Qu'est-ce qu'un design pattern (patron de conception) ?
Une définition : Un design pattern décrit une solution à un problème général et récurrent de conception dans un contexte particulier.
Ce N'est PAS : Une classe individuelle ou bibliothèque, telle que les listes ou les tables d'association | Une conception complète et concrète, ni une implémentation, mais plutôt une description abstraite sur comment résoudre un problème

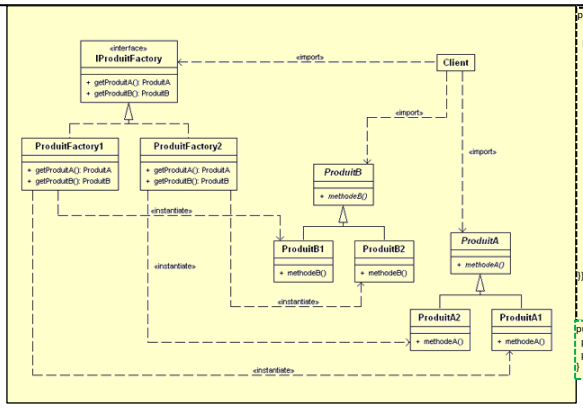
Classification des design patterns (GoF)
But : ce que fait le pattern
Patterns de création : Rendent un système indépendant de comment ses objets sont créés, initialisés et configurés | Sont utiles lorsque le système évolue : les classes qui seront utilisées dans le futur peuvent ne pas être connues maintenant
Patterns de structure : Permettent de composer des classes et des objets afin de former des structures plus importantes | Réduisent le couplage entre des classes (découplage interface et implémentation)
Patterns de comportement : S'intéressent aux interactions entre les objets | Décrivent des flots de contrôle complexes

CREATION
Patterns de création Singleton :
Intention : Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance
Utilisation : Il ne doit y avoir exactement qu'une seule instance d'une classe | Cette instance unique doit être facilement accessible aux clients | Cette instance unique doit être extensible par héritage, et les clients doivent pouvoir utiliser cette instance étendue sans modifier leur code.
Avantages : Permet un accès contrôlé à une unique instance | Permet la réduction de l'espace des noms (vs variables globales) | Peut être étendu par héritage | Permet un nombre variable d'instances (toutes invisibles aux clients) | Est plus flexible que les méthodes de classe
Pattern Fabrique
Intention : Définit une interface pour la création d'un objet, mais le choix de la classe concrète de l'objet est délégué à des sous-classes
Utilisation : Une classe ne peut pas anticiper la classe des objets qu'elle doit créer | Une classe attend de ses sous-classes qu'elles spécifient les objets qu'elle crée
Avantages : En évitant de spécifier le nom de la classe concrète et les détails de son instanciation, le code client devient plus flexible et réutilisable | Le client est uniquement dépendant de l'interface Product et peut fonctionner avec n'importe quelle classe ConcreteProcduct qui implémente cette interface.
Inconvénients : Le client peut avoir à créer une sous-classe de la classe Creator, juste pour instancier un ConcreteProduct particulier
Pattern Fabrique Abstraite
Intention : Fournit une interface pour créer des familles d'objets liés sans spécifier leurs classes concrètes
Utilisation : Un système doit être indépendant de comment ses produits sont créés, composés et représentés | Un système doit être configuré avec l'une des multiples familles de produits | Une famille de produits est conçue pour être utilisée ensemble et il est nécessaire de respecter cette contrainte
Avantages : Isole les clients des classes concrètes, car ils manipulent seulement des interfaces abstraites | Rend facile le changement des familles de produits, car une fabrique concrète particulière supporte une famille complète de produits | Favorise la cohérence entre produits | Impose l'utilisation de produits d'une seule et même famille
Inconvénients : Difficiile d'ajouter de nouveaux produits, car requiert de changer l'interface AbstractFactory et donc d'étendre l'interface de toutes les classes de fabrique concrète

