ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR
Membre de HONORIS UNITED UNIVERSITIES

Matière : JEE

Rapport de Travaux Pratiques

Lien Git du Code : https://github.com/Salah2210/JeeTps

Plan

Introduction
Différence entre couplage faible et couplage fort
Réalisation
Conclusion

Introduction

En informatique, et plus particulièrement en développement logiciel, un patron de conception (souvent appelé design pattern) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels

Différence entre couplage faible et couplage fort

Un couplage fort signifie que les classes et les objets dépendent les uns des autres. En général, le couplage fort n'est pas bon car il réduit la flexibilité et la réutilisation du code, tandis que le couplage faible signifie la réduction des dépendances d'une classe qui utilise directement les différentes classes.

Couplage fort:

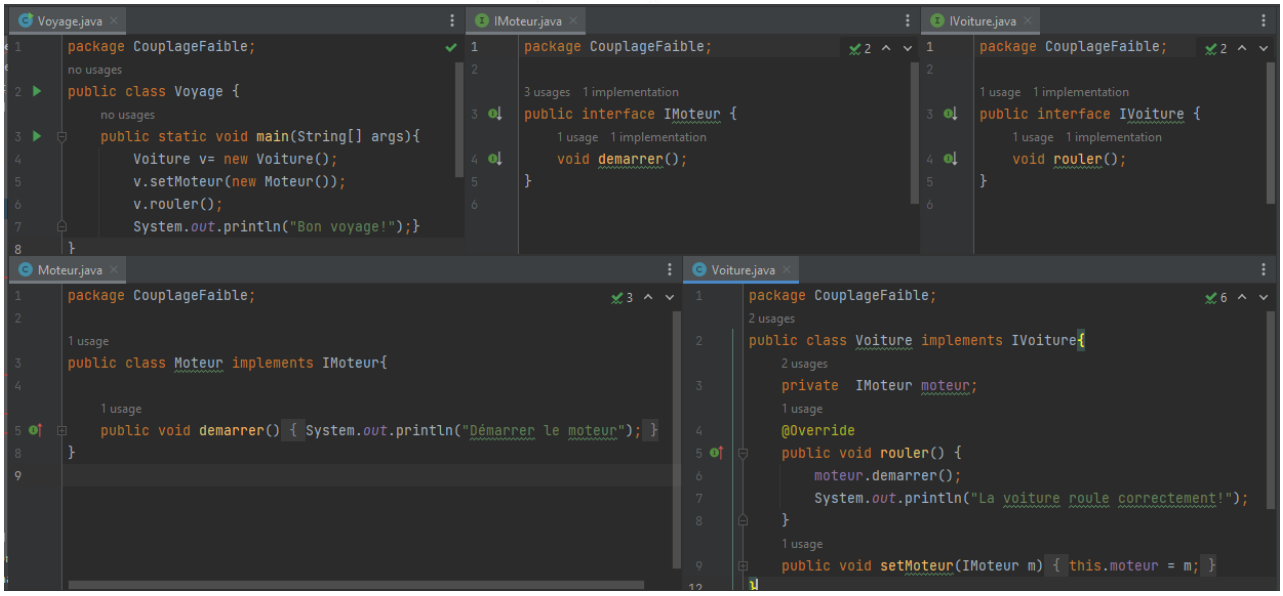
♣ Un objet fortement couplé est un objet qui a besoin de connaître les autres objets et est généralement très dépendant les uns des autres.
♣ La modification d'un objet dans une application fortement couplée nécessite souvent de modifier d'autres objets.

Couplage faible:

♣ Le couplage faible permet de réduire les interdépendances entre les composants d'un système dans le but de réduire le risque que les changements dans un composant nécessitent des changements dans tout autre composant.
♣ Le couplage faible est un concept destiné à augmenter la flexibilité du système, à le rendre plus maintenable et à rendre l'ensemble du Framework plus stable

Réalisation

Couplage Faible :



Voyage.java

```java
package CouplageFaible;

public class Voyage {
    public static void main(String[] args){
        Voiture v= new Voiture();
        v.setMoteur(new Moteur());
        v.rouler();
        System.out.println("Bon voyage!");}
}
```

IMoteur.java

```java
package CouplageFaible;

public interface IMoteur {
    void demarrer();
}
```

IVoiture.java

```java
package CouplageFaible;

public interface IVoiture {
    void rouler();
}
```

