Université Abdelhamid Ibn Badis Mostaganem
Faculté des sciences exactes et informatique

Ingénierie Dirigée par les Modèles (IDM)

Design Patterns

Introduction

- **Ingénierie Dirigée par les Modèles** est une pratique d'ingénierie des systèmes qui permet de **mécaniser** le processus de développement logiciel qui se fait souvent à la main en décrivant au travers de **modèles**, concepts, et langages, le problème posé et sa solution.
- MDA (Architecture dirigée par les modèles) est une approche d'IDM qui repose sur l'idée de génération de code source à partir de **modèles indépendamment** ou non du **langage de programmation** et des **plateformes d'execution**
  - En se basant sur des **niveaux d'abstraction (**méta-modélisation**)**
  - En trouvant des techniques de **transformation** et de raffinement des modèles.
- **Objectif** IDM: améliorer la productivité logiciel (la génération et la réutilisabilité du code source), sa portabilité et intéropérabilité.

Introduction

- IDM est une pratique qui repose historiquement sur des progressions permettant d'améliorer la productivité logicielle:
  - la programmation orientée-objet,
  - puis, les **design patterns**
  - et ensuite le développement orienté modèle (Model Driven Development - MDD)

Objectif du cours

- Savoir qu'est-ce qu'un patron de conception?
- Apprendre quelques patrons de conception les plus utilisés
- Savoir implémenter ces patrons à l'aide d'un langage de programmation
- **Pourquoi?**
  - Indispensable pour le développement professionnel
  - C'est parmi les connaissances de base demandées par les entreprises lors des tests techniques pour le recrutement !

Design Patterns

Design patterns (patrons de conception)

- Les **Patrons de Conception** sont des solutions de conception de logiciel établies à partir des experiences des développeurs et experts durant plusieurs années.
- Origine de l'idée de patron de conception vient du domaine d'architecture

Pourquoi les Design Patterns ?

- Rendre disponible et explicite des pratiques de bonne conception
- Capturer un savoir faire, le rendre pérenne réutilisable, etc.
- Nommer et rendre explicite une structure de haut niveau qui n'est pas directement exprimable sous forme de code
- Créer un vocabulaire commun pour les développeurs et les concepteurs
- Ces modèles sont définis pour pouvoir être utilisés avec un maximum de langages orientés objets.

Catégories de Design Patterns

- Il existe trois grandes catégories de patrons de conception:
  - **les modèles de création** (creational patterns)
  - **les modèles de structuration** (structural patterns)
  - **les modèles de comportement** (behavioral patterns)
- Le motif de conception le plus connu est le modèle **MVC** (Model View Controller) mis en oeuvre en premier avec SmallTalk.
- Ces patrons de conception peuvent êtres aussi combinés

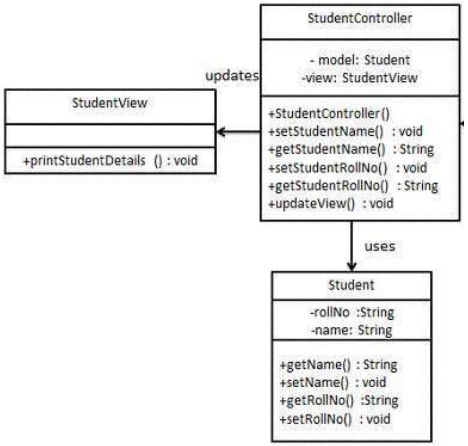
MVC

- MVC (Model-View-Controller) est utilisé pour modulariser les applications en:
  - **Model (Modèle)** - Représente les objets représentants le modèles de données
  - **View (Vue)**- C'est la visualisation des données que le modèle contient.
  - **Controller (Contrôleur)** - Il permet la liaison entre le modèle et la vue. Il permet de contrôler le flot de données et modifier la vue en cas de changement de données.

MVC

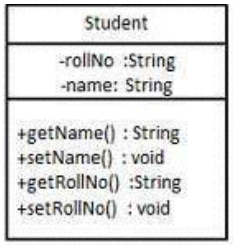
Exemple d'implémentation

- **Student** qui représente le modèle.
- **StudentView** représente la vue qui permet d'afficher les informations sur l'étudiant sur la console.
- **StudentController** est le contrôleur responsable sur l'enregistrement de données dans l'objet Student et modifier la vue.

