Resumé

samedi 4 juillet 2020 17:23

Plan

Introduction

Principes

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

Définitions

- Un patron de conception (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 relations liant les objets
- Solutions connues et éprouvées dont la conception provient de l'expérience de programmeurs

Descriptions

Pour chaque Patron, les éléments suivants sont présentés :

- Nom du Patron
- Description du Patron
- Exemple décrivant le problème et la solution basée sur le Patron
- Structure générique du Patron
  - Schéma en dehors de tout contexte particulier sous la forme d'un diagramme de classes UML
  - Liste des participants du Patron
  - Collaboration au sein du Patron
- Domaine d'application du Patron

Vue Globale

Catalogue

- Structure
  - **Adapter :** a pour but de convertir l'interface d'une classe existante en l'interface attendue par des clients également existants afin qu'ils puissent travailler ensemble
  - Bridge : a pour but de séparer les aspects conceptuels d'une hiérarchie de classes de leur implantation
  - **Composite :** offre un cadre de conception d'une composition d'objets dont la profondeur de composition est variable, la conception étant basée sur un arbre
  - Decorator : permet d'ajouter dynamiquement des fonctionnalités supplémentaires à un objet
  - **Facade :** a pour but de regrouper les interfaces d'un ensemble d'objets en une interface unifiée rendant cet ensemble plus simple à utiliser
  - Flyweight : facilite le partage d'un ensemble important d'objets dont le gain est fin
  - **Proxy :** construit un objet qui se substitue à un autre objet et qui contrôle son accès
- Création
  - **Abstract Factory :** a pour objectif la création d'objets regroupés en familles sans devoir connaître les classes concrètes destinées à la création de ces objets
  - Builder : permet de séparer la construction d'objets complexes de leur implantation de sorte qu'un client puisse créer ces objets complexes avec des implantations différentes
  - Factory Method : a pour but d'introduire une méthode abstraite de création d'un objet en reportant aux sous-classes concrètes la création effective
  - Prototype : permet la création de nouveaux objets par duplication d'objets existants appelés prototypes qui disposent de la capacité de clonage
  - **Singleton :** permet de s'assurer qu'une classe ne possède qu'une seule instance, en fournissant une méthode unique retournant cette instance
- Comportement
  - **Chain of Responsibility :** créer une chaîne d'objets telle que si un objet de la chaîne ne peut répondre à une requête, il puisse la transmettre à ses successeurs jusqu'à ce que l'un d'entre eux y répondre

- Command : a pour objectif de transformer une requête en un objet, facilitant des opérations comme l'annulation, la mise en file des requêtes et leur suivi
- Interpreter : fournit un cadre pour donner une représentation par objets de la grammaire d'un langage afin d'évaluer, en les interprétant, des expressions écrites dans ce langage
- *Iterator : fournit un accès séquentiel à une collection d'objets sans que les clients se préoccupent de l'implémentation de cette collection
- Mediator : construit un objet dont la vocation est la gestion et le contrôle des interactions au sein d'un ensemble d'objets sans que ses éléments se connaissent mutuellement
- Memento : sauvegarde et restaure l'état d'un objet
- **Observer :** construit une dépendance entre un sujet et des observateurs de façon à ce que chaque modification de sujet soit notifiée aux observateurs afin qu'ils puissent mettre à jour leur état
- **State :** permet à un objet d'adapter son comportement en fonction de son état interne
- **Strategy :** adapte le comportement et les algorithmes d'un objet en fonction d'un besoin sans changer les interactions avec les clients de cet objet
- Template Method : permet de reporter dans des sous-classes certaines étapes de l'une des opérations d'un objet, ces étapes étant alors décrites dans les sous-classes
- Visitor : construit une opération à réaliser sur les éléments d'un ensemble d'objets. De nouvelles opérations peuvent ainsi être ajoutées sans modifier les classes de ces objets

Catégorisation des Patrons

Les patrons sont découpées en trois grandes catégories :

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

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

Patrons de création : Ils ont pour but d'organiser la création de nouveaux objets. Ces modèles sont très courants pour désigner une classe chargée de construire des objets

Processus d'utilisation

Une fois qu'un Patron est choisi, son utilisation dans une application comprend plusieurs étapes :

- Étudier de façon approfondie sa structure générique qui sert de base pour utiliser le Patron
- Renommer les classes et les méthodes introduites dans la structure générique
- Adapter la structure générique pour répondre aux contraintes de l'application, ce qui peut impliquer des changements dans le schéma d'objets (Il n'y a pas un seul schéma pour chaque Patron mais plusieurs variantes)

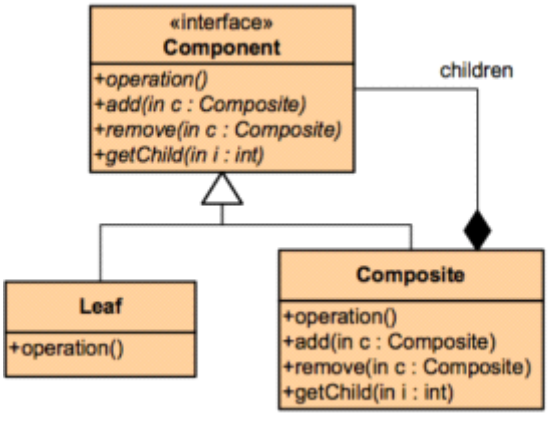
Patrons de structure/conception

Définition

- Les Patrons de conception tentent de composer des classes pour bâtir de nouvelles structures
- Ces structures servent avant tout à ne pas gérer différemment des groupes d'objets et des objets uniques
- Autrement dit, l'objectif est de faciliter l'indépendance de l'interface d'un objet ou d'un ensemble d'objets vis-à-vis de son implémentation (Dans le cas d'un ensemble d'objets, il s'agit aussi de rendre cette interface indépendante de la hiérarchie des classes et de la composition des objets)
- Par exemple : en utilisant un logiciel de dessin vectoriel, tout le monde est amené à grouper des objets
  - Les objets ainsi conçus forment un nouvel objet que l'on peut déplacer, et manipuler sans avoir à répéter ces opérations sur chaque objet qui le compose
  - On obtient donc une structure plus large mais toujours facilement manipulable

Composite : Slide 22

Définition :



