



Génie Logiciel : Conception

Patrons de Conception, niveau Chevalier

UQÀM | Département d'informatique

Crédit Images: Pixabay & Pexels

Sébastien Mosser

INF 5153 - Cours #9 - A19



GitHub repository page for `ace-lectures / pattern-repository`:

- Code**, **Issues** 0, **Pull requests** 1, **Security**, **Insights**, **Settings**
- Branch: master**, **pattern-repository / README.md**
- mosser url fixed** by **d1ae44** on Mar 21
- 1 contributor**
- 77 Lines (57 sloc)**, **3.65 KB**

Catalogue des patrons de conception du GoF

- Auteur : Sébastien Mosser (UQAM)
- Contributeurs: Mireille Blay-Fornarino (UCA), Philippe Collet (UCA)
- Version : 2019.03
- Intégration continue : [build passing](#)

Ce dépôt est un support au cours INF-5153 de l'Université du Québec à Montréal. Il recense des implémentations en Java des patrons de conception vus dans ce cours. Il est créé en collaboration avec l'Université Côte d'Azur (IUT & Polytech).

<https://github.com/ace-lectures/pattern-repository/blob/master/README.md>

Crédits

UNIVERSITÉ **CÔTE D'AZUR**



Philippe Collet

Université Côte d'Azur
Laboratoire I3S, SPARKS

“Prof d’UML” de père en fils depuis 1999

https://www.i3s.unice.fr/Philippe_Collet

Design Patterns page from https://sourcemaking.com/design_patterns

Hooray! After 3 years of work, we've finally released a new ebook on design patterns! [Check it out »](#)

Design Patterns

In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Uses of Design Patterns

Design patterns can speed up the development process by providing well-known solutions for common problems. They help developers apply certain software design techniques to certain problems. Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem.

Attention aux implémentations

In addition, patterns allow developers to communicate using well-known, well-understood names for

Menu		Objectif		
		Création	Structure	Comportement
Portée	classe	Factory <i>Adapter</i> <i>Interpreter</i> <i>Template Method</i>		
	objet	Abstract Factory <i>Builder</i> <i>Prototype</i> Singleton Adapter <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> Command <i>Iterator</i> <i>Mediator</i> <i>Memento</i> Observer <i>State</i> Strategy <i>Visitor</i>	

Trois Niveaux dans l'apprentissage

- Apprentissage "par cœur"
 - Je sais réciter la définition d'un patron
- Compréhension du problème de conception
 - Je sais appliquer le patron dans d'autres contextes
- Approche "intelligente"
 - Je sais utiliser le bon patron au bon moment

Accumulation n'est pas synonyme de bon.

(au mieux ça rime)

Singleton



```

public class ChocolateBoiler {

    private boolean empty;
    private boolean boiled;

    public ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

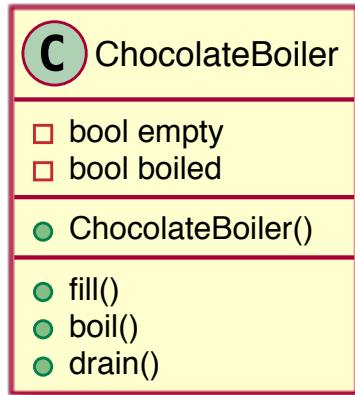
    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            boiled = true;
        }
    }
}

```

La fabrique de Chocolat



La fabrique de Chocolat

```

class ChocolateFactory {

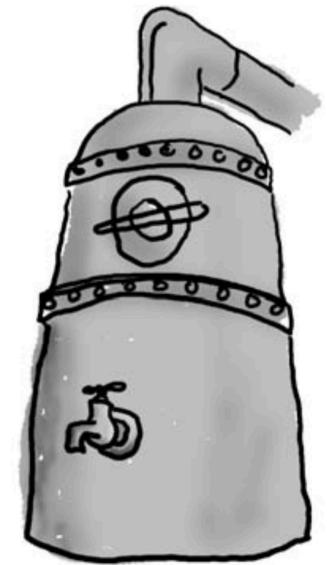
    private ChocolateBoiler theBoiler;

    public ChocolateFactory() {
        this.theBoiler = new ChocolateBoiler();
    }

    public void makeChocolateBars() {
        // ... preprocessing
        if (theBoiler.isEmpty()) {
            theBoiler.fill();
            theBoiler.boil();
            theBoiler.drain();
        } else {
            throw new RuntimeException("Busy Boiler!");
        }
        // ... postprocessing
    }

}

```



```

public void makeChocolateTruffles() {
    // ... preprocessing
    theBoiler = new ChocolateBoiler();
    if (theBoiler.isEmpty()) {
        theBoiler.fill();
        theBoiler.boil();
        theBoiler.drain();
    } else {
        throw new RuntimeException("Busy Boiler!");
    }
    // ... postprocessing
}

```

**Erreur de débutant
Manque de documentation
... whatever ...**



Problème

N'importe qui peut créer une instance de ChocolateBoiler.

Pourtant il n'existe physiquement qu'un seul dispositif.

C'est au développeur de faire attention.



C'est au développeur de faire attention.



C	Singleton
□	<u>Singleton.uniqueInstance</u>
■	Singleton()
●	<u>getInstance(): Singleton</u>

Instance statique
Constructeur } privé
 Accesseur } public

SINGLETON
Object Creational

Intent

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

Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.



Design Patterns
 Elements of Reusable Object-Oriented Software
 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
 Foreword by Grady Booch
PRENTICE HALL INTERNATIONAL EDITIONS

```
class Singleton {
    private static Singleton uniqueInstance = null;
    private Singleton() { ... }
    public static Singleton getInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    Singleton s = new Singleton();
}
Singleton s = Singleton.getInstance();
```

```

public class ChocolateBoiler {

    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance = null;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new ChocolateBoiler();
        return uniqueInstance;
    }

}

```

ChocolateBoiler

- ☐ bool empty
- ☐ bool boiled
- ☐ ChocolateBoiler uniqueInstance
- ChocolateBoiler()
- getInstance(): ChocolateBoiler
- fill()
- boil()
- drain()

```

class ChocolateFactory {

    private ChocolateBoiler theBoiler;
    public ChocolateFactory() {
        // this.theBoiler = new ChocolateBoiler();
        this.theBoiler = ChocolateBoiler.getInstance();
    }

    public void makeChocolateBars() {
        // ... preprocessing
        if (theBoiler.isEmpty()) {
            theBoiler.fill(); theBoiler.boil();
            theBoiler.drain();
        } else {
            throw new RuntimeException("Busy Boiler!");
        }
        // ... postprocessing
    }

    public void makeChocolateTruffles() {
        // ... preprocessing
        // theBoiler = new ChocolateBoiler();
        theBoiler = ChocolateBoiler.getInstance();
        if (theBoiler.isEmpty()) {
            theBoiler.fill(); theBoiler.boil();
            theBoiler.drain();
        } else {
            throw new RuntimeException("Busy Boiler!");
        }
        // ... postprocessing
    }
}

```

Même en cas d'erreur de programmation,
on parle toujours à la même instance
de ChocolateBoiler !