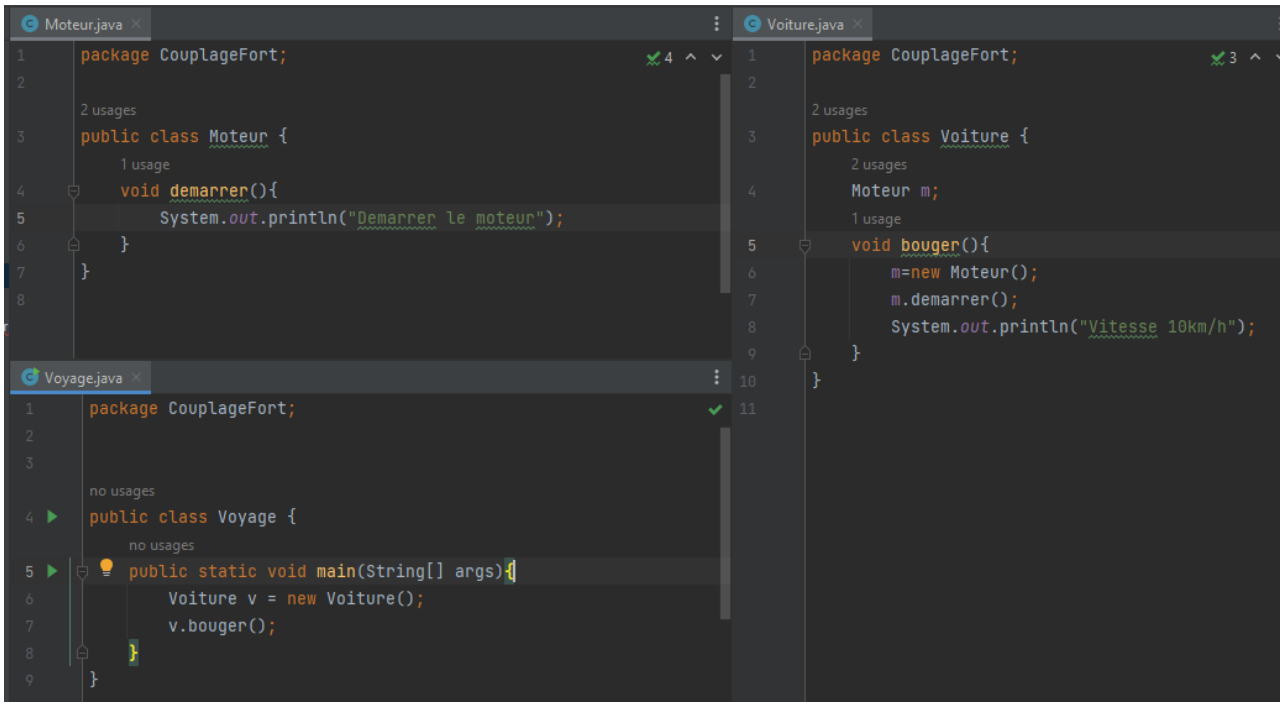
Moteur.java

```java
package CouplageFaible;

public class Moteur implements IMoteur{

    public void demarrer() { System.out.println("Démarrer le moteur"); }
}
```

Voiture.java

```java
package CouplageFaible;

public class Voiture implements IVoiture{
    private  IMoteur moteur;
    @Override
    public void rouler() {
        moteur.demarrer();
        System.out.println("La voiture roule correctement!");
    }
    public void setMoteur(IMoteur m) { this.moteur = m; }
}
```

Couplage Fort :



Moteur.java

```java
package CouplageFort;

public class Moteur {
    void demarrer(){
        System.out.println("Demarrer le moteur");
    }
}
```