STRUCTURE
Pattern Composite
Intention : Compose des objets dans des structures arborescentes pour représenter des hiérarchies composants/composés et permet aux clients de traiter les objets individuels et leurs compositions de manière uniforme
Utilisation : Un système doit représenter des hiérarchies composants/composés | Les clients ne doivent pas être capable de faire la différence entre les objets individuels et leurs compositions (uniformité apparente)
Avantages : Facilite l'ajout de nouveaux types de composants | Simplifie le code client, car il n'a pas à savoir (et ne devrait pas se soucier de) s'il traite avec une feuille ou un composite
Inconvénients : Est difficile de restreindre et vérifier le type des composants d'un composite (e.g., ne doit contenir que certains types de composant)
Pattern Decorateur
Intention : Attache dynamiquement des responsabilités supplémentaires à objet, fournissant une alternative flexible à l'héritage pour étendre des fonctionnalités
Utilisation : Un système doit ajouter des responsabilités à des objets individuels de manière dynamique et transparente (i.e : sans affecter d'autres objets) | Ces responsabilités peuvent être retirées | L'extension par héritage n'est pas pratique (explosion du nombre de sous-classes pour supporter toutes les combinaisons possibles) ou n'est pas possible
Avantages : Offre plus de flexibilité que l'héritage (statique) | Évite de surcharger de fonctionnalités les classes situées en haut des hiérarchies (classes monolithiques)
Inconvénients : Casse l'identité de l'objet (Un décorateur et son composant ne sont pas identiques) | Conduit à des systèmes composés de beaucoup de petits objets
Pattern Procurateur (proxy)
Intention : Fournit un substitut (le proxy) à un autre objet afin d'en contrôler les accès (i.e., les opérations qui lui sont appliquées)
Utilisation : Un client ne fait pas ou ne peut pas faire faire référence à un objet directement, mais veut tout de même interagir avec l'objet | Il y a besoin d'un référencement plus sophistiqué qu'un simple pointeur : Fournir un représentant local d'un objet distant (remote proxy) | Créer ou charger des objets coûteux à la demande (virtual proxy) | Différer la copie d'un objet coûteux jusqu'à ce que ce dernier soit modifié (copy-on-write proxy) | Contrôler l'accès à un objet (protection/access proxy) | Effectuer des opérations supplémentaires lors de l'accès à un objet (smart reference), telles que compter le nombre de références à l'objet (smart pointer) ou encore charger en mémoire un objet persistant quand il est référencé pour la première fois (lazy instantiation)
Avantages : Introduit un niveau d'indirection lors de l'accès à un objet : Pour cacher au client le fait que l'objet réside dans un autre espace d'adressage (distribution) | Pour effectuer des optimisations transparentes pour le client | Pour vérifier si l'appelant a les permissions requises pour effectuer la requête
Inconvénients : Le proxy est simplement une réplique exacte de son sujet réel
Pattern Facade
Intention : Fournit une interface unifiée et de haut niveau à un ensemble d'interfaces d'un sous-système (i.e., un groupe de classes), afin de le rendre plus facile à utiliser
Utilisation : L'interface exposée par les classes d'un sous-système est complexe : Fournir une interface simple et unique à la plupart des clients | Les autres peuvent regarder derrière la façade | | Il y a beaucoup de dépendances entre les classes d'un sous-système et ses clients : Découpler l'implémentation d'un sous-système de ses clients | Promouvoir l'indépendance et la portabilité d'un sous-système | | Les sous-systèmes sont organisés en couches : Définir une façade par point d'entrée dans chaque couche
Avantages : Cache l'implémentation du sous-système aux clients, le rendant plus facile à utiliser | Favorise un couplage faible entre le sous-système et ses clients : Permet de modifier les classes du sous-système sans affecter les clients | | N'empêche pas les clients d'accéder aux classes sous-jacentes : Un client peut utiliser la façade ou le sous-système directement
Inconvénients : Possible perte de fonctionnalités des classes interfacées selon la manière dont la façade est réalisée | N'empêche pas les clients d'accéder aux classes sous-jacentes !
Pattern Adaptateur
Intention : Convertit l'interface d'une classe en une autre interface attendue par les clients et permet ainsi de faire collaborer des classes même si elles ont des interfaces incompatibles
Utilisation : Un système doit utiliser une classe existante dont l'interface ne correspond pas à ses besoins | Il faut créer une classe réutilisable qui collabore avec des classes non liées et encore inconnues | Il y a besoin d'utiliser plusieurs sous-classes existantes dont l'adaptation des interfaces est impossible par héritage
Adaptateur vs Facade : Un adaptateur réutilise une ancienne interface alors qu'une façade définit une nouvelle interface | Un adaptateur fait travailler ensemble deux interfaces existantes, au lieu d'en définir une entièrement nouvelle

