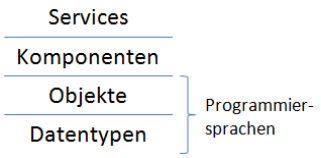
Komponentenbasierte Anwendungen – Grundlagen

SW-Architektur

- die Aufgabe von SW-Architektur ist die strukturelle Beschreibung von SW-Systemen
- Bausteine und Beziehungen zwischen Bausteinen
- präzise Bedeutung der Beschreibungselemente ist essentiell
- Skalierbarkeit: für kleine und große Systeme geeignet
- Bausteine: lokalisierte Beschreibung eines Systemteils
- Beziehungen zwischen Bausteinen: Benutzt / IstTeilVon

Objektbasierte Systeme

- Konzepte
  - Objekt-basierte Sicht (nicht: alles ist ein Bitstring)
  - Klassen als Typbeschreibung von Objekten
  - Komponenten als definierte Systembestandteile
  - Schnittstellen als definierte Übergänge



- Entwicklung: Sichten zur Beschreibung von SW-Eigenschaften

Primitive Datenstrukturen

- Wie werden Daten repräsentiert?
- zunächst: Datenstruktur
  - Variablen oder Konstanten, besitzen einen Wert und einen Identifizierer
- Veränderungen eines Werts mittels Zuweisungen oder komplexer Manipulationen
- die Werte einer Datenstruktur sind flach
  - Integer, natürliche Zahlen, boolesche Werte, Aufzählungen
- Systeme enthalten komplexere Daten
  - primitive Datenstrukturen als Bausteine
  - Zusammenfassung in Objekten

Objekte

- Komposition
  - jedes Objekt hat einen eindeutigen Bezeichner
  - verschiedene Referenzen können auf ein Objekt verweisen
  - benannte Attribute bezeichnen Werte
  - Verweise (Referenzen) bezeichnen u.U. gleiche oder identische Objekte
  - Regeln für Beziehungsgeflecht
    - Zyklen konzeptionell nicht erwünscht
    - kein Objekt darf sich indirekt oder direkt selbst benutzen
    - Bildung von Hierarchien
  - gemeinsame Objekte in azyklischen Beziehungen
- Kapselung
  - die Bindung von Operationen an eine Datenstruktur definiert Schnittstelle
    - Zugriff nur über objektspezifische Operationen
    - ein Objekt kapselt Datenstrukturen mit Hilfe von Operationen
  - Operationen stellen Integritätsbedingungen sicher
  - Objekt ist nur durch objektspezifische Operationen inspizierbar und manipulierbar
  - Operationsausführung können Ausnahmen hervorrufen

Klassen

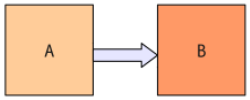
- viele gleichartige Objekte zur Laufzeit
  - strukturelle Eigenschaften (Attribute, Operationen, Ausnahmen)
  - Wertebereiche und Zustände
- Beschreibung unabhängig von einzelnen Instanzen notwendig
  - Daten-Eigenschaften
  - Schnittstelle der Kapselung für Objektzugriff
- Klassen als Beschreibung gleichartiger Objekte
  - Spezifikation eines SW-Systems definiert und beschreibt alle Eigenschaften von Objekten eines Systems
  - mögliche Strukturen der Objektbeziehungen müssen sich in statischer Beschreibung des Systems explizit widerspiegeln
- die Klasse eines Objektes gibt die für alle Objekte dieser Klasse verbindlichen Eigenschaften an
- beschrieben werden invariante Eigenschaften
  - gelten während der gesamten Lebensdauer eines Objekts
- Klassen als Bausteine für das Bilden von SW-Architekturen
- Objekte sind immer exakt einer Klasse zugehörig
- Objekte erfüllen zur Laufzeit eine bestimmte Aufgabe
  - Zusicherung von invarianten Eigenschaften
- Klassen definieren Werte, die einem Objekt in Form der Datenstruktur eingekapselt sind
  - d.h.: zur Definition invarianter Eigenschaften Definition legaler Werte („Wertebereich“) des eingekapselten Datenobjekts
  - bei komplexen Objekten Definition der entsprechenden Konstruktion von Werten
- Kapselung
  - Definition des zulässigen Verhaltens
    - Verhalten wird durch Operationen realisiert
    - Operationen eines Objektes sichern die Einhaltung des zulässigen Verhaltens zu
    - Einkapselung ermöglicht diese Zusicherung
    - die Konsequenz in der Typdefinition sind entsprechende Definitionen
      - wie das Verhalten von Operationen ausfallen darf
      - welche Werte innerhalb des zugehörigen Objekts damit erzeugt werden
  - Einschränkung aktueller Systeme
    - Wertebereichsprüfung nur bei vordefinierten Typen automatisch
    - aber beliebige Prüfung in Operationen möglich
  - Klassen beschreiben die Kapselung in Objekten enthaltener Werte
    - sichtbare Zustände → Objektattribute
    - verfügbare Services → Operationen der Objekte
    - Interaktionen → Ausführungsanforderungen an Objekt-Operationen
    - Ausfall von Services → Ausnahmen
  - Attribute
    - haben einen Namen und einen Typ
    - können einen Objekttyp und einen Nicht-Objekttyp (primitiven Typ) besitzen
    - statischer Typ eines Attributs schränkt die Werte ein, die dieses zur Laufzeit annehmen kann
    - Regeln für Zugriff von außen → entsprechende get-/set-Operationen

- Operationen
  - besitzen eine Signatur
    - Name, Liste von In- (Standard in Java), Out- (leerer Platzhalter, z.B. in CORBA), Inout-Parametern (Referenz), Typ des Rückgabewertes, Liste möglicher Ausnahmen
  - Ausnahmen
    - Durchführung von Ausführungsanforderungen kann scheitern
    - Ausnahmen erhalten erklärende Informationen bzgl. des Ausfalls
    - Ausnahmen können generisch (z.B. Netzwerkfehler) oder spezifisch sein
- Vererbung
  - Ziel: Eigenschaften, die in gleicher Weise in unterschiedlichen Typen definiert werden, sollten nur einmal aufgeschrieben werden
  - Umsetzung: Typhierarchie
  - Untertypen erben Attribute, Ausnahmen und Operationen
  - Untertypen können ererbte Eigenschaften neu definieren
  - Mehrfachvererbung: ein Objekt-Typ kann mehr als einen Supertyp besitzen
    - nicht in allen Middleware-Konzepten unterstützt
    - kann einige Mehrdeutigkeiten hervorrufen (müssen eindeutig geklärt werden)
  - Polymorphismus: Variablen können Werte annehmen, die durch die Untertypen des Typs der Variablen gegeben sind

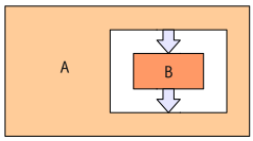
Komponenten

- große Systeme definieren große Anzahl von Klassen
- Gliederung nach Aufgaben sinnvoll
  - Zusammenfassung von Gemeinsamkeiten
  - Verstecken einzelner Bauteile
  - Kapselung aller Bestandteile durch gemeinsame Schnittstelle nach außen
- Komponenten erlauben flexiblere Gestaltung großer Systeme
  - Wartbarkeit und Stabilität durch Zugriff über Schnittstellen
  - Wiederverwendbarkeit
  - ideal: Austauschbarkeit mittels Instanzen von Komponenten
- mögliche Probleme
  - Verwendung an verschiedenen Stellen impliziert Annahmen
- Komponenten müssen die Einhaltung von Invarianten sicherstellen
  - erfüllen somit eine bestimmte Teilaufgabe innerhalb der Architektur
  - Veränderung einer Komponente während der Entwicklung führt dazu, dass sie andere invariante Eigenschaften sicherstellt
- Schnittstellen
  - Schnittstellen als Verbindung zwischen Komponenten
  - Export
    - Kapselung des Inhalts nach außen
    - Ein- und Ausgabedaten
    - „Vertrag“ definiert Leistung und Anforderungen für Verwendung dieser Schnittstellen
  - Import
    - Voraussetzungen der Komponente
    - Export anderer Komponenten wird benötigt

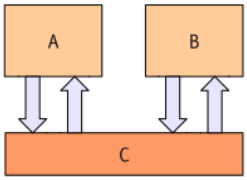
- Komposition
  - Systeme bestehen aus mehreren Komponenten
  - verschiedene Möglichkeiten der Komposition
    1) Benutzt-Beziehung: „A benutzt B“
       - resultierendes System Verbindung der Komponenten A und B
       - Verbindung entlang der Schnittstellen von B und A
       - Schnittstelle und Invarianten von A unverändert
       - B wird von A für gewünschte Leistung benötigt
       - gegenseitige Benutzung möglich („B benutzt auch A“)



    2) Einbettung: „A bettet B ein“
       - A greift auf B zu
       - B kann auch auf A zugreifen
       - nach außen nur Schnittstellen und Invarianten von A



    3) Kommunikation: „A kommuniziert mit B über C“
       - lose gekoppelte verteilte Systeme
       - C ist Middleware für Kommunikation
       - A und B kommunizieren indirekt



- Parametrisierung
  - Erhöhung der Flexibilität von Standardkomponenten
    - Trennung der allgemeinen (d.h. Parameterteile) von den speziellen Aspekten einer Komponente
    - unterschiedliche Flexibilität, unterschiedlicher Einfluss auf die Architektur
  - Ein-/Ausgabeparameter
    - formale / aktuelle Parameter von Operationen
    - Einfluss auf Komponentenentwicklung
    - Aktualisierung bei Systemausführung
    - sichtbar innerhalb der Ausführungsanforderung
  - Komponentenparameter
    - Verbindung zu anderen Komponenten
    - Anforderungen einer Komponente an ihre „Umgebung“
    - Einfluss auf Komponenten- und Architekturentwicklung
    - Aktualisierung bei Komponentenkonfiguration/-integration
    - sichtbar für betroffene Komponente
    - Ergebnis: statische Modulstruktur
      - Ko-Existenz von Komponenten im System
      - mögliche Verbindungen zwischen Komponenten
  - Systemparameter
    - spezielle Parameter für gesamte Komponentenkonfiguration eines Systems
    - Platzhalter für Eigenschaften, die für das ganze System gelten und erst zum Zeitpunkt des Systemstarts angegeben werden müssen
    - Einfluss auf Komponenten- und Architekturentwicklung
    - Aktualisierung bei Systemstart/-konfiguration
    - sichtbar für alle Komponenten

Pakete

- große Zahl von Komponenten möglich
- Komponenten als Bausteine der Systemfunktion
  - Systementwicklung kann diese Bausteine anders aufteilen oder zusammenfassen
  - systematische Gliederung der Implementierungsarbeit nötig
- daher: Gruppierung zu Paketen
  - Fokus: Entwicklungsbereiche, SW-Organisation
  - gruppiert inhaltlich, funktional oder organisatorisch zusammenhängender Klassen
  - hat häufig strukturelle Ähnlichkeit mit der Komponentensicht

Deployment

- Laufzeit-System als konkrete Konfiguration von Komponenten
  - lauffähige Bestandteile, austauschbare Archive, Bibliotheken
- Darstellung des Deployments
  - Artefakte, Ausführungsumgebungen, „Stückliste“ der physischen Auslieferung eines Systems

Fazit

- Objekt-basierte Sicht
- Klassen als Typbeschreibung für Objekte
- Komponenten als definiert Systembestandteile
- Schnittstellen als definierte Übergänge
- Pakete und Deployment zur Laufzeit

Komponentenbasierte Anwendungen – Sichten und Modelle

Sichten und Modelle

- Design von SW-Systemen beinhaltet verschiedene Aspekte
- Grundlage ausführbarer Systeme ist Quellcode → Abstraktion notwendig
- Strukturierung durch Sichten und Modelle
  - Betrachtung aus verschiedenen Blickwinkeln
  - Ausblendung „unnötiger“ Aspekte
  - Methoden manipulieren die Elemente ihrer zugehörigen Sicht
  - alle Aspekte ergeben ein konsistentes Gesamtsystem
- Klassifizierung
  - Betrachtung statischer Strukturen
  - Betrachtung dynamischer Strukturen (Ausführung, Abläufe)
- Anwendungszeitpunkte
  - Anforderungserhebung, Grob- und Feinentwurf, Implementierung, Validierung und Verifikation, Dokumentation für Nutzer
- Ausgangspunkt Objekte
  - Typen, Werte, Operationen
  - detailliert, nahe an der Implementierung
  - Beziehungsgeflecht zwischen Objekten
- Bezug Komponenten
  - Modell funktioniert als Komponente (implementiert also Funktionalität)
  - Modell koordiniert Komponenten (z.B. Prozessmodelle)

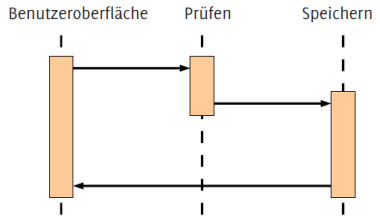
- Nutzung
  - Modell-basierte Entwicklung (eher als Dokumentation, Beschreibung)
  - Modell-getriebene Entwicklung (Entwicklung abgeleitet aus Modellen)
- Abstraktion
  - verschiedene Ebenen existieren
  - Modelle abstrahieren von Programmcode (bzw. von einigen Aspekten davon)
- Überbrückung unterschiedlicher Abstraktionsebenen notwendig
- Möglichkeit 1: Quellcode als Ziel
  - imperative Ausführungslogik
  - manuell: Modell wird vom Entwickler von Hand umgesetzt
  - automatisch: Werkzeuge generieren Code aus Modell
- Möglichkeit 2: Werkzeuge führen Modellspezifikation aus
  - Vorteil: nur *eine* konsistente Beschreibung
- Probleme (bei beiden Möglichkeiten)
  - echte Systeme häufig komplexer als spezielle Modelle
  - Integration von abgeleitetem Code in Rest des Systems
  - Integration ausführbarer Modellspezifikationen in Rest des Systems
- wichtig: System ist die Summe aller spezialisierten Sichten!

