```
1  # Principes de conception orientée-objet
2
3  ## Principes SOLID
4
5  ### SRP
6
7  SRP consiste à créer une classe par fonctionnalité. Cela permet d'éviter qu'une autre classe dépende de plusieurs fonctionnalités à la fois alors qu'elle en dépend que d'une seule. Si une classe possède plusieurs responsabilités, elle aura plusieurs raisons de changer.
8  Il n'est pas nécessaire de découpler les responsabilités si les changements n’ont aucun risque de se produire, ou s’ils se produisent toujours ensemble. (ex : Employe, EmployeSalaire, EmployeCoordonnees).
9
10 ### OCP
11
12 OCP consiste à utiliser l'héritage, l'abstraction de classes et le polymorphisme pour pouvoir facilement étendre des modules mais sans devoir les modifier. (ex : Employe, Vendeur, Manager).
13
14 ### LSP
15
16 LSP consiste à utiliser de l'héritage pour pouvoir avoir des méthodes communes. Si une sous classe ne doit pas implémenter une méthode, il ne faut pas l'implémenter dans la classe mère. (ex : Robot, RobotStatique, RobotDynamique).
17
18 ### ISP
19
20 ISP consiste à utiliser des interfaces découpées en fonction des besoins et ne pas regrouper une grosse inrterface pour tous les besoins. (ex : Printer, Scanner, Copyer, Faxer comme interfaces et SimplePrinter implémente uniquement Printer).
21
22 ### DIP
23
24 DIP consiste à ce que les modules haut niveaux ne doivent pas dépendre de modules bas niveau. Pour cela, on crée une méthode dans la classe métier A qui prend en paramètre une interface I. On crée une classe B qui implémente l'interface I. Dans la méthode de A, on appelle donc l'interface I avec la méthode implémentée par B. (ex : Classe Metier, Interface Logger, Classe ConsoleLogger qui implémente Logger).
25
26 ex : au lieu de faire
27 ```java
28 public class MetierFaux {
29
30     private int val;
31
32
33     public Metier(int val) {
34         this.val=val;
35     }
36
37     public void methode(Logger logger) {
38         System.out.println(LocalDateTime.now()+ ": Début de methode");
39         this.val++;
40         System.out.println(LocalDateTime.now() + ": Fin de methode");
41     }
42
```

```
}
```
il faut faire :

```java
public class Metier {

    private int val;


    public Metier(int val) {
        this.val=val;
    }

    public void methode(Logger logger) {
        //logger = Logger.getLogger("logger");
        logger.log(LocalDateTime.now()+ ": Début de methode");

        this.val++;
        logger.log(LocalDateTime.now() + ": Fin de methode");
    }

}


public class ConsoleLogger implements Logger{
    public ConsoleLogger() {

    }

    public void log(String string) {
        System.out.println(string);
    }
}

public interface Logger {
    public void log(String string);
}
```

Patterns GRASP

Polymorphisme, Expert en information, Créateur, Fabrication pure, Faible couplage, Indirection, Forte cohésion, Protection, Contrôleur

Définitions

- Idiome : construction utilisée de façon récurrente dans un langage de programmation donné pour réaliser une tâche « simple » (i++ par ex)pour parcourir les éléments d'une collection
- Pattern d'architecture : solution générique et réutilisable à un problème d'architecture logicielle
- Pattern d'entreprise : solution pour la structuration d'une application d'entreprise
- Anti-pattern : solution commune à un problème récurrent mais qui est en général inefficace et contre-productive

```java

/* ////////////////////////////////////////////////////////////////////
    SINGLETON garantit qu'une seule instance d'une classe est créée.
*/

// SOLUTION AVEC CLASSE

class Singleton {
    private static Singleton INSTANCE; // instance unique

    private Singleton(){ // constructeur privé
        // ...
    }

    public static Singleton getInstance(){
        if(INSTANCE == null) { // crée l'instance au premier appel
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }

    public void run(String[] args){
        // ...
    }

    public static void main(String[] args){
        getInstance().run(args);
    }
}

// SOLUTION AVEC ENUMERATION

enum Singleton {
    ENVIRONNEMENT;

    public void run(String[] args){
        // ...
