

Softwarekomponenten

Zusammenfassung

Version	1.1
Letzte Aktualisierung	16.06.2010
Autoren	Software - LG171

1.	Einführung in die Softwarearchitektur	7
1.1.	Urbanisation der IT	7
1.2.	Frameworks.....	7
1.2.1.	Das Zachman Framework.....	7
1.2.2.	Architektur.....	8
1.2.3.	Modelle	8
1.3.	Entwicklung von IKT-Systemen.....	8
1.3.1.	Urbanisation	9
2.	Verteilte Systeme und Anwendungen	10
2.1.	Entwicklung von Rechnersystemen.....	10
2.2.	Verteilte Systeme und Anwendungen	10
2.2.1.	Schichten einer Anwendung	10
2.2.2.	Sichten einer Anwendung und physische Konfiguration	11
2.2.3.	Modelle	11
2.2.3.1.	Softwareschichten Modell	11
2.2.3.2.	Architekturmodelle	11
2.2.3.3.	Kommunikation	12
2.2.3.4.	Aufgabenteilung zwischen Client / Server bei 2-Tier-Architektur	12
2.2.3.5.	Anforderungen an Modellauswahl	12
2.2.4.	Midleware	13
3.	Service Oriented Architecture (SOA)	14
3.1.	SOA	14
3.1.1.	Dienst	14
3.1.2.	Rolle von XML in einer SOA	14
3.2.	Web Services	14
3.2.1.	Standards für Web Services	15
4.	Parallele Prozesse/Multithreading	16
4.1.	Parallele Ausführung von Prozessen	16
4.1.1.	Anwendungsbereich	16
4.1.2.	Voraussetzungen für parallele Ausführung	16
4.1.3.	Situation Früher.....	16
4.1.4.	Timesharing Betriebssysteme.....	16
4.1.5.	Prozess-Konzept.....	16
4.1.6.	Threads.....	16
4.2.	Threads in Java	17
4.3.	Zugriff auf gemeinsame Ressourcen und Synchronisation.....	18
4.3.1.	Monitorkonzept (Hochsprachenkonstrukt)	18
5.	Direkte Netzwerkprogrammierung - Sockets	20
5.1.	Prinzip des Socket Aufbaus:.....	20
5.2.	TCP-Sockets in Java:.....	21
5.2.1.	Client:.....	21

5.2.2.	Server:	21
5.3.	Multiserver	22
5.3.1.	Server	22
5.3.2.	Worker Thread:	22

6. Interprozesskommunikation 23

6.1.	Einführung	23
6.2.	Kommunikationsmodelle	23
6.2.1.	Synchrone Kommunikation	23
6.2.2.	Asynchrone Kommunikation	23
6.3.	Datenpräsentation	24
6.4.	Realisierung der IPC	24
6.4.1.	Direkte Netzwerkprogrammierung mit Sockets	24
6.4.2.	Netzwerkprogrammierung mit Middleware	25
6.5.	LPC (Local Procedure Call)	25
6.6.	RPC (Remote Procedure Call)	25
6.6.1.	LPC statt RPC	26

7. Der Entwicklungsprozess von Komponenten 27

7.1.	Wasserfallmodell	27
7.2.	V-Modell	27
7.3.	Spiralmodell	28
7.4.	Wachstumsmodell (Iterativ-Inkrementell)	28
7.4.1.	Unified Process (UP)	28
7.5.	Prototyp-orientiertes Modell	29
7.6.	Agile Software Entwicklung	29
7.6.1.	Extreme Programming (XP)	29
7.6.2.	SCRUM	29

8. Grundlage XML als Grundlagenstandard 30

8.1.	XML	30
8.2.	Lebenszyklus eines XML Dokuments	30
8.3.	XML Beispiel	30
8.4.	Syntaktisches	31
8.5.	Zeichensatz	31
8.6.	Namensraumkonflikte mit Namespaces beheben	31
8.6.1.	Beispiel zu Namensraumkonflikten (xmlns = XML name space)	31

9. Middleware & RMI als konkrete Implementierung 32

9.1.	Arten von Middleware	32
9.1.1.	Kommunikationsorientierte Middleware (KOM)	32
9.1.2.	Anwendungsorientierte Middleware (AOM)	32
9.1.2.1.	Wichtigste Aufgaben der Laufzeitumgebung:	32
9.1.2.2.	Die wichtigsten Dienste der AOM	33

9.1.3.	Nachrichtenorientierte Middleware (NOM) -> wird im Skript nicht erwähnt	33
9.2.	Technologien von Middleware	33
9.2.1.	Object Request Broker (ORB)	33
9.2.2.	Application Server (AS)	34
9.2.3.	Middleware-Plattformen	34

10. Remote Methode Invocation (RMI) - Aufruf entfernter Methoden 35

10.1.	Kommunikation.....	35
10.1.1.	Client.....	35
10.1.2.	Server	35
10.1.3.	Entferntes Objekt.....	35
10.1.4.	Stub und Sekeleton	35
10.1.5.	Namensdienst	35
10.1.6.	Ablauf.....	36
10.2.	Implementierung	36
10.2.1.	Definition der Schnittstelle	36
10.2.2.	Definition der entfernten Klasse	36
10.2.3.	Kompilierung der entfernten Klasse	37
10.2.4.	Anmeldung des entfernten Objekts beim Namensdienst	37
10.3.	Java-Security und Policy-Datei.....	37
10.4.	Verteilung von Klassen.....	37

11. Datenbanken in Verteilten Anwendungen 38

11.1.	JDBC	38
11.1.1.	Einleitung	38
11.1.2.	JDBC Komponenten	38
11.1.3.	JDBC Treibertypen.....	38
11.2.	Ablauf eines Zugriffs auf DB.....	38
11.2.1.	Allgemeiner Ablauf	38
11.2.2.	Laden eines JDBC Treibers	38
11.2.3.	Verbindung aufbauen	39
11.2.4.	Statement kreieren	39
11.2.5.	Schreibender Zugriff.....	39
11.2.6.	Lesender Zugriff	39
11.2.7.	Verbindung schliessen.....	39
11.3.	Transaktionen.....	39
11.4.	DBMS und Property-Dateien	40
11.4.1.	Beispiel db.properties	40
11.4.2.	Auslesen aus .properties Datei.....	40
11.5.	Metadaten	41
11.5.1.	DatabaseMetaData Beispiel: Tabellennamen auslesen.....	41

12. XSLT 42

12.1.	Grundlegende Funktionsweise:	42
--------------	---	-----------

12.2. Anwendungsgebiet von XSLT:	42
12.3. Technischer Ablauf:	42
12.4. Aufbau von XSL (Extensible Stylesheet Language)	42
12.5. XPATH:	42
12.6. XSLT-FO: (Formating Objects)	43
12.7. Knoten	43
12.8. Attribut Ausgabe Beispiel	43
12.8.1. Input File:	43
12.8.2. XSLT File:	44
12.9. Absoluter versus relativer Pfad	44
12.10. Zielformate:	44
12.11. Grundlegender Aufbau eines XSLT Dokumentes	45
12.12. Reihenfolge der Abarbeitung	45
12.13. Namensräume und XSLT	45

13. UML 46

13.1. Use Case Diagram (Anwendungsfalldiagramm)	46
13.1.1. Akteure	46
13.1.2. Assoziationen	46
13.1.3. Anwendungsfall	46
13.1.4. Beschreibung von Anwendungsfällen	46
13.1.5. Beziehungen	47
13.1.5.1. Generalisierungen	47
13.1.5.2. Includes	48
13.1.5.3. Extends	48
13.1.6. Systemgrenze	48
13.1.7. Beispiele	48
13.2. Class Diagram (Klassendiagramm) – nicht prüfungsrelevant	49
13.2.1. Beziehungen	49
13.2.1.1. Assoziation (hat-Beziehung)	49
13.2.1.2. Vererbung (ist-Beziehung)	50
13.2.1.3. Aggregation & Komposition	50
13.2.2. Objekte	50
13.2.3. Schnittstellen (Interface)	50
13.3. Objektdiagramm (Object Diagram) – nicht prüfungsrelevant	50
13.4. Komponentendiagramm (Component Diagram)	50
13.4.1. Ports	51
13.4.2. White-Box-View	51
13.5. Kommunikationsdiagramm	52
13.5.1. Elemente	52
13.5.1.1. Lebenslinie	52
13.5.1.2. Beziehung	52
13.5.1.3. Nachricht (Signal)	52
13.5.1.4. Umrandung	53
13.5.2. Beispiel	53
13.6. Sequenzdiagramm	53
13.6.1. Lebenslinien	54

13.6.2. Nachrichten	54
13.6.3. Lebenslinien Teil 2	54
13.6.4. Fragmente	55
13.6.4.1. Fragmente (Liste)	55
13.6.4.2. Verschachtelung und Gates	56
13.6.5. Kommunikationsdiagramme – Sequenzdiagramme	56

14. Web Services 57

14.1. Web Service Description Language (WSDL).....	57
14.2. Universal Description, Discovery and Integration (UDDI).....	58
14.3. Simple Object Access Protocol (SOAP).....	58
14.3.1. Arbeitsweise:	58
14.3.2. Aufbau einer SOAP- Nachricht:	58
14.3.3. Anhang:	58
14.3.4. Beispiel SOAP:.....	59

15. Message Oriented Middleware (MOM) 60

15.1. JAVA Message Service (JMS)	60
15.1.1. JMS-Klassen	61
15.1.2. Java Naming and Directory Service (JNDI)	61
15.1.3. OpenJMS	62

16. Veranstaltung 10 – Enterprise Service Bus (JBoss) 64

16.1. Enterprise Application (EA)	64
16.2. Enterprise Application Integration (EAI)	64
16.3. Enterprise Service Bus (SOA).....	64
16.4. JBoss ESB	65
16.4.1. Rosetta	65
16.4.2. Services.....	65
16.4.3. Message	65
16.4.4. Listeners	66
16.4.5. Routers	66
16.4.6. Notifiers	66
16.4.7. Konfiguration	66
16.4.8. Erstellen von Nachrichtenschlangen	67
16.4.9. Aktionen und Nachrichten	67
16.4.10. Umgang mit der Nachricht	67
16.4.11. ServiceInvoker	67
16.4.12. Action-Klasse	67

1. Einführung in die Softwarearchitektur

1.1. Urbanisation der IT

















































- Softwaresysteme werden immer **größer**, sind meistens „**verteilt**“, müssen mit anderen Systemen **interagieren** und müssen **zuverlässig** und **sicher** sein.
→ Software wird immer komplexer (komponentenbasierte Schichtsysteme)
- Wegen dieser Komplexität müssen solche Prozesse **abstrakt** dargestellt werden.
- Problematik: Es gibt **keinen einheitlichen Standard** zur Abbildung dieser Abstraktion.
- Drei Arten von Abstraktionen: **Frameworks** (Grundstruktur), **Architekturen** und **Modelle**

1.2. Frameworks

- Das Framework gibt einen Rahmen für die Entwicklung von Software.
- Es ist wie ein **virtuelles Baugerüst**, welches meistens Bibliotheken oder Komponenten wie Laufzeitumgebungen umfasst.
- Zusätzlich zum Framework muss meistens eine **Checkliste** (was muss bei der Modellierung berücksichtigt werden?), einen **Kontext** (zum Verständnis der Zusammenhänge) und ein **konsistentes Grundvokabular** definiert werden.

1.2.1. Das Zachman Framework

- Bildet einen Leitfaden, der Vorschläge enthält, welche Aspekte aus welchen Perspektiven Berücksichtigung finden sollten, um die IT-Architektur einer Unternehmung erfolgreich aufzustellen.

	DATA <i>What</i>	FUNCTION <i>How</i>	NETWORK <i>Where</i>	PEOPLE <i>Who</i>	TIME <i>When</i>	MOTIVATION <i>Why</i>	
SCOPE (CONTEXTUAL)	List of Things Important to the Business 	List of Processes the Business Performs 	List of Locations in which the Business Operates 	List of Organizations Important to the Business 	List of Events Significant to the Business 	List of Business Goals/Strat 	SCOPE (CONTEXTUAL)
<i>Planner</i>	Entity = Class of Business Thing 	Function = Class of Business Process 	Node = Major Business Location 	People = Major Organizations 	Time = Major Business Event 	End/Mean = Major Bus. Goal/Critical Success Factor 	<i>Planner</i>
ENTERPRISE MODEL (CONCEPTUAL)	e.g. Semantic Model 	e.g. Business Process Model 	e.g. Logistics Network 	e.g. Work Flow Model 	e.g. Master Schedule 	e.g. Business Plan 	ENTERPRISE MODEL (CONCEPTUAL)
<i>Owner</i>	Ent = Business Entity Rein = Business Relationship 	Proc. = Business Process IO = Business Resources 	Node = Business Location Link = Business Linkage 	People = Organization Unit Work = Work Product 	Time = Business Event Cycle = Business Cycle 	End = Business Objective Means = Business Strategy 	<i>Owner</i>
SYSTEM MODEL (LOGICAL)	e.g. Logical Data Model 	e.g. "Application Architecture" 	e.g. "Distributed System Architecture" 	e.g. Human Interface Architecture 	e.g. Processing Structure 	e.g. Business Rule Model 	SYSTEM MODEL (LOGICAL)
<i>Designer</i>	Ent = Data Entity Rein = Data Relationship 	Proc. = Application Function IO = User Views 	Node = I/O Function (Processor/Storage, etc.) Link = Line Characteristics 	People = Role Work = Deliverable 	Time = System Event Cycle = Processing Cycle 	End = Structural Assertion Means = Action Assertion 	<i>Designer</i>
TECHNOLOGY MODEL (PHYSICAL)	e.g. Physical Data Model 	e.g. "System Design" 	e.g. "System Architecture" 	e.g. Presentation Architecture 	e.g. Control Structure 	e.g. Rule Design 	TECHNOLOGY CONSTRAINED MODEL (PHYSICAL)
<i>Builder</i>	Ent = Segment/Table/etc. Rein = Pointer/Key/etc. 	Proc. = Computer Function IO = Screen/Device Formats 	Node = Hardware/System Software Link = Line Specifications 	People = User Work = Screen Format 	Time = Execute Cycle = Component Cycle 	End = Condition Means = Action 	<i>Builder</i>
DETAILED REPRESENTATIONS (OUT-OF-CONTEXT)	e.g. Data Definition 	e.g. "Program" 	e.g. "Network Architecture" 	e.g. Security Architecture 	e.g. Timing Definition 	e.g. Rule Specification 	DETAILED REPRESENTATIONS (OUT-OF-CONTEXT)
<i>Sub-Contractor</i>	Ent = Field Rein = Address 	Proc. = Language Stmt IO = Control Block 	Node = Addresses Link = Protocols 	People = Identity Work = Job 	Time = Interrupt Cycle Cycle = Machine Cycle 	End = Sub-condition Means = Step 	<i>Sub-Contractor</i>
FUNCTIONING ENTERPRISE	e.g. DATA 	e.g. FUNCTION 	e.g. NETWORK 	e.g. ORGANIZATION 	e.g. SCHEDULE 	e.g. STRATEGY 	FUNCTIONING ENTERPRISE

- Die Zeilen stellen unterschiedliche Perspektiven (Gesichtspunkte) dar.
- Die Spalten stehen für die unterschiedlichen Fragen, die über das System gestellt werden können.
- Alle Zeilen und Spalten müssen berücksichtigt werden um ein vollständiges Bild zu erhalten.

1.2.2. Architektur

Analog zu
Zachman

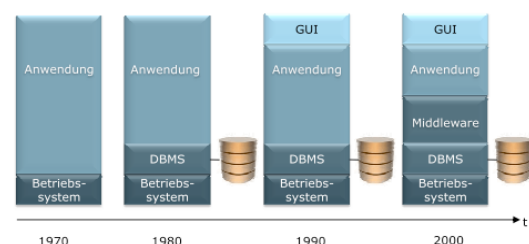
- Die Architektur muss die unterschiedlichen Arten der verfügbaren Bausteinen (**Ressourcen**) definieren und **Regeln** zum Zusammensetzen des Gebildes liefern (Brücken schlagen von Anforderungen zur schlussendlichen Implementation).
- **Bestandteile: Beschreibung der Struktur** der Software-Systeme, **abstrakte Sicht** ohne Details (bezüglich Implementation etc.), **Definition von Blackbox-Elementen** (Übertragungseigenschaft ist bekannt nicht aber deren innere Funktionsweise) und **erste Entwurfsschritte** des Systems.
- **Ziele:** Effiziente Entwicklung, Risiken minimieren, Verständnis schaffen, Kernwissen festhalten
- **Geschäfts-Architektur:** beschreibt was das Geschäft macht und was in Zukunft geplant ist.
- **Komponenten-Architektur:** beschreibt den technologischen Output des Entwicklungs-prozesses, die Module, welche das Unternehmen schlussendlich befriedigen.
- **Technische Architektur:** definiert die technischen Bausteine, die verwendet werden um das IKT-System (Informations- & Kommunikationstechnologie) zu erstellen.
- Architekturen sind **nicht unterschiedliche Schichten** einer Beschreibung (untere Schicht ist detailliertere Beschreibung einer höheren Schicht). **Jede Architektur** beschreibt **etwas anderes**.
- **Zeilen 2,3 und 4** vom Zachman-Framework sind **Architekturen**. Für die anderen brauchen wir **keine Architekturen**: Zeile 1 (unstrukturierte Beschreibung was im Scope liegt), Zeile 5 (enthält technologische Artefakte wie Datendefinitionen oder Programmen) und Zeile 6 (ist das funktionierende Unternehmen selbst).
- **Wie erstellt man eine Software-Architektur?**
 - Geschäftsfall erstellen
 - Anforderungen an Software verstehen
 - Architektur auswählen
 - Architektur analysieren/evaluieren
 - Architektur dokumentieren/veröffentlichen
 - System auf Architektur basierend implementieren
 - Übereinstimmung von Architektur und System sicherstellen

1.2.3. Modelle

- Grundsätzlich eine **Vereinfachung der Realität**
- Ziel ist die Definition, **was getan werden soll**.
- Ein Modell ist ein gutes Mittel um den **Aufgabenbereich** (Scope) eines Vorhabens zu **klären**.
- Als Standard Notation wird häufig **UML** verwendet.

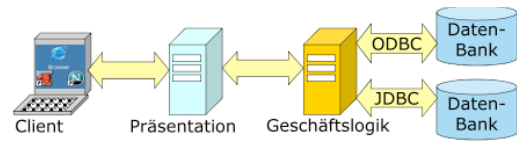
1.3. Entwicklung von IKT-Systemen

- Trend ist immer mehr Richtung **verteilte Systeme**, das heisst **dezentrale** gemeinschaftlich genutzte Systeme.
- Ebenfalls haben die Anzahl Schichten einer Softwarestruktur zugenommen.
Die **Aufgabe der Middleware** ist es die Zugriffsmechanismen auf unterhalb angeordnete Schichten zu vereinfachen und die Details deren Infrastruktur nach aussen hin zu verbergen.
- Auch kann man feststellen, dass heute bei der klassischen Client-Server-Architektur, viel **mehr Softwarekomponenten clientseitig ausgeführt** werden anstatt serverseitig.
- Zudem ist ein Trend von einer Drei-Schicht-Architektur hin zu einer **Vier-Schicht-Architektur**. Diese lagert die Präsentation von Software auf einen zusätzlichen Webserver aus. Dies soll schlussendlich



zu mehr Sicherheit führen (Präsentation ist **demilitalisierte Zone**). Die Geschäftslogik beinhaltet kritische Daten wie zum Beispiel Kundeninfos.

- Eine andere Methode ist der **Application Server**. Er vereinigt die Präsentation mit der Geschäftslogik. Somit hat man hier beide Vorteile: eine **demilitalisierte Zone** sowie eine **zentrale Verwaltung**.



- Bei **webbasierten Anwendungen** muss man natürlich eine **Firewall** verwenden, um unbekannten Nutzern den Zugriff auf vertrauliche Daten zu verwehren.

1.3.1. Urbanisation

- **Urbanisation:** Strukturiertes **Vorgehen** zur Erstellung einer **beherrschbaren IT-Landschaft**.
- **Problematic:** verschiedene IT-Landschaften (Silos): CRM-System (Kundenbeziehungsmanagement), Lagerverwaltung, etc. miteinander zu verbinden und Datenaustausch zu gewährleisten.
- Lösungsversuch 1: **EAI (Enterprise Application Integration)**
Hier werden die verschiedenen Landschaften mittels eines Netzwerks miteinander verbunden. Der Datenaustausch erfolgt dann auf einem Bussystem (EAI).
Nachteile: nur Symptombekämpfung, Software-Updates erfordern EAI Änderungen, nicht benutzerfreundlich wenn mehrere Systeme verwendet werden (verschiedene Menüführungen etc.)
- Lösungsversuch 2: **Integration über Webportale**
Ein Portal mit Konnektoren zu einzelnen Systemen.
Vorteil: benutzerfreundliche, einheitliche Plattform für alle Systeme
Nachteil: Informationen müssen zum Teil zweimal eingegeben werden, da diese nicht an mehrere Landschaften übertragen werden.
- Lösungsversuch 3: **Workflowwerkzeuge**
Sind Prozessmanagement-Tools, welche die Aufgabe haben zu koordinieren wer, was, wann und wie bearbeiten muss.
- **Anforderungen an IT-Landschaft:**
 - rasche Umsetzung von Geschäftsprozessen
 - Echtzeitsicht
 - Wiederverwendbarkeit
 - Sicherung von Informationsaustausch
 - sichere Informationszustellung

2. Verteilte Systeme und Anwendungen

2.1. Entwicklung von Rechnersystemen

1960-1970	Grossrechner, Stapelverarbeitung und Lochkarten / Terminals und die ersten Vernetzungen / Host-Host-Kommunikation
1970 -1980	Grossrechner weiterhin vorherrschend / Timesharing / TCP erfunden
1980 -1990	Personal Computer "erfunden" / Entstehung erster PC-Netzen / TCP/IP setzt sich durch
1990 -2000	WWW / Internet Boom / Vernetzung nimmt rasch zu / Verteilte Systeme immer mehr eingesetzt / Client-Server-Model

Seit Anfang 2000 ist das **Internet omnipräsent**. Es dient als **Businessplattform** und die Orientierung an Internet Technologien nimmt stetig zu.

