



www.uml.org.cn

J2EE轻量型容器框架

-Spring Hibernate Struts



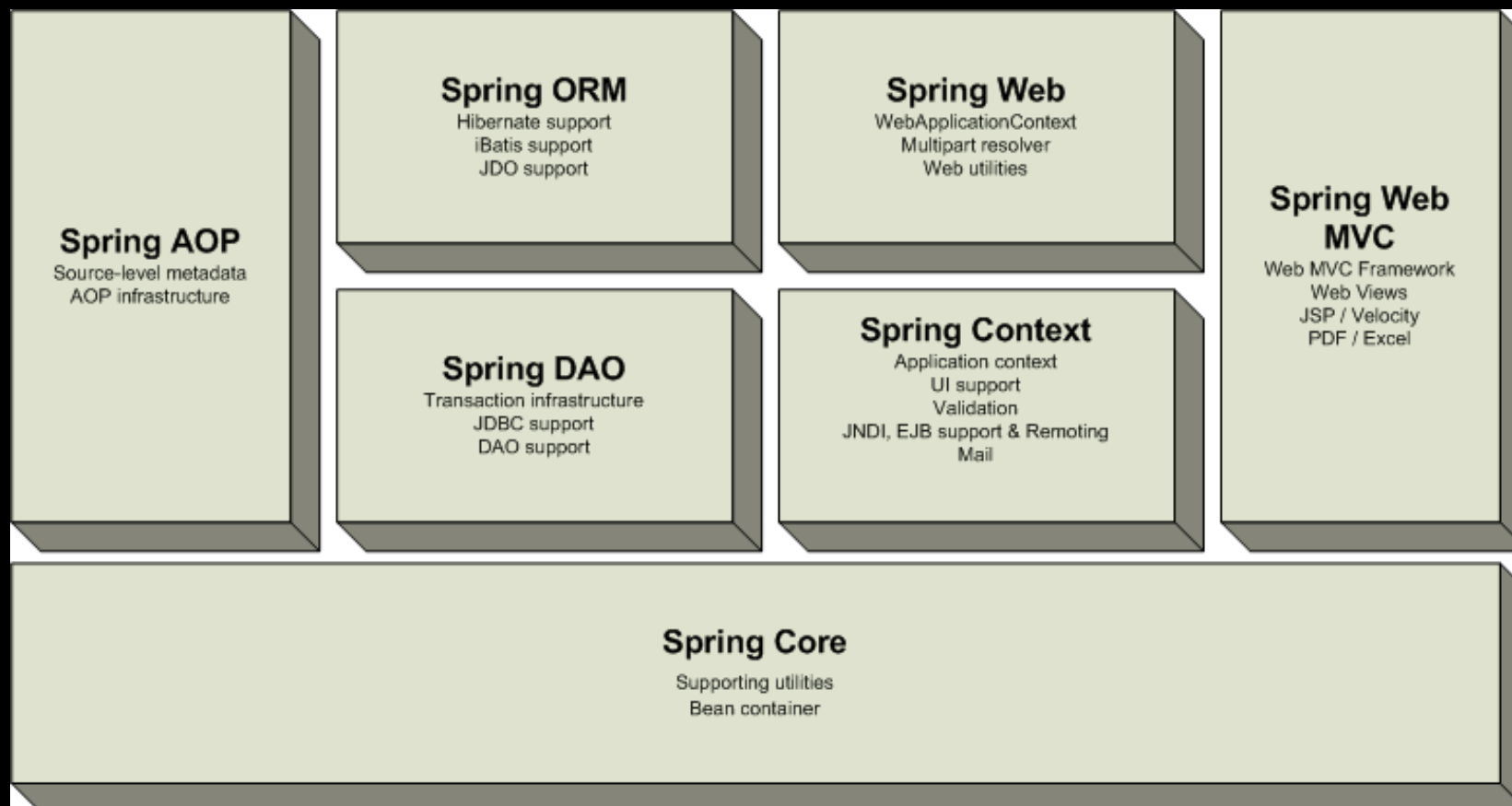
www.uml.org.cn

费书宁

Spring 架构



www.uml.org.cn



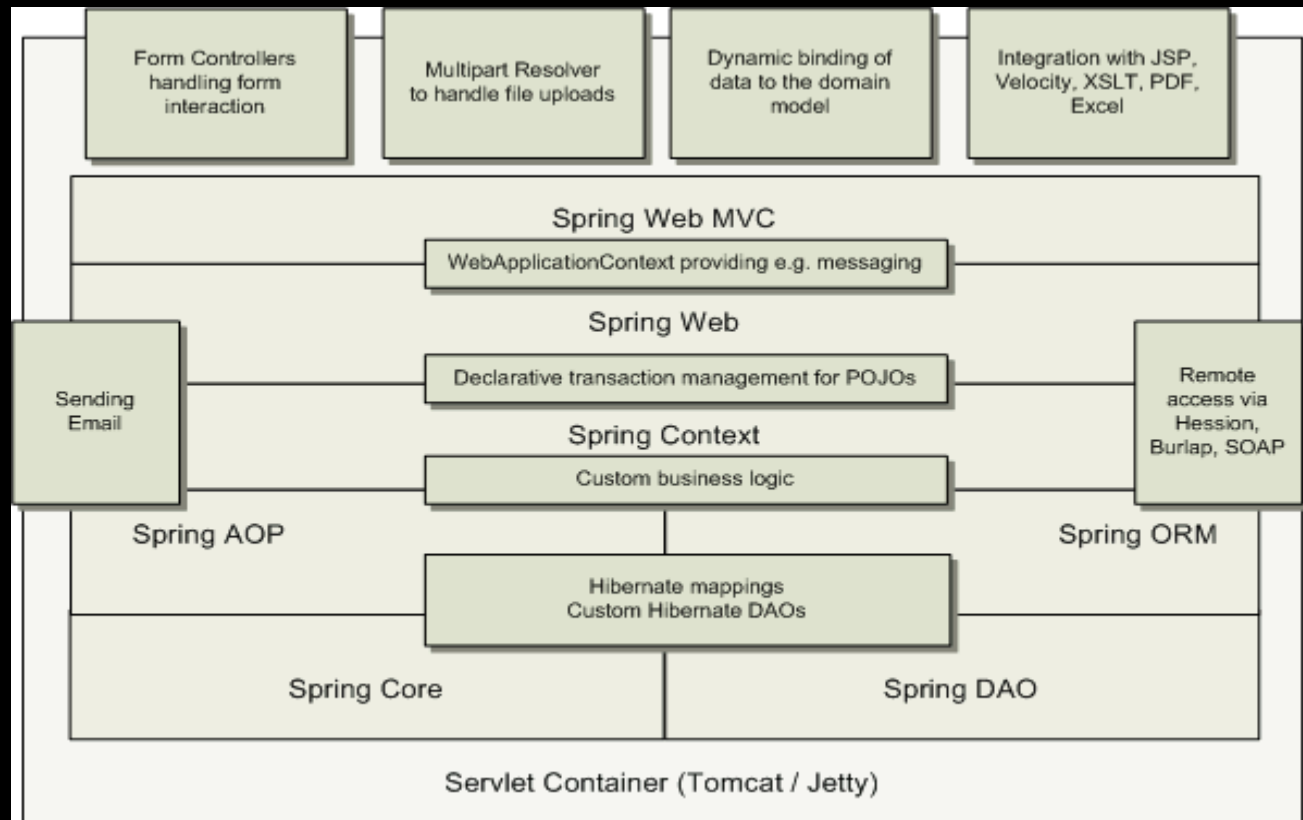
- ◆ **Spring核心模块：**
Spring框架中基础模块，提供了IoC容器，即DI（依赖注入）。
其中重要概念包括：BeanFactory和ApplicationContext
- ◆ **Spring AOP模块：**
实现了AOP联盟中定义的AOP编程实现。比如，提供拦截器实现事务管理。
- ◆ **Spring Context模块：**
直接位于Spring核心模块之上。Spring上下文模块除了继承Spring核心模块的功能外，还添加了资源绑定、事件移植、资源装载以及透明地装载上下文等功能。

- ◆ Spring Web模块：
提供面向Web应用集成的功能。
- ◆ Spring DAO 模块：
提供了JDBC抽象层。
- ◆ Spring ORM模块：
为当前流行的O/R Mapping技术提供集成
- ◆ Spring Web MVC 模块：
Spring 提供的MVC实现

