Advanced Spring Framework

Rod Johnson

CEO
Interface21

www.interface21.com

TS-7755

Aims

Understand the capabilities of the Spring Framework component model, and how you can use them to add new power to your POJO-based applications

Agenda

Spring component model fundamentals

Value-adds out-of-the-box

User extension points

Spring 2.0 configuration extensions

Spring 2.1: New

Scaling out the Spring component model

What Is Spring?

Much More Than an IoC Container…

Core component model

+ Services

+ Patterns (Recipes)

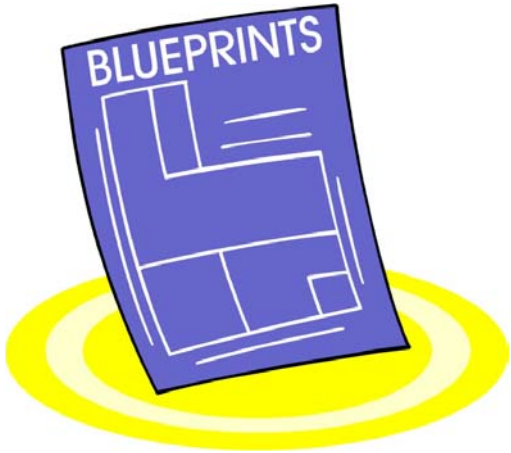
+ Integration (Ecosystem)

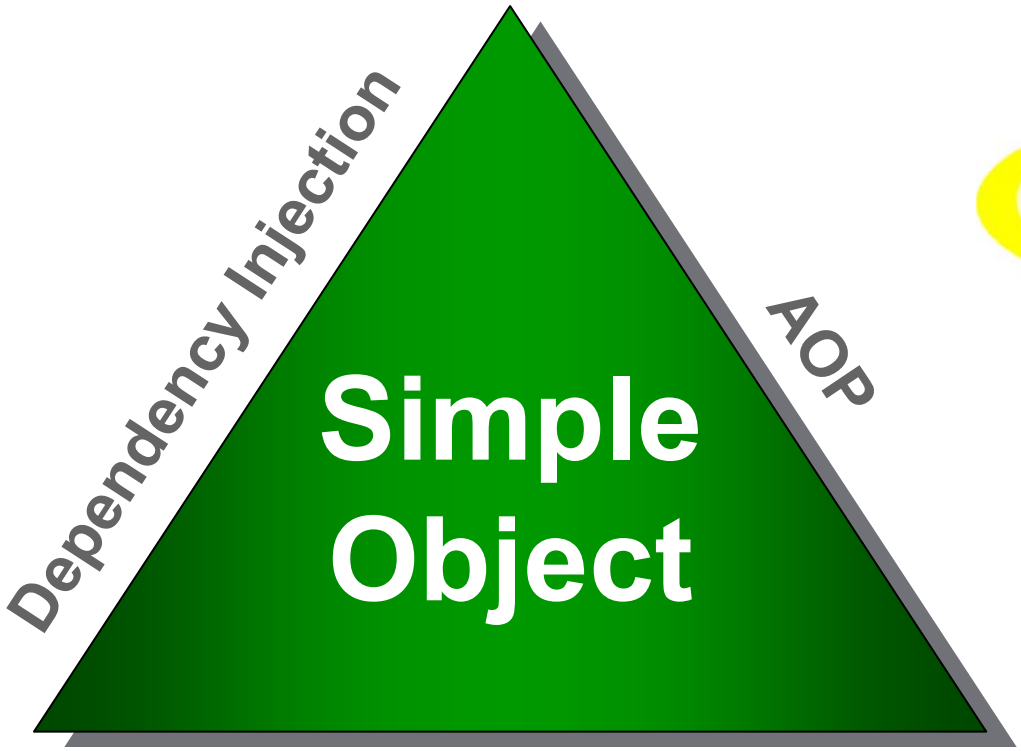
+ Portability (Runs everywhere)

= Universal POJO programming model

- Embraced by more and more enterprise vendors

Enabling Technologies





Portable Service Abstractions (PSA)

Simple POJO

```
public class DefaultWeatherService
                    implements WeatherService {

    public String getShortSummary(String location) { ...
    public String getLongSummary(String location) { ...

    public void update(String location, WeatherData newData)

}
```

- No special requirements
- No dependencies on Spring

Applying Declarative Transactions

Pointcut identifies business service methods

```
<aop:config>
  <aop:pointcut id="businessService"
        expression="execution(* *..*Service+.*(..))"/>
  <aop:advisor pointcut-ref="businessService"
               advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
     <tx:attributes>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="rent*"/>
     </tx:attributes>
 </tx:advice>
```

Advice adds behavior. This advice makes business services transactional.

