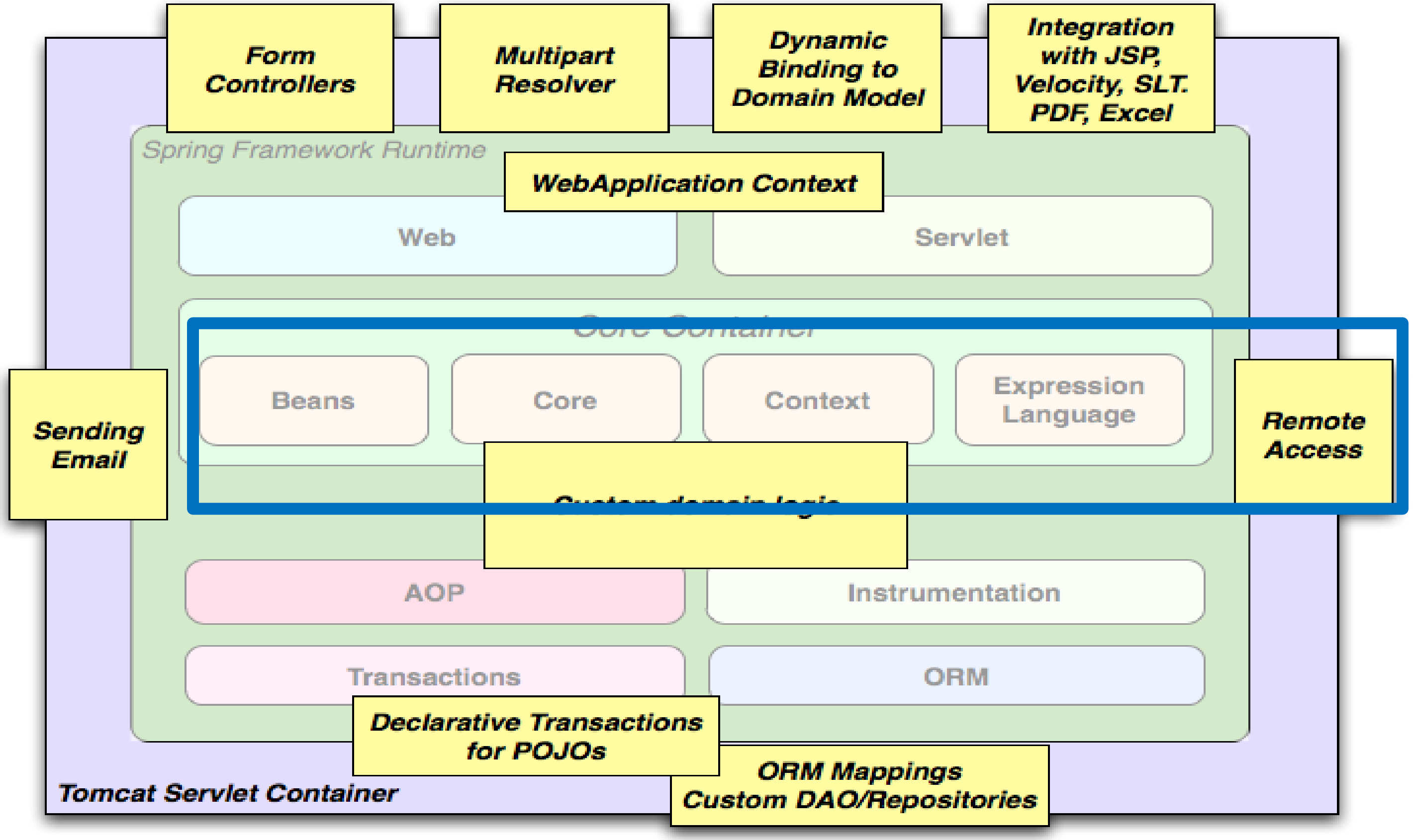
Core Spring Framework

Water the Root

Spring Core


Form Controllers
Multipart Resolver
Dynamic Binding to Domain Model
Integration with JSP, Velocity, SLT. PDF, Excel
Spring Framework Runtime
WebApplication Context
Web
Servlet
Core Container
Beans
Core
Context
Expression Language
Sending Email
Remote Access
Custom domain logic
AOP
Instrumentation
Transactions
ORM
Declarative Transactions for POJOs
ORM Mappings Custom DAO/Repositories
Tomcat Servlet Container

Spring Core Technologies

- **IoC *****

  Inversion of Control Container
- **AOP *****

  Aspect-Oriented Programming
- **Validation*****

  Data Validation W/pluggable interface
- **SpEL ****

  Spring Expression Language that supports querying and manipulating an object graph at runtime.
- **Resource ****

  Common API that abstracts the type of underlying resource such as a URL, file or class path resource.

Core Technologies

Technologies absolutely integral to the Spring Framework.

Foremost is the Inversion of Control (IoC) container.

