# Sensibilisation aux principes fondamentaux en conception et programmation objet et design pattern

Olivier Capuozzo

07 mars 2004

Ce document a été réalisé sous GNU/Linux avec vim [http://www.vim.org], au format docbook [http://www.oasis-open.org/docbook/], mis en page avec le processeur XSLT saxon [http://saxon.sourceforge.net] développé par Michael Kay et les feuilles de styles de Norman Walsh [http://nwalsh.com/].

## Table des matières



# Introduction à la notion de principes de conception objet

## Références

- [Meyer] Conception et programmation orientées objet . Bertrand Meyer. Ed. Eyrolles. Juin 2000
- [Gof] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley 1999.

- [MARTIN00 [http://objectmentor.com]] Design Principles and Design Patterns par Robert C. Martin.
- [Principes avancés de conception objet [http://www.design-up.com]] un dossier d'introduction, en Français, sur ce thème, par l'équipe de *Design-up*. A lire à la suite de cet article.
- [Java Design] Buidling better apps & applets par Peter Coad et Mark Mayfield ed. Prentice Hall PTR
- [JOUP] Objects, UML and Process par Kirk Knoernshild ed. Addison-Wesley, 2002.
- [Larman] UML et les Design Patterns par Graig Larman, ed. CampusPress, 2002.
- [Kent Beck] Extreme Programming la référence, ed. CampusPress, 2002.
- [Robert C. Martin] UML for Java Programmers, ed. Prentice Hall, 2003.


### Objectifs

- Présenter les principaux principes en conception et programmation objet
- Exemple de mise en oeuvre de ces principes et exercices

# Approche objet du logiciel

"Un système bien conçu est facile à comprendre, facile à modifier et facile à réutiliser" (in UML for Java programmers, R. C. Martin, ouvrage mentionné ici Références).

Les principales qualités visées par l'approche objet sont la Robustesse, Maintenabilité, Extensibilité et Réutilisabilité.

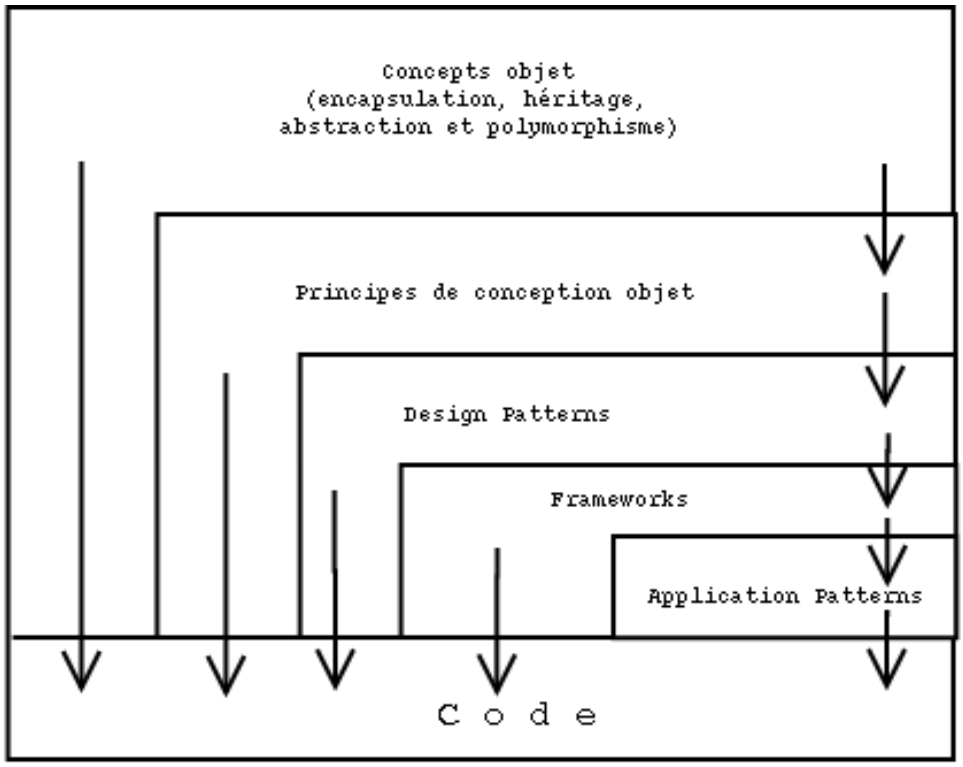
- *Robustesse*: Présence et respect de pré et post conditions, conforme à l'esprit de la *programmation par contrat* de B. Meyer. Gestion des exceptions.
- *Maintenabilité* : Respect de conventions d'écriture et présence de tests unitaires.
- *Extensibilité* : Représentation des parties stables par des classes concrètes. Abstraction des parties extensibles.
- *Réutilisabilité* Gestion des dépendances (interfaces, classes et paquetages) en vue d'une réutilisation dans une autre application.

A eux seuls, les concepts objets ne suffisent pas à produire des logiciels de qualité.

La mise en oeuvre, et le maintien, de ces qualités est assurée par le respect de cetains principes tout au long du cycle de vie du système.

Bien entendu, on peut faire sans ; sans respecter les principes, sans respecter les modèles de réalisation de ces principes, cette "liberté" est représentée par la figure ci-dessous :

### Figure 1. Production de code et outils de conception




Alors que les principes décrivent QUOI faire, les *designs patterns* (modèles de conception) montrent COMMENT le faire, dans un contexte donné. Quant aux frameworks (solution d'architecture applicative), ils démontrent, à leur manière, comment implémenter certains Designs Patterns.

Nous verrons quelques principes fondamentaux, et quelques exemples d'application.

# Avertissement : YAGNI

En conception (*design*), il n'existe pas de règles strictes à appliquer, à l'image des *formes normales* en analyse de données (quitte à les enfreindre ensuite), mais des principes guidant la conception.

Pourquoi parler de guides et non de règles ?

Il n'est pas conseiller de vouloir toujours "faire du générique tout de suite", au risque de consommer du temps inutilement.

YAGNI veut dire *"You aren't gonna need it"* (**Vous n'allez pas avoir besoin de lui**). C'est un principe de précaution fondamental d'Extreme Programming, qui invite à la simplicité, mais pas n'importe laquelle... Si le sujet vous intéresse, l'ouvrage de Kent Beck est reconnu comme une excellente introduction en la matière (Références).

## Coût de la fléxibilité

*Ce qui est flexible est complexe.*

Il toute fois préférable de limiter la compléxité. Il s'agit donc de concilier fléxibilité et simplicité de conception.

Parfois la complexité peut être cachée par l'adoption d'un framework, mais le système devient alors dépendant de celui-ci.

# Principe d'Ouverture/Fermeture

- *Principe d'Ouverture/Fermeture ---- Open-Closed Principle - OCP*

## OCP

*Tout module (package, classe, méthode) doit être ouvert aux extensions mais fermé aux modifications.*

Ce principe, que l'on doit au travail de Bertrand Meyer, est considéré comme le plus important des principes en conception objet [JOUP]. Ces implications sont nombreuses (Design by Contrat) et font l'objet d'autres principes (LSP, DIP...).

- *Ouvert aux extensions*

  Comprendre : le *comportement* du système devrait être *extensible*. En effet, aucun système n'est à l'abri de nouveaux besoins. Les parties changeantes d'un système doivent être abtraites, offrant ainsi une ouverture pour d'autres implémentation que celles initialement prévues.

  Techniques utilisées : Abstraction (classe abstraite, interface), polymorphisme (ne pas tester le type d'un objet avant de lui envoyer un message) et sous-traitance d'instanciation (factory).

