Erste Schritte mit Maven

Hier werden euch ein paar Aufgaben gestellt, die mit Vorkenntnissen binnen Minuten zu erledigen sind. Sie sollen das Ausprobieren und Herumspielen fördern. Hast du eine Idee, dann probiere sie ruhig aus! Die kursiv gesetzten Fragen sollen dich zum Denken anregen. Du brauchst die Antworten nicht niederschreiben, aber denke drüber nach. Sie werden das Verständnis verbessern. Bei Fragen meldet euch einfach bei uns.

Kickstart

Für die folgenden Aufgaben müsst ihr euch auf die Kommandozeile begeben. Falls ihr in eurem Home-Directory noch kein Verzeichnis für Entwicklungsprojekte habt, erstellt euch eines.

Begebt euch in euer persönliches Entwicklungsverzeichnis, bspw.:

```
$ cd ~/development
```

Prüft, ob die Voraussetzungen für Maven auf eurem System gegeben sind

```
$ java -version

$ mvn -v
```

Setzt ein eigenes Projekt mit Maven und dem Archetype-Plugin auf

```
$ mvn archetype:generate
```

Auswahl:
Archetype Nummer 15 (maven-archetype-quickstart)
GroupId: de.fhkoeln.ass
ArtifactID: ff
Version: 1.0-SNAPSHOT
Restliche Einstellungen: default verwenden.

Wechselt in das Verzeichnis und schauen wir uns die Inhalte an:

```
$ cd ff
$ ls -lR
```

Analysiert das Projekt:
Wo liegt der Source-Code?
Was für Sourcen oder Arten von Sourcen gibt es?

Was müsste beim ausführen der Sourcen geschehen?

Kompiliert das Projekt:

```
$ mvn compile
```

Was wird alles beim Kompilieren gebaut / generiert?
Wo ist das Kompilat? Was ist das target Verzeichnis? Braucht ihr noch ein bin Verzeichnis?

Führt die Java-Klasse auf der Kommandozeile aus.

```
$ cd target/classes
$ java de.fhkoeln.ass.App
```

Hat sich deine Erwartung von oben bestätigt?

Führt den Test aus

```
$ mvn test
```

Erstellt ein Eclipse Projekt mit Maven

```
$ mvn eclipse:eclipse
```

Importiert das Projekt in Eclipse
Startet Eclipse!
Falls Eclipse beim Start fragt, wo der Workspace liegen soll, gebt euer Entwicklungs-Verzeichnis an. Falls Eclipse nicht fragt, ändert dies von Hand: File -> Switch Workspace -> Others...

Dann:
File -> Import -> Existing Project into Workspace

Fügt eine eigene Klasse hinzu (Fast.java)
Fast.java:

```
package de.fhkoeln.ass;

public class Fast {


	public String getFast() {
		return "Fast!!";
	}

	public String getFurious() {
		return "Furious!";
	}
}
```

Fügt eine Testklasse hinzu (FastTest.java)
FastTest.java:

```
package de.fhkoeln.ass;

import junit.framework.TestCase;


public class FastTest extends TestCase {

	public void testGetFast() {
		Fast fast = new Fast();
		assertEquals("Fast!!", fast.getFast());
	}

	public void testGetFurious() {
		Fast fast = new Fast();
		assertEquals("Furious!", fast.getFurious());
	}

}
```

Führt den Test aus (s.o.)
Was ist passiert?
Formuliere in deinen Worten, was wir in den Schritten 8-10 gemacht haben.
Was kann man außerdem noch testen?
Siehst du Verbesserungsmöglichkeiten?
Führe diese Verbesserungen durch!

Stelle eine auslieferungsfertige Version der Software her

```
$ mvn package
```

Wo ist das Auslieferungspaket?
Was beinhaltet es?

Lass Maven das Projekt wieder aufräumen.

```
$ mvn clean
```

Fast and Furious:
Apache Maven 2

Fachhochschule Köln
Cologne University of Applied Sciences

- Rekapitulation

Nach den ersten Aufgaben...

- Kickstart hat einen *Projekt-Zyklus* abgebildet
  - Projekt aufsetzen / Klassen schreiben, testen / Auslieferung vorbereiten
  - Maven hat den Entwicklungszyklus von Anfang bis Ende unterstützt

Projekt-Zyklus:

das ist Software-Entwicklungs-Alltag!
Aufsetzen / Entwickeln, Testen / Bauen / Ausliefern

(natürlich nicht alle gleichwertig häufig.)

- Im Vergleich mit Maven

Maven im Vergleich

- make (1977)
  - in C geschrieben
  - erfreut sich weiterhin großer Beliebtheit
  - Explizite Beschreibung des Verhältnisses zwischen Dateien
  - Erweiterung durch shell-Programme
  - Strukturierung durch Whitespaces und Tabs (!)

```
project.exe : main.obj io.obj
     tlink c0s main.obj io.obj, project.exe,, cs /Lf:\bc\lib
main.obj : main.c
     bcc -ms -c main.c
io.obj : io.c
     bcc -ms -c io.c
```

aus: http://www.opussoftware.com/tutorial/TutMakefile.htm

MAKE ab 1977

shell-Programme haben eine Anhängigkeit auf dem Betriebssystem, für das sie geschrieben sind

für die Entwicklung auf verschiedenen Betriebssystemen ungeeignet

Whitespaces und Tabs visuell nicht unterscheidbar... komplizierte Fehlersuche

Maven im Vergleich

- Ant (2000)
  - „Another Neat Tool“
  - in Java geschrieben
    - Erweiterung durch Java-Klassen
  - Konfiguration in XML
    - Entwickler stellen sich eine Baum-Struktur an Ausführungszielen zusammen
  - Write once, run everywhere
  - Notfalls auch shell-Befehle möglich

ANT ab 2000

hauptsächlich für Java Programme genutzt

Aufwändige XML Konfiguration, welche hoch anpassbar ist
wiederholt sich aber auch recht stark in verschiedenen Projekten

Betriebssystem unabhängig! Entwicklung auf allen OS gleichzeitig möglich (solange Java läuft)

```
<project name="MyProject" default="dist" basedir=".">
    <description>
        simple example build file
    </description>
  <!-- set global properties for this build -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist"  location="dist"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
        description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
        description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

aus: http://ant.apache.org/manual/using.html

eine beispielhafte Build.xml

Definition des WAS (init, compile, dist, clean)

und Definition des WIE (compile mit javac; Angabe von build/dist Verzeichnissen)

keine Konventionen, alles wird in der Datei definiert. Wenn ich das build Verzeichnis in „xy“ umbenenne, ist dies problemlos möglich

keine vorgefertigten Tasks, da nicht auf Standards / Konvention zurückgegriffen werden kann.

Libs müssen von Hand eingebunden werden (nicht im Skript zu sehen)

- Apache Maven 2

Apache Maven 2

- Software Project Management und *comprehension* tool*
- Zentrales Konzept: das Project Object Model (POM)

* (Verständnis, Bedeutungsumfang)

Maven definiert sich selbst als Software Projekt Management Tool:

größerer Umfang als make, ant und Konsorten
höherer Anspruch
höhere Abstraktionsebene

Maven wird im Projekt strategisch eingesetzt

Maven ist um ein zentrales Konzept herum entwickelt worden: POM

Anforderungen

- Build ist soll nicht unnötig kompliziert sein
- Entwickler müssen gleiche Versionen von eigenen Komponenten und Fremdbibliotheken haben
  - falls nicht: „Auf meiner Kiste läufts aber!“
- Wissen über den Build darf nicht zentralisiert sein

– Build soll nicht kompliziert sein... dies ist ein Wert für sich!
sonst: am Ende baut sich jedes Projekt seine eigenen Standards / Strukturen auf

– unterschiedliche Konfigurationen von Entwicklungsumgebung und Betriebssystem und unterschiedliche Versionen von Bibliotheken führen dazu, dass der Build nur manchmal läuft

– Wissen darf nicht zentralisiert sein, sonst heißt es „Meier ist nicht da, der macht das sonst immer“
– nicht nur Wissen, auch tools zum Build / Deployment dürfen nicht bei einem Mitarbeiter zentral sein

Ziele von Maven

- Vereinfachung des Build-Prozesses
- Einen gleichförmigen Build bieten
- Qualitative Informationen über das Projekt geben
- Guidelines für Best Practices
- Einfaches updaten von Plugins und Maven selbst

* (Verständnis, Bedeutungsumfang)

Vereinfachung des Build Prozesses

Maven verbirgt einige Details des Build Prozesses (bspw. wohin gehen kompilierte Klassen; welcher Kompiler usw)

Gleichförmiger Build

POM ist Grundlage jeden Builds und somit eine gleichförmige Grundlage (im Gegensatz zu selbst zusammengestrickten build.xml's)

Bestimmte Plugins sind für jedes Maven Projekt verfügbar (bspw. Compile-Plugin)

Maven Builds: Kennste einen, kennste alle!

Qualitative Projektinformationen

Maven generiert aus den Sourcen über die Plugins vielfältige Reports

Guidelines für best practices

Standard Projekt-Struktur (src/main/xxx; target/ usw)

Unit-Testing als Teil des Lifecycle

Unterstützung von Release Management und Issue Tracking

Einfaches Updaten

Updaten von Plugins extrem einfach

Maven update ebenfalls

Features

- Standard Lifecyles
- Dependency Management
- Multi-Modul Projekte
- Erweiterbarkeit durch Plugins

Lifecycles
mehrere Lifecycles mit strukturierten Phasen

Dependency Management
Einbindung von Bibliotheken und Frameworks, Maven lädt JARs selbständig

Multi-Modul Projekte
Zur besseren Struktur innerhalb von großen Projekten

Plugins
Erweitern Maven

Prinzipien von Maven

- Convention over Configuration
- Deklarative Ausführung
- Wiederverwendung von Build-Logik
- Kohärente Organisation von Abhängigkeiten

Fachhochschule Köln
Cologne University of Applied Sciences

Prinzipien erlauben Organisation von Builds auf höherer Abstraktionsebene

Convention over Configuration:
Entwickler kümmern sich darum, was passieren soll und streiten nicht darüber, wie.
Falls Änderungen vom Standard notwendig sind, ist dies möglich. Andernfalls ist jedes Projekt gleich aufgebaut, Einarbeitungszeit wird reduziert.

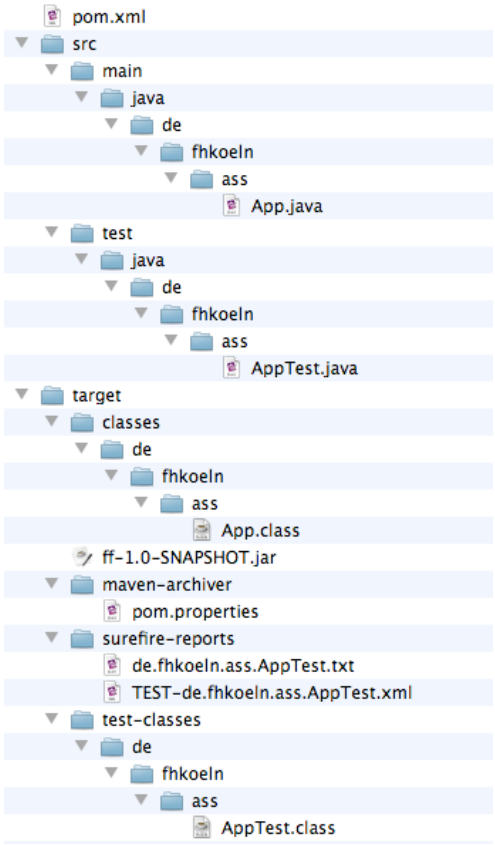
Deklarative Ausführung:
Die Ausführung des Builds in Maven ist bestimmt von Plugins, welche Phasen im Lifecycle zugeordnet sind.
Plugins sind deklarativ im POM beschrieben

Wiederverwenden von Build-Logik:
Build Logik wird gekapselt in Modulen --> Plugins!
Einfache Verteilung und Wiederverwendung

Kohärente Organisation von Abhängigkeiten:
Abhängigkeiten oder Dependencies in Maven-Sprache, werden über ein einheitliches Benennungsschema importiert
Die dazugehörigen Binaries (JARs, WARs, etc) werden automatisch runtergeladen

- POM/Projektstruktur, Lifecycle und Dependency Management im Detail

Projektstruktur im Finder



Das POM befindet sich als pom.xml in unserem Haupt-Projektordner

Sieht auf den ersten Blick nach unnötig vielen Verzeichnissen aus, nachdem man sich aber einmal an die Konvention gewöhnt hat, findet man sofort alles.

POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>de.fhkoeln.ass</groupId>
  <artifactId>ff</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>ff</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```


Wichtig sind hier der zweite Block (Wie heisst das Projekt, wozu gehört es und was erzeugt es) und der Dependencies-Block (was braucht es)

POM - <dependencies>

- Download bei Maven-Aufruf
- Download aus öffentlichen Repositories
- Ablage in lokalem Repo
- ~/.m2/repository/groupid/artifactid/version/ artefaktname-versionsnummer.jar

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>
</project>
```


