# Programmation, Génie Logiciel, Preuves

Zoubida Kedad, Stéphane Lopes
zoubida.kedad@uvsq.fr, stephane.lopes@uvsq.fr

2018–2019

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
UFR DES SCIENCES
université PARIS-SACLAY

# Table des matières

# Table des figures

# Listings

# Liste des exercices

# Chapitre 1

# Préambule

## Sommaire



## 1.1 Objectifs et prérequis

### Objectifs du cours

— Maîtriser les bases de la conception orientée-objet
— Connaître différentes approches pour la persistance des objets
— Mettre en évidence les liens entre modèles (UML) et implémentation

1

### Prérequis

— Connaître un langage de programmation objet
— Connaître la notation UML
— Connaître les outils de développement de base (GIT, MAVEN, ...)

2

## 1.2 Plan

### Plan général

— Principes de conception orientée-objet
— Patterns de conception (*Design Patterns*)
— Persistance des objets
— Liens entre modèles et implémentation

3

## 1.3 Bibliographie

Barnes, David et Michael Kölling (2016). *Objects First with Java. A Practical Introduction using BlueJ*. English. 6th edition. Pearson. isbn : 978-1-292-15904-1. url : http://www.bluej.org/objects-first/.

Bloch, Joshua (2008). *Effective Java*. English. 2nd edition. Addison-Wesley.

Bloch, Joshua et Neal Gafter (2005). *Java Puzzlers*. English. Addison-Wesley. url : http://www.javapuzzlers.com/.

Horstmann, Cay S. et Gary Cornell (2012). *Core Java. Fundamentals*. English. 9th edition. T. 1. Prentice Hall. isbn : 978-0-13-708189-9. url : http://www.horstmann.com/corejava.html.

— (2013). *Core Java. Advanced Features*. English. 9th edition. T. 2. Prentice Hall. isbn : 978-0-13-708160-8. url : http://www.horstmann.com/corejava.html.

Hunt, Andrew et David Thomas (2001). *The Pragmatic Programmer*. the Pragmatic Bookshelf. url : http://www.pragprog.com/titles/tpp/the-pragmatic-programmer.

Martin, Robert C. (2008). *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall. url : http://www.pearsonhighered.com/educator/academic/product/1,3110,0132350882,00.html.

Smart, John Ferguson (2008). *Java Power Tools*. O'Reilly. url : http://www.wakaleo.com/java-power-tools.

## 1.4 Webographie

Neward, Ted (2009). *Essential Java resources*. English. developerWorks. url : http://www.ibm.com/developerworks/java/library/j-javaresources.html.

*UML en français* (p. d.). Français. url : http://uml.free.fr/.

*Unified Modeling Language (UML) Resource Page* (2014). English. Object Management Group (OMG). url : http://www.uml.org/.

# Chapitre 2

# Introduction

## Sommaire



### Conception orientée-objet

— Lors de son exécution, un *système OO* est **un ensemble d'objets qui interagissent**

— La *conception orientée-objet* (COO) consiste donc à créer un *modèle* qui respecte les concepts objet

4

### Difficultés de la conception orientée-objet

— Les concepts objets sont nombreux et complexes (attribut, méthode, objet, classe, héritage, ...)

⇒ plusieurs solutions sont en général envisageables

— Identifier la bonne solution est difficile

— Plusieurs symptômes voire métriques de qualité peuvent guider les choix

### *Programmer en Java ou en C# n'est pas concevoir objet !*

— Seule une analyse objet conduit à une solution objet, i.e. qui respecte les concepts objet

— Le langage de programmation est un moyen d'implémentation qui ne garantit pas le respect des concepts objet

5

### Symptôme d'une conception défectueuse

**Rigidité** résistance aux changements

— un simple changement provoque une cascade de modification dans les modules dépendants

— génère des réticences à se lancer dans des modifications

**Fragilité** tendance du logiciel à avoir des défaillances lors de changements

— défaillance même dans des régions non directement liées au changement

— rend la maintenance difficile

**Immobiblité** impossibilité de réutiliser des modules

— les modules sont tellement dépendants qu'il est très difficile de les utiliser dans un contexte différent

**Viscosité** un changement qui respecte la conception est plus difficile à réaliser qu'un bricolage (*hack*)

6

**Anticiper les changements**

— La cause des problèmes de conception est liée aux changements dans les besoins du client
— Les évolutions sont faites sans forcément respecter la conception initiale
— Les changements (certains au moins) doivent donc être anticipés durant la conception

7

**Gestion des dépendances**

— Les changements qui génèrent des problèmes sont les modifications inattendues dans les dépendances
— Ces dernières doivent donc être gérées durant la conception
— Les principes et les patterns de conception sont principalement liés aux dépendances

8

**Principes, patterns et idiomes**

— Principes généraux : KISS, YAGNI, DRY, Law of Demeter, ...
— Principes de COO : Principes SOLID/Patterns GRASP
— Patterns : Design Patterns, ...
— Idiomes

9

## 2.1 Bibliographie


Barnes, David et Michael Kölling (2016). *Objects First with Java. A Practical Introduction using BlueJ*. English. 6th edition. Pearson. ISBN : 978-1-292-15904-1. URL : http://www.bluej.org/objects-first/.

Bloch, Joshua (2008). *Effective Java*. English. 2nd edition. Addison-Wesley.

Bloch, Joshua et Neal Gafter (2005). *Java Puzzlers*. English. Addison-Wesley. URL : http://www.javapuzzlers.com/.

Horstmann, Cay S. et Gary Cornell (2012). *Core Java. Fundamentals*. English. 9th edition. T. 1. Prentice Hall. ISBN : 978-0-13-708189-9. URL : http://www.horstmann.com/corejava.html.

— (2013). *Core Java. Advanced Features*. English. 9th edition. T. 2. Prentice Hall. ISBN : 978-0-13-708160-8. URL : http://www.horstmann.com/corejava.html.

Hunt, Andrew et David Thomas (2001). *The Pragmatic Programmer*. the Pragmatic Bookshelf. URL : http://www.pragprog.com/titles/tpp/the-pragmatic-programmer.

Martin, Robert C. (2008). *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall. URL : http://www.pearsonhighered.com/educator/academic/product/1,3110,0132350882,00.html.

Smart, John Ferguson (2008). *Java Power Tools*. O'Reilly. URL : http://www.wakaleo.com/java-power-tools.


## 2.2 Webographie


Neward, Ted (2009). *Essential Java resources*. English. developerWorks. URL : http://www.ibm.com/developerworks/java/library/j-javaresources.html.

*UML en français* (p. d.). Français. URL : http://uml.free.fr/.

*Unified Modeling Language (UML) Resource Page* (2014). English. Object Management Group (OMG). URL : http://www.uml.org/.

# Chapitre 3

# Principes de conception orientée-objet

## Sommaire



## 3.1 Principes SOLID

### 3.1.1 Introduction

#### Principes SOLID

— Le premier jet de ces principes a été publié sur le newsgroup `comp.object` par Robert C. Martin en 1995

— Ces principes adressent la question de la gestion des dépendances dans la COO

— La bonne gestion des dépendances est nécessaire à la production d'un logiciel de qualité

— Les principes SOLID sont classés en trois groupes
  — cinq principes concernent la conception des classes (SOLID)
  — trois principes abordent la cohésion des modules
  — trois principes traitent du couplage entre modules

10

**Principes liés à la conception des classes**

| | | |
|---|---|---|
| SRP | Single Responsibility Principle | A class should have one, and only one, reason to change. |
| OCP | Open Closed Principle | You should be able to extend a classes behavior, without modifying it. |
| LSP | Liskov Substitution Principle | Derived classes must be substitutable for their base classes. |
| ISP | Interface Segregation Principle | Make fine grained interfaces that are client specific. |
| DIP | Dependency Inversion Principle | Depend on abstractions, not on concretions. |

11

**Principes liés à la cohésion des modules**

| | | |
|---|---|---|
| REP | Release Reuse Equivalency Principle | The granule of reuse is the granule of release. |
| CCP | Common Closure Principle | Classes that change together are packaged together. |
| CRP | Common Reuse Principle | Classes that are used together are packaged together. |

12

**Principes liés au couplage entre modules**

| | | |
|---|---|---|
| ADP | Acyclic Dependencies Principle | The dependency graph of packages must have no cycles. |
| SDP | Stable Dependencies Principle | Depend in the direction of stability. |
| SAP | Stable Abstractions Principle | Abstractness increases with stability. |

13

### 3.1.2 Single Responsibility Principle (SRP)

**Single Responsibility Principle (SRP)**

**Single Responsibility Principle (SRP)**

A class should have only one reason to change.

— SRP est lié à la mesure de la *cohésion*

— La cohésion mesure le rapport entre une fonctionnalité attendue et le service rendu par une classe (ou un module)

— SRP relie cette notion au concept de changement

— Les exemples sont issus de SRP: The Single Responsibility Principle, Robert C. Martin

14

**Exemple**
*La classe Rectangle viole SRP*
(voir figure 3.1).

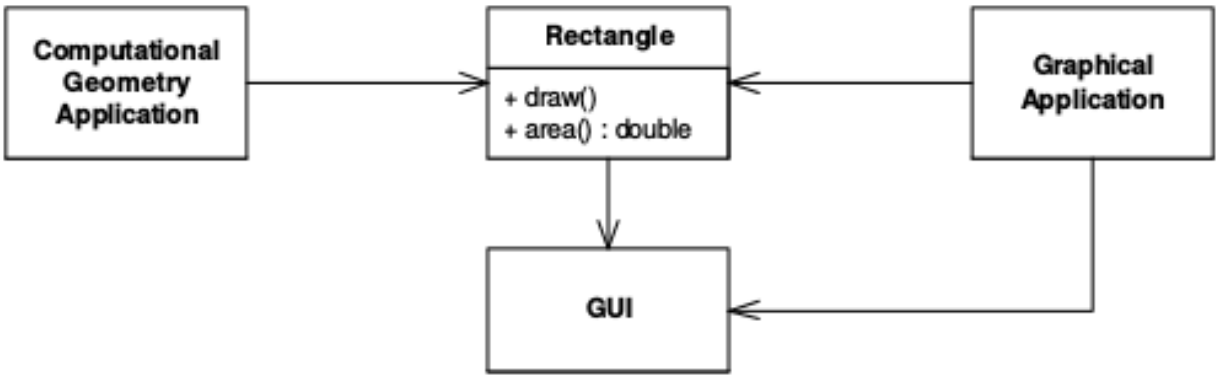


FIGURE 3.1 – Un exemple de violation de SRP.

— La classe `Rectangle` possède deux responsabilités
  — le calcul de surface,
  — l'affichage graphique.
— L'application de calcul géométrique dépend de l'affichage graphique
— Un changement de l'application graphique peut nécessiter un changement dans le rectangle et donc une reconstruction de l'application géométrique

15

**Exemple**
*La classe Rectangle modfifée*
(voir figure 3.2).

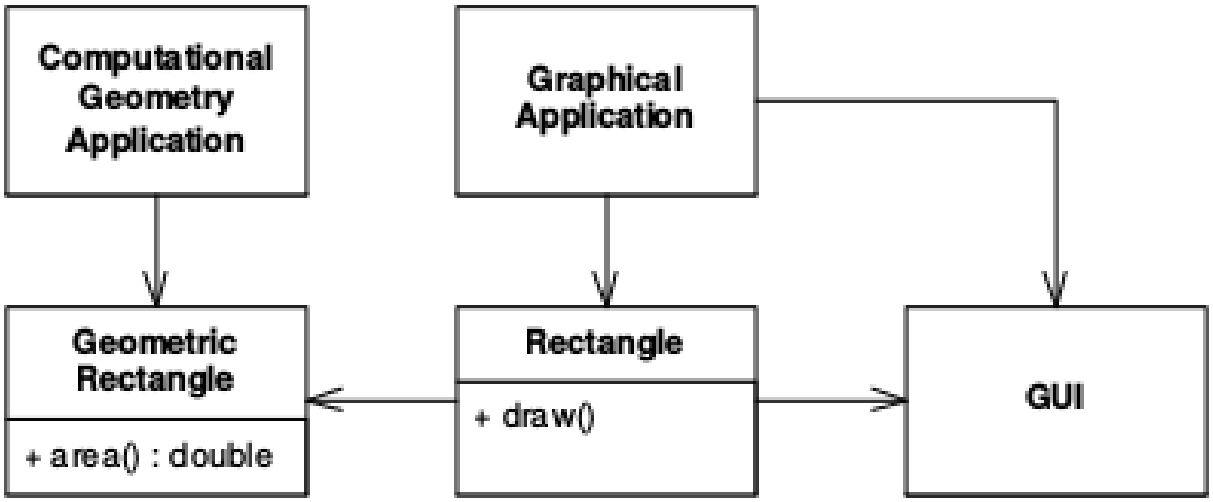


FIGURE 3.2 – Une application de SRP.

— Cette conception minimise l'impact des changements

16

**Responsabilité**

— Dans SRP, une *responsabilité* est définie comme « une cause de changement »
— Un changement dans les besoins provoquera une modification des responsabilités des classes

⇒ si une classe possède plusieurs responsabilités, elle aura plusieurs raisons de changer

— Si une classe possède plusieurs responsabilités, ces dernières sont couplées

⇒ un changement de l'une peut perturber le service de l'autre

— Quand n'est-il pas nécessaire de découpler les responsabilités ?

— si les changements n'ont aucun risque de se produire

— s'ils se produisent toujours ensemble

17

### 3.1.3 Open Closed Principle (OCP)

**Open Closed Principle (OCP)**

**Open Closed Principle (OCP)**

A module should be open for extension but closed for modification.

