

# RAPPORT TP1

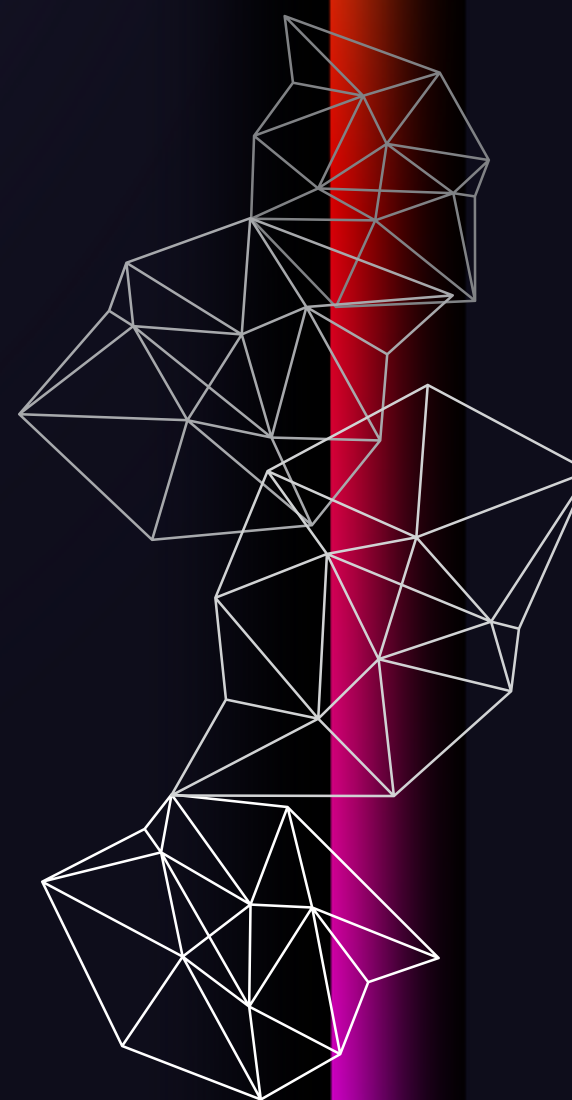
## PRINCIPE DE L'INVERSION DE CONTRÔLE ET INJECTION DES DÉPENDANCES

Réalisé par :

Bouchama Hajar

Demandé par :

Mohamed Youssfi



## Introduction

Dans le cadre de ce TP, j'ai été confronté à la nécessité de concevoir une architecture logicielle robuste et flexible, permettant de séparer les préoccupations et de favoriser la réutilisabilité du code. Pour répondre à ces exigences, j'ai adopté une approche orientée vers les interfaces et l'inversion de contrôle (IoC) comme indiqué dans les vidéos Youtube, facilitant ainsi la gestion des dépendances et la mise en œuvre du couplage faible.

Dans ce rapport, je vais détailler les différentes étapes de conception et d'implémentation de cette solution, en suivant les directives suivantes :

- Création de l'interface IDao avec une méthode getDate, qui représente un contrat définissant les opérations de manipulation des données.
- Développement d'une implémentation de cette interface, garantissant une séparation claire entre le contrat et son implémentation concrète.
- Élaboration de l'interface IMetier avec une méthode calcul, visant à encapsuler la logique métier de ce tp.
- Mise en place d'une implémentation de cette interface en utilisant le couplage faible, permettant ainsi une flexibilité accrue et une meilleure testabilité du code.
- Injection des dépendances à différents niveaux :
  1. Par **instanciation statique**, où les dépendances sont explicitement créées et injectées au moment de l'instanciation.
  2. Par **instanciation dynamique**, offrant une approche plus flexible en déléguant la gestion des dépendances à une entité externe, généralement un conteneur IoC.
  3. Utilisation du **framework Spring** pour gérer l'injection de dépendances, avec deux approches distinctes :
    - La version XML et la version annotation,

À travers ce rapport, je vais explorer chaque étape en détail, en mettant en évidence les avantages et les considérations associés à chaque approche, dans le but ultime de garantir une architecture logicielle robuste, modulaire et facilement évolutive.

