Chap.2 – les principes SOLID

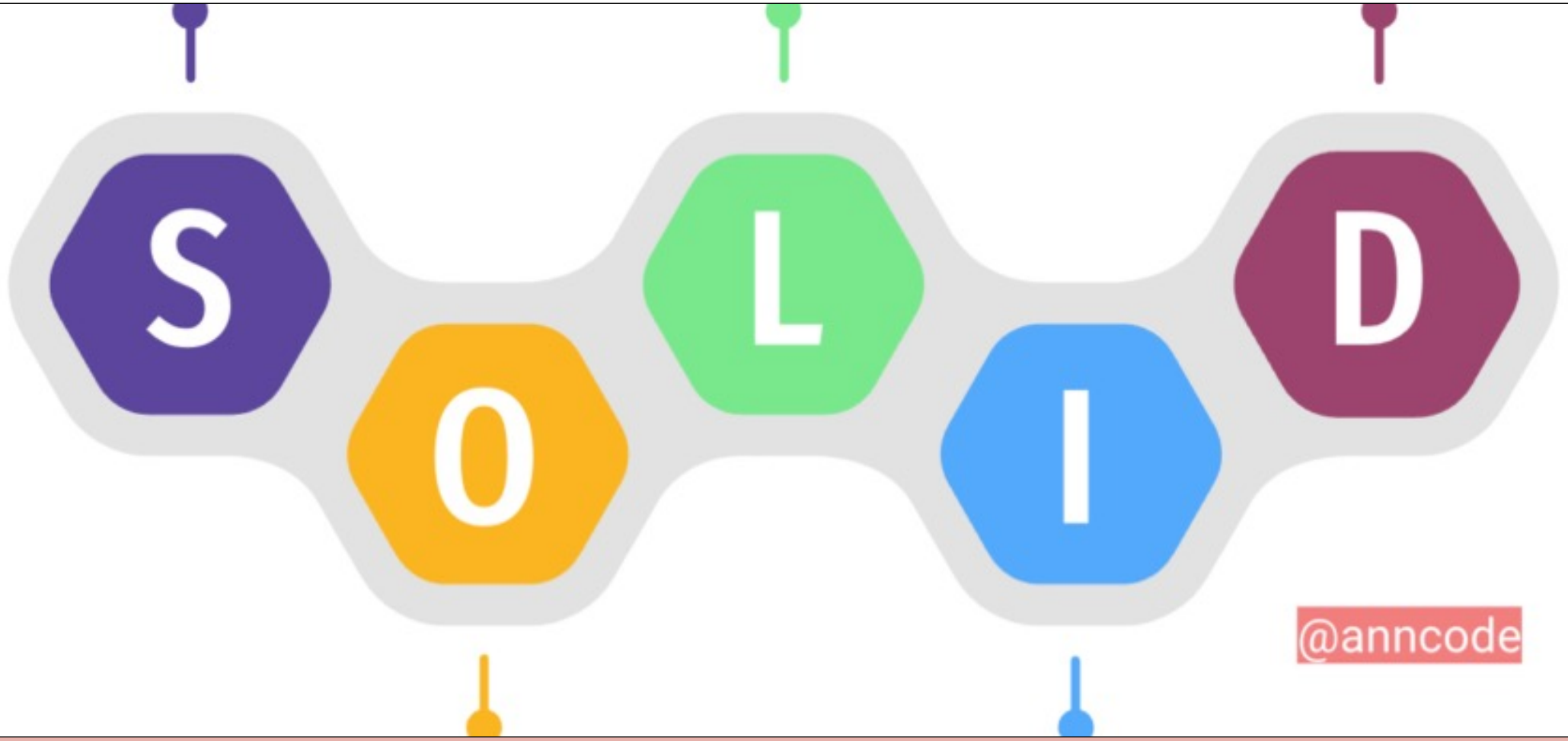
V. Deslandres ©

S1.10 - Qualité de Développement-1

BUT Informatique - ASPE

IUT de Lyon - Université Lyon 1

Sommaire de ce cours



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

Principes d'une Conception robuste

D'où ça vient ?

- L'acronyme SOLID a été inventé par Michael Feathers, blogueur et auteur du livre *Working Effectively With Legacy Code*
- Ils ont été popularisés par Robert C. Martin (Clean Code)

- Définit un code qui se veut **clair, propre, facilement maintenable et facile à faire évoluer**.

- Lorsqu'on parle « facilité » de maintenance ou d'évolution à propos du code, il faut comprendre que cela signifie que le coût nécessaire pour effectuer un changement à l'application devrait toujours être inférieur aux bénéfices apportés par le changement opéré.

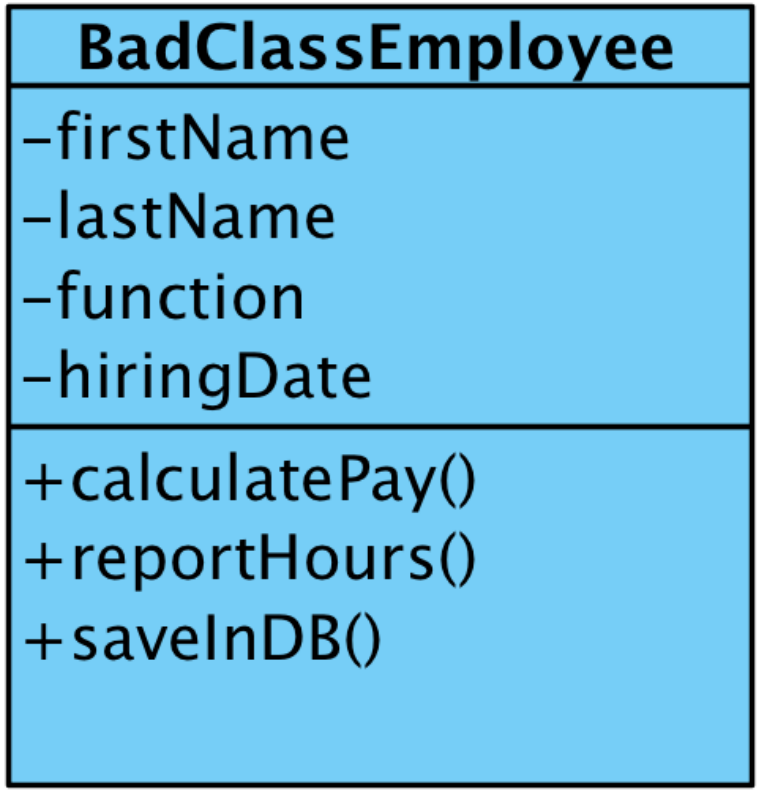
Responsabilité unique (SRP)

- « **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
    - Respecter le concept de cohésion !
- Affecter si possible les responsabilités **correspondant aux informations** décrivant la classe

