Zusammenfassung EA

René Bernhardsgrütter, 26.12.2013
Cnt := Container; Anno[s] := Annotation; Mgmt := Management;

Komplexität

Heute hohe Komplexität in allen Bereichen:

- Verteilung
- Heterogenität
- Menge (Kopplungspunkte wachsen mit n2)
- Anwendungen umfangreich
- Unterschiedliche, anspruchsvolle Benutzergruppen
- Gesetzliche Auflagen

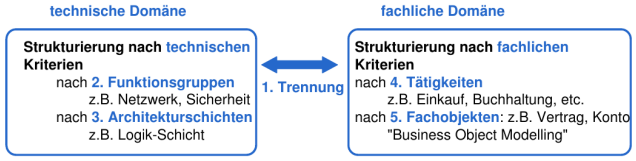
Homogenes System eines Herstellers: Technischer Fortschritt führt zu laufender Migration, das zu ständigen heterogenen Übergangsphasen. „All-fits-one"-Ansatz führt zu Kompromissen! Firmenabhängigkeit.
Heterogene Systeme, Offenheit: Man bringt verschiedene Teilsysteme zusammen. Das erfordert Offenheit von den Systemen, damit die Schnittstellen bekannt sind. [Service Oriented Architectures (SOA) versuchen dies wieder umzukehren, da dort Daten und Software in der Cloud laufen]

Handhabung im Grossfirmenumfeld

Zerlegung in kleinere Teile (Komponenten, Module, etc.). Strukturen so wählen, dass innerhalb der Komponenten eine hohe *Kohäsion* (=innerer Zusammenhalt) besteht. Zwischen den Komponenten sollte dafür eine umso losere (aber explizite) Kopplung bestehen.

Strukturierungsansätze

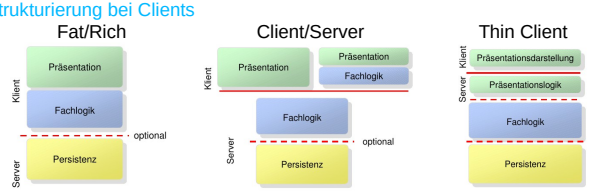
Nach Anforderungen/Dienste:



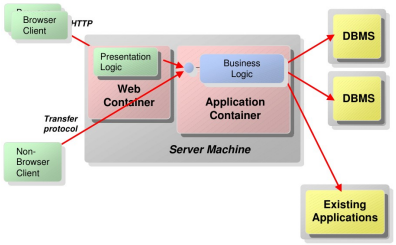
Nach Architektur-Schichten (wie OSI, etc.):
+ Unabhängigkeit bei Erstellung und Betrieb
+ Implementationen austauschbar
+ Licht(er) verständlich, wies funktioniert
- Performace-„Kosten" jeder Schicht
- Gewisse Änderungen nur schwer umsetzbar (z. B. eine zusätzliche Funktionalität kann alle Schichten betreffen)
- Wird häufig angewendet bei Softwarearchitekturen! (3 Tier)

Nach Aufgaben: Ähnlich dem Schichtenmodell, einfach z. B. nach Persistenz (L1), Fachlogik (L2) und Präsentation (L3) aufgeteilt.

Strukturierung bei Clients



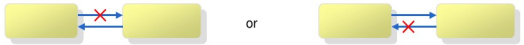
Generisches Modell einer 4-Tier Architektur:



Middleware

Macht, dass verschiedene Dienste unkompliziert zusammenarbeiten können:

- Mapping von Datentypen: Über mehrere Sprachen, löst Little/Big-Endian-Sache, führt zu einfacheren Schnittstellen
- Serialisierung, Byte-Order, kann Graphen/Ringe aufreissen
- IDLs: Kann auch komplexe Datentypen abbilden für das Kabel
- Naming, Location, Service Discovery: Finden von Diensten
- Fehlerbehandlung, QoS: Nur erneut bei Idempotentem probieren! Denn die folgenden Fälle sind ununterscheidbar:



- Transactions: Garantiert korrekte Ausführung einer Anfrage
- Access Control, Authentication: WinNT DC oder Kerberos
- Life Cycle Services: Lazy Loading und restart, falls Dienst down

Dimensionen

Proprietary, single vendor	vs	Standards-based, muliple v.
Small-scaled, leightweight	vs	Large-scaled, heavyweight
Request/Response	vs	Async
Traditional Client/Server Model	vs	Callbacks, Events
Language specific	vs	Language independent

Wahl: Kann in Firmen Jahrzehnte beeinflussen! Oft Design-Frage.

Middleware-Typen

RPC-mässig: simpel, es werden primitive Daten übertragen.
OOM: Object-oriented, z. B. RMI. Kompliziert, wenn 1 Objekt an 2 Orten.
Messaging: Async, daher „zeit-einfach", aber Fehlerhandling komplex.

Logging mit `log4j`

Soll dezentral/zental gespeichert werden, ggf. archiviert.

Logger-Klasse
Levels: `ALL > DEBUG > INFO > WARN > ERROR > FATAL`
Hierarchie:

```
rootLogger --- ch - zhaw
                  └--- zamp – flupo
                       └----- codebeamer
```

Regeln global (`rootLogger`) od. spezifisch festlegbar. Werden vererbt.
Layouts

Standardmässig wird das `PatternLayout` verwendet. Da können mit Variablen Sachen Strings gebildet werden.

Appender
Definieren, wo überall die Log-Streams geschrieben werden.

- ConsoleAppender
- FileAppender
- DailyRollingFileAppender: Pro Tag neues File, unkomprimiert
- SocketAppender/JDBCAppender: Entfernt bzw. in DB loggen

Konfiguration
Man kann's programmatisch oder im `log4j.properties` konfigurieren. Bei grösseren Programmen ist ersteres oft schlecht wiederzufinden, daher sind die Configfiles besser. Man kann die `log4j.properties` auch in einem separeten Jar deployen (für mehrere Projekte die gleichen Logging-Regeln).

