IUT Lyon 1
l'excellence technologique

Chap.5 – Design patterns (part. 2)

V. Deslandres ©

Conception d'Architectures Logicielles

LP DevOps

IUT de Lyon - Université Lyon 1

Sommaire du cours DP - 2

- Notion d'interface #3
- Le patron Proxy ------------------- #4
- Les patterns FactoryMethod #9
- Le pattern Adapter #19
- Le pattern TemplateMethod #30
- Conclusion sur les DP #35

Build the thing right

Préambule : notion *d’interface*

- « **Interface** » : sens général
  - Pas seulement Java
  - Ensemble des méthodes associées à un composant (classe, package, module), par ex.: interface d’une API

Le pattern Proxy

Pattern comportemental à portée Objets

Design Pattern Proxy

Problème :

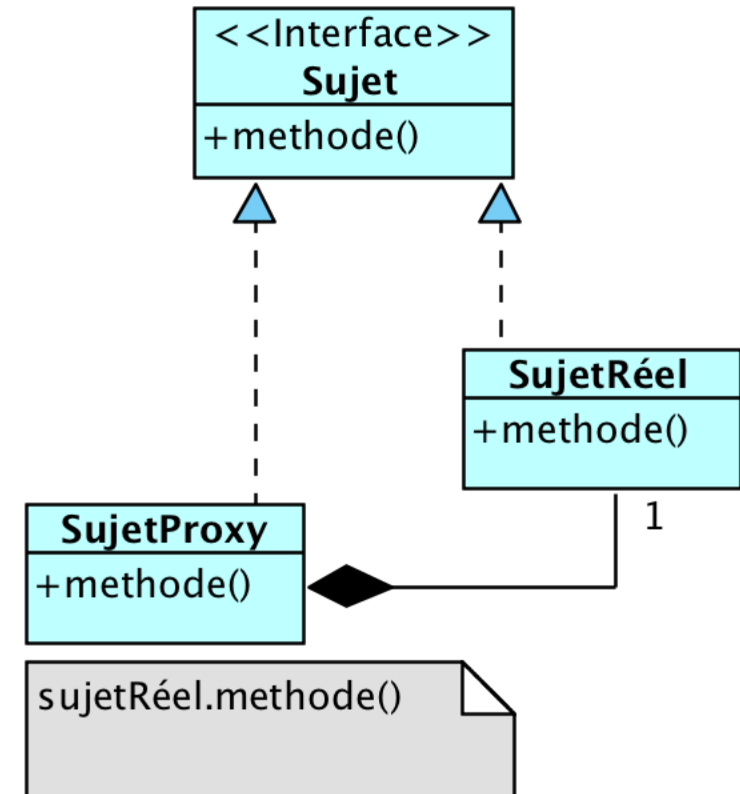
- On a besoin de références à un objet, qui soient plus **créatives** et plus **sophistiquées** qu'un simple pointeur.

Solution :

- **Proxy** fournit à un tier, un *mandataire* pour contrôler l'accès à cet objet, ce dernier étant **encapsulé** dans le proxy.

NOTA

- Proxy fournit **la même interface** que le sujet, mais peut y ajouter des fonctionnalités.
- **Pas d'accès direct** au sujet



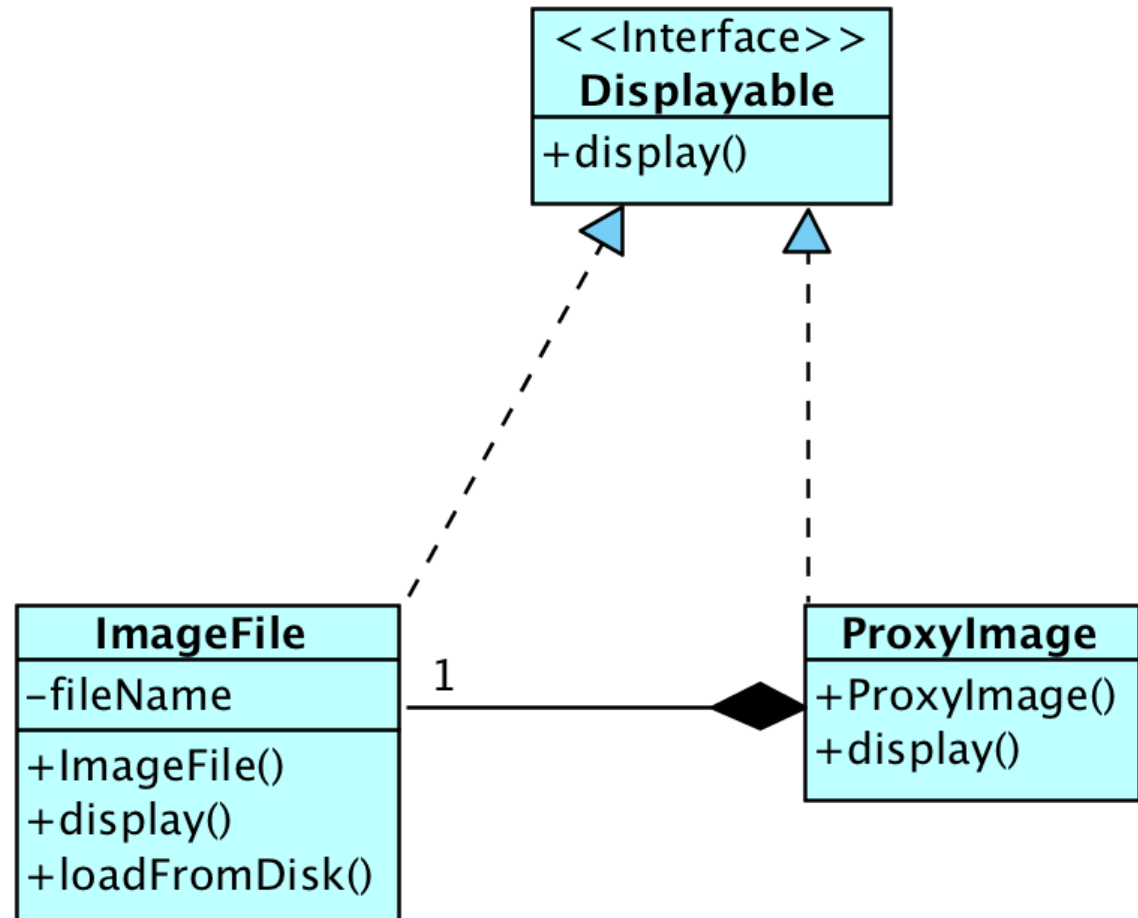
Proxy : utilisations

- Un *proxy à distance* fournit un représentant local d'un objet situé dans un espace adresse différent.
- Un *proxy virtuel* crée des objets lourds à la demande.
- Un *proxy de protection* contrôle l'accès à l'objet original. C'est utile quand les objets ont différents droits d'accès.
- Un *proxy intelligent* est le remplaçant d'un pointeur brut, qui réalise des opérations supplémentaires, lors de l'accès à l'objet. Par exemple :
  - Décompte du nombre des références faites à un objet réel, de sorte que celui-ci puisse être libéré automatiquement, dés qu'il n'y a plus de références ;
  - Charger en mémoire un objet persistant quand il est référencé pour la première fois ;
  - Vérifier, avant d'y accéder, que l'objet réel est verrouillé, pour être sûr qu'aucun autre objet ne pourra le changer.