Illustration

- Soit la classe Employé suivante

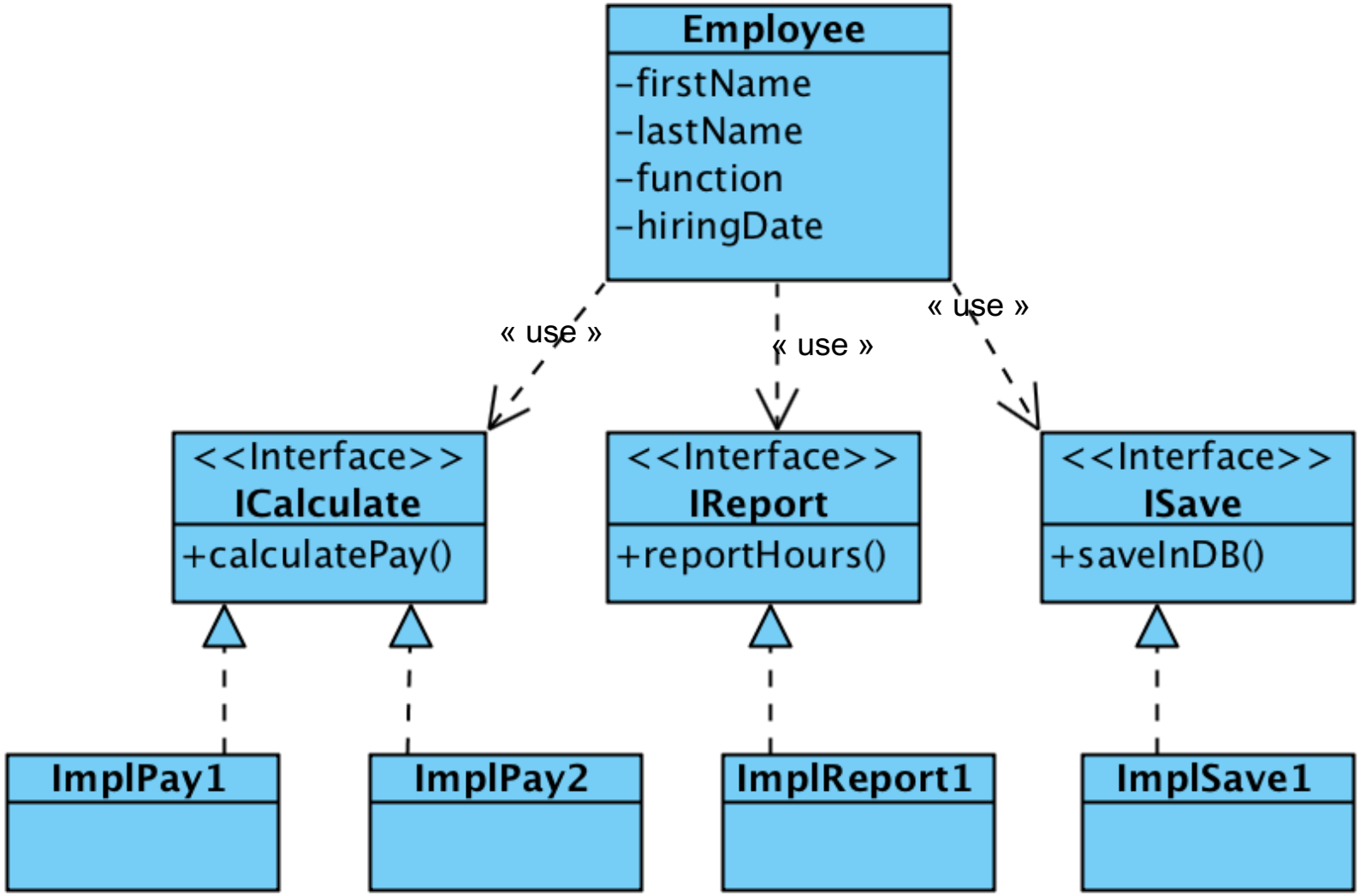
```
class Employee {
    ....
    public Money calculatePay() { .. }
    public String reportHours() {...}
    public void saveInDB() {...}
    }
```

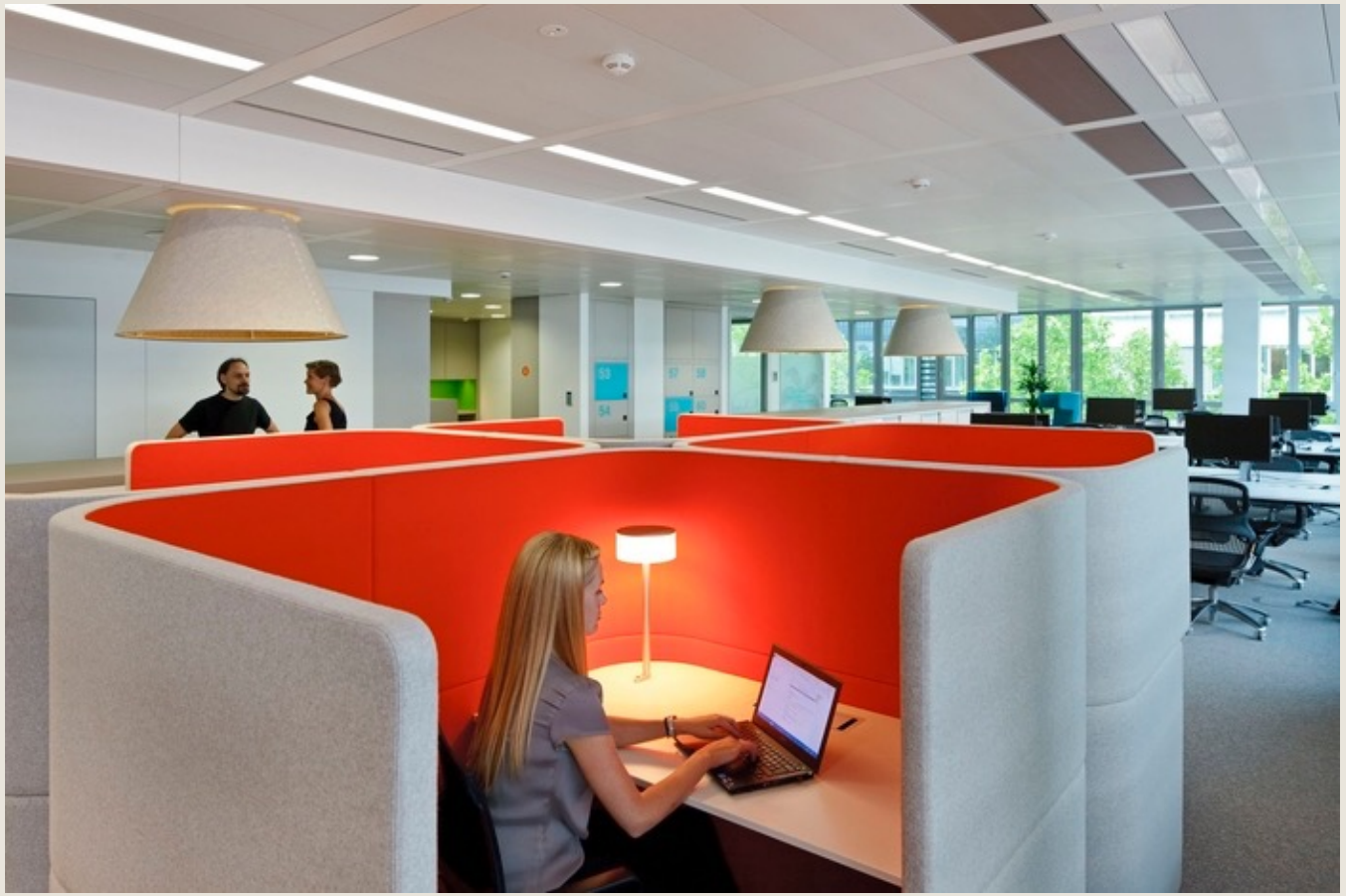


Quelles sont les responsabilités de cette classe ?

A quelles évolutions de code sont sensibles ces méthodes ?

