Chap.8 – Conception UML
Design patterns (part. 1)

V. Deslandres ©

Licence Professionnelle SIL option DevOps

Développeur et Administrateur de Systèmes d'information

IUT de Lyon, site Doua - Université Lyon 1

Sommaire de ce cours

- Introduction ------------------------ #3
- Liste des DP ------------------- #11
- Le patron Singleton ---------------- #33
- Le patron Proxy ------------------- #21
- Le patron State ------------------- #26
- Le patron Façade ------------------- #35
- Le patron Strategy ---------------- #43
- Le patron Adapter ------------------- #47

Les Design Patterns : c'est quoi ?

- **Design patterns** = Modèles de conception (*patrons de conception*) pour la POO
- Répondent à des problèmes **récurrents** de la conception OO
  - **Diminution du couplage**, **Séparation des rôles,** Indépendance vis-à-vis des plateformes, **Réutilisation** du code existant, Facilité **d'extension**
- Proposer un catalogue de **meilleures pratiques** issue de **l'expérience** de concepteurs chevronnés

- 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)

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

© V. Deslandres – Univ. Lyon1

Bénéfices des Design Patterns

1. **Capitalisation** de l'expérience et **réutilisation** de solutions
   - Plus puissant que la réutilisation de codes
   - Amène souvent la réutilisation de *composants*
2. **Vocabulaire** commun pour la conception
   - « On fait un Singleton ? »
3. Niveau d'abstraction **plus élevé**
   - Constructions logicielles de meilleure qualité
4. Souvent : meilleure **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 ?
  - Pas tous du même niveau :
    - 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 d'en voir les avantages

Typologie des Design Pattern / Description

- **Classification des 23 patrons de Gamma :**
  - Selon la **fonction**
    - modèle de **création**,
    - modèle de **structure** (assemblage d'objets),
    - modèle **comportemental**
  - Selon la **portée**
    - **classes** : héritage
    - **objets** : délégation
- Exemples :
  - Pattern Composite = structurel / objets
  - Pattern Abstract Factory = création / classes

Description standard :

- **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
  - Cacher ce qui est créé, qui crée, où, comment et quand.
- Rendre indépendant la façon dont les objets sont créés, composés ou initialisés

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**

Image : titline.fr

Présentation de quelques Patrons de Conception

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

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

* *vus dans ce cours*

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.
  - Permet aussi de limiter l'usage des ressources.

- Objet unique : accessible par les autres instances de classes.

Calendar : un Singleton

- 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 : caractéristiques

- `getInstance(): Singleton` 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 instanciation multiple, même par différents threads

Pattern Singleton

```
Singleton

private static Singleton uniqueInstance
....
singletonData

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

```
return
uniqueInstance
```

Variante : on peut aussi 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.

Java

```java
final class MonSingleton  {

    // variable de classe privée :
    private static MonSingleton uniqueInstance = null;
    // constructeur privé :
    private MonSingleton() {}
    // méthode qui crée une instance unique du singleton :
    public static synchronized MonSingleton getInstance() {
       if (uniqueInstance == null)
            uniqueInstance = new MonSingleton();
         return uniqueInstance;
    }
     // Méthode qui libère l'instance :
    public static synchronized void releaseInstance() {
     uniqueInstance = null;
    }
    // Autre méthode du singleton :
    public static void affiche() {
       System.out.println( »*** On est dans le Singleton");
    }
 } // de MonSingleton
```

Ex. Journalisation

- 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

Un fichier de traces **par jour de semaine**

```
public class FichierTraceJour {

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

    public static FichierTraceJour 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 FichierTraceJour(day);
          }
          return _instance; // retourne l'instance du FichierTraceJour du jour
  }
```

```java
    //constructeur (privé):
        private FichierTraceJour(int day) {
            if (NULL != leFichier) leFichier.close();
            setDateCreation( Calendar.getInstance() );
            leFichier = new FileOutpuStream(); // etc.
     }
   …
} //de FichierTraceJour
```

Le pattern Proxy

Pattern comportemental à portée Objets

Design Pattern Proxy

Problème :

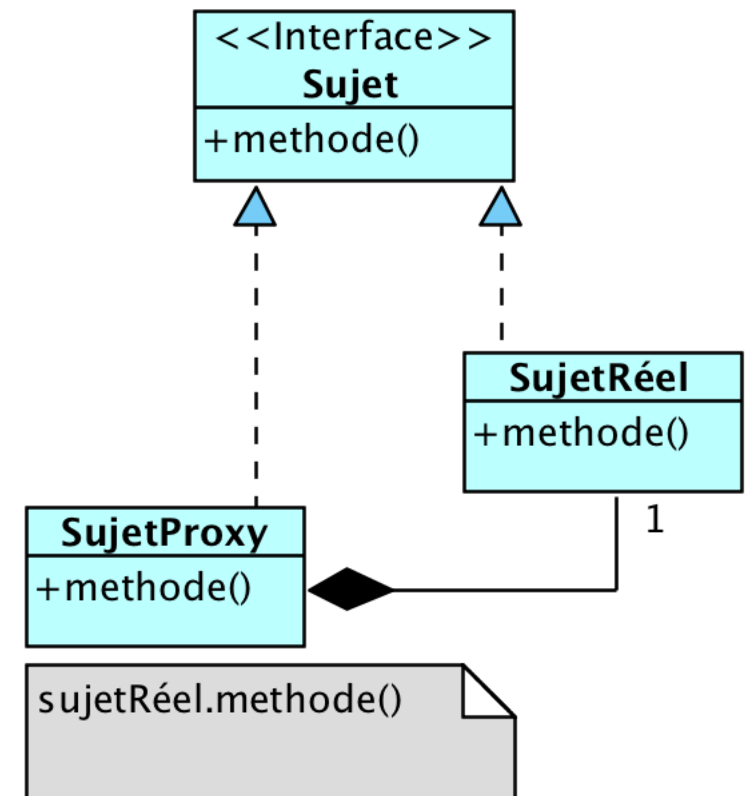
- On a besoin de références à un objet, qui soient plus **créatives** et plus **sophistiquées** qu'un simple pointeur.

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();
}
```