    }

    public static void main(String[] args){
        ENVIRONNEMENT.run(args);
    }
}

/* ////////////////////////////////////////////////////////////////////
    Builder permet de créer une classe avec un grand nombre d'attributs et
    qui doit gérer un grand nombre de constructeurs.
*/

public class StreetMap{
    private final Point origin;
    private final Point destination;

    private final Color waterColor;
    private final Color landColor;
    private final Color highTrafficColor;
    private final Color mediumTrafficColor;
```

```
58      private final Color lowTrafficColor;
59
60      public static class Builder{
61          //required parameters
62          private final Point origin;
63          private final Point destination;
64          // optional parameters initialize with default values
65          private Color waterColor = Color.BLUE;
66          private Color landColor = Color.RED;
67          private Color highTrafficColor = Color.YELLOW;
68          private Color mediumTrafficColor = Color.PURPLE;
69          private Color lowTrafficColor = Color.ORANGE;
70
71          public Builder(Point origin, Point destination){
72              this.origin=origin;
73              this.detination=destination;
74          }
75
76          //faire la meme chose pour chaque parametre
77          public Builder waterColor(Color color){
78              this.waterColor=color;
79              return this;
80          }
81
82          public StreetMap build(){
83              return new StreetMap(this);
84          }
85      }
86
87      private StreetMap(Builder builder){
88          //required parameters
89          origin = builder.origin;
90          destination = builder.destination;
91
92          //optional parameters
93          waterColor = builder.waterColor;
94          landColor = builder.landColor;
95          highTrafficColor = builder.highTrafficColor;
96          mediumTrafficColor = builder.mediumTrafficColor;
97          lowTrafficColor = builder.lowTrafficColor;
98      }
99  }
100
101 public static void main(String args[]){
102     StreetMap s = new StreetMap
103         .Builder(new Point(1,2), new Point(2,3))
104         .landColor(Color.GREY)
105         .waterColor(Color.BLACK)
106         .build();
107 }
108
109 /* ////////////////////////////////////////////////////////////////////
110     Factory method permet la création d'objets sans préciser explicitement la
    classe à utiliser. Les
111     objets sont créés en utilisant une méthode de fabrication redéfinie dans des
    sous-classes.
112 */
```

```java

// Exemple 1 :

public class Client {

    public static void main(String[] args) {

        ComplexeIndustriel usinePomme = new UsinePomme();
        ComplexeIndustriel usinePoire = new UsinePoire();

        Fruit fruit1 = null;
        System.out.println("Utilisation de la premiere fabrique");
        fruit1 = usinePomme.getFruit();
        fruit1.afficheFruit(); // "Je suis une Pomme"

        Fruit fruit2 = null;
        System.out.println("Utilisation de la seconde fabrique");
        fruit2 = usinePoire.getFruit();
        fruit2.afficheFruit(); // "Je suis une Poire"
    }
}

public abstract class ComplexeIndustriel {

    public Fruit getFruit() {
        return createFruit();
    }

    protected abstract Fruit createFruit();
}

public class UsinePomme extends ComplexeIndustriel {

    @Override
    protected Fruit createFruit() {
        return new Pomme();
    }
}

public class UsinePoire extends ComplexeIndustriel {
    @Override
    protected Fruit createFruit() {
        return new Poire();
    }
}

public abstract class Fruit {
    public abstract void afficheFruit();
}

public class Pomme extends Fruit {

    @Override
    public void afficheFruit() {
        System.out.println("Je suis une Pomme");
    }
}
```

```
170
171 public class Poire extends Fruit {
172
173     @Override
174     public void afficheFruit() {
175         System.out.println("Je suis une Poire");
176     }
177 }
178
179 /* ////////////////////////////////////////////////////////////////////
180     Abstract Factory method permet la création d'objets sans préciser explicitement la classe à utiliser. Les
181     objets sont créés en utilisant une méthode de fabrication redéfinie dans des sous-classes.
