

# 10. Design Patterns

## 10.1. Strategy Pattern

## Objectif du pattern stratégie

Permettre à une partie du système de varier indépendamment des autres parties



Encapsuler ce qui est susceptible de varier (encapsulation d'algorithmes)

**Extraire le comportement susceptible de varier:**

- Le placer dans des **interfaces** + classes qui implémentent ces interfaces
- Préférer la composition (lien a-un) à l'héritage (lien est-un)

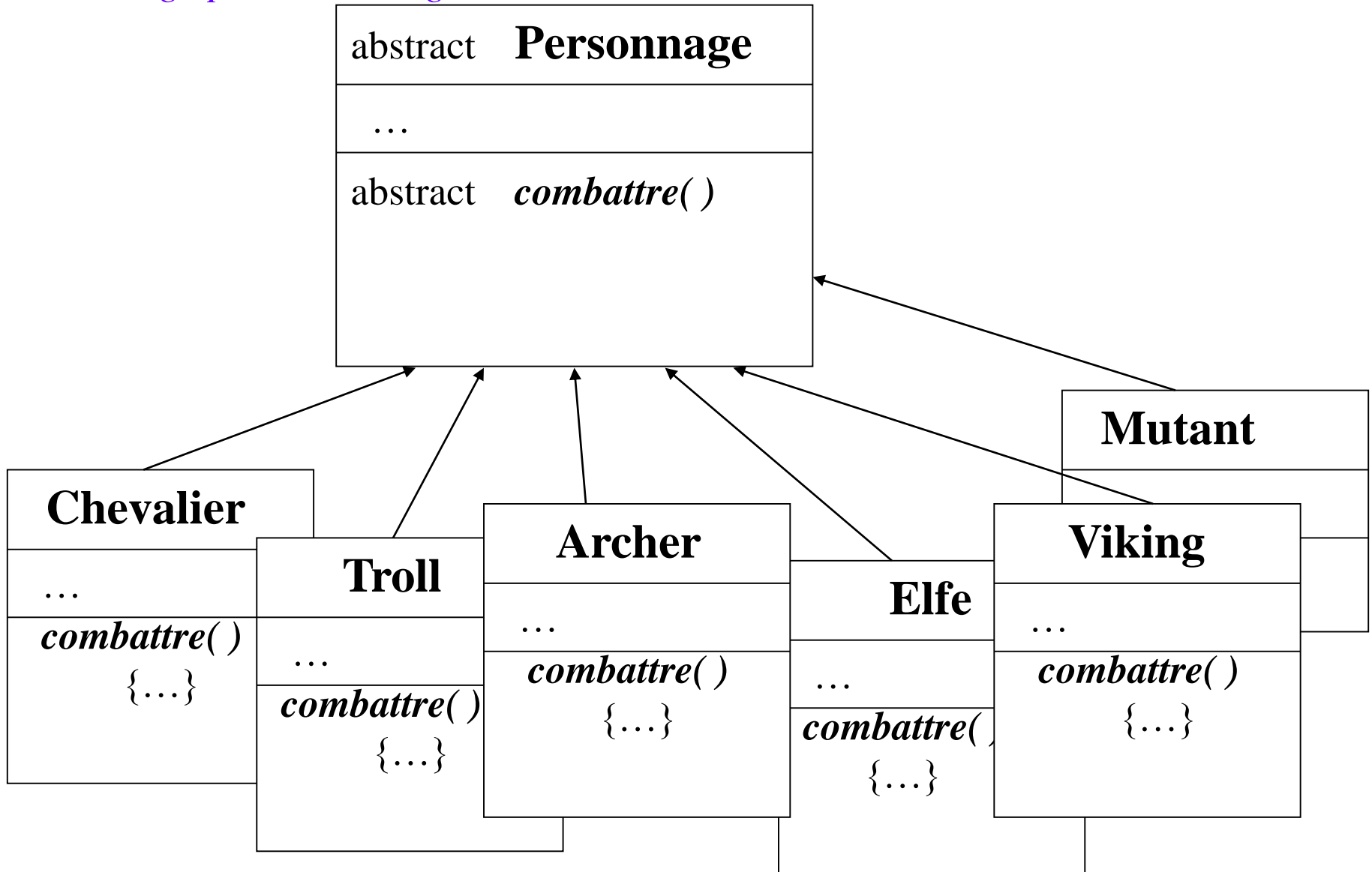
Rappel:

lien is-a (est-un):

lien has-a (a-un):

lien implements:

*Sans design pattern stratégie:*



*Or, plusieurs personnages partagent le même comportement de combat (mêmes armes) : certains utilisent l'épée, d'autres l'arc et les flèches, ou encore la hache, le fusil, ...*

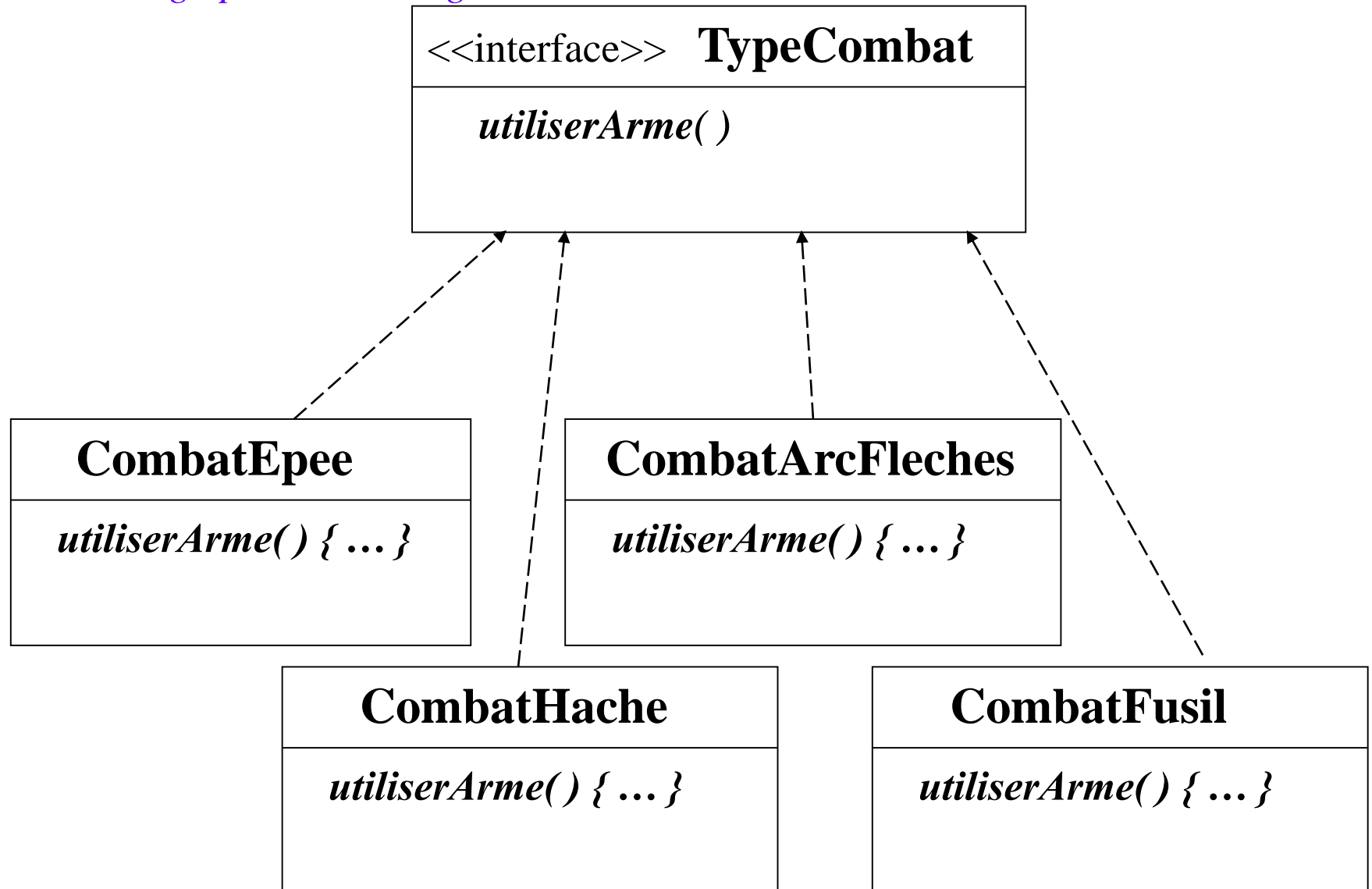
Le même comportement sera donc implémenté plusieurs fois.

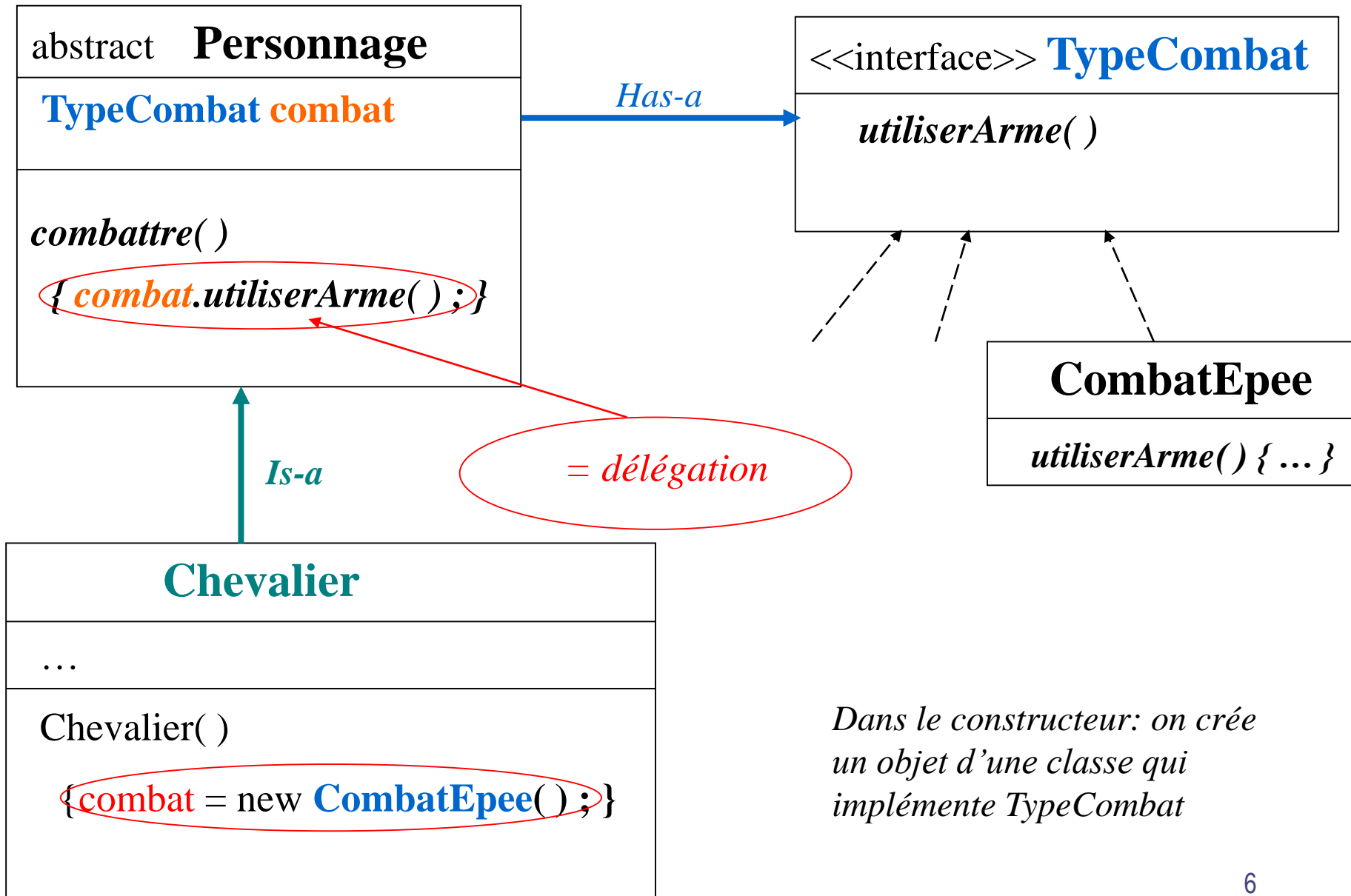
### Conclusion:

extraire le comportement de combat dans une interface et dans des classes qui implémentent cet interface: une classe par type de combat (type d'arme)

+ prévoir un lien entre Personnage et interface TypeCombat

Avec design pattern stratégie:





# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern

## Objectif du pattern *fabrique*

Encapsuler l'instanciation de classes (la création d'objets)

### Exemple:

Une pizzeria (créateur)

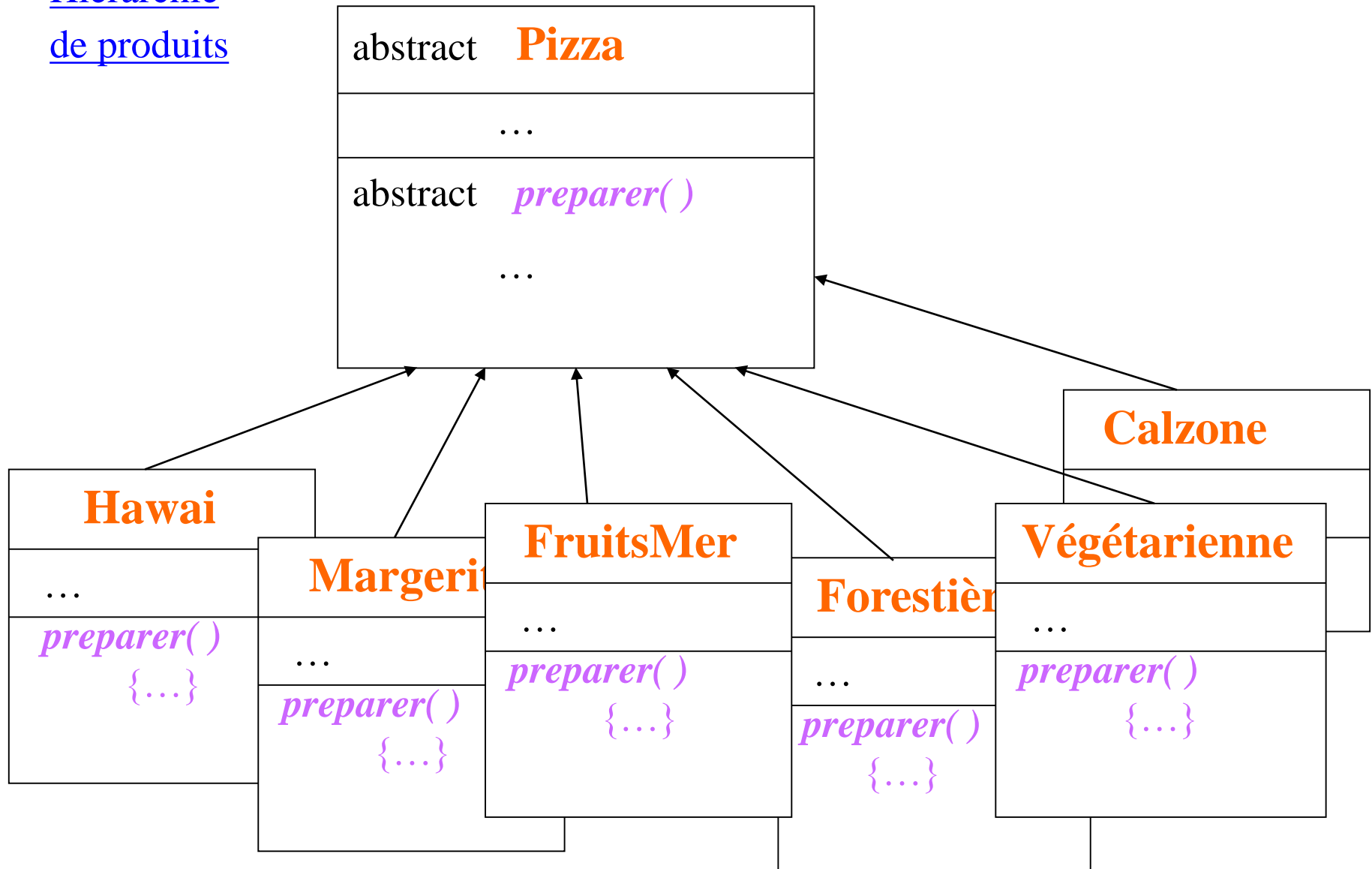
qui manipule des objets de type Pizza (produits)



## Produit

<i>abstract</i> class <b>Pizza</b>
<i>nom</i> <i>pate</i> <i>sauce</i> <i>garniture [ ]</i>
<i>abstract</i> <b>preparer( )</b> <i>cuire ( )</i> <i>couper ( )</i> <i>emballer ( )</i>

## Hiérarchie de produits



Exemple:

```
public class Hawai extends Pizza
{   public void preparer( )
    {   nom = "pizza Hawai";
        pate = "pâte fine";
        sauce = "sauce tomatée";
        garniture[0] = "jambon";
        garniture[1] = "mozzarella";
        garniture[2] = "ananas";
    }
}
```

## Créateur de produits

class **Pizzeria**

Pizza commanderPizza( type)

{

}

Créer une pizza  
en fonction du  
type demandé



```
public class Pizzeria
```

```
{ ...
```

```
    public Pizza commanderPizza (String type)
```

```
    { Pizza pizza;
```

```
        ...
```

*Créer la pizza en fonction du type*

```
        pizza.preparer( );
```

```
        pizza.cuire( );
```

```
        pizza.couper( );
```

```
        pizza.emballer( );
```

```
        return pizza;
```

```
    }
```

```
public class Pizzeria
{
    ...

    public Pizza commanderPizza (String type)
    {
        Pizza pizza;

        if (type.equals("Hawai") )
            pizza = new Hawai( );

        else if (type.equals("Calzone") )
            pizza = new Calzone( );

        else if (type.equals("Fruits de mer") )
            pizza = new FruitsMer( );

        else if (type.equals("Végétarienne") )
            pizza = new Vegetarienne( );

        ...

        return pizza; } }
```

## Problèmes:

Si plusieurs endroits où il faut créer des pizzas: il faut dupliquer ce code

Si nouveau type de pizza, il faut modifier le code plusieurs fois

⇒ Difficile à maintenir !

## Version 1: Principe de la *fabrique simple*:

### *Extraire le code de création des objets du code du créateur*

⇒ Découplage du code de création du produit – code du créateur

⇒ Création d'une classe Fabrique de produits

⇒ Le créateur de produits **délègue à la fabrique de produits le soin de créer les produits**

## Fabrique de produits

```
class FabriqueDePizzas
```

```
    Pizza creerPizza( type)
```

```
{
```

```
}
```

Créer une pizza





```
public class FabriqueDePizzas
{
    ...

    public Pizza creerPizza (String type)
    {
        Pizza pizza;

        if (type.equals("Hawai") )
            pizza = new Hawai( );

        else if (type.equals("Calzone") )
            pizza = new Calzone( );

        else if (type.equals("Fruits de mer") )
            pizza = new FruitsMer( );

        else if (type.equals("Végétarienne") )
            pizza = new Vegetarienne( );

        ...

        return pizza; } }
}
```

## Pizzeria

**FabriqueDePizzas** *fabrique*

Pizza commanderPizza(*type*)

```
{ Pizza pizza;
```

```
  pizza = fabrique.creerPizza(type)
```

```
  ...
```

```
  return pizza;
```

```
}
```

*Has-a*

## FabriqueDePizzas

*creerPizza(type)*

*= délégation*

```
public class Pizzeria
{
    private FabriqueDePizzas fabrique;

    public Pizzeria (FabriqueDePizzas fabrique)
    {
        this.fabrique = fabrique;
    }

    public Pizza commanderPizza (String type)
    {
        Pizza pizza;

        pizza = fabrique.creerPizza(type);

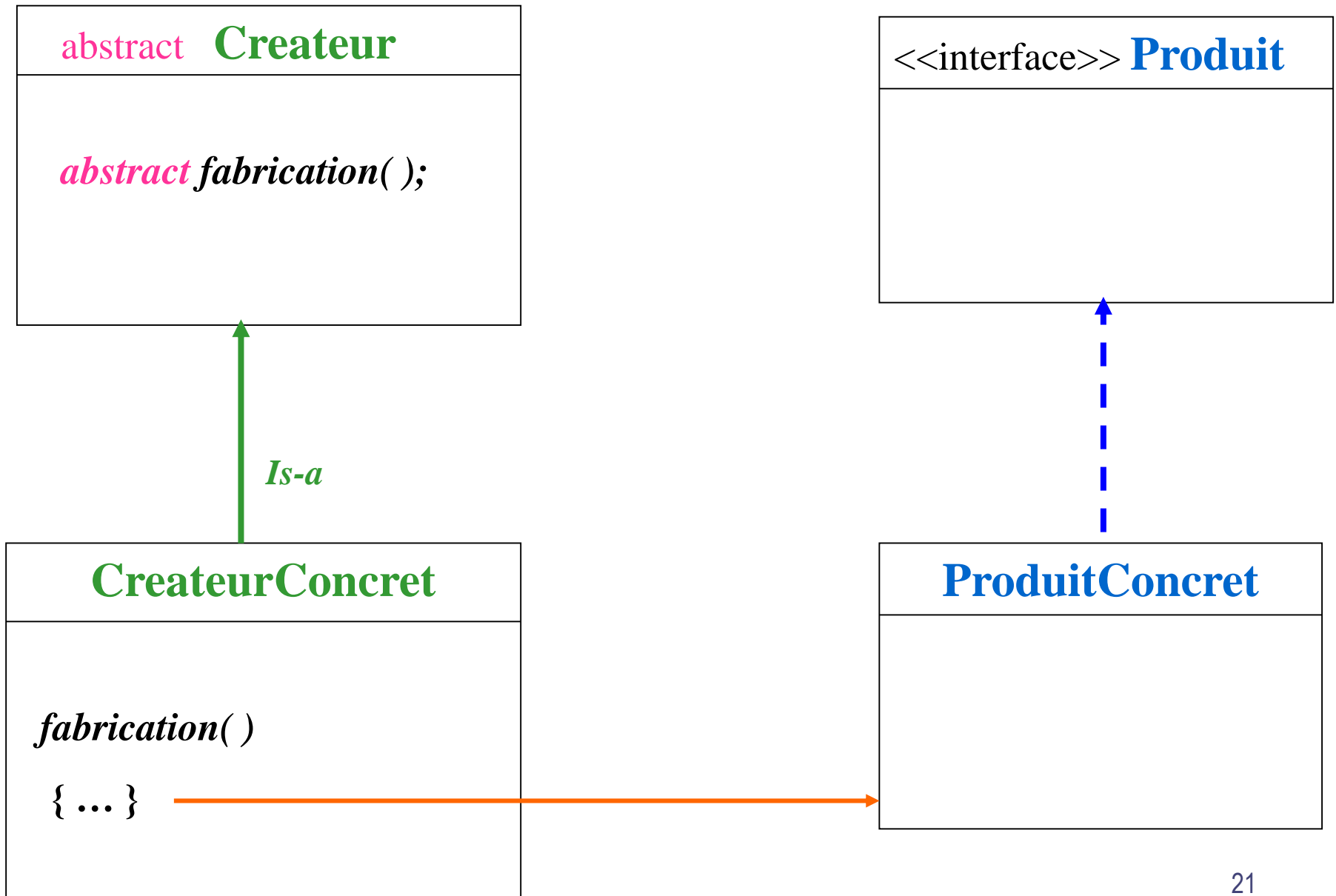
        pizza.preparer( );
        pizza.cuire( );
        pizza.couper( );
        pizza.emballer( );
    }
}
```