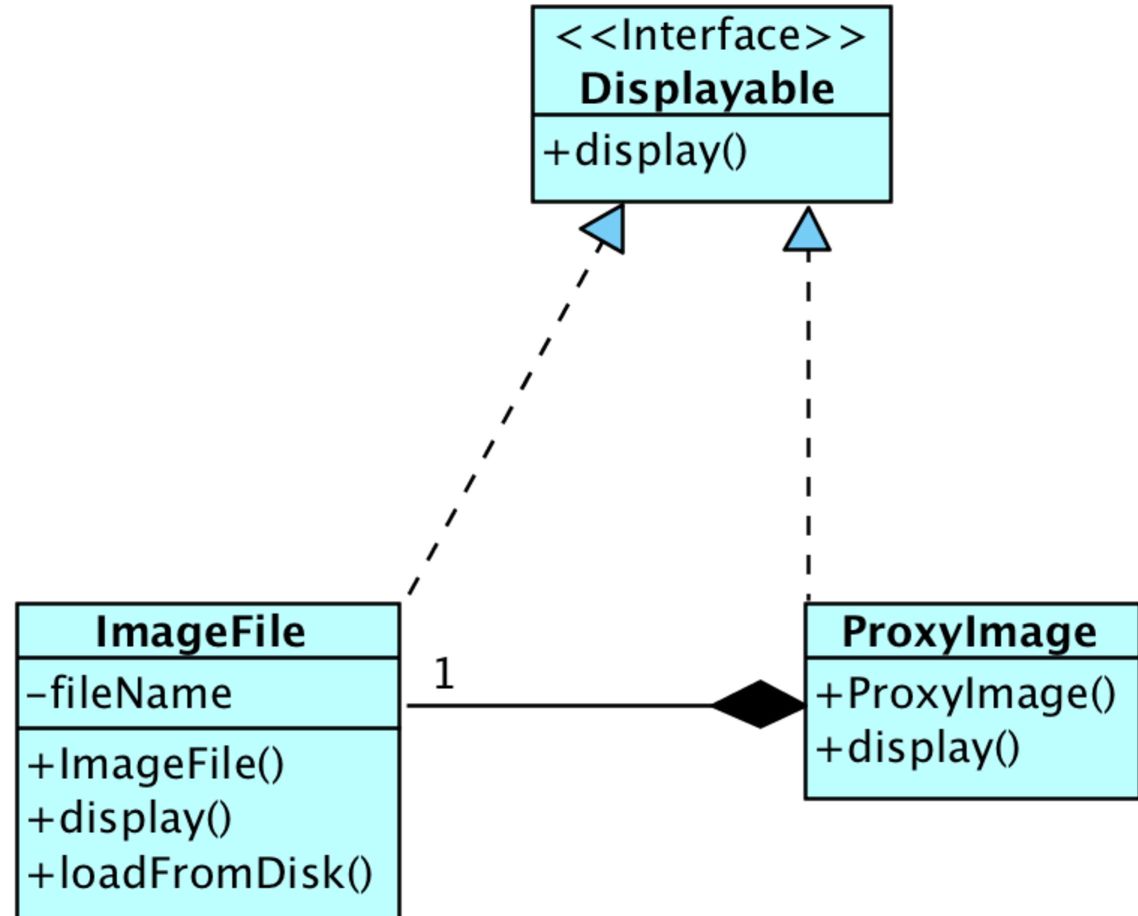
```
public class ImageFile implements Displayable {

   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);
   }
}
```



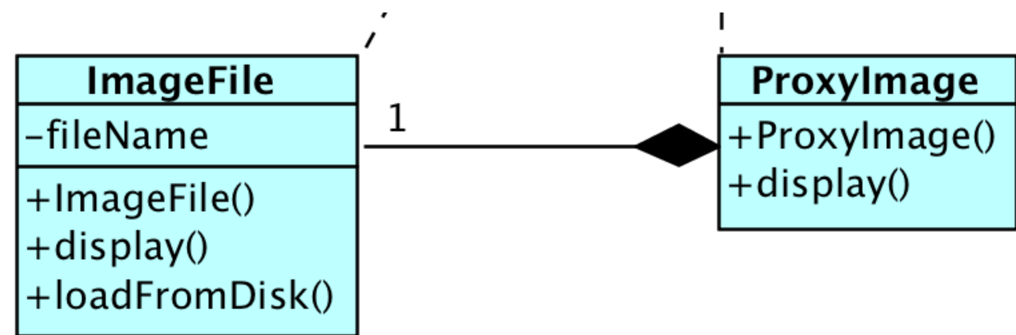
```java
public class ProxyImage implements Displayable {

  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();
  }
}
```

Le pattern State

Pattern comportemental à portée Objets

Etat d'un objet ?

- Si l'état est défini par un seul attribut : ex.: une commande (validée, en cours, etc.)
  - Imaginons que **plusieurs méthodes** peuvent modifier cet état, par ex.: définir(), ajouterProduit(), annuler(), définirDateLivraison(), archiver()
  - De fait on aura peut-etre besoin de distinguer ces différents états : en cours / validée / livrée / archivée … pour savoir ce qu'il est possible de faire.

→ La logique dépend de l'état de l'objet

→ Elle est répartie dans différentes méthodes de la classe

→ Code répété : `si (état == e1) faire ceci sinon si (état == e2) faire cela,` etc…

L'idée de **STATE** :

→ Constituer des classes pour chaque Etat et répartir cette logique dans ces classes.

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 de produits

- 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.

Première implémentation

- 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'un 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,** l'état passe à : « à livrer »

  Dans la méthode `setLivraisonEffectuée(boolean b)` :

  `if (this.état == "à livrer")` // on vérifie que la commande a été livrée et on l'archive

  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

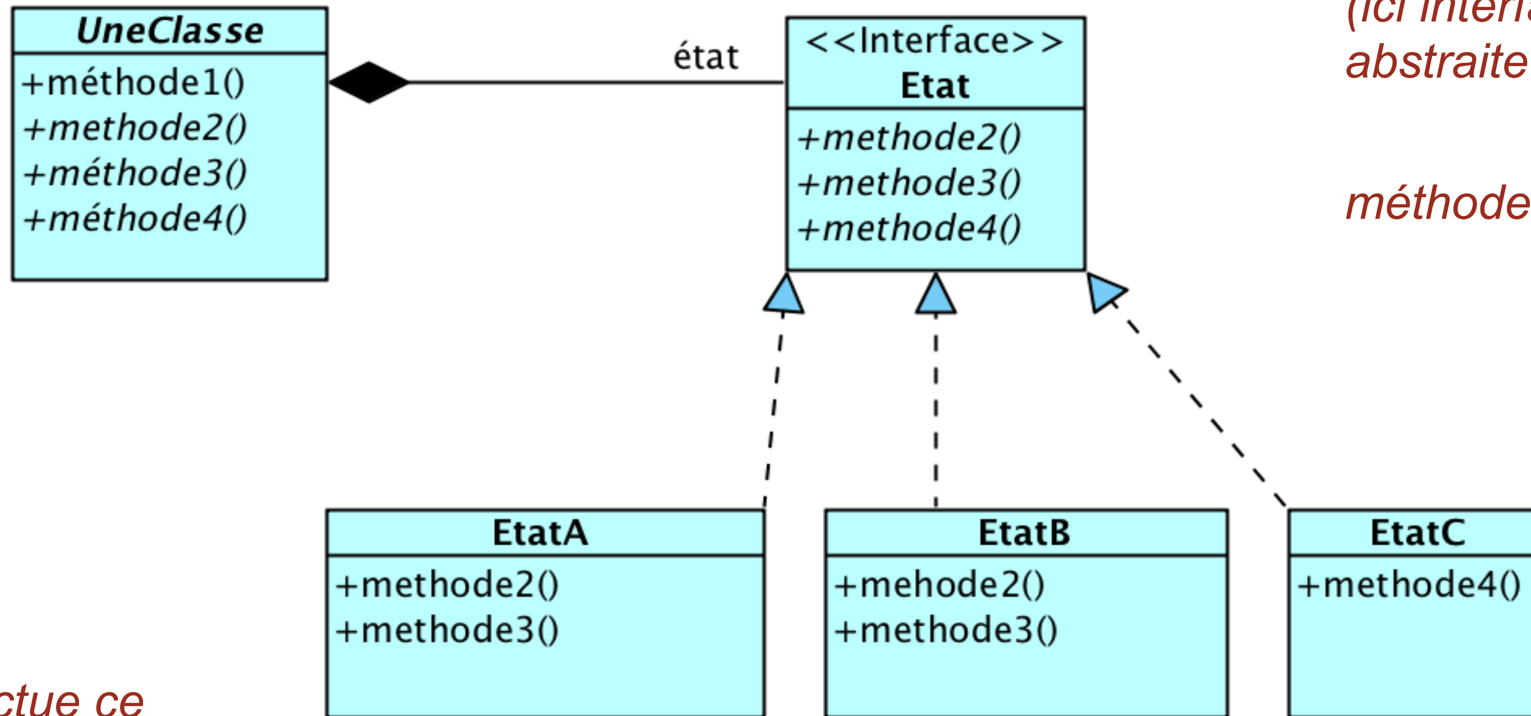
- L'idée serait de gérer ces différentes étapes du traitement de la commande, **indépendamment**.

