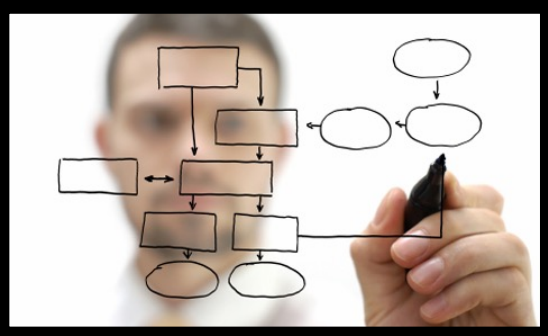
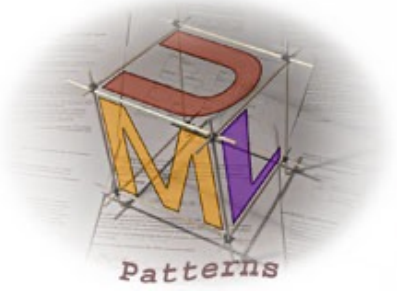
Université de Corse
2025-2026
MASTER DFS 1ère année

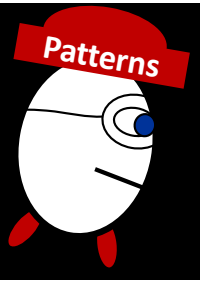
Cours Patterns de Conception

CH 1 - Fondements de l'approche « Patterns »

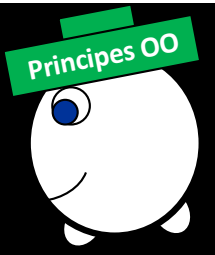


Evelyne VITTORI
vittori@univ-corse.fr



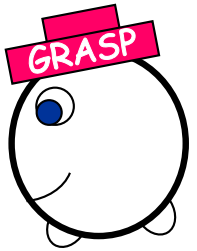


Patterns de Conception
Plan du Cours



CH1 – Fondements de l'approche « Patterns »

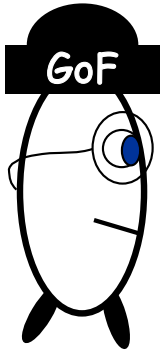
CH2 – Patterns Gof

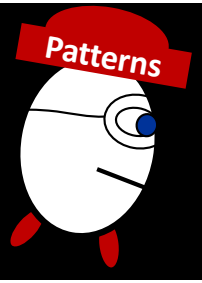


2.1 – Patterns créationnels

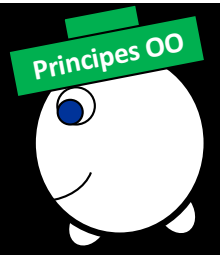
2.2 - Patterns structurels

2.3 - Patterns comportementaux





CH 1 – FONDEMENTS DE L'APPROCHE « PATTERNS »

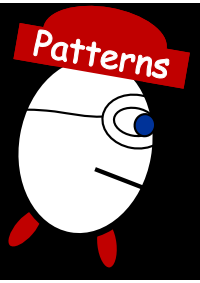


1. Historique

- ✓ Origine des patterns
- ✓ Application à l'informatique
- ✓ Catégories de patterns

2. Notions de base

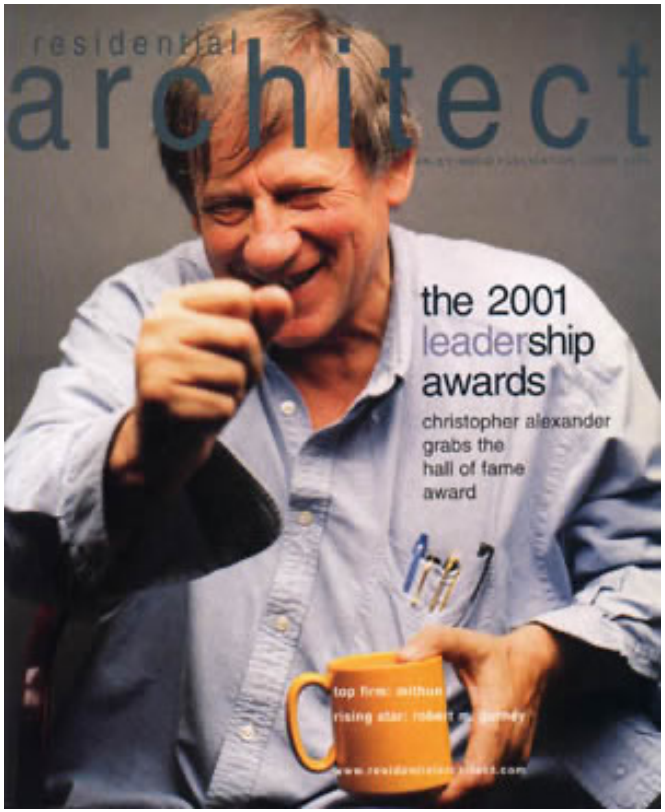
- ✓ Exemple d'introduction
- ✓ Description d'un Design Pattern
- ✓ Design Patterns et ACOO



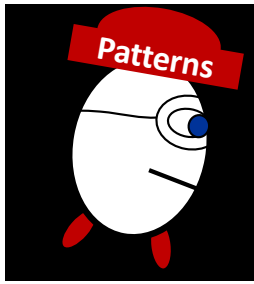
1 – Historique
Origine des patterns

Architecture (1979)

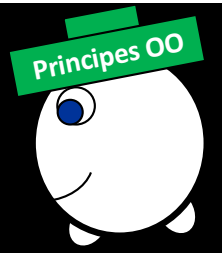
Existe-il des critères de qualité objectifs, irréfutables??



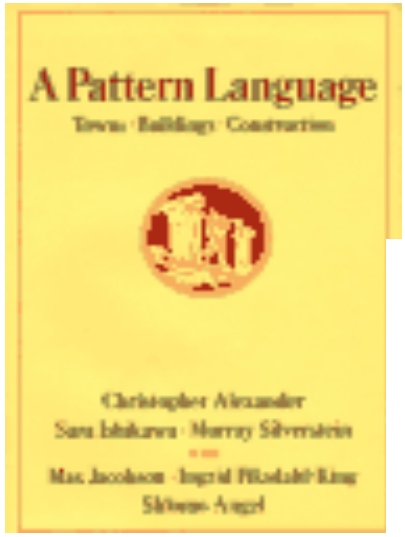
Christopher Alexander dit OUI!!



Origine des patterns



Travaux de Christopher Alexander

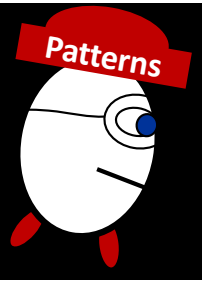


- Mise en évidence des points communs des « bonnes » constructions : Patterns

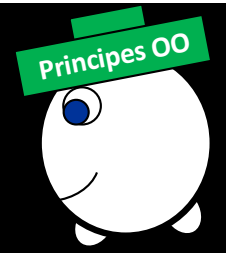
"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way that you can use this solution a million times over, without ever doing it the same way twice"

www.patternlanguage.com: 253 patterns

www.squidoo.com/pattern-language



Origine des patterns
Travaux de Christopher Alexander



Exemple de pattern

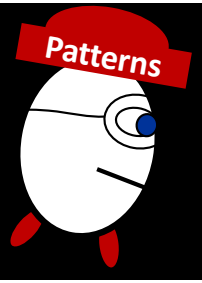
Main Entrance

Problème

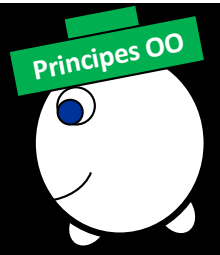
Comment définir la position et la forme de l'entrée principale d'un bâtiment (maison, immeuble, ….)

Solution

Placer l'entrée principale du bâtiment à l'endroit où elle pourra être vue immédiatement à partir des principales voies d'accès et lui donner une forme voyante et visible se détachant du bâtiment.



Origine des patterns
Travaux de Christopher Alexander



Description d'un pattern (selon C. Alexander)

Contexte

Situation de conception dans laquelle survient le problème

Problème *(forces)*

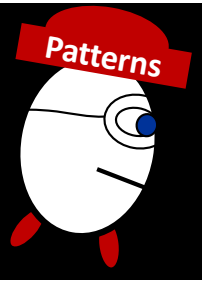
Contraintes ou besoins caractérisant le problème

Solution

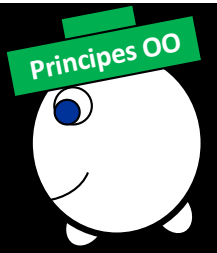
Règles permettant de résoudre le problème
(résolution des contraintes)

Conséquences

Effets résultants de la mise en pratique de la solution



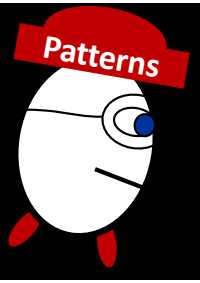
Application à l'informatique



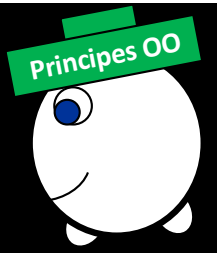
Fin 1980's

Les idées de C. Alexander ne pourraient-elles pas s'adapter à la conception de logiciels?