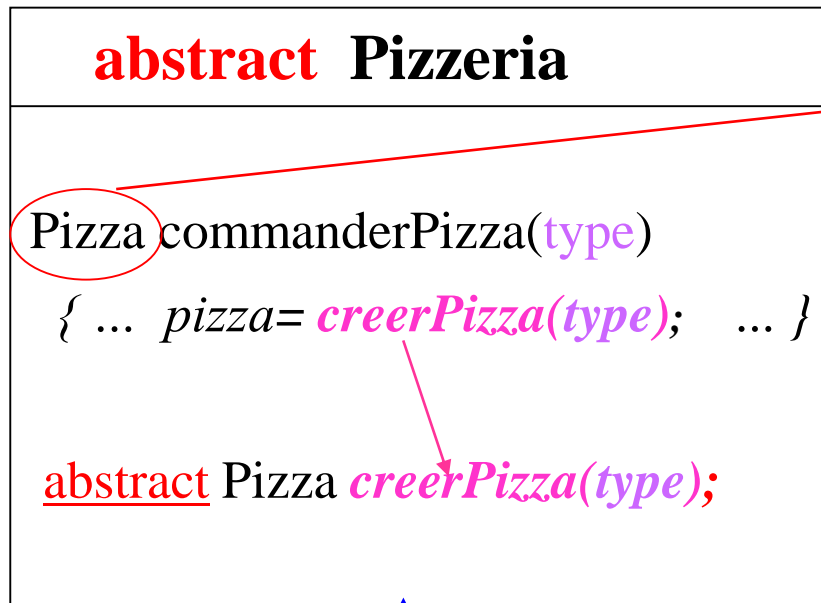


*Pour créer la pizza en fonction du type:  
délégation à la fabrique*

## Version 2: *Pattern Fabrication*

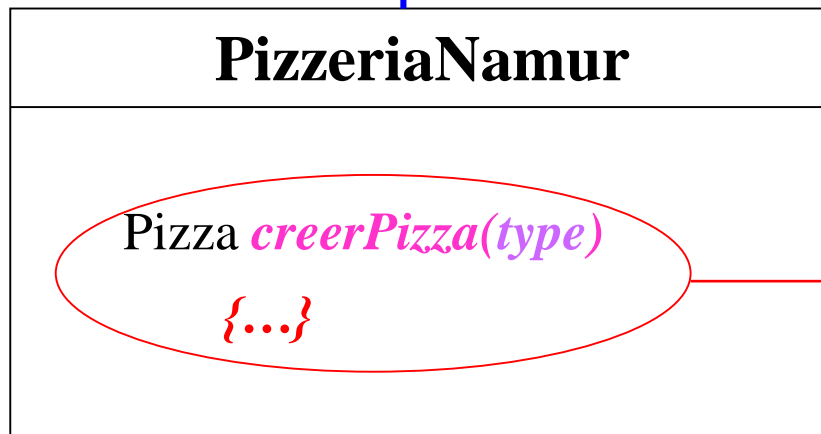
*Une classe créateur abstraite qui délègue l'instanciation des objets produits à ses sous-classes*





*La super-classe manipule des objets abstraits: Pizza est une classe abstraite*

Is-a



*Le code de la création des produits est délégué à la sous-classe: la sous-classe créera des produits concrets (sous-classes de Pizza)*

```
public abstract class Pizzeria
{
    public Pizza commanderPizza (String type)
    {
        Pizza pizza;
        pizza = creerPizza(type);
        pizza.preparer( );
        pizza.cuire( );
        pizza.couper( );
        pizza.emballer( );
    }
}

public abstract Pizza creerPizza(String type);
```

# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern

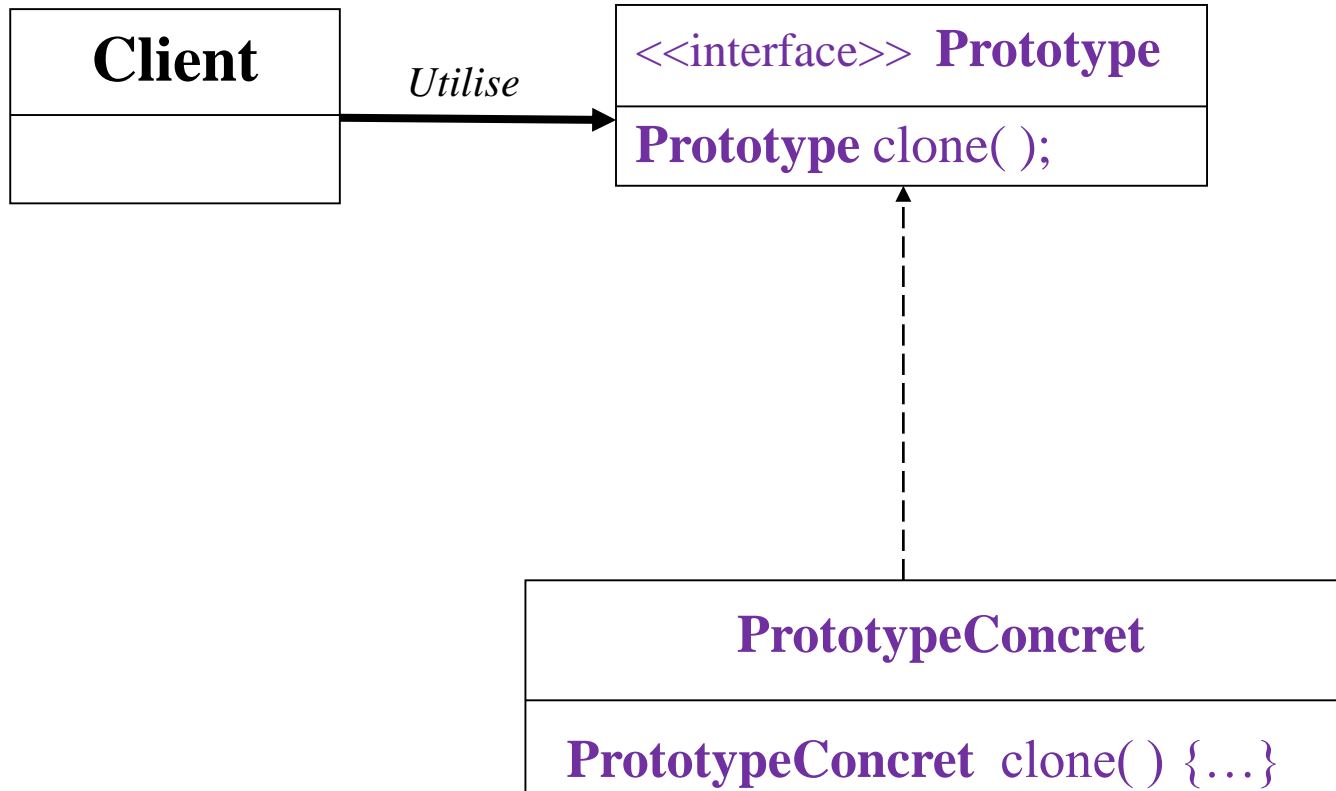


Objectif du pattern **prototype**

Créer des objets sur base d'une instance prototype

⇒ Créer des nouveaux objets en copiant cet objet prototype

⇒ Prévoir la méthode **clone( )** qui crée un objet de la même classe



```
public interface Prototype {  
    Prototype clone( );  
}
```

```
public class Rectangle implements Prototype{  
    private int largeur, hauteur;  
    private String couleur, texture, texte;  
  
    public Rectangle (int largeur, int hauteur, String couleur, String texture, String texte)  
    { this.largeur = largeur; this.hauteur = hauteur; this.couleur = couleur;  
      this.texture = texture; this.texte = texte; }  
  
    public Rectangle clone( )  
    { return new Rectangle (largeur, hauteur, couleur, texture, texte);}  
}
```

```
public class PrototypeDesignPattern {  
  
    public static void main(String[ ] args) {  
  
        Rectangle rectangleModele =  
            new Rectangle( 10,5,"rouge","hachuré","Rectangle type" );  
  
        Rectangle copieRectangle = rectangleModele.clone( ) ;  
  
    }  
}
```

# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern

Objectif du pattern singleton:

Garantir qu'une classe n'a qu'une seule instance

Comment créer un **objet unique** (une seule instance d'une classe)?



**Variable de classe privée** (private static)


+

**Constructeur privé** (private)

+

Méthode (static) **getInstance()** qui retourne l'unique instance

```
public class MySingleton {  
    private static MySingleton uniqueInstance;  
  
    // autres variables d'instance  
  
    private MySingleton (...) {...}  
  
    public static MySingleton getInstance() {  
        if (uniqueInstance == null)  
            { ...  
                uniqueInstance = new MySingleton(...); }  
        return uniqueInstance;  
    }  
  
    // autres méthodes  
}
```



## Adaptation:

```
public class MyClass {  
    ...  
    private static ClassY uniqueInstance;  
    public static ClassY getInstance( ) {  
        if (uniqueInstance == null)  
            { // créer une instance de ClassY  
            }  
        return uniqueInstance;  
    }  
  
    // autres méthodes  
}
```

The diagram illustrates the relationship between the two 'ClassY' identifiers in the code. A purple arrow points from the 'MyClass' identifier to the 'ClassY' identifier in the 'uniqueInstance' field declaration, with the label '2 classes différentes' (2 different classes). A pink arrow points from the 'ClassY' identifier in the 'uniqueInstance' field declaration to the 'ClassY' identifier in the 'getInstance' method signature, with the label 'Même classe' (Same class).



## Utilisation:

```
MySingleton sing = MySingleton.getInstance( ) ;
```

## Cas d'utilisation dans le travail de fin d'année:

### *stockage de l'objet Connection:*

on ne crée la connexion que quand c'est nécessaire;

+ on ne crée qu'une fois la connexion:

## User Interface

MainJFrame

NewBookPanel

AllBooksPanel

AllBooksModel

## Model

## Controller

ApplicationController

Book

## Business Logic

BookManager

AddBookException

AllBooksException

## Data Access

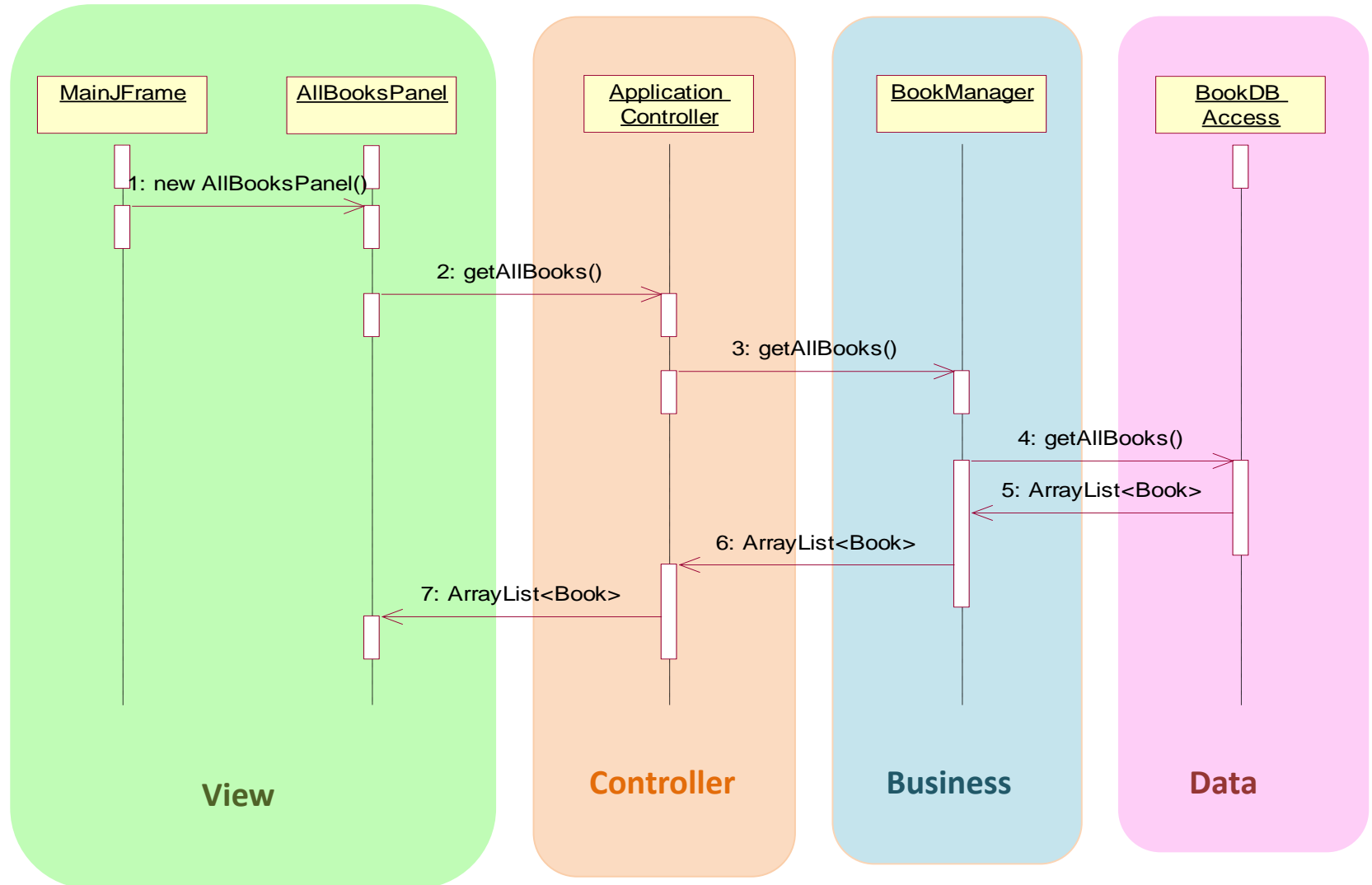
BookDBAccess

SingletonConnexion

### *Gestion de la connexion unique:*

```
public class SingletonConnexion {  
    private static Connection connexionUnique;  
  
    public static Connection getInstance( ) ...  
    { if (connexionUnique == null)  
        { // essayer de créer une connexion à la base de données }  
        return connexionUnique;  
    }  
  
}
```

# Afficher la liste de tous les livres



```
public class BookDBAccess {  
  
    public ArrayList <Book> getAllBooks( ) throws AllBooksException  
  
    {  
  
        // essayer d'accéder à la base de données  
  
        ↪ via SingletonConnexion.getInstance( )  
  
        // essayer de lire les livres dans la table Book  
  
        // créer et retourner une Array List de livres  
  
    }  
  
}
```

# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern
- 10.5. Data Access Object Pattern

## Objectif du pattern *Data Access Object* (DAO)

Séparer la persistance des données de l'accès logique aux données

⇒ indépendance du mécanisme de persistance



1. Encapsuler les accès aux données ⇒ les placer dans une interface

**= Interface public du DAO**

2. Créer des classes qui implémentent ces interfaces

**= Implémentations du DAO**

↳ connaissent la source de données à laquelle se connecter

(ex: BD, XML, Web Service, ...)

↳ spécifiques à une source de données

Le DAO joue le rôle d'**intermédiaire** entre l'application (business) et la couche persistance des données.

Le DAO **transfère des objets** entre la couche business et le stockage des données.

Le DAO **fournit des opérations** sur les données sans exposer les détails du stockage des données.



## Avantages

La logique business peut varier indépendamment de la persistance des données : il suffit d'utiliser la **même interface**.

La couche persistance peut varier : il suffit que **l'interface soit correctement implémentée**

⇒ **Réduit le couplage** entre la logique business et la logique persistance

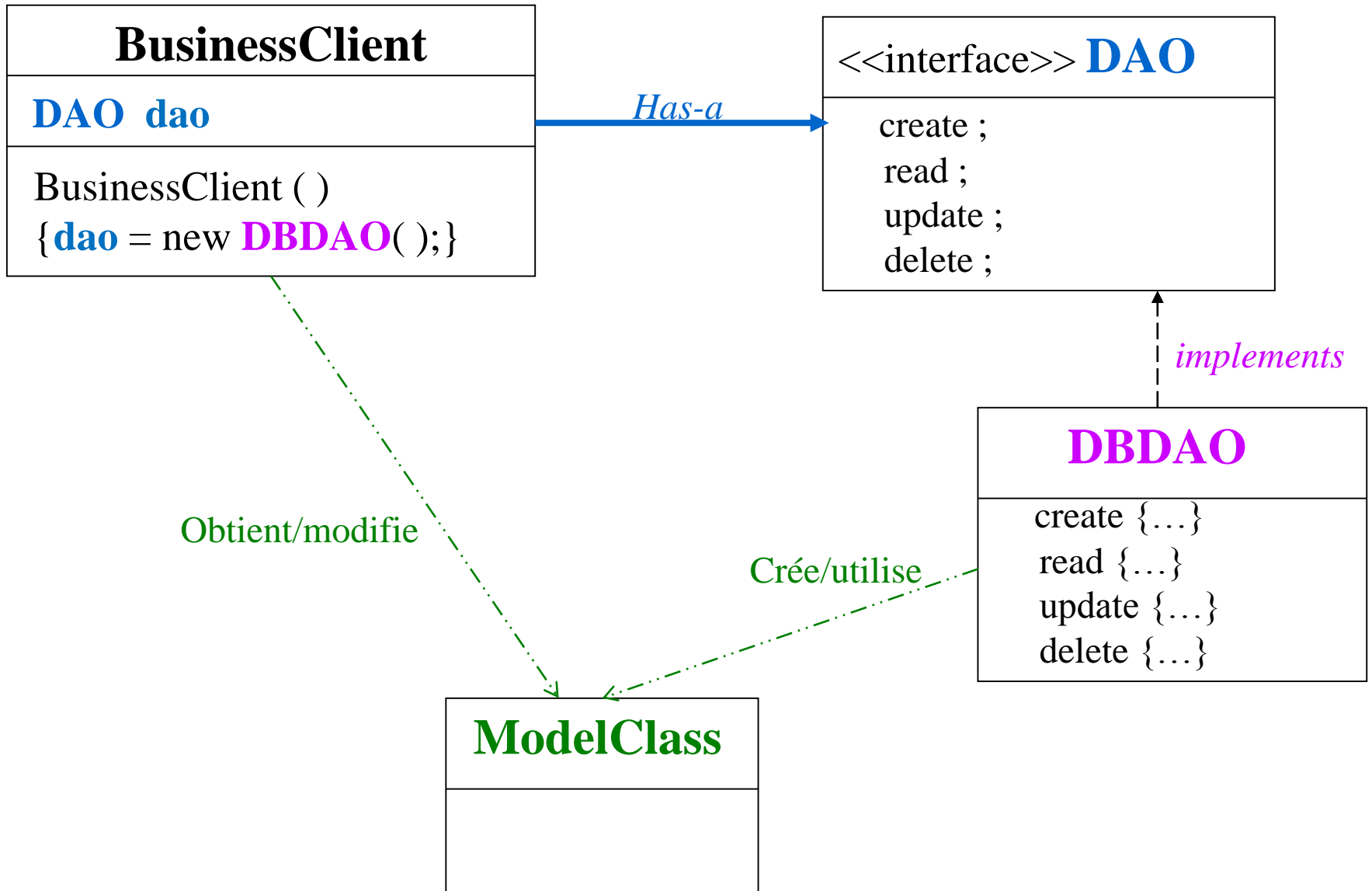
Via l'encapsulation du code des opérations **CRUD**

**C**reate

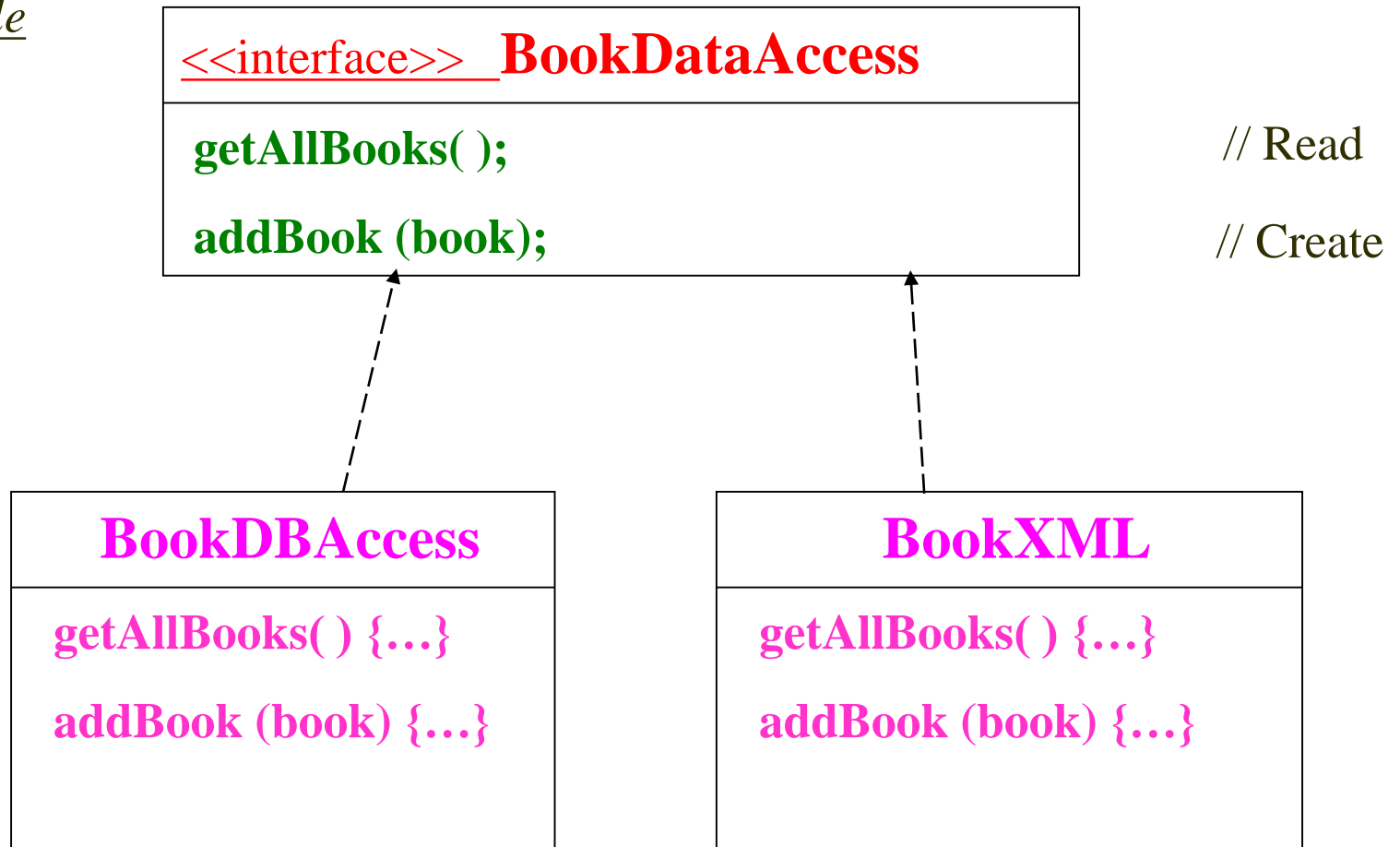
**R**ead

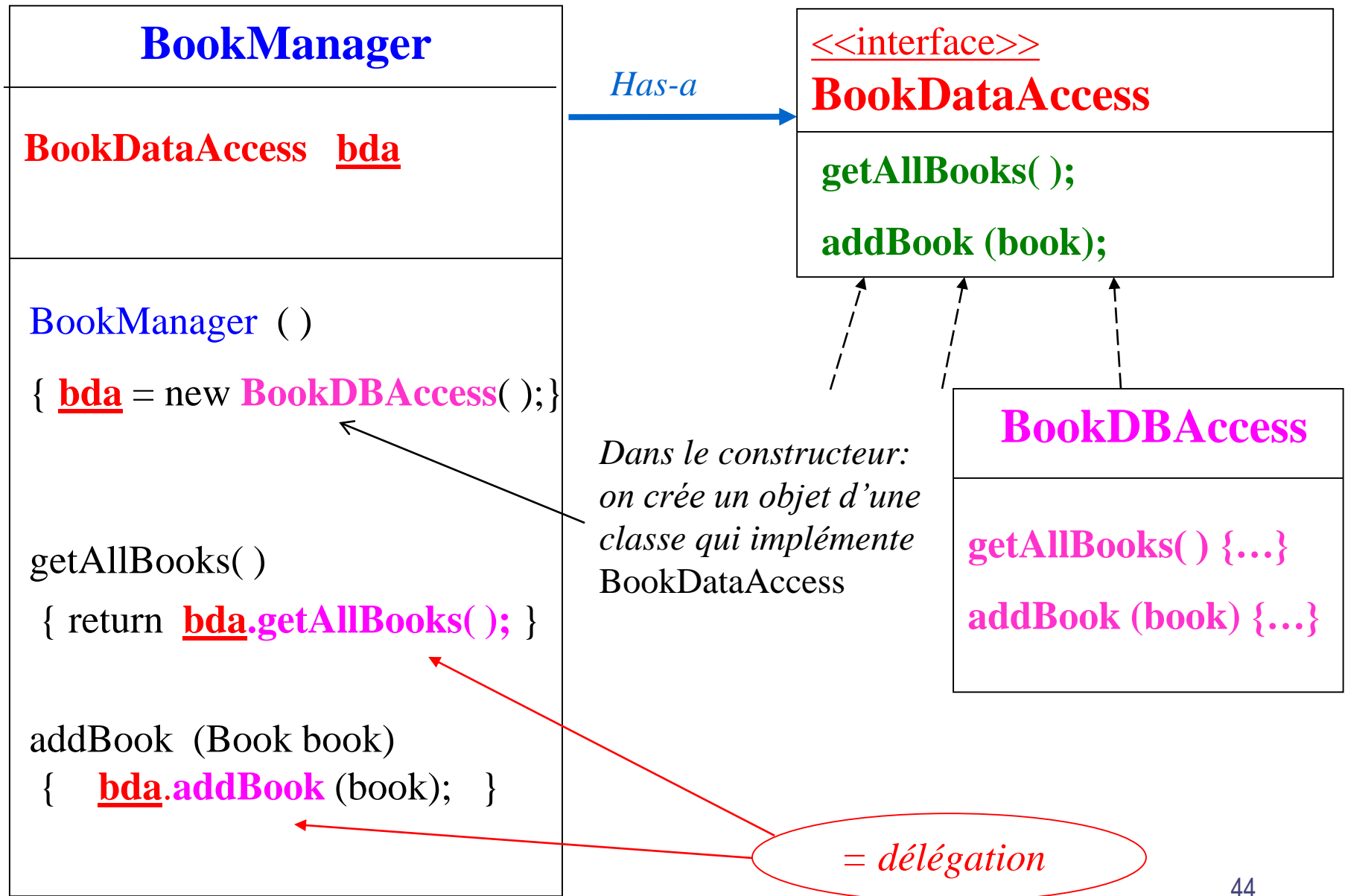
**U**ppdate

**D**elete



Example





# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern
- 10.5. Data Access Object Pattern
- 10.6. Iterator Pattern