- 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)

- Chaque classe Etat mentionne aussi **l'état suivant**, une fois le traitement effectué : permet de **modéliser le processus** (**workflow**)

DP State

*La classe de contexte, qui décrit **tous les comportements possibles**, et possède **différents états** EtatA, EtatB…, où seul l'état courant est gardé*



(ici interface, mais classe abstraite possible)

méthodes abstraites

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

methode2() et methode3() :
//… traitement quand dans l'EtatA, puis :
uneClasse.**setEtat(new EtatB()**);

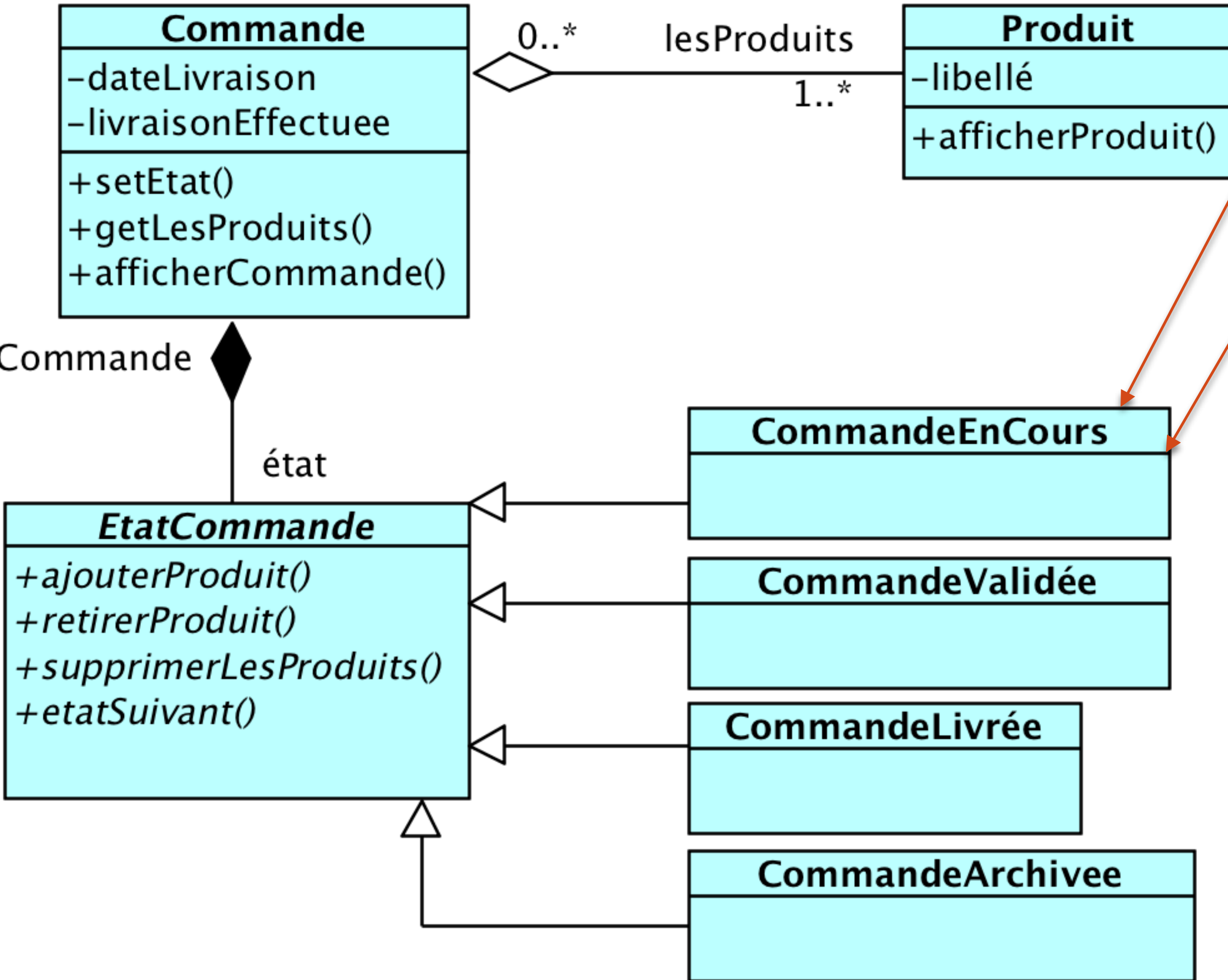
methode2() et methode3() :
//… traitement qd en EtatB, puis
uneClasse.**setEtat(new EtatC());**

Participants

- *UneClasse* : définit l'objet dont ont 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*…



Dans le constructeur de Commande, on définit l'état initial de la commande avec la méthode **setEtat(new CommandeEnCours())**

CommandeEnCours implémente :
ajouterProduit(), supprimerLesProduits(), retirerProduit(), étatSuivant()

par ex.:

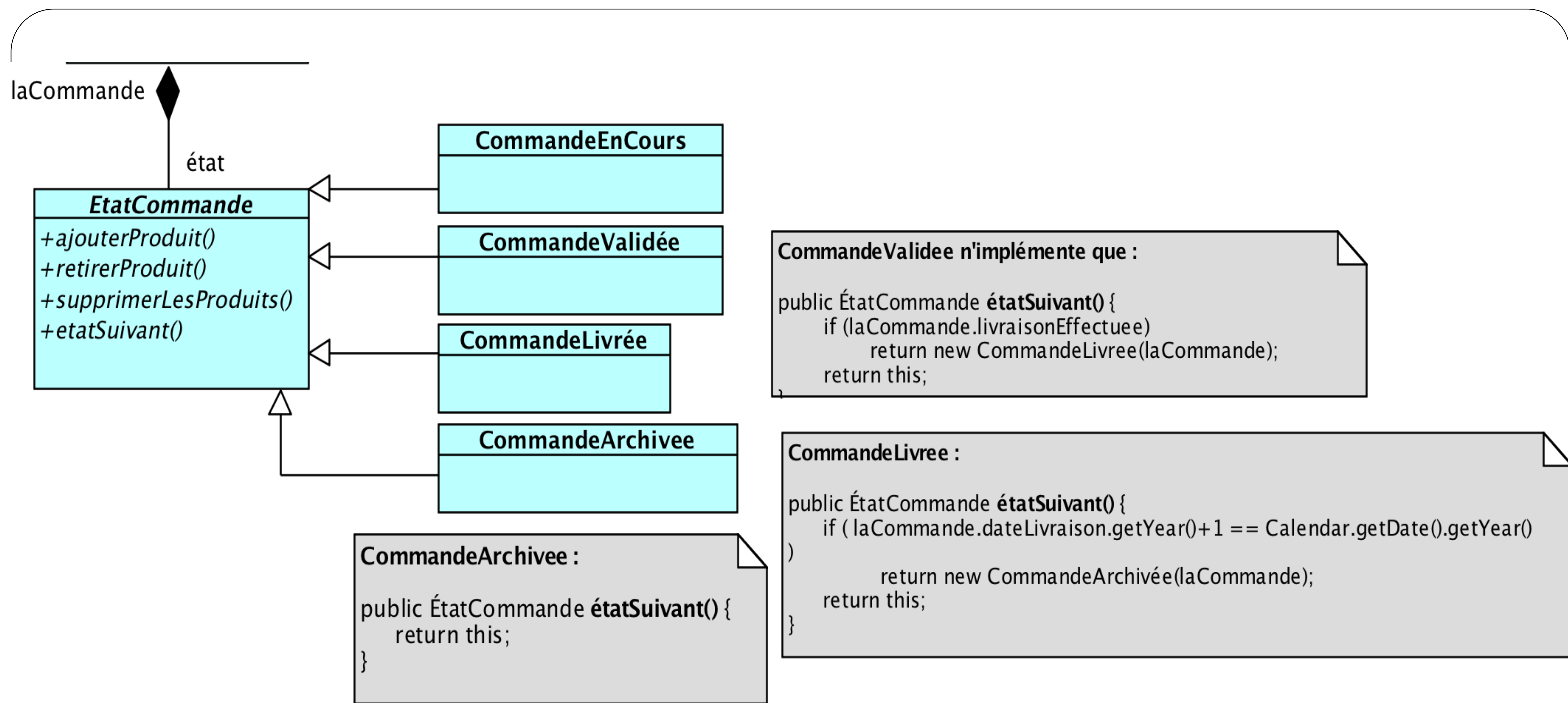
```
public ÉtatCommande étatSuivant() {

    return new CommandeValidée(laCommande);
}
```

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

Exemple : Commande de Produit

© V. Deslandres - IUT Lyon1

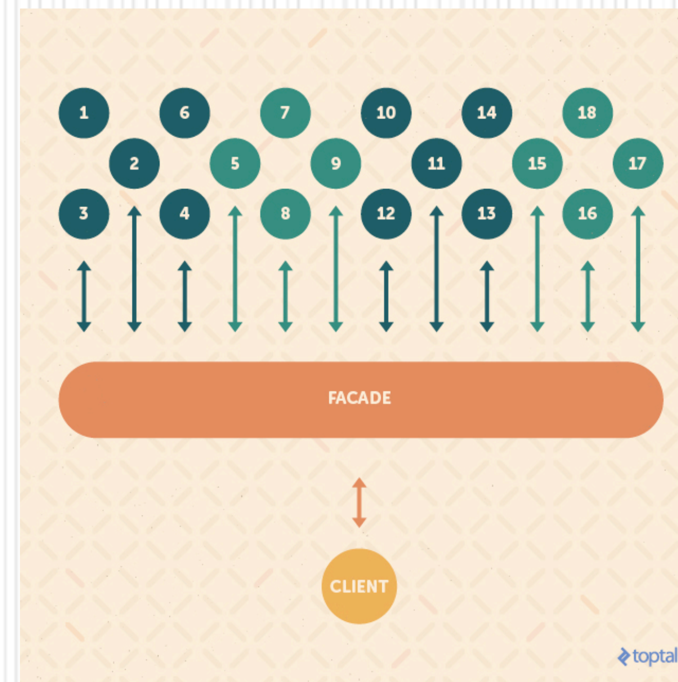


Cf TD

© V. Deslandres - IUT Lyon1

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 certains 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 Cliente
  - Soit on veut limiter l'accès à une partie du sous système

Ex. un Standard

Le standard masque la complexité de l'organisation, le client qui appelle n'en connaît pas la structure

