

Aspect Oriented Programming



Mohamed Youssfi

Laboratoire Signaux Systèmes Distribués et Intelligence Artificielle (SSDIA)

ENSET, Université Hassan II Casablanca, Maroc

Email : med@youssfi.net

Supports de cours : <http://fr.slideshare.net/mohamedyoussfi9>

Chaîne vidéo : <http://youtube.com/mohamedYoussfi>

Recherche : http://www.researchgate.net/profile/Youssfi_Mohamed/publications c

AOP?

- La **programmation orientée aspect** (*Aspect Oriented Programming* ou AOP) est un paradigme de programmation qui permet de traiter séparément
 - **Les préoccupations transversales** (*Cross Cutting Concerns*), qui relèvent souvent des aspects technique (Journalisation, Sécurité, Transaction, ...)
 - **des préoccupations métiers**, qui constituent le cœur d'une application¹.
- Permet de Séparer le code **métier** du code **technique**

Exemple :

- Dans les applications classiques, toutes les méthodes d'une application contiennent du code qui permet de logger des messages au début et à la fin de la méthode.
- Ce qui constitue une répétition du même code à tous les niveaux de l'application. Ce qui peut engendrer des problèmes au niveau de la maintenance.
- Grâce à la programmation orientée aspect, on peut développer notre application sans se préoccuper de la journalisation.
- Une classe séparée (Aspect) pourra être développée par la suite pour doter l'application de cet aspect de journalisation
- Il serait de même pour ajouter d'autres aspects techniques comme
 - La sécurité, Gestion de transaction, Gestion des exceptions, ...

AOP?

- La programmation orientée aspect est bien une technique transversale et n'est pas liée à un langage de programmation particulier
- peut être mise en œuvre aussi bien avec un langage orienté objet comme Java qu'avec un langage procédural comme le C,
- Le seul pré-requis étant l'existence d'un *tisseur d'aspect* pour le langage cible.

OOP / AOP

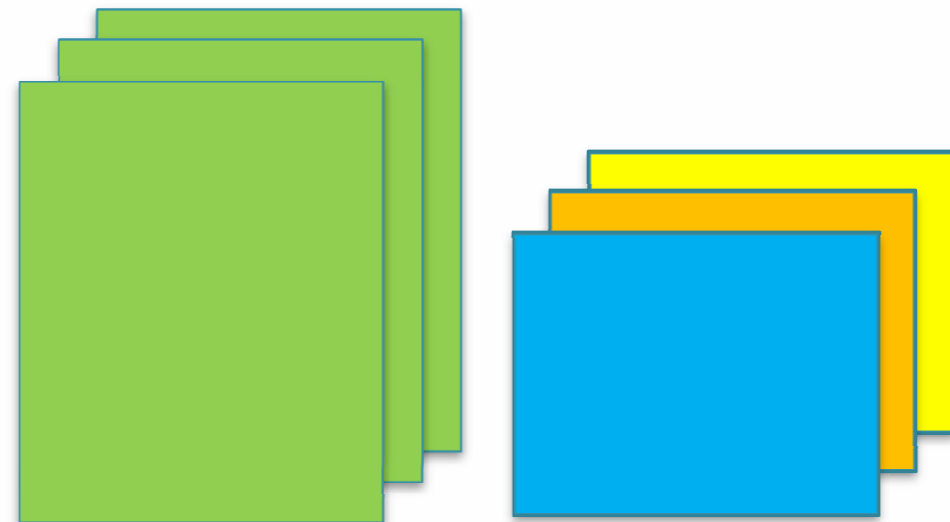
- Logique métier
- Journalisation
- Sécurité
- Persistance

POO



Code des méthodes

POO + AOP

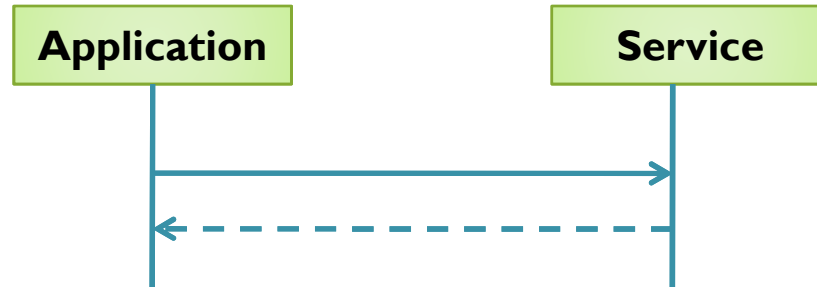


Code des méthodes

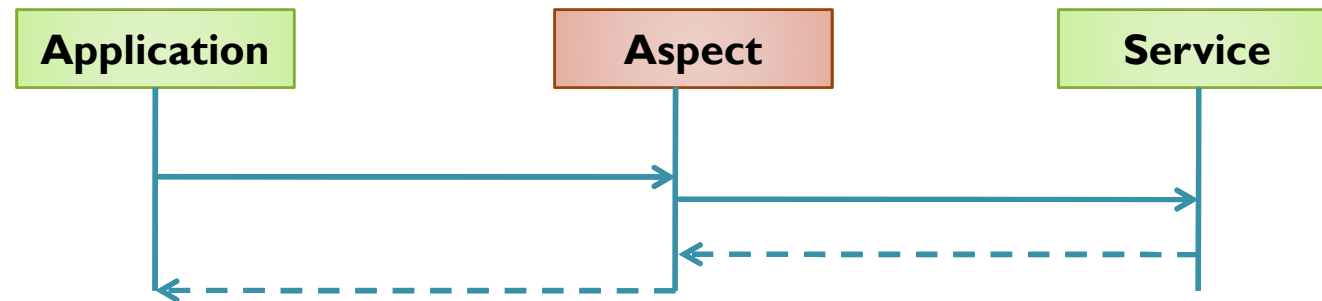
Code des Aspects

Architecture POO / AOP

POO



POO + AOP



Tisseurs d'aspects

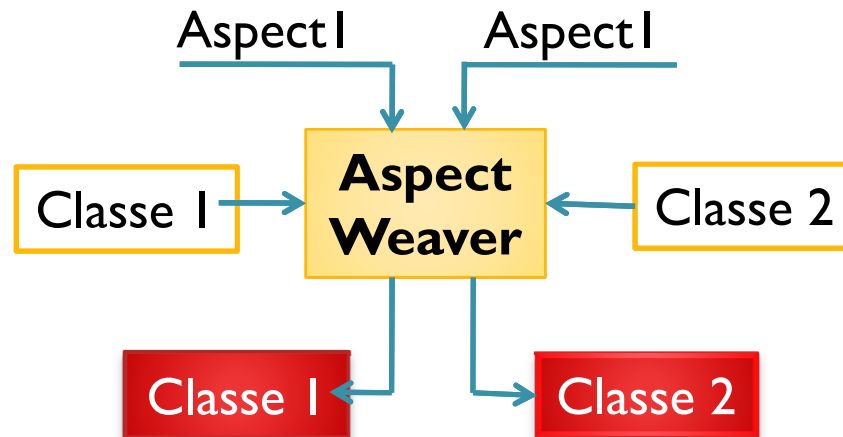
- Un programme codé en orienté aspect peut être découpé en deux parties disjointes :
 - Les classes pour la logique métier
 - Les aspects pour les aspects techniques
- Le programme ne doit pas avoir connaissance des aspects, ils ne sont jamais appelés dans les classes du programme.
- Nous pouvons donc nous demander comment ces aspects sont appliqués sur le programme.
- C'est le but des **tisseurs d'aspects** (ou **aspect weaver**),
- ils ont pour rôle de greffer l'ensemble des aspects sur l'ensemble des classes du programme.
- Les tisseurs d'aspect se différencient sur deux points :
 - le langage de programmation utilisé
 - le moment où le tissage est réalisé.

Tisseurs d'aspects

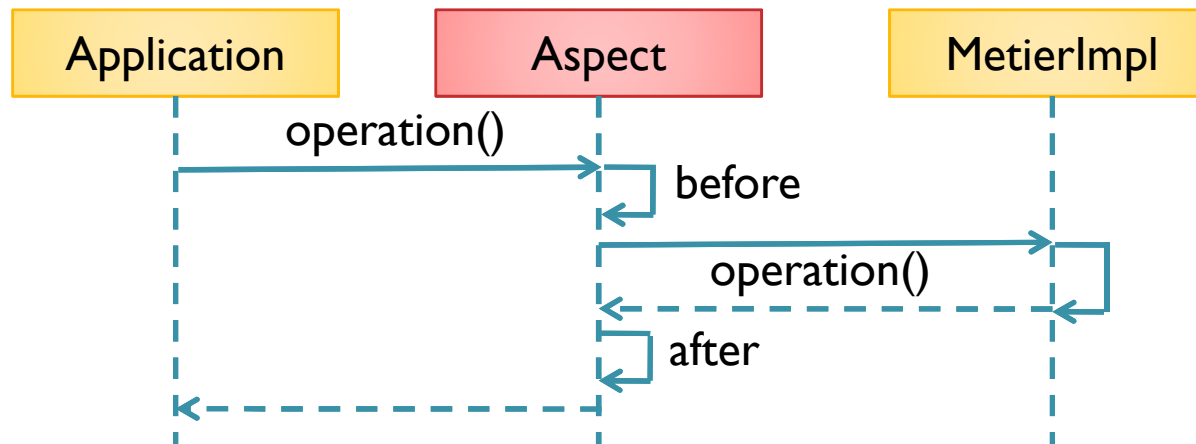
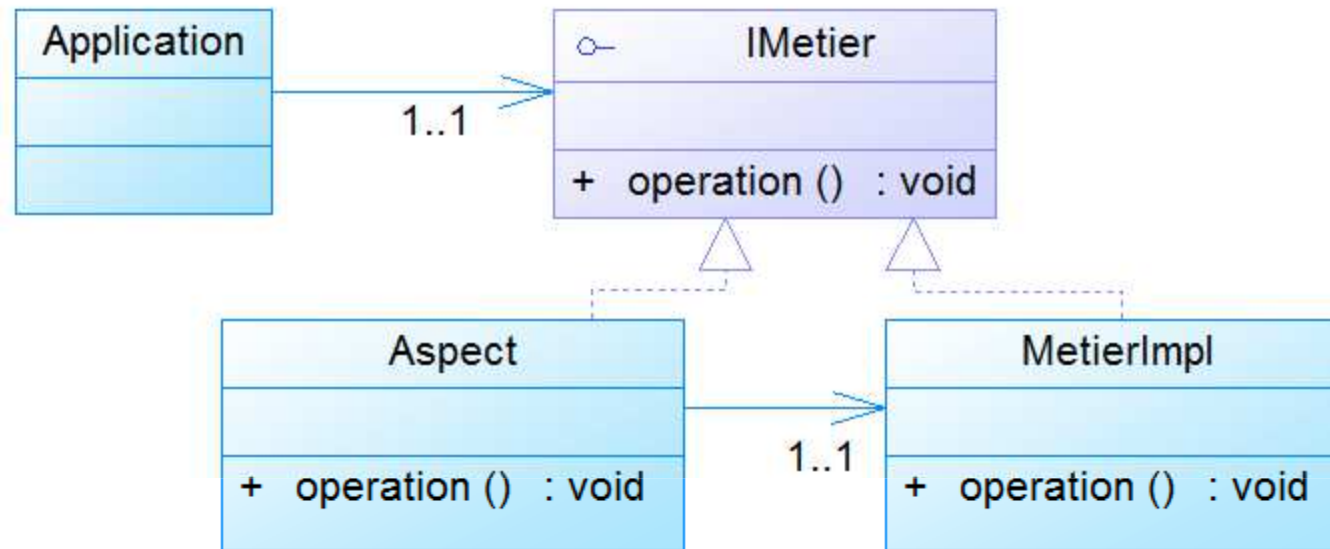
- L'opération de tissage peut être faite à la **compilation** ou à l'**exécution** du programme.
- **Les tisseurs statiques** s'appliquent à la compilation du programme. Ils prennent en entrée un ensemble de classes et un ensemble d'aspects pour fournir un programme compilé augmenté d'aspects.
- **Les tisseurs dynamiques** sont, quant à eux, capables d'appliquer les aspects dynamiquement, pendant l'exécution du programme. Leur principal atout est leur capacité à ajouter, supprimer ou modifier les aspects à chaud pendant l'exécution du programme, ce qui est très utile pour les serveurs d'applications.