Dependency wird heruntergeladen, sobald Maven aufgerufen wird.

Es gibt öffentliche Repositories, aus denen Dependencies geladen werden

Sie werden in ein lokales Repository geladen

Dependencies liegen in /Users/name/.m2/repository/...

POM - <build>

- Angabe von Ressourcen und weiteren Infos
- Einbindung und Konfiguration von Plugins

```
<project ...>
   ...
     <build>
          <finalName>Name der Anwendung</finalName>
          <resources>
             <resource>
                  <directory>src/main/resources</directory>
             </resource>
          </resources>
          <testResources>
             <testResource>
                  <directory>src/test/resources</directory>
             </testResource>
          </testResources>

          <plugins>
               <plugin>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>2.0.2</version>
                    <configuration>
                         <source>1.5</source>
                         <target>1.5</target>
                    </configuration>
               </plugin>
          </plugins>
     </build>
   ...
</project>
```


Plugins werden ganz ähnlich wie Dependencies behandelt (Download, lokales Speichern)

Transitive Abhängigkeiten

- Verwendete Libraries verwenden ihrerseits weitere Libraries.
- Wenn zum Beispiel Hibernate log4j in der Version 1.2.x verwendet, Struts aber die Version 1.1.x, können zur Laufzeit sonderbare Effekte auftreten (NoSuchMethodException/IncompatibleClassChangeError).
- mvn dependency:tree zeigt transitive Abhängigkeiten, verlässt sich dabei auf die pom.xml-Dateien aus den Repositories
- Ohne maven muss das manuell überwacht werden, was in der Praxis meist „vergessen wird“

Die oben aufgeführten Fehler sind sehr schwer zu beseitigen, da sie meist nur in Produktions-Umgebungen überhaupt auffallen.

Beispiel dependency:tree

```
[INFO] ------------------------------------------------------------------------
[INFO] Building Fast and Furious - Multi Module Example Project ** Data Layer **
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
[INFO] [dependency:tree {execution: default-cli}]
[INFO] de.fhkoeln.ass:data-layer:jar:1.0-SNAPSHOT
[INFO] +- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.springframework:spring:jar:2.5.6:compile
[INFO] |  \- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] +- org.apache.cxf:cxf-rt-frontend-jaxws:jar:2.2.3:compile
[INFO] |  +- xml-resolver:xml-resolver:jar:1.2:compile
[INFO] |  +- org.apache.geronimo.specs:geronimo-jaxws_2.1_spec:jar:1.0:compile
[INFO] |  |  \- org.apache.geronimo.specs:geronimo-activation_1.1_spec:jar:1.0.2:compile
[INFO] |  +- org.apache.geronimo.specs:geronimo-ws-metadata_2.0_spec:jar:1.1.2:compile
[INFO] |  +- asm:asm:jar:2.2.3:compile
[INFO] |  +- org.apache.cxf:cxf-api:jar:2.2.3:compile
[INFO] |  |  +- org.apache.cxf:cxf-common-utilities:jar:2.2.3:compile
[INFO] |  |  |  +- org.apache.geronimo.specs:geronimo-stax-api_1.0_spec:jar:1.0.1:compile
[INFO] |  |  |  +- wsdl4j:wsdl4j:jar:1.6.2:compile
[INFO] |  |  |  \- commons-lang:commons-lang:jar:2.4:compile
[INFO] |  |  +- org.apache.ws.commons.schema:XmlSchema:jar:1.4.5:compile
[INFO] |  |  +- org.apache.geronimo.specs:geronimo-annotation_1.0_spec:jar:1.1.1:compile
[INFO] |  |  +- org.codehaus.woodstox:wstx-asl:jar:3.2.8:compile
[INFO] |  |  +- org.apache.neethi:neethi:jar:2.0.4:compile
[INFO] |  |  \- org.apache.cxf:cxf-common-schemas:jar:2.2.3:compile
[INFO] |  +- org.apache.cxf:cxf-rt-core:jar:2.2.3:compile
[INFO] |  |  +- com.sun.xml.bind:jaxb-impl:jar:2.1.12:compile
....
```


Der Lifecycle

- default Lifecycle
  - Bau und Auslieferung des Projekts
- clean Lifecycle
  - Aufräumen von Generaten, Kompilaten usw.
- site Lifecycle
  - Erstellen der Projektdokumentation
- Für alle gilt: das Aufrufen einer Phase beinhaltet den Aufruf aller vorhergehenden Phasen!

Nicht zu verwechseln mit dem Lebenszyklus einer Software (Software-Lifecycle)

default Lifecycle

`validate`	Validierung von Informationen, Vollständigkeit
`generate-sources`	Generiert Sourcen
`compile`	Kompiliert sämtliche Sourcen
`test`	Führt die Testklassen aus
`package`	Legt die Binaries im Auslieferungsformat ab
`integration-test`	Liefert das Paket in eine Auslieferungsumgebung zum Test aus
`verify`	Prüft auf Qualitätskriterien
`install`	Installiert das Paket im lokalen Repository, ermöglicht die Einbindung als Dependency
`deploy`	Kopiert das Paket in ein entferntes Repository, ermöglicht die Einbindung als Dependency für andere

Dies ist eine nicht vollständige Darstellung, es gibt weitere Phase dazwischen (bspw. process-sources) Die dargestellten Phasen sind die zentralen Phasen

site Lifecycle

`pre-site`	Ausführung von Prozessen vor der Generierung der Dokumentation
`site`	Generiert Dokumentation und Reports
`post-site`	Postprocessing der Dokumentation und Vorbereitung der Auslieferung
`site-deploy`	Deployt die Dokumentation auf Webserver

Damit das Erzeugen einer site wirklich Sinn macht, müssen diese Informationen im pom gepflegt sein.
Auch das automatische Erzeugen der Javadocs macht nur Sinn, wenn welche vorhanden sind.
Test-Reports verlangen nach UnitTests....

clean Lifecycle

`pre-clean`	Ausführung von Prozessen vor dem Löschen von target/
`clean`	Löschen von target/
`post-clean`	Ausführung von Prozessen nach dem Löschen von target/

Aufruf einer Phase

```
$ mvn compile
```


Woher weiß Maven, welche Sourcen? und wo?
--> Convention over Configuration

Was wird die Sourcen kompilieren?
--> Wiederverwendbare Build Logik durch Compile-Plugin

Dieser Aufrufe bewirkt das Folgende:

1. Einlesen des POM
2. Herunterladen aller benötigten Dependencies und Plugins
3. Initialisierung aller Plugins
4. Ordnung der Plugins in die Phasen, an die sie sich „dranhängen“ (entweder durch ihr default oder unsere Konfiguration)
5. Durchlauf aller Phasen bis Phase „compile“, Aufruf aller Plugins in diesem Ablauf

Ladies and Gentlemen,

Start your Engines!