Single Responsability Principle (SRP)





OCP (Open / Close Principle)

Ouverture / Fermeture

OCP

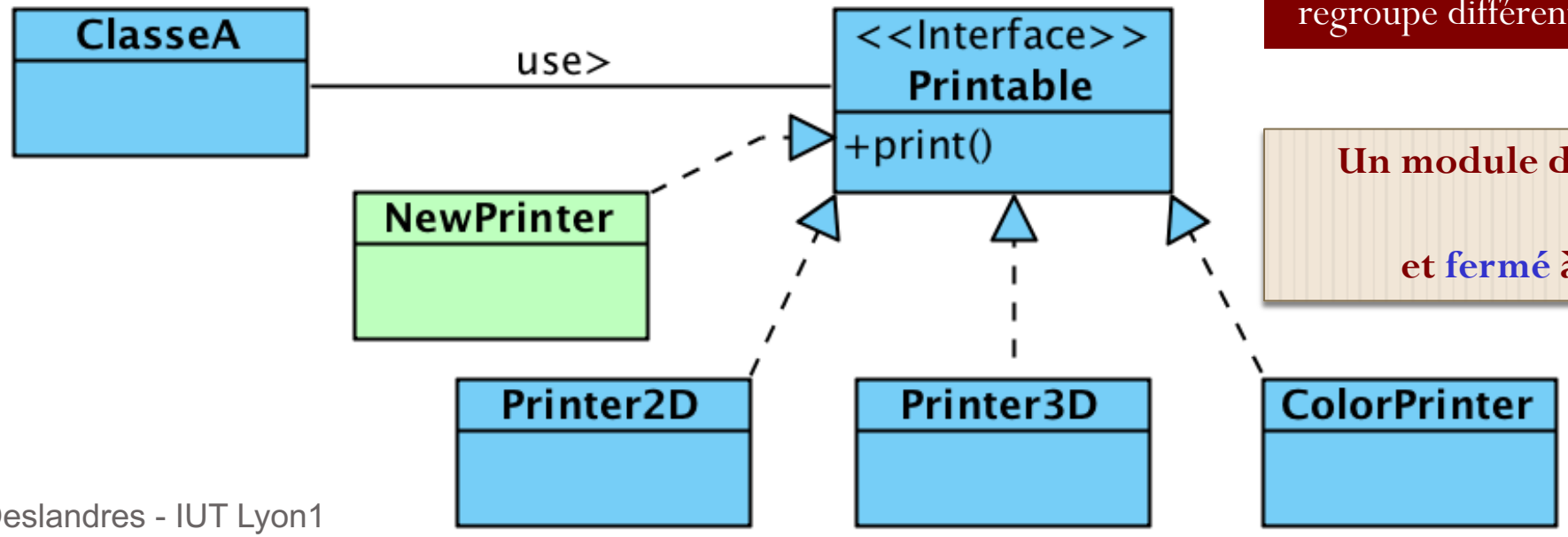
Principe d'ouverture / Fermeture

- Parce qu'une modification de code existant peut provoquer l'apparition de bugs, « Les entités logicielles (classes, packages, etc.) doivent être **ouvertes à l'extension** mais **fermées à la modification** »
- Soit une classe *A* qui utilise des services similaires (ex. print()), présents dans les classes *C1, C2,* etc.
- On va **créer une interface *I*** avec la ou les méthodes des classes concernées et dire que *C1, C2,* … implémentent le service ; et faire dépendre la classe *A* de *I* (couplage plus faible).

➔ 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.

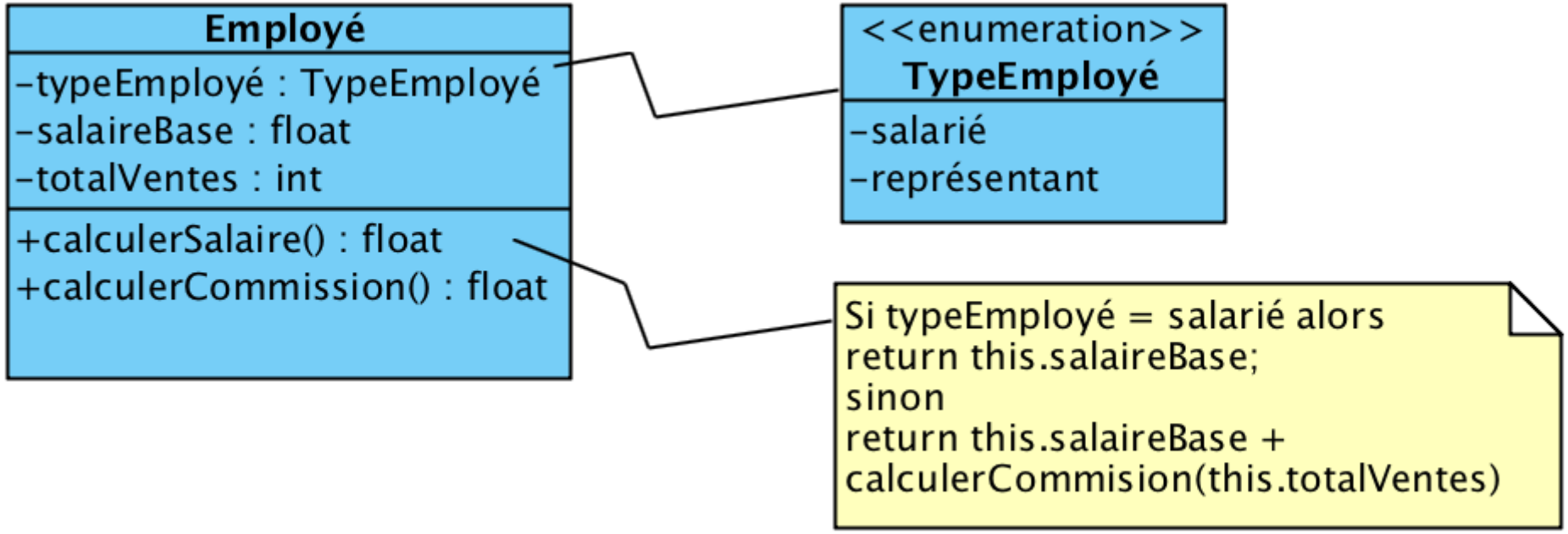
Abstraction : une interface unique regroupe différents comportements



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

OCP

Exemple de non respect d'OCP

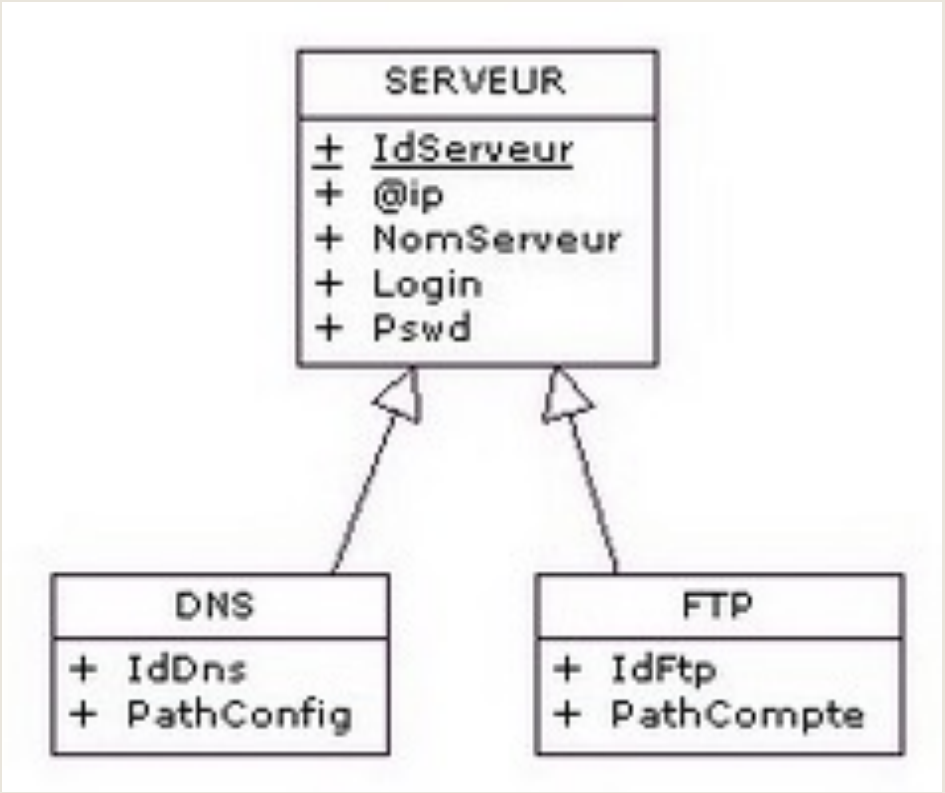


Que se passe-t-il si on doit considérer un nouveau type d'employé ?