## Enoncé !

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 :
  - a. Par instanciation statique
  - b. Par instanciation dynamique
  - c. En utilisant le Framework Spring
    - Version XML
    - Version annotations

Code et explications :

1. Créer l'interface IDao avec une méthode getDate

- J'ai défini une interface nommée IDao dans le package dao. Cette interface contient une méthode getData().

```
pl.java    © Pres2.java    © Presentation.java    © DaoImpl.java

package dao;

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

2. Créer une implémentation de cette interface

- Dans le code ci-dessous j'ai créé une implémentation de l'interface IDao dans la classe DaoImpl. La méthode getData() est implémentée pour simuler la récupération de données à partir d'une base de données. Dans cet exemple, une température aléatoire est générée pour illustrer le processus de récupération de données.

```
package dao;

1 usage
public class DaoImpl implements IDao {
    1 usage
    @Override
    public double getData(){
        double temperature = Math.random()*40;
        return temperature;
    }
}
```

3. Créer l'interface IMetier avec une méthode calcul

- j'ai défini une interface nommée IMetier dans le package metier. Cette interface contient une méthode calcul() qui définit le contrat pour exécuter des opérations de logique métier.

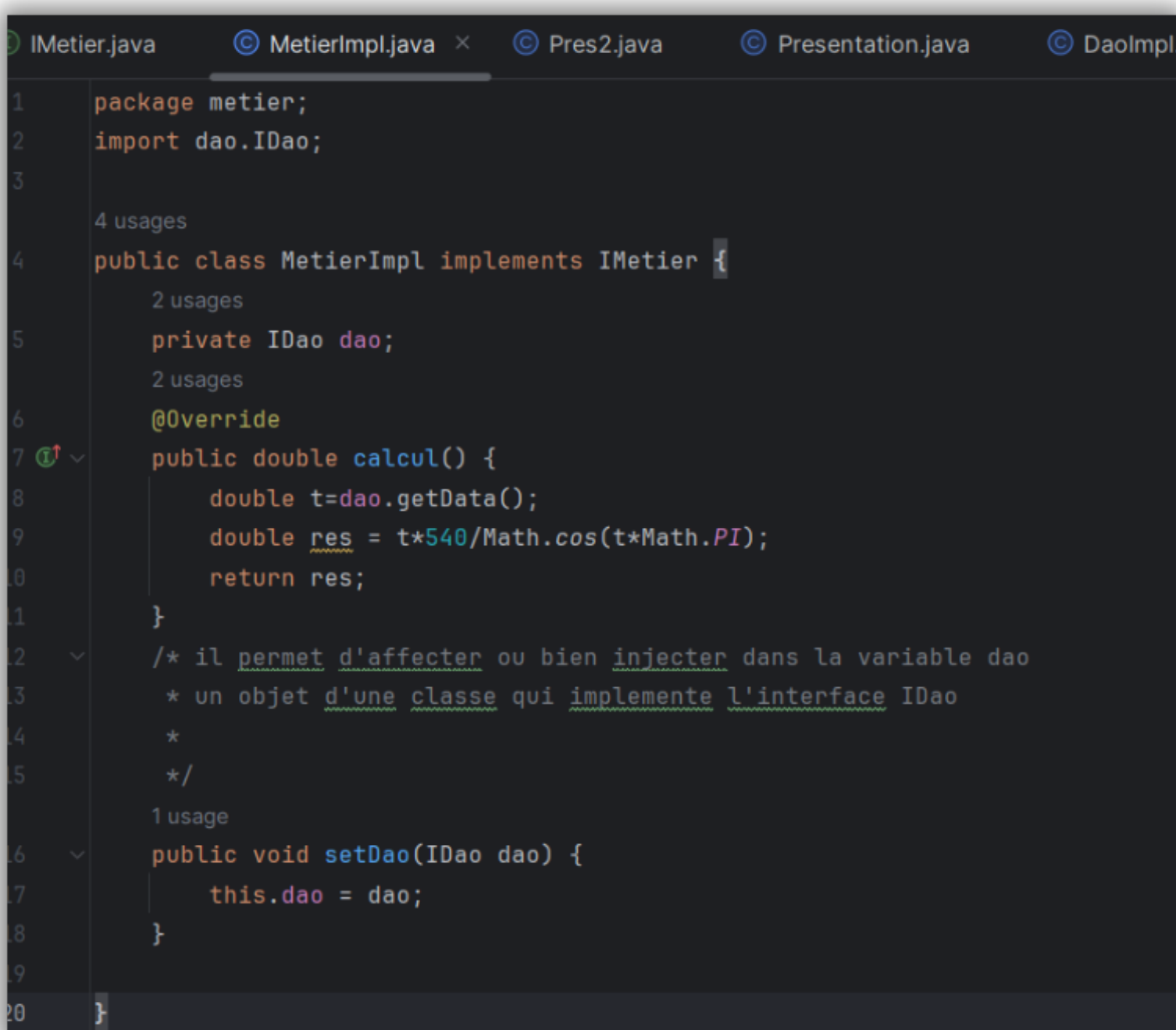
```
IMetier.java x    © MetierImpl.java    © Pres2.java    © Presentation.java    © DaoImpl.java

