

Programmation par Aspects - Cross-cutting
Reuse

Université Montpellier-II

Master Informatique IFPRU - Parcours GL

Notes de cours
Christophe Dony

1 Idée

1.1 Idée de Base

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

Un exemple en Java standard, *synchronized* peut être vu comme une sorte d'aspect.

```
class CompteBanque

    float solde = 0;

    public void synchronized retrait(float montant)
        throws CreditInsuffisant {
        if (solde >= montant)
            solde = solde - montant;
        else throw new CreditInsuffisant();}

    ...
```

1.2 Aspects et méta-programmation

La programmation par aspects peut être vue comme une forme de méta-programmation.

Définir un aspect “synchronization”, comme celui utilisé dans l’exemple précédent, peut être comparé à la réalisation d’une nouvelle méta-classe *synchronized-method*.

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

1.3 Généralisation de l'idée

Permettre l'ajout de fonctionnalités orthogonales à tout code métier, sans modifier ce dernier.

Ouvrage : Aspect-Oriented Programming : Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin, ECOOP 1997.

Concepts : **Aspect**, **Advice**, **Point de coupe (pointcut)**, **Point de jonction (jointpoint)**

2 Problème : séparation des préoccupations (separation of concerns)

Considérons comme exemple^a celui de la programmation d'objets d'une application graphique, instances des classes **Line** et **Point**.

^atiré de “<http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html>”

```
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 un objet a été modifié par l'utilisateur.

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

D'une part une classe annexe 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** doivent être 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.

```
class MoveTracking{                                     //Version 3

    private static Set displacements new HashSet();

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

    public static Set getDisplacements() {
        Set result = displacements;
        displacements= newHashSet();
        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); } }
```

Les aspects offrent une solution originale et alternative à cette solution.

3 Aspect

Construction permettant de représenter un point de vue, ou une fonctionnalité dite orthogonale, sur une structure (par exemple, le point de vue de la mémorisation des modification des instances des classes précédentes).

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

```
aspect MoveTracking { // Aspect version 1
    private boolean flag = false;
    public 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; } }
```

3.1 Advice

Nom donné au code à exécuter à différents points 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; } }
```

3.2 Point de Jonction

Points dans l'exécution d'un programme où l'on souhaite placer un advice.

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 à une méthode publique de la classe **Figure**.

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

3.3 Point de Coupe (pointcut)

Ensemble de *points de jonction* permettant de factoriser l'association à un *advice*.

Exemple :

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

3.4 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

```
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,
```

4.2 Advice de type “before”

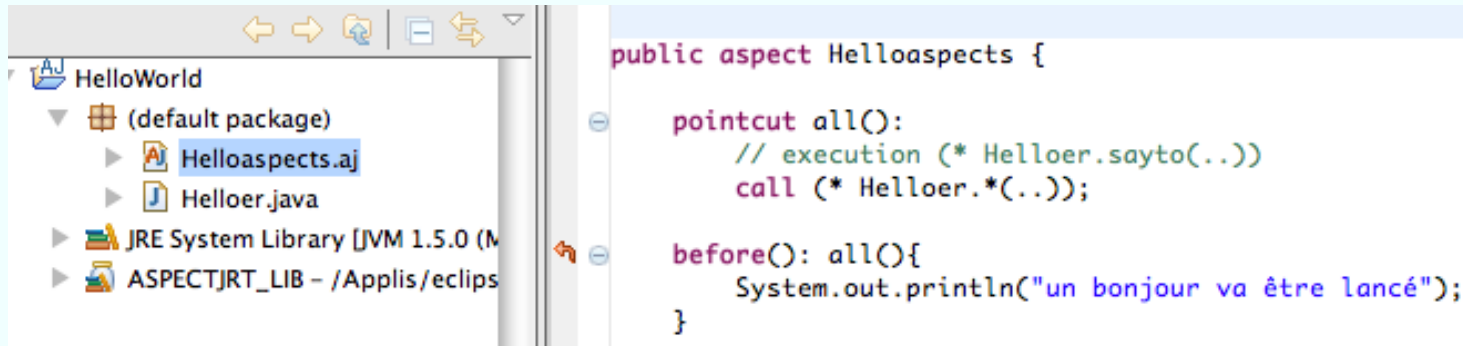


FIG. 1 – advice “before”

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

4.3 Advice de type “around”

```
public aspect Helloaspects {  
    pointcut all():  
        // execution (* Helloer.sayto(..))  
        call (* Helloer.*(..));  
    void around() : all() {  
        System.out.println("---");  
        proceed();  
        System.out.println("---");  
    }  
}
```

FIG. 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 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");}
```

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

4.5 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(ArrayOutOfBoundsExpection)`

– when the object currently executing (i.e. this) is of type SomeType

`this(SomeType)`

- 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()))

4.6 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,
```

4.7 Un point de coupe avec paramètres pour l'application "Figures"

```
aspect MoveTracking {  
  
    private static Set movees = new HashSet();  
  
    public static Set getmovees() {  
        Set result = movees;  
        movees = new HashSet();  
        return result;  
    }  
  
    pointcut moves(FigureElement f, Point p):  
        target(f) && args(p) &&  
        (call(void Line.setP1(Point)) ||  
         call(void Line.setP2(Point))) ;  
}
```

4.8 Point de coupe avec utilisation 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);  
    }  
}
```

Execution :

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

4.9 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(..))
```

4.10 Déclarations inter types

...

5 Une version simplifiée des aspects : les annotations Java

5.1 Définition d'une annotation

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

/**
 * serviceHandler
 * Defines a "serviceHandler" annotation used to associate a handler to a s
 * @author      Sylvain Vauttier
 */
```

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

5.2 Annoter l'annotation : la cible (*target*)

L'énumération *ElementType* contient les éléments suivants :

- * TYPE { Applied only to Type. A Type can be a Java class or interface.
- * FIELD { Applied only to Java Fields (Objects, Instance or Static, de
- * 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. (Variabl
- * ANNOTATION_TYPE { Applied only to Annotation Types.
- * PACKAGE { Applicable only to a Package.

5.3 Annoter l'annotation : la portée (*retention*)

- * Code Source
- * Fichier .class
- * Runtime : pour utilisation par la VM

5.4 Utiliser l'annotation dans le code utilisateur

```
import serviceHandler;

@ServiceHandler(serviceName="getService")
public void handle(noFlightAvailableException e)
{
    System.out.println(" BROKER : getService service handler exception ");
}
```

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

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