aosd.05

International Conference on Aspect-Oriented Software Development

Spring & AspectJ

Rob Harrop

Spring AOP Lead

Interface21

Adrian Colyer

AspectJ Lead

IBM

Agenda

- What is Spring?
- Spring AOP
- Dependency Injection and Aspects
- Aspects and Dependency Injection
- Futures

What is Spring?

- Framework for simplifying J2EE
  - Uses Plain Old Java Objects (POJO)s
  - Eliminates middle-tier glue
  - Addresses end-to-end application requirements
    - Not just a one tier solution
- Comprehensive feature set
  - Highly sophisticated IoC container
  - Pure Java AOP implementation
    - Focuses on solving common J2EE problems
  - Data access abstractions for popular tools
    - TopLink, Hibernate, JDO etc.
  - Much more
    - Web MVC, remoting, management, transaction management
    - Many more…

What is Spring?

- Fully portable across application servers
  - Core container can run in *any* environment, not just an application server
  - Many applications don't *need* an application server: just a web container
- Runs in J2SE 1.3 and above
  - Can take advantage of 1.4 features automatically

aosd.05 International Conference on Aspect-Oriented Software Development

IoC Using Setter Injection

```
public class ServiceImpl implements Service {
  private int timeout;
  private AccountDao accountDao;

   public void setTimeout(int timeout) {
      this.timeout = timeout;
   }

   public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
   }

   // Business methods from Service
…

<bean id="service" class="com.mycompany.service.ServiceImpl">
  <property name="timeout"><value>30</value></property>
  <property name="accountDao"><ref local="accountDao"/></property>
</bean>
```

IoC Using Constructor Injection

```
public class ServiceImpl implements Service {
  private int timeout;
  private AccountDao accountDao;

   public ServiceImpl (int timeout, AccountDao accountDao)
  {
       this.timeout = timeout;
       this.accountDao = accountDao;
   }

   // Business methods from Service


<bean id="service"
  class="com.mycompany.service.ServiceImpl">
  <constructor-arg><value>30</value></constructor-arg>
  <constructor-arg><ref local="accountDao"/></constructor-
  arg>
</bean>
```

Traditional approach

- Hard-code use of *new*
  - What if something changes?
  - How do we externalize configuration from Java code, important if things change
- Use a custom factory
  - More code to write in the application
  - Just move the hard-coding or ad-hoc parameterization one step farther away
- … "Service Locator" approach traditional in J2EE

Benefits of Dependency Injection

- Unit testable
- Dependencies are explicit
- Consistent
- Can wire up arbitrarily complicated graphs
- You don't need to write plumbing code
- Pluggability
  - Reduces cost of programming to interfaces to zero

Spring AOP

- Designed for usability
- Designed with J2EE in mind
- Proxy-based
  - Uses runtime-generated proxies to add concerns
  - Performance is NOT the key driver
- Supports a declarative and programmatic configuration model
- Ideal partner to IoC
  - Any Spring bean can be transparently advised
  - Advice, pointcuts and introductions can be managed and configured using IoC as well

Spring AOP Library

- Comprehensive set of pre-built aspects
  - Transaction management
  - Security (with Acegi)
  - Tracing and debugging
  - Remoting proxies
    - JAX-RPC
    - Hessian
    - Burlap
    - HTTP Invoker
  - Performance monitoring
  - Framework Internals
    - Lock management
    - JMX proxies
    - EJB proxies
    - Concurrency throttling

Case Study: Transaction Management

- Example of AOP solving a real problem in enterprise middleware
- Consistent abstraction
  - `PlatformTransactionManager`
  - Does not reinvent transaction manager
  - Choose between JTA, JDBC, Hibernate, JDO etc *with simple changes to configuration not Java code*
  - No more rewriting application to scale up from JDBC, Hibernate or JDO local transactions to JTA global transactions
  - Use the simplest transaction infrastructure that can possibly work

Programmatic Transaction Management

- Simpler, cleaner API than JTA
  - Exception hierarchy as with DAO
  - No need to catch multiple exceptions without a common base class
  - Unchecked exceptions
- Use the same API for JTA, JDBC, Hibernate etc.
- Write once have transaction management anywhere

Declarative Transaction Management

- Most popular transaction management option
- Built on same abstraction as programmatic transaction management
- Declarative transaction management for any POJO, without EJB: even without JTA (single database)
- More flexible than EJB CMT
  - Declarative *rollback rules*: roll back on MyCheckedException
  - Supports nested transactions and savepoints if the underlying resource manager does
- **Non-invasive: Minimizes dependence on the container**
  - No more passing around EJBContext

