

# Chap.3 – Qualité de Conception

## Design patterns (part. 1)

**V. Deslandres** ©

BUT Informatique, ASPE

Réalisation d'Applications

IUT de Lyon - Université Lyon 1

# Sommaire de ce cours

- Introduction ----- #3
- Liste des DP ----- [#14](#)
- Le patron State ----- [#15](#)
- Le patron Façade ----- [#23](#)
- Le patron **Singleton** ----- [#31](#)
- Le patron Strategy ----- [#40](#)
- Le patron Adapter ----- [#44](#)

# Les design patterns : présentation

« Patrons de conception » favorisant la réutilisation



# Design Patterns

- **Design patterns** = Modèles de conception (*patrons de conception* in French) pour la POO
- Répondent à des problèmes **récurrents** de la conception OO
- Améliorer ou modéliser la conception de parties du système en bénéficiant de **l'expérience des concepteurs** chevronnés
  - Catalogue des meilleures pratiques à adopter
- Analogie avec l'algorithmique :
  - L'algorithmique concerne le corps des méthodes (intra classe), alors que les *patrons* concernent l'organisation des classes entre elles (inter classe)

Réf. (the Gang of Four) - E. GAMMA, R. HELM, R. JOHNSON et J. VLISSIDES, Addison-Wesley, « Design Patterns – Catalogue de Modèles de Conception Réutilisables », International Thomson Publishing France, 1996

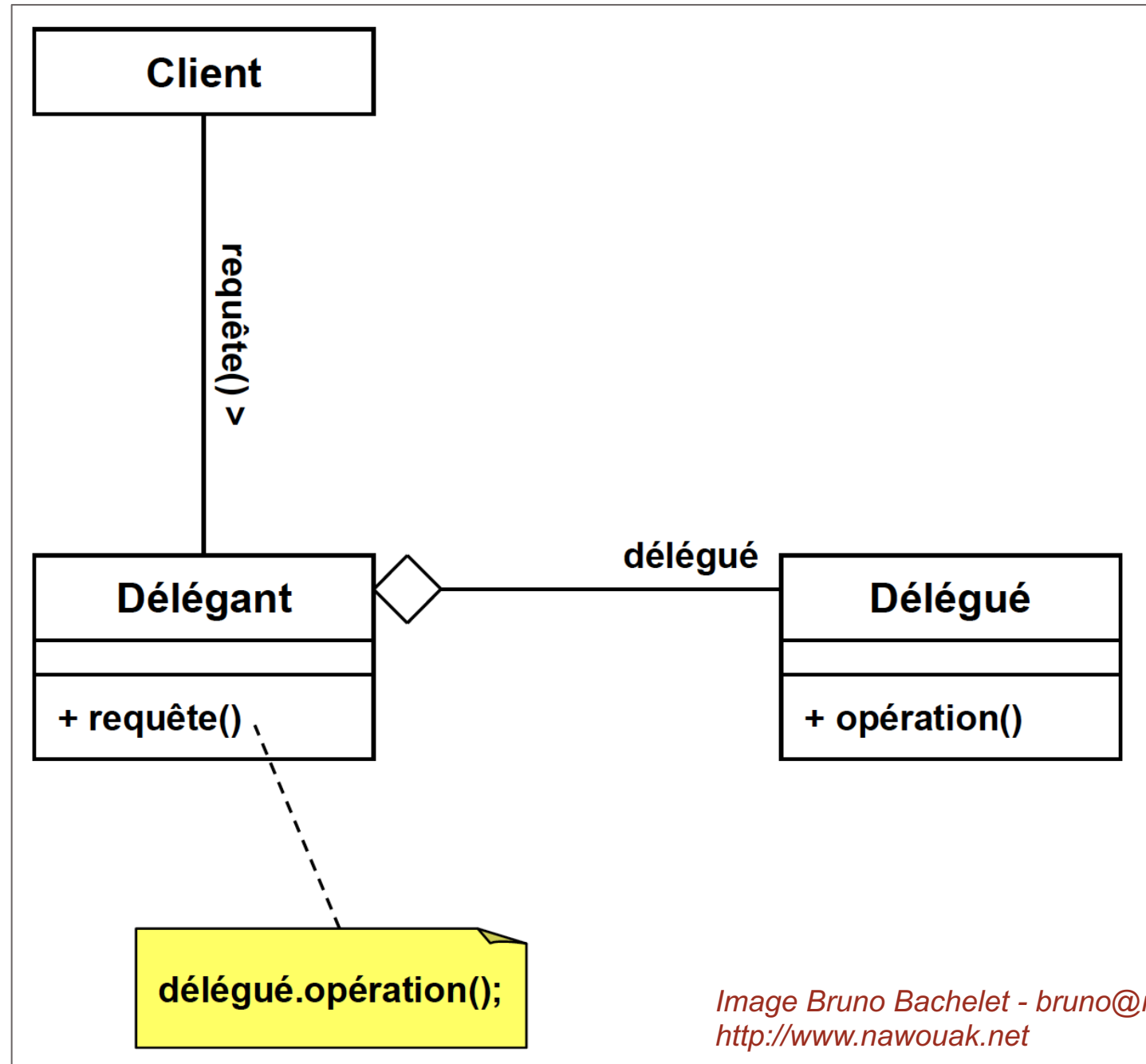
# Problèmes « récurrents » ?

## ■ Ex. en conception orientée objet :

- **Diminution du couplage** entre composants et contexte, pour faciliter l'évolution de code
  - Privilégier les interfaces, encapsuler ce qui varie
- **Séparation des rôles**
  - Permettre à une partie d'un système de varier indépendamment des autres
- **Délégation d'opération** (cf ci-après)
- Indépendance vis-à-vis des plateformes matérielles et logicielles
- **Réutilisation** du code existant
- Facilité **d'extension**
- *Etc.*

*NOTA : les programmeurs s'intéressent à la conception quand ils maîtrisent bien un langage et codent depuis un certain temps.*

# Mécanisme de Délégation



```
public class A {  
    private B refB;    // encapsulation  
                        // un objet B est un attribut de A  
    public A( B unB ) {  
        refB = unB;  
    }  
    void affichage() {  
        refB.afficher(); // délégation de l'opération afficher()  
    }  
}
```

Classe de Test :

```
A unA = new A( new B2() );  
unA.affichage();
```

# Bénéfices des Design Patterns

1. **Capitalisation** de l'expérience et donc **réutilisation** de solutions
  - Plus puissant que la réutilisation de codes
  - Amène souvent la réutilisation de *composants*
2. **Vocabulaire commun** pour la conception
  - chaque NOM de pattern contribuant à ce vocabulaire, ex. « On fait un Singleton ? »
3. Niveau d'abstraction **plus élevé**
  - élaborer des constructions logicielles de meilleure qualité
4. **Souvent** : réduction de la **complexité** du système et **robustesse**
  - Impression de simplicité (ex. les IHM avec OBSERVER)
  - L'API Java en utilise beaucoup dans ses bibliothèques (ex. les flux Java sont des DECORATOR, les menus reposent sur COMMAND)



# Les inconvénients

- Effort de synthèse
  - Difficile à **comprendre** parfois
  - Difficile à **reconnaître**
  - Haut niveau d'abstraction
- Les patrons **se « dissolvent »** dans le code
- Ils sont **nombreux**
  - Lesquels sont identiques ?
  - Ils sont de niveaux différents
    - Certains patterns s'appuient sur d'autres
- Ils nécessitent un **temps d'apprentissage**
  - Pas toujours facile sur du code en production
  - Passer par des exercices : seule la pratique permet de voir les avantages

# Typologie des Design Pattern / Description

## ■ Classification des 23 patrons de Gamma :

- selon la **fonction** d'abord
  - modèle de **création**,
  - modèle de **structure** (assemblage d'objets),
  - modèle **comportemental**
- selon la **portée**
  - **classes** : héritage
  - **objets** : délégation

- Exemples : pattern Abstract Factory = création / classes
  - pattern Composite = structurel / objets

- Nom du design pattern
- **Objectif** = but
- **Problème** = qu'il s'efforce de résoudre
- **Solution** = proposée (contexte donné)
- **Participants** = entités impliquées
- **Conséquences** = ce qui se passera en implémentant le pattern
- **Implémentation** = mise en œuvre concrète (DCL)
- Référence GoF

# Patrons de **Création**

- Objectif : **proposer différentes « formes » de création**



- Abstraire le processus d'instanciation
- Rendre indépendant la façon dont les objets sont créés, composés ou initialisés
- Cacher ce qui est créé, qui crée, où, comment et quand.

