Investigating Dependency Injection and its Frameworks

Sbonelo Mdluli : 1101772

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

***Abstract*—This paper investigates dependency injection which is one of two design pattern which could be used to implement the dependency inversion principle. When correctly implemented dependency injection improves the maintainability of a program by making the code more testable, modifiable and extensible. Dependency injection(DI) is generally implemented through constructor ,setter and interface injection. A Java program is used to demonstrate dependency injection with the aid of a dependency injection framework. A number of frameworks are analysed and Google Guice is chosen to demonstrate the design pattern. The experience in this project developing dependency injection using Guice led to a couple of findings. Dependency injection frameworks can significantly reduce boilerplate code. Whilst this is a great feature most of literature regarding dependency injection is remains inaccessible to those who are inexperienced with pattern.**

I. INTRODUCTION

When writing large object oriented programs achieving maintainable code can be a problem. Dependency injection one of two design patterns that can be used to implement the dependency inversion principle which forms part of the object-oriented SOLID design principles [1]. The dependency inversion principle states that high level modules should not depend on low level modules but upon abstraction and that abstraction should not be based on implementation [2] [3] [4]. The ultimate goal of dependency injection is to increase decoupling between classes. This is because tightly coupled classes are hard to maintain and break encapsulation [1] [3] [5]. The idea behind dependency injection is to separate object creation from usage. This is accomplished by shifting the responsibility of resolving dependencies to a dedicated injector rather than the objects themselves.

II. BACKGROUND

The term and pattern were first popularised by Martin Fowler in his 2004 article where he contrasts the service locator and dependency injection pattern approach to the dependency inversion principle [6]. Compared to service locator dependency injection does not make an explicit call to the locator class [6] [7]. Martin argues that service locator approach is more appropriate than that DI however DI is preferable when designing classes with multiple applications. Dependency injection comes with some benefits compared to direct instantiation such as testability, maintainability and extensibility.

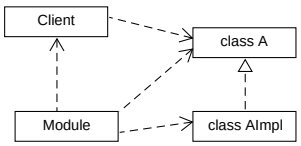

Fig. 1: Dependency Injection Diagram

Figure 1 shows how the dependency graph would be when one is using dependency injection. The module is created in the client code and both the implementation of class A and its implementation are needed by the module for binding. This diagram is adopted from the one provided by Martin Fowler in his article.

A. Advantages of Dependency Injection

Extensibility allows for modification or different implementations of services without having to alter the individual components.
Dependency injection makes unit testing easier as one can provide mock implementations of service classes.
It is also believed that dependency injection improves maintainability of software which is a key component of the software life cycle [8] [9] [10] [5]. Increasing testability facilitates the use of TDD in the software development life cyle .Ekaterina Razina Intuit in [11] conducted an experiment in which they investigated the effects of DI on software maintainability. The experiment was conducted on 20 open source projects however with inconclusive results. With DI you get to eliminate the use of static methods which simplifies writing tests.

B. Disadvantages of Dependency Injection

DI introduces syntathic clutter which can decrease code readability and even maintainability. Configuring the code manually can be an involved task. DI becomes

an anti-pattern when incorrectly used or overused. The other major disadvantage of DI is that is that dependency errors occur at run-time which is more challenging to resolve [4].

C. Dependency Injection Qualifiers

H. Y. Yang *et al* in [3] provide qualifiers for code to be a good candidate for use of dependency injection .Concrete classes do not qualify for DI and any object declared as final objects also rule out the use of DI. Dependency injection should be used for service classes and not client classes. Dependency injection should be used if one is intending to provide a different implementations to a service class.

III. Types of Dependency Injections

A. Constructor injection

The dependencies are passed to the class through the constructor. Constructor injection is more suitable for objects need to be constructed once or immutable ones, however constructor injection may result in messy code when a lot of parameters are passed in. Constructor injection may suffer from the constructor pyramid problem ,this occurs when the same objects need to be injected differently depending on the application [6] [10].

