Aymane Hinane Rapport 4IIR G5 emsi centre

Lien Git du Code :
https://github.com/AymaneHinane/Spring-Boot-tp1.git

Plans du Rapport

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[1].

Domain-Driven Design

Le DDD (ou Domain-Driven Design) est une approche de la conception logicielle qui préconise, entre autres, de mettre le domaine métier au centre du développement logiciel.

Aggregates

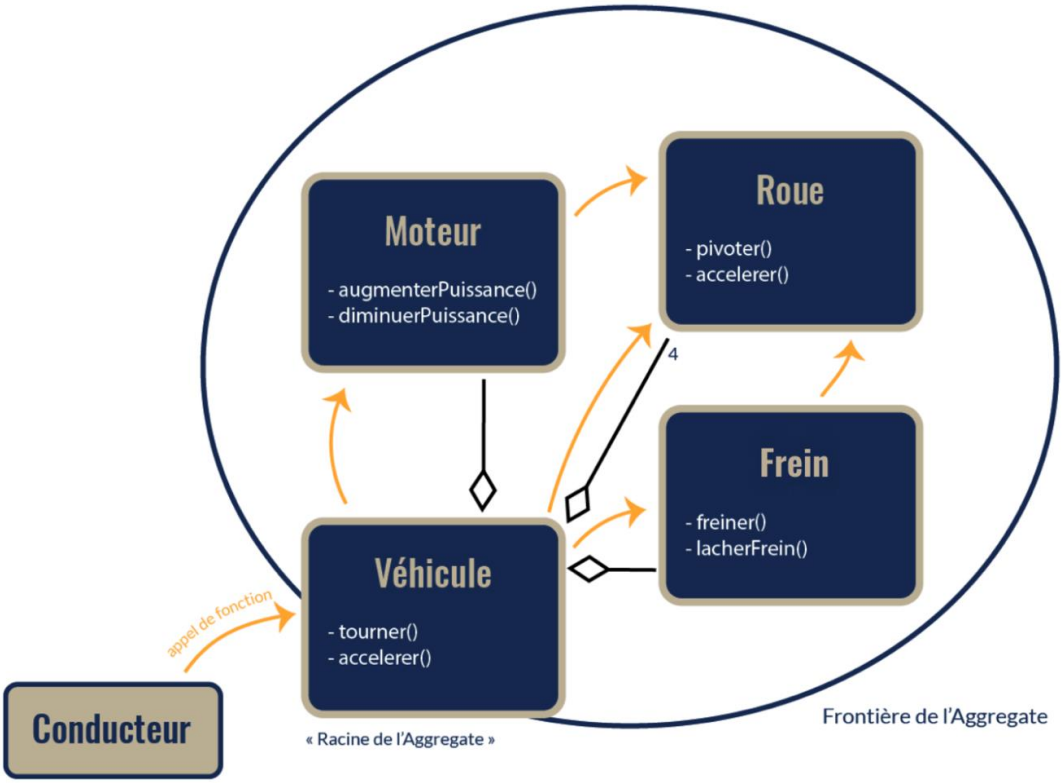
Dans un système complexe (par exemple : distribué, multi-threadé…), il arrive souvent qu'il faille assurer la cohérence d'un ensemble d'objets. Or, cette cohérence peut vite se révéler être un cauchemar à maintenir dans le code.

Un *Aggregate* est un ensemble d'objets métiers (*Value Objects* ou *Entities*) liés ensemble. Parmi ces objets, un seul(en général, une *Entity*) aura un rôle particulier : le rôle de racine de l'*Aggregate*. Tous les changements apportés à l'*Aggregate* par le reste du code devront passer par une méthode de la racine. Il est donc interdit de modifier l'*Aggregate* en modifiant directement un objet non-racine.