Praktische Infos für Maven

Maven Installation & Test

Maven 2 herunterladen und in ein beliebiges Verzeichnis entpacken. Dann die Umgebungsvariable M2_HOME auf dieses Verzeichnis setzen. Dann muss das $M2_HOME/bin Verzeichnis in den Pfad aufgenommen werden.
Test mit:

```
$ mvn --version
```

Woher bekomme ich Dependencies und Plugins?

Hier muss man erstmal unterteilen: Du brauchst zwei Dinge für das Einbinden von Abhängigkeiten und Plugins: 1. die XML-Definition und 2. die Binaries.

Es gibt Quellen, da bekommt man direkt beides:
die Webseite des Plugins / Dependency, welches du suchst
http://repository.apache.org
http://www.artifact-repository.org
http://mvnrepository.com
http://www.mvnbrowser.com
http://www.jarvana.com
http://mavensearch.net
und wie immer: Google

Andernfalls kannst du auch Binaries runterladen und sie von Hand ins lokale Repo installieren. Hierzu benutzt man das maven-install-plugin.

Was kann man alles ins POM schreiben?

Eine vollständige Übersicht bietet die Referenz des POMs:
http://maven.apache.org/ref/2.0.9/maven-model/maven.html

Praktisch im Umgang mit Eclipse ist das Code Completion Feature, welches auch für XML Dateien funktioniert. Außerdem gibts ein Plugin für Eclipse, welches den Umgang mit Maven erleichtert: m2eclipse.

Maven im Einsatz

Die folgenden Aufgaben bauen auf dem Kickstart auf. Du benötigst sowohl das pom.xml sowie die Java Klasse und die Testklasse. Auch hier solltest du wieder eine Kommandozeile benutzen, um verschiedene Kommandos auszuführen.
Auch hier gilt wieder: probiere ruhig deine Ideen aus und frag nach, wenn etwas unklar ist.

Dependencies

Integriere Logging durch einen Dependency-Eintrag ins POM!
Wie bist du an den Dependency-Eintrag gekommen?
Für welche Version hast du dich entschieden?
Gibt es Alternativen?

Setz einen Logeintrag in die getFast-Methode.
Prüfe deinen Code, indem du ihn mit Maven kompilierst!
Was passiert beim Ausführen?
Siehst du die Logausgabe? Wo sollte sie hingehen?
Funktioniert der Test weiterhin?
Funktioniert das Eclipse-Projekt weiterhin?
Falls nein: was funktioniert nicht? Wie kannst du das beheben?

Erweitere die Maven Projektstruktur um ein Resources-Verzeichnis
Maven-Konvention: src/main/resources
Kopiere die log4j.properties Datei in das resources-Verzeichnis

Prüfe das Logging erneut
Lassen sich die Sourcen nun kompilieren?
Meldet das Projekt in Eclipse immer noch Fehler? Oder wieder?
Was ist der Output des Tests?
Gibt es irgendwo eine Log-Ausgabe?
Wenn ja: woher weiß Log4j nun von der Properties-Datei?

Plug-Ins

Integriere ein Plugin, welches die Testklasse(n) in ein eigenes JAR packt!
Das Jar soll immer dann erstellt werden, wenn auch für den Rest der Applikation ein JAR gebaut wird. Das JAR soll nicht erstellt werden, wenn nur kompiliert oder getestet wird.

Integriere ein Plugin, welches die Sourcen der Testklasse(n) in ein eigenes JAR packt!
Das Jar soll immer dann erstellt werden, wenn auch für den Rest der Applikation ein JAR gebaut wird. Das JAR soll nicht erstellt werden, wenn nur kompiliert oder getestet wird.

Projekt-Management

Fülle das POM mit sinnvollen Informationen über das Projekt
Die Maven-Webseite (maven.apache.org) bietet eine Übersicht über gängige Plugins.

Web-Projekt

Erstelle ein neues Web-Projekt (Typ 18, maven-archetype-webapp)

Integriere das jetty-maven-plugin

Führe das jetty:run Goal aus und besuche http://localhost:8080

Fast and Furious:
Apache Maven 2

Der Theorie zweiter Teil

Fachhochschule Köln
Cologne University of Applied Sciences

- Fragen zum letzten praktischen Teil

Euere Meinung?
Vorteile & Nachteile?

- Builds weiter anpassen

Properties strukturieren das POM weiter

- Properties sind key/ value Paare

```
<project ...>
   <properties>
       <!-- Application settings -->
       <copyright.year>2008<copyright.year>
       <!-- Framework dependency versions -->
       <appfuse.version>2.0.2</appfuse.version>
       <spring.version>2.5.4</spring.version>
   </properties>
</project>
```

- Können im POM beliebig verwendet werden

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>${spring.version}</version>
    </dependency>
  </dependencies>
</project>
```


Vorteile bei der Verwendung von Properties: gerade in Multi-Module-Projekten wird so sicher gestellt, dass überall die gleiche Version einer Library verwendet wird.

Properties

- Properties können aber auch
  - geerbt werden (aus Super-POM oder übergeordnetem Projekt)
  - extern gesetzt werden
    - per Kommandozeilen-Parameter
    - per profiles.xml / settings.xml

- Profiles und Settings

Profiles

- Profiles ermöglichen umgebungsabhängige Builds
  - Umgebungsabhängigkeit und Plattformunabhängigkeit bilden Zielkonflikt
- Profiles.xml ist ein Subset der pom.xml
- Profiles modifizieren den Build zur Laufzeit

Profiles

- Profiles sollen ähnliche-aber-unterschiedliche Parameter ermöglichen
  - bspw. die Server URL und Socket für Umgebungen (Entwicklungsumgebung, Testumgebung, Produktivumgebung)
- Profiles verursachen unterschiedliche Builds

Beispiel - Profiles.xml

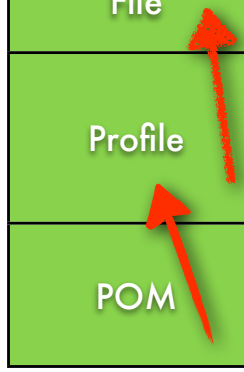
```
<profilesXml>
    <profiles>
        <profile>
            <id>local</id>
            <activation>
                <property>
                    <name>env</name>
                    <value>local</value>
                </property>
            </activation>
            <properties>
                <weblogic.version>9.0</weblogic.version>
                <host>localhost</host>
                <port>7001</port>
                <protocol>t3</protocol>
                <user>weblogic</user>
                <password>geheim</password>
                <target>myLocalServer</target>
            </properties>
        </profile>
    </profiles>
</profilesXml>
```


Definition von Profiles

- im Maven Settings File (meist `<your -home-directory>/.m2/settings.xml`)
- in profiles.xml (liegt im gleichen Verzeichnis wie pom.xml)
- im POM selbst

Definition von Profiles

	Gültigkeit im System	Gültigkeit im Team	Pfad
Maven Settings File	für alle Maven-Projekte auf einem System	1 pro Person	~/.m2/settings.xml
Profile	nur in Projekt und allen Unterprojekten	1 pro Person	profiles.xml neben pom.xml
POM	nur in Projekt und allen Unterprojekten	1 für Alle	pom.xml

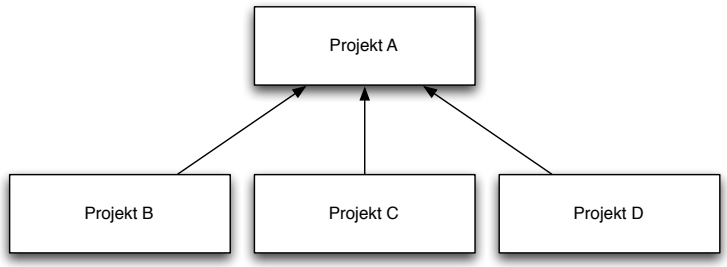


„lokalere“ Optionen überschreiben die globaleren

- Multi-Modul Projekte

Eltern-Kind Beziehung für Projekte

- Projekte B,C,D erben Dependencies von A
- B,C,D haben eigene Plugins und weitere Dependencies



Multi-Module Projekte

Parent

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>de.fhkoeln.ass</groupId>
    <artifactId>ff-parent</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>ff-parent</name>


    <modules>
        <module>A</module>
        <module>B</module>
        <module>C</module>
    </modules>
...
```

