Les patrons de conception

Frédéric DADEAU

Institut FEMTO-ST – Département d'Informatique des Systèmes Complexes

Bureau 410 C

Email : frederic.dadeau@univ-fcomte.fr

Licence 3 – Année 2023-2024

Les patrons de conception - késako ?

Les patrons de conception (également appelés *Design Patterns* dans la littérature) représentent des solutions génériques supposées répondre à des problèmes spécifiques et récurrents liés à la conception d'un logiciel ou d'une partie de celui-ci.

L'idée est de proposer une solution propre (d'un point de vue programmation objet), et surtout maintenable et extensible.

Une grande variété de patrons de conception existent, on se focalisera ici sur les plus connus, ceux issus des 4 auteurs (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides – le *Gang of Four*) du livre référence "Design Patterns – Elements of Reusable Object-Oriented Software" (1994).

D'autres patrons de conception existent (GRASP, patrons d'entreprise, game programming design patterns, ou d'autres...) nous nous focaliserons ici sur ceux du GoF. L'objectif n'est pas de les connaître sur le bout des doigts, mais de les comprendre, notamment pour les plus classiques (quelques TP seront là pour vous aider à les mettre en œuvre).

Avant de commencer...

Nous allons travailler dans ce cours avec des notions liées à la programmation orientée objet (POO) vue en L1-L2.

Êtes-vous à l'aise avec les notions objet classiques :

- classe ? objet ?
- classe abstraite ? interface ? leurs points communs/différences ? dans quels cas on les utilise ?
- notion OO classiques : encapsulation ? spécialisation ? généralisation ? polymorphisme ?
- la notation UML ?

Si non, ça va être compliqué, et pas que pour ces 3h....

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
- DRY
- YAGNI
- GRASP
- SOLID
- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
- YAGNI
- GRASP
- SOLID
- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
  Don't Repeat Yourself
- YAGNI
- GRASP
- SOLID
- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
  Don't Repeat Yourself
- YAGNI
  You Ain't Gonna Need It
- GRASP
- SOLID
- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
  Don't Repeat Yourself
- YAGNI
  You Ain't Gonna Need It
- GRASP
  General Responsibility Assignment Software Patterns/Principles
- SOLID

- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
  Don't Repeat Yourself
- YAGNI
  You Ain't Gonna Need It
- GRASP
  General Responsibility Assignment Software Patterns/Principles
- SOLID
  Single responsibility, Open/close, Liskov substitution principle, Interface segregation, Dependency inversion
- Loi de Déméter

Avant de commencer...

Êtes-vous à l'aise avec les acronymes/concepts suivants :

- KISS
  Keep It Simple Stupid
- DRY
  Don't Repeat Yourself
- YAGNI
  You Ain't Gonna Need It
- GRASP
  General Responsibility Assignment Software Patterns/Principles
- SOLID
  Single responsibility, Open/close, Liskov substitution principle, Interface segregation, Dependency inversion
- Loi de Déméter
  Principe de connaissance minimale ; "je ne parle pas aux gens que je ne connais pas"

Avant de commencer...

Pour bien comprendre cette section sur les patrons de conception, il faut garder à l'esprit :

- que l'on cherche à écrire du code que l'on ne sera pas le seul à utiliser
- que, bien souvent, considérer que "l'autre développeur n'a qu'à bien utiliser le code que je lui laisse" n'est pas une option
- que l'on cherche à produire du code "propre" du point de vue POO (cf. SOLID) qui soit facilement maintenable et extensible

Les patrons de conception du Gang of Four

Les patrons présentés par le Gang of Four se classent en 3 catégories :

- Les patrons de création, qui décrivent comment résoudre des problèmes d'instanciation de classes (plus précisément : création et configuration d'objets),
- Les patrons de structure, qui décrivent comment structurer les classes afin d'avoir un minimum de dépendance entre l'implémentation et l'utilisation dans différents cas,
- Les patrons de comportement, qui décrivent une structure de classes pour le comportement d'une application.

Quelques références

Références en ligne :

- WikiBooks Design Patterns : `https://fr.wikibooks.org/wiki/Patrons_de_conception` (GoF + autres)
- Refactoring Guru : `https://refactoring.guru/design-patterns` (GoF avec comparaison des DP entre eux)
- Les excellents cours de Sébastien Mosser (UQAM) : chaîne YouTube du ACE Research Lab (chapitres 6 à 8)

Bibliographie :

- Design Patterns – Elements of Reusable Object Oriented Software (Gamma, Helm, Johnson, Vlissides) - Addison Wesley (1994)
- Head-First Design Patterns (Freeman, Freeman, Bates, Sierra) - O'Reilly (2004)

Plan du cours

Les patrons de création

Les patrons de structure

Les patrons de comportement

L'architecture MVC

Plan du cours

Les patrons de création

Un patron de création permet de résoudre les problèmes liés à la création et à configuration d'objets.

- Singleton : utilisé quand une classe ne peut être instancié qu'une seule fois.
- Prototype : permet de créer un nouvel objet par copie d'un objet existant.
- Fabrique : permet de construire un objet dont la classe (i.e. le type) dépend d'un paramètre du constructeur.
- Fabrique abstraite : permet de gérer différentes fabriques à travers l'interface d'une fabrique abstraite.
- Monteur : permet la construction d'objets complexes en construisant chacune de ses parties sans dépendre de la représentation concrète de celles-ci.

Singleton

Objectif

Le *singleton* est un patron de conception dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement).

Utilité

Il est utilisé lorsque l'on a besoin d'exactement un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

Singleton

Objectif

Le *singleton* est un patron de conception dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement).

Principe

On implante le singleton en écrivant une classe contenant une méthode :

- qui crée une instance, uniquement s'il n'en existe pas encore
- qui renvoie une référence vers l'objet qui existe déjà.

Singleton

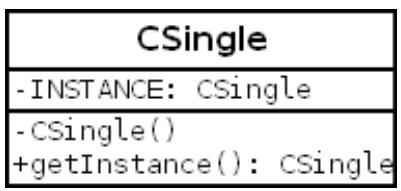
Objectif

Le *singleton* est un patron de conception dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement).

Précautions d'implantation

- Il faudra veiller à ce que le constructeur de la classe soit privé ou bien protégé, afin de s'assurer que la classe ne puisse être instanciée autrement que par la méthode de création contrôlée.
- Le singleton doit être implanté avec précaution dans les applications multi-thread. Si deux processus légers exécutent en même temps la méthode de création alors que l'objet unique n'existe pas encore, il faut absolument s'assurer qu'un seul créera l'objet, et que l'autre obtiendra une référence vers ce nouvel objet (utiliser l'exclusion mutuelle).

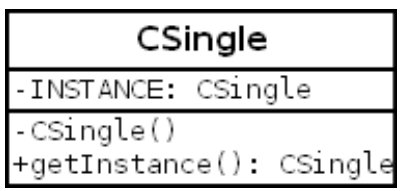
Singleton



Implantation synchronisée (en Java)

```
public class CSingle {

  private static CSingle INSTANCE = null;

  /* Constructeur privé */
  private CSingle() { }

  /* Retourne l'instance du singleton. */
  public synchronized static CSingle getInstance() {
    if (INSTANCE == null) {
      INSTANCE = new CSingle();
    }
    return INSTANCE;
  }

}
```

Singleton



Variante de l'implantation (en Java)

```
public class CSingle {
    /* Construction de l'instance */
  private static final CSingle INST = new CSingle();

  /* Constructeur privé */
  private CSingle() { }

  /* Retourne l'instance du singleton. */
  public static CSingle getInstance() {
    return INST;
  }

}
```

Prototype

Utilité

Le patron de conception *prototype* est utilisé lorsque la création d'une instance est complexe ou consommatrice en temps.

Principe

Plutôt que créer plusieurs instances de la classe, on copie la première instance et on modifie la copie de façon appropriée.

Prototype

Utilité

Le patron de conception *prototype* est utilisé lorsque la création d'une instance est complexe ou consommatrice en temps.

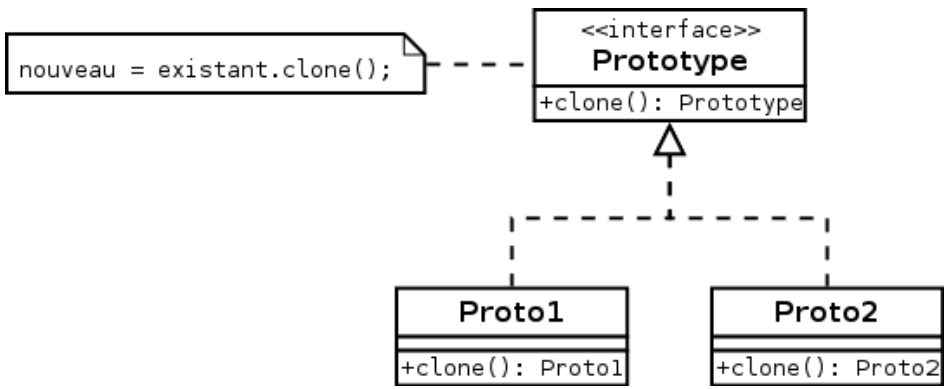
Implantation

Pour implanter ce patron il faut déclarer une classe abstraite spécifiant une méthode abstraite appelée `clone()`. Toute classe nécessitant un constructeur polymorphique dérivera de cette classe abstraite et implantera la méthode `clone()`.
C'est cette méthode qui sera appelée par la classe cliente.

Prototype

Utilité

Le patron de conception *prototype* est utilisé lorsque la création d'une instance est complexe ou consommatrice en temps.



Prototype

Implantation d'un prototype en Java

```
class A implements Cloneable {

  int valeur;

  public A(int i) {
    this.valeur = i;
    this.ressource = ... // opération coûteuse (temps/mémoire)
  }

  public A clone() throws CloneNotSupportedException {
    A r = (A) super.clone();
    r.valeur = this.valeur;
    r.ressource = this.ressource;
    return r;
  }
}
```

Usage :

```
    A objA = new A(42);
    A cloneA = objA.clone();
```

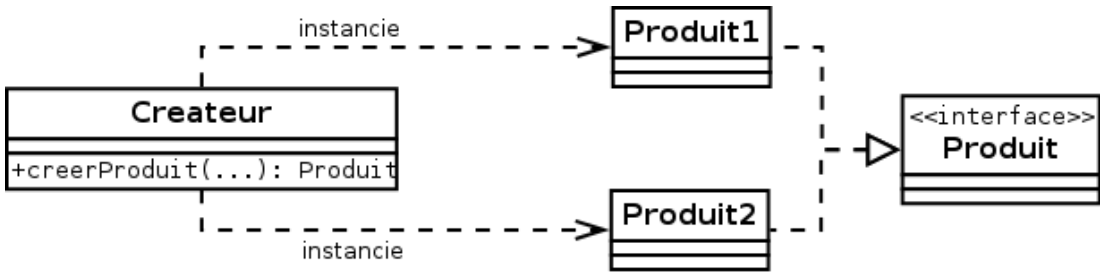
Fabrique

Définition

La *fabrique* (également nommée *factory*) a pour rôle l'instanciation d'objets divers dont le type n'est pas prédéfini : les objets sont créés dynamiquement en fonction des paramètres passés à la fabrique.

Comme en général, les fabriques sont uniques dans un programme, on utilise souvent le patron de conception singleton pour gérer leur création.