Value-Add: Exporting a Remote Endpoint

```
<bean name="/httpInvoker/weatherService"
  class="o.sfw...HttpInvokerServiceExporter">
  <property name="service" ref="weatherService"/>
  <property name="serviceInterface"
    value="com.mycompanyWeatherService“ />
</bean>
```

- Can add any number of server side *Exporters* for each Spring managed bean
- Don’t need to write any Java™ code

Remote Client: POJO

```
public class MyClient {
  private WeatherService ws;

  public void setWeatherService(WeatherService ws) {
    this.ws = ws;
  }

  // Business methods omitted
}
```

Configuring the Remote Client POJO

```
<bean id="weatherService"
  class="o.sfw..HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
   value="http://localhost:8080/httpInvoker/weatherService"
  >
  <property name="serviceInterface"
    value="com.mycompany.WeatherService" />
</bean>

<bean id="myClient" class="...MyClient"
    p:weatherService="weatherService"
>
```

Unit Testing the Dependency Injected Client

```
public void testForMyClient() {
    MyClient mc = new MyClient();
    mc.setWeatherService(myMockService);
    // Test mc
}
```

- Can simply inject proxy into client
- Imagine how much harder testing would be if we had coded the look-up method in the client

Value-Add: Java Management Extensions (JMX™) API Export

```
<bean id="mbeanExporter"
  class="org.sfw…MBeanExporter">
  <property name="beans">
    <map>
      <entry key="i21:service=weatherService">
        <ref local="weatherService"/>
      </entry>
      <!-- other objects to be exported here -->
    </map>
  </property>
</bean>
```

- Can expose any bean as a JMX API MBean without writing JMX API code

Extension Point: Auditing Aspect

- Spring container can weave any managed POJO with aspects
- This aspect monitors access to the methods that ask for weather summary strings
- Single code module addresses single requirement (auditing)
- Type safe with *argument binding*

```
@Aspect
public class WeatherMonitorAuditAspect {

    @After("execution(String *.WeatherService.*(String))
                && args(location) && this(ws)")
    public void onQuery(String location WeatherService ws) {
     // Location and WeatherService are bound
     // to the aspect
    }
}
```

***Pointcut* identifies the methods that should be affected**

***Advice* method contains code to execute**

Applying the Aspect

- Ensure an `<aop:aspectj-autoproxy/>` tag is used to turn on automatic application of any @AspectJ aspects found in the context

- Simply define the aspect as a Spring bean

```
<bean class="…WeatherMonitorAuditAspect" />
```

User Extension Points

- The Spring IoC container provides many extension points
- Can easily modify the behavior of the container to do custom annotation processing, specific callbacks, validation, etc. or even to wrap managed objects in proxies
- For example
  - FactoryBean—object configured by the container that creates a component, introducing a level of indirection
  - BeanPostProcessor—called on every bean instantiation
  - BeanFactoryPostProcessor—can modify container metadata
  - BeanDefinition—provides ability to add custom metadata for processing by post processors

User Extension Point Example

- Example: Introducing a **`LogAware`** interface
  - Components implementing this are given a log instance

- The **`AccountService`** bean needs a **`Log`**:

```
public class AccountService implements LogAware{
  private Log log;
  public void setLog(Log log) {
    this.log = log;
  }
  // Business methods omitted...
}
```

User Extension Point Example (Cont.)

```
public class LogInjectingBeanPostProcessor implements BeanPostProcessor {
   public Object postProcessBeforeInitialization(
           Object bean, String beanName) throws BeansException {

       if (bean instanceof LogAware) {
           injectLog((LogAware) bean);
       }
       return bean;
    }
    public Object postProcessAfterInitialization(
          Object bean, String beanName) throws BeansException {
       return bean;
    }
}
```

Called as every bean is initialized

Return value can be any object that is type compatible with the bean being processed

Activating a BeanPostProcessor

- Like most Spring IoC extension points, simply define as a bean
- Automatically gets applied to all other beans
- Customizes the behavior of the container

```
<bean id="com.mycompany.LogInjectingPostProcessor" />
```

Agenda

Spring component model fundamentals

Value-adds out-of-the-box

User extension points

Spring 2.0 configuration extensions

Spring 2.1

Scaling out the Spring component model

XML Configuration Extensions in Spring 2.0: Important New Extension Point

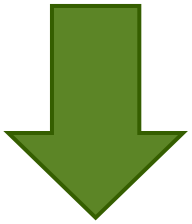
- A bean definition is a recipe for creating one object
- Spring 2.0 added the ability to define new XML tags to produce zero or more Spring bean definitions
- Important new extension point, offering:
  - Higher level of abstraction can simplify many tasks
  - Enables group beans that need to work together into a single configuration unit
  - Can allow existing XML configuration formats to be used to build Spring configuration

How Are XML Configuration Extensions Used?