- *Fermé aux modifications*

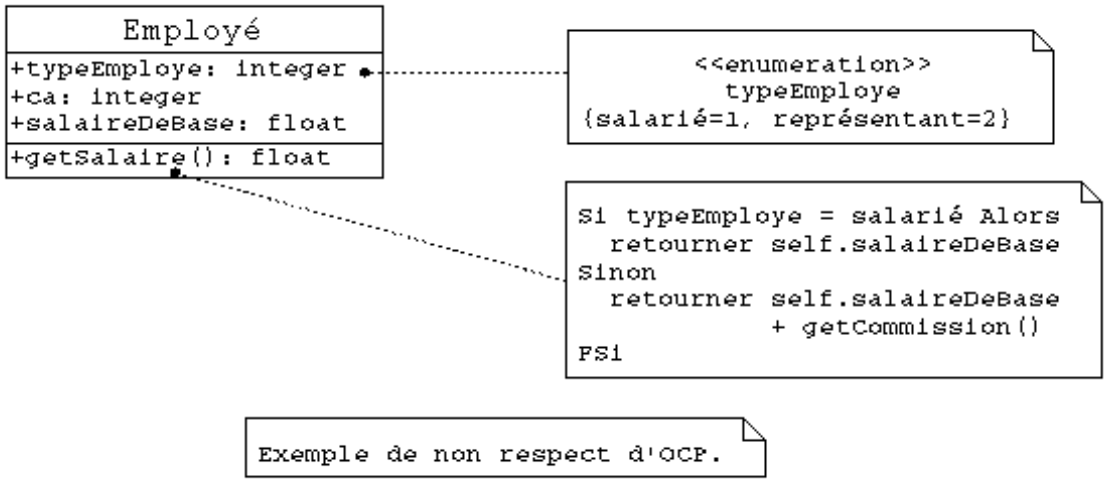
  L'implémentation des classes/opérations/attributs ne doit pas être soumise aux changements.

  Techniques utilisées :

  - Rendre *privés* tous les attributs (principe de rétention d'information de B. Meyer)
  - Seuls sont visibles (*public*), les méthodes qui implémentent les opérations (une «opération» réalise un service).
  - Les invariants algorithmiques sont implémentés, les parties changeantes sont représentées par des méthodes abstraites.

Illustration

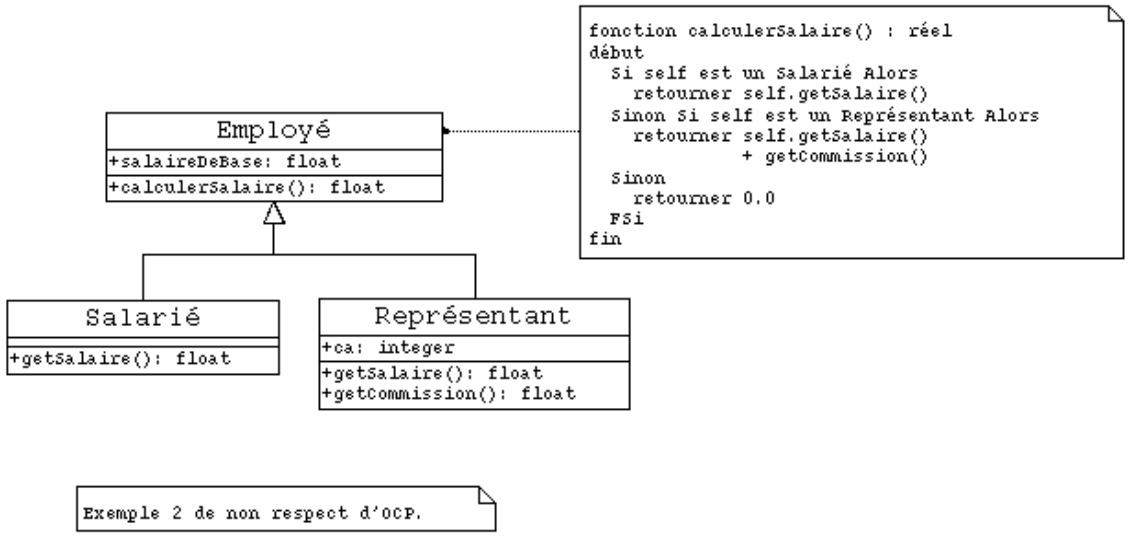
### Figure 2. Non respect d'OCP



Cet exemple ne respecte pas OCP :

- Les attributs ne sont pas cachés (le signe + signifie *public*).
- La méthode *getSalaire* n'est pas ouverte aux changements (une nouvelle catégorie de personnel nécessitera de recoder le comportement de cette méthode, en autres).

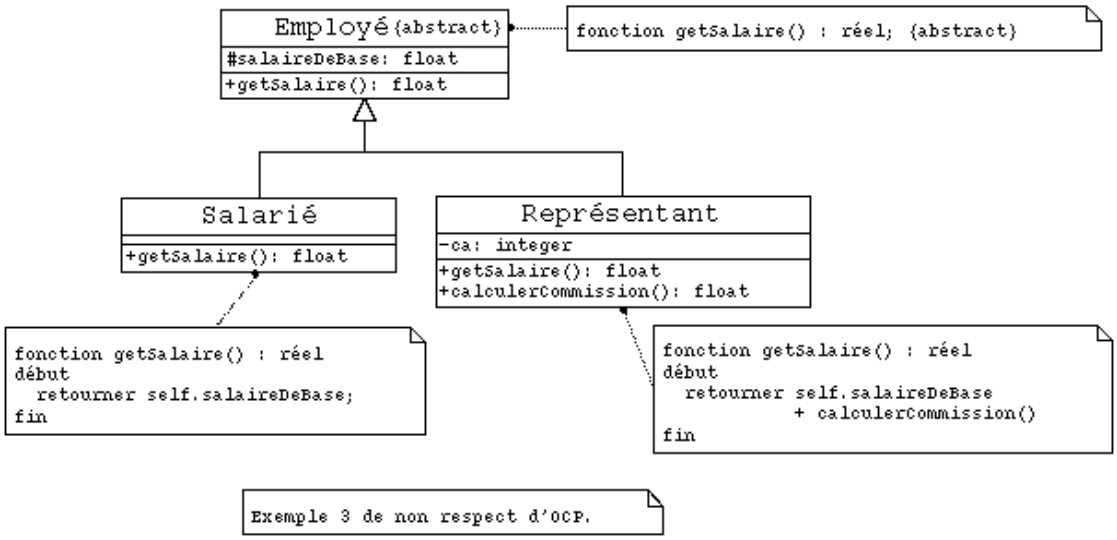
## Figure 3. Non respect d'OCP




Cet exemple ne respecte pas OCP :

- Les attributs ne sont pas cachés.
- La méthode *getSalaire* n'est toutjours pas ouverte aux changements, elle a besoin de tester la classe de l'instance dans sa définition (n'utilise pas le polymorphisme).