Da heute immer mehr Geräte ans Internet angeschlossen sind, werden immer mehr IP Adressen benötigt. Das zur Zeit noch verwendete Internet Protokoll **IPv4** ist aufgrund der Adress-Knappheit bald ausgereizt (maximal $232 = 4.294.967.296$ Adressen). Das neu entwickelte **IPv6** (≈ 340 Sextillionen = $3,4 \cdot 10^{38}$, acht Blöcke zu jeweils 16 Bit unterteilt (Hexadezimal)).

2.2. Verteilte Systeme und Anwendungen

Ein verteiltes System (VS) besteht aus einer Menge autonomer Computer, die durch Computernetzwerke miteinander verbunden sind und mit einer Software für die Koordination ausgestattet sind. Sie kommunizieren miteinander und koordinieren Ihre Aktionen, indem sie Nachrichten austauschen.

Ein verteilte Anwendung (VA) ist eine Anwendung, die aus verschiedenen Komponenten besteht und ein verteiltes System zur Lösung eines Anwendungsproblems benutzt

- Einem Standard-Anwender erscheint eine verteilte Anwendung wie eine gewöhnliche, nicht verteilte Anwendung (transparent).

Folgendes sind Beispiele für verteilte Systeme / Anwendungen:

Internet	Im Internet stehen zahlreiche verteilte Anwendungen zur Verfügung: Standardisierte Anwendungen wie FTP, Email und WWW oder Proprietäre Anwendungen wie Amazon-Buchladen, Swiss-Ticketverkauf im Bereich E-Commerce.
Intranet	Geschlossenes Netzwerk, das für einer bestimmten, klar definierten Gruppe von Anwender benutzt wird. In einem Internet können weitere Subsysteme existieren, die der Abwicklung von unterschiedlichsten Geschäften dienen. Beispiele: Filesharing, Drucken von Dokumenten, Kollaborationsplattformen (ILIAS, MS Share Point, ...)
WWW	World Wide Web ist ein Dienst im Internet und sicher die populärste verteilte Anwendung. Er benutzt das Internet als verteiltes System, um Aufgaben zu lösen und wird als Basis für weitere, neue verteilte Anwendungen eingesetzt.

2.2.1. Schichten einer Anwendung

Eine solche Anwendung lässt sich in folgende Schichten unterteilen:

Datenhaltung Datenbanken / Dateien	Datenverarbeitung Geschäftslogik / Prozesssteuerung	Datenpräsentation Diverse Datensichten Kommunikation mit Benutzer (UI)
--	--	---

2.2.2. Sichten einer Anwendung und physische Konfiguration

Logische Struktur der Anwendung soll so auf die **physische Struktur** (Rechner im Netz) abgebildet werden, dass der maximale Nutzen erreicht wird.

Vorteile / Nachteile eines verteilten Systems

Vorteile	Nachteile
<ul style="list-style-type: none"> • Besseres Abbild der Realität • Wirtschaftlichkeit • Bessere Lastverteilung • Bessere Skalierbarkeit • Fehlertoleranz 	<ul style="list-style-type: none"> • Höhere Komplexität durch Verteilung und Heterogenität • Komplexe Netzinfrastruktur • Höhere Sicherheitsrisiken (Verletzlichkeit)

2.2.3. Modelle

Mit einem Modell werden allgemeine Eigenschaften und Design eines Systems beschrieben.
Mit der Definition eines Modells wird Folgendes angegeben:

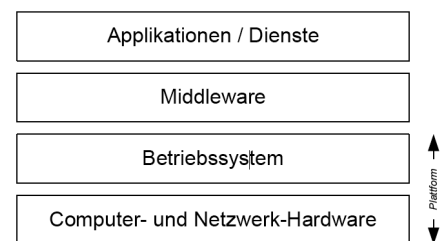
- Die wichtigsten Komponenten des Systems und deren Aufgaben
- Die Interaktion zwischen unterschiedlichen Komponenten des Systems
- Wie das Verhalten der Komponenten (einzeln und kollektiv) beeinflusst werden kann

2.2.3.1. Softwareschichten Modell

Applikationen Unabhängig von einer Plattform, sollte eine spezifisches Middleware-Modell verwenden

Middleware Verbirgt die Heterogenität des verteilten Systems, stellt ein Programmierungsmodell zur Verfügung (API)

Betriebssystem Ermöglicht den Zugriff auf die Systemressource



2.2.3.2. Architekturmodelle

Das Architekturmodell eines verteilten Systems

- vereinfacht und abstrahiert die Erfassung von Funktionen der einzelnen Komponenten
- definiert die Verteilung von Komponenten in einem Netz von Computern
- definiert die Beziehung von Komponenten untereinander (Rolle in der Kommunikation, Kommunikationsmuster)

Architektur Modelle: Peer-To-Peer Modell (P2P)

Jeder Peer-Knoten hat sowohl den Applikationscode als auch Koordinationscode.

Anwendungsbereich: Austauschplattform (Musik, Filme usw.)

Architektur Modelle: Client-Server

Es handelt sich um eine Softwarearchitektur (sowohl Client als auch Server können auf dem gleichen Knoten laufen). Server kann andere Server bei der Beantwortung einer Anfrage "bemühen" und in die Client-Rolle schlüpfen. Ein Server kann i.d.R. mehrere Clients "gleichzeitig" bedienen (Nebenläufigkeit).

Architektur Modelle: Client-Server - Variante Applet

Schritt 1: Client holt sich das Applet (ein in einem Webbrowser laufendes Java-Programm) vom Webserver

Schritt 2: Applet wird auf dem Client-System ausgeführt

Architektur Modelle: Client-Server - Spontane Netzwerke

Spontane Netzwerkverbindungen von mobilen Geräten in einer "fremden" Umgebung

- Zur Zeit Gegenstand intensiver Forschung
- gewinnt mit dem Vormarsch mobiler Geräte (PDA, Laptop, ...) immer mehr an Bedeutung
- Wichtige Begriffe: WirelessLAN, Bluetooth

Beispiel für spontane Netzwerke: WLAN eines Flughafens, Schnelles "Check In", Schnelle Abflug- und Ankunft-Informationen, WLAN eines Hotels, Musikservice, Wäscheservice, Diverse Events, Printservice

Architektur Modelle: Mehrfacher Server

Partition und Replikation von Daten und Diensten. Beispiele: Partition, Replikation

Architektur Modelle Proxy-Server

Der Nutzen: Bessere Performance, Bessere Verfügbarkeit (Cache), Erhöhte Sicherheit (Zugriffskontrolle)

2.2.3.3. Kommunikation

- Synchrone Kommunikation
- Asynchrone Kommunikation

2.2.3.4. Aufgabenteilung zwischen Client / Server bei 2-Tier-Architektur

Zu erledigende Aufgaben: Datenhaltung, Verarbeitung, Präsentation

ThinClient reine Präsentation von Daten

FatClient Präsentation, Verarbeitung und teilweise Datenhaltung

2.2.3.5. Anforderungen an Modellauswahl

Folgende Merkmale spielen bei der Modellauswahl eine tragende Rolle:

- **Anforderungen an Modellauswahl: Performance**

Ein Benutzer hat bestimmte Anforderungen an das System bzw. seine Leistungsfähigkeit
Antworten auf eine Anfrage müssen schnell zur Verfügung stehen und konsistent sein.

Dies wird begünstigt durch

- den Einsatz von **wenigen Komponenten**
- die lokale Kommunikation immer wo möglich
- den Transfer bzw. Austausch von möglichst **kleinen Datenmengen** bzw. Dateneinheiten
Durchsatz von Daten
- die Geschwindigkeit bzw. die Rate, mit der die Rechenarbeit (Berechnung, Zustellung von Daten) erledigt wird
- wird von mehreren Komponenten, die an der Rechenarbeit beteiligt sind, bestimmt
- die schwächste Komponente bestimmt den Durchsatz **Lastbalancierung**
- probiert die stark belasteten Komponenten zu entlasten und die anfallende Arbeit möglichst systemweit zu verteilen
- benötigt unter Umständen die **Replikation** von Daten, womit das ganze System komplizierter wird

- **Anforderungen an Modellauswahl: Dienstgüte (Quality of Service)**

Wenn ein Dienst verfügbar ist, wird probiert, den Dienst möglichst zu verbessern bzw. zu optimieren.
Die Verbesserungen können folgende Punkte beinhalten:

- Zuverlässigkeit und Verfügbarkeit des Systems
- Sicherheit
- Immer mehr **echtzeitorientierte** Parameter

- **Anforderungen an Modellauswahl: Caching und Replikation**

Mit Caching werden bestimmte Daten im Speicher gehalten und auf die Anfrage als Antwort geliefert (schneller Zugriff). Problem: Wie kann die Aktualität von Daten sicher gestellt werden

Mit Replikation soll die Verfügbarkeit von Daten erhöht werden. Problem: Wie kann erreicht werden, dass alle Replikate den gleichen Stand haben.

- **Anforderungen an Modellauswahl: Verlässlichkeit**

Die Verlässlichkeit umfasst

- **Korrektheit**: Das System sollte das erwartete Verhalten zeigen (wie z.B. in der Spezifikation definiert)
- **Sicherheit**: wo ist der sicherste Ablageplatz, die Verfügbarkeit von Daten hat enorme Bedeutung
- **Fehlertoleranz**: inwieweit ein Dienst bzw. System noch zuverlässig arbeitet, nachdem ein Fehler aufgetreten ist

2.2.4. Middleware

Probleme bei der Entwicklung einer verteilten Anwendung

Middleware: Begriff und Aufgaben, Positionierung im OSI-Referenzmodell, Beispiel für Middleware

Das verteilte System wirkt wie eine **Ganzheit**. Der Benutzer nimmt es gar nicht wahr, dass es sich bei dem System, das von ihm benutzt wird, um ein verteiltes System handelt. Es gibt Fälle bzw. Anwendungen, bei denen die **Transparenz** nicht so erwünscht ist.

Transparenztypen

- Access Transparency (Zugriffstransp.)
- Location Transparency (Ortstransp.)
- Concurrency Transparency (Nebenläufigkeitstransp.)
- Replication Transparency
- Failure Transparency (Fehler-bzw. Ausfalltransp.)
- Mobility Transparency (Mobilitätstransp.)
- Performance Transparency
- Scaling Transparency

3. Service Oriented Architecture (SOA)

3.1. SOA

- **Hauptproblematik** bei Softwareentwicklungen sind Unterschiede zwischen dem, was im Informationssystem vor sich geht und was im Prozess aufgezeichnet ist. (**Semantic Gap** zwischen **Geschäftsprozessen** und **Informationssystemen**)
- **SOA** stellt eine **flexible, anpassbare IT-Architektur** speziell **für verteilte Systeme** dar. Sie orientiert sich an **Geschäftsprozessen**, welche auch die Grundlage der SOA sind. Vereinfacht gesagt, ist es eine Struktur, welche probiert alle Unternehmensanwendungen zu integrieren, wobei aber die Komplexität der verschiedenen Anwendungen hinter einer standardisierten Schnittstelle verborgen bleibt.

Mögliche Plattformen sind Java Applikationsserver und .NET von Microsoft.

- **Aufgaben:** Generell ist das die Integration aller Anwendungen, welche auch im wirklichen Geschäftsprozess verwendet werden. Somit kann sehr schnell und flexibel auf Änderungen im Geschäftsprozess reagiert werden.

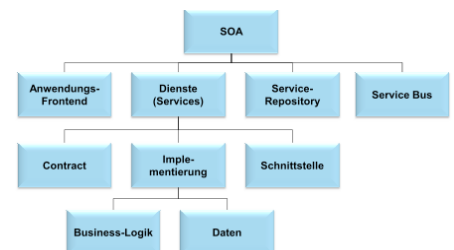
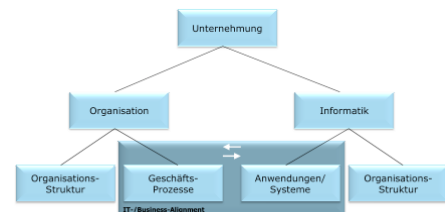
- Präsentation
- Koordination
- Steuerung
- Transaktionen
- Sicherheit

- **Bestandteile:**

- Anwendungs-Frontend ist kein Dienst, sondern reines GUI.
- Service-Repository beinhaltet Programmpakete und deren Beschreibung.
- Service Bus ist eine Art Middleware, welche die Kommunikation über Verbindungswege regelt.

- **Ziel:** Kostensenkung in der Softwareentwicklung und Wiederverwendbarkeit von Applikationen.

- Einige SOA-Produkte: SAP, Oracle SOA Suite, Microsoft Biz Talk, IBM WebSphere



3.1.1. Dienst

- Ein Dienst stellt den Mitarbeitern den **Zugriff auf eine oder mehrere** Geschäftsfunktionen zur Verfügung. Zum Beispiel bei einer Bank stellt die Aufgabe: „Vergib einen Kredit“ einen **Geschäftsprozess**, wo mehrere Personen involviert sind, dar. Hingegen „**Trag den Kunden ins Kundenverzeichnis ein**“ ist ein Dienst.
- **Anforderungen** an einen SOA-Dienst:
 - muss **interoperabel** sein (Kommunikation auf technischer und geschäftlicher Ebene)
 - muss einen **Mehrwert** sowie eine **Qualitätsgarantie** bieten
 - muss **ständig verbessert** werden um Mehrwert zu bieten

3.1.2. Rolle von XML in einer SOA

Die Dienste einer SOA **kommunizieren** meist per XML miteinander. XML ist eigentlich nicht Voraussetzung für eine SOA, hat sich aber als **Standard** durchgesetzt.

3.2. Web Services

- Ein Web Service ist eine **Softwareanwendung**, bei welcher **XML-basierte Nachrichten über Internetprotokolle** ausgetauscht werden.

- Web Services sind **plattform- und implementierungsunabhängige** Softwarekomponenten (Client muss nicht wissen was für eine Sprache, Betriebssystem, etc. der Server verwendet und dass binäre Daten weder gesendet noch empfangen werden).
- Web Services sind mit der **Web Service Description Language** beschrieben (WSDL) (das heisst ein Web Service beschreibt selber, welche Anfragen gemacht werden können, welches Transportmedium verwendet wird etc.).
- Web Services sind auf einem „**registry of services**“ registriert (WS kann einem registry service sagen wo er gefunden werden kann).
- Ein Web Service wird durch eine **deklarierte Programmierschnittstelle** (API) aufgerufen (Antwortformat bekannt).
- Web Services sind **mit anderen Services verbunden** (das heisst ein Service kann gleichzeitig ein Client sein).

3.2.1. Standards für Web Services

- **XML** (eXtended Markup Language): erweiterbares Textformat für den Austausch strukturierter Daten.
- **SOAP** (Simple Object Access Protocol): Transportiert XML Nachrichten.
- **WSDL** (Webservice Description Language): XML Notation zur Beschreibung von Webservices.
- **UDDI** (Universal Description, Discovery and Integration): Verzeichnisdienst zuver Veröffentlichung von Webservices.
- **BPEL** (Business Process Execution Language): Choreographierung und Orchestrierung von Web Services.

4. Parallele Prozesse/Multithreading

4.1. Parallele Ausführung von Prozessen

Mehrere Ausführungseinheiten werden gleichzeitig ausgeführt.

4.1.1. Anwendungsbereich

- Server erbringt Leistungen und bedient dabei gleichzeitig mehrere Clients.
- Robotersteuerungen
- Komplexe Abläufe werden parallel in Subprozesse (Threads) zerlegt (Splitting)
Bsp: Word verarbeitet die grafische Oberfläche und wartet gleichzeitig auf neue Tastatureingaben.

4.1.2. Voraussetzungen für parallele Ausführung

- **Mehrprozessorsystem**
⇒ Waren früher sehr teuer, heute gibt es Multicore-Prozessoren
- **Unterstützung durch das Betriebssystem**
⇒ Muss in der Lage sein, Ausführung auf mehrere Prozessor-Einheiten zu verteilen und zu verwalten.

4.1.3. Situation Früher

Früher hatte man Batch-Betriebssysteme, welche ein Programm von Anfang bis zum Ende ohne Unterbruch abarbeiten.

4.1.4. Timesharing Betriebssysteme

- Durch sogenannte Multitasking-Fähige Betriebssysteme wurde eine scheinbar parallele Ausführung möglich (Präemptives Multitasking)
- Eigentlich arbeitet der Prozessor aber immer noch nur an einem Prozess, er kann aber diesem Prozess die Betriebsmittel über kurze Zeit entziehen, und einen Anderen Prozess abarbeiten. Dies geschieht so schnell, dass es der User meint, die Prozesse werden parallel ausgeführt.

4.1.5. Prozess-Konzept

- Ein Prozess darf nicht merken, dass er unterbrochen wurde und muss nach dem Unterbruch wieder an derselben Stelle weiterarbeiten.
- Jeder **Prozess** hat eine **eigene Umgebung** mit folgenden Informationen:
 - Prozessspezifische Daten (ID, Priorität, ...)
 - Code zum Ausführen
 - Stack (Ablage für die generierten Daten)
 - Pointer (Stack-, Instruction-)
- Es gibt **3 verschiedene Zustände** für einen Prozess:
 - Bereit für die Ausführung
 - In Ausführung
 - Blockiert
- Es gibt verschiedene **Strategien** für die Zuteilung von Betriebsmitteln:
 - First In First Out (First Come First Serve)
 - Shortest Job First
 - Prioritätsscheduling

4.1.6. Threads

- Ein **Prozess** kann **mehrere Threads** beinhalten
- **Kontextwechsel** ist weniger aufwändig als bei Prozessen.
 - Ein Thread benötigt die selben Mittel wie ein Prozess (Pointer, Stack, ...)

- Allerdings sind Programmcode, Prozessspezifische Daten, globale Daten, ... für alle Threads eines Prozesses gemeinsam.
- **Threadzustände:**
 - neu (im Anfangszustand, noch nicht lauffähig, Daten und Methoden können angesprochen werden)
 - bereit zur Ausführung
 - in Ausführung
 - blockiert
 - tot (hat Arbeit erledigt, Daten und Methoden können angesprochen werden, kann nicht mehr gestartet werden)
- **Zustandsübergänge** können durch Methodenaufrufe im Programm oder durch das Betriebssystem hervorgerufen werden.

4.2. Threads in Java

- Java unterstützt das Thread-Konzept
 - Falls Betriebssystem Threads nicht unterstützt, übernimmt es die JVM
 - Falls das Betriebssystem Threads unterstützt kann bei der Verwaltung direkt auf Funktionalitäten des Betriebssystems zugegriffen werden.
- Java stellt für **Umgang mit Threads** folgendes zur Verfügung:
 - java.lang.Thread-Klasse
 - java.lang.Runnable-Schnittstelle
- Auch Monitor-Konzept steht zur Verfügung (Zugriff auf gemeinsame Ressourcen sicher machen).
- Objekt **threadfähig** machen (Instanzen einer Klasse werden als Threads ausgeführt und vom Scheduler verwaltet.)
 - Generierende Klasse **implementiert die Schnittstelle Runnable**.
 - Generierende Klasse wird von der **Klasse Thread abgeleitet**.

```
import java.text.SimpleDateFormat;
public class Uhr extends Thread {
    public void run( ) {
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss:SSS");
        while (true) {
            try {
                System.out.println("Zeit: " + sdf.format(new
java.util.Date()));
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                // Ausnahmebehandlung ...
            }
        }
    }
}
```

```
public class TestClass {  
    public static void main(String[] args) {  
        // Klasse Uhr instanzieren  
        Uhr timeThread = new Uhr();  
        // Thread-Ausfuehrung starten  
        timeThread.start();  
    }  
}
```

Es kann sein, dass eine Klasse nicht von der Klasse Thread abgeleitet werden kann.

Frage: Was kann man dann tun, um die Instanzen einer solchen Klasse threadfähig zu machen werden?

Antwort: Die Klasse muss die Schnittstelle Runnable implementieren.

4.3. Zugriff auf gemeinsame Ressourcen und Synchronisation

- Solange Threads/Prozesse als Ausführungseinheiten **unabhängig voneinander** sind, gibt es **keine Probleme**
- Wenn Threads/Prozesse auf **gemeinsame Ressourcen** (Ausgabegeräte, Daten, ...) zugreifen, kann es zu **Problemen** kommen.
 - Da der Zugriff auf gemeinsame Ressourcen nicht koordiniert erfolgt, kann ein Prozess/Thread die inkonsistente Sicht der Daten erhalten.
 - Ergebnis hängt vom zeitlichen Ablauf ab.
 - Wenn ein Prozess/Thread auf gemeinsame Ressourcen zugreift und diese manipuliert nennt man das race condition (Wettkampfbedingung)
 - Der Abschnitt in dem gemeinsame Daten manipuliert werden = Kritischer Abschnitt
 - Zu einem Zeitpunkt darf sich nur ein Prozess im kritischen Bereich befinden (mutual exclusion)
 - Vor dem Eintritt muss Prozess/Thread um Erlaubnis fragen.

