

Rapport J2EE

Spécialité: Génie INFORMATIQUE RESEAUX

13 mars au 20 mars 2023

Sous la direction de Mme Badri Tijane

Rédiger par: Fatima Ezzahra Lahouir

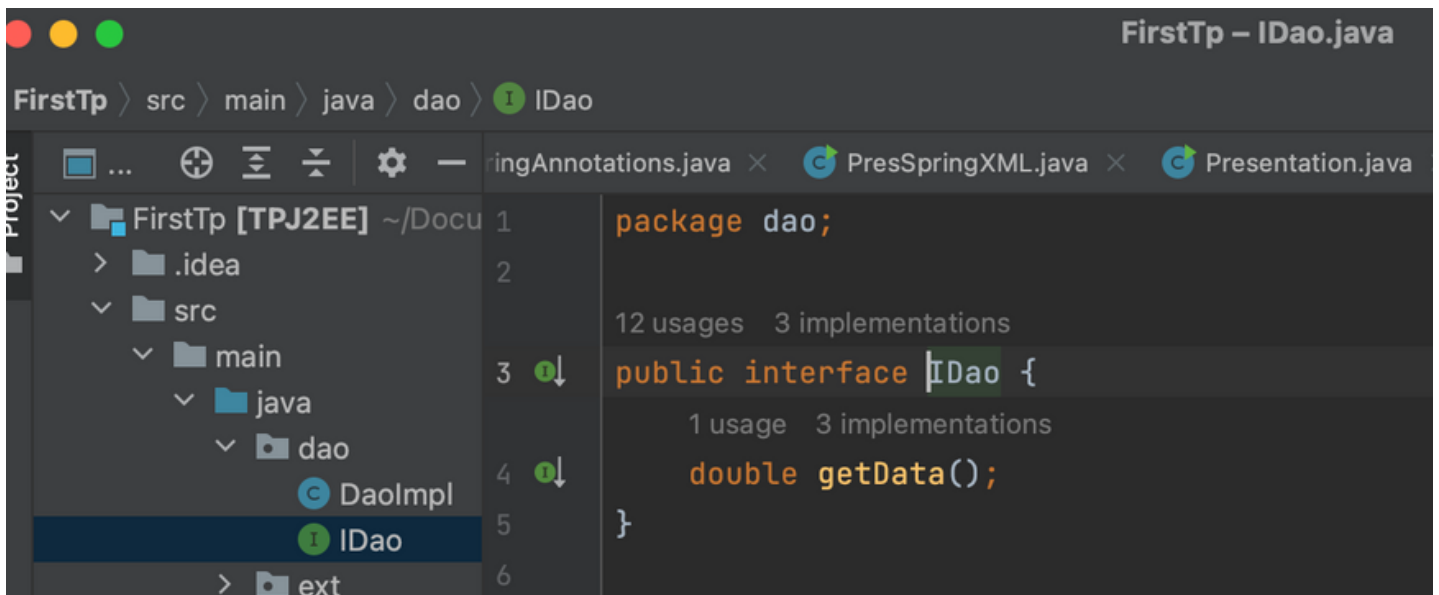
Introduction

Dans un projet, il est important de respecter les exigences fonctionnelles et techniques. Pour appliquer le principe d'inversion de contrôle, il est recommandé d'utiliser un framework(Spring) qui se chargera de la partie technique, tandis que la partie métier sera laissée au développeur.

Pour qu'une application soit facile à maintenir, elle doit être fermée à la modification et ouverte à l'extension. Cela peut être réalisé en utilisant le couplage faible, par exemple en utilisant des interfaces.

Dans ce TP, nous allons voir comment réaliser une application fermée à la modification et ouverte à l'extension en utilisant le couplage faible avec des interfaces.

