

Principe de l'inversion de contrôle et injection des dépendances

Objetifs du TP.....	2
Introduction.....	2
1. Conception.....	3
2. Code.....	4
a. Par instanciation statique.....	6
b. Par instanciation dynamique.....	6
c. En utilisant le Framework Spring.....	8
- Version XML.....	9
- Version annotations.....	10
Conclusion.....	13

Objectifs du TP

1. Créer l'interface IDao
2. Créer une implémentation de cette interface
3. Créer l'interface IMetier
4. Créer une implémentation de cette interface en utilisant le couplage faible
5. Faire l'injection des dépendances :
 - a. Par instanciation statique
 - b. Par instanciation dynamique
 - c. En utilisant le Framework Spring
 - Version XML
 - Version annotations

Introduction

La conception d'une application repend à l'un des principes suivants :

- Une application qui n'évolue pas meurt.
- Une application doit être fermée à la modification et ouverte à l'extension.
- Une application doit s'adapter aux changements
- ...

Les bonnes pratiques de l'ingénierie logicielle reposent sur la construction d'une application facile à maintenir, autrement dit avoir une application fermée à la modification et ouverte à l'extension. Les questions suivantes se posent :

- Comment Créer une application fermée à la modification et ouverte à l'extension ?
- Comment faire l'injection des dépendances ?
- C'est quoi le principe de l'inversion de contrôle ?

C'est quoi une application fermée à la modification et ouverte à l'extension ?

Cela veut dire qu'ajouter de nouvelles fonctionnalités ne devrait pas casser les fonctionnalités déjà existantes. Cela veut dire aussi qu'on ne touche pas le code source déjà écrit, on y ajoute seulement. Une application ouverte à la modification doit se baser sur une bonne conception qui profite du polymorphisme pour établir un couplage faible entre les différentes entités, cela facilite par la suite l'extension de l'application et l'ajout ou la modification des fonctionnalités. Pour se faire, il faut comprendre la différence entre le couplage faible et le couplage fort.

Couplage fort

Quand une classe A est liée à une classe B, on dit que la classe A est fortement couplée à la classe B.

La classe A ne peut fonctionner qu'en présence de la classe B.

Si une nouvelle version de la classe B (soit B2), est créé, on est obligé de modifier dans la classe A.

Modifier une classe implique :

- Il faut disposer du code source.
- Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
- Ce qui engendre un cauchemar au niveau de la maintenance de l'application

Couplage faible

- Pour utiliser le couplage faible, nous devons utiliser les interfaces.
- Considérons une classe A qui implémente une interface IA, et une classe B qui implémente une interface IB.
- Si la classe A est liée à l'interface IB par une association, on dit que le classe A et la classe B sont liées par un couplage faible.
- Cela signifie que la classe B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.
- En effet la classe B ne connaît que l'interface IA. De ce fait n'importe quelle classe implémentant cette interface peut être associée à la classe B, sans qu'il soit nécessaire de modifier quoi que ce soit dans la classe B.
- Avec le couplage faible, nous pourrons créer des applications fermées à la modification et ouvertes à l'extension.

Pour illustrer tous les concepts dans de rapport on va travailler sur un projet Maven dans laquelle on va implémenter une application qui nous retourne le degré de température selon déférente source de l'information.

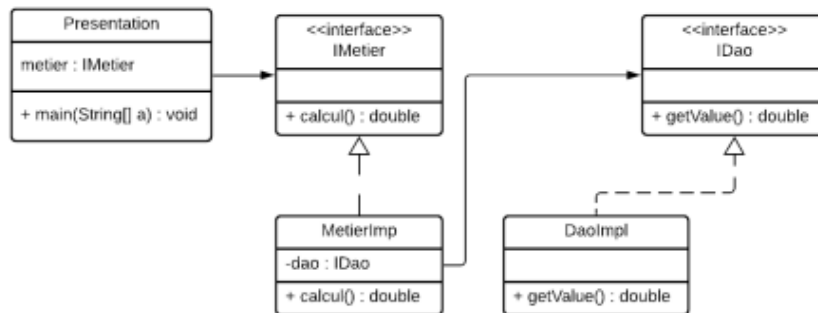
Maven est un outil de construction de projets (build) open source développée par la fondation Apache, initialement pour les besoins du projet Jakarta Turbine. Il permet de faciliter et d'automatiser certaines tâches de la gestion d'un projet Java.

