



Chapitre IV.

Les patrons de conception

Frédéric DADEAU

Département Informatique des Systèmes Complexes - FEMTO-ST

Bureau 410 C

Email : `frederic.dadeau@univ-fcomte.fr`

Modélisation et Conception Orientées Objet
Licence 3 – Année 2016-2017



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.

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”.

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.

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

Singleton

Prototype

Fabrique et Fabrique Abstraite

Monteur

Les patrons de structure

Les patrons de comportement

L'architecture MVC



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.



Les patrons de création

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'exactly 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.



Les patrons de création

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.



Les patrons de création

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).

Les patrons de création



CSingle
-INSTANCE: CSingle
-CSingle()
+getInstance(): CSingle

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;
    }

}
```



Les patrons de création

CSingle
-INSTANCE: CSingle
-CSingle()
+getInstance(): CSingle

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;
    }
}
```



Les patrons de création

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.



Les patrons de création

Utilité

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

Implantation

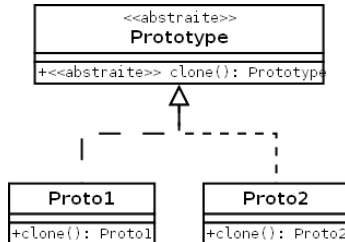
Pour implanter ce patron il faut déclarer une classe abstraite spécifiant une méthode abstraite appelée `clone()`. Toute classe nécessitant un constructeur polymorphique dérivera de cette classe abstraite et implantera la méthode `clone()`. C'est cette méthode qui sera appelée par la classe cliente.

Les patrons de création



Utilité

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





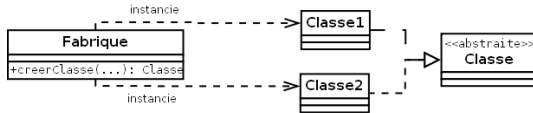
Les patrons de création

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.

Les patrons de création



Implantation d'une fabrique en Java

```

public class FabriqueAnimal {

    public Animal getAnimal(String typeAnimal) throws ExceptionCreation {
        if (typeAnimal.equals("chat"))
            return new Chat();
        else if (typeAnimal.equals("chien"))
            return new Chien();
        else
            throw new ExceptionCreation("Impossible de créer un " + typeAnimal);
    }
}
    
```




Les patrons de création

Définition

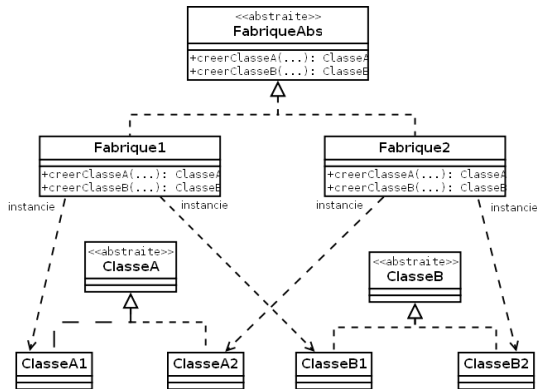
Une *fabrique abstraite* encapsule un groupe de fabriques ayant une thématique commune.

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.

Les patrons de création





Les patrons de création

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");
    }
}

public class Application {
    public static void main(String[] args) {
        GUIFactory aFactory = GUIFactory.getFactory();

        Button aButton = aFactory.createButton();
        aButton.paint();
    }
}
```



Les patrons de création

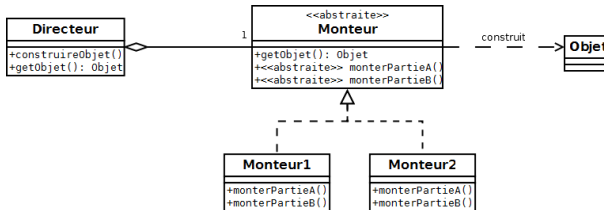
Définition du Monteur

Le *monteur* (également appelé *builder*) est utilisé pour la création d'une variété d'objets complexes à partir d'un objet source. L'objet source 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.

Les patrons de création



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 la classe abstraite Monteur qui ne définissent que la manière de construire ces parties.



Les patrons de création

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 creerPizza() { 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("croisée"); }
    public void monterSauce() {
        p.setSauce("tomate"); }
    public void monterGarniture() {
        pizza.setGarniture("jambon+artichaut"); }
}

class MonteurPizzaThon extends MonteurPizza {
    public void monterPate() {
        pizza.setPate("feuilletée"); }
    public void monterSauce() {
        pizza.setSauce("tomate"); }
    public void monterGarniture() {
        pizza.setGarniture("thon"); }
}
```