Fabrique



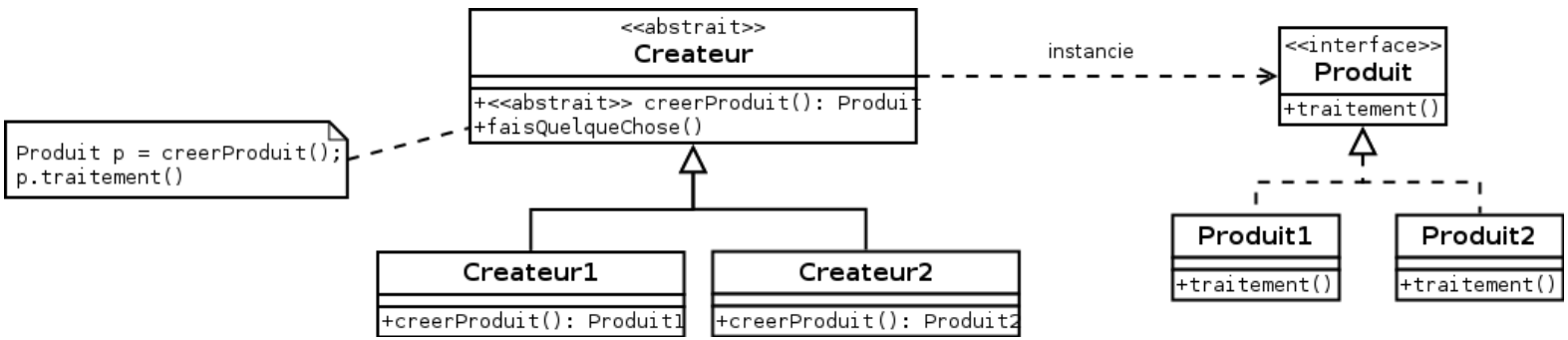
Implantation d'une fabrique en Java

```
public class FabriqueHTMLElement {

  public HTMLElement create(String type) throws ExceptionCreation {
    if (type.equals("div"))
      return new HTMLDivElement();
    if (type.equals("p"))
      return new HTMLParagraphElement();
    if (type.equals("img"))
      return new HTMLImageElement();
    throw new ExceptionCreation("Impossible de créer un " + type);
  }
}
```

Méthode de Fabrique

La définition précédente n'est pas toujours considérée comme étant une bonne utilisation d'une fabrique, jugée relativement restrictive, car elle effectue juste une alternative pour faire des créations d'objets.

Une variante, plus propre, consiste à considérer une méthode de fabrique, abstraite, qui sera en charge de créer un certain type d'objet.



Fabrique Abstraite

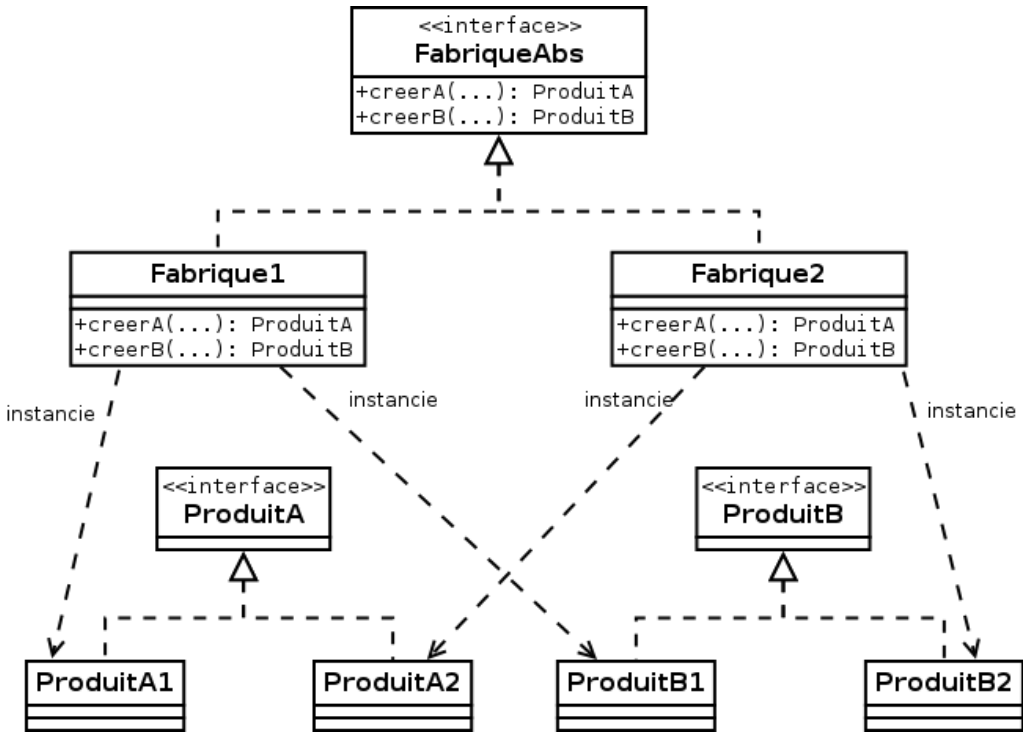
Définition

Une *fabrique abstraite* encapsule un groupe de fabriques ayant une thématique commune, visant chacune à produire une famille de produits. Le code client crée une implémentation concrète de la fabrique abstraite, puis utilise les interfaces génériques pour créer des objets concrets de la thématique. Le client ne se préoccupe pas de savoir laquelle de ces fabriques a donné un objet concret, car il n'utilise que les interfaces génériques des objets produits.

Utilité

Ce patron de conception sépare les détails d'implémentation d'un ensemble d'objets de leur usage générique.

Fabrique Abstraite



La Fabrique1 est en charge de créer la famille de produits 1 (ProduitA1, ProduitB1).
La Fabrique 2 est en charge de créer la famille de produits 2 (ProduitA2, ProduitB2)
$\rightarrow$ pas de risque de créer des produits qui ne sont pas de la même famille

Fabrique Abstraite

Exemple de fabrique abstraite

```
public abstract class GUIFactory {

  public static GUIFactory getFactory() {
    int sys = readFromConfigFile("OS_TYPE");
    return (sys == 0) ?
      new WinFactory() :
      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());
  }
}
```

```
public abstract class Button {
  private String caption;
  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 a OSXButton");
  }
}
```

Usage :

```
GUIFactory f = GUIFactory.getFactory();
Button aButton = f.createButton();
aButton.paint();
```

Monteur

Définition du Monteur

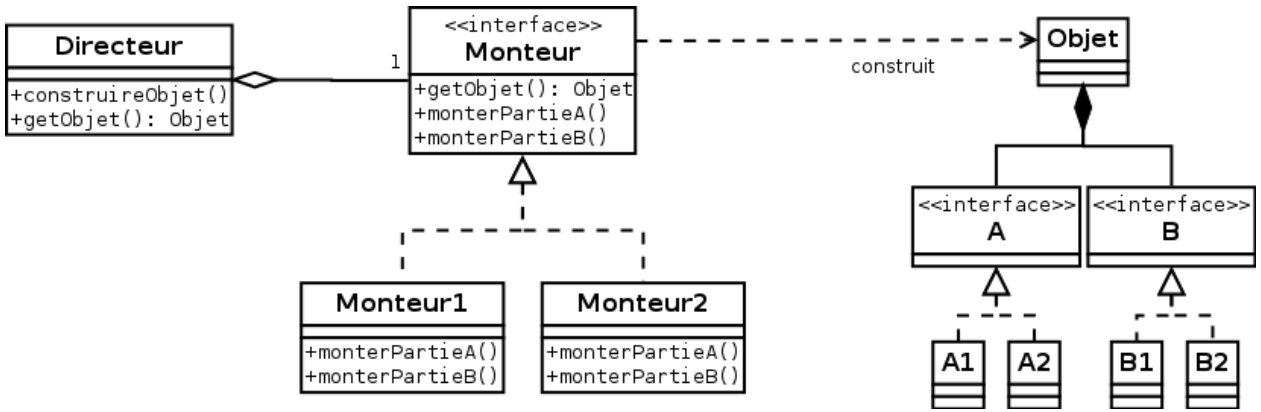
Le *monteur* (également appelé *builder*) est utilisé pour la création d'une variété d'objets complexes qui ont composés de différents composants combinés d'une façon précise. L'objet ainsi composé peut consister en une variété de parties contribuant individuellement à la création de chaque objet complet grâce à un ensemble d'appels à l'interface commune de la classe abstraite Monteur.

Objectif

L'objectif du patron Monteur est

- de séparer la construction d'un objet complexe de la représentation afin que le même processus de construction puisse créer différentes représentations
- d'éviter que la construction d'un objet ne soit réalisée par un constructeur (trop) paramétré
- permettre d'étendre facilement les combinaisons de composants par simple ajout d'un nouveau monteur

Monteur



Le directeur construit un objet en appelant les différentes méthodes afin de construire chaque partie de l'objet complexe. La construction de ces différentes parties est laissée aux implantations de l'interface Monteur qui ne définissent que la manière de construire ces parties.

Monteur

Exemple de patron monteur

```
class Pizza {
  private String pate, sauce, gar;

  public void setPate(String p) {
    pate = p;
  }
  public void setSauce(String s) {
    sauce = s;
  }
  public void setGarniture(String g) {
    gar = g;
  }
}


abstract class MonteurPizza {
  protected Pizza p;

  public Pizza getPizza() {
    return p;
  }
  public void creerPizza() {
    p = new Pizza();
  }
  public abstract void monterPate();
  public abstract void monterSauce();
  public abstract void monterGarniture();
}
```

```
class MonteurPizza4Saisons extends MonteurPizza {

  public void monterPate() {
    p.setPate("fine");
  }
  public void monterSauce() {
    p.setSauce("tomate");
  }
  public void monterGarniture() {
    pizza.setGarniture("poivrons+artichaut");
  }
}

class MonteurPizzaThon extends MonteurPizza {

  public void monterPate() {
    pizza.setPate("épaisse");
  }
  public void monterSauce() {
    pizza.setSauce("tomate");
  }
  public void monterGarniture() {
    pizza.setGarniture("thon+olives");
  }
}
```

Monteur

Exemple de patron monteur

```
class Directeur {

  private MonteurPizza monteurPizza;

  public void setMonteurPizza(MonteurPizza mp) {
    monteurPizza = mp;
  }

  public Pizza getPizza() {
    return monteurPizza.getPizza();
  }

  public void construirePizza() {
    monteurPizza.creerNouvellePizza();
    monteurPizza.monterPate();
    monteurPizza.monterSauce();
    monteurPizza.monterGarniture();
  }

}
```

Usage :

```
Directeur dir = new Directeur();
MonteurPizza mp1 = new MonteurPizza4Saisons();
MonteurPizza mp2 = new MonteurPizzaThon();
dir.setMonteurPizza(mp1);
dir.construirePizza();
Pizza pizza = dir.getPizza();
```

Il est ainsi facile de rajouter un `MonteurPizzaMontDOr` sans rien changer au code déjà existant.

Plan du cours

Les patrons de structure

Un patron de structure permet de résoudre les problèmes liés à la structuration des classes et leur interface.

- Pont : Utilisation d'interface à la place d'implémentation spécifique pour permettre l'indépendance entre l'utilisation et l'implémentation.
- Façade : simplification de l'utilisation d'une interface complexe.
- Adaptateur : permet d'adapter une interface existante à une autre interface.
- Proxy : permet de substituer une classe à une autre en utilisant la même interface afin de contrôler l'accès à la classe (contrôle de sécurité ou appel de méthodes à distance).
- Poids-mouche : permet de diminuer le nombre de classes créées en regroupant les classes similaires en une seule.
- Objet composite : permet de manipuler des objets composites à travers la même interface que les éléments dont ils sont constitués.
- Décorateur : permet d'attacher dynamiquement de nouvelles responsabilités à un objet.

Adaptateur

Définition

L'*adaptateur* permet de convertir l'interface d'une classe en une autre interface que le client attend. Le patron adaptateur fait fonctionner un ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.