# Patrons de **Structure**

- **Expliciter les formes de structure**
  - Comment les objets sont assemblés
  - Comment les patrons sont complémentaires les uns des autres
- En Conception Objet, la structure porte sur :
  - Les **classes**
  - Les **packages**
  - Les **composants** physiques



# Patrons de **Comportement**

- Décrire les formes de comportement :
  - Les algorithmes
  - Les comportements entre objets
  - Les formes de communication entre objet
- Objectif : concevoir des modules à **forte cohésion** et **faible couplage**





## Présentation de quelques Patrons de Conception

State, Strategy, Façade, Singleton, Adapter, Observer, Fabrique, Composite

	Patrons créateurs	Patrons structuraux	Patrons comportementaux
Classes	<i>Factory Method</i>	<i>Adapter*</i> (class)	Interpreter <i>Template Method</i>
Objets	Abstract Factory Builder Prototype <i>Singleton*</i>	Adapter (object) Bridge <i>Composite</i> Decorator <i>Façade*</i> Proxy (ou mediateur)	Command <i>Iterator</i> Mediator Memento <i>Observer</i> <i>State*</i> <i>Strategy*</i> Visitor

# Le pattern State

Pattern comportemental à portée Objets





# State

- **Objectif**
- Il permet à un objet de modifier son comportement quand son état interne change.
- Permet d'exécuter des actions en fonction d'un contexte

## Exemple Commande

- Une commande possède une liste de produits
- Elle passe de l'état en cours, à validée, puis livrée et archivée.
- Seule une commande en cours peut voir sa liste de produits évoluer.
- Une commande validée dont la livraison est effectuée passe à l'état 'Livrée'
- Après une période définie (12 mois), la commande est archivée.

# Premier code

- On pourrait traiter la commande de façon unitaire, de bout en bout, en contrôlant les états et les traitements sur la commande.

- Exemple :

À la **création** d'une nouvelle commande : `setEtat("enCours");`

Dans la méthode `ajouterProduit()` :

```
if (this.état == "enCours") // ajouter un produit
```

Idem pour **modifier** / **supprimer** un produit d'une commande

Dans la méthode `setDateLivraison()` :

```
if (this.état == "validée") // définir la date de livraison
```

Dans la méthode `setLivraisonEffectuée(boolean b)`, on vérifie qu'elle est bien validée,  
Etc.

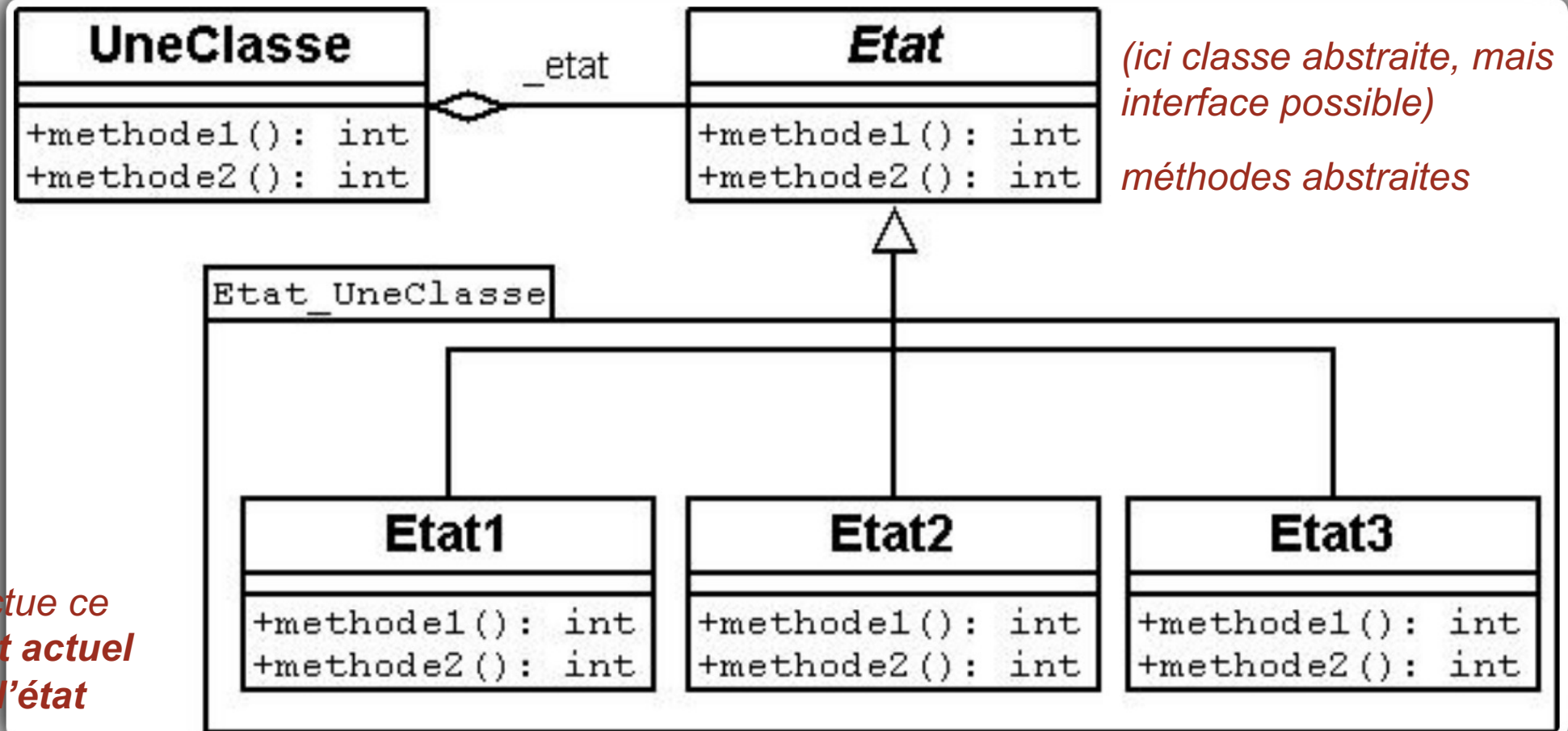
- Rend interdépendants les différents traitements
- Code difficile à faire évoluer et à maintenir : par exemple, si on introduit la possibilité d'annuler une commande (au moins 24h avant sa livraison) → nouveaux tests, enchevêtrement des logiques, etc.

# Coder avec le DP State

- L'idée serait de gérer ces différentes étapes du traitement de la commande, **indépendamment**.
- En créant des objets pour chaque étape, chacun ayant les comportements dédiés à chaque étape
  - Ex. état Validée : on peut définir la date de livraison, mais pas modifier les articles
  - Cela permet d'ajouter par la suite de nouveaux états, sans modifier ce qui existe (OCP)
  - Sépare clairement les rôles des étapes (SRP)

# DP State

Classe de contexte,  
qui possède  
**différents états**  
Etat1, Etat2..., où  
seul l'état courant  
est gardé



Chaque opération effectuée ce  
qui **correspond à l'état actuel**  
de la classe et précise l'état  
suivant

```
methode1() :  
//... traitement qd en Etat1  
uneClasse.setEtat( new Etat3() );
```

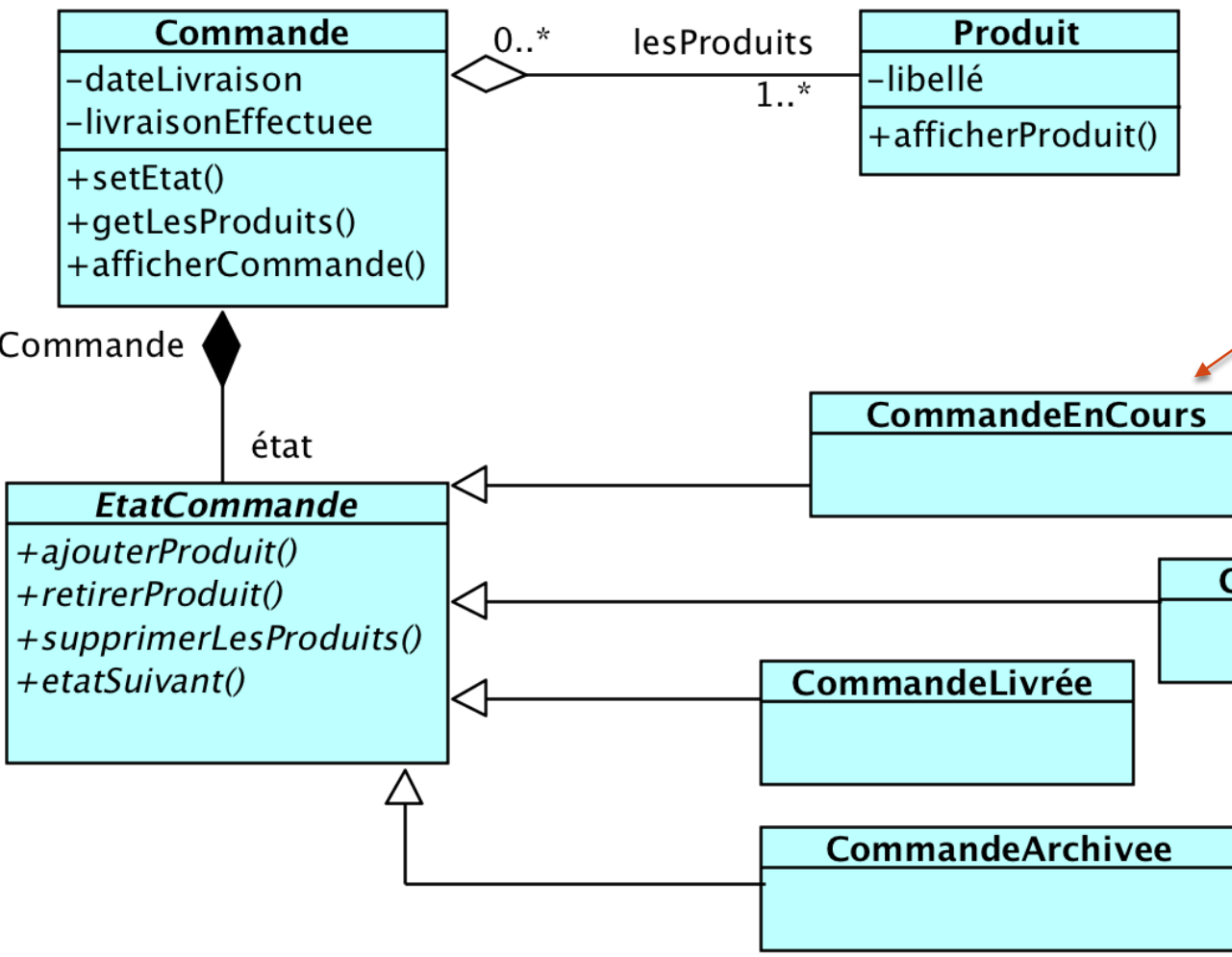
```
methode1() :  
//... traitement qd en Etat3  
uneClasse.setEtat( new Etat2() );
```