- Tags out-of-the-box for common configuration tasks
- Many third-party products integrating with Spring
  - Including Java API for XML Web Services (JAX-WS) Reference Implementation
- User extensions: problem-specific configuration
  - Makes it easier to develop and maintain applications
- Allow XML schema validation
  - Better out-of-the-box tool support
  - Code completion for free
- Exploit the full power of XML
  - Namespaces, schema, tooling

XML Configuration in Spring 2.0

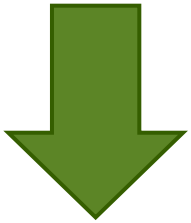
```
<bean id="dataSource" class="...JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/StockData"/>
</bean>
```



```
<jee:jndi-lookup id="dataSource"
      jndiName="jdbc/StockData"/>
```

XML Configuration in Spring 2.0

```
<bean id="properties" class="...PropertiesFactoryBean">
  <property name="location" value="jdbc.properties"/>
</bean>
```



```
<util:properties id="properties"
                 location="jdbc.properties"/>
```

Transaction Simplification

- Specialized tags for making objects transactional
  - Benefit from code assist
- `<tx:annotation-driven />`
- Code assist for transaction attributes

Annotation-Driven Transactions

```
@Transactional(readOnly=true)
interface TestService {

  @Transactional(readOnly=false,

  rollbackFor=DuplicateOrderIdException.class)
  void createOrder(Order order)
      throws DuplicateOrderIdException;

  List queryByCriteria(Order criteria);
}
```

Annotation-Driven Transactions

```
<bean
      class="org.springframework...DefaultAdvisorAutoProxyCreator"/>

<bean class="org.sfw...TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor
                    ref="transactionInterceptor"/>
</bean>

<bean                                          TransactionInterceptor">
    <property name=                            ger"
                    ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <bean class="org.sfw...AnnotationsTransactionAttributeSource">
            </bean>
    </property>
</bean>
```

```
<tx:annotation-driven/>
```

Out-of-the-Box Namespaces

- AOP
- JMX API
- Remoting
- Scheduling
- MVC
- Suggestions and contributions welcome
  - A rich library will build over time

Authoring Custom Extensions: Step 1

- Write an XSD to define element content
  - Allows sophisticated validation, well beyond DTD
  - Amenable to tool support during development
  - Author with XML tools
    - XML Spy

Authoring Custom Extensions: Step 2

- Implement a **`NamespaceHandler`** to generate Spring BeanDefinitions from element content
- Helper classes such as **`BeanDefinitionBuilder`** to make this easy

```
public interface NamespaceHandler {

    void init();

   BeanDefinition parse(Element element,
           ParserContext parserContext);

   BeanDefinitionHolder decorate(Node source,
           BeanDefinitionHolder definition,
           ParserContext parserContext);

}
```

Authoring Custom Extensions: Step 3

- Add a mapping in a META-INF/spring.handlers file

```
http\://www.springframework.org/schema/util=org.springframework.beans.factory.xml.UtilNamespaceHandler
http\://www.springframework.org/schema/aop=org.springframework.aop.config.AopNamespaceHandler
http\://www.springframework.org/schema/jndi=org.springframework.jndi.config.JndiNamespaceHandler
http\://www.springframework.org/schema/tx=org.springframework.transaction.config.TxNamespaceHandler
http\://www.springframework.org/schema/mvc=org.springframework.web.servlet.config.MvcNamespaceHandler
```

Using Custom Extensions

- Import relevant XSD
- Use the new elements

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd">
```

Agenda

Spring component model fundamentals

Value-adds out-of-the-box

User extension points

Spring 2.0 configuration extensions

Spring 2.1

Scaling out the Spring component model

What's New for 2007 JavaOne^{SM} Conference?

- Spring 2.1M1
  - Allows use of annotations for configuration, as well as XML
    - Can mix and match annotations and XML
  - JCA 1.5 support
  - Further enhancements in JPA support
- Aims: Make Spring still easier to use

Java Code: Annotations autoscanned

```
@Component

public class FooServiceImpl

          implements FooService {

    @Autowired
    private FooDao fooDao;

    public String foo(int id) {
        return fooDao.findFoo(id);
    }

}
```

Java Code: Annotations autoscanned

```
@Component
public class FooServiceImpl

          implements FooService {

    @Autowired
    private FooDao fooDao;

    public String foo(int id) {
        return fooDao.findFoo(id);
    }

}
```

Java Code: Annotations autoscanned

```
@Component
public class FooServiceImpl

          implements FooService {

   @Autowired
   private FooDao fooDao;

   public String foo(int id) {
       return fooDao.findFoo(id);
   }

}
```

Java Code: Annotations autoscanned

```
@Component
public class FooServiceImpl
             implements FooService {
    @Autowired
    private FooDao fooDao;

    public String foo(int id) {
        return fooDao.findFoo(id);
    }
}
```

```
@Aspect
public class ServiceInvocationCounter {

    private int useCount;

