



Modern Technology Stacks

TU Darmstadt
18.12.2015

Contact information



Tobias Waller

Software Engineer
tobias.waller@capgemini.com

@i3oot



TECHNISCHE
UNIVERSITÄT
DARMSTADT

#seiipTUD

Agenda

- Introduction
- Dependency Injection
- Aspect Oriented Programming
- Object-relational mapping

What is a “Technology Stack”

- No scientific definition
- Depends on who you ask
 - Deployment-Artifact internal view
 - Deployment-Artifact external view

The definition we will use:



**All the software you use to build, deploy
and run your application**

Perspective of a Developer

Application Stack

- Application Framework
- Libraries
- Web server
- Database + ORM
- Runtime Environment

Tool Stack

- Source Code Management
- Testing
- Defect Tracking
- Build Tools
- Integration Tools

Perspective of an Operator

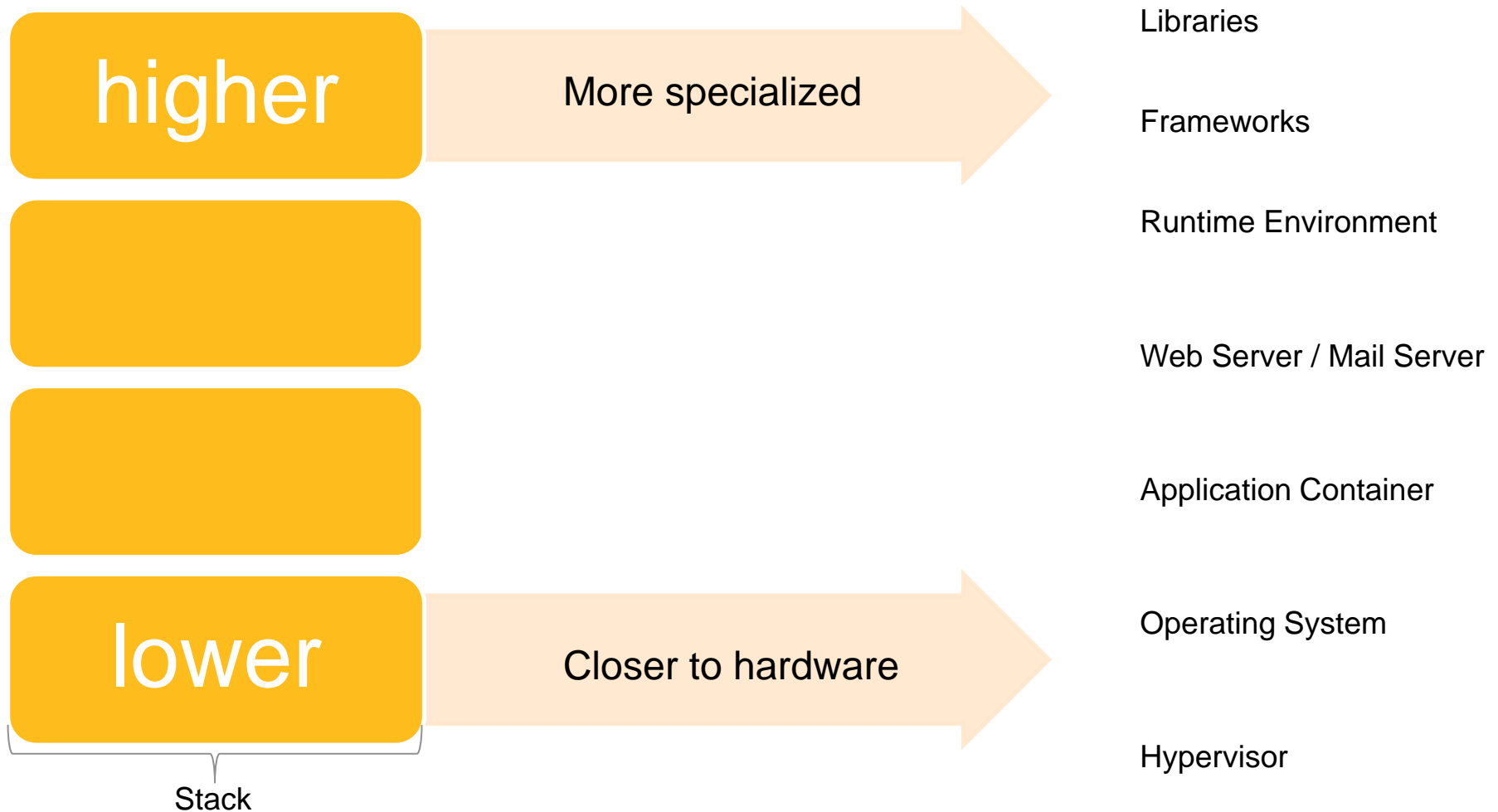
Management Stack

- Provisioning
- Configuration Management
- Load Management

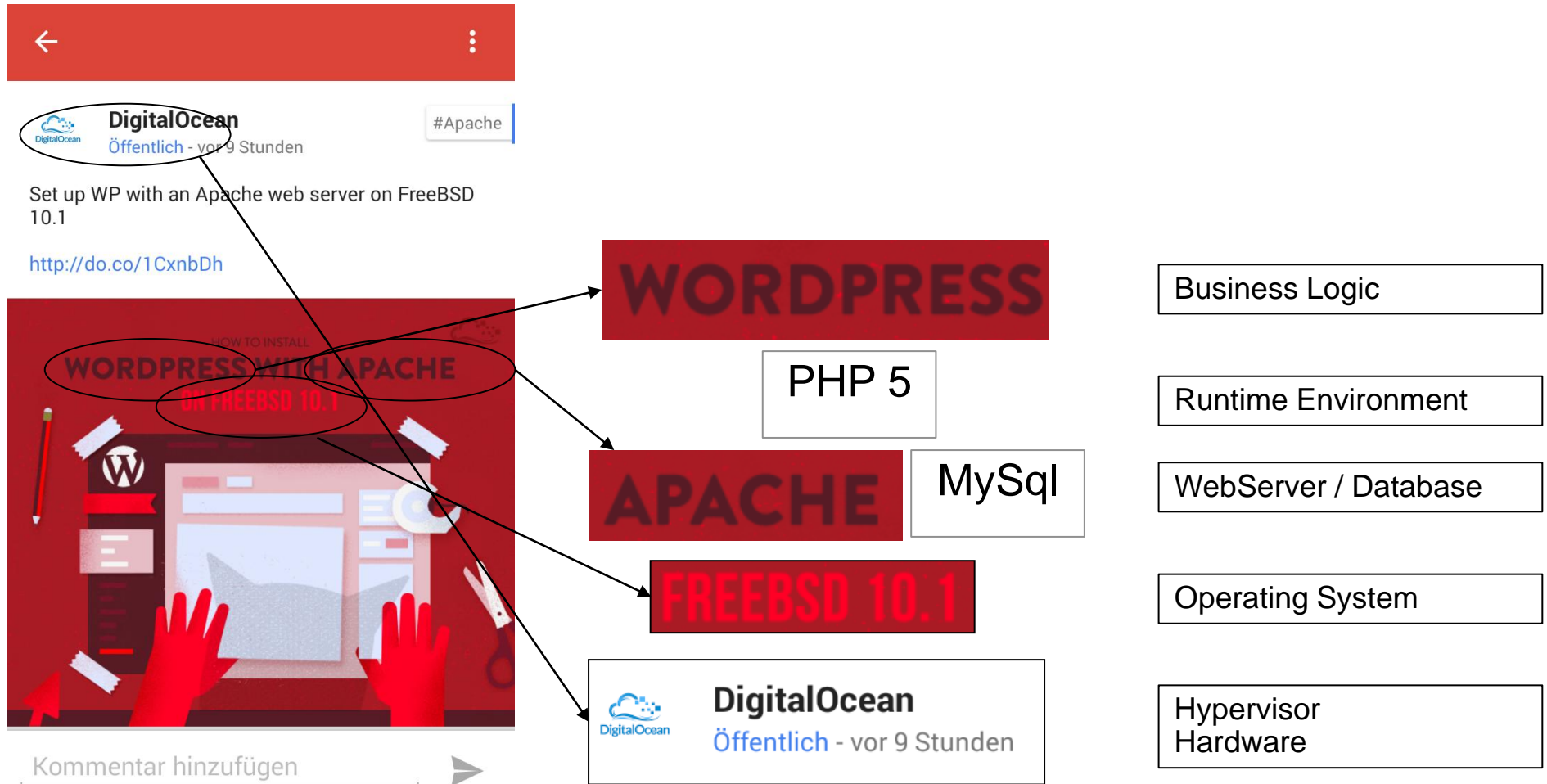
Analysis Stack

- Log aggregation
- Monitoring
 - Uptime Monitoring
