Chap.4 – Design patterns

V. Deslandres ©

BUT Informatique, s3

IUT de Lyon - Université Lyon 1

Sommaire de ce cours

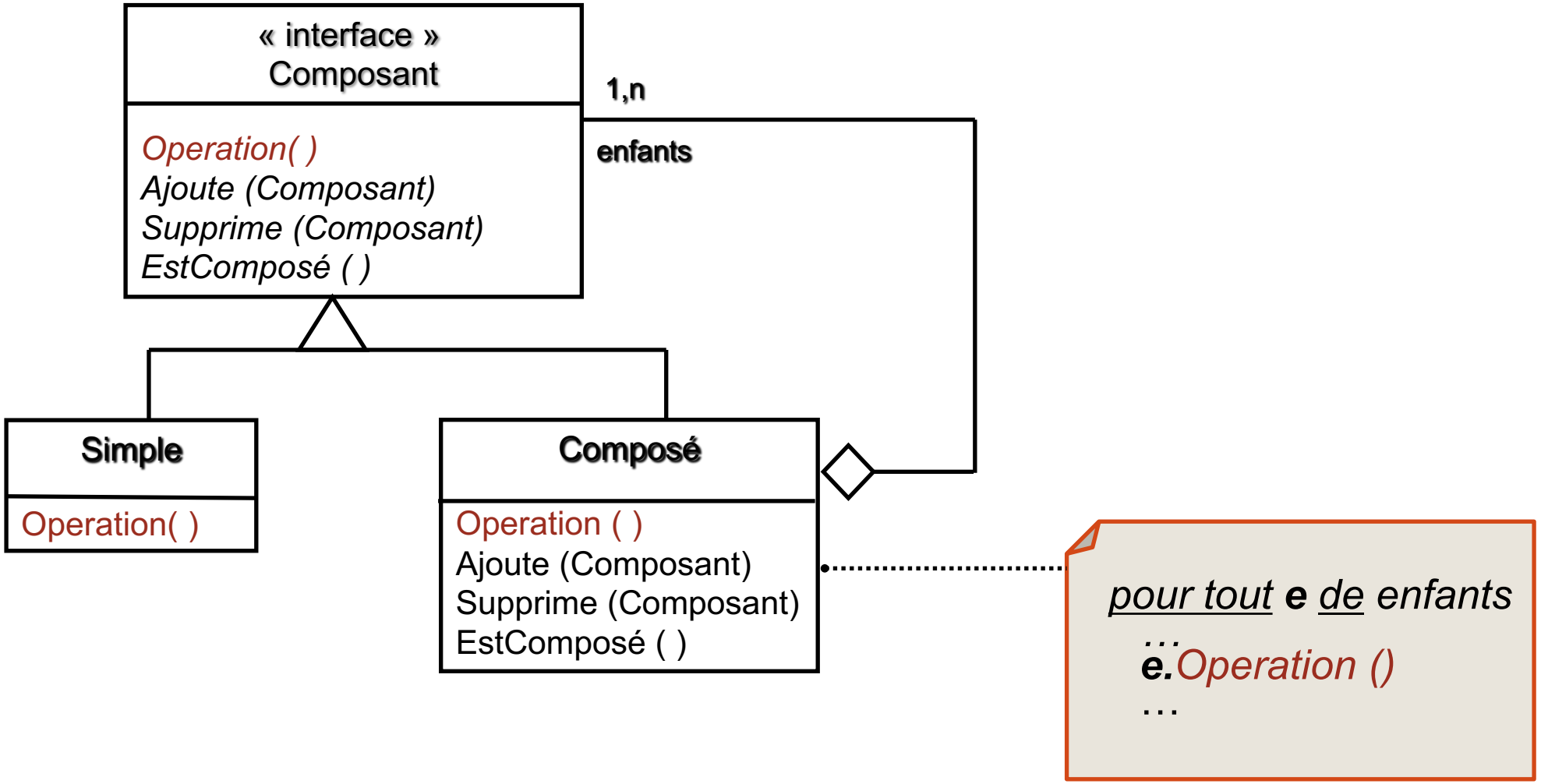
Le Pattern Composite

Patron de Structure

Patron Composite

- Un objet *Composite* est composé d'autres objets
  - Qui peuvent eux-mêmes être composé, ou des objets atomiques
  - Permet de créer une arborescence d'éléments
  - Les traitements s'effectuent sur les objets, indépendamment du fait qu'ils soient Composé ou Atomique
- Chaque objet a une méthode *Operation()* qui est appelée « de façon récursive » sur chaque toute l'arborescence
  - Utilise un itérateur, qui parcourt les agrégats jusqu'aux objets 'feuille'
  - Exemple : calculer le prix de l'objet global à partir du prix de chaque composant
- On pourrait ajouter une méthode *getEnfant()* dans un Composite, qui retourne :
  - Un container des enfants
  - Un itérateur sur la racine, etc.

