

# Chap.7 – Bien concevoir

## Principes SOLID

**V. Deslandres** ©

Licence Professionnelle SIL option DevOps

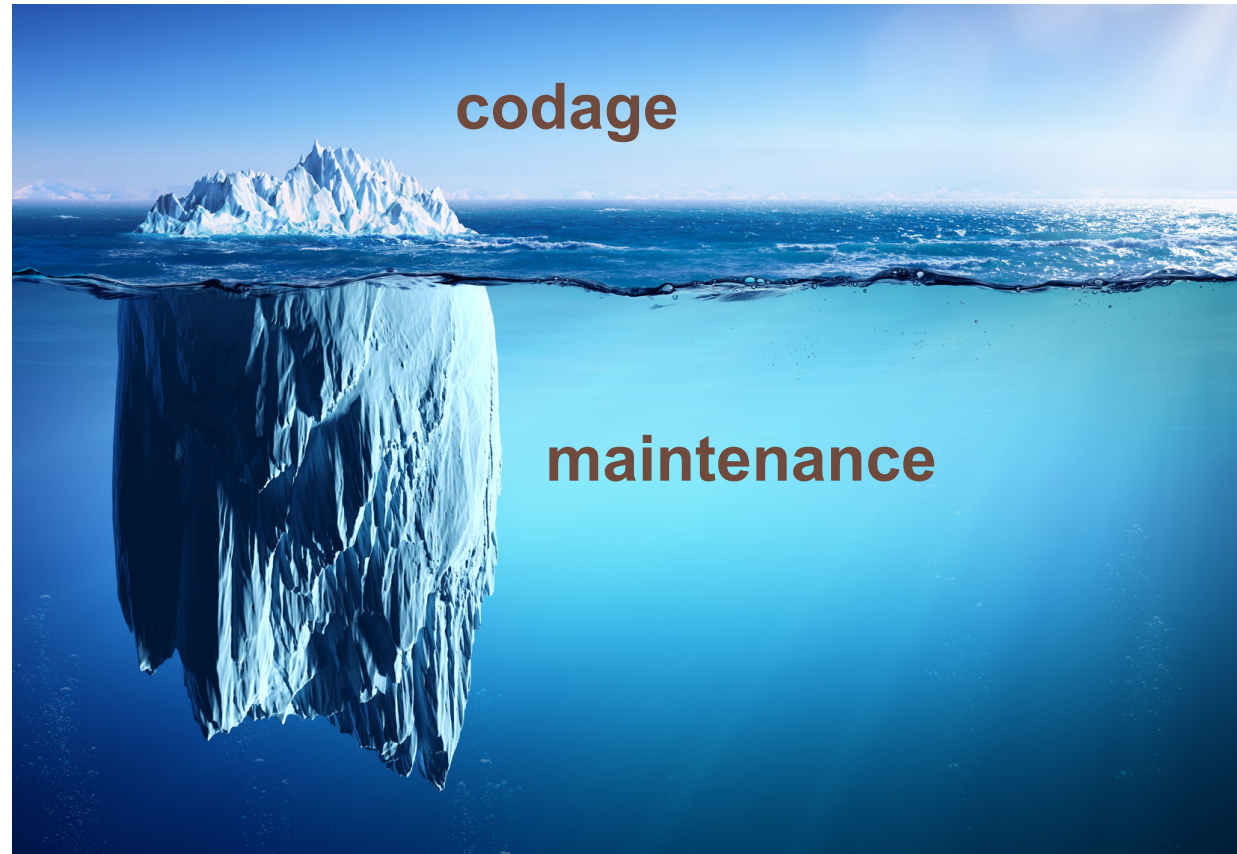
IUT de Lyon - Université Lyon 1

# Pourquoi bien concevoir, est-ce important ?

Concevoir un objet non pas pour ÊTRE, mais qui pourra CHANGER, être REPARÉ.

Conception = définir un logiciel, qui est vivant.

Formaliser des choses non définitives. Il faut toujours anticiper leur possible évolution.



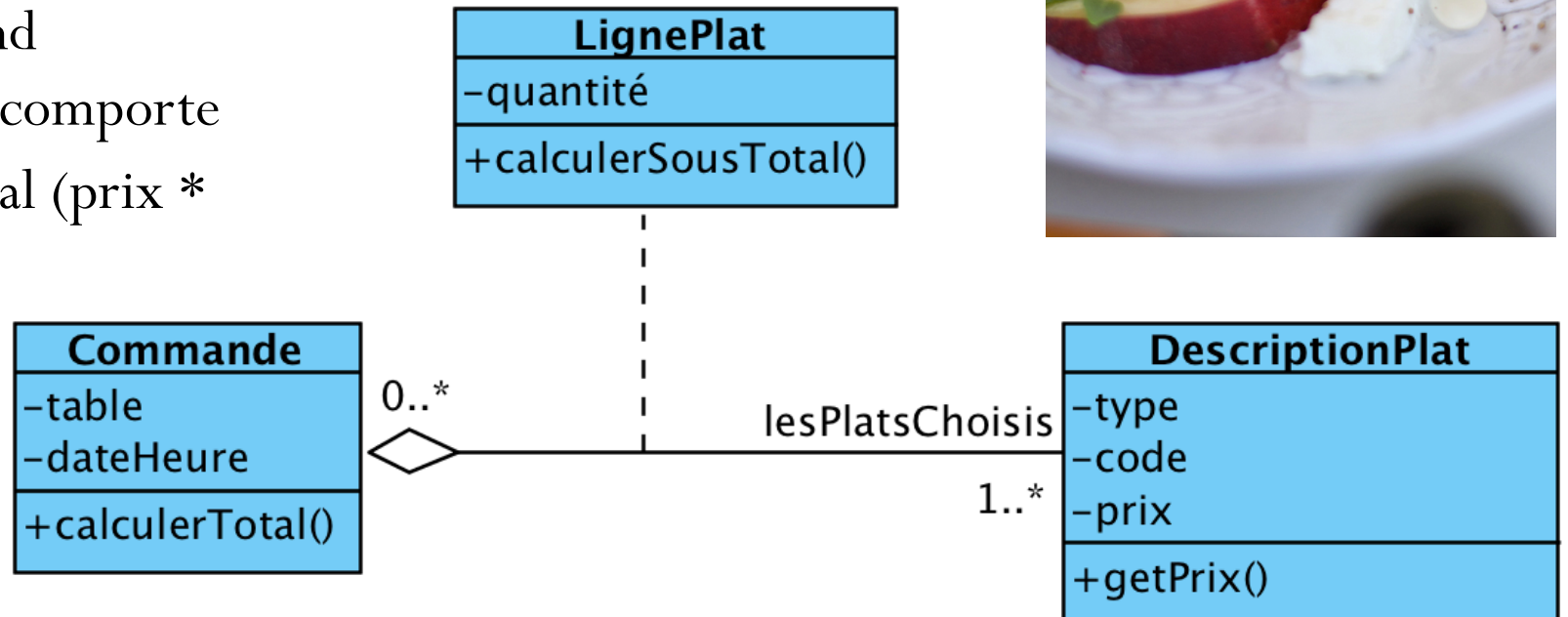
# Sommaire de ce cours

- Concepts de base : contrat, service, cohésion, couplage
- Principes élémentaires de Conception ----- #[14](#)
  
- Les Principes SOLID
  - Responsabilité unique (SRP) ----- #[23](#)
  - Ouverture/Fermeture de code (OCP)----- #[27](#)
  - Substitution de Liskov (L) ----- #[32](#)
  - Séparation des Interfaces (ISP) ----- #
  - Inversion des Dépendances (IoC) ----- #[36](#)

# Concepts de base (1)

d'une bonne conception

- Les **responsabilités** d'une classe :
  - Ce qu'elle **SAIT**
  - Ce qu'elle est capable de **FAIRE**
- Ex. pour la **classe LignePlat** suivante :
  - Elle sait : à quel objet Commande elle appartient, et à quel objet DescriptionPlat elle correspond
  - Elle sait combien de plats elle comporte
  - Elle peut calculer son sous-total (prix \* quantité de plats)

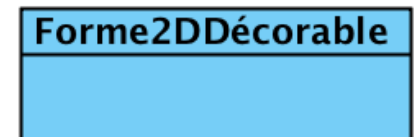
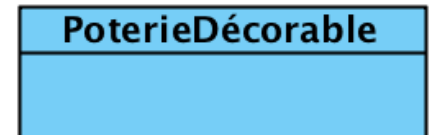
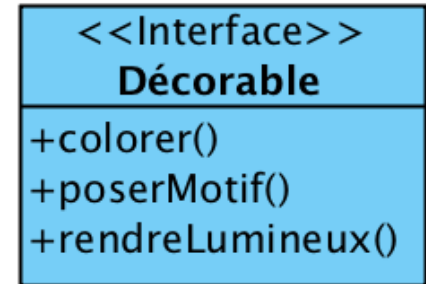


# Concepts de base (2)

- Le **contrat** = services rendus par une classe
  - Exprimé par les opérations de classe ou d'interface
  - Stable
  - Masque les détails de réalisation
  - Syn.: « comportement »
- **L'implémentation**
  - Représente les classes concrètes
  - Peut évoluer
- Toujours chercher à **bien les dissocier**

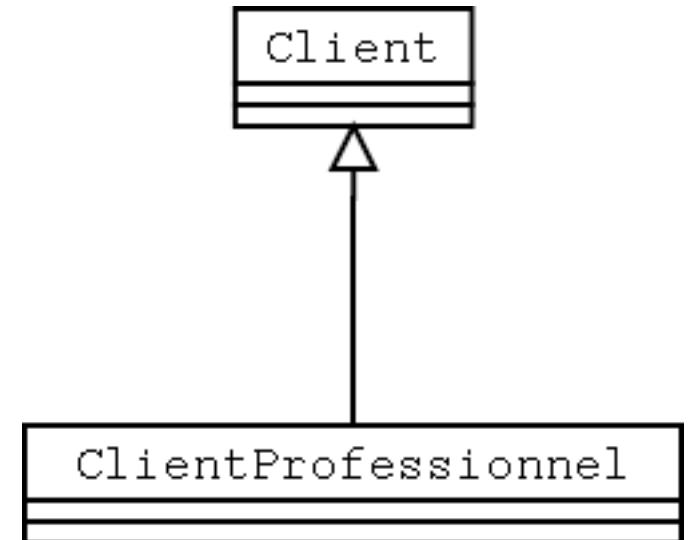
Ensemble des  
méthodes  
(signatures)