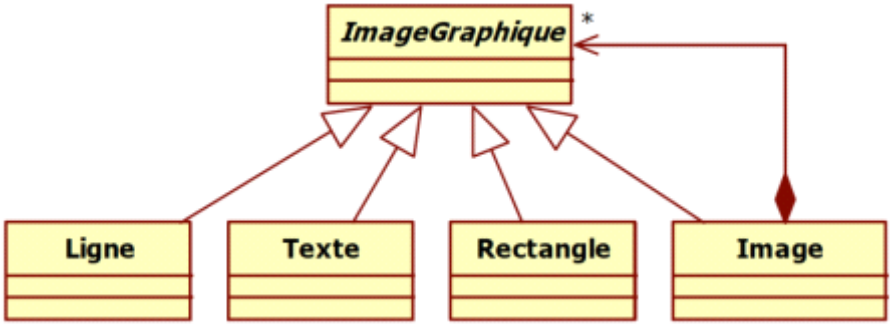
Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

- Composition d'objets dont la profondeur est variable
- Classe A contient plusieurs instances d'elle même
- Il est nécessaire 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

Code :

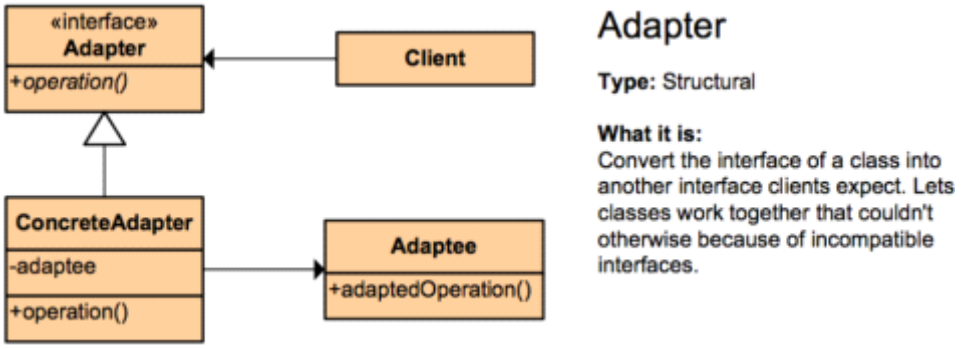



```
// component
import java.awt.Color;
import java.util.*;
public abstract class ImageGraphique {
     protected List<ImageGraphique> elements = new ArrayList<ImageGraphique>();
     Color myColor;
     public void remplisser(Color color) {
          myColor = color;
     }
     public boolean ajouter(ImageGraphique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

// composite
import java.util.*;
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);
          }
     }
}

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

```java
// main
import java.awt.Color;
public class Utilisateur{
	public static void main(String[] args) {
		ImageGraphique ligne = new Ligne();
		ligne.remplisser(Color.red);
		ImageGraphique rectangle = new Rectangle();
		rectangle.remplisser(Color.green);
		ImageGraphique image = new Image();
		image.ajouter(ligne);
		image.ajouter(rectangle);
		ImageGraphique rectangle2 = new Rectangle();
		rectangle2.remplisser(Color.blue);
		ImageGraphique ligne2 = new Ligne();
		ligne2.remplisser(Color.cyan);
		ImageGraphique texte2 = new Texte();
		texte2.remplisser(Color.red);
		ImageGraphique image2 = new Image();
		image2.ajouter(ligne2);
		image2.ajouter(rectangle2);
		image2.ajouter(image);
		image2.ajouter(texte2);
		image2.dessiner(0);
	}
}
```

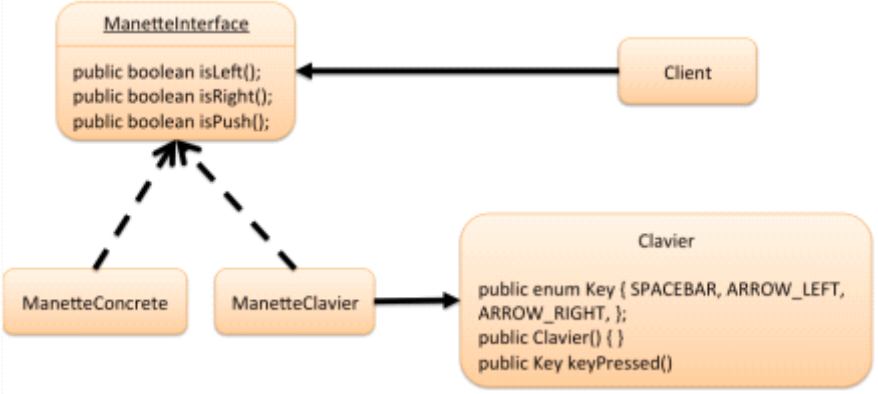
Adapter : Slide 36

Définition :



- Ajout d'une fonctionnalité avec minimum d'impact sur le code existant
- Le but du patron est de convertir l'interface d'une classe existante en l'interface attendue par des clients
- Il s'agit de conférer à une classe existante une nouvelle interface pour répondre aux besoins de clients
- Le patron est utilisé dans les cas suivants :
  - Pour intégrer dans un système un objet dont l'interface ne correspond pas à l'interface requise au sein de ce système
  - Pour fournir des interfaces multiples à un objet lors de sa conception

Code :

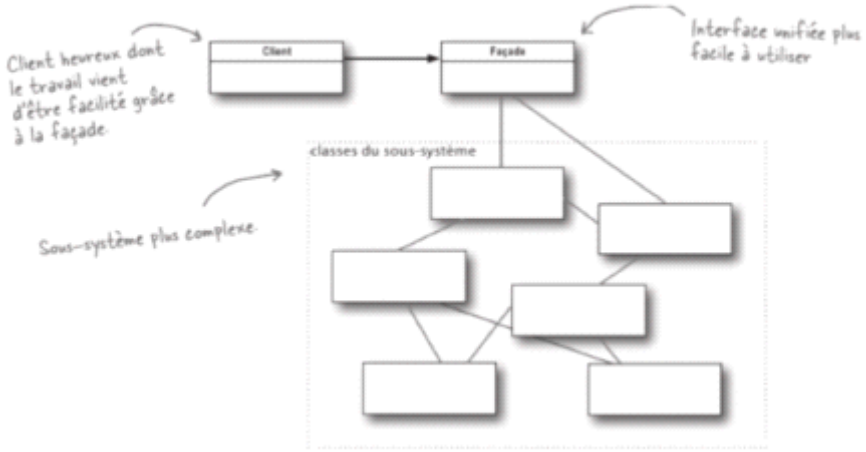



```java
// Adapter
public interface Manette { /* interface pour tous les types de manettes */
	public boolean isLeft();
	public boolean isRight();
	public boolean isPush();
}
```

```
public enum Key { SPACEBAR, ARROW_LEFT, ARROW_RIGHT, /* etc */ };

