Les modèles de conception (Design Patterns)

Cours 9 / GL / LP DA2I

Cédric Lhoussaine

2019-2020

Principes

Définition d'un Design Pattern

- référence : E. Gamma, R. Helm, R. Johnson, J. Vlissides : Design Patterns (éd. Vuibert)

$\rightarrow$ tentative pour répertorier et catégoriser des situations rencontrées fréquemment en conception objet.

- ensemble de classes spécialisées en relation
- augmenter la réutilisabilité

- accélérer la conception $\rightarrow$ solutions génériques

Boîtes à outils et frameworks

deux autres façons d'améliorer la réutilisabilité du code:

- les boîtes à outils (toolkits): ensemble de classes et de méthodes utilitaires (ex : `java.util`)
- les frameworks: ensemble de classes qui coopèrent, qu'on peut facilement dériver pour une application donnée (ex : AWT) $\rightarrow$ assez gros

Boîtes à outils et frameworks

deux autres façons d'améliorer la réutilisabilité du code:

- les boîtes à outils (toolkits): ensemble de classes et de méthodes utilitaires (ex : `java.util`)
- les frameworks: ensemble de classes qui coopèrent, qu'on peut facilement dériver pour une application donnée (ex : AWT) $\rightarrow$ assez gros

  les modèles de conception sont plus abstraits, plus petits et moins spécialisés que les frameworks

Exemple ancien : MVC

pattern Model-View-Controller de Smalltalk destiné à la représentation générique de données

- Model: la classe contenant la source des données ou l'application
- View: la classe donnant une représentation ("visuelle") du modèle. ex: diagramme en barres, tableau de valeurs, interface graphique...
- Controller: gestionnaire d'événements permettant de lier la vue et le modèle.
  - quand l'utilisateur agit sur la vue (saisie...), le modèle est impacté;
  - quand le modèle change (calculs...), la vue est mise à jour.

Typologie des modèles de conception

on "classe" les *Design Patterns* en fonction de leur rôle:

- créateur: abstraction du processus d'instanciation
  Factory method, *Singleton*

Typologie des modèles de conception

on "classe" les *Design Patterns* en fonction de leur rôle:

- créateur: abstraction du processus d'instanciation
  Factory method, *Singleton*
- structurel: composition de classes et objets pour concevoir une structure complexe
  Composite, *Decorator*

Typologie des modèles de conception

on "classe" les *Design Patterns* en fonction de leur rôle:

- créateur: abstraction du processus d'instanciation
  Factory method, *Singleton*
- structurel: composition de classes et objets pour concevoir une structure complexe
  Composite, *Decorator*
- comportemental: aspects algorithmiques et de répartition des responsabilités entre objets
  Iterator, *Template Method*

Patrons de conception comportementaux

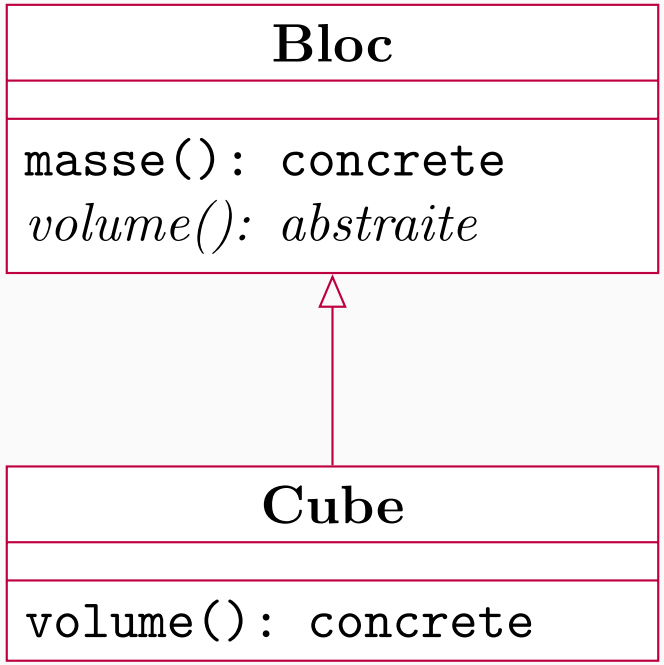
Template method (Principe)