    @Pointcut("execution(* demo.FooService+.*(..))")
    public void serviceExecution() {}

    @Before("serviceExecution()")
    public void countUse() {
        this.useCount++;
    }

    public int getCount() {
        return this.useCount;
    }

}
```

Bootstrap Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-2.1.xsd">


   <context:annotation-scan base-package="demo"/>

   <bean id="fooDaoImpl" class="demo.FooDaoImpl"
         p:dataSource="dataSource"
   />


   <bean id="dataSource"
   class="org.springframework.jdbc.datasource.DriverManagerDataSource"
         p:driverClassName="org.hsqldb.jdbcDriver"
         p:url="jdbc:hsqldb:mem:hsql:foo"
         p:username="sa"
   />

</beans>
```

One Component Model

- Contributions from different sources of configuration
  - XML
  - Java code
- Internal Java metadata not coupled to configuration format

Agenda

Spring component model fundamentals

Value-adds out-of-the-box

User extension points

Spring 2.0 configuration extensions

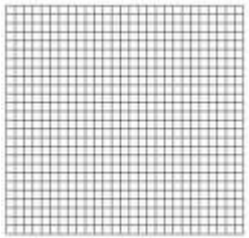
Spring 2.1

Scaling out the Spring component model

Scaling Out Spring

- Spring can consume objects written in dynamic languages
- It can also make it easier to scale out applications by deploying and exposing POJOs in new models
- Examples
  - Other deployment models such as grid
  - SOA with SCA and ESBs
  - OSGi dynamic modularization

Scaling POJOs Out to Grid Deployment



- Spring enables applications to be implemented in POJOs
- Avoids assumptions about environment in application code
  - Environment changes over time
  - Ignorance is bliss: What your objects don't know can't break them if it changes
- Hence deployment model can change without breaking code
  - Servlet
  - Application server
  - Standalone client
  - Grid distribution technology…

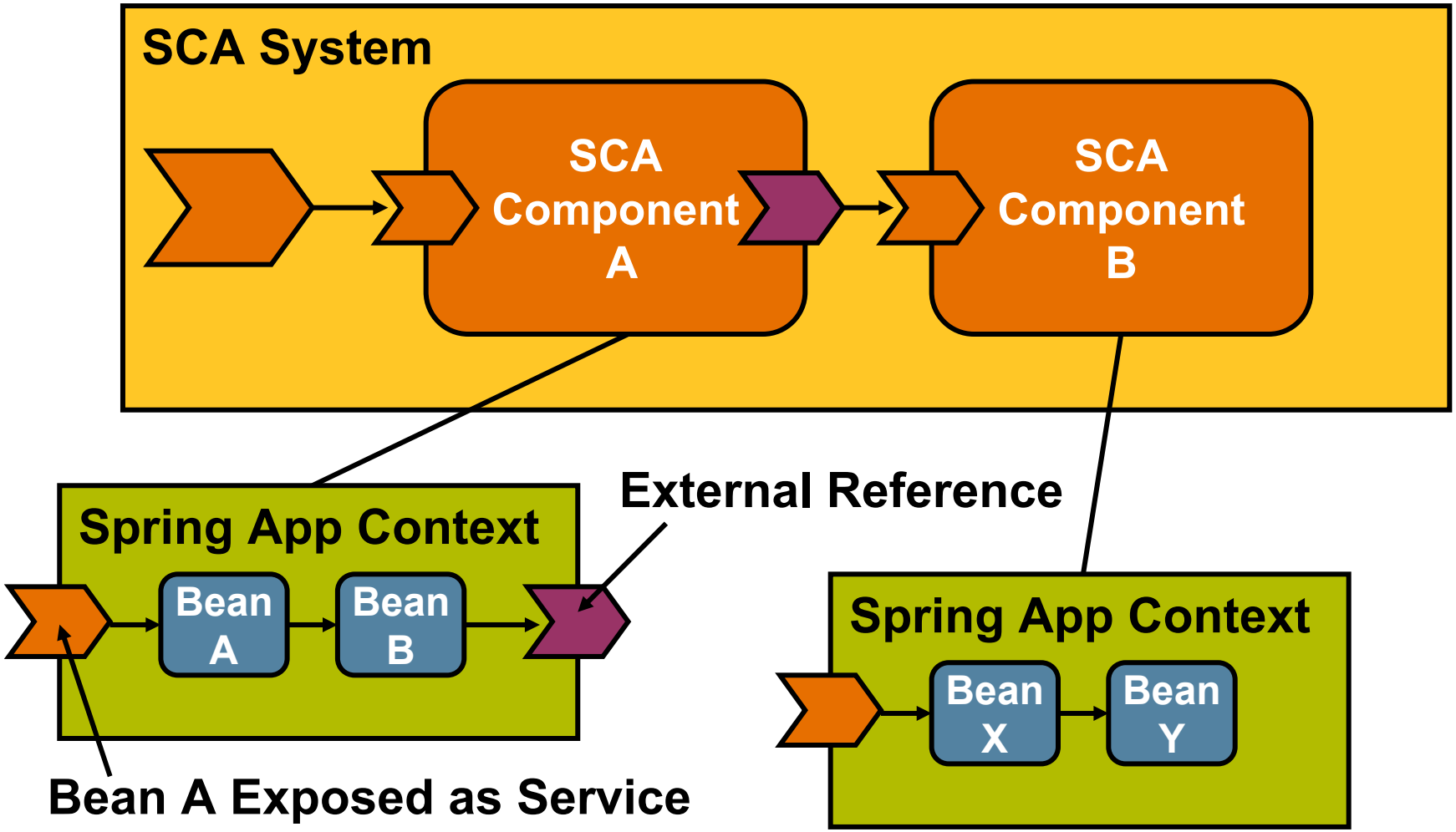
SCA (Service Component Architecture): A Standard for SOA