4.3.1. Monitorkonzept (Hochsprachenkonstrukt)

- Synchronisation mit Hilfe von **Semaphoren**
 - Wird schnell **unübersichtlich**
 - Ist **fehleranfällig** (deadlocks)
- In Java wird das **Monitorkonzept** unterstützt
- **Eigenschaften** eines Monitors
 - Kritische Abschnitte sind Methoden eines Monitors
 - Prozess betritt Monitor durch Aufruf einer Methode des Monitors
- Es kann sich immer nur ein Prozess in einem Monitor befinden. Jeder andere Prozess wird suspendiert und muss in der Queue warten, bis der Monitor frei wird.
- Monitorkonzept wird mit dem Schlüsselwort **synchronized** umgesetzt.
- **Synchronisation von Instanzmethoden**
 - Wenn eine oder mehrere Instanzmethoden mit dem Schlüsselwort **synchronized** versehen werden, wird ein Monitor um alle diese Methoden herum gebaut.
 - Die synchronisierten Instanzmethoden werden dadurch zu Methoden des Monitors
 - Es gibt ein Monitor für alle synchronisierten Methoden einer Klasse (ein Monitor pro Klasse, jede Klasse hat einen eigenen)

```
public class TestClass {  
    public static synchronized methodeX() {  
        //Kritischer Bereich  
    }  
  
    public static synchronized methodeY() {  
        //Kritischer Bereich  
    }  
}
```

Synchronisation von Klassenmethoden: Ähnlich Instanzmethoden

```
public class TestClass {  
    Public synchronized void methodeX() {  
        //Kritischer Bereich  
    }  
  
    Public synchronized void methodeY() {  
        //Kritischer Bereich  
    }  
}
```

Synchronisation von Instanzmethoden

```
public class TestClass {  
    public void doSomething() {  
        //NICHT Kritischer Bereich  
        Synchronized(this) {  
            //kritischer Abschnitt  
        }  
        //weiterer Code...  
    }  
}
```

Synchronisation von einzelnen Codeblöcken und Schlüssel

- Für das Synchronisieren eines Codeblocks wird ein Schlüssel benötigt.
- Als Schlüssel wird ein Objekt verwendet
- Zu einem bestimmten Punkt kann ein Schlüssen auf nur einen synchronisierten Block verwendet werden.
- Es gibt in einem Objekt gleich viele Schlüssel wie Monitore

• **Synchronisation und Schlüssel**

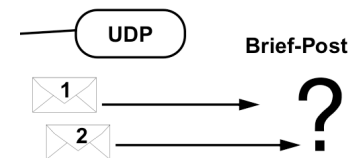
- bei synchronisierten Klassenmethoden kann als Schlüssel das Objekt der Klasse Class verwendet werden.
- bei synchronisierten Instanzmethoden kann als Schlüssel das eigene Objekt verwendet werden.
- bei synchronisierten Codeblöcken kann als Schlüssel das Objekt verwendet werden, auf das die übergebene Referenz zeigt.

5. Direkte Netzwerkprogrammierung - Sockets

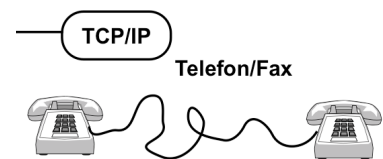
Sockets (engl. "Steckdosen") sind die **Schnittstellen** (Endpunkt) für die Kommunikation über das Netz. Es gibt zwei grundlegende **Datenübermittlungsprotokolle**, die von Sockets verwendet werden, **UDP** und **TCP**.

UDP (User Datagram Protocol) wird verwendet, um kleine Datenpakete

(max. 8KB) zu versenden. Dieses simple Protokoll **garantiert nicht**, dass ein abgesendetes Paket wirklich beim Empfänger (richtig) ankommt. Vorteile: sehr schnell und man kann damit Mitteilungen an mehrere Empfänger versenden. ⇒ Datagrammsockets



TCP (Transmission Control Protocol) ist im Gegensatz zu UDP ein **verbindungsorientiertes Protokoll** und garantiert, dass Datenpakete beliebiger Grösse beim Empfänger ankommen. TCP wird bei fast allen Client/Server Anwendungen benutzt. ⇒



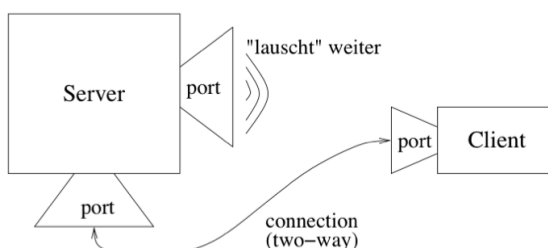
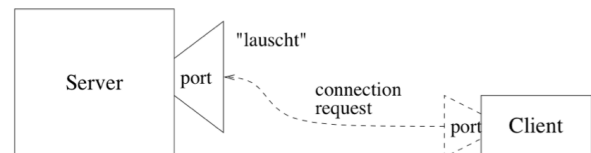
Streamsockets

Ein Socket, definiert durch die IP-Adresse einer Maschine und einer sogenannten Portnummer, stellt Verbindung mit anderem Prozess her.

Wenn ein Socket nur durch die IP-Adresse der Maschine definiert wäre, so könnte man pro Maschine nur eine Verbindung zu einer anderen Maschine erstellen.

5.1. Prinzip des Socket Aufbaus:

- Ein Serverläuft auf einem Rechner und erzeugt einen Server-Socket mit einer bestimmten Port-Nummer
- Dieser Socket "lauscht" am Port auf mögliche Anfragen von Clients (auf diesem oder einem anderen Rechner)
- Ein Client schickt eine Verbindungsanfrage zum konkreten Server-Rechner und konkretem Port
- Beim erfolgreichen Verbindungsversuch wird automatisch ein neuer Socket auf der Server-Seite erzeugt, der sich mit der damit aufgebauten Clientverbindung weiter befasst



- Die zwei neuen Sockets werden jeweils mit einem neuen Port verbunden. Die Portnummern werden vom System vergeben.
- Jetzt kommunizieren Client und Server mittels dieser zwei Sockets
- Der ursprüngliche Server-Socket "lauscht" weiter auf Verbindungsanfragen von (anderen) Clients:
 - In einem ein-thread Programm: nachdem die Verbindung geschlossen ist
 - In einem multi-threaded Programm: gleichzeitig während den laufenden Verbindungen

5.2. TCP-Sockets in Java:

5.2.1. Client:

```
/* Beim Server anklopfen <--*/
Socket socket = new Socket("10.0.0.4",5000);

System.out.println("Verbindung mit Server hergestellt!");

// Streams für Austausch
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();

// Kommunizieren
System.out.println("Sende Zahlen zum Server....");
out.write(1);
out.write(4);
out.flush();

System.out.print("\nResultat ist: ");
System.out.println(in.read());

// beenden
out.close(); in.close(); socket.close();
```

5.2.2. Server:

```
//server-socket einrichten
ServerSocket server_socket = new ServerSocket(5000);

// lauschen...

// neuer socket für den client
Socket client_socket = server_socket.accept();
String hostName = client_socket.getInetAddress().getHostName();
int p = client_socket.getPort();
System.out.println("Verbindung mit ClientNr.: " + hostName + ",
Port: " + p + "\n");

//---> // Streams für Austausch
InputStream in = client_socket.getInputStream();
OutputStream out = client_socket.getOutputStream();

// Kommunizieren
int zahl1 = in.read();
int zahl2 = in.read();
out.write(zahl1 + zahl2);
out.flush();

// beenden
out.close();
in.close();
client_socket.close();
```

5.3. Multiserver

5.3.1. Server

```
package hslu.wi; import java.io.IOException;
import java.net.ServerSocket; import java.net.Socket;

public class MultiServer {
    public static void main(String[] args) {

        try {
            ServerSocket server = new ServerSocket(5000);
            System.out.println("Server wartet auf eingehende
            Verbindung!!");

            while(true) {
                Socket client = server.accept();
                ThreadWorker thread = new ThreadWorker(client)
                thread.start();
            }

        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

5.3.2. Worker Thread:

```
package hslu.wi; import java.io.*; import java.net.Socket;

public class ThreadWorker extends Thread {
    private Socket client_socket;
    public ThreadWorker(Socket client) {
        this.client_socket = client;
    }
    public void run() {
        try {
            String hostName = client_socket.getInetAddress().
            getHostName();
            int p = client_socket.getPort();
            System.out.println("Verbindung mit ClientNr.:
            " + hostName + ", Port: " + p + "\n");
            //---> // Streams für Austausch
            InputStream in = client_socket.getInputStream();
            OutputStream out =client_socket.getOutputStream();
            // Kommunizieren
            int zahl1 = in.read(); int zahl2 = in.read();
            out.write(zahl1 + zahl2); out.flush();
            // beenden
            out.close(); in.close();
            client_socket.close();

            System.out.println("Verbindung zum Client wurde
            beendet!!");
        } catch (IOException e) { e.printStackTrace(); }
    }
}
```

6. Interprozesskommunikation

6.1. Einführung

- Eine Applikation besteht aus einem oder mehreren Prozessen.
- **Prozesse**
 - sind Objekte des Betriebssystems
 - ermöglichen einer Anwendung den sicheren Zugriff auf die Ressourcen des Betriebssystems
 - laufen in eigenen Speicherbereichen und sind dadurch voneinander isoliert
- **Für den Austausch von Daten zwischen Prozessen ist eine Interprozesskommunikation nötig.**
- IPC basiert auf dem **Austausch von Nachrichten** und Daten zwischen Prozessen:
 - ein Prozess sendet die Nachricht bzw. die Daten (Sender)
 - ein anderer Prozess empfängt die Nachricht bzw. die Daten (Empfänger)
- Umsetzungsmöglichkeiten von **IPC**
 - Über **Dateien**
 - Über **Pipes** (nur Lokal)
 - Über **Sockets** (Lokal/Entfernt)
 - Nur connect, read, disconnect Operationen
 - Man muss mit Streams arbeiten
 - **Middleware**

6.2. Kommunikationsmodelle

Legt das Protokoll für den Ablauf der Kommunikation zwischen Prozessen (IPC) fest.
Es gibt zwei Kommunikationsarten.

6.2.1. Synchrone Kommunikation

- Sender Prozess muss nach dem Senden solange warten, bis die Antwort eintrifft.
- Sehr verbreitet
- **Vorteile:**
 - Einfache Implementierung
 - Synchronisation beim Zugriff auf kritische Bereiche sofort erledigt
- **Nachteile:**
 - Ineffizient (durch Warten)
 - Sichere/Schnelle Netzwerkverbindung notwendig
 - Empfängerprozess muss verfügbar sein

6.2.2. Asynchrone Kommunikation

- Sender-Prozess kann nach dem Senden sofort weiterarbeiten, ohne auf die Antwort zu warten.
- Antwortmöglichkeiten:
 - Empfänger wird aktiv und antwortet bei Gelegenheit asynchron
 - Sender holt Antwort bei Gelegenheit selbst ab
- Realisierung mittels Warteschlangen
- **Vorteile:**
 - Lose Koppelung von Prozessen

- Empfänger muss nicht Empfangsbereit sein (Warteschlange)
- Effizienter (kein Warten auf Antwort)
- **Nachteile:**
 - Aufwändig, komplizierte Implementierung
 - Komplizierte Protokolle

6.3. Datenpräsentation

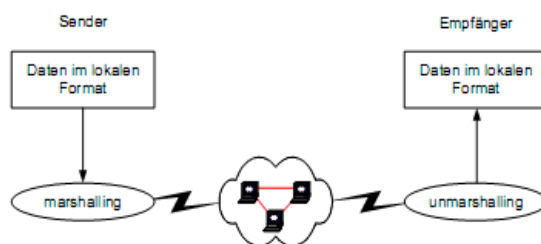
- Verteilte Systeme laufen normalerweise in Heterogenen Umgebungen (Unterschiedliche Hardware, Plattformen, Programmiersprachen)
- Daten müssen deshalb immer dasselbe Format haben.
 - Daten müssen deshalb immer zuerst vom lokalen Format ins allgemeine umgewandelt werden, und umgekehrt.
- Daten können im Speicher verteilt sein (können so nicht übertragen werden)

R	O	L	A	N	D	'\n'	•	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	M	E	I	E	R	'\n'	•	x	x	x	x	x	x
x	x	1	9	4	9	'\n'	x	x	x	x	x	x	x	x	x	x	x	x	x

- **Marshalling** (Transformation einer beliebigen Nachricht in eine übertragbare)
 - Immer auf der Senderseite
 - Daten werden vom Lokalen in das gemeinsame Format übersetzt
 - Datenstruktur wird in eine zusammenhängende Nachricht „planirt“
 - Resultat = bytestrom (Im Speicher hintereinander, zusammenhängend)

H	A	N	S	'\n'	P	O	R	T	M	A	N	N	'\n'	1	9	3	8	'\n'
---	---	---	---	------	---	---	---	---	---	---	---	---	------	---	---	---	---	------

- **Unmarshalling** (Übersetzen vom gemeinsamen in das Lokale Format)
 - Immer auf der Empfängerseite



6.4. Realisierung der IPC

6.4.1. Direkte Netzwerkprogrammierung mit Sockets

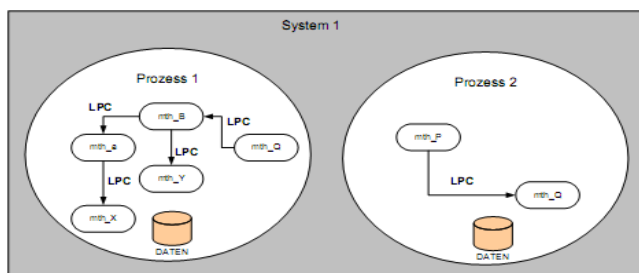
- Läuft auf der Transportschicht (TCP/UDP)
- **Vorteile:**
 - Sehr flexibel bei Entwicklung neuer Protokolle
 - Bessere Performance
- **Nachteile:**
 - Nicht sehr komfortabel
 - Die Datenpräsentation

6.4.2. Netzwerkprogrammierung mit Middleware

- Zusätzliche Schicht zwischen Transportschicht und Anwendung
- **Vorteile:**
 - Bequeme Entwicklung von Applikationen
 - Lösung des Datenpräsentationsproblems
- **Nachteile:**
 - Schlechte Performance
 - Grosser Overhead

6.5. LPC (Local Procedure Call)

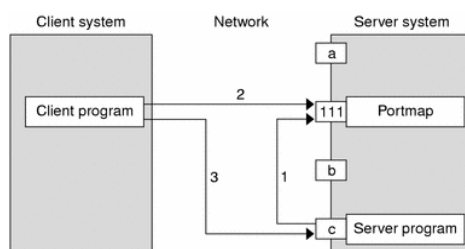
- Aufruf einer Prozedur, welche sich im gleichen Speicherraum (Prozess) befindet
- **Interprozesskommunikation (IPC) ist mit LPC somit nicht möglich.**
- **Vorteile:**
 - Gute Performance (da alle Prozeduren im gleichen Prozess)
 - Einfache Parameterübergabe (lokaler Stack, gemeinsam verfügbar)
 - Einfacher Zugriff auf gemeinsame Daten (da alle Prozessdaten gemeinsam)
- **Nachteil:**
 - Keine IPC möglich!



6.6. RPC (Remote Procedure Call)

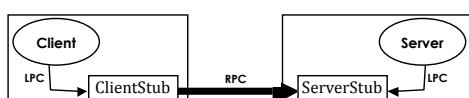
- Entfernter Aufruf von Prozeduren
- Entfernt = In anderem Prozess (anderer Speicherraum)
- **Ziel: Benutzer meint es sei eine lokale Prozedur.**
- Prozesse können auf gleichen oder unterschiedlichen Systemen ausgeführt werden.
- **ClientStub** Implementiert die Server-Schnittstelle, so kann der Client alle Methoden des Servers auf dem ClientStub aufrufen
- **Ablauf auf Clientseite**
 - ClientStub nimmt Aufruf der Server-Methode entgegen, verpackt sie (Marshalling) und sendet Sie zum Server (bzw. zum ServerStub)
 - ClientStub nimmt die Antwort des Servers entgegen, entpackt sie (Unmarshalling) und liefert den Rückgabewert an den Client zurück.
- ServerStub muss alle Prozeduren implementieren, die in der Schnittstelle definiert wurden.
- ServerStub muss eine Referenz auf den Server haben, um seine Prozeduren aufrufen zu können.
- **Ablauf Serverseite**
 - ServerStub nimmt Anfrage vom Client entgegen entpackt diese (Unmarshalling)
 - Ruft gewünschte Methode auf dem Server auf (LPC)
 - Nimmt die Antwort des Servers entgegen

- Verpackt die Antwort (Marshalling)
- Sendet Sie an den Client (ClientStub) zurück
- **Probleme** mit RPC
 - RPC wird von manchen Programmiersprachen (Java, C, C++) nicht von Haus aus unterstützt.
 - Zusätzlicher Compiler zur Generierung von Stubs (Precompiler)
 - Vom Server realisierte Schnittstelle muss als Basis für die Generierung von Stubs zur Verfügung gestellt werden.
 - IDL (Interface Definition Language) zur Definition von Schnittstellen
- **Parameterübergabe mit RPC**
 - Übergabe von **Werten** (pass by value)
 - Wert wird in die Nachricht kopiert
 - Auf der anderen Seite rekonstruiert
 - Übergabe von **Referenzen** (pass by reference)
 - Grundsätzlich nicht möglich
 - Ausnahme: wenn gemeinsamer Speicher vorhanden
- Datenpräsentation mit RPC
 - Probleme mit heterogenen Umgebung (unterschiedliche Hardware, ...)
 - Lösung: gemeinsames Format für Daten
- **RPC und Binding** (Wie wird das Host-System gefunden?)
 - Zuerst nachfragen, ob der gewünschte Service verfügbar ist und wie er angesprochen wird (Portnummer)
 - **Portnummer** darf nicht vorgegeben sein
 - Client erfährt die entsprechende Portnummer mittels dem Binding-Dienst (bsp.: portmapper bei SUN). **Portnummer** des Binders muss bekannt sein
 - Aufgaben eines Binders:
 1. Anmelden von verfügbaren Services (register)
 2. Anfrage nach Portnummer des Services beantworten (lookup)
 3. Abmeldung von Services (unregister)



6.7.LPC statt RPC

- Dem Client wird eine Komponente zur Verfügung gestellt, die im gleichen Prozess ausgeführt wird und aussieht wie der Server. (**ClientStub**)
- Dem Server wird eine Komponente zur Verfügung gestellt, die im gleichen Prozess ausgeführt wird, und aussieht wie der Client (**ServerStub**)
- Der wirkliche Client/Server kommuniziert via LPC mit dem Stub, die beiden Stubs kommunizieren via RPC miteinander, ohne dass der Server/Client etwas davon merkt
- **Der Client/Server bemerkt nicht, dass die beiden Stubs mit RPC kommunizieren.**



7. Der Entwicklungsprozess von Komponenten

Komplexe Software ist nur schwer zu erstellen, darum bedient man sich **Vorgehensmodellen** welche den Entwicklungsprozess in überschaubare Phasen einteilt. Es handelt sich um eine **abstrakte Darstellung von Prozessen**. Die Software wird somit Schritt für Schritt fertiggestellt.

Anforderungen an Vorgehensmodelle

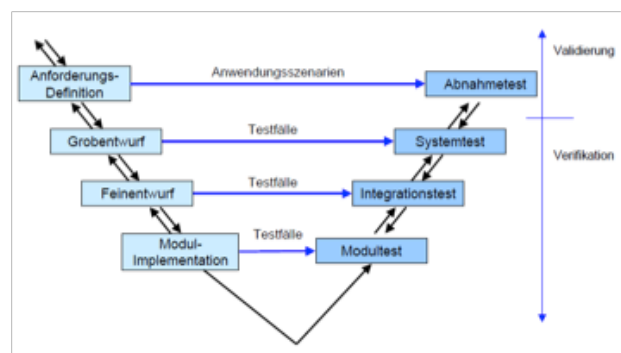
- Ergebnisorientiert
- Flexibilität
- Vollständigkeit
- Einbezug der Dokumentation
- Organisatorische Einbettung ins Unternehmen
- Unterstützung der Datenverarbeitungssysteme

7.1. Wasserfallmodell

- Das **erste** veröffentlichte **Modell** (1970)
- Alle **Aktivitäten** sind streng **hintereinander** auszuführen
- Es wird auf das Ergebnis der vorhergehenden Aktivität aufgebaut
- **Vorteile**
 - Einfach zu verstehen und übersichtlich
 - Bei stabilen Anforderungen sehr effektives Modell
- **Nachteile**
 - Fehler werden immer grösser, komplexer, Zeitaufwand um Fehler im nächsten Schritt zu beheben wird immer grösser.
 - Verpflichtungen werden früh eingegangen, man entwickelt evt. an den Anforderungen vorbei
 - Anwender sehen erst spät ein Ergebnis

7.2. V-Modell

- Entwicklung des deutschen Bundes, verbindlich für alle IT Vorhaben der Bundesbehörde
- Es deckt **Softwareentwicklung, Projektmanagement, Qualitätssicherung** und **Konfigurationsmanagement** ab.
- Beschreibt **Vorgehensweise, Methoden** und **Werkzeug**
- Weiterentwicklung des Wasserfallmodells



Auf dem absteigendem Ast befinden sich konstruktive Tätigkeiten, auf dem aufsteigenden Ast, Arbeitsschritte zur Qualitätssicherung. Beispiel: Treten in der Phase der Modultests Fehler auf, ist der Rückschritt zu „Modul-Implementation“ gering. Stellt man jedoch bei den Abnahme Tests fest, dass die Systemanalyse unzureichend war, könnte es zum Projekt Abbruch führen.

7.3. Spiralmodell

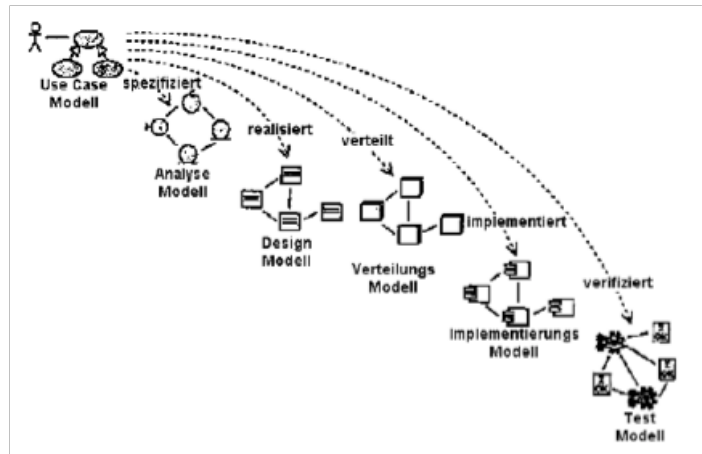
Teilt sich in **einzelne Quadranten: Ziele, Alternativen** (Risikoanalyse), **Entwicklung und Tests, Planung** des nächsten Zyklus. Es ist ein inkrementelles oder iteratives Vorgehensmodell, das sich auf die Risiken konzentriert.

7.4. Wachstumsmodell (Iterativ-Inkrementell)

- Grundidee: Gliederung der zu erstellenden Software in aufeinander aufbauende betriebsfähige Schritte.
- Entwicklung wird in eine Folge von Iterationen unterteilt
- Pro Iteration: Vollständiges Teilergebnis mit betriebsfähiger Software.
- **Vorteile**
 - Frühe Entstehung lauffähiger Teile
 - verkürzt die Rückkoppelungszyklen
 - schrittweise einführbar
- **Nachteile**
 - Gefahr der Zerstörung von Struktur durch schrittweisen Ausbau

7.4.1. Unified Process (UP)

- Ist ein populäres iteratives und inkrementelle Software Entwicklungsprozess Framework.
- Frühzeitige Qualitätskontrolle, jede Iteration wird sofort getestet.
- **Folge von Zyklen: Inception** (Startphase), **Elaboration** (Entwurfsphase), **Construction** (Konstruktionsphase), **Transition** (Übergangsphase).
- Nach jedem Zyklus: Fertiges Produkt-Release aus Code, Handbücher, UML Modellen, Testfällen

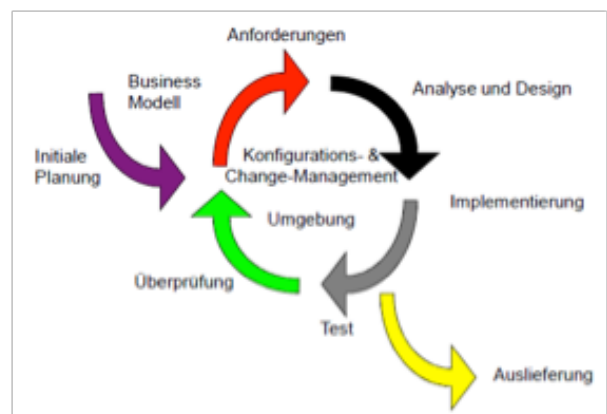


Unified Process hat 4 Grundpfeiler

- **Anwendungsfall (use-case)-getrieben:** Es besteht bei allem, immer eine Rückkoppelung auf das Use Case Modell, und hilft so Benutzer und Anforderungen, Systemarchitektur, Testfälle etc. zu definieren. Für use-case Szenarien können Anwendungsfallbeschreibungen oder Diagramme erstellt werden welche die Szenarien beschreiben.

Definition von „use case“: Bündelt alle möglichen Szenarien die eintreten können, wenn ein Akteur versucht mit Hilfe des betrachteten Systems ein bestimmtes Ziel zu erreichen.

- **Architektur zentriert:** Zuerst ein grober anwendungsunabhängiger Architekturentwurf (Architekturmuster z.B. Client-Server, Schichten, Middleware, DBMS etc.)



- **Iterativ-inkrementell:** Man startet mit einem kleinen Teilbereich der Software und mit jeder Iteration werden neue Änderungen und Funktionen der Software hinzugefügt.
- **Risiko-sensitiv:** Risikoreiche Aspekte der Entwicklung werden sehr früh angegangen, Architektur wird auf diese Risiko Faktoren ausgerichtet. ⇒ Verminderung des Risikos

7.5. Prototyp-orientiertes Modell

- Frühzeitige Erstellung ablauffähiger Modelle (Prototypen) des zukünftigen Systems.
- Dienst der Realisierbarkeit von Anforderungen.
- Prototypen klären Entwicklungsprobleme, sind Diskussionsbasis, zur Sammlung von Erfahrungen

7.6. Agile Software Entwicklung

Agile Softwareentwicklung versucht mit **geringem bürokratischen Aufwand** und wenigen Regeln auszukommen. Es ist eine **Gegenbewegung** zu schwergewichtigen Softwareentwicklungsprozessen wie **UP** oder **V-Modell**.

7.6.1. Extreme Programming (XP)

Idee: Verfahren für die Entwicklung von Kleinsoftware auf grosse Software übertragen.

Neue Funktionalitäten werden permanent entwickelt, integriert und getestet.

Prinzipien: Schnelles Feedback, Annahme von Einfachheit (Komplexität kann später ergänzt werden), weniger Bürokratie ⇒ Zeitersparnis.

Eignung: Kleine Projekte, für bestimmte Arten von SW Projekten.

7.6.2. SCRUM

- Das SCRUM Team ist klein, jeder hat eine klare Rolle, agiert eng im Verbund mit dem Ziel.
- Scrum hat verschiedene Rollen:
 - **ScrumMaster:** Verantwortlich für Erfolg, Verbindungsglied zwischen ⇒ Team ↔ Management
 - **ProductOwner:** Bestimmt den Entwicklungsablauf durch Priorisierung der Produkt-Backlogs.
 - **Team:** Cross-Functional team, ca. 7 Personen die die Analyse, Design, Implementierung umsetzen
- **Product Backlog:** Ist die priorisierte Anforderungsliste für das Produkt
- **Sprint:** Eine Entwicklungsperiode (Sprint) wird auf 30 Tage terminiert. Er beinhaltet Anforderungsanalyse, Design, Programmierung, Integration, Tests, Dokumentation.
- **Ende des Sprints:** Entwickelte Funktionen präsentieren (echte Funktionalität!)
- **Sprint Planning:** Nach dem Sprint bestimmt der Product Owner, welche der nächsten Backlog items zu erledigen sind.
- **Täglicher Scrum:** Tägliches 15 min. Statusmeeting, Ziel ist die Synchronisation.
- **Sprint Review:** Das Team präsentiert dem Management, Kunden, Benutzern, was im letzten Produktinkrement erstellt wurde.

8. Grundlage XML als Grundlagenstandard

8.1. XML

- XML „**Extensible Markup Language**“ ist eine Metasprache. Es lassen sich für jeden beliebigen Zweck Dokumenttypen schaffen. (z.B. <Entsafter>)
- **Funktional gesehen** ist XML ein allgegenwärtiges, plattformneutrales **Transportformat**.
- **Strukturell gesehen** ist XML eine **Syntax für eine Markup Language** (z.B. HTML) mit namentlichen Elementen und Attributen mit rein textuellen Informationen. Tags strukturieren die Daten.
- Notwendig wurde XML, da heutzutage viele Prozesse Daten austauschen, und verschiedene Zeichensatztypen (Heute UTF-8/16/32) und verschiedene Datenstruktur verwenden, was zu Problemen beim Austausch führt.
- Im Gegensatz zu HTML sind bei XML die **Grammatik** (DTD, Dokumenttypdefinition), der **Inhalt** (XML) und die **Darstellung** (XSL) **getrennt**.
- XML ist keine Programmiersprache, ein XML Dokument existiert einfach, es tut nichts.

8.2. Lebenszyklus eines XML Dokuments

Der Marshaller erhält Informationen, und verpackt diese Informationen in Tags. Der Unmarshaller prüft die Wohlgeformtheit (syntaktische Korrektheit) und entpackt die Informationen aus den Tags. (z.B. ein Webbrowser)



8.3. XML Beispiel

- Die Tags bestimmen den Inhalt, Sie sind unbeschränkt verfügbar, und frei definierbar
- Wohlgeformtheit, wenn alle XML Regeln enthält