Modul A

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>de.fhkoeln.ass</groupId>
        <artifactId>ff-parent</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <artifactId>A</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>ff-A</name>
...
```


Modularisierung

- „Teile und Herrsche“ – Angewandt!
- Vertikale oder Horizontale Modularisierung
- Typischer Fall: n-Tier Architektur

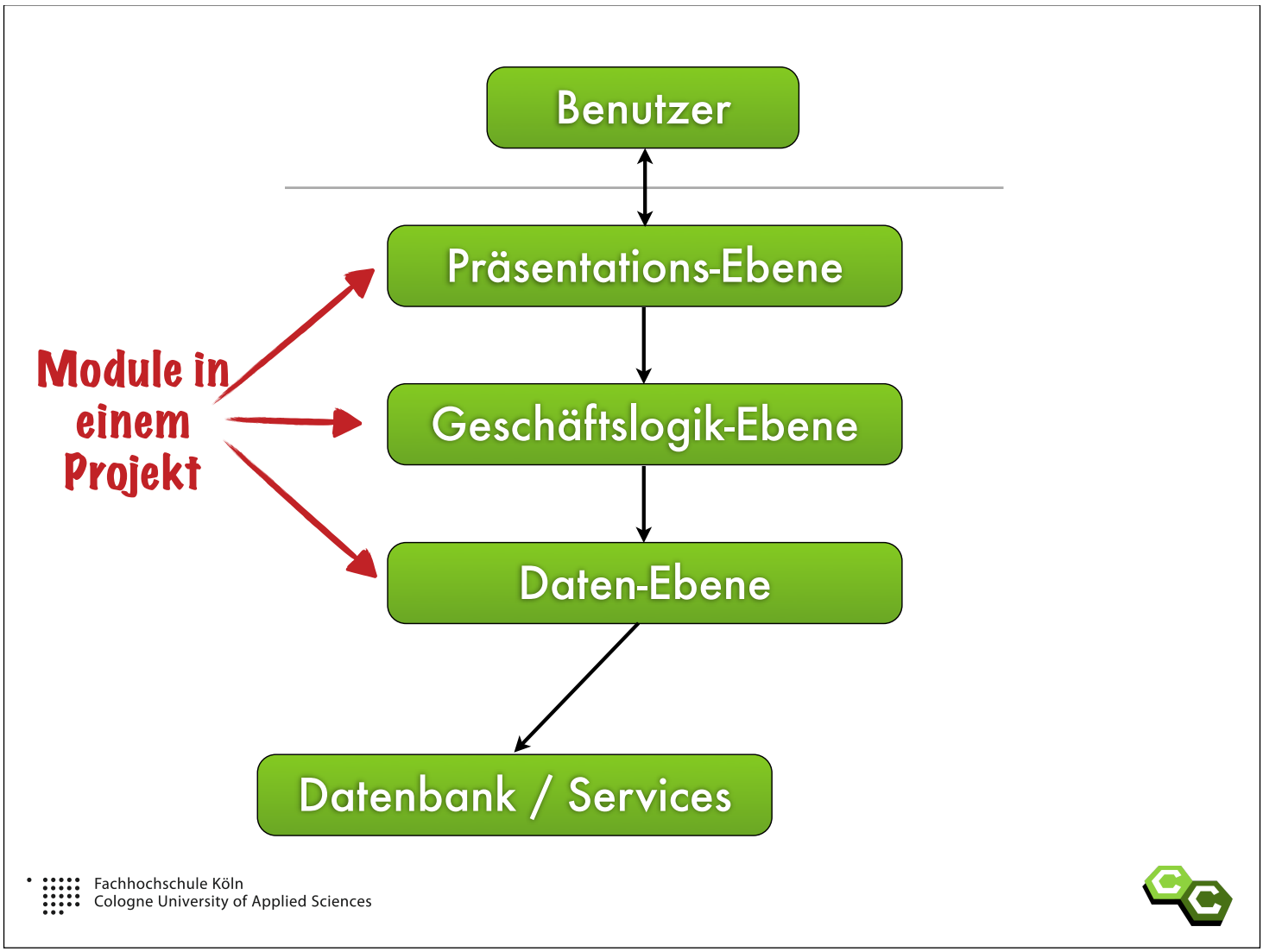
Modularisierung bedeutet angewandtes divide and conquer

Dekorrelierung von Komponenten, die nicht direkt zusammengehören

Verbesserung der Wiederverwendbarkeit (auf Modulebene)
Verbesserung der lokalen Begrenzung von Änderungen (muss nicht alles neu gebaut/getestet werden)

Ein Thema, welches aus den Architekturüberlegungen stammt

Typisch: n-Tier, bzw 3-Tier Architektur



sinnvolle Unterteilung in einer 3-Schichten-Architektur

Interne Projektabhängigkeiten

- Präsentationsschicht nutzt Geschäftslogik nutzt Daten-Ebene
- Abhängigkeiten werden in Modul-POM definiert
- Problem: wie können Entwickler der Präsentationsschicht auf Pakete der Geschäftslogik zugreifen?

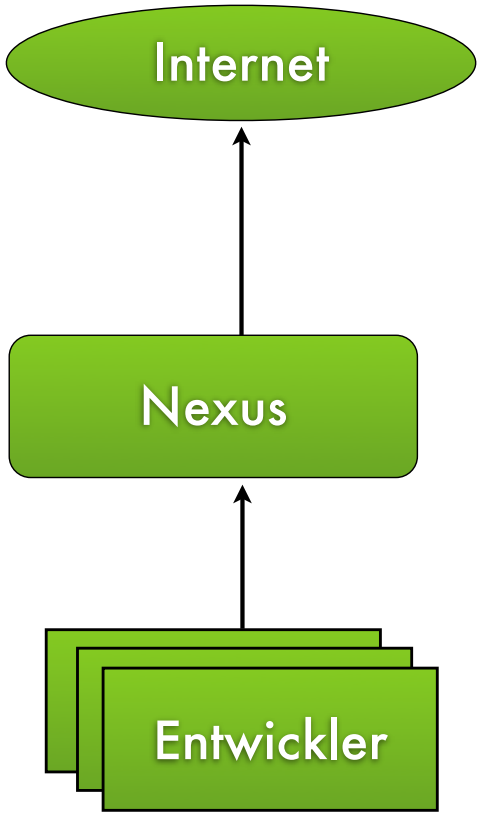
- Dependency Management

Repositories

lokal	~/.m2/repository
LAN	http:// server.mycompany.de/...
WAN	Ibiblio, etc.

Nexus Repository Manager

- Proxy zwischen lokalem und entferntem Repository
  - verringert Traffic und Ladezeit beim Download von Dependencies
- Managing von Release und Snapshot Dependencies
  - Nexus prüft Snapshots regelmäßig
- Sharing von Binaries innerhalb eines LANs


Internet
Nexus
Entwickler

Ladies and Gentlemen,

Start your Engines!

Aufgaben im Team

Multi-Modul Projekt

Die nächsten Aufgaben basieren auf einem Gerüst, welches wir für euch gebaut haben. Die Applikation liest einen Teil der Daten aus einer Datenbank aus und ein weiterer Teil der Daten stammt aus einem Webservice. Diese Datenschicht ist ein eigenes Subprojekt im Multi-Modul Projekt.

Automatisiere den Vorgang der Stub-Generierung aus der WSDL
Im Beispielprojekt wurden mit einem WSDL2Java Tool die Java-Sourcen einmalig aus der WSDL generiert. Dies soll durch eine Integration in Maven automatisiert werden! Informiere dich darüber, wo nach Maven Konvention generierte Dateien üblicherweise abgelegt werden.

Überarbeite das Haupt-POM und füge Properties und Variablen ein, wo es deiner Meinung nach sinnvoll ist.

Erweitere das Projekt um ein Modul für die Geschäftslogik und ein Modul für die Präsentationsschicht
Nutze dabei die bestehenden DAOs als Basis. Binde dich an das Interface, mehr braucht man nicht.
Du könntest bspw folgende Services in der Geschäftslogik-Ebene einbauen: listAllCustomers(), listAllProducts(), order(Product p), getInvoice(Order o) usw.
Bitte achte darauf, dass nicht die Programmieraufgaben im Vordergrund stehen sollen. Falls also Probleme auftreten sollten, erledige lieber erst die restlichen Aufgaben dieses Handouts.

Erweitere das Haupt-POM derart, dass durch das Goal „package“ ein WAR entsteht, welches alle Module beinhaltet.

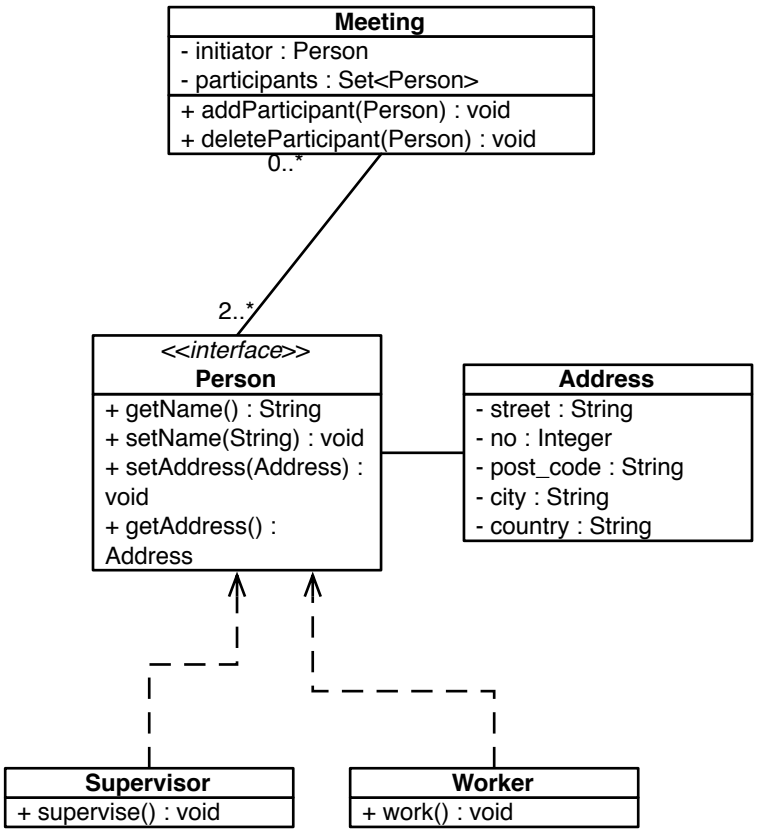
Extra: Generiere aus den Services deiner Geschäftslogik die WSDL für einen eigenen Webservice!
Extra: Sorge dafür, dass das gesamte Projekt kontinuierlich integriert wird.

Erste Schritte für Spring

Aufbau eines Objektnetzes

Wir haben für diesen Abschnitt eine Klassenstruktur entworfen, die ihr nachbauen sollt. Diese Klassenstruktur hätten wir auch anders nennen können, es kommt hier einzig und allein auf die Beziehungen zwischen den Klassen an. Die Aufgabe soll eine Problemstellung vermitteln, die euch möglicherweise noch nicht bewusst und nur schwer zu erkennen ist. Es ist wichtig, dass ihr über eure Probleme mit der Aufgabenstellung nachdenkt.

Erstelle eine Klassenstruktur nach Vorlage der Abbildung.



Erstelle eine Instanz der Klasse Meeting mit mehreren Personen als Teilnehmer.