— Les modules (ou les classes) doivent pouvoir être étendus mais sans devoir être modifiés

— le comportement doit pouvoir être changé sans modification du code source

— Les techniques permettant d'atteindre ce but sont basées sur l'*abstraction* en particulier sur les concepts OO

18

**Intérêt et difficultés d'OCP**

— L'intégration de nouveaux besoins ne nécessite que l'ajout de code et ne modifie pas l'existant

— l'existant ne peut pas être dégradé

— les modifications ne se propagent pas aux modules dépendants

— Il est difficile de respecter à la lettre ce principe

— peut rendre la conception complexe

— respecter OCP partiellement peut déjà apporter beaucoup à la conception

— Les exemples sont issus de The Open-Closed Principle, Robert C. Martin

19

**Exemple**

*Un exemple de violation d'OCP 1/4*

```
public enum ShapeType {
  CIRCLE, SQUARE;
}

public abstract class Shape {
  public final ShapeType type;

  public Shape(ShapeType type) {
    this.type = type;
  }
}
```

Listing 3.1 – La classe Shape

— L'attribut type représente le type de forme

— Suit une approche procédurale (non OO)

20

### Exemple
*Un exemple de violation d'OCP 2/4*

```
public class Circle extends Shape {
  Point2D center;
  double radius;

  public Circle(Point2D center, double radius) {
    super(ShapeType.CIRCLE);
    this.center = center;
    this.radius = radius;
  }
}
```

Listing 3.2 – La classe Circle

21

### Exemple
*Un exemple de violation d'OCP 3/4*

```
public class Square extends Shape {
  Point2D topLeft;
  final double side;

  public Square(Point2D topLeft, double side) {
    super(ShapeType.SQUARE);
    this.topLeft = topLeft;
    this.side = side;
  }
}
```

Listing 3.3 – La classe Square

22

### Exemple
*Un exemple de violation d'OCP 4/4*

```
public static double computeArea(Shape s) {
  double result = 0;

  switch (s.type) {
  case CIRCLE:
    result = computeArea((Circle)s);
    break;
  case SQUARE:
    result = computeArea((Square)s);
    break;
  default:
    assert false : s.type;
  }
  return result;
}
```

Listing 3.4 – Le calcul de la surface

- Ne respecte pas OCP
  - l'ajout d'une forme oblige à modifier computeArea
- Ce motif se répétera dans toutes les fonctions qui devront différencier les formes

23

### Exemple
*Un exemple respectant OCP 1/3*

```
public abstract class Shape {
    public abstract double computeArea();
}

public class Circle extends Shape {
    Point2D center;
    double radius;

    public Circle(Point2D center, double radius) {
        this.center = center;
        this.radius = radius;
    }
```

```
    @Override
    public double computeArea() {
        return PI * pow(radius, 2.0);
    }
}
```

Listing 3.5 – Les classes Shape et Circle

— Le type de forme s'appuie sur le polymorphisme

24

**Exemple**
*Un exemple respectant OCP 2/3*

```
public class Square extends Shape {
    Point2D topLeft;
    final double side;

    public Square(Point2D topLeft, double side) {
        this.topLeft = topLeft;
        this.side = side;
    }

    @Override
    public double computeArea() {
        return pow(side, 2.0);
    }
}
```

Listing 3.6 – La classe Square

25

**Exemple**
*Un exemple respectant OCP 3/3*

```
shapes = new ArrayList<>();
shapes.add(new Circle(new Point2D(1.0, 1.0), 1.0));
shapes.add(new Square(new Point2D(3.0, 4.0), 2.0));

double total = shapes.stream()
    .mapToDouble(Shape::computeArea)
    .sum();
```

Listing 3.7 – Le calcul de surface

— Le calcul utilise le polymorphisme
— Aucune modification de l'existant n'est nécessaire pour ajouter une forme

26

**Implications d'OCP**

— Les attributs doivent être privés
  — quand un attribut change, toute les méthodes qui en dépendent doivent changer aussi
  ⇒ ces méthodes ne sont pas fermées par rapport à cet attribut
    — normal pour les méthodes de la classe elle-même
    — non souhaitable pour les autres méthodes (*encapsulation*)
— Pas de variables globales
  — même argumentaire que pour les attributs mais par rapport aux modules
— Attention à l'usage des informations de typage à l'exécution (`instanceof` en java, `dynamic_cast` en C++, ...)
  — l'ajout d'un nouveau type peut provoquer un changement dans les méthodes

27

### 3.1.4 Liskov Substitution Principle (LSP)

**Liskov Substitution Principle (LSP)**

**Liskov Substitution Principle (LSP)**

Derived classes must be substitutable for their base classes.

- Est issu des travaux de Barbara Liskov
  - if for each object $o_1$ of type $S$ there is an object $o_2$ of type $T$ such that for all programs $P$ defined in terms of $T$, the behavior of $P$ is unchanged when $o_1$ is substituted for $o_2$ then $S$ is a subtype of $T$.
- également lié à l'approche *Design By Contract*, Bertrand Meyer
  - derived methods should expect no more and provide no less.
- Les exemples sont issus de The Liskov Substitution Principle, Robert C. Martin

28

**Conséquence d'une violation du LSP**

- Une méthode qui ne respecte pas LSP doit disposer d'informations sur les sous-classes
  - ⇒ l'ajout d'un sous-classe impose de modifier la méthode
  - ⇒ violation d'OCP

29

**Exemple**

*Un exemple de violation de LSP 1/3*

```
public class Rectangle {
    private int height;
    private int width;

    public int getHeight() { return height; }
    public void setHeight(int height) { this.height = height; }
    public int getWidth() { return width; }
    public void setWidth(int width) { this.width = width; }
}
```

Listing 3.8 – La classe Rectangle

30

**Exemple**

*Un exemple de violation de LSP 2/3*

```
public class Square extends Rectangle {
    private void setSide(int side) {
        super.setHeight(side);
        super.setWidth(side);
    }

    public void setHeight(int height) {
        setSide(height);
    }

    public void setWidth(int width) {
        setSide(width);
    }
}
```

Listing 3.9 – La classe Square

- Mathématiquement, un carré est un rectangle (relation *ISA*)
  - ⇒ modélisé par un héritage entre Rectangle et Square
- Intuitivement, on sent que ce choix est discutable
  - height et width ne sont pas utiles
  - idem pour les getters/setters correspondants
  - bricolage pour que le comportement soit adapté au carré

31

**Exemple**
*Un exemple de violation de LSP 3/3*

```
Rectangle r = new Rectangle();
r.setHeight(3);
r.setWidth(4);
assertThat(r.getHeight() * r.getWidth(), is(12)); // OK

Rectangle r = new Square();
r.setHeight(3);
r.setWidth(4);
assertThat(r.getHeight() * r.getWidth(), is(12)); // Échoue
```

Listing 3.10 – Méthode cliente

— L'utilisateur suppose que la modification de la hauteur n'a pas d'impact sur la largeur (et réciproquement)
— Le comportement n'est pas le même en présence d'un rectangle ou d'un carré
  ⇒ violation de LSP
  ⇒ le code client doit changer pour supporter la classe Square
  ⇒ violation d'OCP

32

**Conséquence de LSP**

— La validité d'un modèle n'est pas intrinsèque
  — dépend de son usage (des hypothèses de l'utilisateur du modèle)
  ⇒ lors de la conception, il faut « imaginer » ce que va supposer l'utilisateur
— La relation ISA porte sur le comportement
  — un carré est bien un rectangle d'un point de vue mathématique
  — un carré ne possède absolument pas le comportement d'un rectangle (indépendance entre hauteur et largeur)
— LSP et conception par contrat
  — l'utilisateur d'un objet d'une classe de base ne connaît que les pré et post-conditions de cette classe
  ⇒ toute sous-classe doit les respecter
  ⇒ la pré-condition ne peut pas être plus restrictive
  ⇒ la post-condition ne peut être que plus forte

33

### 3.1.5 Interface Segregation Principle (ISP)

**Interface Segregation Principle (ISP)**

**Interface Segregation Principle (ISP)**
Client should not be forced to depend upon interfaces that they do not use.

— ISP aborde la question de la taille des interfaces des classes
— Une interface ayant de trop nombreuses méthodes manque de cohésion
— Elle doit être découpée en fonction des besoins des clients
— Un client interagit à travers une interface adaptée à son besoin
— Les exemples sont issus de The Interface Segregation Principle, Robert C. Martin

34

**Exemple**
*Un exemple de violation de ISP 1/3*

```
public class Rectangle {
    private int height;
    private int width;

    public double computeArea() {
        return height * width;
    }

    public void draw(GraphicsContext gc) {
        gc.setFill(Color.GREEN);
        gc.fillRoundRect(110, 60, 30, 30, 10, 10);
    }
}
```

Listing 3.11 – La classe Rectangle

— fournit le calcul d'aire et l'affichage graphique (viole SRP)
— dépend de JavaFX

35

**Exemple**
*Un exemple de violation de ISP 2/3*

```
public class DrawingApp extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        primaryStage.setTitle("Drawing Operations Test");
        Group root = new Group();
        Canvas canvas = new Canvas(300, 250);
        GraphicsContext gc = canvas.getGraphicsContext2D();

        Rectangle r = new Rectangle();
        r.draw(gc);

        root.getChildren().add(canvas);
        primaryStage.setScene(new Scene(root));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Listing 3.12 – L'application graphique

36

**Exemple**
*Un exemple de violation de ISP 3/3*

```
public class GeometricApp {
    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        System.out.println(r.computeArea());
    }
}
```

Listing 3.13 – L'application géométrique

— Dépend inutilement de JavaFX
— Un changement dans les besoins de l'application graphique peut nécessiter une recompilation de l'application géométrique

37

**Exemple**
*Un exemple respectant ISP 1/2*

```
public interface DrawableRectangle {
    void draw(GraphicsContext gc);
}

public interface GeometricRectangle {
    double computeArea();
}
```

Listing 3.14 – Les interfaces

— Vont permettre d'isoler les applications de la classe concrète
— Met en œuvre le pattern ADAPTATEUR

38

**Exemple**
*Un exemple respectant ISP 2/2*

```
public class Rectangle implements GeometricRectangle, DrawableRectangle {
//...
```

Listing 3.15 – La classe Rectangle

```
DrawableRectangle dr = // logique de création de l'instance
dr.draw(gc);
```

Listing 3.16 – Dans l'application graphique

```
GeometricRectangle gr = // logique de création de l'instance
System.out.println(gr.computeArea());
```

Listing 3.17 – Dans l'application géométrique

— L'application graphique ne dépend plus de l'application graphique et de ses changements

39

### 3.1.6 Dependency Inversion Principle (DIP)

**Dependency Inversion Principle (DIP)**

**Dependency Inversion Principle (DIP)**

— High level modules should not depend upon low level modules. Both should depend upon abstractions.
— Abstractions should not depend upon details. Details should depend upon abstractions.
— Les abstractions sont moins sujettes aux changements
  ⇒ elles ne doivent pas dépendre d'éléments moins stables
— Plus de détails dans The Dependency Inversion Principle, Robert C. Martin

40

**Exemple**
*Un exemple d'application de DIP*