```

<?xml version="1.0"? encoding="UTF-8" standalone="yes"?>
<Entsafterliste>
  <Entsafter id="7002.702" leistung="60" hersteller="TRISA">
    <Name>Juicer</Name>
    <Bild>bilder\TR-7002.702.jpg</Bild>
    <Beschreibung>Frische Obstsäfte selbst schnell zubereiten!</Beschreibung>
    <Garantie>24</Garantie>
    <Gewicht></Gewicht>
    <Preis währung="CHF">23.00</Preis>
    <Verkäufer>http://www.trisaelectro.ch</Verkäufer>
  </Entsafter>
  <Entsafter id="TX-A34104" leistung="100" hersteller="TURMIX">
    <Name>Zitruspresse CX 630</Name>
  </Entsafter>
</Entsafterliste>
  
```

- **Legende**

<?xml version="..."	Die erste Zeile wird Prolog genannt
standalone="yes"	Gibt an ob DTD Datei existiert oder nicht
hersteller="TRISA"	Hersteller ist das Attribut, „TRISA“ ist der Attributwert
<Entsafter>	Starttag
</Entsafter>	Endtag

8.4. Syntaktisches

- Das **XML Dokument** besitzt genau **ein Wurzelement**. Als Wurzelement wird dabei das jeweils äußerste Element bezeichnet, z.B. <html> in XHTML.
- Alle Elemente mit Inhalt besitzen eine **Beginn-** und einen **End-Tag**
- **Attribute** befinden sich **im Starttag** eines Elementes: <Entsafter id="7002.702"...>

8.5. Zeichensatz

- **XML** verwendet **standardmässig** Unicode **UTF-8**, andere müssen im Prolog deklariert werden
- Elementnamen beginnen mit einem Buchstaben, einem Underscore oder Doppelpunkt
- Verboten sind { „ , \$ () [] % ; } sowie die Zeichenfolge xml

8.6. Namensraumkonflikte mit Namespaces beheben

- XML erlaubt Elementnamen zu definieren
- Da XML mit anderen Anwendungen kombiniert werden kann, können Namenskonflikte entstehen.
Ein Beispiel: In XHTML beschreibt <p>-Element einen Absatz, in einer XML-Sprache für eine Personendatenbank könnte <p> ein Element für eine Person darstellen.
- **Namensräume ordnen** Element- und **Attributnamen einer** eindeutigen **Quelle zu**. (Es ist wie eine Vorwahl bei der Telefonnummer).

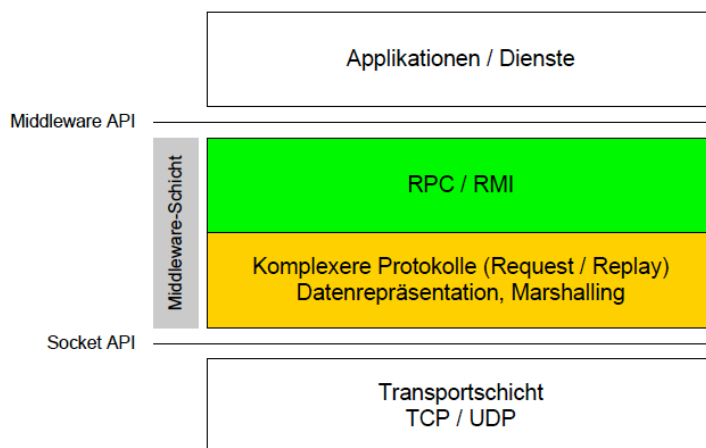
8.6.1. Beispiel zu Namensraumkonflikten (xmlns = XML name space)

Eine typische Bestellung greift auf **zwei** verschiedene **Datenbestände** zu: **Produktdatenbestand** und **Kundendatenbestand**. Es gibt **zwei Namensraumdeklarationen**: **Produkt** und **Kunde**. Zwischen <bestellung> und </bestellung> soll mit Elementen aus beiden Namensräumen gearbeitet werden können. Dazu werden Namensraumdeklarationen **Präfixe** definiert: xmlns:produkt und xmlns:kunde, und jeweils eine Konvention vergebene URI (URI kann fiktiv sein). Somit nimmt produkt:nummer immer Bezug auf den Namensraum xmlns:produkt und kunde:nummer auf xmlns:kunde.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<bestellung xmlns:produkt="http://localhost/XML/produkt" xmlns:kunde="http://localhost/XML/kunde">
  <produkt:nummer>p49393</produkt:nummer>
  <produkt:name>Rasierer VC100</produkt:name>
  <produkt:menge>1</produkt:menge>
  <produkt:preis>69.00</produkt:preis>
  <kunde:nummer>1293</kunde:nummer>
  <kunde:name>Meier, Fritz</kunde:name>
  <kunde:lieferadresse>Donnerstr. 5, 6000 Luzern</kunde:lieferadresse>
</bestellung>
```

9. Middleware & RMI als konkrete Implementierung

- Verteilte Anwendung nutzt verteiltes System als Kommunikationsinfrastruktur
- Das verteilte System bietet nur Auf- und Abbau der Verbindung und Datenübertragung als Byte-Pakete an.
- Für komplexe Aufgaben muss auf einer höheren Ebene gelöst werden -> in der Anwendung selbst oder in einer zusätzlichen Software-Schicht: **Middleware**
- Aufgaben: Interaktion zwischen Anwendungskomponenten zu erleichtern und Komplexität der vernetzten Systemumgebung zu maskieren.



Middleware: Software zwischen Betriebssystem und verteilten Anwendungen

9.1. Arten von Middleware

9.1.1. Kommunikationsorientierte Middleware (KOM)

- Konzentriert sich auf Bereitstellung einer geeigneten Kommunikationsinfrastruktur für Komponenten einer verteilten Anwendung.
- Aufgaben: Kommunikation, Marshalling/Unmarshalling, Fehlerbehandlung
- 3 Programmiermodelle: Entfernte Prozeduraufrufe (RPC), entfernte Methodenaufrufe (RMI) und das nachrichtenorientierte Modell (MOM)
- RPC, Java RMI, Web Services

9.1.2. Anwendungsorientierte Middleware (AOM)

- Erweiterung der KOM, neben Kommunikation auch zusätzliche Dienste (Unterstützung)
- Betriebssystem als Laufzeitumgebung ist für die Anforderungen der verteilten Anwendungen nicht geeignet. Laufzeitumgebung der Middleware baut jedoch auf den Funktionen des Betriebssystems auf und erweitert diese. Wichtigste

9.1.2.1. Wichtigste Aufgaben der Laufzeitumgebung:

Ressourcenverwaltung

Betriebssystem verwaltet grundlegende Ressourcen wie Speicher, Prozesse, Threads etc., kann jedoch Ressourcen auf Ebene der Middleware nicht optimal verwalten. Deshalb muss die Middleware das selber übernehmen -> Getrennte Speicherbereiche anlegen, Verbindungsobjekte und Threads auf Vorrat anlegen um sie bei Bedarf schnell einsetzen zu können. Ziel: Verbesserung Performance, Skalierbarkeit und Verfügbarkeit der Anwendungen.

Nebenläufigkeit

I.d.R. werden Anwendungen von mehreren Anwendern benutzt. Deshalb werden die Aufrufe isoliert in separaten, nebenläufigen Threads oder Prozessen abgearbeitet.

Verbindungsverwaltung

Fixe Zuteilung von Verbindungen zu Prozessen ist wenig sinnvoll. Deshalb wird ein Vorrat (Pool) an Verbindungen angelegt, falls benötigt von dort rausgenommen und falls überflüssig wieder zurückgelegt.

Sicherheit

Daten werden über unsichere Strecken übertragen. Das Sicherheitsmodell unterstützt mindestens die Zugriffskontrolle (Authentifizierung - Sicherstellung der Identität des Benutzers) und die Vergabe von Zugriffsrechten (Autorisierung - Benutzerrechte an Benutzer vergeben).

9.1.2.2. Die wichtigsten Dienste der AOM**Namensdienst**

Ermöglicht, dass Ressource in einer bestimmten Umgebung (Intra- oder Internet) veröffentlicht wird und von interessierten Clients durch einen eindeutigen Namen gefunden werden kann.

Sitzungsverwaltung

Teilt jedem Anwender eine Sitzung zu, welche eine bestimmte Zeit gültig ist und für den Anwender relevante Daten verwaltet. Besonders wichtig für interaktive verteilte Anwendungen.

Transaktionsverwaltung

Durch parallele Zugriffe auf Daten könnten zwei Anwender denselben Datensatz gleichzeitig ändern. Dies wird durch die Verwendung von Transaktionen verhindert. Mit einer Transaktion wird ACID sichergestellt (Atomarität, Konsistenz, Isolation, Dauerhaftigkeit)

Persistenz

Mit Persistenz wird die Gesamtheit aller Mechanismen zur dauerhaften Speicherung von flüchtigen Daten im Hauptspeicher auf ein persistentes Speichermedium bezeichnet. In der Praxis hat sich vor allem eine Art des Persistenzdienstes etabliert: Objektrationale Mapper (OR Mapper)

9.1.3. Nachrichtenorientierte Middleware (NOM) -> wird im Skript nicht erwähnt

- Arbeitet nicht mit Methoden- oder Funktionsaufrufen sondern über den Austausch von Nachrichten.
- Kann Synchron und Asynchron arbeiten.
- Ermöglicht die Übertragung von Nachrichten, deren Format nicht festgelegt (vorgeschrieben) ist.
- Kopplung zwischen einzelnen Teilsystemen wird durch die Verwendung von MOM reduziert.

Vorteile

- Lose Kopplung zwischen Kommunikationspartner, System wird stabiler, bessere Skalierbarkeit
- Belastung einzelner Komponenten kann besser abgefedert werden (Load Balancing)

Nachteil

- Der Ausfall der MOM führt zum Ausfall des ganzen Systems

9.2. Technologien von Middleware**9.2.1. Object Request Broker (ORB)**

- Setzt auf dem Programmiermodell der entfernten Methodenaufrufen auf
- Bietet die Kommunikationsinfrastruktur für die Verwaltung von verteilten Objekten
- Stellt auch zusätzliche Dienste zur Verfügung, die von der verteilten Anwendung genutzt werden können.

- Beispiel: Common Object Request Broker Architecture (CORBA)

9.2.2. Application Server (AS)

- Werden jeweils für einen bestimmten Typ von Anwendungen entwickelt (JEE, .NET etc.)
- Stellen die Kommunikationsinfrastruktur, Laufzeitumgebung und diverse Dienste zur Verfügung.