Tissage à l'exécution

Tissage à la compilation : Statique



Tissage à l'exécution : Dynamique



Exemple de tisseurs d'aspects

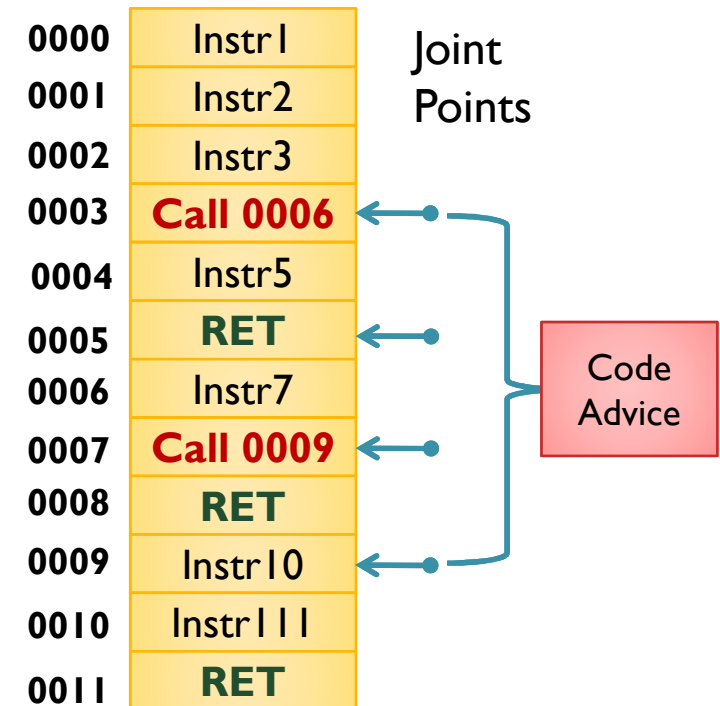
- Java :
 - **AspectJ** : Tisseur Statique (Extension du langage Java)
 - **Spring AOP** : Tisseur Dynamique
 - **JbossAOP** : Tisseur statique et dynamique
 - **JAC** (Java Aspect Components) :Tisseur Dynamique (Framework 100% Java).
 - **AspectWerkz** :Tisseur statique et dynamique (Framework 100% Java)
- C++ :
 - **AspectC++** : Tisseur Statique
- C#,VB :
 - **AspectDNG** : Tisseur Statique
- PHP :
 - **AspectPHP** : Tisseur Statique
- C :
 - **Aspect-C** : Tisseur Statique

Concepts de l'AOP

- Les points de jonction (**JoinPoint**)
- Les points de coupures (**PointCut**)
- Les **Wilcards**
- Les **Codes Advices**
- Mécanisme d'introduction

Les points de jonction (JoinPoint)

- Les points de jonction sont des points du programme autour desquels un ou des aspects ont été greffés.
- Ils peuvent être de différents type, c'est à dire qu'ils font références à des évènements du programme :
 - Appel de méthode
 - Appels de constructeur
 - Les attributs (get et set)
 - Levés des exceptions



Les points de coupure (JoinCut)

- Un point de coupure désigne un ensemble de point de jonctions.
- Il existe plusieurs types de points de coupures :
 - Les coupes d'appels de méthodes désignant un ensemble d'appels de méthodes.
 - Les coupes d'exécution de méthodes désignant l'exécution d'une méthode.
 - Les coupes de modification de données désignant des instructions d'écritures sur un ensemble d'attributs.
- La différence entre les coupes d'exécution et d'appel de méthodes est le contexte dans lequel sera la programme (**thisJoinPoint**)
- Dans le cas de coupe d'appel de méthodes, le contexte sera celui qui a appelé la méthode, et dans le cas de coupe à l'exécution le contexte sera celui de la méthode appelée.

```
pointcut test():call(* *.Point.get*());
```

```
pointcut getXValue(): execution(int *..Point.getX());
```

```
pointcut log():  
    call(void metier.Compte.verser(double)) ||  
    call(void metier.Compte.retirer(double));
```

Les Wilcards

- Un langage de programmation orienté aspect se doit de fournir au développeur une structure syntaxique permettant de déclarer une coupe.
- Chaque langage définit sa propre syntaxe.
- Les wilcards permettent de définir les points de coupures.
- Elles peuvent être comparées à des expressions régulières permettant de caractériser des points du programme.
- Nous pouvons par exemple, à l'aide du wilcard `get*(..)` pour définir l'ensemble des méthodes du programme commençant par get.

```
* *..Point.get*(..)
```

Les Codes Advices

- Les blocs Advices sont des blocs de code qu'exécutera un aspect.
- Les codes Advices caractérisent le comportement de l'aspect.
- Chaque code Advice d'un aspect doit être associé à une coupe pour être exécuté.
- Ils ne seront exécutés que si un événement définie par un point de coupure à été intercepté.
- Un code advice peut être exécuté selon trois modes : avant, après, ou autour d'un point de jonction.
- Lorsqu'il est exécuté autour du point de jonction, il peut carrément remplacer l'exécution de ce dernier, ou bien lui redonner le contrôle.

```
Object around() : getXValue(){  
    System.out.println("=> Entrée dans getX");  
    Object ret = proceed();  
    System.out.println("<= Sortie de la méthode getX");  
    return ret;  
}
```




Mécanisme d'introduction

- Le mécanisme d'introduction permet d'étendre la structure d'une application et non pas le comportement de cette dernière.
- En effet, le mécanisme d'introduction ne s'appuie pas sur la notion de coupe mais va opérer sur des emplacements bien définis dans le programme.
- On peut dire que le mécanisme d'introduction est pour l'orientée aspect ce que l'héritage est pour l'orientée objet puisque ces deux derniers permettent d'étendre la structure et non pas le comportement de l'application.

Premier exemple

```
Application.java FirstAspect.aj FirstAspect.java
package test;

public class Point {
    private int x;
    private int y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}

Application.java FirstAspect.aj FirstAspect.java
package test;

public class Application {
    public static void main(String[] args) {
        System.out.println("Exécution de Main");
        Point p=new Point();
        p.setX(80);
        p.setY(70);
        System.out.println("-----");
        System.out.println(p.getX());
        System.out.println(p.getY());
    }
}
```

Joint
Points

```
Application.java *FirstAspect.aj FirstAspect.java FirstAspect.aj Point.java
package aspects;

public aspect FirstAspect {

    WillCards
    pointcut mainCall():execution(* *.*.main*(..));

    before() : mainCall() {
        System.out.println("Before Main");
    }

    after():mainCall(){
        System.out.println("After Main");
    }

    WillCards
    pointcut callGetters():execution(* *.*.get*(..));

    before():callGetters(){
        System.out.println("before Get");
    }

    after():callGetters(){
        System.out.println("After Get");
    }
}
```

Code
Advice

Code
Advice

PointCut

Code
Advice

Code
Advice

Before Main
Exécution de Main

before Get
After Get
80

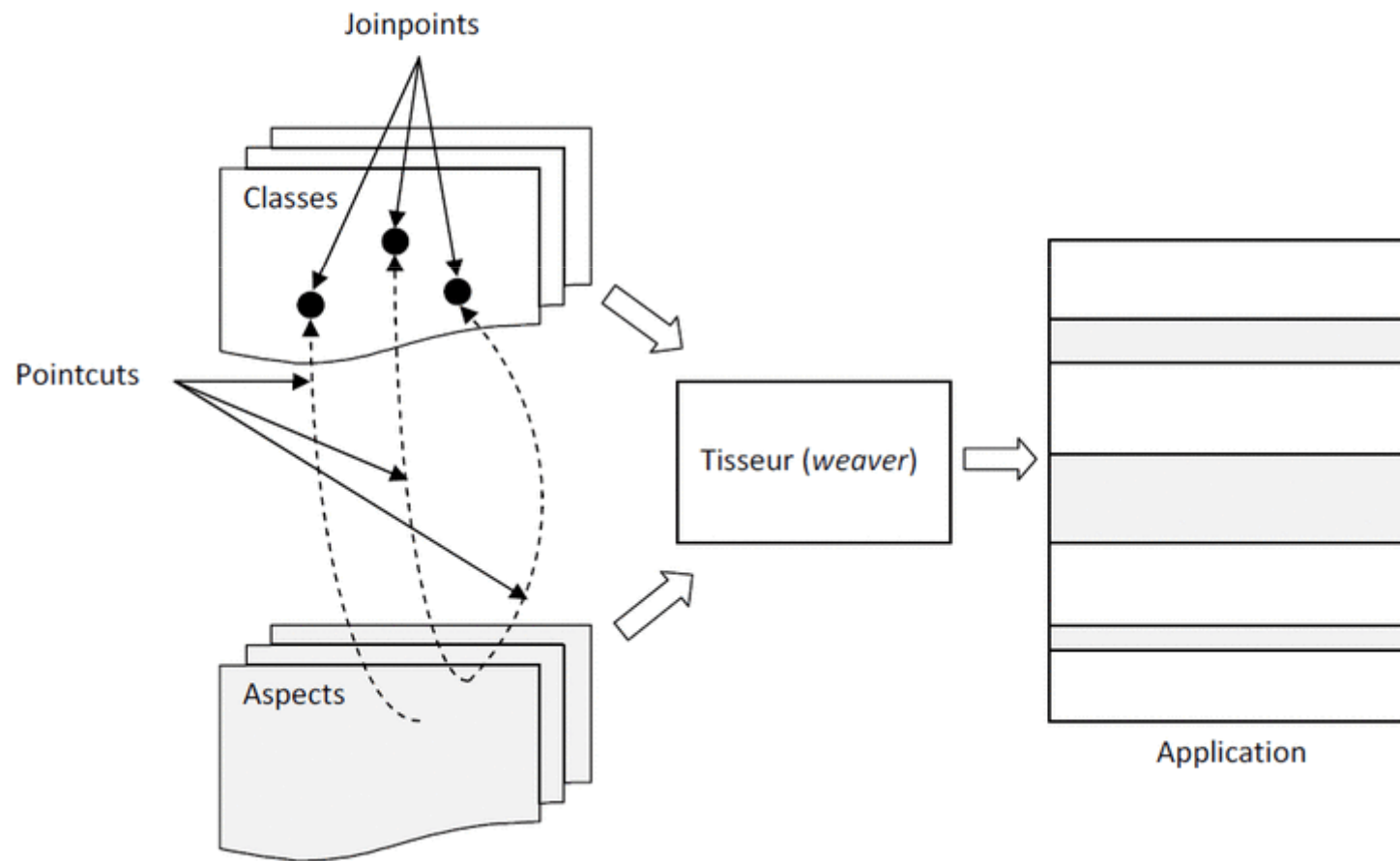
before Get
After Get
70
After Main

Tissage :Waving

- Une application orientée aspect contient des classes et des aspects. L'opération qui prends en entrée les classes et les aspects et produit une application qui intègre les fonctionnalités des classes et des aspects est connu sous le nom de tissage d'aspect (aspect weaving).
- Le programme qui réalise cette opération est appelé tisseur d'aspects (aspect weaver) ou bien tisseur (weaver) tout court.