Les différents  
codes associés  
aux méthodes



# Rappel : Interface vs. classe abstraite

- Similaires (méthodes abstraites) mais différentes (objectifs)
- **Avantages d'une interface** : abstraction complète, héritage multiple d'interfaces, nouvelle implémentation possible à tout moment
  - Analogie : prise de courant = moyen d'obtenir du courant alternatif; pour faire quoi ?
- **Avantages d'une classe abstraite** : définition partielle possible, héritage simple qui permet de spécifier des comportements
  - Ex. : le Client



À partir de Java 8, on peut ajouter 2 éléments dans une interface :  
des **méthodes statiques** et des **méthodes par défaut**  
(**publiques**)



**Module** en COO = une classe, un package, une méthode, un composant physique

# Concepts de base (3) cohésion / couplage

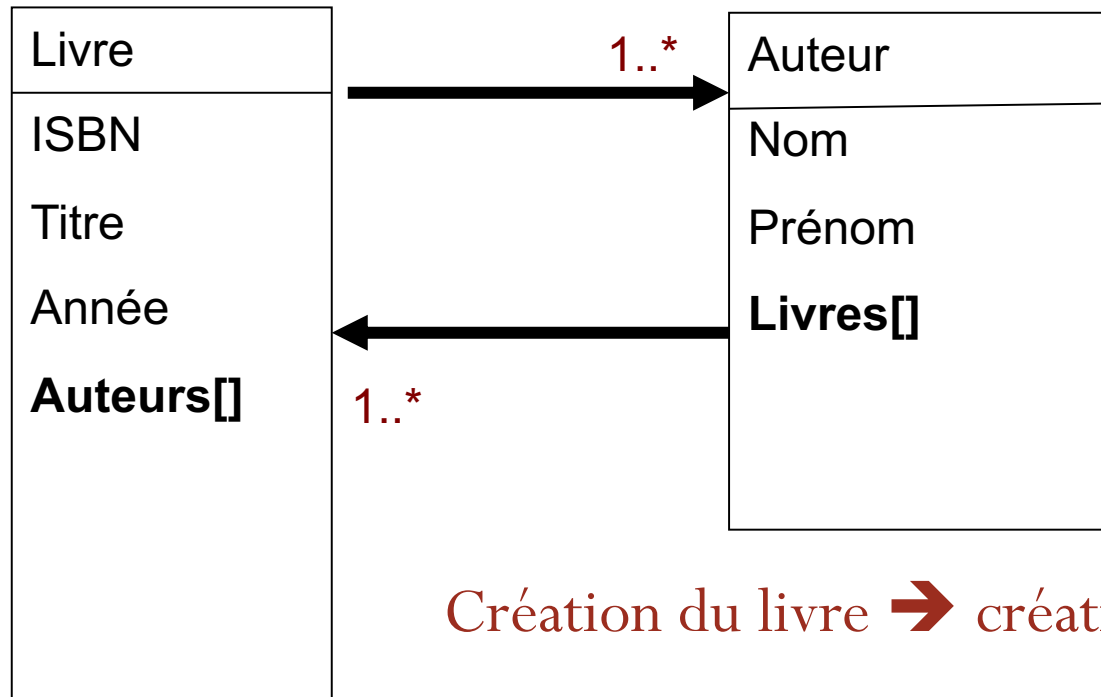
**COHESION** = Esprit de famille

- Degré avec lequel les tâches d'un module sont fonctionnellement reliées entre elles
- Quel est le liant d'un module ?
- Quel est son objectif ?
- Fait-il une ou plusieurs choses ?
- Quelle est sa fonction au sein du système ?
  
- Au sein d'une classe : cohérence des relations entre **méthodes et attributs**

**COUPLAGE** = Dépendance

- Force de l'interaction entre les modules d'un système
- Comment les modules travaillent ensemble ?
- Qu'ont-ils besoin de savoir l'un sur l'autre ?
- Quand font-ils appel aux fonctionnalités de chacun ?
  
- Ex.: une classe qui **crée une instance d'une autre classe** = fort couplage
  - Ne peut pas être testée indépendamment de l'autre classe

# Ex. Couplage fort Livre / Auteur



Création du livre → création de l'auteur

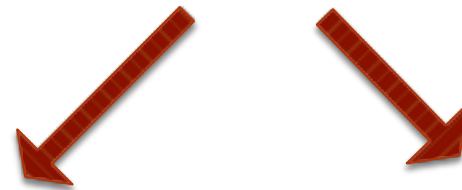
Il y aura toujours des **dépendances** entre certains modules de l'application : l'important est de les identifier et les rendre bien visible. Un fort couplage n'est pas dramatique avec des **composants stables** (ex. java.util).





Quels sont les  
inconvénients d'une  
**faible cohésion** ?

Que penser d'une classe avec 100 méthodes  
et des milliers de lignes de codes ?



***Différents domaines  
sont couverts***

***Manque certain de cohérence  
entre les variables, les  
traitements***

# Faible cohésion



- Une **cohésion** médiocre altère :
  - la compréhension,
  - la réutilisation,
  - la maintenabilité
- Le code est fragile
  - Il subit **toute sorte de changements** très fréquemment, comme il est très vaste
    - Trop d'objets ont besoin de lui



Inconvénients d'un  
trop fort **couplage** ?

# Fort couplage



- Un couplage trop élevé ? Une **assiette de spaghettis**
  - Maintenance difficile, voire impossible
  - Lisibilité faible
- Parfois volontaire : principe d'**obfuscation**
  - Code rendu illisible pour protéger ses sources de rétro-ingénierie

# Forte cohésion : quelques règles

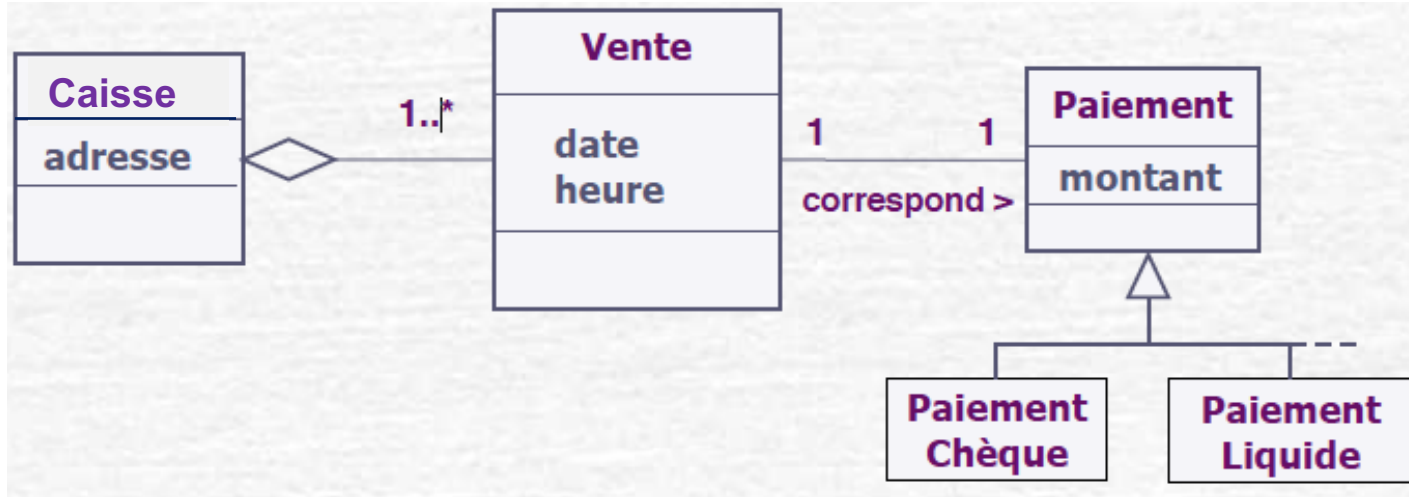
- Regrouper les éléments **en forte relation**
- Regrouper les classes qui rendent des **services de même nature** aux utilisateurs
- Isoler les **classes stables** de celles qui risquent **d'évoluer** au cours du projet
- Isoler les classes **métiers** des classes **applicatives**
- Distinguer les classes dont les objets ont **des durées de vie différentes**



# Faible couplage : les règles

- Préférer s'adresser à une **interface**, pas à une **implémentation**
  - Via l'interface, le client ne sait pas quelle sera l'implémentation, il sait juste ce qu'il *peut* demander de faire à la classe
  - **Couplage** plus faible
- Ne pas ajouter plus de dépendances que nécessaire

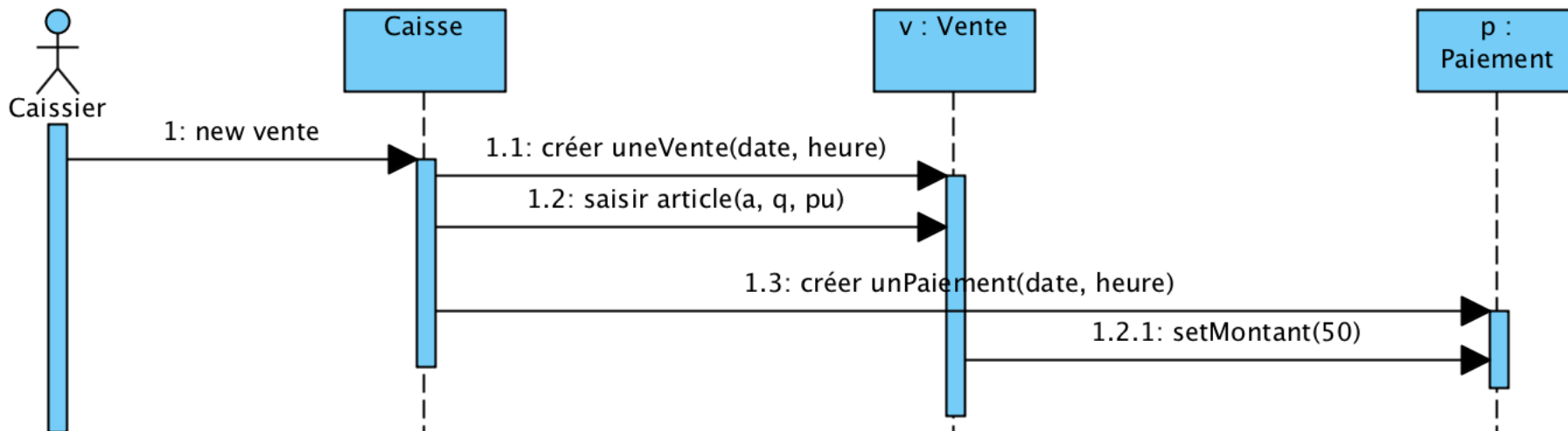
# Illustration : ne mettre que les dépendances nécessaires



