Dependency Injection for loose coupling of Objects

Dr.Soly Mathew Biju

Abstract- Object oriented software may involve a number of objects that are closely coupled, making it very cumbersome for efficient software testing due to dependencies. Managing and keeping track of lifetimes of various objects becomes a difficult task. Dependency Injection is a design pattern that introduces dependency at interface levels .Configuration information of the objects wired together is maintained separately and this information can be changes at runtime. Dependency Injection technique helps in designing software with loosely coupled objects thus provides a better object oriented design.

Index Terms- Dependency injection, interface, implements, dependencies, factory method, Spring, Guice, accidental complexity, Service Oriented Architecture, Test Driven Development.

1. INTRODUCTION

Object-oriented design and development is becoming very popular in today`s software development environment [16].Creating a complex object oriented application will involve creating a number of objects that are tightly coupled. There could be high level of dependencies between objects thus making it difficult to design reusable components as one of the guidelines to design a reusable component is loose coupling among objects. Dependencies between object could also makes it very difficult to design unit tests. Coupling is transitive in nature object A depends on Object B and if object B depends on object C then object A also depends on object C [11].Any change to object C will affect both object B and A.

Component composition used in currently available component modules apply either direct or indirect message passing as connection schemes which lead to tight coupling [12].

Loose coupling between these objects would take away the entire dependency web that exists between these objects giving a clearer and maintainable code that could be easily tested.

Loosely coupled objects could be easily unit tested. Software testing is a comprehensive set of activities conducted with the intent to finding errors in software[15].Test cases generation and methods are one of the most challenging processes during software testing phase[14].

In test driven and software component based development which is gaining a lot of popularity, the focus is on developing reusable and testable objects hence objects and its dependencies must be loosely coupled.

Dependency Injection is a simple pattern offers a design with loose coupling among objects[4].This allows objects to complete their roles in the model and abstracts away queries regarding instantiation or lifecycles of other objects dependent of them(their dependencies). DI allows objects to be injected from outside without relying on the classes to create these objects. Software maintenance consumes about 70% of the software life cycle. Software maintainability could be improved by reducing coupling among modules used in the application. Software coupling has been linked to maintainability [9].Recent studies have shown a trend towards lower coupling numbers in projects with a dependency injection count of 10% or more was observed [8].

2. INTERFACES

In object oriented development wiring of various objects, maintaining their dependencies and managing their lifetime are very important.

A java interface type declares a set of methods and their signatures. An interface is used to specify required operations [2]. Classes can now implement this interface and has the freedom to provide the actual implementation code for the methods of the interface. Different classes may implement these methods in different ways. By coupling an object to an interface instead of a specific implementation, you have the freedom of using any implementation with minimal change and risk [3].

2.1 An Example

The promotion interface in example1 defines an interface having a method increase_salary .The interface does not specify how the method has to be implemented. It only specifies that it provides a service. It is up to the class that implements the interface to determine the actual code based on some company policies.

```
public interface promotion{
public double increase_salary();
}
```

Example 1.Interface promotion

Example 2 shows class manager which implements the interface promotion. The class manager implements the interface promotion .It provides the definition for the method increase_salary which returns the new salary.

```
public class manager implements promotion {
public double increase_salary(double  basic salary)
{
return basic salary*(10/100);
}
}
```

Eaample 2. Class manager implements promotion

Example3 shows the consumer class , class SpecialContractor that uses the method increase_salary implemented in the manager class to calculate the bonus to be given to the contractor.

```
public class SpecialContractor()
  {
   private double salary;
   private final promotion promote;
    public SpecialContractor()
  {  promote= new manager();
    }
  public double bonus()
   {
    return
    promote.increase_salary(salary);
    }
  }
```

Example 3. Class SpecialContractor

The UML diagram depicting the interface dependency of the code given example1,example 2 and example3 is shown in figure1.

The problem with example 3 is that in the constructor of the class SpecialContractor , an object of class manage is created. This binds the SpecialContractor to the concrete implementation of the class manager. The class SpecialContractor is not easily unit testable or reusable as the actual service may have other external dependencies.