COMPORTEMENT
Pattern Fabrique Itérateur
Intention : Fournit un moyen de parcourir séquentiellement un agrégat (collection, conteneur) d'éléments sans connaître sa structure interne
Utilisation : Il faut accéder au contenu (i.e., les éléments) d'un agrégat sans révéler sa représentation interne : Les itérateurs sont souvent utilisés pour traverser récursivement des structures composites | Il faut gérer plusieurs parcours simultanés d'un agrégat | Il faut offrir une interface uniforme pour parcourir différents agrégats (i.e., itération polymorphe)
Avantages : Simplifie l'interface de l'agrégat, en ne la polluant pas de méthodes relatives à son parcours | Permet de gérer plusieurs parcours simultanés sur un agrégat | Permet de modifier l'algorithme de parcours d'un agrégat : En remplaçant l'instance de l'itérateur par une autre différente | En définissant une nouvelle sous-classe de Iterator
Pattern Fabrique Visiteur
Intention : Représente une opération à effectuer sur les éléments d'une structure et permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère
Utilisation : Une structure d'éléments contient beaucoup de classes avec des interfaces différentes et les opérations à effectuer sur ces éléments dépendent de leur classe concrète | Beaucoup d'opérations indépendantes doivent être effectuées sur les éléments d'une structure et il faut éviter de polluer leur classe avec ces opérations | Les classes définissant la structure d'éléments changent rarement, mais il faut souvent définir de nouvelles opérations sur cette structure
Avantages : Facilite l'ajout de nouvelles opérations | Regroupe les opérations communes dans Visitor et sépare celles indépendantes dans leur propre sous-classe ConcreteVisitor : Simplifie à la fois les classes définissant les éléments et les algorithmes définis dans les visiteurs (toutes les structures de données spécifiques à l'algorithme peuvent être cachées dans le visiteur) | Peut accumuler un état, plutôt que de le passer comme un paramètre supplémentaire à l'opération de visite | Permet de parcourir une structure composée d'éléments de types différents (contrairement à un itérateur !)
Inconvénients : Rend difficile l'ajout de nouvelles classes ConcreteElement : Entraîne l'ajout d'une nouvelle opération abstraite dans Visitor + une implémentation correspondante dans chaque classe ConcreteVisitor | L'interface de ConcreteElement doit être assez riche pour permettre au visiteur de faire son travail : Force à fournir des opérations publiques qui accèdent à l'état interne d'un élément, ce qui peut casser son encapsulation
Pattern Fabrique Etat
Intention : Permet à un objet de modifier son comportement lorsque son état interne change
Utilisation : Le comportement d'un objet dépend de son état et il doit changer son comportement à l'exécution en fonction de cet état | Des opérations ont plusieurs grandes parties avec des conditionnelles qui dépendent de l'état de l'objet
Avantages : Sépare les comportements relatifs à chaque état : Ils sont placés dans un objet (i.e., une sous-classe de State) | | Facilite l'ajout et la suppression des états et des transitions | Élimine les conditionnelles | Rend les transitions entre états plus explicites
Inconvénients : Augmente le nombre d'objets
Pattern Fabrique Stratégie
Intention : Définit une famille d'algorithmes, encapsule chacun d'eux, et les rend interchangeables, tout en leur permettant d'évoluer indépendamment des clients qui les utilisent
Utilisation : De nombreuses classes liées ne diffèrent que par leur comportement | Les clients ont besoin de différentes variantes d'un même algorithme à différents moments (e.g., différentes complexités en temps/mémoire) | Un algorithme utilise des structures de données complexes que les clients n'ont pas à connaître | Une classe définit plusieurs comportements qui figurent sous la forme de conditionnelles dans ses opérations
Avantages : Fournit une alternative à l'héritage de la classe Context pour obtenir une variété d'algorithmes ou de comportements | Sépare l'implémentation de l'algorithme de celle du contexte : Chacune peut être modifiée indépendamment l'une de l'autre | L'algorithme peut être changé dynamiquement (i.e., à l'exécution) | La compréhension et l'extension des algorithmes sont facilitées | | Élimine les conditionnelles pour sélectionner le bon comportement | Fournit différentes implémentations du même comportement : Les clients peuvent choisir parmi des stratégies avec des compromis différents en temps/mémoire
Inconvénients : Augmente le nombre d'objets | Nécessite que les clients connaissent les différentes stratégies disponibles et comprennent en quoi elles diffèrent avant de pouvoir choisir la plus appropriée | Tous les algorithmes doivent utiliser la même interface Strategy : Pas nécessaire pour toutes les implémentations de Strategy | Surcoût lié à la communication entre Strategy et Context
Pattern Fabrique Commande
Intention : Encapsule une requête comme un objet, permettant ainsi de la manipuler de différentes manières
Utilisation : Des objets doivent être paramétrés par une action à effectuer : Fonction de rappel (callback) : enregistrée à un moment donné pour être appelée ultérieurement | Version objet des callbacks | | Un système doit permettre de spécifier, mettre en de l'attente et exécuter des requêtes à différents moments | Un système doit pouvoir défaire (undo) des traitements | Un système doit mémoriser ses changements afin de pouvoir les refaire après un éventuel crash : Enregistrement persistant des modifications | |Un système doit être structuré autour d'opérations de haut niveau construites à partir de primitives : Transactions
Avantages : Découple l'objet qui invoque l'opération (i.e., l'invocateur) de celui qui sait comment la réaliser (i.e., le récepteur) | Fait les commandes des objets de première classe : Elles peuvent être manipulées et étendues comme tout objet | | Permet de grouper des commandes dans une commande composite (i.e., une macro-commande) | Facilite l'ajout de nouvelles commandes sans modifier le code existant
Pattern Fabrique Observateur
Intention : Définit une interdépendance de type un à plusieurs, de telle sorte que si un objet change d'état, tous ceux qui en dépendent en soient notifiés et mis à jour automatiquement
Utilisation : Une abstraction a deux aspects, l'un dépendant de l'autre et chacun pouvant être modifié et réutilisé indépendamment | La modification d'un objet nécessite d'en modifier d'autres, mais leur nombre n'est pas connu | Un objet doit être capable d'en notifier d'autres sans savoir qui ils sont (i.e., que ces objets ne doivent pas être fortement couplés)
Avantages : Favorise un couplage faible entre le sujet et ses observateurs : Il est possible de réutiliser les sujets sans réutiliser leurs observateurs et vice versa | Des observateurs peuvent être ajoutés sans modifier le sujet | Le sujet connaît sa liste d'observateurs | Le sujet n'a pas besoin de connaître la classe concrète d'un observateur, juste que chaque observateur implémente l'interface de mise à jour | Les sujets et les observateurs peuvent appartenir à différentes couches d'abstraction | | Fournit un support à la diffusion d'événements (broadcast) : Le sujet envoie une notification à tous les observateurs abonnés | Des observateurs peuvent être ajouté/supprimé à tout moment
Inconvénients : Peut causer une cascade de notifications/mises à jour inopinées : Parce que les observateurs ignorent la présence des uns et des autres, ils doivent faire attention au déclenchement des mises à jour (coûteuses) | | Une simple interface de mise à jour requiert que les observateurs déduisent eux-même de ce qui a changé dans le sujet (difficile) : Besoin d'un protocole additionnel pour les aider

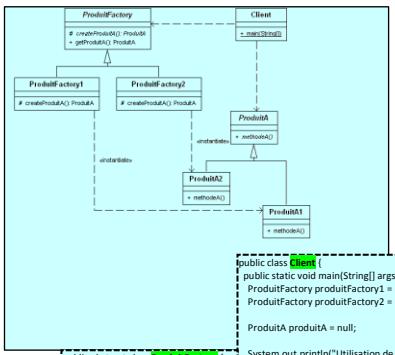
SFORZA Allan 2K16




```
public class Client {

public static void main(String[] args) {
  IProduitFactory produitFactory1 = new ProduitFactory1();
  IProduitFactory produitFactory2 = new ProduitFactory2();

  ProduitA produitA = null;
  ProduitB produitB = null;

  System.out.println("Utilisation de la premiere fabrique");
  produitA = produitFactory1.getProduitA();
  produitB = produitFactory1.getProduitB();
  produitA.methodeA();
  produitB.methodeB();

  System.out.println("Utilisation de la seconde fabrique");
  produitA = produitFactory2.getProduitA();
  produitB = produitFactory2.getProduitB();
  produitA.methodeA();
  produitB.methodeB();
}}
```

```
public class ProduitFactory1 implements IProduitFactory {
 public ProduitA getProduitA() { return new ProduitA1(); }
 public ProduitB getProduitB() { return new ProduitB1(); }
}
```

```
public class ProduitFactory2 implements IProduitFactory {
 public ProduitA getProduitA() { return new ProduitA2(); }
 public ProduitB getProduitB() { return new ProduitB2(); }
}
```




```
public interface IProduitFactory
{
 public ProduitA getProduitA();
 public ProduitB getProduitB();
}
```

```
public abstract class ProduitB {
 public abstract void methodeB();
}
```

```
public abstract class ProduitA {
 public abstract void methodeA();
}
```

```
public class ProduitB1 extends ProduitB {
 public void methodeB() {
  System.out.println("ProduitB1.methodeB()");
 }}
```

```
public class ProduitA1 extends ProduitA {
 public void methodeA() {

System.out.println("ProduitA1.methodeA()");
 }}
```

```
public class ProduitB2 extends ProduitB {
 public void methodeB() {
  System.out.println("ProduitB2.methodeB()");
 }}
```

```
public class ProduitA2 extends ProduitA {
 public void methodeA() {

System.out.println("ProduitA2.methodeA()");
 }}
```

```
public class Client {
 public static void main(String[] args) {
  ProduitFactory produitFactory1 = new ProduitFactory1();
  ProduitFactory produitFactory2 = new ProduitFactory2();

  ProduitA produitA = null;

  System.out.println("Utilisation de la premiere fabrique");
  produitA = produitFactory1.getProduitA();
  produitA.methodeA();

  System.out.println("Utilisation de la seconde fabrique");
  produitA = produitFactory2.getProduitA();
  produitA.methodeA();
}}
```

```
public abstract class ProduitFactory {
 public ProduitA getProduitA() {
  return createProduitA();
 }
```

```
public class ProduitFactory1 extends
ProduitFactory {
 protected ProduitA createProduitA() {
  return new ProduitA1();
}}
```

```
public class ProduitFactory2 extends
ProduitFactory {
 protected ProduitA createProduitA() {
  return new ProduitA2();
 }}
```

```
public abstract class ProduitA {
 public abstract void
methodeA();
}
```

```
public class ProduitA1 extends ProduitA {
 public void methodeA() {
  System.out.println("ProduitA1.methodeA()");
}}
```

```
public class ProduitA2 extends ProduitA {
 public void methodeA() {
  System.out.println("ProduitA2.methodeA()");
}}
```