Das Meeting wird mit mehreren teilnehmenden **Person-Instanzen** besetzt. Es sollen sowohl Supervisor als auch Worker dabei sein.
Versuche, die einzelnen Klassen so unabhängig wie möglich zu schreiben!

Bitte schriftlich in Stichpunkten: Formuliere, wo welche Klassen untereinander gekoppelt sind!
Bspw: Klasse X hat eine direkte Kopplung zu Klasse Y in Methode zXY.
Wie würdest du die Kopplungen beschreiben?

Welche Kopplungen sind notwendig, welche sind es nicht?

Lasse dir das Meeting als Text ausgeben, mit allen Teilnehmern und Adressen.

Falls du es noch nicht getan hast: entwirf eine Klasse, die dir ein Person-Objekt zurückgibt, wenn du einen „Worker“ oder „Supervisor“ anforderst.
Was ist der Vorteil dieses Verfahrens? Wo liegt nun die Kopplung zwischen den Klassen im Gegensatz zu vorher?
Tipp: Für Infos zu diesem Verfahren suche nach dem Factory-Method Design Pattern.

Fast and Furious:
The Spring Framework

Fachhochschule Köln
Cologne University of Applied Sciences

- Rekapitulation

Nach den ersten Aufgaben...

- Was ist Kopplung? Was ist das Problem mit Interfaces? Was ist das Problem mit „new“?
- Kickstart soll Problematik im Aufbau von Objektnetzen zeigen

- Im Vergleich mit Maven

Vergleich mit Maven

- Maven arbeitet „an“ der Software
  - generiert Sourcen, kompiliert
  - sorgt für Paketierung, Auslieferung
  - ist aber im Betrieb nicht mehr präsent
- Spring wirkt „in“ der Software
  - konfiguriert zur Laufzeit
- Maven und Spring haben keine Schnittmenge!

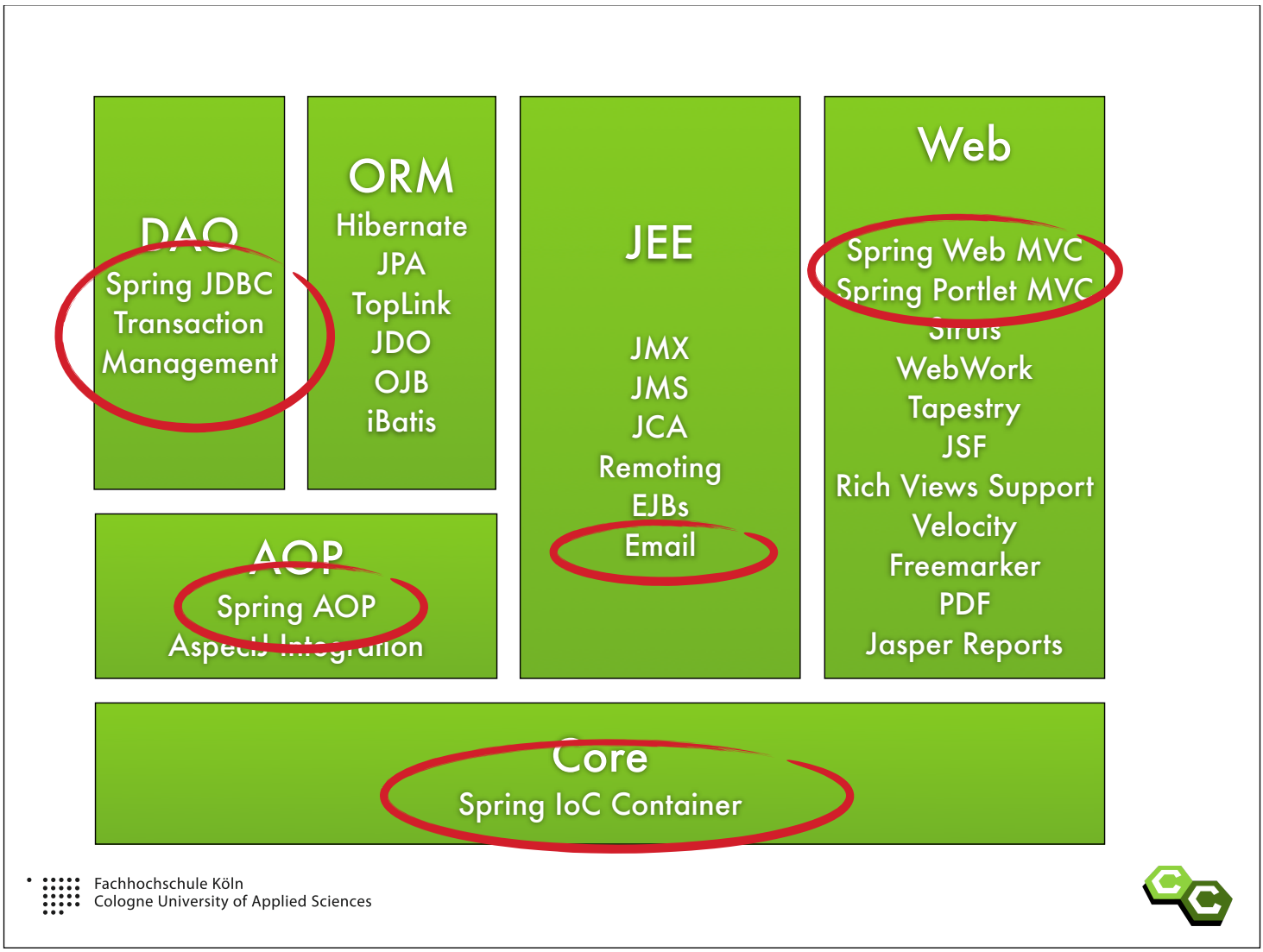
Auch wenn sich beides um Dependencies dreht, sind meist unterschiedliche Stufen gemeint. Überschneidung gibt es z.B. wenn Spring verwendet wird, um eine Klasse zu injizieren, die aus verschiedenen Libraries stammen kann. Z.B. JDBC-Treiber. Dann wird mittels Spring erreicht, dass man mittels Maven verschiedene Treiber anziehen kann.

The Spring Framework

Spring

- Spring ist eine Applikations-Plattform für Java
- Spring ist ein leichtgewichtiger Container
  - Programmcode ist nicht an Spring gebunden, bzw muss es nicht sein
- Spring erlaubt die Konfiguration von Applikations-Komponenten
- Spring integriert Frameworks von Drittanbietern

Spring als Applikations-Plattform oder Grundlage: eng verknüpft mit Architekturthemen
Einsatz von Spring oder vergleichbaren JEE-Technologien eine Grundsatzentscheidung --> Strategie!



Woraus besteht Spring eigentlich?

Was ist Spring eigentlich?

Man kann viele Leute fragen und erhält viele verschiedene Antworten

Spring bringt eigene Elemente mit und integriert fremde Elemente, Frameworks

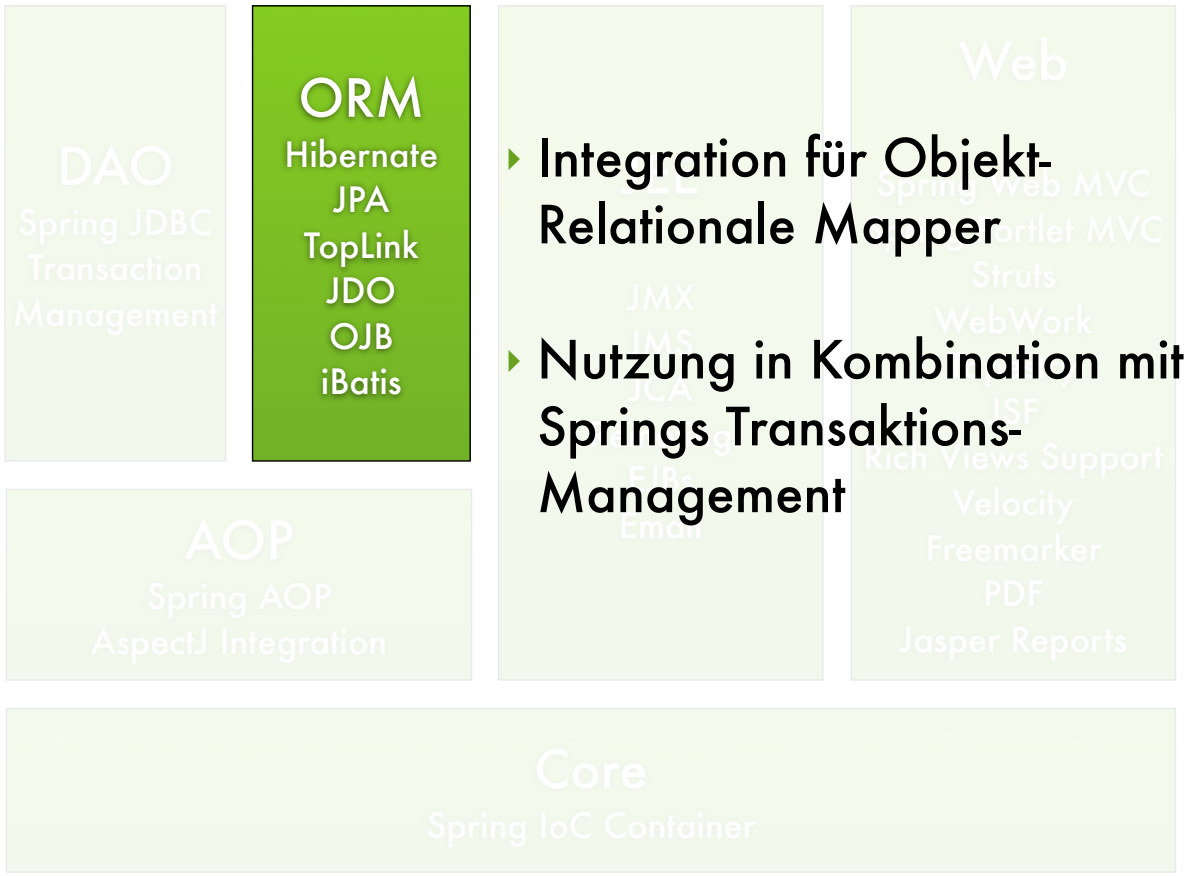
Jetzt drei Elemente im Detail

- Grundlegene Funktionen von Spring
- Dependency Injection
- `BeanFactory / ApplicationContext` als zentrale Konzepte
- Entkopplung von: [Konfiguration und Spezifikation von Abhängigkeiten] / Programm Logik

Core
Spring IoC Container

Spring wird häufig als DI-Framework oder Container bezeichnet. Dies ist unzureichend beschrieben!

- eigene JDBC-Abstraktion
  - kein mühsames JDBC-Coding mehr
  - kein DB-spezifisches SQL mehr
- Programmatisches und deklaratives Transaktions-Management



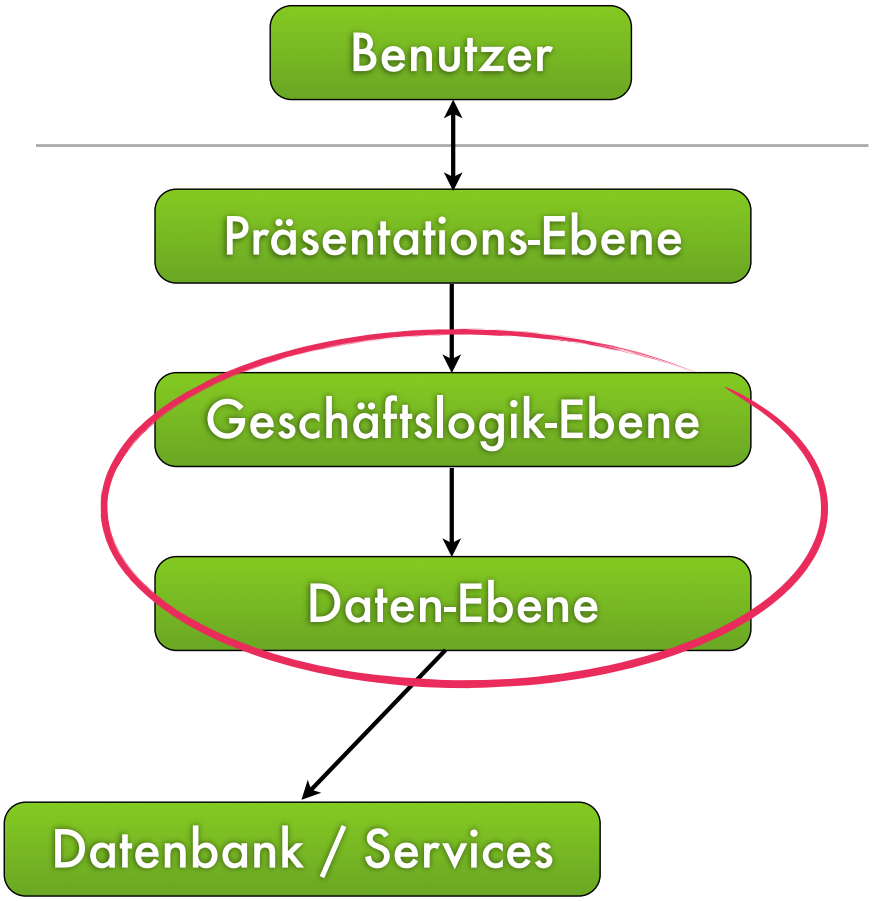
- Integration für Objekt-Relationale Mapper
- Nutzung in Kombination mit Springs Transaktions-Management

Konfiguration

- applicationContext.xml / spring-config.xml / ...
  - Keine Namenskonvention, überlicherweise hierarchische Aufteilung in mehreren Dateien
- Meist zentraler Ort für Konfiguration der Software
- Verrät Infos über Architektur im Großen und Kleinen

Alternative zu XML-Hell: Annotations.
Damit wird aber der Code an Spring gekoppelt.
(@Ressource)
Teilweise Aufgabe von Inversion of Control.

- Dependency Injection mit Spring



Beispiel in diesen Ebenen

Spring hilft, Ebenen voneinander unabhängig zu gestalten

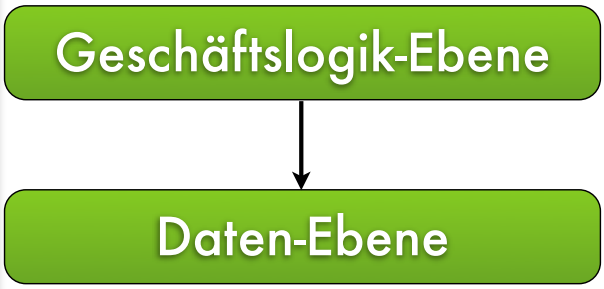
Beispiel