Réutiliser des solutions bien connues pour résoudre des problèmes bien connus



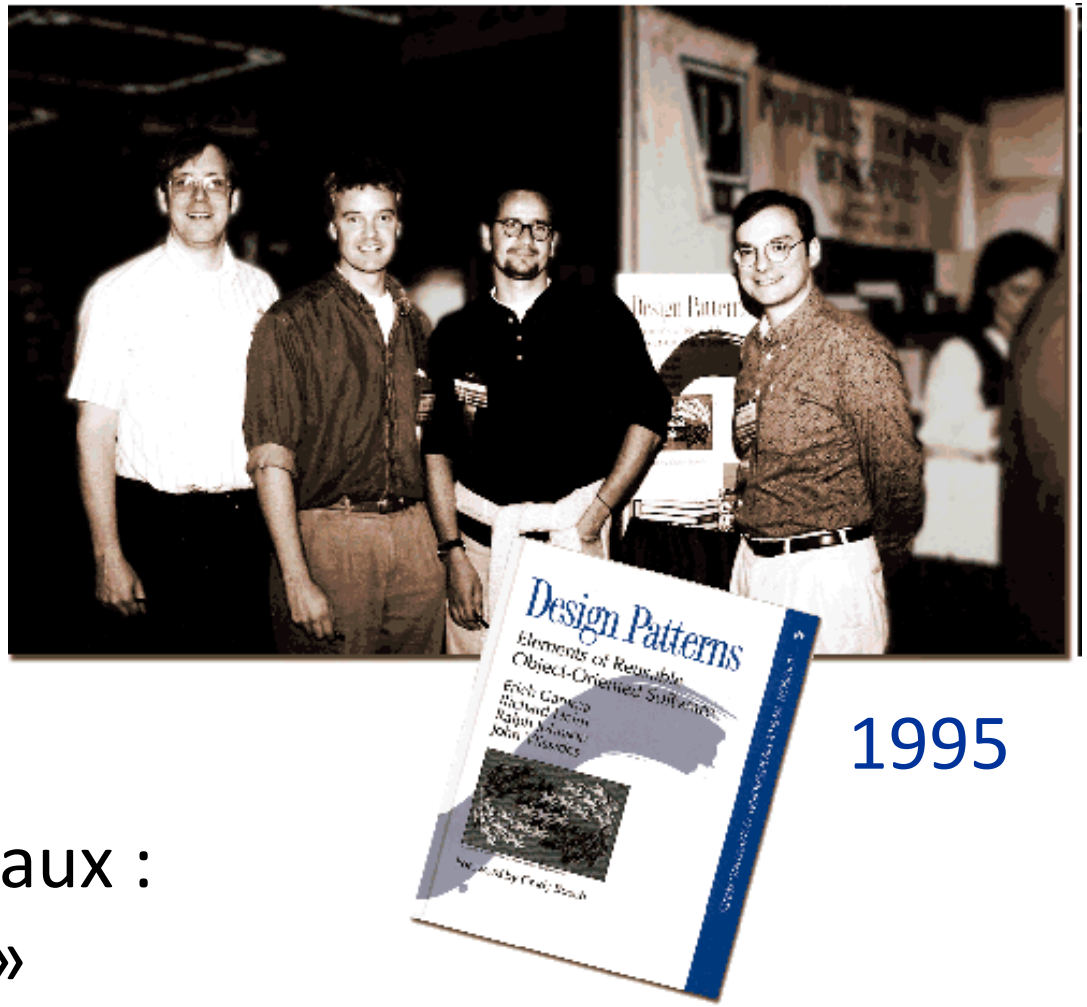
Application à l'informatique



Début 1990's : Un premier catalogue de patterns

"Gang of Four":

Erich Gamma,
Richard Helm,
John Vlissides,
Ralph Johnson

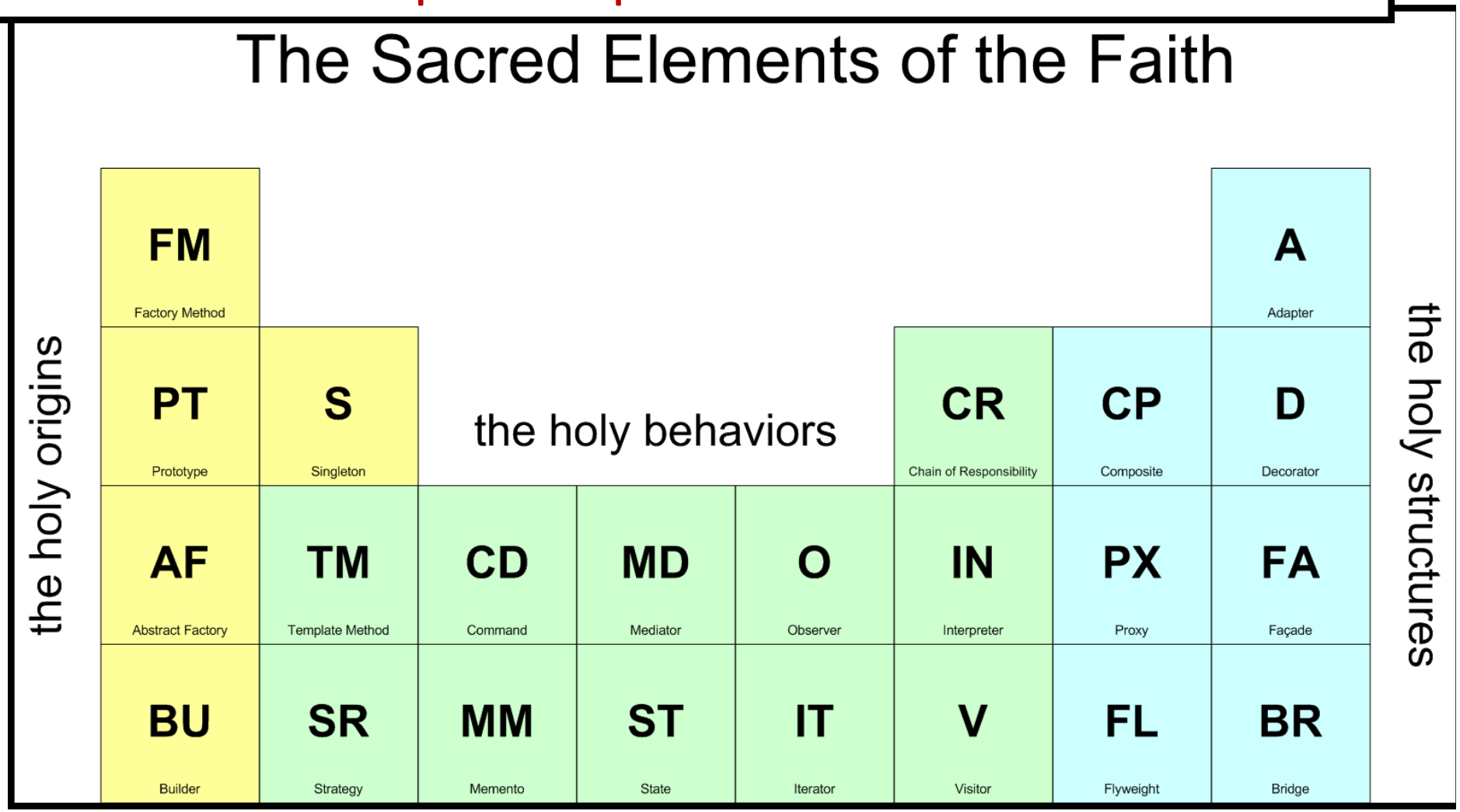


1995

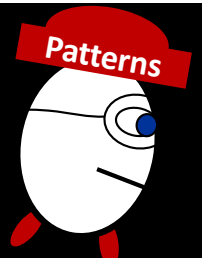
23 patterns fondamentaux :
les « patterns Gof »

Application à l'informatique

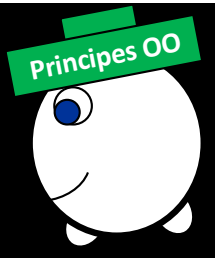
Le « tableau périodique » des 23 Patterns Gof



Source : http://home.earthlink.net/~huston2/images/periodic_table.png



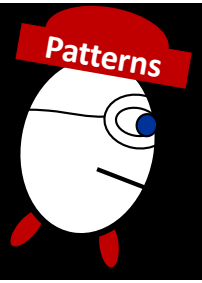
Application à l'informatique



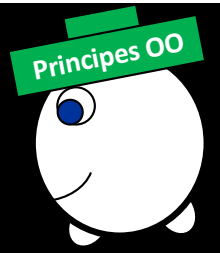
Depuis … l'approche « patterns » s'enrichit

- POSA (Pattern Oriented Software architecture) (Buschmann 1996)
- Patterns GRASP de Craig Larmann (1997)
- *General Responsability Assignement Software Patterns*
- Et bien d'autres approches …
- Bibliothèques de patterns en évolution permanente

Cf.Références



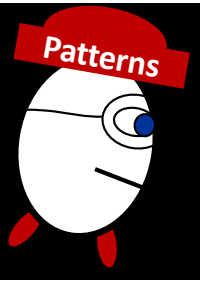
Catégories de Patterns



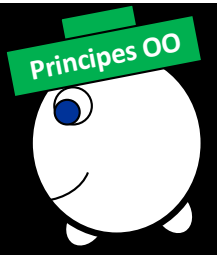
Patterns dits « structuraux »

- Patterns architecturaux
  - Structure organisationnelle globale d'un logiciel
  - Guide pour organiser les liens entre composants
- Design Patterns
  - Organisation interne des composants/classes
  - Caractéristiques d'une structure de conception
- Idiôme
  - Pattern lié à un langage de programmation spécifique

Niveau d'abstraction



CH 1 – FONDEMENTS DE L'APPROCHE « PATTERNS »

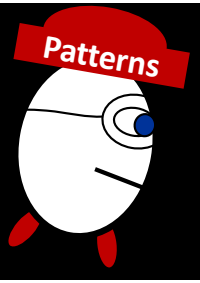


1. Historique

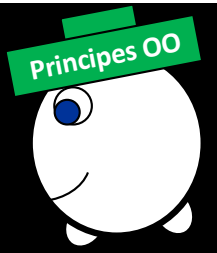
- ✓ Origine des patterns
- ✓ Application à l'informatique
- ✓ Catégories de patterns

