POLYTECH PARIS-SUD

Année 2013-2014

Polytech'Paris-Sud – 4ème année
Département Informatique

« Les Design Patterns »

Frédéric VOISIN

Département Informatique

Bibliographie:

E. Gamma et *alli* : « Design Patterns : elements of reusable Object-Oriented Software », Addison Wesley (en français chez ITPS)

J. Cooper : « Java Design Patterns : a tutorial », Addison-Wesley Longman, March 2000

L. Debrauwer: *Design Patterns pour Java - Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java [2ème édition],* ENI, 2009

« Design Patterns » ?

- « *Pattern* » = modèle, patron (Harrap's) ici « de conception objet »
  - Des **modèles de Programmation à Objets** comme il existe des modèles de programmation impérative:

    recherche linéaire/dichotomique/avec sentinelle/ …;

    parcours de graphes en profondeur/largeur, etc.

  « Recurring patterns of classes and communicating objects »

- Quelques difficultés classiques:
  - Héritage vs Composition vs Délégation ?
  - Classes concrètes *vs* Classes abstraites *vs* Interfaces ?
  - Création d'instances directe ou indirecte ?

- Capturer l'expertise et la rendre accessible aux non-experts:
  - **garde-fous** contre un certain nombre de pièges classiques
  - **prévoir la réutilisabilité et l'évolutivité**

- les noms des patterns = vocabulaire pour mieux communiquer

Plus techniquement…

Les pièges classiques:

- schéma de création d'instances

  ```
  a = new A();
  ```

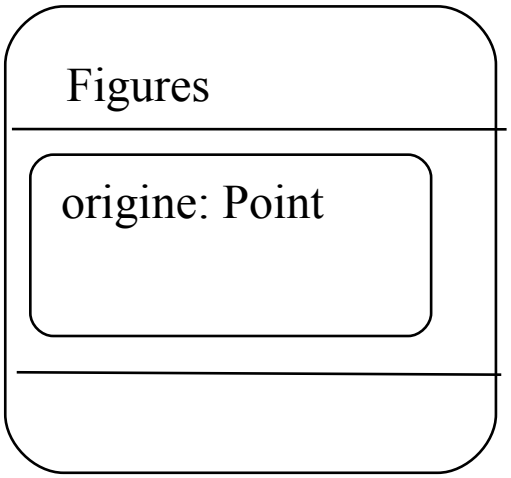
  ou `a := O.create()` où `create` est `static` ?

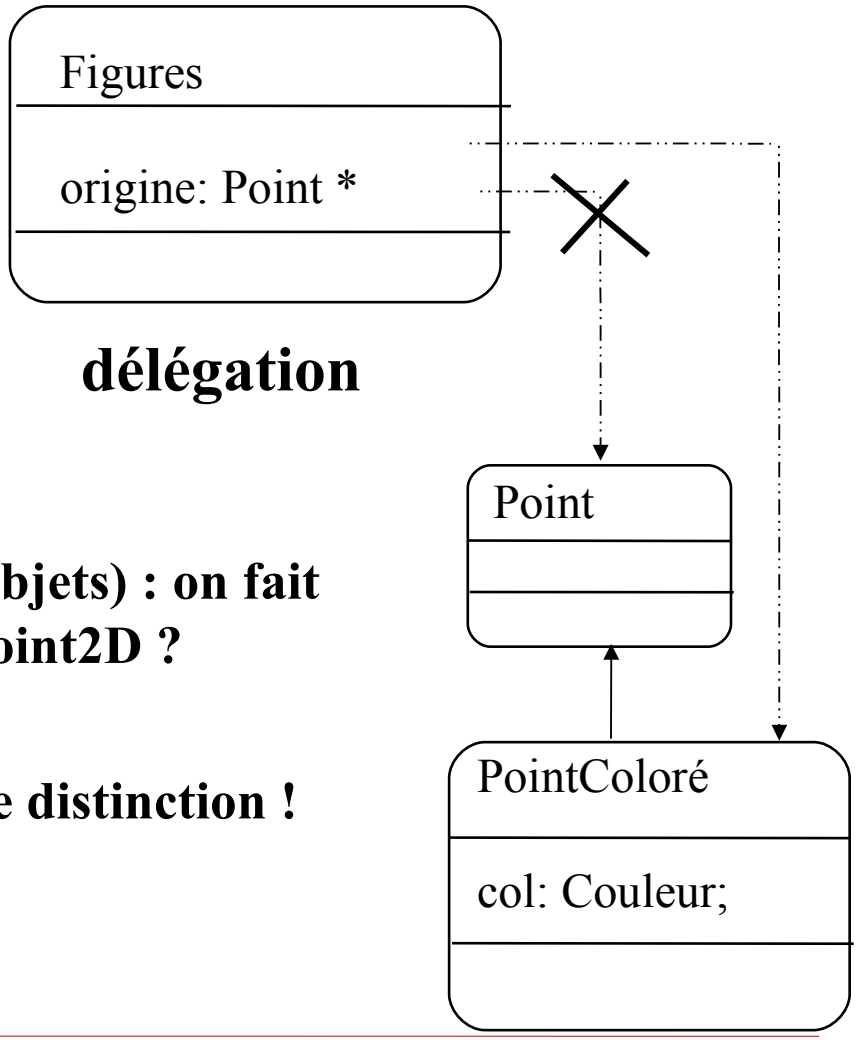
  ou `a := o.create()` où `create` est une méthode d'instance qui retourne statiquement un `A` mais peut être redéfinie (liaison dynamique sur `o`)
- référence à des classes explicites
- usage « intempestif » de l'héritage
- explosion combinatoire des classes (composition de « features »)

Les solutions proposées:

- privilégier la **composition** et la **délégation** sur l'héritage
- insister sur les **classes abstraites/interfaces** comme moyen de forcer des interfaces uniformes; les « clients » se basent sur ces interfaces
- **liaison dynamique de fonctions**
- **Schémas « typiques » de connexions entre objets**

Composition *vs* Délégation





Composition (à la C++ par inclusion d'objets) : on fait comment pour passer d'un Point à un Point2D ?

En Java: que des références, donc pas de distinction !

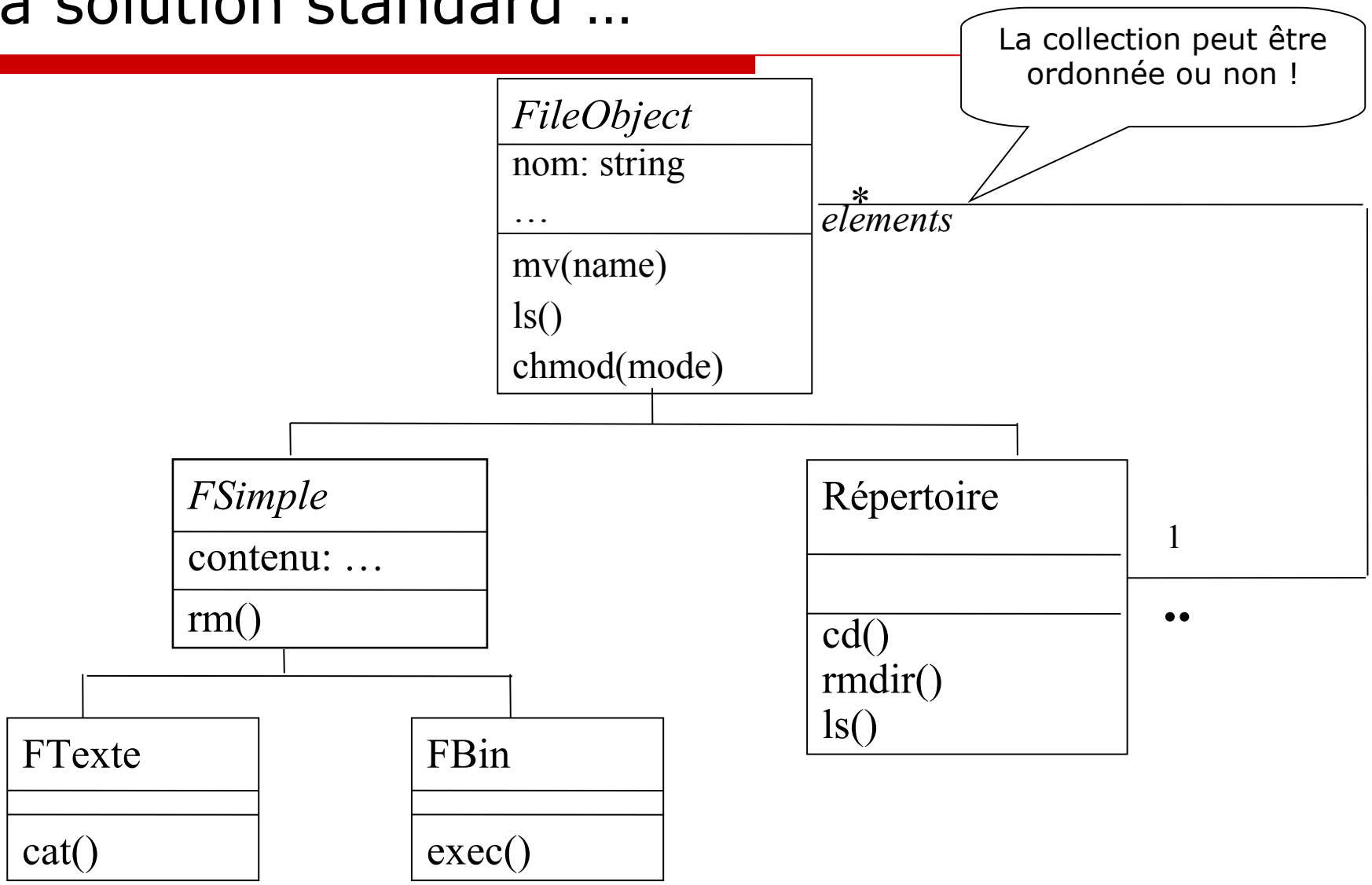
Les « patterns » peuvent-ils quelque chose pour vous ?

