R5.A.08 : Qualité de développement

(Certaines diapos ©I. Borne et figures refactoring.guru)

Cours Design Patterns 2 :

- Intention interfaces : Bridge et Facade
- Intention operations : Command
- Intention extensions : Iterator et Decorator

Classification des patterns selon leur intention

Intention	Patterns
Interfaces	Adapter, Facade, Composite, Bridge
Responsability	Singleton, Observer, Mediator, Proxy, Chain of Responsability, Flyweight
Construction	Builder, Factory Method, Abstract Factory, Prototype, memento
Operations	Template Method, State, Strategy, Command, Interpreter
Extensions	Decorator, Iterator, Visitor

Intention d'interface

- L'interface d'une classe est l'ensemble des méthodes et attributs qu'une classe permet aux objets d'autres classes d'accéder.
- Facade fournit une interface simplifiée à un groupe de sous-systèmes ou un sous-système complexe.
- Bridge se concentre sur l'implémentation d'une abstraction

Le patron Bridge

Bridge découple une abstraction ou une interface de son implémentation afin que les deux puissent varier indépendamment.

Motivation

Développement d'une Tod*o*-list, et on veut de la flexibilité sur sa présentation (liste d'items, contact avec puce, nombre).

On veut pouvoir changer les fonctionnalités de la liste de base ; ajout d'une possibilité de tri, de priorité, ...

Pour cela on développe un groupe de classes de liste, chacune fournissant une façon pour afficher la liste et organiser les informations.

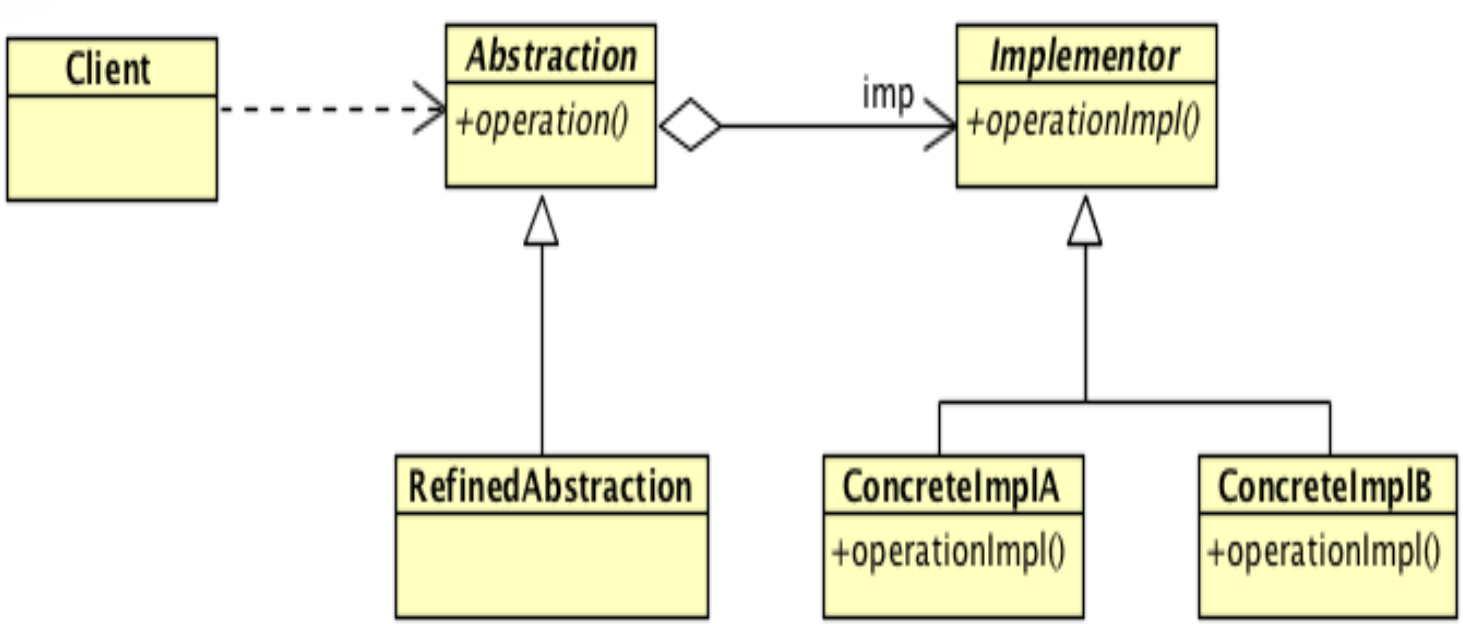
Mais le nombre de combinaisons devient impraticable.

Il est préférable de séparer la représentation de la *Todo*-list de son implémentation.

