# Fachtheorie Portfolio

## Inhaltsverzeichnis

# 1. UML - Klassendiagramm

Klassendiagramme dienen der Beschreibung von Aufbau und Zusammenspiel von Klassen. [Mar14]

## 1.1 Aufbau von Klassen

Im Klassendiagramm werden die Klassen durch Rechtecke dargestellt. Das Rechteck wird dabei vertikal in vier Bereiche unterteilt. Der obere Bereich enthält den Namen und weitergehende Informationen zu der Klasse. Der zweite Bereich enthält die Attribute der Klasse. Der dritte Bereich enthält die Methoden (Operationen) der Klasse. Der vierte und letzte Bereich der Klasse enthält Eigenschaften der Klasse. [Mar14]


Die Bereiche für Attribute, Methoden und Eigenschaften sind optional. Das einfachste Klassendiagramm ist ein einfaches Rechteck, das den Namen der Klasse enthält. Wird nur ein Bereich ausgelassen (z.B. Attribute), sollten jedoch die entsprechenden Trennlinien angezeigt werden. [Mar14]


**Namensbereich**
Im Namensbereich werden die Angaben zentriert angeordnet. Der Name der Klasse wird dabei fett geschrieben. Oberhalb des Klassennamens können in doppelten spitzen Klammern Stereotypen angegeben werden. Ein häufig verwendeter Stereotyp ist: << Interface >> der angibt, dass es sich nicht um eine Klasse, sondern um eine Schnittstellendefinition handelt. [Mar14]

Unterhalb der Klasse können Eigenschaften der Klasse in geschweiften Klammern angegeben werden (z.B.: {abstract}).

**Attributsbereich**
Die Attribute einer Klasse werden im zweiten Bereich des Klassendiagrammes linksbündig angegeben. Die allgemeine Form der Angabe von Attributen ist: [Mar14]

```
+ Name : Typ = Wert
```

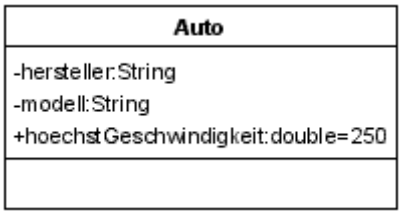
Als erstes wird ein Sichtbarkeitssymbol angegeben (+). Nach dem Sichtbarkeitssymbol wird der Name und der Typ des Attributs durch einen Doppelpunkt getrennt angegeben. Hat das Attribut einen definierten Wert (Initialwert), kann dieser nach einem Gleichheitszeichen angegeben werden. [Mar14]

Folgende Zeichen sind für die Angabe der Sichtbarkeit definiert:

```
+     public
#     protected
-     private
~     package
```

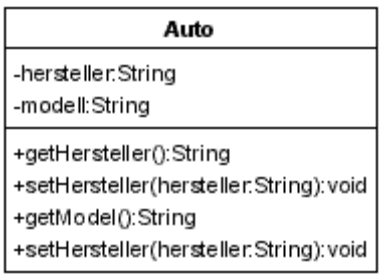
Je nach Detaillierungsgrad bzw. Phase der Softwareentwicklung können Sichtbarkeit, Typ und Initialwert als optional angesehen werden. [Mar14]

Das nachfolgende Beispiel zeigt ein Klassendiagramm mit Attributen:




**Methodenbereich**
Im dritten Bereich des Rechteckes werden die Methoden der Klasse linksbündig angegeben. Als erstes wird das von den Attributen bekannte Zeichen für die Sichtbarkeit angegeben. Hinter dem Sichtbarkeitszeichen folgt der Name der Methode. Abschließend folgt ein rundes Klammerpaar, in dem gegebenenfalls die Parameter der Methode angegeben werden. Ist der Methode ein Rückgabewert zugeordnet, wird dieser hinter einem Doppelpunkt angegeben. Für den Fall das die Methode keinen Rückgabewert hat kann als Rückgabewert void angegeben werden. [Mar14]

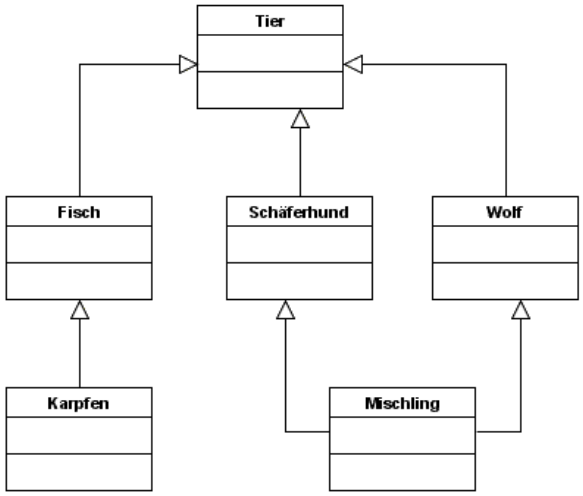
Das nachfolgende Beispiel zeigt ein Klassendiagramm mit angegebenen Methoden:



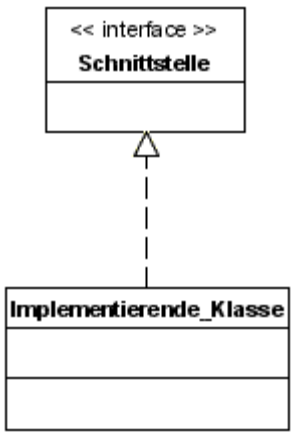
**Abstrakte Klassen und statische Elemente**
Abstrakte Klassen werden in der UML durch Darstellung des Klassennamens in kursiver Schrift dargestellt. Die Darstellung statischer Methoden und Attribute erfolgt durch unterstreichen des jeweiligen Elements. [Mar14]


## 1.2 Beziehungen von Klassen

### 1.2.1 Vererbung
Die Veerbung von Klassen untereinander wird durch eine durchgezogene Linie mit einem geschlossenen nicht ausgefüllten Pfeil, der auf die Oberklasse zeigt, dargestellt. [Mar14]

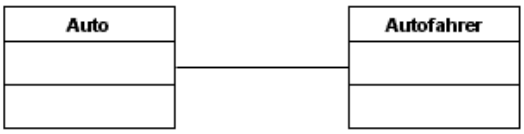



### 1.2.2 Realisierung (Schnittstellenimplementation)
Die Realisierung bzw. Implementation einer Schnittstelle wird mit einer gestrichelten Linie dargestellt. An der Schnittstelle wird ein geschlossener, nicht ausgefüllter Pfeil angebracht. [Mar14]

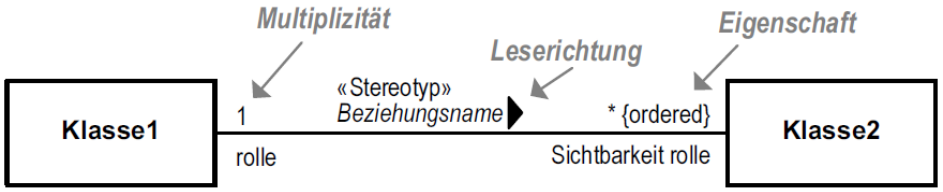


### 1.2.3 Assoziation

Eine Assoziation von Klassen wird durch eine durchgezogene Linie, die beide Klassen miteinander verbindet, dargestellt. Eine Assoziation beschreibt eine Beziehung zwischen zwei oder mehr Klassen. An den Enden von Assoziationen sind häufig Multiplizitäten vermerkt. Diese drücken aus, wie viele dieser Objekte in Relation zu den anderen Objekten dieser Assoziation stehen. [Mar14]




**Beschreibungselemente einer Assoziation:**



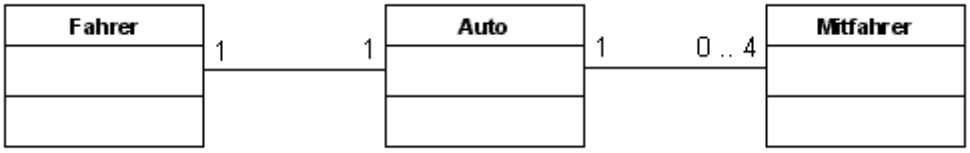

**Multiplizität**

Die Multiplizität eines Objektes gibt an, wie viele Objekte des einen Typs mit Objekten des anderen Typs (oder Objekten anderer Typen) verbunden sein können oder verbunden sein müssen. [Mar14]

Folgende Angaben zur Multiplizität können angegeben werden:

| Beschreibung | Beispiel | Bedeutung |
|---|---|---|
| Angabe einer definierten Zahl | 3 | Genau diese Anzahl von Objekten |
| Angabe von Minimum und Maximum | 3 .. 7 | Mindestens 3 Objekte und maximal 7 Objekte |
| Der Joker | * | Beliebig viele Objekte |
| Angabe von Multiplizitäten durch Trennung von Kommata: | 1, 3, 6<br>0, 4 .. 9<br>0 .. *<br>*<br>4 | Ein, drei oder sechs Objekte<br>Entweder kein oder vier bis neun Objekte<br>Beliebig viele Objekte<br>Beliebig viele Objekte<br>Genau vier Objekte |

Das nachfolgende Diagramm zeigt die Assoziation von Auto, Fahrer und Mitfahrer bei einer Autofahrt. Es wird dabei davon ausgegangen, dass der Wagen für 5 Personen zugelassen ist:

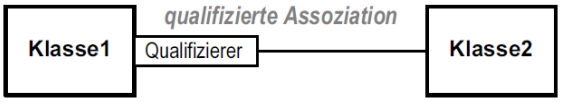


**Gerichtete Assoziation**

Gerichtete Assoziationen sind Beziehungen, die nur in eine Richtung navigierbar sind. Dargestellt wird die Navigation durch eine offen Pfeilspitze, die die zugelassene Navigationsrichtung angibt. [Uml15]

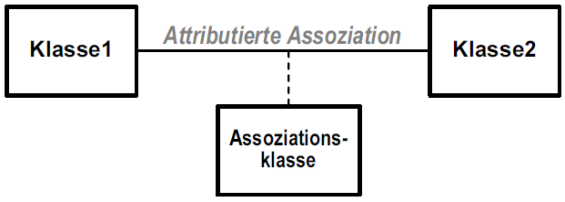

**Qualifizierte Assoziation**

Beziehungen, bei denen ein Objekt viele (*) Objekte der gegenüberliegenden Seite assoziieren kann, werden durch Behälterobjekte implementiert. Beispielsweise wird ein Dictonary definiert, bei dem der Zugriff jeweils durch Angabe eines Schlüssels erfolgt. Diese Schlüssel sollten schon beim Design als qualifizierende Attribute angegeben werden. Diese werden dann in der Notation als Rechteck an der Seite der Klasse dargestellt, die über diesen Schlüssel auf das Zielobjekt zugreift. Der Qualifizierer ist Bestandteil der Assoziation. Es wird bei dieser Beziehung ausschließlich über den Schlüssel navigiert. [Uml15]



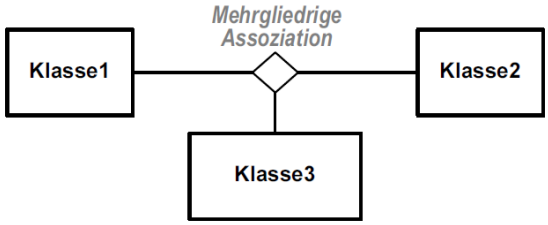

**Attributbasierte Assoziation**

Es existiert auch eine Form in der die Assoziation selbst über Attribute verfügt. Diese Assoziationsattribute sind dann existenzabhängig von der Assoziation. Man spricht von sogenannten attributierten Assoziationen bzw. von degenerierten Assoziationsklassen, da die Klasse keine eigenständigen Objekte beschreibt. [Uml15]



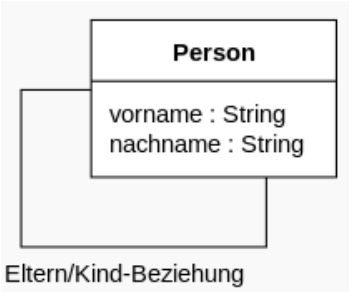
**Mehrgliedrige Assoziation**

Eine weiter Form der Assoziation ist, neben der Zweierbeziehung und der attributierten Assoziation, die mehrgliedrige Assoziation. An ihr können drei oder mehr Klassen beteiligt sein. Sie sollten wegen ihrer komplexen Strukturen explizit als Klasse modelliert werden. [Uml15]




**Reflexive Assoziation (Eltern/Kind-Beziehung)**

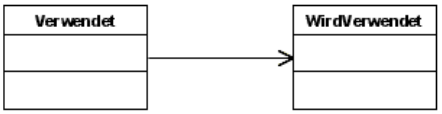
Die beiden Enden der Assoziation zeigen hier auf den gleichen Typ.