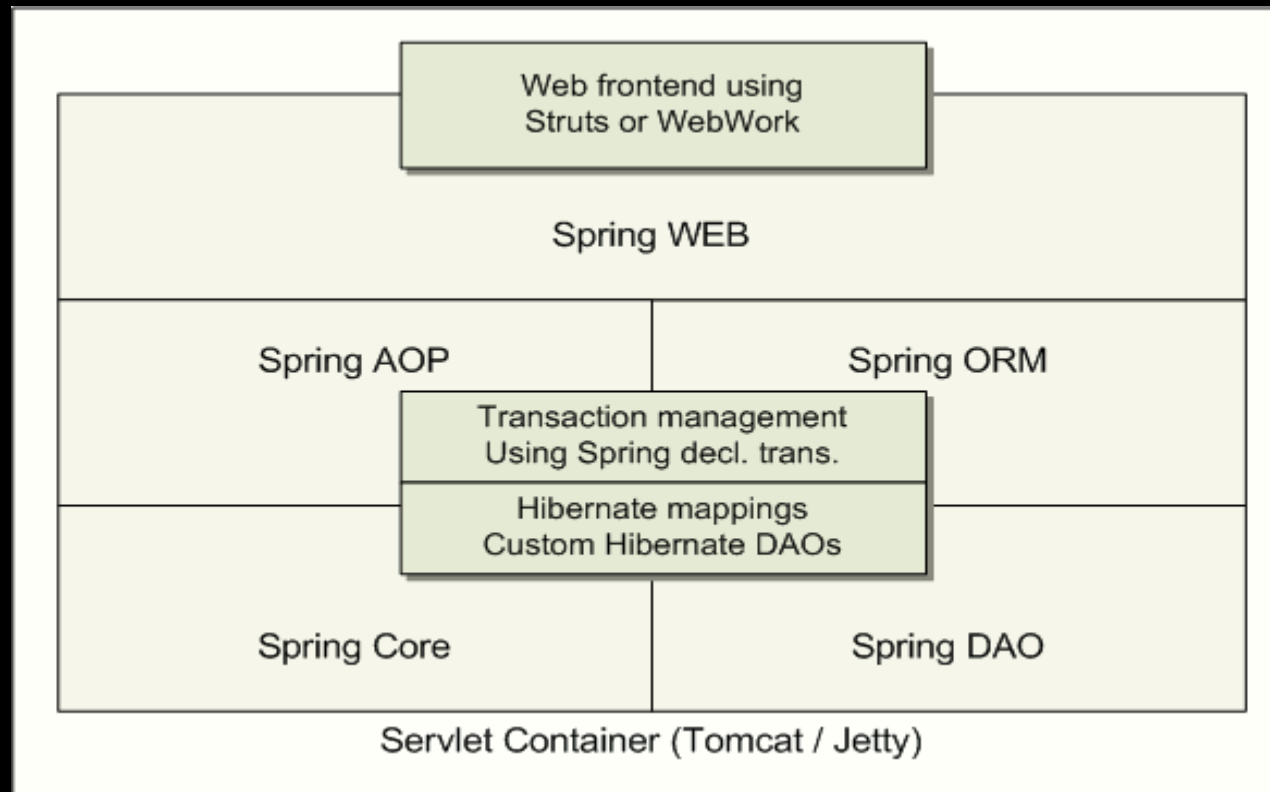
Spring 应用



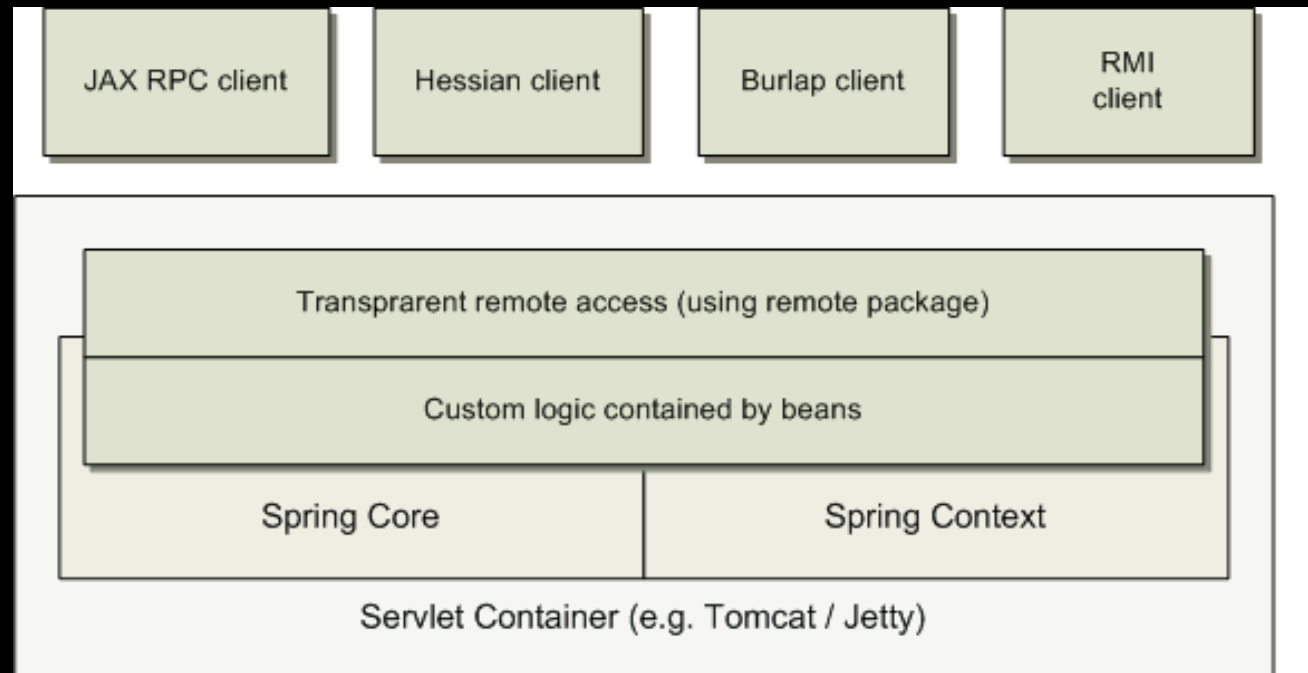
