*Université Montpellier-II*
*UFR des Sciences - Département Informatique*
*Master Informatique - Génie Logiciel*

# Programmation par Aspects (AOP), ou la Réutilisation d'entités Transverses (*"Cross-cutting Reuse"*)

*Notes de cours*
*Christophe Dony*

# 1 Introduction

## 1.1 Un problème général : fonctionnalités orthogonales (cross-cutting concerns) et codes enchevêtrés (tangled)

> "We have found many programming problems for which neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. This forces the implementation of those design decisions to be scattered throughout the code, resulting in "tangled" code that is excessively difficult to develop and maintain. We present an analysis of why certain design decisions have been so difficult to clearly capture in actual code. We call the properties these decisions address aspects, and show that the reason they have been hard to capture is that they cross-cut the system's basic functionality ..."
>
> kiczales&al'97-Aspect-Oriented Programming

```
1  public class BankAccount {
2    private int balance = 1000; //cool !
3    private Semaphore semaphore = new Semaphore(1);
4  ...
5    public void withdraw(int amount) {
6      try { semaphore.acquire(); //fonctionnalité 2 : gérer une section critique
7            if (amount > 0 && balance >= amount)
8              balance = balance - amount; //fonctionnalité 1 : code métier
9            else throw new RuntimeException("Insuffisant Amount");
10     } catch (InterruptedException e) { e.printStackTrace();
11     } finally { // Release the permit, exiting the critical section
12           semaphore.release();  } }
13   ...
14   public static void main(String[] args) {
15     BankAccount account = new BankAccount();
16       for (int i = 0; i < 5; i++) {
17         Thread thread = new Thread(() -> { account.withdraw(300); });
18         thread.start(); } } }
```

***Listing (1)*** *– un exemple : fonctionnalités orthogonales (cross-cutting concern) et codes enchevêtrés (tangled)*

## 1.2 Idée : séparer les codes des différentes fonctionnalités

Bonne pratique (lisibilité, test, réutilisabilité) : séparation du code métier et des codes réalisant des fonctionnalités orthogonales (ou annexes ou transverses).

Comment ?

Parfois une option ad.hoc. du langage offre une solution.

Exemple en Java, le mot-clé synchronized traite l'exemple précédent.

```
1  public class BankAccount {
2    private int balance = 1000; //cool !
3
4    public void synchronized withdraw(int amount) { //fonctionnalité 2 : section critique
5      if (amount > 0 && balance >= amount)
6        balance = balance - amount; //fonctionnalité 1 : code métier
7        else throw new RuntimeException("Insuffisant Amount"); }
```

### 1.3 Généralisation de l'idée

Permettre de façon systématique la séparation du code métier et du code des fonctionnalités "orthogonales".

Article fondateur : Aspect-Oriented Programming, *Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin - European Conference on Object-Oriented Programming, 1997.*

Concepts : **Aspect** (aspect), **Advice** (conseil, règle à suivre, règle à appliquer), **Pointcut** (Point de coupe), **Jointpoint** (Point de jonction).

Mises en oeuvre et variations : *AsjectJ*, *Spring-AOP*, **Annotations**.

### 1.4 Programmation par aspects : méta-programmation

Méta-Programmation : programmation au niveau standard des entités définies à un niveau méta.

La programmation par aspects est une forme de méta-programmation quand elle permet de modifier ou d'enrichir certaines entités du méta-niveau (classes, méthodes, etc).

Définir un aspect "synchronization", comme celui utilisé dans l'exemple précédent, peut être comparé à la réalisation, dans un langage réflexif autorisant l'extension de son niveau méta, d'une nouvelle méta-classe *synchronized-method*.