9.2.3. Middleware-Plattformen

- Erweitert einen AS zu einer vollständigen Plattform für verteilte Anwendungen indem sie diese durch alle Schichten unterstützen.
- JEE mit dem EJB-Komponentenmodell und .NET mit COM-Komponentenmodell
- Beispiel: **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA)

10. Remote Methode Invocation (RMI) - Aufruf entfernter Methoden

- Mechanismus in Java, mit welchem entfernte Objekte bzw. deren Angebote genutzt werden können.
- **Entfernt:** Objekt kann in einer anderen Java Virtual Machine sein, lokal oder auf einem entfernten Rechner
- Ähnlich wie lokaler Aufruf, es müssen aber zusätzlich spezielle Ausnahmen (Verbindungsabbruch) abgefangen werden.
- Einfaches Framework für die Entwicklung von verteilten Anwendungen in Java
- Nur Java to Java möglich

Vorteile

- Details der Netzwerkkommunikation werden „ausgeblendet“
- Verteilung von Objekten durch einen Namensdienst (RMI-Registry)
- Entfernte Objekte sind „multithreaded“

Nachteile

- Basiert auf OO-Konzepten von Java (nur Java to Java möglich)
- Namensdienst kann für manche Anwendungen evtl. nicht genügen.
- Sync bei konkurrierenden Zugriffen wird nicht von RMI realisiert und liegt in der Verantwortung des Benutzers

10.1. Kommunikation

10.1.1. Client

- Nimmt Dienste vom entfernten Objekt in Anspruch
- Fragt beim Namensdienst (vergleiche Telefonauskunft) nach, ob ein passendes Objekt für den gewünschten Dienst registriert wurde
- Ruft Methoden des Entfernten Objekts auf

10.1.2. Server

- Erzeugt das entfernte Objekt und meldet es beim Namensdienst an (Registrierung)
- Meldet es wieder ab, wenn es nicht mehr gebraucht wird. (De-Registrierung)

10.1.3. Entferntes Objekt

- Wird vom Server erzeugt und beim Namensdienst registriert.
- Läuft auf dem Server-Knoten
- Stellt bestimmte Dienste zur Verfügung

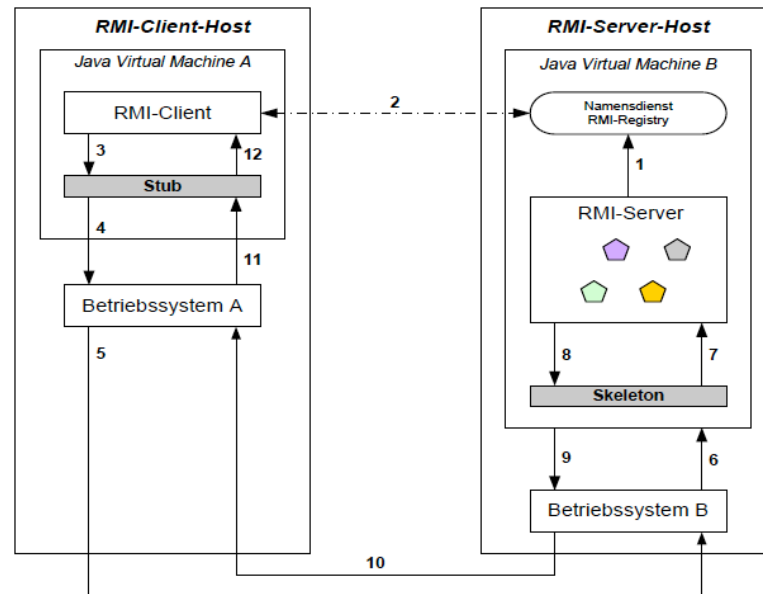
10.1.4. Stub und Sekeleton

- Werden durch `rmic` aus der Klasse erzeugt, welche das entfernte Objekt implementiert.
- Kümmern sich um die Übertragung von Daten über das Netzwerk
- Stub -> Clientseite
- Skeleton -> Serverseite

10.1.5. Namensdienst

- Aufgaben: Registrieren (binding) und Finden (lookup) des entfernten Objekts ermöglichen.
- Wird auf dem Host-System (Server) ausgeführt.
- Vergleiche mit Telefonauskunft („Achtzehn achtzeeheeen... achtzehn achtzeeeeehn...“)

10.1.6.Ablauf



1. RMI-Server erzeugt entferntes Objekt und Registriert es beim Namensdienst
2. RMI-Client fragt beim Namensdienst, welches EO für den gewünschten Dienst zur Verfügung steht.
3. Stub nimmt anfrage des Clients entgegen.
4. Marshalling durch Stub
5. OS überträgt die Daten über das Netzwerk zum entfernten Host.
6. Entferntes OS nimmt Anfrage entgegen und leitet sie an Skeleton weiter.
7. Unmarshalling durch Skeleton Objekt
8. Server erbringt den geforderten Dienst und liefert das Ergebnis an den Aufrufer (Methode aus dem Skeleton Objekt) zurück
9. Skeleton nimmt Antwort entgegen und verpackt Antwort und überträgt sie ans OS
10. Entferntes OS sendet Antwort an lokales OS
11. Lokales OS nimmt Antwort entgegen und leitet es an Stub weiter
12. Client nimmt Antwort entgegen und kann weiterarbeiten. Falls Ausnahme durch Server geworfen wurde, muss der Client entsprechend Reagieren.

10.2. Implementierung

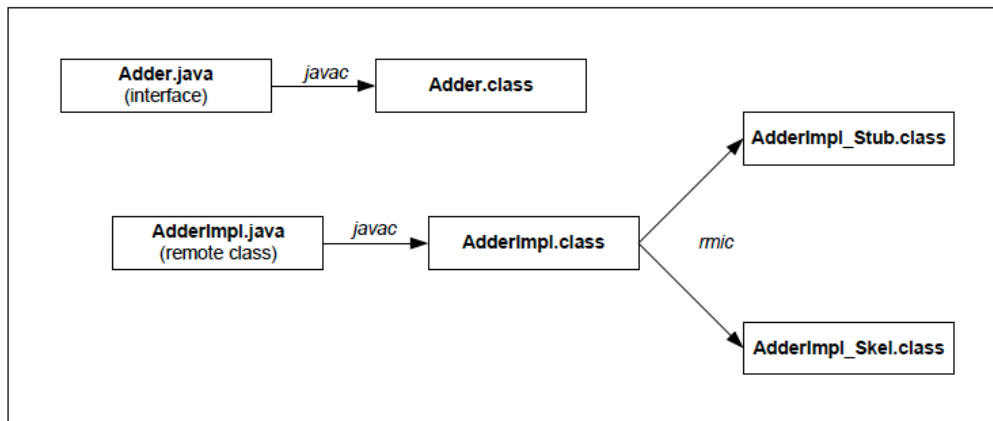
10.2.1.Definition der Schnittstelle

- Methoden eines EO werden in einer passenden Schnittstelle definiert
- Diese Schnittstelle muss von *java.rmi.Remote* abgeleitet werden
- Alle Methoden der Schnittstelle können *java.rmi.RemoteException* werfen und müssen diese deklarieren.

10.2.2.Definition der entfernten Klasse

- Abgeleitet von *java.rmi.UnicastRemoteObject*
- Muss Standardkonstruktor zur Verfügung stellen, welcher die Ausnahme *java.rmi.RemoteException* deklarieren muss.
- Aus der entfernten Klasse werden mithilfe des *rmic*-Tools (RMI Compiler) Stub und Skelton generiert.
- Name der entfernten Klasse = Name der entfernten Schnittstelle + *Impl*

10.2.3. Kompilierung der entfernten Klasse



10.2.4. Anmeldung des entfernten Objekts beim Namensdienst

`start rmiregistry [port]` \Leftarrow freiwillig, Standardport bei nichtangabe = 1099

10.3. Java-Security und Policy-Datei

- In einem verteilten System wird oft der Code von einem anderen Server (fremder Code) geladen und auf dem lokalen System ausgeführt. Dies ist gefährlich, da der Code durch einen Angreifer durch seinen, bösartigen Code, ausgetauscht werden kann.
- Das Downloaden und Ausführen des fremden Codes wird vom RMISecurityManager laufend überwacht.
- RMISecurityManager wurde beim Client „installiert“ und braucht noch klare Anweisungen, welche Aktionen vom fremden Code ausgeführt werden dürfen und welche nicht. Diese Anweisungen werden in einer policy-Datei festgehalten.

10.4. Verteilung von Klassen

Nach der Erstellung und kompilierung der Klassen, müssen diese verteilt werden.

Serverseite

- Entfernte Schnittstelle (Adder.class)
- Entfernte Klasse (AdderImpl.class)
- Server-Klasse welche Methode main enthält

Clientseite

- Entfernte Schnittstelle (Adder.class)
- Client-Klasse welche Methode main enthält
- adder.policy im ROOT-Verzeichnis

11. Datenbanken in Verteilten Anwendungen

11.1. JDBC

11.1.1. Einleitung

Unterschiedliche DBMS haben unterschiedliche Zugriffsprotokolle.

Reimplementierung des Codes nötig bei direkten DBMS zugriff falls ein anderes DBMS verwendet werden will.

Lösung dazu: **ODBC** und **JDBC** dienen als Übersetzerschnittstellen.

ODBC (Open Database Connectivity) von Microsoft ist eine allgemeine Datenbank-Zugriffsschnittstelle.

JDBC (Java Database Connectivity) seit JDK 1.1 Bestandteil des Standard-API. Es gilt als stabiler und robuster als ODBC

11.1.2. JDBC Komponenten

- JDBC Treibermanager (unterschiedliche DBMS Treiber und für Verbindung zwischen einem Java-Programm und DB zuständig)
- JDBC Treiber Test Suite (Treiber test (Zertifizierung))
- JDBC-ODBC-Bridge (Verwendung von ODBC Treibern)
- JDBC API Programmierschnittstelle mit java.sql und javax.sql

11.1.3. JDBC Treibertypen

- **Typ1: JDBC-ODBC Bridge Treiber**
Übergangslösungen, Enthalten Binärcode, Nicht für Web-Anwendungen
- **Typ2: Native-API-Treiber**
Übergangslösungen, Enthalten Binärcode, Nicht für Web-Anwendungen
- **Typ3: JDBC-Net-Treiber**
In Java geschrieben, geeignet für Web-Anwendungen
- **Typ4: Nativen-Protokoll-Treiber**
i.d.R. von DB-Hersteller geschrieben, sehr effizient

11.2. Ablauf eines Zugriffs auf DB

11.2.1. Allgemeiner Ablauf

1. Verbindung erstellen zur Datenbank
2. Erstellung der SQL-Anweisung
3. Senden der Anweisungsfolge zum DB-Server und die Ausführung
4. Verarbeitung der Ergebnisse
5. Schliessen der Ergebnisse

11.2.2. Laden eines JDBC Treibers

- Alle in der jdbc.drivers- Systemeigenschaft angegebenen Treiber werden vom Treibermanager beim Starten automatisch geladen
- Explizites Laden mit der Methode forName der Klasse Class, durch die Erstellung einer Instanz der Klasse Driver
- Beispiel für MySQL Treiber:
 - `Class.forName(„com.mysql.jdbc.Driver“);`

11.2.3.Verbindung aufbauen

```
String url = "jdbc:postgresql://147.88.100.100:5432/raum_db";
String user = "student";
String pwd = "geheim";
// Aufbau der Verbindung
Connection con = DriverManager.getConnection(url, user, pwd);
```

11.2.4.Statement kreieren

Mit einem erzeugten Statement-Objekt (Statement stm = con.createStatement());
Können (**SELECT**, **INSERT**, **DELETE**, **UPDATE**) durchgeführt werden.

11.2.5.Schreibender Zugriff

Execute Update liefert int-Wert Zurück mit Anzahl bearbeiteter Tupeln

```
int anz = 0;
String delQuery =
"DELETE FROM tbl_raum WHERE id_raum=2";
anz = stm.executeUpdate(delQuery)
```

11.2.6.Lesender Zugriff

Methode executeQuery liefert ein ResultSet zurück

```
String query = "SELECT * FROM tbl_raum";
ResultSet rs = stm.executeQuery(query)
public boolean next () throws SQLException
```

(muss auch vor erstem Tupel Zugriff durchgeführt werden um Zeiger darf zu setzen)
Schrittweise Abarbeitung der Ergebnismenge (Tupel für Tupel) Boolean wert gibt an, ob noch weitere
Tupeln vorhanden sind

Ergebnisse Ausgeben

```
public String getString (String columnName) throws SQLException
```

11.2.7.Verbindung schliessen

Connection.close Schliessen alle Statement- und resultSet Instanzen
Statement.close Schliessen der dazugehörigen ResultSet instanz
ResultSet.close

11.3. Transaktionen

Transaktionen bestehen aus mehreren zusammengehörenden Aktionen
JDBC bietet Möglichkeit nicht gelungene Transaktionen zurückzusetzen
setAutocommit, **commit**, **rollback**

Beispiel:

```
catch (SQLException e) {  
    if (connection != null) {  
        connection.rollback();  
    }  
}
```

11.4. DBMS und Property-Dateien

Ziel: DBMS soll ausgetauscht werden können, keine Code Anpassungen nötig

Lösung: Property Datei. spezifische DBMS Daten speichern vom Typ key-value.

Bei einem DBMS austausch muss lediglich die Property-Datei ersetzt werden. Programmcode bleibt unverändert.

Die entsprechende Treiberklasse muss verfügbar und im CLASSPATH sein.

11.4.1.Beispiel db.properties

```
jdbc.url=jdbc:mysql://147.88.111.111:3306/raum_db  
#=== Treiberklasse  
jdbc.drivers=com.mysql.jdbc.Driver  
#=== Benutzername  
jdbc.user=student  
#=== Password  
jdbc.password=geheim
```

11.4.2.Auslesen aus .properties Datei

```
// Properties-Objekt erzeugen  
Properties dbProperties = new Properties();  
// Klassenloader holen  
ClassLoader cLoader = this.getClass().getClassLoader();  
// Properties laden  
dbProperties.load(cLoader.getResourceAsStream("db.properties"));  
// Treiber-Klasse auslesen  
String driverClass = dbProperties.getProperty("jdbc.drivers");  
// Treiber laden  
Class.forName(driverClass);  
  
// URL, Benutzername und das Kennwort auslesen  
String dbUrl = dbProperties.getProperty("jdbc.url");  
String user = dbProperties.getProperty("jdbc.user");  
String pwd = dbProperties.getProperty("jdbc.password");  
// Verbindung zur Datenbank herstellen  
con = DriverManager.getConnection(dbUrl, user, pwd);  
...
```


11.5. Metadaten

Metadaten = neben Nutzdaten auch zusätzliche Informationen über DB, Tabellen, Spalten usw.

Metadaten auslesen über Klassen: **DatabaseMetaData** und **ResultSetMetaData**

11.5.1. DatabaseMetaData Beispiel: Tabellennamen auslesen

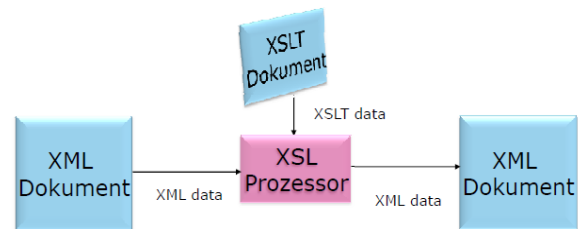
```
// Metadaten für die Datenbank holen
DatabaseMetaData dbMetaData = connection.getMetaData();
// Tabellennamen auslesen
ResultSet rSet = dbMetaData.getTables(null, null, null, new String[]{"TABLE"});
// Tabellennamen ausgeben
while(rSet.next()) {
    System.out.println("TABLE: " + rSet.getString("TABLE_NAME"));
}
```

12. XSLT

XSL Transformation, kurz XSLT, ist eine Programmiersprache (eigene Sprachfamilie) zur Transformation von XML-Dokumenten. Sie ist Teil der Extensible Stylesheet Language (XSL)

12.1. Grundlegende Funktionsweise:

Grundlegend gesagt, wird durch ein XSLT-Stylesheet-File ein bestehendes .xml File (Input), welches mit Daten wie zum Beispiel ein Adressverzeichnis eines Vereins beinhaltet, durch die im XSLT-Stylesheet definierten Regeln transformiert und in einem neuem .xml File (Output) gespeichert.



12.2. Anwendungsgebiet von XSLT:

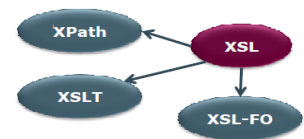
- **POP** (Presentation Oriented Publishing) bezeichnet die Transformation zum Zwecke der Darstellung. Mit unterschiedlichen Stylesheets können die Daten in XHTML, XSL-FO und viele andere Formate umgewandelt werden. Das Zieldokument muss nicht zwingend ein XML-Dokument sein
- **MOM** (Message Oriented Middleware) bezeichnet die Transformation zum Zwecke des Datenaustausches.
Beispiel: Statische Daten liegen als XML vor und werden mit Hilfe unterschiedlicher Transformationen aufbereitet.

12.3. Technischer Ablauf:

- Die Umwandlung findet über einen so genannten **XSLT- Prozessor** statt, welcher nicht nur mit Eclipse wie im Unterricht behandelt, sondern **in jedem gebräuchlichen Browser als Plugin** vorhanden ist.
- Diese in einem Stylesheet definierten Regeln (**XSL**) können die Daten ergänzen, sortieren, filtern usw.
- Bei der Verarbeitung schaut XSLT nach Elementen des Quellbaumes und transformiert diese zu Elementen des Zielbaumes

12.4. Aufbau von XSL (Extensible Stylesheet Language)

Die Programmierung von XSL ist **auf der Basis von XML** aufgebaut. XSL Programmierung wird verwendet um die Modifikationen welche das Output File enthalten sind durchzuführen. Dieser **XSL Code** befindet sich im **XSLT-Stylesheet**. Folgend die einzelnen Komponenten der XSL Programmiersprache:



12.5. XPATH:

Mit XPATH wird innerhalb des **Input .xml** Files in der **Baumstruktur de XML Files** die korrekten **Knoten** angesteuert. Beispiel dazu weiter unten.

Einfacher gesagt wird mit XPATH die **Adressierung des entsprechenden Knotens** vorgenommen welcher behandelt werden soll.

12.6. XSLT-FO: (Formating Objects)

BXSL-FO beschreibt wie Grafische Elemente im Ouput File angeordnet werden sollen. Allerdings wurde im Unterricht nicht häher darauf eingegangen.

12.7. Knoten

Die Achsen der Knoten im .xml Dokument werden fogendermassen angesprochen:

Child-Knoten:	./	(direkt untergeordnete Knoten)
Parent	../	(der direkt übergeordnete Elternknoten)
Self	.	der Kontextknoten selbst
Descendant	..//	untergeordnete Knoten
Attribute	@	Attributknoten ansprechen
Unterelemente	./*	alle Unterelemente des gegenwärtigen Knotens

Alle Knoten haben Informationen (Knoteneigenschaften) gespeichert, welche ebenfalls für die Programmierung verwendet werden können. Diese Knoteneigenschaften sind:

NAME, STRING VALUE, Base URI, Children, Parent node, Attribute:

12.8. Attribut Ausgabe Beispiel

12.8.1.Input File:

```
<?xml version="1.0"?>
<!DOCTYPE Entsafterliste SYSTEM "entsafter.dtd">
<Entsafterliste>
  <Entsafter id="7002.702" leistung="60" hersteller="TRISA">
    <Name>Juicer</Name>
    <Bild>bilder\TR-7002.702.jpg</Bild>
    <Beschreibung>Frische Obstsäfte schnell zubereiten!</Beschreibung>
    <Garantie>24</Garantie>
    <Gewicht></Gewicht>
    <Preis währung="CHF">23.00</Preis>
    <Verkäufer>http://www.trisaelectro.ch</Verkäufer>
  </Entsafter>
</Entsafterliste>
```

12.8.2.XSLT File:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">
<xsl:output method="html" />
<xsl:template match="/">
<HTML>
<BODY>
<table>
  <xsl:for-each select="Entsafterliste/Entsafter" />
    <xsl:if test="Preis/@währung='CHF'" />
      Anzahl der Entsafter:
      <xsl:value-of select="count(//Entsafter)" /><BR />
      Preis aller Entsafter: CHF
      <xsl:value-of select="sum(//Preis)" /><BR />
      Durchschnittspreis: CHF
      <xsl:value-of select="sum(//Preis) div count(//Entsafter)"/><BR />
    </xsl:if>
  </xsl:for-each>
</table>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

12.9. Absoluter versus relativer Pfad

Beim Absoluten und relativen Pfad handelt es sich um zwei Arten, wie der gewünschte Pfad angesprochen werden kann.

Folgend der Unterschied zwischen einem Absoluten und einem relativen Pfad. Im zweiten Beispiel wird aus der Variable typ der entsprechend hinterlegte Pfad ausgelesen und verarbeitet.

Absoluter "XPath"-Ausdruck

(Wir starten von der Wurzel des XML-Dokuments und navigieren hinunter zum gewünschten Element)

```
<xsl:value-of select="/Clubverwaltung/Mitglied/Name" />
```

Relativer "XPath"-Ausdruck

(relativ zum aktuellen Element, gib mit denWert des Typ-Attributs)

```
<xsl:value-of select="@typ" />
```

12.10.Zielformate:

Als Output- File können diverse Formate als Ausgabe verwendet werden. Im Script wurden folgende Ausgabentypen verwendet.

