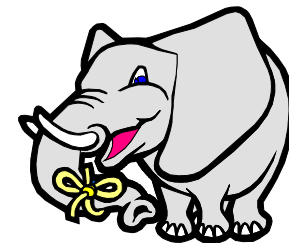
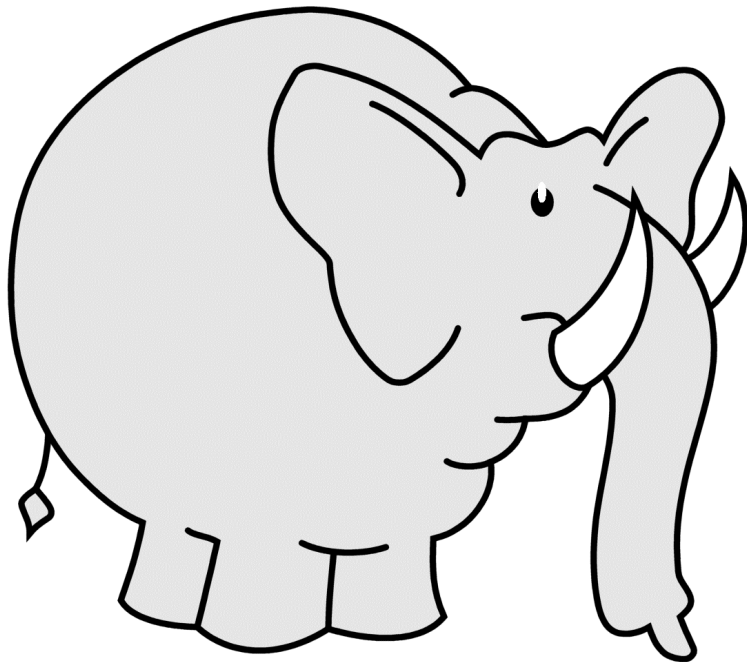


Faster, lighter, better?

Spring time for J2EE

Björn Beskow
Peter Norrhall



The Problem

- Traditional EJB systems are complex
 - Hard to develop
 - Difficult to test
- Not all systems needs all that power (distributed transactions, remoting, load balancing, fail over, ...)



Spring to the rescue!

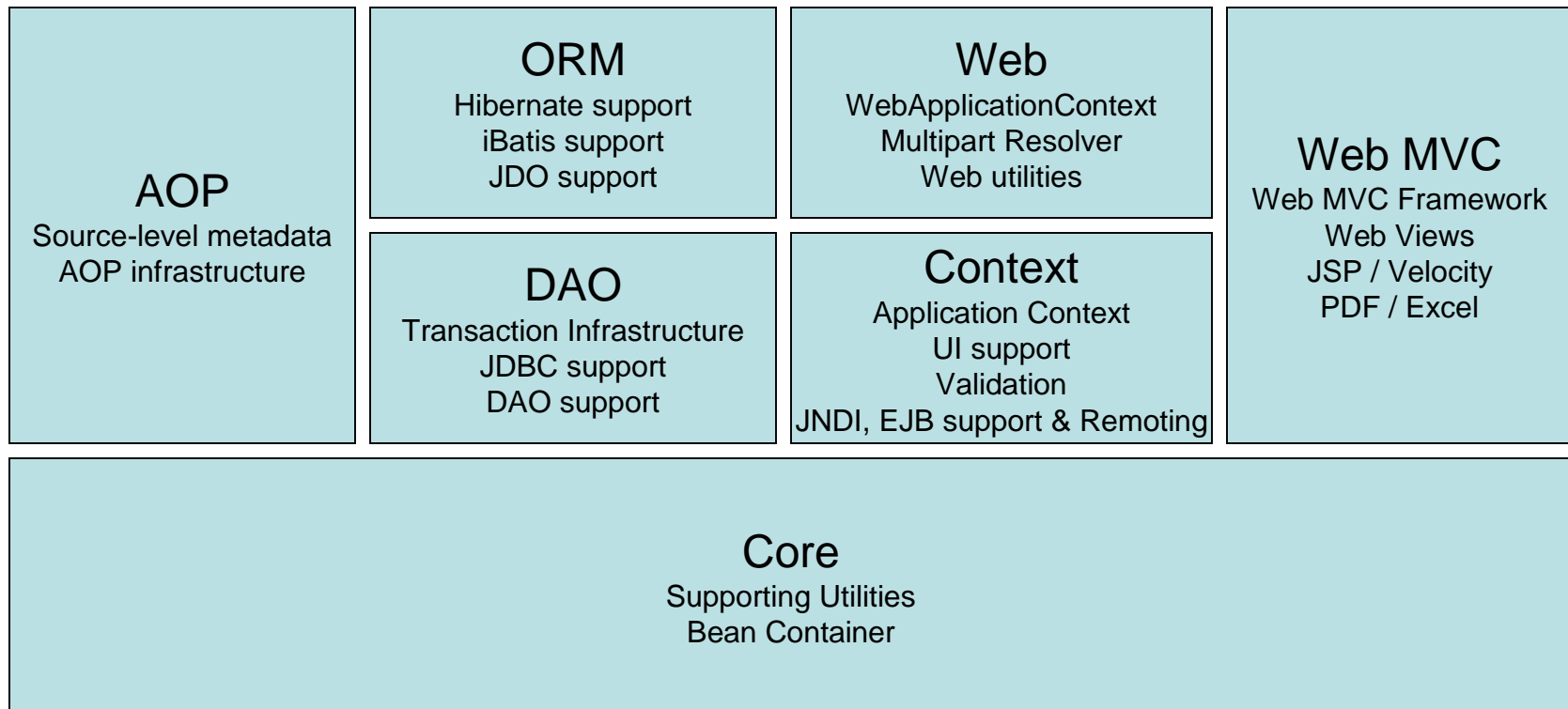


- Open Source framework by Rod Johnson & Jürgen Höller
- Version 1.0 released Q1 2004
- Current version is 1.1.3

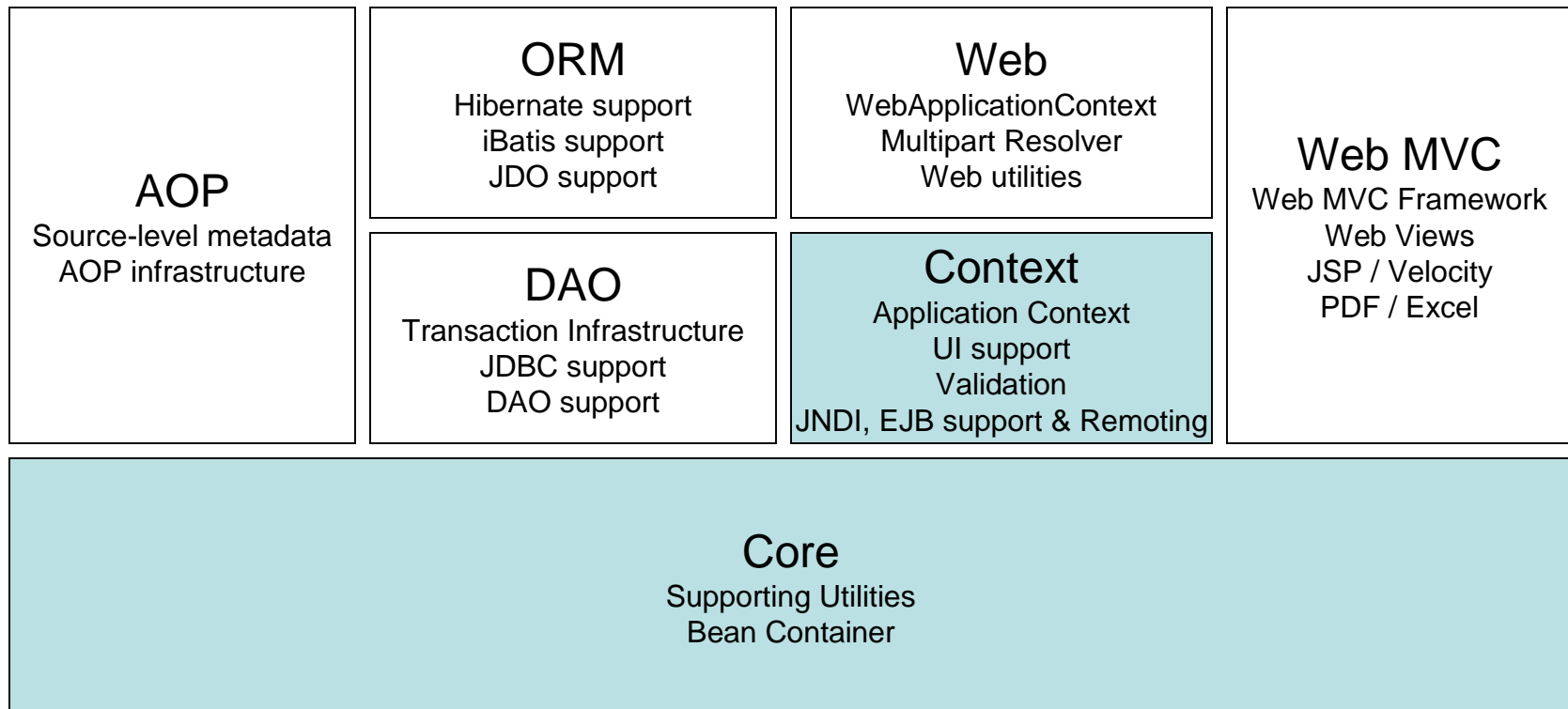
Spring Objectives

- To make J2EE easier to use
- To make middleware services available for ordinary Java objects
- To make deployment in different setups easy (Application server, web application, batch, Swing application, ...)
- To make test-driven development easier

