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.

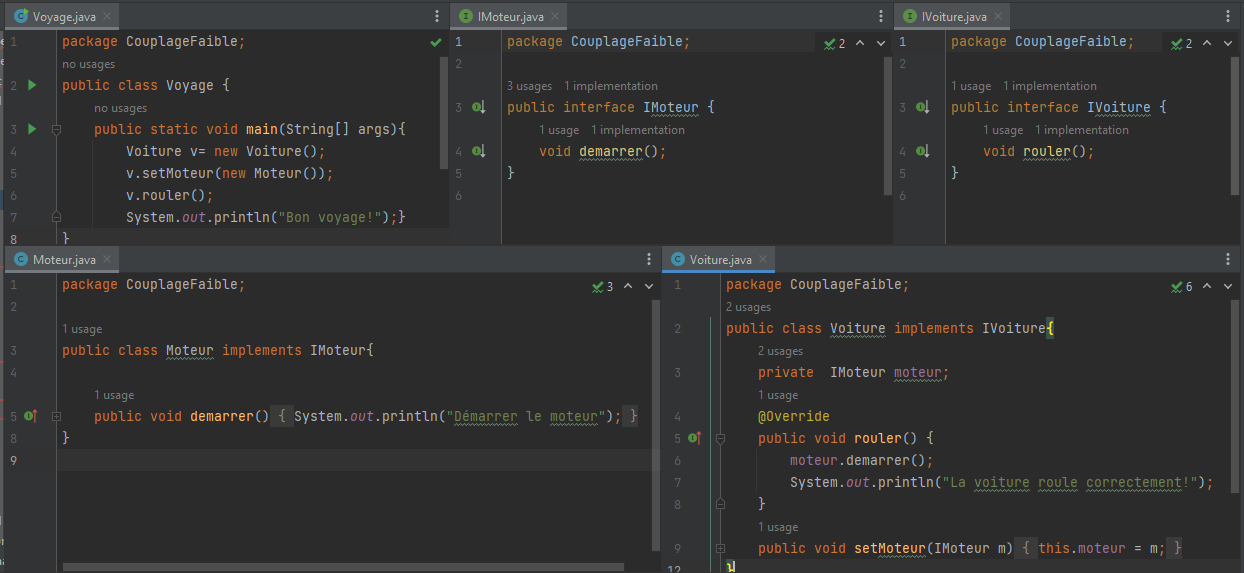
***Couplage faible:***

♣ 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.

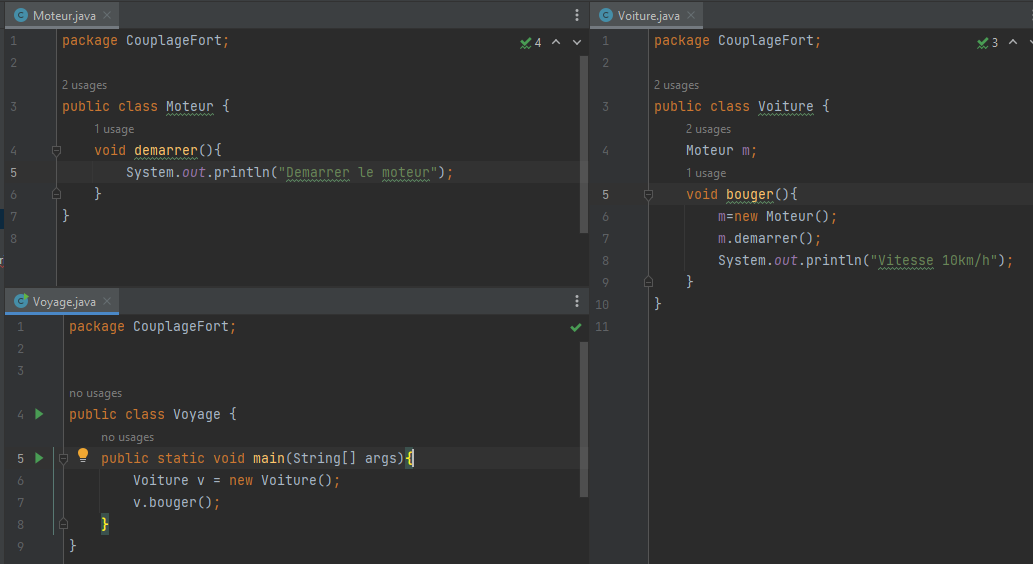
♣ Le couplage faible est un concept destiné à augmenter la flexibilité du système, à le rendre plus maintenable et à rendre l’ensemble du Framework plus stable