Service Commercial

Service Approvisionnement

Comptabilité

L'existant

Client

Facade

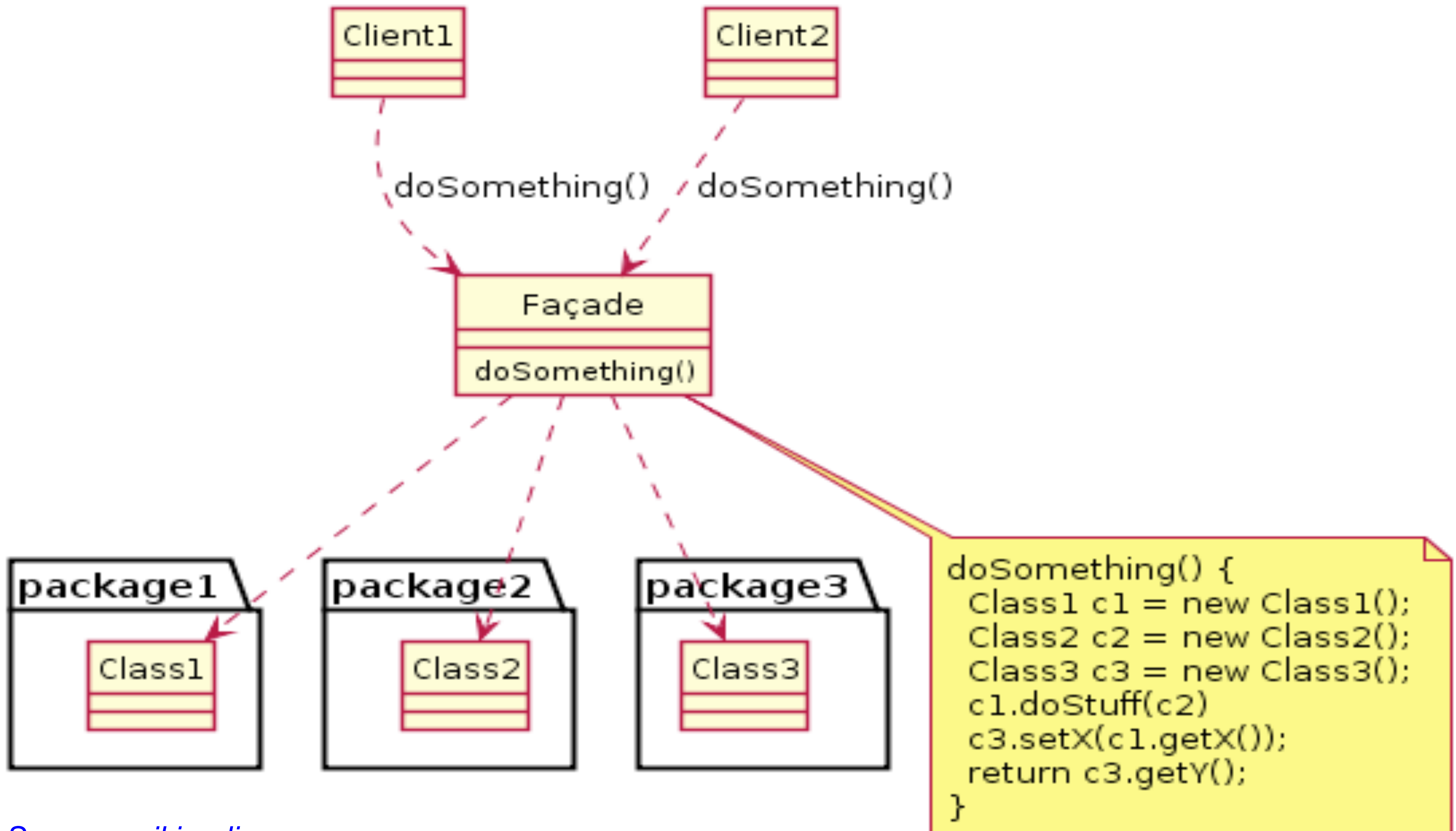
+ operation2()
+ operation4()

ClasseA

+operation1()
+operation2()

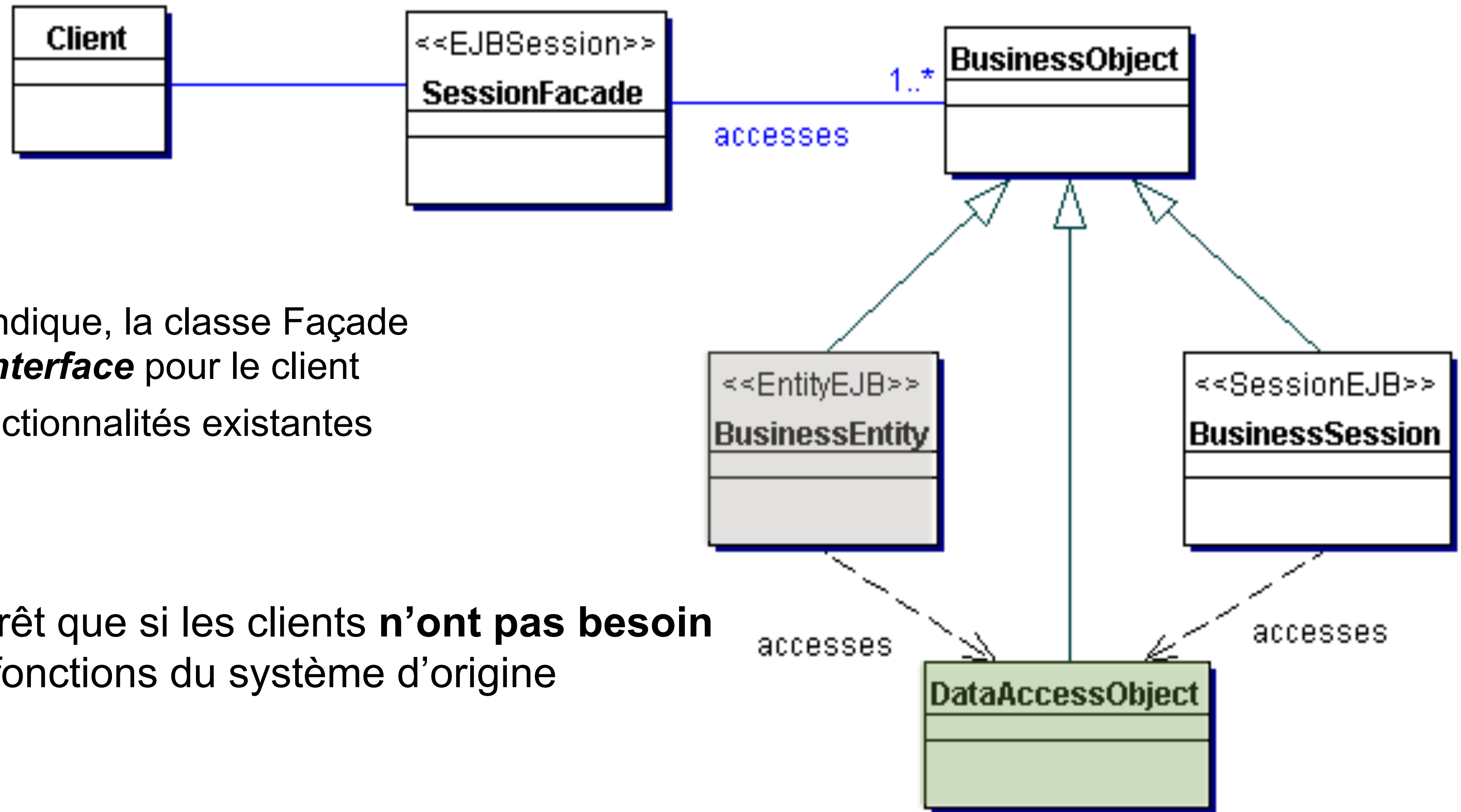
ClasseB

+operation3()
+operation4()



