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

Programmation par Aspects (AOP), ou Définition Modulaire d'Entités Transverses (*"Cross-Cutting Reuse"*)

Notes de cours

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` permet de traiter 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 bougé 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 extends Figure { // 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 extends Figure{ // 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 extends Figure { //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 extends Figure { //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.

Ci-dessous un aspect (version 1) écrit en *AspectJ* qui permet de réaliser la fonctionnalité orthogonale précédente (détection d'un déplacement) sans toucher au 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)) ||
    call(void Line.setP2(Point));

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

2.2.2 Point de Jonction

Points dans l'exécution d'un programme où l'on souhaite exécuter une fonctionnalité orthogonale.

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

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

Le langage de définition des points de jonction accepte des caractères de filtrage, tels "*" (quel que soit) ou ".." (quel que soit le cardinal).

Le point de jonction suivant désigne tout appel d'une méthode publique de la classe `Figure`, quel que soit son type de retour, quel que soit son nom et quels que soient les nombres et types de ses paramètres.

2. note : Un programme client peut invoquer la méthode statique `testAndClear()` (pour savoir si des figures ont bougé et remettre à faux le booléen `flag`) de l'aspect `MoveTracking`, comme il pouvait invoquer la méthode statique `testAndClear` de la classe `MoveTracking` dans la version 2 (ou 3) du programme.

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

2.2.3 Point de Coupe *(pointcut)*

Groupement logique 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, litt. conseil, règle à suivre, règle à appliquer ou greffon : nom donné au code à exécuter en association à un point de coupe quand il est activé.

Un advice fait référence à un point de coupe et spécifie le `quand` (avant, après, ...) et le `quoi` faire quand l'exécution du programme atteint une occurrence de ce point.

Un point de coupe peut être activé (vérifié) un nombre non limité de fois pour chaque exécution.

un *advice* dans l'exemple précédent :

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

2.4 Tissage

Nom donné au processus de compilation d'un programme avec aspects, dans lequel les codes des différents *advices* sont mélés au code standard, aux emplacements donnés par les *points de coupe*, pour produire un exécutable final.

3 Aller plus loin

Le langage AspectJ pousse le concept d'aspect vers ses limites en permettant d'associer des aspects à (presque) tous les éléments logiciels constituant un programme (constructeur, méthode, handler, ...) et à tous les points de l'exécution d'un programme.

3.1 Une seconde application exemple

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

public class Child extends Person {
    Child(String n) {super(n);}
    public String toString() {
        return ("enfant " + super.toString()); }
}
```

Listing (2) *– des personnes*

```
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);}}
```

Listing (3) – *des lieux*

```
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);}
```

Listing (4) – *des diseurs de bonjours*

```
   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);} }
```

Listing (5) – *un main*

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

Listing (6) – *Exécution standard*

3.2 Contrôler l'exécution des advices

“Advice” de type “before” (resp. “after”)

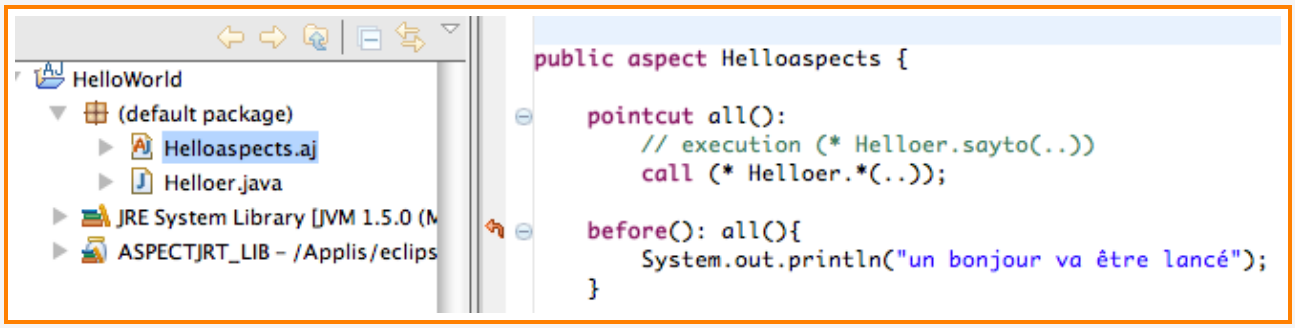

Figure (1) – *advice “before”*