2. Notions de base

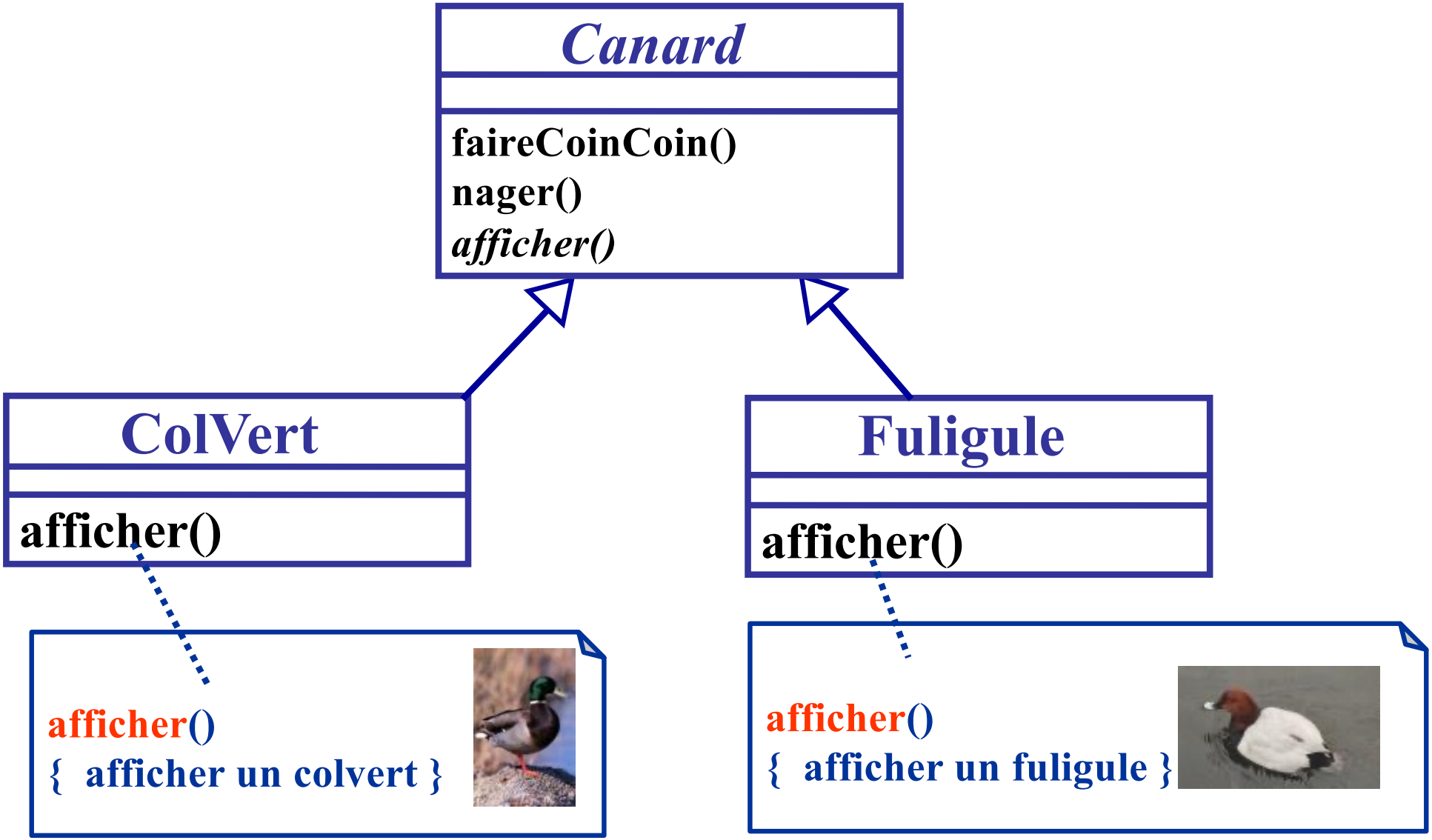
- ✓ Exemple d'introduction
- ✓ Description d'un Design Pattern
- ✓ Design Patterns et ACOO

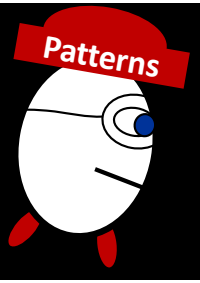


Exemple d'introduction

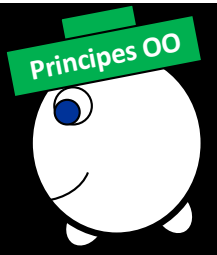


Jeu vidéo : SimUDuck *(cf. Ref : Head First Design Patterns)*

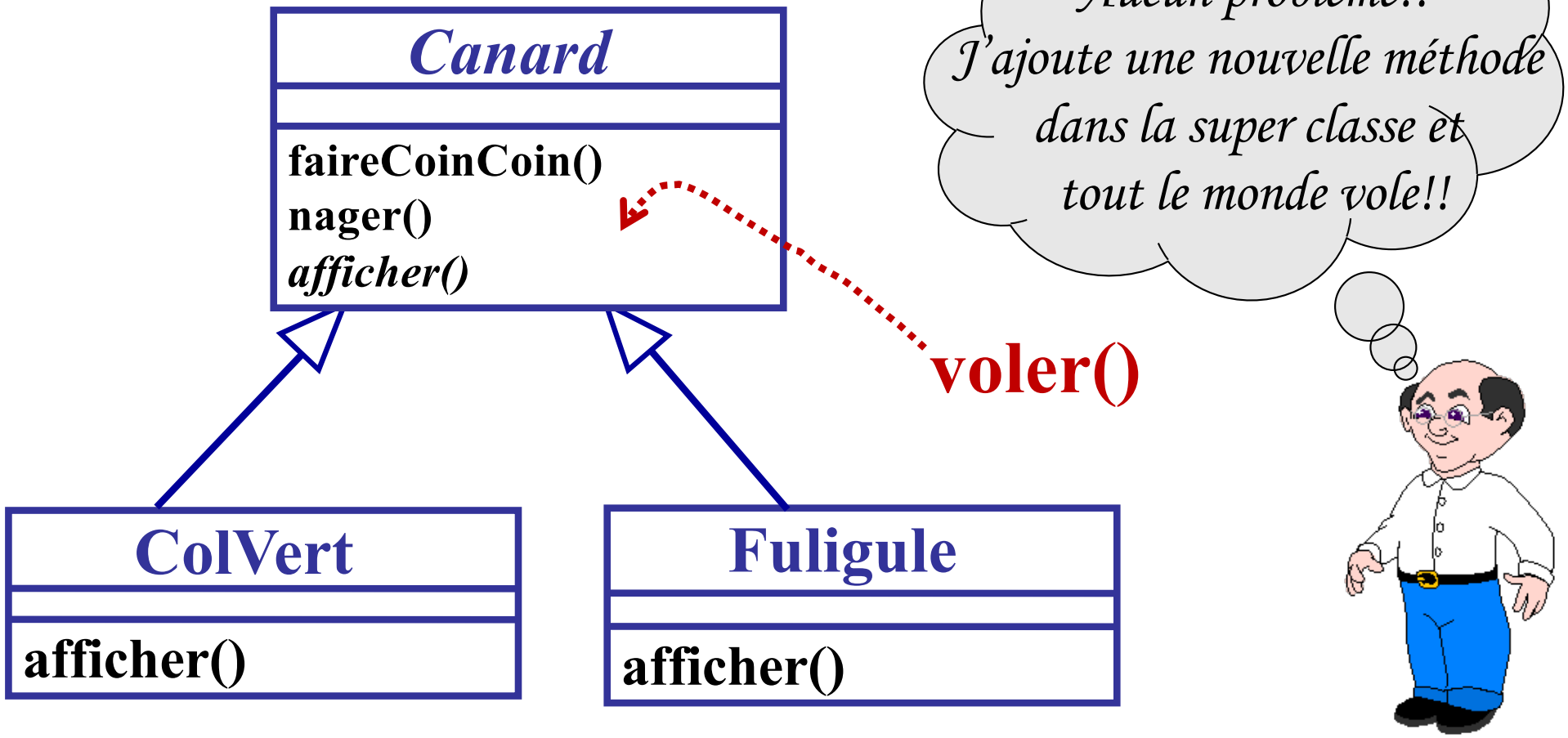


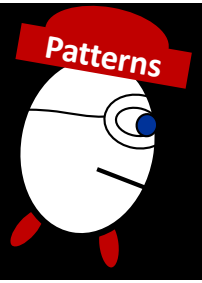


Exemple d'introduction

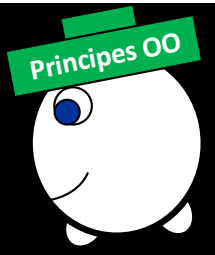


Modification N°1 : Les canards doivent savoir voler!!

