Dependency Injection

Heather Mortensen
SEIS 770, 01
11/20/17

Table of Contents

What is dependency injection? ….....page 1

In which situations is DI of value?.....page 2

Spring Framework ….....page 2

Three Forms of DI .....page 5

Constructor injection with Spring.....page 6

Setter Injection with Spring ….....page 6

Interface Injection.....page 7

Service Locator.....page 7

Broad Challenges & Benefits of DI.....page 7

Figures.....page 9

References.....page 18

What is dependency injection?

This paper explores relevant terminology, moving from broad to increasingly specific terms, related to the development of Fowler's concept of Dependency Injection and its different forms. Definitions are given for Dependency Inversion, Inversion of Control, Dependency Injection, and Service Locator in an effort to uncover how the dependency injection pattern might assist in the development of good architecture. Bob Martin defines 'good' architecture as one that defers decision making regarding what type of database, delivery mechanisms, frameworks, data models, graphical user interfaces, or potential interactions with the web, until as late as possible. (Martin, 2015, Minute 25) Fowler writes that his favorite architectural pattern, layers, is one of the most widely used in enterprise applications. (Fowler 2003, p2)

Robert Martin first published on the Dependency Inversion Principal (DI) in 1994, according to Fowler. Schuchert summarizes the principal as moving, "...towards higher-level abstractions," in his article *DIP in the Wild*. Abstractions are described as concepts that are closer to the domain, or client. (Schuchert, 2013) Robert Martin's original description of the Dependency Inversion Principal, from his book, Agile Software Development, is as follows,

> **"A. High-level modules should not depend on low-level modules. Both should depend on abstractions.**
> **B. Abstractions should not depend on details. Details should depend on abstractions."**

Dependency Inversion is part of a larger paradigm, described as, "...the last "D" of the five SOLID Principals of object oriented programming." (Thompson) Those principals are shown in figure 1.

The Dependency Injection Pattern (IP) is described by Fowler as a specific type of Inversion of Control (IoC). (Fowler p5) It is meant to reduce coupling. Fowler first identified use of the term in 1988, in a paper by Johnson and Foote, titled *Designing Reusable Classes*. (Fowler, 2005) It was there that the term IoC was coined, also known as The Hollywood Principal. IoC was first explained through its relationship with software frameworks,

> **"*One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code...This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.*"** (Fowler, 2005)

In this way, frameworks help us plug concrete software components into abstract components. Control is inverted from our main class into the Framework, upholding the Dependency Inversion Principal. This has been described as a practical alternative to soldering a lamp directly into an outlet (figure 2, Thompson). We can see in figure 3, a UML that shows likeness to many common software patterns, including Observer, in that it increases the level of abstraction.

It is helpful to contrast these concepts. This is done, most briefly, as follows, "DI is about wiring, IoC is about direction, and DIP is about shape." (Schuchert, 2013)

> **"The difference between IoC and dependency inversion is as follows. Inversion of control is used to increase modularity of the program and make it extensible. The term is related to, but different from, the dependency inversion principle, which concerns itself with decoupling dependencies between high-level and low-level layers through shared abstractions. The general concept is also related to event-driven programming in that it is often implemented using IoC..."** (Inversion of control.2017b)

"Dependency Injection is about how one object knows about another, dependent object." (Schuchert, 2013) The most general definition of Dependency Injection, given by Fowler, is that we have a, "...separate object, an assembler, that populates a field in the lister class with an appropriate implementation for the finder interface." (Fowler, p5) Fowler claims, however, that, "Inversion of Control is too generic a term...," since Service Locators and Dependency Injection are two different means to the same end - removing dependencies from applications. (Fowler p5) He believes that less confusion will be generated if we differentiate further, between different forms of Dependency Injection. The three distinct forms that he identifies are Constructor Injection, Setter Injection, and Interface Injection. (Fowler p5)

In which situations is DI of value?

Spring Framework

Spring is a common Java EE framework that is used as a tool for constructing IoC architecture. (figures 4 and 5) It is open source and manages all of your application dependencies. (Moley,) As shown in figure 15, IoC containers manage dependencies by housing them all in a central location. They can be passed as needed, at run time, as opposed to being spread erratically throughout our classes. (Moley,) Dependency Injection and IoC are considered synonyms within the Spring Framework. (Cosmina, et al, p8) "DI frameworks are often driven by XML files that help specify what to pass to whom and when." This helps, "...manage the dependencies and the interactions between objects." (Binstock, 2008) Frameworks are not the only useful examples of IoC, although, "IoC is a common characteristic of frameworks." (Fowler, p4) Fowler includes a menacing, abstract warning, on page 14 of his article, that, while inversion of control is a, "common feature of frameworks...its something that comes at a price." IoC can be also achieved through programming to an interface, through a Factory Pattern, or through a Service Locator Pattern (Havenith, 2016, Stack OverFlow 2015)

"The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans." (Rod Johnson, et al, 2016) Kerschbaumer gives a good summary of how beans and IoC containers work together,

> **"The IoC container manages java objects – from instantiation to destruction – through its BeanFactory. Java components that are instantiated by the IoC container are called beans, and the IoC container manages a bean's scope, lifecycle events, and any AOP features for which it has been configured and coded."** (Kerschbaumer & Winterfeldt, 2009)