Pattern Composite

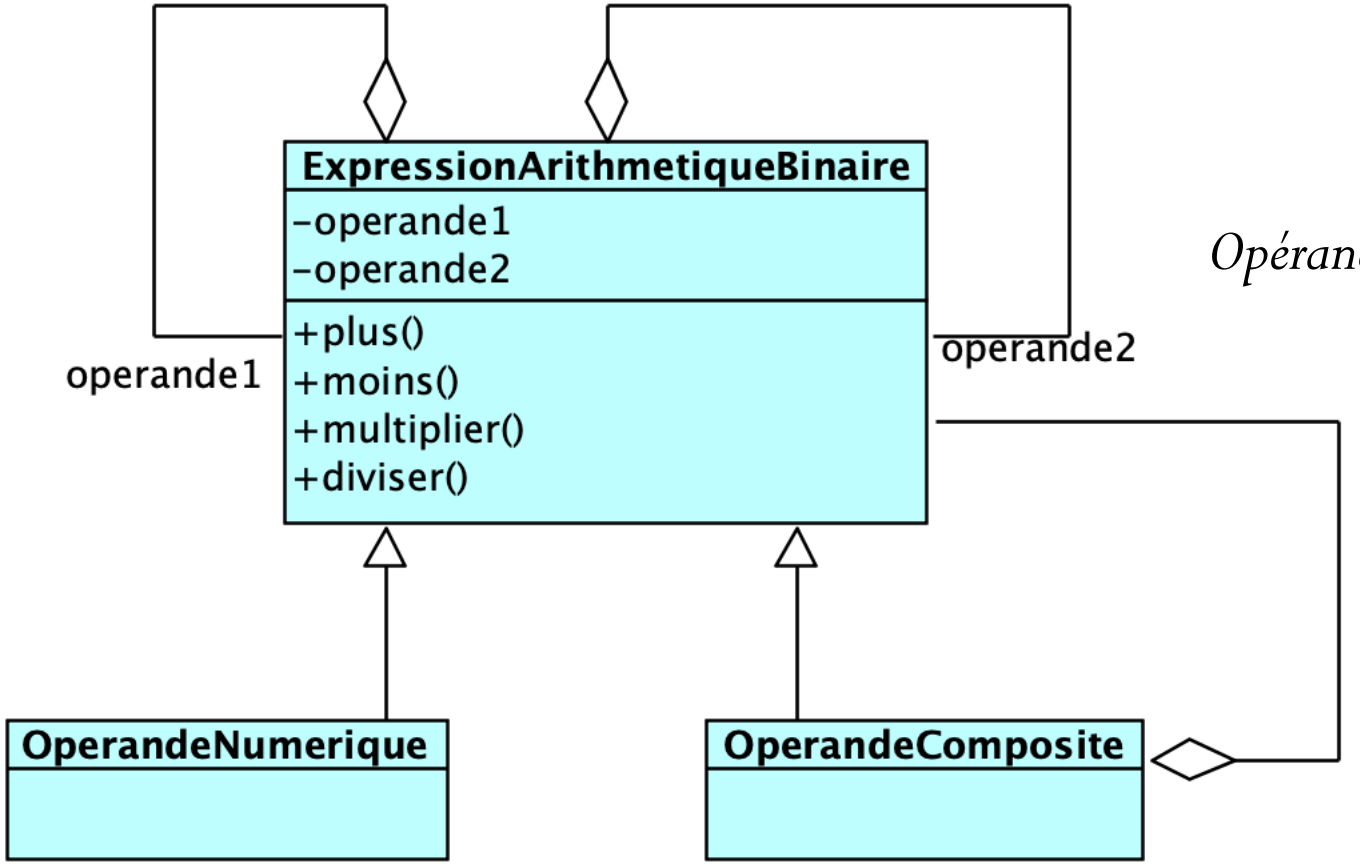


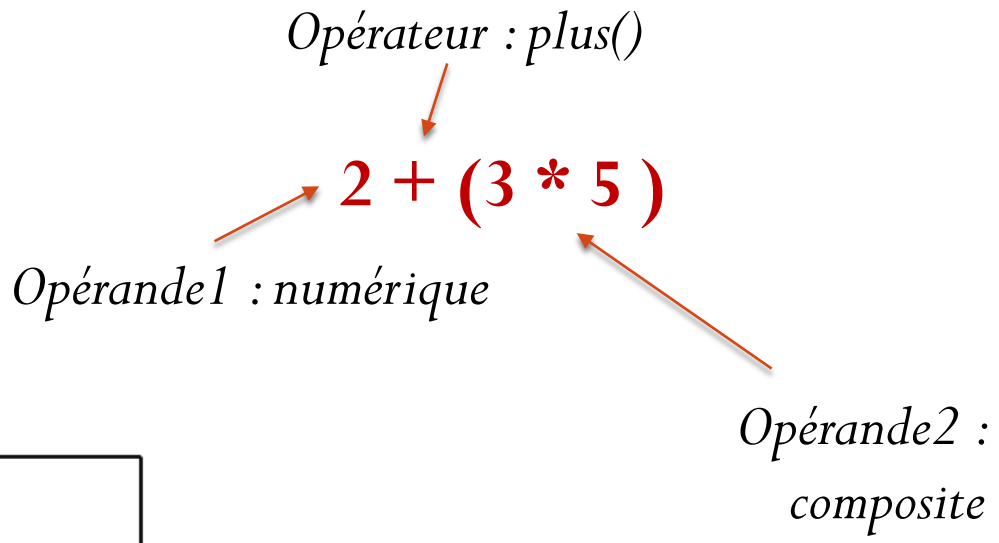
Composite : exemples

- Les **poupées russes**
  - Structure composée uniquement de poupées « composées » sauf la dernière