B. Setter injection

A setter method is the client code used to inject the dependencies into the class. Setter injection reduces the risk of objects being intertwined. Setter injection is more flexible compared to constructor injection because it is easy get the intent of a method. However one should be cautious to hind setter method as not abuse them.

C. Interface injection

In interface injection the client code is required to implement an interface that will expose a setter method to accept the dependencies. Interface injection is similar to setter injection with the exception that each setter is written in a separate interface.
In many many situation interface injection is not preferable due to the fact that one is required to provide multiple interfaces which could be a cumbersome task and can result in code maintainability issues.

IV. Dependency Injection Frameworks

DI frameworks address DI shortcomings mainly by centralizing the object configuration to dedicated injectors. As a result DI frameworks reduce time spent in wiring up the objects. Injectors significantly reduce syntathic clutter by removing the need for factories. Different frameworks provide different configuration methods namely through a XML file or via annotations. Another advantage of DI frameworks is automated object lifetime management.

Static Analysis most frameworks resolve dependencies as needed this limits the static analysis a framework offers .Composability most frameworks are not well suited to handle an interface with multiple implementations or configurations[6] [12] [13]. DI frameworks are implemented with reflection which can decrease the effectiveness of IDE automation [4]. Popular frameworks that implement DI include Spring (Java) ,Google Guice (Java) ,Dagger (Java and Android) and Unity(.NET).

All DI frameworks are required to have a dedicated mechanism for managing the registration/binding of dependencies to instances. They must also provide automatic lifetime management of dependent objects. The framework must automatically and dynamically construct the dependent objects. The framework must be able to do late binding of dependencies.

A. Google Guice

Guice is a Java based and XML free framework for dependency injection. Guice provides binding annotations to allow the developer easily refer to an injection. Bindings provide modules to which the injector can be configured. Guice extends an *AbstractModule* to facilitate configuration. The binding phase in Guice uses a key-value pair to map an interface to a particular implementation. Guice also allows one to inject static members [10] [14].

B. Spring

Perhaps the most popular DI framework in this list is Spring. The framework was developed in 2003 by Rob Johnson. It is one of the most popular open source application development frameworks for Java. It offers abstraction, modules and is easy to integrate with third party libraries. Spring consists of containers which is where objects are wired

up and their lifetimes are managed. The beanFactory in Java provides an interface capable of configuring objects. The *org.springframework.beans* and *org.springframework.context* packages are the basic packages needed with the spring to implement DI. Spring supports both annotation and XML configuration of metadata [10] [15].

C. Dagger

Dagger is a Java and Android compile-time dependency injection framework. The latest release of dagger improves code structure, provides a more precise and consistence API. Dagger is also easy to debug as its dependency graph is validated during compile-time rather than run-time which uses reflection. Dagger is intended to replace factory classes used to implement DI without having to write boilerplate code. Dagger also supports static inspection of code path in order to track object calls [16].

D. Unity

Unity is a lightweight dependency injection framework developed and maintained by Microsoft for .NET application. The container allows one to define a mapping between an interface and concrete class or method.
An important but not essential feature of DI frameworks is the support Aspect Oriented Programming (AOP). All the aforementioned frameworks support AOP with Dagger being the exception. Aspect Oriented Programming is a software design pattern which solves a number of problems in OOP. The main problem AOP solves is crosscutting concerns such security as that appears in the business logic. AOP solves this problem by separating such functionality to independent modules. AOP support is implemented by most of the frameworks but one can implement DI without AOP. AOP is therefore considered as a secondary feature to the framework. Scopes are also considered as secondary features of a DI framework. Scopes simplify the code testing procedure in scopes allow one to manage an objects lifetime [17].

V. Demonstrating Dependency Injection

Guice was chosen as the framework to investigate in detail and implement DI with because it primarily uses java for its dependency configurations. Guice provides type-safe and rigorous configuration implementation and is smaller which makes it faster to learn. Secondary benefits that come with the framework include preloaded JavaDoc a Java documentation standard and JUnit a Java testing framework. Frameworks like Spring on the other hand are full stack application frameworks which doesn't suite the requirements of the demonstration program.