```
public static void copy(BufferedReader from, BufferedWriter to) throws IOException {
    String line = null;
    while ((line = from.readLine()) != null) {
        to.write(line);
    }
}
```

Listing 3.18 – Copie avec `java.io`

— `BufferedReader` et `BufferedWriter` sont des abstractions pour la source et la destination
— La logique de la copie est indépendante du type de source et de destination (fichier, mémoire, réseau, ...)
— La création d'instances viole souvent le DIP
  ⇒ utilisation du pattern ABSTRACTFACTORY ou de l'*injection de dépendances*

41

### 3.1.7 Principes de cohésion des modules

#### Intérêt des modules

- Une classe possède un niveau de granularité trop fin pour une application de taille importante
- Il est nécessaire de disposer d'un outil de plus haut niveau pour organiser ce type d'application
- Les *modules* (ou *package*) représentent cet outil
  - ils permettent d'aborder la conception à un plus haut niveau d'abstraction
  - les classes sont partitionnées selon certains critères et placée dans des modules
  - les relations entre modules expriment l'organisation de haut niveau de l'application
- des questions de conceptions se posent donc vis à vis des modules
  - quels sont les meilleurs critères de partitionnement ?
  - quels liens existent entre les modules ?
  - quels principes gouvernent leurs conceptions ?

42

#### The Release/Reuse Equivalency Principle (REP)

#### The Release/Reuse Equivalency Principle (REP)

The granule of reuse is the granule of release.

- Un élément réutilisable peut être réutilisé uniquement s'il est géré par un système de distribution
- En particulier, il doit être associé à un numéro de version (de distribution)
- Un module est l'unité de distribution
  - ⇒ un module est aussi l'unité de réutilisation
- Plus de détails dans Granularity, Robert C. Martin

43

#### The Common Reuse Principle (CRP)

#### The Common Reuse Principle (CRP)

Classes that aren't reused together should not be grouped together.

- Une dépendance avec un module est une dépendance avec tout ce qu'il contient
- Si une classe change, le module change et les clients doivent s'adapter
  - ⇒ si une classe sans rapport mais se trouvant dans le module change, les clients sont impactés
  - ⇒ les classes non utilisées ensemble ne doivent pas se trouver dans le même module
- Plus de détails dans Granularity, Robert C. Martin

44

#### The Common Closure Principle (CCP)

#### The Common Closure Principle (CCP)

Classes that change together, belong together.

- Une changement dans une classe implique une redistribution du module qui la contient
- On veut minimiser le nombre de modules qui changent entre deux versions de l'application
  - ⇒ regrouper les classes qui changent ensemble permet d'atteindre ce but
- Plus de détails dans Granularity, Robert C. Martin

45

**Combiner les trois principes**

- Il est difficile de satisfaire les trois principes simultanément
- REP et CRP facilitent la réutilisation alors que CCP simplifie la maintenance
- CCP tend à produire de gros module alors que CRP en produit de petits
- On peut par exemple d'abord favoriser CCP pour la maintenance puis introduire REP et CRP quand le projet se stabilise

46

### 3.1.8 Principes liés au couplage entre modules

**The Acyclic Dependencies Principle (ADP)**

**The Acyclic Dependencies Principle (ADP)**
The dependencies betwen packages must not form cycles (DAG).

- Tous les modules se trouvant dans un cycle sont mutuellement dépendants
- Un cycle provoque une augmentation très importante du nombre de dépendances d'un projet
- Si un cycle a été ajouté dans un projet l'application des principes ISP et DIP permet de le supprimer
  - les interfaces sont isolées
  - les dépendances sont inversées
- Plus de détails dans Granularity, Robert C. Martin

47

**The Stable Dependencies Principle (SDP)**

**The Stable Dependencies Principle (SDP)**
Depend in the direction of stability.

- Un module dont dépendent de nombreux modules est difficile à changer
  ⇒ il est considéré comme très stable
- À l'inverse un module dont personne ne dépend est considéré comme instable, i.e. facile à changer
- Une application (et certains modules) doivent permettre les changements (par conception)
  ⇒ doivent être instables
  ⇒ aucuns modules stables ne doit dépendre d'eux
- Chaque module devraient uniquement dépendre de modules plus stables
- Plus de détails dans Stability, Robert C. Martin

48

**The Stable Abstractions Principle (SAP)**

**The Stable Abstractions Principle (SAP)**
Stable packages should be abstract packages.

- Les modules dont dépendent tous les autres sont difficiles à changer
- OCP impose qu'ils soient par contre extensible
  ⇒ ces modules doivent donc être très abstraits
- Une application est donc composée de
  - modules concrets instables (donc faciles à changer)
  - modules abstraits stables (donc faciles à étendre)
- La stabibilité d'un module doit être en accord avec sa stabilité
- SAP est donc directement lié à DIP
- Plus de détails dans Stability, Robert C. Martin

49

# 3.2 Patterns GRASP

## 3.2.1 Introduction

**Patterns GRASP**

- GRASP = General Responsibility Assignment Software Patterns
- Issus du travail de Craig Larman
- Ensemble de principes traitant de l'affectation de *responsabiblités* aux classes
- Tentative pour formaliser les intuitions utilisées par les concepteurs expérimentés
- Neuf patterns
  - Expert en information
  - Créateur
  - Faible couplage
  - Forte cohésion
  - Contrôleur
  - Polymorphisme
  - Fabrication pure
  - Indirection
  - Protection

50

**Qu'est ce qu'une responsabiblité ?**

- Une responsabilité correspond à une tâche qu'un objet ou un groupe d'objets doit réaliser
- Deux types : *Faire* et *Savoir*
- GRASP est un guide pour l'affectation de responsabilités aux objets

51

## 3.2.2 Expert en information

**Expert en information**

***Problème***
Érant donné un objet, quelles responsabilités peut-on lui attribuer ?

*Solution*
Lui sont assignées les responsabilités pour lesquelles il dispose des informations nécessaires à leur réalisation (Expert en information).

- Assez naturellement utilisé

52

## 3.2.3 Créateur

**Créateur**

***Problème***
Qui est responsable de créer une nouvelle instance d'un classe ?

*Solution*
Une classe B est responsable de créer une instance de A si

- B contient ou est composée de A, ou
- B enregistre A, ou
- B utilise A, ou
- B possède les données pour initialiser A.

- Limite le couplage
- Peut être insuffisant en cas de processus de création complexe (cf. Fabrique Abstraite)

53

### 3.2.4 Faible couplage

**Faible couplage**

***Problème***
Comment garantir un faible nombre de dépendances, limiter l'impact des changements et améliorer la réutilisation ?

*Solution*
Affecter les responsabilités de façon à maintenir un faible niveau de couplage.

- Peut être appliqué pour décider entre plusieurs alternatives
- Directement lié aux principes SOLID

54

### 3.2.5 Forte cohésion

**Forte cohésion**

***Problème***
Comment assurer que les objets sont compréhensible et maintenable ?

*Solution*
Assigner les responsabilités de façon à maintenir une forte cohésion.

- Peut être appliqué pour décider entre plusieurs alternatives
- Lié à Faible couplage

55

### 3.2.6 Contrôleur

**Contrôleur**

***Problème***
Comment gérer les interactions entre les messages systèmes (interface utilisateur, ...) et la couche métier ?

*Solution*
Assigner cette responsabilité à une classe parmi

**le contrôleur façade** représente le point d'entrée de l'ensemble du système

**le contrôleur de session** définit un point d'entrée par scénario/cas d'utilisation

- Le contrôleur *délègue* les traitements à la couche métier ou service
- Lié aux patterns d'architecture *MVC* (*Model-View-Controller*)
- Attention aux nombres de responsabilités affectées au contrôleur
  - trop de responsabilités ⇒ ajouter des contrôleurs
  - trop de traitements ⇒ déléguer

56

### 3.2.7 Polymorphisme

**Polymorphisme**

***Problème***
Comment gérer des alternatives basées sur le type ?

*Solution*
Assigner les responsabilités des comportements spécifiques aux classes dont le comportement est spécifique (grâce au polymorphisme)

— Passe en général par l'usage d'une classe abstraite ou une interface
— Les tests explicites sur le type dynamique d'un objet sont à proscrire
— Lié à OCP

57

### 3.2.8 Fabrication pure

**Fabrication pure**

***Problème***
Quel objet doit recevoir une responsabilité en assurant Forte cohésion, Couplage faible quand les autres principes sont inappropriés ?

*Solution*
Créer une classe artificielle (ne représentant pas un concept du domaine) et lui affecter cette responsabilité.

— Toutes les classes d'une application ne sont pas des concepts métiers
— Concerne en particulier les fonctionnalités techniques (persistance, logging, ...)

58

### 3.2.9 Indirection

**Indirection**

***Problème***
Comment assigner une responsabilité en évitant le couplage direct entre des objets ?

*Solution*
Assigner la responsabilité à un objet intermédiaire qui assure la médiation.

— L'intermédiaire est l'Indirection

59

### 3.2.10 Protection

**Protection**

***Problème***
Comment concevoir les éléments d'un système de telle façon que les variations de ces éléments n'aient pas d'effets indésirables sur les autres ?

*Solution*
Anticiper les points de variations (ou d'instabilité) et créer un interface stable autour d'eux.

— Les principes SOLID détaillent ce principe

60

## 3.3 Bibliographie

LARMAN, Craig (2005). *UML 2 et les design patterns*. Français. 3e édition. Pearson education. ISBN : 978-2-7440-7090-7. URL : http://www.craiglarman.com/.

MEYER, Bertrand (2008). *Conception et programmation orientées objet*. Français. Eyrolles. ISBN : 978-2-212-12270-1. URL : http://www.eyrolles.com/Informatique/Livre/9782212122701/livre-conception-et-programmation-orientees-objet.php.

## 3.4 Webographie

BOCK, David (p. d.). *The Paperboy, The Wallet, and The Law Of Demeter*. English. URL : http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf.

FOWLER, Martin (2015). *Software Design*. English. ThoughtWorks. URL : http://martinfowler.com/design.html.

MARTIN, Robert C. (2005). *Principles of OOD*. English. Object Mentor. URL : http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

MATIGNON, Laëtitia (2015). *Cours de Conception Orienté Objet*. Français. UCBL. URL : http://liris.cnrs.fr/laetitia.matignon/enseignementsISI3.html.

OLORUNTOBA, Samuel (2015). *S.O.L.I.D : The First 5 Principles of Object Oriented Design*. English. URL : https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

*Principles Wiki* (2014). English. URL : http://principles-wiki.net/start.

*Recommended Books* (2005). English. Objects By Design. URL : http://www.objectsbydesign.com/books/booklist.html.

*UML en français* (p. d.). Français. URL : http://uml.free.fr/.

## 3.5 Exercices

**Exercice 3.1** (Illustration du principe de responsabilité unique (SRP))
Soit la classe `Employe`

```
class Employe {
    private final String nom;
    private final String adresse;

    // ...

    public double calculSalaire() { return /* calcul du salaire */; }
    public void afficheCoordonnees() { System.out.println(nom + ", " + adresse); }
}
```

1. Cette classe respecte-t-elle SRP ? Pourquoi ?
2. Que se passe-t-il si la méthode de calcul du salaire change ?
3. Que se passe-t-il si l'affichage est remplacé par le stockage dans un fichier CSV ?
4. Proposez une solution respectant SRP.

**Exercice 3.2** (Illustration du principe ouvert/fermé (OCP))
Le salaire d'un employé est de 1500€ auquel s'ajoute 20€ par année d'ancienneté. Le salaire d'un vendeur se calcule sur la même base mais en ajoutant une commission propre à chaque vendeur. On veut pouvoir calculer la somme totale des salaires de l'entreprise.

1. Proposez une solution respectant OCP.

2. Pour le vérifier, ajoutez la classe manager (même base de calcul que l'employé plus 100€ par personne sous ses ordres).

**Exercice 3.3** (Illustration du principe de substitution de Liskov (LSP))
Soient les classes `Robot` et `RobotStatique`

```
class Robot {
    private Position position;
    private Direction direction;

    public void tourne() { /* tourne d'1/4 de tour */ }
    public void avance() { /* avance d'une case */ }
}

class RobotStatique {
    @Override
    public void avance() { thrown new UnsupportedOperationException(); }
}
```

1. Cette solution respecte-t-elle LSP ? Pourquoi ?
2. Implémentez la méthode `avancerTous` qui fait avancer tous les robots.
3. Proposez une solution respectant LSP.

**Exercice 3.4** (Illustration du principe de ségrégation des interfaces (ISP))
Soit le code Java suivant :

```
interface Printer {
    void print();
    void scan();
    void copy();
    void fax();
}

class SimplePrinter implements Printer {
    @Override
    public print() { /* print a document */ }

    @Override
    void scan() { thrown new UnsupportedOperationException(); }

    @Override
    void copy() { thrown new UnsupportedOperationException(); }

    @Override
    void fax() { thrown new UnsupportedOperationException(); }
}
```

1. Quels problèmes peuvent se poser avec cette solution ?
2. Supposons qu'une application utilisant le fac nécessite de changer l'interface de la méthode `fax` en `void fax(List<Document> l);`, quel impact cela aura-t-il sur `SimplePrinter` ?
3. Proposez une solution respectant ISP.

**Exercice 3.5** (Illustration du principe d'inversion des dépendances (DIP))
Soit le code Java suivant :

```
class UneClasseMetier {
    public void uneMethodeMetier() {
        System.out.println(LocalDateTime.now() + ": Début de uneMethodeMetier"); // log message

        // Traitements métiers

        System.out.println(LocalDateTime.now() + ": Fin de uneMethodeMetier"); // log message
    }
}
```

1. Ce code respecte-t-il DIP ? Pourquoi ?
2. Proposez une solution respectant DIP.

# Chapitre 4

# Patterns de conception

## Sommaire



## 4.1 Introduction

### Patron de conception

- Un *patron de conception* (*Design pattern*) est une solution générique d'implémentation répondant à un problème spécifique.
- Communément utilisés dans un contexte OO sous la forme d'une structure de classe.
- Popularisés par le livre **Design Patterns : Catalogue de modèles de conceptions réutilisables**, *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides*, Vuibert, 1999
  - généralement nommé *GoF* (*Gang of Four*)
  - présente 23 patterns et les applique sur des exemples en C++
- Un pattern est la formalisation de bonnes pratiques communément utilisées pour la résolution d'un problème récurrent
  - l'expression d'un pattern est composée d'un ensemble de sections (nom, objectif, applicabilité, structure, ...)

### Utilisation des design patterns

— Les patrons de conception sont des outils supplémentaires pour réaliser une bonne conception
  ⇒ s'intègre naturellement dans un processus de développement
  — La difficulté réside dans l'identification des situations d'application
— Chaque pattern doit être instancié dans le contexte où il est utilisé
— Ils ne sont pas toujours une bonne solution (risque de sur-conception)

62

### Notions liées

**Idiome** construction utilisée de façon récurrente dans un langage de programmation donné pour réaliser une tâche « simple »
  — `i++` à la place de `i = i + 1` en Java, C, ...
  — `for (element : liste) {}` pour parcourir les éléments d'une collection

**Pattern d'architecture** solution générique et réutilisable à un problème d'architecture logicielle
  — modèle *MVC* (*Model-View-Controller*)

**Pattern d'entreprise** solution pour la structuration d'une application d'entreprise
  — Service Layer

**Anti-pattern** solution commune à un problème récurrent mais qui est en général inefficace et contre-productive
  — Anemic domain model
  — God object

63

### Classification des design patterns

**Patrons de création** traitent de la création et de l'initialisation d'objet
  — masque les classes concrètes utilisées et leur mode de création

**Patrons de structure** traitent de l'organisation des relations entre classes

**Patrons de comportement** traitent de la communication entre objets

64

## 4.2 Patrons de création

### 4.2.1 Singleton

#### Le patron de conception Singleton

**Objectif**
SINGLETON garantit qu'une seule instance d'une classe sera créée et fournit un accès uniforme à cet unique objet.

(voir figure 4.1).

65

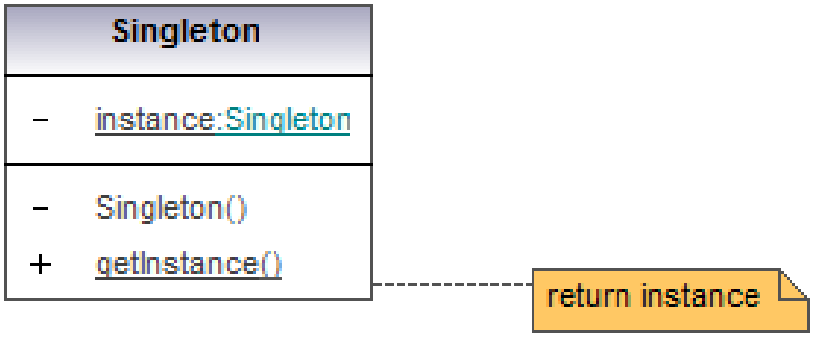


FIGURE 4.1 – Structure du SINGLETON.

### Le Singleton en Java

*Deux approches possibles*

### Avec une classe

— un attribut de classe privé fait référence à l'unique instance de la classe

— une méthode de classe publique permet de récupérer cette instance

— le constructeur de la classe est privé afin de ne pas permettre d'autres instanciations

### Avec une enumération

— la définition d'une énumération en Java est similaire à la définition d'une classe où les seules instances possibles sont connues lors de la compilation

— la seule constante de l'énumération représente l'instance unique



### Exemple

*Le programme principal comme un* SINGLETON *(avec une classe)*

```
class ApplicationSingleton {
    private static ApplicationSingleton INSTANCE; // L'instance unique

    private ApplicationSingleton() { // Constructeur privé
        // ...
    }

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

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

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



### Exemple

*Le programme principal comme un* SINGLETON *(avec une énumération)*

```
enum ApplicationSingleton {
    ENVIRONNEMENT;

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

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

```
    }
}
```

68

### 4.2.2 Builder

#### Le patron de conception Builder

**Objectif**

BUILDER propose une solution pour les cas où la logique de création d'un objet nécessite de nombreux paramètres. Il évite de recourir à une multiplication du nombre de constructeurs utilisés pour gérer les combinaisons possibles de paramètres.

(voir figure 4.2).

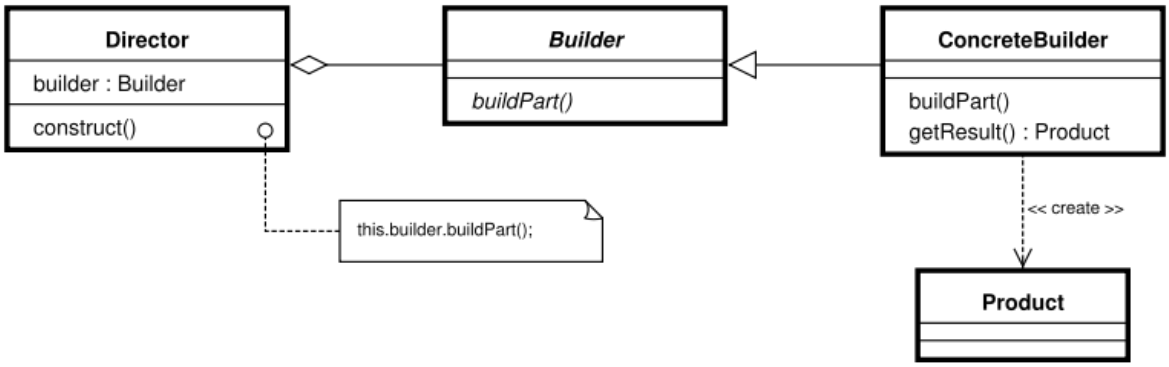


FIGURE 4.2 – Structure du BUILDER.

69

**Exemple**

BUILDER *pour la classe* `StreetMap` *1/3*

