TMA Training Center (TTC)

Spring Framework

Course	Advanced Java
Trainer	Thanh Pham
Designed by	Kien Trinh (Modified)
Last updated	March 10, 2013

Contents

Course Objectives

- Have knowledge about spring framework
- Understand concepts of aspect-oriented and dependency injection programming.
- Use the Hibernate template to integrate Hibernate and Spring.
- Apply Spring framework in real project.

Introduction

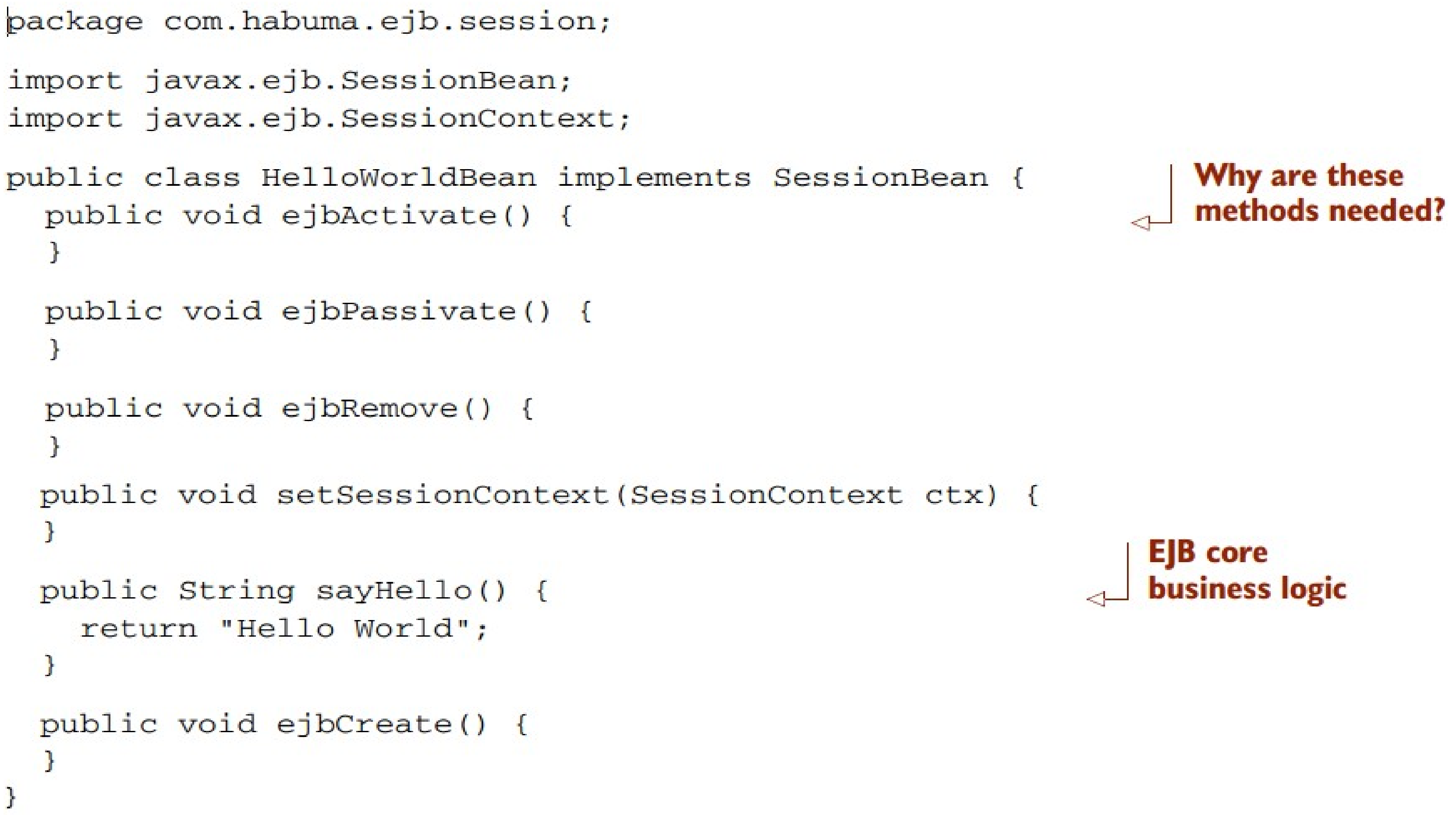
- Spring is an open source framework
- Any Java application can benefit from Spring in terms of simplicity, testability, and loose coupling.
- Spring is a lightweight dependency injection and aspect-oriented container framework
- Spring’s fundamental mission: **Spring simplifies Java development**.

Simplify Java development

- Lightweight and minimally invasive development with plain old Java objects (POJOs)
- Loose coupling through dependency injection and interface orientation
- Declarative programming through aspects and common conventions
- Boilerplate reduction through aspects and templates

Unleashing the power of POJOs

EJB force you to implement things that we don't need ☹

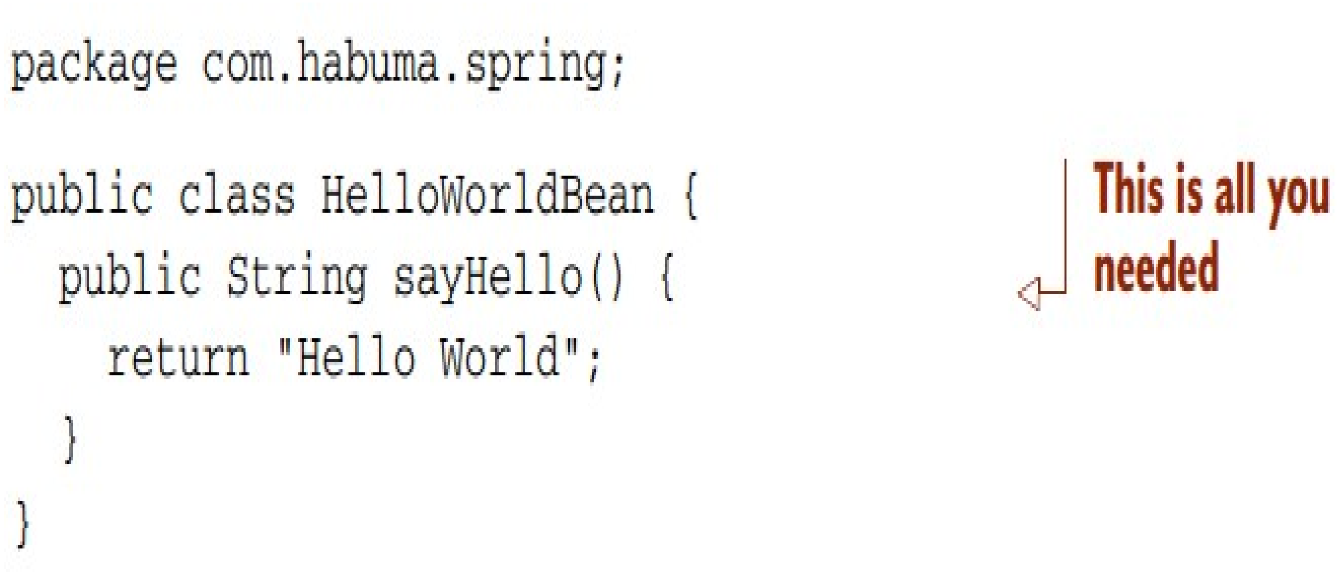


```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {
  public void ejbActivate() {
  }

  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }

  public void setSessionContext(SessionContext ctx) {
  }

  public String sayHello() {
    return "Hello World";
  }

  public void ejbCreate() {
  }
}
```

Why are these methods needed?
EJB core business logic

Unleashing the power of POJOs

EJB force you to implement things that we don’t need ☹



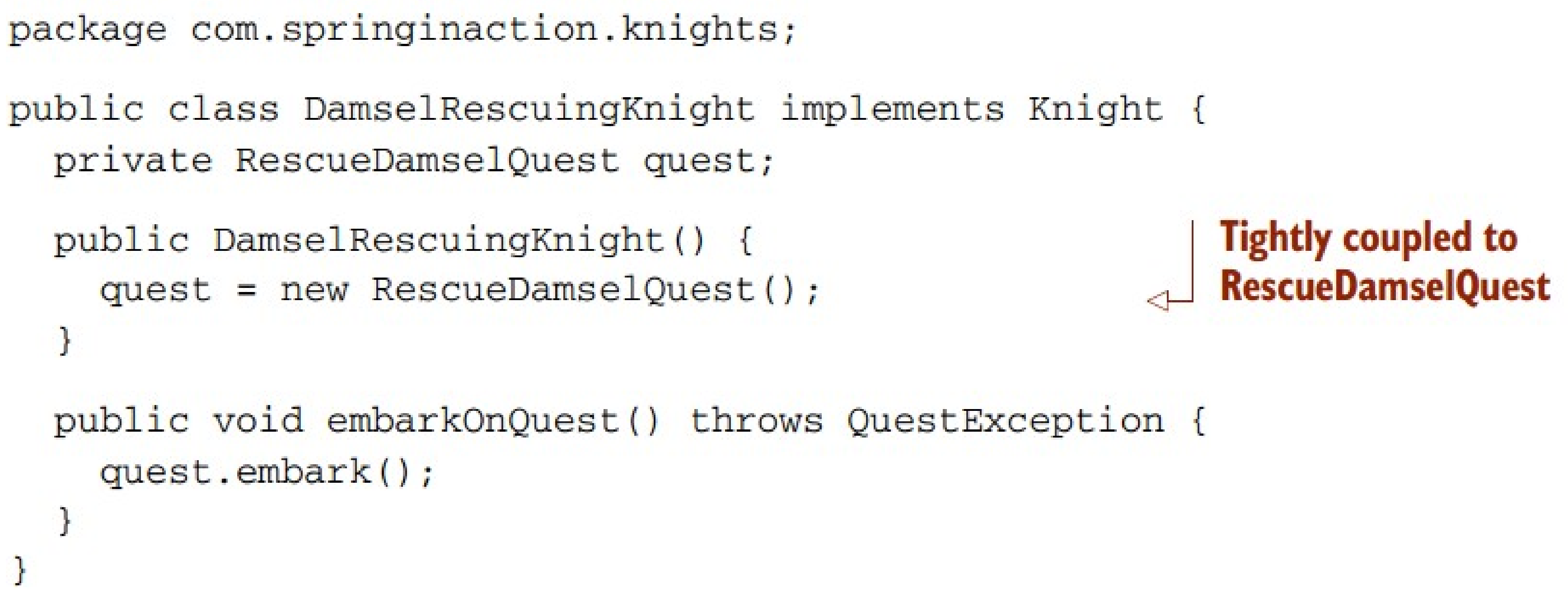
```
package com.habuma.spring;

public class HelloWorldBean {
  public String sayHello() {
    return "Hello World";
  }
}
```

This is all you needed

You just need to focus on your business…and leave the rest to Spring

Injecting dependencies

Tightly coupled code. Very bad design…



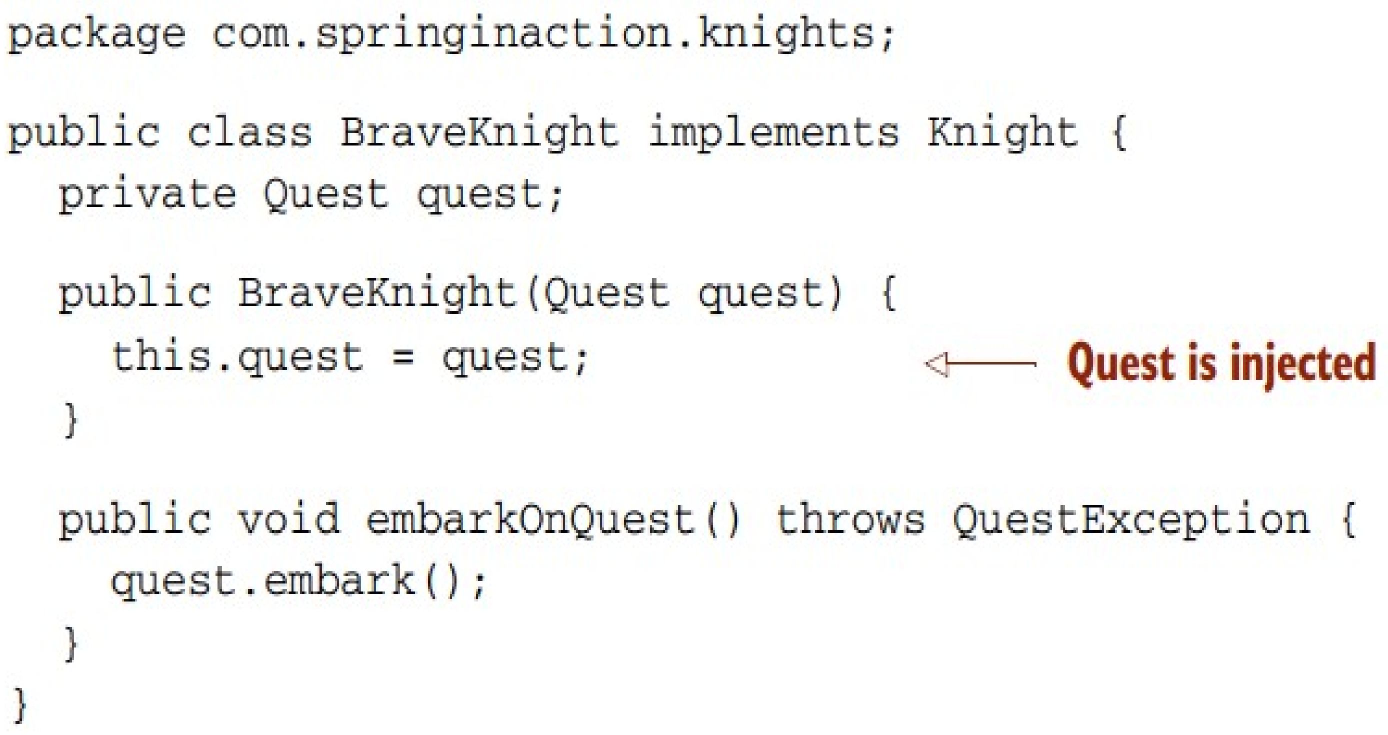
```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    quest = new RescueDamselQuest();
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

Tightly coupled to RescueDamselQuest

Tightly coupled code is difficult to test, difficult to reuse, difficult to understand, and typically exhibits “whack-a-mole” bug behavior

Injecting dependencies

Flexibility code.... ☺



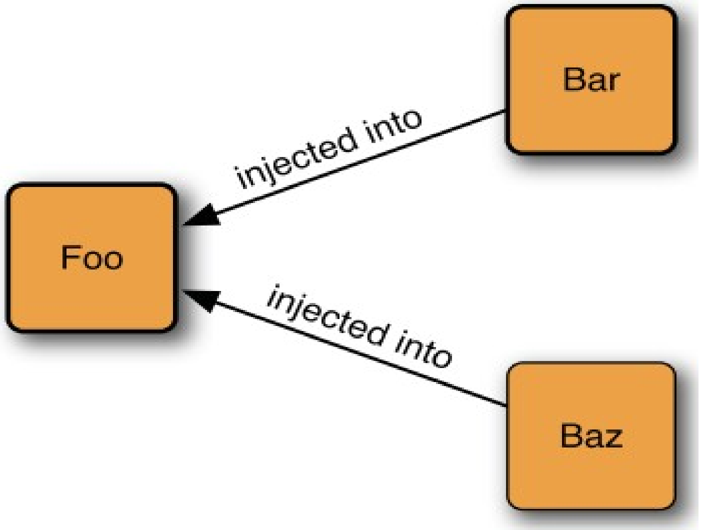
```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;

  public BraveKnight(Quest quest) {
    this.quest = quest;
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