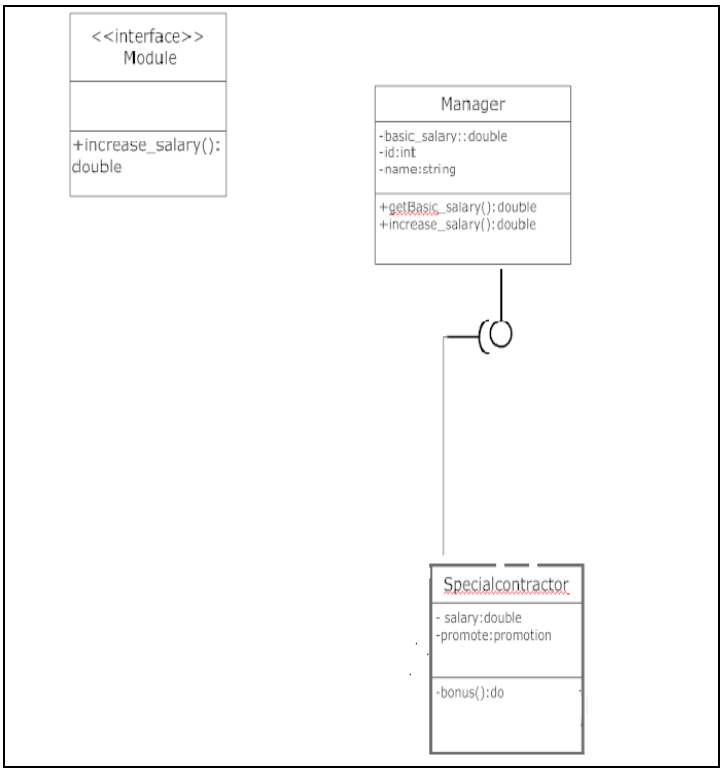

Figure 1 UML diagram showing the interface dependency

3. USING FACTORY PATTERN

Another solution to the problem in example 3 is to use factory pattern to obtain instances of the class manager that implements the service promotion. Gamma et al [4] in their book describe purpose of Abstract Factory pattern as

"To construct and instantiate a set of related objects without specifying their concrete objects."

```
public class SpecialContractor()
  {
   private double salary;
   private final promotion promote;
  protected SpecialContractor()
    {
     promote=ServiceFactory.create();
```

```
   }
 public double bonus()
  {
  return  promote.increase_salary(salary);
   }
 }
```

Example 4. Using factory method

Factories facilitates for a software application to put together various objects and its components without showing the dependencies between these components. The client can then call the factory methods to create instances of the classes without having to configure or create an instance of the class. Factory patterns may not always be the best because all the dependencies of the class must be known to the factory at compile time. Introducing a static factory method solves the problem of depending on a concrete implementation class but makes the code difficult to maintain and less flexible. The disadvantage of using the factory pattern in such a situation is that for all concrete classes separate factory classes have to design causing a lot of boiler plate codes within the main application structure. All the consumers currently instantiating the class using the new operator must be changed to call the factory method.
This will eventually read to accidental complexity in addition to the existing cyclomatic and essential complexity of the code. Accidental complexity relates to problems that we create on our own and can be fixed — for example, the details of writing and optimizing Assembly code or the delays caused by batch processing. Essential complexity is caused by the problem to be solved, and nothing can remove it [7]. Factory classes have to be designed as Singleton classes which have a certain level of complexity and lifecycle management problems.

4. DEPENDENCY INJECTION

Dependency Injection provides solution to all these problems. It refers to a process of providing an external dependency to a software component. It also aids in design by interface and facilitates Test Driven Development(TDD). Test-driven design (TDD), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfil that test and refactoring (Brooks 2003).TDD is primarily a design technique with a side effect of ensuring that your source code is thoroughly unit tested[10].

Dependency injection containers take care of object construction, injection and life cycle management of all the objects and all its dependencies[5] .Some of the lightweight DI container available in the market today are Spring, Guice, HiveMind. Spring and Guice is the most popularly used DI containers.

Spring lets you define separate configuration files which are very similar to deployment files used in java servelets. Below is an example of a deployment file used in servelets.

An example a web.xml file is given below

```
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

<servlet>
<servlet-name>IntroServlet</servlet-name><servlet-
class> javaservlet.example1.IntroServlet </servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>WelcomeServlet</servlet-name>

<url-pattern>/WelcomeServlet</url-pattern>
</servlet-mapping>

</web-app>
```

Example 5 Web.xml file

The web.xml file in the example provides the servlet name and the servlet mapping and the url details.
DI container is a separate file which is responsible for initializing instances of the classes wherever required at runtime. Any change made to the this file does not require any recompilation of the main source code ; the changes are incorporated dynamically at runtime.
In case of example SpecialContractor class, the DI container will inject a concrete instance of promotion hence example4 could be modified as given below

```
public class SpecialContractor
{
  private double salary;
```

```
  private final promotion promote;
   protected SpecialContractor(promotion promote)
  {  this.promote=promote;
//instance of promotion is injected by DI
    }
  public double bonus()
  {   return  promote.increase_salary(salary);
   }
 }
```