**Réalisation**

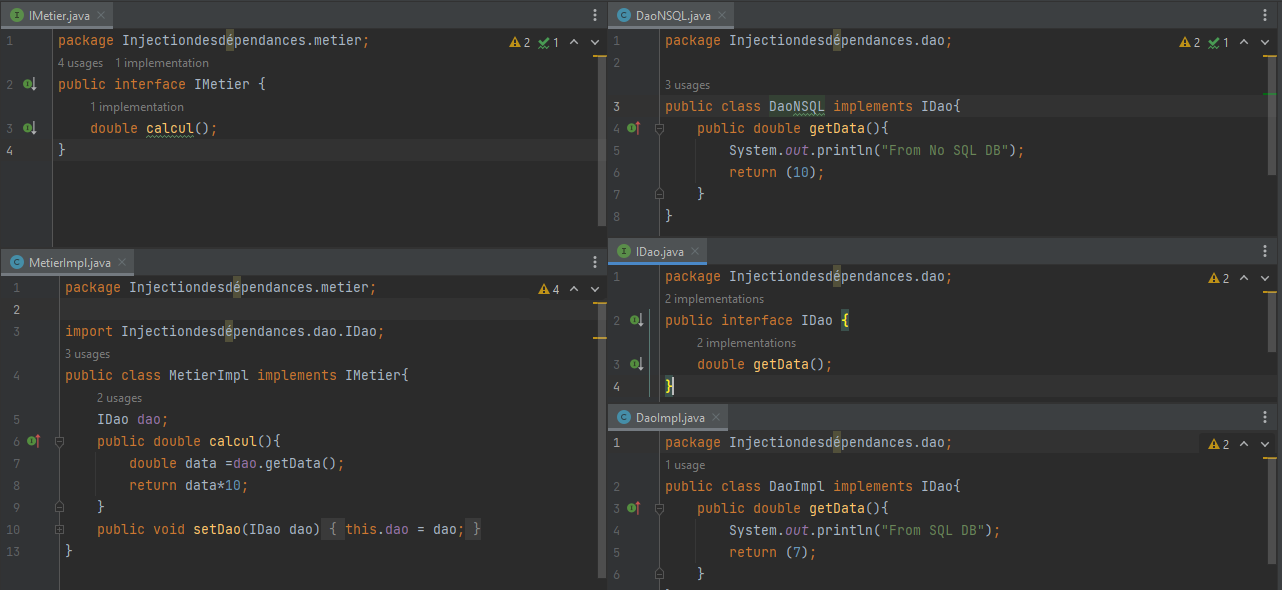
Couplage Faible :



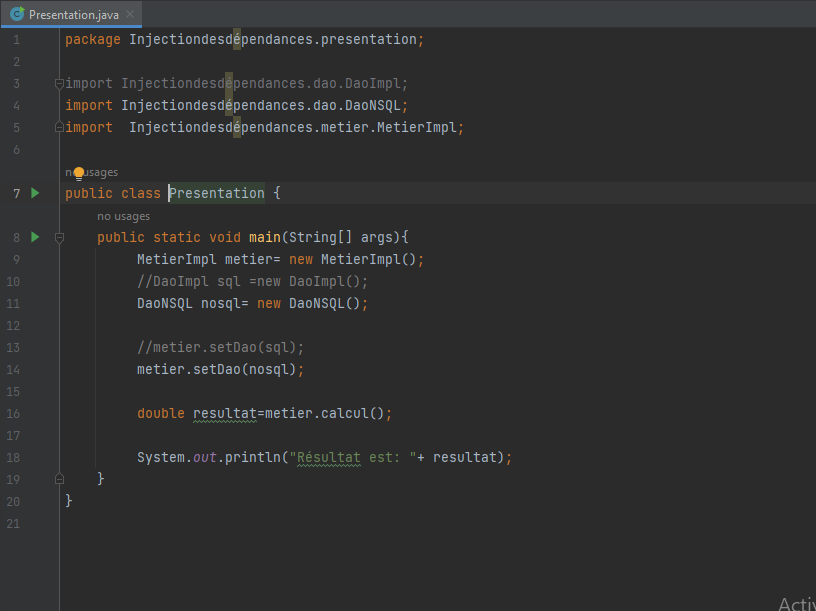
Couplage Fort :