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template method (Exemple 1)

```
public abstract class Bloc {
    public double masse() {
        return masse_volumique * volume() ;
    }

    public abstract double volume();
}

public class Cube extends Bloc {
    public double volume() {
        return cote * cote * cote ;
    }
}
```

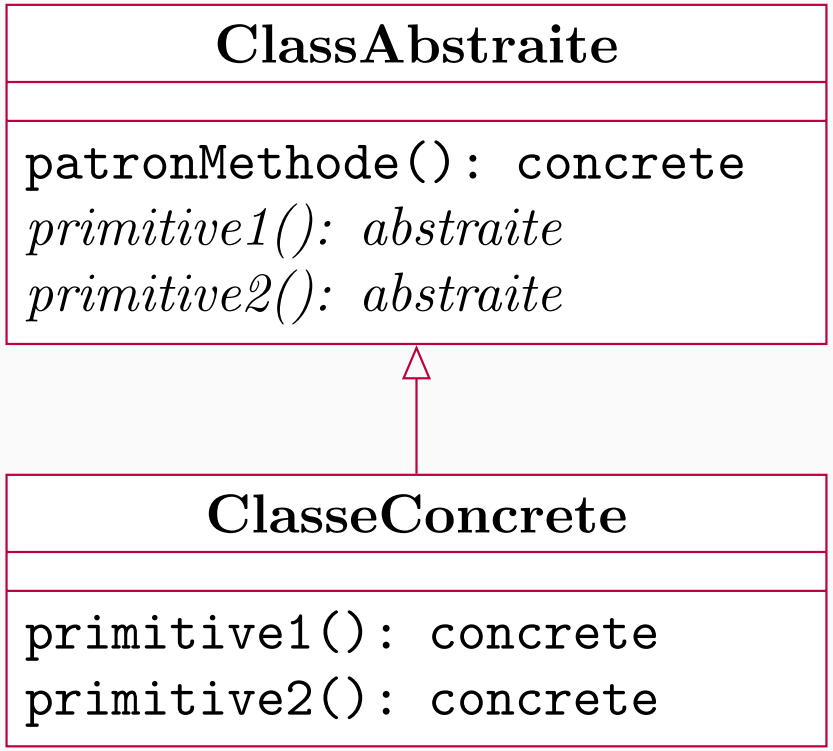
Template method (Exemple 1)



Template method (Exemple 2)

TP7 $\rightarrow$ Transport

Template method (Diagramme)