Il faut faire un tout petit peu plus attention en cas de multi-threading

STUPID code, seriously?¶

This may hurt your feelings, but you have probably written STUPID code already. I have too. But, what does that mean?

- Singleton
- Tight Coupling
- Untestability
- Premature Optimization
- Indescriptive Naming
- Duplication

Singleton, bon ou mauvais ?

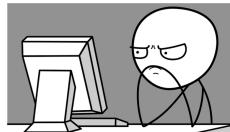
<https://williamdurand.fr/2013/07/30/from-stupid-to-solid-code/>

Singleton : un anti-patron ?

- Le bon singleton ? Il implémente une instance unique, mais c'est un bon singleton... NON
 - Gestion d'un seule instance avec une responsabilité unique
 - Pas d'état, sur la gestion de l'instance
 - Exemple : formatter, cache, logger, interface d'accès à du matériel
- Mauvais singleton ?
 - Représentation d'un utilisateur qui vient de se logger
 - Représentation d'un plateau de jeu partagé
 - Facilité d'accès des valeurs dans plusieurs zones/couches de l'application



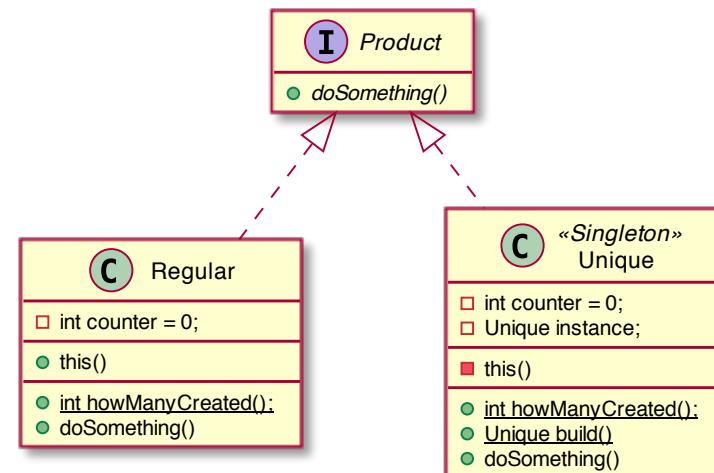
Why singletons suck...



- **Graphe de dépendances entre objets caché**
- **Difficiles à tester**
 - En fait, c'est un couplage fort : en étant globaux, c'est tout leur environnement qui doit gérer leur état
 - Ils sont difficiles à « mocker », on doit écrire du code spécifique pour les tester
 - Ils ne sont pas vraiment extensibles par héritage
- **Pas bon pour la concurrence**
 - Ou pas *thread-safe*
 - Ou goulot d'étranglement en cas d'accès multiples et concurrents
- **Solution : Injection de dépendances**
 - cours ISA au S8 (ceci est un message publicitaire de Sébastien Mosser)

P. Collet

7



```
public class Regular implements Product {
    private static int counter = 0;
    public static int howManyCreated() { return counter; }

    public Regular() {
        System.out.println("==>> A new regular product is instantiated <===");
        counter++;
    }

    @Override
    public void doSomething() { ... }
}

public class Unique implements Product {
    private static int counter = 0;
    public static int howManyCreated() { return counter; }

    private static Unique instance = null;

    public static Unique build() {
        if (instance == null)
            instance = new Unique();
        return instance;
    }

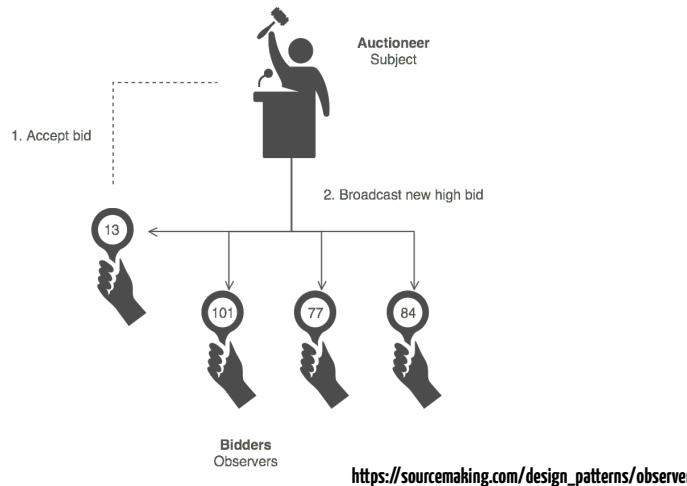
    private Unique() {
        System.out.println("==>> A new unique product is instantiated <===");
        counter++;
    }

    @Override
    public void doSomething() { ... }
}
```

