Patrons de conception :

- Identifiez les aspects de votre code qui varient et séparez-les de ceux qui demeurent constant. - Programmer une interface, non une implémentation. - Préférer la composition à l'héritage.

Définitions : Un patron de concept° ou design pattern est un schéma à objets qui forme une solution à un problème connu et fréquent. Le schéma à objets est constitué par un ensemble d'objets décrits par des classes et des relat° liant les objets.

Catégories des patrons : - **Patrons de structure :** ils servent à faciliter l'organisation de la hiérarchie des classes et de leurs relations. ***Ces modèles tendent à concevoir des agglomérations de classes avec des macro-composants***. (Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy).

- **Patrons de comportement :** ils servent à maîtriser les interactions entre objets en organisant les interactions et en répartissant les traitements entre les objets. ***Ces modèles tentent de répartir les responsabilités entre les classes.*** (Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor).

- **Patrons de création :** ils ont pour but d'organiser la (de gérer les problèmes de) création de nouveaux objets. ***Ces modèles sont très courants pour désigner une classe chargée de construire des objets.*** (Abstract Factory, Builder, Factory Method, Prototype, Singleton).

Patron Composite : But : offrir un cadre de conception d'une composition d'objets dont la profondeur est variable. Utilisation : - nécessité de représenter au sein d'un système des hiérarchies de composition. Les clients d'une composition doivent ignorer s'ils communiquent avec des objets composés ou non.

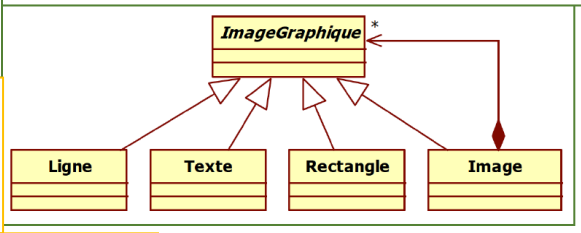
```
public abstract class ImageGraphique {
protected List<ImageGraphique> elements = new ArrayList<ImageGraphique>(); Color myColor;
public void remplisser(Color color) { myColor = color;}
public boolean ajouter(ImageGraphique element) { return false; }
public boolean supprimer(ImageGraphique element) { return false; }
public ImageGraphique getChild(int i) { return null; }
public abstract void dessiner(int i); //operation() }
```

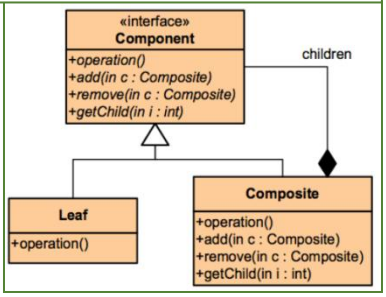
```
public class Image extends ImageGraphique{
public boolean ajouter(ImageGraphique element) { return elements.add(element); }
public boolean supprimer(ImageGraphique element) { return elements.remove(element);}
public ImageGraphique getChild(int i) { return elements.get(i); }
public void dessiner(int i) { System.out.println("L'image " + this + " est composé des elements graphiques suivants : "); i++;  for (ImageGraphique element: elements) { for(int j = 0; j<i; j++)
System.out.print("\t"); element.dessiner(i); } } }
```

```
public class Ligne extends ImageGraphique {
public void dessiner(int i) { System.out.println("Affichage de la ligne " + this + " de couleur " + myColor); } }
```

```
public class Rectangle extends ImageGraphique {
public void dessiner(int i) { System.out.println("Affichage du rectangle " + this + " de couleur " + myColor); }
```

```
public class Utilisateur { public static void main(String[] args) {
ImageGraphique ligne = new Ligne(); ligne.remplisser(Color.red);
ImageGraphique rec = new Rectangle(); rec.remplisser(Color.green);
ImageGraphique image = new Image(); image.ajouter(ligne);
image.ajouter(rec); }
```





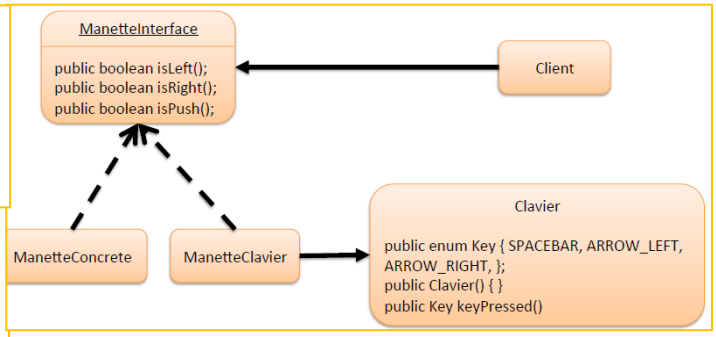
Patron Adapter : But : convertir l'interface d'une classe existante en l'interface attendue par des clients.