Building mit `ant`

Führt automatisiert Aufgaben für das Builden, Dokumentieren, Testen, Deployen, Cleanen aus. Definiert in der `build.xml`.

Ein Standard-build.xml kann so aussehen:

```
<project name="myProject" basedir="." default="build">
    <target name="clean">
        <delete dir="./classes" /></>

    <target name="prepare" depends="clean">
        <mkdir dir="./classes" /></>
    <target name="build" depends="prepare, clean">
        <antcall target="clean" /></></>
```

Typische Targets: `init`, `clean`, `compile`, `run`, `deploy`, `default`, `jar`, `javadoc`, `test`. Etwa 70 Targets sind vordefiniert.
Properties definieren: `<property name="build.dir" value="build/" />` und verwenden `<mkdir dir="${build.dir}" />`
Properties können in ein eigenes File ausgelagert werden.
Tasks: `echo`, `copy`, `delete`, `move`, `mkdir`, `touch`, `get` (von URL), etc..
Man kann z. B. Dateien kopieren:

```
<copy file="myfile.txt" todir="dir/copyto" />
```

Man kann damit auch Dateien filtern:

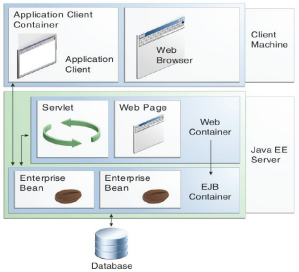
```
<copy todir="${build.dir}" >
    <fileset dir="${src.dir}">
        <include name="**/*.java" />
        <exclude name="test/**/*" />
    </fileset>
</copy>
```

Man kann auch eigene Tasks definieren:

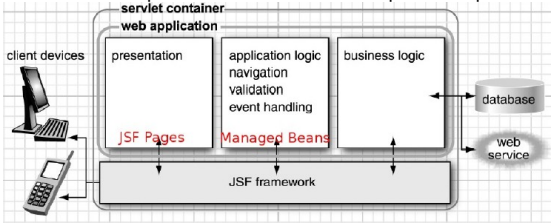
```
<taskdef name="mytask" classname="com.mydomain.MyVeryOwnTask"
    classpath="${basedir}/lib/MyVeryOwnTask.jar"/>
```

JSF: Java Server Faces

Komponenten- orientiertes GUI-Framework. Handhabt: Persistenz, Trennung von GUI und Logik, Flexible Navigation zw. Pages, Komponenten, Events von Links oder Ajax, Enterprise Java Beans (EJB), Template Engine mit #{controller.property}, Templates heissen Facelets Ermöglicht Trennung von Web-Designer und Programmierer.



Schaut für **Sicherheit**, korrekte **Transaktionen** (Single Threading, wo nötig), **JNDI lookup**, **Remote Connectivity** vom Coder abgenommen, Persistenz/ Service-orientiert.
Servlets: extenden HttpServlet und handlen die Requests/Responses.



Übersicht über ein Projekt mit einer Page:

1) faces-config.xml:
```
<?xml version... ?>
<faces-config
version="1.2"... />
<managed-bean... />
<navigation-rule... />
</faces-config>
```

2) JSF-Tags im Template:
```
<f:view>
  <h:form>
  ...
  </h:form>
</f:view>
```

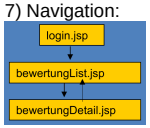
3) Konvertierung:
```
<f:convertDateTime
pattern="dd-MM-
yyyy" />
```
Datum 17-12-2007

4) Validierung:
```
<f:validateLongRange
minimum="0"
```
26
OK

5) Event:
```
action="#{loginController.verifyUser}"
```
ActionEvent
```
public String verifyUser()
  if(bf.verifyUser(username,password))
{
    return OUTCOME_LIST_BEWERTUNGEN;
  }else{
    return OUTCOME_LOGIN_PAGE;
  }
}
```

6) Managed Bean:
```
@ManagedBean
public class Klasse {
  String name;
  public void setName(String name){
    this.name=name;
  }
  public String getName(){
    return name;
  }
}
```

7) Navigation:



8) Lokalisierung:
```
de: Benutzername
en: username
it: utente
```

Java Beans
Wichtig: Es muss einen leeren Standardkonstruktor haben, damit ein Bean auch vom System ohne Argumente erstellt werden kann.

```
public class MyBean{
  String name;
  public void setName(String name){
    this.name=name;
  }
  public String getName(){
    return name;
  }
}
```

Managed Beans (MB)
Sind Beans mit der Anntotation `@ManagedBeans`. Features:
- Sammeln Daten der UI-Komponenten
- Implementieren EventListener
- Können Referenzen auf UI-Komponenten halten
- Können von Containern verwaltet werden (Tomcat)

View kann auf alle MBs zugreifen, daher das verwendete Bean angeben.

Backing Beans
Backing Beans sind JSF Managed Beans und für UI zuständig. Ggf. zusätlcihe Sachen für UI implementiert (Listener, Validators,..)