Exemple: une version « objet » du système de fichiers Unix

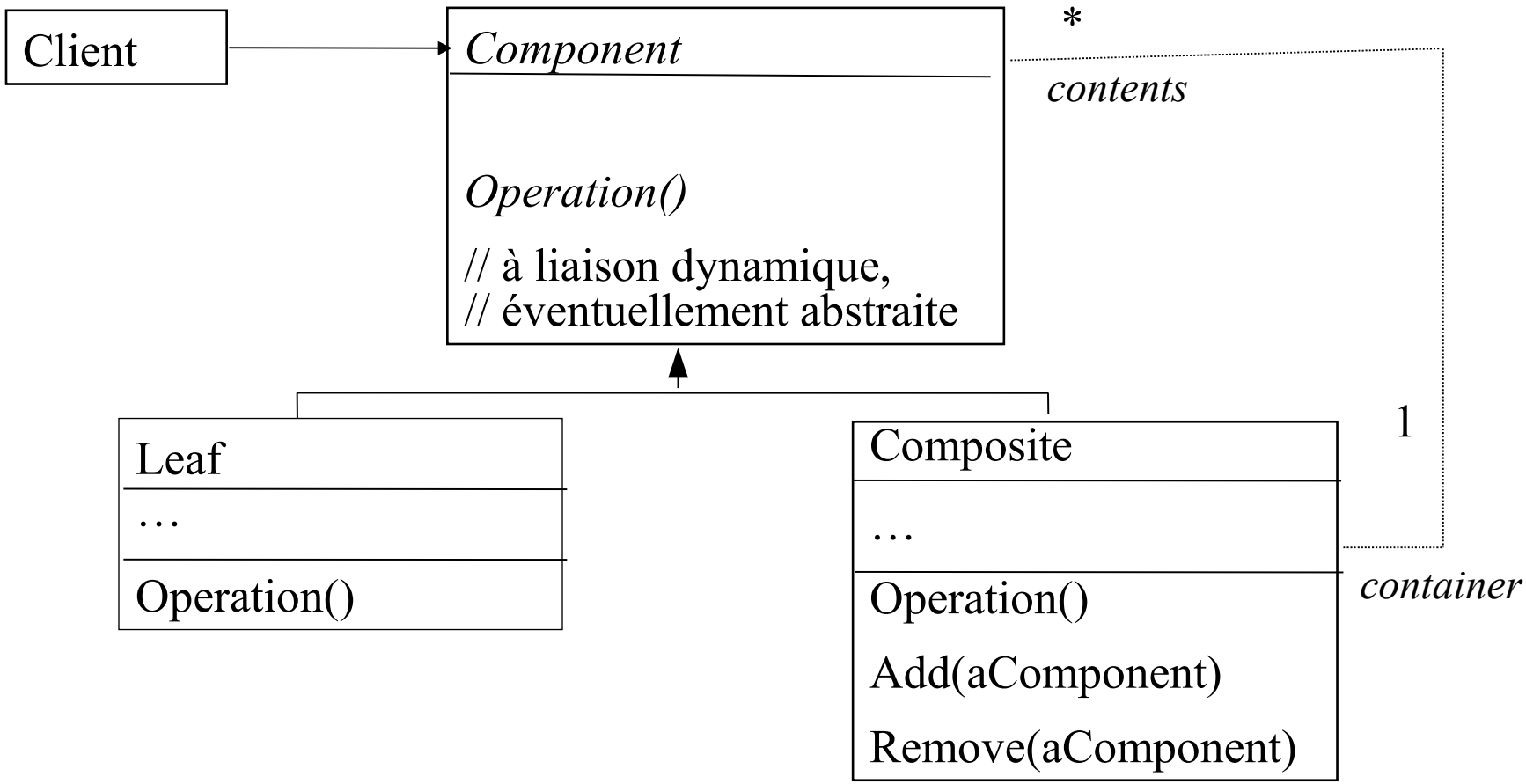
- Les fichiers simples et les répertoires ont un nom, des droits d'accès, un répertoire d'appartenance. On peut leur appliquer *mv, chmod, ls*
- Les fichiers simples ont des commandes spécifiques :
  - Fichiers « texte », avec `cat`
  - Fichiers exécutables, avec `exec`

  aux deux catégories on peut appliquer `rm`
- Les répertoires ont d'autres commands spécifiques : `rmdir`, `cd`, …
- Un répertoire peut contenir des fichiers simples ou des sous-répertoires, avec une **imbrication arbitraire**
- **Extension** possible à de nouveaux cas: les « liens symboliques » ?

La solution standard …



La solution standard = une instance de « *Composite* »




```
Composite::Operation() { for(Component c : children) {c.Operation(); }
```

Un pattern = un nom + 13 rubriques

But : une description en deux lignes
Synonymes : …
Motivation : un exemple illustratif
Applicabilité : quand l'utiliser ? Pour éviter quoi ?
Structure : un diagramme UML des classes et participants
Participants : les rôles des participants de la structure
Collaborations : les interactions entre les participants
Conséquences : le pour et le contre; quelle flexibilité ?
Implémentation : les techniques à utiliser, les pièges (C++ surtout)
Exemple de code : souvent en C++ (une version Java en cours…)
Usages connus : références à des systèmes existants
Patterns associés : les patterns similaires ou complémentaires

Les classes de patterns

3 sortes de patterns: *creational, structural, behavioral*

- « creational » : les mécanismes de création d'instances:
  - *abstract factory*, builder, *prototype*, **singleton**,…
- « structural » : lié à la décomposition en classes et en objets
  - **state**, *bridge*, **composite**, *decorator*, …
- « behavioral » : lié aux comportements dynamiques *iterator*, ***command***, *mediator*, **observer**, *memento*…

un classique en IHM: le modèle MVC (Model-View-Controler)

MCV = *Observer* + *Composite* + *Strategy*

Les « patterns » peuvent-ils quelque chose… (suite) ?

- **Intention**: permettre à une instance de **changer dynamiquement** de comportement **selon son état**, comme s'il changeait de classe !

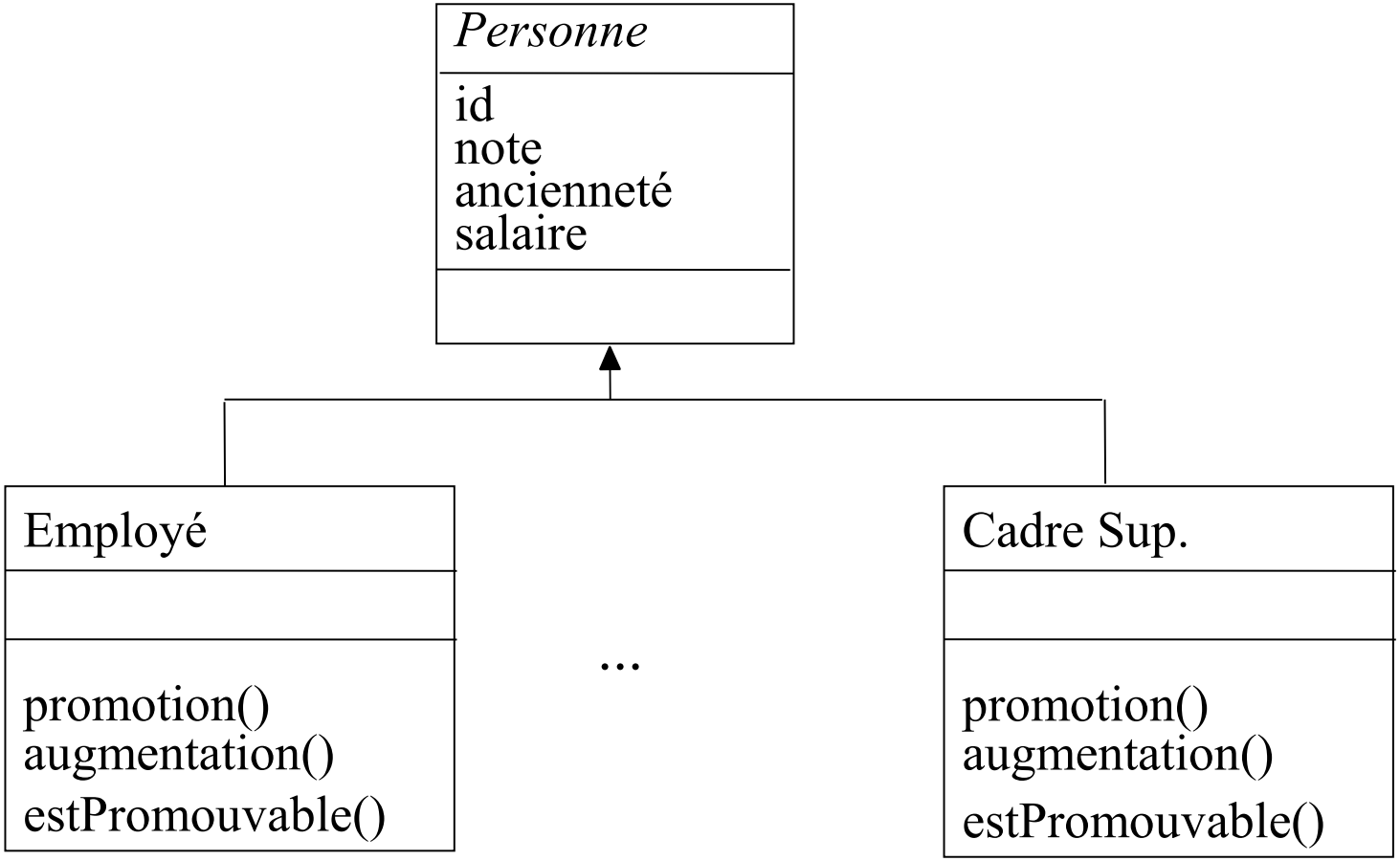
- **Motivation**: Une société a différents niveaux hiérarchiques parmi `Employé`, `Cadre`, `CadreSupérieur` et `Directeur`.