## Objectif du pattern *itérateur*

Fournir un moyen d'accéder séquentiellement à une collection d'objets sans révéler son implémentation

But :

Boucler sur tous les éléments de la collection sans connaître son implémentation

Exemples d'implémentation de collection

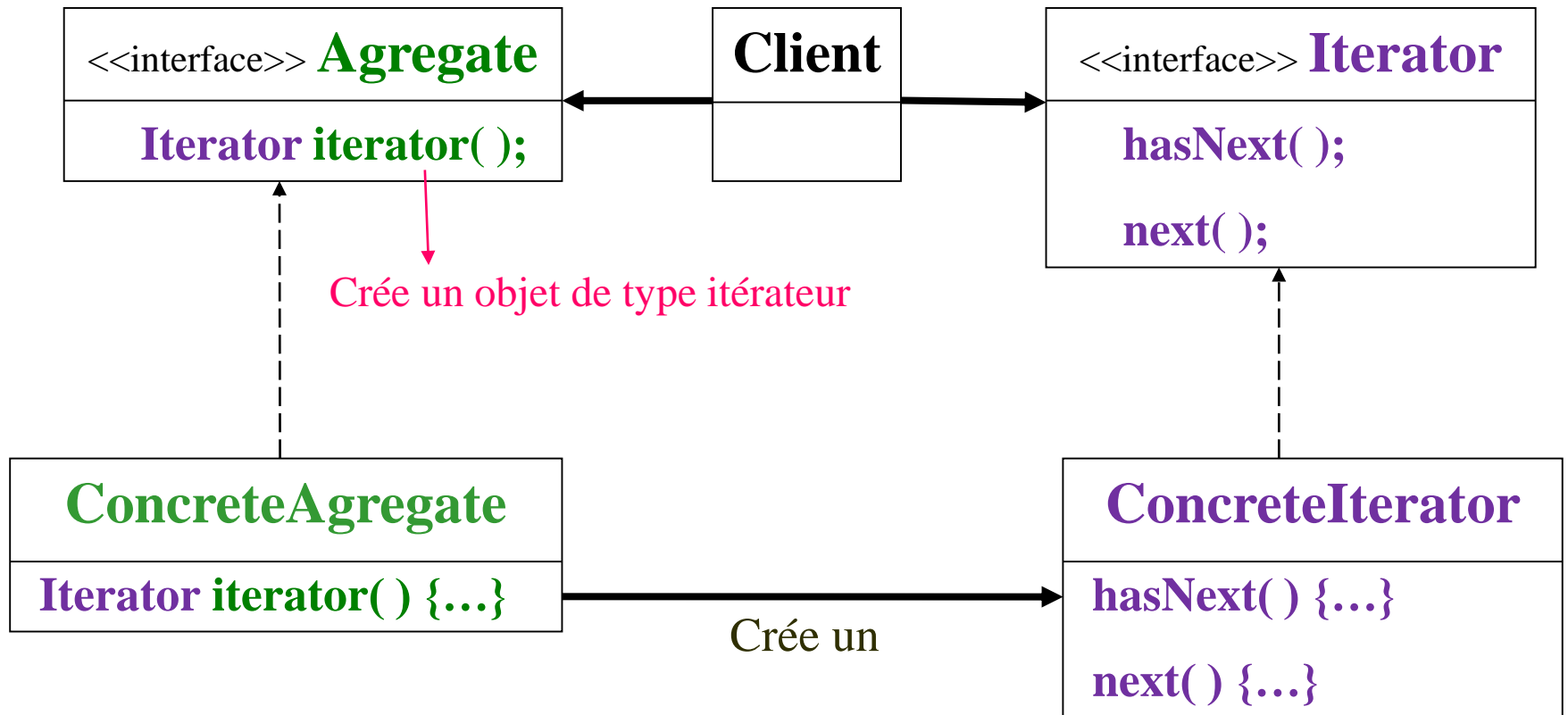
- Tableau d'objets
- Liste chaînée
- ArrayList
- HashMap

⇒ Il faut pouvoir

- demander l'élément **suivant**
- savoir s'il y a **encore** des éléments dans la collection

⇒ Utiliser un objet **itérateur** sur la collection

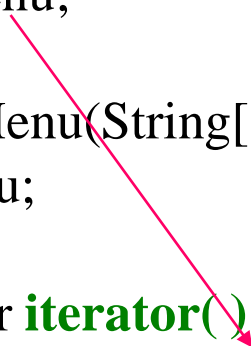
1. Créer un itérateur en lui fournissant la collection
2. Cet itérateur propose les méthodes
  - hasNext ⇒ vrai s'il existe encore au moins un élément dans la collection
  - next ⇒ retourne l'élément suivant de la collection





```
public interface Iterator {  
    Object next( );  
    boolean hasNext( );  
}
```

```
public class RestaurantMenu {  
    private String[ ] menu;  
  
    public RestaurantMenu(String[ ] menu) {  
        this.menu = menu;  
    }  
    public MenuItem iterator() {  
        return new MenuItem(menu);  
    }  
}
```



```
public class MenuItem implements Iterator{
    private String[ ] menu;
    private int position;

    public MenuItem(String[ ] menu) {
        this.menu = menu;
        position = 0;
    }
    public Object next( )
    {return menu[position++];
    }
    public boolean hasNext( )
    { if (position >= menu.length || menu[position]==null)
        return false;
        else return true;
    }
}
```

```
public class IteratorDesignPattern {  
  
    public static void main(String[ ] args) {  
  
        String[ ] menu = {"Choucroute 14,5 euros","Spaghetti bolo 9 euros",  
                           "Pizza 4 fromages 10 euros"};  
  
        RestaurantMenu restoMenu = new RestaurantMenu(menu);  
  
        MenuIterator iterateur = restoMenu.iterator( );  
  
        while (iterateur.hasNext( ))  
        { System.out.println(iterateur.next( ));  
        }  
    }  
}
```

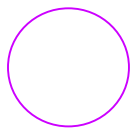
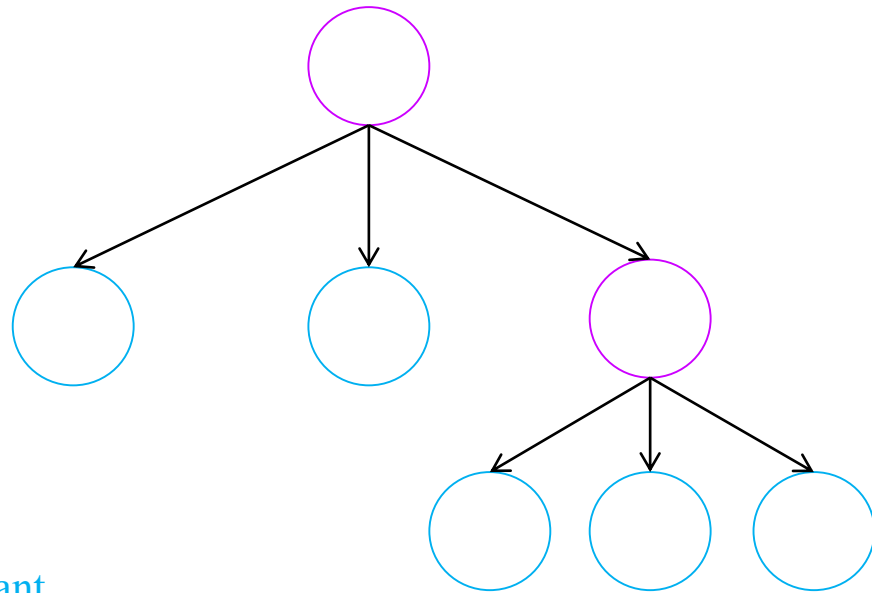
# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern
- 10.5. Data Access Object Pattern
- 10.6. Iterator Pattern
- 10.7. Composite Pattern

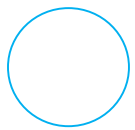
## Objectif du pattern composition

Organiser des objets en arborescence pour représenter des hiérarchies composants/composés

⇒ Permet de traiter de la même façon les objets individuels et les combinaisons de ceux-ci

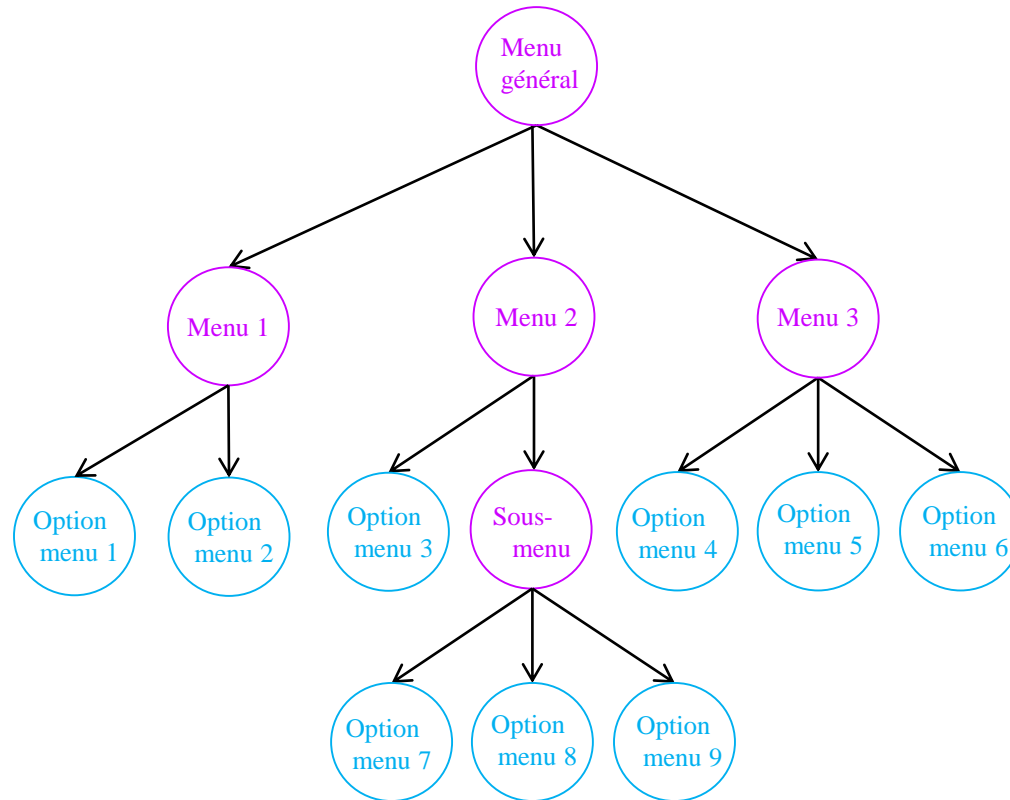


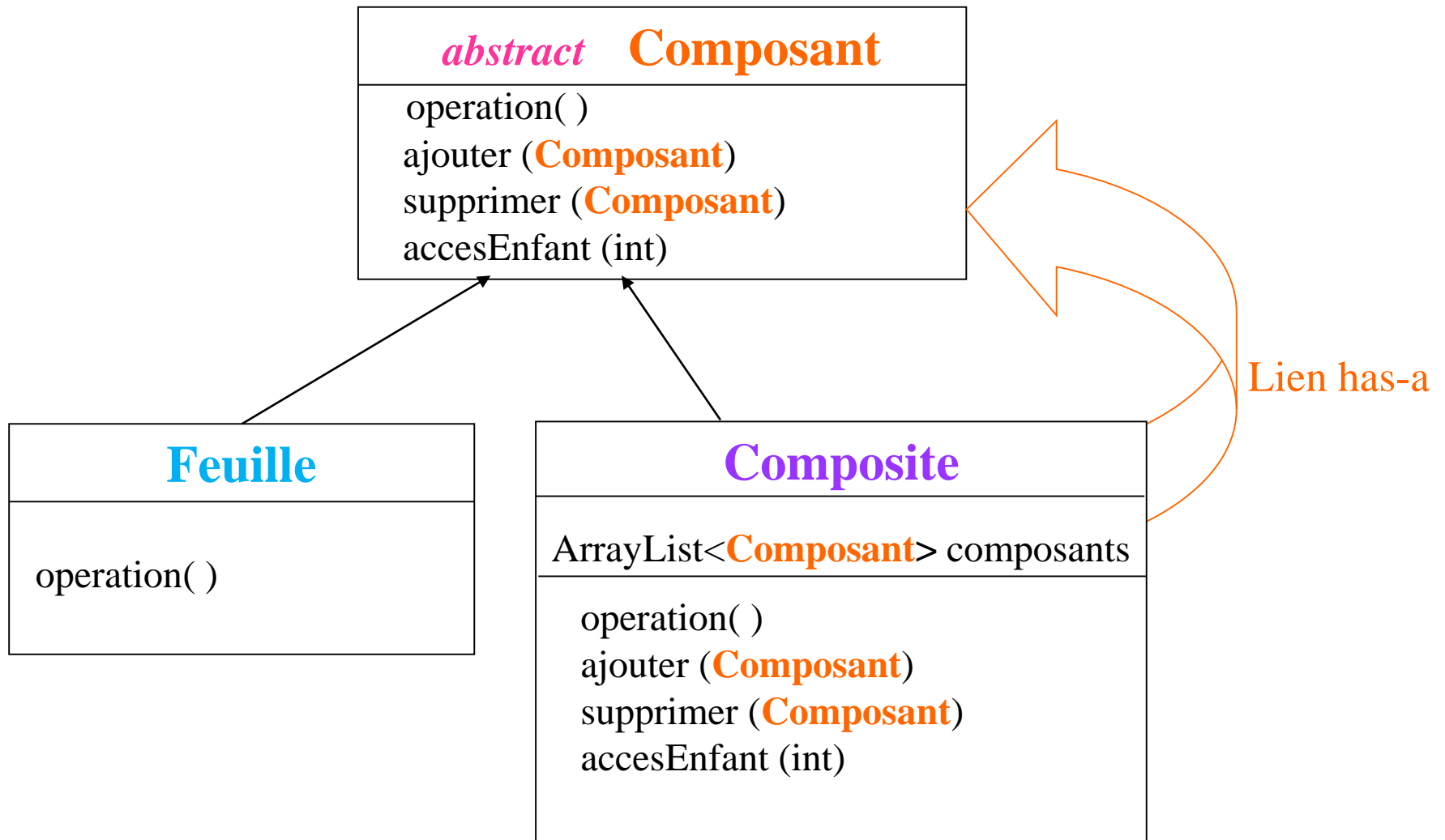
Nœud: élément qui a des enfants



Feuille : élément qui n'a pas d'enfant

## Exemple: arborescence de menus





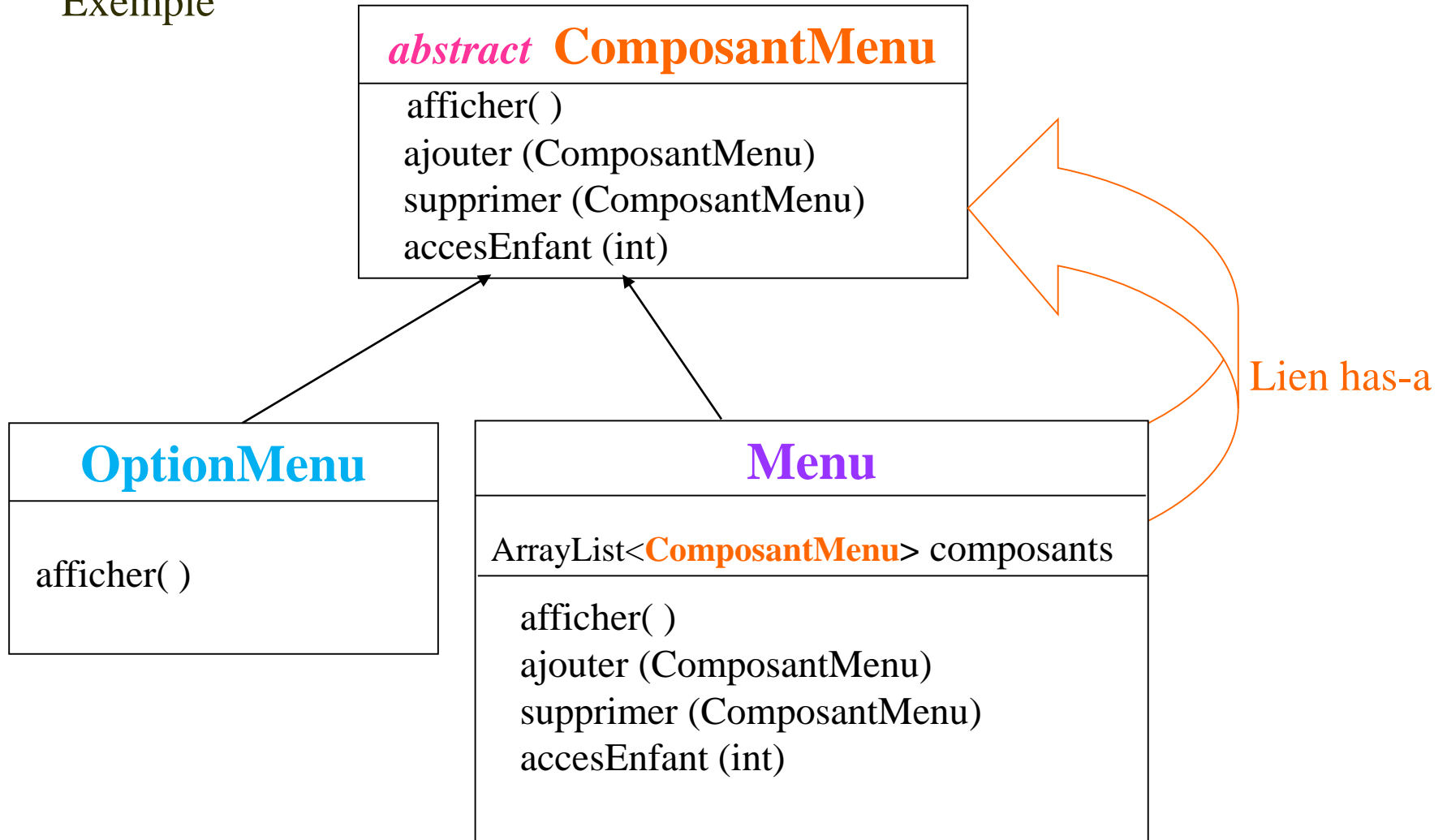
N.B. Certaines méthodes héritées de Compositant n'ont aucun sens pour la classe **Feuille** (ex: ajouter (Compositant), supprimer (Compositant) et accesEnfant (int)).