Ainsi, les *Aggregates* sont un bon moyen pour implémenter les invariants métiers et les règles de gestion. Afin d'assurer la cohérence d'un *Aggregate* au sein d'un environnement multi-threadé

Exemple

Dans l'exemple du domaine métier lié à l'automobile, si l'on veut modéliser en détail une voiture, il faut prendre en compte le fait qu'une voiture contient des pièces qui communiquent entre elles. Ainsi une voiture contient un moteur, des roues, des freins. Toutes ces parties communiquent entre elles : le moteur fait accélérer les roues, tandis que les freins les bloquent, et le volant les fait tourner. Afin de garantir la cohérence des données (éviter qu'au niveau informatique, la voiture tourne à droite, alors que les roues sont orientées à gauche), on peut modéliser la voiture comme un *Aggregate* de pièces, et interdire à tout objet externe d'appeler les méthodes des objets de cet *Aggregate* excepté la racine de cet *Aggregate* : ici la voiture elle-même.



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.

Services

Il est des cas où les *Entities*, *Value Objects*, et *Aggregates* ne suffisent pas pour contenir toute la logique d'un domaine métier.
Dans ce cas on pourra utiliser des *Services* qui sont des classes qui effectuent les traitements métiers qui ne peuvent être réalisés de manière satisfaisante par tout autre objet métier. C'est typiquement le cas des transformations d'objets métiers en d'autres objets.
Un autre exemple est une transaction monétaire entre deux comptes bancaires : quel objet va s'occuper de l'orchestration de la transaction au niveau du code ? Implémenter la transaction dans l'un des deux objets « compte bancaire » serait maladroit car au niveau métier, cela ne fait guère de sens. Du coup, implémenter la transaction au sein d'une classe de type *Services* est une meilleure solution.
Attention toutefois à implémenter uniquement des règles métier qui ne peuvent être prises en compte par aucun objet métier (*Entities*, *Value Object* ou *Aggregates*). Sans quoi on risquerait d'aboutir à un *Anemic Domain Model*, c'est-à-dire une modélisation du métier très pauvre : les V*alue Objects*, *Aggregates* et *Entities* risqueront d'être de simples DTO et toute la logique métier serait dans des classes *Services* obèses et peu maintenables.

Exemple

Toujours dans le cas d'une application de gestion d'une entreprise de location de voiture, supposons que nous voulions créer une méthode calculant le prix de location d'une voiture. Pour cela, il faudra coder une fonction `computePrice` qui calculera le prix de location à partir du véhicule, de la date de location, du lieu de la location, voire du client lui-même (car il peut bénéficier de certaines réductions). S'il est clair que cette méthode devra faire partie de la couche domaine, dans quel type d'objet doit-on placer cette méthode ? Doit-on la coder dans l'*Aggregate* `Vehicule`, dans la *Value Object* `Date`, ou dans l'*Entity* `Client` ? Il est clair qu'aucun de ces objets ne convient parfaitement pour abriter la méthode `computePrice`. Dans ce genre de cas, la meilleure solution est donc de stocker cette méthode dans une classe Service spécialisée (par exemple `RentService`) qui fera partie intégrante du modèle.

Domain Events

Un pattern très utile permet de traiter ce problème : les *Domain Events*. Ce sont des objets qui modélisent une transaction métier sur une *Entity* ou un *Aggregate*. Ces objets ont souvent vocation à être persistés, ce qui permet d'avoir un historique complet des changements qu'a subi le modèle. Un autre avantage des *Domain Events* est qu'ils permettent de simplifier la synchronisation inter-systèmes. Dès qu'un système a été modifié, il peut émettre un *Domain Event* pour notifier aux autres systèmes la modification de son état.

