Analysis of Dependency Injection Frameworks and their Features

Nicholas Kastanos (1393410)
School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa
ELEN4010 Software Development III
¡¡¿¿

Abstract

I. Introduction

II. Dependency Injection

The origins of Dependency Injection are rooted in the concepts of the *Dependency Inversion Principle* (DIP) which was first discussed by Martin in 1996 [1], and the *Inversion of Control* (IoC) principle, by Foote [2].

These principles state that classes should not depend on other concrete classes, but should only depend on abstract interfaces. This allows the programmer to avoid what Martin describes as a "Bad Design", and avoids rigidity, frailty,and immobility in the code.

By having classes depend on abstract interfaces, the implementation of the dependencies can be easily altered without needing the change the code of the class. This makes the code more flexible and less fragile, leading the easier future developments or changes.

Dependency Injection (DI) provides the solution to this problem by linking the required implementation to the interface without the dependant class having to have knowledge of which implementation is being used [3], [4].

It is possible to implement DI using the standard libraries for many languages, however this requires well structured and planned code, and can become complicated and prone to errors in large programs. In order to simplify the inclusion of DI in code, many DI Frameworks have been developed.

III. DI Frameworks

DI frameworks are structured to encourage good use of the underlying principles [5], [6], and are designed to make the use of DI principles easy and simple. DI Frameworks in Java are usually implemented in two structures: XML configuration files, or Injector Java objects. Some of the more popular DI Frameworks are discussed below.

Spring Framework is a comprehensive framework which provides more functionality than DI. It also supports features such as data binding, database connections, and web framework support. The DI Framework section of Spring makes use of both Plain Old Java Objects (POJO) anbd XML files to configure the bindings and injections.

Guice is a lightweight DI Framework developed by Google to provide fast and simple DI using POJO principles. Unlike Spring Framework, Guice only provides DI capabilities.

Dagger 2 is a DI-only framework which is fully static and is implemented at compile time. This fully implements the program's stack at compile time which allows for easier debugging and faster runtimes. Additionally it as able to run on Android systems, which increases the reach of the framework.

IV. Features of DI Frameworks

Many DI frameworks have both similar and different features or types of DI which it can implement. These features can either be classified as Essential Features or Supplementary Features.

Essential Features are central to implementing the DIP and IoC principles, and are generally included in every DI Framework. Supplementary Features, however, provide additional utility or Quality of Life improvements to assist in implementing more complicated DI structures.

1) Constructor Injection: Constructor Injection occurs when the object is injected into the constructor as an argument. This allows an object to be provided with the required dependency on construction without violating the DI Principle. Spring Framework, Guice and Dagger 2 all offer this form of injection.

2) Method Injection: Similarly to Constructor Injection, Method Injection occurs when the dependency is passed into a method as an argument. This form of injection differs from Constructor Injection by creating the dependency when the method is called, as opposed to when the object is constructed. This interaction may result in more distributed computational requirements should the construction be computationally expensive. Spring Framework, Guice and Dagger 2 all offer this form of injection.

3) Field Injection: Field Injection allows a dependency to be injected directly into an object's fields. This generally occurs on instantiation of the object, and allows the constructor of the object to be free of some injections. Guice and Dagger 2 contain this functionality.

4) Defaults or Optional Injection: Default values allow dependency injection to become optional, where the dependency does not have to be created. If the injection does not occur, the program will be able to use a default value for the required dependency. This feature encourages reuse of code,

and reduces fatal program errors, however it may increase the amount of logical errors which are present. All three of the frameworks mentioned above have this feature.

5) Named Linkings: This feature allows links between objects and their dependencies to be referred to by more 'human readable' names. This increases the readability of code, and does not afford extra functionality. While this does not provide the frameworks with more capabilities, Spring, Guice, and Dagger 2 all support this feature.

6) Chained Dependencies: Chained dependencies occur when one object's dependency contains objects which itself is dependent on. While this is possible to implement using Guice and Dagger 2, Spring Framework provides active support for is feature.

7) Provider Injections: Provider injections can be used when multiple values, or objects, need to be provided to an object. This ensures than a different instance of the dependency is provided each time. Spring Framework, Guice and Dagger 2 all offer this form of injection.

A. Essential Features

According to the guidelines set out in Section IV, Constructor Injection, Method Injection, and Provider Injection are Essential Features for a DI Framework.

Each of these methods of injection provide a framework for implementing the DIP and IoC Principles. Constructor Injection allows dependencies to be set up at the beginning of the object's lifetime, while Method Injection allows the dependencies to be created in the middle of the lifetime.

Provider Injections are also essential due to the ability to inject large amounts of data and dependencies. Without it, objects would be limited in the amount of dependencies which can be created.

B. Supplementary Features

The remaining features, Field and Optional Injection, Named Linkings and Chained Dependencies, can be classified as Supplementary Features. Field Injection can be implemented using Constructor or Method injection by passing a dependency into the objects as a parameter, and setting it to a private member. Optional Injections removes the need for a default injected object, which reduces the complexity of the code. Chained Dependencies can be complicated to program and test without the support of the framework. This may result in longer development times or lower product quality.

