute
Kunde	Klient der Bank mit einem Konto		Kontonummern, PIN
Bearbeiter	Benutzer des Systems		
Manager	Privilegierter Benutzer	Kann Saldo direkt ändern	
Konto	Informationen eines Kontos		Kontostand
Abrechnung	Aufzeichnung der Einzahlungen, Abhebungen	Sortiert	
Transaktion	Ein eingezahlter oder abgehobener Betrag		Betrag
Direktbezug	Hält Details einer Direktauszahlung fest.	Arbeitet mit Transaktion	

## Finden von Assoziationen

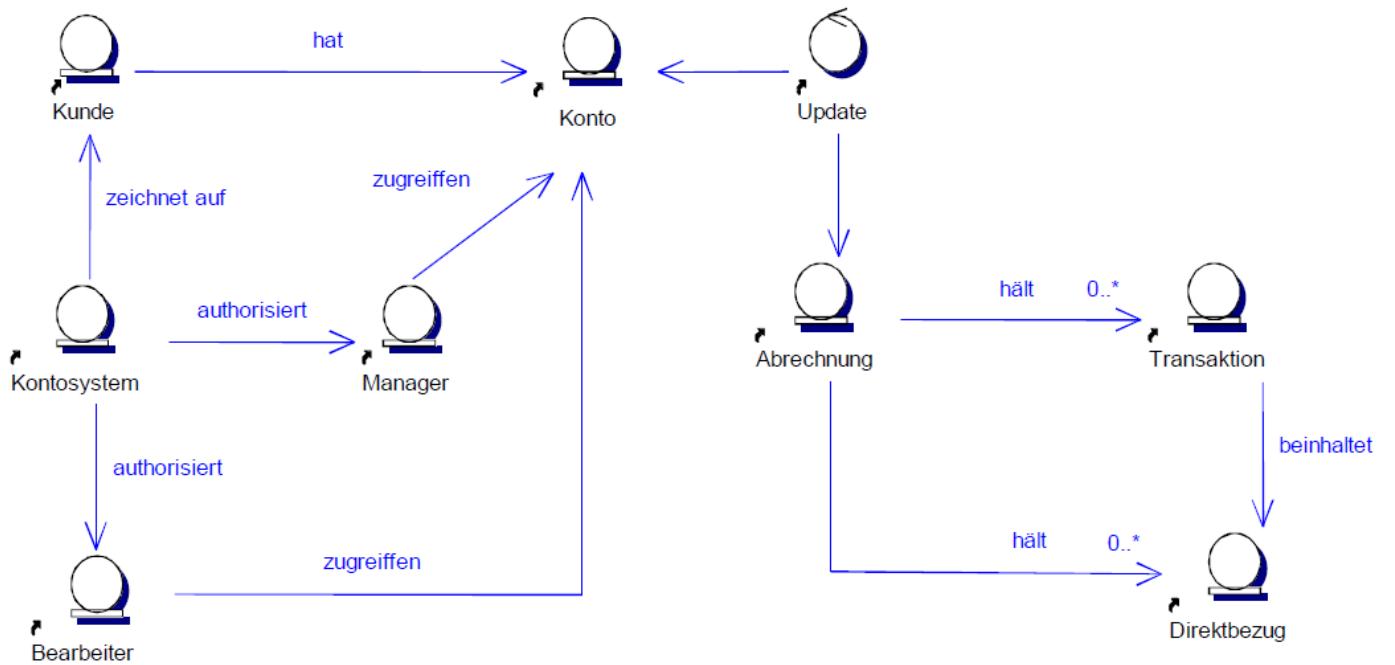
Eine Assoziation repräsentiert eine Verbindung zwischen Analyseobjekte. Sie können durch einen Pfeil orientiert sein. Alle Analyseklassen haben mindestens eine Assoziation. Basis-Assoziationen sind in den Anwendungsfallrealisierung enthalten. Verbindungen zwischen Entity-Klassen können die Architektur beeinflussen und müssen besonders genau betrachtet werden.

Assoziationen können gefunden werden, indem die Verben in der Spezifikation hinterfragt werden. In einer zweiten Phase können die Enden der Assoziationen mit einem Wertebereich (Multiplizität) ergänzt werden. Hinweise:

Physische Platzierung, Beinhaltet, Bestehend aus, konzeptuelles Ganzes (Studenten einer Klasse), Aktionen (fahren), Kommunikation, Besitzer, bestimmte Bedingungen (arbeitet für, verheiratet mit)

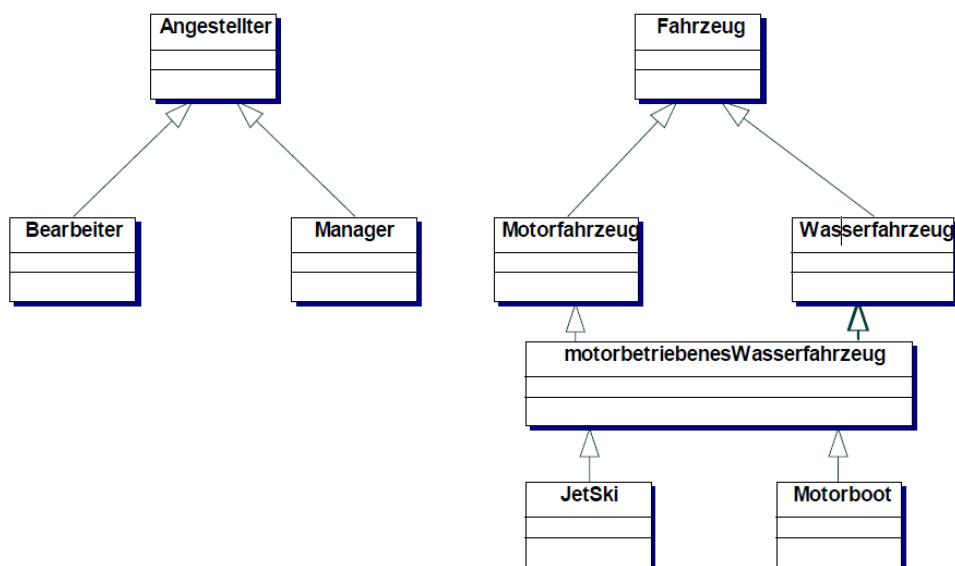


### Beispiel: Online-Bankautomat

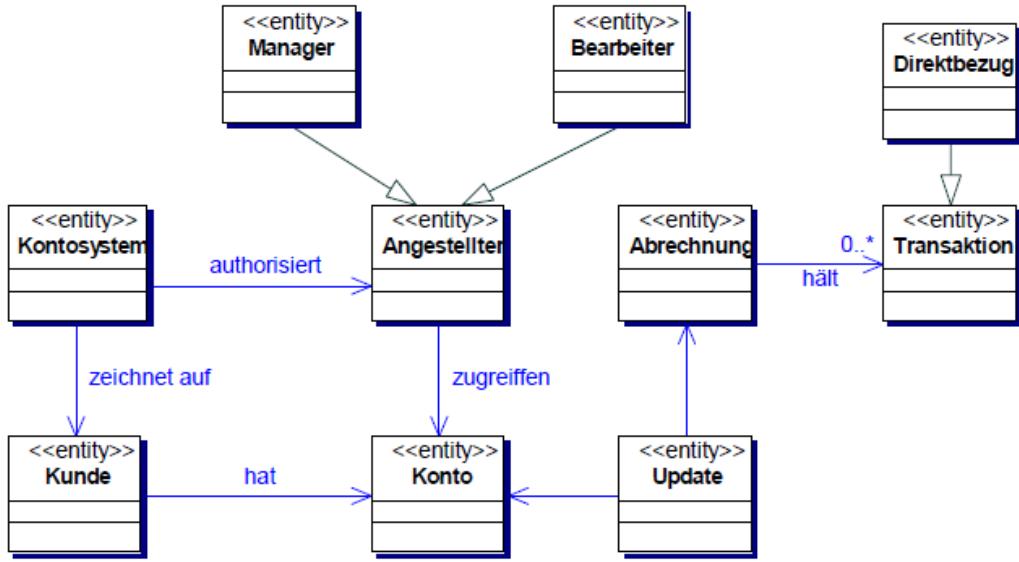


### Vereinfachen der Klassen durch Vererbung

Die Verallgemeinerung (Vererbung) geschieht durch die Identifizierung von gemeinsamem Verhalten. Durch die Verallgemeinerung in der Analyse soll die Klarheit des Modells gefördert werden, nicht die Wiederverwendung. Ziel ist das Verstehen und Zeigen von Gemeinsamkeiten. Verallgemeinerung wird durch einen nicht ausgefüllten Pfeil von der speziellen (Sub-) zur allgemeinen (Super-) Klasse. Es kann auch multiple Vererbung, wie in CLOS oder C++ möglich, dargestellt werden.



## Analyseklassendiagramm für den Online-Bankautomaten mit Vererbung



## Gruppierung der Analyseklassen in Paketen

Der letzte Schritt ist das Gruppieren der Klassen in Pakete. Ein Paket kann Analyseklassen, Anwendungsfallrealisierungen und andere Pakete enthalten. Pakete sollten nach den Klassen, die zusammenarbeiten oder funktional stark abhängig sind, gebildet werden (Kohäsion). Pakete sollten untereinander eine lose Kupplung aufweisen. Um Analysepakete zu finden, kann wie folgt vorgegangen werden:

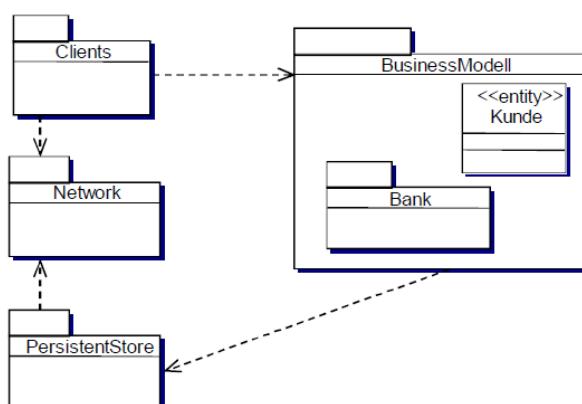
- Zuweisen von Anwendungsfällen zu Paketen basiert auf
- Gleiche Akteure, ähnliche Businessprozesse
- Relationen zwischen Anwendungsfällen
- Zuweisen aller Klassen in das Paket, die mit dem Anwendungsfall zusammenhängen.
- Werden in verschiedenen Paketen gleiche Funktionalitäten verwendet, können diese zu einem neuen Paket gruppiert werden.

## Beispiel für eine Einteilung in Pakete

Pakete werden als Dateien mit Reiter dargestellt. Im Beispiel haben wir die Pakete Clients, Network, PersistentStore, BusinessModell und Bank. Nur das Paket BusinessModell ist im Detail ausgeführt. Das Paket Bank und die Klasse Kunde sind im Paket BusinessModell enthalten.

Die Pfeile repräsentieren Abhängigkeiten zwischen Paketen, d.h. Clients ist abhängig von Network und BusinessModell. Klassen gehören immer zu genau einem Paket, aber können paketübergreifend andere Klassen referenzieren nach dem Schema

*paketName::klassenName*  
z.B.  
*BusinessModell::Kunde*



## UML MODELLE FÜR DIE ANALYSE DISziPLIN 2

### Zustände

Moderne Software orientiert sich heute vermehrt an Geschäftsvorfällen. Die Abarbeitung wird massgeblich durch die Geschäftsobjekte und deren aktuellen Zustände bestimmt. Über flexible Regeln werden anschliessend die anstehenden Arbeitsschritte bestimmt.

→ Mit Zustandsdiagrammen lassen sich solche Szenarien modellieren.

### Zustandsautomat

Ein **Zustandsautomat** stellt das Verhalten eines einzelnen Objekts auf Ereignisse in einem bestimmten Zustand dar.

- **Zustandsautomat** = Zustände + Übergänge (Transitionen)
- **Zustand**: Zeitspanne, in der ein Objekt auf ein Ereignis wartet (andauernde **Aktivität**)
- **Ereignis**: tritt zu einem bestimmten Zeitpunkt auf und hat (konzeptionell) keine Dauer; kann Aktion innerhalb eines Zustands auslösen (durch Operationen realisiert)
- **Transition**: Durch ein Ereignis ausgelöster Zustandswechsel (Aktion)
- Ein Objekt kann sukzessive mehrere Zustände durchlaufen
- Momentaner Zustand und eintretendes Ereignis konstituieren den Folgezustand

### UML Zustandsdiagramm

Ist ein deterministischer endlicher Automat, d.h. ein Automat, der aus endlich vielen Zuständen besteht und bei dem in jedem Zustand der Übergang auf den Folgezustand aufgrund der vorgegebenen Parameter bestimmt ist. Besteht im wesentlichen aus:

- einem Startzustand
- den Zuständen des Objektes
- den Transitionen (Zustandsübergängen)
- einem oder mehreren Endzustände
- Modellierung des Intraobjektverhaltens

UML unterscheidet zwei Typen von Automaten:

1. Die **Verhaltenszustandsautomaten** drücken das Verhalten von Teilen eines Systems aus. Es geht um die Modellierung von diskretem Verhalten durch endliche Automaten. Somit geht es auch um das *Verhalten von Modellelementen*, z.B. von einzelnen Instanzen.
2. Die **Protokollzustandsautomaten** halten das Nutzungsprotokoll von Teilen eines Systems fest. Sie modellieren die zulässigen Transitionen, die ein Classifier auslösen kann. Damit erlauben die Protokollzustandsautomaten, den *Lebenszyklus* von Objekten bzw. die Abfolge von Methodenaufrufen festzuhalten.