182 */
183
184 public class Client {
185
186     public static void main(String[] args) {
187         ComplexeIndustriel usineCarottePomme = new UsineCarottePomme();
188         ComplexeIndustriel usineHaricotPoire = new UsineHaricotPoire();
189
190         Legume legume = null;
191         Fruit fruit = null;
192         System.out.println("Utilisation de la premiere fabrique");
193         legume = usineCarottePomme.getLegume();
194         fruit = usineCarottePomme.getFruit();
195         legume.afficheLegume(); // "Je suis une Carotte"
196         fruit.afficheFruit(); // "Je suis une Pomme"
197
198         System.out.println("Utilisation de la seconde fabrique");
199         legume = usineHaricotPoire.getLegume();
200         fruit = usineHaricotPoire.getFruit();
201         legume.afficheLegume(); // "Je suis un Haricot"
202         fruit.afficheFruit(); // "Je suis une Poire"
203     }
204 }
205
206 public interface ComplexeIndustriel {
207
208     public Legume getLegume();
209     public Fruit getFruit();
210 }
211
212 public class UsineCarottePomme implements ComplexeIndustriel {
213
214     public Legume getLegume() {
215         return new Carotte();
216     }
217
218     public Fruit getFruit() {
219         return new Pomme();
220     }
221 }
222
223 public class UsineHaricotPoire implements ComplexeIndustriel {
224
```

```java
    public Legume getLegume() {
        return new Haricot();
    }

    public Fruit getFruit() {
        return new Poire();
    }
}

public abstract class Legume {
    public abstract void afficheLegume();
}

public class Carotte extends Legume {

    public void afficheLegume() {
        System.out.println("Je suis une carotte");
    }
}

public class Haricot extends Legume {

    public void afficheLegume() {
        System.out.println("Je suis un Haricot");
    }
}

public abstract class Fruit {
    public abstract void afficheFruit();
}

public class Pomme extends Fruit {

    public void afficheFruit() {
        System.out.println("Je suis une Pomme");
    }
}

public class Poire extends Fruit {

    public void afficheFruit() {
        System.out.println("Je suis une Poire");
    }
}
```

```java
/* ////////////////////////////////////////////////////////////////
   Composite permet de créer des structures hiérarchiques pour des relations
   tout-partie.
 */

// peut etre une classe abstraite
interface Element {
    public void print();
}

public abstract class Element{
    protected String nom;
    protected int taille;

    public Element(String nom, int taille) {
        this.nom=nom;
        this.taille=taille;
    }

    public abstract void print();
}

class Fichier implements Element {

    public void print(){
        System.out.Println("je suis un fichier");
    }
}

class Dossier implements Element {

    private List<Element> contenu = new ArrayList <Element>();

    public void print(){
        System.out.Println("je suis un dossier");
        for (Element e : contenu){
            e.print();
        }
    }

    public void add(Element e){
        contenu.add(e);
    }
}

/* ////////////////////////////////////////////////////////////////
  Adapteur permet à une classe d'être utilisée avec une interface qui n'est
  pas la sienne. Il permet d'utiliser des interfaces incompatibles.
 */

/**
 * Définit une interface qui est identifiée
 * comme standard dans la partie cliente.
 */
public interface Standard {

    /**
```

```
58          * L'opération doit multiplier les deux nombres,
59          * puis afficher le résultat de l'opération
60          */
61         public void operation(int pNombre1, int pNombre2);
62     }
63
64     /**
65      * Implémente l'interface "Standard".
66      */
67     public class ImplStandard implements Standard {
68
69         public void operation(int pNombre1, int pNombre2) {
70             System.out.println("Standard : Le nombre est : " + (pNombre1 * pNombre2));
71         }
72     }
73
74     /**
75      * Fournit les fonctionnalités définies dans l'interface "Standard",
76      * mais ne respecte pas l'interface.
77      */
78     public class ImplAdapte {
79
80         public int operationAdapte1(int pNombre1, int pNombre2) {
81             return pNombre1 * pNombre2;
82         }
83
84         /**
85          * Apporte la fonctionalité définie dans l'interface,
86          * mais la méthode n'a pas le bon nom
87          * et n'accepte pas le même paramètre.
88          */
89         public void operationAdapte2(int pNombre) {
90             System.out.println("Adapte : Le nombre est : " + pNombre);
91         }
92     }
93
94     /**
95      * Adapte l'implémentation non standard avec l'héritage.
96      */
97     public class Adaptateur extends ImplAdapte implements Standard {
98
99         /**
100          * Appelle les méthodes non standard
101          * depuis une méthode respectant l'interface.