Tissage :Waving

- Une application orientée aspect contient des classes et des aspects. L'opération qui prends en entrée les classes et les aspects et produit une application qui intègre les fonctionnalités des classes et des aspects est connu sous le nom de tissage d'aspect (aspect weaving)..

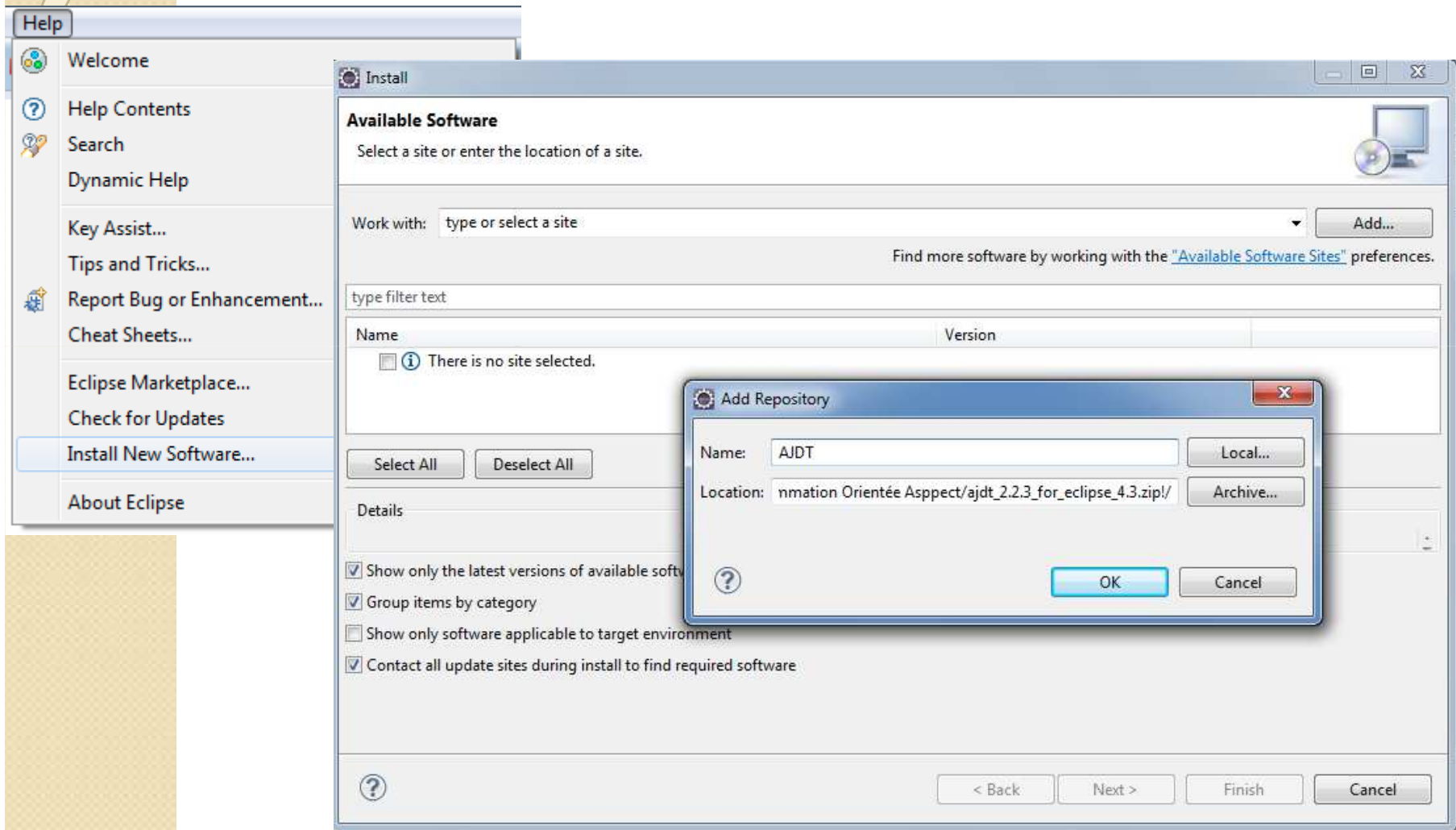




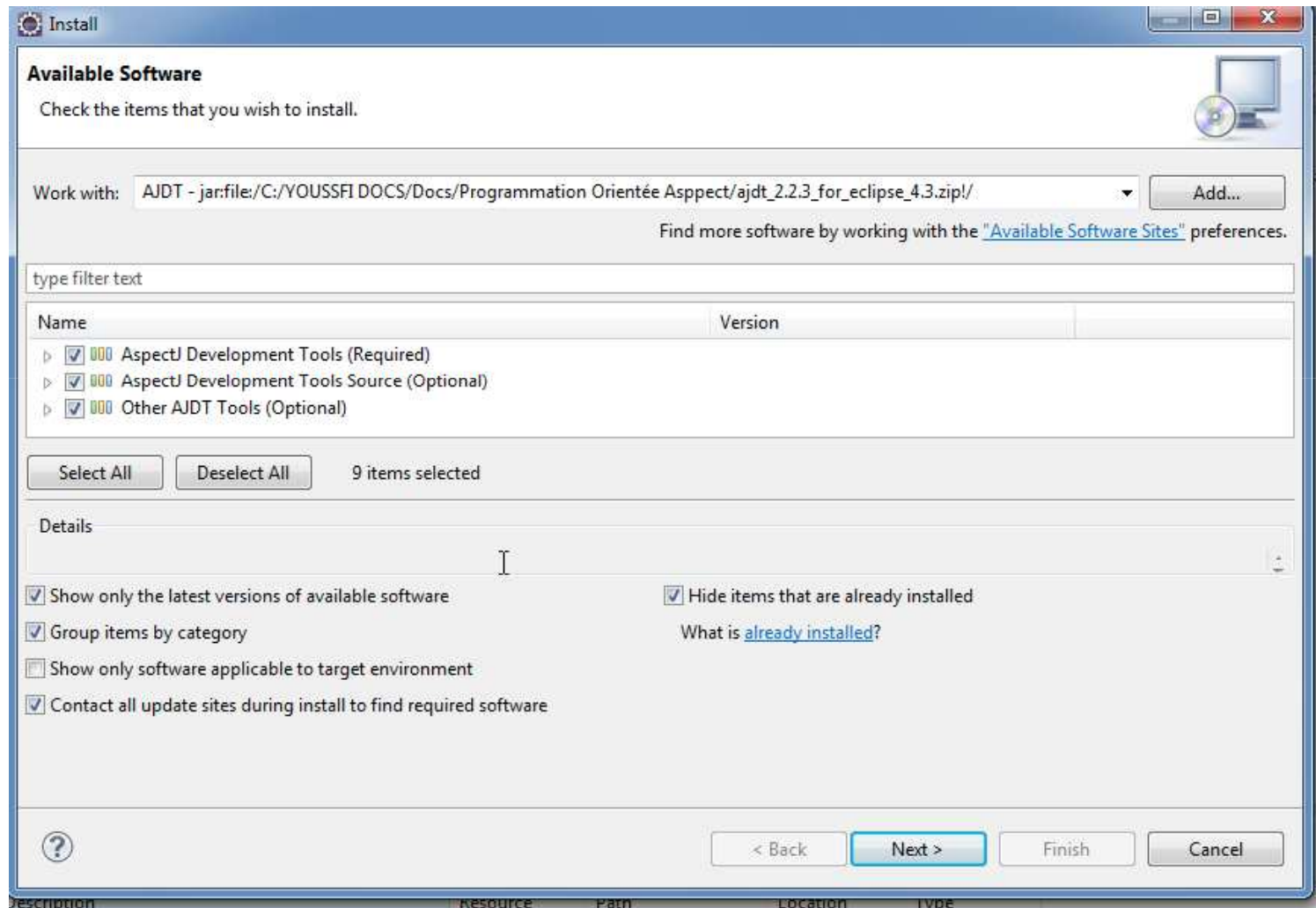
AspectJ : Historique

- L'histoire d'AspectJ est étroitement liée à celle de la programmation orientée aspect.
- En effet, Ce langage a été développé par la même équipe à l'origine de l'AOP.
- Un premier prototype d'AspectJ a été réalisé en 1998.
- La première version officielle d'AspectJ, désigné AspectJ 1.0, a été réalisée en novembre 2001, Durant cette année, l'AOP a été complètement reconnue par la communauté informatique mondiale, et une édition spéciale du journal *Communications of the ACM* a été dédiée à l'AOP.
- En décembre 2002, le projet AspectJ a quitté XEROX PARC et a rejoint la communauté open-source Eclipse.
- Et depuis, le plugin Eclipse *AspectJ Development Tools (AJDT)* est développé.
- Ce plugin intègre AspectJ et permet d'écrire, de compiler et d'exécuter des programmes orientés aspects dans l'environnement de développement Eclipse.