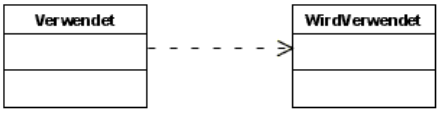
Eltern/Kind-Beziehung

### 1.2.4 Verwendung (Abhängigkeit)

Verwendet eine Klasse eine andere, wird dies mithilfe einer durchgezogenen Linie dargestellt, wobei an der verwendeten Klasse ein offener Pfeil angebracht wird: [Mar14]

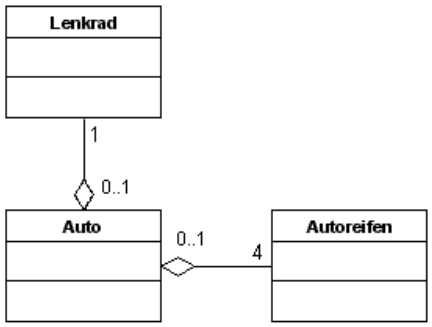



Wird eine Schnittstelle der verwendeten Klasse verwendet, wird eine gestrichelte Linie mit einem offenen Pfeil verwendet. An der verwendeten Klasse kann optional die Multiplizität angegeben werden: [Mar14]



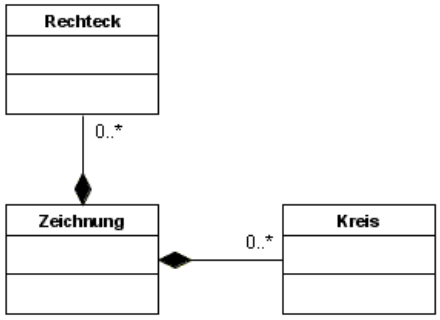

### 1.2.5 Aggregation

Eine Aggregation wird im Klassendiagramm mithilfe einer durchgezogenen Linie dargestellt. Am Ende, wo das Ganze dargestellt wird, wird eine offene Raute gezeichnet. An diesem Ende ist die Multiplizität entwender 0 oder 1. Am anderen Ende (da wo keine Raute ist) können auch höhere Multiplizitäten vorliegen. [Mar14]




### 1.2.6 Komposition

Eine strengere Form der Aggregation ist, wenn die einzelnen Bestandteile vom ganzen existenzabhängig sind. Diese Abhängigkeit wird Komposition bezeichnet. Die Darstellung erfolgt ähnlich wie die der Aggregation, nur dass die Raute ausgefüllt wird und am Teil des Ganzen implizit die Multiplizität 1 angenommen wird. [Mar14]



# 2. Vererbung/Polymorphie

Klassen können von anderen Klassen erben und können belieibig viele Interfaces implementieren. Es besteht dann eine „Ist-Ein"-Beziehung zwischen Subklasse und Superklasse. Folglich werden wichtige Begriffe der Vererbung erklärt: [Ber15]

- **Klassifikation**:
  Eine Klassifikation ist eine Zusammenfassung verschiedener Objekte, die etwas gemeinsam haben zu einer Klasse.

- **Generalisierung**:
  Ist die Verlagerung der Eigenschaften von Subklassen auf die Superklasse.

- **Spezialisierung**:
  Ist die Implementierung einer Klasse, die die Eigenschaften und Fähigkeiten einer übergeordneten Klasse erweitert (Gegenteil von Generalisierung)

- **Faktorisierung**:
  Ist das Erkennen redundanter Implementierungen und Beseitigen der Redundanzen (z.B. durch abstrakte Klassen)

- **Überschreiben**:
  Bezeichnet das Implementieren einer Methode, die in der Superklasse bereits implementiert ist. Bei Methodenaufruf wird die Methode der Subklasse verwendet.

- **Überladen**:
  Bezeichnet das Impelementieren gleichnamiger Methoden in einer Klasse, die sich aber durch die Parameterliste unterscheiden.

- **Polymorphie**:
  Auch Vielgestaltigkeit, erlaubt das Speichern von Objekten der Subklasse in Objektvariablen der Superklasse. Das Speichern von vielgestaltigen Collections ist möglich.

- **Dynamische Bindung**:
  Die dynamische Bindung basiert auf dem Prinzip des Überschreibens. Zur Laufzeit wird die richtige Methode von Polymorphen Objektvariablen gesucht.

- **Abstrakte Methode**:
  Sind Methoden in einer Klasse, die noch nicht implementiert sind und durch Subklassen implementiert werden müssen.

- **Abstrakte Klasse**:
  Eine abstrakte Klasse ist eine Klasse, die nicht instanziert werden kann, weil sie ein oder mehrere abstrakte Methoden besitzt. D.h. von einer abstrakten Klasse kann kein Objekt erzeugt werden.

- **Finale Klassen und Methoden**:
  Finale Klassen bzw. Methoden dürfen nicht vererbt bzw. überschrieben werden. [Ber15]

# 3. Design Patterns

Design Patterns sind Entwurfsmuster, die folgende Ziele haben:

- Möglichkeiten, Code zu verbessern:
  - leichter zu implementieren
  - leichter zu erweitern
  - leichter zu warten
- Möglichkeiten, das Programmieren zu verbessern:
  - Effizienz zu steigern
  - Design-Fähigkeiten zu steigern
  - Qualität der Projekte zu steigern

[Eri04]

Man unterscheidet grundsätzlich zwischen strukturellen und verhaltensbezogenen Design-Patterns.

## 2.1 Strukturelle Design Patterns

Strukturelle Design Patterns sind Entwurfsmuster, die sich mit den Relationen zwischen Klassen beschäftigen bzw. diese nach den genannten Zielen von Design Patterns optimieren. [Eri04]

### 2.1.1 Adapter Pattern

Adapter Pattern arbeitet als eine Brücke zwischen zwei inkompatiblen Schnittstellen. Dieses Pattern kombiniert die Fähigkeiten von zwei unabhängigen Schnittstellen. [Tut15]

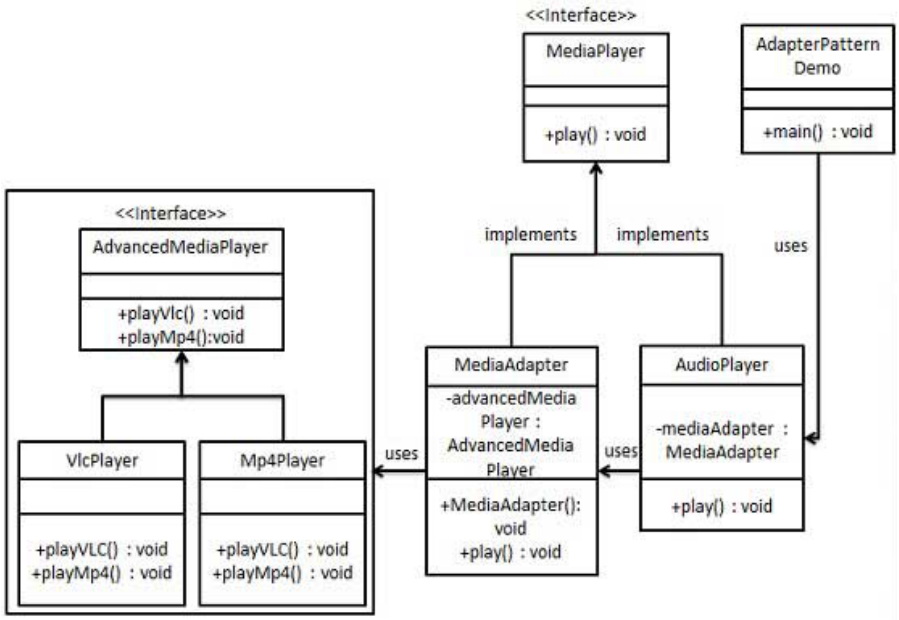



Abbildung: Beispiel für Adapter Pattern [Tut15]

### 2.1.2 Decorater Pattern

Decorater Pattern wird verwendet, um eine Klasse zu erweitern, ohne ihre Struktur zu ändern, indem ein Wrapper (Verpackung, Hülle) hinzugefügt wird. [Tut15]

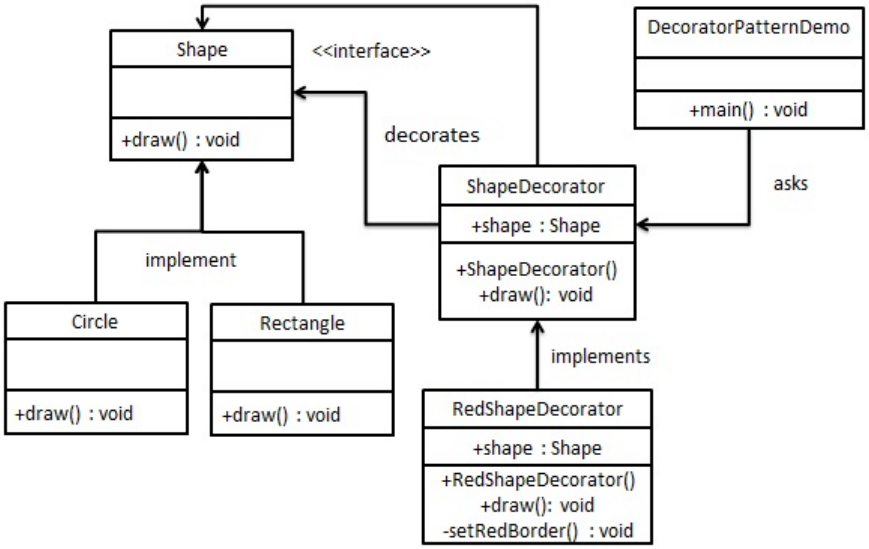



Abbildung: Beispiel für Decorater Pattern [Tut15]

**Beschreibung:**

- Interface, das die grundlegende Funktionalität definiert
- Abstrakte Klasse, die das Interface implementiert und ein Objekt einer implementierenden Klasse hat.
- Klassen, die die abstrakte Klasse konkret umsetzen und sich gegenseitig erweitern können (hier: RedShapeDecorater)

[Tut15]

### 2.1.3 Factory Pattern

Factory Pattern wird verwendet, um die Objekterstellung von der aufrufenden Klasse zu trennen. [Tut15]

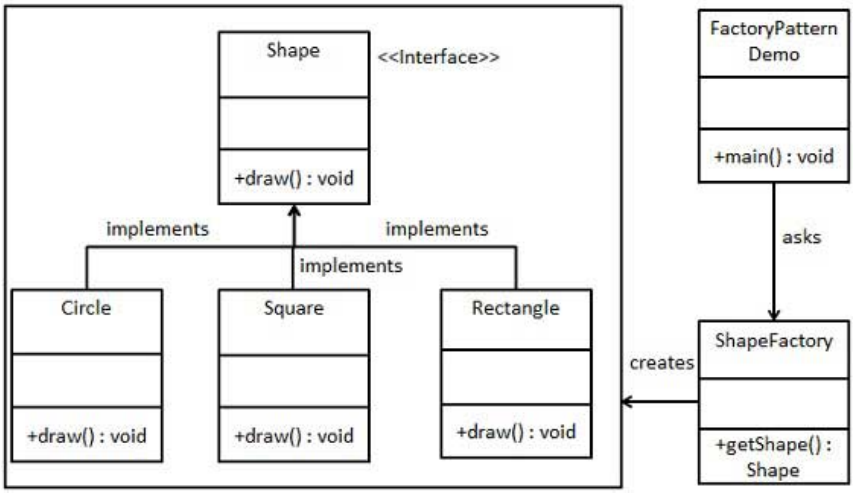



Abbildung: Beispiel für Factory Pattern [Tut15]

**Beschreibung:**

- Interface, das die Funktionalität definiert
- Mehrere Klassen, die das Interface umsetzen
- Anstatt Konstruktoren zu verwenden, erzeugt die Factory die Objekte der implementierenden Klassen nach Bedarf und gibt diese weiter.