 - End-user Monitoring
 - System Monitoring
 - Application Monitoring
- Threat Analysis

Characteristics of a Technology Stack



Example



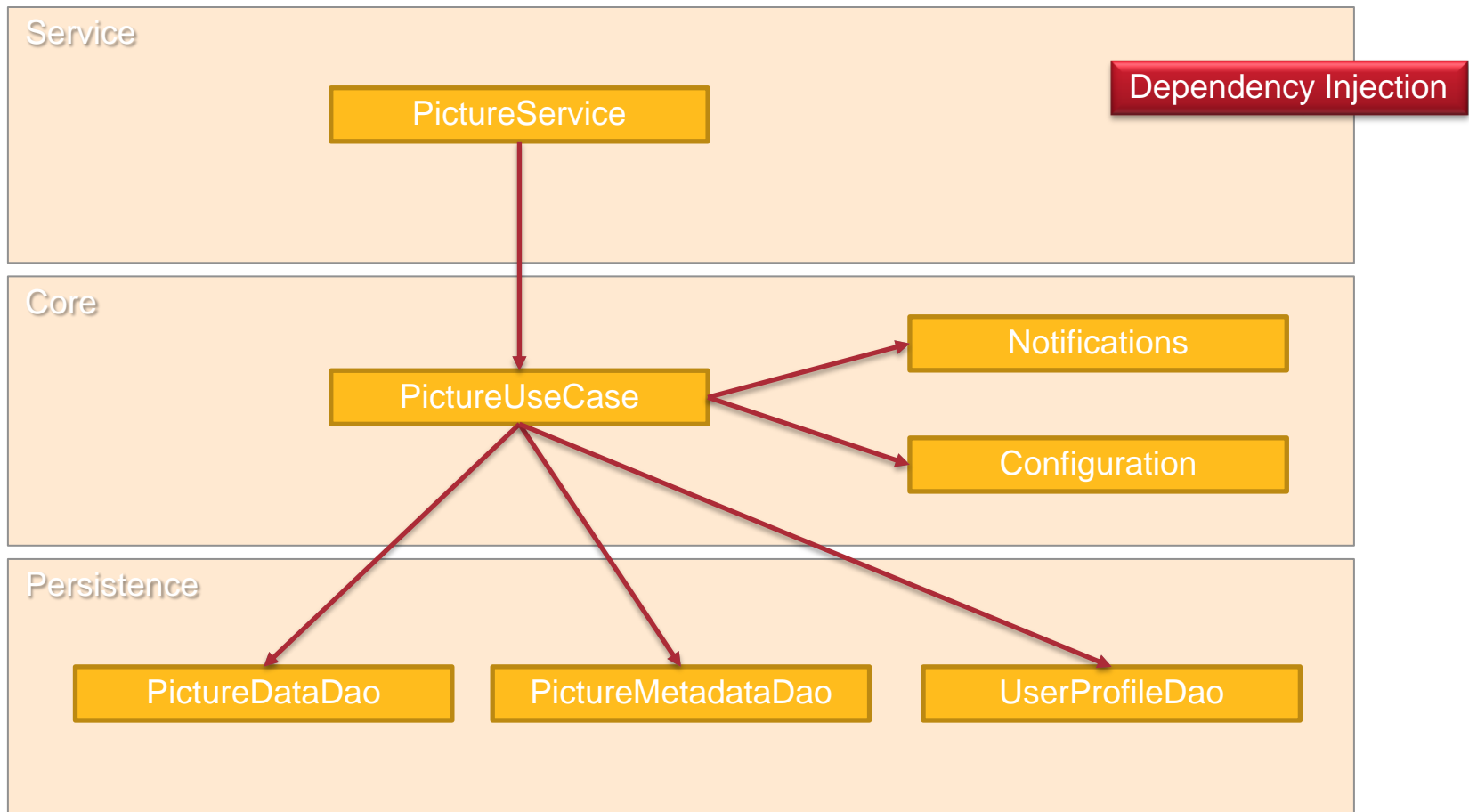
Choosing the right Technology Stack

- Productivity
- Know How / Maintainability
- Scalability
- Active Support / Community
- Licensing

Integrating your app

- Let the framework handle the infrastructure
- Integration is enabled by
 - dependency injection
 - modular design
- The Hollywood Principle
- Inversion of Control
- Avoid the new keyword

Dependency Injection: Example



Dependency Injection: Example

PictureService

```
public class PictureService {  
    private UploadPictureUseCase uploadPictureUseCase;  
  
    public void processRequest(HttpRequest request, HttpResponse response) {  
        UploadedPictureData uploadedPictureData = extractPictureFromRequest(request);  
        PictureIdentifier id = uploadPictureUseCase.uploadPicture(uploadedPictureData);  
        // write id to response ...  
    }  
}
```

How can we get the UseCase Component into the Service?