Modelle für Komponenten und Schnittstellen

- bereits eingeführt: Gruppierung von Objekten zu Komponenten
  - Abstrahierung von Klassen- und Objektdetails
  - Gliederung in funktionale Bausteine
  - abstrakter als Klassen, nicht direkt implementierbar
  - idealerweise: Betrachtung unabhängig vom Verwendungszweck
- Fokus: Interaktion
  - statische Betrachtung
  - Schnittstellen als „Vertrag“
  - Vor- oder Nachbedingungen
  - Garantie von Zuständen von Komponenten
- Abhängigkeit von Komponenten
  - komplexe Zustandsbeschreibungen
  - Seiteneffekte von Änderungen
- formale Spezifikation (extrem schwierig und aufwändig)
  - statische Betrachtung
  - Fokus: mathematische/logische Aussagen über Systemzustände und Ein-/Ausgabe-Relationen
  - automatisch auswertbar, beweisbare Programmeigenschaften
  - unabhängig von der Implementierung
- natürlichsprachliche Beschreibung einer Schnittstelle
  - „Wenn die Studiengebühr bezahlt wird, kann ein Student erfolgreich für einen Kurs angemeldet werden. Die Zahl der angemeldeten Studenten steigt dann genau um 1.“
- formale Spezifikation dieser Schnittstelle (hier im Beispiel mit OCL)
  - context Course::registerStudent(s : Student) : Boolean
    pre: s.tuitionPaid = true
    post: result = true and
    self.students->size() = self.students@pre->size()+1

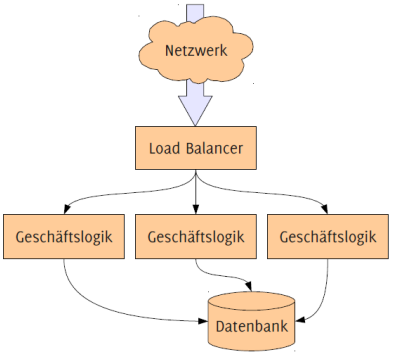
Modelle für Abläufe

- Interaktion von Komponenten können sehr komplex sein
  - beteiligte Komponenten und Schnittstellen nicht erkennbar
  - Implementierung kann Abläufe in mehrere Schritte untergliedern
- explizite Modellierung von Abläufen
  - Betrachtung dynamischer Aspekte der Interaktionen
  - Fokus: (erlaubte) Reihenfolgen von Operationsaufrufen
  - steht im Bezug zu Vor- und Nachbedingungen
  - macht Vorgaben für die Implementierung



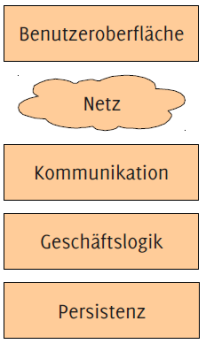
Modelle für Verteilung

- Parallelität ist nur bezogen auf Operationen
  - verteilte Systeme
  - logische oder physische Verteilung
- Darstellung verteilter Systeme
  - statische Strukturen
  - Fokus: logische und physische Verteilung, Aufgabenverteilung
  - relevant für Performanz und Lastverteilung
  - machen Vorgaben für benötigte Kommunikationswege



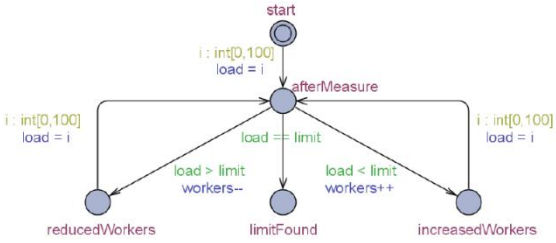
Modelle für Schichten

- große Anwendungen können in Schichten unterteilt werden
  - Strukturierung nach Aufgaben
  - Schichten als Kapselung
- typische Schichten
  - Benutzeroberfläche
  - Kommunikation
  - Kapselung der Funktionalität
  - Geschäftslogik
  - Persistenz



Modelle für Automaten

- Systemzustände verändern sich zur Laufzeit
  - System darf nicht in undefinierte Zustände gelangen
  - ein Systemzustand hat nur eine definierte Menge von Folgezuständen
- Automaten als explizite Abbildung der Zustände
  - Zustandsübergänge dynamisch
  - Fokus: Zustände, Zustandswechsel
  - formal verifizierbar durch temporale Logik
  - anschaulich simulierbar
  - auch möglich: parallele Automaten

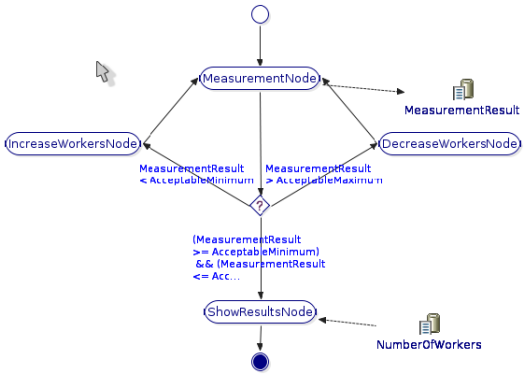


- treibt die Implementierung nicht voran, dient nur der Beschreibung von Eigenschaften

Modelle für Prozesse

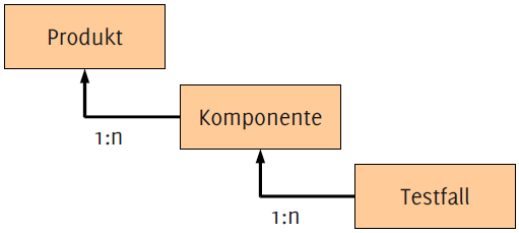
- Herkunft: Geschäftsprozesse
- Erweiterung von Zustandsautomaten

- Fokus auf Prozessabläufe
  - Entscheidungen
  - Aktionen als Knoten
  - Transaktionen
  - Artefakte als zu verarbeitende Daten und Informationen
- Integration
  - Verwendung von Teilsystemen
  - Prozesse als Koordinator



Datenmodelle

- fast jede Anwendung verwaltet Daten
- Daten sollen Realität sinnvoll abbilden
  - Typen, Eigenschaften
  - Beziehungen, Hierarchie
- Modellierung als Mittel zum Datenaustausch
  - Ablegen in DBs, Datenaustausch und Integration
- Technologien
  - ERM für relationale DBs, Objektmodelle, Ontologien für Wissensmodelle und KI



Modelle für Services

- Grenzen für Komponentenmodelle
  - Plattformen, physikalische Systeme, physikalische Verteilung
- Integrationsmechanismen notwendig
- Service-orientierte Architekturen (SOA)
  - Teilsysteme bieten Services mit definierter Schnittstelle an
  - Services als Komponenten einer SOA
- Mechanismen z.B. ESB, Prozesse

Komponentenbasierte Anwendungen – Verteilungen

Abgrenzung: zentrale Organisation

- bisher bekannt und betrachtet: monolithische Systeme
- Eigenschaften
  - das Rechensystem besteht aus einer (meist undurchsichtigen) Komponente
  - diese Komponente wird von allen Benutzern gemeinsam über die gesamte Zeit genutzt
  - alle Ressourcen sind verfügbar
  - die SW wird in einem Prozess betrieben
  - ein einzelner Kontroll-/Ausfallpunkt

verteilte Systeme

- Definition
  - ein verteiltes System ist eine Sammlung von autonomen Rechnern
  - diese Rechner sind durch ein Netzwerk verbunden
  - jeder Rechner führt Komponenten aus
  - diese Komponenten arbeiten auf der Basis einer (Verteilungs-)Middleware
  - die Middleware als Vermittlungsschicht lässt dem Benutzer das gesamte System als ein homogenes Rechnersystem erscheinen

- Eigenschaften
  - mehrfache autonome Komponenten
  - Komponenten werden nicht unter den Benutzern geteilt
  - Ressourcen können auch nicht zugreifbar sein
  - Komponenten werden nebenläufig ausgeführt
  - mehr als ein Kontroll-/Ausfallpunkt
- Beispiele für Middleware
  - Transaktions-orientiert: IBM CICS, BEA Tuxedo
  - Nachrichten-orientiert: Microsoft Message Queue, Sun Tooltalk, Tibco Rendezvouz
  - prozedural: Sun ONC, Linux RPCs
  - Objekt-orientiert: OMG CORBA, Sun Java/RMI, Microsoft COM, Sun Enterprise Java Beans

Heterogenität

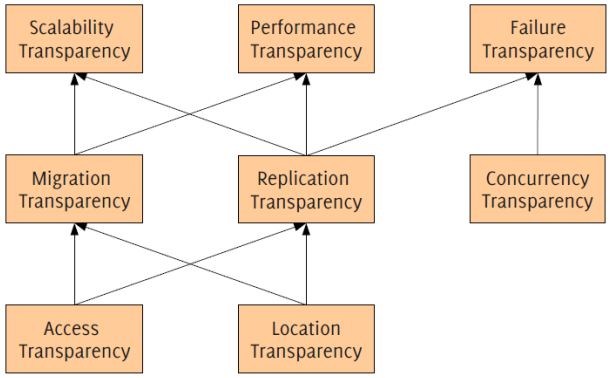
- Heterogenität von
  - **HW-Plattformen, Betriebssystemen, Netzwerk, Programmiersprachen, Komponenten**
- Komponenten
  - Integration existierender Komponenten
  - existierende Komponenten oft Black-Box
    - Altsysteme (Legacy-Komponenten)
    - „Commercial-of-the-Shelf“-Komponenten
  - Gründe für Integration
    - vorhandene Systeme
    - Erweiterung um zugekaufte IT-Infrastruktur
  - kein Reengineering bzw. Modifikation möglich
    - COTS sind proprietär und können i.d.R. nicht modifiziert werden
    - Technologie und Wissen zur Modifikation von Altsystemen nicht mehr vorhanden
- Objekttechnologie
  - Objekte kapseln (wrap) Black-Box-Komponenten
  - Erhöhung der Abstraktionsebene
  - Lösung für das Problem der Heterogenität der Umgebung
  - Skalierbarkeit

Anforderungen an verteilte Systeme (nicht-funktionale Eigenschaften)

- **Teilung der Betriebsmittel**
  - Möglichkeit jede HW/SW/Daten in dem verteilten System zu benutzen
  - Resource-Manager
    - kontrollieren Zugriffsrechte und nebenläufigen Zugriff
    - bieten Namensschemata
  - gemeinsames Benutzungsmodell regelt
    - welche Ressourcen angeboten werden
    - wie sie benutzt werden können
    - wie Anbieter und Nutzer der Ressource interagieren müssen
- **Offenheit**
  - Offenheit bezieht sich auf Erweiterbarkeit und Verbesserung eines verteilten Systems
  - Schnittstellen müssen publiziert werden
  - Integration alter und neuer Komponenten
  - Unterschiede in Darstellung und Präsentation müssen ausgeglichen werden

- **Nebenläufigkeit**
  - Komponenten in verteilten Systemen werden in nebenläufigen Prozessen ausgeführt
  - Komponenten greifen u.U. verändernd auf gemeinsame Ressourcen zu (Variablen, DBs)
  - Verlust der Integrität: Lost Update, Inkonsistenz
- **Skalierbarkeit**
  - Anpassung eines verteilten Systems an
    - Vergrößerung der Benutzerschaft
    - Verbesserung der Antwortzeiten
  - Erweiterung der Prozessorkapazität (mehr und schnellere Prozessoren)
  - Komponenten sollten nicht geändert werden müssen, um Skalierung zu erzielen
  - muss im Design der Komponenten berücksichtigt werden!
- **Fehlertoleranz**
  - alle Komponenten versagen potentiell ihren Dienst (HW/SW/Netzwerk)
  - verteile Systeme haben die Möglichkeit, die Zuverlässigkeit auf allen Ebenen zu erhöhen
  - Verbesserung der Zuverlässigkeit: Recovery, Redundanz
- **Transparenz**
  - der Verteilungsaspekt sollte verborgen bleiben gegenüber Benutzern und Entwicklern
  - Transparenz hat verschiedene Dimensionen
  - orientieren sich an den verschiedenen Eigenschaften verteilter Systeme

Transparenzen



- **Zugriff zu Ressourcen**
  - ermöglicht gleichförmigen Zugriff auf lokale und entfernt liegende Ressourcen
  - Entwicklung: Middleware macht Komponenten entfernt verfügbar
  - Beispiele: entfernte Methodenaufrufe in Java (RMI), Filesysteme (NFS), Navigation im WWW, SQL-Anfragen
- **Ort-Transparenz**
  - Zugriff auf Ressourcen/Objekte ohne die genaue Kenntnis ihres Ortes
  - Entwicklung: Middleware macht Komponenten automatisiert auffindbar
  - Beispiele: Verzeichnisse für Web-Services, Filesysteme (NFS), Navigation im WWW, Tabellen in verteilten DBs
- **Nebenläufigkeit**
  - ermöglicht nebenläufige Ausführung: mehrere Prozesse, Benutzung gemeinsamer Prozesse
  - Entwicklung
    - Middleware führt Komponenten parallel aus
    - Kommunikation findet zwischen Komponenten statt
  - Beispiele: virtuelle Maschinen und Threads, NFS, DBMSs