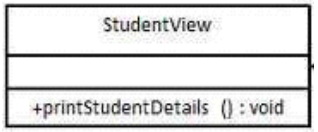


Etape 1: Le modèle

```
public class Student {

   private String rollNo;
   private String name;

   public String getRollNo() {
      return rollNo;
   }

   public void setRollNo(String rollNo) {
      this.rollNo = rollNo;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```



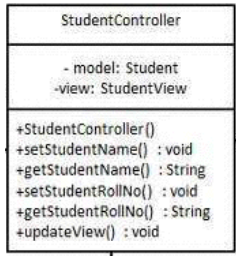
Etape 2: La vue



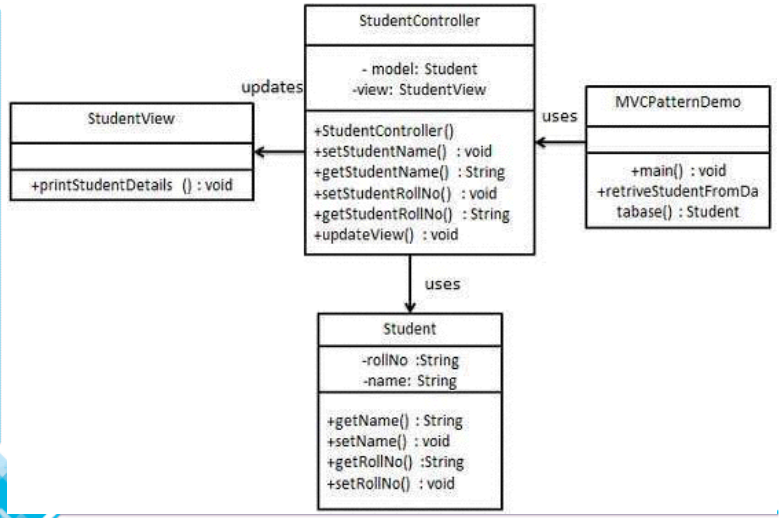

```
public class StudentView {

   public void printStudentDetails(
               String studentName,
               String studentRollNo){

      System.out.println("Student: ");
      System.out.println("Name: " + studentName);
      System.out.println("Roll No: " +
        studentRollNo);
   }
}
```

Etape 3: Le contrôleur

```
public class StudentController {
   private Student model;
   private StudentView view;

   public StudentController(Student model, StudentView view){
      this.model = model;
      this.view = view;
   }
   public void setStudentName(String name){
      model.setName(name);
   }
   public String getStudentName(){
      return model.getName();
   }
   public void setStudentRollNo(String rollNo){
      model.setRollNo(rollNo);
   }
   public String getStudentRollNo(){
      return model.getRollNo();
   }
   public void updateView(){
      view.printStudentDetails(model.getName(), model.getRollNo());
   }
}
```



Etape 4: Le main (utilisation)



Etape 4: Le main (utilisation)

```
public class MVCPatternDemo {
   public static void main(String[] args) {

      //fetch student record based on his roll no from the database
      Student model  = retriveStudentFromDatabase();

      //Create a view : to write student details on console
      StudentView view = new StudentView();

      StudentController controller = new StudentController(model, view);
      controller.updateView();

