COMPTE RENDU : J2EE

Réaliser par : BENSALLAM BADREDDINE

➔IoC: Inversion de contrôle :

Il permet au développeur de s'occuper uniquement du code métier (Exigences fonctionnelles) et c'est le Framework qui s'occupe du code technique(Exigences Techniques)

➔Couplage Fort :

//Prenons deux classes:Aet B

- ***Quand une classe A est liée à la classe B ,on dit que A est fortement couplée à B.***
- ***La classe Ane peut fonctionner qu'en présence de la classe B.***

Un objet fortement couplé est un objetqui a besoin de connaître les autres objets

etestgénéralementtrès dépendantles uns des autres.

La modification d'un objet dans une application fortement couplée nécessite souvent

de modifier d'autres objets.

➔*Couplage Faible :*

Prenons deux classes:Aet B

• La classe A implémente l'interface IA et la classe B implémente l'interface IB.

• Si la classe A est liée à l'interface IB par une association, on dit que la classe A et

la classe B sont liées par un couplage faible. • Dans ce cas, la classe A peut fonctionner avec n'importe quelle classe qui implémente

l'interface IB.

➔*Objectif du projet :*

L'objectif de ce TP est de savoir comment rendre un projet fermé à la modification et ouvert à l'extension en se basant sur le principe de couplage faible qui met l'accent sur les interfaces. Donc à la fin de ce TP, on sera capable à comprendre comment injecter des dépendances avec les deux méthodes (sans framework spring et avec le spring).

//dao :

```java
package dao;

4 usages
public interface IDao {
    no usages
    double getData();
}
```

➔On a créé un interface IDao qui contient les méthodes public que notre classe doit implémentés

```java
package dao;

import org.springframework.stereotype.Component;

no usages
@Component("dao")
public class DaoImpl implements IDao{
    no usages
    @Override
    public double getData() {

        System.out.println("Version base de données");

        double temp=Math.random()*40;
        return temp;
    }
}
```

➔La classe DaoImpl implémente la méthode getData de l'interface IDao. Cette méthode doit permettre de se connecter à la BD pour récupérer les données.

//Metier :

```java
package metier;

3 usages
public interface IMetier {
    no usages
    double calcul();
}
```

```java
package metier;

import ...

2 usages
@Component("metier")
public class MetierImpl implements IMetier{
    2 usages
    @Autowired
    private IDao dao;
    no usages
    @Override
    public double calcul() {
        double tmp=dao.getData();
        double res=tmp*540/Math.cos(tmp*Math.PI);
        return res;
    }


    public void setDao(IDao dao) { this.dao = dao; }
}
```

➔**L'interface IMetier doit contenir que les méthodes qui permet d'atteindre les besoins fonctionnels.**

```java
package ext;

import dao.IDao;

1 usage
public class DaoImpl2 implements IDao {
    no usages
    @Override
    public double getData() {
        System.out.println("Version Capteurs");
        double temp=6000;
        return temp;
    }
}
```

```java
package ext;

import dao.IDao;

1 usage
public class DaoImplVWS implements IDao {

    no usages
    @Override
    public double getData() {
        System.out.println("Version web service");
        return 90;
    }
}
```

➔On ajoute une extension DaoImplVWS dans la couche dao pour éviter de modifier au code.

```java
package pres;

import ...

no usages
public class Presentation {
    no usages
    public static void main(String[] args) {
        /*
        Injection des dépendances par
        instanciation statique => new
         */
        DaoImpl2 dao= new DaoImpl2();
        MetierImpl metier=new MetierImpl();
        metier.setDao(dao);
        System.out.println("Résultat="+metier.calcul());
    }
}
```

```java
package pres;

import ...

no usages
public class PresSpringAnnotations {
    no usages
    public static void main(String[] args) {
        ApplicationContext context= new AnnotationConfigApplicationContext("dao","metier");
        IMetier metier= context.getBean(IMetier.class);
        System.out.println("Résultat=>"+metier.calcul());
    }
}
```

```java
package pres;

import ...

no usages
public class PresSpringXML {
    no usages
    public static void main(String[] args) {
        ApplicationContext context= new ClassPathXmlApplicationContext("applicationContext.xml");
        IMetier metier= (IMetier) context.getBean("metier");
        System.out.println("Résultat=>"+metier.calcul());

    }
}
```

```java
package pres;

import ...

no usages
public class PresSpringXML {
    no usages
    public static void main(String[] args) {
        ApplicationContext context= new ClassPathXmlApplicationContext("applicationContext.xml");
        IMetier metier= (IMetier) context.getBean("metier");
        System.out.println("Résultat=>"+metier.calcul());

    }
}
```

➔Pour faire l'injection des dépendances on va travailler avec les setter qui permet de lier deux objets par une association.

➔On fait appel à la couche metier pour faire le traitement afin d'afficher le résultat final dans la console.

➔*Injection des dependances :Les annotations*

Pour injecter les dépendances avec le framework Spring en utilisant les annotations. Premièrement, on doit créer un projet maven (logiciel d'automatisation de gestion de projet), c'est un projet java qui contient un fichier pom.xml, ce dernier regroupe l'ensemble des dépendances.