

# Le fonctionnement de Spring ; Introspection et Programmation Orientée Aspect

GLG 203/Architectures Logicielles Java

Serge Rosmorduc  
`serge.rosmorduc@lecnam.net`  
Conservatoire National des Arts et Métiers

2019–2020

# Démonstrations

[https://gitlab.cnam.fr/gitlab/glg203\\_204\\_demos/12\\_spring\\_aop.git](https://gitlab.cnam.fr/gitlab/glg203_204_demos/12_spring_aop.git)

# Introduction

Spring pour fonctionner utilise énormément :

- les annotations (vues au second cours) ;
- l'introspection :
  - ▶ grâce aux classes `Class`, `Method`, etc. de Java ;
  - ▶ grâce à l'exploration du classpath  
(`ClassPathScanningCandidateComponentProvider`) ;
- grâce à la **Programmation orienté aspect**

# Introspection - la classe Class

## Définition

Possibilité d'explorer par programme la liste des méthodes d'une classe ou d'un objet et de les appeler.

Utilise la classe `Class<?>` ; instanciée :

- soit par chargement à partir du **nom** de la classe :

```
Class<?> clazz = Class.forName("glg203.Exemple");
```

- directement :

```
Class<Exemple> clazz = Exemple.class;
```

- à partir d'un objet :

```
Class<?> clazz = o.class;
```

**remarque** : variable nommée `clazz` parce que `class` est un mot-clef !

# Utilisation de la classe

`Method` `getMethod(String name, Class<?>... parameterTypes)`

récupère dans l'objet class courant la méthode nommée name, dont les paramètres sont définis par parameterTypes ;

`Method[]` `getMethods()`

Récupère les méthodes de this.

`Annotation[]` `getDeclaredAnnotations()`

Récupère les annotations qui portent sur la classe

La classe `Method` dispose elle aussi de diverses méthodes comme `getDeclaredAnnotations`.

# Introspection, suite

```
Class<?> clazz = Class.forName("introspection.Personne");
// On crée un objet personne, avec le constructeur par défaut :
Object personne = clazz.newInstance();
// On liste les méthodes, repère les getters, et on les utilise pour
afficher
// notre objet :
for (Method m : clazz.getMethods()) {
    // Si m est un "getter" :
    if (m.getName().startsWith("get")
        && m.getParameterCount() == 0) {
        // Nom de la propriété (après "get")
        String propriete = m.getName().substring(3);
        // appel de personne.getXX();
        Object res = m.invoke(personne);
        // Affichage
        System.out.println("propriete " + propriete + " : " + res);
    }
}
```

# Programmation Orientée Aspect/ Aspect Oriented Programming

Un certain nombre d'opérations se superposent systématiquement à l'algorithme principal d'un programme, entraînant une programmation très répétitive :

- vérification des droits d'accès ;
- log des opérations ;
- ouverture/fermeture de transactions...
- envoi de messages (pattern observateur) ...

```
public void sauver(Personne p) {  
    log.info("sauver[" + p + "]"); // code répétitif  
    repository.save(); // "vrai" algorithme  
    log.info("sauver[" + p + "] effectué"); // autre code  
}
```

## Définition

Ces opérations qui se superposent au code principal sont les **préoccupations transverses** (*cross-cutting concerns*).

# En première approximation

- La programmation orienté aspect permet d'ajouter automatiquement du code avant et après certains appels de méthodes
- Similaire au système de *filters* qu'on trouve dans la couche Web...



# Intérêt de la programmation aspect

- sans les aspects, le code liés aux *préoccupations transverses* est :
  - ▶ soit répétitif (log, sécurité...)
  - ▶ soit intrusif — mise en place *explicite* de filtres par exemple (ce qui revient à faire de l'AOP « à la main »)
- L'AOP permet d'écrire le code sans se soucier (en théorie!) des *préoccupations transverses*
- ...et de les ajouter et traiter *systématiquement* et de manière *déclarative*.

# Outils derrière l'implémentation bas niveau de l'AOP

- par introspection : la classe `java.lang.reflect.Proxy` ;
- par création de bytecode java à la volée : la bibliothèque `CGLib` ;

# La classe Proxy

voir projet proxy

Classe standard de Java qui permet de créer *dynamiquement* une implémentation pour une interface quelconque.

→ ne fonctionne que si on a une interface à notre disposition.

On utilise la méthode `newProxyInstance` :

`Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler handler)`

**loader** objet en charge de gérer et charger les classes en mémoire ;

**interfaces** liste des interfaces à implémenter ;

**handler** l'objet qui fait tout le travail : on définit sa méthode `invoke`

## Exemple : définition d'un DTO à partir d'une Map et d'une interface

- une interface fournit des getters et des setters ;
- on peut l'implémenter « mécaniquement » en utilisant une Map
- et on peut créer un proxy qui le fait par introspection...