Utilisation : - intégrer dans un système un objet dont l'interface ne correspond pas à l'interface requise au sein de ce système.

- fournir des interfaces multiples à un objet lors de sa conception.

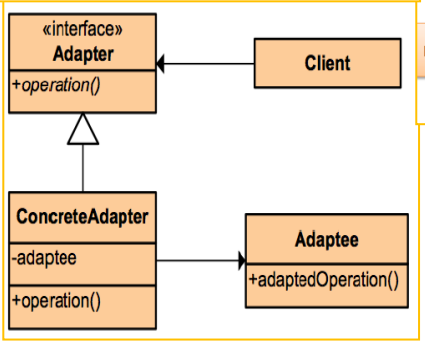
```
public class JeuMain { public static void main(String[] args) { Manette manette = new ManetteClavier(new Clavier()); Jeu jeu = new Jeu(manettee); jeu.mainLoop(); }
```

```
public class ManetteClavier implements Manette{ private Clavier clavier;
public ManetteClavier(Clavier clavier) { this.clavier = clavier; }
public boolean isLeft() { if(clavier.keyPressed() == Key. ARROW_LEFT) return true; return false;}
public boolean isRight() { if(clavier.keyPressed() == Key. ARROW_RIGHT) return true; return false;}
public boolean isPush() { if(clavier.keyPressed() == Key. SPACEBAR) return true; return false;} }
```



Patron Façade : But : permettre à un client d'utiliser de façon simple et transparente un ensemble de classes qui collaborent pour réaliser une tâche courante.// Moyen : Une classe fournit une façade en proposant des méthodes qui réalisent les tâches usuelles en utilisant les autres classes.

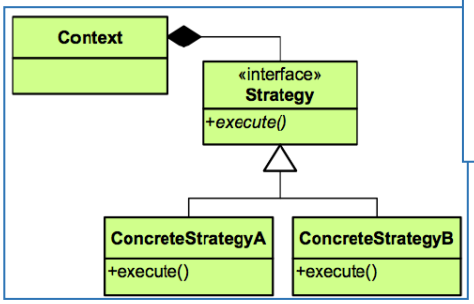
Comparaison : **Adaptateur et façade** ont une interface différente de ce qu'ils enveloppent. Mais l'adaptateur dérive de l'interface existante, tandis que la façade crée une nouvelle interface.



Patron Strategy : But : adapter le comportement et les algorithmes d'un objet avec les clients.

Cas d'utilisation : - Le comportement d'une classe peut être implanté par différents algorithmes dont certains sont plus efficaces en temps d'exécution ou en consommation mémoire ou encore contiennent des mécanismes de mise au point. - L'implantation du choix de l'algorithme par des instructions conditionnelles est trop complexe. - Un système possède de nombreuses classes identiques à l'exception d'une partie de leur comportement.

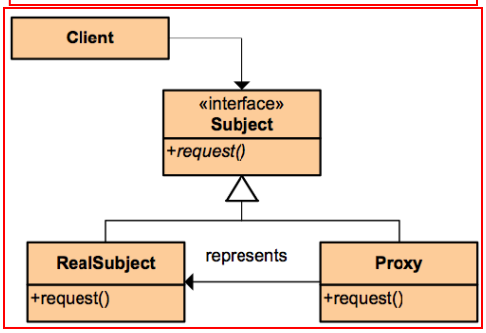
Dans ce cas, le Patron Strategy permet de grouper ces classes en une seule, ce qui simplifie l'interface pour les clients.



1. La nouvelle classe nommée FacadeHomeCinema expose un **petit nombre de méthodes simples** comme `regarderFilm()`.
2. Elle traite les **composants du home cinéma comme un sous-système**, et fait appel au sous-système pour implémenter sa méthode `regarderFilm()`.
3. **Le code du client appelle maintenant les méthodes sur la Façade**, pas sur le sous-système. Désormais, pour regarder un film, il suffit d'appeler une seule méthode, `regarderFilm()`, et celle-ci communique à notre place avec les lumières, le lecteur de DVD, le projecteur, l'amplificateur, l'écran et la machine à pop-corn.
4. La Façade permet toujours d'utiliser directement **le sous-système qui demeure accessible**. Si vous avez besoin des fonctionnalités avancées des classes du sous-système, elles sont disponibles.

