Conception et Programmation Objet Avancées
Quelques principes : SOLID

Petru Valicov
petru.valicov@umontpellier.fr

https://github.com/IUTInfoMontp-M3105

2019-2020

PO avancée - objectifs

- Diminuer le **couplage** (l'inter-dépendance entre les objets et les modules d'une application en général)
- Favoriser l'**encapsulation**
- Améliorer la **cohésion** – chaque classe doit effectuer des tâches liées sémantiquement entre elles
- **Simplifier** le code :
  - éviter la duplication
  - des petites classes avec des méthodes claires
  - KISS
- Lutter contre les *"code smells"*

PO avancé : SOLID

Cinq principes de base à appliquer au développement objet :

Single Responsibility Principle (**SRP**) – une et une seule raison pour changer

Open/Closed Principle (**OCP**) – étendre sans modifier

Liskov Substitution Principle (**LSP**) – respect du contrat des parents

Interface Segregation Principle (**ISP**) – granularité

Dependency Inversion Principle (**DIP**) – ne pas dépendre des implémentations concrètes

Pour plus de détails :
Agile Software Development, Principles, Patterns, and Practices - R.C. Martin, 2002.

Responsabilité unique (SRP)

Qu'en pensez-vous ?

```java
public interface Message{
    public void setDestinataire(String destinataire);
    public void setEmetteur(String emetteur);
    public void setMessage(String message);
    public boolean envoyerMessage();
}

public class Email implements Message {
    public void setDestinataire(String destinataire) { /* corps */ }
    public void setEmetteur(String emetteur) { /* corps */ }
    public void setMessage(String message) { /* corps */ }
    public boolean envoyerMessage() { /* corps */ }
}

public class Sms implements Message {
    public void setDestinataire(String destinataire) { /* corps */ }
    public void setEmetteur(String emetteur) { /* corps */ }
    public void setMessage(String message) { /* corps */ }
    public boolean envoyerMessage() { /* corps */ }
}
```

Responsabilité unique (SRP)

Et maintenant ?

```
public interface Message{
    public void setDestinataire(String destinataire);
    public void setEmetteur(String emetteur);
    public void setMessage(String message);
}

public class Email implements Message {
    public void setDestinataire(String destinataire) { /* corps */ }
    public void setEmetteur(String emetteur) { /* corps */ }
    public void setMessage(String message) { /* corps */ }
}

public class Sms implements Message {
    public void setDestinataire(String destinataire) { /* corps */ }
    public void setEmetteur(String emetteur) { /* corps */ }
    public void setMessage(String message) { /* corps */ }
}

public interface Serveur {
    public boolean envoyerMessage(Message message)
}
```


Responsabilité unique (SRP)

" Si une classe a plus d'une responsabilité, alors ces responsabilités deviennent couplées. Des modifications apportées à l'une des responsabilités peuvent porter atteinte ou inhiber la capacité de la classe de remplir les autres. Ce genre de couplage amène à des architectures fragiles qui dysfonctionnent de façon inattendues lorsqu'elles sont modifiées."

Robert C. Martin

En gros : évitez de créer des classes ou des packages qui font trop de choses

SRP - exemple

```
public static void additionner(){
    try (BufferedReader b = new BufferedReader(new InputStreamReader(System.in)))
    {
        System.out.println("Veuillez saisir le premier opérande : ");
        int x = Integer.parseInt(b.readLine());

        System.out.println("Veuillez saisir le second opérande");
        int y = Integer.parseInt(b.readLine());

        System.out.println(x+y);
    } catch (IOException e) {
        System.out.println("Erreur de saisie du nombre");
        System.out.println("Corrigez svp");
    }
}
```

- s'il y a changement de formule d'addition ? – à priori c'est ok
- s'il y a changement de modalité de saisie ? – ça tient encore
- je veux ajouter la soustraction ! – il faut tout refaire → refactor

Amélioration ?

