

Programmation Orientée Objet 2/2

prérequis : l'encapsulation : la notion d'attribut, méthode et constructeur

Les concepts de base de la programmation objet sont généralement résumés par quelques concepts : Objet, Classe, Abstraction, Polymorphisme, Principes de Conception.

Nous avons présenté les notions Objet, Classe et Interface, nous poursuivons ici avec l'Abstraction, le polymorphisme et des principes de conception.

Quelques principes fondamentaux en conception

Il y a 2 grandes règles (formulée ici par Kent Beck) que tout développeur devrait connaître :

1. Le système (code et tests pris dans leur ensemble) doit **communiquer** tout ce que vous avez l'intention de communiquer. Ce principe a donné lieu à une pratique de conception appelée « **ubiquitous language** » (Erci Evans 2003 – DDD) <http://institut-agile.fr/ubiquitous.html>
2. Le système ne doit contenir **aucune duplication de code**.

=> Ces deux contraintes constituent la règle : **Une Fois et Une Seule**.

A cela on ajoute :

3. Le système doit contenir un nombre minimal de classes.
4. Le système doit contenir un nombre minimal de méthodes.

En respectant ces contraintes, le développeur produit du **code clair et simple** : les classes ne sont pas chargées de responsabilité et les méthodes courtes (selon différents points de vue : <= 5 lignes, <= 10 lignes ou qu'importe si le contenu est clairement lisible et ne contient pas de duplication).

Exceptions courantes : implémentation d'un algorithme connu, recherche d'optimisation... le tout bien commenté.

Références :

- [Kent Beck] Extreme Programming la référence, ed. CampusPress, 2002.
- [Folwer] "Refactoring" de Martin Folwer
<http://books.google.fr/books?id=1MsETFPD3I0C>
- Martin Fowler : « Any fool can write code that a computer can understand. Good programmers write code that humans can understand. »
- Martin Fowler : <https://martinfowler.com/bliki/UbiquitousLanguage.html>

Mise en oeuvre d'une bonne communication

1/ Bien choisir le **nom des identificateurs** en **adéquation avec leur rôles** : variable, fonction, classe, constante...

Exemple avec une variable :

Éviter :

```
int var1;
```

Préférer :

```
int nbColis;
```

Ainsi le code gagne en communication et les commentaires se font plus rares. On dit que le code est **auto-documenté** et **guidé par la logique métier**.

2/ **Ne pas surcharger en responsabilité** les classes et fonctions. (faire une chose et le faire bien)

Une fonction ne devrait pas déborder de sa responsabilité. Par exemple une fonction qui détermine le login d'une personne sur la base de son nom et prénom (et de logins existants dans un annuaire...) ne devrait pas afficher un message du type « opération impossible » si elle ne peut réaliser son travail, elle devrait, si le cas est envisageable, plutôt créer une exception. En se concentrant sur un seul domaine, on renforce la **forte cohésion** de la classe.

Mise en oeuvre de la non redondance

La redondance nuit à la maintenance et augmente inutilement la taille du programme.

Un principe dédié à ce problème : principe DRY (*Don't Repeat Yourself* en anglais)

Deux principales façons de factoriser le code:

1. Placer le code dupliqué dans une fonction, puis remplacer le code dupliqué par un appel à cette dernière. (*factorisation fonctionnelle*)
2. Placer le code dupliqué dans une classe (si ce n'est déjà fait), et le réutiliser soit par héritage soit par composition. (*factorisation structurelle*)

Seul l'héritage est une technique propre à la programmation objet.

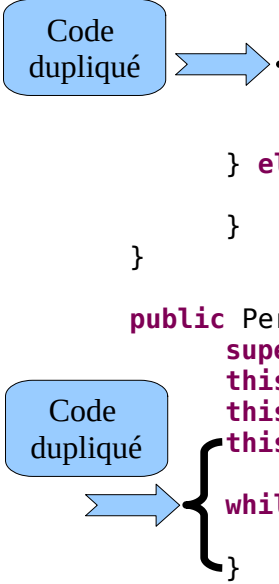
Exemple de factorisation fonctionnelle.

