International Journal of Scientific Research and Engineering Development-– Volume 4 Issue 3, May -June 2021

Available at www.ijsred.com

RESEARCH ARTICLE OPEN ACCESS

Overview of Dependency Injection Design Pattern

Shreesha Bhat, Dr. Vinay Hegde
(Computer Science and Engineering, RV College of Engineering, Bangalore
Email: shreeshabhat.cs17@rvce.edu.in)
(Computer Science and Engineering, RV College of Engineering, Bangalore
Email: vinayvhegde@rvce.edu.in)

--************************----------------------------------

Abstract:

During application development, it is desirable to provide loose coupling of their modules, so that program could be flexible, testable and maintainable. Within this paper, a design pattern is explained, which complies with the above requirements. Dependency Injection (DI), as one of the mechanisms for achieving the Inversion of Control (IoC) principle, represents a design pattern by which a particular class acquires dependency on external sources, rather than creating it itself.Also, the usage of DI within the google guice framework is presented, through the concrete examples of the implementation of a twitter client, presented in the code. With its usage, the displayed code becomes much more flexible and maintainable and eventually upgradable.

Keywords — **dependency injection, inversion of control, guice**

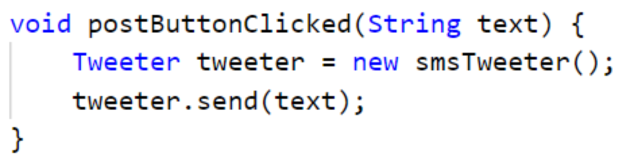
--************************----------------------------------

I. INTRODUCTION

One of the main pillars of Object-Oriented Programming is Modularity. Modularity refers to decomposition of the problem into multiple modules. The aim is to make the modules as loosely coupled as possible. The primary method of making a project highly modular is to reduce the dependencies between modules. Dependency is a broad software engineering term used to refer to a piece of software (module) that relies on another one.Dependency injection is one of the many design patters used to decrease coupling. By using dependency injection, the code becomes more reusable, modular and easily testable and less code will be written to achieve this.

II. MOTIVATION

The main motivation behind the concept of dependency injection is that the existing design patterns used to achieve loose coupling are substandard. This paper uses the example of implementation of a twitter module used to send the tweets out. An example for a click handler used to send out tweets is given below.



```
void postButtonClicked(String text) {
    Tweeter tweeter = new smsTweeter();
    tweeter.send(text);
}
```

In this example, it can be said that the postButtonClicked method depends on the tweeter object. The code in this example builds its dependency immediately by calling the constructor for the type of tweeter object that is needed, in this case a smsTweeter. A disadvantage of using constructors to create dependencies is that the code is not testable. Testing the postButtonClicked method would lead to an actual tweet being created.

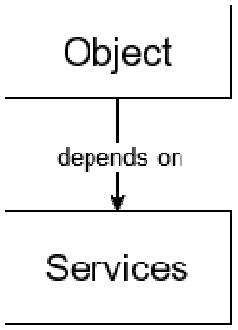


Figure 1: Direct dependency block diagram

The traditional solution to this problem has been the factory design pattern. A factory is a third-party object which separates a module from the services that it requires. An example on the same domain is given below.

```
void postButtonClicked(String text) {
    Tweeter tweeter = TweeterFactory.get();
    tweeter.send(text);
}
```

Here the tweeter we get is not necessarily an SmsTweeter. It is whatever the get method returns. This design allows us to return an alternate implementation of a tweeter that could be used for testing.

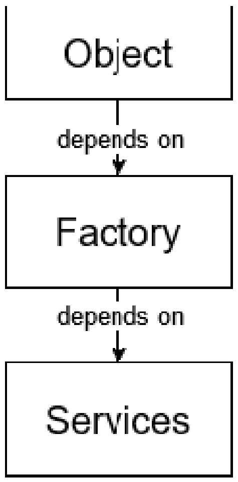


Figure 2: Factory pattern block diagram

By using the factory design pattern, the method is not directly dependent on the service (Tweeter), however the factory itself is directly dependent on it. The factory pattern introduces a layer of abstraction between the object and the services it requires instead of a hard direct dependency. Since there still is an indirect dependency, an application written using the factory pattern has a compilation dependency due to the monolithic dependency shown in figure 2. Therefore, a change to a module would require a rebuild of the entire application. Another drawback of the factory pattern is that the mock objects created for testing have to be cleaned up explicitly as the scope of the mock objects would belong to the factory. Moreover, the factory class must be written for every object, which means the number of files is effectively doubled for a project.

