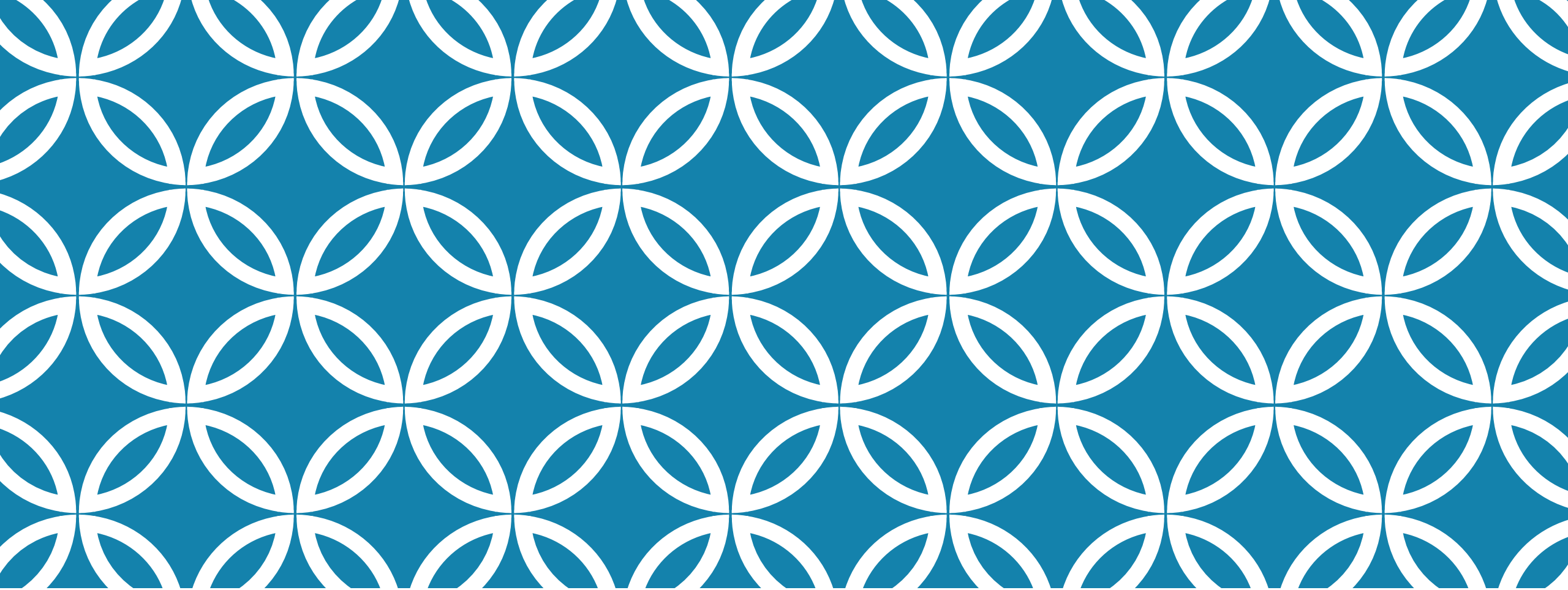
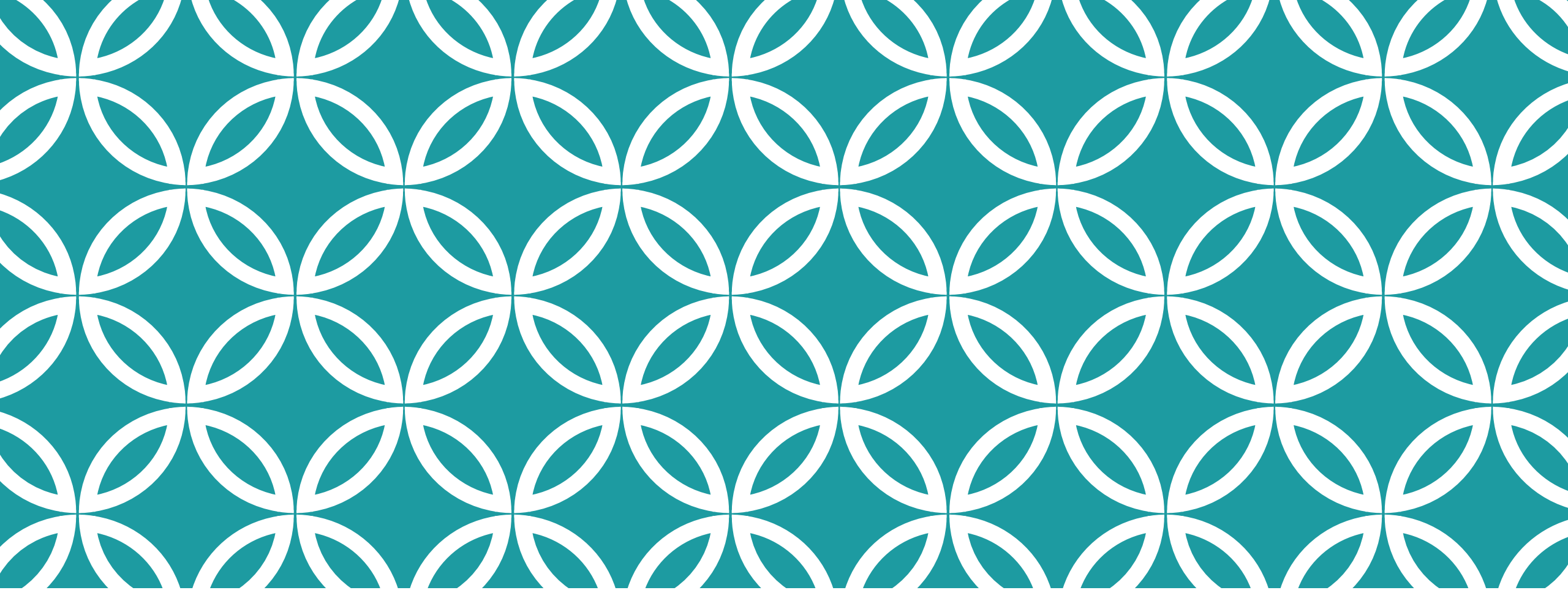