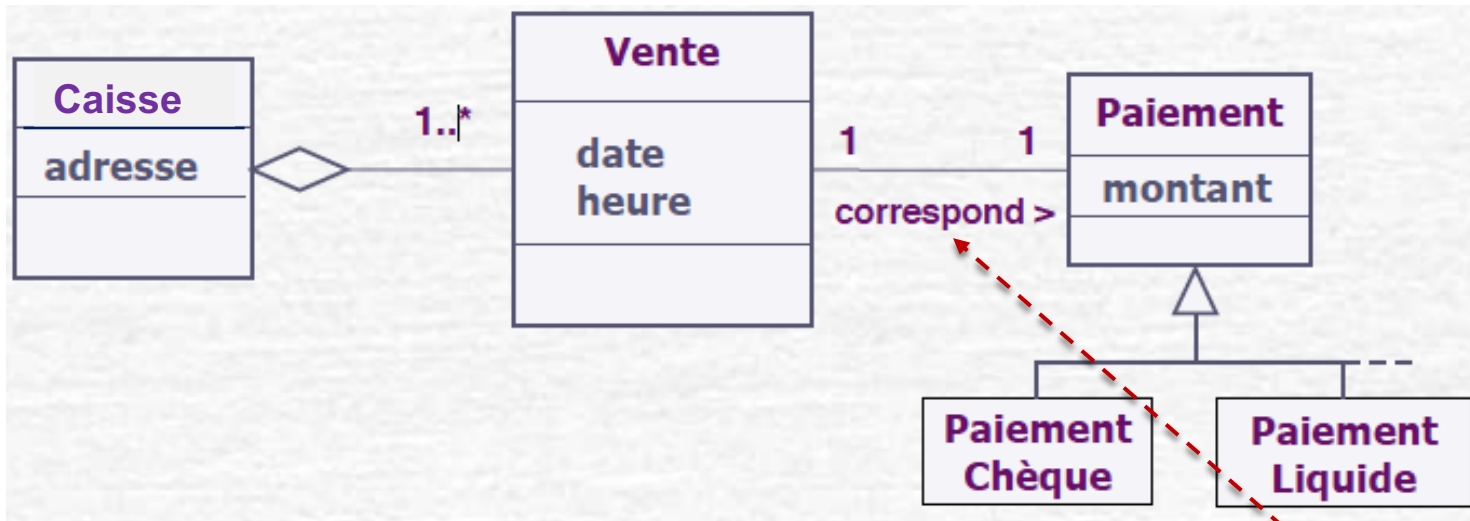
*Si c'est la Caisse qui crée le paiement, on **ajoute** un couplage de Caisse à Paiement, qui n'existait pas dans le DCL*

Imaginons qu'on ait à déterminer quel composant gère le paiement

**solution 1 :**

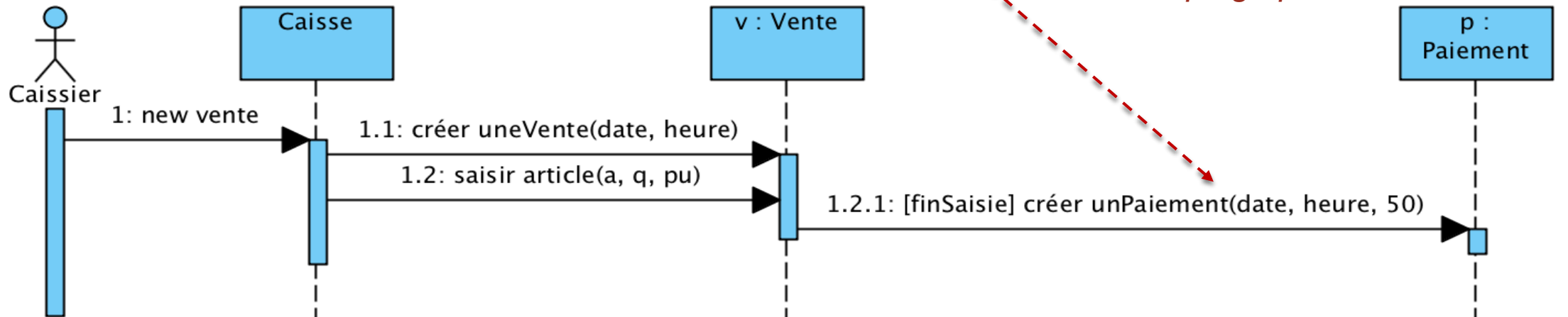


# Faible couplage : règles (2)



*Le Paiement est plus naturellement lié à la Vente, qui peut par ailleurs avoir d'autres méthodes liées aux Paiements : choixTypePaiement, encaisser (close la Vente), ...*

## solution 2



*Si c'est la vente qui crée le paiement, on traduit le couplage présent dans le DCL*



# Principes élémentaires

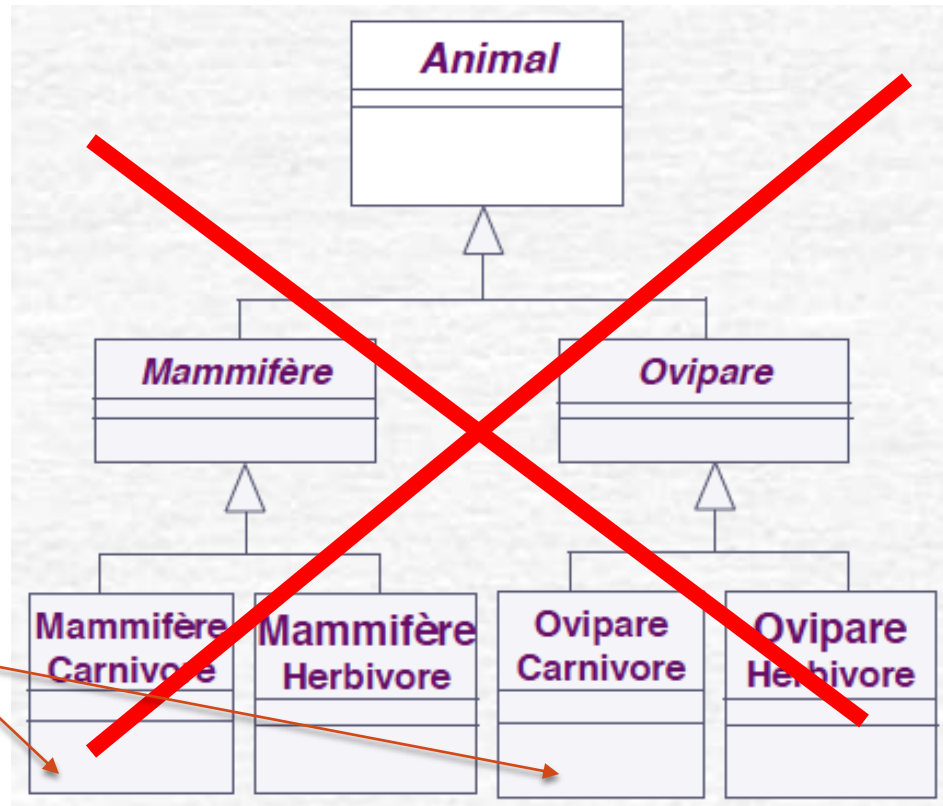
---

Bonne conception

*illustrés ci-après...*

- **Ne pas mettre des accesseurs / mutateurs** pour tous les attributs systématiquement
  - On perdrait les bénéfices de l'encapsulation !
- Ne jamais dériver une classe pour **ne tirer parti que** de certains attributs et méthodes
- Préférer la **composition** à l'héritage

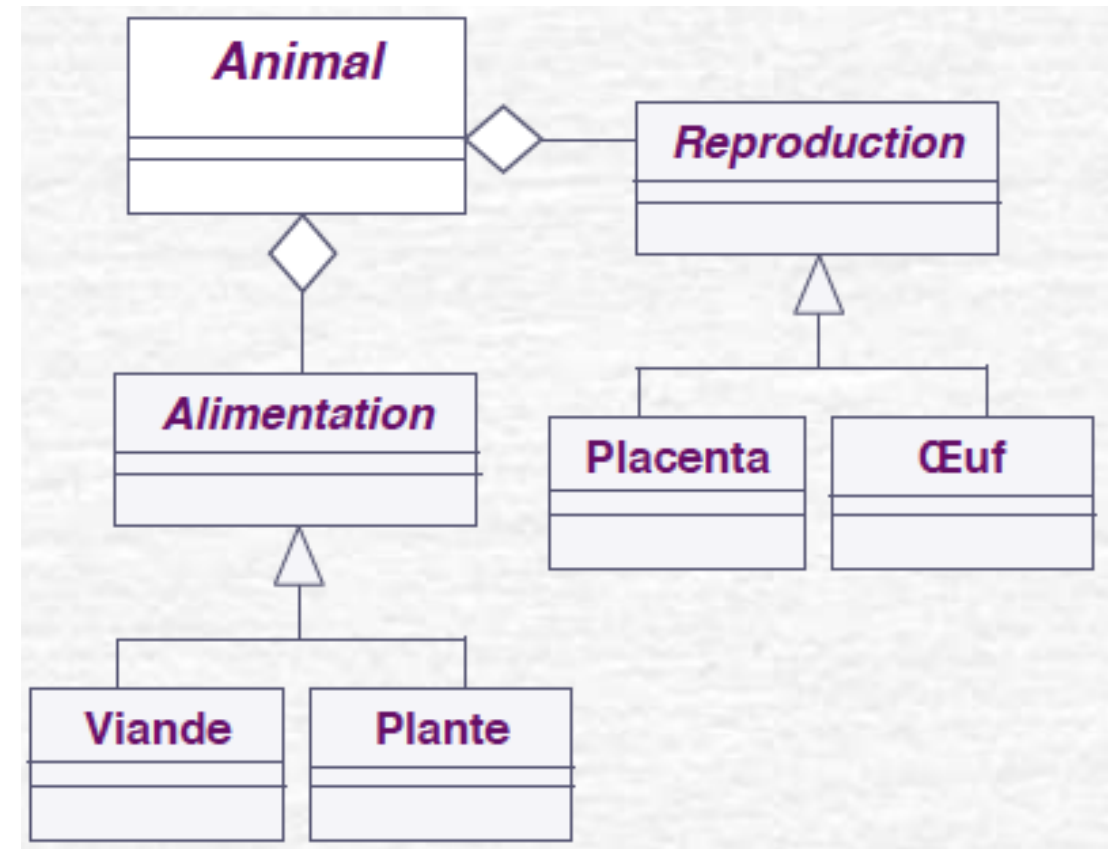
# « Préférer la composition à l'héritage »



