ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

07 Chapitre

Programmation orientée aspect

2I1AC3 : Génie logiciel et Patrons de conception

Régis Clouard, ENSICAEN - GREYC

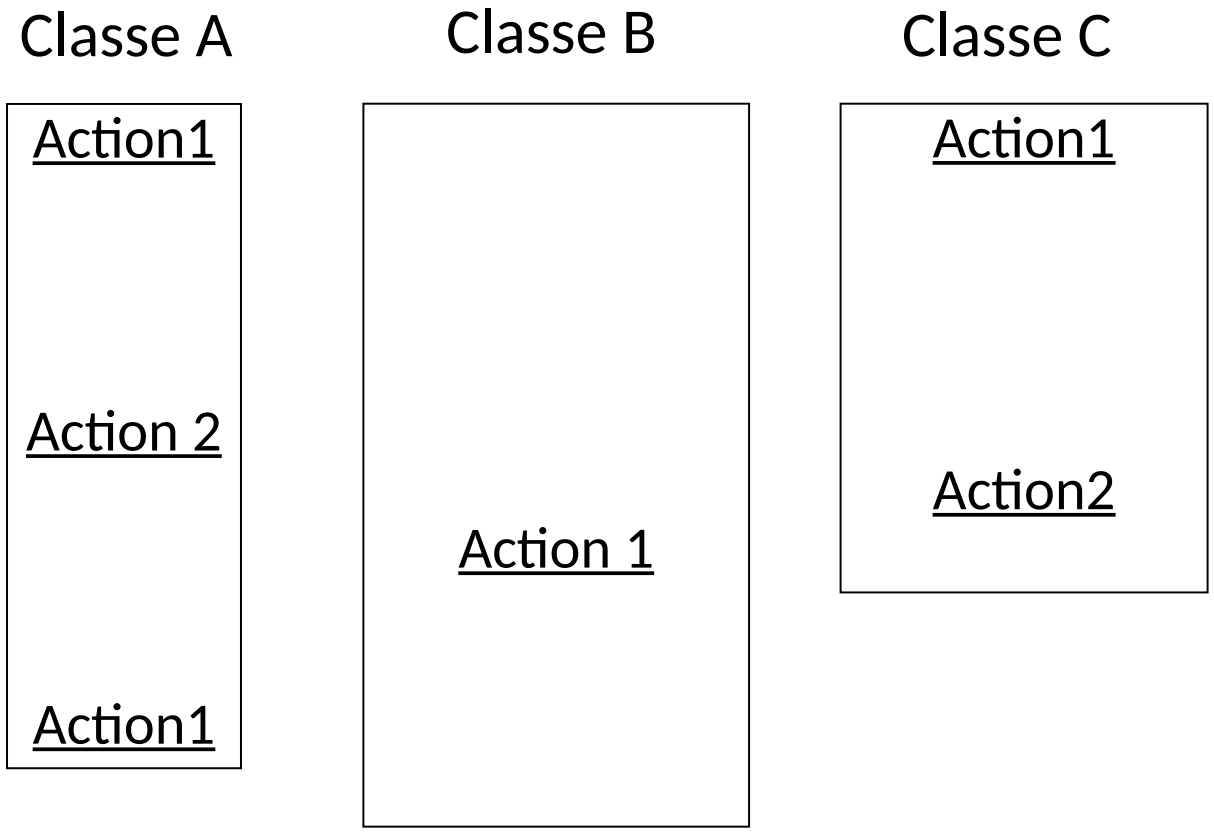
« L'homme est le meilleur ordinateur que l'on puisse
embarquer dans un engin spatial,
et le seul qui puisse être produit en masse
par du travail non qualifié. »
Wernher von Braun

Introduction

- Limitations de la programmation orientée objet :
  - Redondance de code
  - Code enchevêtré (*code tangling*)
    - mélange de plusieurs responsabilités à un même endroit du code.
  - Code éparpillé (*code scattering*)
    - une responsabilité qui est éclatée en plusieurs endroits du code.
  - Fonctionnalité transversale (*cross-cutting*)
    - une responsabilité qui s'applique à plusieurs endroits.