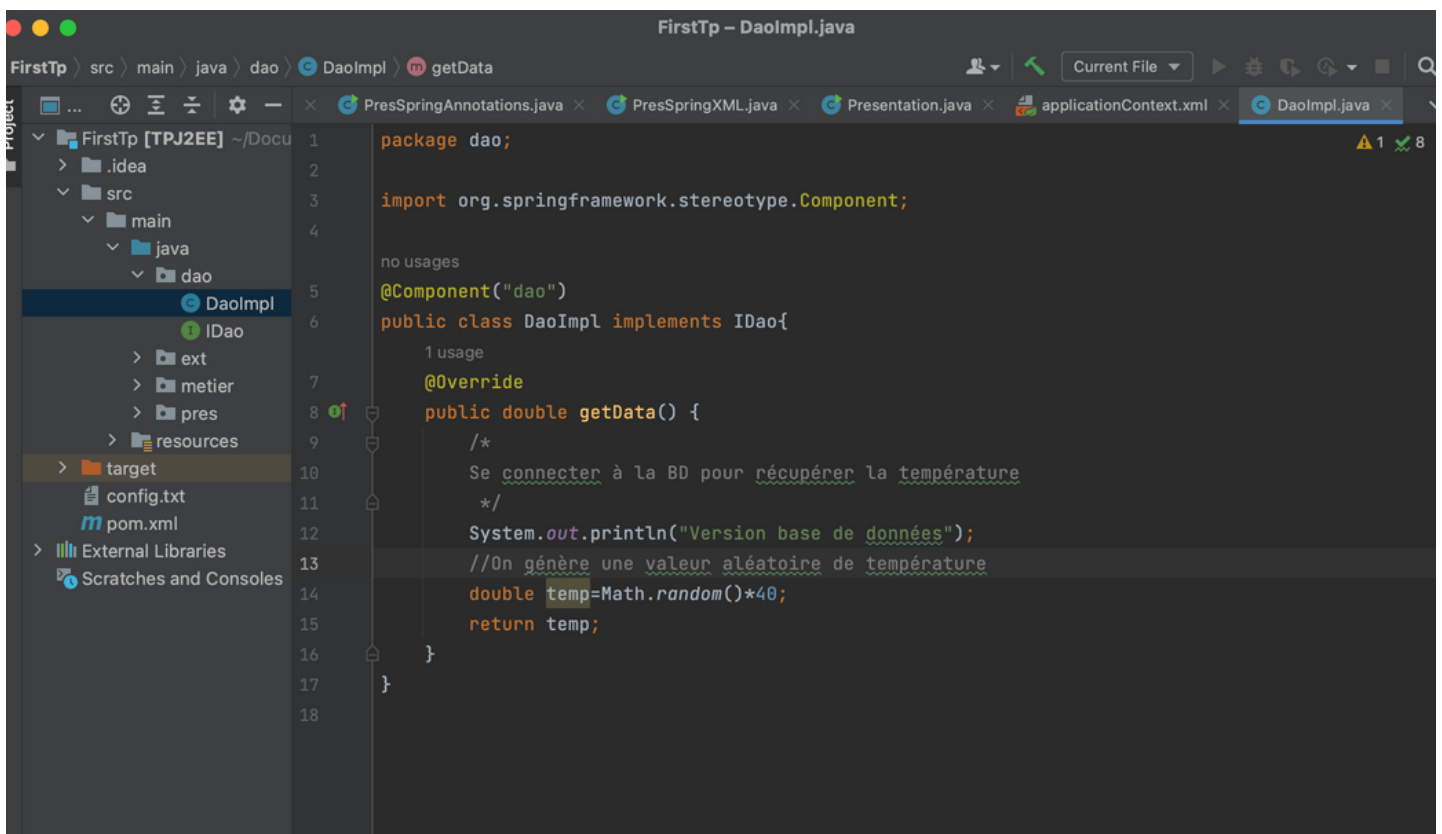
couche dao



```
FirstTp – IDao.java
FirstTp > src > main > java > dao > IDao

package dao;

12 usages 3 implementations
public interface IDao {
    1 usage 3 implementations
    double getData();
}
```