- **Replikation**
  - ermöglicht die Benutzung mehrerer Kopien eines Objektes ohne um die Kopien zu wissen (Zuverlässigkeit, Antwortzeiten)
  - Entwicklung: Benutzung der Middleware statt expliziter Verwendung von Kopien
  - Beispiele: Caches, Spiegelungen von Webseiten
- **Fehlertoleranz**
  - ermöglicht das Verstecken von Fehlerzuständen
  - trotz des Auftretens von Fehlern kann die gewünschte Aufgabe ausgeführt werden
  - Entwicklung
    - Komponenten behandeln ihre Fehler selbst
    - Middleware stellt fehlertolerante Komponenten zur Verfügung
  - Beispiele: DBMSs
- **Migration**
  - Objekte können an andere Orte bewegt werden
  - Benutzung des Objekts wird nicht beeinträchtigt
  - Entwicklung: Benutzung der Middleware statt expliziter Definition von Orten
  - Beispiele: NFS, Webseiten
- **Leistung**
  - ermöglicht (automatische) Lastverteilung
  - Entwicklung: Middleware unterstützt Leistungsverteilung
  - Beispiele: Serverfarmen, Cluster von Java Application Servern
- **Skalierbarkeit**
  - Erweiterbarkeit ohne Unterbrechung des Betriebs und der Umstellung der Benutzung
  - Entwicklung: Verwendung einer Middleware, die das Einbinden von neuen Plattformen erlaubt
  - Beispiele: Webseiten, verteilte DBs, Cluster von Java Application Servern

Middleware für (Objekt)Verteilung – Prinzipien

vernetzte Rechensysteme

- ISO/OSI-Referenzmodell, hier im Fokus: Transport-Schicht (Ebene 4)
- zwei Varianten
  - Transmission Control Protocol (TCP)
    - bidirektionaler Fluss von Zeichen (bytes) zwischen zwei Komponenten
    - verbindungsorientiert
    - zuverlässig aber langsam
    - Pufferung entkoppelt die ggfs. unterschiedlichen Geschwindigkeiten der verteilten Komponenten
    - Anwendung: OO-Middleware, ssh, VNC
  - User Datagram Protocol (UDP)
    - unidirektionaler Fluss von Zeichen (bytes) zwischen zwei Komponenten
    - verbindungslos
    - Nachricht enthält Adressierung des Empfängers
    - unzuverlässig aber schnell
    - Nachrichtenlänge begrenzt
    - Pufferung obliegt dem Empfänger
    - Anwendung: Streaming

Middleware für Verteilung

- Abgrenzung: direkte Benutzung des Netzwerks
  - manuelle Abbildung von komplexen Anforderungen auf Zeichenfolgen
  - manuelle Auflösung der Heterogenität
  - manuelle Implementierung der Komponenten-Aktivierung
  - keine Typsicherheit (Zeichenfolgen)
  - manuelle Synchronisation der beteiligten Komponenten
  - keine QoS-Garantien
- Nutzen von Middleware
  - Schicht zwischen Netzwerk/OS und der Applikation
  - erhöht die Verteilungstransparenz
  - löst die Heterogenität bzgl. HW, OS, Netzwerken, Programmiersprachen
  - liefert eine Umgebung für die Entwicklung und Laufzeit von verteilten Systemen

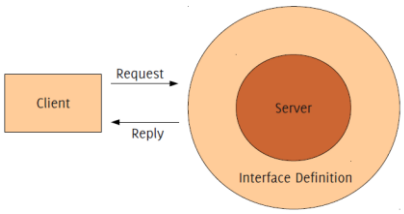
Remote Procedure Call (RPC)

- ermöglicht Prozedur-Aufrufe über Rechnergrenzen hinweg
- Schnittstellen werden mit einer speziellen IDL beschrieben
- RPC-Compiler generiert die notwendigen Rahmen für die Implementierung (Presentation / Session Layer)
- Beispiel

```
const NL=64;
struct Player {
   struct DoB {int day; int month; int year;}
   string name<NL>;
};
program PLAYERPROG {
   version PLAYERVERSION {
      void PRINT(Player)=0;
      int STORE(Player)=1;
      Player LOAD(int)=2;
   } = 0;
} = 105040;
```

- auf Ebene der Presentation Layer
  - Marshalling: Daten für die Übertragung aufbereiten
  - Unmarshalling: Daten aus der Übertragung wiederherstellen
- Stubs
  - Code für das Marshalling und Unmarshalling schwierig und fehleranfällig
  - Code kann automatisch generiert werden
  - wird in sog. Stubs im Client eingebaut
  - Server und Client werden der jeweils anderen Seite durch entsprechende Stubs repräsentiert
  - Stubs erzielen Typsicherheit
  - Synchronisation kann in Stubs erfolgen
- Synchronisation
  - Ziel: Methodenaufruf und Object-Request sollen möglichst ähnlich (nach außen) funktionieren
  - Stubs leisten:
    - Client Stub sendet die Aufforderung und wartet bis der Server fertig ist
    - Server Stub wartet auf die Aufforderung und ruft den Server auf, wenn die Aufforderungsdaten ankommen

- Typsicherheit
  - Ziele … wie kann sichergestellt werden, dass
    - Server die gewünschten Operationen ausführen kann
    - die aktuellen Parameter des Client den formalen Parametern des Server entsprechen
    - Ergebnistypen dito
  - Middleware arbeitet als Mediator zwischen Client und Server um diesen Aspekt der Typsicherheit zu erzielen
  - wird durch entsprechende Definition von für beide Seiten verbindlichen Interfaces erreicht



Server nach außen hin versteckt,
Kapselung durch Interface Definition

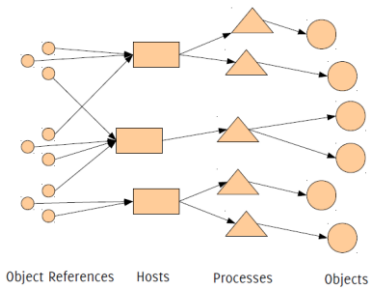
- Session Layer realisiert
  - Identifizierung/Aktivierung von RPC-Servern
  - Verteilung der Aufrufe an die richtige Stelle
  - Middleware verbirgt die Transportschicht

objektorientierte Middleware

- Eigenschaften
  - jede OO-Middleware hat eine IDL
  - zusätzlich zu RPC Unterstützung für
    - Objekttypen als Parameter
    - Ausnahmen und deren Behandlung
  - OO-Middleware realisiert
    - Generierung von Client & Server Stubs
    - Session & Presentation Layer
- Beispiel für IDL (Beschreibung, *was* Komponente macht, nicht *wie*)

```
interface Player : Object {
   typedef struct _Date {
      short day; short month; short year;
   } Date;
   attribute string name;
   readonly attribute Date DoB;
};
interface PlayerStore : Object {
   exception IDNotFound{};
   short save (in Player p);
   Player load(in short id) raises(IDNotFound);
   void print(in Player p);
};
```

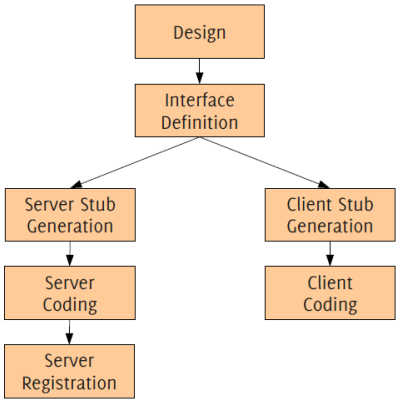
- Realisierung Session Layer



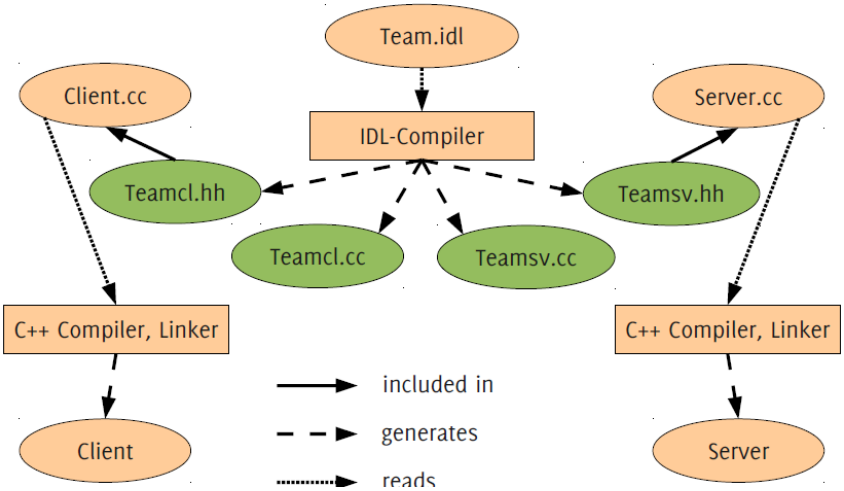
- Realisierung Presentation Layer
  - zusätzlich zu RPC
    - Definition einer Transport-Repräsentation für Objekt-Eigenschaften und -Referenzen (IDL)
    - Ausnahmebehandlung (Fehler sind wieder Objekte)
    - Marshall-Behandlung von ererbten Attributen
  - OO-Konzepte (Semantik, Schnittstellen, Zugriff) werden weiterverwendet
  - relevant für Entwicklung: Stubs, Transparenzen

Entwicklung mit OO-Middleware

- Vorgehen



- Erzielen der Zugriffstransparenz
  - Client Stubs haben die gleichen Operationen wie Server Stubs
  - daher
    - im Client wird lokaler Methoden-Aufruf codiert (zum Client Stub)
    - oder lokaler Aufruf zum Server
    - ohne den Aufruf zu ändern
  - Middleware kann die Kommunikation optimieren, wenn lokale Objekte benutzt werden (Umgehung des Stubs)
- Ort-Transparenz
  - Objekt-Identität, Objekt-Referenzen
  - Client-Aufrufe werden durch Objekt-Referenzen vermittelt
  - keine Information im Client notwendig bzgl. des physischen Ortes
  - Problem: Wie kommt man an die Objekt-Referenzen?
- Stub-Generierung und Client-/Server-Realisierung



hh = Header,
cc = Implementierung
Beispiel in C, in Java nur 1 Datei

- Typ-Sicherheit
  - Alternativen
    - Erweiterung einer Klasse
    - Implementierung eines Interfaces
- Server Registrierung
  - Objekt-Adapter müssen den Server finden und starten können
  - Server-Objekte werden in einem entsprechenden Repository registriert (ist Produkt- und Middleware-spezifisch)
  - Objekt-Adapter sucht für die Objekt-Aktivierung im Repository

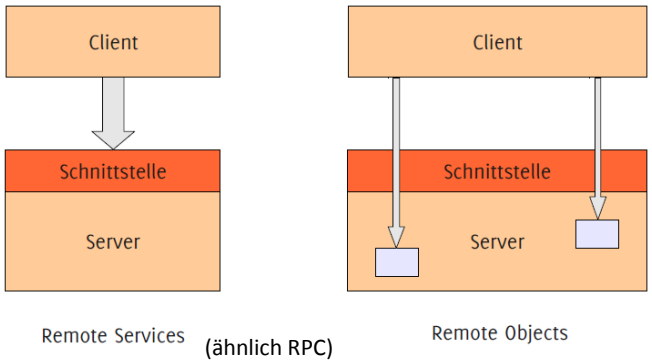
Zusammenfassung

- Middleware realisiert Transport-Layer
- unterschiedliche Formen von Middleware verfügbar
- OO-Middleware basiert auf IDLs
- OO-Middleware realisiert Session & Presentation Layer
- Presentation Layer Implementierung durch Client/Server Stubs wird aus IDL generiert
- Session Layer wird in den Objekt-Adaptern realisiert

Middleware für (Objekt)Verteilung – Java RMI

Grundlagen Remote Method Invocation (RMI)

- Ausgangspunkt Java VM
  - geschlossene Welt in Bezug auf Objekte und Datenaustausch
  - Schnittstellen nach außen explizit notwendig
- RMI ermöglicht Kommunikation zwischen JVMs
  - Netzwerkmechanismen, Socket-basiert, Protokoll, Rechnergrenzen sind nicht relevant
- enge Integration mit Java (→ Entwickler kann in seiner gewohnten Welt arbeiten)
  - minimiert die Änderungen an der Sprache Java und der VM
  - arbeitet in einer homogenen (Java-)Umgebung
- Java-Objektmodell wird verwendet
  - Interfaces und Remote Objects, Klassen, Attribute, Operationen, Ausnahmen, Erweiterung/Vererbung
- Funktionalität

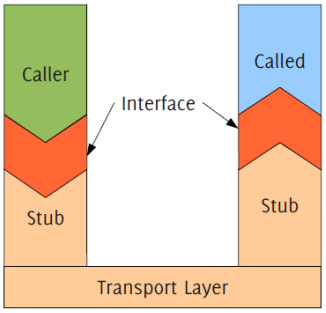


Remote Services (ähnlich RPC) Remote Objects

Remote Services

- Schnittstelle
  - Ziel: Dienst wird veröffentlicht
  - Grundlage: Netzwerk
    - Protokoll TCP, Socket „lauscht“, Clients können sich verbinden

- Datentaustausch auf Java-Ebene
  - Protokoll RMI, Objekte werden serialisiert (Session Layer), Schnittstelle wird angesprochen (Presentation Layer)
- Schnittstellenbeschreibung notwendig (IDL)
  - angebotene Services sind Methoden (wie bei RPC)
  - angebotene Methoden müssen beschrieben werden
- Heterogenität
  - Java löst Heterogenität bereits teilweise auf: HW, OS
  - aber: Objektmodell ist gegeben
  - Einschränkung: Zugriff innerhalb der Java-Welt
- Anforderung
  - Beschreibung der Schnittstelle
  - keine Beschreibung der Implementierung
  - enge Bindung an Java
- Konzept bereits vorhanden → Interfaces
  - Entkopplung von Beschreibung und Implementierung (auch räumlich möglich)
  - daher keine separate IDL
- vordefiniertes Interface *Remote*
  - Service-Interface erweitern *Remote*: "Remote Interfaces"
  - Klassen der Service-Implementierungen implementieren Service-Interface: "Remote Classes"
- Lookup
  - Services werden unter Namen veröffentlicht
  - Client erhält bei Angabe des Namens eine Referenz



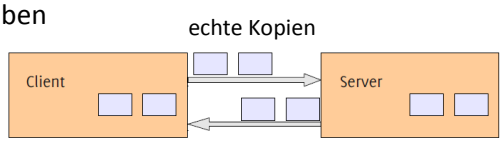
- Beispiel für Implementierung

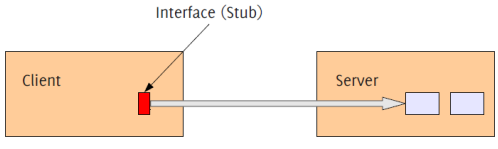
Package Name
Interface Name
Deklaration als Remote-Interface
definiert keine Methoden, dient der Typsicherheit

```
import java.rmi.Remote;
package persons;
interface PersonManagement extends Remote {
   Person create(String name) throws RemoteException;
   Person[] getAll() throws RemoteException;
   Person[] getByName(String n) throws RemoteException;
   void delete(Person p) throws RemoteException;
};
```

Remote Operations
Remote Exceptions
für alle Methoden notwendig!

