Conception et programmation objet avancées
Notions avancées de POO

Petru Valicov
petru.valicov@univ-amu.fr

2017-2018

Institut Universitaire de Technologie
Aix-Marseille Université
Département Informatique, Aix

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** – les classes doivent effectuer des tâches liées sémantiquement entre elles
- Principe du **contrôleur**
- **Simplifier** le code

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**) – rester général et pas dépendre de l'implémentation

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

Responsabilité unique (SRP)

Qu'en pensez-vous ?

```
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)

"*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

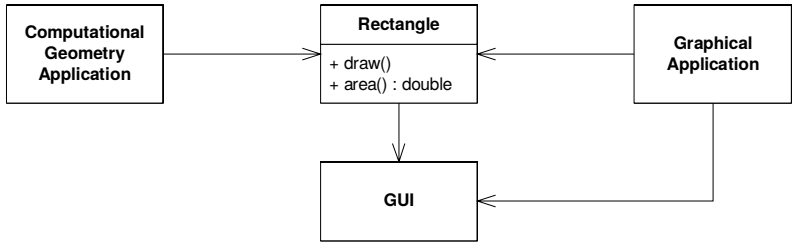
Responsabilité unique (SRP) – exemple

```
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 méthode d'addition ?
- s'il y a changement de méthode de saisie ?
- s'il y a changement de méthode d'addition ?
- je veux ajouter la soustraction !

Amélioration ?

Responsabilité unique (SRP) - autre exemple

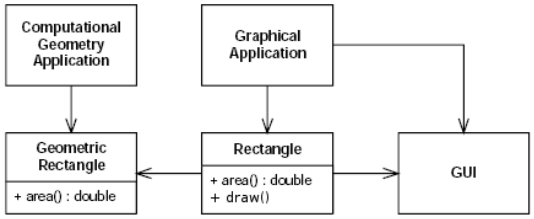


Problèmes avec ce modèle :

- 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) - Morale

- 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

Principe Ouvert/Fermé (OCP) – exemples

Vous en connaissez quelques uns :

- Évidents
  - on ne change pas facilement le type/visibilité d'une variable dans un programme existant
  - on n'efface pas facilement une méthode dans un programme existant
  - on ne change pas facilement la signature d'une méthode dans un programme existant
- Un peu moins... mais vous les connaissez quand même ! ☺
  - publique *vs* privé
  - abstraction pour rester général

Principe Ouvert/Fermé (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 ?

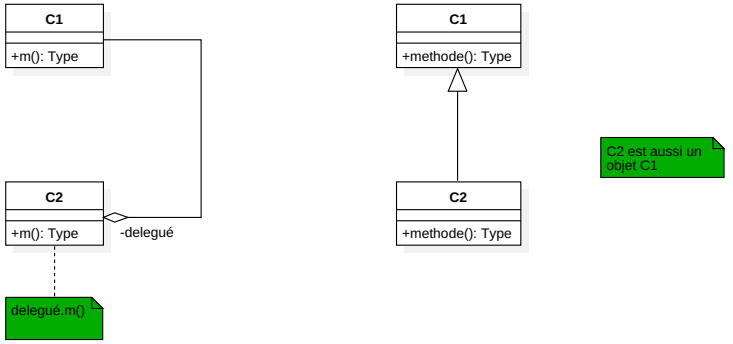
Principe de Substitution de Liskov (LSP)

- *"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
- les types de bases doivent être remplaçables par les sous-types
- Le non-respect de ce principe = l'utilisateur doit connaître **tous** les détails d'implémentation des classes dérivées. Souvent dans ce cas, il y a violation d'OCP.
- Donc : pas de cast et pas de `instanceof` !

Des exemples ?

Héritage vs Délégation

Objectif : Réutiliser du code tout en gardant la sémantique



Prenons le cas des figures géométriques : carré, sphère, rectangle.
Méthodes standards : accesseurs, modifieurs, l'aire et le périmètre

Est-ce que le carré est un rectangle ?

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) {
        changerHauteur(t);
        changerLargeur(t);
    }

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

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

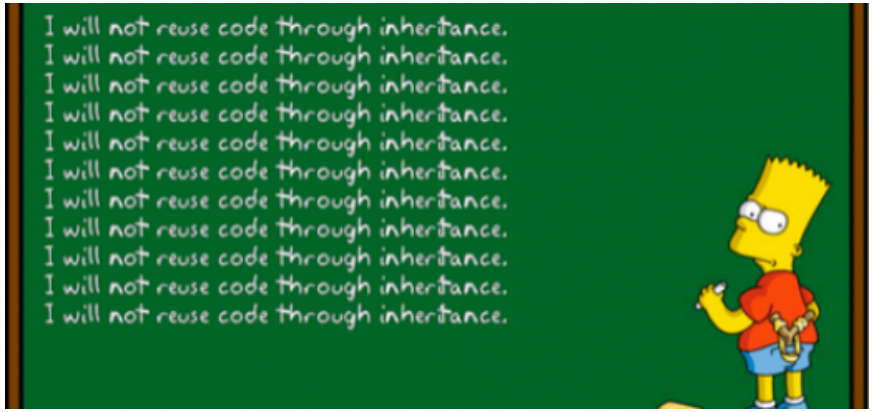
```java
public class Rectangles {

    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 – Conclusion



Séparation des Interfaces (ISP)

Les clients d'une entité logicielle ne doivent pas avoir à dépendre d'une interface qu'ils n'utilisent pas.

- Plus l'interface est compliquée $\Longrightarrow$ plus l'abstraction devient vaste $\Longrightarrow$ plus 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

Séparation des Interfaces (ISP) – exemple

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

```java
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
    }
}
```

```java
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
    }
}
```

```java
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` !

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 ca */ ;
    }
}
```

```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();
    }
}
```

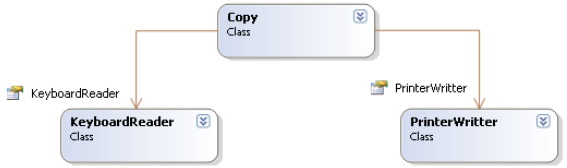
```java
public class Etudiant implements Utilisateur {
    /* des attributs */

    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

Le programme Copy utilise un reader pour lire depuis la console et pour écrire sur la console.



Et si on veut écrire sur un fichier ? Ou lire depuis un fichier et écrire dans une socket ?

Conclusion

- Les principes **SOLID** sont tout d'abord des règles de bon sens
- Objectif : le développement doit se résumer à une suite d'évolutions (et pas de modifications)
- Les évolutions doivent être simples à intégrer
- Les Patrons de Conception (Design Patterns) permettent de mieux les respecter