جامعة مولاي إسماعيل
UNIVERSITÉ MOULAY ISMAÏL

كلية العلوم
FACULTÉ DES SCIENCES

Master Intelligence Artificielle et Analyse des Données

Systèmes Distribués

COMPTE RENDU DU TP1

Réalisé par :

HALIMA DAOUDI

Année universitaire : 2023 – 2024

INTRODUCTION

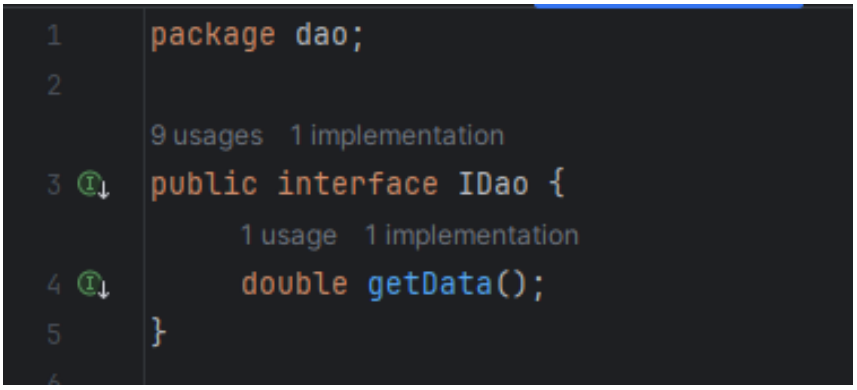
Dans ce TP, je vais me plonger dans les concepts avancés de programmation orientée objet, notamment l'inversion de contrôle (IoC) et l'injection de dépendances, pour appréhender leur importance dans la création d'applications modulaires et facilement maintenables.

Je vais mettre en pratique ces principes en développant une application simple mais instructive, qui me permettra d'explorer le couplage faible et les différentes méthodes d'injection de dépendances. Voici les tâches que je vais accomplir :

1. Créer l'interface **IDao** avec une méthode getDate.
2. Créer une implémentation de cette interface.
3. Créer l'interface **IMetier** avec une méthode calcul.
4. Créer une implémentation de cette interface en utilisant le couplage faible.
5. Faire **l'injection des dépendances** en utilisant différentes approches :
   a. Par instanciation statique.
   b. Par instanciation dynamique.
   c. En utilisant le Framework Spring, à la fois en version XML et en version annotations.

1. Créer l'interface IDao avec une méthode

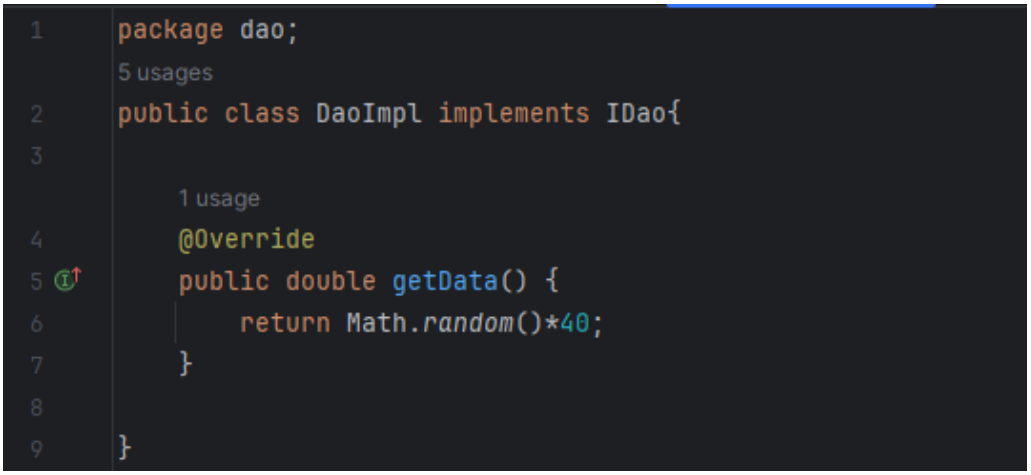
J'ai utilisé le package **dao** pour créer une interface appelée **IDao**. Cette interface définit une méthode **getData** qui renvoie un double qui sera utilisée pour récupérer des données dans notre application. En définissant cette méthode dans une interface, nous fournissons une abstraction pour accéder aux données de manière uniforme, ce qui facilite l'extension et la maintenance de notre code.



2. Créer une implémentation de cette interface

J'ai ensuite créé une classe nommée **DaoImpl** dans le package "**dao**", qui implémente l'interface **IDao**. Dans cette classe, j'ai fourni une implémentation concrète de la méthode getData qui retourne un double aléatoire entre 0 et 40, simulant la récupération de données à partir d'une source réelle.