Spring uses JavaBeans and interfaces to, "...gain the flexibility of defining dependency configuration within...applications in different ways (for example, XML files, Java configuration classes, annotations within your code, or the new Groovy bean definition method)." (Cosmina, Harrop, Schaefer, & Ho, 2017,

p8) DI decreases the amount of repetitive code required to wire together components of large applications. (Cosmina, et al, p10) While use of the 'new' operator is simple, the resulting code required to perform the wiring together of pieces that make up the whole, is cumbersome because of the, "...need to look up dependencies in JNDI repository or when the dependencies cannot be invoked directly, as is the case with remote resources." (Cosmina, et al, p10) Configuration is simplified by allowing for the interchanging of different dependency implementations. DI may, for example, simplify the swapping out of different database types after some minor reconfiguration of business object dependencies. (Cosmina, et al, p10) Fowler writes that Spring, "...includes abstraction layers for transactions, persistence frameworks, web application development, and JDBC." (Fowler p7) The Spring IoC container is accessed through an interface, org.springframework.context.ApplicationContext. (Rod Johnson, et al, 2016) The life cycle of a bean and relevant methods are shown in figure 8. Fowler writes that there are mixed opinions regarding the value of beans and that their use may tie you forever to a dependence on the EJB server, which may become difficult to remove. He suggests a suitable alternative as the construction of a Transaction Script on top of a Row or Table Gateway. (Fowler 2003, pp100, 101) However, Spring will be investigated here, as an example that illustrates two of the three different types of dependency injection.

Susan Kerschbaumer shows the simplest example that I have seen for using the Spring container on her website *Spring by Example*. Any java object can be a bean. Step 1 of Figure 9 shows java code for the creation of a Bean. The bean uses an id called *message* in order to register with its container. This is done via the XML code shown in step 2 of Figure 9. (Kerschbaumer & Winterfeldt, 2009) Now, instead of instantiating an object using 'new DefaultMessage(),' the container will instantiate the message bean. Fowler mentions that the configuration can also be achieved through java code, but describes XML as, "...the expected way to do it." (Fowler, p7)

Three Forms of DI

According to Wikipedia, Fowler's three identified forms of DI, differ in that,

> **"Setter and constructor injection differ mainly by when they can be used. Interface injection differs in that the dependency is given a chance to control its own injection. All require that separate construction code (the injector) take responsibility for introducing a client and its dependencies to each other."** (Dependency injection.2017)

We might want to use constructor injection when we are able to build all of our dependencies first. This is said to ensure that the client object's state is always valid and prevents null references. It helps make our client immutable, or changeless, over time. This predictability makes it thread safe. However, we will not be able to change the dependencies down the road. (Dependency injection.2017) In general, Fowler recommends using constructor injection, unless you are forced into using another form by problems that arise. (Fowler p18)

We might, instead, prefer Setter Injection when we prefer additional flexibility regarding the state of the dependency references. Complications may arise if we attempt to use the client before it is properly wired. Not every injection may have occurred at the time that we attempt to use the client. This

increases the risk of a null dependency. We must run verification that all required injections have occurred before using the client. (Dependency injection.2017)

"Any object that may be used can be considered a service. Any object that uses other objects can be considered a client." (Dependency injection.2017) Interface injection allows complete decoupling of the dependencies and the clients. The injection takes place via a method and is provided by an interface. We require an assembler. "The assembler would take a reference to the client, cast it to the setter interface that sets that dependency, and pass it to that dependency object which would turn around and pass a reference-to-self back to the client." (Dependency injection.2017) Wikipedia continues its explanation,

> **"For interface injection to have value, the dependency must do something in addition to simply passing back a reference to itself. This could be acting as a factory or sub-assembler to resolve other dependencies, thus abstracting some details from the main assembler. It could be reference-counting so that the dependency knows how many clients are using it. If the dependency maintains a collection of clients, it could later inject them all with a different instance of itself."** (Dependency injection.2017)

Again, in general, Fowler recommends using constructor injection, unless you are forced into using another form. (Fowler p18) Fowler describes the choice between constructor and setter injection as one that is often asked, more generally, in OO programming. This question is if you should, "...fill fields in a constructor or with setters." Fowler's answer is that we should build objects, to the highest degree possible, within the constructor. (Fowler, p15) He describes, however, a complication appearing from having, "multiple ways to construct a valid object." In this case, he says that a Factory Method might be useful. However, a problem arises when we use factory methods to for assembling components because because factory methods are typically static. Interfaces don't allow for static methods. (Fowler, p15) He highlights the case that multiple constructors, using inheritance, may lead to a prohibitively large, "explosion of constructors," which will force us away from constructor injection.

Constructor injection with Spring

The tutorial site baeldung.com defines 'Constructor Injection' as the moment where, "...required components are passed into a class at the time of instantiation." Kerschbaumer's example shows the code for this in figure 10. The addition of an explicit constructor into the DefaultMessage class and a constructor-arg element into the XML code creates a better alternative to using the 'new' operator. Fowler states that Spring allows for constructor injection and setter injection, but that it is tailored more for setter injection. (Fowler, p7)

Setter Injection with Spring

XML configures by collecting property names that are inferred from the class's set methods. For this reason, it is important to name according to the whats advised in the JavaBeans specifications. These can be found at http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html. (Kerschbaumer & Winterfeldt, 2009) The JavaBeans architecture is described in the specs and includes a summary of the interface, class, exceptions, and an annotation. Beans can be read or written by using the XMLDecoder and XMLEncoder classes. In general, Spring will use the lowercase name that follows "set" in the name of the setter method. For the example that I have included, setMessage() is the name of the setter within the SetterMessage class (figure 11). Therefore, the property name implied

would be message. This allows us to eliminate the need for configuration files. It uses an injected constructor and property values. These values are static. If we prefer to inject values by reference, this is also possible by injecting one beans definition into another. This is done using the constructor-arg or property's ref attributes instead of the value attribute in order to reference another bean's id. This is shown in figure 11. (Kerschbaumer & Winterfeldt, 2009)