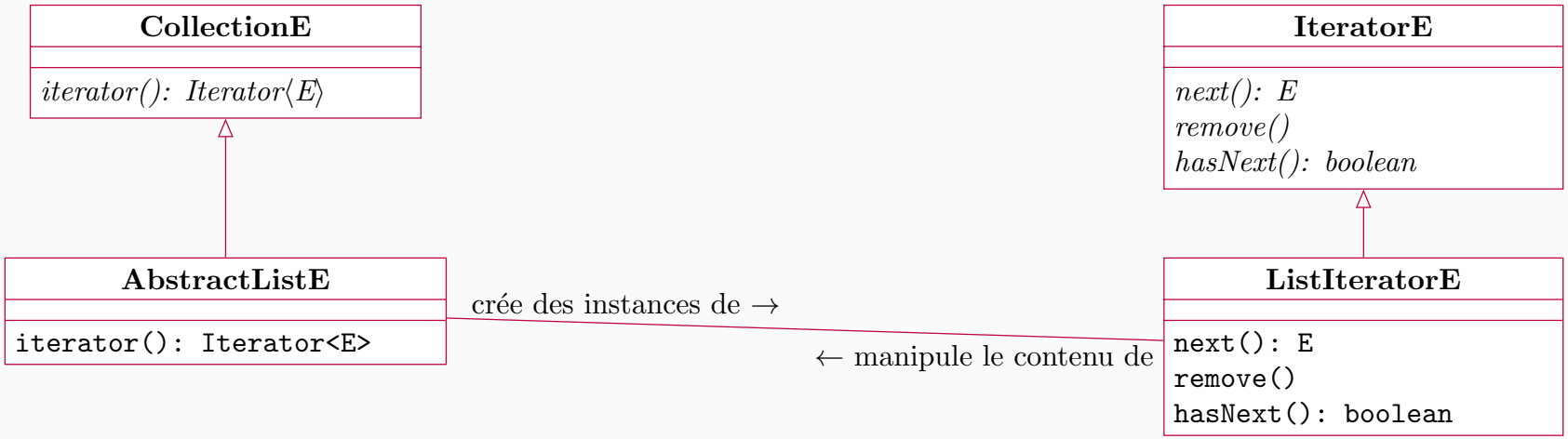


- le code de `patronMethode` fait appel aux primitives abstraites;
- celles-ci ne sont définies que dans les sous-classes.

Iterator (Principe)

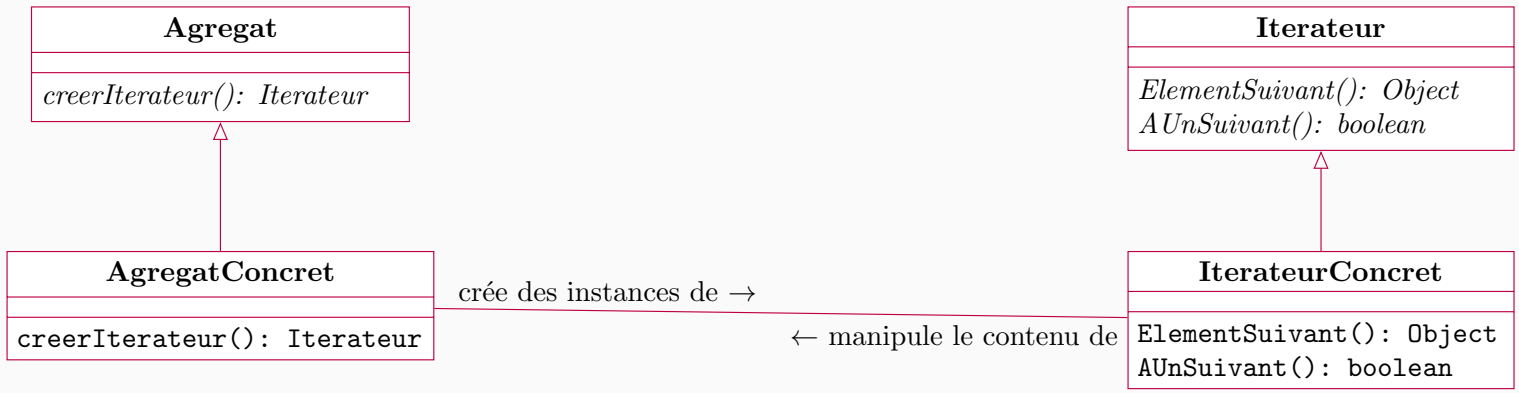
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator (dans `java.util`)



Les itérateurs des collections en Java sont des classes internes

Iterator (Diagramme)



Iterator (Exemple. . .)