Dependency Injection: Example

1. Instantiate dependencies in the component

```
public PictureService() {  
    this.uploadPictureUseCase = new UploadPictureUseCaseImpl();  
}
```

- You cannot swap implementations -> no tests
- Changes potentially propagate through your whole codebase

Dependency Injection: Example

2. Wire dependencies programmatically

```
public static void main(String[] args) {  
  
    PictureService pictureService = new PictureService(uploadPictureUseCase);  
    //run the picture service  
}
```

- You have to manually maintain the order of dependencies
- You have to repeat the wiring of your components for your tests
- Does not scale

Dependency Injection: Example

3. Factory Pattern

```
public PictureService() {  
    this.uploadPictureUseCase = Factory.getUploadPictureUseCase();  
}
```

- Instantiation of Components is centralized -> swapping Implementations is possible
- Classes are coupled to the Factory -> not reusable

Dependency Injection: Example

4. Dependency Injection Framework (Spring as example)

Setter Injection:

```
public class PictureService {  
    private UploadPictureUseCase uploadPictureUseCase;  
  
    public void setUploadPictureUseCase(UploadPictureUseCase uploadPictureUseCase) {  
        this.uploadPictureUseCase = uploadPictureUseCase;  
    }  
}
```

Spring Configuration:

```
<bean class="PictureService">  
    <property name="uploadPictureUseCase" ref="uploadPictureUseCase" />  
</bean>  
  
<bean id="uploadPictureUseCase" class="UploadPictureUseCaseImpl">  
    <property ... />  
</bean>
```


Dependency Injection: Example

4. Dependency Injection Framework (Spring as example)

Constructor Injection:

```
public class PictureService {  
    private UploadPictureUseCase uploadPictureUseCase;  
  
    public PictureService(UploadPictureUseCase uploadPictureUseCase) {  
        this.uploadPictureUseCase = uploadPictureUseCase;  
    }  
}
```

Spring Configuration:

```
<bean class="PictureService">  
    <constructor-arg>  
        <ref bean="uploadPictureUseCase"/>  
    </constructor-arg>  
</bean>  
  
<bean id="uploadPictureUseCase" class="UploadPictureUseCaseImpl">  
    <property ... />  
</bean>
```

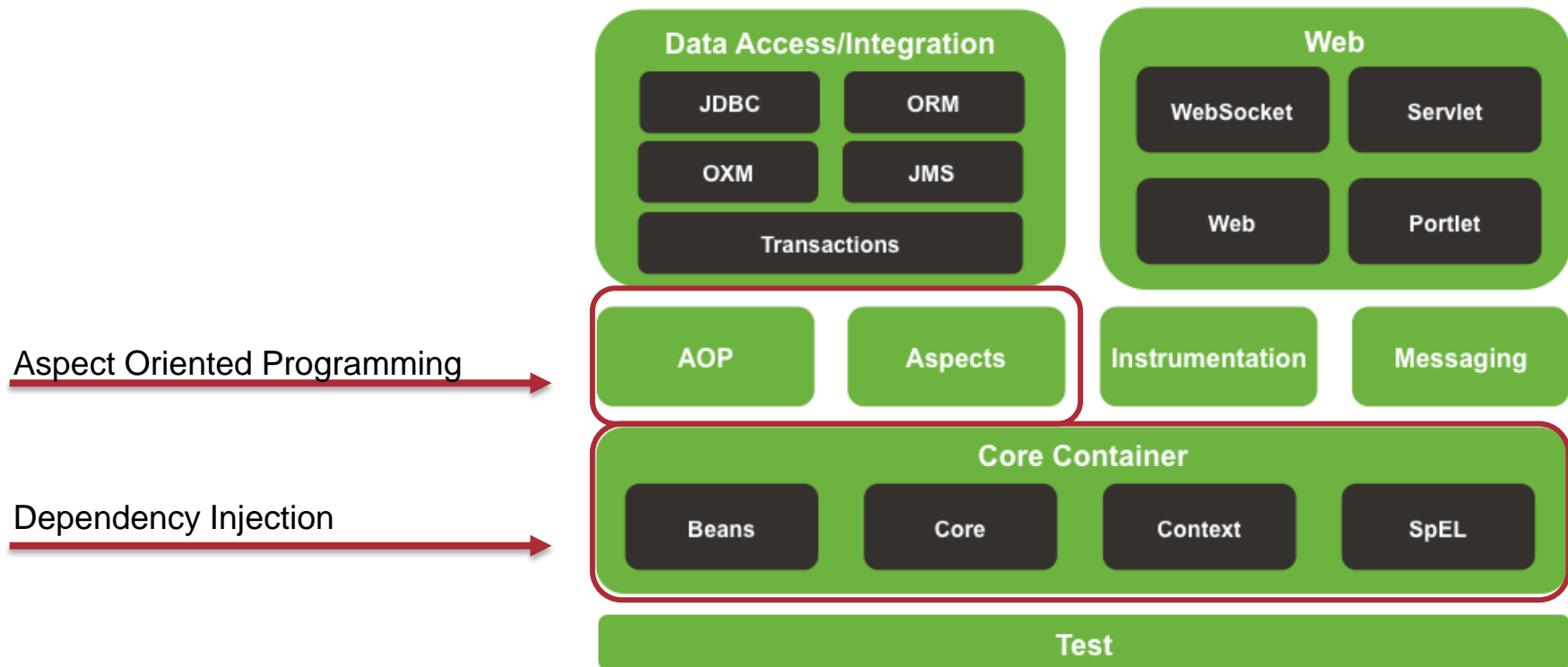
Dependency Injection

- No dependencies to the framework (or the factory)
- Non-invasive (classes do not know about the DI framework)
- Components are reusable
- Implementations can be swapped for test stubs
- Dependency description is centralized in the configuration file

But wait,... there is more



Spring Framework Runtime



Aspect Oriented Programming

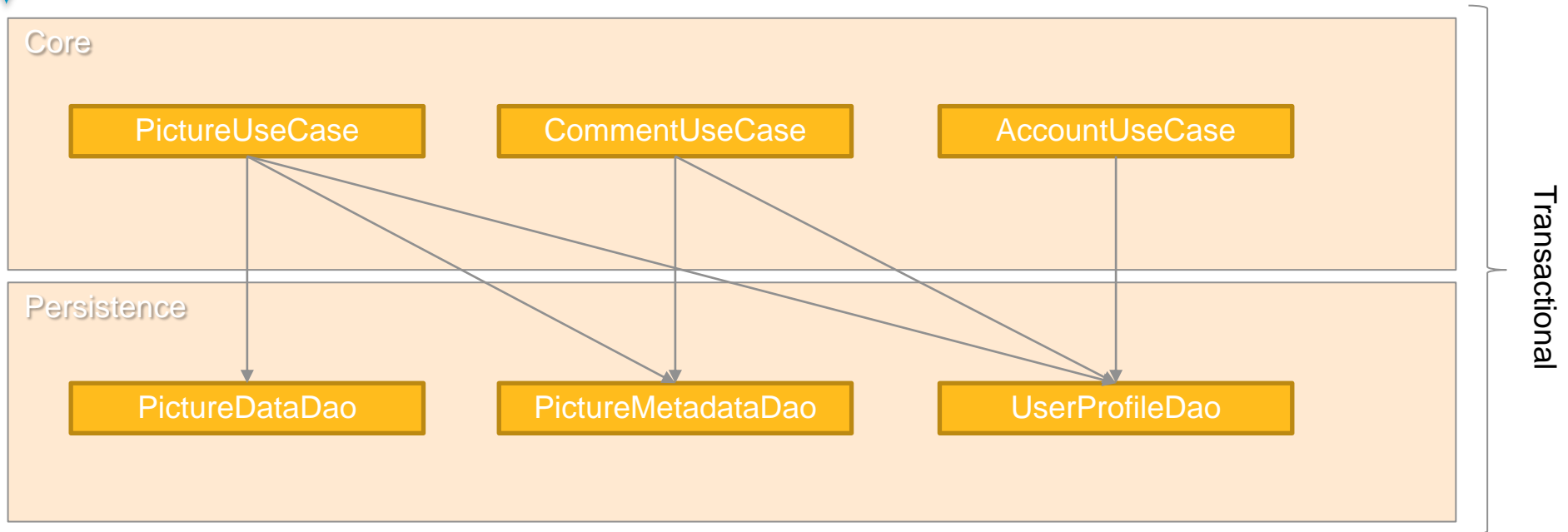
What

- Addresses Cross-Cutting-Concerns in OOP
- Encourages the Single Responsibility Principle