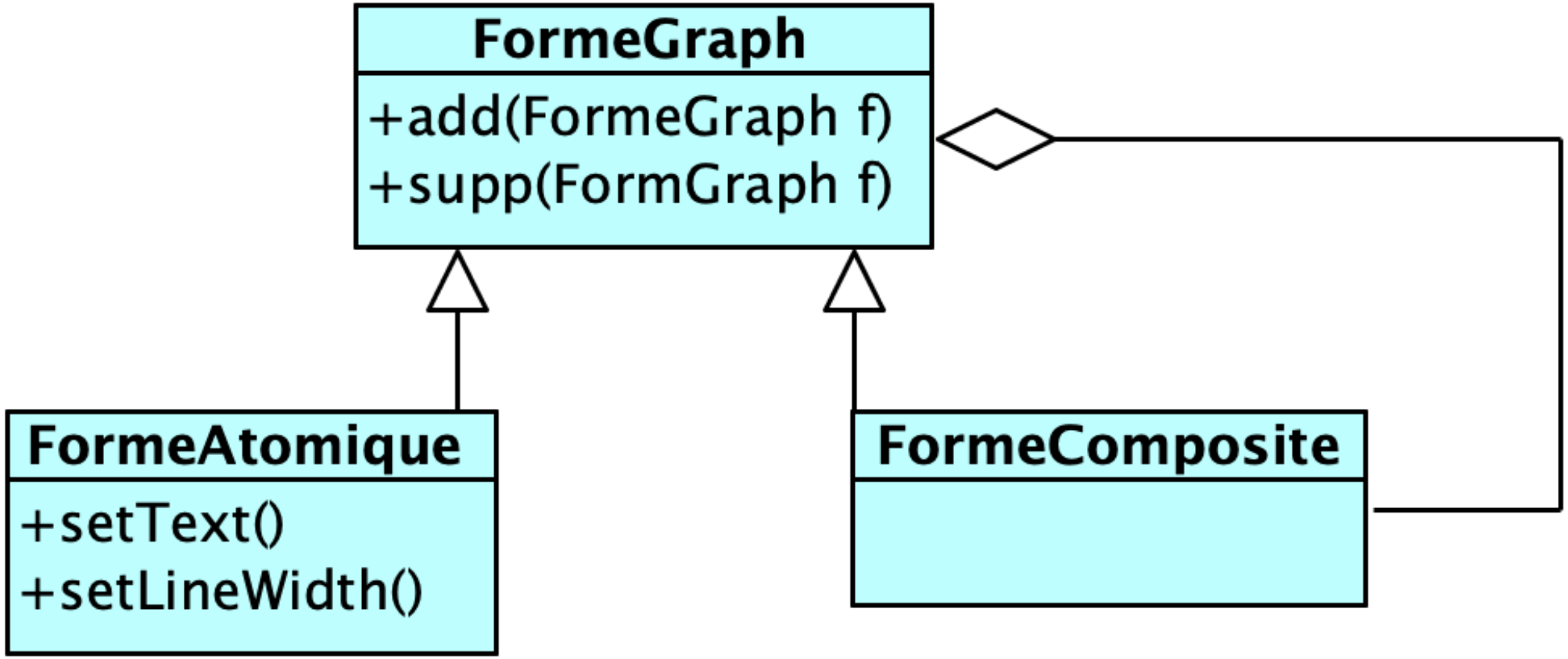
- Toute **structure de données récursive**
  - Des containers graphiques
  - Une structure de document (chapitre/section/paragraphe…)
  - Structure du cloud : un réseau composé de serveurs et d'autres réseaux