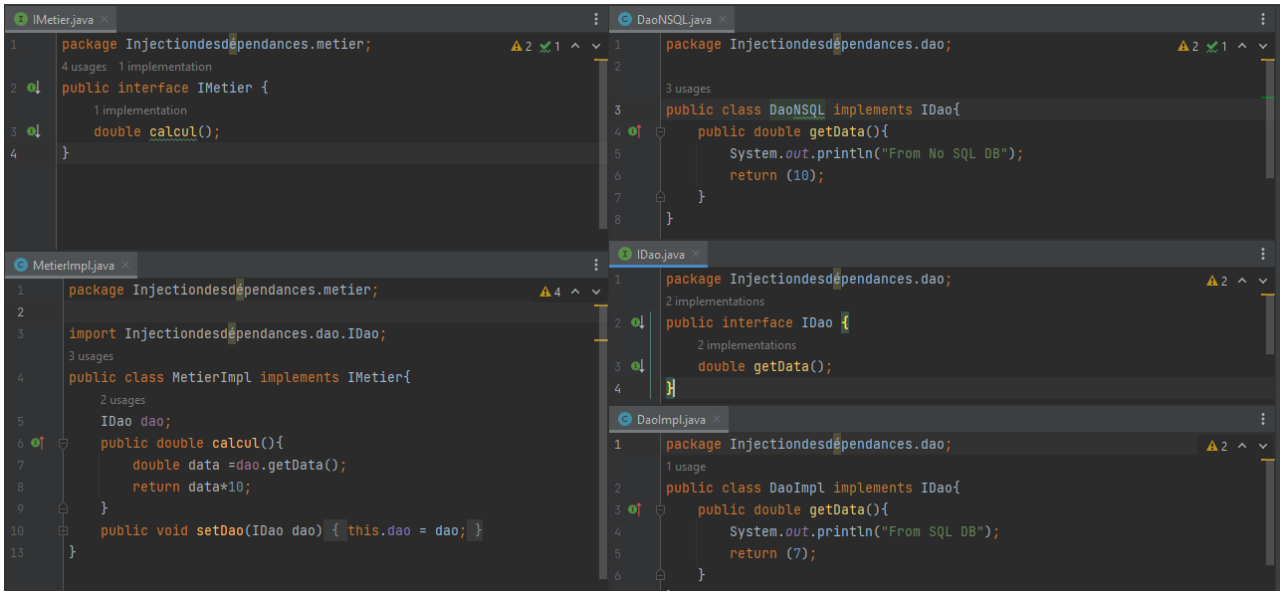
Voiture.java

```java
package CouplageFort;

public class Voiture {
    Moteur m;
    void bouger(){
        m=new Moteur();
        m.demarrer();
        System.out.println("Vitesse 10km/h");
    }
}
```

Voyage.java

```java
package CouplageFort;

public class Voyage {
    public static void main(String[] args){
        Voiture v = new Voiture();
        v.bouger();
    }
}
```

Injection des dépendances :



IMetier.java

```
package Injectiondesdépendances.metier;
4 usages  1 implementation
public interface IMetier {
    1 implementation
    double calcul();
}
```

DaoNSQL.java

```
package Injectiondesdépendances.dao;

3 usages
public class DaoNSQL implements IDao{
    public double getData(){
        System.out.println("From No SQL DB");
        return (10);
    }
}
```

MetierImpl.java

```
package Injectiondesdépendances.metier;

import Injectiondesdépendances.dao.IDao;
3 usages
public class MetierImpl implements IMetier{
    2 usages
    IDao dao;
    public double calcul(){
        double data =dao.getData();
        return data*10;
    }
    public void setDao(IDao dao) { this.dao = dao; }
}
```