How

- Intercepting method calls and
 - Manipulate behavior
 - Inject new behavior
 - Suppress behavior

Example



Pure Object Oriented Programming:

Each UseCase will have to:

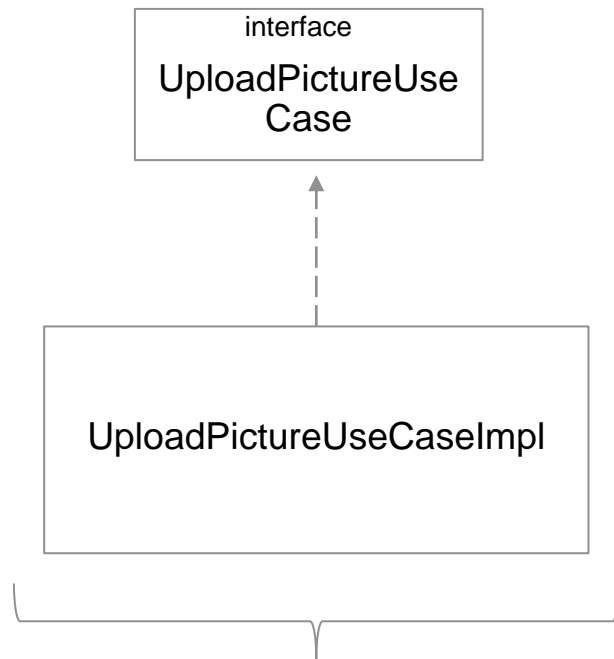
1. **Open a transaction**
2. **Perform business logic**
3. **Rollback / commit the transaction**



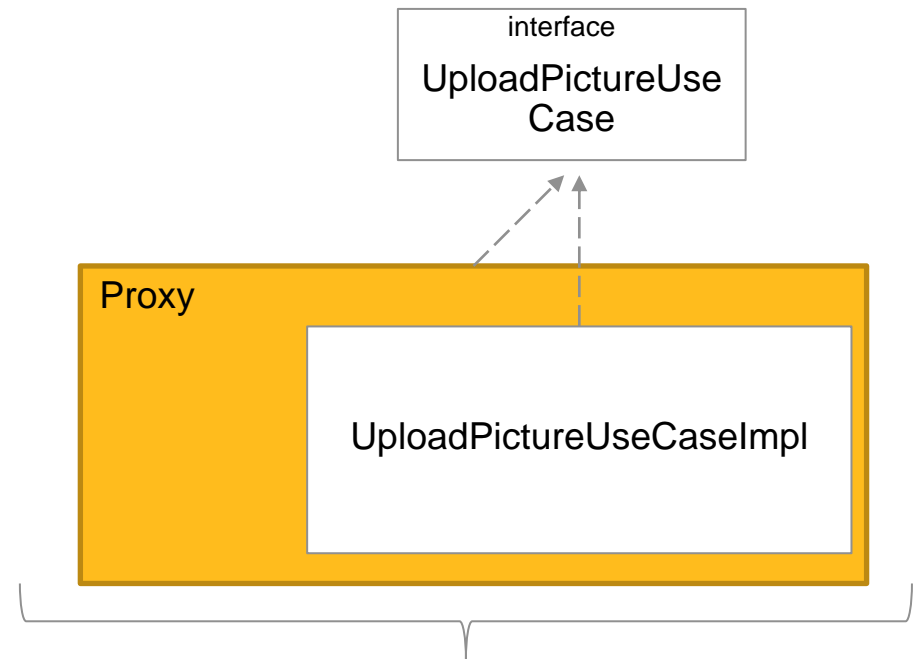
- **Transaction Management** Code is scattered around several Classes
- UseCase Components implement multiple concerns

Aspect Oriented Programming with Spring

- By using Dependency Injection we have put Spring in control of creating our objects
- This can be leveraged to achieve AOP
- Instead of the real implementation Spring returns a proxy
 - This proxy delegates to the real implementation
 - And intercepts certain method calls and adds extra functionality



Original Class Hierarchy



Object returned by Spring

Writing your own Interceptor

1. Implement the MethodInterceptor Interface

```
public Object invoke(final MethodInvocation invocation) throws Throwable {  
    TransactionInfo txInfo = createTransactionIfNecessary();  
    Object retVal = null;  
    try {  
        retVal = invocation.proceed();  
    }  
    catch (Throwable ex) {  
        rollbackTransactionAfterThrowing(txInfo, ex);  
        throw ex;  
    }  
    commitTransactionAfterReturning(txInfo);  
    return retVal;  
}
```

1. Create Transaction

2. Proceed with the original method call

3. Rollback if something Went wrong

4. Commit transaction
And return the return-value
Of the original method call

Writing your own Interceptor

2. Tell Spring when to use the interceptor

```
<bean id="transactionInterceptor" class="TransactionInterceptor" />

<aop:config>
  <aop:pointcut id="transactionalPointcut" expression="@annotation(Transactional)" />
  <aop:advisor pointcut-ref="transactionalPointcut"
              advice-ref="transactionInterceptor" />
</aop:config>
```