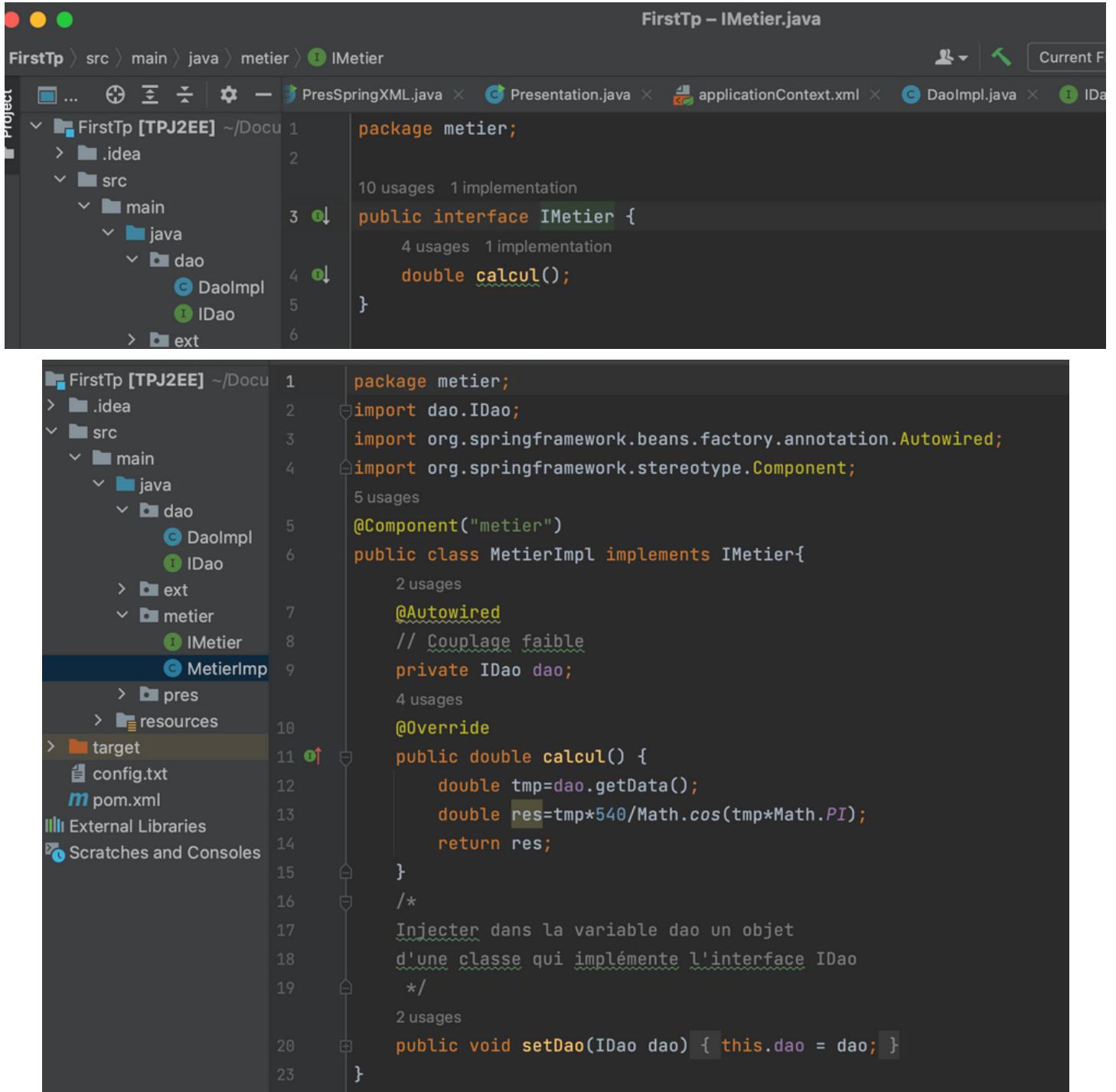
```
FirstTp – DaoImpl.java
FirstTp > src > main > java > dao > DaoImpl > getData

package dao;

import org.springframework.stereotype.Component;

no usages
@Component("dao")
public class DaoImpl implements IDao{
    1 usage
    @Override
    public double getData() {
        /*
        Se connecter à la BD pour récupérer la température
        */
        System.out.println("Version base de données");
        //On génère une valeur aléatoire de température
        double temp=Math.random()*40;
        return temp;
    }
}
```

Dans la couche DAO on a créé une interface iDao qui contient getdata qu'on va l'implémenter dans la class DaoImpl c'est pour ce connecter à la base de données puis on a géré une valeur aléatoire de la variable temp.



```

FirstTp - IMetier.java
FirstTp > src > main > java > metier > IMetier
PresSpringXML.java x Presentation.java x applicationContext.xml x DaoImpl.java x IDao

1 package metier;
2
3 10 usages 1 implementation
4 public interface IMetier {
5     4 usages 1 implementation
6     double calcul();
7 }

FirstTp [TPJ2EE] ~/Docu
> .idea
> src
  > main
    > java
      > dao
        DaoImpl
        IDao
      > ext
    > metier
      IMetier
      MetierImpl
    > pres
    > resources
  > target
    config.txt
    pom.xml
  External Libraries
  Scratches and Consoles

1 package metier;
2 import dao.IDao;
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Component;
5 5 usages
6 @Component("metier")
7 public class MetierImpl implements IMetier{
8     2 usages
9     @Autowired
10    // Couplage faible
11    private IDao dao;
12    4 usages
13    @Override
14    public double calcul() {
15        double tmp=dao.getData();
16        double res=tmp*540/Math.cos(tmp*Math.PI);
17        return res;
18    }
19    /*
20    Injecter dans la variable dao un objet
21    d'une classe qui implémente l'interface IDao
22    */
23    2 usages
24    public void setDao(IDao dao) { this.dao = dao; }
25 }
  