```
1 un bonjour va être lancé ...
2 Hello to: pierre
3 un bonjour va être lancé ...
4 Hello to theatre: paul, jean,
```

Listing (7) – *Nouvelle execution du main du listing 5*

“Advice” de type “around”

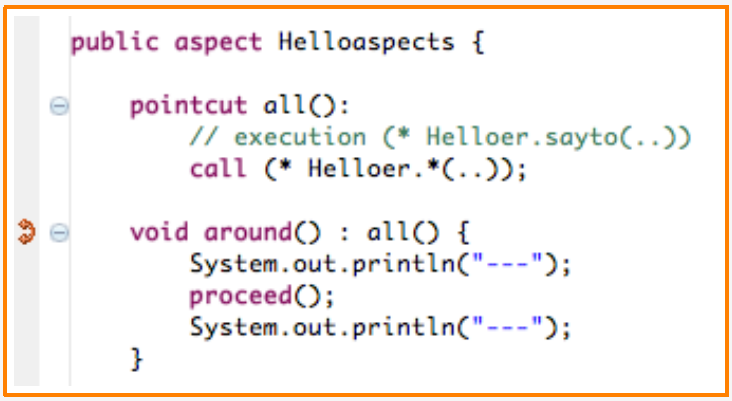

Figure (2) – *advice “around”*

Execution :

```
1 ---
2 Hello to: pierre
3 ---
4 ---
5 Hello to theatre: paul, jean,
6 ---
```

Variante syntaxique équivalente :

```
1 public aspect Helloaspects {
2     void around(): call (* Helloer.*(..)) {
3         System.out.println("---");
4         proceed();
```

```
        System.out.println("---");
    }
```

"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 et rendue : " + x);
        return x;
    }
```

3.3 Points de jonction avec filtrage (pattern matching)

```
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,
```

3.4 Point de jonction et de coupe avec filtrage et paramètres

Les advices peuvent avoir des paramètres si les points de coupe les déclarent.

```
public aspect Helloaspects {

    pointcut toPerson(Helloer h, Person p):
        target(h) && //h sera liée au receveur
        args(p) && //p sera liée à la ''Person'' argument
        call (* Helloer.*(Person)); //des appels aux méthodes compatibles

    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 Précisions sur le langage d'expression des points de jonctions ...

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

4.1 call (* T.m(..))

j jonction avec un envoi du message m(..) via une variable de type statique T et de type dynamique ST <= T.

Exemple : `call(void *.setX(int))` jointe avec l'envoi de tout message setX(int) quel que soit le receveur s'il comprend ce message.

4.2 execution(m)

jonction avec l'exécution d'un corps de méthode de signature `m`, avec prise en compte des redéfinitions.

Exemple : `execution (void Helloer.helloTo(..))` jointe avec l'exécution de toute méthode `helloTo` de la classe `Helloer` ou l'une de ses redéfinitions.

4.3 this(T)

jonction quand l'objet courant durant l'exécution d'une méthode (i.e. this) est de type T

Exemple. **thisJoinPoint** est une pseudo-variable contenant un objet représentant le point de jonction courant, celui pour lequel l'*advice* est exécuté.