Les limites du principe 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 LISKOV

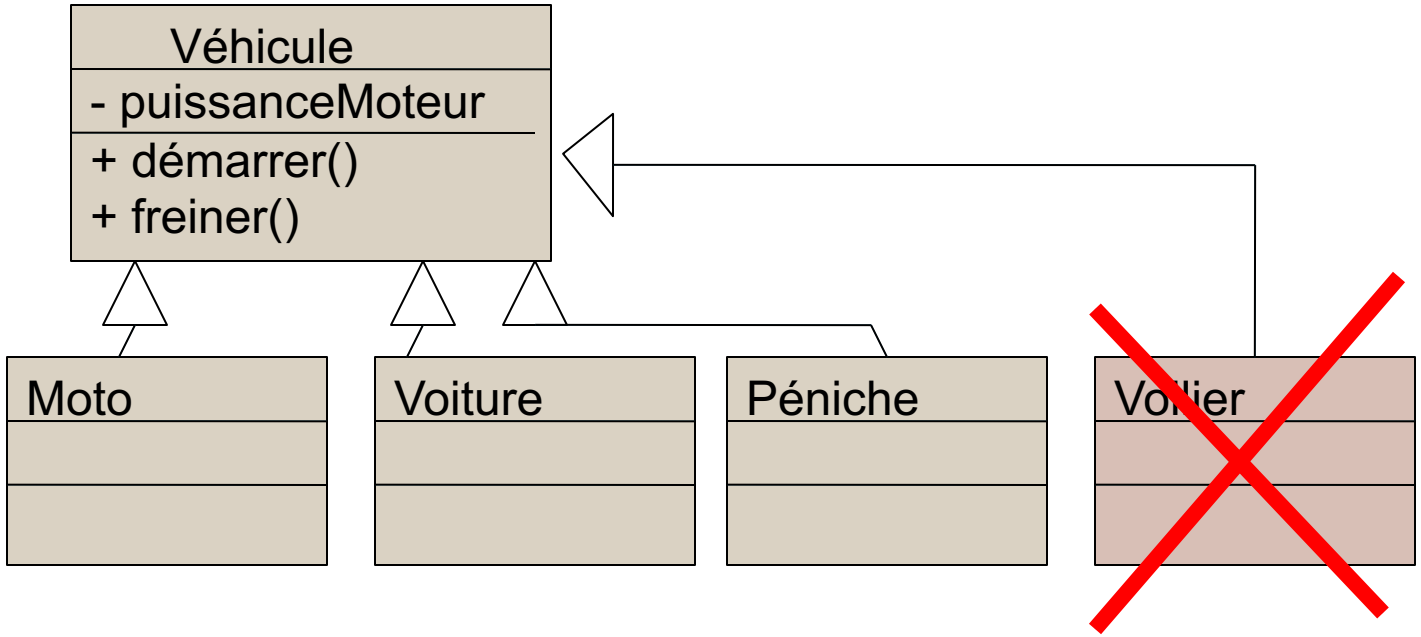
En cas d'héritage

PRINCIPE de Substitution de LISKOV

- *« On doit pouvoir placer la sous-classe partout dans le code où figure la classe parent »*
- 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.

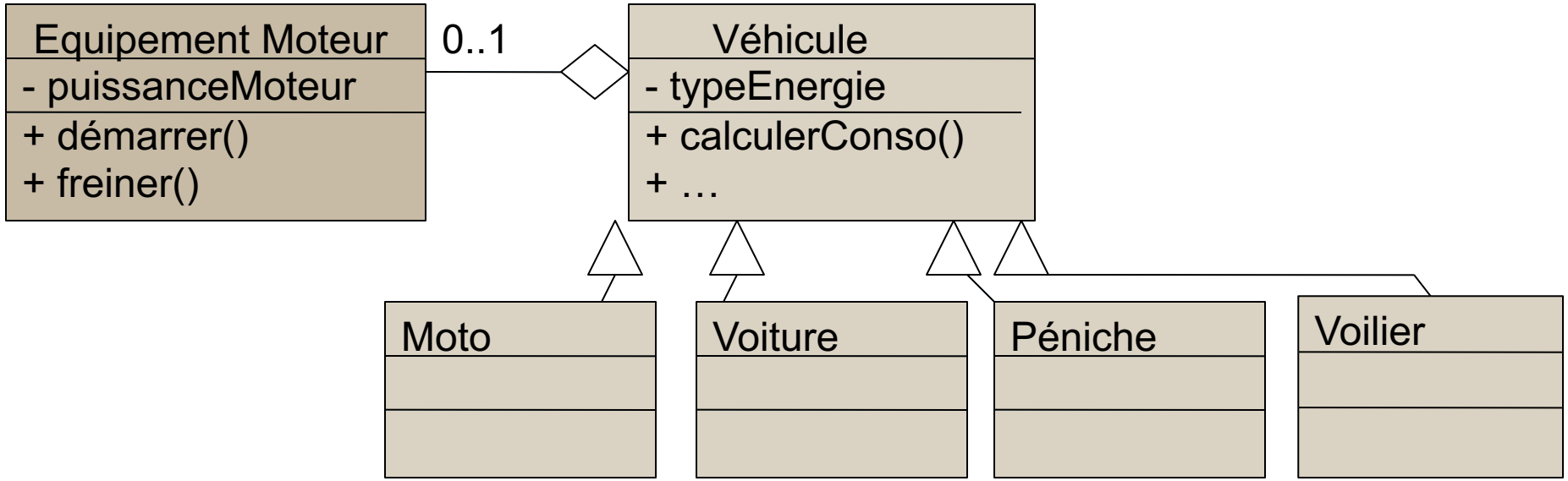
- Technique : appliquer la **règle des 100%**
  - La sous-classe hérite totalement de sa superclasse (attributs, méthodes, relations)

Principe de Liskov non respecté



- 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**
- (**Voilier** n'a pas d'équipement Moteur)

```java
public class Vehicule {
    String typeEnergie;
    String type;
    EquipementMoteur equipementMoteur;

    public Vehicule(String typeEnergie, String type, EquipementMoteur equipementMoteur) {
        this.typeEnergie = typeEnergie;
        this.type = type;
        this.equipementMoteur = equipementMoteur;
    }
```

```java
public class Moto extends Vehicule {

    public Moto(String ch, EquipementMoteur e) {
        super(ch, "moto", e);
    }

}
```

```java
public class Voilier extends Vehicule {

    public Voilier(String en, EquipementMoteur e) {
        super(en, "voilier", null);
    }

}
```

```java
public static void main(String[] args) {

    List<Vehicule> maListe = new ArrayList<>();
    // On crée une moto et un voilier
    Vehicule maMoto = new Moto("essence", new EquipementMoteur());
    Vehicule monVoilier = new Voilier("air", null);
    maListe.add(maMoto);
    maListe.add(monVoilier);

    for (Vehicule unVehicule : maListe) {

        if (unVehicule.equipementMoteur != null) {

            unVehicule.equipementMoteur.demarrer();
            System.out.println("Le véhicule "+ unVehicule + " a démarré");
        }
        monVoilier.equipementMoteur.freiner(); // génère une erreur à l'exécution

    }
```

Liskov appliquée aux interfaces

- Robert MARTIN a depuis légèrement modifié son principe : il concerne surtout les interfaces / classes abstraites

- Le principe de Liskov appliqué aux interfaces est :

  « *Un module qui utilise une interface ne doit pas être confondu avec l'implémentation de cette interface* »

  Ce principe consiste à avoir les abstractions **claires** et **bien définies**.