SPRING

Jérémy PERROUAULT



SPRING ASPECT

Spring AOP

PRÉSENTATION AOP

Aspect-Oriented Programming

POO est limité

- On trouve souvent du code technique dans du code métier
  - Journalisation, sécurité, transaction, …
- Ce code technique est dit « préoccupation transversale »
- La maintenance et la réutilisabilité des composants s'en trouve diminuée

PRÉSENTATION AOP

AOP permet de séparer le code métier et le code technique

C’est une surcouche à POO

Un « Aspect » correspond donc une préoccupation transversale

PRINCIPE

Intercepter les méthodes métier et appliquer un aspect associé

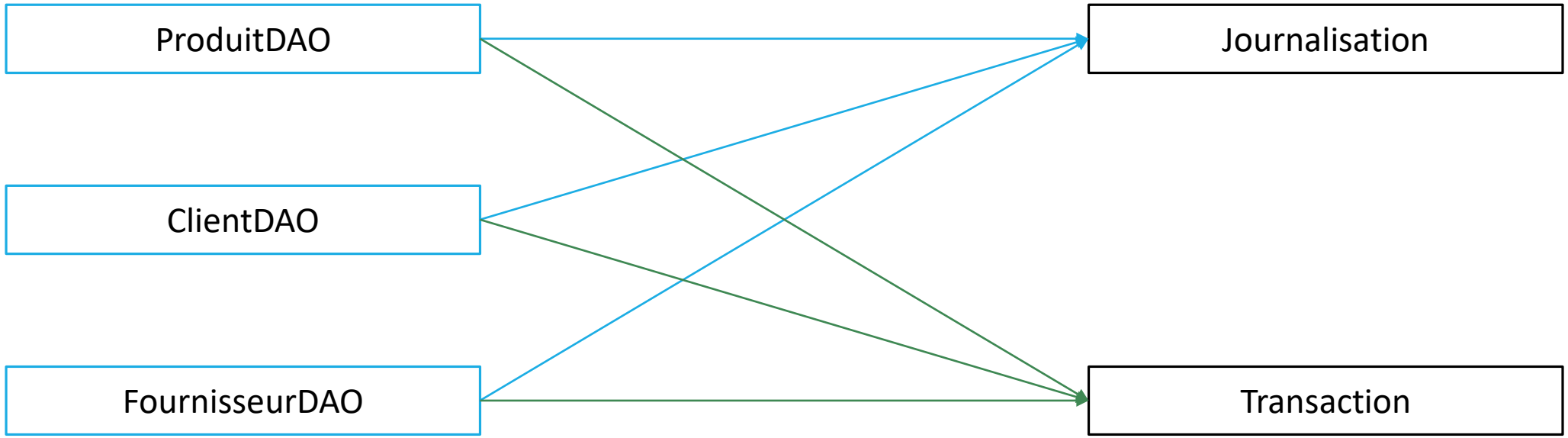
- Intercepter l'ajout d'un utilisateur en base de données pour journaliser cette action