Interface Injection

The Spring Framework does not support interface injection. Fowler uses a Framework called Avalon to illustrate an example of its use. Examples of interface injection were difficult to find, so I must rely on Fowler's example exclusively, although Apache Avalon has been retired. (The apache avalon project.2004) Wikipedia summarizes interface injection as one that, "...provides an injector method that will inject the dependency into any client passed to it. Clients must implement an interface that exposes a setter method that accepts the dependency." (Dependency injection.2017)

Service Locator

Fowler differentiates between a service locator and dependency injection. "The basic idea behind a service locator is to have an object that knows how to get hold of all the services that an application might need." (Fowler, p10) Fowler advises us to think of a service as a 'registry.' (p11) Oracle documentation describes a registry as, "...a remote interface to a simple remote object registry that provides methods for storing and retrieving remote object references bound with arbitrary string names...The lookup and list methods are used to query the current name bindings." (Interface registry.1993) Online sources seem divided on the matter that a service locator may be an antipattern and/or violate the Dependency Inversion Principal. The main problem with using a service locator, according to Fowler, is the fact that the application may have multiple authors and thus control of the service locator becomes, "...out of control of the writer." (page 15)

Broad Challenges & Benefits of DI

Benefits of Dependency Injection include a reduction in noise, coupling, and problems arising from misconstruction. It may also allow us to, "focus more attention on the API contract." (Moley,) Wiki writes that DI helps us: 1.) Create an application that is independent of object creation; 2.) Configure objects in a separate, isolated file; and 3.) Create an application that supports different configurations. (Dependency injection.2017) Spring supports Junit testing, a JDMC template, and a Hibernate Configuration. (1. introduction to spring framework.) DI helps us reduce dependencies, which are a high risk point for change over time. It make the code more reusable in that classes must merely be reconfigured, as opposed to rewritten, in order to serve in another context. As dependencies are move to a higher level, in interfaces, they are easier to pick out, creating more readable code. (Jenkov, 2014)

One challenge, still clearly exists in agreeing upon, and communicating, the definitions of related terms and concepts. Fowler, himself, writes that, "The topic of wiring elements together drags me almost immediately into the knotty terminology problems that surround the terms service and components." (p2) Various authors and programmers define additional forms of Dependency Injection, beyond those defined by Fowler.

A blog for Applied Information Sciences, writes, “Despite the terms Dependency Inversion Principle (DIP), Inversion of Control (IoC), Dependency Injection (DI) and Service Locator existing for many years now, developers are often still confused about their meaning and use.” (Blumenauer, 2013) I have, as an example of this, noticed misunderstandings about whether generic typing represents a form of Dependency Injection. Himanshu Manjarawala mistakenly includes it (figure 13), highlighting the importance of hashing out a subtle extension of Fowler's definition that may not be immediately clear – that we hope to delay concrete implementations of dependencies past compile time into run time. A Stack Overflow post from 2012, clarifies that generics do not allow component choice to be made at run time, but merely at compile time. Thus, they are not an example of dependency injection. Microsoft supports this position as well. IoC can help us remedy the fact that, “The concrete implementations of the dependencies have to be available at compile time.” (Inversion of control.2017a) This previous statement highlights an additional challenge of IoC, which is that late binding that may contribute to run time errors, as opposed to compile time errors. A challenge mentioned in JavaWorld is the difficulty in, “...writing an entire application using it.” (Binstock, 2008) “For entire applications, you frequently want a framework to manage the dependencies and the interactions between objects.” (Binstock, 2008) Wikipedia writes that, “Ironically, dependency injection can encourage dependence on a dependency injection framework." (Dependency injection.2017)

Figures

Initial	Concept
S	**Single responsibility principle**[4] a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class)
O	**Open/closed principle**[5] "software entities … should be open for extension, but closed for modification."
L	**Liskov substitution principle**[6] "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.
I	**Interface segregation principle**[7] "many client-specific interfaces are better than one general-purpose interface."[8]
D	**Dependency inversion principle**[9] one should "depend upon abstractions, [not] concretions."[8]

Figure 1. SOLID (object-oriented design). (2017).

Figure 2. DIP Principal (Thompson)

