

The Spring Framework™



Rod Johnson

Interface21

Spring Framework goals

- Make J2EE easier to use
- Address end-to-end requirements rather than one tier
- Eliminate need for middle tier “glue”
- Provide the best **Inversion of Control** solution
- Provide a pure Java **AOP** implementation, focused on solving common problems in J2EE

...*Spring Framework goals*

- 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

...*Spring Framework goals*

- **“Non-invasive” framework**
 - Application code has minimal or *no* dependency on Spring APIs
 - **Key principal seen throughout Spring’s design**
 - More power to the POJO
 - Application code is decoupled from framework
 - Reduces lock-in to Spring
 - Allows Spring to evolve without breaking backward compatibility
 - Spring has an excellent record of backward compatibility
- *Facilitate unit testing*
 - Allow effective TDD
 - Allow business objects to be unit tested outside the container
- *Facilitate OO best practice*
 - We’re used to implementing “EJB” or “J2EE” applications rather than OO applications.
 - **It doesn’t have to be this way.**

...Spring Framework goals

- Provide a good alternative to EJB for many applications
 - Yet also adds value if you choose to use EJB, by simplifying EJB implementation and accessing existing EJBs
- Enhance productivity compared to “traditional” J2EE approaches

Part of a new wave of frameworks

- Big change in how J2EE applications are written
- Less emphasis on EJB
 - EJB has been grossly overused
- Not just Spring
 - PicoContainer, HiveMind and other Inversion of Control frameworks
 - EJB 3.0 (2005) programming model looks a *lot* like
(Spring IoC – important features) + Hibernate
 - EJB has some unique capabilities for a minority of apps (distributed transaction management, RMI remoting) but for the most part is looking like legacy
- These lightweight frameworks *are* different from earlier frameworks
- Spring is the most mature, most powerful and by far the most popular

Unique Spring capabilities

- Declarative transaction management for POJOs with or without JTA
- Unique consistent approach to **data access**, with common exception hierarchy
 - Greatly simplifies working with JDBC, Hibernate, JDO etc
 - Eliminates common causes of bugs in application code
- **IoC/AOP integration**
- Powerful consistent approach to **remoting** across multiple protocols
- Integration with a wide variety of popular products
- **Gestalt**
 - Consistency makes it more than the sum of its parts

A layered framework

- Web MVC
- AOP framework
 - Integrates with IoC
- **IoC container**
 - **Dependency Injection**
- Transaction management
- Data access
- One stop shop but can also use as modules

Layers of Spring

AOP

AOP core
AOP Alliance
AspectJ

Source-level
metadata

Enterprise

JDBC

ORM

Transactions

Remoting

Messaging

Context

Web-based

Factories

Access

Event

JNDI

Web MVC

Web MVC
Framework

Views
JSP, Velocity,
PDF, Excel...

Spring Core

Beans

Utilities

Web MVC

- Most similar in design to Struts
 - Single shared Controller instance handles a particular request type
- *Controllers, interceptors run in the IoC container*
 - Important distinguishing feature
 - Spring eats its own dog food

Web MVC: Advantages over Struts

- Allows multiple Dispatcherservlets
 - More elegant than Struts 1.1 approach
 - Dispatcherservlets can share a base “application context”
- Interceptors as well as controllers
 - No need to abuse inheritance to put interception behaviour in a common base Action
- No need for custom ActionForms
 - Reuse domain objects or TOs as form objects
 - Avoid code duplication
- *Interface* not *class* based: Easy to customize
- Less tied to JSP
 - Clean separation of controller, model and view

Web tier integration

But I love WebWork/Tapestry/Struts/JSF/whatever

- **The customer is always right**
 - ...unless the customer has only ever used Struts
- We don't dictate how to use Spring
 - You can preserve your investment (tools etc.)
 - You can refactor to use what parts of Spring you need
- Seamless WebWork, Tapestry, Struts integration
- JSF integration

Spring in the Middle Tier

- Complete solution for managing business objects
 - Write your business objects as POJOs
 - Spring handles wiring and lookup
 - Simple, consistent, XML format (commonest choice)
 - But the IoC container is *not* tied to XML
- Application code has few dependencies on the container—often *no* dependencies on the container
 - Spring Pet Store has **no** dependencies on Spring IoC
 - No imports, no magic annotations for IoC: *nothing* Spring-specific
- Easy unit testing. TDD works!

