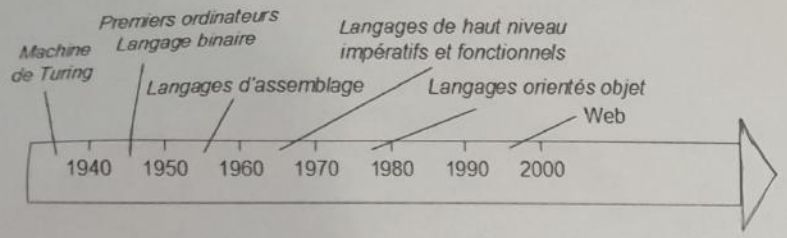
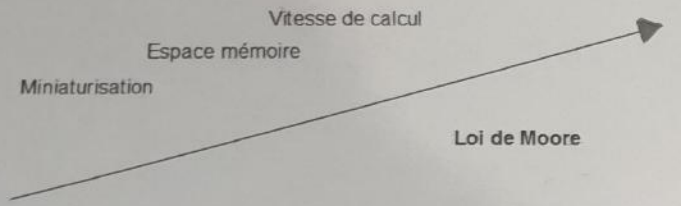
Un peu d'histoire





Evolutions technologiques

1946 ENIAC : 100 KHz,
mémoire = cablage puis 2 Ko,
30 Tonnes

1975 IBM 5100 : 2 MHz,
mémoire = 64 Ko,
22 Kg

1984 Apple Macintosh : 512 MHz,
mémoire = 512 Ko,
8 Kg

2017 PC portable : 2,5 GHz,
mémoire = 8 Go,
1 Kg

smartphone : 2,3 GHz,
mémoire = 4 Go,
150 g

Evolutions logicielles

Contrairement aux autres outils, l'ordinateur est programmable

Informatique : science qui étudie les programmes qu'un ordinateur est capable d'exécuter.

- *Au départ :* langage binaire *(= langage de la machine)*
- *Besoin d'abstraction :* Langage d'assemblage
  - *Nécessitent des assembleurs*
  - *Restent plus proches des possibilités techniques de la machine que des capacités de modélisation du développeur*
- *Encore plus d'abstraction :* Langage de haut niveau =>
  - *Bibliothèques de fonctions*
  - *Applications complexes (compilateurs, SE, appli dédiées...)*
  - *IHM, réseaux, multi-utilisateurs, embarquées,...*

Evolutions méthodologiques

Q : Pourquoi de la méthodologie ?
R : Pour aider le développeur à maitriser les éléments logiciels qu'il utilise, conçoit et fait interagir.

⇒ Modularisation :

1. Découpage des grosses structures en structures plus fines
2. Regroupement au sein d'une même structure, des éléments sémantiquement liés

L'élaboration de bibliothèques et la COO participent à cette tendance.

La COA (conception orientée aspect) aussi.
Elle n'est pas spécifique à la POO. Mais elle y est particulièrement développée, notamment autour de Java, avec l'outil AspectJ sur lequel nous nous focaliserons.

POO et modularisation

En quoi la POO contribue-t-elle à la modularisation ?

Concept de base (dynamique) : l'objet
Concept de base (statique) : la classe

Classe : module encapsulant une SdD et des méthodes agissant dessus (autrement dit, la description de la représentation interne d'un objet et de son comportement

La POO produit donc naturellement des applications vérifiant les 2 propriétés caractérisant une bonne modularisation (à condition de définir les bonnes classes à la conception).

POO : théorie vs pratique

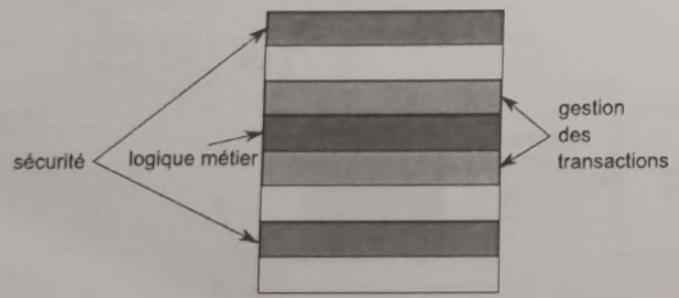
En pratique des éléments étrangers à la nature même de la classe apparaissent souvent dans sa définition.

```
public class UneClasseMetier {
    ... attributs métiers
    ... flux de log
    ... verrou de contrôle pour la concurrence

    public void uneOperation(<paramètres>) {
        ... vérifier que l opération est autorisée
        ... prendre le verrou
        ... démarrer une transaction
        ... logger le début de l opération
        ... réaliser l opération proprement dite
        ... logger l achèvement de l opération
        ... terminer la transaction (commit ou rollback)
        ... relacher le verrou
    }

    ... d'autres méthodes construites sur le même schéma
}
```

Ce schéma de classe présente un mélange de préoccupations métiers (core concerns) et de préoccupations transversales (crosscutting concerns).

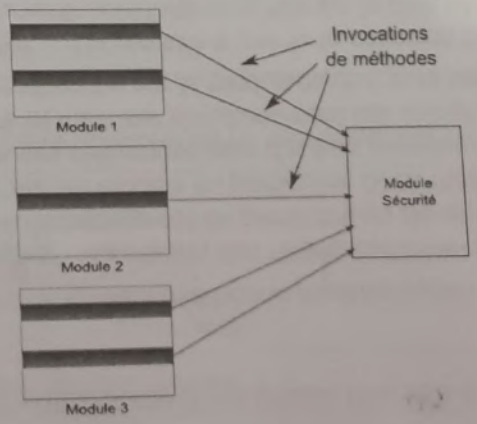
POO : défaut 1

L'enchevêtrement de code (code tangling)

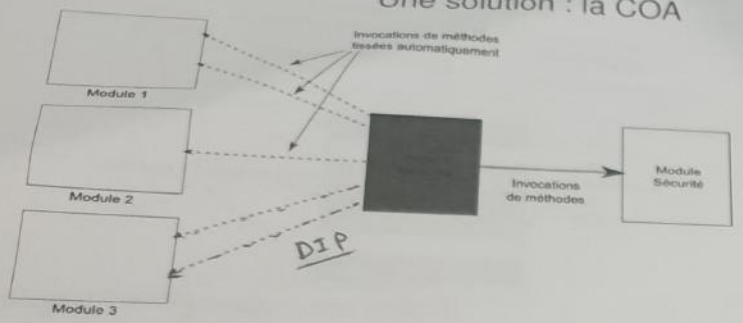


POO : défaut 2

L'éparpillement de code (code scattering)



Une solution : la COA



- Supression de l'enchevêtrement de code dans les modules 1, 2, 3.
- Suppression de l'éparpillement de code grâce à l'aspect sécurité.