```
public class BusinessLogic {
  private SpecialDataLayer dataLayer;

  public void doBusinessLogic() {
    if(that) {
      dataLayer.doThis();
    }
  }
}


public class SpecialDataLayer
  implements DataLayer {

  public void doThis() {
    this.accessDB();
  }
}
```



Beziehung ohne Kopplung

- BusinessLogic-Klasse ist an DataLayer-Interface gebunden
- Problem: Interfaces sind nicht instanziierbar
- Lösung: Dependency Injection

```
public class BusinessLogic {
  private DataLayer dataLayer;

  public void doBusinessLogic() {
    if(that) {
      dataLayer.doThis();
    }
  }
}


public class SpecialDataLayer
  implements DataLayer {

  public void doThis() {
    this.accessDB();
  }
}
```


Kontrollfluss bei BusinessLogic

```
public class BusinessLogic {

   ... = new SpecialDataLayer();


}
```

Was ist das Problem mit new??

Es gibt keines!

Das Problem ist die Veränderung, die in Klassen natürlicherweise eintritt.

Die einzige Konstante ist die Veränderung!

Alternative: setter

```
public class BusinessLogic {

   private DataLayer dataLayer;


   public void setDataLayer(DataLayer dataLayer) {
      this.dataLayer = dataLayer;
   }

}
```

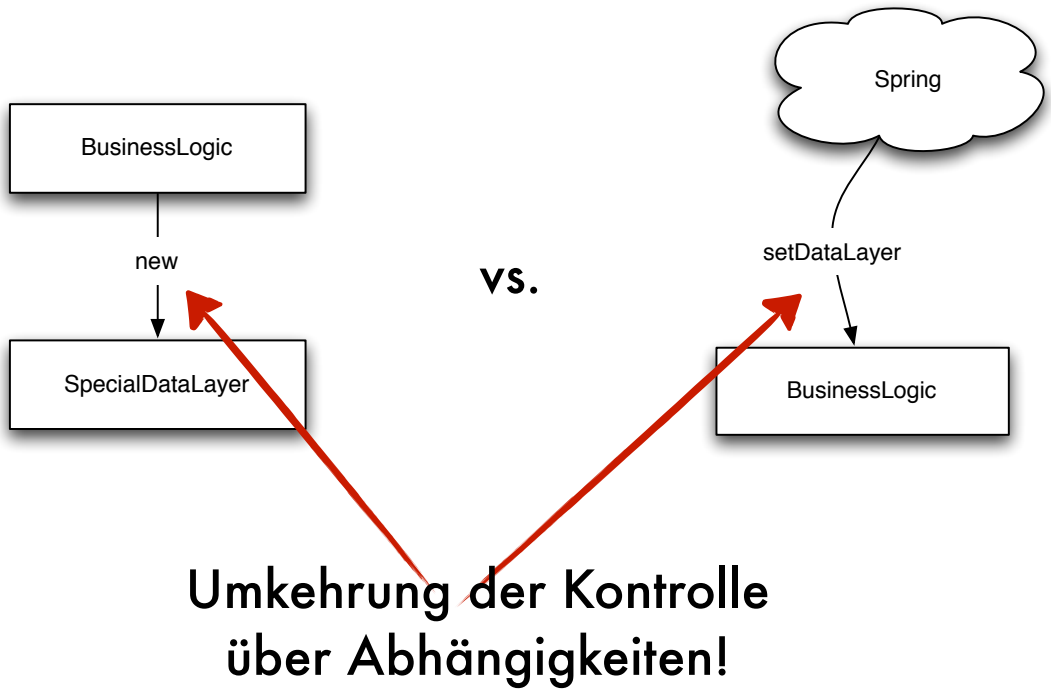

Ergebnis:

Kopplung via Interface – gewünscht!

Keine Bindung an die Klasse.

Instanziierung geschieht von extern.

Design Prinzip: Binde dich an Schnittstellen, nicht an Implementierungen!
Implementierungen ändern sich, Schnittstellen (eigentlich) nicht.



Umkehrung der Kontrolle
über Abhängigkeiten!

Abhängigkeit liegt nun bei Spring, genauer: in der Konfiguration.

Aber: das ist uns EGAL! Denn das kann man in Kauf nehmen...

Spring Konfiguration

```
<beans>

  <bean name="businessLogic" class="de.fhkoeln.ass.ff.business.BusinessLogic">
    <property name="dataLayer" ref="specialDataLayer" />
  </bean>