  La classe cliente qui exploite l'interface doit **avoir une idée précise** de ce que **fait** l'interface sinon on entre dans des sous cas à considérer, avec un risque de prolifération de *if / then / else,* qui sont des **sources d'erreurs connues,** en cas d'évolution du code.

Interfaces : ISP (Interface Segregation Principle)

Une séparation claire des contrats

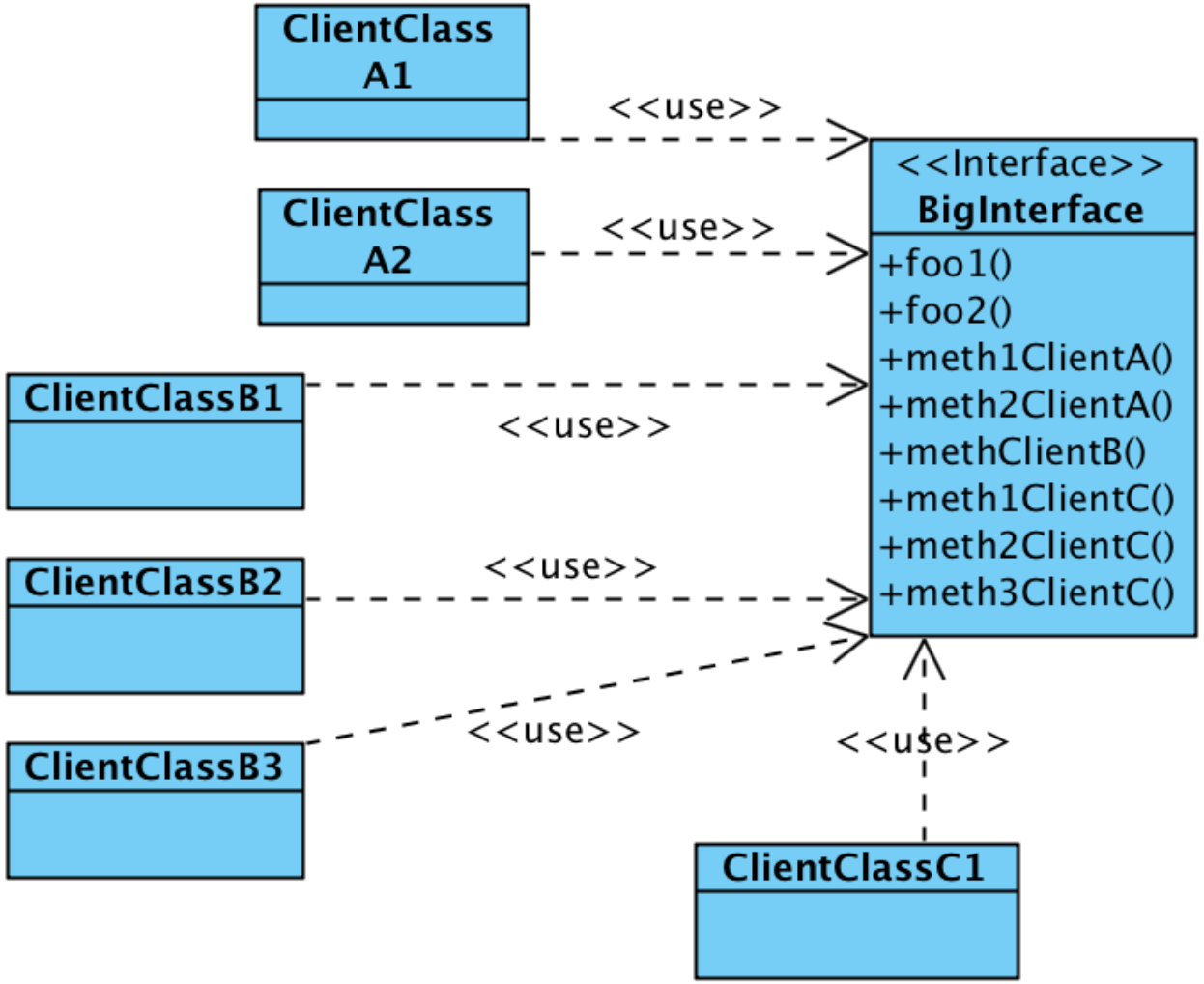
ISP Séparation claire des Interfaces

- « *Les classes ne doivent pas avoir à dépendre d'une interface ayant des méthodes qu'elles n'utilisent pas* »

- Toute **classe Client** qui utilise une BigInterface possède un comportement flou
  - Quels services précis utilise-t-elle ?

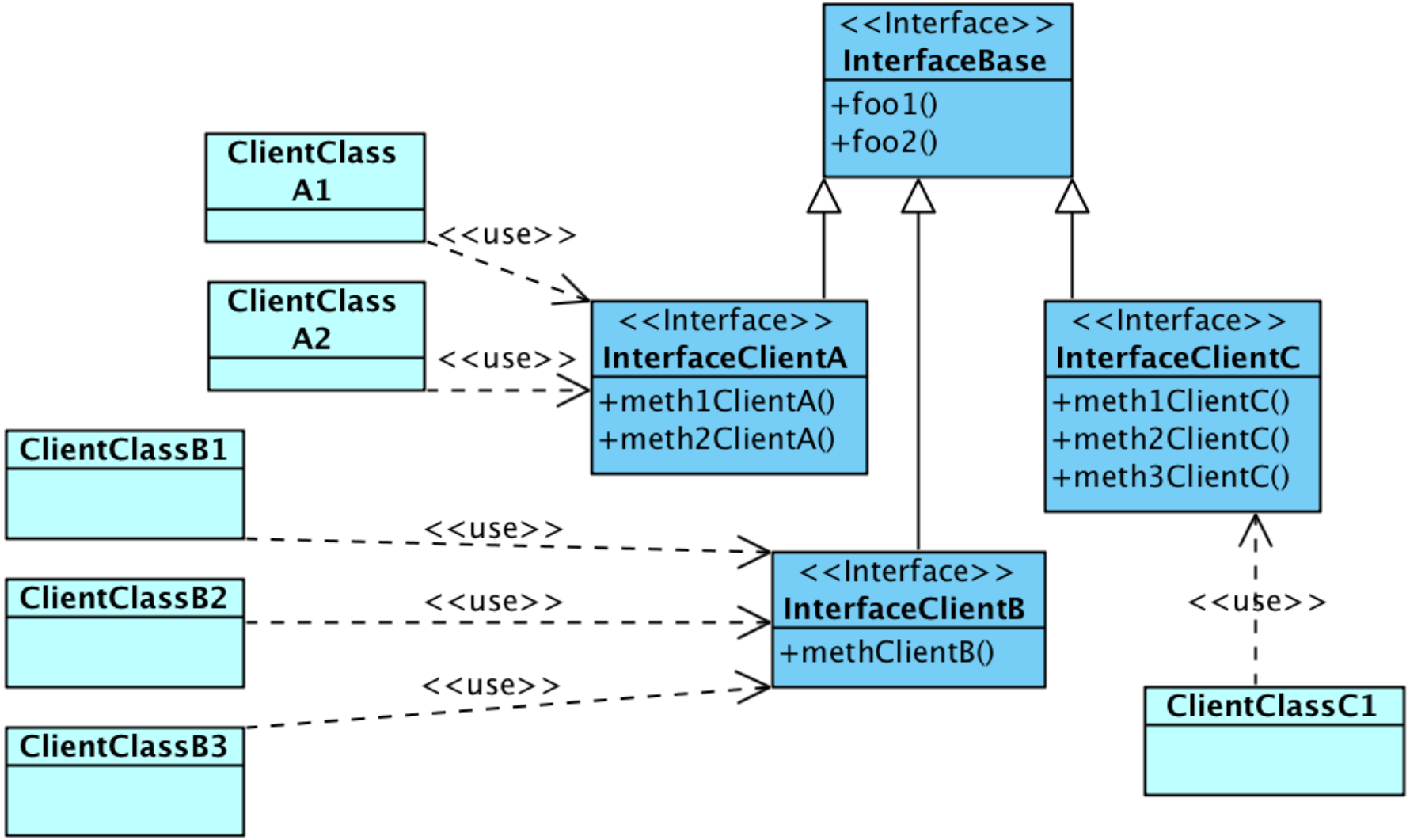
- Toute classe **réalisant une BigInterface** doit implémenter chacune de ses fonctions