```
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;
    private final Color lowTrafficColor;

    public static class Builder {
        // cf. slide suivant
    }

    private StreetMap(Builder builder) {
        // Required parameters
        origin      = builder.origin;
        destination = builder.destination;

        // Optional parameters
        waterColor          = builder.waterColor;
        landColor           = builder.landColor;
        highTrafficColor    = builder.highTrafficColor;
        mediumTrafficColor  = builder.mediumTrafficColor;
        lowTrafficColor     = builder.lowTrafficColor;
    }
}
```

70

**Exemple**

*La classe imbriquée* BUILDER *de* `StreetMap` *2/3*

```
    public static class Builder {
        // Required parameters
        private final Point origin;
        private final Point destination;

        // Optional parameters - initialize with default values
        private Color waterColor         = Color.BLUE;
        private Color landColor          = new Color(30, 30, 30);
        private Color highTrafficColor   = Color.RED;
        private Color mediumTrafficColor = Color.YELLOW;
        private Color lowTrafficColor    = Color.GREEN;

        public Builder(Point origin, Point destination) {
            this.origin      = origin;
            this.destination = destination;
        }

        public Builder waterColor(Color color) {
            waterColor = color;
            return this;
        }

        // idem pour landColor, highTrafficColor, mediumTrafficColor
        // et lowTrafficColor

        public StreetMap build() {
            return new StreetMap(this);
        }
    }
```

71

### Exemple

*Utilisation du* BUILDER *3/3*

```
    public static void main(String args[]) {
        StreetMap map = new StreetMap
            .Builder(new Point(50, 50), new Point(100, 100))
            .landColor(Color.GRAY)
            .waterColor(Color.BLUE.brighter())
            .build();
    }
```

72

## 4.2.3 Factory method

### Le patron de conception Factory method

### Objectif

FACTORY METHOD permet la création d'objets sans préciser explicitement la classe à utiliser. Les objets sont créés en utilisant une *méthode de fabrication* redéfinie dans des sous-classes.

73

### Exemple

*Un labyrinthe avec des salles classiques 1/2*

```
public class MazeGame {
    public MazeGame() { // Template method
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() { // Factory method
        return new OrdinaryRoom();
    }
}
```

74

### Exemple
*Un labyrinthe avec des salles magiques 2/2*

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```

75

### Méthode de fabrication

— Une *méthode de fabrication* (*factory method*) est une méthode qui retourne un « nouvel » objet
— Peut être implémentée à l'aide d'un méthode de classe ou du patron Factory method
— Une *Fabrique* est une classe concrète qui crée effectivement les objets

76

### Exemple
*Une classe* `Complex`

```
class Complex {
    public static Complex fromCartesianFactory(double real, double imaginary) {
        return new Complex(real, imaginary);
    }

    public static Complex fromPolarFactory(double modulus, double angle) {
        return new Complex(modulus * cos(angle), modulus * sin(angle));
    }

    private Complex(double a, double b) {
        //...
    }
}

//...

Complex product = Complex.fromPolarFactory(1, pi);
```

77

## 4.2.4 Abstract factory

### Le patron de conception Abstract factory

### Objectif
Abstract factory permet d'encapsuler un groupe de *Fabriques* qui partagent un thème commun sans spécifier les classes concrètes.

— Le client crée une implémentation concrète de la fabrique abstraite et utilise l'interface générique pour créer les objets
— Le client ne sait pas quel objet concret a été créé
— L'emploi de ce pattern augmente la complexité du code

78

### Structure de Abstract factory
(voir figure 4.3).

79

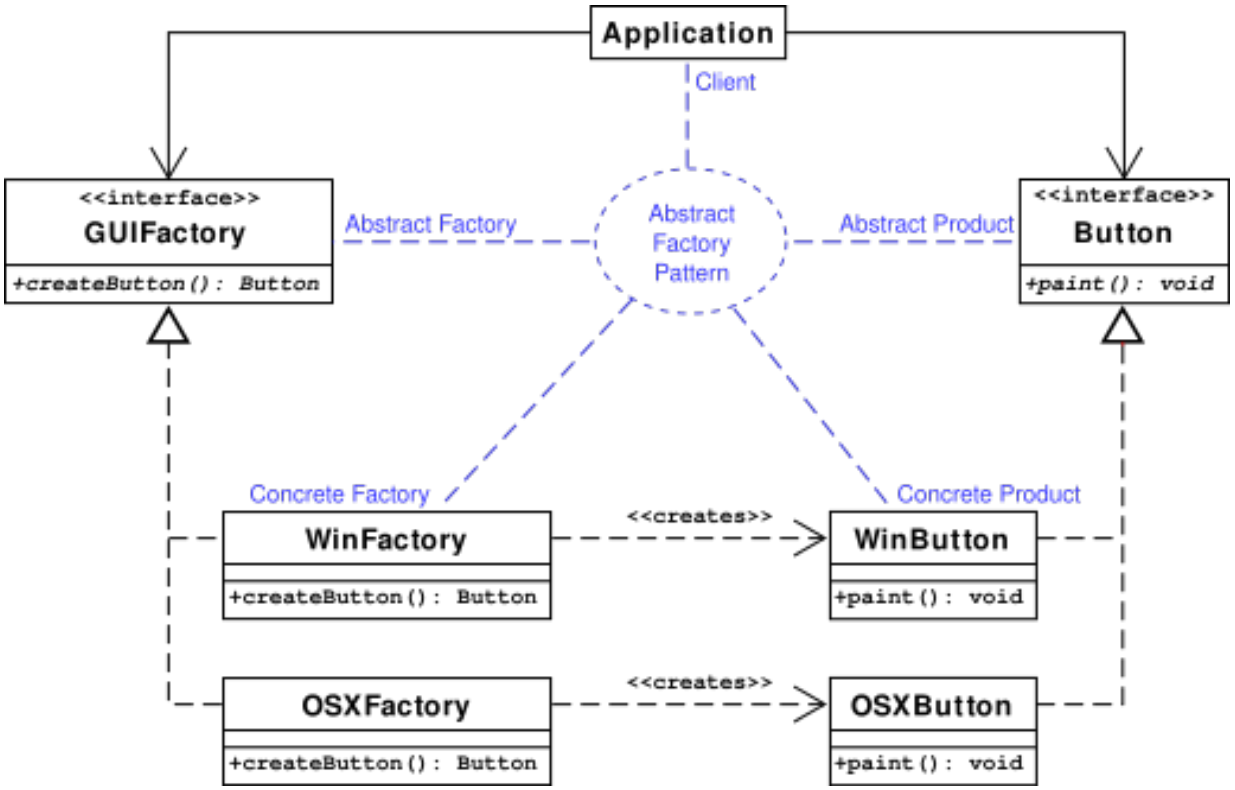


FIGURE 4.3 – Structure du ABSTRACT FACTORY.

## Exemple
ABSTRACT FACTORY *1/4*

```
interface GUIFactory { //Abstract Factory
    Button createButton();
    Label createLabel();
}

interface Button { //Abstract Product
    void paint();
}

interface Label { //Abstract Product
    void paint();
}
```

80

## Exemple
ABSTRACT FACTORY *2/4*

```
class WinFactory implements GUIFactory { //Concrete Factory
    public Button createButton() {
        return new WinButton();
    }

    public Label createLabel() {
        return new WinLabel();
    }
}

class OSXFactory implements GUIFactory { //Concrete Factory
    public Button createButton() {
        return new OSXButton();
    }

    public Label createLabel() {
        return new OSXLabel();
    }
}
```

81

### Exemple
ABSTRACT FACTORY *3/4*

```
class OSXButton implements Button { //Concrete Product
    public void paint() {
        System.out.println("I'm an OSXButton");
    }
}

class WinButton implements Button { //Concrete Product
    public void paint() {
        System.out.println("I'm a WinButton");
    }
}

class OSXLabel implements Label { //Concrete Product
    public void paint() {
        System.out.println("I'm an OSXLabel");
    }
}

class WinLabel implements Label { //Concrete Product
    public void paint() {
        System.out.println("I'm a WinLabel");
    }
}
```

82

### Exemple
ABSTRACT FACTORY *4/4*

```
// Le client se contente de choisir la fabrique concrète à utiliser
class Application {
    public Application(GUIFactory factory) {
        Button button = factory.createButton();
        Label label = factory.createLabel();
        button.paint();
        label.paint();
    }
}

public class ApplicationRunner {
    public static void main(String[] args) {
        new Application(createOsSpecificFactory());
    }

    public static GUIFactory createOsSpecificFactory() {
        String osname = System.getProperty("os.name").toLowerCase();
        if(osname != null && osname.contains("windows"))
            return new WinFactory();
        else
            return new OSXFactory();
    }
}
```

83

# 4.3 Patrons de structure

## 4.3.1 Composite

### Le patron de conception Composite

### Objectif
COMPOSITE permet de manipuler un groupe d'objets de la même façon qu'un objet simple.

— Il permet de créer des structures hiérarchiques pour des relations *tout-partie*.

84

### Structure du Composite
(voir figure 4.4).

85

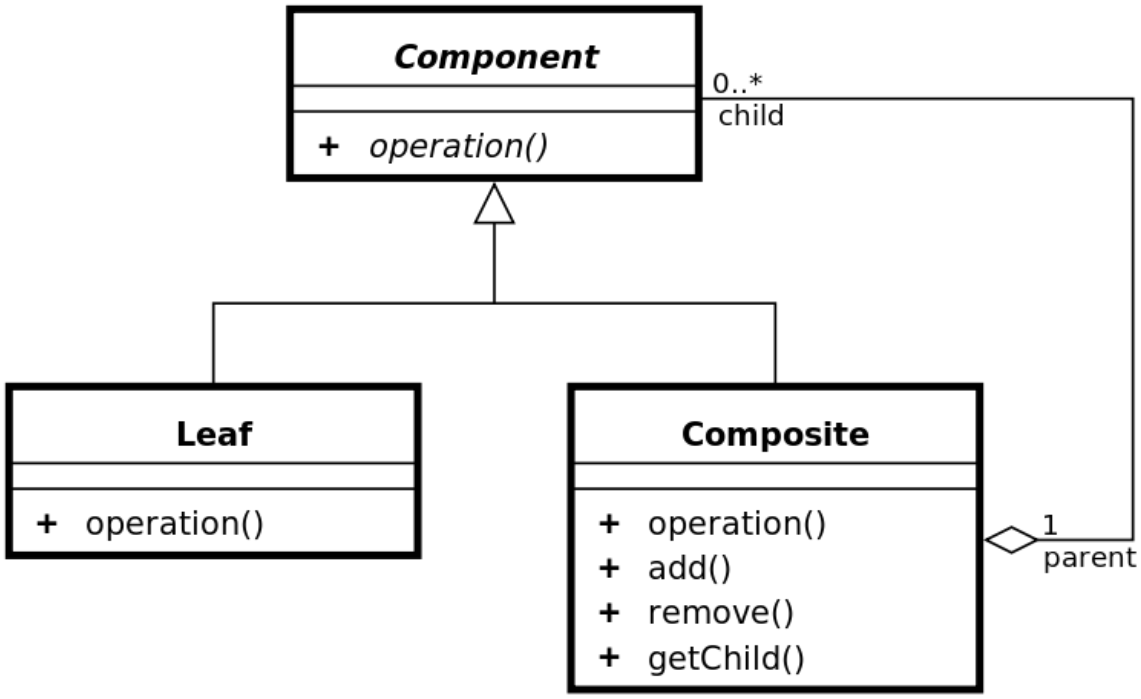


FIGURE 4.4 – Structure du COMPOSITE.

### Exemple 1/4
*Une forme en Java*

```
interface Graphic { // Component
    public void print();
}
```

86

### Exemple 2/4
*Ellipse dérive de forme*

```
class Ellipse implements Graphic { // Leaf
    @Override
    public void print() {
        System.out.println("Ellipse");
    }
}
```

87

### Exemple 3/4
*Le composite dérive de forme*

```
class CompositeGraphic implements Graphic { // Composite
    private List<Graphic> childGraphics = new ArrayList<Graphic>();

    @Override
    public void print() {
        for (Graphic graphic : childGraphics) {
            graphic.print();
        }
    }

    public void add(Graphic graphic) {
        childGraphics.add(graphic);
    }
```

```
    public void remove(Graphic graphic) {
        childGraphics.remove(graphic);
    }
}
```

88

### Exemple 4/4

*Utilisation*

```
Ellipse ellipse1 = new Ellipse();
Ellipse ellipse2 = new Ellipse();
Ellipse ellipse3 = new Ellipse();
Ellipse ellipse4 = new Ellipse();