## Participants

- *UneClasse* : définit l'objet dont on veut gérer l'état. L'attribut *\_état* définit l'état courant de l'objet. Cet attribut est lui-même un objet implémentant « *Etat* ».
- *Etat* : Définit une interface pour *encapsuler le comportement* correspondant à un état de l'objet
- *Etat1, Etat2...* : sous-classes définissant chacune un état concret et surtout le comportement possible de cet état (faire passer l'objet d'un état à un autre). Les états n'ont pas conscience les uns des autres. On peut en ajouter, en supprimer, sans modifier *UneClasse*.

## Fonctionnement

- *UneClasse* délègue les invocations des opérations à l'objet « *Etat* » représentant l'état courant.
- Le changement d'état d'un objet est défini dans les opérations des sous-classes *Etat1, Etat2...*



**Constructeur** appelé dans le constructeur de Commande avec appel à la méthode `setEtat( new CommandeEnCours())`

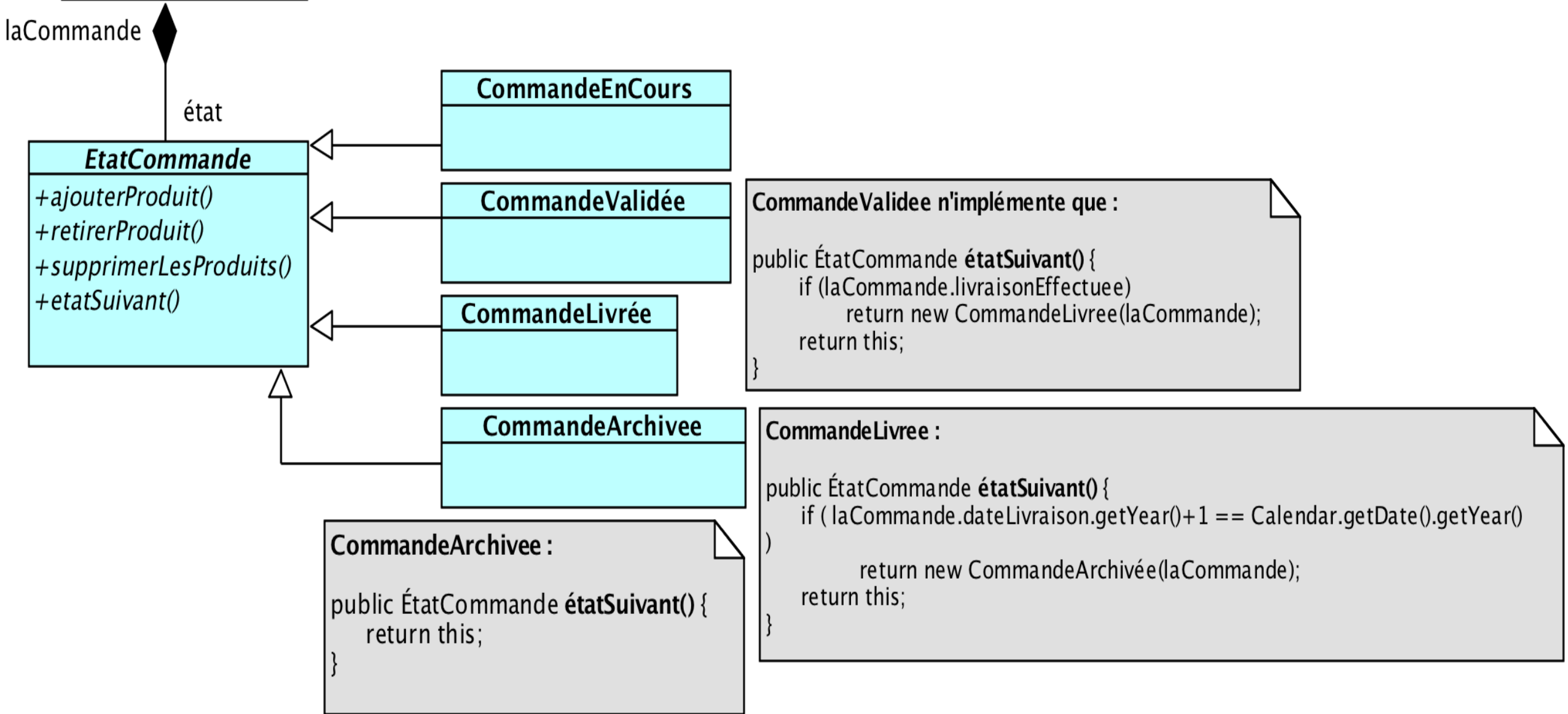
Implémente **ajouterProduit()**, **supprimerLesProduits()**, **retirerProduit()**, **étatSuivant()**

**par ex.:**

```
public ÉtatCommande étatSuivant() {
    return new CommandeValidée(laCommande);
}
```

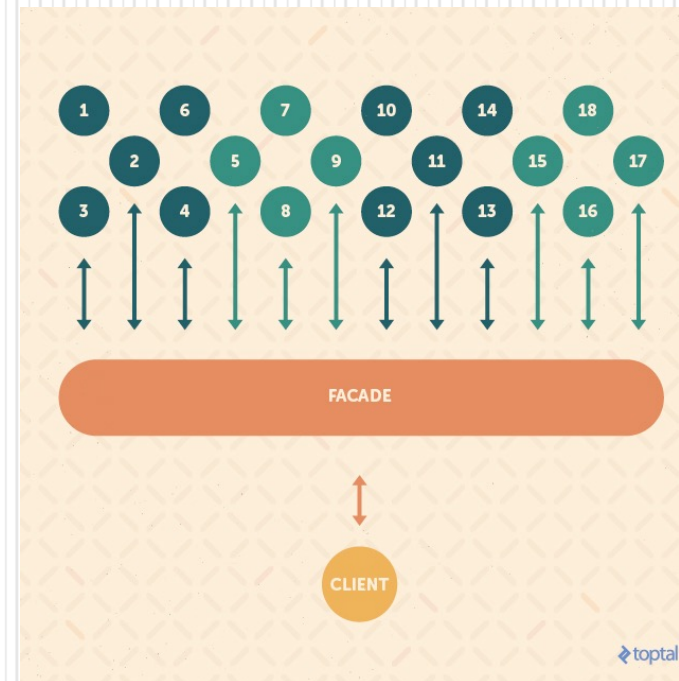
```
public void supprimerLesProduits() {
    laCommande.getLesProduits().clear();
}
```

## Exemple : Commande de Produit



# Le pattern Façade

Un modèle de conception de type Structure à portée Objets





# Le pattern « Façade »

- **Problème** : on a besoin de n'utiliser qu'un sous-ensemble d'un système existant
- **But** : offrir une interface **s**implifiée à un ensemble de composants
- **Conséquence** : fournit une interface de plus haut niveau
  - Cela rend le sous-système plus facile à utiliser
  - Mais certaines fonctionnalités pourront rester inaccessibles au client.