Observateur



Exemple

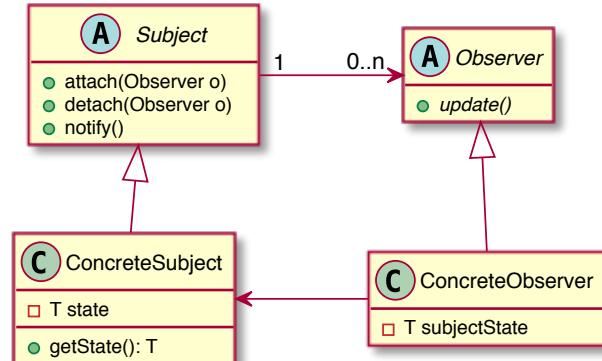


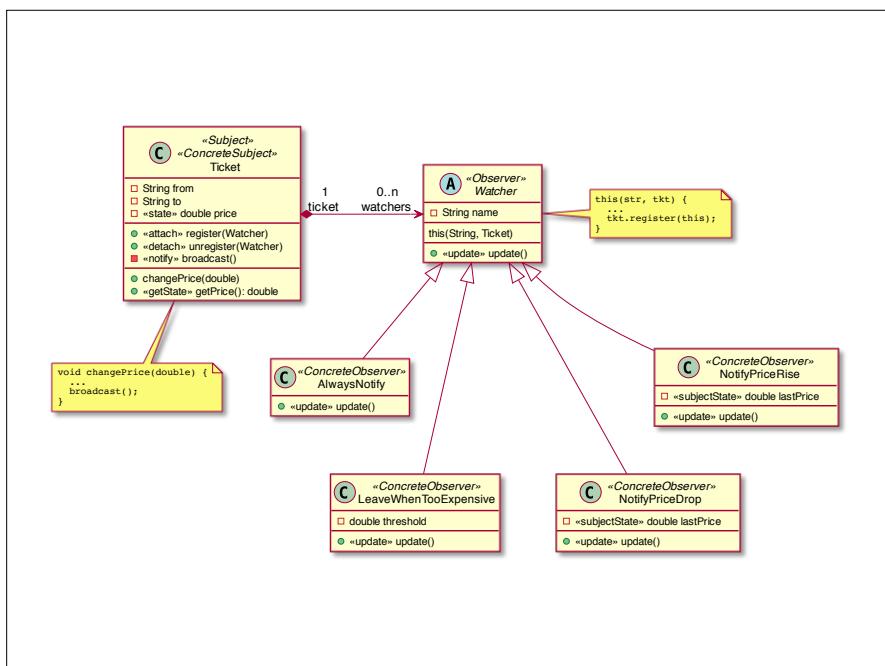
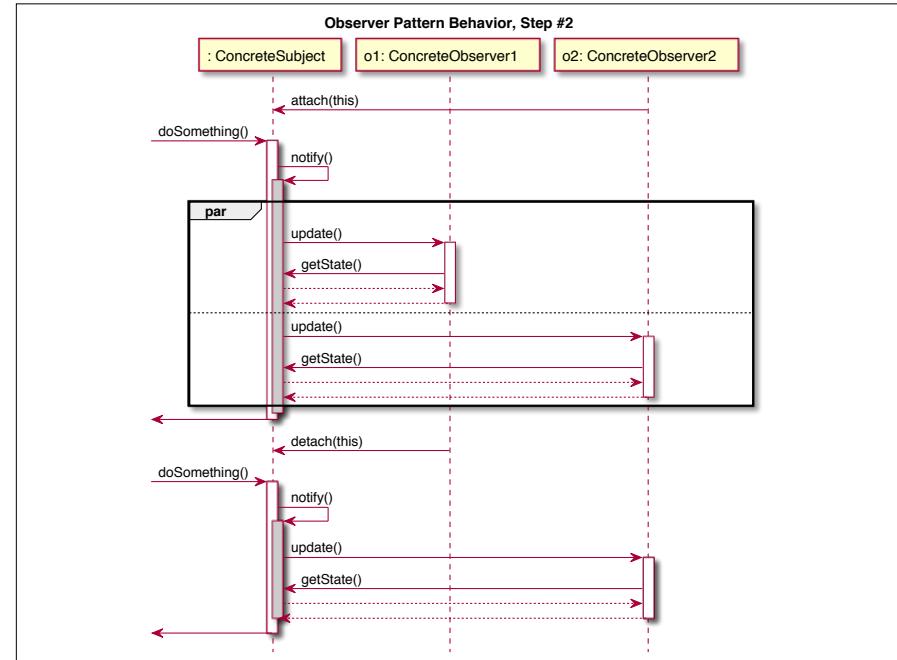
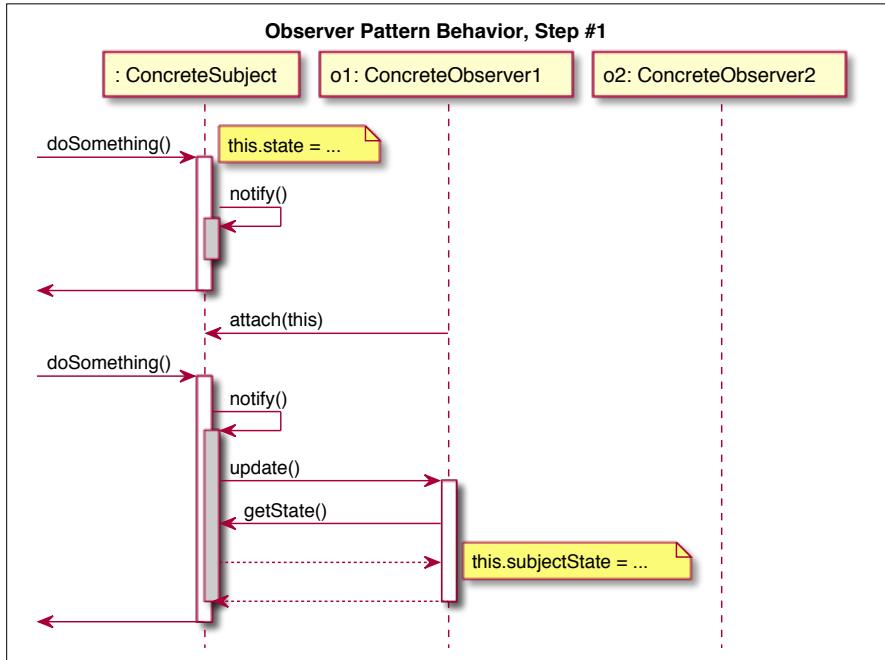
Problème

- Une application a deux aspects co-dépendant
- Un changement sur un objet impose de modifier les autres
- On ne sait pas à l'avance combien d'objets sont impactés
- Les objets en questions ne peuvent être fortement couplés

Intention

Définir une relation “1-n” de telle façon que si on change l’objet unique d’état, tout ses dépendants sont prévenus et mis à jour automatiquement.





```

public class Ticket {
    private String from;
    private String to;
    private double price;

    public Ticket(String from, String to, double price) {
        // ...
        this.watchers = new HashSet<>();
    }

    public void changePrice(double percentage) {
        // ...
        broadcast();
    }

    public double getPrice() {
    }

    private Set<Watcher> watchers;
    public void register(Watcher w) {
        this.watchers.add(w);
    }

    public void unregister(Watcher w) {
        this.watchers.remove(w);
    }

    private void broadcast() {
        Set<Watcher> targets = new HashSet<>(this.watchers);
        targets.parallelStream().forEach(Watcher::update);
    }
}
  
```

Invasif !

```

public class LeaveWhenTooExpensive extends Watcher {
    private double threshold;
    public LeaveWhenTooExpensive(String n, Ticket t, double percentage) {
        super(n,t);
        this.threshold = t.getPrice() * percentage;
    }
    @Override
    public void update() {
        super.update();
        if(this.ticket.getPrice() < threshold)
            System.out.println(" -> Sending email notification, price is still correct");
        else {
            System.out.println(" -> too expensive, not interested anymore");
            this.ticket.unregister(this);
        }
    }
}

```

Conséquences

- Variations abstraites entre sujets et Observateurs
- Absence de couplage (géré au niveau méta)
- “Broadcast” des communications avec les observateurs

Où le trouver ?