. des modules il y que du code métier

COA et Modularisation

module Métier ne dépend pas du module technique

- L'aspect sécurité diminue le couplage, puisque les modules métiers ne contiennent plus aucune référence vers le module sécurité.
- L'aspect sécurité est un nouveau module à forte cohésion.

Le principe, ici mis en œuvre, est appelé principe d'inversion de dépendance (DIP, Dependency Inversion Principle).
On avait une référence des modules métiers vers le module sécurité.
La dépendance a été inversée pour aller de l'aspect sécurité vers les modules métiers.

Modulariser efficacement

Modularisation = découpage d'une grosse structure en structures plus fines.

Les 2 critères caractérisant une bonne modularisation :

1. Faible couplage : Limitation des références inter-modules. L'intérêt du découpage est de permettre au développeur de se concentrer sur un module en particulier. Si ce module comporte de nombreuses références vers d'autres modules, il va devoir comprendre et maitriser simultanément ces autres modules, et l'avantage recherché par la modularisation sera perdu.
2. Forte cohesion : Un module a une forte cohésion si les éléments qu'il fait intervenir sont fortement liés sur le plan sémantique. Ce point est en rapport direct avec le précédent, puisque de tels éléments, s'ils étaient répartis dans plusieurs modules, généreraient un couplage fort entre ces modules.

POA

Principalement, la POA consiste à

- coder les préoccupations transversales et
- préciser où les injecter.

Comment code t-on en POA ?

- Les préoccupations transversales sont essentiellement codées à l'aide du langage support (Java pour AspectJ).
- Les injections sont définies, selon les outils, via un langage dédié ou via le langage support. AspectJ propose les 2 approches :
  - le langage dédié, AspectJ, permet d'exploiter toute la puissance de l'outil ;
  - le langage @AspectJ, en fait un jeu d'annotations Java, ne donne accés qu'à certaines fonctionnalités (les plus usuelles) mais peut être compilé avec ajc.

Vocabulaire

- Point de jonction (join point) : il s'agit d'un emplacement dans le code exécuté où un aspect peut intervenir. Tous les emplacements ne sont pas des points de jonction, mais AspectJ permet de sélectionner les plus importants (appels ou exécutions de méthodes, lectures ou écritures d'attributs,...).
- Coupe (pointcut) : il s'agit d'une expression permettant de sélectionner certains points de jonction.
- Greffon (advice) : il s'agit du code décrivant le comportement additionnel que l'on souhaite injecter.
- Aspect : c'est l'unité modulaire regroupant les coupes et les greffons relatifs à une préoccupation particulière.

Premiers exemples

On se limite à des points de jonction liés à des appels de méthodes. Un tel point de jonction délimite la portion de code exécuté qui

- démarre avec l'appel de la méthode (donc, après que les arguments aient été évalués),
- comprend toute l'exécution de la méthode et
- se termine juste après le retour de la méthode (que celui-ci se fasse normalement ou par le biais d'une levée d'exception).

Attention : chaque appel d'une même méthode sur un même objet donne lieu à des points de jonction différents (même si ces appels proviennent de la même ligne de code source).

Coupes (i)

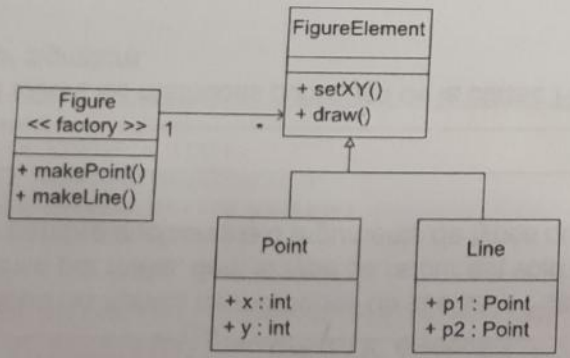


FIGURE – (source : tutoriel eclipse - AspectJ)

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

Cette coupe permet de sélectionner tous les points de jonction correspondant à des appels à la méthode setX sur des instances de Point. Le lexème call est un mot clé du langage AspectJ.

Coupes (ii)

Les expressions de coupes peuvent être combinées avec les opérateurs booléens traditionnels : ||, &&, !

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

Une coupe peut être nommée via un deuxième mot clé, pointcut :

```
pointcut move():
    call(void FigureElement.setXY(int,int)) ||
    call(void Point.setX(int))              ||
    call(void Point.setY(int))              ||
    call(void Line.setP1(Point))            ||
    call(void Line.setP2(Point))
```

Ce nom peut être réutilisé ensuite :

```
move() && !call(void FigureElement.setXY(int,int))
```

Jokers

Les définitions de coupes peuvent contenir des jokers.

```
call(void Figure.make*(..))
```

sélectionne tous les appels de méthodes de la classe Figure, dont le nom commence par make, dont le type de retour est void et qui prennent un nombre quelconque d'arguments de types quelconques.

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

sélectionne toutes les méthodes publiques de la classe Figure quelle que soit leur signature.

Advices (i)

Il existe trois catégories d'advices : before, after et around

```
before() : move() {
    System.out.println(
        "On entre dans une méthode qui fait bouger."
    );
}
```

Un advice ressemble à une méthode, avec du code Java entre accolades. La principale différence est la mention d'une coupe (anonyme ou nommée) juste avant ce bloc. Cette coupe indique les points de jonction où l'advice doit être injecté.