[Tut15]

### 2.1.4 Abstract-Factory Pattern

Abstract-Factory Pattern arbeitet genauso wie der Factory Pattern, aber es wird noch zusätzlich eine weitere Abstraktionsebene hinzugefügt, indem Factories ebenfalls von einer Factory erzeugt werden. [Tut15]

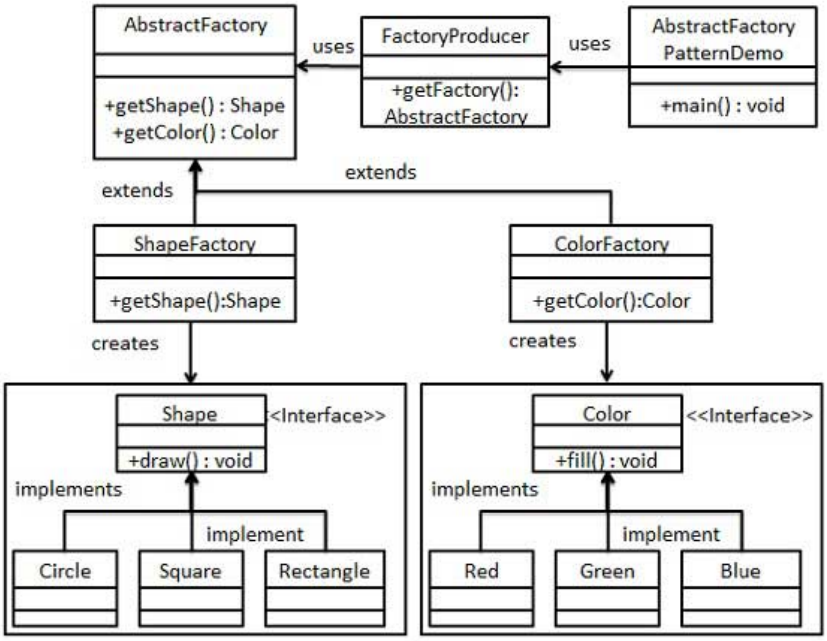



Abbildung: Beispiel für Abstract Factory Pattern [Tut15]

## 2.2 Verhaltensbezogene Design Patterns

Verhaltensbezogene Design Patterns sind Entwurfsmuster, die sich mit der Kommunikation zwischen Klassen beschäftigen bzw. diese nach den genannten Zielen von Design Patterns optimieren. [Eri04]

### 2.2.1 Observer Pattern

Observer Pattern wird verwendet, um über neue Daten zu informieren. [Tut15]

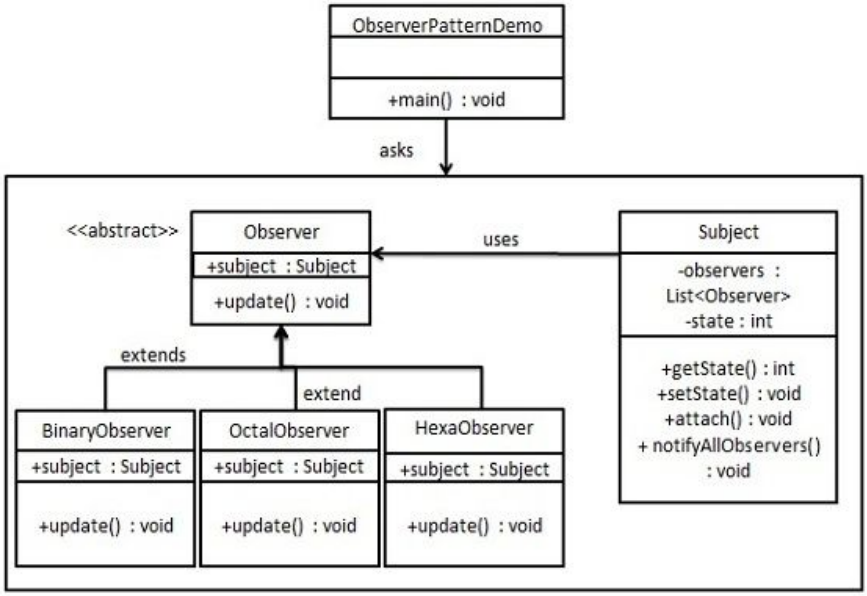



Abbildung: Beispiel für Observer Pattern [Tut15]

**Beschreibung:**

- Abstrakte Klasse Observer, die die Methode update() beinhaltet, in der neue Daten vom Subject angefragt werden.
- Eigentliche Klassen, die Observer konkret umsetzen.
- Subject verwaltet die Observer einer Quelle und kann diese hinzufügen, entfernen und über Änderungen benachrichtigen.

[Tut15]

### 2.2.2 Strategy Pattern

Strategy Pattern wird verwendet, um Verhalten während der Laufzeit austauschen zu können, indem die Strategy gewechselt wird. [Tut15]

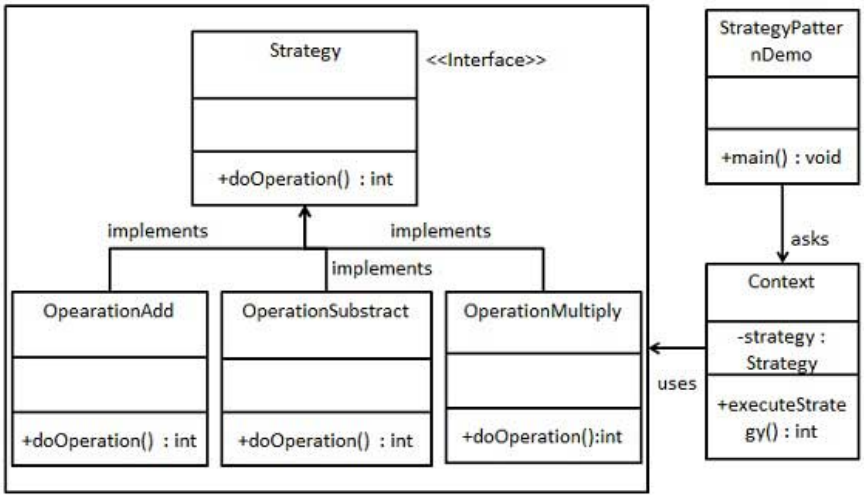



Abbildung: Beispiel für Strategy Pattern [Tut15]

**Beschreibung:**

- Interface definiert eine Funktionalität.
- Klasse hat ein Objekt einer Klasse, die das Interface als Attribut implementiert und ruft die Methode aus dem Interface auf.
- Mehrere Klassen, die die Funktionalität des Interfaces auf verschiedene Art umsetzen existieren und sind beliebig austauschbar.

[Tut15]

### 2.2.3 Command Pattern

Command Pattern wird verwendet, um Requests in einer Queue speichern zu können. [Tut15]

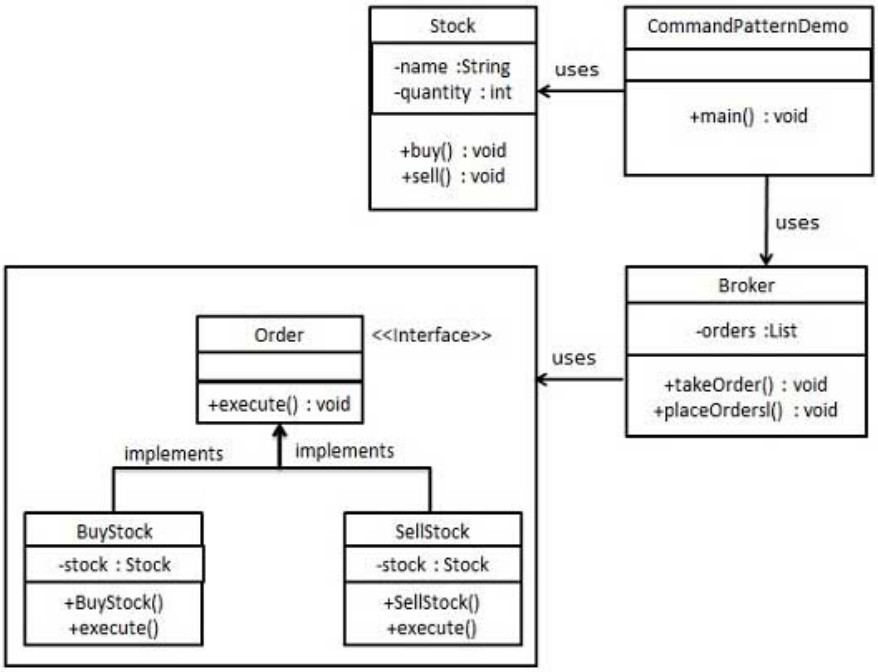



Abbildung: Beispiel für Command Pattern [Tut15]

**Beschreibung:**

- Interface definiert execute().
- Klassen setzen execute() um, Klassenname beschreibt Funktion.
- Eine Klasse, die pro Methode eine implementierende Klasse aufruft. (hier: Broker)

[Tut15]

## 2.3 MVC Pattern

MVC Pattern wird bei Programmen mit Benutzeroberfläche (GUI) angewendet, um die Datenhaltung, Anzeige (View) und die Steuerung von Programmen aufzuteilen. [Tut15]

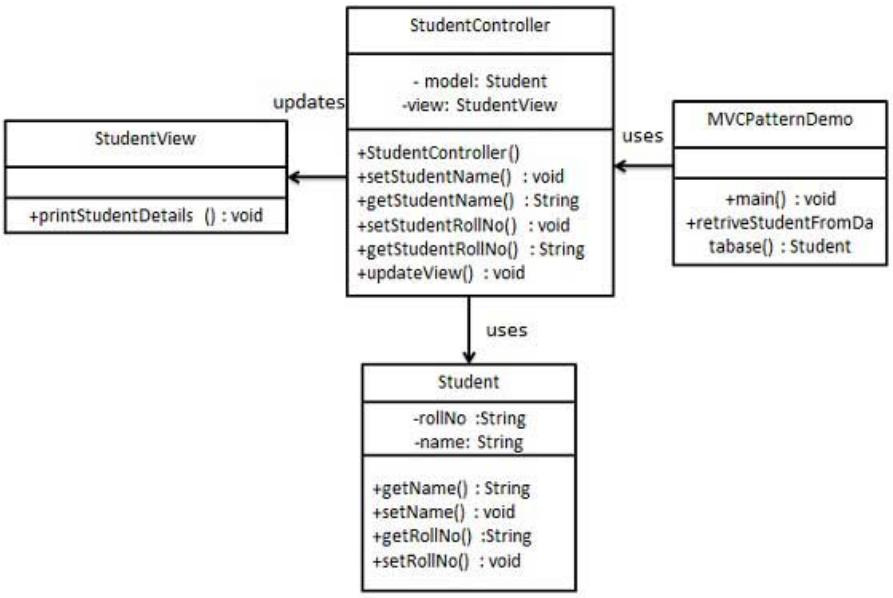



Abbildung: Beispiel für MVC Pattern [Tut15]

**Beschreibung:**

- Aufteilung in Model, View und Controller
- Model ist das Datenmodell
- View ist die Benutzeroberfläche
- Controller beinhaltet die Logik und verbindet Model und View miteinander

[Tut15]

## 2.4 Singleton Pattern

Singleton Pattern wird verwendet, um nur ein einziges Objekt einer Klasse zu erzeugen. [Tut15]

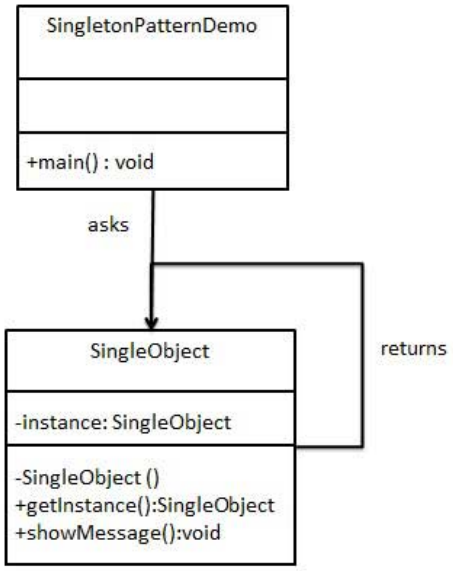



Abbildung: Beispiel für Singleton Pattern [Tut15]

**Beschreibung**:

- Klasse speichert Objekt von sich selbst beim ersten Konstruktoraufruf und gibt es über getInstance() zurück.

[Tut15]

# 3. Versionierung über GIT

Damit Sourcode und Dateien einer Applikation versioniert werden können, ist ein Tool für die Versionierung notwendig. Hierzu bietet sich die Verwendung von GIT an.