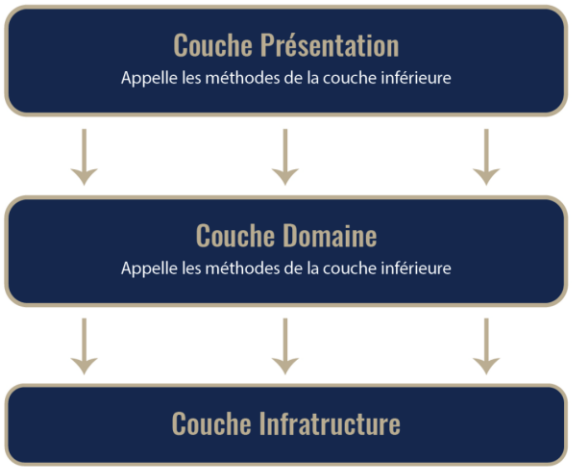
Dans le domaine bancaire, des exemples classiques de *Domain Events* sont les évènements « Comptes crédités » et « Comptes débités » des *Entities* « Comptes bancaires ».

Layered Architecture

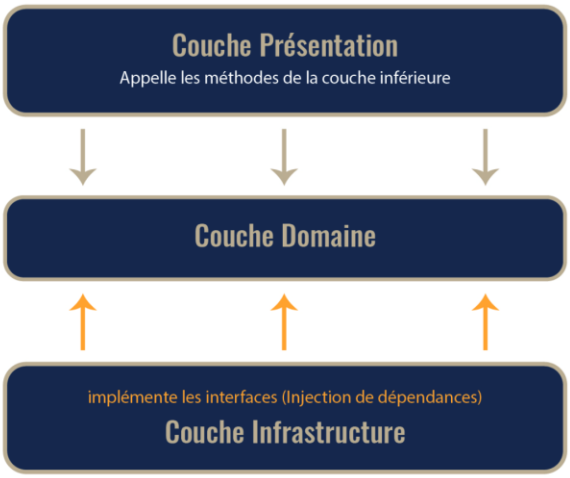
Tous les patterns vus jusqu'à présent permettent de modéliser un domaine métier. Mais si l'on n'isole pas le code métier du reste du code, ces patterns perdent beaucoup de leur valeur.

Du coup, une des manières d'isoler le code métier est d'utiliser la classique architecture en couches. Il suffit alors de réserver une de ces couches au code métier afin de séparer les problématiques métiers des autres composants d'un logiciel (interface graphique, couche de données, …).

Toutefois, si l'on reste sur une architecture en couches « classique », la couche « métier » sera dépendante du code des couches du dessous (stockage de données, infrastructure). Le code métier sera ainsi dépendant des choix techniques du projet, et devra donc être modifié si ces choix changent. Or, cela est contradictoire avec un des objectifs du DDD qui est que le code métier ne change que lorsque le métier change (ou lorsque la compréhension du métier change). Pour éviter cet écueil, une astuce consiste à utiliser la technique d'injection de dépendances afin que cela soit la couche technique qui dépende du code métier et non l'inverse. Ainsi, la couche métier ne dépendra d'aucune autre couche, ce sont les autres couches qui dépendront d'elle.