Anti Patron : **Abstraction inverse - Action à distance - Ancre de bateau - Attente active - Interblocages et Famine - Erreur de copier/coller - Programmation Spaghetti.**

Patron Proxy : But : la conception d'un objet qui se substitue à un autre objet (le sujet) et qui en contrôle l'accès. L'objet qui effectue la substitution possède la même interface que le sujet, ce qui rend cette substitution transparente vis-à-vis des clients.



public interface **Image** { public void showImage();}

public class **ImageProxy** implements Image { private String imageFilePath; private Image proxifiedImage;

public ImageProxy(String imageFilePath) { this.imageFilePath = imageFilePath; }

public void showImage() { proxifiedImage = new HighResolutionImage(imageFilePath); proxifiedImage.showImage(); } }}

public class **HighResolutionImage** implements Image {

public HighResolutionImage(String imageFilePath) { loadImage(imageFilePath); }

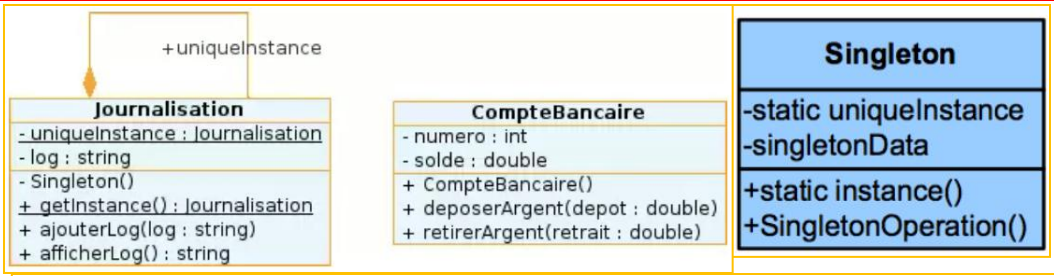
private void loadImage(String imageFilePath) { // load Image from disk into memory, this is heavy and costly operation }

public void showImage() { // Actual Image rendering logic } }

public class **ImageViewer** { public static void main(String[] args) { Image highResolutionImage1 = new ImageProxy("1.jpeg"); Image highResolutionImage2 = new ImageProxy("2.jpeg"); highResolutionImage1.showImage() ; } }

Patron Singleton : But : assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode de classe unique retournant cette instance.

Utilisation : - Il ne doit y avoir qu'une seule instance d'une classe. - Cette instance ne doit être accessible qu'au travers d'une méthode de classe. - Il offre la possibilité de ne plus utiliser de variables globales.



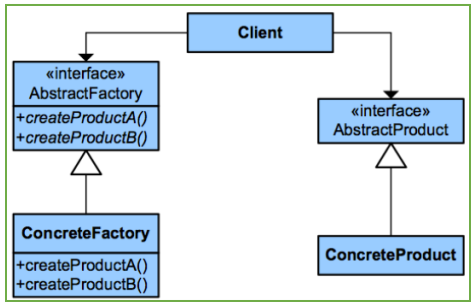
public class **Journalisation** { private static Journalisation uniqueInstance; private String log;

private Journalisation() { log = new String(); } public String afficherLog() { return log; } }

public static synchronized Journalisation getInstance() { if(uniqueInstance==null) { uniqueInstance = new JournalisaKon(); } return uniqueInstance; }

public void ajouterLog(String log) { Date d = new Date(); DateFormat dateFormat = new SimpleDateFormat("dd/MM/yy HH'h'mm"); this.log+="["+dateFormat.format(d)+"] "+log+"\n"; }

Patron Abstract Factory : But : permettre à un client de créer des objets sans savoir leurs types précis.