Quest is injected

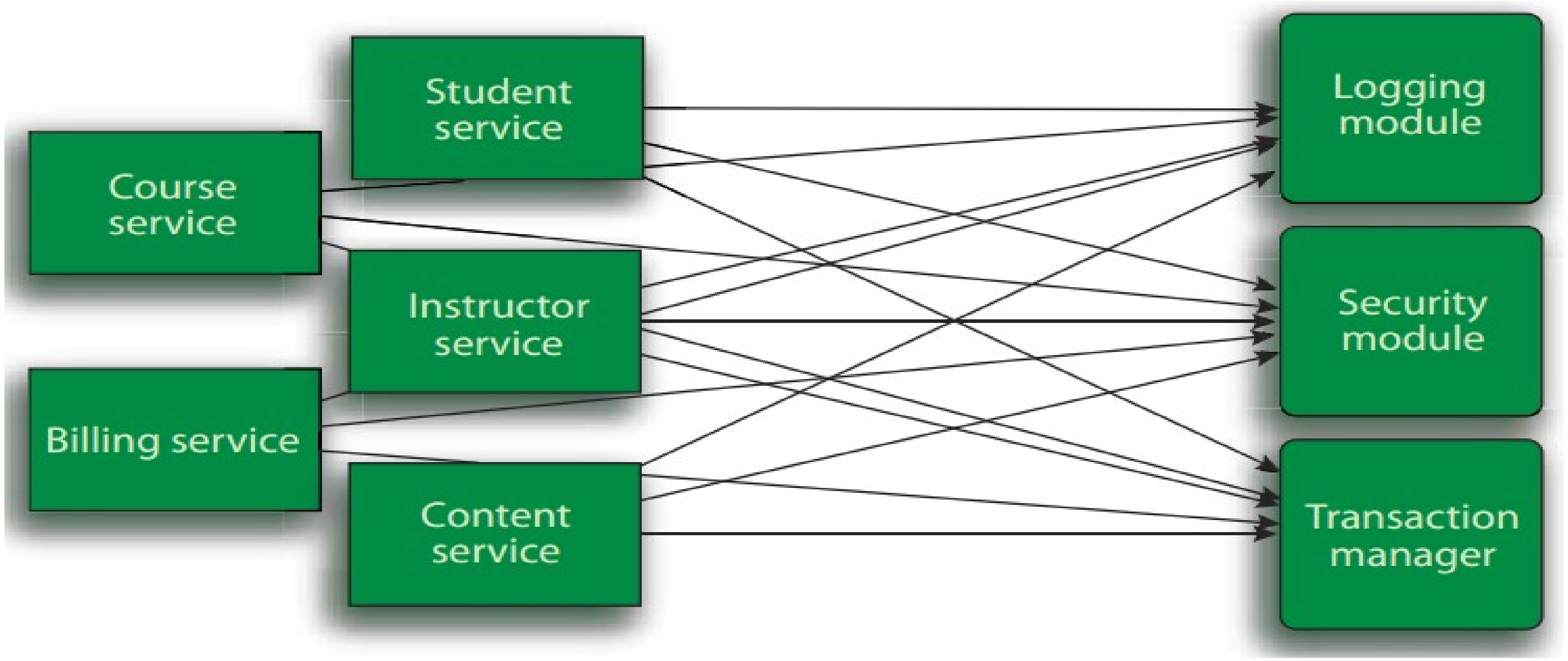
BraveKnight isn't coupled to any specific implementation of Quest

Injecting dependencies: The idea

- Objects are given their dependencies at creation time by some third party that coordinates each object in the system.
- Objects aren’t expected to create or obtain their dependencies —dependencies are injected into the objects t
- **Loose coupling**: the key benefit of DI



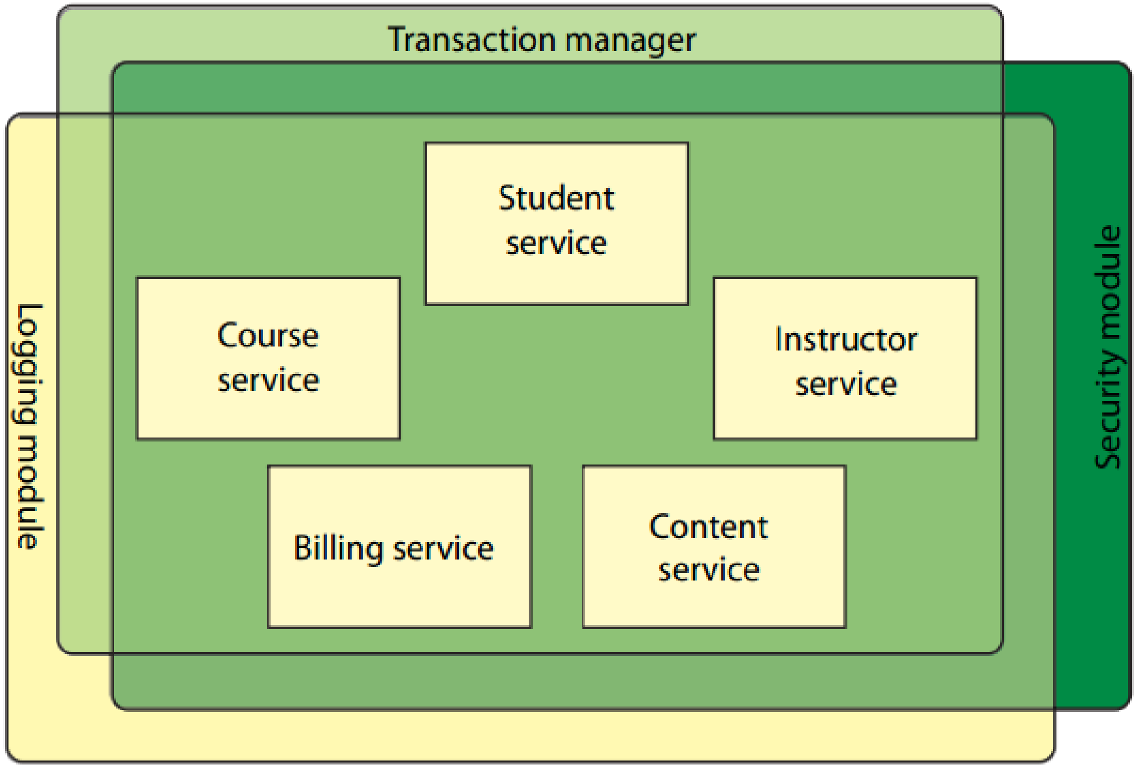
Applying aspects



How to apply Logging Module, Security Module and Transaction Manager to all of courses and services?

Applying aspects

- DI makes it possible to tie software components together loosely
- **Aspect-oriented programming enables you to capture fu ur applicatior**



Eliminating boilerplate code with templates

Have you ever written some code and then felt like you'd already written the same code before?

1. Prepare connection
2. **Prepare statement**
3. **Customize result**
4. Cleanup connection/stuffs

Why do we need these stuffs?

Eliminating boilerplate code with templates

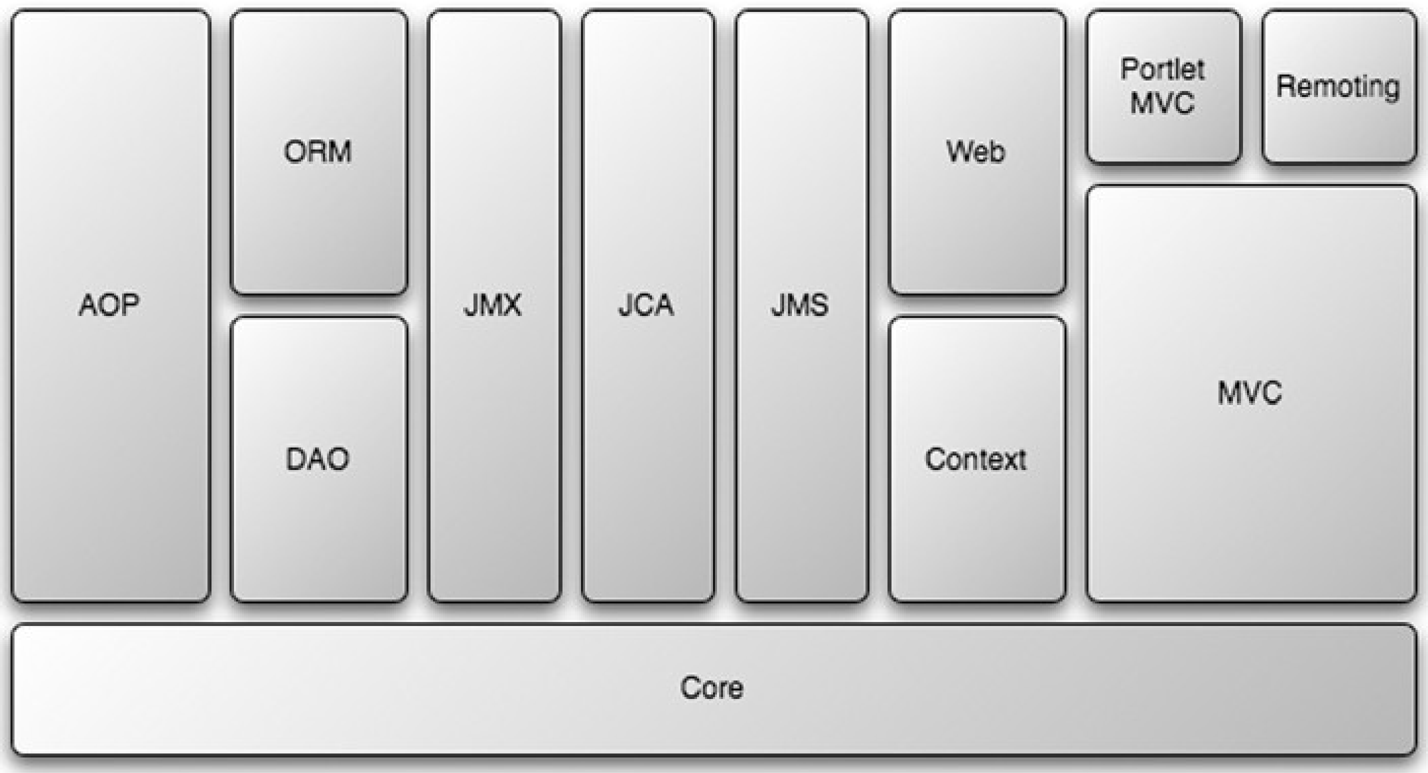
- **Spring seeks to eliminate boilerplate code by encapsulating it in templates.** Spring's JdbcTemplate makes it possible to perform database operations without all of the ceremony required by traditional JDBC.