// Adaptee
public class Clavier {
     public Clavier() { /* constructeur */ }
     public Key keyPressed() {
          return Key.SPACEBAR; /* retourne le type de la touche */
     } /* etc */
}

// Client
public class Jeu {
     private Manette manette;
     public Jeu(Manette manette) { this.manette = manette; }
     public void mainLoop() { /* boucle principale du jeu */
          if(manette.isLeft()) { /* déplacement à gauche */ }
          else if(manette.isRight()) { /* déplacement à droite */ }
          else if(manette.isPush()) { /* appui sur le bouton */}
     }
}

// Concrete Adapter
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;
     }
}

// Old Main
public class JeuMain {
     public static void main(String[] args) {
          Manette manette = new SuperControleur(); /* un certain type de manette */
          Jeu jeu = new Jeu(manette);
          jeu.mainLoop();
     }
}

// Main
public class JeuMain {
     public static void main(String[] args) {
          Manette manette = new ManetteClavier(new Clavier()); /* un certain type de
          manette */
          Jeu jeu = new Jeu(manette);
          jeu.mainLoop();
     }
}
```

Façade : Slide 46

Définition :

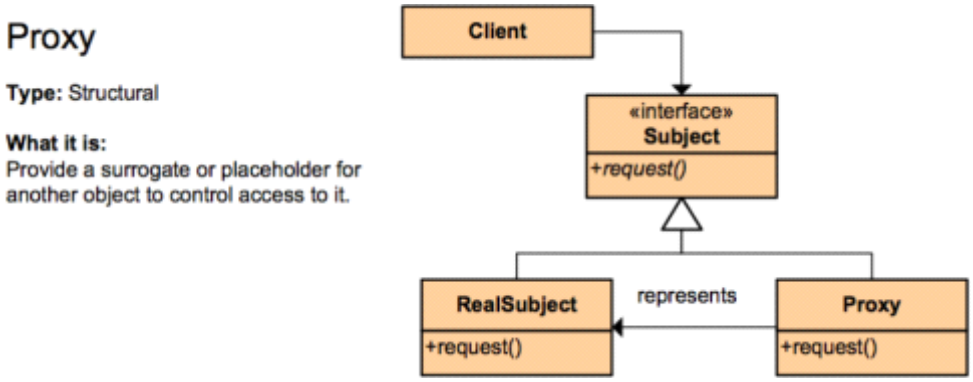


- Le but du patron façade est de permettre à un client d'utiliser de façon simple et transparente un ensemble de classes qui collaborent pour réaliser une tâche courante
- Moyen : Une classe fournit une façade en proposant des méthodes qui réalisent les tâches usuelles en utilisant les autres classes
- Read : C'est exactement un API
- Avantages :

  Découplage
  - Le patron Façade vous permet de découpler l'implémentation de votre client de tout sous-système.
  - Ainsi, si le client dépend bien de la façade et non du sous-système, vous pouvez ajouter de nouveaux composants qui ont des interfaces différentes aux sous-systèmes, sans changer le code client, juste en modifiant la façade
- Comparaison Adaptateur et Façade
  - 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
  - Façade simplifie une interface

Proxy : Slide 54

Définition :



- Le but du Patron Proxy est 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

  Domaine d'application :

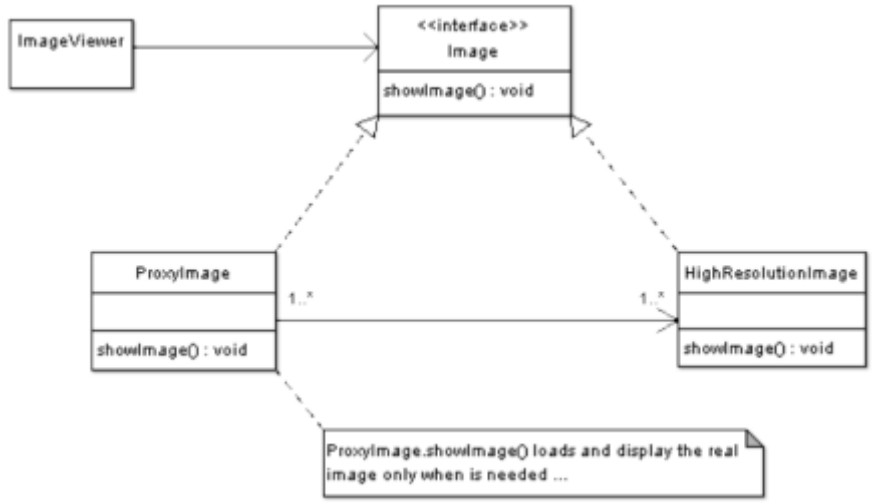
  Le Patron est utilisé en programmation par objets. Il existe différents types de proxy :

  Proxy virtuel : permet de créer un objet de taille importante au moment approprié

  Proxy remote : permet d'accéder à un objet s'exécutant dans un autre environnement. Ce type de proxy est mis en œuvre dans les systèmes d'objets distants (CORBA, Java RMI)

  Proxy de protection : permet de sécuriser l'accès à un objet, par exemple par des techniques d'authentification

Code :




```
public interface Image { //Subject Interface
     public void showImage();
}

public class ImageProxy implements Image { // Proxy
     private String imageFilePath; //Private Proxy data
     private Image proxifiedImage; //Reference to RealSubject
     public ImageProxy(String imageFilePath) {
          this.imageFilePath = imageFilePath;
     }
     @Override
     public void showImage() {
          // create the Image Object only when the image is required to be shown
          proxifiedImage = new HighResolutionImage(imageFilePath);
          // now call showImage on realSubject
          proxifiedImage.showImage();
     }
}

public class HighResolutionImage implements Image { //RealSubject
     public HighResolutionImage(String imageFilePath) {
          loadImage(imageFilePath);
     }

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

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

public class ImageViewer { //Image Viewer program
     public static void main(String[] args) {
          // assuming that the user selects a folder that has 3 images, create the 3
          images
          Image highResoluKonImage1 = new ImageProxy("sample/veryHighResPhoto1.jpeg");
          Image highResoluKonImage2 = new ImageProxy("sample/veryHighResPhoto2.jpeg");
          /* assume that the user clicks on Image one item in a list, this would cause
          the program to call showImage() for that image only */
          highResoluKonImage1.showImage(); /* note that in this case only image one
          was loaded into memory */
          // consider using the high resolution image object directly
          Image highResoluKonImageNoProxy1 = new
          HighResolutionImage("sample/veryHighResPhoto1.jpeg");
          Image highResoluKonImageNoProxy2 = new
          HighResolutionImage("sample/veryHighResPhoto2.jpeg");
          // assume that the user selects image two item from images list
          highResoluKonImageNoProxy2.showImage();
```

