



**Génie Logiciel : Conception
Patrons de Conception (padawan)**

images: Pixabay

Sébastien Mosser
INF-5153, Hiver 2019, Cours #7

UQÀM 

1

Crédits

UNIVERSITÉ **CÔTE D'AZUR** 



Philippe Collet

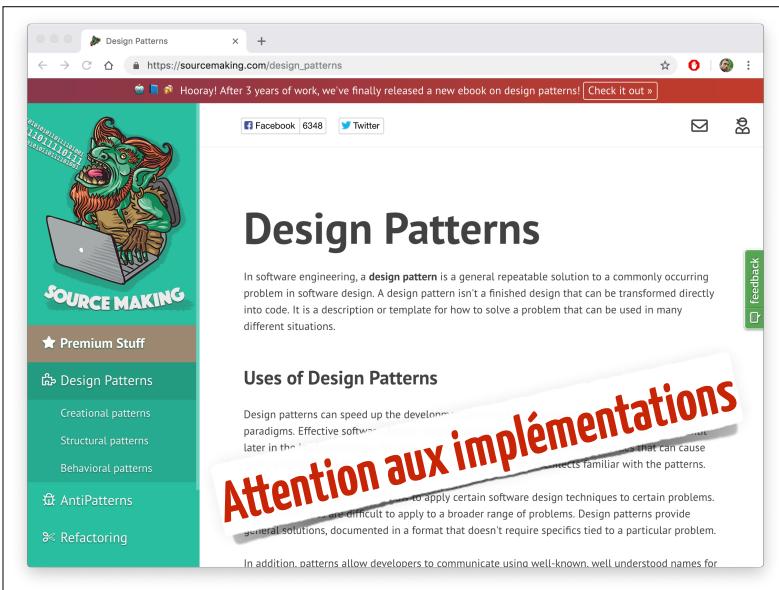
Université Côte d'Azur
Laboratoire I3S, SPARKS

"Prof d'UML" de père en fils depuis 1999

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

2

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



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 to reuse existing code and to apply certain software design techniques to certain problems. However, it's often difficult to apply a broader range of problems. Design patterns provide several 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

3

4

Accumulation n'est pas synonyme de bon.

(au mieux ça rime)

5

Observateur (rappels)

7

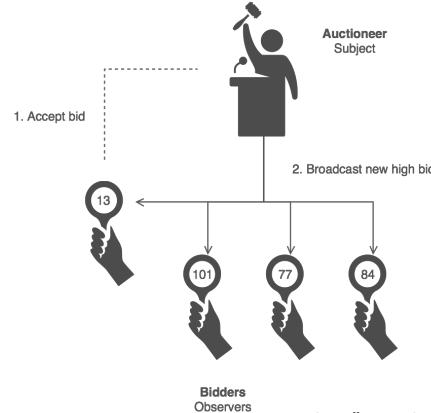
Mais au fond quelle différence y a t il entre le bon et le mauvais chasseur ?

Bon y faut expliquer tu vois y'a le mauvais chasseur, y voit un truc qui bouge y tire.
Le bon chasseur y voit un truc y tire,
mais c'est un bon chasseur.