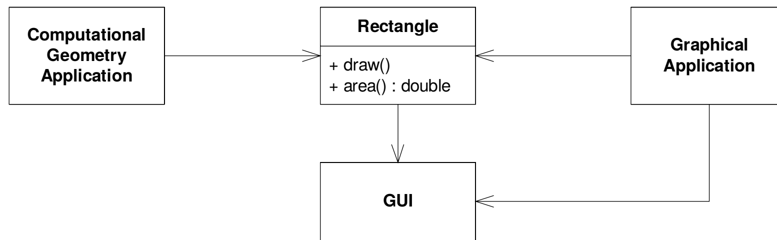
SRP - exemple

```
class Calculette {
    public static int additionner(int x, int y) {
        return x + y;
    }

    public static int soustraire(int x, int y) {
        return x - y;
    }
}
```

```
public class Imprimante {

    // les paramètres de configuration de l'imprimante

    public void imprimer(int valeur) {
        /* Préformatage de la valeur et
           impression correspondante */
    }
}
```

```
public class Client {
    public static void main(Sring[] args) {
        try (BufferedReader b = new BufferedReader(new InputStreamReader(System.in)))
        {
            System.out.println("Veuillez saisir le premier opérande : ");
            int x = Integer.parseInt(b.readLine());
            System.out.println("Veuillez saisir le second opérande");
            int y = Integer.parseInt(b.readLine());
        }
        catch (IOException e) {
            System.out.println("Erreur de saisie du nombre");
            System.out.println("Corrigez svp");
        }

        int resultat = Calculette.additionner(x, y);
        Imprimante i = new Imprimante();
        i.imprimer(resultat);
    }
}
```

SRP - autre exemple

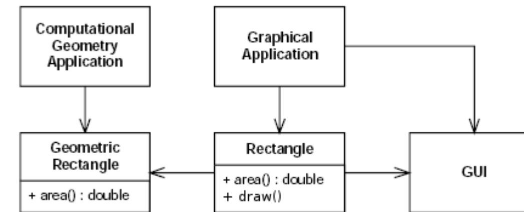


Problèmes avec ce modèle :

- Les responsabilités de calcul de l'aire et de dessin sont couplées
- Deux applications différentes par nature utilisent la classe `Rectangle`

Responsabilité unique (SRP) – exemple

Solution : séparer les responsabilités dans deux classes distinctes



La partie sur les calculs de `Rectangle` est dans `GeometricRectangle`.

La partie sur le dessin n'affecte plus la partie sur le calcul.

Responsabilité unique (SRP) - Moralité

- C'est un principe simple à exprimer mais difficile à respecter
- On a **malheureusement** souvent tendance à donner trop de responsabilités à un objet
- Analyser le code et vérifier les dépendances externes
- Essayer d'obtenir **que** des méthodes de même "nature"

Principe Ouvert/Fermé (OCP)

- Tous les systèmes **changent** (ou plutôt **évoluent**) durant leurs cycles de vies
- Les entités logicielles (classes, modules, fonctions, etc.) doivent être *ouvertes* pour l'*extension*, mais **fermées** à la **modification**

But : ajouter des nouveaux comportements sans en modifier le principe de fonctionnement interne.

Avantages :

- flexibilité par rapport à l'évolution
- diminution du couplage
- meilleure réutilisation
- meilleure maintenance

OCP – exemples

"Forcés" par le compilateur en vous interdisant de :

- changer le type/visibilité d'un attribut du programme existant
- effacer une méthode du programme existant
- changer la signature d'une méthode dans un programme existant

Un peu moins évidents :

- on ne *modifie* pas facilement le *corps* d'une méthode
- publique *vs* privé
- abstraire pour rester général

OCP – exemples

```
public class Rectangle{
  private double largeur, hauteur;

  public double getLargeur(){ ... }

  public double getHauteur(){ ... }

  public void setLargeur(){ ... }

  public void setHauteur(){ ... }
}
```

```
public class CalculetteDeGrandeurs{
  public double calculerAire(Rectangle[] formes){
    double aire = 0;
    for (Rectangle r : formes){
      aire += r.getLargeur() * r.getHauteur();
    }
    return aire;
  }
}
```

On veut étendre le calcul aux cercles :

```
public class CalculetteDeGrandeurs{
  public double calculerAire(Object[] formes){
    double aire = 0;
    for (Object r : formes){
      if (r instanceof Rectangle) //très mal, mais supposons qu'on peut l'utiliser
        aire += ((Rectangle) r).getLargeur() * ((Rectangle) r).getHauteur();
      else
        aire += ((Cercle) r).getRayon() * ((Cercle) r).getRayon() * Math.PI();
    }
    return aire;
  }
}
```

Le principe est-il respecté ? Comment faire ?