A. Problem Definition

This exercise develops a program to assist restaurant personnel in processing customer orders. It automates the task of taking and processing orders. This gives the restaurant personnel more time to complete tacks that need human intervention. This program will make it easy for restaurants to manage customer information. This information can be used to bring about business insight in the form of customer trends.

A customer can register an account linked to their banking details, they can login to place an order and this order will automatically process their bill.

The chef will be one only one who is granted permission to prepare the menu. The customers will get notifications after registration and after purchasing a meal.

B. Implementing Dependency Injection with Guice

1) Setter Injection: A customer creates an account by requesting a registration form where they fill in their credentials and banking details. The customer can log in using their account details. Upon logging in a chef object is required to create a menu for the customer.The chef is injected using setter injection this is because the application should allow the menu object to be modifiable during runtime. Such an instance is when chef updated the menu by removing certain menu options which are no longer available because they run out of ingredients. The customer places an order by specifying an item and quantity of the item they want on the menu. A processOrder object is required to process the order. The processOrder objects completes an order given that the customer has a sufficient bank balance. An exception is thrown to deny the order should the customer have insufficient funds. Setter injection is again used to provide more flexibility to allow for multiple processOrder objects to be accessed as needed.

2) Constructor Injection: The chef accepts a menu through constructor injection. The chef constructor adds all the meals that will be offered to all the customer. Based on this reason constructor injection is more appropriate than the other types for this task. A notifier class is implemented that accepts a customer through constructor injection. Constructor injection is chosen because the notifier class only sends out notification confirming customer details. The use of constructor injection when creating a notification message was justified by the fact that the constructor accepts a single arguments and does not attempt to manipulate customer information it only prints it out.

3) Providers: When a customer creates an account they are added to a text file which models a database(textfile) with a capacity of 10 users. When the initial database has reached its capacity new customers are redirected to a new database. To decrease the response time when the application is running the size of the database is determined at the beginning of the program. A provider is used to establish which database a customer belongs to. In the context of the program the database id is the object id of the customerRepository object. The customerRepository class is the one that implements and controls how a database is created. Provider<customerDatabase>database is used to define a setter injection method which returns the database id. The number of customers currently added in the database is implemented in provider get() method.

4) Instance Bindings with Custom Binding: Instance binding is used to create simple objects as the google guice documentation suggests. Instance binding is used in conjunction with @Named annotation. This annotation allows one to define their own custom binding annotation using a string as an identifier. However one should be caution not overuse this annotation as it likely to result in type errors. Instance binding is used to bind an integer class to a constant. The first use is used to create the default food quantity, which assigns a food item to each order and this is then overridden by the customers placeOrder method when it takes a variable quantity from provided by the customer. This annotation is also used when specifying the capacity of the customer database.

5) Untargeted Bindings: Untargeted binding allows one to create bindings without having to configure or resolve the dependencies in a module. The annotation @ImplementedBy or @ProvidedBy can be used specify a particular concrete class implementation. An untargeted binding was used to make the injector aware that the customerImpl class implements the customer class interface. The untargeted binding is used to specify a default implementation of the customer class. This binding is deemed appropriate for the this class because it contains the main logic for the program and should be default unless another implementation is provided.