Factory

```
// Classe abstraite decorateurIngredient qui hérite de dessert.
public abstract class DecorateurIngredient extends Dessert
{
    protected Dessert dessert;// Dessert sur lequel on applique l'ingrédient.
    // On oblige les ingrédients à implémenter la méthode getLibelle().
    public abstract String getLibelle();
    // On oblige les ingrédients à implémenter la méthode getPrix().
    public abstract double getPrix();
}
```

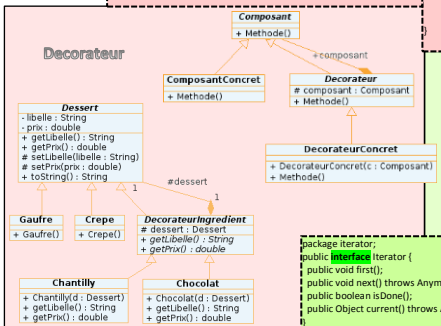
```
// Classe abstraite dessert.
public abstract class Dessert
{
    private String libelle;// Libellé du dessert.
    private double prix;// Prix du dessert.
    // Accesseurs en lecture pour le libellé et le prix.
    public String getLibelle()  { return libelle; }
    public double getPrix()  { return prix; }
    // Accesseurs en écriture pour le libellé et le prix.
    protected void setLibelle(String libelle) {this.libelle = libelle;}
    protected void setPrix(double prix)  {this.prix = prix;}
    // Méthode utilisée pour l'affichage d'un dessert.
    public String toString()
    {
        NumberFormat format=NumberFormat.getInstance();
        format.setMinimumFractionDigits(2);// 2 chiffres après la virgule suffisent pour l'affichage.
        return getLibelle()+" : "+format.format(getPrix())+"€";
    }
}
```

```
// Classe gaufre qui hérite de dessert
public class Gaufre extends Dessert
{
    // Constructeur qui intialise le libellé et le prix.
    public Gaufre()
    {
        setLibelle("Gaufre");
        setPrix(1.80);
    }
}
// Classe crêpe qui hérite de dessert.
public class Crepe extends Dessert
{
    // Constructeur qui initialise le libellé et le prix.
    public Crepe()
    {
        setLibelle("Crêpe");
        setPrix(1.50);
    }
}
```

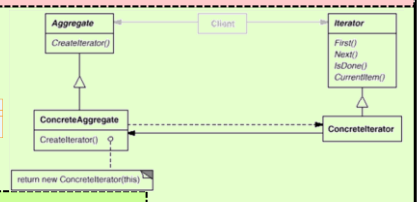
```
// Classe principale de l'application.
public class Main
{
    public static void main(String[] args)
    {
        // Création et affichage d'une gaufre au chocolat.
        Dessert d1 = new Gaufre();
        d1 = new Chocolat(d1);
        System.out.println(d1);
        // Création et affichage d'une crêpe au chocolat et chantilly.
        Dessert d2 = new Crepe();
        d2 = new Chocolat(d2);
        d2 = new Chantilly(d2);
        System.out.println(d2);
    }
}
```

```
//Classe chantily qui hérite de decorateurIngredient et donc indirectement de dessert.
public class Chantilly extends DecorateurIngredient
{
    // Constructeur qui prend en paramètre le dessert.
    public Chantilly(Dessert d) { dessert = d; }
    // On affiche le libellé du dessert et on rajoute le libellé de l'ingrédient chantilly.
    public String getLibelle() { return dessert.getLibelle()+", chantilly"; }
    // On additionne le prix du dessert et le prix de l'ingrédient chantilly.
    public double getPrix() { return dessert.getPrix()+0.50; }
}
// Classe chocolat qui hérite de decorateurIngredient et donc indirectement de dessert.
public class Chocolat extends DecorateurIngredient
{
    // Constructeur qui prend en paramètre le dessert.
    public Chocolat(Dessert d) {dessert = d;}
    // On affiche le libellé du dessert et on rajoute le libellé de l'ingrédient chocolat.
    public String getLibelle()    { return dessert.getLibelle()+", chocolat"; }
    // On additionne le prix du dessert et le prix de l'ingrédient chocolat.
    public double getPrix()  { return dessert.getPrix()+0.20; }
}
```