AOP fonctionne grâce à un « tisseur d'aspect »

- Dans le cas de Spring, il s'agit de « Spring AOP »
- Spring AOP s'appuie sur AspectJ

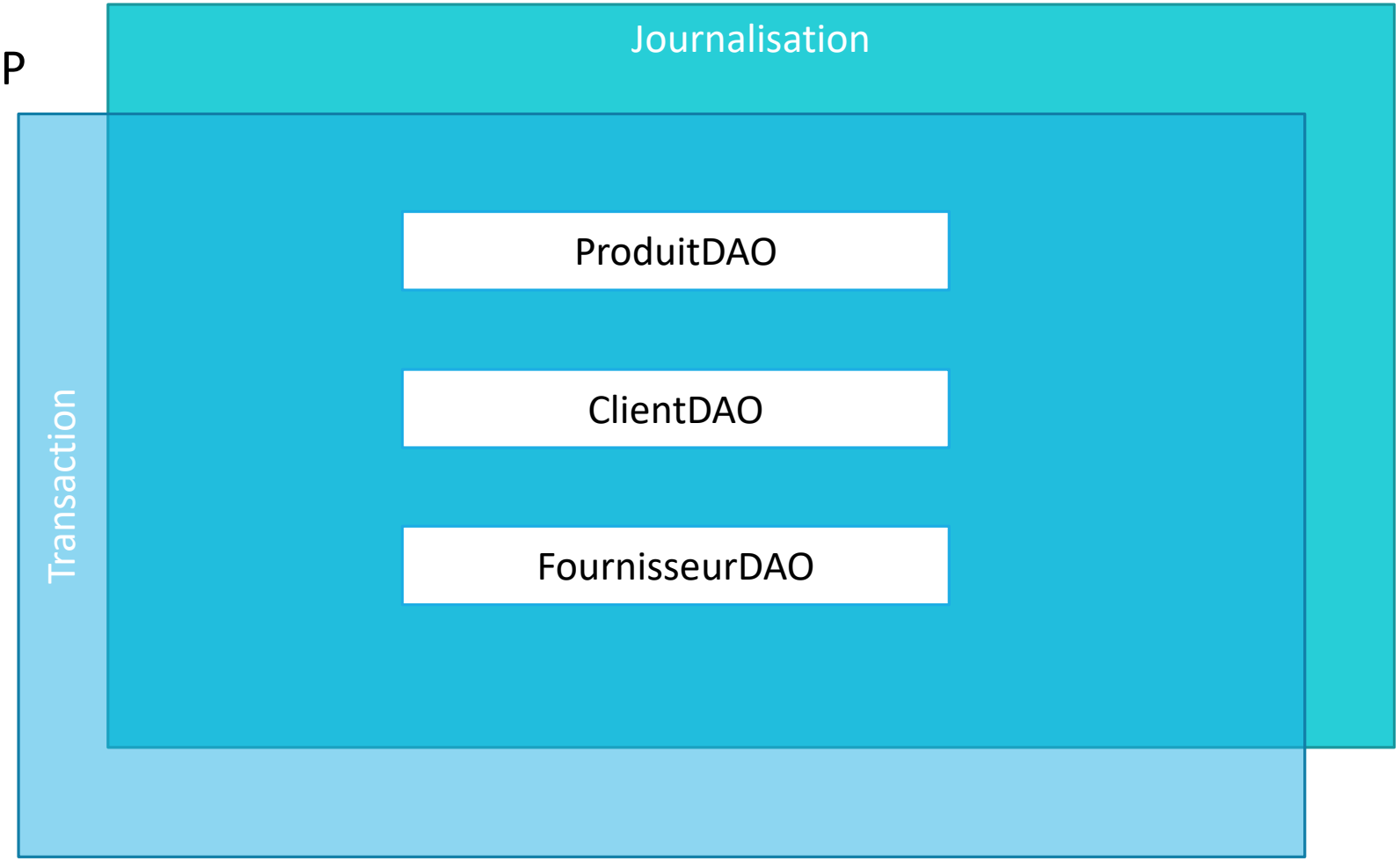
PRINCIPE

Sans AOP



PRINCIPE

Avec AOP



PRINCIPE

Sans AOP

```
public class ProduitDAO {
  public Produit find(int id) {
    log.add("On cherche un produit ...");
    return em.find(Produit.class, id);
  }
}

public class Log {
  public void add(String message) {
    //..Ajouter le message dans un fichier txt
  }
}
```

PRINCIPE

Avec AOP

```
public class ProduitDAO {
  public Produit find(int id) {
    return em.find(Produit.class, id);
  }
}

public class Log {
  public void add(String message) {
    //..Ajouter le message dans un fichier txt
  }
}
```

La méthode « add » de Log va écouter
la méthode « find » de ProduitDAO
et effectuer son action soit avant, soit après

PRINCIPE

Attention, Spring AOP ne fonctionne qu'avec des beans Spring

Dépendances

- **aspectjrt**
- **aspectjweaver**

VOCABULAIRE

AOP	Définition
Aspect	Service, préoccupation transversale
Point de jonction (JoinPoint)	Méthode spécifique (Expression) pour laquelle il est possible d'insérer un greffon (avant ou après la méthode, pas pendant)
Coupe (Pointcut)	Ensemble de points de jonction
Greffon (Advice)	Méthode qui sera activée à un certain point d'exécution, précisé par un point de jonction
Cible (Target)	Objet sur lequel appliquer Aspect
Tissage	Application d'un aspect à une cible

VOCABULAIRE

Greffons	Définition
Before	Exécution avant le point de jonction
After	Exécution après le point de jonction
After-returning	Exécution après, si succès