CompositeGraphic graphic = new CompositeGraphic();
CompositeGraphic graphic1 = new CompositeGraphic();
CompositeGraphic graphic2 = new CompositeGraphic();

graphic1.add(ellipse1);
graphic1.add(ellipse2);
graphic1.add(ellipse3);

graphic2.add(ellipse4);

graphic.add(graphic1);
graphic.add(graphic2);

graphic.print(); // affiche la hiérarchie
```

89

## 4.3.2 Adapter

### Le patron de conception Adapter

### Objectif

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

(voir figure 4.5).

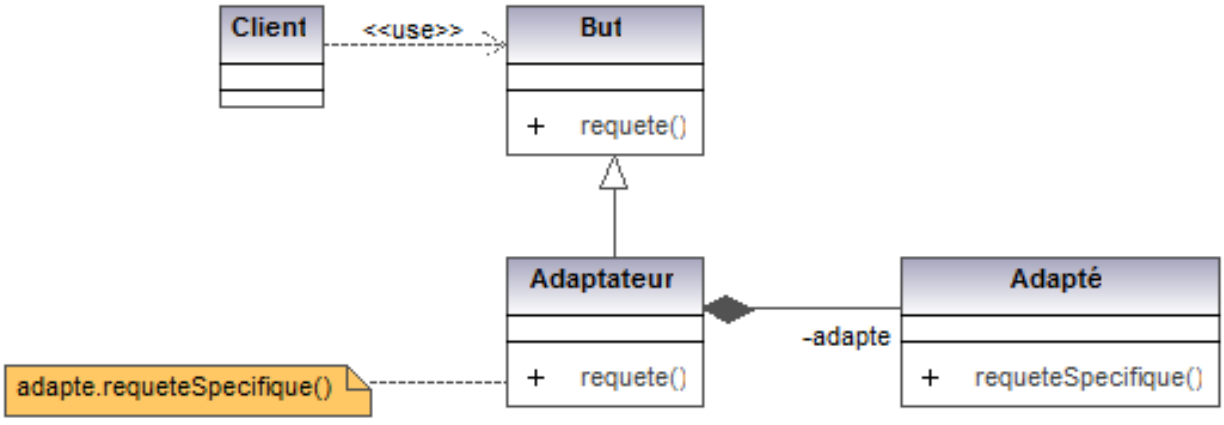


FIGURE 4.5 – Structure du ADAPTER.

90

### Exemple

*Les classes* wrapper *pour les types de bases en Java*

(voir figure 4.6).

91

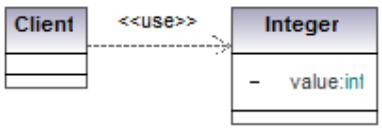


FIGURE 4.6 – `Integer` vu comme un ADAPTER.

**Exemple 1/2**
*Écouteur d'évènements et* ADAPTER

```
public class MouseBeeper extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }
}
```

92

**Exemple 2/2**
*Écouteur d'évènements et* ADAPTER

```
public class MouseBeeper implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        Toolkit.getDefaultToolkit().beep();
    }

    public void mousePressed(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
}
```

93

### 4.3.3 Decorator

**Le patron de conception Decorator**

**Objectif**
DECORATOR étend dynamiquement les fonctionnalités d'un objet.

— Ce pattern apporte une aide pour respecter le principe SRP
— C'est une alternative à l'héritage pertinente en particulier quand les possibilité d'extension sont nombreuses et indépendantes

94

**Structure du Decorator**
(voir figure 4.7).

95

**Exemple**
*`BufferedInputStream` vu comme un* DECORATOR
(voir figure 4.8).

96

**Exemple**
*Flux de filtrage et* DECORATOR

— Un flux de filtrage est construit à partir d'un autre flux selon le modèle de conception DECORATOR
— Le flux résultant propose des fonctionnalités plus riches que le flux initial

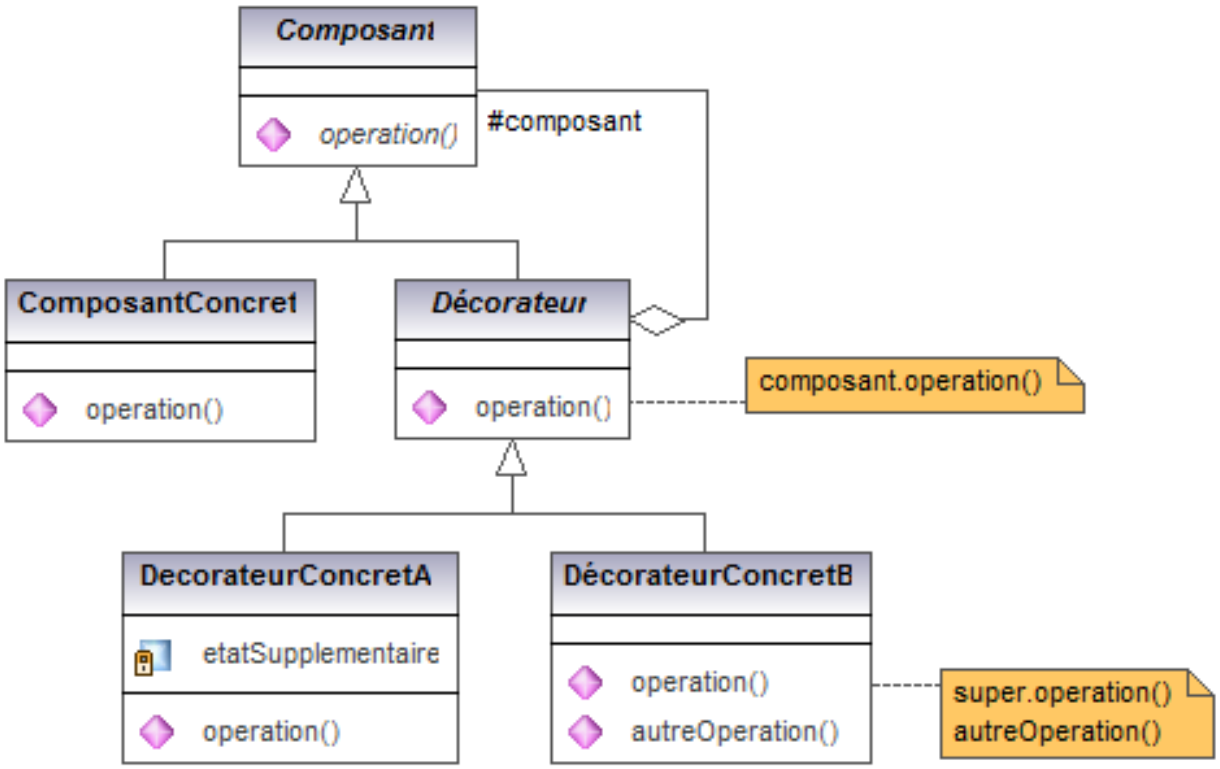


FIGURE 4.7 – Structure du DECORATOR.

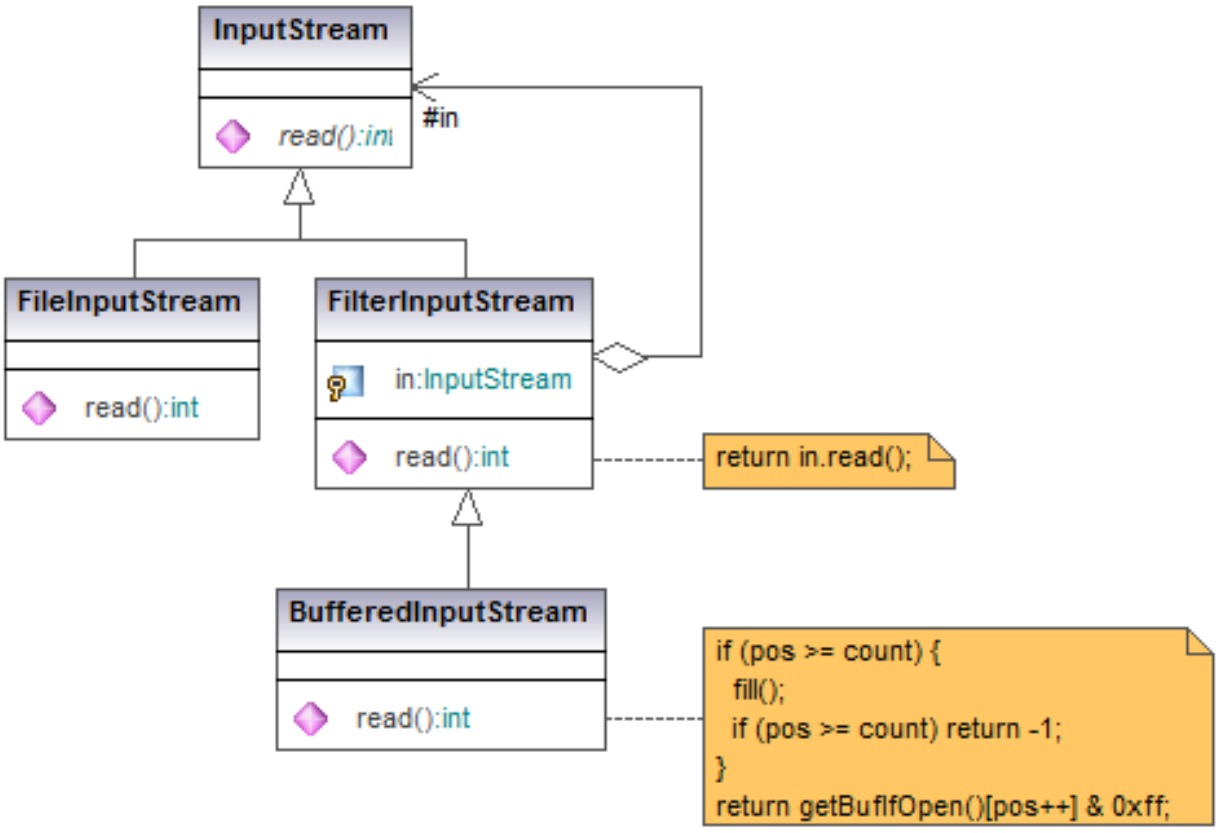


FIGURE 4.8 – `BufferedInputStream`.

```
FileReader words = new FileReader("words.txt");
BufferedReader in = new BufferedReader(words);
```

97

### 4.3.4 Facade

#### Le patron de conception Facade

**Objectif**

FACADE fournit une interface simplifiée pour un ensemble complexe de classes (bibliothèque par exemple).

— Simplifie l'usage et la compréhension d'une bibliothèque (adaptation de l'interface au contexte)

— Réduit le couplage entre les clients et les classes de la bibliothèque

98

**Structure du Facade**

(voir figure 4.9).

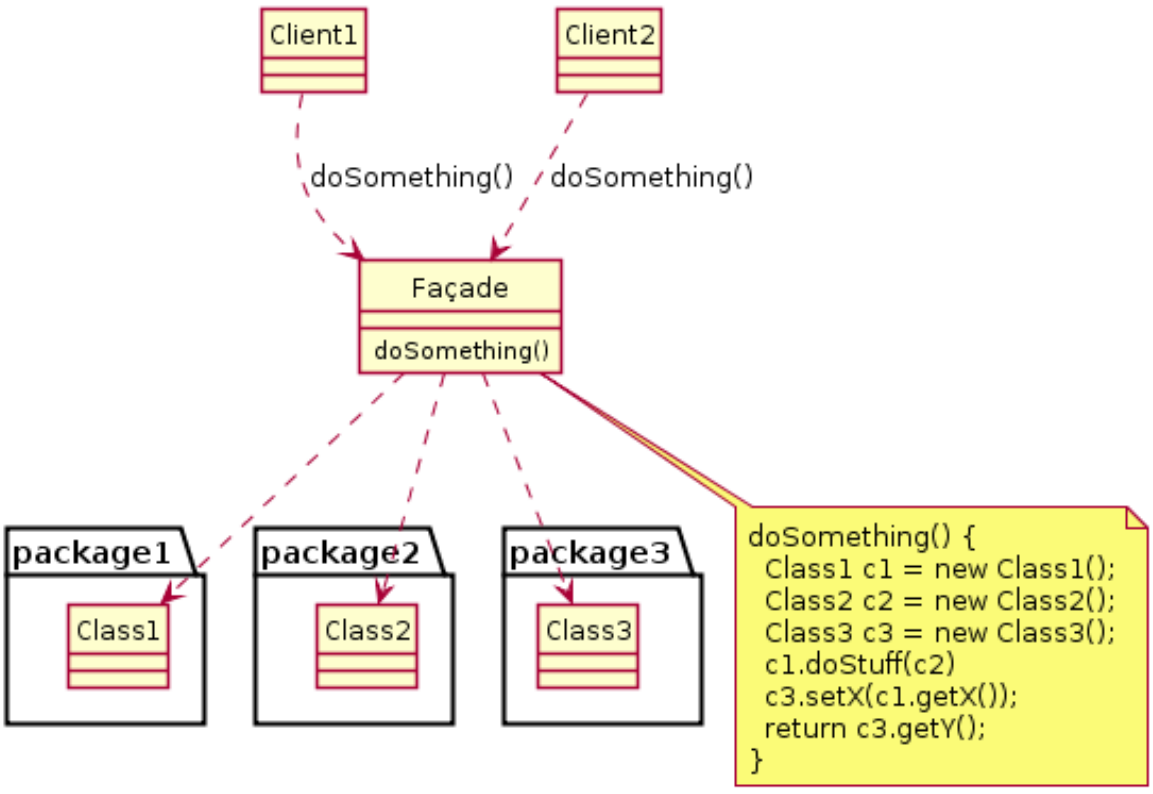


FIGURE 4.9 – Structure du FACADE.

99

### 4.3.5 Bridge

#### Le patron de conception Bridge

**Objectif**

BRIDGE découple une abstraction de son implémentation afin que les deux éléments puissent être modifiés indépendamment l'un de l'autre.

(voir figure 4.10).

100

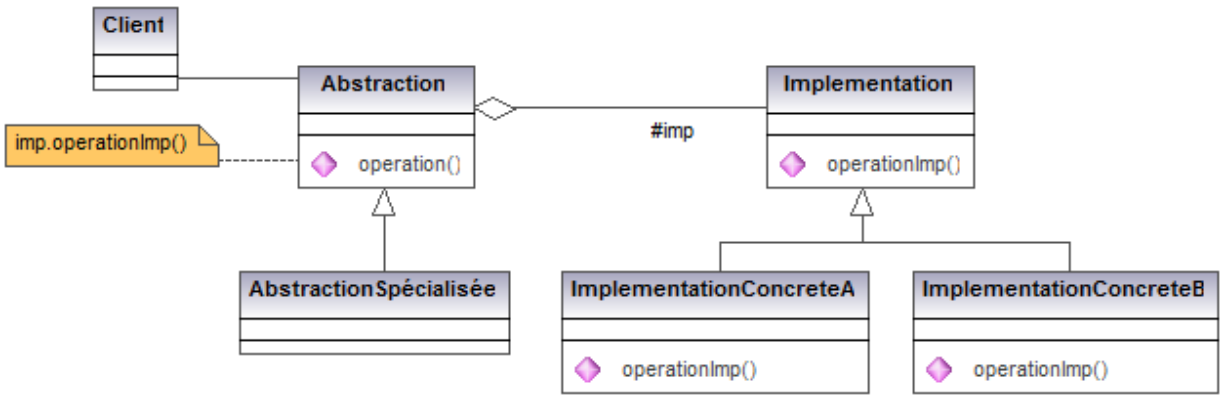


FIGURE 4.10 – Structure du BRIDGE.

**Exemple**
*Interfaces et implémentations dans la bibliothèque de collections Java*
(voir figure 4.11).

101

## 4.4 Patrons de comportement

### 4.4.1 Command

#### Le patron de conception Command

**Objectif**
COMMAND utilise un objet pour encapsuler les informations pour réaliser une action. Le *client* transmet la *commande* (*command*) à l'*appelant* (*invoker*) qui se charge de l'exécuter en interagissant avec le récepteur (*receiver*).

— Usages communs : action d'une GUI (classe `Action` en Java), enregistrement de macros, *undo* sur plusieurs niveaux

102

**Structure du Command**
(voir figure 4.12).

103

**Exemple 1/3**
COMMAND *en Java 8*