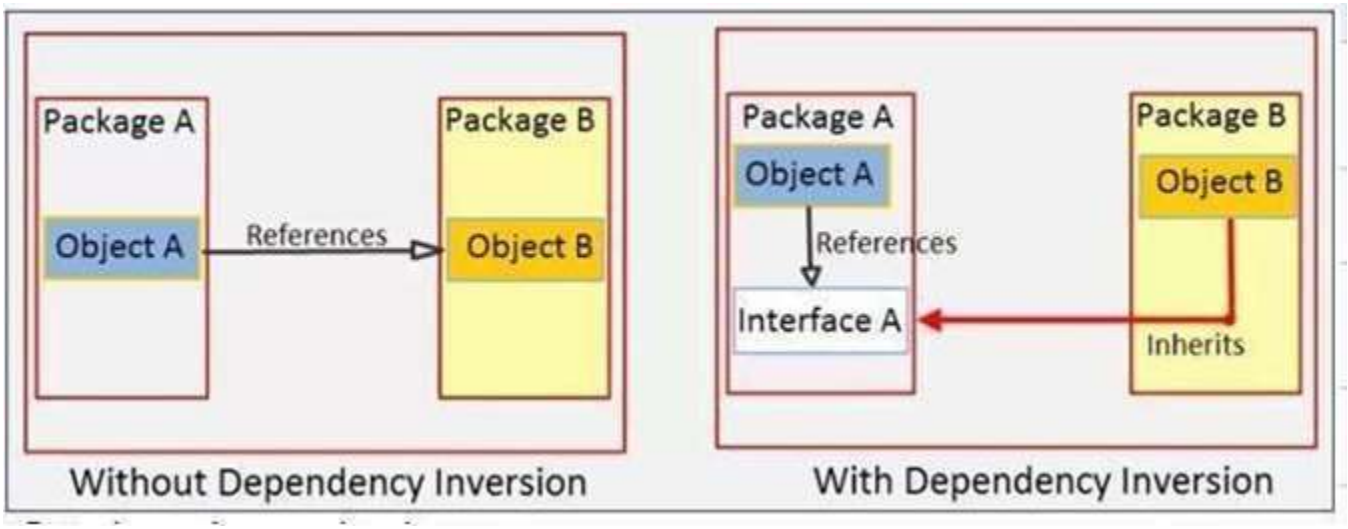


Figure 3. UML Without and With DIP (Thompson)

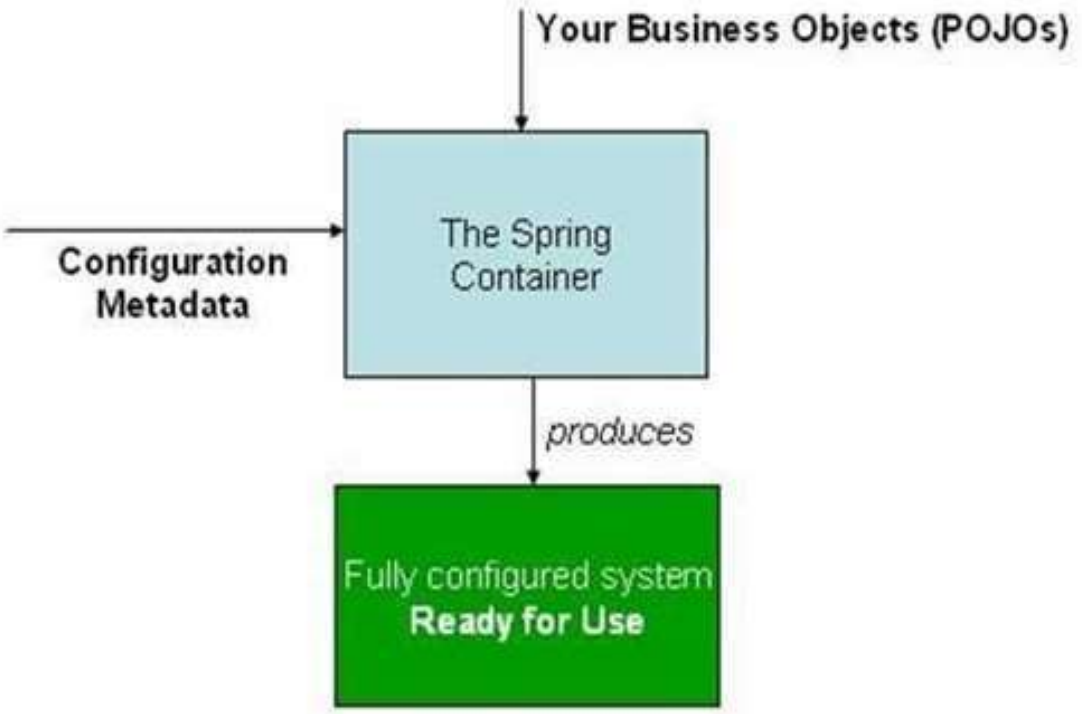


Figure 4. High-level view of how Spring works
(5.1 introduction to the spring IoC container and beans.)