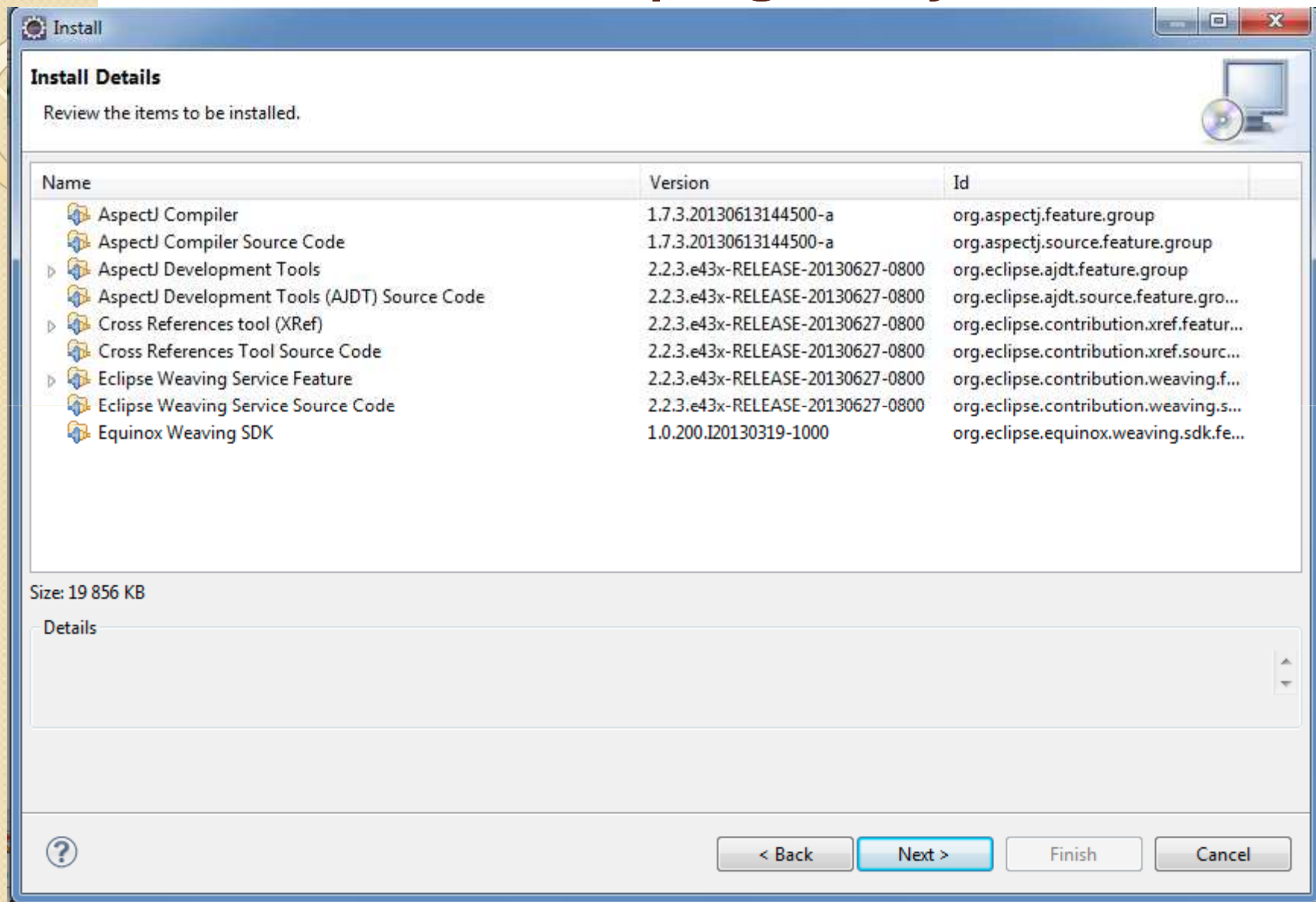
Installation du plugin :AJDT



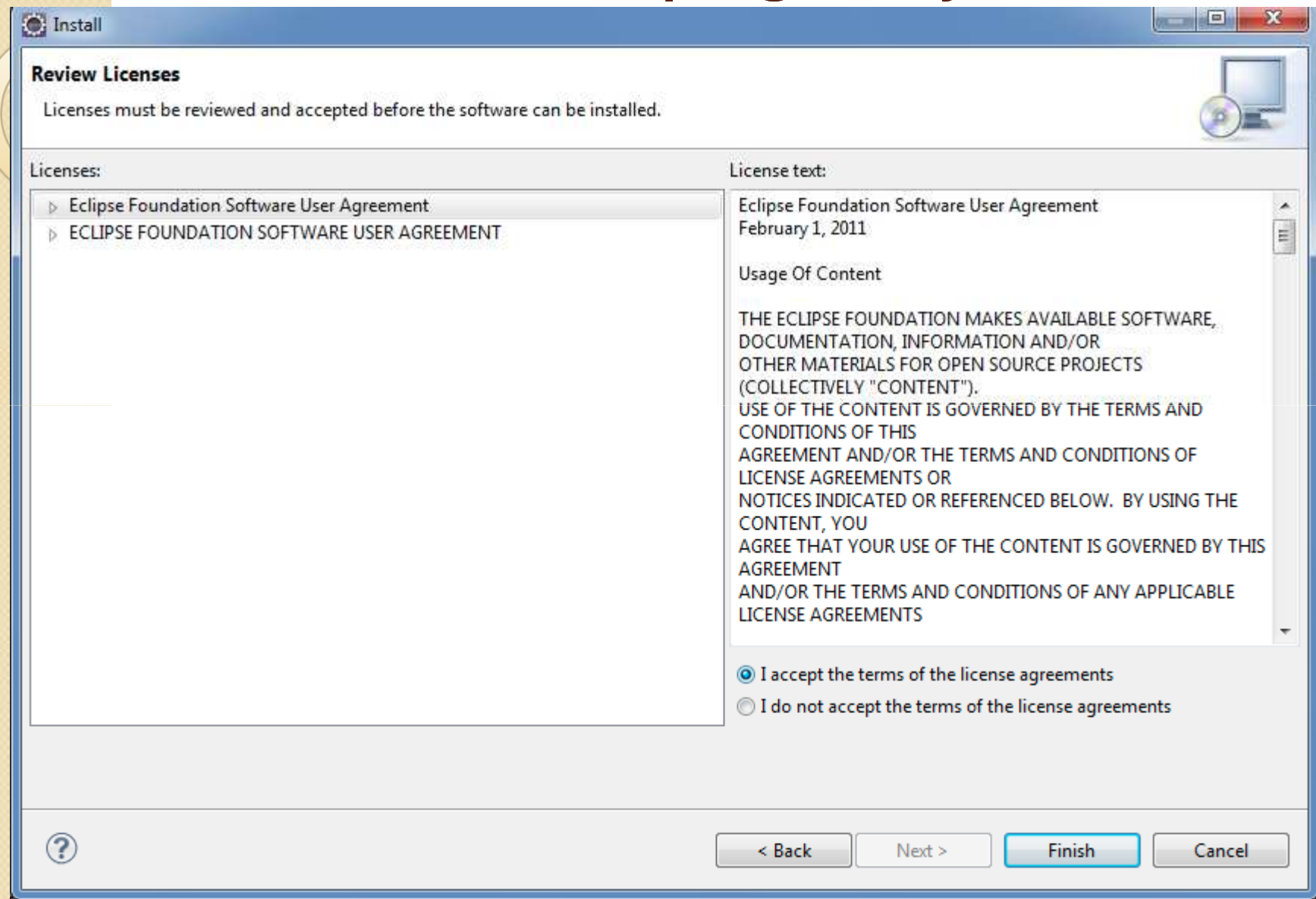
Installation du plugin :AJDT



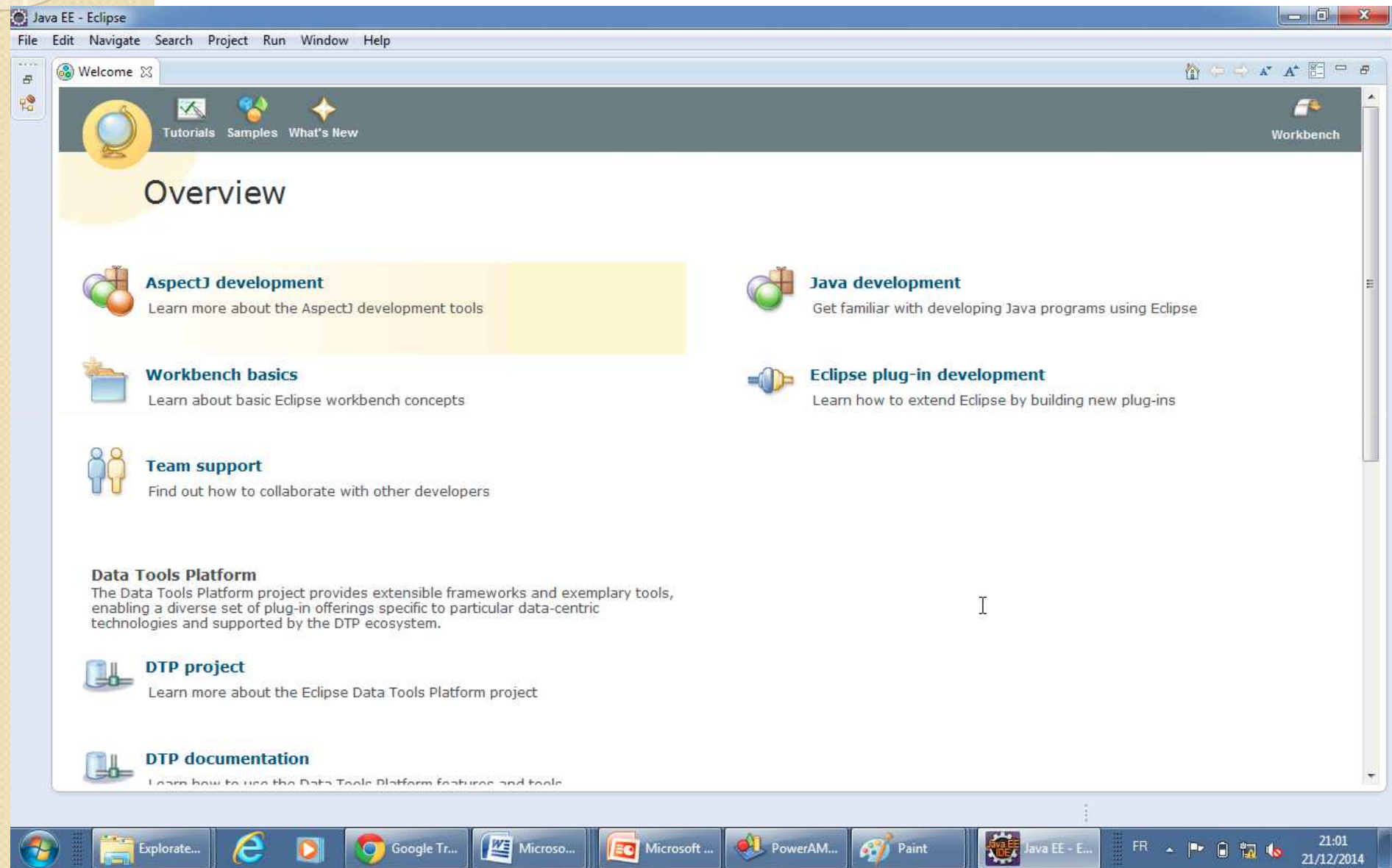
Installation du plugin :AJDT



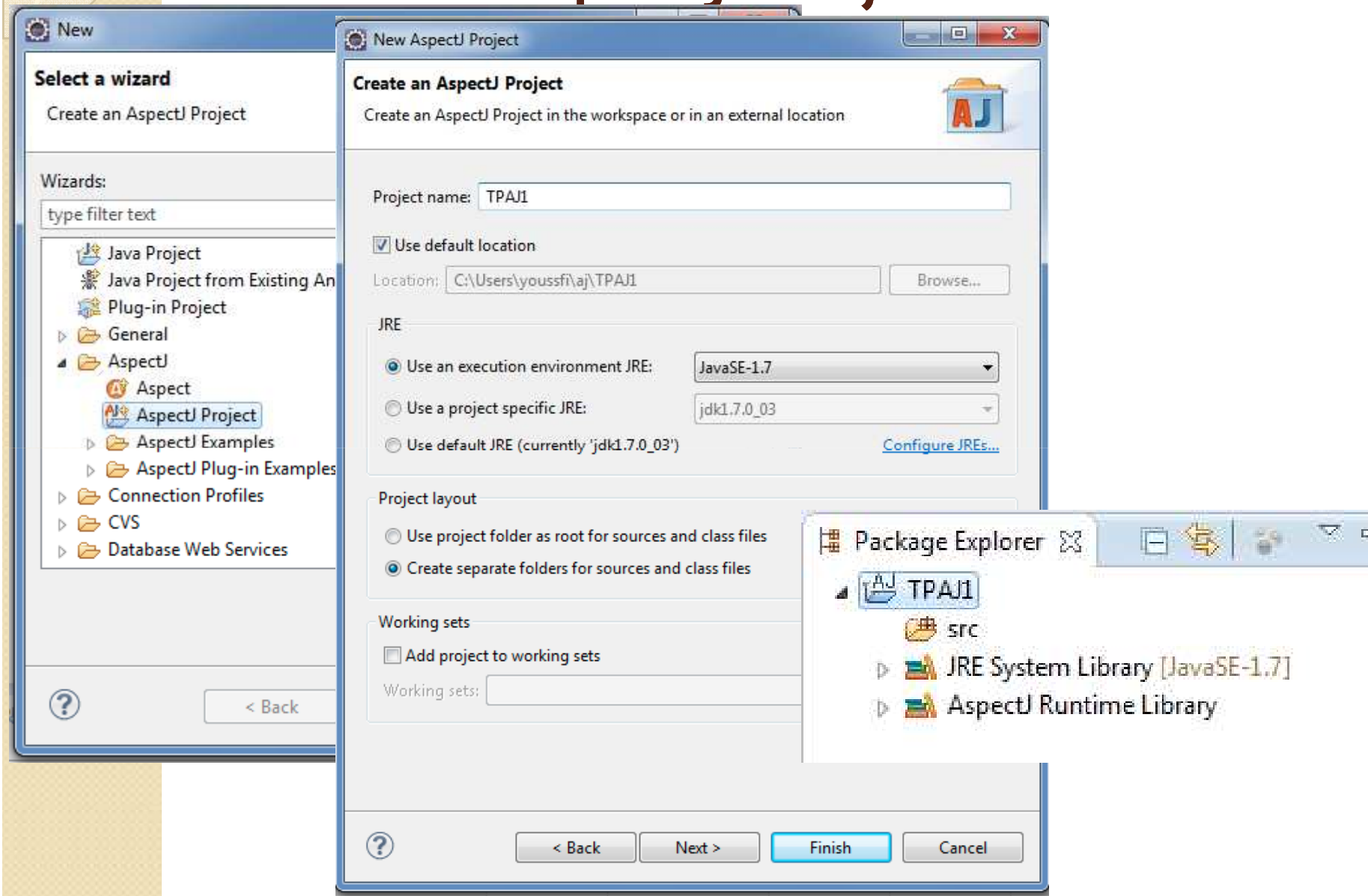
Installation du plugin :AJDT



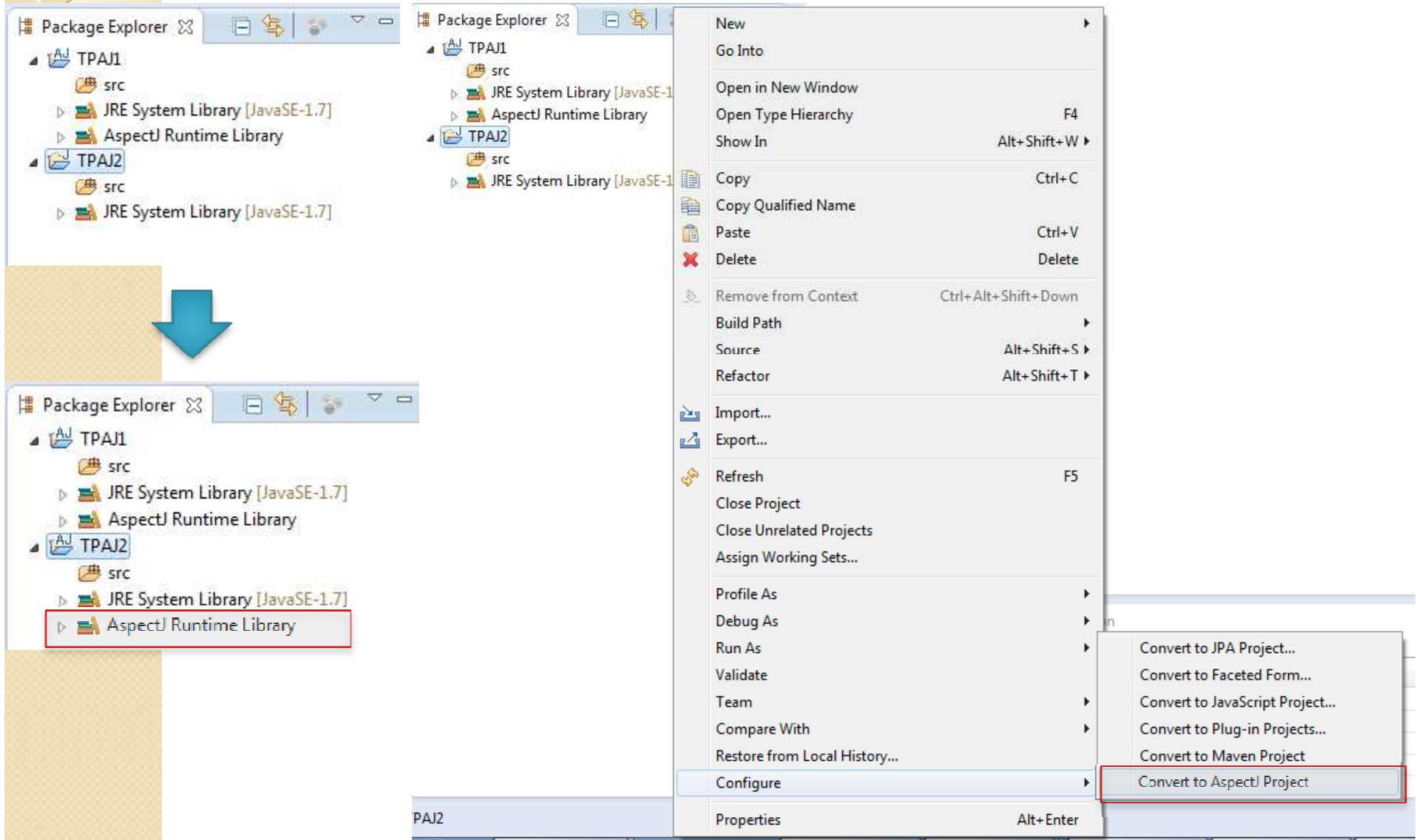
Installation du plugin :AJDT



Nouveau AspectJ Projet



Ajouter AspectJ à un projet quelconque



Aspect J : Présentation

- AspectJ est aujourd'hui une implémentation orientée aspect phare qui fournit un excellent support pour appréhender les concepts de la programmation orientée aspect.
- AspectJ est une extension orientée aspect du langage de programmation Java.
- Il permet de déclarer des aspects, des coupes, des codes advices et des introductions.
- Il offre aussi un tisseur d'aspect appelé **ajc** (pour *AspectJ/Compiler*) qui prends en entrée des classes java et des aspects, et produit en sortie des classes dont le comportement est augmenté par les aspects.
- AspectJ permet de définir deux types de transversalités avec les classes de base : transversalité statique (static crosscutting) et transversalité dynamique (dynamic crosscutting).

Aspect J : Transversalité statique

- **Static crosscutting :**
 - La transversalité statique consiste à augmenter la structure des classes.
 - Pour cela AspectJ offre la notion de mécanisme d'introduction qui permet entre autres d'ajouter des éléments structuraux comme des attributs ou des méthodes aux classes.
 - Le mécanisme d'introduction d'AspectJ permet aussi d'ajouter des liens entre des classes comme l'héritage et l'implémentation d'interfaces.
 - Concrètement, AspectJ offre un support simple et facile à appréhender pour implémenter ce genre de transversalité.

Aspect J : Transversalité dynamique

- **Dynamic crosscutting :**
 - La transversalité dynamique consiste à augmenter le comportement des classes.
 - Pour cela AspectJ offre les notions de coupes et de code advices.
 - Les coupes servent à sélectionner des points précis dans les classes.
 - Et les advices iront se greffer avant, après ou autour de ces points afin d'étendre leur comportement.

Points de jonction

- En réalité, Un point de jonction est n'importe quel point d'exécution dans un système.
- Parmi tous les points de jonction possibles dans un système on cite de façon non exhaustive :
 - L'appel à une méthode ;
 - L'exécution d'une méthode ;
 - L'affectation de variable ;
 - L'appel au constructeur d'une classe ;
 - Une instruction conditionnelle (i.e. IF/THEN/ELSE) ;
 - Le traitement d'une exception ;
 - Les boucles (i.e. FOR, WHILE, DO/WHILE) ;
 - etc...

Points de jonction dans AspectJ

Point de jonction	Description
Method call	Quand une méthode est appelée
Method execution	Quand le corps d'une méthode est exécuté
Constructor call	Quand un constructeur est appelé
Constructor execution	Quand le corps d'un constructeur est exécuté
Static initializer execution	Quand l'initialisation statique d'une classe est exécutée
Object pre-initialization	Avant l'initialisation de l'objet
Object initialization	Quand l'initialisation d'un objet est exécutée
Field reference	Quand un attribut non-constant d'une classe est référencé
Field set	Quand un attribut d'une classe est modifié
Handler execution	Quand un traitement d'une exception est exécuté
Advice execution	Quand le code d'un advice est exécuté

- Dans AspectJ, Tout les points de jonction ont un contexte associé à eux.
 - Par exemple, le contexte d'un point de jonction correspondant à un appel de méthode contient l'objet appelant, l'objet appelé, et les arguments de la méthode.
 - De la même manière, le contexte d'un point de jonction correspondant au traitement d'une exception contient l'objet courant, et l'exception levée.