      //update model data
      controller.setStudentName("John");
      controller.updateView();
   }
   private static Student retriveStudentFromDatabase(){
      Student student = new Student();
      student.setName("Robert");
      student.setRollNo("10");
      return student;
   }
}
```


Design Patterns : Créateurs (*creational patterns*)

Patrons Créateurs

- Dans cette catégorie, on trouve par exemple les patrons:
  - Factory (Fabrique)
  - Abstract Factory (Fabrique abstraite)
  - Singleton (Singleton)

Patrons Créateurs

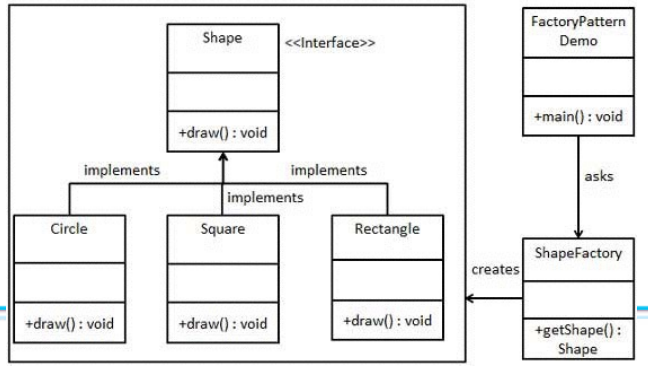
- Dans cette catégorie, on trouve par exemple les patrons:
  - **Factory (Fabrique)**
  - Abstract Factory (Fabrique abstraite)
  - Singleton (Singleton)

Factory

- Le patron Factory est parmi les patrons les plus utilisés en orienté objet, en particulier en Java.
- Ce type de patron donne une meilleure façon pour créer un objet
- Dans ce patron, on crée l'objet sans exposer la logique de création au client et on fait référence à l'objet en utilisant une interface commune

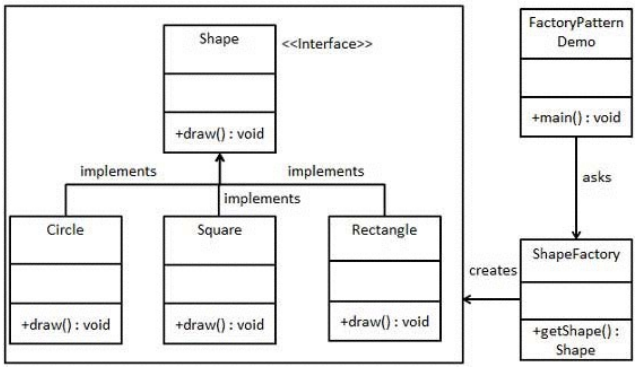
Factory

- **Exemple d'implementation**
- On crée une **interface** Shape et des classes qui implémentent cette interface.
- Une **classe factory** ShapeFactory est défini ensuite
- Main utilise ShapeFactory pour obtenir un objet Shape. Main passera l'information (CIRCLE / RECTANGLE / SQUARE) à ShapeFactory pour obtenir le type d'objet souhaité.



Etape 1: Créer une interface

```
public interface Shape {
   void draw();
}
```



Etape 2: créer les classes qui impémentent cette interface

```
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}

public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}
```


Etape 3: Créer la classe Factory qui génère des objets en se basant sur l'information donnée

```
public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}
```


Etape 4: Utiliser la classe Factory dans le main pour obtenir des objets Shape

```
public class FactoryPatternDemo {

   public static void main(String[] args) {

      ShapeFactory shapeFactory = new ShapeFactory();

      Shape shape1 = shapeFactory.getShape("CIRCLE");
      shape1.draw();

      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      shape2.draw();

      Shape shape3 = shapeFactory.getShape("SQUARE");
      shape3.draw();
   }
}
```

Resultat

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

- **Quand utiliser ce patron?**
  - Quand on a besoin de standardiser le modèle architectural pour un ensemble d'applications
  - Au même temps, on veut permettre à des applications individuelles de définir elles-mêmes leurs propres objets à créer

Patrons Créateurs

- Dans cette catégorie, on trouve par exemple les patrons:
  - Factory (Fabrique)
  - **Abstract Factory** (Fabrique abstraite)
  - Singleton (Singleton)

Abstract Factory

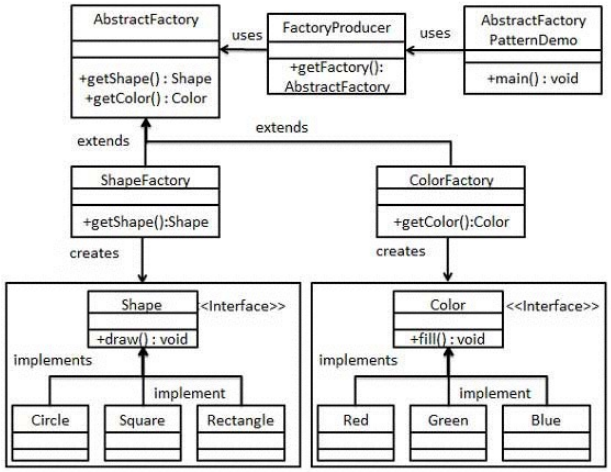
- Le pattern Abstract Factory permet la création d'une super-factory qui créé d'autres factories (factory of factories)
- **But** : permettre de créer des familles de produits en masquant les mécanismes de choix des classes de mise en œuvre de ces produits
- **Quand l'utiliser?**
  - un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
  - un système repose sur un produit d'une famille de produits
  - on veut définir une interface unique à une famille de produits concrets

Abstract Factory

- **Exemple:** On veut développer une application graphique multiplateformes
  - il existe une bibliothèque graphique pour chaque système
  - d'une plate-forme à l'autre les classes d'IHM sont différentes
  - les plate-formes sont WindowsTM, MacOSTM, Linux, SolarisTM
- **Solutions possibles:**
  - Quatre applications différentes: quatre sources qui vont vite diverger
  - Un seul source
    - avec des si alors sinon
    - avec des #ifdef #endif
  - Emploi de Abstract Factory !

Abstract Factory

- **Exemple:**



Etape 1: Créer les interfaces