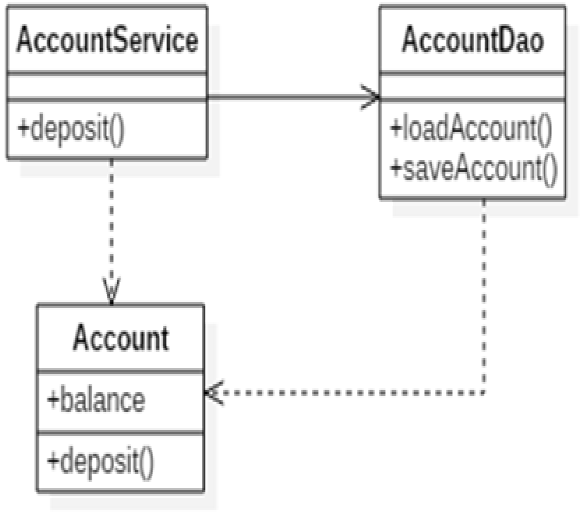
Understanding Spring & DI

- 4 ways connect objects together
  - Instantiate Objects Directly
  - P2I for more flexibility, but still instantiate
  - Use a factory object to instantiate
  - Use Spring and DI

Instantiate Objects Directly

```
public class AccountService {
  private AccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



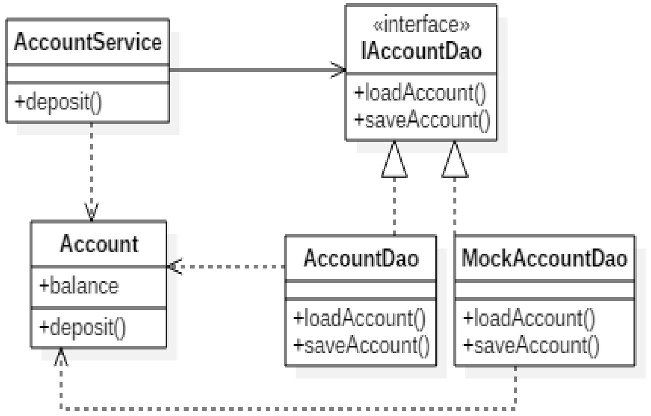
- Relation between AccountService and AccountDao is hardcoded
  - To Change AccountDao implementation have to change the code

Use an Interface

Flexibility

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    accountDAO = new AccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



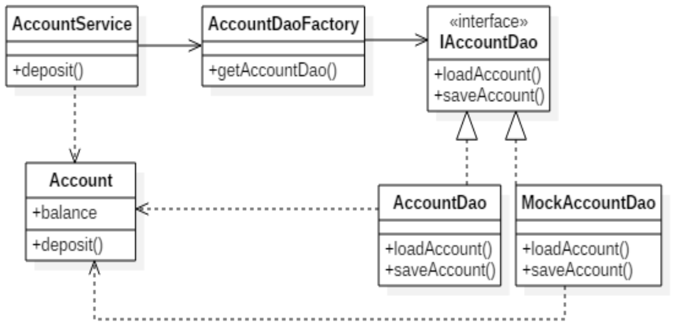
By using an interface (P2I) we've gained the flexibility (two implementations)

- But the relationship is still hard coded
- To switch, we still have to change code

Use a Factory

```
public class AccountService {
  private IAccountDAO accountDAO;

  public AccountService() {
    AccountDAOFactory daoFactory = new AccountDAOFactory();
    accountDAO = daoFactory.getAccountDAO();
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Hardcoded



The relation between AccountService and AccountDao is still hardcoded

– We have more flexibility, but if you want to change the AccountDao implementation you have to change the code in the AccountDaoFactory

Spring Dependency Injection

```
public class AccountService {
  private IAccountDAO accountDAO;

  public void setAccountDAO(IAccountDAO accountDAO) {
    this.accountDAO = accountDAO;
  }

  public void deposit(long accountNumber, double amount){
    Account account=accountDAO.loadAccount(accountNumber);
    account.deposit(amount);
    accountDAO.saveAccount(account);
  }
}
```

Setter for Injection

Config for creating and Injecting

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

What if the Factory created both Account Service and the AccountDao?

What if the Factory could read a config file?

That's in essence what Spring does

Summary

- Spring is a container for the service layer
  - Started as a replacement EJB container
  - Focuses on POJO based (flexible, best practices)
- In essence spring is a fancy factory that:
  - reads a config file, creates objects, and connects them