# Pattern Façade (2)

- **Implémentation** : définir une nouvelle classe possédant l'interface requise; implémenter cette classe à l'aide des fonctionnalités du système existant
- **Cas d'utilisation** :
  - Soit interface actuelle **pas assez conviviale**
  - Soit on cherche à utiliser le système **d'une façon particulière**
    - Ex. utiliser un logiciel 3D pour faire de la 2D : on va isoler les fonctionnalités utiles pour la partie Client
  - Soit on **veut limiter l'accès** à une partie du sous système

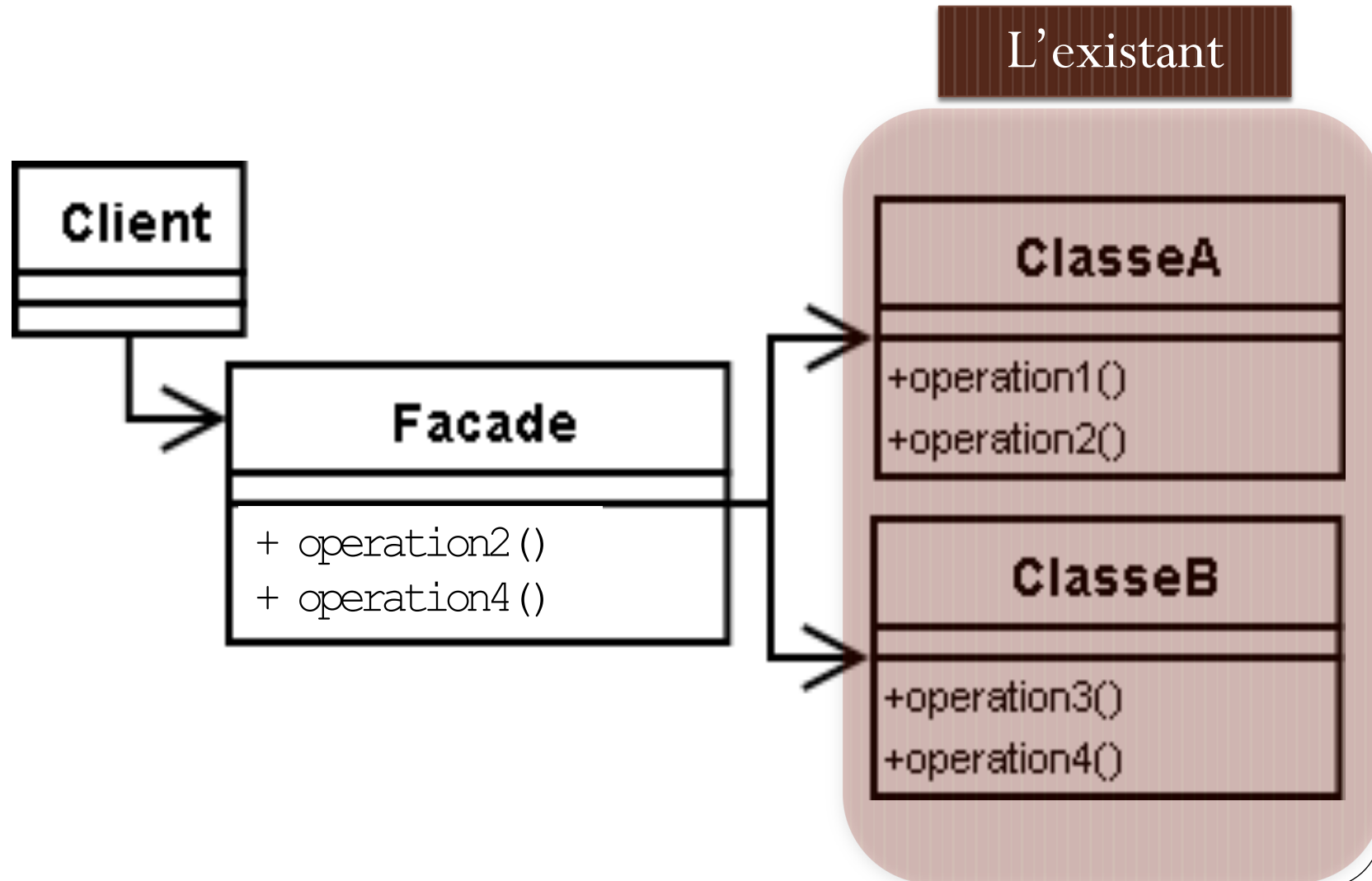
# Ex. un Standard



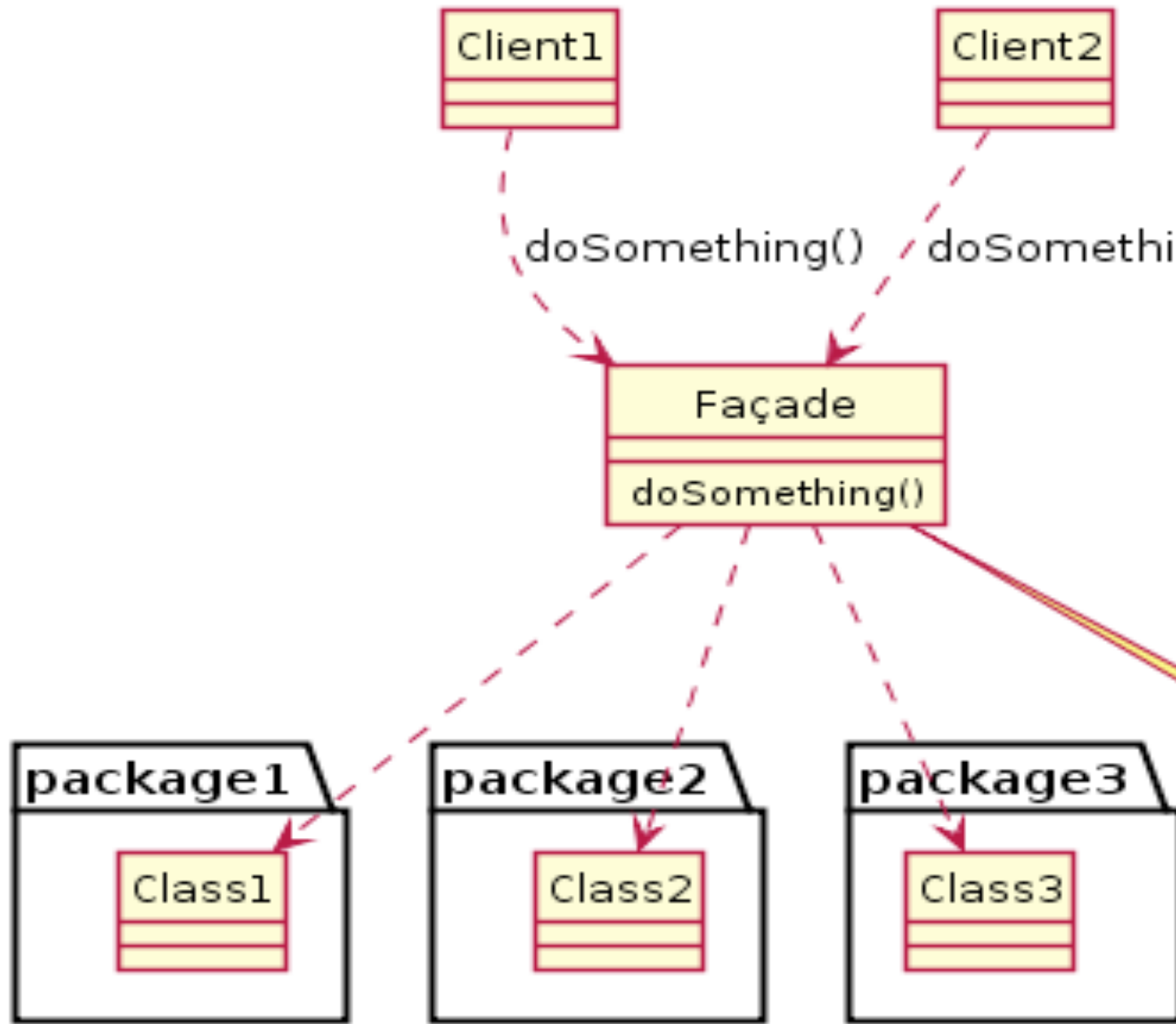
Service Commercial

Service Approvisionnement

Comptabilité

