Spring - Architectures, Patterns and Large Applications

© 2010 SpringSource, A division of VMware. All rights reserved

Who are you?

springsource
A division of vmware

f VMware. All rights reserved

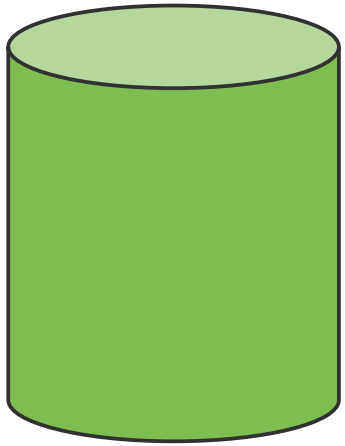
Spring in One Slide

```
<bean id="dataSource" class="....BasicDataSource">
  <property name="username" value="ewolff" />
</bean>

<bean id="transactionManager"
 class="....DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

Before we even start

- **Let us talk about infrastructure configuration first**
  - Database, transactions etc
- **Code is not under your control**
  - e.g. BasicDataSource, PlatformTransactionManager etc
  - So no way to add annotations
- **Different configurations**
  - test: Tomcat
  - production: full Java EE



Tips on Infrastructure Configuration

- **Separate from rest of configuration**
- **Probably in an XML file**
  - Can be changed using a text editor
  - No changes to code needed
  - i.e. no recompile, redeploy …
- **XML file choose the type of environment**
  - Tomcat
  - Java SE for JUnit Tests
  - Full Java EE
- **Use PropertyPlaceholderConfigurer to set machine specific values**

Example

```
<bean class="....PropertyPlaceholderConfigurer">
  <property name="location" value="db.properties" />
</bean>
<bean id="transactionManager"
 class="....DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>

<bean id="dataSource" class="....BasicDataSource"
  destroy-method="close">
  <property name="driverClassName"
   value="${db.driverClassName}" />
  <property name="url" value="${db.url}" />
  <property name="username" value="${db.username:sa}" />
  <property name="password" value="${db.password:}" />
</bean>
```

Specific for the type of infrastructure: Java SE Doesn't use Java EE

Specific for a machine

Alternative: context namespace

- + shorter
- - much less flexible

```
<context:property-placeholder location="db.properties" />
<bean id="transactionManager"
 class="....DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
<bean id="dataSource" class="....BasicDataSource"
  destroy-method="close">
  <property name="driverClassName"
   value="${db.driverClassName}" />
  <property name="url" value="${db.url}" />
  <property name="username" value="${db.username:sa}" />
  <property name="password" value="${db.password:}" />
</bean>
```

Specific for a machine

This approach is quite powerful

- Spring Beans can be created depending on the environment

```
<alias name="dataSource.${environment}"
 alias="dataSource" />

<bean id="dataSource.jse"
 class="org.apache.commons.dbcp.BasicDataSource"
 lazy-init="true" />