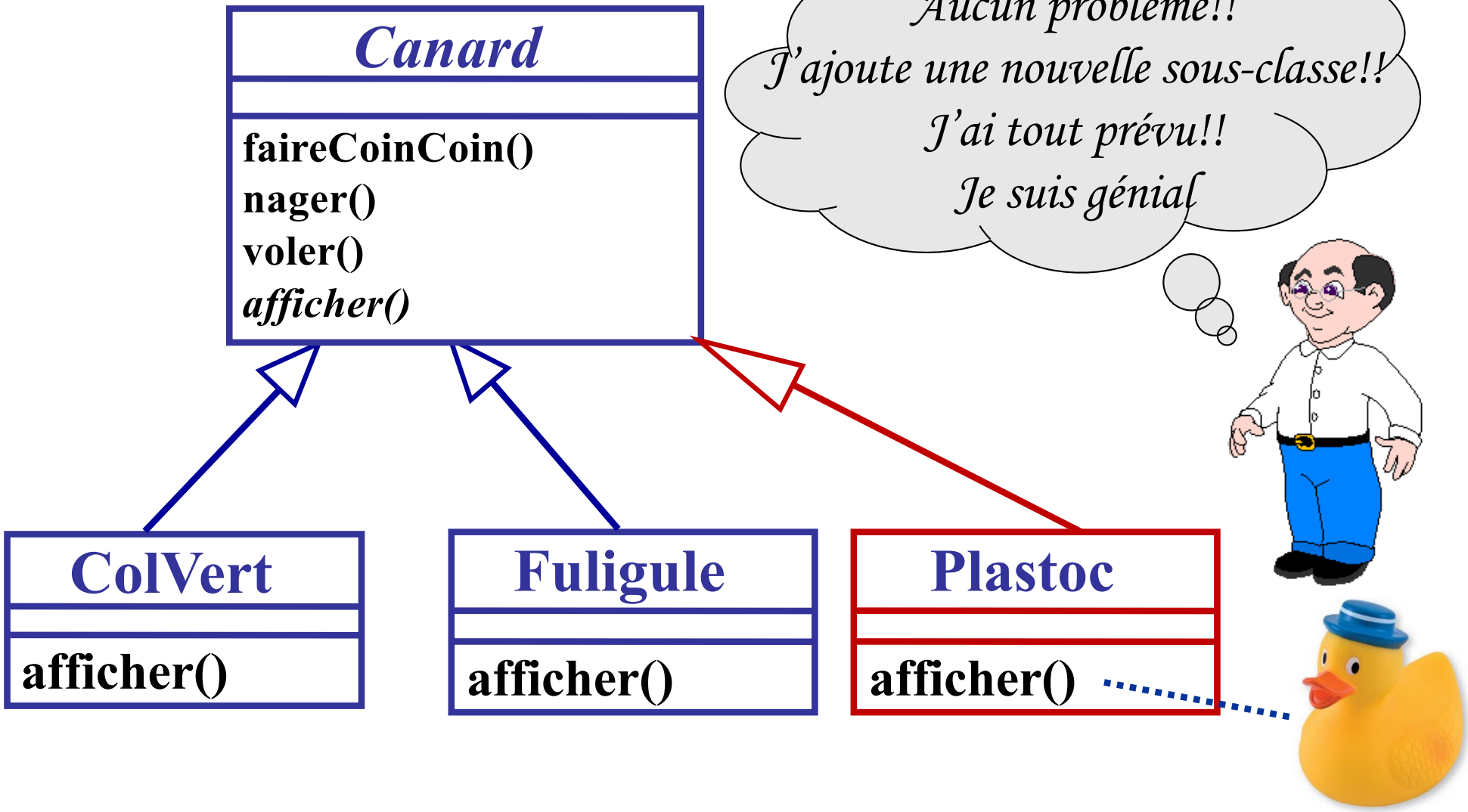


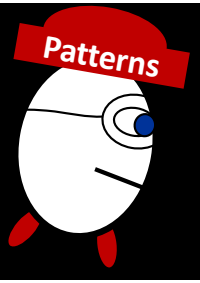


Exemple d'introduction

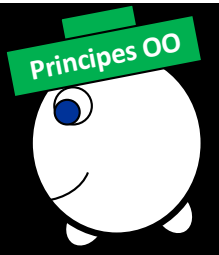


Modification N°2 : Il faut ajouter des canards « plastoc »



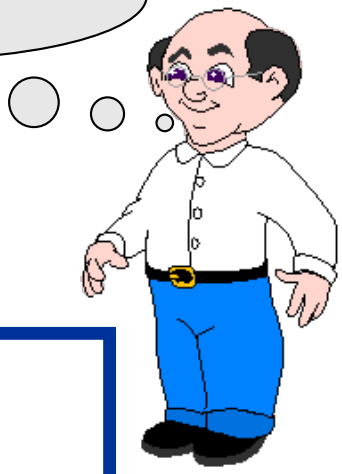


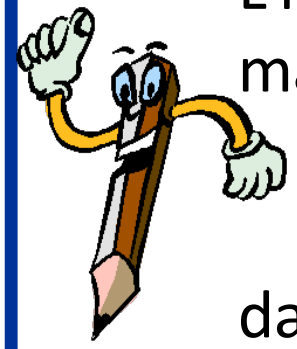
Exemple d'introduction



Problème : Plastoc ne doit pas voler et doit faire coin-coin en sifflant!!

Oups!! Je n'avais pas vu les choses sous cet angle! J'aurais du prévoir!!

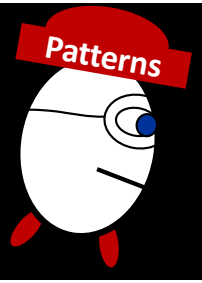




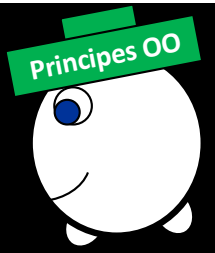
L'héritage peut conduire à des problèmes de maintenance

p1

Effets de bord d'une modification dans une super-classe



Exemple d'introduction

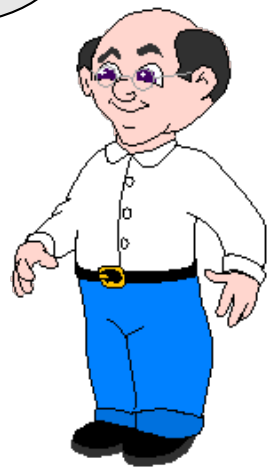


Solution N°1 : Redéfinir les méthodes héritées

Canard

faireCoinCoin()
nager()
voler()
afficher()

Bon j'ai une solution!!
Je redéfinis les méthodes
faireCoinCoin() et voler()

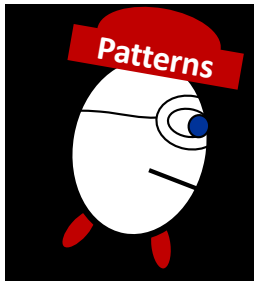


Plastoc

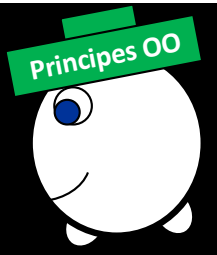
afficher()
faireCoinCoin()
voler()

Redéfinie en
« faire coin-coin en sifflant »

Redéfinie en « ne rien faire »

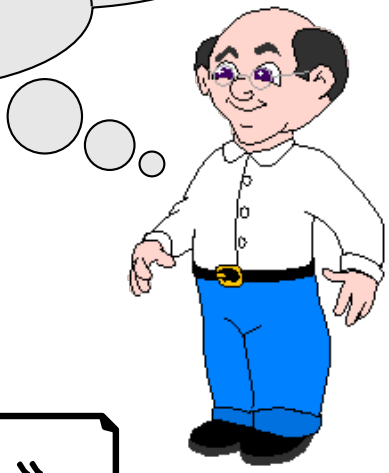


Exemple d'introduction



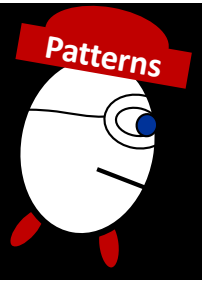
Modification N°3 : Il faut ajouter des canards « bois » qui ne font pas du tout coin-coin et qui ne volent pas!!

Euh… J'applique le même principe.
Une nouvelle classe avec rédéfinition des méthodes faireCoinCoin() et voler()

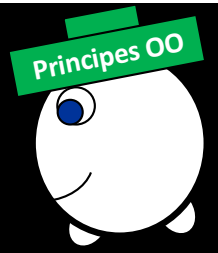


CanardBois

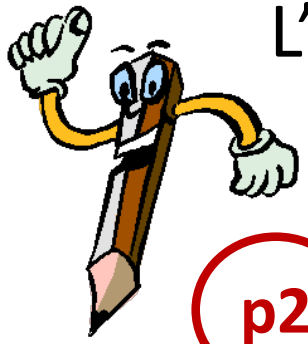
- **afficher()**
- **faireCoinCoin()** — *Redéfinie en « ne rien faire »*
- **voler()** — *Redéfinie en « ne rien faire »*



Exemple d'introduction



Problème : Et s'il y a 5O méthodes à redéfinir selon le même modèle!
N'est-ce pas un peu lourd??

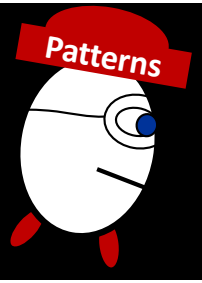


L'héritage peut conduire à une duplication de code dans les sous-classes

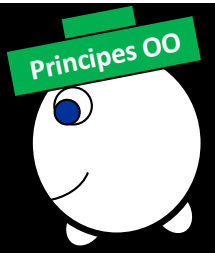
p2

Nombreuses redéfinitions obligatoires et identiques

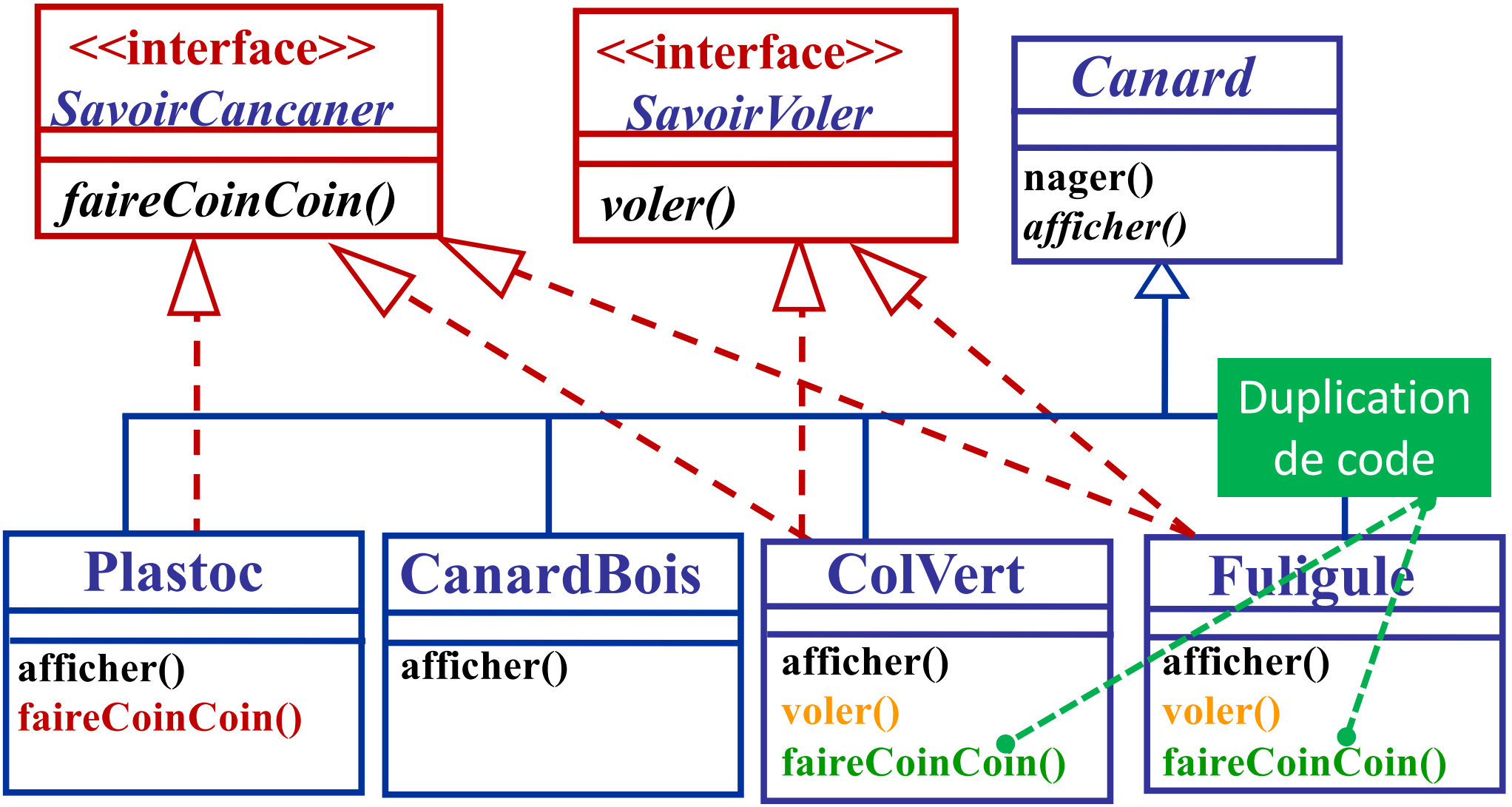
D'accord, D'accord!! L'héritage n'est pas l'idéal.
Mais là je ne vois pas d'autre solution!!