L'advice étant de catégorie before, le code qu'il définit sera exécuté juste avant les points de jonction concernés.

Advices (ii)

```
after() : move() {
    System.out.println(
        "On sort d'une méthode qui fait bouger."
    );
}
```

Cet advice sera exécuté juste après les points de jonction définis par la coupe move().

Les advices de la catégorie around permettent d'exécuter du code avant et après les points de jonction.

Contexte (i)

En plus de sélectionner certains points de jonction, les coupes peuvent donner accès à certains éléments du contexte associé aux points de jonction. Ces éléments peuvent ensuite être utilisés par les advices.

```
pointcut setXY(FigureElement fe, int x, int y) :
    call(void FigureElement.setXY(int, int))
    && target(fe)
    && args(x, y);
```

Les coupes target et args sont prédéfinies. Elles permettent de récupérer la cible et les paramètres réels associés à un point de jonction. La coupe est alors paramétrée par ces valeurs.

Contexte (ii)

Un advice peut ensuite être paramétré en s'appuyant sur les paramètres de sa coupe. Il pourra utiliser ces paramètres dans son bloc de code, de la même manière qu'une méthode Java utilise ses paramètres.

```
after(FigureElement fe, int x, int y) : setXY(fe, x, y) {
    System.out.println(
        fe + " subit une translation de (" + x + ", " + y + ")."
    );
}
```


Aspects

Le mot clé aspect permet de regrouper coupes et advices au sein de modules similaires à des classes :

```
aspect Logging {
    private OutputStream logStream = System.err;

    pointcut move():
        call(void FigureElement.setXY(int,int)) ||
        call(void Point.setX(int))              ||
        call(void Point.setY(int))              ||
        call(void Line.setP1(Point))            ||
        call(void Line.setP2(Point));

    before() : move() {
        logStream.println("Ca va bouger");
    }
}
```

Les aspects, comme les classes, peuvent contenir des attributs et des méthodes.
Une différence notoire : on ne peut pas instancier un aspect à l'aide de la commande `new`. Nous en reparlerons.

Exemple complet

```
package chap01.messaging;

public class MessageCommunicator {
  public void deliver(String message) {
    System.out.println(message);
  }

  public void deliver(String person, String message) {
    System.out.println(person + " : " + message);
  }
}
```



```
package chap01.main;

import chap01.messaging.MessageCommunicator;

public class Main {
  public static void main(String[] args) {
    MessageCommunicator messageCommunicator =
            new MessageCommunicator();
    messageCommunicator.deliver("Le Master GIL, c'est sympa !");
    messageCommunicator.deliver("Bruno", "Surtout la POA !");
  }
}
```

```
> ajc -d bin
    src\chap01\messaging\MessageCommunicator.java
    src\chap01\main\Main.java
%% Je me place dans le répertoire bin
> java chap01.main.Main
Le Master GIL, c'est sympa !
Bruno : Surtout la POA !
```

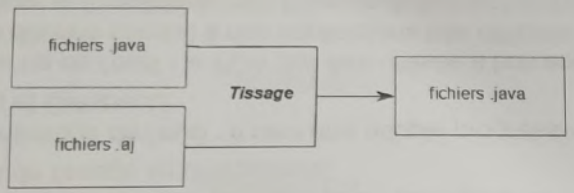
Remarque : ajc, le compilateur d'AspectJ, contient un compilateur Java.

Tissage

Tissage : Injection du code des advices dans le code métier.

Cette opération est réalisée par un tisseur (weaver).

A l'origine, le tissage ne concernait que le code source :



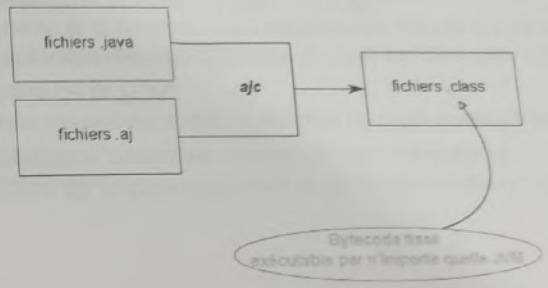
Mais cette approche est limitée au code source et, en cas d'erreur, il était difficile de remonter aux fichiers initiaux.
↪ Technique abandonnée.

La perte de son principal avantage (code tissé visible) est aujourd'hui compensée par les fonctionnalités des outils d'édition.

Tissage de code source

AspectJ offre 3 modes de tissage.

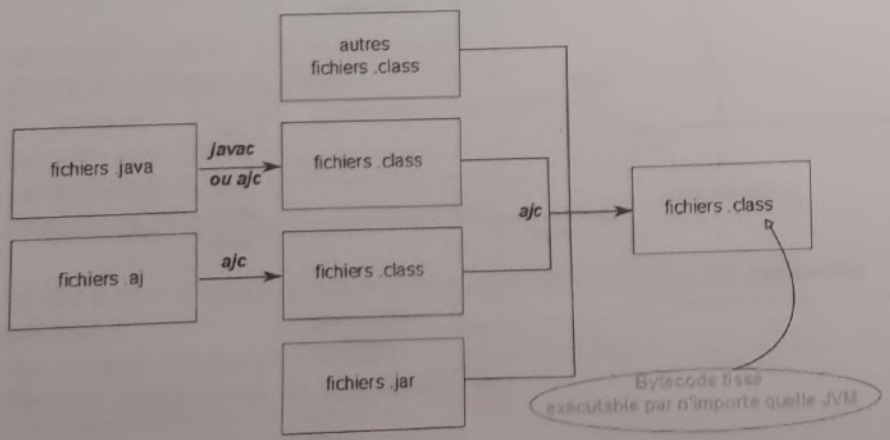
Celui que nous avons utilisé sur notre exemple est le tissage de code source (source-code weaving) :



Remarque : l'ensemble des fichiers sources et aspects doit être fourni en une seule commande.