OCP – exemples

```
public class Figure {
  private int type;

  public int getType(){
    return type;
  }

  public void setType(int t){
    type = t;
  }
}
```

```
public class Cercle extends Figure {
  public Cercle(){
    super.setType(1);
  }
}
```

```
public class Rectangle extends Figure {
  public Rectangle(){
    super.setType(2);
  }
}
```

```
public class Renderer {

  public void dessinerFigure(Figure f){
    System.out.println("Début du dessin ...");
    if (f.getType() == 1)
      dessinerCercle();
    else {
      if (f.getType() == 2)
        dessinerRectangle();
      else
        System.out.println("Figure inconnue");
    }
    System.out.println("... fin du dessin");
  }

  public void dessinerCercle(){
    System.out.println("Je dessine un cercle");
  }

  public void dessinerRectangle(){
    System.out.println("Je dessine un rectangle");
  }
}
```

Si on ajoute une nouvelle figure ?

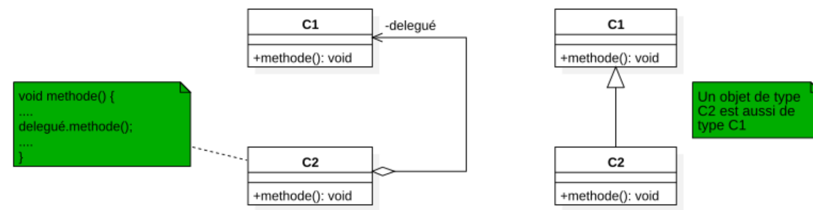
Principe de Substitution de Liskov (LSP)

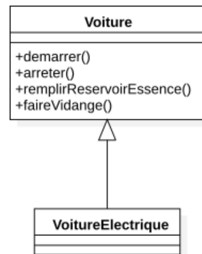
- *"Les fonctions utilisant des pointeurs ou des références vers de classes de base doivent pouvoir utiliser des objets des classes dérivées sans le savoir"* – Barbara Liskov
- Appelé communément **principe de substitution**
- Les types de bases doivent être *remplaçables* par les sous-types
- Le non-respect de ce principe $\Longrightarrow$ l'utilisateur doit connaître les détails d'implémentation des classes dérivées.
- Souvent si non-respect de LSP, alors il y a violation d'OCP.

Des exemples ?

Héritage vs Délégation

Objectif : Réutiliser du code tout en gardant la sémantique






```java
public class App {
    public static void main (String args[]) {
     Voiture v = new Voiture(...);
     v.demarrer();
     v.remplirReservoirEssence();
     v.faireVidange();

     v = new VoitureElectrique(...);
     v.demarrer();
     v.remplirReservoirEssence(); // ?????????????
     v.faireVidange(); // ?????????????
    }
}
```


LSP - illustration

```java
public class Rectangle {
    private int hauteur;
    private int largeur;

    public Rectangle(int hauteur, int largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }

    public int getHauteur() { return hauteur; }

    public int getLargeur() { return largeur; }

    public void changerHauteur(int h) {
        hauteur = h;
    }

    public void changerLargeur(int l) {
        largeur = l;
    }

    public int getPerimetre() {
        return 2 * (hauteur + largeur);
    }

    public int getSurface() {
        return hauteur * largeur;
    }
}
```

```java
public class Carre extends Rectangle {
    public Carre(int taille) {
        super(taille, taille);
    }

    public int getTaille() {
        return getHauteur();
    }

    public void changerTaille(int t) {
        super.changerHauteur(t);
        super.changerLargeur(t);
    }

    public void changerHauteur(int h) {
        changerTaille(h);
    }

    public void changerLargeur(int l) {
        changerTaille(l);
    }
}
```

```java
public class UtilisateurRectangles {
    public static void doublerRectangle(Rectangle r){
        r.changerHauteur(2 * r.getHateur());
        r.changerLargeur(2 * r.getLargeur());
    }
}
```