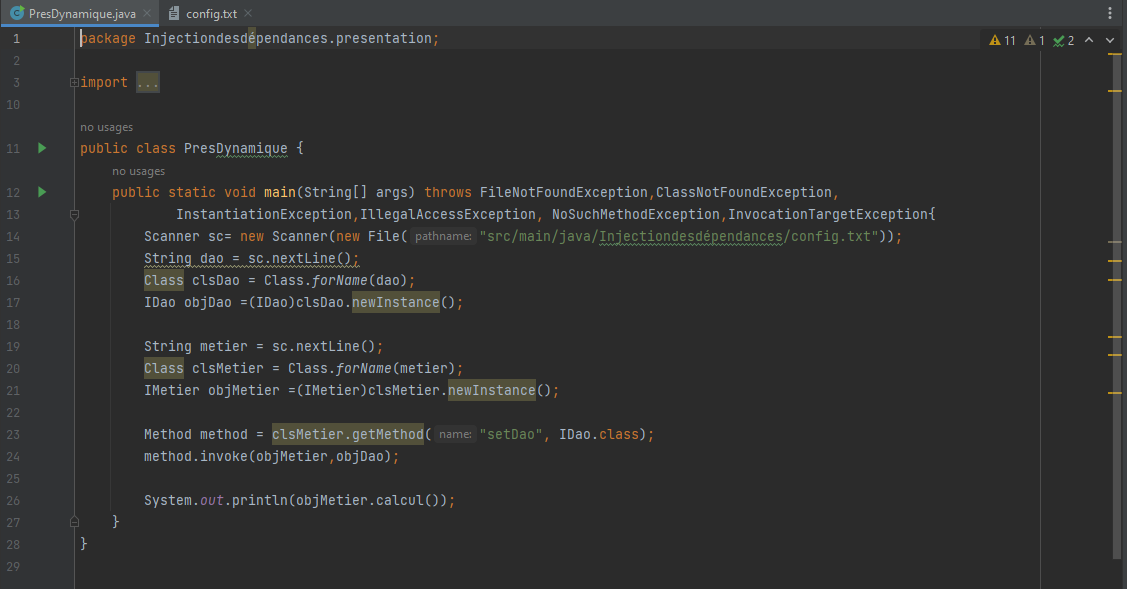
Injection des dépendances :

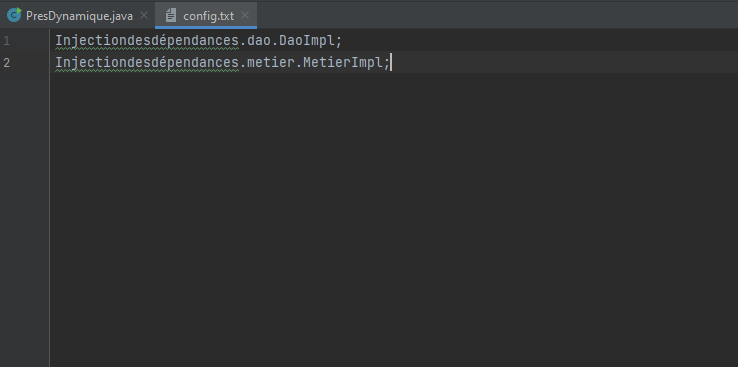


Par Instanciation statique :