Proxy : illustration

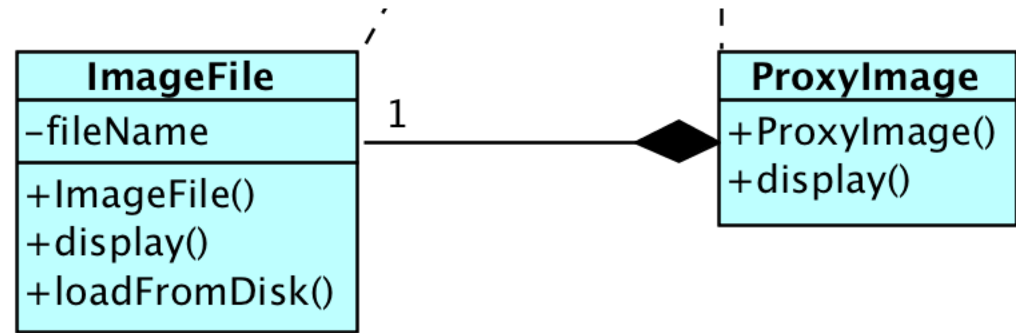
```
public interface Displayable {
   void display();
}
```

```
public class ImageFile implements Displayable {

   private String fileName;

   public ImageFile(String fileName) {
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName) {
      System.out.println("Loading " + fileName);
   }
}
```



```java
public class ProxyImage implements Displayable {

   private ImageFile realImage; // sujet réel encapsulé
   private String fileName;

   public ProxyImage(String fileName) {
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if (realImage == null) {
         realImage = new ImageFile(fileName);
      }
      realImage.display();
   }
}
```

On contrôle dans display() si le sujet a été déjà chargé. Si oui on l'affiche. Sinon, on le charge d'abord, et on l'affiche.




```java
public class ProxyPatternMain {

   public static void main(String[] args) {

      Displayable image = new ProxyImage("image_10mb.jpg");

      // image chargée à partir du disque:
      image.display();

      // image non (re)chargée :
      image.display();
   }
}
```

FactoryMethod

Un pattern de **création** ciblé sur les **objets**

Simple Factory (pas un vrai pattern)

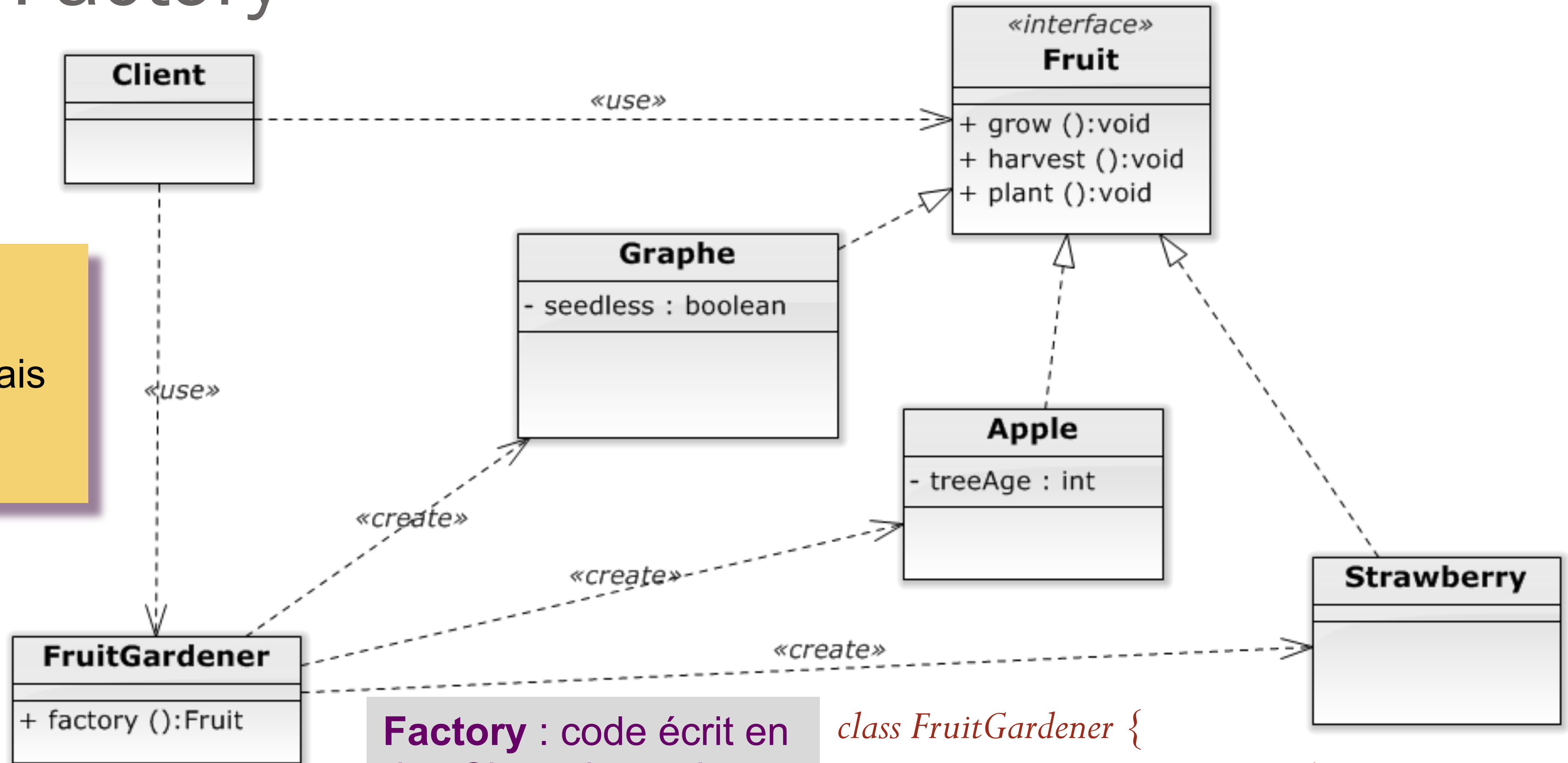
Un idiome de programmation

- Objectif : **créer un objet dont le type dépend du contexte**
- Contexte :
  - Une classe Client a besoin de créer des objets d'une famille et d'utiliser leurs services, sans savoir nécessairement quel objet précisément instancier, cette connaissance dépendant d'une autre classe.
  - Ex.: une classe Client veut utiliser un service de paiement en ligne, et on va choisir celui avec lequel on a le meilleur contrat publicitaire (le « meilleur » change fréquemment, les classes qui utilisent le paiement en ligne ne savent pas).