Quand utiliser le patron Bridge

- Pour éviter un lien permanent entre une structure abstraite et son implémentation concrète.
- Les abstractions et les implémentations doivent être extensibles par sous-classage.
- Le sous-classage est approprié mais on veut gérer les deux aspects du système séparément.
- Tout changement de l'implémentation doit être invisible des clients
- Pour cacher les détails d'implémentation aux clients. Les changements dans l'implémentation devraient de ne pas avoir d'impact sur les clients

Structure du pattern Bridge



Les rôles des classes

- **Abstraction** (abstract class) : définit l'interface abstraite et maintient la référence sur l'implémenteur
- **RefinedAbstraction** (concrete class) : étend l'interface définie par l'Abstraction
- **Implementor** (abstract class or interface) : définit l'interface pour les classes d'implémentation
- **ConcreteImplementor** (concrete class) : hérite de Implementor ou implémente l'interface Implementor

Bénéfices et défauts de Bridge

Bénéfices

- L'implémentation peut être sélectionnée ou échangée à l'exécution.
- L' abstraction et l'implémentation peuvent être étendues ou composées indépendamment.

Défauts/conséquences

- Il peut y avoir une double indirection
- Il est important de définir proprement quelles responsabilités appartiennent à l'abstraction fonctionnelle et lesquelles appartiennent à la classe d'implémentation interne

Exemple du pattern Bridge

Des formes géométriques auxquelles on souhaite associer des couleurs.
Mélange de hiérarchies à la fois dans les interfaces et dans les implémentations.

Exemple : solution avec Bridge

```
// Implementor interface
interface Device {
    void turnOn();
    void turnOff();
    void setVolume(int volume);
    boolean isEnabled();
    int getVolume();
}
```

Exemple : solution avec Bridge

```
// Concrete Implementor 1
class TV implements Device {
    private boolean on = false; private int volume = 30;
    @Override
    public void turnOn() {
        on = true; System.out.println("TV is turned on.");
    }
    @Override
    public void turnOff() {
        on = false; System.out.println("TV is turned off.");
    }
    @Override
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("TV volume set to " + this.volume);
    }
    @Override
    public boolean isEnabled() { return on; }
    @Override
    public int getVolume() { return volume; }
}
```

Exemple : solution avec Bridge

```
// Concrete Implementor 2
class Radio implements Device {
    private boolean on = false; private int volume = 20;
    @Override
    public void turnOn() {
        on = true; System.out.println("Radio is turned on.");
    }
    @Override
    public void turnOff() {
        on = false; System.out.println("Radio is turned off.");
    }
    @Override
    public void setVolume(int volume) {
        this.volume = volume;
        System.out.println("Radio volume set to " + this.volume);
    }
    @Override
    public boolean isEnabled() { return on; }
    @Override
    public int getVolume() { return volume; }
}
```

Exemple : solution avec Bridge

```
// Abstraction
abstract class RemoteControl {
    protected Device device;
    public RemoteControl(Device device) {
        this.device = device;
    }
    public void togglePower() {
        if (device.isEnabled()) {
            device.turnOff();
        } else {
            device.turnOn();
        }
    }
    public void volumeUp() {
        device.setVolume(device.getVolume() + 10);
    }
    public void volumeDown() {
        device.setVolume(device.getVolume() - 10);
    }
}
```

Exemple : solution avec Bridge

```
// Refined Abstraction 1
class BasicRemote extends RemoteControl {
    public BasicRemote(Device device) {
        super(device);
    }
}

// Refined Abstraction 2
class AdvancedRemote extends RemoteControl {
    public AdvancedRemote(Device device) {
        super(device);
    }

    public void mute() {
        device.setVolume(0);
        System.out.println("Device is muted.");
    }
}
```

Exemple : solution avec Bridge

```
// Client Code
public class BridgePatternDemo {
    public static void main(String[] args) {
        Device tv = new TV();
        Device radio = new Radio();

        RemoteControl basicRemote = new BasicRemote(tv);
        RemoteControl advancedRemote = new AdvancedRemote(radio);

        System.out.println("Testing Basic Remote with TV:");
        basicRemote.togglePower(); // Turn on TV
        basicRemote.volumeUp();    // Increase TV volume
        basicRemote.volumeDown();  // Decrease TV volume

        System.out.println("\nTesting Advanced Remote with Radio:"
        advancedRemote.togglePower(); // Turn on Radio
        advancedRemote.volumeUp();    // Increase Radio volume
        ((AdvancedRemote) advancedRemote).mute(); // Mute Radio
    }
}
```

Avantages du patron Bridge

• Découplage de l'abstraction et de l'implémentation : l'abstraction (`RemoteControl`) et l'implémentation (`Device`) peuvent évoluer indépendamment ;
• Flexibilité accrue : vous pouvez ajouter de nouveaux types de devices ou de télécommandes (*remote control*) sans modifier le code existant ;
• Facilité de maintenance améliorée : les modifications apportées aux détails de l'implémentation n'affectent pas l'abstraction ou le code client.