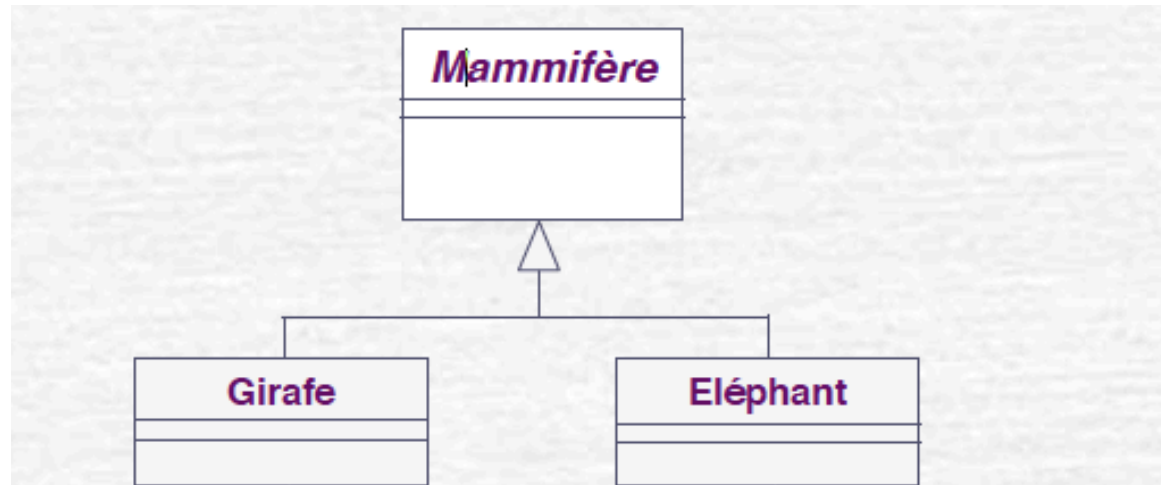
code de Carnivore dupliqué

- Explosion combinatoire
- Duplication de code

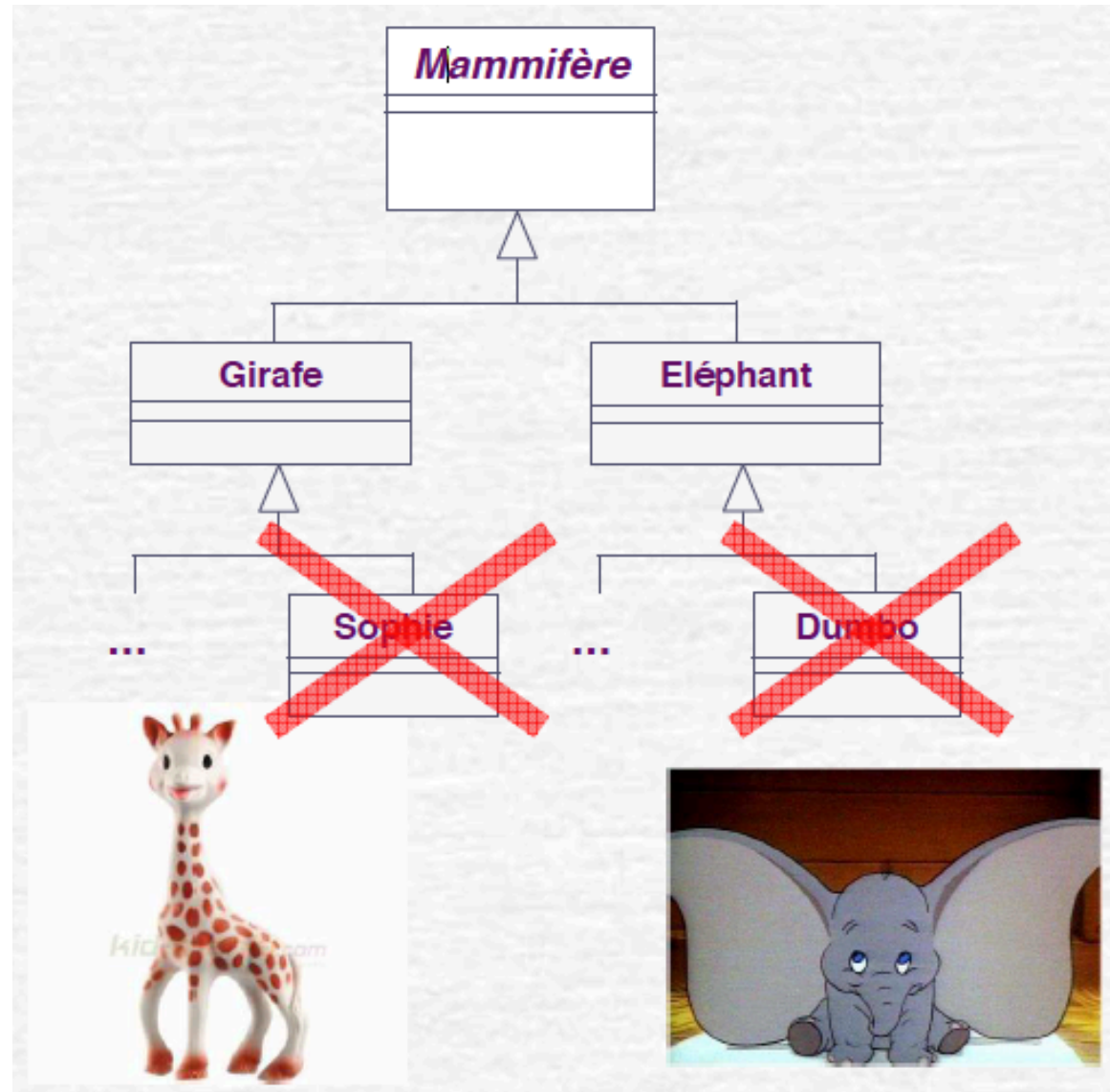
Encore appelé principe **d'indirection** ou de **délégation** : on délègue à la classe *Alimentation* le comportement *manger()* de l'*Animal*



« Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »



« Ne JAMAIS dériver une classe pour certains **seulement de ses** attributs et méthodes »



Le jouet *Girafe Sophie* a la texture d'une girafe (telle que représentée dans des films d'animation)...



## SOLID (Single, Open, Liskov, Interface, Dependency)

Principes fondamentaux



# Responsabilité unique (SRP)

- « **Une seule responsabilité = une seule raison d'être modifiée** »
- Observation : on a souvent tendance à donner trop de responsabilités à un objet
- Comment procéder ?
  - Analyser les méthodes de la classe
  - Les regrouper pour constituer des ensembles homogènes
    - ex.: accès à un BD, à une API spécifique, celles qui touchent un même ensemble d'attributs
    - (concept de cohésion précédent)
- Affecter si possible les responsabilités **correspondants aux informations** que la classe possède

# Illustration

- Soit la classe Employé suivante

```
class Employee {  
    ....  
    public Money calculatePay() { .. }  
    public String reportHours() {...}  
    public void saveInDB() {...}  
}
```

BadClassEmployee
-firstName -lastName -function -hiringDate
+calculatePay() +reportHours() +saveInDB()

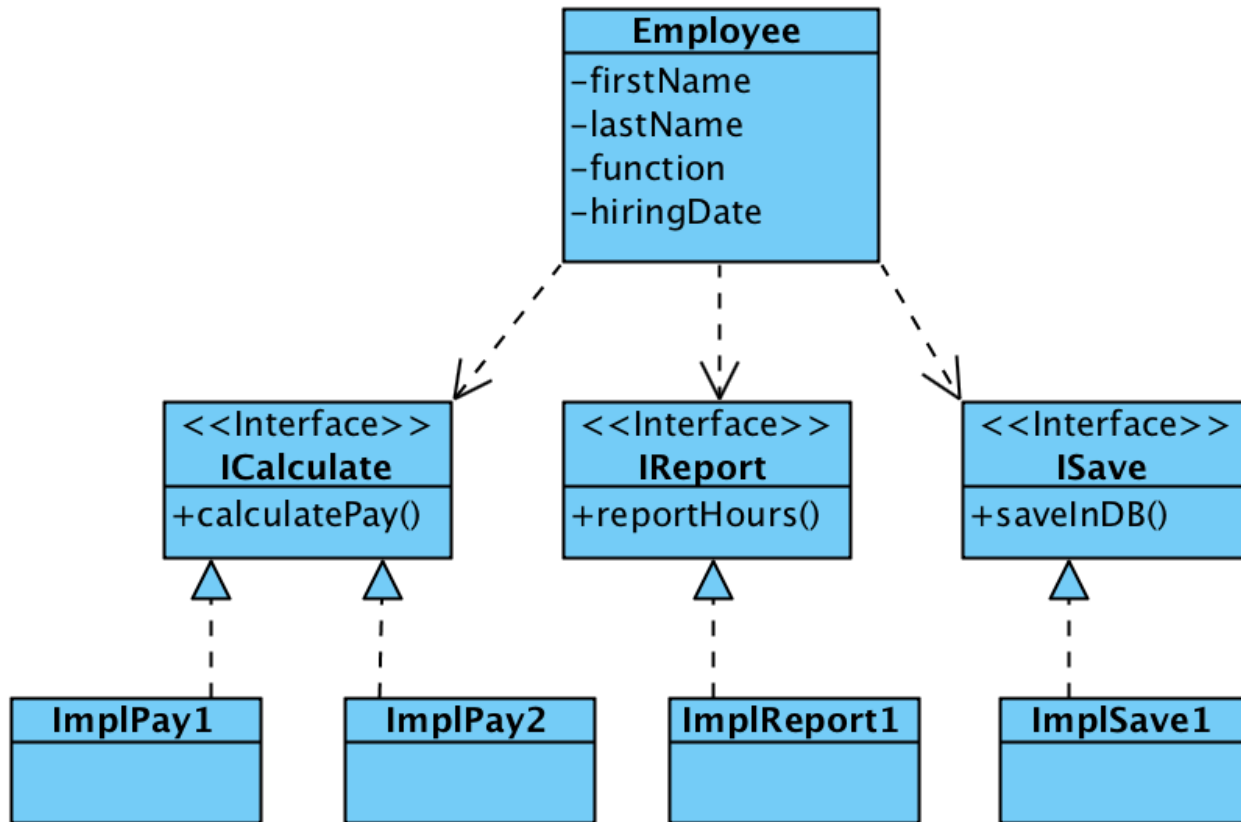
Quelles sont les responsabilités de cette classe ?