Tissage de code binaire

Deuxième mode de tissage :



Sur notre exemple, on peut faire du tissage de code binaire (byte-code weaving) de la façon suivante :

```
> javac -d classes
    src\chap01\messaging\MessageCommunicator.java
    src\chap01\main\Main.java
> ajc -d aspects -classpath classes;%CLASSPATH% -source 5
    src\chap01\security\SecurityAspect.aj
    src\chap01\security\*.java
> ajc -d woven -inpath classes;aspects -aspectpath aspects
> java -classpath woven;%CLASSPATH% chap01.main.Main
```

Programmation Orientée Aspect

Chapitre 2
Points de jonction et coupes

B. Patrou

Master 2 GIL
Université de Rouen

Thèmes abordés :

1. Points de jonction
2. Signature d'un élément de code
3. Coupes
   - Kinded
   - Non kinded

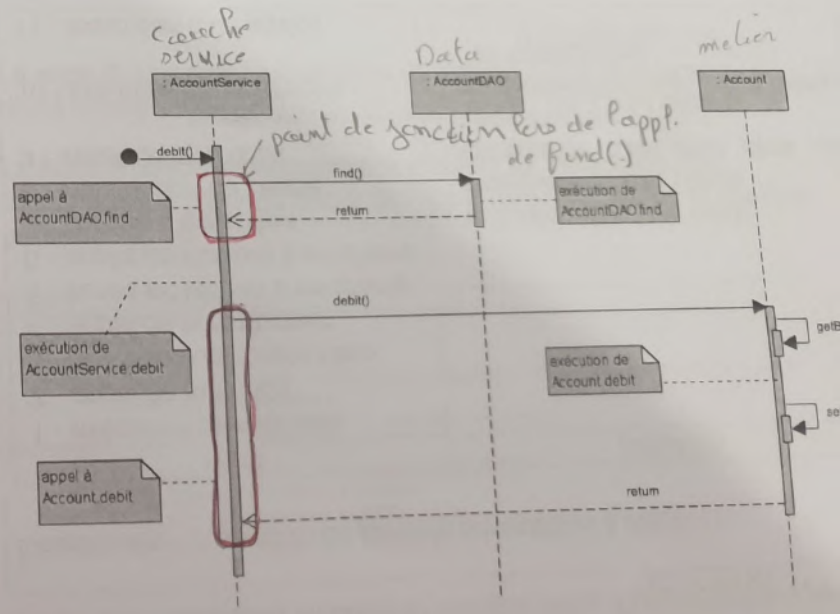
Point de jonction

Définition (trop) simple : un point particulier repéré dans le code.

Un point ? zone
Plutôt une portion du code.

Le code java ? Non
Le byte-code ? Non
Plutôt le flot d'instructions à l'exécution.

Le flot d'exécution n'est pas visible. Comment expliciter les points de jonction ?
Sur le code source (cf Eclipse).
Sur des diagrammes de séquence UML.



Contexte

Un point de jonction dispose d'un contexte constitué, selon les cas,

- des objets source et cible
  - d'un appel ou
  - d'une exécution
    - de méthode ou
    - de constructeur
- des arguments d'un appel
  - de méthode ou
  - de constructeur
- d'attributs
- de types
- ...

Les différentes sortes de points de jonction

Classification des points de jonction accessibles à AspectJ :

sorte	précision
1 - exécution de méthode	
2 - appel de méthode	
3 - exécution de constructeur	
4 - appel de constructeur	
5 - accés en lecture à un champ	
6 - accés en écriture à un champ	
7 - gestion d'exception	- exécution d'un bloc catch.
8 - initialisation de classe	- chargement d'une classe
9 - initialisation d'objet	- initialisation d'un objet dans un constructeur
10 - pré-initialisation d'objet	- pré-initialisation d'un objet dans un constructeur
11 - exécution d'un advice	

Appel et exécution de méthodes

Les sortes 1 et 2 désignent les points de jonctions les plus fréquemment utilisés.

Exemple :

```
public class Account {
  ...
  public void debit(double amount)
      throws InsufficientBalanceException {
    if (balance < amount) {
      throw new InsufficientBalanceException(
              "not enough money !"
      );
    } else {
      balance -= amount;
    }
  }
```


Ce code pourrait contribuer à produire le flot d'exécution suivant :

```
1  appel à myAccount.debit(30.0)
2  test 100.0 < 30.0
3  écraser 100.0 par 100.0 - 30.0
4      dans l'emplacement mémoire associé au champ balance
5  remonter à la méthode appelante
6      dans la pile des appels de méthodes
7  ...
8  appel à myAccount.debit(80.0)
9  test 70.0 < 80.0
10 créer une instance de InsufficientBalanceException
11 donner la main au gestionnaire d'exceptions
```

- Blocs 2-4 et 9-10 : points de jonction associés à des exécutions de la méthode `debit`.
- Blocs 1-6 et 8-11 : points de jonction associés à des appels de la méthode `debit`.

Appel

Les sortes 3 et

Détail

Appel et exécution de constructeurs

Les sortes 3 et 4 sont similaires aux deux premières.

Détail spécifique : un point de jonction relatif à l'exécution d'un constructeur ne contient pas les appels à this *ou à* super*, qu'ils soient explicites ou implicites.*

Exemple :

```
1  public class SavingsAccount extends Account {
2      ...
3
4      public SavingsAccount (int accountNumber, boolean overdraftAllowed) {
5          super(accountNumber);
6          this.overdraftAllowed = overdraftAllowed;
7      }
8      public SavingsAccount (int accountNumber) {
9          this(accountNumber, false);
10         this.minimumBalance = MIN;
11     }
12 }
```

Les lignes 6 et 10 correspondent chacune à un point de jonction relatif à l'exécution d'un des constructeurs de la classe `SavingsAccount`.

Gestion d'exceptions

La sorte 7 concerne les blocs catch exécutés lorsqu'une exception est attrapée.

Exemple d'exploitation de tels points de jonction :
Un aspect peut enregistrer les levées d'exceptions ainsi attrapées et les enregistrer dans un fichier de logs.

Lecture et écriture de champs

Les sortes 5 et 6 concernent les accés aux champs de classes (champs statiques) ou d'objets.