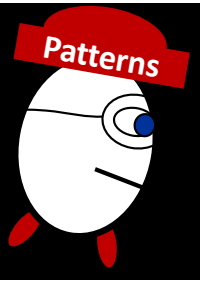


Exemple d'introduction

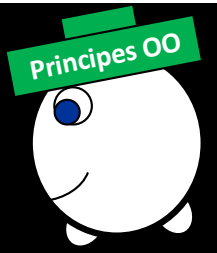


Solution N°2 : Utiliser des interfaces





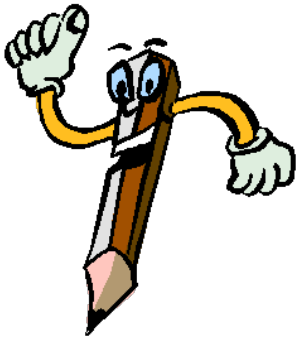
Exemple d'introduction



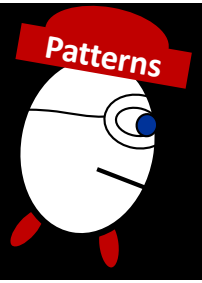
Solution N°2 : Utiliser des interfaces

OK, il n'y a pas d'effets de bord.
Mais pour la lourdeur des redéfinitions…
Vous ne résolvez rien!
Au lieu de redéfinir, vous implémentez mais c'est toujours
Lourd lourd!!

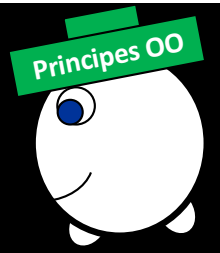
Les interfaces peuvent conduire à une duplication de code dans les classes d'implémentation



P2' Nombreuses implémentations identiques



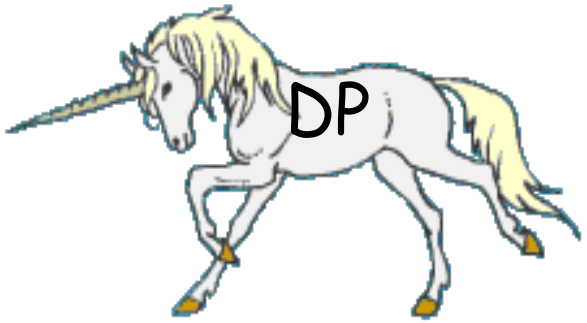
Exemple d'introduction



Conclusion

Héritage ou Interfaces, on a toujours des problèmes en cas de changement de comportements!

Il faut anticiper les changements!



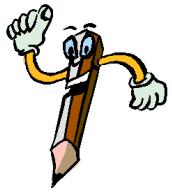
Trouver une solution qui minimise l'impact sur le code existant

Objectif des Design Patterns

PRINCIPES DE CONCEPTION

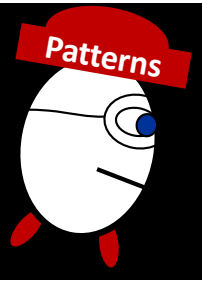
Identifier les parties « changeantes» de l'application et les isoler du reste

« Encapsulates what varies »

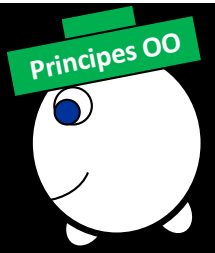


Programmer à travers des interfaces (ou classes abstraites) et non des implémentations (classes concrètes).

« Program through an interface not to an implementation »



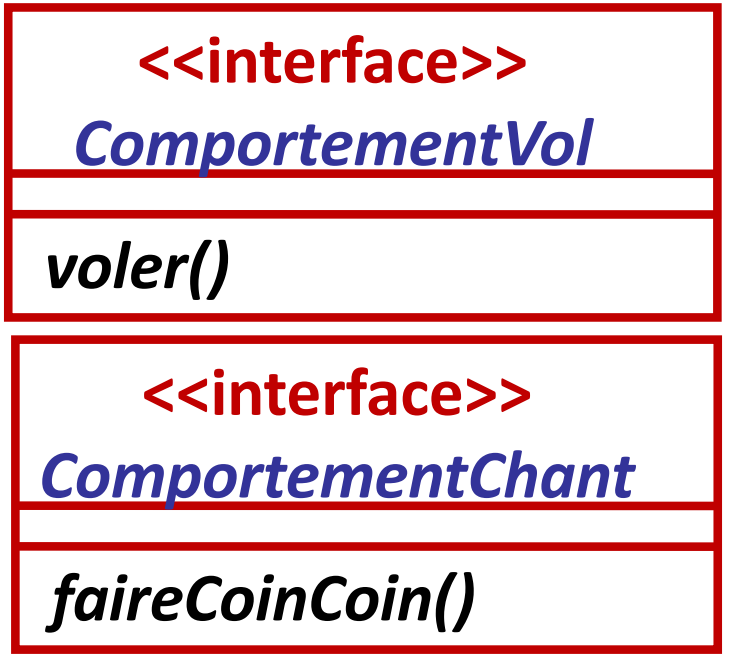
Exemple d'introduction

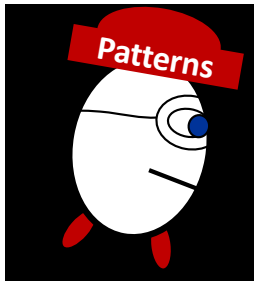


Parties variables du comportement du canard

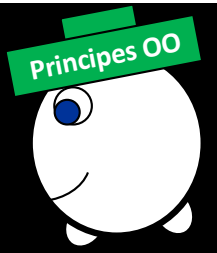
voler (*ComportementVol*)
faireCoinCoin *(ComportementChant)*

On isole ces comportements dans des interfaces

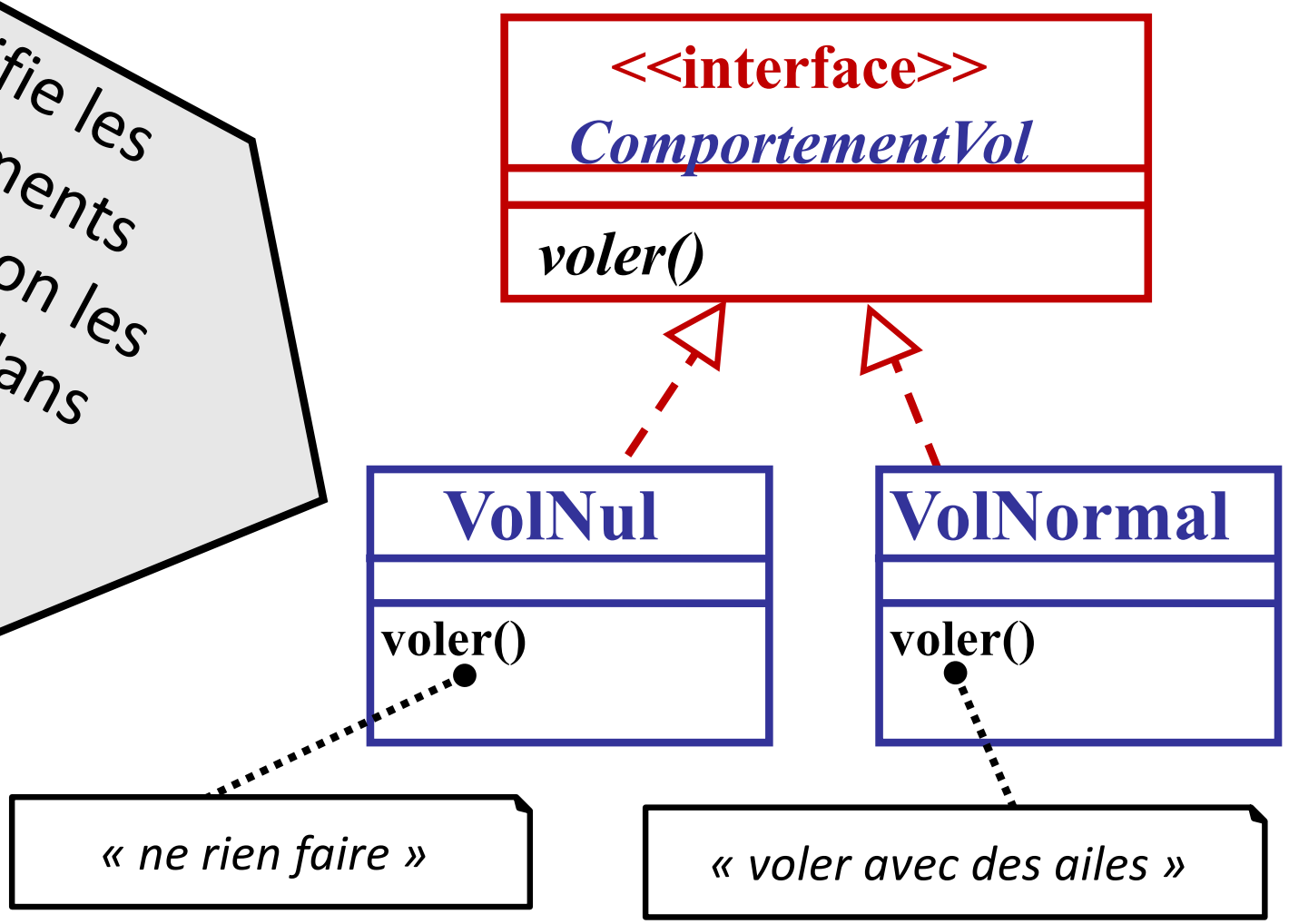




Exemple d'introduction

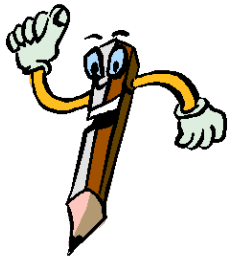


On identifie les comportements répétitifs et on les implémente dans des classes séparées