Patron de conception Façade

Problème

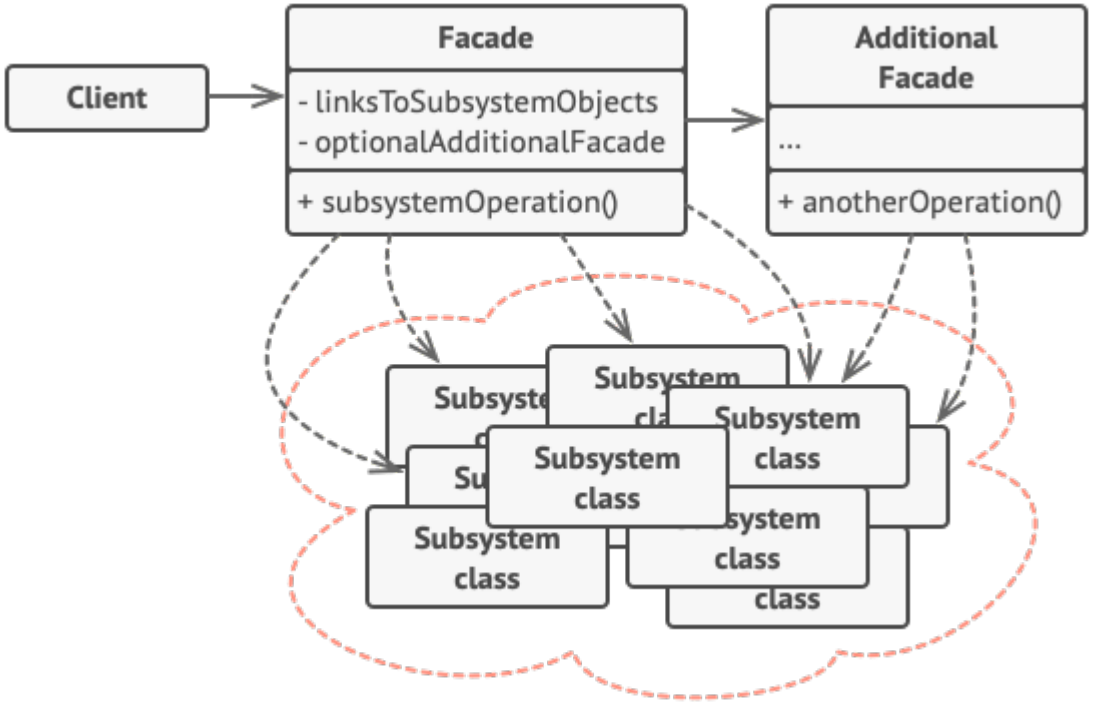
• On aimerait travailler avec de multiples objets appartenant à une grande bibliothèque
• Il faudrait initialiser tous ces objets, résoudre leurs dépendances puis exécuter les méthodes dans le bon ordre
• Le code écrit aura de fortes dépendances avec la bibliothèque (dépendances noyées dans le code client, maintenance difficile)

Solution

• Façade : classe qui expose une simple interface à un ou plusieurs systèmes
• La classe expose une fonctionnalité limitée, uniquement les fonctions utiles au client

Analogie avec le monde réel : le guichet unique (de l'auto-entrepreneur, ...)

Structure du patron Façade



Patron de conception Command

Intention : transformer une requête en un objet qui contient toutes les informations nécessaires à l'exécution de la requête. Ceci permet de : passer une requête comme argument de méthode, retarder l'exécution d'une requête (la mettre en file d'attente), faire un undo, …

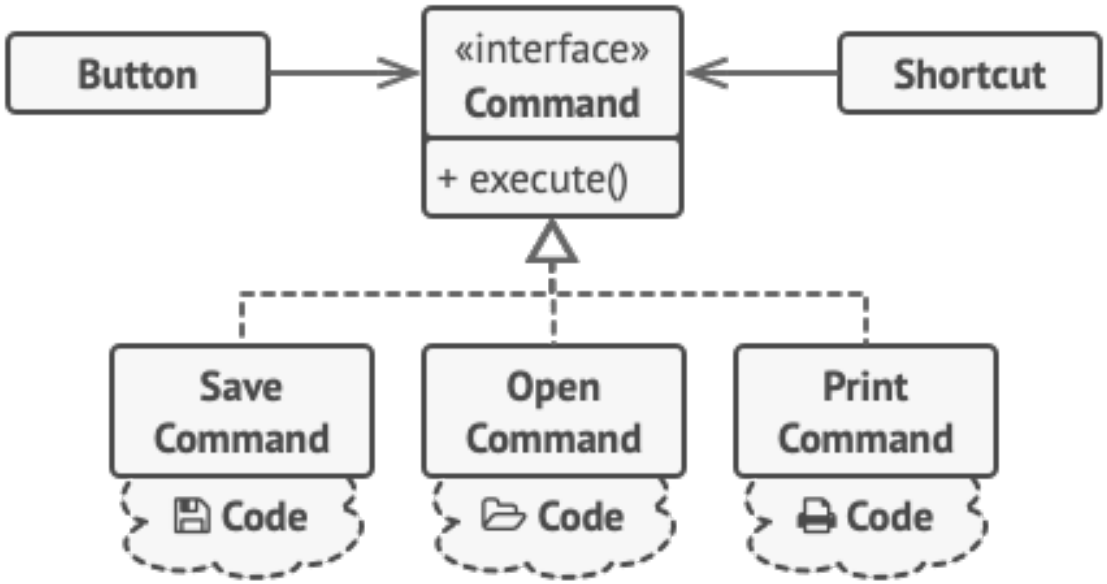
Problème :

- Un comportement utilisable de différentes façons
- Ex : un undo dans un éditeur de texte peut être utilisé via un bouton dédié dans la barre d'outil, le menu « Edition » ou avec le raccourci « Ctl-Z »
- Dans un design à base de classes (UndoButton, UndoShortcut, UndoMenuItem) est-ce qu'on duplique le code dans ces différentes classes ?
  Nooon, surtout pas.