La délégation pour respecter LSP

```java
public class Rectangle {
    private int hauteur;
    private int largeur;

    public Rectangle(int hauteur, int largeur) {
        this.hauteur = hauteur;
        this.largeur = largeur;
    }

    public int getHauteur() { return hauteur; }

    public int getLargeur() { return largeur; }

    public void changerHauteur(int h) {
        hauteur = h;
    }

    public void changerLargeur(int l) {
        largeur = l;
    }

    public int getPerimetre() {
        return 2 * (hauteur + largeur);
    }

    public int getSurface() {
        return hauteur * largeur;
    }
}
```

```java
public class Carre {

    private Rectangle delegué;

    public Carre(int taille) {
        delegué = new Rectangle(taille, taille);
    }

    public int getTaille() {
        return delegué.getHauteur();
    }

    public void changerTaille(int t) {
        delegué.changerHauteur(t);
        delegué.changerLargeur(t);
    }

    public int getPerimetre() {
        return delegué.getPerimetre();
    }

    public int getSurface() {
        return delegué.getSurface();
    }
}
```


LSP - deuxième exemple

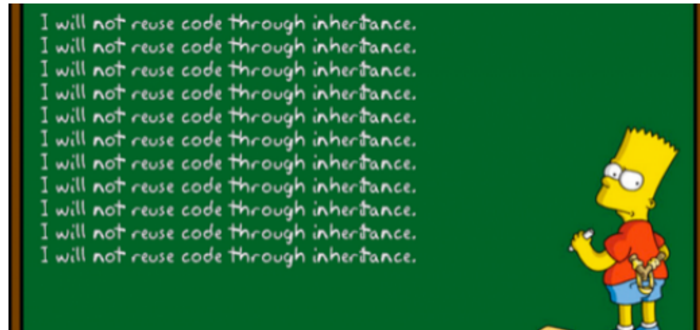
```java
public class Point {
    private double coordX;
    private double coordY;

    public Point(double coordX, double coordY) {
        this.coordX = coordX;
        this.coordY = coordY;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;

        Point p = (Point) obj;
        return p.coordX == coordX &&
                p.coordY == coordY ;
    }
}
```

```java
import java.awt.Color;

public class Test {
    public static void main(String[] args) {
        Point p1 = new Point(0,0);
        Point p2 = new PointColoré(0, 0, Color.BLUE);

        System.out.println(p1.equals(p2));
        System.out.println(p2.equals(p1));
    }
}
```

```java
import java.awt.Color;

public class PointColoré extends Point {

    private Color couleur;

    public PointColoré(double coordX, double coordY,
          Color couleur) {
        super(coordX, coordY);
        this.couleur = couleur;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof PointColoré))
            return false;

        PointColoré pc = (PointColoré) obj;
        if (couleur == null && pc.couleur != null)
            return false;
        if (!couleur.equals(pc.couleur))
            return false;

        return super.equals(obj);
    }

}
```

LSP – Conclusion

Si vous avez besoin d'un cast ou d'un `instanceof`, méfiez-vous !



Favor composition over inheritance

Séparation des Interfaces (ISP) – exemple

```
public interface Animal {
    void voler();
    void courir();
    void aboyer();
}
```

```
public class Oiseau implements Animal {
    public void aboyer(){
        System.out.println("non defini");
    }
    public void courir() {
        // du code pour faire courir l'oiseau
    }
    public void voler() {
        // du code pour faire voler l'oiseau
    }
}
```

```
public class Chien implements Animal {
    public void voler(){
        System.out.println("Propriété non définie");
    }
    public void aboyer() {
        // du code pour faire aboyer le chien
    }
    public void courir() {
        // du code pour faire courir le chien
    }
}
```

```
public class Chat implements Animal {
    public void voler() {
        System.out.println("Propriété non définie");
    }
    public void aboyer() {
        System.out.println("Propriété non définie");
    }
    public void courir() {
        // du code pour faire courir le chat
    }
}
```

Pas terrible : pour chacune des classes `Oiseau`, `Chien` et `Chat`, il y a des méthodes inutiles qu'il faut implémenter.