Chaque employé est caractérisé par :

une identification
une catégorie (son statut)
une ancienneté**,** une note attribuée par la DRH.
un salaire brut annuel

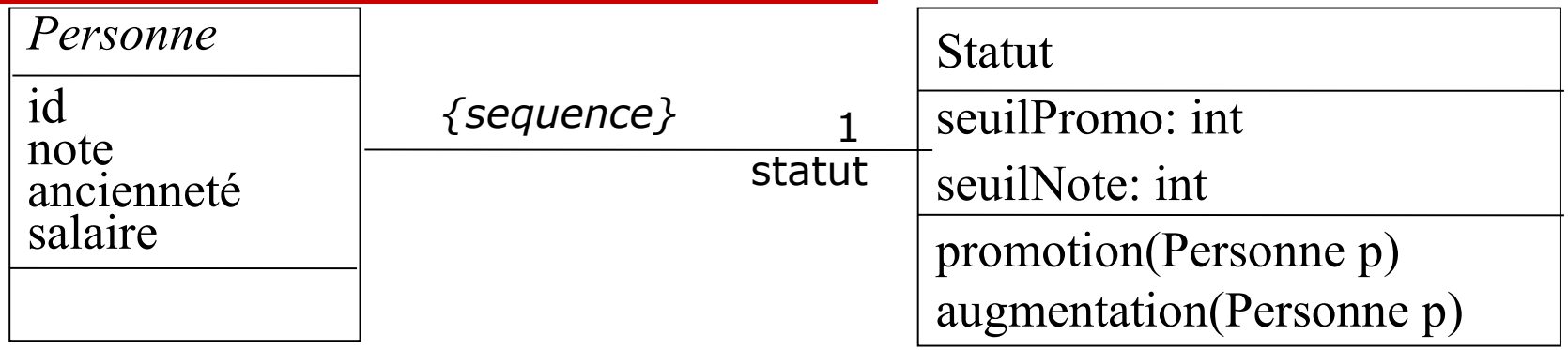
- Les augmentations de salaire et les promotions dépendent de la note, de l'ancienneté et **de la catégorie** …
- Une promotion fait **changer de catégorie**
- On risque d'avoir besoin de créer des catégories
- On risque de modifier les règles de promotion, de promotion, …

La solution intuitive est la mauvaise !!!



Comment changer de catégorie sans changer d'instance ?

Une autre pseudo-solution (?)

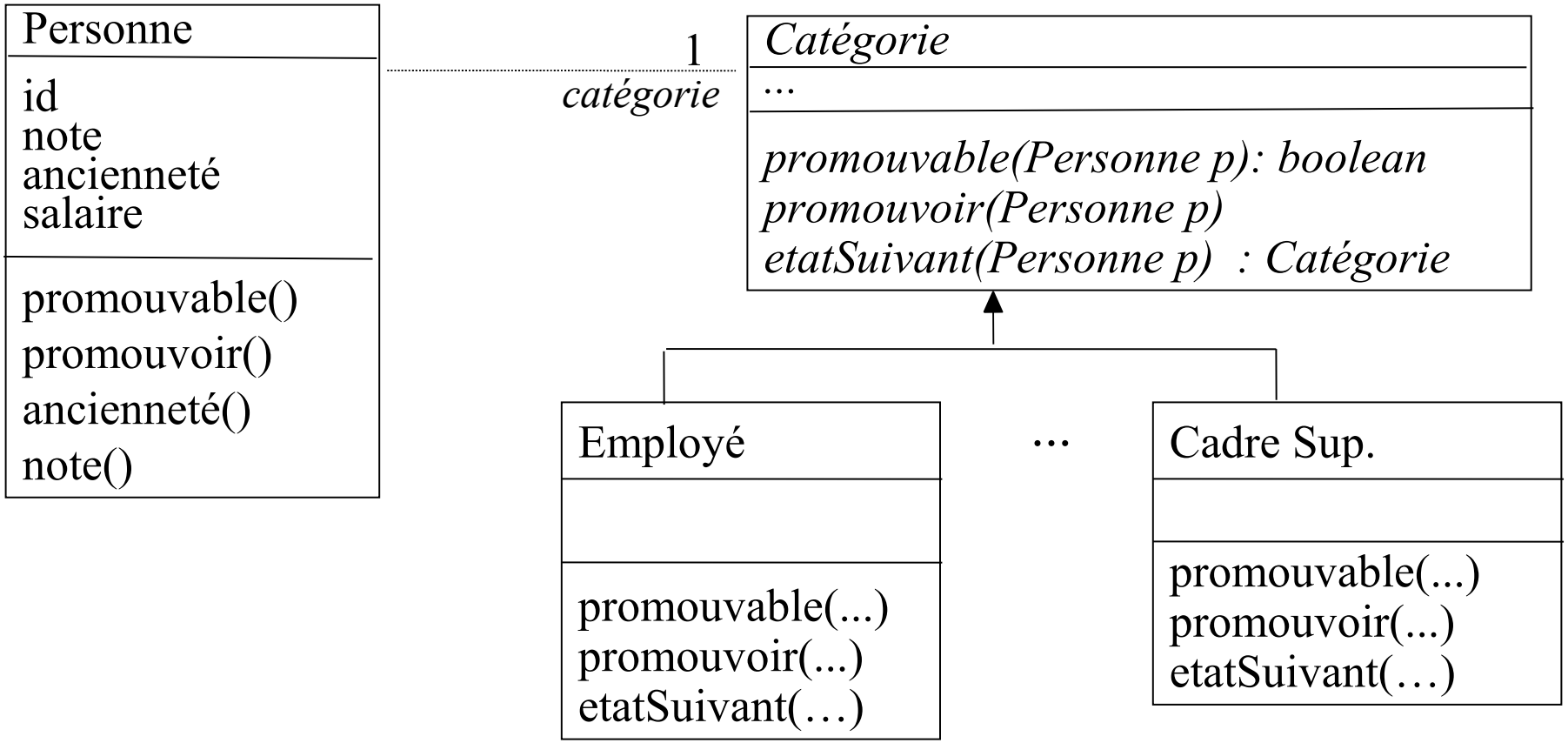


Une collection (statique, privée, ordonnée) d'instances d'une unique classe concrète :

- chaque instance représente un statut, avec ses deux valeurs de seuil caractéristiques.
- l'ordre dans la séquence représente la hiérarchie
- Les méthodes, uniformes ou presque, sont définies par la classe.

Que faire si les règles d'augmentation ne sont pas/plus uniformes ?
Pas de redéfinition possible de la méthode `estPromouvable`
Attention à la création (ordre, non duplication) des instances de « Statut »