III. DEPENDENCY INJECTION

Dependency Injection is a design pattern that addresses the problem of tight coupling and poor testing techniques. It uses the Hollywood Principle, “Don’t call us, we’ll call you”. An example of a twitter client using dependency injection is given below

```
class TweetClient {
    private Tweeter tweeter;

    TweetClient(Tweeter tweeter) {
        this.tweeter = tweeter;
    }

    void postButtonClicked(String text) {
        tweeter.send(text);
    }
}
```

Here we pass the dependencies through the constructor so that the responsibility of choosing which implementation of the dependency is to be used is with the caller of the constructor. This is known as inversion of control [3].

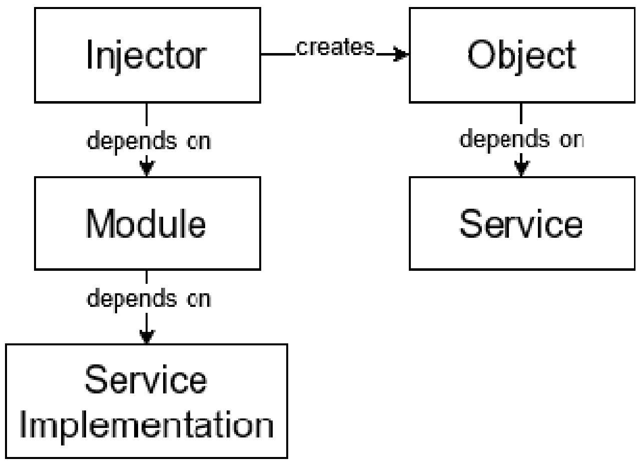


Figure 3: Dependency Injection block diagram

The guice dependency injection framework [1] consists of a Module and an Injector as shown in figure 3. A module is a class which binds the services needed by the object (dependencies) to their implementation. An injector is a factory which encompasses the entire application and doesn't need to be written or maintained.

```
import com.google.inject.AbstractModule;

class TweetModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(Tweeter.class).to(SmsTweeter.class);
    }
}
```

Above code shows the bind method which we use to bind the interface to the implementation. This replaces all the boilerplate factory code.The only change to the client code is the inject annotation above the constructor as shown in the code below. This is a signal to the guice framework which allows it to keep track of the constructor to call when it needs to create that object

```
@Inject
TweetClient(Tweeter tweeter) {
    this.tweeter = tweeter;
}
```

Therefore, the coupling between modules is stored entirely in the guice module and this makes the application loosely coupled. During compilation, when a service is needed, the bindings are checked to get the implementation of the service and that implementation is created by calling the constructor which is annotated in the client.

If there is a need to change the implementation of the service later on, then this can be done by just changing the binding in the module to the new implementation.

Another advantage of dependency injection is that the dependencies are part of the API. Therefore, there is no concern of breaking down the dependencies or forgetting to create a factory for every dependency.

IV. CONCLUSION

Creating an application can be a very complex process and it requires a compound approach. Applying DI design pattern [4] is an advantage that significantly contributes to it. In this paper, concrete mechanisms for using this pattern in two different frameworks are presented. Also, it can be noted that this pattern is applicable, whether it's a back-end or front-end part, even within multilayer applications. When DI is applied, the code of each application becomes less complicated for maintenance, testing and multiple use, and the components within it become loosely coupled.Moreover, it reduces the amount of code that has to be written by programmers to take care of all their dependencies. Therefore, it has a significant impact on the efficiency and quality of the applications themselves.

REFERENCES

[1] Marko Bojkić,Miroslav Stefanović and ĐorđePržulj"Usage of Dependency Injection within different frameworks" 202019th International Symposium INFOTEH-JAHORINA

[2] Hong Yul Yang, Ewan Tempero, Hayden Melton "An Empirical Study into Use of Dependency Injection in Java" 2020 University of Auckland,Auckland, New Zealand

[3] M. Fowler, "Inversion of control containers and the dependency injection pattern," 2004.

[4] R. C. Martin. The Dependency Inversion Principle. C++Report, 8(6):61–66, June 1996

[5] A. Shvets, M. Pavlova, and G. Frey, "Design patterns explained simply," URL: https://sourcemaking. com Online, 2015.