AVANT (attention, implémentation incorrecte – bug potentiel !)

```
public class Personne {
    private String nom;
    private String prenom;
    private String login;

    public Personne(String nom, String prenom, String login) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        if (login == null || login.isEmpty()) {
            this.login = this.prenom.substring(0, 1) +
                        this.nom.substring(0, 7);
            while (this.login.length() < 8) {
                this.login = (this.login + this.nom).substring(0, 8);
            }
        } else {
            this.login = login;
        }
    }

    public Personne(String nom, String prenom) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        this.login = this.prenom.substring(0, 1) +
                    this.nom.substring(0, 7);
        while (this.login.length() < 8) {
            this.login = (this.login + this.nom).substring(0, 8);
        }
    }
}
```

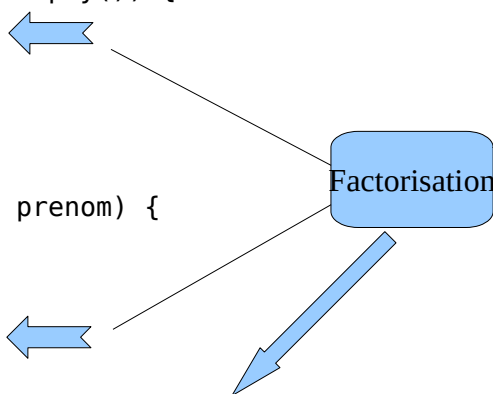


APRÈS (attention, implémentation incorrecte - bug potentiel !)

```
public Personne(String nom, String prenom, String login) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    if (login == null || login.isEmpty()) {
        makeLogin();
    } else {
        this.login = login;
    }
}

public Personne(String nom, String prenom) {
    super();
    this.nom = nom;
    this.prenom = prenom;
    makeLogin();
}

private void makeLogin() { // attention BUG !
    this.login = this.prenom.substring(0, 1) + this.nom.substring(0, 7);
    while (this.login.length() < 8) {
        this.login = (this.login + this.nom).substring(0, 8);
    }
}
```



}

Exemple de factorisation structurelle.

Une application de gestion utilisateur contient 2 classes : Utilisateur et UtilisateurUnix

AVANT

```
public class Utilisateur {
    private String nom;
    private String prenom;
    private String login;

    public Utilisateur(String nom, String prenom, String login) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        this.login = login;
    }

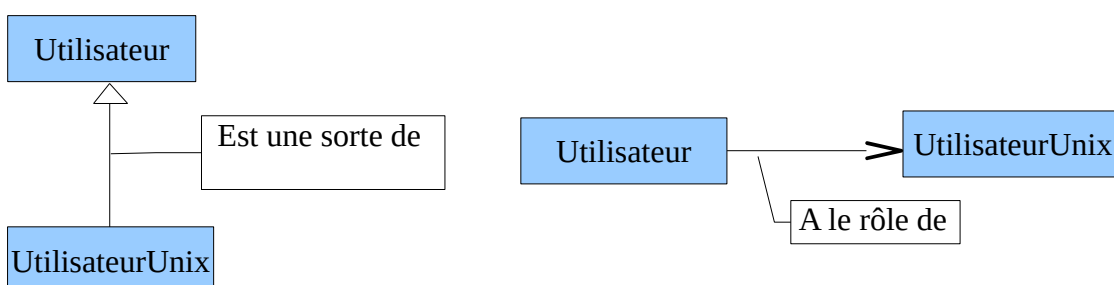
    @Override
    public String toString() {
        return this.nom + " " + this.prenom + " " + this.login;
    }
}

public class UtilisateurUnix {
    private String nom;
    private String prenom;
    private String login;
    private String shell;
    private String homeDirectory;

    public UtilisateurUnix(String nom, String prenom, String login,
        String shell, String home) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        this.login = login;
        this.shell = shell;
        this.homeDirectory = home;
    }

    @Override
    public String toString() {
        return this.nom + " " + this.prenom + " " + this.login + " shell:"
            + this.shell + " home:" + this.homeDirectory;
    }
}
```

On constate que ces deux classes ont des caractéristiques communes (attributs et comportement) que nous pouvons factoriser. Il existe pour cela 2 façons d'opérer, par héritage ou composition :



La relation « est-une sorte de » (**is-a**) est appelée relation d'**héritage**. Cette relation lie définitivement la classe enfant à sa classe mère (la classe héritée).

La relation « a le rôle de » ou « a un » (**has-a**) une relation de **composition** (une association). Cette relation a un caractère plus dynamique que l'héritage. En effet, en cours d'exécution, une instance peut changer de rôle, mais pas de classe mère.

