

Bases de la programmation orientée objet

Compléments

Denis Poitrenaud

IUT de l'Université Paris Descartes
Denis.Poitrenaud@ParisDescartes.fr

DUT Informatique – 2ème Semestre

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

- ▶ La qualité logicielle est un sujet central en génie logiciel (voir http://fr.wikipedia.org/wiki/Qualité_logicielle pour une introduction et une bibliographie générale).
- ▶ (*extrait*) Parmi les critères de qualité, on distingue :
 - ▶ la complétude des fonctionnalités,
 - ▶ la précision des résultats,
 - ▶ la fiabilité,
 - ▶ la tolérance de pannes,
 - ▶ la facilité et la flexibilité de son utilisation,
 - ▶ la performance,
 - ▶ la consistance,
 - ▶ l'intégrité des informations,
 - ▶ la compatibilité et la portabilité,

 - ▶ la simplicité,
 - ▶ l'extensibilité,
 - ▶ la facilité de correction et de transformation.

- ▶ Nous nous focalisons ici uniquement sur les critères internes de qualité (i.e. la qualité des sources)
- ▶ **Les classes de vos projets doivent respecter ces critères**
- ▶ Des outils informatiques peuvent vous aider à repérer une bonne part des erreurs les plus courantes
 - ▶ **Checkstyle** (<http://checkstyle.sourceforge.net/>)
Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.
 - ▶ **Findbugs** (<http://findbugs.sourceforge.net/>)
FindBugs is a program to find bugs in Java programs. It looks for instances of "bug patterns" — code instances that are likely to be errors.
- ▶ Les deux logiciels ci-dessus sont disponibles sous la forme de plugins Eclipse
 - ▶ <http://eclipse-cs.sourceforge.net>
 - ▶ <http://findbugs.cs.umd.edu/eclipse/>

Simplicité et facilité de compréhension

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

- ▶ Un des critères internes de qualité permettant d'obtenir une bonne qualité externe est la simplicité des fichiers sources
- ▶ Un programmeur (ou un correcteur) devrait pouvoir comprendre les algorithmes mis en œuvre dans un programme à sa simple lecture :
 - ▶ Bien choisir les identificateurs (des classes, méthodes, attributs, variables, constantes, etc)
 - ▶ Repousser les détails dans des méthodes auxiliaires (privées au besoin)
 - ▶ Éviter les variables intermédiaires inutiles.
 - ▶ Employer les idiomes classiques de la programmation
 - ▶ Exemple :

```
        return a < b;
```

plutôt que

```
        if (a < b) return true; else return false;
```

Autre exemple d'idiome

- ▶ 3 boucles différentes pour exécuter n fois des instructions

```
for (int i = 1; i <= n; i = i + 1) {  
    ...  
}
```

```
for (int i = 0; i < n; ++i) {  
    ...  
}
```

```
int i = 0; // la portée de i n'est pas la même  
while (i < n) {  
    ...  
    i += 1;  
}
```

- ▶ La seconde est la plus employée \implies il faut la privilégier

Une bonne présentation facilite grandement la lecture et la compréhension

- ▶ L'indentation doit être correcte et homogène
 - ▶ Réindentation automatique avec Eclipse
menu Sources → Format
- ▶ Le programme doit pouvoir être imprimé tout en préservant une indentation correcte
 - ▶ Chaque ligne ne doit pas dépasser 60 caractères
 - ▶ Les tabulations doivent correspondre à 2 espaces
 - ▶ Le formatage proposé par Eclipse réorganise automatiquement les lignes pour qu'elles aient la bonne longueur

