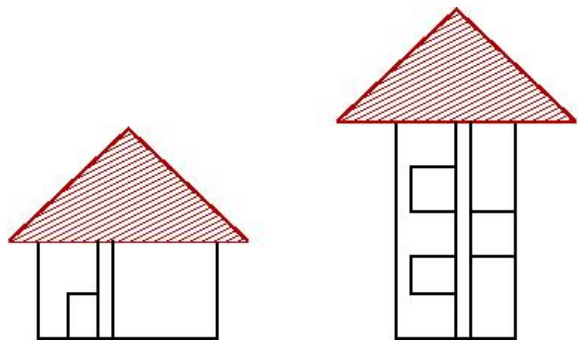
Introduction aux Design Patterns

C'est un architecte C. Alexander[1] qui a développé l'idée d'identifier des objets apportant :

> "Une solution précise à problème donné, dans un context précis"

https://java.developpez.com/tutoriels/programmation-orientee-objet/principes-avances/



- Définir un pattern, c'est fournir le *pourquoi* et le *comment* de chaque solution.

[1] "Timeless way of building", 1979.

Spécificité des “Design Patterns”

E. Gamma and R. Helmand and R. Johnson and J. Vlissides, 1995.
Introduction du concept de patterns dans la modélisation objet.

- Décrire aussi les relations avec les autres objets du modèle et les conséquences liées à leur utilisation.
- Les patterns modélisent des objets qui ***n’existent pas*** en tant qu’***entités dans le système cognitif humain***.

Bibliographie

- *The Design Patterns Java Companion* James W. Cooper
  `www.patterndepot.com/put/8/JavaPatterns.htm`
- *Object-Oriented Software Development Using Java* Xiaoping Jia, Ph.D
  `se.cs.depaul.edu/Java/chap10.html`

Qu'est ce qu'un Design Pattern ?

- Les Design Patterns (DP) sont des architectures de classes permettant d'apporter une solution à des problèmes fréquemment rencontrés lors des **phases d'analyse et de conception** d'applications.
- Ces solutions sont facilement adaptables (donc réutilisables), elles sont utilisables sans aucun risque dans la grande majorité des langages de programmation orientés objet.

Intérêt des Design Patterns

- **Avantages** : les Design Patterns ont une architecture facilement compréhensible et identifiables pour un programmeur (améliore la communication).
- **Objectifs** : *ne pas réinventer la roue*
- **Remarque** : Les Design Patterns ne sont donc pas vraiment une solution miracle pour les problèmes, mais ce sont plutôt des méthodes de résolutions.
  C'est comme une formule mathématique, c'est la solution mais encore faut-il l'appliquer au bon moment avec les bonnes variables.

Design Patterns : le catalogue

Ce travail a été initié dans la thèse de doctorat d'E. Gamma et finalisé par *"The Gang of Four"* dans un ouvrage qui fait référence.

- 23 design patterns classifiés en 3 groupes d'après leur **rôle**

Création	**Structuration**	**Def. de Comportement**
AbstractFactory	Adapter, Bridge	Interpreter, Command
Factory	Composite,	Chain of Responsability
Builder	Facade, Proxy	Iterator, Mediator,Template
Prototype	Flyweight	Memento, Observer
Singleton	Decorator	State, Strategy, Visitor

Création d'objets - Singleton

- **Principes général**
  - Un singleton, c'est un objet construit conformément à sa classe et dont on a la garantie qu'il n'existe qu'une et une seule instance en mémoire à un instant donné.
  - En cas d'accès concurrent lors de l'instanciation d'un singleton, il faut veiller à ce que cet aspect soit pris en compte par un mécanisme de verrous.
  - En général, on souhaite que le singleton ne s'initialise pas entièrement, mais seulement à son premier appel, afin d'économiser de la mémoire. On appelle cela le mécanisme *"lazy"*.
- Exemple **Implémentation en Java**
- Une classe n'est chargée que lors de son premier appel
- Le chargement d'une classe est *thread-safe*, c'est un mécanisme garanti par la hiérarchie de ClassLoaders de la JVM.

Création d'objets - Singleton

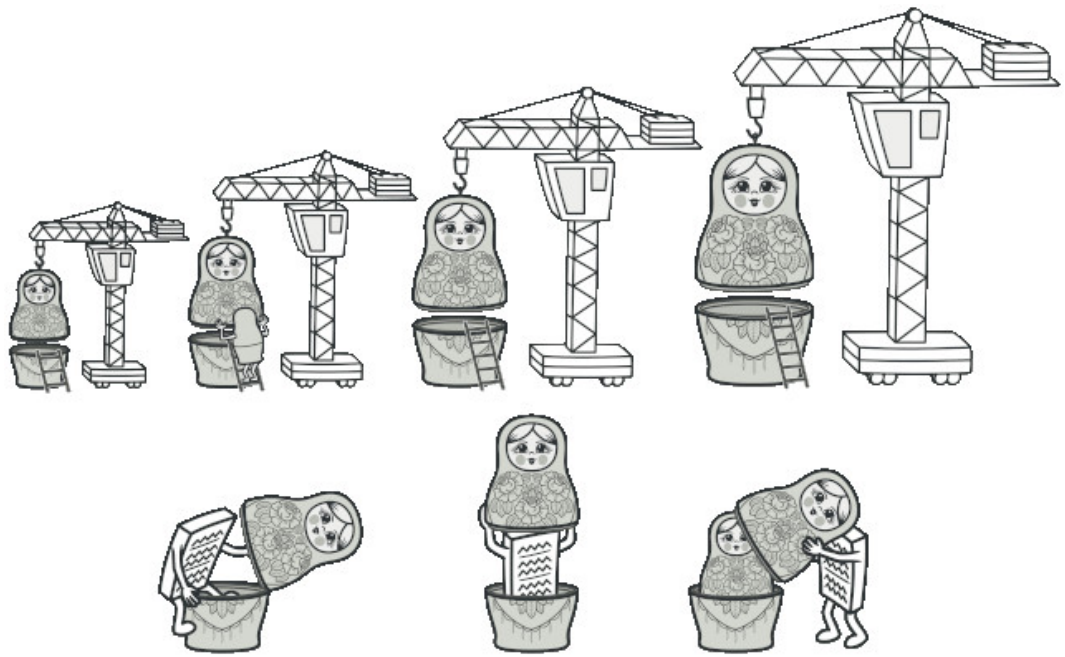
- Exemple **Implémentation en Java**
  - Un singleton est une classe qui ne produit qu'une seule instance, et cette dernière est accessible de partout.
  - Pour cela, il faut que la construction d'une nouvelle instance grâce à l'opérateur `new` soit interdite.
  - La solution est de rendre le constructeur de la classe privée.
  - Mais maintenant comment récupérer une instance de la classe ?

Code

```
public class ClassicSingleton {
  private static ClassicSingleton instance = null;

  protected ClassicSingleton() {
     // Exists only to defeat instantiation.
  }
  public static ClassicSingleton getInstance() {
     if(instance == null) {
        instance = new ClassicSingleton();
     }
     return instance;
  }
}
```

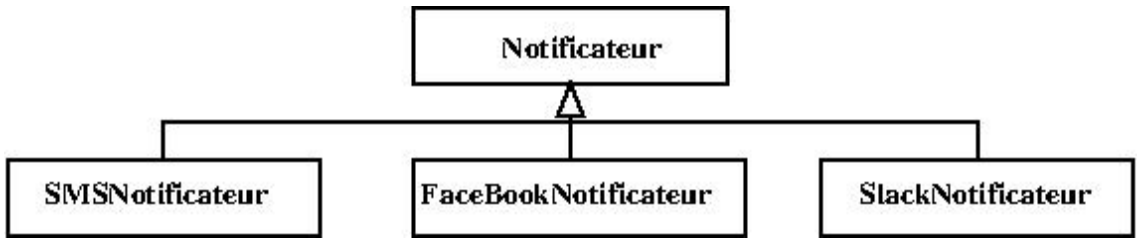
DP Structurel - Decorator

- **Intention** : affecter dynamiquement de nouveaux attributs ou comportements à des objets en les placant dans des *emballeurs* qui implémentent ces caractéristiques.



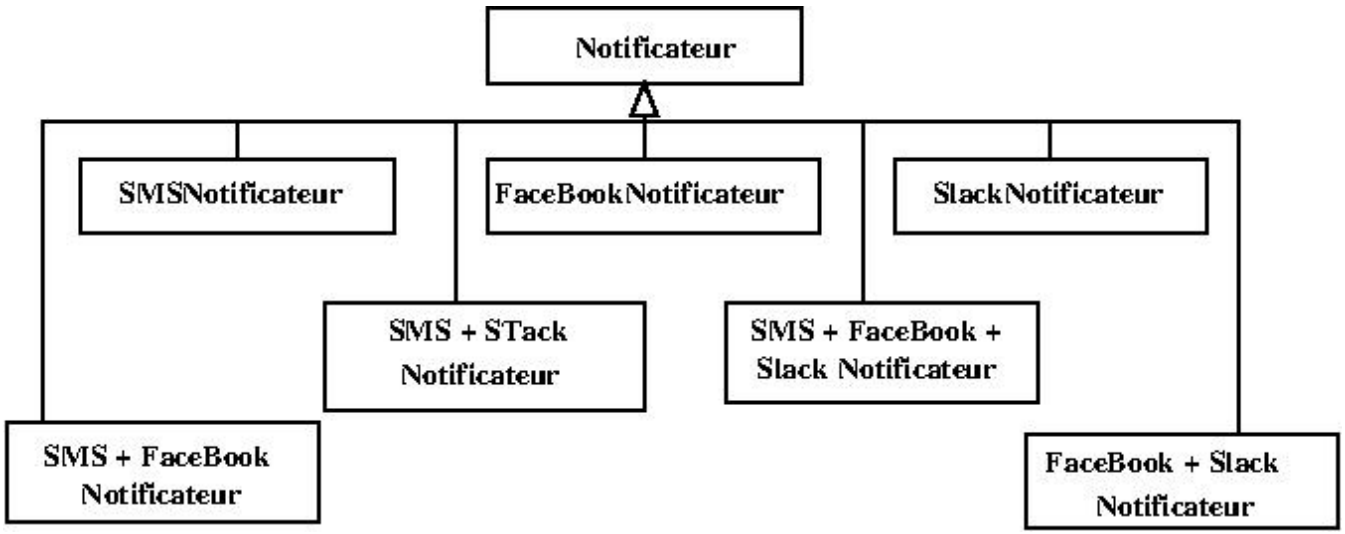
Decorator - Exemple

- Vous avez écrit une bibliothèque qui permet de notifier des évènements aux utilisateurs.
- Votre version 1 est très simple, votre classe `Notificateur` a une méthode `envoyer` qui prend un message en paramètre et envoie ce message à une liste de clients.
- Après quelques temps, vos clients demandent à être notifiés de différentes facons .... vous créez des sous classes !



Decorator - Exemple

- Désormais vos clients doivent instancier de nouvelles classes pour utiliser ces nouvelles capacités.
- Mais ... nouveau rebondissement, certains clients désirent être notifiés avec plusieurs applications .... nouvelles sous classes !!!



Decorator - Exemple

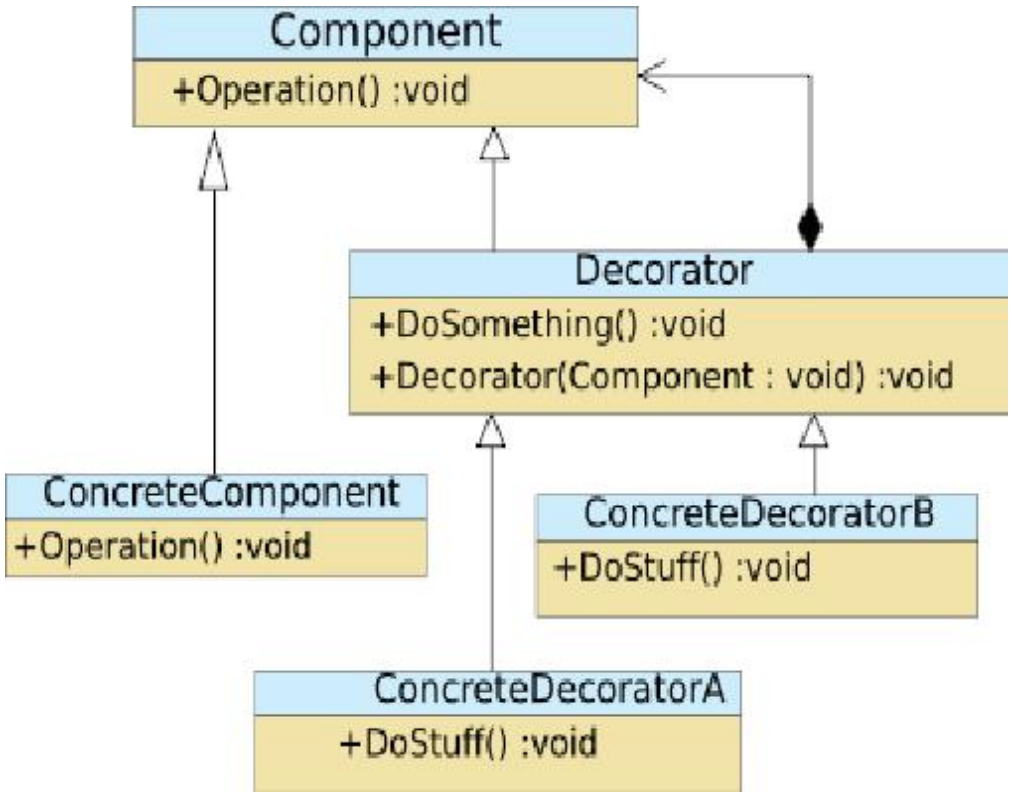
- La solution mise en place spontanément devient de plus en plus difficile à tenir.
- Inflation du code,
  - toujours plus de modifications à répercuter,
  - nécessité de recompiler les traitements précédents pour accèder aux nouvelles capacités ....