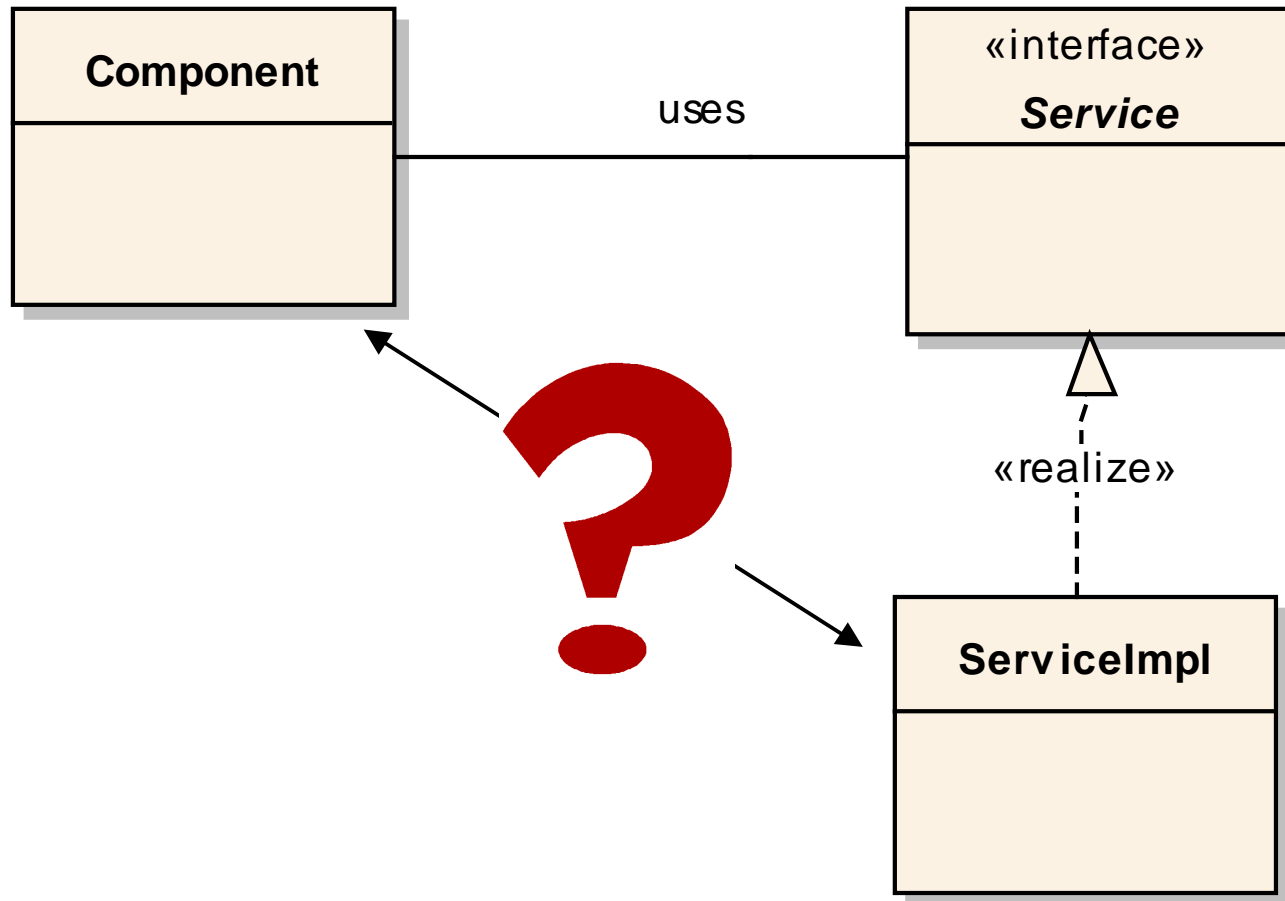
Spring Overview



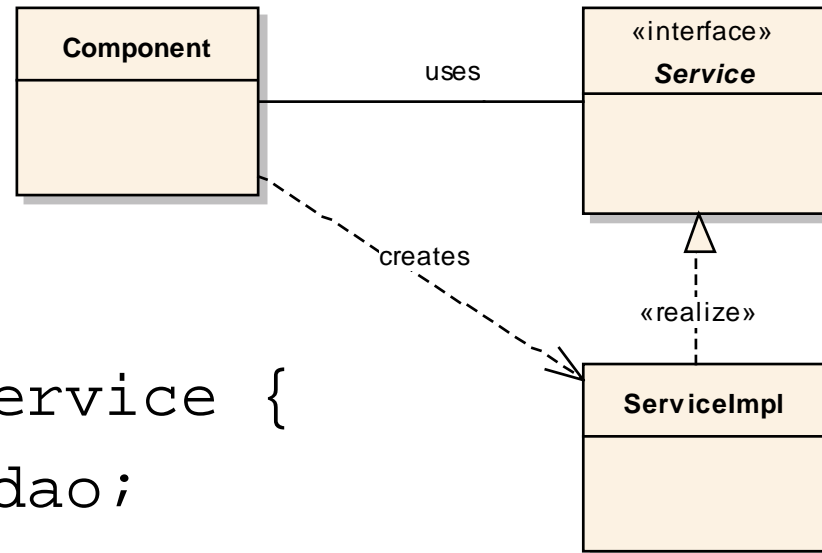
Spring Core: Configuration



Managing Dependencies



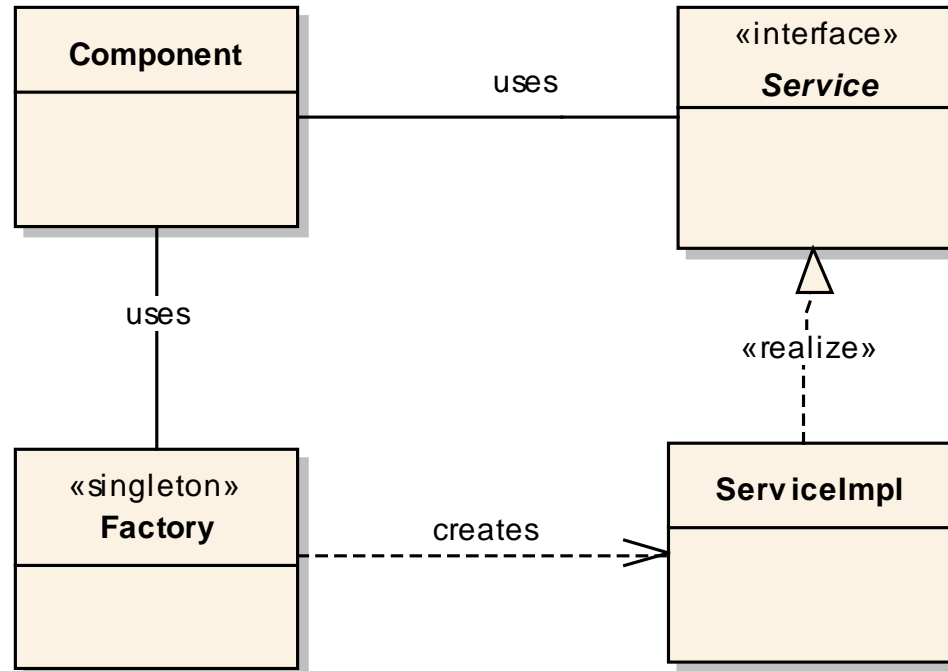
Tight Coupling (ouch!)



```
public class CustomerService {
    private CustomerDao dao;

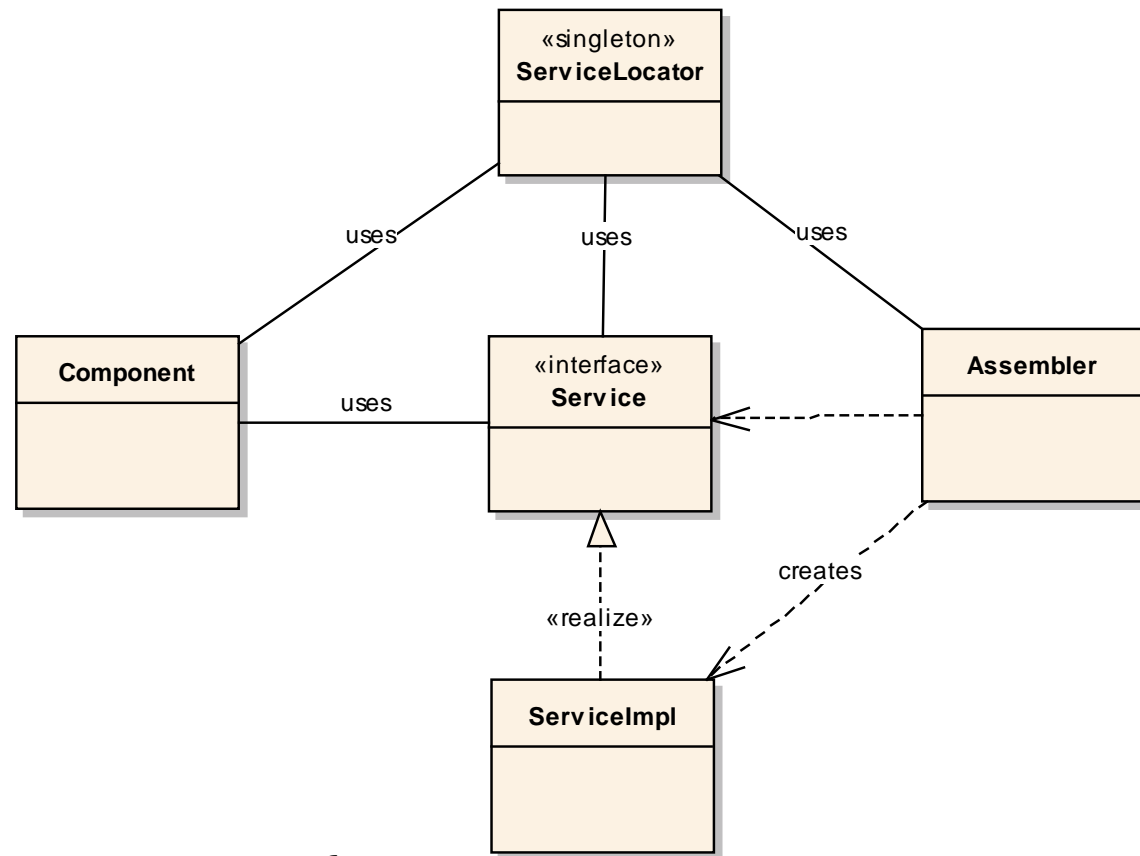
    public CustomerService () {
        dao = new CustomerDaoImpl();
    }
}
```