Respect de l'encapsulation

Qualité logicielle

Simplicité et facilité de compréhension

Respect de l'encapsulation

Atomicité des méthodes publiques

Erreurs usuelles

Dépendances

Généricité

Classes génériques

Méthodes génériques

Objets non modifiables

Objets immuables

Énumération avec données associées

Classes internes

Respect de l'encapsulation

- ▶ Le principe d'encapsulation impose que l'état d'un objet ne puisse être modifié qu'au travers les méthodes

```
public class Etudiant {
    private String nom;
    private ArrayList<Double> notes;

    public Etudiant(String nom) {
        this.nom = nom;
        this.notes = new ArrayList<Double>();
    }

    public void ajouter(Double note) {
        notes.add(note);
    }

    public double moyenne() { ... }

    public String toString() {
        return nom + " (moyenne: " + moyenne() + ")";
    }
}
```

Respect de l'encapsulation ?

- ▶ La classe ne permet pas de modifier une note d'un étudiant une fois qu'elle a été ajoutée
- ▶ On veut pouvoir créer un étudiant avec une liste de notes initialement non vide
- ▶ On ajoute un nouveau constructeur :

```
public class Etudiant {  
    ...  
    public Etudiant(String nom, ArrayList<Double> notes) {  
        this.nom = nom;  
        this.notes = notes;  
    }  
    ...  
}
```

- ▶ Le principe d'encapsulation est-t-il encore respecté ?

Violation du principe d'encapsulation

```
public class Scolarité {
    public static void main(String [] args) {
        ArrayList<Double> liste = new ArrayList<Double>();
        liste.add(8.);
        liste.add(10.);

        Etudiant e = new Etudiant("toto", liste);
        System.out.println(e.toString());

        liste.set(0, 15.);

        System.out.println(e.toString());
    }
}
```

Ce programme affiche :

```
toto (moyenne : 9.0)
toto (moyenne : 12.5)
```

- ▶ Il faut garder une copie de la liste fournie plutôt que la liste elle-même

```
public class Etudiant {  
    ...  
    // On conserve une copie de la liste  
    public Etudiant(String nom, ArrayList<Double> notes) {  
        this.nom = nom;  
        // Usage du constructeur par copie  
        this.notes = new ArrayList<Double>(notes);  
    }  
  
    // De même, nous pouvons introduire un getter  
    // mais il doit retourner une copie de la liste  
    public ArrayList<Double> getNotes() {  
        return new ArrayList<Double>(notes);  
    }  
    ...  
}
```

Respect de l'encapsulation

- ▶ On ne doit pas conserver en mémoire des références fournies par l'utilisateur
- ▶ Toutefois, peut-on modifier le nom d'un étudiant *a posteriori* ?

```
public class Scolarité {  
    public static void main(String [] args) {  
        String nom = "toto";  
        Etudiant e = new Etudiant(nom);  
        e.ajouter(5.);  
        System.out.println(e.toString());  
  
        nom = "titi"; // sans effet sur e  
  
        System.out.println(e.toString());  
    }  
}
```

- ▶ La classe `String` n'offre aucune méthode permettant de modifier la chaîne. La copie préventive du nom est donc inutile.

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

Atomicité des méthodes publiques

- ▶ Tout constructeur doit initialiser toutes les données de l'objet et les positionner à des valeurs cohérentes
- ▶ Toute méthode publique invoquée sur un objet doit laisser ses données dans un état cohérent
- ▶ Contre-exemple :

```
public class JeuMalConçu {  
    ...  
    public void jouer(Coup c) {  
        ...  
    }  
  
    public void ajouterPoints(Coup c) { ... }  
  
    public void passerAuJoueurSuivant() { ... }  
}
```

► Solution :

```
public class JeuBienConçu {  
    ...  
    public void jouer(Coup c) {  
        ...  
        this .ajouterPoints(c);  
        this .passerAuJoueurSuivant();  
    }  
  
    private void ajouterPoints(Coup c) { ... }  
  
    private void passerAuJoueurSuivant() { ... }  
}
```