Exemple : expression arithmétique binaire





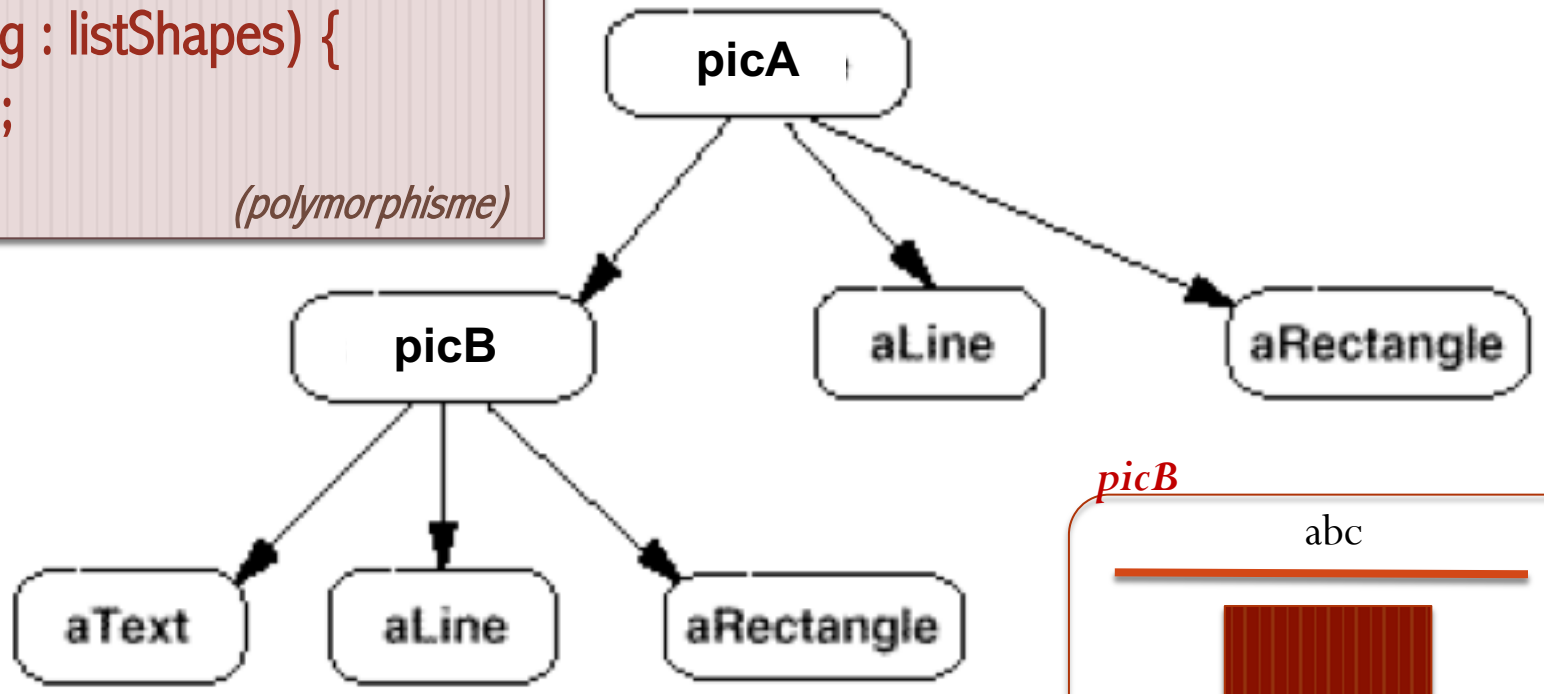
Autre exemples : arborescence de formes graphiques



Pour dessiner :

```
for (Shape g : listShapes) {
    g.draw();
}
```

(polymorphisme)

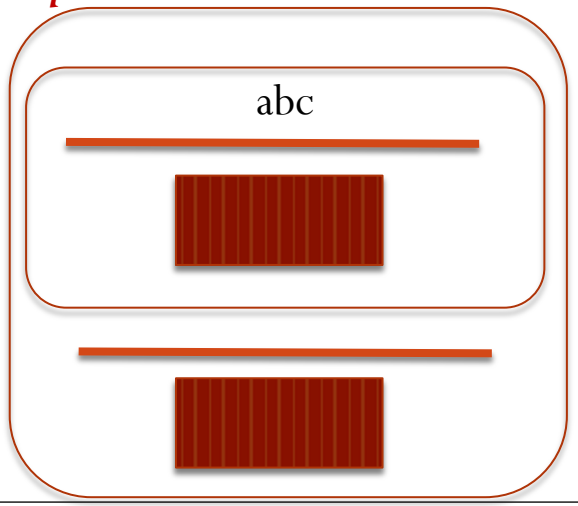



```
Shape aRect = new Rect();
Shape aLine = new Line();
aLine.setLineWidth(5px);
Shape aText = new Text();
aText.setText("abc");
```

picB


```
picB = new Shape();
picB.add(aText);
picB.add(aLine);
picB.add(aRect);
```

picA




```
picA = new Shape();
picA.add(picB);
picA.add(aLine);
picA.add(aRect);
```

Pour aller plus loin...

- Le code d'écriture de l'arborescence avec le pattern Composite n'est pas toujours très simple…

- On peut utiliser le **Builder** pour simplifier :
- Le patron **Builder** sépare la construction d'un objet complexe de sa représentation de sorte que le même processus de construction peut créer des représentations différentes
  - Cf https://sourcemaking.com/design_patterns/builder
- Souvent, les conceptions démarrent par le **FactoryMethod** (moins compliqué, plus personnalisable, avec beaucoup de sous-classes) et évoluent vers **l'AbstractFactory**, **Prototype** ou **Builder** (qui sont plus flexibles, plus complexes) au fur et à mesure du besoin croissant en terme de flexibilité de code.