  <bean name="specialDataLayer"
  class="de.fhkoeln.ass.ff.datalayer.SpecialDataLayer" />

</beans>
```


Sobald ich mir von Spring ein BusinessLogic Objekt geben lasse, wird es zur Laufzeit erstellt

Vorteile der Konfig:
- Flexibilität: schnelle Umkonfiguration auf Mocks, andere DB, anderes ORM

Patterns – Das Singleton

- Was sind Patterns?
- Spring verwendet standardmässig Singletons.
- Vor-/Nachteile von Singletons

Singleton: Highlander-Prinzip. Häufig fehlerhaft implementiert (Double-Null-Check-Antipattern)
Vorteil: Weniger Garbage, weniger Initialisierungs-Aufwand, zentrale Kontroll-Punkte (synchronized, begrenzte Ressourcen)
Achtung bei synchronized: Singletons können auch die Performance zerstören, wenn sie falsch angewendet werden.
Was muss man bei Spring tun, wenn man keine Singletons möchte?

- Spring JDBC

and now to something completely different, spring jdbc support

JDBC-Unterstützung

- JDBC ist Schnittstelle zwischen DB und Java
- JDBC Coding
  - aufwändig
  - fehlerbehaftet

```
Statement stmt = conn.createStatement();
try {
    ResultSet rs = stmt.executeQuery( "SELECT * FROM MyTable" );
    try {
         doStuffWith(rs);
        }
    } finally {
        rs.close();
    }
} finally {
    stmt.close();
}

...

conn.close()
```

Aufbau von Connection, Statement, ResultSet

Exception-Handling (Treiber-spezifisch. Durch Spring vereinheitlicht!)

Aufräumen von Ressourcen (Häufige Fehlerquelle, besonders schwierig zu debuggen, Anwendung läuft stunden oder Tage und bleibt dann einfach stehen....)

Vereinfachung von Spring: JDBC-Template

- Erzeugung aus DataSource, durch Dependency Injection oder Vererbung
- JdbcTemplate enthält Methoden zum Ausführen von SQL-Statements:
  - `execute(), query(), update(), batchUpdate()`
  - selbständige Erzeugung von Connections und Aufräumen von Ressourcen
  - notfalls auch Arbeit mit JDBC-Klassen direkt möglich.

Z.B. auch Blobhandler: Ist DB-spezifisch und wird durch Spring vereinheitlicht

Beispiel

```
getJdbcTemplate()
   .update(„DELETE FROM KUNDE WHERE ID=?“, new Object[] {new Integer(id)} );
```


Ladies and Gentlemen,

Start your Engines!

Konfigurationen mit Spring

Das letzte Handout sollte zeigen, dass man gezwungen ist, irgendwo Klassen zu instantiieren. Dabei ist es egal, ob man Interfaces verwendet oder nicht, irgendwo werden Abhängigkeiten geschaffen und sei es zu / in einer Factory.
Nun versuchen wir, diese Probleme mit Hilfe von Spring zu umgehen. Wir werden eine andere Klassenstruktur verwenden, die ein Beispiel für die Geschäftslogik- und Daten-Ebene ist. Wir möchten die beiden Ebenen soweit wie möglich entkoppeln.

Aufbau eines Objektnetzes

Für diese Aufgaben stellen wir ein Projekt-Grundgerüst zur Verfügung.

Füge in dein Maven-Projekt eine Dependency für Spring in Version 2.5.6 ein

Erstelle die Klassen OrderService, CustomerService und ProductService. Die Klassen sollen getter/setter für die Order / Customer / Product DAO Klassen besitzen.
Diese Klassen stellen deine Service-Ebene dar. Sie nutzen die DAO Klassen, um Geschäftslogik zu ermöglichen.

Erweitere die DAO Klassen um sinnvolle Methoden.
Beispielsweise Methoden wie save, getAllXXX, update, getXXXByName, usw.

Nutze Springs Dependency Injection, um die Klassen und deren Abhängigkeiten zu strukturieren.
Hierfür muss die von uns vorbereitete Spring-Konfiguration angepasst werden: Konfiguriere das Objektnetz in Spring, strukturiert in Service/Manager-Ebene, DAO-Ebene, Infrastruktur-Beans. Sorge dafür, dass jede deiner Service-Klassen ein entsprechendes DAO und jedes DAO eine SessionFactory durch Spring injiziert bekommt.

Spring JDBC

Erstelle in der Spring Konfiguration eine DataSource, welche auf die Datenbank zugreifen kann.

Erstelle eine Klasse, welche über ein Feld JdbcTemplate jdbcTemplate verfügt und injiziere eine DataSource in deine eigene Spring-Bean (Vgl. Beispiel):

```
private JdbcTemplate jdbctemplate;
public void setDataSource(DataSource dataSource) {
      this.jdbctemplate = new JdbcTemplate(dataSource);
}
```

Erstelle in deiner Bean ein JdbcTemplate-Objekt (Alternativ: Leite deine Klasse von JdbcDaoSupport ab (Vgl. Beispiel oben).

Benutze die query/execute/update Methoden des JdbcTemplates, um Daten in der Datenbank abzurufen und zu verändern.

Spring Resources

Resources erlauben den Input in Klassen über Dateien im Filesystem, wie .properties Dateien, Textdateien, etc.

Füge in eine deiner Beans ein Feld vom Typ Resource ein und benutze die Properties im Code

Injiziere Daten über folgende Konfiguration

```
<bean id="myBean" class="...">
      <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

Das template Feld in der Bean muss vom Typ Resource sein. Gib die Daten als Text in der Konsole oder dem Logfile aus.

Spring Validation

Validiere die Felder deiner Klassen mit Springs Validation-Interface und der ValidationUtils Klasse

Fast and Furious:

Wahnsinnige Spring Features

Fachhochschule Köln
Cologne University of Applied Sciences

Spring hat große Bandbreite an Themen, die man machen kann

Hier wird Überblick über die wichtigsten gegeben (oder die, die wir als wichtig erachten..)

es ist wichtig zu wissen, was Spring alles kann, es ist jedoch zu komplex, um es in einem Workshop zu erlernen

wir werden also praktisch weiter in diese Funktionen eintauchen, hier in dieser Präsentation aber nur oberflächlich beschreiben.

- Rekapitulation

Spring

- Fragen zu Dependency Injection

- Spring Testing

Unit / Integration Testing

- Unit Testing
  - Unit als kleinste kompilierbare Einheit (In Java: Klasse)
  - Beinhaltet nicht den Aufruf von Sub-Komponenten oder jegliche sonstige Kommunikation mit anderen Komponenten
  - Besteht zu anderen Komponenten ein Abhängigkeitsverhältnis
    - Wird die Komponente ge-mockt
    - simuliert
    - oder durch vertrauenswürdige Komponenten ersetzt

Warum ist Unit-Testing ohne Spring unmöglich? Weil nur durch den Austausch der Laufzeit-Abhängigkeiten durch Mocks überhaupt erst Units entstehen, die einzeln getestet werden können.

Unit / Integration Testing

- Integration Testing
  - Komponenten werden in ihrem Zusammenspiel getestet
  - Beinhaltet Aufruf von Sub-Komponenten
  - Im Gegensatz zum Unit-Testing muss kompletter Kontext / Infrastruktur vorhanden sein
  - Längere Laufzeiten

Da maven bei jedem Build unit-Tests durchführt, Integration-Tests aber oft zu zeitaufwendig sind, um immer wieder ausgeführt zu werden, sollte man diese dem Continous-Integration-Server überlassen. Dazu werden die Unit-Tests in verschiedene Gruppen aufgeteilt („simple“, „complex“, „integration“). Sonst führt das zu laufender Verwendung von -Dmaven.test.skip=true

Spring und Testing

- Unit-Testing
  - Mocks für JNDI, Servlet API und Portlet API
  - Supportclasses vor allem für die Nutzung von Reflections und Testing von Spring MVC
- Integration Testing
  - Testing ohne Deployment
  - Testet korrekte IoC-Konfiguration
  - Support für Transaktionen (Rollbacks), Test Fixtures, uvm.

Mocking in Spring jedoch nur in Teilen sinnvoll, andere Mocking-Frameworks leisten bessere und umfangreichere Dienste

- Spring Transaction-Management

Transaktionen

- einzelne und unteilbare Transaktionen müssen im Ganzen gelingen oder scheitern
- ACID Kriterien
- Commit oder Rollback der Transaktion

ACID criteria (Atomicity, Consistency, Isolation, Durability)

Spring Transaktionen

- Reichhaltige Unterstützung von Transaktionen
- Einheitliches Transaktionsmodell über andere Transaktions-APIs (JTA, JDBC, Hibernate, usw)
- Deklarativ oder Programmatisch
- Integriert mit Springs Datenzugriffs-Abstraktionen

Beispiel

```
public class ServiceClass {

  @Transactional
  public void update(Thing thing) {
    dao.update(thing);
    ...
  }

}
```


Transaktions-Support für Methode als Beispiel

es ist nicht abgebildet: es wird weitere Konfiguration in XML Datei benötigt

- Spring ORM Integration

ORM

- (O)bject (R)elational (M)apper
- Bilden Objekte auf relationale Strukturen ab
- ORM-Tools automatisieren Zugriff auf Datenbank

Springs ORM Support

- Integration mit Hibernate, JDO, Oracle TopLink, iBatis, JPA
- Unterstützung mit einheitlichem Transaktionsmanagement und Exception-Hierarchien
- Nutzung von Spring DAO Templates oder direkte Nutzung der ORM API