~~1. Prepare connection~~
2. **Prepare statement**
3. **Customize result**
~~4. Cleanup connection/stuffs~~

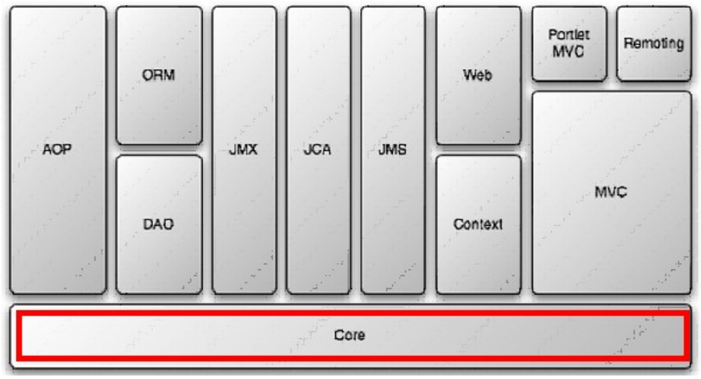
Architecture Overview

- Understand the role of the main components in spring

Architecture Overview

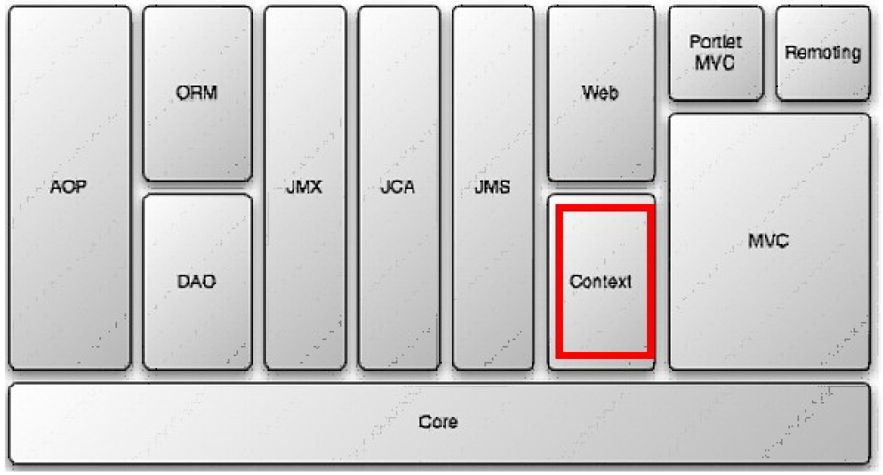


The core container



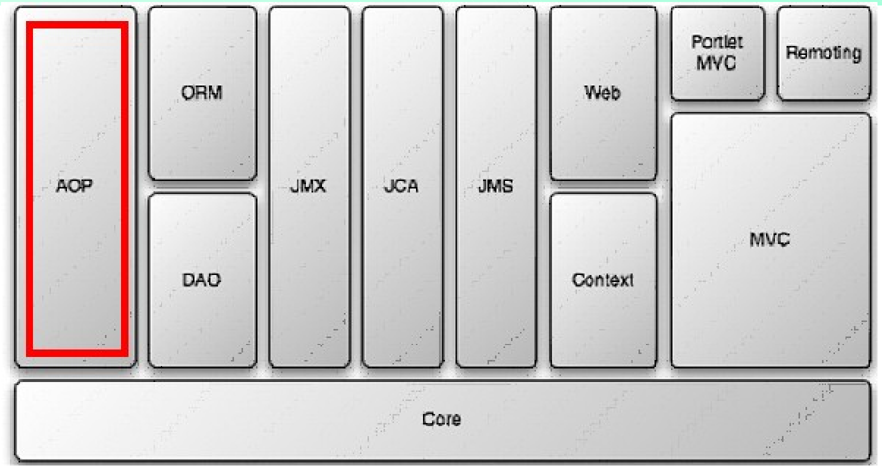
- The container defines how beans are created, configured, and managed.
- IOW, core module is the core container will create the objects, wire them together, configure them, and manage their complete lifecycle from cradle to grave.

Application context module



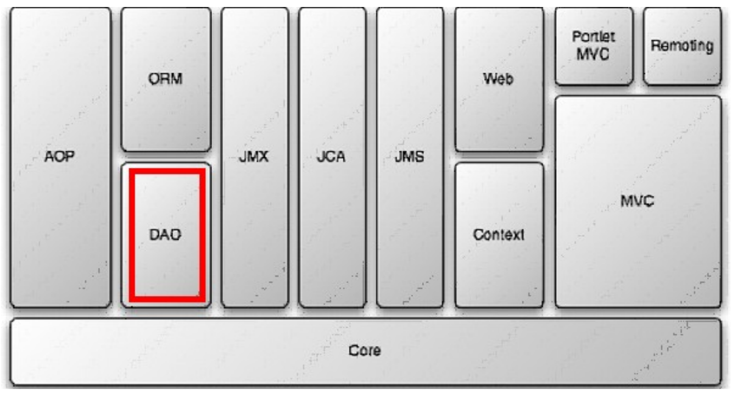
- The core module’s BeanFactory makes Spring a container, but the context module is what makes it a framework.
- Support for internationalization (I18N) messages, application lifecycle events, and validation. In addition, this module supplies many enterprise services such as email, JNDI access, EJB integration, remoting, and scheduling.

AOP



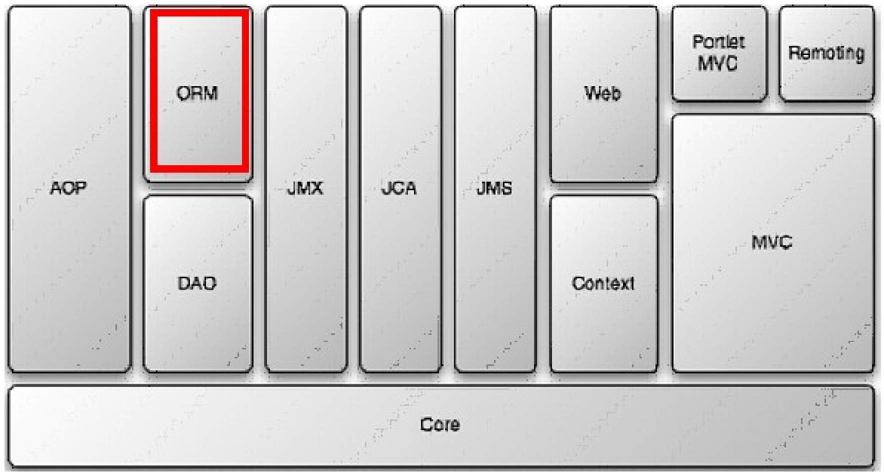
- ***Spring's AOP module***
  - Spring provides rich support for aspect-oriented programming in its AOP module.
  - This module serves as the basis for developing your own aspects for your Spring enabled application. Like DI, AOP supports loose coupling of application objects.
  - With AOP, however, applicationwide concerns (such as transactions and security) are decoupled from the objects to which they are applied.

JDBC abstraction and the DAO module



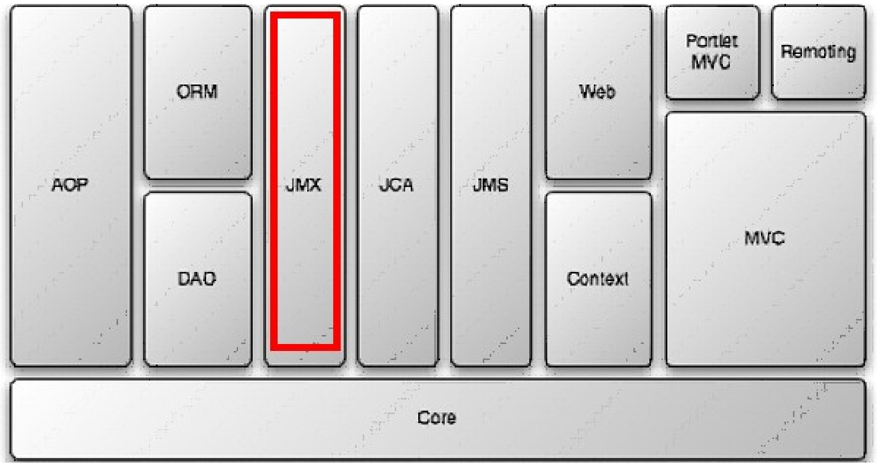
- Spring's JDBC and Data Access Objects (DAO) module abstracts away the boilerplate code/reused code so that you can keep your database code clean and simple, and prevents problems that result from a failure to close database resources.
- In addition, this module uses Spring's AOP module to provide transaction management services for objects in a Spring application.

ORM(Object-relational mapping)



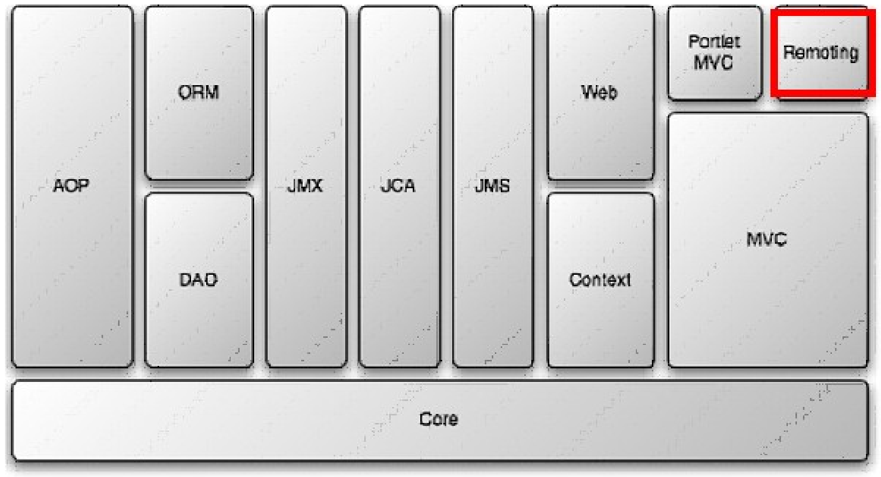
- Spring's ORM support builds on the DAO support, providing a convenient way to build DAOs for several ORM solutions.
- Spring doesn't attempt to implement its own ORM solution, but does provide hooks into several popular ORM frameworks, including Hibernate, Java Persistence API, Java Data Objects, and iBATIS SQL Maps.
- Spring's transaction management supports each of these ORM frameworks as well as JDBC.

JMX



- ***Java Management Extensions (JMX)***
  - Spring's JMX module makes it easy to expose your application's beans as JMX MBeans.

Remoting



- Spring’s remoting support enables you to expose the functionality of your Java objects as remote objects.
- Or if you need to access objects remotely, the remoting module also makes simple work of wiring remote objects into your application as if they were local POJOs.

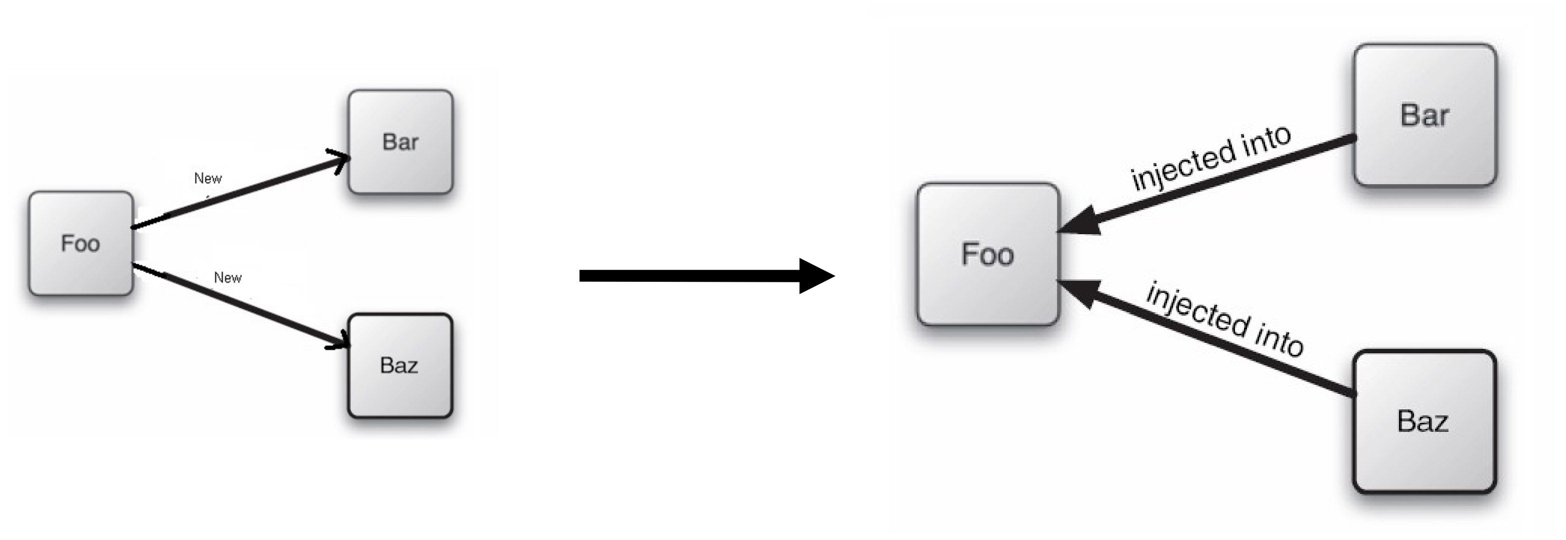
Where Can I Start?

- Download:
  - http://www.springsource.org/download/community
- Documentation and samples:
  - http://www.springsource.org/get-starte
- Common Logging:
  - http://commons.apache.org/proper/commons-logging//download_logging.cgi

Inversion of Control / Dependency Injection

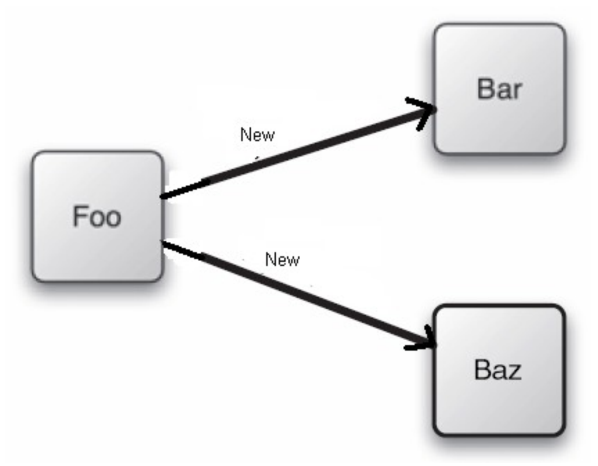
- Understand what IOC/ID is in spring

IOC/DI



Understanding IOC/DI

- Each object is responsible for obtaining its own references to the objects it collaborates with (its dependencies).
- This can lead to highly coupled and hard-to-test code.



