



# Chapitre 6

# Design Patterns

*Templates and best practices in software engineering  
to solve common problems when designing an application*

# Design Pattern

- Dans la conception (design) d'une application
- Un Design Pattern est
  - un modèle de conception de logiciel
  - une bonne pratique formalisée
  - une solution générale réutilisable
    - pour résoudre des problèmes similaires

# Catégories de Design Patterns

## Catégories de Design Patterns

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

# Catégories de Design Patterns

## 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*

# Catégories de Design Patterns

## 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*

# Catégories de Design Patterns

## Structural Patterns *(suite)*

### 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*

# Catégories de Design Patterns

## 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*

# Catégories de Design Patterns

## Behavioral Patterns (suite)

### 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 précédents*

### 8. Mediator Pattern

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



# Catégories de Design Patterns

## Autres Patterns

### DAO Pattern

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

# Design Patterns

## 1. Strategy Pattern

# Strategy Pattern

## Objectif du pattern **stratégie**

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



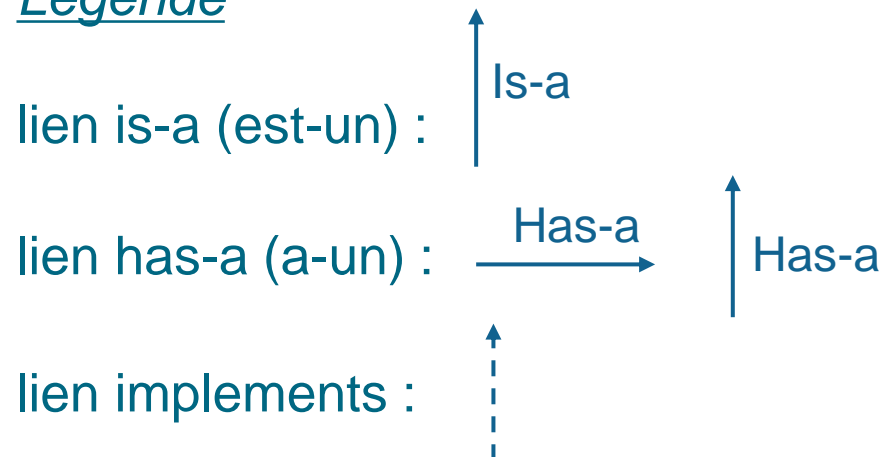
Encapsuler ce qui est susceptible de varier (encapsulation d'algorithmes)

**Extraire le comportement susceptible de varier :**

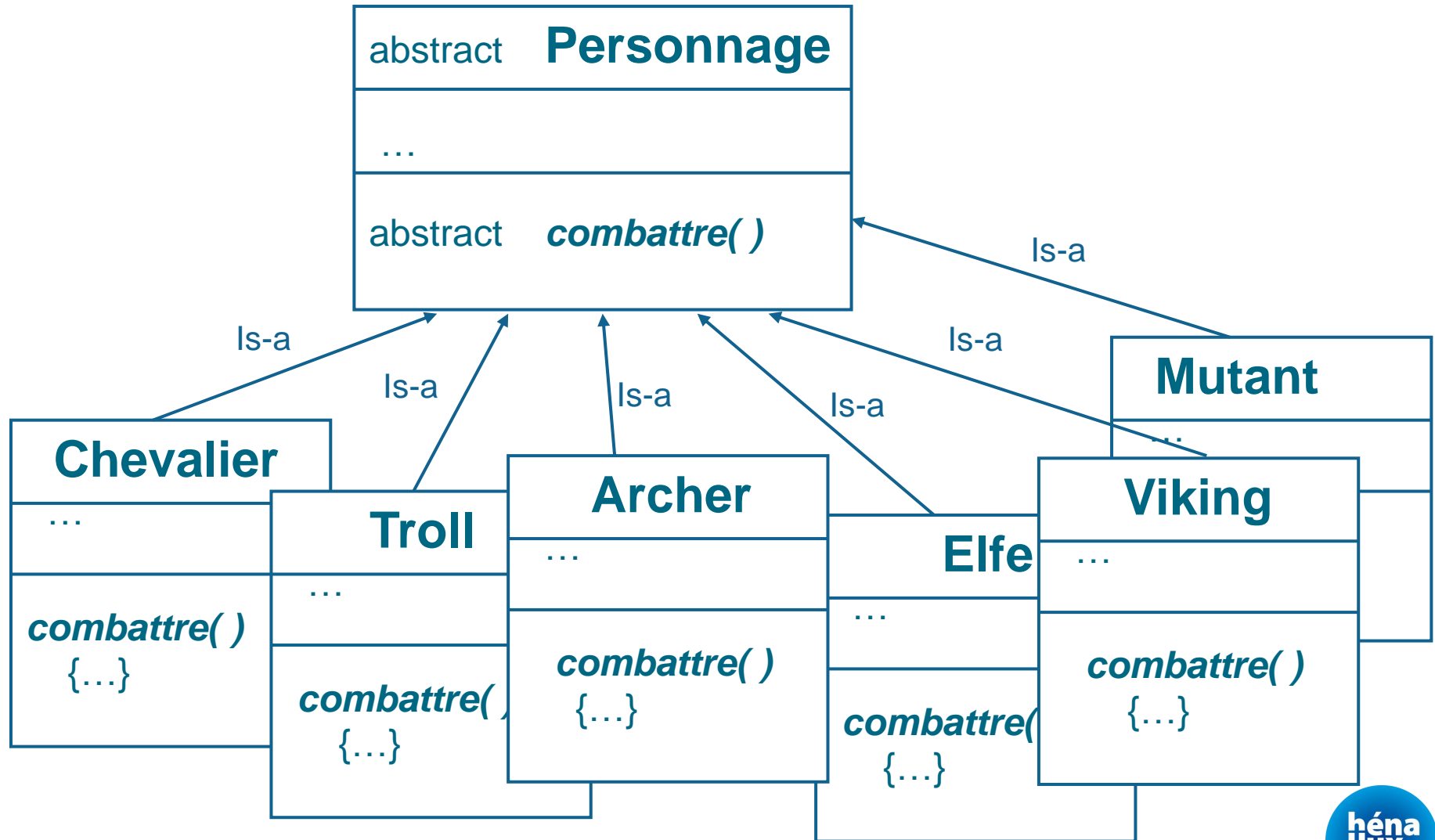
- Le placer dans des **interfaces** + classes qui implémentent ces interfaces
- Préférer la composition (**lien a-un**) à l'héritage (lien est-un)

# Strategy Pattern

## Légende



# Sans Design Pattern Stratégie



# 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...*

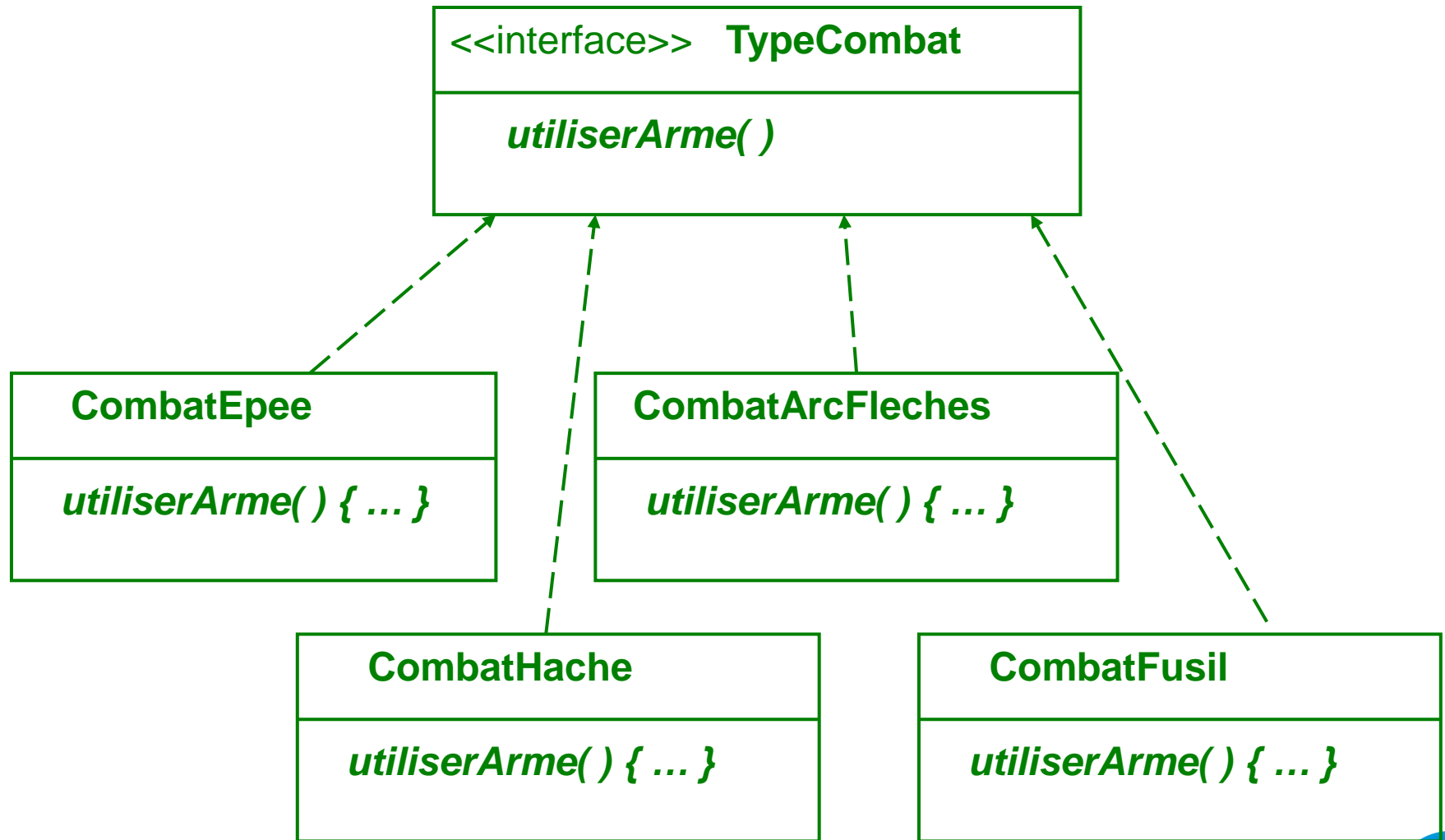
Le même comportement sera donc implémenté plusieurs fois

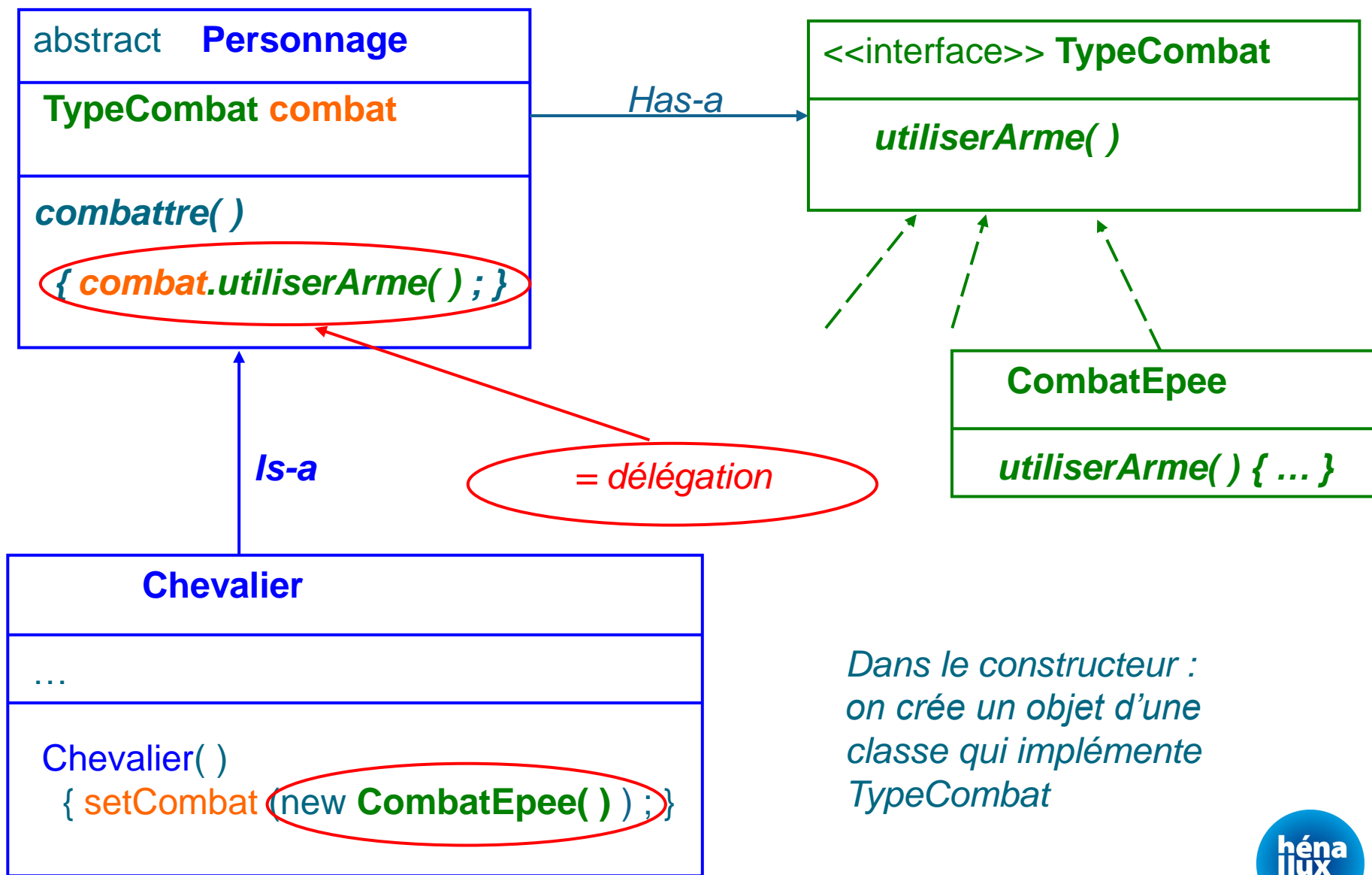
## Conclusion

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 Strategy





Dans le constructeur :  
on crée un objet d'une  
classe qui implémente  
TypeCombat



# Design Patterns

1. Strategy Pattern
2. Factory Pattern

# 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)

# Factory Pattern

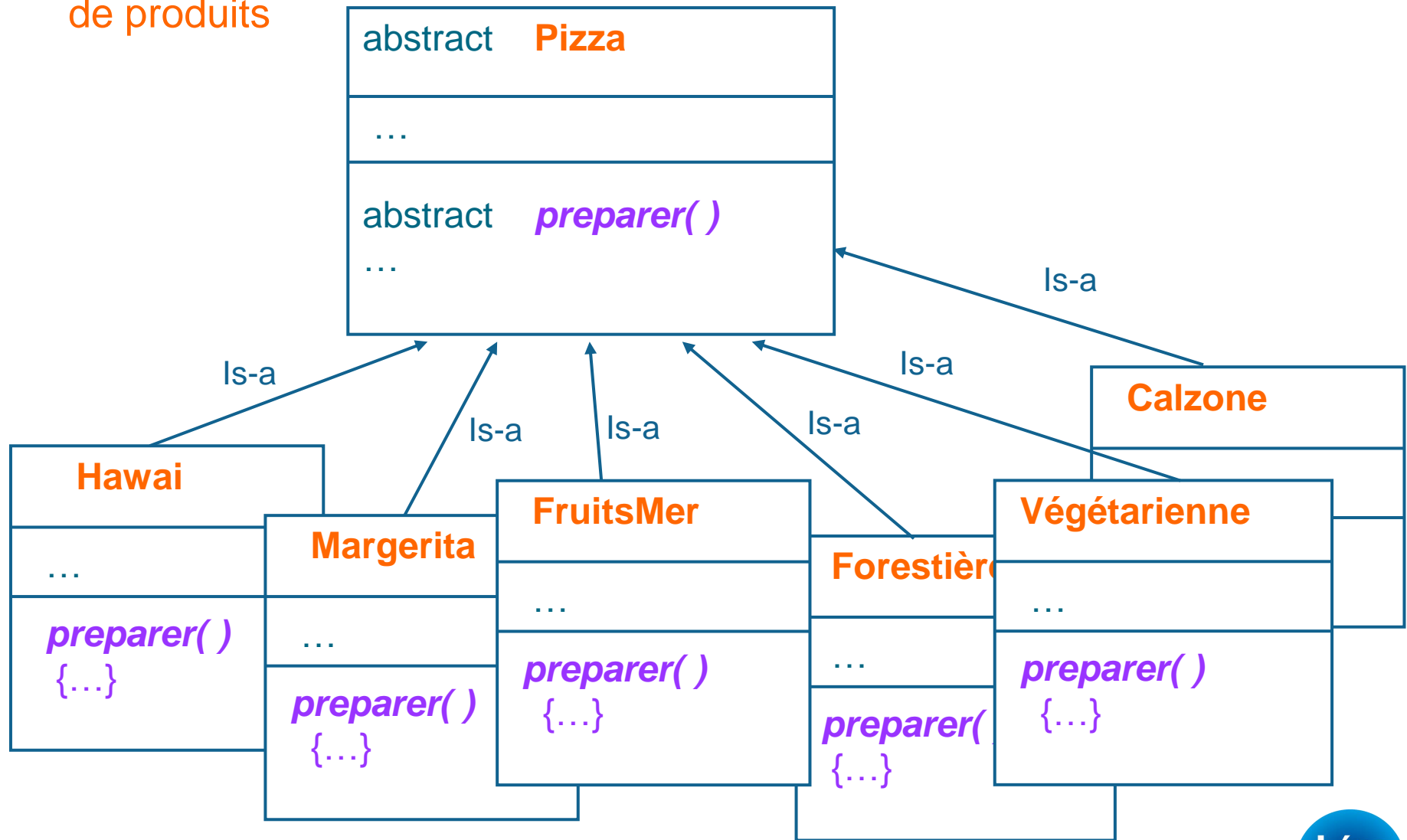
## Produit

*abstract* class **Pizza**

nom  
pate  
sauce  
garniture [ ]

*abstract* **preparer( )**  
cuire ( )  
couper ( )  
emballer ( )

## Hiérarchie de produits



# Factory Pattern

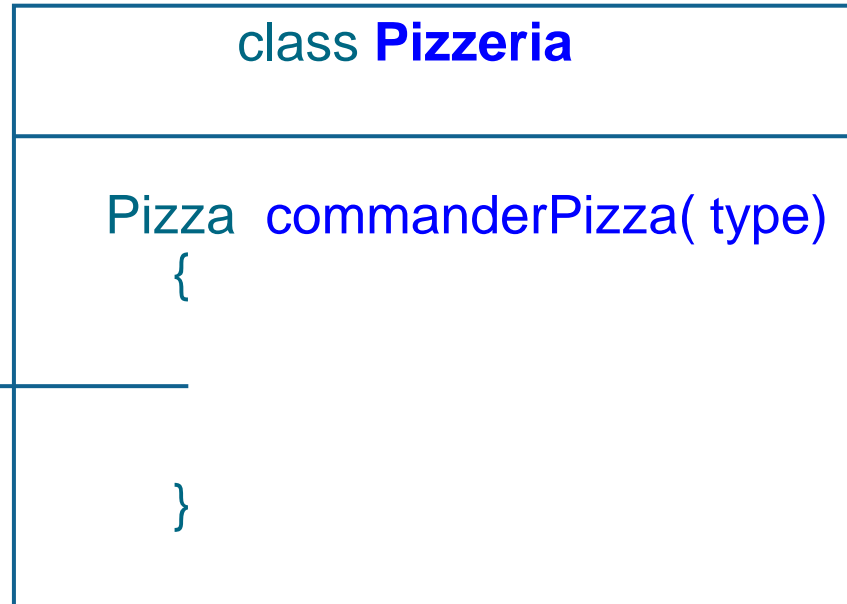
## Exemple

```
public class Hawai extends Pizza {  
    public void preparer( ) {  
        setNom("pizza Hawai");  
        setPate("pâte fine");  
        setSauce("sauce tomatée");  
        setGarniture([ "jambon","mozzarella","ananas" ]);  
    }  
}
```

# Factory Pattern

Créateur de produits

Créer une pizza  
en fonction du  
type demandé



# Factory Pattern

```
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;
```

```
}
```

# Factory Pattern

```
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;  
    }  
}
```



# Factory Pattern

## 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**

# Factory Pattern

Fabrique de produits

```
class FabriqueDePizzas
```

```
    Pizza creerPizza( type)  
    {
```

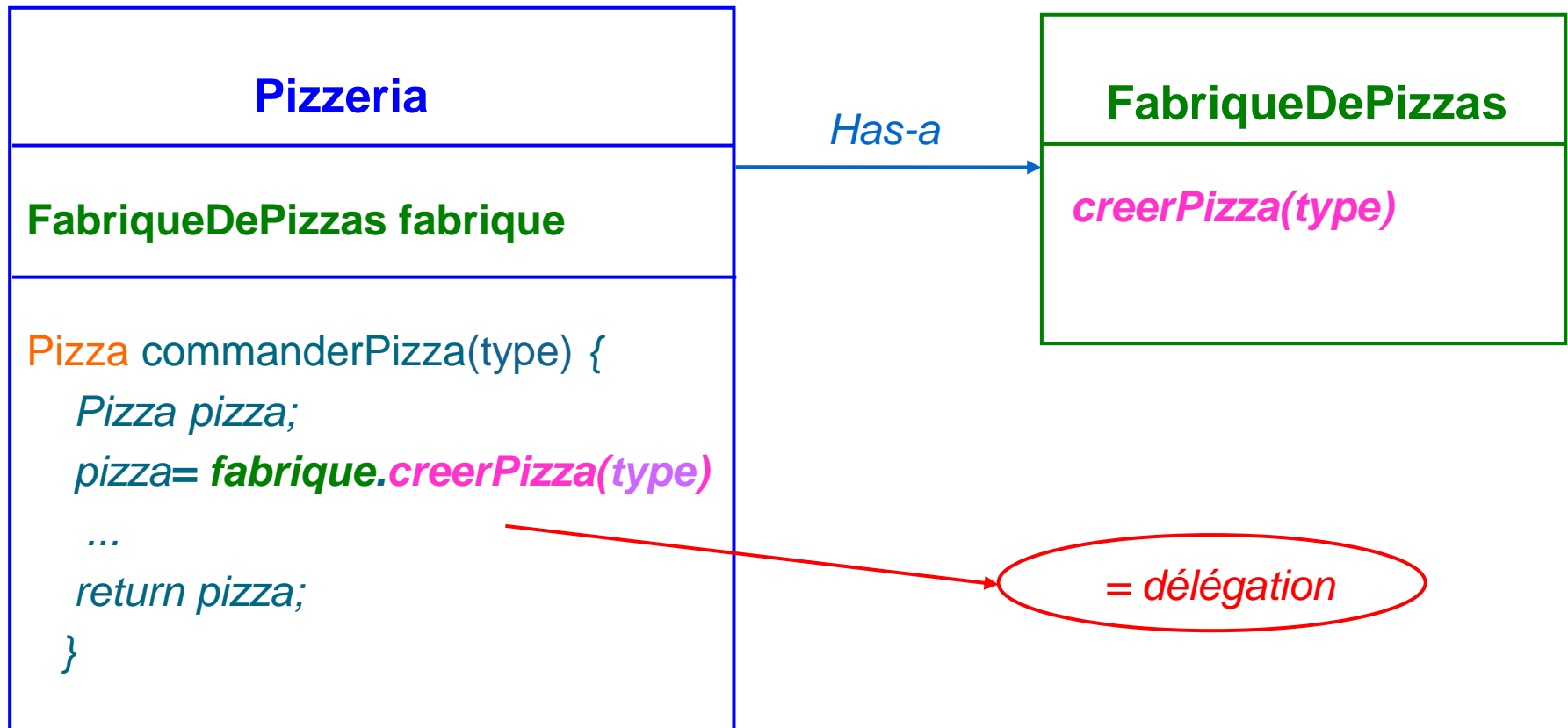
Créer une pizza

```
    }
```

# Factory Pattern

```
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;  
    }  
}
```

# Factory Pattern



N.B. Un objet **FabriqueDePizzas** doit être fourni au constructeur de **Pizzeria** !

