Recap

- Why Spring?
  - Description of the problem
  - Overall view
  - Inspecting a sample of an Spring-Hibernate Application → List of Questions!
- Detailed study of Spring:
  - Dependency Injection
  - Beans and their wiring
  - Interaction with DB
  - Transactions
- Security

Part 1 of Solution: Dependency injection
1. Create interfaces to hide the details of implementation behind it
2. provide different implementations (GoodImplementation(), Better...)
3. Instead of making the class (Investor) responsible for getting the Investment, assign the Investment to the Investor (this is done via wiring: an xml file)
4. Use Spring container to implement the wiring by wiring the components together.

Part 2 of Solution: AoP
Aspect oriented Programming allows you
1) define crosscutting concerns in one place as an object called aspect.
2) declaratively specify when and where the functionality should be applied
Hence
- You modify them in one place (easier and cheaper maintenance)
- Reduce the code at the business logic

Case study
The application contains
1. pojo
2. implementation
3. a number of Data Access Objects (DAO) interfaces describing methods for interaction with database
4. implementation of DAO classes in hibernate
5. a service class ShopService
6. main method for accessing the system
Model and implementation

- model
  - Book.java
  - Customer.java
  - OrderDetail.java
  - Order.java
  - Review.java
- impl
  - BookImpl.java
  - CustomerImpl.java
  - OrderDetailImpl.java
  - OrderImpl.java
  - ReviewImpl.java

Data Access Object (DAO) classes

- dao
  - BookDAO.java
  - CustomerDAO.java
  - OrderDAO.java
  - OrderDetailDAO.java
  - ReviewDAO.java

Implementation of DAO classes

- using hibernate
  - impl
    - BookImpl.java
    - CustomerImpl.java
    - OrderDetailImpl.java
    - OrderImpl.java
    - ReviewImpl.java

Implementation of DAO classes

Notice the use of sessionFactory. This is a bean which wired to the BookImp

```java
public void save(Book book) {
    this.getSessionFactory().
    getCurrentSession().save(book);
}
```

hbm files

Check hbm files.
Nothing new here.

business logic

Specified in ShopService.java and is extended by the interface
HibernateShopService

The implementation of it is in
ShopServiceImpl

Notice that access to the database is handled through DAO methods, not directly.

populate() is implemented here.
main() method

Pay attention to `ApplicationContext`

```java
private static ApplicationContext context = new ClassPathXmlApplicationContext("shop-spring.xml");
```

`HibernateShopService` `shopService` = (HibernateShopService) `context`.getBean("shopService");

Next we look at `shop-spring.xml`

A bean for database and driver

Package `org.apache.commons.dbcp2`

Implements `jdbc4` in `java7`

Check the API and see wide range of methods among them

- `getDriverClassName`
- `getUrl()`
- `getUsername()`
...

Wire BasicDataSource... check it.

SessionFactory

Objects are created via `BeanFactory`

In `hibernate5` we use implementation `org.springframework.orm.hibernate5.LocalSessionFactoryBean` which implements a bean factory.

LocalSessionFactoryBean

API `sayDescription`

This is the usual way to set up a `shared` `Hibernate SessionFactory` in a Spring application context; the `SessionFactory` can then be passed to `Hibernate-based data access objects` via dependency injection.

```xml
<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Wire `DataSource` to `LocalSessionFactoryBean`

`SessionFactory` needs to know password, driver,... wire in the `dataSource` (see earlier slides)

```xml
<property name="dataSource" ref="dataSource"/>
```

Wire `sessionFactory` to `Dao` classes

Why? They use this bean.

```xml
<bean id="bookDAO"
  class="uk.ac.bham.cs.book.hibernate5.dao.impl.BookDAOImpl">
  <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```
Beans for service

```xml
<bean id="shopService" class="uk.ac.bham.cs.book.hibernate.ShopServiceImpl">
  <property name="bookDAO" ref="bookDAO" />
  ... // and others customerDAO, orderDAO,...
</bean>
```

Handling of transaction delegated to org.springframework.orm.hibernate5.HibernateTransactionManager API say

... Binds a Hibernate Session from the specified factory to the thread, potentially allowing for one thread-bound Session per factory. SessionFactory.getCurrentSession() is required for Hibernate access code that needs to support this transaction...

API goes on by saying

This transaction manager is appropriate for applications that use a single Hibernate SessionFactory for transactional data access, but it also supports direct DataSource access within a transaction (i.e. plain JDBC code working with the same DataSource). This allows for mixing services which access Hibernate and services which use plain JDBC (without being aware of Hibernate)! ... cool :-)

SessionFactory is wired in

```xml
<bean id="transactionManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Creating advice

We will see this in details later

```xml
<tx:advice id="transactionAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED" />
  </tx:attributes>
</tx:advice>
```

Can be applied to any method!

Applying advice to pointcut

```xml
<aop:config proxy-target-class="true">
  <aop:pointcut expression="execution(* uk.ac.bham.cs.book.hibernate.ShopServiceImpl.*(..))"
    id="shopServiceOperation" />
  <aop:advisor advice-ref="transactionAdvice" pointcut-ref="shopServiceOperation" />
</aop:config>
```