www.uml.org.cn



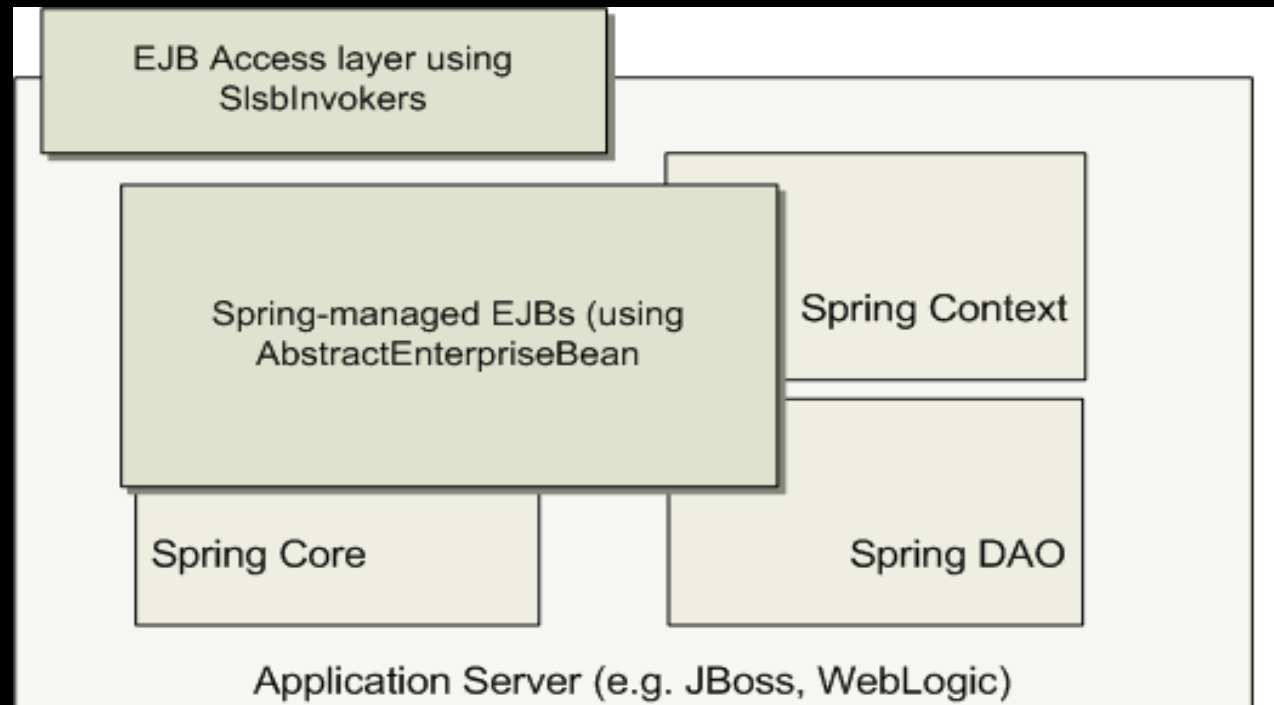
Typical full-fledged Spring web application