# Factory Pattern

```
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( );  
    }  
}
```

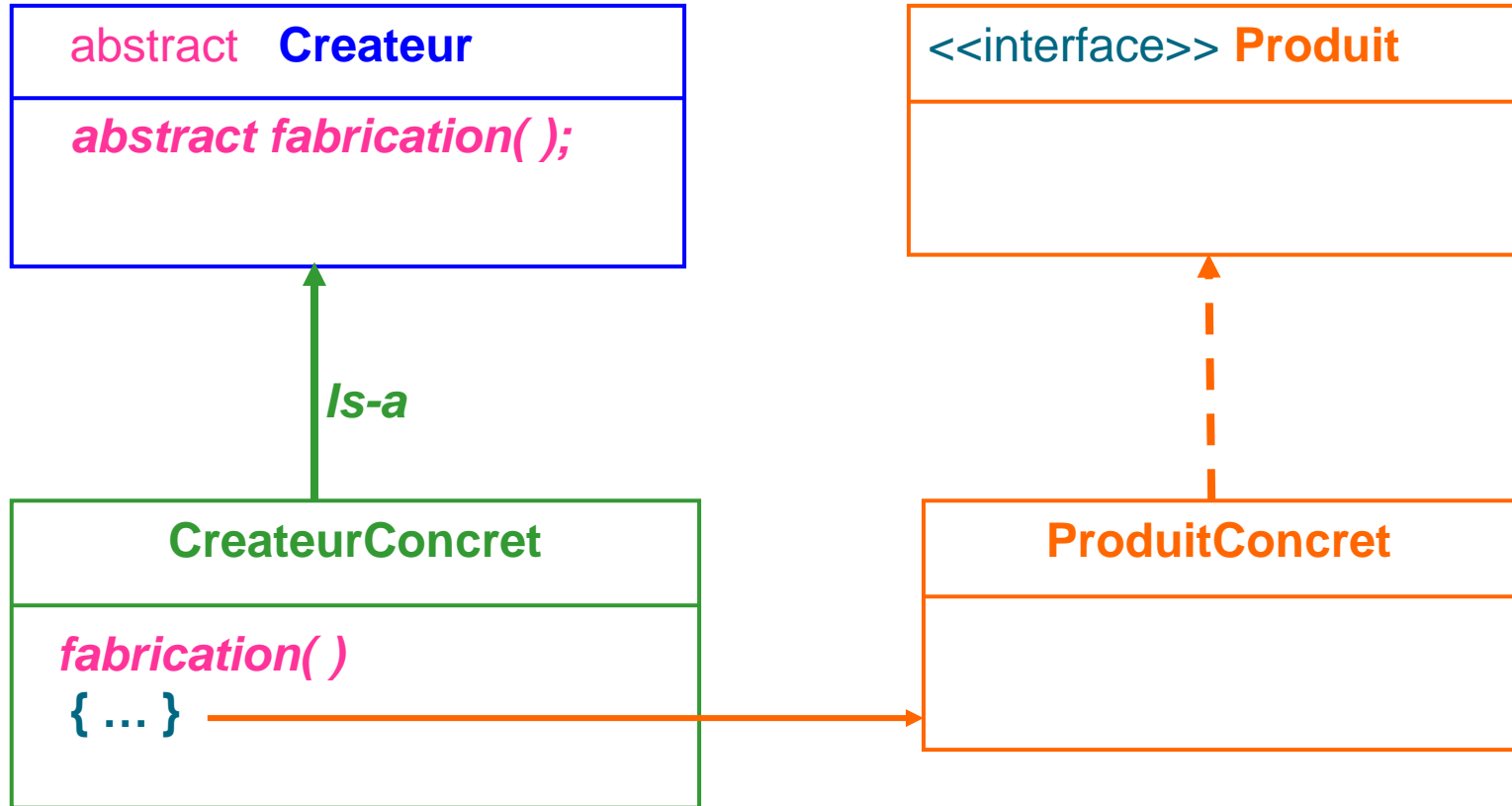
*Pour créer la pizza en fonction du type :  
délégation à la fabrique*

# Factory Pattern

Version 2 : *Pattern Fabrication*

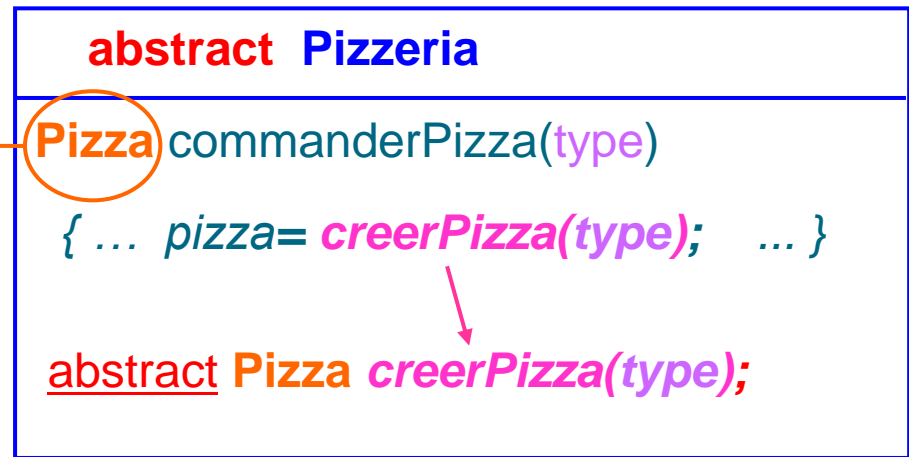
*Une classe créateur abstraite qui délègue l'instanciation des objets produits à ses sous-classes*

# Factory Pattern



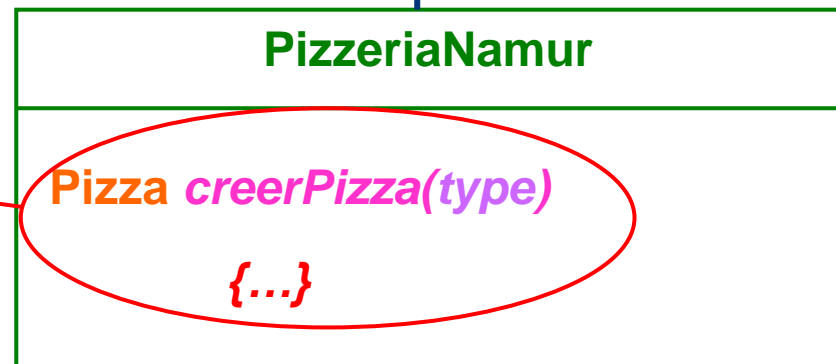
# Factory Pattern

*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)*





# Factory Pattern

```
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);  
}
```

Toute sous-classe devra implémenter  
cette méthode creerPizza



# Design Patterns

1. Strategy Pattern
2. Factory Pattern
3. Prototype Pattern

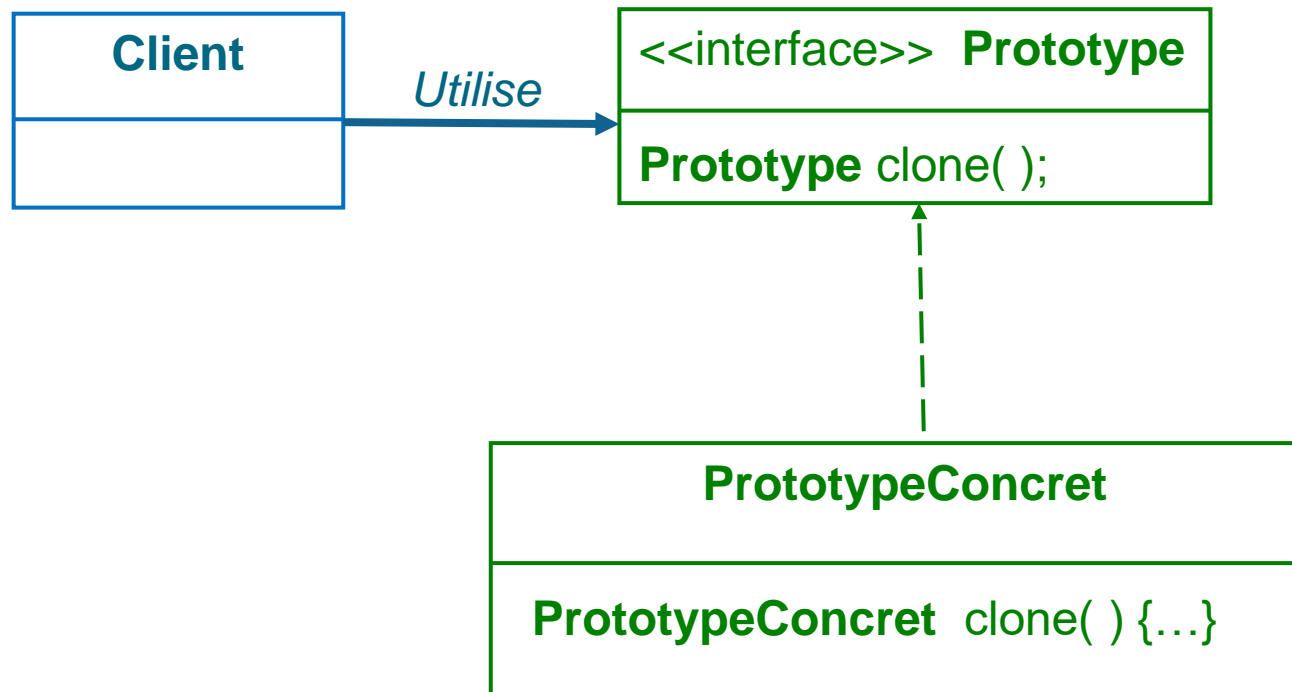
# Prototype Pattern

## 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

# Prototype Pattern



# Prototype Pattern

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

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

# Prototype Pattern

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

# Design Patterns

- 1. Strategy Pattern**
- 2. Factory Pattern**
- 3. Prototype Pattern**
- 4. Singleton Pattern**

# 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



# Singleton Pattern

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

# Singleton Pattern

## Adaptation

```
public class ClassX {  
    ...  
    private static ClassY uniqueInstance;  
    public static ClassY getInstance( ) {  
        if (uniqueInstance == null)  
            { ... }           // Créer une instance de ClassY  
        return uniqueInstance;  
    }  
    ...                       // Autres méthodes  
}
```

*2 classes différentes*

*Même classe*

# Singleton Pattern

## Utilisation

Singleton singleton = **Singleton.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

## Data Access

BookDBAccess

SingletonConnexion

# Singleton Pattern

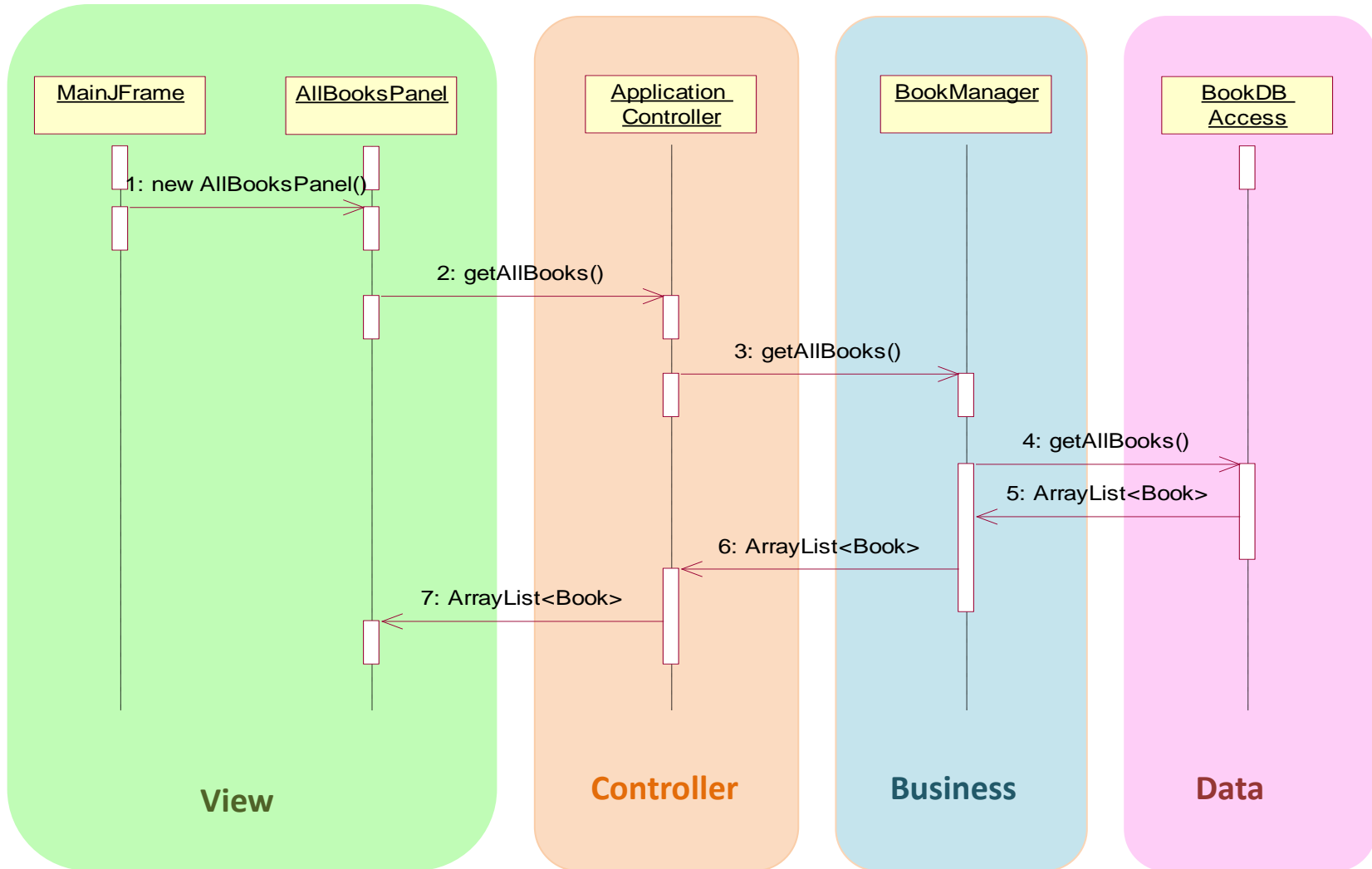
## Gestion de la connexion unique

```
public class SingletonConnexion {  
    private static Connection connexionUnique;
```

*Classe existante ⇒ impossible à modifier  
⇒ Obligation de créer une autre classe  
pour stocker et gérer le singleton  
Connection*

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

## Exemple : Afficher la liste de tous les livres



# Singleton Pattern

```
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 ArrayList de livres  
  
    }  
}
```

# Design Patterns

1. Strategy Pattern
2. Factory Pattern
3. Prototype Pattern
4. Singleton Pattern
5. Data Access Object Pattern



# 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

# Data Access Object Pattern

1. Encapsuler les accès aux données  $\Rightarrow$  les placer dans une interface  
**= Interface public du DAO**
2. Créer des classes qui implémentent ces interfaces  
**= Implémentations du DAO**
  - $\hookrightarrow$  connaissent la source de données à laquelle se connecter  
(ex: BD, XML, Web Service, ...)
  - $\hookrightarrow$  spécifiques à une source de données

# Data Access Object Pattern

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.

# Data Access Object Pattern

## 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

# Data Access Object Pattern

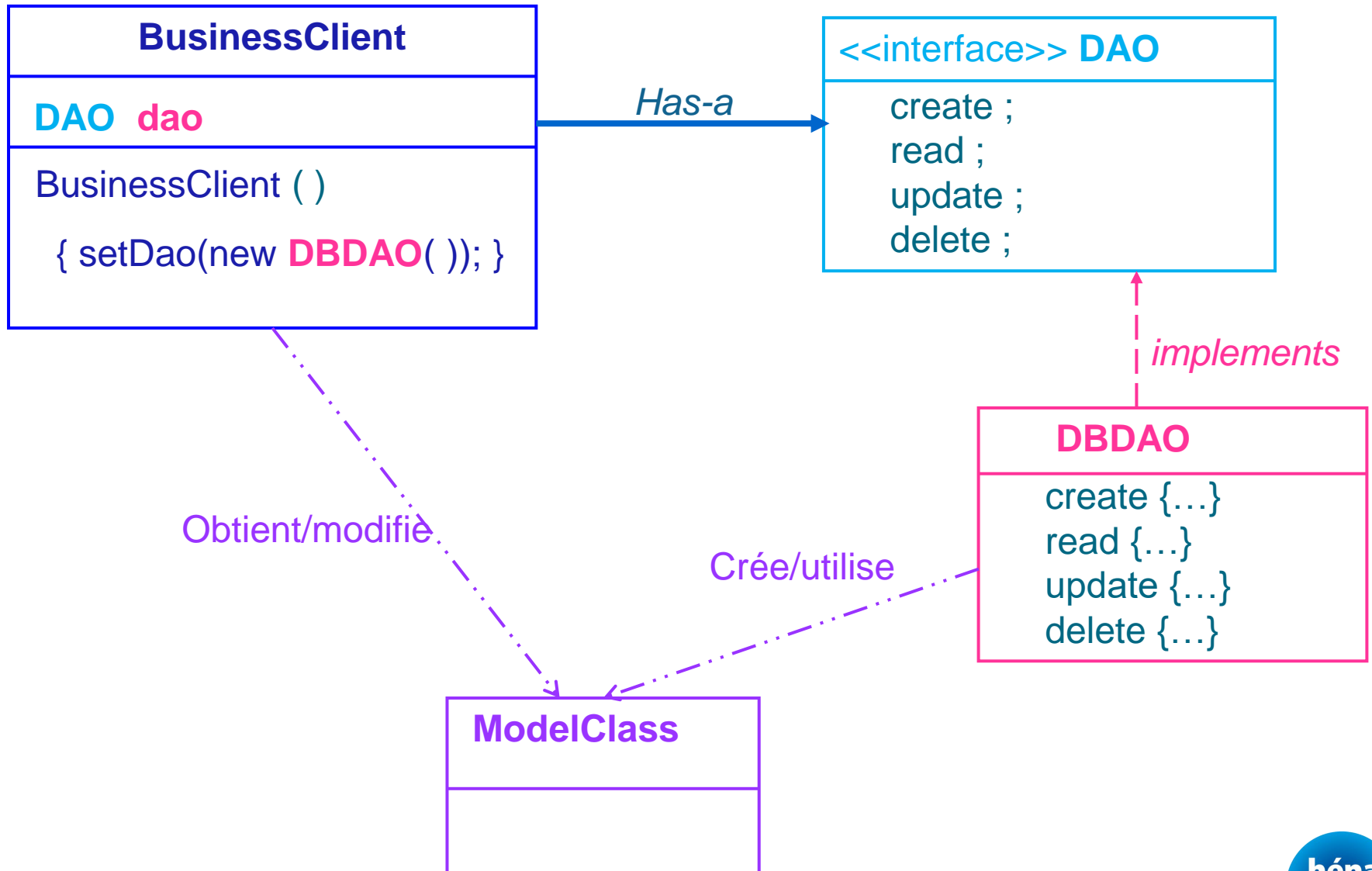
Via l'encapsulation du code des opérations **CRUD**