<https://www.youtube.com/watch?v=QuGcoQJKXT8>

6

Exemple



https://sourcemaking.com/design_patterns/observer

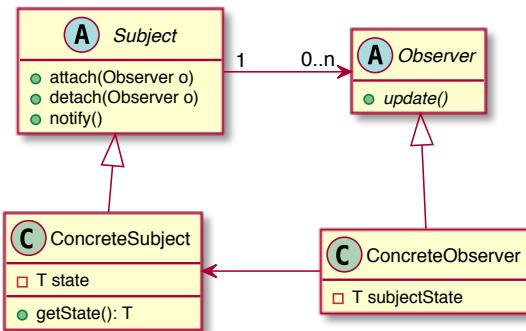
8

7_patrons_padawan - March 13, 2019

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

9

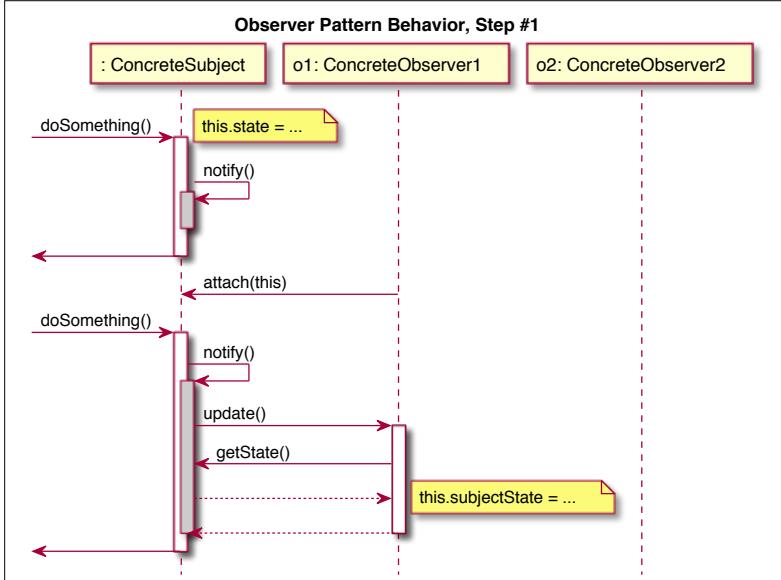


11

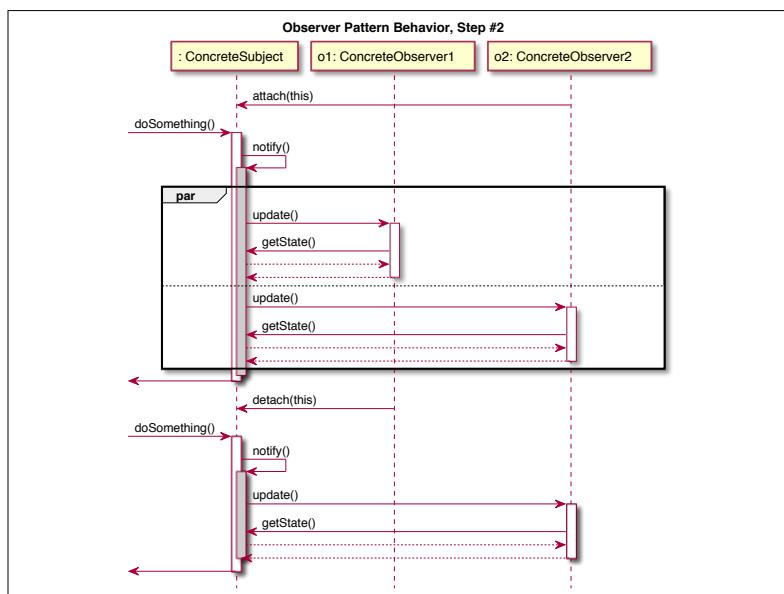
Intention

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.

10



12



13

```

class HexObserver extends Observer {
    public HexObserver(Subject subject) {
        this.subject = subject;
        this.subject.attach(this);
    }

    public void update() {
        System.out.print(" " + Integer.toHexString(subject.getState()));
    }
}

class OctObserver extends Observer { ... }

class BinObserver extends Observer { ... }

public class ObserverDemo {
    public static void main( String[] args ) {
        Subject sub = new Subject();
        new HexObserver(sub);
        new OctObserver(sub);
        new BinObserver(sub);
        Scanner scan = new Scanner(System.in);
        for (int i = 0; i < 5; i++) {
            System.out.print("\nEnter a number: ");
            sub.setState(scan.nextInt());
        }
    }
}

```

<https://sourcemaking.com>

15

```

abstract class Observer {
    protected Subject subject;
    public abstract void update();
}

class Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;

    public void attach(Observer o) { observers.add(o); }

    private void notify() {
        for (Observer observer : observers) {
            observer.update();
        }
    }

    public int getState() { return state; }

    public void setState(int value) {
        this.state = value;
        notify();
    }
}

```

<https://sourcemaking.com>

14

Conséquences

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

16

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

17

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 ...