Decorator - Analyse du problème

- La solution par héritage a quelques inconvénients :
  - L'héritage est statique, vous ne pouvez pas modifier le comportement d'un objet à l'exécution.
  - Vous devez remplacer la totalité de l'objet par un autre, généré grâce à une sous-classe différente.
- Quid de l'agrégation/composition d'objets ?
  - Permet la délégation d'une partie du travail à accomplir à un autre objet.
  - Un objet peut utiliser le comportement de diverses classes.
  - Il devient facile de remplacer l'objet référencé par un autre, ce qui modifie le comportement du conteneur à l'exécution.

Decorator - Solution

- L'agrégation et la composition sont des principes clés de ce design pattern.



Exemple du WindowDecorator

```
interface Window {
    public void draw(); // draws the Window
    public String getDescription(); // returns a description of the Window
}

class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }
    public String getDescription() {
        return "simple window";
    }
}

abstract class WindowDecorator implements Window {
    protected Window decoratedWindow;
    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
}
```

Java Code

```
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        decoratedWindow.draw();
    }
    private void drawVerticalScrollBar() {
        // draw the vertical scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including vertical scrollbars";
    }
}

class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        decoratedWindow.draw();
    }
    private void drawHorizontalScrollBar() {
        // draw the horizontal scrollbar
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + ", including horizontal scrollbars";
    }
}
```

Java Code

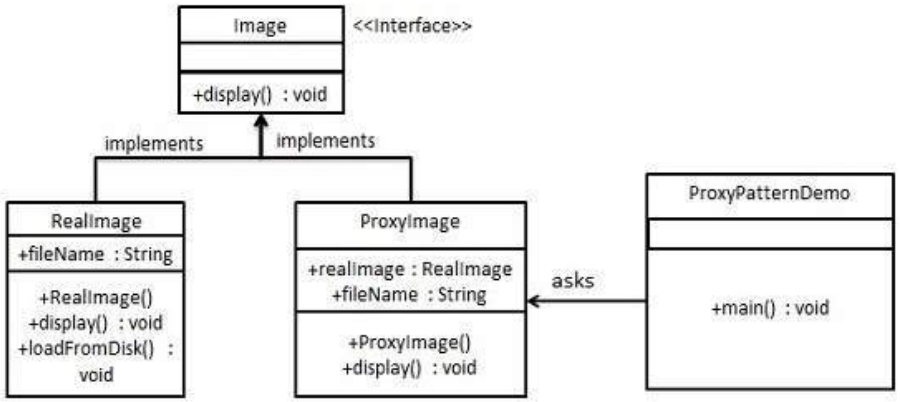
```
public class DecoratedWindowTest {
    public static void main(String[] args) {
        // create a decorated Window with horizontal and vertical scrollbars
        Window decoratedWindow = new HorizontalScrollBarDecorator (
                new VerticalScrollBarDecorator(new SimpleWindow()));

        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}
```

Decorator - Résumé

- Le décorateur de base possède un attribut pour référencer un objet emballé.
- Le composant déclare l'interface commune pour les décorateurs et les objets décorés.
- Les décorateurs concrets définissent les comportements qui seront ajoutés dynamiquement aux composants. Ils redéfinissent les méthodes du décorateur de base (super classe) et exécutent leur propre méthode.
- Les composants concrets contiennent les objets à décorer. Ils définissent un comportement par défaut qui pourra être modifié par les décorateurs.

DP Structurel - Proxy

- **Intention** : substituter un objet à un autre dans le but de controler cet objet. Simplifier l'accès à un objet complexe ou consommateur de ressources, de temps par exemple.
- Proxy crée un objet qui possède l'objet original et interface ses fonctionalités.
- Un proxy est réservé à une seule classe. Tout message envoyé au proxy est redirigé par lui vers la classe originale.



Proxy Code

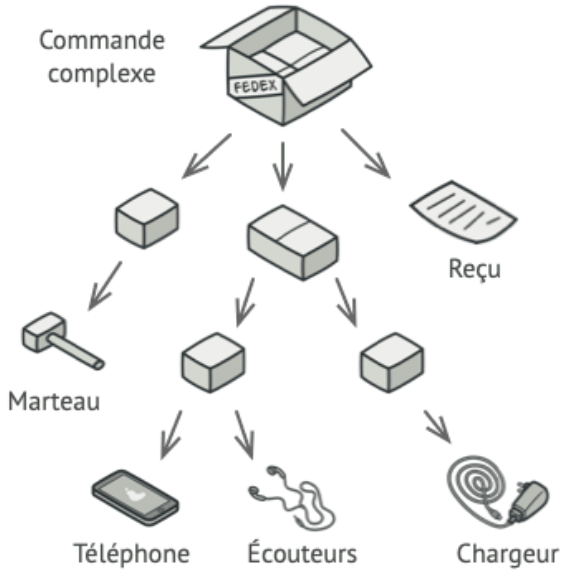
```
interface Image {
    public void displayImage();
}

class RealImage implements Image {
    private String filename;
    public RealImage(String filename) {
        this.filename = filename;
        loadImageFromDisk();
    }

    private void loadImageFromDisk() {
        System.out.println("Loading   "+filename);
    }

    public void displayImage()
    { System.out.println("Displaying "+filename); }
}

class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename) { this.filename = filename; }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); // load only on demand
        }
        image.displayImage();
    }
}
```

Java Code

```
class ProxyExample {
    public static void main(String[] args) {
        ArrayList<Image> images = new ArrayList<Image>();
        images.add( new ProxyImage("HiRes_10MB_Photo1") );
        images.add( new ProxyImage("HiRes_10MB_Photo2") );
        images.add( new ProxyImage("HiRes_10MB_Photo3") );

        images.get(0).displayImage();
        images.get(1).displayImage();
        images.get(0).displayImage();
    }
}
```

DP Structurel- Composite

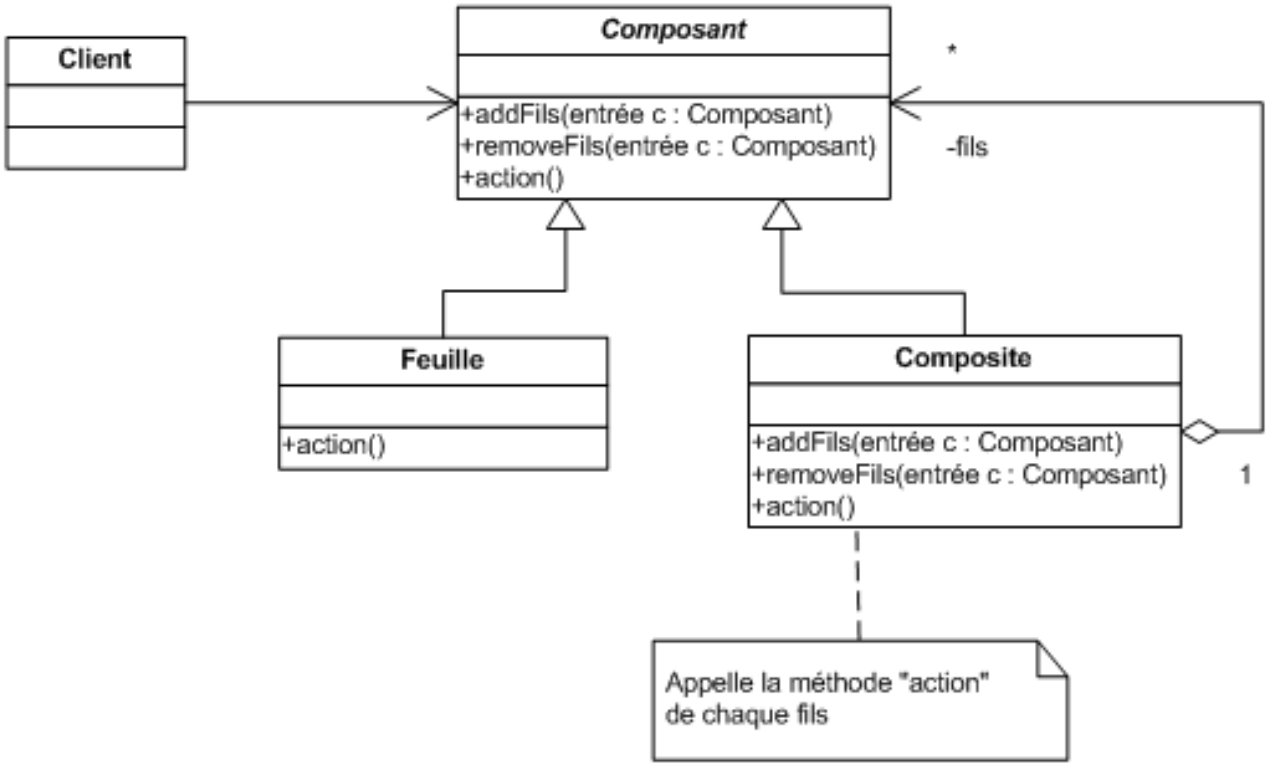
- **Intention** : permettre de facon transparente la composition/agencement d'objets. Cette composition étant vue comme un objet individuel par le client.
- Utilise des structures arborescentes et le principe de récursion.
- Exemple de problème à résoudre.



DP Structurel - Composite

- Une approche classique consistant à écrire une boucle qui parcourt chaque ensemble afin de savoir si il contient ou non un autre ensemble suppose de définir à l'avance (compilation) les possibles compositions.
- La solution consistant à combiner héritage et composition permet d'apporter une réponse générique à ce problème.
- On pourra alors traiter de la même facon un objet individuel ou une collection d'objets.

DP Structurel - Composite



Composite Code

```
/** "Component" */
interface Graphic {
   //Prints the graphic.
    public void print();
}
/** "Composite" */
class CompositeGraphic implements Graphic {

    //Collection of child graphics.
    private List<Graphic> childGraphics = new ArrayList<Graphic>();

    //Prints the graphic.
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }
    //Adds the graphic to the composition.
    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }

    //Removes the graphic from the composition.
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}
```

Java Code

```
/** "Leaf" */
class Ellipse implements Graphic {

    //Prints the graphic.
    public void print() {
        System.out.println("Ellipse");
    }
}
/** Client */
public class Program {
    public static void main(String[] args) {
        //Initialize four ellipses
        Ellipse ellipse1 = new Ellipse();
        Ellipse ellipse2 = new Ellipse();
        Ellipse ellipse3 = new Ellipse();
        Ellipse ellipse4 = new Ellipse();

        //Initialize three composite graphics
        CompositeGraphic graphic = new CompositeGraphic();
        CompositeGraphic graphic1 = new CompositeGraphic();
        CompositeGraphic graphic2 = new CompositeGraphic();

        //Composes the graphics
        graphic1.add(ellipse1);
        graphic1.add(ellipse2);
        graphic1.add(ellipse3);
        graphic2.add(ellipse4);
        graphic.add(graphic1);
        graphic.add(graphic2);

        //Prints the complete graphic (Four times the string "Ellipse").
        graphic.print();
    }
}
```

Composite Résumé

- Composite propose deux éléments de base qui partagent la même interface: des feuilles de l'arborescence (objet simple) ou des conteneurs complexes.
- Un conteneur peut être composé de feuilles ou de conteneurs complexes grâce au mécanisme d'héritage.
- Votre structure récursive sera composés d'objets imbriqués qui ressemble à un arbre.
- Utiliser ce pattern lorsque vous voulez permettre au client d'interagir de facon uniforme avec des éléments simples ou complexes.
- Avantages : profite au maximum des capacités du polymorphisme, vous pouvez introduire de nouveaux types d'objets sans modifier l'existant.
- Inconvénients : difficile de restreindre le type des composants.