Pointcut: Set of Join Points

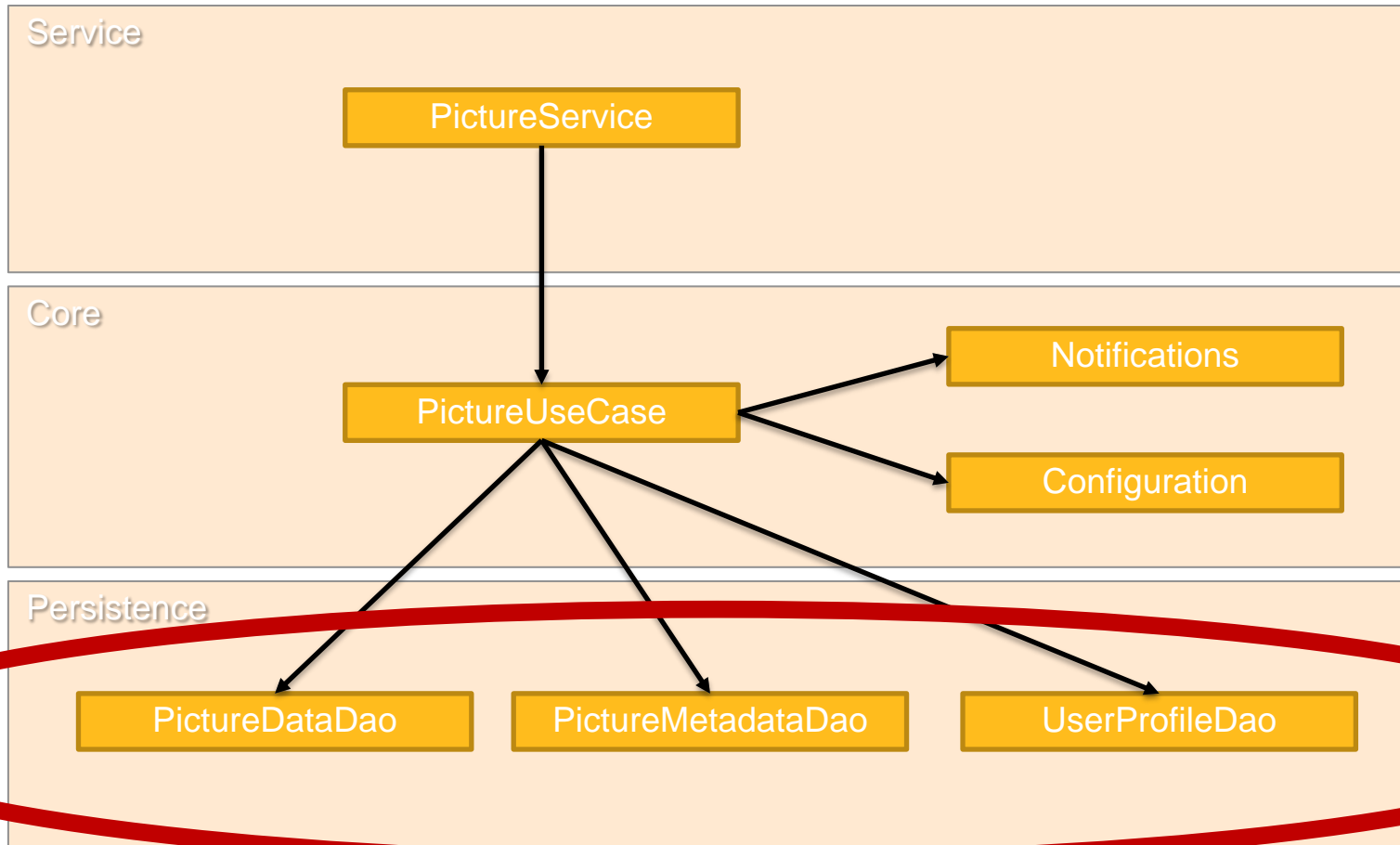
Join Point: A point in the control flow of a program

Advice: A description of how to modify the control flow

Agenda

- Introduction
- Dependency Injection
- Aspect Oriented Programming
- Object-relational mapping

Persistence layer





今帰仁城跡





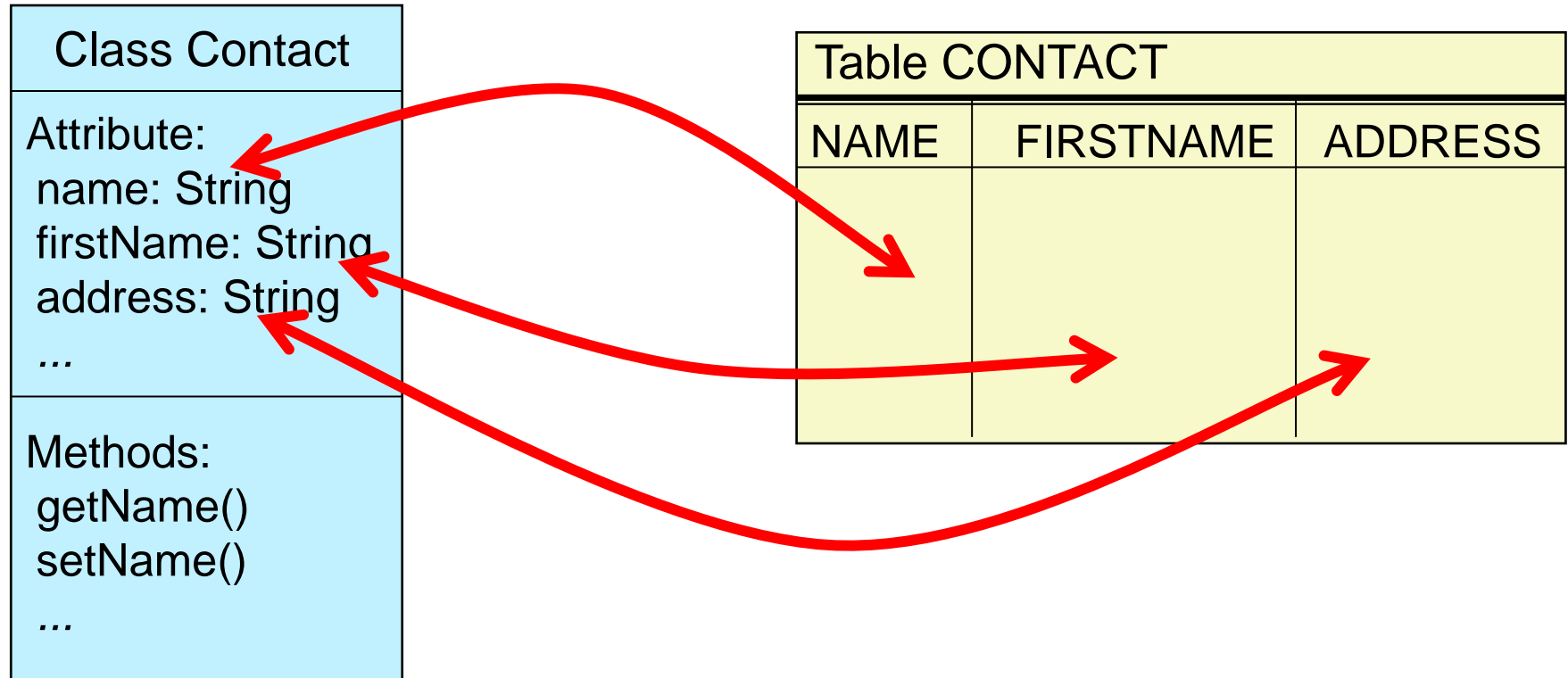




Object-relational impedance mismatch



Object-relational mapping (ORM)





Hibernate - well-established O/R-Mapper for Java

Characteristics

Supports many RDBMS

Easy to use

High performance

Open Source

Headline

Oracle, PostgreSQL, MySQL, DB2, Sybase, MS SQL Server, MS Access, Apache Derby, HypersonicSQL, Mckoi SQL, SAP DB, Interbase, Pointbase, Progress, FrontBase, Ingres, Informix, Firebird

- Working with natural objects and features (inheritance, associations, etc.)
- Hibernate query language easy to use
- Lazy initialization/ loading with different fetching strategies
- Generates much SQL at system startup instead of runtime
- Hibernate is Open Source
- Can use other Open Source Frameworks for
 - Caching (e.g. EHCACHE, OSCache, SwarmCache, JBoss-Cache)
 - Connection Pooling (e.g. C3P0, Proxool, Commons DBCP)