- Principe : passer par une classe spéciale chargée de CRÉER les objets spécifié par le Client (via un paramètre); **pas de new X().**
  - L'objet retourné est donc toujours **du type de la classe mère**
  - Grâce au **polymorphisme** les traitements exécutés sont ceux de l'instance créée

Une Simple Factory

Un client demande des fruits au jardinier… il ne connait pas les noms mais décrit son souhait : croquant, juteux, etc.

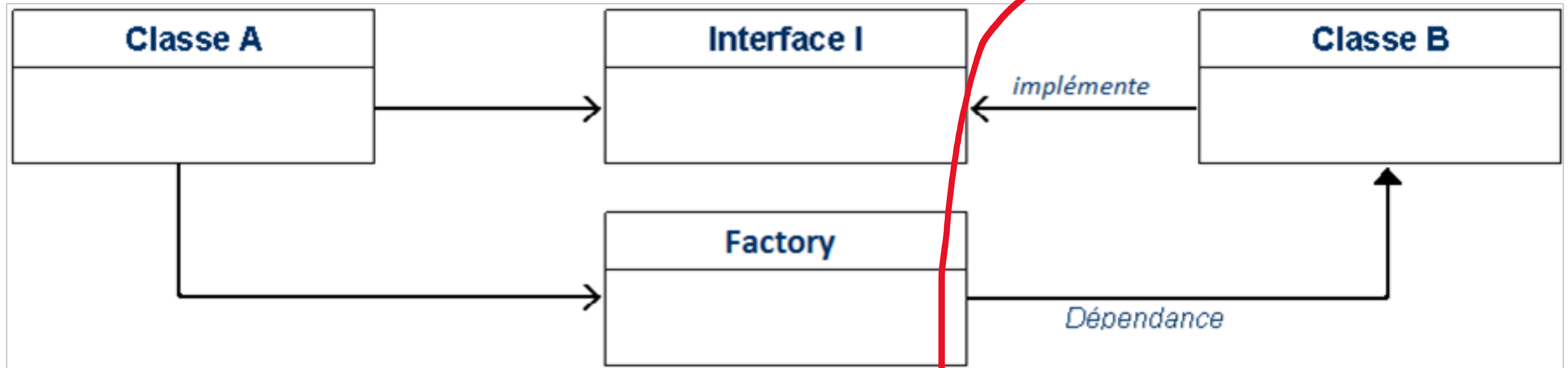


Factory : code écrit en dur. Si on ajoute des fruits, on doit modifier son code et faire l'appel au nouveau constructeur

```
class FruitGardener {
  public Fruit factory(String f) {
    if (f.equals("croquant")
      return new Apple()
    else if (f.equals("juteux"))
      return new Graphe()
    etc…
```

Implémentation du DIP

- C'est une classe **Factory** qui va gérer les dépendances vers les classes concrètes.
- Cette *factory* possède des méthodes qui vont instancier la dépendance en fonction du contexte (ici **Classe B**) et la retourner.
- Chaque fois qu'une dépendance devra être résolue (besoin d'un objet de type **Interface I**), la classe appelante utilisera la *factory*.



Vision Utilisation d'un service

Vision Création

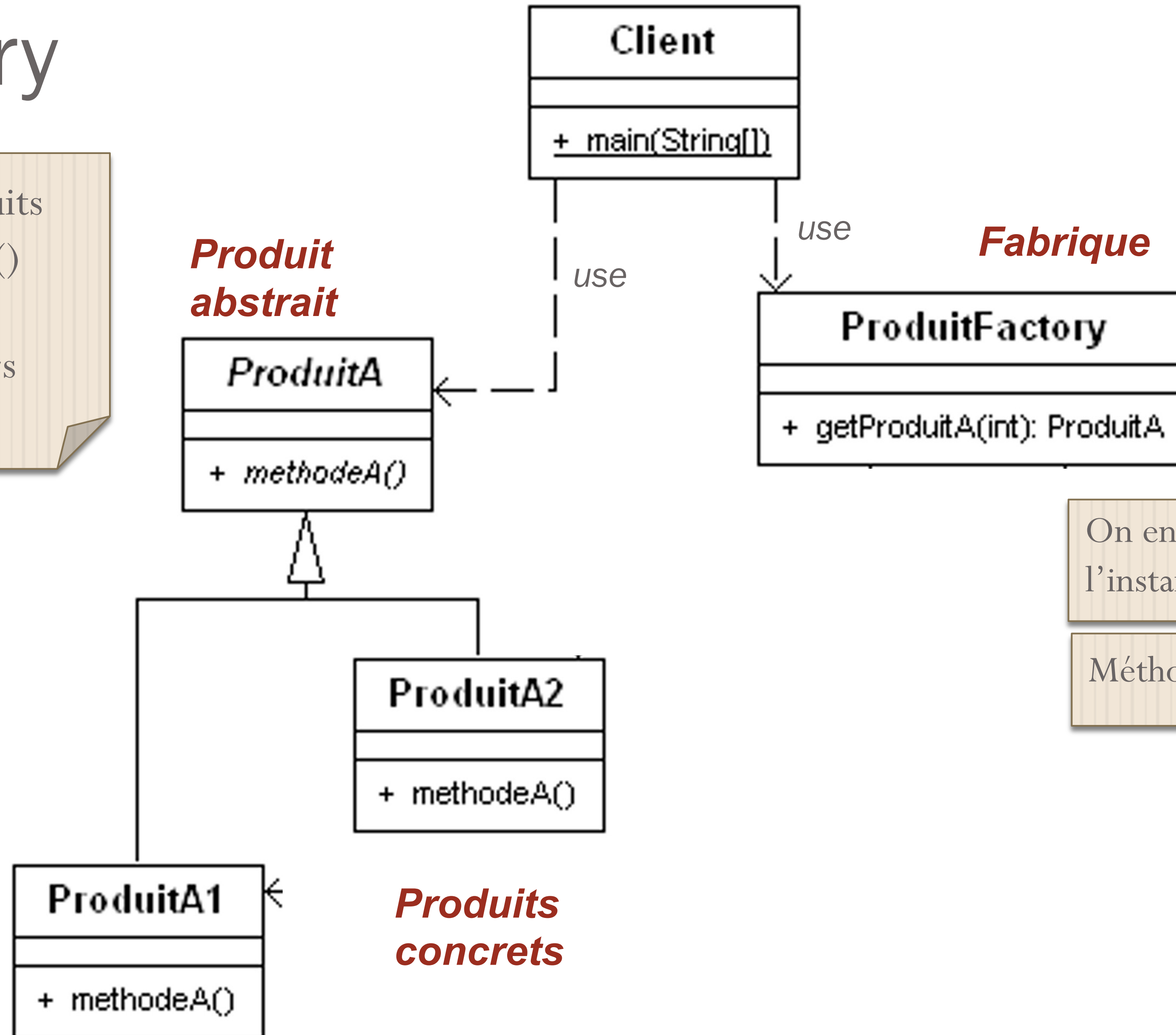
Code associé

```
public class Factory {

   public I getDependency(int x) {
      switch (x) {
            case 1: return new B1();
            case 2: return new B2();
            …
            case 12: return new B12();
            default:  return new B1();
  } }
}
```

```
// Classe Cliente A
public class A {
            int leContexte;
 // Constructeur qui définit le contexte requis par la classe
    A(int c) {
            leContexte = c;
    }

    public static void main(String[] args) {
            …
// Transmet à la factory le contexte pour le bon B :
        I b = new factory().getDependency(leContexte);
        b.someMethod();
    }
}
```