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

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

Le couplage faible est une approche où les interactions entre les classes sont basées sur des interfaces plutôt que des implémentations directes. Par exemple, si une classe A implémente une interface IA et une classe B implémente une interface IB, et que la classe A est liée à l'interface IB, on dit qu'il y a un couplage faible entre A et B. Cela permet à la classe B de fonctionner avec n'importe quelle classe implémentant l'interface IA, car elle n'a besoin de connaître que cette interface.

- Cette partie représente une implémentation de l'interface IMetier dans la classe MetierImpl. La méthode calcul() est implémentée pour effectuer un calcul en utilisant les données récupérées à partir d'un objet dao, la classe comporte également un constructeur prenant en paramètre une instance de IDao, permettant ainsi l'injection de dépendances lors de l'instanciation.
- La méthode setDao(IDao dao) est fournie pour permettre une injection de dépendances ultérieure.



```
1 package metier;
2 import dao.IDao;
3
4 4 usages
5 public class MetierImpl implements IMetier {
6     2 usages
7     private IDao dao;
8     2 usages
9     @Override
10    public double calcul() {
11        double t=dao.getData();
12        double res = t*540/Math.cos(t*Math.PI);
13        return res;
14    }
15    /* il permet d'affecter ou bien injecter dans la variable dao
16     * un objet d'une classe qui implémente l'interface IDao
17     *
18     */
19    1 usage
20    public void setDao(IDao dao) {
21        this.dao = dao;
22    }
23 }
```

Grâce à cette approche, la classe MetierImpl est faiblement couplée à l'implémentation de IDao. Elle n'est pas directement dépendante de l'implémentation concrète de IDao, ce qui rend le système plus flexible et permet de remplacer ou de modifier l'implémentation de IDao sans affecter MetierImpl. Cela favorise également la réutilisabilité et la testabilité du code.