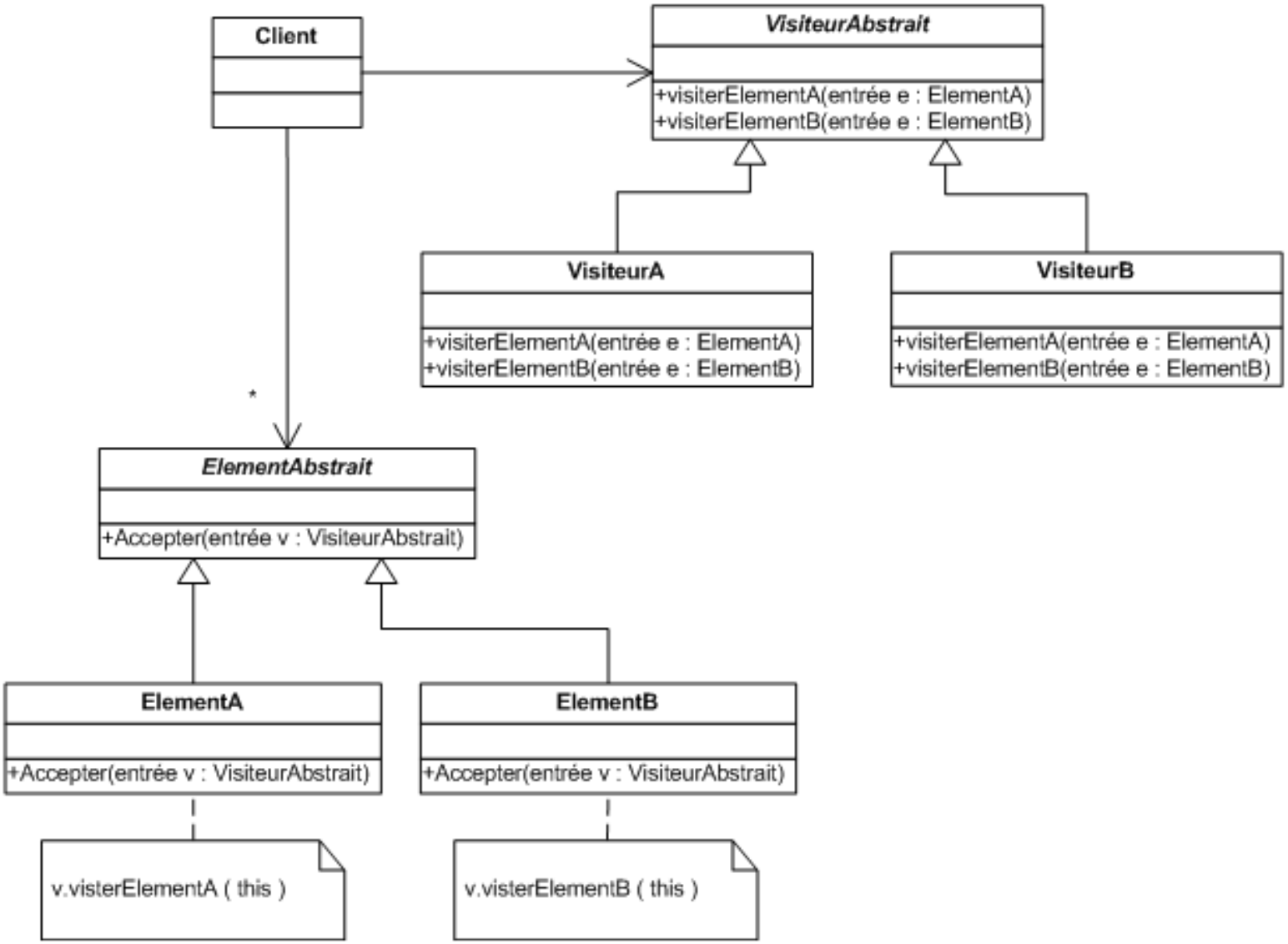
DP Comportemental - Visitor

- **Intention** : séparer un algorithme de la structure de données.
- Permet d'ajouter des traitements sans modifier la structure.
- Est aussi utilisé pour mettre en oeuvre le principe ouvert/fermé.
- Le principe ouvert fermé repose sur le fait que chaque unité logiciel (modules, classes, méthodes) est à la fois ouverte aux extensions et fermée aux modifications.
- **Utilité** : le traitement d'un objet pour différer en fonction des traitements précédents grâce à l'état du visiteur.
- **Exemple** : un visiteur affichant une structure arborescente peut présenter les noeuds de l'arbre en utilisant une indentation dont le niveau est stockée comme valeur de l'état du visiteur.

DP Comportemental - Visitor

- **Mise en oeuvre** :
  - Le comportement concerné est placé dans une classe séparée qu'on appelle `Visitor`.
  - La classe `Visitor` possèdera une version du comportement pour chaque classe susceptible d'utiliser ce visiteur.
  - La classe à visiter sera recue en paramètre, le polymorphisme permettra d'exécuter la bonne version de la méthode.

DP Comportemental - Visitor



Affichage des éléments d'une voiture

```
interface CarElementVisitor{
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visitCar(Car car);
}

interface CarElement{
    void accept(CarElementVisitor visitor);
    // Méthode à définir par les classes implémentant CarElements
}

class Wheel implements CarElement{
    private String name;

    Wheel(String name){
        this.name = name;
    }

    String getName(){
        return this.name;
    }

    public void accept(CarElementVisitor visitor){
        visitor.visit(this);
    }
}
```

Affichage des éléments d'une voiture

```
class Engine implements CarElement{
    public void accept(CarElementVisitor visitor){
        visitor.visit(this);
    }
}

class Body implements CarElement{
    public void accept(CarElementVisitor visitor){
        visitor.visit(this);
    }
}

class Car{
    CarElement[] elements;
    public CarElement[] getElements(){
        return elements.clone(); // Retourne une copie du tableau de références.
    }

    public Car(){
        this.elements = new CarElement[]{
                new Wheel("front left"),
                new Wheel("front right"),
                new Wheel("back left"),
                new Wheel("back right"),
                new Body(),
                new Engine()
            };
    }
}
```

Affichage des éléments d'une voiture

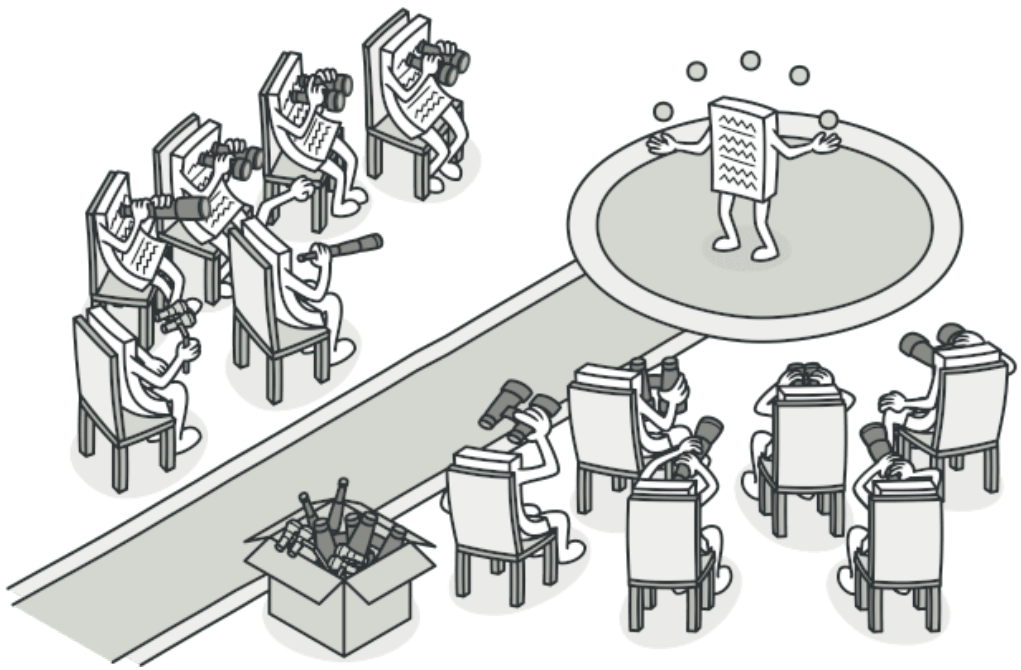
```
class CarElementPrintVisitor implements CarElementVisitor{
    public void visit(Wheel wheel){
        System.out.println("Visiting "+ wheel.getName() + " wheel");
    }

    public void visit(Engine engine){
        System.out.println("Visiting engine");
    }

    public void visit(Body body){
        System.out.println("Visiting body");
    }

    public void visitCar(Car car){
        System.out.println("\nVisiting car");
        for(CarElement element : car.getElements())
        {
            element.accept(this);
        }
        System.out.println("Visited car");
    }
}
```

Affichage des éléments d'une voiture

```
class CarElementDoVisitor implements CarElementVisitor{
    public void visit(Wheel wheel){
        System.out.println("Kicking my "+ wheel.getName());
    }

    public void visit(Engine engine){
        System.out.println("Starting my engine");
    }

    public void visit(Body body){
        System.out.println("Moving my body");
    }

    public void visitCar(Car car){
        System.out.println("\nStarting my car");
        for(CarElement carElement : car.getElements()){
            carElement.accept(this);
        }
        System.out.println("Started car");
    }
}

public class VisitorDemo{
    static public void main(String[] args)
    {
        Car car = new Car();

        CarElementVisitor printVisitor = new CarElementPrintVisitor();
        CarElementVisitor doVisitor = new CarElementDoVisitor();

        printVisitor.visitCar(car);
        doVisitor.visitCar(car);
    }
}
```

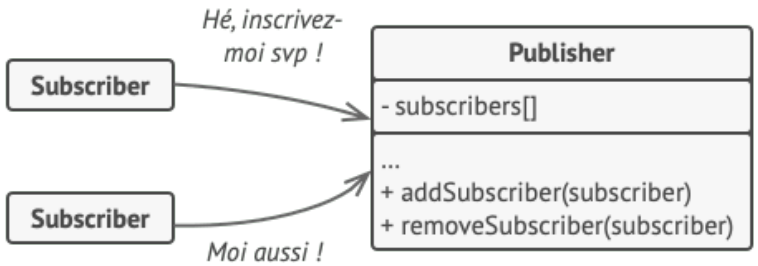
DP Structurel - Observer

- **Intention** : permet de créer des dépendances entre objets (qui ne sont pas dans la même hiérarchie) afin que lorsque l'un d'eux change d'état, l'autre soit informé.



DP Structurel - Observer

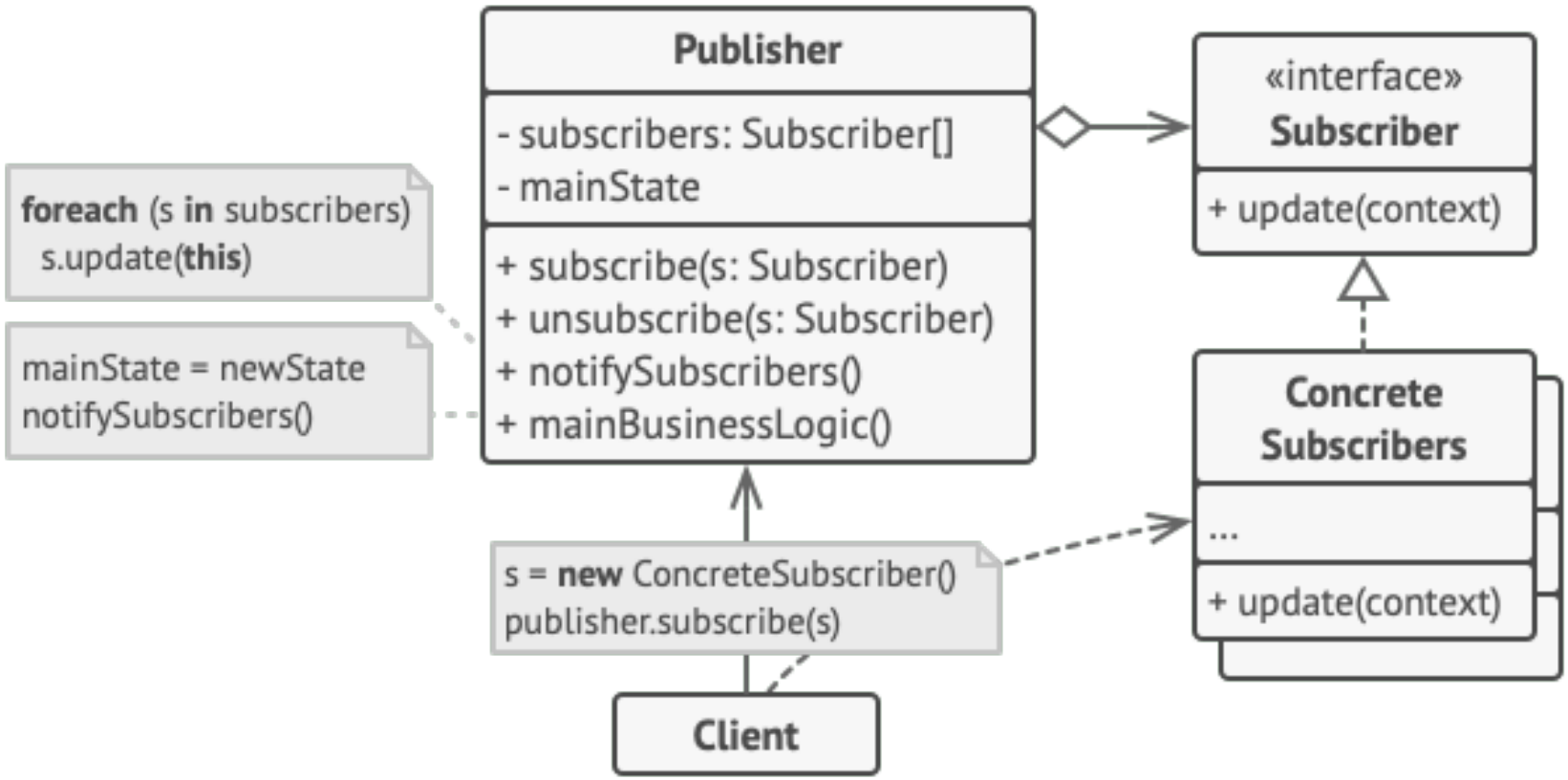
- **Utilisation** : quand un objet doit signaler quelque chose sans avoir de connaissance sur l'objet qu'il notifie.
- **Exemple** : Un client attend un produit particulier et se rend au magasin pour le chercher. Ce produit est absent. Le client va-t-il se rendre tous les jours à la boutique pour demander si le produit est disponible ?
  - On (le publisher) va préférer faire prévenir les clients de l'arrivée du produit.
  - Mais avertir tous les clients est également contre-productif.
  - On va avertir seulement ceux qui ont déclaré attendre le produit(le sujet). Ces clients seront les *subscribers*.



DP Structurel - Observer

- L'interface `Subscriber` déclare les méthodes de notification. En général, la méthode `update`.
- Les `ConcreteSubscribers` exécute les méthodes en réponse aux notifications envoyées par le *publisher*.
- Le client crée des objets *publisher* et *subscriber* séparemment et inscrit les *subscribers* aux mises à jour des *publishers*.
- Le `Publisher` envoie les événements aux autres objets. Il possède la méthode qui permet d'inscrire ou désinscrire les nouveaux *subscribers*.
- Quand un événement survient, le *publisher* parcourt la liste des objets inscrits et appelle la notification.