→ Confusion, on ne sait pas exactement ce que chaque classe utilise de l'interface sinon en allant voir le code implémenté…



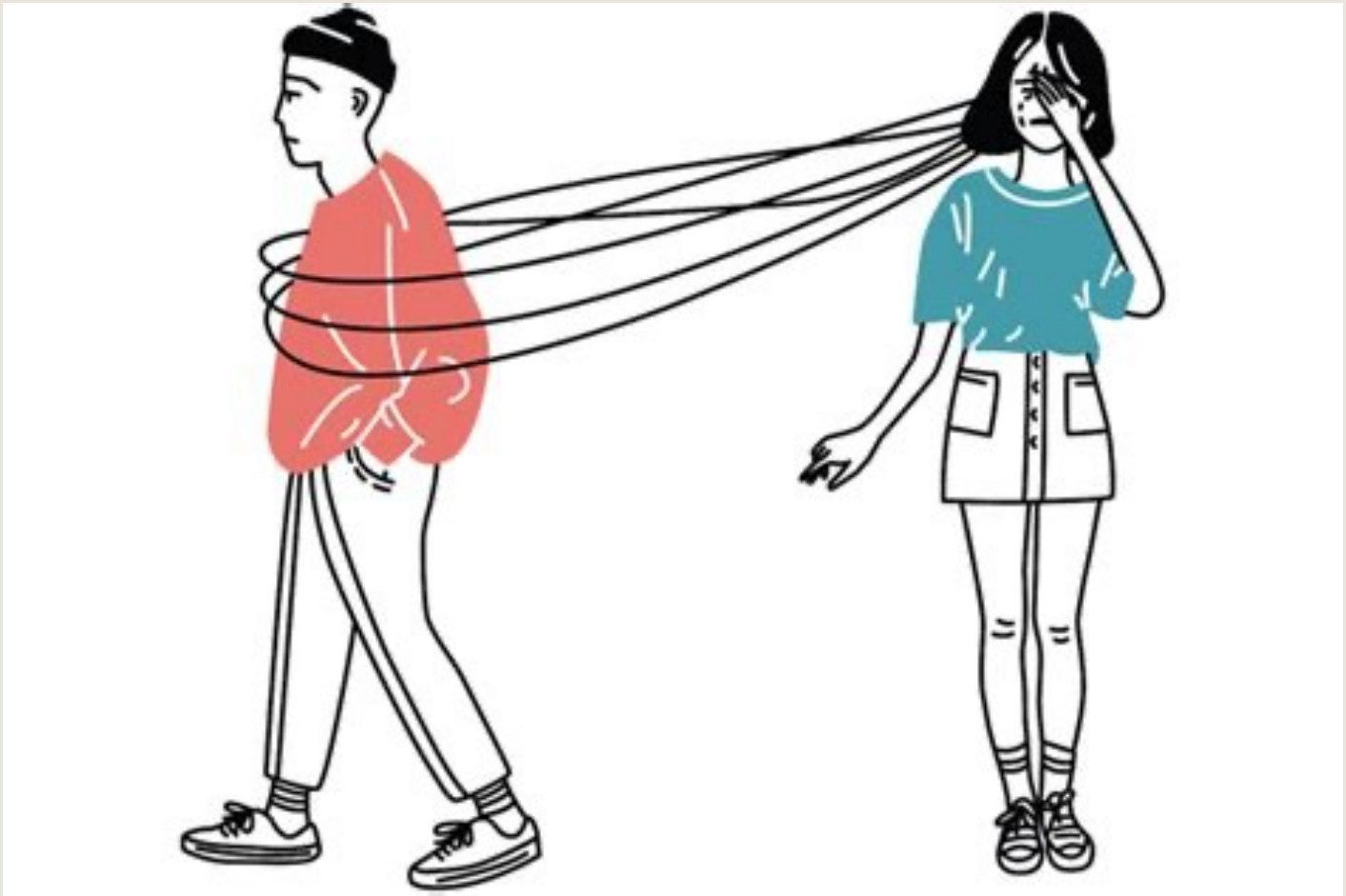
ISP

Séparation des interfaces (suite)



Ex. la classe List de .NET

- Elle implémente de multiples interfaces, dont le rôle est **explicité par leur nom** :
  - IList,
  - ICollection,
  - IReadOnlyList,
  - IReadOnlyCollection,
  - IEnumerable



Dependancy inversion

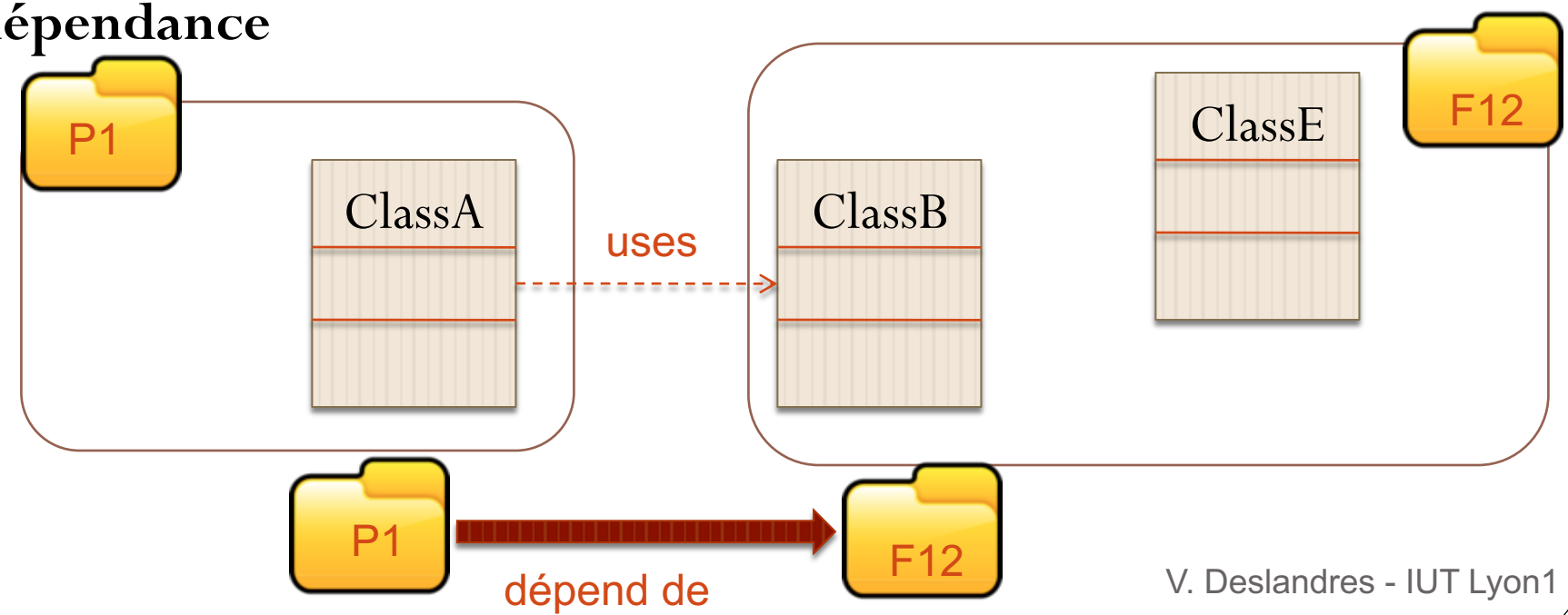
Inversion des Dépendances ou Inversion de contrôle (IoC)