102          * 1°) Appel de la méthode réalisant la multiplication
103          * 2°) Appel de la méthode d'affichage du résultat
104          * La classe adaptée est héritée, donc on appelle directement les méthodes
105          */
106         public void operation(int pNombre1, int pNombre2) {
107             int lNombre = operationAdapte1(pNombre1, pNombre2);
108             operationAdapte2(lNombre);
109         }
110     }
111
112     // OU BIEN
113     /**
114      * Adapte l'implémentation non standard avec la composition.
```

```
115   */
116  public class Adaptateur implements Standard {
117
118      private ImplAdapte adapte = new ImplAdapte();
119
120      /**
121       * Appelle les méthodes non standard
122       * depuis une méthode respectant l'interface.
123       * 1°) Appel de la méthode réalisant la multiplication
124       * 2°) Appel de la méthode d'affichage du résultat
125       * La classe adaptée compose l'adaptation,
126       * donc on appelle les méthodes de "ImplAdapte".
127       */
128      public void operation(int pNombre1, int pNombre2) {
129          int lNombre = adapte.operationAdapte1(pNombre1, pNombre2);
130          adapte.operationAdapte2(lNombre);
131      }
132  }
133
134  public class Main {
135
136      public static void main(String[] args) {
137          // Création d'un adaptateur
138          final Standard lImplAdapte = new Adaptateur();
139          // Création d'une implémentation standard
140          final Standard lImplStandard = new ImplStandard();
141
142          // Appel de la même méthode sur chaque instance
143          lImplAdapte.operation(2, 4);
144          lImplStandard.operation(2, 4);
145
146          // Affichage :
147          // Adapte : Le nombre est : 8
148          // Standard : Le nombre est : 8
149      }
150  }
151
152  /* ////////////////////////////////////////////////////////////
153   * Decorateur permet d'implémenter une classe puis de lui rajouter des
154   * fonctionnalités.
155   */
156
157  // Déclarations
158  public abstract class Voiture {
159
160      private String nom;
161      private String marque;
162
163      abstract int getPrix();
164      abstract int getPoids();
165  }
166
167  class DS extends Voiture{
168
169      public DS() {
170          this.nom = "DS"; this.marque = "Citroën";
171      }
```

```java
    int getPrix() {return 30000;}
    int getPoids() {return 1500;}
}

// Décorateurs
abstract class VoitureAvecOption extends Voiture{
    Voiture voiture;
}

class VoitureAvecToitOuvrant extends VoitureAvecOption{

    int getPrix() {return voiture.getPrix() + 10000;}
    int getPoids() {return voiture.getPoids() + 15;}
}

//On garde le nom du pattern Decorator pour savoir qu'on wrap un objet
class DSAvecToitOuvrantDecorator extends VoitureAvecToitOuvrant{
    public DSAvecToitOuvrantDecorator(DS ds) {
        this.voiture = ds;
    }
}

public class Main {
    // Implémentation
    public static void main(String[] args) {
        Voiture ds = new DS();
        Voiture dsOption = new DSAvecToitOuvrantDecorator((DS) ds);
    }
}

/* ////////////////////////////////////////////////////////////////
 * Facade permet d'utiliser les fonctionnalités de plusieurs classes à
 * partir d'une seule (Facade)
 */

/**
 * Classe implémentant diverses fonctionnalités.
 */
public class ClasseA {

    public void operation1() {
        System.out.println("Methode operation1() de la classe ClasseA");
    }

    public void operation2() {
        System.out.println("Methode operation2() de la classe ClasseA");
    }
}

/**
 * Classe implémentant d'autres fonctionnalités.
 */
public class ClasseB {

    public void operation3() {
        System.out.println("Methode operation3() de la classe ClasseB");
    }
```

```
229
230         public void operation4() {
231             System.out.println("Methode operation4() de la classe ClasseB");
232         }
233     }
234
235     /**
236      * Présente certaines fonctionnalités.
237      * Dans ce cas, ne présente que la méthode "operation2()" de "ClasseA"
238      * et la méthode "operation41()" utilisant "operation4()" de "ClasseB"
239      * et "operation1()" de "ClasseA".