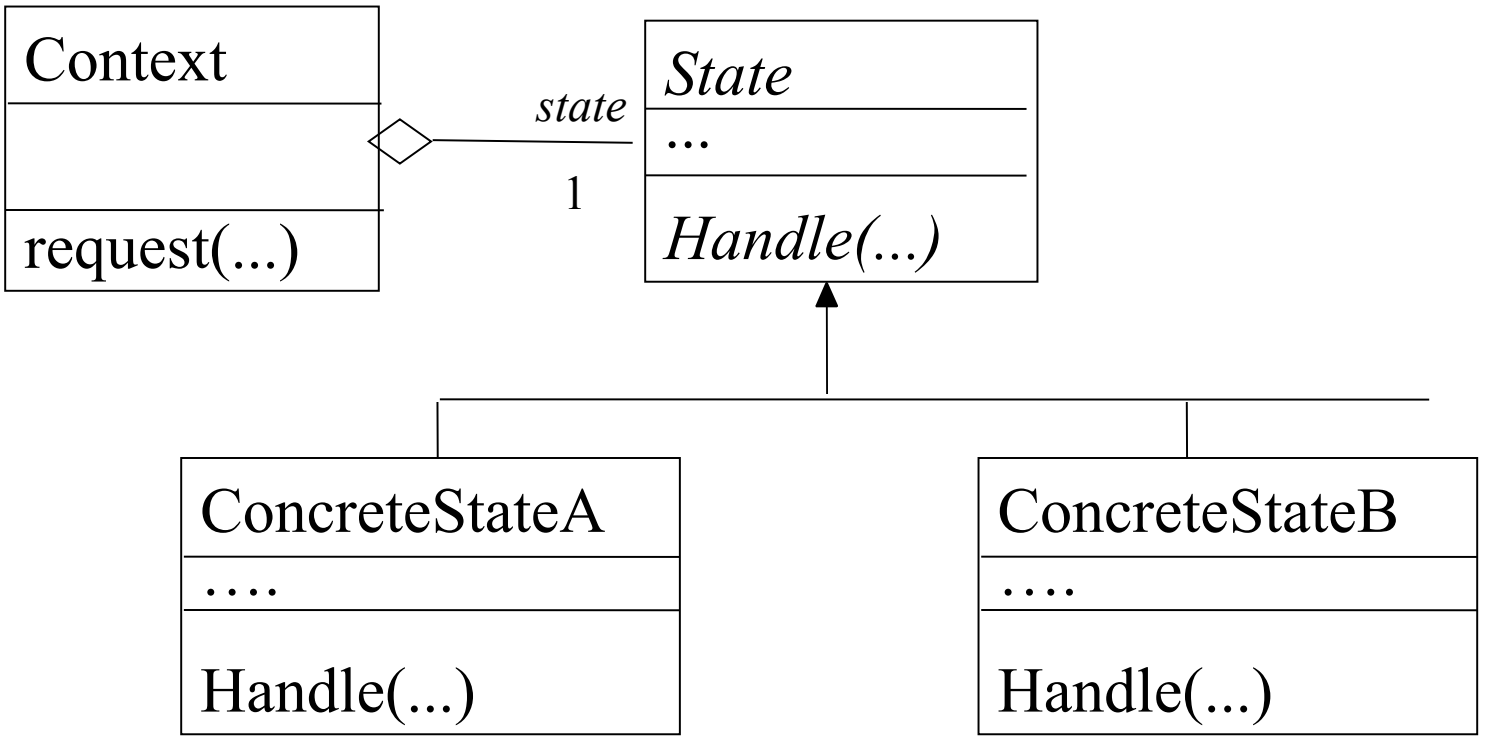
Une meilleure solution …



Personne::promouvable() { return (catégorie->promouvable(*this*)); }
Personne::promouvoir() { catégorie = catégorie->etatSuivant(*this*); }

Une instance de « State » en fait

Structure:




```
Context::request(…) { state.Handle(this); }
```

« State »: Utilisation et Participants

Applicabilité:

- le comportement **dépend de l'état** qui peut **changer dynamiquement**
- les comportements ont une **structure « par cas », selon l'état**

Participants:

- *Context* définit l'interface d'intérêt pour les clients et mémorise l'état courant (une instance d'une des sous-classes *ConcreteState*)
- *State*: définit l'interface qui encapsule le comportement associé à un des états possibles pour *Context*
- les sous-classes *ConcreteState* implémentent les comportements associés à un des états possibles pour *Context*.

« State » : les collaborations

- *Context* délègue les requêtes qui dépendent de l'état à l'instance courante de *ConcreteState*

- *Context* peut se passer en paramètre avec la requête pour que l'instance de *ConcreteState* puisse accéder au contexte

```
Personne::promouvable(){
   return catégorie->promouvable(this);
}
Employe::promouvable(Personne p) {
   return p.note() > 15;
}
```

- *Context* est l'interface pour les clients. Une fois le contexte configuré (e.g. statut à l'embauche), ils ne se soucient plus de l'état.

- *Contexte* et les sous-classes *ConcreteState* peuvent décider des transitions entre états (état suivant et conditions de passage)

« State »: Conséquences

- **Encapsule** dans la hiérarchie issue de *State* **les différences de comportements** dépendant de l'état et les partitionne selon les états

- Rend **explicite les transitions entre états**, par passage entre les différentes sous-classes possibles de *State*.

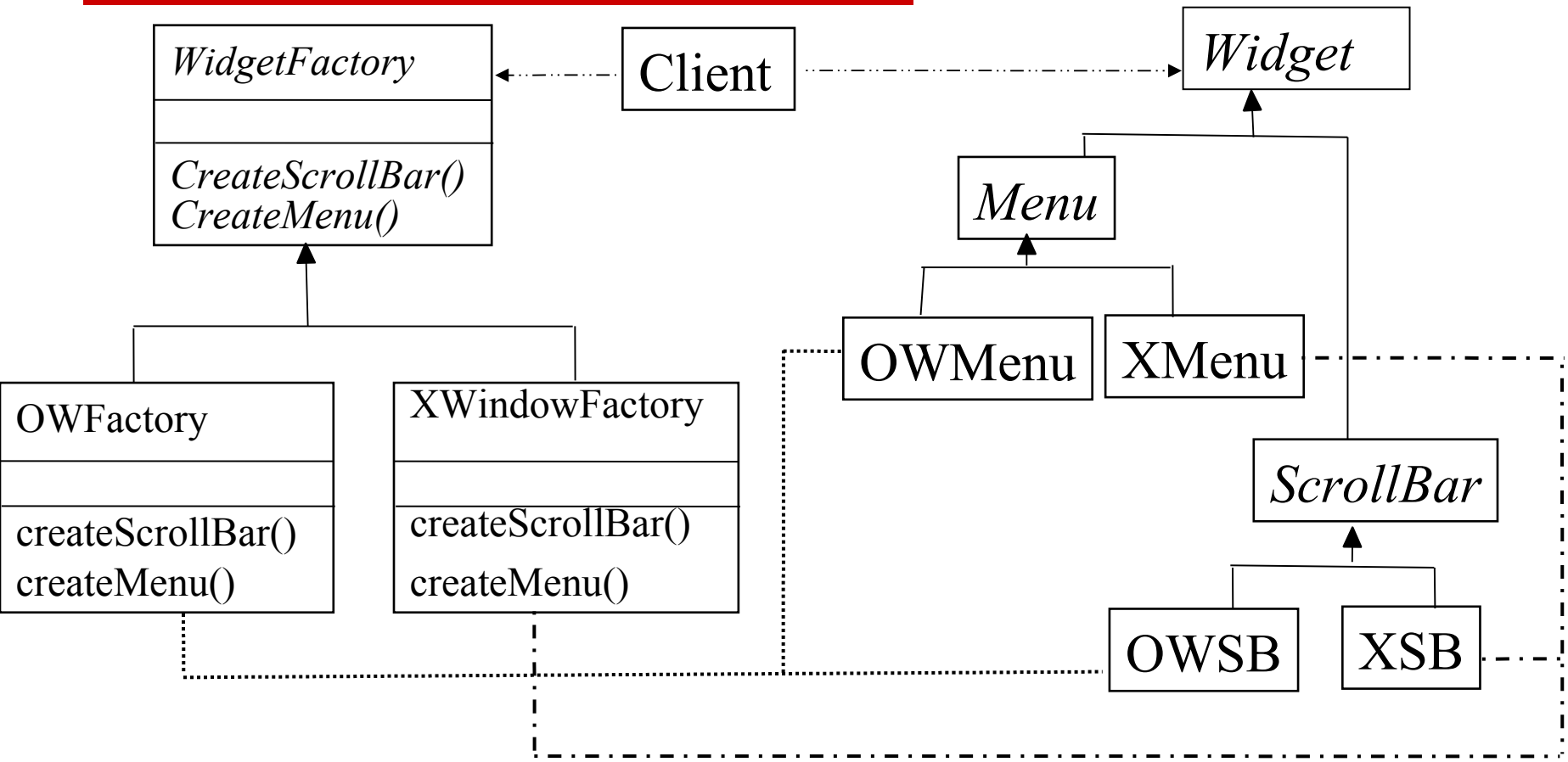
- Autorise le partage de `State`, si celui-ci ne contient pas de variable d'instance mais sert uniquement à mémoriser l'état (`Singleton` ?). Dans l'exemple, les sous-classes concrètes de `State` sont des singletons.