Premier Aspect

```
package aspects;
public aspect FirstAspect {
    // Coupe pointcut:
    pointcut mainCall() :
    execution(public static void main(String[]));
    // Code Advice :
    ↪ before() : mainCall() {
        System.out.println("#### Aspect ####");
        System.out.println("Avant Main");
    }
    // Code Advice :
    ↪ after() : mainCall() {
        System.out.println("#### Aspect ####");
        System.out.println("Après Main");
    }
}
```

Au moment de la compilation, le tisseur d'aspects ajoutera le comportement défini dans le code advice FirstAspect.aj avant et après l'exécution de la méthode main.

```
package test;
public class Application {
    //Point de jonction (jointpoint)
    ↪ public static void main(String[] args) {
        System.out.println("*****");
        System.out.println("Main message...");
        System.out.println("*****");
    } }
```

Exécution de l'application

```
#### Aspect ####
Avant Main
*****
Main message...
*****
#### Aspect ####
Après Main
```

Premier Aspect en utilisant les annotations

```
package aspects;

import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SecondAspect {

    @Pointcut("execution(* *.*.main(..))")
    public void pc1(){}

    @Before("pc1()")
    public void beforeMain(){
        System.out.println("Aspect 2");
        System.out.println("Avant");
    }

    @After("pc1()")
    public void afterMain(){
        System.out.println("Aspect 2");
        System.out.println("Après");
    }
}
```

Coupes simples:

- Une coupe est introduite grâce au mot clé **pointcut**.
- La syntaxe est :
pointcut nomDeLaCoupe(paramètres) : définitionDeLaCoupe ;
- Le mot clé alors utilisé pour définir la coupe peut être **call** ou **execution**. Cela dépend si l'on souhaite exécuter le code advice lors de l'appel ou dans le code de la méthode exécutée. Pour résumer la différence entre call et execution, voici l'ordre d'exécution des codes advice liés :
 - 1 - Code Advice de type « before » associé au point de jonction call
 - 2 - Code Advice de type « before » associé au point de jonction Exécution
 - 3 - Code de la méthode
 - 4 - Code Advice de type « after » associé au point de jonction call
 - 5 - Code Advice de type « after » associé au point de jonction execution

Tableau récapitulatif des points de jonctions possibles

Syntaxe	Description du point de jonction
call (methodeExpression)	Appel d'une méthode dont le nom vérifie methodeExpression.
execution (methodeExpression)	Exécution d'une méthode dont le nom vérifie methodeExpression.
get (attributExpression)	Lecture d'un attribut dont le nom vérifie attributExpression. Exemple : get (int Point.x)
set (attributExpression)	Écriture d'un attribut dont le nom vérifie attributExpression.
handler (exceptionExpression)	Exécution d'un bloc de récupération d'une exception (catch) dont le nom vérifie exceptionExpression. Exemple : handler (IOException+)
initialization (constanteExpression)	Exécution d'un constructeur de classe dont le nom vérifie constanteExpression. Exemple : initialization (Customer. new (..))
preinitialization (constanteExpression)	Exécution d'un constructeur hérité dont le nom vérifie constanteExpression.
staticinitialization (classeExpression)	Exécution d'un bloc de code static dans une classe dont le nom vérifie classeExpression. Exemple : staticinitialization (Point)
adviceexecution ()	Exécution d'un code advice.

Coupes génériques : wildcards

- Une première manière consiste à utiliser des wildcards.
- Les wildcards sont des caractères permettant de définir de manière plus vaste une méthode ou une classe, etc...
- Les wildcards permettent d'obtenir ce que l'on pourrait appeler des expressions régulières simples.