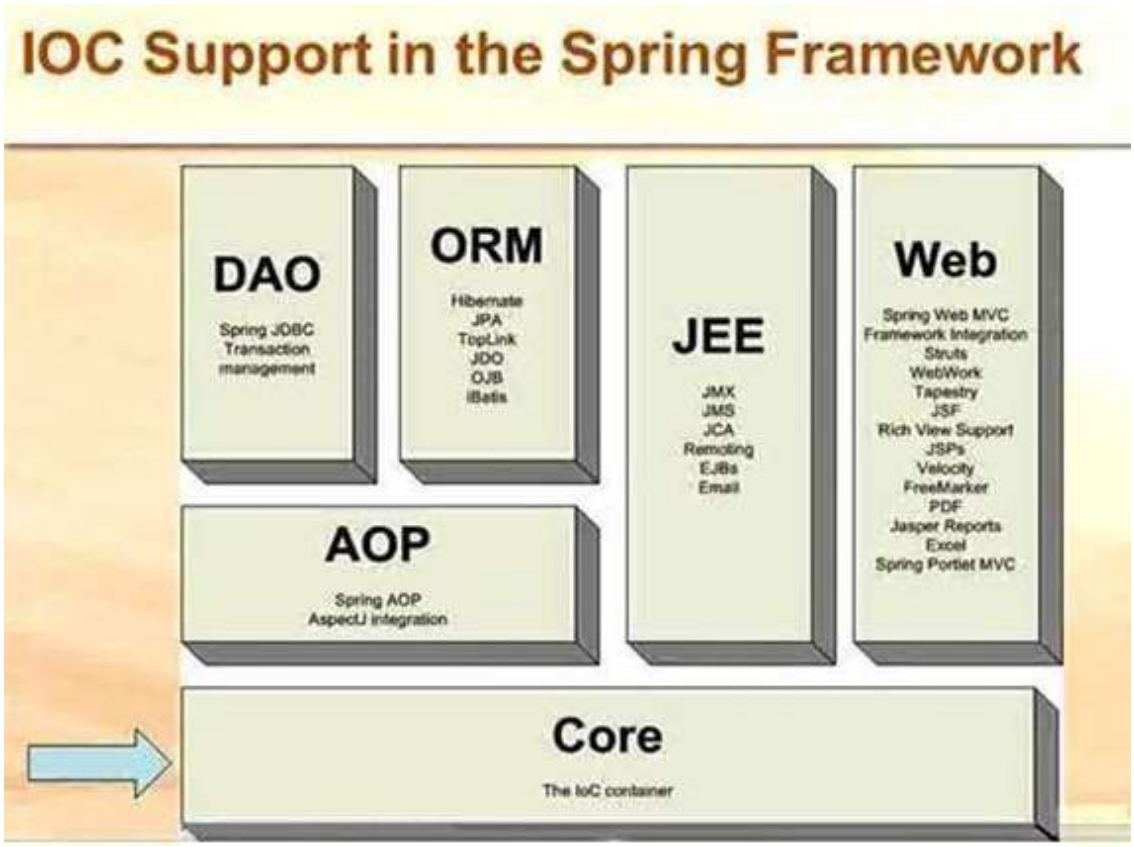


Figure 5. Low-level view of how Spring works (*InversionOfControlPresentationVideo-GLB.*)

Constructor Injection Example

```
// Constructor
Client(Service service, Service otherService) {
    if (service == null) {
        throw new InvalidParameterException("service must not be null");
    }
    if (otherService == null) {
        throw new InvalidParameterException("otherService must not be null");
    }

    // Save the service references inside this client
    this.service = service;
    this.otherService = otherService;
}
```

Figure 6. Constructor Injection Example (Dependency injection.2017)

Setter Injection Example

```
// Set the service to be used by this client
public void setService(Service service) {
    if (service == null) {
        throw new InvalidParameterException("service must not be null");
    }
    this.service = service;
}

// Set the other service to be used by this client
public void setOtherService(Service otherService) {
    if (otherService == null) {
        throw new InvalidParameterException("otherService must not be null");
    }
    this.otherService = otherService;
}
```

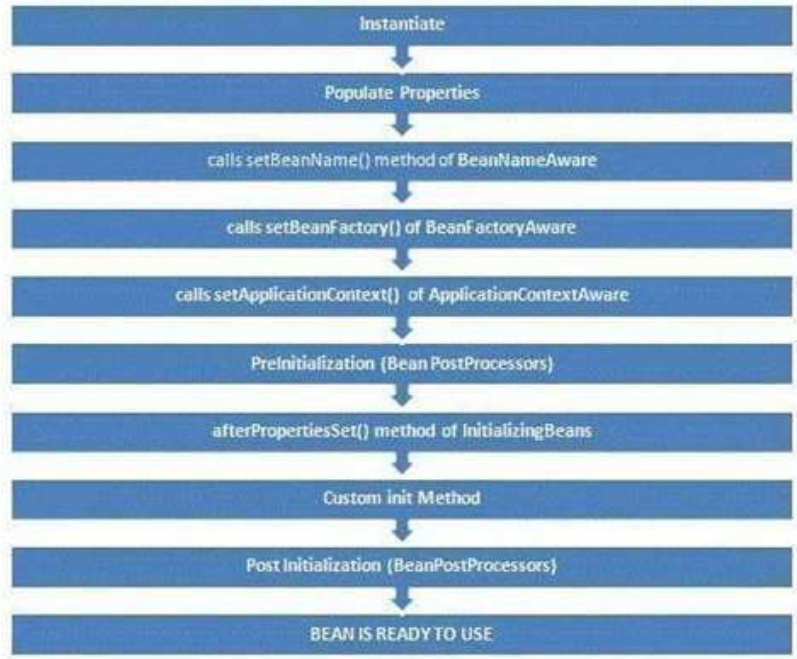
```
// Set the service to be used by this client
public void setService(Service service) {
    this.service = service;
}

// Set the other service to be used by this client
public void setOtherService(Service otherService) {
    this.otherService = otherService;
}

// Check the service references of this client
private void validateState() {
    if (service == null) {
        throw new IllegalStateException("service must not be null");
    }
    if (otherService == null) {
        throw new IllegalStateException("otherService must not be null");
    }
}

// Method that uses the service references
public void doSomething() {
    validateState();
    service.doYourThing();
    otherService.doYourThing();
}
```

Figure 7. Setter Injection Example (Dependency injection.2017)



ollowing diagram shows the method calling at the time of destruction.

Figure 8. Life Cycle of a Bean (07 - spring bean life cycle.)

Step 1. Create a Bean class called DefaultMessage

```
public class DefaultMessage {

    private String message = "Spring is fun.";

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

Step 2. XML Configuration for the bean called message

```
<?xml version="1.0" encoding="UTF-8"?>
<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="message"
          class="org.springbyexample.di.xml.DefaultMessage" />

</beans>
```

Figure 9. Creation of a Spring Bean (Kerschbaumer & Winterfeldt, 2009)

Constructor Injection

```
public class ConstructorMessage {

    private String message = null;

    /**
     * Constructor
     */
    public ConstructorMessage(String message) {
        this.message = message;
    }

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

The constructor-arg element injects a message into the bean using the constructor-arg element's value attribute

```
<?xml version="1.0" encoding="UTF-8"?>
<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="message"
          class="org.springbyexample.di.xml.ConstructorMessage">
        <constructor-arg value="Spring is fun." />
    </bean>

</beans>
```

Figure 10. Creation of a Spring Bean using **Constructor Injection** (Kerschbaumer & Winterfeldt, 2009)

Setter Injection Example Code - Pass by value

```
public class SetterMessage {

    private String message = null;

    /**
     * Gets message.
     */
    public String getMessage() {
        return message;
    }

    /**
     * Sets message.
     */
    public void setMessage(String message) {
        this.message = message;
    }

}
```