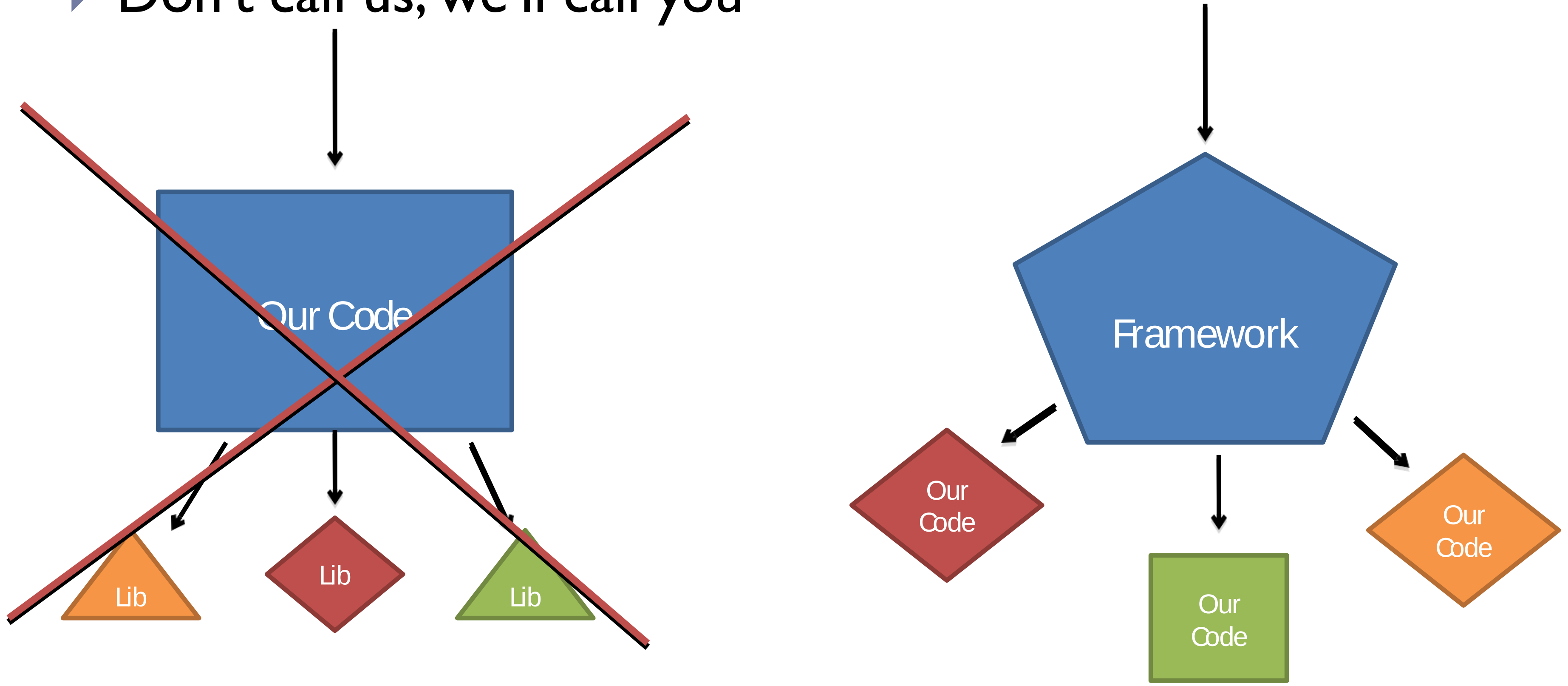
Containers

- A container is a piece of software that manages your objects.
- Common Examples:
  - Web (servlet) Container
  - EJB / Spring Bean Container
  - JPA EntityManager

- The primary principle that containers use is **Inversion of Control (IoC)**

IOC/ Hollywood Principle

- Don’t call us, we’ll call you



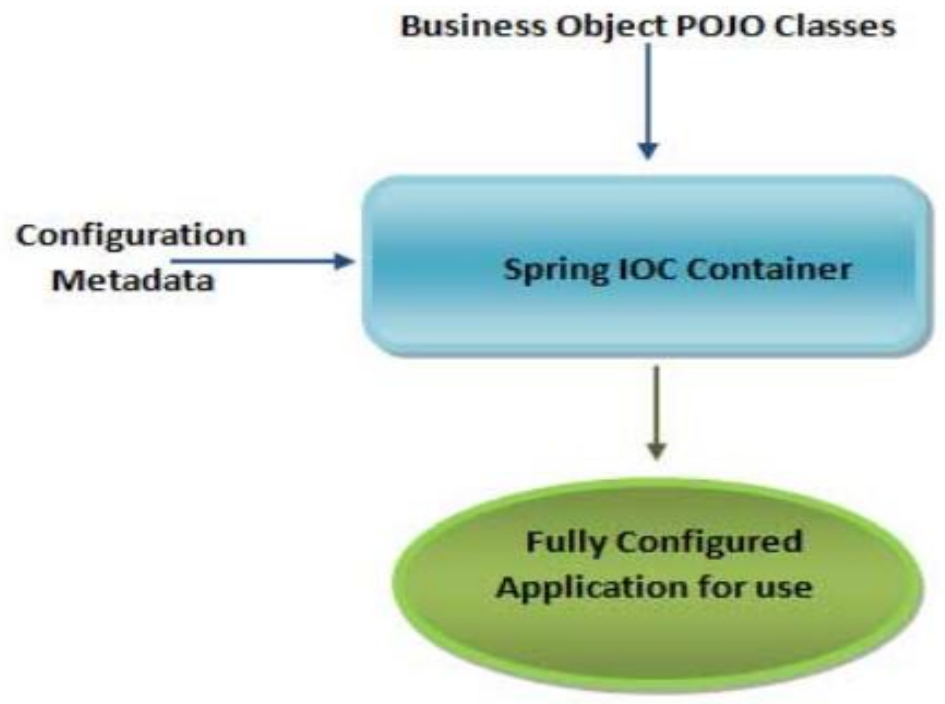
Model © Prof. Rene de Jong

Dependency Injection [DI]

- **Reduces “glue” code**
  - Less setup of dependency in component
- **Simplifies Configuration**
  - Declaratively re-configure to change implementations
- **Improves Testability**
  - Substitute “mock” implementations
- **Fosters good design**
  - Design to Interfaces….