⇒ Implémentation par défaut de ces méthodes dans la classe abstraite **Compositant**

Ex: lever des exceptions du type **UnsupportedOperation**



## Exemple



```
public abstract class ComposantMenu

{
    public void afficher( )
        { throw new UnsupportedOperationException( ); }

    public void ajouter (ComposantMenu composantMenu)
        { throw new UnsupportedOperationException( ); }

    public void supprimer (ComposantMenu composantMenu)
        { throw new UnsupportedOperationException( ); }

    public ComposantMenu accesEnfant (int indice)
        { throw new UnsupportedOperationException( ); }

}
```

```
public class OptionMenu extends ComposantMenu
```

```
{  
    public void afficher( )  
        {  
            // affichage de l'option de menu  
        }  
}
```

```

public class Menu extends ComposantMenu
{
    private ArrayList <ComposantMenu> composants;

    public void ajouter (ComposantMenu composantMenu)
        { composants.add(composantMenu); }

    public void supprimer (ComposantMenu composantMenu)
        { composants.remove(composantMenu); }

    public ComposantMenu accesEnfant (int indice)
        { return composants.get(indice); }

    public void afficher( )
        {for (ComposantMenu composant : composants)
            {composant .afficher( );}
        }
}

```

# 10. Design Patterns

- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern
- 10.5. Data Access Object Pattern
- 10.6. Iterator Pattern
- 10.7. Composite Pattern
- 10.8. Decorator Pattern

## Objectif du pattern décorateur

Attacher dynamiquement des responsabilités supplémentaires à un objet

### Exemple:

#### Carte

##### Cafés:

*Colombie*

*Brésil*

*Déca*

*Espresso*

##### Suppléments:

*Lait*

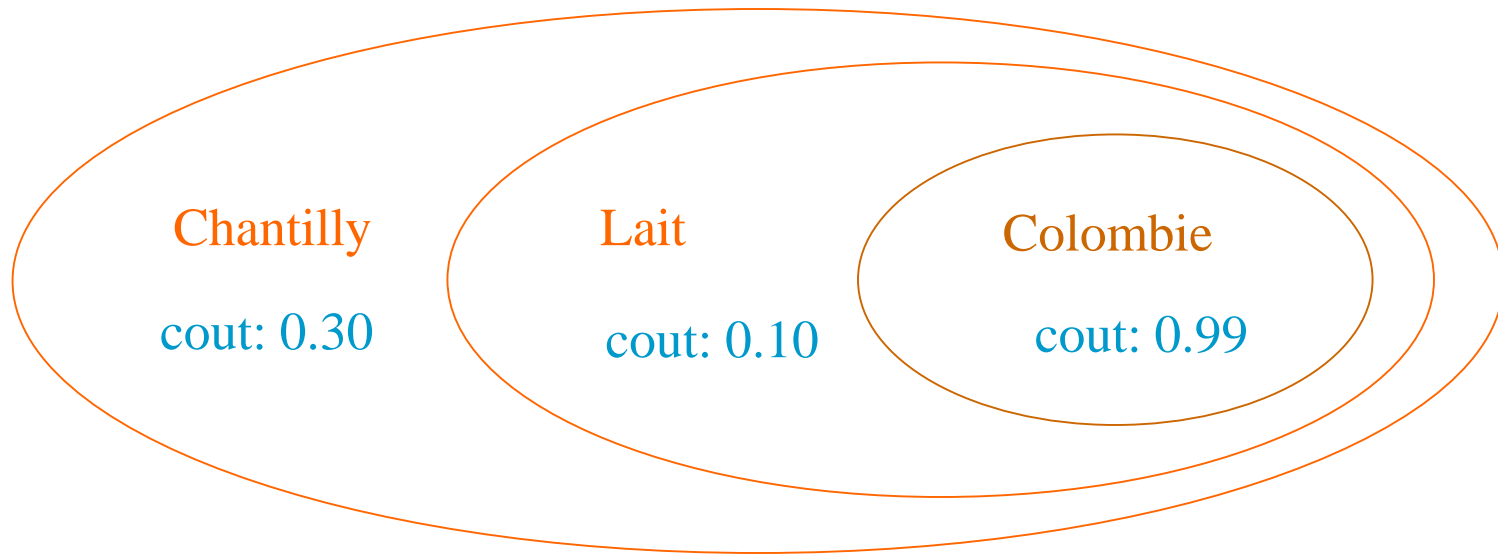
*Chocolat*

*Chantilly*

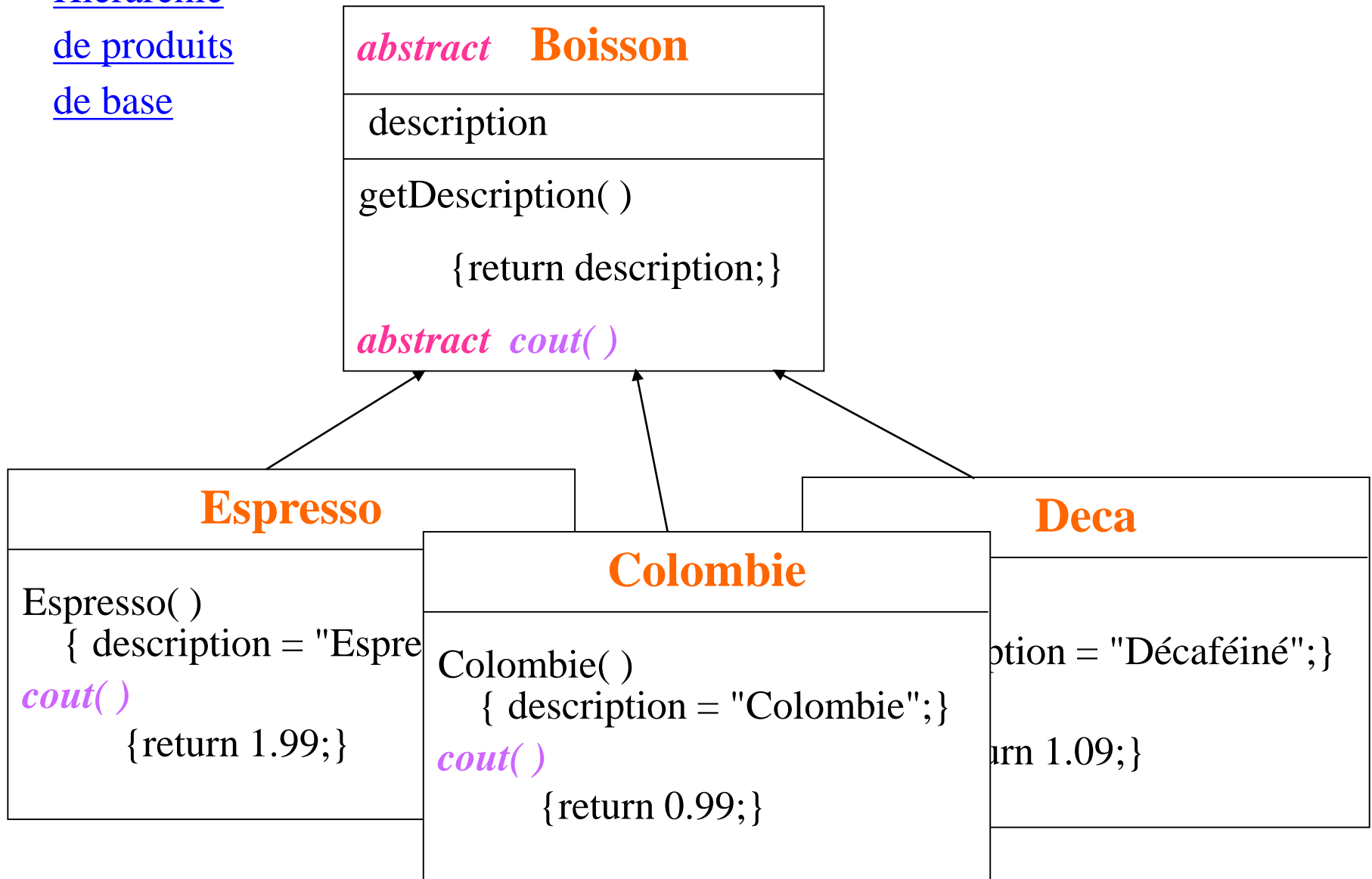
On peut choisir un café

+ un ou plusieurs suppléments

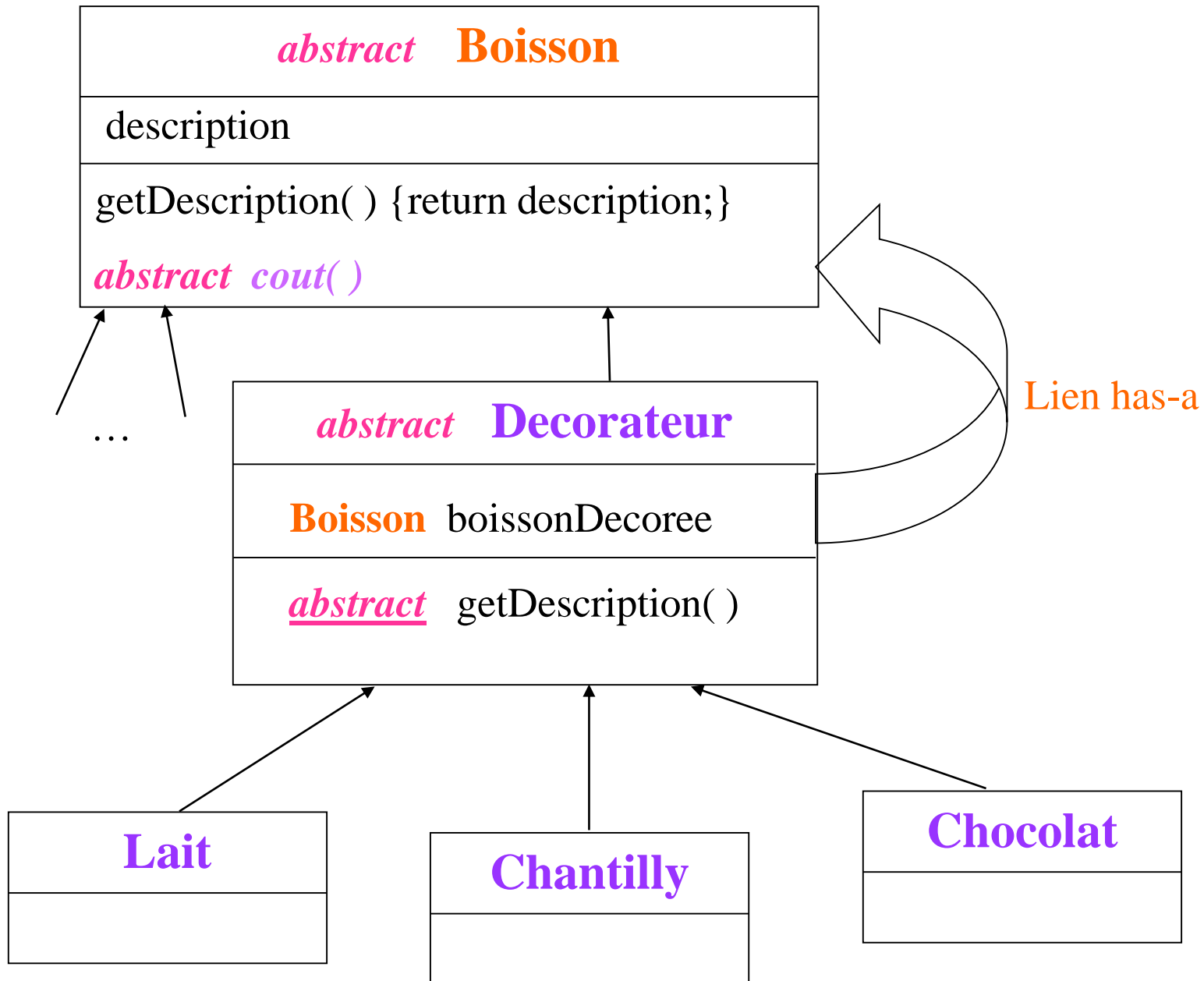
Ex: *Colombie + Lait + Chantilly*



Hiérarchie  
de produits  
de base

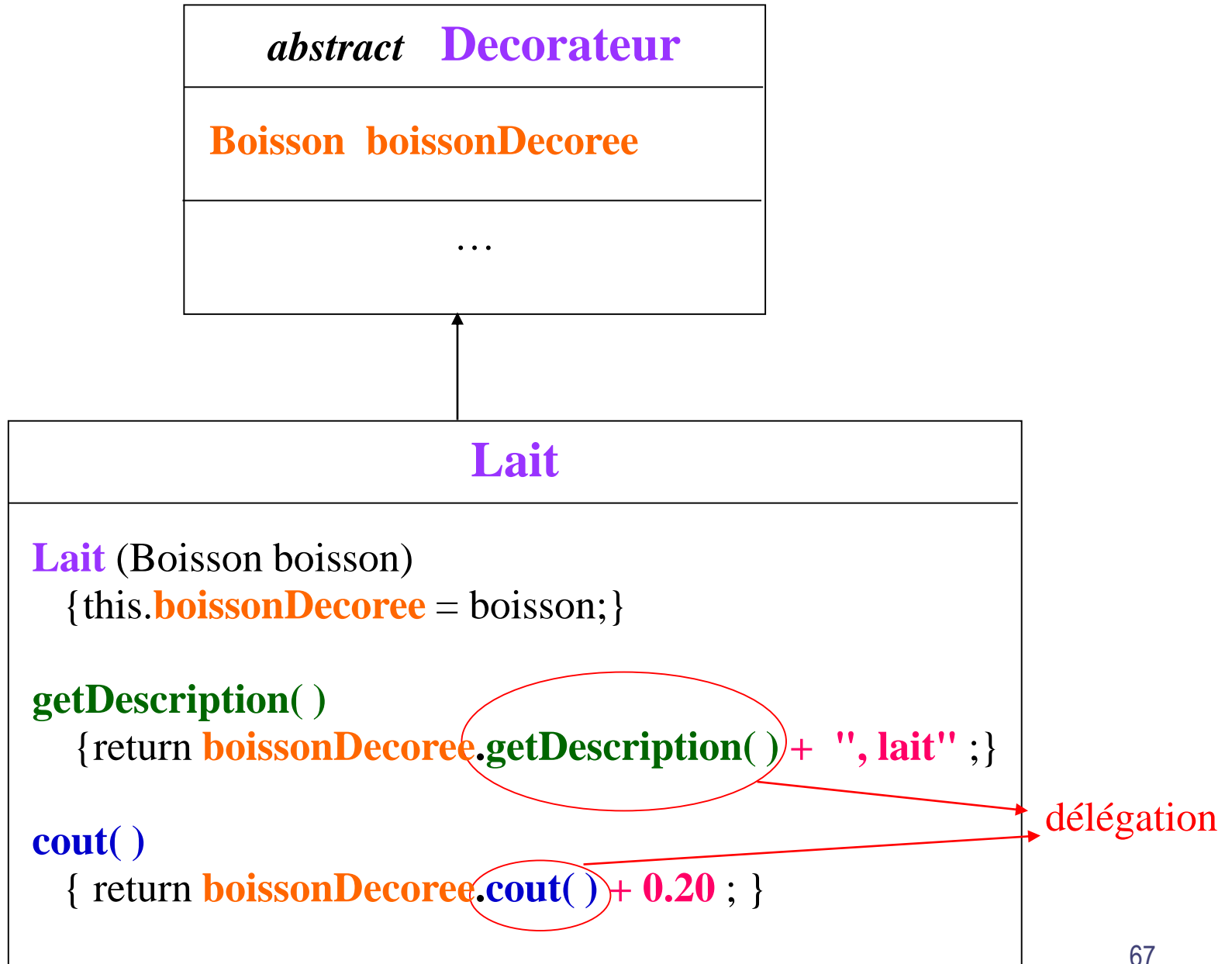


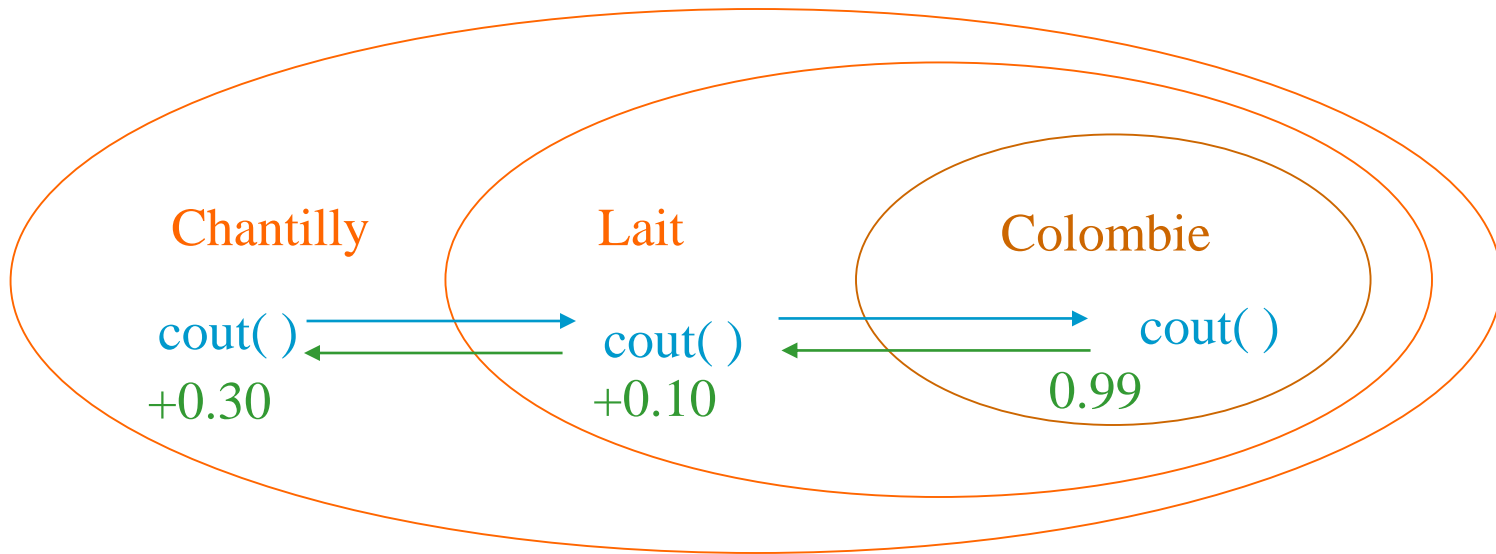




## Attention

- Tout objet d'une sous-classe de Décorateur a **un lien vers un objet qui est la boisson qu'il redécore** (c'est-à-dire auquel il ajoute un supplément: lait, chantilly, ...)  
Comme ce principe doit être **récuratif**, cet objet relié doit être un objet implémentant la super-classe abstraite
- Toute sous-classe de Décorateur doit *redéfinir* les méthodes *cout* et *description* pour y inclure la décoration supplémentaire





# 10. Design Patterns

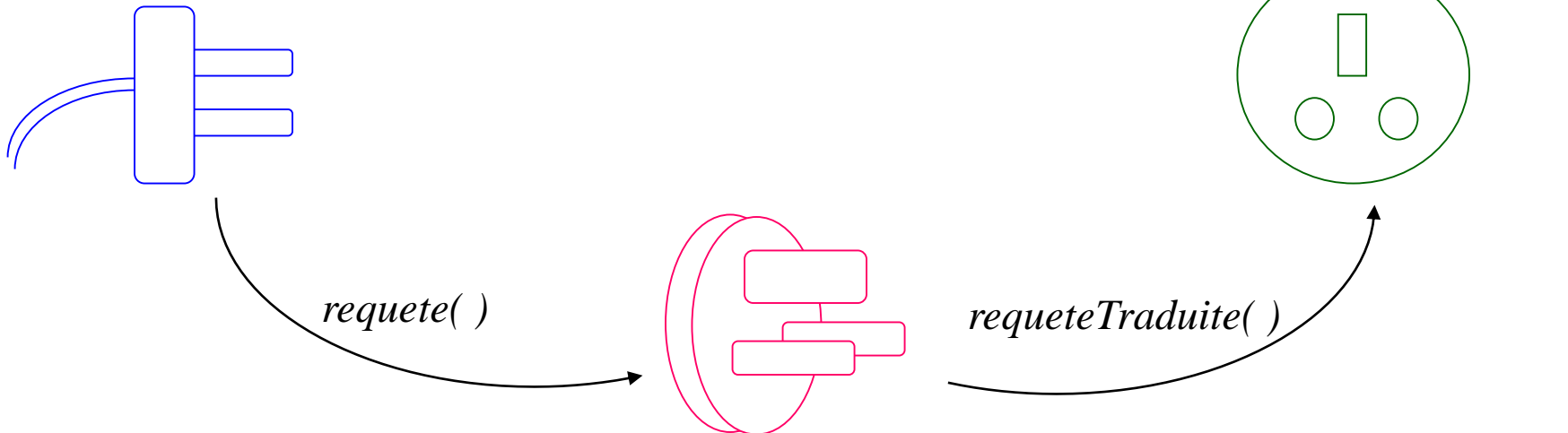
- 10.1. Strategy Pattern
- 10.2. Factory Pattern
- 10.3. Prototype Pattern
- 10.4. Singleton Pattern
- 10.5. Data Access Object Pattern
- 10.6. Iterator Pattern
- 10.7. Composite Pattern
- 10.8. Decorator Pattern
- 10.9. Adaptor Pattern

## Objectif du pattern adaptateur

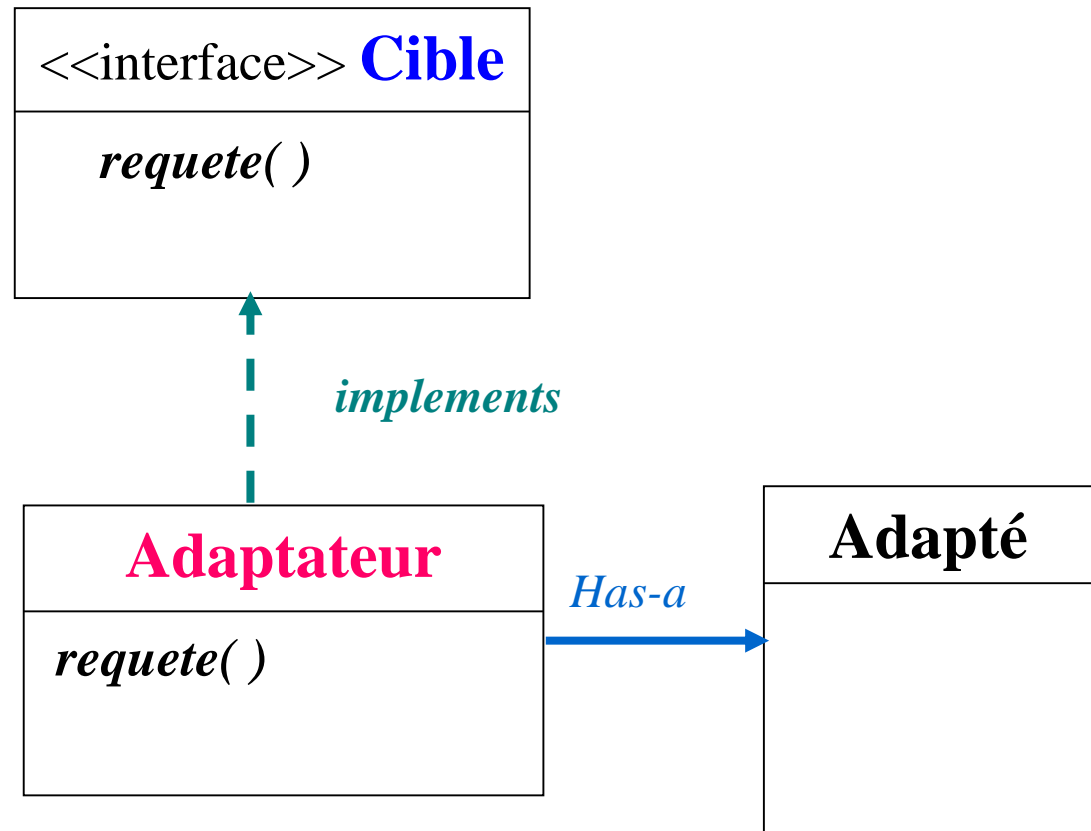
Rendre compatible deux interfaces incompatibles

➡ Permettre à des classes de collaborer alors qu'elles utilisent des interfaces incompatibles

*Client utilise méthodes d'une interface cible*

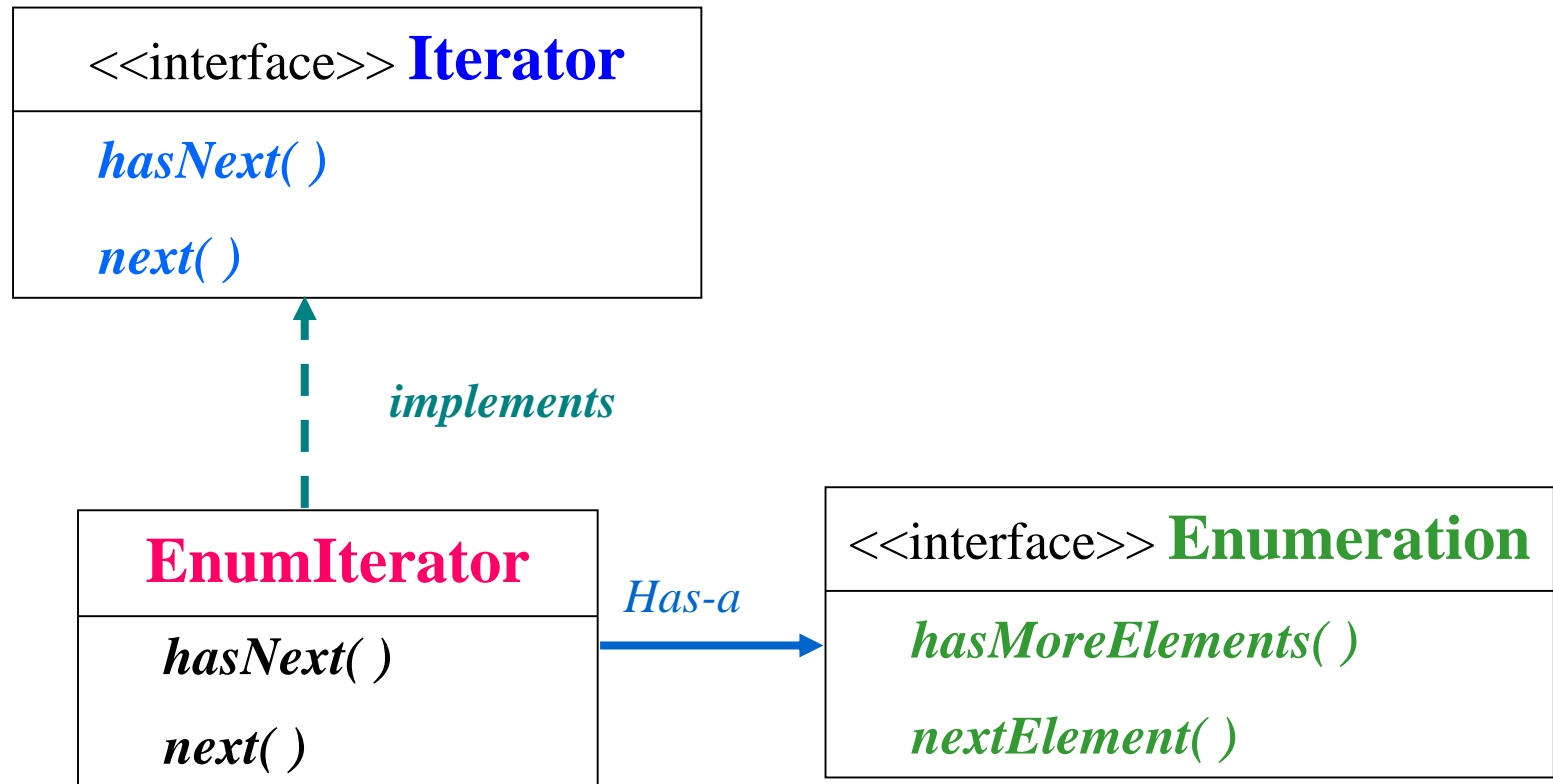


*L'adaptateur implémente l'interface cible et contient une référence vers une instance de l'adapté*



L'adaptateur traduit la requête du client en instructions compréhensibles par l'adapté (via appels de méthodes disponibles)

*Exemple: Le programme client manipule des itérateurs, alors que le programme cible manipule des énumérations*  
*On permet aux deux programmes de communiquer en créant l'adaptateur EnumIterator*





```
public class EnumIterator implements Iterator
```

```
{ private Enumeration enum;
```

```
    public EnumIterator (Enumeration enum)
```

```
    { this.enum = enum; }
```

```
    public boolean hasNext( )
```

```
    { return enum.hasMoreElements( ) ; }
```

```
    public Object next( )
```

```
    { return enum.nextElement( ) ; }
```

```
}
```

### Utilisation:

```
...  
Enumeration enum = ... ;  
EnumIterator adaptateur = new EnumIterator(enum);  
...  
while (adaptateur.hasNext() )  
    { ... adaptateur.next() ...  
    }
```

# 10. Design Patterns



...