- Signaturen
  - Parameter-Übergabe – Normalfall
    - primitive Datentypen werden als Wert übergeben
    - Objekte werden i.d.R. als Wert übergeben
    - Serialisierung: Objekte, Objektgeflechte (normal in Java als Referenz, ist hier anders!)
    - Verbindung ist in Bezug auf Objekte zustandslos
  - Parameter-Übergabe – Remote Objects
    - Implementierung Interface *Remote*
    - werden als entfernte Referenz übergeben
    - RMI als Middleware verwaltet Referenzen





- Parameter-Übergabe
  - primitive Datentypen werden als Wert übergeben
  - Remote Objects werden als entfernte Referenz übergeben
  - andere Objekte werden als Wert übergeben
- Attribute: nicht von Java unterstützt, über set-/get-Operationen realisiert
- Ausnahmen
  - vordefinierte Ausnahme RemoteException
  - typspezifische Ausnahmen möglich
    - Definition in Signatur
    - Kapselung in RemoteException
- Beispiel

```
class Address implements Serializable {
   private String street;
   private String postcode;
   private String city;

   public String getStreet() { return street; }
   // usw. ...
}

interface AdressInfo extends Remote {
  Address getLocation() throws RemoteException;
}
```

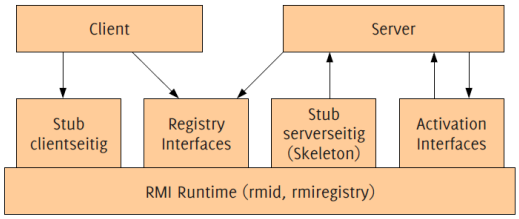
jedes Objekt, was kein Remote Object ist, aber serialisiert werden soll, braucht das Interface Serializable

Gibt eine Kopie der Addresse zurück!

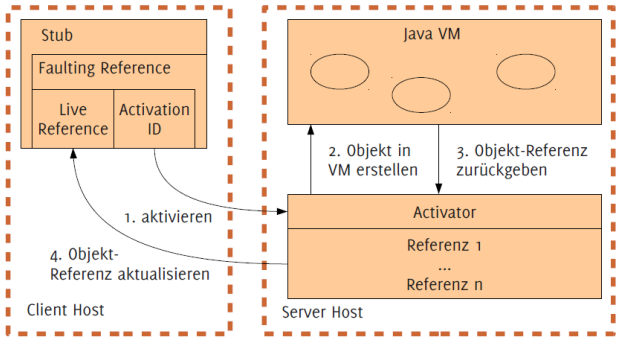
Remote Service

get-Methode kapselt Geschäftslogik

- Architektur



- Aktivierung von Diensten

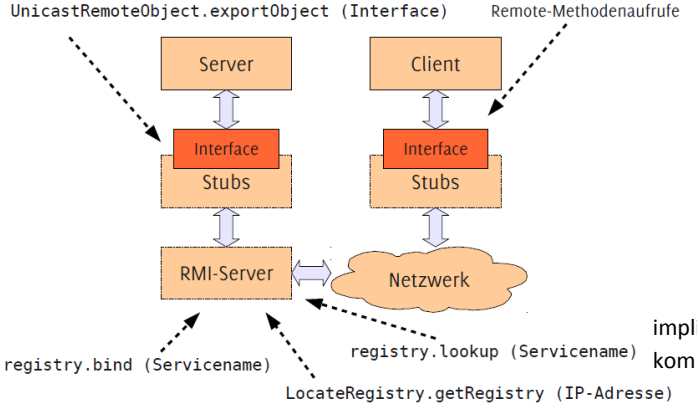


- Entwicklung

explizite Stub-Erstellung

`UnicastRemoteObject.exportObject (Interface)`

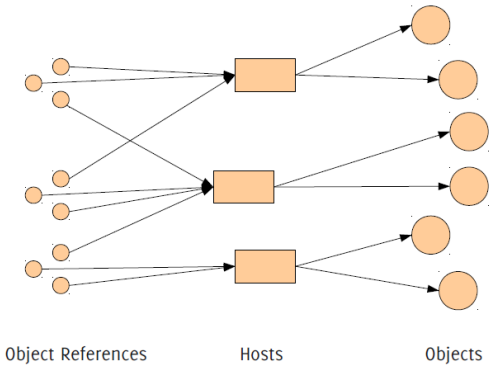
Remote-Methodenaufrufe



implizite Stub-Erstellung, Service kommt als Objekt zurück

Remote Objects

- Ausgangslage
  - Services stellen Methoden bereit
  - Objektbeziehungen können komplexer sein
- Ziel
  - Remote Services als Einstiegspunkte
  - Dienste können so aufgerufen werden
    - grob-granular
    - schwierig abzubilden für Transaktionen oder längere Prozesse
- Zugriffe auf entfernte Objekte erwünscht
  - Datenobjekte
  - detailliertere Informationen



auf beiden Seiten eigene JVMs,
beide führen Garbage Collection aus

Verwaltung von Remote Objects
sehr aufwändig

- Remote Objects sind alle Objekte, auf die über eine entfernte Referenz zugegriffen wird
- immer: Schnittstellen als Einstiegspunkt
  - hinter jeder Schnittstelle gibt es ein Remote Object, welches sie implementiert
- optional
  - Schnittstellen-Implementierung kann auf weitere Objekte verweisen
  - diese werden nicht serialisiert, sondern ihre Referenzen übertragen
- Beispiel

```
class Address implements AdressRemote {
   private String street;
   private String postcode;
   private String city;

   public String getStreet()
   { return street; }
   // usw. ...
}
interface AdressRemote extends Remote {
  Address getStreet() throws RemoteException;
   // usw. ...
}
```

Remote Object für Daten

ähnlich wie das vorherige Beispiel,
da wurde jedoch nur serialisiert

```
interface AdressInfo extends Remote {
  Address getLocation() throws RemoteException;
}
```

Service

- Technologie
  - komplexe Verwaltung durch JVMs notwendig
    - Referenzen werden „durchgeleitet“
    - Zugriffe werden geroutet
  - Herausforderungen
    - JVM-Grenzen, Garbage Collection, Abbruch von Verbindungen, verschiedene Java-/ Komponenten-Versionen, gleichzeitige Zugriffe

Bewertung

- Anforderungen an verteilte Systeme
  - Teilung der Betriebsmittel
    - Verteilung von Programmen auf mehrere JVMs
    - an potentiell mehreren Standorten
  - Offenheit
    - Kommunikation über wohldefiniertes Protokoll
    - Beschränkung auf Java-Welt
  - Nebenläufigkeit
    - mehrere Clients können gleiche Services verwenden
    - das sagt allerdings nichts über Nutzung paralleler HW aus!
  - Skalierbarkeit
    - RMI selbst bietet keine Unterstützung
    - abhängig von JVM
  - Fehlertoleranz
    - Kapselung von Fehlermeldungen in Exceptions
    - keine Ausfallsicherheit
- Transparenzen
  - Zugriffstransparenz → gegeben
    - Zugriff auf entfernte Objekte über ein Java-Interface
    - kein Unterschied zum lokalen Fall
  - Orttransparenz → stark eingeschränkt
    - Kenntnis über Ort der Services notwendig
      - Lookup, Ansprechen einer Registry unter Nutzadresse
    - keine weiteren Transparenzen erfüllt

Zusammenfassung

- allgemein
  - Kommunikation zwischen JVMs über das Netzwerk
  - starke Integration in Java
- Remote Services
  - entfernte Dienste
  - Beschreibung über Java-Interfaces
- Remote Objects
  - entfernte Objekte mit Referenzen
  - es werden Interface und Implementierung unterschieden
  - Kapselung und Ausnahmebehandlung auch für den entfernten Fall
- gemischte Bewertung bzgl. Anforderungen und Transparenzen

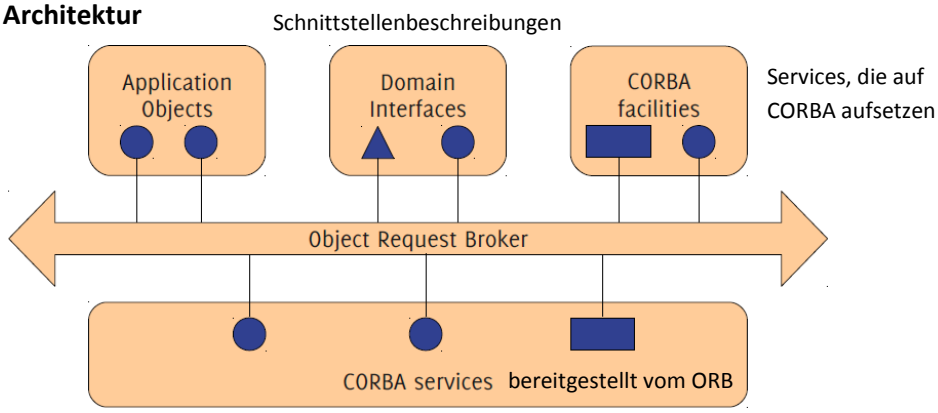
Middleware für (Objekt)Verteilung – CORBA

Grundlagen

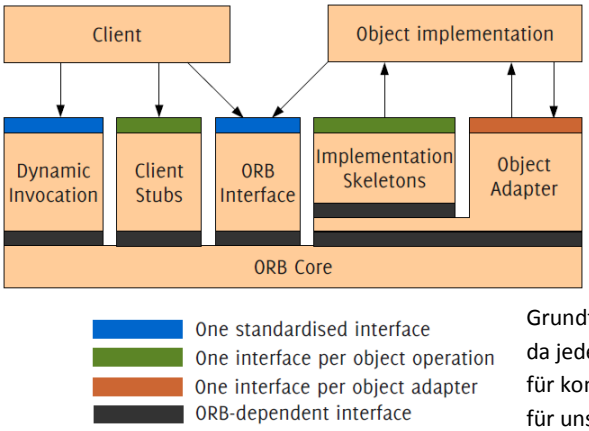
- „Common Object Request Broker Architecture“
- bisher (RMI)
  - Kommunikation in einem Plattformtyp
  - Überwindung von Prozess- (JVM) und Rechnergrenzen

- Motivation CORBA
  - heterogene Systeme
  - umfangreichere Beschreibung und Konnektivität
- Ziele
  - Unterstützung verteilter und heterogener Ausführungsanforderungen für Objekte, die transparent für Benutzer und Applikation sind
  - Realisierung der Integration von neuen Komponenten mit Alt-Applikationen (Legacy-SW)
- Organisation
  - offener Standard, freie Verwendung möglich
  - basiert auf Konsens der industriellen Partner der OMG (u.a. auch für UML zuständig)
- Funktionalität
  - Object Request Broker (ORB)
  - IDL
    - allgemeine Schnittstellenbeschreibung
    - Mapping auf konkrete Programmiersprachen
  - Kommunikation: statisch, dynamisch
  - Verteilung: Peer-to-Peer, Client und Server als Rollen (immer gleiche Funktionalität)

Architektur



- Object Services
  - grundlegend: Ereignisse, Lebenszyklus, Namensdienst, Persistenz, Objektbeziehungen
  - erweitert (optional): z.B. Nebenläufigkeit, Transaktionen, Timer, Sicherheit, Lizenzen
  - Naming: Context in ORB
    - Grundlage für Objektverteilung
    - Aktionen: Bind, Unbind, Resolve (Auflösen des Namens)
- Objekte
  - Objekte werden veröffentlicht
  - jedes Objekt besitzt einen eindeutigen Bezeichner innerhalb des ORB
  - mehrfache Referenzen zu einem Objekt möglich
  - Referenzen unterstützen die Lokalitäts-Transparenz
  - Objekt-Referenzen sind persistent
- Zugriff
  - statisch
    - Stubs realisieren Zugriff, Generierung möglich
  - dynamisch
    - Arbeiten mit Meta-Informationen, Objekte und Schnittstellen, dynamische Anfragen, Interpretation der Ergebnisse