...Spring in the Middle Tier

- Named “beans”—objects—provide a clear point for pluggability
- Objects that depend on a given bean depend on its interface(s) and/or class
 - Better practice to depend on interfaces
- Can change any object’s implementation with zero impact on callers

...Spring in the Middle Tier

- **The most complete IoC container**
 - *Setter Dependency Injection*
 - Configuration via JavaBean properties
 - *Constructor Dependency Injection*
 - Configuration via constructor arguments
 - Pioneered by PicoContainer
 - *Method Injection*
 - *Dependency Lookup*
 - Avalon/EJB-style callbacks

Middle Tier: 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>
```

Middle Tier: 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>
```

Dependency Injection

- Dependencies are expressed in pure Java
- **Push** configuration (container **injects** dependencies) is much better than **pull** configuration (objects look up dependencies)
 - No ad hoc lookup
 - Code is self-documenting, describing its own dependencies
 - Can apply consistent management strategy everywhere
 - Easy to test
 - No JNDI to stub, properties files to substitute, RDBMS data to set up...

Middle Tier: Dependency Injection

- Simple (configuration) or object properties
 - Configuration (timeout)
 - Dependencies on collaborators (accountDao)
- Configuration properties are also important
- Can run many existing classes unchanged
- “Autowiring”
- Trivial to test application classes outside the container, without Spring
- Can *reuse* application classes outside the container
- Hot swapping, instance pooling (with AOP)

Spring in the Middle Tier

- Advanced IoC features
 - Manage lists, maps or sets, with arbitrary nesting
 - Create objects using new or from factory methods
 - Object instances can be **shared** or **non-singleton** (per user)
 - **Method Injection**
 - Container can override abstract or concrete methods at runtime
 - Can use to avoid dependence on Spring
 - Override moves dependency on Spring API from code to configuration
 - **Nearly any existing Java object can be used in a Spring context**
 - Leverages standard JavaBeans `PropertyEditor` machinery
 - Register custom property editors
 - Many hooks to customize container behaviour

Why AOP?

- Dependency Injection takes us a long way, but not quite enough on its own
- AOP complements IoC to deliver a non-invasive framework
- Externalizes *crosscutting concerns* from application code
 - Concerns that cut across the structure of an object model
 - AOP offers a different way of thinking about program structure to an object hierarchy
- EJB interception is conceptually similar, but not extensible and imposes too many constraints on components
 - Can't define new aspects
 - Constrained pointcut model (methods on the component interface)
- Spring provides important out-of-the box aspects, such as:
 - Declarative transaction management for any POJO
 - Pooling

AOP + IoC: A unique synergy

- AOP + IoC is a match made in heaven
- Any object obtained from a Spring IoC container can be transparently advised based on configuration
- Advisors, pointcuts and advices can themselves be managed by the IoC container
- Spring is an integrated, consistent solution

Custom AOP

- Complements, rather than conflicts with, OOP
 - Email administrator if a particular exception is thrown
 - Apply custom declarative security checks
 - Performance monitoring
 - Auditing
 - Caching
- Objects have clear responsibilities
 - No need to abuse inheritance
- Integration with AspectJ/AspectWerkz when you need a full-blown AOP framework

Spring DAO

- Integrated with Spring transaction management
 - Unique synergy
 - Gestalt again...
- Doesn't reinvent the wheel
 - *There are good solutions for O/R mapping, we make them easier to use*
- Out-of-the-box support for
 - JDBC
 - Hibernate
 - JDO
 - iBATIS
 - TopLink
- Model allows support for other technologies