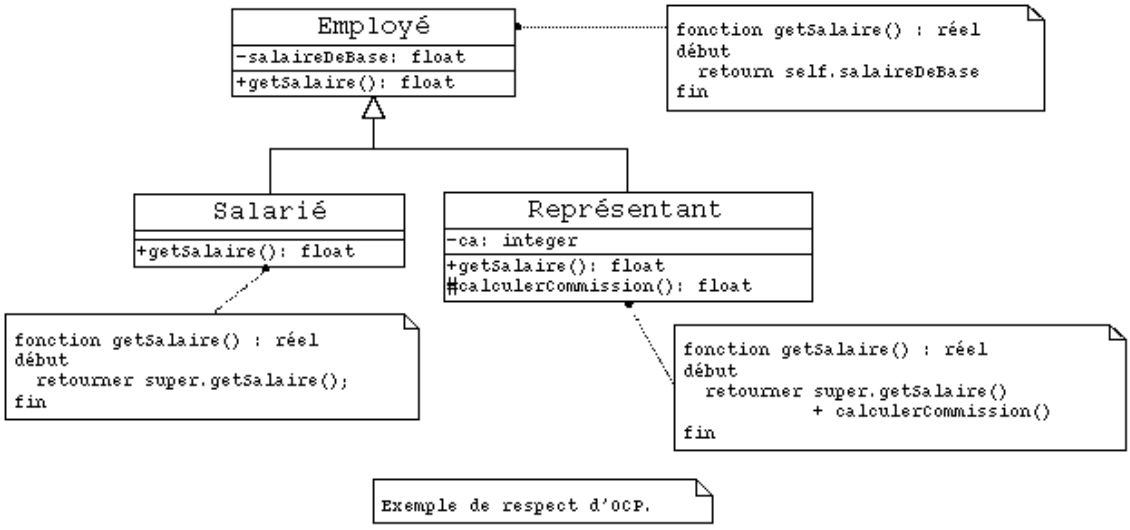
## Figure 4. Non respect d'OCP




Cet exemple ne respecte pas OCP :

- Les attributs ne sont pas cachés aux classes descendantes.
- Si elle n'est utilisée que par *getSalaire*, la méthode *calculerCommission* devrait être cachée.

## Figure 5. Respect d'OCP




Nous appliquons OCP à la classe `Représentant` si nous "gelons" la méthode `Représentant::getSalaire` et ne fournissons aucun moyen aux classes descendantes de modifier le `ca`.

Une classe descendante, par exemple `RepréssentantInterim hérite de Représentant`, pourra personnaliser le comportement de `Représentant::getSalaire` en implémentant différement la méthode protégée `RepréssentantInterim::calculerCommission`.

La plupart du temps ce type de solution fait intervenir une `Interface`.

*Quand appliquer OCP ?*

Les données devraient toujours être cachées. Dans ce cas, elles ne sont accessibles par des opérations (dite *getter/setter* ou par un mecanisme plus puissant : les propriétés, *property*, respectant ainsi le principe d'accès uniforme de B. Meyer).

Par contre, il convient d'être plus réservé quant à une mise en oeuvre systématique de l'abstraction (via des interfaces) qui augmente de façon non négligeable le nombre de classes du système, et le temps de développment. On appliquera OCP sur des parties « qui en valent la peine», à forte probalité de changements.

A ce sujet, Design-Up [http://www.design-up.com] nous conseille *d'identifier correctement les points d'ouverture/fermeture de l'application, en s'inspirant :*

- *Des besoins d'évolutivité exprimés par le client*
- *Des besoins de flexibilité pressentis par les développeurs*
- *Des changements répétés constatés au cours du développement*

*La mise en oeuvre de ce principe reste donc une affaire de bon sens, sachant que la meilleure heuristique reste la suivante : on n'applique l'OCP que lorsque cela simplifie le design.*

- *Technique utilisée*

  - Implémenter les parties stables (classe, méthode) et abstraire les parties changeantes (interface) - voir le design pattern Template/Hook -
  - Encapsuler systématiquement les attributs.

# Principe de Substitution de Liskov

- *Principe de Substitution de Liskov ---Liskov Substitution Principle - LSP*

## LSP

*Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.*

LSP est le fruit d'un travail du Barbara Liskov qui est dérivé du concept de *Design by Contrat* de Bertrand Meyer, en particulier les notions de pré-condition et post-condition.

Une *pré-condition* est un contrat que doit respecter le client d'un service. Si une pré-condition d'un méthode ne peut être respectée, cette méthode ne doit pas être appelée.

Une *post-condition* est un contrat que doit respecter le fournisseur d'un service. Si une méthode ne peut assurée une post-condition, elle ne doit pas retourner.

- *Quand appliquer LSP ?*

  Chaque fois que l'héritage est mise en oeuvre : Héritage d'implémentation (redéfinition de méthodes) et héritage d'interface (redéfinition de assertions).

- *Technique utilisée*

Contrôler les contrats par une gestion des exceptions.

Remarque : Il est très difficile, en l'absence de pré et post conditions, de vérifier le respect de ce principe.

Les pré-conditions définies par les sous-classes ne doivent pas être plus restrictives que celles héritées.

Les post-conditions définies par les sous-classes ne doivent pas être plus larges que celles héritées.

*Exemple de non respect de LSP*

```
interface A {
 /**
  @pre  : x in [1..10]
  @post : m(x) in [1..20]
 */
  int m(int x) throws NumberFormatException;
}

interface B extends A {
 /**
  @pre  : x in [1..5]
  @post : m(x) in [0..20]
 */
 int m(int x) throws Exception;
}

class C implements B {
 public int  m(int x) throws NumberFormatException {
   System.out.println("C::m()");
   throw new NumberFormatException();
 }
}
```

# Principe d'Inversion des Dépendances

- *Principe d'Inversion des Dépendances ---Dependency Inversion Principle - DIP*

## DIP

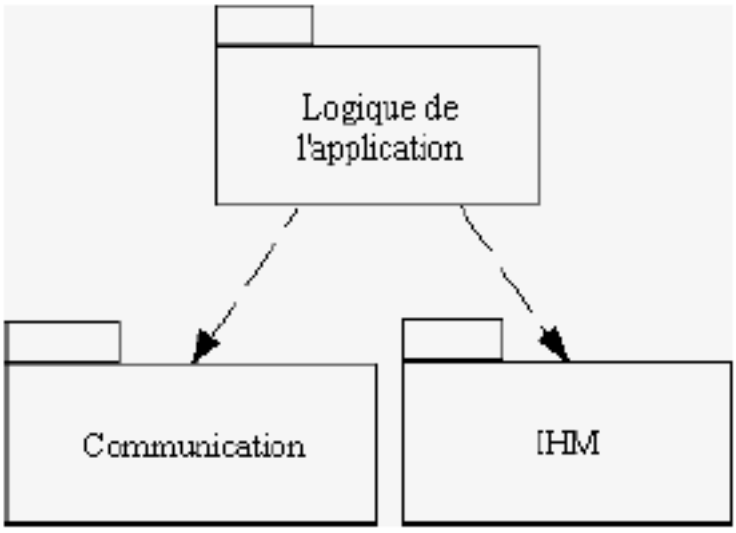
*A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions.*

*B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.*

DIP. Contrairement aux idées reçues, les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. En effet, le changement d'un module de bas niveau impacte l'ensemble des modules qui lui sont dépendants, necessitant une recompilation en chaîne. Ce type de dépendance nuit également à la réultisation de modules de haut niveau.

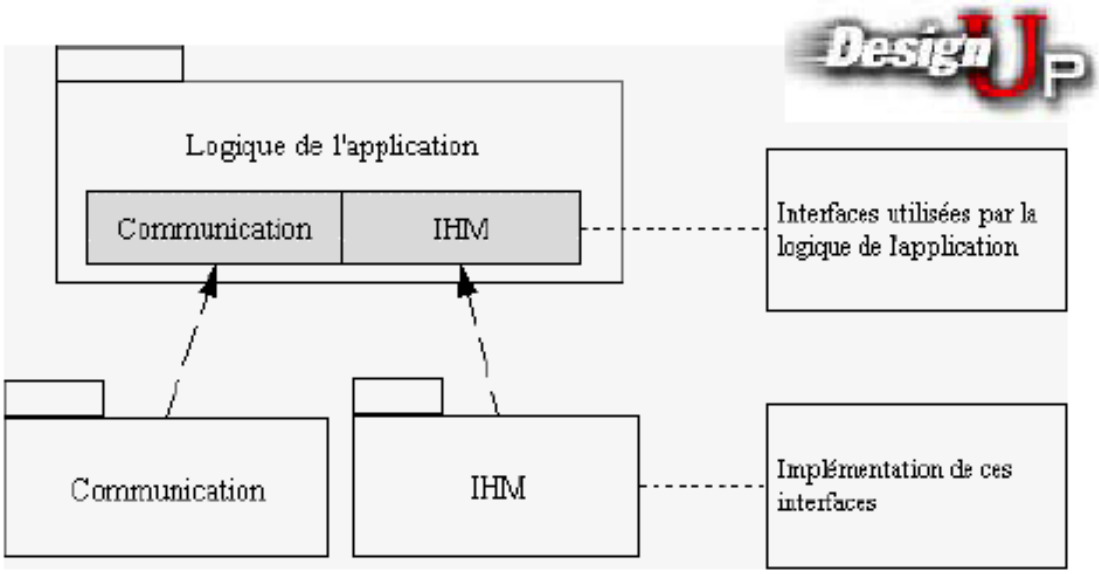
Illustration (*Design-up*).