## 2 La programmation par aspects avec AspectJ : présentation par l'exemple

### 2.1 Exemple de séparation des préoccupations (*separation of concerns*)

Exemple [1] : objets d'une application graphique, instances des classes `Line` et `Point`, sous-classes de `Figure`.

```
1  class Line extends Figure { // version 1
2    private Point p1, p2;
3    Point getP1() { return p1; }
4    Point getP2() { return p2; }
5    void setP1(Point p1) { p1 = p1; }
6    void setP2(Point p2) { p2 = p2; } }
7
8  class Point extends Figure { // version 1
9    private int x = 0, y = 0;
10   int getX() { return x; }
11   int getY() { return y; }
12   void setX(int x) { this.x = x; }
13   void setY(int y) { this.y = y; } }
```

Considérons le problème consistant à modifier ces classes afin qu'il soit possible de savoir si une de leurs instances a été modifiée via les actions d'un utilisateur de l'application.

1. tiré de http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html

Voici un exemple de solution à ce problème en programmation standard.

D'une part une classe spécifique est définie :

```
class MoveTracking { // Version 2
  private static boolean flag = false;

  public static void setFlag() { flag = true;}

  public static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;}
}
```

D'autre part les classes `Line` et `Point` sont modifiées de la façon suivante :

```
class Line { // Version 2
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) { p1 = p1; MoveTracking.setFlag(); }
  void setP2(Point p2) { p2 = p2; MoveTracking.setFlag(); } }
```

```
class Point{ // Version 2
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
    MoveTracking.setFlag(); }
  void setY(int y) {
    this.y = y;
    MoveTracking.setFlag(); } }
```

Cette solution a deux inconvénients importants :

1. Elle nécessite de modifier le code des classes `Point` et `Line`
2. Elle ne supporte pas l'évolution. Par exemple si l'on souhaite maintenant mémoriser l'historique des déplacements, il faut d'une part modifier la classe `MoveTracking`, ce qui est admissible mais également modifier à nouveau les classes `Point` et `Line`, comme indiqué dans la version 3 ci-après.

```
class MoveTracking{ //Version 3

  private static Set deplacements = new HashSet();

  public static void collectOne(Object o) {
    deplacements.add(o); }

  public static SetgetDeplacements() {
     Set result = deplacements;
     deplacements = new HashSet();
     return result; } }
```

```
class Line { //Version 3
```

```
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) { p1 = p1; MoveTracking.collectOne(this); }
  void setP2(Point p2) { p2 = p2; MoveTracking.collectOne(this); } }

class Point{ //Version 3
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) {
    this.x = x;
    MoveTracking.collectOne(this); }
  void setY(int y) {
    this.y = y;
    MoveTracking.collectOne(this); } }
```

## 2.2 Programmation par Aspect

### 2.2.1 Aspect

Construction permettant de représenter une fonctionnalité orthogonale à un programme,

par exemple, la fonctionnalité “détection d’un déplacement” ou “mémorisation de l’historique des déplacements” de l’exemple précédent.

Voici un aspect (version 1) écrit en *AspectJ* qui permet de réaliser la fonctionnalité orthogonale précédente (détection d’un déplacement) sans modification du code métier original.

```
aspect MoveTracking { // Aspect version 1
   private static boolean flag = false;
   public static boolean testAndClear() {
      boolean result = flag; flag = false; return result; }

   pointcut moves():
      call(void Line.setP1(Point)) ||
      call(void Line.setP2(Point));

   after(): moves() {
      flag = true; } }
```

Question : qui invoque `testAndClear()` et quand ? (voir *advices*)

### 2.2.2 Point de Jonction

Points dans l’exécution d’un programme où l’on souhaite réaliser une fonctionnalité orthogonale.

Un *point de jonction* dans l’exemple précédent :

```
call(void Line.setP1(Point))
```

Le langage de définition des points de fonction accepte les caractères de filtrage.

Le point de jonction suivant désigne tout appel d’une méthode publique de la classe `Figure`.

```
call(public * Figure.* (..))
```

### 2.2.3 Point de Coupe *(pointcut)*

Groupement et nommage d'un ensemble de *points de jonction*.