Factory abstraction



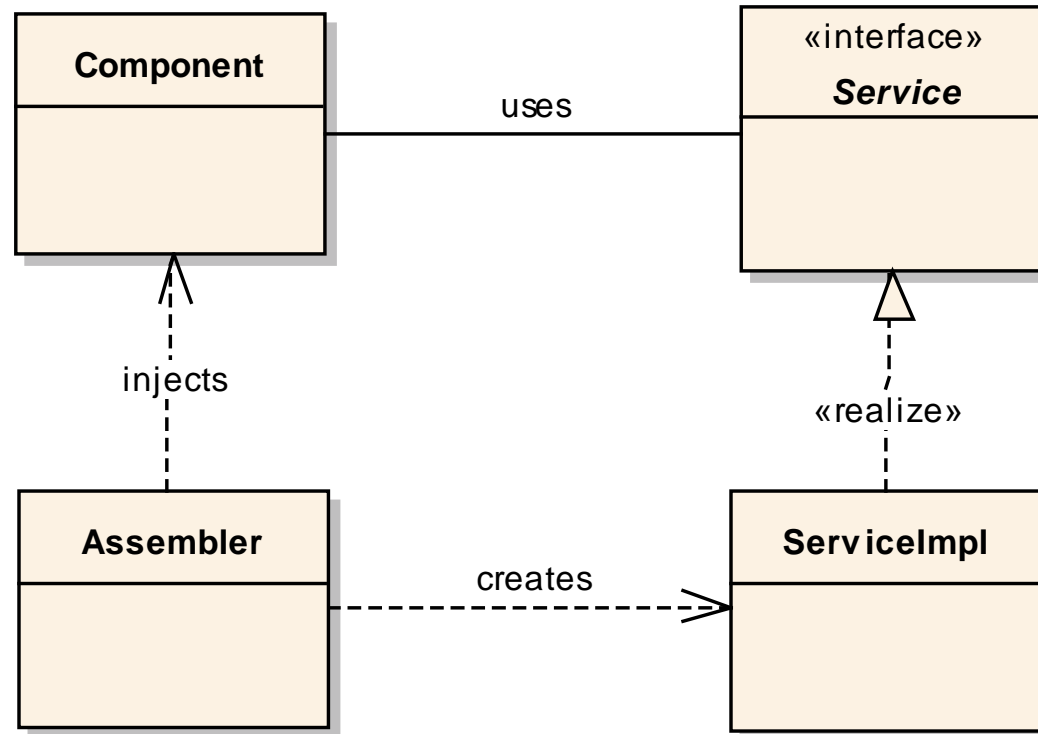
```
public CustomerService () {
    dao = CustomerDaoFactory.getCustomerDao();
}
```

Service Locator



```
public CustomerService () {
    dao = ServiceLocator.getService(CustomerDao.class);
}
```

Dependency Injection

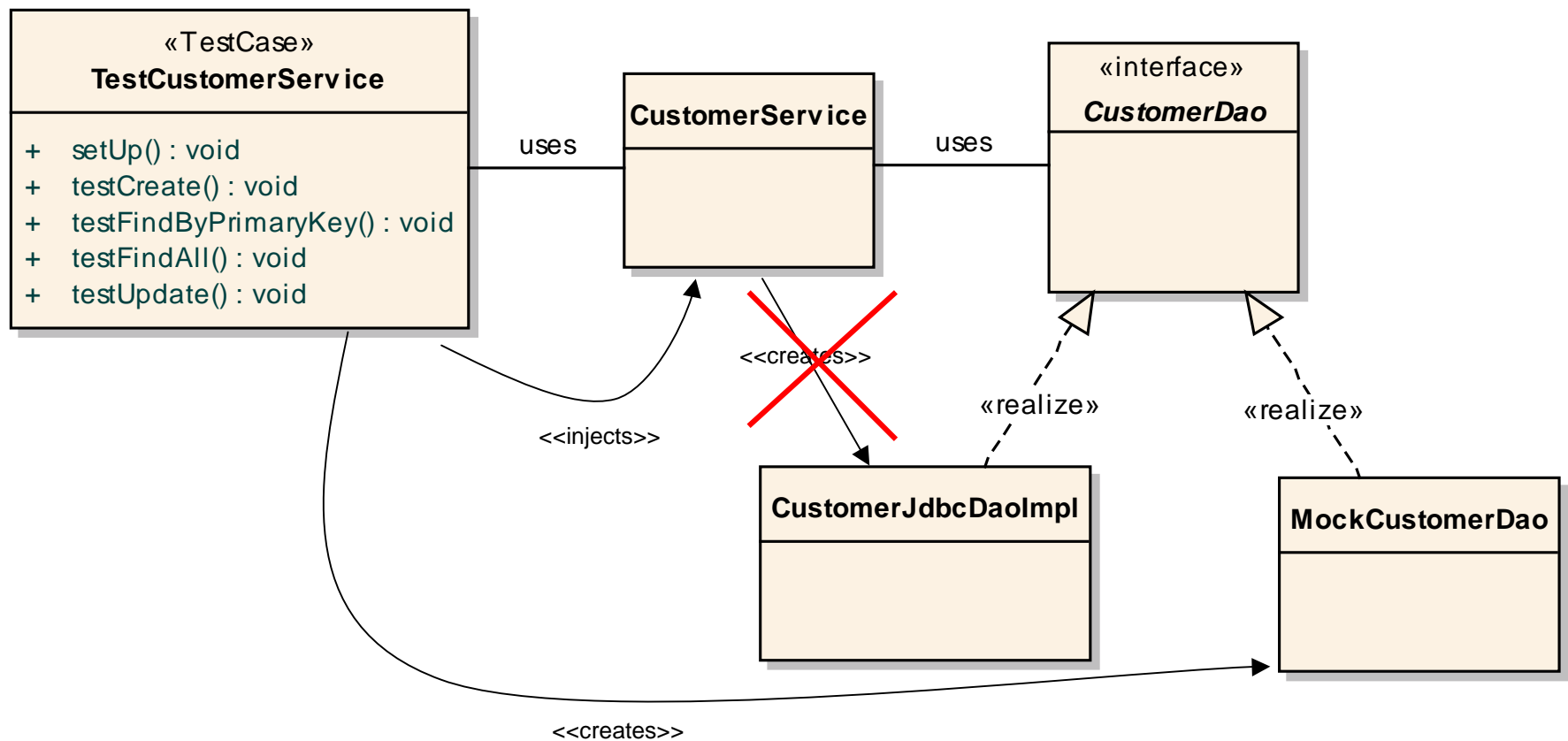


```
public void setDao(CustomerDao dao) {
    this.dao = dao;
}
```

Setter Injection

```
public class CustomerService {  
    private CustomerDao dao;  
  
    public void setDao(CustomerDao dao) {  
        this.dao = dao;  
    }  
}
```

Prime motivator: Testability

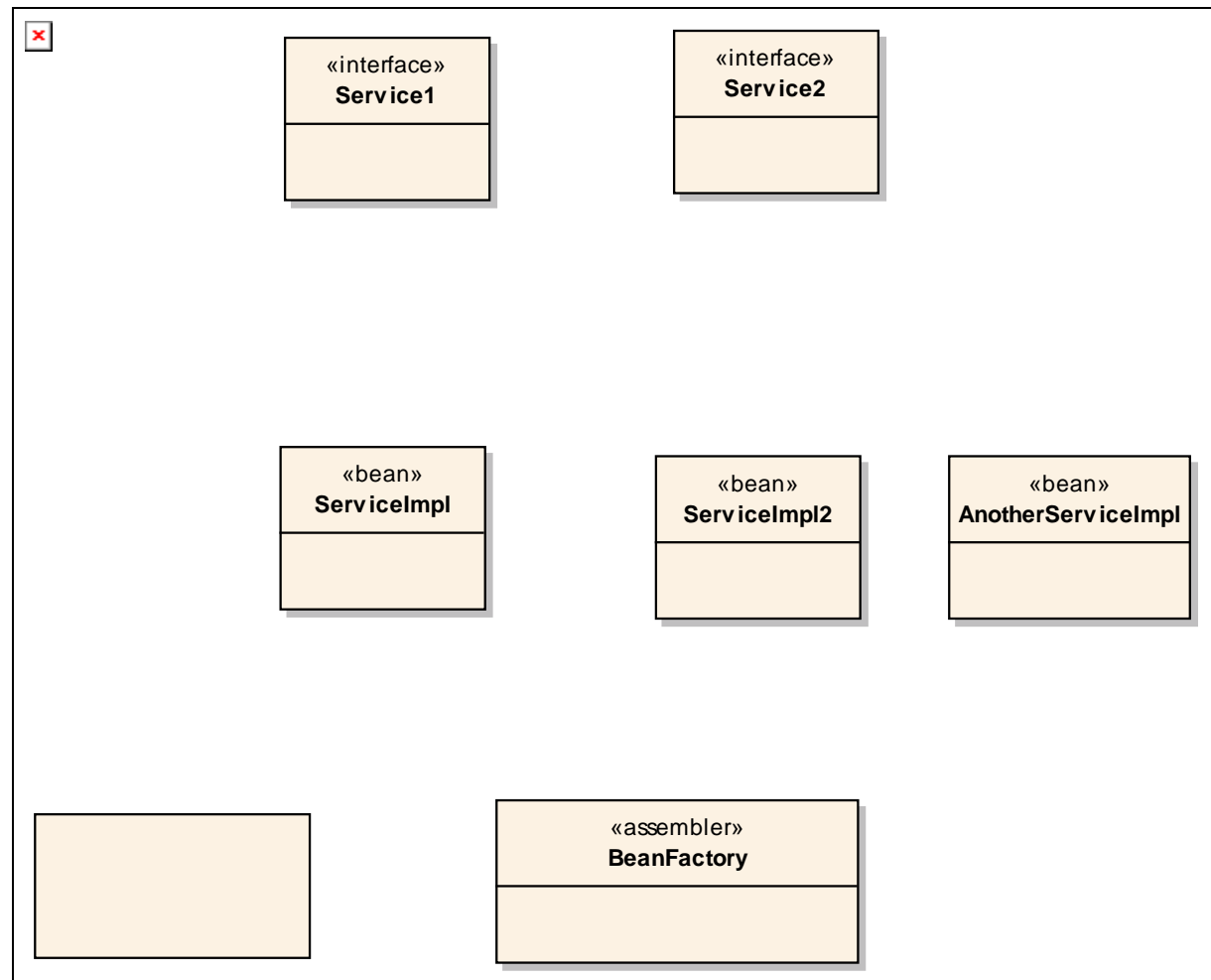


Injecting Test Objects

```
public class TestCustomerService {  
  
    public void testCreateCustomer() {  
        customerService = new CustomerService();  
        dao = new MockCustomerDao();  
        customerService.setDao(dao);  
        ...  
    }  
}
```