- Assembly model
  - Service components, references, wires
- Policy framework
  - QoS, etc.
- Service implementation and client API
  - Spring
  - Java API
  - C++
  - BPEL
  - EJB™ architecture
- Bindings
  - Web Services, Java Message Service (JMS), Java Cryptography Architecture (JCA)

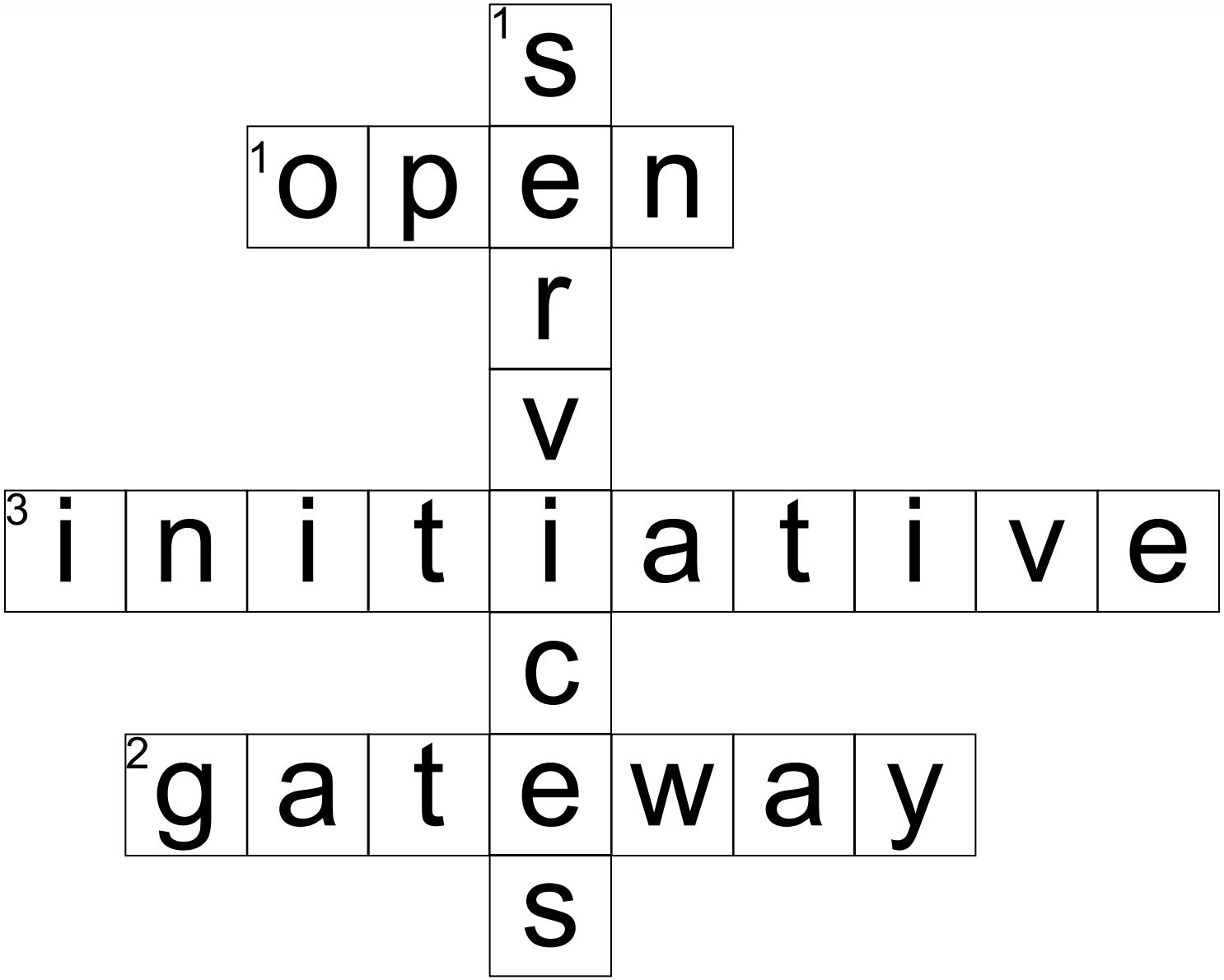
The Open SOA Collaboration

- BEA
- IBM
- Oracle
- SAP
- Sun Microsystems
- Interface21
- Red Hat
- Cape Clear Software
- IONA
- Primeton Technologies
- Sybase
- Siemens
- Software AG
- TIBCO
- Rogue Wave Software

Any Spring Application Is "SCA-Ready"…



OSG—what?


1 s
1 o p e n
r
v
3 i n i t i a t i v e
c
2 g a t e w a y
s

OSGi injection

The Dynamic Module System for Java technology

OSGi: A Module System…

- Partition a system into modules
  - “Bundles”
- Strict visibility rules
- Resolution process
  - Satisfies dependencies of a module
- Understands versioning!

…and It's Dynamic!

- Modules can be
  - Installed
  - Started
  - Stopped
  - Uninstalled
  - Updated
- …at runtime!

Spring and OSGi: Complementary Technologies

- Both are the best at what they do
  - Injection/AOP component model
  - Dynamic runtime infrastructure
- Both run **everywhere**
- Little overlap
- Natural to combine dynamic power of OSGi with ease of use of Spring component model
- Spring/OSGi integration likely to make its way into OSGi specifications

Spring OSGi—Project Goals

- Use Spring container to configure modules (bundles)
- Make it easy to publish and consume services
  - Across a number of bundles
- Enable applications to be coded without dependency on OSGi APIs
  - Easy unit and integration testing
- Provide the needed bundles and infrastructure to deploy OSGi-based applications to application servers

Project Collaborators

- **Led by Interface21**
- Committers from BEA and Oracle also active on the project
- Input to the specification and direction from:
  - OSGi Alliance (technical director and CTO)
  - BEA, Oracle, IBM
  - Eclipse Equinox
  - Felix
  - And many individuals

OSGi Packaging for Spring

- Spring modules packaged as OSGi bundles
  - Spring-core
  - Spring-beans
  - Spring-aop
  - etc.
- All the necessary import and export package headers defined
- Enables an OSGi application to import and use Spring packages and services
- Currently done in Spring-OSGi project
  - Spring module jars will come this way “out-of-the-box” in Spring 2.1

Spring Makes It Easy!