**C**reate

**R**ead

**U**ppdate

**D**elete



## User Interface

MainJFrame

NewBookPanel

AllBooksPanel

AllBooksModel

## Model

## Controller

ApplicationController

Book

## Business Logic

BookManager

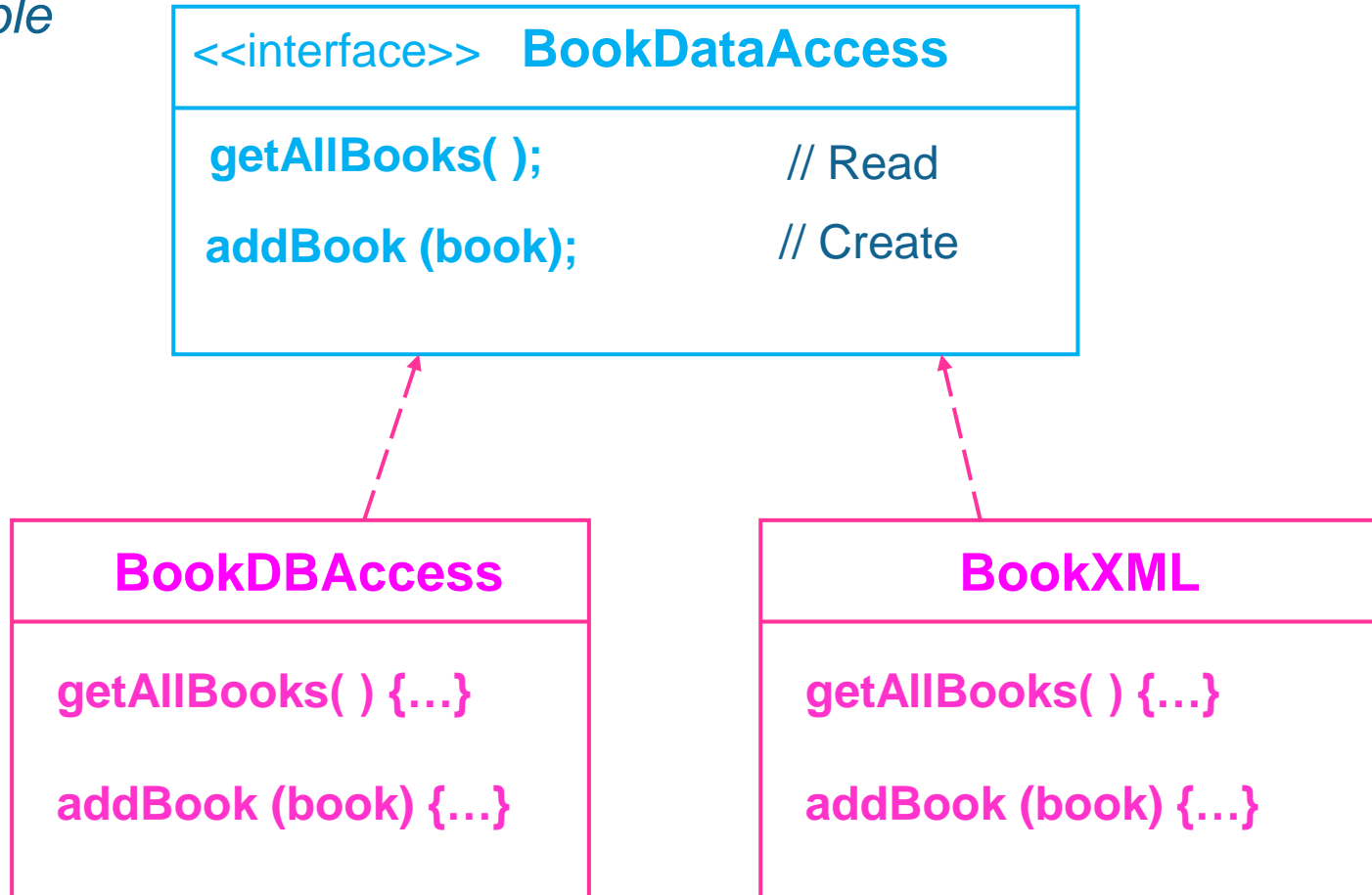
## Data Access

BookDBAccess

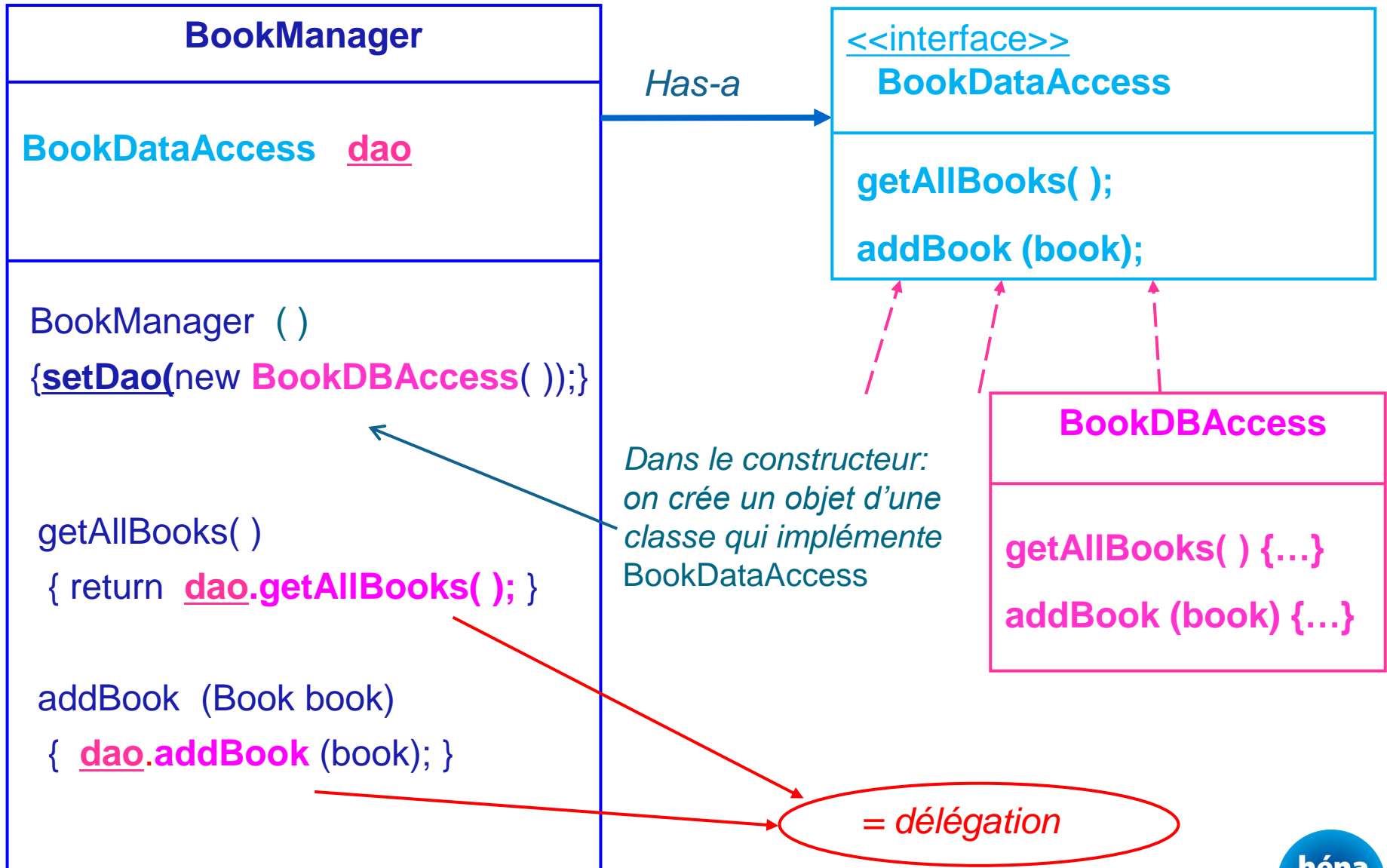
SingletonConnexion

# Data Access Object Pattern

## Exemple







# Design Patterns

- 1. Strategy Pattern**
- 2. Factory Pattern**
- 3. Prototype Pattern**
- 4. Singleton Pattern**
- 5. Data Access Object Pattern**
- 6. Iterator Pattern**

# Iterator Pattern

## Objectif du pattern *itérateur*

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

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

## *Exemples d'implémentation de collection*

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

# Iterator Pattern

⇒ Il faut pouvoir

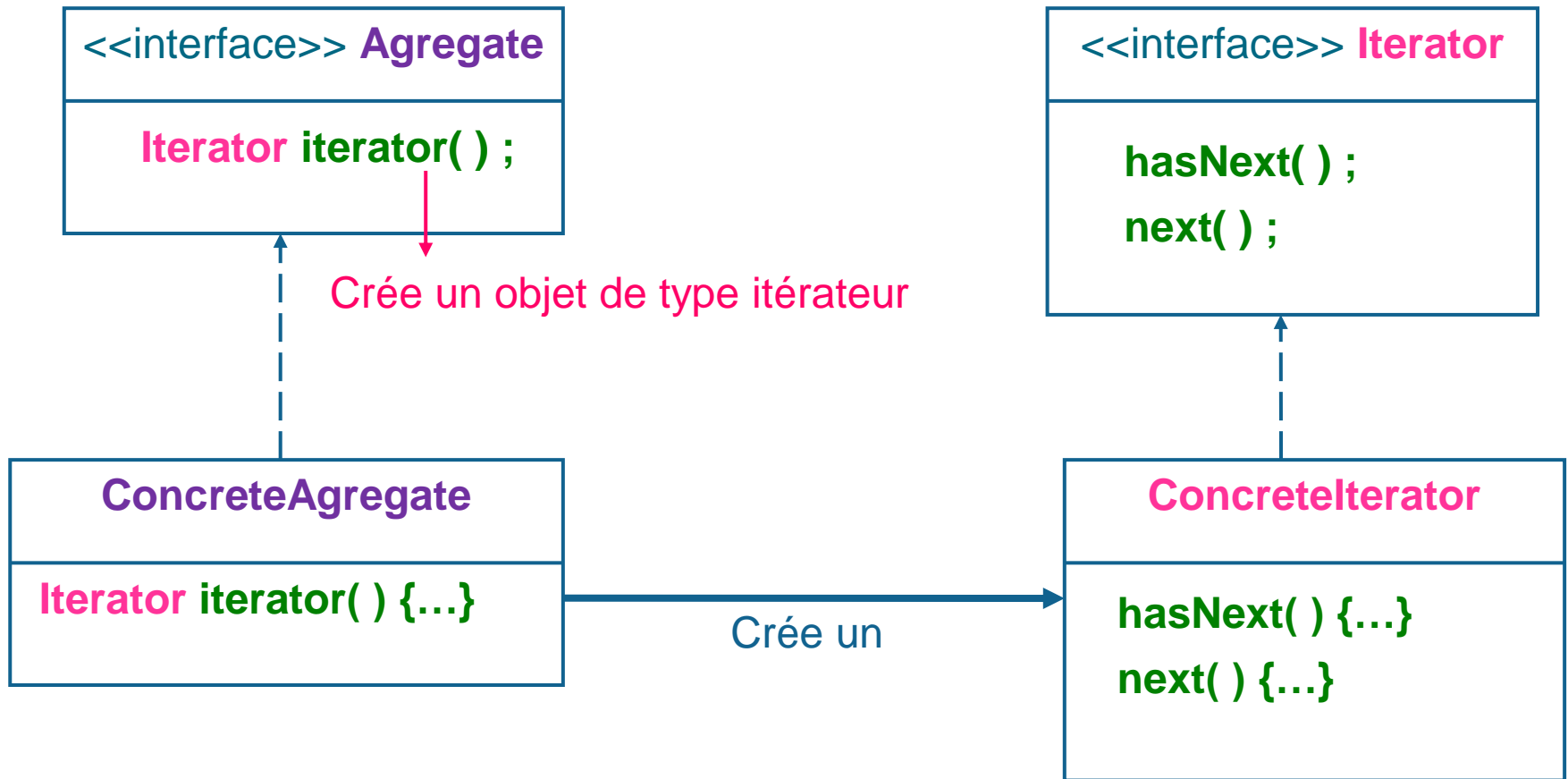
- demander l'élément **suisant**
- savoir s'il y a **encore** des éléments dans la collection

⇒ 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

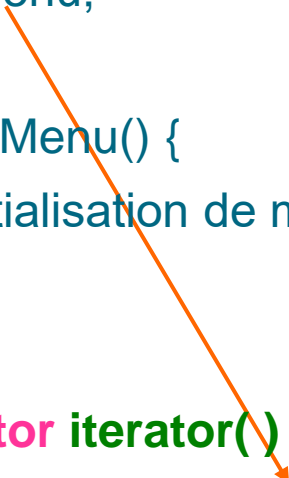
# Iterator Pattern



# Iterator Pattern

```
public interface Agregate {  
    Iterator iterator( );  
}
```

```
public class RestaurantMenu implements Agregate {  
    private String[ ] menu;  
  
    public RestaurantMenu() {  
        menu = ... // initialisation de menu, par exemple via accès à une BD  
    }  
  
    public MenuIterator iterator() {  
        return new MenuIterator(menu);  
    }  
}
```



# Iterator Pattern

```
public class MenuIterator implements Iterator {  
    private String[ ] menu ;  
    private int position ;  
  
    public MenuIterator(String[ ] menu) {  
        this.menu = menu ;  
        position = 0 ;  
    }  
  
    public Object next( ) {  
        return menu[position++] ;  
    }  
  
    public boolean hasNext( ) {  
        return !(position >= menu.length || menu[position]==null)  
    }  
}
```

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

# Iterator Pattern

```
public class IteratorDesignPattern {  
  
    public static void main(String[ ] args) {  
  
        RestaurantMenu restoMenu = new RestaurantMenu() ;  
  
        MenuIterator menuIterator = restoMenu.iterator( ) ;  
  
        while ( menuIterator.hasNext( ) ) {  
            System.out.println(menuIterator.next( ) ) ;  
        }  
    }  
}
```



# Design Patterns

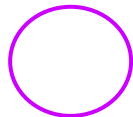
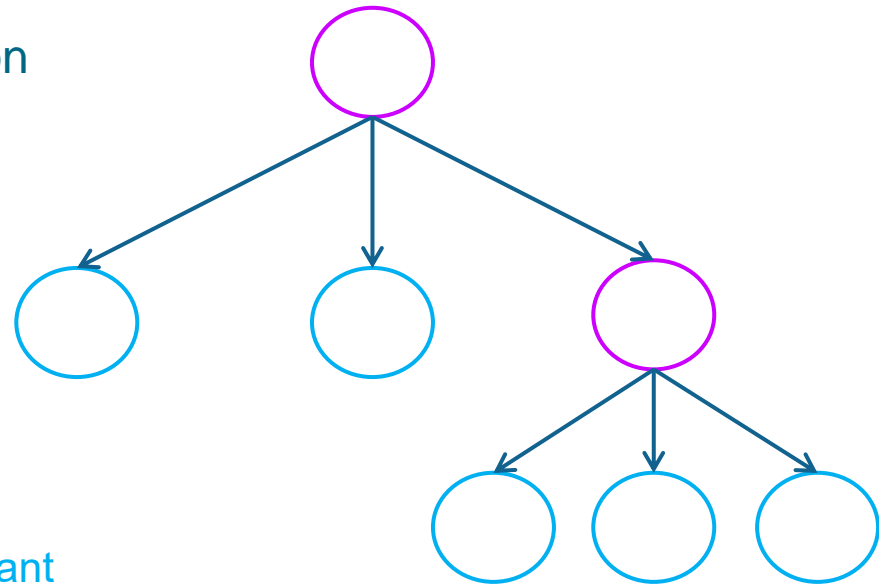
- 1. Strategy Pattern**
- 2. Factory Pattern**
- 3. Prototype Pattern**
- 4. Singleton Pattern**
- 5. Data Access Object Pattern**
- 6. Iterator Pattern**
- 7. Composite Pattern**

# Composite Pattern

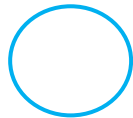
## Objectif du pattern **composition**

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



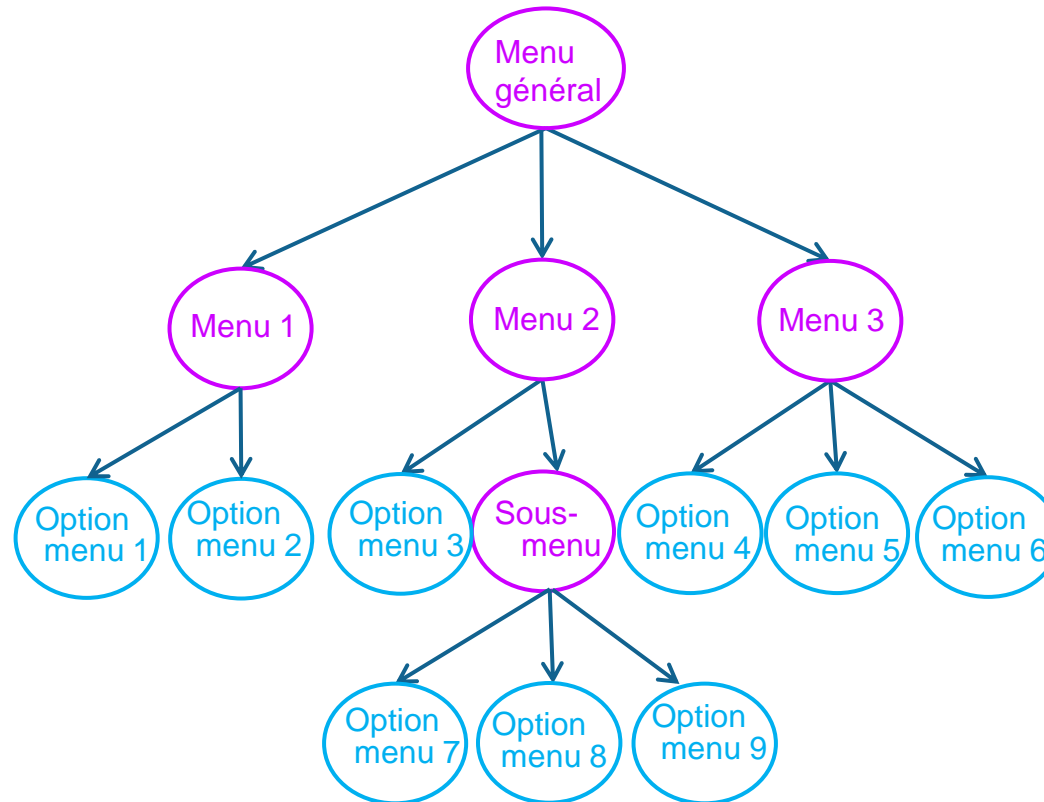
Nœud: élément qui a des enfants



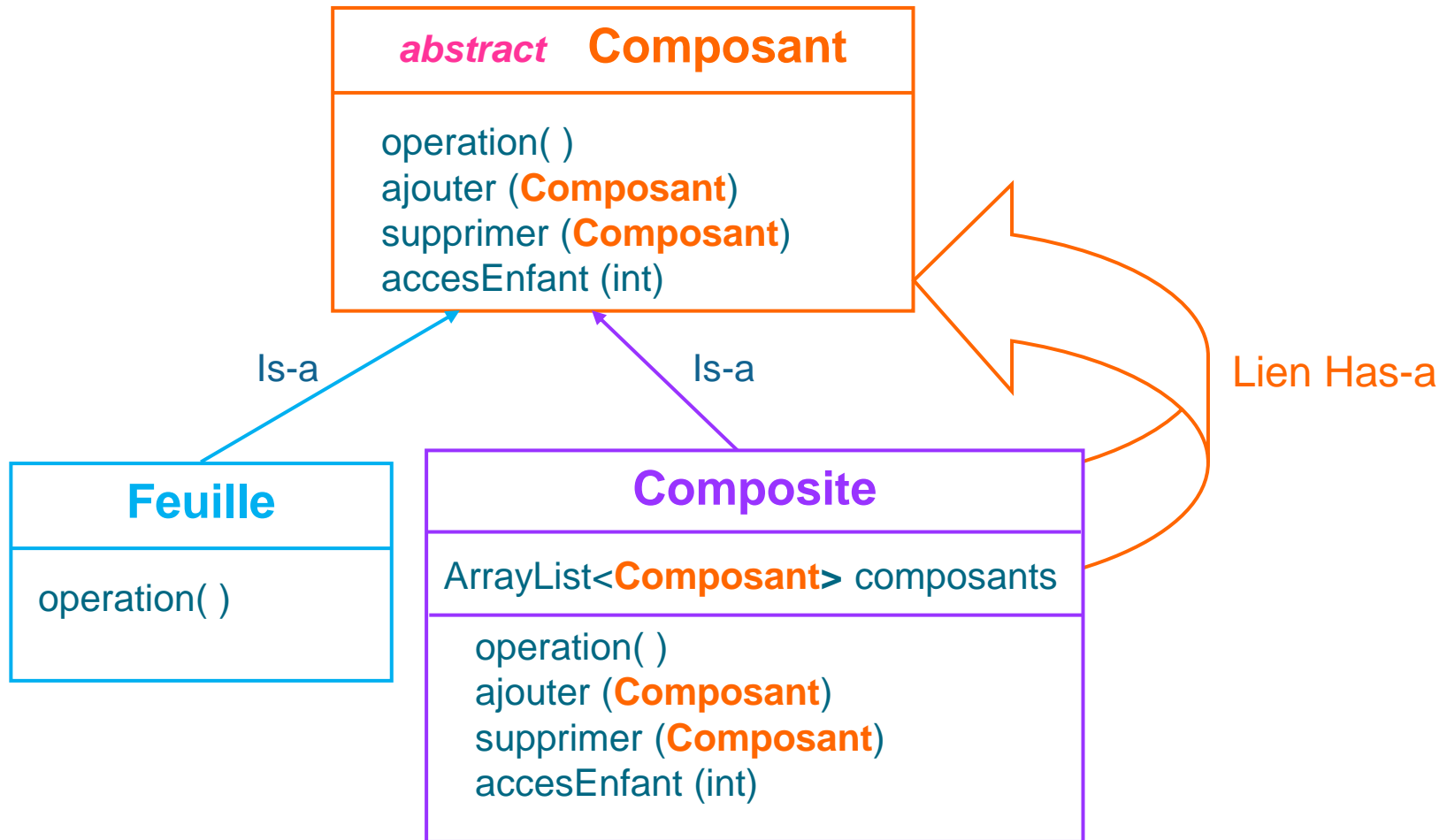
Feuille : élément qui n'a pas d'enfant

# Composite Pattern

Exemple : arborescence de menus



# Composite Pattern



# Composite Pattern

N.B. Certaines méthodes héritées de Composant n'ont aucun sens pour la classe Feuille