- API Java (depuis JDK 1.0)
- Interface **Observer**, Class **Observable**
- Principe classique en Interaction Personne-Machine
 - L'activation d'un bouton déclenche des comportements
- C'est une simplification “objet” des architectures Pub/Sub
 - Micro-services et chorégraphies d'évenements

Décorateur



Exemple

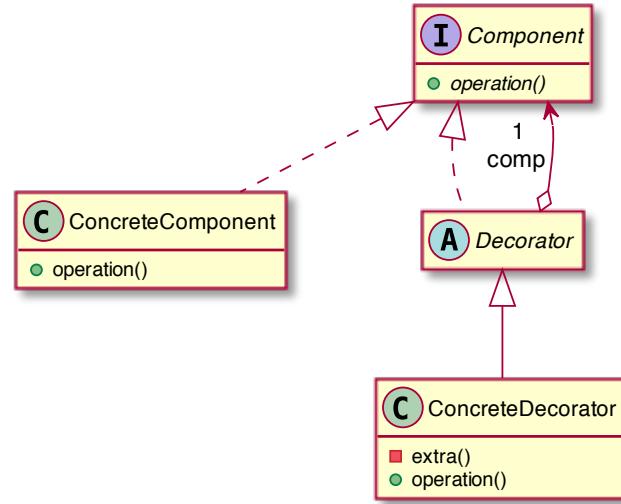
- Dans le système d'info de la marque “Van Horton” :
 - Comment calculer le prix d'un café ?
 - D'un café avec lait de soja ?
 - D'un café avec crème fouettée et une dose d'expresso ?
 - D'un café avec ...

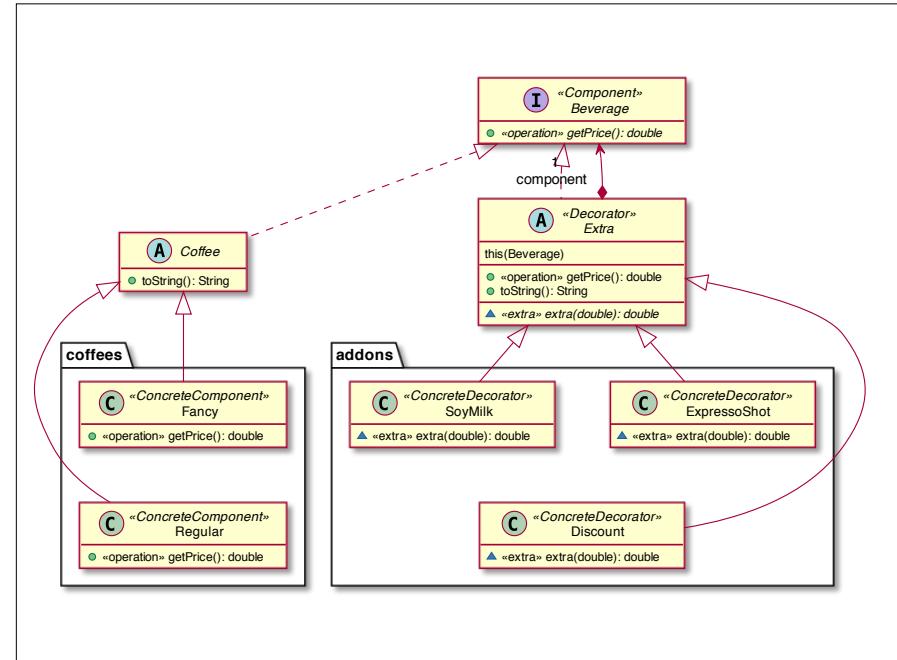
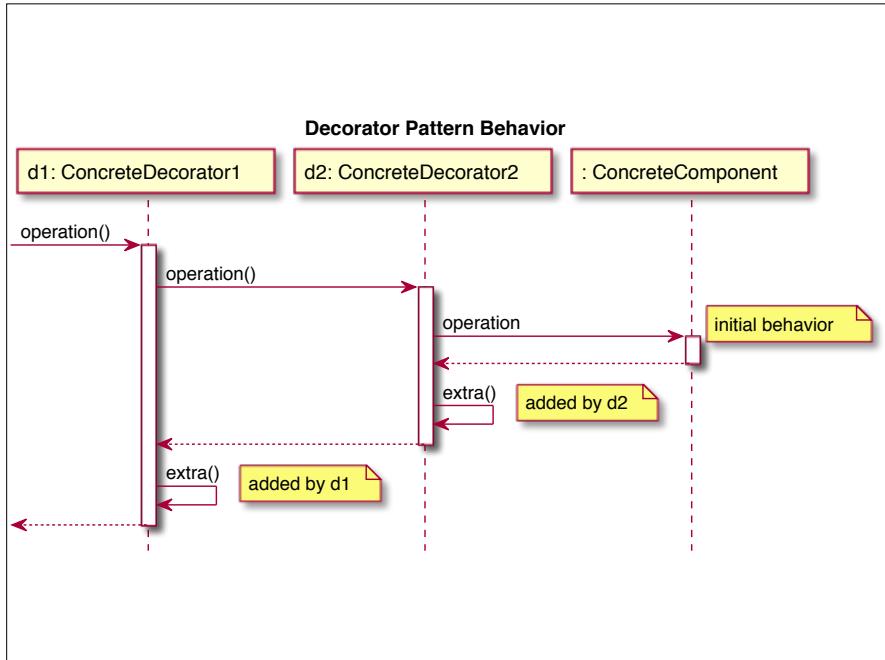
Problème

- Un objet doit pouvoir changer dynamiquement de comportement
- Ces comportements ne sont pas forcément connus à l'avance
- L'approche par héritage génère une explosion combinatoire
- Les changements sont au niveau de l'objet et pas de la classe

Intention

- Attacher dynamiquement au niveau de l'objet des capacités additionnelles
- Fournir une alternative plus flexible à l'héritage





```

public static void main(String[] args) {
    System.out.println("# Selling plain coffees");
    displayBeverage(new Regular());
    displayBeverage(new Fancy());

    System.out.println("\n# Adding extras");
    displayBeverage(new SoyMilk(new Regular()));
    displayBeverage(new EspressoShot(new Fancy()));
    displayBeverage(new SoyMilk(new EspressoShot(new Fancy())));
    displayBeverage(new EspressoShot(new EspressoShot(new Regular())));

    System.out.println("\n# Messing up with discounts (aka order matters)");
    displayBeverage(new Discount(new Regular()));
    displayBeverage(new Discount(new EspressoShot(new Regular())));
    displayBeverage(new EspressoShot(new Discount(new Regular())));
}

private static void displayBeverage(Beverage b) {
    System.out.println(b + ": $" + String.format("%.2f", b.getPrice()));
}
  