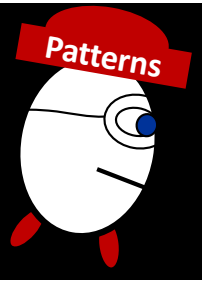
PRINCIPES DE CONCEPTION

Principe d'inversion des dépendances

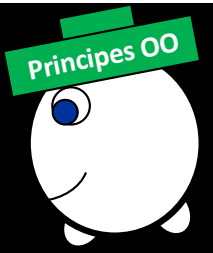


Lier les classes « pérennes » (non changeantes) à des interfaces (ou classes abstraites) et non à des classes concrètes

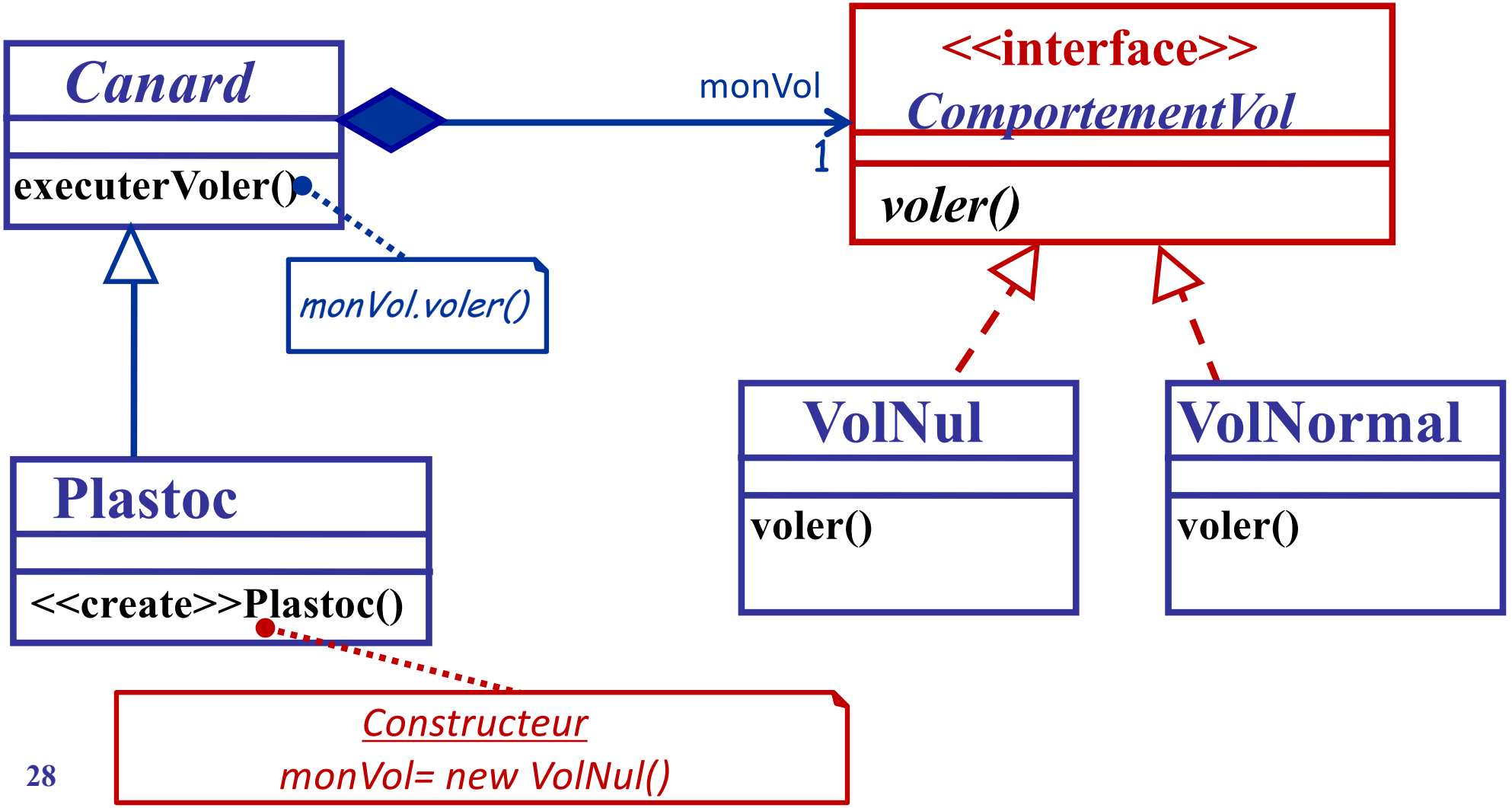
«Depend upon abstractions. Do not depend upon concrete classes.»

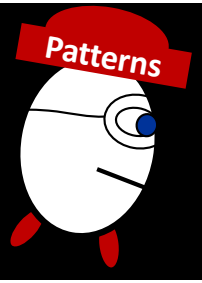


Exemple d'introduction

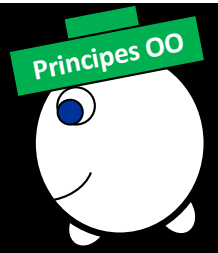


On relie la classe racine aux interfaces de comportements



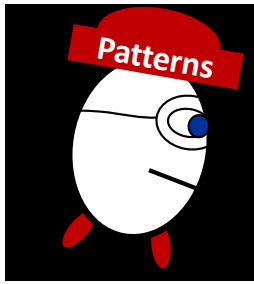


Exemple d'introduction



- Scénario de test
  - Créer un canard plastoc
  - Le faire voler
  - Changer son comportement de vol
  - Le faire voler à nouveau

```
Class TestJeu{
  public void test(){
     Canard c=new Plastoc();
     c.executeVoler();
     ComportementVol v=new VolNormal();
     c.changerVol(v);
     c.executerVoler();}
}
```



Exemple d'introduction

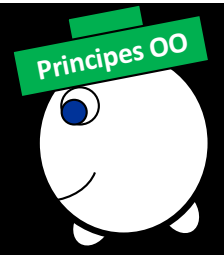
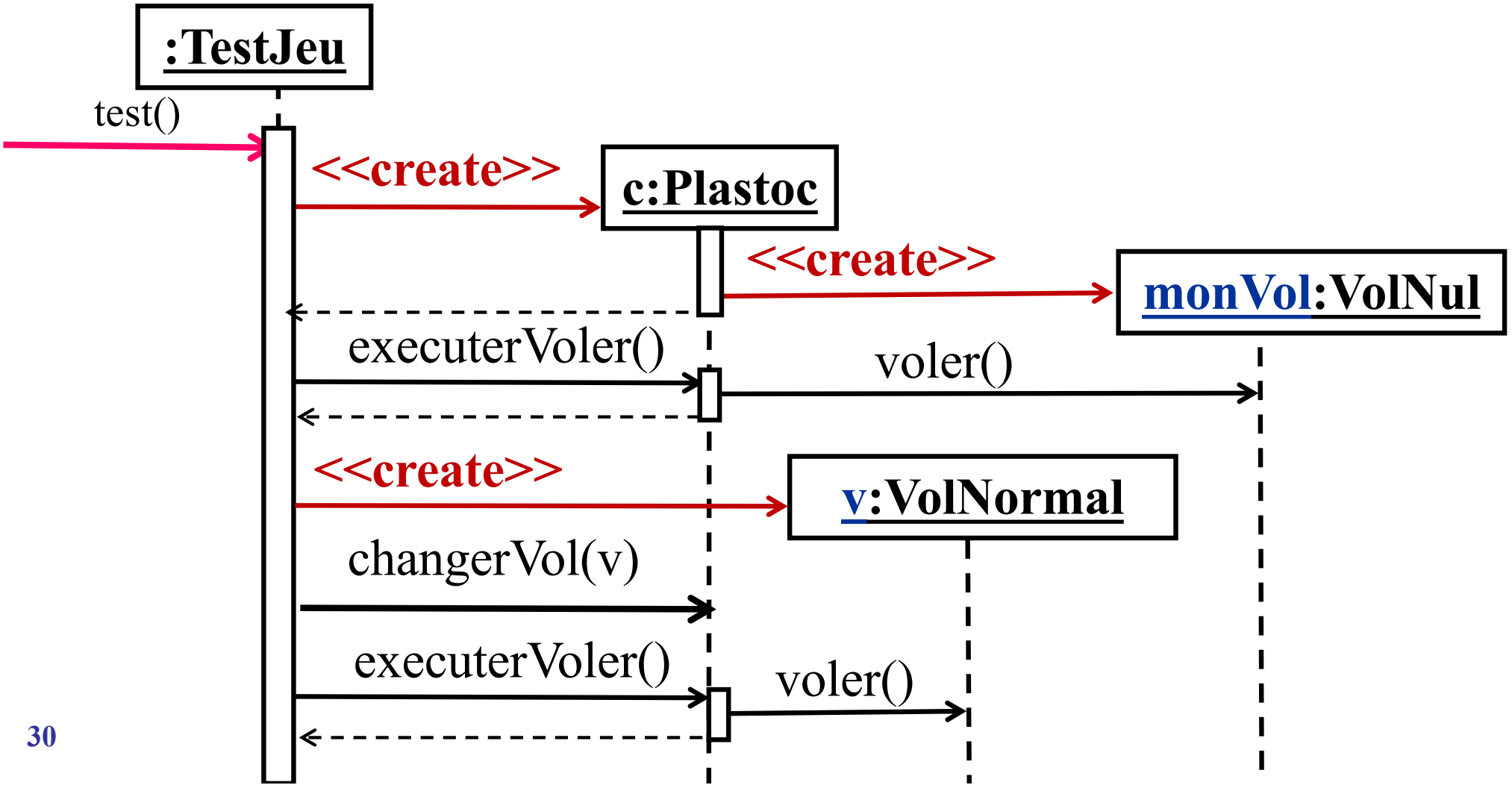
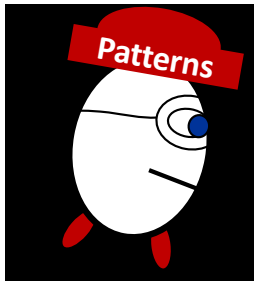