Example

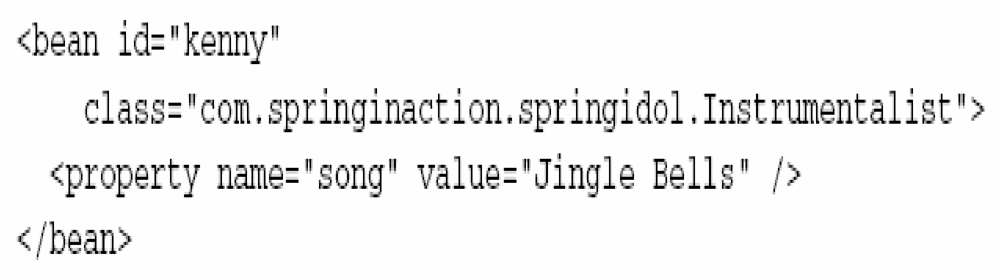
- Student Management
  - Address
    - String province
    - String district
  - Student
    - Name - String
    - Age - int
    - Address – Address
  - StudentClass
    - Name
    - studentList – List<Student>
    - positionMap – Map<String, List<Integer>>

Injecting Dependencies In Spring

- **Injecting into bean properties**
  - Injecting simple values
  - Referencing other beans
  - Wiring collection
  - Wiring nothing (null)
- Injecting into bean constructor
- Autowire

Injecting Dependencies In Spring

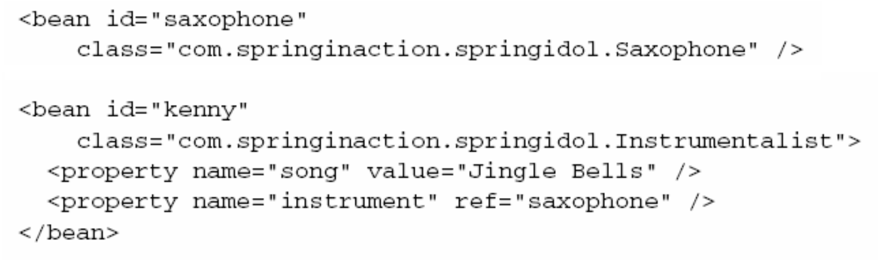
- Injecting into bean properties
  - **Injecting simple values**
  - Referencing other beans
  - Wiring collection
  - Wiring nothing (null)
- Injecting into bean constructor
- Autowire



```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
</bean>
```

Injecting Dependencies In Spring

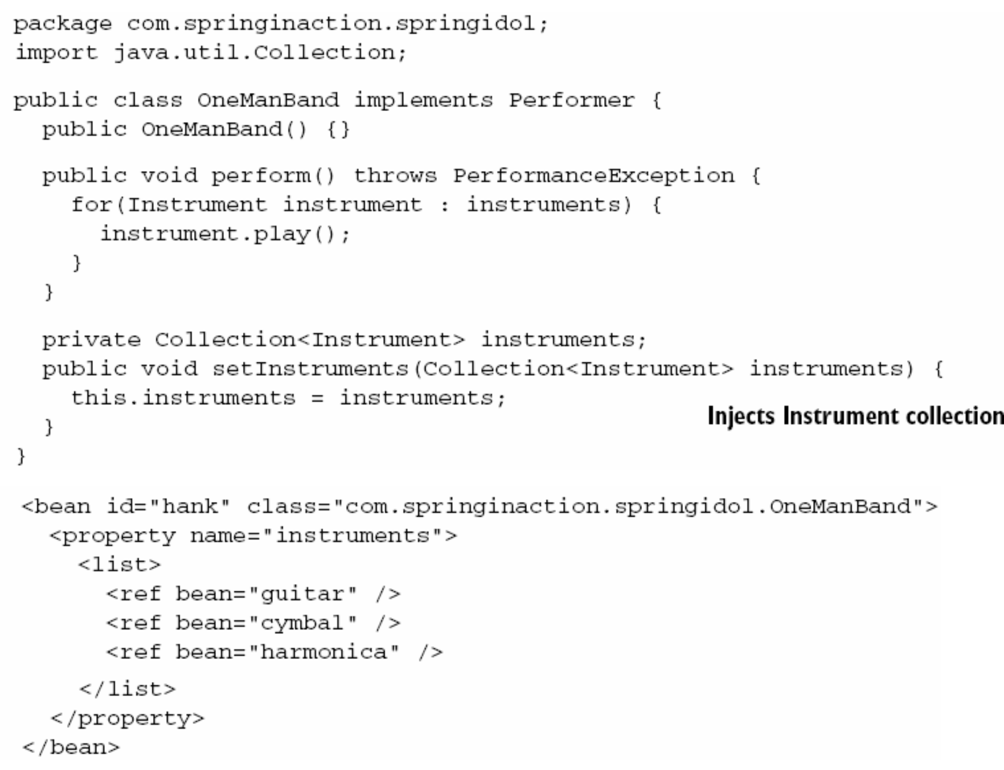
- Injecting into bean properties
  - Injecting simple values
  - **Referencing other beans**
  - Wiring collection
  - Wiring nothing (null)
- Injecting into bean constructor
- Autowire



```
<bean id="saxophone"
    class="com.springinaction.springidol.Saxophone" />

<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument" ref="saxophone" />
</bean>
```

Injecting Dependencies In Spring

- Injecting into bean properties
  - Injecting simple values
  - Referencing other beans
  - **Wiring collection**
  - Wiring nothing (null)
- Injecting into bean constructor
- Autowire



```
package com.springinaction.springidol;
import java.util.Collection;

public class OneManBand implements Performer {
  public OneManBand() {}

  public void perform() throws PerformanceException {
    for(Instrument instrument : instruments) {
      instrument.play();
    }
  }

  private Collection<Instrument> instruments;
  public void setInstruments(Collection<Instrument> instruments) {
    this.instruments = instruments;
  }
}
```

Injects Instrument collection

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
  <property name="instruments">
    <list>
      <ref bean="guitar" />
      <ref bean="cymbal" />
      <ref bean="harmonica" />
    </list>
  </property>
</bean>
```

Injecting Dependencies In Spring

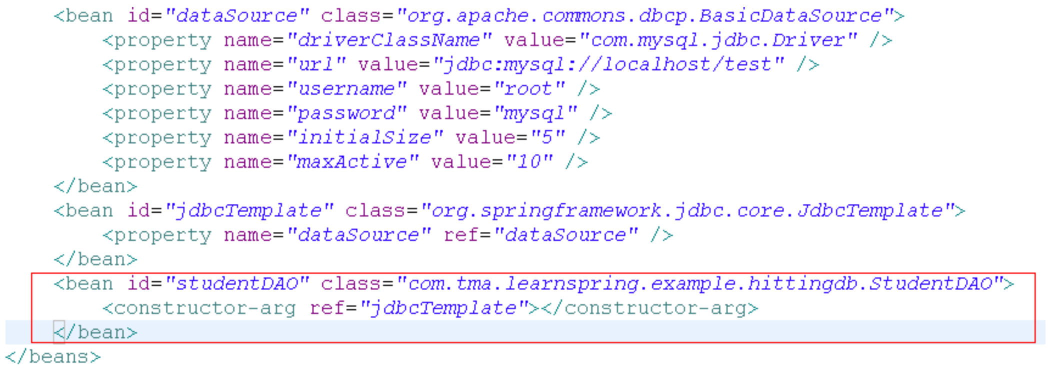
- Injecting into bean properties
  - Injecting simple values
  - Referencing other beans
  - Wiring collection
  - **Wiring nothing (null)**

```
<property name="someNonNullProperty"><null/></property>
```

- Injecting into bean constructor
- Autowire

Injecting Dependencies In Spring

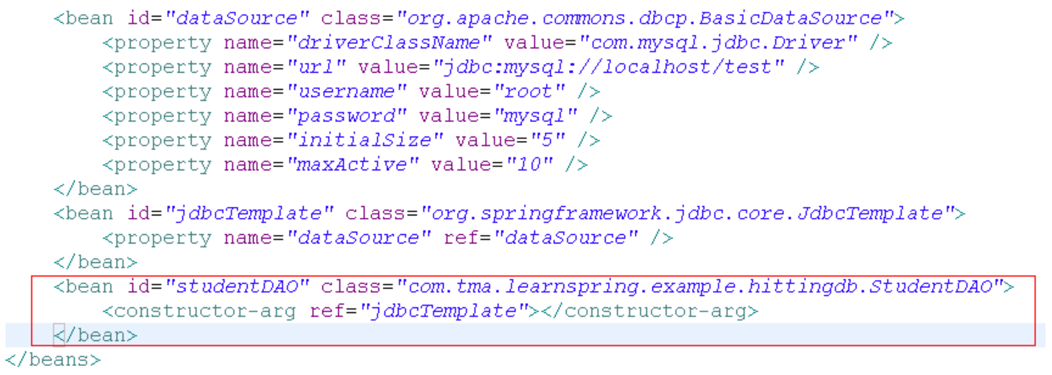
- Injecting into bean properties
  - Injecting simple valu
  - Referencing other beans
  - Wiring collection
  - Wiring nothing (null)
- **Injecting into bean constructor**
- Autowire



```xml
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost/test" />
        <property name="username" value="root" />
        <property name="password" value="mysql" />
        <property name="initialSize" value="5" />
        <property name="maxActive" value="10" />
    </bean>
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="studentDAO" class="com.tma.learnspring.example.hittingdb.StudentDAO">
        <constructor-arg ref="jdbcTemplate"></constructor-arg>
    </bean>
</beans>
```

Injecting Dependencies In Spring

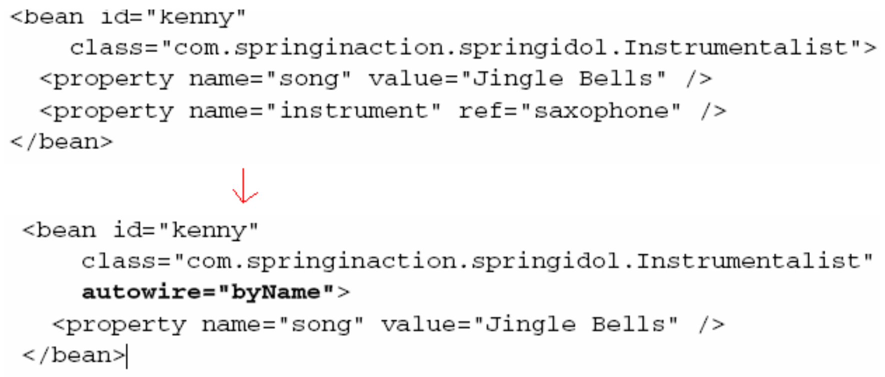
- Injecting into bean properties
  - Injecting simple valu
  - Referencing other beans
  - Wiring collection
  - Wiring nothing (null)
- **Injecting into bean constructor**



```xml
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver" />
      <property name="url" value="jdbc:mysql://localhost/test" />
      <property name="username" value="root" />
      <property name="password" value="mysql" />
      <property name="initialSize" value="5" />
      <property name="maxActive" value="10" />
  </bean>
  <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
      <property name="dataSource" ref="dataSource" />
  </bean>
  <bean id="studentDAO" class="com.tma.learnspring.example.hittingdb.StudentDAO">
      <constructor-arg ref="jdbcTemplate"></constructor-arg>
  </bean>
</beans>
```

Injecting Dependencies In Spring

- Injecting into bean properties
  - Injecting simple values
  - Referencing other beans
  - Wiring collection
  - Wiring nothing (null)
- Injecting into bean constructor
- **Autowire**
  - byName, byType, constructor and autodetect

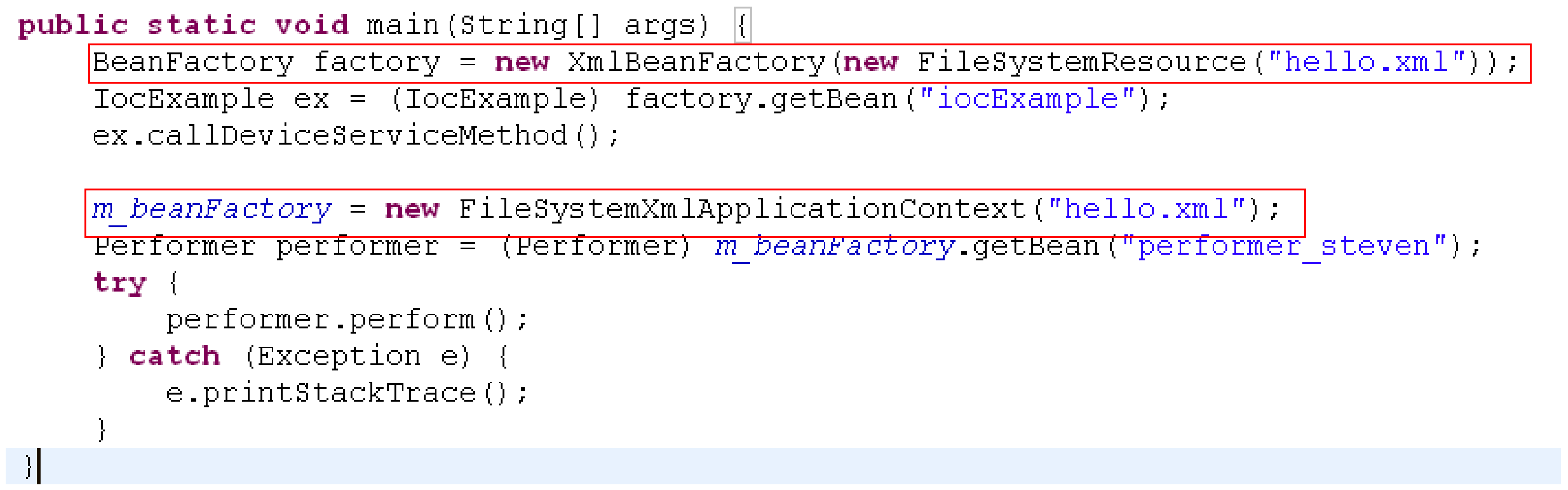


```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist">
  <property name="song" value="Jingle Bells" />
  <property name="instrument" ref="saxophone" />
</bean>
```

↓

```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist"
    autowire="byName">
  <property name="song" value="Jingle Bells" />
</bean>
```

Spring Container



```java
public static void main(String[] args) {
    BeanFactory factory = new XmlBeanFactory(new FileSystemResource("hello.xml"));
    IocExample ex = (IocExample) factory.getBean("iocExample");
    ex.callDeviceServiceMethod();

    m_beanFactory = new FileSystemXmlApplicationContext("hello.xml");
    Performer performer = (Performer) m_beanFactory.getBean("performer_steven");
    try {
        performer.perform();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- BeanFactory(XmlBeanFactory)
  - Knows about many objects within an application
  - Create associations between collaborating objects as they are instantiated.
  - Takes part in the lifecycle of a bean
  - Etc…
- ApplicationContext(FileSystemXmlApplicationContex)
  - More advantages -> is preferred to use.