Zielformate einer XLST Verarbeitung: **xml**(Standart), **html**, **text**

12.11.Grundlegender Aufbau eines XSLT Dokumentes

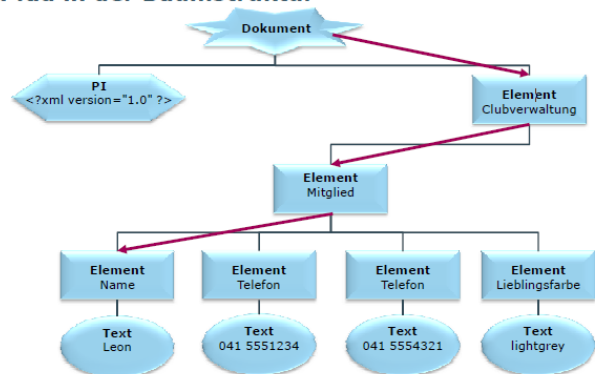
Ein XPath-Ausdruck adressiert Teile eines XML-Dokuments, das dabei als Baum betrachtet wird.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    [Aktionen]
  </xsl:template>
  <xsl:template match="Clubverwaltung">
    [Aktionen]
  </xsl:template>
</xsl:stylesheet>
```

12.12.Reihenfolge der Abarbeitung

Folgend wird die Reihenfolge dargestellt, in welcher hierarchisch auf die einzelnen Elemente zugegriffen wird. Wenn für ein Element durch das Template eine Aktion definiert ist, wird sie für alle Unterelemente angewendet. Es sei denn auf einer tieferen Stufe sind für das Child Element weitere Aktionen definiert, dann wird diese spezifische Aktion durchgeführt und die erste weggelassen.

Pfad in der Baumstruktur



12.13.Namensräume und XSLT

Dokument mit Namensraum

XML-Daten bestehen aus Elementen, auch Tags genannt, sowie null oder mehreren Attributen und deren Werten.

Der Namensraum gibt den genauen Ort des Files an.

Dies ist nötig, falls mehrere Tags mit demselben Namen in unterschiedlichen .xml Files vorkommen.

- **Local-name()** ⇒ gibt lokalen Namen eines Elementes zurück.
- **Namespace-uri()** ⇒ Gibt URI eines Namensraumes zurück.
- **Name()** Funktion die den Namen mitsamt dem Namesraum-Präfix zurückgibt.

13. UML

- UML ist eine grafische Notation zur Darstellung bzw. Modellierung von Softwaresystemen. Aber UML ist keine Programmiersprache und kein vollständiger Ersatz für die Textbeschreibung.
- UML wurde entwickelt um die Softwareentwicklung zu vereinheitlichen.
- UML ist standardisiert (OMG; Object Management Group und ISO) und wird laufend weiterentwickelt (aktuelle Version 2.2)
- Normalerweise wird UML mit Hilfe von sog. Case Tools erstellt. (z.B. ArgoUML)
- Das Ziel von UML ist es, eine möglichst einheitliche Sprache zu werden, die für möglichst viele Zwecke, und von möglichst vielen Entwicklern gebraucht wird.

13.1. Use Case Diagram (Anwendungsfalldiagramm)

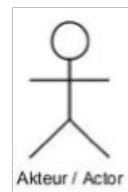
Das Anwendungsfalldiagramm ist aus **Sicht des Benutzers** erstellt. Es hilft dabei, die konkreten Anforderungen an ein Softwaresystem zu verstehen.

- „Ein Anwendungsfalldiagramm beschreibt die Zusammenhänge zwischen den einzelnen Anwendungsfällen und den Akteuren“
 - „Das Diagramm beschreibt nur, welche Anwendungsfälle es gibt und wer daran beteiligt ist. Wie Abläufe und Reihenfolgen dargestellt werden, sehen Sie im Abschnitt über Aktivitätsdiagramme“
- (Die UML-Kurzreferenz für die Praxis S. 22, 23)

Ein Anwendungsfalldiagramm ist eine grafische Darstellung eines Anwendungsfall-Modells und kann **Anwendungsfälle**, **Akteure** und **Beziehungen** enthalten.

13.1.1. Akteure

Akteure werden als „Strichmännchen“ dargestellt, und können Menschen, Systeme oder Prozesse sein. Ein Akteur initiiert einen Anwendungsfall – ohne Akteur kein Start. Die Beziehung zwischen Akteur und Anwendungsfall wird mit einer einfachen Linie (Assoziation) dargestellt.

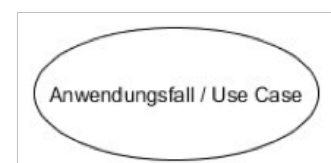


13.1.2. Assoziationen

Assoziationen sind Beziehungen zwischen Akteuren und Anwendungsfällen. Sie werden gelegentlich gerichtet („mit Pfeil“) angegeben. Der Autor des Buches „Die UML-Kurzreferenz 2.3 für die Praxis“ empfiehlt aber keine gerichteten Assoziationen zu verwenden. Weiter besteht die Möglichkeit eine „Multiplizität“ anzugeben. (Wird gleich Dargestellt wie Kardinalität im Fach Informationssysteme. Null (0), Eins (1) oder Mehrere (*))

13.1.3. Anwendungsfall

Anwendungsfälle werden in Ellipsen dargestellt. Sie müssen beschrieben werden. Ein Anwendungsfall ist ein Ablauf oder eine Aktivität, die isoliert betrachtet werden muss. Jeder Anwendungsfall wird von einem Akteur angestoßen. Ein Akteur kann ein Benutzer oder ein anderer Prozess / ein anderes System sein.



13.1.4. Beschreibung von Anwendungsfällen

Der Anwendungsfall muss genau beschrieben werden, damit Missverständnisse und Fehlplanungen möglichst selten sind. Oft wird ein Anwendungsfall mit Hilfe eines Formulars beschrieben.

ID	L1UC6
Name	Geld am Automat auszahlen

Beschreibung	Einem Verfügungsberechtigten wird durch einen Geldautomaten ein vom Verfügungsberechtigten geforderter Geldbetrag ausgezahlt und vom zugehörigen Konto abgebucht.			
Akteur	Verfügungsberechtigter, Kontoführungssystem			
Vorbedingung	Der Geldautomat ist bereit, eine Karte aufzunehmen.			
Auslösendes Ereignis	Der Verfügungsberechtigte steckt eine Karte in den Geldautomat			
essenzieller Ablauf (happy path)	<ul style="list-style-type: none"> • Verfügungsberechtigten identifizieren • Angeforderten Geldbetrag bestimmen • Auszahlungsmöglichkeit prüfen • Auszahlung auf Konto buchen • Geldbetrag übertragen • Karte auswerfen 			
Alternativer Ablauf	– ...			
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

(Beispiel einer Anwendungsfallbeschreibung mit Formular)

13.1.5. Beziehungen

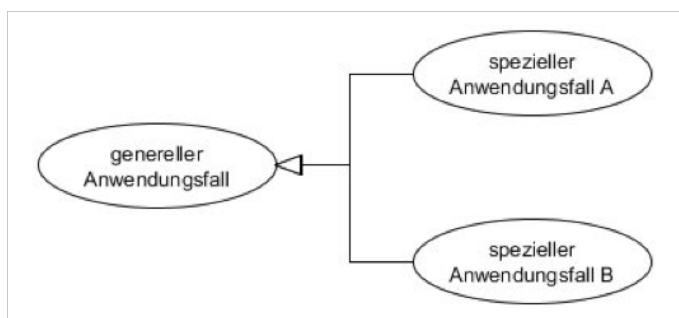
UML kennt folgende Beziehungen unter Anwendungsfällen:

- Generalisierung
- Includes
- Extends

13.1.5.1. Generalisierungen

Eine Generalisierung wird verwendet, wenn ein allgemeingültiges Verhalten als Basisanwendungsfall abgebildet werden kann. Dazu werden dann spezialisierte Anwendungsfälle erstellt, die den Basisanwendungsfall erweitern oder überschreiben. (↑ Generalisierung, ↓ Spezialisierung).

Spezialisierungen werden als „Strich mit Pfeil“ notiert und werden **immer importiert!**



Auch Akteure können generalisiert werden!

(A. Roggers Beispiel dazu „Produkte einlagern“ als genereller Fall sowie „Verderbliche Produkte einlagern“ und „Nichtverderbliche Produkte einlagern“)

13.1.5.2. Includes

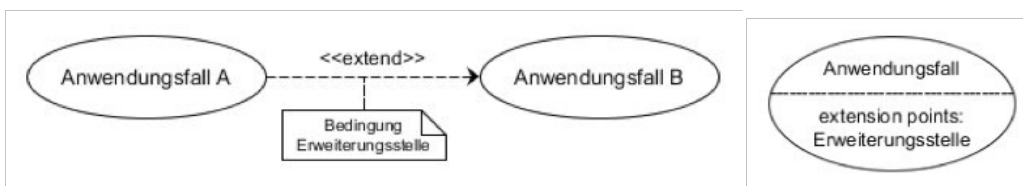
Includes werden benutzt, wenn die gleiche Aktion an mehreren Stellen des Dokuments vorkommt. Includes dienen nur der Verminderung von Redundanzen. Die Notation ist eine gestrichelte Linie mit Pfeil **zum Include** hin. Anwendungsfall A beinhaltet Anwendungsfall B. Includes werden immer inkludiert.



13.1.5.3. Extends

Eine Extends-Beziehung ist eine Art Include-Beziehung die nicht zwangsläufig inkludiert werden muss. Es wird davor eine Entscheidung fallen, ob das importieren der Funktionalitäten notwendig ist. Dies passiert am sogenannten extension point (Erweiterungspunkt).

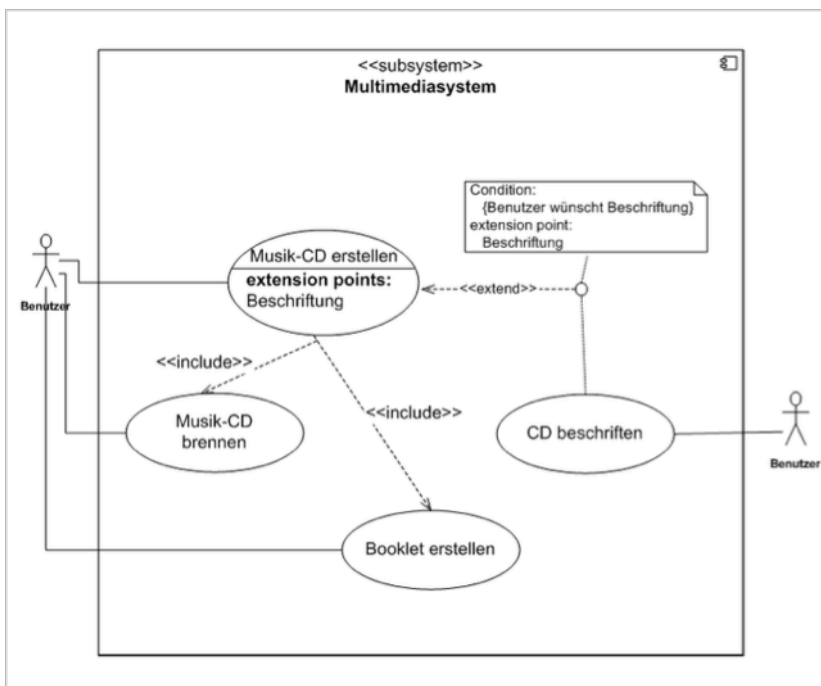
Achtung: Der Pfeil geht vom Extend weg, also zum extension point hin!

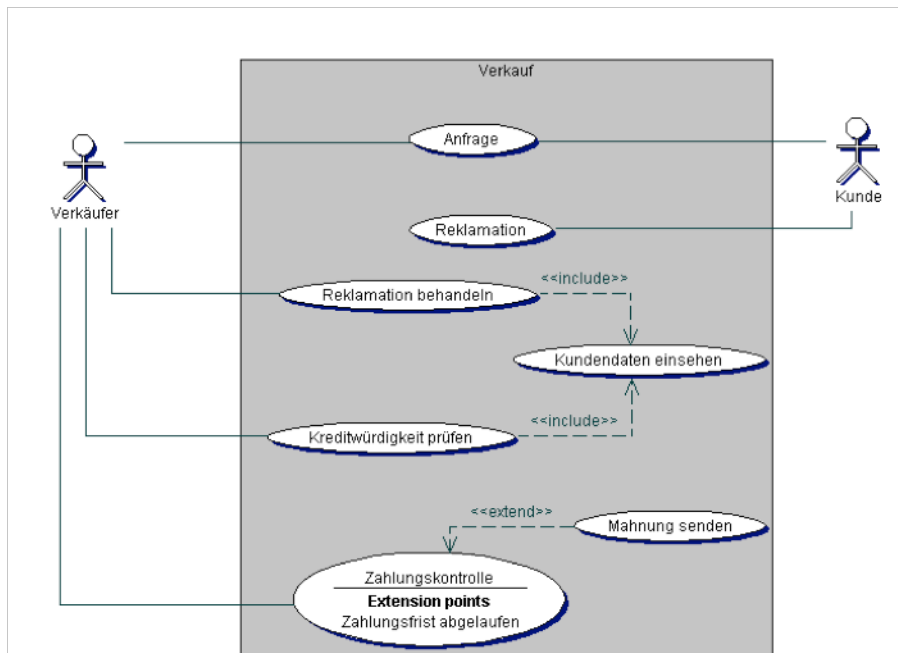


13.1.6. Systemgrenze

Anwendungsfälle werden von einem System umschlossen. Das System definiert dabei eine klare Systemgrenze, die von Akteuren „übertreten“ wird, wenn sie einen Anwendungsfall initiieren. Die Systemgrenze wird dabei als „Rahmen um die Anwendungsfälle dargestellt, und trägt den Systemnamen. (Siehe Beispiele)

13.1.7. Beispiele

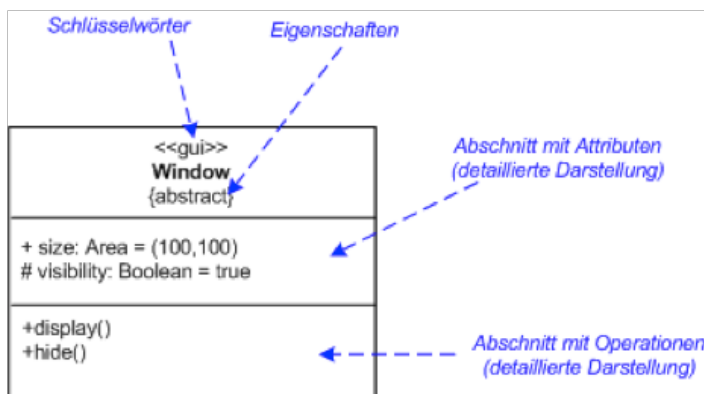




Quelle Grafiken: <http://de.wikipedia.org/wiki/Anwendungsfalldiagramm>, Foliensatz Rogger

13.2. Class Diagram (Klassendiagramm) – nicht prüfungsrelevant

Das Klassendiagramm wird hauptsächlich zur Modellierung von objektorientierten Programmen verwendet. Kann aber auch für betriebliche Modellierungen verwendet werden (zB Geschäfts- oder Fachklassen)



- Eine Klasse wird in die Teile: Name, Attribute und Operationen geteilt
- Eine Klasse kann abstrakt sein (wird nie eingesetzt, es wird lediglich von ihr geerbt)
- Die Sichtbarkeit der Operationen und Attribute kann angegeben werden:
 - + public
 - # protected
 - private
 - ~ package

13.2.1. Beziehungen

13.2.1.1. Assoziation (hat-Beziehung)

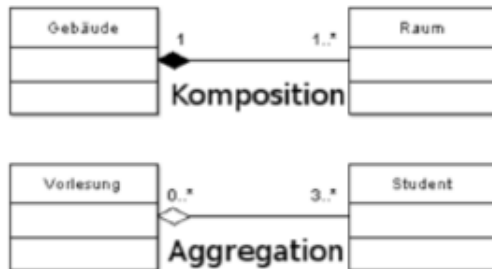
- Assoziationen (siehe 1.1.2) werden gleich dargestellt wie im Anwendungsfalldiagramm. (Ein „normaler Strich“).
- Kardinalität ist möglich.
- Richtung ist möglich.

13.2.1.2. Vererbung (ist-Beziehung)

Vererbungen sind Generalisierungen (siehe 1.1.5.1). Die spezielle Klasse „erbt“ von der generellen Klasse. Sie wird mit einer durchgezogenen Linie mit geschlossenem Pfeil dargestellt.

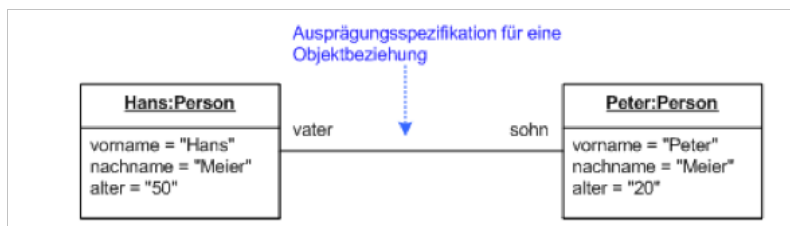
13.2.1.3. Aggregation & Komposition

Die Aggregationen (Besteht-aus-Beziehung) und Kompositionen (starke Besteht-aus-Beziehung) drücken die Beziehung zwischen „dem Ganzen und seinen Teilen“ aus. Eine Aggregation sowie die Komposition steht dabei für ein „X Besteht aus Y“. Bei der Komposition ist das Ganze jedoch von mindestens einem Teil abhängig.



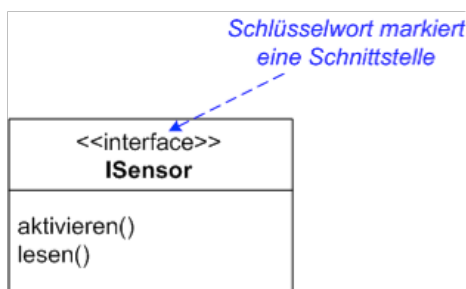
13.2.2. Objekte

Ein Objekt ist die Instanz einer Klasse. Der Klassenname wird unterstrichen, und folgendermassen dargestellt: Exemplarname:Klassenname



13.2.3. Schnittstellen (Interface)

Schnittstellen werden lediglich mit dem Schlüsselwort **Interface** versehen.



13.3. Objektdiagramm (Object Diagram) – nicht prüfungsrelevant

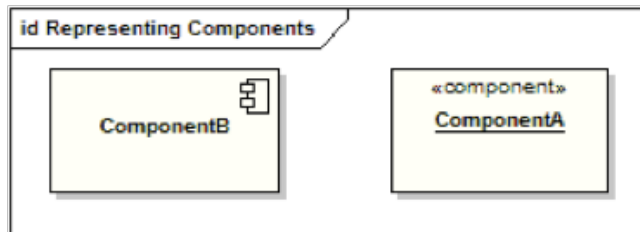
„Das Objektdiagramm zeigt eine ähnliche Struktur wie das Klassendiagramm, jedoch statt der Klassen exemplarisch eine Auswahl der zu einem bestimmte Zeitpunkt existierenden Objekte mit ihren **augenblicklichen Werten**. Ein Objektdiagramm ist sozusagen ein Schnappschuss der Objekte im System zu einem bestimmten Zeitpunkt“. (Die UML Kurzreferenz 2.3 für die Praxis S. 99)

13.4. Komponentendiagramm (Component Diagram)

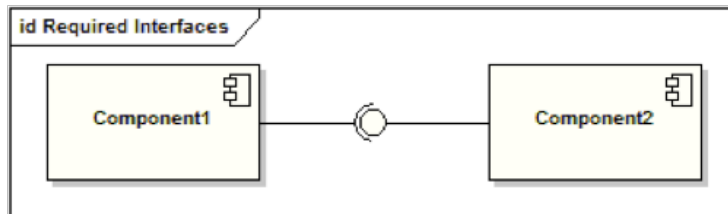
Ein Komponentendiagramm stellt die Struktur eines Systems zur Laufzeit dar. „Wie ist mein System strukturiert und wie werden diese Strukturen zur Laufzeit erzeugt?“

Es hat ein höheres Abstraktionsniveau als das Klassendiagramm, eine Komponente kann mehrere Klassen / Objekte / Steuerungen enthalten.