10.6. Iterator Pattern



10.7. Composite Pattern



10.8. Decorator Pattern



10.9. Adaptor Pattern



10.10. Proxy Pattern

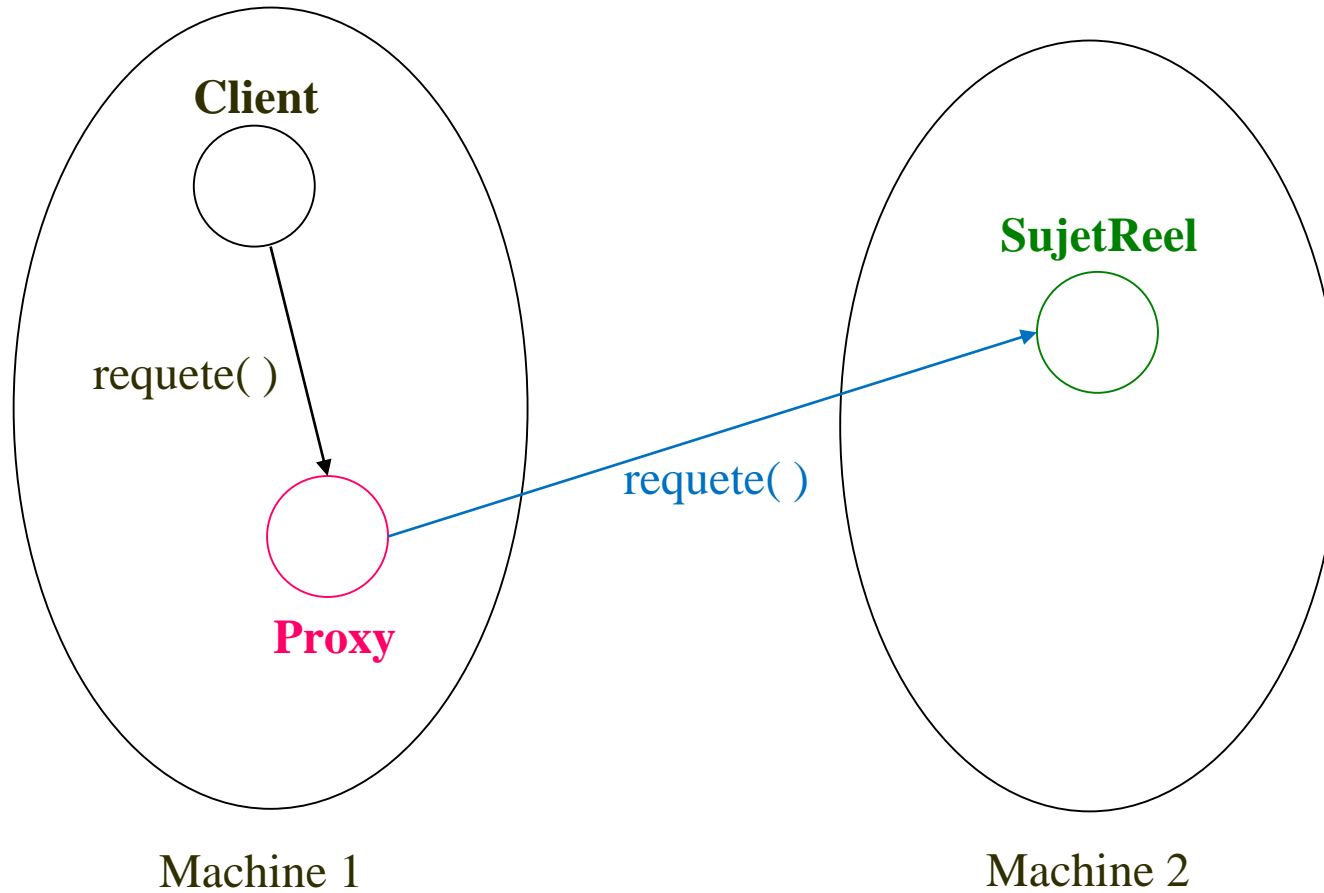
## Objectif du pattern **Proxy**

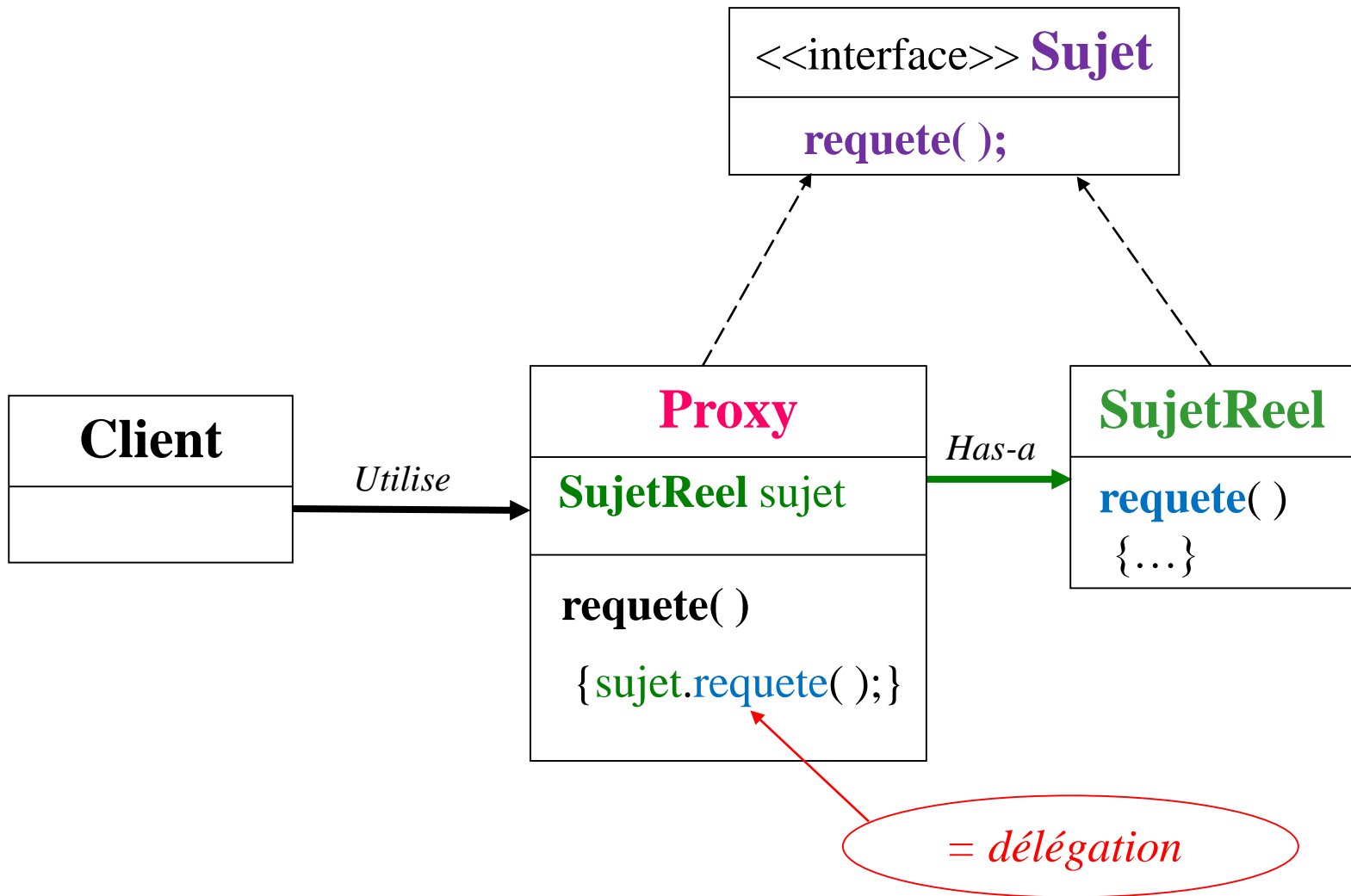
Fournir un objet remplaçant qui contrôle l'accès à un autre objet

*Pour objets*

- *distants (proxy distant)*
- *couteux à créer (proxy virtuel)*
- *qui doivent être sécurisés (proxy de protection)*

## Proxy distant





# 10. Design Patterns



...



10.6. Iterator Pattern



10.7. Composite Pattern



10.8. Decorator Pattern



10.9. Adaptor Pattern



10.10. Proxy Pattern

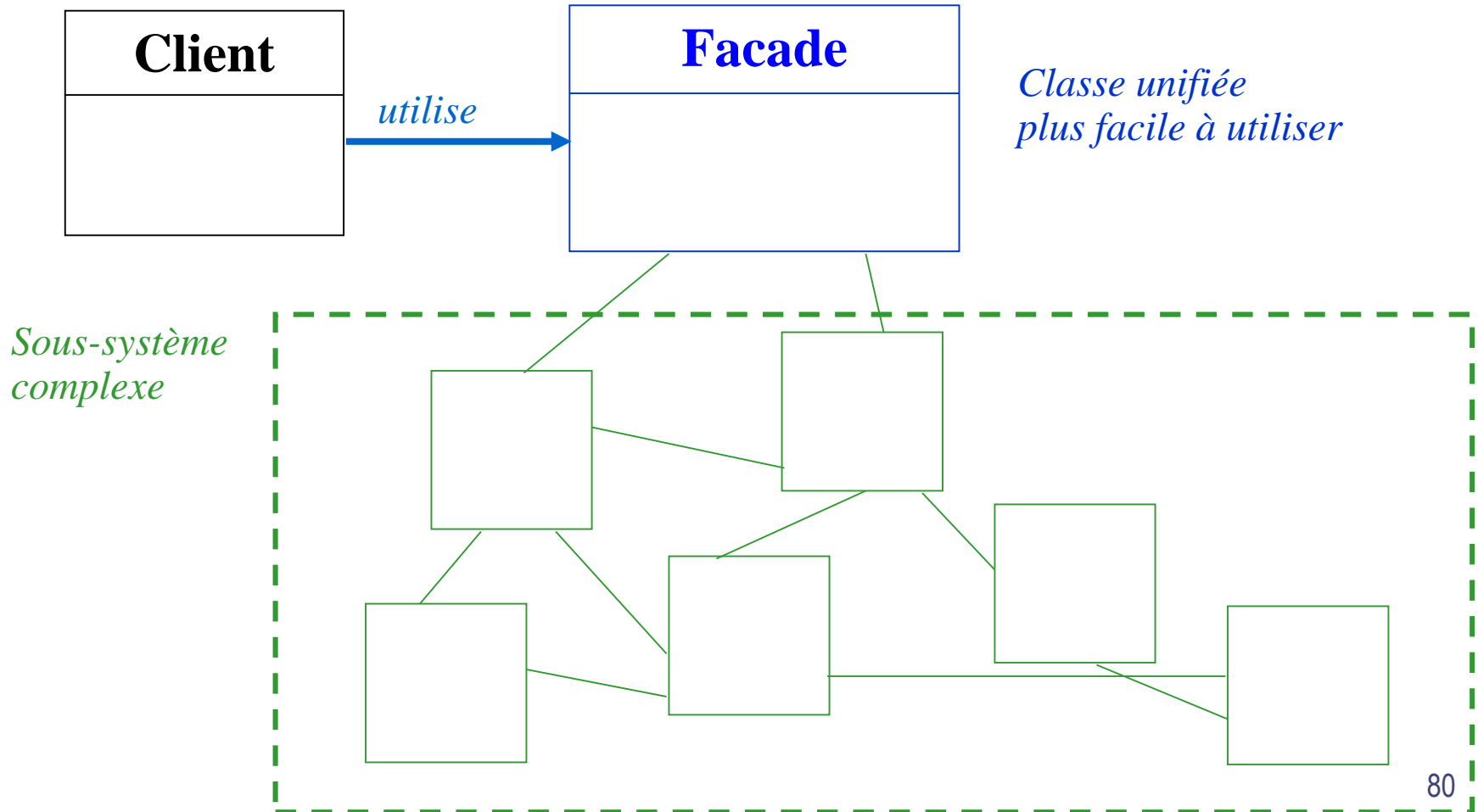


10.11. Facade Pattern

## Objectif du pattern façade

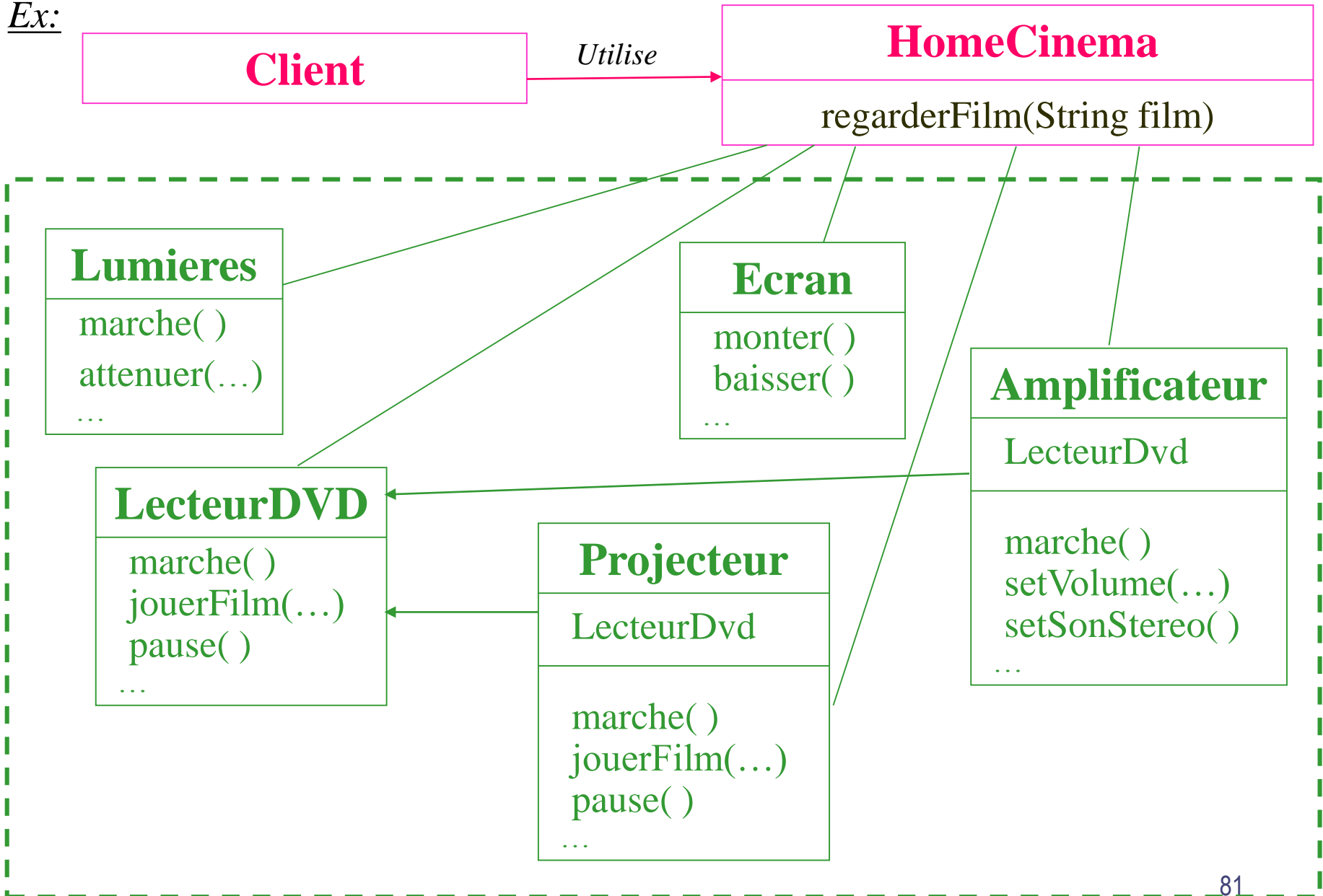
Faciliter l'utilisation d'un système complexe

➡ proposer une classe simplifiant et unifiant plusieurs classes plus complexes appartenant à un sous-système





Ex:



```
public class HomeCinema  
{private LecteurDVD dvd;  
    private Lumieres lumiere;  
    private Ecran ecran;  
    private Amplificateur ampli;  
    private Projecteur projo;  
  
    public HomeCinema (...) {...}  
  
    public void regarderFilm(String film)  
        {lumiere.attenuer(10);  
         ecran.baisser( );  
         projo.marche( );  
         ampli.marche( );  
         ampli.setSonStereo( );  
         ampli.setVolume(5);  
         dvd.marche( );  
         dvd.jouerFilm(film); }  
}
```

# 10. Design Patterns



10.6. Iterator Pattern

10.7. Composite Pattern

10.8. Decorator Pattern

10.9. Adaptor Pattern

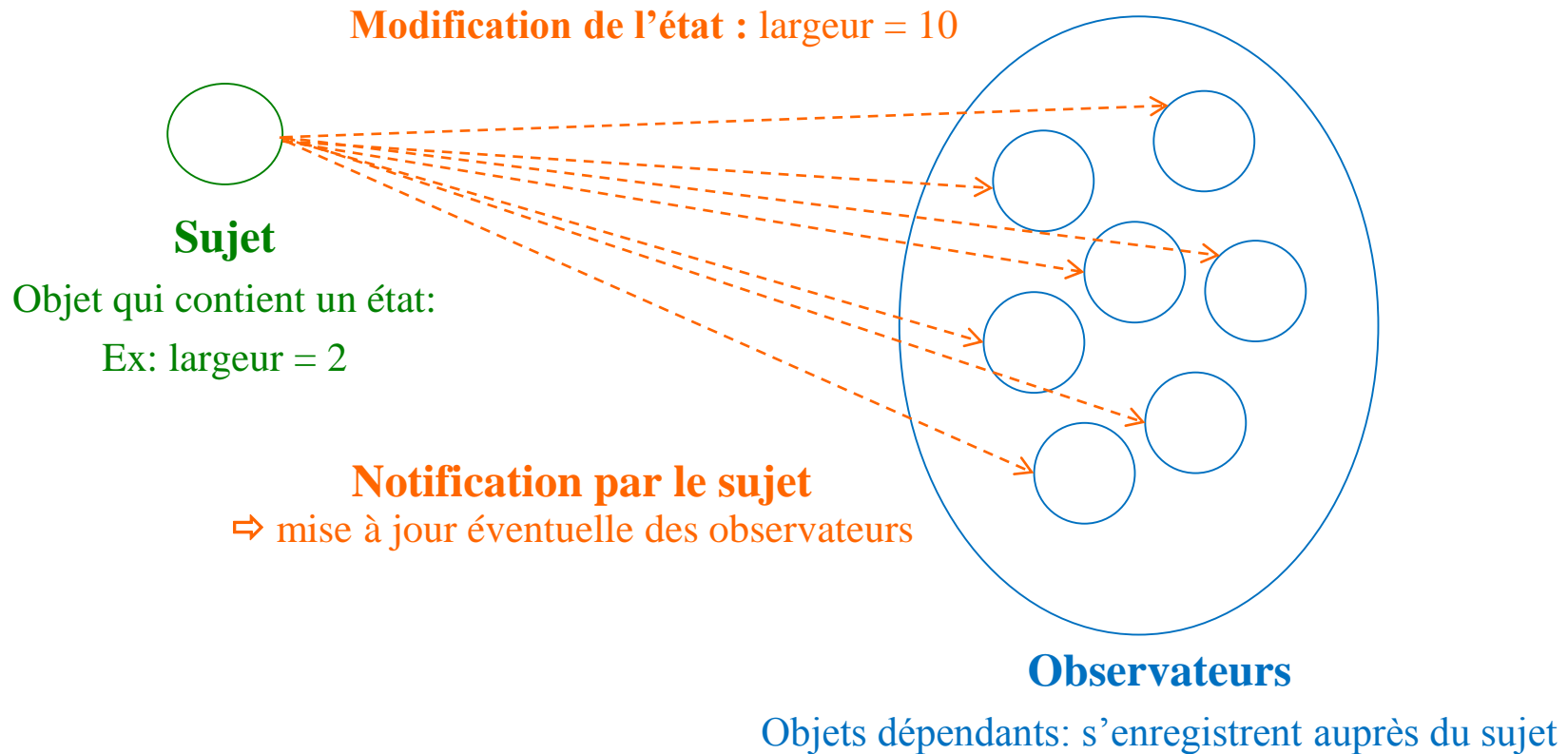
10.10. Proxy Pattern

10.11. Facade Pattern

10.12. Observer Pattern

## Objectif du pattern *observateur*

Lorsqu'un objet change d'état, notifier tous ceux qui en dépendent afin qu'ils soient mis à jour automatiquement (+ réaction éventuelle)



Le sujet contient

- une **liste des observateurs**
- une méthode pour **ajouter/supprimer un observateur** de la liste
- une méthode qui **boucle sur les observateurs pour les actualiser**:

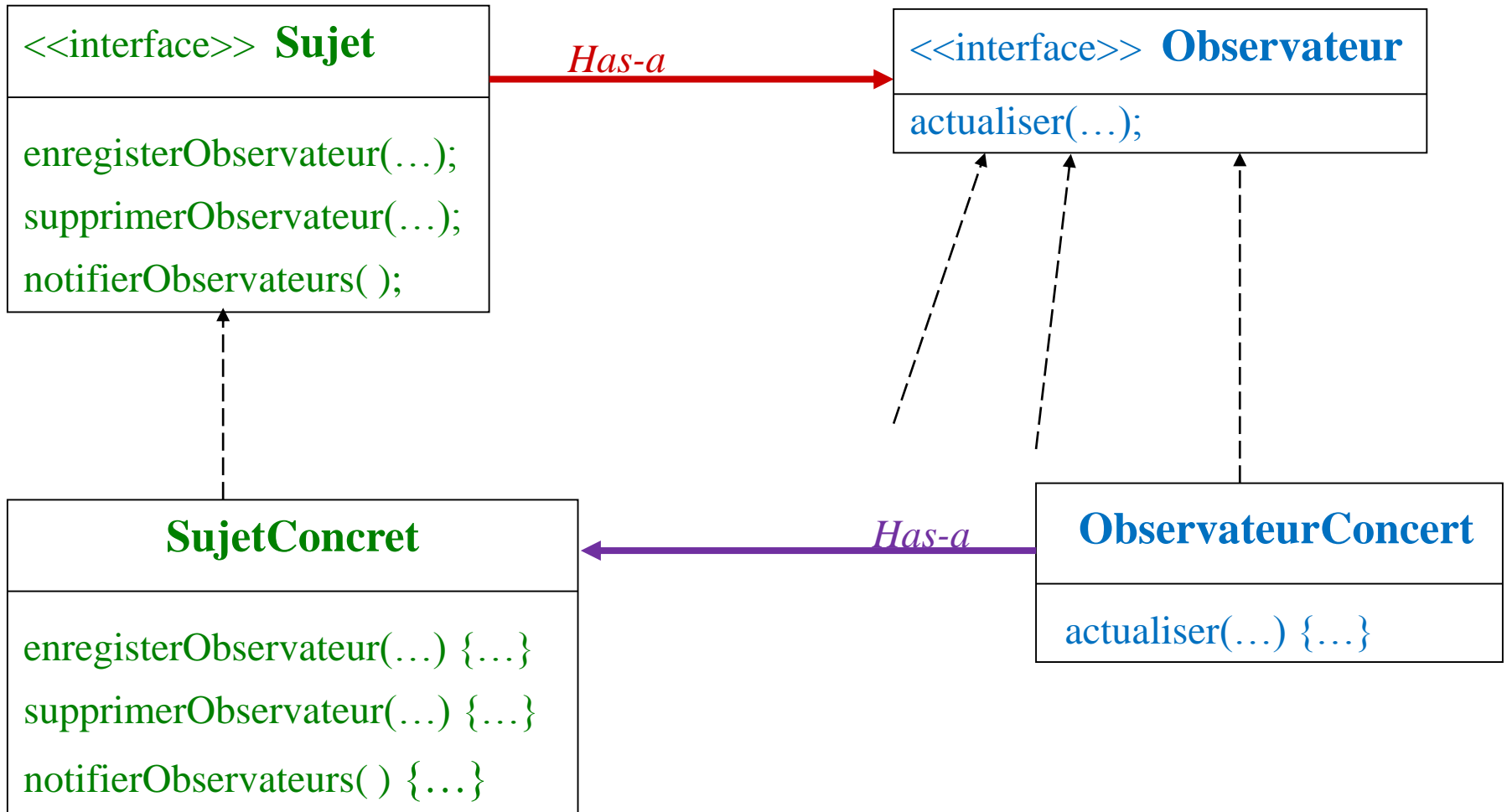
Appel d'une méthode sur chacun d'eux



Quelle méthode?