UEL: Unified Expression Language → Property Notation
Etwas mächtiger als bei Tapestry, da auch Business Logic in der Notation geschrieben werden kann (wie bei Velocity).
```
value="#{user.username}"
rendered="#{user.username !=null}
value="#{bill.sum * x}"
style="#{grid.displayed ? 'display:inline;': 'display:none;'}"
value="#{listBean[5]}"
value="Hello user #{user.username}"
action="#{user.storeUser}
actionListener="#{dataTableBean.deleteRow}"
value="#{mapBean['index']}"
value="#{mapBean['user.username']}"
```

h:dataTable
```
<h2>Programmers at #{company1.companyName}</h2>

<h:dataTable var="programmer"
             value="#{company1.programmers}"
             border="1">  The border attribute of h:dataTable just becomes the border attribute of table.
  <h:column>#{programmer.firstName}</h:column>
  <h:column>#{programmer.lastName}</h:column>
  <h:column>#{programmer.level}</h:column>
  <h:column>#{programmer.languageList}</h:column>
</h:dataTable>
```
Laufvariable `var` definiert innerhalb der `<h:dataTable>`-Tags.
Header kann mit `<f:facet name="header">Heading</f:facet>` kann zur Tabelle hinzugefügt werden. Dies ist ein vorkonfiguriertes Kindelement von `h:dataTable`. Es gibt auch `footer`, etc..
HTML-Attribute in `<h:dataTable>`-Tag können wie bei HTML verwendet werden (`border`, `bgcolor`, etc..).
CSS-Classes können mit den Attributen `styleClass`, `captionClass`, `headerClass`, etc. zu den Elementen hinzugefügt werden. Diese Classes werden im `<h:dataTable>`-Tag deklariert.

Standardkomponenten
Wichtige Standardkomponenten aus HTML- und Core-Library:

HTML-Library		Core-Library	
Code	*HTML-Replacement*	*Code*	*Beschreibung*
`<h:command Button>`	`<input type="submit">`	`<f:view>`	Oberstes Element. Tags sind immer innerhalb dieses.
`<h:commandLink>`	`<a href>`	`<f:attribute>`	Fügt der Parent-Komponente ein Attribut hinzu.
`<h:dataTable>`	`<table>`	`<f:facet>`	Ist ein SubElement eines anderen Elements.
`<h:form>`	`<form>`	`<f:validator>`	
`<h:inputText>`	`<input type="text">`	`<f:convertNumber>`	

Converter
`<f:convertDateTime pattern="dd-MM-yyyy" />`
Eigene Converter: Implementieren `Converter` und bieten `public Object getAsObject(..)` und `public void setAsObject(..)` an. Sie müssen in der faces-config.xml eingetragen werden.

Validatoren
`<f:validateLongRange minimum="0" maximum="100" />`
Validiert client- und serverseitig. Man kann mehrere kombinieren und es gehen auch Regex mit `<f:validateRegex pattern="..." />`. Eigene Validatoren implementiert `Validator` und bietet `public void validate(..) throws ...` Sie müssen in der faces-config.xml eingetragen werden.

Scopes
JSF hat 4 Scopes: Request/Conversation/Session/ApplicationScope.
In einer Session können mehrere Conversations stattfinden, z. B. beim Shop (Warenkorb → Lieferadresse → Zahlung).

Kommunikation mit Maps
Die einzelnen Komponenten können mit anderen kommunizieren, via Maps: Request/Conversation/Session/ApplicationMap.

```
// Context holen
FacesContext c = FacesContext.getCurrentInstance();

// Schreiben
c.getExternalContext().getSessionMap().put("username",username);

// Lesen, hier u = username
String u = (String)
c.getExternalContext().getSessionMap().get("u");
```
Auch aus Beans über den Controller. Aus den Views heraus mit `request/../applicationScope[,username']`.

Message(s)
Mit Tags können (Fehler-)Meldungen ausgegeben werden:
`<h:messages /> // alle Messages von der Page`
`<h:message for="label" /> // nur ein Component`
Die Tags orientieren sich an den Komponenten-IDs.

WEB-INF/faces-config.xml
Konfiguration von Model, View und Controller. Hat 3 Subnodes:
- `<managed-bean> // Stellt Bean-Instanz bereit`
- `<navigation-rule> // Navi-Regeln der App`
- `<application> // Allgemeines, z. B. Locale`

i18n
Wie bei Tapestry mit message[_de].properties, man muss aber die Sprachen in faces-config.xml:/application konfigurieren.

Facade-Pattern
Facade abstrahiert Zugriff von komplex zu einfach. Z. B. damit Libraries oder alte Software einfacher verwendet werden können.
JSF: SessionBeans sind Fassaden zw. aussen (GUI/REST/..) und Logik. Bei MVC dem Model zugeordnet, warum auch immer.

Navigation
Navigation-Hanlder können Action-Events behandeln (und Werte zurückgeben) und für Navi verwendet werden.

Implizit
Bei Methodenaufurf, welche als `return` weiterleitet:

```
<h:commandButton value="..." action="ziel.xhtml" />
<h:commandButton value="..." action="#{handlr.action}"/>
```
Nicht ratsam, da Verwechslungsgefahr mit Navigationsregeln besteht (faces-config.xml geht vor impliziter Navi!). Nur bei kleinen Projekten so machen, bei grösseren faces-config verwenden.

View-To-View
In faces-config.xml können Regeln eingetragen werden, die Routing einzelnen Views zu anderen Views definieren. Jede View (JSP-Seite) hat dabei eine eigene ID. Definiert durch
```
<navigation-rule> // mit Regex hier geschrieben
    (<(from|to)-view-id>|<from-(outcome|action)>)</>
```

Z. B. als Rückkehr von etwas:
```
<navigation-rule>
    <from-view-id>/pages/*</f>
    <navigation-case>
        <from-outcome>back</>
        <to-view-id>/pages/main.html</></></></>
```
Reihenfolge im XML ist dabei relevant.

Action-Event-Navigation
Wird dorthin navigiert, wenn die Methode aufgerufen wird:
```
<navigation-case>
    <from-action>#{naviHandler.rot}</>
    <to-view-id>/pages/nav/ganzrot.xhtml</></>
```

Bedingte Navigation
If/Else-Tags enthalten mit eq == equals, ge == greater-equals...:
```
<navigation-case>
    <from-outcome>bedingt</>
    <if>#{naviHandler.wertlt100}</>
    <to-view-id>/pages/bedingt1.xhtml</></>
```

Redirects
Leeres `<redirect />` lädt beim Browser die Page neu.

