

Principes de conception orientée-objet

Principes SOLID

SRP

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.

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).

OCP

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).

LSP

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).

ISP

ISP consiste à utiliser des interfaces découpées en fonction des besoins et ne pas regrouper une grosse interface pour tous les besoins. (ex : Printer, Scanner, Copyer, Faxer comme interfaces et SimplePrinter implémente uniquement Printer).

DIP

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).

ex : au lieu de faire

```java

public class MetierFaux {

private int val;

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

public void methode(Logger logger) {  
 System.out.println(LocalDate.now()+ ": Début de methode");  
 this.val++;  
 System.out.println(LocalDate.now() + ": Fin de methode");  
 }

```
43 }
44 ```
45 il faut faire :
46
47 ```java
48 public class Metier {
49
50 private int val;
51
52
53 public Metier(int val) {
54 this.val=val;
55 }
56
57 public void methode(Logger logger) {
58 //logger = Logger.getLogger("logger");
59 logger.log(LocalDate.now()+ ": Début de methode");
60
61 this.val++;
62 logger.log(LocalDate.now() + ": Fin de methode");
63 }
64 }
65
66
67
68 public class ConsoleLogger implements Logger{
69 public ConsoleLogger() {
70
71 }
72
73 public void log(String string) {
74 System.out.println(string);
75 }
76 }
77
78 public interface Logger {
79 public void log(String string);
80 }
81 ```
82
83 ## Patterns GRASP
84
85 Polymorphisme, Expert en information, Créateur, Fabrication pure, Faible couplage, ↗
86 Indirection, Forte cohésion, Protection, Contrôleur
87
88 ## Définitions
89
90 - Idiomme : construction utilisée de façon récurrente dans un langage de ↗
91 programmation donné pour réaliser une tâche « simple » (i++ par ex) pour parcourir ↗
92 les éléments d'une collection
93
94 - Pattern d'architecture : solution générique et réutilisable à un problème ↗
95 d'architecture logicielle
96
97 - Pattern d'entreprise : solution pour la structuration d'une application ↗
98 d'entreprise
99
100 - Anti-pattern : solution commune à un problème récurrent mais qui est en général ↗
101 inefficace et contre-productive
```

```
1
2 /* //
3 SINGLETON garantit qu'une seule instance d'une classe est créée.
4 */
5
6 // SOLUTION AVEC CLASSE
7
8 class Singleton {
9 private static Singleton INSTANCE; // instance unique
10
11 private Singleton(){ // constructeur privé
12 // ...
13 }
14
15 public static Singleton getInstance(){
16 if(INSTANCE == null) { // crée l'instance au premier appel
17 INSTANCE = new Singleton();
18 }
19 return INSTANCE;
20 }
21
22 public void run(String[] args){
23 // ...
24 }
25
26 public static void main(String[] args){
27 getInstance().run(args);
28 }
29 }
30
31 // SOLUTION AVEC ENUMERATION
32
33 enum Singleton {
34 ENVIRONNEMENT;
35
36 public void run(String[] args){
37 // ...
38 }
39
40 public static void main(String[] args){
41 ENVIRONNEMENT.run(args);
42 }
43 }
44
45 /* //
46 Builder permet de créer une classe avec un grand nombre d'attributs et
47 qui doit gérer un grand nombre de constructeurs.
48 */
49
50 public class StreetMap{
51 private final Point origin;
52 private final Point destination;
53
54 private final Color waterColor;
55 private final Color landColor;
56 private final Color highTrafficColor;
57 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.destination=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
111 classe à utiliser. Les
112 objets sont créés en utilisant une méthode de fabrication redéfinie dans des
113 sous-classes.
114 */
```

```
113
114 // Exemple 1 :
115
116 public class Client {
117
118 public static void main(String[] args) {
119
120 ComplexeIndustriel usinePomme = new UsinePomme();
121 ComplexeIndustriel usinePoire = new UsinePoire();
122
123 Fruit fruit1 = null;
124 System.out.println("Utilisation de la premiere fabrique");
125 fruit1 = usinePomme.getFruit();
126 fruit1.afficheFruit(); // "Je suis une Pomme"
127
128 Fruit fruit2 = null;
129 System.out.println("Utilisation de la seconde fabrique");
130 fruit2 = usinePoire.getFruit();
131 fruit2.afficheFruit(); // "Je suis une Poire"
132 }
133 }
134
135 public abstract class ComplexeIndustriel {
136
137 public Fruit getFruit() {
138 return createFruit();
139 }
140
141 protected abstract Fruit createFruit();
142 }
143
144 public class UsinePomme extends ComplexeIndustriel {
145
146 @Override
147 protected Fruit createFruit() {
148 return new Pomme();
149 }
150 }
151
152 public class UsinePoire extends ComplexeIndustriel {
153
154 @Override
155 protected Fruit createFruit() {
156 return new Poire();
157 }
158 }
159
160 public abstract class Fruit {
161 public abstract void afficheFruit();
162 }
163
164 public class Pomme extends Fruit {
165
166 @Override
167 public void afficheFruit() {
168 System.out.println("Je suis une Pomme");
169 }
170 }
```

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