## 3.1 GIT-Befehlsliste

| Befehl | Beschreibung |
|---|---|
| git init | Erzeugt neues Repository |
| git status | Listet alle neuen oder geänderten Dateien |
| git diff | Alle Dateiunterschiede |
| git add [file] | Fügt Datei hinzu |
| git commit –m „Message“ | Commited die Änderungen mit der Beschreibung |
| git reset [file] | Entfernt Datei ohne zu löschen |
| git branch | Listet alle lokalen Branches |
| git branch name | Erzeugt einen neuen Branch |
| git branch –d name | Löscht den Branch |
| git checkout name | Wechselt zu dem Branch |
| git merge name | Fügt den Branch mit dem aktuellen zusammen |

## 3.2 Vorgehensweise bei der Versionierung eines Projektes

1. Ordner erstellen bzw. bestehenden Ordner verwenden, indem sich die zu versionierenden Daten befinden bzw. befinden sollen
2. `$ git init`
3. `$ git remote add origin <Link_zum_Repository>`
4. Änderungen durchführen im initialisierten Ordner
5. `$ git add *`
6. `$ git commit –m “Daten geaendert”`
7. `$ git push origin master`
8. Anmelden mit dem Benutzernamen und Passwort

Wenn man sich Daten vom Remote-Repository runterladen möchte, dann muss man folgenden Befehl verwenden:

```
$ git pull origin master
```

Anschließend sollten sich alle Änderungen im lokalen Repository (Ordner) befinden.

Wenn man den derzeitigen Zustand im lokalen Ordner anschauen möchte, kann man den folgenden Befehl verwenden:

```
$ git status
```

Die Verwendung von diesen Befehlen sollte für eine schnelle Versionierung ausreichend sein.

# 4. Java Enterprise Edition (JEE)

## 4.1 Architektur

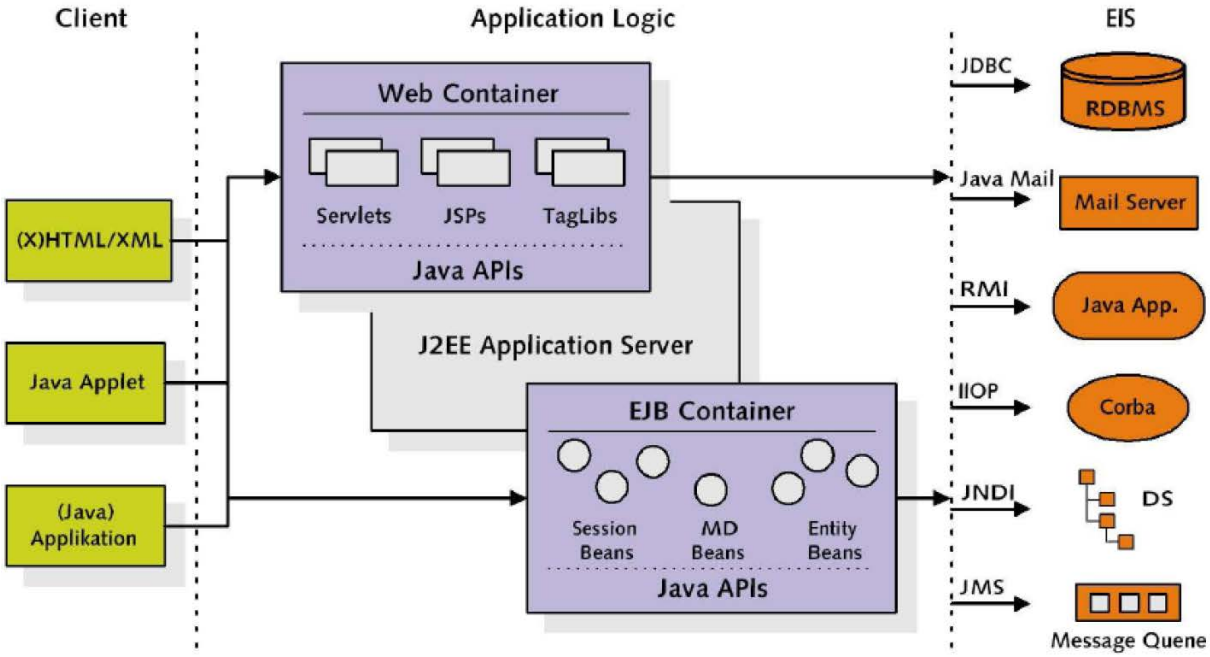



Abbildung: Architektur von JEE

## 4.2 Java Server Faces (JSF)

Java Server Faces ist eine Technologie, die im Grunde genommen die Controller-Funktionalität (Servlets) abnimmt. Der Benutzer muss sich nur um das Model und um die View kümern. Die folgende Grafik beschreibt das MVC-Prinzip bei JSF:

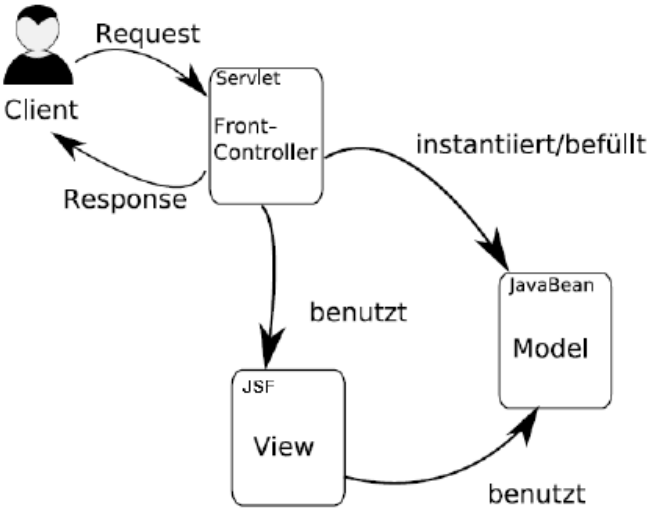



Abbildung: MVC-Prinzip bei JSF

**Model**

Die Model-Komponente bei JSF besteht aus den sogenannten ManagedBeans. Sie sind einfache POJOs (Plain Old Java Objects) und haben bestimmte Gültigkeitsbereiche. Sie enthalten die Geschäftslogik (Businesslogic) und können mittels EL (Expression Language) von der View-Komponente angesprochen werden.

Zur Deklaration gibt es die Möglichkeiten über XML (veraltet) oder über Annotation. Folglich ist ein Beispiel mit Annotations angeführt:

```
//Annotation to tell this Class is a ManagedBean with the name receivemai
@ManagedBean(name = "receivemail")
//Set the Bean to SessionScope (Serializable is needed)
@SessionScoped
public class ReceiveEmailBean implements Serializable {
   // Fields
   // Getter and Setter of ALL FIELDS
   // equals, hashcode, toString
}
```

Unterschiedliche Gültigkeitsbereiche (Scopes) für ManagedBeans:

| | |
|---|---|
| @javax.faces.bean.NoneScoped | In keiner Facelet Seite verfügbar |
| @javax.faces.bean.RequestScoped | Wird für jede Request neu instanziert |
| @javax.faces.bean.ApplicationScoped | Wird für jede Session neu instanziert (pro User) |
| @javax.faces.bean.ApplicationScoped | Ist für jeden User gleich und bleibt solange erhalten, wie das Programm läuft |

**View**

Die sogenannten Facelets stellen die View-Komponente dar. Bei Facelets werden mit XHTML Dateien Komponentenstrukturen definiert, die dann entsprechend übersetzt und als normales HTML dem Client angezeigt wird (Client bekommt nicht das XHTML Dokument zu sehen).

Facelets arbeiten mit der Expression Language (EL), um auf die Model-Komponenten zuzugreifen. EL-Statements erlauben direkte oder zeitversetzte Auswertung.

Direkte Auswertung:

```
${4>4.1}
```

Zeitversetzte Auswertung:

```
#{managedbeanname.attribut}
```

Wenn man in ein InputFeld beispielsweise den obigen Ausdruck schreibt, wird der Inhalt direkt in das Attribut gespeichert. Beim nächsten Aufruf der Seite steht der Wert des Attributs im InputFeld.

Beispiel für ein Facelet:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Book-Management</title>
        <h:outputStylesheet name="css/jsfcrud.css"/>
    </h:head>
    <h:body>
        <br />
        <h:outputText id="result" value="#{bookController.message}" />
    <h:body>
</html>
```

**Benutzerfunktionen**

Für die Navigation zu Seiten gibt es `commandButton` und `commandLink`, die jeweils ein Ziel als action haben. Von hier aus ist der Aufruf von Funktionen der ManegedBeans möglich, vorausgesetzt der Rückgabewert ist ein String. Beispiel:

```
public String receive() {
   String redirect = "";
   try {
      //BusinessLogic
      redirect = "/next.xhtml"
   } catch (Exception ex) {
      redirect = redirect + "/error.xhtml";
   }
   return redirect;
}
```

**Validierung**

Um das lästige Prüfen von Eingaben zu erleichtern, gibt es in JSF Validatoren (ist auch notwendig, da der Controller vom FacesServlet übernommen wurde). Man hat die Möglichkeit direkt im Facelet die Validatoren zu definieren, aber eine besser Variante wäre die Validatoren selbst zu schreiben, weil somit eine personalisierte Fehlermeldung ermöglicht wird. Die folgende Klasse ist ein Validator, der auf einen regulären Ausdruck prüft (Email):

```
@FacesValidator(value="emailvalidator")
public class EmailValidator implements Validator {
	static Pattern p = Pattern.compile(
	"\\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\\.[A-Z]{2,4}\\b");
	public EmailValidator() {}

	public void validate(FacesContext context, UIComponent component,
	Object object) throws ValidatorException {
		String enteredEmail = (String) object;

		//Match the given string with the pattern
		Matcher m = p.matcher(enteredEmail.toUpperCase());

		//Check whether the email matches
		if (!m.matches()) {
			//Create a FaceMessage, do give an ouput when wrong input
			FacesMessage message = new FacesMessage();
			message.setSummary("Field Sender not valid.");
			message.setSeverity(FacesMessage.SEVERITY_ERROR);
			throw new ValidatorException(message);
		}
	}
}
```

Um den Validator den entsprechenden Inputfeldern zuzuweisen, muss das im `validator` Attribut angegeben werden:

```
<h:panelGrid id="grid2" columns="2">
     <h:outputLabel value="Sender:" for="sender" />
     <h:inputText id="sender" value="#{sendmail.email.sender}" required="true"
     validator="emailvalidator"/>
</h:panelGrid>
```

## 4.3 Enterprise Java Beans (EJB)

Enterprise Java Beans sind zentrale Komponenten von Java EE, die die Businesslogik bieten und werden in EJB Container gestartet. [Sch15]

Die Enterprise Java Beans (EJB) Spezifikation definiert verteilte, serverseitige Komponenten zur serverseitigen Implementierung der Anwendungslogik. [Sch15]

Der EJB Container ist die Laufzeitumgebung der Enterprise Beans:

- Er kontrolliert den Life-Cycle der Enterprise Bean Instanzen: Erzeugeugung, Löschung, Aktivierung, Passivierung und Verteilung über Prozesse und Server
- Er bietet Dienste über standardisierte Schnittstellen: Transaktionsverwaltung, Persistenz und Security-Management können vom EJB-Container selbst kontrolliert werden.

[Sch15]

### 4.3.1 Architektur

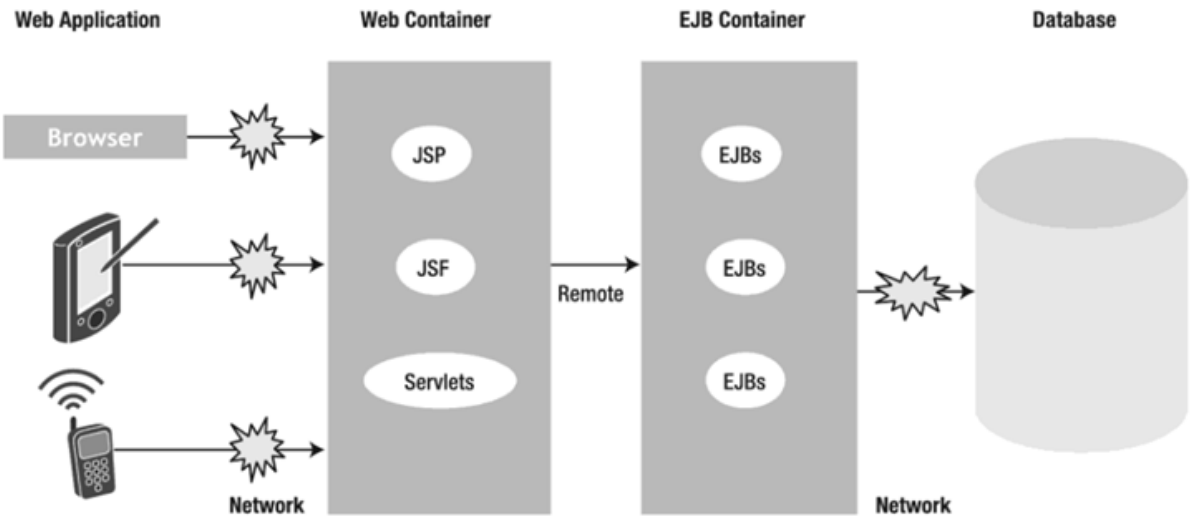



Abbildung: Architektur von EJB

### 4.3.2 Session Beans

- **Stateless SB**: `@Stateless`
  - Das Objekt wird bei jedem Methodenaufruf neu instanziert
  - Performant, leichter skalierbar
- **Statefull SB**: `@Stateful`
  - Das Objekt (Zustand) wird zwischen Methodenaufrufen gespeichert. 1:1
  - Beziehung zu Clients
  - Anwendung: z.B.: Warenkorb im E-Shop
- **Singleton SB**: `@Singleton`
  - Entsprechen dem gleichnamigen Pattern
  - Dem StatelessSB recht ähnlich, müssen ThreadSafe gemacht werden
  - Lebenszyklus: Anwendungsdauer

[Ora13]

### 4.3.3 Message-Drive Beans

Message-Drive Beans sind Stateless-SessionBeans, die asynchron per Nachrichtenversand erreichbar sind. Man verwendet sie beispielsweise für die asynchrone Kommuniikation (Emails verschicken). Sie wird aktiviert durch eigene JMS Nachricht. [Ora13]

Java Message Service:

- JMS Queue: Nachrichten werden in einer Queue gespeichert
- JMS Topic: Broadcast-Service, nach bestimmter Zeit gelöscht

[Ora13]

Da eine Message-Driven Bean von einem Client nicht direkt ansprechbar ist (asynchron), wird bei ihrer Implementierung auch kein Business-Interface (weder remote noch local) benötigt. Sie besteht nur aus der Bean-Implementierung. Sie muss mit der Annotation @MessageDriven versehen werden und ein Interface für den Nachrichtenmechanismus implementieren.

Beispiel für eine Message-Drive Bean:

```
@MessageDriven(name="asd", activationConfig={
      @ActivationConfigProperty(propertyName="destinationType",
                              propertyValue="javax.jms.Queue"),
      @ActivationConfigProperty(propertyName="destination",
                              propertyValue="queue/A"),
})
public class BestellannahmeImpl implements MessageListener{
      public void onMessage(Message nachricht){
            ...
      }
}
```

Bei der Annotation müssen einige Daten übergeben werden (in Form von key-value):

- destinationType: gibt an, ob die Bean über Queue oder Topic erreichbar ist.
- destination: JNDI-Name, über den diese Bean erreichbar ist.

[Ora13]

### 4.3.4 Dependency Injection

Die Dependency Injection überlässt dem EJB Container die Instanzierung von Klassen. Standardmäßig über `@Resource` Injection in der ManagedBean. [Ora13]

```
// Declaration of Businesslogic EJBs
@EJB(name = "AdressenPersistorBean")
AdressenPersistorLocal ap;
```

### 4.3.5 Java Persistence API (JPA)

**EntityBeans:**

EntityBeans repräsentieren POJOs (Plain Old Java Objects), die in der Datenbank persistiert werden können (so wie Hibernate). [Ora13] Beispiel für eine EntityBean:

```
@Entity
@Table(name = "buch")
public class Buch implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "id")
    private Integer id;
    @Size(max = 50)
    @Column(name = "name")
    private String name;
    …
}
```

**EntityManager:**

Der EntityManager verwaltet die Entities (CRUD Operationen auf der Datenbank). Beispiel für die Injection und Benutzung des EntityManagers:

```
@PersistenceContext
EntityManager em;

public void persist(Adresse adresse) {
      adresse.setId(counter);
      em.persist(adresse);
}

public void delete(Adresse adresse) {
      Adresse adr = em.find(Adresse.class, adresse.getId());
      em.remove(adr);
}
```

**persistence.xml:**

Die Einstellungen des EntityManagers (Datenbankname, etc.) werden in der persistence.xml Datei spezifiziert. Datei aus dem Übungsprojekt für die Matura:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  <persistence-unit name="FTWebApplicationPU" transaction-type="JTA">
    <jta-data-source>ftJNDI</jta-data-source>
    <exclude-unlisted-classes>false</exclude-unlisted-classes>
    <properties/>
  </persistence-unit>
</persistence>
```

### 4.3.6 Java Transaction API (JTA)

Bei Transaktionen wird zwischen Container Managed und Bean Managed Transaktionen unterschieden. Hierbei werden beide folgendermaßen über die Klasse deklariert:

```
@TransactionManagement(TransactionManagementType.CONTAINER/BEAN)
```

#### *4.3.6.1 Container Managed Transactions*

Die Gültigkeit (Scope) der Container Managed Transactions wird mithilfe von `@TransactionAttribute` angegeben. Rollbacks passieren automatisch, wenn RuntimeExceptions auftreten. Folgende Transaktionsattribute stehen zur Verfügung: [Ora13]

| | |
|---|---|
| `@NotSupported` | Transaktion wird bis zur Beendigung der Methode unterbrochen |
| `@Supports` | Falls bereits eineTransaktion für den aktuellen Client-Request vorhanden ist, wird diese verwendet |
| `@Required (default)` | Ein Sonderfall von "Supports": Falls noch keine Transaktion vorhanden ist, wird eine neue gestartet. |
| `@RequiresNew` | Es wird immer eine neue Transaktion gestartet. |
| `@Mandatory` | Eine Client-Transaktion ist Voraussetzung für die Verwendung der EJB, ansonsten wird eine entsprechende Exception geworfen |
| `@Never` | Die EJB darf nicht in einem Transaktionsscope verwendet werden ansonsten wird eine entsprechende Exception geworfen |

Im nachfolgenden Beispiel wird eine Container Managed Transaction veranschaulicht, wobei über den SessionContext auf die Transaktion zugegriffen wird:

```
@TransactionManagement(TransactionManagementType.CONTAINER)
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
@Stateless
public class ShopImpl implements Shop{
    @Resource
    SessionContext sc;
    public method(){
        try{
        //trysomething
        } catch(NotRuntimeException nre){
            sc.setRollbackOnly();
            throw nre;
        }
    }
}
```

#### *4.3.6.2 Bean Managed Transactions*

Die Bean Managed Transactions müssen explizit durch den Programmierer behandelt werden, wie im folgenden Beispiel ersichtlich:

```
@TransactionManagement(TransactionManagementType.BEAN)
@Stateless
public class ShopImpl implements Shop{
    @Resource
    SessionContext sc;

    public method(){
        sc.getUserTransaction().begin();
        //dosomething
        sc.getUserTransaction().commit();
        ...
    }
}
```

### 4.3.7 Timer

Timer-Services müssen in einem EJB stattfinden! Hierbei wird zwischen automatic und programmatic Timern unterschieden, wobei automatic Timer für kalendarische Ereignisse gut geeignet sind und programmatic Timer besser für Zeitaufzeichnungen (ab einem gewissen Event mitzählen) geeignet sind.

#### *4.3.7.1 Automatischer Timer*

Beispiele für einen automatischen Timer:

```
@Schedule(dayOfWeek = "Sun", hour = "2")
public void doService() {
    System.out.println("ServiceMode ON");
    serviceMode = true;
}
```

```
@Timeout
// Mark this method as the EJB timeout method for timers created
// programmatically.
// If we're just ceating a timer via @Schedule, @Timer is not required.
@Schedule(dayOfMonth = EVERY, month = EVERY, year = EVERY, second = ZERO, minute =
ZERO, hour = EVERY)
// This tmeout will be created on deployment and fire every hour on the hour;
declarative creation
public void processViaTimeout(final Timer timer) {
      // Just delegate to the business method
      this.process();
}
```

#### *4.3.7.2 Programmatischer Timer*

Hierfür muss in der EJB Session Bean ein TimerService als Attribut deklariert werden, welches durch die Annotation @Resource initialisiert wird. Durch die folgenden Methoden wird ein Timer gesetzt und die timeout-Methode implementiert das erwünschte Verhalten, wenn der Timer abläuft. Zu Beachten ist, dass beim setTimer die Zeitzone als „GMT-2“ angegeben wird, da das Service nicht mit der österreichischen Zeit rechnet, und daher immer 2 Stunden hinzufügt. Außerdem ist zu beachten, dass bei der ts.createTimer()-Methode abgesehen von der Zeit auch ein Serializable-Objekt übergeben werden kann. Somit kann genau auf dieses Objekt beim Timeout durch timer.getInfo() zugegriffen werden, um es in diesem Fall beispielsweise zu löschen:

```
@Resource
private TimerService ts;

public void setTimer(Sonderangebot sbg) {
      // Aufgaben SPEZIFISCH
    Calendar start = new GregorianCalendar();
    start.setTime(sbg.getStartZeit());
      // Umrechnen auf ms unter der Voraussetzung dass die Zeit in Minuten
      // eingegeben wird
    long sum = sbg.getDauer()*60*1000+start.getTimeInMillis();

      // Allgemeiner Teil
    Calendar cal = new GregorianCalendar(TimeZone.getTimeZone("GMT-2"));
    cal.setTimeInMillis(sum);
    ts.createTimer(cal.getTime(), sbg);

    String outp = new SimpleDateFormat().format(cal.getTime());
    System.out.println("Timer started and will expire in " + outp);
}

@Timeout
public void timeout(Timer timer) {
    System.out.println("Timer expired!");
      // Erwuenschtes Verhalten bei Timeout
    this.remove((Sonderangebot) timer.getInfo());
}
```

### 4.3.8 Security-Management

Beispiel für rollenbasierte Zugriffsrechte:

```
@Singleton
@Local(SecureSchoolLocalBusiness.class)
@DeclareRoles(
{Roles.ADMIN, Roles.STUDENT, Roles.JANITOR})
@RolesAllowed({})
@Startup
public class SecureSchoolBean implements SecureSchoolLocalBusiness {
       @PermitAll
       public boolean isOpen() { ... }

       @RolesAllowed(Roles.ADMIN)
       // Only let admins open and close the school
       public void close() { ... }

       @PostConstruct
       // School is open when created
       @RolesAllowed(Roles.ADMIN)
       // Only let admins open and close the school
       public void open() { ... }

       ...
}
```

#### *4.3.8.1 Umsetzung von AAA (Authentication & Authorization) mittels des Frameworks „Apache Shiro“ [Shi13]*

Nachdem die benötigten Dependencies für Apache Shiro heruntergeladen wurden mittels Maven, kann Apache Shiro im Projekt genutzt werden. Hierzu muss in /WEB-INF/web.xml eine Änderung vorgenommen werden und in diesem Verzeichnis (/WEB-INF/) die Datei shiro.ini erstellt werden.

