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

Programmation par Aspects (AOP), ou la Réutilisation Transversale (*"Cross-cutting Reuse"*)

Notes de cours
Christophe Dony

1 Idée

1.1 Base

Séparation du code métier et du code réalisant des fonctionnalités annexes, par exemple techniques.

Un exemple en Java standard, *synchronized* isole le code métier du code, placé ailleurs, réalisant une mise en section critique ; c'est un exemple d'"aspect".

```
1  class CompteBanque
2
3    float solde = 0;
4
5    public void synchronized retrait(float montant)
6          throws CreditInsuffisant {
7       if (solde >= montant)
8          solde = solde - montant;
9          else throw new CreditInsuffisant();}
10
11   ...
```

1.2 Généralisation

Permettre l'ajoût de fonctionnalités "orthogonales" à un code métier sans le modifier.

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**, **Advice**, **Point de coupe (pointcut)**, **Point de jonction (jointpoint)**

1.3 Aspects et méta-programmation

La programmation par aspects est aussi une forme de méta-programmation. Elle donne accès à certaines entités du méta-niveau de représentation ou d'exécution.

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*.

La programmation par aspects fourni un moyen conceptuellement plus simple d'étendre une construction standard d'un langage.

2 Problème : séparation des préoccupations (*separation of concerns*)

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

```
class Line extends Figure { // version 1
  private Point p1, p2;
  Point getP1() { return p1; }
  Point getP2() { return p2; }
  void setP1(Point p1) { p1 = p1; }
  void setP2(Point p2) { p2 = p2; } }

class Point extends Figure { // version 1
  private int x = 0, y = 0;
  int getX() { return x; }
  int getY() { return y; }
  void setX(int x) { this.x = x; }
  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.

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

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); } }
```

3 Programmation par Aspect

3.1 Aspect

Construction permettant de représenter un point de vue, ou une fonctionnalité dite orthogonale, sur une stucture (par exemple, le point de vue de détection d'un déplacement ou de mémorisation de l'historique des déplacements).

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 [2]

```
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)) ||
```

ça nous permet d'ajouter du code dans la classe Line sans toucher du code de la classe Line

2. Discussion : qui invoque `testAndClear()` ?

```
8       call(void Line.setP2(Point));

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

3.2 Point de Jonction

Points dans l'exécution d'un programme où l'on souhaite faire quelque chose.

un *point de jonction* dans l'exemple précédent :

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

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

Lle point de jonction suivant désigne par exemple tout appel à une méthode publique de la classe `Figure`.

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

3.3 Point de Coupe *(pointcut)*

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

Exemple :

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

3.4 *Advice*

Nom donné au code à exécuter en association 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 :

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

on met cette ligne partout où il y a une appel "setP1"

3.5 Tissage

Nom donné au processus de compilation dans lequel les codes des différents advice sont intégrés au code standard selon les indications données par les points de coupe.

4 Aller plus loin

4.1 Une seconde application

```
1  class Person{
2      String name;
3      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,
```

4.2 “Advice” de type “before”

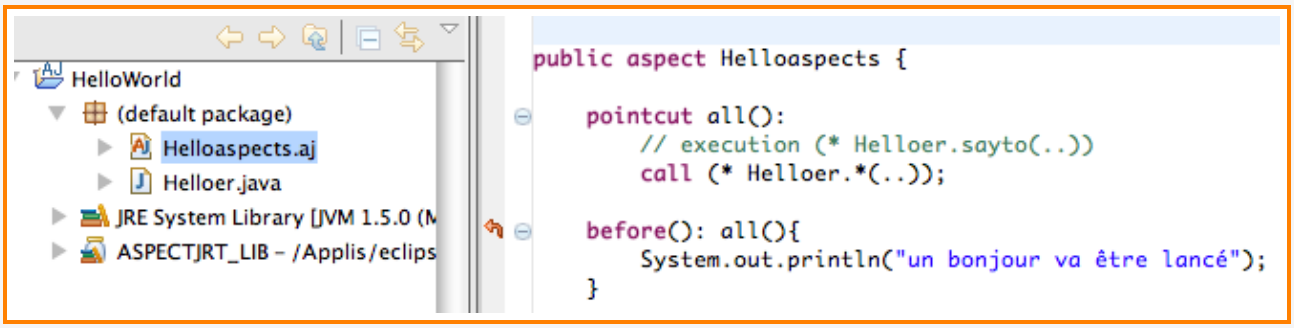

Figure *(1)* – *advice “before”*

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

4.3 “Advice” de type “around”

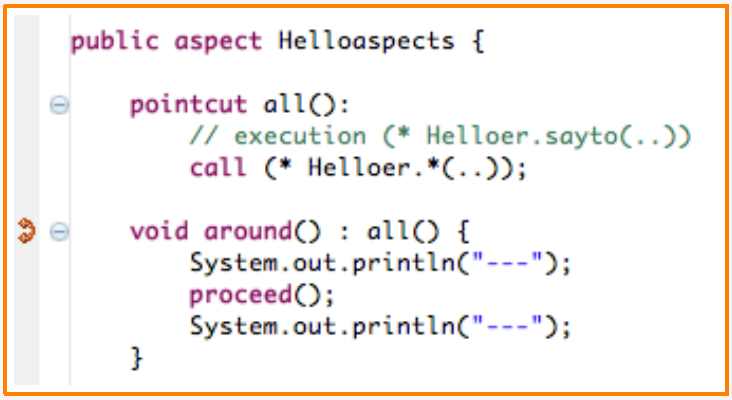

Figure *(2)* – *advice “around”*

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("---");
```

```
    }
```