```

```

public class Regular extends Coffee {
    @Override public double getPrice() { return 2.0; }
}

public class Fancy extends Coffee {
    @Override public double getPrice() { return 3.25; }
}

public class EspressoShot extends Extra {
    public EspressoShot(Beverage inner) { super(inner); }
    @Override protected double extra(double price) { return price + 0.80; }
}

public class SoyMilk extends Extra {
    public SoyMilk(Beverage inner) { super(inner); }
    @Override protected double extra(double price) { return price + 1.25; }
}
  
```

Conséquences

- Plus de flexibilité qu'avec une approche par héritage
- Evite la surcharge initiale de classe
- Le décorateur et son composant sont deux objets différents
- Pleins de tous petits objets, beaucoup d'appels inter objets

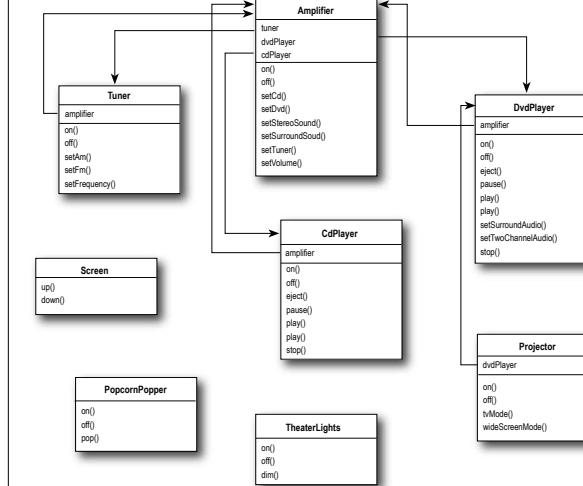
Façade

4

Où le trouver ?

- Système de définition d'Interfaces Graphiques
- Mise en forme de documents
 - CSS en Web, Format de fichier Word / LibreOffice
- L'API de gestion des I/O dans Java
 - `BufferedReader br = new BufferedReader(new FileReader(FILENAME))`

Exemple



Système
HiFi

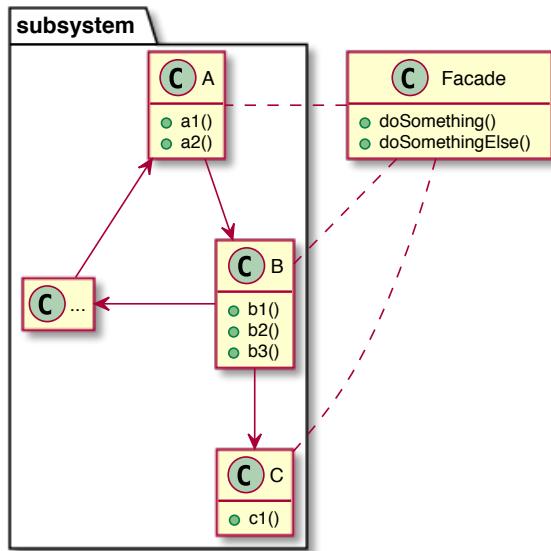
[HeadFirst]

Problème

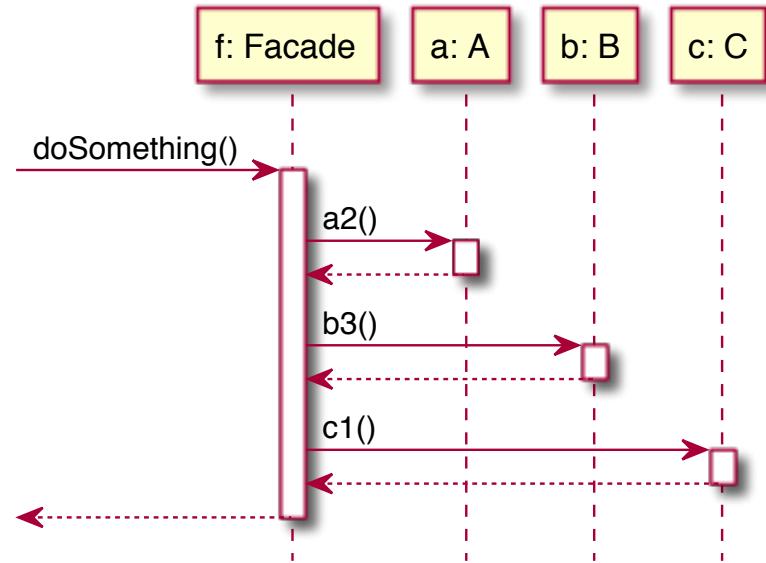
- L'application développée possède un ensemble d'interface complexe, qu'il faut coordonner
- Chaque client doit se coupler avec des sous-éléments du système
- Le sous-système peut difficilement évoluer

Intention

- Permettre de continuer à faire évoluer le sous-système
- Découpler client et sous-système pour ne pas avoir de dépendances fortes
- Fournir une interface unique et simplifiée qui servira à isoler le client du système



Facade Pattern Behavior



Conséquences

- Facilite l'utilisation du système
- Permet la diminution du couplage
- Plus flexible que de la visibilité (on peut outrepasser la facade)
- Permet de modifier le système sans modifier le client
- L'interface Facade peut-être trop restrictive

Fabrique



Où le trouver ?

- Classique en application réparties
 - Interfaces de services / micro-services
 - Mot clé : "API Gateway"
- Développement d'application par composants
 - Promotion / exposition d'interfaces