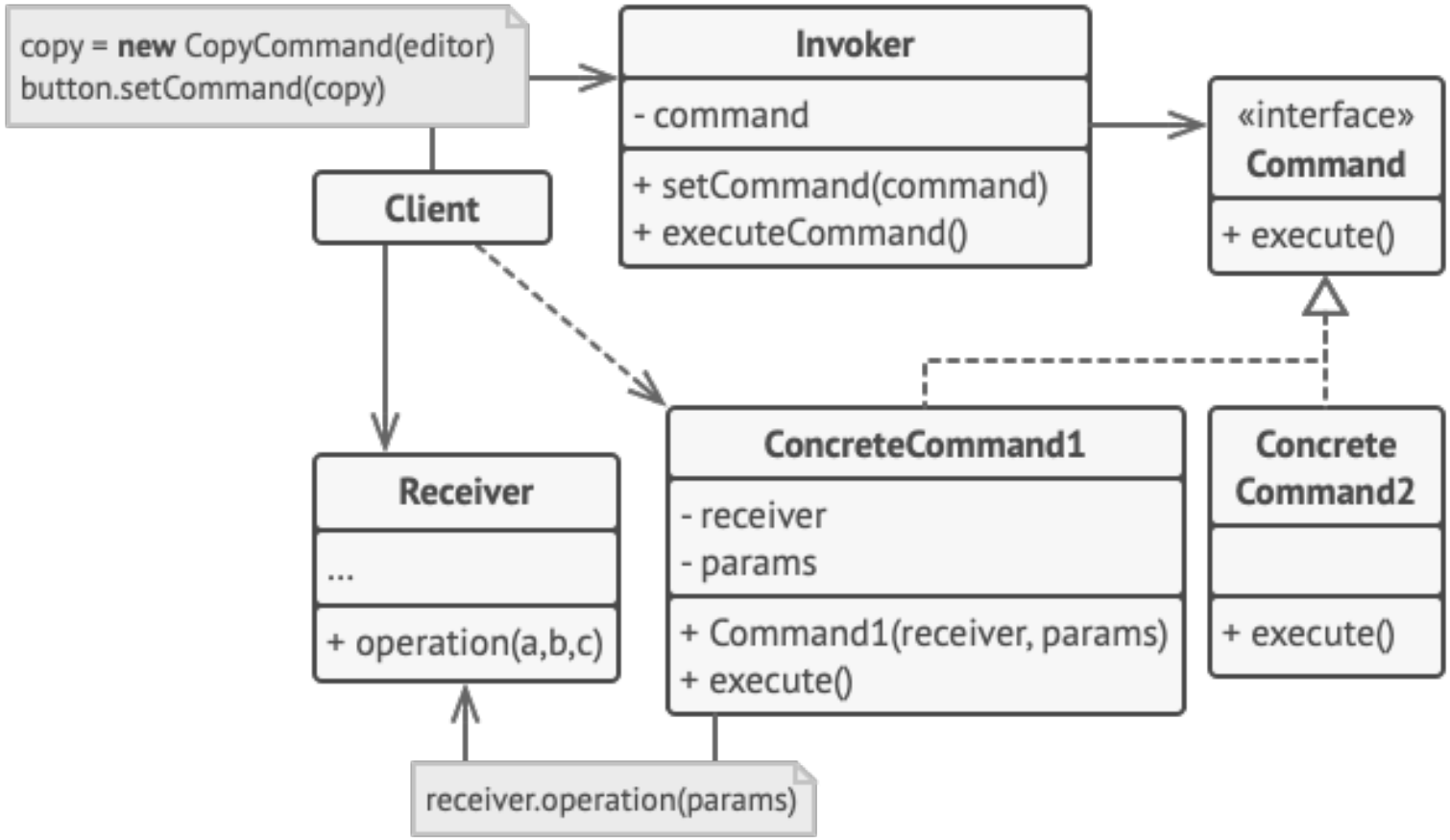
Patron de conception Command

Solution : instancier une classe de commande et y embarquer l'ensemble des informations de la requête : objet receveur, sa méthode et ses arguments

Ensuite appeler une méthode unique sur cet objet : la méthode execute



Structure du patron Command



Exemple d’instanciation du patron

```
// Command interface
public interface Command { void execute(); }


// Receiver class
class Light {
  public void turnOn() {
    System.out.println("The light is On.");
  }
  public void turnOff() {
    System.out.println("The light is Off.");
  }
  public void toggle() {
    System.out.println("The light is toggled.");
  }
}
```

Exemple d’instanciation du patron

```
// Concrete Command to turn the light on
class TurnOnCommand implements Command {
  private Light light;
  public TurnOnCommand(Light light) {
    this.light = light;
  }
  @Override public void execute() {
    light.turnOn();
  }
}

// The same with Concrete Command to turn off
class TurnOffCommand implements Command { … }
```

Exemple d’instanciation du patron

```
// Invoker class
class RemoteControl {
  private Command command;
  public void setCommand(Command command) {
    this.command = command;
  }
  public void pressButton() {
    command.execute();
  }
}
```