240      */
241     public class Facade {
242
243         private ClasseA classeA = new ClasseA();
244         private ClasseB classeB = new ClasseB();
245
246         /**
247          * La méthode operation2() appelle simplement
248          * la même méthode de ClasseA
249          */
250         public void operation2() {
251             System.out.println("--> Méthode operation2() de la classe Facade : ");
252             classeA.operation2();
253         }
254
255         /**
256          * La méthode operation41() appelle
257          * operation4() de ClasseB
258          * et operation1() de ClasseA
259          */
260         public void operation41() {
261             System.out.println("--> Méthode operation41() de la classe Facade : ");
262             classeB.operation4();
263             classeA.operation1();
264         }
265     }
266
267     public class FacadePatternMain {
268
269         public static void main(String[] args) {
270             // Création de l'objet "Facade" puis appel des méthodes
271             Facade lFacade = new Facade();
272             lFacade.operation2();
273             lFacade.operation41();
274         }
275     }
276
277     /* ////////////////////////////////////////////////////////////
278      * BRIDGE
279      */
280
281     /**
282      * Définit l'interface de l'implémentation.
283      * L'implémentation fournit deux méthodes
284      */
285     public interface Implementation {
```

```
286
287         public void operationImpl1(String pMessage);
288         public void operationImpl2(Integer pNombre);
289     }
290
291     /**
292      * Sous-classe concrète de l'implémentation
293      */
294     public class ImplementationA implements Implementation {
295
296         public void operationImpl1(String pMessage) {
297             System.out.println("operationImpl1 de ImplementationA : " + pMessage);
298         }
299
300         public void operationImpl2(Integer pNombre) {
301             System.out.println("operationImpl2 de ImplementationA : " + pNombre);
302         }
303     }
304
305     /**
306      * Sous-classe concrète de l'implémentation
307      */
308     public class ImplementationB implements Implementation {
309
310         public void operationImpl1(String pMessage) {
311             System.out.println("operationImpl1 de ImplementationB : " + pMessage);
312         }
313
314         public void operationImpl2(Integer pNombre) {
315             System.out.println("operationImpl2 de ImplementationB : " + pNombre);
316         }
317     }
318
319     /**
320      * Définit l'interface de l'abstraction
321      */
322     public abstract class Abstraction {
323
324         // Référence vers l'implémentation
325         private Implementation implementation;
326
327         protected Abstraction(Implementation pImplementation) {
328             implementation = pImplementation;
329         }
330
331         public abstract void operation();
332
333         /**
334          * Lien vers la méthode operationImpl1() de l'implémentation
335          * @param pMessage
336          */
337         protected void operationImpl1(String pMessage) {
338             implementation.operationImpl1(pMessage);
339         }
340
341         /**
342          * Lien vers la méthode operationImpl2() de l'implémentation
```

```
343         * @param pMessage
344         */
345        protected void operationImpl2(Integer pNombre) {
346            implementation.operationImpl2(pNombre);
347        }
348    }
349
350    /**
351     * Sous-classe concrète de l'abstraction
352     */
353    public class AbstractionA extends Abstraction {
354
355        public AbstractionA(Implementation pImplementation) {
356            super(pImplementation);
357        }
358
359        public void operation() {
360            System.out.println("--> Méthode operation() de AbstractionA");
361            operationImpl1("A"); operationImpl2(1); operationImpl1("B");
362        }
363    }
364
365    /**
366     * Sous-classe concrète de l'abstraction
367     */
368    public class AbstractionB extends Abstraction {
369
370        public AbstractionB(Implementation pImplementation) {
371            super(pImplementation);
372        }
373
374        public void operation() {
375            System.out.println("--> Méthode operation() de AbstractionB");
376            operationImpl2(9); operationImpl2(8); operationImpl1("Z");
377        }
378    }
379
380    public class BridgePatternMain {
381
382        public static void main(String[] args) {
383            // Création des implémentations
384            Implementation lImplementationA = new ImplementationA();
385            Implementation lImplementationB = new ImplementationB();
386
387            // Création des abstractions
388            Abstraction lAbstractionAA = new AbstractionA(lImplementationA);
389            Abstraction lAbstractionAB = new AbstractionA(lImplementationB);
390            Abstraction lAbstractionBA = new AbstractionB(lImplementationA);
391            Abstraction lAbstractionBB = new AbstractionB(lImplementationB);
392
393            // Appels des méthodes des abstractions
394            lAbstractionAA.operation(); lAbstractionAB.operation();
395            lAbstractionBA.operation(); lAbstractionBB.operation();
396        }
397    }
398
```

```java
/* ////////////////////////////////////////////////
 * Command