Pas que des avantages non plus: induit des classes concrètes de type `Singleton` *(?), avec des attributs et des méthodes tous statiques !*

Abstract Factory

- **But**: *une interface pour créer des familles d'objets similaires ou dépendants, sans spécifier de classes concrètes.*
- **Motivation**:
  - *définir une boîte à outils pour des interfaces utilisateurs, selon différents standards pour les fenêtres, ascenseurs, boutons,etc.*
  - *s'accommoder de deux hiérarchies orthogonales: celle des objets graphiques, et celle des systèmes de fenêtrage (« Look and Feel » …)*
- Définir une classe abstraite, « interface » du système de fenêtrage,
- Définir des classes abstraites pour les objets graphiques.
- Définir les sous-classes concrètes respectives
- Définir une fonction d'initialisation qui garantit qu'on ne crée que des instances du même standard => **Interdire l'accès direct aux constructeurs**
- Définir des assemblages complexes (« fenêtre avec deux scrollbars »)

Abstract factory : la structure



A l'initialisation, on utilise soit une instance de *OWFactory*, soit une instance de *XWindowFactory,* qui ne créeront que des instances de widgets homogènes…

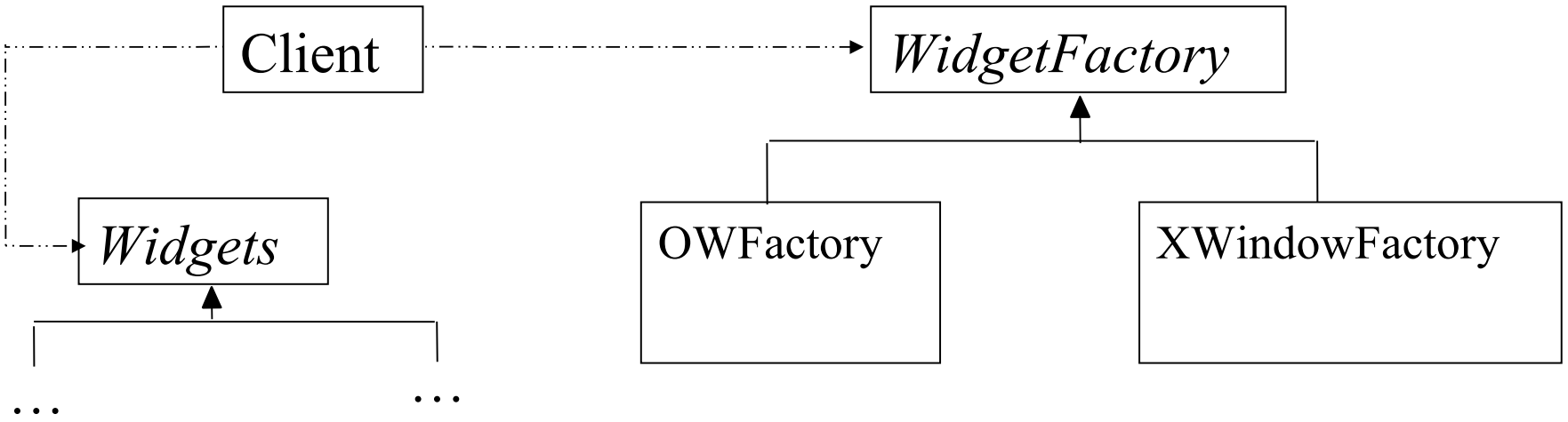
Abstract factory: l'applicabilité

- le système doit être indépendant de comment ses composants sont représentés, créés et composés (le client ne se repose que sur les classes abstraites de `Widget`)
- un système doit être configuré avec une parmi plusieurs familles de composants qui n'ont pas exactement les mêmes interfaces (détails de construction cachés dans les versions de `createScrollBar` dans `OWFactory` et `XWindowFactory`)
- une famille d'objets produit est conçue pour qu'ils soient utilisés ensemble
- on ne veut révéler que l'interface des produits d'une librairie (`Widget`) et non pas l'implémentation

Ça ne marche pas toujours si simplement !

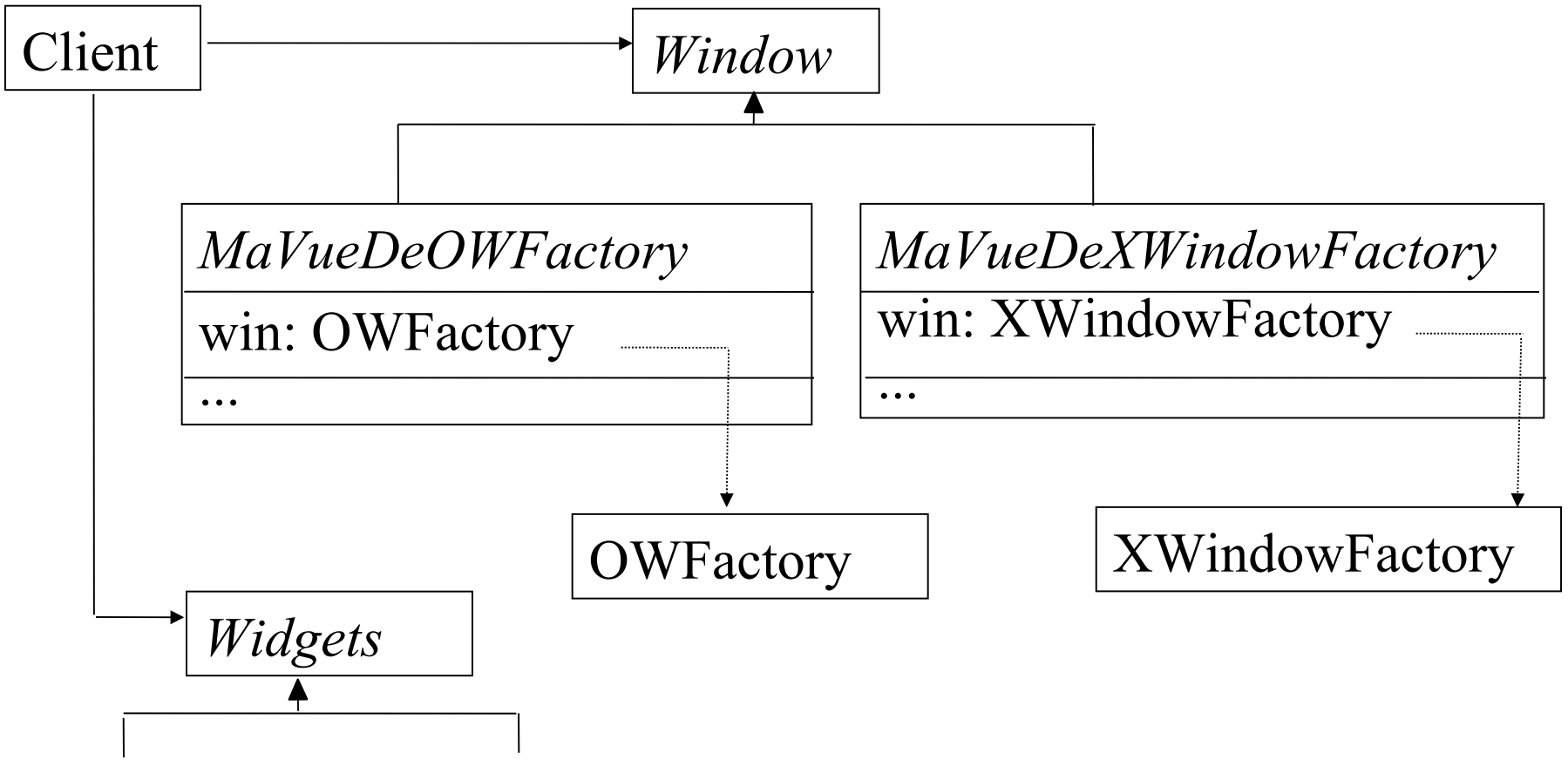
Et si *OWFactory* et/ou *XWindowFactory* existent déjà (i.e. ont leur propre hiérarchie) ou si les deux n'ont pas la même interface ??

On ne peut pas forcément avoir:



si les interfaces de `OWFactory` et `XWindowFactory` ne coïncident pas !

Encore un petit d'effort d'abstraction...



Abstraction via une interface commune + délégation vers l'implémentation

« Observer »

But: Maintenir la cohérence entre des **« vues » multiples** d'un objet;

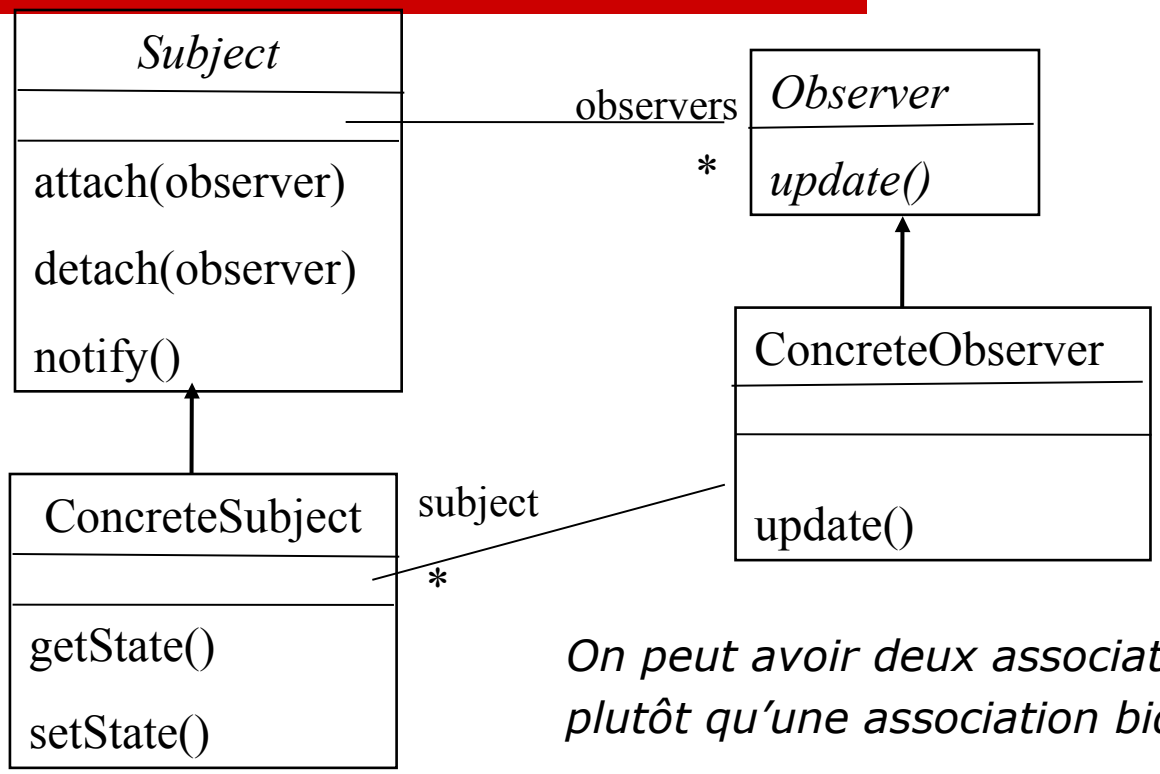
Motivation: découpler la gestion d'un objet de la mise à jour des objets qui en dépendent: ne pas avoir à gérer ces multiples « vues »