```
@FunctionalInterface
public interface Command { // Command
    public void apply();
}
```

104

**Exemple 2/3**
COMMAND *en Java 8*

```
public class CommandFactory {
    private final Map<String, Command>  commands;

    private CommandFactory() {
        this.commands = new HashMap<>();
    }

    public void addCommand(String name, Command command) {
        this.commands.put(name, command);
    }
```

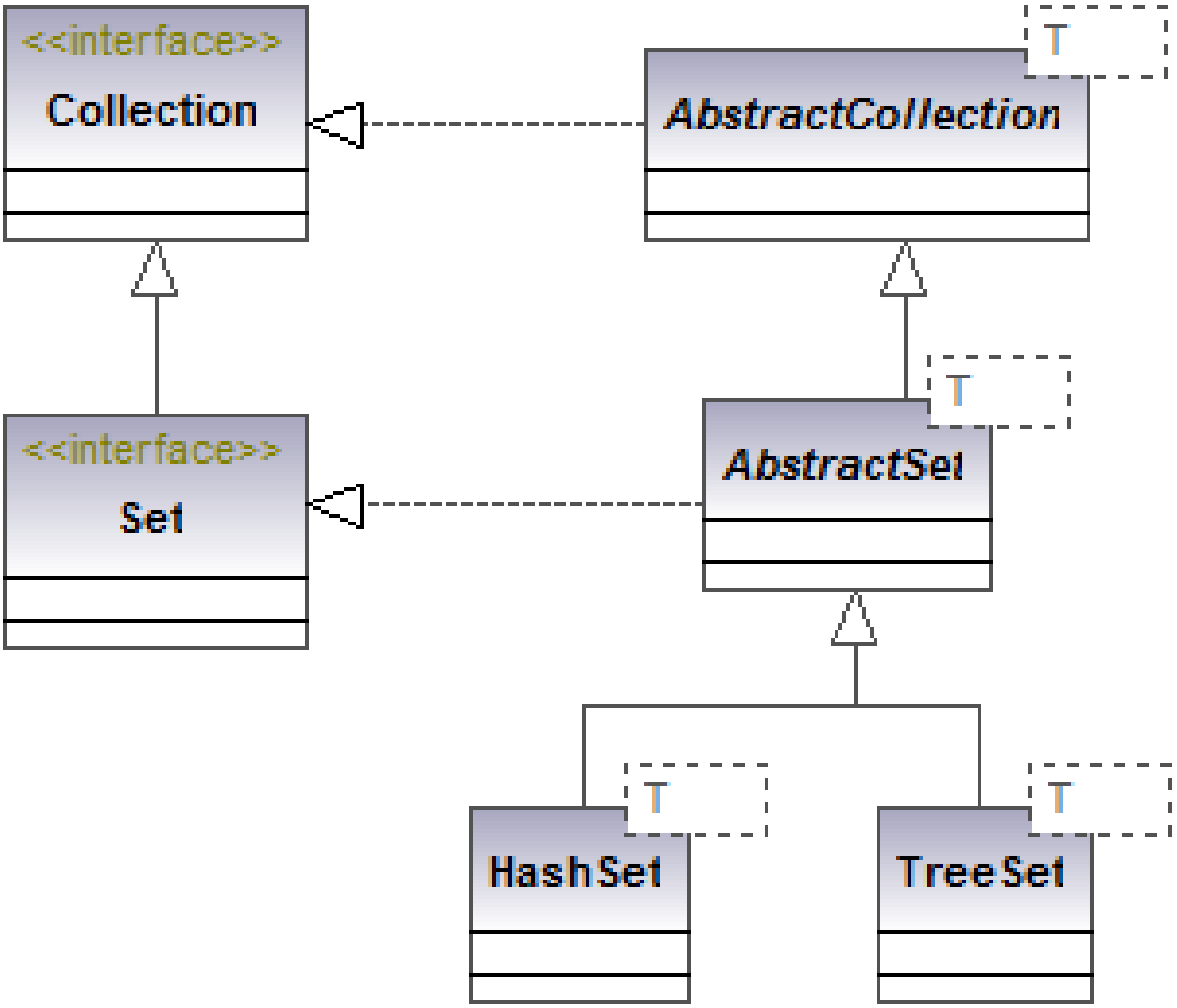


FIGURE 4.11 – BRIDGE dans la bibliothèque de collections en Java.

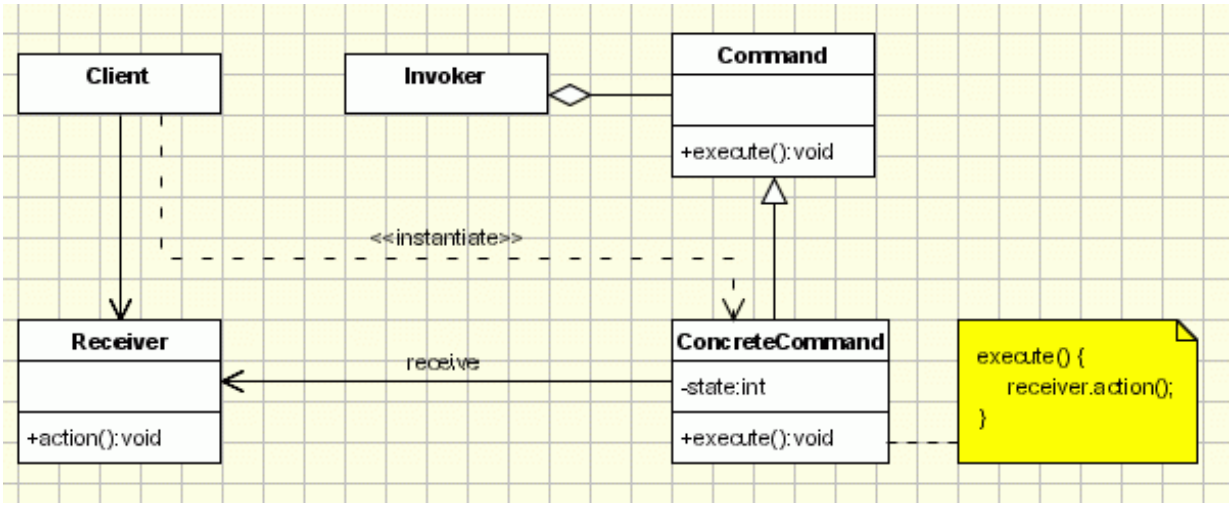


FIGURE 4.12 – Structure du COMMAND.

```
    public void executeCommand(String name) {
        if (this.commands.containsKey(name)) {
            this.commands.get(name).apply();
        }
    }

    public static CommandFactory init() {
        CommandFactory cf = new CommandFactory();
        cf.addCommand("Light on", () -> System.out.println("Light turned on"));
        cf.addCommand("Light off", () -> System.out.println("Light turned off"));
        return cf;
    }
}
```

105

### Exemple
*Usage*

```
public class Main {
    public static void main(String[] args) {
        CommandFactory cf = CommandFactory.init();
        cf.executeCommand("Light on");
        cf.executeCommand("Light off");
    }
}
```

106

## 4.4.2 Iterator

### Le patron de conception Iterator

### Objectif
ITERATOR fournit un moyen d'accès séquentiel aux éléments d'un agrégat d'objets sans mettre à découvert la représentation interne de ce dernier.

107

### Structure de Iterator
(voir figure 4.13).

108

### Exemple de Iterator 1/2
*Un itérateur de la bibliothèque Java*
(voir figure 4.14).

109

### Exemple de Iterator 2/2
*Un itérateur de la bibliothèque Java*

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
    void remove();    // Optional
}

//...

for (Iterator<String> i = uneCollectionDeChaines.iterator(); i.hasNext(); ) {
    String element = i.next(); // Récupère l'élément et passe au suivant
    // ...
}
```

110

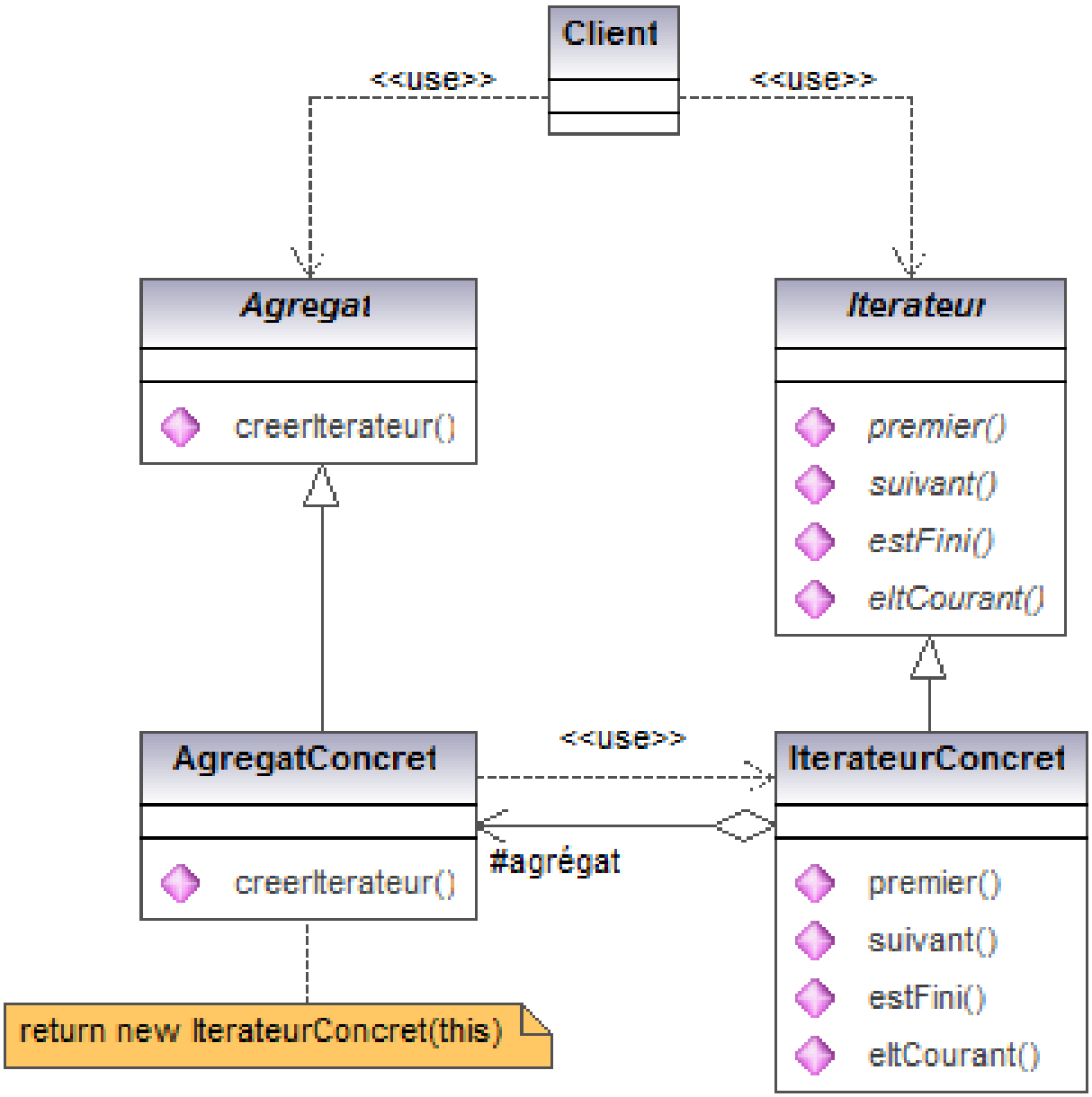


FIGURE 4.13 – Structure du ITERATOR.

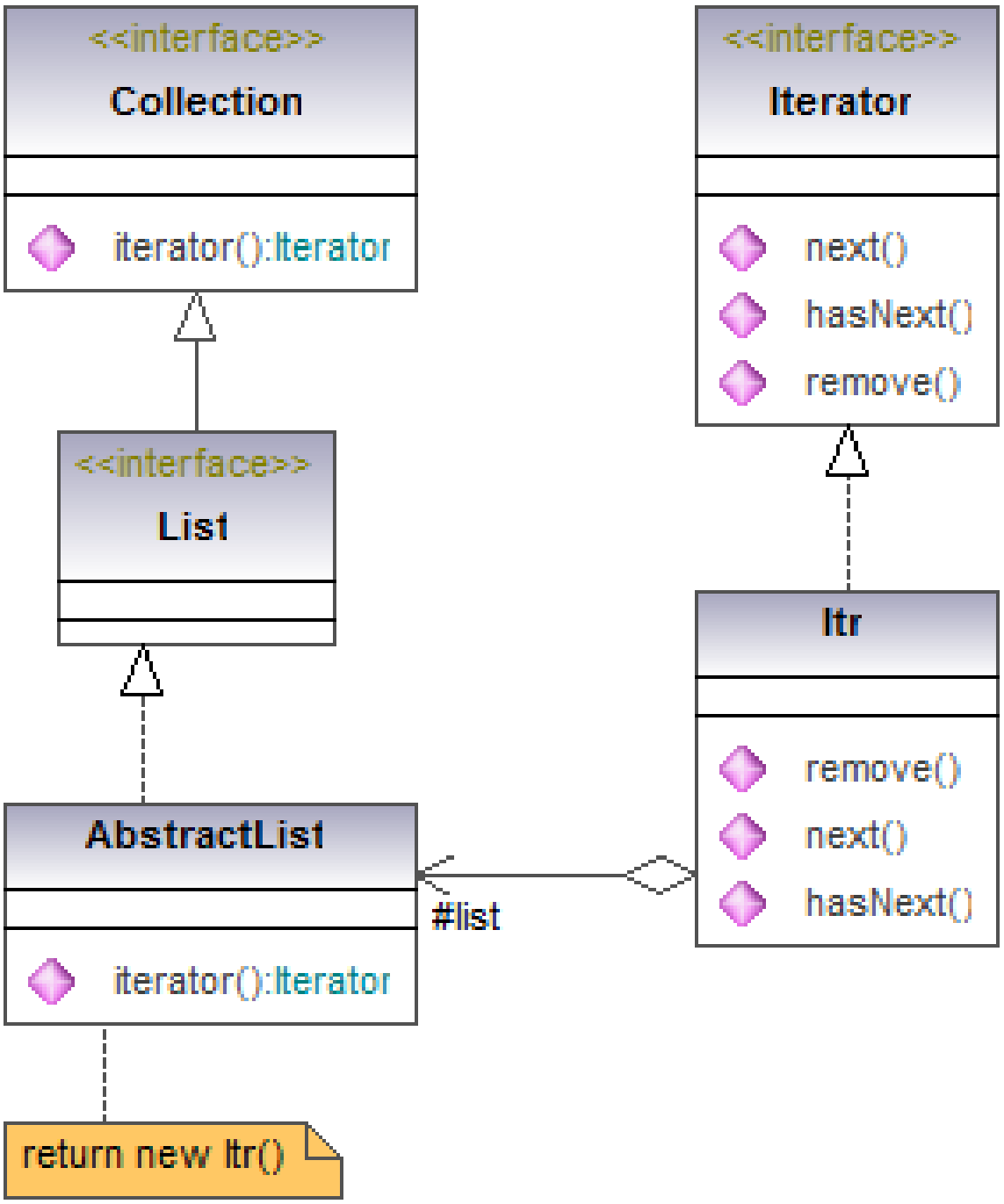


FIGURE 4.14 – ITERATOR en Java.