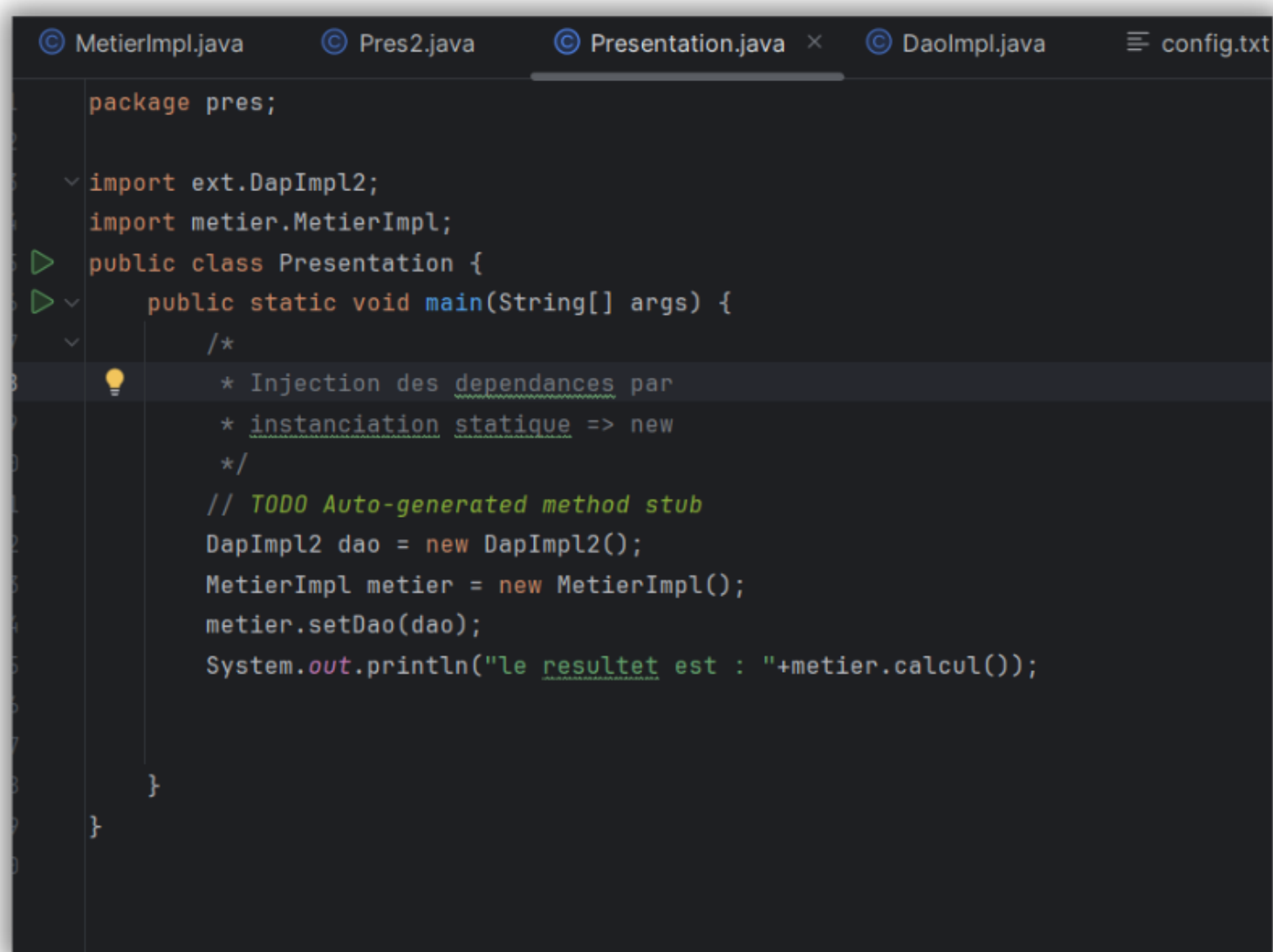
#### 5. Faire l'injection des dépendances :

##### a. Par instanciation statique

L'injection de dépendances par instanciation statique consiste à créer directement les instances des classes nécessaires dans le code source, souvent à l'intérieur des méthodes ou des constructeurs. Dans l'exemple que nous traitant, cette approche est illustrée en créant explicitement les instances des classes DaoImpl2 et MetierImpl à l'intérieur de la méthode main() en utilisant le mot-clé "new".