Simple Factory

Client fait appel à des produits présentant une methodeA() (polymorphisme).
Il pourrait y avoir plusieurs méthodes.



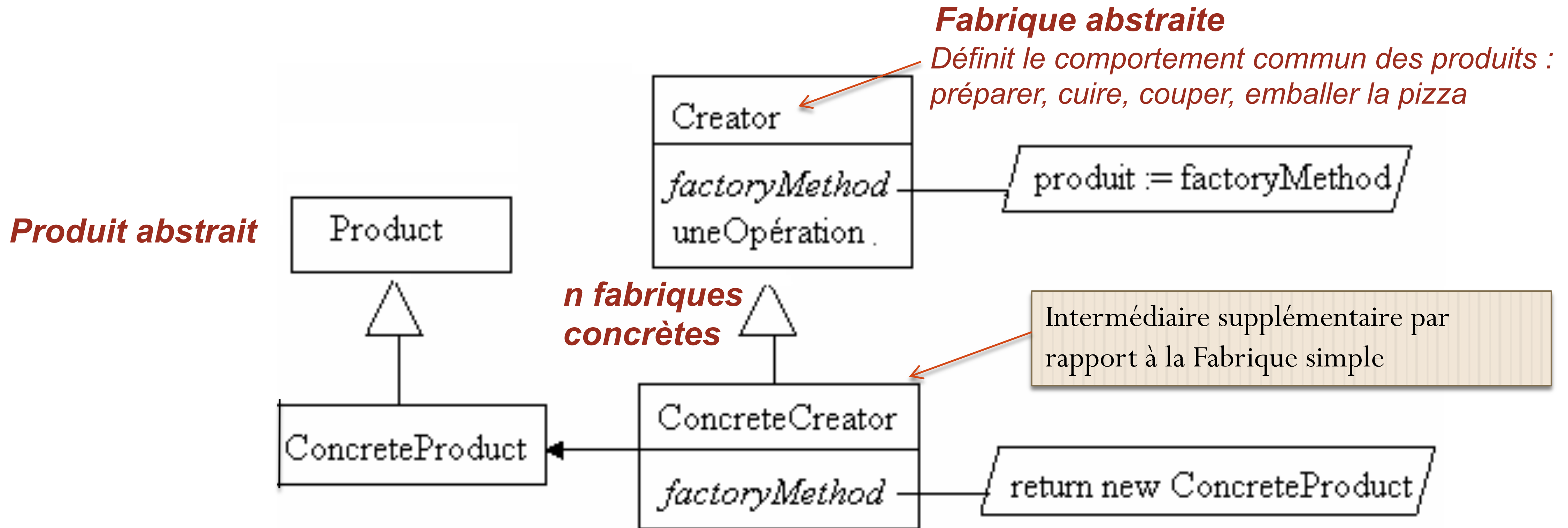
On envoie le paramètre de l'instance à créer

Méthode **statique** ou non

Design pattern « Factory Method »

- Plusieurs factories : ex. contrats publicitaires dans différents pays
- On ajoute une **abstraction** supplémentaire : une classe abstraite (*la fabrique*) délègue l'instanciation des objets (*les produits*) à une *fabrique concrète*, et il peut y en avoir *n*.
  - *Ex. créer des pizzas, pizzas de Brest ou de Marseille, de Brest végétarienne ou aux lardons, de Marseille végétarienne ou aux fruits de mer*
  - On **factorise le mécanisme de création, qui est commun à tous les produits** : une pizza se prépare, se cuit, se coupe et s'emballe, pour toutes les pizzas ; la pizza créée sera spécifique au besoin de création.
- Retourne une instance du produit spécifique créé (produit adapté)
  - Faciliter la création en respectant le principe d'O/F de code