6) AOP using Interception: Aspect Oriented Programming is used in the program to send a confirmation message after the customer has registered an account. Method interception is used to resolve cross cutting concerns. The cross cutting concern in the application is generating a confirmation message for a customer which can be interpreted as a form of logging. A custom binding annotation is created that will be used to invoke a specific method. The binding annotation can be created as below.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface notifications{}
```

The annotation can then be applied to the method that is to be intercepted, which in the program is the registrationForm() method found in the customerImpl class. A method matcher is used to match classes and methods in the MethodInterceptor which is used to trace an interceptor.

VI. Framework and Program Analysis

A. Implementation Analysis

Throughout the development of the program caution was taken not expose methods used for setter injection to client code in order to mitigate security concerns. The custom binding @Named was used sparingly as to limit the likely hood of type error occurring. Injecting indirect dependencies was avoided in all the the injection demonstrated in the code. The injector is created at a single location using Guice.createInjector() which is in the bootstrapping code in the main method of the program taking advantage of the centralised configuration in the module to resolve the required dependencies. Guice best practices were followed as best as possible during the development of the demonstration program. However one of constructors with dependency injection were not hidden in that they were made public. This design could raise several issues at a later stage which could make refactoring an involved task and could even lead to tight coupling.

B. Framework Analysis

Most of the frameworks assume some knowledge of dependency injection as a concept and don't cater for those with little or no background in the concept. Concepts are presented at a high level which could be daunting for someone who is not familiar with the terminology and technologies mentioned in the documentation. As a consequence of this high level approach the examples provided are often not detailed enough or are hard to follow since they build upon or modify a single a example.

The use of a DI framework significantly reduced the amount of boilerplate code that would have been needed whenever one instantiates an object. Getting rid of the boilerplate made the code more readable and therefore easily maintainable .Using mock object simplified unit testing the application.

VII. Conclusion

This report investigated dependency injection through the use of a framework. After investing several frameworks and their features Guice was chosen as the framework most appropriate for the demonstration of dependency injection .In addition several features of the framework were demonstrated through a program written to automate some of the restaurant duties. Dependency injection helped in improving the testability of the program however the experience gained from this projects showed that most frameworks either offered examples which were too simple to be of nay practical use or too difficult to follow. This approach hinders people inexperienced with the patterns to learn or grasp the concepts presented by the authors.

References

[1] T. Janssen, "Solid design principles explained: Dependency inversion principle with code examples." https://stackify.com/dependency-inversion-principle/. Accessed: 2019-03-10.

[2] R. C. Martin, "The Dependency InversionPrinciple", journal =,"

[3] H. Y. Yang, E. Tempero, and H. Melton, "An empirical study into use of dependency injection in java," in *19th Australian Conference on Software Engineering (aswec 2008)*, pp. 239–247, March 2008.

[4] B. Karia, "A quick intro to dependency injection: what it is, and when to use it." https://medium.freecodecamp.org/a-quick-intro-to-dependency-injection\-what-it-is-and-when-to-use-it-7578c84fa88f. Accessed: 2019-03-10.

[5] M. Seemann and S. van Deursen, *Dependency Injection Principles, practices, Patterns*. Manning Publications, 2019.

[6] M. Fowler, "Inversion of control containers and the dependency injection pattern." https://www.martinfowler.com/articles/injection.html#ConstructorVersusSetterInjection. Accessed: 2019-03-10.

[7] I. Vorobiov, "Dependency injection vs service locator." https://medium.com/@ivorobioff/dependency-injection-vs-service-locator-2bb8484c2e20. Accessed: 2019-03-10.

[8] K. Jeek, L. Hol, and P. Brada, "Dependency injection refined by extra-functional properties," in *2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 255–256, Sep. 2012.

[9] R. Vanbrabant, *Google Guice: Agile Lightweight Dependency Injection Framework*. APRESS, 2008.

[10] D. R. Prasanna, *Dependency Injection: Design Patterns Using Spring And Guice*. Manning Publications, 2009.

[11] E. R. Intuit, "Effects of dependency injection on maintainability," in *ICSE 2007*, 2007.

[12] N. Schwarz, M. Lungu, and O. Nierstrasz, "Seuss: Decoupling responsibilities from static methods for fine-grained configurability," *Journal of Object Technology*, vol. 11, pp. 1–23, 2012.

[13] M. D. Ekstrand and M. Ludwig, "Dependency injection with static analysis and context-aware policy," *Journal of Object Technology*, vol. 15, pp. 1:1–31, 2016.

[14] S. Berlin, "User's guide." https://github.com/google/guice/wiki/Motivation. Accessed: 2019-03-30.

[15] Spring, "The ioc container." https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html. Accessed: 2019-03-10.

[16] Google, "User's guide."

[17] F. S. Dominic Betts, Grigori Melnik and M. Subramanian, *Dependency Injection with Unity*. Microsoft, 2013.