Qualité logicielle

Simplicité et facilité de compréhension

Respect de l'encapsulation

Atomicité des méthodes publiques

Erreurs usuelles

Dépendances

Généricité

Classes génériques

Méthodes génériques

Objets non modifiables

Objets immuables

Énumération avec données associées

Classes internes

- ▶ Méthodes trop longues (moins lisible)
- ▶ Code dupliqué (moins lisible et reproduction des bugs)
- ▶ Attributs inutiles (souvent des variables locales mal placées)
- ▶ Paramètres inutiles
- ▶ Responsabilités mal placées (méthode d'instance au lieu de méthode statique ou à déplacer vers une autre classe)
- ▶ Affichages et saisies mal placés (il vaut mieux laisser l'utilisateur de la classe afficher ou saisir les données nécessaires)
- ▶ Types de données mal choisis (des chaînes de caractère partout par exemple)
- ▶ Nombres (ou chaînes, caractères, ...) magiques

Qualité logicielle

Simplicité et facilité de compréhension

Respect de l'encapsulation

Atomicité des méthodes publiques

Erreurs usuelles

Dépendances

Généricité

Classes génériques

Méthodes génériques

Objets non modifiables

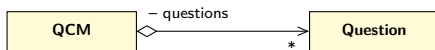
Objets immuables

Énumération avec données associées

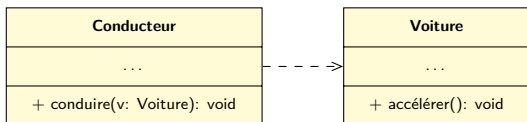
Classes internes

Dépendances inter-classes

- ▶ Une classe A dépend d'une classe B si le code de la classe A manipule des données de type B
- ▶ Dans un diagramme de classe, on distingue la relation d'agrégation (association/composition) :

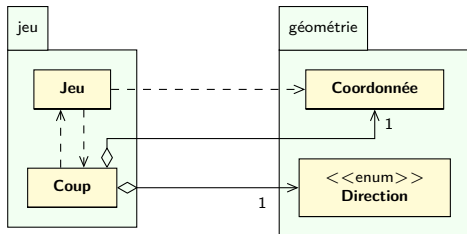


- ▶ de la simple relation d'utilisation :

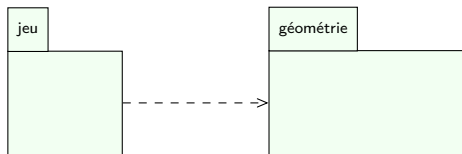


Dépendances inter-paquetages

- ▶ Un paquetage A dépend d'un paquetage B si une classe de A dépend d'une classe de B



- ▶ Diagramme de paquetage



- ▶ Un paquetage est l'unité de livraison (i.e. un ensemble minimal de classes pouvant être fourni à un utilisateur)
- ▶ Si un paquetage A dépend d'un paquetage B alors
 - ▶ B doit être livré avec A
 - ▶ si B est modifié ou corrigé alors les utilisateurs de A doivent être livrés à nouveau
- ▶ Les circuits de dépendances entre paquetages ($A \rightarrow B \rightarrow \dots \rightarrow A$) doivent être évités
- ▶ Ils dénotent généralement d'une erreur de conception (classe mal placée, dépendance inutile, ...)

Qualité logicielle

- Simplicité et facilité de compréhension
- Respect de l'encapsulation
- Atomicité des méthodes publiques
- Erreurs usuelles
- Dépendances

Généricité

- Classes génériques
- Méthodes génériques

Objets non modifiables

- Objets immuables
- Énumération avec données associées

Classes internes

Classes génériques

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques**

- Méthodes génériques

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

Classes génériques

Les classes peuvent être paramétrées par des types :

```
public class Exemple {  
    public static void main(String [] args) {  
        // les listes sont paramétrées par le type  
        // des éléments qu'elles contiennent  
        ArrayList<String> a = new ArrayList<String>();  
        a.add("un_élément");  
        a.add("un_deuxième_élément");  
  
        // les tableaux associatifs par le type des clés  
        // et celui des données associées  
        HashMap<String, String> annuaire;  
        annuaire = HashMap<String, String>();  
        annuaire.put("Mickey", "06-62-54-37-43");  
        annuaire.put("Daisy", "07-14-78-29-03");  
        String tél = annuaire.get("Daisy");  
    }  
}
```