Exemple d'instanciation du patron

```
// Client code
public class CommandPatternDemo {
  public static void main(String[] args) {
   // Create a light (Receiver)
   Light livingRoomLight = new Light();
   // Create concrete command objects
   Command turnOn=new TurnOnCommand(livingRoomLight);
   Command turnOff=new TurnOffCommand(livingRoomLight);
   // Create the invoker (RemoteControl)
   RemoteControl remote = new RemoteControl();
   // Turn the light on
   remote.setCommand(turnOn);
   remote.pressButton(); // Output: The light is On.
   // Turn the light off
   remote.setCommand(turnOff);
   remote.pressButton(); // Output: The light is Off.
  }
}
```

Avantages de la solution

• Découple l'expéditeur et le récepteur : l'expéditeur ne connaît que l'interface de commande, pas l'implémentation du récepteur ;

• Facile à étendre et à ajouter de nouvelles commandes : vous pouvez ajouter de nouvelles commandes sans modifier le code existant ;

• Prend en charge les opérations d'annulation/rétablissement : en ajoutant une méthode undo() à l'interface Command, le patron Command peut prendre en charge la fonctionnalité d'annulation.

Patron de conception Iterator

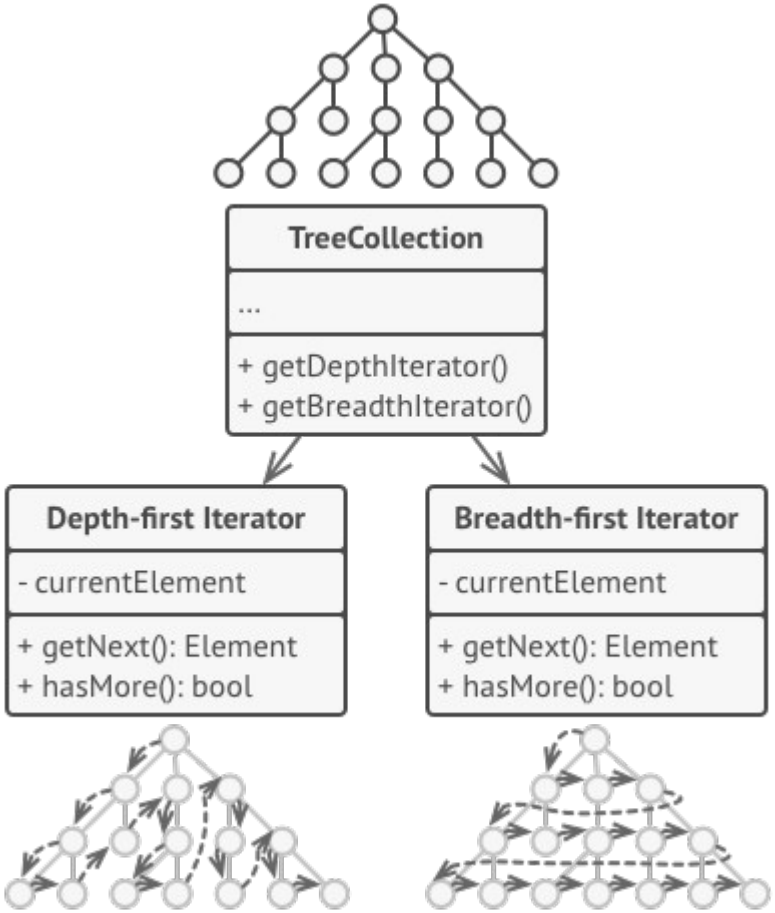
Intention : Permettre aux utilisateurs d'une collection de **parcourir** ses éléments de façon **séquentielle**

Problème :

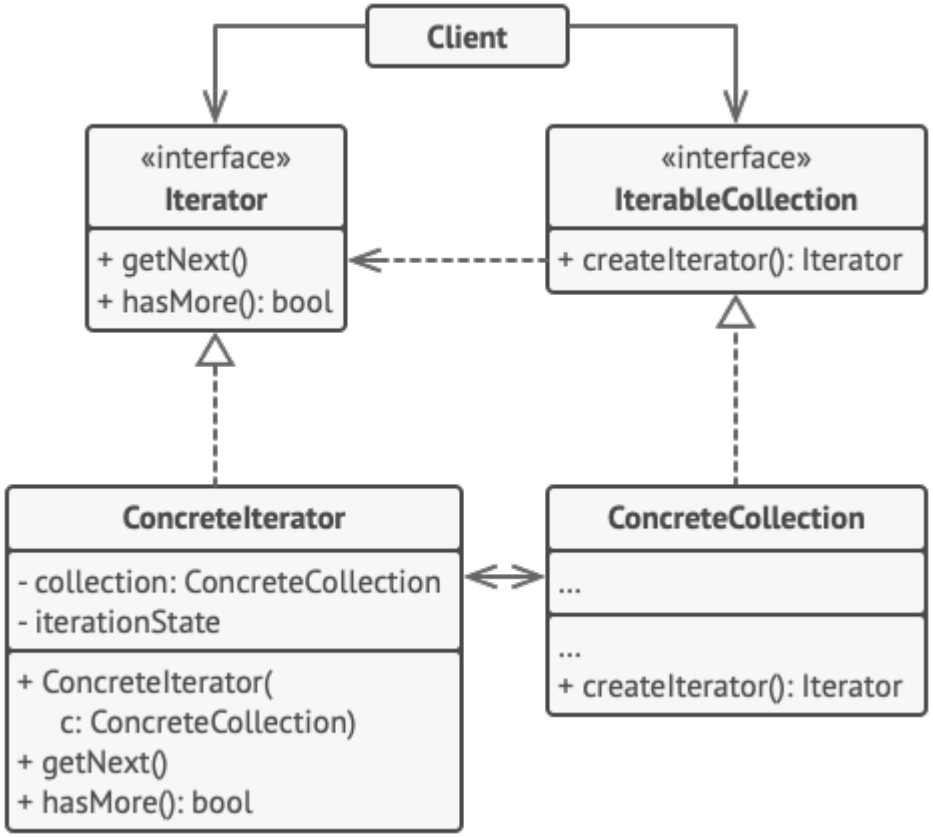
- On a défini une classe représentant une certaine abstraction qui encapsule une collection d'éléments
- Nous aimerions permettre le parcours séquentiel de ces éléments (ça peut être une collection de type arbre et nous aimerions permettre un parcours en profondeur d'abord, ou en largeur d'abord, ou les 2, ou …)
- La classe munie de méthodes permettant ce(s) type(s) de parcours aura trop de **responsabilités**
- De plus, le code client ne doit pas se préoccuper de quel type concret de collection la classe utilise