<bean id="dataSource.jee"
class="org.springframework.jndi.JndiObjectFactoryBean"
 lazy-init="true">
  <property name="jndiName" value="jdbc/dataSource" />
</bean>
```

Jürgen spoke about upcoming features in 3.1 for this

© 2010 SpringSource, A division of VMware. All rights reserved

On to the rest of the system

© 2010 SpringSource, A division of VMware. All rights reserved

Architecture

The software architecture of a program or computing system is the

structure

or structures of the system, which comprise software components,
the

externally visible properties

of those components, and *the*

relationships

between them.

- **How can you define an architecture using Spring?**

The example: Spring Biking!

- **We need...**
  - Catalog of all available Mountain Bike parts
  - System to configure and build custom Moun
  - System for customer data
  - Track orders and repairs

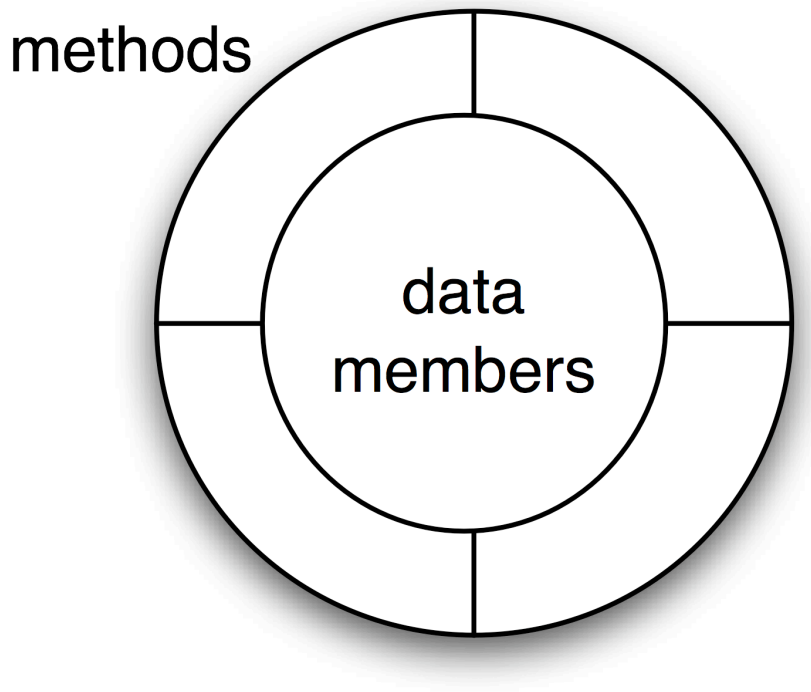
Parts of an Architecture: Prolog

springsource
A division of vmware

© 2010 SpringSource, A division of VMware. All rights reserved

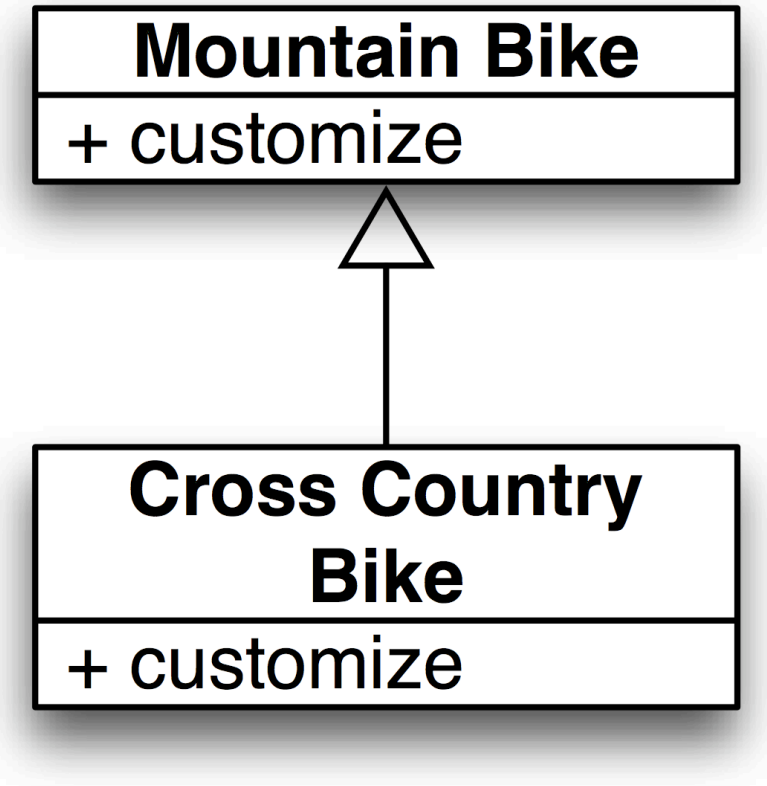
Parts of an Architecture - Prolog

- Object: Information Hiding
- Data may not be accessed from the outside directly

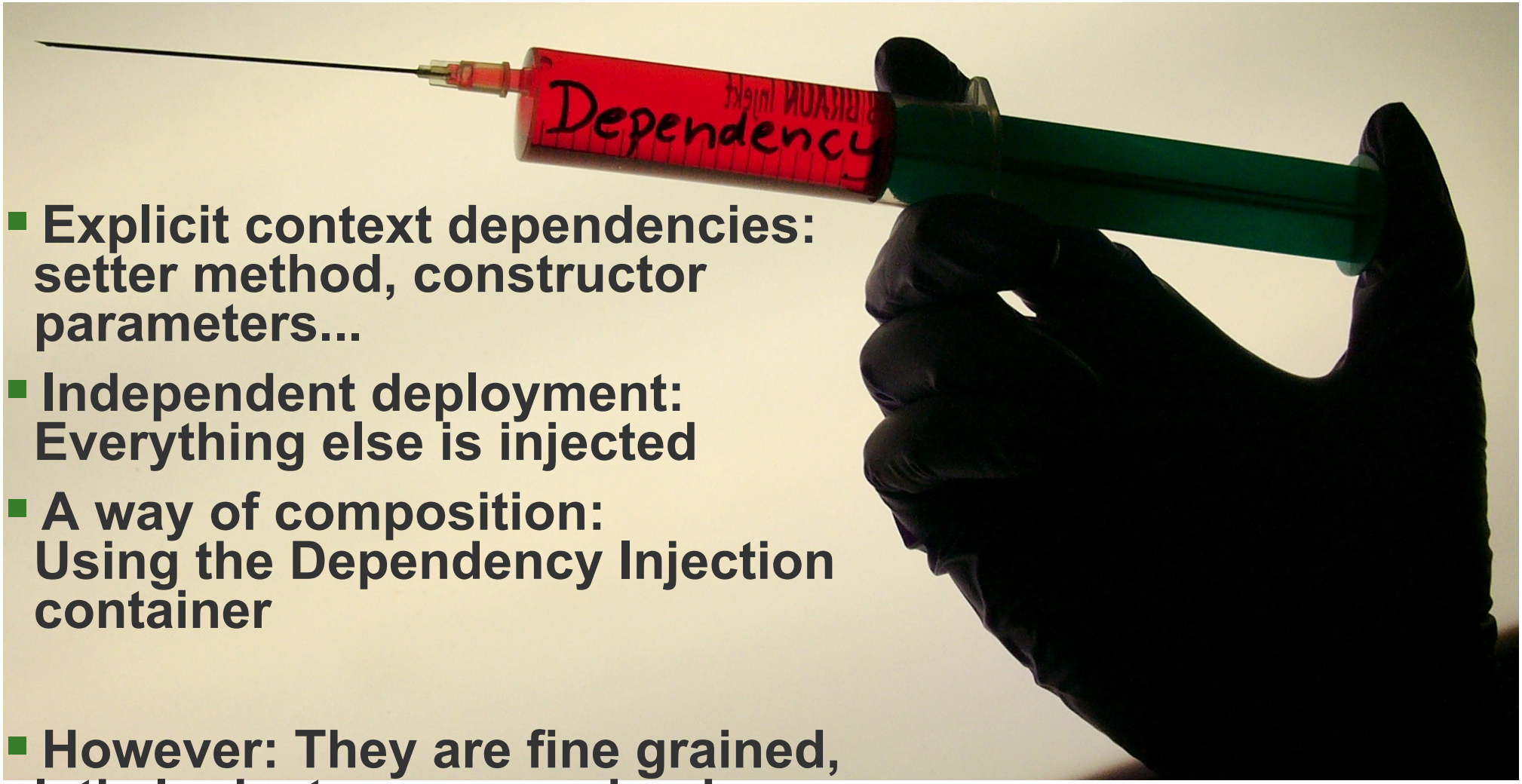


Classes

- ...define types of objects
- May provide specific implementations for methods (e.g. customize())
- White box reuse



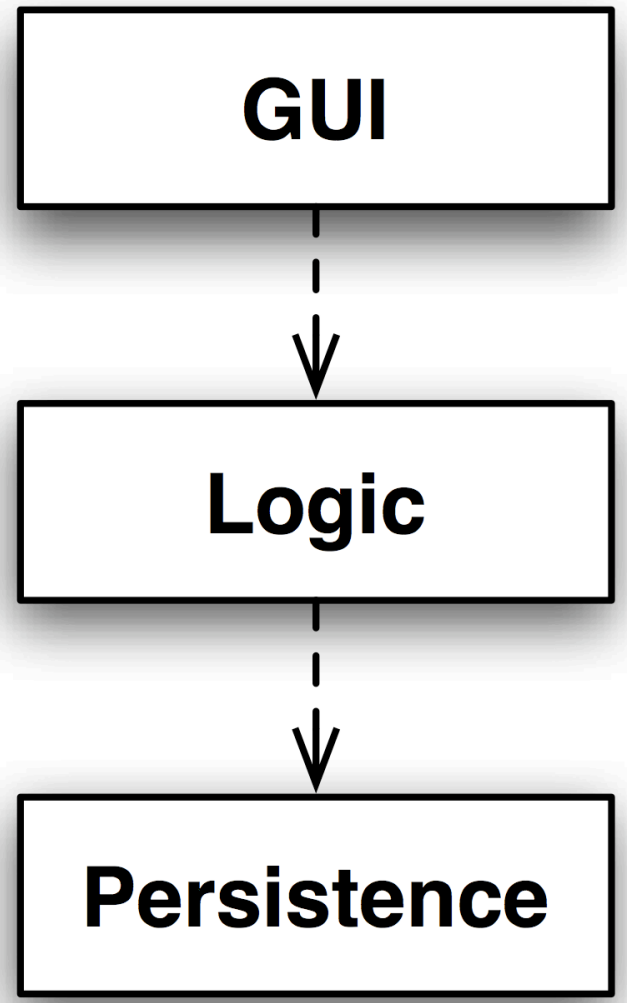
Dependency Injection adds



- Explicit context dependencies: setter method, constructor parameters...
- Independent deployment: Everything else is injected