### 4.4.3 Observer

#### Le patron de conception Observer

**Objectif**

Observer fait en sorte que quand un objet change d'état, tout ceux qui en dépendent en soient notifiés et automatiquement mis à jour.

(voir figure 4.15).

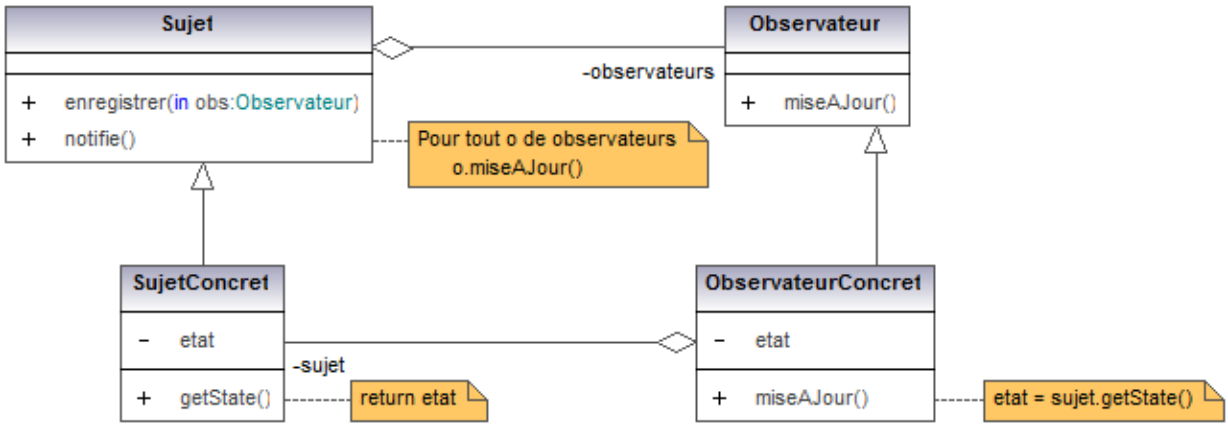


Figure 4.15 – Structure du Observer.

111

#### Observer en Java 1/2

*La classe* `Observable`

- Un modèle est une classe qui étend `java.util.Observable` (fournit l'infrastructure d'enregistrement/notification)
- Les accesseurs et mutateurs appropriés doivent ensuite être définis
- Chaque mutateur doit appeler les méthodes `void setChanged()` et `void notifyObservers()` (ou `void notifyObservers(Object arg)`)
- Le modèle peut éventuellement être découpé en plusieurs classes dérivants de `Observable`

112

#### Observer en Java 2/2

*L'interface* `Observer`

- Un vue doit implémenter l'interface `java.util.Observer`
- Cette interface déclare la méthode `void update(Observable o, Object arg)`
- La vue doit aussi être enregistrée auprès du modèle en appelant la méthode `void addObserver(Observer o)` du modèle afin d'être notifiée des changements par la suite
- Une vue peut être enregistrée auprès de plusieurs modèles

113

### 4.4.4 Template method

#### Le patron de conception Template method

**Objectif**

Template method définit le squelette d'un algorithme dans une méthode qui délègue certaines étapes aux sous-classes.

114

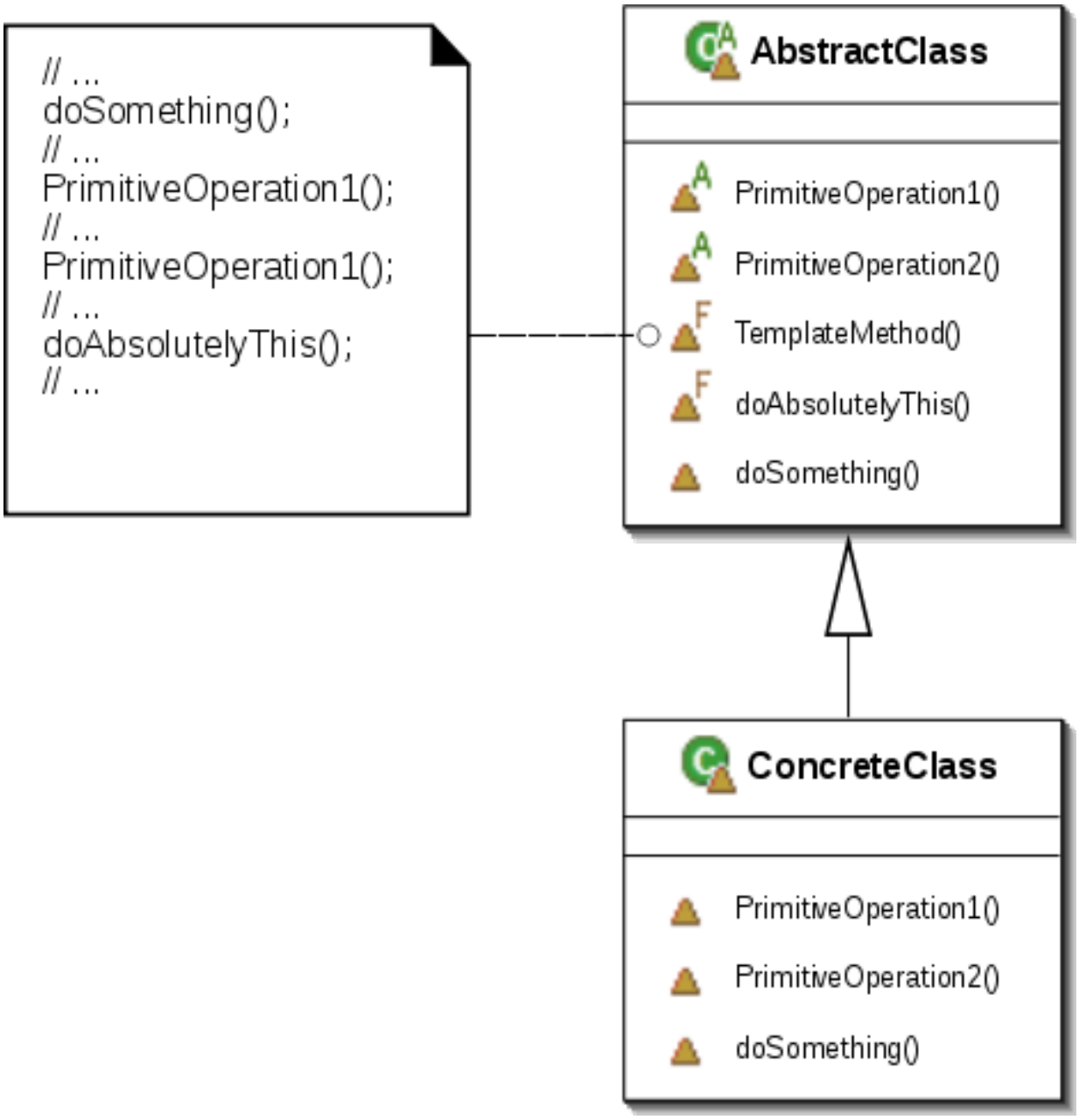


FIGURE 4.16 – Structure de TEMPLATE METHOD.

**Structure de Template method**

(voir figure 4.16).

115

**Exemple de Template method**

*Une classe* Game *abstraite*

```
abstract class Game {
    protected int playersCount;

    abstract void initializeGame();
    abstract void makePlay(int player);
    abstract boolean endOfGame();
    abstract void printWinner();

    public final void playOneGame(int playersCount) { // Template method
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j);
            j = (j + 1) % playersCount;
        }
        printWinner();
    }
}
```

116

**Exemple**

*Le jeu de Monopoly*

```
class Monopoly extends Game {
    void initializeGame() {
        // Initialize players
        // Initialize money
    }

    void makePlay(int player) {
        // Process one turn of player
    }

    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }

    void printWinner() {
        // Display who won
    }

    /* Specific declarations for the Monopoly game. */
}
```

117

## 4.5 Bibliographie

GAMMA, Erich et al. (1999). *Design Patterns - Catalogue de modèles de conceptions réutilisables*. Français. Vuibert.

LARMAN, Craig (2005). *UML 2 et les design patterns*. Français. 3e édition. Pearson education. ISBN : 978-2-7440-7090-7. URL : http://www.craiglarman.com/.

## 4.6 Exercices

CONTRAINTES

— Dans ces exercices, vous utiliserez systématiquement un SINGLETON pour représenter le programme principal.

**Exercice 4.1** (Builder, Composite, Iterator)
Dans cet exercice, vous réaliserez un annuaire des personnels d'une organisation.

Un personnel possède un nom, un prénom, une date de naissance, des numéros de téléphone (fixe pro, fixe perso, portable, ...) et des fonctions (par exemple directeur de XXX, chargé de mission XXX, ...).

L'annuaire reflétera la structure hiérarchique de l'organisation (par exemple, département/service/équipe). Chaque niveau peut comporter des individus ou des sous-hiérarchies. Les différents niveaux de structuration ne sont pas connus à priori.

On veut pouvoir afficher les personnels de l'organisation de deux manières : hiérarchiques (en profondeur) et par groupe (en largeur).

1. Représentez les personnels par une classe immuable `Personnel`. Le nom, le prénom et les fonctions seront implémentés par des chaînes de caractères, la date de naissance par `java.time.LocalDate` et les numéros de téléphone par une collection d'un type que vous définirez. L'initialisation d'un personnel respectera le pattern Builder
2. Définissez la notion de groupe de personnels en vous appuyant sur le pattern Composite
3. Implémentez les deux types d'affichage en définissant deux stratégies de parcours de la structure. Ces dernières se baseront sur le pattern Iterator. Plus précisément, vous vous appuierez sur les interfaces `java.util.iterator` et `java.lang.iterable`.

**Exercice 4.2** (Command)
Dans cet exercice, on souhaite réaliser une calculatrice fonctionnant en mode RPN (*Reverse Polish Notation*). Cette notation post-fixée permet de représenter des formules arithmétiques sans parenthèses. Par exemple, l'expression « $2 \times (3 + 4)$ » pourra s'écrire « $234 + \times$ ».

Cette calculatrice devra supporter les opérations de base (`+`, `-`, `*`, `/`) sur des nombres entiers. L'interface utilisateur utilisera un interpréteur en mode texte. L'utilisateur saisira au clavier soit un nombre, soit une opération, soit `undo` pour annuler la saisie précédente, soit `exit` pour sortir. Chaque saisie se terminera par *entrée*.

L'implémentation pourra utiliser une pile de la façon suivante :

— les opérandes sont empilées lors de leur saisie,

— les opérations sont effectuées immédiatement en considérant les opérandes se trouvant au sommet de la pile,

— le résultat d'une opération est empilé.

Après chaque saisie, l'interpréteur affichera le contenu de la pile.

Pour la conception, vous pourrez consulter Example calculator design.

1. Implémentez un interpréteur générique `Interpreteur` qui supporte uniquement les commandes `undo` et `quit`. La commande `quit` stoppe le programme. La commande `undo` supprime la dernière commande de l'historique. Vous utiliserez le pattern Command pour implémenter les actions.
2. Dérivez la classe `MoteurRPN` de l'interpréteur. Elle devra permettre de :
   — enregistrer une opérande,
   — appliquer une opération sur les opérandes,
   — retourner l'ensemble des opérandes stockées.

   Vous utiliserez le pattern Command pour implémenter les actions.
3. Implémenter la classe `SaisieRPN` qui gère les interactions avec l'utilisateur et invoque le moteur RPN. La classe `java.util.Scanner` permet de gérer les saisies.
4. Implémenter le programme principal `CalculatriceRPN`.

# Chapitre 5

# Persistance des objets

## Sommaire



## 5.1 Introduction

### Qu'est-ce que la persistance ?

**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

118

### Difficultés liées à la persistance

— La persistance nécessite de définir des correspondances (*mappings*) entre des structures en mémoire et des structures persistantes
  — peut être complexe si les modèles diffèrent (c'est le cas en général)
— Comment sauvegarder l'état d'un objet ?

— Comment gérer les références (en mémoire) ?
— Comment gérer les *handles* vers des ressources (fichiers, ...) ?
— Comment recréer les bonnes structure lors de la restauration ?

119

**Object-Relational impedance mismatch**

*Les applications sont écrites dans un langage objet* | *Les données sont stockées dans des SGBD relationnels*

| | Modèle objet | Modèle relationnel |
|---|---|---|
| Fondements | Concepts OO | Théorie des ensembles |
| Types | Complexes et déf. par l'utilisateur | 1NF et prédéfinis |
| Accès | Navigation | Requête |
| Identification | id. d'objet | Clé |

120

**Alternatives pour la persistance**

— Bibliothèque d'E/S du langage + sérialisation
— Interfaçage avec un SGBD
  — API de bas niveau (JDBC, ODBC, ...)
  — API de haut niveau (JPA, JDO, SDO)
  — *Mapping objet-relationnel* (ORM)
  — API REST

121

**Comparaison des approches**

| Approche | Simplicité | Transparence | Contrôle | Fonctionnalités |
|---|---|---|---|---|
| Sérialisation | + | – | ++ | – |
| API de bas niveau | ++ | – | ++ | + |
| API de haut niveau | - | ++ | - | + |
| ORM | - | ++ | - | ++ |

122

# 5.2 Sérialisation

## 5.2.1 Introduction

### Qu'est-ce que la sérialisation ?

**Sérialisation**

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

123

### 5.2.2 Rappels sur les entrées/sorties en Java

#### Vue d'ensemble

— Le package `java.io` (JDK 1.0) fournit les opérations de base pour les I/O
— Les packages `java.nio` (JDK 1.4) proposent des fonctionnalités plus avancées
— Le package `java.nio.file` (JDK 1.7) enrichit l'interface avec le système de fichiers

124

#### `java.io`

— I/O par flux
  — `InputStream`/`OutputStream` pour les flux d'octets
  — `Reader`/`Writer` pour les flux de caractères
  — `FilterXXX` pour les flux de filtrage
  — `Console` et `PrintStream` pour l'entrée et la sortie standard
  — `ObjectInputStream`/`OutputStream` pour la **sérialisation**
— Accès aléatoires (`RandomAccessFile`)
— Accès au système de fichiers (`File`) (remplacé et étendu par `java.nio.file`)

125

#### `java.nio.file`

— `Path` représente un chemin dans le système de fichiers
— `FileSystems`, `Paths` et `Files` pour manipuler le système de fichiers
— Accès au contenu (voir figure 5.1).