- **Exporting a service:**

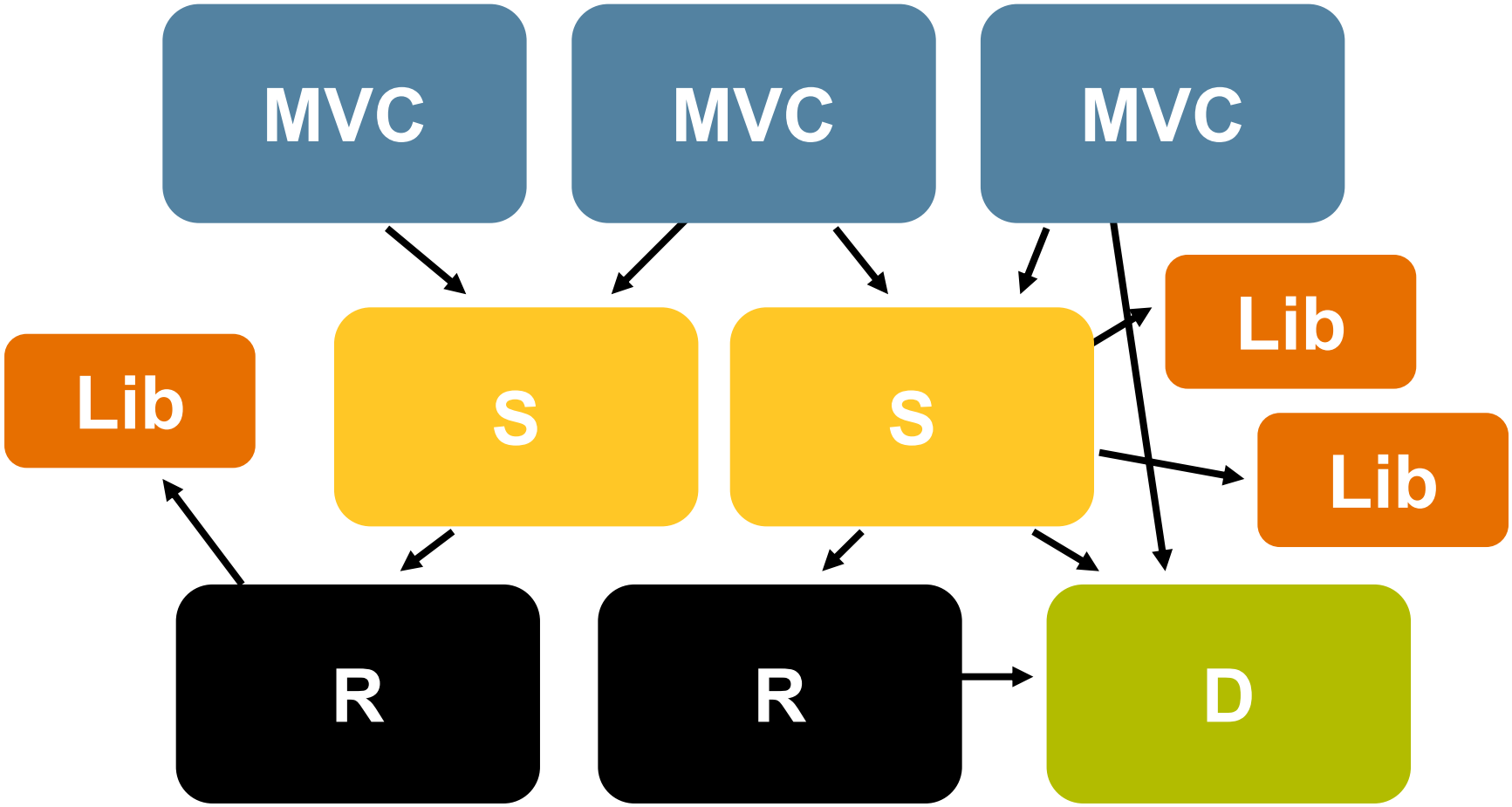
```
<osgi:service id="simpleServiceOsgi"
   ref="simpleService"
   interface=
      "org.sfw.osgi.samples.ss.MyService"/>