Components of Hibernate

Components

Hibernate ORM

- Core functionalities for OR-Mapping
- Java-Annotations for defining mapping model
- Hibernate Query Language (HQL)

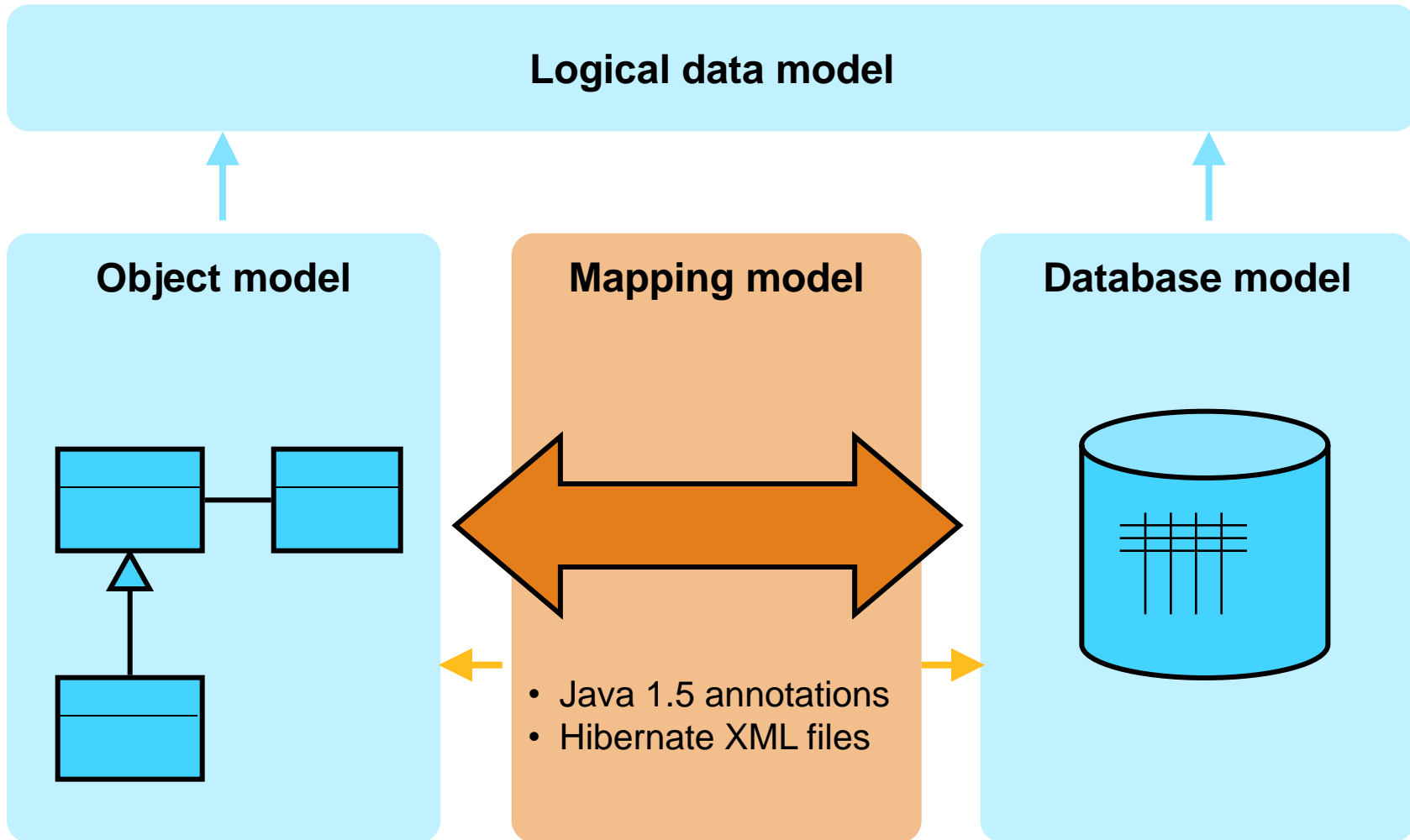
Hibernate EntityManager

- Wrapper around Hibernate ORM
- Implements Java Persistence API (JPA)
- Java Persistence Query Language (JPQL) instead of HQL



More components

- Hibernate Search: full-text search with Apache Lucene
- Hibernate Validator: reference implementation of Bean Validation
- Hibernate OGM (Object/Grid Mapper): for NoSQL Databases
- Many more

Mapping between objects and database



Basic principles for mapping

	Object model		Database model
Data	Class		Table
	Attribute		Column
Structure	Relationen		Foreign key
	Inheritance		Relational model
Meta data	Primary key Loading Strategies Cascading		

Mapping with Hibernate

	Hibernate XML approach	Java 5 annotations (JPA/Hibernate)
Type	<code><class name=„Cd“ table=„CD“></code>	<code>@Entity</code> <code>@Table(name = "CD")</code> public class Cd {...}
Attribute	<code><property name= „lyrics“</code> <code>column=„LYRICS“ type=„clob“/></code>	<code>@Column(name = „LYRICS“)</code> <code>@Lob</code> public String getLyrics() {...}
Relation	<code><set name=“tracks“ cascade=“all“></code> <code><key column=“CD_ISBN“/></code> <code><one-to-many class=“de..Track“/></code> <code></set></code>	<code>@OneToMany(cascade = CascadeType.ALL)</code> <code>@JoinColumn(name = "CD_ISBN")</code> public Set<Track> getTracks() {...}

Mapping via Java 5 annotations

```
@Entity
@Table(name = "PERSON")
public class Person {

    private Integer id;
    private String name;

    @Id
    @GeneratedValue
    public Integer getId() {
        return id;
    }

    @NotNull
    @Column(length = 40)
    public String getName() {
        return name;
    }
}
```


Some more annotations...

```
[...]
private Date birthday;
private Set<Thing> ownedThings;

@NotNull
@Temporal(TemporalType.DATE)
@Column(updatable = false)
public Date getBirthdate() {
    return birthday;
}

@OneToMany(fetch = FetchType.LAZY, mappedBy = "owner")
@OrderBy("name ASC")
public Set<Thing> getOwnedThings() {
    return ownedThings;
}
[...]
```