Externe Links
```
<h:outputLink value="http://www.jsfpraxis.de">
    <h:outputText value="DasBuch"/></>
```
Geht auch von Methode aus mit einem `ExternalContext`.

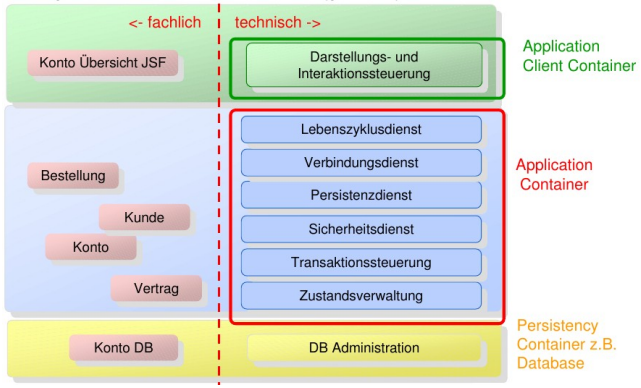
Includings
Andere JSF-Dateien mit `<ui:include src="input.xhtml"/>` einbinden. `input.xhtml` ist XHTML-Datei, die body und `<ui:composition>REPLACEMENT</>` ethält, wobei nur `REPLACEMENT` included wird. Statt `input.xhtml` kann auch ein #{..}-Ausdruck stehen, der dynamisch ausgewertet wird.

Komponenten
Erzeugbar in folgender Reihenfolge:
`composition <- component <- fragment <- decorate` wobei immer `<ui:NAME />`.

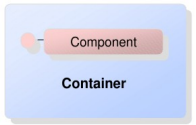
Ajax
```
<h:commandButton id="btn" value="cp"
   action="#{handler.action}">
   <f:ajax execute="@form" render="@form"/></>
```
action ist eine normale Methode.
execute: serverseitig zu verarbeitende Daten
render: Seitenteile, die nach Antwort neu angezeigt werden sollen
@form: Literal für „ganzes Formular"

Listener
```
<f:ajax listener="#{hndlr.actn}" render="e1 e2"/>
```
Bei jeder Änderung des umbgebenden Elements (z. B. Textfeld) wird der Listener aktiviert. Nachher werden `e1` und `e2` neu gezeichnet.

JEE
Trennen von Verantwortlichkeiten (Klassen), Aufteilen nach Rolle (Darstellung/Logik/Persistenz), Abtrennung von geteilten Diensten (business, technisch).

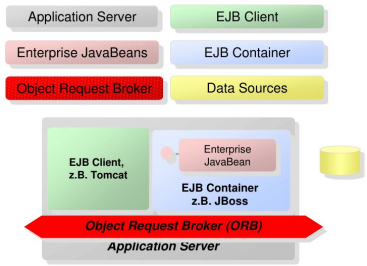


Komponenten (Kmp)
Unabhängige Funktionseinheiten mit Interfaces, leben in Laufzeitumgebung oder Container. Zu grösseren Funktionseinheiten aggregierbar. Auf Binärcodeebene wiederverwendbar, selbstbeschreibend bezüglich ihrer Dienste, konfigurierbar, einzeln deploybar.



Container (Cnt)
Laufzeitumgebung der Kmp. Verantwortlich für Lebeszyklus (Instanziierung, etc.) von Kmp, Sicherheit und andere Infrastruktursachen. Kmp können über Cnt-Schnittstellen die Dienste beanspruchen.

Vorteile: Schichtenarchitektur erlaubt Austausch von einzelnen Layern ohne alles zu ändern. 1 Logik für n Uis. 2-10-50-Regel (2 Jahre GUI, 10 Jahre Logik, 50 Jahre Daten). Man kann beste Lösungen von verschiedenen Anbietern nehmen.
Problem: Gefahr, dass one-fits-all zu komplex. Wenn Kmp eng untereinander gekoppelt sind, schwer entwickelbar.

JEE-Architektur


AppServer
Laufzeitumgebung für EJB-Cnt: Verwaltet Threads, Prozesse, Ressourcen, NW, Dbs. Linux, Windows, ...

EJB-Cnt
Stell Laufzeitumgebung als **Dienste** für **EJB** zur Verfügng.

Cnt/Bean-Interaktionen:
- Abfangen/Interception: Z. B. für Sicherheit/Transactions/Persistenz
- Aufruf: Beans können Cnt-Methoden aufrufen
- Rückruf: Beans über Interfaces/Annos angerufen werden (callback).

ORB
Ermöglicht Zugriff auf EJB-Methoden über Maschinengrenzen. Via JDNI werden JVMs/EJBs gefunden. RMI erforderlich, CORBA, SOAP möglich, Java Typen müssen serialisierbar sein.

EJB
Kmp welche App-Logik enthalten. Meist mehrere spezifische pro App (bspw: Kunde, Konto, Vertrag). Implementiert fachliche + technische Schnittstellen.

	Session Bean	***Entity Bean***	***MDB***
Haupt-anwendung	Prozess/Logik	Zustand	Vermittler
Zustand (indiv. für Klient)	stateless:nein statefull:ja	ja	nein
Zustands-/ Persistenz-verwalter	stateless:- statefull:Container	BMP:Bean CMP:Container	-
Lebensdauer des Zustands	Aufruf/Session	∞	Aufruf
Zugriff	entfernt/lokal	lokal	entfernt
Teil von Transaktion	optional	immer	optional

SessionBean (SB)
Repräsentiert **Prozess/Verhalten/Ablauf**, hat Business-Logik aber **keine** persistenten Daten. Ist stateless oder statefull.

```
@Stateless||Statefull
@Remote||Local||beides
public class HelloBean { // „Bean" zwingend!
   public String hello() { return „FU";}
}
```

Stateless (SLSB):
Speichert keine Infos zwischen Aufrufen, Beans werden in Pool verwaltet, schneller als Statefulls, Location-Transparent (ein Aufruf ist unabhängig vom einzelnen Server), Load-Balancing besser.