After-throwing	Exécution après, si échec (Exception levée)
Around	Exécution autour du point de jonction

EXPRESSIONS D'UN POINT DE JONCTION

Les expressions d'un point de jonction

```
execution(<scope de la méthode> <nom_classe>.<methode>(parametres))
```

- Toutes les méthodes publiques de ProduitDAO

```
execution(public fr.formation.dao.ProduitDAO.*(..))
```

- N'importe quelle méthode « find », publique ou privée

```
execution(* *.find(..))
```

CONFIGURATION XML

```
<bean id="monAspectInterceptor" class="fr.formation.aspect.MonAspectInterceptor" />

<aop:config>
  <aop:pointcut id="serviceToStringPointCut" expression="execution(* fr.formation.model.*.toString(..))" />

  <aop:aspect id="monAspect" ref="monAspectInterceptor">
    <aop:before method="interceptToString" pointcut-ref="serviceToStringPointCut" />
    <aop:after-returning method="interceptorToStringReturning"
        returning="result" pointcut-ref="serviceToStringPointCut" />
  </aop:aspect>
</aop:config>
```

EXERCICE

Exécuter une méthode « interceptToString » de « MonAspectInterceptor »

- Après qu'un Guitariste ait exécuté sa méthode toString
- La méthode fait un simple System.out

EXERCICE

Créer une méthode qui

- Retourne une chaine de caractères
- Est capable de lever une Exception

Configurer l’aspect qui intercepte sa valeur de retour

Configurer l’aspect qui intercepte son Exception

Tester

CONFIGURATION PAR ANNOTATION

Activer la configuration Aspect par annotation

- Configuration XML

```
<aop:aspectj-autoproxy />
```

- Configuration Java (sur la classe de configuration)

```
@EnableAspectJAutoProxy
```