Spring Core – IoC Container

The Essence of a Spring Application


Business Object POJO Classes
Configuration Metadata
Spring IOC Container
Fully Configured Application for use

JavaBean vs POJO vs Spring Bean

- JavaBean
  - Adhere to Sun’s JavaBeans specification
  - Implements Serializable interface
  - Must have default constructor, setters & getters
  - Reusable Java classes for visual application composition
- POJO
  - ‘Fancy’ way to describe ordinary Java Objects
  - Doesn’t require a framework
  - Doesn’t require an application server environment
  - Simpler, lightweight compared to ‘heavyweight’ EJBs
- Spring Bean
  - Spring managed - configured, instantiated and injected

- A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

- **Spring Documentation**:
  - “Component” is used interchangeably with POJO class. Both mean a Java class from which an object instance is created.
  - “Bean” is used interchangeably with POJO instance. Both mean object instance created from a Java class.

Main Point

The Inversion of Control Container manages the lifecycle for the objects required by our application, allowing us to focus on the functionality of our logic and giving flexibility for future implementations.

Science of Consciousness: *Through the holistic field of life, the full range of life, the pure nature of creative intelligence, we can enrich all aspects of life.*

Hello World

```
package edu.mum;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

BUT the Message is HARDCODED!

"Externalize" Message & Display mechanisms

Modularized "bean" configuration

```
public class ConfigInMemory {

   // Singleton Instance
   private static final ConfigInMemory instance = new ConfigInMemory();

   Map<String, Object> beans = new HashMap<String, Object>();

    // PRIVATE constructor -  No other classes can create an instance...
    private ConfigInMemory() {

       // HARD Code the "managed" beans
       StandardOutMessageDisplay standardOutMessageDisplay = new StandardOutMessageDisplay();
       HelloWorldMessageSource helloWorldMessageSource = new HelloWorldMessageSource();
       // Manually DI
       standardOutMessageDisplay.setMessageSource(helloWorldMessageSource);

       beans.put("MessageDisplay", standardOutMessageDisplay);
       beans.put("MessageSource", helloWorldMessageSource);
    }

    // Access Singleton instance
    public static ConfigInMemory getInstance() {
        return instance;
    }

    public Object getBean(String key) {
        return beans.get(key);
    }
}
```

HelloWorld Redesigned

Here's the "NEW" Look

```
public class HelloWorldReDesigned {
    public static void main(String[] args) {

        // "Configure" application - set up HARDCODED managed beans
        ConfigInMemory configInMemory = ConfigInMemory.getInstance();

        MessageDisplay messageDisplay = (MessageDisplay)
configInMemory.getBean("MessageDisplay");
        messageDisplay.display();

    }
}
```

Nice Improvement – SoC, Modularization, Resource Management

BUT WAIT! – the "Managed Beans" are STILL HARDCODED!!!
AND – the Dependency Injection is STILL Manual!!

Externalize Configuration of Resources

```
public class MessageConfiguration {
    // Is invoked only once [since final] ... happens when we call getInstance() below
    private static final MessageConfiguration instance = new MessageConfiguration();

    Map<String, Object> beans = new HashMap<String, Object>();

    private Properties properties;

    // PRIVATE: No other classes can create an instance...
    private MessageConfiguration() {
        properties = new Properties();
        InputStream input = null;

        String fileName = "HelloWorld.properties";

            // Use ClassLoader to find resource...
            input = this.getClass().getClassLoader().getResourceAsStream(fileName);
            if (input == null) {
                System.out.println("Unable to find " + fileName);
                return;
            }
            //load in properties file data
            properties.load(input);
            // Enumerate through declared components
            Object bean = null;
            Enumeration enumeration = properties.keys();
            while (enumeration.hasMoreElements()) {
                String key = (String) enumeration.nextElement();
                bean = ObjectFactory.getInstance((String) properties.get(key));
                beans.put(key, bean);
            }

        // Process Annotations
        ProcessAnnotations.handleAnnotations(beans);
    }
}
```

Get components "declaratively" from properties file

Build map of component instances [beans]

@Autowired Annotation

```
@Retention(java.lang.annotation.RetentionPolicy.RUNTIME)
@Target({java.lang.annotation.ElementType.FIELD})
public @interface AutoWired {}
```

```
public class StandardOutMessageDisplay implements MessageDisplay {

   @AutoWired
   private MessageSource messageSource;

    public void display() {
        if (messageSource == null) {
            throw new RuntimeException(
                "You must set the property messageSource of class:"
                + StandardOutMessageDisplay.class.getName());
        }

        System.out.println(messageSource.getMessage());
    }


}
```

We are using @Autowired Annotation to implement Dependency Injection

Add External Configuration [Cont.]