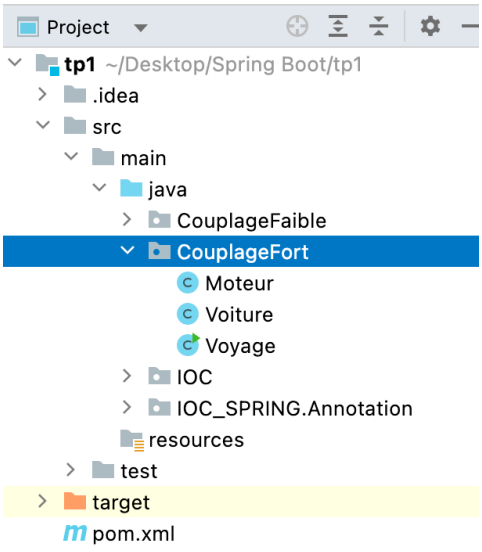
Architecture en couche classique

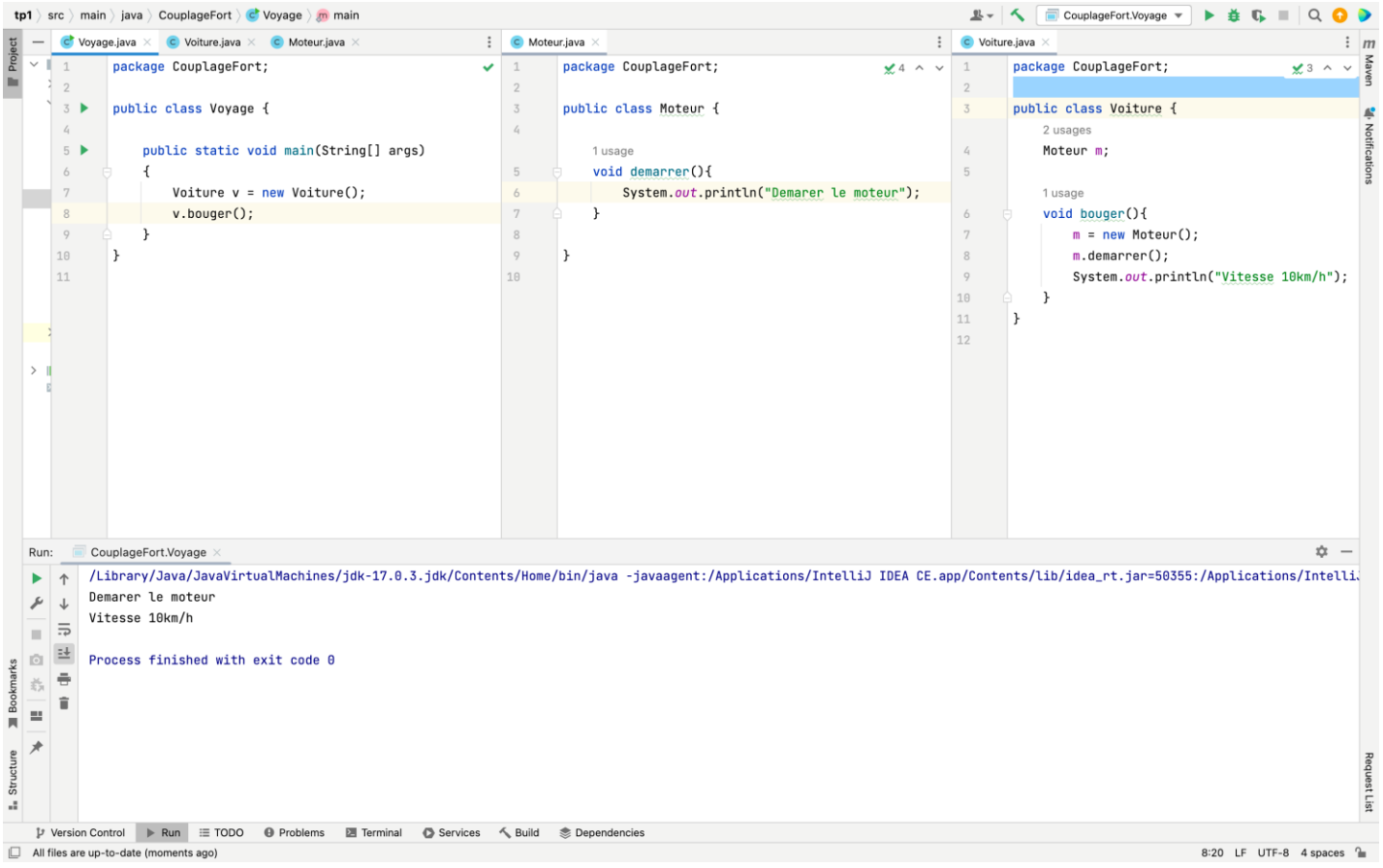


Architecture en couche conseillée pour le DDD

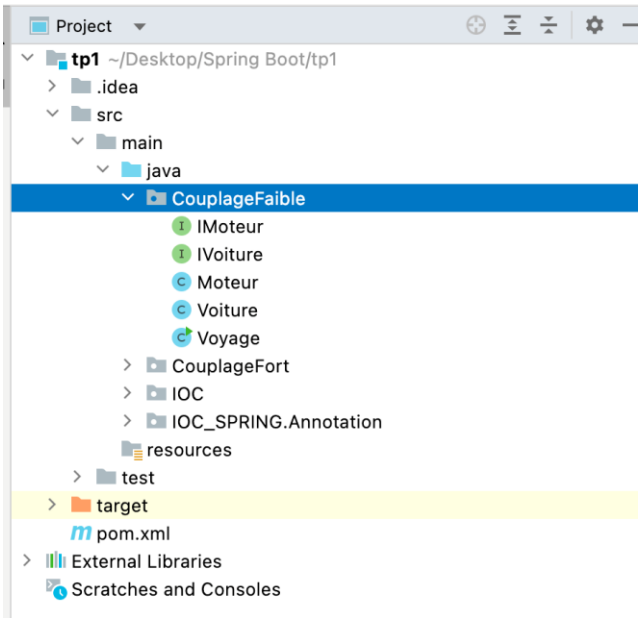
TP Realisation

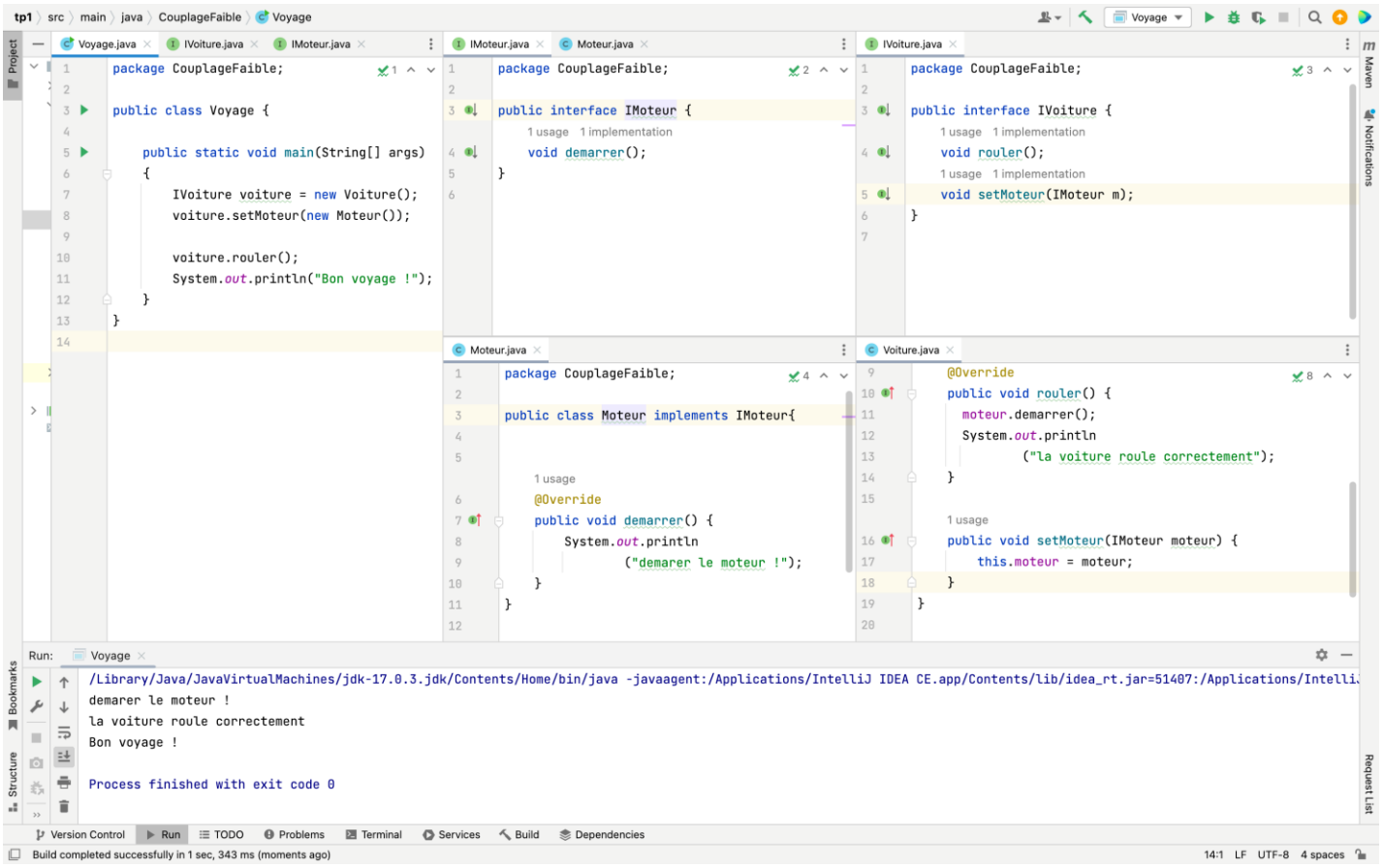
Couplage Fort