SetterMessageTest-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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="message"
          class="org.springbyexample.di.xml.SetterMessage">
        <property name="message" value="Spring is fun." />
    </bean>

</beans>
```

Figure 11. Creation of a Spring Bean using **Setter Injection – Pass by value**
(Kerschbaumer & Winterfeldt, 2009)

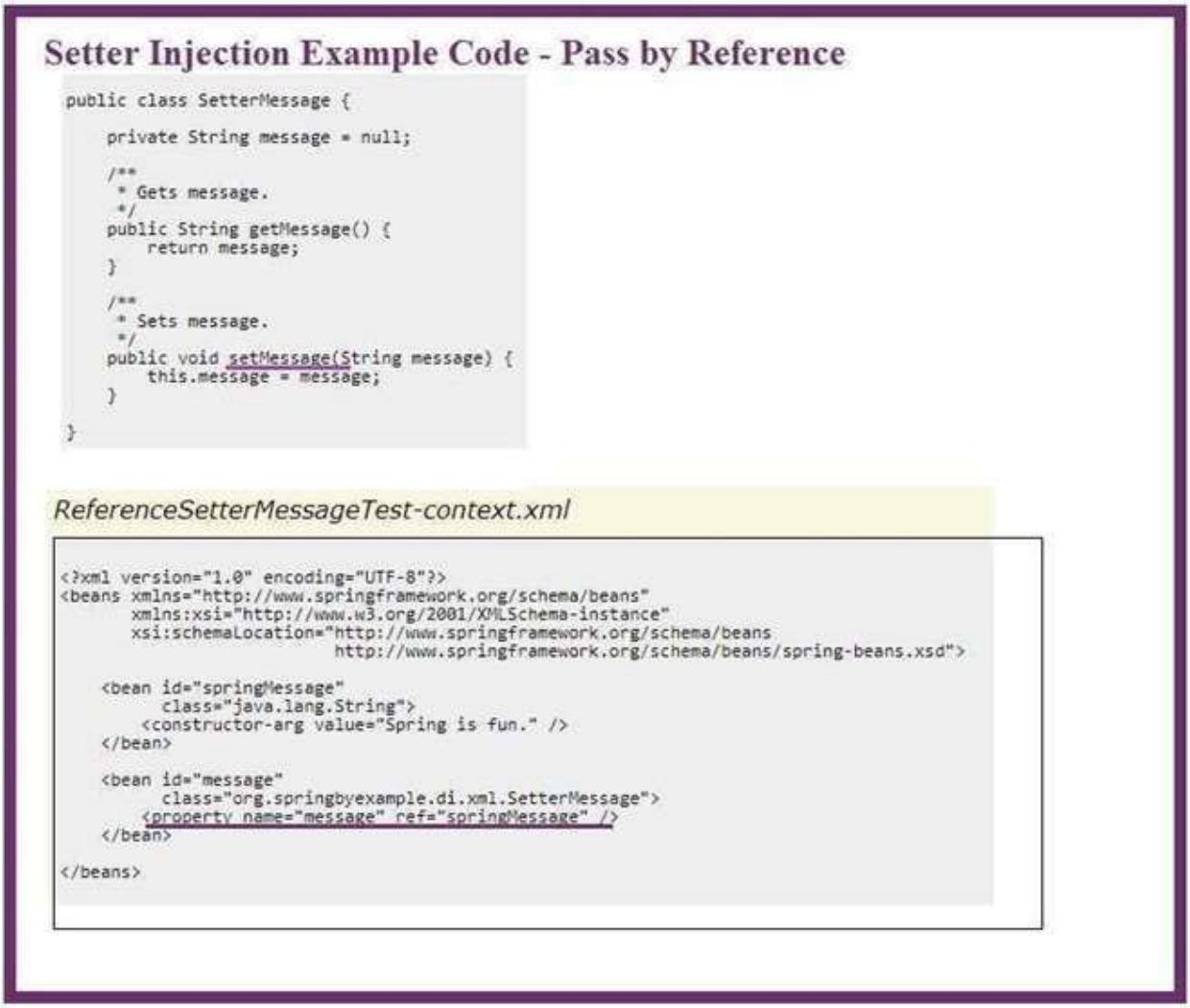


Figure 12. Creation of a Spring Bean using **Setter Injection – Pass by Reference**
(Kerschbaumer & Winterfeldt, 2009)

This is the basic concept for implementing the Inversion of Control (IoC) pattern.

- It eliminates tight coupling between objects.
- Makes objects and application more flexible.
- It facilitates creating more loosely coupled objects and their dependencies.

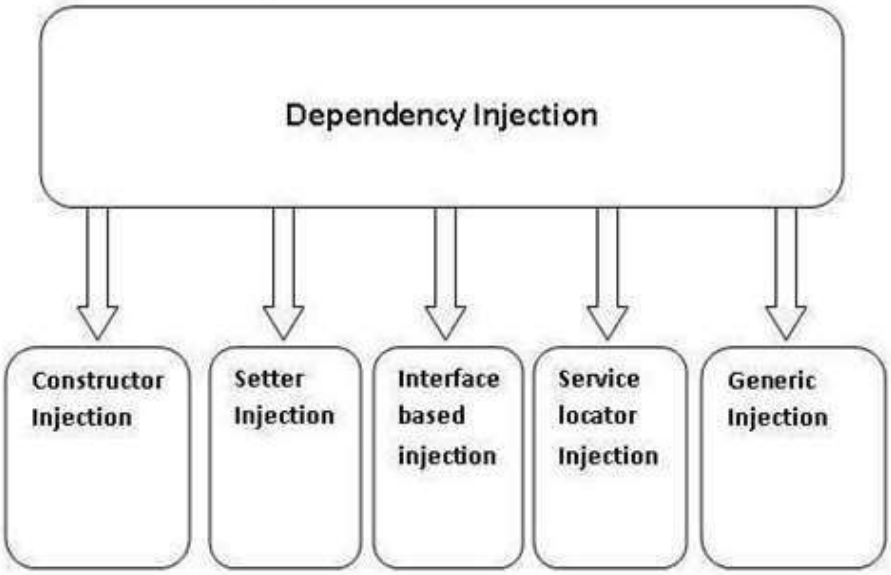


Figure 13. Misinterpretation of term Dependency Injection (Manjarawala, 2012)

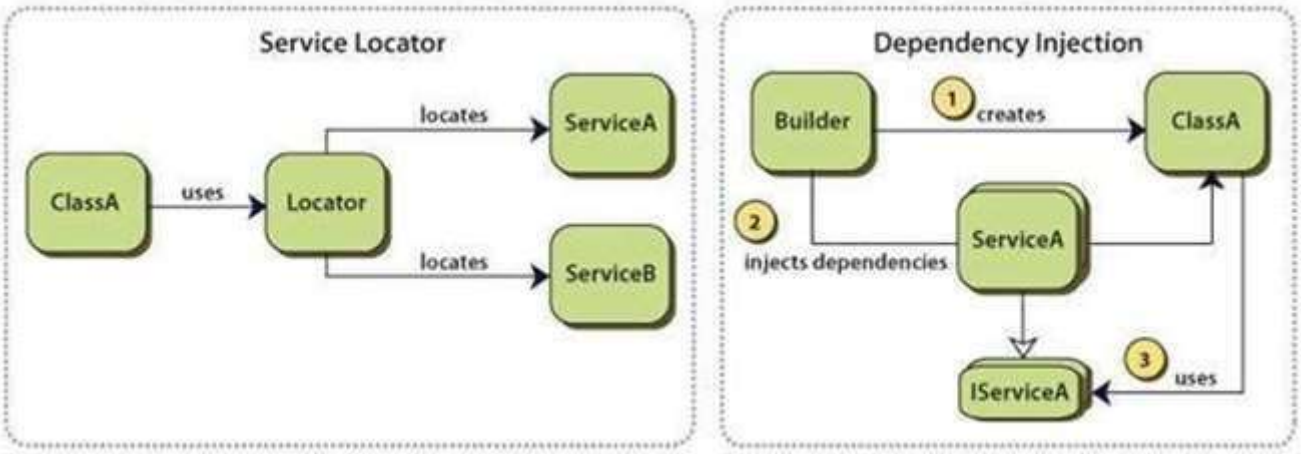


Figure 14. Service Locator VS DI (Inversion of control.2017)

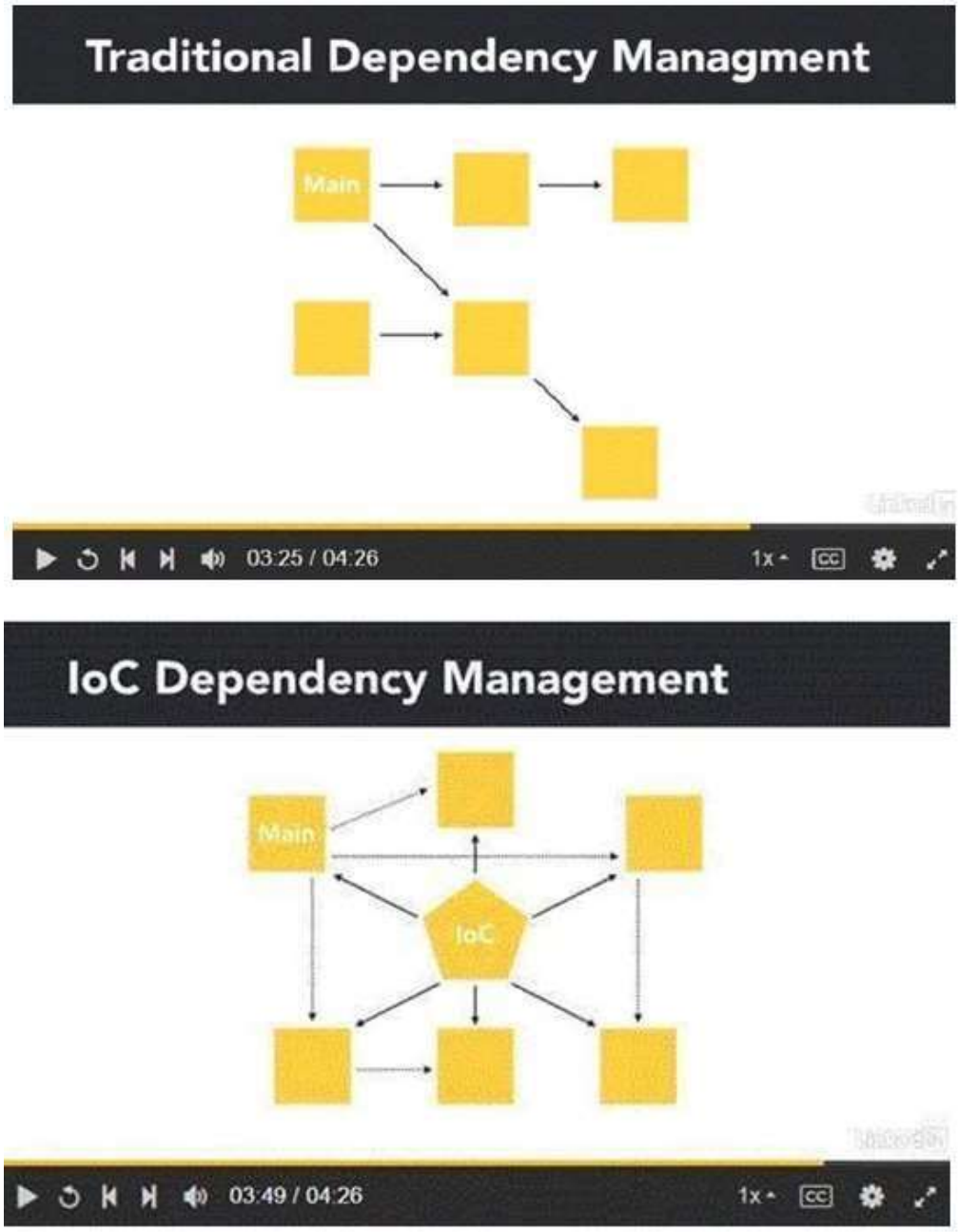


Figure 15. How IoC Container Manages Dependencies, Compared with Traditional Dependency Management (Moley, min 3:49)

References

07 - spring bean life cycle. Retrieved from http://www.wideskills.com/spring/spring-bean-lifecycle

1. introduction to spring framework. Retrieved from https://docs.spring.io/spring/docs/3.2.x/springframeworkreference/html/overview.html

The apache avalon project. (2004). Retrieved from https://avalon.apache.org/closed.html

Binstock, A. (2008). Excellent explanation of dependency injection (inversion of control). Retrieved from https://www.javaworld.com/article/2071914/excellent-explanation-of-dependency-injectioninversionofcontrol-.html