Folgendes muss in /WEB-INF/web.xml:

```
    <listener>
        <listener-
class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class>
    </listener>

    <filter>
        <filter-name>shiroFilter</filter-name>
        <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>shiroFilter</filter-name>
        <url-pattern>/*</url-pattern>
        <dispatcher>REQUEST</dispatcher>
        <dispatcher>FORWARD</dispatcher>
        <dispatcher>INCLUDE</dispatcher>
        <dispatcher>ERROR</dispatcher>
    </filter-mapping>
```

Example für shiro.ini:

```
# =============================================================================
# Tutorial INI configuration
#
# Usernames/passwords are based on the classic Mel Brooks' film "Spaceballs" :)
# =============================================================================

# -----------------------------------------------------------------------------
# Users and their (optional) assigned roles
# username = password, role1, role2, ..., roleN
# -----------------------------------------------------------------------------
[users]
root = secret, admin
guest = guest, guest
presidentskroob = 12345, president
darkhelmet = ludicrousspeed, darklord, schwartz
lonestarr = vespa, goodguy, schwartz

# -----------------------------------------------------------------------------
# Roles with assigned permissions
# roleName = perm1, perm2, ..., permN
# -----------------------------------------------------------------------------
[roles]
admin = *
schwartz = lightsaber:*
goodguy = winnebago:drive:eagle5
```

Login-Java-Klasse:

```
@Named
@RequestScoped
public class Login {

    public static final String HOME_URL = "app/index.xhtml";

    private String username;
    private String password;
    private boolean remember;

    public void submit() throws IOException {
        try {
            SecurityUtils.getSubject().login(new UsernamePasswordToken(username,
password, remember));
            SavedRequest savedRequest =
WebUtils.getAndClearSavedRequest(Faces.getRequest());
            Faces.redirect(savedRequest != null ? savedRequest.getRequestUrl() :
HOME_URL);
        }
        catch (AuthenticationException e) {
            Messages.addGlobalError("Unknown user, please try again");
            e.printStackTrace(); // TODO: logger.
        }
    }

    // Add/generate getters+setters.
}
```

Login.xhtml:

```
<h2>Login</h2>
    <h:form id="login">
        <h:panelGrid columns="3">
            <h:outputLabel for="username" value="Username:" />
            <h:inputText id="username" value="#{login.username}" required="true">
                <f:ajax event="blur" render="m_username" />
            </h:inputText>
            <h:message id="m_username" for="username" />

            <h:outputLabel for="password" value="Password:" />
            <h:inputSecret id="password" value="#{login.password}" required="true">
                <f:ajax event="blur" render="m_password" />
            </h:inputSecret>
            <h:message id="m_password" for="password" />

            <h:outputLabel for="rememberMe" value="Remember Me:" />
            <h:selectBooleanCheckbox id="rememberMe" value="#{login.remember}" />
            <h:panelGroup />

            <h:panelGroup />
            <h:commandButton value="Login" action="#{login.submit}" >
                <f:ajax execute="@form" render="@form" />
            </h:commandButton>
            <h:messages globalOnly="true" layout="table" />
        </h:panelGrid>
    </h:form>
```

shiro.ini:

```
[main]
user.loginUrl = /login.xhtml

[users]
admin = password

[urls]
/login.xhtml = user
/app/** = user
```

Logout-Java-Klasse:

```
@Named
@RequestScoped
public class Logout {

    public static final String HOME_URL = "login.xhtml";

    public void submit() throws IOException {
        SecurityUtils.getSubject().logout();
        Faces.invalidateSession();
        Faces.redirect(HOME_URL);
    }

}
```

Logout im xhtml-File (wo sich der Logout-Button befindet):

```
<h:form>
       <h:commandButton value="logout" action="#{logout.submit}" />
</h:form>
```

## 4.4 Vorgehensweise bei der Erstellung einer Java EE Webapplikation unter NetBeans 8.0.2

Um in NetBeans eine WebApplikation zum Laufen zu kriegen, sind folgende Schritte notwendig:

### 4.4.1 Einrichten der Datenbank für die Datenpersistierung

Um die Daten der Webapplikation persistieren zu können, ist die Verwendung einer Datenbank wie beispielsweise MySQL, PostgreSQL, etc. notwendig. Wir haben uns entschieden eine PostgreSQL-Datenbank zu benutzen.

Anlegen eines Benutzers in der PostgreSQL-Datenbank und zuweisen der Rechte:

```
$ sudo su postgres
$ psql
```

Wenn man nun in der Postgres-Datenbank ist, folgende Befehle eingeben:

```
postgres=# create user schueler with password ‘schueler‘;
postgres=# alter user schueler with superuser;
```

Nun erstellen wir eine Datenbank in PostgreSQL, um die Daten der Webapplikation darin persistieren zu können (Nachdem ein User angelegt wurde, die DB schließen mit Strg+D oder \q um sich als schueler anzumelden):

```
$ psql –d template1 –U schueler –w
```

Nach der Eingabe dieses Befehls, sollte man als schueler in der Datenbank angemeldet sein. Nun sind wir bereit eine Datenbank zu erstellen:

```
postgres=# create database ft;
postgres=# \c ft;
```

Mit \c haben wir uns zu der erstellten Datenbank verbunden. Jetzt können wir das DDL-Script in die Datenbank kopieren, um die DB-Tabellen zu erstellen. Hiermit sollte Datenbank für die Nutzung eingerichtet werden.

Falls die Datenbank gestoppt und neugestartet werden muss, sind folgende Befehle auszuführen im Terminal:

```
$ sudo /etc/init.d/postgresql stop
$ sudo /etc/init.d/postgresql start
```

### 4.4.2 Einrichten des GlassFish-Servers

Der GlassFish-Server ist in NetBeans default-mäßig enthalten als Server, aber sie muss am System ebenfalls heruntergeladen werden, damit der heruntergeladene GlassFish-Ordner im Installation-Folder unter den GlassFish-Properties angegeben werden kann.

Um den „Services"-Fenster zu öffnen, muss man „Strg + 5" drücken. Dann hat man eine Auflistung der Services, wie zum Beispiel Datenbanken, Server, etc.

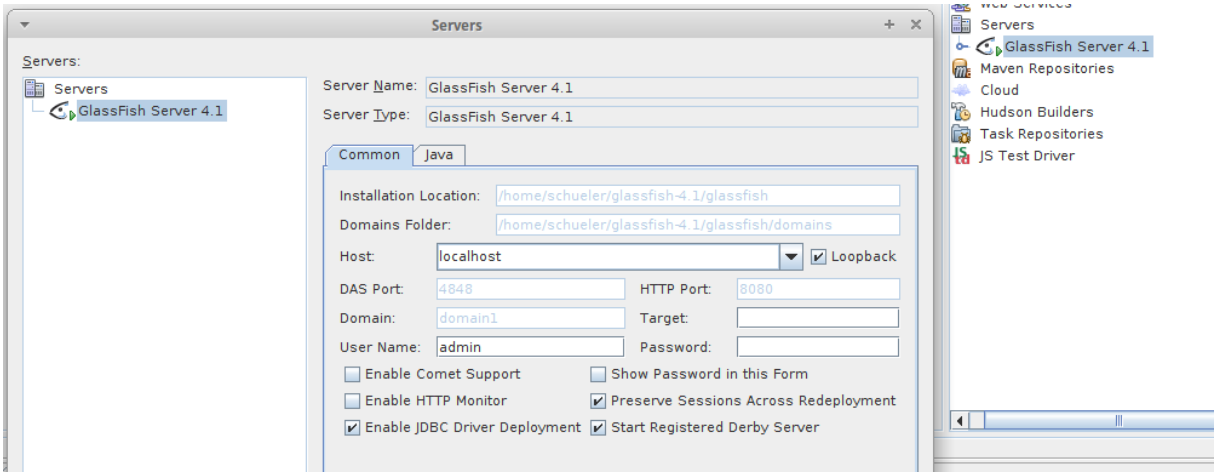



Abbildung: Einstellungen des GlassFish-Servers (im Installation-Location muss der Pfad vom heruntergeladenen GlassFish-Ordner angegeben werden)

Wenn der GlassFish-Server in NetBeans wie oben konfiguriert ist, kann sie (wenn man noch keine Webapplikation starten will, sondern nur GlassFish) durch Rechtsklick und „Start" gestartet werden bzw. mit „Stop" gestoppt werden. Wenn man GlssFish über die Konsole stoppen und starten möchte, muss man folgende Befehle verwenden:

```
$ cd /home/schueler/glassfish-4.1/bin/
$ sudo ./asadmin stop-domain1
$ sudo ./asadmin start-domain1
```

Wenn der GlassFish-Server gestartet ist, dann sollte sie immer unter der URL „localhost:4848" erreichbar sein:

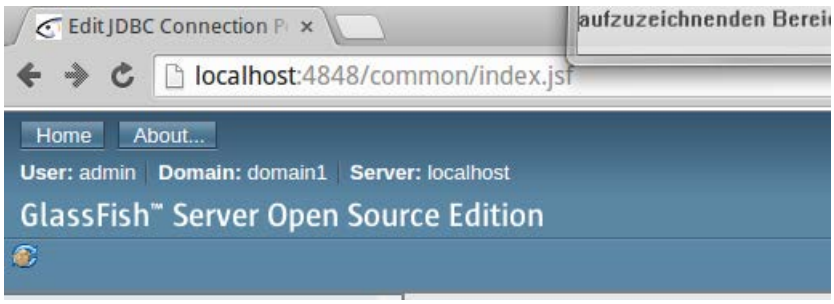



Abbildung: Abrufen des GlassFish-Servers im Browser

Somit ist der GlassFish-Server eingerichtet und kann von der Webapplikation verwendet werden.

### 4.4.3 Einrichten der Verbindung zwischen dem GlassFish-Server und der Datenbank

Bis jetzt wurde eine PostgreSQL-Datenbank und der GlassFish-Server eingerichtet. Da die Webapplikation über den GlassFish-Server die Datenpersistierung durchführt, muss eine Verbindung vom GlassFish-Server zur PostgreSQL-Datenbank umgesetzt werden.

Hierzu erstellt man unter dem Punkt „JDBC Connection Pools“ eine neue Connection Pool, indem man auf „New“ klickt und folgende Einstellungen durchführt:

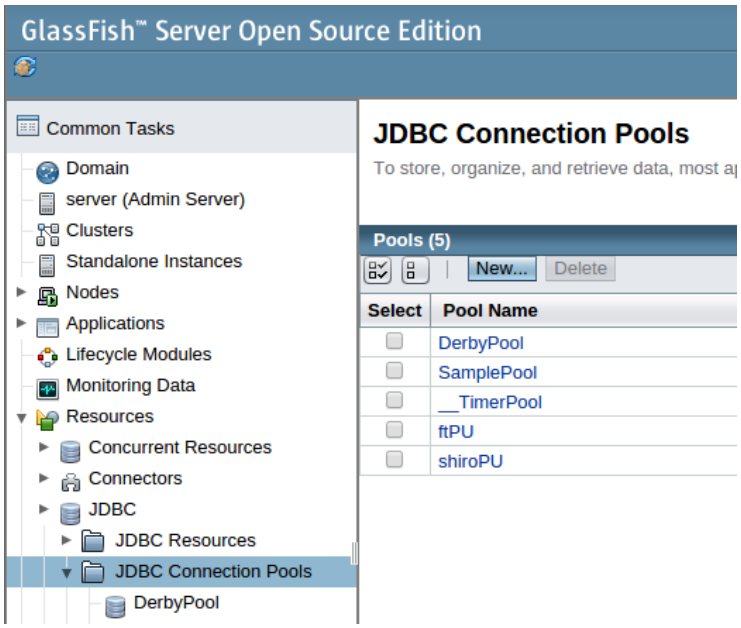



Abbildung: Ansicht JDBC Connection Pool

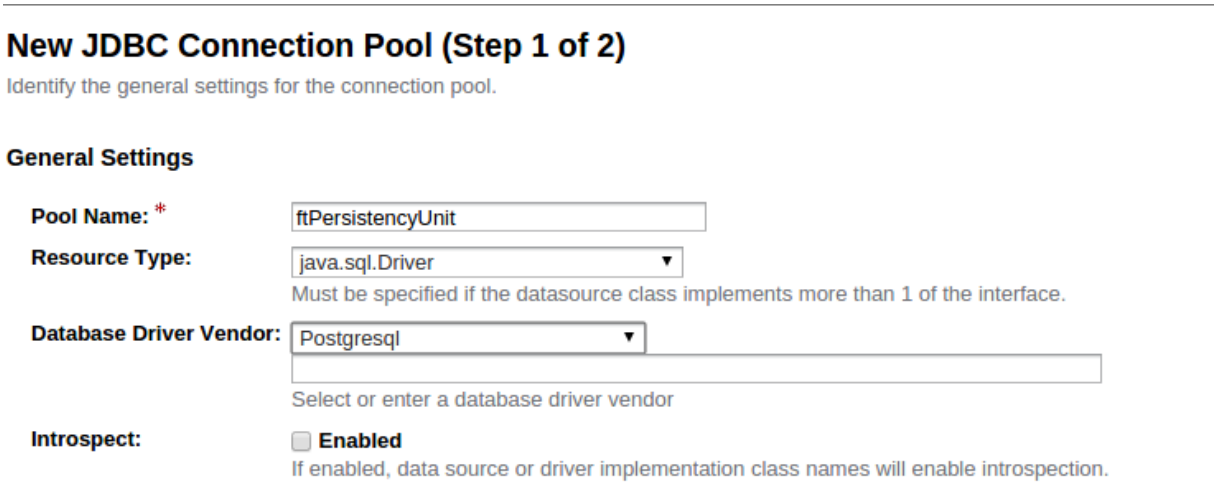



Abbildung: Grund-Einstellungen durchführen

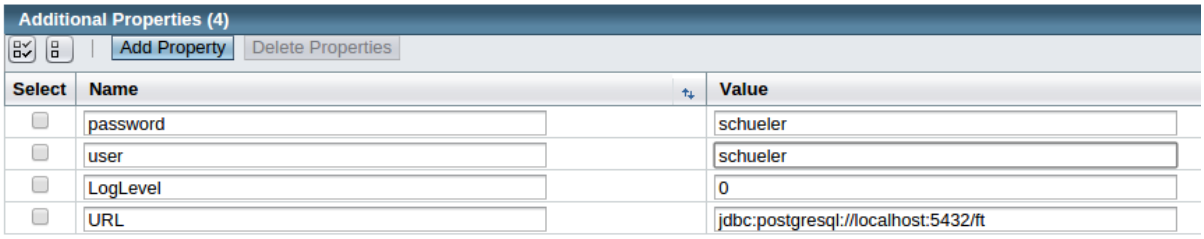



Abbildung: Advanced-Einstellungen durchführen (nach Klick auf weiter)