1. Conception

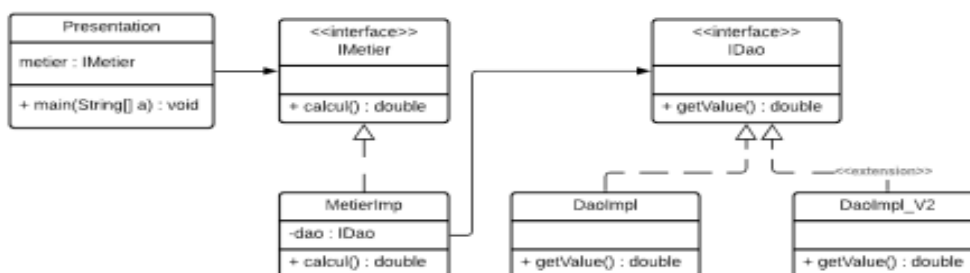
Le module ci-dessous ulliustre les notions mentionnées dans l'introduction, alors maintenant si on veut modifier la méthode getValue() il suffit d'ajouter une nouvelle classe qui dépend de l'interface <<IDao>> et redéfinir la méthode getValue().

Aucun problème sera rencontré puisque la classe MetierImpl peut fonctionner avec n'importe quelle classe qui implémente l'interface <<IDao>>.

Voici comment si facile détendre les fonctionnalités de notre application sans toucher le code déjà écrit.



Dans ce cas on peut facilement utiliser l'un des deux implémentations de Dao et on peut ajouter d'autres versions d'implémentations sans modifier le code source initial de l'application mais seulement avec l'ajout des extensions.



2. Code

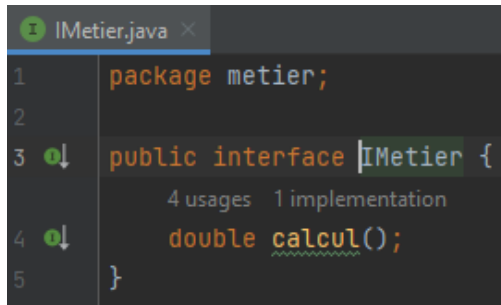
La création de l'interface IDao qui contient la méthode getData() et une implémentation de cette interface.

```
IDao.java
1 package dao;
2
3 public interface IDao {
4     double getData();
5 }

DaoImpl.java
1 package dao;
2
3 public class DaoImpl implements IDao{
4     @Override
5     public double getData() {
6         /*
7          * Se connecter a la bdd pour recuperer la temperature
8          *
9          */
10        System.out.println("version BDD");
11        double temp=Math.random()*40;
12        return temp;
13    }
14 }
```

DaoImpl est la première implémentation de l'interface IDao on va la nommée version DB (base de données) car il nous permet de récupérer la température à partir d'une base de données.

La création de l'interface IMetier qui contient la méthode calcul() et une implémentation de cette interface.



```
1 package metier;
2
3 public interface IMetier {
4     double calcul();
5 }
```



```
1 package metier;
2
3 import ...
4
5 5 usages
6
7 public class MetierImpl implements IMetier {
8     //couplage faible
9     3 usages
10    private IDao dao;
11
12    2 usages
13    public MetierImpl(IDao dao) { this.dao = dao; }
14
15    4 usages
16    @Override
17    public double calcul() {
18        double tmp=dao.getData();
19        double res=tmp*7658*Math.sin(tmp/Math.PI);
20        return res;
21    }
22    /*
23     injecter dans la variable dao un objet
24     d'une classe qui implemente l'interface IDao
25     */
26    public void setDao(IDao dao) { this.dao = dao; }
27 }
```

La méthode calcul() récupère les données de la méthode getData() et avec une formule il nous permet d'avoir le résultat qui est la température.