```
                /* note that in this case all images have been loaded into memory and not
                all have been actually displayed, this is a waste of memory resources. */
            }
        }
```

Patrons de création

Définition

- On se trouve en programmation objet souvent confronté au problème d'évolution des classes
- Une classe hérite d'une autre classe pour en spécialiser certains éléments
- On aimerait donc qu'un objet puisse appartenir à telle ou telle classe (dans une même famille par héritage) sans avoir à chercher la classe de gestion de ces objets et la ligne de code qui effectue l'instanciation

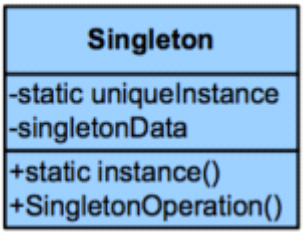
Singleton : Slide 66

Définition :

Singleton

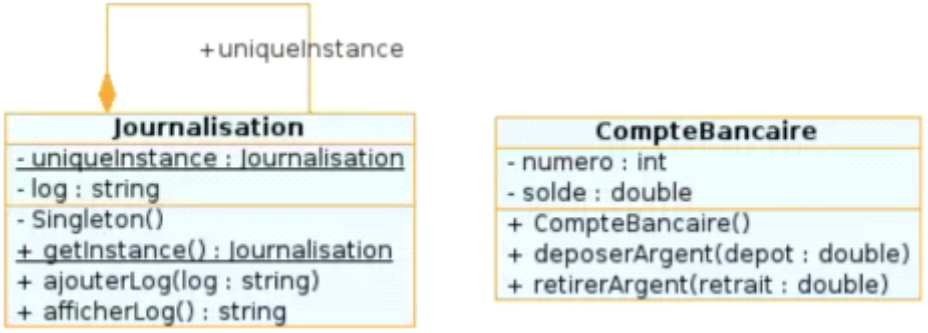
Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.



- Garantir que le programme n'utilise qu'une seule instance d'une classe
- Le but du patron Singleton est d'assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode unique retournant cette instance
- Le patron singleton est utilisé dans le cas suivant :
  - 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

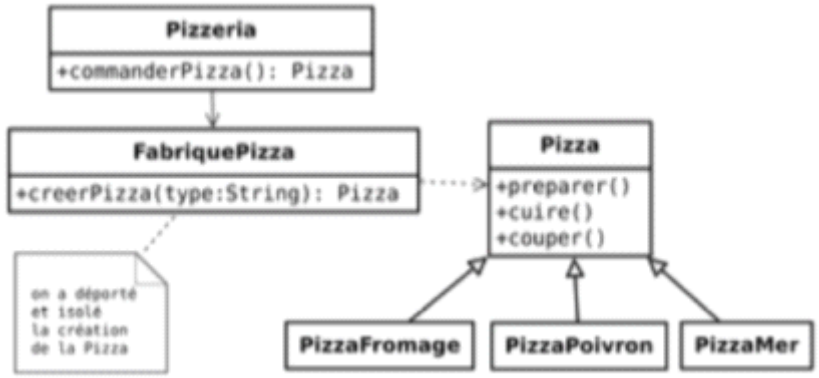
Code :




```
/* Classe basée sur le pattern Singleton qui permet la journalisation de
l'application. */
public class Journalisation {
     // Stockage de l'unique instance de cette classe.
     private static Journalisation uniqueInstance;
     // Chaine de caractères représentant les messages de log.
     private String log;
     // Constructeur en privé (donc inaccessible à l'extérieur de la classe).
     private Journalisation() { log = new String(); }
     /* Méthode statique qui sert de pseudo-constructeur (utilisation du mot clef
     "synchronized" pour le multithread). */
     public static synchronized Journalisation getInstance() {
          if(uniqueInstance==null) { uniqueInstance = new Journalisation(); }
          return uniqueInstance;
     }
     // Méthode qui permet d'ajouter un message de log.
     public void ajouterLog(String log) {
          // On ajoute également la date du message.
          Date d = new Date();
          DateFormat dateFormat = new SimpleDateFormat("dd/MM/yy HH'h'mm");
          this.log+="["+dateFormat.format(d)+"] "+log+"\n";
     }
     // Méthode qui retourne tous les messages de log.
     public String afficherLog() {return log;}
}
```

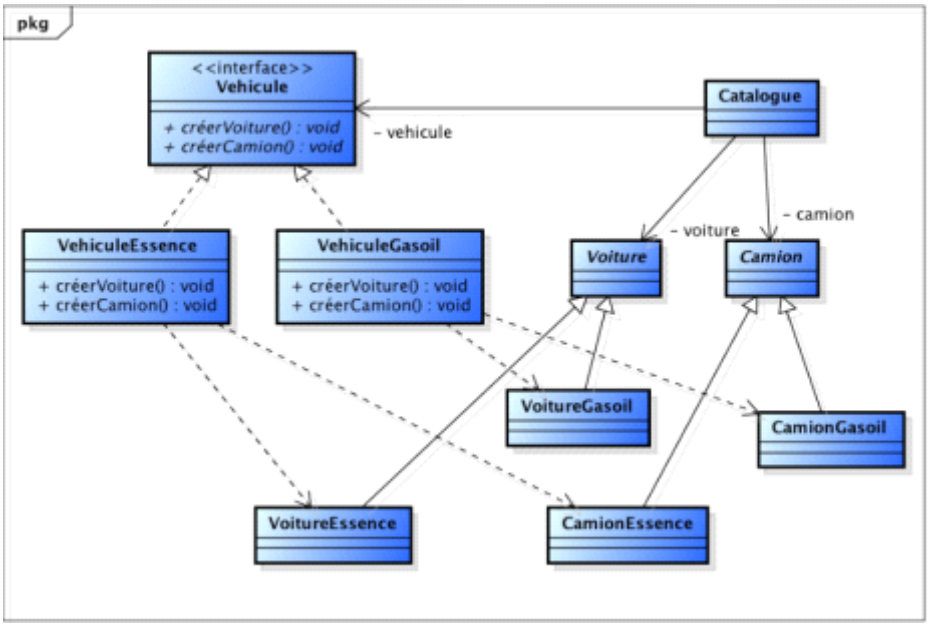
Abstract Factory : Slide 74

Définition :



- But : Permettre à un client de créer des objets sans savoir leur type précis
- Moyen : Les produits sont représentés par une classe abstraite dont dérivent toutes les classes d'objets. Une usine abstraite permet aux clients de fabriquer des produits. Chaque usine instanciée crée un produit de type concret correspondant
- Les classes doivent être ouverte à l'extension mais fermées à la modification
  - Principe de fermeture à la modification : on a déporté la création concrète des pizzas dans un autre classe pour ne pas modifier Pizzeria si d'autres types de pizzas étaient ajoutés ou certains types éliminés
- Dépendez d'abstractions ne dépendez pas de classes concrètes
  - Principe d'inversion des dépendances : on évite un couplage fort

Code :