```

- **Importing a service:**

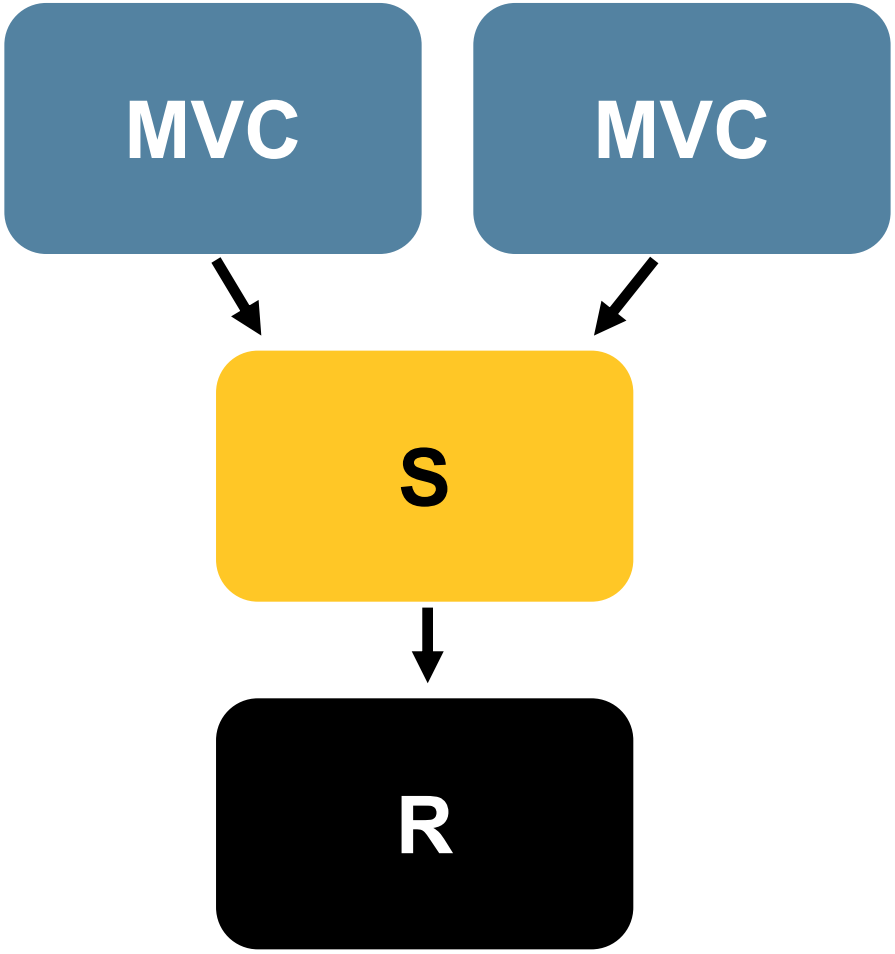
```
<osgi:reference id="aService"
  interface= "org.sfw.osgi.samples.ss.MyService"/>
```

