SPRING – AOP

esprit
Se former autrement

UP ASI
Bureau E204

Plan du Cours

- Définition **AOP**
- **AOP vs IoC** (ID)
- Programmation : Procédurale / Orientée Objets / **Orientée Aspects**
- Principes de l'AOP : **SoC / DRY / Crosscutting Concerns**

- Implémentation : **JoinPoint, PointCut, Advise, Aspect, Weaving**
- Types d'Advise : **Before, After, Around, After Returning, After Throwing**
- **Spring AOP ou AspectJ**

- TP1 **AOP** : Journalisation
- TP2 **AOP** : Performance

AOP

- **AOP** : **Aspect Oriented Programming**, ou **Programmation Orientée Aspect**

- Permet de rajouter des comportements à des classes ou des méthodes existantes
  - Ajouter des traces (logs),
  - Ajouter la gestion des transactions,
  - Ajouter la gestion de la sécurité,
  - Ajouter du monitoring,
  - Il s'agit de problématiques transverses (**Crosscutting concerns**), en général, techniques.

AOP vs IoC (ID)

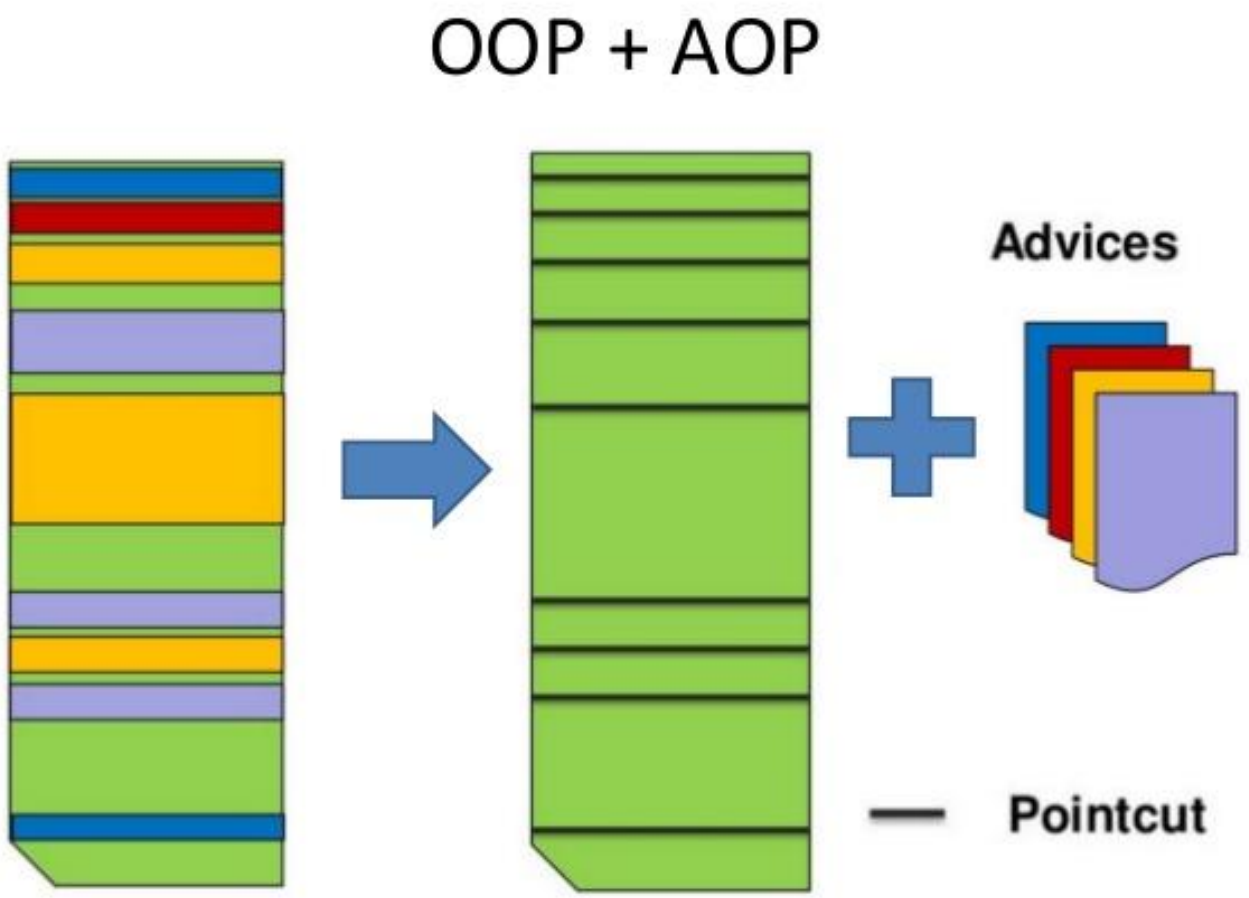
- L'AOP est l'un des deux concepts principaux de Spring (avec l'Inversion de Contrôle IOC – Injection de Dépendance ID)