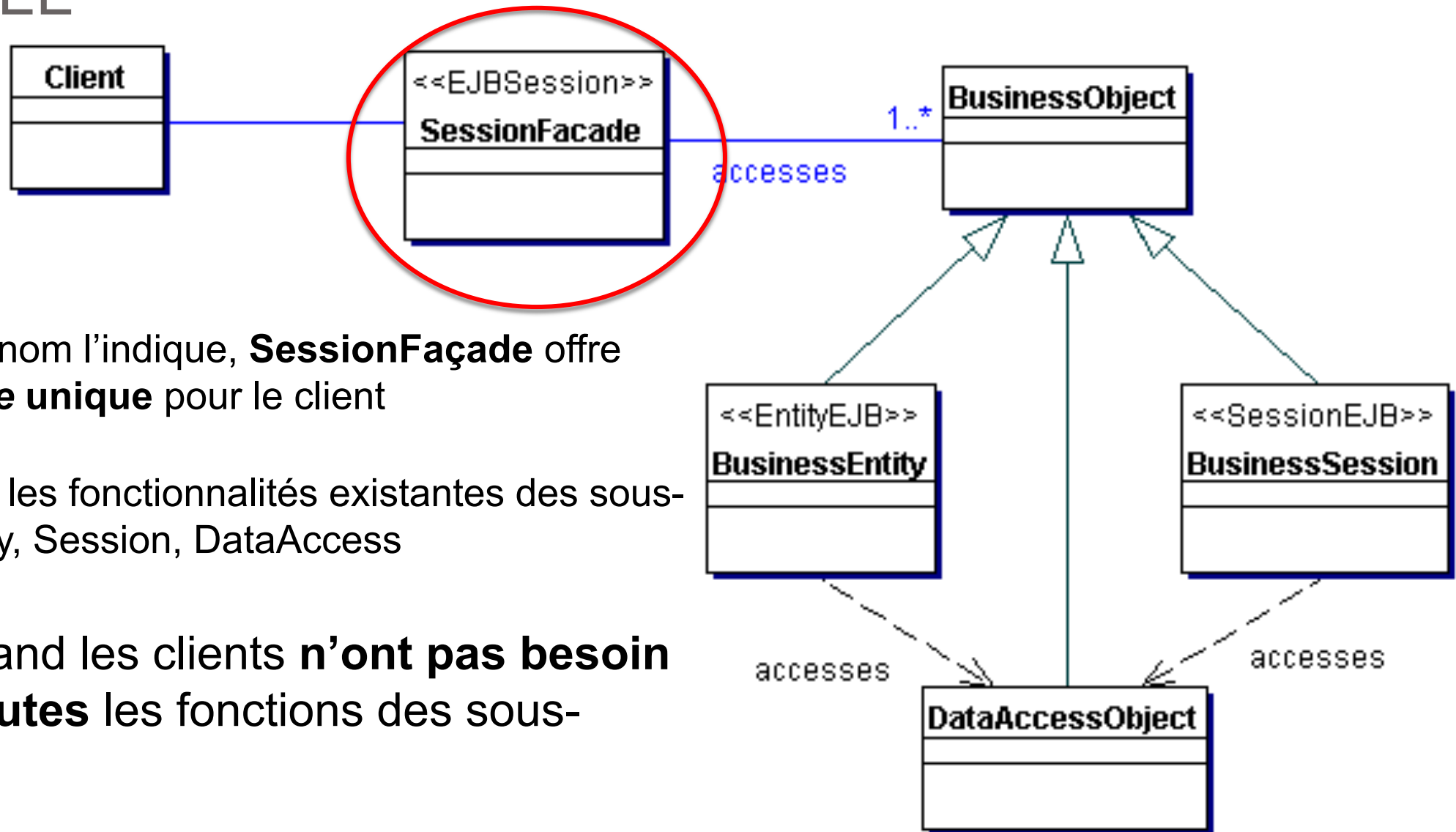


Ex. une méthode qui requiert les méthodes de différents objets



```
doSomething() {
    Class1 c1 = new Class1();
    Class2 c2 = new Class2();
    Class3 c3 = new Class3();
    c1.doStuff(c2)
    c3.setX(c1.getX());
    return c3.getY();
}
```

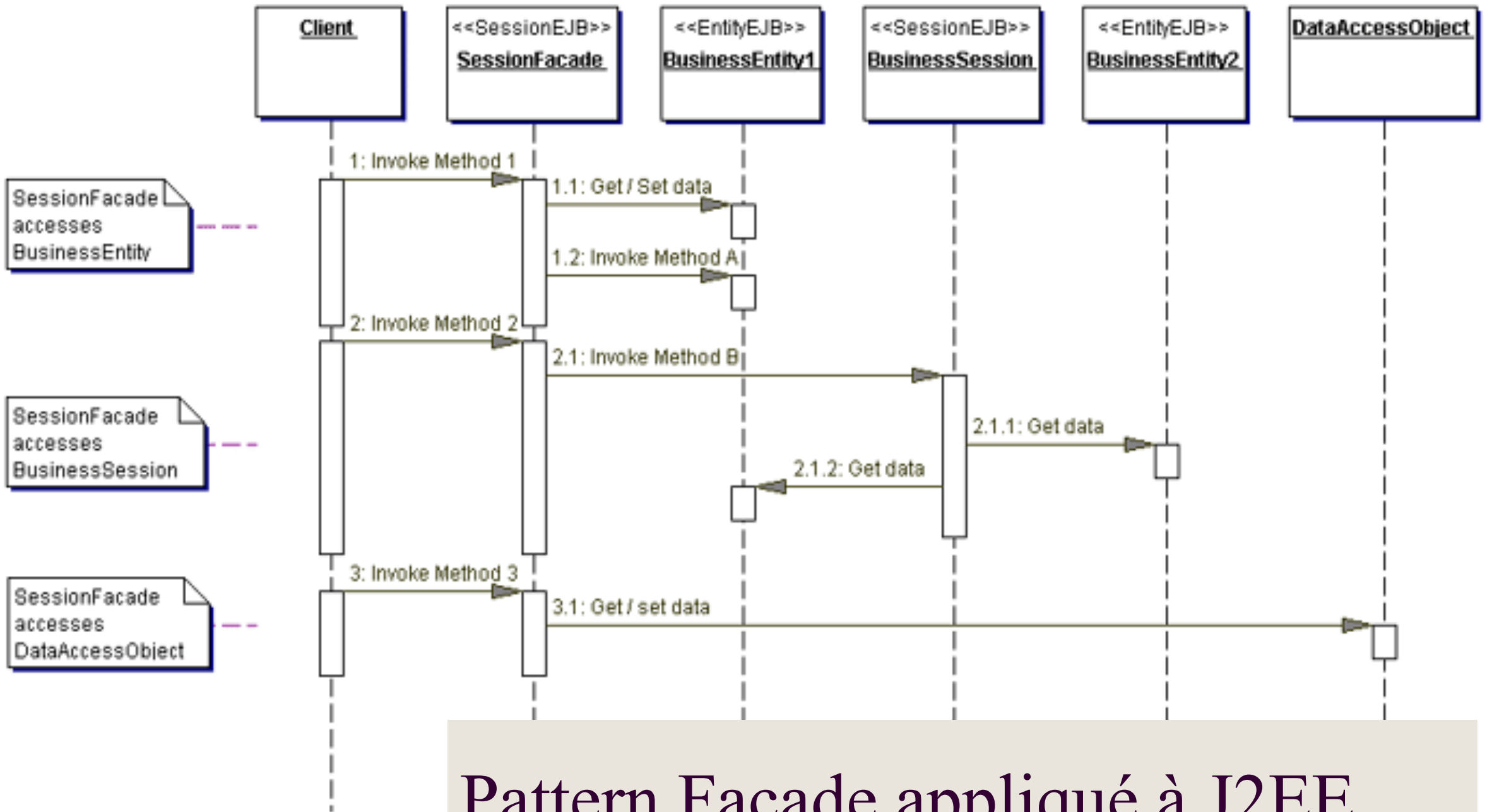
# Illustration : Façade pour accéder aux services et données avec J2EE



Comme son nom l'indique, **SessionFaçade** offre une **interface unique** pour le client

Élaborée sur les fonctionnalités existantes des sous-classes Entity, Session, DataAccess

Intérêt : quand les clients **n'ont pas besoin** d'utiliser **toutes** les fonctions des sous-classes