Beispiel:



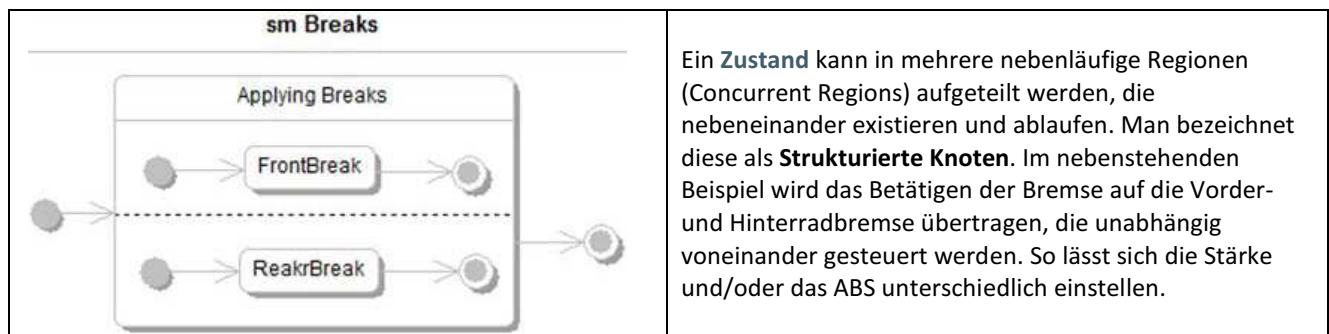
- **Zustände:** offen, geschlossen, versperrt
- **Anfangszustand:** offen (Zustand nach dem Start-Symbol)
- **Ereignisse:** erstelle, öffne, schliesse, sperre\_ab, sperre\_zu
- **Wächterbedingung (Guard Condition):** Ist die Tür im Zustand Offen, dann darf sie nur dann auf das Ereignis schliesse reagieren, wenn die Bedingung "Durchgang ist frei" erfüllt ist.

### Symbole für Zustandsdiagramme

	Der <b>Zustand (State)</b> wird als abgerundetes Rechteck mit Bezeichner dargestellt.
	Ein Zustandsautomat muss einen einzigen <b>Anfangszustand</b> (Initial State) und kann keinen, einen oder mehrere <b>Endzustände</b> (Final State). Fehlt der Endzustand, so kann der Automat nicht geordnet beendet werden. Man muss ihm seine "Lebensenergie" wegnehmen (den Stecker ziehen).
	Ein <b>Übergang (Transition)</b> sind gerichtete Kanten von einem Zustand zu einem zweiten. An einem Übergang kann ein <b>Auslöser (Trigger)</b> angebracht werden, der den Übergang anstösst. Es handelt sich um einen Sammelbegriff für <b>Signal</b> , <b>Ereignis</b> , <b>Wechsel eines Wertes</b> , <b>Wechsel eines anderen Zustands</b> , <b>Ablauf eines Zeitintervalls</b> usw. Der <b>Wächter</b> (Guard, Bedingung in eckigen Klammern) ist eine Nebenbedingung, die den Übergang einschränkt. Der <b>Effekt</b> (Effect) ist der Aufruf einer Aktion, die das Objekt vom alten in den neuen Zustand überführt. Es müssen nicht immer alle Elemente angegeben werden, die Beschriftung sollte aber klarstellen, wodurch die Transition ausgelöst wird.
	Statt den Effekt an den Übergang zu notieren, kann man diesen auch in einer Erweiterung des Zustands in Form von <b>Zustandsaktionen</b> notieren. Dazu wird der Name des Auslösers mit der aufgerufenen Aktion verbunden. Dies ist dann interessant, wenn der Zustand auf eine Vielzahl unterschiedlicher Ereignisse (Pfeile auf sich) mit der gleichen Reaktion antwortet. Typische Aktionen dabei sind: <ul style="list-style-type: none"> <li>• <b>entry</b> (beim Eintritt)</li> <li>• <b>exit</b> (beim Verlassen)</li> <li>• <b>do</b> (dauernd)</li> <li>• <b>Trigger [Guard] / Aktivität</b></li> <li>• <b>Trigger [Guard] / defer (Verzögerung)</b></li> </ul>

<pre> graph TD     Waiting[Waiting] -- "after 2 seconds / poll input" --&gt; Waiting   </pre>	<p><b>Schlaufen</b> (direkte Übergänge) oder Schleifen (indirekte Übergänge über andere Zustände) sorgen dafür, dass der Automat iteriert. Diese Selbstübergänge sind zusammen mit einem Effekt interessant wie zum Beispiel: Fehlermeldung, Polling, Blinken, Wechsel der Reklameeinblendung usw.</p>
<pre> graph TD     Start(( )) --&gt; CheckPIN[Check PIN]     CheckPIN -- "Enter PIN" --&gt; EnterPIN[Enter PIN]     EnterPIN -- "/check PIN" --&gt; Decision{ }     Decision -- "[pin OK]" --&gt; SearchNetwork[Search Network]     Decision -- "[pin invalid]" --&gt; CheckPIN     SearchNetwork -- "network found" --&gt; Ready[Ready]     SearchNetwork -- "power off" --&gt; Off[Off]     Ready -- "power off" --&gt; Off   </pre>	<p>Ein zusammengesetzter Zustand kann wie im Beispiel Subautomaten enthalten.</p>
<pre> graph TD     Start(( )) --&gt; Initializing[Initializing]     Initializing --&gt; Ready[Ready]     Ready --&gt; Ready     Skip(( )) --&gt; Ready   </pre>	<p>Soll ein Subautomat nicht mit seinem Urzustand, sondern einem anderen starten, so benutzen wir benannte <b>Eintrittspunkte</b>, um die Initialisierung zu umgehen. So ist es beispielsweise möglich, Automaten mit Gedächtnis zu entwerfen, die also Informationen über ihr Vorleben speichern.</p>
<pre> graph TD     NotInitialized[Not Already Initialized] --&gt; PA[Performing Activity]     AlreadyInitialized[Already Initialized] --&gt; PA     PA -- "Skip Initializing" --&gt; PA   </pre>	<p>Der oben gezeigte Subautomat mit mehreren Startzuständen wird wie im linken Beispiel aufgerufen.</p>

<pre> graph LR     Start((Initial)) --&gt; PI1[Processing Instructions]     PI1 --&gt; End1(((Final)))     PI1 --&gt; Decision{Failed to Read}     Decision --&gt; WER[Writing Error Report]     WER --&gt; DR[Displaying Results]     DR --&gt; End2(((Final)))   </pre>	<p>Ähnlich den Eintrittspunkten gibt es normale und benannte <b>Austrittspunkte</b> (Exit Point), die von internen Bedingungen abhängen. Ein benannter Austrittspunkt steht für einen Abbruch.</p>
<pre> graph LR     Start((Selecting Message Format)) --&gt; VO[Creating Voice Message]     Start --&gt; SM[Creating SMS Message]     Start --&gt; FA[Creating Fax Message]   </pre>	<p>An einer <b>Entscheidung</b> (Choice Pseudo-State) verläuft der Steuerfluss zu einer der Alternativen. Dazu muss der vorherige Zustand die entsprechenden Entscheidungsinformationen bereitstellen.</p>
<pre> graph LR     VO1[Receiving Voice Message] --&gt; J1(( ))     SM1[Receiving SMS Message] --&gt; J1     FA1[Receiving Fax Message] --&gt; J1     J1 --&gt; VO2[Creating Voice Message]     J1 --&gt; SM2[Creating SMS Message]     J1 --&gt; FA2[Creating Fax Message]     VO2 --&gt; Guard1{[Reply=voice]}     SM2 --&gt; Guard1     FA2 --&gt; Guard1     Guard1 --&gt; VO3[Creating Voice Message]     Guard1 --&gt; SM3[Creating SMS Message]     Guard1 --&gt; FA3[Creating Fax Message]   </pre>	<p>An einer <b>Kreuzung</b> (Junction Pseudo-State) werden mehrere <b>Übergänge</b> (Transition) (mindestens je eine eingehende und eine ausgehende) ohne verbindende Zustände verknüpft. Sie sind semantikfrei. Wächter (Guards) an den ausgehenden Transitionen sorgen für eine eindeutige Entscheidung. Daher dürfen sich die Wächter <i>nicht</i> überschneiden. Eine der Transitionen darf das Schlüsselwort else enthalten. Sie ist dann für alle nichtzutreffende Fälle zuständig.</p>
<pre> graph LR     Start(( )) --&gt; PowerOff[Power Off]     PowerOff --&gt; H((H))     H --&gt; Wash[Washing]     H --&gt; Rinse[Rinsing]     H --&gt; Spin[Spinning]     Wash --&gt; Rinse     Rinse --&gt; Spin     Spin --&gt; End(((Final)))     PowerOff -- "restore power" --&gt; H     PowerOff -- "power cut" --&gt; H   </pre>	<p>Ein <b>Historiezustand</b> (History State) dient dazu, dass ein Unterzustandsautomat seinen Zustand speichert, an dem er unterbrochen wurde, um ihn dort fortzusetzen. Der Historiezustand ersetzt nach der Unterbrechung den normalen Eintrittspunkt.</p>
<p>Es werden flache (H) (Shallow History) und tiefe (H*) (Deep History) Historien unterschieden. Tiefe Historien setzen wir dann ein, wenn der Zustandsautomat mehrfach geschachtelt ist und alle Ebenen wieder in den alten Zustand gesetzt werden sollen. Im obenstehenden Beispiel ist eine flache Historie dargestellt: Die Waschmaschine kann drei Zustände annehmen. Wird die Energie unterbrochen, so soll sie sich merken, in welchem der Zustände sie war und dort den Waschvorgang fortsetzen.</p>	



Übungen 1-3 aus den Folien „Analyse 2“

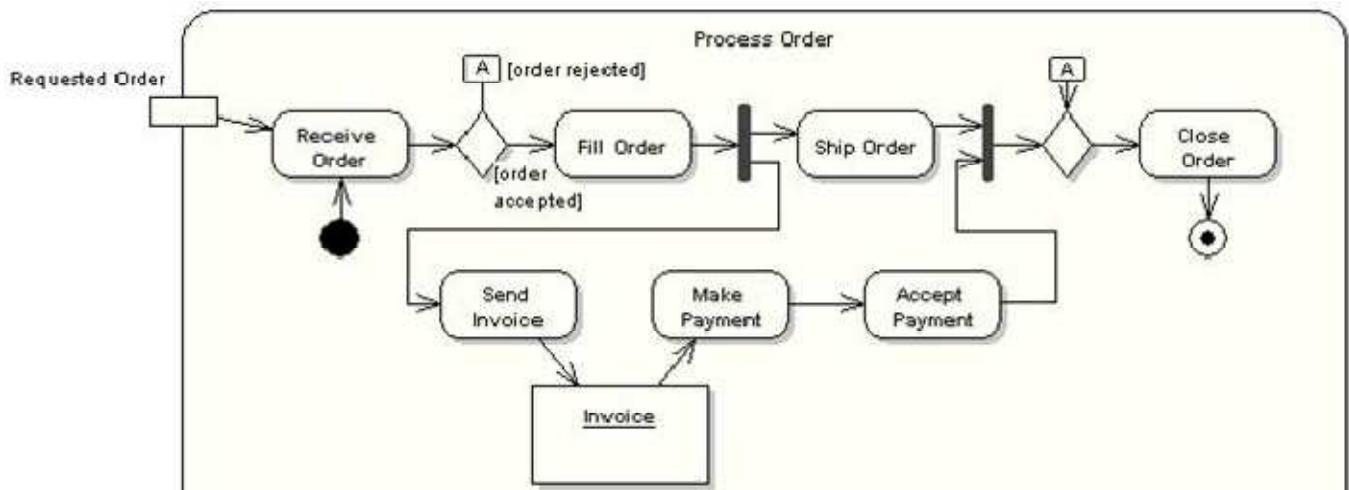
### Aktivitätsdiagramme

Im **Aktivitätsdiagramm** werden die grundlegenden Abläufe im System mit Entscheidungen und ggf. mit Nebenläufigkeit dargestellt.