- un observateur peut observer plusieurs cibles; cette liste peut varier dynamiquement
- un observé peut être la cible de plusieurs observateurs; cette liste peut varier dynamiquement
- un observé **ne doit pas savoir comment/pourquoi il est observé** : ce n'est pas à lui de connaître les changements d'états « intéressants »

Collaborations:

- l'observateur s'abonne aux modifications d'état de l'observé;
- l'observé notifie tous ses abonnés à chaque changement d'état; chaque observateur interroge l'observé, décide s'il est intéressé par la modification et met à jour sa« vue » si besoin.
- L'observateur peut avoir besoin de conserver ses propres informations sur l'observé (observation de « transitions entre états » plutôt que de « l'état courant »).

« Observer » (la structure)



On peut avoir deux associations monodirectionnelles plutôt qu'une association bidirectionnelle

Repris en Java:

`interface` `observer` et classe `observable` à dériver…

En ajoutant des « contrôleurs » (interactions) on obtient MVC.

« Adapter »

But: convertir (déguiser ?) l'interface d'une classe en l'interface que les clients attendent. Faire coopérer des classes qui auraient sinon des interfaces incompatibles (revoir `AbstractFactory`)

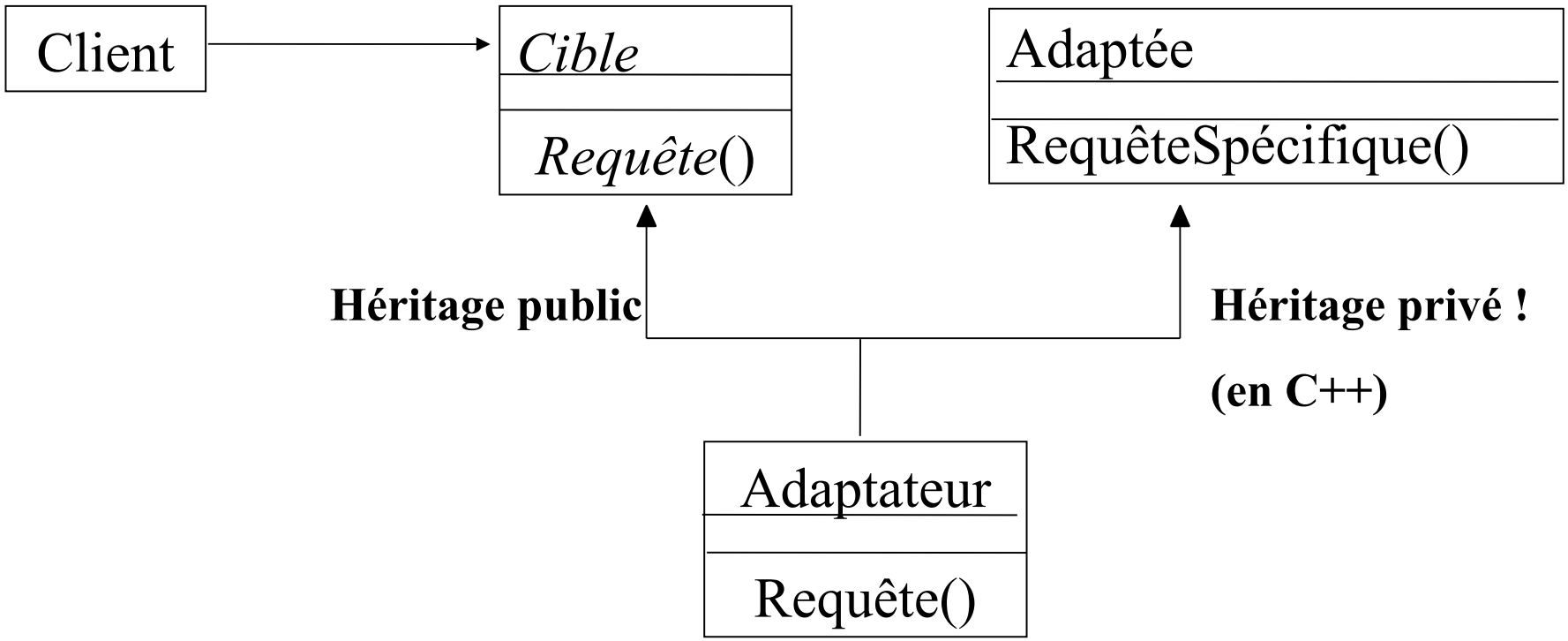
Motivation: donner une image uniforme de classes qui ont des comportements très différents (ex. objets graphiques et textuels)

Applicabilité:

- réutiliser une classe existante dont l'interface ne correspond pas
- créer une classe réutilisable qui coopère avec des classes non reliées, ou même qui n'existent pas encore
- (en tant qu'**adaptateur d'instances**) utiliser des sous-classes existantes, mais sans avoir besoin d'être une sous-classe de chacune

Existe sous forme d'adaptateur de **classe** ou **d'instance**

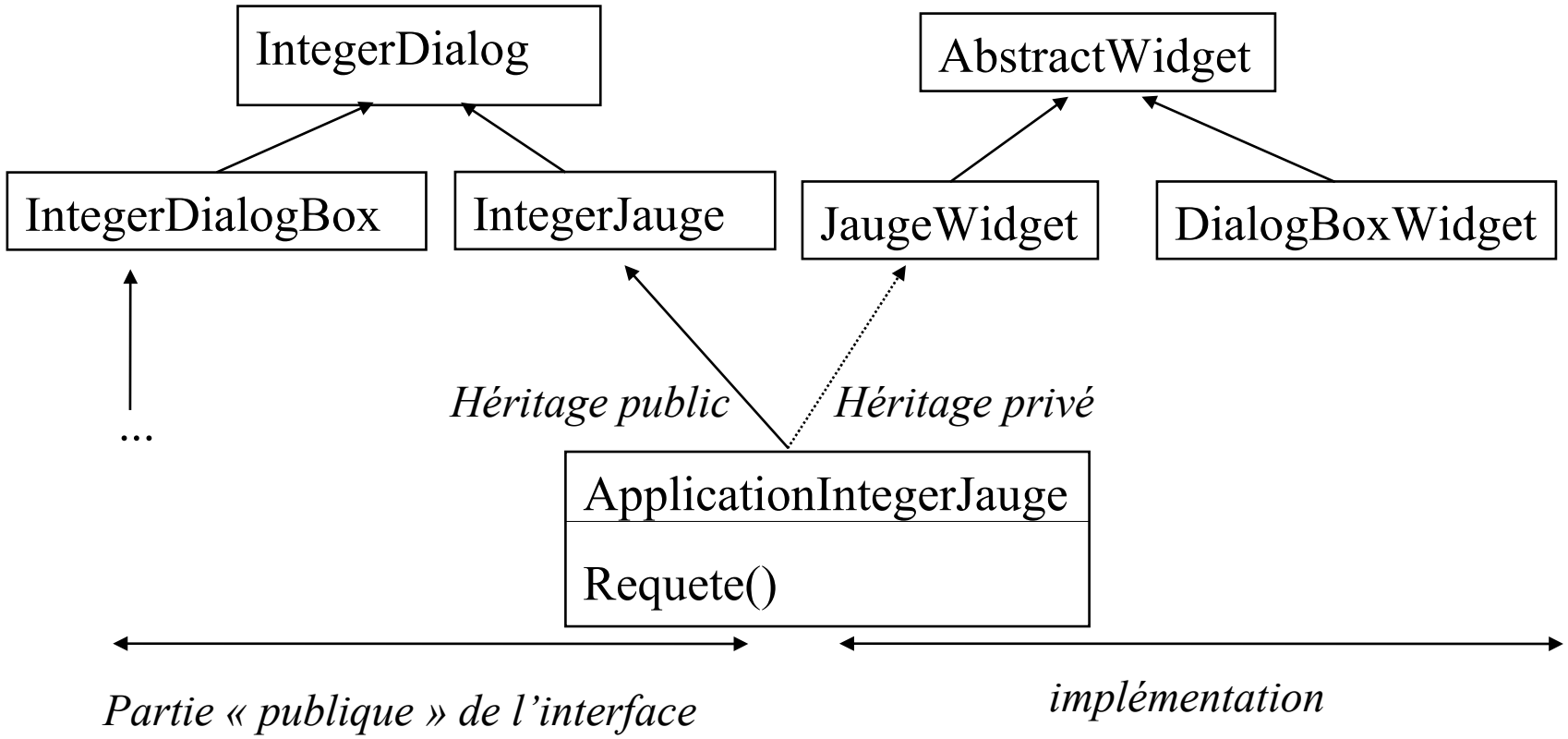
Structure (« class adapter »)




```
Adaptateur::Requête() { RequêteSpécifique(); }
```

