DESIGN PATTERNS

INTRO ET STRATEGY

Supports Frédéric Moal

Yohan BOICHUT
Université d'Orléans

M2 INSA

année 2019/2020

Design Patterns : Plan

- Qu’est que les “design patterns” ?
- Trois familles de Design patterns
  - Patterns de création
  - Patterns de structure
  - Patterns comportementaux
- Présentations de différents design patterns par l’exemple

Bibliographie…

- « A System of Pattern » Bushmann et All
- « Design Patterns » Gamma et All
- « Applying UML and Patterns » Larman
- « Design Patterns, tête la première » O’reilly
- developpez.com, rubrique Conception
- Wikipédia, patrons de conception / design patterns (eng)

Modélisation d’objets

Des objectifs *souvent* antagonistes :

- Encapsuler des données sans en empêcher l’accès
- Trouver le bon niveau de granularité des objets
- Limiter les dépendances entre objets
- Concevoir des objets polyvalents, flexibles, réutilisables
- Simplicité d’utilisation
- Implémentation performante

Modélisation d'applications

Modéliser correctement une application :

- Processus complexe
- Expertise acquise au fil des « expériences » (bétises…)
- Problèmes de conceptions récurrents : des *Design Patterns*

Un "livre de recettes" :

Design Patterns, Elements of Reusable Object-Oriented Software
E. Gamma, R. Helm, R. Johnson, J. Vlissides
1995 - Addison Wesley

Devenir un maitre aux échecs

- Apprendre les règles de base
  - le nom des pièces, les mouvements autorisés, la géométrie …
- Apprendre les principes
  - la valeur relative de chaque pièce, les positions stratégiques, le roc…
- Mais pour devenir un maitre, il faut étudier les parties des autres maitres
  - Ces parties contiennent des “schémas de jeu” qui doivent être compris, mémorisés, pour pouvoir être réutilisés “en contexte”
- Il y a des milliers de ces “patterns”

Devenir un maitre en developpement

- Apprendre les règles de base
  - Le SDD
- App
  - la ulaire, ori
- Mai aut étu s des autr
  - Ce schémas” qu s, pour po
- Il y a des milliers de ces “patterns”

Design patterns

- Le but général des patrons de conception est de minimiser les interactions qu'il peut y avoir entre les différentes classes (ou modules, plus généralement) d'un même programme
- Les avantages de ces patrons sont
  - de diminuer le temps nécessaire au développement d'un logiciel (!)
  - d'augmenter la qualité du résultat, notamment en appliquant des solutions déjà existantes à des problèmes courants de conception
  - utiles pour définir un vocabulaire commun entre les différents acteurs de l'écriture d'un logiciel
- Ces patrons sont décrits sous une forme abstraite, sans s'attacher aux détails du problème à résoudre.

Un Design Pattern

Nom
Exposé du problème
Contexte de mise en œuvre, limitations
Description de la solution proposée
Exemple d'implémentation
Conseils d'implémentation
Confrontation avec d'autres Design Patterns

- Modèles parfois (souvent ?) triviaux
- Relative standardisation du nommage des Design Patterns

Principales classes de Design Patterns

Patterns de création

- Création d'objets sans instanciation directe d'une classe

Patterns de structure

- Composition de groupes d'objets

Patterns comportementaux

- Modélisation des communications inter-objets et du flot de données

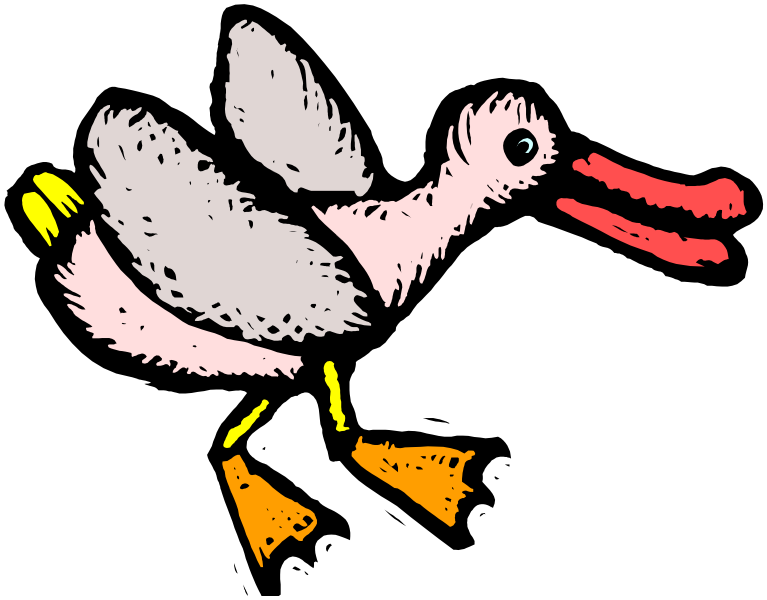
parallèle avec UML : les deux premiers modèles liés à des diagrammes statiques (de classes), dernier modèle est davantage lié à un diagramme dynamique (de séquence)