- A way of composition: Using the Dependency Injection container
- However: They are fine grained, let's look at coarse grained examples

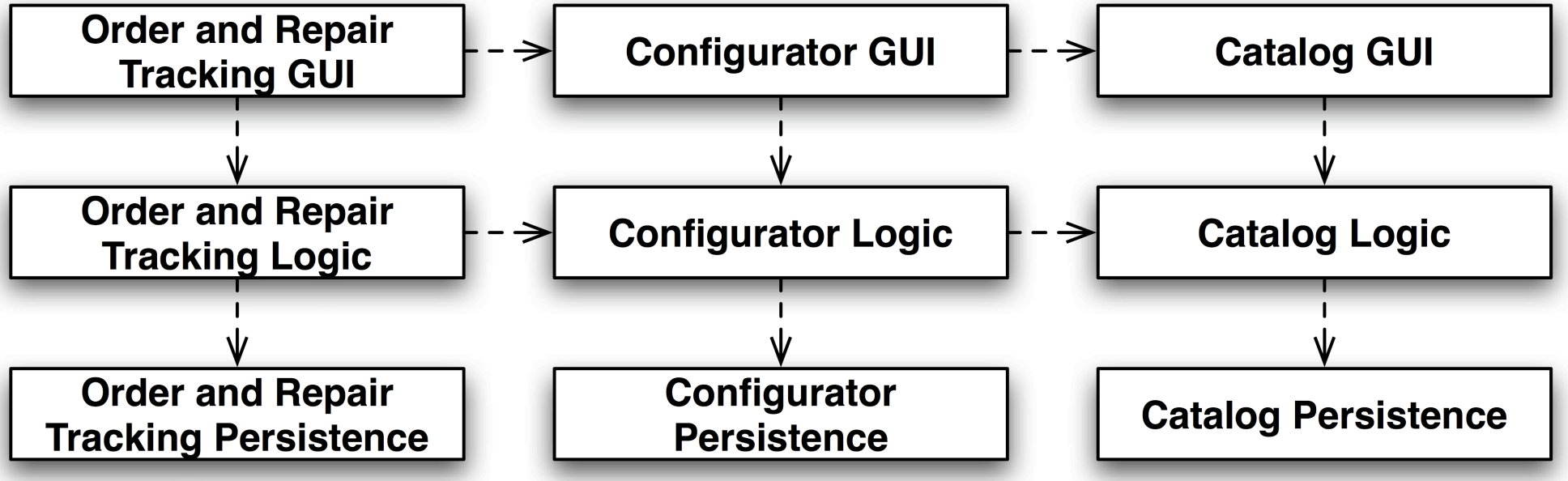
Layer

- Each layer may only depend on layers below it -> better dependency management
- Typical technical
- Can have a Facade



Vertical Slices

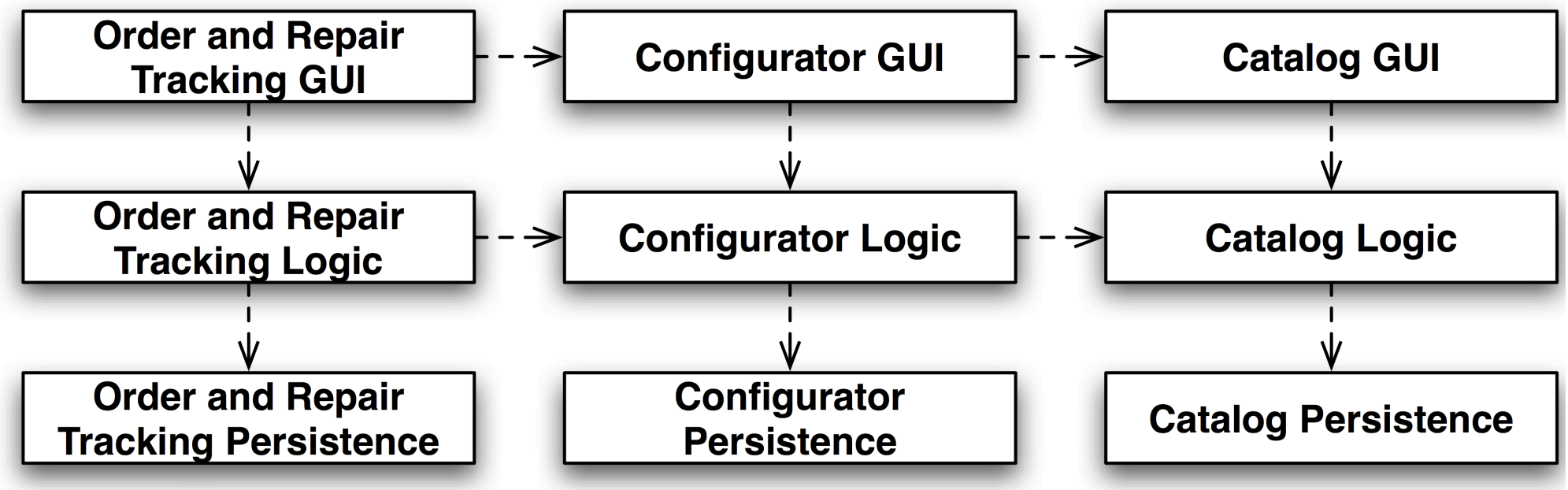
- Typical business domains
- Example with Vertical Slices and Layers:



Layers and Slices using only Spring

springsource
A division of vmware
© 2010 SpringSource, A division of VMware. All rights reserved

The example again...



The obvious solution

- Note the additional infrastructure configuration file (javase.xml)
- Different infrastructure for test / production / etc easily possible