```
public class HelloWorldReDesignedWithConfiguration {
    public static void main(String[] args) {
       MessageConfiguration messageConfiguration =
MessageConfiguration.getInstance();

        MessageDisplay messageDisplay = (MessageDisplay)
messageConfiguration.getBean("MessageDisplay");
        messageDisplay.display();
    }
}
```

Here's the "NEW" Look

Lookin' Pretty GOOD –
BUT STILL lots of CUSTOM "glue" code…
Reading configuration file…

Spring Solution

```
public class HelloWorld {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new ClassPathXmlApplicationContext("spring/applicationContext.xml");
        MessageDisplay messageDisplay = applicationContext.getBean("display", MessageDisplay.class);
        messageDisplay.display();
    }
}
```

IOC – Dependency Lookup

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="source" class="edu.mum.component.impl.HelloWorldMessageSource"/>

    <bean id="display" class="edu.mum.component.impl.StandardOutMessageDisplay" >
        <property name="messageSource" ref="source" />
    </bean>
 </beans>
```

Spring manages components

Spring “injects” source dependency [DI]

```
Or:
<context:component-scan base-package= "edu.mum" />
```

SEE Demo HelloWorldSpringX HelloWorldSpringAX

Spring Configuration Metadata

- XML based
  - Wire components without touching their source code or recompiling them.
  - CLAIM: Annotated classes are no longer POJOs ****
  - Configuration centralized and easier to control.
- Annotation [Version 2.5]
  - Component wiring close to the source
  - Shorter and more concise configuration.
- JavaConfig [Version 3.0]
  - Define beans external to your application classes by using Java rather than XML files

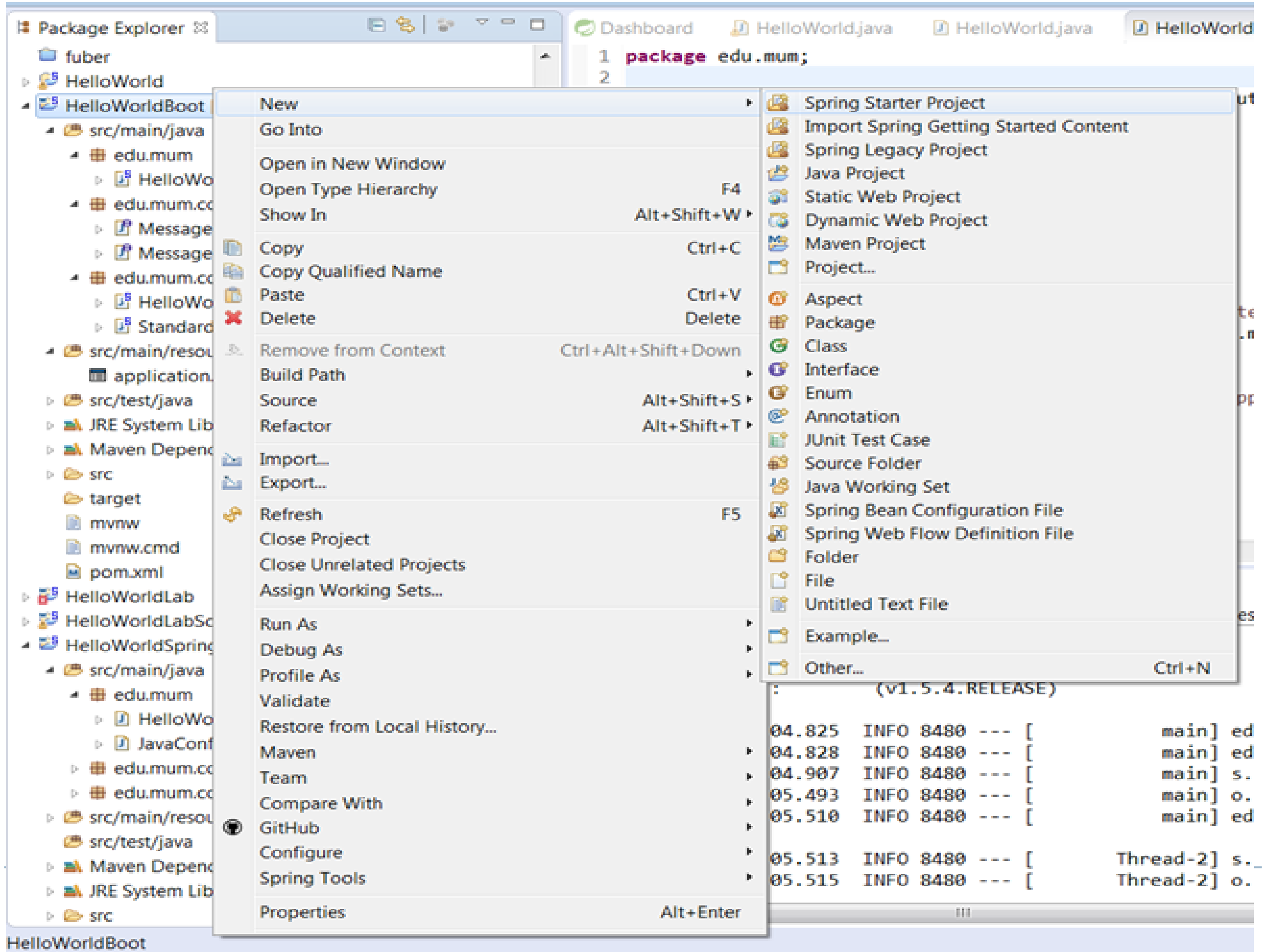
- Annotation injection is performed *before* XML injection. Therefore XML injection takes precedence over Annotation injection. It is the "last word"