Exemple :

```
pointcut moves():
   call(void Line.setP1(Point)) ||
   call(void Line.setP2(Point));
```

## 2.3 *Advice*

*Advice* ou greffon : nom donné au code à exécuter en association (avant, après, autour) avec un point de coupe.

Un advice fait référence à un point de coupe et spécifie le `quand` (avant ou après) et le `quoi` faire.

un *advice* dans l'exemple précédent :

```
after(): moves() { flag = true; } }
```

## 2.4 Tissage

Nom donné au processus de compilation dans lequel les codes des différents *advices* sont mélés au code standard, selon les indications données par les *points de coupe* et les *advices*, pour produire un exécutable final.

# 3 Aspects avancés

Le langage AspectJ pousse le concept d'aspect vers ses limites en permettant d'associer des aspects à tous les éléments logiciels et à tous les points de l'exécution d'un programme.

## 3.1 Une seconde application exemple

```
class Person{
    String name;
    Person(String n){name = n;}
    public String toString() {return (name);}}
```

```
class Place{
    String name;
    List<Person> l = new ArrayList<Person>();
    Place(String n){name = n;}
    public void add(Person p) {l.add(p);}
    public String toString() {
        String s = name + ": ";
        for (Person p: l){
            s += p.toString();
            s += ", ";}
        return(s);}}
```

```
public class Helloer {
```

```
    public void helloTo(Person p) {
        System.out.println("Hello to: " + p);}

    public void helloTo(Place p) {
        System.out.println("Hello to " + p);}
```

```
    public static void main(String[] args) {
        Helloer h = new Helloer();
        Person pierre = new Person("pierre");
        Person paul = new Person("paul");
        Person jean = new Person("jean");
        Place ecole = new Place("ecole");
        ecole.add(pierre);
        ecole.add(paul);
        Place theatre = new Place("theatre");
        theatre.add(paul);
        theatre.add(jean);

        h.helloTo(pierre);
        h.helloTo(theatre);} }
```

Execution standard :

```
Hello to: pierre
Hello to theatre: paul, jean,
```

## 3.2 “Advice” de type “before”

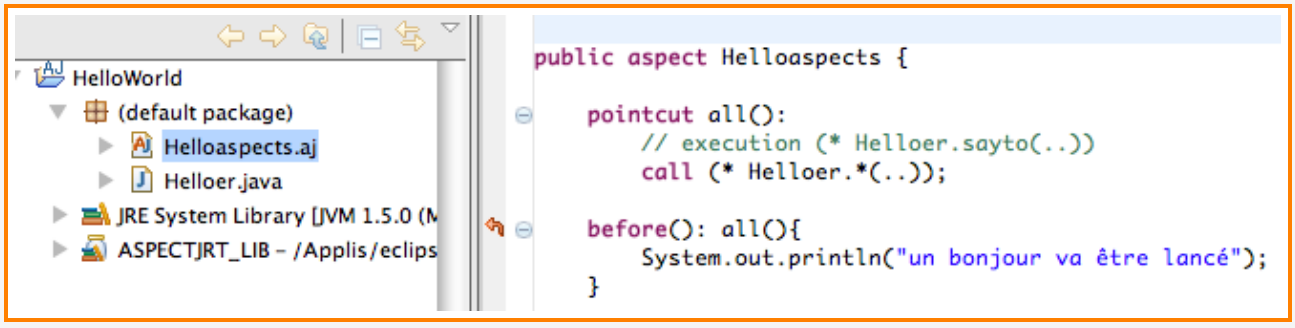


***Figure (1)*** – *advice “before”*

```
un bonjour va être lancé ...
Hello to: pierre
un bonjour va être lancé ...
Hello to theatre: paul, jean,
```

## 3.3 “Advice” de type “around”

```
public aspect Helloaspects {

    pointcut all():
        // execution (* Helloer.sayto(..))
        call (* Helloer.*(..));

    void around() : all() {
        System.out.println("---");
        proceed();
        System.out.println("---");
    }
```