Le pattern TemplateMethod

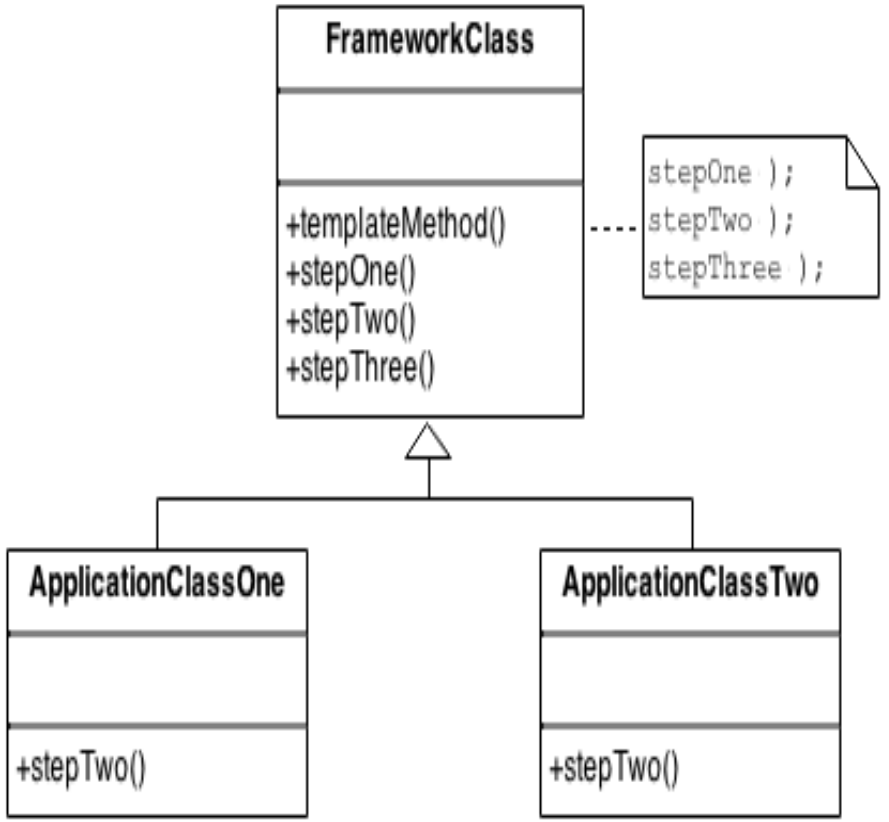
Pattern comportemental à portée de Classe

Fonctionnement et structure

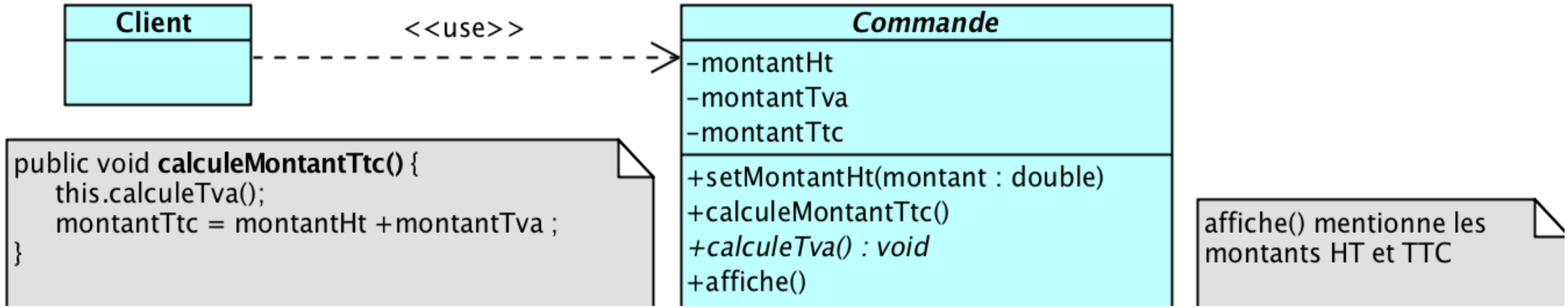
- Une classe de base définit un algorithme (appelé *template method*) qui est composé de différentes étapes (= une succession de méthodes), certaines communes, d'autres spécifiques.
  - Des sous-classes définissent les méthodes spécifiques.

- Le client appelle la bonne version de la méthode ou peut recréer une sous-classe avec une nouvelle version (OCP)

→ Les Frameworks utilisent beaucoup le DP Template Method.