Annotation Based JavaConfig Configuration

```
@Configuration
@ComponentScan("edu.mum")
public class JavaConfiguration {
}


public class HelloWorld {
    public static void main(String[] args) {
        ApplicationContext applicationContext = new
AnnotationConfigApplicationContext(JavaConfiguration.class);
        applicationContext.getBean(MessageDisplay.class).display();
    }

}
```

SEE Demo HelloWorldSpringAJ

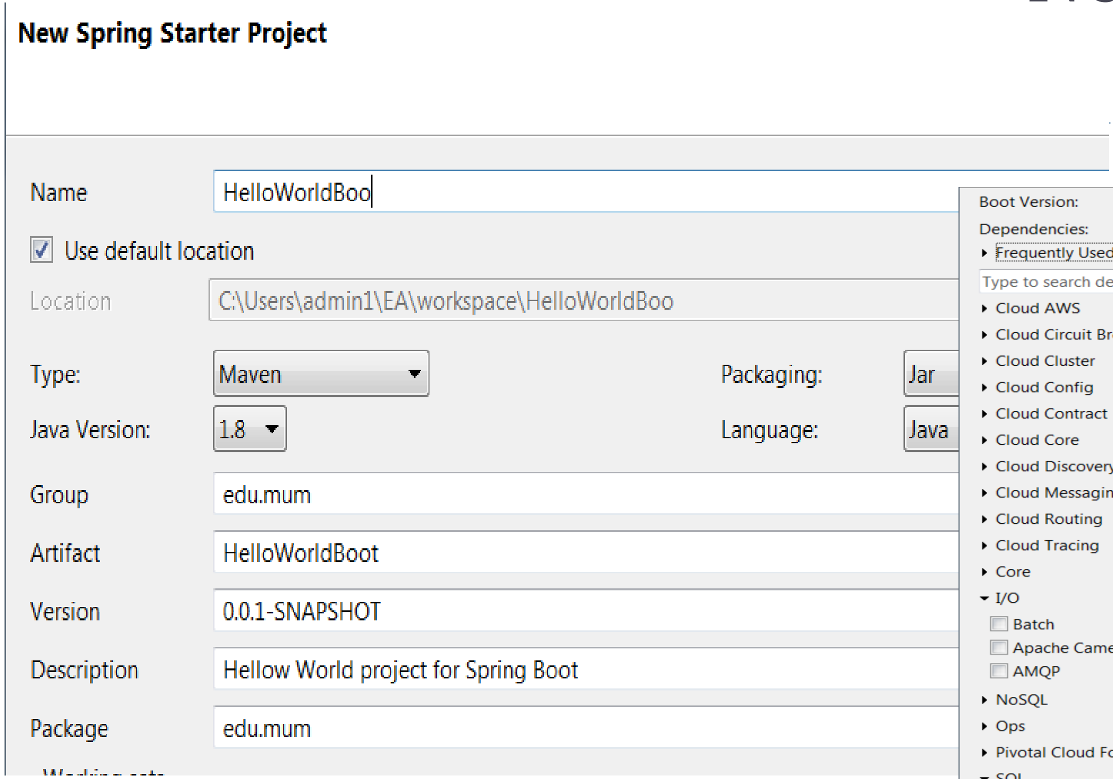
Spring Boot Project



```
package edu.mum;
```

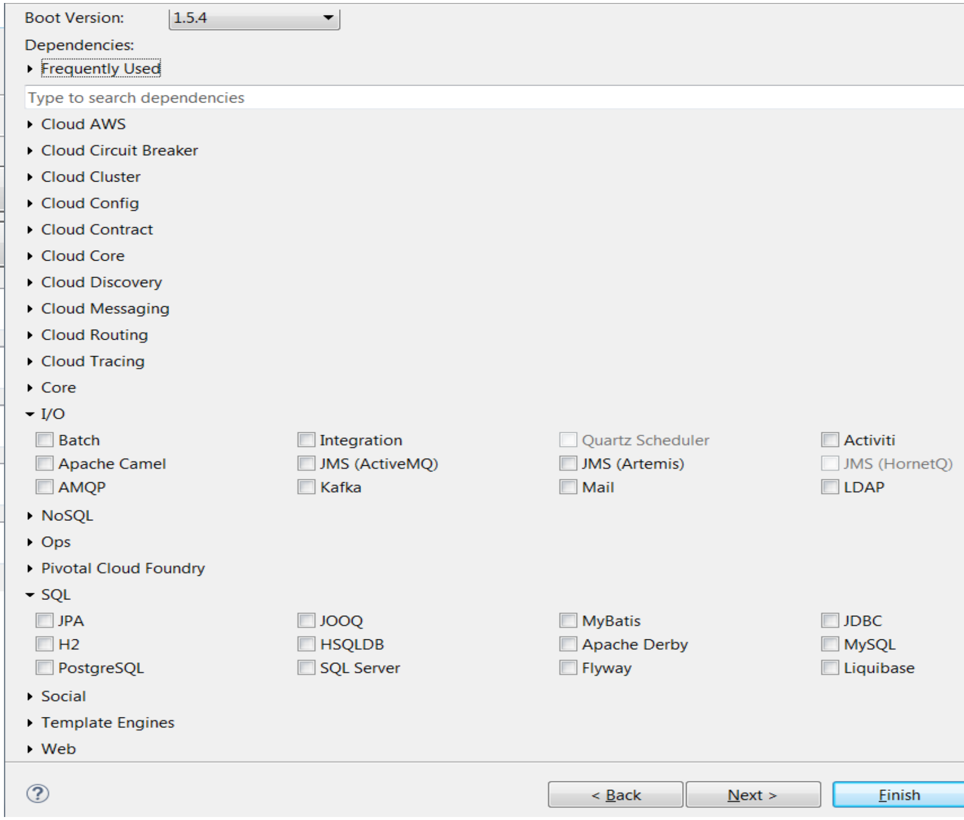
```
:                (v1.5.4.RELEASE)

04.825  INFO 8480 --- [           main] ed
04.828  INFO 8480 --- [           main] ed
04.907  INFO 8480 --- [           main] s.
05.493  INFO 8480 --- [           main] o.
05.510  INFO 8480 --- [           main] ed

05.513  INFO 8480 --- [       Thread-2] s.
05.515  INFO 8480 --- [       Thread-2] o.
```