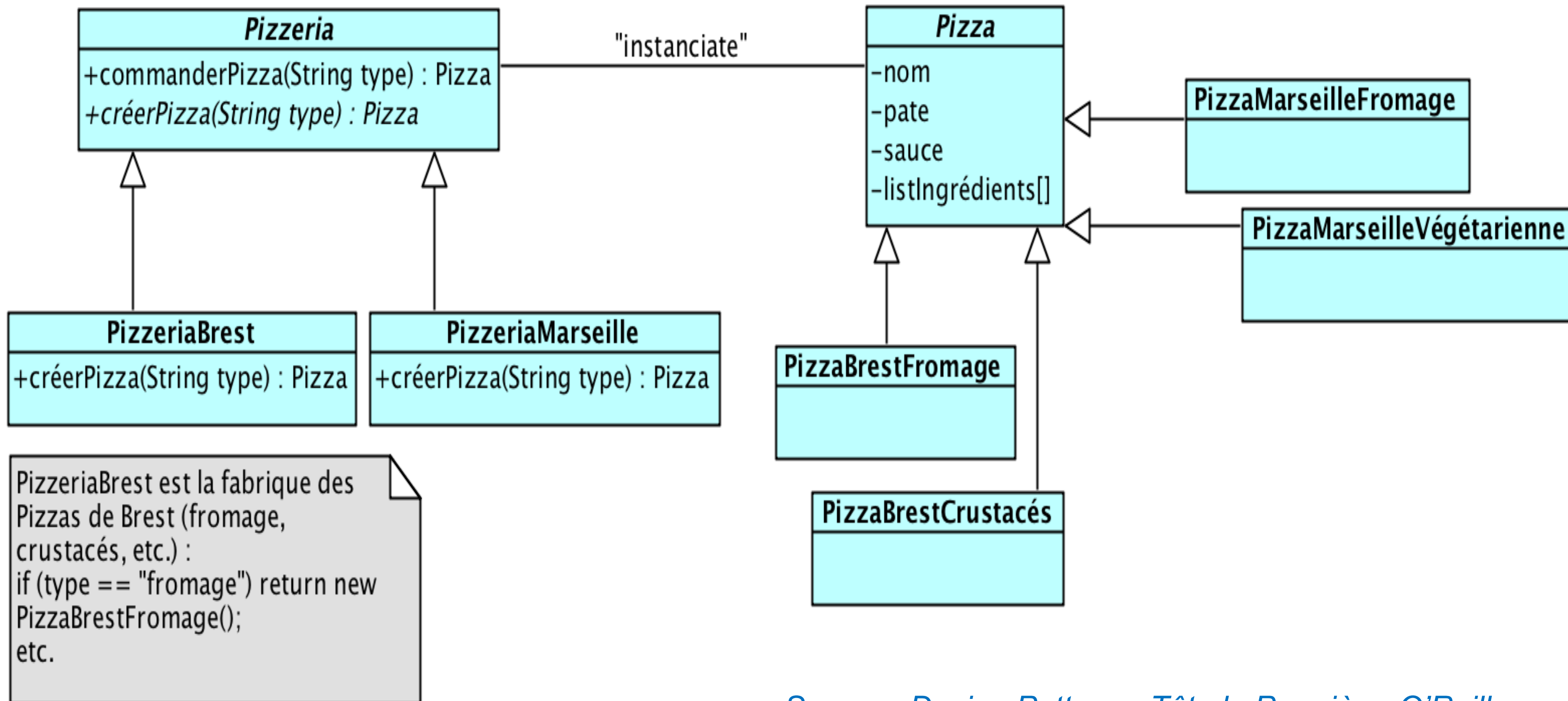
Principe de Factory Method



Définir une interface de création, et laisser les sous-classes décider du type

Illustration Factory Method

La méthode *créerPizza()* est **déléguée** aux fabriques concrètes.
commanderPizza() est la méthode définissant le comportement générique des pizzas : elle crée une pizza en appelant *créerPizza(type)* et fait appel à : *pizza.préparer(), pizza.cuire(), pizza.couper(),* etc.



Source : Design Patterns, Tête la Première, O'Reilly

Abstract Factory

- Il existe une 3ème forme de design pattern FactoryMethod
  - Avec un **niveau d'abstraction supplémentaire**
- *Abstract Factory* définit une fabrique abstraite pour *chaque étape du processus commun* de création d'une instance :
  - Ex.: *FabriqueIngrédientPate, FabriqueIngrédientSauce*, cuisson etc., avec *n* implémentations pour chacune : *PateFine, PateFeuilletée, SauceTomate, SauceViande,* etc.
  - C'est la fabrique concrète qui choisit l'implémentation qui convient à chaque étape :
    - PizzeriaBrest, pour sa pizza Fromage, va utiliser la Fabrique concrète PateFine, SauceTomate etc. (dans son constructeur)
- http://www.tutorialspoint.com/design_pattern/abstract_factory_pattern.htm

ADAPTER

L’adaptateur, un autre patron de structure

Design pattern ADAPTER

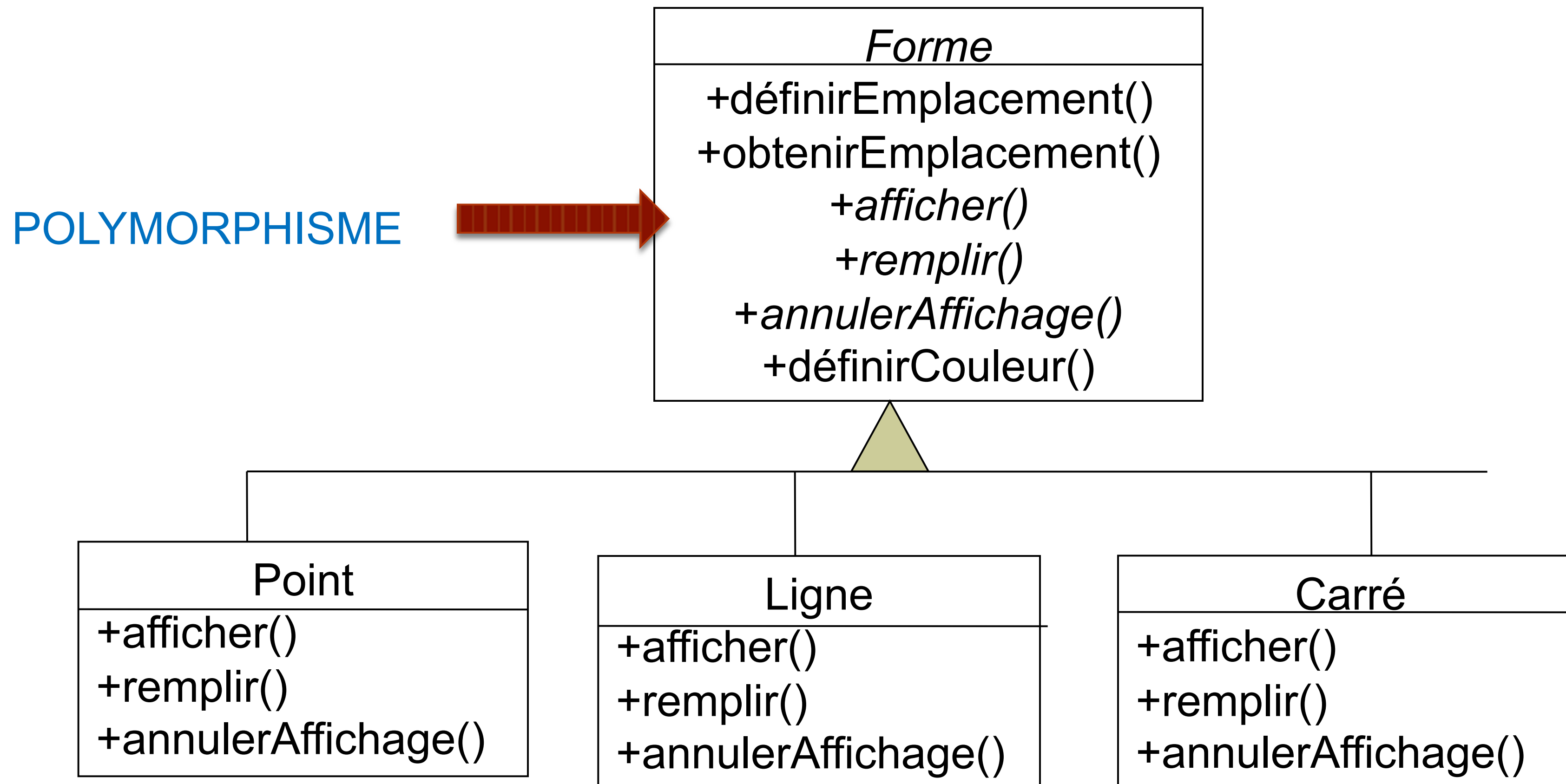
- Il consiste à transformer
  - par **délégation**
- les points d'entrée d'un composant
  - que l'on désire intégrer
  - à l'interface souhaitée par le concepteur