Définition d'une classe générique

- ▶ Le nom de la classe doit être suivi du nom des paramètres formels.
- ▶ Ces paramètres seront remplacés par des types concrets à l'instanciation (i.e. lors de l'utilisation).

```
public class Couple<T> {  
    private T premier, second;  
  
    public Couple(T un, T deux) {  
        premier = un;  
        second = deux;  
    }  
    ...  
}
```

Définition d'une classe générique – suite

```
public class Couple<T> {  
    ...  
  
    public T getPremier() {  
        return premier;  
    }  
  
    public T getSecond() {  
        return second;  
    }  
  
    public String toString() {  
        return "(" + premier.toString() + ", " +  
                second.toString() + ")";  
    }  
}
```

Utilisation d'une classe générique

```
public class Appli {  
    public static void main(String [] args) {  
        Couple<Integer> ci = new Couple<Integer>(1, 2);  
        System.out.println(ci.toString());  
  
        Couple<Couple<Integer>> cc;  
        cc = new Couple<Couple<Integer>>(ci, ci);  
        System.out.println(cc.toString());  
    }  
}
```

► Le programme affiche :

(1, 2)

((1, 2), (1, 2))

Utilisation d'une classe générique – suite

Les seuls paramètres génériques effectifs autorisés sont des classes et des tableaux

```
public class Appli {  
    public static void main(String [] args) {  
  
        Couple<String> cs; // ok  
        cs = new Couple<String>("a", "b");  
  
        Couple<int[]> ct; // ok  
        ct = new Couple<int[]>(new int [2], new int [3]);  
  
        Couple<int> ci; // not ok  
    }  
}
```

Paramètres génériques

- ▶ En conséquence, on ne peut appliquer aux données de ces types que ce qui est commun aux classes et aux tableaux \implies les protocoles standards :

```
public class Couple<T> {  
    ...  
    public boolean contient(T e) {  
        return premier.equals(e) || second.equals(e);  
    }  
  
    public String toString() {  
        return "(" + premier.toString() + ", " +  
            second.toString() + ")";  
    }  
}
```

Nous verrons comment aller plus loin dans la 2^{nde} partie du cours.

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques**

Objets non modifiables

- Objets immuables

- Énumération avec données associées

Classes internes

Définition de méthodes génériques

- ▶ Une méthode peut être paramétrée par des types
- ▶ Les paramètres génériques formels précèdent le prototype

```
public class Exemple {  
    public static <T> T choixAléatoireParmi(T[] tab) {  
        Random rd = new Random();  
        return tab[rd.nextInt(tab.length)];  
    }  
}
```

```
public static <T> void permute(Couple<T> c) {  
    T tmp = c.getPremier();  
    c.setPremier(c.getSecond());  
    c.setSecond(tmp);  
}  
}
```

Utilisation de méthodes génériques

- ▶ Les paramètres génériques formels sont automatiquement déduits de l'invocation de la méthode

```
public class Exemple {
    public static void main(String [] args) {
        Integer [] tab = { 1, 2, 3, 4, 5 };
        // T reçoit Integer
        int i = Exemple.choixAléatoireParmi(tab);
        System.out.println(i);

        Couple<Integer> ci = new Couple<Integer>(1, 2);
        System.out.println(ci);
        // T reçoit Integer
        Exemple.permute(ci);
        System.out.println(ci);
    }
}
```

Objets non modifiables

Qualité logicielle

- Simplicité et facilité de compréhension
- Respect de l'encapsulation
- Atomicité des méthodes publiques
- Erreurs usuelles
- Dépendances