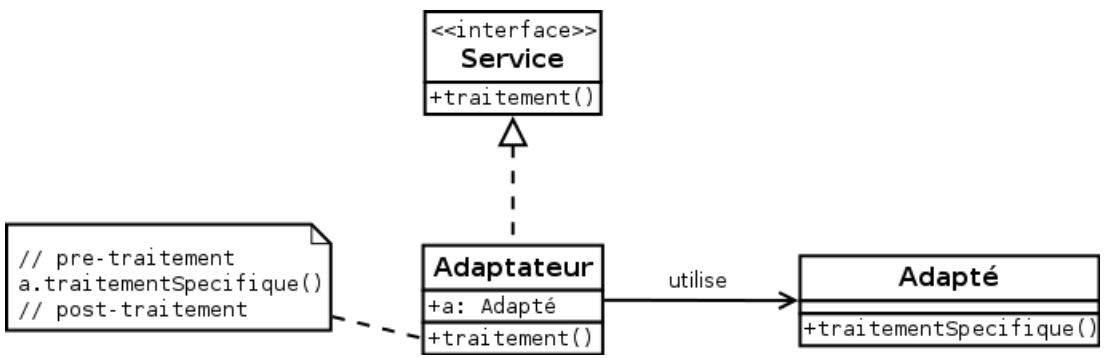
Utilité

- Adapter une API tiers qui convient au besoin fonctionnel du client, dont mais la signature des méthodes ne convient pas.
- Normaliser l'utilisation d'anciennes classes sans pour autant en reprendre tout le code.

Adaptateur

Définition

L'*adaptateur* permet de convertir l'interface d'une classe en une autre interface que le client attend. Le patron adaptateur fait fonctionner un ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces.



Adaptateur

Exemple d'adaptateur

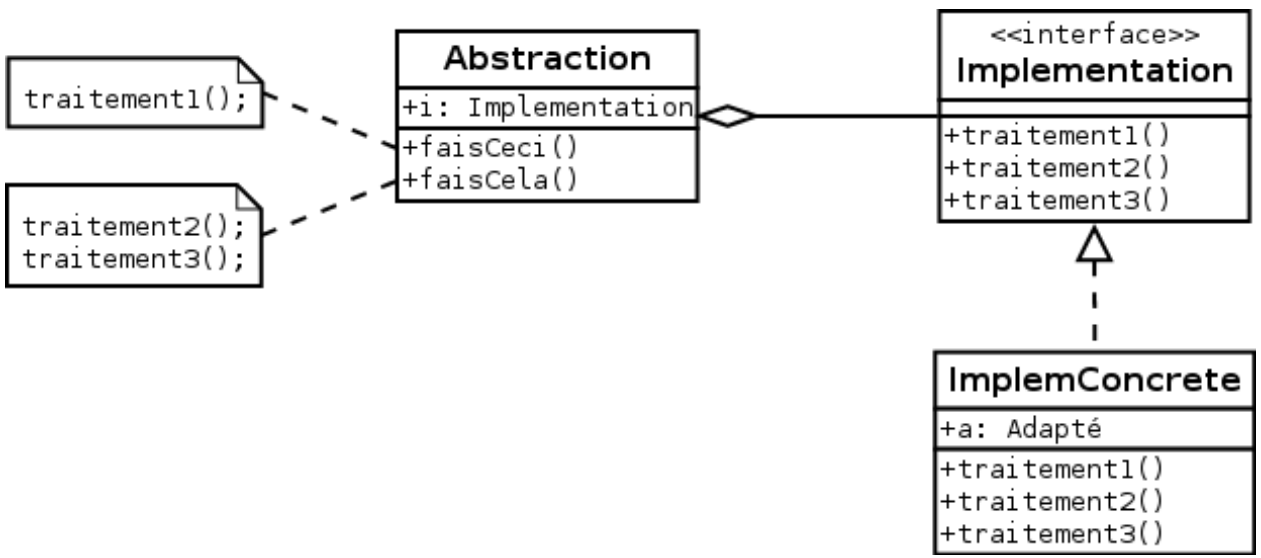
```
public interface IDeveloppeur {
  String EcrireCode();
}

class Enseignant {
  public String EcrireAlgorithme() {
    return "/* algo nickel */";
  }
}

class Adaptateur implements IDeveloppeur {
  Enseignant _ens;
  public Adaptateur (Enseignant e) {
    _ens = e;
  }
  public String EcrireCode() {
    return "int main(String[] args) {" +
           _ens.EcrireAlgorithme() + "}";
  }
}
```

```
class DeveloppeurL3 implements IDeveloppeur {

  public String EcrireCode() {
    return "int main(String[] args) {" +
           "/* code plein de bugs */ }";
  }
}

class Client {
  void Utiliser(IDeveloppeur developpeur) {
    System.out.println(developpeur.EcrireCode());
  }

  public static void main(String[] args) {

    Client client = new Client();
    IDeveloppeur dev1 = new DeveloppeurL3();
    client.Utiliser(dev1);

    Enseignant ens = new Enseignant();
    IDeveloppeur dev2 = new Adaptateur(ens);
    client.Utiliser(dev2);
  }
}
```

Pont

Définition

Le *pont* (également appelé *bridge*) est un patron de conception qui permet de découpler l'interface d'une classe et son implémentation, en déléguant la réalisation de certains traitements à un objet dédié. Ainsi l'interface et l'implémentation peuvent varier séparément.



Pont

Définition

Le *pont* (également appelé *bridge*) est un patron de conception qui permet de découpler l'interface d'une classe et son implémentation, en déléguant la réalisation de certains traitements à un objet dédié. Ainsi l'interface et l'implémentation peuvent varier séparément.

Utilisation

Le pont est lui utilisé pour découpler l'interface de l'implémentation. Ainsi, il est possible de modifier ou de changer l'implémentation d'une classe sans devoir modifier le code client (si l'interface ne change pas bien entendu).

Pont

Exemple : les figures géométriques et leur dessin à l'écran

Des figures géométriques (cercles, rectangles, carrés, etc.)

- propriétés communes (par exemple, une couleur),
- méthodes abstraites communes (par exemple, le calcul de l'aire) implantées dans des classes dérivées (méthode polymorphe).

Toutes les formes peuvent se dessiner sur l'écran, mais la façon de dessiner la forme dépend de facteurs externes aux classes (environnement graphique, OS, etc.).

Au lieu d'ajouter une méthode pour chaque instance de ces différents facteurs (une méthode par environnement, par OS, etc.) on utilise d'une interface spécifique pour les primitives de dessin de manière à ne pas dépendre de l'implémentation de l'objet considéré.

Pont

Exemple : les figures géométriques et leur dessin à l'écran

```
interface DrawingAPI {
  public void drawCircle(double x, double y,
                         double radius);
}

class DrawingAPI1 implements DrawingAPI {
  public void drawCircle(double x, double y,
                         double radius) {
    System.out.println("API1.cercle");
  }
}

class DrawingAPI2 implements DrawingAPI {
  public void drawCircle(double x, double y,
                         double radius) {
    System.out.println("API2.cercle");
  }
}

interface class Shape {
  public void draw();
  public void getSurface(double pct);
}
```

```
class CircleShape implements Shape {
  private double x, y, radius;
  private DrawingAPI drawingAPI;

  public CircleShape(double x, double y,
                     double radius,
                     DrawingAPI drawingAPI) {
    this.x = x;
    this.y = y;
    this.radius = radius;
    this.drawingAPI = drawingAPI;
  }

  // bas niveau (spécifique à une implémentation)

  public void draw() {
    drawingAPI.drawCircle(x, y, radius);
  }

  // haut niveau (spécifique à l'abstraction)
  public void getSurface(double pct) {
    ...
  }
}
```

Façade

Définition

Le patron de conception *façade* a pour but de cacher une conception et une interface ou un ensemble d'interfaces complexes difficiles à comprendre.

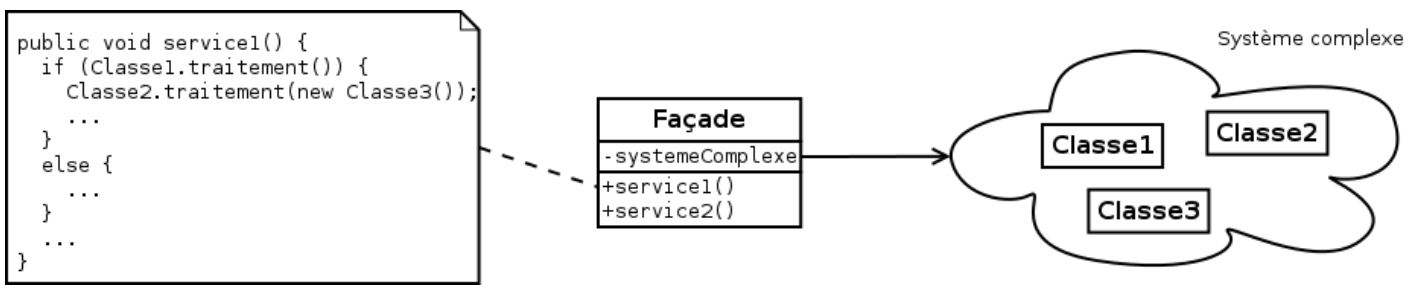
Utilité

La façade permet de simplifier cette complexité en fournissant une interface simple du sous-système. Habituellement, la façade est réalisée en réduisant les fonctionnalités de ce dernier mais en fournissant toutes les fonctions nécessaires à la plupart des utilisateurs.

Façade

Définition

Le patron de conception *façade* a pour but de cacher une conception et une interface ou un ensemble d'interfaces complexes difficiles à comprendre.



La façade sert de guichet pour réaliser des tâches complexes, orchestrées sans que le client ne les voit.

Façade

Classe Clavier

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;

public class Clavier {

  private static BufferedReader flux = new BufferedReader(new InputStreamReader(System.in));