Decorateur



Iterateur



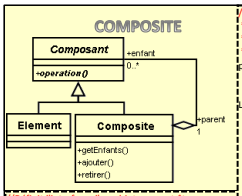

```
package iterator;
public interface Iterator {
 public void first();
 public void next() throws AnymoreElementException;
 public boolean isDone();
 public Object current() throws AnymoreElementException;
}
```

```
package iterator;
public class AnymoreElementException extends Exception {
 private static final long serialVersionUID = 1L;
}
```

```
package iterator;

import flower.Flower;
import flower.FlowerField;

public class FlowerIterator implements Iterator {
 protected FlowerField field;
 protected int currentFlower;
 public FlowerIterator(FlowerField field) {
  this.field = field;
  this.currentFlower = 1;
 }
 @Override
 public void first() {
  this.currentFlower = 1;
 }
 @Override
 public void next() throws AnymoreElementException {
  if (isDone()) {
   throw new AnymoreElementException();
  }
  this.currentFlower++;
 }
 @Override
 public boolean isDone() {
  return this.currentFlower > this.field.nbFlowers();
 }
 @Override
 public Flower current() throws AnymoreElementException {
  if (isDone()) {
   throw new AnymoreElementException();
  }
  return this.field.getFlower(this.currentFlower - 1);
 }}
```

```
package iterator;

import flower.Flower;
import flower.FlowerField;

public class SpeciesIterator<T extends Flower> extends FlowerIterator {
//Ne peut contenir que Flower ou des objets qui héritent de Flower

//Une classe de type T : donc une espèce de fleur ou une fleur
 private Class<Flower> species;

 public SpeciesIterator(FlowerField field, Class<Flower> species) {
  super(field);
  this.species = species;
  first();
 }

 public void first() {
  this.currentFlower = 1;
  try {
   //on vérifie que l'objet courant n'est pas une instance de la classe de dessus
   while (!species.isInstance(current())) {
    super.next();
   }
  } catch (AnymoreElementException e) {
  }}

 @Override
 public void next() throws AnymoreElementException {
  do {
   super.next();
  } while (!isDone() && !species.isInstance(current()));
 }
}
```

COMPOSITE



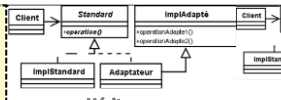

```
//Définit l'interface d'un objet pouvant être un composant
//d'un autre objet de l'arborescence.
public abstract class Composant {
  // Nom de "Composant"
  protected String nom;
  //Constructeur
  //@param pNom Nom du "Composant"
  public Composant(final String pNom) {
    nom = pNom;
  }
  //Opération commune à tous les "Composant"
  public abstract void operation();
}
```

```
/**
 * Implémente un objet de l'arborescence
 * n'ayant pas d'objet le composant.
 */
public class Element extends Composant {
  public Element(final String pNom) {
    super(pNom);
  }
  /**
   * Méthode commune à tous les composants :
   * Affiche qu'il s'agit d'un objet "Element"
   * ainsi que le nom qu'on lui a donné.
   */
  public void operation() {
```

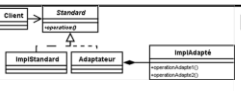
```
/**
 * Implémente une objet de l'arborescence
 * ayant un ou des objets le composant.
 */
public class Composite extends Composant {
  // Liste d'objets "Composant" de l'objet "Composite"
  private List<Composant> liste = new LinkedList<Composant>();
  public Composite(final String pNom) {
    super(pNom);
  }
  /**
   * Méthode commune à tous les composants :
   * Affiche qu'il s'agit d'un objet "Composite"
   * ainsi que le nom qu'on lui a donné,
   * puis appelle la méthode "operation()"
   * de tous les composants de cet objet.
   */
  public void operation() {
    System.out.println("Op. sur un 'Composite' (" + nom + ")");
    final Iterator<Composant> lIterator = liste.iterator();
    while(lIterator.hasNext()) {
      final Composant lComposant = lIterator.next();
      lComposant.operation();
    }
  }
  /**
   * Retourne la liste d'objets "Composant"
   * @return La liste d'objets "Composant"
   */
  public List<Composant> getEnfants() {
    return liste;
  }
  /**
   * Ajoute un objet "Composant" au "Composite"
   * @param pComposant
   */
  public void ajouter(final Composant pComposant) {
    liste.add(pComposant);
  }
  /**
   * Retire un objet "Composant"
   * @param pComposant
   */
  public void retirer(final Composant pComposant) {
    liste.remove(pComposant);
  }
}
```

```
public class CompositePatternMain {
  public static void main(String[] args) {
    // Création des objets "Composite"
    final Composite lComposite1 = new Composite("Composite 1");
    final Composite lComposite2 = new Composite("Composite 2");
    final Composite lComposite3 = new Composite("Composite 3");
    // Création des objets "Element"
    final Element lElement1 = new Element("Element 1");
    final Element lElement2 = new Element("Element 2");
    final Element lElement3 = new Element("Element 3");
    // Ajout des "Composant" afin de constituer l'arborescence
    lComposite1.ajouter(lElement1);
    lComposite1.ajouter(lComposite2);
    lComposite1.ajouter(lElement2);
    lComposite2.ajouter(lComposite3);
    lComposite3.ajouter(lElement3);
  }
}
```

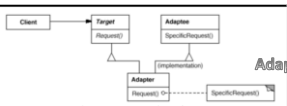
Héritage



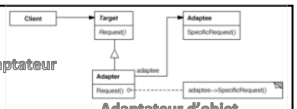
Composition



Adaptateur de classe



Adaptateur d'objet



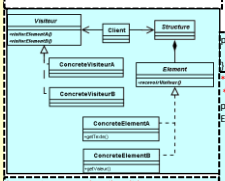

```
//Définit une interface qui est identifiée
// comme standard dans la partie cliente.
public interface Standard {
  // L'opération doit multiplier les deux nombres
  // puis afficher le résultat de l'opération
  public void operation(int pNombre1, int pNombre2);
}
```

```
//Implémente l'interface "Standard".
public class ImplStandard implements Standard {
  public void operation(int pNombre1, int pNombre2) {
    System.out.println("Standard : Le nombre est : " + (pNombre1 * pNombre2));
  }
}
```

```
//Fournit les fonctionnalités définies dans l'interface "Standard",
//mais ne respecte pas l'interface.
public class ImplAdapté {
  public int operationAdapte1(int pNombre1, int pNombre2) {
    return pNombre1 * pNombre2;
  }
  * Apporte la fonctionalité définie dans l'interface,
  * mais la méthode n'a pas le bon nom
  * et n'accepte pas le même paramètre.
  */
  public void operationAdapte2(int pNombre) {
    System.out.println("Adapte : Le nombre est : " + pNombre);
  }
}
```

```
//Adapte l'implémentation non standard avec l'héritage.
public class Adaptateur extends ImplAdapte implements Standard {
  * Appelle les méthodes non standard
  * depuis une méthode respectant l'interface.
  * 1°) Appel de la méthode réalisant la multiplication
  * 2°) Appel de la méthode d'affichage du résultat
  * La classe adaptée est héritée, donc on appelle directement les méthodes
  public void operation(int pNombre1, int pNombre2) {
    int lNombre = operationAdapte1(pNombre1, pNombre2);
    operationAdapte2(lNombre);
  }
}
```