Spring middle-tier using a third-party web framework



Remoting usage scenario



EJBs - Wrapping existing POJOs

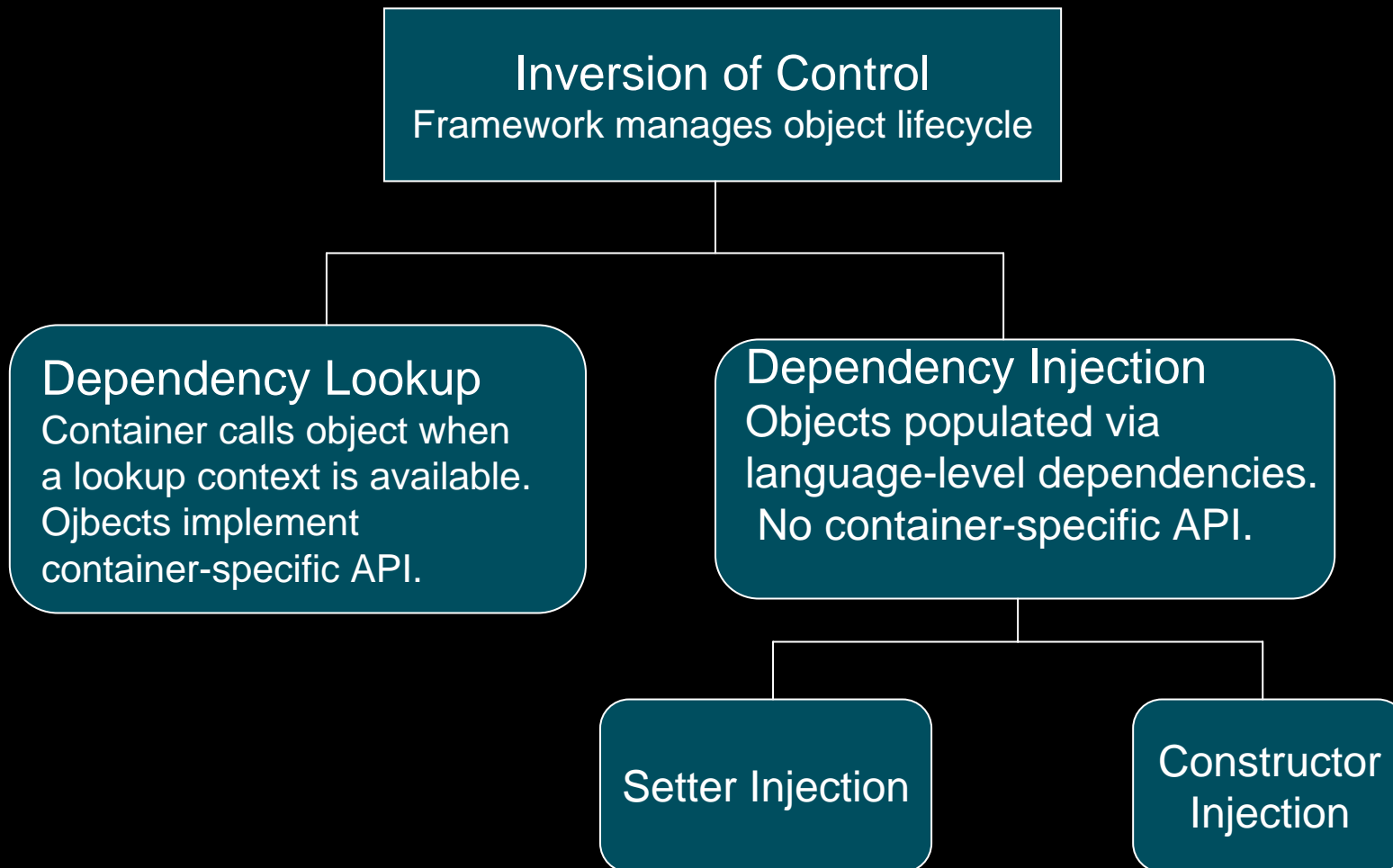
◆ 什么是IoC?