Visibility

Each bundle is a segregated class space

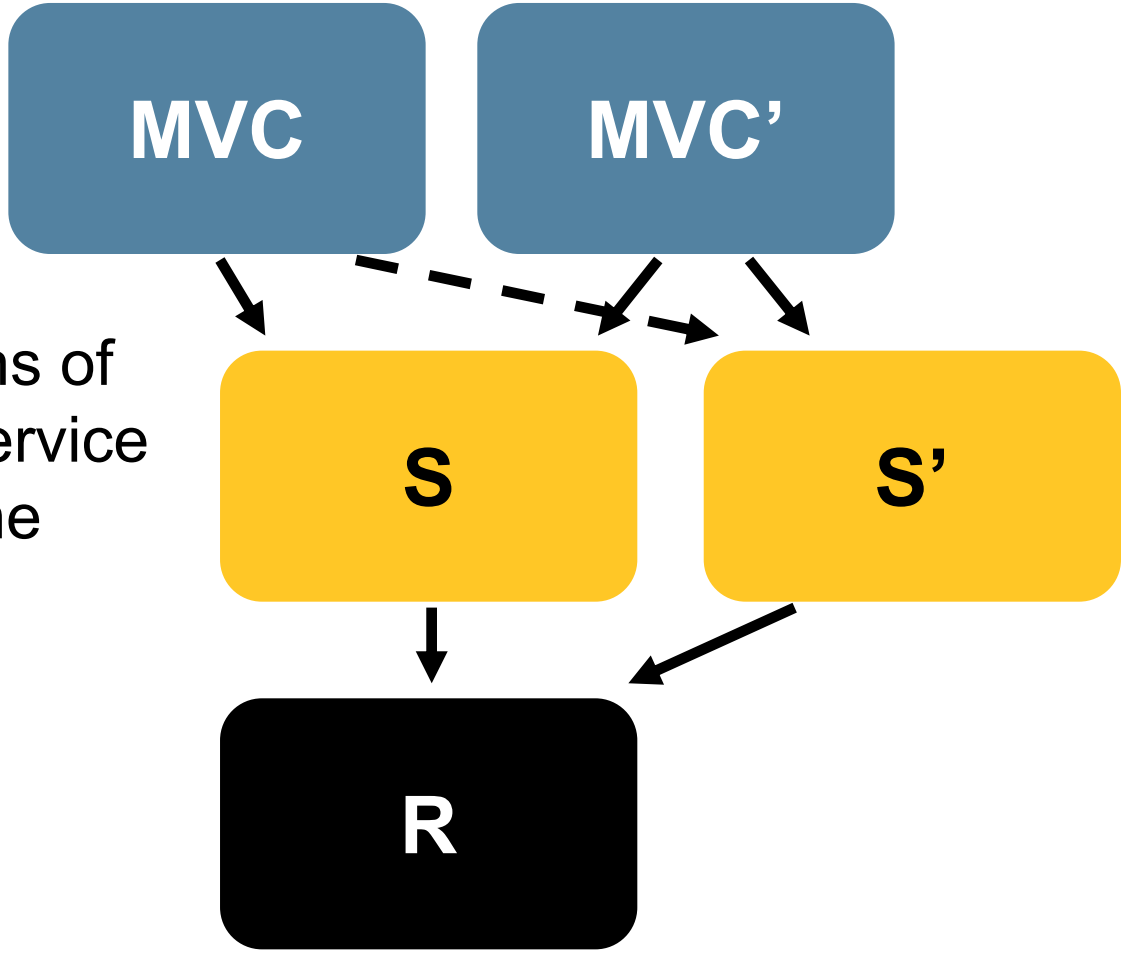


Versioning



Versioning

Two versions of the same service types…at the same time!



The Spring Portfolio

- Spring Framework
- Spring WebFlow
- Spring Web Services
- Spring Security
- Spring Rich Client
- Spring LDAP
- Spring IDE
- Spring OSGi
- Spring Modules
- Spring.NET

- Takes familiar Spring concepts to a wide range of areas
- Consistent themes of simplicity and power

Summary

- Spring provides a highly extensible component model
- POJOs used as “Spring beans” in a Spring application benefit from many potential services for free
  - Many value-adds out-of-the-box
  - Many extension points for users
- The Spring component model is ready for the challenges of tomorrow; build out directions include:
  - Dynamic language support
  - SCA
  - OSGi

Q&A

Advanced Spring Framework

Rod Johnson

CEO
Interface21

www.interface21.com

TS-7755