ADAPTER

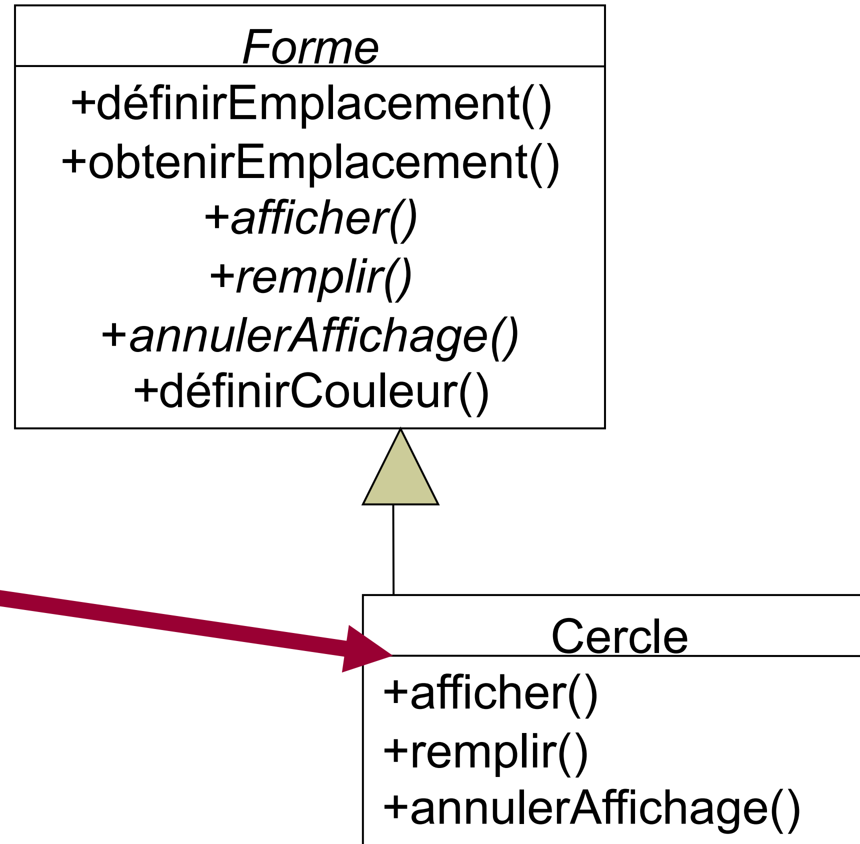
- Objectif : transforme l'interface d'une classe en **une autre interface souhaitée**
  - conforme à ce qu'attendent les classes clientes
- Permet à des classes de collaborer
  - qui n'auraient pu le faire du fait d'interfaces incompatibles
- Ex.: on dispose de classes **Point, Ligne, Carré**
  - ayant des méthodes **Afficher(), Remplir()**
- Les classes clientes appellent ces formes pour les afficher et les remplir

Pattern ADAPTER : illustration

- On crée une classe abstraite Forme :



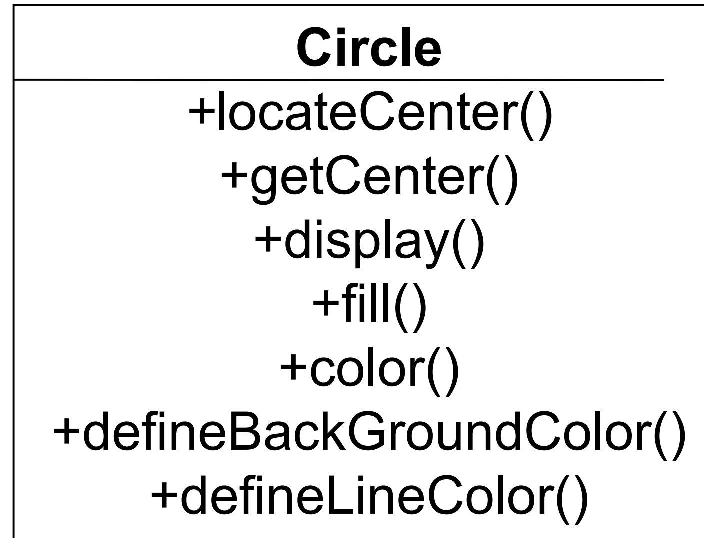
- Imaginons qu'on ait besoin d'une autre forme : le cercle



On pourrait créer une classe Cercle dérivée de Forme

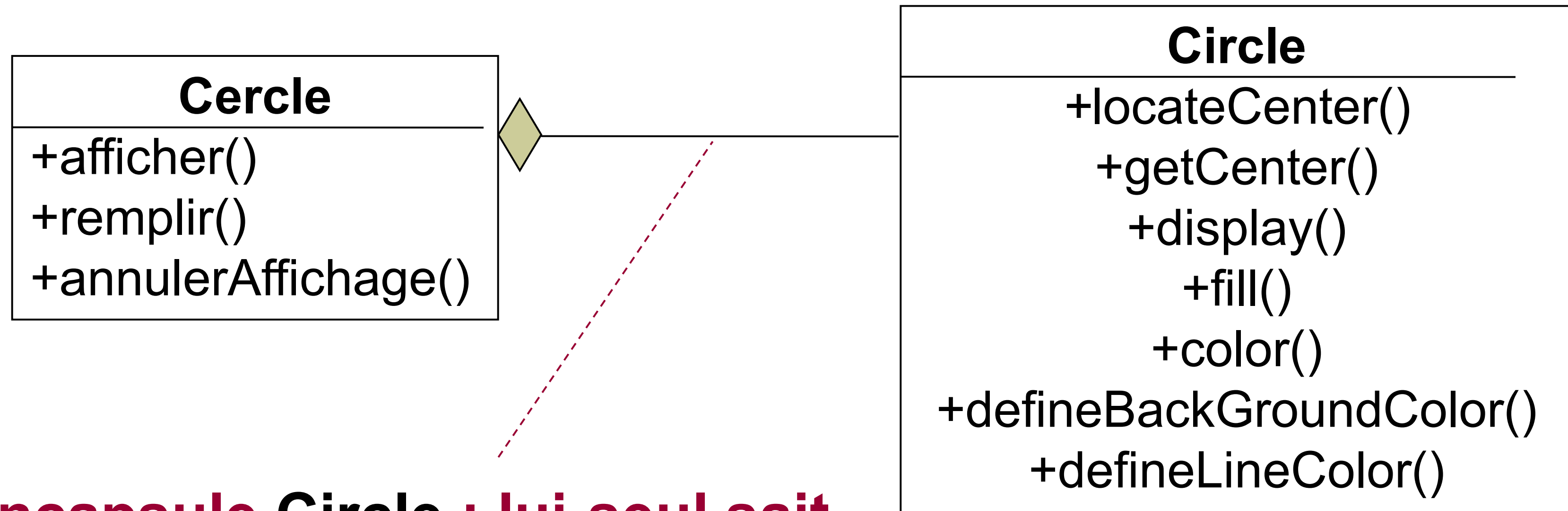
- Supposons que l'on ait déjà une classe **Circle** existante

On pourrait modifier les opérations de la classe Circle pour les adapter aux méthodes de Forme



Pas OCP : risque de bug !