Inversion de contrôle

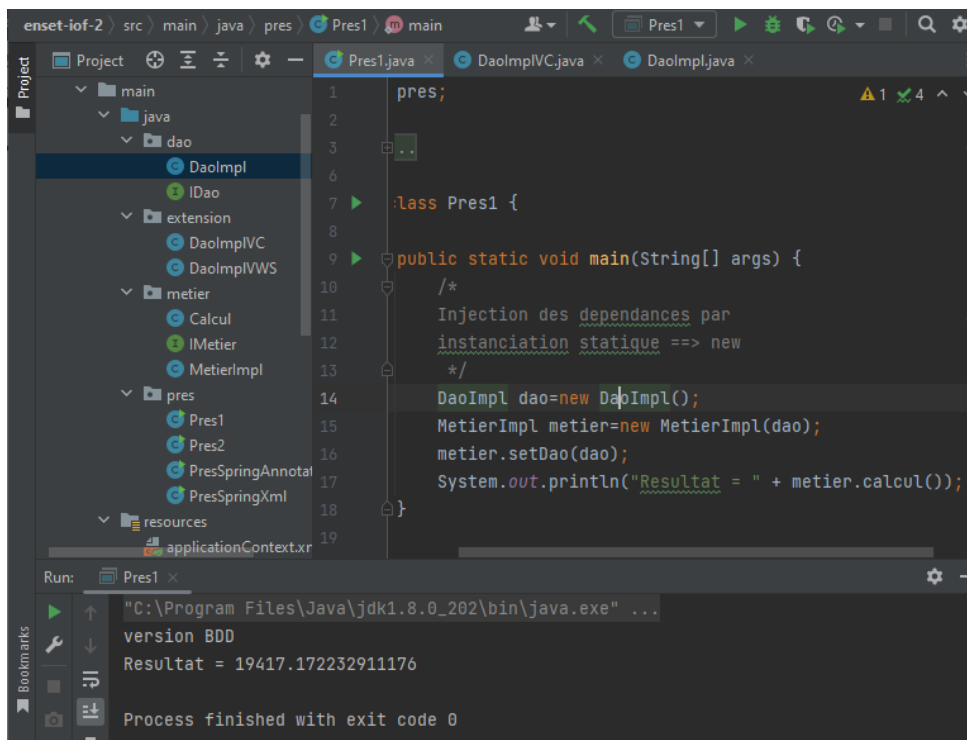
Il s'agit d'un mécanisme qui facilite la mise en place des dépendances par l'injection automatique des objets. Il permet au développeur de se concentrer sur le code métier et le Framework va s'occuper du code technique.

Injection des dépendances

L'injection de dépendances est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle. Il consiste à créer dynamiquement les dépendances entre les différents objets en s'appuyant sur une description ou de manière programmatique

a. Par instantiation statique

Voilà le code de première méthode de l'injection des dépendances (méthode statique) en utilisant la mot clé **new**, et le résultat de la version base de données.



```

1  pres;
2
3  ..
4
5
6
7  class Pres1 {
8
9  public static void main(String[] args) {
10     /*
11      Injection des dependances par
12      instantiation statique ==> new
13      */
14     DaoImpl dao=new DaoImpl();
15     MetierImpl metier=new MetierImpl(dao);
16     metier.setDao(dao);
17     System.out.println("Resultat = " + metier.calcul());
18 }
19

```

Run: Pres1 x

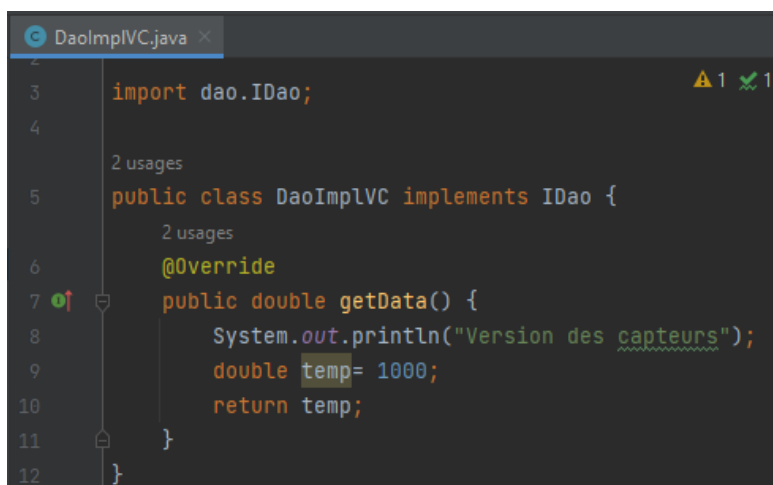
```

"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
version BDD
Resultat = 19417.172232911176
Process finished with exit code 0

```

b. Par instantiation dynamique

Maintenant on veut récupérer la température en utilisant des capteurs et non pas les données de base de données donc on doit créer une nouvelle package extension qui contient la nouvelle méthode de calcul cette nouvelle class qui doit implémenter l'interface IDao on va la nommée DaoImplVC (Dao implémentation version capteur).