V. Practical Implementation using Guice

In order to explore the features and usability of DI Frameworks, a sample Java program has been developed using Google's Guice framework (see Appendix B). The sample program emulates how different payment modules can be integrated into a payment or credit transaction system. The user is presented with a login screen, where a username and password is accepted. Once the user has logged in, the user is placed into a loop where they can repeatedly make transactions until they logout.

The type of transaction is implemented using the Guice DI Framework. An administrator account can change the `Transaction` class's implementation between a `SecureTransaction`, `FastTransaction`, and `InsecureTransaction`. This implementation is then injected into the `PaymentService`'s constructor via a Provider class, and is used to complete the transaction using different methods.

While the implementations of the application are fairly trivial, they highlight the capabilities and features of the framework.

VI. Analysis of DI Frameworks

The use of DI Frameworks in code affects it in a multitude of ways, as the decoupling of classes leads in increased maintainability, extensibility, and reusability [4], [7].

DI Frameworks assist in decoupling dependent objects from one another, which reduces the sensitivity to changes between different parts of the program. This allows the different parts of the program to be changed easily without needing to change other sections of code, increasing flexibility. A program or library which already has clients making use of its interface will not be required to change their own program should the implementation be loosely coupled. A side-effect of using the framework is that the code becomes highly dependent on the DI framework, which does limit possible development of the program. Should the framework not support a feature which is essential

to an aspect of the program or expansion of the program, the entire application must be reprogrammed to accommodate a new framework.

Since code which is implemented using a DI Framework is highly flexible, the code becomes highly reusable as the implementation can be modified to suit a specific need. The sample application in Section V which using DI to change the security level of a transaction can be modified to change the service provider from MasterCard to Visa or a cryptocurrency.

For many DI Frameworks, the lifetime of the injected objects is controlled by the parent object. This decreases the amount of unnecessary memory usage of the program.

While DI Frameworks provide many benefits, there are drawbacks to using the frameworks. The frameworks each require a specific structure, be that XML files or additional POJO classes. These additional structures take time to learn and implement, increasing the development time of the project. Additionally, the dependencies need to be linked at runtime, which increases the running times of the program. Dagger 2, however, accomplishes all of its linking at compile time, and is significantly more efficient at runtime than other frameworks.

VII. Conclusion

References

[1] R. C. Martin, "The dependency inversion principle," *The C++ Report*, vol. 8, no. 6, pp. 61–66, June 1996.

[2] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, 1988.

[3] T. Marston, "Dependency injection in evil," [Online] http://www.tonymarston.net/php-mysql/dependency-injection-is-evil.html, Oct 2016.

[4] 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. Perth, WA, Australia: IEEE, Mar 2008, pp. 239–247.

[5] R. Johnson, "J2ee development frameworks," *Computer*, vol. 38, no. 1, pp. 107 – 110, Jan 2005.

[6] D. Doomen, "Dont blame the dependency injection framework," [Online] Java Zone - Tutorial https://dzone.com/articles/dont-blame-the-dependency-injection-framework, May 2018.

[7] E. Razina and D. Janzen, "Effects of dependency injection on maintainability," in *IASTED International Conference on Software Engineering and Applications*, Cambridge, MA, Nov 2007, pp. 7–12.

Appendix A
Program Screenshots

APPENDIX B
SOURCE CODE

```
package paymentapp;

import com.google.inject.Guice;
import com.google.inject.Injector;
import javax.swing.JOptionPane;

public class PaymentApp {

    public static void main(String[] args) {
        String username = JOptionPane.showInputDialog("Enter your username (Q or q to Quit).");
        if (username.equalsIgnoreCase("q")) {
            System.exit(0);
        }
        do {
            String passwrd = JOptionPane.showInputDialog("Enter your password.");
            if (username.equalsIgnoreCase("admin") && passwrd.equalsIgnoreCase("admin")) {
                PaymentApp.changeTransactionType();
            } else {
                PaymentApp.transactionMenu(username, passwrd);
            }
            username = JOptionPane.showInputDialog("Enter your username (Q or q to Quit).");
        } while (!username.equalsIgnoreCase("q"));
    }

    private static void changeTransactionType() {
        int selection = Integer.parseInt(JOptionPane.showInputDialog("Choose a Transaction
    Type:\n1) Secure;\n2) Fast;\n3) Insecure;\nEnter the number of your selection."));
        while (selection < 1 || selection > 4) {
            JOptionPane.showMessageDialog(null, "Invalid selection.", "Error",
    JOptionPane.WARNING_MESSAGE);
            selection = Integer.parseInt(JOptionPane.showInputDialog("Choose a Transaction
    Type:\n1) Secure;\n2) Fast;\n3) Insecure;\nEnter the number of your selection:"));
        }
        switch (selection) {
            case 1:
                TransactionProvider.setTransactionType(TransactionType.SECURE);
                JOptionPane.showMessageDialog(null, "Set Transaction Type to SECURE.");
                break;
            case 2:
                TransactionProvider.setTransactionType(TransactionType.FAST);
                JOptionPane.showMessageDialog(null, "Set Transaction Type to FAST.");
                break;
            case 3:
                TransactionProvider.setTransactionType(TransactionType.INSECURE);
                JOptionPane.showMessageDialog(null, "Set Transaction Type to INSECURE.");
                break;
            default:
                JOptionPane.showMessageDialog(null, "Invalid Selection.", "Error",
    JOptionPane.WARNING_MESSAGE);
                System.exit(1);
        }
    }

    private static void transactionMenu(final String username, final String passwrd) {
        JOptionPane.showMessageDialog(null, "Successfully Logged In.");
        Injector injector = Guice.createInjector(new PaymentServiceModule());
        PaymentService paymentService = injector.getInstance(PaymentService.class);
        int selection = 1;
        while (selection != 0) {
            do {
                selection = Integer.parseInt(JOptionPane.showInputDialog("Choose an
    Option:\n0) Logout;\n1) Transaction;\nEnter the number of your selection."));
```

```
            } while (!(selection < 2 && selection >= 0));
            switch (selection) {
                case 0:
                    continue;
                case 1:
                    break;
                default:
                    JOptionPane.showMessageDialog(null, "Invalid Selection.", "Error",
    JOptionPane.ERROR_MESSAGE);
                    System.exit(1);
            }
            double amount = Double.parseDouble(JOptionPane.showInputDialog("Enter the amount
    to transfer (Use numbers only)."));
            String destination = JOptionPane.showInputDialog("Enter the destination of the
    transfer:");
            paymentService.makePayment(username, passwrd, amount, destination);
        }
    }
}
```

