# IFT3912 Développement, Maintenance de Logiciels

# Démo6 : Les Patrons de Conception

Professeur: Bruno Dufor

Démonstrateurs: Marouane Kessentini

Hassen Grati

## I. Définition

Un patron de conception (Design Pattern) est une solution à un problème récurrent dans un contexte donné. On peut considérer un patron de conception comme une formalisation de bonnes pratiques, ce qui signifie qu'on privilégie les solutions éprouvées.

Les patrons de conception suivent un format particulier :
_ Une description du problème avec un exemple concret et une solution qui lui est spécifique
_ Un résumé des choses à considérer pour formuler une solution générale
_ La solution générale
_ Les conséquences positives et négatives de l'utilisation de la solution
_ Une liste des patrons qui sont liés

## II. Différentes catégories de patrons

3 catégories de patrons :
- Création : concerne le processus de création des objets
- Structure : lié à la composition des classes ou objets
- Comportement : caractérise les façons dont les classes ou les objets interagissent ou distribuent les responsabilités.

### Création

- Fabrique abstraite (Abstract Factory)
- Monteur (Builder)
- Fabrique (Factory Method)
- Prototype (Prototype)
- Singleton (Singleton)

### Structure

- Adaptateur (Adapter)
- Pont (Bridge)
- Objet composite (Composite)
- Décorateur (Decorator)
- Façade (Facade)
- Poids-mouche ou poids-plume (Flyweight)
- Proxy (Proxy)

## Comportement

- Chaîne de responsabilité (Chain of responsibility)
- Commande (Command)
- Interpréteur (Interpreter)
- Itérateur (Iterator)
- Médiateur (Mediator)
- Memento (Memento)
- Observateur (Observer)
- État (State)
- Stratégie (Strategy)
- Patron de méthode (Template Method)
- Visiteur (Visitor)

## 1. Singleton

_ Garantit qu'une classe n'a qu'une et une seule instance
_ Tous les objets qui utilisent une instance de cette classe, utilisent la même instance

*Singleton S'assure de l'unicité d'une instance de classe et évidemment le moyen d'accéder à cette instance unique.*

**Exemples :**
Gestion centralisée d'une ressource :
_ Interne : un objet qui est un compteur « global »
_ Externe : un objet qui gère la réutilisation d'une connexion à une base de données
Classes qui ne devraient avoir qu'une seule instance à la fois :
_ Horloge du système
_ Fenêtre principale d'une application
_ Générateur de nombre aléatoire (random number generator)

Code en Java :

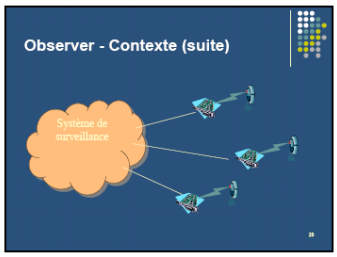
```
public class Singleton {

public static synchronized Singleton getInstance(){

if (instance == null)

instance = new Singleton();

return instance;

}



public void setValue(int value){

i = value;
```

```
}

public int getValue(){
return i;
}

private Singleton(){
System.out.println("Construction du Singleton");
}

private static Singleton instance;
private int i;

public static void main(String[] args) {
Singleton s = Singleton.getInstance();
System.out.println("La valeur de i est " + s.getValue());
Singleton s2 = Singleton.getInstance();
System.out.println("La valeur de i est " + s2.getValue());
// Changement de la valeur du Singleton
s.setValue(20);
System.out.println("La valeur de i est " + s.getValue());
System.out.println("La valeur de i est " + s2.getValue());
}
}
```

## 2. Observateur

Dépendances dynamiques entre objets
_ inscription d'un objet envers un autre
_ objet "observé" notifie les objets "inscrits" de ses changements d'état

**Exemples :**
- Détecteur de fumée




Code en Java

L'exemple ci-dessous montre comment utiliser l'API du langage Java qui propose des interfaces et des objets abstraits liées à ce patron de conception.

- On crée une classe qui étend *java.util.Observable* et dont la méthode de mise à jour des données *setData* lance une notification des observateurs (1) :

```
class Signal extends Observable {

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

On crée le panneau d'affichage qui implémente l'interface *java.util.Observer*. Avec une méthode d'initialisation (2), on lui transmet le signal à observer (2). Lorsque le signal notifie une mise à jour, le panneau est redessiné (3).