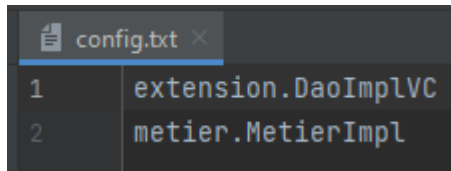
```

1  DaoImplVC.java
2
3  import dao.IDao;
4
5  2 usages
6  public class DaoImplVC implements IDao {
7
8  2 usages
9  @Override
10 public double getData() {
11     System.out.println("Version des capteurs");
12     double temp= 1000;
13     return temp;
14 }
15 }

```

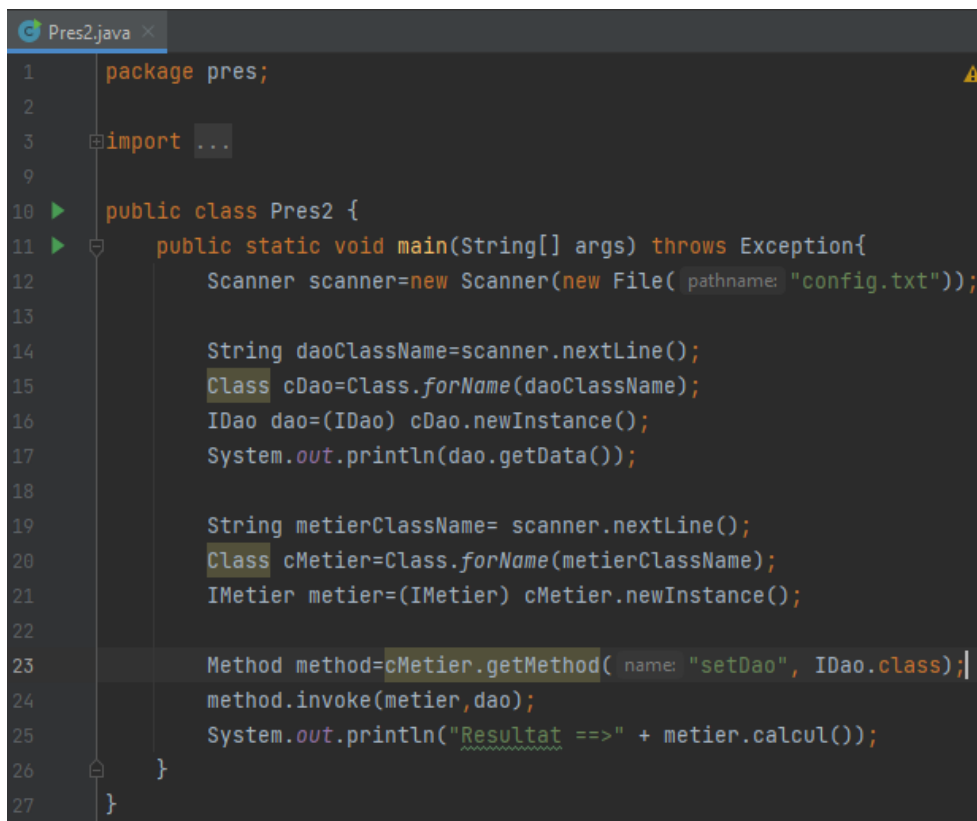
Est-ce que on peut utiliser la nouvelle version sans modifier le code source de notre application la réponse est non donc pour résoudre de problème on va utiliser **instanciation dynamique**

La première action est de créer un fichier de configuration qui contient les noms de classes qu'on veut injecter.