# Pattern Façade appliqué à J2EE

## Ex. Façade pour une utilisation simplifiée du calendrier de l'API Java

```
/* pattern Façade */
class UserfriendlyDate {
    GregorianCalendar gcal;
    public UserfriendlyDate(String isodate_ymd) {
        String[] a = isodate_ymd.split("-");
        gcal = new GregorianCalendar(Integer.parseInt(a[0]),
            Integer.parseInt(a[1])-1 /* !!! */, Integer.parseInt(a[2]));
    }
    public void addDays(int days) {
        gcal.add(Calendar.DAY_OF_MONTH, days);
    }
    public String toString() {
        return String.format("%1$tY-%1$tm-%1$td", gcal);
    }
}

/* Client */
class FacadePattern {
    public static void main(String[] args) {
        UserfriendlyDate d = new UserfriendlyDate("2015-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```

# Singleton

Un design pattern de  
type *Création* à  
portée *Objet*



<http://yavkata.co.uk>



# Design pattern « Singleton »

- Une des techniques les plus utilisées en conception objet
- « Comment s'assurer de n'instancier qu'**une seule fois** une classe (utilisée plusieurs fois) ? »
- Permet de référencer l'instance d'une classe lorsqu'elle est, ***par construction***, le seul et unique représentant de la classe
  - ex. : une connexion à une BD, un fichier de log, un spooler d'imprimante, un gestionnaire de cache, le moteur d'un jeu, etc.
- Objet unique : accessible par les autres instances de classes.

# Singleton (2)

- Ex. classe *java.util.Calendar* utilise un singleton pour renvoyer la date courante
- Un singleton est donc une classe appelée *Singleton* composé d'un attribut :
  - *instance* qui recevra la référence de l'objet unique
- et d'une opération :
  - *getInstance()* qui va chercher cette référence et la stocke dans *instance* si l'objet existe

# Singleton (3)

- `getInstance()` se charge d'automatiquement construire l'objet **unique** au 1er appel :

```
public synchronized static Singleton getInstance() {  
    if (_instance == null)  
        _instance = new Singleton();  
    return _instance;  
}
```

- Le constructeur est **privé**
- `synchronized` empêche toute instantiation multiple, même par différents threads

# Pattern Singleton

**synchronized** : en cas d'accès multi-threads

## Singleton

```
private static Singleton uniqueInstance
....
singletonData

private Singleton()
public static synchronized Singleton getInstance()
public static synchronized void releaseInstance()
....
singletonOperation()
```

```
return
uniqueInstance
```

Variante : on peut créer l'instance lors de la définition de la variable :

```
private static final Singleton _instance = new Singleton();
```

Du coup, plus besoin de **synchronized** : **getInstance()** retourne simplement l'instance.

```
final class Singleton {  
    private static Singleton uniqueInstance = null;  
    // constructeur privé  
    private Singleton() {}  
    // méthode qui crée une instance unique du singleton  
    // protégée en cas d'accès multi-threads par synchronized  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null)  
            uniqueInstance = new Singleton();  
        return uniqueInstance;  
    }  
    public static synchronized void releaseInstance() {  
        uniqueInstance = null;    // libère l'instance  
    }  
    // méthode du singleton  
    public static void affiche() {  
        System.out.println("on est dans le Singleton!!!");  
    }  
} // Singleton
```

Ici une méthode  
pour libérer  
l'instance de la  
classe

# Ex. Logistique

- Pour la journalisation, le concepteur désire créer en local un **seul** et **unique** fichier de traces **par jour**.
- On va utiliser un *singleton*
- Et enrichir la méthode `getInstance()` pour contrôler la date de création
  - N'avoir qu'un fichier de trace par jour de la semaine

```
public class FichierTrace {
```

```
private Date _dateCreation; // date de création du dernier fichierTrace créé  
private static FichierTrace _instance; // l'objet fichierTrace créé pour le jour  
private FileOutputStream leFichier; // le fichier de trace du jour de semaine
```

```
public static FichierTrace getInstance() {
```

```
// la classe Calendar utilise aussi un Singleton pour la date courante :  
int day = Calendar.getInstance().get(Calendar.DAY_OF_WEEK);
```

```
    if (_instance == NULL || _dateCreation.getDay() != day) {  
        _instance = new FichierTrace(day);  
    }
```

```
    return _instance; // retourne l'instance du FichierTrace du jour
```

```
}
```

```
//constructeur (privé):
```

```
private FichierTrace(int day) {  
    if (NULL != leFichier) leFichier.close();  
    _dateCreation = Calendar.getInstance();  
    leFichier = new FileOutputStream(); // etc.
```

```
...}
```

Un fichier  
de traces  
par jour  
de semaine

On enregistre la date de création  
de la dernière instance créée

Astuce

## Éviter synchronized dans getInstance()

```
public final static Singleton getInstance() {  
  
    /* Pour éviter un appel coûteux à synchronized, une fois que  
       l'instanciation est faite : */  
    if (Singleton.instance == null) {  
        /* Le mot-clé synchronized sur ce bloc empêche toute instanciation  
           multiple même par différents "threads" */  
        synchronized(Singleton.class) {  
            if (Singleton.instance == null {  
                Singleton.instance = new Singleton();  
            }  
        }  
    }  
    return Singleton.instance;  
}
```



# Le pattern Strategy

Pattern comportemental à portée Objets



# Le pattern STRATEGY (1/2)

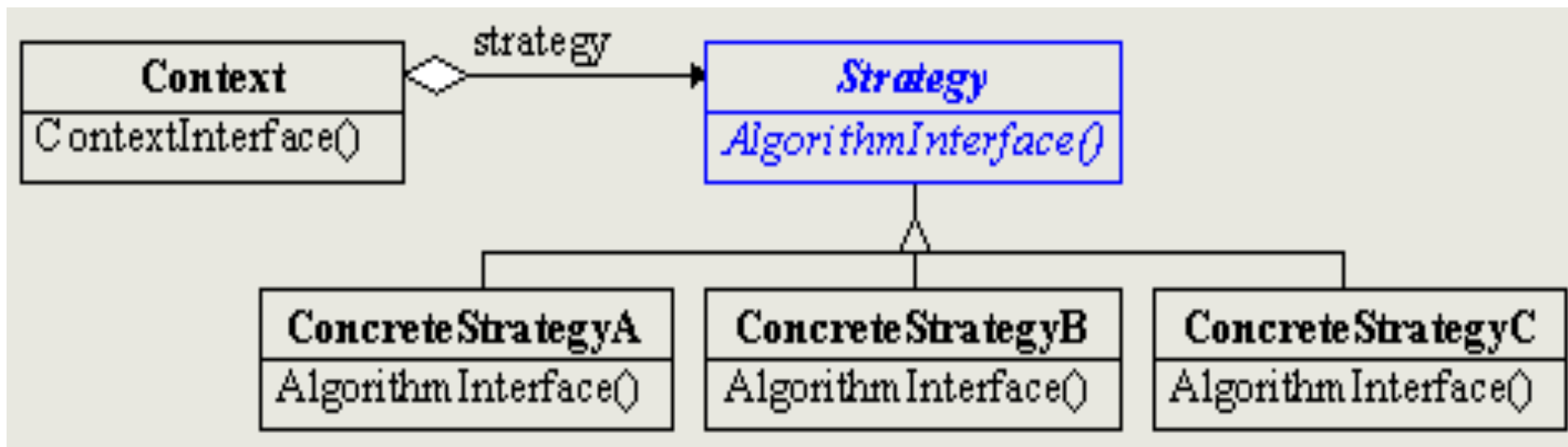
- **Problème** : réorganiser une solution particulière (ex.: algorithme) pour en faire une solution générique
- **But** : définir un ensemble d'algorithmes répondant à un même problème, encapsuler chacun et les rendre interchangeables
- **Conséquence** : le pattern définit une famille d'algorithmes, définit des classes de réalisation indépendantes, les rend dynamiquement interchangeables.
  - On peut ajouter / supprimer des algorithmes
  - Il est possible d'échanger dynamiquement d'algorithme sans modifier les classes clients qui les utilisent

# Le pattern STRATEGY (2/2)

## Cas d'utilisation

- On a une hiérarchie de classes nombreuses qui se distinguent uniquement par leurs comportements
  - Ex.: le comportement alimentaire ou de reproduction des animaux vertébrés, l'énergie et le terrain de déplacement d'un véhicule, etc.
- Différentes versions d'algorithmes sont nécessaires
- Une classe définit plusieurs comportements traduits par des branches conditionnelles dans ses méthodes
  - *Switch / case* ou *if* imbriqués

# Architecture STRATEGY



**Délégation d'opération** via un attribut *uneStratégie* de type *Strategy*, dans *Context* :