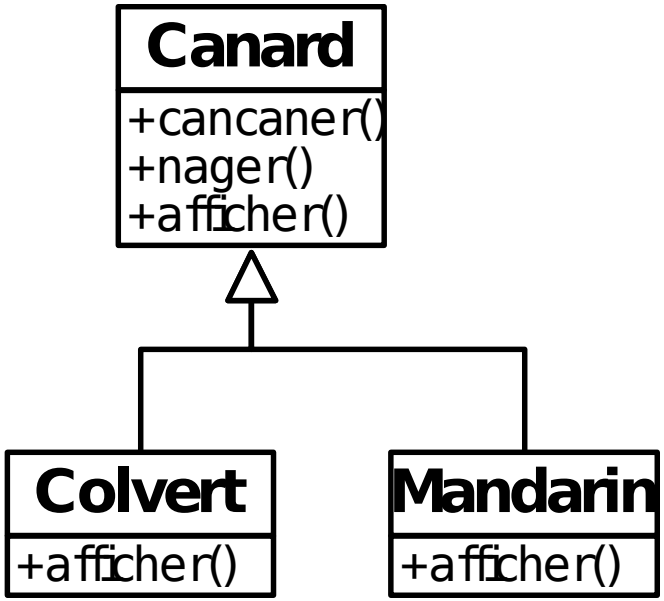
Les Design Patterns

	Utilisation		
	Création	Structure	Comportement
Scope	Factory Method Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Interpreter Template Method Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor Callback

« Petite » introduction

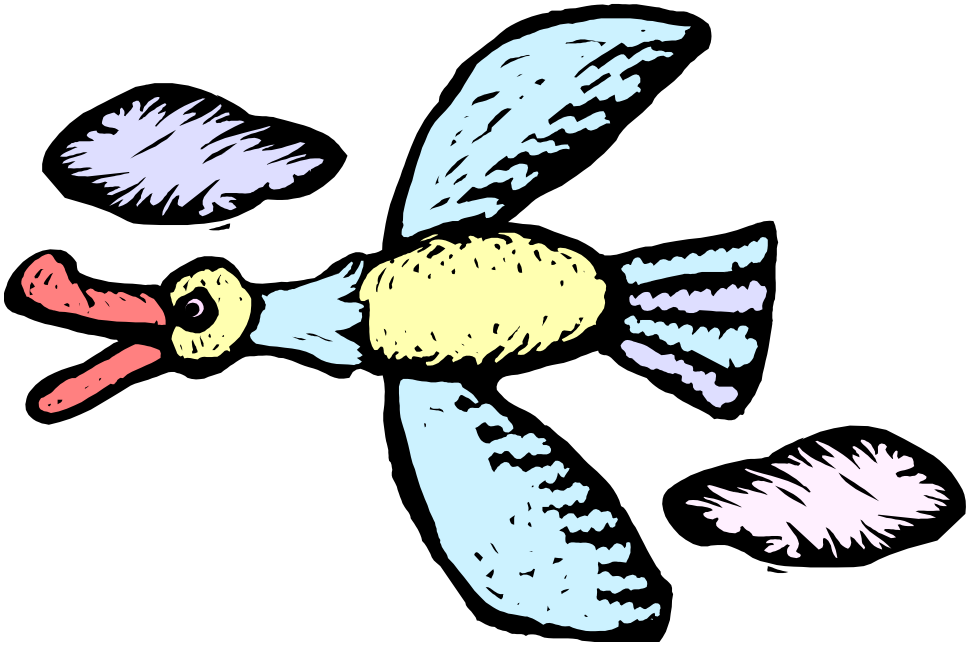
Application La mare aux Canards, grand succès commercial !