19

Décorateur



18

Problème

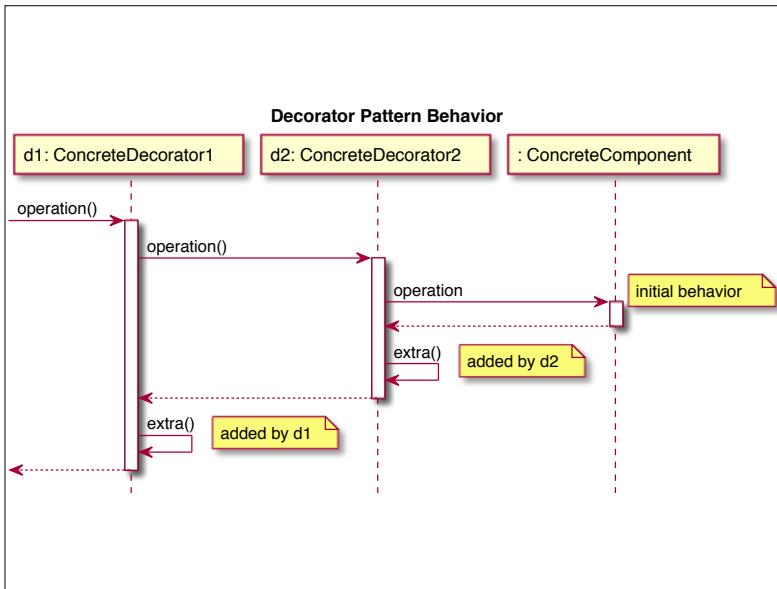
- Un objet doit pouvoir changer dynamiquement de comportement
- Ces comportements ne sont pas forcément connus à l'avance
- L'approche par héritage génère une explosion combinatoire
- Les changements sont au niveau de l'objet et pas de la classe

20

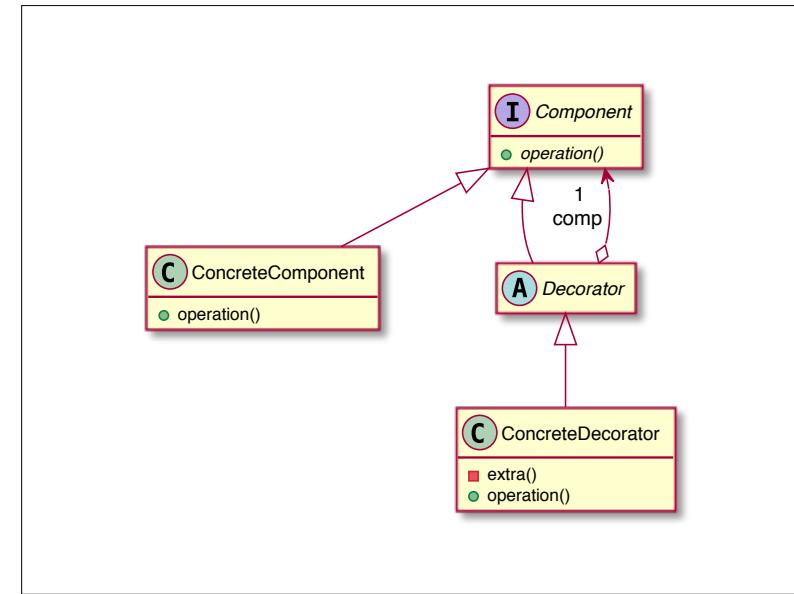
Intention

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

21



23



22

```

interface I {
    void doIt();
}

class A implements I {
    public void doIt() {
        System.out.print('A');
    }
}

abstract class D implements I {
    private I core;
    public D(I inner) { core = inner; }
    public void doIt() { core.doIt(); }
}

class X extends D {
    public X(I inner) { super(inner); }

    public void doIt() {
        super.doIt();
        doX();
    }

    private void doX() {
        System.out.print('X');
    }
}

class Y extends D {
    public Y(I inner) { ... }
    public void doIt() { ... }
    private void doY() { ... }
}

class Z extends D {
    public Z(I inner) { ... }
    public void doIt() { ... }
    private void doZ() { ... }
}
  