Blumenauer, J. (2013). Lost in translation: Dependency inversion principle, inversion of control, dependency injection & service locator. Retrieved from https://blog.appliedis.com/2013/12/10/lost-intranslation-dependencyinversionprincipleinversion-of-control-dependency-injection-service-locator/

Cosmina, I., Harrop, R., Schaefer, C., & Ho, C. (2017). *Pro spring 5 : An in-depth guide to the spring framework and its tools* (5th ed.). Berkeley, CA: Apress. doi:10.1007/978-1-4842-2808-1

Dependency injection. (2017). Retrieved from https://en.wikipedia.org/wiki/Dependency_injection

FOWLER M. (2003). *Patterns of enterprise application architecture*. BOSTON: MA : ADDISON WESLEY. Retrieved from https://www.epo.org/index.html

Fowler, M. (2005a). Inversion of control. Retrieved from https://martinfowler.com/bliki/InversionOfControl.html

Fowler, M. (2005b). InversionOfControl. Retrieved from https://martinfowler.com/bliki/InversionOfControl.html

Generic vs dependency injection   (2012). Retrieved from https://stackoverflow.com/questions/11715317/generic-vs-dependency-injection

Havenith, S. (2016). Dependency inversion vs dependency injection vs inversion of control vs programming to interfaces. Retrieved from https://www.equinox.co.nz/blog/dependencyinversiondependency-injection-inversion-controlprogramming-interfaces

Interface registry. (1993). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/rmi/registry/Registry.html

Inversion of control. (2017a). Retrieved from https://msdn.microsoft.com/en-us/library/dd458907.aspx

Inversion of control. (2017b). Retrieved from https://en.wikipedia.org/wiki/Inversion_of_control

Inversion of control vs dependency injection   (2016). Retrieved from https://stackoverflow.com/questions/6550700/inversion-of-control-vs-dependency-injection

Berosik, G. (Producer), & . (). *InversionOfControlPresentationVideo-GLB.* [Video/DVD]