# Le handler

```
class ImplementationHandler implements InvocationHandler {
    Map<String, Object> map = new HashMap<>();
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        if (method.getName().startsWith("get"))
            return get(proxy, method.getName().substring(3));
        else if (method.getName().startsWith("set"))
            return set(proxy, method.getName().substring(3), args);
        else // un peut trop brutal...
            throw new UnsupportedOperationException(method.getName());
        //(on aurait equals et quelques autres à définir)
    }

    Object set(Object proxy, String propName, Object[] args) {
        return map.put(propName, args[0]);
    }

    Object get(Object proxy, String propName) {
        return map.get(propName);
    }
}
```

# Le constructeur d'objets...

```
public class DTOFactory {  
  
    @SuppressWarnings(value = "unchecked")  
    public static <T> T creerObjet(Class<T> clazz) {  
        ClassLoader currentClassLoader =  
            Thread.currentThread().getContextClassLoader();  
        Class<?> classes[] = { clazz };  
        return (T) Proxy.newProxyInstance(currentClassLoader, classes,  
                                           new ImplementationHandler());  
    }  
}
```

- la classe Proxy va créer un proxy qui implémente l'interface clazz;
- chaque objet créé sera réellement implémenté par un ImplementationHandler.

# Utilisation

```
public interface IPersonne {  
    void setNom(String nom);  
    void setPrenom(String prenom);  
    String getNom();  
    String getPrenom();  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        IPersonne p1 = DTOFactory.creerObjet(IPersonne.class);  
        p1.setNom("toto1");  
        p1.setPrenom("foo");  
        System.out.println(p1.getNom());  
        System.out.println(p1.getPrenom());  
    }  
}
```

# Utilisation de proxy pour créer un décorateur

→ pattern **Décorateur**.

```
class LogDecoratorImpl<T> implements InvocationHandler {  
    private T object;  
    private PrintWriter out;  
  
    public LogDecoratorImpl(T object, PrintWriter out) {  
        this.object = object;  
        this.out = out;  
    }  
  
    public Object invoke(Object proxy, Method method, Object[] args)  
        throws Throwable {  
        out.println("appel de " + method.getName());  
        out.flush();  
        Object res = method.invoke(object, args);  
        return res;  
    }  
}
```



# Création du décorateur

```
@SuppressWarnings("unchecked")
public static <T> T creerLogDecorator(T cible, PrintWriter out) {
    ClassLoader loader =
        Thread.currentThread().getContextClassLoader();
    // Les interfaces implémentées par "cible"
    Class<?> classes[] = cible.getClass().getInterfaces();
    // On retourne un proxy qui utilise "cible" et écrit sur "out"
    return (T) Proxy.newProxyInstance(loader, classes,
        new LogDecoratorImpl<T>(cible, out));
}
```

# Utilisation du décorateur

## Main

```
ICompteur c = LogDecoratorFactory.creerLogDecorator(  
    new CompteurImpl(),  
    new PrintWriter(System.out)  
);  
c.setValeur(10);  
c.incr();  
int res= c.getValeur();
```

## Sortie

```
appel de setValeur  
appel de incr  
appel de getValeur
```

Le Proxy ne peut fonctionner que si les classes utilisées implémentent des interfaces.

## CGLib et ASM

La bibliothèque de haut niveau CGLib utilise la bibliothèque ASM qui génère du bytecode à la volée. Elle permet d'ajouter ou de modifier des comportements à un objet de manière dynamique.

# Décorateur/Logger en CGLib

```
public class LoggerFactory {
    public static <T> T creerLogger(T object, PrintWriter out) {
        // Le Enhancer sert à créer nos décorateurs.
        Enhancer enhancer = new Enhancer();
        enhancer.setSuperclass(object.getClass());
        enhancer.setCallback(new LoggerCallback(out));
        return (T)enhancer.create();
    }
    private static class LoggerCallback implements MethodInterceptor {
        private final PrintWriter out;
        public LoggerCallback(PrintWriter out) {
            this.out=out;
        }
        @Override
        public Object intercept(Object obj, Method method, Object[] args,
                                MethodProxy proxy) throws Throwable {
            out.println("Appel de "+ method.getName()+ Arrays.asList(args));
            // appel de l'objet d'origine.
            return proxy.invokeSuper(obj, args);
        }
    }
}
```

# Notions de programmation aspect

Rappel : le but est d'ajouter *dynamiquement* des traitements supplémentaires à certains endroits précis de classes existantes ;

On a besoin :

- de dire *où* ces traitements seront ajoutés ;
- de décrire les traitements en question

# Vocabulaire d'AOP

**Point de jonction/Join point** un endroit où on peut ajouter un traitement :  
à l'appel d'une méthode, à la levée d'une exception...