Exemple

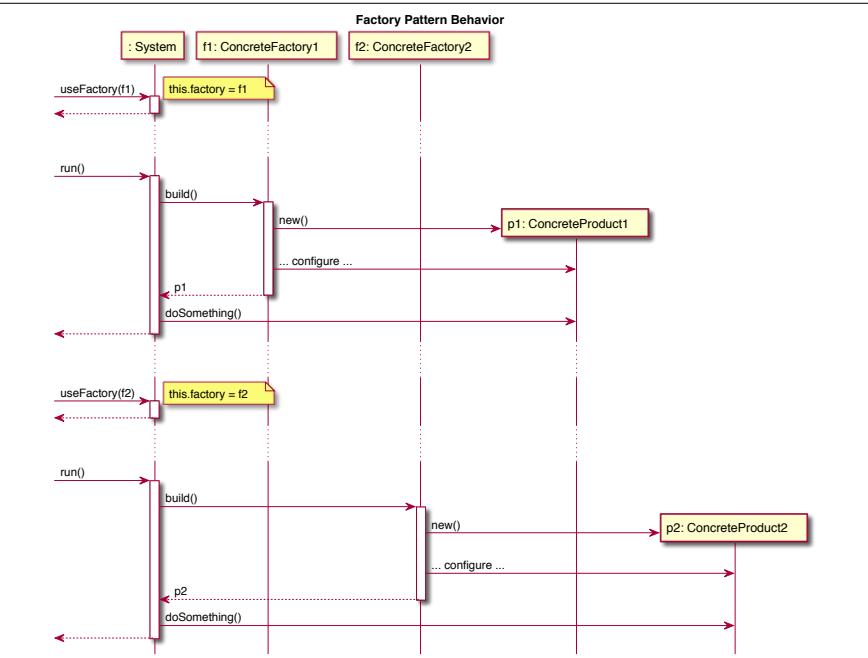
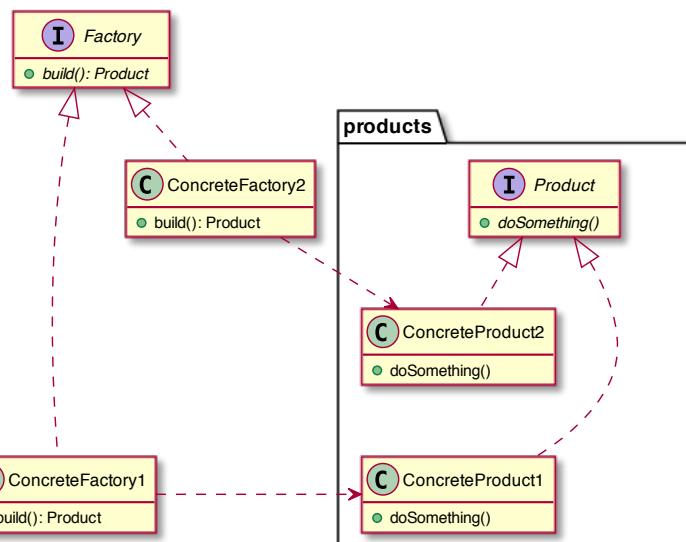
- Utilisation d'un système de journalisation (Log)
 - Le journal peut-être local ou distant
 - On peut journaliser en JSON, XML, TXT, ...
- On peut utiliser Log4J, SLF4J, util.Logger, ...
- On peut vouloir changer d'avis

Problème

- Masquer la complexité d'un cadriel
- Reposer uniquement sur les interfaces (abstractions)
- Configurer des objets potentiellement complexes
- Ne pas faire dépendre le client de cette complexité
- Impossible d'anticiper ce qu'il faut construire a priori

Intention

- Définir une interface pour la création d'un objet
- Laisser aux sous-classes le choix de l'instantiation
- Déléguer à une classe "qui sait faire" les instantiations complexes



```

public interface Shape {
    void draw();
}

public class Rectangle implements Shape {
    @Override
    public void draw() { ... }
}

public class Square implements Shape {
    @Override
    public void draw() { ... }
}

public class Circle implements Shape {
    @Override
    public void draw() { ... }
}

```

https://www.tutorialspoint.com/design_pattern/factory_pattern.htm

```

public class ShapeFactory {

    public Shape getShape(String shapeType){
        if(shapeType == null){
            return null;
        }
        if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        } else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        return null;
    }
}

```



Non, non, non !

Exemple de fabrique “correcte”

- Une interface “buildX”, avec X ce qu'on veut construire
 - *buildRectangle, buildTriangle, ...*
- Des familles de produits différentes :
 - *Des triangles en JavaFX, en Swing, en HTMLCanvas, ...*
- On choisit la fabrique et on l'utilise pour cette famille donnée.
- Les fabriques qui prennent des “strings” en paramètres et font un switch sont souvent de “mauvaises” fabriques.

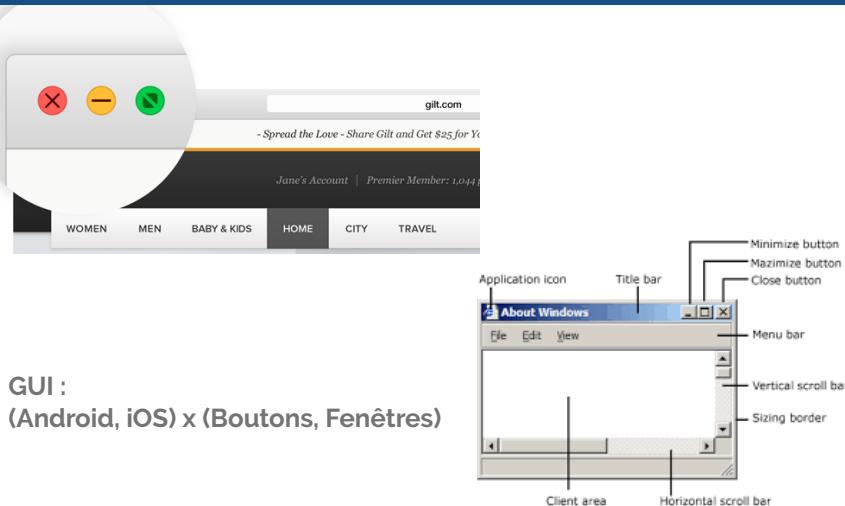
Conséquences

- La fabrication dispense d'être adhérent aux classes spécifiques dans le code
- Lien avec le patron GRASP de “Création”
- Gain en flexibilité dans les interfaces des objets construits

Où le trouver ?

- Dans le JDK:
 - Les API utilisant getInstance() et valueOf()
 - (souvent attachée à des singletons)
- Dans les cadriels réseaux
 - Construire la connexion au système distant
- Dans les approches de modélisation / génération de code
 - Une interface pour plusieurs implémentation possible