- Dans le contexte de l'exemple, l'objet dao de type DapImpl2 est créé en instanciant directement la classe DapImpl2. De même, l'objet metier de type MetierImpl est créé en instanciant directement la classe MetierImpl, en lui passant l'objet dao nouvellement créé comme dépendance lors de son instantiation.



```
MetierImpl.java  Pres2.java  Presentation.java  DaoImpl.java  config.txt

package pres;

import ext.DapImpl2;
import metier.MetierImpl;

public class Presentation {

    public static void main(String[] args) {

        /*
         * Injection des dependances par
         * instantiation statique => new
         */
        // TODO Auto-generated method stub
        DapImpl2 dao = new DapImpl2();
        MetierImpl metier = new MetierImpl();
        metier.setDao(dao);
        System.out.println("le resultat est : "+metier.calcul());

    }

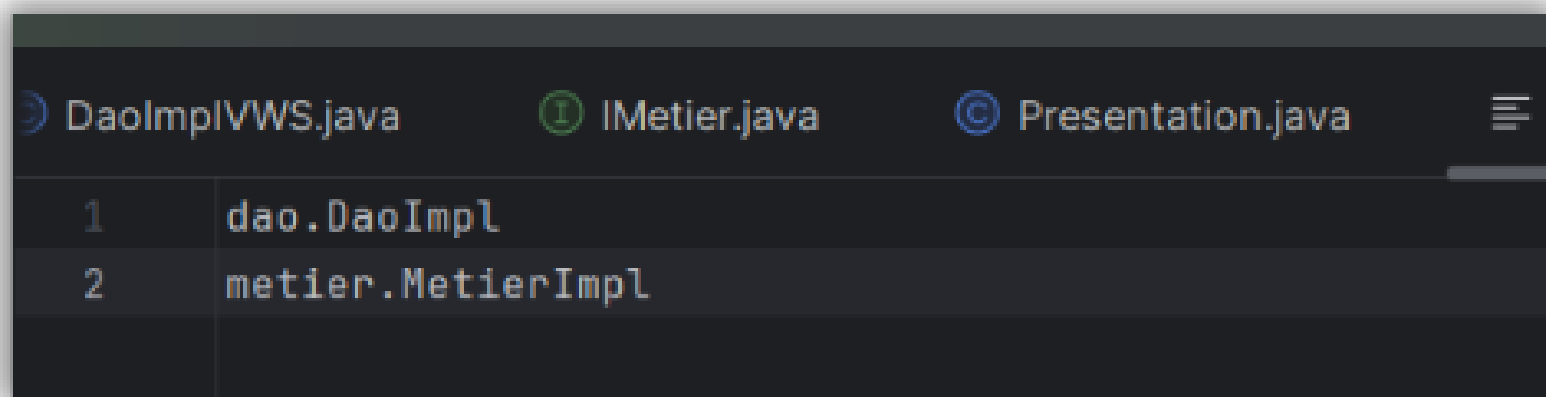
}
```

Cependant, cette approche présente quelques inconvénients. Elle rend le code plus rigide et moins modulaire, car les dépendances sont directement intégrées dans le code source, ce qui rend le code plus difficile à maintenir et à tester. De plus, cela rend le code plus difficile à réutiliser, car les dépendances sont fortement couplées avec les classes qui les utilisent.

**b. Par instantiation dynamique**

l'injection de dépendances est réalisée de manière dynamique en utilisant un fichier de configuration externe (config.txt)

- Chaque ligne de ce fichier représente le nom de la classe Java à utiliser pour une dépendance spécifique. Dans cet exemple, les lignes dao.DaoImpl et metier.MetierImpl sont fournies.



```
DaoImplVWS.java  IMetier.java  Presentation.java

1  dao.DaoImpl
2  metier.MetierImpl
```

- par exemple la premiere ligne contient :

**dao.DaoImpl:** elle indique que la classe à utiliser pour l'implémentation du DAO est DaoImpl située dans le package dao. Cela signifie que l'instanciation dynamique utilisera cette classe pour créer l'objet DAO.

```

1 package pres;
2 import dao.IDao;
3 import metier.IMetier;
4
5 import java.io.File;
6 import java.lang.reflect.Method;
7 import java.util.Scanner;
8
9 public class Pres2 {
10     public static void main(String[] args) throws Exception {
11
12         // TODO Auto-generated method stub
13         Scanner scan = new Scanner(new File("src/config.txt"));
14
15         String daoClassName = scan.nextLine();
16         Class cDao = Class.forName(daoClassName);
17         IDao dao = (IDao) cDao.newInstance();
18
19         String metierClassName = scan.nextLine();
20         Class cMetier = Class.forName(metierClassName);
21         IMetier metier = (IMetier) cMetier.newInstance();
22         // c'est spring qui va permettre de ne pas creer ses ligne il fait l'injection de dependances
23
24         Method method = cMetier.getMethod("setDao", IDao.class);
25         method.invoke(metier, dao);
26         System.out.println("Resultat = "+metier.calcul());
27     }
28 }