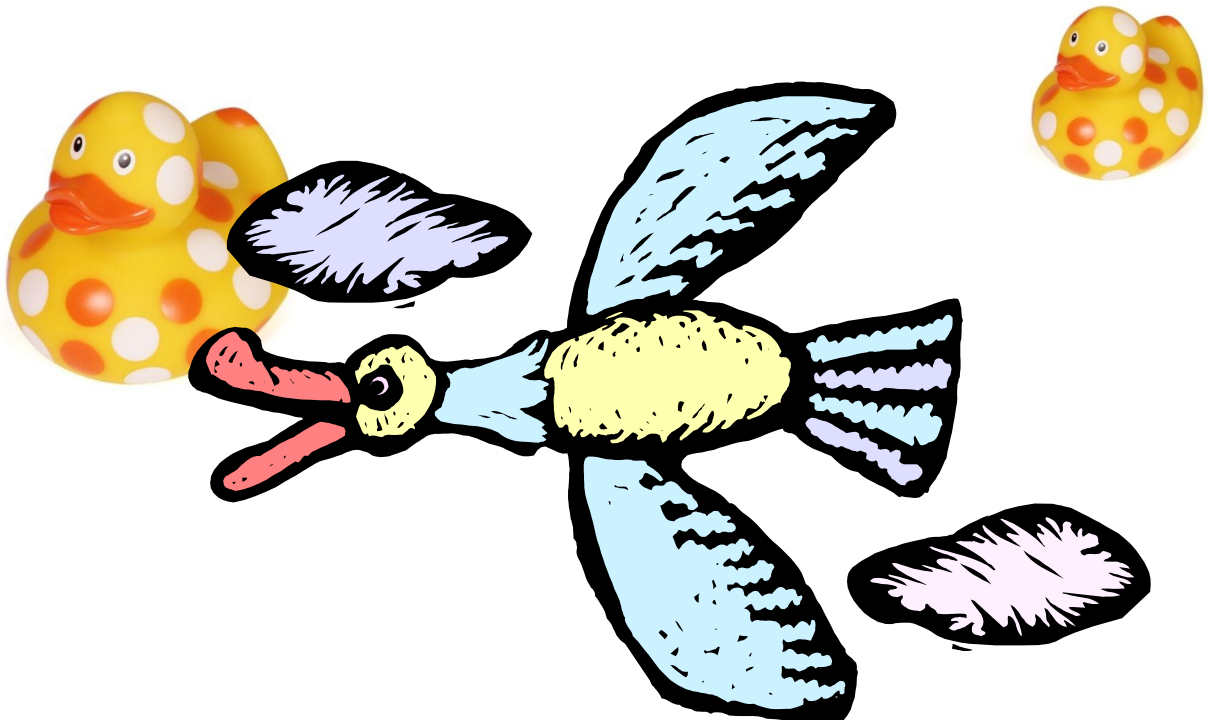
« Petite » introduction

Développement d’une suite : ils volent !



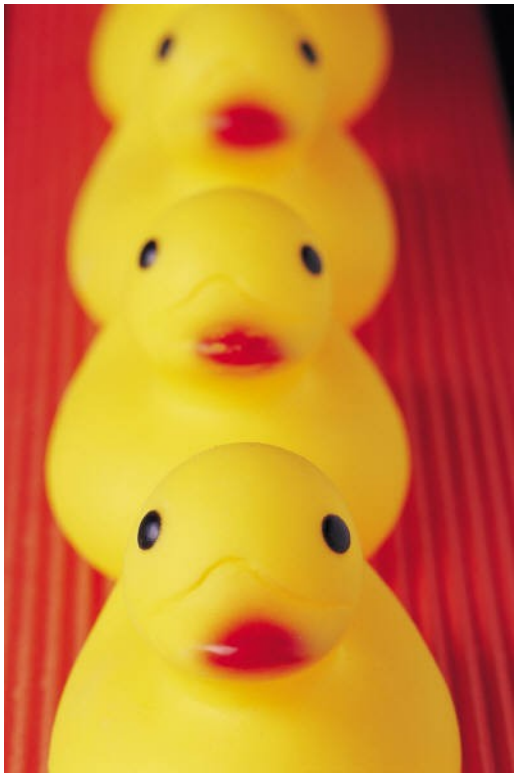
« Petite » introduction

Aïe le bug !



« Petite » introduction

Des interfaces ?



« Petite » introduction

Principe de conception :

Identifiez les aspects de votre application qui varient et séparez les de ceux qui demeurent constants

Pour les canards :
Extraire les comportements de vol et de cancanement dans des structu
classes distinctes

« Petite » introduction

Principe de conception :

Programmer une interface, non une implémentation

Pour les canards :
Utiliser une interface pour chaque type de comportement qui sera implémentée par des classes spécifiques

« Petite » introduction

Le Canard va déléguer ses comportements
Voler et Cancaner

Comment ?