```
//
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);
	}
}


//
```

```
public class VoitureEssence extends Voiture {
     public void afficher(){
          System.out.println("Voiture à Essence de modèle : " + modele + " de
          puissance " + puissance);
     }
}

//
public abstract class Camion {
     protected int categorie;
     protected int puissance;
     public Camion(int categorie, int puissance) {
          this.categorie = categorie;
          this.puissance = puissance;
     }
     public abstract void afficher();
}

//
public class CamionGasoil extends Camion {
     public void afficher(){
          System.out.println("Camion à gasoil de catégorie : " + modele + " de
          puissance " + puissance);
     }
}

//
public class CamionEssence extends Camion {
     public void afficher(){
          System.out.println("Camion à Essence de catégorie : " + modele + " de
          puissance " + puissance);
     }
}

//
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)
     };
     Camion creerCamion(int categorie, int puissance) {
          return new CamionGasoil(categorie, puissance)
     };
}

//
public class FabriqueVehiculeEssence implements Vehicule {
     Voiture creerVoiture(String modele, int puissance) {
          return new VoitureEssence(modele, puissance)
     };
     Camion creerCamion(int categorie, int puissance){
          return new CamionEssence(categorie, puissance)
     };
}

//
import java.util.*;
public class catalogue {
     public static void main(String[] args){
          FabriqueVehicule fabrique;
          Scanner reader = new Scanner(System.in);
          System.out.println(" Voulez-vous utiliser des véhicules gasoil \(1\) ou
          essence \(2\)" );
          String choix = reader.next();
          If (choix.equals("1"))
```

```
                fabrique = new FabriqueVehiculeGasoil();
            else
                fabrique = new FabriqueVehiculeEssence();
            Voiture v = fabrique.creerVoiture("X", 2);
            Camion c = fabrique.creerCamion(1, 6);
            v.afficher();
            c.afficher();
        }
    }
```

Patrons de comportement

Définition

Simplifient l'organisation d'exécution des objets

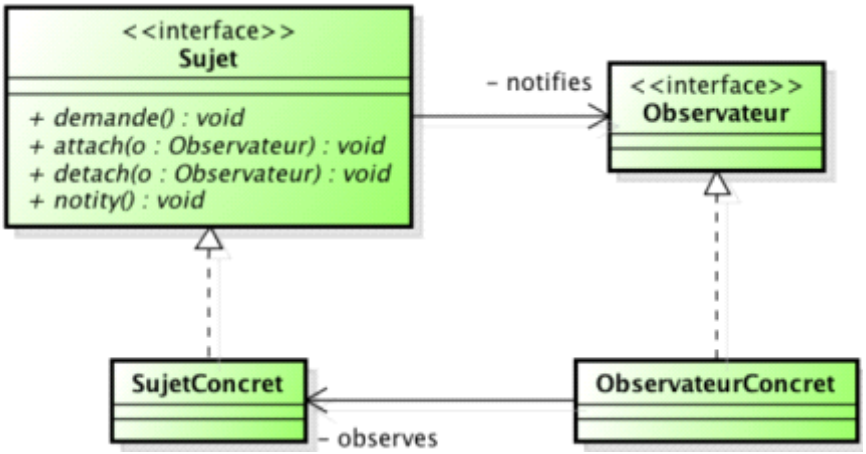
Typiquement, une fonction est composée d'un ensemble d'actions qui parfois appartiennent à des domaines différents de la classe d'implémentation

On aimerait donc pouvoir "déléguer" certains traitements à d'autres classes

D'une manière générale, un modèle de comportement permet de réduire la complexité de gestion d'un objet ou d'un ensemble d'objets

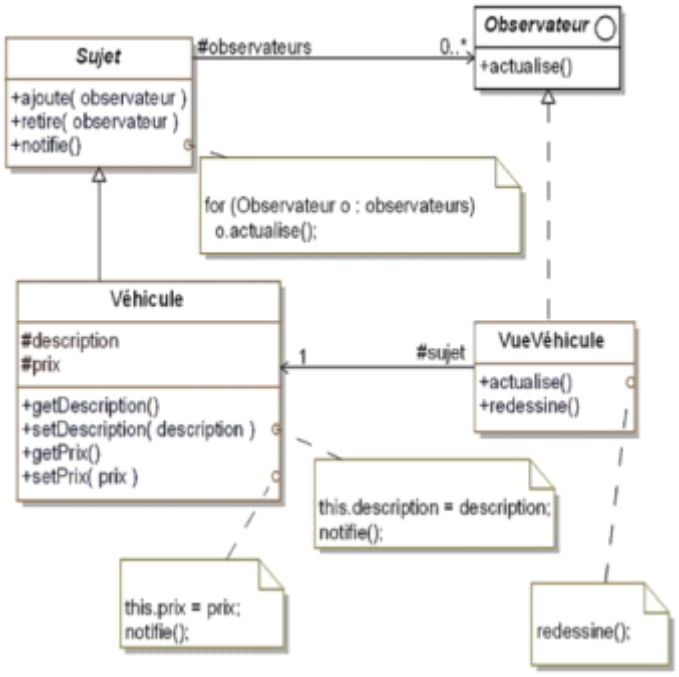
Observer : Slide 90

Définition :



- 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
- Le patron Observer est utilisé dans les cas suivants :
  - Une modification dans l'état d'un objet engendre des modifications dans d'autres objets qui sont déterminés dynamiquement
  - Un objet veut prévenir d'autres objets sans devoir connaître leur type, c'est-à-dire sans être fortement couplé à ceux-ci
  - On ne veut pas fusionner deux objets en un seul
- Pouvoir du couplage faible :
  - Lorsque deux objets sont faiblement couplés, il peuvent interagir sans pratiquement se connaître
  - Le pattern Observer permet une conception dans laquelle le couplage entre Sujet et Observateur est faible
    - Le sujet ne sait qu'une seule chose à propos de l'Observateur : il implémente une certaine interface (interface observateur)
    - Possibilité d'ajout d'autres observateurs à tout moment
    - Pas besoin de modifier le sujet pour ajouter de nouveaux types d'observateurs
    - Possibilité de réutiliser le sujet et les observateurs indépendamment les uns des autres (couplage faible)
    - Les modifications des sujet n'affectent pas les observateurs et inversement (seule obligation : les objets doivent continuer à implémenter les interfaces)

Code :

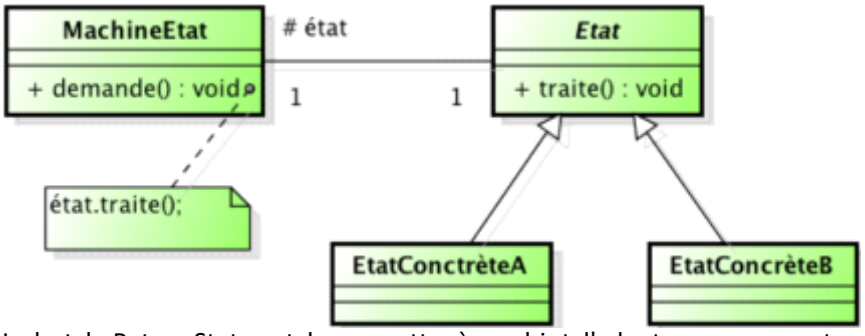