*Exemples : ajouter (Composant), supprimer (Composant) et accesEnfant (int).*

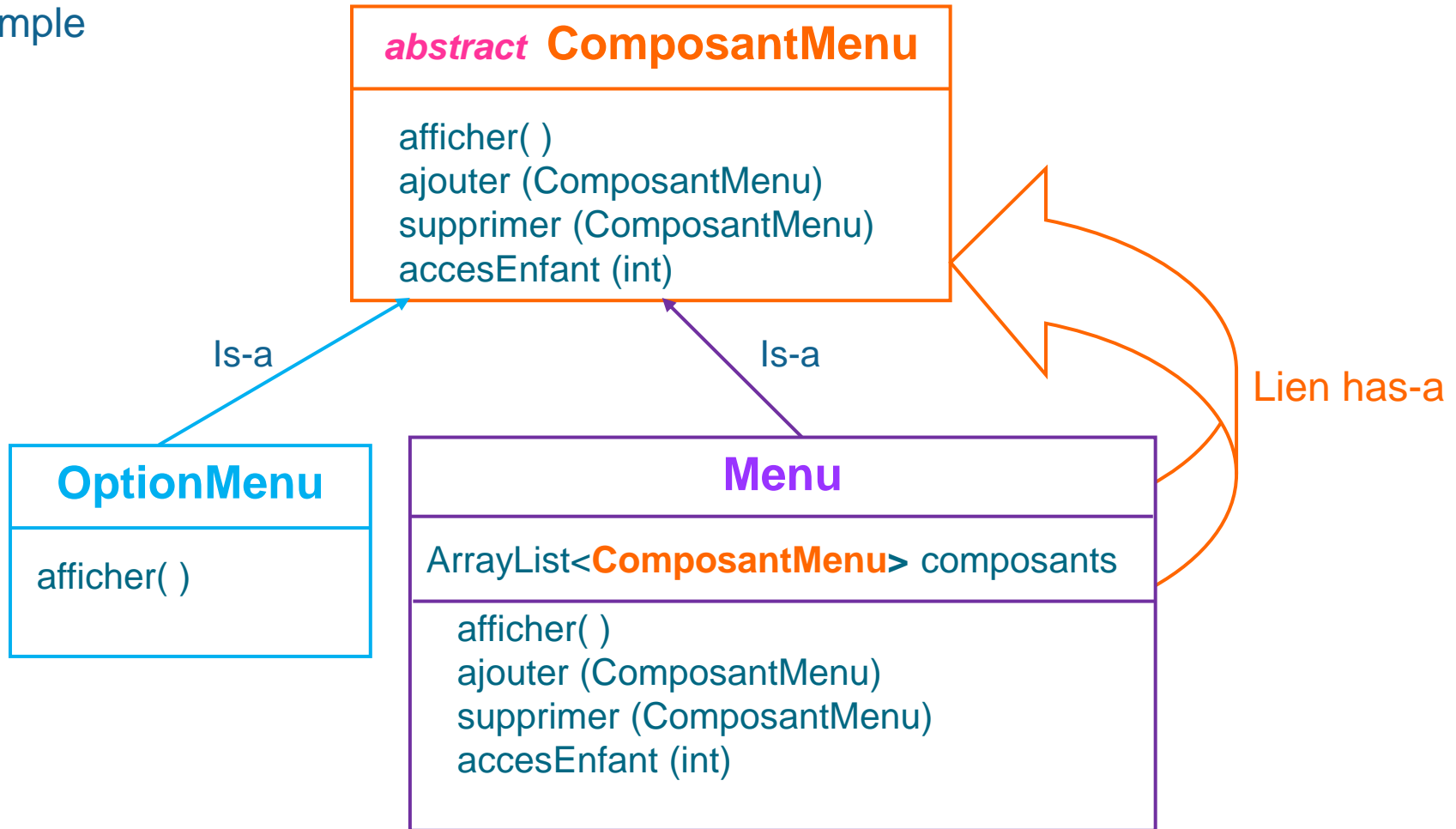
⇒ Implémentation par défaut de ces méthodes dans la classe abstraite

**Composant**

*Ex: Lever des exceptions du type UnsupportedOperationException*

# Composite Pattern

Exemple



# Composite Pattern

```
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( );  
    }  
}
```

# Composite Pattern

```
public class OptionMenu extends ComposantMenu {  
    ... // constructeur  
  
    public void afficher( ) {  
        ... // Affichage de l'option de menu  
    }  
}
```



# Composite Pattern

```
public class Menu extends ComposantMenu {  
    private ArrayList <ComposantMenu> composants;  
    ... // constructeur  
    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( );  
    }  
}
```

# Design Patterns

1. Strategy Pattern
2. Factory Pattern
3. Prototype Pattern
4. Singleton Pattern
5. Data Access Object Pattern
6. Iterator Pattern
7. Composite Pattern
8. Decorator Pattern

# Decorator Pattern

## Objectif du pattern **décorateur**

Attacher dynamiquement des responsabilités supplémentaires à un objet

### *Exemple*

#### **Carte**

##### Cafés

Colombie

Brésil

Déca

Espresso

##### Suppléments

Lait

Chocolat

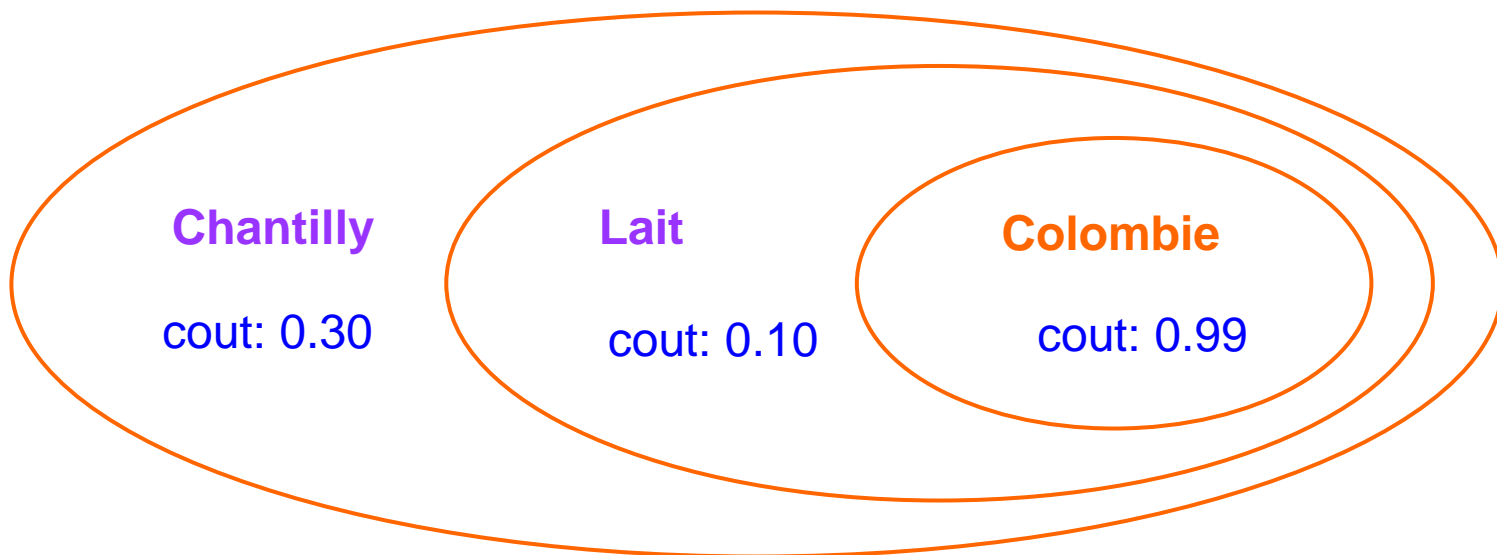
Chantilly

On peut choisir un **café**

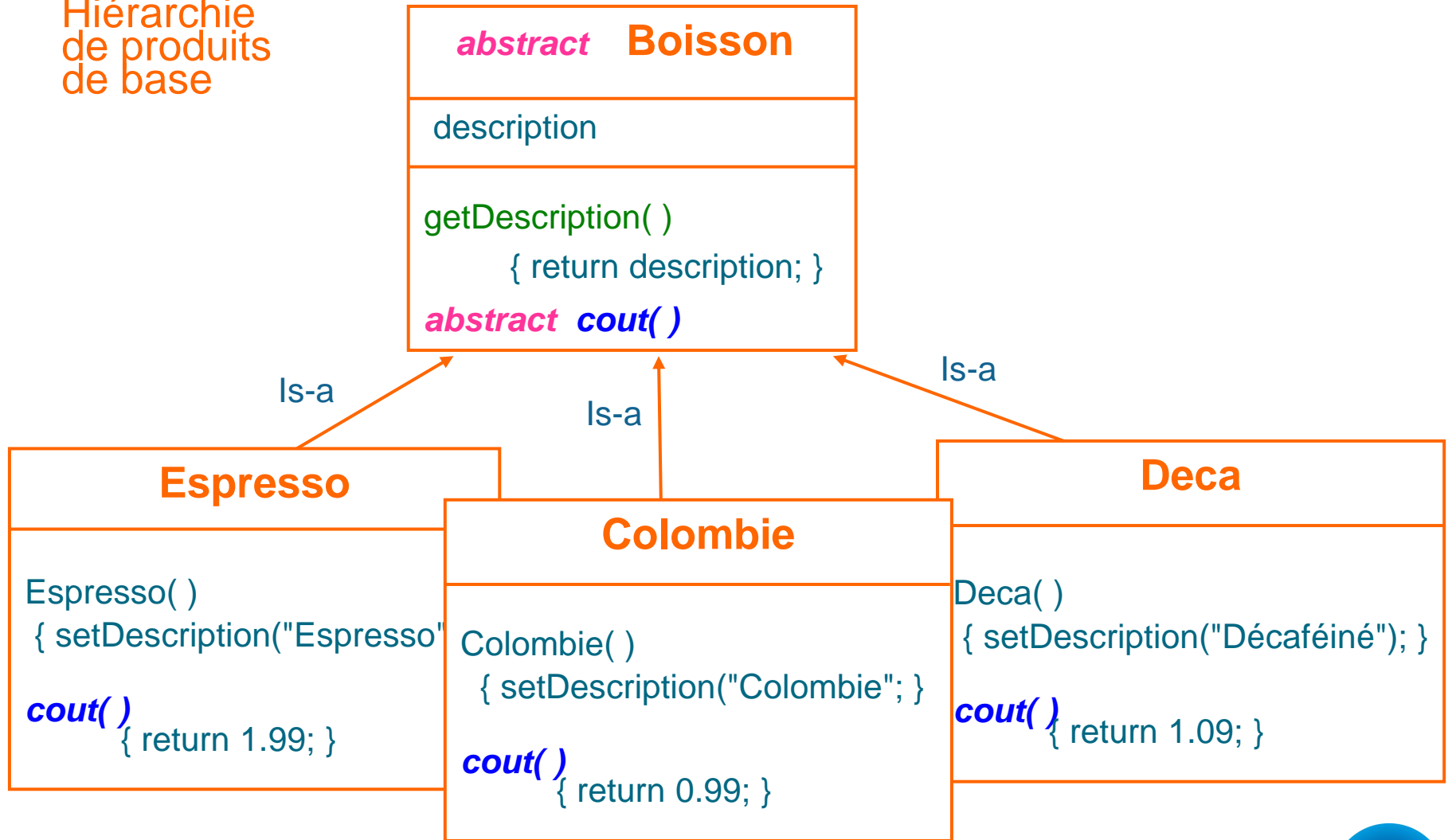
+ un ou plusieurs **supplément(s)**

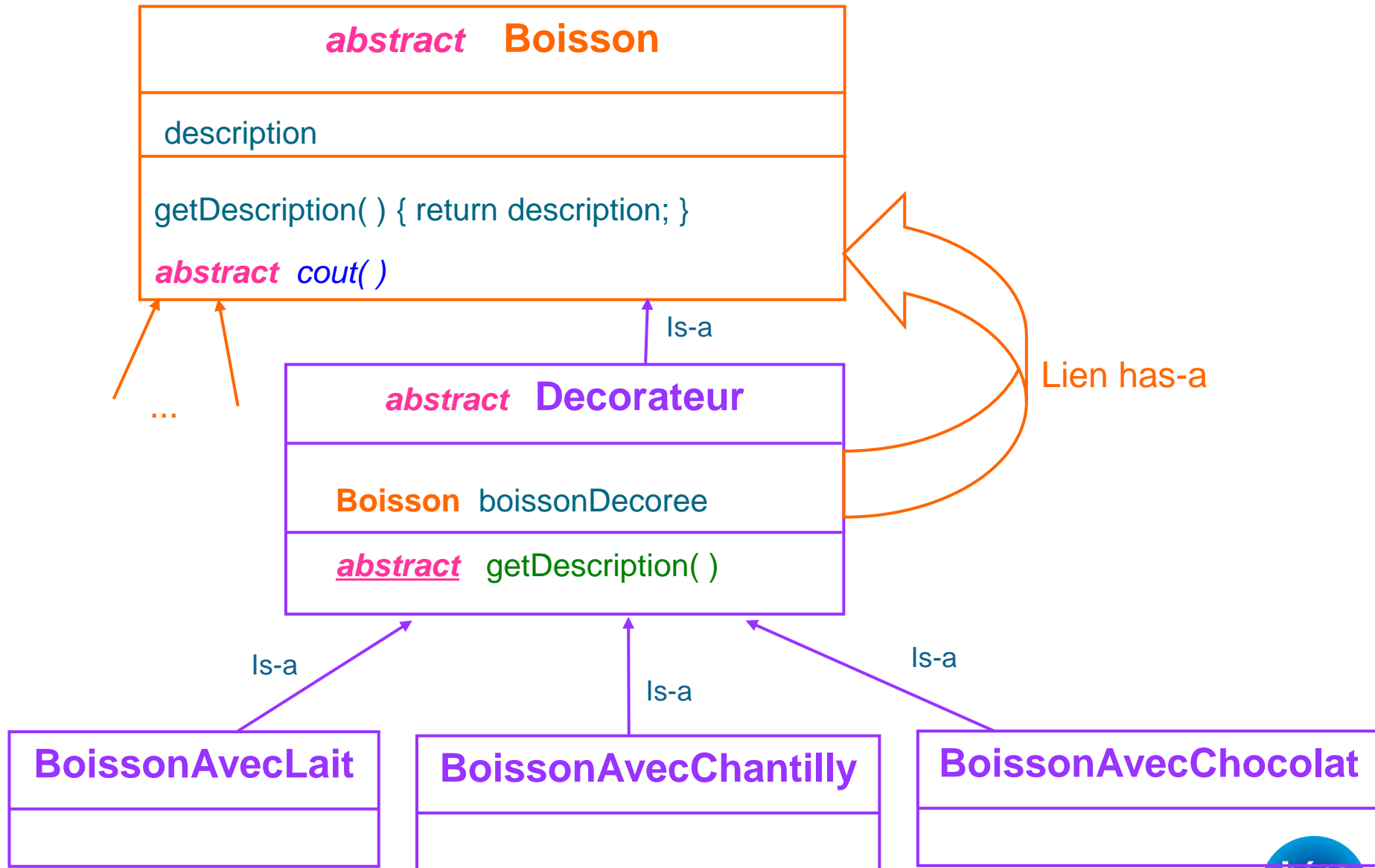
*Exemple : Colombie + Lait + Chantilly*

# Decorator Pattern



## Hiérarchie de produits de base

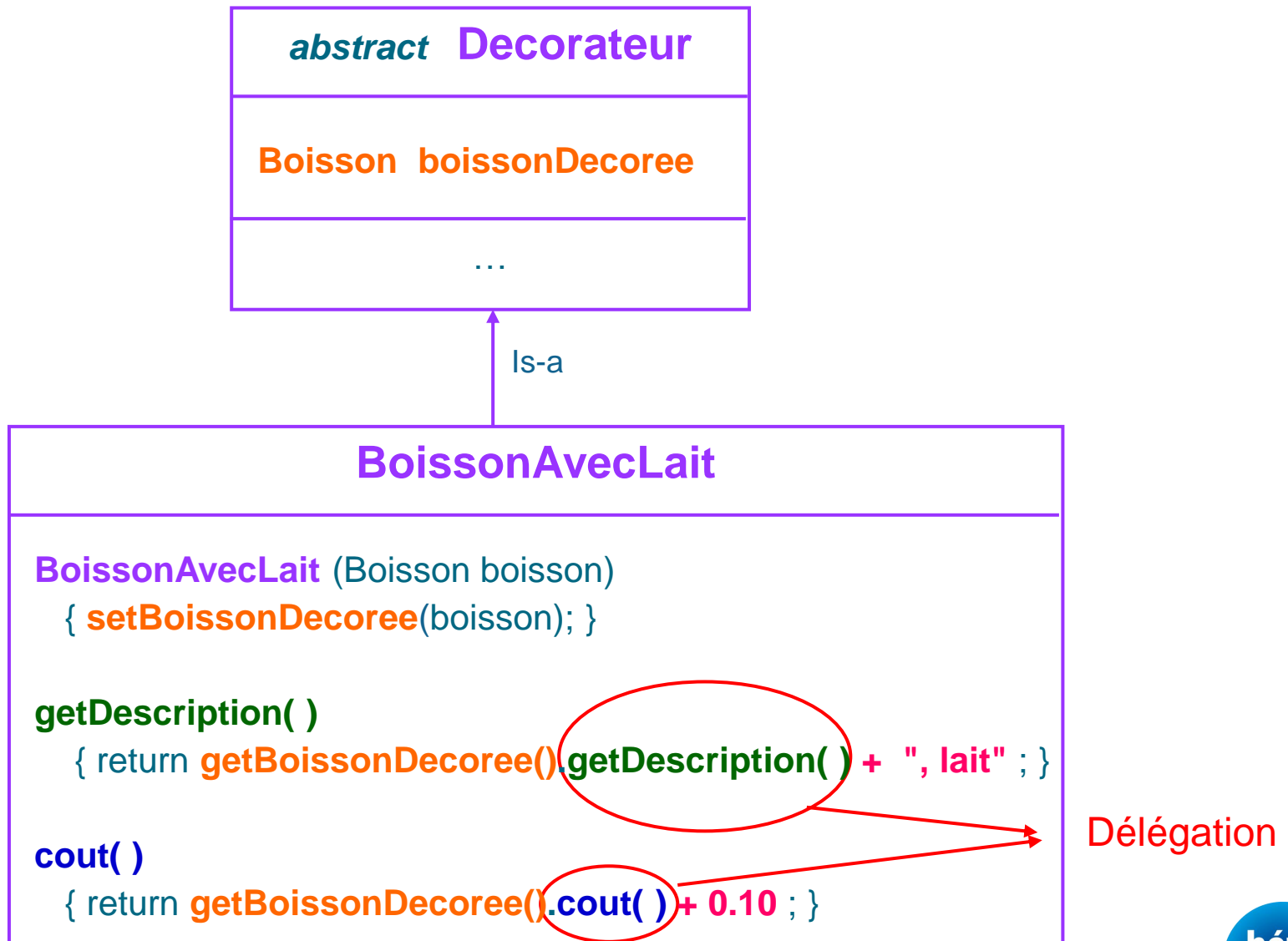




# Decorator Pattern

## 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 **getDescription** pour y inclure la décoration supplémentaire





# Decorator Pattern

*Exemple d'utilisation :*

```
Colombie cafe = new Colombie(...);
```

```
Boisson cafeAuLait = new BoissonAvecLait(cafe);
```

```
Boisson cafeAuLaitEtChantilly = new BoissonAvecChantilly(cafeAuLait);
```