Fonctionnalité transversale (*cross-cutting*)

- Mélange de plusieurs responsabilités à un même endroit

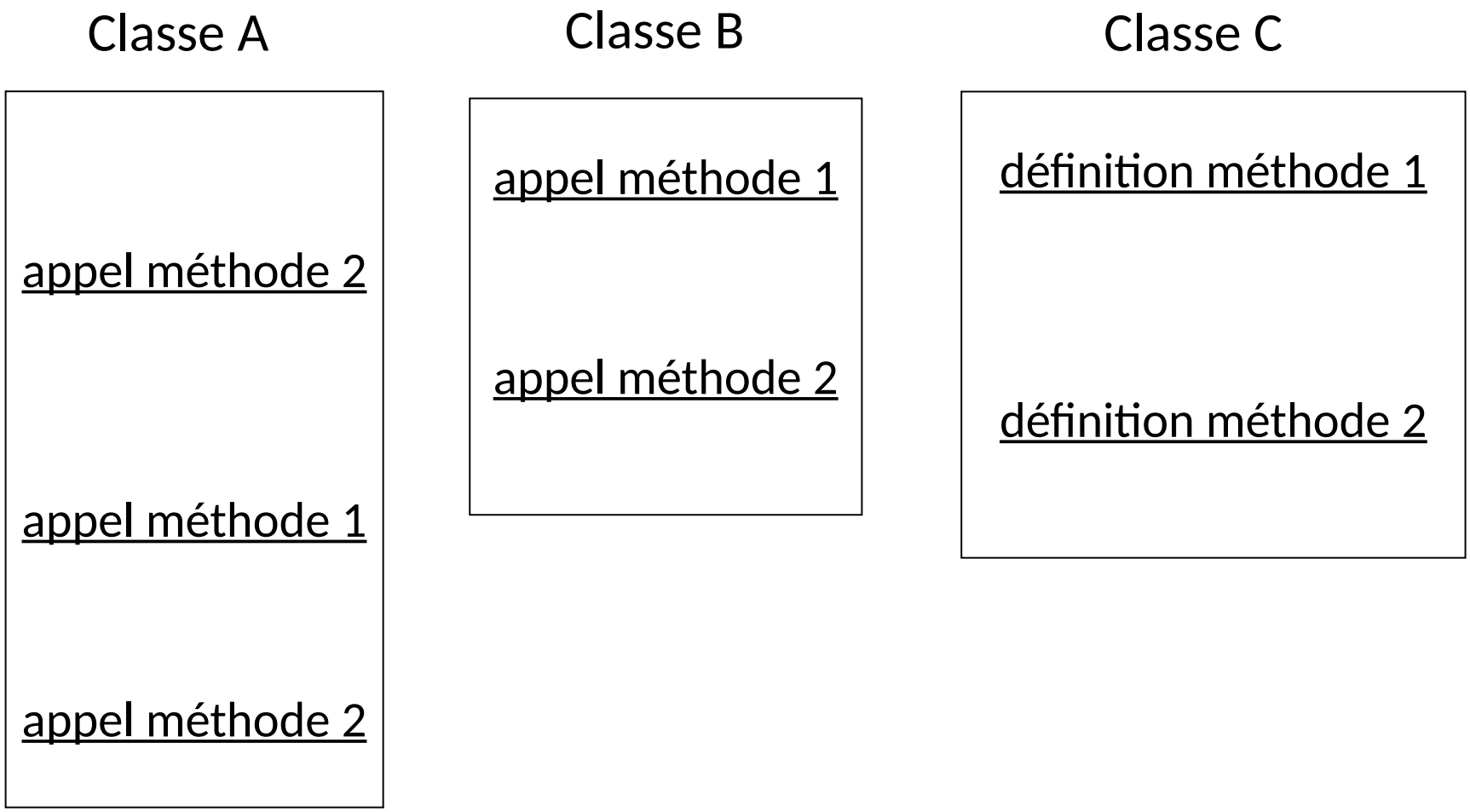


Exemple

- Contraintes d'intégrité fonctionnelle
  - Un objet Client ne doit pas être supprimé tant qu'une Commande par ce client n'est pas honorée.
- Approche POO
  - Modifier la méthode de suppression de la classe client pour vérifier l'absence de commande non honorée.
  - Schéma : Commande ----> Client
- Mais c'est une mauvaise solution pour 2 raisons :
  - La vérification de la commande non honorée n'appartient pas à la logique de la classe client. Elle est induite par la façon dont sont gérées les commandes.
  - La classe client n'est pas sensée connaître toutes les contraintes d'intégrité imposées par les autres classes.
- Il faudrait un moyen de définir toutes ces contraintes ailleurs, mais que ces contraintes s'implantent comme le ferait un programmeur POO.

Code éparpillé (*Code-Scattering*)

- Une responsabilité éclatée en plusieurs endroits



Exemple

- Persistance des données
  - Le stockage des valeurs d'attributs d'un certain nombre de classes dans une BDD à chaque fois que l'on modifie la valeur d'un attribut.