En utilisant un **couplage faible**, l'implémentation de IDao ne dépend pas directement de l'implémentation de DaoImpl, mais plutôt de l'interface IDao, ce qui rend notre code plus flexible et facile à maintenir.



3. Créer l'interface IMetier avec une méthode calcul

Dans le package "**metier**", j'ai défini une interface **IMetier**. Cette interface propose une méthode calcul pour effectuer des opérations métier.

En créant cette interface, nous établissons une norme pour les calculs métier dans notre application, facilitant ainsi la cohérence et la flexibilité du code.

```
package metier;

10 usages  1 implementation
public interface IMetier {
    4 usages  1 implementation
    double calcul();
}
```

4. Créer une implémentation de cette interface en utilisant le couplage faible

Dans le package "**metier**", j'ai implémenté l'interface **IMetier** avec la classe **MetierImpl**. Cette classe utilise un couplage faible en dépendant de l'interface **IDao** plutôt que d'une implémentation concrète.

La méthode calcul de MetierImpl récupère des données à partir de l'objet IDao et effectue un calcul métier. Ce couplage faible permet une meilleure modularité et extensibilité du code, car la classe MetierImpl n'est pas directement liée à une implémentation spécifique de IDao.

```
package metier;
import dao.IDao;

5 usages
public class MetierImpl implements IMetier{

    // couplage faible
    2 usages
    private IDao dao;
    4 usages
    @Override
    public double calcul() {
        double temp = dao.getData();
        return temp *1000 / Math.cos(temp * Math.PI);
    }
    no usages
    public void setDao(IDao dao) { this.dao = dao; }
}
```

5. Faire l'injection des dépendances :

a. Par instanciation statique

Dans la classe **Presentation** du package "**pres**", j'ai effectué l'injection des dépendances en instanciant directement les objets **DaoImpl** et **MetierImpl** avec l'opérateur "**new**" dans la méthode main. Ensuite, j'ai utilisé la méthode **setdao** de MetierImpl pour fournir l'instance de DaoImpl.

Bien que cette méthode soit simple, elle rend le code moins flexible et plus difficile à tester, car les dépendances sont codées en dur dans le code source.

```java
package pres;

import dao.DaoImpl;
import metier.MetierImpl;

public class Presentation {
    public static void main(String[] args) {
        // injection des dependances par instanciation statique ( new )
        DaoImpl dao = new DaoImpl();
        MetierImpl metier= new MetierImpl() ;
        metier.setDao(dao);
        System.out.println("Résultat : " + metier.calcul());

    }
}
```

b. Par instanciation dynamique

Pour réaliser l'injection des dépendances de manière dynamique, j'ai créé un fichier de configuration "**config.txt**" qui contient les noms des classes à utiliser pour l'implémentation de **IDao** et **IMetier**.

```
1    dao.DaoImpl
2    metier.MetierImpl
```

Dans la classe **Presentation2** du package "**pres**", j'ai mis en œuvre l'injection des dépendances de manière dynamique en utilisant un fichier « **config.txt** ».

Dans la méthode main, j'ai utilisé un scanner pour lire les noms de classe. Ensuite, j'ai utilisé la réflexion pour charger dynamiquement les classes spécifiées et créer des instances des interfaces IDao et IMetier. Et j'ai également invoqué la méthode setdao de l'objet IMetier pour injecter l'objet IDao créé précédemment.

Cette approche permet une grande flexibilité, car les dépendances sont spécifiées dans un fichier externe, facilitant les modifications ultérieures sans nécessiter de modification du code source.

```java
package pres;
import dao.IDao;
import metier.IMetier;
import java.io.File;
import java.lang.reflect.Method;
import java.util.Scanner;

public class Presentation2 {
    public static void main(String[] args) throws Exception{
        Scanner sc = new Scanner(new File( pathname: "config.txt"));

        // lit la 1er ligne du file qui represente le nom de la classe qui implemente Idao
        String daoClass = sc.nextLine();
        // instancier l'interface Idao en utilisant son nom chargé par scanner
        Class cDao = Class.forName(daoClass);
        IDao dao = (IDao) cDao.newInstance() ;
        // lit la ligne suivante du file qui represente le nom de classe qui implemente Imetier
        String Metierclass = sc.nextLine();
        // instancier l'interface Imetier en utilisant son nom chargé par scanner
        Class cMetier = Class.forName(Metierclass);
        IMetier metier = (IMetier) cMetier.newInstance() ;

        Method method = cMetier.getMethod( name: "setDao" , IDao.class) ;
        method.invoke(metier,dao) ; // = metier.setDao(dao)
        System.out.println("Résultat : " + metier.calcul());
    }
}
```

c. En utilisant le Framework Spring :

