



April 2018, IPT Course  
Introduction to Spring 5

# Inversion of Control (IoC) and Dependency Injection (DI) in Spring

Trayan Iliev

[tiliev@iproduct.org](mailto:tiliev@iproduct.org)

<http://iproduct.org>

Copyright © 2003-2018 IPT - Intellectual  
Products & Technologies

# Where to Find the Demo Code?

Introduction to Spring 5 demos and examples are available @ GitHub:

<https://github.com/iproduct/course-spring5>

# Agenda for This Session

- ❖ Lookup vs. injection
- ❖ Constructor, setter and field-based DI
- ❖ Instantiating the container
- ❖ Beans and BeanFactory implementations
- ❖ Configuring ApplicationContext – basic configuration, classpath scanning and filters, component declaring meta-annotations.
- ❖ XML-based configuration: GenericXmlApplicationContext
- ❖ Java annotations configuration (@Bean, @Configuration, @ComponentScan)
- ❖ AnnotationConfigApplicationContext

# Agenda for This Session

- ❖ Mixing XML & Java - `@Import`, `@ImportResource`
- ❖ Instantiating beans using constructor and static/instance factory methods
- ❖ Dependency resolution process
- ❖ Dependencies and configuration in detail – values, bean references, inner beans, collections, maps, null handling, p- and c-namespaces, compound property names, depends-on, lazy initialization, autowiring
- ❖ Excluding a bean from autowiring
- ❖ Limitations and disadvantages of autowiring

# Component Oriented Engineering

- ❖ Ralph Johnson: Do Components Exist?  
[<http://www.c2.com/cgi/wiki?DoComponentsExist>]
- ❖ They have to exist. Sales and marketing people are talking about them. Components are not a technology. Technology people seem to find this hard to understand. Components are about how customers want to relate to software. They want to be able to buy their software a piece at a time, and to be able to upgrade it just like they can upgrade their stereo. They want new pieces to work seamlessly with their old pieces, and to be able to upgrade on their own schedule, not the manufacturer's schedule. They want to be able to mix and match pieces from various manufacturers. This is a very reasonable requirement. It is just hard to satisfy.