  public static int saisirInt() {
    int i = 0;
    boolean ko = true;
    while (ko) {
      try {
        i = Integer.valueOf(flux.readLine()).intValue();
        ko = false;
      } catch (Exception e) {
        System.err.println("Erreur : la valeur saisie n'est pas un int. Recommencez.");
      }
    }
    return i;
  }
  ...
}
```

Proxy

Définition

La patron *proxy* (également appelé procuration) est une classe se substituant à une autre classe. Par convention et simplicité, le proxy implémente la même interface que la classe à laquelle il se substitue. Le proxy sert à gérer l'accès à un objet, il agit comme un intermédiaire entre la classe utilisatrice et l'objet.

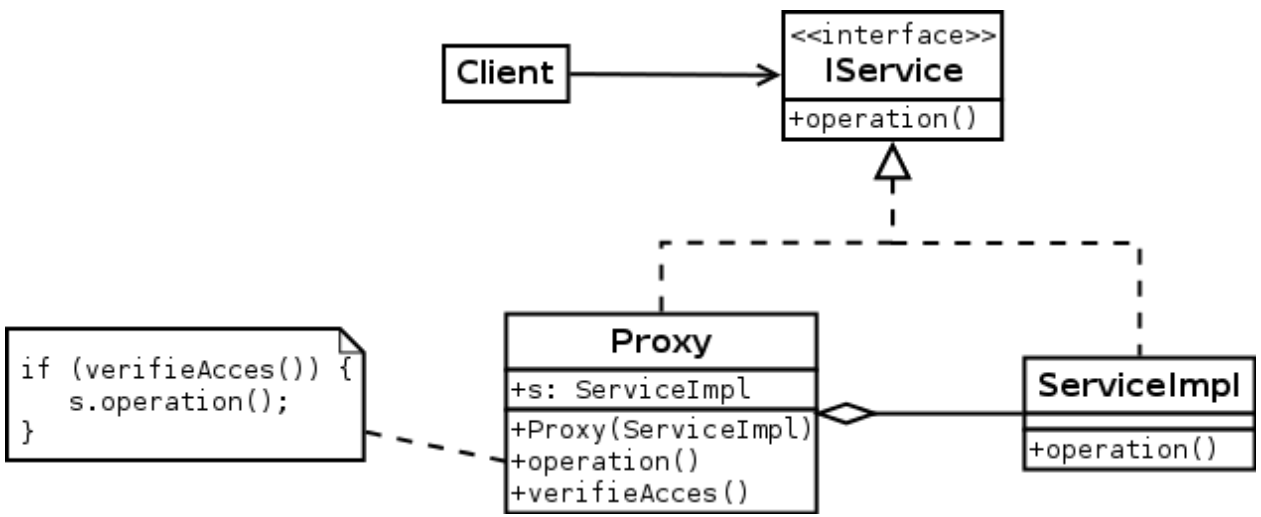
Utilité

Un proxy est utilisé principalement pour contrôler l'accès aux méthodes de la classe substituée.
Outre l'utilisation principale du proxy (contrôle des accès), ce dernier est également utilisé pour simplifier l'utilisation d'un objet complexe à la base. Par exemple, si l'objet doit être manipulé à distance (via un réseau) ou si l'objet est consommateur de temps.

Proxy

Définition

La patron *proxy* (également appelé procuration) est une classe se substituant à une autre classe. Par convention et simplicité, le proxy implémente la même interface que la classe à laquelle il se substitue. Le proxy sert à gérer l'accès à un objet, il agit comme un intermédiaire entre la classe utilisatrice et l'objet.



Proxy

Exemple de Proxy

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

class RealImage implements Image {

  private String file;

  public RealImage(String file) {
    this.file = file;
    loadImageFromDisk();
  }

  private void loadImageFromDisk() {
    System.out.println("Chargement de "+file);
    // Opération potentiellement coûteuse en temps

  }

  public void displayImage() {
    System.out.println("Affichage de "+file);
  }

}
```

```
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);
    }
    image.displayImage();
  }
}

class ProxyExample {
  public static void main(String[] args) {
    Image image1 = new ProxyImage("Photo1");
    Image image2 = new ProxyImage("Photo2");
    Image image3 = new ProxyImage("Photo3");
    image1.displayImage(); // chargement
    image2.displayImage(); // chargement
    image1.displayImage();
    // troisième image jamais chargée
  }
}
```

Poids-mouche

Objectif

L'objectif du poids-mouche est de limiter la consommation de mémoire lorsqu'une très grande quantité de petits objets avec des propriétés communes sont créés.

Nuage de particules

On souhaite créer un nuage de particules dans un jeu vidéo. Chaque particule possède :

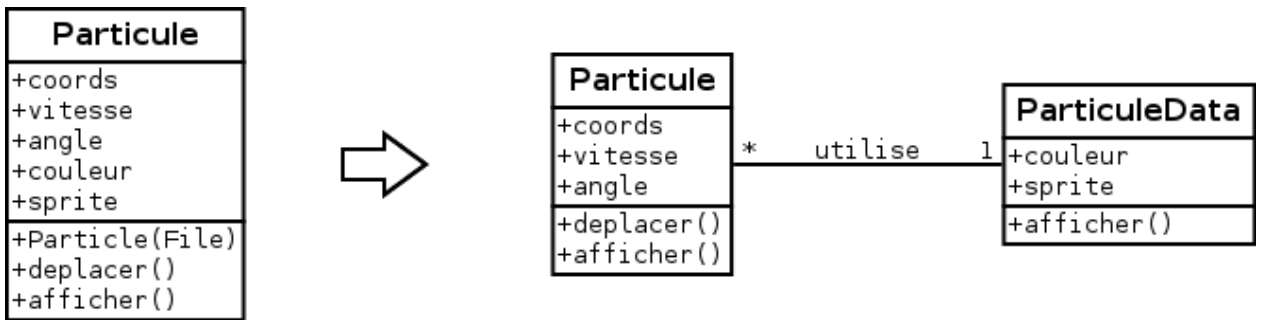
- des coordonnées, une vitesse, un angle ; propres aux particules ($\sim$ 30 octets)
- un sprite, une couleur : 3 combinaisons différentes ($\sim$ 20 Ko pour chacune)
- des méthodes pour faire se déplacer la particule et l'afficher à l'écran

Sprites et couleurs, bien qu'en nombre limités, sont dupliqués dans chaque objet : 1.000.000 particules = $\sim$ 20 Go

Poids-mouche

Principe

Le principe du poids-mouche consiste à séparer les propriétés intrinsèques (spécifique à l'objet) des propriétés extrinsèques (factorisables par ailleurs). Ces dernières sont encapsulées dans des objets spécifiques dont on garde une référence qui sera donnée aux objets ainsi instanciés.
Les appels de méthodes sont généralement propagés aux références.



$\Rightarrow$ 1.000.000 particules = $\sim$ 30 Mo (vs. $\sim$ 20 Go)

Poids-mouche

Exemple de poids-mouche

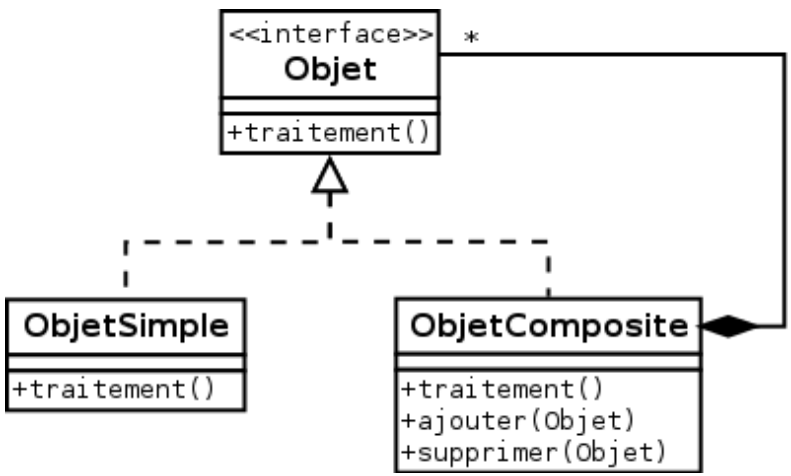
```
public class Particle {

  short x, y;
  byte vecX, vecY;
  short angle;
  ParticleData data;

  public Particle(short x, short y,
                  File f, int color) {
    this.x = x;
    this.y = y;
    this.vecX = this.vecY = 0;
    this.data =
        ParticleData.get(new Pair(f, color));
  }

  public void deplacer() {
    this.x += ...
  }

  public void afficher() {
    data.afficher(this.x, this.y, ...);
  }

}
```

```
class ParticleData {
  int color;
  Image sprite;
  Map<Pair,ParticleData> map =
      new Map<Pair,ParticleData>();

  private ParticleData(File f, int color) {
    this.sprite = loadFromFile(f);
    this.color = c;
  }

  public ParticleData get(File f, int c) {
    ParticleData p;
    Pair key = new Pair(f, c);
    if (map.keySet().contains(key)) {
      p = map.get(key);
    }
    else {
      map.put(key, p = new ParticleData(f, c));
    }
    return p;
  }

  public void afficher(short x, short y, ...) {
    // affichage
  }
}
```

Objet composite

Définition

Un *objet composite* est constitué d'un ou de plusieurs objets similaires (ayant des fonctionnalités similaires). L'idée est de manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet. Les objets ainsi regroupés doivent posséder des opérations communes, c'est-à-dire un "dénominateur commun".



Objet composite

Exemple d'objet composite : dossiers et fichiers

```
interface Node {
  String name;
  void ls(String prefix);
}

class Directory implements Node {

  ArrayList<Node> mChildNode =
    new ArrayList<Node>();

  public void ls(String prefix) {
    System.out.println(prefix + name);
    for (Node n : mChildNode) {
      n.ls(prefix + " ");
    }
  }

  public void add(Node n) {
    mChildNode.add(n);
  }

  public void remove(Node n) {
    mChildNode.remove(n);
  }

}
```

```
class File implements Node {
  public void ls(String prefix) {
    System.out.println(prefix + name);
  }
}

class Program {
  public static void main(String[] args) {
    Directory d1 = new Directory("d1");
    Directory d2 = new Directory("d2");
    File f1 = new File("f1");
    File f2 = new File("f2");
    File f3 = new File("f3");
    d1.add(f1);
    d1.add(d2);
    d2.add(f2);
    d2.add(f3);
    d1.ls("");
  }
}
```

Décorateur

Définition

Un *décorateur* permet d'attacher dynamiquement de nouveaux comportements ou responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.

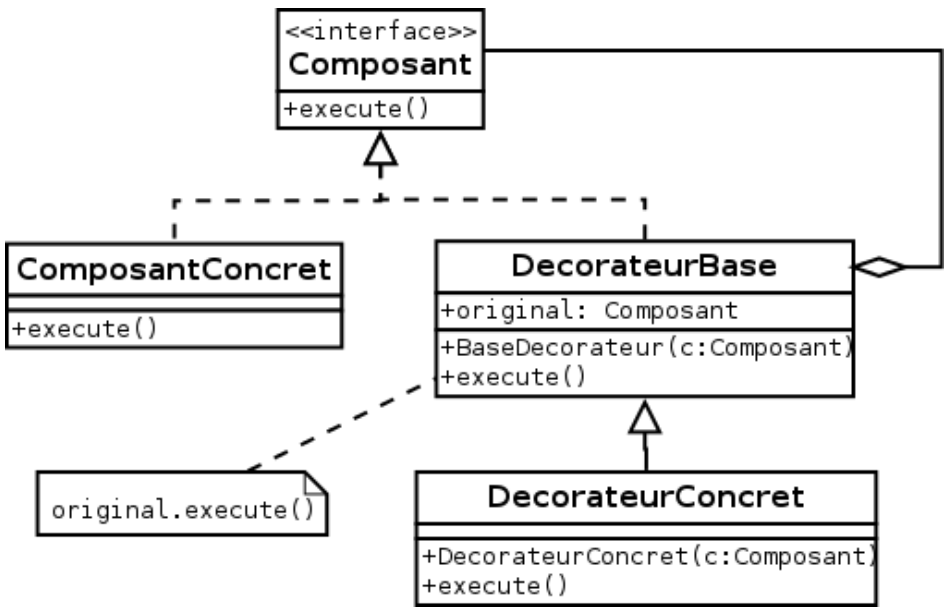
Utilité

Ce patron est à utiliser lorsque l'on souhaite ajouter de nouveaux comportements "à la carte" sans avoir à faire des hiérarchies d'héritages complexes qui prennent en compte toutes les combinaisons possibles d'ajouts.

Décorateur

Définition

Un *décorateur* permet d'attacher dynamiquement de nouveaux comportements ou responsabilités à un objet. Les décorateurs offrent une alternative assez souple à l'héritage pour composer de nouvelles fonctionnalités.



Décorateur

Exemple de décorateur

```
interface Voiture {
  public double prix();
}

class AstonMartin implements Voiture {
  public double prix() { return 200000.00; }
}

class Option implements Voiture {

  Voiture _originale;
  double tarifOption;

  public Option(Voiture o, double t) {
    _originale = o;
    tarifOption = t;
  }

  public double prix() {
    return _originale.prix() + tarifOption;
  }

}
```

```
class Climatisation extends Option {
  public Climatisation(Voiture o) {
    super (o,1000); }
}

class Parachute extends Option {
  public Parachute(Voiture o) {
    super (o,10000); }
}

class LanceMissile extends Option {
  public LanceMissile(Voiture o) {
    super (o,25000); }
}

class Program {
  public static void main(String[] args) {
    Voiture martin = new AstonMartin();
    martin = new Climatisation(martin);
    martin = new Parachute(martin);
    martin = new LanceMissile(martin);
    System.out.println("prix = " + martin.prix());

  }
}
```

Plan du cours

Les patrons de comportement

Un patron de comportement permet de résoudre les problèmes liés aux comportements et à l'interaction entre les classes.