- Kann verwendet werden, um den Ablauf
  - eines Geschäftsprozesses
  - eines Anwendungsfalls
  - eines Workflows
  - einer Operation
  - etc.

Eine **Aktivität** ist eine Menge möglicher Abläufe, die in der Realität (zur Laufzeit) unter bestimmten Randbedingungen ablaufen können.

Beispiel:



Das Aktivitätsdiagramm ist ein gerichteter Graph mit folgenden Elementen:

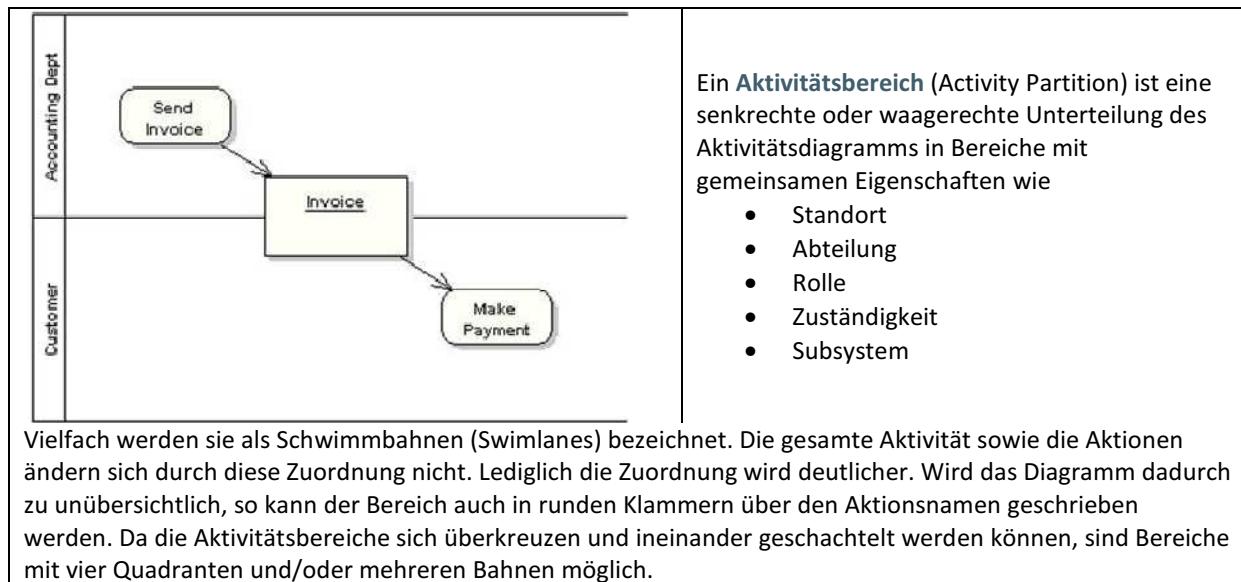
- Aktivität
- Aktion
- Objektknoten
- Steuernode
- gerichtete Kanten

## Symbole für Aktivitätsdiagramme

	In einer <b>Aktivität</b> (Activity) wird eine parametrisierte Folge von Aktionen spezifiziert, die ein bestimmtes Verhalten erzeugen. Die Aktivität umfasst alle Aktionen, Steuerflüsse und anderen Elemente. Die Übergabeparameter werden auf die Umrandung gesetzt.
	Eine <b>Aktion</b> (Action) ist ein Einzelschritt in einer Aktivität. Die nächsten 5 Symbole sind spezielle Aktionen:
	Objekte für Token mit dem Typ <b>Signal</b>
	Aktion zum <b>Senden</b> eines Signals (SendSignalAction)
	Aktion zum <b>Empfangen</b> eines Signals (AcceptEventAction)
	Aktion zum Empfangen eines <b>Zeitsignals</b>
	Aktion zum Aufrufen eines <b>Verhaltens</b> (CallBehaviorAction)
	Wir können eine Aktion mit <b>Einschränkungen</b> (Activity Constraints) versehen, die Vor- bzw. Nachbedingungen genannt werden.
	Ein <b>Objektknoten</b> (Object Node) stellt eine Instanz von Classifiern (meist Klassen bzw. primitiven Datentypen) dar. Dies sind somit Objekte, Variablen oder andere Speicherbereiche. Objektknoten dienen dazu, den Datenfluss durch die Aktivität zu beschreiben. Die beteiligten Aktionen verändern dabei den Inhalt der durchlaufenden Daten-Marken und manifestieren sich im Objektknoten. Der Objektknoten ist aber selbst keine (konkrete Instanz). Er repräsentiert nur eine solche. Ein Objektknoten Rechnung stellt einen logischen Stellvertreter des Datentyps Rechnung (aus dem Klassendiagramm) dar, ist aber nicht die konkrete Rechnung an die Fa. Mayer. Somit verfügt er nicht über Attribute oder Operationen.
	Der <b>Steuerfluss</b> (Control Flow) zeigt die Weitergabe der Steuerung von einer zur nächsten Aktion in Richtung des Pfeils.

	<p>Eine Aktivität verharrt im <b>Startknoten</b> (Initial Node, Start Node), bis sie ausgelöst wird. Damit eine Aktivität deterministisch (zu jedem Zeitpunkt voraussagbar) ist, darf sie <i>nur einen</i> Startknoten haben.</p>
	<p>Entlang eines <b>Objektflusses</b> (object flow) werden Daten bzw. Objekte transportiert. Dabei repräsentiert der Objektknoten das Ergebnis der vorangehenden Aktion bzw. die Eingabe für die nachfolgende Aktion.</p>
	<p>Mit Erreichen des <b>Aktivitätsendes</b> (Activity Final Node) enden auch alle anderen Steuerflüsse in der Aktivität. Logisch gesehen synchronisiert eine Endknoten alle nebenläufigen Prozesse, möglicherweise etwas abrupt. Dies vermeiden wir durch vorherige Synchronisation. Es muss <i>mindestens ein</i> Aktivitätsende geben. Alle Pfade vom Anfangsknoten müssen mindestens einen Weg bis zum Endknoten finden.</p>
	<p>An einem Flussende endet ein nebenläufiger Fluss. Selbstverständlich muss er vorher zur Erledigung der Aktivität beigetragen haben. Existiert nur ein Steuerfluss so wird dieser Endknoten auch als Aktivitätsende interpretiert (obwohl man sich fragt, ob die Verwendung dieses Symbols dann sinnvoll ist). Gibt es mehrere Flussenden, so synchronisieren sich diese gegenseitig, d. h., die Aktivität endet erst, wenn das letzte Flussende erreicht ist.</p>
	<p>Wird die explizite Darstellung des Objektknotens nicht benötigt (z. B. weil nur eine Aktion den Objektknoten generiert), so gibt es eine verkürzte Darstellung, bei dem der Objekttyp direkt an die Aktion angehängt wird. Diese wird als Pin-Notation bezeichnet.</p>
	<p>Noch einfacher wird die Notation mit einem einzelnen Pin zwischen den Objektknoten. Diese wird als alleinstehende Pin-Notation bezeichnet.</p>
	<p>Wenn die Flusspfeile fehlen, sieht man es den Pins nicht an, ob sie zur Ein- oder Ausgabe dienen. Daher kann die Richtung in den Pins ergänzt werden.</p>
	<p>Tritt bei der Verarbeitung eine Ausnahme auf, so wird dies durch ein Dreieck am Ausgabepin modelliert.</p>
	<p>Soll der Objektstrom (streaming) besonders hervorgehoben werden, so füllen wir die Pfeilspitzen bzw. die Pins aus.</p>
	<p>Um wilde Verzweigungen zu vermeiden, treten in der UML die <b>Verzweigungs-</b> (Decision Node) und <b>Verbindungsknoten</b> (Merge Node) paarweise auf. Alle Pfade, die von derselben Verzweigung ausgehen, müssen am selben Verbindungsknoten zusammenkommen.</p>

	<p><b>Parallelisierung (Fork) und Synchronisation</b> (Join) erzeugen nebenläufige Steuerungsflüsse und vereinigen sie wieder durch Synchronisation. Beide Symbole sehen ähnlich aus, haben aber eine unterschiedliche Durchlaufrichtung. Im Gegensatz zum Verbindungsknoten kann ein Synchronisationsknoten erst durchschritten werden, wenn alle Vорbedingungen erfüllt sind.</p>
	<p>Sollen Aktionen auf Elementmengen angewandt werden, so nutzen wir den <b>Mengen-verarbeitungsbereich</b> (Expansion Region). In den Programmiersprachen finden wir hierzu die Zählschleife (for i=0 to n) für indizierte Auflistungen oder allgemeiner (foreach) für beliebige Objektmengen. Über <b>Mengenknoten</b> zur Ein- und Ausgabe werden externe Objektknoten "expandiert" bzw. "komprimiert", woraus sich die englische Bezeichnung ergibt.</p>
<p>Die Schlüsselwörter</p> <ul style="list-style-type: none"> <li>• iterativ</li> <li>• parallel</li> <li>• stream</li> </ul> <p>in der linken oberen Ecke des Bereichs zeigen die Art der Verarbeitung an. Diese Knotenart gehört zum allgemeinen Typ der strukturierten Knoten, die durch abgerundete Rechtecke mit gestrichelter Berandung dargestellt werden.</p>	
	<p><b>Ausnahmebehandler</b> (Exception Handler) dienen zur Behandlung erwarteter Ausnahmen, die in einer geschützten Aktion auftreten können.</p> <p>Mit dem Auftreten der Ausnahme wird ein entsprechendes Ausnahmeobjekt erzeugt, das an den Behandler weitergereicht und dort verarbeitet wird. Dabei ersetzt die Ausnahmeaktion die geschützte Aktion, so dass kein "Rücksprung" (Resume) oder ähnliches möglich ist. Eine geschützte Aktion kann beliebig viele, unterschiedliche Ausnahmebehandler haben, von denen einer aufgrund des Ausnahmetyps ausgewählt wird. Da der Ausnahmeknoten die Funktion des geschützten Knotens im Fehlerfall übernimmt, übernimmt er auch die Ein-/Ausgabeknoten des geschützten Knotens.</p>
	<p>Ein Unterbrechungsbereich (Interruptible Activity Region) umschließt eine Gruppe von Aktionen, die unterbrochen werden können. Neben den Kanten für die "Normalverarbeitung" verlässt eine zusätzliche "Unterbrechungskante" den Bereich. Erhält der Bereich das Unterbrechungssignal, so wird die Normalverarbeitung abgebrochen. Der Bereich wird aufgeräumt und über die Unterbrechungskante verlassen.</p>



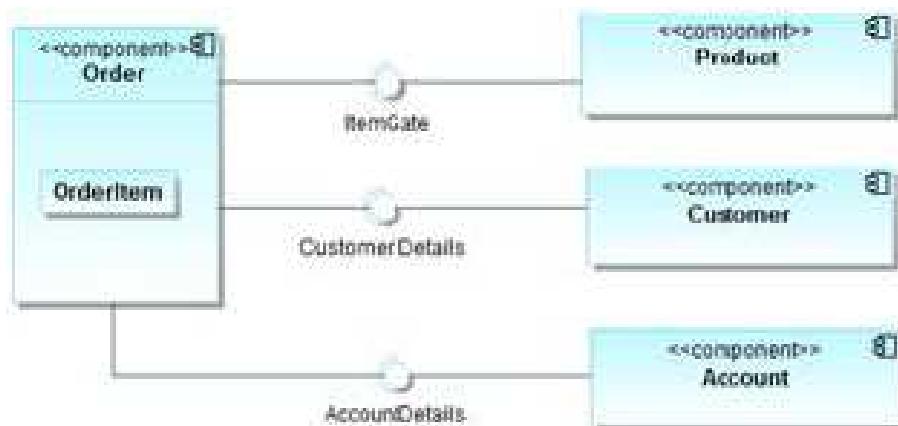
Übung 1+2 aus den Folien „Analyse 2“

## UML Diagramme

### Komponentendiagramm

Ein **Komponentendiagramm** stellt die Struktur eines Systems zur *Laufzeit* dar.