```
public aspect testThis {
    pointcut receveur(Object o): args(o) && this(Helloer);
    after(Object o): receveur(o){
       System.out.println(
       "an Helloer is doing : " + thisJoinPoint + ", with : " + o); }
```

Listing (8) – *Point de jonction avec "this"*

```
...
an Helloer is doing : execution(void HelloerTest.Helloer.helloTo(Person)), with : pierre
...
an Helloer is doing : execution(void HelloerTest.Helloer.helloTo(Place)), with : theatre: paul, jean,
```

Listing (9) – *résultat*

4.4 target(T)

jonction quand l'objet concerné par l'opération en cours est de type T,

similaire à "this" mais plus général car l'opération en cours peut être autre chose que l'exécution d'une méthode, l'exécution d'un constructeur par exemple.

4.5 `within(C)`

jonction quand le code en exécution appartient à la classe C.

4.6 Différence et ordre entre `within call target this` et `execution`

4.6.1 Sans héritage

Soit la classe `Person` de l'application `Helloer`, considérons l'aspect du listing 10, il définit un point de coupe pour chacun des 5 mots-clés, et un *advice* pour chaque point de coupe. chaqie *advice affiche* un message et le point de jonction courant (`thisJoinPoint`).

```
1  public aspect testTargetCallExecuteThisWithin2 {
2      pointcut testTarget1(): target (Person);
3      pointcut testThis1(): this (Person);
4      pointcut testCall1(): call (String Person.toString());
5      pointcut testExecute1(): execution (String Person.toString());
6      pointcut testWithin1(): within (Person);
7      ...
8      // les advices, non listés
```

Listing (10) –

Comment s'exécute le main suivant ?

```
1  public static void main(String[] args) {
2      Person andréa = new Person("andréa");
3      andréa.toString();  }
```

Exécution (partie 1) :

```
// static init constructeur
within Person : staticinitialization(HelloerTest.Person.<clinit>)
// pre init constructeur
within Person : preinitialization(HelloerTest.Person(String))
// init constructeur
this Person : initialization(HelloerTest.Person(String))
within Person : initialization(HelloerTest.Person(String))
// execution constructeur
target Person : execution(HelloerTest.Person(String))
this Person : execution(HelloerTest.Person(String))
within Person : execution(HelloerTest.Person(String))
// affectation de l'attribut name dans le constructeur
target Person : set(String HelloerTest.Person.name)
this Person : set(String HelloerTest.Person.name)
within Person : set(String HelloerTest.Person.name)
```

Exécution (partie 2) :

```
//appel de la méthode toString()
call toString de Person : call(String HelloerTest.Person.toString())
target Person : call(String HelloerTest.Person.toString())
//execution de la méthode toString()
exec toString de Person : execution(String HelloerTest.Person.toString())
target Person : execution(String HelloerTest.Person.toString())
this Person : execution(String HelloerTest.Person.toString())
within Person : execution(String HelloerTest.Person.toString())
//accès à l'attribut name dans toString()
target Person : get(String HelloerTest.Person.name)
this Person : get(String HelloerTest.Person.name)
within Person : get(String HelloerTest.Person.name)
```

***Listing (11)** –*

Considérer la séquence exec, target, this, within relative l'exécution de la méthode.

4.6.2 Avec héritage

Avec le même aspect (listing 10), on exécute :

```
1 public static void main(String[] args) {
2         Child andréa = new Child("andréa");
3         andréa.toString();
```

`Child` (listing 2) est sous-classe de `Person` et redéfinit `toString()`.

Exécution (vue partielle) :

```
// ...
target Person : initialization(HelloerTest.Person(String)) //constructeur Person
//...
target Person : initialization(HelloerTest.Child(String)) // constructeur Child
// ...
//Le point de jonction sur Person.toString() jointe sur l'envoi de message toString() à un Child.
call toString de Person : call(String HelloerTest.Child.toString())
target Person : call(String HelloerTest.Child.toString())
//exécution de la méthode toString() de Child
exec toString de Person : execution(String HelloerTest.Child.toString())
target Person : execution(String HelloerTest.Child.toString())
this Person : execution(String HelloerTest.Child.toString())
this Person : call(java.lang.StringBuilder(String))
// exec toString() de Person, envoi de message à super
exec toString de Person : execution(String HelloerTest.Person.toString())
target Person : execution(String HelloerTest.Person.toString())
this Person : execution(String HelloerTest.Person.toString())
within Person : execution(String HelloerTest.Person.toString())
target Person : get(String HelloerTest.Person.name)
 //...
```

4.6.3 Avec affectation polymorphique

Avec le même aspect (listing 10), on exécute ce nouveau programme :