Exemple



Fabrique Abstraite

Le retour

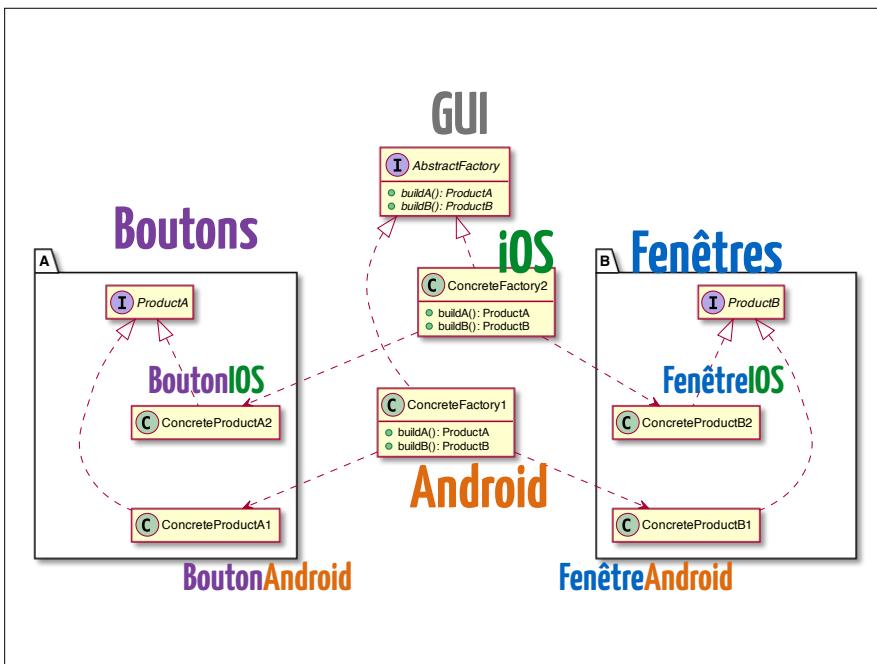
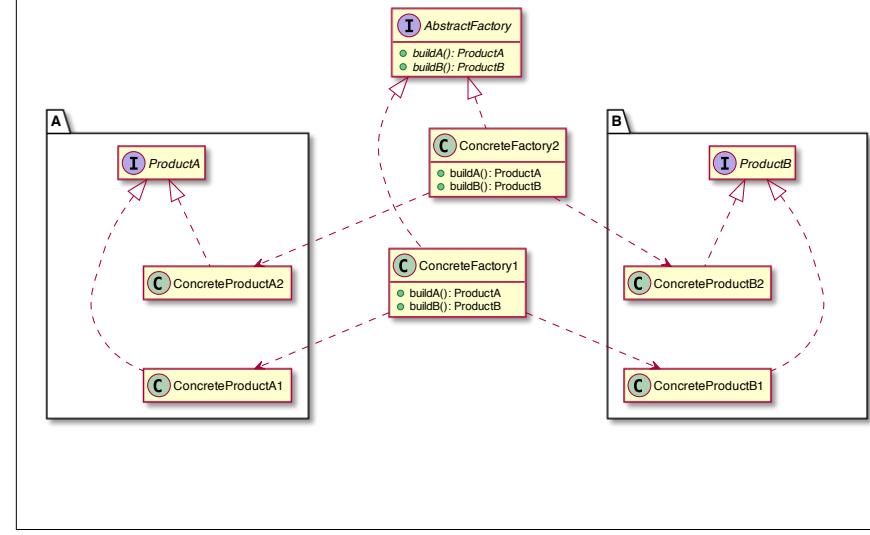


Problème

Comment garantir la consistance de familles de systèmes, proche mais différents ?

Intention

- Fournir une interface pour créer des familles d'objets
- Pas de couplage avec les implémentations réelles
- Configuration aisée d'une multitude de produits cohérents



Conséquences

- Isolation des classes concrètes
- Échange / Remplacement facile de famille de produit
- Maintien de la cohérence
- Difficile d'introduire de nouvelles familles de produits
 - Doit modifier toutes les fabriques ...

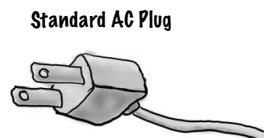
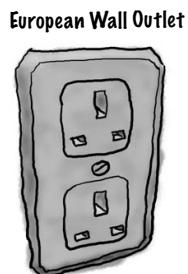
Où le trouver ?

- Dans les cadriels multi-plateformes
- Générateurs de code visant plusieurs cibles
- API gérant des collections uniformes
 - Python (defaultdic), Java Collections (presque)

Adaptateur



Exemple

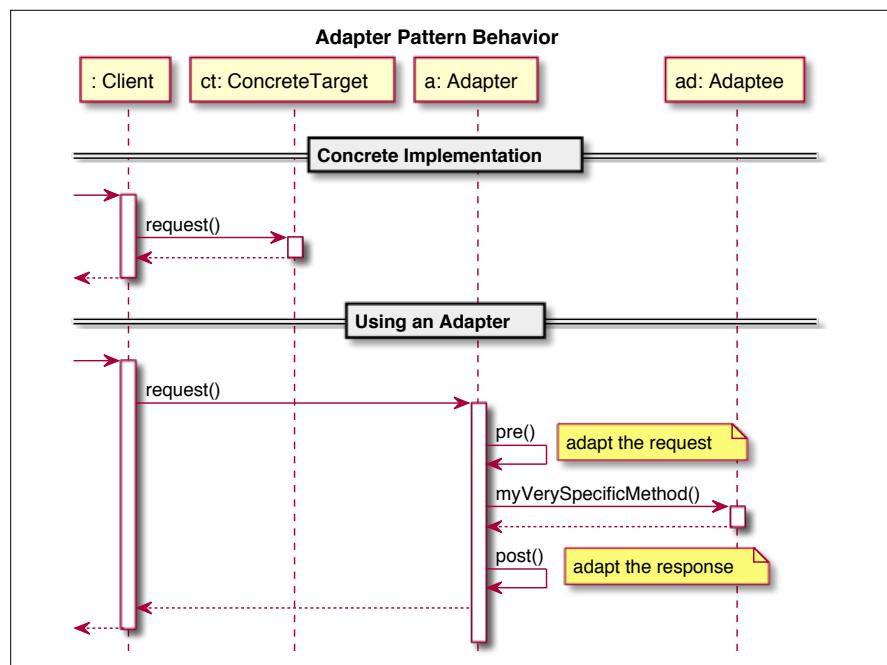
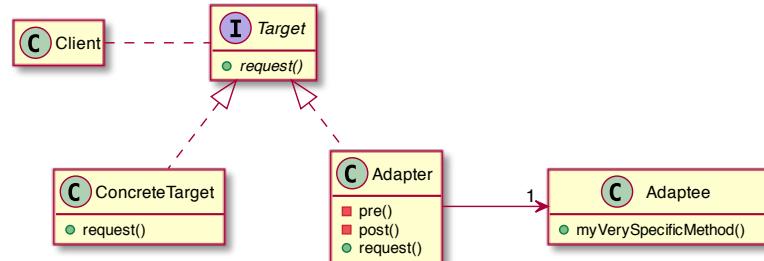


Problème

Comment intégrer du code défini selon une interface X alors que c'est une autre interface Y qui est attendu ?