所谓IoC，就是由容器控制程序之间的关系，而非传统实现中由程序代码控制。

◆ IoC实现策略：

-依赖查找：容器提供回调接口和上下文环境给组件。这样，组件使用容器提供的API来查找资源和协作对象，仅有的控制反转只体现在那些回调方法上：容器将调用这些回调方法，从而让应用代码获得相关资源。

-依赖注入(DI)：组件不做定位查询，只提供普通的Java方法让容器去决定依赖关系。容器全权负责组件的装配，它会把符合依赖关系的对象通过JavaBean属性或构造函数传递给需要的对象。通过JavaBean属性注射依赖关系称设值方法注入(Setter Injection)；将依赖关系作为构造方法参数传入称为构造子注入(Constructor Injection)



- ◆ DI是目前最流行的IoC类型
让容器去全权负责依赖查询，受管对象只需暴露JavaBean的Setter方法或者带参数的构造方法，使容器可以在初始化时组装对象的依赖关系。
 - 查找定位操作与应用代码完全无关。
 - 不依赖容器的API
 - 不需要特殊的接口
- ◆ Type1 接口注入（依赖查找）

```
public class ClassA {  
    private InterfaceB clzB;  
    public init() {  
        Object obj =  
            Class.forName(Config.BImplementation).newInstance();  
        clzB = (InterfaceB)obj;  
    }  
    .....  
}
```

对于一个Type1型IoC容器而言，加载接口实现并创建其实例的工作由容器完成，如J2EE开发中常用的Context.lookup（ServletContext.getXXX），都是Type1型IoC的表现形式。

◆ Type2 构造子注入

```
public class DIByConstructor {  
    private final DataSource dataSource;  
    private final String message;  
    public DIByConstructor(DataSource ds, String msg) {  
        this.dataSource = ds;  
        this.message = msg;  
    }  
    .....  
}
```

在Type2类型的依赖注入机制中，依赖关系是通过类构造函数建立，容器通过调用类的构造方法，将其所需的依赖关系注入其中。

◆ Type3 设值注入

```
public class DIBySetter{  
    private final DataSource dataSource;  
    private final String message;  
    public) {  
    }  
    public void setDataSource(DataSource dataSource){  
        this.dataSource = dataSource;  
    }  
    public void setMessage(String message){  
        this.message = message;  
    }  
    .....  
}
```

◆ Type2 构造子注入的优势：

1. “在构造期即创建一个完整、合法的对象”，对于这条Java设计原则，Type2无疑是最好的响应者。
2. 避免了繁琐的setter方法的编写，所有依赖关系均在构造函数中设定，依赖关系集中呈现，更加易读。
3. 由于没有setter方法，依赖关系在构造时由容器一次性设定，因此组件在被创建之后即处于相对“不变”的稳定状态，无需担心上层代码在调用过程中执行setter方法对组件依赖关系产生破坏，特别是对于Singleton模式的组件而言，这可能对整个系统产生重大的影响。
4. 同样，由于关联关系仅在构造函数中表达，只有组件创建者需要关心组件内部的依赖关系。对调用者而言，组件中的依赖关系处于黑盒之中。对上层屏蔽不必要的信息，也为系统的层次清晰性提供了保证。

5. 通过构造子注入，意味着我们可以在构造函数中决定依赖关系的注入顺序，对于一个大量依赖外部服务的组件而言，依赖关系的获得顺序可能非常重要，比如某个依赖关系注入的先决条件是组件的DataSource及相关资源已经被设定。
 - ◆ Type3 设值注入的优势
1. 对于习惯了传统JavaBean开发的程序员而言，通过setter方法设定依赖关系显得更加直观，更加自然。
2. 如果依赖关系（或继承关系）较为复杂，那么Type2模式的构造函数也会相当庞大（我们需要在构造函数中设定所有依赖关系），此时Type3模式往往更为简洁。
3. 对于某些第三方类库而言，可能要求我们的组件必须提供一个默认的构造函数（如Struts中的Action），此时Type2类型的依赖注入机制就体现出其局限性，难以完成我们期望的功能。

Spring IoC容器实现了IoC设计模式：

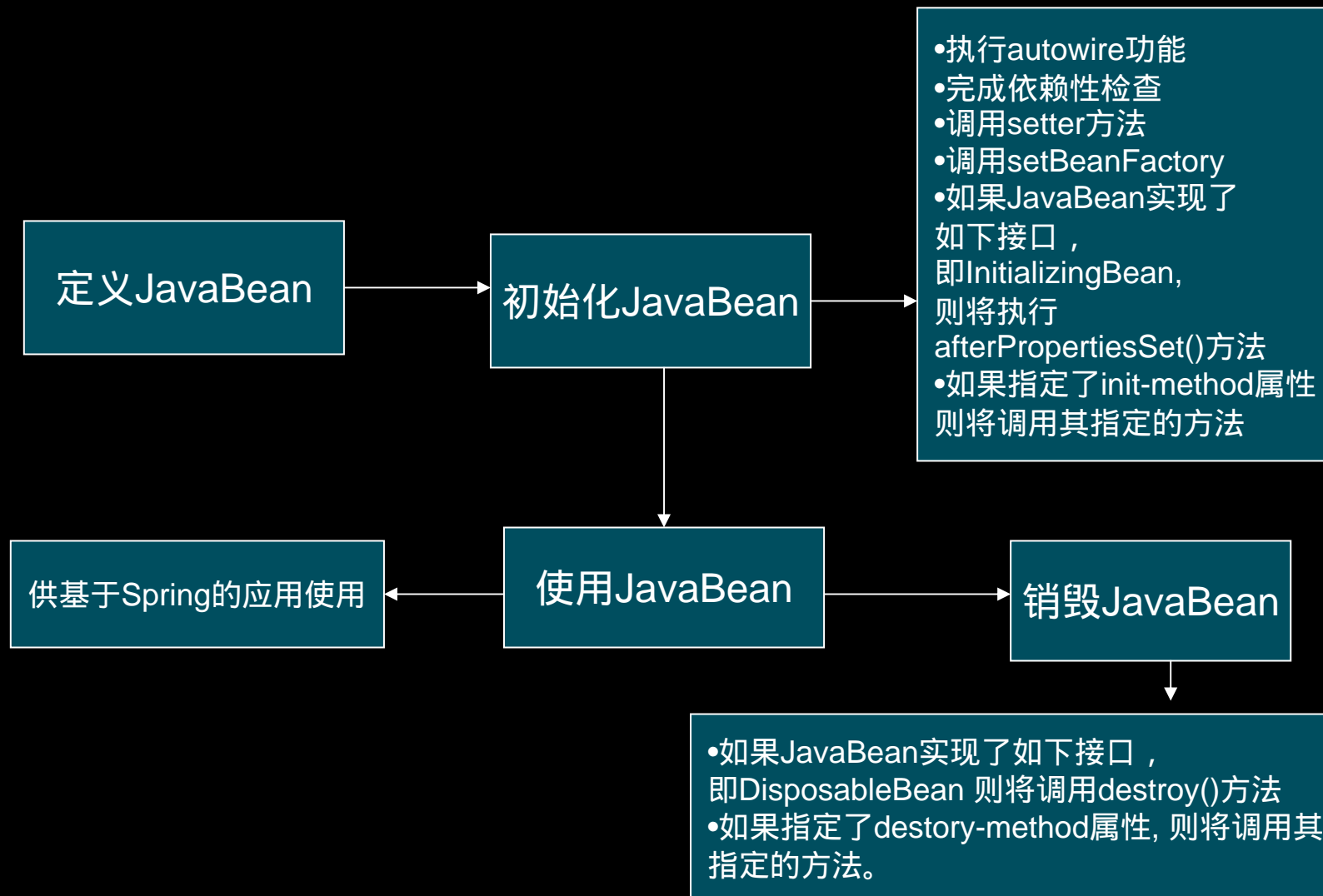
- ◆ BeanFactory:

位于org.springframework.beans.factory包中开发者借助于配置文件(xml或属性文件)，能够实现对JavaBean的配置与管理。

- ◆ ApplicationContext

位于org.springframework.context包中。ApplicationContext构建在BeanFactory基础之上，既继承于它。除了具有BeanFactory功能外，还添加了大量功能：处理消息资源，事件分发、声明(非)容器提供的服务等，用于J2EE应用

◆ Bean的声明周期：



◆ 利用XML组装JavaBean

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="..." class="...">
        .
        ..
    </bean>
    <bean id="..." class="...">
        ...
    </bean>
    ...
</beans>
```

- ◆ 对于采用手工控制事务，即程序控制事务的编程方式，Spring提供的事务抽象易于使用
- ◆ 对于任何事务API，Spring提供了一致的编程模型。
- ◆ Spring支持以声明方式管理事务
- ◆ 能够同Spring的DAO抽象进行集成
- ◆ 在不同事务服务间切换，只涉及Spring配置文件的修改，不会涉及到代码的修改。

◆ 事务策略

org.springframework.transaction.PlatformTransactionManager

```
public interface PlatformTransactionManager {  
    TransactionStatus getTransaction(  
        TransactionDefinition definition) throws TransactionException;    void  
    commit(TransactionStatus status) throws TransactionException;    void  
    rollback(TransactionStatus status) throws TransactionException;  
}
```

-JDBC 事务: DataSourceTransactionManager

-JTA事务: JtaTransactionManager、WebLogicJtaTransactionManager

-Hibernate事务: HibernateTransactionManager

-JDO事务 : JdoTransactionManager

-OJB事务 : PersistenceBrokerTransactionManager

-JMS事务 : JmsTransactionManager

◆ Hibernate事务：

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource">
        <ref local="dataSource"/></property>
    <property name="mappingResources">
        org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml
    </value>
</property>
```

Spring 事务管理抽象



www.uml.org.cn

```
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            ${hibernate.dialect}
        </prop>
    </props>
</property>
</bean>
```

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

◆ 声明式事务

<!-- this example is in verbose form, see note later about concise for multiple proxies! -->

<!-- the target bean to wrap transactionally -->

<bean id="petStoreTarget">

...

</bean>

<bean id="petStore" class=

"org.springframework.transaction.interceptor.TransactionProxyFactoryBean">

<property name="transactionManager">

<ref bean="transactionManager"/>

</property>

Spring 事务管理抽象



```
<property name="target">
  <ref bean="petStoreTarget"/>
</property>
<property name="transactionAttributes">
  <props>
    <prop key="insert*">
      PROPAGATION_REQUIRED,MyCheckedException
    </prop>
    <prop key="update*">
      PROPAGATION_REQUIRED
    </prop>
    <prop key="*">
      PROPAGATION_REQUIRED,readonly
    </prop>
  </props>
</property>
</bean>
```

- ◆ Spring DAO抽象：
 - JdbcDaoSupport
 - HibernateDaoSupport
 - JdoDaoSupport
- ◆ JDBC
 - org.springframework.jdbc.core
 - org.springframework.jdbc.datasource
 - org.springframework.jdbc.object
 - org.springframework.jdbc.support

JDBC Template

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource">
            <ref local="dataSource"></ref>
        </property>
    </bean>
```

◆ Hibernate抽象

<beans>

<bean id="myDataSource"

class="org.springframework.jndi.JndiObjectFactoryBean">

<property name="jndiName">

<value>java:comp/env/jdbc/myds</value>

</property>

</bean>

<bean id="mySessionFactory" class=

"org.springframework.orm.hibernate.LocalSessionFactoryBean">

<property name="mappingResources">

<list>

<value>product.hbm.xml</value>

</list>

</property>

Spring DAO



```
<property name="hibernateProperties">
  <props>
    <prop key="hibernate.dialect">
      net.sf.hibernate.dialect.MySQLDialect
    </prop>
  </props>
</property>
<property name="dataSource">
  <ref bean="myDataSource"/>
</property>
</bean>
...
</beans>
```

Spring DAO



```
<beans>
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="sessionFactory">
      <ref bean="mySessionFactory"/>
    </property>
  </bean>
  ...
</beans>
```

Spring DAO