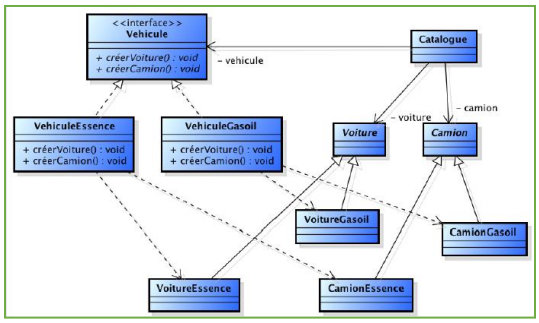
public abstract class **Voiture** { protected String modele; protected int puissance; public Voiture(String modele, int puissance) {this.modele = modele; this.puissance = puissance;} public abstract void afficher(); }

public class **VoitureGasoil** extends Voiture{ public void afficher(){ System.out.println("Voiture à gasoil de modèle : " + modele + " de puissance " + puissance); } } // De même pour VoitureEssence, Camion, ...

public interface **FabriqueVehicule** { Voiture creerVoiture(String modele, int puissance); Camion creerCamion(int categorie, int puissance); }

public class **FabriqueVehiculeGasoil** implements Vehicule { Voiture creerVoiture(String modele, int puissance) { return new VoitureGasoil(modele, puissance) }; // De même pour creer Camion, VehiculeEssence, ...

import java.util.*; public class **Catalogue** { public static void main(String[] args) { FabriqueVehicule fabrique; fabrique = new FabriqueVehiculeGasoil(); Voiture v = fabrique.creerVoiture("X", 2); Camion c = fabrique.creerCamion(1, 6); v.afficher(); c.afficher(); }}



Patron Observer : But : construire une dépendance entre un sujet et des observateurs de sorte que chaque modification du sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état.

public interface **Observateur** { void actualise(); }

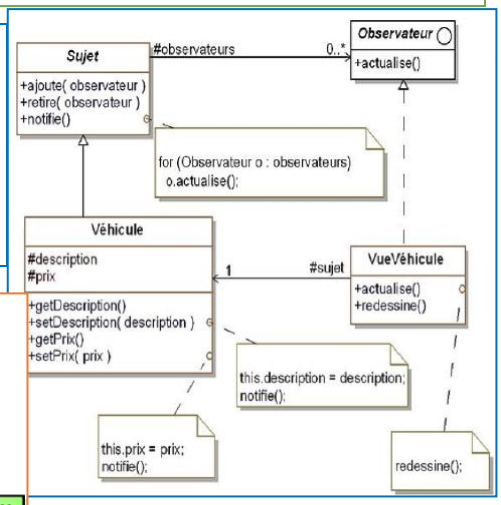
public class **VueVéhicule** implements Observateur

{ protected Véhicule veh; protected String txt ="";

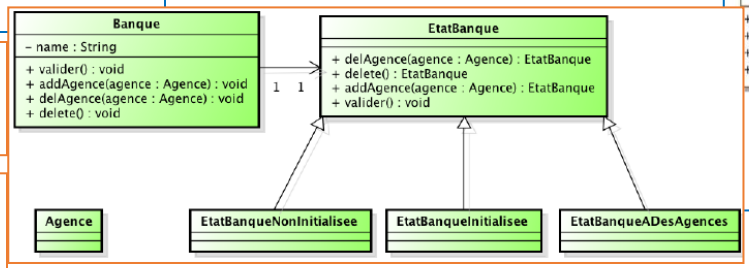
public VueVéhicule (Véhicule veh) {this.veh = veh; veh.ajoute(this); actualise(); }

public void actualise() { txt = "bla bla bla"}

public void affiche() { System.out.println(txt); }



Patron State : But : permettre à un objet d'adapter son comportement en fonction de son état interne.



public class **Banque** { private Collection agences; private EtatBanque etat; private String nom;

public Banque() {

etat = new EtatNonInitialise(this); agences = new ArrayList(); }

public void validate() { etat.validate(); }

// + methodes }

public class **EtatBanqueNonInitialisee** implements EtatBanque { private final Banque banque;

public EtatBanqueNonInitialisee(Banque banque) { if (banque == null) { throw new IllegalArgumentException("no"); }

this.banque = banque; } public void delAgence(Agence agence) { throw new IllegalStateException("no"); }

public void delete() { throw new IllegalStateException("no"); } public void addAgence(Agence agence) { throw new IllegalStateException("no");} public void validate() { if (null == banque.getNom()) { throw new IllegalStateException("no"); }

banque.setEtat(new EtatBanqueInitialisee(banque)); } } (et de même pour chacune avec ses méthodes).