- IOC ou ID : **Injecte** des Beans dans nos objets Java
- AOP : **Enrichit** nos objets Java

Programmation : Procédurale / Orientée Objets / Orientée Aspects

Modèle	Préoccupation	Elément
Programmation Procédurale	Découper le code en portions	Fonction, procédure
OOP : Programmation Orientée Objets	Données sous forme d'objets	Classe
AOP : Programmation Orientée Aspects	Fonctionnalités transverses Crosscutting Concerns	**Aspect**

Programmation : Orientée Objets / Orientée Aspects



Problématique : Cross Cutting Concerns

- Exemple de “cross cutting concerns” : Tracing, caching, Transaction, security, performance monitoring, Error handling.

- Impact des “cross-cutting concerns” sur notre système (sans AOP) :
  - **Tangling** : Mélange du code métier avec du code de (sécurité, traçage ...).
  - **Scattering** : Duplication d’un bout de code dans plusieurs endroits. (tel que test de sécurité).

- ⇒ Problème de **maintenabilité**.
- L’AOP peut être une solution à ces problèmes (Tangling, Scattering).

SoC / DRY / Crosscutting Concerns

- Separation of Concerns (**SoC**) : Au lieu d'avoir un appel direct à un module technique depuis un module métier, en AOP, le code du module en cours de développement est concentré sur le but poursuivi (la logique métier).
- L'AOP permet d'enrichir ce code métier avec ces fonctionnalités transversale (**Crosscutting Concerns**).
- Exemple : Ajout de logs dans une application existante).

- Don't Repeat Yourself (**DRY**) : Cela évite la duplication de code.

Avantages et Inconvénients de l'AOP

- Les avantages sont :
  - Facilité de **maintenance**, puisque les fonctionnalités transverses sont regroupées dans les aspects.
  - Particulièrement adapté pour les fonctionnalités techniques
  - Permet une meilleure **modularité** du code et des applications ce qui augmente la **réutilisation** du code et la modularité des systèmes.

- Les inconvénients sont :
  - La lecture du code contenant les traitements ne permet pas de connaître les aspects qui seront exécutés (sans utiliser un outil).
  - Nécessite un temps de prise en main.

Implémentation de l'AOP

- L'AOP peut être utilisée :
  - Indirectement, lors de l'utilisation des annotations Spring, tel que, @Configuration et @Transactional.
  - Directement, pour mettre en œuvre ses propres **Aspects** : Spring facilite alors cette mise en œuvre

- L'AOP peut être mise en œuvre via **Spring AOP** ou AspectJ (qui a sa propre syntaxe).

- L'AOP utilise le Design Pattern **Proxy**.
  - Un proxy est une classe se substituant à une autre classe. Le proxy implémente la même interface que la classe à laquelle il se substitue.
  - Dans notre cas, Spring va créer une classe "**proxy**" qui implémente **IStockService** et va l'injecter à la place du bean "**StockServiceImpl**".
  - Cette classe proxy contient les aspects et les méthodes de l'interface.