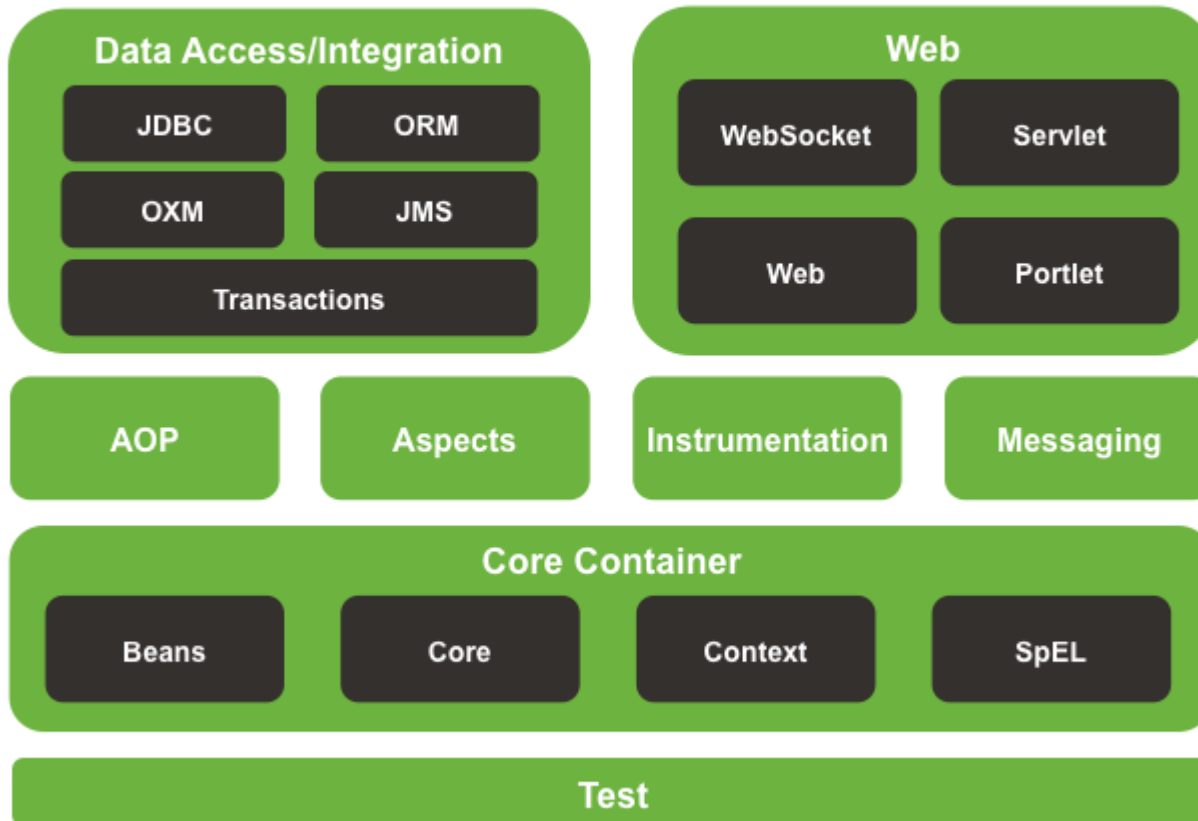
# Injection vs. Construction vs. Lookup

- ❖ **Dependency Injection (DI)** is mechanism for provisioning component dependencies and managing these dependencies throughout their life cycles
- ❖ DI is a process whereby objects define their dependencies, (the other objects they work with), only through: A) **constructor arguments**; B) **arguments to a factory method**; C) **properties or fields** that are set on the object instance; after it is constructed or returned from a factory method.
- ❖ The **container then injects** those dependencies when it creates the bean – inverse (**Inversion of Control – IoC**) of the bean itself controlling the **instantiation** or **location** of its dependencies by direct construction of classes, or a mechanism such as the **Service Locator** pattern (e.g. JNDI).

# Spring Framework 4.2 Main Modules



## Spring Framework Runtime

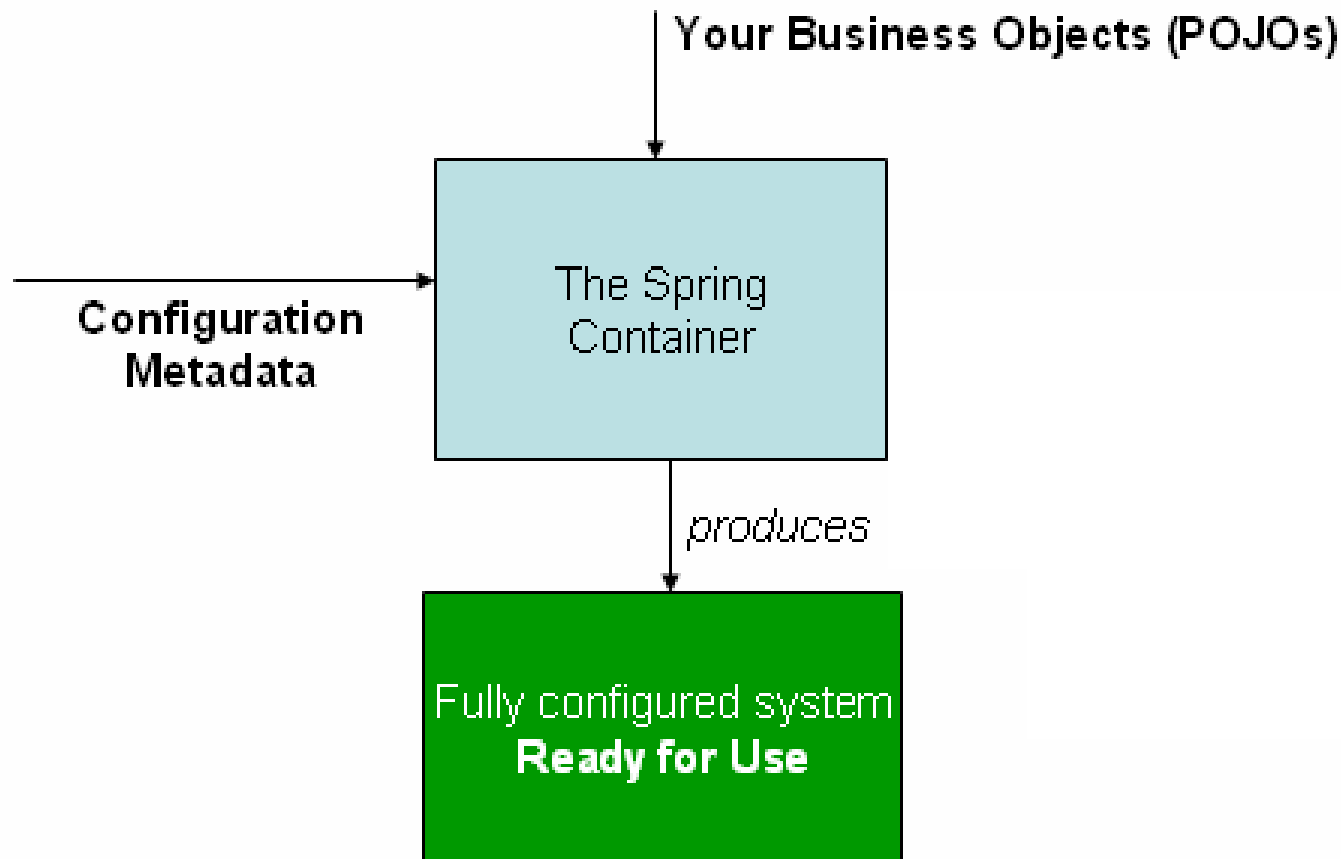


# Advantages of DI

- ❖ **Decoupling between components** – dependency only from interface (contract), not from implementation
- ❖ Easy switching between **different implementations**
- ❖ Better program **modularity** – every module has single purpose, easy replacing of modules
- ❖ Easier testing of components by isolating the component, and **mocking its dependencies**