Exemple d'exploitation de tels points de jonction :
Un aspect peut repérer les accés en écriture à certains champs. Vérifier s'ils modifient la valeur de ce champ et, le cas échéant, positionner un drapeau.
Un processus automatique consultera ce drapeau pour éventuellement mettre à jour une BdD.

Initialisations

La sorte 8 concerne l'initialisation de classes, i.e. le chargement d'une classe et l'execution des blocs statiques qui y sont définis.

La sorte 9 concerne l'initialisation d'objets, i.e. l'exécution d'un constructeur à partir du retour de l'appel (direct ou indirect et implicite ou non) à super.

```
1  public class SavingsAccount extends Account {
2      ...
3
4      public SavingsAccount (int accountNumber, boolean overdraftAllowed) {
5          super(accountNumber);
6          this.overdraftAllowed = overdraftAllowed;
7      }
8      public SavingsAccount (int accountNumber) {
9          this(accountNumber, false);
10         this.minimumBalance = MIN;
11     }
12 }
```

La ligne 6 correspond au point de jonction relatif à l'initialisation d'une instance de `SavingsAccount` lors d'un appel au 1er constructeur.
Les lignes 6 et 10, lors d'un appel au 2nd constructeur.

Pré-initialisation et exécution d'advices

La sorte 10 recouvre le code exécuté par un constructeur avant l'appel à `super` (essentiellement, l'évaluation des paramètres de l'appel).

La sorte 11 concerne l'exécution des advices (globalement - sans distinction).

Signatures

Les points de jonction exploités par des aspects sont sélectionnés à partir de coupes.
Les coupes sont construites à partir de la signature des éléments de code Java.

Les éléments significatifs d'un programme Java possèdent tous une signature.

On peut spécifier plusieurs signatures en une seule expression grâce aux opérateurs booléens et à des jokers

- `*` : qui remplace un nombre quelconque de caractères autre que le point. Ce symbole est généralement utilisé à la place de tout ou partie du nom d'un paquetage, d'une classe ou d'une méthode.
- `..` : qui remplace un nombre quelconque de caractères. Cette notation permet souvent d'abréger la hiérarchie de paquetages conduisant à un type ou la liste des arguments d'une méthode.
- `+` : désigne n'importe quel sous-type d'un type donné.

Signatures de types

Syntaxe :

[<annotations>] <nom de type> ['<'<paramètres de type>'>']

Signature	Description	Exemples
Account	Le type exact (ni un super-type, ni un sous-type)	
*Account	Tout type dont le nom se termine par Account	UserAccount
java.*.Date	Le type Date dans n'importe quel sous-paquetage immédiat du paquetage java	java.util.Date
java..*	Tout type du paquetage java ou d'un sous-paquetage direct ou indirect du paquetage java	java.awt.event
javax..*Button+	Tout type du paquetage javax et de ses sous-paquetages directs et indirects, dont le nom se termine par Button, ou tout sous-type d'un tel type	javax.swing.AbstractButton ou javax.swing.JMenu (qui est un descendant de AbstractButton)
@Secured Account	Le type Account marqué par l'annotation Secured	
@Business* Customer+	Le type Customer et ses sous-types quand ils sont marqués par une annotation dont le nom commence par Business	@BusinessEntity class PrivilegedCustomer extends Customer
*<Account>	Tout type paramétré par le type Account	Collection<Account>
Collection <? extends Account>	Le type Collection paramétré par Account ou par un descendant de Account	Set<DefaultAccount>
!Collection	Tout type autre que Collection	Set
java.util.RandomAccess+ && java.util.List+ @Secured || Sensitive) *	Tout type qui implémente directement ou indirectement les deux interfaces	java.util.ArrayList
	Tout type marqué par au moins l'une des deux annotations	

TABLE – Signatures de types (classes ou interfaces)

Signatures de méthodes

Syntaxe :

[<annotations>] [<modificateurs>] <type de retour> [<nom de type>'.']<nom de méthode> '('<paramètres>')' [<clauses de levées d'exceptions>]

Signature	Description	Exemples
public void Account.set*(*)	Toute méthode publique de la classe Account dont le nom commence par set, retournant void et prenant un seul paramètre	
public void Account.*()	Toute méthode publique sans paramètre de la classe Account qui retourne void	
* Account.*(..)	Toute méthode de la classe Account	
* *.*(..) ou * *(..)	Toute méthode	
!public * Account.*(..)	Toute méthode non publique de Account	
* *(..) throws IOException	Toute méthode qui déclare lever des IOExceptions	
* java.io.Reader.read(char[], ..)	Toute méthode read de la classe java.io.Reader dont le premier paramètre est de type char[]	
* javax..*.add*Listener(EventListener+)	Toute méthode dont le nom commence par add et termine par Listener, dans le paquetage javax ou l'un de ses sous-paquetage direct ou indirect et prenant un paramètre de type EventListener ou l'un de ses sous-types.	TableModel.addTable TableModelLis)
@Secured * *(..)	Toute méthode marquée par l'annotation Secured	
(@Sensitive *) *(..)	Toute méthode dont le type de retour est marqué par l'annotation Sensitive	
* (@BusinessEntity *).*(..)	Toute méthode définie dans un type marqué par l'annotation BusinessEntity	
* *(@RequestParam *))	Toute méthode dont l'unique paramètre est marqué par l'annotation RequestParam	void show(@Re
* *(@Sensitive *)	Toute méthode dont le type de l(unique paramètre est marqué par l'annotation Sensitive	

TABLE – Signatures de méthodes

Signatures de constructeurs

Les signatures de constructeurs sont similaires à celles des méthodes.

Trois points les distinguent :

- Le terme `new` est utilisé à la place du nom de la méthode.
- Il est interdit de mentionner le type de retour (les constructeurs n'en ont pas).
- Le modificateur `static` est interdit (on ne peut l'appliquer aux constructeurs).