Stateless-Interfaces:
„@Remote": alle Methoden Teil des Remote-Interface
„@Remote(Reml.class)": Reml ist Interface
„@Remote"+"@Local": Hello ist Remote-I, HelloLocal ist Local-I
Nichts angegeben: automatisch alle public-Methoden in Local-I

Stateless-Lebenszyklus:
@PostConstruct: Methodenaufruf sofort nach Instanziierung
@PreDestroy: Methodenaufruf bevor ungenutzten Instanz im Pool vom Container zerstört wird

Stateful (SFSB):
Zustand wird zwischen Client-Aufrufen gehalten. Mit `.remove()` kann man ein Bean vom Client wieder lösen (oder automatisch nach z. B. 30 Min.). Weniger effizient, nicht Location-Transparent, da in Server-Ram gespeichert. Müssen Serializable sein.

Stateful-Lebenszyklus:
@PostConstruct @PostActivate: Sofort nach Instanziierung bzw. Deserialisierung aus dem Ram
@PreDestroy @PrePassivate: Methodenaufruf bevor Instanz im Pool vom Container zerstört bzw. wenn in Ram serialisiert
@Remove: Wenn .remove() gemacht wurde

Generell: JNDI-Lookup:
```
InitialContext c = new InitialContext();
helloBean=(cast)c.lookup("java:global/trail.HelloSession
"); helloBean.hello()
```
Container intercepted dies und erstellt Remote-Instanz als Bean.

EntityBean (EB)
Repräsentiert **Zustand**, persistiert Werte von Buisness-Entities, bildet Datenquelle auf Klasse ab. Jede Instanz hat einen Datensatz, z. B. Konto von Alice. Überlebt Verbindungs- und Container-Lebenzeit! Jedes EB hat eine eigene ID (UserId z. B.). Der EntityManager (EM) managed diese. SessionBeans greifen oft via EM auf EB zu, welche auf auf die DB zugreifen.

```
@Entity
@Table(name=“Dozenten“)
public class Dozent impl Serializable {...}
```

Service-Klasse für EM:
```
@Stateless public class DozentService{
   @Persist.Ctx.(unitName=“ds“) EntityManager em;
   add(Doz. d){em.persist(d);} // neuer Dozent
   rm(Doz. d){em.remove(d);} // Dozent löschen
}
```

MessageDrivenBean (MDB)
Realisiert **async** Prozess oder ist **Vermittler**. Erlaubt z**eitliche Entkopplung** von Verarbeitungsteilen. Inhaltlich wie das SessionBean. MDB sind Listener von Meldungs-Broker, werden via JMS angesprochen. Können mit anderen Systemen kommunizieren und werden so auch als Vermittler verwendet.

Reliable Messaging: Von EJB-Conainer wird garantiert, dass die Meldung zugestellt wurde, nicht aber, dass sie korrekt verarbeitet wurde. Ist wesentlich komplexer für Entwickler, vor allem im Fehlerfall.

Einsatzzweck: Bei Spitzen können Meldungen gepuffert werden, wenn Empfänger nicht immer online, um non-blocking zu haben (z. B. bei langen Berechnungen).

Publish-and-Subscribe: Client sendet Msg an Topic-Kanal, jeder Subscriber erhält eine Kopie davon.

Queue (P-t-P): Client sendet Msg an Queue. Empfänger pollen die Queue und können die Msg filtern. Jede Meldung wird sicher mind. durch 1 Empfänger verarbeitet.

Java Messaging API: Connection Factories erzeugen Connections. Über Queue-Objekt werden Meldungen versendet. Mit Session-Objekt werden Meldungen konsumiert (Session erstellt Mesasge Producer, Message Consumer und Messges). Mit Message Producers versendet man Nachrichten effektiv. Die Message Consumers kann man pollen, um neue Nachrichten zu erhalten. Ein Message-Objekt ist die Nachricht und hat Header (Typ, Routing, ...) und Body (TextMessage, Object, ...).

Typen: TextMessage = String, MapMessage = Key/Value-Paare (alle Java-Typen erlaubt), ObjectMessage = JavaObj, StreamMessage = Streams ähnlich DataOutputStream, BytesMessage = Byte-Rohdaten.

Beans: @MessageDriven-Anno, wird von Pool verwaltet. Wenn neue Nachricht kommt, wird onMessage-Methode aufgerufen. Innerhalb des Beans ist es Single Threaded, also für andere Zugriffe gesperrt.

Remote Interface
Schnittstelle auf jedes EJB: BeanName**Remote**.java (Konvention), hat selbe Methoden wie originales Bean. Stub+Interface werden generiert.

Deployment Descriptor
ejb-jar.xml: Konfigurtationseinstellungen von EJB, werden vom Container am Anfang geladen.

EJB Implementation
Business-Logik in BeanName**Bean**.java (Konvention). Callback-Methode wird mit @PostConstruct deklariert.

Validators
Können via Anno vor Attribut gesetzt werden. Jedes Mal, wenn der Inhalt befüllt wird, wird geschaut, ob dieser dem Validator entspricht. Wenn nicht, wird dies zum Benutzer zurückgegeben.