Visiteur




```
public interface Element {
  public void recevoirVisiteur(Visiteur pVisiteur);
}
```

```
* Sous-classe concrète d'un élément.
* Contient un donnée texte
public class ConcreteElementA implements Element {
  public String texte;
  public ConcreteElementA(String pTexte) {
    texte = pTexte;
  }
  public String getTexte() {
    return texte;
  }
  public void recevoirVisiteur(Visiteur pVisiteur) {
    pVisiteur.visiterElementA(this);
  }
}
```

```
* Présente une interface de haut niveau permettant
* de visiter les objets "Element" la composant.
public class Structure {
  private Element[] elements = new Element[] {
      new ConcreteElementA("texte1"),
      new ConcreteElementA("texte2"),
      new ConcreteElementB(new Long(1)),
      new ConcreteElementA("texte3"),
      new ConcreteElementB(new Long(2)),
      new ConcreteElementB(new Long(3))
  };
  // Méthode de visite
  public void visiter(Visiteur pVisiteur) {
    for(int i=0;i<elements.length;i++) {
      elements[i].recevoirVisiteur(pVisiteur);
    }
  }
}
```

```
* Sous-classe concrète d'un élément.
* Contient un donnée numérique
public class ConcreteElementB implements Element {
  public Long valeur;
  public ConcreteElementB(Long pValeur) {
    valeur = pValeur;
  }
  public Long getValeur() {
    return valeur;
  }
  public void recevoirVisiteur(Visiteur pVisiteur) {
    pVisiteur.visiterElementB(this);
  }
}
```

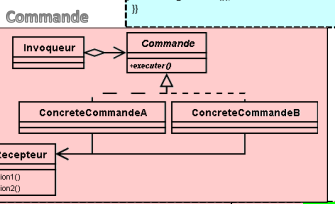
```
* Définit l'interface d'un visiteur
public interface Visiteur {
  public void visiterElementA(ConcreteElementA pElementA);
  public void visiterElementB(ConcreteElementB pElementB);
}
```

```
* Sous-classe concrète d'un visiteur.
public class ConcreteVisiteurA implements Visiteur {
  public void visiterElementA(ConcreteElementA pElementA)
  {
    System.out.println("Visiteur A : ");
    System.out.println("  Texte de l'element A : " + pElementA.getTexte());
  }
  public void visiterElementB(ConcreteElementB pElementB)
  {
    System.out.println("Visiteur A : ");
    System.out.println("  Valeur de l'element B : " + pElementB.getValeur());
  }
}
```

```
* Sous-classe concrète d'un visiteur.
public class ConcreteVisiteurB implements Visiteur {
  public void visiterElementA(ConcreteElementA pElementA) {
    System.out.println("Visiteur B : ");
    System.out.println("  Texte de l'element A : " + pElementA.getTexte());
  }
  public void visiterElementB(ConcreteElementB pElementB) {
    System.out.println("Visiteur B : ");
    System.out.println("  Valeur de l'element B : " + pElementB.getValeur());
  }
}
```

```
public class VisiterPatternMain {
  public static void main(String[] args) {
    // Création des visiteurs
    Visiteur lVisiteurA = new ConcreteVisiteurA();
    Visiteur lVisiteurB = new ConcreteVisiteurB();
    // Création de la structure
    Structure lStructure = new Structure();
    // Appels des méthodes de réception des visiteurs
    lStructure.visiter(lVisiteurA);
    lStructure.visiter(lVisiteurB);

    // Affichage :
    // Visiteur A :
    // Texte de l'element A : texte1
    // Visiteur A :
    //   Texte de l'element A : texte2
    // Visiteur A :
    //   Valeur de l'element B : 1
    // Visiteur A :
    //   Texte de l'element A : texte3
    // Visiteur A :
    //   Valeur de l'element B : 2
    // Visiteur A :
    //   Valeur de l'element B : 3
    // Visiteur B :
    //   Texte de l'element A : texte1
    // Visiteur B :
    //   Texte de l'element A : texte2
    // Visiteur B :
    //   Valeur de l'element B : 1
    // Visiteur B :
    //   Texte de l'element A : texte3
    // Visiteur B :
    //   Valeur de l'element B : 2
    // Visiteur B :
    //   Valeur de l'element B : 3
  }
}
```

Commande




```
/**
 * Déclenche les commandes.
 */
public class Invoqueur {
  // Références vers les commandes
  private Commande commandeA;
  private Commande commandeB;
  // Méthodes pour invoquer les commandes
  public void invoqueurA() {
    if(commandeA != null) {
      commandeA.executer();
    }
  }
  public void invoqueurB() {
    if(commandeB != null) {
      commandeB.executer();
    }
  }
  // Méthodes pour fixer les commandes
  public void setCommandeA(Commande pCommandeA) {
    commandeA = pCommandeA;
  }
  public void setCommandeB(Commande pCommandeB) {
    commandeB = pCommandeB;
  }
}
```

```
public interface Commande {
  public void executer();
}
```