Example 6. Class SpecialContractor using DI

In Example 6, the factory method has been changed to use the Spring BeanFactory API to create the bean.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
  "-//SPRING//DTD BEAN//EN"
  "http://www.springframework.org/dtd/spring-
beans.dtd">
<beans>
  <bean id="contractor" class=
        "org.elj.SpecialContractor ">
     <constructor-arg><ref bean=
        "managers"/></constructor-arg>
  </bean>
  <bean id="managers" class="org.elj.Manager"/>
</beans>
```

Example 7. Spring beans.xml configuration file.

Setter functions could also be used as injectors instead of using constructors as injector. It is advantageous to use setter as injectors as compared to using constructors as injectors as they are simple and the name describes the function and the parameter it requires. Moreover, Setters are inherited unlike constructors which are not implicitly inherited.

However it is a design level decision to select the appropriate injectors. But in applications that require immutable objects it is preferable to use constructor injection [3].

5. USING GUICE

There are a number of dependency containers available in the industry today. We have seen examples of spring. Another very popular DI containers is Guice. Google Guice [6] uses java5 annotation to provide the same injection service. For example considered above the class SpecialContractor that will now have the constructor annotated with @inject in order to request injection from the Guice engine:

```
public class SpecialContractor
{
  private double salary;
  private @Inject promotion promote;
  protected SpecialContractor(promotion promote)
  {  this.promote=promote;
//instance of promotion is injected by DI
   }
  public double bonus()
  {   return  promote.increase_salary(salary);
   }}
```

Example 8. Class SpecialContractor using Guice

The @Inject annotation indicates where –to-inject. Guice provides a Module that specifies what-to-inject.The module for the class in example7 is as follows

```
public class AppModule implements Module
{ public void configure(Binder binder)
  {
Binder.bind(promotion.class).to(manager.class)
   .in(scopes.SINGLETON);
}
}
```

Example 9. Guice Module, another DI container

Efficient and flexible solution based on Service Oriented Architecture has been proposed for various services like sending services to Web Phone [13].In a service Oriented Architecture (SOA) adopting layered approaches, all dependencies between the service layer, business logic and data access layer can be injected using DI.

6. CONCLUSION

The paper can be concluded by highlighting the benefits of using DI in Object oriented technology.

The configuration information regarding the implementation classes can be changed at runtime. Dynamic runtime changes can be implemented at the architectural level using the DI approach.

Evolution and extensibility: All systems evolve over time, and similarly this can be represented by switching components and changing wiring This can easily be applied through DI. Different objects can be wired using the deployment file.

DI helps in reducing the amount of boilerplates and the resulting code is more maintainable.

Behaviour driven design for components

Components communicate via interfaces hence it is

possible to model the protocol of a component (and interfaces) using an extended sequence diagram or using a simple textual language.
With DI dependency can be restricted to the interface level and have mock independent classes which can be injected using a DI container thus making unit testing easier
DI facilitates better Object Oriented Design and aids in the reusable component based development.
DI helps in making the modules simpler and unit testing easier.

REFERENCES

[1] Brooks,2003. *The Mythical Man-Month: Essays on Software Engineering*, Anniversary Edition (2nd Edition) Addison-Wesley.
[2] H.Cay, *Bi g Java*., John Wiley & sons ,Inc.Hoboken,, USA.
[3] W. Jeremy 2006, "Dependency injection and testable objects", Dr.Dobbs. http://www.drdobbs.com/tools/185300375
[4]Gamma,E.,Helm,R., Vlissides ,R.J,1995. *Design Patterns*, Addison-Wesley.
[5] G. Debasish, 2008. "Dependency Injection- A pattern for loosely coupled collaboration of Objects", Computer society of India Communications ,31-34.
[6] User Guide, 1995: http://code.google.com/p/google-guice.
[7] Reference manual, 2010: http://static.springframework.org/spring/docs/2.5.x/reference/index.html
[8] R.Ekaterina, J.David , 2007. Effects of dependency injection on maintainability, Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, pg 7-12
[9] P. Denys, M. Andrian , The Conceptual Coupling Metrics for Object-Oriented Systems, Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006, pg 469 - 478
[10] A. David , ,2003."Test-driven development: A practical guide", Prentice Hall.
[11] R. James, Blaha ,1991. Object Oriented Modeling and design, Prentice Hall.

[12] Sanatnama, H., A.A.A. Ghani, N.K. Yap and M.H. Selamat, 2008. Mediator connector for composition of loosely coupled software components. J. Applied Sci., 8: 3139-3147.

[13] Wang, S. and L. Jun, 2011. A framework-based content-orientated services delivery technology for 3G network. Inform. Technol. J., 10: 779-788.

[14] osindrdecha, N. and J. Daengdej, 2010. A test case generation process and technique. J. Software Eng., 4: 265-287.

[15] Roongruangsuwan, S. and J. Daengdej, 2010. A test case prioritization method with practical weight factors. J. Software Eng., 4: 193-214.

[16] Parthasarathy, S. and N. Anbazhagan, 2006. Analyzing the software quality metrics for object oriented technology. Inform. Technol. J., 5: 1053-1057.