Grundfunktionalität immer gleich
da jeder Stub nur für ein Objekt gilt
für konkrete Instanzen
für uns nicht relevant

- Ausführungsanforderungen
  - Aufrufe werden in Clienten formuliert
  - Aufruf besteht aus: Referenz zum Serverobjekt, Name der aufgerufenen Operation, aktuellen Parametern, Kontextinformation
  - Aufruf wird synchron ausgeführt
  - Definition von Aufrufen statisch oder dynamisch möglich

Objektmodell (somit unabhängig von Programmiersprachen)

- CORBA-Objekt
  - Ausgangspunkt: Objektorientierung
- Prinzipien
  - Kapselung: Schnittstelle (ORB) vs. Implementierung, Vererbung, Polymorphismus
- Umsetzung
  - IDL beschreibt Schnittstelle
  - Implementierung nicht in CORBA definiert
    - Anfragen werden weitergeleitet an Implementierungssprache
    - Mapping auf Implementierungssprache
- IDL
  - Sprache zur Formulierung aller Aspekte des CORBA Objekt-Modells
  - OMG/IDL ist unabhängig von Programmiersprachen, an C++ angelehnt, nicht Turing-vollständig (da keine Implementierung)
- Übersicht
  - Objektschnittstellen, Typen, Module, Attribute, Operationen, Ausführungsanforderungen, Ausnahmen, Subtypen
- Objektschnittstellen
  - Grundlage für Definition in IDL
  - enthalten Typen, Attribute, Operationen, Ausnahmen
  - werden definiert in Modulen, mit Vererbung

Impliziter Supertyp: Object

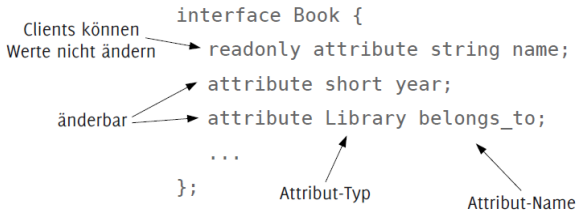
Ererbt von Library

```
interface Organization {
  readonly attribute string name;
};
```

Supertypen

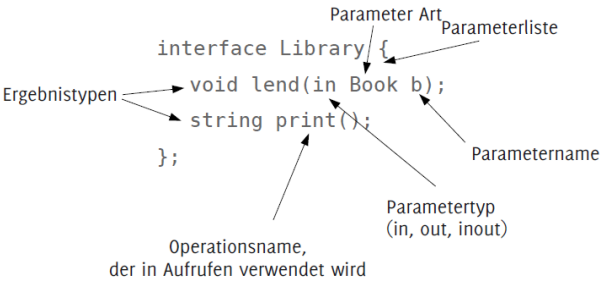
```
interface Library : Organization {
  exception BookNotAvailable{};
  readonly attribute short noOfBooks;
  readonly attribute Address location;
  attribute BookList books;
  void lend(in Book b) raises BookNotAvailable;
};
```

- Typen
  - vordefinierte Basis-Typen: Zahlenwerte, Character, „Any“ (wie Object in Java)
  - komplexe Typen: Enum, Struct, Union, String, Sequence, Array, Fixed
- Attribute



statt get/set nur deskriptive Beschreibung

- Operationen



- Ausnahmen
  - generische Ausnahmen (Netzwerkprobleme, falsche Referenzen, …)
  - Typ-spezifische Ausnahmen
- Module ähneln Namensräumen, z.B. Library:Address und University:Address

Java-Integration

- CORBA beschreibt keine Implementierung
- CORBA verbindet bestehende Plattformen
- Anbindung an Sprache notwendig
  - Mapping der IDL-Funktionalität
  - insb. Betrachtung von Einschränkungen
- Java als Beispiel
  - OO-Sprache
  - Java-Interfaces haben Ähnlichkeit mit IDL
  - Unterschiede existieren dennoch
- Realisierung: Mapping-Komponenten definieren Schnittstellen-Programmcode
- Interface
  - erweitert Basisklasse org.omg.corba.Object
  - Herausforderungen
    - Mapping auf Clientseite
    - Objekt-Adapter auf Serverseite
- Typen
  - Basis-Datentypen
    - Java-Datentypen können CORBA-Datentypen repräsentieren
    - Holder-Klassen enthalten konkrete Werte
  - komplexe Datentypen
    - String, Enums, Arrays, … haben Entsprechungen in Java
    - Struct: Klassen mit entsprechenden Namen (Attribute, get-/set-Methoden)
    - Unions: Java-Klassen enthalten alle möglichen Werte

        - Sequenzen: Arrays mit Sicherstellung der Länge durch Entwickler
        - Any: vordefinierter Typ in CORBA-Bibliothek
- Attribute
    - lesender Zugriff: Get-Methode, immer vorhanden
    - schreibender Zugriff: Set-Methode, nur vorhanden, wenn Attribut nicht „readonly“
- Operationen
    - Aufbau Signatur ähnlich
    - unterschiedliche Parametertypen
        - In: Pass-by-value
            - primitive Datentypen: wie in Java
            - Objekt: Serialisierung und Kopien
        - Out: Pass-by-result
            - Übergabe von Holder-Objekten
        - Inout: Pass-by-reference
            - Übergabe von Holder-Objekten
    - Exceptions: Mapping von Typen

Bewertung

- Anforderungen an verteilte Systeme
    - Teilung der Betriebsmittel
        - P2P-Architektur
        - Einbeziehung von Ressourcen möglich
    - Offenheit
        - Kommunikation über wohldefiniertes Protokoll
        - offen für diverse Programmiersprachen und -plattformen
    - Nebenläufigkeit
        - mehrere Clients können gleiche Services nutzen
        - das sagt allerdings nichts über Nutzung paralleler HW aus!
    - Skalierbarkeit
        - abhängig von ORB
    - Fehlertoleranz
        - Kapselung von Fehlermeldungen in Exceptions
        - das bedeutet nicht automatisch Ausfallsicherheit
- Transparenzen
    - Zugriffstransparenz → gegeben
        - Zugriff auf entfernte Objekte über ein Interface der jeweiligen Programmiersprache
        - kein Unterschied zum lokalen Fall
    - Ortstransparenz
        - eingeschränkt, da Verbindung zum ORB explizit aufgebaut wird
        - Namensservice als Umgehung (braucht aber auch eine IP-Adresse)
    - weitere Transparenzen
        - ORBs laufen als separate SW
        - Verbindungen sind für Clients transparent
        - ORB-Implementierungen können weitere Transparenzen anbieten

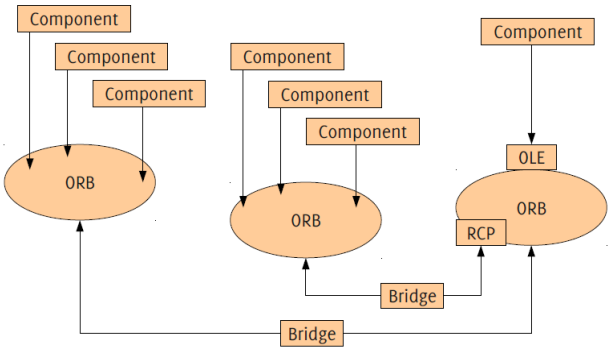
Zusammenfassung

- Object Request Broker
- P2P-Architektur in heterogenen Umgebungen
- Architektur für Verwaltung und Bereitstellung von Objekten
- IDL beschreibt Schnittstellen und Typen
- Mappings für Programmiersprachen existieren
- ORB als eigene SW-Komponente erfüllt mehr Anforderungen und Transparenzen als z.B. RMI

Middleware für (Objekt)Verteilung – Datenrepräsentation und XML

Heterogenität

- verteilte (OO-)Applikationen sind ganz natürlich durch Heterogenität und Flexibilität gekennzeichnet
  - kein Rechnerknoten ist wie der andere (SW, HW)
  - neue Rechnerknoten und Anwendungen kommen hinzu
- verteilte Objekte
  - verschiedene Programmiersprachen
    - gemeinsames Objektmodell (wie Objekte im Speicher dargestellt werden)
    - gemeinsame IDL
    - Anbindung an Programmiersprachen (z.B. über CORBA)
  - heterogene Middleware
    - Interoperation
    - Interworking
  - heterogene Datenrepräsentationen
- Darstellung primitiver Datentypen
  - die Datenrepräsentation auf der Bit-Ebene ist sehr unterschiedlich
    - Reihenfolge der Bytes im Hauptspeicher und in Maschinenregistern
    - big endians (größter Wert zuerst) und little endians (geringster Wert zuerst)
- unterschiedliche Zeichendarstellungen: EBCDIC, ASCII, ISO-8859-1, UCS
  - auch in Programmiersprachen, z.B. Speicherung eines Strings
- Charakteristiken der Lösung des Problems
  - Datenrepräsentationen müssen umgewandelt werden
  - Umwandlungen sollten transparent für den Anwendungsentwickler sein
  - Umwandlungen können implementiert sein in Application Layer / Presentation Layer / Plattform (Presentation + Session Layer)
- Ansätze zur verteilten Datenrepräsentation
  - Presentation Layer: Sun XDR, OMG CDR
  - Anwendungsebene: ASN.1, XML
  - Plattform: virtuelle Maschinen, z.B. Java
- Middleware-Implementierungen
  - unterschiedliche Middleware-Implementierungen
    - verfügbare Programmiersprachen-Anbindungen
    - verfügbare Services & Funktionen
    - unterstütze HW / OS / …
  - Trennung von Sicherheitsbereichen
  - Verteilung im großen Maßstab



Bridge verbindet immer 2 Instanzen, sind individuell für alle versch. Programmiersprachen

- Middleware-Integration
  - Interoperation
    - unterschiedliche Implementierungen des gleichen Standards arbeiten zusammen
      - z.B. JBoss und Glassfish, beides JEE-Implementierungen
    - Interaktionsprotokolle
  - Interworking (viel komplexer)
    - unterschiedliche Middleware-Ansätze arbeiten zusammen
    - Abbildung auf Objektmodelle
    - Definition von Interaktionsprotokollen

XML

- Motivation
  - Strukturbeschreibung von Daten als Basis für Flexibilität
    - verteilte Systeme müssen auf Änderungen nach dem eigentlichen Design reagieren
    - Austausch komplexer Objekte ist mit RMI/CORAB u.U. sehr aufwändig
  - Anforderungen an Systeme
    - Verarbeitung von Dokumenten wird system- und herstellerunabhängig
    - Darstellung und Verbreitung von Informationen über unterschiedliche Medien möglich
  - Standard Generalized Markup Language (SGML) als Grundlage
    - Met-Sprache, 1986 als ISO-Standard 8879 verabschiedet
- Aufbau
  - XML-Dokumente basieren auf SGML
    - einfacher strukturiert, um es besser automatisch verarbeitbar zu machen
  - Strukturierung
    - Inhalt mit Nutzdaten und Strukturbeschreibung
    - Strukturdefinition (= Grammatik von Dokumenten): DTD und XML Schema
    - Darstellung
  - Document Object Model (DOM)
    - standardisiertes Datenzugriffsmodell
    - plattform- und sprachunabhängige Schnittstelle für dynamischen Zugriff auf Inhalt, Struktur und Layout eines XML-Dokuments
- Inhalte
  - Parsen
    - Parser ermöglichen Zugriff auf Inhalte eines XML-Dokuments
    - DOM-Parser (flexibler, einfacher)
      - ganzes Dokument, Transformation in hierarchischer Baumstruktur (DOM)
    - SAX-Parser (ressourcenschonender)
      - schrittweiser Zugriff, Dokument als Datenstrom, Zugriff auf einzelne Inhalte

        - Werkzeuge
            - große Anzahl für fast alle Programmiersprachen, z.B. Apache Xerces
    - Validierung
        - einfacher Parser vs. validierender Parser (Zugriff auf Schemadefinition)
        - Ergebnisse: gültig (valid), wohlgeformt (grundlegende Anforderungen an XML-Syntax), ungültig
    - Arbeiten mit Inhalten
        - Transformation: XSLT übersetzt XML-Quelldateien von einer XML-Strukturdefinition in eine andere
        - Selektoren: XPath
        - Gestaltungsaspekte: Formatting Objects
        - Linking: XLink, XPointer
- Struktur
    - DTD als kontextfreie Grammatik, Ableitung von Wörtern durch Produktionsanwendungen
    - XML Schema bietet mehr Möglichkeiten als DTD: einfaches Typsystem, reichhaltige Menge von Basistypen, mehr indirekte Verweise (Flexibilität)
- Verwendung
    - Bewertung
        - Verwendung von XML kann die Flexibilität erhöhen
        - verschieden Varianten möglich
        - Aspekte: Kommunikation, Verarbeitung (Client-/Server-)
        - Vor- und Nachteile: Performance, Skalierbarkeit, Flexibilität
    - Gegenbeispiel: Datenaustausch „von Hand“
        - strukturierte Daten werden beispielsweise per TCP/IP ausgetauscht
        - Protokollfragen müssen separat behandelt werden (insb. Fehlerbehandlung)
        - Vorgehen
            - Client stellt Anfrage zusammen
            - Versenden an den Server
            - Auspacken der Anfrage und Bearbeitung
    - Verarbeitung der Strukturen
        - im Prinzip Standardverfahren des Marshalling/Unmarshalling
            - aber hier auf der Ebene der Applikation
        - Vorteile
            - Berücksichtigung spezieller Eigenschaften möglich, Verzicht auf teure Automatismen
        - Nachteile
            - spezielle Lösungen, fehlendes RE, Fehler, …
    - Standardisierung
        - erhöhter Aufwand wird durch systematische Verarbeitung vermieden
        - Systematik basiert auf der Benutzung des Standards XML
            - Festlegung der gemeinsamen Strukturen
            - Kodierung der ausgetauschten Strukturen
            - systematische Verarbeitung der entstehenden Baumstrukturen
            - strukturiertes Vorgehen bei fehlenden / zusätzlichen Elementen
            - zumindest innerhalb einer Applikationslandschaft standardisierte Fehlerbehandlung
    - Voraussetzungen
        - Definition der gemeinsamen Sprache