```
public interface Shape {
   void draw();
}

public interface Color {
   void fill();
}
```

Etape 2: Créer les classes qui implémentent ces interfaces

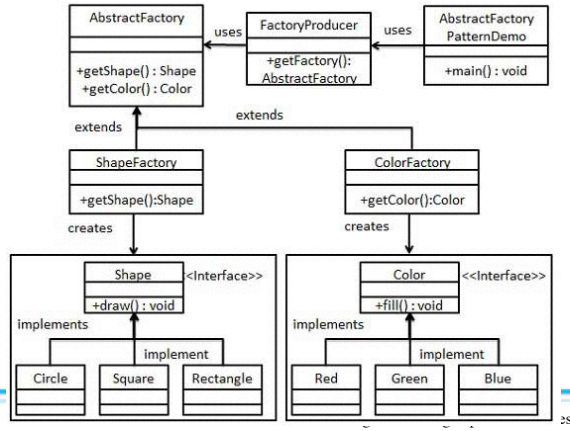
```
public class Rectangle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw()
method.");
   }
}

public class Square implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Square::draw()
method.");
   }
}

public class Circle implements Shape {
   @Override
   public void draw() {
      System.out.println("Inside Circle::draw()
method.");
   }
}
```


Etape 2: Créer les classes qui implémentent ces interfaces

```
public class Red implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Red::fill() method.");
   }
}

public class Green implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Green::fill() method.");
   }
}

public class Blue implements Color {
   @Override
   public void fill() {
      System.out.println("Inside Blue::fill() method.");
   }
}
```


Etape 3: Créer une classe Abstraite pour obtenir les factories d'objets Color et Shape

```
public abstract class AbstractFactory {
   abstract Color getColor(String color);
   abstract Shape getShape(String shape) ;
}
```



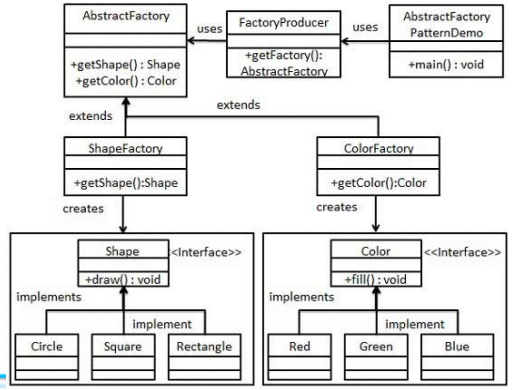
Etape 4: créer les classes Factory héritants de AbstractFactory

```java
public class ShapeFactory extends AbstractFactory {
   @Override
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      }else
if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      }else
if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;
   }
   @Override
   Color getColor(String color) {
      return null;
   }
}
```

```java
public class ColorFactory extends
AbstractFactory {
   @Override
   public Shape getShape(String shapeType)
{
      return null;
   }
   @Override
   Color getColor(String color) {
      if(color == null){
         return null;
      }
      if(color.equalsIgnoreCase("RED")){
         return new Red();
      }else
if(color.equalsIgnoreCase("GREEN")){
         return new Green();
      }else
if(color.equalsIgnoreCase("BLUE")){
         return new Blue();
      }
      return null;
   }
}
```


Etape 5: Création d'un générateur de Factory

```java
public class FactoryProducer {
   public static AbstractFactory getFactory(String choice){

      if(choice.equalsIgnoreCase("SHAPE")){
         return new ShapeFactory();

      }else if(choice.equalsIgnoreCase("COLOR")){
         return new ColorFactory();
      }

      return null;
   }
}
```



Step 6 (Main): Utiliser FactoryProducer pour obtenir une AbstractFactory afin de créer les objets

```java
public class AbstractFactoryPatternDemo {
   public static void main(String[] args) {