AOP in Transaction Management

- Uses advised proxies behind the scenes
- Users don’t see AOP
- Provides the necessary infrastructure to enhance object behaviour at runtime
- Removes the need for a deploy-time code generation

Make ServiceImpl POJO Transactional

```
public class ServiceImpl implements Service {
    private int timeout;
    private AccountDao accountDao;

    public void setTimeout(int timeout) {
                this.timeout = timeout;
    }

    public void setAccountDao(AccountDao accountDao) {
                this.accountDao = accountDao;
    }

    public void doSomething() throws ServiceWithdrawnException {
    }
}

<bean id="serviceTarget" class="com.mycompany.service.ServiceImpl">
    <property name="timeout"><value>30</timeout></property>
    <property name="accountDao"><ref local="accountDao"/></property>
</bean>
```

Make ServiceImpl Transactional

- Create an advised proxy to the service implementation:

```
<bean id="service"
   class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"/>
   <property name="target">
            <ref local="serviceTarget"/>
   </property>
   <property name="transactionManager">
            <ref local="localTransactionManager"/>
   </property>
   <property name="transactionAttributes">
            <props>
                        <prop key="do*">
                           PROPAGATION_REQUIRED,-ServiceWithdrawnException
                        </prop>
            </props>
   </property>
</bean>
```

Make ServiceImpl Transactional

- Rollback rule means that we don't need to call `setRollbackOnly()`
  - Spring also supports programmatic rollback
- Can run this from a JUnit test case
  - Doesn't depend on a heavyweight container
- Can work with JTA, JDBC, Hibernate, JDO, iBATIS transactions…
  - Simply change definition of transaction manager

Make ServiceImpl Transactional

- Don’t actually need this much XML per transactional object
- Alternative approaches, simpler in large applications:
  - Use “auto proxy creator” to apply similar transaction attributes to multiple beans
  - Use a “template” bean definition to capture common properties (transactionManager, transaction attributes)
  - Use metadata (annotations) or another pointcut approach to apply transactional behaviour to multiple classes

AOP in Spring Summary

- Spring is:
  - Framework for simplifying J2EE
  - Simple introduction to AOP
  - Solving real world problems today
- AOP is integral to Spring
  - Many framework internals build on AOP
  - Many external features use AOP behind the scenes