Standardvalidatoren: @Not/Null, @AssertTrue/False, @Min/Max[Decimal] (Wertebereich), @Size(Textlänge/ Elementanzahl), @Digits(Anz. Ziffern), @Past/Future (Datum), @Pattern (Regex)

Validation API: Man kann eigenen Validator machen, Klasse muss einfach ConstraintValidator implementieren.

Aspektorientierte Programmierung
Anforderungen an Software:
- Funktional (Bankkonto): gut machbar mit OOP
- Technisch (Security, Logging, DB): nicht/schlecht mit OOP
- nicht-funktionale Anforderungen (Performance, Stabilität, ..): abgedeckt durch System-Architektur, allgemeine, nicht im Code lokalisierte Anforderungen

Begriffe:
- **Concern**: Anforderung an die Funktionalität eines Programm
- **Core Level Concern**: Hauptanforderung an Komponente (anwendungsspezifisch)
- **Cross Cutting/System Level Concerns**: sind schwer getrennt kapselbar, haben mehrfaches Auftreten in verschiedenen Objekten, behindern Wiederverwendung der Objekte, senken Verständlichkeit
- **Joint Points**: Typische Stellen im Programmfluss, z. B. vor/nach Methodenaufruf, bei Instanziierung von Objekten, im Fehlerfall
- **Advise**: Code des Aspekts, wird Joint Point ausgeführt
- **Code-Tangling (Durcheinander)**: gleichzeitige Präsenz mehrerer Aspekte in einer Klasse: schlechte Lesbarkeit, Wartbarkeit
- **Code-Scattering (Streuung)**: Verteilung eines Aspekt über verschiedene Klassen: erschwert Wartbarkeit, verletzt Kapselungsprinzip

Lösungen:
Deklarativ programmieren, mit **XMLs** oder **Annos**. Einerseits gibt es **Interception** durch Container, z. B. zum Logging. Andererseits auch **Dependency Injection** von Werten durch Container.

Annos
Verwendung:
```
@Obs pulic class MyClass{..}
@Override public void myMeth(){};
```
Definition:
mit @ vor *normalem* Interface, ohne Parameter:
```
public @interface Obs{} // ohne Parameter
```
Standard-Parameter (Methode muss „value“ heissen):
```
public @interface Obs{String value();}
=> @Obs(„hello“)
```
Mehrere Parameter:
```
public @interface Obs{String s(); int i();}
=> @Obs(s=“hello“, i=32)
```
Arrays als Parameter:
```
public @interface Obs{String[] sarr();}
=> @Obs(sarr=“h1“) ODER @Obs(sarr={“h1“, „h2“})
```
Attribute von Annos festlegen:
@Target: Annotierte Enität (lasse, Methode, ..)
@Retention: Sichtbarkeit der Anno (Compiler oder auch Laufzeit)
@Documented: Anno soll teil der Javadoc sein
@Inherited: Anno auch in vererbten Entitäten (z. B. Klasse) gültig

```
@Target({TYPE,METHOD,CONSTRUCTOR,PACKAGE})
@Retention(value=RUNTIME)
@Documented
public @interface Obs{..}
```

Annos über `obj.getClass().getAnnotations()` ausgelesen.

Vorteile: Trennung von Core und System Level Concerns, deklarative Programmierung und sind dort, wo sie wirken.

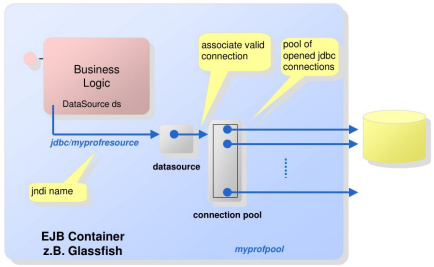
Nachteile: Können ohne Neu kompilation nicht geändert werden (im Gegensatz zu Config-Files).

Custom Interceptors
Infrastruktur-Aufgaben werden vom Code getrennt (Logging, Security, Transaktionssteuerung, etc.). Custom Interceptors werden vor/nach Aufruf angestossen.

Impl: Klasse mit Methode, welche @AroundInvoke-Anno hat. In der Methode wird entschieden, was gemacht wird.

Verwendung: In EJB wird mit @Interceptors(MyInterceptor.class) vor der Klassendeklaration gesagt, dass der Interceptor verwendet werden soll. Jeder Aufruf der Klasse geht dann durch den Interceptor. Alternativ in ejb-jar.xml für alle EJB festlegen.

DataSource
Alternative zu JDBC.
```
InitialContext c = new InitialContext();
DataSource ds = (cast) ctx.lookup("jdbc/myds");
Connection co = ds.getConnection(); ...
co.close();
```



Abfrage wie gewohnt mit PreparedStatements.

Java Persistence Architecture (JPA)

Java Persistence Query Language (JPQL)

Objektorientiert, frei von Fremd-/Schlüsselbeziehungen in WHERE condition, Ergebnis immer ein Objekt, Polymorphism möglich (Vererbung funktioniert)
SQL-ähnliche Abfragen möglich:

```
SELECT p FROM Dozent r WHERE r.name LIKE :name
```

Datenbankunabhängig (keine Dialekte), `:name` ist eine Variable vom Java-Aurfuf mit `.setParameter(„name“, namePattern)`, das Resultat kann eine Collection sein.

Klassen-Annos

@Table: Gibt den Datenbank Tabellen Namen an, falls nicht identisch mit Java-Klasse.
@SecondaryTable: Falls Felder der Entity-Bean in mehrere Tabellen abgelegt sind. Hat Parameter JoinColums, womit SecondaryTable mit PrimaryTable gejoint werden kann.