# How Dependency Injection Works?



# Spring Beans and Bean Factories

- ❖ **Spring Beans** are POJOs managed (instantiated, assembled, etc.) by the **Spring IoC container**
- ❖ Beans, and the dependencies among them, are reflected in the **configuration metadata** used by a container.
- ❖ **Spring Framework's IoC container** base packages: [org.springframework.beans](http://org.springframework.beans), [org.springframework.context](http://org.springframework.context)
- ❖ **BeanFactory** interface provides an advanced configuration mechanism capable of managing any type of object.
- ❖ **ApplicationContext** is a sub-interface of **BeanFactory** – adds integration with Spring's **AOP features**; message **resource handling** (for use in internationalization); **event publication**; and app-layer specific contexts e.g. **WebApplicationContext**

# Interface BeanDefinition - I

- ❖ `getBeanClassName()`
- ❖ `getConstructorArgumentValues()`
- ❖ `getDependsOn()`
- ❖ `getDescription()`
- ❖ `getFactoryBeanName()`
- ❖ `getFactoryMethodName()`
- ❖ `getOriginatingBeanDefinition()`
- ❖ `getParentName()`
- ❖ `getPropertyValues()`
- ❖ `getResourceDescription()`

# Interface BeanDefinition - II

- ❖ `getRole()`
- ❖ `getScope()`
- ❖ `isAbstract()`
- ❖ `isAutowireCandidate()`
- ❖ `isLazyInit()`
- ❖ `isPrimary()`
- ❖ `isPrototype()`
- ❖ `isSingleton()`
- ❖ `setBeanClassName(String beanClassName)`
- ❖ ...

# Beans and BeanFactory Implementations

- ❖ **BeanFactory** - responsible for containing and otherwise managing the beans, provides the underlying basis for Spring's IoC functionality but it is only used directly in integration with other third-party frameworks, historical. Common implementation: **DefaultListableBeanFactory** (**XmlBeanFactory** is deprecated).
- ❖ BeanFactory related interfaces: **BeanFactoryAware**, **InitializingBean**, **DisposableBean**, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.
- ❖ Often third-party components that can not use more modern equivalents such as **@PostConstruct** or **@PreDestroy** in order to avoid a dependency on JSR-250.



# BeanFactory Lifecycle Initialization - I

1. BeanNameAware's **setBeanName**
2. BeanClassLoaderAware's **setBeanClassLoader**
3. BeanFactoryAware's **setBeanFactory**
4. ResourceLoaderAware's **setResourceLoader** (application context only)
5. ApplicationEventPublisherAware's **setApplicationEventPublisher** (application context only)
6. MessageSourceAware's **setMessageSource** (app context)
7. ApplicationContextAware's **setApplicationContext**
8. ServletContextAware's **setServletContext** (web application context only)

# BeanFactory Lifecycle - II

9. BeanPostProcessors' **postProcessBeforeInitialization** method
10. InitializingBean's **afterPropertiesSet** (or **@PostConstruct**)
11. a custom **init-method** definition
12. BeanPostProcessors' **postProcessAfterInitialization** method

## ❖ On BeanFactory shutdown:

1. DisposableBean's **destroy** (or **@PreDestroy**)
2. a custom **destroy-method** definition

# Types of IoC Lookup

- ❖ **Dependencies Pull (service locator pattern)** – e.g. JNDI API for programmatic lookup for EJB components in J2EE. Spring also supports dependencies lookup – for example:

```
ArticlePresenter presenter =  
    ctx.getBean("presenter", ArticlePresenter.class);
```

- ❖ **Contextualized Dependencies Lookup** – the lookup is performed against specific container (context) managing the resource, not from a single centralized registry

# Types of IoC DI

## ❖ Constructor-based DI:

`@Autowired` // or `@Inject`

```
public ConsoleArticlePresenter(ArticleProvider provider) {  
    this.provider = provider;  
}
```

```
<bean id="provider" class="org.iproduct.MockArticleProvider"/>  
<bean id="presenter" name="presenter" class="...">  
    <constructor-arg type="ArticleProvider">  
        <ref bean="provider" />  
    </constructor-arg>  
</bean>
```

OR

```
<constructor-arg type="ArticleProvider" index="0"  
    name="provider" ref="provider" /> OR  
  
<bean id="presenter" name="presenter" class="..."  
    c:provider-ref="provider" />
```

# Types of IoC DI

## ❖ Static factory method-based DI:

```
<bean id="provider" factory-method="createInstance"  
      class="org.iproduct.spring.xmlconfig.MockArticleProvider"/>
```

## ❖ Instance factory method-based DI:

```
<bean id="presenterFactory" name="presenterFactory"  
      class="org.iproduct.spring.xmlconfig.ArticlePresenterFactory"  
      c:provider-ref="provider"/>  
<bean id="presenter" name="presenter" factory-  
      bean="presenterFactory" factory-method="createPresenter" />
```



# Types of IoC DI

## ❖ Property (setter) -based DI:

```
@Autowired //@Inject //@Resource  
public void setArticleProvider(ArticleProvider provider) {  
    this.provider = provider;  
}
```

```
<bean id="presenter" name="presenter"  
    class="org.iproduct.spring.xmlconfig.ConsoleArticlePresenter"  
    p:articleProvider-ref="provider"/>
```