Intention

- Convertir une interface en une autre
- Faire collaborer des objets qui ne pourraient pas en l'état
- Garantir de l'évolutivité sur les éléments adaptés



```
class Line {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("Line from point A(" + x1 + ";" + y1 + "), to point B(" + x2 + ";" + y2 + ")");
    }
}

class Rectangle {
    public void draw(int x, int y, int width, int height) {
        System.out.println("Rectangle with coordinate left-down point (" +
+ x + ";" + y + "), width: " + width + ", height: " + height);
    }
}

interface Shape {
    void draw(int x, int y, int z, int j);
}
```

<https://sourcemaking.com/>

```

class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter(Rectangle rectangle) {
        this.adaptee = rectangle;
    }

    @Override
    public void draw(int x1, int y1, int x2, int y2) {
        int x = Math.min(x1, x2);
        int y = Math.min(y1, y2);
        int width = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);
        adaptee.draw(x, y, width, height);
    }
}

```

<https://sourcemaking.com/>

Où le trouver ?

- Transformation d'espaces de couleurs (RGB, CMYK)
- En système distribué, quasiment partout
- Architectures hexagonales

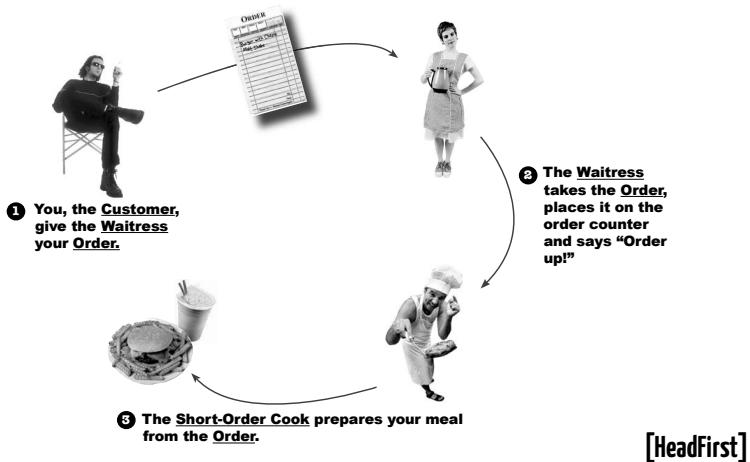
Conséquences

- Intégration aisée d'interfaces "exotiques"
- Mais tout à un prix :
 - Le code d'adaptation est potentiellement "sale"
 - Impossible d'adapter une famille de produit
 - Idem pour une hiérarchie de classes

Commande



Exemple

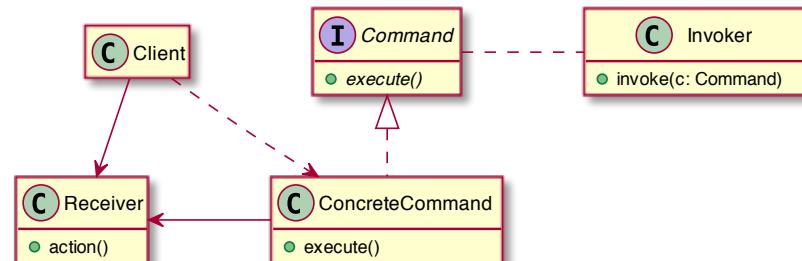


Problème

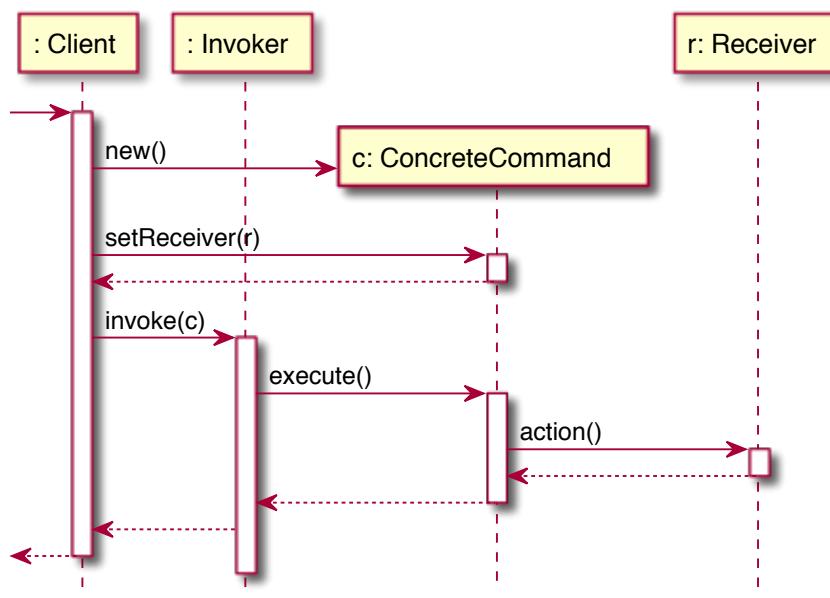
Comment réaliser un traitement sans forcément savoir qui va l'effectuer, ni de quoi il s'agit concrètement ?

Intention

- Encapsuler une requête comme un objet
- Découpler émission de la requête et exécution
- Permettre de défaire les traitements effectués
- Gérer des files d'attentes ou de priorités



Command Pattern Behavior



```

public class Light{
    private boolean on;
    public void switchOn(){ on = true; }
    public void switchOff(){ on = false; }
}

```

```

public interface Command{
    public void execute();
}

```

```

public class LightOnCommand implements Command{
    Light light;
    public LightOnCommand(Light light){
        this.light = light;
    }
    public void execute(){
        light.switchOn();
    }
}

```

<https://dzone.com/articles/design-patterns-command>

```

public class RemoteControl{
    private Command command;
    public void setCommand(Command command){
        this.command = command;
    }
    public void pressButton(){
        command.execute();
    }
}
public class Client{
    public static void main(String[] args) {
        RemoteControl control = new RemoteControl();
        Light light = new Light();
        Command lightsOn = new LightOnCommand(light);
        Command lightsOff = new LightsOffCommand(light);
        //switch on
        control.setCommand(lightsOn);
        control.pressButton();
        //switch off
        control.setCommand(lightsOff);
        control.pressButton();
    }
}

```

<https://dzone.com/articles/design-patterns-command>

Conséquences

- Découplage entre invocation et réalisation
- Une action est un objet comme les autres
- Possibilité de faire des “Macros-commandes”
 - En couplant avec le patron Composite
- Ouvert/Fermé sur l’évolution des commandes disponibles

Où le trouver ?

- Protocole de communication (e.g., réseau, inter-objets)
- Architectures événementielles
 - Micro-services, bus de messages répartis
- Système de gestion de version