- Chaîne de responsabilité : construit une chaîne de traitement d'une requête.
- Mémento : permet à un objet de sauvegarder/restaurer son état interne.
- Commande : encapsule l'invocation d'une commande.
- Stratégie : permet de changer dynamiquement le comportement d'une méthode.
- Etat : permet de définir différents comportements à un objet en fonction de son état.
- Itérateur : permet de parcourir un ensemble d'objets à l'aide d'un curseur.

Les patrons de comportement

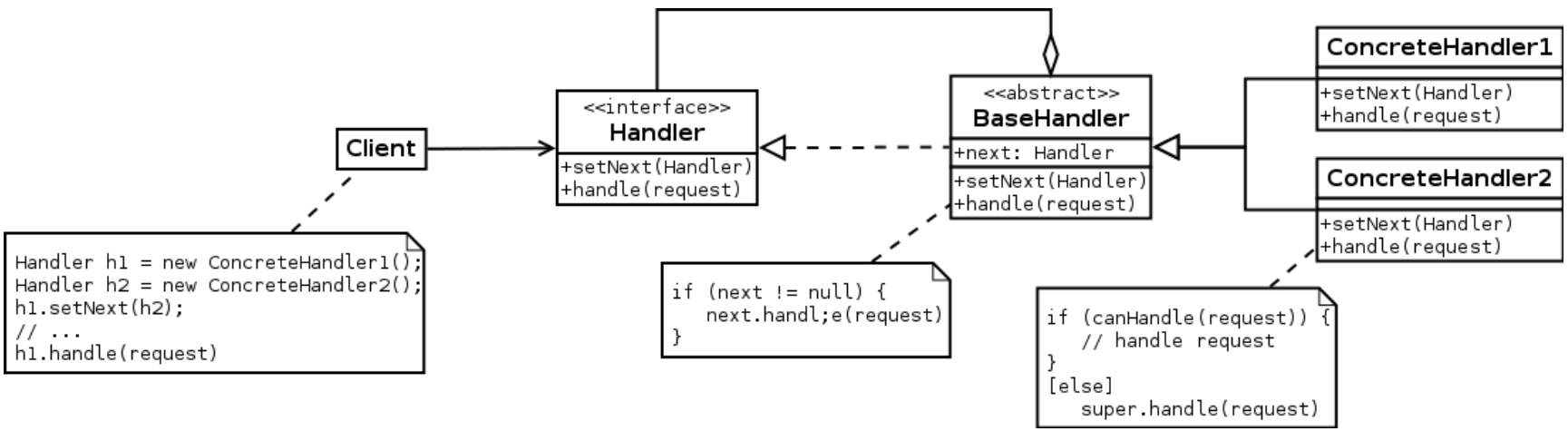
Un patron de comportement permet de résoudre les problèmes liés aux comportements et à l'interaction entre les classes.

- Observateur : intercepter un évènement pour le traiter.
- Médiateur : centralise les interactions entre objets.
- Patron de méthode : définir un modèle de méthode avec des méthodes abstraites.
- Interpréteur : interpréter un langage spécialisé.
- Visiteur : découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.

Chaîne de responsabilité

Définition

Le patron de conception *chaîne de responsabilité* permet à un nombre quelconque de classes d'essayer de répondre à une requête sans connaître les possibilités des autres classes sur cette requête. Le seul lien commun entre ces objets étant cette requête qui passe d'un objet à l'autre jusqu'à ce que l'un des objets puisse répondre (ou que chacun ait répondu).



Chaîne de responsabilité

Exemple de chaîne de responsabilité : un logger

```
abstract class Logger {
  public static final int ERR=0,NOTICE=1,DEBUG=2;

  protected int level;

  protected Logger(int level) {
    this.level = level;
    this.next = null;
  }
  // Le suivant dans la chaine
  protected Logger next;
  public Logger setNext(Logger l) {
    next = l;
    return l;
  }
  public void message(String msg, int prio) {
    if (prio <= level) { writeMessage(msg); }
    if (next!=null) { next.message(msg, prio); }
  }
  abstract protected void writeMessage(String m);

}

class StdoutLogger extends Logger {
  public StdoutLogger(int lvl) { super (lvl); }
  protected void writeMessage(String m) {
    System.out.println("Sur stdout: " + m);
  }
}
```

```
class EmailLogger extends Logger {
  public EmailLogger(int lvl) { super (lvl); }
  protected void writeMessage(String m) {
    System.out.println("Via email: " + m);
  }
}

class StderrLogger extends Logger {
  public StderrLogger(int lvl) { super (lvl); }
  protected void writeMessage(String m) {
    System.err.println("Via stderr: " + m);
  }
}

public class ChainOfResponsibilityExample {
  public static void main(String[] args) {
    Logger l ,l1;
    l1=l=new StdoutLogger(Logger.DEBUG);
    l1=l1.setNext(new EmailLogger(Logger.NOTICE));

    l1=l1.setNext(new StderrLogger(Logger.ERR));
    l.message( "Entering y().", Logger.DEBUG);
    l.message( "Step1 complete.", Logger.NOTICE);
    l.message( "Error occurred.", Logger.ERR );
  }
}
```

Mémento

Objectif

Le patron Memento a pour objectif de permettre à un objet de sauvegarder et restaurer son état interne, sans pour autant exposer son modèle de données à l'extérieur de l'objet.

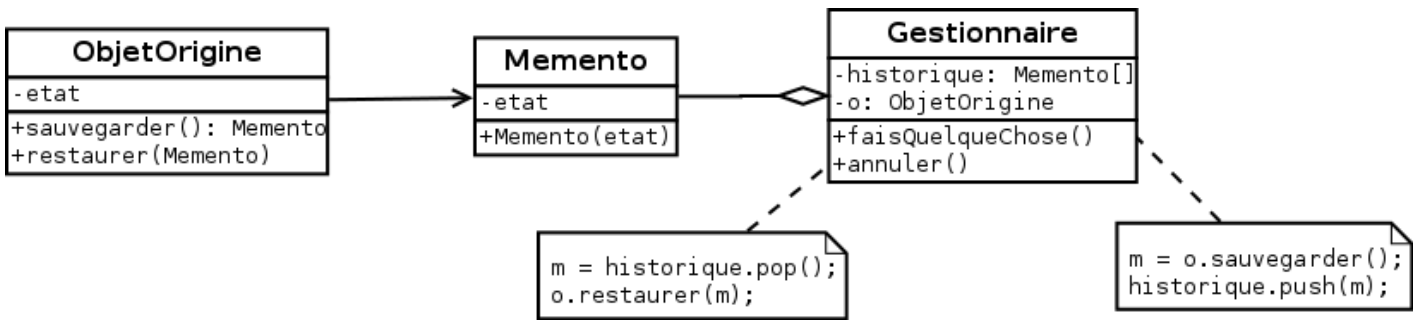
Principe

Le principe consiste à créer un instantané (snapshot) à partir de l'état courant de l'objet. Le client enregistre ces instantanés (par exemple dans un historique) et les restaure à la demande.

Mémento

Objectif

Le patron Memento a pour objectif de permettre à un objet de sauvegarder et restaurer son état interne, sans pour autant exposer son modèle de données à l'extérieur de l'objet.



Commande

Définition

Commande est un patron de conception qui encapsule la notion d'invocation. Il permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même.

Utilité

Ce patron de conception est souvent utilisé dans les interfaces graphiques où, par exemple, un item de menu peut être connecté à différentes Commandes de façon à ce que l'objet d'item de menu n'ait pas besoin de connaître les détails de l'action effectuée par la Commande.

Commande

Définition

Commande est un patron de conception qui encapsule la notion d'invocation. Il permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même.

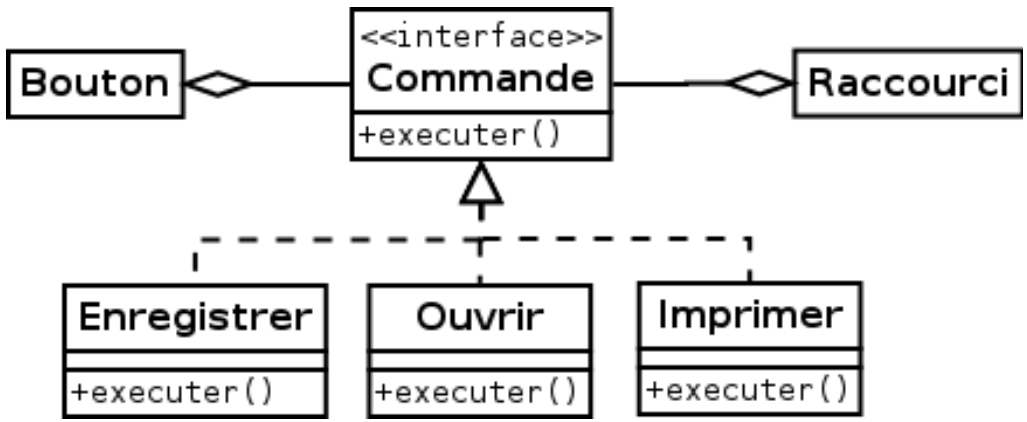
Principes

Un objet Commande sert à communiquer une action à effectuer, ainsi que les arguments requis. L'objet est envoyé à une seule méthode dans une classe, qui traite les Commandes du type requis. L'objet est libre d'implémenter le traitement de la Commande par un switch, ou un appel à d'autres méthodes (en particulier, polymorphes).

Commande

Définition

Commande est un patron de conception qui encapsule la notion d'invocation. Il permet de séparer complètement le code initiateur de l'action, du code de l'action elle-même.



Commande

Exemple de commande : un interrupteur

```
public class Switch {

  private Command flipUpCommand;
  private Command flipDownCommand;

  public Switch(Command fUp, Command fDown) {
    this.flipUpCommand=fUp;
    this.flipDownCommand=fDown;
  }

  public void flipUp() {
    flipUpCommand.execute();
  }

  public void flipDown() {
    flipDownCommand.execute();
  }
}
```

```
public class Light {

  public Light()

  public void turnOn() {
    System.out.println("The light is on");
  }

  public void turnOff() {
    System.out.println("The light is off");
  }
}
```

Commande

Exemple de commande : un interrupteur

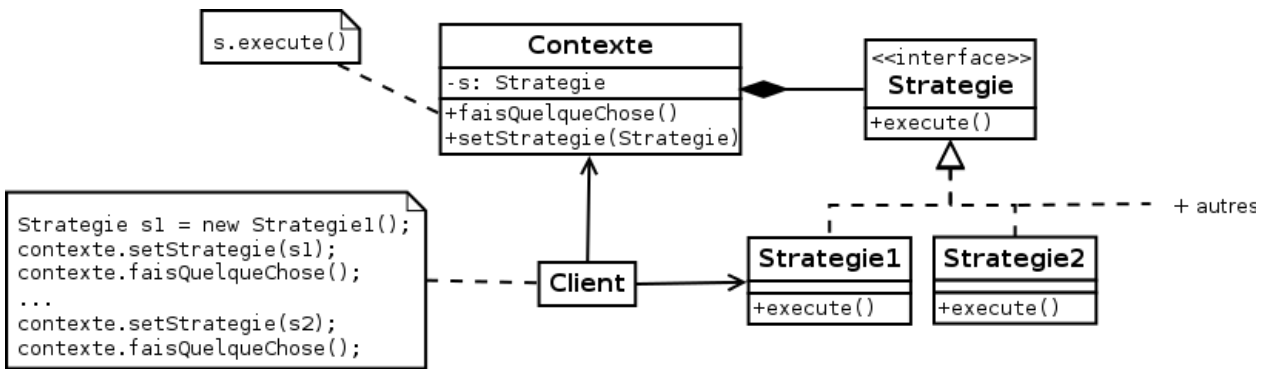
```
public interface Command {
  void execute();
}

public class TurnOnCommand implements Command {

  private Light theLight;

  public TurnOnCommand(Light light) {
    this.theLight=light;
  }

  public void execute() {
    theLight.turnOn();
  }
}
```

```
public class TurnOffCommand implements Command {

  private Light theLight;

  public TurnOffCommand(Light light) {
    this.theLight=light;
  }

  public void execute() {
    theLight.turnOff();
  }
}


public class TestCommand {
  public static void main(String[] args) {
    Light lamp = new Light();
    Command switchUp=new TurnOnCommand(lamp);
    Command switchDown=new TurnOffCommand(lamp);
    Switch s=new Switch(switchUp,switchDown);
    s.flipUp();
    s.flipDown();
  }
}
```