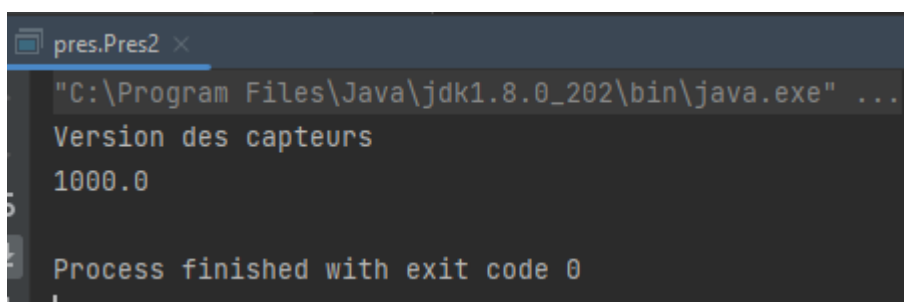


```
config.txt
1 extension.DaoImplVC
2 metier.MetierImpl
```

Et après dans la présentation on va lire ce fichier et instancier ces class de manier dynamique.




```
Pres2.java
1 package pres;
2
3 import ...
4
5
6
7
8
9
10 public class Pres2 {
11     public static void main(String[] args) throws Exception{
12         Scanner scanner=new Scanner(new File( pathname: "config.txt"));
13
14         String daoClassName=scanner.nextLine();
15         Class cDao=Class.forName(daoClassName);
16         IDao dao=(IDao) cDao.newInstance();
17         System.out.println(dao.getData());
18
19         String metierClassName= scanner.nextLine();
20         Class cMetier=Class.forName(metierClassName);
21         IMetier metier=(IMetier) cMetier.newInstance();
22
23         Method method=cMetier.getMethod( name: "setDao", IDao.class);
24         method.invoke(metier,dao);
25         System.out.println("Resultat ==>" + metier.calcul());
26     }
27 }
```



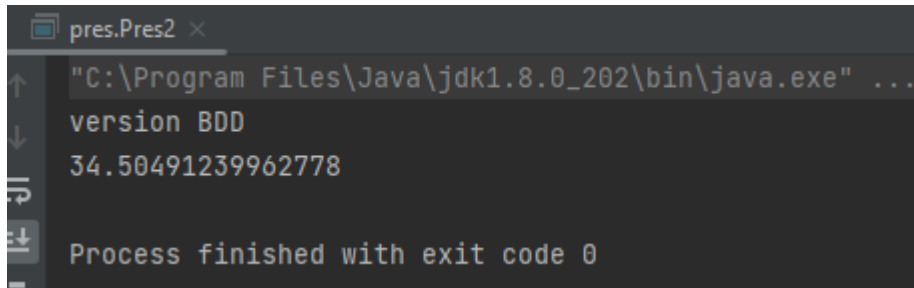
```
pres.Pres2
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
Version des capteurs
1000.0
Process finished with exit code 0
```

Est-ce qu'on peut revenir à la première version de la base de données sans modifier le code source.

La réponse est oui. Avec l'instanciation dynamique on peut facilement revenir à la première version il faut seulement modifier le contenu du fichier config.txt et changer la ligne qui contient le non de la extension.DaoImplVC par dao.DaoImpl



```
config.txt x
1 dao.DaoImpl
2 metier.MetierImpl
```



```
pres.Pres2 x
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
version BDD
34.50491239962778
Process finished with exit code 0
```

c. En utilisant le Framework Spring

Donc jusqu'à maintenant on est arrivé à créer une application qui est fermée à la modification et ouverte à l'extension mais il existe des Framework qui nous permettent de faciliter ce travail. C'est le principe de l'Inversion de contrôle : ce dernier 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). Dans notre cas on va utiliser Spring.

Spring

Spring est un Framework qui facilite la maîtrise du cycle de vie de l'application grâce à de nombreuses bonnes pratiques de développement. Le concept fondamental du Spring est l'injection des dépendances (dependency injection). C'est une réponse à 3 facteurs du mauvais codage, définis en 1994 par Robert C. Martin :

Rigidité : il est difficile de changer quoi que ce soit car chaque changement affecte trop le fonctionnement du système.

Fragilité : un changement provoque le mauvais fonctionnement des parties qui devraient fonctionner correctement.

Immobilité : il est difficile de réutiliser le code dans une autre application.

Ces 3 problèmes sont appelés RFI. L'Injection des dépendances est une solution à cette problématique.

Sous Spring, les objets s'appellent des beans. L'injection des dépendances automatique peut se faire à travers les fichiers XML aussi bien qu'à travers des annotations. Le rôle du container consiste donc à regrouper les beans avec le modèle des données. Ce regroupement produit ensuite le système prêt à être déployé. Le container est donc un contexte d'application (application context) qui gère des beans.