Nun wurde unsere Connection-Pool angelegt:

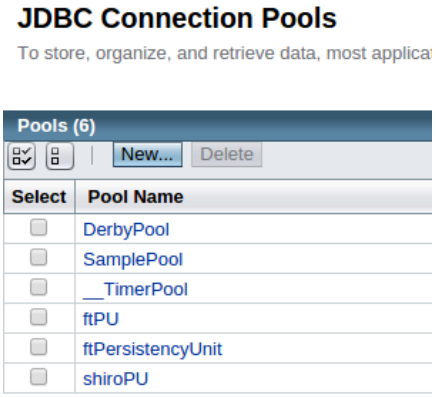



Abbildung: Connection-Pool „ftPersistencyUnit“ angelegt

Nun muss eine neue JDBC Resource angelegt werden:

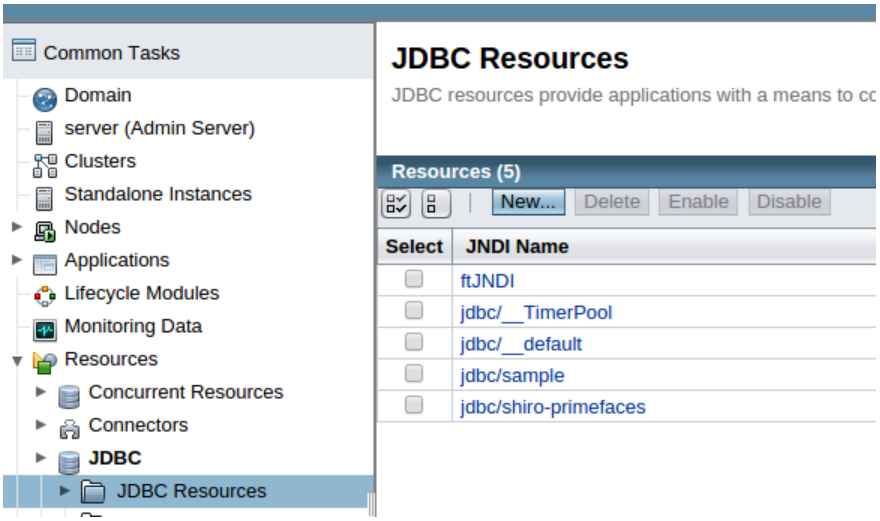



Abbildung: JDBC Resources

Durch Klicken auf „New“ erscheint eine neue Seite, in der wir folgende Einstellungen vornehmen müssen:


Abbildung: Erstellung einer neuen JDBC Resource

Das Pingen der PostgreSQL-Datenbank sollte nun über dem GlassFish-Server möglich sein:

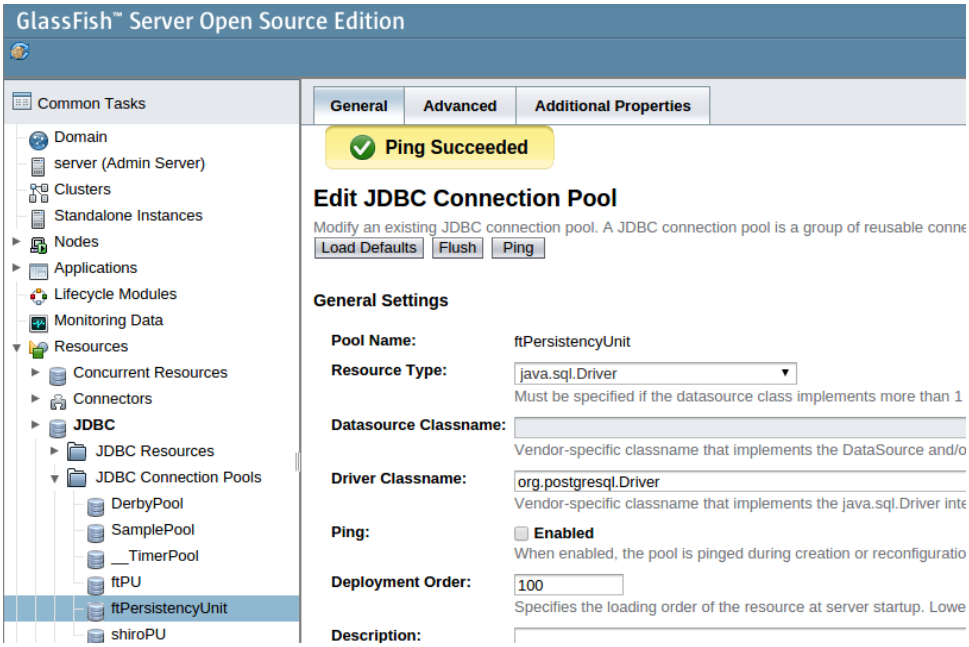



Abbildung: Pingen der Datenbank erfolgreich

Falls das Pingen nicht funktioniert, muss der JDBC-Treiber (das JAR-File also) in den lib-Ordner von GlassFish kopiert werden.

In der VM (für die Matura) sollten die JAR-Files unter „`/home/.m2/repositories`“ sein, die dann in den GlassFish-lib-Ordner kopiert werden können.

Das Schema der Datenbank muss auf „public“ gesetzt werden:

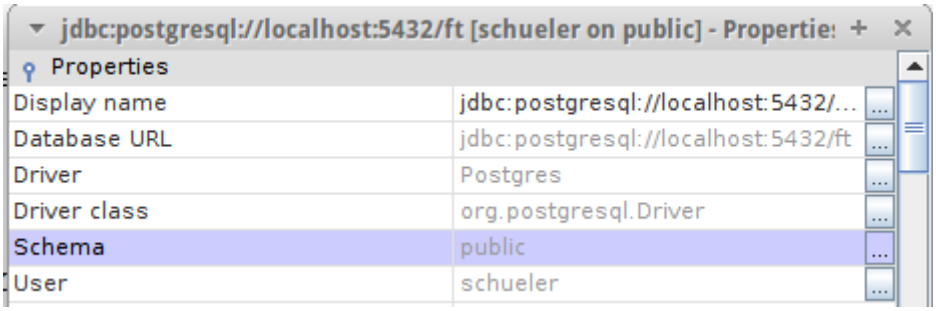
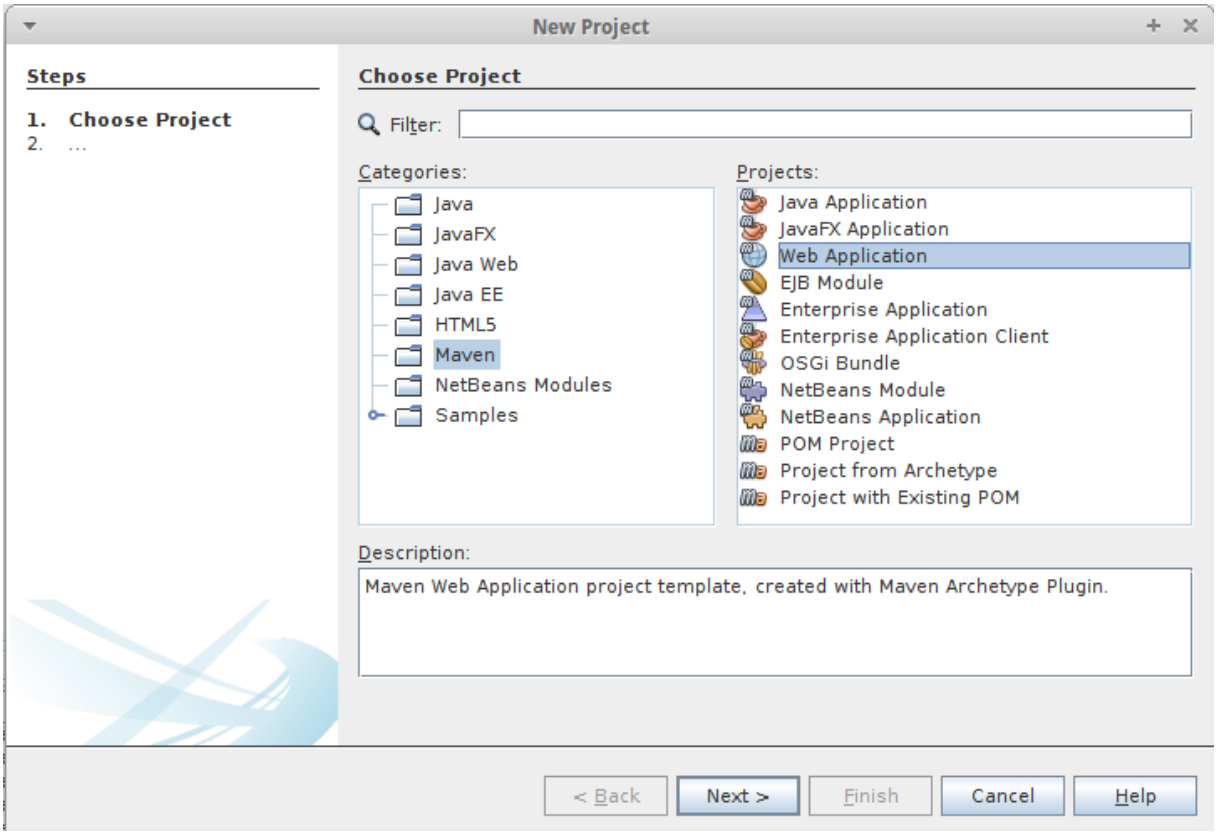



Abbildung: Einstellungen der Datenbank

### 4.4.4 Erstellen eines Webapplikation-Projektes in NetBeans

Wenn man eine neue Webapplikation erstellen will, muss man in NetBeans unter „File > New Project“ klicken in der MenuBar und folgendes Auswählen:




Abbiildung: Erstellung einer Maven Web-Applikation

Das Projekt sollte nach der Erstellung links im „Projects“-Tab erscheinen.

### 4.4.5 Laden der Dependencies mittels des „Maven“-Build-Tools

Damit alle benötigten Abhängigkeiten für die Webapplikation geladen werden können, muss man die Dependencies im „pom.xml“-File ergänzen auf die benötigten Abhängigkeiten und das Projekt dann mit „Build with Dependencies“ ausführen (durch Rechtsklick auf das Projekt).