DP Comportemental - Observer



Java Code

```
public interface WeatherObserver {

  void update(WeatherType currentWeather);
}

@Slf4j
public class Orcs implements WeatherObserver {

  @Override
  public void update(WeatherType currentWeather) {
    LOGGER.info("The orcs are facing " + currentWeather.getDescription() + " weather now");
  }
}

@Slf4j
public class Hobbits implements WeatherObserver {

  @Override
  public void update(WeatherType currentWeather) {
    switch (currentWeather) {
      LOGGER.info("The hobbits are facing " + currentWeather.getDescription() + " weather now");
    }
  }
}
```

Java Code

```
public class Weather {

  private WeatherType currentWeather;
  private final List<WeatherObserver> observers;

  public Weather() {
    observers = new ArrayList<>();
    currentWeather = WeatherType.SUNNY;
  }

  public void addObserver(WeatherObserver obs) {
    observers.add(obs);
  }

  public void removeObserver(WeatherObserver obs) {
    observers.remove(obs);
  }
 /**
   * Makes time pass for weather.
   */
  public void timePasses() {
    var enumValues = WeatherType.values();
    currentWeather = enumValues[(currentWeather.ordinal() + 1) % enumValues.length];
    LOGGER.info("The weather changed to {}.", currentWeather);
    notifyObservers();
  }

  private void notifyObservers() {
    for (var obs : observers) {
      obs.update(currentWeather);
    }
  }
}
```

Java Code

```
class Apply{
 public static void main (String []arv)
  {
    var weather = new Weather();
    weather.addObserver(new Orcs());
    weather.addObserver(new Hobbits());
    weather.timePasses();
    weather.timePasses();
    weather.timePasses();
    weather.timePasses();
  }

program output :
 The weather changed to rainy.
The orcs are facing rainy weather now
The hobbits are facing rainy weather now
The weather changed to windy.
The orcs are facing windy weather now
The hobbits are facing windy weather now
The weather changed to cold.
The orcs are facing cold weather now
The hobbits are facing cold weather now
The weather changed to sunny.
The orcs are facing sunny weather now
The hobbits are facing sunny weather now
```

Observer - Résumé

- **Avantages** :
  - s'inscrit dans le principe ouvert/fermé. On peut ajouter de nouvelles classes de *subscriber* sans modifier le code du *publisher*.
  - Des liens entre les objets peuvent être créés à l'exécution.
- **Inconvénients** : difficulté à maitriser l'ordre d'avertissement des *subscribers*.
- **Remarque** : les classes `Observer` et `Observable` sont implémentées en Java. On peut étendre `Observable` pour implémenter des envois de notifications.

Code

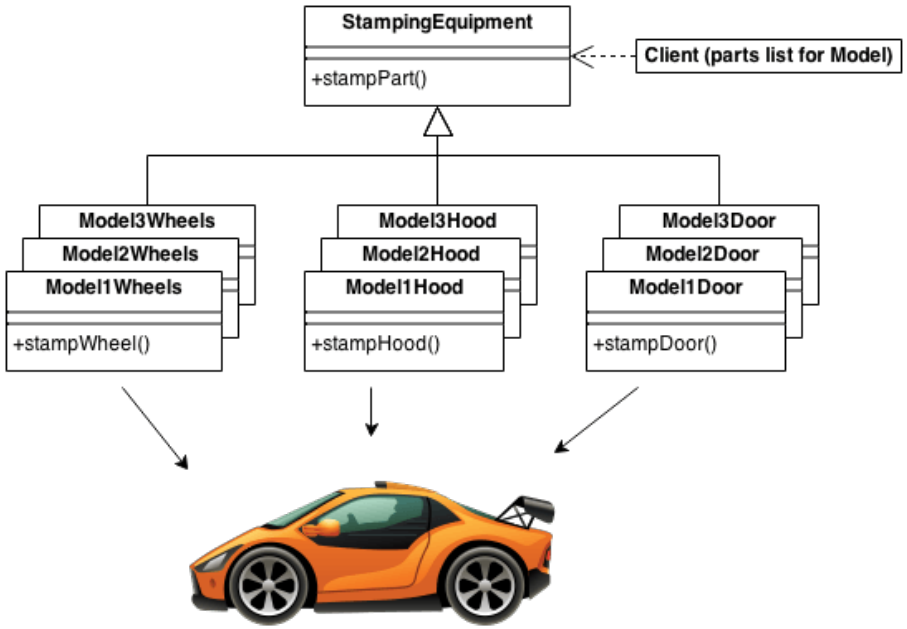
```
class Signal extends Observable
{
    void setData(byte[] lbData)
    {
        setChanged(); // Positionne son indicateur de changement
        notifyObservers(); // (1) notification
    }
}


class JPanelSignal extends JPanel implements Observer
{
    void init(Signal lSigAObserver)
    {
        lSigAObserver.addObserver(this); // (2) ajout d'observeur
    }

    void update(Observable observable, Object objectConcerne)
    {
        repaint();  // (3) traitement de l'observation
    }
}
```

DP Structurel - AbstractFactory

- Design Pattern lié à `Factory`. Encapsule un groupe de `Factory`.
- Crée une famille d'objets apparentés sans préciser la classe concrète.
- Crée une hiérarchie de classe - les classes qui implémentent la fabrique abstraite suivent le plus souvent le pattern `Factory` mais le pattern `Prototype` peut aussi être utilisé.
- `Abstract Factory` est une alternative du pattern `Facade` ... ou inversement :)

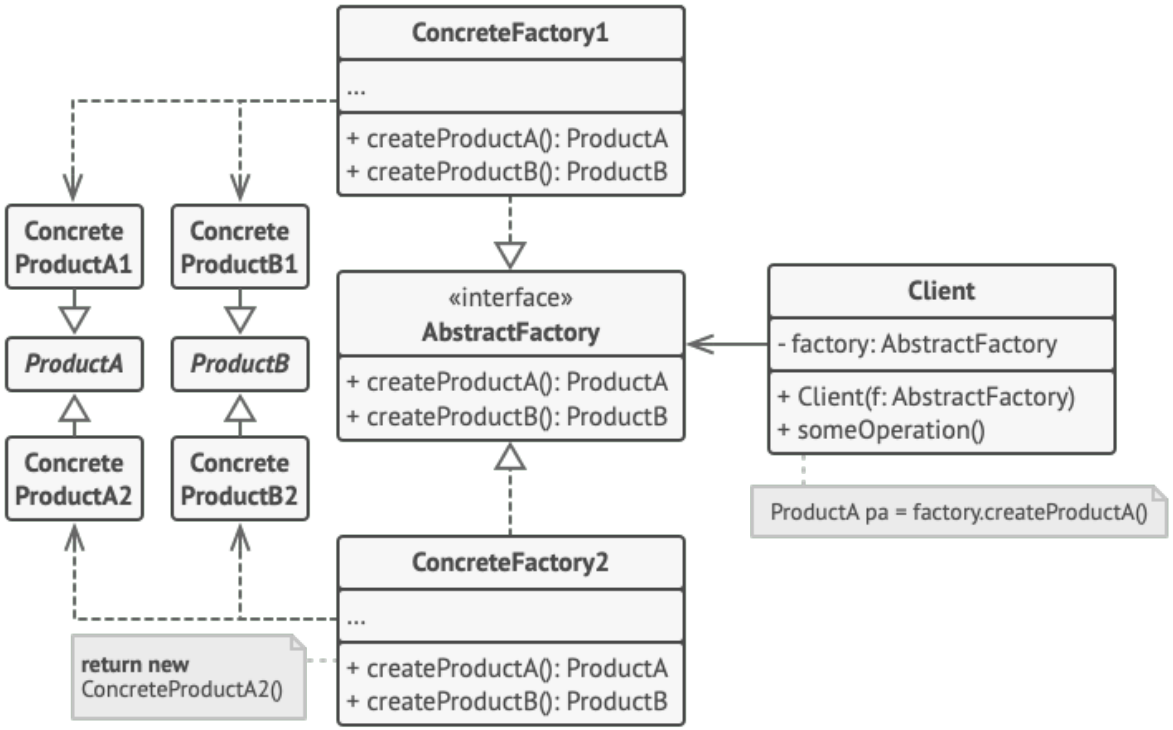
AbstractFactory Exemple

- Créer les différentes parties d'un véhicule.
- Abstract Factory est en charge de créer automatiquement ces différentes parties : porte droite ou gauche, aile droite ou gauche ... pour différents modèles.



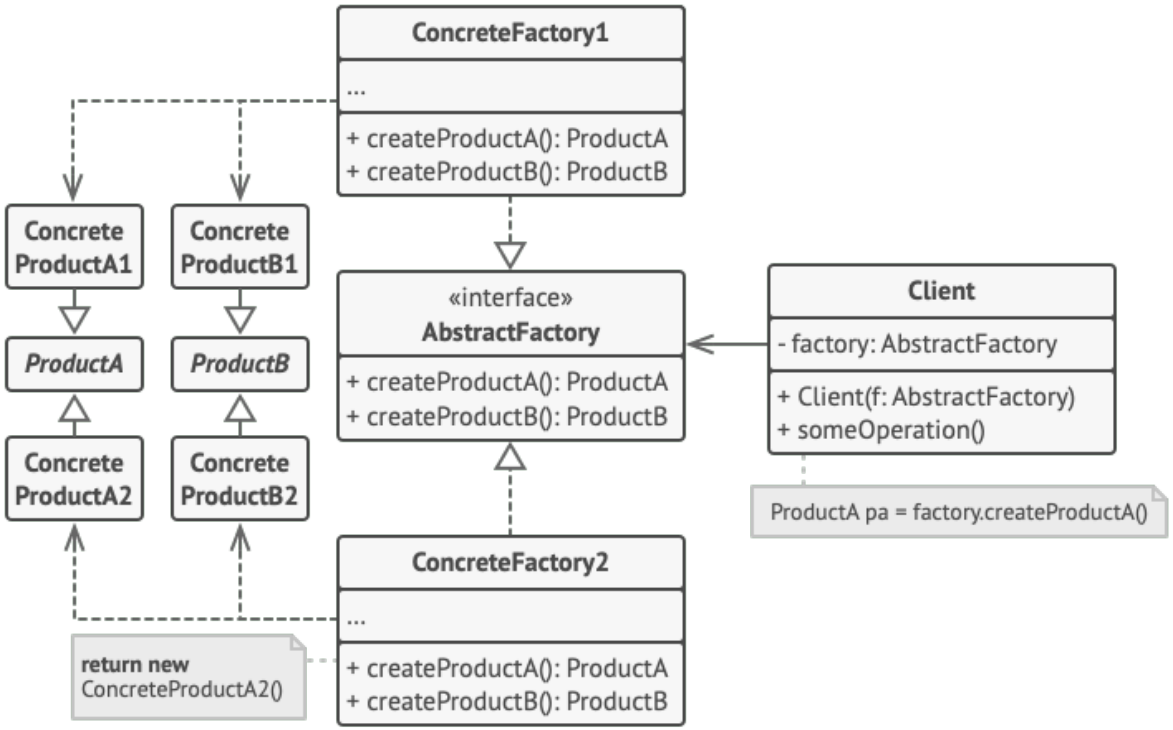
AbstractFactory Structure

- Les produits abstraits déclarent une interface pour un ensemble d'objets distincts mais apparenté qui forment une famille de produits.



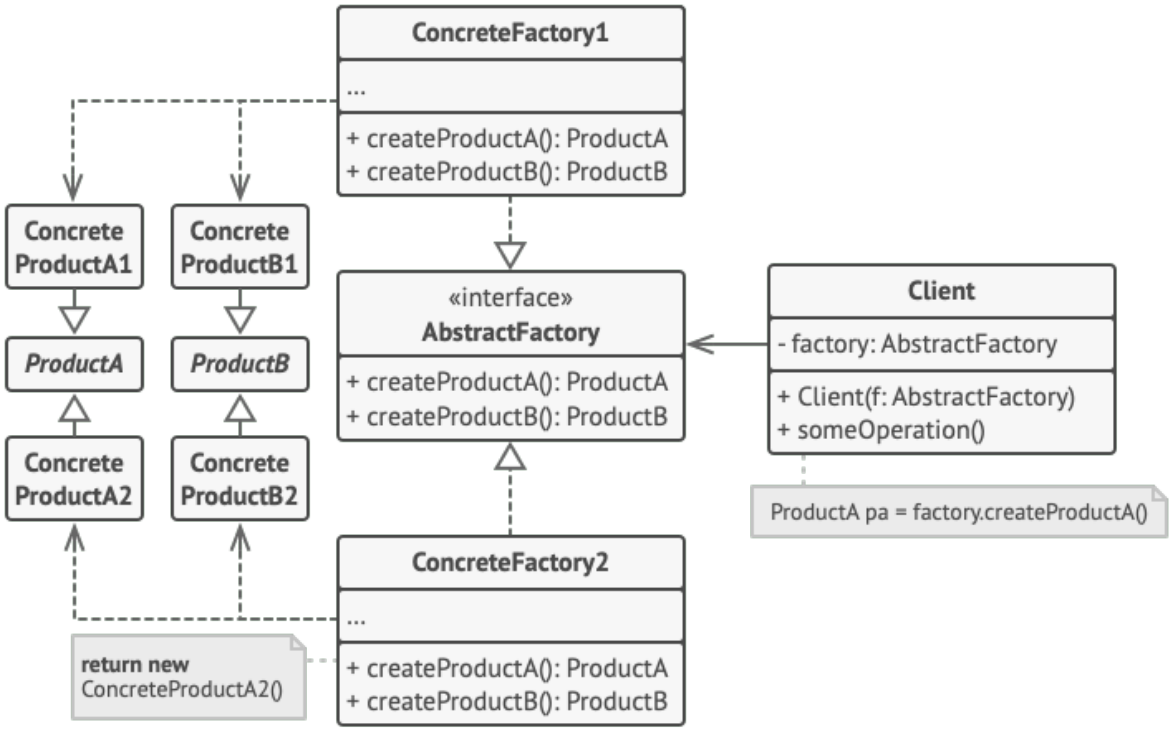
AbstractFactory Structure

- Les produits concrets sont des implémentations des produits abstraits groupés par variantes. Chaque produit abstrait doit être implémenté dans toutes les variantes.