Attribut-Annos

@Transient: Für reine Instanz-Variablen, also nicht in Tablle.
@Column: Überschreibt Standard-Config von Spalten (Name, Länge, unique, nullable, ..).
@Enumerated: Java-Enum wird in DB als String od. Zahl gespchrt.
@Lob: Large Object: In Java i. d. R. als String gespeichert.
@Temporal: Datums-/Zeitpunktfeld.
@Version: Bestimmt Varialbe für optimistisches Locking.
@Id: PrimaryKey von Table.
@GeneratedValue(strategie=GenerationType.AUTO): Erzeugt PrimaryKey von @Id-Anno mit jeweils neuer ID..

Optimistic Locking

Wenn Wert gändert, wird Version um 1 erhöht. Wert wird vor jedem Schreiben ausgelesen. Wenn sich anderer persistenter Wert verändert hat, muss man Daten neu verarbeiten.

Beziehungen

1:1: 1 Kunde mit 1 Adresse. Zwei Klassen mit je @Entity. Person kennt Geb.Ort, Geb.Ort kennt Person ggf. auch.
1:n: 1 Person hat mehrere Adressen. Person hat @OneToMany Anno auf Collection<Adresse>. Kann auch via Join-Table sein, dann Anno @JonTable hinzufügen. Wenn Adresse Person kennt, hat es in Adresse @ManyToOne auf Person-Attribut.
n:n: @ManyToMany bei Attributen beider Klassen.

Persistenzarten

Transitiv: Veränderungen an persistenten Objekten werden automatisch in die DB committed. Gibt ggf. noch mehr.

Fetch Types

Sagt, wie assoziierte Objekte von Relationen aufgelöst werden.
EAGER: Es werden sofort alle Objektreferenzen rekursiv geladen.
LAZY: Laden, wenn benötigt.

Peristence Context

Speichert die vom EM verwalteten Objekte. Kann Transcation scoped sein (pro Transaction, sinnvoll für stateless beans) oder als extended persistence context über mehrere Transactions (sinnvoll für Satefull beans, z. B. wenn man etwas in den Warenkorb tut).

Transaktionen (Trns)

Es gilt das ACID-Prinzip (Atomic, Consistent, Isolated, Durable).

Ablauf: Start Trns {Änderungen machen} wenn keine Fehler {commit} sonst {Rollback}.

Bean Managed Trns Demarcation (BMTD): TrnsGrenzen können über Ressourcen (z. B. 1 DB) gesetzt werden. Der Trns Manager (TM) koordiniert das. Nested Trns bei EJB nicht zugelassen. Clients sollten keine Trns starten können, da Absturzgefahr und dann wäre DB gelockt.

Cnt Managed Trns Demarcation (CMTD): Cnt steuert Trns. Bei Bedarf wird eine gemacht. Das garantiert auch im Fehlerfall korrektes Rollback.

Deklarative TrnsSteuerung: Cnt steuert, Bean sagt, wie etwa.
@TrnsMgmt(CONTAINER): Cnt regelt TrnsGrenzen (default)
@TrnsAttribute(Attr): Die verschiedenen Attr sind REQUIRED (EJB soll innerhalb einers TrnsContext laufen, entweder ist dieser vom Client oder wird neu erzeugt), SUPPORTS (EJB wahlweise mit oder ohne TrnsContext ausgeführt, Entwickler zuständig), NOTSUPPORTED (falls Aufruf in TrnsContext, wird diese suspendiert bis Methodenaufruf fertig), REQUIRES_NEW (immer neuer TrnsContext erstellt), MANDATORY (muss in bestehendem TrnsContext aufgerufen werden, sonst Exception), NEVER (darf nie mit TrnsContext aufgerufen werden, sonst Exception).

Fehlerbehandlung: Kann setRollbackOnly gesetzt werden, dann gibts nur Rollback, kann auch Exception+Rollback oder nur Exception geworfen werden.
Verteilte Trns: Trns können mehrere EJB betreffen, z. B. wenn EJB1 eine Trns startet und dann EJB2 aufruft.

Entity Manager (EM)

Wird mit @PersistenceContext geholt. Verwaltet alle Entities und z. B. DB-Zugriff. Greift auf DataSource zu, welche in ejb-jar.xml definiert wurde.

Methoden

persist(): Neues Objekt in DB.
refresh(): Existierendes Objekt wird erneut ausgelesen.
remove(): Persistiertes Obj. wird gelöscht.
merge(): Nicht persistiertes Objekt wird unter existierender ID in DB geschrieben.
flush(): Alle Änderungen werden sofort in DB geschrieben.
find(): Objekt wird gesucht.
createQuery(): query erstellen und absetzen.
close(): Gibt alle Ressourcen des EM wieder frei

Zustände von EJB in EM

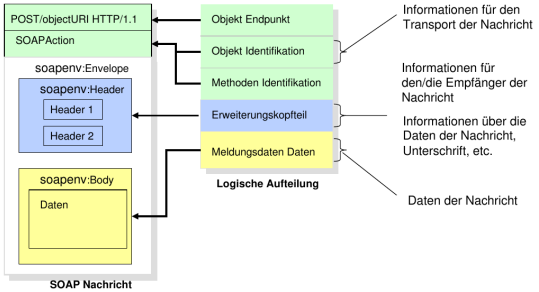
new/transient: EM kennt Bean nicht, auch nicht persistent.
managed: Bekannt und von EM verwaltet, auch persistent.
detached: Bean in DB aber nicht in EM.
removed: Zum Löschen markiert.

Callback-Annos

@Pre/PostPersist, @Pre/PostRemove, @Pre/PostUpdate, @PostLoad.
Methoden der Entity-Klassen können mit diesen Annos deklariert werden, womit diese zu Callback-Methoden werden. Bei der Klasse muss man noch @EntityListener hinzufügen.