**Point de coupe/Pointcut** un sélecteur qui désigne un ou plusieurs point de jonction précis dans un programme (par exemple : « tous les appels de méthode sur la classe A » ; « l'appel d'une méthode nommée f » ...)

**Greffon/Advice** un code associé à un point de coupe précis ;

**Aspect** groupes de greffons (liés par un thème commun).

**Tissage/Weaving** opération de lier les greffons au programme.

# Spring et l'AOP

- Spring fournit des fonctionnalités d'Aspect Oriented Programming en utilisant Proxy ou CGLib ;
- il couvre un sous-ensemble d'AspectJ, extension orientée aspect de Java ;
- si besoin, on peut recourir à AspectJ ;
- on utilise (surprise !) soit des annotations, soit des déclarations XML.

# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }

}
```



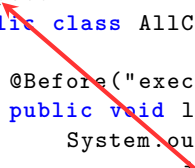
# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```



Ceci est un aspect : un ensemble de greffons qui s'attacheront au code de l'application.

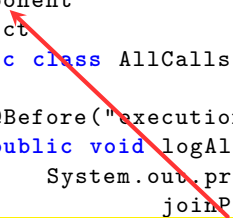
# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```



Les aspects en Spring doivent être des beans (@Component, @Bean...)

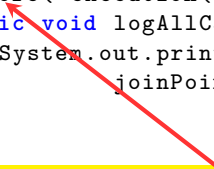
# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```



Un greffon est la donnée d'un point de coupe et d'un code à exécuter

# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```

**@Before** marque un greffon de type  
« before » (à exécuter avant l'appel).

# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```

Le point de coupe correspond à l'appel de méthodes du package glg203 et de ses descendants.

# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```

On passe au greffon un argument qui permet d'extraire facilement les informations sur la méthode interceptée...


# Un exemple simple

On met ici en place un aspect qui trace tous les appels :

- de méthodes sur des **composants** Spring (et seulement eux) ;
- placés dans le package glg203.

```
@Component
@Aspect
public class AllCalls {

    @Before("execution(* glg203..*(..))")
    public void logAllCalls(JoinPoint joinPoint) {
        System.out.println("log appels " +
            joinPoint.getSignature().getName());
    }
}
```



ici, joinPoint.getSignature().getName()  
est le nom de la méthode interceptée.

# Types de greffons

Les méthodes associées au greffons suivants *peuvent* prendre comme argument un `JoinPoint` ; les annotations prennent a priori comme argument une annotation `pointcut`

**@Before** appelé avant l'appel d'une méthode :

```
@Before(pointcut="execution(* glg203..*(..))")
```

**@After** appelé après l'appel d'une méthode (qu'elle se termine normalement ou qu'elle lève une exception) ;

**@AfterReturning** appelé quand une fonction renvoie un résultat.

**@AfterThrowing** appelé après la levée d'une exception ;



## @AfterReturning

On veut souvent pouvoir consulter la valeur renvoyée.

- la méthode qui implémente le greffon prend alors un argument du type de la valeur de retour ;
- on précise le nom de l'argument dans @AfterReturning avec l'attribut `returning` :

```
@AfterReturning(  
    pointcut = "execution(glg203.aop.PersonneDTO glg203..*(..))",  
    returning = "value")  
public void readPersonneAdvice(PersonneDTO value) {  
    System.out.println("** On retourne " + value.getNom() + " ");  
}
```

## @AfterThrowing

Ici, on peut nommer l'exception pour la recevoir comme paramètre.

```
@AfterThrowing(  
    throwing = "ex",  
    pointcut = "execution(* glg203..*(..))")  
public void ListeExceptions(Exception ex) {  
    System.out.println("** on a vu l'exception " +ex);  
}
```

# @Around

- Un greffon @Around englobe un appel ;
- il peut éventuellement décider de la poursuite ou non du traitement ;
- Il peut être utilisé :
  - ▶ pour ouvrir *et* refermer des ressources ;
  - ▶ mais aussi pour gérer la sécurité ;
  - ▶ éventuellement pour modifier la valeur de retour d'une méthode (l'encapsuler dans un proxy, par exemple).

Le greffon @Around reçoit le point de jonction non sous forme d'un objet `joinPoint` mais de `ProceedingJoinPoint`

## Exemple : vérification de connexion

limite l'accès aux services aux utilisateurs connectés.

```
@Component
@Aspect
public class SecurityAspect {
    @Autowired UserConnexion connexion;

    @Around("execution(* glg203.*Service.*(..))")
    public Object checkSecurity(ProceedingJoinPoint joinPoint)
        throws Throwable {
        if (connexion.estConnecte()) {
            return joinPoint.proceed();
        } else {
            throw new UtilisateurNonConnecteException();
        }
    }
}
```

## Exemple : vérification de connexion

limite l'accès aux services aux utilisateurs connectés.