```
225 public Legume getLegume() {
226 return new Haricot();
227 }
228
229 public Fruit getFruit() {
230 return new Poire();
231 }
232 }
233
234 public abstract class Legume {
235 public abstract void afficheLegume();
236 }
237
238 public class Carotte extends Legume {
239
240 public void afficheLegume() {
241 System.out.println("Je suis une carotte");
242 }
243 }
244
245 public class Haricot extends Legume {
246
247 public void afficheLegume() {
248 System.out.println("Je suis un Haricot");
249 }
250 }
251
252 public abstract class Fruit {
253 public abstract void afficheFruit();
254 }
255
256 public class Pomme extends Fruit {
257
258 public void afficheFruit() {
259 System.out.println("Je suis une Pomme");
260 }
261 }
262
263 public class Poire extends Fruit {
264
265 public void afficheFruit() {
266 System.out.println("Je suis une Poire");
267 }
268 }
269
```

```
1 /* //
2 Composite permet de créer des structures hiérarchiques pour des relations
3 tout-partie.
4 */
5
6 // peut etre une classe abstraite
7 interface Element {
8 public void print();
9 }
10
11 public abstract class Element{
12 protected String nom;
13 protected int taille;
14
15 public Element(String nom, int taille) {
16 this.nom=nom;
17 this.taille=taille;
18 }
19
20 public abstract void print();
21 }
22
23 class Fichier implements Element {
24
25 public void print(){
26 System.out.Println("je suis un fichier");
27 }
28 }
29
30 class Dossier implements Element {
31
32 private List<Element> contenu = new ArrayList <Element>();
33
34 public void print(){
35 System.out.Println("je suis un dossier");
36 for (Element e : contenu){
37 e.print();
38 }
39 }
40
41 public void add(Element e){
42 contenu.add(e);
43 }
44 }
45
46 /* //
47 Adapteur permet à une classe d'être utilisée avec une interface qui n'est
48 pas la sienne. Il permet d'utiliser des interfaces incompatibles.
49 */
50
51 /**
52 * Définit une interface qui est identifiée
53 * comme standard dans la partie cliente.
54 */
55 public interface Standard {
56
57 /**
```



```
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 fonctionnalité 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 }
```

```
172 int getPrix() {return 30000;}
173 int getPoids() {return 1500;}
174 }
175
176 // Décorateurs
177 abstract class VoitureAvecOption extends Voiture{
178 Voiture voiture;
179 }
180
181 class VoitureAvecToitOuvrant extends VoitureAvecOption{
182
183 int getPrix() {return voiture.getPrix() + 10000;}
184 int getPoids() {return voiture.getPoids() + 15;}
185 }
186
187 //On garde le nom du pattern Decorator pour savoir qu'on wrap un objet
188 class DSAvecToitOuvrantDecorator extends VoitureAvecToitOuvrant{
189 public DSAvecToitOuvrantDecorator(DS ds) {
190 this.voiture = ds;
191 }
192 }
193
194 public class Main {
195 // Implémentation
196 public static void main(String[] args) {
197 Voiture ds = new DS();
198 Voiture dsOption = new DSAvecToitOuvrantDecorator((DS) ds);
199 }
200 }
201
202 /* //
203 * Facade permet d'utiliser les fonctionnalités de plusieurs classes à
204 * partir d'une seule (Facade)
205 */
206
207 /**
208 * Classe implémentant diverses fonctionnalités.
209 */
210 public class ClasseA {
211
212 public void operation1() {
213 System.out.println("Methode operation1() de la classe ClasseA");
214 }
215
216 public void operation2() {
217 System.out.println("Methode operation2() de la classe ClasseA");
218 }
219 }
220
221 /**
222 * Classe implémentant d'autres fonctionnalités.
223 */
224 public class ClasseB {
225
226 public void operation3() {
227 System.out.println("Methode operation3() de la classe ClasseB");
228 }
229 }
```

```
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 "operation4l()" 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 operation4l() appelle
257 * operation4() de ClasseB
258 * et operation1() de ClasseA
259 */
260 public void operation4l() {
261 System.out.println("--> Méthode operation4l() 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.operation4l();
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
```

```
1 /* //
2 * Command
3 */
4
5 public interface Command {
6 void execute();
7 }
8
9 public class Interrupteur {
10
11 private List<Command> history = new ArrayList<Command>();
12
13 public Interrupteur() {
14 }
15
16 public void storeAndExecute(Command cmd) {
17 this.history.add(cmd); // optional
18 cmd.execute();
19 }
20 }
21
22 public class Light {
23
24 public Light() {
25 }
26
27 public void turnOn() {
28 System.out.println("The light is on");
29 }
30
31 public void turnOff() {
32 System.out.println("The light is off");
33 }
34 }
35
36 public class Allumer implements Command {
37
38 private Light theLight;
39
40 public Allumer(Light light) {
41 this.theLight = light;
42 }
43
44 public void execute(){
45 theLight.turnOn();
46 }
47 }
48
49 public class Eteindre implements Command {
50
51 private Light theLight;
52
53 public Eteindre(Light light) {
54 this.theLight = light;
55 }
56
57 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 }
```