```
public void contexteInterface() {  
    uneStratégie.algorithmInterface();  
}
```

(le choix de la stratégie est affectée lors de l'instanciation du contexte, et peut être modifiée ensuite de manière transparente)

# ADAPTER

L'adaptateur, un autre patron de structure



# Design pattern ADAPTER

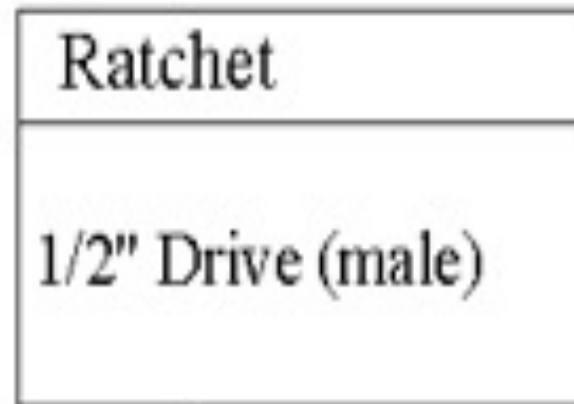
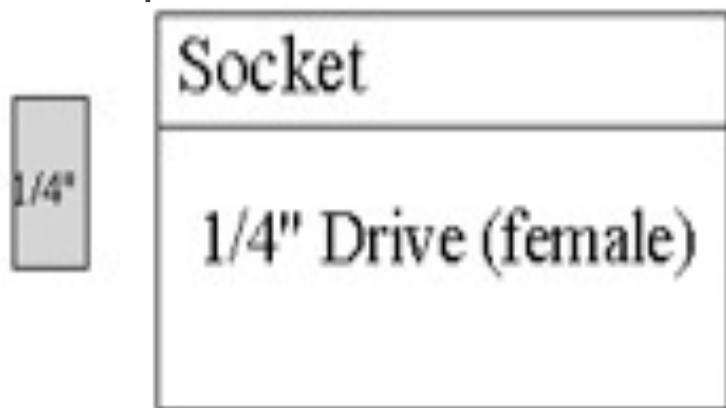
- Il consiste à transformer les points d'entrée d'un composant
  - que l'on désire intégrer
  - à l'interface souhaitée par le concepteur
- Mécanisme de **délégation**



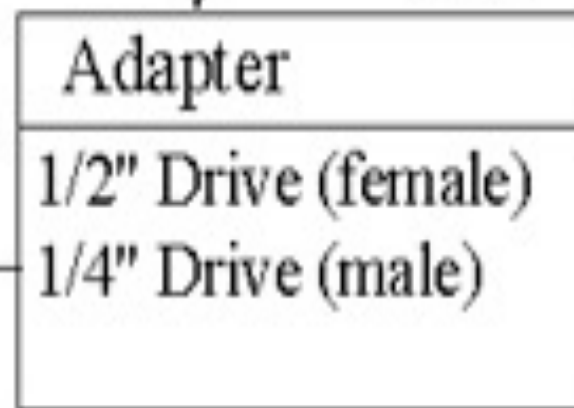
Une clé à cliquet (ratchet) utilise une série de douilles (socket)



Les petites douilles nécessitent un raccord 1/4 pouces



La clé ne fournit que le raccord 1/2 pouces



Source : Michael Duell  
"Non-software examples of software design patterns"

# ADAPTER (2)

- Objectif : transformer 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 une méthode **Afficher()**
- **Les objets clients appellent ces formes pour les afficher**



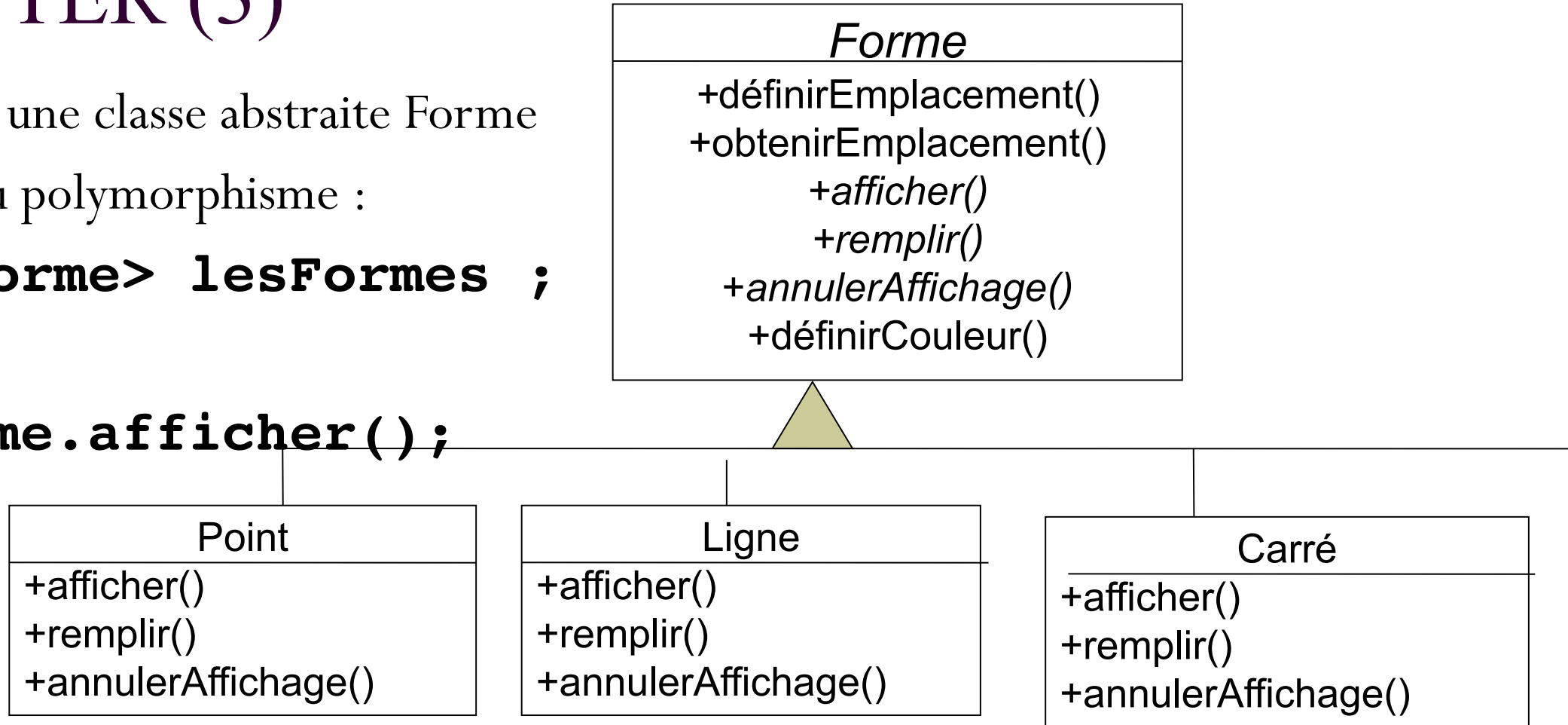
# ADAPTER (3)

- On crée une classe abstraite *Forme*
- Grâce au polymorphisme :