Implémentation de l'AOP

- **Join point** : L'endroit où l'on veut qu'un aspect s'applique; comme l'appel d'une méthode ou le lancement d'une exception
- **Pointcut** : Une expression, qui permet de sélectionner plusieurs Join points. Par exemple, «toutes les méthodes public dans un package précis».
- **Advice** : Le code que l'on veut rajouter. On peut ajouter ce code avant, après, autour de la méthode...
- **Aspect** : Une classe qui encapsule une fonctionnalité transverse et elle est composée d'un ou de plusieurs **Pointcut** et **Advice**. La classe est annotée **@Aspect**.
- **Weaving** (tissage) : action d'insertion des aspects (Fait par Spring AOP).

Types d'Advise

- Il est possible de définir 5 types d'advices
- **Before advice** : s'exécute avant le Join point. S'il lance une Exception, le Join point ne sera pas appelé
- **After returning advice** : s'exécute après le Join point, si celui-ci s'est bien exécuté (s'il n'y a pas d'Exception)
- **After throwing advice** : s'exécute si une Exception a été lancée pendant l'exécution du Join point
- **After advice** : s'exécute après le Join point, qu'il y ait eu une Exception ou non
- **Around advice** : s'exécute autour du Join point. C'est l'advice le plus puissant.

Advise

- `"execution(Modifiers-pattern? Ret-type-pattern Declaring-type-pattern?Name-pattern(param-pattern) Throws-pattern?)"`

- **"?"** veut dire optionnel
- **Modifiers-pattern?** : public, private ...
- **Ret-type-pattern** : le type de retour.
- **Declaring-type-pattern?** : nom de la classe y compris le package.
- **Name-pattern** : nom de la méthode.
- **Throws-pattern?** : l'exception.
- **".."** veut dire, 0 ou plusieurs paramètres

Exercice Advise

- Indiquer quel est **l'Aspect, le JoinPoint et le PointCut, l'Advise et le type d'advise** :
- Attention : l'une de ces notions ci-dessus ne se trouve pas dans le code ci-dessous, laquelle? :

```
package tn.esprit.esponline.config;
import ……

@Component
@Aspect
public class LoggingAspect {

private static final Logger logger = LogManager.getLogger(LoggingAspect.class);

@Before("execution(* tn.esprit.esponline.service.*.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.info("In method " + name + " : ");
}

}
```

Exercice Advise

- Expliquer les **PointCut** suivants :

- ```
  @Before("execution(* tn.esprit.esponline.service.*.*(..))")
  ```

- ```
  @Before("execution(public * *(..))")
  ```

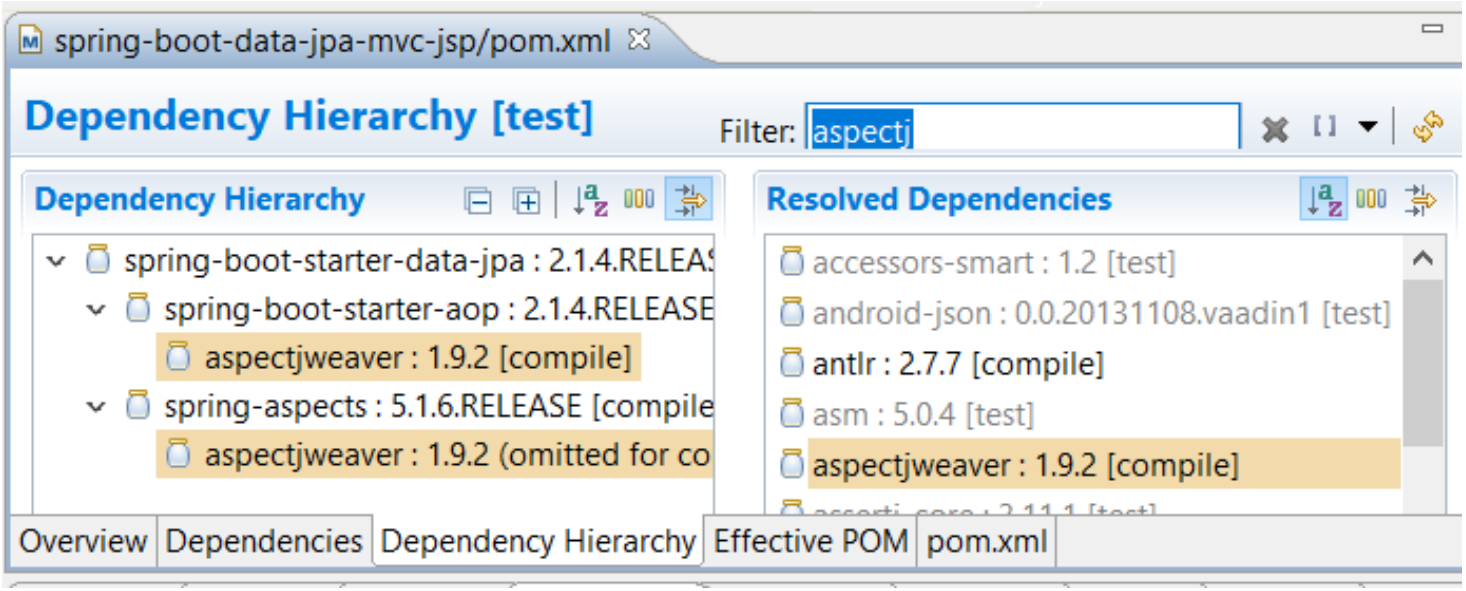
- ```
  @Before("execution(* set*(..))")
  ```

- ```
  @Before("execution(* tn.esprit.esponline..*.*(..))")
  ```