***Figure (2)*** – *advice “around”*

Execution :

```
---
Hello to: pierre
---
---
Hello to theatre: paul, jean,
---
```

Variante Syntaxique

```
public aspect Helloaspects {

    void around(): call (* Helloer.*(..)) {
        System.out.println("---");
        proceed();
        System.out.println("---");
    }
```

## 3.4 “Advice” de type “around” et retour de valeur

```
    int around(): call (* Point.setX(int)) {
        System.out.println("interception d'un accès en lecture");
        int x = proceed();
        System.out.println("abcisse lue : " + x);
        return x;
    }
```

## 3.5 Point de coupe avec filtrage

```
public aspect Helloaspects {

    pointcut toPerson():
       call (* Helloer.*(Person));

    pointcut toPlace():
```

```
        call (* Helloer.*(Place));

    before(): toPerson(){
        System.out.println("Appel individuel");}

    before(): toPlace(){
        System.out.println("Appel aux personnes dans un lieu");}
```

Exécution :

```
Appel individuel
Hello to: pierre
Appel Aux personnes dans un lieu
Hello to theatre: paul, jean,
```

## 3.6 Points de coupe sur différents points de l'exécution

http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

— jonction avec un envoi de message (appel de méthode),

`call(void *.setX(int))`

— jonction avec l'exécution d'un corps de méthode, prends en compte sous-typage et redéfinitions : fonctionne correctement si une redéfinition de `helloTo` sur une sous-classe de `Helloer` est invoquée.

`execution (void Helloer.helloTo(..));`

`execution(void Point.setX(int))`

— jonction avec l'exécution d'un handler

`handler(ArrayOutOfBoundsException)`

— coupe quand l'objet courant durant l'exécution d'une méthode (i.e. this) est d'un certain type

`this(SomeType)`

— coupe quand le receveur d'un envoi de message est d'un certain type

`target(SomeType)`

— coupe quand le code en exécution est définie sur une classe C

`within(C)`

— jonction avec tout point de jonction se trouvant dans le flot de contrôle (la portée dynamique) de l'exécution d'une méthode m d'une classe C

`cflow(call(void C.m()))`

## 3.7 Point de coupe avec filtrage et paramètres

```
public aspect Helloaspects {

    pointcut toPerson(Helloer h, Person p):
        target(h) &&
        args(p) &&
        call (* Helloer.*(Person));

    pointcut toPlace():
        call (* Helloer.*(Place));

    before(Helloer h, Person p): toPerson(h, p){
```

```
        System.out.println("Appel individuel à " + h + " pour " + p);
    }
```

Exécution :

```
Appel individuel à Helloer@ec16a4 pour pierre
Hello to: pierre
Hello to theatre: paul, jean,
```

## 3.8 exemple de ”this”

```
public aspect Helloaspects {

    pointcut recev(Object o):
        args(o) &&
        this(Helloer);

    after(Object o): recev(o){
        System.out.println("an Helloer executing, context : " +
            thisJoinPoint + ", with : " + o);
    }
```

**thisJoinPoint** est une pseudo-variable contenant un objet représentant le point de jonction courant

Execution :

```
an Helloer executing, context : call(java.lang.StringBuilder(String)), with : Hello to:
an Helloer executing, context : call(StringBuilder java.lang.StringBuilder.append(Object)), with : pierre
Hello to: pierre
an Helloer executing, context : call(void java.io.PrintStream.println(String)), with : Hello to: pierre
an Helloer executing, context : execution(void Helloer.helloTo(Person)), with : pierre
an Helloer executing, context : call(java.lang.StringBuilder(String)), with : Hello to
an Helloer executing, context : call(StringBuilder java.lang.StringBuilder.append(Object)), with : theatre: paul,
    jean,
Hello to theatre: paul, jean,
an Helloer executing, context : call(void java.io.PrintStream.println(String)), with : Hello to theatre: paul, jean,
an Helloer executing, context : execution(void Helloer.helloTo(Place)), with : theatre: paul, jean,
```