La relation d'héritage peut être envisagée si la classe enfant (celle qui hérite) respecte l'ensemble de responsabilités (les contrats) de sa classe mère. Cette contrainte est connue sous le nom de « principe de Barbara Liskov » (du nom d'une informaticienne). Les objets de type *UtilisateurUnix* devrait être 100% compatibles avec ceux de type *Utilisateur*. C'est **conceptuellement le cas ici**, nous pouvons donc appliquer cette technique.

EXEMPLE DE TRANSFORMATION PAR HÉRITAGE

```
public class Utilisateur {
    private String nom;
    private String prenom;
    private String login;

    public Utilisateur(String nom, String prenom, String login) {
        super();
        this.nom = nom;
        this.prenom = prenom;
        this.login = login;
    }

    @Override
    public String toString() {
        return this.nom + " " + this.prenom + " " + this.login;
    }
}

public class UtilisateurUnix extends Utilisateur{
    private String shell;
    private String homeDirectory;

    public UtilisateurUnix(String nom, String prenom, String login,
        String shell, String home) {
        super(nom, prenom, login);
        this.shell = shell;
        this.homeDirectory = home;
    }

    @Override
    public String toString() {
        return super.toString()
            + " shell:" + this.shell + " home:" + this.homeDirectory;
    }
}
```

Héritage

Appel du constructeur hérité

Appel d'une méthode héritée

La classe *UtilisateurUnix* est redéfinie sur la base de la classe *Utilisateur*. On évite ainsi les redondances de déclaration car *UtilisateurUnix* hérite de l'implémentation de *Utilisateur*.

Bien que la méthode *toString* porte le même nom dans les deux classes, mais elle se comporte différemment dans la classe *Utilisateur* et *UtilisateurUnix*. En effet, comme on peut le constater dans le corps de *toString*, la classe *UtilisateurUnix* profite de (fait un appel à) l'implémentation de la méthode *toString* de sa classe parent (ou sur-classe) via le mot clé **super**.

On remarquera la différence de syntaxe de l'utilisation de **super** entre un constructeur (de plus **super** doit être la première instruction) et une méthode où *super* est utilisé en notation pointée et désigne une méthode directement héritée.

EXEMPLE DE TRANSFORMATION PAR COMPOSITION

La relation de composition va souvent de pair avec la mise en place d'interfaces, ce qui permet d'étendre le type d'une classe (pour ne pas se limiter à la classe *Object*).

Dans l'exemple qui suit, le « type » de *Utilisateur* est caractérisé par un lien de composition (délégation).

```
interface SysShell {  
    public String getShell();  
    public String getHomeDirectory();  
}
```

Pour parfaire la stratégie de composition, la technique consiste à abstraire la partie commune

```
public class Utilisateur {  
    private String nom;  
    private String prenom;  
    private String login;  
    private SysShell sysShell;
```

```
    public Utilisateur(String nom, String prenom, String login, SysShell sysShell) {  
        super();  
        this.nom = nom;  
        this.prenom = prenom;  
        this.login = login;  
        this.sysShell = sysShell;  
        // sysShell référence un objet qui implémente l'interface SysShell
```

l'objet référencé par sysShell implémente les opérations déclarées par l'interface SysShell

```
    }  
    @Override  
    public String getNom() { return this.nom; }  
    [...]  
    + getter/setter sur sysShell  
    @Override  
    public String toString() {  
        return this.nom + " " + this.prenom + " " +  
            this.login + " " + this.sysShell ? this.sysShell : "";  
    }  
}
```

La technique de composition a un caractère plus dynamique. Le lien de composition est réalisé lors de l'instanciation (ici dans le constructeur – c'est à dire en phase d'initialisation de l'objet). Ce lien est sous la responsabilité du développeur et peut-être sous-traité à un sous-système comme avec le framework Spring, et **modifié dynamiquement, ce que l'héritage ne permet pas.**

Résumé

Nous avons vu deux techniques structurelles d'élimination de la redondance : mise en place d'une hiérarchie d'héritage (relation parent – enfant) et mise en place d'un graphe de composition (délégation de rôle à des objets) via une interface commune.

Parfois les deux techniques se combinent : délégation de rôles à des objets de type abstrait comportant des implémentations.

Nous présentons maintenant la notion de type abstrait.