- Solution POO
  - Modifier l'accès en écriture aux attributs (accesseurs) et mettre du code de stockage.
- Mauvaise solution pour 2 raisons:
  - Il y a nécessité de créer des setters avec du code dupliqué.
  - Extrêmement laborieux et chronophage.
- C'est un frein à la maintenance et à l'évolution des applications. Toute modification de la manière d'utiliser un service entraîne des modifications nombreuses, coûteuses et sujettes à régression.

Solutions

- **Programmation orientée aspect**
  - Gregor Kiczales en 1996 à Xerox PARC.
  - Inspiration: Intelligence artificielle
    - Langage à objets (pas seulement orienté objet. Les classes sont aussi des objets)
    - Méta-programmation
    - Réflexivité
  - Nouveau paradigme de programmation.
- Histoire de la programmation :
  - Paradigme procédural
  - Paradigme POO
  - Paradigme POA

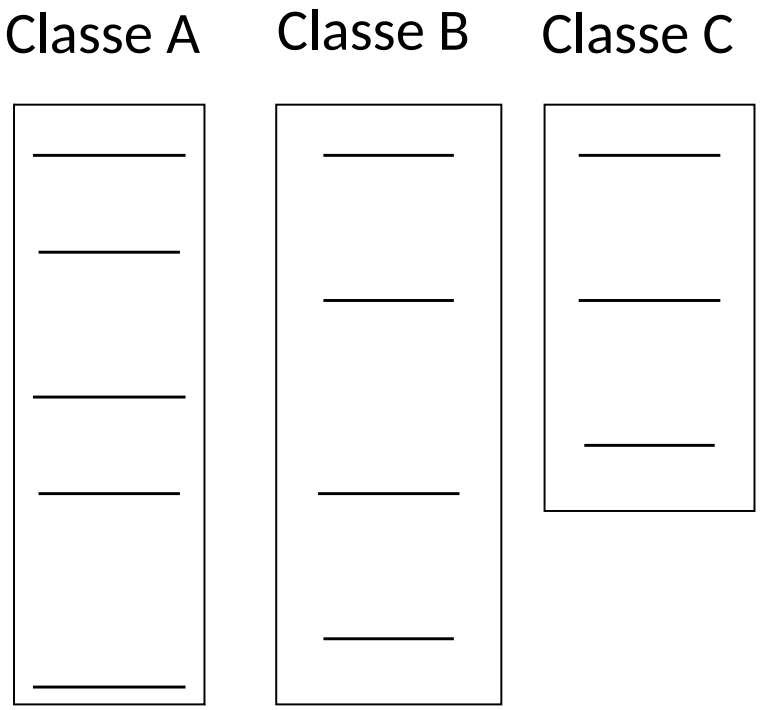
Paradigme de POA

- Séparer les parties fonctionnelles et non fonctionnelles
  - Partie fonctionnelle : responsabilité métier
    - p.ex., ajouter ou supprimer un employé.
  - Partie non fonctionnelle : responsabilités transverses
    - p. ex., contrôle d'accès et sécurité.