```
public class ProductDaoImpl implements ProductDao {  
    private SessionFactory sessionFactory;  
    public void setSessionFactory(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
    public List loadProductsByCategory(final String category) {  
        HibernateTemplate hibernateTemplate =  
            new HibernateTemplate(this.sessionFactory);  
        return (List) hibernateTemplate.execute(  
            new HibernateCallback() {
```

Spring DAO



```
public Object doInHibernate(Session session) throws HibernateException {  
    List result = session.find(  
        "from test.Product product where product.category=?",  
        category, Hibernate.STRING);  
    // do some further stuff with the result list  
    return result;  
}  
}  
);  
}  
}
```


Spring DAO



```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {  
  
    public List loadProductsByCategory(String category) {  
        return getHibernateTemplate().find(  
            "from test.Product product where product.category=?", category,  
            Hibernate.STRING);  
    }  
}
```

使用AOP Interceptor 替代Template

```
<beans>
```

```
...
```

```
<bean id="myHibernateInterceptor"  
      class="org.springframework.orm.hibernate.HibernateInterceptor">  
  <property name="sessionFactory">  
    <ref bean="mySessionFactory"/>  
  </property>  
</bean>
```

Spring DAO



```
<bean id="myProductDaoTarget" class="product.ProductDaoImpl">
  <property name="sessionFactory">
    <ref bean="mySessionFactory"/>
  </property>
</bean>
<bean id="myProductDao" class="org.springframework.aop.framework.ProxyFactoryBean"
  <property name="proxyInterfaces">
    <value>product.ProductDao</value>
  </property>
```

Spring DAO



```
<property name="interceptorNames">
    <list>
        <value>myHibernateInterceptor</value>
        <value>myProductDaoTarget</value>
    </list>
</property>
</bean>

...

</beans>
```

Spring DAO



```
public class ProductDaoImpl extends HibernateDaoSupport
    implements ProductDao {
    public List loadProductsByCategory(final String category) throws MyException {
        Session session = SessionFactoryUtils.getSession(getSessionFactory(), false);
        try {
            List result = session.find(
                "from test.Product product where product.category=?",
                category, Hibernate.STRING);
        } catch (Exception e) {
            throw new MyException(e);
        }
        return result;
    }
}
```

Spring DAO



```
if (result == null) {  
    throw new MyException("invalid search result");  
}  
return result;  
}  
catch (HibernateException ex) {  
    throw SessionFactoryUtils.convertHibernateAccessException(ex);  
}  
}  
}
```

Hibernate 事务

<beans>

...

```
<bean id="myTransactionManager"
      class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref bean="mySessionFactory"/>
  </property>
</bean>

<bean id="myProductService" class="product.ProductServiceImpl">
  <property name="transactionManager">
    <ref bean="myTransactionManager"/>
  </property>
```

Spring DAO



```
<property name="productDao">  
    <ref bean="myProductDao"/>  
</property>  
</bean>
```

```
</beans>
```

```
public class ProductServiceImpl implements ProductService {  
    private PlatformTransactionManager transactionManager;  
    private ProductDao productDao;  
    public void setTransactionManager(PlatformTransactionManager transactionManager)  
    {  
        this.transactionManager = transactionManager;  
    }  
}
```



```
public void setProductDao(ProductDao productDao) {
    this.productDao = productDao;
}

public void increasePriceOfAllProductsInCategory(final String category) {
    TransactionTemplate transactionTemplate = new
    TransactionTemplate(this.transactionManager);
    transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGAT
    QUIRED);
    transactionTemplate.execute(
        new TransactionCallbackWithoutResult() {
            public void doInTransactionWithoutResult(TransactionStatus status) {
                List productsToChange = productDAO.loadProductsByCategory(catego
                ...
            }
        }
    );
}
```

Spring DAO



```
<beans>
```

```
...
```

```
<bean id="myTransactionManager"
```

```
  class="org.springframework.orm.hibernate.HibernateTransactionManager">
```

```
    <property name="sessionFactory">
```

```
      <ref bean="mySessionFactory"/>
```

```
    </property>
```

```
</bean>
```

```
<bean id="myTransactionInterceptor"
```

```
  class="org.springframework.transaction.interceptor.TransactionInterceptor">
```

Spring DAO



```
<property name="transactionManager">
    <ref bean="myTransactionManager"/>
</property>
<property name="transactionAttributeSource">
    <value>
        product.ProductService.increasePrice*=PROPAGATION_REQUIRED

        product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
    </value>
</property>
</bean>
```

Spring DAO



```
<bean id="myProductServiceTarget" class="product.ProductServiceImpl">
  <property name="productDao">
    <ref bean="myProductDao"/>
  </property>
</bean>
<bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
    <value>product.ProductService</value>
  </property>
```

Spring DAO



```
<property name="interceptorNames">
  <list>
    <value>myTransactionInterceptor</value>
    <value>myProductServiceTarget</value>
  </list>
</property>
</bean>
</beans>
```

Spring DAO



```
public class ProductServiceImpl implements ProductService {  
  
    private ProductDao productDao;  
  
    public void setProductDao(ProductDao productDao) {  
        this.productDao = productDao;  
    }  
    public void increasePriceOfAllProductsInCategory(final String category) {  
        List productsToChange = this.productDAO.loadProductsByCategory(category);  
        ...  
    }  
    ...  
}
```

- ◆ Spring 与 Struts集成两种策略
 - 配置Spring管理Action，使用ContextLoaderPlugin
 - 使用*ActionSupport*

使用ContextLoaderPlugin

在struts-config.xml中：