Wildcard	Utilisation
*	Remplace un nom (de classe, de paquetage, de méthode, d'attribut, etc..) ou simplement une partie de nom. Il peut aussi remplacer un type de retour ou un paramètre . Exemple : <code>call(* x.y.*.test.*.set*(int, String, *));</code>
..	Utilisé pour omettre les paramètres des méthodes ou le chemin complet des paquetages. Exemple pour une expression sur une méthode : <code>public void org..Test.methode(..)</code>
+	Permet de définir n'importe quel sous-type d'une classe ou d'une interface. Exemple pour une expression sur une méthode : <code>void x.y.test.IMouseListener+.set* (..) ;</code> Cet exemple signifie « toutes les méthodes commençant par set des classes implémentant l'interface IMouseListener.

Coupes génériques : wildcards

- Les wildcards permettent de généraliser des coupes.
- Par exemple, l'expression suivante permet par exemple de réunir tous les points de jonctions de type appel à une méthode commençant par set :

pointcut allSet() : **call** (* *..set* (..));

Opérateurs logiques et mot-clef de filtrage

- Il est également possible de généraliser des coupes via des opérateurs logiques.
- En effet, il est possible de combiner des points de jonction avec l'opérateur logique ou (**||**).

pointcut allGetSet():

```
call(void *..set* (..)) ||
```

```
call(* *..get*());
```

Opérateurs logiques et mot-clef de filtrage

Mot-clef	Signification
&&	Et logique.
 	Ou logique.
!	Négation logique.
if (ExpressionBooléenne)	Exemple : pointcut test(): if (thisJoinPoint .getArgs().length == 1) && execution (* *.*(..));
withincode (methodeExpression)	Vrai lorsque le point de jonction est défini dans une méthode dont la signature vérifie methodeExpression.
within (typeExpression)	Vrai si l'événement se passe pendant l'exécution d'un code embarqué par une classe définie par typeExpression (cela inclut les méthodes statiques). Permet essentiellement de limiter la portée d'un autre pointcut à une certaine classe.
this (typeExpression)	Vrai lorsque l'événement se passe à l'intérieur d'une instance de classe de type typeExpression.
target (typeExpression)	Vrai lorsque le type de l'objet destination du point de jonction vérifie typeExpression.
cflow (coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (y compris l'entrée et la sortie).
cflowbelow (coupe)	Vrai pour tout point de jonction situé entre l'entrée dans la coupe et sa sortie (sauf pour l'entrée et la sortie).

Exemples

- Par exemple, l'expression suivante désigne tous les appels à la méthode `setX` de `Point` depuis la classe nommée `MainExemple`.

call (* *.Point.setX(..)) **&& this** (* *.MainExemple)

Cela permet d'exclure de la coupe les appels à la méthode effectués depuis les autres classes.

- De même on peut désigner les appels depuis un code de méthode particulier

get (* ..Point.x) **&& !withincode** (* ..Point.getX())

Dans cet exemple nous désignons tous les endroits où le paramètre `x` est lu en dehors de la méthode `getX`.

- Il est également possible de définir toutes les lectures de `x` à l'extérieur de la classe `Point` grâce au mot-clef `within`.

get (* ..Point.x) **&& !within** (* ..Point)

Code Advice

- Une fois la coupe définie, il faut écrire le code à exécuter au moment où l'événement décrit par la coupe est levé.
- Avant d'écrire ce code advice il faut décider à quel moment exécuter le code : **avant** l'événement, **après** ou **autour** de l'événement.
- Le mot-clef **before** permet d'exécuter du code avant d'entrer dans le code lié à l'événement
- Le mot-clef **after** permet quant à lui d'exécuter du code après l'événement.
- Le mot-clef **around** permet soit de remplacer l'événement en lui-même, soit d'exécuter du code avant et après le point de jonction.
- Pour exécuter le code de l'événement il faut utiliser le mot-clef **proceed** à l'intérieur du code advice.
- Si la méthode a un type de retour, alors il faut faire précéder le mot-clef around de **Object**.
- L'appel à proceed renvoie alors un type Object qu'il est possible de récupérer pour le renvoyer.

Code Advice

- Voici un exemple très simple permettant de tracer l'exécution de la méthode getX.

```
pointcut getXValue(): execution(int *..Point.getX());  
Object around() : getXValue(){  
    System.out.println("=> Entrée dans getX");  
    Object ret = proceed();  
    System.out.println("<= Sortie de la méthode getX");  
    return ret;  
}
```

Introspection et paramétrage des coupes

- L'introspection permet de connaître dans un code advice quelles sont les informations sur le point de jonction ayant provoqué l'événement.
- Le mot-clef **thisJoinPoint** utilisé dans un code advice renvoie un objet de la classe *org.aspectj.lang.JoinPoint*

Méthodes	Signification
Object[] getArgs()	retourne les arguments de l'appel.
String getKind()	Retourne une chaîne de caractères représentant le type du point de jonction.
Signature getSignature()	Retourne la signature d'un point de jonction.
SourceLocation getSourceLocation()	Retourne la localisation du point de jonction dans le code source.
JoinPoint.StaticPart getStaticPart()	Retourne un objet encapsulant les informations statiques concernant ce point de jonction.
Object getTarget()	renvoie l'instance de l'objet appelé.
Object getThis()	Retourne l'objet appelant
String toLongString()	Retourne une chaîne représentant une description complète du point de jonction.
String toShortString()	Retourne une chaîne décrivant succinctement le point de jonction.

Exemple d'utilisation de thisJoinPoint

```
package aspects;
import graphics.Point; import test.Application;
public aspect FirstAspect {
pointcut test():call(* *.Point.get*());
Object around():test(){
    System.out.println("*****");
    Application source=(Application) thisJoinPoint.getThis();
    Point target=(Point)thisJoinPoint.getTarget();
    System.out.println("Avant l'appel de la méthode :");
    System.out.println(thisJoinPoint.getSignature());
    System.out.println("de l'objet "+target);
    System.out.println("par :"+source);
    Object res=proceed();
    System.out.println("Après exécution :");
    System.out.println("Résultat="+res);
    System.out.println("*****");
    return res;
}}
```


Mécanisme d'introduction

- Un aspect peut ajouter des comportements comme de la journalisation à des classes mais il peut également ajouter des attributs, des méthodes, des constructeurs ou encore des interfaces ou une classe parente.
- Ce mécanisme s'appelle l'introduction et utilise encore une fois avec AspectJ une syntaxe particulière.
- Imaginons que nous ayons un certain nombre de classes pour lesquelles nous souhaitons pouvoir positionner une date et la récupérer.
- Nous souhaitons de plus initialiser cette date dans le constructeur.
- Nous allons alors définir une interface `DateInside` précisant les méthodes possibles sur ce type de classe (typiquement `getDate` et `setDate`).
- Nous allons ensuite définir un aspect permettant d'ajouter ces fonctions à une classe `ClassI`.
- Le code de l'interface est le suivant :

Exemple d'introduction

- On considère une classe vide suivante:

```
package dao;  
public class DaoImpl { }
```

- Une interface suivante

```
package dao;  
public interface IDao {  
    public double getvalue();  
    public void setValue(double v);  
}
```

- On veut créer un aspect qui permet qui permet d'ajouter à la classe DaoImpl :
 - Implémentation de l'interface IDao.
 - Un attribut value de type double
 - Une implémentation des deux méthodes de l'interface

Exemple d'aspect d'introduction

- On veut créer un aspect qui permet qui permet d'ajouter à la classe DaoImpl :
 - Implémentation de l'interface IDao.
 - Un attribut value de type double
 - Une implémentation des deux méthodes de l'interface

```
DaoAspect.aj
package dao;

public aspect DaoAspect {
    // Ajout de l'interface IDao à la classe DaoImpl
    declare parents:DaoImpl implements IDao;
    // Ajout d'un attribut value de type double à la classe
    private double DaoImpl.value;
    // Ajout de l'implémentation des deux méthodes de l'interface
    public double DaoImpl.getValue(){
        return value;
    }
    public void DaoImpl.setValue(double v){
        this.value=v;
    }

    // Advice avant instantiation
    before():initialization(DaoImpl.new(..)){
        System.out.println("Avant instantiation");
        DaoImpl dao=(DaoImpl) thisJoinPoint.getTarget();
        dao.setValue(80);
    }
}
```

```
DaoAspect.aj  DaoImpl.java
package dao;

public class DaoImpl {
}
```

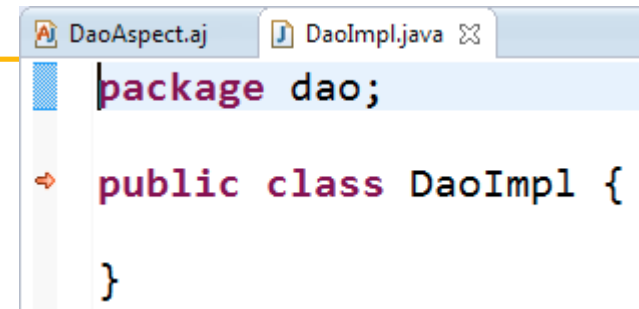
Exemple d'aspect d'introduction

- On veut créer un aspect qui permet qui permet d'ajouter à la classe DaoImpl :
 - Implémentation de l'interface IDao.
 - Un attribut value de type double
 - Une implémentation des deux méthodes de l'interface

```
package dao;
public aspect DaoAspect {
    // Ajout de l'interface IDao à la classe DaoImpl
    declare parents:DaoImpl implements IDao;
    // Ajout d'un attribut value de type double à la classe
    private double DaoImpl.value;
    // Ajout de l'implémentation des deux méthodes de l'interface
    public double DaoImpl.getValue(){
        return value;
    }
    public void DaoImpl.setValue(double v){
        this.value=v;
    }
    // Advice avant instantiation
    before():initialization(DaoImpl.new(..)){
        System.out.println("Avant instantiation");
        DaoImpl dao=(DaoImpl) thisJoinPoint.getTarget();
        dao.setValue(80);
    }
}
```

Exemple d'aspect d'introduction

```
package dao;  
public class Test {  
    public static void main(String[] args) {  
        DaoImpl dao=new DaoImpl();  
        System.out.println("V="+dao.getValue());  
        dao.setValue(90);  
        System.out.println("V="+dao.getValue());  
    }  
}
```



```
package dao;  
  
public class DaoImpl {  
  
}
```

Avant instantiation

V=80.0

V=90.0



Application

Exemple I : Une classe Compte

```
package metier;

public class Compte {
    private int code; private double solde;
    public Compte() { }
    public void verser(double mt){ solde+=mt; }
    public void retirer(double mt){ solde-=mt; }
    public void virement(Compte cp2,double mt){
        this.retirer(mt);
        cp2.verser(mt);
    }
    public int getCode() { return code; }
    public void setCode(int code) { this.code = code; }
    public double getSolde() { return solde; }
    @Override
    public String toString() { return "Compte [code=" + code + ", solde=" +
        solde + "]\n";
    }
}
```

