MphasiS
an HP company

Think Beyond. Think MphasiS.

WHITE PAPER

Dependency Injection - A Critical Look

Mohan Borwankar

August, 2010

Table of Contents

1 Summary 2

2 Overview of DI 2

3 Critical Evaluation of DI 2

4 Conclusion 3

References 4

Acknowledgements 4

1 Summary

Martin Fowler penned the expression “Dependency Injection” (DI) in his article “Inversion of Control Containers and the Dependency Injection Pattern[2]”. The concept spread like wildfire and the Spring framework extensively used and advocated the use of DI for server side Java architecture and in the process the Spring framework became synonymous with DI.

The developer community, which was overwhelmed by the heavyweight containers and EJB, quickly adopted lightweight Spring and DI. Dependency Injection, like the use of plastics, soon became a necessity and was overused as well as abused. Over a period the side effects of the excessive and ineffective use of DI have started to surface - as said by Charles Handy: *“When Efficiency is ineffective; when too much of a good thing is bad”* [1].

This paper will have a critical look at dependency injection by touching upon the basic concept of DI. The web has a wealth of information on DI and this paper certainly doesn’t plan to copy paste the same information from the web. DI is oversold in most of the papers on the web and they carefully avoid mentioning the shortcomings of DI. This paper will focus the other side of DI and come up with guidelines for using or not using DI.

2 Overview of DI

Answers to the following basic questions will give a quick overview of DI. What is DI? Where do you use DI? How do you use DI? And why do you use DI?

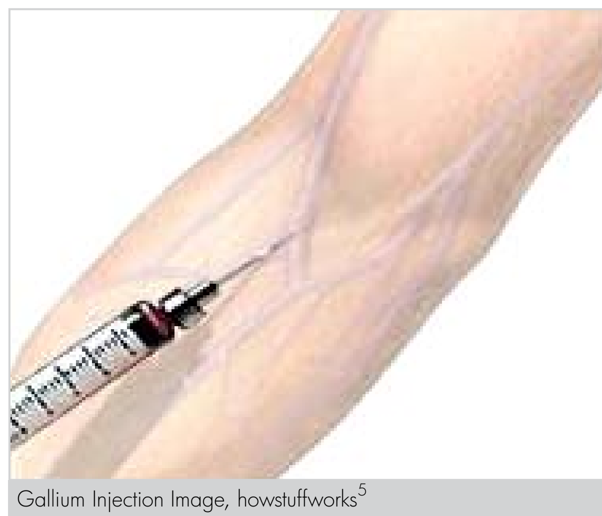
A neatly designed and implemented Object Oriented (OO) application will have a set of well defined objects. Each of these objects has a well defined but narrow scope and responsibility. In order to accomplish the objective of the end application, one or more objects need to be invoked in a well defined manner. Traditional code will create, invoke and dispose the objects. However in DI style of coding, the objects are created, maintained and injected into the main code by the container runtime. The main code is only aware of what the injected objects will accomplish and not aware of how the objects are implemented. Thus, in short, DI is a technique in which the dependent objects are externalized and injected into the main code runtime[3].

Why do you use DI or what are the benefits of using DI? The most obvious is the reduction in the dependency on other components. A code is heavily dependent on the components it uses and the components are susceptible to changes. As long as the objects stick to the well defined interface, the objects can be upgraded or seamlessly replaced. The components, too can be better reused across the code. DI also helps in unit testing as mock implementations can be easily injected. DI also easily manages the nested dependencies of the injected object. Thus, in short, DI helps build truly dynamic, flexible applications.

Where should DI be used? DI is highly recommended in the following situations.[4]

- to inject configuration data into one or more components.
- to inject the same dependency into multiple components.
- to inject different implementations of the same dependency.
- to inject the same implementation in different configurations.
- to effectively use some of the services provided by the container.

3 Critical evaluation of DI



Gallium Injection Image, howstuffworks[5]

The picture above may look a little out of place and scope and is sure to add a lot of confusion. How is DI similar to the medical injection analogy? The syringe is a standardized interface and the liquid that is injected changes from time to time and patient to patient. It shouldn’t be too difficult to stretch this to DI. The interface is fixed and the implementation is injected at runtime. The implementation can be upgraded seamlessly. The critical difference however is that the patients blindly trust the liquid that is injected. Will or should the programmers blindly trust the piece of code that is injected? The healthcare industry has stringent quality checks. The software quality checks are no match for the healthcare quality checks - sad but true.
The biggest concern is about injecting the third party components into your code. The injected code meets the interface requirement however there is no idea about how the interface is actually implemented. In most of the cases, this is a welcome

1 The Hungry Spirit, Charles Handy, 1999. ISBN 978-0767901888

2 Inversion of Control Containers and the Dependency Injection pattern, Martin Fowler, 2004, on the web at http://martinfowler.com/articles/injection.html

3 What is Dependency Injection? Jakob Jenkov, no date. On the web at http://tutorials.jenkov.com/dependency-injection/index.html

4 What is Dependency Injection? Jakob Jenkov, no date. On the web at http://tutorials.jenkov.com/dependency-injection/index.html

5 Gallium Injection Image, on the web at http://healthguide.howstuffworks.com/gallium-injection-picture.htm

feature. The injected code may not meet the stringent security requirements and simply log or print sensitive data onto console or to a file – simply printing passwords or credit card numbers. The exception handling in the injected code may not follow the standards. If the injected code throws runtime exception, it will be impossible to detect that during the development phase. As a result, it will throw a few nasty surprises at runtime. When the third part components are injected into the code, a robust exception handling framework has to exist around the injected code. The injected code also has to pass the security audit. As mentioned earlier, in DI the dependent objects are created, maintained and injected into the code. Though this is actively sold as an advantage of DI, it leaves the main code no control over the lifecycle of the dependent object. The code is at the mercy of the container to manage the lifecycle of the dependent object. Most of the containers optimize the object creation and have singleton instances created of the dependent object. This helps in stateless implementations but if the injected object holds state or has reference of another object which is stateful, it will create havoc in a multithreaded environment. If the singleton object has a locking implementation, it will impact the performance. If singleton objects are not used, a large number of objects will be created and most of the containers don't have an 'object pool' implemented.