CONFIGURATION PAR ANNOTATION

Annoter les classes et les méthodes

Annotation	Description
@Aspect	Configuration de la classe comme Aspect
@Pointcut	Méthode exécutée au point de jonction
@Before	Méthode exécutée avant
@After	Méthode exécutée après
@AfterReturning	Méthode exécutée après, si succès
@AfterThrowing	Méthode exécutée après, si échec
@Around	Méthode exécuter avant et/ou après, ou remplace

CONFIGURATION PAR ANNOTATION

```java
@Component
@Aspect
public class MonAspectInterceptor {
  @Before("execution(* fr.formation.musicien.*.toString())")
  public void interceptToString() {
    System.out.println("Un toString va être appelé !!");
  }

  @AfterReturning(pointcut = "execution(* fr.formation.musicien.*.toString(..))", returning = "result")
  public void interceptToStringReturning(String result) {
    System.out.println("Le toString a retourné : " + result);
  }
}
```

CONFIGURATION PAR ANNOTATION

```java
@Component
@Aspect
public class MonAspectInterceptor {
  @Pointcut("execution(* fr.formation.musicien.*.toString(..))")
  public void intercept() { }


  @Before("intercept()")
  public void interceptToString() {
    System.out.println("Un toString va être appelé !!");
  }


  @AfterReturning(pointcut = "intercept()", returning = "result")
  public void interceptToStringReturning(String result) {
    System.out.println(" Le toString a retourné : " + result);
  }
}
```

CONFIGURATION PAR ANNOTATION

```
@Before("intercept()")
public void interceptToString(JoinPoint joinPoint) {
  if (joinPoint.getTarget() instanceof Pianiste) {
    System.out.println("C’est un pianiste qui va exécuter son toString");
  }

  System.out.println("Un toString va être appelé !!");
}
```

CONFIGURATION PAR ANNOTATION

```
@Around("execution(* *.*(..))")
public void aroundAll(ProceedingJoinPoint proceedingJoinPoint) throws Throwable {
  //Avant la méthode

  //Exécuter la méthode
  //Donc ne pas indiquer cette ligne n'exécutera pas la méthode
  proceedingJoinPoint.proceed();


  //Après la méthode
}
```

CONFIGURATION PAR ANNOTATION

On peut créer une annotation personnalisée

- Annoter les méthodes qu'on veut écouter
- Ecouter sur l'exécution de l'annotation

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MonAnnotation { }
```

```java
@MonAnnotation
public void methodeEcouteeViaAnnotation() {
}
```

```java
@Before("@annotation(fr.formation.annotation.MonAnnotation)")
public void beforeAnnotation() {
  System.out.println("via Mon Annotation !");
}
```

EXERCICE

Remplacer la configuration XML par la configuration par annotation

Créer une annotation personnalisée

Créer une méthode annotée dans **Guitariste**

Créer une méthode qui écoute cette annotation

INJECTER LES ARGUMENTS

Il est possible d'injecter les arguments d'une méthode dans un greffon

- Préciser la liste des arguments dans le point de jonction
- Ajouter le paramètre au greffon

```
@Around("execution(* fr.formation.dao.ProduitDAO.save(..)) && args(produit)")
public void aroundSaveProduit(ProceedingJoinPoint proceedingJoinPoint, Produit produit) throws Throwable {
  //... partie est utilisable ici ...
}
```

- *(version Spring DATA-JPA)*

```
@Around("execution(* org.springframework.data.repository.Repository+.save(..)) && args(produit)")
public void aroundSaveProduit(ProceedingJoinPoint proceedingJoinPoint, Produit produit) throws Throwable {
  //... partie est utilisable ici ...
}
```

EXERCICE

Créer une méthode avec un paramètre

Intercepter la méthode avec un @**Around**

Récupérer le paramètre et sa valeur

- Tester avec un system.out par exemple

EXERCICE

Mise en musique (avec 2 musiciens : un pianiste et un trompettiste)

- Le premier musicien joue un air
  - Avant, le public s'installe
  - Après, le public applaudit
  - S'il y a une fausse note, le public siffle
  - Lorsque le premier musicien a terminé, le public demande au deuxième de jouer : il joue alors

Implémenter cette histoire avec Spring AOP