        - Definition der Grammatik der Sprache: DTD oder XML Schema
        - ggfs. Kodierung der Terminalsymbole beachten (Umlaute, Zahlenformate, …)
        - es ist zu beachten, dass Erweiterungen später eingebaut werden können
    - Anfragen auf Client-Seite
        - Durchlauf der zu verpackenden (internen) Struktur
        - zu beachten ist der Aufbau der Struktur
            - Regelmäßigkeit erhöht Strukturiertheit und Systematik der Lösung
            - wünschenswert: Baum, Graph etc., möglichst keine speziellen Verzweigungen, Verkettungen
        - Achtung: alle Gemeinsamkeiten von Strukturdurchläufen sind zu berücksichtigen
        - Baum-Strukturen (z.B. modifizierter Inorder-Durchlauf)
    - Anfragen auf Server-Seite
        - zunächst muss entschieden werden, wie die Daten verarbeitet werden (SAX, DOM)
            - Baumstruktur liefert die Basis für einen systematischen Durchlauf
            - der Durchlauf kann fehlende oder zusätzliche Elemente „entdecken“
            - systematische Behandlung dieser Ausnahmen notwendig
            - standardmäßig werden dann die Elemente des Baums in die Zielstrukturen im Server übertragen

Bewertung

- Motivation für Verwendung von XML: Datenaustausch außerhalb der Middleware, höhere Flexibilität, Persistenz
- Ausdrucksfähigkeit: Struktur, Trennung verschiedener Aspekte
- Verwendung: Flexibilität, weniger Funktionalität da kein Framework

Middleware für (Objekt)Verteilung – Web Services

Grundlagen

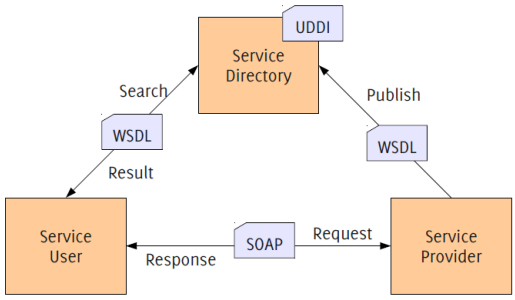
- Motivation
    - Datenrepräsentation XML
        - zunächst muss entschieden werden, wie die Daten verarbeitet werden (SAX, DOM)
        - einfach, keine Plattform, keine Tool-Unterstützung für Verteilungs-Aspekte
    - Wie können Dienste im Internet angeboten werden?
        - Skalierbarkeit, keine zentralen Verzeichnisse, keine zentrale Kontrolle
    - Dienste müssen sich selbst beschreiben
    - Unterstützung für das Auffinden von Diensten
- Definition
    - standardisiertes Verfahren für die Realisierung eines entfernten Aufrufs durch Web-Techniken
    - Protokoll: SOAP
        - Transportschicht: TCP, Anfragen mit HTTP, Verpackung von Nachrichtendaten in XML
    - zusätzlich Transparenzen, CORBA-ähnliche Eigenschaften
- Analogie zu CORBA
    - Interface-Beschreibung: WSDL (analog zur IDL)
    - Verzeichnisdienste: UDDI (analog zum ORB)
    - Abwicklung des Aufrufs: SOAP (analog zum CORBA-Protokoll)
    - Kombination von Services: ebXML

Schnittstellen

- Ziele
  - XML-Basierung beeinflusst Schnittstelle
    - Datentypen für XML, Adapter für Sprachen / Plattformen notwendig
  - Frage nach Bedeutung und Wirkung eines Service
    - Beschreibung, Auffinden anhand semantischer Kriterien, Kombination mit anderen Services
  - daher: eine automatisch verarbeitbare Semantikbeschreibung notwendig
  - Umsetzung: Web Service Description Language (WSDL)
- Datentypen
  - Datentypen (types) zur Beschreibung auszutauschender Nachrichten (Attribute, Zugriff)
  - Nutzdaten (Nutzdaten, Nachrichten)
- Nachrichten
  - Nachrichten (message) zur abstrakten Definition übertrageneder Daten
  - logische Bestandteile der Nachricht
  - Verknüpfung mit Datentypdefinitionen
- Port-Typen
  - Port-Typen (port type) zur Beschreibung angebotener Dienstes
  - Definition von Operationen (Anfrage, Antwort)
- Ports
  - konkrete Benennung und Veröffentlichung, meist als URI
  - Instanz von Port-Typ
- Service
  - Aggregation zusammengehöriger Ports

Architektur

- Standards: XML, SOAP, WS Business Activity, WS Security, WSDL, WS Policy, UDDI, BPEL
- Spezifikation



- Konstruktion komplexer Services: ebXML (Electronic Business using eXtensible Markup Language)

Datenaustausch

- Nachrichten ebenfalls XML-basiert
- Nutzdaten der Nachrichten verpackt in „Umschlägen“
- Zuordnung der Nutzdaten zu Definitionen aus WSDL

Bewertung

- Offenheit
  - umfangreiche Schnittstellenbeschreibung
    - Integration und Konvertierung für vorhandene Systeme einfach
    - semantische Beschreibung

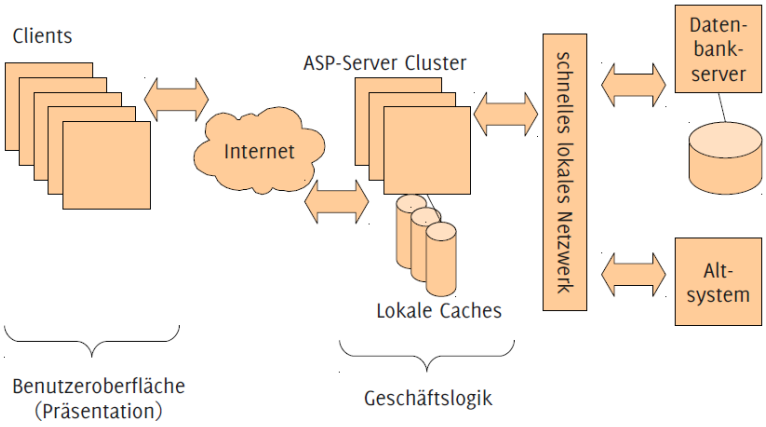
  - Anbindungen existieren an fast alle aktuellen Systeme
  - Stub-Generierung aus WSDL möglich
  - keine „native“ Integration
  - Marshalling / Unmarshalling notwendig
  - Verpacken beliebiger Objekte als XML
  - Performance
- weiter Anforderungen an verteilte Systeme
  - abhängig von Implementierung, aber Verteilung wird vereinfacht
- Zugriffstransparenz
  - nicht nativ, da XML nicht direkt in Programmiersprachen integriert ist
  - Erstellung von Bindings möglich, die zugriffstransparent sind
- Ortstransparenz
  - möglich mit Verzeichnisdiensten
- weitere Transparenzen
  - abhängig von Implementierung, aber Verteilung wird vereinfacht
- Effizienz
  - Bedeutung von Inhalten wird beschrieben
  - Overhead durch zusätzliche Daten
  - Performance der Übertragung

Middleware für Komponenten

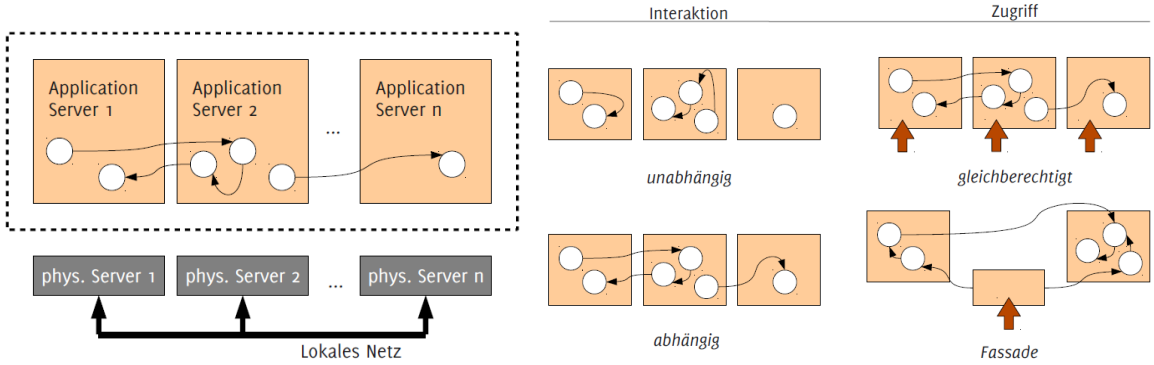
Application Server

- Komponentenausführung
  - Komponenten erfüllen verschiedene Zwecke
  - Ausführungsumgebung für Komponenten benötigt
  - Verwaltung des Lebenszyklus von Komponenten (Deployment, Start, Stop, Abhängigkeiten)
  - Kommunikation zwischen Komponenten (Abhängigkeiten, Benennung, Referenzen und Aktivierung: „Lookup“)
  - Ausführung soll transparent sein
    - Verteilung (Client/Server, Cluster, …), Nebenläufigkeit, Skalierbarkeit, (Ausfall-)Sicherheit
  - Bereitstellung allgemeiner Dienste
    - Sicherheit, Persistenz, Anbieten von Diensten nach außen über verschiedene Protokolle
  - Gesamtziel: einfachere Entwicklung
    - Integration und Konvertierung für vorhandene Systeme einfach
- Verteilung
  - Client-seitig
    - Versorgung großer Zahl verteilter Arbeitsplätze
    - SW-Verteilung
    - geringe Anforderungen an Ressourcen
    - gute Möglichkeiten, Grafiken, GUI etc. lokal darzustellen und zu verarbeiten
    - schlechte Möglichkeiten bzgl. Datenhaltung / Persistenz
  - Server-seitig
    - Server sind leichter skalierbar (stärkere Rechner, schnellere Netzwerke)
    - zentrale Dienste (z.B. E-Mail) und die genannten Aspekte (Persistenz etc.) sind einfacher zu realisieren
    - Administration und SW-Verteilung sind zentral einfacher zu lösen

- Szenario



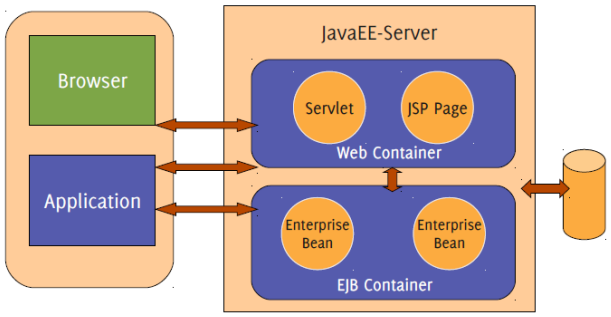
- Nutzung
  - Zugriff
    - Web-Oberflächen
    - „Rich Clients“: auf dem Client ausgeführte SW
    - Terminal-Server: nur Interaktion im Frontend (z.B. Citrix)
  - kommerzielle Nutzung: pay per use (z.B. Cloud Computing), Abonnement
  - Bewertung
    - Vorteile: einfache Verfügbarkeit für den Kunden, überschaubare Kostenstruktur, …
    - Nachteile: die Informationen und Daten des Kunden werden i.d.R. außerhalb seines Einflussbereichs verwaltet
- Skalierung (Cluster)
  - Cluster
  - Server-Instanzen arbeiten auf verschiedenen physikalischen Maschinen im Verbund
  - transparent für ausgeführte Komponenten
  - Last wird verteilt zwischen Maschine
  - Server muss Weiterleiten von Anfragen koordinieren
  - Fragen
    - Was wird verteilt, und nach welchen Kriterien?
    - Wie werden Schnittstellen und Performance davon beeinflusst?
    - Welche Komponente entscheidet über Verteilung?