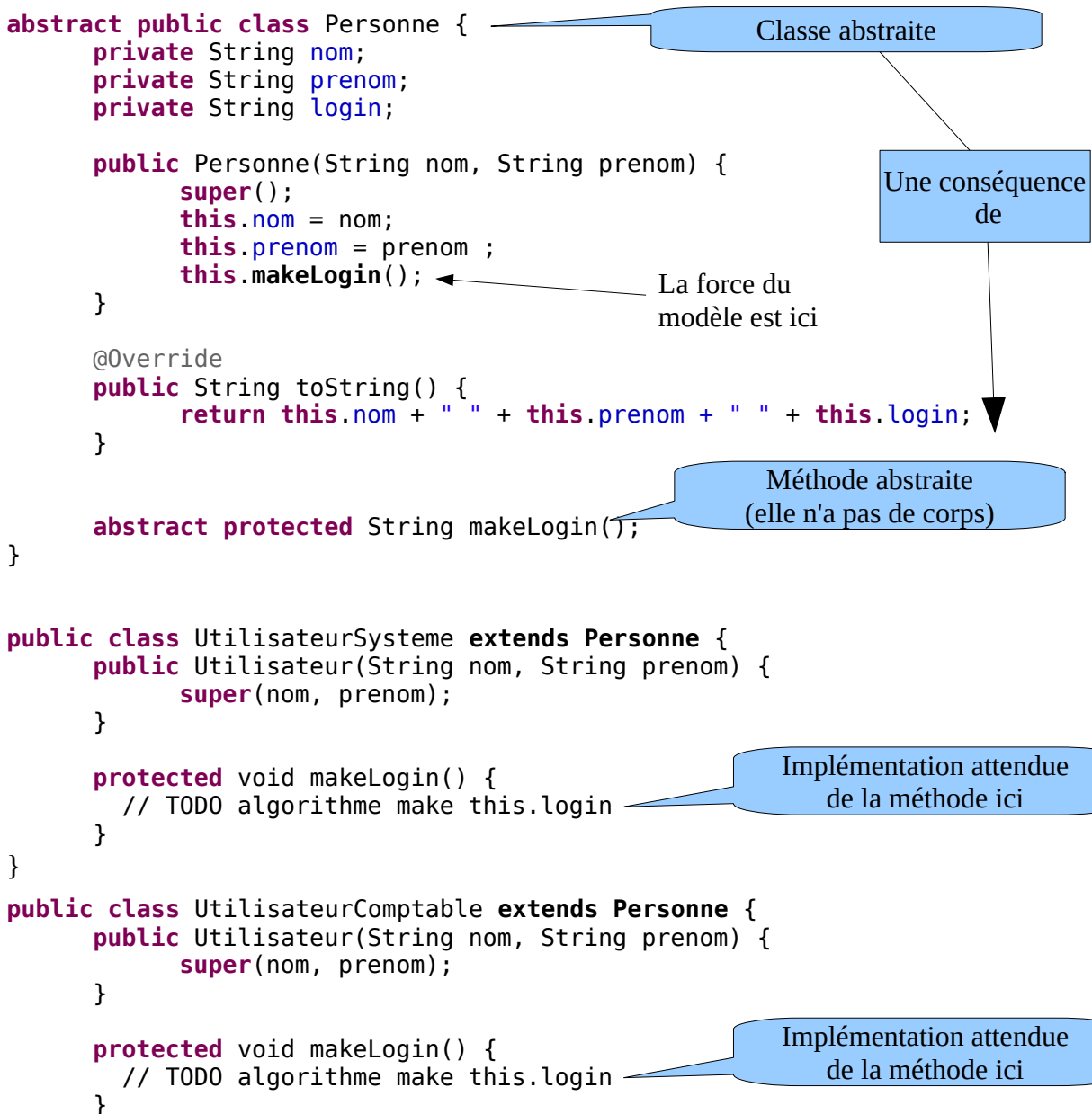
Interface et classe abstraite (rappel)

Une interface est une structure définissant un type. Elle est peut être vide ou composée de déclarations d'opération (et de constantes statiques). Voir par exemple l'interface [Shape](#) ou [List](#) en Java.

Ces structures ne peuvent pas être instanciées car elle sont incomplètement définies : les méthodes n'ont pas de corps (on parle alors d'opérations). Une interface est une abstraction. Depuis Java8, les interface peuvent déclarer des opérations avec une implémentation par défaut.