A quelles évolutions de code sont sensibles ces méthodes ?



- **Modification du calcul de Paye** → par le Service Comptable
- **Modification de la structure de la BD** → par le DBA
- **Modification du format**, pour le reporting des heures → par les Gestionnaires
  
- Ça fait beaucoup ! Notre classe n'est pas cohésive, elle a trop de **responsabilités**
  - Une idée ?
    - Repenser la classe : quelle *unique chose* devrait faire Employee ?
    - Séparer ces fonctions dans des classes différentes de manière à ce que chaque modification ait lieu sans modifier la classe Employee partout où elle est utilisée
  - Définir une interface pour Save (ISave), calculate (ICalculatePay) et report (IReportHours)

# Single Responsibility Principle (SRP)



- Si le calcul de paye évolue, on implémente une nouvelle classe pour ce nouveau calcul
  - *Sans tout changer à la classe Employee*
- Idem pour les modifications de formats pour le reporting, de BD pour la persistance
- Un **Employee** sait donner son nom/prénom, sa fonction et sa date d'embauche. Point.



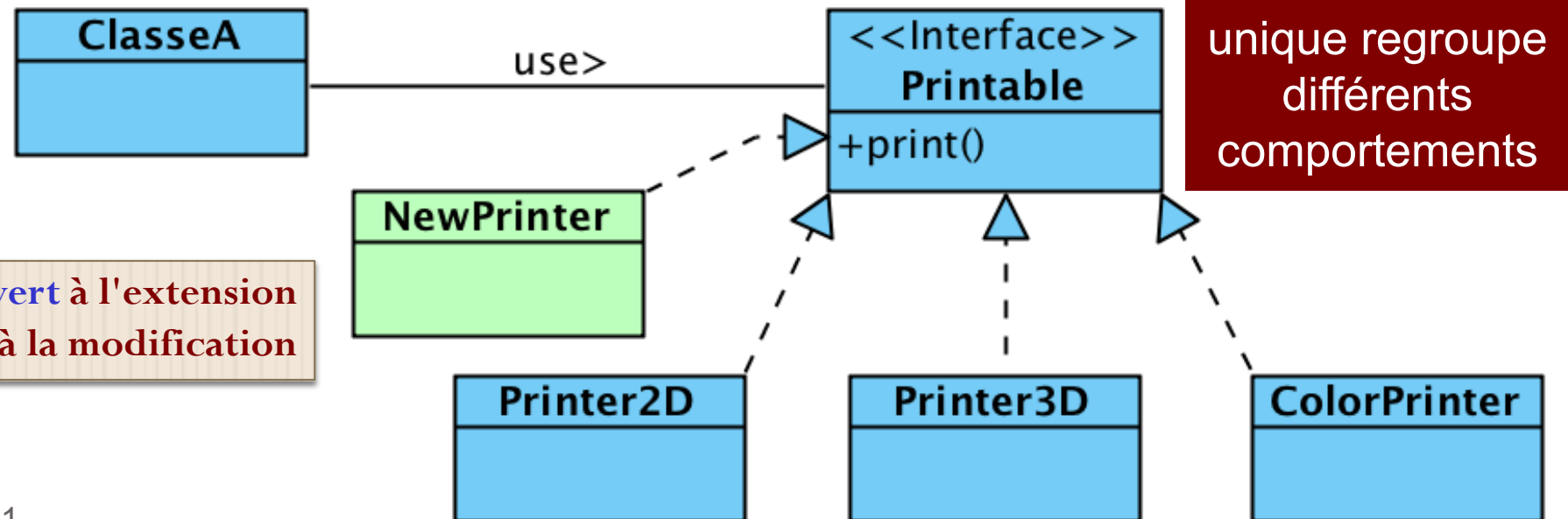
## GRASP OCP (Open / Close Principle)

Ouverture / Fermeture

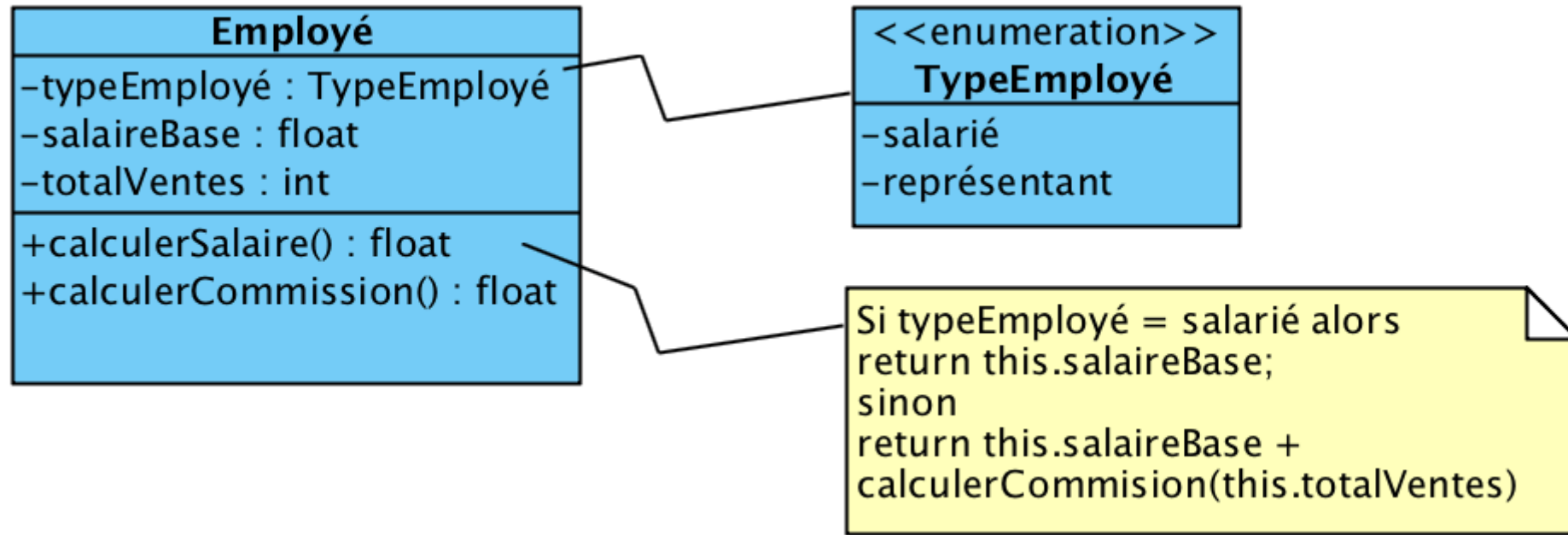
# Principe d'ouverture / Fermeture

- « Les entités logicielles (classes, packages, etc.) doivent être **ouvertes à l'extension** mais **fermées à la modification** »
- Soit une classe *A* qui utilise une interface *I* avec des services implémentés dans les classes concrètes *C1*, *C2*, etc.
- En cas de nouvelle fonctionnalité, on peut étendre *I* avec une nouvelle classe **sans impacter** le code de *A*.
- Le code existant, stable, n'est pas modifié.