- Java Enterprise Edition
  - Java EE als reales System
  - Integration verschiedener Komponenten-Technologien
  - verschiedene Anbieter (z.B. JBoss, IBM WebSphere)
  - Standards vorhanden, es werden aber nicht alle Teilbereiche abgedeckt

Container

- Voraussetzungen
  - Komponenten erfüllen verschiedene Zwecke
  - Komponenten sind je nach Zweck unterschiedlich aufgebaut
  - Komponenten benötigen daher unterschiedliche Ausführungsumgebungen
  - Application Server müssen diese integrieren
- Konzept
  - Application Server führt Komponenten nicht direkt aus, sondern Container
  - Container können Komponenten zweckspezifisch verwalten
  - Nutzung von gemeinsamen Diensten des Application Servers
    - Zugriff auf Funktionen, Adaption von Funktionen



Web Container rendert z.B. HTML

Servlet = vordefinierte Basisklasse, die auf HTTP-Requests reagiert

JSP: mehr Skriptansatz, im Hintergrund werden daraus Servlets generiert

- Namensdienst
  - verschiedene Komponenten werden angeboten
  - Komponenten können interagieren
  - Voraussetzung: Komponenten sind auffindbar
  - Namensdienst notwendig
    - Komponenten werden benannt (automatisch/manuell)
    - Namensdienst kann überall angesprochen werden
    - „Lookup“ von Komponenten möglich
- Typen
  - Geschäftslogik, z.B. Enterprise Java Beans (EJB) Session Beans
  - persistente Daten, z.B. Enterprise Java Beans (EJB) Entity Beans
  - Web-Komponenten, z.B. Servlets
  - Systemverwaltung, z.B. Java Management Extensions (JMX)
  - Web Services, AOP-Erweiterungen, Datenquellen, Prozesssteuerung (z.B. jBPM)
- Metadaten
  - Metadaten werden benötigt
    - Application Server muss Komponenten Containern zuordnen
    - Container erhalten Informationen über spezielle Komponenten
    - Komponenten starten und stoppen sich nicht selbst, müssen sich dafür aber umfassend beschreiben (Schnittstellen und Einstiegspunkt)
  - Beschreibungen
    - Konfigurationsdateien (Deployment Deskriptoren), meist XML
    - Metadaten im Quellcode (Java: Annotationen)
    - Strukturen von Klassen in Komponenten
  - Formen von Komponenten
    - Geschäftslogik als kompilierte Bibliotheken mit Metadaten
    - nur Beschreibung über Metadaten, Laden von externen Bibliotheken

- Pakete
  - Anwendungen bestehen aus Klassen, Deployment Deskriptoren und binären Ressourcen
  - Pakete sammeln diese Daten in einer bestimmten Struktur
    - Klassendateien in package-Ordnern
    - Deskriptoren in reservierten Ordern (META-INF)
  - Container interpretieren definierte Struktur und dann die enthaltenen Daten
    - damit möglich, dass Container die deployten Daten lesen und starten kann
  - Typen von Paketen
    - Bibliotheken von Klassen
      - Java Archive – JAR
      - können Meta-Informationen für Komponenten enthalten (Ordner META-INF)
    - Web-Anwendungen (im Prinzip gezippte Version einer Webseite)
      - Web Archive – WAR
      - Meta-Informationen für Web-Anwendungen (Ordner WEB-INF)
      - Klassen, Bibliotheken und statische Ressourcen für Web-Anwendungen
    - Anwendungen mit mehreren Komponenten
      - Enterprise Archive – EAR
      - Zusammenstellung von JARs und WARs, Deployment-Deskriptor beschreibt Inhalte

Komponenten für Geschäftslogik

- Motivation
  - Geschäftslogik als eigentliche Ausführung einer Anwendung
  - Einfluss verteilter Systeme
    - Zeitpunkt der Ausführung
      - kein starrer Ablauf
      - Ausführung auf Anfrage – „Inversion of Control“ (Anwendung kontrolliert sich nicht selbst, beschreibt nur was mit den Komponenten gemacht werden kann)
      - Aufruf von verschiedenen physikalischen Orten
    - Nebenläufigkeit
      - Anfragen zu verschiedenen Zeitpunkten
      - gleichzeitige Ausführung notwendig
      - parallele Ausführung nicht vorhersehbar
      - gleichzeitiger Zugriff auf Daten
  - Strukturierung großer Systeme notwendig
- Funktionalität
  - Auffinden von Komponenten
    - Implementierung der Geschäftslogik in komplexen Strukturen, Auffinden über Namen
  - Aktivierung von Komponenten
    - Start von Instanzen, Wiederverwendung, Pools von Instanzen
  - Deaktivierung von Komponenten
    - Freigabe von Ressourcen beendeter Verbindungen
    - Entdecken von Verbindungsabbrüchen und Timeouts (z.B. über Heuristiken)
  - Verwaltung von Abhängigkeiten zwischen Komponenten
- Schnittstellen
  - Komponenten müssen Schnittstellen veröffentlichen (Methoden von außen ansprechbar)
  - Art der Verbindung zwischen Client und Server: zustandsbehaftet / zustandslos

- EJB Session Beans
  - Enterprise Java Beans (EJB)
    - ein Standard der Java Enterprise Edition, definiert serverseitige Komponenten
  - Session Beans
    - realisieren Geschäftslogik
    - Bean-Implementierungen werden über Interfaces angesprochen (wie bei RMI)
  - Konfiguration
    - Java-Annotationen an implementierende Klasse
    - alternative Konfiguration oder Customizing über XML-Dateien möglich
  - Zugriff
    - Definition verschiedener Interfaces für verschiedene Zugriffsarten
    - lokal → Annotation @Local
      - Bean auf dem gleichen Rechner/Cluster
    - entfernt → Annotation @Remote
      - Bean auf dem gleichen oder einem entfernten Rechner/Cluster
      - System verbirgt Übertragung über RMI
  - Zustandsbehaftung
    - aktiviert mit Annotation @Stateful
    - nicht aktiviert mit Annotation @Stateless
  - Ansprechen von Session Beans
    - Session Beans haben eindeutige Namen
    - (leider) kein allgemeiner Standard für Benennung
    - Beispiel JBoss Application Server: Name der Bean-Klasse und Art des Interfaces, z.B. KlickBean/local
  - Beispiel

KlickBean.java

```
@Stateful
@Local(Klick.class)
public class KlickBean implements Klick
{
    private int anzahlKlicks = 0;

    public void klick()
    {
        anzahlKlicks++;
    }

    public int zeigeAnzahl()
    {
        return anzahlKlicks;
    }
}
```

zeigt auf Klick-Interface

Klick.java

```
public interface Klick
{
    void klick();
    int zeigeAnzahl();
}
```

KlickServlet.java

```
public class KlickServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse respon
            throws ServletException, IOException
    {
        try {
            Klick bean = (Klick)request.getSession().getAttribute("bean");
            if (bean == null) {
                bean = (Klick) new InitialContext().lookup("KlickBean/local");
                request.getSession().setAttribute("bean", bean);
            }

            response.setContentType("text/html");
            ServletOutputStream os = response.getOutputStream();
            os.print("<html><body>");
            bean.klick();
            bean.zeigeAnzahl();
            os.print("</body></html>");
        } catch (Exception e) {
            throw new ServletException(e);
```

Name-/Wert-Paar

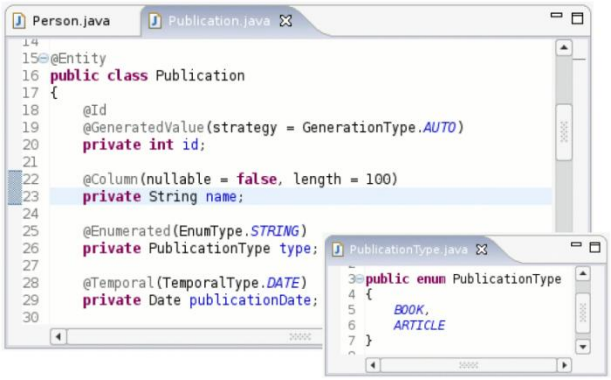
gibt's noch keine Session, wird ein neues Bean erstellt und eine Session gespeichert

    - Stateless mit/ohne Referenz: Hochzählen funktioniert manchmal
    - Stateful ohne Referenz: Hochzählen funktioniert nicht
    - Stateful mit Referenz: Hochzählen funktioniert

Komponenten für Persistenz

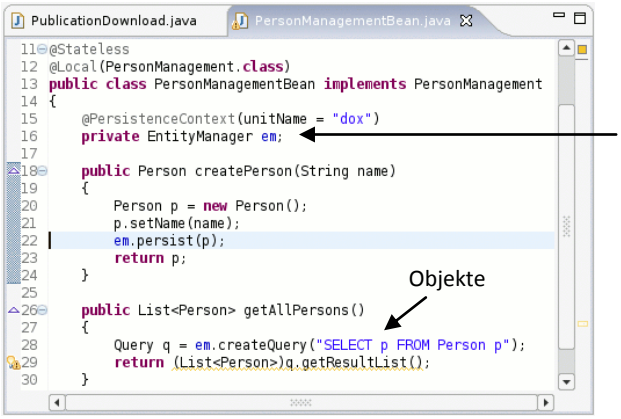
- Motivation
  - Anwendungen verarbeiten Daten, Daten werden in DBs verwaltet
  - Zugriff soll in Form von Komponenten geschehen
    - Verwendung von Objekt-Technologie
    - Objekt-Relationales Mapping (ORM) → zentrales Konzept
  - Mapping von DB-Informationen auf Objekte
    - Attribute (Datentyp, Restriktionen)
    - Beziehungen (Ziel, Kardinalität)
- Attribute
  - Mapping von Objekt-Attributen auf DB-Attribute
  - Typdefinition im Objektsystem
  - Restriktionen für Attribute: Nullwerte, Eindeutigkeit, Feldlänge, Genauigkeit etc.
- Instanzen und Beziehungen
  - Identifikation von Instanzen anhand des Primary Keys
  - Mapping von DB-Key auf Objektinstanzen (Primary Key, Foreign Key)
  - Typ-Bestimmung
    - Ziel-Typ über Mapping definiert, Typsicherheit oberhalb der DB
- EJB Entity Beans
  - Enterprise Java Beans (EJB)
    - Verwendung der persistenten Daten in bereits definierter Geschäftslogik
  - Entity Beans
    - repräsentieren Objekt-Instanzen persistenter Datensätze
    - Verwendung als einfache Java-Objekte (Plain Old Java Objects – POJO)
  - Konfiguration
    - Java-Annotationen an Klassen und Attributen
    - alternative Konfiguration oder Customizing über XML-Dateien möglich
  - Deklaration von Typen
    - Klassen werden mit @Entity annotiert
    - alle Attribute werden in der DB gespeichert (Ausnahmen werden definiert)
    - Attribute sind Member-Variablen, get/set-Methoden, abhängig von Position der Annotation
    - ID-Attribut mit @Id annotiert
  - allgemeine Spaltendefinition mit @Column
    - nullable, unique, length, precision, insertable/updateable
  - datentypspezifische Spaltendefinition
    - für Enumerationen mit Annotation @Enumerated (als Nummer oder als String)
    - Datum und Zeit mit Annotation @Temporal
  - Beziehungen
    - Objekt-Attribute als Foreign Key in der DB
    - @OnetToOne, @ManyToOne, @OneToMany, @ManyToMany (über Zwischentabelle)
  - Integration: Zugriff auf Entity Beans erfolgt über Session Beans
  - Zugriff
    - Session Beans können einen Entity Manager beinhalten
    - Entity Manager wird über Dependency Injection referenziert
    - Abfragesprache EJBQL auf Objektebene

- Beispiel



Umsetzung in DB – erzeugte Tabelle:

```
CREATE TABLE publication
(
  id int4 NOT NULL,
  name varchar(100) NOT NULL,
  "type" varchar(255),
  publicationdate date,
  CONSTRAINT publication_pkey PRIMARY KEY (id)
)
```



Einstiegspunkt in Persistenz-Container
nur Bereitstellung, wird nicht aktiv erstellt

Web-Komponenten

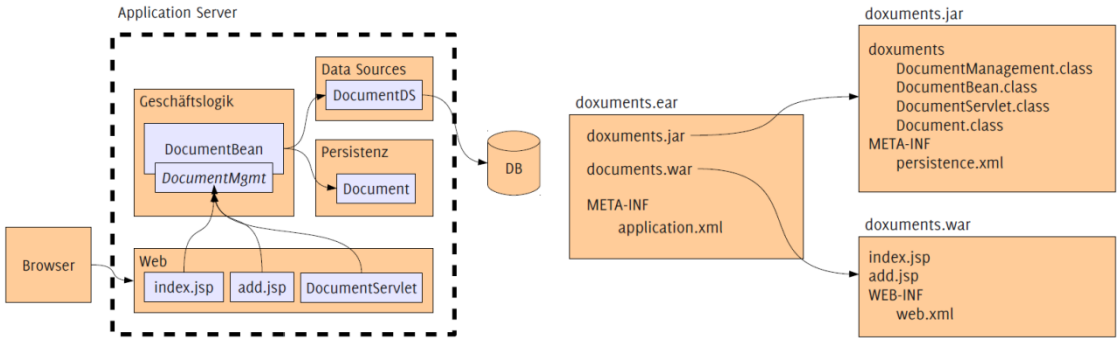
- Grundlagen
  - Präsentationsschicht / Benutzeroberfläche
  - Anforderungen an Web-Komponenten
    - Protokoll: HTTP, Darstellung im Browser: HTML
  - Inhalte
    - statisch, z.B. Bilder
    - dynamisch, z.B. Listen mit Daten aus einer Auswertung
  - Generierung
    - statisch, z.B. werden Bilder nur ausgeliefert (können auch aus DB kommen)
    - vorlagenbasiert, z.B. HTML-Templates, die mit Inhalten befüllt werden
    - Programmcode, z.B. Behandlung spezieller Anfragen
- Java Servlets
  - Servlets als Grundlage für Web-Komponenten in Java
    - Low-Level-Behandlung von Anfragen
    - Eingabe- und Ausgabe-Datenströme (auch über eine längere Zeit)
    - Abstraktion in der Plattform
      - Parsen der übergebenen Parameter
      - Verwaltung von Sitzungen durch Cookies oder IDs in der URL
  - Verwendung
    - spezielle Arten von Requests (z.B. Auslesen von Bildern aus einer DB)
    - Grundlage für alle einfacheren Mechanismen (Web-Frameworks)
- Java Server Pages
  - Motivation: Schreiben von Servlets (o.ä.) aufwändig
    - Programmcode wird benötigt für jede einzelne Anweisung