Dans le but toujours de factoriser du code, il peut arriver que l'on soit amené à placer de l'implémentation (des attributs) dans ce type de structure, or cela ne peut se faire que dans une classe. Ces classes là sont souvent incomplètes, car certaines méthodes ne sont pas implémentées. Ce type de classe, non destiné à l'instanciation directe, est qualifié d'**abstrait** (*abstract*). On parle alors de classes abstraites.

Exemple :



}

EXERCICES

Vous intervenez sur un projet qui utilise la classe Point (voir annexe A).

Voici un exemple d'utilisation.

```
1  public static void main(String[] args) {  
2      Point pA = new Point();  
3      Point pB = new Point();  
4      pA.setRectangular(10, 10);  
5      if (UtilPoint.equalsXY(pA, pB))  
6          System.out.println("les points sont égaux en X et Y.");  
7      else  
8          System.out.println("les points sont différents en X et Y.");  
11     System.out.println(pA);  
10 }
```

1. L'instruction en ligne 4 peut être évitée. Prouvez-le !
2. Entre la ligne 4 et 5, ajouter une instruction qui augmente la position en X du point pB de 20 unités, **sans** utiliser la méthode setRectangular.
3. Suite à votre solution, à l'exécution, quelle sera la sortie produite ?

Certaines parties du projet nécessitent de travailler avec des points en trois dimensions (x, y et z pour la profondeur).

1. Concevoir une classe répondant à ce problème, et qui soit compatible avec les classes du projet utilisant la classe Point (comme dans l'exemple d'utilisation)
2. Que faudrait-il changer au programme exemple pour qu'il travaille avec des instances de la nouvelle classe ?

Annexe A : Classes Point et UtilPoint

```
/*  
Copyright (c) Xerox Corporation 1998-2002. All rights reserved.
```

Use and copying of this software and preparation of derivative works based upon this software are permitted. Any distribution of this software or derivative works must comply with all applicable United States export control laws.

This software is made available AS IS, and Xerox Corporation makes no warranty about the software, its performance or its conformity to any specification.

```
*/  
package bean;  
  
class Point {  
  
    protected int x;  
    protected int y;  
  
    /**  
     * Constructor : Initializes instance  
     */  
    public Point() {  
        this.x = 0;  
        this.y = 0;  
    }  
  
    /**  
     * Constructor : Initializes instance from given x,y  
     */  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    /**  
     * Return the X coordinate  
     */  
    public int getX(){  
        return x;  
    }  
  
    /**  
     * Return the y coordinate  
     */  
    public int getY(){  
        return y;  
    }  
  
    /**  
     * Set the x and y coordinates  
     */  
    public void setRectangular(int newX, int newY){  
        setX(newX);  
        setY(newY);  
    }  
}
```

```

/**
 * Set the X coordinate
 */
public void setX(int newX) { x = newX; }

/**
 * set the y coordinate
 */
public void setY(int newY) {
    y = newY;
}

/**
 * Move the point by the specified x and y offset
 */
public void offset(int deltaX, int deltaY){
    setRectangular(x + deltaX, y + deltaY);
}

/**
 * Make a string of this
 */
public String toString(){
    return this.getClass().getName() + " : (" + getX() + ", " + getY() + ")";
}
}

```

Classe utilitaire

```

package bean;
public class UtilPoint {
    /**
     * teste l'égalité en X de 2 points
     * @param p1 le premier point
     * @param p2 le deuxième point
     * @return vrai si p1 et p2 ont même valeur de X, faux sinon
     */
    static public boolean equalsX(Point p1, Point p2) {
        return p1.getX() == p2.getX();
    }

    /**
     * teste l'égalité en Y de 2 points
     * @param p1 le premier point
     * @param p2 le deuxième point
     * @return vrai si p1 et p2 ont même valeur de Y, faux sinon
     */
    static public boolean equalsY(Point p1, Point p2) {
        return p1.getY() == p2.getY();
    }

    /**
     * teste l'égalité en X et Y de 2 points
     * @param p1 le premier point
     * @param p2 le deuxième point
     * @return vrai si p1 et p2 ont même valeur de X et Y, faux sinon
     */
    static public boolean equalsXY(Point pa, Point pb) {
        return UtilPoint.equalsX(pa, pb) && UtilPoint.equalsY(pa, pb);
    }
}

```