Spring "IoC" Container

- Lifecycle management
- Lookup
- Configuration
- Dependency resolution



XML-based Configuration

```
<?xml version="1.0"?>
```

```
<beans>
```

```
  <bean id="customerDao"  
        class="se.callista.store.dao.CustDaoImpl" />
```

```
</beans>
```

Unique name of the "bean"

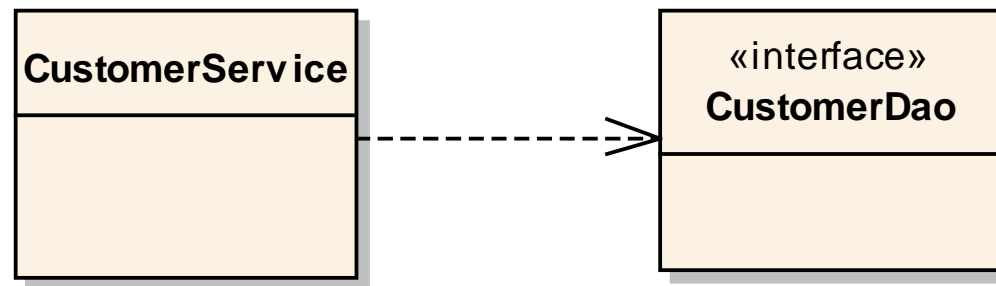


Implementation class



Expressing Dependencis

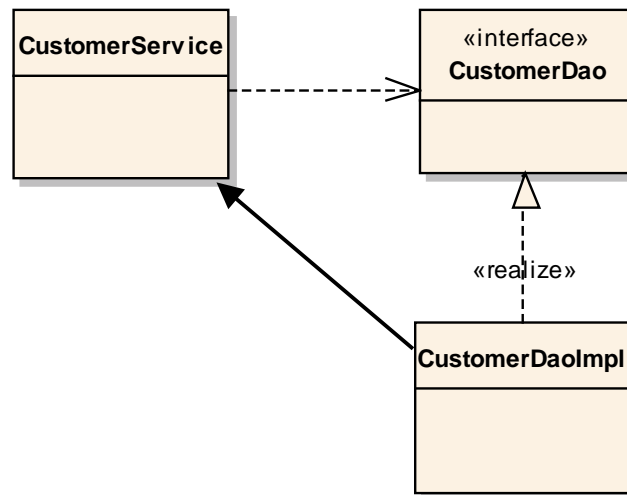
```
public class CustomerService {  
    private CustomerDao dao;  
  
    public void setDao(CustomerDao dao) {  
        dao = dao;  
    }  
}
```



Configuring dependencies

```
<bean id="customerDao"  
      class="se.callista.store.dao.CustomerDaoImpl" />  
  
<bean id="customerService"  
      class="se.callista.store.CustomerService">  
  <property name="dao">  
    <ref bean="customerDao" />  
  </property>  
</bean>
```

Wire the DAO implementation to the service "dao" property



Properties-based Configuration

```
customerDao.class = se.callista.store.CustomerDaoImpl
```

```
customerService.class = se.callista.store.CustomerService
```

```
customerService.dao = customerDao
```

Simpler, but less powerful

Application configuration

```
<bean id="dataSource" class="...SomeDataSourceImpl">
```

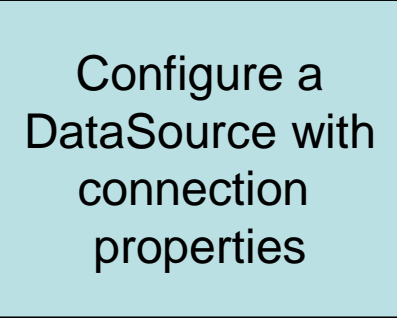
```
  <property name="driverClassName">  
    <value>org.hsqldb.jdbcDriver</value>  
  </property>
```

```
  <property name="url">  
    <value>jdbc:hsqldb:file:store</value>  
  </property>
```

```
  <property name="username"><value>sa</value></property>
```

```
  <property name="password"><value></value></property>
```

```
</bean>
```



Configure a
DataSource with
connection
properties

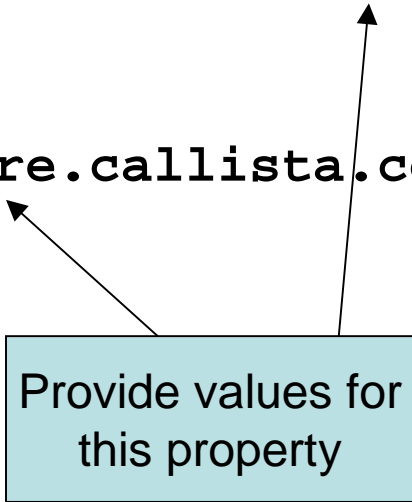
Complex properties

```
public class CustomerService {  
    private Map customerSupport;  
  
    public void setCustomerSupport(Map customerSupport)  
    {  
        this.customerSupport = customerSupport;  
    }  
}
```

This component expresses a need for a map of email addresses to customer support in different countries

Configuring complex properties

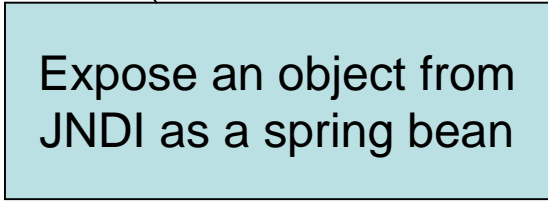
```
<bean id="customerService"  
      class="se.callista.store.CustomerService">  
  <property name="customerSupport">  
    <map>  
      <entry key="sv_SE" >  
        <value>support@store.callista.se</value>  
      </entry>  
      <entry key="en_US" >  
        <value>support@store.callista.com</value>  
      </entry>  
    </map>  
  </property>  
</bean>
```



Provide values for this property

Accessing EJBs and JNDI

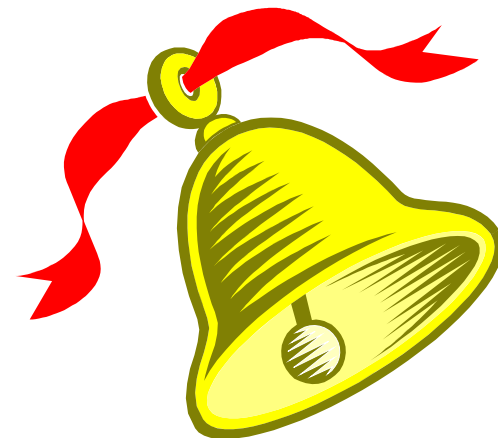
```
<bean id="productCatalog"  
  class="org.springframework.jndi.JndiObjectFactoryBean">  
  <property name="jndiName">  
    <value>  
      java:comp/env/ejb/ProductCatalogHome  
    </value>  
  </property>  
</bean>
```



Expose an object from
JNDI as a spring bean

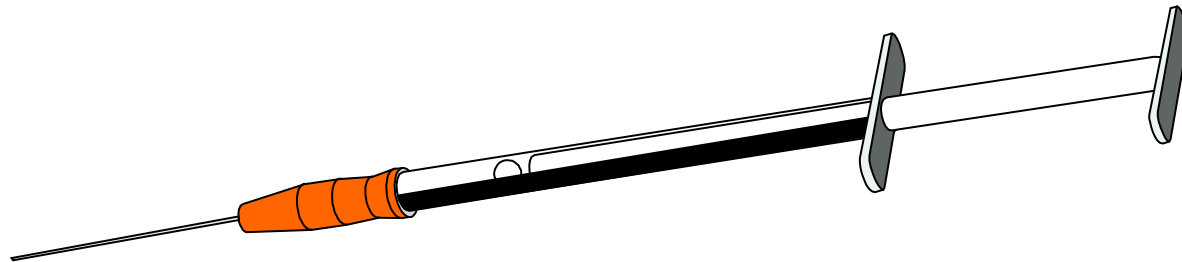
Configuration Bells and Whistles

- Autowiring – byName/byType
- Dependency Checking – simple/object/all
- Lifecycling – init/destroy
- Scope – singleton/non-singleton
- Constructor Resolution
- Custom Property Editors
- ...