TP1 : Logs

- **Nous allons nous appuyer sur le projet existant et l'enrichir avec des Aspects (Journalisation ou Logs) :**

- La dépendance **aspectjweaver est déjà fournie par Spring Boot grâce au starter spring data JPA. Pas besoin de l'ajouter à vote pom.xml :**



TP1

- Classe de configuration BeansConfiguration.java :

  Ajouter l'annotation **@EnableAspectJAutoProxy** sur la classe main. Elle permet l'activation de Spring AOP :

```
@SpringBootApplication
@EnableAspectJAutoProxy
public class SpringBootDataJpaMvcJspApplication {
public static void main(String[] args) {
SpringApplication.run(SpringBootDataJpaMvcJspApplication.class, args);
}}
```

TP1

- Ajouter un Aspect de Journalisation (logs) à la classe service StockService et ré-exécuter le test unitaire : Dans le package config, ajouter l'aspect :

```
package tn.esprit.esponline.config;
@Component
@Aspect
public class LoggingAspect {
private static final Logger l = LogManager.getLogger(LoggingAspect.class);
@Before("execution(* tn.esprit.esponline.service.StockServiceImpl.*(..))")
public void logMethodEntry(JoinPoint joinPoint) {
String name = joinPoint.getSignature().getName();
logger.info("In method " + name + " : ");
}
@After("…….")
public void logMethodExit….
}
```

TP1

- Exécuter le test unitaire (dans le dossier src/test/java, vous avez des déjà une clase de tests unitaires dédié à la partie Stock):

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class StockServiceImplTest {

@Autowired
IStockService stockService;

@Test
public void testAddStock() throws ParseException {
……
}
```

- 2020-03-30 13:02:48 - INFO - tn.esprit.spring.service.UserServiceImpl - In method : addStock:
- 2020-03-30 13:02:49 - INFO - tn.esprit.spring.service.UserServiceImpl - Out of method : addStock

TP2 : Mesure de Performance

- En vous inspirant du TP1, créer un nouvelle classe PerformanceAspect, créer un aspect qui permet de calculer et afficher dans les logs, la durée d'exécution de chaque méthode appelée de la couche service.
- Utiliser la méthode :

```
@Around("execution(* tn.esprit.spring.service.*.*(..))")
public Object profile(ProceedingJoinPoint pjp) throws Throwable {
long start = System.currentTimeMillis();
Object obj = pjp.proceed();
long elapsedTime = System.currentTimeMillis() - start;
l.info("Method execution time: " + elapsedTime + " milliseconds.");
return obj;
}
```

Si vous avez des questions, n'hésitez pas à nous contacter :

Département Informatique
UP ASI

Bureau E204