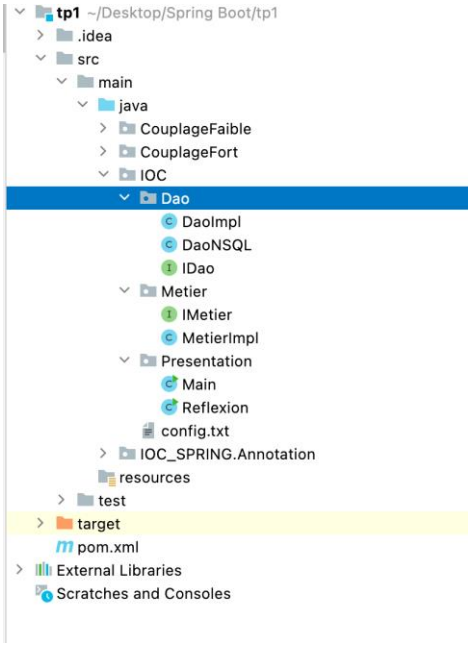


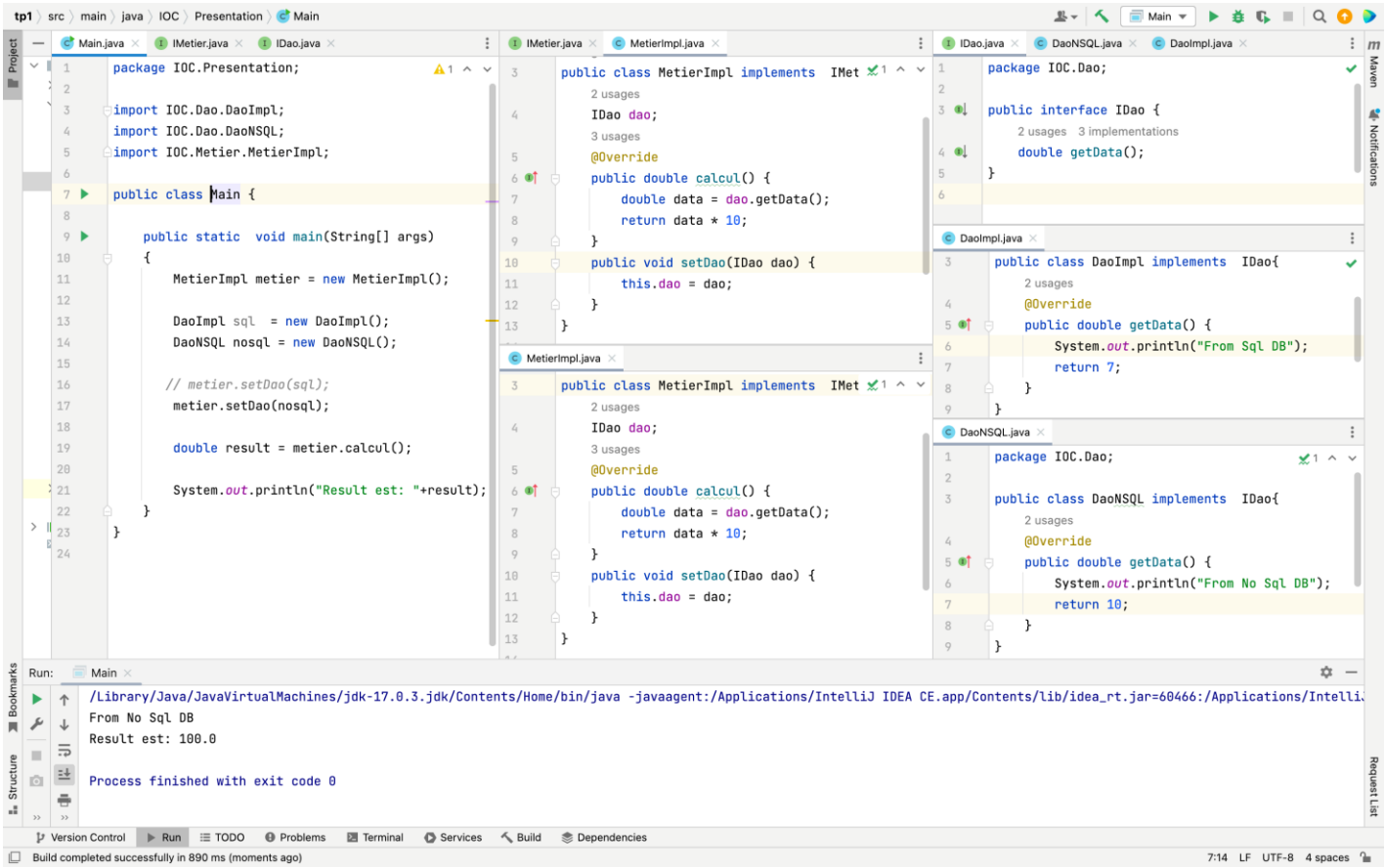
Couplage Faible





l’injection des dépendances par instanciation statique





l’injection des dépendances par instanciation dynamique

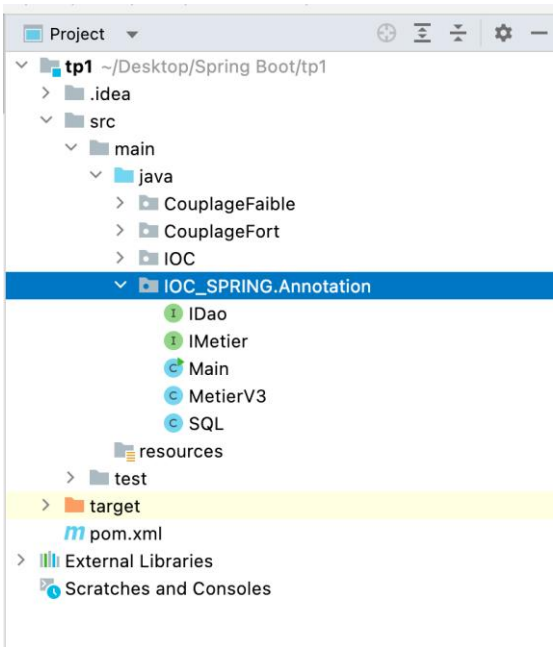
Reflexion.java config.txt

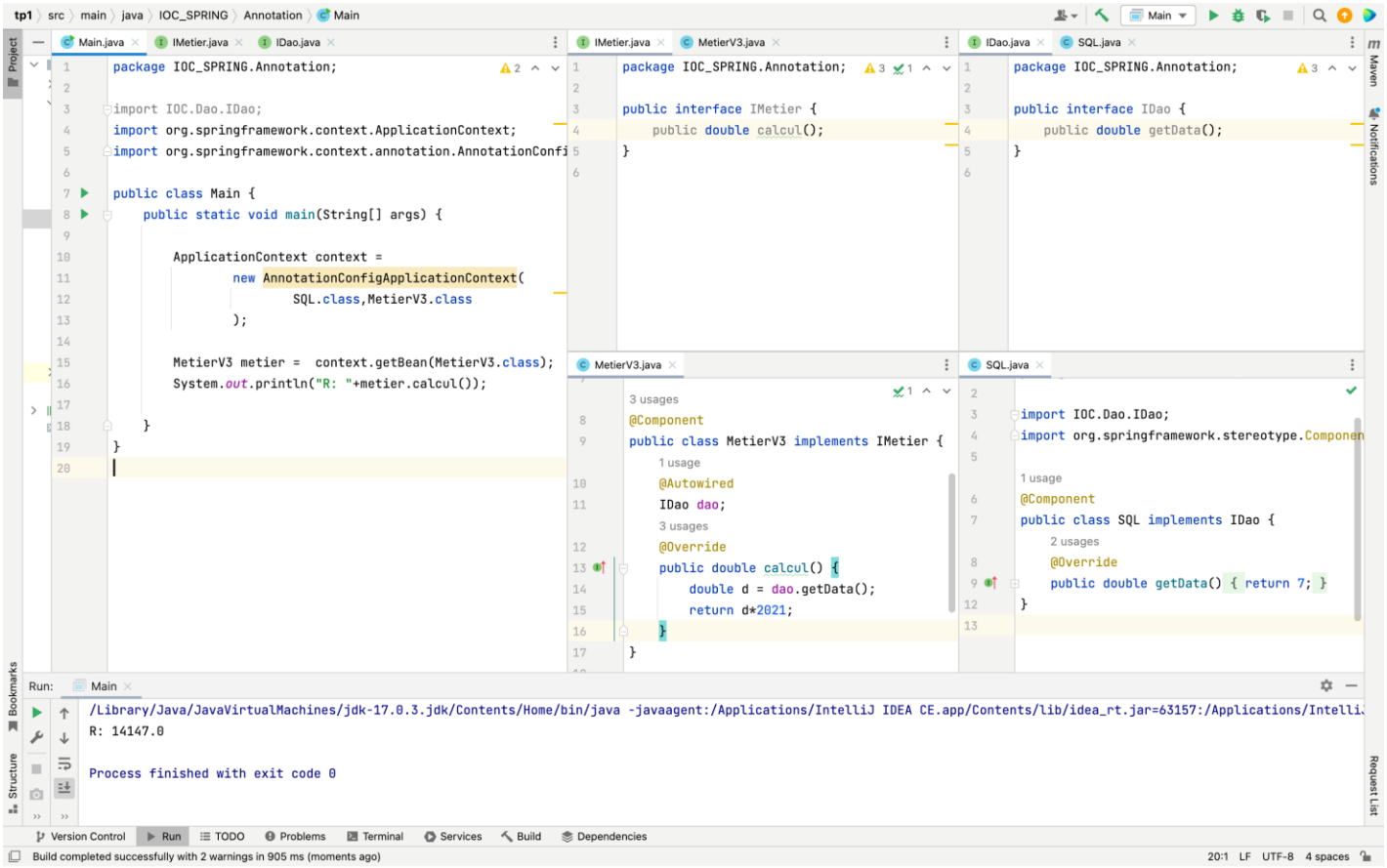
```java
import java.io.FileNotFoundException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.Scanner;

public class Reflexion {

    public static void main(String[] args) throws FileNotFoundException,
            ClassNotFoundException, InstantiationException, IllegalAccessException,
            NoSuchMethodException, InvocationTargetException {

        Scanner sc = new Scanner(new File( pathname: "src/main/java/IOC/config.txt"));

        String dao = sc.nextLine();
        System.out.println(dao);
        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());
    }
}
```

Reflexion.java config.txt

```
IOC.Dao.DaoImpl
IOC.Metier.MetierImpl
```

Injection des dépendances avec Spring: Beans





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 la 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 dispose 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.