      //get shape factory
      AbstractFactory shapeFactory = FactoryProducer.getFactory("SHAPE");

      //get an object of Shape Circle
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Shape Circle
      shape1.draw();

      //get an object of Shape Rectangle
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Shape Rectangle
      shape2.draw();

      //get an object of Shape Square
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of Shape Square
      shape3.draw();
```



```java
      //get color factory
      AbstractFactory colorFactory = FactoryProducer.getFactory("COLOR");

      //get an object of Color Red
      Color color1 = colorFactory.getColor("RED");

      //call fill method of Red
      color1.fill();

      //get an object of Color Green
      Color color2 = colorFactory.getColor("Green");

      //call fill method of Green
      color2.fill();

      //get an object of Color Blue
      Color color3 = colorFactory.getColor("BLUE");

      //call fill method of Color Blue
      color3.fill();
   }
}
```

Patrons Créateurs

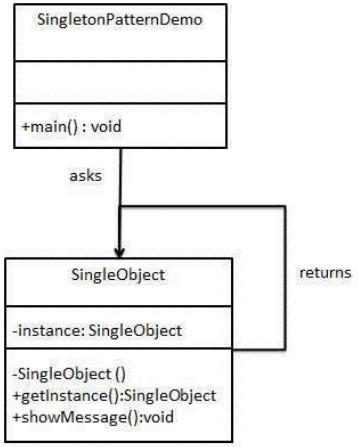
- Dans cette catégorie, on trouve par exemple les patrons:
  - Factory (Fabrique)
  - Abstract Factory (Fabrique abstraite)
  - **Singleton** (Singleton)

Singleton

- Singleton est parmi les patrons les plus simples. Il permet de répondre au besoin d'unicité d'une classe
- Ce pattern utilise une seule classe qui est responsable sur la création d'objets
- l'unicité de l'instance est complètement contrôlée par la classe elle même.
- Ce patron peut facilement être étendu pour permettre la création d'un nombre donné d'instances
- **Quand l'utiliser?** Quand il n'y a qu'une unique instance d'une classe et qu'elle doit être accessible de manière connue

Singleton

- **Implémentation**
  - On crée une classe **SingleObject** qui a un constructeur privé et a une instance statique d'elle même
  - La classe **SingleObject** possède une méthode statique pour obtenir son instance statique à partir de l'extérieur.



Etape 1: Créer une classe Singleton

```
public class SingleObject {

   //create an object of SingleObject
   private static SingleObject instance = new SingleObject();

   //make the constructor private so that this class cannot be
   //instantiated
   private SingleObject(){}

   //Get the only object available
   public static SingleObject getInstance(){
      return instance;
   }

   public void showMessage(){
      System.out.println("Hello World!");
   }
}
```

Etape 2: Obtenir un seul objet en passant par la classe qui est en singleton

```
public class SingletonPatternDemo {
   public static void main(String[] args) {

      //illegal construct
      //Compile Time Error: The constructor SingleObject()
      //is not visible
      //SingleObject object = new SingleObject();

      //Get the only object available
      SingleObject object = SingleObject.getInstance();

      //show the message
      object.showMessage();
   }
}
```

Résultat:

```
Hello World!
```


Design Patterns : Structuration (*structural patterns*)

Patrons Structuration

- Dans cette catégorie, on trouve par exemple les patrons:
  - Composite (composition)
  - Bridge (Pont)

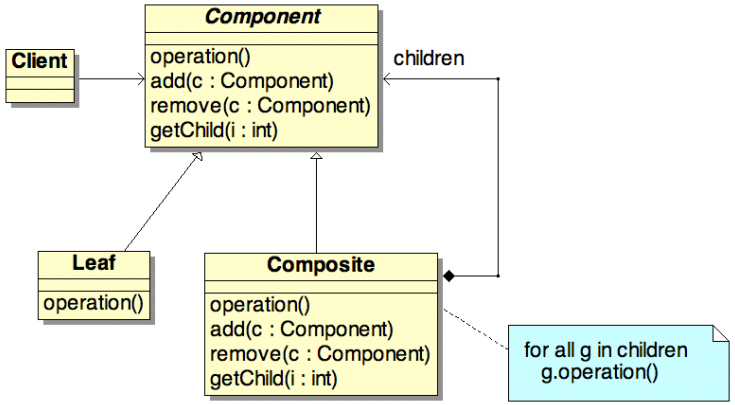
Patrons Structuration

- Dans cette catégorie, on trouve par exemple les patrons:
  - **Composite**
  - Bridge

Composite

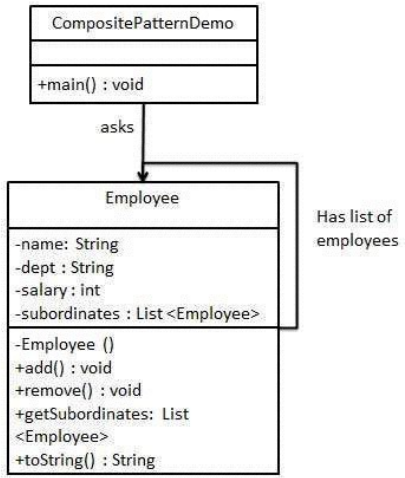
- Ce patron est utilisé quand on a besoin de traiter un groupe d'objets de la même manière comme étant un seul objet.
- Le patron Composite compose les objets en une structure d'arbre pour représenter une partie ou l'ensemble d'hiérarchie.
- Ce patron consiste à créer une classe qui contient un groupe de ces objets. Cette classe fournit des moyens pour modifier ses objets identiques..
- Exemple: hiérarchie d'employés dans une entreprise.

Composite



Composite

- **Exemple d'Implémentation**



Etape 1: Créer la classe Employee qui possède une liste d'objets Employee

```
public class Employee {
   private String name;
   private String dept;
   private int salary;
   private List<Employee> subordinates;
   // constructor
   public Employee(String name,String dept, int sal) {
      this.name = name;
      this.dept = dept;
      this.salary = sal;
      subordinates = new ArrayList<Employee>();
   }
   public void add(Employee e) {
      subordinates.add(e);
   }
   public void remove(Employee e) {
      subordinates.remove(e);
   }
   public List<Employee> getSubordinates(){
     return subordinates;
   }
   public String toString(){
      return ("Employee :[ Name : " + name + ", dept : " + dept + ",
            salary :" + salary+ " ]");
   }
}
```

Etape 2: Utiliser la classe Employee

```
public class CompositePatternDemo {
   public static void main(String[] args) {

      Employee CEO = new Employee("John","CEO", 30000);

      Employee headSales = new Employee("Robert","Head Sales", 20000);

      Employee headMarketing = new Employee("Michel","Head Marketing", 20000);

      Employee clerk1 = new Employee("Laura","Marketing", 10000);
      Employee clerk2 = new Employee("Bob","Marketing", 10000);

      Employee salesExecutive1 = new Employee("Richard","Sales", 10000);
      Employee salesExecutive2 = new Employee("Rob","Sales", 10000);

      CEO.add(headSales);
      CEO.add(headMarketing);

      headSales.add(salesExecutive1);
      headSales.add(salesExecutive2);

      headMarketing.add(clerk1);
      headMarketing.add(clerk2);

      //print all employees of the organization
      System.out.println(CEO);

      for (Employee headEmployee : CEO.getSubordinates()) {
         System.out.println(headEmployee);

         for (Employee employee : headEmployee.getSubordinates())
         {
            System.out.println(employee);
         }
      }
   }
}
```


- **Résultat:** Affichage de l'hiérarchie d'employés créée