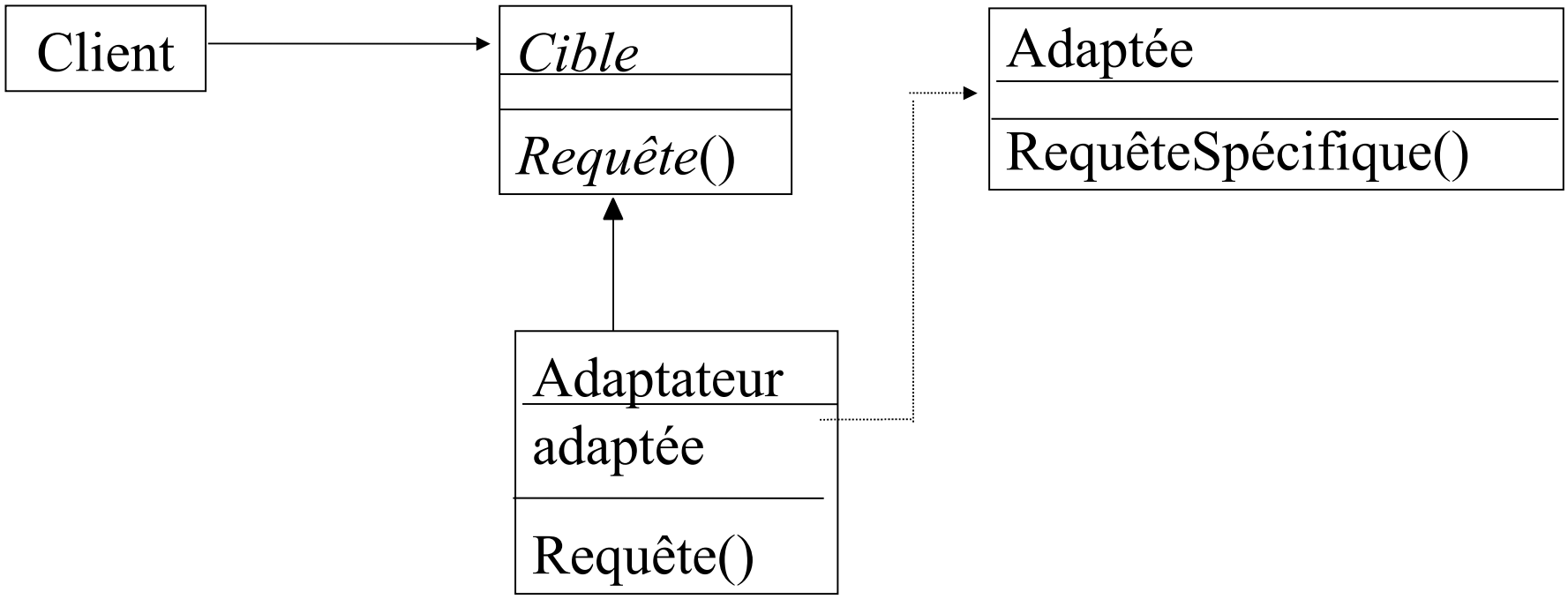
À base d'héritage **multiple et privé** (en C++ !)

En plus concret…



ApplicationIntegerJauge peut héritéer et redéfinir des méthodes qui proviennent tant de *IntegerJauge* que de *JaugeWidget*

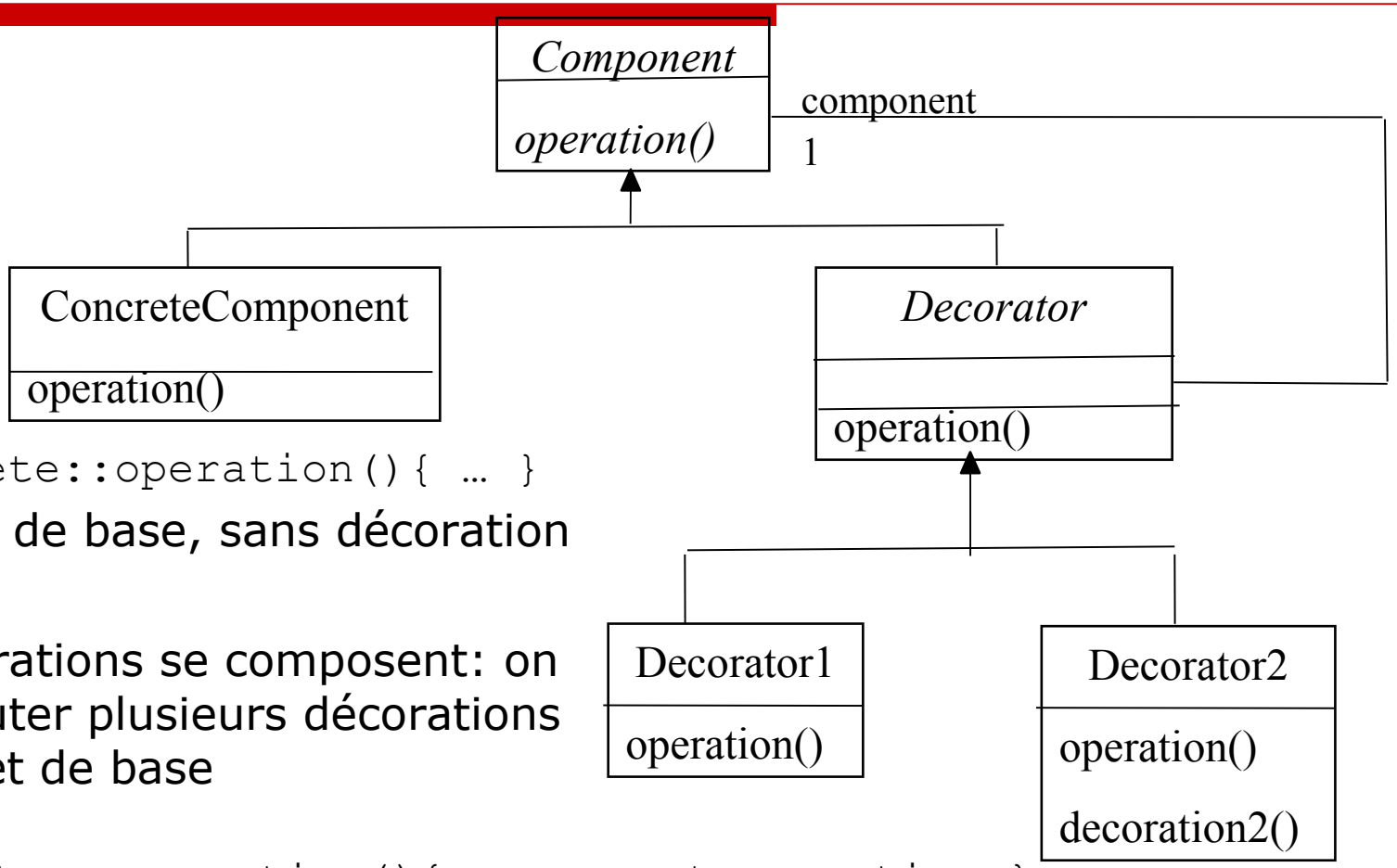
Structure (« object adapter »)




```
Cible::Requête() { adaptée->RequêteSpécifique(); }
```

La version du pauvre : simule l'héritage multiple par de la délégation explicite !

Le Pattern « Decorator »