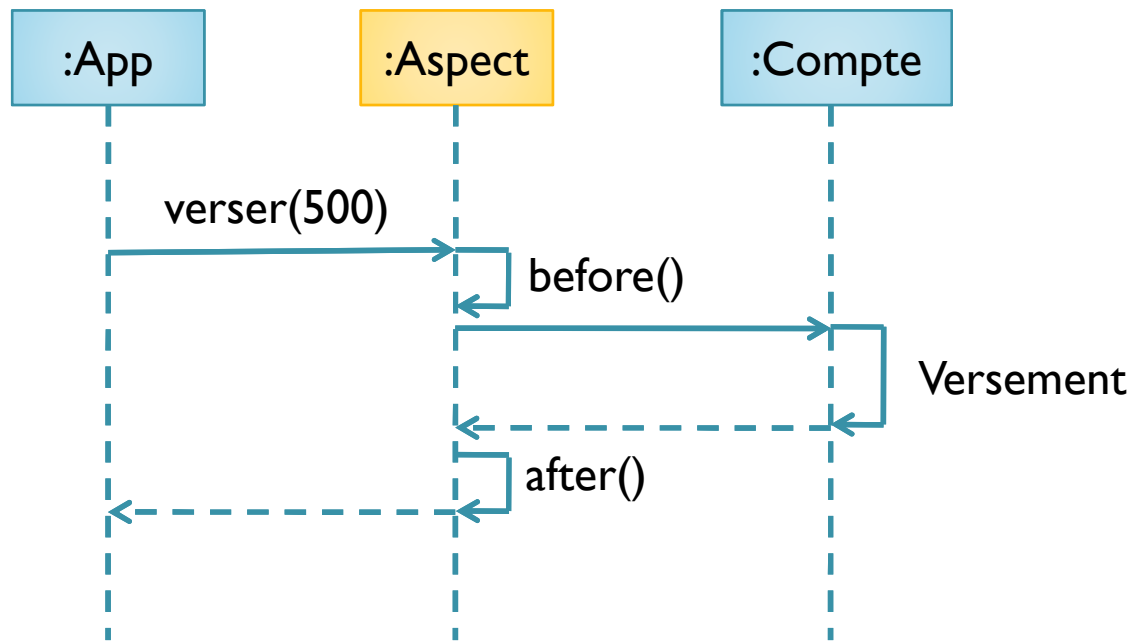

Exemple I : Utilisation de la classe Compte

```
package test;
import metier.Compte;
public class Test {
    public Test() {
        Compte cp1=new Compte(); Compte cp2=new Compte();
        cp1.setCode(1); cp1.verser(800); cp1.retirer(9000);
        cp2.setCode(2); cp1.virement(cp2, 9000);
        System.out.println(cp1.toString());
        System.out.println(cp2.toString());
    }
    public static void main(String[] args) { new Test(); }
}
```

Compte [code=1, solde=-17200.0]
Compte [code=2, solde=9000.0]

Premier Aspect

- Créer un aspect qui permet de :
 - Journaliser un message avant et après l'exécution des méthodes verser, retirer et virement
 - Journaliser la durée d'exécution de chacune des trois méthodes



Premier Aspect

- Créer un aspect qui permet de :
 - Journaliser un message avant et après l'exécution des méthodes verser, retirer et virement
 - Journaliser la durée d'exécution de chacune des trois méthodes

```
package aspects;
import java.util.logging.Logger;
public aspect LoggerAspect {
    Logger logger=Logger.getLogger(this.getClass().getName());
    long t1,t2;
    pointcut log():
        call(void metier.Compte.verser(double))||call(void metier.Compte.retirer(double))||
        call(void metier.Compte.virement(..));

    before():log(){
        t1=System.currentTimeMillis();logger.info("-----");
        logger.info("Avant "+thisJoinPoint.getSignature());
    }
    after():log(){
        t2=System.currentTimeMillis();logger.info("-----");
        logger.info("Après "+thisJoinPoint.getSignature());
        logger.info("Durée d'exécution="+(t2-t1));
    }
}
```

En utilisant les annotations

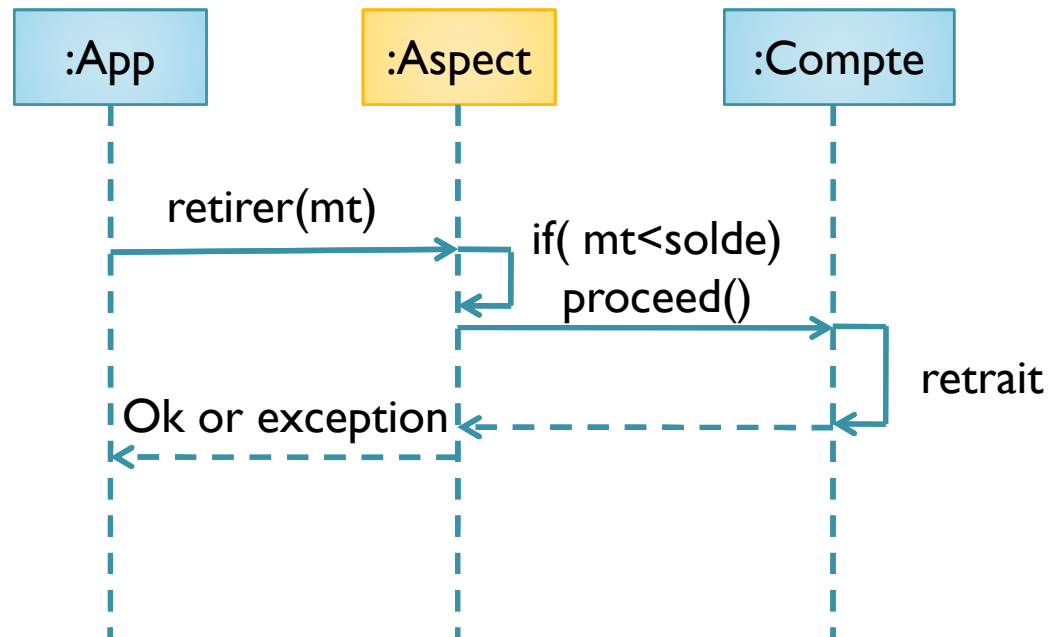
- Créer un aspect qui permet de :
 - Journaliser un message avant et après l'exécution de toutes les méthodes de la classe Compte
 - Journaliser la durée d'exécution de chacune chaque méthode

```
package aspects;
import java.util.logging.Logger; import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After; import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class LoggerAspect2 {
    Logger logger=Logger.getLogger(this.getClass().getName());
    long t1,t2;

    @Before("call(* metier.Compte.*(..))")
    public void avant(JoinPoint thisJoinPoint){
        t1=System.currentTimeMillis();logger.info("*****");
        logger.info("Avant "+thisJoinPoint.getSignature());
    }
    @After("call(* metier.Compte.*(..))")
    public void apres(JoinPoint thisJoinPoint){
        t2=System.currentTimeMillis(); logger.info("*****");
        logger.info("Après "+thisJoinPoint.getSignature());
        logger.info("Durée d'exécution="+t2-t1);
    }
}
```

Exemple 2

- Créer un aspect qui permet de :
 - Pour chaque appel de la méthode retirer, de vérifier le solde du compte est supérieur au montant à retirer.
 - Si le solde est insuffisant l'aspect doit générer une exception



Exemple 2

- Créer un aspect qui permet de :
 - Pour chaque appel de la méthode retirer, de vérifier le solde du compte est supérieur au montant à retirer.
 - Si le solde est insuffisant l'aspect doit générer une exception

```
package aspects;

import metier.Compte;
public aspect PatcherAspect {
    pointcut patch(double mt):call(* metier.Compte.retirer(double)) && args(mt);

    void around(double mt) : patch(mt) {
        Compte compte=(Compte)thisJoinPoint.getTarget();
        if(mt<compte.getSolde()){
            proceed(mt);
        }
        else{
            throw new RuntimeException("Solde insuffisant");
        }
    }
}
```

Exemple 2 : Avec Annotations

```
package aspects;
import metier.Compte;
import org.aspectj.lang.*; import org.aspectj.lang.annotation.*;
@Aspect
public class PatchAspect2 {
    @Pointcut("call(* metier.Compte.retirer(..)) && args(mt)")
    public void pathPointCut(double mt,ProceedingJoinPoint pjp){}

    @Around("pathPointCut(mt,pjp)")
    public void patch(double mt,JoinPoint joinPoint,ProceedingJoinPoint pjp){
        Compte compte=(Compte)joinPoint.getTarget();
        if(compte.getSolde(>mt){
            try {
                pjp.proceed(new Object[]{mt});
            } catch (Throwable e) {
                e.printStackTrace();
            }
        }
        else{
            throw new RuntimeException("Solde insuffisant [mt="+mt+"
Solde="+compte.getSolde()+"]");
        }
    }
}
```


Exemple 3 : SecurityAspect

- Soit l'application non sécurisée suivante :

```
package metier;
import java.util.Scanner;
public class App {
public static void main(String[] args) {
try {
Compte compte =new Compte(); Scanner clavier=new Scanner(System.in);
System.out.print("Code :"); int code=clavier.nextInt();
compte.setCode(code);
while(true){
    System.out.print("Montant à verser :");
    double mt1=clavier.nextDouble();
    compte.verser(mt1);
    System.out.println(compte.toString());
    System.out.print("Montant à Retirer :");
    double mt2=clavier.nextDouble();
    compte.retirer(mt2);
    System.out.println(compte.toString());
}}
catch (Exception e) {
    System.out.println(e.getMessage());
}}}
```

```
Code :4
Montant à verser :8000
Compte [code=4, solde=8000.0]
Montant à Retirer :900
Compte [code=4, solde=7100.0]
Montant à verser :
```

Exemple 3 : SecurityAspect

- On souhaite sécuriser cette application par un aspect qui oblige l'utilisateur à saisir un login et un mot de passe valide avant d'accéder à cette application

```
package aspects; import java.util.Scanner;
public aspect SecurityAspect {
    private String login;
    private String pass;
    pointcut security1():execution(* *.main(..));
    void around() : security1() {
        if(login==null){
            Scanner clavier=new Scanner(System.in);
            System.out.print("Login:");
            String l=clavier.next();
            System.out.print("Pass:");
            String p=clavier.next();
            if(l.equals("root") && p.equals("root")){
                login=l; pass=p;
                proceed ();
            }
            else System.out.println("Accès refusé");
        }
    }
}
```

```
Login:aa
Pass:aa
Accès refusé
```

```
Login:root
Pass:root
Code :2
Montant à verser :8000
Compte [code=2, solde=8000.0]
Montant à Retirer :2000
Compte [code=2, solde=6000.0]
Montant à verser :
```

Exemple 4 : AspectJ et constructeurs

- Avant et après instanciation de la classe Compte

```
public aspect IntializationAspect {  
    pointcut instatiation():initialization(*.Compte.new(..));  
    before() : instatiation() {  
        System.out.println("Avant Instantiation");  
    }  
    after():instatiation(){  
        System.out.println("Après instanciation");  
    }  
}
```

```
package test;  
import metier.Compte;  
public class Test {  
    public static void main(String[] args) {  
        for(int i=0;i<8;i++){  
            Compte cp=new Compte();  
            System.out.println(cp.toString());  
        }  
    }  
}
```

Exemple 4 :AspectJ et constructeurs

- Contrôler le nombre d'instances :

```
import java.util.ArrayList; import java.util.List;
import metier.Compte;
public aspect IntializationAspect {
    private List<Compte> comptes=new ArrayList<Compte>();
    pointcut instatiation():call(*.Compte.new(..));
    Compte around() : instatiation() {
        System.out.println("*****");
        System.out.println("Avant instantiation");
        if(comptes.size()<3){
            Compte cp=(Compte)proceed();
            System.out.println("Après instantiation");
            cp.setCode(comptes.size()+1);
            comptes.add(cp);
            System.out.println("initialisation de l'instance");
            System.out.println("*****");
            return cp;
        }
        else{
            throw new RuntimeException("Nombre d'instances dépassées");
        }
    }
}
```

Avant instantiation

Après instantiation

initialisation de l'instance

Compte [code=1, solde=0.0]

Avant instantiation

Après instantiation

initialisation de l'instance

Compte [code=2, solde=0.0]

Avant instantiation

Après instantiation

initialisation de l'instance

Compte [code=3, solde=0.0]

Avant instantiation

Exception in thread "main"

java.lang.RuntimeException:

Nombre d'instances dépassées

at

test.Test.init\$_aroundBody1\$advice(Test.java:23)

at test.Test.main(Test.java:5)