```
import java.util.*;
public abstract class Sujet {
    protected List<Observateur> observateurs = new ArrayList<Observateur>();
    public void ajoute(Observateur observateur) {
        observateurs.add(observateur);
    }
    public void retire(Observateur observateur) {
        observateurs.remove(observateur);
    }
    public void notifie() {
        for (Observateur observateur : observateurs)
            observateur.actualise();
    }
}

public class Vehicule extends Sujet {
    String description;
    Double prix;
    public String getDescription() {
        return description;
    }
    public void setDescription(String desription) {
        this.description = description;
        notifie();
    }
    public Double getPrix() {
        return prix;
    }
    public void setPrix(Double prix) {
        this.prix = prix;
        notifie();
    }
}

public interface Observateur {
    void actualise();
}

public class VueVehicule implements Observateur {
    protected Vehicule vehicule;
    protected String texte = "";
    public VueVehicule(Vehicule vehicule) {
        this.vehicule = vehicule;
        vehicule.ajoute(this);
        actualise();
    }
```

```
        public void actualise() {
            texte = "Description " + vehicule.getDescription() + " Prix : " +
    vehicule.getPrix();
        }
        public void affiche() {
            System.out.println(texte);
        }
    }

    public class Utilisateur {
        public static void main(String[] args) {
            Vehicule vehicule = new Vehicule();
            vehicule.setDescription("Véhicule bon marché");
            vehicule.setPrix(5000.0);
            VueVehicule vueVehicule = new VueVehicule(vehicule);
            vueVehicule.affiche();
            vueVehicule vueVehicule2 = new VueVehicule(vehicule);
            vehicule.setPrix(5500.0);
            vueVehicule.affiche();
            vueVehicule2.affiche();
        }
    }
```

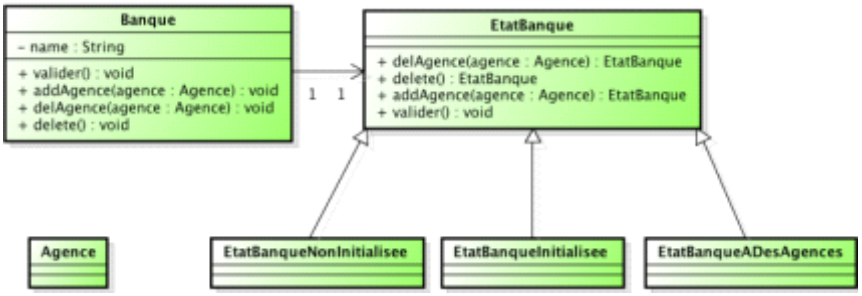
State : Slide 104

Définition :



- Le but du Patron State est de permettre à un objet d'adapter son comportement en fonction de son état interne
- Le patron State est utilisé dans le cas suivant :
  - Le comportement d'un objet dépend de son état
  - L'implantation de cette dépendance à l'état par des instructions conditionnelles est trop complexe

Code :




```
import java.util.*;
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();
     }
     public void addAgence(Agence agence) {
          etat.addAgence(agence);
     }
     public void delAgence(Agence agence) {
          etat.delAgence(agence);
```

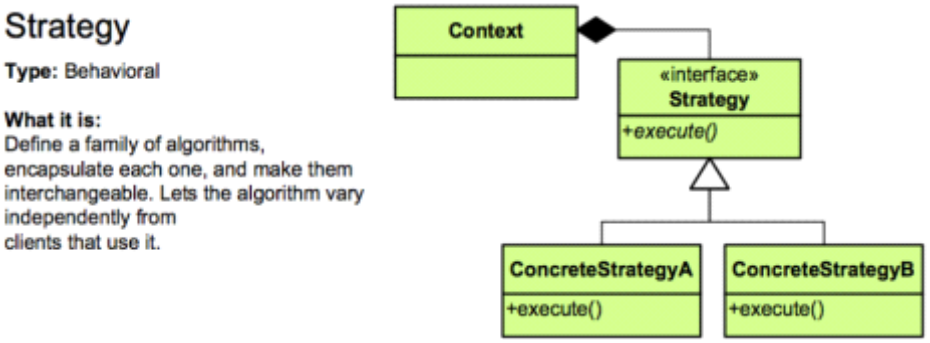
```
        }
        public void delete() {
            etat.delete();
        }
        public Collection getAgences() {
            return agences;
        }
        public String getNom() {
            return nom;
        }
        public void setAgences(CollecEon collection) {
            agences = collection;
        }
        public void setNom(String string) {
            nom = string;
        }
        void setEtat(EtatBanque etat) {
            this.etat = etat;
        }
}

public class Agence {
        public Agence() {
            super();
        }
}

public interface EtatBanque {
        // Suppression d'une agence de la banque
        public void delAgence(Agence agence);
        // Efface la banque
        public void delete();
        //Ajoute une agence à la banque
        public void addAgence(Agence agence);
        //valide les informations de la banque
        public void validate();
}

public class EtatBanqueNonInitialisee implements EtatBanque {
        private final Banque banque;
        public EtatBanqueNonInitialisee(Banque banque) {
            if (banque == null) {
                throw new IllegalArgumentException("état d'une banque est associe a la
                banque qu'il représente");
            }
            this.banque = banque;
        }
        public void delAgence(Agence agence) { //ne peut pas être prise en compte dans
        cet état
            throw new IllegalStateException("L'agence n'est pas initialisée, impossible
            de supprimer des agences");
        }
        public void delete() {
            throw new IllegalStateException("L'agence n'est pas initialisée, impossible
            de supprimer la banque");
        }
        public void addAgence(Agence agence) {
            throw new IllegalStateException("L'agence n'est pas initialisée, impossible
            d'ajouter des agences");
        }
        public void validate() {
            if (null == banque.getNom()) {// correspond à la garde décrite sur la
            transition
                throw new IllegalStateException("Le nom de banque doit être non null");
            }
            banque.setEtat(new EtatBanqueInitialisee(banque));
        }
} //il ne reste plus qu'à changer d'état
```