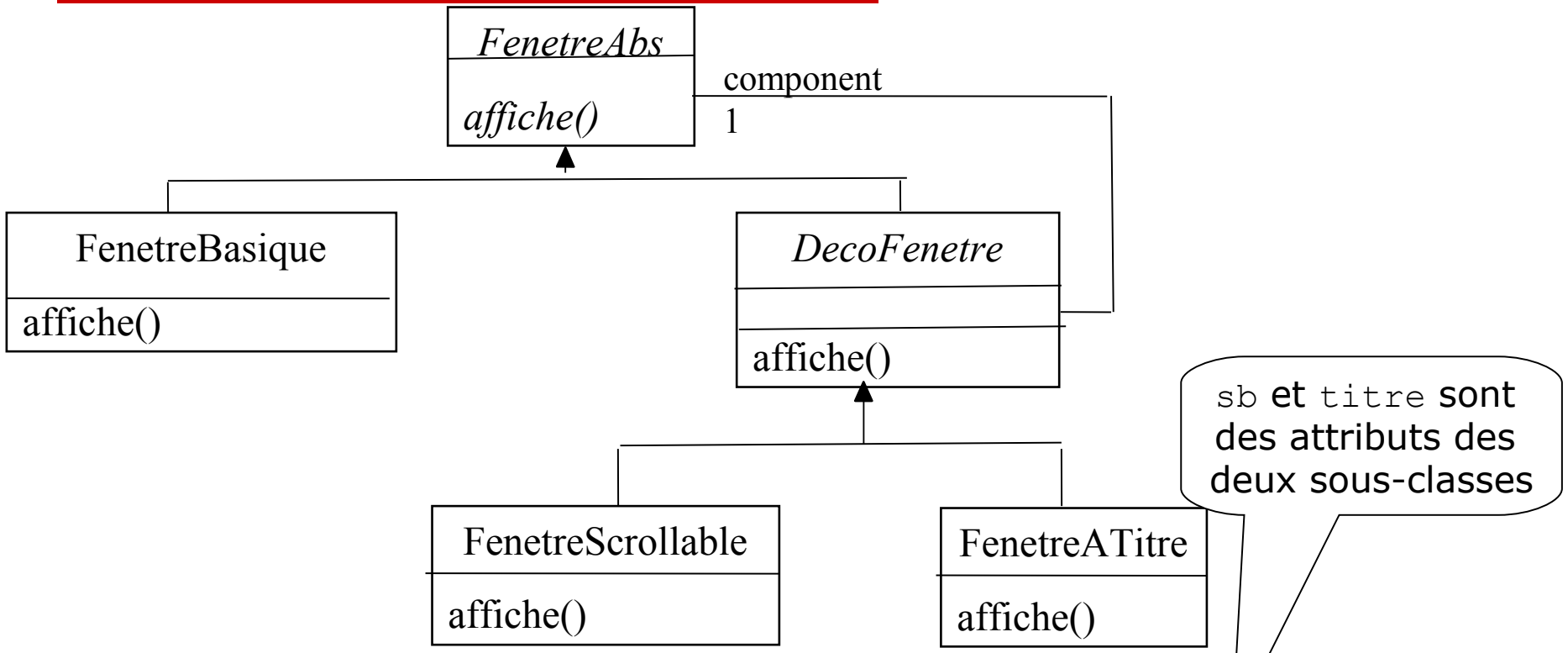

```
Concrete::operation(){ … }
```

L'objet de base, sans décoration

Les décorations se composent: on peut ajouter plusieurs décorations à un objet de base

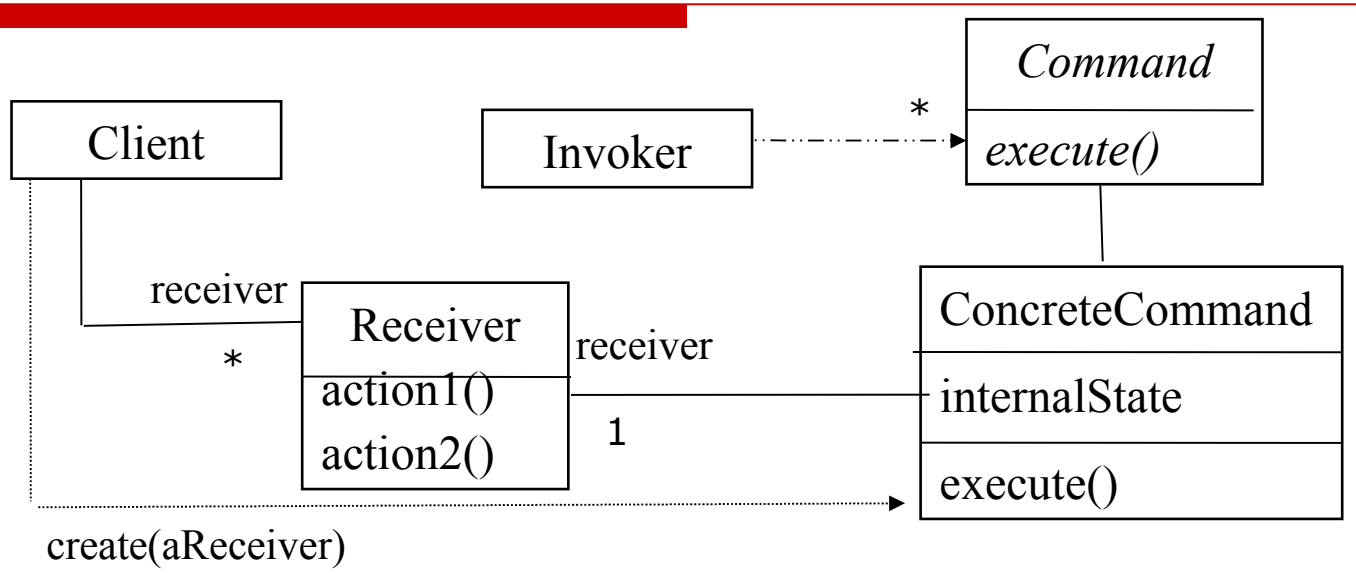
```
Decorator::operation(){ component.operation;}
Decorator2::operation(){ super.operation(); decoration2(); }
```

Exemple d'instance de « Decorator »




```
DecoFenetre::affiche() { component.affiche(); }
FenetreScrollable::affiche() {super.affiche(); sb.affiche();}
FenetreATitre::affiche() { super.affiche(); titre.affiche(); }
```

Le pattern « Command » …




```
ConcreteCommand::execute() { receiver.action(); …
```

- `ConcreteCommand` encapsule les commandes en gardant dans son état interne les paramètres à utiliser
- `Invoker` lance les commandes.
- Le client crée les instances de `ConcreteCommand`, leur fournit le receveur sur lequel la commande agira et les enregistre dans `Invoker`

« Command » (suite)

- Rien ne force le client et l'invocateur à être distincts !

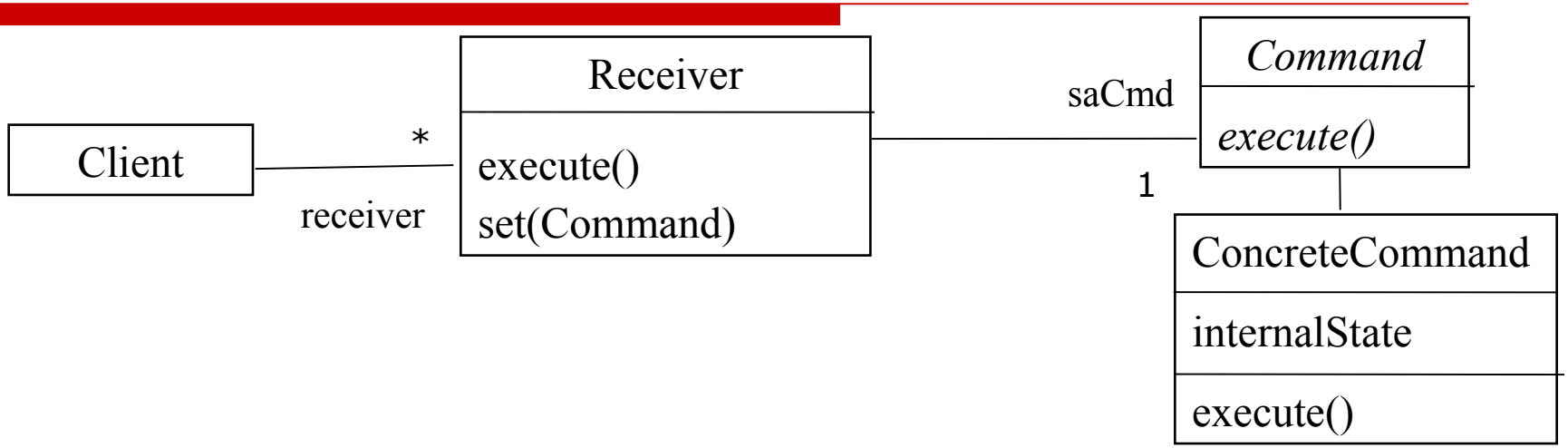
Exemple de distinction : l'invocateur est un menu de lancement des actions avec lequel le client interagit…

Typiquement: implantation de « callBacks »

- C'est la commande qui mémorise l'objet sur lequel elle agit : plus besoin de passer l'objet cible en paramètre pour lancer l'action
- C'est la commande qui contrôle comment l'action est exécutée (composition d'actions élémentaires; paramètres de l'action)
- Les « commandes » peuvent être regroupées, stockées dans des attributs, passées en paramètre, etc.

les commandes sont des objets de « première classe »

… et « Foncteur »



- Ici c'est le client qui agit sur les receveurs
- Le receveur mémorise « sa » (version courante de la) commande…
- En dérivant `Command` on obtient plusieurs versions possibles d'application d'une même commande à cet objet
- Généralisation à plusieurs commandes ??
- **Une autre version « objet » du passage de fonction en paramètre** : on l'encapsule dans un objet qu'on passe en paramètre ou qu'on stocke dans un attribut (« réification »)

Les patterns: les inconvénients

Ils ne dispensent pas de réfléchir (Applicabilité/Conséquences) :-)

- Encore faut-il les reconnaître !
- Un style à base de classes abstraites, plus lourd, qui peut déconcerter (« sur-généralisation ») ?
  - Peuvent entraîner une multiplication du nombre de classes.
  - Plus proches de la programmation que de la conception
  - Certains ne font que palier l'absence en Programmation Objets de mécanismes d'autres langages !
- Utiles comme catalogue de styles de hiérarchies de classes
- Si on a un problème qui ressemble à un pattern mais que la solution diffère, se poser des questions !
- Aident à se poser des questions auxquelles on n'avait pas pensé
- Étendus à des pattern de programmation parallèle ou distribuée