Généricité

- Classes génériques
- Méthodes génériques

Objets non modifiables

- Objets immuables
- Énumération avec données associées

Classes internes

Objets immuables

Qualité logicielle

- Simplicité et facilité de compréhension

- Respect de l'encapsulation

- Atomicité des méthodes publiques

- Erreurs usuelles

- Dépendances

Généricité

- Classes génériques

- Méthodes génériques

Objets non modifiables

- Objets immuables**

- Énumération avec données associées

Classes internes

Objets immuables

- ▶ Un objet est dit immuable si aucune méthode de la classe ne permet de modifier l'état de l'objet.
- ▶ Exemple :

```
public final class Coordonnée {  
    private final int x, y;  
  
    public Coordonnée(int x, int y) {  
        this.x = x; this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
}
```

- ▶ Toutefois, la classe peut offrir des méthodes retournant de nouveaux objets du même type
- ▶ Exemple :

```
public class Coordonnée {  
    ...  
  
    public Coordonnée déplace(int dx, int dy) {  
        return new Coordonnée(x + dx, x + dy);  
    }  
}
```

- ▶ La bibliothèque standard propose de nombreuses classes d'objets immuables
- ▶ Exemples :
 - ▶ String,
 - ▶ Integer, Long, BigInteger,
 - ▶ Float, Double, BigDecimal,
 - ▶ Color, ...
- ▶ Ils offrent différents avantages
 - ▶ Il n'est pas nécessaire de faire des copies défensives pour assurer l'encapsulation
 - ▶ Ils n'ont pas à offrir des moyens de les copier (constructeur par copie, clone, ...)
 - ▶ Ils sont de bons candidats en tant que clé dans les tableaux associatifs

La bibliothèque standard permet de rendre immuable une liste.

```
import java.util.*;

public final class Panier {
    private final List<Denrée> articles;

    public Panier(List<Denrée> articles) {
        this.articles = Collections.unmodifiableList(
            new ArrayList<Denrée>(articles));
    }

    public List<Denrée> getArticles() {
        return articles;
    }

    public double total() {
        double cumul = 0;
        for (Denrée d: articles) cumul += d.getPrix();
        return cumul;
    }
}
```

Énumération avec données associées

Qualité logicielle

- Simplicité et facilité de compréhension
- Respect de l'encapsulation
- Atomicité des méthodes publiques
- Erreurs usuelles
- Dépendances

Généricité

- Classes génériques
- Méthodes génériques

Objets non modifiables

- Objets immuables
- Énumération avec données associées

Classes internes

```
package circulation ;  
  
public enum Couleur {  
    ROUGE, ORANGE, VERT ;  
}
```

- ▶ Un type énuméré définit un type pouvant prendre un nombre limité de valeurs (celles qui sont énumérées)
- ▶ Un type énuméré permet de donner des noms symboliques (plus facile à interpréter qu'un code numérique) à des valeurs particulières
- ▶ Les valeurs peuvent être vues comme des objets immuables

Énumération – utilisation

```
import static circulation.Couleur.*;

public class Feu {
    private Couleur couleur = VERT;

    public void suivant() {
        switch (couleur) {
            case VERT :
                couleur = ORANGE; break;
            case ORANGE :
                couleur = ROUGE; break;
            case ROUGE :
                couleur = VERT; break;
        }
    }

    public boolean peutPasser() {
        return couleur == VERT;
    }
}
```