Un module doit être ouvert à l'extension et fermé à la modification

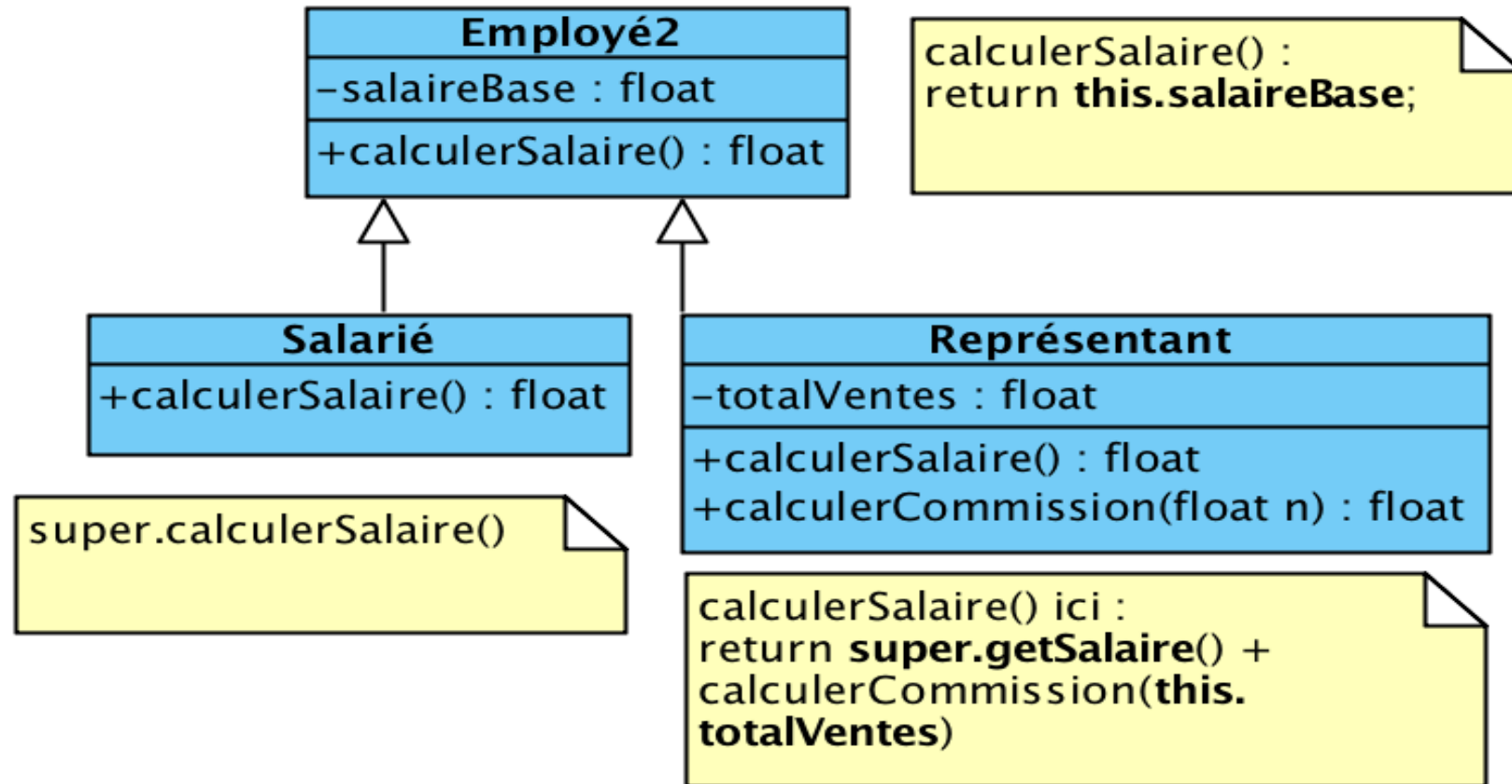


# Exemple de non respect d'OCP



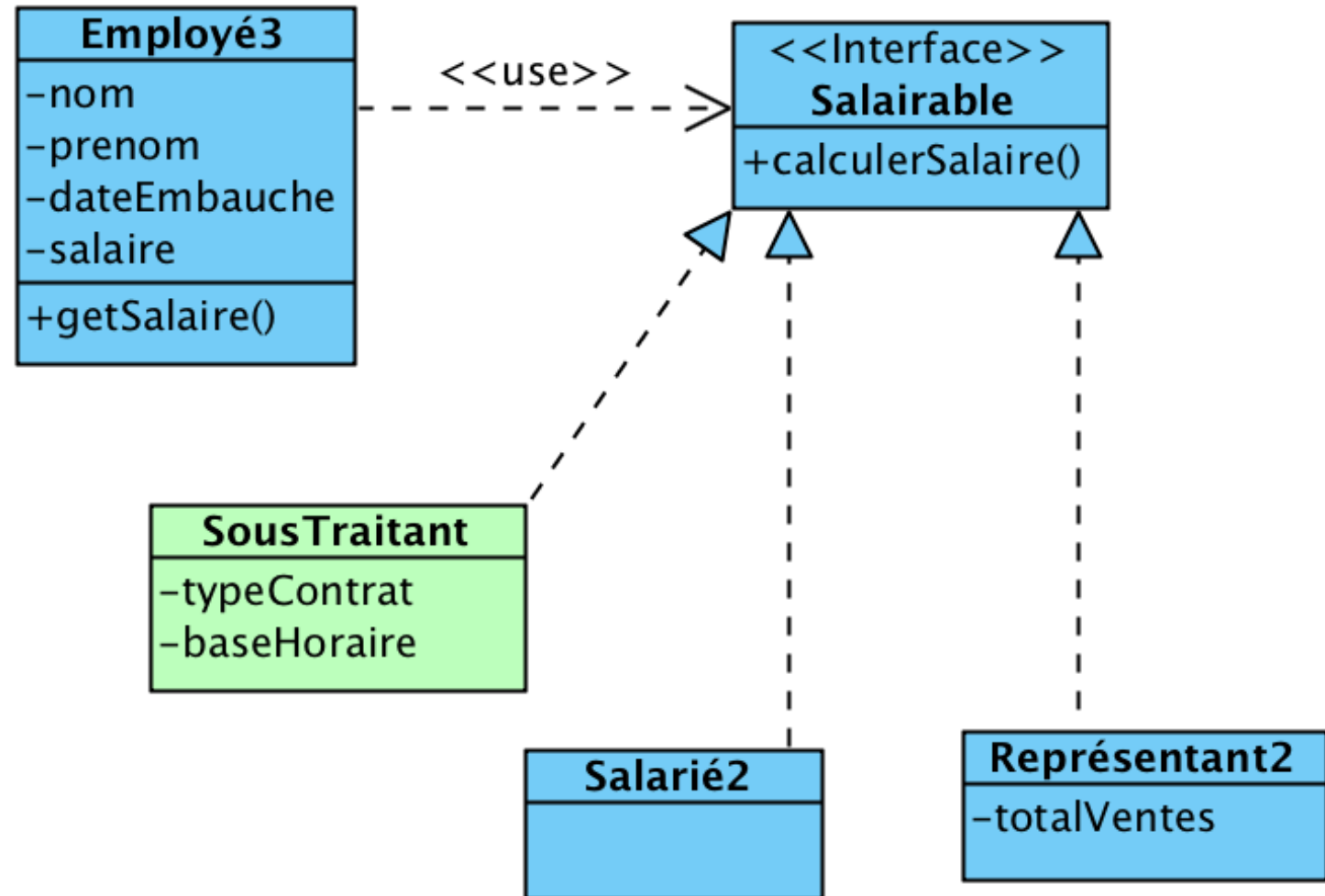
Que se passe-t-il si on doit considérer un nouveau type d'employé ?

# Première version du principe d'OCP



Que se passe-t-il si on doit considérer un nouveau type d'employé ?

# Encore mieux : OCP avec une interface



# Les limites d'OCP

Attention : **ne pas chercher à ouvrir/fermer toutes les classes** de l'application

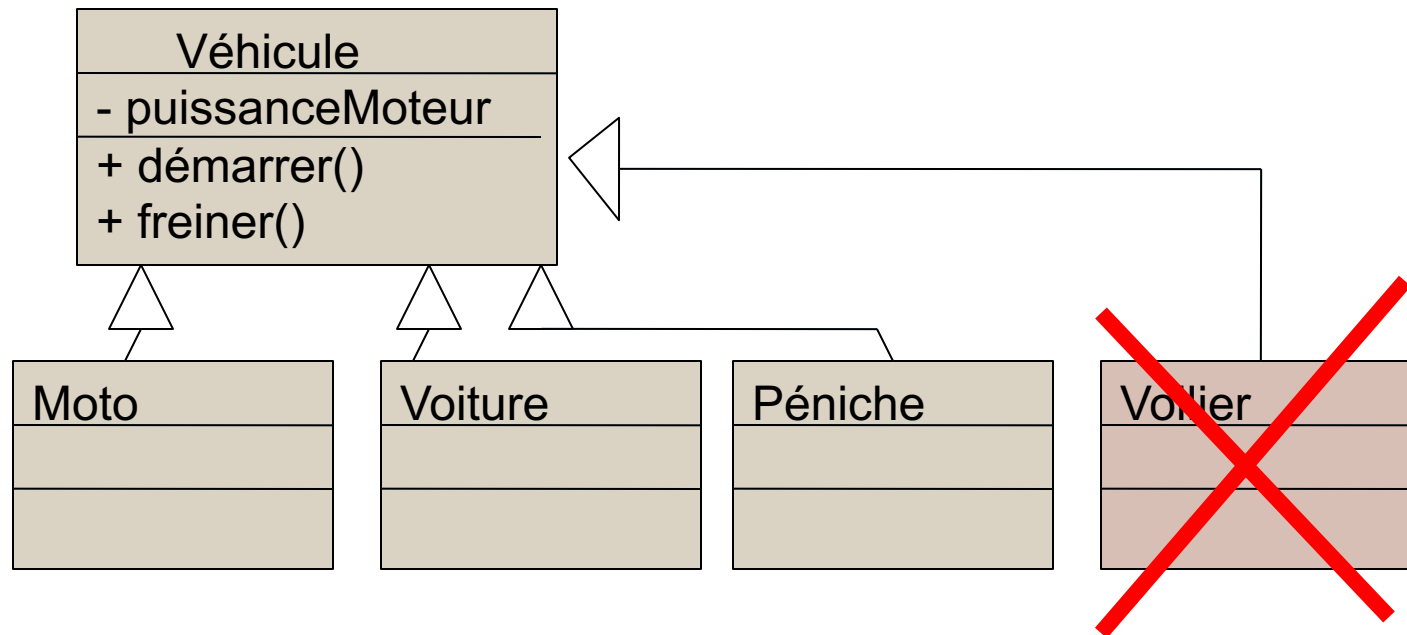
- Cela constitue une erreur car la mise en œuvre de l'OCP ajoute de la complexité qui **devient néfaste** si la flexibilité recherchée n'est pas réellement exploitée.
- Il convient de s'inspirer :
  - des besoins d'évolutivité exprimés par le client,
  - des besoins de flexibilité pressentis par les développeurs,
  - des changements répétés constatés au cours du développement



# PRINCIPE de Substitution de LISKOV

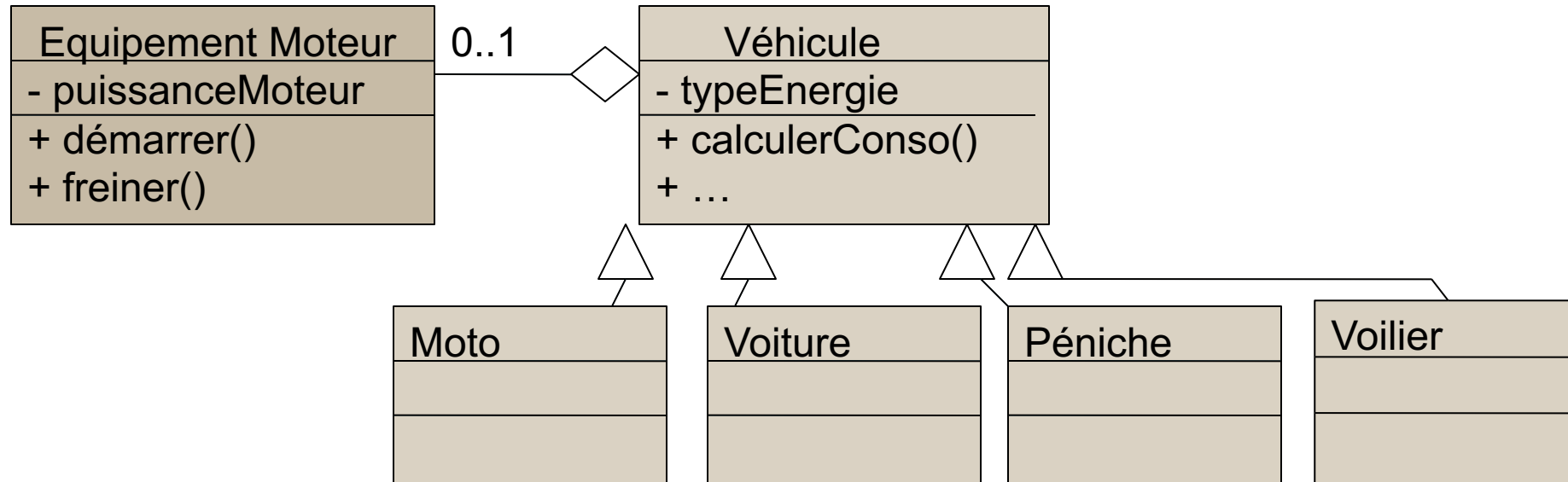
- « *On doit pouvoir placer la sous-classe partout dans le code où figure la classe parent* »
- Va contre l'idée répandue que **l'héritage est mis en œuvre pour factoriser du code** entre plusieurs classes
- Quand on utilise l'héritage, il faut penser aux **comportements**, pas simplement aux attributs ; notamment :
  - Les pré-conditions définies par les sous-classes **ne doivent pas être** plus **restrictives** que celles héritées.
  - Les post-conditions définies par les sous-classes **ne doivent pas être** plus **larges** que celles héritées.
- Technique : appliquer la **règle des 100%**
  - La sous-classe hérite totalement de sa superclasse (attributs, méthodes, relations)

# Principe de Liskov respecté ?



- Pas vraiment !
- Pour Voilier, il faudrait redéfinir `démarrer()` et `freiner()` en méthodes vides (qui ne font rien)...