Pour utiliser les fonctionnalités fournies par **Spring** et **Junit** dans mon application Java, j'inclus ces dépendances dans le fichier pom.xml de mon projet Maven.

```xml
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>6.1.5</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>6.1.5</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>6.1.5</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

- Version XML :

Pour l'injection des dépendances avec Spring en utilisant XML, On configure les beans dans un fichier XML. J'ai défini deux **beans** : "**dao**" et "**metier**", représentant respectivement les implémentations de **IDao** et **IMetier**.

Et, j'ai utilisé **<constructor-arg>** pour spécifier la dépendance de IMetier vers IDao, bien que **<property>** soit également une option.Cela garantit que **MetierImpl** est correctement initialisé avec sa dépendance, fournissant ainsi une gestion claire et efficace des dépendances dans notre application.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="dao" class="dao.DaoImpl"></bean>

    <bean id="metier" class="metier.MetierImpl">
        <!-- <property name="dao" ref="dao"></property> -->
        <constructor-arg ref="dao"></constructor-arg>
    </bean>
</beans>
```

Et j'ai **ajouté** un constructeur prenant en paramètre une instance de **IDao**. Cela permet à Spring d'injecter la dépendance de manière automatique lors de la création de l'objet **MetierImpl**. Et j'ai **supprimé** la méthode **setdao**, car l'injection de dépendances se fait désormais par le constructeur, alors cette méthode n'est plus nécessaire avec cette approche.

```java
package metier;
import dao.IDao;
5 usages
public class MetierImpl implements IMetier {

    // couplage faible
    2 usages
    private IDao dao;

    2 usages
    public MetierImpl(IDao dao) {
        this.dao = dao;
    }
    4 usages
    @Override
    public double calcul() {
        double temp = dao.getData();
        return temp * 1000 / Math.cos(temp * Math.PI);
    }
}
```

Dans ma classe **PresentationSpringXml** du package "**pres**", j'utilise Spring avec une configuration XML pour gérer les dépendances. J'instancie le contexte Spring à partir du fichier XML **"applicationcontext.xml",** puis je récupère l'instance de IMetier à l'aide de la méthode **getbean**. Spring se charge de l'injection des dépendances selon la configuration XML. Enfin, j'appelle la méthode calcul de IMetier et affiche le résultat.

```java
package pres;

import metier.IMetier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class PresentationSpringXML {

    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "ApplicationContext.xml");
        IMetier m =(IMetier) context.getBean( name: "metier");
        System.out.println(m.calcul());
    }
}
```

- Version Annotations :

Dans la classe **DaoImpl** du package "dao", j'ai annoté la classe avec **@Component("dao"),** ce qui indique à Spring de gérer cette classe comme un **bean** avec l'identifiant "dao".

```java
package dao;

import org.springframework.stereotype.Component;

5 usages
@Component("dao")
public class DaoImpl implements IDao{

    1 usage
    @Override
    public double getData() { return Math.random()*40; }


}
```

Dans la classe **MetierImpl** du package "metier", j'ai annoté la classe avec **@Component("metier").** J'ai utilisé **@Autowired** pour injecter automatiquement une instance de **IDao** dans la propriété dao, évitant ainsi la nécessité d'une méthode setter. La méthode calcul() utilise l'objet IDao pour récupérer des données et effectuer un calcul métier.

```java
package metier;
import dao.IDao;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

5 usages
@Component("metier")
public class MetierImpl implements IMetier{
    // couplage faible
    2 usages
    @Autowired
    private IDao dao;
    4 usages
    @Override
    public double calcul() {
        double temp = dao.getData();
        return temp *1000 / Math.cos(temp * Math.PI);
    }
    1 usage
    public void setDao(IDao dao) {
        this.dao = dao;
    }
}
```

J'ai utilisé **annotationconfigapplicationcontext** Dans la classe **PresAnnotation** du package "**pres**" pour charger le contexte Spring en spécifiant les packages à scanner pour rechercher les **beans annotés.** En récupérant l'instance de IMetier à partir du contexte Spring avec **getbean**(), j'ai pu appeler la méthode calcul pour obtenir le résultat du calcul métier.

```java
package pres;
import metier.IMetier;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PresAnnotation {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext( ...basePackages: "dao","metier");
        IMetier m = context.getBean(IMetier.class);
        System.out.println(m.calcul());
    }
}
```

CONCLUSION

En concluant ce TP, j'ai non seulement renforcé ma compréhension des principes fondamentaux de l'inversion de contrôle et de l'injection de dépendances, mais j'ai aussi mis en pratique ces concepts à travers différentes approches d'implémentation. Cette expérience m'a permis de voir concrètement l'impact du couplage faible sur la flexibilité et la maintenabilité du code.

L'exploration de différentes méthodes d'injection de dépendances, en particulier à travers le framework Spring, a enrichi ma boîte à outils de développeur, me préparant à créer des applications plus robustes et évolutives