Iterator (Utilisation)

Indications d'utilisation :

- pour accéder séquentiellement au contenu d'un agrégat sans en révéler la représentation interne
- pour offrir une interface uniforme pour les parcours au travers de diverses structures d'agrégats (polymorphisme)

Patrons de conception créateurs

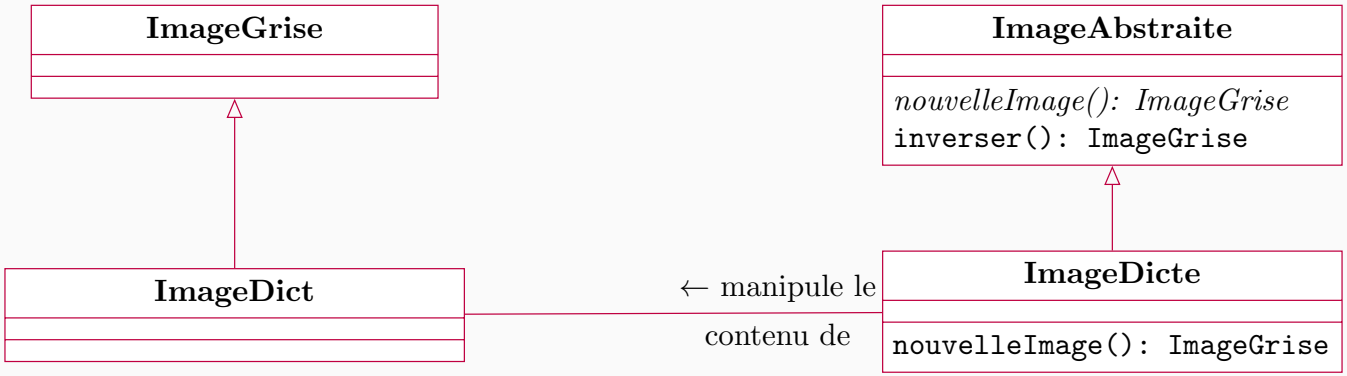
Factory method (Principe)

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

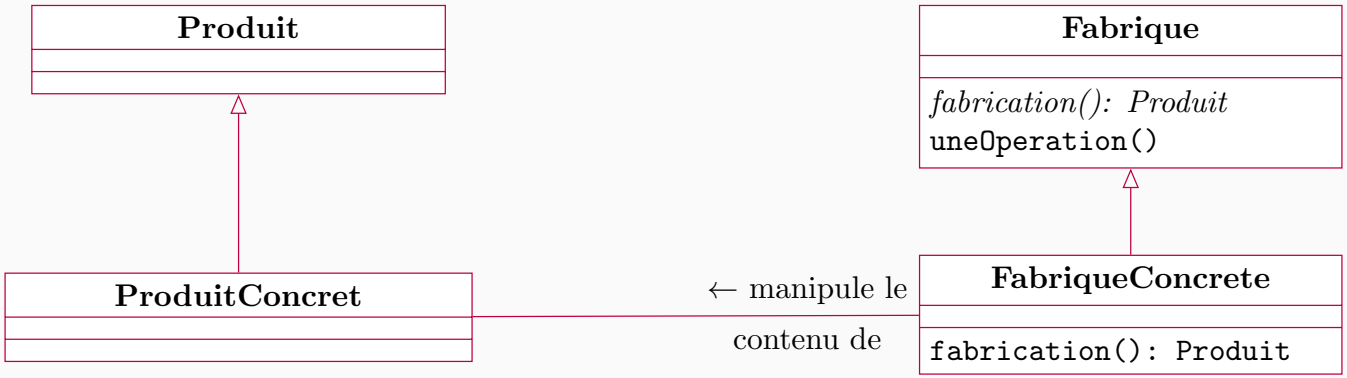
Factory method (Exemple)

Fabrique de Blocs. . .

Factory method (Exemple)



Factory method (Diagramme)



Factory method (Utilisation)

- une classe doit créer des instances d'objets dont elle connaît le type mais pas la classe;
- une classe attend de ses sous-classes qu'elles spécifient explicitement les objets qu'elles créent.