Source : wikipedia

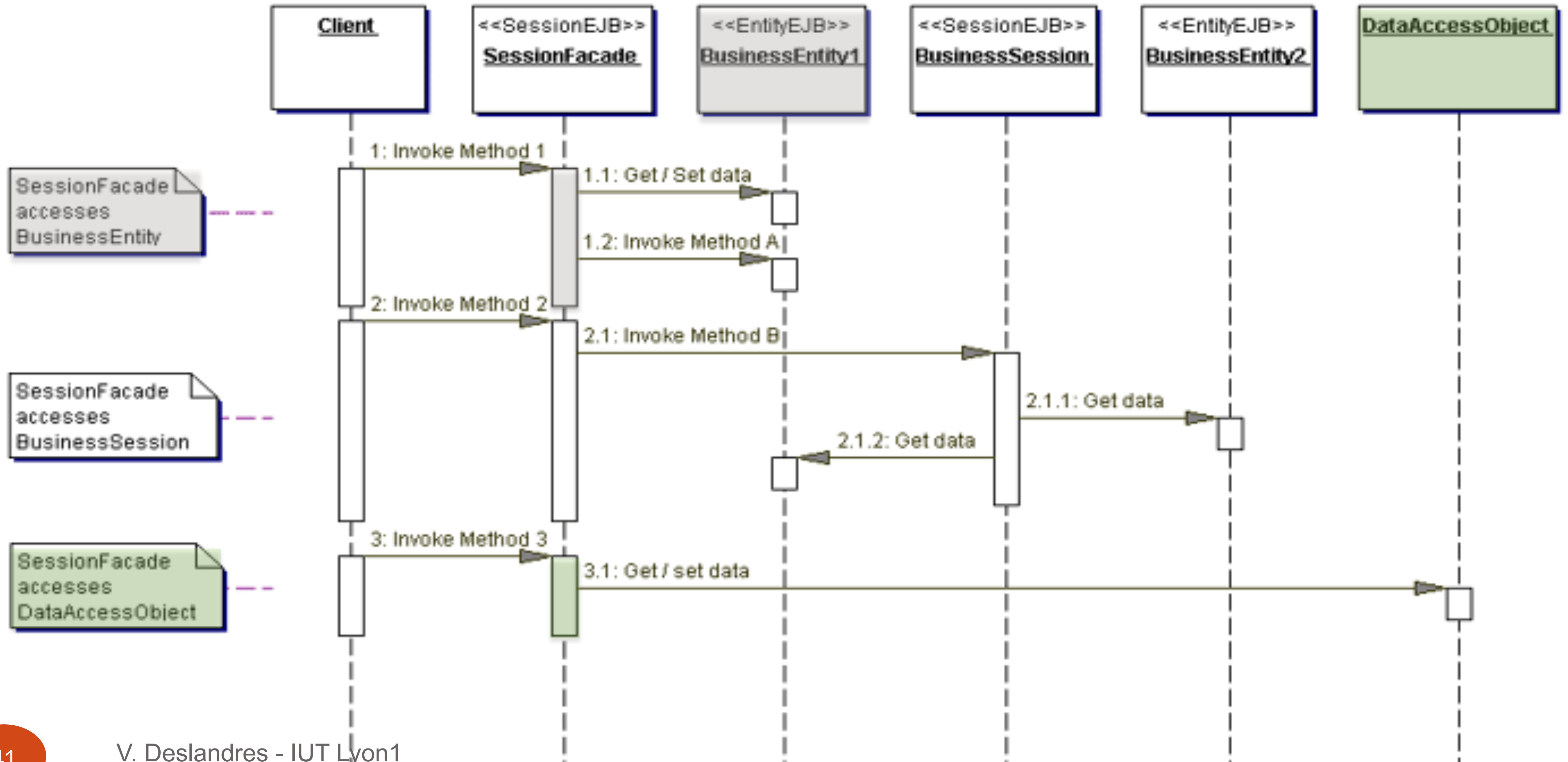
Ex. Pattern Façade : accès aux services et données avec JEE



- Comme son nom l'indique, la classe Façade offre une nouvelle ***interface*** pour le client
- Élaborée sur les fonctionnalités existantes

Ce pattern n'a d'intérêt que si les clients **n'ont pas besoin** d'utiliser **toutes** les fonctions du système d'origine

Pattern Façade appliqué à JEE (suite)



```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 /* careful ! */, 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 TestFacadePattern {
    public static void main(String[] args) {
        UserfriendlyDate d = new UserfriendlyDate("2018-08-20");
        System.out.println("Date : "+d);
        d.addDays(20);
        System.out.println("20 jours après : "+d);
    }
}
```

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

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

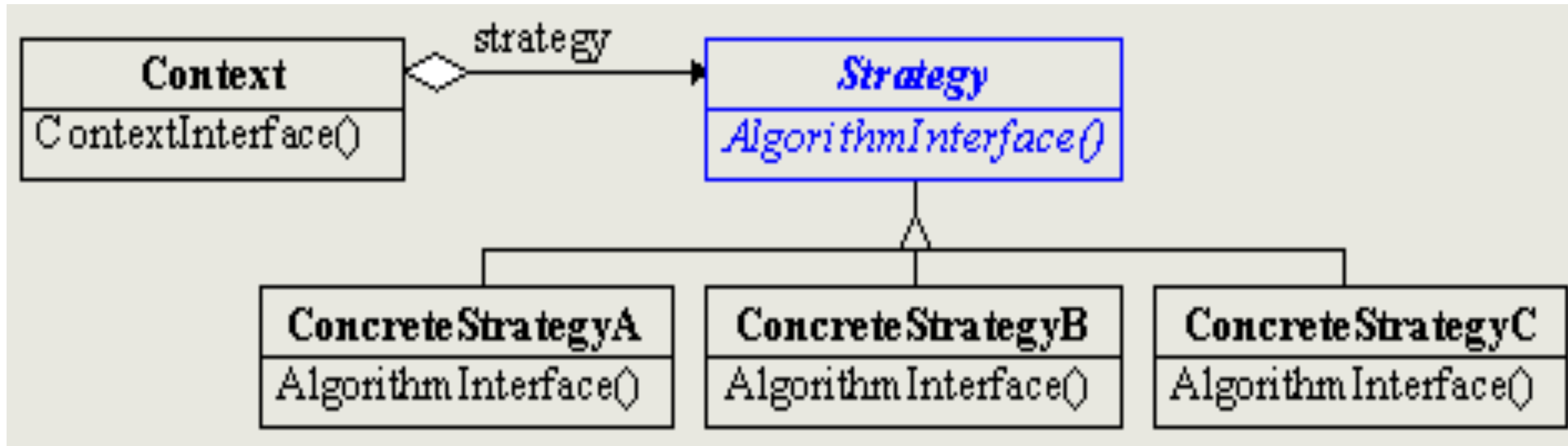
- **Conséquence** : le pattern définit une famille d'algorithmes, définit des classes de réalisation indépendantes, les rend dynamiquement interchangeables.
  - On peut ajouter / supprimer des algorithmes
  - Il est possible d'échanger dynamiquement d'algorithme sans modifier les classes clients qui 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 qui sont autant de **branches conditionnelles** dans ses méthodes
  - Swith/case
  - If imbriqués