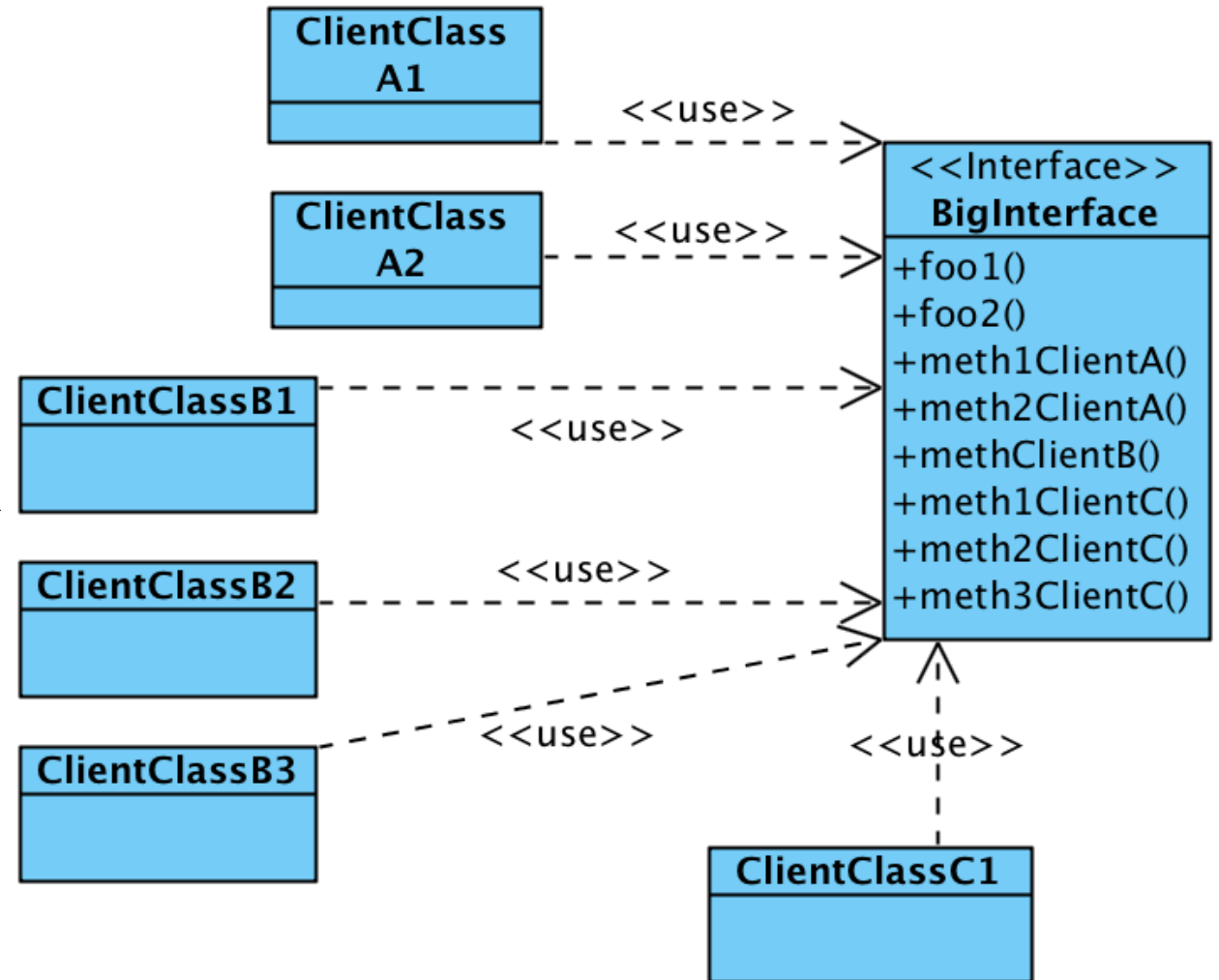
## → Préférer **encapsuler** le moteur



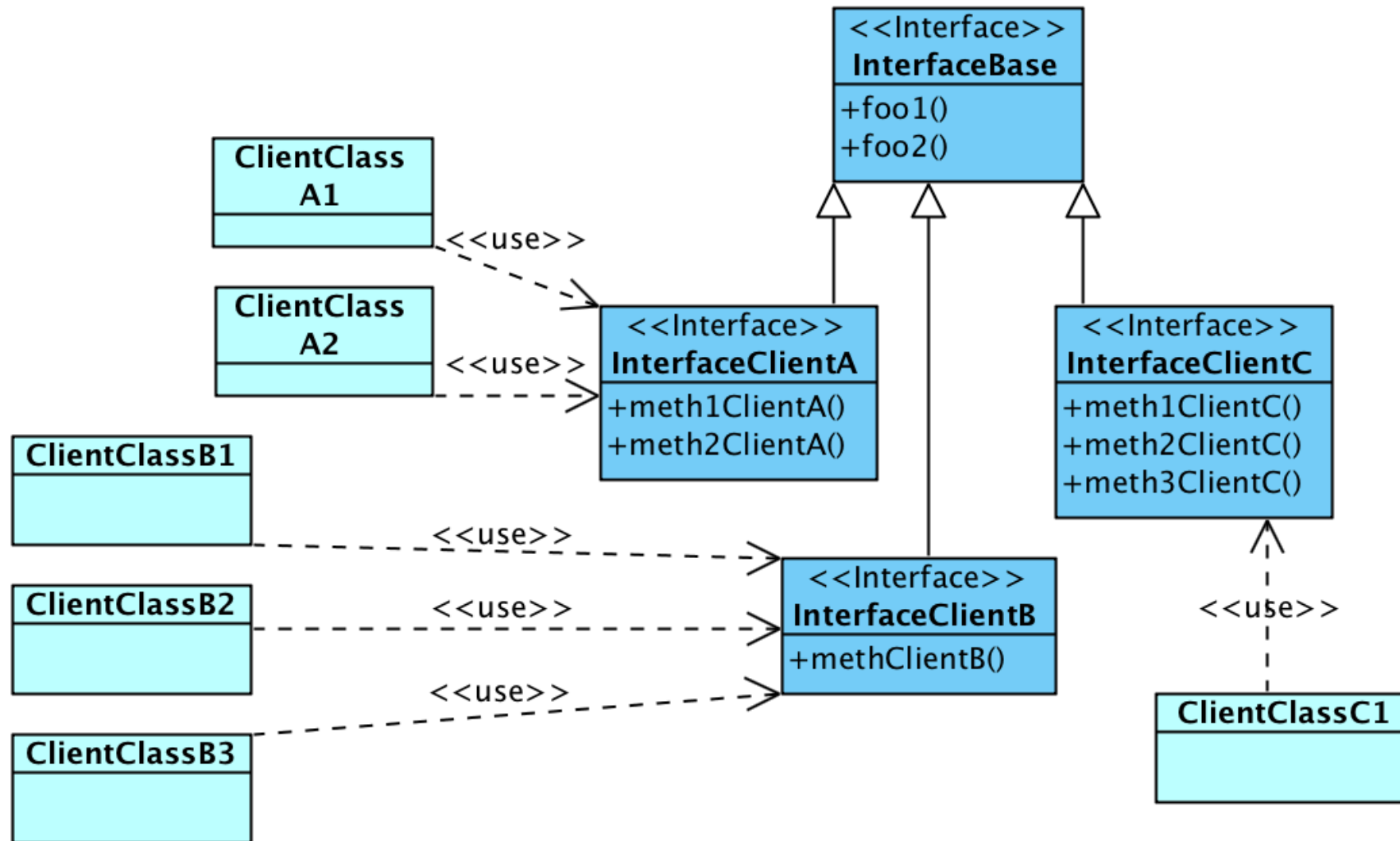
- Un véhicule possède -ou pas- un moteur.
- Les méthodes **démarrer()** et **freiner()** sont propres au composant **EquipementMoteur**.

# Séparation des Interfaces (ISP)

- « Les classes ne doivent pas avoir à dépendre d'une interface qu'ils n'utilisent pas »
- Toute **classe Client** qui utilise une **BigInterface** possède un comportement flou
  - Quels services précis utilise-t-elle ?
- Toute classe **réalisant une BigInterface** doit implémenter chacune de ses fonctions  
→ confusion au niveau des **rôles** des classes