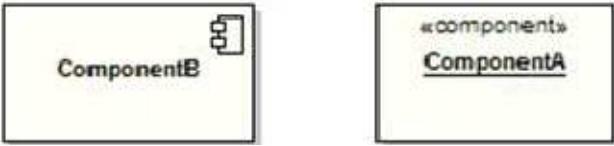
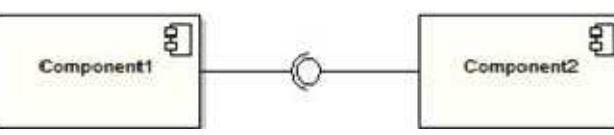
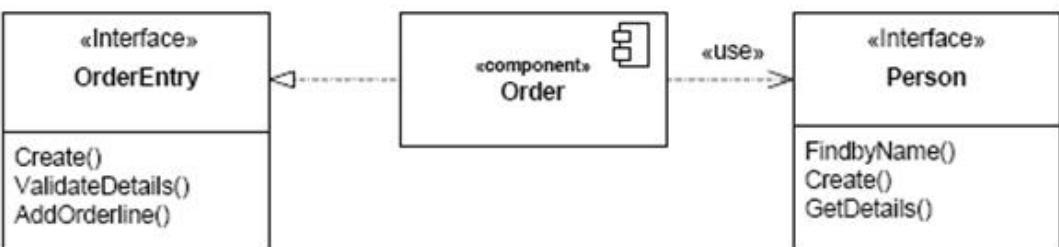
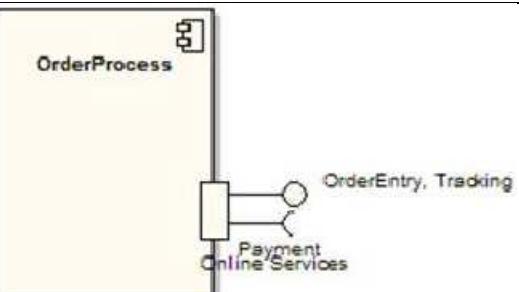
- Ein Komponentendiagramm gibt Antwort auf die Frage: "Wie ist mein System strukturiert und wie werden diese Strukturen zur Laufzeit erzeugt?"
- Komponentendiagramm stellen die Teile eines Systems wie Software, Steuerungen, usw. dar. Es hat damit ein höheres Abstraktionsniveau als ein Klassendiagramm. Normalerweise wird jede *Komponente* durch eine oder mehrere Klassen (bzw. Objekte) zur Laufzeit realisiert.
- Komponenten können selbst wieder Modellelemente enthalten, so dass eine schrittweise Verfeinerung vom Groben ins Detail möglich ist

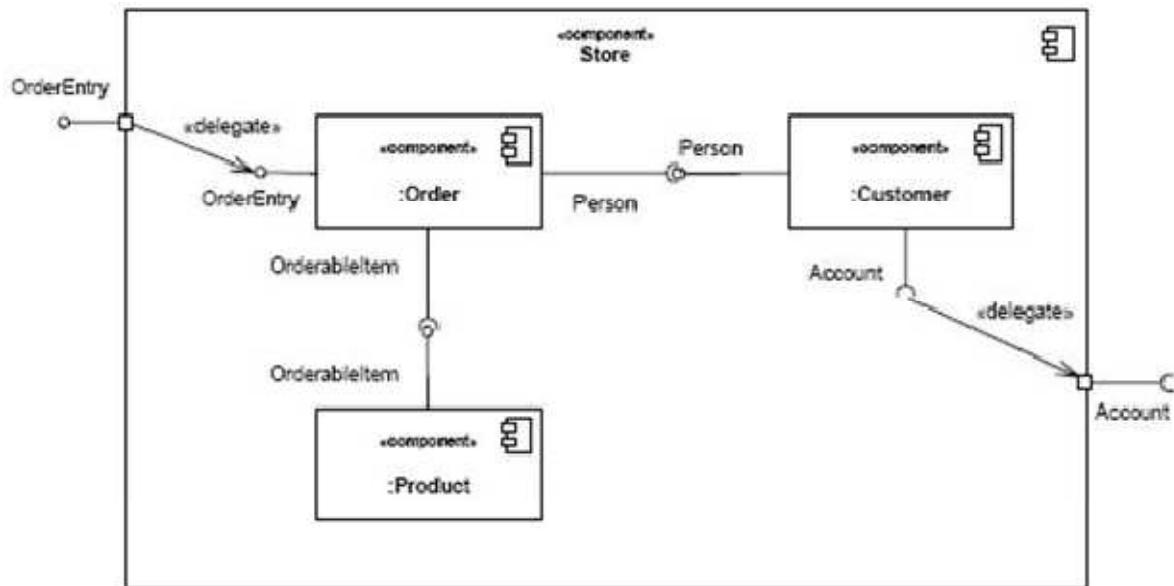


Das Diagramm zeigt einige Komponenten und deren Beziehungstypen.

- Assembly-Konnektoren "verknüpfen" die bereitgestellten Schnittstellen von Produkt und Kunde mit den fordernden Schnittstellen von Bestellung.
- Ein Abhängigkeitsbeziehungstyp bildet die Details einer entsprechenden Zahlung des Kunden auf die anfordernde Schnittstelle Zahlung ab, die durch die Rechnung eröffnet wird.
- *lollipop*-Symbole zeigen an, dass es Dienstanbieter und -verbraucher gibt.

Elemente:

	<p>Die Komponente kann auf zweierlei Weise gezeichnet werden, entweder mit einem Rechteck und zusätzlichem Symbol oder mit dem Stereotyp «component».</p>
	<p>Assembly-Konnektoren verbinden die Komponenten mit den bereitstellenden und den anfordernden Schnittstellen. So ist es möglich, dass eine Komponente die Dienste einer anderen in Anspruch nimmt. Die Dienste sind eine Menge von Operationen, die bei Bedarf Attribute enthalten..</p>
	<p>Ein Assembly-Konnektor zeigt somit an, welche Komponente welche Dienste (service) zur Verfügung stellt bzw. benötigt. Auf der anderen Seite ist der Konnektor an eine Schnittstelle oder einen Port der jeweiligen Komponente angeschlossen. Entsprechend der Detailierungsstufe können wir die Assembly-Konnektoren vollständig darstellen.</p>
	<p>Verwenden wir Ports zur Darstellung der Schnittstellen, so passen sich die Dienste der jeweiligen Umgebung an. Ports können dabei Ein- und Ausgabe oder beides repräsentieren. Das Beispiel zeigt einen Port <b>Online Services</b> mit den bereitgestellten Schnittstellen OrderEntry (Bestelleingabe) und Tracking (Auftragsverfolgung) und der anfordernden Schnittstelle Payment (Zahlung).</p>

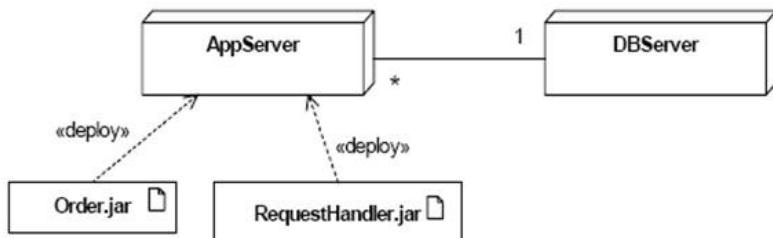


In der Glashausansicht (white-box view) können wir eine Komponente in Parts aus Detailkomponenten weiter zerlegen. Es ist aber auch möglich, andere Modellelemente wie Klassen in die Verfeinerung zu zeichnen.

### Verteilungsdiagramm

Das **Verteilungsdiagramm** zeigt die dynamische Zuordnung von Softwarekomponenten auf Hardwareeinheiten, die **Knoten** genannt werden.

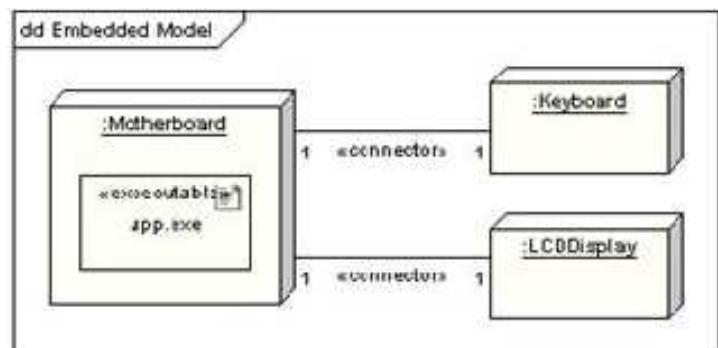
Das Verteilungsdiagramm zeigt also die Laufzeitarchitektur des Systems an. Dazu gehören die Hardwareknoten und die Software, die auf ihnen läuft.



Das Beispiel zeigt die Kommunikationspfade zwischen zwei Knoten zum Verteilen von Softwarekomponenten von einem Datenbankserver auf mehrere Anwendungsserver mit den dazugehörigen Artefakten (hier jar-Dateien). Eine Kante zwischen zwei Knoten im Verteilungsdiagramm zeigt eine Kommunikation zwischen diesen auf. Die gestrichelten Pfeile mit dem Schlüsselwort «deploy» ermöglichen es dem Knoten, einen Komponententyp zu unterstützen, mit dem wir die verteilten Artefakte (hier Softwarekomponenten) auflisten können.

Elemente:

	Ein Knoten (Node) ist ein Hard- oder Softwareelement.
	Eine Knoteninstanz (Node Instance) ist die physische Ausprägung eines Knotens. Der Name ist unterstrichen. Dann folgt nach dem Doppelpunkt der Knotentyp. Damit können wir einen konkreten Knoten von den Knotenklassen unterscheiden. Typischerweise stellen wir so konkrete Server usw. dar.
	Ein Artefakt (Artifact) ist ein Produkt des Software-Entwicklungsprozesses. Das beginnt bei den Prozessmodellen, geht über zu den Quelltexten, den ausführbaren Programmen, den Entwicklungsdokumenten, den Testberichten, den Prototypen, den Benutzerhandbüchern usw.
	Eine Reihe von Stereotypen (Stereotypes) ist vordefiniert. Handelt es sich um Hardware, so sprechen wir von Geräten, bei Software von Ausführungsumgebungen.
<p>Die Assoziation (Association) zeigt in einem Verteilungsdiagramm die Kommunikationswege zwischen den Knoten. Das Beispiel zeigt ein Verteilungsdiagramm für ein Netzwerk mit seinen Netzwerkprotokollen als Stereotypen. Die Assoziationen sind mit Kardinalitäten belegt.</p>	



Ein Knoten kann als Containerknoten fungieren, indem es andere Elemente wie Komponenten und Artefakte aufnimmt. Das Beispiel zeigt ein Verteilungsdiagramm für ein eingebettetes System, bei dem die Anwendung auf der Mutterplatine enthalten ist.

### Erlaubte Pfadsymbole

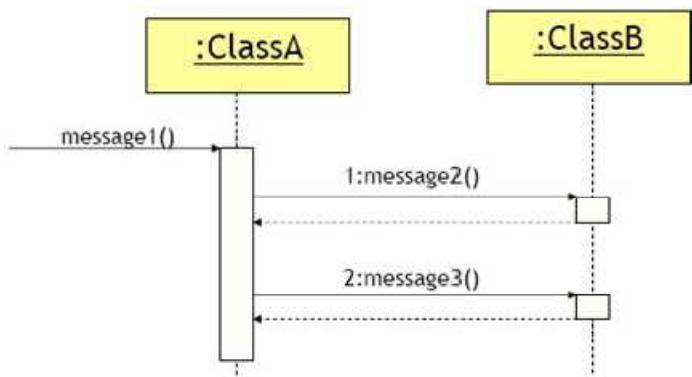
- Assoziation (Association)
- Abhängigkeit (Dependency)
- Generalisierung (Generalization)
- «deploy» Verteilung (Deployment)
- «manifest» Manifest (Manifestation)

### Sequenzdiagramm

Ein **Sequenzdiagramm** stellt die Interaktionen zwischen Kommunikationspartnern dar.

Dabei werden neben der Tatsache, dass kommuniziert wird, auch die Daten und die zeitlichen Abhängigkeiten dargestellt. Die wesentlichen Notationselemente sind:

- Interaktionen
- Lebenslinien
- Nachrichten
- Kommunikationspartner (in der Notation: jedes Objekt der angegebenen Klasse)
- Symbole zur Ablaufsteuerung



#### Nachrichten:

Eine **Nachricht** (Message) repräsentiert den Informationsfluss zweier Kommunikationspartner während einer Interaktion. Nachrichten haben immer einen Sender und einen oder mehrere Empfänger.