## 3.9 jonction avec une instantiation

L’instantiation se détecte dans un point de jonction, par une exécution d’une méthode **new** (même si **new** n’est pas une méthode en Java).

```
public aspect testNew {

        pointcut newExec(String n):
            args(n) &&
            execution (Place.new(String));

        before(String n): newExec(n){
            System.out.println("execution d’un constructeur de Place: " + n);}
}
```

## 3.10 Différence entre target et this

Permet de cibler distinctement un envoi de message dont le receveur est d'un type donné (`target`) et l'exécution d'une méthode dont le receveur est d'un type donné (`this`).

```
public aspect DifferenceTargetThis {

    pointcut testTarget():
        target(Person);

    pointcut testThis():
        this(Person);

    before(): testTarget(){
        System.out.println("Envoi de message à une personne : " + thisJoinPoint);}

    before(): testThis(){
        System.out.println("Exécution méthode dont receveur est une personne : " + thisJoinPoint);}
}
```

## 3.11 Contrôler l'instantiation

Exemple : réaliser un aspect qui

```
import java.util.ArrayList;
import java.util.Collection;

public class ClassMemo {
    private String name;

    public ClassMemo(String n) { name = n; }

    public String toString() { return (name); }
}
```

```
public aspect MemoClasssAJ2 {

    private static ArrayList<Object> instances = new ArrayList<Object>();

    public static ArrayList<Object> getInstances() { return instances; }

    pointcut instantiation():
       execution (ClassMemo.new(..));

    after() : instantiation(){
      System.out.println("ajout d'instances");
      Object mo = thisJoinPoint.getThis();
      instances.add(mo); }
  }

public class MainMemo2 {
    public static void main(String[] args) {
        System.out.println("before : " + MemoAJ2.getInstances());
        new ClassMemo("memo1");
        new ClassMemo("memo2");
        System.out.println("after : " + MemoAJ2.getInstances()); }
}
```

***Listing (2)*** – *Un aspect pour réaliser une MémoClasse, verson 1. (*Courtesy of *Blazo Nastov)*

## 3.12 Test “Within”

Quand le code en exécution appartient à une classe donnée.

```
public aspect testWithin {

    pointcut testWithin():
        within(Person);

    after(): testWithin(){
        System.out.println("The executing method code belongs to class
        Person, context : " + thisJoinPoint);}
}
```

## 3.13 Test “cflow”

*Cflow* permet d’intervenir pour tout évènement survenant dans la portée dynamique d’un appel de méthode.

```
public aspect testCflow {

        pointcut stackHello():
        cflow(call(* Helloer.helloTo(Person))) && (!within(testCflow));

        before(): stackHello(){
            System.out.println(
                "The executing code is up the helloTo(Person) stack frame"
                + thisJoinPoint);}
}
```

## 3.14 Instantiation

Une instance d’un aspect est créée automatiquement par le compilateur.

Par défaut un aspect est un *Singleton*.

> *an aspect has exactly one instance that cuts across the entire program*

Un programme peut obtenir une instance d’un aspect via la méthode statique `aspectOf(..)`.

## 3.15 Héritage

Un aspect peut étendre une classe (la réciproque est fausse) et implanter une interface.

Un aspect peut étendre un aspect abstrait (mais pas un aspect concret), les points de coupes sont alors hérités.