```
ApplicationContext applicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "tracking-gui.xml",
   "tracking-logic.xml", "tracking-persistence.xml",
   "mtb-configurator-gui.xml",
   "mtb-configurator-logic.xml",
   "mtb-configurator-persistence.xml",
   "mtb-catalog-gui.xml", "mtb-catalog-logic.xml",
   "mtb-catalog-persistence.xml", "javase.xml" });
```

The other obvious solution

```
<beans ...>

<import resource="tracking-gui.xml" />
<import resource="tracking-logic.xml" />
<import resource="tracking-persistence.xml" />
<import resource="mtb-configurator-gui.xml" />
<import resource="mtb-configurator-logic.xml" />
<import resource="mtb-configurator-persistence.xml" />
<import resource="mtb-catalog-gui.xml" />
<import resource="mtb-catalog-logic.xml" />
<import resource="mtb-catalog-persistence.xml" />
<import resource="javase.xml" />

</beans>
```

Obvious != good

- Each configuration file should be a layer
- But: Each Spring Bean can see each other Spring Bean...
- ...no matter which layer they are in.

- Also: No explicit dependencies

- There is no interface for a layer – what may you use?
- Solution: Add a Facade as an interface

Facade: Example

- Configurator Logic: Logic to configure a custom Mountain Bike, calculate price and delivery date
- Note the Facade and the poor man's namespace

```
<beans ...>
  <bean id="configurator-logic-facade"
   class="configurator.ConfiguratorFacadeImpl">
    <property name="deliveryCalculator"
     ref="configurator-logic-delivery-calculator" />
    <property name="priceCalculator"
     ref="configurator-logic-price-calculator" />
  </bean>
  <bean id="configurator-logic-delivery-calculator"
   class="configurator.DeliveryCalculatorImpl" />
  <bean id="configurator-logic-price-calculator"
   class="configurator.PriceCalculatorImpl" />
</beans>
```

Structured (sort of)

Same done differently

- Each components is a JAR file
- ...with its own build (Maven/ANT) script
- The JAR contains the configuration (and probably a test configuration) in a well known place

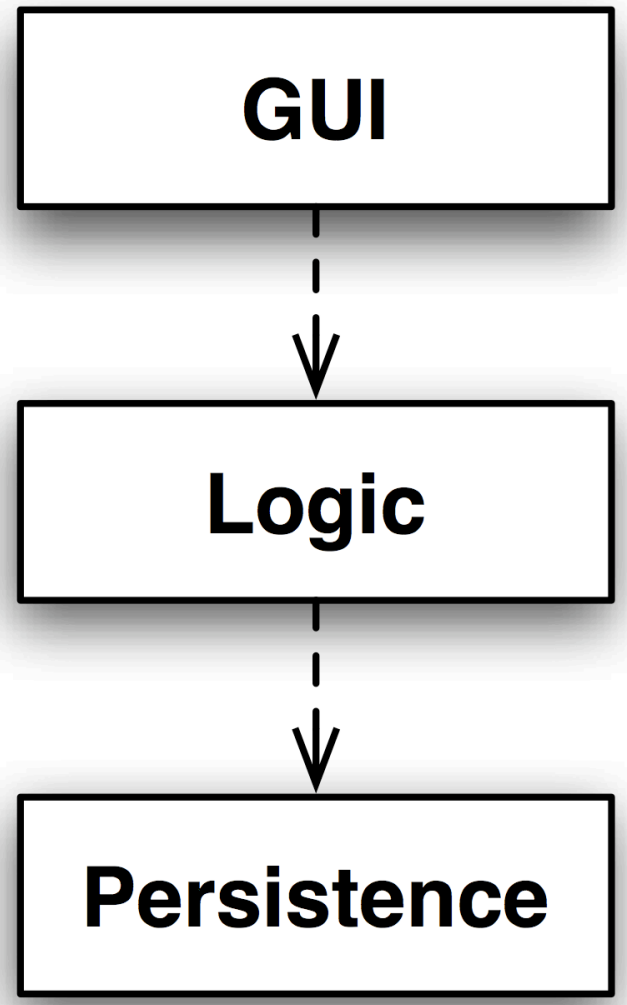
- Use classpath* to merge them:

```
ApplicationContext applicationContext =
  new ClassPathXmlApplicationContext(
    "classpath*:/config/appContext.xml");
```

Each JAR is a Component

What about layers?

- Current situation: Each component may use each other component (even in higher layers)
- Layer: Only components in a layer below may be used.
- Implemented using `ApplicationContext` hierarchy
- Popular example:
  - `ContextLoaderListener`'s `ApplicationContext` (lower layer)
  - `DispatcherServlet` specific `ApplicationContext` (higher layer)



Layer

```
ApplicationContext environmentApplicationContext =
 new ClassPathXmlApplicationContext(
  "javase.xml");

ApplicationContext persistenceApplicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "classpath*:*-persistence.xml" },
 environmentApplicationContext);

ApplicationContext logicApplicati
 new ClassPathXmlApplicationConte
  new String[] { "classpath*:*-logic.xml" },
  persistenceApplicationContext);

ApplicationContext guiApplicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "classpath*:*-gui.xml" },
  logicApplicationContext);
```

Layer

- **Is it worth it?**
  - We don’t violate layering anyway, do we? ;-)
- **And: What about the vertical slices?**
  - You care about them at least as much as you care about the layers
  - Probably more: The are units of functionality i.e. what we are paid for
  - Dependency management in this area might be more important

Vertical Slices

```
ApplicationContext environmentApplicationContext =
 new ClassPathXmlApplicationContext("javase.xml");

ApplicationContext catalogApplicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "classpath*:/catalog-*.xml"},
 environmentApplicationContext);

ApplicationContext configuratorApplicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "classpath*:/configurator-*.xml" },
 catalogApplicationContext);

ApplicationContext trackingApplicationContext =
 new ClassPathXmlApplicationContext(
  new String[] { "classpath*:/tracking-*.xml"},
configuratorApplicationContext);
```

Vertical Slices

- Same approach as for layering
- Dependencies between Vertical Slices are enforced
- But: Now layering is not enforced
- And the infrastructure does not really fit in.

Components using Spring Java Configuration

New in 3.0!

springsource
A division of vmware

© 2010 SpringSource, A division of VMware. All rights reserved

Spring Java Configuration

Strong typing, IDE support, ...

```
@Configuration
public class ConfiguratorLogic {

  @Bean
  public ConfiguratorFacade configuratorFacade() {
    ConfiguratorFacadeImpl configuratorFacade =
     new ConfiguratorFacadeImpl();
    // some configuration
    return configuratorFacade;
  }

  @Bean
  public PriceCalculator priceCalculator() {
    return new PriceCalculatorImpl();
  }


  @Autowired
  private ModelDAO modelDAO;
}
```

Just a Factory, but:
with scopes, autowiring, configuration for properties , ...

Explicit Dependency

Activated with Component Scan

```
<beans ...>

  <context:component-scan
    base-package="de.spring_book.configuration" />

</beans>
```

Spring Java Configuration: Advantages

- Hierarchical decomposition is easily possible using Java packages
- Explicit dependencies: Using `@Autowired`
- Composition: Using XML and multiple Java Configuration classes
- Best of all: No XML ☺
- But no XML namespaces
- Feels less declarative

Used to solve Visibility Problem

```
@Configuration
public class ConfiguratorLogic {

  @Bean
  public ConfiguratorFacade configuratorFacade() {
    ConfiguratorFacadeImpl configuratorFacade =
     new ConfiguratorFacadeImpl();
    // some configuration
    return configuratorFacade;
  }

  @Bean
  protected PriceCalculator priceCalculator() {
    return new PriceCalculatorImpl();
  }


  @Autowired
  private ModelDAO modelDAO;
}
```

protected:
Not visible
outside this class!

This feature is gone
Please vote for
http://jira.springframework.org/browse/SPR-7170
to bring it back!

Components using Spring Dynamic Modules for the OSGi™

© 2010 SpringSource, A division of VMware. All rights reserved

Spring Dynamic Modules 4 OSGi platforms

- OSGi offers bundles
- Bundles = JARs with special headers
- May export services and classes / interfaces
- Services can come and go at runtime
- Spring DM can export Spring Beans as OSGi services

Components using Spring

- Each component becomes a bundle
- Facade is exported
- Other services can be imported

Spring DM example