Architecture STRATEGY (à voir en TD)



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

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

(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

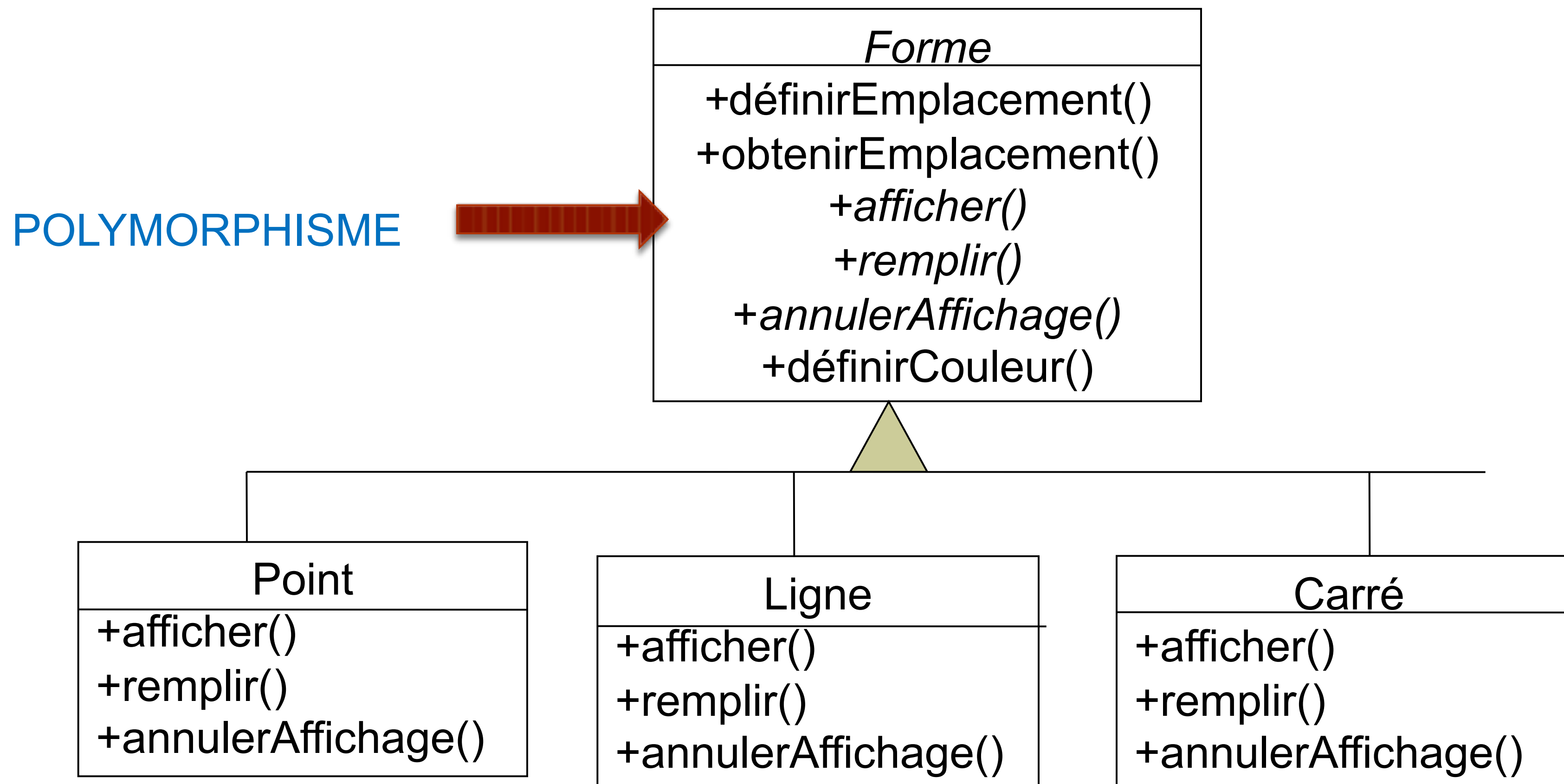
- Il consiste à transformer
  - par **délégation**
- les points d'entrée d'un composant
  - que l'on désire intégrer
  - à l'interface souhaitée par le concepteur

ADAPTER

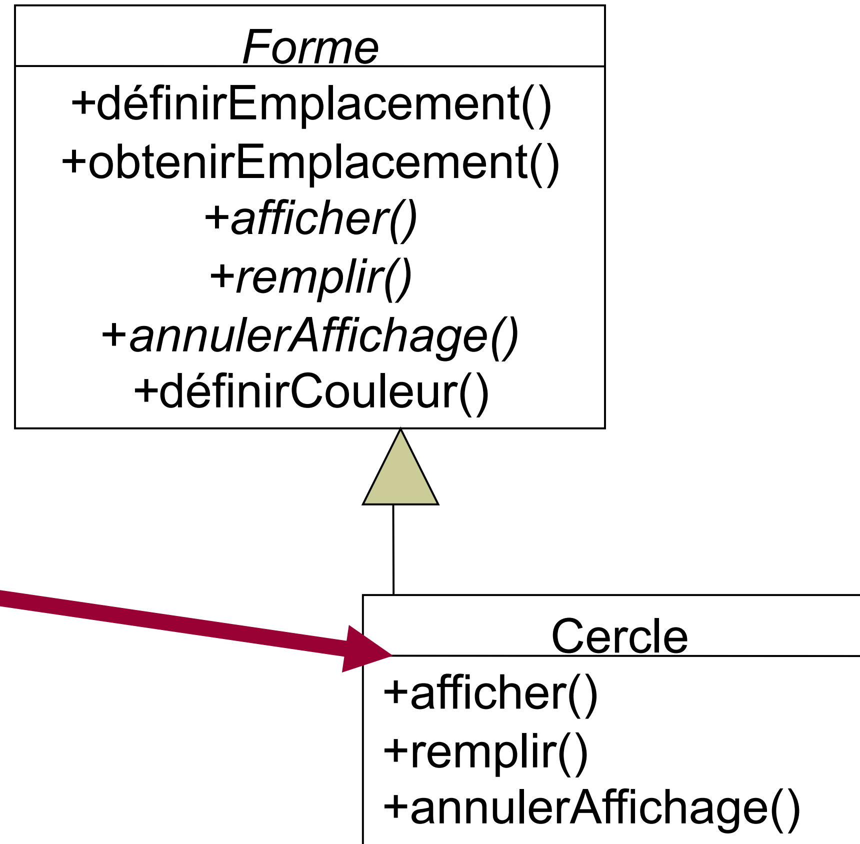
- Objectif : transforme l'interface d'une classe en **une autre interface souhaitée**
  - conforme à ce qu'attendent les classes clientes
- Permet à des classes de collaborer
  - qui n'auraient pu le faire du fait d'interfaces incompatibles
- Ex.: on dispose de classes **Point, Ligne, Carré**
  - ayant 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