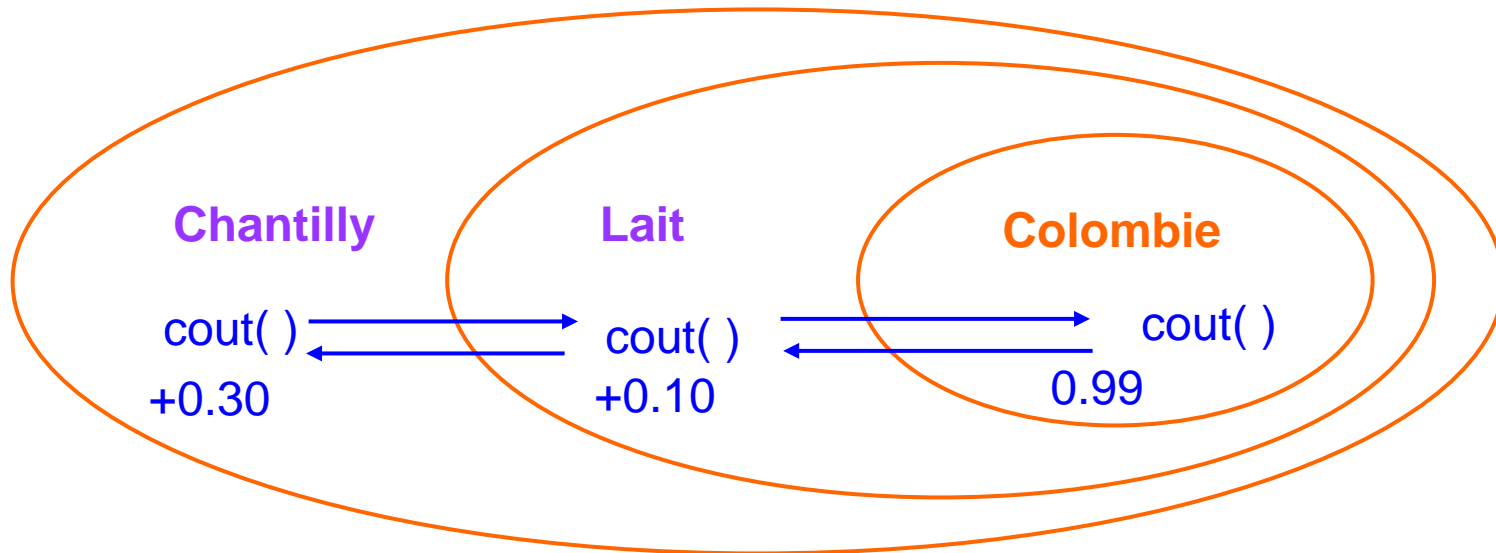
```
System.out.println(cafeAuLaitEtChantilly.getDescription());
```

*⇒ Affiche : Colombie, lait, chantilly*

```
System.out.println(cafeAuLaitEtChantilly.cout());
```

*⇒ Affiche : 1.39*

# Decorator Pattern



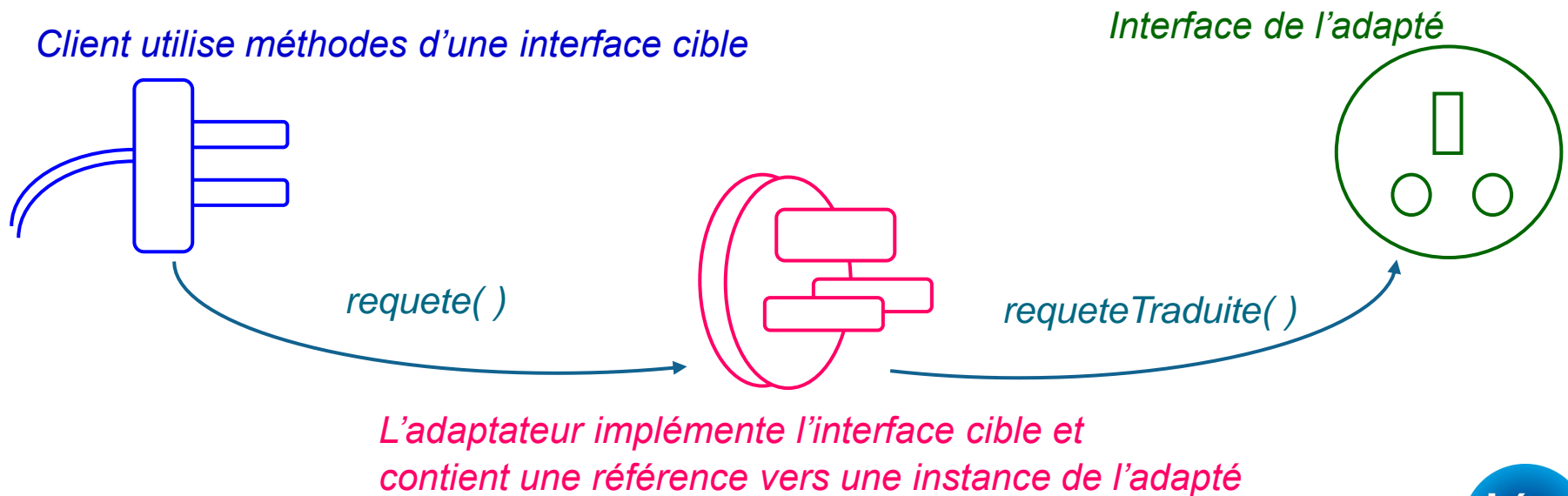
# Design Patterns

1. Strategy Pattern
2. Factory Pattern
3. Prototype Pattern
4. Singleton Pattern
5. Data Access Object Pattern
6. Iterator Pattern
7. Composite Pattern
8. Decorator Pattern
9. Adaptator Pattern

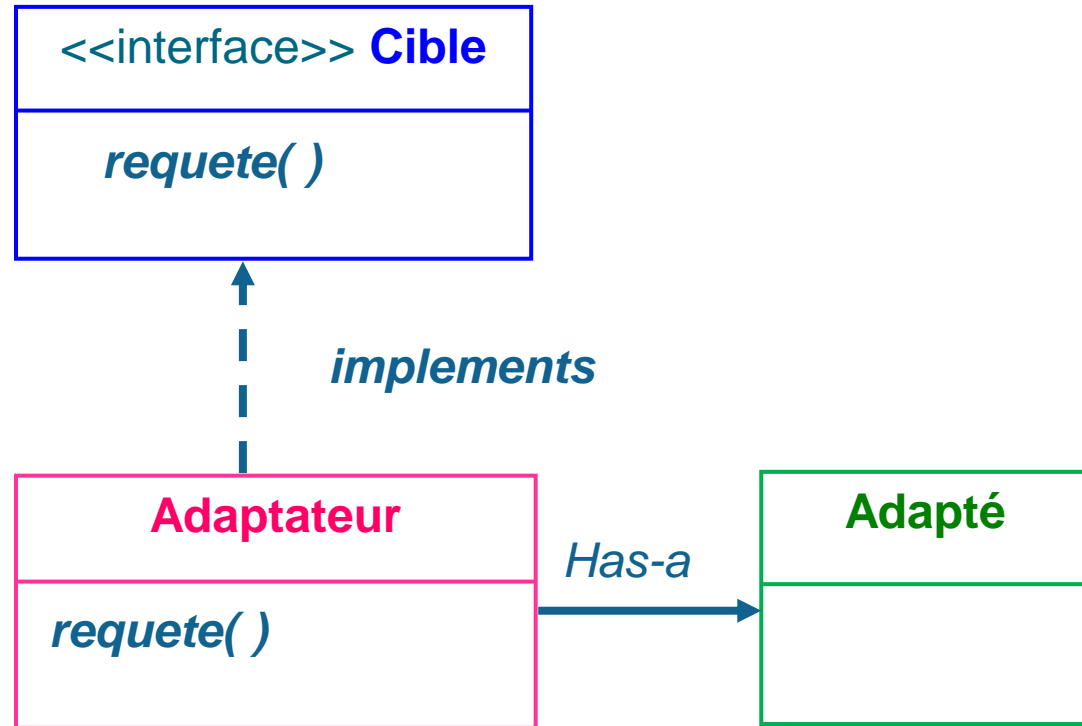
## Objectif du pattern **adaptateur**

Rendre compatible deux interfaces incompatibles

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



# Adaptator Pattern



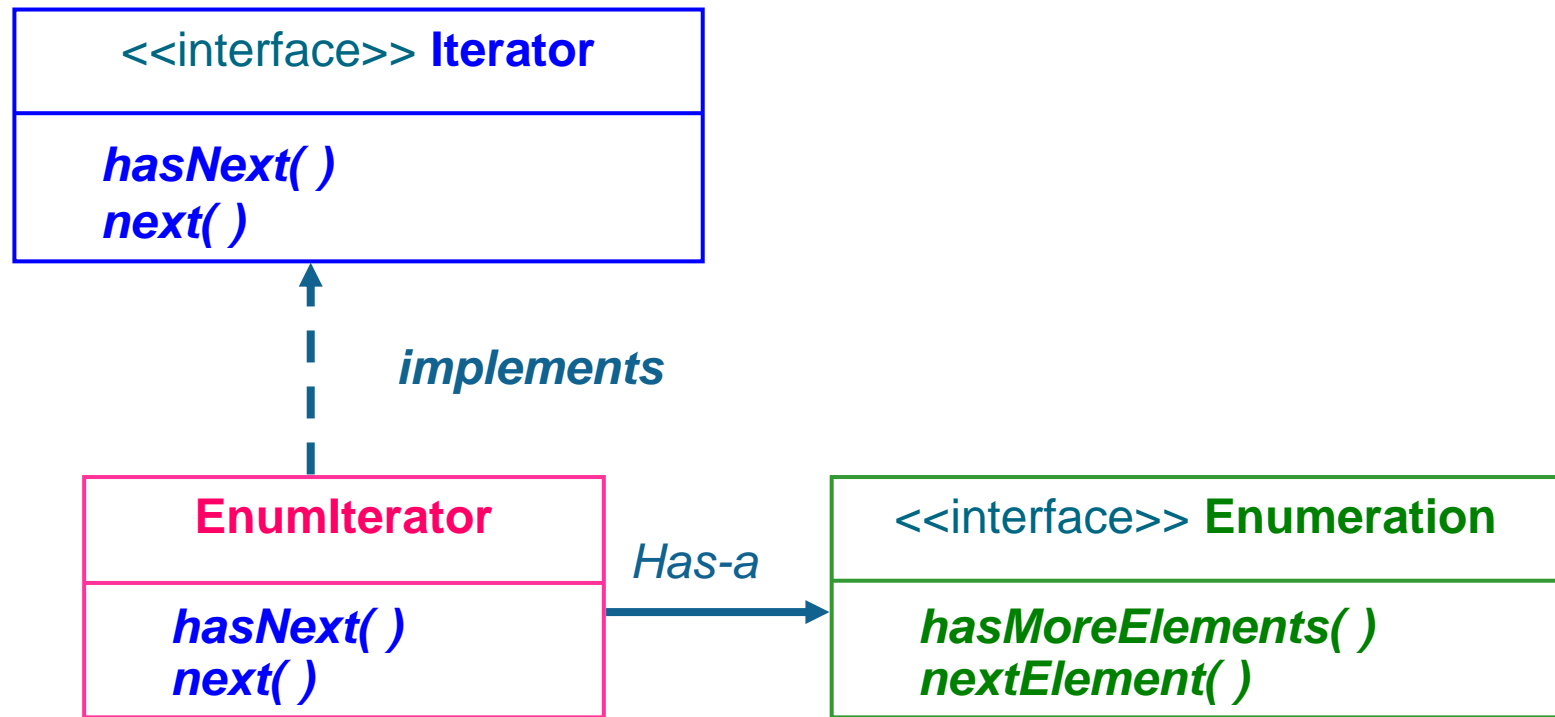
L'adaptateur traduit la requête du client en instructions compréhensibles par l'adapté (via appels de méthodes disponibles)

# Adaptator Pattern

## 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 Enumlterator*



# Adaptator Pattern

```
public class Enumlterator implements Iterator {  
    private Enumeration enumeration;  
  
    public Enumlterator (Enumeration enumeration) {  
        ...  
    }  
  
    public boolean hasNext( ) {  
        return enumeration.hasMoreElements( ) ;  
    }  
  
    public Object next( ) {  
        return enumeration.nextElement( ) ;  
    }  
}
```

# Adaptator Pattern

## Utilisation

...

Enumeration **enumeration** = ... ;

Enumliterator **adaptateur** = new Enumliterator(**enumeration**);

...

```
while (adaptateur.hasNext( ) ) {  
    ... adaptateur.next( ) ...  
}
```



# Design Patterns

1. Strategy Pattern
2. Factory Pattern
3. Prototype Pattern
4. Singleton Pattern
5. Data Access Object Pattern
6. Iterator Pattern
7. Composite Pattern
8. Decorator Pattern
9. Adaptator Pattern
10. Proxy Pattern

# Proxy Pattern

## Objectif du pattern **Proxy**

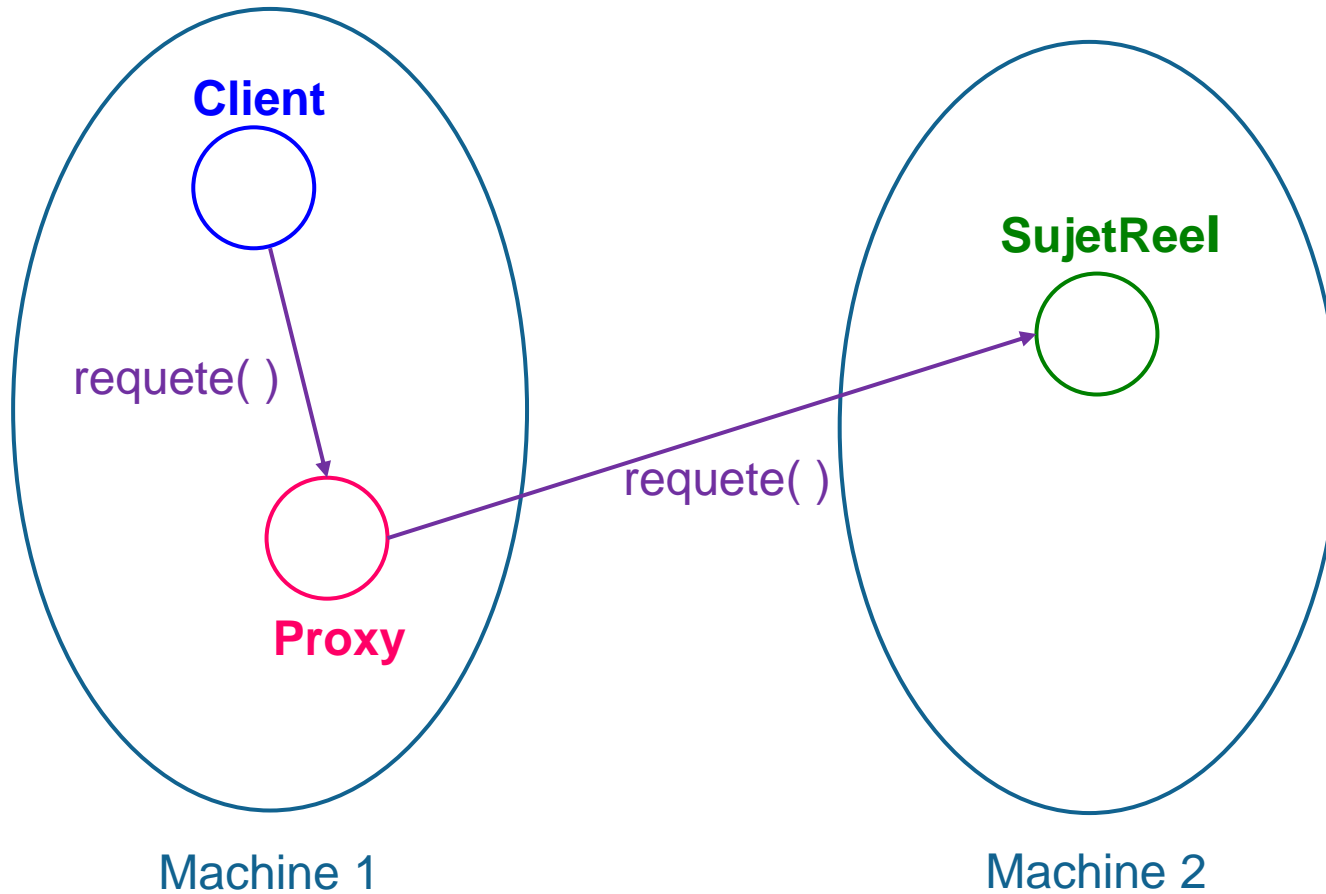
Fournir un objet remplaçant qui contrôle l'accès à un autre objet

### *Pour objets*

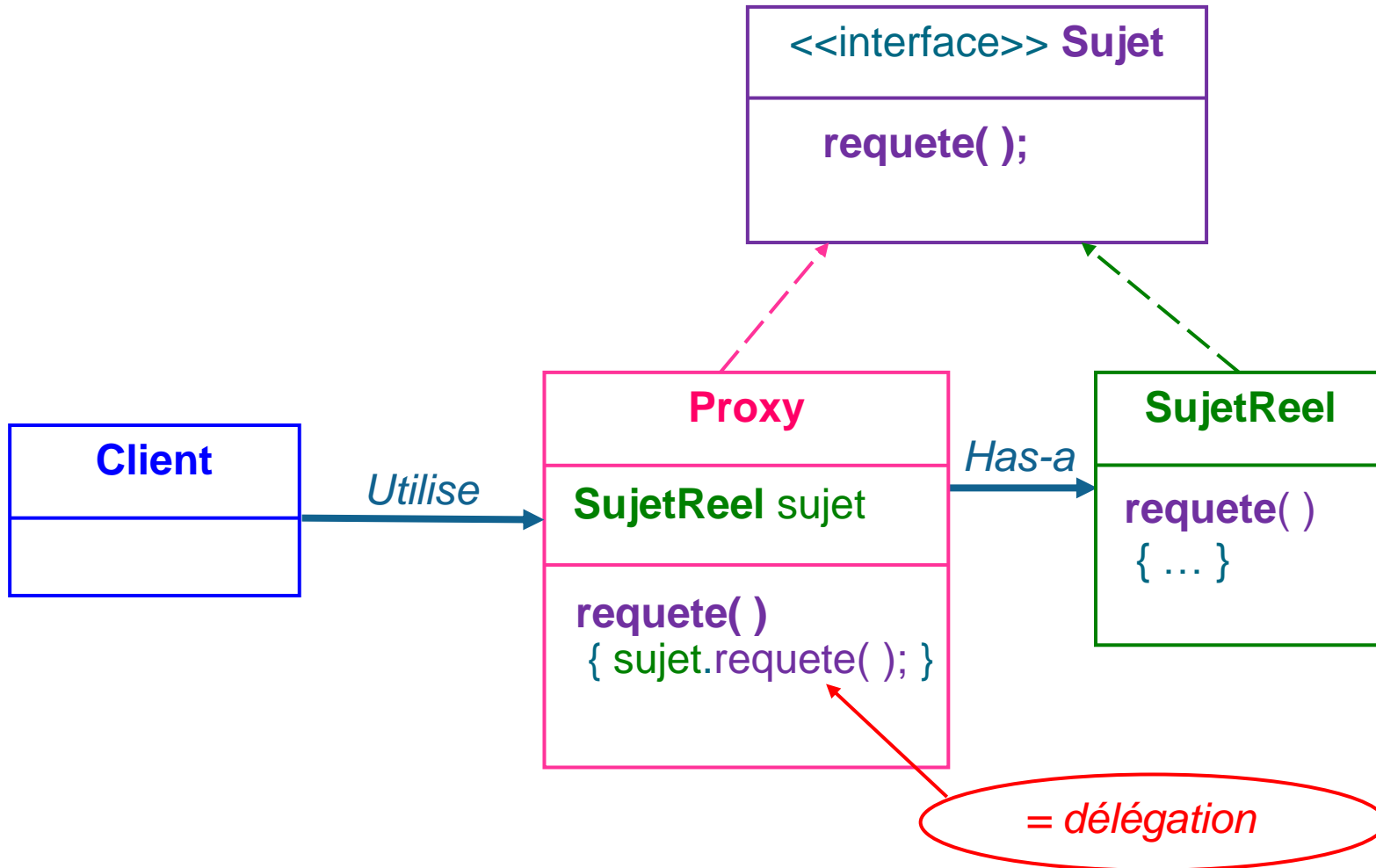
- *distants (proxy distant)*
- *couteux à créer (proxy virtuel)*
- *qui doivent être sécurisés (proxy de protection)*