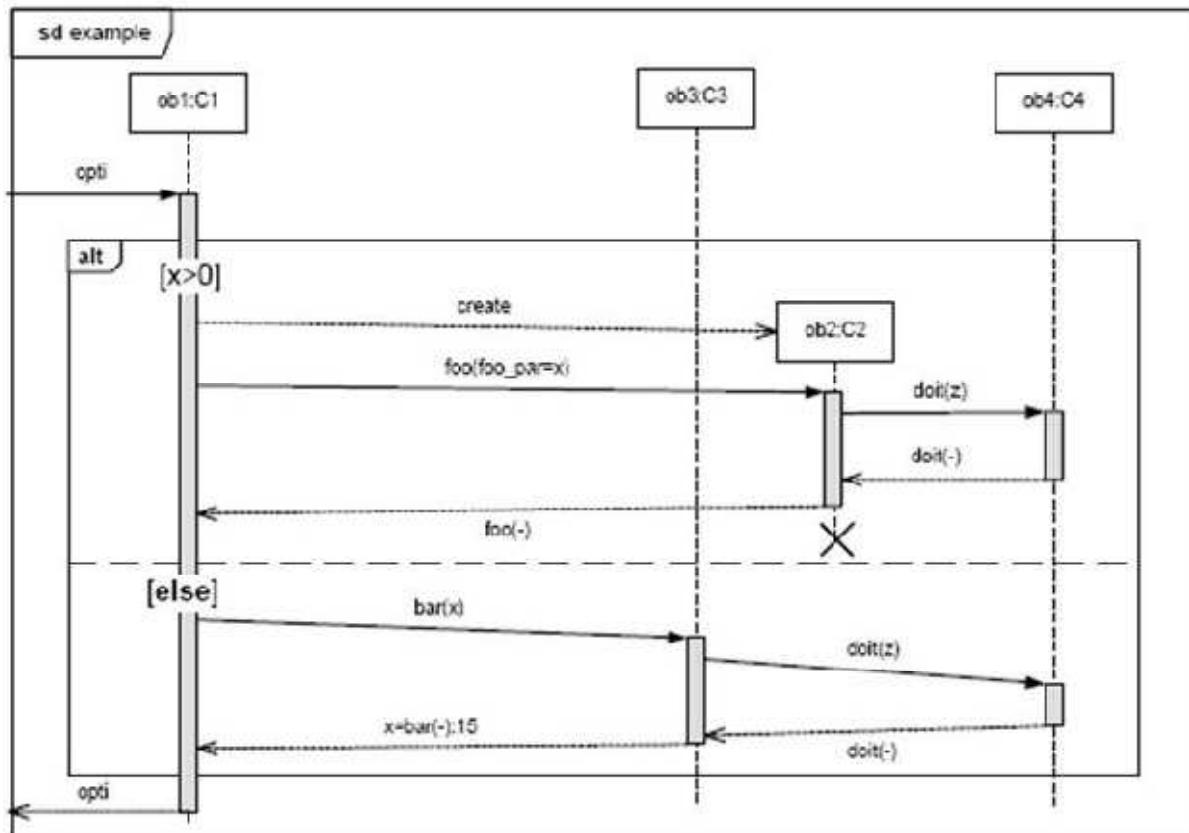
Nachrichten können synchron und asynchron verarbeitet werden. Bei der **synchronen Kommunikationsart** wartet der Sender, bis der Empfänger die Nachricht in geeigneter Weise bestätigt. Im Allgemeinen hat er dann die Aufgabe ausgeführt. Damit muss eine Bestätigung (Quittung) zurücklaufen, die weitere Ergebnisdaten enthalten kann. Bei der **asynchronen Kommunikationsart** setzt der Sender seine Nachricht ab und arbeitet normal weiter. Damit laufen mindestens zwei Programmfäden (Multithreading) parallel ab. Für die Nachrichtenverarbeitung gilt

das **Kausalitätsgesetz**, dass keine Nachricht ankommen kann, bevor sie ausgesandt wurde. Andererseits kann die Reihenfolge des Nachrichtenversands mehrerer Nachrichten entlang einer Lebenslinie streng sequentiell oder unabhängig voneinander erfolgen.

UML erlaubt eine Reihe spezieller Nachrichten:

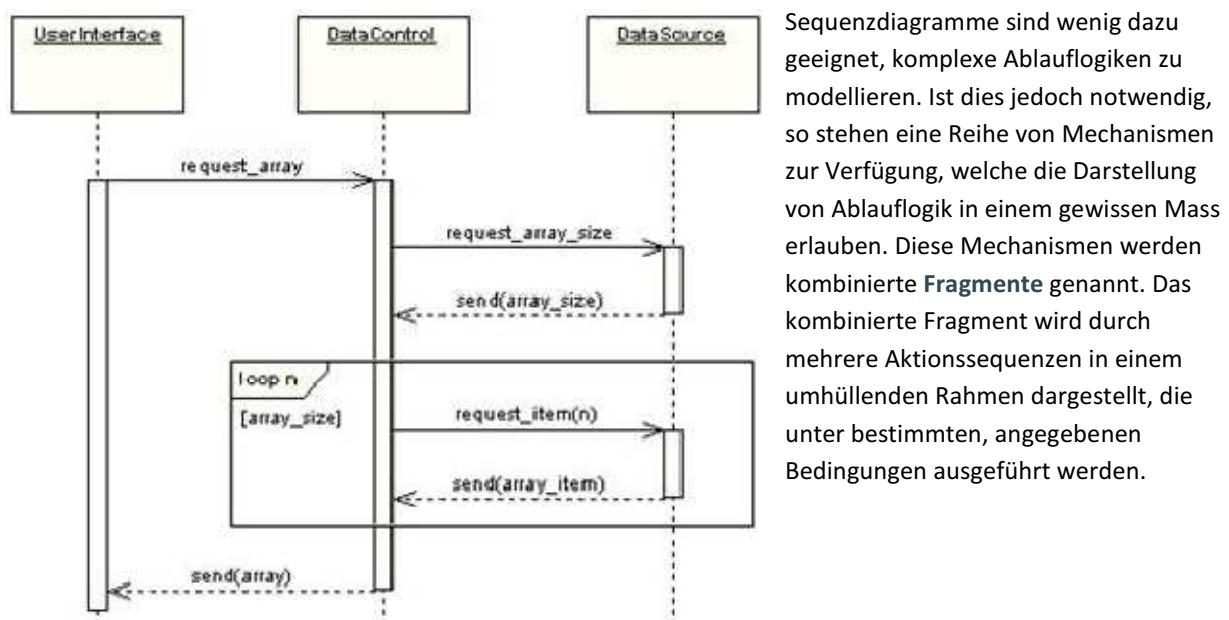
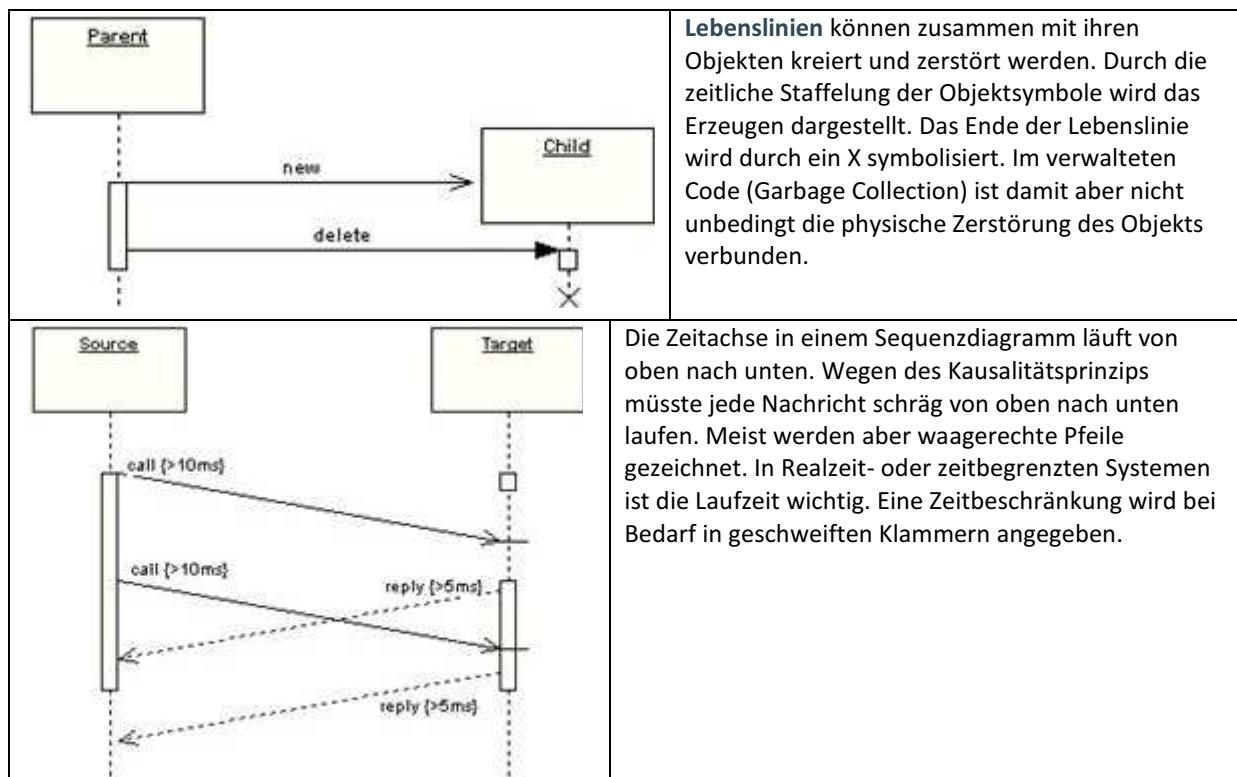
- **Erzeugungsnachricht** (Create Message) kreieren Kommunikationspartner (typischerweise Objekte) und die mit ihnen verbundenen Lebenslinien. Dies entspricht dem Instanziieren von Objekten.
- **Verlorene Nachricht** (Lost Message) sind Nachrichten, deren Empfänger nicht modelliert werden. Landläufig handelt es sich um Rundsprüche (Broadcast Message). Die Nachrichten gehen also nicht wirklich verloren.
- **Gefunden Nachricht** (Found Message) sind Nachrichten, bei denen der Sender nicht modelliert wird. Alle Sender ausserhalb des Modellierungsbetriebs gehören zu diesen Sendern. Der Empfang eines Rundspruchs gehört zu den typischen Ereignissen.
- **Signal** (Signal) sind kleine Datenpakete, die zwischen den Kommunikationspartnern ausgetauscht werden. Bei den Signalen steht das Auslösen einer Operation oder die Übertragung eines einzelnen Wertes im Vordergrund.

Beispiel:



Elemente:

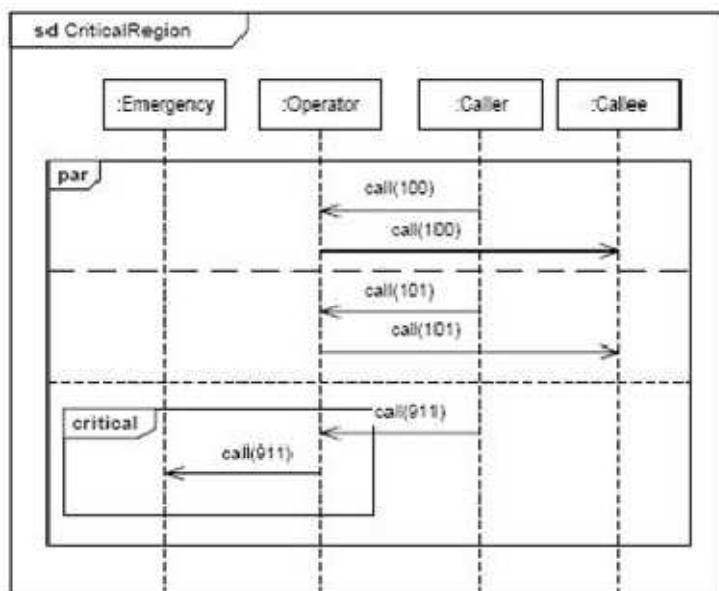
	Individuelle Teilnehmer (Participants) am Systemablauf werden mit je einer Lebenslinie (Lifeline) dargestellt. Self zeigt die Lebenslinie des Classifiers, der das Sequenzdiagramm enthält, an. Rechts sehen wir die Darstellung eines Objekts (Klasseninstanz). Gilt das Diagramm für alle Objekte einer Klasse, so lassen wir den Objektname weg.
	Bei speziellen Akteuren werden die Lebenslinien mit dem Akteursymbol ergänzt. Dies tritt normalerweise dann auf, wenn das Sequenzdiagramm zu einem Anwendungsfalldiagramm gehört.
<pre> sequenceDiagram     participant Source     participant Target     Source-&gt;&gt;Target: return:= message(parameter)     activate Target     Note over Target:          message(parameter)     deactivate Target     Target--&gt;&gt;Source: message(return)   </pre>	Nachrichten können komplett (bestätigt), verloren oder gefunden; synchron oder asynchron; Aufrufe oder Signale sein. Im nebenstehenden Diagramm ist die erste Nachricht synchron (gefüllte Pfeilspitze) und vollständig (mit impliziter Rückmeldung). Die zweite Nachricht ist asynchron, ebenso wie die asynchrone Rückmeldung. Die eigentliche Ausführung der Aktion ( <b>Aktionssequenz</b> ) wird (optional) durch ein schmales Rechteck entlang der Lebenslinie eines Objekts dargestellt. Die Aktionssequenz beginnt mit einem Startereignis und hört mit einem Endereignis auf. Nachrichten können nur während der Ausführung einer Aktion gesendet werden. Der Empfang einer Nachricht startet oft die Aktion im Empfängerobjekt.
<pre> sequenceDiagram     participant Source     Source-&gt;&gt;Source: selfmessage     activate Source     Note over Source: recursion     deactivate Source   </pre>	Mit einer <b>rekursive Nachricht</b> können wir den rekursiven Aufruf einer Operation darstellen. Es ist aber auch denkbar, dass eine zweite Methode desselben Objekts aufgerufen wird, die wiederum als Aktionssequenz in die aktuelle Sequenz geschachtelt wird.
<pre> sequenceDiagram     participant Lifeline     Lifeline-&gt;&gt;Target: lost_message     Lifeline--&gt;&gt;Source: found_message   </pre>	<b>Verlorene Nachrichten</b> sind solche, die zwar gesendet werden, aber nicht den vorgesehenen Empfänger erreichen, oder die an einen nicht im Diagramm dargestellten Empfänger gerichtet sind. <b>Gefundene Nachrichten</b> erhält ein Objekt von einem unbekannten oder nicht im Diagramm dargestellten Sender.