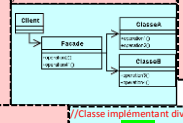
```
/**
 * Reçoit la commande.
 */
public class Recepteur {
  public void action1() {
    System.out.println("Traitement numero 1 effectué.");
  }
  public void action2() {
    System.out.println("Traitement numero 2 effectué.");
  }
}
```

```
/**
 * Implémente une commande.
 * Appelle la méthode action1() lorsque la commande est exécutée.
 */
public class ConcreteCommandeA implements Commande {
  private Recepteur recepteur;

  public ConcreteCommandeA(Recepteur pRecepteur) {
    recepteur = pRecepteur;
  }

  public void executer() {
    recepteur.action1();
  }
}
```

```
/**
 * Implémente une commande.
 * Appelle la méthode action2() lorsque la commande est exécutée.
 */
public class ConcreteCommandeB implements Commande {

  private Recepteur recepteur;
  public ConcreteCommandeB(Recepteur pRecepteur) {
    recepteur = pRecepteur;
  }
  public void executer() {
    recepteur.action2();
  }
}
```

```
public class CommandPatternMain {

  public static void main(String[] args) {
    // Création d'un récepteur
    Recepteur lRecepteur = new Recepteur();
    // Création des commandes
    Commande lCommandeA = new ConcreteCommandeA(lRecepteur);
    Commande lCommandeB = new ConcreteCommandeB(lRecepteur);
    // Création et initialisation de l'invoqueur
    Invoqueur lInvoqueur = new Invoqueur();
    lInvoqueur.setCommandeA(lCommandeA);
    lInvoqueur.setCommandeB(lCommandeB);
    // Appel des méthodes d'invocation
    // NB : Cette classe représente la partie cliente.
    // Donc, normalement l'invocation
    // ne se passe pas dans la partie cliente
    // Dans l'exemple, elle est ici par souci de concision
    lInvoqueur.invoqueurA();
    lInvoqueur.invoqueurB();
    // Affichage :
    // Traitement numero 1 effectué.
    // Traitement numero 2 effectué.
  }
}
```

Facade



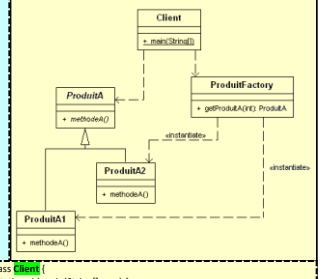

```
//Classe implémentant diverses fonctionnalités.
public class ClasseA {
  public void operation1() {
    System.out.println("Methode operation1() de la classe ClasseA");
  }
  public void operation2() {
    System.out.println("Methode operation2() de la classe ClasseA");
  }
}
```

```
// Classe implémentant d'autres fonctionnalités.
public class ClasseB {
  public void operation3() {
    System.out.println("Methode operation3() de la classe ClasseB");
  }
  public void operation4() {
    System.out.println("Methode operation4() de la classe ClasseB");
  }
}
```

```
public class FacadePatternMain {
  public static void main(String[] args) {
    // Création de l'objet "Facade" puis appel des méthodes
    Facade lFacade = new Facade();
    lFacade.operation2();
    lFacade.operation41();

    // Affichage :
    // --> Méthode operation2() de la classe Facade :
    // Methode operation2() de la classe ClasseA
    // --> Méthode operation41() de la classe Facade :
    // Methode operation4() de la classe ClasseB
    // Methode operation1() de la classe ClasseA
  }
}
```

```
/**
 * Présente certaines fonctionnalités.
 * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
 * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
 * et "operation1()" de "ClasseA".
 */
public class Facade {
  private ClasseA classeA = new ClasseA();
  private ClasseB classeB = new ClasseB();
  /**
   * La méthode operation2() appelle simplement
   * la même méthode de ClasseA
   */
  public void operation2() {
    System.out.println("--> Méthode operation2() de la classe Facade :");
    classeA.operation2();
  }
  /**
   * La méthode operation41() appelle
   * operation4() de ClasseB
   * et operation1() de ClasseA
   */
  public void operation41() {
    System.out.println("--> Méthode operation41() de la classe Facade :");
    classeB.operation4();
    classeA.operation1();
  }
}
```

Factory (Int)




```
public abstract class ProduitA {
  public abstract void methodeA();
}
```

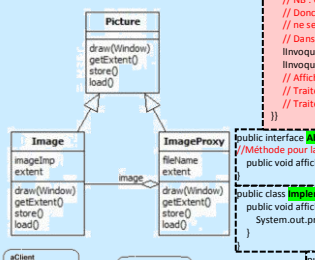
```
public class ProduitA1 extends ProduitA {
  public void methodeA() {
    System.out.println("ProduitA1.methodeA()");
  }
}
```

```
public class ProduitA2 extends ProduitA {
  public void methodeA() {
    System.out.println("ProduitA2.methodeA()");
  }
}
```

```
public class ProduitFactory {
  public static final int TYPE_PRODUITA1 = 1;
  public static final int TYPE_PRODUITA2 = 2;

  public ProduitA getProduitA(int typeProduit) {
    ProduitA produitA = null;

    switch (typeProduit) {
      case TYPE_PRODUITA1:
        produitA = new ProduitA1();
        break;
      case TYPE_PRODUITA2:
        produitA = new ProduitA2();
        break;
      default:
        throw new IllegalArgumentException("Type de produit inconnu");
```

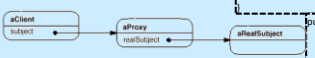
```
public class Client {
  public static void main(String[] args) {
    ProduitFactory produitFactory = new ProduitFactory();

    ProduitA produitA = null;
    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA1);
    produitA.methodeA();
    produitA = produitFactory.getProduitA(ProduitFactory.TYPE_PRODUITA2);
    produitA.methodeA();
    produitA = produitFactory.getProduitA(3);
    produitA.methodeA();
```

Proxy




```
public interface Abstraction {
//Méthode pour laquelle on souhaite un "Proxy"
  public void afficher();
}
```