```

Dans la couche DAO nous avons créé une interface iMetier qui contient la méthode calcul() que nous allons implémenter dans la classe MetierImpl pour faire le calcul nous devons appeler getdata() depuis la couche dao et ne pas dépendre de la classe dont nous dépendrons L'interface. pour l'injection de dépendances, nous avons ajouté un setter car dao est nul

couche presentation

```

main > java > pres > Presentation > main
Presentation.java x applicationContext.xml x DaoImpl.java x IDao.java x

1 package pres;
2
3 import ...
4
5 no usages
6 public class Presentation {
7     no usages
8     public static void main(String[] args) {
9         /*l'injection des dependances par
10         l'instanciation static (new=>couplage fort)
11         */
12         DaoImpl2 dao= new DaoImpl2();
13         //DaoImpl dao=new DaoImpl();
14         MetierImpl metier=new MetierImpl();
15         //l'injection des dependances
16         metier.setDao(dao);
17         System.out.println("Résultat="+metier.calcul());
18     }
19 }

```

```

Pres2.java x PresSpringAnnotations.java x PresSpringXML.java x Presentation.java x applicationContext.xml
TPJ2EE] ~/Document

3 import ...
4
5 no usages
6 public class Pres2 {
7     //FileNotFoundException, ClassNotFoundException, InstantiationException, IL
8     no usages
9     public static void main(String[] args) throws Exception{
10         /*l'injection des dependances par
11         l'instanciation dynamique
12         */
13         Scanner scanner=new Scanner(new File( pathname: "config.txt"));
14         //il faut connaitre le nom de la class
15         String daoClassName=scanner.nextLine();
16         //charger la classe en memoire
17         Class cDao=Class.forName(daoClassName);
18         //New instance c'est un objet de type object ici exactemet de type IDao
19         IDao dao=(IDao) cDao.newInstance();
20
21         String metierClassName=scanner.nextLine();
22         Class cMetier=Class.forName(metierClassName);
23         IMetier metier=(IMetier) cMetier.newInstance();
24
25         Method method=cMetier.getMethod( name: "setDao", IDao.class);
26         // metier.setDao(dao)=>static invoquer dao sur lobjet metier
27         //la 2eme methode invoke=>dynamique -l'injection des dependances dynami
28         method.invoke(metier,dao);
29
30         System.out.println("Résultat="+ metier.calcul());
31     }
32 }

```

```

1 package pres;
2
3 import ...
4
5 no usages
6
7 public class PresSpringXML {
8
9     no usages
10
11     public static void main(String[] args) throws BeansException {
12         ApplicationContext context= new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
13         IMetier metier= (IMetier) context.getBean( s: "metier");
14         System.out.println("Résultat=>" + metier.calcul());
15     }
16 }
17

```

```

1 package pres;
2
3 import ...
4
5 no usages
6
7 public class PresSpringAnnotations {
8
9     no usages
10
11     public static void main(String[] args) {
12         ApplicationContext context= new AnnotationConfigApplicationContext( ...basePackages: "dao","metier");
13         IMetier metier= context.getBean( "metier");
14         System.out.println("Résultat=>" + metier.calcul());
15     }
16 }
17

```

Dans la couche de présentation, nous avons fait 4 méthodes d'injection de dépendances la première est l'instanciation statique avec new (le couplage fort), la 2ème doit utiliser l'instanciation dynamique pour rendre l'application facile à maintenir dans cette méthode nous avons besoin d'un fichier txt qui sera lu en premier puis lira la première ligne du fichier puis il trouvera le nom de la classe qui se chargera puis il créera une instance de cette classe

Dans la couche de présentation, nous avons fait 4 méthodes d'injection de dépendances la première est l'instanciation statique avec new (le couplage fort), la 2ème doit utiliser l'instanciation dynamique pour rendre l'application facile à maintenir dans cette méthode nous avons besoin d'un fichier txt qui sera lu en premier puis lira la première ligne du fichier puis il trouvera le nom de la classe qui se chargera puis il créera une instance de cette classe

après il lira la deuxième ligne et l'instanciera puis on créera son objet de méthode et spécifiera son nom et nous appellerons la méthode d'invocation, la 3ème méthode on crée un objet de type contexte d'application et on lui donne le fichier xml que l'on a créé avec les beans puis on lui demande un objet de type Imetier, et la dernière méthode avec les annotations est comme la méthode précédente sauf qu'au lieu de écrire "ClassPathXmlApplicationContext" écrit : "AnnotationConfigApplicationContext" et on doit fournir les Packages qui doivent être scannés