Patron de conception Iterator

Solution : extraire le comportement de traversée d'une collection dans un objet séparé appelé itérateur. Cet objet stocke les détails du parcours, la position actuelle, …



Structure du patron de conception Iterator



Exemple avec le patron de conception Iterator

```
// Iterator interface
public interface Iterator {
    boolean hasNext(); // or hasMore()
    Object next(); // or getNext()
}

// Container interface
public interface Container {
    Iterator getIterator(); // or createIterator()
}
```

Exemple avec le patron de conception Iterator

```
// Concrete collection class
public class NameRepository implements Container {
    private String[] names = {"John", "Jane", "Robert", "Lucy"};
    @Override
    public Iterator getIterator() { return new NameIterator(); }
    // Inner class implementing the Iterator interface
    private class NameIterator implements Iterator {
        int index;
        @Override
        public boolean hasNext() { return index < names.length;  }
        @Override
        public Object next() {
            if (this.hasNext()) { return names[index++]; }
            return null;
        }
    }
}
```

Exemple avec le patron de conception Iterator

```
// Client Code
public class IteratorPatternDemo {
   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for (Iterator iter = namesRepository.getIterator(); iter.hasNext(); ) {
         String name = (String) iter.next();
         System.out.println("Name: " + name);
      }
   }
}
```

Output :

```
Name: John
Name: Jane
Name: Robert
Name: Lucy
```

Avantages du patron Iterator

• Simplifie l'interface de la collection : permet à la structure interne de la collection de changer sans affecter le code client.

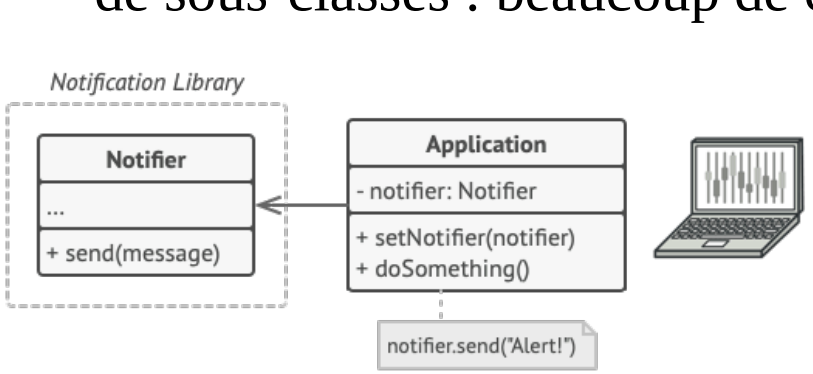
• Prend en charge différents algorithmes de parcours : plusieurs itérateurs peuvent être créés pour différentes manières de parcourir la collection.

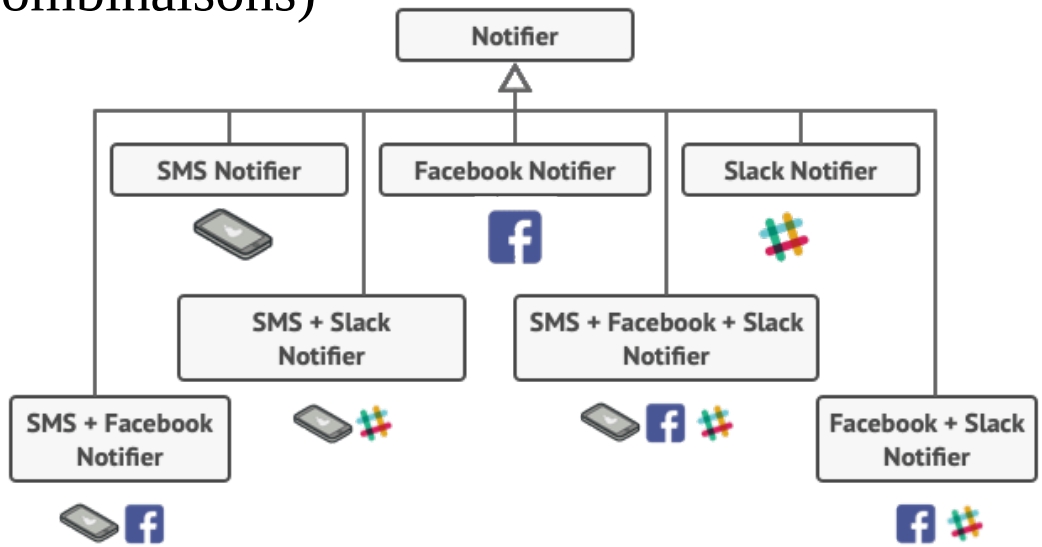
• Découple les classes de collection des opérations de parcours : la collection n'a pas besoin d'implémenter de code de parcours ; elle délègue cette responsabilité à l'itérateur.