```

<https://sourcemaking.com>

24

```

public class DecoratorDemo {

    public static void main( String[] args ) {
        I[] array = {
            new X(new A()),
            new Y(new X(new A())),
            new Z(new Y(new X(new A()))));
    }

    for ( I anArray : array ) {
        anArray.doIt();
        System.out.print("  ");
    }
}

```

<https://sourcemaking.com>

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

26

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))

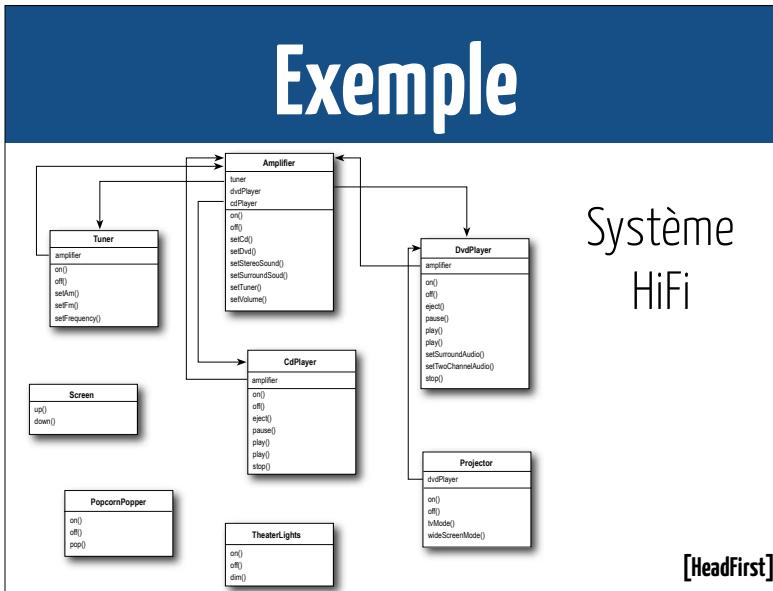
27

Façade



28

Exemple

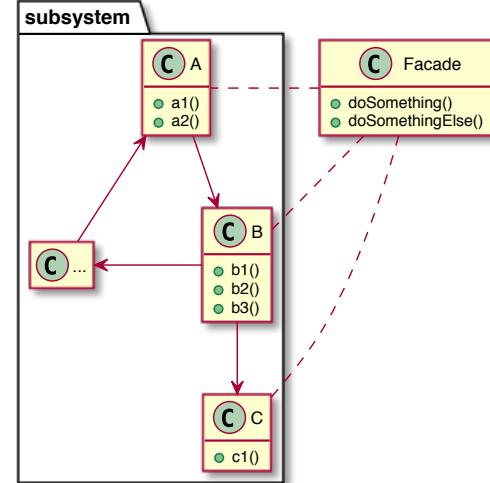


29

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

30



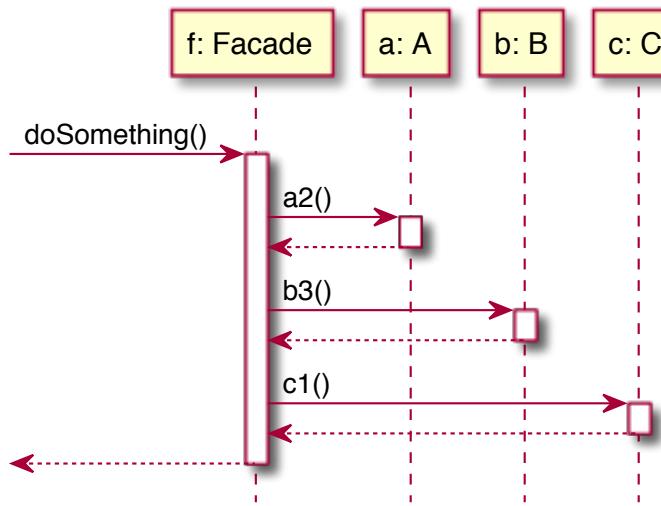
32