```
<beans ...>
  <osgi:service ref="facade"
   interface="configurator.ConfiguratorFacade" />

  <bean id="facade"
   class="configurator.ConfiguratorFacadeImpl">
    <property name="deliveryCalculator"
     ref="delivery-calculator" />
    <property name="priceCalculator"
     ref="price-calculator" />
  </bean>
  <bean id="price-calculator"
   class="configurator.PriceCalculatorImpl" />

  <osgi:reference id="modelDao" interface="dao.ModelDAO" />
</beans>
```

Export the Façade as OSGI service

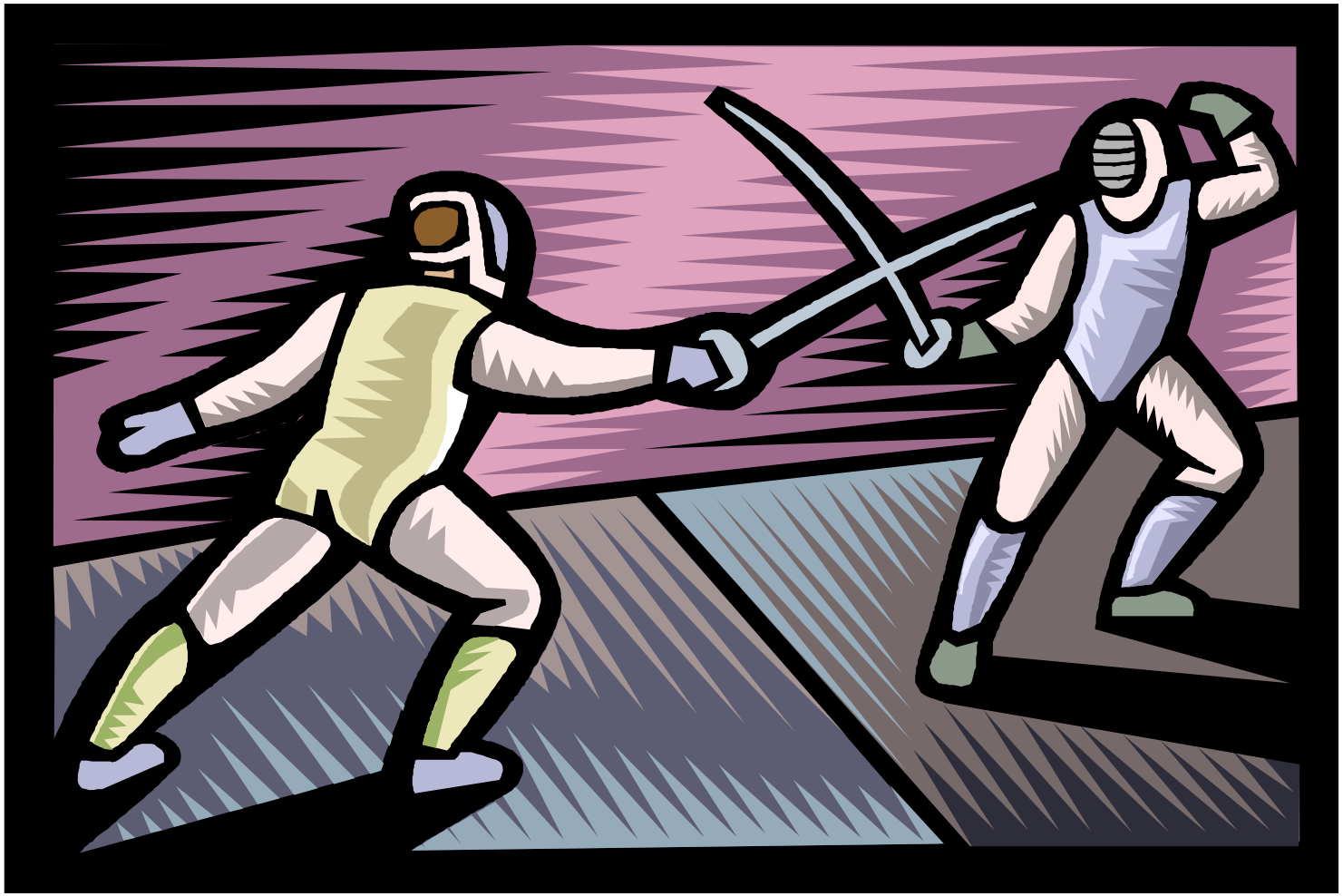
Import an OSGi service

Spring DM: Advantages

- **Facade is an OSGi Service**
  - only the Facade and the exported interfaces / classes can be accessed
- **Independent deployment**
- **Actually the focus of OSGi**

- **Strong modularization**
- **Might not be sexy but solves modularity quite nicely**

XML vs. Annotations



springsource
A division of vmware

© 2010 SpringSource, A division of VMware. All rights reserved

XML vs Annotations?

- Traditional question: Shall I use Annotations (@Service, @Component) or XML?
- That is actually not the question
- Shall I use code patterns (annotations, packages) or XML to define Spring Beans?
- …as we will see later on

Traditional Answer

- XML: structure is defined in one place
- XML is more familiar to most
- Can be used for all code – not just your code
- XML namespaces allow flexible extension
- Java Config is very similar

- Annotations for frequently changing beans
- …but configuration information is distributed
- Only works for your code

What do we actually configure?