Listing 1. PaymentApp.java

```
package paymentapp;

import com.google.inject.Inject;

public class PaymentService {
    private Transaction transaction;

    @Inject
    public PaymentService(Transaction transaction){
        this.transaction = transaction;
    }

    public void makePayment(final String username, final String passwrd, final double amount,
    final String destination){
        transaction.makeTransaction(username, passwrd, amount, destination);
    }
}
```

Listing 2. PaymentService.java

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

public class PaymentServiceModule extends AbstractModule{
    @Override
    protected void configure(){
        bind(Transaction.class).toProvider(TransactionProvider.class);
    }
}
```

Listing 3. PaymentServiceModule.java

```
package paymentapp;

public interface Transaction {
    public void makeTransaction(final String username, final String passwrd, final double
    amount, final String destination);
}
```

Listing 4. Transaction.java

```
package paymentapp;

import javax.swing.JOptionPane;
```

```
public class SecureTransaction implements Transaction{

    @Override
    public void makeTransaction(String username, String passwrd, double amount, String
    destination) {
        String check = JOptionPane.showInputDialog(amount + " is to be transfered to " +
    destination + "\nPlease re-enter your password to continue.");
        if(check.equals(passwrd)){
            JOptionPane.showMessageDialog(null, "Transfer Successful!");
        } else {
            JOptionPane.showMessageDialog(null, "Transfer Successful!", "Error",
    JOptionPane.ERROR_MESSAGE);
        }
    }
}
```

Listing 5. SecureTransaction.java

```
package paymentapp;

import javax.swing.JOptionPane;

public class FastTransaction implements Transaction{

    @Override
    public void makeTransaction(String username, String passwrd, double amount, String
    destination) {
        JOptionPane.showMessageDialog(null, "Transaction of " + amount + " transfered to " +
    destination);
    }

}
```

Listing 6. FastTransaction.java

```
package paymentapp;

import javax.swing.JOptionPane;

public class InsecureTransaction implements Transaction{

    @Override
    public void makeTransaction(String username, String passwrd, double amount, String
    destination) {
        JOptionPane.showMessageDialog(null, "Your username " + username + " and password " +
    passwrd + " have been broadcast publicly.");
    }

}
```

Listing 7. InsecureTransaction.java

```
package paymentapp;

import com.google.inject.Provider;
import javax.swing.JOptionPane;

public class TransactionProvider implements Provider<Transaction>{

    private static TransactionType type = TransactionType.SECURE;

    @Override
    public Transaction get() {
        switch(type){
            case SECURE:
```

```
                JOptionPane.showMessageDialog(null, "Secure Transaction Mode");
                return new SecureTransaction();
            case FAST:
                JOptionPane.showMessageDialog(null, "Fast Transaction Mode");
                return new FastTransaction();
            case INSECURE:
                JOptionPane.showMessageDialog(null, "Insecure Transaction Mode");
                return new InsecureTransaction();
            default:
                JOptionPane.showMessageDialog(null, "Invalid Selection. Using Secure
    Transactions", "Error", JOptionPane.WARNING_MESSAGE);
                return new SecureTransaction();
        }
    }
    public static void setTransactionType(TransactionType type){
        TransactionProvider.type = type;
    }

}
```

Listing 8. TransactionProvider.java

```
package paymentapp;

public enum TransactionType {
    SECURE,
    FAST,
    INSECURE
}
```

Listing 9. TransactionType.java