D'abord pour travailler avec il faut l'ajouter comme des dépendances dans le fichier pom.xml. Il faut 3 dépendances qui sont : spring-core, spring-context et spring-beans.


```

pom.xml (enset-ic-2) x
16 <!-- https://mvnrepository.com/artifact/org.springframework/spring-core -->
17 <dependency>
18 <groupId>org.springframework</groupId>
19 <artifactId>spring-core</artifactId>
20 <version>5.3.16</version>
21 </dependency>
22 <!-- https://mvnrepository.com/artifact/org.springframework/spring-context -->
23 <dependency>
24 <groupId>org.springframework</groupId>
25 <artifactId>spring-context</artifactId>
26 <version>5.3.16</version>
27 </dependency>
28
29 <!-- https://mvnrepository.com/artifact/org.springframework/spring-beans -->
30 <dependency>
31 <groupId>org.springframework</groupId>
32 <artifactId>spring-beans</artifactId>
33 <version>5.3.16</version>
34 </dependency>

```

- Version XML

Spring lit le fichier spring-ioc.xml et à base de lui il crée les nouveaux beans et injecter les dépendances entre eux. Pour utiliser l'injection des dépendances avec Spring version xml il faut créer un fichier xml nommée ApplicationContext.xml

```

applicationContext.xml x
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4     xsi:schemaLocation="http://www.springframework.org/schema/b
5     <bean id="dao" class="dao.DaoImpl"></bean>
6     <bean id="metier" class="metier.MetierImpl">
7         <property name="dao" ref="dao"></property>
8         <constructor-arg ref="dao"></constructor-arg>
9     </bean>
10
11 </beans>

```

Maintenant on va créer une présentation pour la version xml.

```

PresSpringXml.java x
1 package pres;
2
3 import ...
4
5
6
7
8
9 public class PresSpringXml {
10     public static void main(String[] args) {
11         ApplicationContext context=
12             new ClassPathXmlApplicationContext("applicationContext.xml");
13         IMetier metier=(IMetier) context.getBean("metier");
14         System.out.println("Resultat==> "+metier.calcul());
15     }
16 }

```

pres.PresSpringXml x

```

"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
version BDD
Resultat==>42641.01908415925
Process finished with exit code 0

```

on peut facilement récupérer le bean désiré depuis le contexte de l'application via la méthode `getBean()`

Pour utiliser l'autre version de capteurs il faut seulement modifier le fichier `ApplicationContext.xml`.

The image shows two screenshots from an IDE. The top screenshot displays the `applicationContext.xml` file. It contains an XML configuration for two beans. The first bean, with id `dao`, is of class `extension.DaoImplVC` and is highlighted with a red box. The second bean, with id `metier`, is of class `metier.MetierImpl` and has a property `dao` that references the `dao` bean. The bottom screenshot shows the `PresSpringXml.java` file. It contains a `main` method that creates an `ApplicationContext` using `ClassPathXmlApplicationContext`, retrieves the `metier` bean, and prints the result of `metier.calcul()`. The IDE's output window at the bottom shows the execution results: `Version des capteurs` and `Resultat==>-6481076.955989769`.

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

```
package pres;

import ...

public class PresSpringXml {
    public static void main(String[] args) {
        ApplicationContext context=
            new ClassPathXmlApplicationContext("applicationContext.xml");
        IMetier metier=(IMetier) context.getBean("metier");
        System.out.println("Resultat==>" + metier.calcul());
    }
}
```

```
Run: pres.PresSpringXml
"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...
Version des capteurs
Resultat==>-6481076.955989769
Process finished with exit code 0
```