```
public class OrderService {

  private ModelDAO mtbModelDAO;

  @Required
  public void setMtbModelDAO(ModelDAO mtbModelDAO) {
    this.mtbModelDAO = mtbModelDAO;
  }
...
}
```

- How many implementation of `ModelDAO` has the system?
- Quick Type Hierarchy reveals the answer

Stating the obvious is just a waste of time!

© 2010 SpringSource, A division of VMware. All rights reserved

Autowiring to the rescue

- **With Autowiring you is obvious configuration not needed any more**
- **What do you do if >1 compatible bean exists?**

- **Bean can be marked as only option**
  - XML: primary=true
  - Annotations: @Primary
- **…or as no option**
  - XML: autowire-candidate=false
  - default-autowire-candidates with a name pattern

Another explanation

- **Convention over configuration school**
  - Ruby on Rails, Grails etc
  - Why should I write anything obvious?
  - And I have packages etc. to structure
- **Traditional Spring school**
  - I want to configure it explicitly
  - …and see the graph in STS etc
- **I guess a project will not fail because of this decision.**

Annotations
vs.
Code Structure

springsource
A division of vmware
© 2010 SpringSource, A division of VMware. All rights reserved

Another dimension...

- So far: Decomposition into Components
- No focus on code structure
- Also important for architectures
- So what do you do about the code structure with Spring?

- We can define components
- But: What separates the different types of layers?
- What services does a layer need?

AOP: Add behavior

- Add transactions to all DAOs:

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

- Talks about code structure
- Typically each parts of an architecture needs certain services (transactions, security, your own service…)
- AOP adds the appropriate services to a components
- Can we use AOP ideas also to define components?

component-scan done differently

- Now every service implementation automatically becomes a Spring Bean
- May use regular expressions or a superclass / interface instead
- You still need @Autowired for dependencies

```
<beans ...>
 <context:component-scan
  base-package="com.springsource" >
  <context:include-filter type="aspectj"
   expression="com.springsource..service.*Impl"/>
 </context:component-scan>
</beans>
```

Naming conventions put to life

- You can create Spring Beans just by naming conventions.
- Spring != XML
- You can add behavior using AOP pointcuts based on the naming conventions

- Spring annotation can be used to create Spring Beans
- OK – can I also use Spring annotations to define pointcuts?

AspectJ Pointcuts for architecture

```
@Aspect
public class SystemArchitecture {

    @Pointcut("call(* (@Service *).*(..))")
    public void callServiceLayer() {
    }

    @Pointcut("call(* (@Repository *).*(..))")
    public void callDAOLayer() {
    }

    @Pointcut("within(@Repository *)")
    public void inDAOLayer() {
    }

}
```

So...

- **You can set up your system using package structures only**
- **Package structure become meaningful**
- **You can also define pointcuts for them**
  - to add Aspects (“log all exceptions in services!”)
  - to manage dependencies
- **You can also use Spring's annotations to define pointcuts**
- **...and you don’t depend on Spring / AspectJ in the business code at all**

It is just about how you want to define the structure of your

© 2010 SpringSource, A division of VMware. All rights reserved

Spring XML Annotations Packages…

© 2010 SpringSource, A division of VMware. All rights reserved

Sum up

springsource
A division of vmware
© 2010 SpringSource, A division of VMware. All rights reserved

Sum Up

- Spring offers a lot of flexibility to define architectures
- XML configuration is "default" but other advanced alternatives are available
- Spring Java Configuration is powerful and interesting
- Spring DM is very powerful and supports deployment best
- Component scan + pointcuts offer an alternative approach
- Your choice - decide for yourself!

- ewolff@vmware.com
- Twitter: @ewolff
- Blog: http://JandlandMe.blogspot.com