Exemple : calcul de montants TTC



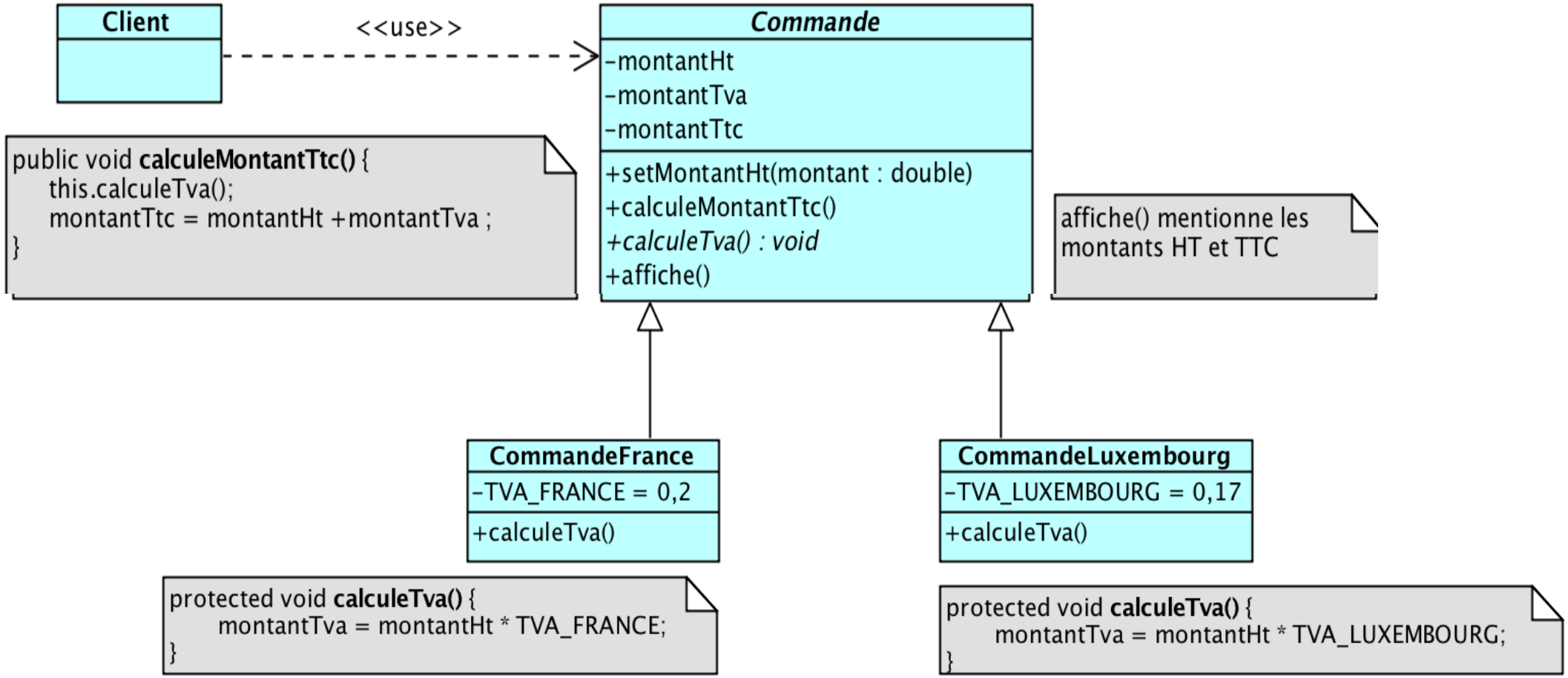
*La **templateMethod** ici est :*

- *setMontantHt()*
- *calculeMontantTtc()*
- *affiche()*

*Ici **this** fera appelle à la « bonne » implémentation (celle de la sous-classe instanciée)*

*L'**implémentation** des méthodes **communes** à toutes les sous-classes sont écrites dans la classe de **base***

Calcul de montants TTC (suite)



*Les méthodes **spécifiques aux sous-classes** sont abstraites ici et implémentées dans les sous-classes*

TemplateMethod vs. d'autres patterns

- **Strategy** ressemble à **Template Method**, mais :
  - **Template Method** utilise ***l'héritage*** pour la part variable de l'algorithme alors que **Strategy** utilise la ***délégation*** pour des versions entières d'algorithmes ;
  - **Strategy** modifie la logique des objets individuels. **Template Method** modifie la logique de **toute une classe**.

- **Factory Method** est une spécialisation de **Template Method**.

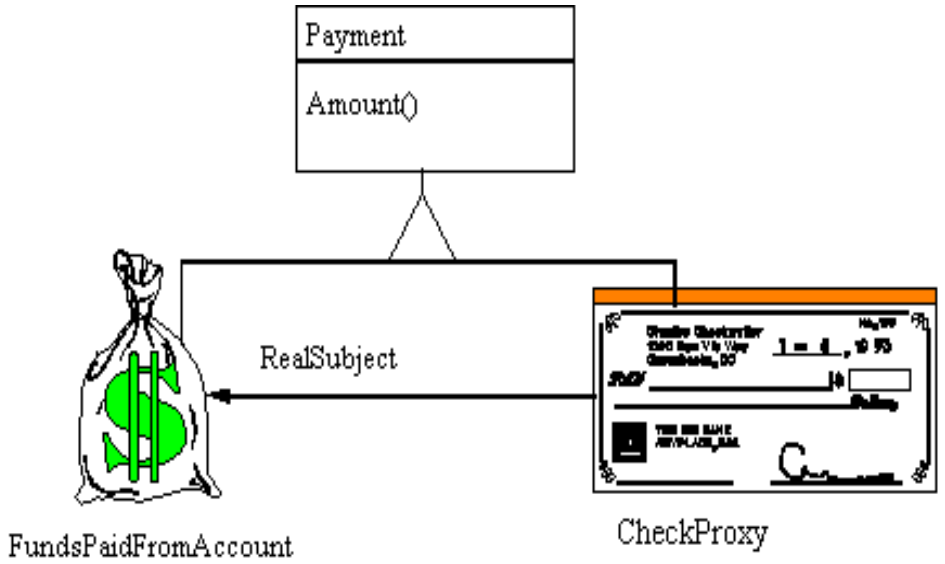
Le pattern Proxy

Pattern de comportement

Source : Non-Software Examples of Software Design Patterns, Michael Duell

Description

- Le proxy fournit une façon détournée d'accéder à un objet donné.
- Par exemple, un **chèque** est un moyen indirect de paiement. Le chèque peut être utilisé à la place d'espèces pour effectuer des achats : c'est un proxy de l'argent du compte.



Un proxy peut fournir un objet de substitution **pendant le chargement** d'un objet plus complexe.

- Durant le chargement le Proxy recourt à ses propres méthodes ;
- Une fois le chargement terminé, les méthodes du Proxy délèguent leurs actions aux méthodes de l'objet.

- Exemple d'utilisation : chargement d'un objet reporté au moment réellement nécessaire, préaffichage d'une image alternative pendant son téléchargement.