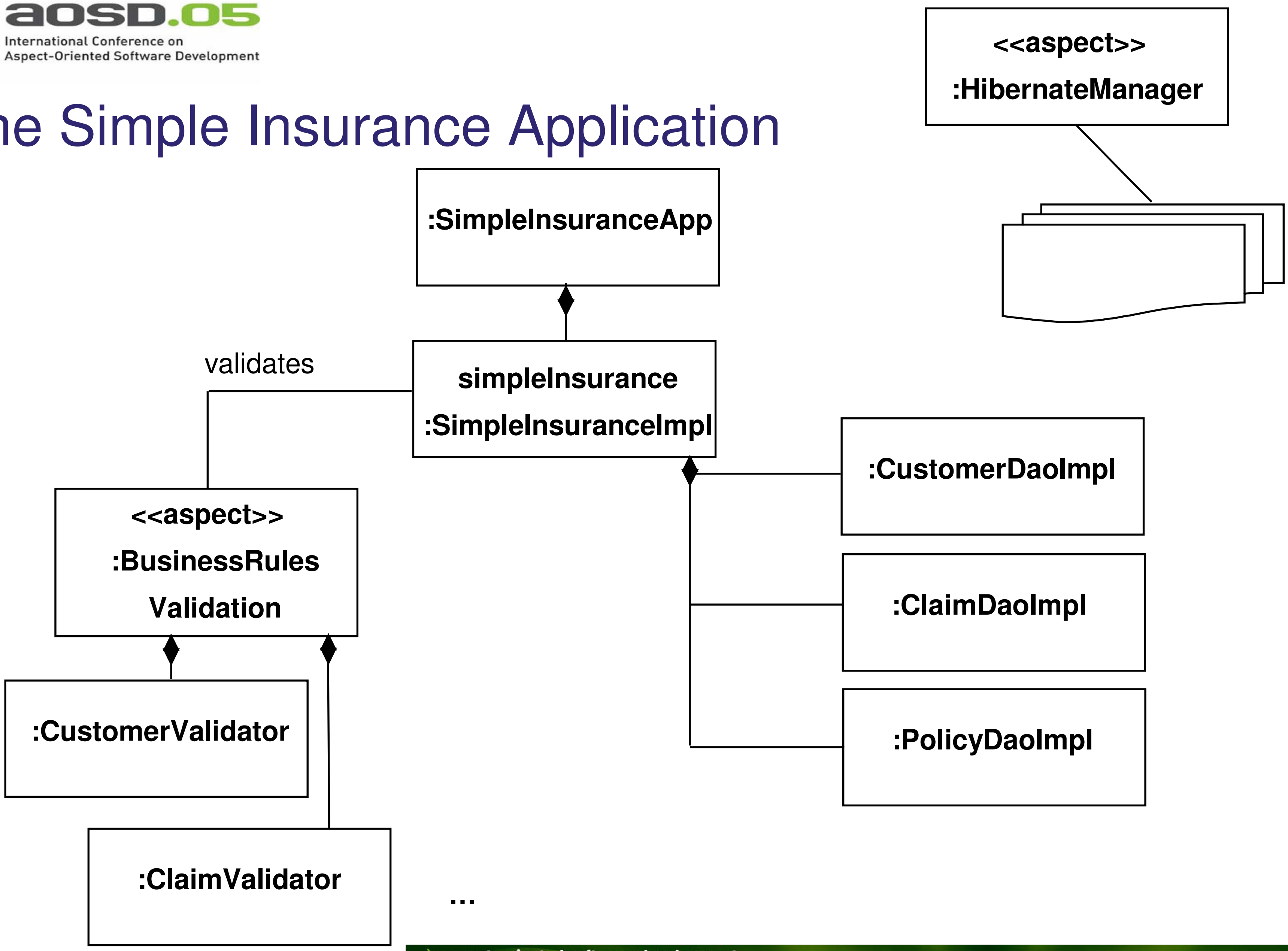
Spring AOP and AspectJ

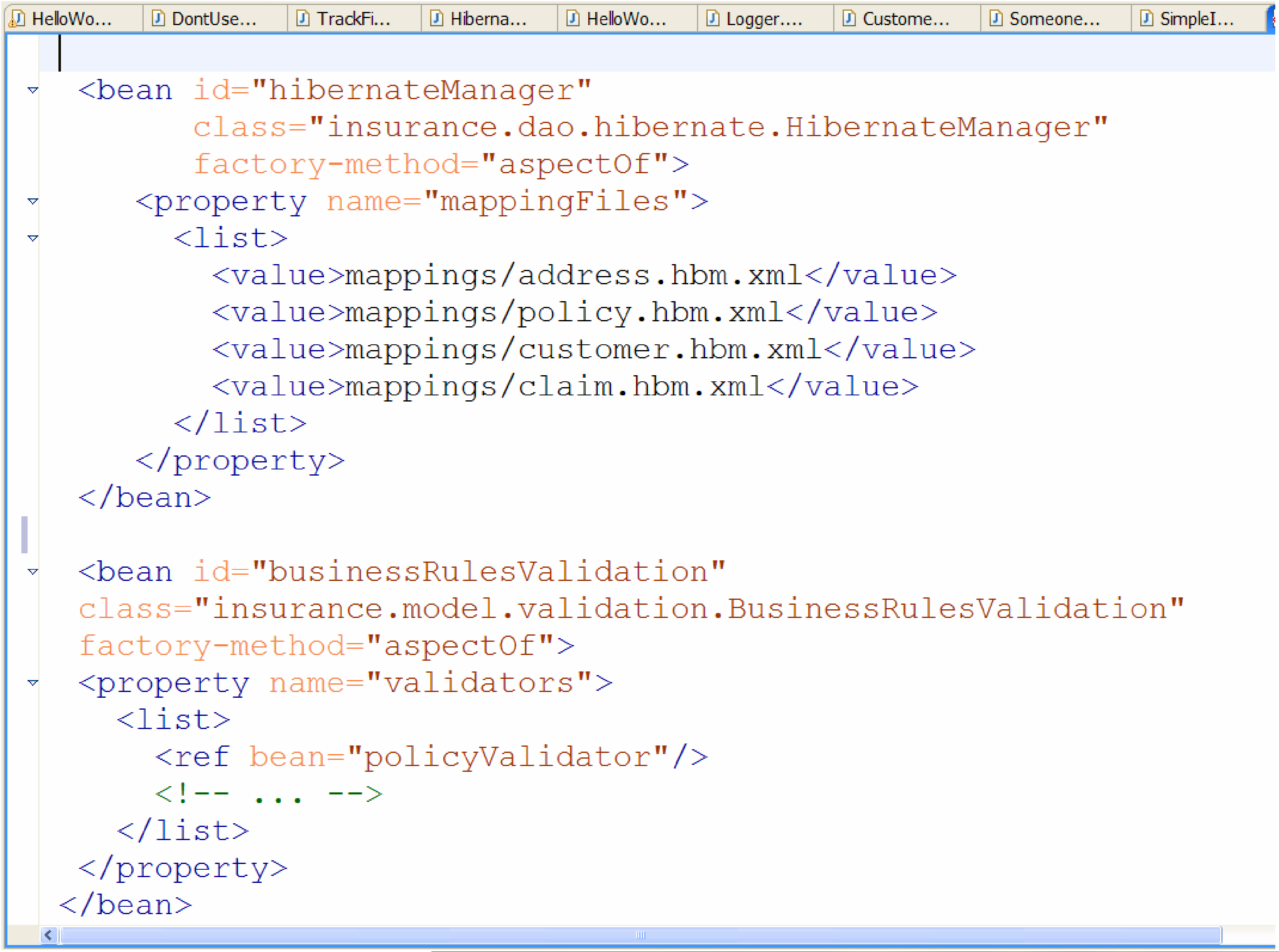
- Spring AOP well suited to
  - Coarse grained application
  - Enterprise services
  - Working with Spring beans
- AspectJ well suited to
  - Fine grained application
  - Aspect-oriented programming
- Why not use them together?
  - …