New Spring Boot Project



New Spring Starter Project

Name	HelloWorldBoo		
[✓] Use default location			
Location	C:\Users\admin1\EA\workspace\HelloWorldBoo		
Type:	Maven	Packaging:	Jar
Java Version:	1.8	Language:	Java
Group	edu.mum		
Artifact	HelloWorldBoot		
Version	0.0.1-SNAPSHOT		
Description	Hellow World project for Spring Boot		
Package	edu.mum		



Boot Version: 1.5.4

Dependencies:

- Frequently Used

Type to search dependencies

- Cloud AWS
- Cloud Circuit Breaker
- Cloud Cluster
- Cloud Config
- Cloud Contract
- Cloud Core
- Cloud Discovery
- Cloud Messaging
- Cloud Routing
- Cloud Tracing
- Core
- I/O

☐ Batch	☐ Integration	☐ Quartz Scheduler	☐ Activiti
☐ Apache Camel	☐ JMS (ActiveMQ)	☐ JMS (Artemis)	☐ JMS (HornetQ)
☐ AMQP	☐ Kafka	☐ Mail	☐ LDAP

- NoSQL
- Ops
- Pivotal Cloud Foundry
- SQL

☐ JPA	☐ JOOQ	☐ MyBatis	☐ JDBC
☐ H2	☐ HSQLDB	☐ Apache Derby	☐ MySQL
☐ PostgreSQL	☐ SQL Server	☐ Flyway	☐ Liquibase