Beispiel

```
public class HibernateProductDao extends HibernateDaoSupport implements
ProductDao {
   public Collection loadProductsByCategory(String category) throws
   DataAccessException, MyException {
      Session session = getSession(false);
      try {
         Query query = session.createQuery("from test.Product product where
         product.category=?");
         query.setString(0, category);
         List result = query.list();
         if (result == null) {
            throw new MyException("No search results.");
         }
         return result;
      }
      catch (HibernateException ex) {
         throw convertHibernateAccessException(ex);
      }
   }
}
```


- Spring Web Integration

Web Frameworks

- Qual der Wahl bei Web Frameworks
- Spring MVC
  - eigenes request-basiertes MVC Framework
- Integration anderer Frameworks
  - Struts, JSF 1 & 2, WebWork 2, Tapestry 3 & 4

- Spring Web Service Integration

Unterstützung von Standard WS-APIs

- Veröffentlichen mit JAX-RPC
- Zugreifen mit JAX-RPC
- Veröffentlichen mit JAX-WS
- Zugreifen mit JAX-WS

2 WS Standards:

JAX-RPC für J2EE 1.4
RPC hat jedoch an Bedeutung und Beliebtheit verloren

JAX-WS für JEE 5

Veröffentlichen eines WS mit JAX-WS

- JAX-WS funktioniert weitgehend mit Annotationen
- SpringBeanAutowiringSupport-Klasse als Basis
  - Wrapping unserer Services

```
@WebService(serviceName="AccountService")
public class AccountServiceEndpoint extends SpringBeanAutowiringSupport {

   @Autowired
   private AccountService biz;

   @WebMethod
   public void insertAccount(Account acc) {
      biz.insertAccount(acc);
   }

   @WebMethod
   public Account[] getAccounts(String name) {
      return biz.getAccounts(name);
   }
}
```


schon haben wir einen Endpunkt konstruiert!

- Spring 3.0

Spring 3.0

- Seit Mitte Dezember ist 3.0 final
- basiert auf Java 5, unterstützt Java 6
- kompatibel mit J2EE 1.4, JEE 5 und teilweise mit JEE 6

- Spring Expression Language
- IoC Verbesserungen
- Objekt / XML Mapping (OXM, aus Spring Web Services Projekt)
- leistungsfähiger REST Support
- @MVC Annotations
- Deklarative Validierung
- Unterstützung eingebetteter Datenbanken

Ladies and Gentlemen,

Start your Engines!

Advanced Spring Features

Transaktionen

Sorge dafür, dass alle Methoden in der Service-Ebene als Transaktionen durchgeführt werden.
Zu beachten ist, dass außer der deklarativen oder programmatischen Integration auch die Konfiguration verändert werden muss. Alle Infos sind in Abschnitt 9.5.6 der Spring Spezifikation zu finden.

Probiere verschiedene Einstellungen für die @Transactional Annotation aus.
Wie kann man auf unterschiedliche Exceptions reagieren? Was sind Propagations und welche Propagations sind für unsere Anwendungsfälle sinnvoll?

Schreibe JUnit Tests für deine Klassen, die transaktionsbasiert sind.
Teste auch, ob die Exceptions im Fehlerfall korrekt geworfen werden. Außerdem überprüfe, ob nicht doch irgendwie Werte in der Datenbank verändert wurden, die nicht hätten geändert werden dürfen.
In den Resourcen findest du die Beschreibung vieler hilfreicher Annotationen, die beim Testen helfen. Allerdings muss JUnit hierfür geupdated werden, falls du sie benutzen willst.

Web Services

Veröffentliche eine deiner Service-Klassen als WebService.
Wie kann man seinen Web Service testen?
Wo kommt die WSDL her? Wofür braucht man eigentlich die WSDL?
Was musst du zuerst erstellen: den Service oder die WSDL?

Baue einen Client für deinen Web Service.

Fast and Furious:
The Complete Picture

Fachhochschule Köln
Cologne University of Applied Sciences

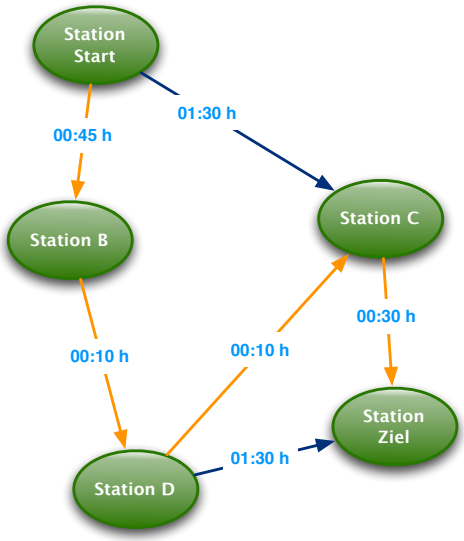
- Rekapitulation

Nach den Grundlagen...

- Nachdem wir nun Module kennengelernt haben, nutzen wir das Wissen für ein „größeres“ Projekt

Routenplaner

Aufgabe: Ermitteln der schnellsten Verbindung von einem Startbahnhof zum Zielbahnhof: ein einfaches Problem aus der Graphentheorie (gewichtete Kanten)



Projektstruktur

Ein Server-Projekt stellt einen Webservice zur Verfügung und ein Client-Projekt bietet auf einer Weboberfläche eine einfache GUI dazu an. Unterschiedliche Teams sollen parallel an den einzelnen Modulen arbeiten. Und möglichst oft integrieren.

Module (Vorschlag)

- Team 1: Modell und Persistenz
- Team 2: Webservice (Server) und Business-Logik (Server)
- Team 3: Webservice (Client) und Business-Logik (Client)
- Team 4: Web-Oberfläche
- Team 5: Import-Modul (csv oder xml)

Was ist zuerst zu tun?

- Nach einer kurzen Lösungs-Skizze müssen die Schnittstellen zwischen den Teams vereinbart werden.
- Das ist zum einen ganz klar die WSDL, diese ergibt sich aber erst aus dem (noch nicht existierenden) Web-Service
- D.h. am Anfang wird das ein Blindflug auf der Basis von Skizzen (So ist das in der Realität...)

Weiteres Vorgehen

- Wenn die ersten Ideen auf Schmierpapier ausgetauscht sind, sollte zuerst jeweils eine Integrations-Umgebung aufgesetzt werden.
- Diese darf und sollte auf Unit-Tests basieren.
- Dadurch kann früh auf die ersten (unvollständigen) Lieferungen aus anderen Teams reagiert werden.

The Complete Picture

Softwareentwicklung im Team

Entwickelt werden soll eine Lösung für Routenplanung mit der Bahn. Dabei soll zum einen eine Server-Anwendung erstellt werden, die die Verbindungsdaten per Web-Service zur Verfügung stellt. Zum Anderen eine Web-Anwendung, die auf einem zweiten Server laufen soll und die Nutzung des Systems erlaubt.

Das Vorgehen soll dabei Extreme-Programming mäßig stattfinden, d.h. wir unterteilen den Tag in mehrere Phasen, wobei jede Phase das Ziel hat, etwas abzuliefern. Zu Anfang jeder Phase führen wir ein Anwender-Interview zur Feststellung der Anforderungen durch. Anwender in diesem Sinne sind nicht nur die Kursleiter, die die fachlichen Anforderungen haben, sondern auch die anderen Teams, die ja ebenfalls (wenn auch eher technische) Anforderungen haben.

Team 1 -- Modell und Persistenz

Deliverable: Ein oder mehrere jars, die von anderen Teams benutzt werden können, um die benötigten Daten in einer Datenbank abzulegen und die benötigten Suchen durchzuführen. Darüberhinaus eine API (Interfaces) zur Beschreibung des Zugriffes.

Mögliche Umsetzung (Vorschläge): Ein Maven Projekt, das das Modell als jar liefert. Modell mit JPA Annotations (für die Persistierung) und eventuell JAXB Annotations (zur Steuerung der Serialisierung über den Web-Service)
Ein weiteres Maven-Projekt mit DAO Klassen und Interfaces, das die Persistierung in einer HSQL Datenbank mittels Spring JDBC oder Hibernate erledigt.

Team 2 -- Webservice (Server) und Business Logic (Server)

Deliverable: Ein .war das auf einer Servlet Engine (Tomcat oder Jetty) deployt werden kann und Webservices zur Verfügung stellt, die ein geeigneter Client nutzen kann, um die benötigten Funktionen durchzuführen. Darüber hinaus eine wsdl zur Beschreibung der Web-Services.

Mögliche Umsetzung (Vorschläge): Ein Maven Projekt, das mittels Spring, CXF und JAX-WS einen Web-Services zur Verfügung stellt.

Team 3 -- Webservice (Client) und Business-Logic (Client)

Deliverable: Ein oder mehrere jars, die von anderen Teams benutzt werden können, um auf die Webservices von Team 2 zuzugreifen und die benötigte Fachfunktionalität (Suche der kürzesten Route) auszuführen. Darüber hinaus eine API (Interfaces) zur Beschreibung des Zugriffs.

Mögliche Umsetzung (Vorschläge): Ein Maven Projekt, das mittels Spring, CXF und JAX-WS auf den Web-Service zugreift. Ein weiteres Maven Projekt, das die Business-Logik als jar zur Verfügung stellt.

Team 4 -- Web-Oberfläche

Deliverable: Ein .war, das auf einer Servlet Engine (Tomcat oder Jetty) deployt werden kann und dem Benutzer die Funktionalität des Systems zur Verfügung stellt.

Mögliche Umsetzung (Vorschläge): Ein Maven Projekt, das mittels Spring-Webmvc eine Benutzer-Oberfläche darstellt.

Team 5 -- Import-Funktion

Deliverable: Ein jar, das per Kommandozeile (oder über eine API aufgerufen) Massendaten in XML entgegen nimmt und über den Web-Service-Client (Team 3) persistiert.

Mögliche Umsetzung (Vorschläge): Ein Maven Projekt, das mittels JAXB Daten einliest.