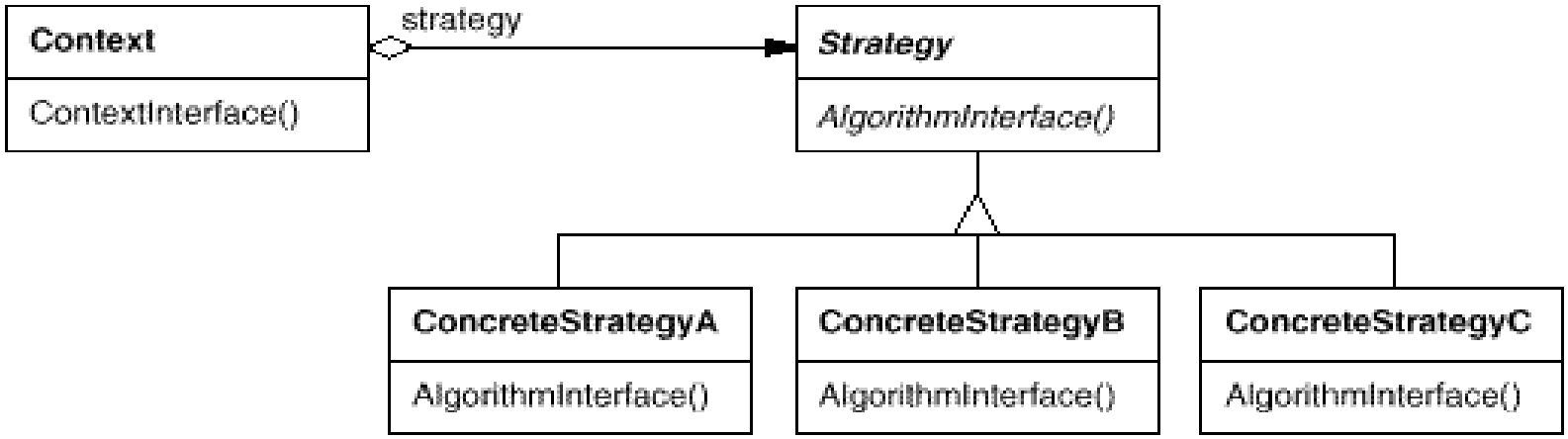
Leçons de l’histoire

- Principes de conception
  - Identifiez les aspects de votre application qui varient et séparez les de ceux qui demeurent constants
  - Programmez une interface et non une implémentation
  - Préférez la composition à l’héritage !

Stratégie / Strategy

Objectif : utiliser de manière non spécifique une collection d'algorithme proches

Exemples : algorithmes de tris de collections de données, comportements des canards



Stratégie / Strategy : exemple 2

- **Le tri de listes en Java :**

Il existe 2 fonctions de tri dans Collections :

`sort(List)`

et

`sort(List,Comparator)`

Stratégie / Strategy : exemple 2

- **sort(List)**

  Cette fonction trie la liste passée en paramètre en fonction de leur "ordre naturel"

Tous les éléments de la liste doivent implémenter l'interface Comparable, et donc redéfinir la fonction `compareTo(Object)` :

- renvoie 0 si les deux objets sont égaux,
- moins si l'objet appelant est "inférieur"
- plus si l'objet appelant est "supérieur"

Stratégie / Strategy : exemple 2

- **sort(List)**

```
public class Personne implements Comparable {
   public String nom,prenom;
   Personne(String nom, String prenom) {
   this.nom=nom;
      this.prenom=prenom;
   }
   public int compareTo(Object o) {
      Personne p=(Personne)o;
      if(nom.equals(p.nom)) {
         return prenom.compareTo(p.prenom);
      }
      return nom.compareTo(p.nom);
   }
}
```

-> Collections.sort(l)

Stratégie / Strategy : exemple 2

- **sort(List,Comparator)**

Cette fonction trie la liste passée en paramètre en fonction du **`Comparator`** passé en paramètre. Supposons la classe Personne précédemment définie(sans implémenter **`Comparable`**). Il faut créer une classe implémentant l'interface **`Comparator`**, et donc redéfinissant les fonctions ***`compare(Object,Object)`*** et ***`equals(Object)`***

Stratégie / Strategy : exemple 2

- ***sort(List,Comparator)***

```
import java.util.Comparator;
public class MonComparator implements Comparator {
  public int compareTo(Object arg0, Object arg1) {
  Personne p1 = (Personne) arg0;
  Personne p2 = (Personne) arg1;
  int result = p1.name.compareTo(p2.name);
  if(result==0)
  result = p1.prenom.compareTo(p2.prenom);
  return result;
  }
} -> Collections.sort(l,new MonComparator())
```