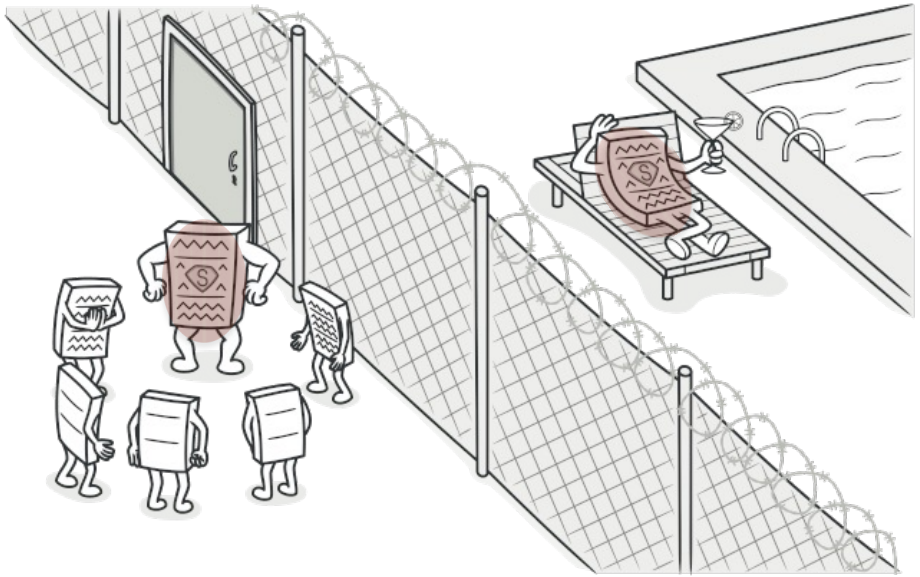
Cas d'utilisation

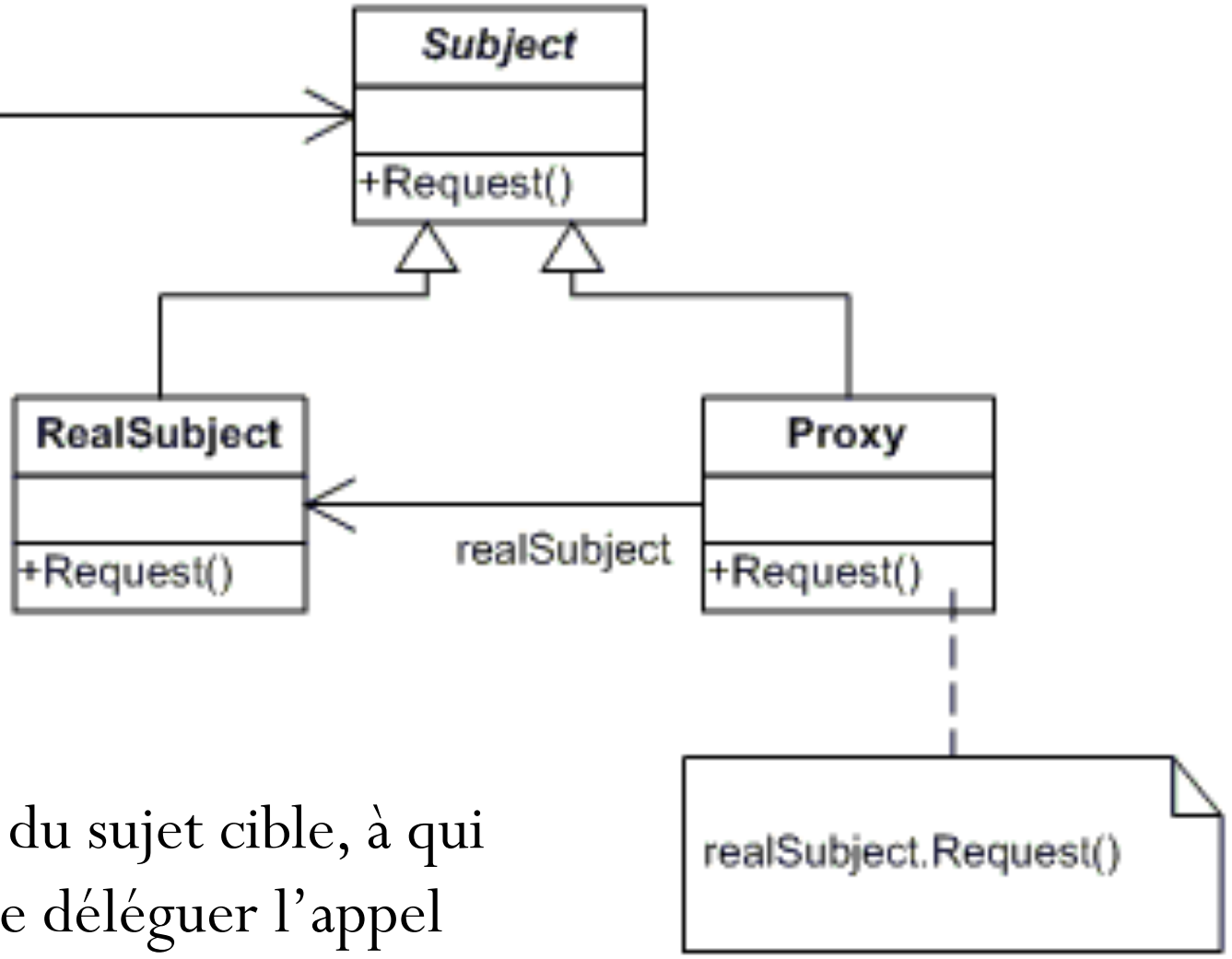
- En présence d'objets gourmands en ressources : on ne souhaite pas instancier ces objets avant qu'ils ne soient effectivement demandés par le client.