Ioc vs dependency injection (di) vs. service locator (sl). Retrieved from https://gridwizard.wordpress.com/2014/05/28/dependency-injection-vs-service-locator/

Jenkov, J. (2014). Dependency injection benefits. Retrieved from http://tutorials.jenkov.com/dependency-injection/dependency-injection-benefits.html Kerschbaumer, S., & Winterfeldt, D. (2009). A practical introduction to inversion of control. Retrieved from https://www.springbyexample.org/examples/intro-to-ioc.html Manjarawala, H. (2012). Inversion of control: Overview with examples. Retrieved from https://www.codeproject.com/Articles/380748/Inversion-of-Control-Overview-with-Examples

The S.O.L.I.D. principles of OO and agile design. Martin, B. (Director). (2015, Feb 4,).[Video/DVD] Arlington Heights, IL:

Moley, F.Introduction to spring. Retrieved from https://www.lynda.com/SpringFrameworktutorials/Introduction-Spring/606088/669334-4.html

Package java.beans. (2015). Retrieved from https://docs.oracle.com/javase/6/docs/api/java/beans/packagesummary.html

Rao, L. (2015). Resource injection vs. dependency injection explained! Retrieved from https://dzone.com/articles/resource-injection-vs

Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, Andy Clement, Dave Syer, Oliver Gierke, Rossen Stoyanchev, Phillip Webb. (2016). Introduction to the spring IoC container and beans. Retrieved from https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html

Schuchert, B. L. (2013). DIP in the wild. Retrieved from https://martinfowler.com/articles/dipInTheWild.html

The service locator pattern. Retrieved from https://msdn.microsoft.com/en-us/library/ff648968.aspx

SOLID (object-oriented design). (2017). Retrieved from https://en.wikipedia.org/wiki/SOLID_(objectoriented_design

Thompson, J.Dependency inversion principle. Retrieved from https://springframework.guru/principlesofobject-oriented-design/dependency-inversion-principle/