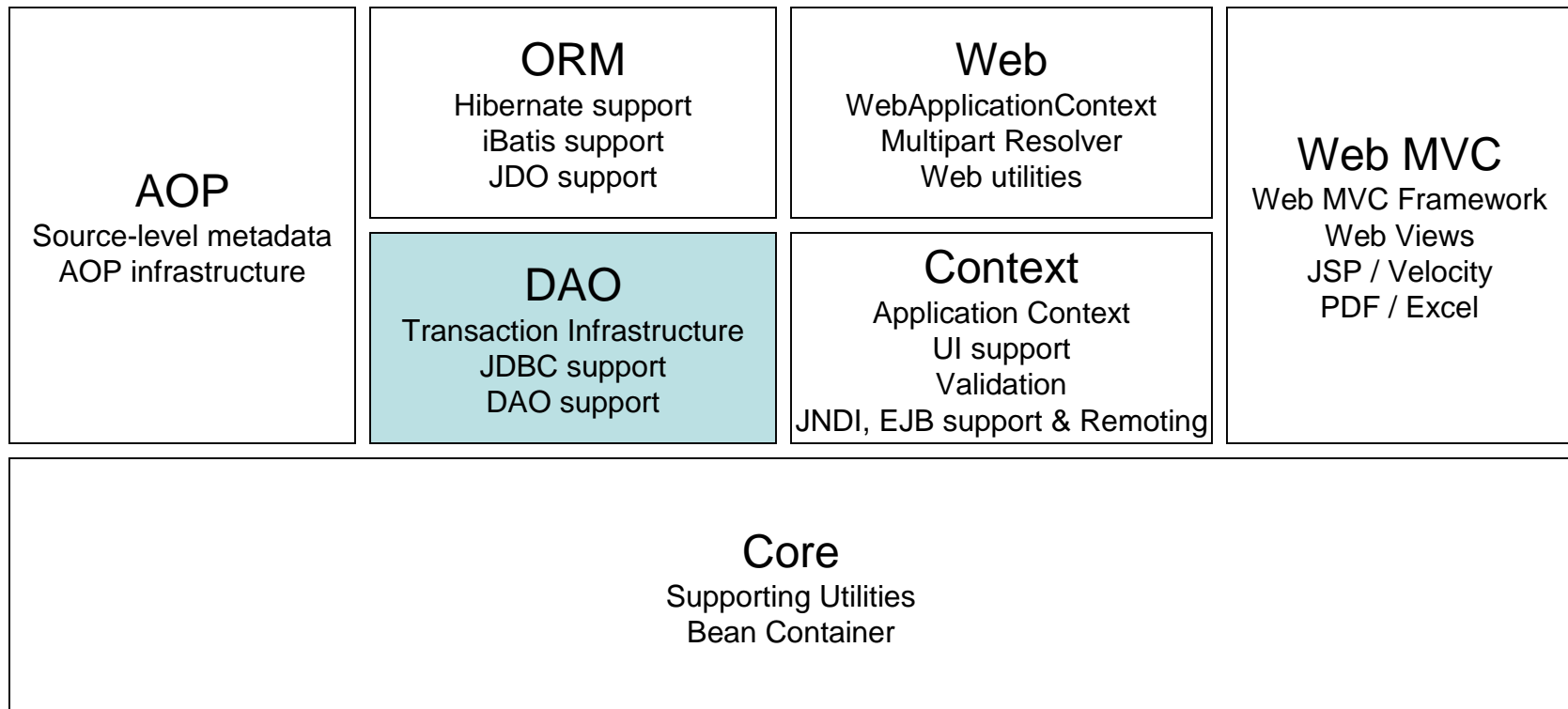


Dependency Injection summary

- Extremely simple (almost too simple?!), yet very powerful
- Proven technology (even though the term Dependency Injection is new!)
- Opens up lots of interesting, unexpected possibilities

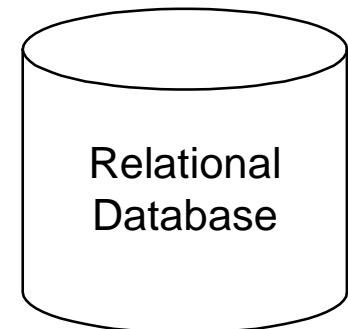


Data Access



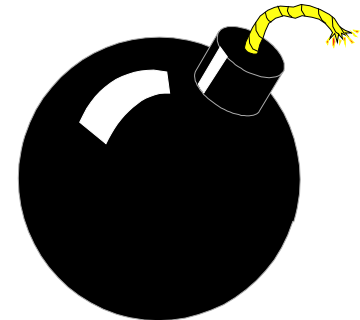
Spring DAO support

- Several different Data Access techniques exists
 - JDBC based Data Access Objects
 - Object Relational mapping techniques such as Hibernate, JDO etc.
- Using **technology agnostic** DAO interfaces is a best practice, but involves several challenges
 - API complexity and annoyances
 - Different Exceptions strategies and (mis)uses
 - Must be integrated with transaction management strategy
- Spring provides a set of **abstractions** and supporting **templates** which makes things a bit easier

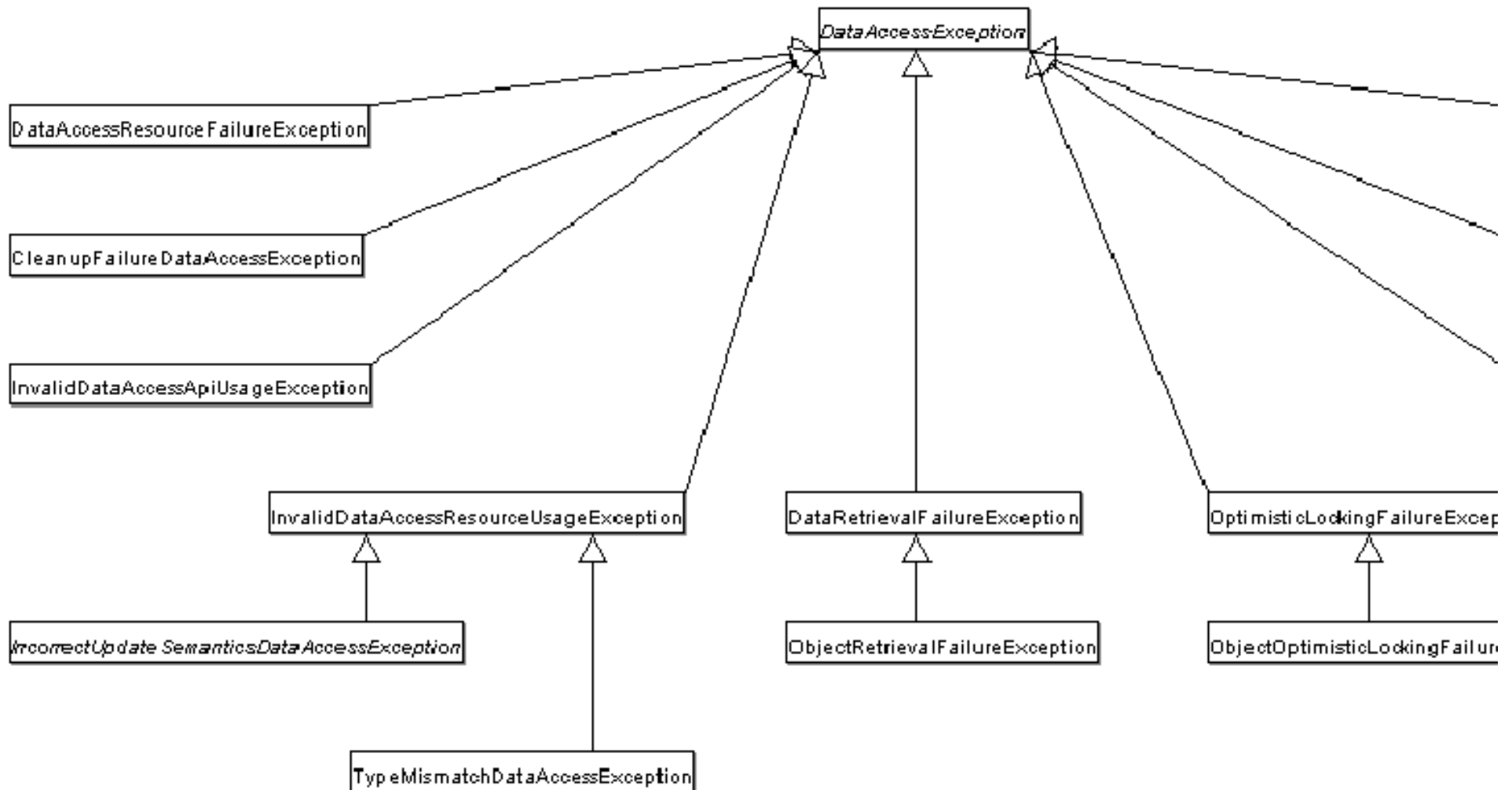


JDBC Exceptions

- The Exception model in JDBC sucks in several respects:
 - No meaningful hierarchy – only SQLException
 - No easy way to extract semantic information out of an SQLException (needs access to Vendor Code)
 - Using Checked Exceptions for usually non-recoverable error conditions is extremely obtrusive



Spring Data Access Exceptions



Annoyance: Classical JDBC

```
Connection con = datasource.getConnection();
PreparedStatement stmt = null;
try {
    stmt = con.prepareStatement("UPDATE products SET price = ?");
    stmt.setInt(1, 200);
    stmt.executeUpdate();
} finally {
    if (stmt != null) {
        try {
            stmt.close();
        } catch (SQLException ex) {
            logger.warn("Could not close statement", ex);
        }
    }
    try {
        con.close();
    } catch (SQLException ex) {
        logger.warn("Could not close connection", ex);
    }
}
```

Spring JDBC Template

Good example of Inversion of Control (also known as the Hollywood Principle: *Don't call us, we'll call you!*)

```
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
jdbcTemplate.update(  
    "UPDATE products SET price = ?",  
    new PreparedStatementSetter() {  
        public void setValues(PreparedStatement ps) throws SQLException {  
            ps.setInt(1, 200);  
        }  
    }  
);
```

The JDBC template takes care of creating and releasing connections and statements, catches SQLExceptions and transforms them into Spring generalized exceptions