Singleton (Principe)

- *Ensure a class only has one instance, and provide a global point of access to it.*

Singleton (Principe)

- *Ensure a class only has one instance, and provide a global point of access to it.*

- classe qui n'a qu'une seule instance
- accès à cette instance par une méthode dédiée

Singleton (Utilisation)

- accès contrôlé à une instance unique
- alternative "propre" aux variables globales

Essentiel: veiller à empêcher ou à intercepter les tentatives de création d'instances non autorisées !

Singleton mal réalisé

```
public class BadSingleton {
    private String nom = "TOTO !";
    protected static BadSingleton instanceUnique = creerInstance();

    private static Singleton creerInstance() {
        BadSingleton s = new BadSingleton();
        s.nom = "UNIQUE";
        return s;
    }

    public static BadSingleton getInstance() { return instanceUnique; }

    public String toString() { return "Je suis l'instance " + nom; }

    public static void main(String [] a) {
        System.out.println(BadSingleton.getInstance());
        System.out.println(BadSingleton.getInstance());
        System.out.println(new BadSingleton());
    }
}
```

Résultat ??

Singleton mal réalisé

```
Je suis l'instance UNIQUE
Je suis l'instance UNIQUE
Je suis l'instance TOTO !
```

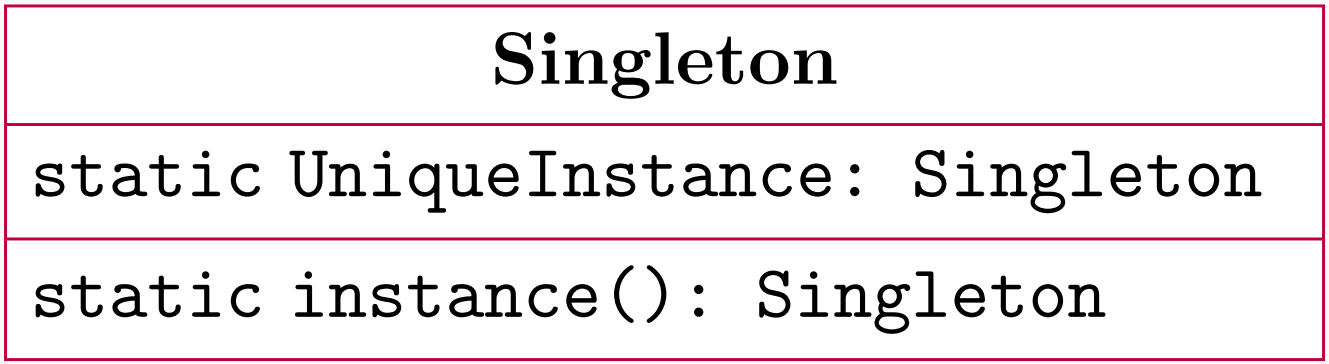
Singleton correct (v1)

```
public class Singleton {
    private String nom = "TOTO !";
    protected static Singleton instanceUnique = creerInstance();

    private static Singleton creerInstance() {
        try {
            Singleton s = new Singleton();
            s.nom = "UNIQUE";
            return s;
        }
        catch (Exception e) { return null; }
    }

    public static Singleton getInstance() { return instanceUnique; }

    public Singleton() throws Exception {
        if (instanceUnique != null)
            throw new Exception("Interdit !");
    }

    public String toString(){
        return "Je suis l'instance unique " + nom;
    }
}
```

Singleton correct (v2)

```
public class Singleton {
    private String nom;
    protected static Singleton instanceUnique = creerInstance();

    private static Singleton creerInstance() {
        Singleton s = new Singleton();
        s.nom = "UNIQUE";
        return s;
    }

    public static Singleton getInstance() { return instanceUnique; }

    private Singleton() { }

    public String toString() {
        return "Je suis l'instance unique " + nom;
    }
}
```

Singleton (Diagramme)