 */

public interface Command {
   void execute();
}

public class Interrupteur {

   private List<Command> history = new ArrayList<Command>();

   public Interrupteur() {
   }

   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
}

public class Light {

   public Light() {
   }

   public void turnOn() {
      System.out.println("The light is on");
   }

   public void turnOff() {
      System.out.println("The light is off");
   }
}

public class Allumer implements Command {

   private Light theLight;

   public Allumer(Light light) {
      this.theLight = light;
   }

   public void execute(){
      theLight.turnOn();
   }
}

public class Eteindre implements Command {

   private Light theLight;

   public Eteindre(Light light) {
      this.theLight = light;
   }

   public void execute() {
```

```
58          theLight.turnOff();
59       }
60    }
61
62    public class PressSwitch {
63
64       public static void main(String[] args){
65          Light lamp = new Light();
66          Command allumer = new Allumer(lamp);
67          Command eteindre = new Eteindre(lamp);
68
69          Interrupteur s = new Interrupteur();
70
71          try {
72             if (args[0].equalsIgnoreCase("ON")) {
73                s.storeAndExecute(allumer);
74                System.exit(0);
75             }
76             if (args[0].equalsIgnoreCase("OFF")) {
77                s.storeAndExecute(eteindre);
78                System.exit(0);
79             }
80             System.out.println("Argument \"ON\" or \"OFF\" is required.");
81          } catch (Exception e) {
82             System.out.println("Argument's required.");
83          }
84       }
85    }
86
87    /* ////////////////////////////////////////////////
88     * Iterator
89     */
90
91    public interface Iterator {
92       public boolean hasNext();
93       public Object next();
94    }
95
96    public interface Container {
97       public Iterator getIterator();
98    }
99
100   public class NameRepository implements Container {
101      public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};
102
103      @Override
104      public Iterator getIterator() {
105         return new NameIterator();
106      }
107
108      private class NameIterator implements Iterator {
109         int index;
110
111         @Override
112         public boolean hasNext() {
113            if(index < names.length){
114               return true;
```

```
115             }
116             return false;
117         }
118 
119         @Override
120         public Object next() {
121             if(this.hasNext()){
122                 return names[index++];
123             }
124             return null;
125         }
126     }
127 }
128 
129 public class Main {
130 
131     public static void main(String[] args) {
132         NameRepository namesRepository = new NameRepository();
133 
134         for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
135             String name = (String)iter.next();
136             System.out.println("Name : " + name);
137         }
138 
139         /* Name : Robert, Name : John, Name : Julie, Name : Lora */
140     }
141 }
142 
143 
144 /* ////////////////////////////////////////////////
145  * Observateur
146  */
147 
148 /** Classe représentant un GPS (appareil permettant de connaître sa position). */
149 public class Gps extends Observable{
150         private String position;// Position du GPS.
151         private int precision;// Précision accordé à cette position
152 
153         // Constructeur.
154         public Gps(){
155                 position="inconnue";
156                 precision=0;
157         }
158 
159         // Méthode permettant de notifier tous les observateurs lors d'un
        changement d'état du GPS.
160         public void notifierObservateurs(){
161                 setChanged();// Méthode de l'API.
162                 notifyObservers();// Egalement une méthode de l'API.
163         }
164 
165         // Méthode qui permet de mettre à jour de façon artificielle le GPS.
166         // Dans un cas réel, on utiliserait les valeurs retournées par les capteurs.
167         public void setMesures(String position, int precision){
168                 this.position=position;
169                 this.precision=precision;
170                 notifierObservateurs();
```

```
171           }
172
173           public String getPosition() return position;
174           public int getPrecision() return precision;
175   }
176
177   /** Affiche un résumé en console des informations (position) du GPS. */
178   public class AfficheResume implements Observer {
179         // Méthode appelée automatiquement lors d'un changement d'état du GPS.
180         public void update(Observable o, Object obj){
181                 if(o instanceof Gps){
182                         Gps g = (Gps) o;
183                         System.out.println("Position : "+g.getPosition());
184                 }
185         }
186   }
187
188   public class Main{
189         // Méthode principale.
190         public static void main(String[] args){
191                 // Création de l'objet Gps observable.
192                 Gps g = new Gps();
193                 // Création de deux observeurs AfficheResume et AfficheComplet
194                 AfficheResume ar = new AfficheResume();
195                 // On ajoute AfficheResume comme observeur de Gps.