## ❖ Field-based DI

```
@Autowired //@Inject or @Resource  
private ArticleProvider provider;
```

# Application Context Implementations

- ❖ **FileSystemXmlApplicationContext** – loads the bean definitions from an XML file using provided full path of the XML bean configuration file in the constructor argument.
- ❖ **ClassPathXmlApplicationContext** – loads the bean definitions from an XML file using provided configuration file available on the java CLASSPATH
- ❖ **XmlWebApplicationContext** – loads the XML file with bean definitions from within a web application
- ❖ **AnnotationConfigApplicationContext** – with bean definitions based on annotation configuration
- ❖ **GenericApplicationContext** – allows to programmatically register beans, and then call `ctx.refresh()` ;

# Basic XML-Based Config Example

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <bean id="provider"
          class="org.iproduct.spring.xmlconfig.MockArticleProvider"/>

    <bean id="presenter" name="presenter"
          class="org.iproduct.spring.xmlconfig.ConsoleArticlePresenter"
          c:provider-ref="provider" />
</beans>
```

# XML Config Example using Factories

```
<?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:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-
       context.xsd">
  <bean id="provider" factory-method="createInstance"
        class="org.iproduct.spring.xmlconfig.MockArticleProvider"/>
  <bean id="presenterFactory" name="presenterFactory"
        class="org.iproduct.spring.xmlconfig.ArticlePresenterFactory"
        c:provider-ref="provider"/>
  <bean id="presenter" name="presenter" factory-bean=
    "presenterFactory" factory-method="createPresenter" /></beans>
```

# XML Config - Inner Beans

```
<bean id="article" class="org.iproduct.spring.xmlconfig.Article"
scope="prototype">
```

*<!-- instead of using a reference to a target bean, simply  
define the target bean inline -->*

```
<property name="author">
```

```
<bean class="org.iproduct.spring.xmlconfig.Author">
```

*<!-- this is the inner bean -->*

```
<property name="name" value="Fiona Apple"/>
```

```
<property name="age" value="25"/>
```

```
</bean>
```

```
</property>
```

```
</bean>
```



# Bean Definition Inheritance & Prop Merging

```
<bean id="parent" abstract="true"
class="org.iproduct.spring.xmlconfig.Author">
  <property name="emails">
    <props>
      <prop key="administrator">fiona@example.com</prop>
      <prop key="support">support@example.com</prop>
    </props>
  </property>
</bean>
<bean id="author" parent="parent">
  <property name="emails">
    <!-- merge specified on child collection definition -->
    <props merge="true">
      <prop key="sales">f.apple@gmail.com</prop>
      <prop key="support">support@example.co.uk</prop>
    </props>
  </property>
</bean>
```

# Annotation-Based Configuration

```
@Configuration
@PropertySource("classpath:articles.properties")
@ComponentScan(basePackages = "org.iproduct.spring.programmatic")
public class SpringProgrammaticAnnotationConfig {
    @Value("${listOfValues}")
    private String[] articleTitles;

    @Bean
    public ArticleProvider provider() {
        return new MockArticleProvider(articleTitles);
    }

    @Bean
    public ArticlePresenter presenter() {
        ArticlePresenter presenter =
            new ConsoleArticlePresenter();
        presenter.setArticleProvider(provider());
        return presenter;
    }
}
```

# Classpath Scanning and Filters

## ❖ Annotation-based configuration:

```
@Configuration
@PropertySource("classpath:articles.properties")
@ComponentScan(basePackages = "org.example",
    includeFilters = @ComponentScan.Filter(
        type = FilterType.REGEX, pattern = ".*Stub.*Repository"),
    excludeFilters = @ComponentScan.Filter(Repository.class))
public class SpringProgrammaticAnnotationConfig {
    ...
}
```

## ❖ XML-based configuration:

```
<context:component-scan base-package="org.example">
    <context:include-filter type="regex"
        expression=".*Stub.*Repository"/>
    <context:exclude-filter type="annotation"
        expression="org.springframework.stereotype.Repository"/>
</context:component-scan>
```

# Problem 1 - Comments: Annotation DI

We want to develop a simple comments handling service and client, capable of storing comments in memory and listing them on the console. Please, implement the following artifacts:

1. A **Comments** model class with 3 attributes – 1) comment text, 2) author email, 3) comment date and time
2. **CommentsService** interface with three business methods:
  1. void **addComment(Comment comment)**
  2. List<Comment> **getAllComments()**
  3. List<Comment> **getCommentsByEmail(String email)**
3. **CommentsServiceImpl** class implementing the above methods (preferably using Java 8 Stream API and lambdas)

# Problem 1 - Comments: Annotation DI

( - continues - )

4. **CommentsLoggerService** interface with following methods:
  1. void **dumpAllComments()**
  2. void **dumpCommentsByAuthor(String authorEmail)**
5. **CommentsConsoleLoggerImpl** class implementing **CommentsLoggerService** interface
6. Class **CommentsDemoAnnotationDI** wiring the above services using Spring 5 **AppricationContext** and property-based annotation DI. The main method should add 5 comments by 2 authors and dump to console: 1) all comments, 2) comments of the first author only.



# Domain Driven Design (DDD)

We need tools to cope with all that complexity inherent in robotics and IoT domains.

Simple solutions are needed – cope with problems through divide and concur on different levels of abstraction:

**Domain Driven Design (DDD)** – back to basics:  
domain objects, data and logic.

Described by Eric Evans in his book:

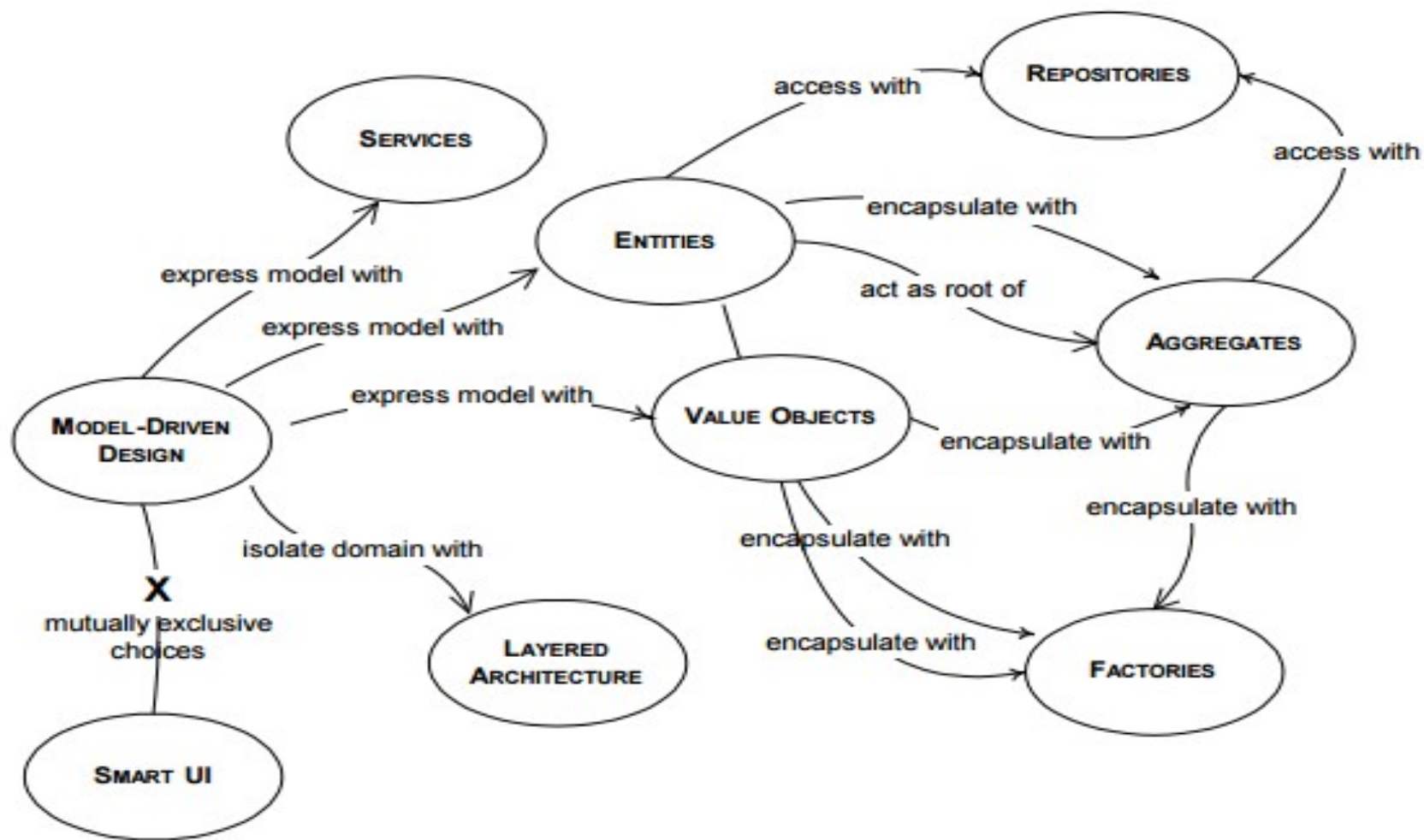
Domain Driven Design: Tackling Complexity in the Heart of Software, 2004

# Domain Driven Design (DDD)

## Main concepts:

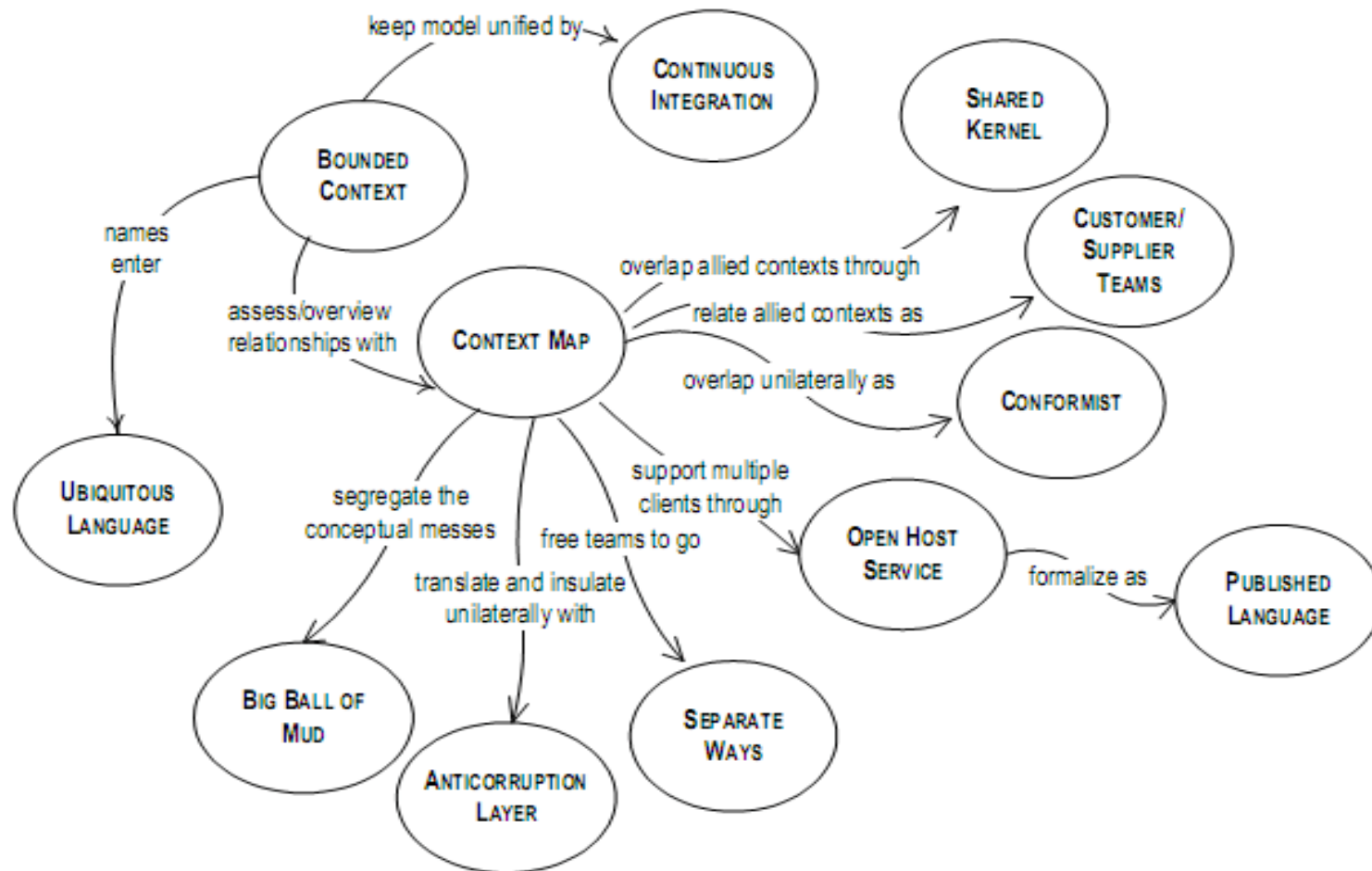
- ❖ Entities, value objects and modules
- ❖ Aggregates and Aggregate Roots [Haywood]:  
**value < entity < aggregate < module < BC**
- ❖ Repositories, Factories and Services:  
**application services <-> domain services**
- ❖ Separating interface from implementation

# Domain Driven Design (DDD)



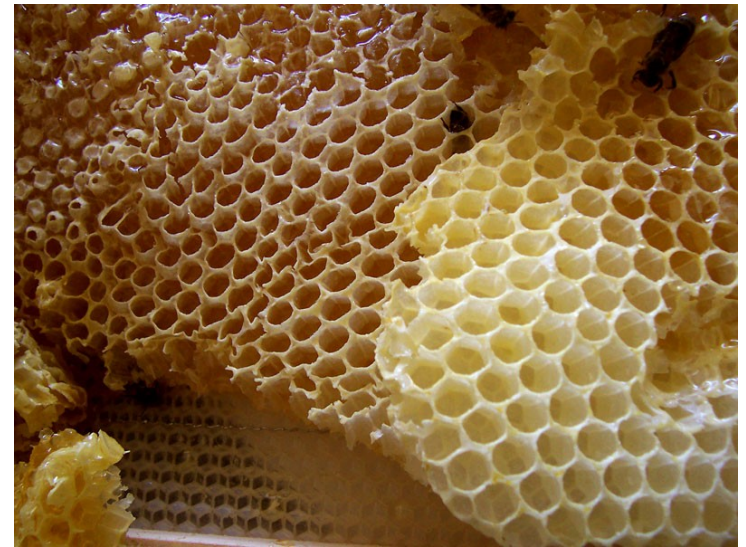
# Domain Driven Design (DDD)

## Maintaining Model Integrity



# Domain Driven Design (DDD)

- ❖ Ubiquitous language and Bounded Contexts
- ❖ DDD Application Layers:  
Infrastructure, Domain, Application, Presentation
- ❖ Hexagonal architecture :  
OUTSIDE  $\leftrightarrow$  transformer  $\leftrightarrow$   
( application  $\leftrightarrow$  domain )  
[A. Cockburn]

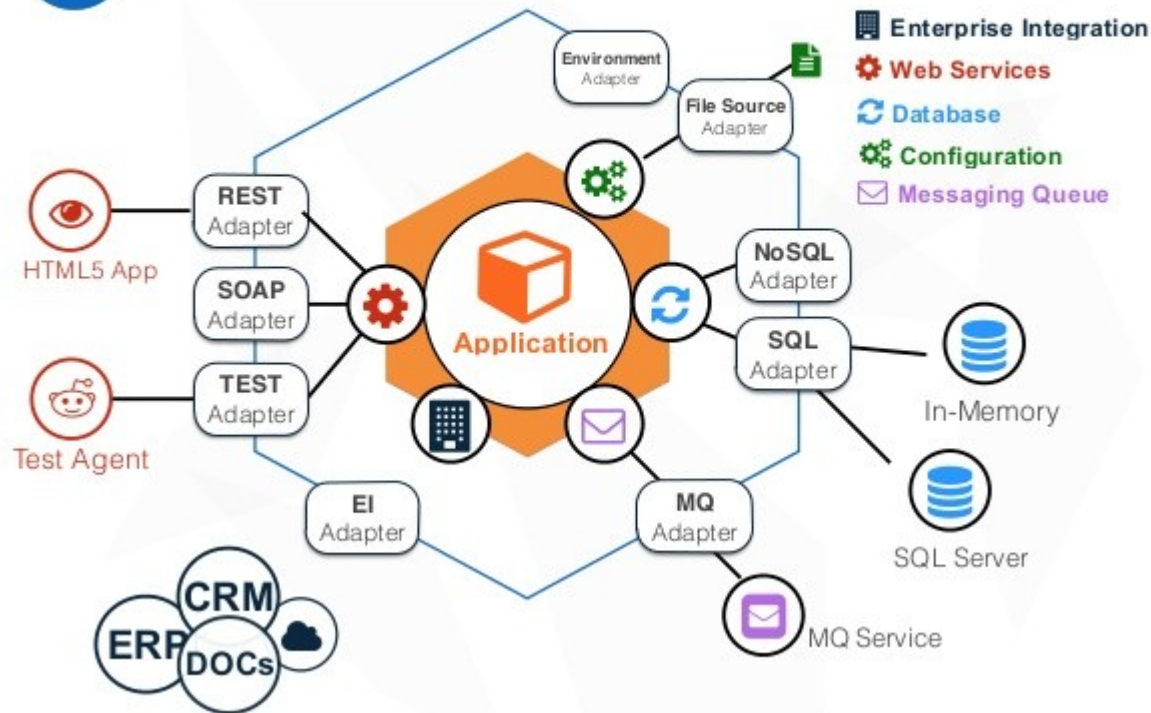