Summary of IOC/DI

- Dependences are given at creation time
- Loose coupling
- Easy to test

Q&A

AOP

- Understand AOP in spring

AOP

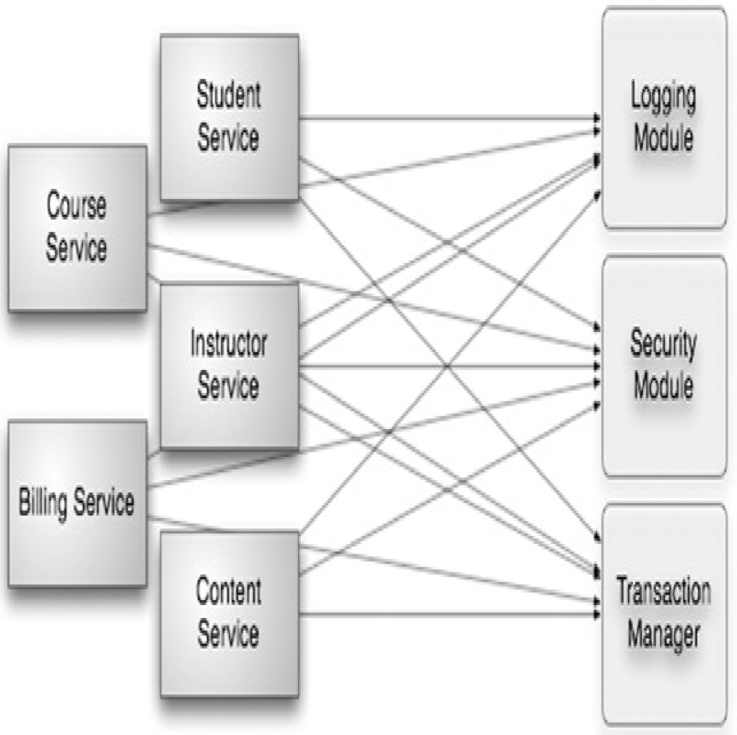
Aspect oriented programming enables you to capture functionality that is used throughout your application in reusable components.

Aspect-oriented programming is often defined as a programming technique that promotes separation of concerns within a software system.

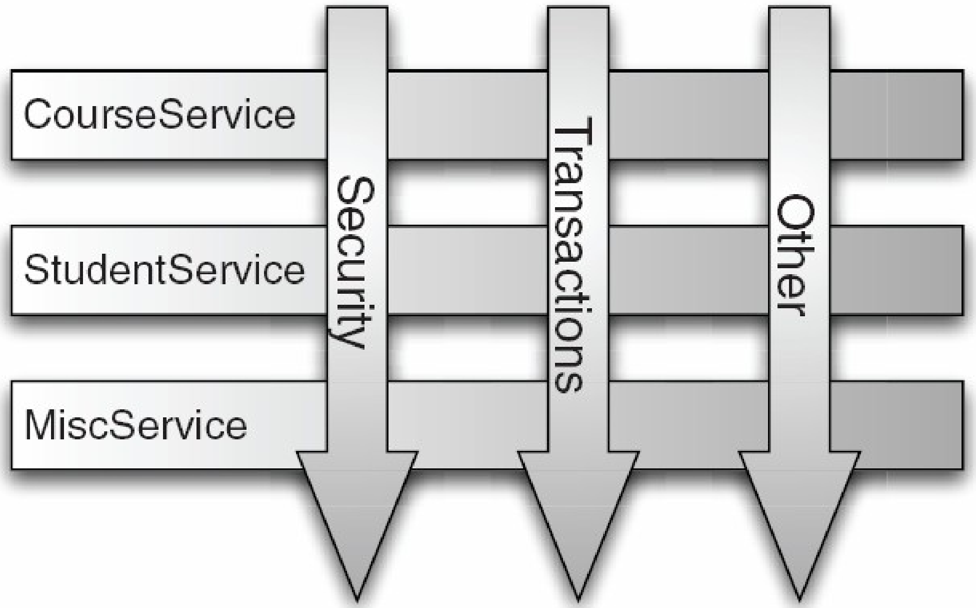
Systems are composed of several components, each responsible for a specific piece of functionality.

Understand AOP

-
  ```
  Public void commonFunction(){
    If(alreadyLogin()){
        beginTransaction();
        try{
         insertFirstThingIntoDb();
         insertDetailOfFirstThingIntoDb();
        }catch(Exception ex){
         rollback();
        }
        endTransaction();
    }
  }
  ```

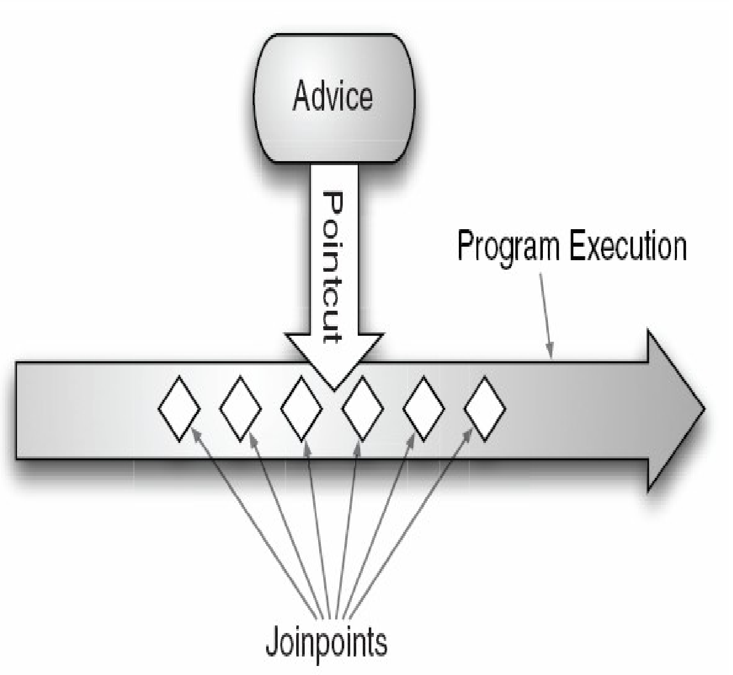


AOP


CourseService
StudentService
MiscService
Security
Transactions
Other

AOP Jargon

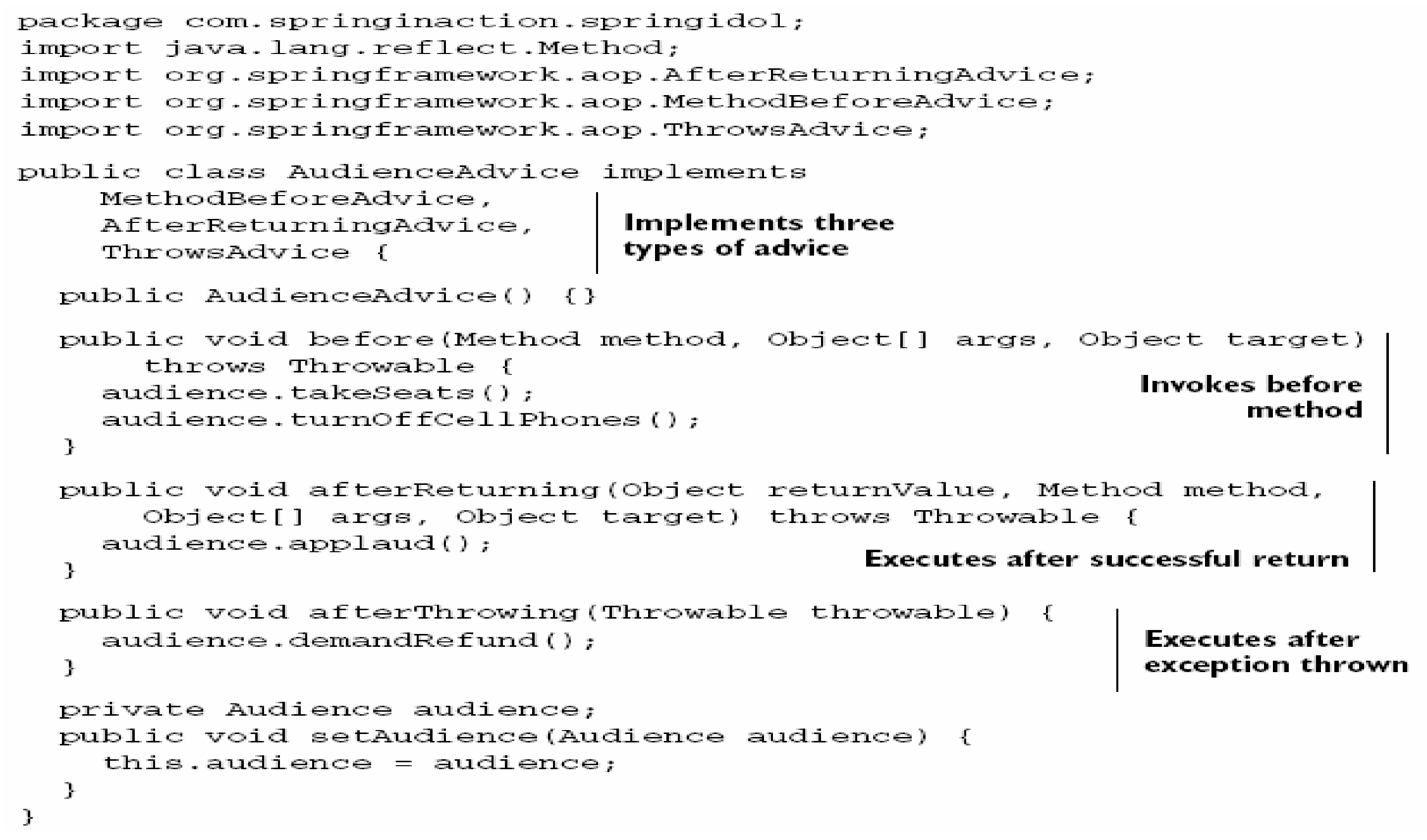
- **Advice**: defines both the *what an aspect does* and the *when* does it (applied before, after or when an exception occurs). MethodBeforeAdvice, AfterReturningAdvice,ThrowsAdvice
- **Pointcut**: defines where, one pointcut definition matches one or more joinpoints. ex: execution(* *.perform(..))
- **Joinpoints**: A *joinpoint* is a point in the execution of the application where an aspect can be plugged in (method, field, constructor etc..). Spring only supports method joinpoints



Advices

- Advice:
  - Before advice: MethodBeforeAdvice
  - After returning advice:AfterReturningAdvice
  - After Throwing advice:ThrowsAdvice
  - Around advice:is effectively before, after-returning, and after-throwing advice. In spring it is defined in MethodInterceptor.

Example of advice



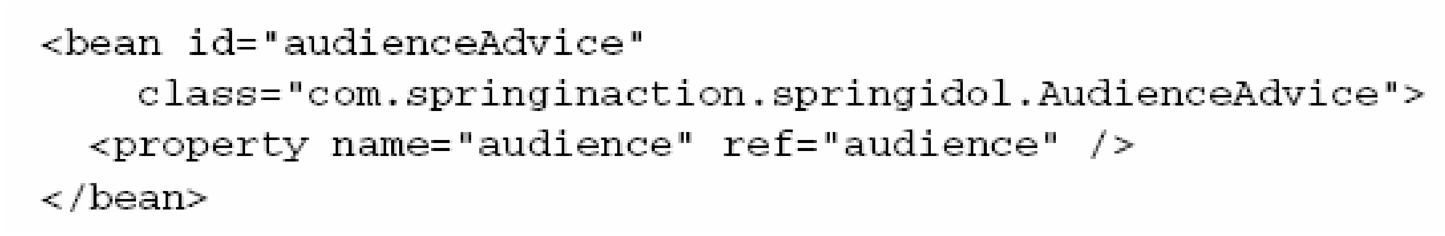
```java
package com.springinaction.springidol;
import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;

public class AudienceAdvice implements
    MethodBeforeAdvice,
    AfterReturningAdvice,
    ThrowsAdvice {

  public AudienceAdvice() {}

  public void before(Method method, Object[] args, Object target)
      throws Throwable {
    audience.takeSeats();
    audience.turnOffCellPhones();
  }

  public void afterReturning(Object returnValue, Method method,
      Object[] args, Object target) throws Throwable {
    audience.applaud();
  }

  public void afterThrowing(Throwable throwable) {
    audience.demandRefund();
  }

  private Audience audience;
  public void setAudience(Audience audience) {
    this.audience = audience;
  }
}
```

Implements three types of advice

Invokes before method

Executes after successful return

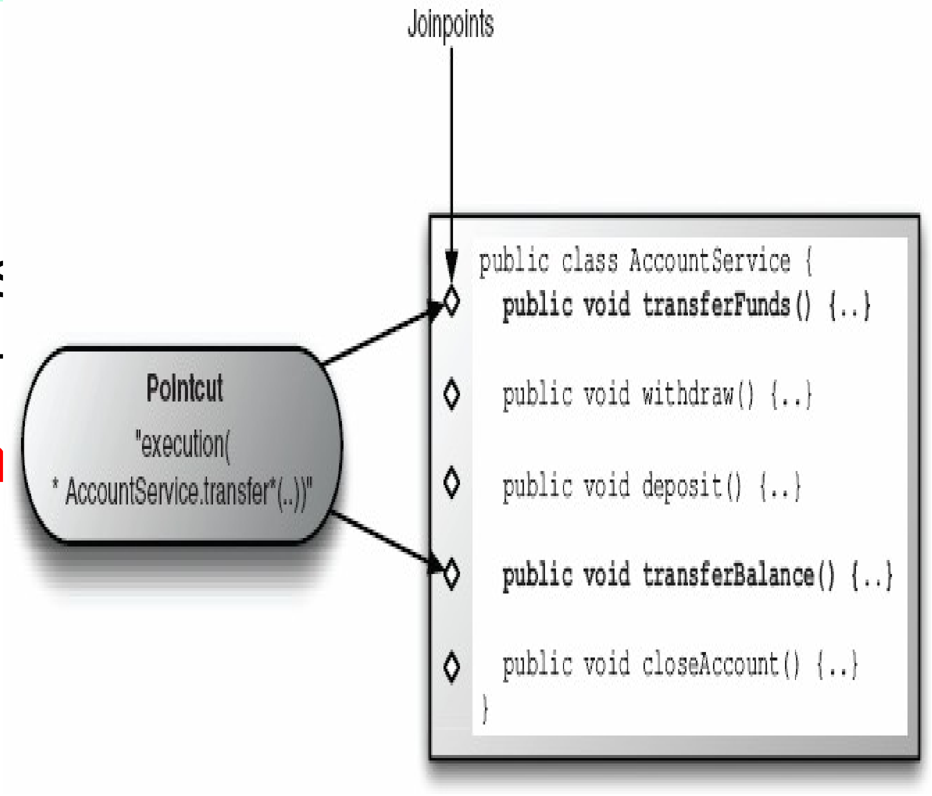
Executes after exception thrown



```xml
<bean id="audienceAdvice"
    class="com.springinaction.springidol.AudienceAdvice">
  <property name="audience" ref="audience" />
</bean>
```