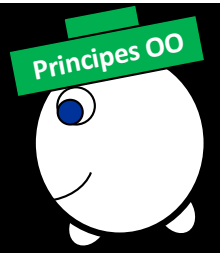
Diagramme de séquence

Exécution de la méthode test sur un objet de la classe TestJeu





Exemple d'introduction



Bilan sur la solution finale

Nouveau canard

- Nouvelle sous-classe de Canard
- Initialisation du comportement dans le constructeur

Résolution des problèmes

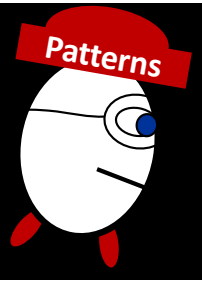
P1 Pas d'héritage de méthode non souhaité

P2 Pas de duplication de code

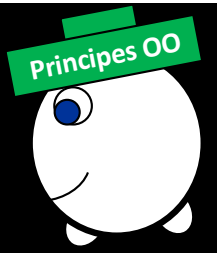
Pas de redéfinition/ré-implémentation de méthode Si le comportement est déjà défini (p2)

Si le comportement n'existe pas, ajout d'une nouvelle classe de comportement (exemple: Vol supersonique)

++ Le comportement d'un canard peut évoluer dynamiquement

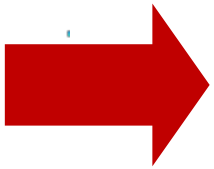


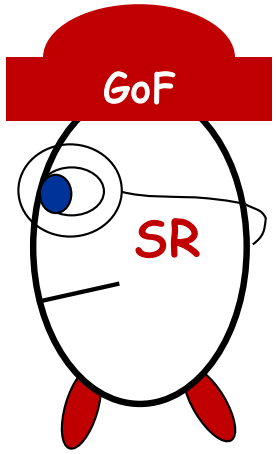
Exemple d'introduction

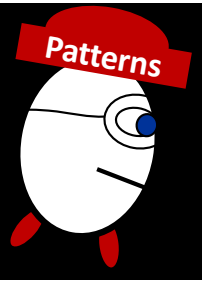


Bilan sur la solution finale

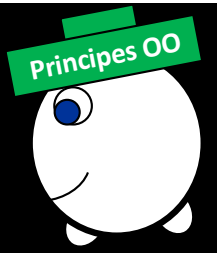
Le Design Pattern Stratégie nous a donné la solution



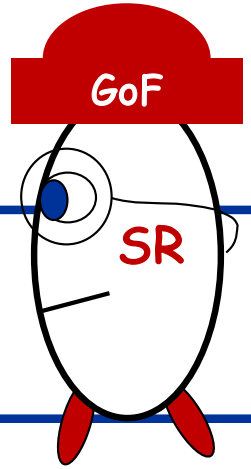




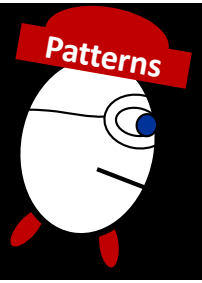
Design Pattern STRATEGY



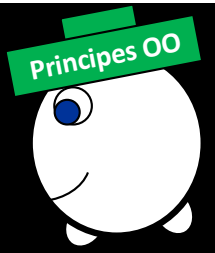
Description du pattern Strategy



Nom	Strategy
Problème	La sélection d'un algorithme applicable sur un objet Context dépend de sa nature et peut changer dynamiquement.
Solution	Définir chaque algorithme/politique/stratégie dans une classe distincte mais avec une interface commune

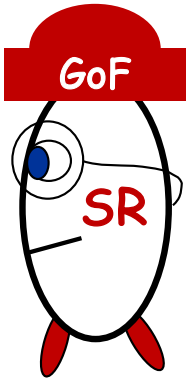


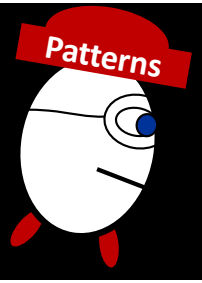
Design Pattern STRATEGY



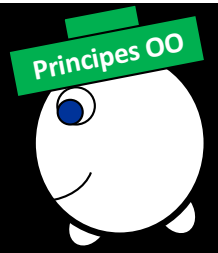
Description du pattern Strategy

Diagramme de classe UML



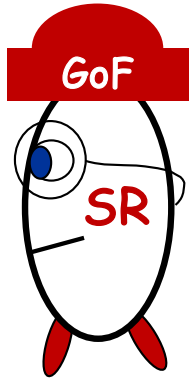


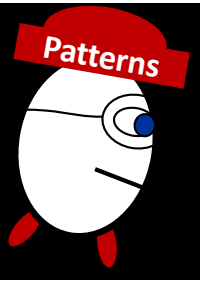
Design Pattern STRATEGY



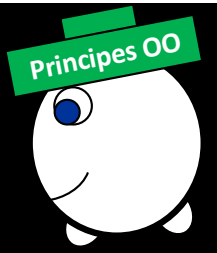
Description du pattern Strategy

Diagramme de classe UML (variante avec classe abstraite)



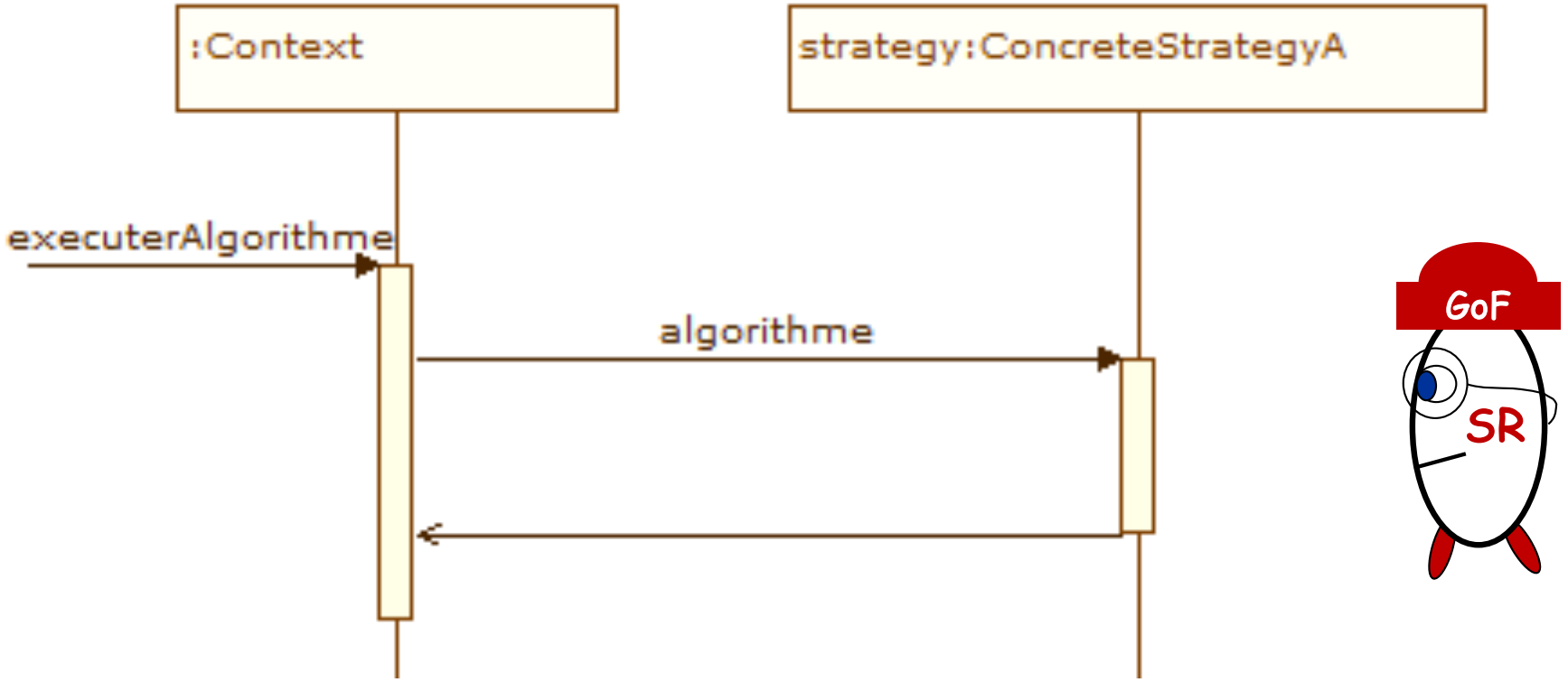


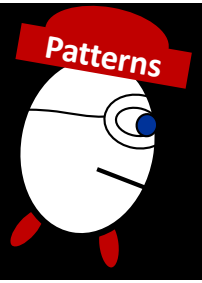
Design Pattern STRATEGY



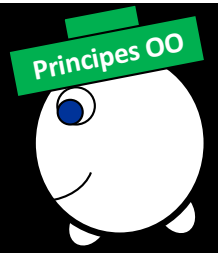
Description du pattern Strategy

Diagramme de séquence UML





2 – Notions de base

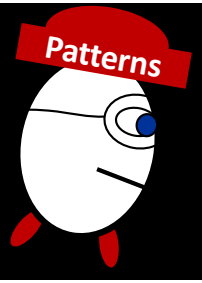


Qu'est ce qu'un Design Pattern?

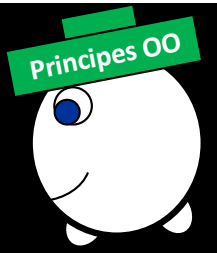
- Une solution à un problème récurrent de conception
- Pas une solution "concrète" mais un schéma ou squelette de solution générique
- Un guide d'application des «best practices» de l'orienté objet

« Nouveau pattern » est un oxymore!

Craig Larman



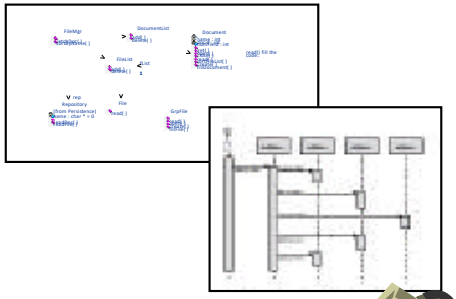
Description d'un Design Pattern

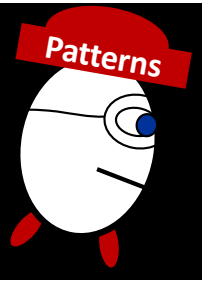


Comment décrire un Design Pattern?