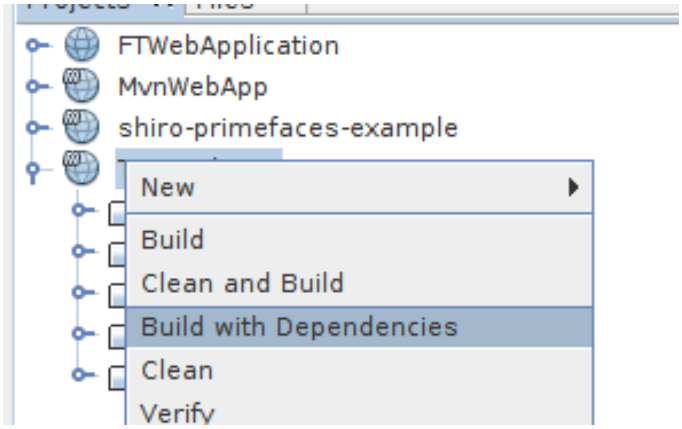



Abbildung: Ausführen der Webapplikation mit den Maven-Dependencies

Folgendes muss im erstellten „pom.xml“-File ergänzt werden:

```
    <properties>
        …
    </properties>

    <dependencies>
        <dependency>
            <groupId>xerces</groupId>
            <artifactId>xercesImpl</artifactId>
            <version>2.11.0</version>
        </dependency>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>2.45.0</version>
        </dependency>
        <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-server</artifactId>
            <version>2.45.0</version>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-all</artifactId>
            <version>1.9.5</version>
        </dependency>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>eclipselink</artifactId>
            <version>2.5.2</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.eclipse.persistence</groupId>
            <artifactId>org.eclipse.persistence.jpa.modelgen.processor</artifactId>
            <version>2.5.2</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.primefaces</groupId>
            <artifactId>primefaces</artifactId>
            <version>5.2</version>
        </dependency>
        <dependency>
            <groupId>org.primefaces.extensions</groupId>
            <artifactId>primefaces-extensions</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
            <type>jar</type>
        </dependency>
        <dependency>
            <groupId>javax</groupId>
            <artifactId>javaee-web-api</artifactId>
            <version>7.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.shiro</groupId>
            <artifactId>shiro-core</artifactId>
            <version>1.2.3</version>
        </dependency>
        <dependency>
            <groupId>org.apache.shiro</groupId>
            <artifactId>shiro-web</artifactId>
            <version>1.2.3</version>
        </dependency>
    </dependencies>

    <build>
        …
    </build>

    <repositories>
        <repository>
            <url>http://repository.primefaces.org/</url>
            <id>PrimeFaces-maven-lib</id>
            <layout>default</layout>
            <name>Repository for library PrimeFaces-maven-lib</name>
        </repository>
    </repositories>
```

### 4.4.6 Erzeugen der Entity-Klassen aus der Datenbank

Das Erzeugen von EntityKlassen geht in NetBEans relativ einfach. Hierzu muss man bei den Source-Packages des Projektes auf „New > Entity Classe from Database“ klicken:

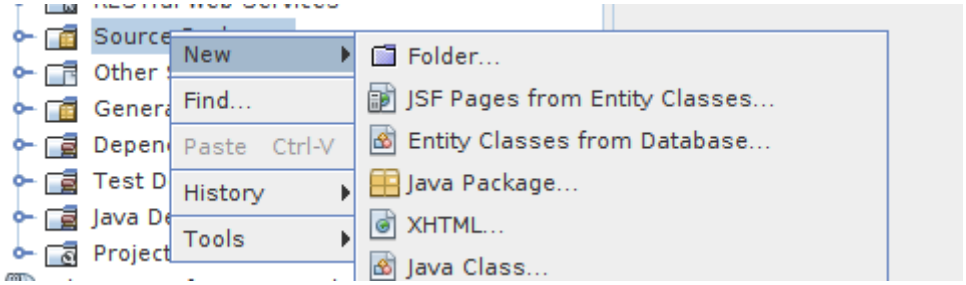



Abbildung: Erstellung der EntityKlassen aus der Datenbank

Anschließend wählt man die Data Source (JDBC Datasource, die am GlassFish-Server erstellt wurde) aus. Durch die Auswahl der datasource werden die Datenbanktabellen aufgelistet, aus denen die EntityKlassen generiert werden können.

### 4.4.7 Erzeugen der JSF (bzw. PrimeFaces)-Pages aus den Entity-Klassen

Wenn man nun die EntityKlassen erstellt hat, kann man aus diesen Klassen, die JSF Pages erstellen (Hiermit werden CRUD-Operationen in der GUI generiert).

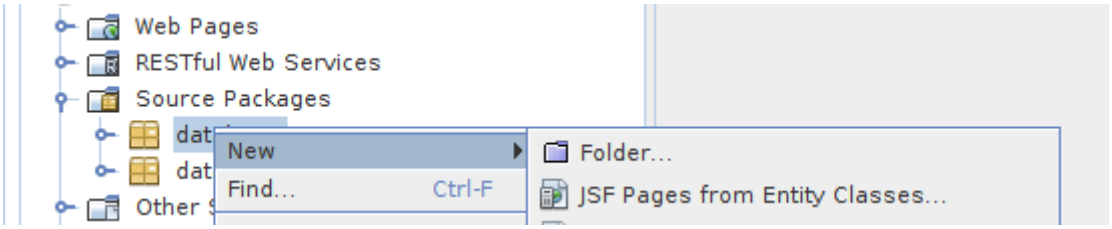



Abbildung: Erstellung der JSF Pages aus den EntityKlassen

Bei der Generierung der Pages, dürfen keine Pfade zusätzlich angegeben werden, sondern die Standard-Einstellungen lassen. Bei den Templates muss aber noch „PrimeFaces“ ausgewählt werden und nicht „Standard Java Server Faces“. Nach der Generierung der Pages muss das „index.html“-File gelöscht werden, da in diesem Projekt „index.**x**html“ verwendet wird.

### 4.4.8 Testen der Webapplikation

Das Testen der Webapplikation erfolgt ganz einfach. Folglich wird das Testen mit JUnit veranschaulicht.

#### *4.4.8.1 Unit-Testing – EJB Container*

Wenn man Unit-Tests für jede Klasse separat macht, d.h. für jede Klasse eine eigene JUnit-Testklasse erstellt, dann kann man diese Testklassen über TestSuites gleichzeitig ausführen.

Beispiel für eine TestSuite:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({database.unit_tests.AutorCRUDTests.class,
database.unit_tests.BuchCRUDTests.class, database.unit_tests.VerlagCRUDTests.class,
database.unit_tests.AllCRUDTests.class})
public class CRUDTestSuite {
    public CRUDTestSuite() {
    }

    @BeforeClass
    public static void setUpClass() {

    }

    @AfterClass
    public static void tearDownClass() {

    }

    @Before
    public void setUp() {

    }

    @After
    public void tearDown() {

    }
}
```

Man gibt bei den SuiteClasses die Namen der auszuführenden Testklassen an (inklusive der Packagestruktur).

In einer JUnit-Testklasse muss folgendes enthalten sein, damit die Tests richtig implementiert werden können:

Diese Attribute müssen in der Testklasse deklariert werden:

```
private static EJBContainer container; // Referenz auf den Embedded
EJBContainer
private static AutorFacade proxy; // Proxy-Referenz der SessionBean
```

Bei BeforeClass-Methode muss folgendes geschrieben werden:

```
@BeforeClass
public static void setUpClass() {
        System.out.println("Opening the container");

        Map<String, Object> properties = new HashMap<String, Object>();
        properties.put(EJBContainer.MODULES, new File("build/jar"));
        container = EJBContainer.createEJBContainer(properties); //
EJBContainer erstellen

        Context ctx = container.getContext(); // Context erstellen

        try {
            proxy = (AutorFacade)
ctx.lookup("java:global/classes/AutorFacade"); // Proxy-Referenz mittels
JNDI
        } catch (NamingException ex) {

Logger.getLogger(AutorCRUDTests.class.getName()).log(Level.SEVERE, null,
ex);
        }
}
```

Bei AfterClass-Methode muss folgendes geschrieben werden:

```
@AfterClass
public static void tearDownClass() {
        System.out.println("Closing the container");

        container.close();
}
```

Beispiel für einen Unit-Test:

```
@Test
public void testCreateAutor() {
        // Erstellen eines Random-IDs fuer den Autor
        Random random = new Random();
        this.id = random.nextInt(1000);

        char[] chars = "abcdefghijklmnopqrstuvwxyz".toCharArray();

        // Erzeugen eines Random-Strings fuer den Vornamen des Autors
        StringBuilder sb = new StringBuilder();
        random = new Random();
        for (int j = 0; j < 10; j++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }
        this.vname = sb.toString();

        // Erzeugen eines Random-Strings fuer den Nachnamen des Autors
        sb = new StringBuilder();
        random = new Random();
        for (int j = 0; j < 10; j++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }
        this.nname = sb.toString();

        // Erzeugen eines Random-Strings fuer die Email-Adresse des Autors
        random = new Random();
        for (int j = 0; j < 10; j++) {
            char c = chars[random.nextInt(chars.length)];
            sb.append(c);
        }
        this.email = sb.toString() + "@hotmail.com";

        Autor autor = new Autor();

        autor.setId(this.id);
        autor.setVname(this.vname);
        autor.setNname(this.nname);
        autor.setEmail(this.email);
        autor.setSvnr(this.svnr);

        int countBeforeCreation = proxy.count();

        try {
            proxy.create(autor); // Persistieren des Autors
        } catch (Exception ex) {

Logger.getLogger(AutorCRUDTests.class.getName()).log(Level.SEVERE,    null,
ex);
        }

        int countAfterCreation = proxy.count();

        assertEquals(countBeforeCreation+1,     countAfterCreation);     //
Testen, ob der Datensatz wirklich in der DB erstellt wurde
}
```

### 4.4.9 Implementieren einer Suchfunktion

Definieren einer NamedQuery bei der EntityBean der jeweiligen Klasse:

```
@Entity
@NamedQueries({
    @NamedQuery(name = "findByBhf",
    query = "SELECT e FROM Zug e WHERE e.ende.name = :zbhf AND
e.start.name=:sbhf")
})
public class Zug implements Serializable {}
```

Die NamedQuery muss nun von einer Façade bzw. Session Bean aufgerufen werden und konkrete Parameter erhalten:

```
@Override
public List<Zug> findWithKeyword(String sbhf, String zbhf) {
    return em.createNamedQuery("findByBhf").
            setParameter("zbhf", zbhf).
            setParameter("sbhf", sbhf).
            getResultList();
}
```

Nun muss die Methode der Facade bzw. Session Bean von der ManagedBean (Controller) aufgerufen werden. Diese Methode sollte die vorhandene List.xhtml verwenden, indem die Rückgabeliste als DataModel gespeichert wird und die Methode „List“ zurückgibt, damit zu dieser Seite navigiert wird. Die gebrauchten Attribute für evtl. Textfelder sollten im Controller deklariert werden.

```
private String sbhf;
private String zbhf;
public String findZug() {
    this.items = new ListDataModel(ejbFacade.findWithKeyword(sbhf, zbhf));
    return "List";
}
```

Die Methode der ManagedBean (also vom Controller) kann dann in der JSF-Datei (.xhtml) durch Klicken eines Buttons beispielsweise aufgerufen:

```
<h:outputText value="Abfahrtsort"/>
<h:inputText value="#{zugController.sbhf}" />
<h:outputText value="Ankunfts"/>
<h:inputText value="#{zugController.zbhf}" />
<br></br>
<h:commandButton action="#{zugController.findZug()}" value="Search"/>
```

# Quellenverzeichnis


[Mar14] Marcel Paggen, Klassendiagramm (class diagram), URL: http://timpt.de/topic95.html, zuletzt geändert am 08.06.2014 (besucht am 28.04.2015)

[Uml15] Unified Modelling Language, UML-Tutorial, URL: http://www-ivs.cs.uni-magdeburg.de/~dumke/UML/12.htm (besucht am: 29.04.2015)

[Ber15] Bernhard Lahres; Gregor Rayman, Objektorientierte Programmierung – Vererbung und Polymorphie, URL: http://openbook.rheinwerk-verlag.de/oop/oop_kapitel_05_001.htm (besucht am 29.04.2015)

[Eri04] Eric Freeman; Elisabeth Freeman, Head First: Design Patterns, 2004, Paperback, 611 Seiten

[Tut15] Tutorialspoint, Design Patterns Tutorial, URL: http://www.tutorialspoint.com/design_pattern (besucht am 29.04.2015)

[Sch15] Dr. Bernhard Schiefer, Enterprise JavaBeans (EJB), URL: http://www.fh-kl.de/~schiefer/lectures/download/javaee/javaee_01_intro.pdf (besucht am 15.05.2015)

[Ora13] Oracle, The Java EE 6 Tutorial, URL: http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html (besucht am 15.05.2015)

[Shi13] The BalusC Code, Apache Shiro, is it ready for Java EE 6? (a JSF2-Shiro Tutorial), URL: http://balusc.blogspot.co.at/2013/01/apache-shiro-is-it-ready-for-java-ee-6.html (besucht am 15.05.2015)