Énumération avec données associées

```
public enum Direction {  
    EST(1, 0), OUEST(-1, 0), NORD(0, -1), SUD(0, 1);  
  
    // Données associées  
    // Aucune méthode (autre que le constructeur)  
    // ne doit les modifier  
    private final int dx, dy;  
  
    // Le constructeur ne peut être employé que pour les  
    // valeurs ci-dessus (quelque soit sa visibilité)  
    Direction(int dx, int dy) {  
        this.dx = dx; this.dy = dy;  
    }  
  
    public int getDx() { return dx; }  
  
    public int getDy() { return dy; }  
}
```

```
public class Coordonnée {  
    private int x, y;  
  
    public Coordonnée(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public void déplaceVersLe(Direction d) {  
        x += d.getDx();  
        y += d.getDy();  
    }  
  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

Qualité logicielle

- Simplicité et facilité de compréhension
- Respect de l'encapsulation
- Atomicité des méthodes publiques
- Erreurs usuelles
- Dépendances

Généricité

- Classes génériques
- Méthodes génériques

Objets non modifiables

- Objets immuables
- Énumération avec données associées

Classes internes

Plusieurs classes au sein d'un même fichier

- ▶ Un même fichier source peut contenir la définition de plusieurs classes mais une seule peut être déclarée publique
- ▶ Les autres classes doivent avoir la visibilité paquetage (et elles pourront être employées uniquement dans le paquetage)
- ▶ Le nom du fichier doit correspondre à celui de la classe publique si elle existe ou d'une des classes du fichier (n'importe laquelle) dans le cas contraire

Classes définies au sein d'une autre classe

- ▶ La définition d'une classe peut contenir la définition d'autres classes
- ▶ Leur visibilité peut être publique, privée ou de niveau paquetage
- ▶ Hors de la classe englobante et lorsqu'elles sont visibles, il faut employer leur nom complet (le nom de la classe englobante suivi du nom de la classe englobée) pour les désigner

Les classes internes

- ▶ Une classe interne (englobée dans une autre classe) peut être statique ou non-statique
- ▶ Une classe interne statique est une classe traditionnelle
- ▶ La déclarer au sein d'une autre classe permet de limiter sa visibilité (privée) et/ou de marquer sa fonction utilitaire par rapport à la classe englobante

```
public class Liste {  
    private static class Maillon {  
        int valeur;  
        Maillon suivant;  
        public Maillon(int val, Maillon suiv) {  
            valeur = val; suivant = suiv;  
        }  
    }  
  
    private Maillon tête = null;  
  
    public void ajouter(int v) {  
        tête = new Maillon(v, tête);  
    }  
}
```

- ▶ Une classe interne non-statique est une classe dont chaque objet est lié à un objet de la classe englobante
- ▶ L'objet englobant est désigné à la construction de l'objet englobé
- ▶ L'objet englobé détient une référence vers l'objet englobant
- ▶ Cette référence permet à l'objet englobé d'accéder aux données (privées ou non) de l'objet englobant

Classe interne non-statique – Exemple

```
package banque;  
  
public class Banque {  
    private String nom;  
  
    public Banque(String n) {  
        nom = n;  
    }  
  
    public String toString() {  
        return nom;  
    }  
  
    ....  
}
```

Classe interne non-statique – Exemple

```
package banque;

public class Banque {
    ...
    public class Compte {
        // attributs statiques non-constants interdits
        private int num;

        public Compte(int n) {
            num = n;
        }

        public String toString() {
            // accès aux données de l'objet englobant
            // (nom désigne l'attribut nom de la banque)
            return "compte_␣n°" + num + "␣(" + nom + ")";
        }

        public Banque getBanque() {
            // accès à l'objet englobant
            return Banque.this;
        }
    }
}
```

Classe interne non-statique – Utilisation

```
import banque . Banque . Compte ;

public class Appli {
    public static void main (String [] args) {
        Banque b = new Banque ("CL");
        Compte c ;
        // un compte ne peut être créé qu'au sein
        // d'une banque particulière
        c = b.new Compte (1);
        System.out.println (c.toString ());
        System.out.println (c.getBanque ().toString ());
        // avec une variable anonyme
        c = new Banque ("CA").new Compte (2);
        System.out.println (c.toString ());
        System.out.println (c.getBanque ().toString ());
    }
}
```

compte n°1 (CL)

CL

compte n°2 (CA)

CA