```
public class EtatBanqueInitialisee implements EtatBanque {
     private final Banque banque;
     public EtatBanqueInitialisee(Banque banque) {
          if (banque == null) {
               throw new IllegalArgumentException("état d'une banque est associe a la
               banque qu'il représente");
          }
          this.banque = banque;
     }
     public void delAgence(Agence agence) { /* Pas d'agence enregistrée donc
     impossible d'en supprimer */
          throw new IllegalStateException("La banque ne contient pas d'agence,
          impossible d'en supprimer");
     }
     public void delete() { /* Nous n'avons pas d'agence, il est donc possible de
     supprimer cette banque */
          // code de suppression de la banque
     }
     public void addAgence(Agence agence) { // Ajoutons donc cette agence
          if (agence == null) {
               throw new IllegalArgumentException("Impossible d'ajouter une agence
               nulle");
          }
          banque.getAgences().add(agence);
          banque.setEtat(new EtatBanqueADesAgences(banque));
     } // et la transition
     public void validate() { //On à déjà validé cette banque c'est assez inutile de
     recommencer
     }
}

public class EtatBanqueADesAgences implements EtatBanque {
     private final Banque banque;
     public EtatBanqueADesAgences(Banque banque) {
          if (banque == null) {
               throw new IllegalArgumentException("état d'une banque est associé à la
               banque qu'il représente");
          }
          this.banque = banque;
     }
     public void delAgence(Agence agence) { /* Suppression d'une agence et
     éventuellement retour dans l'état initialisé */
          if (agence == null) {
               throw new IllegalArgumentException("Impossible de supprimer un agence
               vide");
          }
          if (! banque.getAgences().contains(agence)) { // c'est pas mon agence
               throw new IllegalArgumentException("Tentative de suppression d'une
               agence qui n'est pas associee a cette banque");
          }
          banque.getAgences().remove(agence);
          if (banque.getAgences().isEmpty()) { // retour à l'état initialisé
               banque.setEtat(new EtatBanqueIniKalisee(banque));
          }
     }
     public void delete() { //effacement de la banque impossible depuis cet état
          throw new IllegalStateException("Impossible de supprimer une banque qui
          contient encore des agences");
     }
     public void addAgence(Agence agence) { //ajout d'une agence
          if (null == agence) { // je n'ajoute que des agences qui soient
          effectivement des agences
               throw new IllegalArgumentException("Impossible d'ajouter une instance
               vide d'agence");
          }
          banque.getAgences().add(agence);
     }
     public void validate() {//On à déjà valider cette banque c'est assez inutile de
     recommencer
     }
```

```
}

public class EtatBanqueADesAgences implements EtatBanque {
     private final Banque banque;
     public EtatBanqueADesAgences(Banque banque) {
          if (banque == null) {
               throw new IllegalArgumentException("état d'une banque est associé à la
               banque qu'il représente");
          }
          this.banque = banque;
     }
     public void delAgence(Agence agence) { //Suppression d'une agence et
     éventuellement retour dans l'état initialisé
          if (agence == null) {
               throw new IllegalArgumentException("Impossible de supprimer un agence
               vide");
          }
          if (!banque.getAgences().contains(agence)) { // c'est pas mon agence
               throw new IllegalArgumentException("Tentative de suppression d'une
               agence qui n'est pas associee a cette banque");
          }
          banque.getAgences().remove(agence);
          if (banque.getAgences().isEmpty()) { // retour à l'état initialisé
               banque.setEtat(new EtatBanqueInitialisee(banque));
          }
     }
     public void delete() { //effacement de la banque impossible depuis cet état
          throw new IllegalStateException("Impossible de supprimer une banque qui
          contient encore des agences");
     }
     public void addAgence(Agence agence) { //ajout d'une agence
          if (null == agence) { // je n'ajoute que des agences qui soient
          effectivement des agences
               throw new IllegalArgumentExcepKon("Impossible d'ajouter une instance
               vide d'agence");
          }
          banque.getAgences().add(agence);
     }
     public void validate() {//On à déjà valider cette banque c'est assez inutile de
     recommencer
     }
}
```

Strategy : Slide 116

Définition :



- But : Le patron Strategy a pour objectif d'adapter le comportement et les algorithmes d'un objet avec les clients
- Ce besoin peut relever de plusieurs aspects comme de aspect de présentation, d'efficacité en temps ou en mémoire, de choix d'algorithmes, de représentation interne, etc.
- Les interactions entre l'objet et ses clients restent inchangées
- Le patron Strategy est utilisé dans les cas suivants :
  - 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 indentiques 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

Anti-patterns : Slide 122

- Abstraction inverse : Offre des interfaces qui implémentent des méthodes plus complexes dont on a pas besoin (utilisation de calcul flottant dur les entiers)
- Action à distance : Utilisation de variables globales et interdépendances accrues entre objets
- Ancre de bateau : Composant inutilise mais garde pour des raisons politiques (peut être utilises plus tard)
- Attente active : Définition d'une boucle avec une seule instruction ou on attend qu'une variable change d'état pour continuer , c'est déconseille car on gaspille les ressources (il faut utiliser la programmation évènementielle ou utiliser des signaux)
- Interblocage et famine : qd beaucoup de process tente d'accéder à une même ressource
- Erreur de copier/coller : duplication de code sans vérification (solution : factoriser le code au lieu de dupliquer)
- Programmation spaghetti : programme impossible à maintenir

Lasse9

dimanche 5 juillet 2020 13:00

Effet spaghetti : Concepteur finit par être saturé par la complexité du codage.

Définition du patron de conception ou de modèle de conception

- Un schéma a objet décrit par des classes et des relations qui forme une solution a un problème connu et fréquent (Solutions connues et prouves par les programmeurs)

Catégories patron de conception

- Patrons de structure : Ils servent à faciliter l'organisation des classes et leurs relations (Construire des agglomérations de classes)
- Patrons de comportements : Servent à contrôler les interaction entre les objets (repartir les responsabilités entre classes)
- Patrons de création : Organiser la création des objets (Designer une classe pour construire les objets)

Principes de la 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 a l'héritage

Patron de conception

Patron composite

- un patron composite est un patron de conception (design pattern) structurel1. Ce patron permet de concevoir une structure arborescente, par exemple une représentation d'un dossier, ses sous-dossiers et leurs fichiers.
- Il y'a Feuille / Compose / Composant
- Exemple :

  Une image est compose d'autres images ou de feuilles comme Ligne, rectangle

Patron Adapter

Le but du patron est de convertir l'interface d'une classe existante en l'interface attendue par les clients

Composants :

- Interface (Manette) : interface d'origine
- Adaptateur (ManetteClavier) = implémente l'interface d'origine, tout en invoquant les méthodes de l'objet Adaptee
- Adaptee (Clavier) : introduit l'objet dont l'interface doit être adaptée pour correspondre à Interface.
- Client : interagi avec les objets correspondant à l'interface

Patron Facade

Définition :

- Le but du patron façade est de permettre à un client d'utiliser de façon simple et transparente un ensemble de classe qui collaborent pour réalise une tache courante

Composants :

- interface nouvelle qui communique avec le client
- classe ou on gère le sous système
- Client : interagi avec les objets correspondant à l'interface

Exemple :

- Au lieu d'utiliser tout un système pour regarder un film (Création de classes Film, Lecteur, Popcorn et d'invocation de méthodes directement on crée une interface qui contient des méthodes simples pour exécuter le sous-système, et exécuter le sous-système à part dans une autre classe abstraite), sa différence avec ADAPTER est qu'elle crée une nouvelle interface et non une qui dérive de l'ancienne

Patron proxy

Définition :

- Conception d'objets qui se substitue à un autre objet et qui en contrôle l'accès, l'objet qui effectue la subissions possède la même interface que l'objet réel (substitution transparente vis à vis du client)

Composant :

- Subjet (Interface)
- Proxy (Implémentant l'interface Sujet)
- RealSubject (Implémentant l'interface Subject et dont le proxy contrôle l'accès)
- Client : interagi avec les objets correspondant à l'interface

Exemple :

- RMI, Cobra, ...

Patron de création

Patron Singleton

- Assurer qu'une classe ne possède qu'une seule instance et de fournir une méthode de classe (Méthode statique) unique retournant cette instance, Il offre la possibilité de ne pas utiliser les variables globales (Utilisation d'un

constructeur prive), si dans la méthode de classe statique l'attribut contenant l'objet est nulle , on crée l'objet)

- Composants :
  - Singleton (exemple classe journalisation)
- Exemple de code :