        - HTML ist eigentlich deklarativ, Bezug zu HTML ist schlecht erkennbar
    - Konzept
        - statisches HTML wird deklarativ geschrieben
        - Java-Code wird an gewünschter Stelle eingebettet
        - intern werden JSPs bei erster Ausführung zu Servlets kompiliert
    - Bewertung
        - besserer Bezug zum angezeigten Ergebnis
        - schlechteres Auffinden von Fehlern durch zwei Bedeutungen im Dokument
        - Erweiterung: Java Server Faces (semantische Informationen zu Web-Inhalten)

Schnittstellen von Application Servern

- Funktionalität innerhalb des Servers bereitgestellt
- Container steuern mögliches Angebot
- Schnittstellen nach außen werden benötigt
    - Zugriff von anderen Systemen (z.B. Clients), Aufruf von Diensten, Datenaustausch
- Container realisieren Schnittstellen
    - Mapping von Datenstrukturen und angebotenen Diensten, Application Server koordiniert
- Benennung und Auffindbarkeit abhängig von Containern
- RMI
    - Ziele: einfacher Zugriff innerhalb der Java-Welt
    - Funktionsweise
        - Container können Schnittstellen als RMI-Dienst veröffentlichen+
        - Trennung in Interface und Implementierung notwendig
        - EJB Session Beans realisieren das bereits (Übertragung auf Remote-Interface)
    - Java-Integration
        - Verwendung vorhandener Schnittstellen
        - Verwendung des Namensdienstes
        - Problem: enge Kopplung an Java-Version und Server-Implementierung
- CORBA
    - Application Server beinhalten einen ORB
    - Objekte können entsprechend dem Java-Mapping dort veröffentlicht werden
    - veröffentlichte Objekte können mit anderen Komponenten innerhalb des Servers interagieren
- Web
    - Veröffentlichung von Web-Komponenten möglich
    - menschliche Interaktion
        - Server stellt Benutzeroberfläche bereit
        - direkte Interaktion mit menschlichem Benutzer
        - Format: HTML
    - automatische Interaktion
        - Web-Anwendungen können beliebige Inhalte darstellen / Anfragen entgegennehmen
        - beliebiger Datenaustausch möglich, z.B. XML, Binärdaten
- Web Services
    - Java Application Server Bestandteil der Java-Plattform
        - Sprache, Objektmodell, Datentypen, Schnittstellen
    - Offenheit nach außen mit automatisierten Protokoll

  - Web Services als verbreitete Technologie
  - Komponenten stellen Inhalte bereit (z.B. Session Beans)
  - Container mappt Daten (Web Services können z.B. nicht mit Konstruktoren umgehen)
- Beispiel: Dokumentenverwaltung „Doxuments“



Architekturbeispiele

- .NET App ⟺ Netzwerk über Webservices ⟺ EJB (Business Logic) ⟺ DB
- Rich Clients ⟺ Netzwerk über RMI Remote Interface ⟺ EJB ⟺ DB (geht nur bei gleicher Java-Version)
- Browser ⟺ Netzwerk über HTTP ⟺ JSP ⟺ Local Interface ⟺ EJB ⟺ DB

Bewertung

- Teilung der Betriebsmittel
  - Application Server sind Bestandteil verteilter Systeme
  - Clustering möglich
    - Betriebsmittele werden in enger Kopplung geteilt, Lastverteilung etc. möglich
- Offenheit
  - verschiedene Schnittstellen verfügbar
  - Integration in Plattform: verschiedene Ausprägungen
  - Web Services als plattformunabhängiger Zugriff
  - beliebige Interaktion möglich (z.B. Servlets und XML, andere Ports, …)
- Nebenläufigkeit
  - Komponenten werden nebenläufig ausgeführt
- Skalierbarkeit
  - Clustering bietet Möglichkeit zur Skalierung
  - Aufgabenteilung bietet Möglichkeit zur Skalierung (z.B. separate DB)
- Fehlertoleranz
  - Clustering erlaubt Verstecken einzelner Fehler (HW)
  - weitere Fehlertoleranz abhängig von Containern (Verwaltung von Komponenten, Umleitung von Anfragen)
- Zugriffstransparenz
  - verschiedene Zugriffsmöglichkeiten auf Application Server
  - unterschiedliche Repräsentation auf Client-Seite
  - Möglichkeit 1: native Unterstützung
    - Verteilung von Java-Interfaces, beschränkt auf Plattform
  - Möglichkeit 2: Unterstützung durch Stubs
    - Stubs auf Clients für diverse Protokolle möglich
    - Stubs können Zugriffstransparenz herstellen

- Ortstransparenz
  - innerhalb des Application Servers
    - Namensdienst abstrahiert vom Clustering
    - Komponenten interagieren unabhängig von ihrem physikalischen Ort
  - vom Application Server nach außen
    - Namensdienst kann bereitgestellt werden, beschränkt auf Plattform, Adressen als Konfiguration
  - Schnittstellen des Servers
    - verbergen Clustering nach außen
    - Einstiegspunkt ist bekannt, Details der internen Verteilung nicht
  - unabhängig vom Server
    - Eintrag in Verzeichnisdiensten, z.B. für Web Services
- Migrationstransparenz
  - Clustering versteckt Migration, Zugriff von außen ohne Wissen darüber möglich
- Replikationstransparenz
  - Schnittstellen erlauben Replikation, lokal, im Cluster, abhängig von Containern
- Nebenläufigkeitstransparenz
  - Komponenten werden nebenläufig ausgeführt, nebenläufige Ausführung wird versteckt
- Skalierungstransparenz
  - Clustering erlaubt Skalierung
  - neue physikalische Server können einbezogen werden
  - keine weitere Interaktion mit HW
- Leistungstransparenz
  - gegeben durch Skalierung, Nebenläufigkeit, Replikation
- Fehlertransparenz
  - Fehler auf unteren Ebenen können versteckt werden
  - Fehler auf Ebene der Logik einzelner Komponenten müssen von Containern verwaltet werden

Services

Motivation

- bisher: Technologie für verteilte Systeme und Komponenten
- Herausforderungen
  - sehr große Systemlandschaften, verschiedene Technologien, Interoperation und Interworking notwendig
- Ziele
  - Verbinden aller beteiligten Systeme, einheitliche Plattform, langfristig verwendbare Standards
- relevanten Eigenschaften verteilter Systeme
  - Heterogenität aufgrund von HW, OS, Middleware, Applikationen
  - Selbstständigkeit: Systeme funktionieren eigenständig, Abgrenzung durch SW-Stack, echte Nebenläufigkeit, keine zentrale Steuerung
- Anforderungen
  - Selbstständigkeit muss erhalten bleiben: Anforderungen, Komplexität, organisatorische Fragen und Verantwortlichkeiten, geographische Verteilung
  - Herausforderungen: organisatorische / technische Integration
- Konsistenz

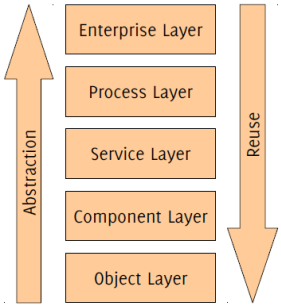
- Verteilung von Daten (Aktualität, Synchronisierung und Performance)
- Verteilung von Aufgaben (Transaktionen)

Services

- Ausgangspunkt Komponenten
  - wohldefinierte Schnittstelle
    - angebotene Dienste, Vor- und Nachbedingungen, Eingabe-Ausgabe-Relation, Kapselung
  - Wiederverwendung
    - Definition von Funktionalität nur einmal, Struktur und Übersichtlichkeit, Kosten, Datenkonsistenz, Vermeidung von Redundanzen
- Verwendung
  - Fokus: Wiederverwendung
  - Komponenten: Einbettung, viele Instanzen (bei komplexeren Systemen Einbettung von Komponenten nicht mehr zielführend)
  - Services: Verweise, viele Referenzen
- Eigenschaften
  - Komponenten: Bestandteile selbstständiger Systeme
  - Services: eigenstände Systeme
- Verteilung
  - Grundlage: Netzwerk
  - möglichst geringe Redundanz erwünscht
  - Verteilung daher Grundprinzip
  - eigenständige Systeme ermöglichen Verteilung
- Schnittstellen
  - Services bietet Operationen: Ausführungsanforderungen, Ein- und Ausgabe
  - Spezifikation der Schnittstelle notwendig: Vor- und Nachbedingungen, Invarianten, Syntax für Aufrufe
- Schnittstellen-Verträge
  - Schnittstelle ist Grundlage: Service bietet Operationen, Ähnlichkeit zu Komponenten
  - besondere Anforderungen an Stabilität
    - viele Nutzer, viele Organisationseinheiten, Änderungen nicht leicht möglich, Schnittstelle muss gleiches Angebot garantieren
  - daher: „Verträge“ für Schnittstellen
    - verbindliche Definitionen, technische Grundlage unverändert, Organisationsfrage
  - Erreichung der Stabilität
    - Präzision der Definitionen, Dokumentation, organisat. Verpflichtungen, Versionierung
- Analyse und Design
  - Modellierung
    - welche Modelle sind geeignet für Services?
    - Beschreibung der Schnittstellen, Interaktion von Services
  - Wiederverwendung
    - Spezifikation soll nur einmal notwendig sein, Zuständigkeit für Aufgaben & Datenhaltung
  - lose Kopplung: Verträge, Schutz gegen Ausfälle
  - gemeinsame Definitionen
    - Welche gemeinsamen Informationen sind verfügbar?
    - Daten, Datentypen

  - Granularität
    - Welcher Service erfüllt welche Aufgabe?
    - Aufgabenteilung und Wiederverwendung
  - Vorgehen
    - Identifikation von Services
    - Kategorisierung von Services (inhaltlich, technisch)
  - Spezifikation von Services: Schnittstellenbeschreibung
  - Realisierung: Implementierung, organisatorische Einbindung (Veröffentlichung, Migrationen)
- Umsetzung
  - relevante Eigenschaften
    - eigenständige Systeme, Abstraktion von technischen Details
  - Ableitung von Anforderungen
    - Offenheit, Skalierung, Fehlertoleranz, Transparenzen, …
    - nicht mehr bezogen auf ein System, sondern eine ganze Systemlandschaft
  - Technologien sind etabliert
  - CORBA erfüllt relevante Eigenschaften bereits seit ca. 20 Jahren
    - Offenheit, Protokolle, Integration, dezentrale Steuerung
  - Web Services erweitern technische Aspekte
    - Fokus auf Service-Idee
    - relevante Spezifikationen für Sicherheit, Transaktionen, …
    - wichtig: Web Services sind keine Services, wie sie hier besprochen werden (organisatorischer Aspekt fehlt, z.B. Analyse und Design), sondern nur Mittel zur Umsetzung
  - ebenso wichtig wie die Technik: Organisation

Service-orientierte Architekturen



- Motivation: viele Services, Koordinierung notwendig
- Strukturierung
  - Abstraktionsebenen
  - Modelle für Interaktion
  - Governance
- Modelle – Prozesse
  - Anforderungen
    - Abfolgen von Aktionen, Entscheidungen, Aktivitäten
  - Geschäftsprozesse
    - Ausführung durch ein System, koordinierte Services, Entscheidungen werten Anfragen aus, Aktivitäten stoßen Anfragen an
    - z.B. BPMN, BPEL, …
- Modelle – Ontologien
  - Anforderungen
    - exakte Definition gemeinsam verwendeter Daten, Vermeidung von Missverständnissen
  - Ontologien
    - Namensräume, Bedeutung, Beziehungen, Graph-Mechanismen
    - z.B. OWL, RDF, …
- Konsistenz und Transaktionen
  - Daten werden von verschiedenen selbständigen Systemen verwaltet
  - Daten referenzieren sich gegenseitig

  - lokaler Fall
    - DB verwaltet Transaktionen, singuläre Instanz zur Wahrung der Konsistenz
  - Fragen für SOAs
    - Wer kontrolliert Konsistenz bei unabhängigen Systemen?
    - Wie werden Transaktionen ermöglicht und sichergestellt?
    - Wie wird die Gültigkeit von Transaktionen bei teilweise verfügbaren Daten sichergestellt?
- Interaktivität
  - Anfragen werden über das Netzwerk versendet
  - Fragen für SOAs
    - Welche Granularität ist angemessen?
    - Welche Möglichkeit zu Rückmeldungen gibt es?
    - Welche Verzögerungen entstehen durch das Netzwerk?
- Design Patterns
  - offene Fragen führen oft zu ähnlichen Problemstellungen
  - Design Patterns für Programmierung
    - Vorlage für Lösungen, individuelle Anpassung für Spezialfall
  - Übertragung auf SOAs: Vorlagen für Strukturen
  - Patterns beschreiben
    - Problem, Lösungsweg, Anwendbarkeit (ggf. Einschränkungen), Auswirkungen
  - Inhalte
    - technisch: Strukturen der Systemlandschaft, Datenquellen, Verbindungen, …
    - organisatorisch: Entscheidungsstrukturen, Verantwortlichkeiten, Verträge, …
- Organisation
  - Organisationsstrukturen: Unternehmen, Abteilungen, Standorte
  - Personengruppen: Entscheider, Anwender, Fachabteilung, Administratoren
  - Fragen
    - Wer entscheidet über Standards/Implementierungen?
    - Wer erhebt Anforderungen?
    - Wie werden Änderungen erfasst und umgesetzt?
    - Wer administriert welches System?
  - Begriff: SOA Governance
    - systematische Verankerung im Unternehmen: Aktivitäten, Entscheidungen, Verantwortlichkeiten und Rollen

Fazit

- Motivation Komplexität: Verteilung, Heterogenität
- Ziele
  - systematische Architektur für große verteilte Systemlandschaften
  - Dienste im Fokus
- Inhalte
  - technisch: Verbindungen, Definitionen, Datenmodelle, …
  - organisatorisch: Zuständigkeiten, Verträge