**Figure 6. Avant DIP (Design-up)**




Selon ce principe, la relation de dépendance doit être inversée :*Les modules de bas niveau doivent se conformer à des interfaces définies et utilisées par les modules de haut niveau.*

**Figure 7. Après DIP (Design-up)**




- *Quand appliquer DIP ?*

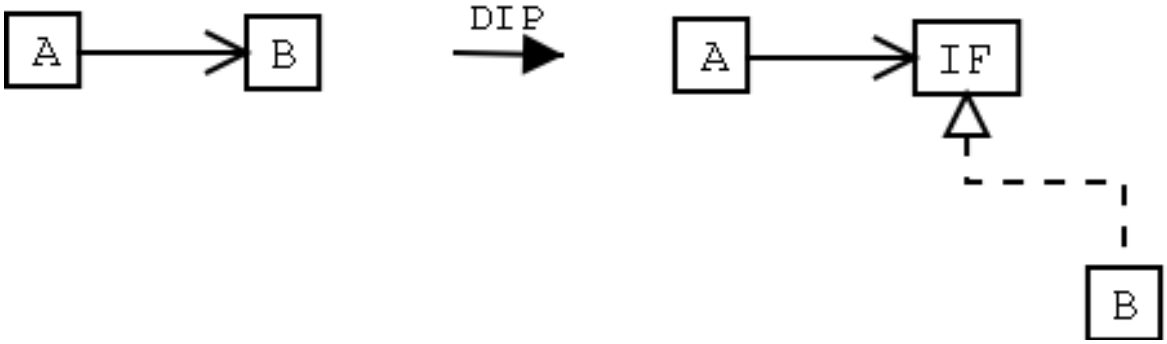
  Chaque fois que l'OCP est envisagé.

- *Technique utilisée*

  Ce sont les techniques utilisées pour OCP, couplées avec LSP.

  Plus précisemment, lorsqu'un module A dépend d'un module de bas niveau B (couplage concret), on crée une interface I que le module A utilise (couplage abstrait) et le module B réalise. Le module A est alors libéré du module B, devenu substituable.

**Figure 8. Appliquer DIP**



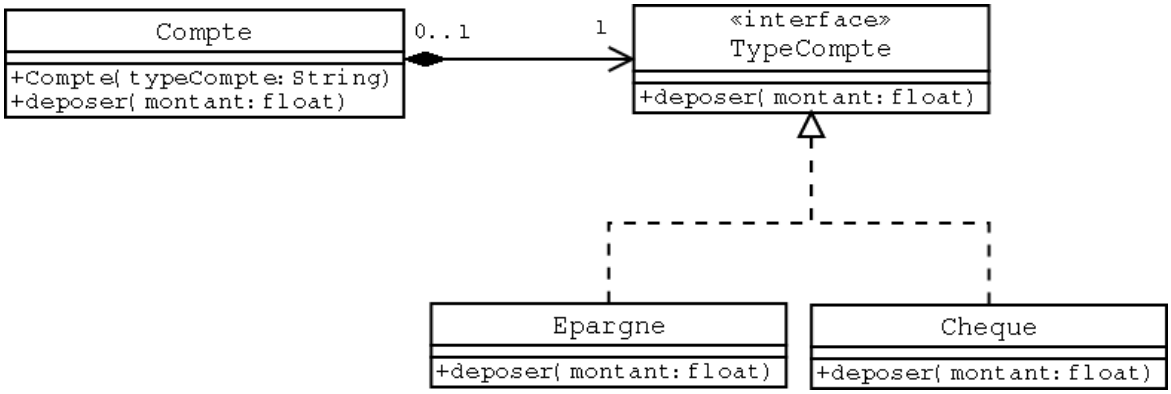

*Exemple de non respect de DIP*

```
class Client {
  OracleDB oracle;
  ...
  static Client getInstance(String id){
    Client res = null;
    String sql = "Select ...";
    ...
    ResultSet rs = oracle.execute(sql);
    ...
    return res;
  }
}

class OracleDB {
  String chaineConnect = "...";
  ...
  ResultSet execute(String sql) { ... }
  ...
}
```

*Même exemple respectant DIP*

```
class Client {
  DB db;
  ...
  static Client getInstance(String id){
    Client res = null;
    String sql = "Select ...";
    ...
    ResultSet rs = db.execute(sql);
    ...
    return res;
  }
}

interface DB {
  ...
  ResultSet execute(String sql);
  ...
}

class OracleDB implements DB {
  String chaineConnect = "...";
  ...
  ResultSet execute(String sql) { ... }
  ...
}
```

**Figure 9. Autre exemple DIP [JOUP]**




Voici un exemple d'implémentation :

```
class Compte {
  private TypeCompte _typec;

  public Compte(String typeCompte) throws Exception {
     Class c = Class.forName(typeCompte);
     this._typec = (TypeCompte) c.newInstance();
  }

  pubic void deposer (float montant){
     this._typec.deposer(montant);
  }
}

interface TypeCompte {
  void deposer(float montant);
}

class CompteEpargne implements TypeCompte {
  public void deposer(float montant){
    System.out.println();
    System.out.println("Montant déposé sur le compte épargne : " + montant);
    System.out.println();
    System.out.println();
  }
}

class CompteCheque implements TypeCompte {
  public void deposer(float montant){
    System.out.println();
    System.out.println("Montant déposé sur le compte chèque : " + montant);
    System.out.println();
    System.out.println();
  }
}
```

On constate que le couplage abstrait est respecté. Le lien entre un objet de la classe Compte et objet de type TypeCompte est réalisé par une simple chaine de caractères passée au constructeur de Compte. Le schéma est :

```
Class c = Class.forName(<nom d'une classe enfant>);
<Une classe parent> ancetre = (<Une classe parent>) c.newInstance();
```

Toute fois cette approche nécessite, pour des questions de sécurité, de prendre quelques précautions. En général, on applique quelques unes des idées présentées par les designs patterns de type Créateur (*factory*).