Programmation orientée aspect

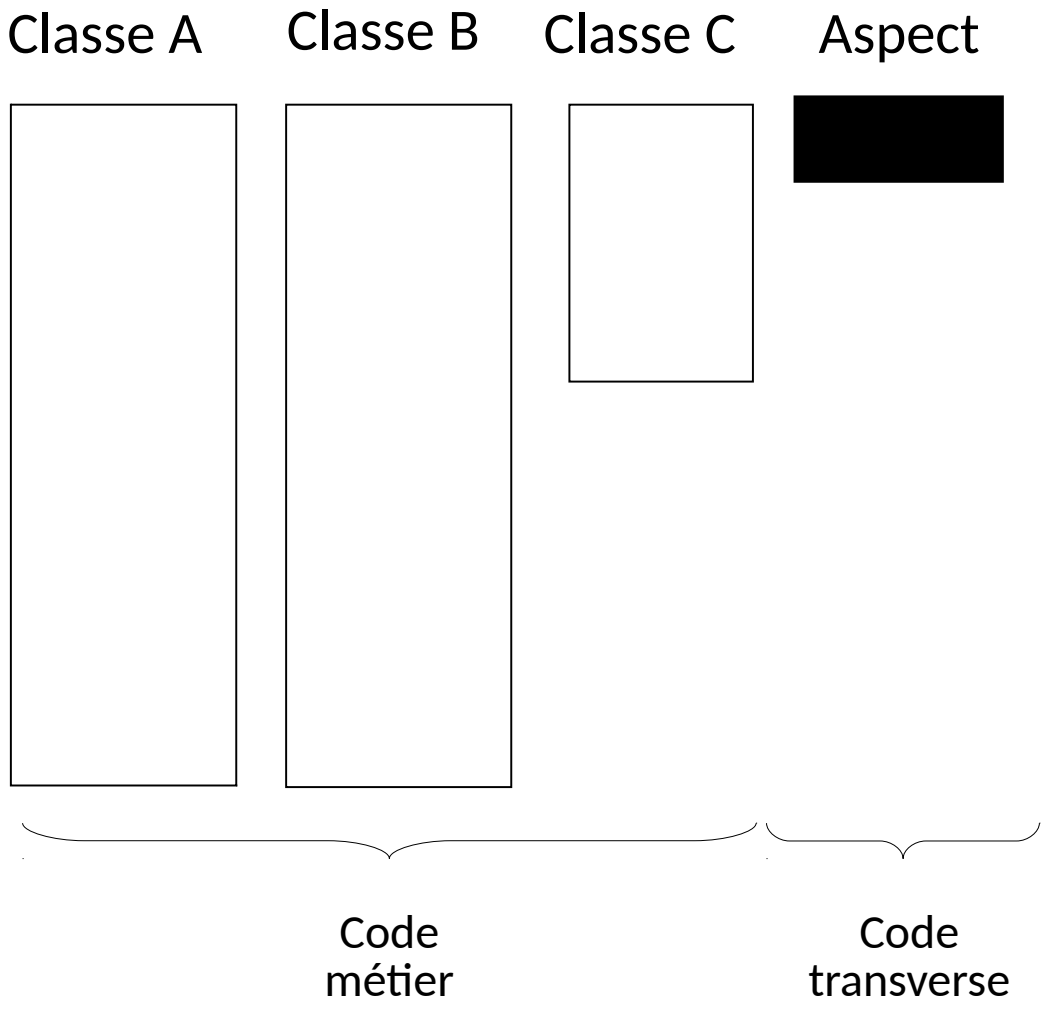
POO sans aspect

Les fonctionnalités transverses sont dispersées dans le code.