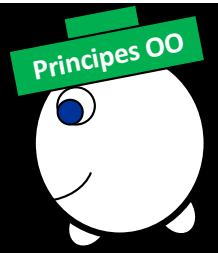
Description de base

Nom	Nom identifiant de manière unique le pattern
Problème	Problème résolu par le pattern
Solution	Solution apportée dans un contexte donné
Diagrammes UML	Diagramme(s) de classes *[Diagrammes de séquence]*
Exemple	Exemple concret d'application du pattern





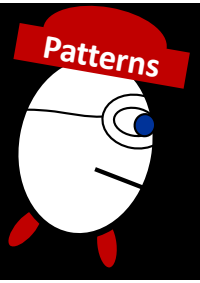
Description d'un Design Pattern



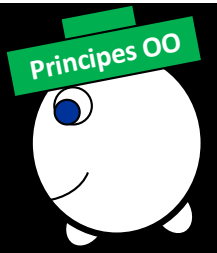
Comment décrire un Design Pattern?

Description détaillée

Participants	Entités (classes, interfaces,..) impliquées
Conséquences	Résultats positifs et négatifs de l'application du pattern
Implémentation	Mise en œuvre concrète dans un langage



Description d'un Design Pattern

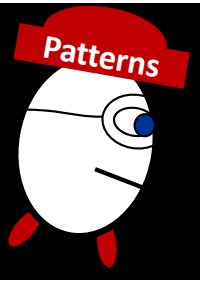


Comment décrire un Design Pattern?

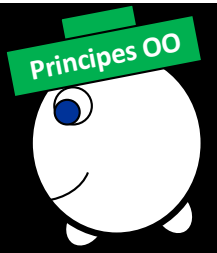
Exemple de format de description d'un pattern

(site d'échange de patterns de Microsoft: http://patternshare.org/)

Context: The context in which the pattern emerges, expressed in a few sentences.
Problem: The essence of the problem, expressed as a single question.
Solution: The solution stated in a few sentences, plus a picture.
Related Patterns: A list of patterns that are closely related to this one.



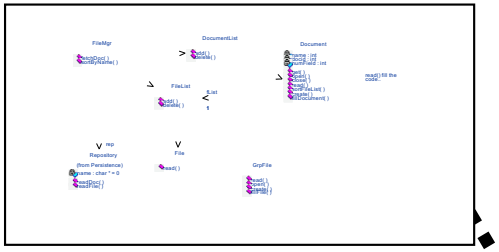
Langage de patterns



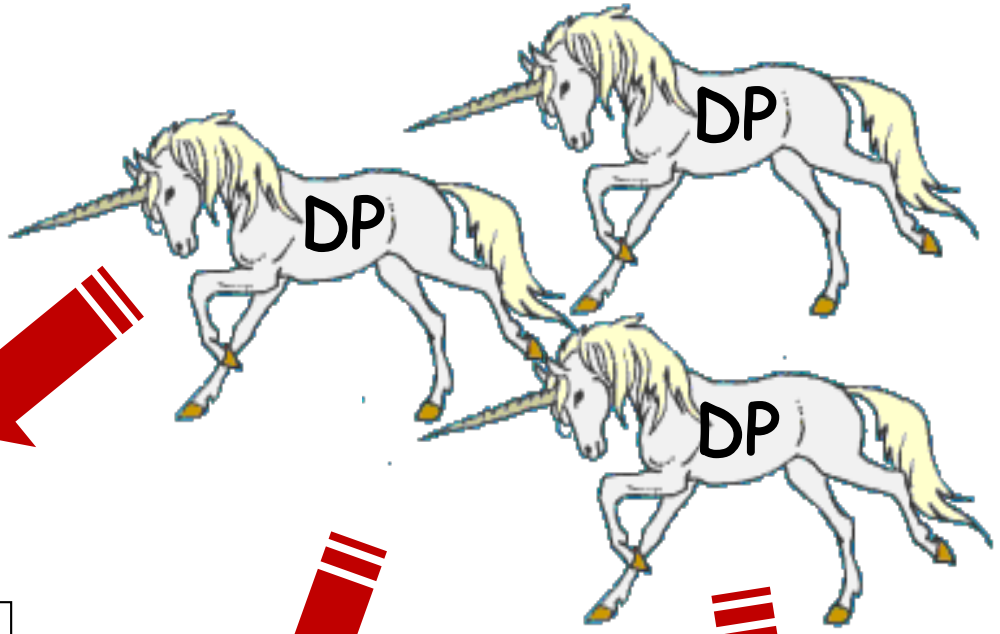
- Utilisation/combinaison de plusieurs patterns
- Vocabulaire universel connu de l'ensemble des développeurs.

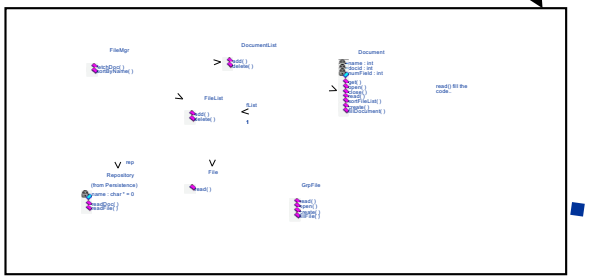
Je suggère une **Stratégie** générée à partir d'une **Fabrique Abstraite** pour prendre en charge la **Protection des variations** et le **Faible Couplage**.
Qu'en pensez-vous?

Quand utiliser des Design Patterns?



Diagrammes de classes du domaine

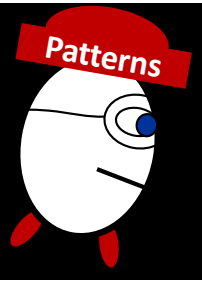




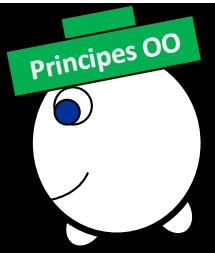
Diagrammes de classes de conception

Programmes

Refactoring



Design Patterns et ACOO



Comment savoir utiliser les Design Patterns?

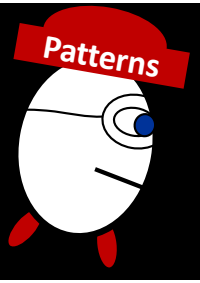
Etude des Principes (ou Patterns Grasp)

Maîtriser les principes OO de base

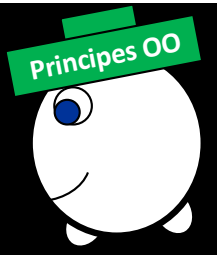
Etude des Patterns GOF

- Savoir reconnaître les problèmes de conception
- Savoir identifier le(s) DP à utiliser
- Savoir utiliser un DP

Ne pas succomber à la « pattern mania » !!



Références



Ouvrages

- Head First Design Patterns

  Eric Freeman, Elisabeth A. Freeman,

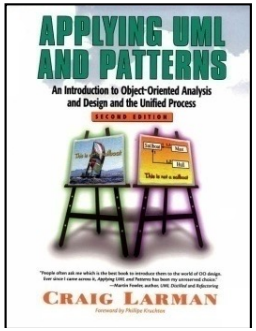
  Kathy Sierra

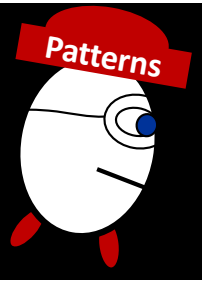
  Editions O'Reilly Media

- UML 2 et les Design Patterns

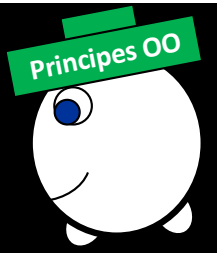
  Craig Larman, M.- C. Baland, L. Carité, E. Burr

  Editions Pearson Education





Références

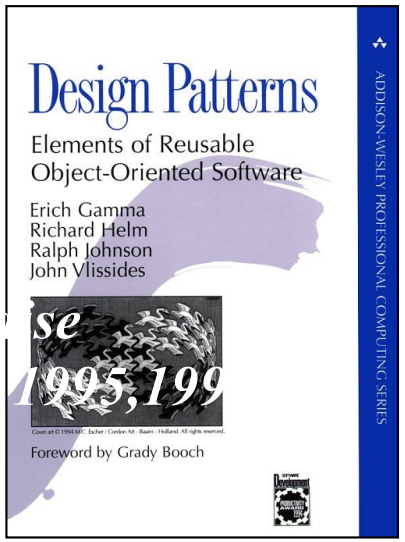


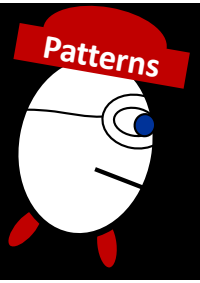
Ouvrages

- Design patterns. Catalogue des modèles de conception réutilisables

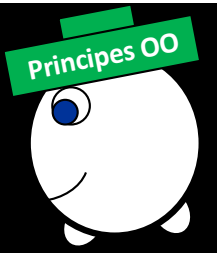
  Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides

  Editions Vuibert informatique (1999)





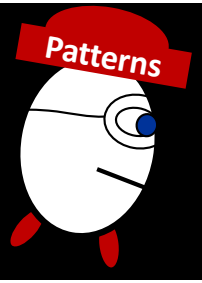
Références



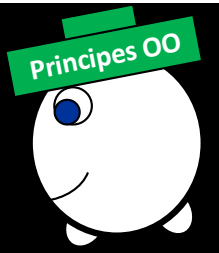
Sites

- Site de référence entièrement consacré aux “software patterns” hillside.net/patterns/
- Le site de Brad Appleton sur les patterns www.cmcrossroads.com/bradapp/docs/patterns-intro.html
- Le site de Vince Huston sur les patterns home.earthlink.net/~huston2/dp/patterns.html
- Un catalogue de patterns avec les diagrammes UML associés

  www.dofactory.com/Patterns/Patterns.aspx#list



Références



Sites

- Des articles/interviews d'Erich Gamma (cf. GoF) sur l'utilisation des patterns et les principes de conception
  - ✓ www.artima.com/lejava/articles/gammadp.html
  - ✓ www.artima.com/lejava/articles/reuse.html
  - ✓ www.artima.com/lejava/articles/designprinciples.html

- Une présentation trés claire des cinq premiers patterns Grasp de Craig Larman
  - ✓ wpetrus.developpez.com/uml/grasp/grasp.pdf