...*Spring DAO*

- Using clearly defined **technology agnostic** DAO interfaces is a best practice
 - Implement with JDO, JDBC, Hibernate, iBATIS...
 - Spring doesn't *enforce* this, but it makes it very easy
- Consistent `DataAccessException` hierarchy allows truly technology-agnostic DAOs

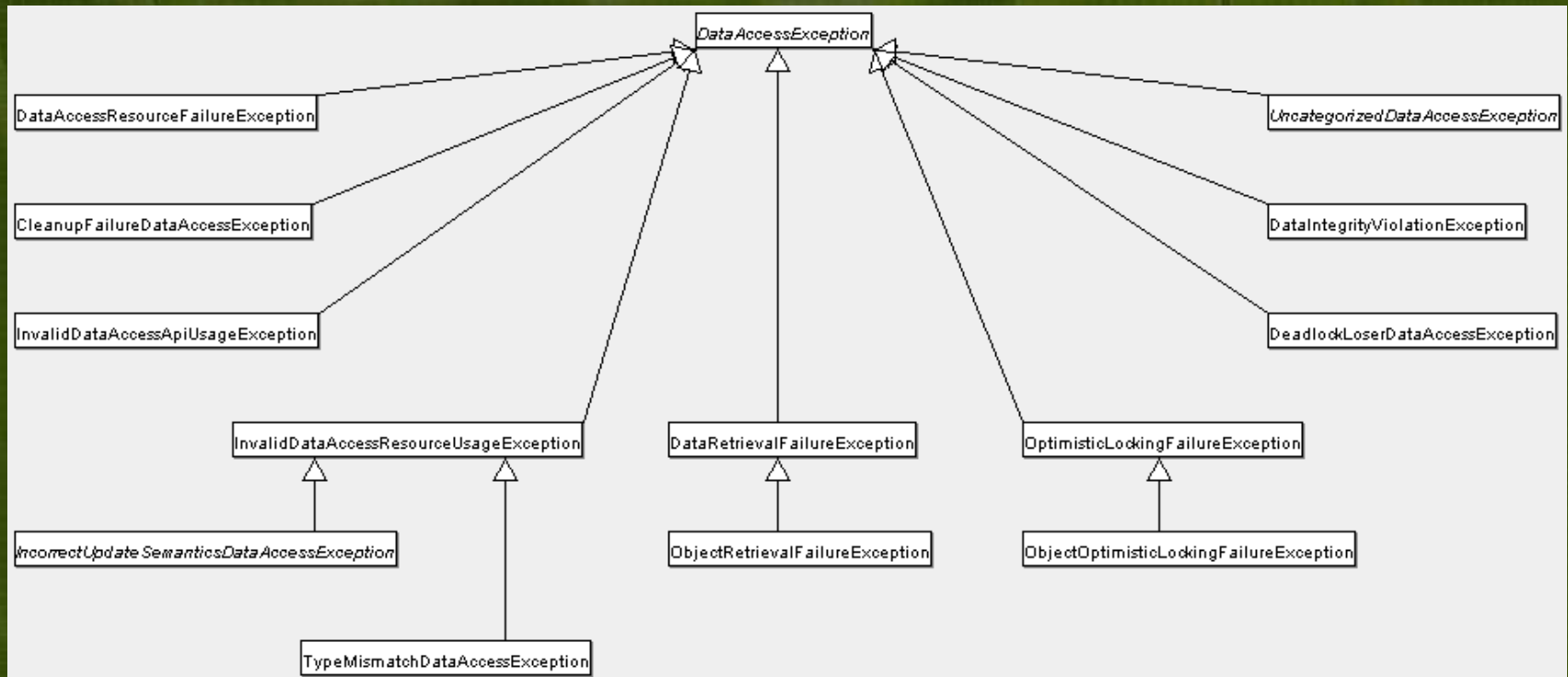
DAO interface example

```
public interface ReminderDao {  
  
    public Collection  
    findRequestsEligibleForReminder()  
        throws DataAccessException;  
  
    void persist(Reminder reminder)  
        throws DataAccessException;  
  
}
```