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

31

Facade Pattern Behavior



33

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

34

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

35

Fabrique



36

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

37

Intention

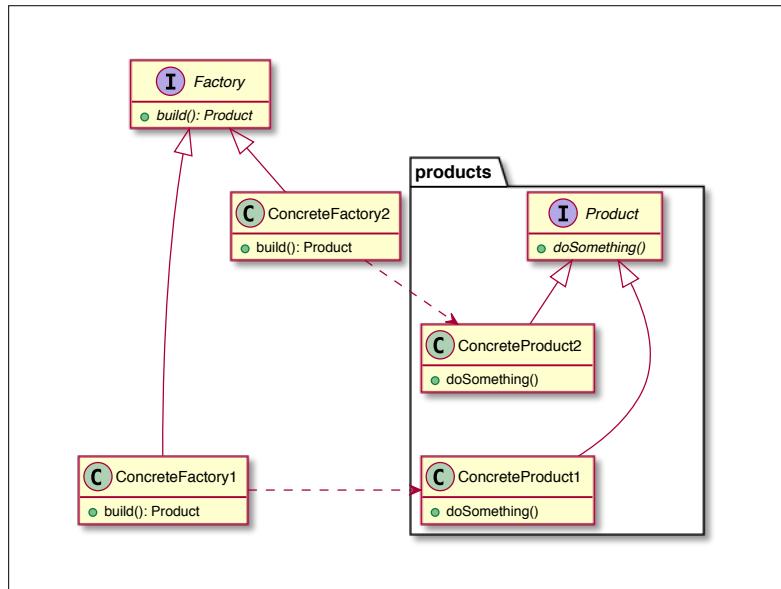
- Définir une interface pour la création d'un objet
- Laisser aux sous-classes le choix de l'instantiation
- Déléguer à une classe "qui sait faire" les instantiations complexes

39

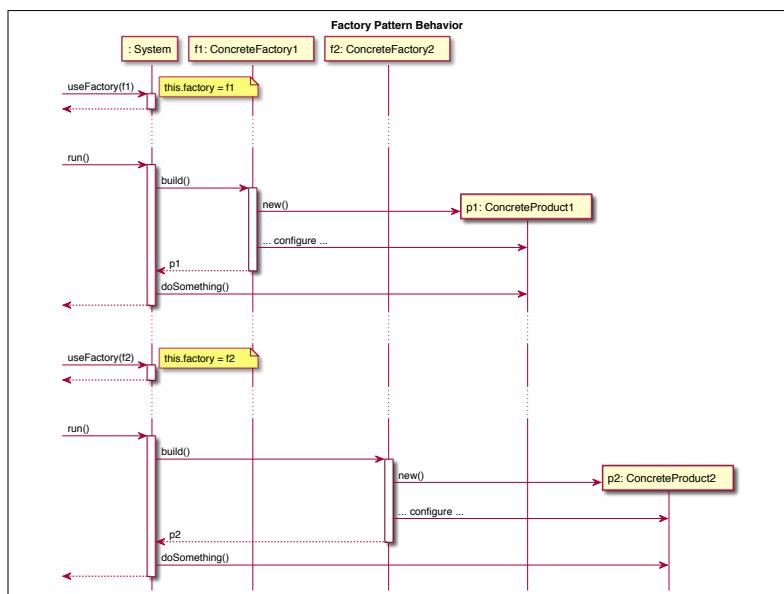
Problème

- 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

38



40



41

```

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

42

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