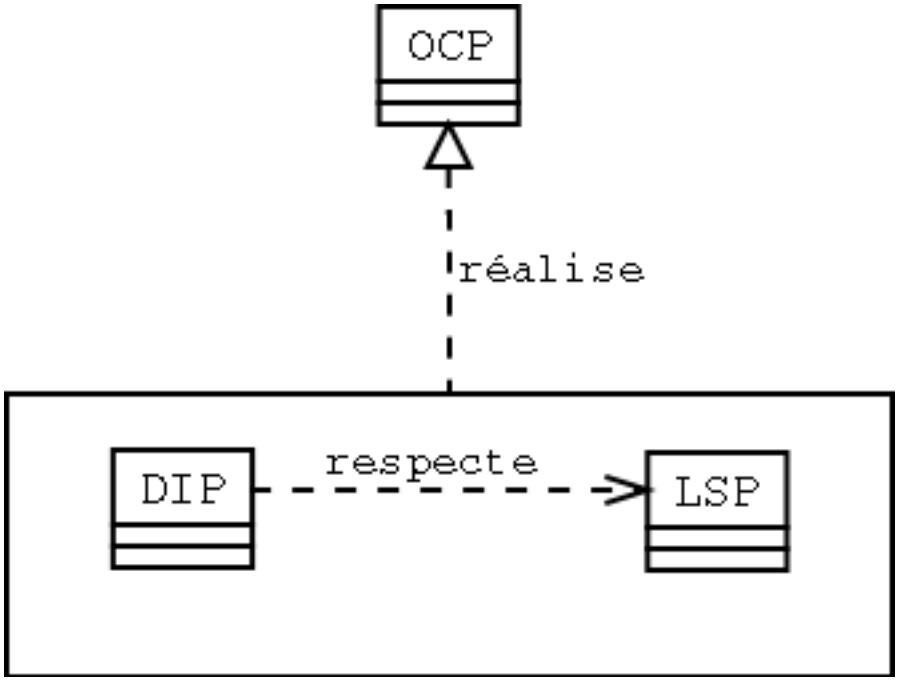
### Exercice

- Concevoir un programme (en mode console) qui crée un compte chèque et y dépose 300 euro, puis 200 euro sur un compte epargne.

# Remarque

Il existe une relation étroite entre ces trois principes. DIP, associé à LSP, nous explique comment adhérer à OCP. En effet, les parties fermées (OCP) doivent s'appuyer sur des interfaces (DIP) clairement exprimées et correctement réalisées (LSP).

**Figure 10. Les trois grands principes**




# Principe de séparation des interfaces

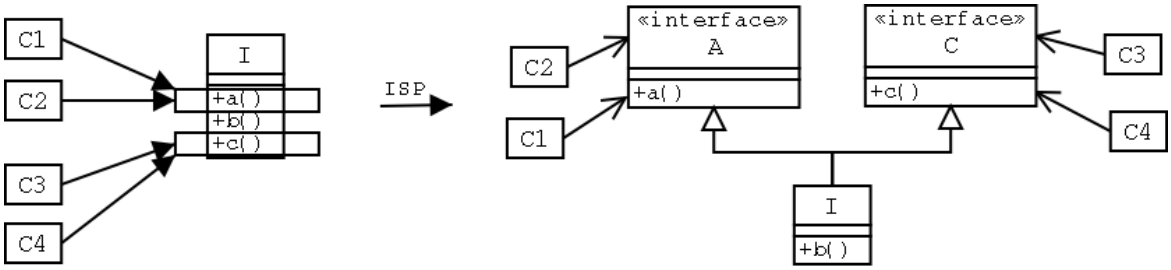
- *Principe de séparation des interfaces ---Interface Segregation Principle - ISP*

### ISP

*Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.*

Les opérations d'une interface doivent servir le même but.

**Figure 11. Application d'ISP**




- *Quand appliquer ISP ?*

  Lors de la création d'une interface, ISP aide à mettre met l'accent sur sa cohérence.

- *Technique utilisée*

  Création d'interfaces et héritage multiple.

# Principe de Réutilisation par Composition

- *Principe de Réutilisation par Composition ---Composite Reuse Principle - CRP*

  ## CRP

  ```
  Préférer la composition d'objets à l'héritage de classes.
  ```

  Ce principe est discuté pour la première fois dans Gof. Les développeurs ont tendance à abuser de l'héritage d'implémentation.

- *Quand appliquer CRP ?*

  Prenons le problème à l'envers. Coad a défini 5 règles qui doivent être toutes vérifiées pour une bonne utilisation de l'héritage :

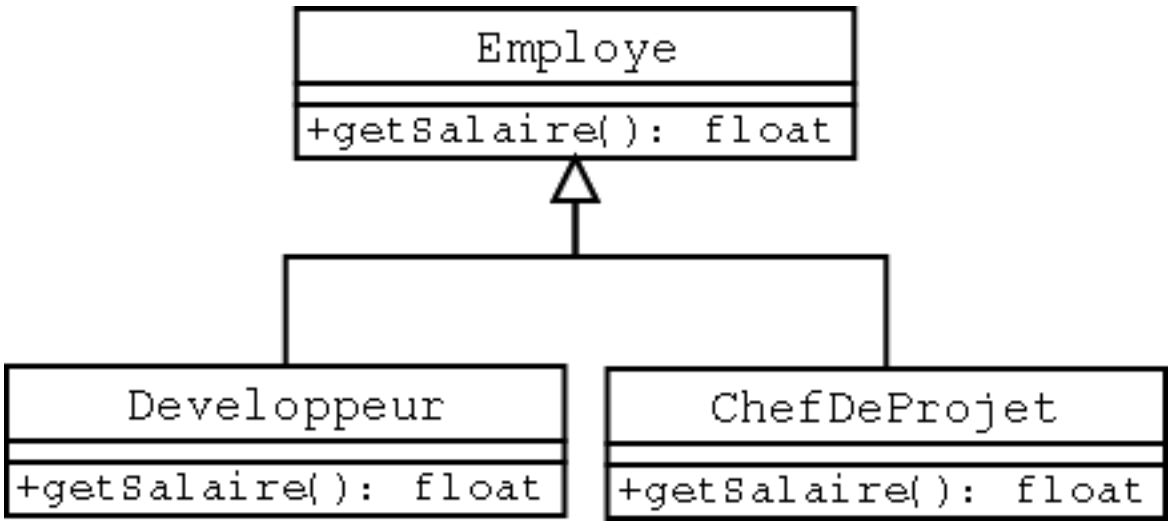
  1. La relation de sous-type est «est une sorte spéciale de» et non «est un rôle joué par un».
  2. Un objet de la classe n'a jamais besoin de transmuter (changer de classe).
  3. La sous-classe étend mais ne nullifie pas les comportements hérités.
  4. Ne pas sous-classer pour de simples raisons pratiques, pour simplifier des problèmes techniques.
  5. A l'intérieur du domaine du problème, la relation est «est une sorte spéciale de {rôles, transactions ou choses}».

  Si l'ensemble de ces 5 règles n'est pas vérifié, alors la délégation (composition d'objets) doit être préférée à l'héritage.

- *Technique utilisée*

  Délégation.

  **Figure 12. Exemple de non respect de CRP**



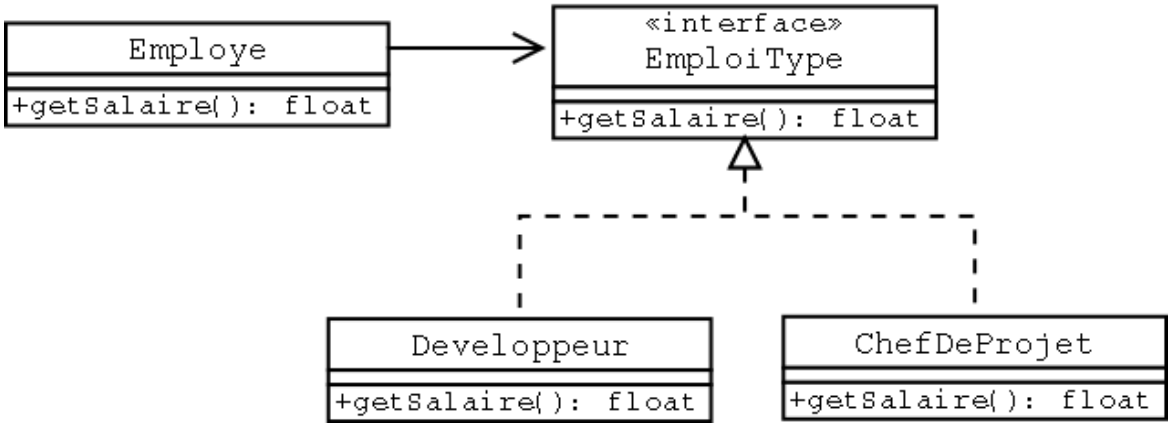

```
abstract class Employe {
  ...
  abstract public float getSalaire()
}

class Developpeur extends Employe {
  public float getSalaire() { ... }
  ...
}

class ChefDeProjet extends Employe {
  public float getSalaire() { ... }
  ...
}
```

Vérifions les règles :

1. Faux
2. Hum...
3. Vrai
4. Vrai
5. Vrai

3 critères sur 5.