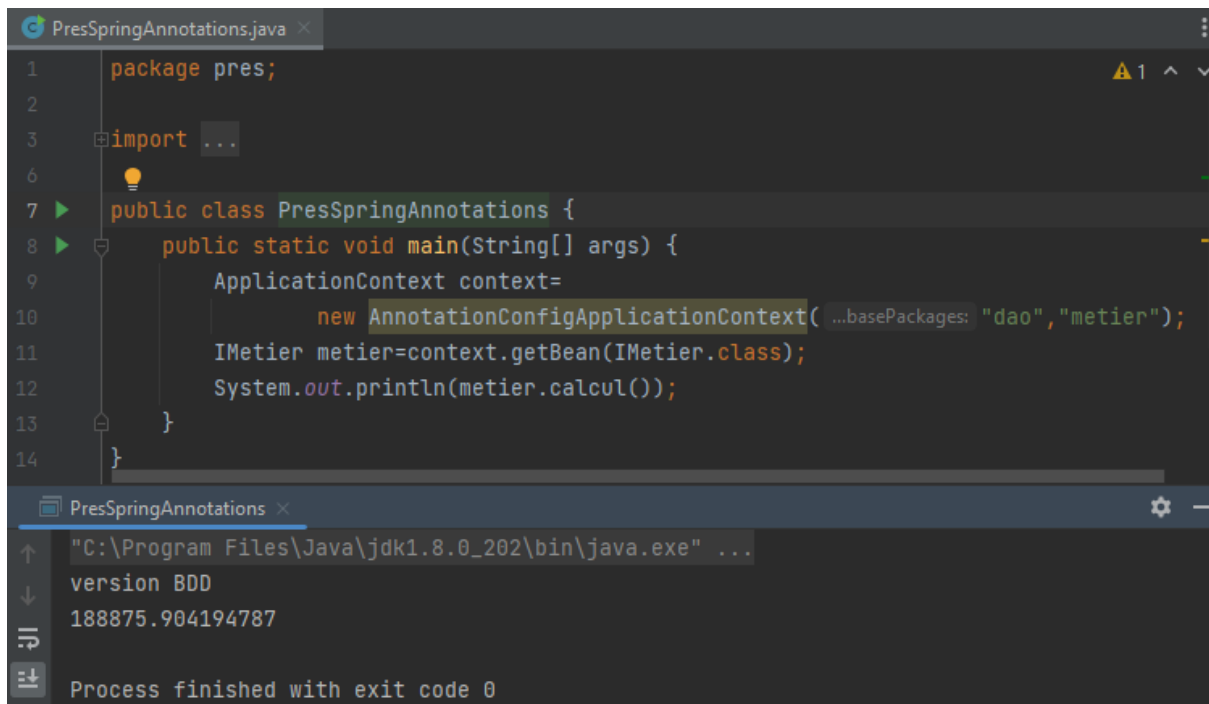
- Version annotations

Pour utiliser l'injection des dépendances avec Spring version annotation on doit respecter quelques règles : Ajouter l'annotation `@Component("nom")` à chaque class qu'on veut instancier. Le nom est facultatif si tu ne donnes pas un nome Spring utilise le nom de la classe Pour faire l'injection des dépendances on utilise l'annotation `@Autowired`.

```
MetierImpl.java x
5 usages
7 @Component
8 public class MetierImpl implements IMetier {
9     //couplage faible
10    3 usages
11    @Autowired
12    private IDao dao;
13
14    2 usages
15    public MetierImpl(IDao dao) { this.dao = dao; }
16
17    3 usages
18    @Override
19    public double calcul() {
20        double tmp=dao.getData();
21        double res=tmp*7658*Math.sin(tmp/Math.PI);
22        return res;
23    }
24    /*
25     injecter dans la variable dao un objet
26     d'une classe qui implemente l'interface IDao
27     */
28    public void setDao(IDao dao) { this.dao = dao; }
29 }
30
```

```
DaoImpl.java x
1 package dao;
2
3 import org.springframework.stereotype.Component;
4
5 4 usages
6 @Component("metier")
7 public class DaoImpl implements IDao{
8     2 usages
9     @Override
10    public double getData() {
11        /*
12         * Se connecter a la bdd pour recuperer la temperature
13         *
14         */
15        System.out.println("version BDD");
16        double temp=Math.random()*40;
17        return temp;
18    }
19 }
20
```

Présentation pour la version annotation



```
1 package pres;
2
3 import ...
4
5
6
7 public class PresSpringAnnotations {
8     public static void main(String[] args) {
9         ApplicationContext context=
10             new AnnotationConfigApplicationContext(...basePackages: "dao","metier");
11         IMetier metier=context.getBean(IMetier.class);
12         System.out.println(metier.calcul());
13     }
14 }
```

PresSpringAnnotations

"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...