WebServices

SOAP = **S**imple **O**bject **A**ccess **P**rotocol = Standardformat für Transport von XML via HTTP, SMTP, FTP. Ist stark erweiterbar, z. B. um Sicherheitsfeatures oder Proprietäres. Sehr komplex. Server muss Endpoint.publish veröffentlichen, Client ruft das ab (Schnittstellendefinition). Client/Stub können generiert werden



WSDL = **W**eb **S**ervice **D**escription **L**anguage = Beschreibt Schnittstelle von Web Service (Ein/Ausgabe-Parameter, ...). Von Mensch/Maschine lesbar. Kompliziert. Über Umbegungen hinweg standardisiert.

UDDI = **U**niversal **D**escription, **D**iscovery, and **I**ntegration = Verzeichnis von Web Services in dem gesucht werden kann.

Dependency Injcetion

Bei vielen Sachen möglich, z. B. DS oder EJB:

```
@Resource(mappedName = "jdbc/myprofresource")
public DataSource DozentenDS ;
@EJB(mappedName="Pfad zu RemoteSB")
Professor prof;
```

Monitoring/Management Services

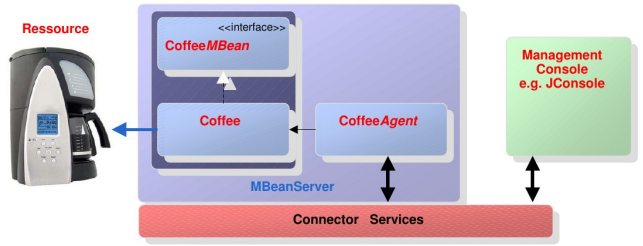
Viele Server/Services/Beans sollen auf einem Schirm einheitlich überwacht und konfiguriert werden können. Problem: Viele Komponenten, unterschiedliche Standards.

- Performance-Daten: Visits von Servlets/JSP, Ressurcennutzung, Antwortzeit, Logs/Probleme
- Transaktiionen: Anz. Commits/Rollbacks pro EJB, Ø Anz. Beans gelockt
- Connection Pools: Anz. Connections, Ø Wartezeit auf Connection
- Eigene App: Lesen/Schreiben von Parametern zur Laufzeit
- Alarmierung: Bei Fehler, speziellem Event

Java Mgmt Extension (JMX)

Besteht aus Architektur und API von Java Apps und Infrastruktur-Komp.

*RessourceName***MBean** (Konvention) ist Interface
RessourceName implementiert das Mbean-Interface
*RessourceName***Agent** registriert Mbean



Jedes Java-Objekt kann als JMX dargestellt werden, wenn Mbean vorhanden.
Standard MBeans: Statische Schnittstelle mit Interface.
Dynamic MBeans: Schnittstelle durch MBeanInfo-Klasse definiert und kann **zur Laufzeit** verändert werden, dafür komplexer.

OSGi

Ist dynamisches Modulsystem für Java. Bitete Bundles (Softwarekomponenten) und Services (Dienste). Können zur Laufziet geladen/gestoppt werden.

Bundles

Modul mit zusammenhängenden Klassen und Ressourcen. Ist eine JAR-Datei. Hat ein Manifest und eine Activator-Klasse.

Aktivierung

Wird via Bundle-Activator gestartet.

Services

Einfaches Java-Objekt (POJO), wird aber an zentraler Stelle registriert. Können registriert+gestartet werden. In Slieds nachschauen, wie genau.

Deployment

Releases

Major: Neue Features -> Benutzerschulung, Kompatibilitätsbruch.
Minor: Keine neue (wesentliche) Funktion hinzugekommen, notwenige Anpassung wg. Änderung der Infrastruktur, Interfaces bleiben i. d. R. erhalten. Fehlerkorrekturen.
Emergency: Sofortige Korrektur ist notwendig, z. B. bei kritischer Sicherheitslücke.

Versionierungsmodell: Major.Minor.Build

Wenn es beim Upgrade/Update Probleme gibt, muss schnell zum alten System zurück gewechselt werden können.

Verteilungsmodell

Push: Server pusht synchron die neue Version zu allen Clients.
+ Kontrolle und sofortige Fehlerreaktion
+ Zentrale Steuerung und Kontrolle
+ Wissen, welche Clients werlche Version haben.
+ Update kann erzwungen werden.
+ Einfachheit

Pull: Client holt selber neue Version.
+ Async: Netzwerk weniger ausgelastet zur gleichen Zeit.
+ Orphan Client Update möglich: Beim nächsten Start Update, wenn Rechner über längere Zeit nicht mehr verwendet.

Big Bang vs Phased

Big Bang: Neue Software übers Wochenende installiert
+ Einfache Datenmigration
+ Altlasten rauswerfbar
+ Keine Unklarheiten, welche Software verwenden

Phased: Alte und neue Software laufen Parallel
+ Kein Zeitdruck
+ Weniger Risiko eines Betriebsunterbruchs
+ Fallback einfach
+ Benutzer kann bei Problemen das alte System verwenden
+ Orphan Client Updates ebenfalls möglich

Verteilungsablauf

1) **Experimental**: Für Infrastruktur
2) **Test**: Für Entwickler
3) **Integration**: Benutzer-/Integrationstests (schauen ob SW in Env. läuft, ob es mit realen Daten zurecht kommt, auch ob die Users damit klar kommen)
4) **Produktion**: *Scharfes* System

Softwareauswahl

Benutzerauswahl:
+ Kennt für ihn geeignete Anforderugnen und Anwendung
+ SW-Entscheidungen werden auf der richtigen Ebene getroffen

Zentrale Insanz:
+ Qualitäts- und Sicherheitskontrolle einfacher
+ Langlebigkeit einfacher sicherstellbar

Pakete oder Einzelsoftware

Wird unterschieden zw. Paketen (Office Familie) und Einzelsoftware (einzelnen Schreibeprogrammen).