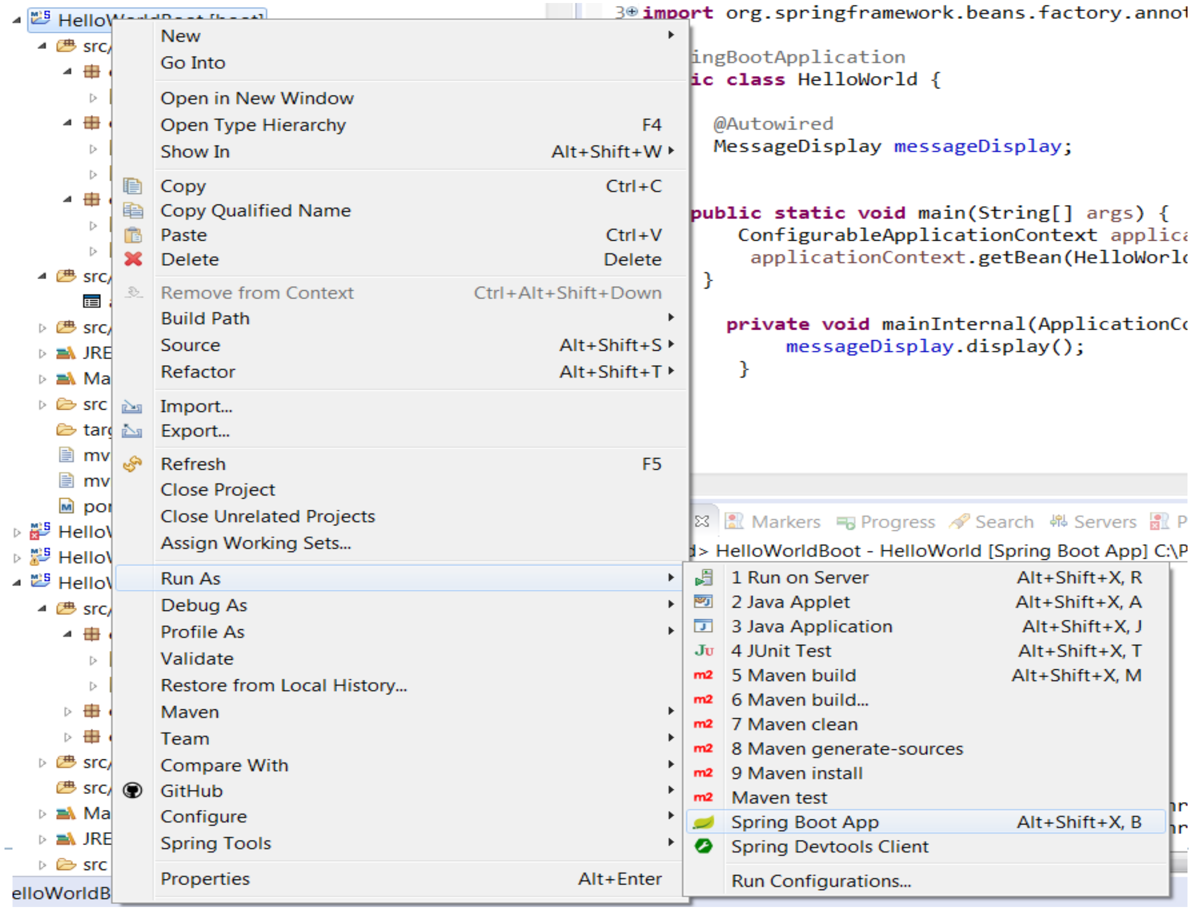
- Social
- Template Engines
- Web

Spring Boot Example

Looks Similar to Java Config

```
@SpringBootApplication
public class HelloworldbootApplication {

    public static void main(String[] args) {
        ConfigurableApplicationContext applicationContext =
SpringApplication.run(HelloworldbootApplication.class, args);
        applicationContext.getBean(MessageDisplay.class).display();
    }

}
```

Run As Boot Application



```
import org.springframework.beans.factory.annot

ingBootApplication
ic class HelloWorld {

    @Autowired
    MessageDisplay messageDisplay;


public static void main(String[] args) {
    ConfigurableApplicationContext applica
     applicationContext.getBean(HelloWorl
}

    private void mainInternal(ApplicationC
        messageDisplay.display();
    }
```

```
HelloWorldBoot - HelloWorld [Spring Boot App] C:\P
```

Dependency Injection Annotations

Annotation	Package	Source
@Resource	javax.annotation	JSR 250
@Inject	javax.inject	JSR 330
@Autowired	org.springframework.bean.factory	Spring
@Qualifier	org.springframework.bean.factory	Spring

Name a component example –

```
@Autowired
@Qualifier(“production")
private MemberService MemberService;
```

```
@Component(“production”)
public class MemberServiceImpl implements MemberService
```

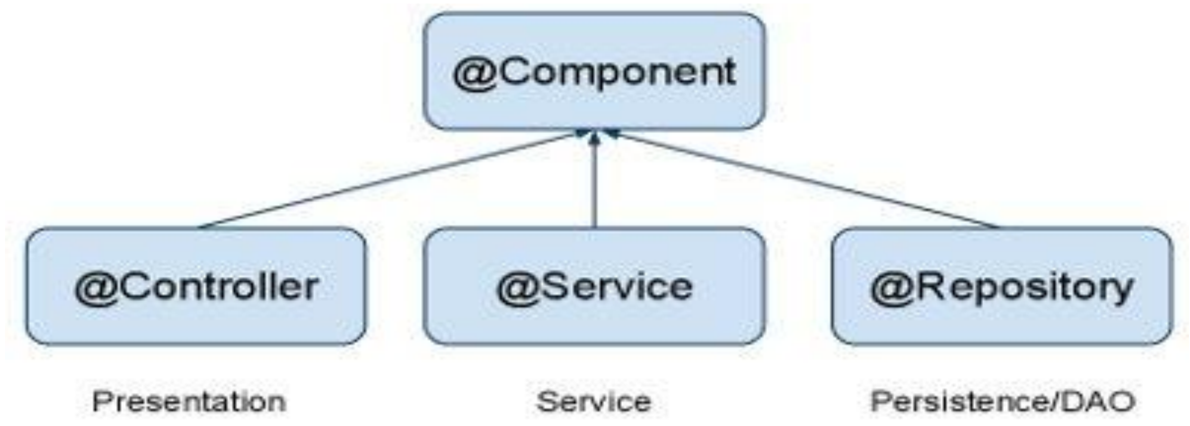
Named Component Comparison Examples

```
@Autowired
@Qualifier("production")
private MemberService memberService;

@Inject
@Qualifier("production")
private MemberService memberService;

@Resource(name="production")
private MemberService memberService;
```

Referenced Component:

```
@Component(“production")
 public class MemberServiceImpl implements MemberService
```

Spring Component Annotations



@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.

Dependency Injection [DI] placement

- DI exists in three major variants
- Dependencies defined through:
  - Property-based dependency injection.
  - Setter-based dependency injection.
  - Constructor-based dependency injection

- Container *injects* dependencies when it creates the bean.

Dependency Injection examples

- **Property based[byType]:**

```
@Autowired
ProductService productService;
```

- **Setter based[byName]:**

```
ProductService productService;
@Autowired
public void setProductService(ProductService productService){
            this.productService = productService;
```

- **Constructor based:**

```
 ProductService productService;
 @Autowired
public ProductController(ProductService productService) {
                 this.productService = productService;
```

When do we use DI?

- **MAINLY** when referencing components **BETWEEN** layers
- When an object references another object whose implementation might change
  - You want to plug-in another implementation
- When an object references a plumbing object
  - An object that sends an email
  - A DAO object
- When an object references a resource
  - For example a database connection

Main Point

Dependency Injection allows us to support better separation of concerns and creates more malleable applications with minimal effort.

Science of Consciousness: *The experience of transcending gives us a better understanding of the order and flexibility in life in an effortless way.*