Signatures de champs

Syntaxe :

[<annotations>] [<modificateurs>] <type> [<nom de type>.]<nom de champ>

Signature	Description
private double Account.balance	Le champ privé balance de type double de la classe Account
* Account.*	Tous les champs de la classe Account (y compris statiques)
* Account+.*	Tous les champs de la classe Account et de ses sous-classes
@Sensitive * *.*	Tout champs marqué par l'annotation Sensitive

TABLE – Signatures de champs

Coupes

- Une coupe est un élément de programme AspectJ permettant de sélectionner un ensemble de points de jonction.
- Une coupe peut être anonyme ou nommée.

Syntaxe d'une définition de coupe anonyme :

<type de coupe>'('<paramètres de la coupe>')'

Syntaxe d'une définition de coupe nommée :

<modificateur de visibilité> `pointcut` <nom de la coupe>'('<paramètres de la coupe>')' : <coupe anonyme>

Types de coupe

Les coupes sont obtenues à l'aide des opérateurs booléens (!, &&, ||) appliqués à des coupes simples.

Les coupes simples se divisent en 2 grandes catégories :

1. Les coupes correspondant à une sorte de points de jonction (kinded pointcuts). Nous avons vu qu'il existait 11 sortes de points de jonction. Cette catégorie se subdivise donc en 11 sous-catégories.
2. Les coupes transverses (non-kinded pointcuts) qui sélectionnent des points de jonctions appartenant généralement à plusieurs sortes distinctes.

sorte →

Kinded pointcuts

Sorte	coupe
1 - exécution de méthode	`execution`(<signature de méthode>)
2 - appel de méthode	`call`(<signature de méthode>)
3 - exécution de constructeur	`execution`(<signature de constructeur>)
4 - appel de constructeur	`call`(<signature de constructeur>)
5 - lecture d'un champ	`get`(<signature de champ>)
6 - écriture dans un champ	`set`(<signature de champ>)
7 - exécution d'un bloc catch	`handler`(<signature de type>)
8 - initialisation de classe	`staticinitialization`(<signature de type>)
9 - initialisation d'objet	`initialization`(<signature de constructeur>)
10 - pré-initialisation d'objet	`preinitialization`(<signature de constructeur>)
11 - exécution d'advices	`adviceexecution`():

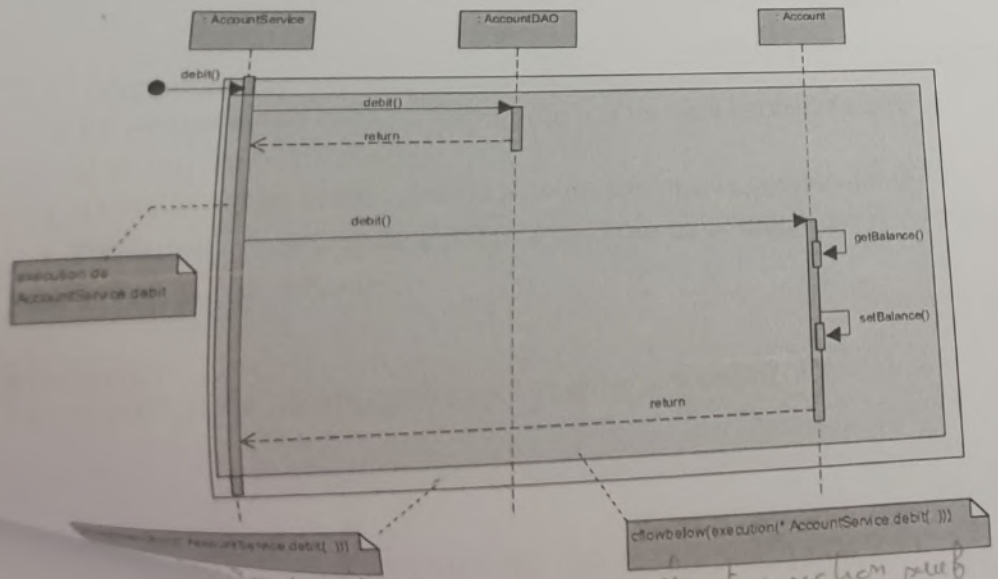
Les coupes basées sur le flot d'exécution

Syntaxe : `cflow`(<coupe>) et `cflowbelow`(<coupe>).

Un point de jonction est sélectionné par une telle coupe s'il est strictement inclus dans le flot d'exécution associé à un point de jonction repéré par la coupe fournie en paramètre.

Dans le cas d'une coupe de type `cflow`, les points de jonction repérés par la coupe fournie en paramètre sont également sélectionnés.

Exemple en UML



Autres exemples

coupe	description
`cflow(execution(* Account.debit(..))`	Tout point de jonction dans le flot de contrôle de l'exécution de n'importe quelle méthode debit du type Account (y compris l'exécution de la méthode debit).
`cflowbelow(execution(* Account.debit(..))`	idem, mais exécution de la méthode debit exclue.
`cflow(execution(@Transactional * *(..))`	Tout point de jonction dans le flot de contrôle de l'exécution de n'importe quelle méthode marquée par l'annotation Transactional.
`cflow(transacted())`	Tout point de jonction dans le flot de contrôle d'un point de jonction sélectionné par la coupe transacted

Une situation classique...

... d'utilisation d'une coupe de type [illegible]

Permet d'éviter de sélectionner des points de jonction en cascade, lors [illegible] sélectionnés par la coupe transactée).

Remarque : ces coupes sont dynamiques par nature. Elles ne peuvent être déterminées qu'à l'exécution.

Exemples

Coupe	Description
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Une situation classique...

... d'utilisation d'une coupe de type `cflowbelow` :

```
transacted() && !cflowbelow(transacted())
```

Permet d'éviter de sélectionner des points de jonction en cascade, lors d'appels récursifs sélectionnés par la coupe transacted().

Remarque : ces coupes sont dynamiques par nature. Elles ne peuvent être déterminées qu'à l'exécution.