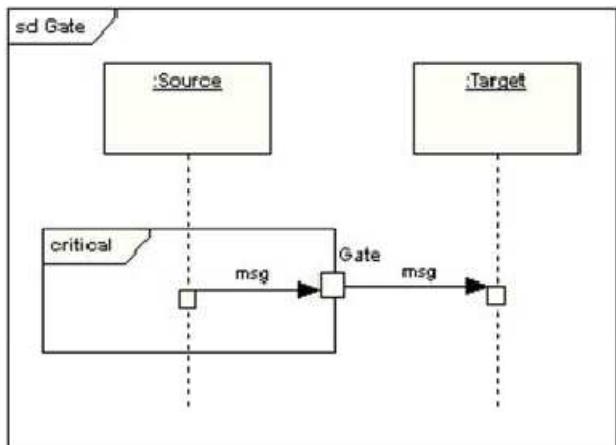
### Fragmente:

- Alternative Fragmente (**alt**) (Alternative Fragment) modellieren Auswahl-Konstrukte if...then...else...
- Optionale Fragmente (**opt**) modellieren 0/1-Auswahlen. Das Fragment wird je nach Bedingung ausgeführt oder nicht. Es entspricht dem alternativen Fragment mit leerer Alternative.
- Abbruchfragmente (**break**) modellieren einen abweichenden Steuerfluss, indem sie den Rest des Diagramms ersetzen.

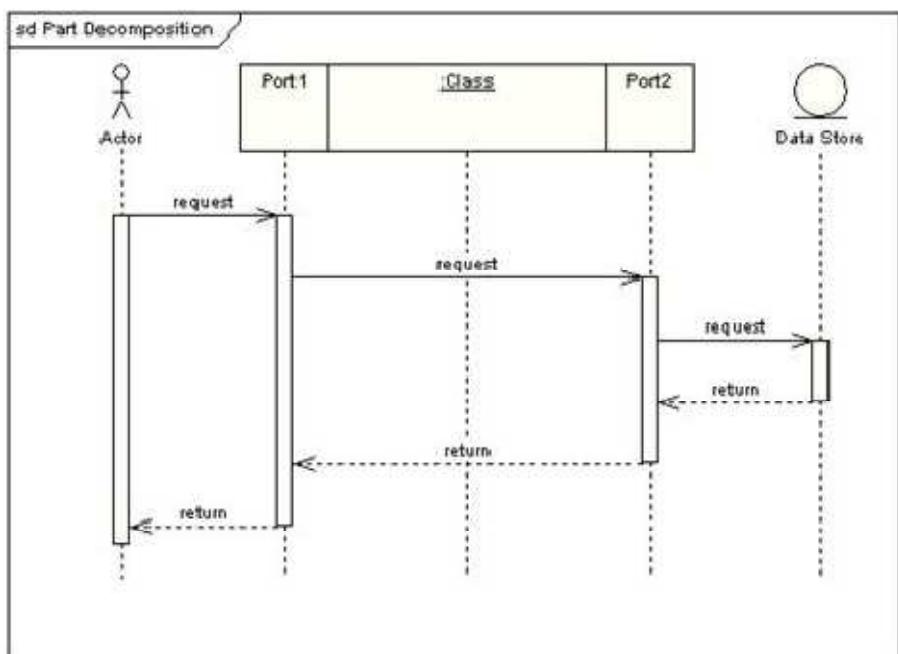
- Parallelle Fragmente (**par**) modellieren nebenläufige Prozesse.
- Fragmente mit loser Ordnung (**seq**) (Weak Sequencing) umschließen eine Anzahl von Sequenzen, für die alle Nachrichten eines vorauslaufenden Segments abgearbeitet sein müssen, bevor das nächste Segment starten kann. Es gelten jedoch keine Beschränkungen bei der Reihenfolge von Nachrichten, welche nicht die Lebenslinie betreffen.
- Fragmente mit strenger Ordnung (**strict**) (Strict Sequencing) umschließen eine Serie von Nachrichten, die genau in der vorgegebenen Reihenfolge ablaufen müssen.
- Kritische Fragmente (**critical**) umschließen einen kritischen (nicht unterbrechbaren) Abschnitt.
- irrelevante Nachrichten (**ignore**) werden von einem Fragment ignoriert. Dies ist dann sinnvoll, wenn:
  - wir auf einen Modellierungsaspekt für die Interaktion verzichten wollen;
  - beim Ablauf neben den modellierten auch bewusst nicht modellierte Nachrichten auftreten (Zeitgebersignale, keep-alive-Nachrichten usw.)
- Relevante Nachrichten (**consider**) sind das Gegenteil der irrelevanten Nachrichten. Nur diese Nachrichten werden verarbeitet, alle anderen werden als unwichtig eingestuft.
- Sicherstellendes Fragmente (**assert**) ignorieren alle Nachrichten, die nicht in der vorgesehenen Reihenfolge auftreten.
- Schleifen-Fragmente (**loop**) umschließen Nachrichten, die wiederholt werden.
- Interaktionsreferenzen (**ref**) (Interaction Occurrence) sind Bereiche in einer Interaktion, auf die eine andere (ausgelagerte) Interaktion referenziert. Dies wird zur Modellierung klassischer Unterprogrammaufrufe benutzt. Dazu wird die referenzierte Interaktion aufgerufen und abgearbeitet. Nach der Abarbeitung kehrt der Steuerfluss hinter das Referenzfragment zurück.



Fragmente können geschachtelt werden. So arbeitet die Vermittlung bei normalen Telefonnummern (quasi) parallel. Ein Notruf muss dagegen sofort durchgestellt werden, ist also kritisch.



Ein **Verknüpfungspunkt** (Gate) sind Punkte auf dem Rahmen einer Interaktion oder eines Fragments, über den interne Nachrichten nach aussen hin ausgetauscht werden. Das Diagramm enthält ein **kritisches Fragment**, d. h. einen Abschnitt, der nicht unterbrochen werden darf



Ein Objekt kann mehrere Lebenslinien besitzen. Dies erlaubt die Darstellung von Inter- und Intra- objektnachrichten innerhalb desselben Diagramms.

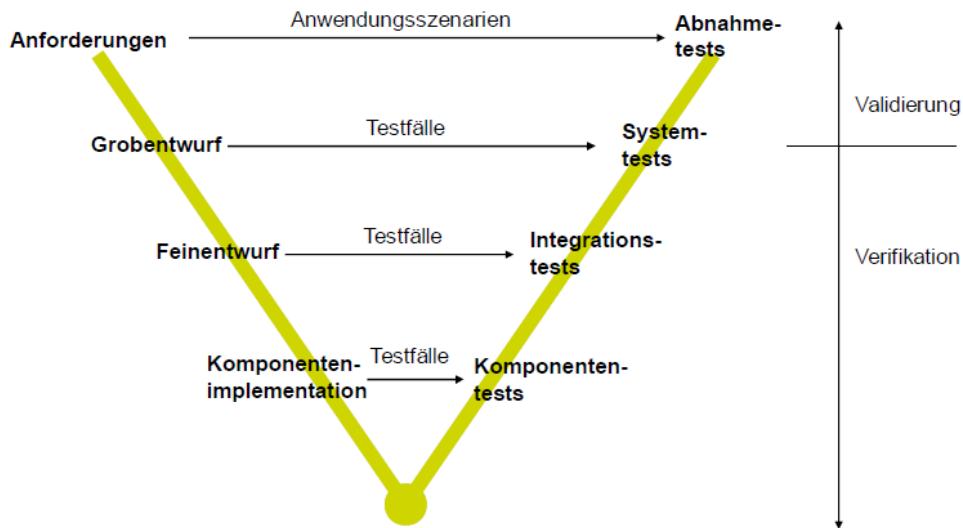
### Regeln beim Entwurf von Sequenzdiagrammen:

Wir können folgende Regeln aufstellen:

- Ein Sequenzdiagramm wird normalerweise für ein Szenario entwickelt.
- An der Kommunikation nehmen konkrete Objekte teil. Wenn es sich um alle Objekte einer Klasse handelt, wird der Klassennname nach dem Doppelpunkt eingetragen.
- Die Objekte werden (wenn möglich) von links nach rechts so sortiert, dass die Nachrichten in dieser Richtung laufen, während die Antworten in Gegenrichtungen erfolgen.
- An den Lebenslinien werden die Aktivierungen eingetragen. Der Steuerfluss erfolgt innerhalb der Aktivierung von oben nach unten.
- Die Aktivierung wird mit einer Nachricht ausgelöst und endet mit der letzten Rückantwort (reply message).
- Gehören Sender- und Empfängerobjekt zur selben Klasse, so werden die Aktivierungen übereinander gestapelt.

## Testen von Komponenten: Funktionsorientierte, dynamische Tests

## Testen im V-Modell

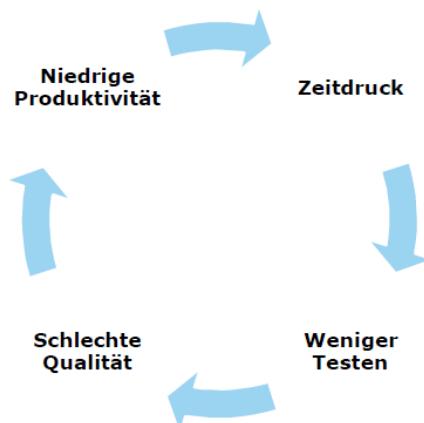


Testen ist nur halbherzig in den Softwareentwicklungsprozess integriert:

- Testen setzt viel zu spat ein, z.B. erst bei der Integration.
  - Testen von Code, der vor Tagen geschrieben wurde, ist schwierig
  - Die Testabteilung testet. Die konnen das eh viel besser
  - Manuelles Testen
    - Aufwendig
    - Testen wird nicht grundlich genug durchgefrt
    - Vermeidung von Regressionstests
  - Code Reviews **statt** Testen des ausfuhrbaren Codes
  - Testen ist ein psychologisches Problem ("Testen von Software ist langweilig und stupide")

→ Mangelnde Qualität der Produkte

## Der Teufelskreis schlechter Tests

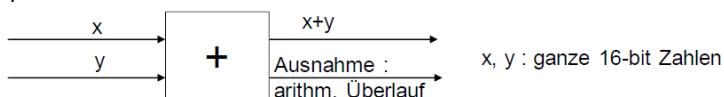


Testen von Software ist jede Ausführung eines Testobjekts, die zur Überprüfung dessen dient.

Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.

## Ziel von Tests

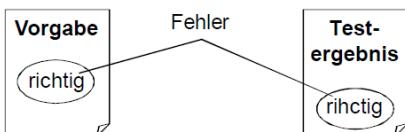
- Wird ein Programm sorgfältig getestet (und sind alle gefundenen Fehler korrigiert), so steigt die *Wahrscheinlichkeit*, dass das Programm sich auch in den nicht getesteten Fällen wunschgemäß verhält.
- Die Korrektheit eines Programms kann durch Testen (ausser in trivialen Fällen) *nicht bewiesen* werden.  
Grund: alle Kombinationen aller möglichen Werte der Eingabedaten müssten getestet werden.
- Beispiel:



- ◆ Anzahl möglicher Eingaben:  $2^{16} \cdot 2^{16} = 2^{32}$
- ◆ Ein vollständiger Test braucht mehr als 4'000'000'000 Testfälle

## Testvoraussetzung

- Testen setzt voraus, dass die erwarteten Ergebnisse bekannt sind
  - Entweder muss **gegen** eine **Spezifikation**
  - oder **gegen vorhandene Testergebnisse** (z.B. bei der Wiederholung von Tests nach Programm-Modifikationen) getestet werden (sogenannter *Regressionstest*)
- Unvorbereitete und nicht dokumentierte Tests sind sinnlos

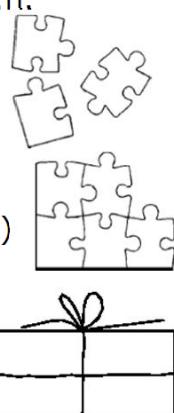


- Mit Testen können nicht alle Eigenschaften eines Programms geprüft werden (z.B. Wartbarkeit)
- Mit testen werden nur Fehlersymptome, nicht aber Fehlerursachen gefunden

## Testgegenstand

- Testgegenstand sind "Programme", d.h.