AspectJ et les exceptions

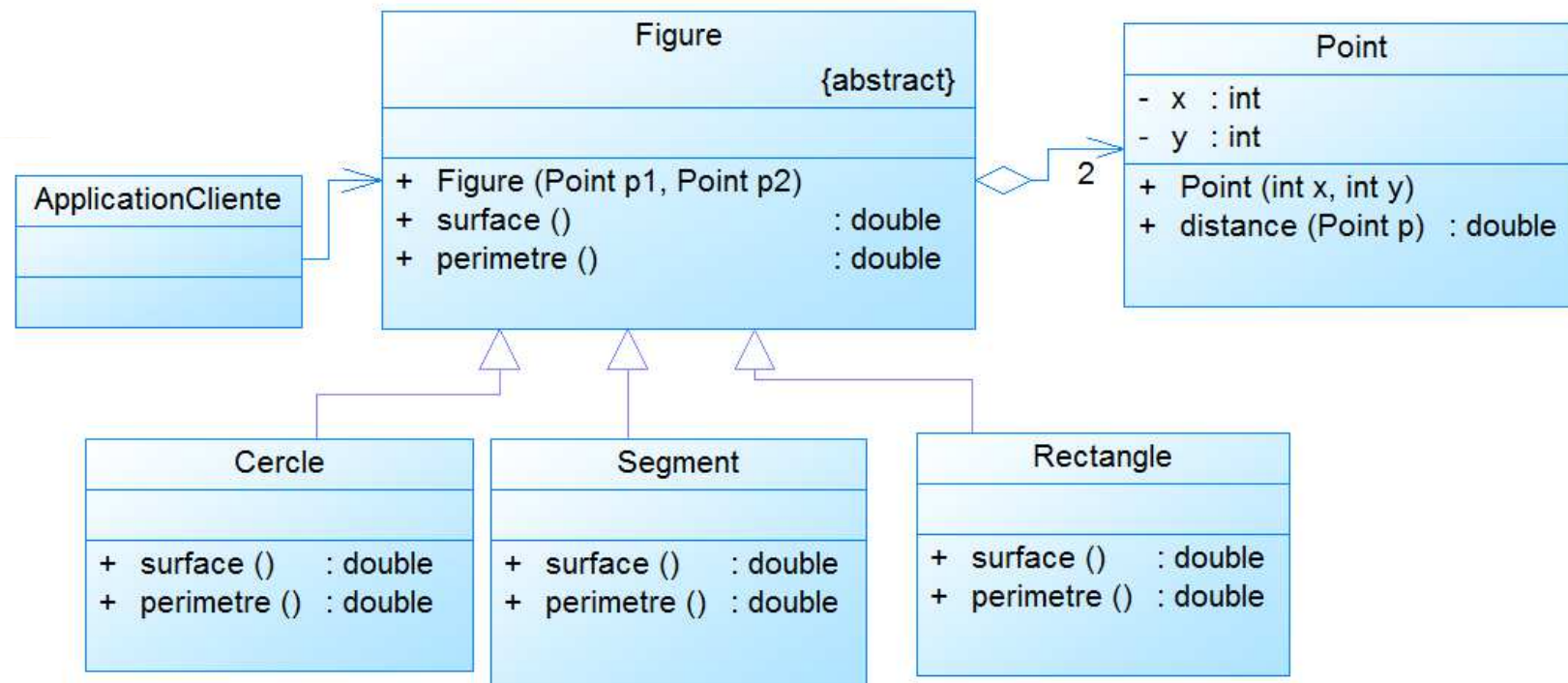
Après chaque exception générée, on affiche la méthode qui a généré l'exception et les informations sur l'exception.

```
public aspect ExceptionHandlerAspect {  
    pointcut anyMethod():call(* *.*.*(..));  
    after()throwing(Exception e): anyMethod(){  
        System.out.println(thisJoinPoint.getSignature());  
        System.out.println(e.getClass());  
        System.out.println(e.getMessage());  
    }  
}
```

```
package test;  
  
public class Test {  
  
    public void traitement() throws Exception{  
        System.out.println("Traitement...");  
        throw new Exception("Problème d'initialisation");  
    }  
    public static void main(String[] args) throws Exception {  
        Test t=new Test();  
        t.traitement();  
    }  
}
```

Exemple de modèle OO

- Une figure se compose de deux points
- La figure peut être soit un cercle, un segment ou un rectangle
- On souhaite calculer la surface et le périmètre de chaque figure



AspectJ et les exceptions

Translation des exceptions :

```
public aspect ExceptionHandlerAspect {  
    pointcut anyMethod():call(* *.*.*(..));  
    Object around(): anyMethod() {  
        try {  
            return proceed();  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
            throw new RuntimeException(e.getMessage());  
        }  
    }  
}
```


Classe Point

```
package figures;

public class Point {
    private int x;    private int y;
    public Point() { }
    public Point(int x, int y) { this.x = x; this.y = y; }
    public int getX() { return x; }
    public void setX(int x) { this.x = x; }
    public int getY() { return y; }
    public void setY(int y) { this.y = y; }
    public double distance(Point p){
        int a=p.x-this.x;
        int b=p.y-this.y;
        return Math.sqrt(a*a+b*b);
    }
    @Override
    public String toString() {
        return "Point [x=" + x + ", y=" + y + "]";
    }
}
```

Classe Figure

```
package figures;

public abstract class Figure {
    protected Point p1; protected Point p2;
    public Figure(Point p1, Point p2) { this.p1 = p1; this.p2 = p2; }
    public abstract double surface();
    public abstract double perimetre();
    public Point getP1() { return p1; }
    public void setP1(Point p1) { this.p1 = p1; }
    public Point getP2() { return p2; }
    public void setP2(Point p2) { this.p2 = p2; }
    @Override
    public String toString() {
        return "[p1=" + p1 + ", p2=" + p2 + "]";
    }
}
```

Classe Cercle

```
package figures;

public class Cercle extends Figure {
    public Cercle(Point p1, Point p2) { super(p1, p2); }

    @Override
    public double surface() {
        double rayon=p1.distance(p2);
        return Math.PI*rayon*rayon;
    }

    @Override
    public double perimetre() {
        double rayon=p1.distance(p2);
        return 2*Math.PI*rayon;
    }

    @Override
    public String toString() {
        return "Cercle [" + super.toString() + "];"
    }
}
```

Classe Rectangle

```
package figures;

public class Rectangle extends Figure {
    public Rectangle(Point p1, Point p2) { super(p1, p2); }
    @Override
    public double surface() {
        int L=Math.abs(p1.getX()-p2.getX());
        int H=Math.abs(p1.getY()-p2.getY()); return L*H;
    }
    @Override
    public double perimetre() {
        int L=Math.abs(p1.getX()-p2.getX());
        int H=Math.abs(p1.getY()-p2.getY()); return 2*(L+H);
    }
    @Override
    public String toString() {
        return "Rectangle [" + super.toString() + "]";
    }
}
```

Classe Segment

```
package figures;

public class Segment extends Figure {
    public Segment(Point p1, Point p2) { super(p1, p2); }
    @Override
    public double surface() { return 0; }
    @Override
    public double perimetre() { return p1.distance(p2); }
    @Override
    public String toString() {
        return "Segment [" + super.toString() + "]";
    }
}
```

Classe Application

```
package figures;

public class Application {

    public static void main(String[] args) {
        Figure f1=new Cercle(new Point(50, 60), new Point(80,80));
        Figure f2=new Rectangle(new Point(50, 60), new Point(80,80));
        Figure f3=new Segment(new Point(50, 60), new Point(80,80));
        System.out.println("-----");
        System.out.println(f1.toString());
        System.out.println("Surface="+f1.surface());
        System.out.println("P rim tre="+f1.perimetre());
        System.out.println("-----");
        System.out.println(f2.toString());
        System.out.println("Surface="+f2.surface());
        System.out.println("P rim tre="+f2.perimetre());
        System.out.println("-----");
        System.out.println(f3.toString());
        System.out.println("Surface="+f3.surface());
        System.out.println("P rim tre="+f3.perimetre());
    }
}
```

Ex cution

```
-----
Cercle [[p1=Point [x=50, y=60], p2=Point [x=80, y=80]]]
Surface=4084.070449666731
P rim tre=226.54346798277953
-----
Rectangle [[p1=Point [x=50, y=60], p2=Point [x=80, y=80]]]
Surface=900.0
P rim tre=120.0
-----
Segment [[p1=Point [x=50, y=60], p2=Point [x=80, y=80]]]
Surface=0.0
P rim tre=36.05551275463989
```

Effectuer la coupe avec AspectJ

- Dans AspectJ, les coupes correspondent à plusieurs points de jonctions dans le flot d'un programme.

- Par exemple, la coupe :

```
pointcut log() : call (double figures.Figure.surface());
```

- Capture chaque point de jonction correspondant à un appel à la méthode `surface` d'une figure.

- En utilisant le joker `*`, on peut m'écrire autrement :

```
pointcut log() : call (* *.Figure.surface(..));
```

- Une coupe peut être construite à partir d'autre coupes en utilisant les opérateurs `and`, `or` et `not` (respectivement `&&`, `||` et `!`).

- Par exemple, la coupe :

```
call(void figures.Point.setX(int)) ||  
call(void figures.Point.setY(int));
```

- Désigne les points de jonction correspondant à un appel à la méthode `Point.setX()` ou un appel à la méthode `Point.setY()`.

Effectuer la coupe avec AspectJ

- Les coupes peuvent identifier des points de jonction de différentes classes, en d'autres termes, elles peuvent être transverses aux classes. Par exemple, la coupe :

```
call(double figures.Figure.surface()) ||
```

```
call(void figures.Point.setX(int)) ||
```

```
call(void figures.Point.setY(int)) ||
```

- capture chaque point de jonction qui est un appel à une des trois méthodes (la première méthode est une méthode d'interface).
- On peut simplifier en utilisant le joker *.

```
call(* *.*.surface(..)) ||
```

```
call(* *.Point.set*(..)) ||
```

Première forme d'un aspect : LoggerAspect

```
package aspects;
import java.util.logging.Logger;
public aspect LoggerAspect {
    private long t1;
    private long t2;
    private Logger logger=Logger.getLogger(LoggerAspect.class.getName());
    pointcut log() : call (* *.Figure.surface());

    before():log(){
        logger.info("Before");
        t1=System.currentTimeMillis();
    }
    after():log(){
        t2=System.currentTimeMillis();
        logger.info("After :");
        logger.info("Durée d'exécution :"+(t2-t1));
    }
}

System.out.println("Surface="+f1.surface());
System.out.println("Périmètre="+f1.perimetre());
System.out.println("-----");
System.out.println(f2.toString());
System.out.println("Surface="+f2.surface());
System.out.println("Périmètre="+f2.perimetre());
System.out.println("-----");
System.out.println(f3.toString());
System.out.println("Surface="+f3.surface());
System.out.println("Périmètre="+f3.perimetre());
```

Deuxième forme d'un aspect :: SecurityAspect

```
package aspects;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
@Aspect
public class SecutityAspect {
    @Before("call(* *.Figure.*(..)) ")
    public void verification(){
        System.out.println("***** SECURITY*****");
        System.out.println("Vérification de l'identité");
        System.out.println("***** SECURITY*****");
    }
}

Figure f1=new Cercle(new Point(50, 60), new Point(80,80));
Figure f2=new Rectangle(new Point(50, 60), new Point(80,80));
Figure f3=new Segment(new Point(50, 60), new Point(80,80));
System.out.println("-----");
System.out.println(f1.toString());
System.out.println("Surface="+f1.surface());
System.out.println("Périmètre="+f1.perimetre());
System.out.println("-----");
System.out.println(f2.toString());
System.out.println("Surface="+f2.surface());
System.out.println("Périmètre="+f2.perimetre());
System.out.println("-----");
System.out.println(f3.toString());
System.out.println("Surface="+f3.surface());
System.out.println("Périmètre="+f3.perimetre());
```