JDBC Template: Query

```
final List products = new ArrayList();
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
jdbcTemplate.query(
    "SELECT * FROM products WHERE price > ?",
    new PreparedStatementSetter() {
        public void setValues(PreparedStatement ps) throws SQLException {
            ps.setInt(1, 200);
        }
    },
    new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
            ProductEntityValue product = new ProductEntityValue();
            product.setId(rs.getInt("id"));
            product.setImage(rs.getString("image"));
            products.add(product);
        }
    }
);
```

Spring Hibernate Template

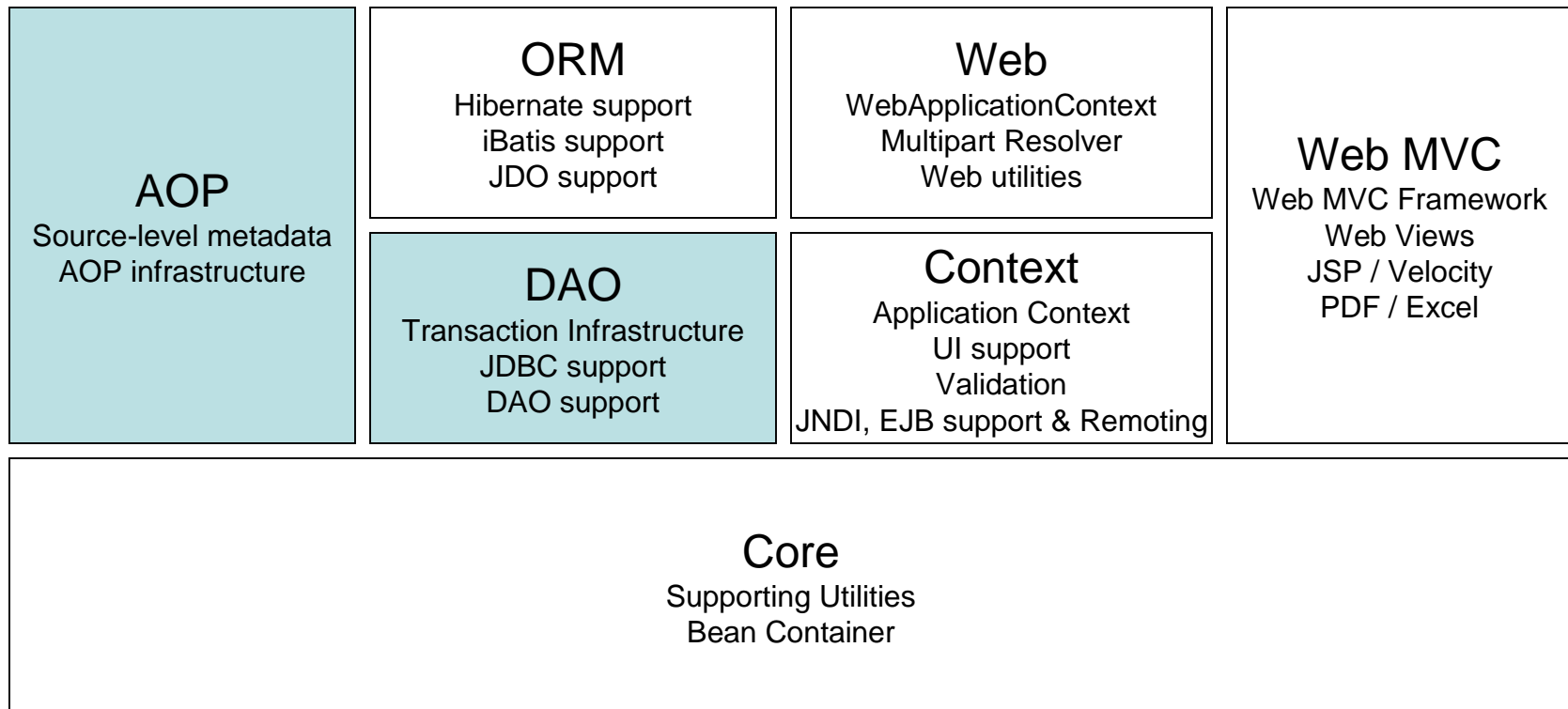
```
HibernateTemplate template = new
    HibernateTemplate(sessionFactory);

List customers = (List) template.execute(
    new HibernateCallback() {
        public Object doInHibernate(Session session)
            throws HibernateException {
            return session.find("FROM Customer WHERE name = ?",
                customerName, Hibernate.STRING);
        }
    }
);
```

Spring Data Access Benefits

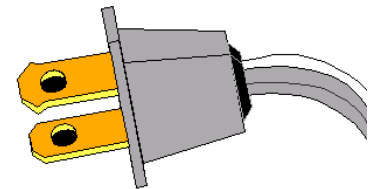
- Arguably simpler programming model than raw JDBC/Hibernate/JDO?
 - No more try/catch/finally blocks
 - No more leaked connections/sessions
- Consistent, *meaningful* exception hierarchy
 - No more vendor code lookups
- Consistent transaction management integration

Spring Transaction Management



Light Transaction Infrastructure

- Programmatic transaction demarcation
- Declarative transaction demarcation for ordinary POJOs
- Pluggable transaction strategies
 - Single JDBC DataSource
 - JTA (if distributed transactions are needed)
 - JDO PersistenceManager Factory or Hibernate SessionFactory



Programmatic TX demarcation annoyance: Classical JTA usage

```
InitialContext ictx = new InitialContext();
UserTransaction utx =
    (UserTransaction) ictx.lookup("java:comp/UserTransaction");
try {
    utx.begin();
    // Do some work on a transactional resource
    utx.commit();
} catch (RollbackException e) {
    // Recovery strategy?
} catch (HeuristicMixedException e) {
    // Recovery strategy?
} catch (HeuristicRollbackException e) {
    // Recovery strategy?
} catch (NotSupportedException e) {
    // Recovery strategy?
} catch (SystemException e) {
    // Recovery strategy?
}
```

Spring Programmatic TXs

```
TransactionTemplate txTemplate =
    new TransactionTemplate(txManager);
Object result = txTemplate.execute(
    new TransactionCallback() {
        public Object doInTransaction(TransactionStatus status) {
            // Do some work on a transactional resource
            // Return some result object or throw an unchecked exception
        }
    }
);
```

PlatformTransactionManager
interface provides abstraction
over different transaction strategies

Declarative TX demarcation: Classical EJB CMT

```
<container-transaction >
```

```
  <method >
```

```
    <ejb-name>StoreService</ejb-name>
```

```
    <method-intf>Local</method-intf>
```

```
    <method-name>createOrder</method-name>
```

```
    <method-params>
```

```
      <method-param>java.lang.String</method-param>
```

```
      <method-param>java.lang.String</method-param>
```

```
    </method-params>
```

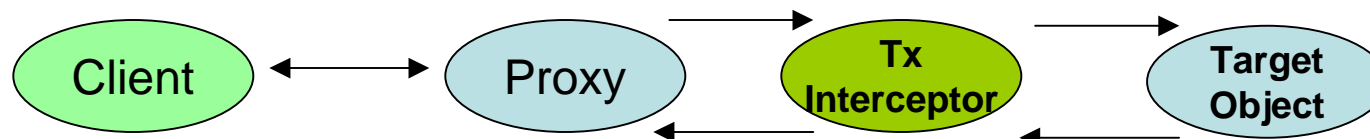
```
  </method>
```

```
  <trans-attribute>Required</trans-attribute>
```

```
</container-transaction>
```

Spring Declarative TX demarcation: AOP!

- AOP + Dependency Injection is a match in heaven
 - Any object obtained from the BeanFactory can be transparently ***advised***
 - Advisors result in a **Proxy** being inserted between the Client and the advised object, which can intercept calls and decorate them with e.g. transaction demarcation



Explicitly defining a Proxy

```
<bean id="storeService"  
  class="org.springframework.ProxyFactoryBean">  
  <property name="target">  
    <bean  
      class="se.callista.store.StoreService"/>  
  </property>  
  <property name="interceptorNames">  
    <list><value>txInterceptor</value></list>  
  </property>  
</bean>
```

Defines a new bean as a proxy for the original bean, but adds a transaction interceptor

AutoProxying

```
<bean id="storeService" class="...">...</bean>
<bean id="customerService" class="...">...</bean>

<bean id="beanNameProxyCreator"
      class="org.springframework.beans.factory.annotation.AnnotationMethodAdvisor"
      <property name="beanNames">
        <value>*Service</value>
      </property>
      <property name="interceptorNames">
        <list><value>txInterceptor</value></list>
      </property>
    </bean>
```

Consistently apply an interceptor to all applicable objects based on bean names (efficiently hiding the non-advised beans)

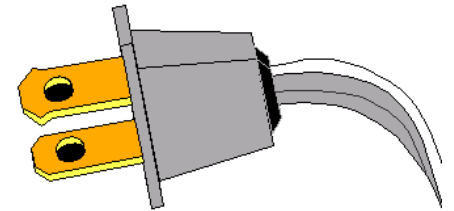
Attribute-Driven Declarative TXs

```
public class StoreService {  
  
    /**  
     * Create a new order.  
     * @return Returns the created order  
     *  
     * @@DefaultTransactionAttribute()  
     */  
    public OrderValue createOrder(String username,  
                                   OrderValue order) {  
  
        // ...  
  
    }  
}
```

Transaction Strategy: JTA

```
<bean id="dataSource"  
    class="org.springframework.jndiobjectfactorybean">  
    <property name="jndiName">  
        <value>java:comp/env/jdbc/callistaStore</value>  
    </property>  
</bean>
```

```
<bean id="transactionManager"  
    class=" org.springframework.jtaTransactionManager"/>
```



A specific implementation of the PlatformTransactionManager interface is configured as a bean and injected into all other beans that needs it

Transaction Strategy: JDBC

```
<bean id="dataSourceImpl" class="org.apache.dbcp...BasicDataSource">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/callistaStore</value>
  </property>
</bean>
```

```
<bean id="dataSource"
      class="org.springframework.transaction.TransactionAwareDataSourceProxy">
  <property name="targetDataSource">
    <ref local="dataSourceImpl"/>
  </property>
</bean>
```