```
public static void main(String[] args) {
        Person andréa = new Child("andréa");
        andréa.toString();
```

Execution :

```
// ...
//appel de méthode Person.toString(), la liaison dynamique est à venir
//thisJoinPoint correspond au type statique du receveur
call toString de Person : call(String HelloerTest.Person.toString())
target Person : call(String HelloerTest.Person.toString())
//exécution de la méthode, liaison dynamique faite
exec toString de Person : execution(String HelloerTest.Child.toString())
target Person : execution(String HelloerTest.Child.toString())
this Person : execution(String HelloerTest.Child.toString())
this Person : call(java.lang.StringBuilder(String))
// ..
```

4.7 Jonction avec une instantiation

Une instantiation se détecte en faisant référence à une exécution de la méthode `new` (même si `new` n'est pas une méthode en Java).

```
public aspect testNew {

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

    before(String n): instantiation(n){
        System.out.println("Instantiation : " + thisJoinPoint + n); }}
```

Execution du main initial (listing 5).

```
Instantiation : execution(HelloerTest.Place(String))ecole
Instantiation : execution(HelloerTest.Place(String))theatre
```

4.8 Application : instrumenter l'instantiation

Exercice : réaliser un aspect qui mémorise [3] la liste des instances d'une classe.

```
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); } }
```

Listing (12) – *Un aspect pour réaliser une MémoClasse, v2. (*Courtesy of *Blazo Nastov)*

3. Il s'agit d'une transposition d'un exercice classique de méta-programmation, la méta-classe MémoClass, dont les instances mémorisent la liste de leurs instances.

Utilisation :

```
public class MainMemo2 {
    public static void main(String[] args) {
        System.out.println("debut : " + MemoAJ2.getInstances());
        new Person("andréa");
        new Person("noah");
        System.out.println("fin : " + MemoAJ2.getInstances()); } }
```

Exécution :

```
debut : []
ajout d'instances : a Person named : andréa
ajout d'instances : a Person named : noah
fin : [a Person named : andréa, a Person named : noah]
```

Amélioration : Pourrait-on faire un aspect qui mémorise la liste des instances de toutes les classes ? En une seule collection ? Avec une collection pour chaque classe ?

4.9 cflow(C.m)

Jonction quand le point d'exécution se trouve dans la portée dynamique [4] de l'exécution d'une méthode m d'une classe C, donc pour tout ce qui "au dessus" du bloc de pile créé par l'activation de la méthode m, tant que la méthode m est en exécution.

```
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);}
}
```

A essayer en TP. Sauvegarder avant.

4.10 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 Aspects et déclarations inter types

Un aspect peut

- ajouter des attributs et des méthodes à une classe existante.

- déclarer qu'une classe existante implante une nouvelle interface

- ... https://eclipse.dev/aspectj/doc/released/progguide/language-interType.html

4. Au sens de la portée dynamique des identificateurs, voir e.g. le declare special de Common-Lisp.

6 Fonctionnement de AspectJ

6.1 Instantiation des aspects

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(..)`.

6.2 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. (Et les advices ?).

7 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.

8 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.

8.1 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.
```

8.2 la portée ("retention")

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

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

8.3.1 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 de la 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        //... ce que l'on écrit usuellement dans un bloc catch
10     }
11   }
```

8.3.2 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 }
```

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

Le système de gestion des exceptions du système implanté recherche s'il existe des annotations *methodHandler* associées aux méthodes de la classe du receveur.

```
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)
7                        .methodName().toString()
```

```
                        .equals(methodName))
                if (m.getParameterTypes()[0].equals(e.getClass())) {
                    print("Finding and Running Method handler");
                    runHandler(this, m, e);
                    return;
                }
    }
```