

Table of Contents

Notion de bases liées à la POO.....	1
Principes initiaux	2
Quelques principes fondamentaux en conception	2
Bonne communication	3
Mise en oeuvre d'une bonne communication.....	3
Forte cohésion.....	3
Limiter la redondance.....	4
Factorisation fonctionnelle	4
Factorisation structurelle.....	5

Notion de bases liées à la POO

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.

Principes initiaux

Quelques principes fondamentaux en conception

Il y a 2 grandes règles (formulées 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. Les classes doivent 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 (on parle de **forte cohésion**) 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>

Bonne communication

Mise en oeuvre d'une bonne communication

Bien nommer

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.

Forte cohésion

Consiste à 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 (existants dans une base de données, 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 logique de la classe.

Limiter la 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)

Il existe deux façons de factoriser le code:

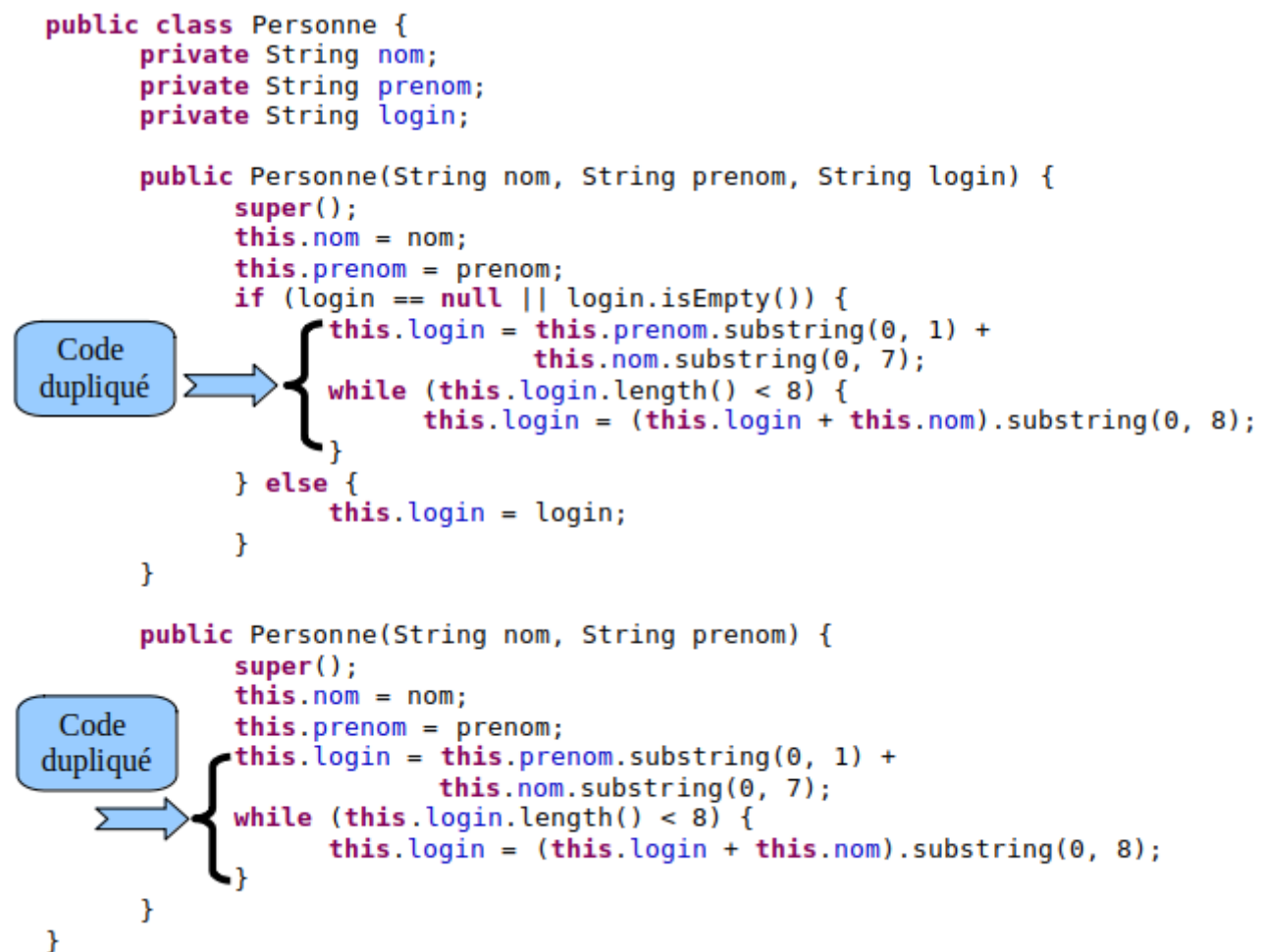
1. Placer le code dupliqué dans une fonction, puis remplacer le code dupliqué par un appel à cette dernière. (factorisation fonctionnelle)
2. Placer le code dupliqué dans une classe (si ce n'est déjà fait), et le réutiliser soit par héritage soit par composition. (factorisation structurelle)