```
List<Forme> lesFormes ;
```

...

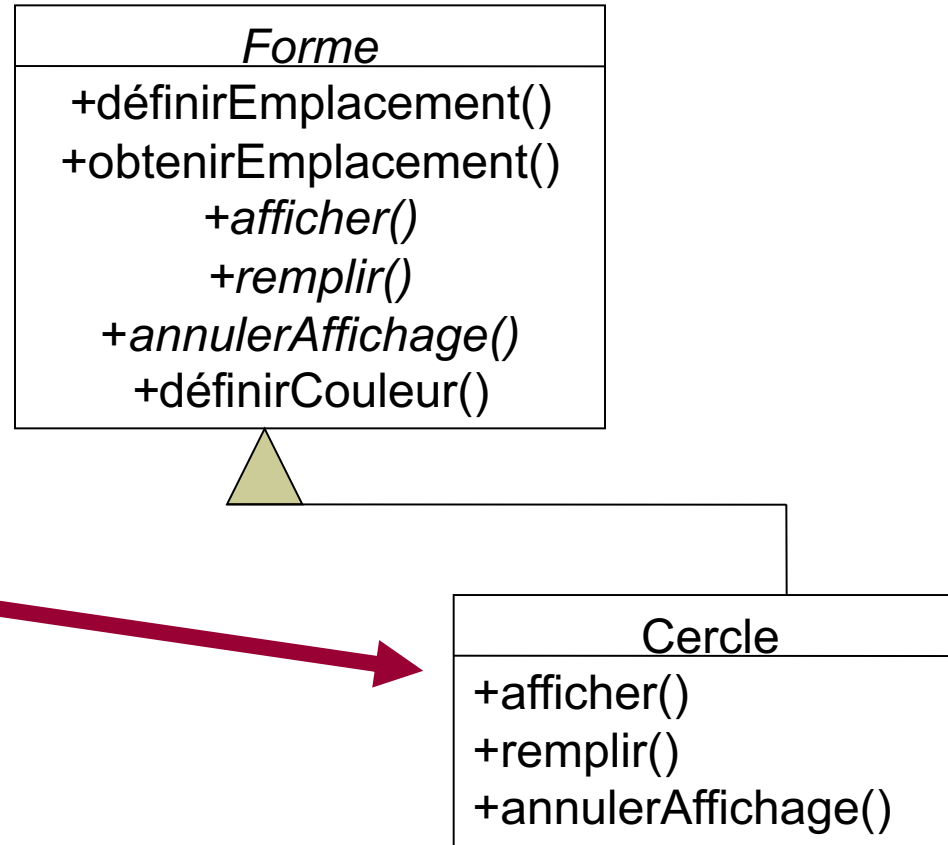
```
uneForme.afficher();
```



# ADAPTER (4)

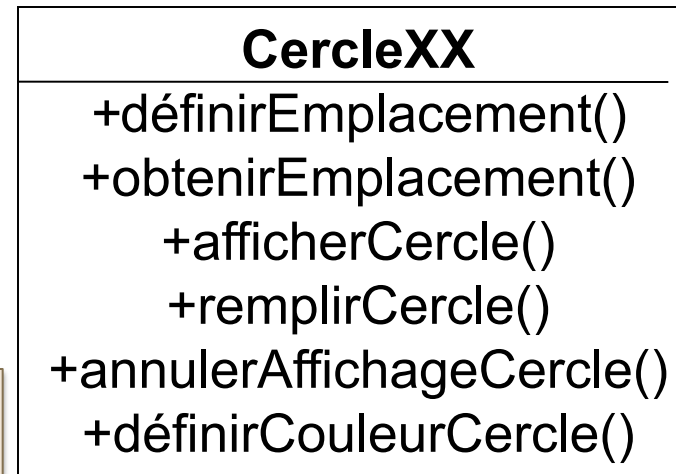
- Imaginons qu'on ait besoin d'une autre forme :
- le cercle

**On pourrait créer une classe Cercle dérivée de Forme**



# ADAPTER (5)

- Mais supposons que l'on ait déjà une classe **CercleXX** existante : elle possède les méthodes requises de **Forme**, mais pas la bonne signature



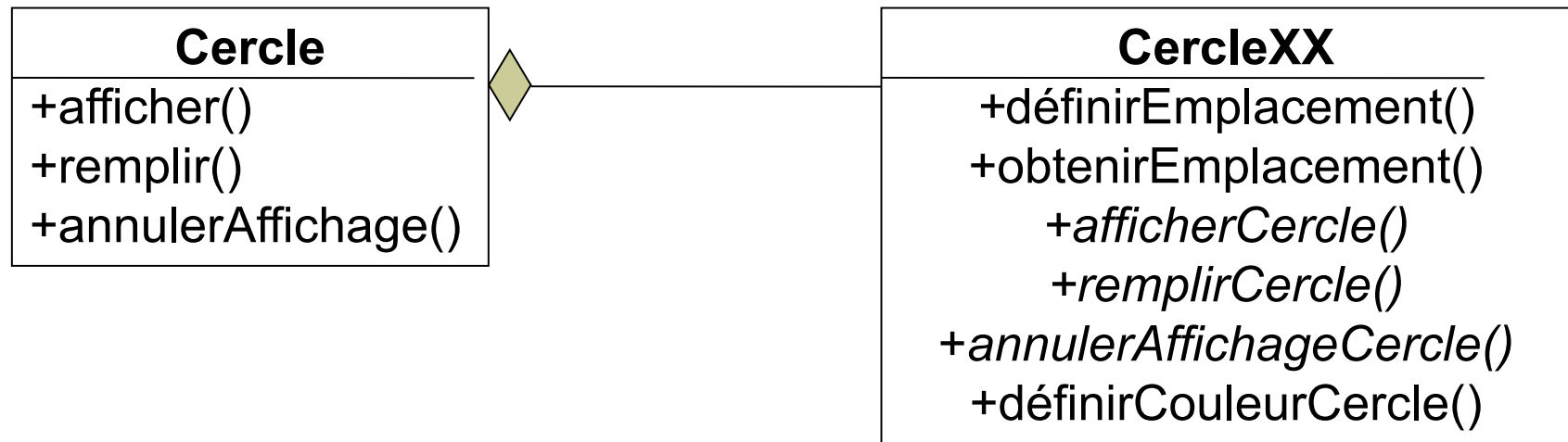
On pourrait modifier les opérations de la classe **CercleXX** pour les adapter au format de **Forme**



Mais **CercleXX** est utilisée dans d'autres programmes !

# Adaptateur (6)

- On va réaliser un objet Adaptateur : **Cercle**
  - Qui va contenir (**encapsuler**) l'objet **CercleXX** existant
  - Implémenter les méthodes attendues de **Forme**
- Tout ce que fait l'objet **Cercle** est transmis à l'objet **CercleXX** en faisant appel à ses opérations



# ADAPTER (7)

- Extrait du code Java correspondant :

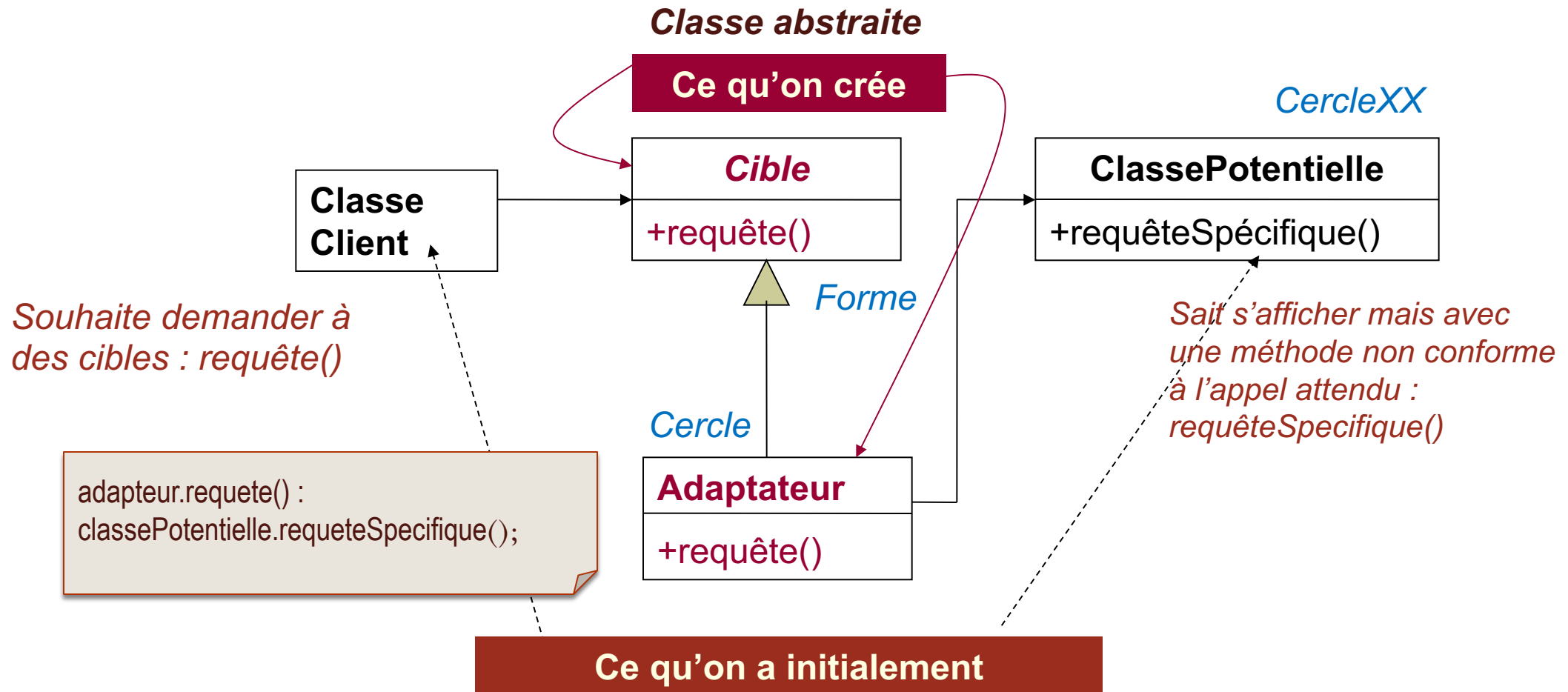
```
Class Cercle extends Forme {  
    ...  
    private CercleXX pcx;  
    ...  
    public Cercle() {  
        pcx = new CercleXX();  
    }  
    void public afficher() {  
        pcx.afficherCercle();  
    }  
}
```

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

# ADAPTER, selon le GoF(9)

- **Objectif** : faire correspondre à une interface donnée un objet existant
- **Contexte** : 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 par délégation.

# ADAPTER : les classes



# ADAPTER vs. Façade (1)

- Ils impliquent tous deux des classes existantes qui n'ont pas l'interface voulue
- Le pattern *Façade* **simplifie** l'interface alors que *l'Adaptateur* **convertit** (encapsule) un objet pour coller avec l'interface **existante**
  - *Façade* : encapsulation pour faciliter une utilisation
  - *Adaptateur* : encapsulation pour un besoin structurel
- Le pattern *Façade* masque généralement *plusieurs* classes, *l'Adaptateur* une seule
- Leurs méthodes d'encapsulation diffèrent
  - Cf tableau comparatif



# Adaptateur vs. Façade (2)

	Façade	Adaptateur
Classes existantes ?	OUI	OUI
Utiliser une interface spécifique ?	NON	OUI
Polymorphisme nécessaire ?	NON	OUI
Simplifier l'interface ?	OUI	NON