```
class JPanelSignal extends JPanel implements Observer {

  void init(Signal lSigAObserver) {
    lSigAObserver.addObserver(this); // (2) ajout d'observateur
  }

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

## 3. Composite

_ Ce patron permet la conception et la construction d'objets complexes en utilisant la composition récursive d'objets similaires représentée par un arbre
_ Ce patron est également connu sous le nom de composition récursive

**Exemples :**

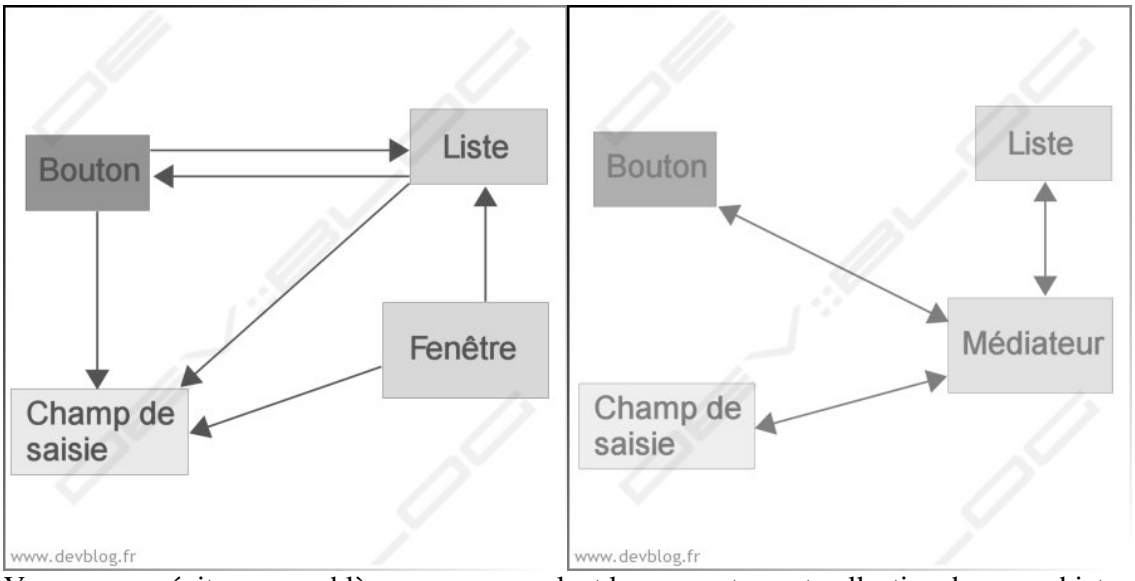
Programme de traitement de texte
_ Mets en forme les caractères en lignes de texte qui sont organisées en colonnes qui sont organisées en pages.
_ Plusieurs liens complexes :
_ Un document contient d'autres éléments
_ Colonnes et pages peuvent contenir des cadres qui eux, peuvent contenir des colonnes
_ Colonnes, cadres et lignes de texte peuvent contenir des images

## 4. Médiateur

Définit un objet qui encapsule comment un ensemble d'objets interagissent. Il est utilisé pour réduire les dépendances entre plusieurs classes.

Au fur et à mesure que l'on ajoute des classes complexes à une application, la communication entre plusieurs classes complexes devient difficile à gérer
_ Le couplage entre les classes devient plus important
_ Ce patron cherche à pallier ce problème avec la conception d'une classe « médiatrice » qui, elle seule, connaît la complexité de chaque classe.
_ Les classes complexes se communiquent entre elles à travers la classe médiatrice.
Lorsqu'une classe désire interagir avec une autre, elle doit passer par le médiateur qui se chargera de transmettre l'information à la ou les classes concernées.

**Exemple :**
Boîte de dialogue dans une interface graphique usager. Une boite de dialogue utilise ne fenêtre pour présenter une collection de composants graphiques tels que des boutons, menus, champs :
Souvent, il existe des dépendances entre ces composants graphiques dans la boîte de dialogue. Par exemple, un bouton est désactivé lorsqu'un certain champ est vide. Sélectionner une entrée dans une liste de choix peut changer le contenu d'un champ dans une zone de texte. Et inversement, taper du texte dans un champ peut automatiquement sélectionner une ou plusieurs entrées dans une liste.