4.4 “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;
    }
```

a[3]

4.5 Point de coupe avec filtrage

```
public aspect Helloaspects {

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

    pointcut toPlace():
        call (* Helloer.*(Place)); tous les appels qui intercepte la classe Helloer qui appelle la méthode avec  *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,
```

4.6 Spécification détaillée des points de coupe

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

- exécution de méthode
  execution(void Point.setX(int))
- envoi de message (appel de méthode), prends en compte le sous-typage
  call(void *.setX(int))
- exécution d'un handler
  handler(ArrayOutOfBoundsException)
- when the object currently executing (i.e. this) is of type SomeType
  this(SomeType)

3. merci à Sullivan

— when the target object is of type SomeType
target(SomeType)

— when the executing code belongs to class MyClass
within(MyClass)

— when the join point is in the control flow of a call to a Test's no-argument main method
cflow(call(void Test.main()))

4.7 Point de coupe avec filtrage et paramètres

```
1  public aspect Helloaspects {
2
3      pointcut toPerson(Helloer h, Person p):
4          target(h) &&
5          args(p) &&
6          call (* Helloer.*(Person));
7
8      pointcut toPlace():
9          call (* Helloer.*(Place));
10
11     before(Helloer h, Person p): toPerson(h, p){
12     System.out.println("Appel individuel à " + h + " pour " + p);
13     }
```

si la méthode appelé est de type object target(h) et argument est de type p
on envoie une message à cette méthode

on print d'abord cette ligne quand la methode toPerson() est appelé

Exécution :

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

4.8 Exemple de point de coupe avec paramètres pour l'application "Figures"

```
1  aspect MoveTracking {
2
3    private static Set movees = new HashSet();
4
5    public static Set getmovees() {
6      Set result = movees;
7      movees = new HashSet();
8      return result; }
9
10   pointcut moves(FigureElement f, Point p):
11     target(f) && args(p) &&
12     (call(void Line.setP1(Point)) ||
13      call(void Line.setP2(Point))) ;
14
15   after(Figure f, Point p): moves(f,p) {
16     movees.add(f);}
```

4.9 "this" : le receveur courant là où l'on coupe

```
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);
    }
```

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,
```

4.10 Point de coupe et constructeur

```
public aspect testConstructeur {

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

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

4.11 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 d onné (**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);}
```

```
}
```

4.12 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 code belongs to class
        Person, context : " + thisJoinPoint);}
}
```

4.13 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);}
}
```

4.14 Contrôler l’instantiation

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

public class MemoObject {
        private static ArrayList<MemoObject> instances = new ArrayList<MemoObject>();

        public MemoObject(){
        }

        public ArrayList<MemoObject> getInstances(){
                return instances;
        }
}
```

```
public aspect MemoAJ {
        pointcut addInstance()
                : execution(* MemoObject.MemoObject());

        after() returning() : addInstance() {
                System.out.println("ajoutd'instances");
                MemoObject mo = (MemoObject)thisJoinPoint.getThis();
                mo.getInstances().add(mo); }
```

```
}
```

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

4.15 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(..)`.

4.16 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.

4.17 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(..))
```

5 Une mise en oeuvre 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.

5.1 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.
```

5.2 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.
```

5.3 Exemple de Définition d'une annotation

Soit à définir une annotation nommée "methodHandler" permettant d'ajouter un handler d'exception au niveau d'une méthode d'une classe, sans modifier le code de la méthode.

```java
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
* methodHandler
* Defines a "methodHandler" annotation used to associate a handler to a method.
* @author Sylvain Vauttier
*/

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface methodHandler {
    String methodName();
}
```

5.4 Utiliser l'annotation dans le code utilisateur

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

```java
import methodHandler;

class Flight{

  ...

  @methodHandler(methodName="BookFlight")
  public void handle(noFlightAvailableException e) {
      System.out.println(" BROKER : getMethod method handler exception ");
    }
  }
```

cela evite de mettre des try-catch partout

5.5 Prise en compte de l'annotation à l'exécution

```java
  import java.lang.reflect.AnnotatedElement

  Method[] meths = this.getClass().getMethods();
  for (Method m : meths)
  { if (m.getAnnotation(methodHandler.class) != null)
          if (m.getAnnotation(methodHandler.class).methodName().toString().equals(methodName))
              if (m.getParameterTypes()[0].equals(e.getClass())) {
                  print("Finding and Running Method handler");
                  runHandler(this, m, e);
                  return;
              }
  }
```