Les **observateurs doivent implémenter une interface**



## Exemple 1

Gestion des évènements des composants Swing:

Sujet: *JButton* bouton

Observateur: objet (*ecouteur*) d'une classe qui implémente *ActionListener*

① L'observateur s'enregistre auprès du sujet:

⇒ bouton.*addActionListener*(*ecouteur*)

② Quand clic sur le bouton:

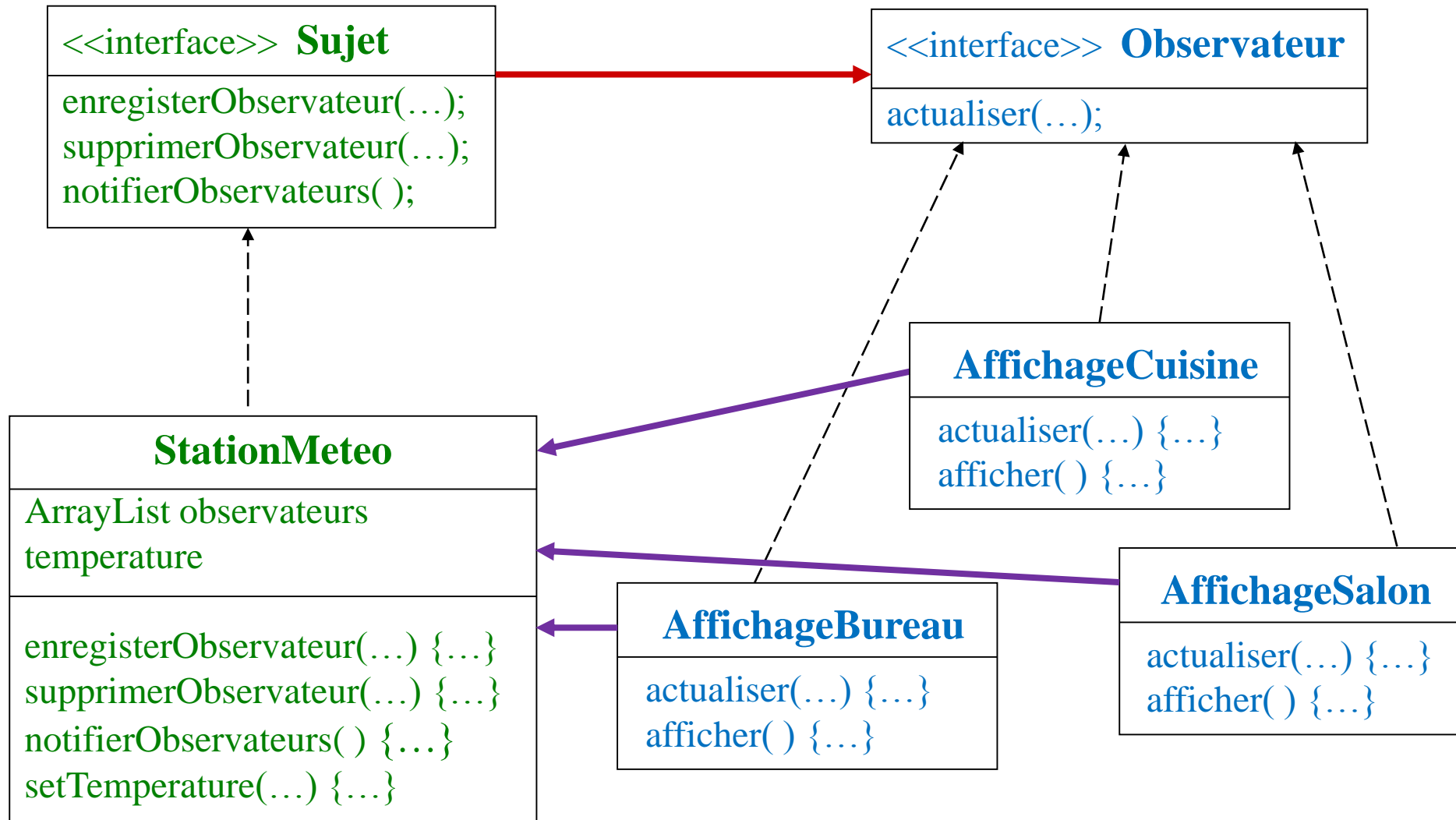
⇒ Appel par le sujet de la méthode *actionPerformed* sur tous les observateurs enregistrés

## Exemple 2

**Sujet:** station météo qui capte la température

**Observateurs:** appareils qui affichent la température captée par la station





```
public interface Sujet
{
    public void enregistrerObservateur (Observateur o);
    public void supprimerObservateur (Observateur o);
    public void notifierObservateurs ( );
}
```

```
public interface Observateur
{
    public void actualiser (float temperature);
    public void afficher ( );
}
```

public *class* **StationMeteo** implements **Sujet**

```
{ private ArrayList<Observateur> observateurs;  
  private float temperature;  
  
  public StationMeteo ( )  
  { observateurs = new ArrayList<Observateur>( ); }  
  
  public void enregistrerObservateur (Observateur o)  
  { observateurs.add(o); }  
  
  public void supprimerObservateur (Observateur o)  
  { observateurs.remove(o); }  
  
  public void notifierObservateurs ( )  
  { for (Observateur o: observateurs)  
    { o.actualiser(temperature); } }  
  
  public void setTemperature (float newTemperature)  
  { temperature = newTemperature;  
    notifierObservateurs( ); }  
}
```

A chaque modification de température,  
les observateurs sont notifiés

public *class* AffichageSalon implements Observateur

```
{ private Sujet donneesMeteo;  
  private float temperature;
```

```
  public AffichageSalon (Sujet donneesMeteo)  
  { this.donneesMeteo = donneesMeteo;  
    donneesMeteo.enregistrerObservateur(this); }
```

L'observateur s'enregistre auprès du sujet

```
  public void actualiser (float temperature)  
  { this.temperature = temperature;  
    afficher( ); }
```

L'observateur met à jour ses données  
( + réaction) quand il est notifié d'un  
changement du sujet

```
  public void afficher ( )  
  { // afficher température }  
}
```

Initialisation du sujet et des observateurs (ex: dans main)

StationMeteo **donneesMeteo** = new StationMeteo( );

AffichageSalon **affichageSalon** = new AffichageSalon (**donneesMeteo**);

AffichageSalon **affichageCuisine** = new AffichageCuisine (**donneesMeteo**);

AffichageSalon **affichageBureau** = new AffichageBureau (**donneesMeteo**);



# 10. Design Patterns



...



10.9. Adaptor Pattern



10.10. Proxy Pattern



10.11. Facade Pattern



10.12. Observer Pattern



10.13. State Pattern

Objectif du pattern **Etat**

Permettre à un objet de modifier son comportement  
quand son état interne change



*Comme si le code des méthodes appelées changeait  
en fonction de l'état de l'objet*



Comme si l'objet changeait de classe

### *Exemple: Classe distributeur de bonbons*

#### Etats possibles du distributeur

- *Pas de pièce*
- *A une pièce*
- *Plus de bonbon*
- *Bonbon vendu*

#### Actions possibles (méthodes)

- *Insérer une pièce*
- *Tourner poignée*
- *Ejecter une pièce*
- *Délivrer un bonbon*



## *Sans design pattern Etat:*

Etats du distributeur représentés par des constantes:

- Pas de pièce  $\Rightarrow$  SANS\_PIECE
- A une pièce  $\Rightarrow$  A\_PIECE
- Plus de bonbon  $\Rightarrow$  EPUISE
- Bonbon vendu  $\Rightarrow$  VENDU

+ mémoriser l'état courant

$\Rightarrow$  Pour chacune des méthodes, les réactions (codes des méthodes) diffèrent en fonction des états:

$\Rightarrow$  Dans chaque méthode, switch à faire sur les états

$\Rightarrow$  Lourd, répétitif et difficile à maintenir !!!

```

public class Distributeur {
    public final static int EPUISE = 0;
    public final static int SANS_PIECE = 1;
    public final static int A_PIECE = 2;
    public final static int VENDU = 3;

    private int etatCourant = EPUISE;
    private int nombreBonbons=0;

    public Distributeur (int nombre) {
        nombreBonbons = nombre;
        if (nombreBonbons > 0)
            { etatCourant = SANS_PIECE;}
    }

    public void insérerPiece()
    { switch (etatCourant) {
        case A_PIECE:
            System.out.println("Vous ne pouvez plus insérer de pièce!"); break;
        case SANS_PIECE:
            etatCourant = A_PIECE;
            System.out.println("Vous avez inséré une pièce."); break;
        case EPUISE:
            System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock!"); break;
        case VENDU:
            System.out.println("Veuillez patienter, le bonbon va tomber!"); break;}
    }
}

```

```

public void ejecterPiece( )
{ switch (etatCourant) {
    case A_PIECE:
        System.out.println("pièce retournée!");
        etatCourant = SANS_PIECE; break;
    case SANS_PIECE:
        System.out.println("Vous n'avez pas inséré de pièce."); break;
    case VENDU:
        System.out.println("Vous avez déjà tourné la poignée!"); break;
    case EPUISE:
        System.out.println("Ejection impossible, vous n'avez pas inséré de pièce!"); break;
}
}

public void tournerPoignee( )
{ switch (etatCourant) {
    case VENDU:
        System.out.println("Inutile de tourner deux fois!");
        break;
    case SANS_PIECE:
        System.out.println("Vous avez tourné mais il n'y a pas de pièce!");
        break;
    case EPUISE:
        System.out.println("Vous avez tourné mais il n'y a pas de bonbon!");
        break;
    case A_PIECE:
        System.out.println("Vous avez tourné ...");
        etatCourant = VENDU;
        delivrer(); break;
}
}

```

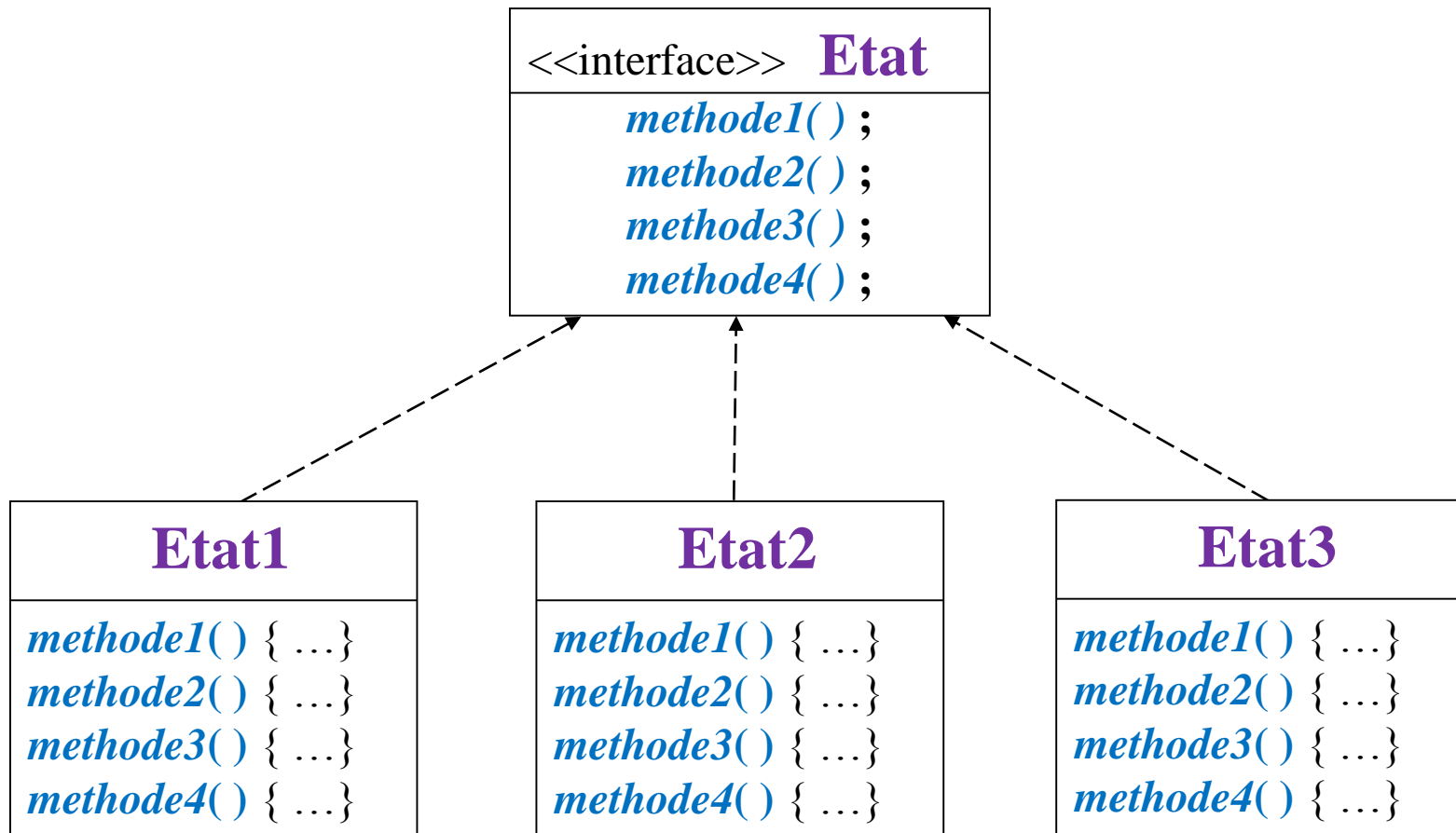
```

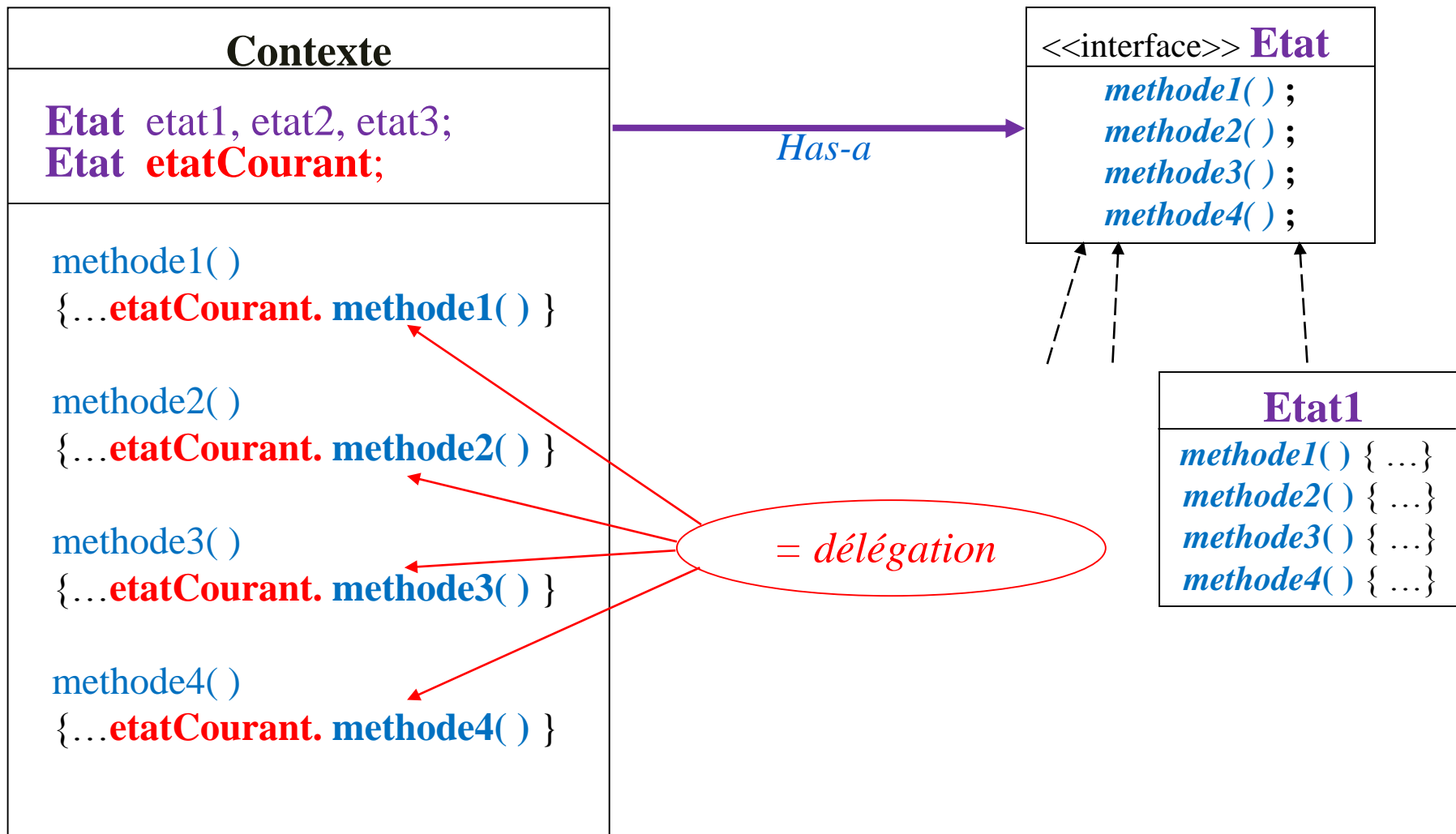
public void delivrer()
{ switch (etatCourant) {
    case VENDU:
        System.out.println("Un bonbon va sortir!");
        nombreBonbons -= 1;
        if (nombreBonbons == 0)
        {System.out.println("Plus de bonbon!!!");
        etatCourant = EPUISE;
        } else {etatCourant = SANS_PIECE;} break;
    case SANS_PIECE:
        System.out.println("Il faut payer d'abord!"); break;
    case EPUISE:
        System.out.println("Pas de bonbon délivré!"); break;
    case A_PIECE:
        System.out.println("Pas de bonbon délivré!"); break;}
}
}

```

*Avec design pattern Etat:*

Créer une hiérarchie d'états





```
public interface Etat {  
    void insererPiece( );  
    void ejecterPiece( );  
    void tournerPoignee( );  
    void delivrer( ); }
```

```
public class EtatSansPiece implements Etat{  
    private Distributeur distributeur;  
  
    public EtatSansPiece (Distributeur distributeur)  
    { this.distributeur = distributeur; }  
  
    public void insererPiece( )  
    { System.out.println("Vous avez inséré une pièce!");  
      distributeur.setEtatCourant(distributeur.getEtatAPiece()); }  
  
    public void ejecterPiece( )  
    { System.out.println("Vous n'avez pas inséré de pièce!"); }  
  
    public void tournerPoignee( )  
    { System.out.println("Vous avez tourné, mais il n'y a pas de pièce!"); }  
  
    public void delivrer( )  
    { System.out.println("Il faut payer d'abord!"); }  
}
```

```
public class EtatAPiece implements Etat{
    private Distributeur distributeur;

    public EtatAPiece (Distributeur distributeur)
    { this.distributeur = distributeur; }

    public void insererPiece ( )
    { System.out.println("Vous ne pouvez pas insérer d'autre pièce!"); }

    public void ejecterPiece( )
    { System.out.println("Pièce retournée!");
      distributeur.setEtatCourant(distributeur.getEtatSansPiece()); }

    public void tournerPoignee( )
    { System.out.println("Vous avez tourné...");
      distributeur.setEtatCourant(distributeur.getEtatVendu()); }

    public void delivrer( )
    { System.out.println("Pas de bonbon délivré!"); }
}
```



```

public class EtatVendu implements Etat{
    private Distributeur distributeur;

    public EtatVendu (Distributeur distributeur)
    { this.distributeur = distributeur; }

    public void insererPiece( )
    { System.out.println("Veuillez patienter, le bonbon va tomber!"); }

    public void ejecterPiece( )
    { System.out.println("Vous avez déjà tourné la poignée!"); }

    public void tournerPoignee( )
    { System.out.println("Inutile de tourner deux fois!"); }

    public void delivrer( )
    { distributeur.liberer();
      if (distributeur.getNombreBonbons()>0)
      { distributeur.setEtatCourant(distributeur.getEtatSansPiece()); }
      else
      { System.out.println("Plus de bonbon!!!");
        distributeur.setEtatCourant(distributeur.getEtatEpuise()); }
      }
    }
}

```

```
public class EtatEpuise implements Etat{
    private Distributeur distributeur;

    public EtatEpuise (Distributeur distributeur)
    { this.distributeur = distributeur; }

    public void insererPiece( )
    { System.out.println("Vous ne pouvez pas insérer de pièce, nous sommes en rupture de stock!"); }

    public void ejecterPiece( )
    { System.out.println("Ejection impossible, vous n'avez pas inséré de pièce!"); }

    public void tournerPoignee( )
    { System.out.println("Vous avez tourné, mais il n'y a pas de bonbon"); }

    public void delivrer( )
    { System.out.println("Pas de bonbon délivré!"); }
}
```

```

public class Distributeur {
    private Etat etatSansPiece, etatAPiece, etatVendu, etatEpuise;
    private Etat etatCourant;    private int nombreBonbons;

    public Distributeur(int nombreBonbons) {
        this.nombreBonbons = nombreBonbons;
        etatSansPiece= new EtatSansPiece(this);  etatAPiece = new EtatAPiece(this);
        etatVendu = new EtatVendu(this);  etatEpuise = new EtatEpuise(this);
        if (nombreBonbons > 0) { etatCourant = etatSansPiece; } else { etatCourant = etatEpuise; } }

    public void insererPiece( ) { etatCourant.insererPiece( ); }

    public void ejecterPiece( ) { etatCourant.ejecterPiece( ); }

    public void tournerPoignee( )
        { etatCourant.tournerPoignee( );
          etatCourant.delivrer( ); }

    public void liberer( )
        { System.out.println("Un bonbon va sortir...");
          if (nombreBonbons != 0) { nombreBonbons --; } }

    // + getters et setters
}

```

# 10. Design Patterns



...