Vous pouvez éviter ces problèmes en encapsulant le comportement collective dans un objet médiateur à part. Un médiateur est responsable de contrôler et coordonner les interactions

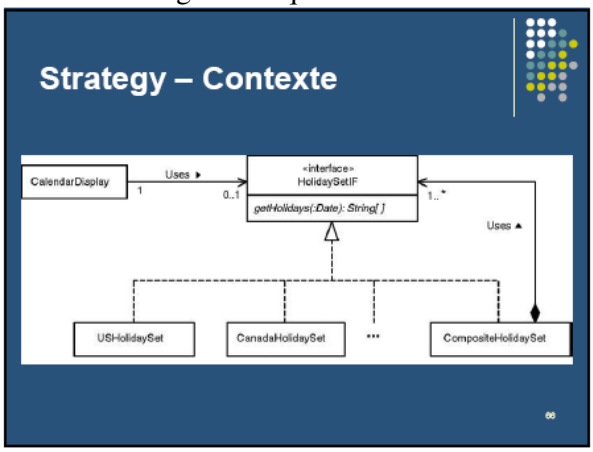
d'un groupe d'objets. Le médiateur sert comme intermédiaire qui garde les objets dans le groupe pour se référer entre eux explicitement. Les objets connaissement seulement le médiateur, ce qui réduit le nombre d'interconnections.

## 5. Stratégie

Définit une famille d'algorithmes, encapsule chacun d'eux, et les rend interchangeable. Statégie permet à l'algorithme de varier indépendamment des clients qui les utilisent. La sélection de l'algorithme peut varier selon l'objet et/ou le temps.

**Exemples :**

- Plusieurs algorithmes existent pour couper un flot de texte en lignes.
- Programme qui affiche un calendrier selon l'usager.




## 6. Pont

Sépare et découple une abstraction et son implémentation permettant à chacune d'évoluer indépendamment.
Le pont est lui utilisé pour découpler l'interface de l'implémentation. Ainsi, vous pouvez modifier ou changer l'implémentation d'une classe sans devoir modifier le code client (si l'interface ne change pas bien entendu).
On désire avoir la possibilité d'ajouter une nouvelle implémentation sans avoir à réimplémenter la logique de l'abstraction.

**Exemples :**
Affichage de fenêtres graphiques spécifiques à des plateformes ex XWindow et PMWindow

## 7. Adaptateur

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

On peut également utiliser un adaptateur lorsque l'on ne veut pas implémenter toutes les méthodes d'une certaine interface. Par exemple, si l'on doit implémenter l'interface MouseListener en Java, mais que l'on ne souhaite pas implémenter de comportement pour

toutes les méthodes, on peut dériver la classe MouseAdapter. Celle-ci fournit en effet un comportement par défaut (vide) pour toutes les méthodes de MouseListener.

Exemple avec le MouseAdapter : <source lang="java">

public class MouseBeeper extends MouseAdapter { public void mouseClicked(MouseEvent e) { Toolkit.getDefaultToolkit().beep(); } }

</source>

Exemple avec le MouseListener : <source lang="java">

public class MouseBeeper implements MouseListener { public void mouseClicked(MouseEvent e) { Toolkit.getDefaultToolkit().beep(); } public void mousePressed(MouseEvent e) { } public void mouseReleased(MouseEvent e) { } public void mouseEntered(MouseEvent e) { } public void mouseExited(MouseEvent e) { } }

## 8. Visiteur

Découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.

L'exemple suivant montre comment afficher un arbre de nœuds (les composants d'une voiture). Au lieu de créer des méthodes d'affichage pour chaque sous-classe (Wheel, Engine, Body, et Car), une seule classe est créée (CarElementPrintVisitor) pour afficher les éléments. Parce que les différentes sous-classes requiert différentes actions pour s'afficher proprement, la classe CarElementPrintVisitor répartit l'action en fonction de la classe de l'argument qu'on lui passe.