Mapping via Hibernate XML files

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="example">

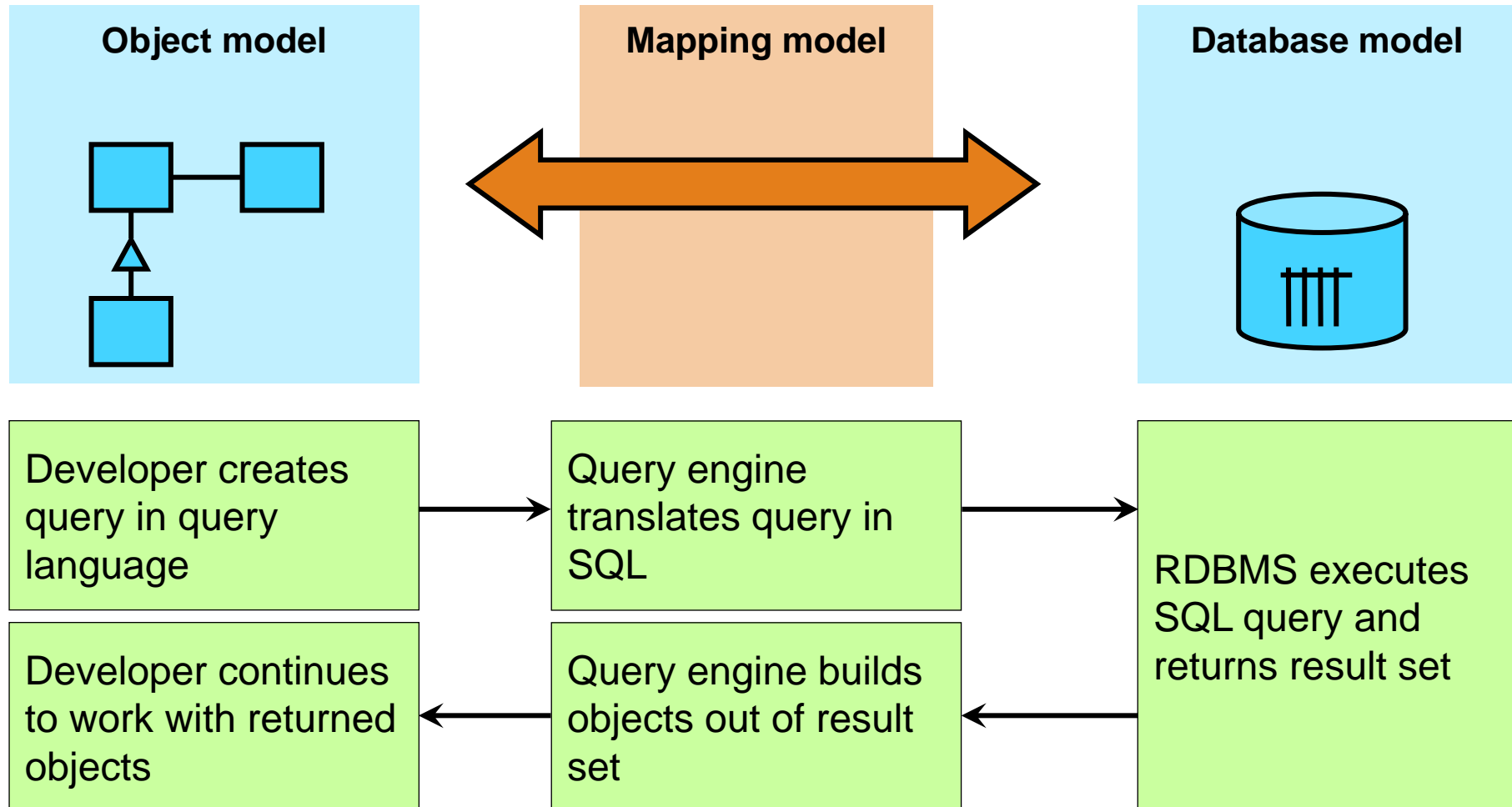
    <class name="Person" table="PERSON">
        <id name="id">
            <generator class="native" />
        </id>
        <property name="name" type="string" length="40" not-null="true" />
        <property name="birthdate" type="date" not-null="true" update="false" />
        <set name="ownedThings" inverse="true" lazy="true" order-by="name ASC">
            <key column="owner" />
            <one-to-many class="Thing" />
        </set>
    </class>

</hibernate-mapping>
```

(Dis-) Advantages of mapping definitions

	advantages	disadvantages
Annotations	<ul style="list-style-type: none">+ compact+ easy to maintain (refactoring etc.)+ usage of JPA standard+ compiled java byte code→ faster than xml files	<ul style="list-style-type: none">– binary dependency– configuration is „widespreaded“
XML files	<ul style="list-style-type: none">+ single point of configuration+ easy to modify without software re-compilation/rebuilding	<ul style="list-style-type: none">– poor readability, especially in huge applications

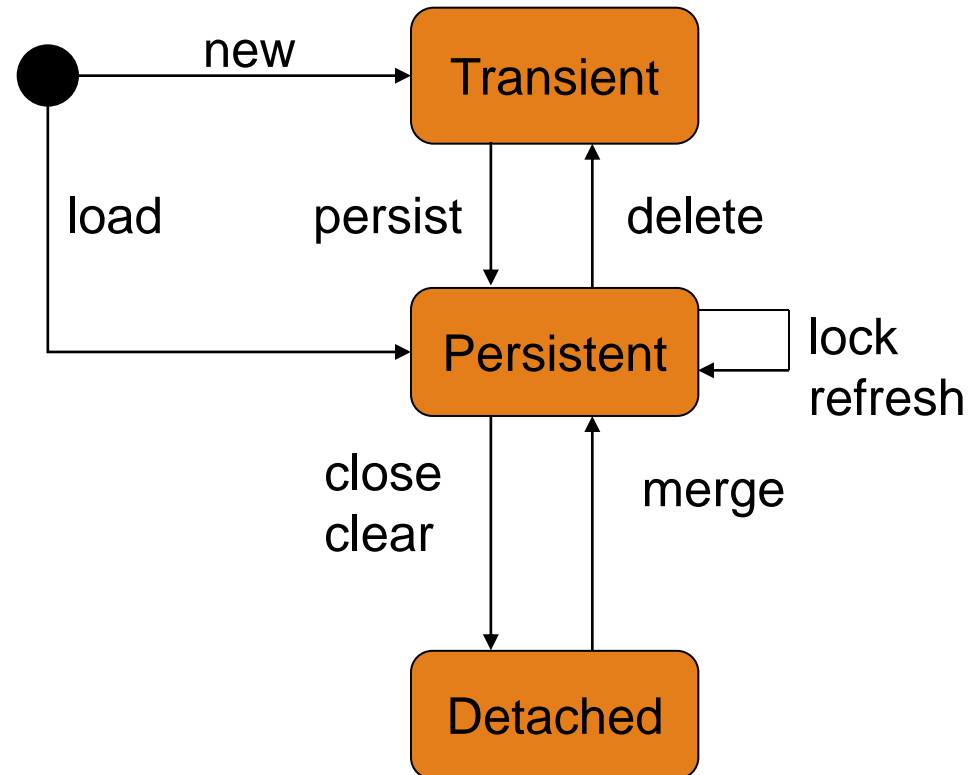
Basic idea of querying with O/R-Mapper



Working with Hibernate objects

Session-operations:

- **load**: loads object with primary key
- **persist**: saves new object
- **delete**: deletes object
- **refresh**: reloads object
- **Close/clear**: ends session
- **merge**: syncs object with database (reattach)



Querying methods in Hibernate

Hibernate Query Language (HQL)

Criteria API

Native SQL queries

Features

- works on persistence objects
 - fully object-oriented: understands notions like inheritance, polymorphism and association
 - database joins modeled via associations of persistence objects
 - returns persistence objects
-
- query against persistence class
 - adding constraints/restrictions (criteria) that narrow results
 - object-based API
-
- pure SQL, (only) needed for DBMS specifics like query hints or performance tuning
 - returns array of java objects
 - mapping to (persistence) objects manually (or, if table/column-names equals setter, automatically)

Examples of Hibernate Query Language (HQL)