Les coupes basées sur la localisation géographique

Trois coupes sont basées sur la référence au code source des points de jonction qu'elles sélectionnent :

- `within`(<signature de types>)
- `withincode`(<signature de méthodes>)
- `withincode`(<signature de constructeurs>)

Les points de jonction sélectionnés par de telles coupes doivent être directement situés dans le code source associé aux types, aux méthodes ou aux constructeurs repérés par la signature fournie en paramètre.

Exemples

coupe	description
`within(Account)`	Tout point de jonction dans le texte de la classe `Account` (y compris dans une classe interne).
`within(Account+)`	Tout point de jonction dans le texte de la classe `Account` ou de l'une de ses sous-classes (y compris dans une classe interne).
`within(@javax.persistence.Entity *)`	Tout point de jonction dans le texte d'un type marqué par l'annotation `Entity`.
`withincode(* Account.debit(..))`	Tout point de jonction dans le texte d'une méthode `debit` du type `Account`.

Un exemple classique :

```
traced() && !within(TraceAspect)
```

Permet d'éviter de tracer des points de jonction situés dans l'aspect de traçage lui-même.

Les coupes basées sur des éléments de contexte

Trois coupes dépendent des objets mis en jeu au niveau du point de jonction :

- `this`(<type>)
- `target`(<type>)
- `args`(<type>, ...)

Les coupes de type `this` sélectionnent les points de jonction où l'objet désigné par la variable java "this" est du type fourni en paramètre.

Les coupes de type `target` sélectionnent les points de jonction où l'objet cible est du type fourni en paramètre.

Les coupes de type `args` sélectionnent les points de jonction qui font intervenir des arguments dont le type correspond aux types fournis en paramètre.

Plus précisément, `this instanceof <type>` doit être vrai (même chose avec la cible et les paramètres associés à un point de jonction).

Remarques :

- Ces coupes étant traitées dynamiquement, les éventuels paramètres génériques auront été éliminés par javac. On ne peut donc pas fournir de type générique.
- Pour la même raison, on ne peut pas utiliser les jokers pour fournir le type attendu en paramètre de ces trois coupes.

Pour les coupes de type args, les objets considérés comme arguments varient selon le type du point de jonction repéré :

- pour les points de jonction liés à des méthodes ou des constructeurs (call, execution, initialization et preinitialization), les arguments sont ceux de l'appel de fonction associé.
- pour les points de jonction liés à des levées d'exception, l'argument est l'exception levée.
- pour les accés en écriture à des champs, l'argument est la valeur à affecter.

Exemples : `args(Account,..,int)` et `args(IOException)`

Différences entre within et this :

```
1  public class Account {
2      ...
3      public void debit (double amount)
4              throws InsufficientBalanceException {
5              ...
6      }
7      private static class Helper {
8          ...
9      }
10 }
11
12 public class SavingsAccount extends Account {
13     ...
14 }
```

- `within(Account)` sélectionne les points de jonction situés entre les lignes 2 et 9.
- `this(Account)` sélectionne les points de jonction situés entre les lignes 2 et 6 et à la ligne 13.

Un exemple classique :

```
this(Type) && !within(Type)
```

Permet de sélectionner les points de jonction uniquement situés dans des sous-classes du type `Type`.

Les coupes basées sur des annotations

L'utilisation d'annotations est très prisée par les développeurs J2EE. Il peut donc être utile de définir des coupes en fonction des annotations attachées aux éléments de code.

coupe	description
`@this`(<signature de type>)	Tout point de jonction pour lequel l'objet this est marqué par l'annotation dont le type est fourni en paramètre.
`@target`(<signature de type>)	Tout point de jonction pour lequel l'objet cible est marqué par l'annotation dont le type est fourni en paramètre.
`@args`(<signature de type>)	Tout point de jonction pour lequel un argument au moins est marqué par l'annotation dont le type est fourni en paramètre.
`@within`(<signature de type>)	Tout point de jonction situé dans le code d'un type marqué par l'annotation dont le type est fourni en paramètre.
`@withincode`(<signature de type>)	Tout point de jonction situé dans le code d'une méthode marquée par l'annotation dont le type est fourni en paramètre.
`@annotation`(<signature de type>)	Tout point de jonction dont le sujet est marqué par l'annotation dont le type est fourni en paramètre.

Accés aux attributs d'un type

Les instances de `Class` *permettent d'accéder aux :*

- *champs,*
- *méthodes et*
- *constructeurs*

de la classe associée.

Elles disposent pour cela de méthodes variées fonctionnant selon diverses modalités :

- selon que l'on souhaite accéder à un élément particulier ou à tous ;
- que l'on désire accéder à un élément déclaré dans le type même, quel que soit son niveau de _____ ;
- que l'on veuille accéder à un élément publique mais _____.

Le tableau ci-après résume ces possibilités pour les champs :

	Liste des champs ?	champs hérités ?	Champs privés ?
`getDeclaredField()`	non	non	oui
`getField()`	non	oui	non
`getDeclaredFields()`	oui	non	oui
`getFields()`	oui	oui	non

Les méthodes et les constructeurs s'obtiennent à l'identique en remplaçant le motif `Field` par le motif `Method` ou `Constructor` dans les noms de méthodes indiqués dans le tableau.

Une fonction d'introspection qui décrit les attributs et les classes internes définis par un type donné :

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Member;
import static java.lang.System.out;

enum ClassMember { CONSTRUCTOR, FIELD, METHOD, CLASS, ALL }

public class ClassSpy {
    public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            out.format("Class:%n  %s%n%n", c.getCanonicalName());

            Package p = c.getPackage();
            out.format("Package:%n  %s%n%n",
                    (p != null ? p.getName() : "--  No  Package  --"));
```

```
            for (int i = 1; i < args.length; i++) {
                switch (ClassMember.valueOf(args[i])) {
                case CONSTRUCTOR:
                    printMembers(c.getConstructors(), "Constructor");
                    break;
                case FIELD:
                    printMembers(c.getFields(), "Fields");
                    break;
                case METHOD:
                    printMembers(c.getMethods(), "Methods");
                    break;
                case CLASS:
                    printClasses(c);
                    break;
                case ALL:
                    printMembers(c.getConstructors(), "Constructors");
                    printMembers(c.getFields(), "Fields");
                    printMembers(c.getMethods(), "Methods");
                    printClasses(c);
                    break;
                default:
                    assert false;
                }
            }