Stratégie / Strategy : exemple 2

- **`Collections.sort(liste)`**

  Approche statique

  Comportement unique déterminé à la compilation

  Implémentation de l'interface Comparable

- **`Collections.sort(l,new MonComparateur())`**

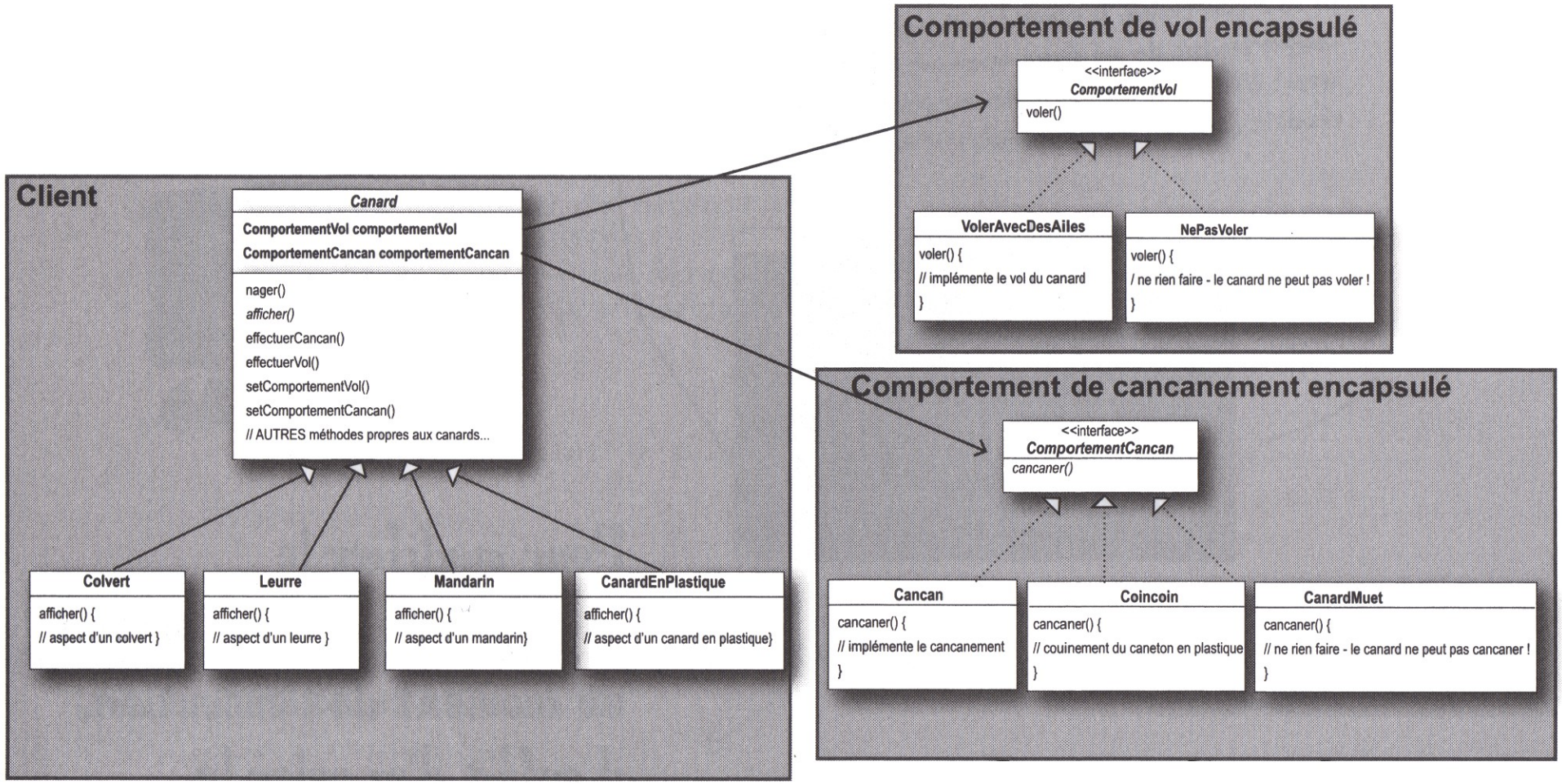
  Approche dynamique

  Comportement modifiable en cours d'exécution

  Ajout d'un nouveau comportement par implémentation de Comparator

Comportement dynamique

- Résumé de l'exemple 1 :



Conclusions

- Principes de conception
  - Identifiez les aspects de votre application qui varient et séparez les de ceux qui demeurent constants
  - Programmez des interfaces et non des implémentations
  - Préférez la composition à l'héritage !
- Patron de conception
  - Stratégie : définit une famille d'algorithmes, encapsule chacun d'eux et les rends interchangeables