```

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;
    }
}
  
```

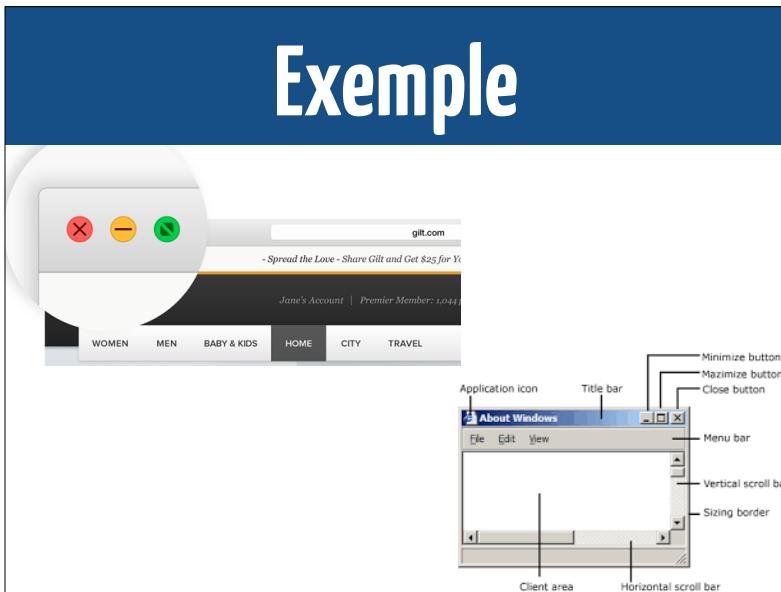
43

44

Où le trouver ?

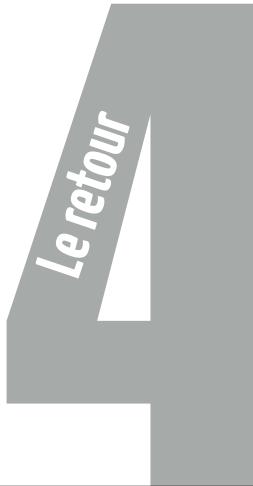
- Dans le JDK:
 - Les API utilisant getInstance() et valueOf()
 - (souvent attachée à des singletons)
- Dans les cadriels réseaux
 - Construire la connexion au système distant
- Dans les approches de modélisation / génération de code
 - Une interface pour plusieurs implémentation possible

45



47

Fabrique Abstraite



46

Problème

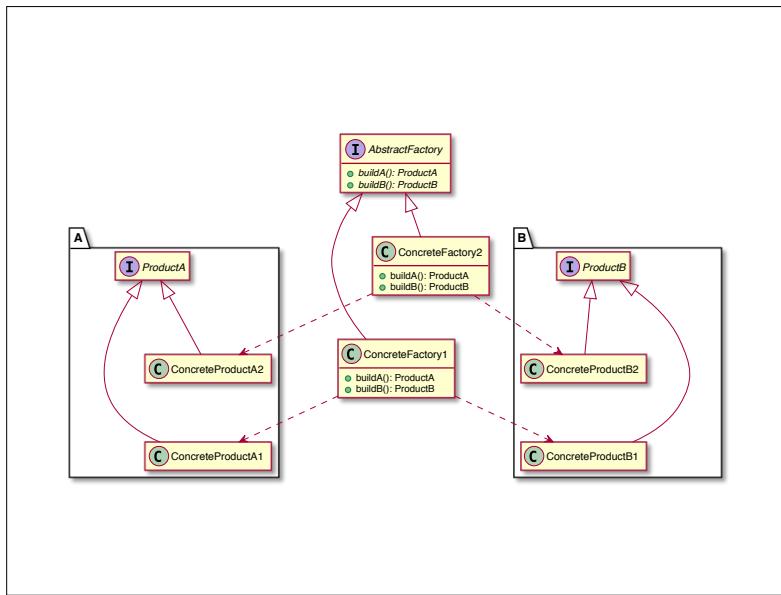
Comment garantir la consistance de familles de systèmes, proche mais différents ?

48