- On va réaliser un Adaptateur : Cercle qui va contenir (**encapsuler**) l'objet **Circle** existant
- Tout ce que fait l'objet Cercle est transmis à l'objet **Circle** en faisant appel à ses opérations



Cercle encapsule Circle : lui-seul sait qu'un objet Circle existe

ADAPTER

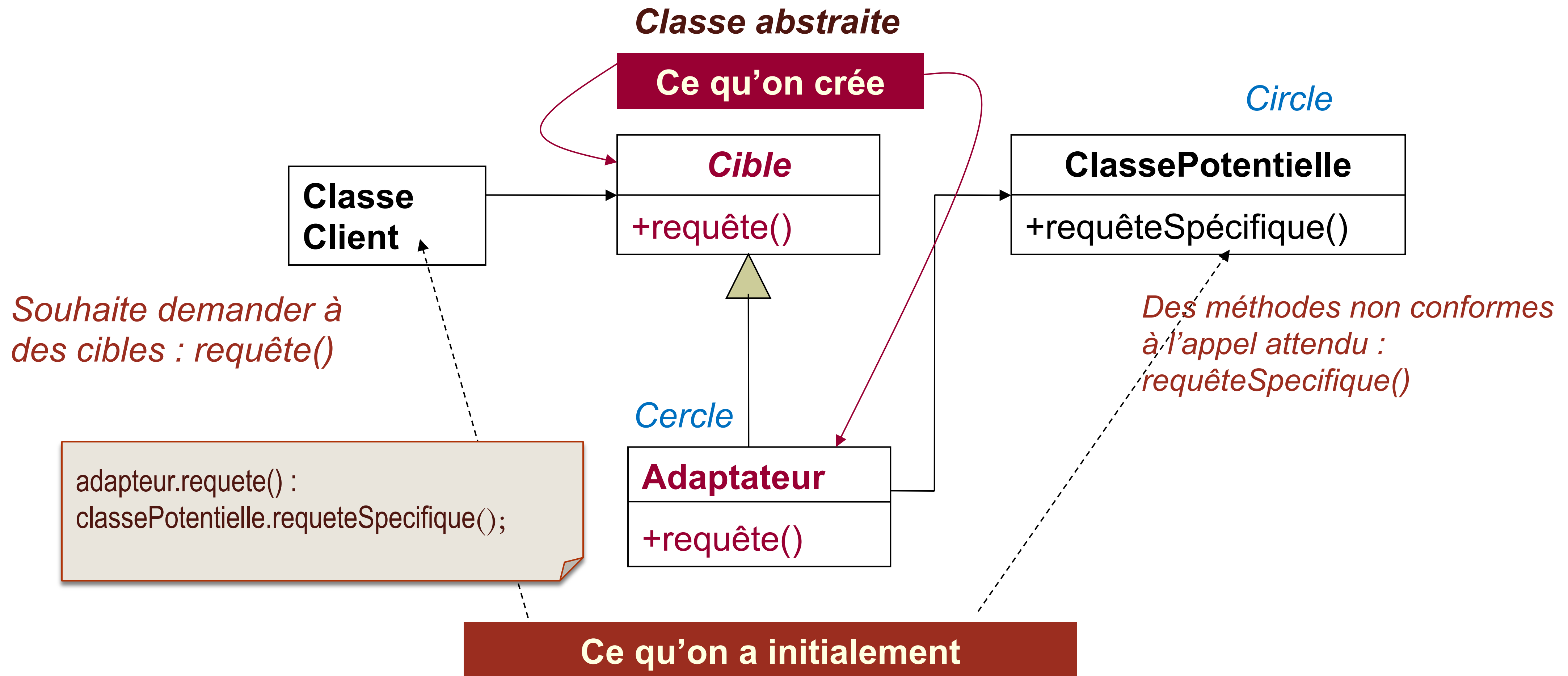
Extrait du code Java correspondant :

```
Class Cercle extends Forme {
    …
    private Circle leCercle;
    …
    public Cercle() {
      leCercle = new Circle();
    }
    void public afficher() {
      leCercle.display();
    }
    void public remplir() {
      leCercle.fill();
     }

}
```

ADAPTER, selon le GoF(9)

- **Objectif** : faire correspondre à une interface donnée un objet existant
- **Problème** : un système donné a les bons objets et les bonnes méthodes, mais pas la bonne interface
- **Implémentation** : intégrer la classe existante dans une autre classe. La classe qui encapsule est compatible avec l'interface voulue et appelle les méthodes de la classe encapsulée

Design Pattern ADAPTER : les classes



ADAPTER vs. Façade (1)

- Ils impliquent tous deux des classes existantes qui n'ont pas l'interface voulue
- Le pattern *Façade* **s**implifie l'interface alors que *l'Adaptateur* encapsule un objet pour coller avec l'interface **existante**

- **Façade** :
  - Encapsulation *des* classes pour faciliter leur utilisation
- **Adaptateur** :
  - Réutilisation, encapsulation *d'une* classe pour un besoin structurel

Vous avez dit « encapsuler » ?

- NOTA : on peut **encapsuler**
- des **attributs**
  - celles de Point, Ligne… sont masquées,
- des **méthodes**
  - ex. définirEmplacement() de Cercle,
- des **classes**
  - Point, Ligne… sont masquées au client par Forme
- des **objets**
  - seul Cercle sait que Circle existe