```
@Component
@Aspect
public class SecurityAspect {
    @Autowired UserConnexion connexion;

    @Around("execution(* glg203...Service.*(..))")
    public Object checkSecurity(ProceedingJoinPoint joinPoint)
        throws Throwable {
        if (connexion.estConnecte()) {
            return joinPoint.proceed();
        } else {
            throw new UtilisateurNonConnecteException();
        }
    }
}
```




Un greffon @Around a besoin de prendre comme argument un ProceedingJoinPoint

## Exemple : vérification de connexion

limite l'accès aux services aux utilisateurs connectés.

```
@Component
@Aspect
public class SecurityAspect {
    @Autowired UserConnexion connexion;

    @Around("execution(* glg203...Service.*(..))")
    public Object checkSecurity(ProceedingJoinPoint joinPoint)
        throws Throwable {
        if (connexion.estConnecte()) {
            return joinPoint.proceed();
        } else {
            throw new UtilisateurNonConnecteException();
        }
    }
}
```



parce qu'il faut qu'il lance  
explicitement la méthode qu'il a  
interceptée

# Ordre des greffons

- Quand un plusieurs greffons peuvent s'appliquer à un point de jonction, ils s'appliquent *tous*
- On peut annoter les aspects (**les classes**) par l'annotation `@Order`, qui prend comme attribut un entier ;
- plus l'ordre est *petit*, plus tôt le greffon s'exécute ;
- si une classe d'aspect est annotée avec `@Order(1)`, ses greffons s'exécutent avant ceux d'une classe annotée par `@Order(100)`.

# Définition des points de coupe

- L'argument `pointcut` des greffons utilise un langage spécifique, avec des expressions qui peuvent se combiner entre elles ;
- il est possible de définir des points de coupe et de les nommer pour alléger l'écriture ;
- les principaux éléments de ce langage sont :
  - `execution` correspond à l'exécution d'une méthode ;
  - `within` limite le point de coupe aux classes d'un package ;
  - `args` permet de récupérer facilement les arguments de la méthode appelée ;
- on peut les combiner avec des opérateurs comme « et » (`&&`), « ou » (`||`) et « non » (`!`), et utiliser des parenthèses ;
- les noms de classe et de méthode **doivent a priori comporter les noms de packages.**



# execution

Forme :

```
@Before("execution(* glg203..*.readById(Long))")
```

package et méthode concernée. Ici, toute méthode `readById` dans un composant du package `glg203`.

type de retour des appels reconnus par le point de coupure. Ici, tout type `(*)`.

# langage de description des pointcuts

« \* »

« \* » : « wildcard » : reconnaît n'importe quel élément :

```
* glg203.aop.*Repository.*(..)
```

= n'importe quelle méthode d'une classe dont le nom se termine par `Repository` dans le package `glg203.aop` et qui retourne n'importe quoi ;

« .. »

« .. » reconnaît une suite d'éléments (arguments de fonctions, packages...)

```
* glg203...*(..)
```

= N'importe quelle méthode d'une classe dans un package qui descend de `glg203`, et qui prend n'importe quels arguments.

Simplifie la vie en nommant les arguments :

```
@Before("args(id) && execution(* glg203...*.readById(Long))")
public void beforeFind( JoinPoint joinPoint, Long id) {
    System.out.println("on appelle readById "+id+ " dans "
        + joinPoint.getSignature().getDeclaringTypeName());
}
```

- args(id) : l'argument de la méthode intercepté sera copié dans l'argument « id » du greffon

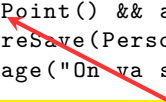
# Nommage des points de coupe

- Il est possible de nommer les points de coupe (pour les réutiliser ou simplement améliorer la lisibilité ;
- on utilise l'annotation @Pointcut sur une méthode vide ;
- le nom de la méthode sert de nom au point de coupe.

```
@Component @Aspect @Order(10)
public class ServiceAdviceLevel10 {
    @Autowired AOPLog aopLog; // notre classe de log.

    @Pointcut("execution(* glg203.aop.PersonneRepository.save(..))")
    private void saveCutPoint() {}

    @Before("saveCutPoint() && args(personne)")
    public void beforeSave(Personne personne) {
        aopLog.addMessage("On va sauver " + personne.getNom());
    }
}
```



On utilise le point de coupe  
« saveCutPoint » défini juste au  
dessus.

# Généricité et AOP

Attention : comme à l'exécution, on ne connaît pas les types paramétriques, les types des paramètres génériques apparaîtront comme `Object` (ou `*`, ou `..`) dans les règles.

```
@Pointcut(  
    "execution(* glg203..PersonneRepository.save(Object))")  
private void saveCutPoint() {}
```

fonctionne.

Mais on ne peut pas remplacer `Object` par

`glg203.aop.model.Personne`

car le repository est générique.