DIP

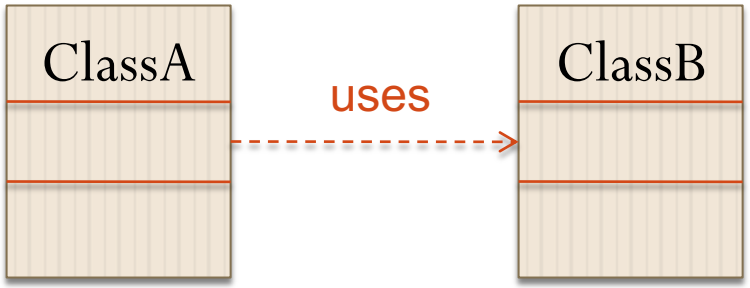
Inversion des Dépendances

- "Program to an interface, not to an implementation"
- *« La dépendance doit aller dans le sens de l'abstraction. Les composants de haut niveau ne devraient pas dépendre des composants de bas niveau »*
- **Illustration** : dans un logiciel, on peut être amené à dupliquer certains codes (par ex. persistance) pour s'adapter au *fwk*, selon la BD utilisée. On préfèrerait que cette responsabilité (liées à la persistance, de bas niveau) soit du côté du *fwk* et non plus du logiciel (haut niveau).
- Objectif : **inverser la dépendance**

Une classe A d'un package Business utilise la classe B d'un autre package (Persistance)



Aparté : que signifie « A utilise B » ?




```
public ClassA {
   public static void main(String[] args)
      ClassB b = new ClassB();
      b.someMethod();
   }
}
```

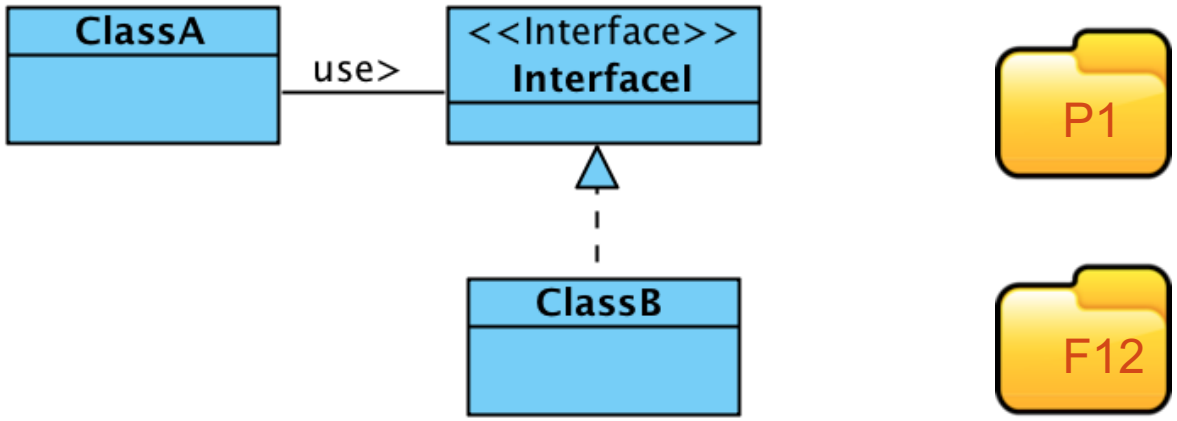
Inconvénient :

ClassA a besoin de connaitre le code de ClassB (b.someMethod())

- Code figé, statique
- Disperse les dépendances dans le code

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

DIP



- 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 :

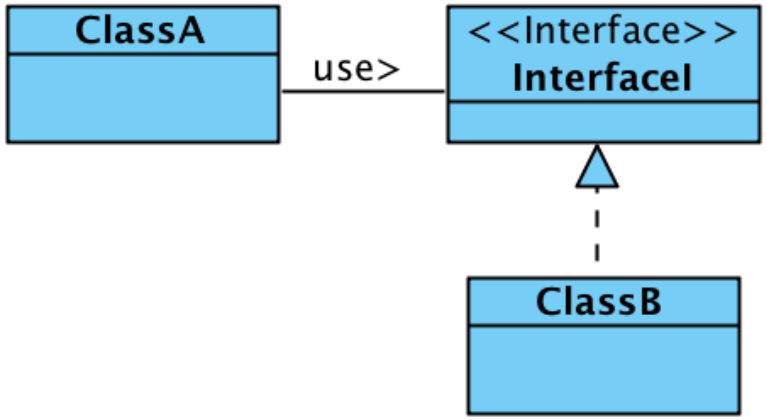
Injection de la dépendance

- 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** (ex. ci-après)
  - par mutateur : on passe l'objet **b à une méthode de A qui va par exemple modifier un attribut** (c'est le cas du framework Spring par ex.)
  - par interface
  - par les attributs : ex. ci-après

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

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

Injection de la dépendance via le constructeur




```
public Class A {
   InterfaceI instance;

   public Class A(InterfaceI inst) {
      this.instance = inst;
   }

   public void doSomeWork() {
       instance.someMethod();
   }
}
```

Et dans la classe appelante :

```
public static void main(String[] args) {
   ClassA a = new ClassA(new ClassB());
   a.doSomeWork();
}
```