Patron de conception Decorator

Intention : permet d'attacher de nouveaux comportements aux objets en plaçant ces objets à l'intérieur d'objets wrapper spéciaux contenant les comportements.

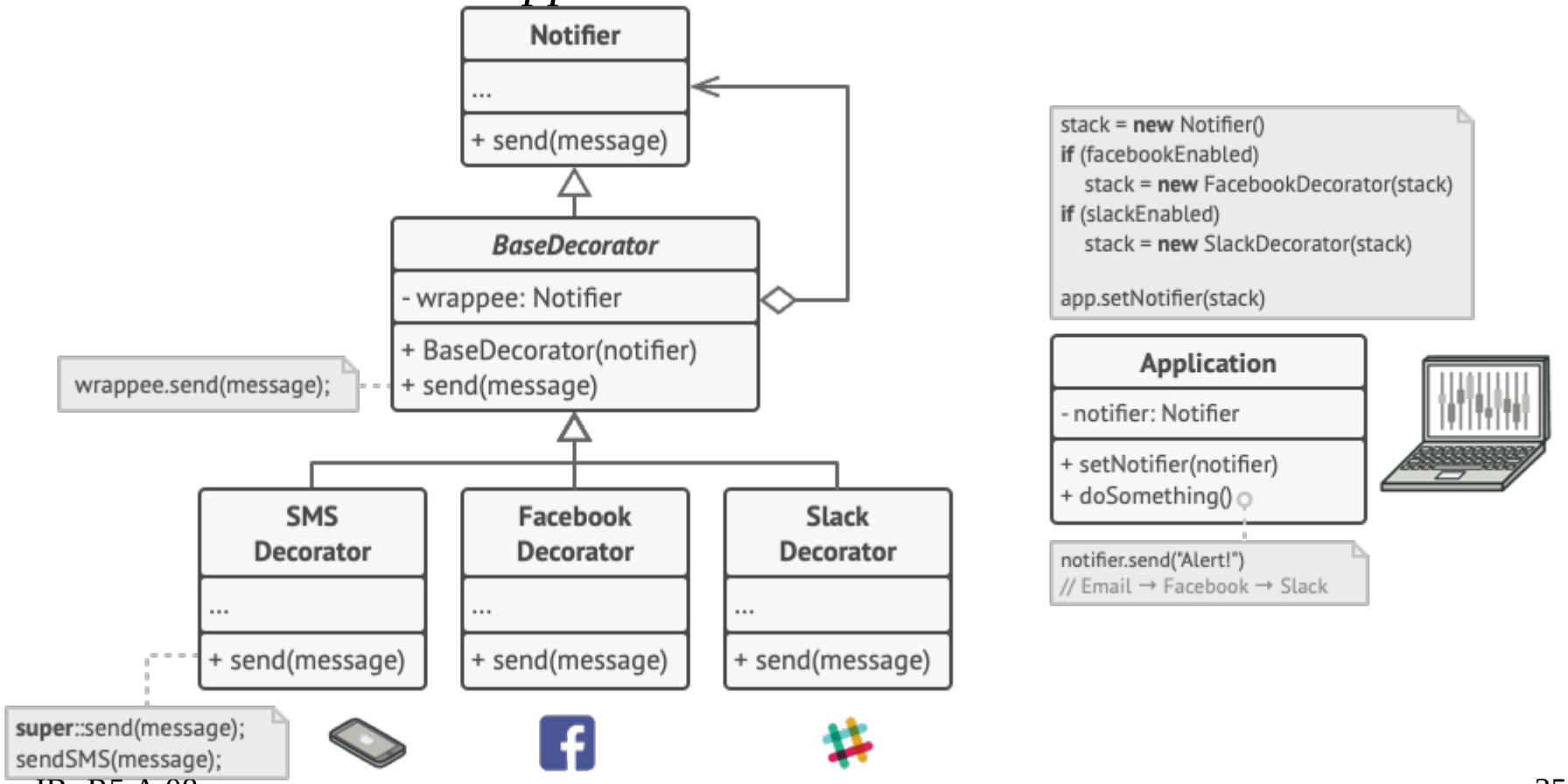
Problème : supposons une app qui utilise un notifier
Au fil du temps, besoin de développer divers notifiers (trop de sous-classes : beaucoup de combinaisons)