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

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

class Wheel implements CarElement
{
    private String name;

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

    String getName()
    {
        return this.name;
```

```
    }

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

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

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

class Car
{
    CarElement[] elements;

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

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

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

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

    public void visit(Body body)
```

```
    {
        System.out.println("Visiting body");
    }

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

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

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

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

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

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

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

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

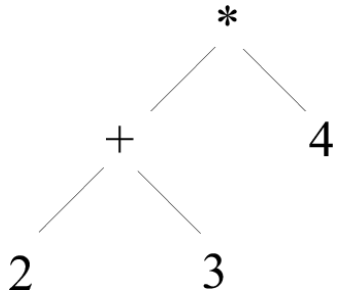
# Exercices :

## Exercice 1 : Patron Composite

Une figure simple peut être un point, une ligne ou un cercle. Une figure peut être composée d'autres figures, simples ou elles-mêmes composées d'autres figures. Toutes les figures peuvent être dessinées ou translatées.
**Question** : Utilisez le pattern composite pour construire un diagramme de classe rendant compte de cette hiérarchie d'objets.

## Exercice 2 : Patron Composite

Une expression arithmétique peut se représenter de manière arborescente. Par exemple, l'expression (2+3)*4 peut se représenter comme le résultat de l'opération * appliquée à 4 et au résultat d'une seconde opération + appliquée à 2 et à 3. 4 et +(2,3) sont dits opérandes de l'expression qui a * comme opérateur. Selon ce principe, une autre notation pour cette expression arithmétique est *(+(2,3),4), aussi appelée notation _ polonaise _et notamment utilisée sur les calculatrice HP. Cette notation peut se représenter de avec un arbre comme suit. Il est à noter qu' il devient ainsi possible de se passer tout à fait de parenthèses : il n'y a aucune ambiguïté lorsqu'on utilise la notation *+234.




Il y a deux types d'expressions : les expressions binaires comme +(2,3) et les expressions unaires qui utilisent des opérations ne prenant qu'un seul argument, comme par exemple l'opérateur de changement de signe dans l'expression. Ces deux types d'expression sont caractérisées par un opérateur et disposent d'une opération calculerValeur(). Le terme est un concept plus général qu'une expression : il peut être soit une valeur constante (1,2, 3 ou 4) soit une expression comme +(2,3). Dans tous les cas, un terme doit disposer d'une opération calculerValeur(). Un autre type de terme peut être une variable qui, en plus d'une valeur comme pour les constantes, dispose d'un nom.

**Question** : Utilisez le pattern composite pour produire un diagramme de classes adéquat pour la représentation des expressions arithmétiques.

## Exercice 3 : Patron Adapter

Un éditeur de jeux développe un jeu permettant aux enfants de connaître les animaux. Les enfants peuvent, en particulier, apprendre la forme et le cri des animaux parmi lesquels le chat et la vache. Le chat est modélisé par la classe LeChat possédant au moins les deux méthodes formeChat() et criChat() et la vache est modélisée par la classe LaVache possédant les deux méthodes criVache() et formeVache(). Comme le montrent les noms des méthodes, la première spécification de ce jeu est propre aux animaux modélisés. L'éditeur souhaite

améliorer ce jeu en créant une interface commune à tous les animaux qui lui permette d'en ajouter de nouveaux, sans modifier l'interface avec le code client, et d'utiliser le polymorphisme dans la gestion des animaux (manipuler des troupeaux hétéroclites...).
**Question** : Proposez une modélisation des classes pour cette nouvelle version du jeu en faisant apparaître le client. Les classes seront Chat, Vache... et non plus LeChat, LaVache...

On souhaite réutiliser tout le code développé dans la version précédente (avec LeChat, LaVache...) dans le nouveau logiciel utilisant les nouvelles classes (Chat, Vache...).
**Question** : Proposez une modélisation permettant d'incorporer les anciennes méthodes pour éviter de les récrire. Vous pourrez utiliser le patron de conception adapter

## Exercice 4 : Patron Visitor

Une figure géométrique peut être un cercle ou un rectangle. On souhaite proposer aux clients (au sens informatique) un outil de dessin des figures géométriques qui s'adapte à l'environnement informatique.
Les interfaces, la qualité des dessins dépendent de l'environnement. Le client est associé aux figures géométriques. Son objectif est de les dessiner. La figure géométrique est composée d'un ensemble de propriétés, dont celles liées au dessin (trait, couleur, etc.). Ces propriétés dépendent de l'environnement (elles sont spécialisées).
**Question** : En utilisant le patron visiteur, proposez une modélisation qui isole le dessin d'une figure géométrique et qui la spécialise en fonction de l'environnement.