Les patrons de création



Exemple de patron monteur

```
class Serveur {
    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();
    }
}
```

```
class ExempleMonteur {
    public static void main(String[] args) {
        Serveur serveur = new Serveur();
        MonteurPizza mp1 = new MonteurPizza4Saisons();

        MonteurPizza mp2 = new MonteurPizzaThon();
        serveur.setMonteurPizza(mp1);
        serveur.construirePizza();
        Pizza pizza = serveur.getPizza();
    }
}
```



Plan du cours

Les patrons de création

Les patrons de structure

Pont

Façade

Adaptateur

Objet composite

Proxy

Décorateur

Les patrons de comportement

L'architecture MVC



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.
- ▶ Adapateur : permet d'adapter une interface existante à une autre interface.
- ▶ Objet composite : permet de manipuler des objets composites à travers la même interface que les éléments dont ils sont constitués.
- ▶ 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).
- ▶ Décorateur : permet d'attacher dynamiquement de nouvelles responsabilités à un objet.



Les patrons de structure

Définition

Le *pont* (également appelé *bridge*) est un patron de conception qui permet de découpler l'interface d'une classe et son implémentation. Ainsi l'interface et l'implémentation peuvent varier séparément.

Utilisation

Le pont est 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).



Les patrons de structure

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.



Les patrons de structure

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) {
        ...
    }
}
```



Les patrons de structure

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.



Les patrons de structure

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;
    }
    ...
}
```



Les patrons de structure

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

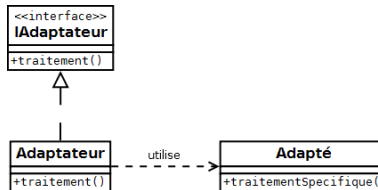
- ▶ Adapter une API tiers qui convient au besoin fonctionnel du client, dont mais la signature des méthodes ne convient pas.
- ▶ Normaliser l'utilisation d'anciennes classes sans pour autant en reprendre tout le code.



Les patrons de structure

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.



Les patrons de structure



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) {" +
            "/* algo bogué */ }";
    }
}

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

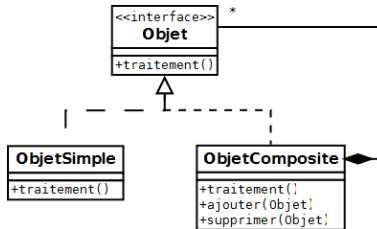
    public static void main(String[] args) {
        Client client = new Client();
        IDeveloppeur dev1 = new DeveloppeurL3();
        client.Utiliser(dev1);

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

Les patrons de structure

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".



Les patrons de structure



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("");
    }
}
```



Les patrons de structure

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.

Les patrons de structure



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
        image3.displayImage();
        // troisième image jamais chargée
    }
}
```



Les patrons de structure

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.



Les patrons de structure

Exemple de décorateur

```
abstract class Voiture {
    public abstract double prix();
}

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

class Option extends Voiture {
    Voiture _originale;
    double tarifOption;

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

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

class Climatatisation extends Option {
    public Climatatisation(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 Climatatisation(martin);
        martin = new Parachute(martin);
        martin = new LanceMissile(martin);
        System.out.println("prix = " + martin.prix());
    }
}
```



Plan du cours

Les patrons de création

Les patrons de structure

Les patrons de comportement

Chaîne de responsabilité

Commande

Interpréteur

Itérateur

Observateur

Patron de méthode

Visiteur

L'architecture MVC



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.
- ▶ Commande : encapsule l'invocation d'une commande.
- ▶ Interpréteur : interpréter un langage spécialisé.
- ▶ Itérateur : parcourir un ensemble d'objets à l'aide d'un curseur.
- ▶ Médiateur : réduire les dépendances entre classes en utilisant un intermédiaire de communication.
- ▶ Memento : mémoriser l'état d'un objet pour pouvoir le restaurer ensuite.
- ▶ ...



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.
- ▶ État : gérer différents états à l'aide de différentes classes.
- ▶ Stratégie : changer dynamiquement de stratégie (algorithme) selon le contexte.
- ▶ Patron de méthode : définir un modèle de méthode avec des méthodes abstraites.
- ▶ Visiteur : découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.