```
<plug-in className=  
    "org.springframework.web.struts.ContextLoaderPlugIn">  
    <set-property property="contextConfigLocation"  
value="/WEB-INF/action-servlet.xml.xml,  
                                /WEB-INF/applicationContext.xml"/></plug-in>
```

- ◆ 用Spring的 DelegatingRequestProcessor 覆盖掉Struts RequestProcessor

在<action-mapping> 中使用 *DelegatingActionProxy*

```
<controller>
```

```
  <set-property property="processorClass"
```

```
    value="org.springframework.web.struts.DelegatingRequestProcessor"/>
```

```
</controller>
```

```
<action path="/user" type="com.whatever.struts.UserAction"/>
```

```
  <action path="/user"/>
```

```
<bean name="/admin/user"/>
```


Spring Struts集成



```
<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"
        name="userForm" scope="request" validate="false" parameter="method">
    <forward name="list" path="/userList.jsp"/>
    <forward name="edit" path="/userForm.jsp"/>
</action>
```

```
<bean name="/user" singleton="false" autowire="byName"
        class="org.example.web.UserAction"/>
```

ActionSupport Classes

```
public class UserAction extends DispatchActionSupport {
```

```
    public ActionForward execute(ActionMapping mapping,  
                                ActionForm form,  
                                HttpServletRequest request,  
                                HttpServletResponse response)  
        throws Exception {  
        if (log.isDebugEnabled()) {  
            log.debug("entering 'delete' method...");  
        }  
    }
```

Spring Struts集成



www.uml.org.cn

```
WebApplicationContext ctx = getWebApplicationContext();
    UserManager mgr = (UserManager) ctx.getBean("userManager");

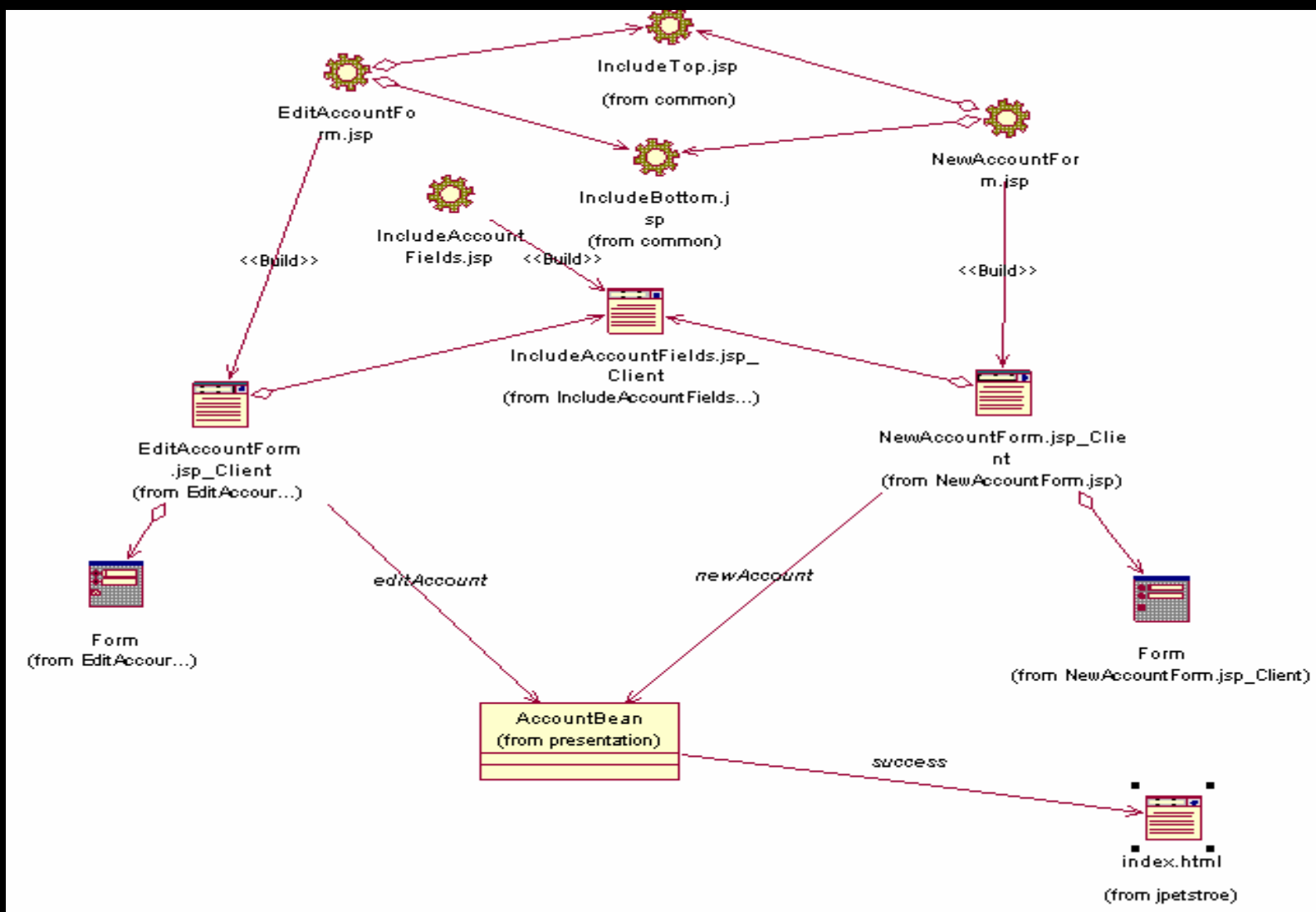
    // talk to manager for business logic

    return mapping.findForward("success");
}
}
```

案例：JPetStore



www.uml.org.cn



案例：JPetStore



www.uml.org.cn