# Proxy Pattern

Proxy distant



# Proxy Pattern



# Design Patterns

**5. Data Access Object Pattern**

**6. Iterator Pattern**

**7. Composite Pattern**

**8. Decorator Pattern**

**9. Adaptator Pattern**

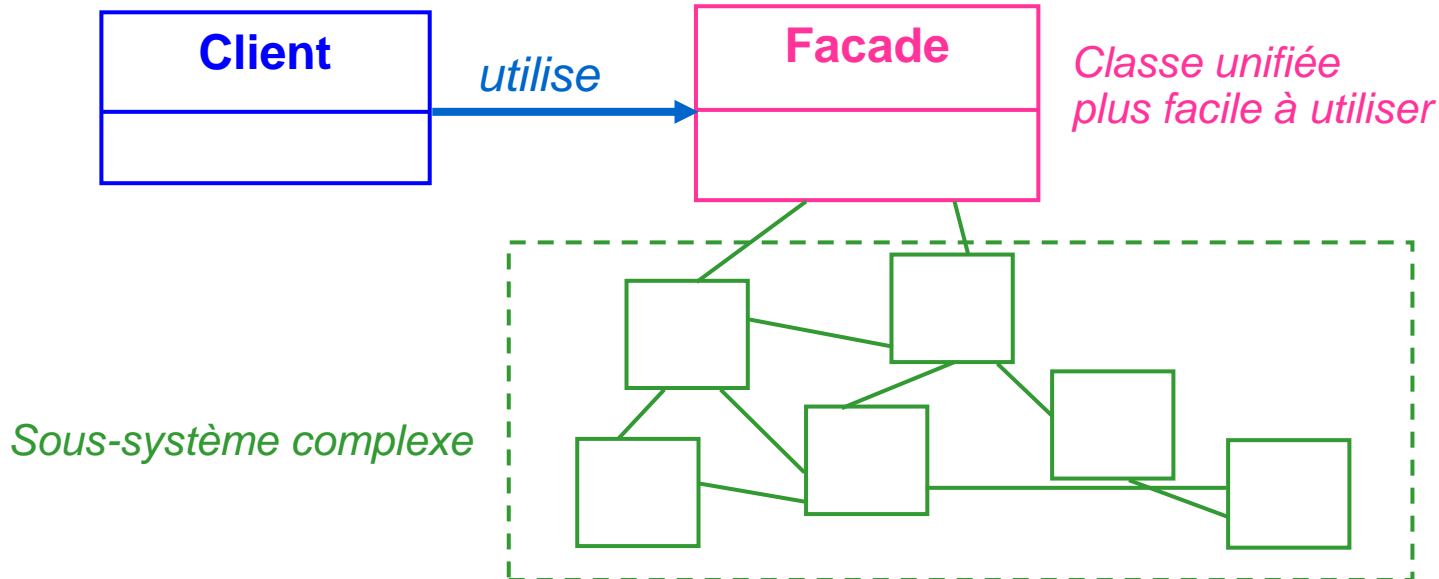
**10. Proxy Pattern**

**11. Facade Pattern**

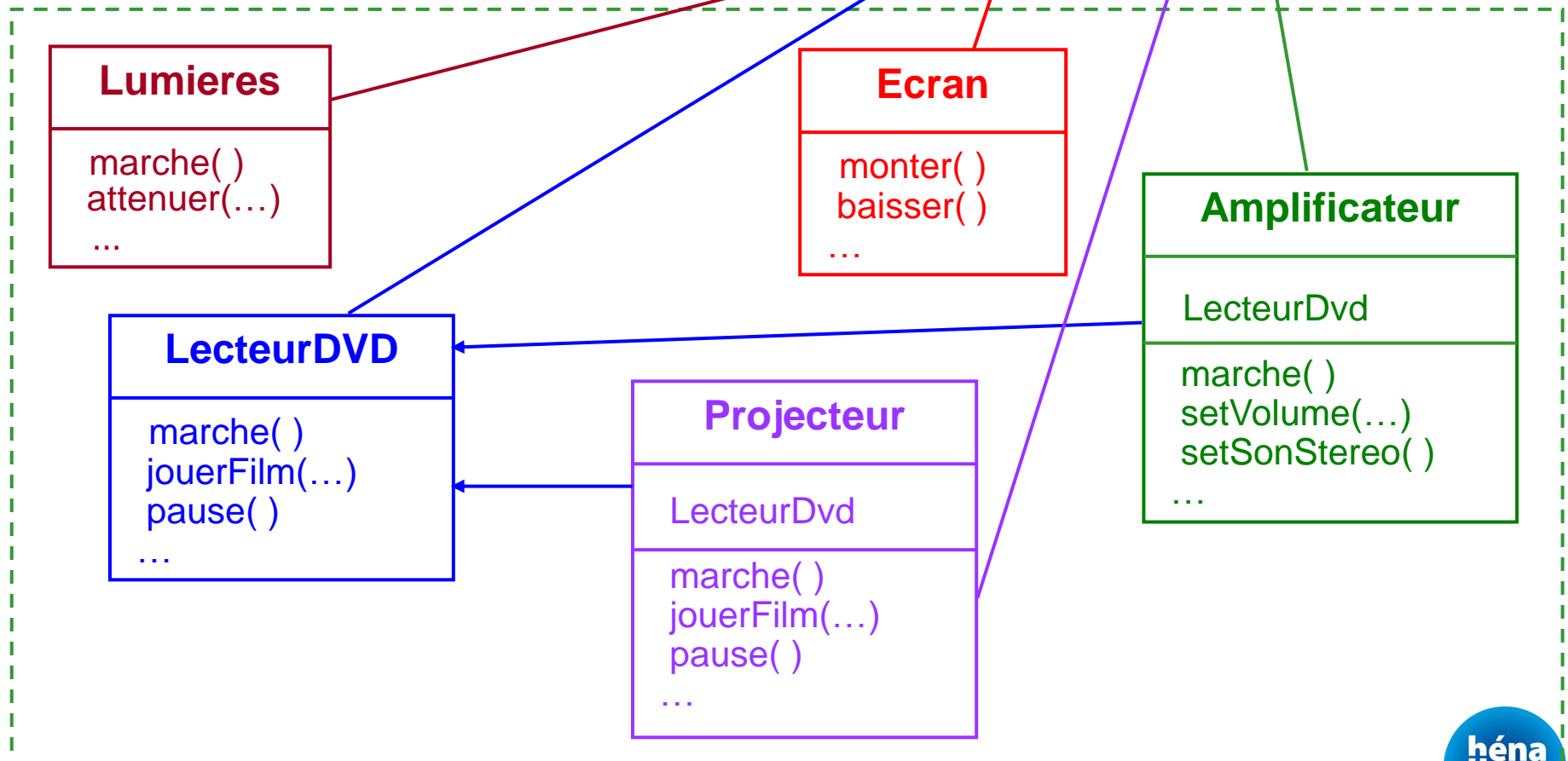
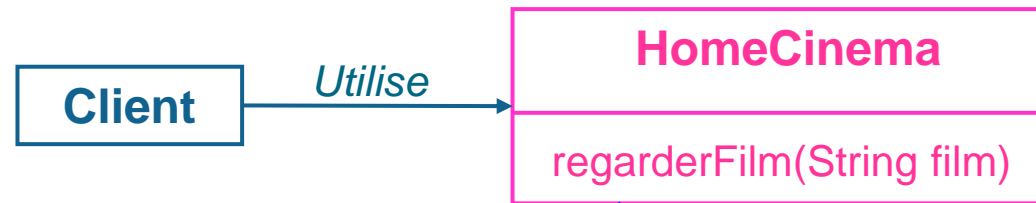
## Objectif du pattern *façade*

Faciliter l'utilisation d'un système complexe

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



Exemple



# Facade Pattern

```
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);  
    }  
}
```



# Design Patterns

**5. Data Access Object Pattern**

**6. Iterator Pattern**

**7. Composite Pattern**

**8. Decorator Pattern**

**9. Adaptator Pattern**

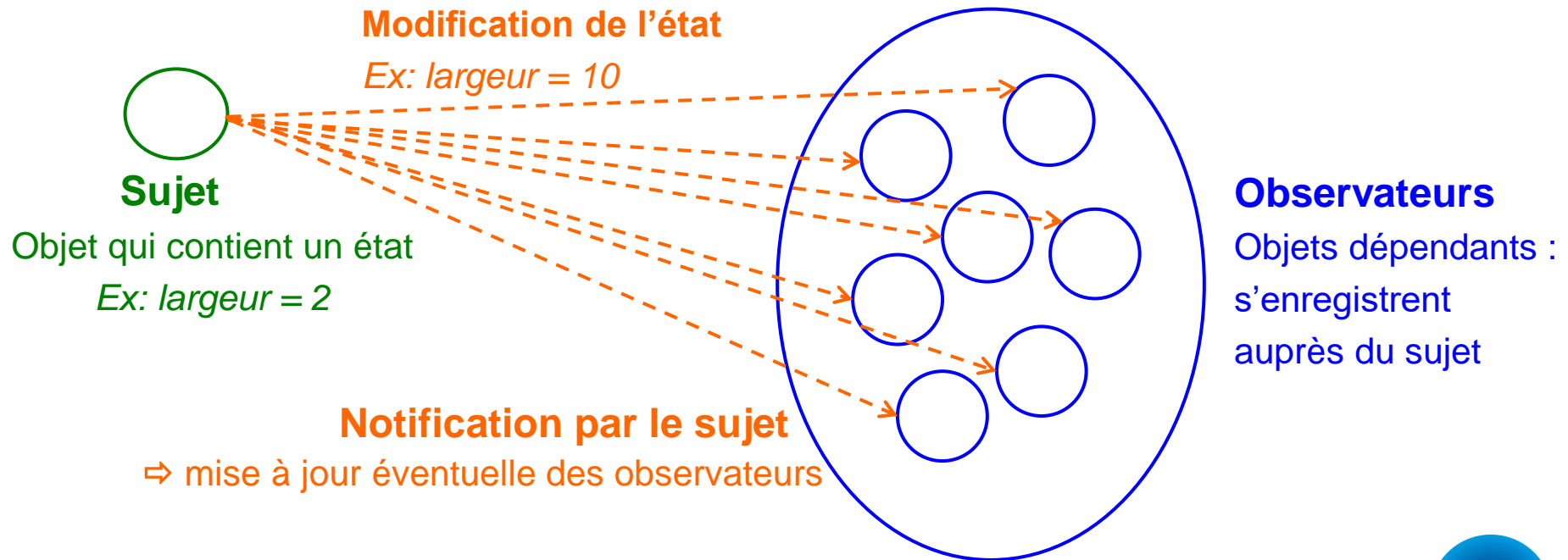
**10. Proxy Pattern**

**11. Facade Pattern**

**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)



# Observer Pattern

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

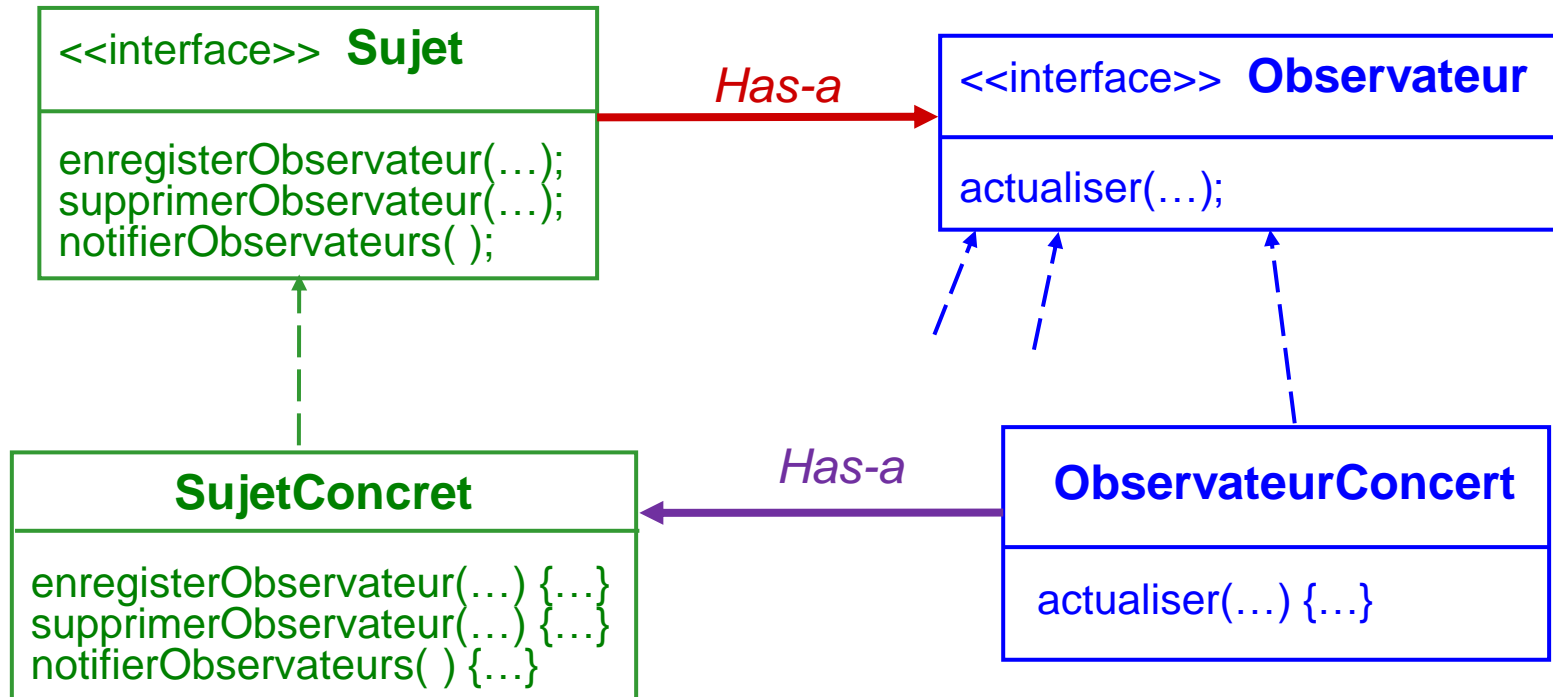


Quelle méthode?



Les observateurs doivent implémenter une interface connue du sujet

# Observer Pattern



# Observer Pattern

## Exemple 1

Gestion des événements 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

# Observer Pattern

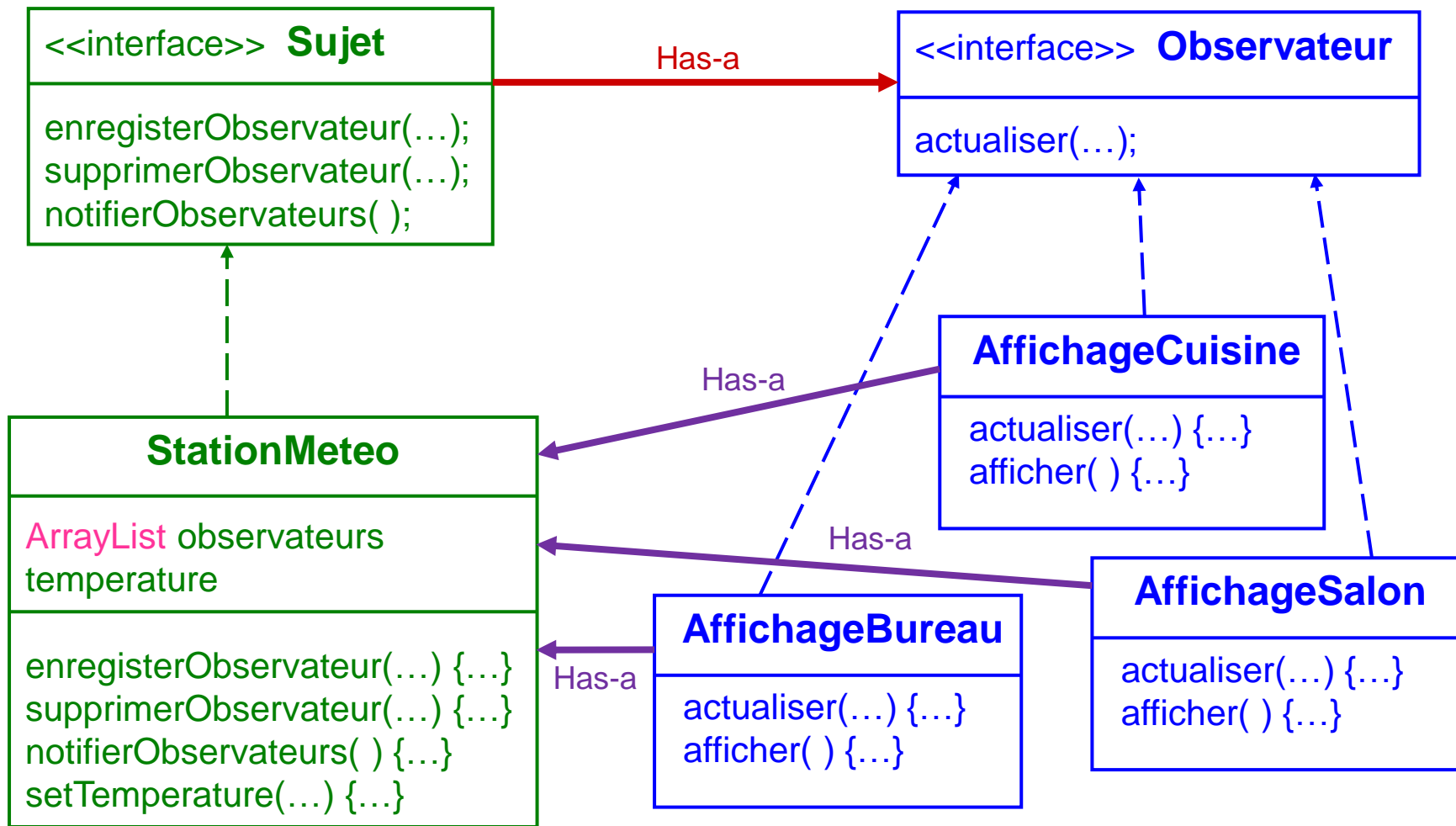
## Exemple 2

### **Sujet**

station météo qui capte la température

### **Observateurs**

appareils qui affichent la température captée par la station



# Observer Pattern

```
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 observateur : observateurs)
            observateur.actualiser(temperature);
    }

    public void setTemperature (float newTemperature) {
        temperature = newTemperature;
        notifierObservateurs( );
    }
}

```

A chaque modification de température,  
les observateurs sont notifiés

# Observer Pattern

```
public class AffichageSalon implements Observateur {  
    private Sujet donneesMeteo;  
    private float temperature;  
  
    public AffichageSalon (Sujet donneesMeteo) {  
        setDonneesMeteo(donneesMeteo);  
        donneesMeteo.enregistrerObservateur(this);  
    }  
  
    public void actualiser (float temperature) {  
        this.temperature = temperature;  
        afficher( );  
    }  
  
    public void afficher ( ) {  
        // afficher température  
    }  
    ...  
}
```

L'observateur s'enregistre auprès du sujet

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

# Observer Pattern

*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**);

# Design Patterns

- 5. Data Access Object Pattern**
- 6. Iterator Pattern**
- 7. Composite Pattern**
- 8. Decorator Pattern**
- 9. Adaptator Pattern**
- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**

# 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

# State Pattern

*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 State Pattern

## *Sans design pattern Etat*

Etats du distributeur représentés par des constantes :

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

+ mémoriser l'état courant

- ⇒ Pour chacune des méthodes, les réactions (codes des méthodes) diffèrent en fonction des états
- ⇒ Dans chaque méthode, switch à faire sur les états
- ⇒ Lourd, répétitif et difficile à maintenir !!!

# Sans State Pattern

```
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; }  
    }  
}
```



# Sans State Pattern

```
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;  
    }  
}
```

# Sans State Pattern

```
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;  
    }  
}
```

# Sans State Pattern

```
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;  
    }  
}
```

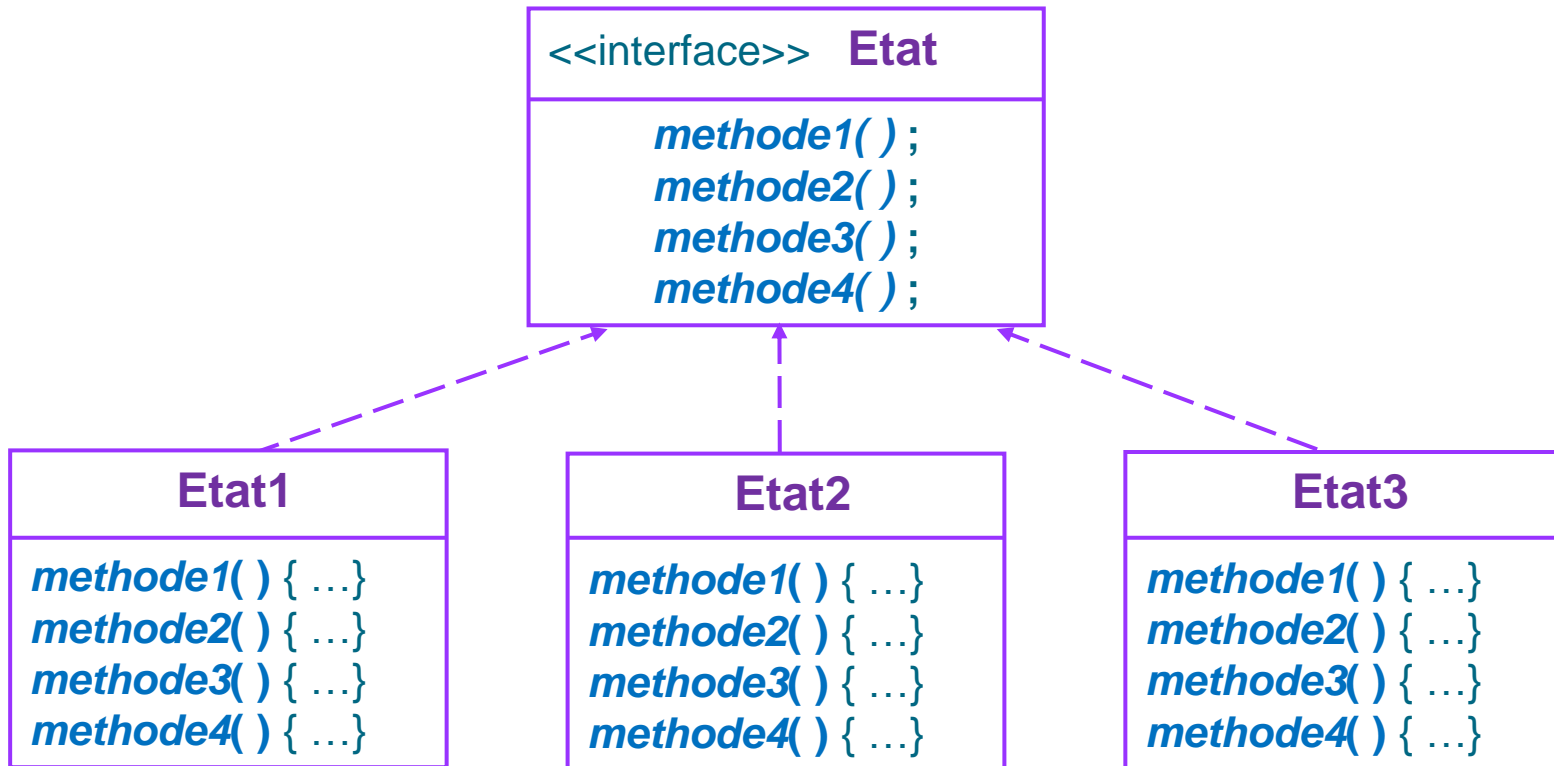
# Sans State Pattern

```
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;  
    }  
}
```