Spring DAO: JDBC

- Class library offers simpler programming model than raw JDBC
- No more try/catch/finally blocks
- No more leaked connections
 - Spring will *always* close a connection: no scope for programmer error
- Meaningful exception hierarchy
 - No more vendor code lookups
 - Spring autodetects database and knows what Oracle, DB2 error codes mean
 - More portable code
 - More readable code
 - `catch (BadSqlGrammarException ex)`
- Sophisticated, *portable* stored procedure and BLOB/CLOB support
- Can refactor to clean up JDBC without adopting Spring overall
 - *Incremental adoption: Step by step*

Spring DAO: Consistent exception hierarchy



Consistent exception hierarchy

- More informative than SQLException
- Independent of persistence technology (JDBC, JDO, Hibernate, TopLink etc)
- In all cases Spring takes care of translation from underlying API exceptions into DataAccessException hierarchy
 - Abstracts calling code from underlying API

Spring DAO: Hibernate / other ORM

- **Manages Hibernate Sessions or JDO PersistenceManagers**
 - No more custom ThreadLocal sessions
 - Sessions are managed within Spring transaction management
 - Works with JTA if desired
 - Works within EJB container with CMT if desired
- **HibernateTemplate makes common operations easy**
 - Simpler, consistent exception handling
 - Many operations become one-liners
 - **Less, simpler, code compared to using Hibernate alone**
- **Mixed use of Hibernate and JDBC *within the same transaction***

HibernateTemplate DAO example

```
public class HibernateReminderDao extends HibernateDaoSupport implements ReminderDao {  
  
    public Collection findRequestsEligibleForReminder() throws DataAccessException {  
        getHibernateTemplate().find("from Request r where r.something = 1");  
    }  
  
    public void persist(Reminder reminder) throws DataAccessException {  
        getHibernateTemplate().saveOrUpdate(reminder);  
    }  
}
```

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">  
    <property name="dataSource"><ref local="dataSource"/></property>  
    <property name="mappingResources">  
        <value>mycompany/mappings.hbm.xml</value>  
    </property>  
    <property name="hibernateProperties">  
        <props>  
            <prop key="hibernate.dialect">net.sf.hibernate.dialect.HSQLDialect</prop>  
        </props>  
    </property>  
</bean>
```

```
<bean id="myDao" class="com.mycompany.HibernateReminderDao"  
    autowire="autodetect"  
>
```

Spring DAO: JDO

- Comparable to Hibernate support
- PersistenceManager management just like Hibernate Session management
- Mixed use of JDO and JDBC *within the same transaction*
- Support JDO 2.0 features and vendor extensions in a portable manner
 - All major vendors support similar concepts
- We work closely with JDO vendors, including:
 - SolarMetric (Kodo)
 - Open source: JPOX JDO

Spring Transaction

- 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`

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</value></property>
    <property name="accountDao"><ref local="accountDao"/></property>
</bean>
```