Injection de la dépendance par les attributs

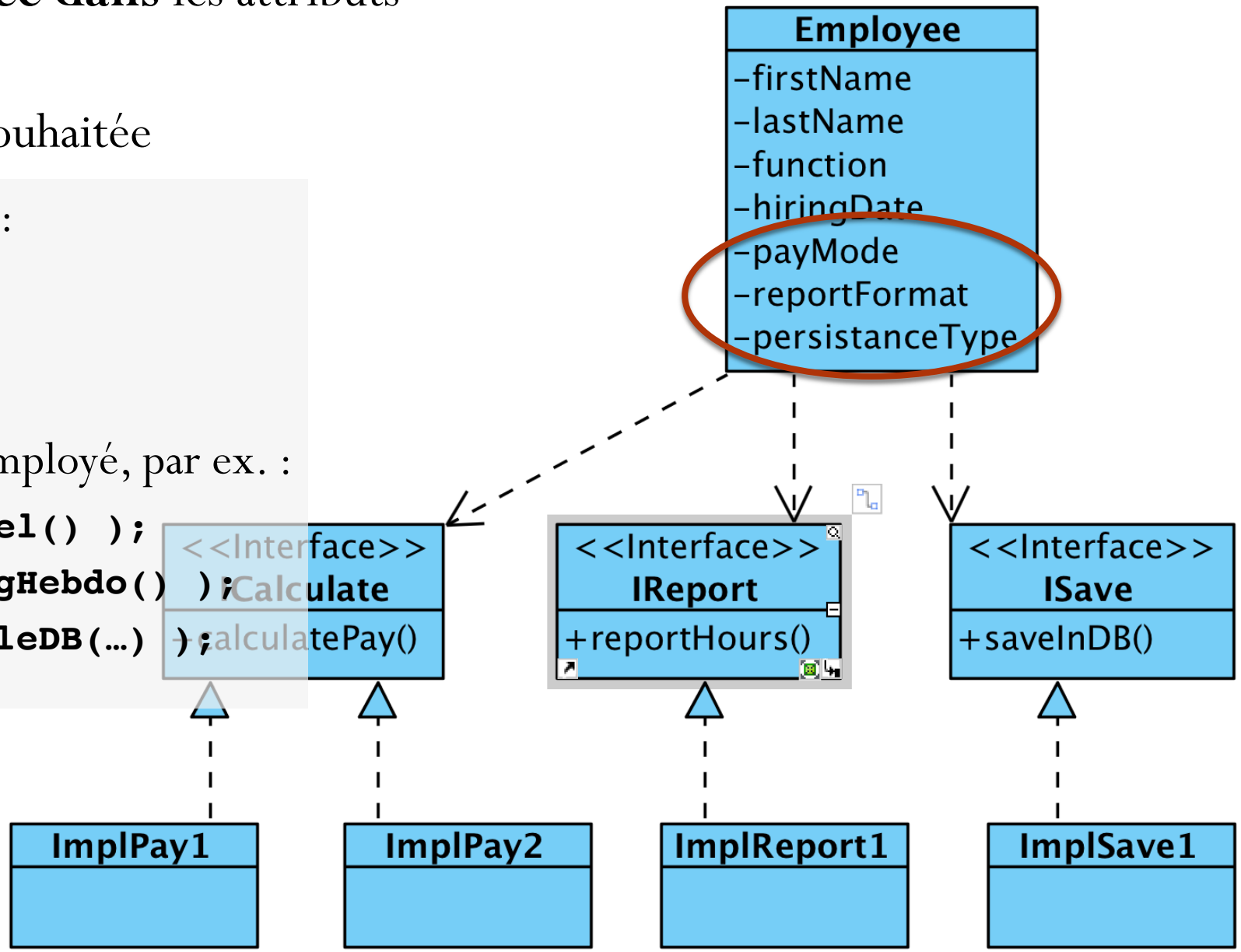
- Ex. classe *Employee* précédente
- On **va injecter la dépendance dans** les attributs d'*Employee*
- On lui injecte l'*implémentation* souhaitée

Déclaration dans la classe Employee :

```
ICalculate payMode ;
IReport reportFormat;
ISave persistanceType;
```

Lors de la création de l'instance d'employé, par ex. :

```
setPayMode( new PaiementMensuel() );
setReportFormat( new ReportingHebdo() );
setPersistanceType ( new OracleDB(…) );
```

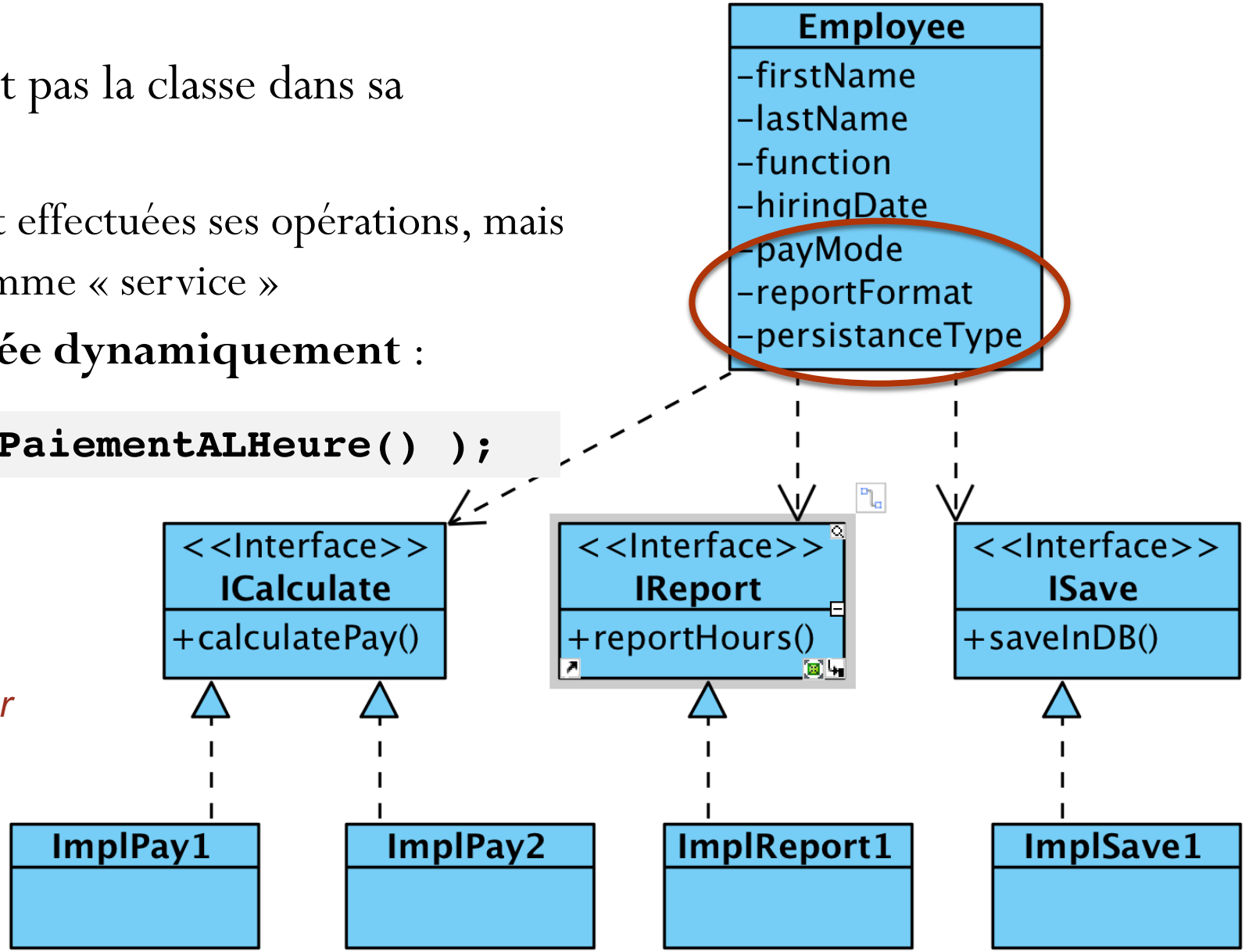


Classes d'implémentation

Injection de la dépendance par les attributs

- 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** :

```
unEmploye.setPayMode( new PaiementALHeure() );
```



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

Je retiens....

- **S** comme SRP : « Une seule raison de modifier le code »
- **O** comme OCP : « Un code ouvert à l'extension et fermé à la modification »
- **L** comme LISKOV : « On est d'abord l'enfant de son père avant d'être ce qu'on est »
- **I** comme Interface Segregation : « On exploite toutes les méthodes des interfaces utilisées »
- **D** comme Dependancy Inversion (IoC) : « Dépend d'une interface / classe abstraite, pas d'une implémentation »