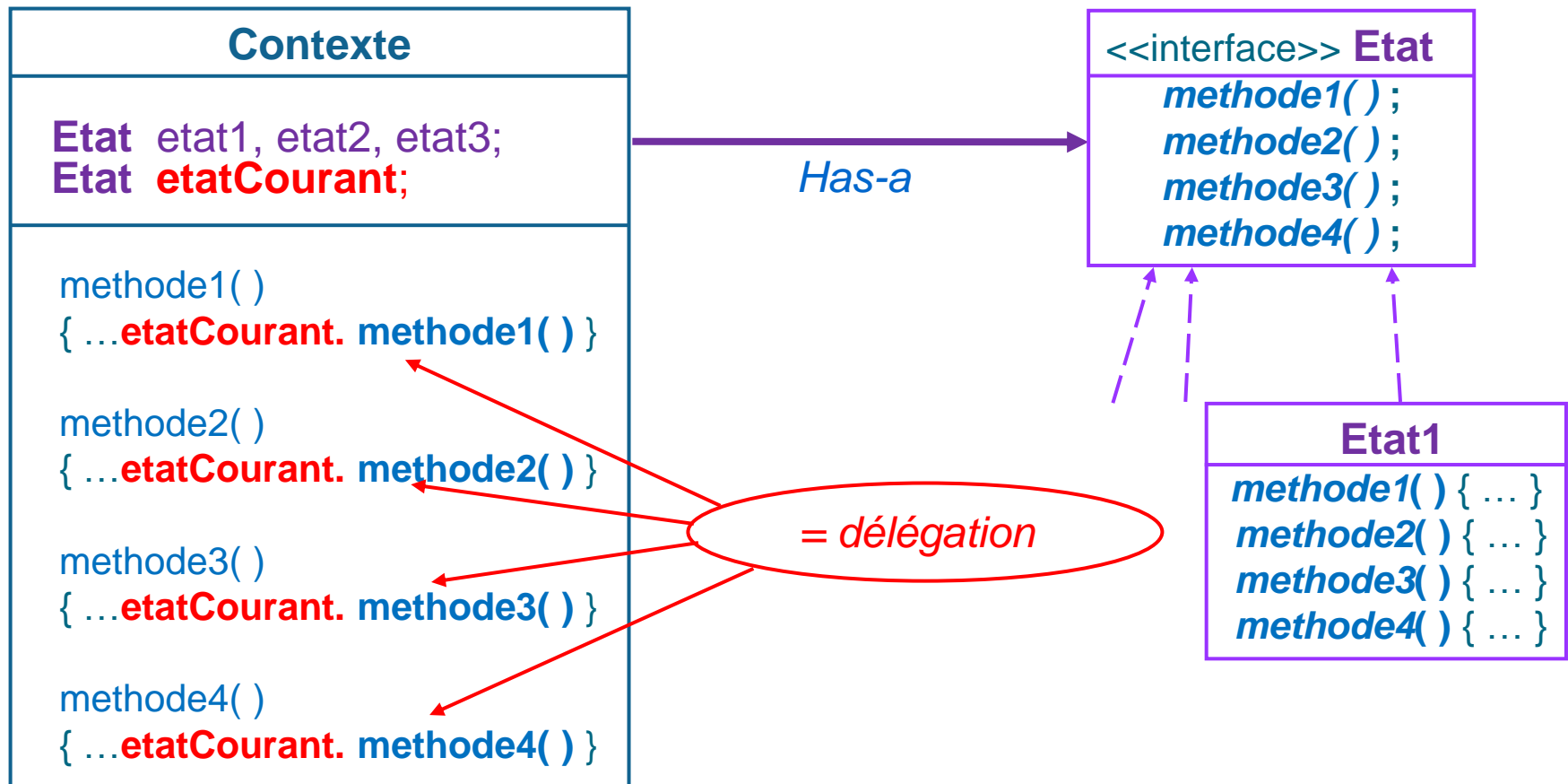
# Avec State Pattern

## Avec design pattern Etat

Créer une hiérarchie d'états

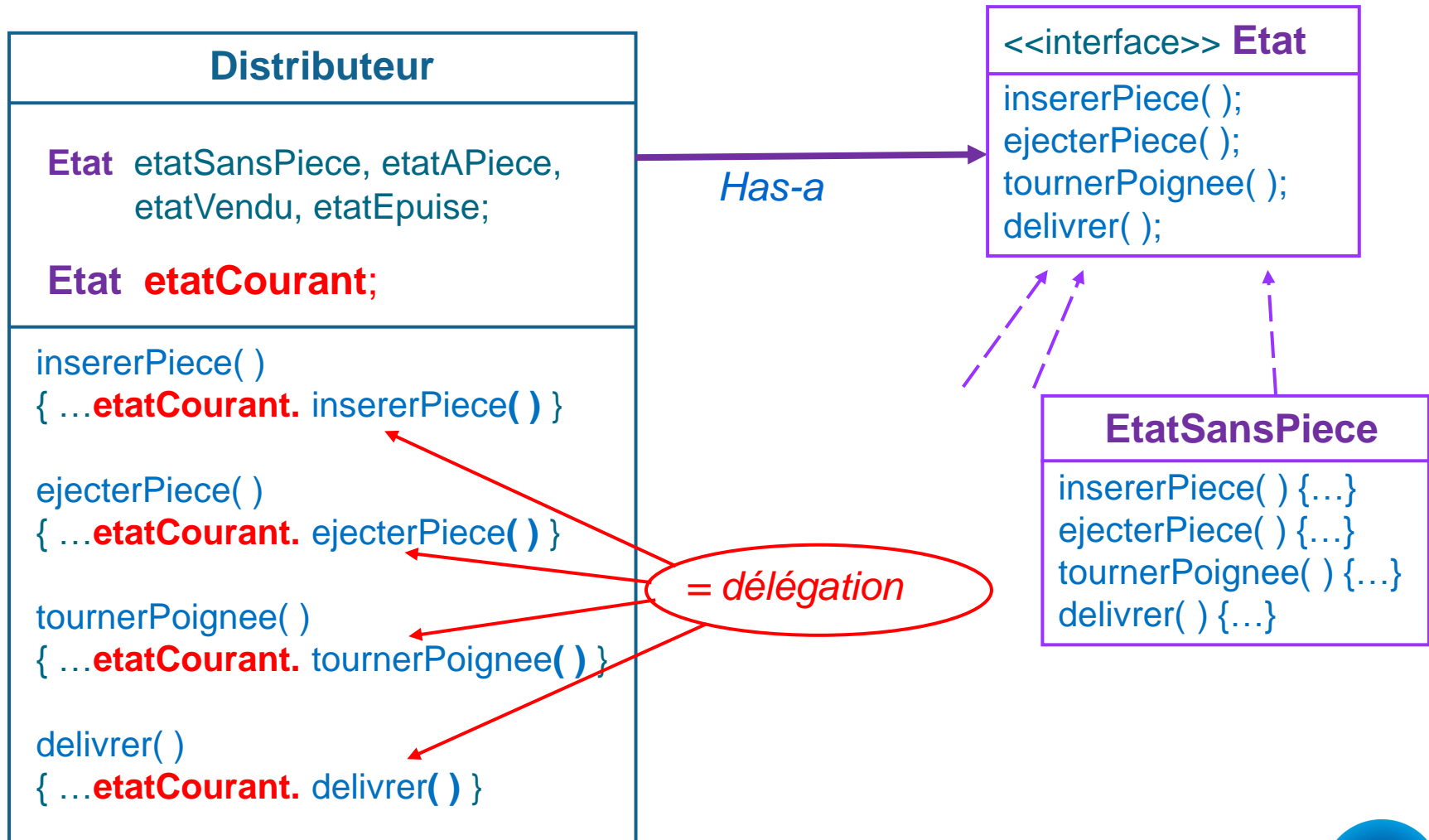


# Avec State Pattern



# Avec State Pattern

## Exemple du distributeur



```
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!"); }  
}
```



# Avec State Pattern

```
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!"); }  
}
```

# Avec State Pattern

```
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()); }  
        }  
}
```

# Avec State Pattern

```
public class EtatEpuise implements Etat {  
    private Distributeur distributeur;  
  
    public EtatEpuise (Distributeur distributeur) { this.distributeur = distributeur;}  
  
    public void insérerPiece( )  
        { 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 }

```

# Design Patterns

- 7. Composite Pattern**
- 8. Decorator Pattern**
- 9. Adaptator Pattern**
- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**

# Template Method Pattern

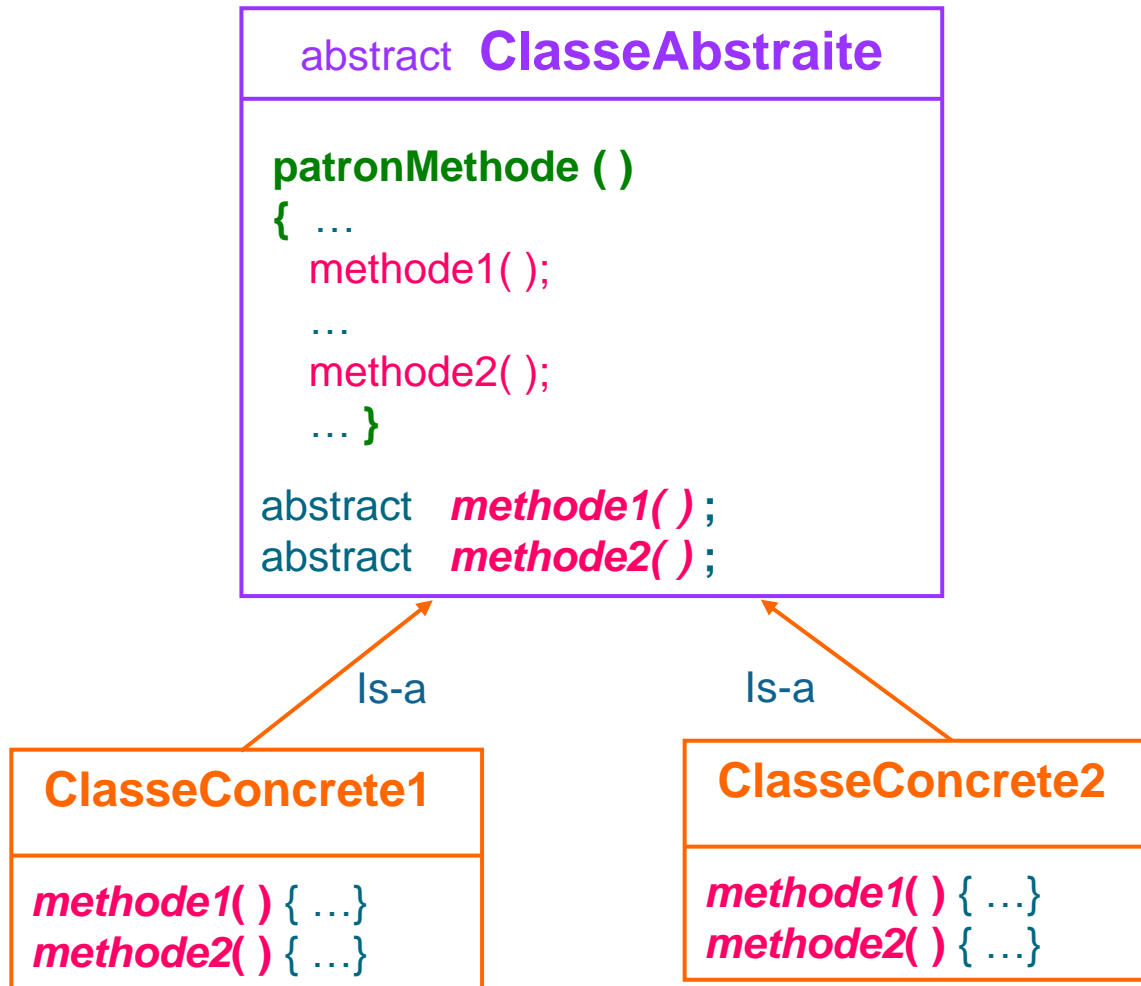
## Objectif du pattern *Patron de méthode*

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



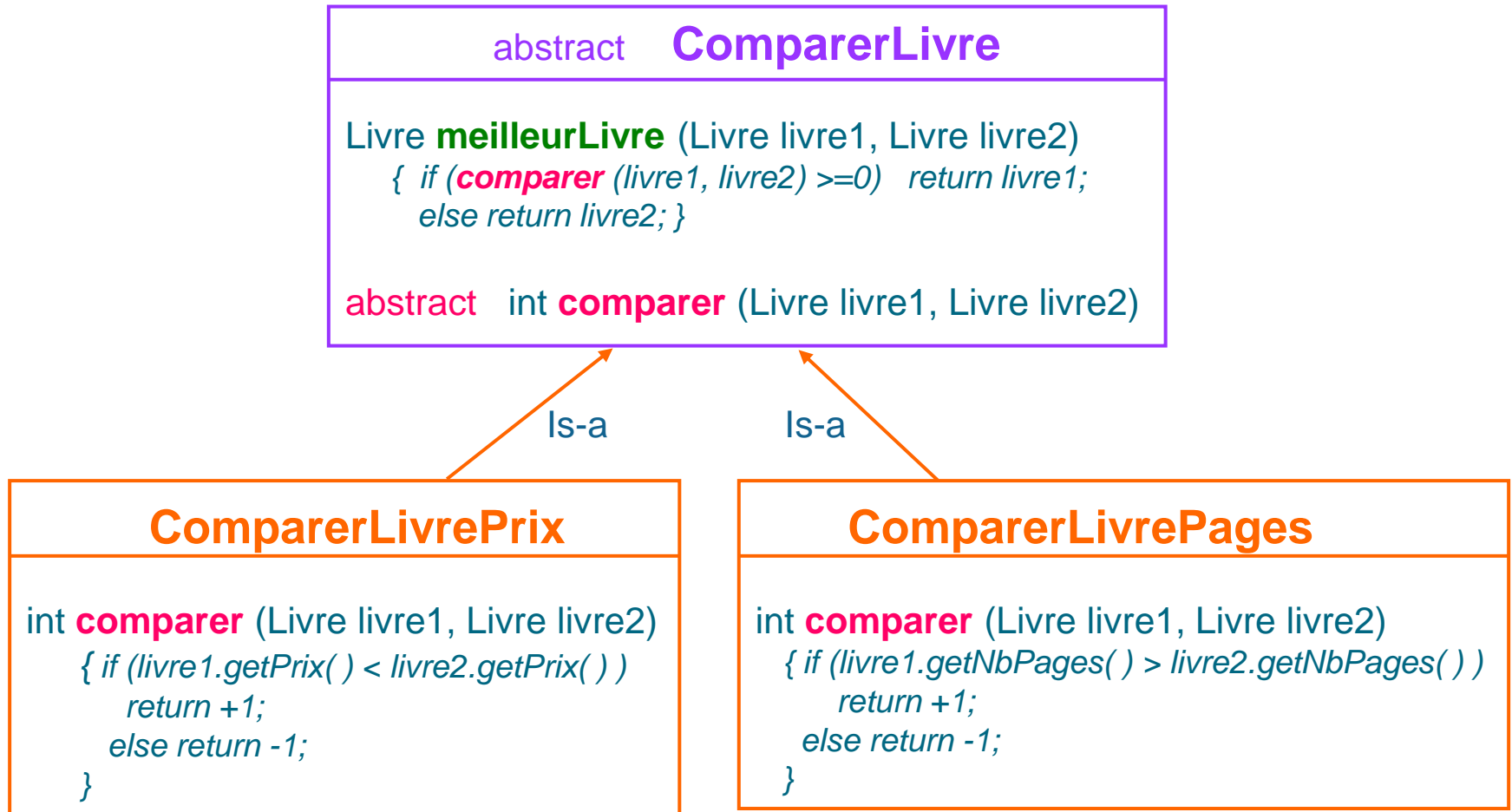
Les sous-classes redéfinissent certaines étapes d'un algorithme sans modifier la structure de celui-ci

# Template Method Pattern



# Template Method Pattern

## Exemple 1





# Template Method Pattern

## 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*

# Template Method Pattern

## Exemple 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
```

```
    ...
```

```
}
```

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

# Template Method Pattern

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

# Template Method Pattern

## *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; }  
}
```

# Template Method Pattern

Utilisation :

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

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

# Design Patterns

- 7. Composite Pattern**
- 8. Decorator Pattern**
- 9. Adaptator Pattern**
- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**
- 15. Flyweight Pattern**

# 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

# Flyweight Pattern

- Si un rectangle de cette couleur existe dans la HashMap
  - ⇒ on le retourne
- Sinon
  - ⇒ on crée un rectangle de cette couleur
  - ⇒ on le place dans la HashMap
  - ⇒ on retourne ce rectangle

## FabriqueRectangle

**HashMap** <String, **Forme**> rectangles

**Forme** getRectangle ( couleur )  
{ ... }

Has-a

<<interface>> **Forme**

*dessiner( );*

## Rectangle

*couleur*

*dessiner ( ) { ... }*



# Flyweight Pattern


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

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

# Flyweight Pattern

```
public class FabriqueRectangle {  
    private static final HashMap<String, Forme> rectangles = new HashMap<>( );  
    ...  
    public static Forme getRectangle (String couleur) {  
        Forme 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;  
    }  
}
```