Patrons de conception structurels

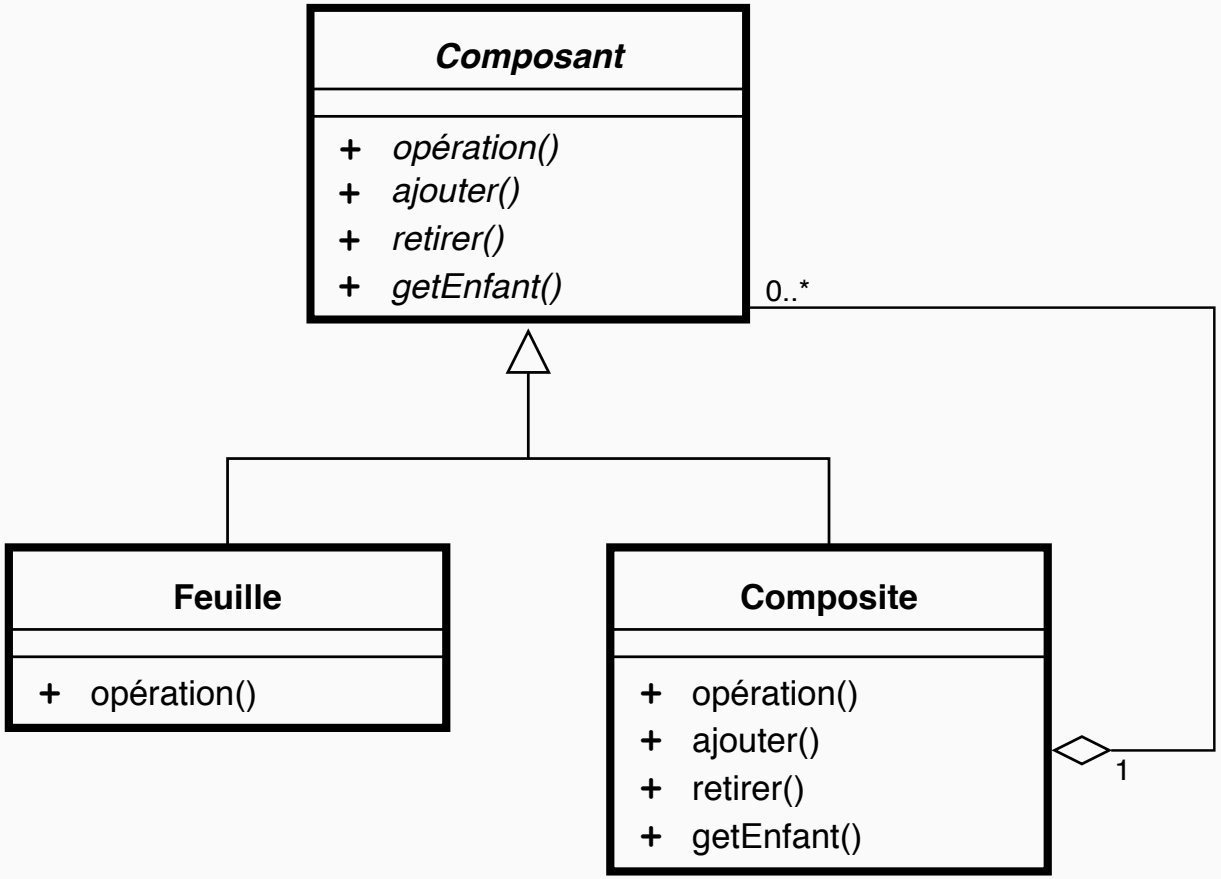
Composite (Principe)

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Composite (Exemple)

Employés. . .

Composite (Diagramme)



Composite (Utilisation)

Indications d'utilisation :

- on souhaite représenter des hiérarchies ou des emboîtements
- on souhaite que l'utilisateur n'ait pas à se préoccuper de la différence entre:
  - des combinaisons d'objets d'une part
  - des objets "atomiques" d'autre part

$\rightarrow$ tous les objets de la structure composite peuvent être traités de façon uniforme.