Le pattern TemplateMethod

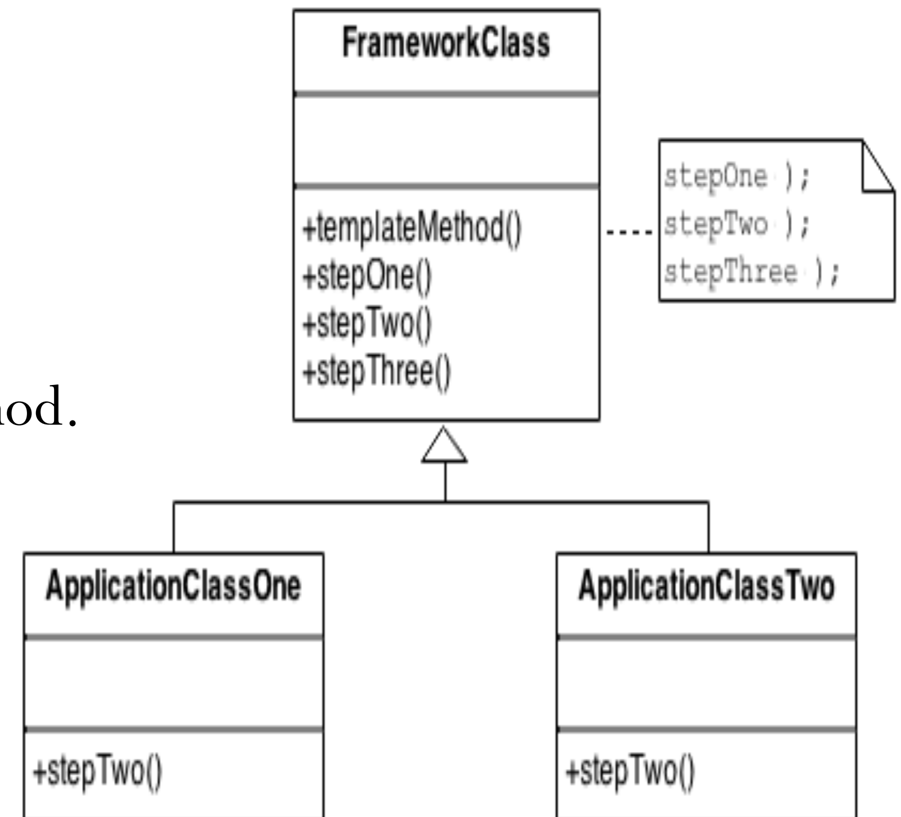
Pattern comportemental à portée de Classe

Fonctionnement et structure

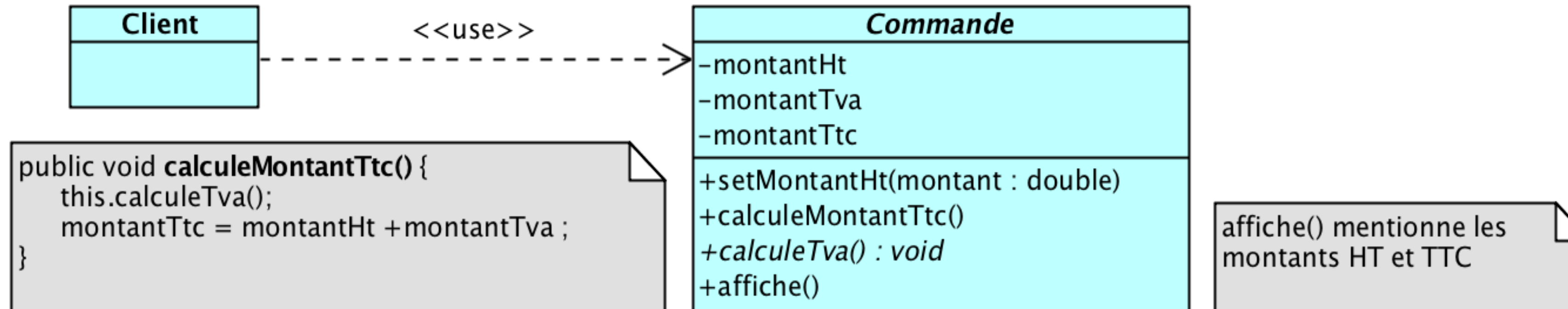
- Une classe de base définit un algorithme (appelé *template method*) qui est composé de différentes étapes (= une succession de méthodes), certaines communes, d'autres spécifiques.
  - Des sous-classes définissent les méthodes spécifiques.

- 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



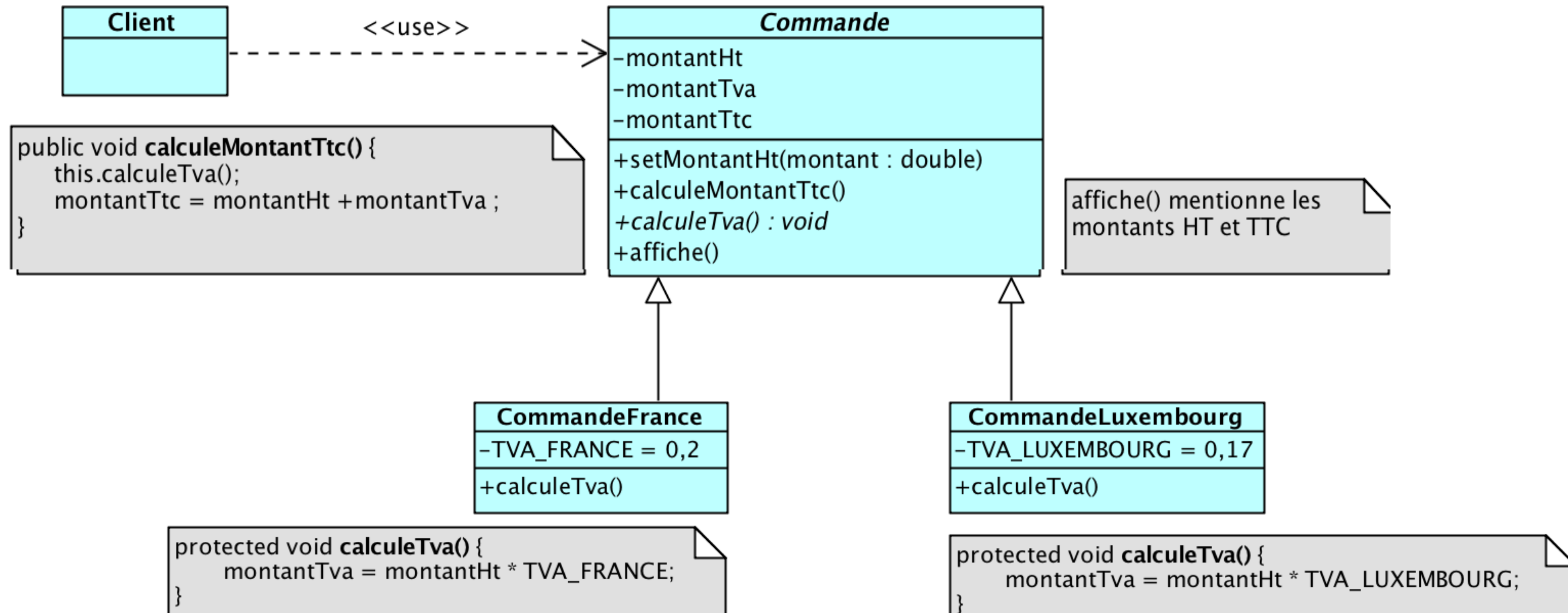
*Ici **this** fera appelle à la « bonne » implémentation (celle de la sous-classe instanciée)*

*La **templateMethod** ici est :*

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

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

Exemple : calcul de montants TTC



*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**.

Conclusion

sur les Design 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.