# Hexagonal Architecture

02

## Overview





# Hexagonal Architecture Principles

- ❖ Allows an application to equally be driven by **users, programs, automated test or batch scripts**, and to be developed and tested in isolation from its eventual run-time devices and databases.
- ❖ As events arrive from the outside world at a port, a **technology-specific adapter** converts it into a **procedure call** or **message** and passes it to the application
- ❖ Application sends messages through **ports** to **adapters**, which signal data to the receiver (human or automated)
- ❖ The application has a **semantically sound interaction** with all the adapters, **without actually knowing the nature of the things** on the other side of the adapters

# Bean Stereotype Annotations

- ❖ **@Component** - generic annotation, ensuring that the class will be found during classpath scanning and that it will be registered in the context as a bean
- ❖ **@Service** - business logic of the application (domain service in DDD terms)
- ❖ **@Repository** – DAO, data access layer, automatic persistence exception translation (**DataAccessExeption**, **PersistenceExceptionTranslationPostProcessor**)
- ❖ **@Controller** - marks class as Spring MVC controller
- ❖ **Meta annotations** – for example:  
**@RestController** = **@Controller** + **@ResponseBody**

# Bean Scopes

- ❖ **singleton** - (Default) Scopes a single bean definition to a single object instance per Spring IoC container.
- ❖ **prototype** - Scopes a single bean definition to any number of object instances
- ❖ **request** - scopes a single bean definition to the lifecycle of a single HTTP request; Only valid in the context of a web-aware Spring ApplicationContext.
- ❖ **session** - scopes a single bean to an HTTP Session
- ❖ **application** - scopes bean to a ServletContext
- ❖ **websocket** - scopes a single bean definition to the lifecycle of a WebSocket.

# Dependency Resolution Process

- ❖ The **ApplicationContext** is created and initialized with configuration **metadata describing all beans**. This metadata can be specified using XML, Java code, or annotations.
- ❖ For each bean, its dependencies are expressed in the form of **properties**, **constructor arguments**, or **arguments to the static-factory method** (if used instead of constructor).
- ❖ Each property or constructor argument is an actual definition of the **value to set**, or a **reference to another bean**.
- ❖ Each property or constructor argument which is a **value is converted from its specified format to the actual type** of that property or constructor argument. Spring can convert a supplied string value to all built-in types: int, long, String, etc.

# Lazy Bean Instantiation

## ❖ Using XML config:

```
<bean id="provider" factory-method="createInstance"  
      class="org.iproduct.MockArticleProvider" lazy-init="true"/>
```

## ❖ Using Java annotation config: **@Lazy**

- May be used on any class annotated with **@Component** or on methods annotated with **@Bean**
- If this annotation is not present on a **@Component** or **@Bean** definition, eager initialization will occur.
- If present on a **@Configuration** class, this indicates that all **@Bean** methods within that should be lazily initialized.
- If placed on **@Autowired** or **@Inject** injection points it leads to creation of a lazy-resolution proxy (as an alternative of using **ObjectFactory** or **Provider**).



# Autowiring

- ❖ **Autowiring** = allowing Spring to **resolve collaborators** (other beans) automatically for your bean by inspecting the contents of the **ApplicationContext**
- ❖ Autowiring can **significantly reduce** the need to specify **properties** or **constructor arguments**.
- ❖ Autowiring can **update a configuration** as your objects evolve – e.g. if you need to add a dependency to a class, that dependency can be satisfied automatically **without you needing to modify the configuration**.
- ❖ Autowiring can be especially useful during development, without negating the option of **switching to explicit wiring when the code base becomes more stable**.



# Autowiring – How To

- ❖ When using XML-based configuration metadata, you specify **autowire mode** for a bean definition with the **autowire attribute** of the **<bean/>** element.
- ❖ Example:

```
<bean id="articleProvider" factory-method="createInstance"  
      class="org.iproduct.spring.xmlconfig.MockArticleProvider"  
      autowire-candidate="true"/>
```

```
<bean id="presenter" name="presenter"  
      class="org.iproduct.spring.xmlconfig.ConsoleArticlePresenter"  
      autowire="byName"/>
```

# Types of Autowiring - I

- ❖ **no** – This option is default for spring framework and it means that autowiring is OFF. You have to explicitly set the dependencies using **<property>** tags in bean definitions.
- ❖ **byName** – This option enables the dependency injection **based on bean names**. When autowiring a property in bean, **property name** is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. Otherwise an error is thrown.
- ❖ **byType** – This option enables the dependency injection based on **bean types**. When autowiring a property in bean, **property's class type** is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property. Otherwise an error is thrown.

# Types of Autowiring - II

- ❖ **constructor** – similar to **byType**, but applies to constructor arguments. In autowire enabled bean, it will look for **class type of constructor arguments**, and then do a autowire by type on all constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, an error is thrown.
- ❖ **autodetect** – uses either of two modes i.e. constructor or byType modes. First it will try to look for **valid constructor with arguments**, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the **autowire byType mode is chosen**.

# Autowiring Annotations

- ❖ **@Autowired** – marks a **constructor, field, setter method** or **config method** as to be autowired by Spring's dependency injection facilities. Has a **required** parameter (true by default) If field/parameter is of **Collection** or **Map**, the container autowires **all beans matching the declared type** – **@Order** or **@Priority** annotations can be used to define order. Exact match sequence is: **1) by Type; 2) by Qualifier; 3) by Name**
- ❖ **@Inject** – part of JSR-330 Java standard, similar to **@Autowired**, has no 'required' parameter.
- ❖ **@Resource** – part of JSR-250. Injection by bean name is preferred – there is a '**name**' parameter. The sequence of matching is: **1) by Name; 2) by Type; 3) by Qualifier**

# Autowiring Circular Dependencies

- ❖ The circular injection problem: **Bean A** → **Bean B** → **Bean A**
- ❖ If **Bean A** → **Bean B** → **Bean C** – Spring will create bean C, then bean B (injecting bean C into it), then create bean A (injecting bean B into it). But, when having a circular dependency, Spring cannot decide which should be created first – **BeanCurrentlyInCreationException** is raised.
- ❖ This happens when using constructor injection. If using other types of injection, there is no problem since dependencies will be injected when they are needed and not on loading.
- ❖ Solutions: 1) Redesign; 2) using **@Lazy**; 3) using **setters**; 4) using **@PostConstruct**; 5) using **ApplicationContextAware** and **InitializingBean** + manual bean lookup in init method



# Disadvantages of Autowiring

- ❖ Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called simple properties such as primitives, Strings, and Classes (and arrays of such simple properties)
- ❖ Autowiring is less exact than explicit wiring.
- ❖ Wiring information may not be available to tools that may generate documentation from a Spring container.
- ❖ Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not a problem. When expected a single value, this ambiguity is not arbitrarily resolved, but an exception is thrown.



# Additinal Examples

Learning Spring 5 book examples are available @ GitHub:

<https://github.com/PacktPublishing/Learning-Spring-5.0>

Spring 5 Core Reference Documentation: Learning

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html>

Spring 5 @Configuration annotation Javadoc:

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html>

# Thank's for Your Attention!



**Trayan Iliev**

**CEO of IPT – Intellectual Products  
& Technologies**

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>

<https://plus.google.com/+IproductOrg>