```
public class Journalisation {
     private static Journalisation uniqueInsatance;
     private Journalisation() {
     }
     public static Journal getInstance() {
          if(uniqueInstance == null) uniqueInstance = new Journalisation();
          else return uniqueInstance;
     }
}
```

Patron Abstract Factory

- Permettre à un client de créer des objets sans savoir leurs types précis, dépendre d'abstraction et non de classe concrète (principe d'inversion de dépendances)
- Composants :
  - FabriqueAbstaraite : interface spécifiant les signatures des méthodes créant les différentes méthodes
  - FabriqueConcrete1, FabriqueConcrete2 : sont les classes concrètes implantant les méthodes créant les produits pour chaque type de famille de produit
  - ProduitAbstrait1, ProdutAbstrait2 : Classe abstraite des produites indépendamment de leurs famille
  - Produit_1, Produit_2, ... , Produit_n

Patron de Comportement

Définition

D'une manière générale, un modèle de comportement permet de réduire la complexité de gestion d'un objet ou d'un ensemble d'objet

Patron observer

Définition :

Construire une dépendance entre les sujets et les observateurs de sorte que chaque modification du sujet soit notifié aux différents observateurs, afin de mettre à jour leur état (Notifier les Observers sans au préalable connaitre leurs types , interface Observateur) => C'est le principe du couplage faible (Interaction entre objets sans)

Composition :

Sujet : est la classe abstraite (ou interface) qui introduit l'association avec les observateurs ainsi que les méthodes pour ajouter, retirer et notifier les observateurs
Observateur : interface a implémenter pour recevoir des notifications
Sujet concret : classe d'implémentation de sujet, qui contient des références et les notifie en cas de modification
Observateur concret : a une référence vers son sujet, et demande des infos à son sujet (sujet.getEtat()) pour modifier une vue externe

Patron State :

Définition :

Permettre à un objet d'adopter son comportement en fonction de état interne

Composition :

MachineEtat : classe concrète décrivant des objets qui sont des machines à état => possèdent un ensemble d'état
État : Classe abstraite qui introduit la signature des méthodes gérant l'état des machines à état
EtatConcretA, EtatConcretB,.... : sous classes concrètes implémentant les méthodes de chaque état.

Utilisation :

Comportement d'un objet dépend de son état
Implantation de cette état par des instructions conditionnelles est complexe.

Patron Strategy

Définition :

- Définie une famille d'algorithme, ou on encapsule chacun, et on les rend interchangeables, l'algorithme varie en fonction du client qui l'utilise.

Composition :

- Strategy = Interface implémentant les signatures des algorithmes
- ConcreteStrategyA, ConcreteStrategyB,.... : Sous classes concretes qui implémentent les différents algorithmes
- Context : Classe qui fait appel aux Strategies par référence

Utilisation :

- Implémentation par instructions conditionnelles est trop complexes
- Système se composant de classes identiques à l'exception de leurs comportements

Anti_patron :

- Abstraction inverse : Offre des interfaces qui implémentent des méthodes plus complexes dont on a pas besoin (utilisation de

calcul flottant dur les entiers)

- Action à distance : Utilisation de variables globales et interdépendances accrues entre objets
- Ancre de bateau : Composant inutilise mais garde pour des raisons politiques (peut être utilises plus tard)
- Attente active : Définition d'une boucle avec une seule instruction ou on attend qu'une variable change d'état pour continuer, c'est déconseillé car on gaspille les ressources (il faut utiliser la programmation évènementielle ou utiliser des signaux)
- Interblocage et famine : qd beaucoup de process tente d'accéder à une même ressource
- Erreur de copier/coller : duplication de code sans verification (solution : factoriser le code au lieu de dupliquer)
- Programmation spaghetti : programme impossible à maintenir