Dependency Injection and Aspects

- Aspects are a first class part of your system
  - Why wouldn't you want to configure them just like any other component in the design?
  - This is especially true of infrastructure/auxiliary aspects
- AspectJ aspects can easily be configured by Spring just like any other Spring bean

The Simple Insurance Application





```
<bean id="hibernateManager"
      class="insurance.dao.hibernate.HibernateManager"
      factory-method="aspectOf">
   <property name="mappingFiles">
     <list>
       <value>mappings/address.hbm.xml</value>
       <value>mappings/policy.hbm.xml</value>
       <value>mappings/customer.hbm.xml</value>
       <value>mappings/claim.hbm.xml</value>
     </list>
   </property>
</bean>

<bean id="businessRulesValidation"
class="insurance.model.validation.BusinessRulesValidation"
factory-method="aspectOf">
<property name="validators">
  <list>
    <ref bean="policyValidator"/>
    <!-- ... -->
  </list>
</property>
</bean>
```

Non-singleton aspects

- Singleton aspects fit well with the Spring bean model
- Other aspect instantiation models are more complex
  - Separate instantiation and configuration

Instantiation model	**aspectOf() signature**
singleton	aspectOf()
perthis	aspectOf(Object)
pertarget	aspectOf(Object)
percflow	aspectOf() (in cflow)
percflowbelow	aspectOf() (in cflowbelow)
pertypewithin	aspectOf(Class)

Non-singleton aspects

- Let AspectJ manage the aspect instantiation
- Let Spring manage the configuration
- Basic strategy…
  - after returning… from the initialization of an aspect bean
    - ask the Spring BeanFactory to configure it

@Bean

```
@Retention(RetentionPolicy.RUNTIME)
@interface Bean {
   String value default “”;
}
```

@Bean usage

```
@Bean("SessionManager")
public aspect SessionManager percflow(session()) {
  private Session session;
  private SessionFactory factory;
  public void setSessionFactory(SessionFactory factory) {
    this.factory = factory;
  }

  pointcut session() : …;

  before() : session() {
     session = factory.beginSession();
  }
  after() : session() { session.close(); }
}
```

Configuration aspect

```
public abstract aspect BeanConfigurator {

  pointcut beanCreation(Bean beanAnnotation,
                                    Object beanInstance) :
      initialization((@Bean *).new(..)) &&
      @this(beanAnnotation) &&
      this(beanInstance);

  after(Bean beanAnnotation, Object beanInstance) returning :
    beanCreation(beanAnnotation,beanInstance)
  {
    String beanName = beanAnnotation.value();
    if (beanName.equals(“”)) beanName = beanInstance.getClass().getName();
    configureBean(beanInstance,beanName);
  }

  protected abstract void configureBean(Object bean,
                                                          String beanName);
}
```

Spring Configuration…

```
public aspect SpringBeanConfigurator extends BeanConfigurator
implements BeanFactoryAware {

  private AutowireCapableBeanFactory beanFactory;
  public void setBeanFactory(BeanFactory factory) {
    this.beanFactory = (AutowireCapableBeanFactory) factory;
  }

  protected void configureBean(Object bean, String beanName) {
    beanFactory.applyBeanPropertyValues(bean,beanName);
  }

}
```