Intention

- Fournir un substitut à un autre objet pour en contrôler l'accès.
- Ajouter une enveloppe et une délégation pour protéger l'accès au composant réel qui est d'une complexité excessive.

PROXY



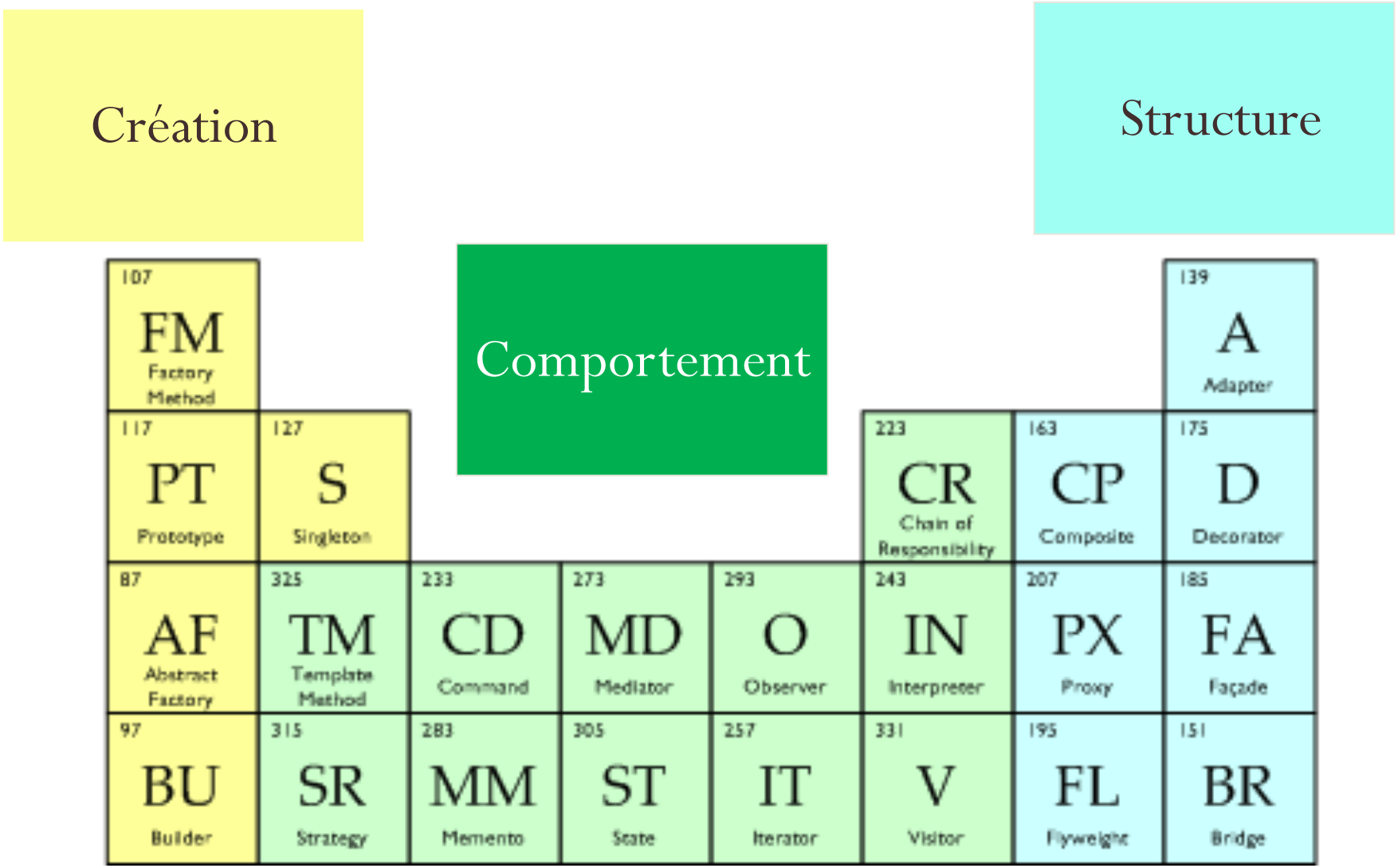


Le PROXY est un intermédiaire, copie du sujet cible, à qui s'adresse une classe Client. Il ne fait que déléguer l'appel au service requis du vrai sujet.

Conclusion

Sur les Design Patterns

Classification des patterns



Conclusion sur les DP

- Les Design Patterns fournissent un **outil puissant :**
  - **d'abstraction** des problèmes rendant le code facile à faire évoluer (ce qui augmente significativement la durée de vie du projet) ;
  - de documentation de **savoir-faire**;
  - de **nommage de concepts** universellement utilisés;
  - de **réutilisation** dans les projets.

- **Points négatifs**
  - *Complexification* du code car ils augmentent le nombre de classes ;
  - L'utilisation du polymorphisme propre aux design patterns *pénalisent les performances* en terme d'exécution, de mémoire et de compilation.

Design pattern : Je retiens....

- Ce que c'est
- Les patterns vus en cours et en TD : Façade, Adapter, State, Strategy, Singleton, Observer et FactoryMethod
- Leurs avantages et leurs inconvénients