POO avec aspect

Les Les fonctionnalités transverses sont regroupées à un endroit.



Définitions

- **Aspect**
  - Unité de modularité.
  - Peut contenir des attributs et des méthodes comme une classe ordinaire.
- **Greffon** (Code Advice)
  - Le code à exécuter au point d'insertion.
- **Point d'insertion** (*Join Point*)
  - Place dans le code où une fonctionnalité traverse peut être ajoutée.
  - p. ex., appel de méthode, accès aux attributs, création d'objets.
- **Point d'action** (*Pointcut*)
  - Spécification des points d'insertion dans le code à l'aide d'expressionss régulières.

Illustration

Partie objet

Classe A

Point insertion A1

Point insertion A2

Point insertion A3

Classe B

Point insertion B1

Point insertion B2

Classe C

Point insertion C1

Point insertion C2

Partie aspect

Aspect

PointAction1(A1,A2,B1,C2)
PointAction2(A3,B2,C1,C2)

Au PointAction1 :
Code greffon 1

Aux PointAction1 & PointAction2 :
Code greffon 2

Modèle de point d'insertion

- L'appel d'une méthode peut être être capturée pour insérer un greffon.

```
public class HelloWorld {
    private int max = 10;

    public static void main( String args[]) {
        new HelloWorld().affichage();
    }

    public void affichage() {
        for (int i = 0; i < max; i++) {
            try {
                System.out.println("HelloWorld!");
            } catch (Exception e) {
                System.err.println("Problème d'affichage");
            }
        }
        max--;
    }
}
```

Modèle de point d'insertion

- L'appel de constructeur peut être capturé pour insérer un greffon.

```
public class HelloWorld {
    private int max = 10;

    public static void main( String args[]) {
        new HelloWorld().affichage();
    }

    public void affichage() {
        for (int i = 0; i < max; i++) {
            try {
                System.out.println("HelloWorld!");
            } catch (Exception e) {
                System.err.println("Problème d'affichage");
            }
        }
        max--;
    }
}
```

Modèle de point d'insertion

- Le lancement d'une exception peut être capturé pour insérer un greffon.

```
public class HelloWorld {
    private int max = 10;

    public static void main( String args[]) {
        new HelloWorld().affichage();
    }

    public void affichage() {
        for (int i = 0; i < max; i++) {
            try {
                System.out.println("HelloWorld!");
            } catch (Exception e) {
                System.err.println("Problème d'affichage");
            }
        }
        max--;
    }
}
```

Modèle de point d'insertion

- L'accès en lecture et écriture à la valeur d'un attribut peuvent être capturés pour insérer un greffon.

```
public class HelloWorld {
    private int max = 10;

    public static void main( String args[]) {
        new HelloWorld().affichage();
    }

    public void affichage() {
        for (int i = 0; i < max; i++) {
            try {
                System.out.println("HelloWorld!");
            } catch (Exception e) {
                System.err.println("Problème d'affichage");
            }
        }
        max--;
    }
}
```

Greffon (*Code Advice*)

- Le greffon est un code qui sera activé à un certain point d'insertion du système, précisé par un point d'action.
- Trois façons d'activer le code du greffon à un point d'insertion :
  - **Before advice**
    - exécuté avant le point d'insertion.
  - **After advice**
    - exécuté après le point d'insertion.
  - **Around advice**
    - Dans le code du greffon, le mot clé '*proceed*' est utilisé pour spécifier où sera inséré le code au point d'insertion. Ainsi :
      - Si le mot clé est absent, le code du point d'insertion n'est pas exécuté.
      - S'il est ajouté plusieurs fois, le code du point d'insertion est exécuté plusieurs fois.
      - On peut ajouter des choses avant et après le code d'insertion.