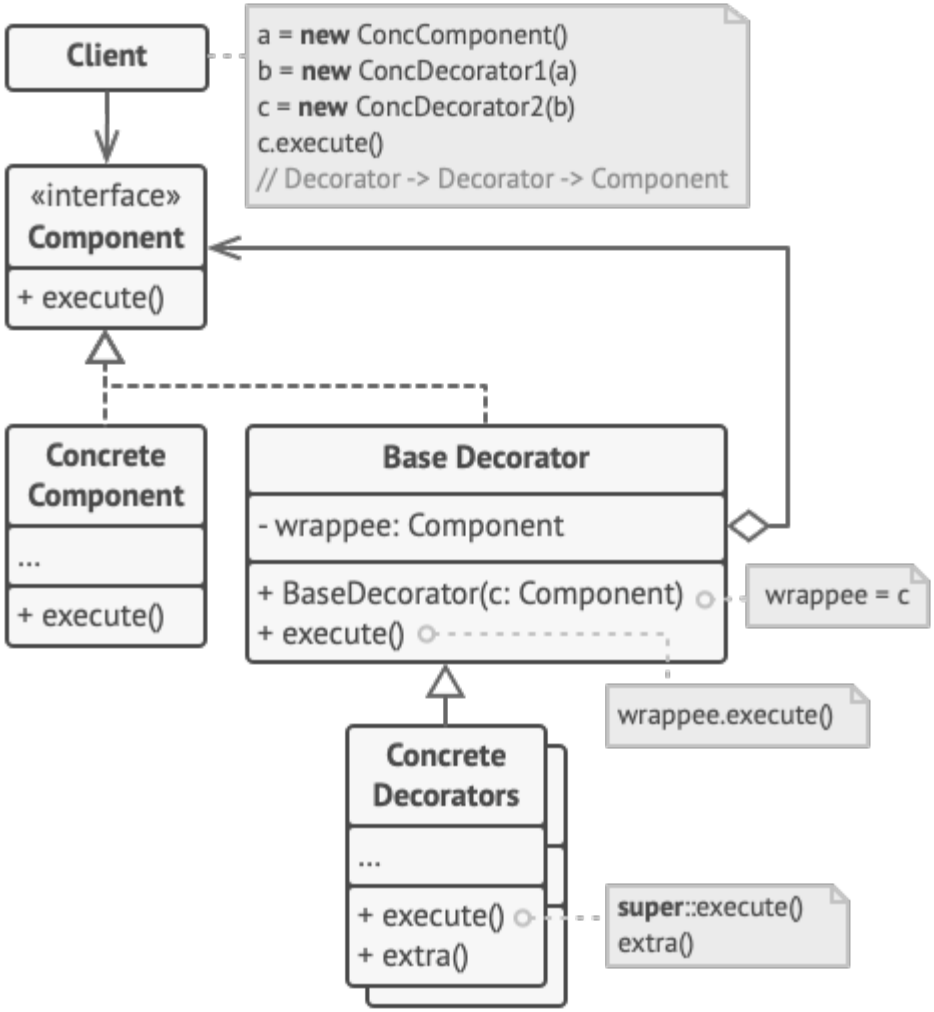


Patron de conception Decorator

Solution : combiner héritage et agrégation (délégation) pour créer un décorateur ou *wrapper*



Structure du patron de conception Decorator



Exemple avec le patron Decorator

```
// Component interface
public interface Coffee {
    String getDescription();
    double getCost();
}
// Concrete Component
class PlainCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Plain Coffee";
    }
    @Override
    public double getCost() {
        return 5.0;
    }
}
```

Exemple avec le patron Decorator

```
// Abstract Decorator
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;
    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }
    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }
}
```

Exemple avec le patron Decorator

```
// Concrete Decorator 1
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 1.5;
    }
}
```

Exemple avec le patron Decorator

```
// Concrete Decorator 2
class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost() + 0.5;
    }
}
```

Exemple avec le patron Decorator

```
// Client Code
public class DecoratorPatternDemo {
    public static void main(String[] args) {
        // Create a Plain Coffee
        Coffee plainCoffee = new PlainCoffee();
        System.out.println(plainCoffee.getDescription() + " Cost: $"
                                + plainCoffee.getCost());
        // Add Milk to the Coffee
        Coffee milkCoffee = new MilkDecorator(plainCoffee);
        System.out.println(milkCoffee.getDescription() + " Cost: $"
                                + milkCoffee.getCost());
        // Add Sugar to the Coffee
        Coffee sugarMilkCoffee = new SugarDecorator(milkCoffee);
        System.out.println(sugarMilkCoffee.getDescription() + " Cost: $"
                                + sugarMilkCoffee.getCost());
    }
}
```

Avantages du patron Decorator

- Alternative flexible au sous-classement : vous pouvez ajouter de nouvelles fonctionnalités aux objets sans modifier leur code ;

- Ajout de comportement dynamique : vous pouvez ajouter ou supprimer des fonctionnalités au moment de l'exécution ;

- Combine plusieurs comportements : différents décorateurs peuvent être combinés pour ajouter plusieurs fonctionnalités.