Make ServiceImpl transactional

```
<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

Spring J2EE

- No more JNDI lookups
 - `JndiObjectFactoryBean`
 - Generic proxy for `DataSources` etc.
- No more EJB API dependencies, even in code calling EJBs
 - “Codeless EJB proxies”
 - No more `home.create()`, Service Locators or Business Delegates
 - Callers depend on Business Methods interface, not EJB API
 - Enhanced testability
- Maximize code reuse by minimizing J2EE API dependencies
- Full power of your application server lies under the covers

Spring OOP

- No more Singletons
 - An antipattern as commonly used
- **Program to interfaces, not classes**
 - Facilitates use of the **Strategy** pattern
 - Makes good OO practice much easier to achieve
 - Reduces the cost of programming to interfaces to near zero
- Dependency Injection keeps the container from messing up your object model
 - Base object granularity on OO considerations, not Spring considerations
- Combine with transparent persistence to achieve a **true domain model**

Spring Productivity

- Less code to develop in house
- Focus on your domain
- Reduced testing effort
- “Old” J2EE has a poor productivity record
 - Need to simplify the programming model, not rely on tools to hide the complexity

“Old” J2EE vs Spring

- Write a SLSB
 - 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 working around EJB issues
 - *If you want parameterization it gets even more complex*
 - *Need custom code, IoC container or “environment variables” (ouch)*
- Implement a Spring object
 - Business interface
 - Implementation class
 - Straightforward XML configuration
 - The first two steps are necessary in Java anyway
 - *Oops, I really meant “implement a Java object,” not “implement a Spring object”*
 - *If you want to manage simple properties or object dependencies, it’s easy*

“Old” J2EE vs Spring

- Use your SLSB
 - Write a Service Locator and/or Business Delegate: need JNDI code
 - Each class that uses the service needs to depend on EJB interface (home interface) *or* you need a Business Delegate with substantial code duplication
 - *Hard to test outside a container*
- Use your Spring object
 - Just write the class that uses it in plain old Java
 - Express a dependency of the business interface type using *Java* (setter or constructor)
 - Simple, intuitive XML configuration
 - *No lookup code*
 - *Easily test with mock object*

Productivity dividend

- Spring removes unnecessary code
- You end with *no* Java plumbing code and relatively simple XML
 - If your Spring XML is complex, you're probably doing things you couldn't do the old way without extensive custom coding
- The old way you have lots of Java plumbing code *and* lots of XML
 - A lot of the XML is *not* standard
- Combine with Hibernate or JDO for transparent persistence and the advantage is huge compared to traditional J2EE
 - Real example: financial application with 15% of the code of a failed attempt to deliver a "blueprints" solution

Testing

- Spring is designed to facilitate a comprehensive testing strategy
- Unit tests work without *any* container
- Spring provides powerful support for integration testing outside an application server
 - `org.springframework.test` package
 - Designed to minimize setup cost and allow rapid execution of multiple, transactional, integration tests

The Spring community

- www.springframework.org
- 17 developers
 - Around 6 “core” developers, 4 employed by Interface21
 - Significant number of contributors
- Framework is developed test-first
- Vibrant community
 - *Very* active mailing lists and forums
- JIRA Issue tracker at Atlassian
- Approaching 150,000 downloads with Spring 1.1.4 release

Momentum

- At least 8 books on Spring in 2004/early 2005
 - *Spring Live*: Matt Raible
 - ***J2EE Without EJB*: Johnson/Hoeller**
 - *Professional Spring Development* (Q1 2005): Johnson/Risberg/Hoeller/Arendsen
 - ***Better, Faster Lighter Java* (Bruce Tate)**
 - *Spring Developer's Handbook* (Bruce Tate, O'Reilly, Q4)
 - Manning *Spring in Action* (Q4)
 - Addison-Wesley are doing a book on Spring in standalone apps
 - APress! Spring book due out Q4

Related projects

- Spring Rich Client Project
 - Brings Spring IoC container to Swing applications
- Acegi Security for Spring
 - Powerful security solution for Spring applications
 - Much more flexible than out of the box J2EE security
- Spring.NET
 - Ports core IoC and AOP concepts to .NET, offering a consistent programming model for Java and .NET projects

Roadmap

- Continuing vigorous development and innovation
- Spring 1.2 (March-April 2005)
 - JMX support
 - JMX-enable any POJO managed by Spring
 - TopLink integration (donated by Oracle)
 - Simplification of XML configuration
 - Java 5.0 support for those projects able to adopt Java 5.0 already

Spring 1.3

- “Dynamic” bean definitions, from database or properties file, with thread safe refresh
- Further integration with AspectJ 5
- Implement any interface in a scripting language instead of Java
 - Groovy, Beanshell, Jython...extensible
 - Implementation language is transparent to other objects
 - Can modify script without redeploying application
 - Full Dependency Injection support both ways
 - Scripts can be configured using Dependency Injection
 - Other objects can depend on scripts, with reference honoured even if script is modified and reloaded

Questions