Tissage

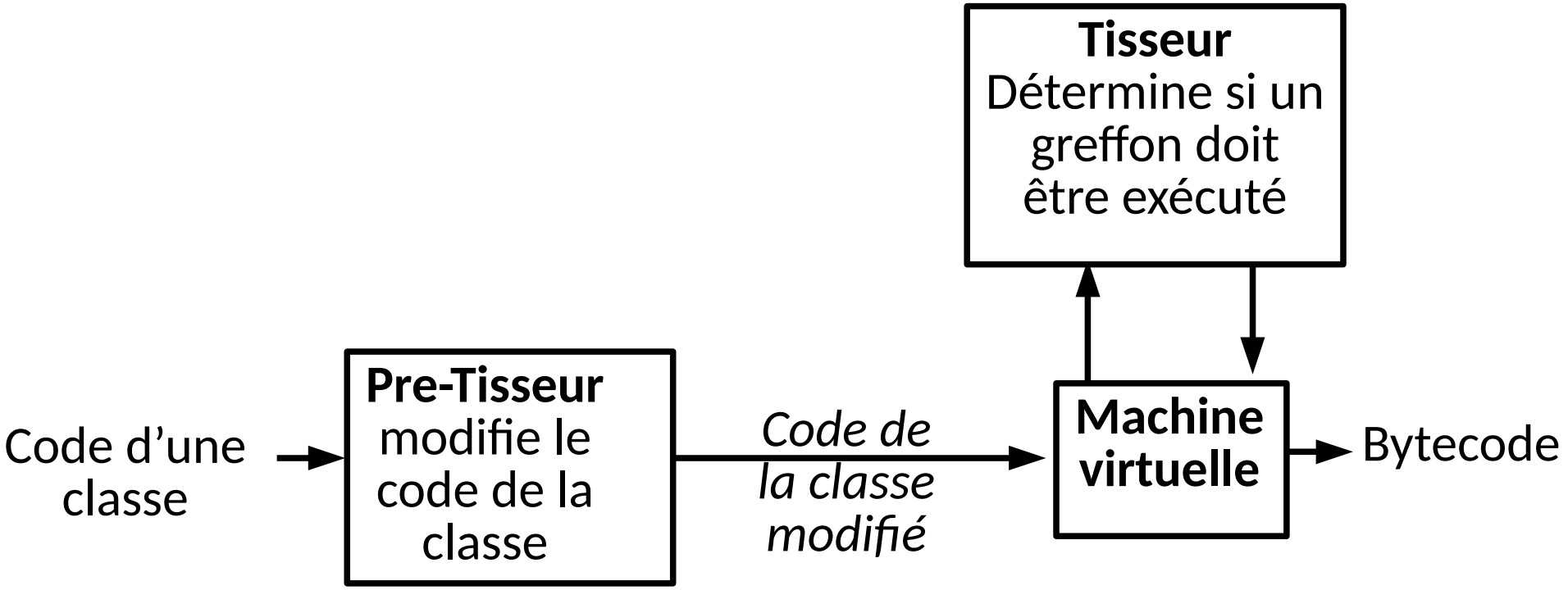
- Une application POA
  - Ensemble de classes
  - Ensemble d'aspects
- Tissage
  - Les règles de tissage spécifie comment intégrer le code des aspects dans les classes pour construire le système final.
- Deux types de tisseurs :
  - Tisseur statique
  - Tisseur dynamique

Tissage statique

- Tissage au moment de la compilation.
  - Il prend en entrée un ensemble de classes et un ensemble d'aspects pour fournir un programme compilé augmenté d'aspects.
    - P. ex. en Java, le tisseur produit du code Java classique compilable par la machine virtuelle.
    - Le tisseur n'est qu'un pré-compilateur.