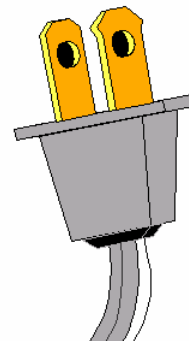
```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource"><ref bean="dataSourceImpl"/></property>
</bean>
```



Transaction Strategy: Hibernate Local Session

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref local="dataSource"/>
  </property>
  <property name="mappingResources">
    <value>hibernate/callistaStore.hbm.xml</value>
  </property>
  <property name="hibernateProperties">
    <props><prop key="hibernate.dialect">${hibernate.dialect}</prop></props>
  </property>
</bean>

<bean id="transactionManager"
  class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>
```

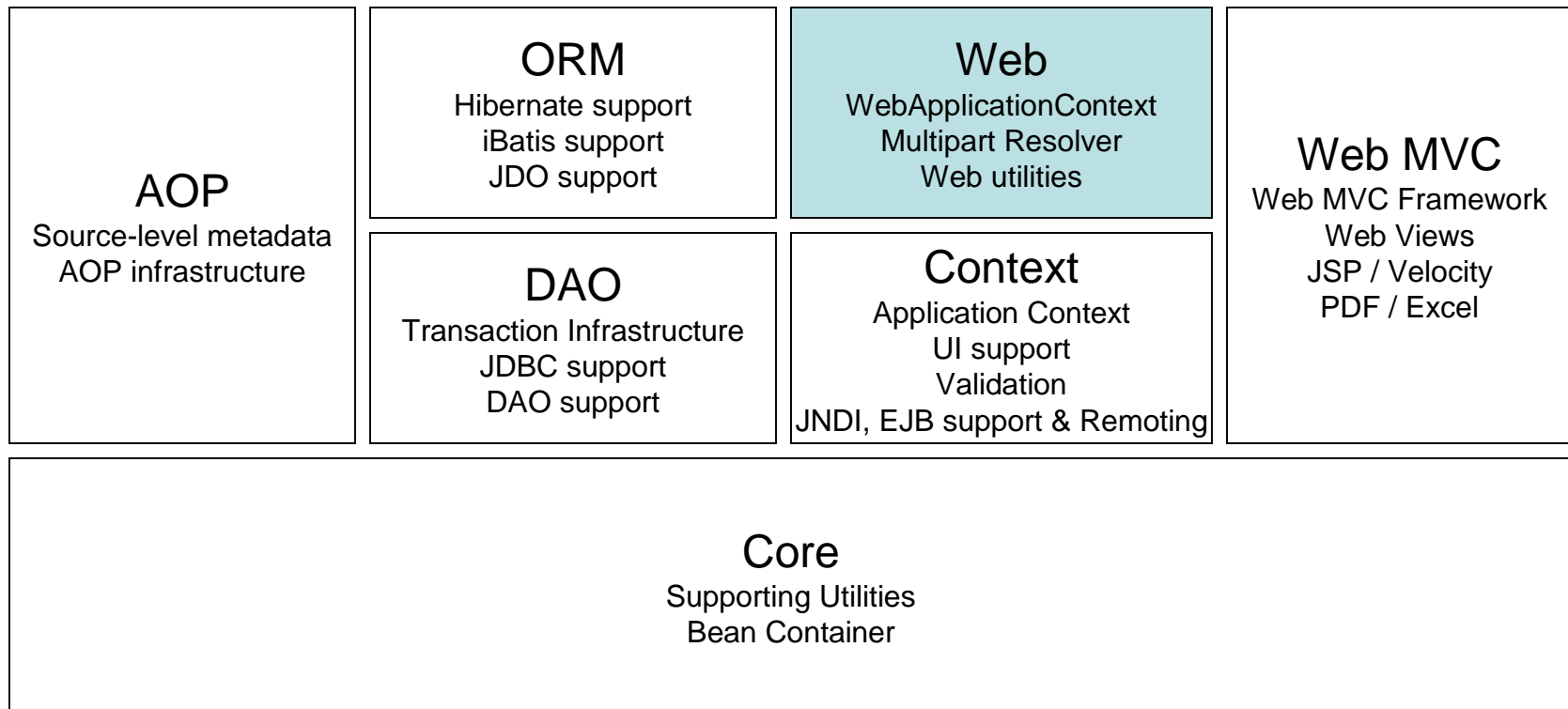


Spring Transaction Benefits

- Same power as EJB, but to a lower price
 - Works with ordinary POJOs
- Pay-as-you-go model
 - Single JDBC DataSource
 - JTA (if distributed transactions are needed)
- Seamless integration with Spring Data Access abstractions



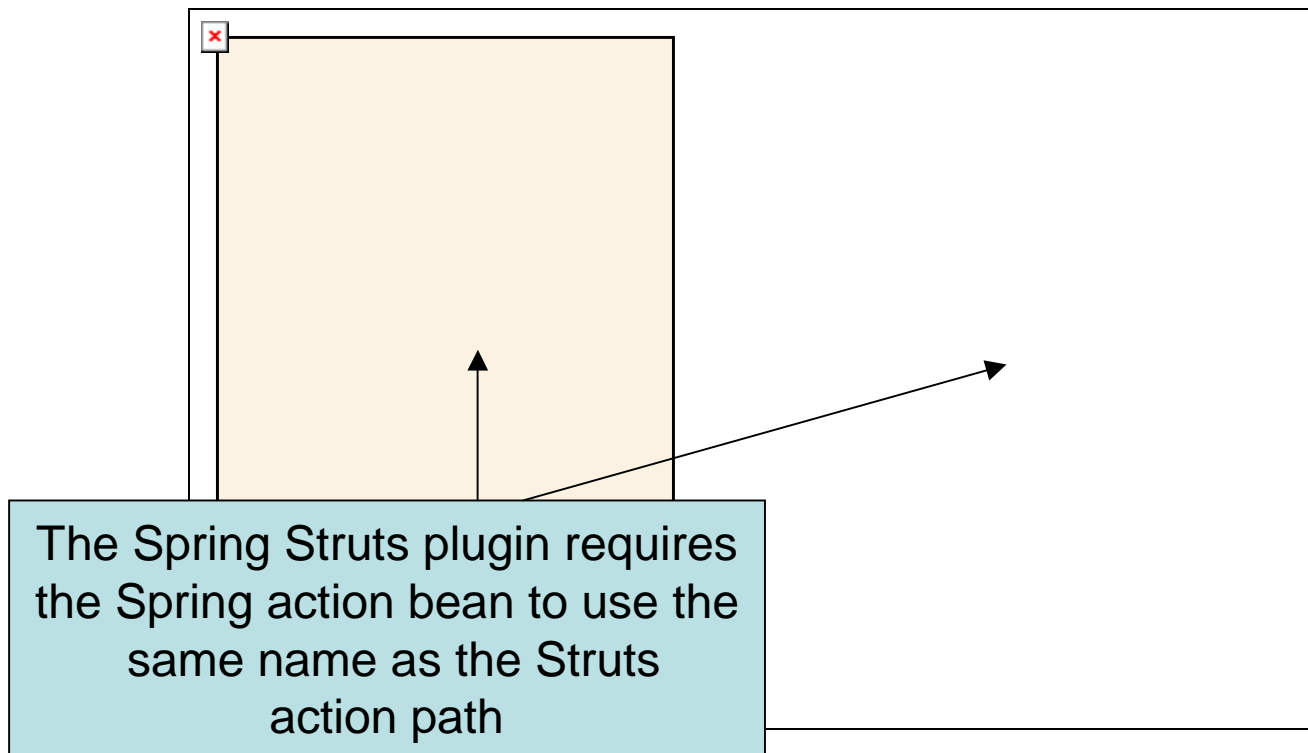
Spring Web



Spring Web

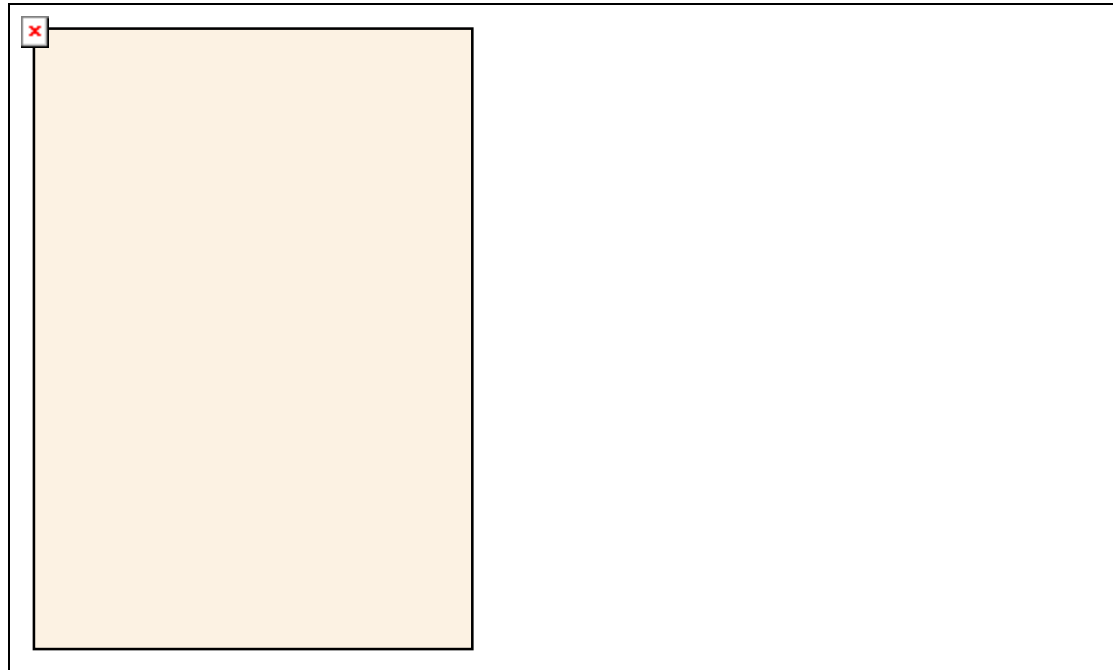
- Web Intergration layer, with out-of-the-box integration with
 - Struts
 - JSF
- Reasonably simple to integrate your own, favourite Web framework

Struts - Spring



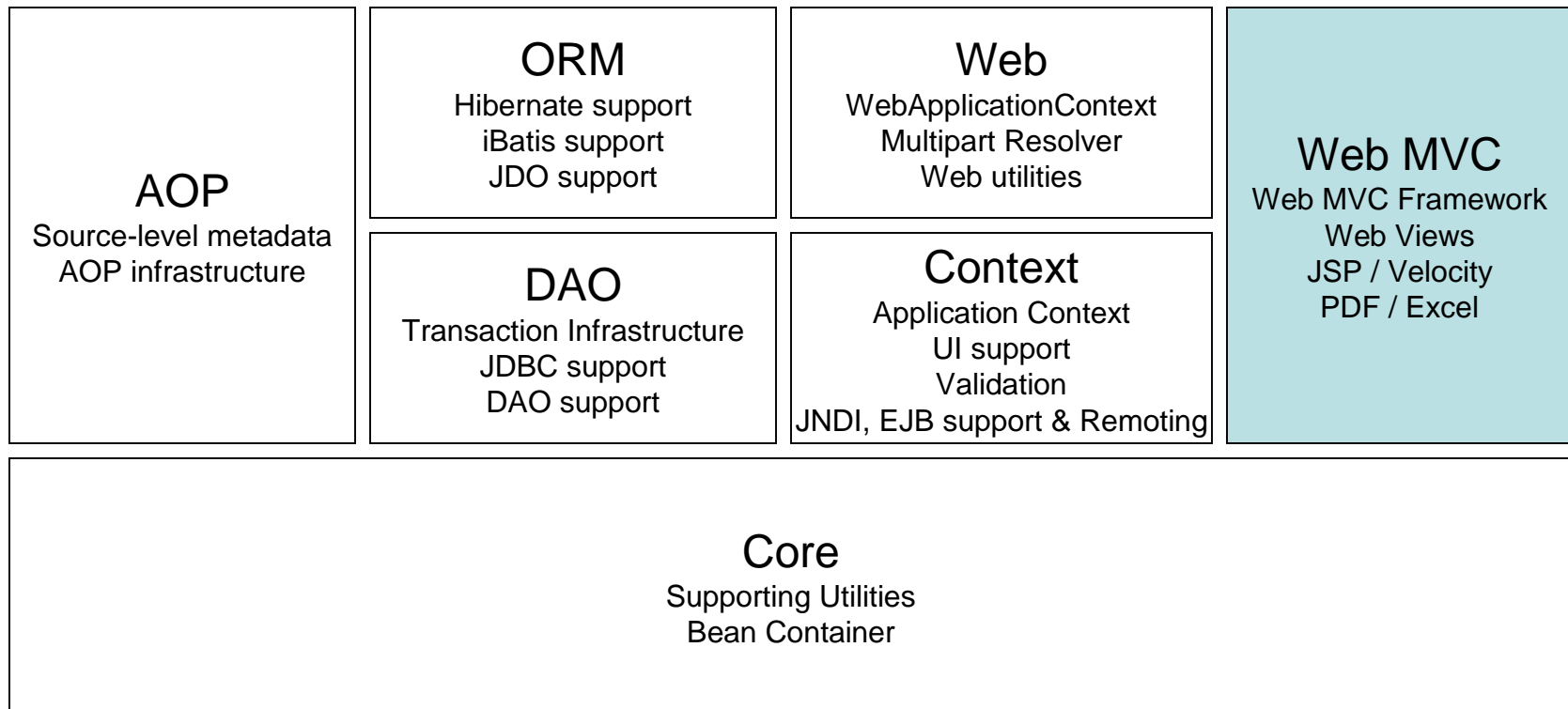
```
<action path="/logon"  
type="org.springframework.web.struts.DelegatingActionProxy">  
    <forward name="success" path="/logon.jsp"/>  
</action>
```

JSF - Spring