Pointcuts and Advisor

- Defining pointcuts and advis
- Two of the most useful poin
  are regular expression poin
  and AspectJ expression
  pointcuts.



```
public class AccountService {
  public void transferFunds() {..}
  public void withdraw() {..}
  public void deposit() {..}
  public void transferBalance() {..}
  public void closeAccount() {..}
}
```

Declaring a regular expression pointcut

- Define a pointcut using

```
<bean id="performancePointcut"
    class="org.springframework.aop.support.JdkRegexpMethodPointcut">
  <property name="pattern" value=".*perform" />
</bean>
```

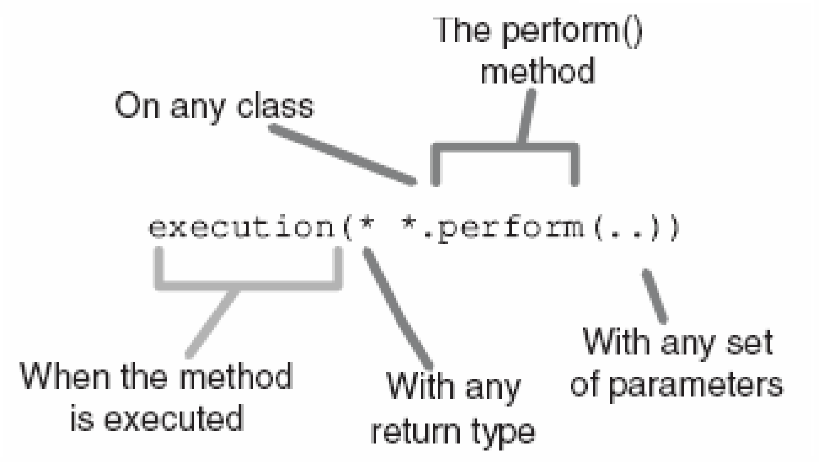
-
```
<bean id="audienceAdvisor"
    class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="pointcut" ref="performancePointcut" />
</bean>
```

```
<bean id="audienceAdvisor"
    class="org.springframework.aop.support.
          ➡ RegexpMethodPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="pattern" value=".*perform" />
</bean>
```

Defining AspectJ pointcuts

- Define pointcut

```
<bean id="performancePointcut"
    class="org.springframework.aop.aspectj.
            ➥ AspectJExpressionPointcut">
  <property name="expression" value="execution(* Performer+.perform(..))" />
</bean>
```

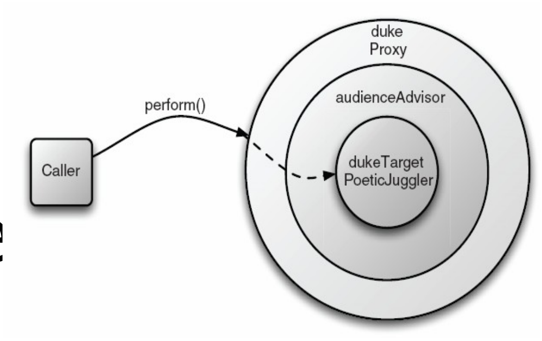

The perform() method
On any class
execution(* *.perform(..))
When the method is executed
With any return type
With any set of parameters

- Associate pointcut and ad

```
<bean id="audienceAdvisor"
    class="org.springframework.aop.aspectj.
          ➥ AspectJExpressionPointcutAdvisor">
  <property name="advice" ref="audienceAdvice" />
  <property name="expression" value="execution(* *.perform(..))" />
</bean>
```

Using ProxyFactoryBean

- Advisors completely define an aspect by associating advice with a pointcut.
- But aspects in Spring are proxied.
- You'll still need to proxy your targe beans for the advisors to take

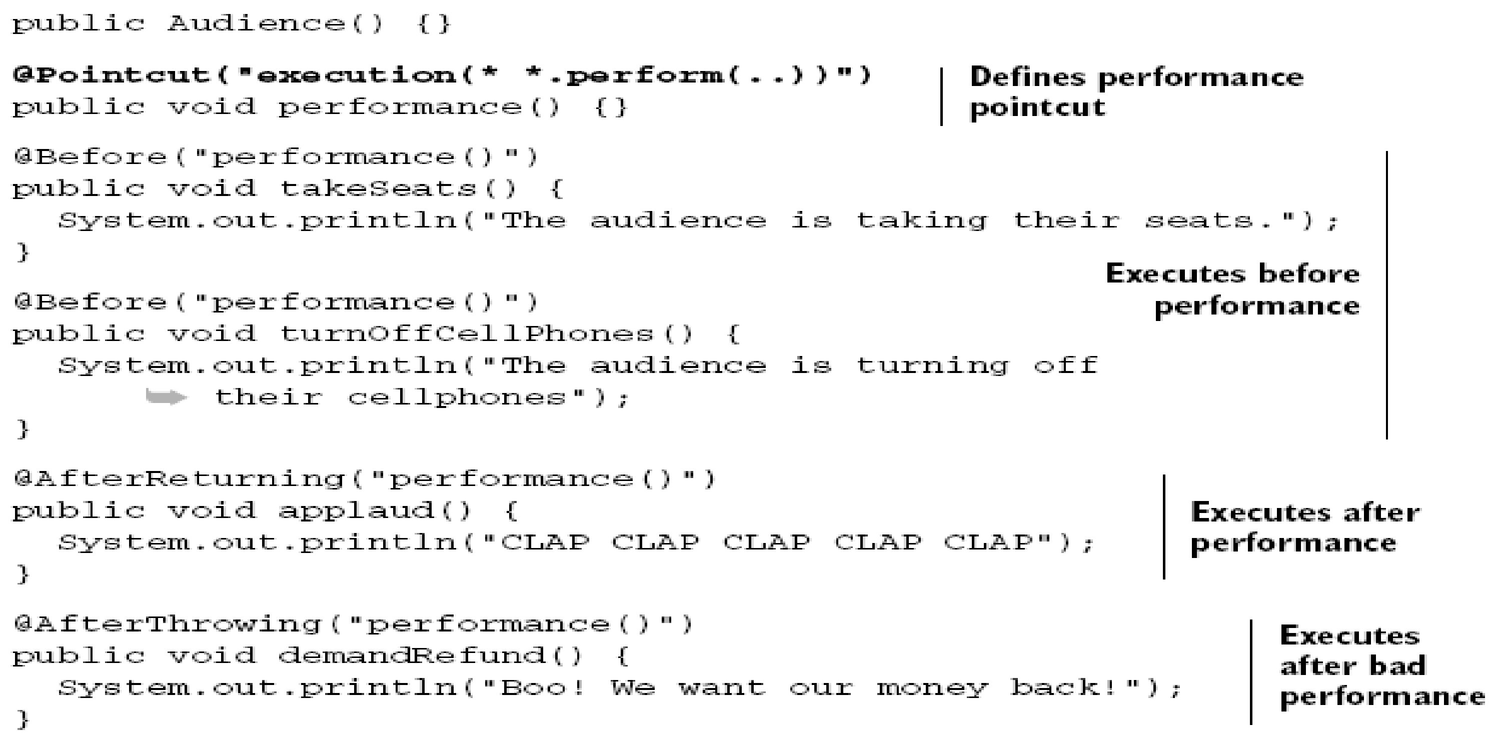


```
<!-- Using ProxyFactoryBean proxy-->
<bean id="duke" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="dukeTarget" />
    <property name="interceptorNames" value="advisor" />
    <property name="proxyInterfaces" value="com.tma.learnspring.example.aop.Performer" />
</bean>
<bean id="dukeTarget" class="com.tma.learnspring.example.aop.Juggler"></bean>
```

AOP with annotations ->learning by yourselves ☺

- **AutoProxying**: DefaultAdvisorAutoProxyCreator
  <bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />

- **Annotation**



```
public Audience() {}

@Pointcut("execution(* *.perform(..))")
public void performance() {}

@Before("performance()")
public void takeSeats() {
  System.out.println("The audience is taking their seats.");
}

@Before("performance()")
public void turnOffCellPhones() {
  System.out.println("The audience is turning off
      ➡ their cellphones");
}

@AfterReturning("performance()")
public void applaud() {
  System.out.println("CLAP CLAP CLAP CLAP CLAP");
}

@AfterThrowing("performance()")
public void demandRefund() {
  System.out.println("Boo! We want our money back!");
}
```

Defines performance pointcut
Executes before performance
Executes after performance
Executes after bad performance

Notes and best practices using this technique

- Logging
- Security component
- Translations
- Common pre-checking

Summary

- AOP
  - AOP is a powerful complement to object-oriented programming.
  - With aspects, you can now group application behavior that was once spread throughout your applications into reusable modules.
  - You can then declaratively or programmatically define exactly where and how this behavior is applied.
  - This reduces code duplication and lets your classes focus on their main functionality.

Q&A

Hitting Database

- Understand how spring support working with database

Spring's data access philosophy

- Designing of data access tier

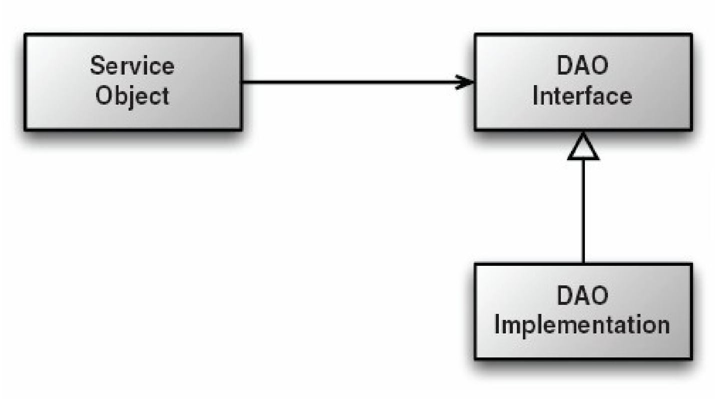
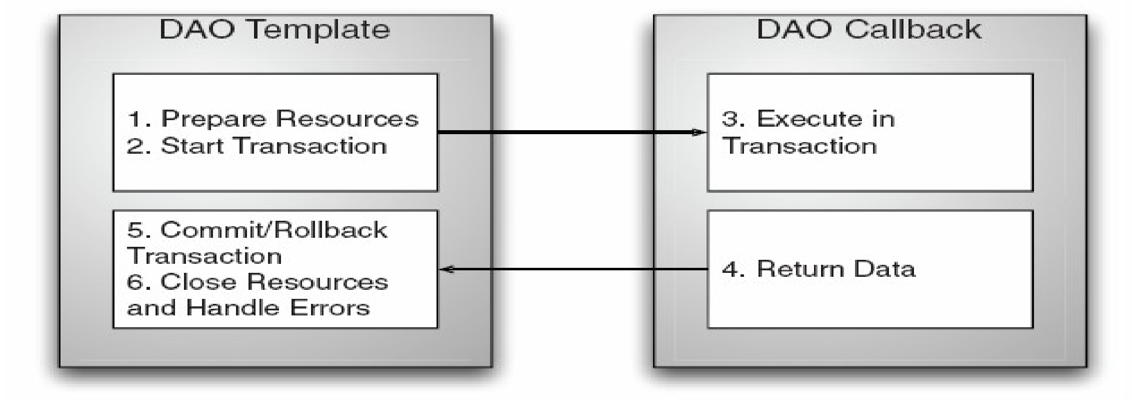


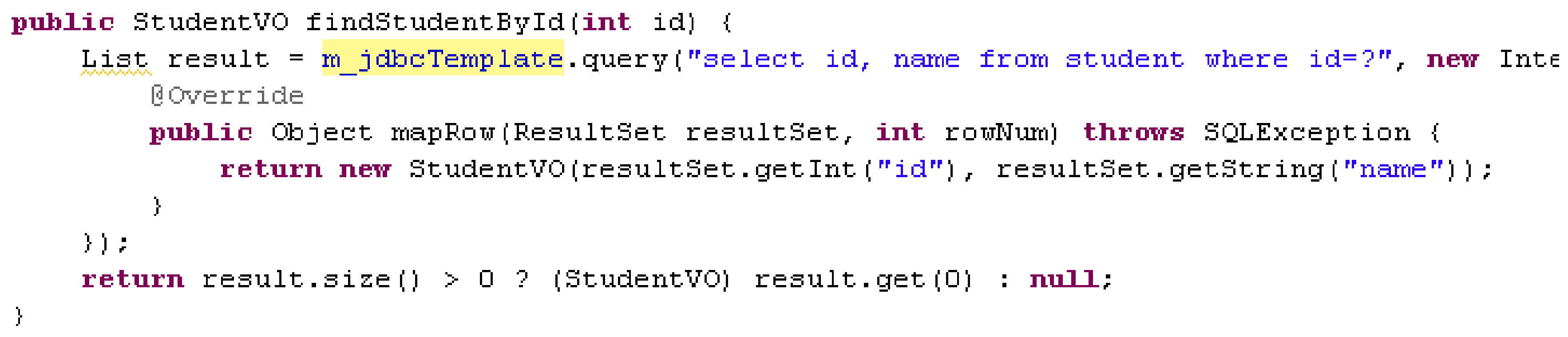
Figure 5.2
Service objects do not handle their own data access. Instead, they delegate data access to DAOs. The DAO's interface keeps it loosely coupled to the service object.

- service objects easily testable since they are not coupled to a specific data access implementation
- relevant data access methods are exposed through the interface -> persistence approach is isolated to the DAO

Templates

Templates manage the fixed part of the process while your custom data access code is handled in the callbacks.