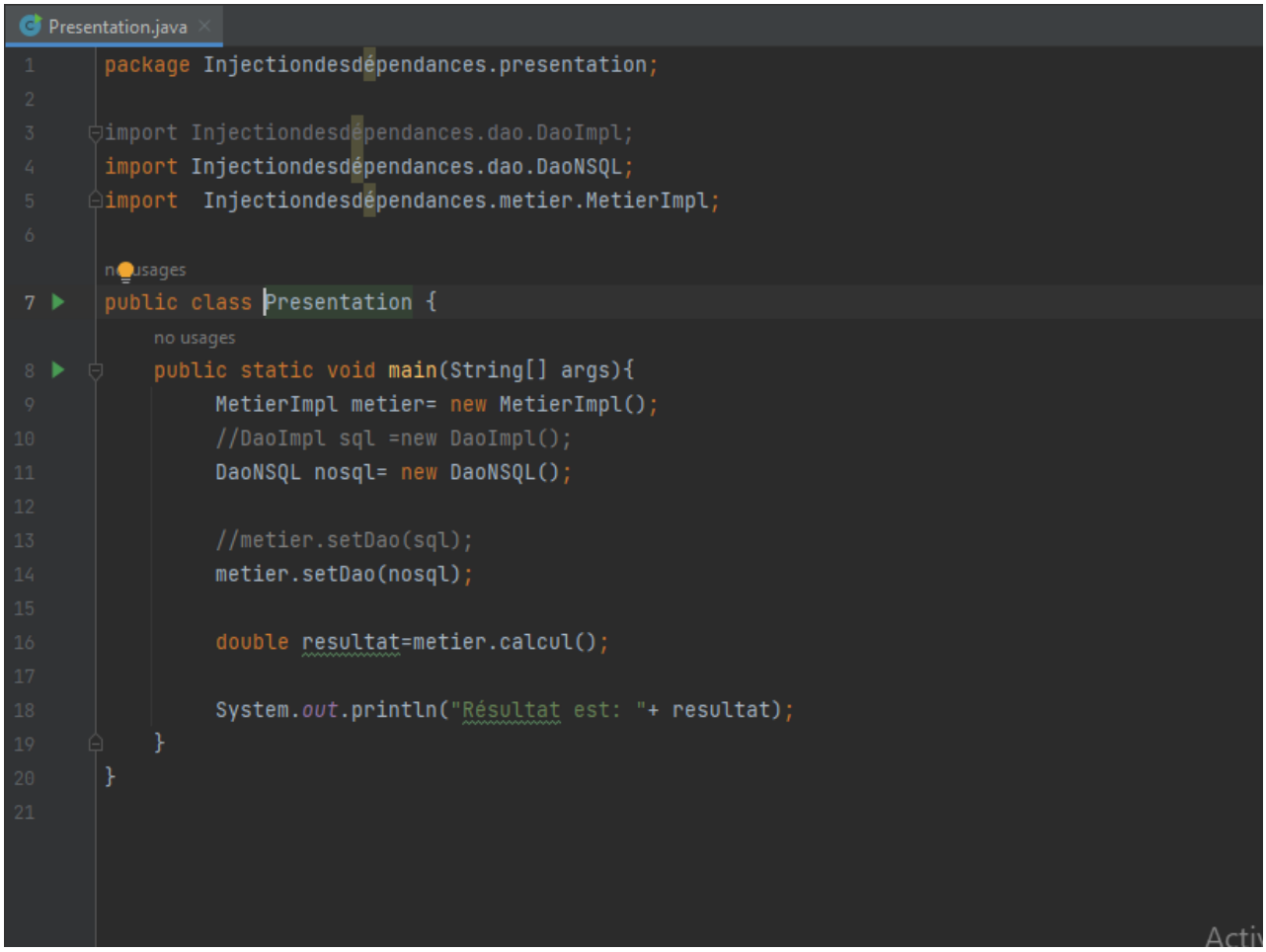
IDao.java

```
package Injectiondesdépendances.dao;
2 implementations
public interface IDao {
    2 implementations
    double getData();
}
```

DaoImpl.java

```
package Injectiondesdépendances.dao;
1 usage
public class DaoImpl implements IDao{
    public double getData(){
        System.out.println("From SQL DB");
        return (7);
    }
}
```

Par Instanciation statique :



Par instanciation dynamique :

PresDynamique.java × config.txt ×

```java
package Injectiondesdépendances.presentation;

import ...

no usages
public class PresDynamique {
    no usages
    public static void main(String[] args) throws FileNotFoundException,ClassNotFoundException,
            InstantiationException,IllegalAccessException, NoSuchMethodException,InvocationTargetException{
        Scanner sc= new Scanner(new File( pathname: "src/main/java/Injectiondesdépendances/config.txt"));
        String dao = sc.nextLine();
        Class clsDao = Class.forName(dao);
        IDao objDao =(IDao)clsDao.newInstance();

        String metier = sc.nextLine();
        Class clsMetier = Class.forName(metier);
        IMetier objMetier =(IMetier)clsMetier.newInstance();

        Method method = clsMetier.getMethod( name: "setDao", IDao.class);
        method.invoke(objMetier,objDao);

        System.out.println(objMetier.calcul());
    }
}
```

PresDynamique.java × config.txt ×

```
Injectiondesdépendances.dao.DaoImpl;
Injectiondesdépendances.metier.MetierImpl;
```

Injection des dépendances avec Spring :

DaoImpl.java

```java
package Injectiondesdépendances_Spring.dao;

2 usages
public class DaoImpl implements  IDao{
    public double getData(){
        System.out.println("From SQL DB");
        return (7);
    }
}
```

IDao.java

```java
package Injectiondesdépendances_Spring.dao;

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

SQL.java

```java
package Injectiondesdépendances_Spring.dao;

3 usages
public class SQL implements IDao{

    public double getData(){
        System.out.println("From SQL DB");
        return (10);
    }
}
```

DaoNSQL.java