```
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 observateurs AfficheResume et AfficheComplet
194 AfficheResume ar = new AfficheResume();
195 // On ajoute AfficheResume comme observateur 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
```

```
1 Persistence
2 En P00, la persistance est la propriété permettant à un objet de continuer à
3 exister après la des-
4 truction de son créateur.
5 – C’est la capacité de sauvegarder l’état des objets, i.e. les données finales de
6 l’application
7 – La persistance est dite orthogonale ou transparente si la propriété est
8 intrinsèque à l’environnement d’exécution
9
10 Serialisation
11 La sérialisation permet la transformation d’un objet en un flux d’octets.
12 – Permet le stockage des objets sur disque, leur transmission par le réseau, . . .
13 – L’opération inverse se nomme désérialisation
14 – Marshalling/unmarshalling sont des concepts équivalents
15
16 public class Personne {
17 private String nom;
18 private int age;
19
20 public Personne(String nom, int age) {/**/}
21
22 @Override
23 public String toString(){/**/}
24 }
25
26 public abstract class DAO<T>{
27 protected Connection connect = /* ... */;
28
29 public abstract T create(T obj);
30 public abstract T find(String id);
31 public abstract T update(T obj);
32 public abstract void delete (T obj);
33 }
34
35 public class PersonneDAO extends DAO<Personne>{
36 @Override
37 public Personne create(Personne obj){
38 try{
39 PreparedStatement prepare = connect.prepareStatement(
40 "INSERT INTO personnes (nom, age) VALUES (?, ?)");
41 prepare.setString(1, obj.getNom());
42 prepare.setLong(2, obj.getAge());
43 int result = prepare.executeUpdate();
44 assert result == 1;
45 }
46 catch(SQLException e){
47 e.printStackTrace();
48 }
49 return obj;
50 }
51
52 @Override
53 public Personne find(String id){
54 Personne p = new Personne();
55
56 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
```

```

1 //////////////////////////////////////:
2 // JPA Persistence
3 //////////////////////////////////////
4
5 @Entity
6 public class Todo {
7 @Id
8 @GeneratedValue(strategy = GenerationType.IDENTITY)
9 private Long id;
10 private String summary;
11 private String description;
12
13 public String getSummary() {return summary;}
14 public void setSummary(String summary) {this.summary = summary;}
15 public String getDescription() {return description;}
16 public void setDescription(String description) {this.description = description;}
17
18 @Override
19 public String toString(){
20 return "Todo [summary=" + summary + ", description=" + description + "];"
21 }
22 }
23
24 @Entity
25 public class Task {
26 @Id
27 @GeneratedValue(strategy = GenerationType.IDENTITY)
28 private Long id;
29 private String title;
30
31 @OneToMany(mappedBy = "task")
32 private final List<Task> todos;
33 }
34
35 public class Main {
36 private static final String PERSISTENCE_UNIT_NAME = "todos";
37 private static EntityManagerFactory factory;
38
39 public static void main(){
40 factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
41 EntityManager em = factory.createEntityManager();
42 em.getTransaction().begin();
43 Todo todo = new Todo();
44 todo.setSummary("This is a test");
45 todo.setDescription("This is a test");
46 em.persist(todo);
47 em.getTransaction().commit();
48 em.close();
49 }
50 }
51
52 // COURS
53 //////////////////////////////////////
54
55 - Les patterns sont des outils supplémentaires pour réaliser une bonne conception.
56 - Les principes concernant la conception des classes (SOLID), abordent la cohésion ↗
 des modules, traitent du couplage entre modules

```

```
57 --> Les principes et les patterns de conception sont principalement liés aux ↗
 dépendances.
58 - Les idiomes sont une construction utilisée de façon récurrente dans un langage ↗
 de programmation donné pour
59 réaliser une tâche « simple » ex :i++
60
61 ///
62
63 - ORM : Mapping objet-relationnel
64 - Hibernate est un ORM open source et est une implémentation de JPA
65 - JPA fournit un modèle de persistance pour les objetsJava (POJO) basé l'ORM (haut ↗
 niveau)
66 - JDBC fournissent un moyen pour interfacer un programme avec un SGBD (bas niveau)
67 - SGBD : systeme de gestion de base de donnees qui permet de manipuler des infos ↗
 dans une bdd
68
69 ///
70
71 Une unité de persistance (persistance unit) configure l'accès au SGBD pour un ↗
 ensemble d'entités.
72 Elle est décrite dans le fichier XML META-INF/persistence.xml
73
74 ///
75
76 - Marquer une entité : @Entity
77 - marquer une cle primaire : @Id
78 - avec un ORM, on gere les associations n-n avec @ManyToMany (mappedBy = "task")
79
```