Tisseur dynamique

- Tissage au moment de l'exécution.
  - Il est capable d'appliquer les aspects dynamiquement, pendant l'execution du programme.
  - Il a ainsi la capacité d'ajouter, de supprimer ou de modifier les aspects à chaud pendant l'execution du programme.
  - Il faut une nouvelle machine virtuelle.



Exemple d'un tisseur Java

- AspectJ (Open Source)
  - Tisseur statique
  - URL
    - AspectJ
      - http://www.eclipse.org/aspectj
    - Plug-in Eclipse
      - http://eclipse.org/ajdt

Exemple 1 : HelloWorld (avant tissage)

Hello.java

```
public class Hello {
  public static void main(String a[]){
    new Hello().sayHello();
  }

  public void sayHello() {
    System.out.println("Hello!");
  }
}
```

World.aj

```
public aspect World {
  pointcut salutation():
    execution(*Hello.sayHello(..));

  after(): salutation() {
    System.out.println("Salut à toi");
  }
}
```

Exemple 1 : HelloWorld (après tissage)

Hello.java

```
public class Hello {
  public static void main(String a[]) {
    new Hello().sayHello();
    System.out.println("Salut à toi");
  }

  public void sayHello() {
    System.out.println("Hello!");
  }
}
```

World.aj

```
public aspect World {
  pointcut salutation():
    execution (*Hello.sayHello(..));

  after(): salutation() {
    System.out.println("Salut à toi");
  }
}
```

Exemple 2 : Debug (avant tissage)

Exemple2.java

```
public class Exemple2 {
  private int _x=0;
  public static void main(String args){
    for (int i=0;i<10;i++) {
      Exemple2.increment();
    }
  }
  public void increment() {
    _x++;
  }
}
```

Debug.aj

```
public aspect Debug {
  pointcut methodEx2():
    execution(*Exemple2.*(..));
  pointcut CallIncr():
    execution(*increment(..));
  pointcut Ensemble():
    methodEx2() & CallIncr();

  around(): Ensemble() {
    System.out.println(_x);
    proceed
    System.out.println(_x);}
}
```

Exemple 2 : Debug (après tissage)

Exemple2.java

```
public class Exemple2 {
  private int _x=0;
  public static void main(String  a[]){
    for (int i=0;i<10;i++){
      System.out.println(_x);
      Exemple2.increment();
      System.out.println(_x);
    }
  }
  public void increment() {
    _x++;
  }
}
```

Debug.aj

```
public aspect Debug {
  pointcut methodEx2():
    execution(*Exemple2.*(..));
  pointcut CallIncr():
    execution(*increment(..));
  pointcut Ensemble():
    methodEx2() & CallIncr();

  around(): Ensemble() {
    System.out.println(_x);
    proceed
    System.out.println(_x);
  }
}
```

Les patrons de conception revisités avec la POA

- La POA permet d'améliorer l'implémentation des patrons
- Exemple du patron Singleton
  - Singleton en POO : inconvénients
    - Construction d'une instance non classique : `getInstance()`
    - Interdit l'héritage des singletons.
  - Singleton en POA
    - Ajoute un point d'insertion sur le constructeur.
    - Un attribut stocke la référence de l'instance.
    - Le code du greffon retourne une nouvelle instance (1er appel) ou la référence détenue par l'attribut.

Conclusion

- Nouveau paradigme
  - Facile à appréhender.
  - Peut résoudre rapidement des problèmes considérés comme insurmontables sans cela.
    - p.ex. ajout d'une responsabilité transverse dans un code de plusieurs MLOC.
- POO + POA = POO
  - C'est donc un ajout.
- POA est une extension de nombreux langages de programmation
  - Java
  - C++
  - Php
  - C#
  - D
  - Smalltalk
- POA est aussi utilisé dans plusieurs framework
  - p.ex., Spring