```
Session session = sessions.openSession(); // open a new hibernate session
```

```
Query q1 = session.createQuery("from Cat cat where cat.name = :name");  
q1.setString("name", "Fritz");
```

```
List<String> names = new ArrayList<>();  
names.add("Izi");  
names.add("Fritz");  
Query q2 = session.createQuery("from Cat cat where cat.name in (:namesList)");  
q2.setParameterList("namesList", names);
```

```
Date date = Calendar.getInstance().getTime();  
Query q3 = session.createQuery("from Person as pers where pers.birthdate < ?")  
    .setDate(0, date);
```

```
Query q4 = session.createQuery("select h.color from House h where h.address.city in  
(:cityList)").setParameterList("cityList", cities);
```

```
List results = q4.list(); // obtain results from database
```

Examples of Criteria API

```
Session session = sessions.openSession(); // open a new hibernate session

Criteria criteria = session.createCriteria(House.class);
criteria.setMaxResults(50)
    .addOrder( Order.asc("number") )
    .add( Restrictions.eq("color", "red") )
    .createCriteria("address")
        .add( Restrictions.in("city", new String[] {"Frankfurt", "Darmstadt"}) );

List houses = criteria.list(); // obtain results from database
```


Examples of native SQL

```
Session session = sessions.openSession(); // open a new hibernate session
```

```
// Standard SQL query
```

```
Query q1 = session.createQuery("select h.color from house h, address a where h.id = a.id  
and a.city in ('Frankfurt', 'Darmstadt')");
```

```
// Mapping via hibernate entity
```

```
Query q2 = session.createQuery("select * from house where color like '%red%'")  
.addEntity(House.class);
```

Using named queries

1. Annotation at persistence class:

```
@Entity
@NamedQuery(name = "cat.byNameAndMaximumWeight", query = "from Cat cat where
    cat.name = :name and cat.weight > :weight")

public class Cat {
    [...]
}
```

2. External named query file:

```
<query name="cat.byNameAndMaximumWeight"><![CDATA[
    from example.Cat as cat
    where cat.name = :name
    and cat.weight > :weight
]]>
</query>
```

Resulting query definition in Java:

```
List results = session.getNamedQuery("cat.byNameAndMaximumWeight")
    .setString("name", name).setFloat("weight", weight).list();
```

Pros and Cons of different query methods

method	advantages	disadvantages
HQL	<ul style="list-style-type: none">+ database independent (including SQL dialects)+ comfortable attribute navigation (joins)	<ul style="list-style-type: none">– no standard– „tinkering“ with dynamic SQL queries
Criteria API	<ul style="list-style-type: none">+ (dynamic) creation via clear API+ „query by example“	<ul style="list-style-type: none">– hard to estimate complexity of SQL query
SQL	<ul style="list-style-type: none">+ clear control of SQL query+ use of database-specific functions	<ul style="list-style-type: none">– database dependent

Transaction management

- Transaction is sequence of instructions that are only reasonable together
- fulfill ACID properties:
 - Atomicity: "all or nothing"
 - Consistency: ensures that any transaction will bring database from one valid state to another
 - Isolation: ensures that concurrent execution of transactions results in system state that would be obtained if transactions were executed serially
 - Durability: transaction has been committed
- Definition of transaction demarcation (boundaries) in applications necessary for database communication
- Two types of demarcation:
 - Unmanaged
 - Managed

Example of unmanaged transaction demarcation

```
Session session = sessions.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // do some work
    tx.commit();
} catch (Throwable e) {
    if (tx != null) {
        tx.rollback();
    }
    throw e;
} finally {
    session.close();
}
```

Examples of managed transaction demarcation

most common pattern in a multi-user client/server application is session-per-request:

- ServletFilter
- AOP interceptor with a pointcut on the service methods
- proxy/interception container

Concurrency control

"The most important point about Hibernate and concurrency control is that it is easy to understand." --Hibernate doc

Optimistic concurrency control

- approach for high concurrency and high scalability
- Optimistic locking assumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data.
- automatic versioning (version numbers, timestamps, to detect conflicting updates and to prevent lost updates)
- exceptions, if conflicts occur

Pessimistic locking

- Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.
- hibernate do not lock his own persistence objects, uses only db-locking mechanisms
- eg. `Per Query.setLockMode()`. Modes:
write, `PESSIMISTIC_WRITE`, `upgrade_nowait`, read, none

The End

Happy Holidays

next lecture: 15th January 2016

People matter, results count.

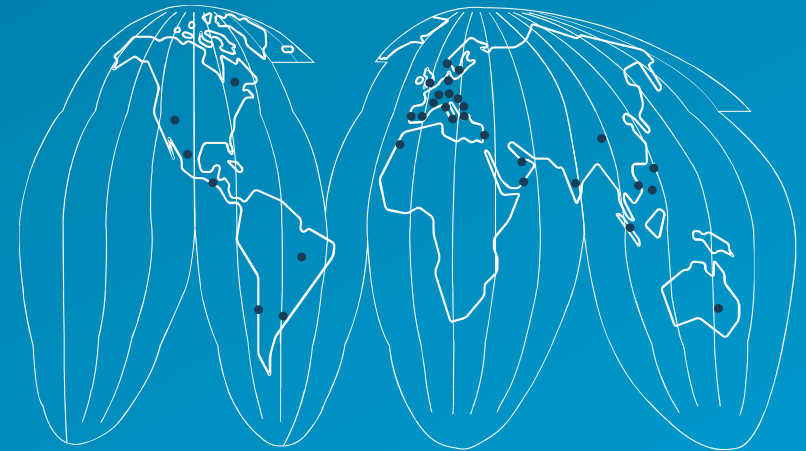


About Capgemini

With more than 130,000 people in 44 countries, Capgemini is one of the world's foremost providers of consulting, technology and outsourcing services. The Group reported 2012 global revenues of EUR 10.3 billion.

Together with its clients, Capgemini creates and delivers business and technology solutions that fit their needs and drive the results they want. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.

Rightshore® is a trademark belonging to Capgemini



www.capgemini.com



Research and science live on the exchange of ideas, the clear arrangements are thereby useful:

The content of this presentation (texts, images, photos, logos etc.) as well as the presentation are copyright protected. All rights belong to Capgemini, unless otherwise noted.

Capgemini expressly permits the public access to presentation parts for non-commercial science and research purposes.

Any further use requires explicit written permission von Capgemini.

Disclaimer:

Although this presentation and the related results were created carefully and to the best of author's knowledge, neither Capgemini nor the author will accept any liability for it's usage.

If you have any questions, please contact:

Capgemini | Offenbach

Dr. Martin Girschick

Berliner Straße 76, 63065 Offenbach, Germany

Telephone +49 69 9515-2376

Email: martin.girschick@capgemini.com