10.9. Adaptor Pattern



10.10. Proxy Pattern



10.11. Facade Pattern



10.12. Observer Pattern



10.13. State Pattern



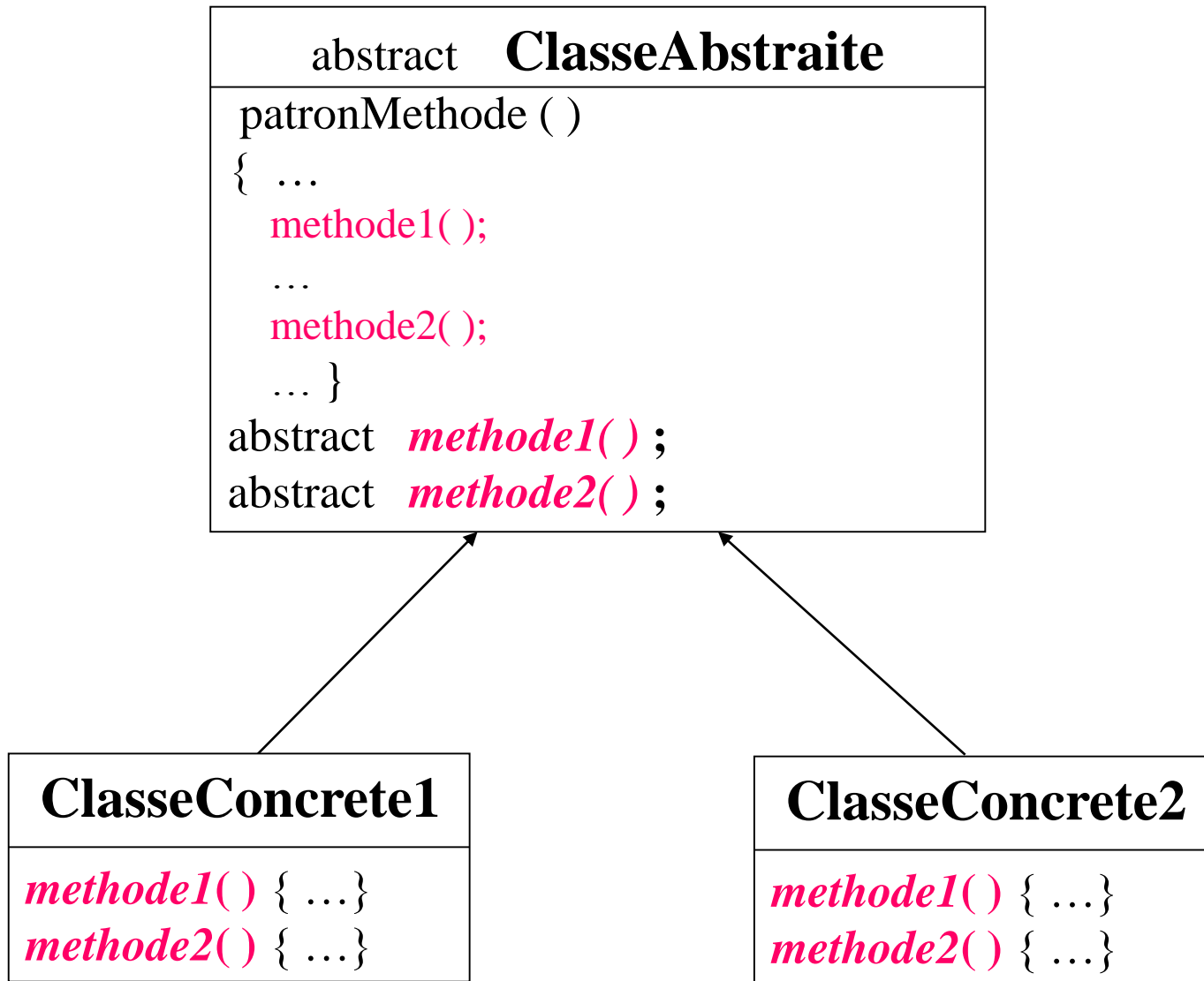
10.14. Template Method Pattern

*Objectif du pattern **Patron de méthode***

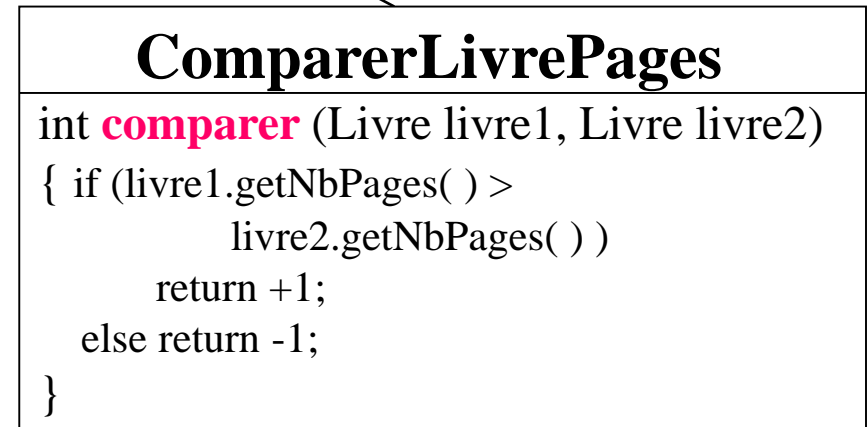
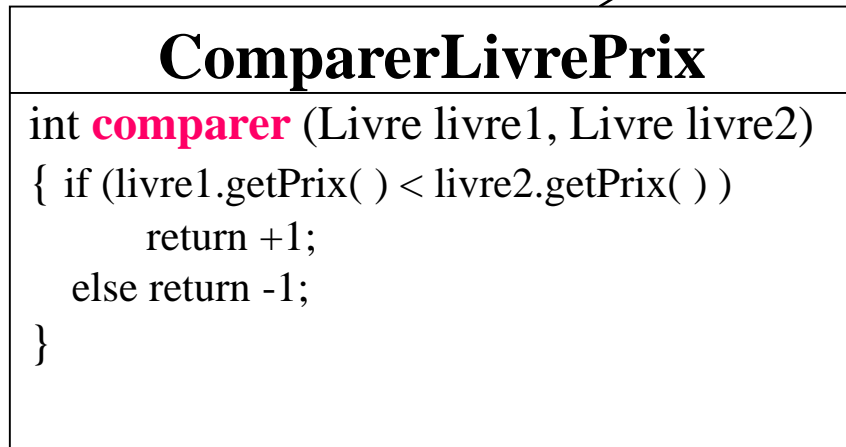
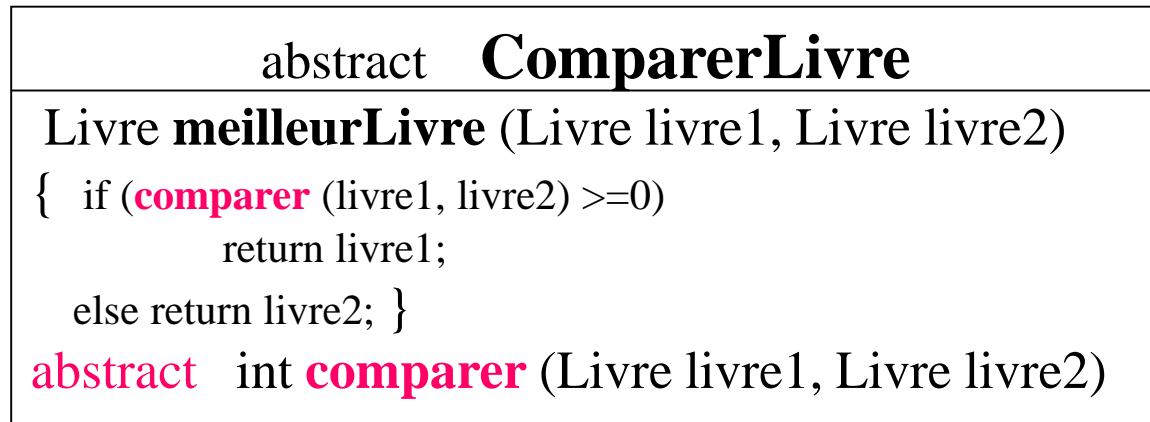
Définir le squelette d'un algorithme dans une méthode,  
en déléguant certaines étapes aux sous-classes



Les sous-classes redéfinissent certaines étapes d'un algorithme  
sans modifier la structure de celui-ci



*Ex. 1:*



## ***Utilisation:***

// Constructeur de Livre: premier argument = nombre de pages, second argument = prix

Livre livre1 = new Livre (100,10);

Livre livre2 = new Livre (200,50);

System.out.println("Meilleur livre : "+ new ComparerLivrePrix( ).meilleurLivre(livre1,livre2));

*//⇒Meilleur livre: livre1*

System.out.println("Meilleur livre: " +new ComparerLivrePages( ).meilleurLivre(livre1,livre2));

*//⇒Meilleur livre: livre2*



## Ex 2. Variante du patron de méthode

Méthode *sort* de la classe *Arrays*

```
public static void sort (Object[ ] a) {  
    Object[ ] aux = (Object[ ]) a.clone( );  
    mergeSort (aux, a, 0, a.length, 0);  
}
```

```
private static void mergeSort (Object[] src, Object[] dest, int low, int high, int off)
```

```
{ ...
```

```
    for (int i=low; i<high; i++)
```

```
        for (int j=i; j>low &&
```

```
            ( (Comparable) dest[j-1]).compareTo(dest[j])>0; j--)
```

```
            swap (dest, j, j-1); // méthode d'inversion de cellules existant dans la classe Arrays
```

```
        return;
```

```
    ...
```

```
}
```

Appel de *compareTo (...)* sur des objets de classes implémentant l'interface *Comparable*

```
public interface Comparable <T> {  
    public int compareTo(T o);  
}
```

***Utilisation:***

```
public class Rectangle implements Comparable{
```

```
    private int largeur, hauteur;
```

```
    public int surface ()
```

```
    { return largeur * hauteur; }
```

```
@Override
```

```
public int compareTo(Object objet) {
```

```
    Rectangle autreRectangle = (Rectangle)objet;
```

```
    if (this.surface( )< autreRectangle.surface( ))
```

```
        return -1;
```

```
    else if (this.surface( )== autreRectangle.surface( ))
```

```
        return 0;
```

```
    else return +1; }
```

```
}
```

Utilisation :

```
Rectangle[ ] rectangles = { ... }
```

```
Arrays.sort(rectangles);
```

# 10. Design Patterns



...



10.9. Adaptor Pattern



10.10. Proxy Pattern



10.11. Facade Pattern



10.12. Observer Pattern



10.13. State Pattern



10.14. Template Method Pattern



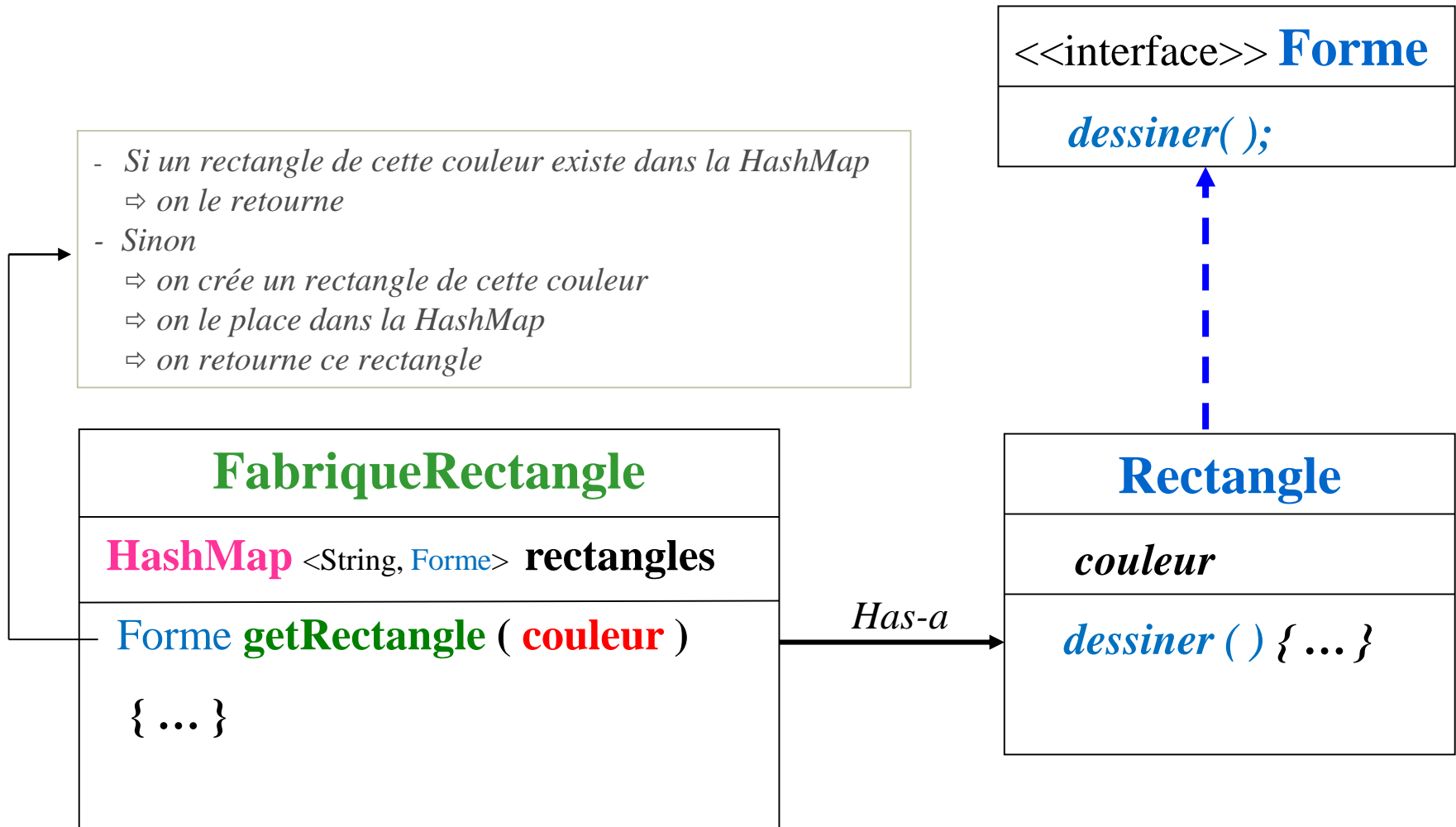
10.15. Flyweight Pattern

### Objectif du pattern *Flyweight* (poids mouche)

Réduire le nombre d'objets créés

- ⇒ pour diminuer la mémoire utilisée
- ⇒ et augmenter la performance

- ⇒ On stocke les objets créés
  - ⇒ On essaye de réutiliser un objet existant
  - ⇒ On ne crée un nouvel objet que si on ne trouve pas un objet similaire dans la zone de stockage



```
public interface Forme {  
    void dessiner( );  
}
```

```
public class Rectangle implements Forme{  
    private int largeur, hauteur;  
    private String couleur;  
  
    public Rectangle (String couleur) { this.couleur = couleur; }  
  
    @Override  
    public void dessiner( ) { System.out.println("Dessiner le rectangle "+ couleur); }  
  
    public void setLargeur(int largeur) { this.largeur = largeur; }  
    public void setHauteur(int hauteur) { this.hauteur = hauteur; }  
}
```



```

public class FabriqueRectangle {

    private static final HashMap<String, Forme> rectangles = new HashMap<>( );

    public static Forme getRectangle (String couleur)
    { Rectangle rectangle = (Rectangle) rectangles.get(couleur);
      if (rectangle == null)
      { rectangle = new Rectangle(couleur);
        rectangles.put(couleur,rectangle);
        System.out.println ("Création d'un nouveau rectangle " + couleur);
      }
      return rectangle;
    }
}

```

## Utilisation:

```
String[ ] couleurs = {"bleu", "rouge", "vert", "jaune"};
```

```
Rectangle rectangle;
```

```
for (int i=0; i<20; i++)
```

```
{ // Génération d'une couleur au hasard
```

```
String couleurAleatoire = couleurs[(int)(Math.random()*couleurs.length)];
```

```
// Demande d'un rectangle de cette couleur
```

```
rectangle = (Rectangle) FabriqueRectangle.getRectangle(couleurAleatoire);
```

```
rectangle.dessiner( );
```

```
}
```

## Exemples de sorties

*Création d'un nouveau rectangle bleu*  
*Dessiner le rectangle bleu*  
*Création d'un nouveau rectangle rouge*  
*Dessiner le rectangle rouge*  
*Création d'un nouveau rectangle vert*  
*Dessiner le rectangle vert*  
*Dessiner le rectangle vert*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle vert*  
*Création d'un nouveau rectangle jaune*  
*Dessiner le rectangle jaune*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle jaune*  
*Dessiner le rectangle rouge*  
*Dessiner le rectangle vert*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle jaune*  
*Dessiner le rectangle jaune*  
*Dessiner le rectangle jaune*  
*Dessiner le rectangle vert*  
*Dessiner le rectangle bleu*  
*Dessiner le rectangle bleu*

# 10. Design Patterns



...



10.12. Observer Pattern



10.13. State Pattern



10.14. Template Method Pattern



10.15. Flyweight Pattern



10.16. PlayerRole Pattern

### Objectif du pattern *PlayerRole* (Gestion des rôles)

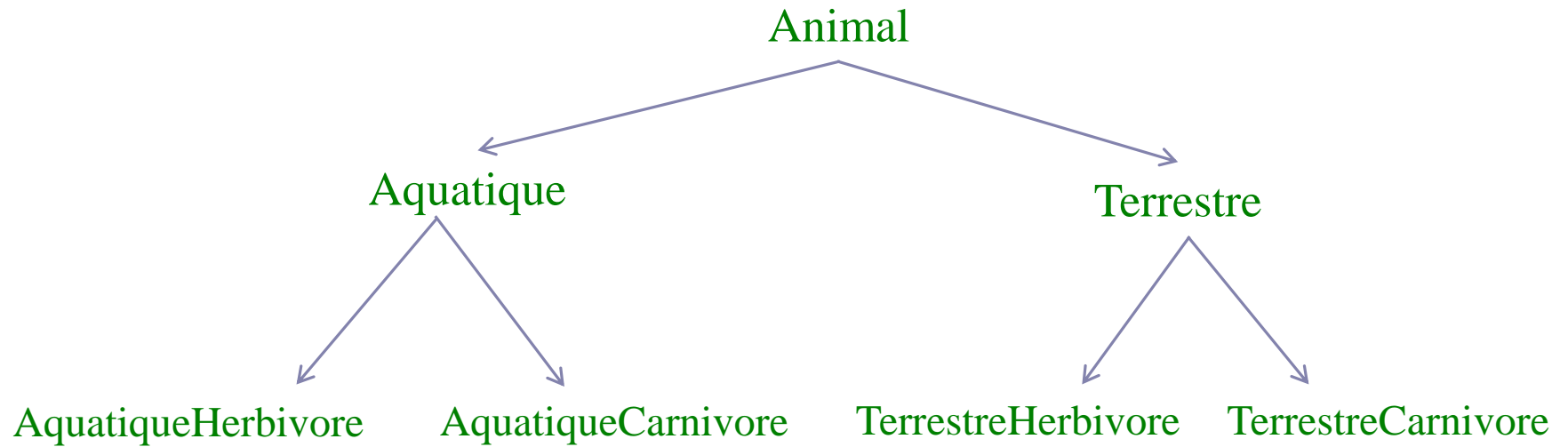
Permettre à un objet de jouer plusieurs rôles

+ de changer ses rôles dynamiquement

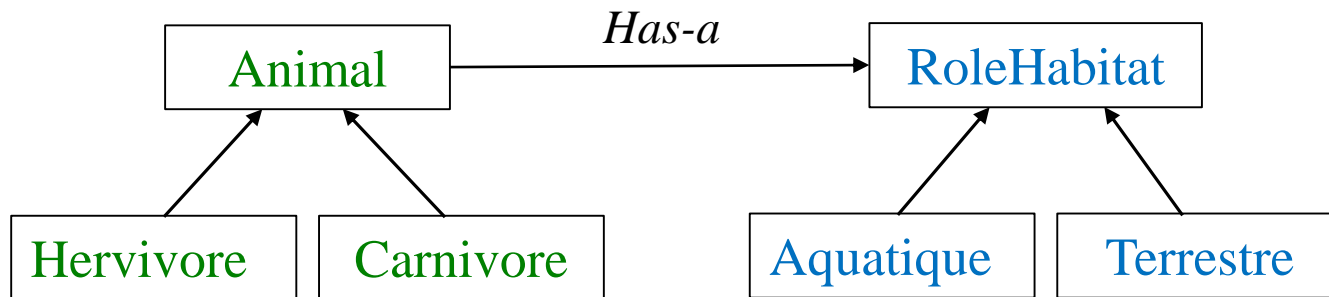
Avantages:

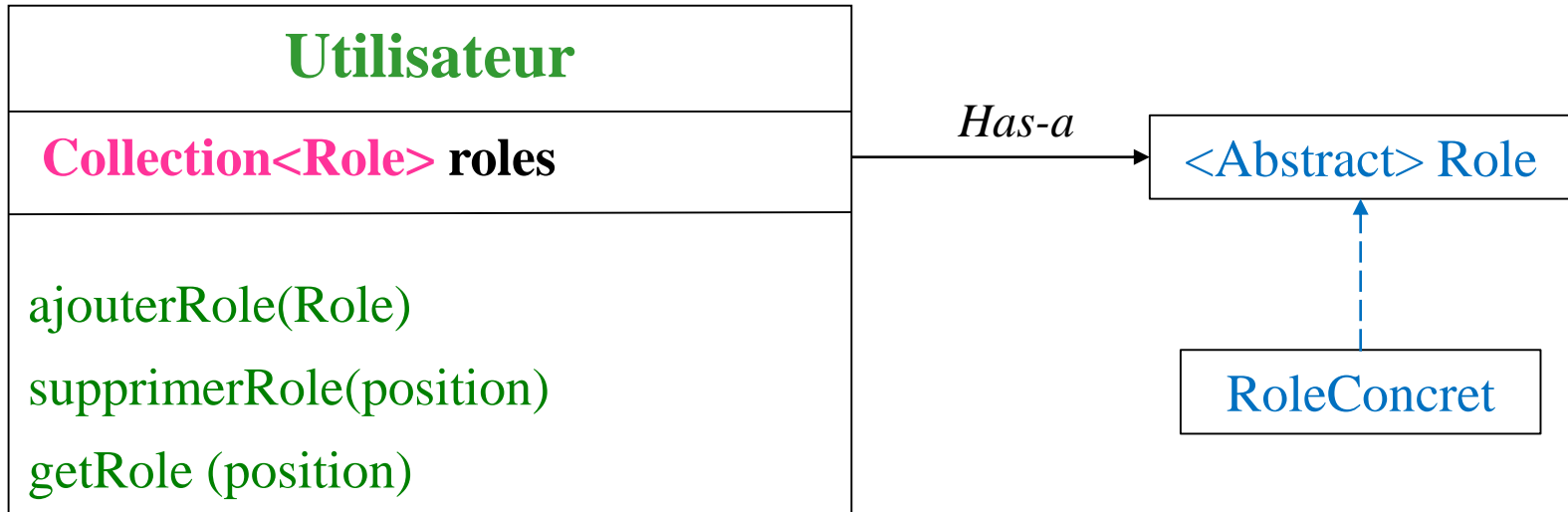
- Restreindre le couplage entre objets : on n'hardcode pas le comportement dans les utilisateurs
- Les rôles dynamiques empêchent les utilisateurs d'accéder à des méthodes interdites

## Sans Design Pattern (contre-exemple)



Avec Design Pattern PlayerRole







# 10. Design Patterns



...