```java
package Injectiondesdépendances_Spring.dao;

no usages
public class DaoNSQL implements IDao{
    public double getData(){
        System.out.println("From No SQL DB");
        return (10);
    }
}
```

MetierV3.java

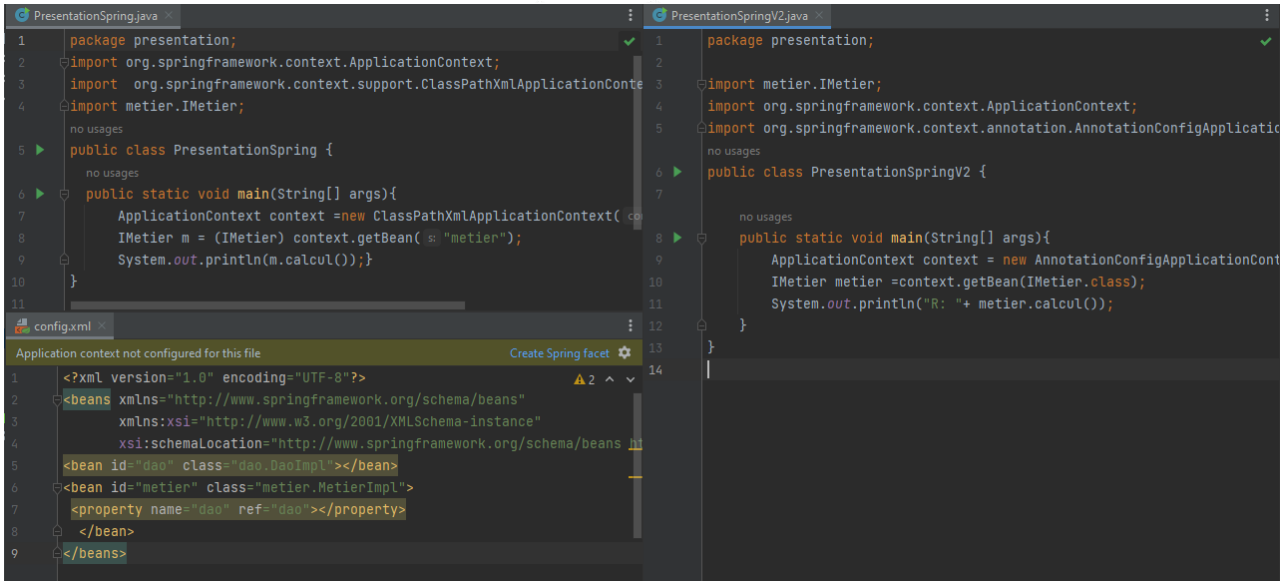
```java
package Injectiondesdépendances_Spring.metier;

import Injectiondesdépendances_Spring.dao.IDao;

no usages
public class MetierV3 implements IMetier{

    2 usages
    IDao dao;

    @Override
    public double calcul() {
        double d =dao.getData();
        return d*2021;
    }

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

MetierImpl.java

```java
package Injectiondesdépendances_Spring.metier;

import Injectiondesdépendances_Spring.dao.IDao;
4 usages
public class MetierImpl implements IMetier{
    2 usages
    IDao dao;
    public double calcul(){
        double data =dao.getData();
        return data*10;
    }
    public void setDao(IDao dao) { this.dao = dao; }
}
```

IMetier.java

```java
package Injectiondesdépendances_Spring.metier;

11 usages  2 implementations
public interface IMetier {
    2 implementations
    double calcul();
}
```



Conclusion

Tous les patterns présentés ici ont pour but de faciliter l'expression du modèle du domaine dans le code. En effet, ils aident:

♣ À séparer le logique métier et la logique technique (Repository, Layered Architecture),

♣ À organiser le code (Value Object, Entities, Aggregates, Modules, Service),

♣ À réfléchir aux compromis à faire entre concurrence et cohérence (Aggregates, Domain Events),

♣ À articuler clairement les relations entre concurrence et logique métier (Domain Event),

♣ À forcer l'explicitation des concepts métiers dans le code (Value Object, Entities, Domain Events).

Ainsi, appliquer ces patterns permet d'obtenir un code plus clair, plus organisé, mieux adapté aux systèmes distribués, et dans lequel la logique métier se dégage clairement. De plus, mettre en place ces patterns au sein d'un projet permet d'initier l'appétence au métier pour les développeurs. Ces patterns sont aussi un excellent point de départ pour appliquer les concepts du DDD au sein d'un projet informatique. En particulier, les Value Objects et les Aggregates sont de puissants « absorbeurs » de complexité, et sont très indiqués lorsque l'on commence un refactoring du code.

Toutefois, s'il s'agit d'un très bon point de départ, le DDD ne se limite pas aux patterns tactiques et dispos d'autres outils et approches qui ne se limitent pas au code. C'est ce que nous verrons dans la suite de notre série : notre prochain article traitera de Supple Design et le dernier présentera les patterns stratégiques du DDD