La clé est la couleur



# Flyweight Pattern

## 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( );
```

```
}
```

# Flyweight Pattern

## 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

# Design Patterns

- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**
- 15. Flyweight Pattern**
- 16. PlayerRole Pattern**

# PlayerRole Pattern

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

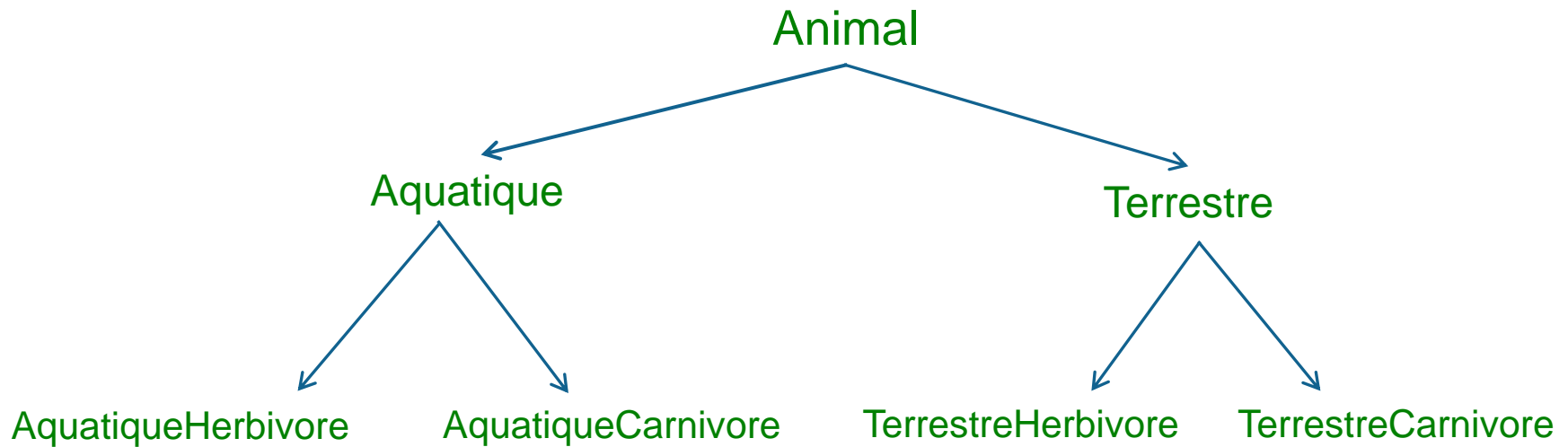
Permettre à un objet de jouer plusieurs rôles  
+ de changer ses rôles dynamiquement

## 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

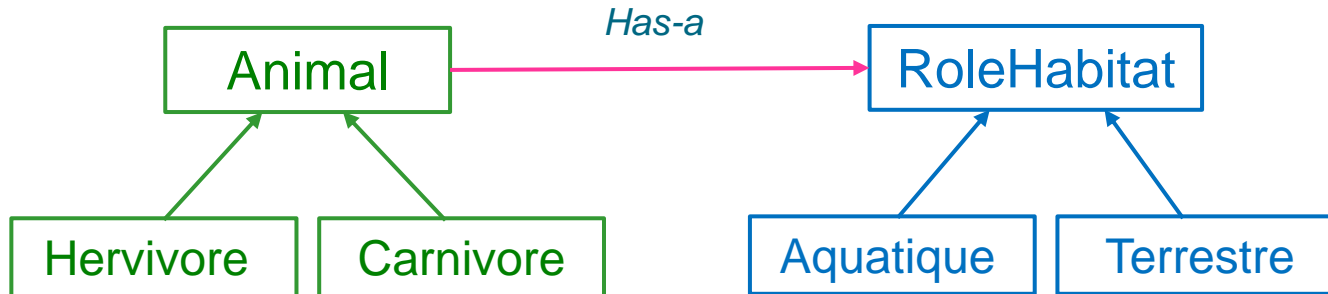
# Sans PlayerRole Pattern

Sans Design Pattern :  
Contre-exemple



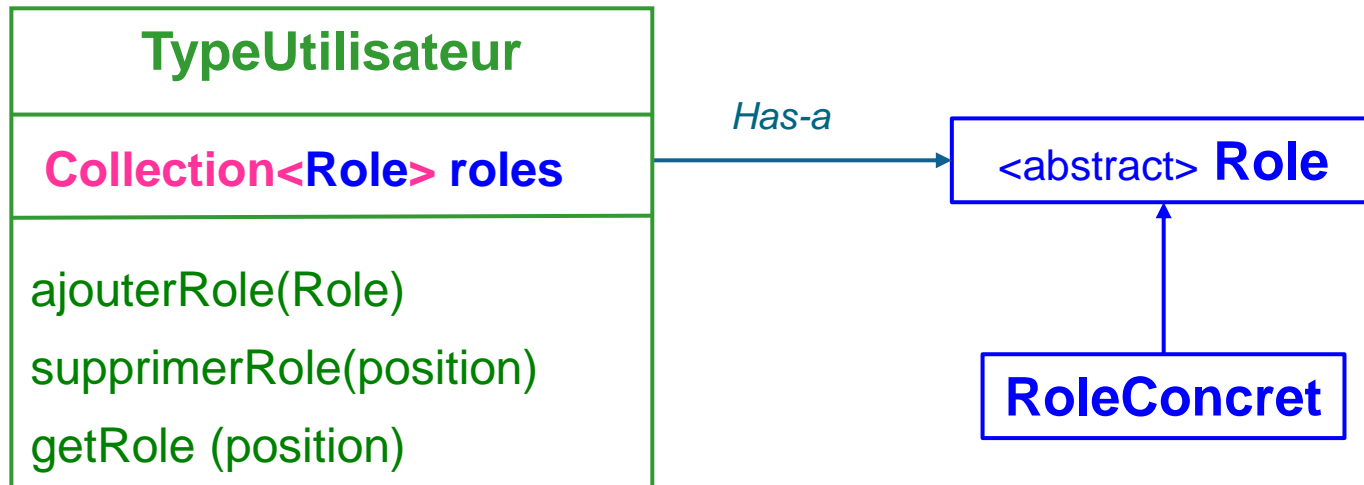
# Avec PlayerRole Pattern

Avec Design Pattern

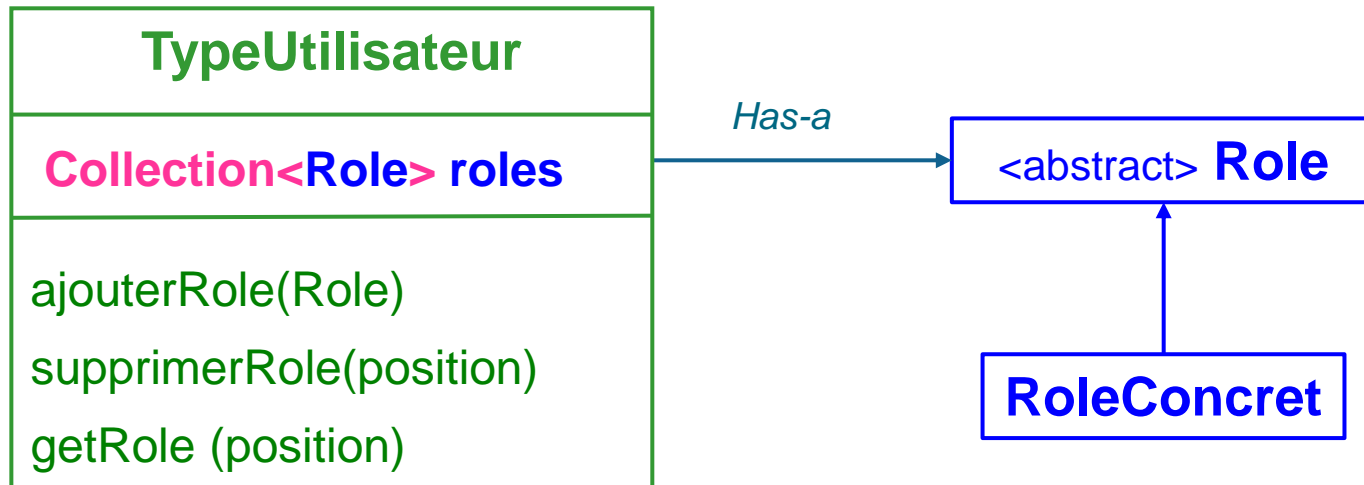




# PlayerRole Pattern



# PlayerRole Pattern



# Design Patterns

- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**
- 15. Flyweight Pattern**
- 16. PlayerRole Pattern**
- 17. Visitor Pattern**

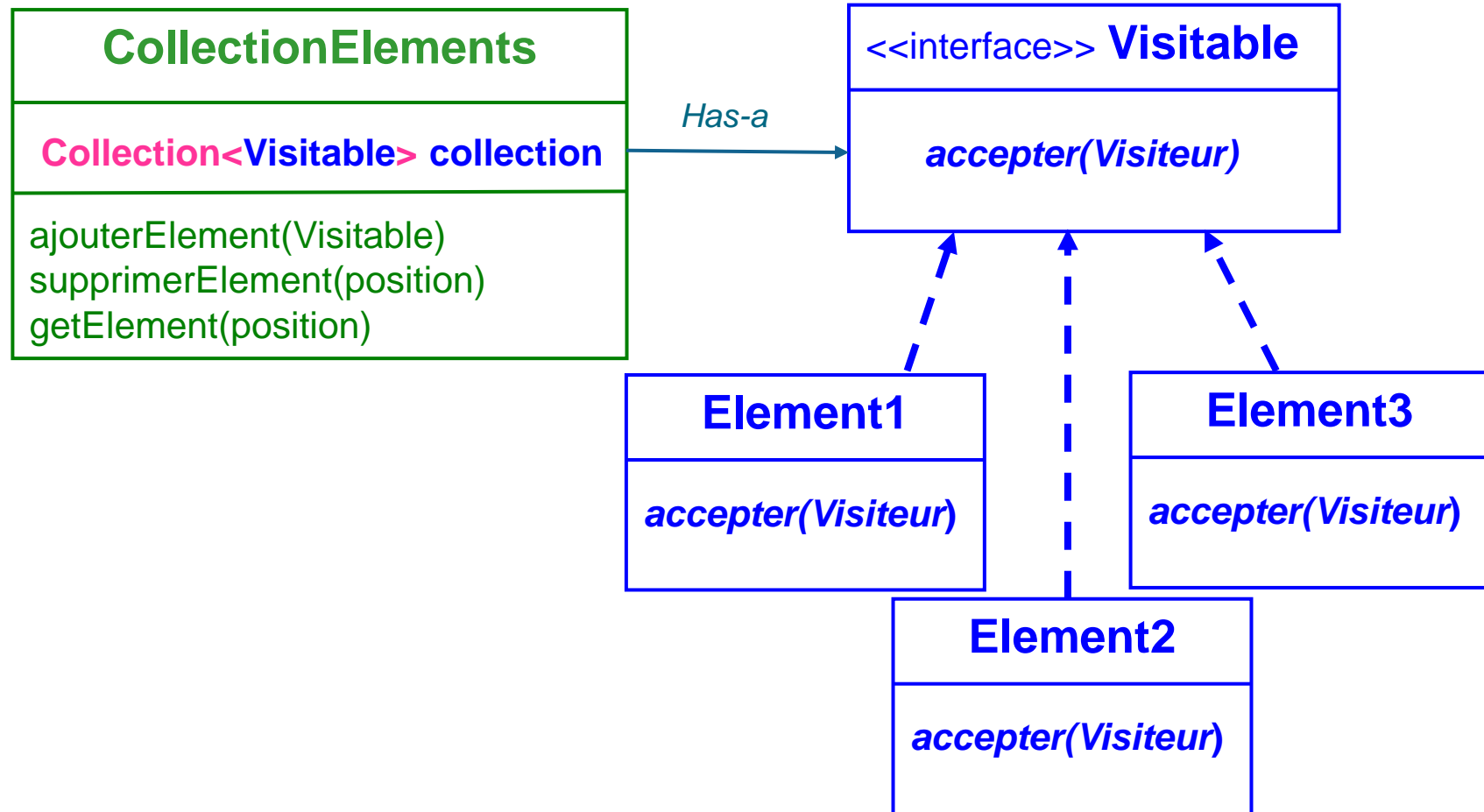
# Visitor Pattern

## Objectif du pattern *Visiteur*

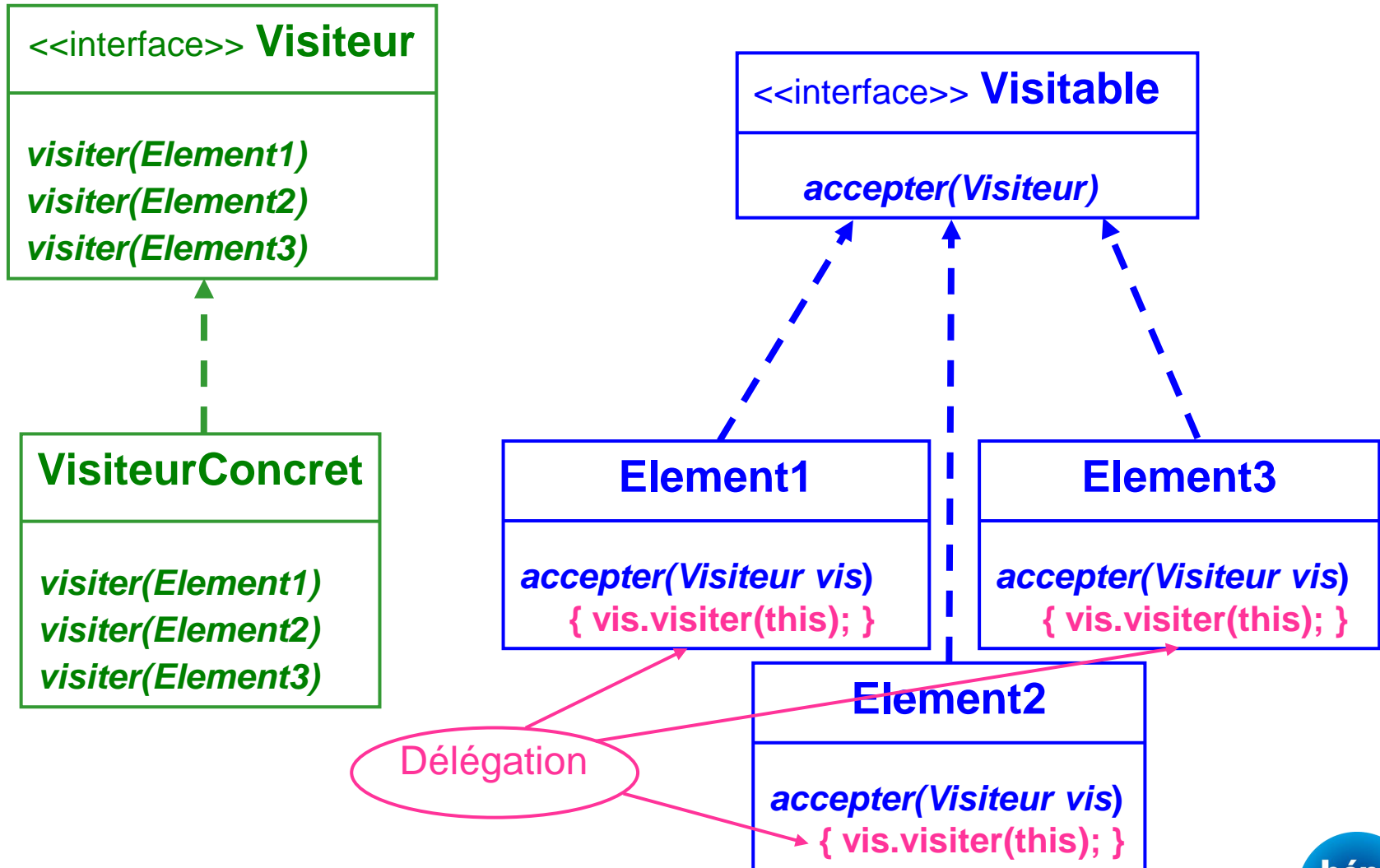
Permettre d'appliquer une ou plusieurs opérations (algorithmes) sur un **ensemble d'éléments** tout en découplant les opérations de la structure des objets.

- ⇒ 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

# Visitor Pattern

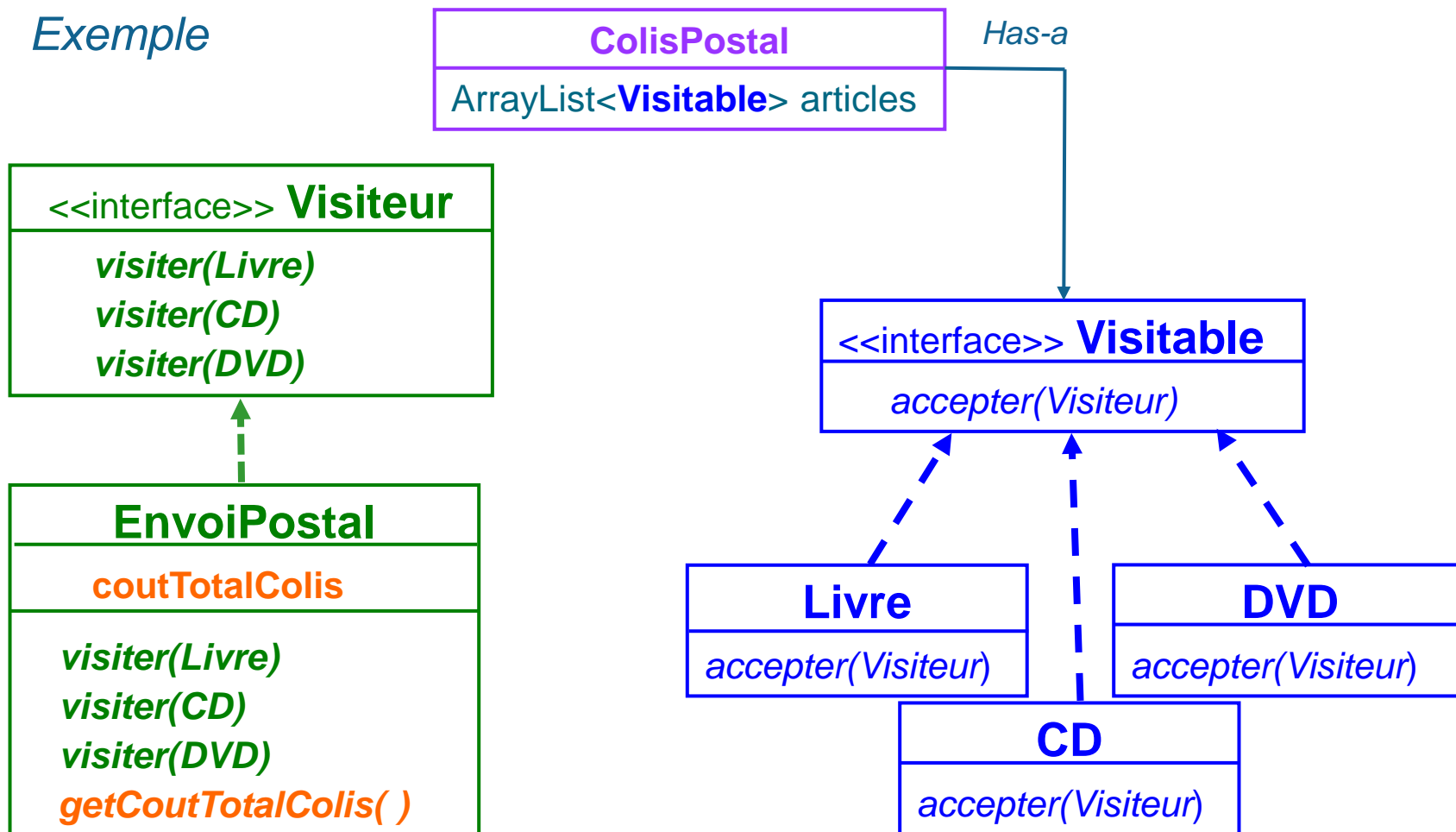


# Visitor Pattern



# Visitor Pattern

## Exemple



# Visitor Pattern

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

```
public class Livre implements Visitable {  
    private double prix, poids;  
  
    public Livre(...) { ... }  
  
    @Override  
    public void accepter (Visiteur visiteur) {  
        visiteur.visiter(this);  
    }  
    ...  
}
```



# Visitor Pattern

```
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) // gratuit pour les livres > 10 euros  
        { if ( livre.getPrix( ) < 10.0) { 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; }  
}
```

# Visitor Pattern

```
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*

# Design Patterns

- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**
- 15. Flyweight Pattern**
- 16. PlayerRole Pattern**
- 17. Visitor Pattern**
- 18. Memento Pattern**

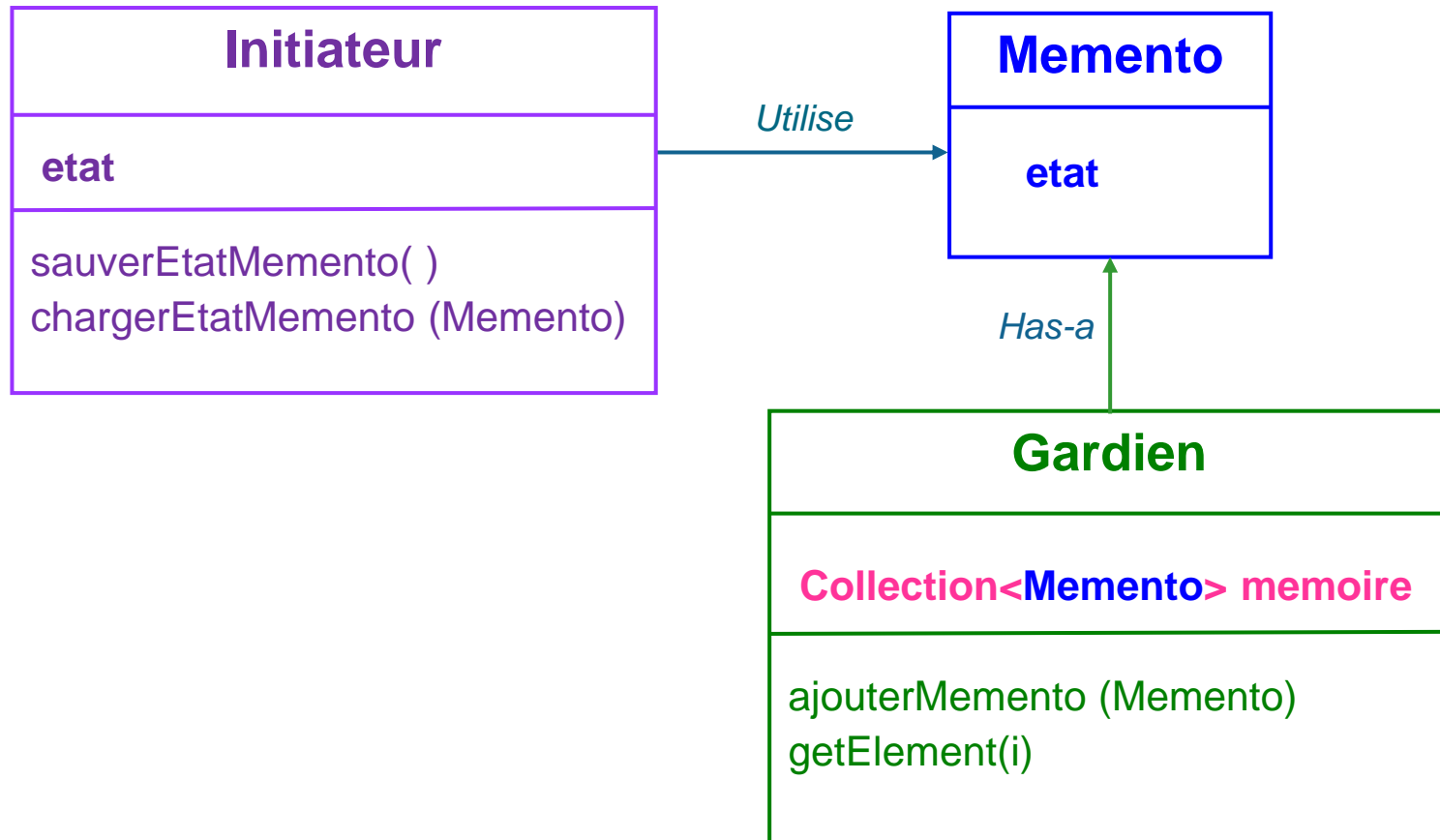
# Memento Pattern

## Objectif du pattern **Memento**

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

- ⇒ 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

# Memento Pattern



# Memento Pattern

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

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

```
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( ));
```

# Design Patterns

- 10. Proxy Pattern**
- 11. Facade Pattern**
- 12. Observer Pattern**
- 13. State Pattern**
- 14. Template Method Pattern**
- 15. Flyweight Pattern**
- 16. PlayerRole Pattern**
- 17. Visitor Pattern**
- 18. Memento Pattern**
- 19. Mediator Pattern**



# 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**

# Mediator Pattern



*Exemple*



# Mediator Pattern

```
public class Utilisateur {  
    private String nom;  
    public Utilisateur( ... ) { ... }  
    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);  
    }  
}
```

# Other Design Patterns

*Mais encore ...*

- Builder Pattern
- Bridge Pattern
- Filter Pattern
- Chain of Responsibility Pattern
- Command Pattern
- Interpreter Pattern
- Null Object Pattern
- Template Pattern
- MVC Pattern
- Business Delegate Pattern
- Composite Entity Pattern
- Front Controller Pattern
- Intercepting Filter Pattern
- Service Locator Pattern
- Transfer Object Pattern
- ...

# Sources Design Pattern

## *Bibliographie*

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

## *Webographie*

<https://refactoring.guru/design-patterns>  
[http://www.tutorialspoint.com/design\\_pattern/](http://www.tutorialspoint.com/design_pattern/)