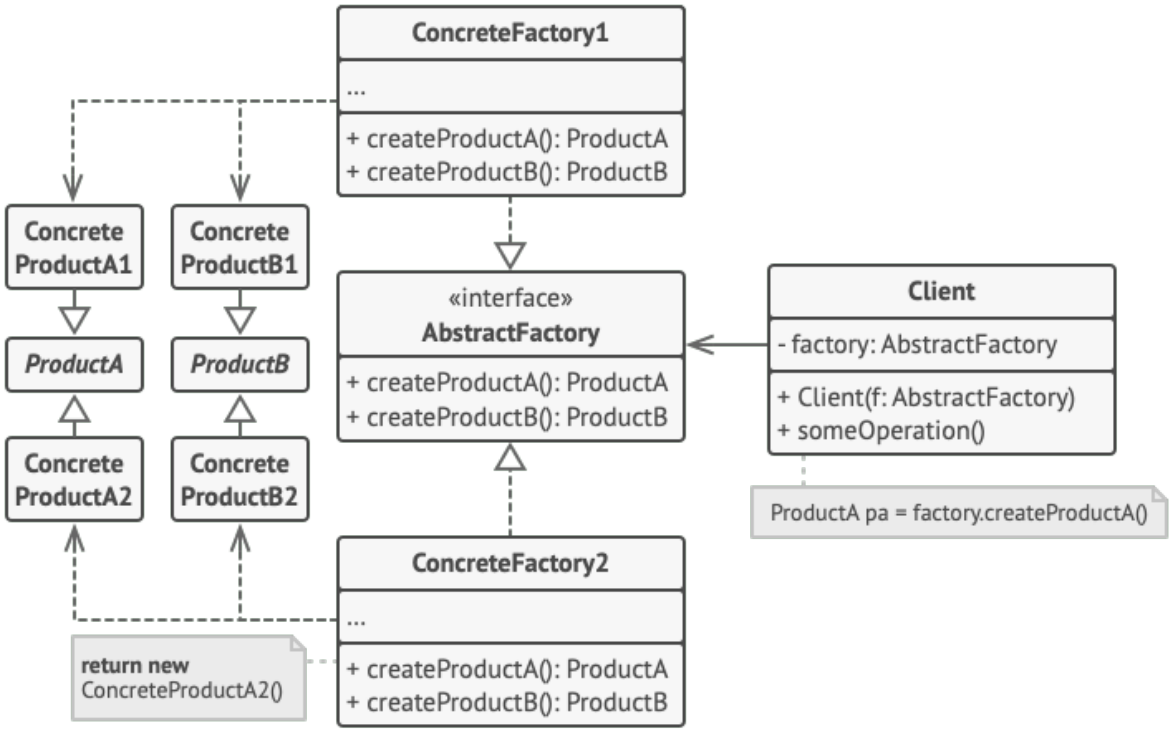
AbstractFactory Structure

- L'interface `Fabrique Abstraite` déclare un ensemble de méthodes pour créer chacun des produits abstraits.



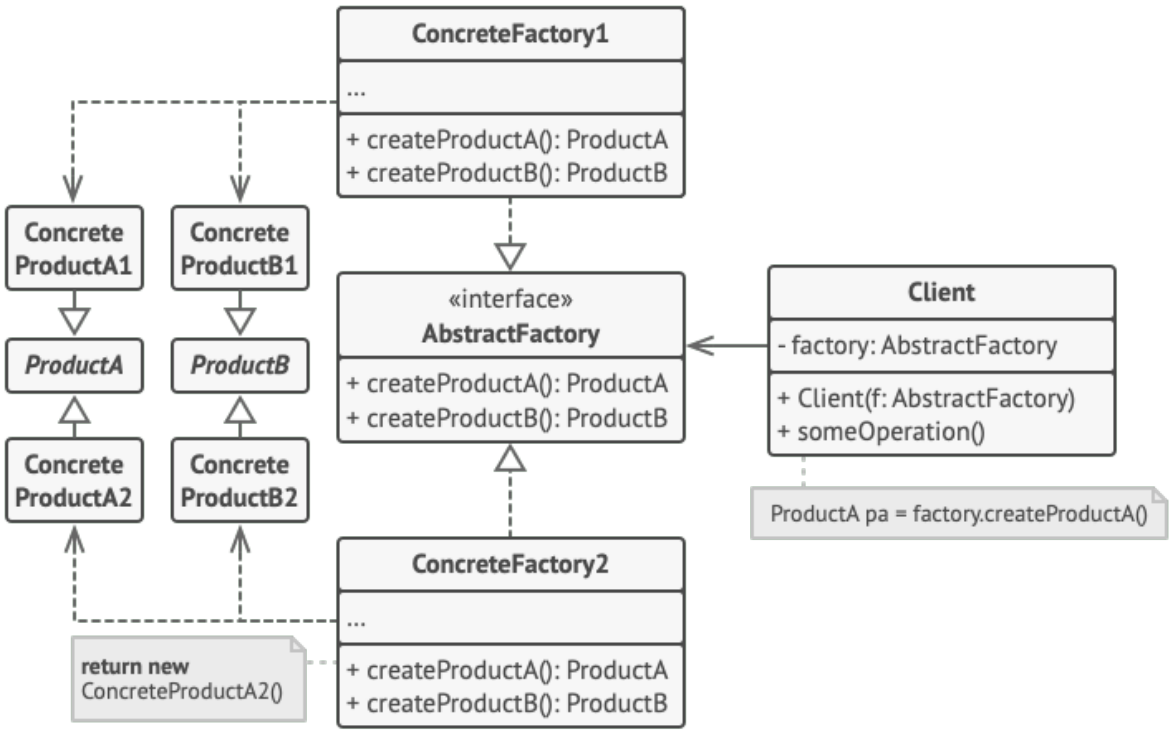
AbstractFactory Structure

- Les fabriques concrétes implémentent les opérateurs de création d'objets de la fabrique abstraite. Chaque fabrique concrète correspond à une variante spécifique de produits et ne crée que ces variantes.



AbstractFactory Structure

- La valeur de retour des fabriques concrètes est un produit abstrait. De cette facon le code client est isolé de la variante du produit obtenu. Le client peut travailler avec n'importe quelle variante de fabrique et de produit dans la mesure où il n'interagit qu'avec les interfaces abstraites.



Java Code

```
abstract class GUIFactory {
    public static GUIFactory getFactory() {
        int sys = readFromConfigFile("OS_TYPE");
        if (sys == 0) {
            return new WinFactory();
        } else {
            return new OSXFactory();
        }
    }
    public abstract Button createButton();
}

class WinFactory extends GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}
class OSXFactory extends GUIFactory {
    public Button createButton() {
        return new OSXButton();
    }
}
```

Java Code

```
abstract class Button {
    public abstract void paint();
}
class WinButton extends Button {
    public void paint() {
        System.out.println("I'm a WinButton: ");
    }
}
class OSXButton extends Button {
    public void paint() {
        System.out.println("I'm an OSXButton: ");
    }
}
public class Application {
    public static void main(String[] args) {
        GUIFactory factory = GUIFactory.getFactory();
        Button button = factory.createButton();
        button.paint();
    }
}
```

DP Structurel - AbstractFactory

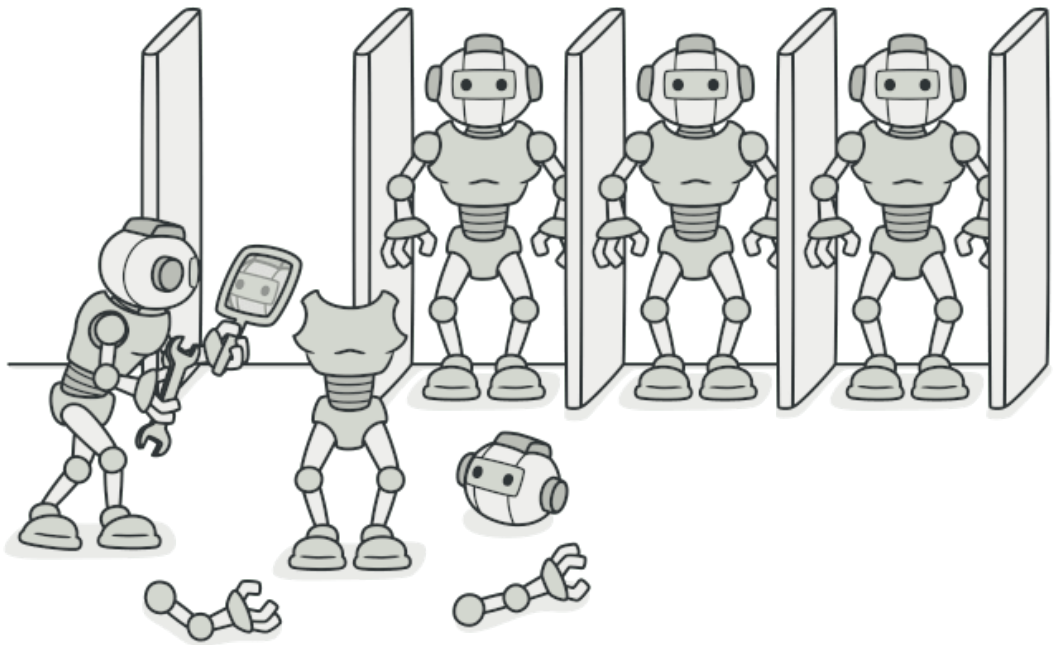
- Liens avec d'autres patterns :
  - Le pattern `Factory` est utilisé en début de conception, plus facile pour caractériser les premiers objets, il évolue ensuite vers `Abstract Factory` pour dégager les concepts.
  - `Abstract Factory` et `Prototype` peuvent être considérés comme équivalents et/ou complémentaires.
  - Le pattern `Builder` est souvent associé pour créer les objets suivant un protocol complexe. Ce pattern a la capacité de réaliser une conception incrémentale des objets.
  - Il existe un pattern `Factory Method` qui n'appartient pas à la liste initiale des Design Patterns, qui est parfois utilisé pour commencer le design si une grande flexibilité est nécessaire.

DP Structurel - Factory Method

- Recouvre très largement le pattern `Abstract Factory`.
- Définit une méthode de classe `static` qui déclare la fabrication d'un objet. La valeur de retour peut être un objet du type d'une sous-classe, contrairement à la méthode `new`.
- Ce pattern est aussi appelé `Template Method`.

DP de Création - Prototype

- **Intention** : créer des copies exactes d'un objet sans connaitre sont types. Le code est donc indépendant de la classe.
- Peut être vu comme une extension du constructeur par copie (C++). Utilise le clonage d'objets.

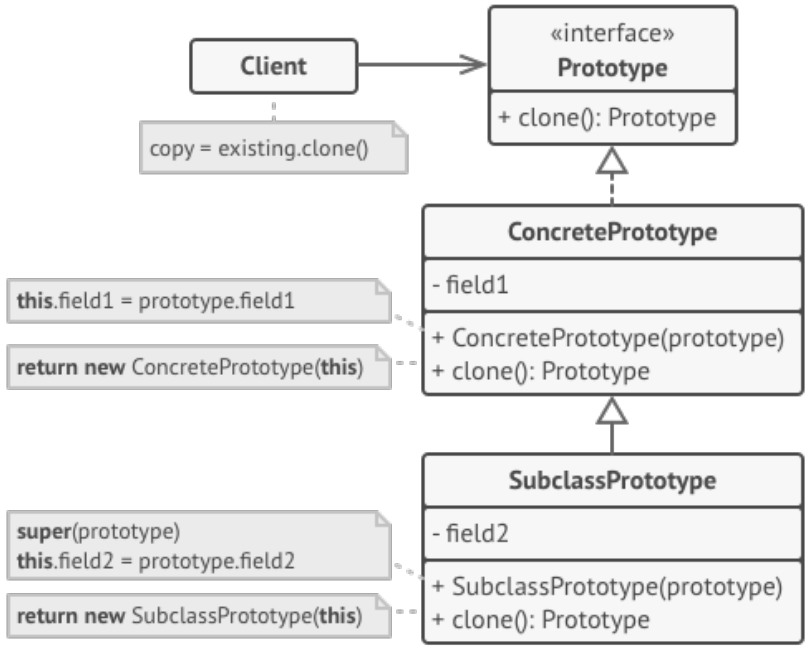


DP de Création - Prototype

- Pour copier un objet il faut parcourir ses attributs, qui théoriquement doivent être privés.
- Pour copier un objet, dans la solution directe (classique) le code est dépendant de la classe à copier et cela suppose de connaitre la classe concrète pas seulement son interface.
- **Solution** : déléguer à l'objet le processus de clonage.

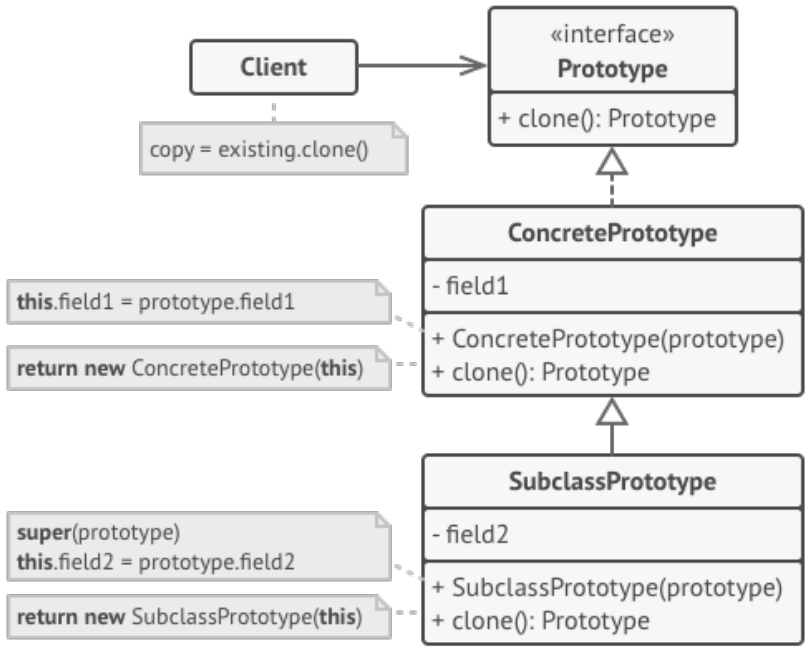
Prototype - Structure

- L'interface `Prototype` déclare les méthodes de clonage.
- La classe `Prototype Concret` implémente la méthode de clonage. Copie les objets et éventuellement prend en charge la complexité de ces objets (objets imbriqués).