Spring Configuration…

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean name=“SpringBeanConfigurator"
           class="org.aspectj.spring.SpringBeanConfigurator"
           factory-method="aspectOf"/>

   <bean name=“SessionManager” class=“org.xyz.SessionManager”>
     <property name=“sessionFactory”>
       <ref bean=“SessionFactory”/>
     </property>
   </bean>

   <bean name=“SessionFactory” … />
</beans>
```

Aspects and Dependency Injection

- You can obviously use the @Bean annotation on any type
    - Not just aspects
- Can also use aspects to perform dependency injection directly
- Let’s look at two examples:
    - Context IoC
    - Per-execution dependency injection

Context IoC

```
public interface INeedInsuranceDAOs {
  void setCustomerDAO(CustomerDAO custDAO);
  void setClaimDAO(ClaimDAO claimDAO);
  void setPolicyDAO(PolicyDAO policyDAO);
}
```

Implemented by any type that needs access to the insurance DAOs…

Context IoC

```
public aspect HibernateManager {

  private ClaimDAO claimDao;
  private CustomerDAO custDao;
  private PolicyDAO policyDao;

  pointcut needsDAOsCreation(INeedInsuranceDAOs inNeed)
    : initialization(INeedInsuranceDAOs+.new(..)) &&
      this(inNeed);

  after(INeedInsuranceDAOs inNeed) returning :
    needsDAOsCreation(inNeed) {
    inNeed.setClaimDAO(claimDao);
    inNeed.setCustomerDAO(custDao);
    inNeed.setPolicyDAO(policyDao);
  }
}
```

Per-execution DI

```
class HibernateDao {
  private Session session;
  public void setSession(Session session) {this.session = session;}
  protected Session getSession() { return session; }
}

public class CustomerDao extends HibernateDao{

  public void insertCustomer(Customer cust) {
     getSession().save(cust);
  }

}
```

Per-execution DI

```
aspect … {

    …
    pointcut hibernateTransaction(HibernateDao dao) :
      execution(* HibernateDao+.*(..)) && this(dao) &&
      !within(HibernateDao);

    before(HibernateDao dao) : hibernateTransaction(dao) {
      dao.setSession(session);
    }

}
```

Futures for Spring/AspectJ integration

- Shared pointcut language
- Out-of-the-box support for @Bean
- Improvements to Spring XML Schema for aspects
- Joint work on aspect libraries
  - Make more of the Spring aspect libraries easily accessible to AspectJ users
  - Potentially additional AspectJ-only Spring aspects for finer-grained scenarios

Library Example: Acegi

- The Spring Acegi security library has AspectJ support built in

```
<bean id="bankManagerSecurityInterceptor"
    class="net.sf.acegisecurity.intercept.method.aspectj.AspectJSecurityInterceptor">

   <property name="validateConfigAttributes"><value>true</value></property>
   <property name="authenticationManager">
     <ref bean="authenticationManager"/>
   </property>
   <property name="accessDecisionManager">
    <ref bean="accessDecisionManager"/>
   </property>
   <property name="runAsManager">
    <ref bean="runAsManager"/>
   </property>
   <property name="afterInvocationManager">
    <ref bean="afterInvocationManager"/>
   </property>
   <property name="objectDefinitionSource">
    <value>
    net.sf.acegisecurity.context.BankManager.delete*=ROLE_SUPERVISOR,RUN_AS_SE
    RVER
              net.sf.acegisecurity.context.BankManager.getBalance=ROLE_TELLER,
              ROLE_SUPERVISOR,BANKSECURITY_CUSTOMER,RUN_AS_SERVER
    </value>
   </property>
</bean>
```

Library Example: Acegi

```
public aspect BankingSecurityManager extends
   AcegiSecurityManager {

  protected pointcut securedOperations() :
    execution(* BankManager+.*(..));

}

<bean id=“bankingSecurityManager"
          class=“BankingSecurityManager"
          factory-method="aspectOf">
   <property name="securityInterceptor">
      <ref bean=“bankManagerSecurityInterceptor"/>
   </property>
</bean>
```

Summary

- Spring has a coarse-grained AOP framework
  - Used for enterprise services
  - And also extensively in the construction of Spring itself
- AspectJ and Spring are complementary
  - DI of aspects, aspects for DI
- Ongoing collaboration to increase integration between Spring and AspectJ
  - Pointcut language, configuration, libraries