- Programmkomponenten  
(Modultest, [unit test])
- Systemteile beim Zusammenbau  
(Integrationstest [integration test])
- vollständige Systeme  
(Systemtest [system test],  
Abnahme [acceptance test])



## Testablauf (I)

- Planung
  - Teststrategie: was - wann - wie - wie lange
  - Einbettung des Testens in die Entwicklungsplanung:
    - welche Dokumente sind zu erstellen
    - Termine und Kosten für Testvorbereitung, Testdurchführung und Testauswertung
  - Wer testet
- Durchführung
  - Auswahl der Testfälle
  - Bereitstellen der Testumgebung
  - Erstellung der Testvorschrift
  - Testumgebung einrichten
  - Testfälle nach Testvorschrift ausführen
  - Ergebnisse notieren
  - Prüfling während des Tests nicht verändern

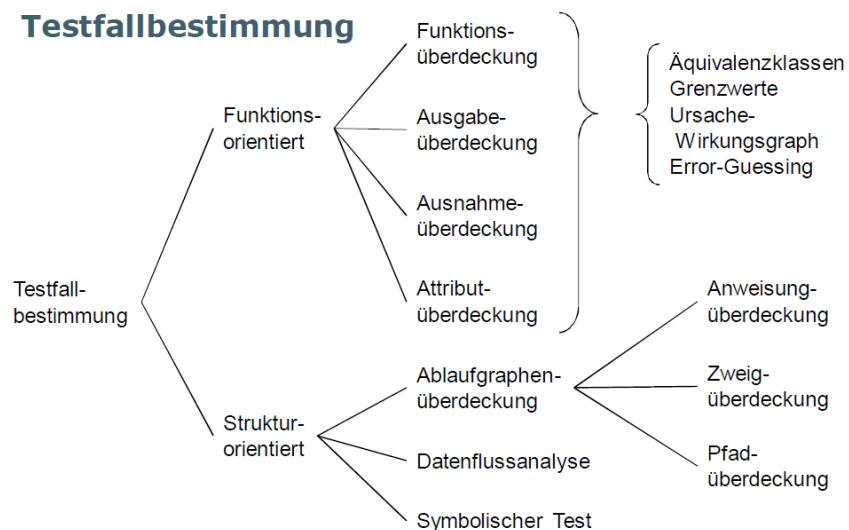
## Testablauf (II)

- Auswertung
  - Testergebnisse in geeigneter Form festhalten (z.B. BugZilla)
  - Testbefunde zusammenstellen.
  - Bei fachlichen Aspekten von Tests ist der Tester auf Drittpersonen angewiesen.
- [ **Fehlerbehebung** (ist nicht Bestandteil des Tests!)
  - gefundene Fehler(symptome) analysieren
  - Fehlerursachen bestimmen (Debugging)
  - Fehler beheben ]

## Testfälle

- Auswahl der Testfälle ist eine zentrale Aufgabe des Testens
- Anforderungen an Testfälle
  - repräsentativ
  - fehlersensitiv
  - redundanzarm
  - ökonomisch
- Ziel: Mit einer möglichst kleinen Auswahl von Testfällen möglichst vielen Fehlern auf die Spur kommen

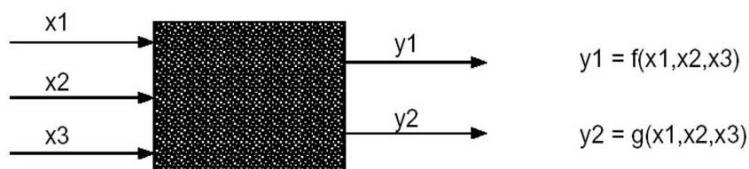
## Testfallbestimmung



## Testverfahren

### ■ funktionsorientiert (Black-Box-Test)

- Testfall-Auswahl aufgrund der Spezifikation
- Programmstruktur kann unbekannt sein



## Funktionsorientierte Tests

- **Strategien für die Testfall-Auswahl:**
  - Abdeckung der Funktionalität gemäss Spezifikation
  - Überprüfung von Leistungsverhalten und Attributen
- Wahl von Testfällen so, dass
  - jede spezifizierte Funktion mindestens einmal aktiviert wird (Funktionsüberdeckung)
  - jede spezifizierte Ausgabe mindestens einmal erzeugt wird (Ausgabeüberdeckung)
  - jede spezifizierte Ausnahme- bzw. Fehlersituation mindestens einmal erzeugt wird (Ausnahmeüberdeckung)
  - die geforderten Attribute (soweit wie dies möglich ist) getestet werden (Attributüberdeckung)
    - insbesondere die Erreichung aller spezifizierten Leistungsanforderungen
      - unter normalen Bedingungen
      - unter möglichst ungünstigen Bedingungen (Belastungstest)

## Techniken der Testfallauswahl

### ■ Äquivalenzklassenbildung

Um eine repräsentative Menge von Eingabedaten zu testen, werden die Eingaben in *Äquivalenzklassen* eingeteilt. Aus jeder Klasse wird ein *Repräsentant* getestet.

### ■ Grenzwertüberprüfung

Da an den Grenzen zulässiger Datenbereiche erfahrungsgemäss häufig Fehler auftreten, werden ausserdem auch Testfälle für solche *Grenzfälle* definiert.

### ■ Ursache-Wirkungs-Graphen

Mit Ursache-Wirkungs-Graphen können Kombinationen von Eingabedaten, die zur Erzielung einer gewünschten Wirkung erforderlich sind, bestimmt werden.

### ■ Fehler raten (Error guessing)

Intuitive Testfallauswahl aufgrund von Erfahrung. Ergänzt andere Methoden zur Testfallbestimmung

## Beispiel

- Gegeben sei ein Programm, das folgende Spezifikation erfüllen soll:  
Das Programm fordert zur Eingabe von drei nicht negativen reellen Zahlen auf und liest die eingegebenen Werte.

Das Programm interpretiert die eingegebenen Zahlen als Strecken  $a$ ,  $b$  und  $c$ . Es untersucht, ob die drei Strecken ein Dreieck bilden und klassifiziert gültige Dreiecke.

Das Programm liefert folgende Ausgaben:

- kein Dreieck wenn  $a+b \leq c$  oder  $a+c \leq b$  oder  $b+c \leq a$
- gleichseitiges Dreieck, wenn  $a=b=c$
- gleichschenkliges Dreieck, wenn  $a=b$  oder  $b=c$  oder  $a=c$
- unregelmässiges Dreieck sonst

Das Programm zeichnet ferner alle gültigen Dreiecke winkeltreu und auf maximal darstellbare Grösse skaliert in einem Fenster der Grösse 10x14 cm. Die Seite  $c$  liegt unten parallel zur Horizontalen. Alle Eckpunkte haben einen Minimalabstand von 0,5 cm vom Fensterrand.

Das Programm liefert eine Fehlermeldung, wenn andere Daten als drei nicht negative reelle Zahlen eingegeben werden. Anschliessend wird mit einer neuen Eingabeaufforderung versucht, gültige Werte einzulesen.

## Testüberdeckungskriterien

### ■ Aktivierung aller Funktionen

- Prüfen und Klassifizieren
- Skalieren und Zeichnen

### ■ Erzeugen aller Ausgaben

- kein Dreieck
- gleichseitiges Dreieck
- gleichschenkliges Dreieck
- unregelmässiges Dreieck

### ■ Erzeugung aller Ausnahmesituationen

- ungültige Eingabe

## Testfall-Auswahl durch Äquivalenzklassenbildung

Klasse, Subklasse	Repräsentant
■ kein Dreieck	
■ a grösste Seite	4.25, 2, 1.3
■ b grösste Seite	1.3, 4.25, 2
■ c grösste Seite	2, 1.3, 4.25
■ gleichseitiges Dreieck	4.2, 4.2, 4.2
■ gleichschenkliges Dreieck	
■ a=b	4.71, 4.71, 2
■ b=c	3, 5.6, 5.6
■ a=c	11, 6, 11
■ unregelmässiges Dreieck	
■ $\alpha$ spitz, $\beta$ spitz	3, 5, 6

## Testfall-Auswahl durch Äquivalenzklassenbildung

Klasse, Subklasse	Repräsentant
■ unregelmässiges Dreieck	
■ $\alpha$ spitz, $\beta$ spitz	3, 5, 6
■ $\alpha$ spitz $\beta$ rechtwinklig	3, 5, 4
■ $\alpha$ spitz $\beta$ stumpf	3, 6, 4
■ $\beta$ spitz, $\gamma$ spitz	6, 3, 5
■ $\beta$ spitz $\gamma$ rechtwinklig	4, 3, 5
■ $\beta$ spitz $\gamma$ stumpf	4, 3, 6
■ $\gamma$ spitz, $\alpha$ spitz	5, 6, 3
■ $\gamma$ spitz $\alpha$ rechtwinklig	5, 4, 3
■ $\gamma$ spitz $\alpha$ stumpf	6, 4, 3
■ ungültiger Eingabe	
■ negative Zahlen	2.3, -1.5, 3
■ Text statt Zahl	2.3, 1.5, xrfk.q
■ unvollständige Eingabe	2.3, 1.5

## Testfall-Auswahl durch Grenzwert-Betrachtung

■ Grenzfall Testwerte
■ kein Dreieck
■ $a=b=c=0$ 0, 0, 0
■ $a=b+c$ 6, 2, 4
■ $b=a+c$ 2, 6, 4
■ $c=a+b$ 2, 4, 6
■ sehr flaches Dreieck
■ $c=a+b - \epsilon$ , $\epsilon$ sehr klein 3, 4, 6.99999999999999
■ $b=a+c - \epsilon$ , $\epsilon$ sehr klein 3, 6.99999999999999, 4
■ sehr steiles Dreieck
■ $c$ klein, $a=b$ sehr gross 107, 107, 5

## Testverfahren

- **strukturorientiert** (White-Box-Test, Glass-Box-Test)
  - Testfall-Auswahl aufgrund der Programmstruktur
  - Spezifikation muss ebenfalls bekannt sein (wegen der erwarteten Resultate)

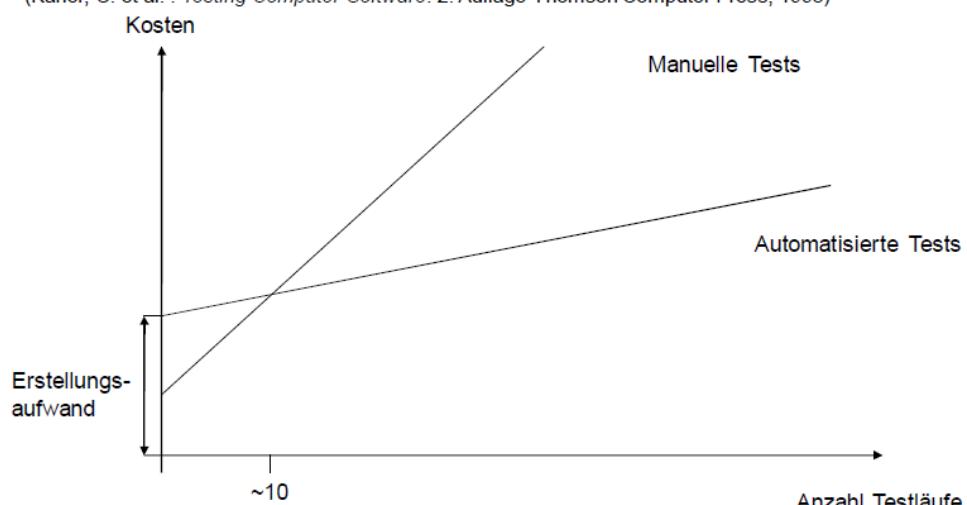
## Üblich: Gray-Box-Tests

- "Gray-Box-Testing consists of methods and tools derived from the knowledge of the applications internals and the environment with which it interacts, that can be applied in Black-Box-testing to enhance productivity, bug finding, and bug analyzing efficiency.  
(Nguyen, 2001)
- Komponentenbasierte IT-Struktur
- Profitieren vom Hintergrundwissen
  - Steigerung der Testeffizienz
  - Konzentration auf Risiken

Funktion	Struktur
Black-Box	Gray-Box

## Kosten für automatisierte Tests

Kaner schätzt, dass bei etwa 10 Läufen der Break-even für eine Testautomatisierung erreicht ist.  
(Kaner, C. et al. : *Testing Computer Software*. 2. Auflage Thomson Computer Press, 1993)