```
Employee :[ Name : John, dept : CEO, salary :30000 ]
Employee :[ Name : Robert, dept : Head Sales, salary :20000 ]
Employee :[ Name : Richard, dept : Sales, salary :10000 ]
Employee :[ Name : Rob, dept : Sales, salary :10000 ]
Employee :[ Name : Michel, dept : Head Marketing, salary :20000 ]
Employee :[ Name : Laura, dept : Marketing, salary :10000 ]
Employee :[ Name : Bob, dept : Marketing, salary :10000 ]
```

Quand utiliser?

- représenter une hiérarchie d'objets
- ignorer la différence entre un composant simple et un composant en contenant d'autres. (interface uniforme)

Patrons Structuration

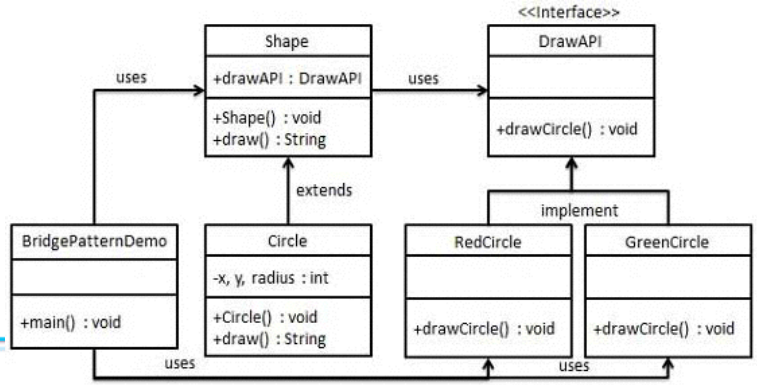
- Dans cette catégorie, on trouve par exemple les patrons:
  - Composite
  - **Bridge**

Bridge

- Le patron Bridge est utilisé quand on a besoin de faire une séparation entre une abstraction et son implémentation en créant une sorte de Pont
- Le bridge est assuré à travers une interface
- **Exemple:** Un Cercle peut être dessinée par différentes couleurs en utilisant la même méthode de la classe abstraite mais avec différentes implémentations du bridge.

Exemple d'implémentation

- DrawAPI c'est l'interface qui joue le rôle du bridge et les classes RedCircle, GreenCircle implémentent cette interface
- Shape est une classe abstraite qui utilise les objets de DrawAPI.
- BridgePatternDemo, le main qui utilise la classe Shape pour dessiner les cercles colorées.



Etape 1: Créer l'interface bridge

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

Etape 2: Créer les implémentations du bridge

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " +
       radius + ", x: " + x + ", " + y + "]");
   }
}

public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " +
       radius + ", x: " + x + ", " + y + "]");
   }
}
```


Etape 3: Créer une classe Abstraite Shape utilisant le bridge

```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

Etape 4: Créer l'implémentation de Shape

```
public class Circle extends Shape {
   private int x, y, radius;

   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
  public void draw() { drawAPI.drawCircle(radius,x,y); }
}
```


Etape 5: main

```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

Résultat:

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[  color: green, radius: 10, x: 100, 100]
```

Design Patterns : Comportement (*behavioral patterns*)

Patrons Comportement

- Dans cette catégorie, on trouve par exemple les patrons:
  - Observer (Observateur)
  - State (Etat)

Patrons Comportement

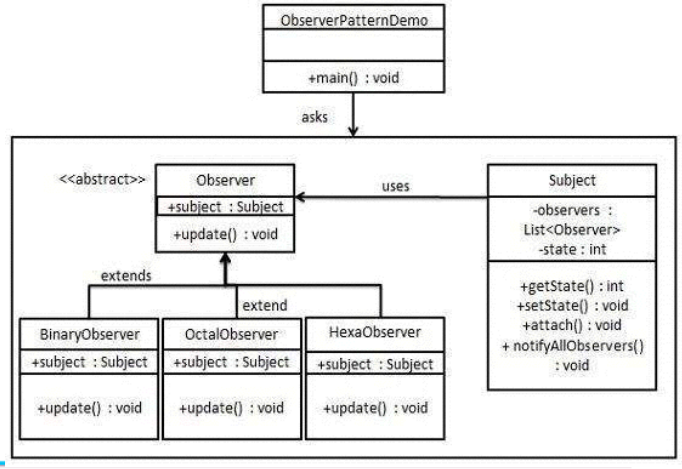
- Dans cette catégorie, on trouve par exemple les patrons:
  - **Observer**
  - State

Observer

- Le patron Observateur est utilisé quand il existe une à plusieurs relations entre les objets et que la modification d'un objet doit notifié le reste.
- Le patron Observer utilises trois classes: Sujet (Subject), l'Observateur (Observer) et le Client.
- Subject contient les méthodes pour associer ou dissocier des observateurs d'un objet client

Implementation

- On crée une classe abstraite Observer et une classe Subject qui l'utilise.



Etape 1: Créer une classe Subject

```
public class Subject {

   private List<Observer> observers = new ArrayList<Observer>();

   private int state;

   public int getState() {
      return state;
   }
   public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }
   public void attach(Observer observer){
      observers.add(observer);
   }
   public void notifyAllObservers(){
      for (Observer observer : observers) {
         observer.update();
      }
   }
}
```


Etape 2: Créer la classe abstraite Observer

```
public abstract class Observer {
   protected Subject subject;
   public abstract void update();
}
```


Etape 3: Créer les classes concrètes observer