Stratégie

Définition

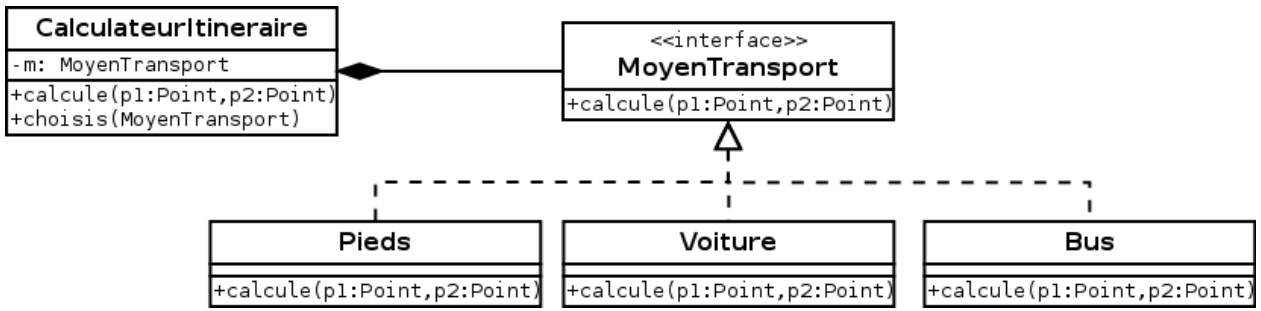
Le patron de conception *stratégie* est utilisé pour des applications ayant besoin de changer dynamiquement des algorithmes employés pour réaliser un traitement. On définit ainsi une famille d'algorithmes, chacun encapsulé dans un objet, qui vont pouvoir être interchangés à la demande.
Chaque objet implantant une stratégie est indépendant des autres.



Stratégie

Exemple de stratégie

On souhaite réaliser un calcul d'itinéraire entre deux endroits (points GPS). Suivant les moyens de transports que l'on utilise (à pieds, en voiture, en train), différents calculs et résultats sont possibles.



Cette construction évite un `if` ou un `switch` dans le code du client pour décider quel traitement effectuer. Par ailleurs, rajouter un moyen de transport (ex. itinéraire en vélo) est relativement aisé.

Etat

Définition

Le patron *Etat* (ou State) permet à un objet de changer son comportement lorsque son état interne change. Bien que l'objet présente toujours les mêmes méthodes, celles-ci présentent une exécution différente en fonction de l'état dans lequel l'objet se trouve.
Ce patron est utilisé pour implanter des machines à états (diagrammes d'états-transitions ou autres automates).

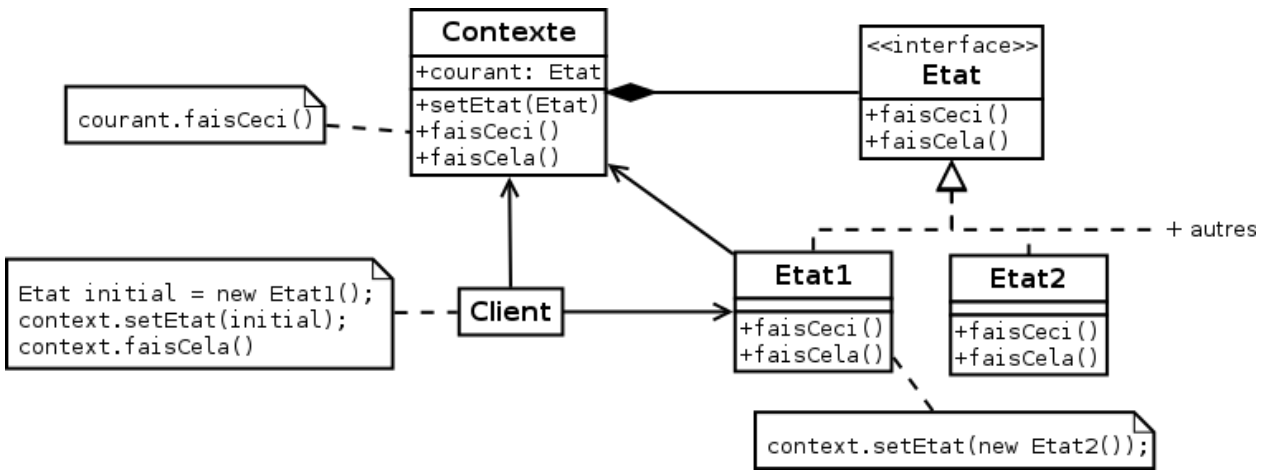
Principe

Plutôt que d'avoir, dans chaque méthode de l'objet considéré des alternatives qui définissent le comportement de la méthode en fonction de l'état, on encapsule chaque état dans un objet avec son comportement associé. L'objet considéré délèguera alors l'appel de méthode à l'état courant.
On notera que chaque état peut lui-même faire évoluer l'état courant de l'objet (à la différence de la Stratégie où les traitements délégués n'ont pas accès/connaissance de leur contexte).

Etat

Définition

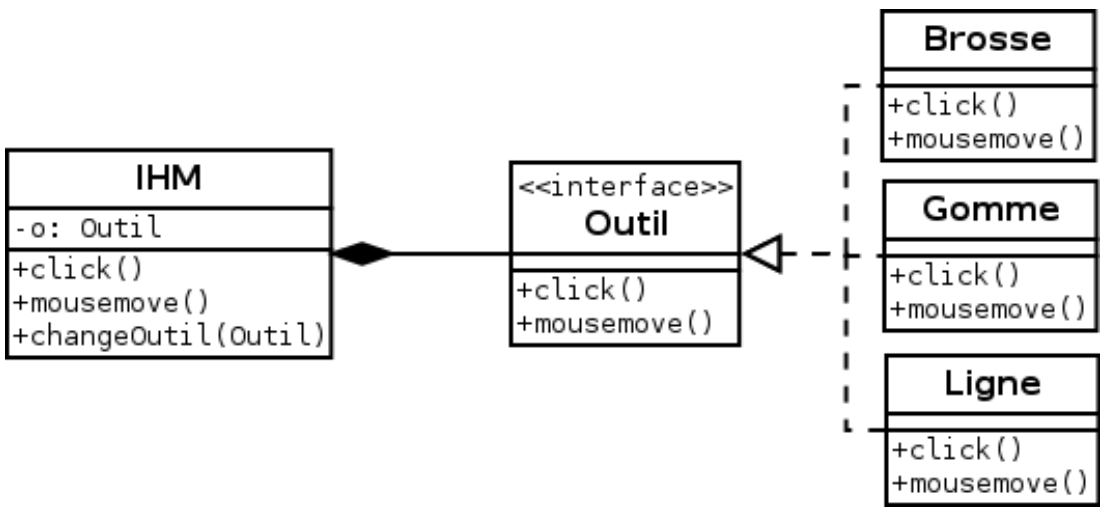
Le patron *Etat* (ou State) permet à un objet de changer son comportement lorsque son état interne change. Bien que l'objet présente toujours les mêmes méthodes, celles-ci présentent une exécution différente en fonction de l'état dans lequel l'objet se trouve.
Ce patron est utilisé pour implanter des machines à états (diagrammes d'états-transitions ou autres automates).



Etat

Exemple de patron Etat

On souhaite réaliser un logiciel de dessin qui se compose de différentes fonctionnalités : brosse, gomme, tracé de ligne, etc. Pour une même action sur la zone de dessin (clic, déplacement de la souris, etc.), le résultat sera différent suivant l'outil actuellement sélectionné.
On peut voir chaque outil sélectionné comme un état particulier de l'application. Le client transmet au contrôleur l'action qui est réalisée (clic, survol, ...) et celui-ci le délègue à l'état courant.



Itérateur

Définition

Un *itérateur* est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc.). Un synonyme d'itérateur est curseur, notamment dans le contexte des bases de données.

Utilité

Le but d'un itérateur est de permettre à son utilisateur de parcourir le conteneur, c'est-à-dire d'accéder séquentiellement à tous ses éléments pour leur appliquer un traitement, tout en isolant l'utilisateur de la structure interne du conteneur, potentiellement complexe.

Itérateur

Définition

Un *itérateur* est un objet qui permet de parcourir tous les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, etc.). Un synonyme d'itérateur est curseur, notamment dans le contexte des bases de données.

Principe

Un itérateur fournit au minimum les primitives suivantes :

- accéder à l'élément courant
- se déplacer vers l'élément suivant

L'itérateur, à sa création, doit pointer sur le premier élément. Il peut également fournir une fonction permettant de déterminer si la totalité des éléments du conteneur ont été modifiés.

Itérateur

Exemple d'itérateur sur un arbre

```
class Depth1stIterator implements Iterator<Node>
{

  Node root, current;

  public Depth1stIterator(Node r) {
    root = r;
    current = null;
  }

  public void reset() {
    current = null;
  }

  public boolean hasNext() {
    return getNext() != null;
  }

  public Node next() {
    return current = getNext();
  }

  public void remove() { }

  private Node getNext() {
    Node current = this.current;
    if (current == null) {
      return root;
    }
    if (current.getNumChildren() > 0) {
      return current.getChild(0);
    }
    do {
      Node parent = current.getParent();
      int i=0;
      while (parent.getChild(i) != current) {
        i++;
      }
      if (i < parent.getNumChildren()-1) {
        return parent.getChild(i+1);
      }
      current = parent;
    }
    while (current != root);
    return null;
  }

}
```

Itérateur

Exemple d'itérateur sur un arbre

```
class Tree implements Iterable<Node> {

  Node root;

  public Tree(Node r) {
    root = r;
  }

  public Iterator<Node> iterator() {
    return new Depth1stIterator(root);
  }
}

class Node {

  Object value;
  ArrayList<Node> children;
  Node parent;

  ...

}
```

```
public class TestIterateur {

  public static void main(String[] args) {

    Node n1 = new Node(1);
    Node n2 = new Node(2);
    n1.addChild(n2);
    n1.addChild(new Node(3));
    n2.addChild(new Node(4));
    Node n3 = new Node(5);
    n3.addChild(new Node(6));
    n2.addChild(n3);
    n1.addChild(new Node(7));

    Tree t = new Tree(n1);
    Iterator<Node> it = t.iterator();
    while (it.hasNext()) {
      System.out.println(it.next().getValue());
    }
    // affiche : 1 2 4 5 6 3 7 - tout comme :
    for (Node n : t)) {
      System.out.println(n.getValue());
    }
  }
}
```

Observateur

Définition

Le patron de conception *observateur/observable* est utilisé pour envoyer un signal à des modules qui jouent le rôle d'observateur. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les "observables").

Utilité

Les observateurs ont pour principale fonction de gérer les événements et les actions effectuées en réponse à ces événements.