```

- j'ai utilisé la réflexion pour réaliser l'injection de dépendances dynamique en se basant sur les informations lues à partir du fichier config.txt. Il lit successivement les noms de classes pour le Dao et le métier à partir du fichier, charge dynamiquement ces classes, crée des instances de ces classes, injecte la dépendance dao dans l'objet métier et enfin, appelle une méthode sur l'objet métier pour obtenir un résultat du calcul.

### c. En utilisant le Framework Spring

#### - Version XML

La version XML de la configuration dans le framework Spring offre une approche claire et déclarative pour définir les beans et leurs dépendances. En utilisant des fichiers XML dédiés, on peut séparer la configuration de l'application de son code source, favorisant ainsi la modularité et la réutilisabilité.

- Le fichier applicationContext.xml est une composante essentielle dans le développement d'applications utilisant le framework Spring. Il s'agit d'un fichier de configuration XML dans lequel sont définis les différents éléments de l'application, tels que les beans, les dépendances et les configurations

```

<?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>
    </bean>
</beans>

```

```

aolImpl.java  MetierImpl.java  DaoImplVWS.java  IMetier.java  applicationContext.xml  PresSpringXM.java
<?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="ext.DaoImplVWS"></bean>
    <bean id="metier" class="metier.MetierImpl">
        <constructor-arg ref="dao"></constructor-arg>
    </bean>
</beans>

```

- pour cette partie la classe PresSpringXML fonctionne en utilisant le framework spring pour charger et exécuter une application basée sur une configuration XML. Lors de l'exécution, elle crée un contexte d'application en chargeant la configuration définie dans le fichier applicationContext.xml. Ensuite, elle récupère le bean nommé "metier" à partir du contexte Spring et l'utilise pour exécuter une opération spécifique, en l'occurrence la méthode calcul() de l'interface IMetier.

```
(TP1.Part2)  </> applicationContext.xml  © PresSpringXM.java  ×  © Pres2.java  © DaoImpl.java  © MetierImpl.java  © DaoImp

1  package pres;
2  import metier.IMetier;
3  import org.springframework.context.ApplicationContext;
4  import org.springframework.context.support.ClassPathXmlApplicationContext;
5
6  public class PresSpringXM {
7      public static void main(String[] args){
8          ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "applicationContext.xml");
9          IMetier metier = (IMetier) context.getBean( name: "metier");
10         System.out.println("le resultat est : "+metier.calcul());
11     }
12 }
13 }
14
```