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

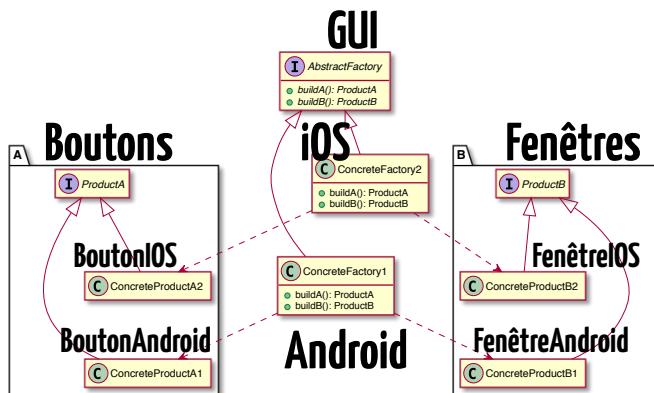
49



50

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 ...



51

52

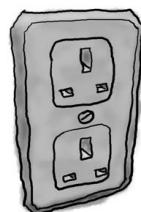
Où le trouver ?

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

53

Exemple

European Wall Outlet



Standard AC Plug



[HeadFirst]

55

Adaptateur



54

Problème

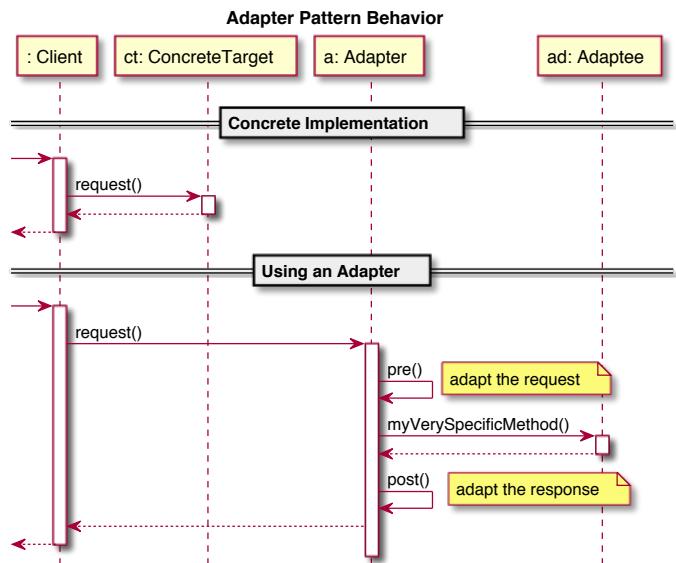
Comment intégrer du code défini selon une interface X alors que c'est une autre interface Y qui est attendu ?

56

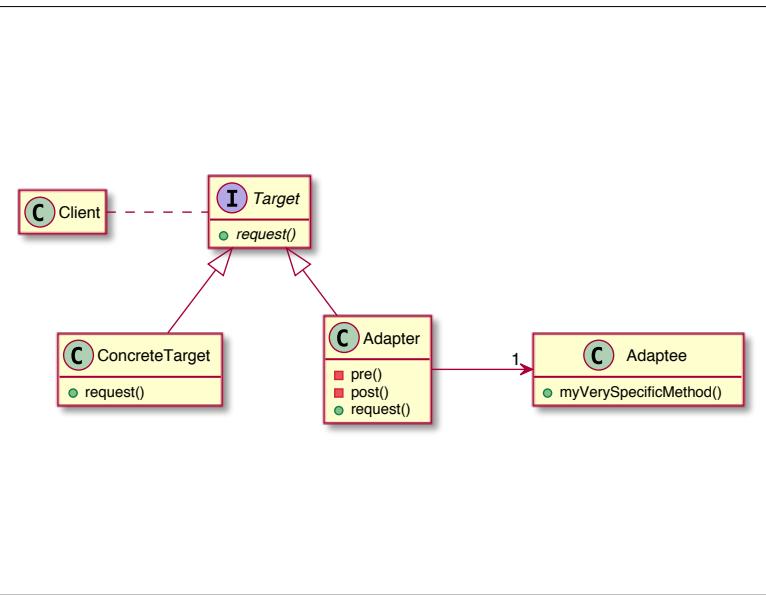
Intention

- 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

57



59



58