Decorator (Principe)

Attach additional responsibilities to an object dynamically.
Decorators provide a flexible alternative to subclassing for extending functionality.

Decorate (Exemple 1)

Formes colorées

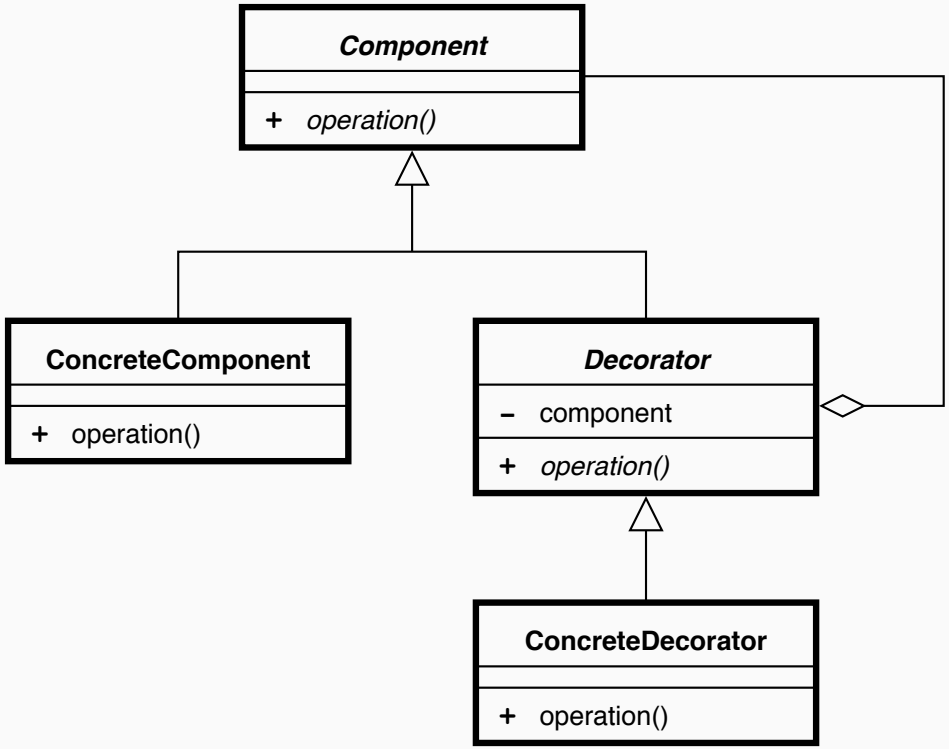
Decorate (Exemple 2)

```
public interface Plat { // Component
    int nbCalories();
}

public class Fromage implements Plat { // ConcreteComponent
    int nbCalories(){ return 1000; }
}

public class Chocolat implements Plat { // ConcreteComponent
    int nbCalories(){ return 1300; }
}

public class Yaourt implements Plat { // ConcreteComponent
    int nbCalories(){ return 600; }
}
```

Decorate (Exemple 2)

```
abstract class PlatModifie implements Plat { // Decorator
    private Plat aliment;

    public PlatModifie(Plat aliment){ this.aliment = aliment; }

    int nbCalories() { return aliment.nbCalories(); }
}

class PlatAllégé extends PlatModifié { // ConcreteDecorator
    private double allegement(){ return 0.5; }

    public PlatAllégé(Plat aliment){ super(aliment); }

    int nbCalories() {
        return super.nbCalories()*allegement();
    }
}
```

Decorator (Diagramme)



Utilisation

Indications d'utilisation :

- on veut ajouter dynamiquement des comportements à des objets individuels, sans affecter les autres (de la même classe), et de façon réversible;
- on veut pouvoir composer ces fonctionnalités les unes avec les autres;
- on veut éviter la prolifération des sous-classes.

Emacs 26.2 (Org mode 9.1.9)