**Figure 13. Exemple de respect de CRP**




```
class Employe {
  EmploiType emploi;
  float getSalaire() { return emploi.getSalaire(); }
  ...
}

interface EmploiType {
  public float getSalaire();
}

class Developpeur implements EmployeType {
  public float getSalaire() { ... }
  ...
}

class ChefDeProjet implements EmployeType {
  public float getSalaire() { ... }
  ...
}
```

# Exercice

Voici un programme qui construit une page HTML représentant les caractères affichables de la table ASCII (code 32 à 127), accompagnés de leur valeur ordinale exprimée en base dix ou deux, selon l'argument fourni par l'utilisateur.

```
class TableAsciiToHTML {
  private char typeRepr;


  public TableAsciiToHTML(String typeRepr) {
    this.typeRepr= (typeRepr == null) ? 'd' : typeRepr.charAt(0);
    printHTML();
  }

  private void printHTML() {
    int deb = 32;
    int fin = 128;
    int nbCol = 10;
    int cpt = 0;
    System.out.println("<html><head /><body><center><h1>TABLE DE CARACTERES</h1>");
    System.out.println("<table border="1">");
    for (int i = deb; i < fin; i++, cpt++) {
      if (cpt%nbCol == 0) {
         if (i>deb)  System.out.println("</tr>");
         System.out.println("<tr>");
      }
      System.out.println("<td align=\"center\">");
```

```
      System.out.println("<table border=\"1\"> <tr>");
      System.out.println("<td bgcolor=\"teal\" align=\"center\">");
      switch (this.typeRepr) {
       case 'd' :
         System.out.println(toDecString(i));
         break;
       case 'b' :
         System.out.println(toBinString(i));
         break;
       default :
         System.out.println(toDecString(i));
      }
      System.out.println("</td></tr><tr>");
      System.out.println("<td bgcolor=\"#CC3300\" align=\"center\">");
      System.out.println(" &#" + i + "; </td></tr></table></td>");
    }
    System.out.println("</tr></table></center></body></html>");
  }

  private String toDecString(int n) {
     return String.valueOf(n);
  }

  private String toBinString(int n) {
     return Integer.toBinaryString(n);
  }

}

public class AppTableAscii {
  static void main(String[] args) {
     String arg = (args.length>0) ? args[0] : null;
     TableAsciiToHTML app = new TableAsciiToHTML(arg);
  }
}
```

Recopier ce programme puis compiler le.

```
javac AppTableAscii.java
```

Exécuter le en redirigeant la sortie standard vers une fichier portant l'extension *.html*.

```
java AppTableAscii > ascii.html
```

Visualiser le résultat avec un navigateur.

Recommencer en passant une valeur ('b') en ligne de commande

```
java AppTableAscii b > ascii.html
```

Visualiser le résultat avec un navigateur.

Bon, ok, le programme fonctionne. Toute fois il n'est pas très propre, les parties extensibles ne sont pas abstraites.

**On souhaiterait proposer de nouvelles représentations des entiers** *sans avoir à retoucher l'existant (une fois retouché bien entendu).*

On vous demande d'appliquer OCP (et DIP) sur cet existant. La refonte de l'application ne doit pas entrainer de changement visible de son comportement, les fonctionnalités restent identiques et l'utilisateur n'y voit que du feu... Remarque : La refonte d'une partie du code d'une application, sans impact sur ses fonctionnalités est appelée *Refactoring*, une activité quotidienne du développeur reconnue par eXtreme Programming.

Idée : Réaliser un couplage abstrait entre la logique de l'application (contruction d'une page HTML) et la

représentation des nombres.

Objectif et test : Une fois l'application reconstruite, introduire une nouvelle représentation des valeurs ordinales en base 16, et ce sans intervenir sur le code existant de l'application.

Coad et Mayfield [Java Design] préconisent la stratégie suivante :

1. Rechercher la caractéristique polymorphe
2. Identifier un ensemble de noms de méthodes correspondant à cette caractéristique
3. Ajouter une interface
4. Identifier les implémentations

# Correction de l'exercice Table Ascii - OCP DIP

Coad et Mayfield [Java Design] préconisent la stratégie suivante :

1. *Rechercher la caractéristique polymorphe*

   Le jeu de caractères ? Possible si on étend ceux-ci au jeu UNICODE.

   La représentation des valeurs ordinales de chacun des caractères affichés ? Certainement, c'est déjà ce que réalise le programme.

   On retiendra donc cette dernière caractéristique : *Représentation des nombres.*

2. *Identifier un ensemble de noms de méthodes correspondant à cette caractéristique*

   L'objectif étant de représenter une valeur ordinales, un entier, dans une base donnée constituée de symboles, eux-même représentés sous la forme d'un caractère. Une suite ordonnée de caractères est un type bien connu (String), nous proposons de nommer l'opération :

   ```
   String toString(int n)
   ```

   Une fonction dont la valeur (une chaîne de caractère) est la représention du nombre (n) qu'elle recoit en argument.

3. *Ajouter une interface*

   ```
   interface Representation {
     String toString(int n);
   }
   ```

4. *Identifier les implémentations*

   Concevons les deux classes d'implémentation de l'interface *Representation* qui réalisent la représentation en base 10 et en base 2, conformément à l'existant. Rappel, les fonctionalités qui existent avant une activité de refactoring, doivent absolument être retrouvées après la refonte du code.

   ```
   class Decimal implements Representation {
    public String toString(int i) {
        return String.valueOf(i);
    }
   }
   ```

```
class Binaire implements Representation {
  public String toString(int i) {
     return Integer.toBinaryString(i);
  }
}
```

Modifions la partie qui décide de la représentation des nombres à appliquer (limitée actuellement à seulement deux représentations possibles) en la couplant à un objet, nommé *repr*, de type *Représentation*.