```
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/>

60

```

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/>

61

Où le trouver ?

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

63

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

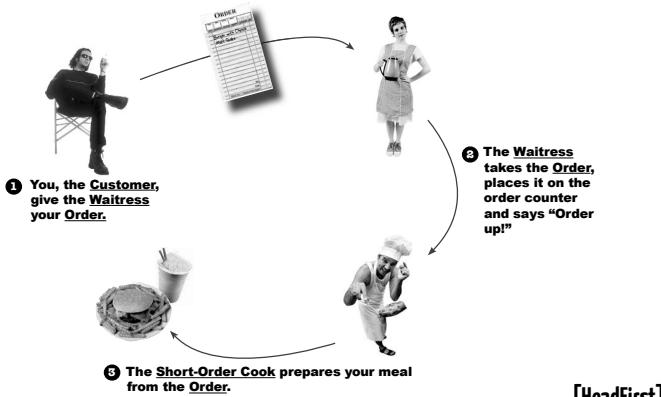
62

Commande



64

Exemple



65

Problème

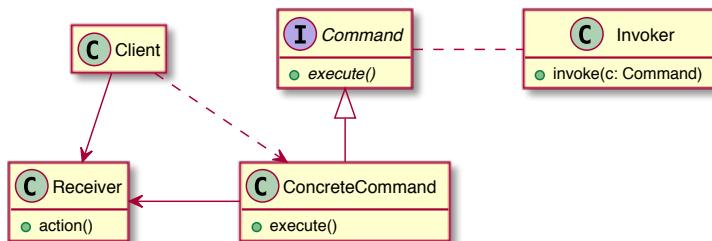
Comment réaliser un traitement sans forcément savoir qui va l'effectuer, ni de quoi il s'agit concrètement ?

66

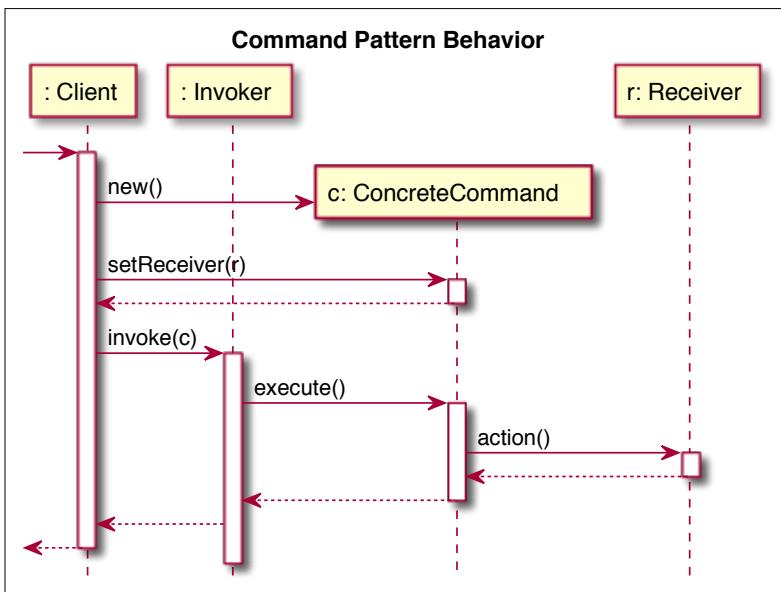
Intention

- 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

67



68



69

```

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>

70

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

```

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 LightsOnCommand(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>

71

72

Où le trouver ?

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