DAO Template
1. Prepare Resources
2. Start Transaction
5. Commit/Rollback Transaction
6. Close Resources and Handle Errors
DAO Callback
3. Execute in Transaction
4. Return Data



```
public StudentVO findStudentById(int id) {
    List result = m_jdbcTemplate.query("select id, name from student where id=?", new Inte
        @Override
        public Object mapRow(ResultSet resultSet, int rowNum) throws SQLException {
            return new StudentVO(resultSet.getInt("id"), resultSet.getString("name"));
        }
    });
    return result.size() > 0 ? (StudentVO) result.get(0) : null;
}
```

Data access templates

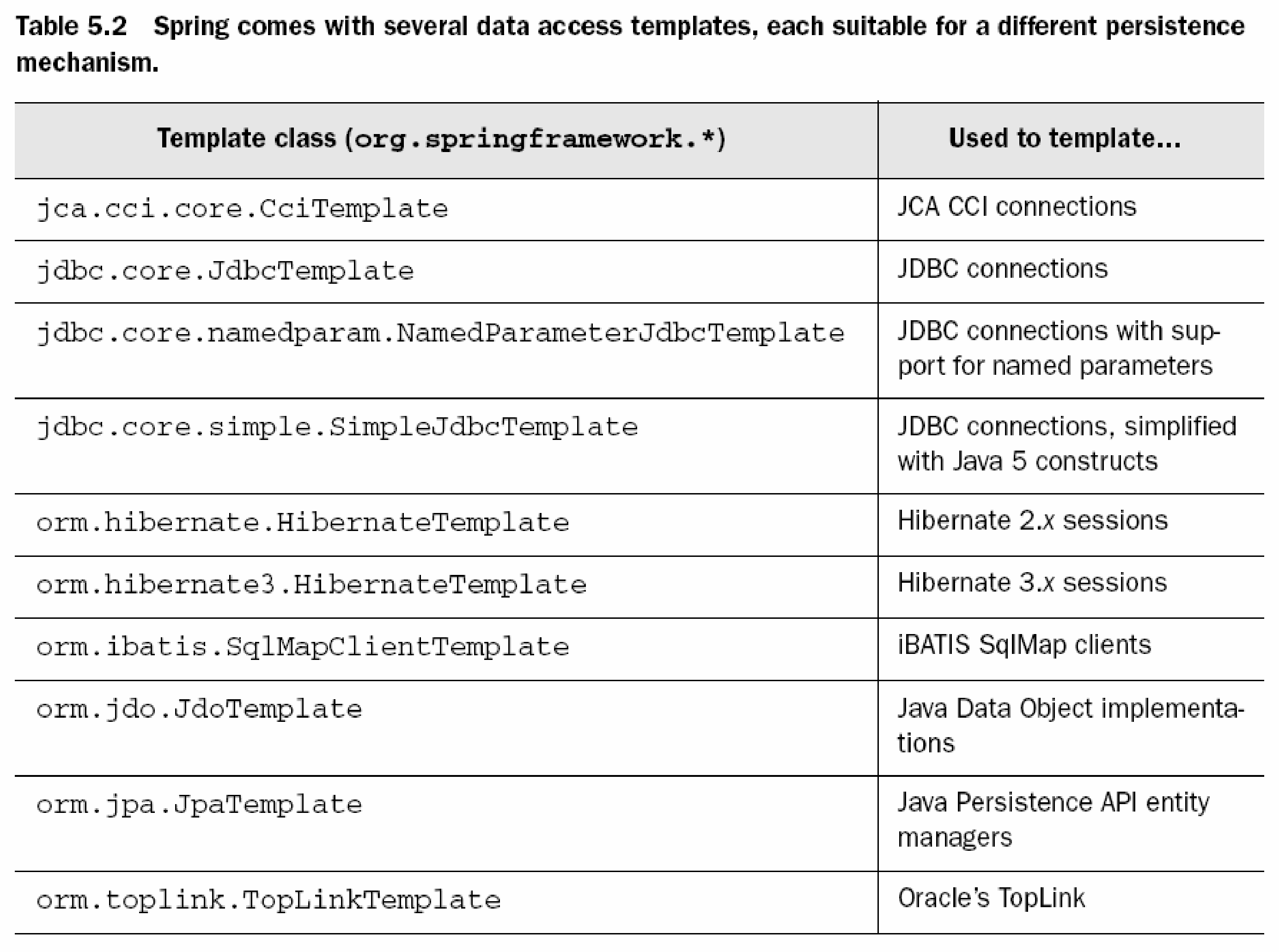


Table 5.2 Spring comes with several data access templates, each suitable for a different persistence mechanism.

Template class (`org.springframework.*`)	Used to template...
`jca.cci.core.CciTemplate`	JCA CCI connections
`jdbc.core.JdbcTemplate`	JDBC connections
`jdbc.core.namedparam.NamedParameterJdbcTemplate`	JDBC connections with support for named parameters
`jdbc.core.simple.SimpleJdbcTemplate`	JDBC connections, simplified with Java 5 constructs
`orm.hibernate.HibernateTemplate`	Hibernate 2.*x* sessions
`orm.hibernate3.HibernateTemplate`	Hibernate 3.*x* sessions
`orm.ibatis.SqlMapClientTemplate`	iBATIS SqlMap clients
`orm.jdo.JdoTemplate`	Java Data Object implementations
`orm.jpa.JpaTemplate`	Java Persistence API entity managers
`orm.toplink.TopLinkTemplate`	Oracle's TopLink

Spring DAO Support

- Relationship between Spring’s DAO, application DAO and template classes

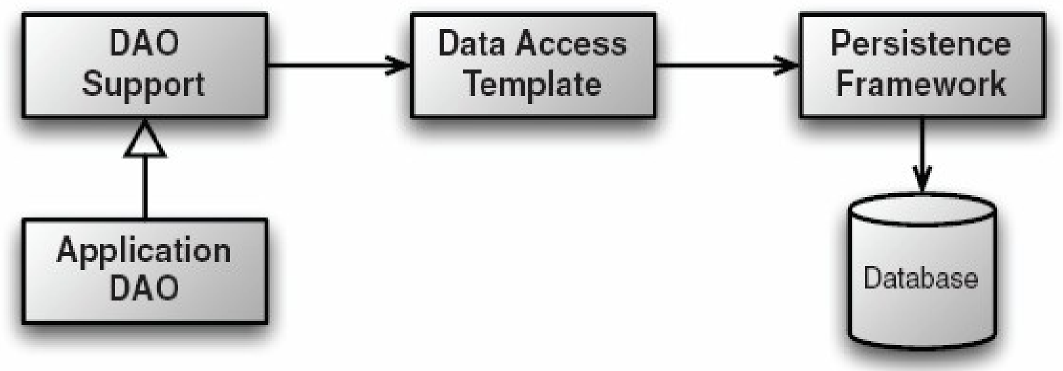


Figure 5.4
The relationship between an application DAO and Spring’s DAO support and template classes

DAO support classes

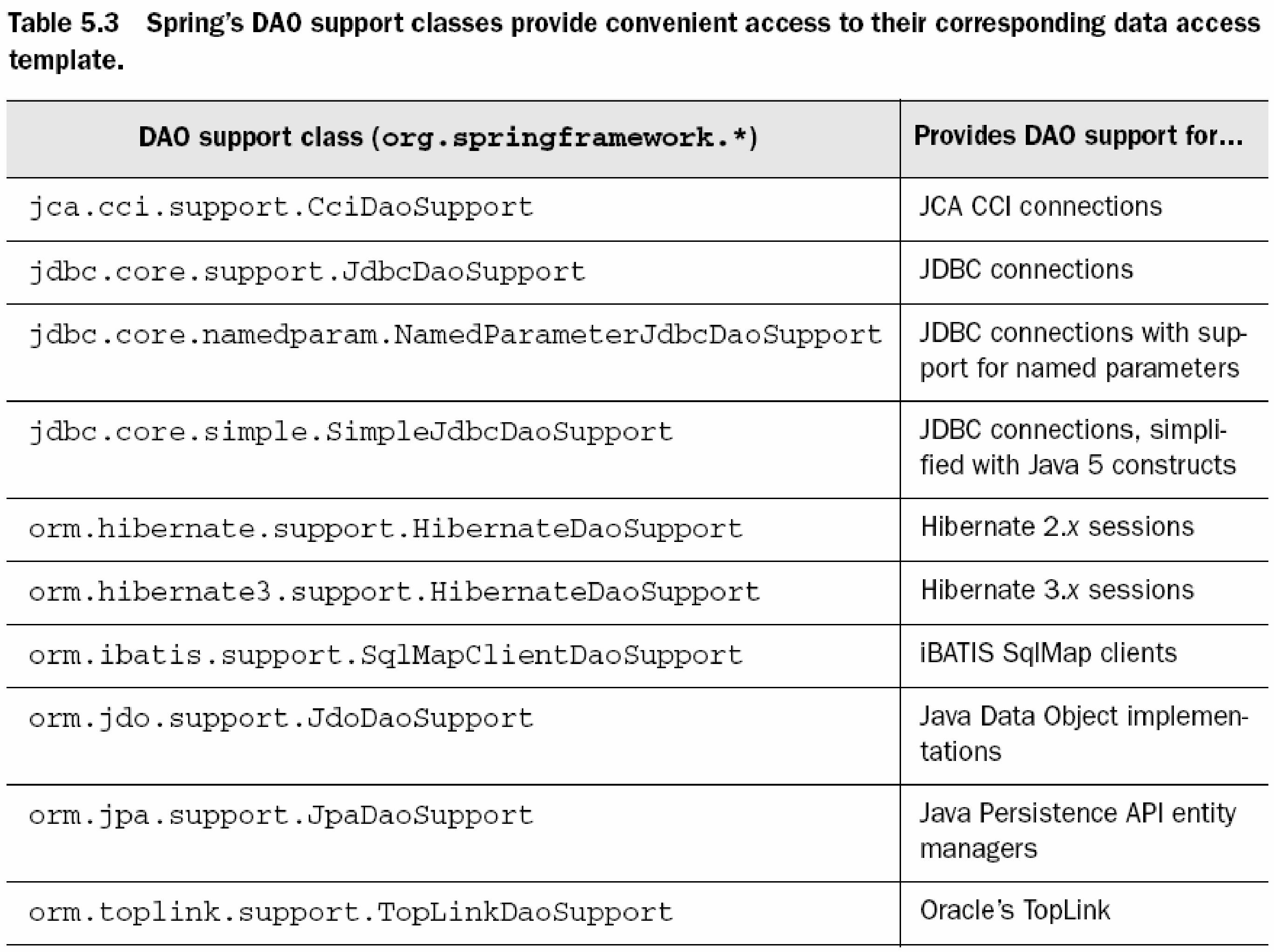


Table 5.3 Spring's DAO support classes provide convenient access to their corresponding data access template.

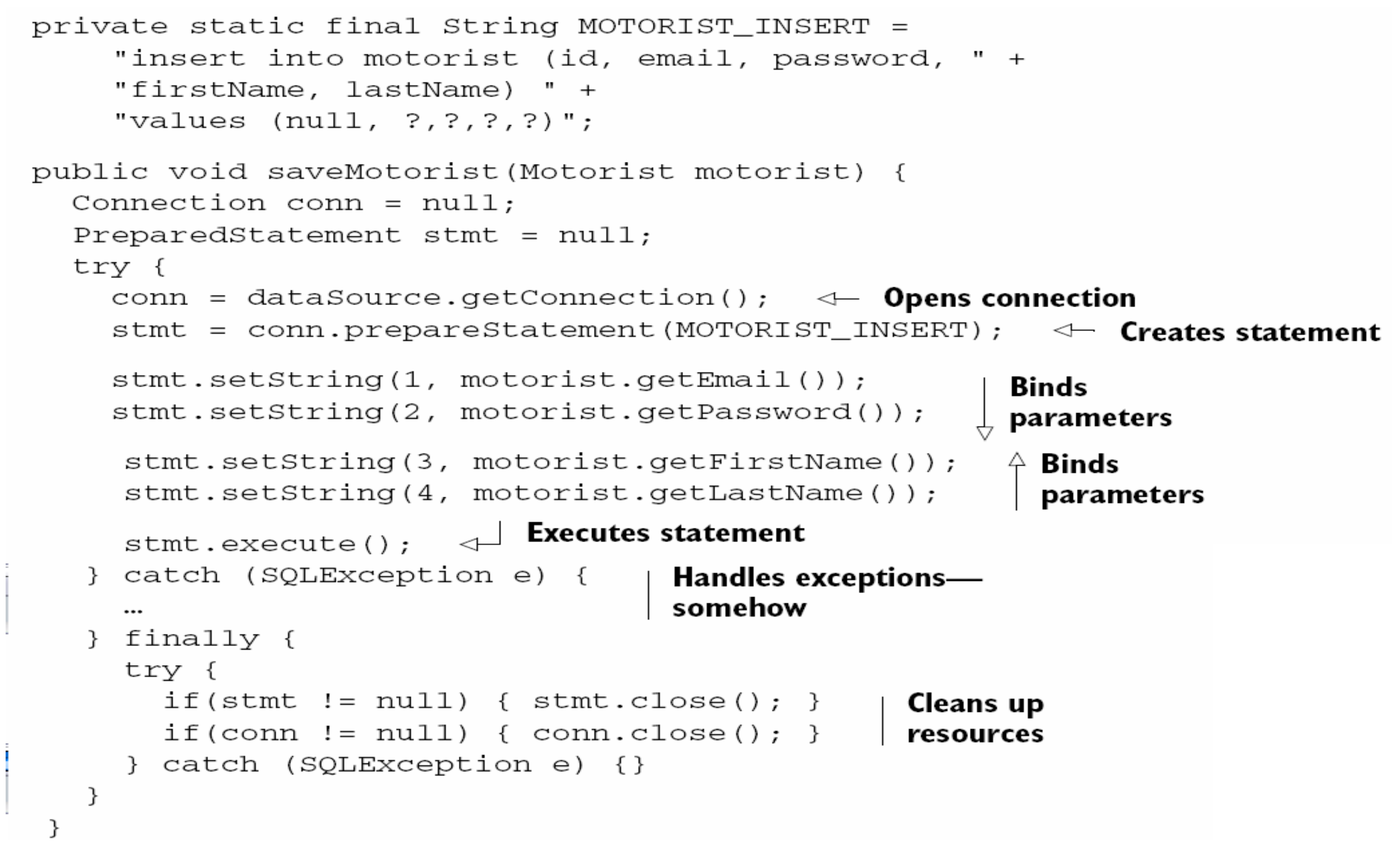
DAO support class (`org.springframework.*`)	Provides DAO support for...
`jca.cci.support.CciDaoSupport`	JCA CCI connections
`jdbc.core.support.JdbcDaoSupport`	JDBC connections
`jdbc.core.namedparam.NamedParameterJdbcDaoSupport`	JDBC connections with support for named parameters
`jdbc.core.simple.SimpleJdbcDaoSupport`	JDBC connections, simplified with Java 5 constructs
`orm.hibernate.support.HibernateDaoSupport`	Hibernate 2.*x* sessions
`orm.hibernate3.support.HibernateDaoSupport`	Hibernate 3.*x* sessions
`orm.ibatis.support.SqlMapClientDaoSupport`	iBATIS SqlMap clients
`orm.jdo.support.JdoDaoSupport`	Java Data Object implementations
`orm.jpa.support.JpaDaoSupport`	Java Persistence API entity managers
`orm.toplink.support.TopLinkDaoSupport`	Oracle's TopLink