Les patrons de comportement

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. Cela permet de diminuer le couplage entre objets. 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.

Variante

Une variante de ce patron de conception est un arbre de responsabilité, où chaque nœud de traitement transmet l'objet non plus à un seul autre nœud mais à plusieurs nœuds (exemple : un interpréteur de document XML).

Les patrons de comportement



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 chaîne
    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("Sending to stderr: " + m);
    }
}

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

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



Les patrons de comportement

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.



Les patrons de comportement

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).



Les patrons de comportement

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");
    }
}
```



Les patrons de comportement

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();
    }
}
```




Les patrons de comportement

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

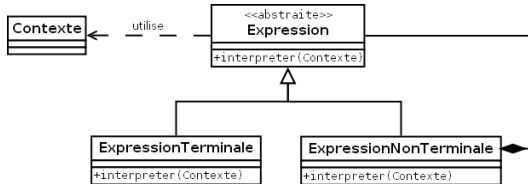
Le modèle de conception *Interpréteur* définit la grammaire de ce langage et utilise celle-ci pour interpréter des états dans ce langage.

Les patrons de comportement



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*.





Les patrons de comportement

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());
    }
}
```



Les patrons de comportement

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.



Les patrons de comportement

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.

Les patrons de comportement



Exemple d'itérateur sur un arbre

```
class Depth1stTreeIterator implements Iterator {
    Node root, current;

    public Depth1stTreeIterator(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;
    }
}
```



Les patrons de comportement

Exemple d'itérateur sur un arbre

```
class Tree {
    Node root;

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

    public Depth1stTreeIterator iterator() {
        return new Depth1stTreeIterator(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);
        Depth1stTreeIterator it = t.iterator();
        while (it.hasNext()) {
            System.out.println(it.next().getValue());
        }
        // affiche : 1 2 4 5 6 3 7
    }
}
```



Les patrons de comportement

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.



Les patrons de comportement

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();
        }
    }
}
```



Les patrons de comportement

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.



Les patrons de comportement

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() { ... }
}
```



Les patrons de comportement

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

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.



Les patrons de comportement

Exemple d'interpréteur d'expression algébrique

```
public abstract class Expr {
    Object accept(Visiteur e, Object data);
}

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

public class Nombre extends Expr {

    double valeur;

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

    public double getValeur() { return valeur; }

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

public class Multiplie implements Expr {

    Expr op1, op2;

    ...

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

public class Plus implements Expr {

    Expr op1, op2;

    ...

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



Les patrons de comportement

Exemple d'interpréteur d'expression algébrique

```
public class Interpreter implements Visiteur {

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

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

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

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(), null);

        System.out.println(o);
        // affiche : 42
    }
}
```



Les patrons de comportement

Exemple traducteur en Scheme d'une expression algébrique

```
public class SchemeTsltr implements Visiteur {

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

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

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

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(), null));

        System.out.println(o);
        // affiche (* (+ 2.0 4.0) 7.0)
    }
}
```



Plan du cours

Les patrons de création

Les patrons de structure

Les patrons de comportement

L'architecture MVC



L'architecture MVC

Définition

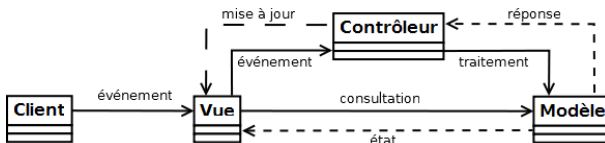
L'architecture Model-View-Controller (ou Modèle-Vue-Contrôleur) est un modèle d'architecture créé en 1979, et proposant de structurer l'application en 3 couches qui communiquent entre elles.

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).

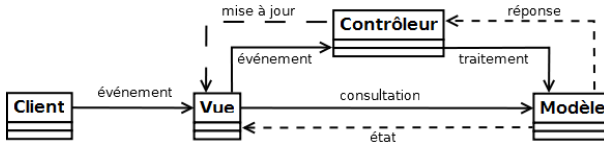
L'architecture MVC



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.

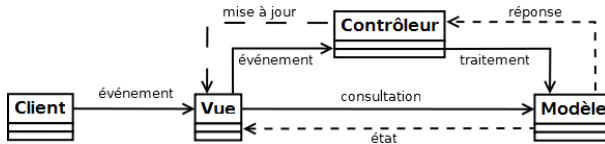
L'architecture MVC



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.