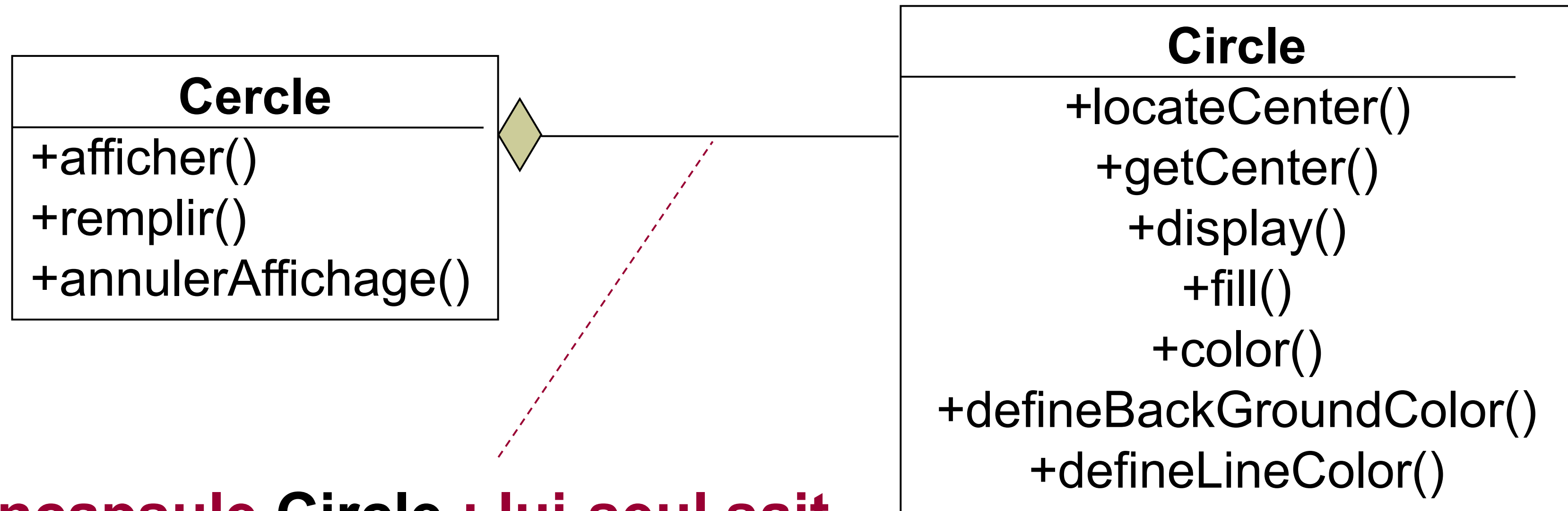
- Supposons que l'on ait déjà une classe **Circle** existante

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

Circle
+locateCenter()
+getCenter()
+display()
+fill()
+color()
+defineBackGroundColor()
+defineLineColor()

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



ADAPTER

Extrait du code Java correspondant :

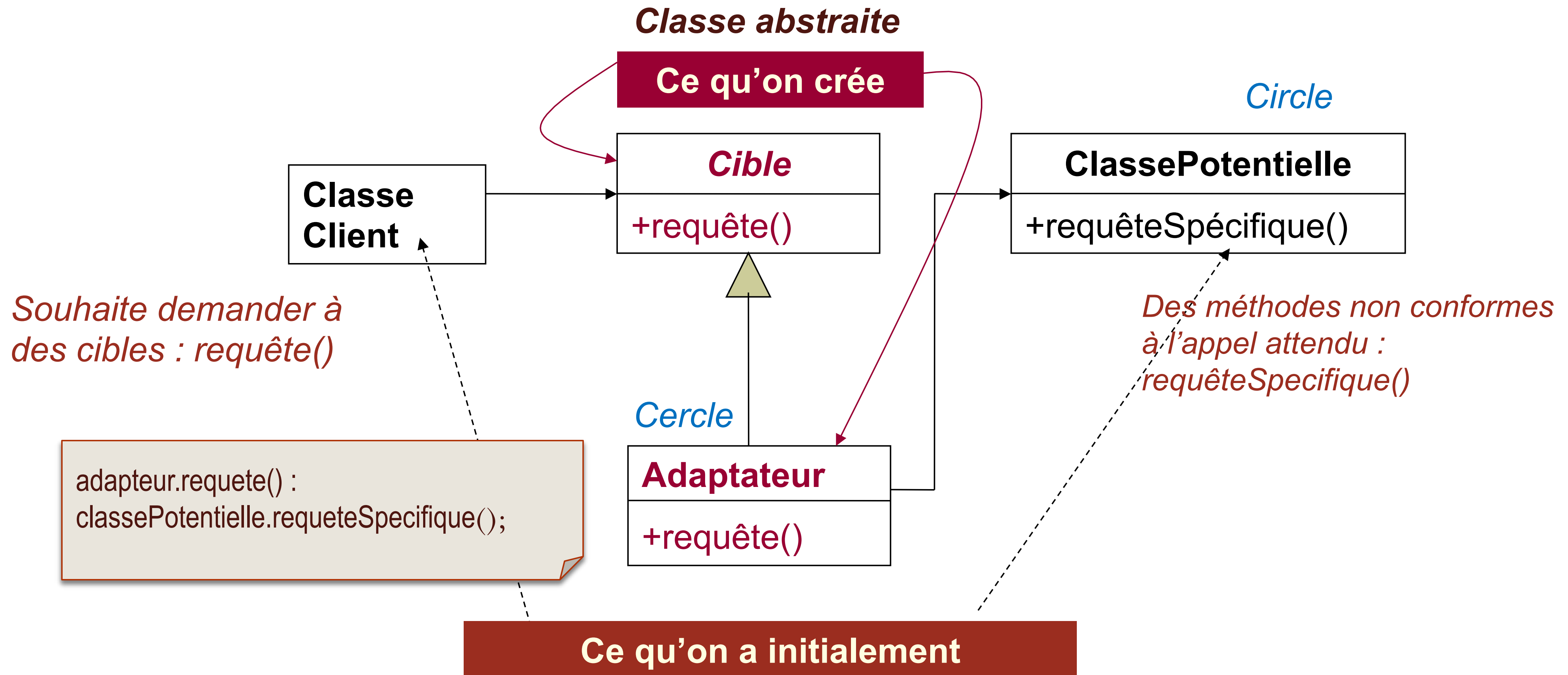
```
Class Cercle extends Forme {
    …
    private Circle leCercle;
    …
    public Cercle() {
      leCercle = new Circle();
    }
    void public afficher() {
      leCercle.display();
    }
    void public remplir() {
      leCercle.fill();
     }

}
```

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 externe
- **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 CercleXX existe