## 3.16 Point de jonction pour “adviser” une méthode statique

`call` n’est pas adapté car l’appel d’une méthode statique n’est pas un envoi de message.

```
execution (public static void X.main(..))
```

# 4 Exemple de méta-programmation avec AspecJ

Réalisation de l’équivalent d’une méta-classe MemoClasse, classe des classes qui mémorisent la liste de leurs instances.

```
public aspect MemoAJ3 {

    private static ArrayList<Object> instances =
      new ArrayList<Object>();

    public static ArrayList<Object> getInstances(){
      return instances;}

    pointcut instantiation():
       //les classes A et B deviennent des MémoClasses
       execution (A.new(..)) || execution (B.new(..)) ;

     after() : instantiation(){
       Object mo = thisJoinPoint.getThis();
       instances.add(mo); }
}
```

# 5 Une mise en oeuvre dans un framework pour le web : Spring-AOP

https://docs.spring.io/spring-framework/docs/4.3.15.RELEASE/spring-framework-reference/html/aop.html

Voir https://docs.spring.io/spring-framework/docs/2.5.x/reference/aop.html.

Je suis intéressé par un exemple de code qui fonctionne.

# 6 Une mise en oeuvre (réduite) de l'idée d'aspect : les annotations Java

Une annotation permet d'associer des méta-données à un élément de programme Java, susceptibles de modifier sa sémantique ou de générer des comportement additionnels lors de sa définition ou de son exécution.

## 6.1 la cible (*target*) - lien avec point de jonction

*ElementType* :

```
    * TYPE — Applied only to Type. A Type can be a Java class or interface or an Enum or even an Annotation.
    * FIELD — Applied only to Java Fields (Objects, Instance or Static, declared at class level).
    * METHOD — Applied only to methods.
    * PARAMETER — Applied only to method parameters in a method definition.
    * CONSTRUCTOR — Can be applicable only to a constructor of a class.
    * LOCAL_VARIABLE — Can be applicable only to Local variables. (Variables that are declared within a method
        or a block of code).
    * ANNOTATION_TYPE — Applied only to Annotation Types.
    * PACKAGE — Applicable only to a Package.
```

## 6.2 la portée ("retention")

```
   * {\tt Source—file} : lue par le compilateur, non conservée dans le ``.class''
   * {\tt Classfile}, : comme {\tt source—file}, puis conservée dans le
   ``.class'', non accessible à l'exécution
   * {\tt Runtime} : comme {\tt class-file}, puis accessible à l'exécution.
```

## 6.3 Exemple : une annotation pour associer un try-catch au corps d'une méthode sans modifier son code

### 6.3.1 Utiliser l'annotation

Le code ci-dessus utilise une nouvelle annotation nommée `methodHandler`, pour dire : associer un handler (try-catch) à la méthode *BookFlight* de la classe *Flight*.

Le code du handler (i.e. le corps de l'équivalent clause `catch`) est la méthode `handle` à laquelle est associée l'annotation.

```
1  import methodHandler;
2
3  class Flight{
4
5    public void bookFlight( ... ) { ... }
6
7    @methodHandler(methodName="BookFlight")
8    public void handle(noFlightAvailableException e) {
9        System.out.println(" BROKER : getMethod method handler exception ");
10     }
11   }
```

### 6.3.2 Implantation de l'annotation - 1 @interface

```
1  import java.lang.annotation.ElementType;
2  import java.lang.annotation.Retention;
3  import java.lang.annotation.RetentionPolicy;
4  import java.lang.annotation.Target;
5
6  /**
7  * methodHandler
8  * Defines a "methodHandler" annotation used to associate a handler to a method.
9  * @author Sylvain Vauttier
10 */
11
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.METHOD)
14 public @interface methodHandler {
15     String methodName();
16 }
```

### 6.3.3 Implantation de l'annotation - 2 Prise en compte de l'annotation à l'exécution

```
1    import java.lang.reflect.AnnotatedElement
2
3    Method[] meths = this.getClass().getMethods();
4    for (Method m : meths)
5    { if (m.getAnnotation(methodHandler.class) != null)
6            if (m.getAnnotation(methodHandler.class).methodName().toString().equals(methodName))
7                if (m.getParameterTypes()[0].equals(e.getClass())) {
8                    print("Finding and Running Method handler");
9                    runHandler(this, m, e);
10                   return;
11               }
12   }
```