196                 g.addObserver(ar);
197                 // On simule l'arrivée de nouvelles valeurs via des capteurs.
198                 g.setMesures("N 39°59°993 / W 123°00°000", 4);
199                 // Nouvelle simulation d'arrivée de nouvelles valeurs via des capteurs.
200                 g.setMesures("N 37°48°898 / W 124°12°011", 5);
201         }
202   }
203
204   /* ////////////////////////////////////////////////
205    * Template method permet de deleguer des methodes aux sous classes
206    */
207
208    public abstract class Game {
209      abstract void initialize();
210      abstract void startPlay();
211      abstract void endPlay();
212
213      //template method
214      public final void play(){
215
216         //initialize the game
217         initialize();
218
219         //start game
220         startPlay();
221
222         //end game
223         endPlay();
224      }
225   }
226
227   public class Cricket extends Game {
```

```
228
229     @Override
230     void endPlay() {
231        System.out.println("Cricket Game Finished!");
232     }
233
234     @Override
235     void initialize() {
236        System.out.println("Cricket Game Initialized! Start playing.");
237     }
238
239     @Override
240     void startPlay() {
241        System.out.println("Cricket Game Started. Enjoy the game!");
242     }
243  }
244
245  public class Football extends Game {
246
247     @Override
248     void endPlay() {
249        System.out.println("Football Game Finished!");
250     }
251
252     @Override
253     void initialize() {
254        System.out.println("Football Game Initialized! Start playing.");
255     }
256
257     @Override
258     void startPlay() {
259        System.out.println("Football Game Started. Enjoy the game!");
260     }
261  }
262
263  public class TemplatePatternMain {
264     public static void main(String[] args) {
265
266        Game game = new Cricket();
267        game.play();
268        System.out.println();
269        game = new Football();
270        game.play();
271     }
272  }
273
```

Persistance
En POO, la persistance est la propriété permettant à un objet de continuer à exister après la destruction de son créateur.
– C'est la capacité de sauvegarder l'état des objets, i.e. les données finales de l'application
– La persistance est dite orthogonale ou transparente si la propriété est intrinsèque à l'environnement d'exécution

Serialisation
La sérialisation permet la transformation d'un objet en un flux d'octets.
– Permet le stockage des objets sur disque, leur transmission par le réseau, . . .
– L'opération inverse se nomme désérialisation
– Marshalling/unmarshalling sont des concepts équivalents

```java
public class Personne {
    private String nom;
    private int age;

    public Personne(String nom, int age) {/**/}

    @Override
    public String toString(){/**/}
}

public abstract class DAO<T>{
    protected Connection connect = /* ... */;

    public abstract T create(T obj);
    public abstract T find(String id);
    public abstract T update(T obj);
    public abstract void delete (T obj);
}

public class PersonneDAO extends DAO<Personne>{
    @Override
    public Personne create(Personne obj){
        try{
            PreparedStatement prepare = connect.prepareStatement(
                "INSERT INTO personnes (nom, age) VALUES (?, ?)");
            prepare.setString(1, obj.getNom());
            prepare.setLong(2, obj.getAge());
            int result = prepare.executeUpdate();
            assert result == 1;
        }
        catch(SQLException e){
            e.printStackTrace();
        }
        return obj;
    }

    @Override
    public Personne find(String id){
        Personne p = new Personne();

        try{
```

```
55                  PreparedStatement prepare = connect.prepareStatement(
56                      "SELECT * FROM personnes WHERE nom = ? ");
57                  prepare.setString(1, id);
58                  int result = prepare.executeQuery();
59                  if(result.first()){
60                      p=new Personne(result.getString("nom"), result.getString("age"));
61                  }
62              }
63              catch(SQLException e){
64                  e.printStackTrace();
65              }
66              return obj;
67          }
68      }
69
70      public class DAO Factory {
71          public static DAO<Personne> getPersonneDAO(){
72              return new PersonneDAO();
73          }
74      }
75
76      public class Main {
77          public static void main(String[] args){
78              DAO<Personne> personneDao = DAOFactory.getPersonneDAO();
79              System.out.println(personneDao.find("Dupond"));
80          }
81      }
82
83      ////////////////////////////////////////////////////////////////////:
84
85      Exemple classe immuable :
86      public final class Personnel{
87          private final String nom;
88          private final String prenom;
89      }
90
```