version BDD

188875.904194787

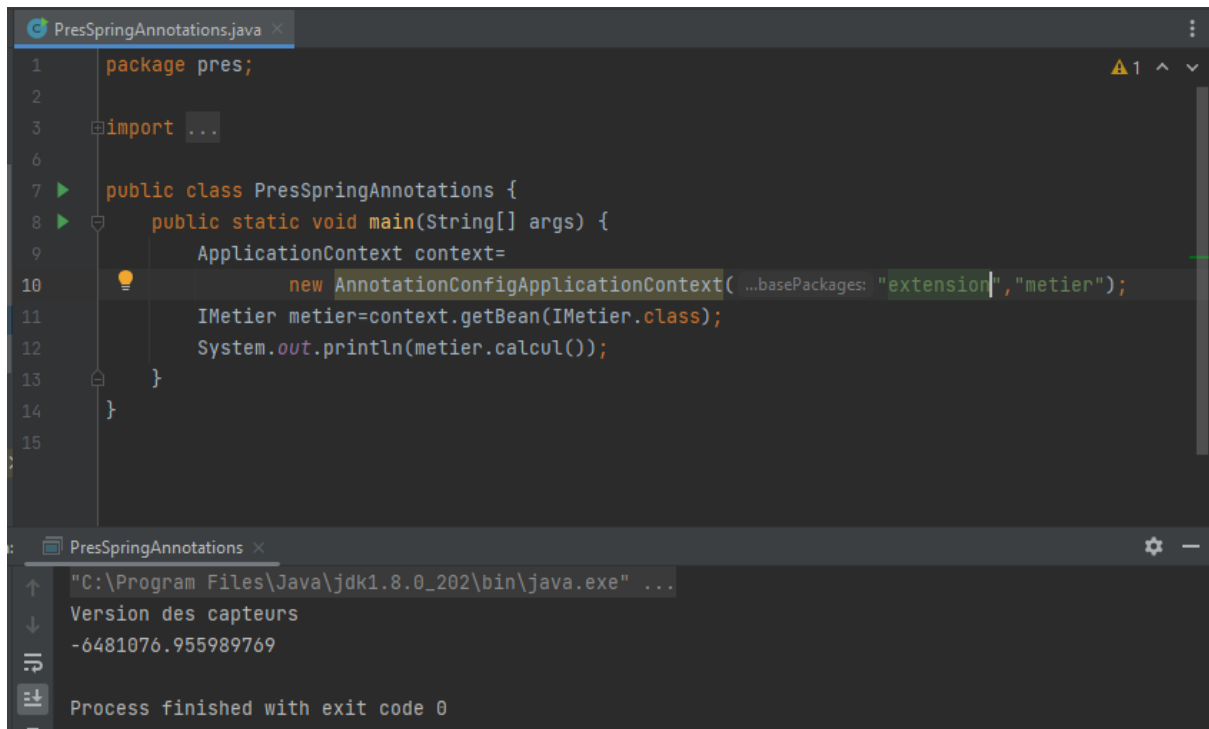
Process finished with exit code 0

Il faut mentionner à spring les packages qui va scanner Comme c'est le cas avec Spring XML, on peut facilement récupérer le bean désiré depuis le contexte de l'application via la méthode `getBean()`

Pour utiliser l'autre version il faut seulement ajouter `@Component` à la nouvelle implémentation que tu veux ajouter au projet et aussi mettre le nom de son package dans `AnnotationConfigApplicationContext` pour permettre à Spring de les scanner.



```
3 import dao.IDao;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class DaoImplVC implements IDao {
8     @Override
9     public double getData() {
10         System.out.println("Version des capteurs");
11         double temp= 1000;
12         return temp;
13     }
14 }
```



```
1 package pres;
2
3 import ...
4
5
6
7 public class PresSpringAnnotations {
8     public static void main(String[] args) {
9         ApplicationContext context=
10             new AnnotationConfigApplicationContext(...basePackages: "extension", "metier");
11         IMetier metier=context.getBean(IMetier.class);
12         System.out.println(metier.calcul());
13     }
14 }
15
```

PresSpringAnnotations

"C:\Program Files\Java\jdk1.8.0_202\bin\java.exe" ...

Version des capteurs
-6481076.955989769

Process finished with exit code 0

Conclusion

Dans l'ingénierie logicielle, une application doit être fermée à la modification et ouverte à l'extension. Dans ce sens, le principe de l'Inversion de Contrôle et Injection des dépendances vient pour assurer cette facilité de maintenabilité.