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

Factorisation fonctionnelle

Cette technique consiste à **factoriser** le code redondant dans une **fonction**.

AVANT (attention, implémentation incorrecte – bug potentiel !)



APRÈS (attention, implémentation incorrecte - bug potentiel !)

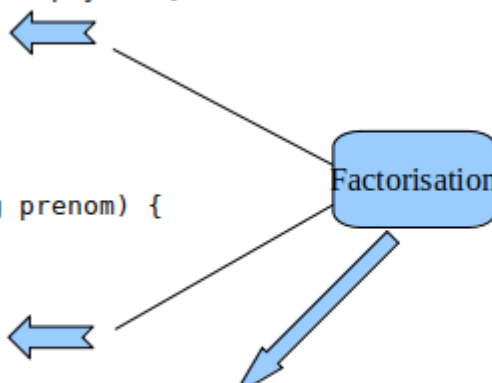
```

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);
    }
}

```



The diagram illustrates the concept of factorization. A blue rounded rectangle labeled 'Factorisation' has three arrows pointing to specific lines of code in the provided Java snippet. One arrow points to the makeLogin(); call inside the first constructor. Another arrow points to the makeLogin(); call inside the second constructor. A third, thicker arrow points to the makeLogin() method definition at the bottom of the code block.



On remarquera que la fonction créée pour des raisons interne n'est pas publique.

Factorisation structurelle

Cette technique consiste à factoriser par conception de nouvelles classes (et interfaces)

Comme base d'exemple prenons une application de gestion d'utilisateurs contenant 2 classes : `Utilisateur` et `UtilisateurUnix`

```

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;
    }
}

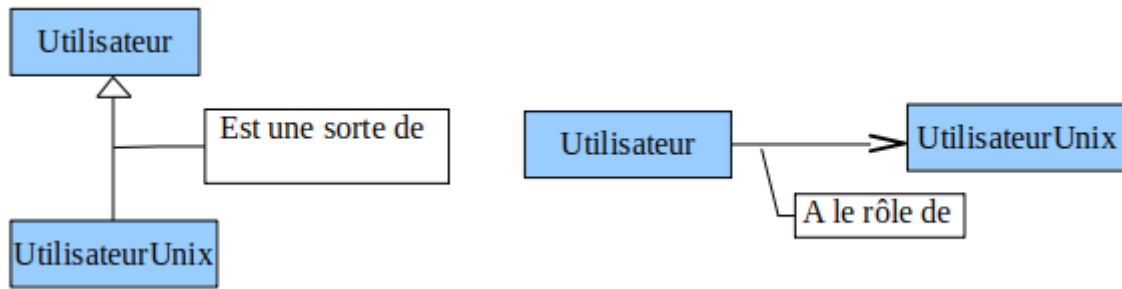
public class UtilisateurUnix {
    private String nom;
    private String prenom;
    private String login;
    private String shell;
    private String homeDirectory;

    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*) est 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 devraient être 100% compatibles avec ceux de type Utilisateur. C'est conceptuellement le cas ici, nous pouvons donc appliquer cette technique.

Transformation par héritage

Cette technique consiste à faire porter les propriétés communes à une classe de base (**Utilisateur**).


```

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.

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 une association, un lien de composition (délégation).

```

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

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;
    }
    @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: "";
    }
}

```

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

l'objet référencé par sysShell implémente les opérations déclarées par l'interface 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, dans la plupart du temps).

Ce lien est sous la responsabilité du développeur et peut sous-traiter l'instanciation à un sous-système qui effectuera alors lui-même l'**injection** de l'objet souhaité. Une technique (inversion de dépendance) supportée par nombre de frameworks comme **Spring**, **Symfony**, **Angular** par exemple.

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.

Références

- [Principe d'Inversion de Dépendance](#)
- [Principe de substitution de Barbara Liskov](#)
- Principes de conception : [SOLID](#)