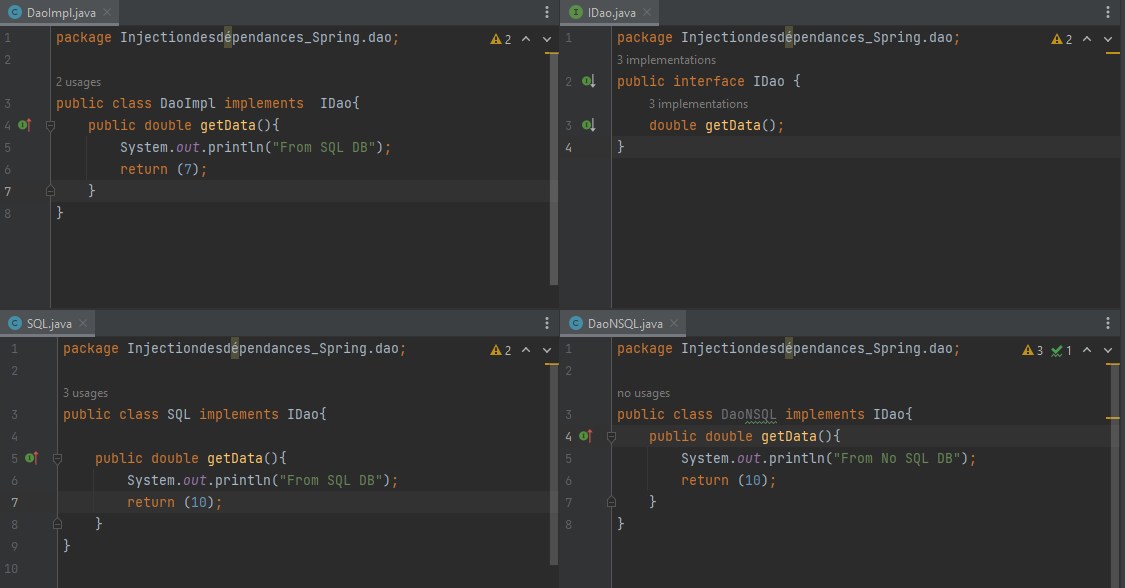


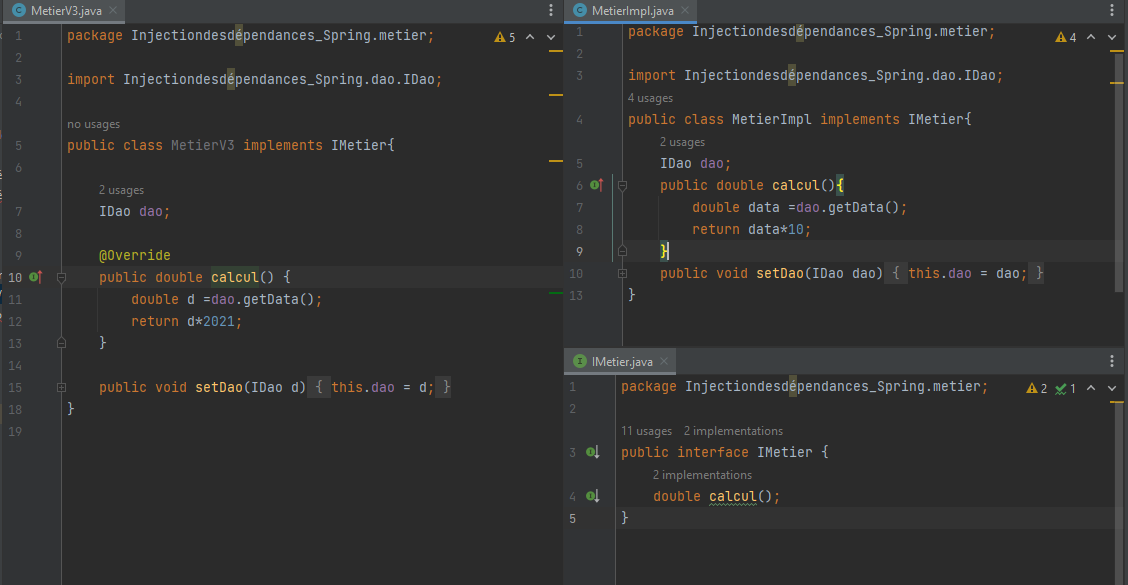
Par instanciation dynamique :

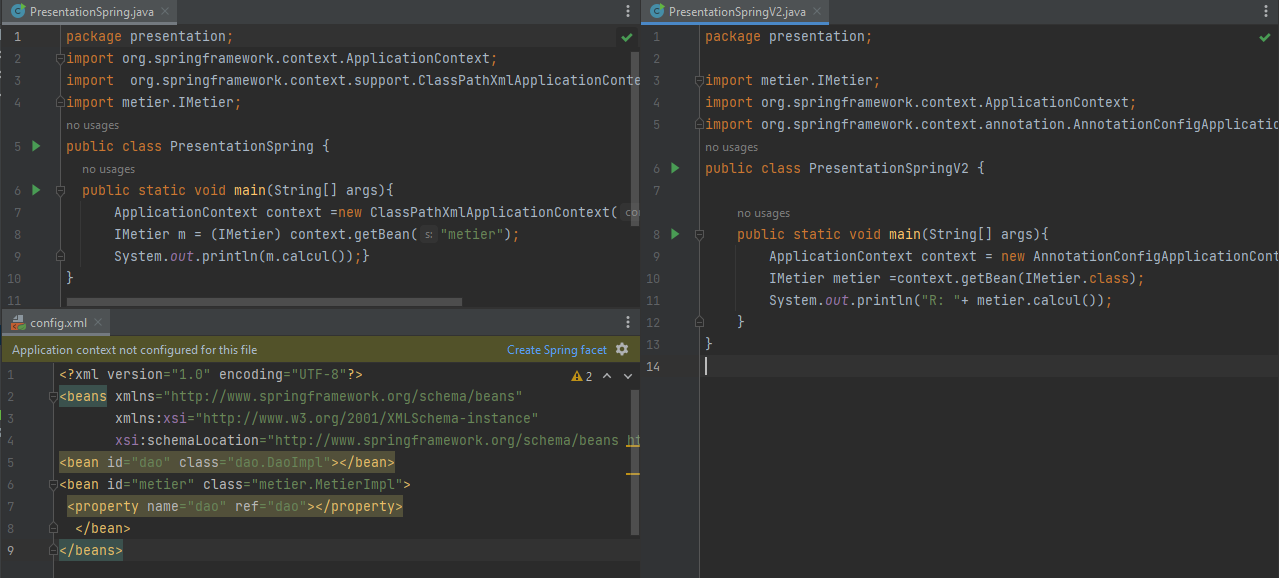




Injection des dépendances avec Spring :







**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