Il faut décomposer l'interface `Animal` !

Séparation des Interfaces (ISP)

Le client d'une entité logicielle ne doit pas dépendre d'une interface qu'il n'utilise pas.

- Plus l'interface est compliquée
  $\Longrightarrow$ plus l'abstraction devient vaste
  $\Longrightarrow$ plus elle devient susceptible de changer dans le temps
- Éviter les "grosses interfaces" - pour éviter le couplage entre les clients qui n'ont rien à avoir un avec l'autre
- Éviter les interfaces spécifiques à un seul client
- Décomposer les fonctionnalités

Inversion des Dépendances (DIP)

Autrefois (il y a bien longtemps) dans les approches de programmation structurée, le haut-niveau dépendait du bas-niveau.

Aujourd'hui :

- Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
- Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.
- Découle directement d'une application correcte d'OCP et LSP.

DIP – Quel est le problème ?

```java
public class Etudiant {
    private String nom, prenom;

    public void Etudiant(String nom, String prenom, String email) {
        //construction
    }
    public String getNom() { return nom; }

    public String getPrenom() { return prenom; }
}
```

```java
public class CompteUniversitaire {
    private Etudiant detenteur;
    private String login;
    private String motDePasse;

    public CompteUniversitaire(String nom, String prenom, String email) {
        detenteur = new Etudiant(nom, prenom, email);
        login = genererLogin();
        motDePasse = genererMdp();
    }

    //On utilise le nom et le prenom du détenteur pour crypter
    public String genererLogin(){
        return detenteur.getNom().substring(0,6) + "." + detenteur.getNom().substring(0,1));
    }

    public String genererMdp(){
        return /* fonction très magique, secrète et tout ça */ ;
    }
}
```



```java
public interface Utilisateur {

    public String getNom();

    public String getPrenom();
}
```

```java
public class CompteUniversitaire {
    private Utilisateur detenteur;

    public CompteUniversitaire(Utilisateur u) {
        detenteur = u;
        login = genererLogin();
        motDePasse = genererMdp();
    }
}
```

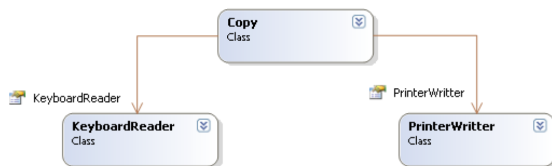
```java
public class Etudiant implements Utilisateur {
    /* des attributs */

    public void Etudiant(String nom, String prenom, String email) {
        //construction
    }
    // implémentation des méthodes de l'interface Utilisateur
}
```

```java
public class Enseignant implements Utilisateur {
    /* des attributs */

    public void Enseignant(String nom, String prenom, String email) {
        //construction
    }
    // implémentation des méthodes de l'interface Utilisateur
}
```

La classe `CompteUniversitaire` ne dépend plus des utilisateurs concrets.

DIP – exemple

Besoin : copier la saisie de l'utilisateur sur un périphérique quelconque (imprimante, écran, disque dur etc.)



Le programme `Copy` utilise un reader pour lire depuis la console et pour écrire sur la console. Jusque là tout baigne...

Qu'est-ce qui se passe si on veut écrire sur un fichier ?
Ou lire depuis un fichier et écrire dans une socket ?

On pense au respect de l'OCP !

$\Longrightarrow$ Étendre en passant par deux interfaces

Conclusion

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Jon Wood (concepteur des jeux vidéos)

- Les principes **SOLID** sont tout d'abord des règles de bon sens
- Beaucoup d'autres principes et bonnes pratiques complémentaires :
  - **GRASP**
  - https://java-design-patterns.com/principles/
- Le développement doit se résumer à une suite d'évolutions (et pas de modifications) simples à intégrer
- Luttez contre les mauvaises odeurs : https://sourcemaking.com/refactoring/smells
- Les Modèles de Conception (Design Patterns) aident souvent à mieux respecter les différents principes