SYSTÈMES DISTRIBUÉS

MASTER INTELLIGENCE ARTIFICIELLE ET ANALYSE DES DONNÉES

RÉALISÉE PAR :

ANOADA NOHAYLA

ENCADRÉ PAR :

M. MOHAMED YOUSSFI

ACTIVITÉ PRATIQUE N° 1
INJECTION DES DÉPENDANCES

PARTIE 1 :

1. CRÉER L'INTERFACE IDAO AVEC UNE MÉTHODE GETDATE

```
public interface IDao
{
    2 usages  2 implementations
    double getData();
}
```

Ce code représente une interface Java appelée IDao. Cette interface définit une méthode getData() qui renvoie un nombre de type double.

2. CRÉER UNE IMPLÉMENTATION DE CETTE INTERFACE

```
public class DaoImpl implements IDao
{
    2 usages
    @Override
    public double getData()
    {
        System.out.println("Version 1");
        return Math.random()*40;
    }
}
```

Ce code représente une classe appelée DaoImpl qui implémente l'interface IDao. La méthode getData() est redéfinie dans cette classe pour fournir une implémentation spécifique.

3. Créer l'interface IMetier avec une méthode calcul

```java
public interface IMetier
{
    1 usage  1 implementation
    double calcul();
}
```

Ce code représente une interface Java appelée IMetier. Cette interface définit une méthode calcul() qui renvoie un nombre de type double.

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

```java
public class MetierImpl implements IMetier
{
    2 usages
    private IDao dao=null;
    1 usage
    @Override
    public double calcul()
    {
        double tmp=dao.getData();
        return tmp*540/Math.cos(tmp*Math.PI);
    }
    1 usage
    public void setDao(IDao dao)
    {
        this.dao = dao;
    }
}
```

L'implémentation de l'interface IMetier dans la classe MetierImpl utilise le principe du couplage faible en dépendant d'une interface (IDao) plutôt que d'une implémentation concrète. Cela favorise la flexibilité et la réutilisabilité du code en permettant de changer l'implémentation de IDao sans modifier MetierImpl

5. Faire l'injection des dépendances :

A. Par instanciation statique

```
public class Presentation
{
    public static void main(String[] args)
    {
        DaoImpl dao=new DaoImpl();
        MetierImpl metier=new MetierImpl();
        metier.setDao(dao);
        System.out.println("Resultat "+metier.calcul());
    }
}
```

```
Version 1
Resultat 12039.514101174655

Process finished with exit code 0
```

L'injection de dépendances par instanciation statique consiste à créer explicitement les instances des classes et à les injecter dans d'autres classes via des méthodes setter ou des constructeurs. Cela entraîne un couplage fort entre les classes.

B. Par instanciation dynamique

```
public class Presentation2
{
    public static void main(String[] args) throws Exception
    {
        Scanner scanner=new Scanner(new File( pathname: "config.txt"));
        String daoClasseName = scanner.nextLine();
        Class cDao = Class.forName(daoClasseName);
        IDao dao=(IDao)cDao.newInstance();
        System.out.println(dao.getData());

        String metierClasseName = scanner.nextLine();
        Class cMetier= Class.forName(metierClasseName);
        IMetier metier=(IMetier)cMetier.newInstance();

        Method method=cMetier.getMethod( name: "setDao", IDao.class);
        method.invoke(metier,dao);
    }
}
```

config.txt

```
dao.DaoImpl
metier.MetierImpl
```

```
Version 1
Resultat 16674.453594041643

Process finished with exit code 0
```

Dans ce code, les noms des classes DAO et Metier sont lus à partir d'un fichier de configuration. En utilisant la réflexion, les classes sont chargées dynamiquement et des instances sont créées à l'exécution.

C. EN UTILISANT LE FRAMEWORK SPRING

- VERSION XML

```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/

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

<bean id="metier" class="metier.MetierImpl">
    <property name="dao" ref="dao"/>
</bean>

</beans>
```

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

```
Version 1
20002.272036798582

Process finished with exit code 0
```

Dans ce code nous avons utilise le Framework Spring pour charger le contexte Spring à partir du fichier XML (applicationContext.xml). Ensuite, vous récupérez le bean "metier" du contexte Spring et utilisez ses fonctionnalités, y compris l'injection de dépendance automatique avec le bean "dao" défini dans le fichier XML.

- ## VERSION ANNOTATIONS

INJECTION PAR ATTRIBUT (@AUTOWIRED) :

```java
@Component("metier")
public class MetierImpl implements IMetier
{
    @Autowired
    private IDao dao=null;
```

- Cette annotation est utilisée pour injecter automatiquement une dépendance dans un attribut d'une classe.
- L'injection par attribut est simple à utiliser mais peut poser des problèmes si la dépendance est requise à la création de l'objet.

INJECTION PAR CONSTRUCTEUR :

```java
@Component("metier")
public class MetierImpl implements IMetier
{
    3 usages
    private IDao dao=null;

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

- Cette annotation est utilisée pour injecter automatiquement une dépendance via le constructeur d'une classe.
- L'injection par constructeur garantit que la dépendance est initialisée dès la création de l'objet, ce qui peut rendre le code plus sûr et testable.

INJECTION PAR SETTER :

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

- Cette annotation est utilisée pour injecter automatiquement une dépendance via une méthode setter d'une classe.
- L'injection par méthode est utile lorsque vous avez besoin de modifier la dépendance après la création de l'objet.

INJECTION PAR NOM (@QUALIFIER) :

```
@Component("metier")
public class MetierImpl implements IMetier
{
    @Autowired
    @Qualifier("dao")
    private IDao dao=null;
```

- Lorsque plusieurs beans du même type sont disponibles dans le contexte Spring, l'annotation @Qualifier est utilisée pour spécifier le nom du bean à injecter.

```java
@Component("dao")
public class DaoImpl implements IDao
{
    2 usages
    @Override
    public double getData()
    {
        System.out.println("Version 1");
        return Math.random()*40;
    }
}
```

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

```
Version 1
1186.358421759364

Process finished with exit code 0
```

Dans ces codes nous avons utilise AnnotationConfigApplicationContext pour charger les configurations Spring basées sur les annotations @Component des classes DaoImpl et MetierImpl. En utilisant "dao", "metier" comme arguments de AnnotationConfigApplicationContext, Spring identifie et gère les beans correspondants dans le contexte Spring.