        // production code should handle these exceptions more gracefully
        } catch (ClassNotFoundException x) {
            x.printStackTrace();
```

La classe `Field`

Elle fournit des outils permettant d'obtenir des informations sur les champs d'un type et de modifier la valeur de ces champs.

Commençons par inspecter le type d'un champ :

```
import java.lang.reflect.Field;
import java.util.List;

public class FieldSpy<T> {
    public boolean[][] b = {{ false, false }, { true, true } };
    public String name  = "Alice";
    public List<Integer> list;
    public T val;

    public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            Field f = c.getField(args[1]);
            System.out.format("Type: %s%n", f.getType());
            System.out.format("GenericType: %s%n", f.getGenericType());
        // production code should handle these exceptions more gracefully
        } catch (ClassNotFoundException x) {
            x.printStackTrace();
        } catch (NoSuchFieldException x) {
            x.printStackTrace();
        }
    }
}
```


Exemples d'exécution :

```
> java FieldSpy FieldSpy b
Type: class [[Z
GenericType: class [[Z
> java FieldSpy FieldSpy name
Type: class java.lang.String
GenericType: class java.lang.String
> java FieldSpy FieldSpy list
Type: interface java.util.List
GenericType: java.util.List<java.lang.Integer>
> java FieldSpy FieldSpy val
Type: class java.lang.Object
GenericType: T
```

↪ La méthode `getGenericType` consulte le fichier source pour retrouver les types génériques éventuellement utilisés.

Si ce fichier n'est pas accessible, elle se comporte comme `getType`.

Le programme suivant examine le type des champs et précise si ce sont des champs synthétiques (générés par le compilateur) ou des constantes d'un type énuméré.

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import static java.lang.System.out;

enum Spy { BLACK , WHITE }

public class FieldModifierSpy {
    volatile int share;
    int instance;
    class Inner {}

    public static void main(String... args) {
        try {
            Class<?> c = Class.forName(args[0]);
            int searchMods = 0x0;
            for (int i = 1; i < args.length; i++) {
                searchMods |= modifierFromString(args[i]);
            }
```



```
            Field[] flds = c.getDeclaredFields();
            out.format("Fields in Class '%s' containing modifiers: %s%n",
                       c.getName(),
                       Modifier.toString(searchMods));
            boolean found = false;
            for (Field f : flds) {
                int foundMods = f.getModifiers();
                // Require all of the requested modifiers to be present
                if ((foundMods & searchMods) == searchMods) {
                    out.format("%-8s [ synthetic=%-5b enum_constant=%-5b ]%n",
                               f.getName(), f.isSynthetic(),
                               f.isEnumConstant());
                    found = true;
                }
            }

            if (!found) {
                out.format("No matching fields%n");
            }

        // production code should handle this exception more gracefully
        } catch (ClassNotFoundException x) {
            x.printStackTrace();
        }
    }
```

Programmation Orientée Aspect

Chapitre 4
Fonctionnalités transversales

B. Patrou

Master 2 GIL
Université de Rouen

Thèmes abordés :

1. Les advices
   - Les 3 sortes
   - Le contexte d'un advice
   - Les variables spéciales
2. Les Déclarations inter-types
3. Les modifications de la hiérarchie des types
4. Quelques joyeusetés supplémentaires

Les 2 catégories de fonctionnalités transversales

- fonctionnalités dynamique : modification du comportement des applications
- fonctionnalités statiques: modification de la structure des applications

Fonctionnalités dynamiques

Deux éléments sont nécessaires au tissage d'une fonctionnalité dynamique :

- un advice (quoi faire) et
- une coupe (quand le faire).

Nous avons étudié les coupes au chapitre 2.
Détaillons maintenant les advices.

Syntaxe :

<type de l'advice>'('<paramètres>')' [déclaration de levée d'exceptions] ' :' <définition de coupe> '{'<corps de l'advice>'}'

Advices vs méthodes : similitudes

- Une méthode a un nom.
  Un advice, non, mais il est possible de lui en donner un par le biais de l'annotation pré-définie org.aspectj.lang.annotation.AdviceName.
- Les deux peuvent prendre des paramètres. Ceux d'une méthode sont fournis par l'instruction d'appel de la méthode. Ceux d'un advice sont fournis par la coupe (on n'appelle pas directement un advice - raison pour laquelle les advices ne sont pas nommés par défaut).
- Les deux peuvent lever des exceptions.
- Les deux peuvent utiliser `this` pour faire référence à l'objet courant ou l'aspect courant.
- Les advices de type `around` doivent retourner une valeur, et donc déclarer un type de retour.

Remarque : les déclarations de levées d'exceptions sont soumises aux mêmes contraintes que les redéfinitions de fonctions. La covariance impose les règles suivantes :

- Un advice ne peut déclarer lever une exception que si tous les points de jonction repérés par la coupe en font de même.
- Il peut déclarer lever des sous-exceptions de celles de ses points de jonction.
- Il peut lever des runtime exceptions.

Advices vs méthodes : différences

- Un advice n'a généralement pas de nom.
- On ne peut pas appeler un advice.
- Aucun modificateur n'est autorisé pour les advices (puisqu'ils ne sont pas appelables).
- Pas de type de retour pour les advices de type `before` et `after`.
- Les advices disposent de trois variables spéciales : `thisJoinPoint`, `thisJoinPointStaticPart` et `thisEnclosingJoinPointStaticPart`.
- Les advices de type `around` disposent du mot-clé `proceed` pour lancer le code associé aux points de jonction.

Advices `before`

Les advices de ce type sont les plus simples.
Ils sont exécutés avant que le code associé au point de jonction ne le soit.
Si l'advice lève une exception, le code associé au point de jonction n'est pas exécuté.

Exemple :

```
before() : execution(@Secured * *(..)) {
    ... vérification des droits de l'utilisateur ...
}
```