Prototype - Structure

- Le client peut produire n'importe quel objet implémentant l'interface `Prototype`.



Java Code

```
interface Person {
    Person clone();
}

class Tom implements Person {
    private final String NAME = "Tom";

    @Override
    public Person clone() {
        return new Tom();
    }

    @Override
    public String toString() {
        return NAME;
    }
}
```

Java Code

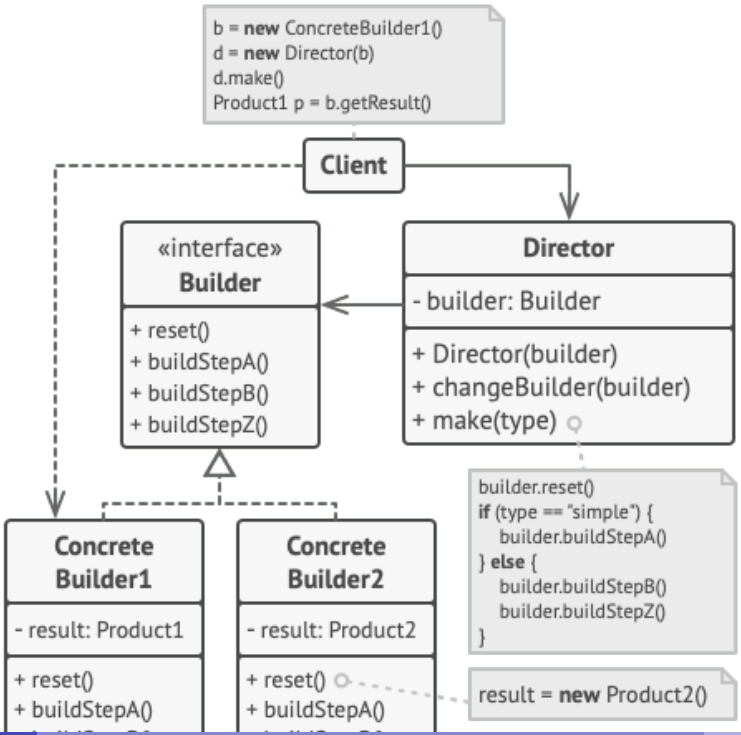
```
class Dick implements Person {
    private final String NAME = "Dick";

    @Override
    public Person clone() {
        return new Dick();
    }

    @Override
    public String toString() {
        return NAME;
    }
}

class Harry implements Person {
    private final String NAME = "Harry";

    @Override
    public Person clone() {
        return new Harry();
    }

    @Override
    public String toString() {
        return NAME;
    }
}
```

Java Code

```
class Factory {
    private static final Map<String, Person> prototypes = new HashMap<>();

    static {
        prototypes.put("tom", new Tom());
        prototypes.put("dick", new Dick());
        prototypes.put("harry", new Harry());
    }

    public static Person getPrototype(String type) {
        try {
            return prototypes.get(type).clone();
        } catch (NullPointerException ex) {
            System.out.println("Prototype with name: " + type + ", doesn't exist");
            return null;
        }
    }
}
public class PrototypeFactory {
    public static void main(String[] args) {
        if (args.length > 0) {
            for (String type : args) {
                Person prototype = Factory.getPrototype(type);
                if (prototype != null) {
                    System.out.println(prototype);
                }
            }
        } else {
            System.out.println("Run again with arguments of command string ");
        }
    }
}
```

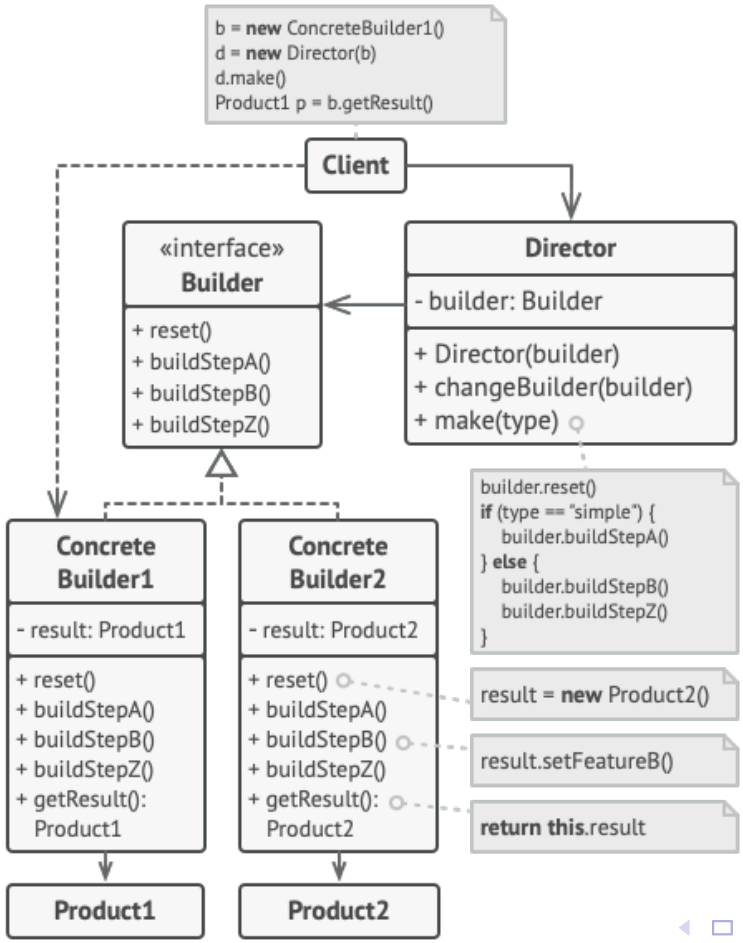
DP Création - Builder

- **Intention** : séparer la construction d'un objet complexe des éléments qui le compose.
- L'interface du `Builder` déclare les étapes communes pour la construction du produit entre tous les builders. Il organise la construction d'objets à partir de ces étapes.
- Le `Builder` extrait le code du constructeur d'objet de sa classe et le déplace (délégation) dans des objets appelés builder.



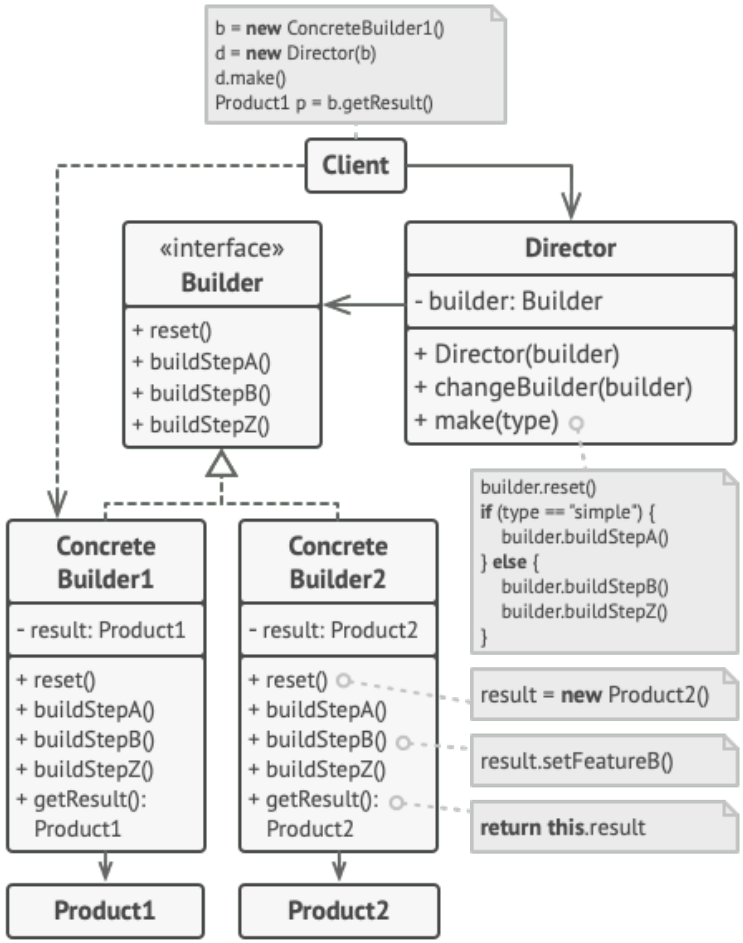
DP Création - Builder

- Les builders concrets possèdent plusieurs implémentation de ces étapes.
- Les builders concrets peuvent créer des objets qui ne reprennent pas l'interface commune.



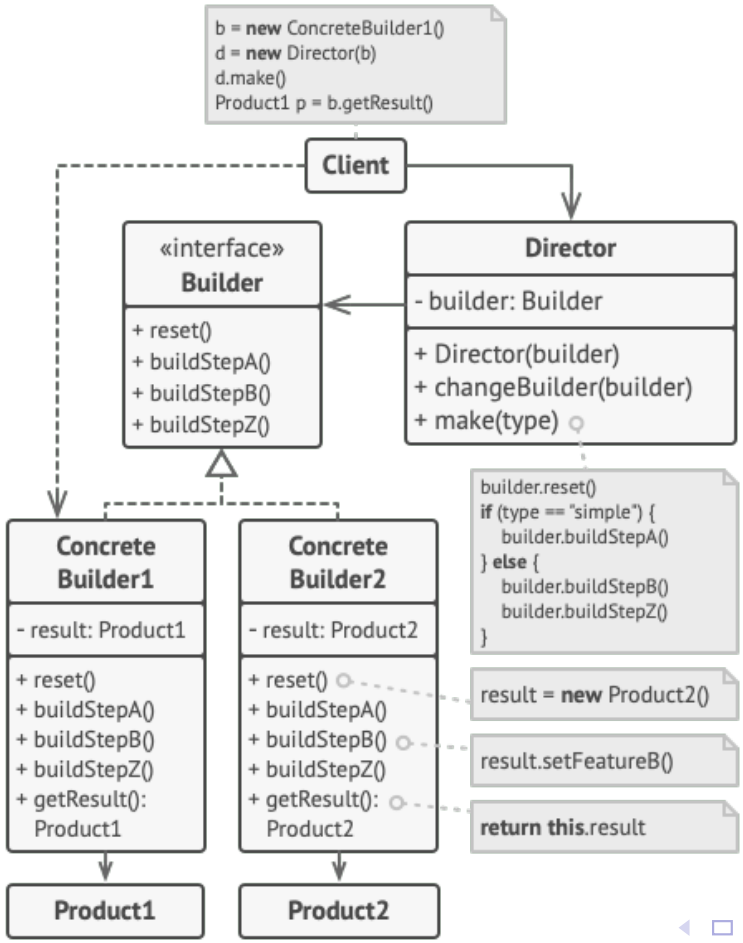
DP Création - Builder

- Les produits sont les résultats retournés.
- Les produits ne sont pas obligés d'appartenir à la même hiérarchie ni d'avoir la même interface.



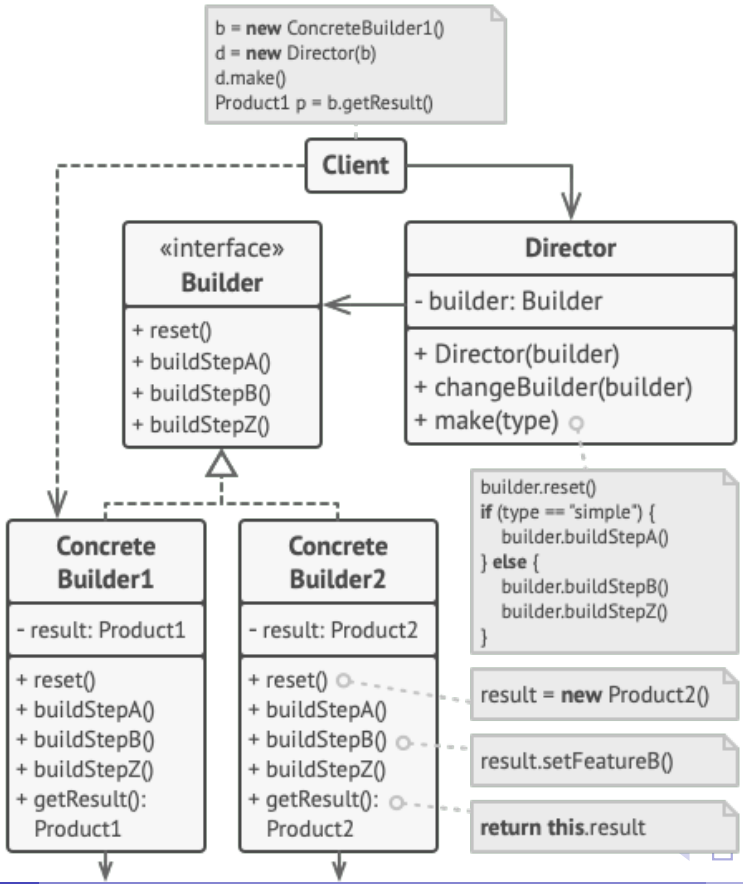
DP Création - Builder

- Le responsable des traitements, le directeur, indique l'ordonnancement des étapes.
- Il devient possible de créer et/ou réutiliser des configurations spécifiques pour les produits.



DP Création - Builder

- L'association entre un directeur est un builder est réalisé par le client.
- Il est aussi possible au client de passer un objet monteur à une méthode de production du directeur. Un monteur différent peut être utilisé à chaque production.