L'ancien code :

```
    System.out.println("<table border=\"1\"> <tr>");
    System.out.println("<td bgcolor=\"teal\" align=\"center\">");
    switch (this.typeRepr) {
      case 'd' :
         System.out.println(toDecString(i));
         break;
      case 'b' :
         System.out.println(toBinString(i));
         break;
      default :
         System.out.println(toDecString(i));
    }
    System.out.println("</td></tr><tr>");
    System.out.println("<td bgcolor=\"#CC3300\" align=\"center\">");
    System.out.println(" &#" + i + "; </td></tr></table></td>");
```

Le nouveau code :

```
    System.out.println("<table border=\"1\"> <tr>");
    System.out.println("<td bgcolor=\"teal\" align=\"center\">");

    System.out.println(repr.toString(i));

    System.out.println("</td></tr><tr>");
    System.out.println("<td bgcolor=\"#CC3300\" align=\"center\">");
    System.out.println(" &#" + i + "; </td></tr></table></td>");
```

L'objet responsable de la représentation des valeurs ordinales est fournit par l'appelant à la création de l'application.

```
  public TableAsciiToHTML(Representation repr) {
      this.repr = repr;
      printHTML();
  }
```

# Listing de la solution

```
interface Representation {
  String toString(int i);
}
```

```
class Decimal implements Representation {
  public String toString(int i) {
      return String.valueOf(i);
  }
}

class Binaire implements Representation {
  public String toString(int i) {
     return Integer.toBinaryString(i);
  }
}

class TableAsciiToHTML {
  private Representation repr;

  public TableAsciiToHTML(Representation repr) {
      this.repr = repr;
      printHTML();
  }

  private void printHTML() {
    int deb = 32;
    int fin = 128;
    int nbCol = 10;
    int cpt = 0;
    System.out.println("<html><head /><body><center><h1>TABLE DE CARACTERES<h1>");
    System.out.println("<table border=\"1\">");
    System.out.println("<tr>");
    for (int i = deb; i < fin; i++, cpt++) {
      if (cpt%nbCol == 0) {
         if (i>deb)  System.out.println("</tr>");
         System.out.println("<tr>");
      }
      System.out.println("<td align=\"center\"");
      System.out.println("<table border=\"1\"> <tr>");
      System.out.println("<td bgcolor=\"teal\" align=\"center\">");

      System.out.println(repr.toString(i));

      System.out.println("</td></tr><tr>");
      System.out.println("<td bgcolor=\"#CC3300\" align=\"center\">");
      System.out.println(" &#" + i + "; </td></tr></table></td>");
    }
    System.out.println("</table></center></body></html>");
  }

}

class AppTableAscii {
  public static void main(String[] args) {
     String arg = (args.length>0) ? args[0] : "Decimal";
     try {
         Class c = Class.forName(arg);
         Representation repr = (Representation) c.newInstance();
         TableAsciiToHTML app = new TableAsciiToHTML(repr);
     }
     catch (ClassNotFoundException e) {
       System.out.println("Erreur : " + arg + " n'est pas une classe implémentée.");
     }
     catch (InstantiationException e) {
       System.out.println("Erreur : " + arg + " n'est pas n'est pas du type attendu.");
     }
     catch (IllegalAccessException e) {
       System.out.println("Erreur : " + arg + " n'est pas accessible.");
     }
  }
}
```

# Test

Nous allons maintenant tester la qualité *Ouvert-Fermé*, due au respect d'*OCP* (POF Principe

d'Ouverture/Fermeture).

Créons une nouvelle classe d'implémentation de *Representation*.

```
public class Hexadecimal implements Representation {
  public String toString(int n) {
    return Integer.toHexString(n);
  }
}
```

Après compilation, nous exécutons le programme en lui passant en argument le nom de cette nouvelle classe.

```
  java AppTableAscii Hexadecimal > res.html
```

Le tour est joué.

# Conclusion

Nous venons d'étendre le comportement de l'application sans intervenir sur son code.

Nous avons pour cela respecté OCP (POF) et appliqué DIP (PID).

### Fichier de configuration XML

Notez que la fonction d'instanciation "paramétrée" est très souvent déléguée à une classe spécialisée (*factory*). Celle-ci puise très souvent les informations dont elle a besoin dans un (ou plusieurs) fichiers de configuration XML. Exemple (extrait d'un fichier de configuration d'une application Struts):

```
<form-beans>
  <form-bean
     name="addQuestionForm"
     type="org.reseaucerta.qcm.presentation.AddQuestionForm"/>
     ...
</form-beans>

<!--  Action Mapping Definitions  -->
<action-mappings>
   <action path="/addQuestion"
     type="org.reseaucerta.qcm.application.AddQuestionAction"
     name="addQuestionForm"
     scope="session"
     validate="true"
     input="/jsp/addQuestion.jsp">
     <forward
        name="success"
        path="/jsp/confirmAddQuestion.jsp"/>
     <forward
        name="echec"
        path="/jsp/echecAddQuestion.jsp"/>
   </action>
   ...
<action-mappings>
```

# Initiation au design pattern Factory

## Introduction

Il existe une relation étroite entre les *design patterns* (modèles de conception) et les principes de conception objet.

Qu'est-ce qu'un *design pattern* ?

Un design pattern est une *description d'une solution logicielle réutilisable face à un problème récurrent en développement informatique.* (Mark Grand in *Patterns in Java vol. 1*).

L'origine : les modèles de construction architecturale par Christopher Alexander [1977].

Patterns logiciels : Kent Beck [1980] et Ward Cunningham [1987 et 1994].

*Typologie des Patterns*

- Patterns d'analyse : méthodes pour faire une bonne analyse (Fowler).
- Patterns de conception : solutions standard de conception (gof).
- Patterns d'implémentation : façon de programmer un problème dans un langage particulier.

Nous nous interessons aux patterns de conception (*design patterns*).

Ouvrage de référence :

- *design patterns* de Erich Gamma, Richard Heml, Ralph Johnson et John Vlissides. Ouvrage connu sous le nom de Gof (*gang of four*) et disponible en français aux éditions Vuibert.

Les design patterns offrent de nombreux avantages :

- Capturent l'expérience de développeurs, d'ingénieurs, d'experts.
- Permettent à n'importe quel développeur de réutiliser un savoir-faire (ne pas réinventer la roue).
- Donnent un nom à des éléments d'architecture (enrichissement du vocabulaire pour une meilleure communication).

Les design patterns sont rangés dans des catalogues selon deux critères : le rôle (créateur, structurel, comportemantal) et le domaine (classe -statique- et objet -dynamique-) [Gof].

## Figure 14. Catalogue GOF

**espaces des modèles de conception selon 'design Patterns'**

| | | rôle | | |
|---|---|---|---|---|
| | | créateur | structurel | comportement |
| Domaine | classe | Fabrication | Adaptateur(classe) | Interprète<br>Patron de méthodes |
| | objet | Fabrique abstraite<br>Monteur<br>Prototype<br>Singleton | Adaptateur(objet)<br>Pont<br>Composite<br>Décorateur<br>Façade<br>Poids mouche<br>Procuration(proxy) | Chaîne de responsabilité<br>Commande<br>Itérateur<br>Médiateur<br>Mémento<br>Observateur<br>Etat<br>Stratégie<br>Visiteur |

D'autres catalogues sont proposés, notamment GRASP (*General Responsability Assignement Software*

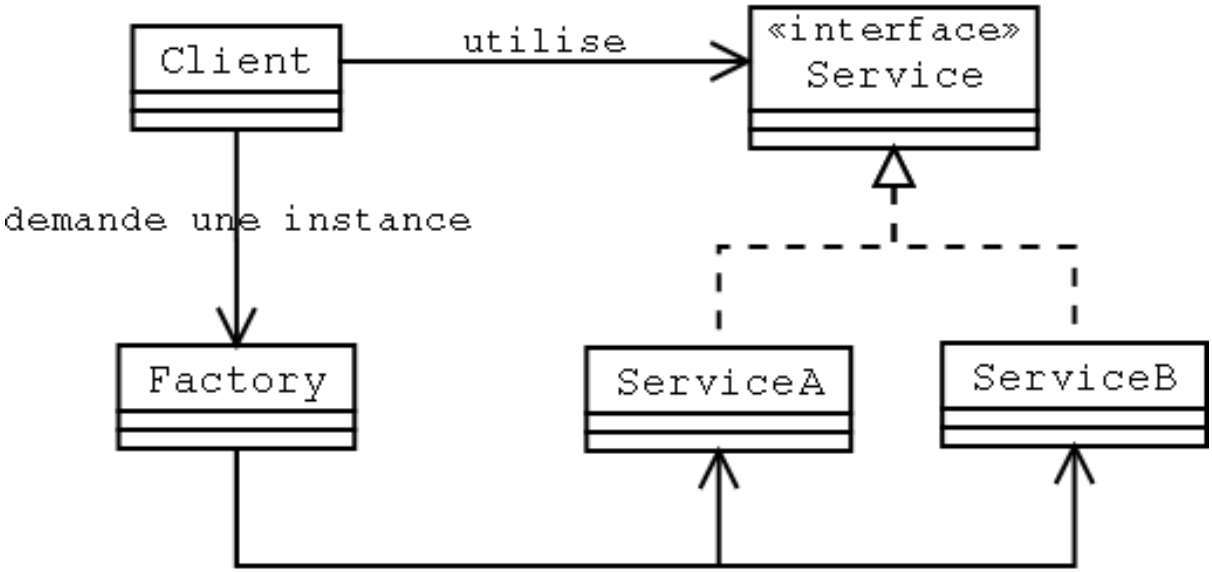
), ou patterns généraux d'affectation des responsabilités. Ces patterns décrivent quelques principes fondamentaux en conception objet (Expert, Créateur, Faible couplage, Forte cohésion, Contrôleur).