Configuring a data source

- Configuring data sources

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
  <property name="url"
      value="jdbc:hsqldb:hsql://localhost/roadrantz/roadrantz" />
  <property name="username" value="sa" />
  <property name="password" value="" />
  <property name="initialSize" value="5" />
  <property name="maxActive" value="10" />
</bean>
```

Using JDBC with Spring



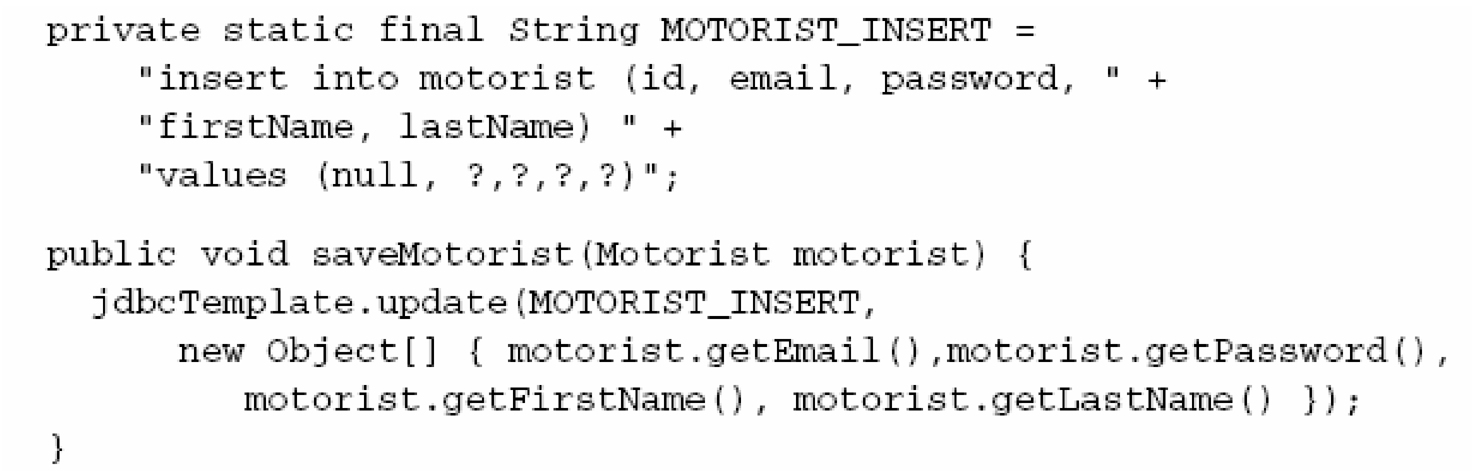
```
private static final String MOTORIST_INSERT =
     "insert into motorist (id, email, password, " +
     "firstName, lastName) " +
     "values (null, ?,?,?,?)";

public void saveMotorist(Motorist motorist) {
  Connection conn = null;
  PreparedStatement stmt = null;
  try {
    conn = dataSource.getConnection();    <- Opens connection
    stmt = conn.prepareStatement(MOTORIST_INSERT);    <- Creates statement

    stmt.setString(1, motorist.getEmail());             Binds
    stmt.setString(2, motorist.getPassword());          parameters

    stmt.setString(3, motorist.getFirstName());         Binds
    stmt.setString(4, motorist.getLastName());          parameters

    stmt.execute();   <- Executes statement
  } catch (SQLException e) {           Handles exceptions—
    ...                                somehow
  } finally {
    try {
      if(stmt != null) { stmt.close(); }       Cleans up
      if(conn != null) { conn.close(); }       resources
    } catch (SQLException e) {}
  }
}
```

- **->** more than 20 line just to insert a simple object

Using JdbcTemplate

- **<bean id="jdbcTemplate“ class="org.springframework.jdbc.core.JdbcTemplate">**
  <property name="dataSource" ref="dataSource" />
  </bean>

  Inject jdbc template into using class



```
private static final String MOTORIST_INSERT =
    "insert into motorist (id, email, password, " +
    "firstName, lastName) " +
    "values (null, ?,?,?,?)";

public void saveMotorist(Motorist motorist) {
  jdbcTemplate.update(MOTORIST_INSERT,
      new Object[] { motorist.getEmail(),motorist.getPassword(),
         motorist.getFirstName(), motorist.getLastName() });
}
```

Integrating Hibernate with Spring

- The main interface for interacting with hibernate is org.hibernate.Session
- SessionFactory is responsible for opening, closing, and managing Hibernate Sessions.
- -> Spring provides a HibernateTemplate is to simplify the work of opening, closing and to convert Hibernate exception to Spring exception.

Integrating Hibernate with Spring

- Configure Spring HibernateTemplate

```
<bean id="hibernateTemplate"

   class="org.springframework.orm.hibernate3.Hibernate
   Template">
   <property name="sessionFactory"
   ref="sessionFactory" />
</bean>
```

Using classic Hibernate mapping files

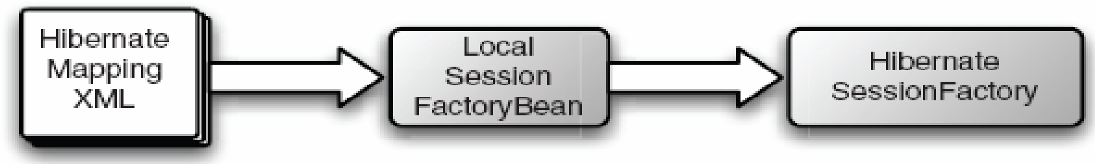


Figure 5.6 Spring's `LocalSessionFactoryBean` is a factory bean that loads one or more Hibernate mapping XML files to produce a Hibernate `SessionFactory`.

The following chunk of XML shows how to configure a `LocalSessionFactory-Bean` that loads the mapping files for the RoadRantz domain objects:

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
       ➥ LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>com/roadrantz/domain/Rant.hbm.xml</value>
      <value>com/roadrantz/domain/Motorist.hbm.xml</value>
      <value>com/roadrantz/domain/Vehicle.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
    </props>
  </property>
</bean>
```

Working with annotated domain objects

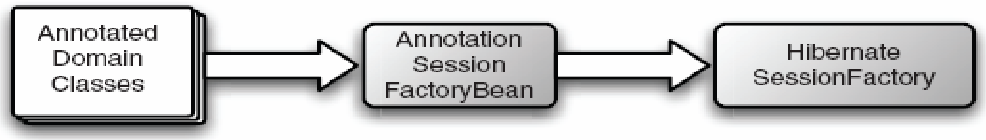


Figure 5.7 Spring's `AnnotationSessionFactoryBean` produces a Hibernate `SessionFactory` by reading annotations on one or more domain classes.

annotations and Hibernate-specific annotations for describing how objects should be persisted. For annotation-based Hibernate, Spring's `AnnotationSessionFactoryBean` works much like `LocalSessionFactoryBean`, except that it creates a `SessionFactory` based on annotations in one or more domain classes (as shown in figure 5.7).

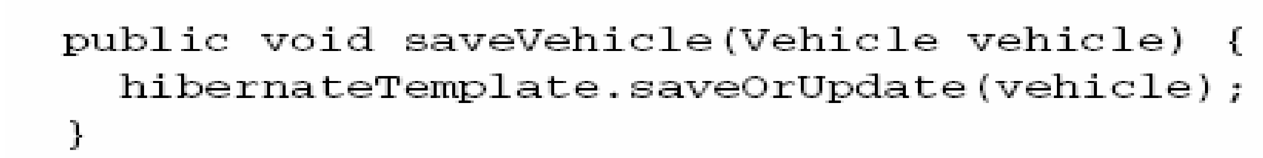
The XML required to configure an `AnnotationSessionFactoryBean` in Spring is similar to the XML for `LocalSessionFactoryBean`:

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.
        ➥ annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="annotatedClasses">
    <list>
      <value>com.roadrantz.domain.Rant</value>
      <value>com.roadrantz.domain.Motorist</value>
      <value>com.roadrantz.domain.Vehicle</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
    </props>
  </property>
</bean>
```

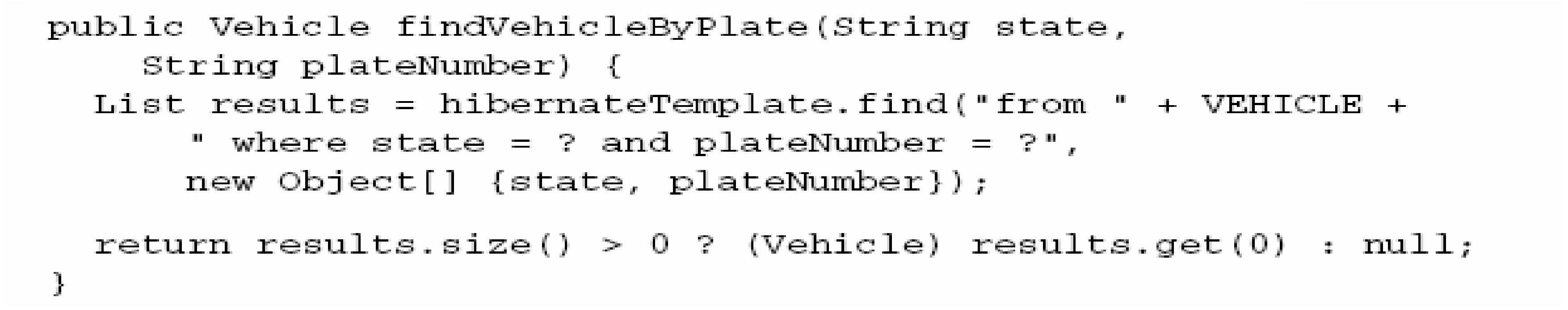
Accessing data through the Hibernate template

- Inject template into DAO object

```
<bean id="rantDao"
    class="com.roadrantz.dao.hibernate.HibernateRantDao">
    <property name="hibernateTemplate" ref="hibernateTemplate"
    />
</bean>
```

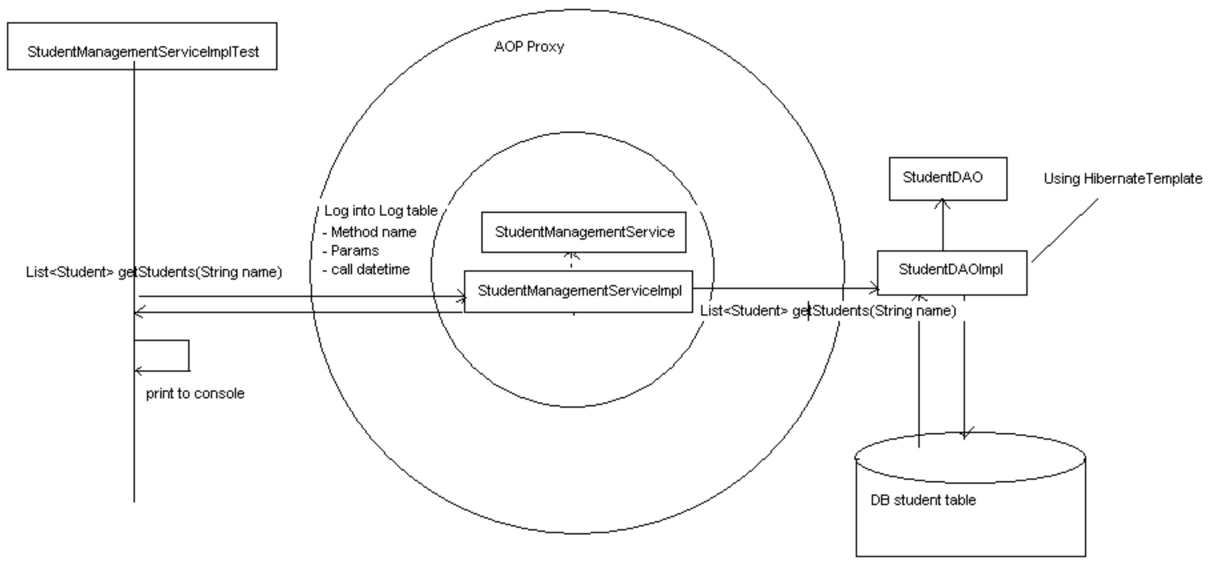
- 

```
public void saveVehicle(Vehicle vehicle) {
  hibernateTemplate.saveOrUpdate(vehicle);
}
```



```
public Vehicle findVehicleByPlate(String state,
      String plateNumber) {
   List results = hibernateTemplate.find("from " + VEHICLE +
         " where state = ? and plateNumber = ?",
         new Object[] {state, plateNumber});

   return results.size() > 0 ? (Vehicle) results.get(0) : null;
}
```

Exercise description



- Unit test is required.
- IOC/DI must be used
- Use HibernateTemplate for DB access
- Use AOP to log user activities

References

- http://www.springsource.org/documentation
- Manning.Spring.in.Action.2nd.Edition.Aug.2007.pdf

Thanks