10.12. Observer Pattern



10.13. State Pattern



10.14. Template Method Pattern



10.15. Flyweight Pattern



10.16. PlayerRole Pattern



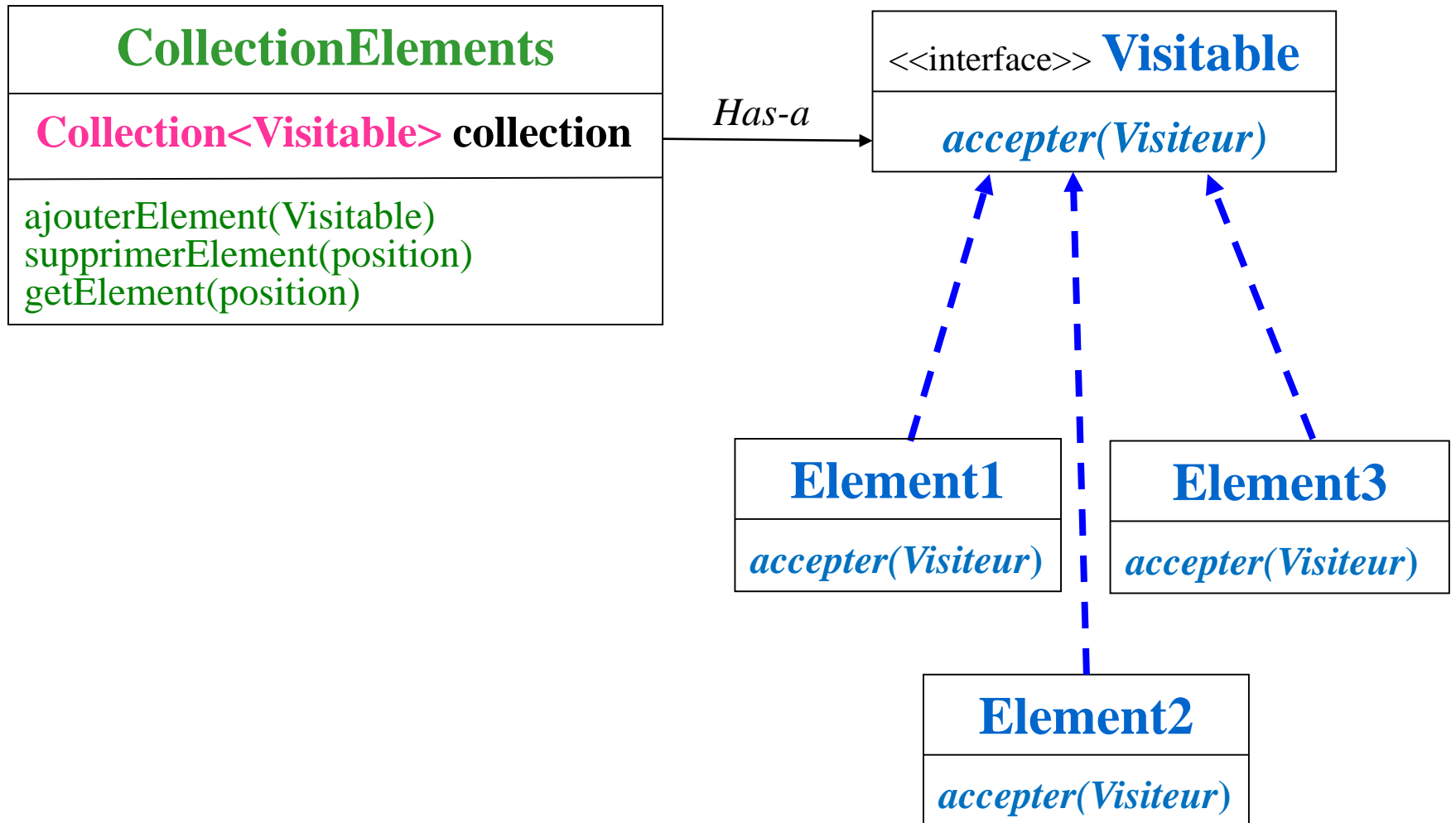
10.17. Visitor Design Pattern

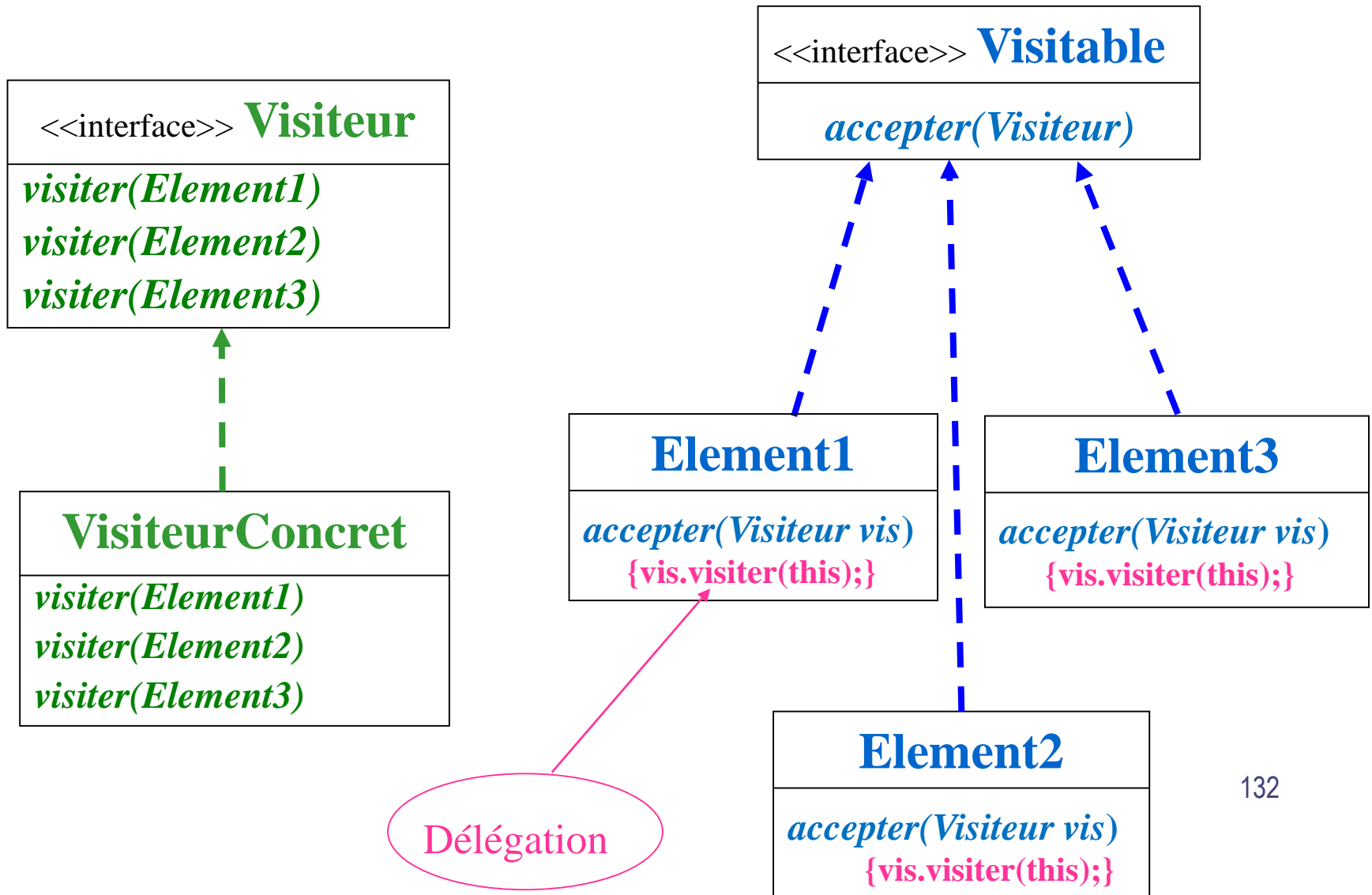
### Objectif du pattern *Visiteur*

Permettre d'appliquer une ou plusieurs opérations (algorithmes) sur un **ensemble d'éléments**

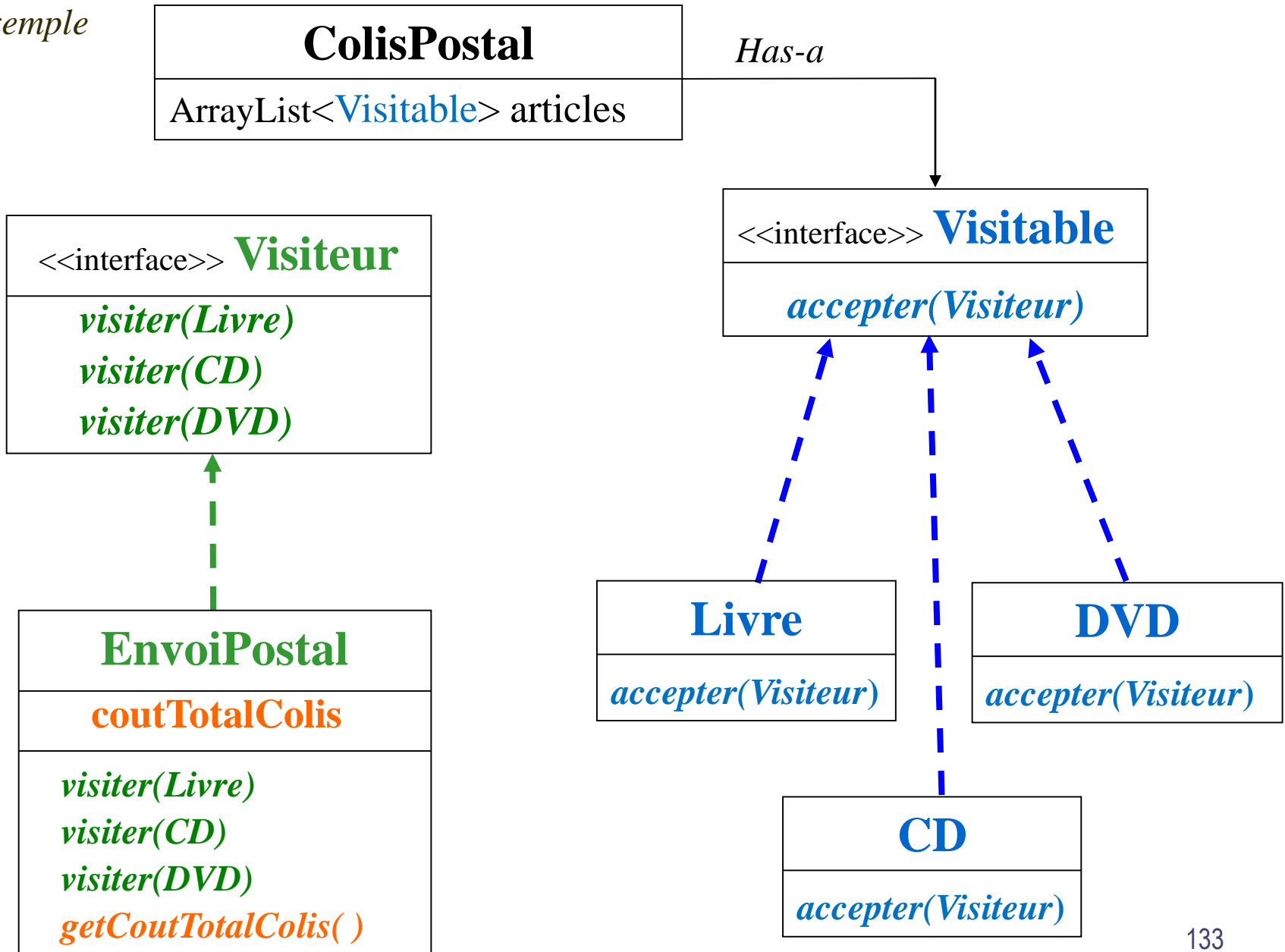
tout en découplant les opérations de la structure des objets.

- ⇒ Encapsuler dans un objet visiteur une opération (algorithme) à effectuer sur les éléments d'une structure
- ⇒ Chaque objet élément de la structure doit accepter l'objet visiteur de sorte qu'il puisse effectuer l'opération sur cet élément
- ⇒ Si le visiteur change, l'algorithme change





Exemple



```
public interface Visitable {  
    void accepter(Visiteur visiteur);  
}
```

```
public class Livre implements Visitable{  
    private double prix, poids;  
  
    public Livre(double prix, double poids)  
        { this.prix = prix; this.poids = poids; }
```

@Override

```
public void accepter (Visiteur visiteur)  
    { visiteur.visiter(this); }
```

```
public double getPrix( ) {return prix;}
```

```
public double getPoids( ) {return poids;}  
}
```

```
public interface Visiteur {  
    void visiter (Livre livre);  
    void visiter (CD cd);  
    void visiter (DVD dvd);  
}
```

```
public class EnvoiPostal implements Visiteur{  
    private double coutTotalColis;  
  
    @Override public void visiter (Livre livre)  
        { if (livre.getPrix( ) < 10.0)           // gratuit pour les livres > 10 euros  
            { coutTotalColis += livre.getPoids( ) * 2; } }  
  
    @Override public void visiter (CD cd)  
        { /* calcul cout envoi postal des CD et ajout à coutTotalPaquet*/ }  
  
    @Override public void visiter (DVD dvd)  
        { /* calcul cout envoi postal des DVD et ajout à coutTotalPaquet*/ }  
  
    public double getCoutTotalColis( ) { return coutTotalColis; }  
}
```

```

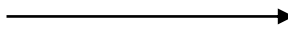
public class ColisPostal {
    private ArrayList<Visitable> articles;

    public ColisPostal ( ) { articles = new ArrayList<>( ); }

    public void ajouterArticle (Visitable visitable) { articles.add(visitable); }

    public double calculerCoutEnvoiPostal( )
    { EnvoiPostal poste = new EnvoiPostal( );
      for (Visitable article : articles)
      { article.accepter(poste); }
      return poste.getCoutTotalColis( );
    }
}

```


*Provoque l'appel de la méthode visiter  
de l'objet poste qui calcule le coût postal*



# 10. Design Patterns



...



10.12. Observer Pattern



10.13. State Pattern



10.14. Template Method Pattern



10.15. Flyweight Pattern



10.16. PlayerRole Pattern



10.17. Visitor Design Pattern

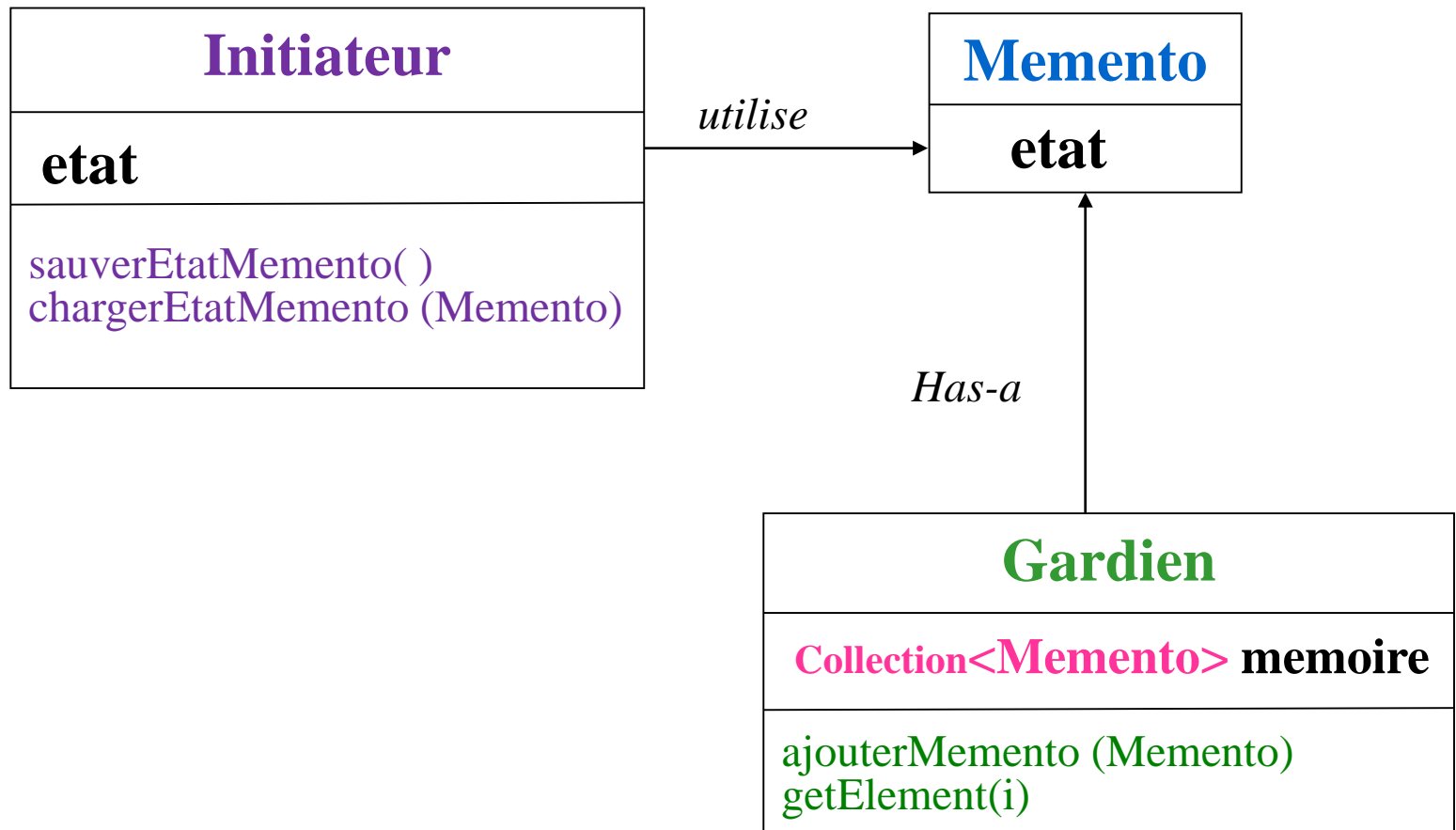


10.18. Memento Pattern

### Objectif du pattern **Memento**

Restaurer l'état d'un objet en y recopiant un de ses états précédants

- ⇒ Mémoriser à un moment donné l'état d'un objet dans un (objet) memento
- ⇒ Stocker des objets de type memento dans une collection gérée par un objet gardien
- ⇒ Rétablir quand nécessaire un des ces états antérieurs mémorisés



```
public class Memento {  
    private String etat;  
    public Memento(String etat) { this.etat = etat; }  
    public String getEtat( ) { return etat;}  
}
```

```
public class Initiateur {  
    private String etat;  
    public Initiateur(String etat) { this.etat = etat; }  
    public Memento sauverEtatMemento( )  
    { return new Memento (etat); }  
    public void ChargerEtatMemento (Memento memento)  
    { etat = memento.getEtat( ); }  
    public String getEtat( ) ...  
    public void setEtat(String etat) ...  
}
```

```

public class Gardien {
    private ArrayList<Memento> memoire ;
    public Gardien ( ) { memoire = new ArrayList< >( ); }
    public void ajouterMemento (Memento memento) { memoire.add(memento); }
    public Memento getElement (int i) { return memoire.get(i); }
}

```

Exemples d'utilisation:

```

Gardien gardien = new Gardien( );
Initiateur initiateur = new Initiateur("Samedi 16 avril");
initiateur.setEtat("Mercredi 15 juin");
// Sauvegarder l'état courant
gardien.ajouterMemento(initiateur.sauverEtatMemento( ));
initiateur.setEtat("Vendredi 24 juin");
initiateur.setEtat("Samedi 2 juillet");
gardien.ajouterMemento(initiateur.sauverEtatMemento( ));
initiateur.setEtat("Jeudi 21 juillet");
// Restaurer le dernier état sauvé
initiateur.setEtat(gardien.getElement(1).getEtat( ));

```

# 10. Design Patterns



...



10.12. Observer Pattern



10.13. State Pattern



10.14. Template Method Pattern



10.15. Flyweight Pattern



10.16. PlayerRole Pattern



10.17. Visitor Design Pattern



10.18. Memento Pattern



10.19. Mediator Pattern

### Objectif du pattern *Médiateur*

Réduire la complexité de la communication entre objets multiples

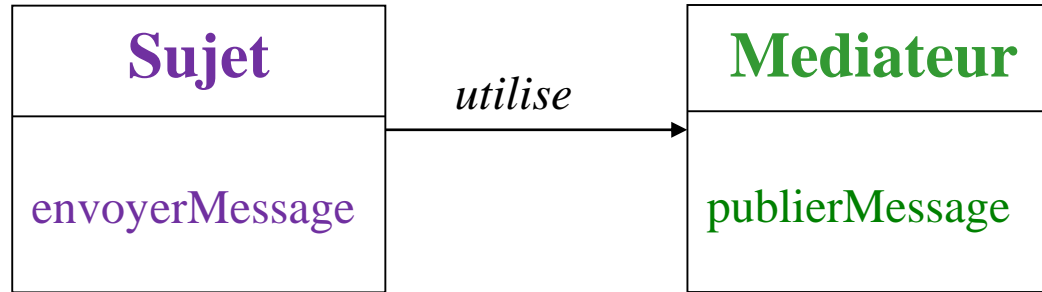
⇒ Encapsuler le processus de communication entre objets dans un médiateur

⇒ Diminue le couplage entre objets

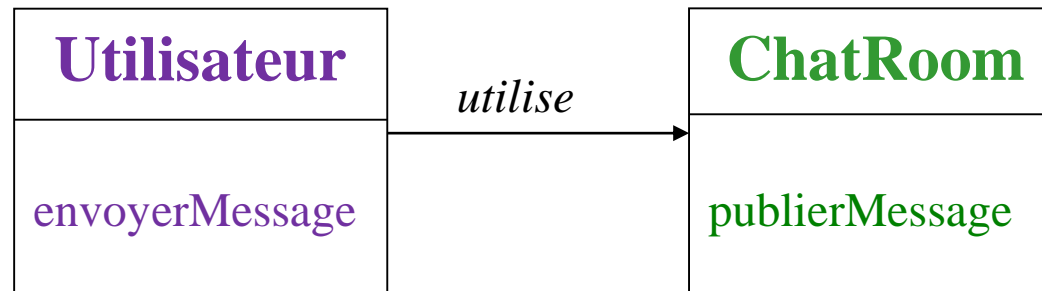
Empêche les objets de se référencer les uns les autres (**N à N**)

⇒ Facilite la maintenance

Transforme une relation N à N en une relation **1 à N**



*Exemple:*





```
public class Utilisateur {  
    private String nom;  
    public Utilisateur(String nom) { this.nom = nom; }  
    public String getNom( ) { return nom; }  
    public void envoyerMessage( String message)  
        { ChatRoom.publierMessage(this, message);}  
}
```

```
public class ChatRoom {  
    public static void publierMessage (Utilisateur utilisateur, String message){  
        System.out.println (utilisateur.getNom( ) + " a écrit : " + message);  
    }  
}
```

# 10. Design Patterns



...



10.20. Catégories de Design patterns

## Catégories de Design Patterns

1. Creational Patterns
2. Structural Patterns
3. Behavioral Patterns
4. Autres

# Creational Patterns

## 1. Factory Pattern

*Encapsuler la création d'objets*

## 2. Prototype Pattern

*Créer des objets sur base d'une instance prototype*

## 3. Singleton Pattern

*Garantir qu'une classe n'a qu'une seule instance*

# Structural Patterns

## 1. Composite Pattern

*Représenter des hiérarchies composants/composés*

## 2. Decorator Pattern

*Attacher dynamiquement des responsabilités supplémentaires à un objet*

## 3. Adaptor Pattern

*Rendre compatible deux interfaces incompatibles*

## 4. Proxy Pattern

*Fournir un objet remplaçant qui contrôle l'accès à un autre objet*

## 5. Facade Pattern

*Faciliter l'utilisation d'un système complexe*

## 6. Flyweight Pattern

*Réduire le nombre d'objets créés*

## 7. PlayerRole Pattern

*Permettre à un objet de jouer plusieurs rôles*

# Behavioral Patterns

## 1. Strategy Pattern

*Permettre à une partie du système de varier indépendamment des autres*

## 2. Iterator Pattern

*Accéder séquentiellement à une collection d'objets sans révéler son implémentation*

## 3. Observer Pattern

*Lorsqu'un objet change d'état, notifier tous ceux qui en dépendent afin qu'ils soient mis à jour automatiquement*

## 4. State Pattern

*Permettre à un objet de modifier son comportement quand son état interne change*

# Behavioral Patterns

## 5. Template Method Pattern

*Définir le squelette d'un algorithme en déléguant certaines étapes aux sous-classes*

## 6. Visitor Design Pattern

*Appliquer un algorithme sur un ensemble d'éléments, tout en découplant les opérations de la structure des objets*

## 7. Memento Pattern

*Restaurer l'état d'un objet en y recopiant un de ses états précédants*

## 8. Mediator Pattern

*Encapsuler le processus de communication entre objets dans un médiateur*

# Autres Patterns

## DAO Pattern

*Séparer la persistance des données de l'accès logique aux données*



*Bibliographie*

Tête la première, Design Patterns, Eric et Elisabeth Freeman, O'Reilly

*Webographie*

[http://www.tutorialspoint.com/design\\_pattern/](http://www.tutorialspoint.com/design_pattern/)