# Factory (Fabrique)

Principe directement concerné : DIP

La mise en oeuvre du couplage abstrait, que préconise DIP, nécessite toute fois un mécanisme d'instanciation afin de lier concrètement les classes, à un moment donné. C'est le rôle des patterns créateurs, en particulier ceux de type *Factory*.

**Figure 15. Pattern Factory**




Les solutions les plus connues sont **méthode de fabrique** (*method factory*) et *fabrique abstraite* (*abstract factory*).

La méthode de fabrique se charge de construire une instance, par exemple en fonction d'un discriminant reçu en argument.

La fabrique abstraite utilise l'héritage (et le polymorphisme) comme discriminant. Un système très souple qui permet à un client de choisir son fournisseur de classes concrètes.

Exemple 1

```
class RepresentationFactory {
  static Representation getInstance(char typeRepr) {
     Representation repr;
     switch (typeRepr) {
          case 'd' :
             repr = new Decimal();
             break;
          case 'b' :
             repr = new Binaire();
             break;
           default :
             repr = new Decimal();
     }
     return repr;
  }
}
```

Exemple 2

```
interface RepresentationFactory {
  public Representation getInstance();
}

class ReprBinaire implements RepresentationFactory {
  public Representation getInstance(){
     return new Binaire();
  }
}

class ReprDec implements RepresentationFactory {
  public Representation getInstance(){
     return new Decimal();
  }
}
```

- *Exercices*

  Appliquer le pattern Factory à l'exercice TableAscii-HTML.

# Travaux pratiques

Sujet d'après un exemple présenté par Tony Sintes sur JavaWorld.com (2002).

Considérons le besoin suivant : On souhaite offrir aux programmes écrits en java la possibilité de «tracer» des mesages de debogage et d'erreur soit dans un fichier soit sur la console, et ceci de manière transparente.

*Listing 1*

```
public interface Trace {

       // placer le debogage à on ou off
       public void setDebug( boolean debug );

       // ecrire un message de debug
       public void debug( String message );

       // ecrire un message d'erreur
       public void error( String message );

}
```

Supposons que nous ayons écrit deux implementations. Une implémentation (Listing 2) écrit les messages sur la console, tandis que l'autre (Listing 3) les écrit dans un fichier.

*Listing 2*

```
public class FileTrace implements Trace {

       private java.io.PrintWriter pw;
       private boolean debug;

       public FileTrace() throws java.io.IOException {
              // dans une version réelle, FileTrace aurait besoin
              // d'obtenir d'une manière ou d'une autre le nom du fichier
              // pour cet exemple, il sera codé en dur
              pw = new java.io.PrintWriter( new java.io.FileWriter( "c:\trace.log" ) );
       }

       public void setDebug( boolean debug ) {
              this.debug = debug;
       }
```

```
	public void debug( String message ) {
		if( debug ) {  // imprimer seulement si debug est true
			pw.println( "DEBUG: " + message );
			pw.flush();
		}
	}
	public void error( String message ) {
		// toujours imprimer les erreurs
		pw.println( "ERREUR: " + message );
		pw.flush();
	}
}
```

*Listing 3*

```
public class SystemTrace implements Trace {

	private boolean debug;

	public void setDebug( boolean debug ) {
	    this.debug = debug;
	}

	public void debug( String message ) {
	    if( debug ) {  // imprimer uniquement si debug est true
	         System.out.println( "DEBUG: " + message );
	    }
	}
	public void error( String message ) {
	    System.out.println( "ERREUR: " + message );
	}
}
```

Pour utiliser une de ces classes, nous nous y prendions comme cela :

*Listing 4*

```
class Test {
  public void run() {
    int x = 2;
    SystemTrace log = new SystemTrace();
    log.debug( "debut du log" );
    try {
      int x = 1/(x-2);
    }
    catch (Exception e) {
       log.error(e.getMessage());
    }
    log.debug("Valeur de x : " + x);
  }
  ...
}
```

On souhaite pouvoir changer de politique de trace (console, fichier ou autres) sans toucher au code des applications utilisant les services de trace.

Proposez une solution.

# Exemple de solution

## Introduction

Nous devons découpler les programmes utilisateurs des fonctions de trace et les classes implémentant les

services de Trace.

Nous respectons ainsi le principe ennoncé dans [Gof] : *Programmer pour une interface et non pour une implémentation.*

*Remarque : Dans la version française de [Gof], le mot «développement» a été préféré (?) à «implementation».*

# Listing 1

Dans la version proposée par l'auteur, les programmes clients délèguent entièrement le choix de la classe d'implémentation de Trace à une Factory.

```
//... some code ...
Trace log = traceFactory.getTrace();
//... code ...
log.debug( "entering loog" );
// ... etc ...
```

Bien entendu, afin de gagner en souplesse, la *factory* devra initialement être obtenu au moyen du design pattern *Abstract Factory* :

```
interface TraceFactory {
   public  Trace getTrace();
}
```

# Listing 3

Version initiale : les traces sont réalisées sur la console.

```
public class TraceConsoleFactory implements TraceFactory {
      public Trace getTrace() {
            return new SystemTrace();
      }
}
```

Variante (*sans intervenir sur les programmes client*) : les traces sont réalisées dans un fichier, toute fois, si cela s'avère impossible, les traces se feront sur la console.

```
public class TraceFileFactory implements TraceFactory {
   public Trace getTrace() {
     try {
        return new FileTrace();
     } catch ( java.io.IOException ex ) {
        Trace t = new SystemTrace();
        t.error( "could not instantiate FileTrace: " + ex.getMessage() );
        return t;
     }
  }
}
```

Dès lors nous pouvons imaginer une classe :

```
public class AbstracTraceFactory {

   public static TraceFactory getTraceFactory()
   throws CreateTraceFactoryException {
     try {
        // recherche dans un fichier de configuration
        // la factory à instancier
        // ...
        // par exemple :
```

```
            return new FileTraceFactory();

        } catch ( Exception ex ) {
            throw new CreateTraceFactoryException(ex);
        }

    }

}
```

# Conclusion

Nous venons de présenter le lien qu'il existe entre des principes de conception et programmation objet et les designs patterns sur un exemple mettant en oeuvre DIP (le principe) et Factory (le pattern).

Le domaine d'application et d'étude des modèles de conception est vaste, et en continuelle évolution.

N'hésitez pas à investir ce sujet (livres et articles sur le web), et à faire un parallèle avec les principes objet sous-jacents. Vous trouverez dans le livre "Design patterns par la pratique" (en français), des auteurs A.Shalloway et J.Trott, une présentation et des exemples d'applications des modèles de conception courants.