Java Code

```
/** "Product" */
class Pizza
{
    private String dough = "";
    private String sauce = "";
    private String topping = "";

    public void setDough(String dough)
    { this.dough = dough; }
    public void setSauce(String sauce)
    { this.sauce = sauce; }
    public void setTopping(String topping)
    { this.topping = topping; }
}
/** "Abstract Builder" */
abstract class PizzaBuilder
{
    protected Pizza pizza;

    public Pizza getPizza()
    {
        return pizza;
    }
    public void createNewPizzaProduct()
    {
        pizza = new Pizza();
    }
    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

Java Code

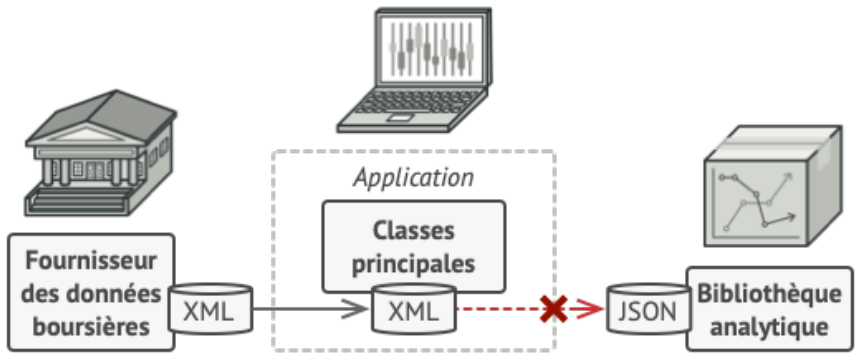
```
 /** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder
{
    public void buildDough()
    {
        pizza.setDough("cross");
    }
    public void buildSauce()
    {
        pizza.setSauce("mild");
    }
    public void buildTopping()
    {
        pizza.setTopping("ham+pineapple");
    }
}

/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder
{
    public void buildDough()
    {
        pizza.setDough("pan baked");
    }
    public void buildSauce()
    {
        pizza.setSauce("hot");
    }
```

Java Code

```
 /** "Director" */
class Cook
{
    private PizzaBuilder pizzaBuilder;
     public void setPizzaBuilder(PizzaBuilder pb)
    {
        pizzaBuilder = pb;
    }
    public Pizza getPizza()
    {
        return pizzaBuilder.getPizza();
    }
    public void constructPizza()
    {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
 /** A given type of pizza being constructed. */
class BuilderExample
{
    public static void main(String[] args)
    {
        Cook cook = new Cook();
        PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
        PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();
        cook.setPizzaBuilder( hawaiianPizzaBuilder );
        cook.constructPizza();
        Pizza pizza = cook.getPizza();
    }
}
```

DP Structurel- Adapter

- Permet de faire communiquer des objets ayant des interfaces incompatibles.
- Exemple : vous surveillez les cours de la bourse. Votre application télécharge des données au format XML depuis différentes sources. Tout fonctionne parfaitement.
  Après quelques temps vous désirez étendre les fonctionnalités en utilisant une bibliothèque existante qui ne fonctionne qu'avec des données JSON.
- Modifier la bibliothèque est une mauvaise idée.

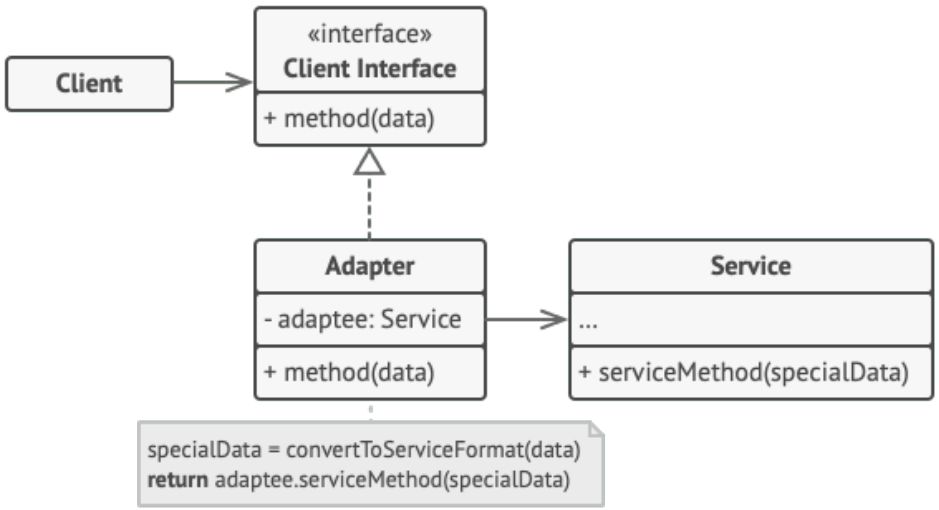


DP Structurel- Adapter

- **Solution** : Créer un objet qui convertit l'interface d'un objet pour qu'un autre objet puisse le comprendre.
- Un adapter encapsule un des objets pour masquer la complexité de la conversion.
- L'objet encapsulé ne connait pas l'adapter.
- Il est possible de créer un adapter qui travaille dans les 2 sens.

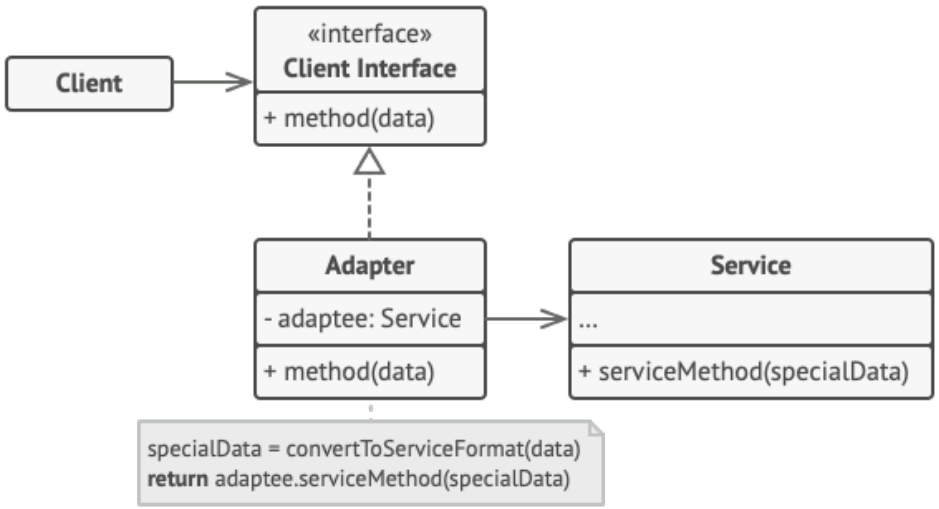
Adapter - Structure

- Le client est une classe qui contient la logique métier du programme
- L'interface Client décrit un protocole que les autres classes doivent implémenter afin de pouvoir collaborer avec le code client.



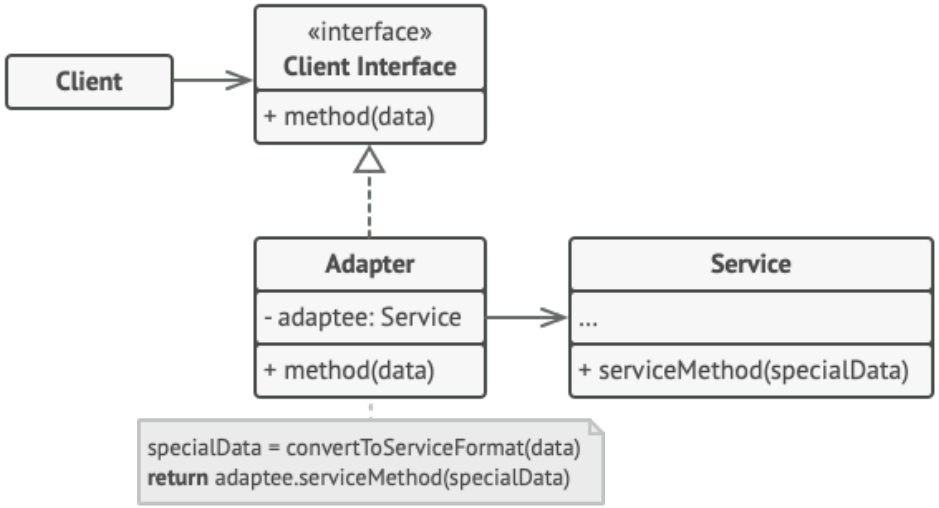
Adapter - Structure

- Le Service est une classe (souvent externe) qu'on veut utiliser mais que ne peut l'être directement - incompatibilité d'interface.



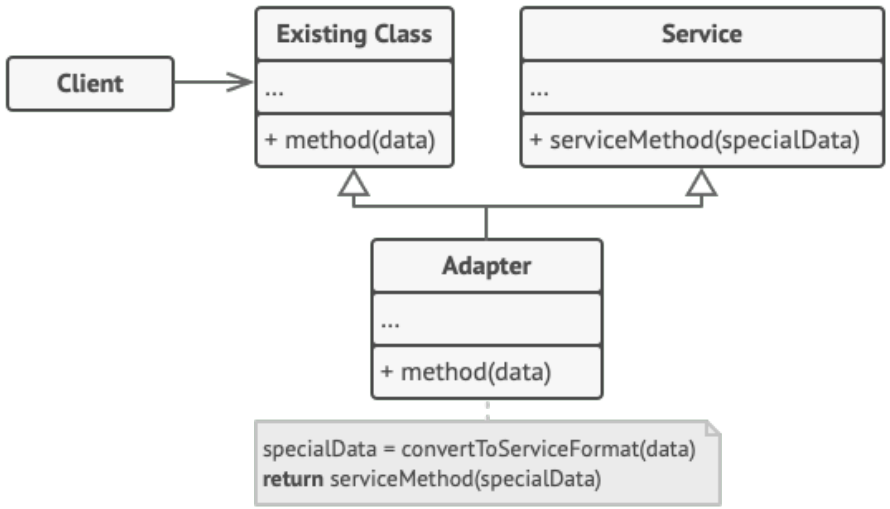
Adapter - Structure

- L'adapteur interagit avec le client et le service qu'il encapsule. L'adapteur convertit les appels du client en appel vers le service dans le format adéquat.
- Le client n'est pas couplé avec la classe de l'adapteur concret. Cela permet de créer de nouveaux adapteurs sans modifier le client.



Adapter de Classe

- Cette implémentation utilise l'héritage.
- L'adaptateur n'a pas besoin d'encapsuler l'objet car il hérite des comportements du client et des services.
- Suppose de pouvoir implémenter l'héritage multiple ou de définir des interfaces.
- L'adaptation se déroule à l'intérieur des méthodes redéfinies.



Exemple - Adapter

- Vous désirez créer un chargeur universel pour tous les portables de votre stock, hors vos portables nécessitent différens voltage. Votre adapter les prendra en charge jusqu'à 10 volts.

```
public class PortableSonneEricSonne
{
   // ne se recharge qu'avec du 10 volts
   public void ChargerBatteries(int volts)
   {
       Console.WriteLine("Portable SonneEricSonne en charge");
       Console.WriteLine("voltage: {0}", volts.ToString());
   }
}
public class PortableSamSaoule
{
   // ne se recharge qu'avec du 5 volts
   public void ChargerPortable(int volts)
   {
       Console.WriteLine("Portable SamSaoule en charge");
       Console.WriteLine("voltage: {0}", volts.ToString());
   }
}
```

Exemple - Adapter

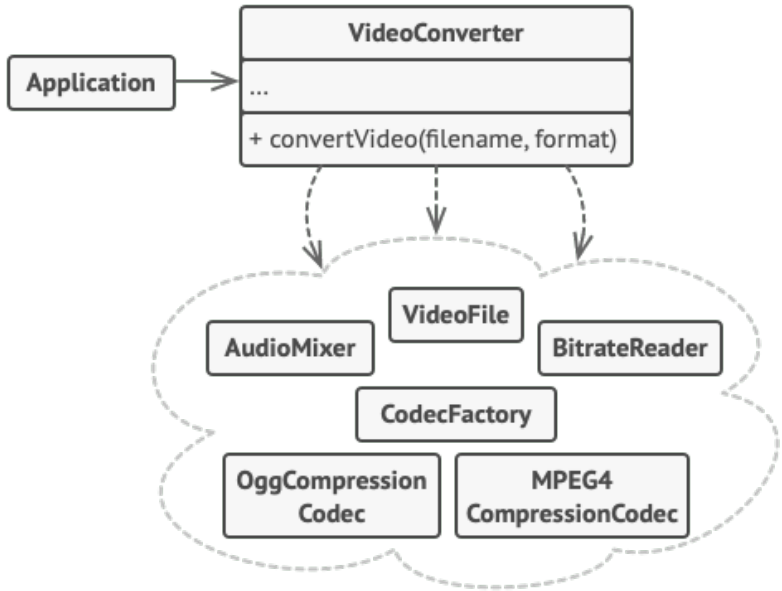
```
 public class Chargeur
 {

   private IChargeable telephone;
   private const int voltage = 10;

   public void brancherPortable(IChargeable portable)
   {
       Console.WriteLine("branchement d'un portable");
       this.telephone = portable;
       this.telephone.Recharger(voltage);
   }
}
public interface IChargeable
{
   void Recharger(int volts);
}
public class PortableDeTest : IChargeable
{
   public void Recharger(int volts)
   {
       Console.WriteLine("Portable de test en charge");
       Console.WriteLine("voltage: {0}", volts.ToString());
   }
}
```

```
 public class AdaptateurSamSaoule : IChargeable
 {
   private PortableSamSaoule telephone;
   public AdaptateurSamSaoule(PortableSamSaoule portable)
   {
       this.telephone = portable;
   }

   public void Recharger(int volts)
   {
       int nouveauVoltage = volts > 5 ? 5 : volts ;
       this.telephone.ChargerPortable(nouveauVoltage);
   }
}
public class AdaptateurSonneEricSonne : IChargeable
{
   private PortableSonneEricSonne telephone;
   public AdaptateurSonneEricSonne(PortableSonneEricSonne portable)
   {
       this.telephone = portable;
   }
   public void Recharger(int volts)
   {
       this.telephone.ChargerBatteries(volts);
   }
}
```