```
public class BinaryObserver extends Observer{

   public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Binary String: " +
      Integer.toBinaryString( subject.getState() ) );
   }
}
```

```
public class OctalObserver extends Observer{

   public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
     System.out.println( "Octal String: " +
     Integer.toOctalString(subject.getState()));
   }
}
```



```
public class HexaObserver extends Observer{

   public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Hex String: " +
        Integer.toHexString(subject.getState()).
           toUpperCase() );
   }
}
```


Etape 4: Utiliser Subject and les observateurs dans main

```
public class ObserverPatternDemo {
   public static void main(String[] args) {
      Subject subject = new Subject();

      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);

      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```


Résultat

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

Patrons Comportement

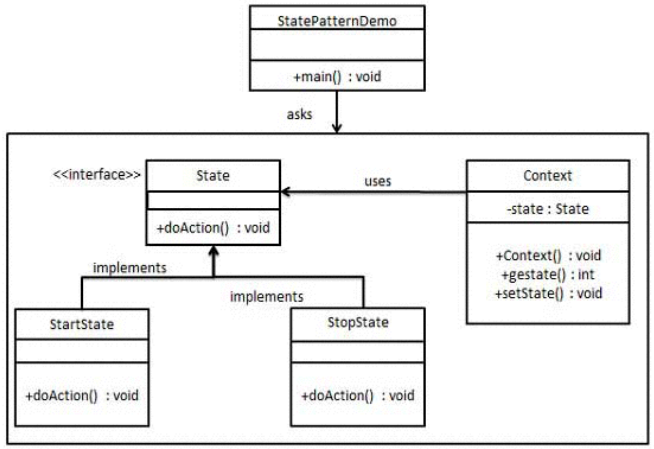
- Dans cette catégorie, on trouve par exemple les patrons:
  - Observer
  - **State**

State

- Dans le patron Etat, le comportement d'une classe change selon son état.
- On crée des objets qui représentent les différents états et un objet contexte dont le comportement peut varier quand son objet état change.

Implementation

- On crée une interface State interface définissant une action à faire et des classes qui implémentent cette interface.
- Context est une classe avec des états



Etape 1: Créer l'interface State et ses implémentations

```
public interface State {
   public void doAction(Context context);
}
public class StartState implements State {
   public void doAction(Context context) {
      System.out.println("Player is in start state");
      context.setState(this);
   }
   public String toString(){
      return "Start State";
   }
}

public class StopState implements State {
   public void doAction(Context context) {
      System.out.println("Player is in stop state");
      context.setState(this);
   }
   public String toString(){
      return "Stop State";
   }
}
```

Etape 2: la classe contexte

```
public class Context {
   private State state;

   public Context(){
      state = null;
   }

   public void setState(State state){
      this.state = state;
   }

   public State getState(){
      return state;
   }
}
```


Etape 3: main utilisant Context pour voir des changements dans le comportement quand l'état change

```
public class StatePatternDemo {
   public static void main(String[] args) {
      Context context = new Context();

      StartState startState = new StartState();
      startState.doAction(context);

      System.out.println(context.getState().toString());

      StopState stopState = new StopState();
      stopState.doAction(context);

      System.out.println(context.getState().toString());
   }
}
```

Résultat:

```
Player is in start state
Start State
Player is in stop state
Stop State
```


Patrons de conception

- Il existe bien d'autres patterns que nous n'avons pas détaillés

	PURPOSE		
SCOPE	**CREATIONAL**	**STRUCTURAL**	**BEHAVIOURAL**
CLASS	Factory Method	Adapter (class)	Interpreter
			Template Method
OBJECT	Abstract Factory	Adapter (object)	Command.
	Builder	Bridge	Iterator
	Prototype	Composite	Mediator
	Singleton	Decorator	Memento
		Facade	Observer
		Flyweight	State
		Proxy	Strategy
			Visitor
			Chain Of Resp.

Patrons de conception

- Le plus difficile quand on veut appliquer un Pattern:
  - Trouver les bons objets
  - Bien choisir la granularité des objets
  - Spécifier les interfaces des objets

Patrons de conception

- **Avantages:**
  - Un vocabulaire commun, facilite la communication
  - Capitalisation de l'expérience
  - Un niveau d'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité
  - Réduire la complexité
  - Guide/catalogue de solutions

#Fin