Observateur

Exemple d'observateur

```
import java.util.Observable;
import java.util.Observer;

class Person extends Observable {

  int age;

  public Person() {
    age = 0;
  }

  public int getAge() {
  return age;
  }

  public void retraite() {
    System.out.println("Enfin la quille !!!");
  }

  public void birthday() {
    age++;
    System.out.println("J'ai "+age+" ans");
    setChanged();
    notifyObservers();
  }
}
```

```
class PersonObserver implements Observer {

  public void update(Observable o, Object args) {

    if (o instanceof Person) {
      if (((Person)o).getAge() == 60) {
        ((Person)o).retraite();
      }
    }
  }
}


public class Program {

  public static void main(String[] args) {
    Person p = new Person();
    p.addObserver(new PersonObserver());
    for (int i=1; i < 90; i++) {
      p.birthday();
    }
  }
}
```

Médiateur

Définition

Le patron *Médiateur* permet de réduire les dépendances chaotiques entre de nombreux objets qui coexistent au sein de l'application. Plutôt que chacun des objets n'interagisse avec tous les autres, il passe par un objet central (nommé médiateur) qui se charge de capturer toutes les interactions avec les autres objets.

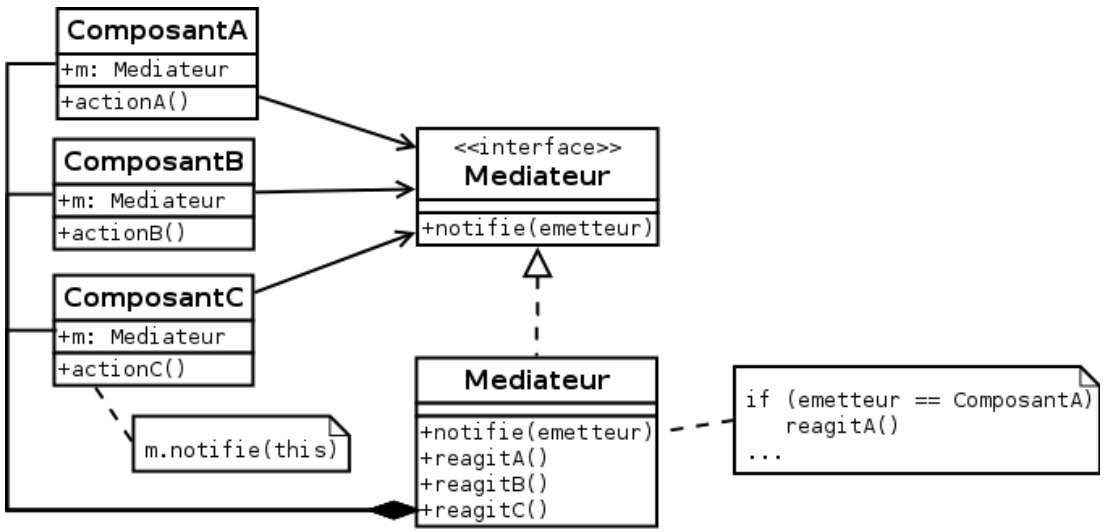
Illustration du besoin d'un médiateur

On dispose d'un formulaire dans une application qui propose des cases à cocher, des boutons radios, des zones de textes, des boutons, etc. On imagine que, pour des considérations d'UX, certains éléments du formulaire vont être masqués/affichés en fonction de certaines sélections (par ex. cocher la case "je veux être prévenu par SMS" affiche une zone de saisie du numéro de téléphone).
$\Rightarrow$ plutôt que de laisser chaque composant diriger tous les autres, on passe par une entité centrale, qui reçoit les notifications de chaque composant et répercute celle-ci sur les autres composants.

Médiateur

Définition

Le patron *Médiateur* permet de réduire les dépendances chaotiques entre de nombreux objets qui coexistent au sein de l'application. Plutôt que chacun des objets n'interagisse avec tous les autres, il passe par un objet central (nommé médiateur) qui se charge de capturer toutes les interactions avec les autres objets.



Patron de méthode

Définition

Un *patron de méthode* définit le squelette d'un algorithme à l'aide d'opérations abstraites dont le comportement concret se trouvera dans les sous-classes, qui implémenteront ces opérations.

Utilité

Cette technique, très répandue dans les classes abstraites, permet de :

- Fixer clairement des comportements standards qui devraient être partagés par toutes les sous-classes, même lorsque le détail des sous-opérations diffère.
- Factoriser du code qui serait redondant s'il se trouvait répété dans chaque sous-classe.

Patron de méthode

Exemple de patron de méthode

```
abstract class JeuDeSociete {
  protected int nombreDeJoueurs;
  abstract void initialiserLeJeu();
  abstract void faireJouer(int joueur);
  abstract boolean partieTerminee();
  abstract void proclamerLeVainqueur();

  // Méthode socle (héritée mais pas redéfinie)
  final void jouerUnePartie(int nbJoueurs) {
    this.nombreDeJoueurs = nbJoueurs;
    initialiserLeJeu();
    int j = 0;
    while (! partieTerminee()) {
      faireJouer(j);
      j = (j + 1) % nombreDeJoueurs;
    }
    proclamerLeVainqueur();
  }
}
```

```
class Monopoly extends JeuDeSociete {
  // attributs spécifiques au jeu du monopoly
  void initialiserLeJeu() { ... }
  void faireJouer(int joueur) { ... }
  boolean partieTerminee() { ... }
  void proclamerLeVainqueur() { ... }
}

class Echecs extends JeuDeSociete {
  // attributs spécifiques au jeu d'échecs
  void initialiserLeJeu() { ... }
  void faireJouer(int joueur) { ... }
  boolean partieTerminee() { ... }
  void proclamerLeVainqueur() { ... }
}
```

Interpréteur

Définition

Le patron de conception *Interpréteur* est utilisé pour des logiciels ayant besoin d'un langage afin de décrire les opérations qu'ils peuvent réaliser (exemple : SQL pour interroger une base de données).

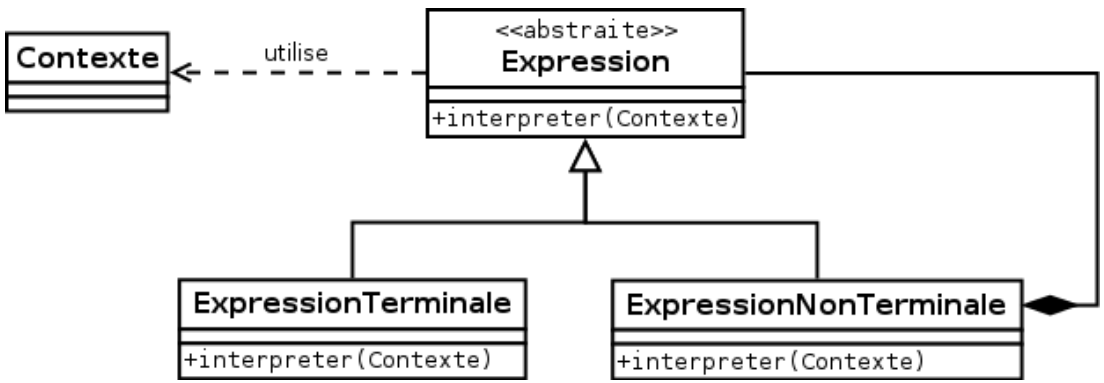
Principes

Le modèle de conception *Interpréteur* définit la grammaire de ce langage et utilise celle-ci pour interpréter des états dans ce langage.

Interpréteur

Mise en place

Ce patron définit comment interpréter les éléments du langage. Dans ce patron de conception, il y a une classe par symbole terminal et non-terminal du langage à interpréter. L'arbre de syntaxe du langage est représenté par une instance du patron de conception *Objet composite*.



Interpréteur

Exemple d'interpréteur d'expression algébrique

```
public interface Expr {
  double interpreter();
}

public class Nombre implements Expr {
  double valeur;
  public Nombre(double v) { valeur = v; }
  public double interpreter() {
    return valeur;
  }
}

public class Multiplie implements Expr {
  Expr op1, op2;
  public Multiplie(Expr e1, Expr e2) {
    op1 = e1;
    op2 = e2;
  }
  public double interpreter() {
    return op1.interpreter() *
           op2.interpreter();
  }
}
```

```
public class Plus implements Expr {
  Expr op1, op2;
  ...
  public double interpreter() {
    return op1.interpreter() +
           op2.interpreter();
  }
}

public class Program {
  public static void main(String[] args) {
    // e == (2 + 4) * 7
    Expr e1 = new Nombre(2.0);
    Expr e2 = new Nombre(4.0);
    Expr e3 = new Nombre(7.0);
    Expr e4 = new Plus(e1,e2);
    Expr e = new Multiplie(e3,e4);
    System.out.println(e.interpreter());
  }
}
```

Visiteur

Définition

Un *visiteur* est une manière de séparer un algorithme d'une structure de données sur laquelle il effectue un traitement. Un visiteur possède une méthode par type d'objet traité.

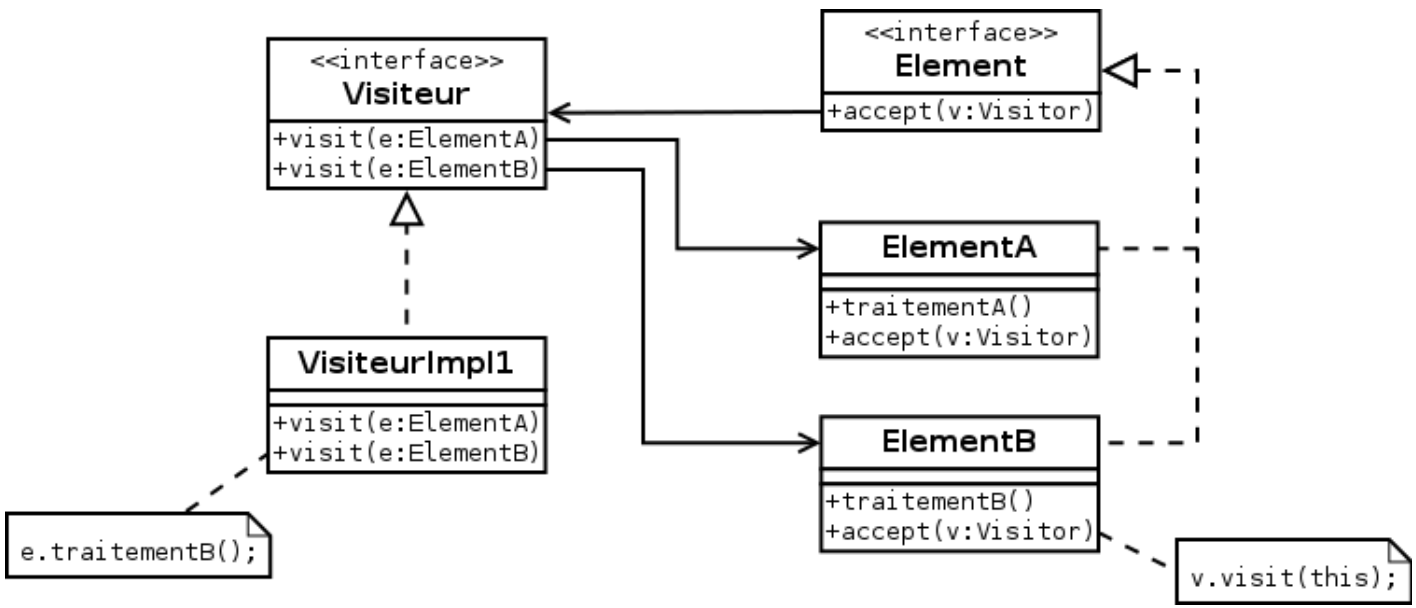
Principe

Le visiteur est une interface qui propose de visiter chaque entité de la structure de données qu'il parcourt. Toutes les entités qui peuvent être parcourues par le visiteur déclarent une méthode `accept(v : Visiteur)` dont le code est toujours le même (`v.visit(this)`).
C'est le visiteur qui implémente le traitement pour chacune des entités, via des méthodes `visit` s'appuyant sur le polymorphisme paramétrique (les méthodes ne diffèrent que par le type du paramètre considéré).