```
  static void main(string[] args)
{
    //on crée le chargeur
    Chargeur chargeur = new Chargeur();

    /******************** Portable SonneEricSonne************************/

    //on crée le portable et son adaptateur
    PortableSonneEricSonne portableSonne = new PortableSonneEricSonne();
    AdaptateurSonneEricSonne adapateurSonne = new AdaptateurSonneEricSonne(portableSonne);

    //on donne le portable à charger, mais en utilisant son adaptateur
    chargeur.brancherPortable(adapateurSonne);

    Console.WriteLine();

    /******************** Portable SamSaoule*************************/

    //on crée le portable et son adaptateur
    PortableSamSaoule portableSam = new PortableSamSaoule();
    AdaptateurSamSaoule adapateurSam = new AdaptateurSamSaoule(portableSam);

    //on donne le portable à charger, mais en utilisant son adaptateur
    chargeur.brancherPortable(adapateurSam);

    Console.ReadLine();
}
```

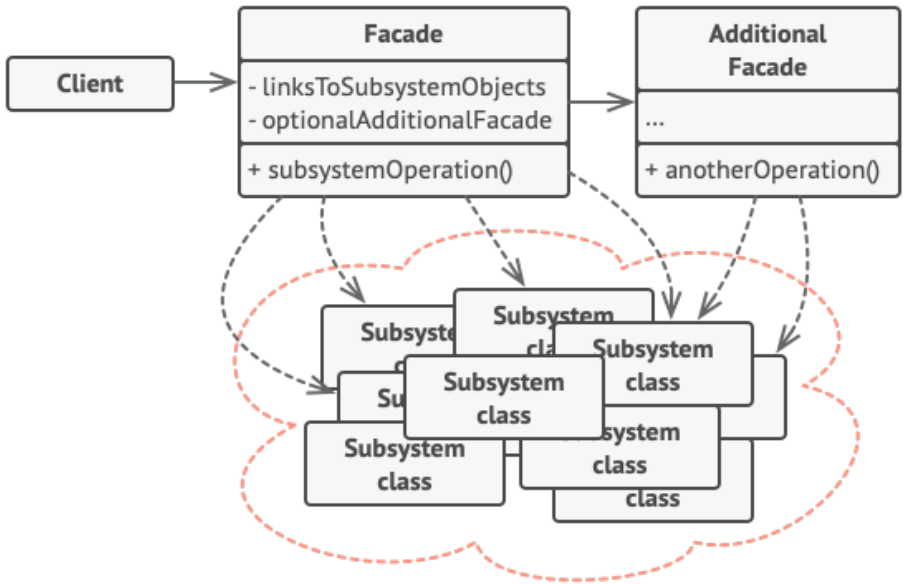
DP Structurel - Facade

- Donne une interface simple à un sous système complexe.
- Les interactions utilisant la facade sont plus limitées que si on interagissait directement avec le sous-système.
- Exemple : plutot que d'adapter le code à des dizaines de classes, la facade pourra vous offrir une interface unique pour convertir vos vidéos.
- Faire évoluer le framework signifie simplement modifier les implémentations des méthodes de la facade.

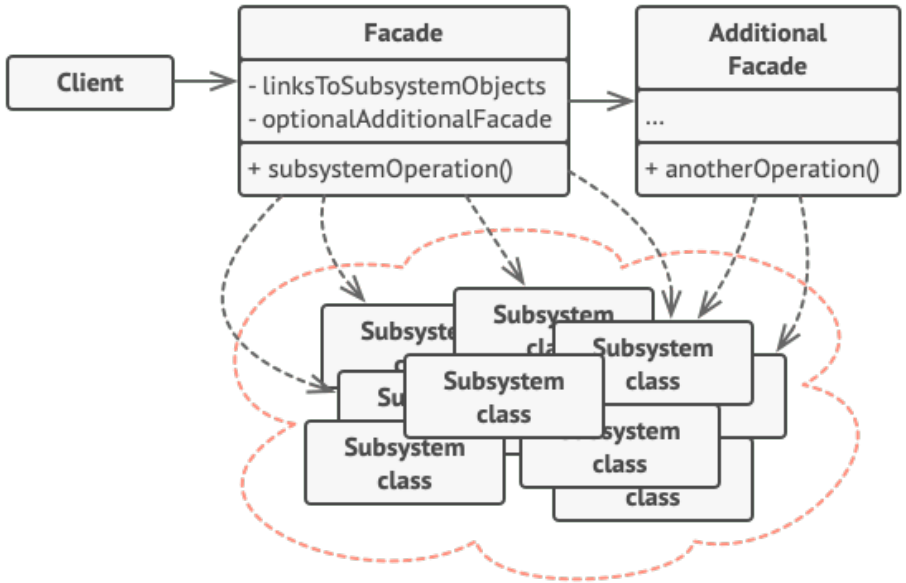


DP Structurel - Facade

- L'objectif est de donner la même interface à un sous-système composé de plusieurs interfaces et objets complexes.
- La facade sait où diriger les requêtes du client et comment utiliser les différentes parties du système (librairie).
- Une facade additionnelle peut éviter la surcharge de la facade principale.



- Le sous-système complexe est composé de dizaines d'objets variés.
- Les classes du sous-système ne connaissent pas la/les facade(s).
- Le client appelle la facade sans jamais se préoccuper du travail effectué par les sous-systèmes.



Java Code

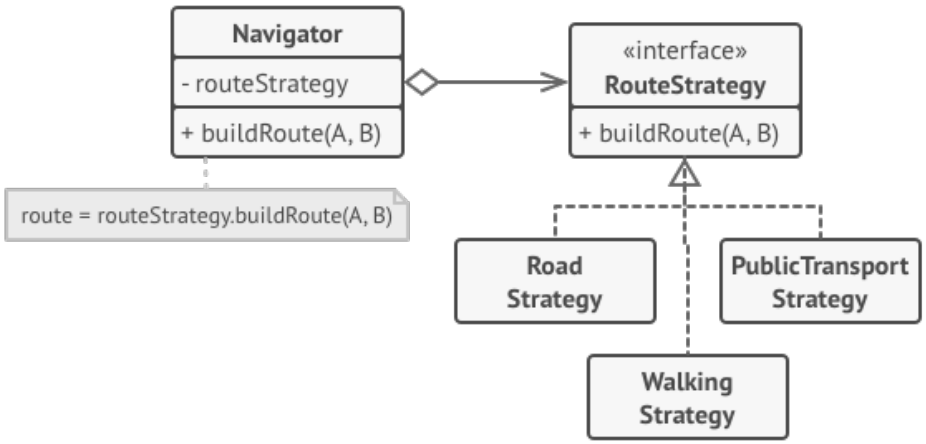
```
/** Facade **/
class UserfriendlyDate
{
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");

    public UserfriendlyDate (String isodate_ymd) throws ParseException {
        Date date = sdf.parse(isodate_ymd);
        cal.setTime(date);
    }

    public void addDays (int days) {
        cal.add (Calendar. DAY_OF_MONTH, days);
    }

    public String toString() {
        return sdf.format(cal.getTime());
    }
}
/** "Client" **/
class FacadePattern
{
    public static void main(String[] args) throws ParseException
    {
        UserfriendlyDate d = new UserfriendlyDate("1980-08-20");
        System.out.println ("Date: " + d.toString());
        d.addDays(20);
        System.out.println ("20 days after: " + d.toString());
    }
}
```

DP - Comportement - Strategy

- Utilisé pour effectuer des échanges d'algorithmes dynamiquement.
- Strategy peut définir des familles de fonctionnalités.
- Ce pattern est directement implémenté dans les langages qui possèdent des fonctions de première classe : python, CLOs ...

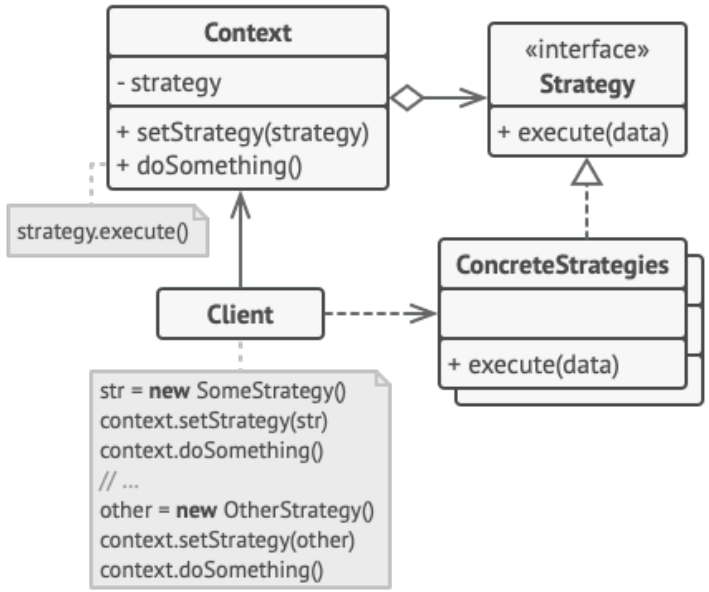
DP - Comportement - Strategy

- Exemple : vous avez envie de créer le nouveau *waze* en prenant en compte que vous pouvez vous déplacer en voiture, à pied ou en transports en commun.



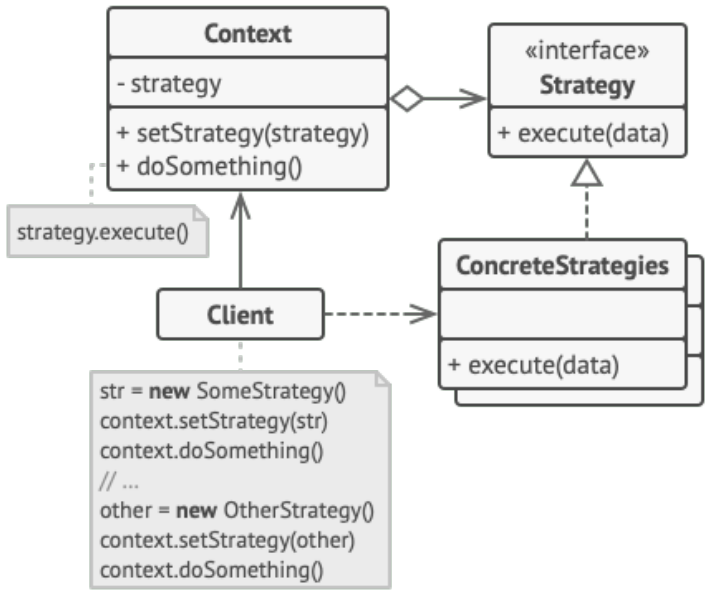
DP - Comportement - Strategy

- Le contexte garde une référence vers une des strategies concrètes et communique avec cet objet uniquement au travers de l'interface strategy.
- L'interface strategie déclare une méthode que le contexte utilise pour exécuter une stratégie.
- Strategie concrète implémente les variantes d'algorithmes.



DP - Comportement - Strategy

- Le contexte appelle la méthode il ne connait pas le fonctionnement de la stratégie.
- Le client crée l'objet Strategie et l'envoie au contexte. Il stocke cet objet.



Strategie - Java Code

```
public interface PaymentStrategy {

   public void pay(int amount);
}

public class CreditCardStrategy implements PaymentStrategy {

   private String name;
   private String cardNumber;
   private String cvv;
   private String dateOfExpiry;

   public CreditCardStrategy(String nm, String ccNum, String cvv, String expiryDate){
      this.name=nm;
      this.cardNumber=ccNum;
      this.cvv=cvv;
      this.dateOfExpiry=expiryDate;
   }
   @Override
   public void pay(int amount) {
      System.out.println(amount +" paid with credit/debit card");
   }

}
```

Strategie - Java Code

```
public class PaypalStrategy implements PaymentStrategy {
   private String emailId;
   private String password;
   public PaypalStrategy(String email, String pwd){
      this.emailId=email;
      this.password=pwd;
   }
   public void pay(int amount) {
      System.out.println(amount + " paid using Paypal.");
   }
}

public class Item {
   private String upcCode;
   private int price;
   public Item(String upc, int cost){
      this.upcCode=upc;
      this.price=cost;
   }

   public String getUpcCode() {
      return upcCode;
   }

   public int getPrice() {
      return price;
   }
}
```

Strategie - Java Code

```
public class ShoppingCart {
   List<Item> items;

   public ShoppingCart(){
      this.items=new ArrayList<Item>();
   }

   public void addItem(Item item){
      this.items.add(item);
   }

   public void removeItem(Item item){
      this.items.remove(item);
   }

   public int calculateTotal(){
      int sum = 0;
      for(Item item : items){
         sum += item.getPrice();
      }
      return sum;
   }

   public void pay(PaymentStrategy paymentMethod){
      int amount = calculateTotal();
      paymentMethod.pay(amount);
   }
}
```

Strategie - Java Code

```
public class ShoppingCartTest {

   public static void main(String[] args) {
      ShoppingCart cart = new ShoppingCart();

      Item item1 = new Item("1234",10);
      Item item2 = new Item("5678",40);

      cart.addItem(item1);
      cart.addItem(item2);

      //pay by paypal
      cart.pay(new PaypalStrategy("myemail@example.com", "mypwd"));

      //pay by credit card
      cart.pay(new CreditCardStrategy("Pankaj Kumar", "1234567890123456", "786", "12/15"));
   }
}
```

DP - Comportement - State

- Permet de modifier le comportement interne d'un objet quand son état interne change.
- Donne l'impression que l'objet change de classe.
- Propose de créer une nouvelle classe par état de l'objet et d'y introduire les comportements spécifiques à cet état. L'objet initial devient alors le contexte.