Eine Komponente wird entweder durch das Schlüsselwort <<component>> oder durch das „Lego Symbol“ markiert.



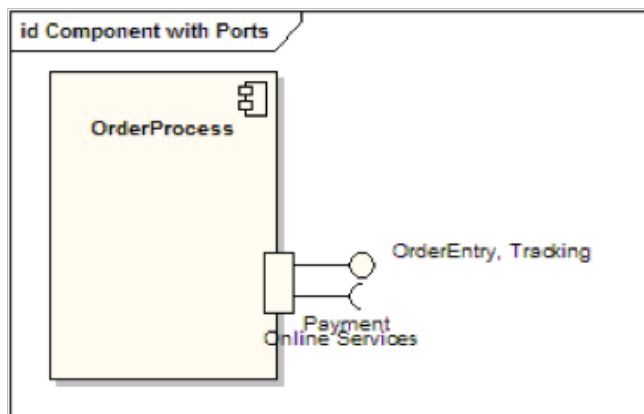
Komponenten kommunizieren via Schnittstellen miteinander. Diese werden Assembly-Konnektoren genannte (lollypop)



Wir haben dabei immer eine Komponente die einen Dienst oder eine Schnittstelle zur Verfügung stellt (Kreis, „Männchen“) und Informationen liefert sowie eine Komponente die eine anfordernde Schnittstelle zur Verfügung stellt (Halbkreis „Weibchen“) und Informationen anfordert.

13.4.1.Ports

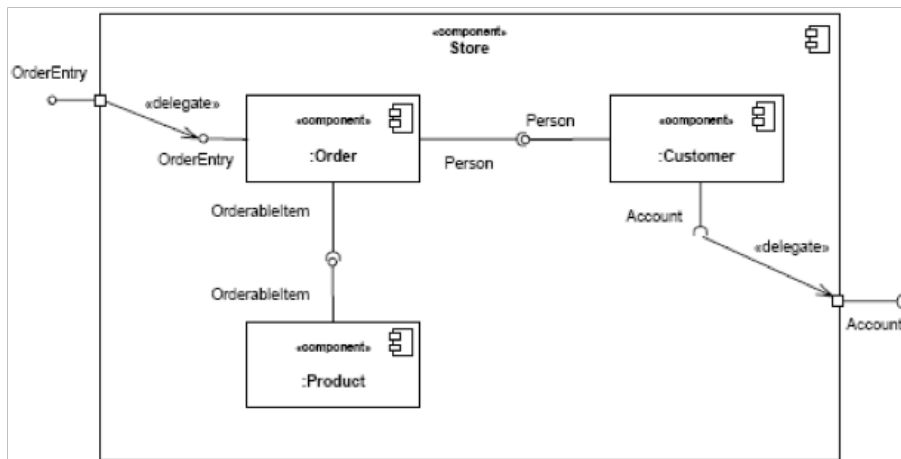
Es ist auch möglich, Ports zu modellieren. Dabei wird die folgende Notation verwendet (Kasten):



Hier bezieht sich „Online Services“ auf den Kasten. Der Port heisst also „Online Services“ und hat die beiden bereitstellenden Schnittstellen „OrderEntry“ und „Tracking“ sowie die anfordernde Schnittstelle „Payment“.

13.4.2.White-Box-View

Eine Komponente kann in der White-Box-View (Glashausansicht) wiederum in die eigenen Komponenten zerlegt werden.



Es ist möglich hier andere Modellelemente hineinzumodellieren, z.B. Klassen.

13.5. Kommunikationsdiagramm

- Das Kommunikationsdiagramm ist ein **Verhaltensdiagramm** der UML
- „Das Kommunikationsdiagramm zeigt Interaktionen zwischen Teilen einer meist komplexen Struktur“.
- Bis zur UML 1.5 trug dieser Diagrammtyp den Namen Kollaborationsdiagramm (Collaboration Diagram)

Das Kommunikationsdiagramm zeigt den zeitlichen Ablauf von Interaktionen („Kommunikation“) zwischen verschiedenen Objekten der UML. (Objekten im Sinne von Programmierobjekten)

13.5.1. Elemente

13.5.1.1. Lebenslinie

Im Sequenzdiagramm hat jedes „Objekt“ eine Lebenslinie, das Kommunikationsdiagramm ist eine Abstraktion des Sequenzdiagrammes. Daher werden die „Objekte“ hier auch Lebenslinie (Lifeline) genannt. Sie werden gleich wie im Klassendiagramm dargestellt.

13.5.1.2. Beziehung

Eine mögliche Beziehung zwischen zwei Objekten wird mit einer einfachen Linie dargestellt.

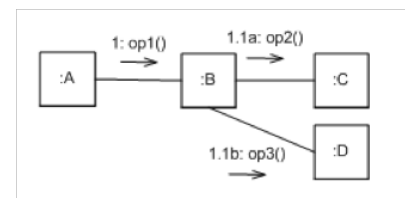
13.5.1.3. Nachricht (Signal)

Eine Nachricht hat einen Namen und eine Sequenznummer. Die zeitliche Abfolge der Nachrichten wird über die Sequenznummern dargestellt.

Nebenläufigkeiten sind möglich und werden nummeriert. Man findet unterschiedliche „Regeln“ für die Nummerierung. Gemäss Wikipedia wird jede gestartete Aktion mit einer Nummer versehen, ihre Schritte werden nach dem Punkt hinauf gezählt, mit Buchstaben werden Nebenläufigkeiten dargestellt.

Mittels des **Sequenzausdrucks** können

- **sequentielle und geschachtelte Nachrichten:** 1, 2, 3, geschachtelte Nachrichten im Sinne eines untergeordneten Prozesses: 3, 3.1, 3.2, 3.2.1, 3.2.2
- **nebenläufige Nachrichten:** 5a, 5b
- **bedingte Nachrichten:** [Bedingung in Pseudocode]:Nachricht
- **iterative Nachrichten:** 3*[für jeden Wert]:Nachricht durch Verwendung des Iteratorsterns * modelliert werden

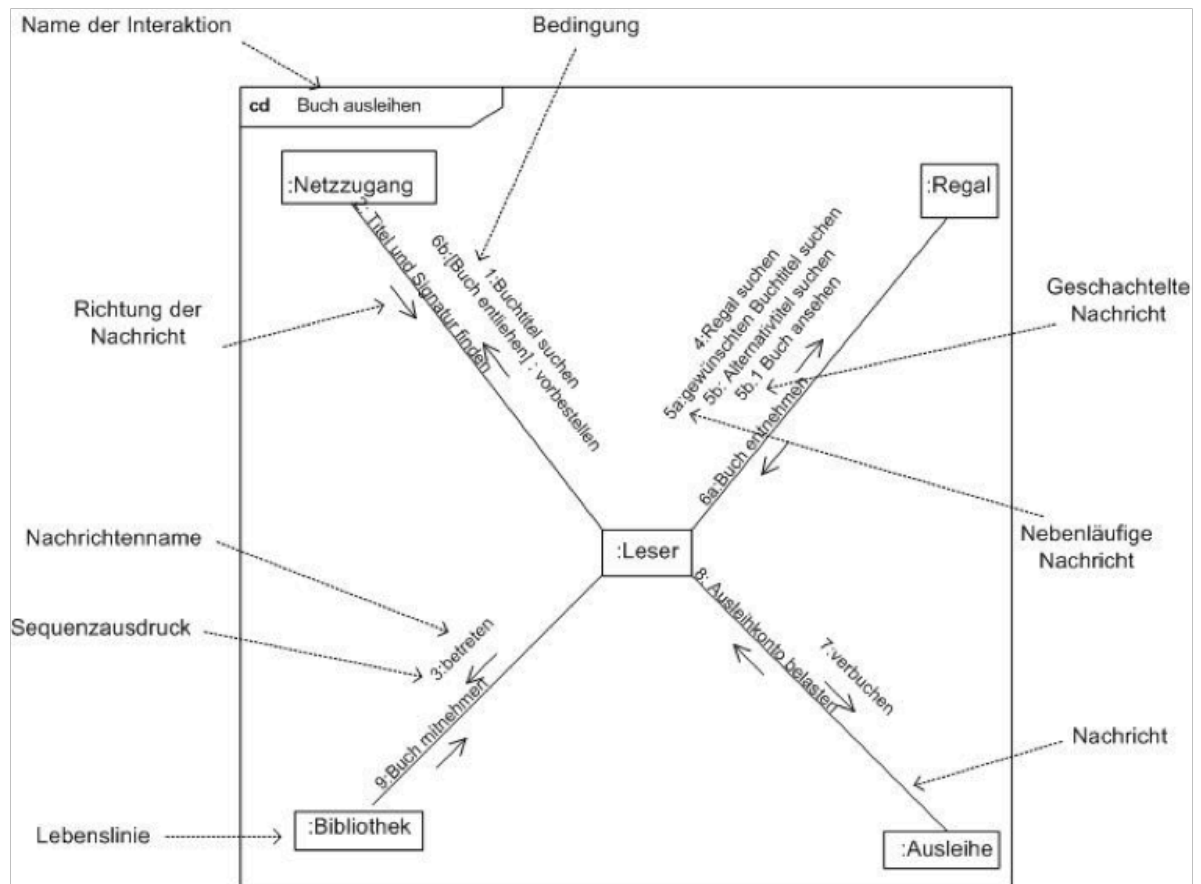


Quelle: <http://se.cs.uni-magdeburg.de/tutorial/UML2/Kommunikationsdiagramm.htm>

13.5.1.4. Umrandung

Jedes Kommunikationsdiagramm hat einen Rand und einen Namen. Vor dem Namen stehen die Buchstaben *sd* oder *interaction* (gemäss Wikipedia auch *cd*).

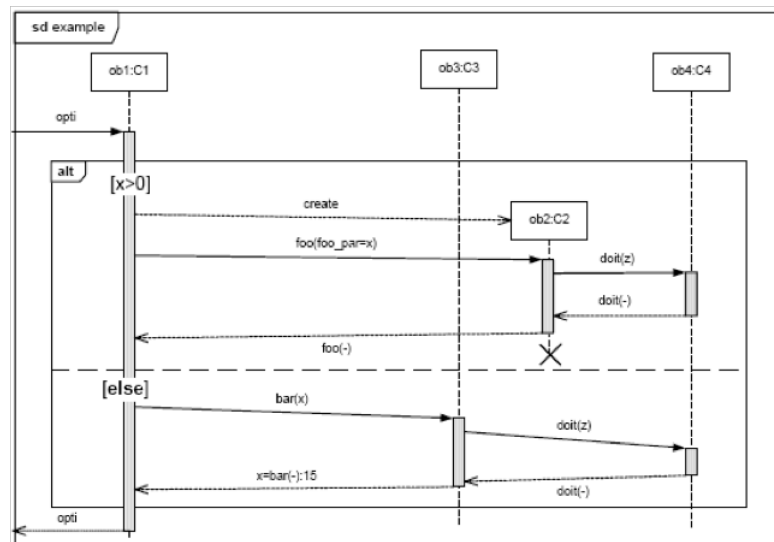
13.5.2. Beispiel



13.6. Sequenzdiagramm

Sequenzdiagramme zeigen mit Hilfe von Lebenslinien die Interaktionen zwischen Kommunikationspartner (Objekte) auf.

Dabei werden – wie im Kommunikationsdiagramm – zeitliche Abläufe mit Hilfe sog. Lebenslinien dargestellt. Zusätzlich können aber auch die Daten, die versandt werden, dargestellt werden.



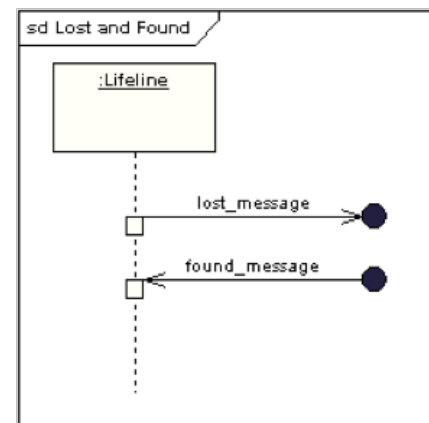
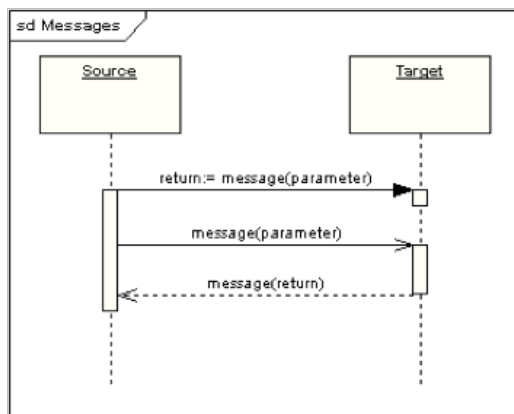
13.6.1. Lebenslinien

Jedes Sequenzdiagramm hat Lebenslinien. Eine Lebenslinie ist ein zeitlicher „von oben nach unten – Ablauf“ eines Objektes. Dabei wird die Beschriftung *Klassenname:Objektname* verwendet werden. Gilt eine Lebenslinie für alle Objekte einer Klasse, kann wiederum der Objektname weggelassen werden.

Wird eine Aktion entlang der Lebenslinie ausgeführt, nennt sich das *Aktionssequenz* und wird mit einem Kasten entlang der Lebenslinie dargestellt. Dies ist jedoch optional. Eine Aktionssequenz hat ein Starterereignis (oft Empfang einer Nachricht) sowie ein Endereignis. Nur dazwischen können wiederum Nachrichten versandt werden.

13.6.2. Nachrichten

Die Kommunikation kann entweder synchron (gefüllte Pfeilspitze) oder asynchron (offene Pfeilspitze) erfolgen. (Bild links)



Dazu kann eine Nachricht **verloren** bzw. **gefunden** sein. Dies wird modelliert, wenn der Empfänger bzw. der Sender nicht Bestandteil des Diagrammes ist. (Bild rechts)

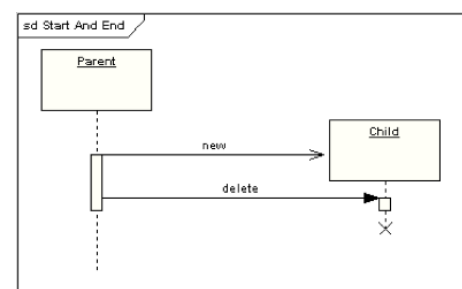
Nachrichten können **rekursiv** (an sich selber adressiert) sein. Wird normalerweise verwendet um eine andere Methode des eigenen Objekts aufzurufen.

Nachrichten können Erzeugungsnachrichten sein. Sie kreieren normalerweise Objekte und sind mit *create* beschriftet.

13.6.3. Lebenslinien Teil 2

Mit jedem kreierten Objekt wird auch die entsprechende Lebenslinie erstellt. Das Ende einer Lebenslinie wird mit einem X markiert. Auch bei der Erstellung der Objekte wird der zeitliche Ablauf beachtet (Objekt wird „weiter unten“ dargestellt).

Bild: Beim Aufruf einer Aktionssequenz kann eine Zeitbeschränkung mitgegeben werden: reply {>5ms}

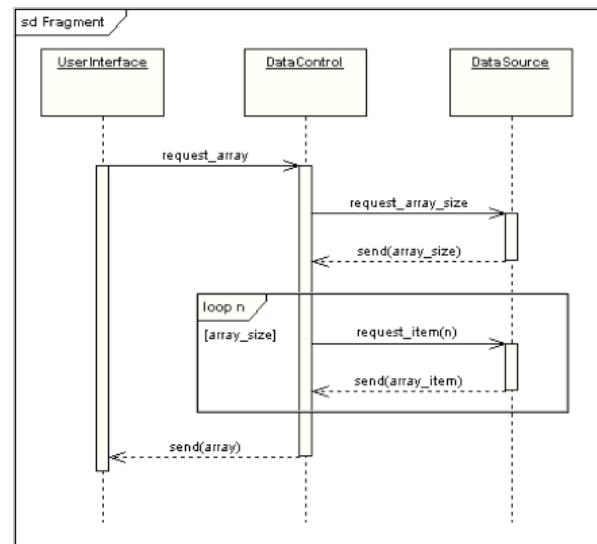


13.6.4. Fragmente

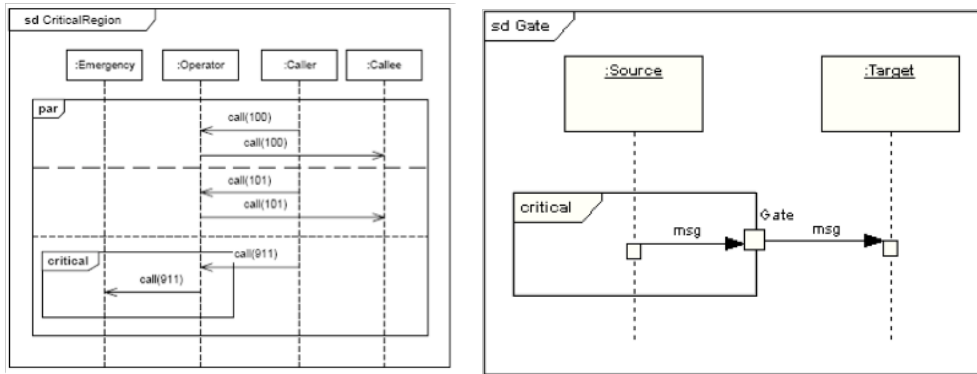
Eigentlich ist ein Sequenzdiagramm nicht dazu gedacht eine Ablauflogik darzustellen. Dennoch ist dies möglich. Dafür werden sog. Fragmente verwendet. Die Idee ist, dass ein Teil des Ablaufs in einen Rahmen gepackt wird, der dann mit Hilfe eines Namens eine spezielle Funktion erhält.

13.6.4.1. Fragmente (Liste)

- Alternative Fragmente (**alt**) (Alternative Fragment) modellieren Auswahl-Konstrukte `if...then...else...`
- Optionales 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.
- Parallele 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 Occurence) 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.



13.6.4.2. Verschachtelung und Gates



- Fragmente können geschachtelt werden. So arbeitet die Vermittlung bei normalen Telefonnummern (quasi) parallel. Ein Notruf muss dagegen sofort durchgestellt werden, ist also kritisch
- Um aus einem Fragment hinaus oder hinein zu kommunizieren, wird ein Gate verwendet.

Quelle: Script Rogger

13.6.5. Kommunikationsdiagramme – Sequenzdiagramme

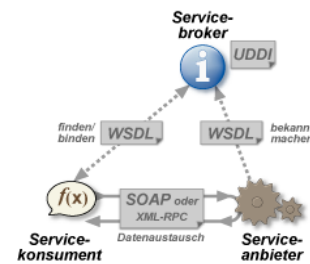
„In Sequenzdiagrammen werden Szenen dargestellt an denen wenige Klassen beteiligt sind, die viele Nachrichten austauschen. Kommunikationsdiagramme eignen sich zur Darstellung von Szenen mit vielen Klassen, die aber wenige Nachrichten austauschen.“

Quelle: <http://www.fbi.h-da.de/labore/case/uml/kommunikationsdiagramm.html>

14. Web Services

Sie sind **plattform-** und **implementierungsunabhängige** Softwarekomponenten, das heisst, der Client muss nicht wissen, was für eine Sprache, Betriebssystem oder Computertyp der Server verwendet.

- programme tauschen Daten über Netzwerke
- starten auf entfernten Rechnern Funktionen (RPC)
- Fast alle diese Protokolle basieren auf XML (ausser UDDI)



WSDL	Beschreibt die Softwarekomponente
UDDI	Dienstmakler
SOAP	Kommunikationsprotokoll, um XML Dokumente über das Internet zu schicken
BPEL	Choreographierung von WebServices

Definition:

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.

- W3C

14.1. Web Service Description Language (WSDL)

WSDL ist eine **Metasprache**, mit deren Hilfe die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle eines Webservice beschrieben werden können. Es werden im Wesentlichen die **Operationen definiert, die von aussen zugänglich sind**.

Ein WSDL-Dokument beinhaltet Angaben zu:

- der Schnittstelle
- Zugangsprotokoll und Details zum Deployment
- Alle notwendigen Informationen zum Zugriff auf den Service, in maschinenlesbarem Format

import	importieren von anderen Dokumenten
types (Datentypen)	Typdefinitionen für die ausgetauschten Nachrichten
message (Nachricht)	Nachrichten, die beim Aufruf einer Service- Operation ausgetauscht werden
portType (Schnittstellentypen)	Eintrittspunkte (Schnittstellen) mit der Definition der Nachrichten
binding (Bindung)	Abgleich des verwendeten Protokolls mit den portType-Schnittstellen
port	
service (Service)	Physische Adressen der Services

14.2. Universal Description, Discovery and Integration (UDDI)

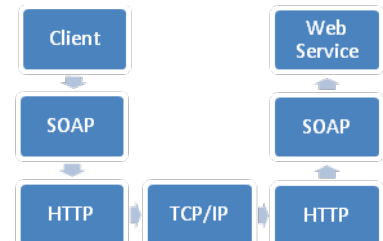
Bezeichnet einen **standardisierten Verzeichnisdienst**, der die zentrale Rolle in einem Umfeld von dynamischen Web Services spielen sollte.

- **standardisierten Verzeichnisdienst** zur Registrierung von Webservices.
- ermöglicht das dynamische Finden des Webservices
- Allerdings wird UDDI nur in eher kleineren Firmennetzwerken verwendet und hat sich nie global durchgesetzt.

14.3. Simple Object Access Protocol (SOAP)

SOAP ist ein Protokoll **zum Austausch XML-basierter Nachrichten** über ein Computernetzwerk.

Es **stellt ein Rahmenwerk** (framework) **zur Verfügung** welches erlaubt, dass beliebige applikationsspezifische Informationen übertragen werden können.



14.3.1. Arbeitsweise:

- Damit SOAP funktioniert muss der Client eine Anwendung haben, die eine SOAP-Anfrage aufbauen kann.
- SOAP braucht HTTP/ TCP zur Übertragung

14.3.2. Aufbau einer SOAP- Nachricht:

Umschlag (Envelope)

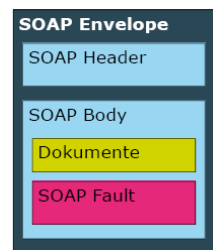
- Definiert den Inhalt der Nachricht

Kopf (Header, optional)

- Enthält Verwaltungsinformation Erweiterungen.
- Informationen für Sicherheitsanforderungen

Rumpf (Body)

- Enthält Aufruf- und Antwortinformationen



14.3.3. Anhang:

Beispiel WSDL:

```

1 <definitions>
2 <types>
... hier werden die verwendeten Datentypen in XML Schema-Syntax beschrieben
4 </types>
5 <message name="KontoAnfrage">
... hier werden die Parameter einer Nachricht beschrieben
7 </message>
8 <message name="KontoAuskunft"> ... </message> ...
9 <portType name="KontoZugriffsPort">
10   <operation name="KontoZugriff">
11     <input message="KontoAnfrage" />
12     <output message="KontoAuskunft" />
13   </operation> ...
14 </portType> ...
15 <binding name="KontoAuskunftSoap"> ... </binding>
16 <service name="KontoAuskunftsService"> ... </service>
17 </definitions>
  
```

14.3.4.Beispiel SOAP:

Anfrage:

```
1 <?xml version="1.0"?>
2 <s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
3   <s:Body>
4     <m:TitleInDatabase xmlns:m="http://www.lecture-db.de/soap">
5       DOM, SAX und SOAP
6     </m:TitleInDatabase>
7   </s:Body>
8 </s:Envelope>
```

Antwort:

```
1 <?xml version="1.0"?>
2 <s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
3   <s:Header>
4     <m:RequestID xmlns:m="http://www.lecture- db.de/soap">a3f5c109b
5     </ m:RequestID>
6   </s:Header>
7   <s:Body>
8     <m:DbResponse xmlns:m="http://www.lecture-db.de/soap">
9       <m:title value="DOM, SAX und SOAP">
10         <m:Choice value="1">Arbeitsbericht Informatik</m:Choice>
11         <m:Choice value="2">Seminar XML und Datenbanken</m:Choice>
12       </m:title>
13     </m:DbResponse>
14   </s:Body>
15 </s:Envelope>
```

15. Message Oriented Middleware (MOM)

Es werden **drei Middleware-Modelle** unterschieden:

- Prozeduraufruf basierte Middleware
- Objekt basierte Middleware
- Nachrichten basierte Middleware

Message Oriented Middleware (MOM)

- MOM ermöglicht die **Übertragung von Nachrichten**, deren **Format nicht festgelegt** (vorgeschrieben) ist
- Mit der MOM ist es möglich, sowohl die **synchrone als auch asynchrone Kommunikation** in einer verteilten Anwendung zu realisieren

Vorteile:

- Kopplung zwischen Kommunikationspartner
- Das ganze System wird **stabiler**, da Abhängigkeiten zwischen Teilsystemen reduziert
- Bessere **Skalierbarkeit**
- Die Belastung einzelner Komponenten kann besser abgefedert werden (**Load Balancing**)

Nachteil:

- Ausfall der MOM führt zum **Ausfall des ganzen Systems**

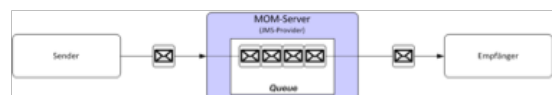
15.1. JAVA Message Service (JMS)

JMS ist eine Schnittstelle, mit der das Senden und Empfangen von Nachrichten mit Hilfe einer MOM **standardisiert** werden soll. Es muss von jemandem implementiert werden, dann steht ein **JMS Provider** zur Verfügung.

Kommunikation zwischen unterschiedlichen Applikation bzw. deren Komponenten realisiert werden

Nachrichtenmodelle:

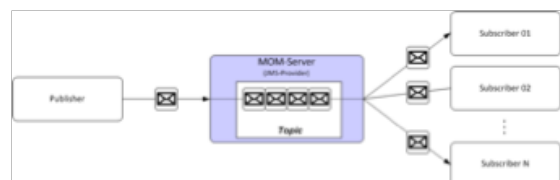
- **Nachrichtenschlangen** (Message Queue) :
Kommunikation über einer Queue als Destination: der Sender stellt die Nachricht in die Queue, der Empfänger nimmt die Nachricht an, sobald er bereit ist.



- Nachrichten werden aufbewahrt

- **Kein Verlust von Nachrichten**

- **Anmelde-Versendesystem** (Publish-Subscribe) :
Ein oder mehrere Publisher stellen Nachrichten in ein Topic: Ein oder mehrere Subscriber nehmen die Nachrichten aus dem Topic



- Bekommen Nachricht nur wenn sie online sind.

- **Verlust von Nachrichten**

- Option: dauerhaftes Abonnieren (persistent subscription)

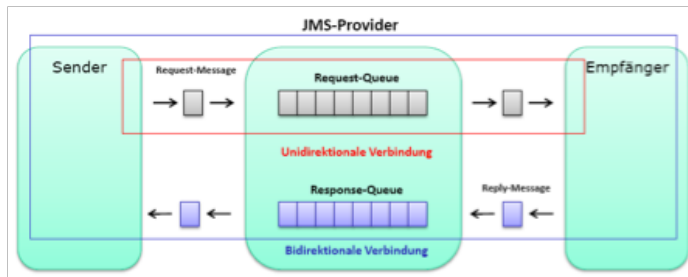
- **One-Way:**

Bei der asynchronen Kommunikation wird in der Regel die One-WayMEP (Message Exchange Pattern) verwendet: Der Sender sendet die Nachricht und erwartet keine Antwort.

- **Request-Reply:**

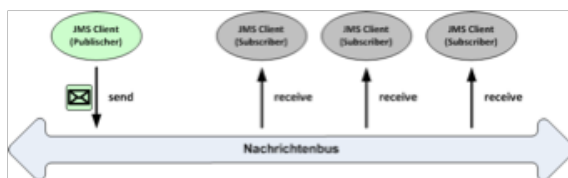
Bei der **synchronen Kommunikation braucht der Sender die Antwort**, um weiter arbeiten zu können, dazu wird eine temporäre Queue angelegt:

2. der Empfänger stellt die Antwort in diese Queue
3. der Sender liest die Antwort aus der Queue (Rollenwechsel)
4. die Queue wird anschliessend gelöscht



Java Message System besteht aus:

- Directory-Service (JNDI)
- JMS-Server(JMS Provider)
- JMS Clients
- Messages
- ConnectionFactory
- Destinationen (Queue und Topic)



Headers: Die Adressinformationen und diverse Metadaten

Properties: Zusätzliche Eigenschaften, die fallbezogen definiert werden können

Nutzdaten: Der Inhalt, der an sich von Interesse ist

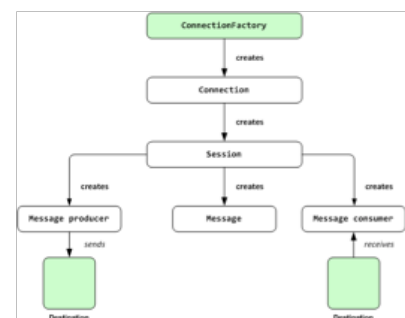
Headers	Properties	Daten
<div> <div>JMSDestination</div> <div>JMSDeliveryMode</div> <div>JMSMessageID</div> <div>JMSTimestamp</div> <div>JMSExpiration</div> <div>JMSRedelivered</div> <div>JMSPriority</div> <div>JMSReplyTo</div> <div>JMSCorrelationID</div> <div>JMSType</div> </div>	<div> <div>... nach Bedarf ...</div> </div>	<div> <div>Message</div> <div>TextMessage</div> <div>StreamMessage</div> <div>MapMessage</div> <div>ObjectMessage</div> <div>ByteMessage</div> </div>

15.1.1.JMS-Klassen

ConnectionFactory und **Destinationen** werden auch **administrierte Objekte** genannt, da sie von einem Administrator erzeugt und bei dem entsprechenden **Verzeichnisdienst (JNDI-Provider)** **angemeldet** werden.

ConnectionFactory: Wird für die Herstellung der Verbindung zwischen einem JMS-Client und JMS-Provider benutzt.

Destination: Die Ablage für Nachrichten, die Sender und Empfänger beim Nachrichtenaustausch benutzen (Queues und Topics).



15.1.2.Java Naming and Directory Service (JNDI)

Schnittstelle für verzeichnisorientierte Dienste und soll den Zugriff auf Verzeichnisdienste vereinfachen bzw. Standardisieren.

Bei einem JNDI-Provider publizierte Objekte können von Clients abgefragt werden (lookup)

15.1.3.OpenJMS

Für das Erzeugen der InitialContext-Instanz müssen folgende Properties angegeben werden:

- `java.naming.factory.initial`
- `java.naming.provider.url`

Möglichkeiten:

- Properties werden in eine **Hashtable abgelegt** und die dem Konstruktor der Klasse InitialContext übergeben.
- Oder die beiden Properties werden in der **jndi.properties Datei angegeben**, die ihrerseits im CLASSPATH sein muss
 - Einfachere Variante, da kein Zugriff auf den Code notwendig ist.

Beispiel jndi.properties:

```
#=====#  
# JNDI Properties #  
#=====#  
  
# InitialContextFactory  
java.naming.factory.initial=org.exolab.jms.jndi.InitialContextFactory  
  
# Provider URL  
java.naming.provider.url=http://10.9.35.119:80
```

Damit die Kommunikation realisiert werden kann, werden folgende Komponenten benötigt:

- JMS Provider (in diesem Falls OpenJMS)
- JNDI Verzeichnisdienst (in diesem Fall im JMS Provider integriert)
- Sender und Empfänger
- Falls die Kommunikation über http realisiert werden soll, muss ein WebServer verwendet werden (beispielsweise Tomcat)

Code des Senders

```
import javax.naming.Context;
import javax.naming.InitialContext;

/* InitialContext erzeugen, ConnectionFactory und Queue 'myWeatherdataQueue'
holen */

InitialContext jndiContext = new InitialContext();
ConnectionFactory conFactory = (ConnectionFactory) jndiContext.lookup
("ConnectionFactory");
Queue queue = (Queue) jndiContext.lookup("myWeatherdataQueue");

/* Connection erstellen und anschliessend Session von der Connection holen */

Connection con = conFactory.createConnection();
Session session = (Session) con.createSession(false, Session.AUTO_ACKNOWLEDGE);

/* Sender erstellen */
MessageProducer sender = session.createProducer(queue);

/* Die ObjectMessage aus dem WeatherdataMessage-Objekt erzeugen */

ObjectMessage objMsg = session.createObjectMessage(weatherdataMessage);

/* Message senden */

sender.send(objMsg);
```

Code des Empfängers

```
/* InitialContext erzeugen, ConnectionFactory und Queue 'myWeatherdataQueue'
holen */

InitialContext jndiContext = new InitialContext();
ConnectionFactory conFactory = (ConnectionFactory) jndiContext.lookup
("ConnectionFactory");
Queue queue = (Queue) jndiContext.lookup("myWeatherdataQueue");

/* Connection erstellen und anschliessend Session von der Connection holen */

Connection con = conFactory.createConnection();
Session session = (Session) con.createSession(false,
Session.AUTO_ACKNOWLEDGE);
/* Empfänger erstellen */
MessageConsumer receiver= session.createConsumer(queue);
/* Connection starten */
connection.start();
/* Message empfangen */
ObjectMessage objMsg= (ObjectMessage)receiver.receive();

// Message verarbeiten
```

<http://openjms.sourceforge.net/usersguide/using.html>

16. Veranstaltung 10 – Enterprise Service Bus (JBoss)

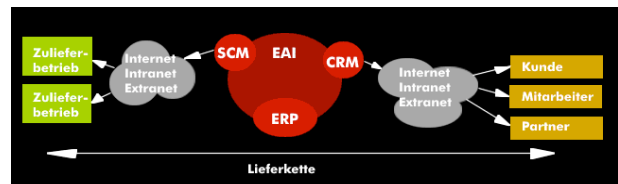
16.1. Enterprise Application (EA)

Heutzutage sind in Unternehmen fast ausschliesslich **Grossapplikationen** (EA) im Einsatz. Die rasante Entwicklung führt oft zu **neuen Technologien**, welche dann vielfach in bestehende EA's **integriert** werden müssen.

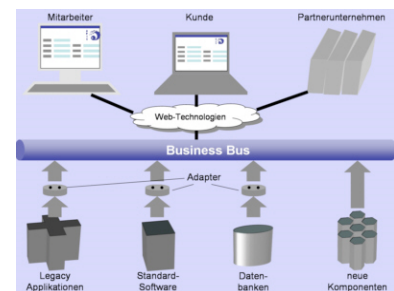
16.2. Enterprise Application Integration (EAI)

- EAI ist der Oberbegriff für **Projekte, Methoden** und **Werkzeuge** zur wechselseitigen **Verbindung herkömmlicher** oder **neu entwickelter Applikationen**.

Das **Ziel** ist die **integrierte Geschäftsabwicklung** durch ein Netzwerk **unternehmensinterner Applikationen verschiedener Generationen** und **Architekturen**.



- Im Gegensatz zu anderen Integrationstechniken, wie der Funktionsintegration oder der Datenintegration, werden beim EAI-Ansatz die **Implementationen** der einzelnen **Geschäftsfunktionen nicht verändert**. Alle funktionalen Schnittstellen werden mittels **Adaptern** (Schnittstellenumsetzer) abstrahiert.
- EAI ist nichts anderes als eine **Middleware**, welche eine für die Unternehmung **homogene Plattform** schafft. Sie setzt zunehmend auf **standardisierte Lösungen** wie UDDI, SOAP, XMAL, J2EE und .NET.

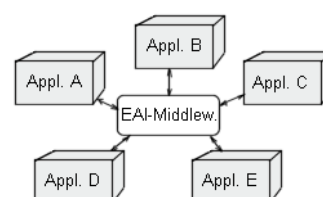
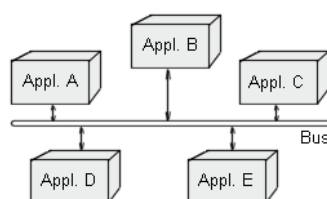


16.3. Enterprise Service Bus (SOA)

- Stellt eigentlich nur die **technische Grundlage zur Kommunikation** der verschiedenen Dienste (Services) in standardisierter Form zur Verfügung. Dies erfolgt normalerweise über die **Message Oriented Middleware (MOM)**, die den Bus bildet und über die der Nachrichtentausch über Standard-Protokolle läuft.
- ESB**: Nachrichten werden über Bussystem verteilt (Anbindung an den Bus über verteilte Software-Komponenten). Der ESB kann als **Fortsetzung von EAI** gesehen werden. Während bei EAI der Fokus eher auf der technischen Anbindung von Systemen und Applikationen liegt, unterstützt der ESB darüberhinaus als Grundlage für SOA eine agile Gestaltung der fachlichen Geschäftsprozesse ("Bei **EAI integriere** ich **hinterher**, bei **SOA** mache ich mir **vorher** Gedanken").

ESB: Nachrichten werden über Bussystem verteilt (Anbindung an den Bus über verteilte Software-Komponenten).

EAI: Da hier die Software-Komponenten nicht verteilt sind, kann es eher zu einem Flaschenhals kommen.

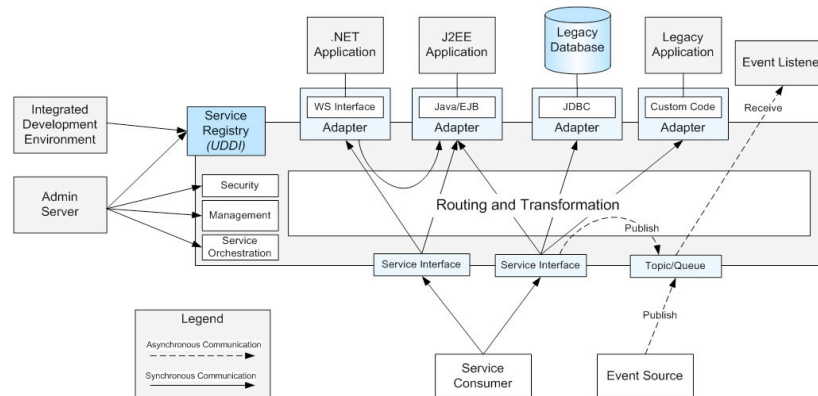


16.4. JBoss ESB

16.4.1. Rosetta

Der Kern der JBoss Middleware wird **Rosetta** genannt und hat **vier Komponenten**:

- **Message Listener** und **Message Filterung**
- Komponente für die **Transformation** von Daten (Daten in XML oder Java-Klassen)
- **Routing Service** (richtige Weiterleitung von Daten)
- **Message Repository** (hier werden diverse Meldungen verwaltet)



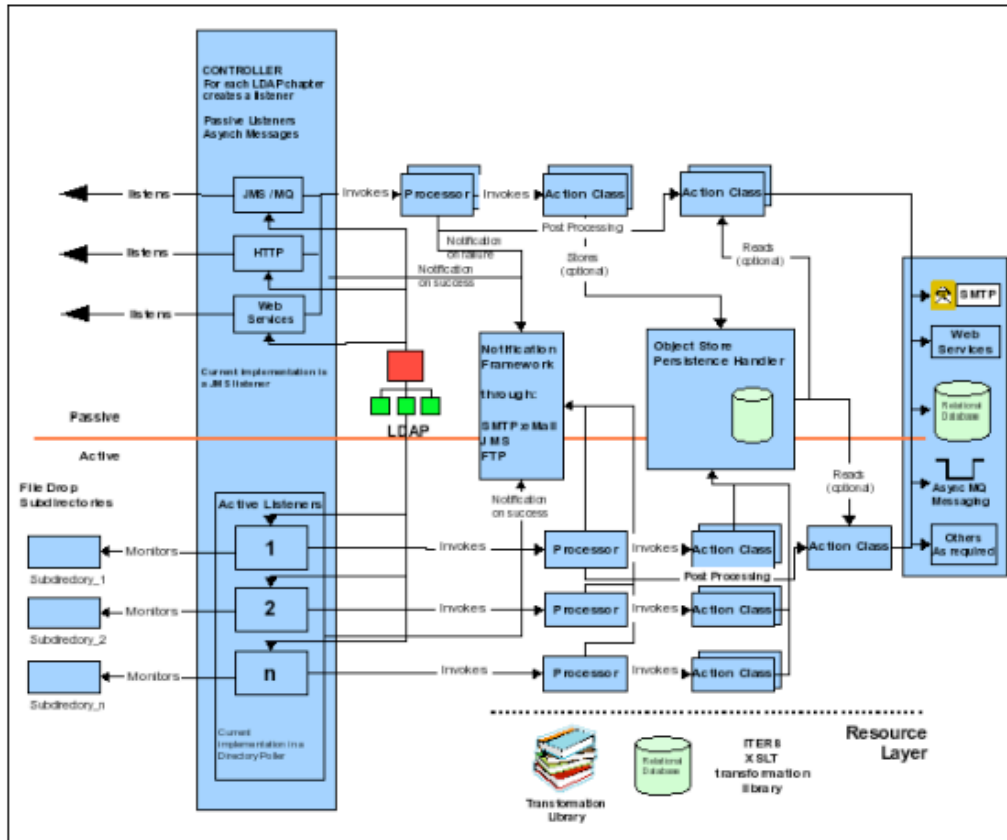
16.4.2. Services

- Ein **Service** besteht aus einer **Liste von Aktionen (Action Pipeline)** welche von der ESB Message bearbeitet wird.
- Ein **Service** hat **zwei Attribute**: **category** (WeatherServices) und **name** (Receiving)
Anhand der **ServiceInvoker-Instanz** kann clientseitig ein **Service aufgerufen** werden:
`ServiceInvoker invoker = new ServiceInvoker(„WeatherServices“, „Receiving“);`
Diese beiden **Attribute** werden dazu verwendet um sich **bei den Service-Endpunkten/Listener** zu **registrieren** (Listener empfängt request, extrahiert XML-Body, konvertiert in lokales Format und leitet die Nachricht schliesslich an die gewünschte Komponente weiter).
- Pro Service werden in der Regel zwei Listener definiert:
 - **Gateway Listener**: empfängt Nachrichten von **aussen**, **normalisiert** diese und gibt sie in die **Action Pipeline**.
 - **ESB Listener**: arbeitet **innerhalb** von **ESB** mit standardisierten Nachrichten und ermöglicht den **Austausch** zwischen den einzelnen **ESB Komponenten**.
- Beim Austausch von Nachrichten innerhalb der **gleichen Virtuellen Maschine (JVM)** kann auf die explizite Definition der Listener verzichtet werden. Jedoch muss **implizit** definiert werden:
`invmScope=„GLOBAL“`
- Damit Nachrichten empfangen werden können, braucht es einen **Provider** (JMS-, FTP-, oder HTTP-Provider). Dieser stellt die nötige **Infrastruktur** zur Verfügung und muss im **Listener verknüpft** werden.

16.4.3. Message

- Ein **Message-Objekt** besteht aus folgenden Teilen: **Header**, **Body**, **Attachments** und **Properties**.
- Im Header-Teil wird auf das Call-Objekt referenziert. Im Wesentlichen beinhaltet ein **Call-Objekt** folgende Informationen:
 - Wer sendet die Nachricht? (**From**)
 - Wer ist der Empfänger? (**To**) → obligatorisch

- Wer soll die Antwort erhalten? (**ReplyTo**)
- Wer soll informiert werden, falls Probleme auftauchen? (**FaultTo**)
→ alle diese Angaben sind optional bis auf To
- Mit dem **EPR (endpoint reference)** wird der **Endpunkt einer Kommunikation** definiert (Service, welcher die Nachricht sendet oder empfängt). → Definition von EPR's: ServiceCategory:ServiceName
- JBoss unterstützt zwei Typen von Messages: MessageType.JAVA_**SERIALIZED** und MessageType.JBOSS_**XML**



- JBoss ESB verwendet verschiedene Komponenten: **Listeners**, **Routers**, **Notifiers** und **Actions**.

16.4.4. Listeners

Der **Listener** kapselt den Endpunkt für den **Empfang** von Messages. Wird eine Nachricht empfangen, wird sie vom Listener dem **Prozessor** mit der entsprechenden Action Pipeline übergeben. Nach der letzten Aktion wird die resultierende Nachricht zum Endpunkt (Reply To) geleitet.

16.4.5. Routers

Die Aufgabe des **Routers** ist es die empfangene **Anfrage** an den entsprechenden Endpunkt zu **leiten**.

16.4.6. Notifiers

Der Notifier protokolliert was wo passiert und liefert somit über Erfolg oder Misserfolg bei der Bearbeitung einer Anfrage. Notifiers werden nicht von allen Providern unterstützt und müssen einfach sein (Array oder Zeichenkette).

16.4.7. Konfiguration

Die Konfiguration von JBoss erfolgt in der Datei jboss-esb.xml. Diese Datei enthält Angaben zu Provider (JMS, HTTP oder FTP) und Services (Angaben über Listener und Aktionen).

16.4.8. Erstellen von Nachrichtenschlangen

Nachrichtenschlangen (**Queues**) werden in der Regel von einem **Administrator** angelegt. Dies wird entweder in der Administration **Console** von JBoss oder in der destinations-service.xml gemacht.

16.4.9. Aktionen und Nachrichten

- Eine Aktion wird mit dem Eintreffen einer Nachricht ausgelöst. Diese **Aktion muss wissen** wo sich der **Nutzzinhalt** in der Nachricht befindet (**Body**, **Attachment** oder **Properties**).
- Da eine Nachricht von mehreren Aktionen bearbeitet werden kann, ist es möglich dass die **Nutzzinhalte** für die unterschiedlichen Aktionen an verschiedenen Orten abgelegt wurden.

16.4.10. Umgang mit der Nachricht

- Man kann auf zwei Arten definieren was mit der Antwort nach Bearbeitung der Anfrage passiert: **implizit** oder **explizit**.
- **Implizites response handling** basiert auf die Antworten der einzelnen Aktionen:
 - Wenn eine Aktion null zurück liefert wird keine Antwort gesendet.
 - Wenn die Aktion eine Antwort (no-error) liefert, dann wird diese an die EPR (ReplyTo) gesendet. Falls kein ReplyTo definiert wurde, wird an From gesendet.
- **Explizites response handling** basiert auf den Konfigurationen in der jboss-esb.xml:
 - Ist Action Pipeline als OneWay definiert wird keine Antwort gesendet.
 - Ist Action Pipeline als RequestResponse definiert wird an ReplyTo resp. From gesendet.
 - <actions mep="OneWay"> oder <actions mep="RequestResponse">

16.4.11. ServiceInvoker

- Das **Aufrufen eines Services** kann mittels der Klasse ServiceInvoker gemacht werden (Service wird **direkt** angesprochen ohne über Queues zu gehen).
- Der ServiceInvoker kann **innerhalb einer Aktion** aus der Action Pipeline als auch von **einem entfernten Client** verwendet werden.

16.4.12. Action-Klasse

- Enthält **Aktionen** welche **nach Eintreffen der Nachricht** ausgeführt werden.
- Eine solche Klasse kann **mehrere Methoden** haben, welche in der Action Pipeline aufgerufen werden.