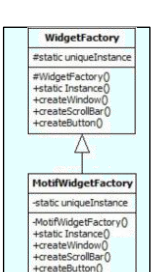
```
public class Implementation implements Abstraction {
  public void afficher() {
    System.out.println("Méthode afficher() de la classe d'implementation");
  }
}
```




```
public class ProxyPatternMain {
  public static void main(String[] args) {
    // Création du "Proxy"
    Abstraction lProxy = new Proxy();
    // Appel de la méthode du "Proxy"
    lProxy.afficher();
    // Affichage :
    // --> Méthode afficher() du Proxy :
    // --> Création de l'objet Implementation au besoin
    // --> Appel de la méthode afficher() de l'objet Implementation
    // Méthode afficher() de la classe d'implementation
  }
}
```

```
public class Proxy implements Abstraction {
  //Instancie l'objet "Implementation", pour appeler la vraie implémentation de la méthode.
  public void afficher() {
    System.out.println("--> Méthode afficher() du Proxy : ");
    System.out.println("--> Création de l'objet Implementation au besoin");
    Implementation lImplementation = new Implementation();
    System.out.println("--> Appel de la méthode afficher() de l'objet Implementation");
    lImplementation.afficher();
  }
}
```

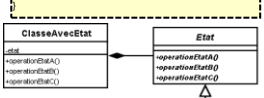
Singleton

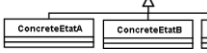



```
public final class MonSingleton {
  private static MonSingleton instance;
  private MonSingleton() {}

  public static MonSingleton getInstance() {
    if (instance == null) {
      instance = new MonSingleton();
    }
    return instance;
  }
```

```
public final class MonSingleton {
  private static MonSingleton instance = new MonSingleton;
  private MonSingleton() {}

  public static MonSingleton getInstance() {
    return instance;
  }
  private MonSingleton() {}
```

Etat





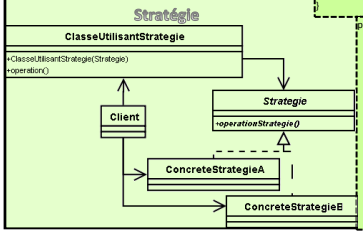

```
public class ClasseAvecEtat {
  private Etat etat;
  public static abstract class Etat {
    protected void setEtat(ClasseAvecEtat pClasse, Etat pEtat) {
      pClasse.etat = pEtat;
    }
    // Méthodes pour changer d'état
    public abstract void operationEtatA(ClasseAvecEtat pClasse);
    public abstract void operationEtatB(ClasseAvecEtat pClasse);
    public abstract void operationEtatC(ClasseAvecEtat pClasse);
    // Affichage de l'état courant
    public abstract void afficher();
  }
  // Constructeur avec initialisation à l'état A
  public ClasseAvecEtat() {
    etat = new ConcreteEtatA();
  }
  // Méthodes pour changer d'état
  public void operationEtatA() {
    etat.operationEtatA(this);
  }
  public void operationEtatB() {
    etat.operationEtatB(this);
  }
  public void operationEtatC() {
    etat.operationEtatC(this);
  }
  // Affichage de l'état courant
  public void afficher() {
    etat.afficher();
  }
}
```

```
public class ConcreteEtatA extends ClasseAvecEtat.Etat {
  // Méthodes pour changer d'état
  public void operationEtatA(ClasseAvecEtat pClasse) {
    System.out.println("Classe déjà dans l'état A");
  }
  public void operationEtatB(ClasseAvecEtat pClasse) {
    setEtat(pClasse, new ConcreteEtatB());
    System.out.println("Etat changé : A -> B");
  }
  public void operationEtatC(ClasseAvecEtat pClasse) {
    setEtat(pClasse, new ConcreteEtatC());
    System.out.println("Etat changé : A -> C");
  }
  //Affichage de l'état courant
  public void afficher() {
    System.out.println("Etat courant : A");
  }
}
```

```
public class ConcreteEtatB extends ClasseAvecEtat.Etat {
  // Méthodes pour changer d'état
  public void operationEtatA(ClasseAvecEtat pClasse) {
    System.out.println("Changement d'état (B -> A) non possible");
  }
  public void operationEtatB(ClasseAvecEtat pClasse) {
    System.out.println("Classe déjà dans l'état B");
  }
  public void operationEtatC(ClasseAvecEtat pClasse) {
    setEtat(pClasse, new ConcreteEtatC());
    System.out.println("Etat changé : B -> C");
  }
  // Affichage de l'état courant
  public void afficher() {
    System.out.println("Etat courant : B");
  }
}
```

```
public class StatePatternMain {
  public static void main(String[] args) {
    // Création de la classe avec état
    ClasseAvecEtat lClasse = new ClasseAvecEtat();

    lClasse.operationEtatA();
    lClasse.operationEtatB();
    lClasse.operationEtatA();
    lClasse.afficher();
    lClasse.operationEtatB();
    lClasse.operationEtatC();
    lClasse.operationEtatA();

    // Affichage :
    // Classe déjà dans l'état A
    // Etat changé : A -> B
    // Changement d'état (B -> A) non possible
    // Etat courant : B
    // Classe déjà dans l'état B
    // Etat changé : B -> C
    // Etat changé : C -> A
```

Stratégie




```
public interface Strategie {
  public void operationStrategie();
}
```

```
//Définit une stratégie
public class ConcreteStrategieA implements Strategie {
  public void operationStrategie() {
    System.out.println("Operation de la strategie A");
  }
}
```

```
//Définit une stratégie
public class ConcreteStrategieB implements Strategie
{
  public void operationStrategie() {
    System.out.println("Operation de la strategie B");
  }
}
```

```
public class ClasseUtilisantStrategie {
  private Strategie strategie;
  public ClasseUtilisantStrategie(Strategie pStrategie) {
    strategie = pStrategie;
  }
  // Délègue le traitement à la stratégie
  public void operation() {
    strategie.operationStrategie();
  }
}
```

```
public class StrategyPatternMain {
  public static void main(String[] args) {
    // Création d'instance des classes "Strategie"
    Strategie lStrategieA = new ConcreteStrategieA();
    Strategie lStrategieB = new ConcreteStrategieB();
    // Création d'instance de la classe qui utilise des "Strategie"
    ClasseUtilisantStrategie lClasseA = new ClasseUtilisantStrategie(lStrategieA);
    ClasseUtilisantStrategie lClasseB = new ClasseUtilisantStrategie(lStrategieB);
    // Appel de la méthode de la classe
    // utilisant une stratégie
    lClasseA.operation();
    lClasseB.operation();
    // Affichage :
    // Operation de la strategie A
    // Operation de la strategie B
  }
}
```