## Testen von Komponenten

- Klassen sind Ausgangspunkt für Testen
  - Klassen sind ideale „Units“ für Komponententests
    - Klarer Vorteil gegenüber prozeduraler Programmierung
    - Was ist die Unit bei prozeduraler Programmierung?  
Prozedur? Modul?
  - Jede Klasse sollte sich selber testen können
  - Mit der Klasse wird auch der Testcode geschrieben
    - Testcode in der zu testenden Klasse oder in separater Klasse?
  - Unit kann auch mehrere Klassen umfassen  
(Komponente)

- Probleme
  - Polymorphismus: kombinatorische Explosion der Testszenarien
  - Vererbung: Änderung einer Klasse → Klienten und Erben testen
- Ziel: Testen ist so einfach wie Compilieren
  - Automatisierbar
  - Schnell → motiviert die Tests nach jeder Änderung durchzuführen

## JUnit

- Framework für „codifizierte“ Tests
- Basis für wiederholbare Komponenten Tests
- Automatisierung
- Infrastruktur für Zusicherungen (Assertions)
- Tests können zu Test Suites zusammengefasst werden
- Benutzerschnittstelle
  - Textbasiert
  - eclipse
  - Swing
  - Ant / XML
- xUnit
  - Erhältlich für praktisch alle gängigen Programmiersprachen

### Ein einfacher Test I

```
class StatModel extends Observable {  
    private Vector list = new Vector();  
    public int sum() {  
        int sum = 0;  
        for (Enumeration e = this.elements(); e.hasMoreElements(); ) {  
            sum = sum + ((Integer)e.nextElement()).intValue();  
        }  
        return sum;  
    }  
    //...  
}
```

### Ein einfacher Test II

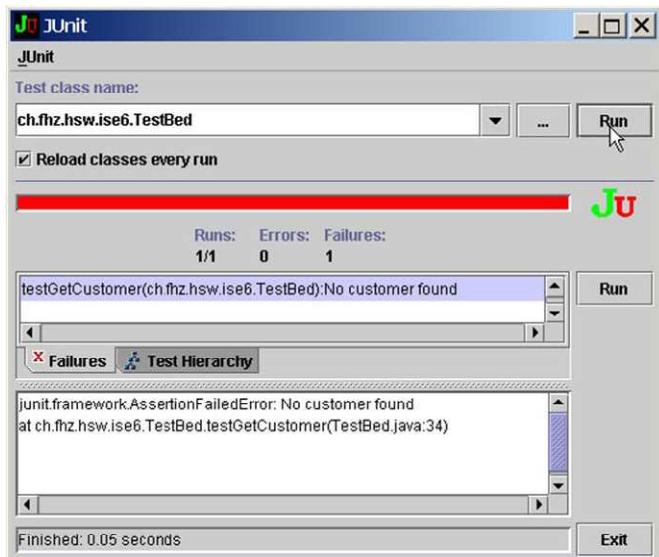
```
import org.junit.*;  
  
class StatModelTest {  
  
    @Test  
    public void testSum() {  
        StatModel m = new StatModel();  
        m.addValue(3);  
        m.addValue(4);  
        m.addValue(5);  
        assertEquals(3, m.size());  
        assertEquals(12, m.sum());  
    }  
}
```

## Test Case

- Test sind mit der Annotation `@Test` (`org.junit.Test`) ausgezeichnet
- Testobjekte sind in Instanzvariablen
- Aufbau
  - Code, der Testobjekte erstellt
  - Ausführung des Tests
  - Verifizierung der Ergebnisse
- Test haben keine Seiteneffekte!
  - Reihenfolge der Tests darf keinen Einfluss haben
  - Vor jedem Test werden Methoden, die mit `@Before` annotiert sind, aufgerufen
  - Nach jedem Test werden Methoden, die mit `@After` annotiert sind, aufgerufen
  - Faktorisierung von Testauf- und -abbau

## JUnit TestRunner II

- Ausführen von Test Suites
  - Einfaches Ausführen einer Menge von Tests
  - Klasse muss lediglich `public static Test suite()` vorsehen
- Visualisierung von Erfolg/Misserfolg
  - „Keep the bar green to keep the code clean“
- Selektives Ausführen von Tests
  - TestBrowser
- Nachladen von Klassen
  - Vermeidet Neustarten des TestRunners
- Auflistung von „Failures“ und „Errors“
  - Fehlermeldungen
  - Stack Traces



## Organisation von Code

- Testcode in einem separaten Package
  - Einfache Organisation
  - Beispiel: Code: `myapp.util`, Testcode: `myapp.utiltest`
  - Klare Trennung zwischen produktivem Code und Testcode
  - Kein Zugriff auf „package private“ Variablen und Methoden
- Testcode im selben Package
  - Zugriff auf „package private“ Variablen und Methoden
  - Keine klare Trennung durch Package
  - Lösungsansatz
    - Verzeichnis `myapp/tests/myapp/util` enthält Testcases
    - `myapp/tests` zum classpath hinzufügen

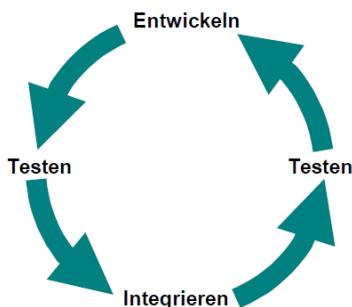
### Anwendung von JUnit I

- Unit Tests
  - Entwickler schreibt Unit Tests für seinen Code
  - Wichtig: Jeder Test muss für sich lauffähig sein
  - → Abhängigkeiten vermeiden
  - Unit Tests sind meistens White-Box Tests
- Funktionale Tests
  - Anwender/Entwickler schreiben Funktionale Tests (Use Cases)
  - Abhängigkeiten schwer vermeidbar
  - Funktionale Tests sind Black-Box Tests

### Anwendung von JUnit II

- Wann soll ein Test Case für ein Stück Code geschrieben werden?
  - Interface ist unklar → zuerst Test Case implementieren
  - Implementierung unklar → zuerst Test Case implementieren
  - Komplexes Interface → ausführbare Spezifikation durch Test Cases
  - Fehler tritt auf → Test Case dafür implementieren
  - Refactoring → zuerst mit Test Case Verhalten des alten Systems „dokumentieren“
- Wie soll ein Test Case für ein Stück Code geschrieben werden?
  - Fehler provozieren (Grenzfälle testen)
- Wann ist eine Test Suite vollständig?
  - Keine weiteren nicht-trivialen Tests vorstellbar

### Ideale Vorgehensweise I



### Ideale Vorgehensweise II

- Implementieren einer Unit
  - Mit Test anfangen, dann Unit implementieren
  - Sobald alle Tests funktionieren, ist die Unit fertig
- Debuggen
  - Für jedes Problem, das auftaucht, wird ein Test geschrieben
  - Debuggen und korrigieren, bis der Test läuft
- Integration
  - Test Suites zusammenfassen
  - Automatisierbares Testen essentiell
- Problem: Laufzeit von grossen Test Suites
  - Nicht mehr möglich, alle Tests nach Änderungen durchzuführen
  - Tests sollten mindestens einmal pro Tag durchgeführt werden
    - Über Nacht, während Sitzungen, über Mittag, etc.

## Funktionales Testen mit JUnit

- Problem
  - Aufsetzen der Testumgebung
  - Testumgebung/Testbett
  - Simulation der Laufzeitumgebung
    - Datenbanken
    - Serverkomponenten
- Lösungsansatz
  - Definition der Server/Datenbankschnittstelle durch Interface
  - Testbett und reale Laufzeitumgebung implementieren Interface
  - Erzeugung der benötigten Umgebung über Fabrik/Fabrikmethode

## Attrappen: Dummy- und Mock-Objekte

- Grundsatz:  
Der einzelne Testfall soll so lokal wie möglich sein, d.h. er soll nur das Objekt testen, das gerade unter der Lupe steht (OUT : Object under Test)
- Problem:  
In einer durchschnittlich komplizierten Anwendung kommt kaum ein Objekt ohne die Mitwirkung zahlreicher anderer Objekte aus.  
Nicht immer sind alle Objekte bereits implementiert oder zur Verfügung!
- Idee:  
Verwendung von Dummy-Objekten (Attrappen), die die anderen Objekte auf eine einfache Art und Weise simulieren.
- Ein Mock-Objekt (Nachahmung) ist ein Dummy-Objekt mit zusätzlicher Funktionalität, z.B. die Verifikation des gewünschten Verhaltens.
- Mit Dummy- und Mock-Objekten ist es möglich, eine Top-Down-Entwicklung zu machen!

## Testumgebung Beispiel I

- Client-Server Applikation
  - Enthält unter anderem Service für Zugriff auf Kundendaten
- Anforderungen
  - Zugriff auf Datenbank über Serviceschicht
  - Bereitstellung des Service zu Testzwecken
  - Austauschen des Testservice ohne Sourcecodeänderungen (d. h. zur Laufzeit)

```
public class Customer {  
    private String lastName;  
    private String firstName;  
    private int id;  
    public Customer(int id, String lName, String fName) {  
        this.id = id;  
        lastName = lName;  
        firstName = fName;  
    }  
    //...  
}
```

## Testumgebung Beispiel II

- Service wird durch Interface beschrieben
- Implementierung durch
  - CustomerServiceTest: zu Testzwecken
  - CustomerServiceDB: für produktiven Zugriff auf Datenbank

```
public interface CustomerService {  
    public void addCustomer(Customer customer)  
        throws DuplicateCustomerException;  
    public Customer getCustomer(int id)  
        throws CustomerNotFoundException;  
    public void deleteCustomer(int id)  
        throws CustomerNotFoundException;  
    public void updateCustomer(Customer customer)  
        throws CustomerNotFoundException;  
    void reset();  
}
```

## Testumgebung Beispiel III

- CustomerServiceTest

```
import java.util.Vector;  
public class CustomerServiceTest  
    implements CustomerService {  
    static Vector customers = new Vector();  
    static Customer def = new Customer();  
  
    public void addCustomer(Customer customer)  
        throws DuplicateCustomerException {  
        int index = getIndex(customer.getId());  
        if (index >= 0)  
            throw new DuplicateCustomerException();  
        customers.add(customer);  
    }  
    //...  
}
```

## Testumgebung Beispiel IV

- ServiceFactory

```
public class ServiceFactory {  
    private static CustomerService customerService;  
    public static Properties serviceProps;  
    public static CustomerService getCustomerService() {  
        try {  
            if (customerService == null) {  
                customerService =  
                    (CustomerService) Class.forName(getProperties().getProperty(  
                        "ServiceFactory.CustomerService")).newInstance();  
            }  
        } catch (Exception e) {  
            System.out.println("Class not found");  
        }  
        return customerService;  
    }  
    //...  
}
```

## Testumgebung V

- Aktueller Service wird über Property eingestellt  
ServiceFactory.CustomerService = CustomerServiceTest  
#ServiceFactory.CustomerService = CustomerServiceDB
- Verwendung des Service

```
public class TestBed {  
    CustomerService service;  
  
    @Test  
    public void testGetCustomer() {  
        service = ServiceFactory.getCustomerService();  
        assertNotNull("No service available", service);  
        try {  
            Customer c = service.getCustomer(1);  
            assertNotNull("No customer found", c);  
            assertEquals(1, c.getId());  
        } catch (Exception e) {  
            fail("No customer found");  
        }  
    }  
}
```

### Konsequenzen

- Starke Integration von Testen in Softwareentwicklung
- Kurzfristig
  - Vertrauen in Code
  - Produktivität (signifikant weniger Fehlersuche notwendig)
- Langfristig
  - System ist länger lebensfähig
  - Automatisiertes Testen erlaubt wesentlich grössere Änderungen
- Wie viele Fehler sind erlaubt?
  - Unit Tests: keine
  - Funktionale Tests: Abhängig vom Aufwand und Priorisierung

## Übung

- Implementieren sie die Klasse „Rectangle“ zusammen mit JUnit Tests:
  - Die Klasse „Rectangle“ soll folgende Funktionen bieten:
    - add: fügt Punkt zum Empfänger hinzu
    - contains: ist ein Punkt im Empfänger enthalten
    - contains: ist ein Rechteck im Empfänger enthalten
    - grow: vergrössere Empfänger um h und v
    - intersection: gib ein neues Rechteck zurück, das die Überschneidung zwischen Empfänger und Rechteck enthält.
    - isEmpty: ist die Fläche des Receiver 0?
    - union: gib ein neues Rechteck zurück, das Receiver und Parameter enthält.
- Sie können sich beim Design der Schnittstelle an der Klasse `java.awt.Rectangle` orientieren. Die Implementierung darf jedoch nicht an diese Klasse delegiert werden.
- JUnit ist bereits in Eclipse integriert. Weitere Informationen unter <http://www.junit.org>