```
<managed-bean>  
  <managed-property>  
    <property-name>storeService</property-name>  
    <value>#{storeService}</value>  
  </managed-property>  
</managed-bean>
```

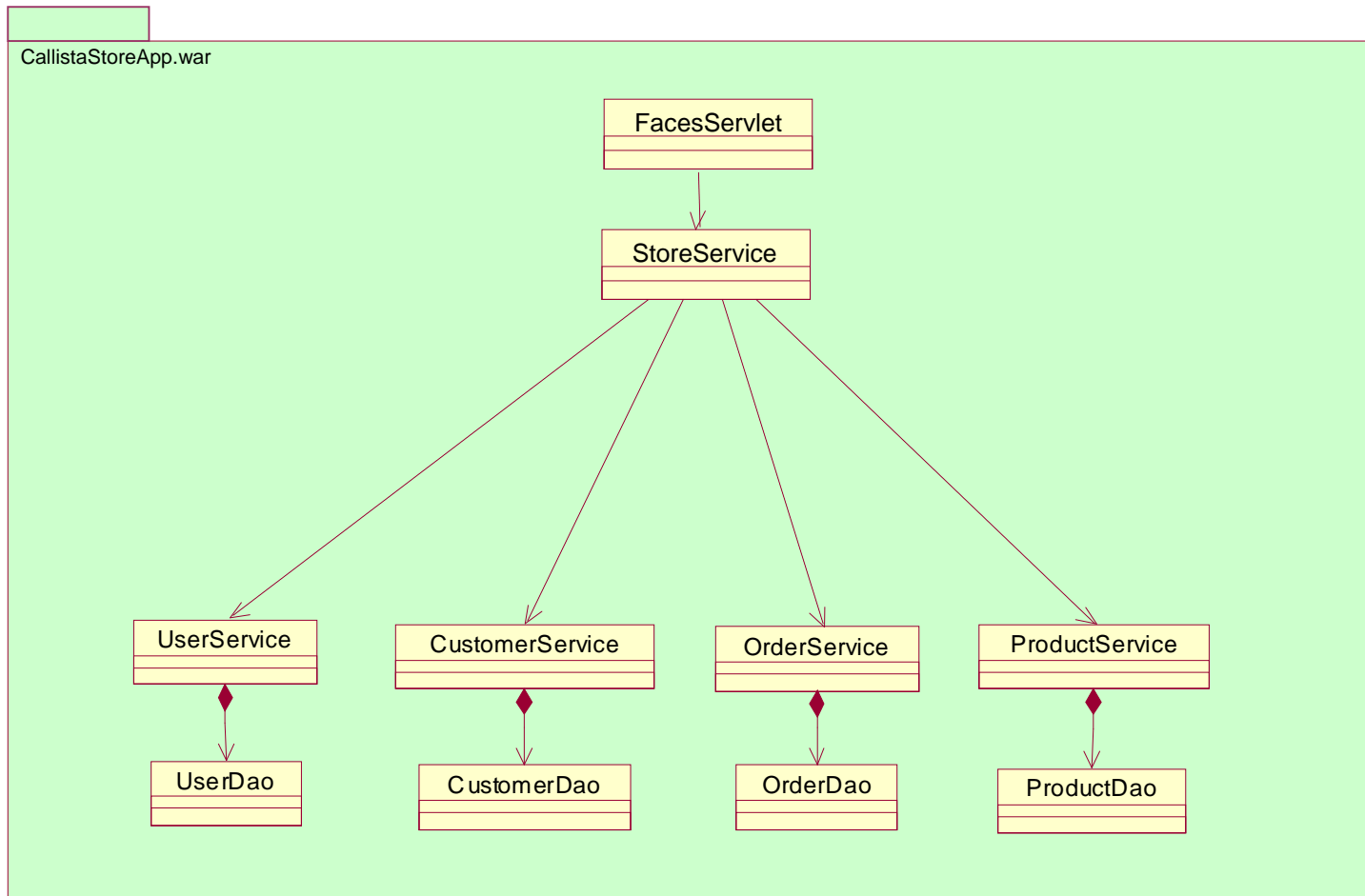
Spring Web MVC



Spring Web MVC

- Spring defines its own MVC Framework
- Supports different Web Views
 - JSP
 - XML/XSLT
 - Tapestry
 - Velocity
 - Excel
 - PDF
 - ...

CallistaStore™, Spring Edition!



EJB

vs

Spring

- Write a Session Bean
 - Home interface
 - Component interface
 - “Business methods” interface
 - Bean implementation class
 - Complex XML configuration
 - POJO delegate behind it if you want to test outside the container
- Much of this is technical plumbing
- If you want parameterization it gets even more complex

- Implement a Spring managed service
 - Business interface
 - Implementation class
 - Reasonably straightforward XML configuration
- Managing property configuration or object dependencies is easy

EJB

vs

Spring

- Using a Session Bean
 - Write a Service Locator and/or Business Delegate: need JNDI code
 - Depend on EJB interface (home interface) *or use* a Business Delegate with substantial code duplication
- *Hard to test outside a J2EE container*

- Using a Spring managed bean
 - Just write the class that uses it in plain old Java
 - Express a dependency of the business interface type
 - Reasonably straightforward XML configuration
 - *No lookup code*
- *Easy to test with mock objects*

Spring Bottom Line Value Proposition

- Almost all of your application code can be written with little or no dependency at all on any specific infrastructure or execution environment
- Low entry-level complexity, with a pay-as-you-go model for additional power and quality of services



but ...

- Open Source framework, not an approved standard
- Single "Vendor" lock-in?
- Still a fairly young, dynamic, innovative framework
- Documentation and examples not yet comprehensive (but improving)
- Is there a natural migration path into EJB3?



Time for Questions!



To conclude ...

J2EE doesn't have to be that complex! The times are changing – **there are faster, lighter, better ways.** Initiatives like Spring are only the beginning ...