The containers, especially the Spring container, have simplified the process of DI to a great extent and most of the DI is fully configuration driven. This is an extremely welcome feature for the developers. It makes the process of DI invisible to the tools like profilers, code analyzers and compilers. By definition, DI is a runtime process and hence none of the tools will be able to analyze the injected code. All the errors will surface at runtime only. This also impacts the overall efficiency of the development process. New developers often find the DI code difficult to read and trace.

If dependent objects are injected and never replaced with another implementation during the lifecycle of the project, then DI is a complete misfit. If the injected objects are not changed at runtime based on runtime parameters, then DI is not really needed. The project will be better off using a single pattern of the implementation. DI is also used to configure objects. If a different configuration is never needed in the code, DI is overkill. DI allows code to be extremely flexible and have loose coupling between the main code and injected objects. Unless the code really requires this flexibility and runtime loose coupling, DI should not be considered. Most of the projects tend to overlook what they actually need and whether DI is really needed. Buzzwords like DI, Spring, AOP are often used to impress the client and give a false sense of achievement.

Reusability and flexibility are two important keywords in the context of DI. Unless the focus is on building reusable code or a framework which needs runtime flexibility, dependency injection should not be considered. This suggestion may sound little provocative however a deeper analysis of DI will lead to the fact that DI provides a great deal of flexibility, isolation of interface from implementation and reusability of base code. Hence, consider DI when reusability, flexibility and isolation is really needed.

4 Conclusion

If the only tool you have is a hammer, you tend to see every problem as a nail.[6] *Why inject when the oral dose is as effective?*

DI is a wonderful technology and is here to stay[7]. It is a natural choice when frameworks and large scale systems which are based on reusable components are built. However, armed with the knowledge of DI, designers and developers often tend to over-inject.

Is DI really the right choice of technology for the project? A great deal of thought must go behind answering this question. Does the project really need reusability, runtime flexibility and configurability provided by DI? Unless it is really needed don't use DI or else the 'efficient technology will become ineffective'. Special care has to be taken about the implementation of the injected components. They should not throw runtime exceptions, violate security guidelines for logging or have stateful implementations.

The people who work on the DI design and implementation need to have a deeper understanding of the DI. Adequate training and mentorship has to be ensured before the project starts.

In short, DI is an extremely efficient technology if used properly. Do proper homework on the need to use dependency injection and use it effectively.

6 The law of the instrument, also known as Maslow's hammer, after Abraham Maslow, 1966. On the web at http://en.wikipedia.org/wiki-/Law_of_the_instrument

7 The Rich Engineering Heritage Behind Dependency Injection, Andrew McVeigh, no date. On the web at http://www.javalobby.org/arti-cles/di-heritage/

References:

- http://martinfowler.com/articles/injection.html
- http://en.wikipedia.org/wiki/Dependency_injection
- http://tutorials.jenkov.com/dependency-injection/index.html
- http://www.javalobby.org/articles/di-heritage/
- http://www.springsource.org/

Acknowledgements:

The author wishes to thank Jie Fang for his critical review and comments.

Contact us

USA
MphasiS
460 Park Avenue South
Suite # 1101, New York
NY 10016, U.S.A.
Tel: +1 212 686 6655
Fax: +1 212 686 2422

UK
MphasiS
88 Wood Street
London EC2V 7RS, UK
Tel: +44 208 528 1000
Fax: +44 208 528 1001

AUSTRALIA
MphasiS
410 Concord Road
Rhodes, NSW 2138, Australia
Tel: +61 290 221 146
Fax: +61 290 221 134

INDIA
MphasiS
Bagmane Technology Park
Byrasandra
C.V. Raman Nagar
Bangalore 560 093, India
Tel: +91 80 4042 6000
Fax: +91 80 2534 6760

About MphasiS

MphasiS is a global service provider with $1B in revenues, delivering technology based solutions to clients across the world. With currently over 39,000 people, MphasiS services over 200+ clients in Banking and Capital Markets, Insurance, Manufacturing, Communications, Media & Entertainment, Healthcare & Life Sciences, Transportation & Logistics, Retail & Consumer Packaged Goods, Energy & Utilities, and Governments around the world. Our competency lies in our ability to offer integrated service offerings in Applications, Infrastructure Services & Business Process Outsourcing capabilities. We are uniquely positioned to service our clients with best cost-performance. To know more about MphasiS, log on to www.mphasis.com

MphasiS and the MphasiS logo are registered trademarks of MphasiS Corporation. All other brand or product names are trademarks or registered marks of their respective owners. MphasiS is an equal opportunity employer and values the diversity of its people. Copyright © MphasiS Corporation. All rights reserved.

1210

MPHASIS
an HP company