# Séparation des interfaces (suite)



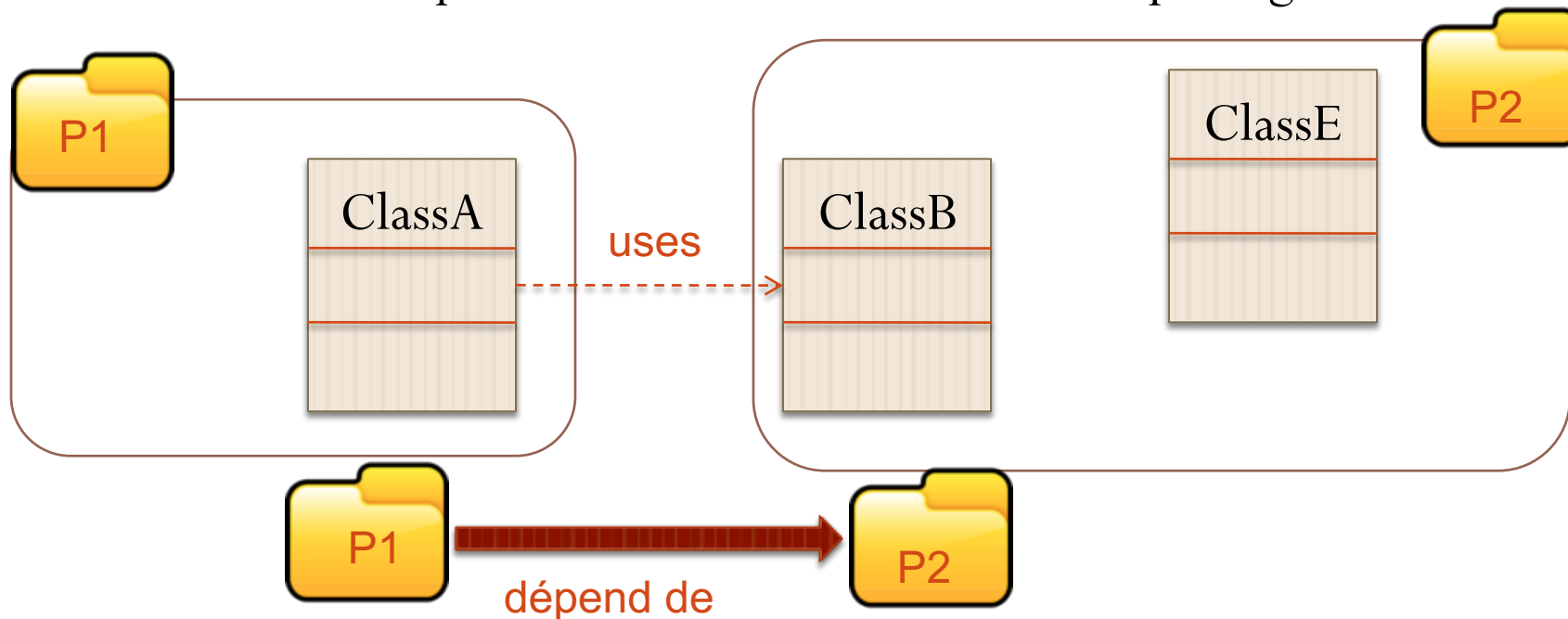
# Ex. la classe List de .NET

- Elle implémente des interfaces au rôle explicité par leur nom :
  - IList,
  - ICollection,
  - IReadOnlyList,
  - IReadOnlyCollection,
  - IEnumerable

# Injection de Dépendance (ou de contrôle : IoC)

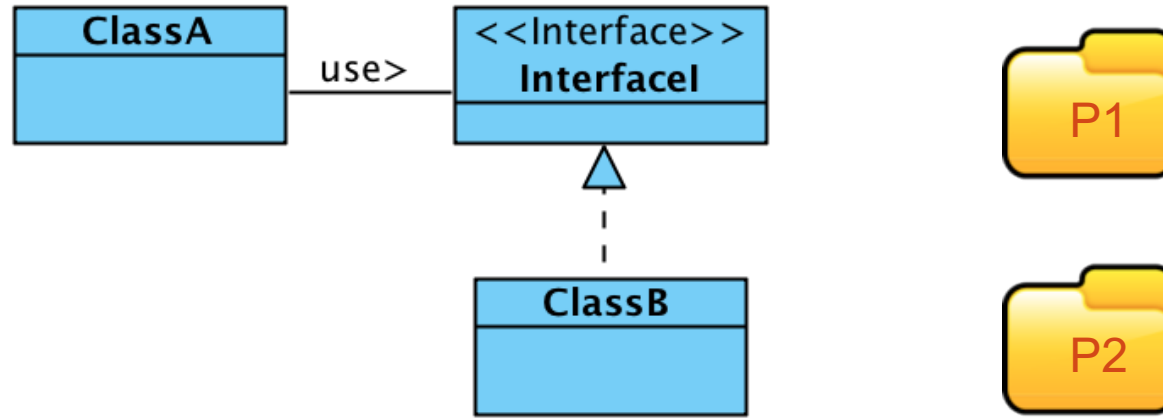
(Principe plus secondaire)

- **Constat** : souvent le flot d'exécution d'un logiciel dépend des *frameworks* utilisés. On duplique même le code (par ex. persistance) pour s'adapter au *fwk* selon la BD utilisée. Or on préfèrerait que cette responsabilité (liées à la persistance) soit du côté du *fwk* et non plus du logiciel.
- Objectif : **inverser la dépendance**
- **Illustration** : soit une classe A qui utilise une classe B d'un autre package



# Injection de Dépendances (2)

- On va généraliser le comportement de B en une interface I, que B va implémenter :



- Placer l'interface I dans P1 (ou dans P3) qui contiendra toutes les méthodes que A peut appeler sur B
- Indiquer que B implémente l'interface I
- Remplacer toutes les références au type B par des références à l'interface I dans A, ainsi maintenant :





# Injection de la dépendance

- Pb : plusieurs versions de  $B$  peuvent implémenter  $I$
- **Comment  $A$  récupère la bonne référence (de type  $I$ ) sur l'instance  $B$  dont il doit utiliser les services ?**
- Solution : injecter **dynamiquement** dans  $A$  la dépendance vers le  $B$  à utiliser (créer, par exemple, un objet  **$b$  de type  $B$  et l'injecter** dans un objet de type  $A$ ).
- Plusieurs mécanismes pour cela :
  - par **constructeur** : on passe l'objet  **$b$**  à l'instanciation de  $A$
  - par **mutateur** : on passe l'objet  **$b$**  à une méthode de  $A$  qui va par exemple **modifier un attribut**
  - par **interface**
  - par **champs** : ex. ci après

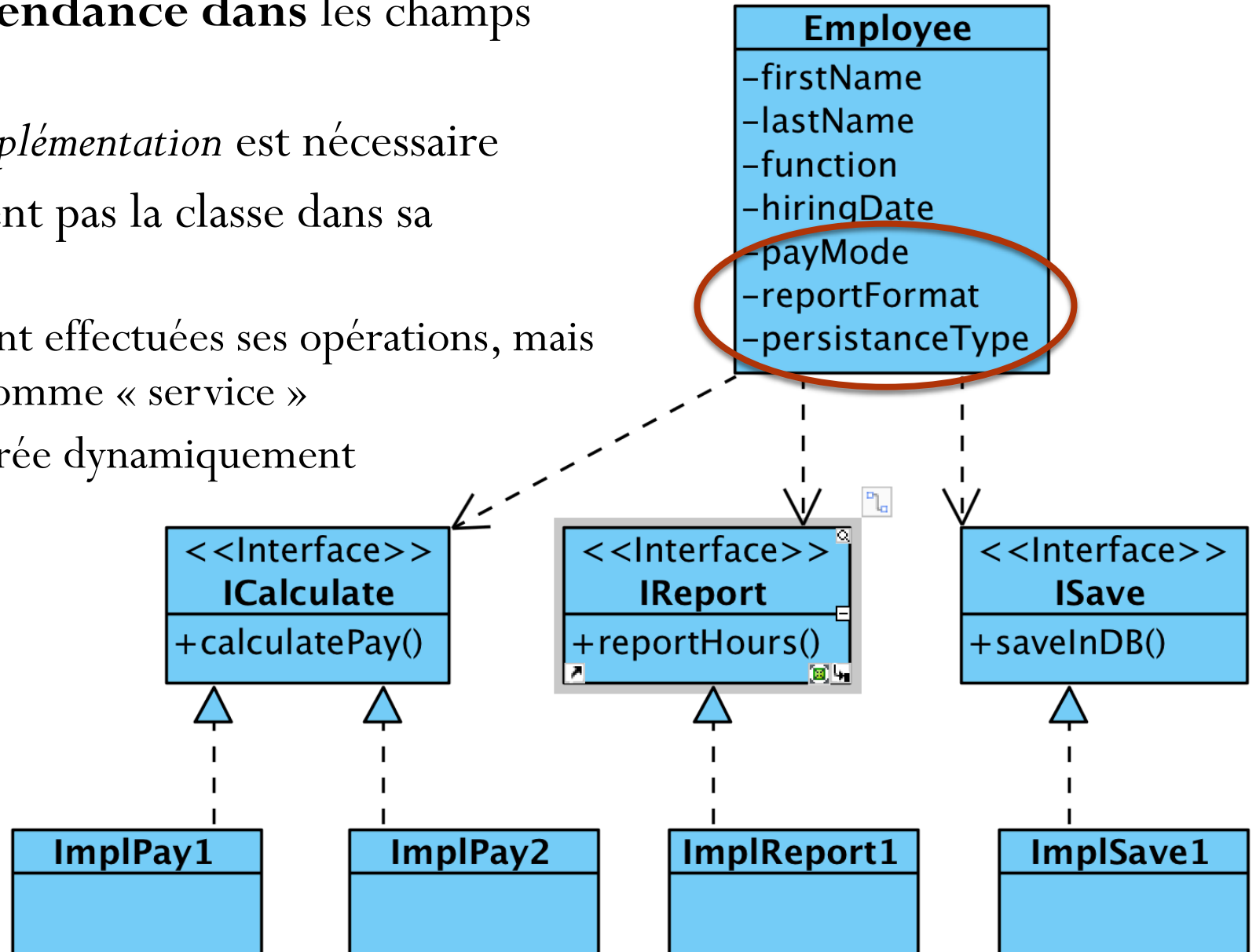
*S'appuie sur le **pattern Factory***

*Nota : Il existe de nombreux **fwk d'inversion de dépendance** (Spring, Google Guice)*

- Ex. classe *Employee* précédente
- On va injecter de la dépendance dans les champs d'*Employee*
- On lui injecte quel *type d'implémentation* est nécessaire
- Les modifications n'atteignent pas la classe dans sa structure
  - Elle ne sait pas comment sont effectuées ses opérations, mais ce qu'elle peut demander comme « service »
  - La dépendance peut être gérée dynamiquement

**ATTENTION** - L'injection de dépendance peut entraîner un couplage plus fort

Classes  
d'implémentation



# Pour finir, lien Conception / Codage : importance des structures de données du langage

- Certaines structures de données peuvent traduire et **simplifier les méthodes imaginées en phase d'analyse**
- Ex. : une classe d'analyse **Contact** avec une méthode **vérifierDoublon()**
  - si l'ID est défini par nom+prénom : il suffit de placer les objets Contact dans un conteneur *set* de Java (*set* ne tolère pas les doublons)
- Ex.: pour un **contrôle d'accès de salles**, on choisira une HashTable avec les numéros de salle (clef) et le code d'accès (valeur: true/false).
- Il est donc important de **bien connaître** les structures de données évoluées : conteneurs Java, arbres bicolores, skip-list, etc.

Contact
-nom
-prénom
-email
+vérifierDoublon()