## - Version annotations

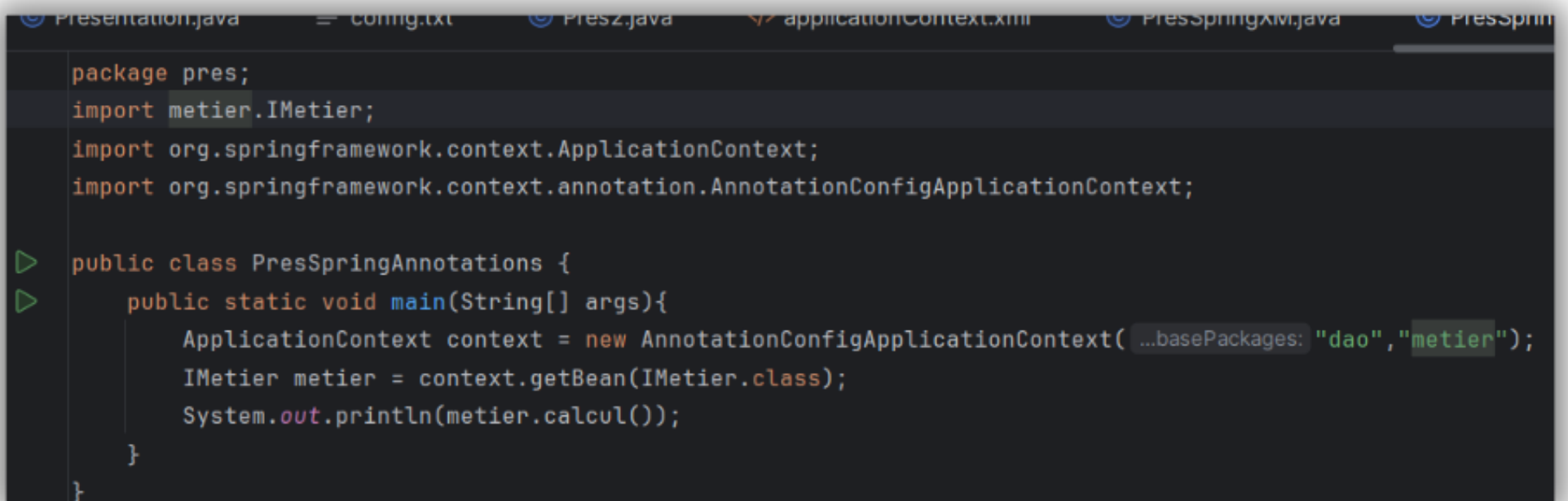
Dans cette version utilisant les annotations dans le framework Spring, la configuration est effectuée au moyen d'annotations directement dans les classes Java, sans nécessiter de fichier XML de configuration.

```
© DaoImpl.java  © MetierImpl.java  ×  © PresSpringAnnotations.java  © DapImpl2.java

1  5 usages
2
3
4
5
6  @Component
7  public class MetierImpl implements IMetier {
8
9      2 usages
10     @Autowired
11     private IDao dao;
12     4 usages
13     @Override
14     public double calcul() {
15         double t=dao.getData();
16         double res = t*540/Math.cos(t*Math.PI);
17         return res;
18     }
19     /* il permet d'affecter ou bien injecter dans la variable dao
20     * un objet d'une classe qui implemente l'interface IDao
21     *
22     */
23     public void setDao(IDao dao) { this.dao = dao; }
24
25
```

- Cette classe est annotée avec @Component, ce qui indique à Spring qu'elle doit être gérée en tant que bean. La dépendance IDao est injectée via le constructeur annoté avec @Autowired, permettant ainsi à Spring d'injecter automatiquement l'implémentation de IDao lors de la création de l'instance de MetierImpl. La méthode calcul() effectue un calcul basé sur les données récupérées via IDao.

- Cette classe représente le point d'entrée de l'application. Elle utilise `AnnotationConfigApplicationContext` pour charger la configuration Spring à partir des packages spécifiés en paramètres (dao et metier). Cela permet à Spring de scanner ces packages à la recherche de composants annotés, puis de les gérer en tant que beans. En récupérant le bean `IMetier` à partir du contexte Spring, cette classe peut ensuite appeler la méthode `calcul()` sur cet objet et afficher le résultat.



```
package pres;

import javax.annotation.processing.Processor;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class PresSpringAnnotations {

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

## Conclusion :

Ce TP m'a permis d'explorer les différentes approches de configuration et d'utilisation du framework Spring pour la gestion des dépendances et l'inversion de contrôle dans une application Java.

J'ai commencé par examiner la configuration XML, qui m'a offert une approche déclarative pour définir les beans et leurs dépendances dans un fichier XML dédié. Cette méthode m'a permis de séparer clairement la logique métier de la configuration de l'infrastructure, ce qui a favorisé la modularité et la réutilisabilité de mon code.

Ensuite, j'ai exploré l'utilisation des annotations dans Spring, une approche plus moderne et concise qui m'a permis de définir les composants et les dépendances directement dans les classes Java à l'aide d'annotations telles que `@Component` et `@Autowired`. Cette méthode a simplifié ma configuration et rendu mon code plus lisible, tout en offrant une flexibilité et une gestion efficace des dépendances par Spring.

Enfin, ce tp m'a permis de comprendre l'importance de la gestion des dépendances et de l'inversion de contrôle dans le développement d'applications Java, ainsi que les différentes manières dont Spring peut faciliter ces aspects du développement logiciel.