Visiteur

Définition

Un *visiteur* est une manière de séparer un algorithme d'une structure de données sur laquelle il effectue un traitement. Un visiteur possède une méthode par type d'objet traité.



Visiteur

Définition

Un *visiteur* est une manière de séparer un algorithme d'une structure de données sur laquelle il effectue un traitement. Un visiteur possède une méthode par type d'objet traité.

Intérêt

Pour ajouter un traitement, il suffit de créer une nouvelle classe dérivée de la classe Visiteur et on l'applique sur la structure. On n'a donc pas besoin de modifier la structure des objets traités, contrairement à ce qu'il aurait été obligatoire de faire si on avait implémenté les traitements comme des méthodes de ces objets.

Visiteur

Définition

Un *visiteur* est une manière de séparer un algorithme d'une structure de données sur laquelle il effectue un traitement. Un visiteur possède une méthode par type d'objet traité.

Extensions

Pour donner un peu de généricité, on peut donner aux méthodes de visite un paramètre et un type de retour (qui devra être le même que dans les méthodes `accept` des éléments visités).

Visiteur

Exemple d'interpréteur d'expression algébrique

```
public interface class Expr {
  Object accept(Visiteur e);
}

public interface Visiteur {
  Object visit(Nombre n);
  Object visit(Multiplie m);
  Object visit(Plus p);
}

public class Nombre extends Expr {

  double valeur;

  public Nombre(double v) { valeur = v; }

  public double getValeur() { return valeur; }

  public Object accept(Visiteur ev) {
    return ev.visit(this);
  }
}
```

```
public class Multiplie implements Expr {

  Expr op1, op2;

  ...

  public Object accept(Visiteur ev) {
    return ev.visit(this);
  }
}

public class Plus implements Expr {

  Expr op1, op2;

  ...

  public Object accept(Visiteur ev) {
    return ev.visit(this);
  }
}
```

Visiteur

Exemple d'interpréteur d'expression algébrique

```
public class Interpreter implements Visiteur {

  Object visit(Nombre n) {
    return n.getValue();
  }

  Object visit(Multiplie m) {
    return
        (double)m.getOper1().accept(this) *
        (double)m.getOper2().accept(this);
  }

  Object visit(Plus p) {
    return
        (double)p.getOper1().accept(this) +
        (double)p.getOper2().accept(this);
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    // e == (2 + 4) * 7
    Expr e1 = new Nombre(2.0);
    Expr e2 = new Nombre(4.0);
    Expr e3 = new Nombre(7.0);
    Expr e4 = new Plus(e1,e2);
    Expr e = new Multiplie(e3,e4);
    Object o = e.accept(new Interpreter());
    System.out.println(o);
      // affiche : 42
  }
}
```

Visiteur

Exemple traducteur en Scheme d'une expression algébrique

```
public class SchemeTsltr implements Visiteur {

  Object visit(Nombre n) {
    return n.getValue();
  }

  Object visit(Multiplie m) {
    return "( * " +
        (String) m.getOper1().accept(this) +
        " " +
        (String) m.getOper2().accept(this) +
        ")";
  }

  Object visit(Plus p) {
    return "( + " +
        (String) m.getOper1().accept(this) +
        " " +
        (String) m.getOper2().accept(this) +
        ")";
  }
}
```

```
public class Program {
  public static void main(String[] args) {
    // e == (2 + 4) * 7
    Expr e1 = new Nombre(2.0);
    Expr e2 = new Nombre(4.0);
    Expr e3 = new Nombre(7.0);
    Expr e4 = new Plus(e1,e2);
    Expr e = new Multiplie(e3,e4);
    Object o = e.accept(new SchemeTsltr());
    System.out.println(o);
      // affiche (* (+ 2.0 4.0) 7.0)
  }
}
```

Plan du cours

L'architecture MVC

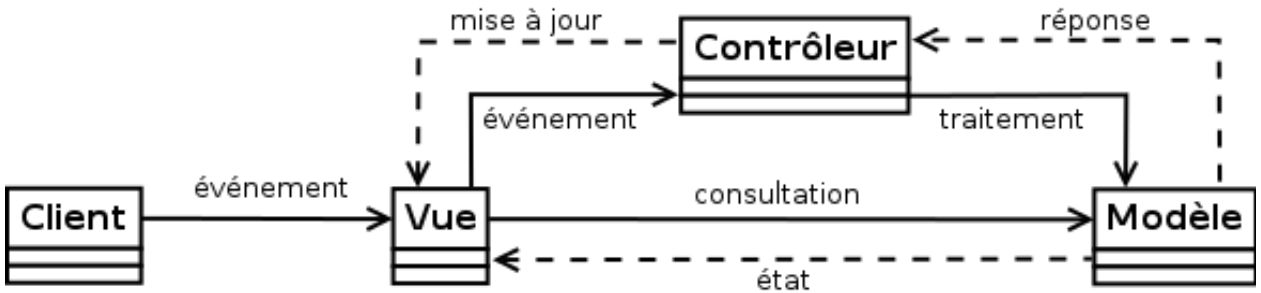
Définition

L'architecture Model-View-Controller (ou Modèle-Vue-Contrôleur) est un modèle d'architecture créé en 1979, et proposant de structurer l'application en 3 couches qui communiquent entre elles.

Objectif

Le patron de conception vise à séparer la présentation des informations (la vue) de leur représentation interne (le modèle). Une tierce partie (le contrôleur) est en charge d'assurer la cohérence entre ces deux couches.
Ce patron est principalement utilisé dans les interface graphiques (Java Swing).

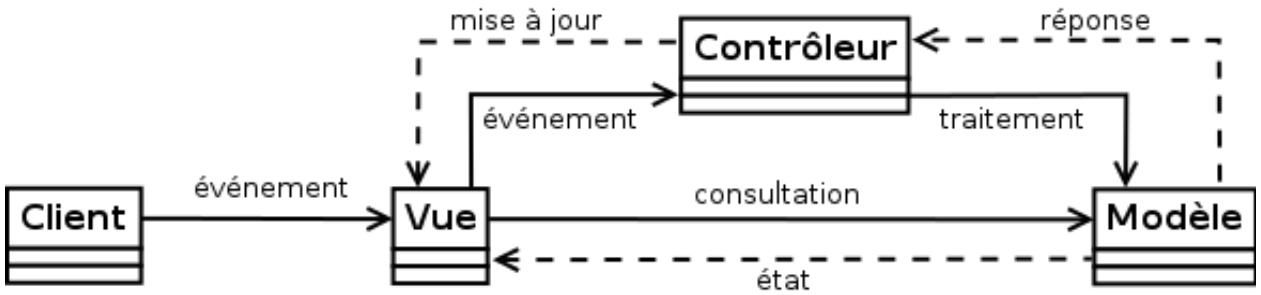
L'architecture MVC



La couche modèle

- C'est la couche métier, le coeur de l'application.
- Elle représente le comportement de l'application : contient les données de l'application, et effectue des traitements sur ces données.
- L'interface du modèle propose de consulter ou de mettre à jour les données, pas de les présenter.

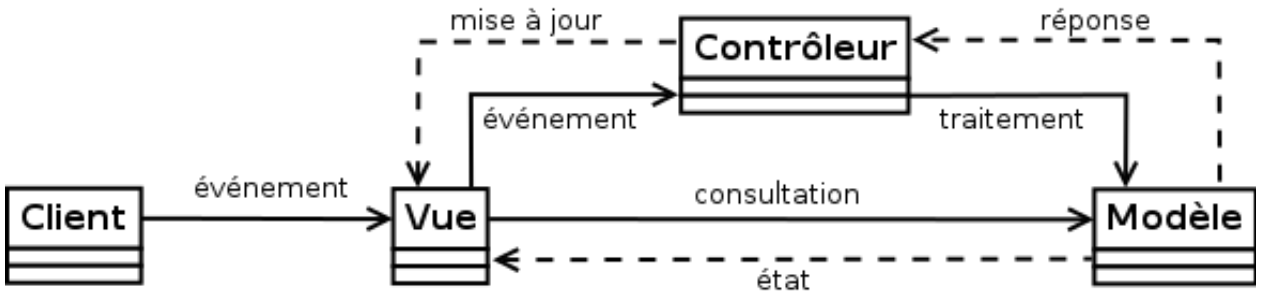
L'architecture MVC



La couche vue

- C'est l'interface avec l'utilisateur
- D'une part, elle est en charge de présenter les résultats du modèle en interrogeant celui-ci (lecture seule).
- D'autre part, elle permet de capturer les actions de l'utilisateur, soit pour mettre à jour la vue (si elle est la seule concernée par l'action), soit pour transmettre une requête spécifique au contrôleur (elle n'effectue aucun traitement sur les données de l'application).
- Remarque : plusieurs vues sont possibles pour les mêmes données.

L'architecture MVC



La couche contrôleur

- C'est elle qui gère les événements de synchronisation.
- Elle reçoit les événements de l'utilisateur, via la vue.
- Elle déclenche les actions à effectuer : transmet des requêtes de mise à jour au modèle, et informe la vue de se mettre à jour.
- Remarque : le contrôleur n'effectue aucun traitement, et ne modifie aucune donnée.

L'architecture MVC

Bilan

- Maintenance facilitée par le découplage vue-modèle
- Combinaison de différents patrons du GoF
- Principalement utilisé pour les interface graphiques

Exemple de MVC : les Swing Java

L'utilisation du MVC est native en Swing. La plupart des composants Swing (hormis les conteneurs) utilisent une classe spécifique pour contenir leurs données.

- Les listes (JList) utilisent un objet de classe ListModel pour les données et un objet de classe ListSelectionModel pour gérer les sélections.
- Les arbres (JTree) utilisent un objet de classe TreeModel pour les données et un objet de classe TreeSelectionModel pour gérer les sélections.
- Les tables (JTable) utilisent des objets de classe TableModel et TableColumnModel pour les données et un objet de classe ListSelectionModel pour gérer les sélections.

Anti-patterns

Il existe également des anti-patrons qui décrivent de mauvaises situations :

- God Object (objet divin - blob) : unique objet qui centralise tous les fonctionnalités essentielles de l'application (un énorme objet qui fait tout + petits objets qui stockent juste de l'information).
- Abstraction inverse : offrir uniquement des services complexes, pas nécessairement utiles, que les développeurs sont obligés d'utiliser pour faire des traitements simples.
- Programmation spaghetti : les objets du systèmes sont tous interdépendants ; il est impossible de modifier une partie du système sans altérer tous ses composants.
- Coulée de lave : mauvaise solution employée tôt dans le développement, mais jamais corrigée, et sur laquelle tout le reste du développement a été construit.
- Marteau doré : utilisation d'une technologie familière de façon obsessionnelle, sans réflexion préalable sur sa réelle utilité (*"quand on a un marteau dans la main, tous les problèmes ressemblent à des clous"*) ;
- Magic Numbers : les célèbres constantes magiques

Une liste exhaustive : `https://wiki.c2.com/?AntiPattern`

Conclusion

La patrons de conception représentent des recettes de cuisines pour résoudre des problèmes récurrents en objet, tout en respectant au maximum les principes de bonne programmation (type SOLID).

Avec l'expérience, vous apprendrez à reconnaître les situations qui appellent plutôt à tel ou tel pattern, et vous saurez les mettre en oeuvre.

Dans la pratique : la conception d'une application évolue sans cesse (en particulier dans les méthodes agiles). Dans le cadre du refactoring, il n'est pas rare de repenser le design pour y intégrer un patron de conception (pour faire "plus propre"). Les patrons de conception sont donc un outil qui vous permettra d'écrire du code maintenable et extensible.

Lien avec les tests : dans le cadre du refactoring (mise à jour du code, iso-fonctionnel), utilisez les tests existants pour assurer la non-régression de votre application.