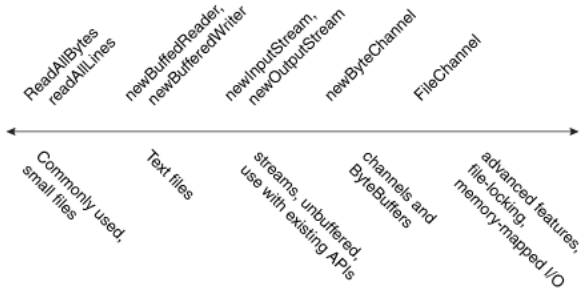


FIGURE 5.1 – Accès au contenu d'un fichier.

— `FileVisitor` pour parcourir une hiérarchie de répertoires

126

### 5.2.3 Sérialisation en Java

#### Flux d'objets

— La sérialisation est assurée par les flux d'objets
  — `ObjectInputStream` et `ObjectOutputStream`
— `ObjectOutputStream` implémente l'interface `ObjectOutput` (sous interface de `DataOutput`)
— `ObjectInputStream` implémente l'interface `ObjectInput` (sous interface de `DataInput`)
— Les méthodes `readObject` et `writeObject` permettent de lire et d'écrire des objets
— Lors de la lecture, un *cast* est en général nécessaire
  — `ClassNotFoundException` en cas d'échec

127

### Écrire des objets

```
try (ObjectOutputStream out = new ObjectOutputStream(
                                  new BufferedOutputStream(
                                      new FileOutputStream(dataFile)))) {
    out.writeObject(Calendar.getInstance());
    for (int i = 0; i < prices.length; i++) {
        out.writeObject(prices[i]);
        out.writeInt(units[i]);
        out.writeUTF(descs[i]);
    }
}
```

128

### Lire des objets

```
try (ObjectInputStream in = new ObjectInputStream(
                                new BufferedInputStream(
                                    new FileInputStream(dataFile)))) {
    Calendar date = (Calendar)in.readObject();
    try {
        while (true) {
            BigDecimal price = (BigDecimal) in.readObject();
            int unit = in.readInt();
            String desc = in.readUTF();
        }
    } catch (EOFException e) {}
}
```

129

### Rendre une classe sérialisable

— Un objet est sérialisable uniquement si sa classe implémente l'interface `Serializable`

— L'interface `Serializable` ne comporte aucune méthode et ne sert qu'à marquer les classes sérialisables

130

### Contrôler la sérialisation

— La version d'un classe pour la sérialisation est gérée par l'attribut `static final long serialVersionUID`
  — en cas de différence à la lecture, une exception est levée (`InvalidClassException`)

— Le comportement par défaut est fourni par les méthodes `defaultWriteObject` de `ObjectOutputStream` et `defaultReadObject` de `ObjectInputStream` et stocke
  — la classe de l'objet,
  — la signature de la classe,
  — la valeur des attributs d'instances y compris les références (mais pas les attributs `transcient`).

— Il est possible d'adapter le comportement par défaut en redéfinissant `writeObject` et `readObject`

— L'interface `Externalizable` permet d'avoir un contrôle complet du processus de sérialisation

131

### Autres bibliothèques de sérialisation pour Java

— FlexJSON (JSON)

— Google GSON (JSON)

— Kryo (format spécifique)

— SOJO (JSON, XML, CSV)

132

# 5.3 JDBC : une API d'accès bas niveau

## 5.3.1 Introduction

### But des API d'accès bas niveau

— Ces API fournissent un moyen pour interfacer un programme avec un SGBD
— Elles sont conçues pour que le programme soit indépendant d'un SGBD spécifique
  — un *driver* se charge de la traduction vers un SGBD spécifique
— Les API les plus connus sont ODBC et JDBC
  — ciblent les SGBDR

### JDBC

— *Java database connectivity technology* (*JDBC*) a été introduit dans le JDK 1.1
— Est contenu dans les modules `java.sql` et `javax.sql`

### Architecture

(voir figure 5.2).

### Driver

— Le *driver JDBC* permet de communiquer avec un SGBD particulier
— 4 types
  **Type 1** (*JDBC-ODBC bridge*) s'appuie sur un driver ODBC
  **Type 2** écrit en Java et appel l'API native du SGBD
  **Type 3** écrit en Java et utilise un *middleware*
  **Type 4** écrit en Java et utilise le protocole natif du SGBD
— Sous la forme de fichier `jar` à inclure dans le `CLASSPATH`

## 5.3.2 Connexion à un SGBD

### Se connecter à une source de données

— Une *source de données* peut être un SGBD ou un autre système disposant d'un *driver JDBC*
— La connexion à un source de données nécessite :
  — de charger le driver JDBC,
  — de construire la *chaîne de connexion* pour cette source,
  — d'établir la connexion avec `DriverManager` ou `DataSource`.

### Chargement du driver JDBC

— *L'archive `jar` du driver doit être dans le `CLASSPATH`*
— À partir de la version 4.0 de JDBC, le chargement du driver est automatique
  — la classe `DriverManager` charge l'ensemble des drivers disponibles dans le `CLASSPATH`
— Avec les versions précédentes

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver"); // pour Derby ou
Class.forName("org.postgresql.Driver"); // pour PostgreSQL ou
Class.forName("com.mysql.jdbc.Driver"); // pour MySQL
```

— Un autre possibilité est d'utiliser la propriété `jdbc.drivers`

```
java -Djdbc.drivers=le.driver.jdbc applicationClass
```

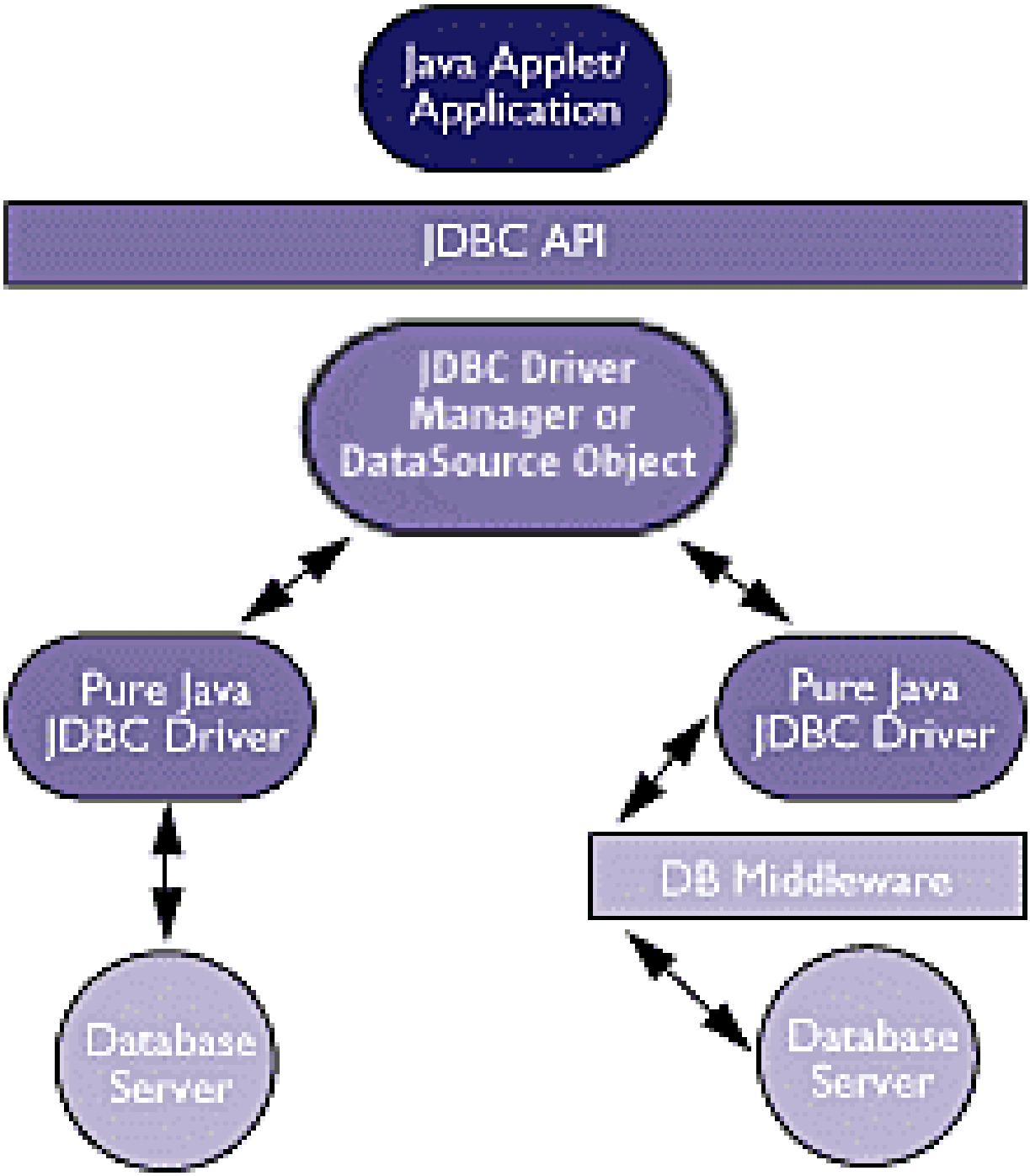


FIGURE 5.2 – Architecture de JDBC.

### Chaîne de connexion

— La chaîne de connexion (*database connection URL*) est une chaîne de caractères précisant les informations nécessaires pour se connecter au SGBD

— Elle est spécifique à la chaque driver mais débute toujours par *jdbc* :

— Par exemple

— pour Derby, `jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*`

```
String dburl = "jdbc:derby:test;create=true";
```

— pour PostgreSQL, `jdbc:postgresql://host[:port]/[database]`

```
String dburl = "jdbc:postgresql://localhost/test";
```

— pour MySQL, `jdbc:mysql://[host][,failoverhost...][:port]/[database]`

```
String dburl = "jdbc:mysql://localhost/test";
```

139

### Établir la connexion

— La méthode `DriverManager.getConnection` ouvre une connexion avec le SGBD

```
Properties connectionProps = new Properties();
connectionProps.put("user", userName);
connectionProps.put("password", password);

try (Connection conn = DriverManager.getConnection(
        "jdbc:somejdbcvendor:other data needed by some jdbc vendor",
        connectionProps) ) {
    // ...
}
```

— L'interface `Connection` permet l'interaction avec le SGBD

— En cas d'erreur, une exception `SQLException` est lancée

140

### Fermer la connexion

— Il est important de fermer correctement la connexion (*try-with-ressource*

```
try (Connection conn = DriverManager.getConnection(/* ... */) {
    // ...
}
```

— ou bloc *finally*)

```
Connection connection = null;

try {
    connection = DriverManager.getConnection(/* ... */);
    // ...
} finally {
    if (connection != null) {
        try { conn.close(); } catch (Exception e) { /* ignoré */ }
    }
}
```

141

### 5.3.3 Exécuter une requête

#### Créer une instruction

— L'interface `Statement` représente une instruction SQL
— Un objet `Statement` est créé à partir d'un objet `Connection`
— Il peut être exécuté et produit en général un objet `ResultSet`
— Il en existe trois types :

**`Statement`** pour une instruction SQL simple

**`PreparedStatement`** pour une instruction SQL pré-compilée avec des paramètres

**`CallableStatement`** pour une procédure stockée

142

#### Instruction pré-compilée

— Une *instruction pré-compilée* (*prepared statement*) est une instruction SQL préalablement compilée par le SGBD

```
String updateString = "UPDATE " + dbName + "." + dbTable +
    "SET attr1 = ? WHERE attr2 = ?";
update = con.prepareStatement(updateString);
```

— Une telle instruction peut être paramétrée (par position marquée par `?`)

```
update.setInt(1, 12);
update.setString(2, "toto");
```

143

#### Exécuter une requête

— Une instruction est exécutée avec

**`execute`** retourne `true` si la requête retourne un (ou plusieurs) `ResultSet`

```
if(stmt.execute("...") == false) {
    int num = stmt.getUpdateCount(); // Get the update count
    // ...
} else {
    ResultSet          rs = stmt.getResultSet();
    ResultSetMetaData md = rs.getMetaData();
    // ...
}
```

**`executeQuery`** retourne un `ResultSet`

```
ResultSet rs = stmt.executeQuery(query);
```

**`executeUpdate`** retourne le nombre de tuples affectés (pour DDL et DML)

```
int numberOfAffectedRows = update.executeUpdate();
```

144

#### Traiter les résultats

— L'interface `ResultSet` permet de parcourir les résultats d'une exécution
— Il maintient un curseur sur le tuple courant
— La méthode `next` fait avancer le curseur et renvoie `false` s'il n'y a plus de tuples
— L'interface `ResultSetMetaData` décrit les colonnes d'un `ResultSet`
— L'interface `RowSet` propose une extension de `ResultSet`

145

**Types de `ResultSet`**

— Par défaut, un `ResultSet` est non modifiable et ne peut être parcouru qu'une fois
— Lors de la création de l'instruction, il est possible de modifier ce comportement par défaut
— Type
`TYPE_FORWARD_ONLY` un seul parcours (par défaut)
`TYPE_SCROLL_INSENSITIVE` tout type de parcours, ne reflète pas les modifications
`TYPE_SCROLL_SENSITIVE` tout type de parcours, reflète les modifications
— Mode de mise à jour
`CONCUR_READ_ONLY` pas de mise à jour (par défaut)
`CONCUR_UPDATABLE` mise à jour possible
— Mode de conservation
`HOLD_CURSORS_OVER_COMMIT` maintenu ouvert après un *commit*
`CLOSE_CURSORS_AT_COMMIT` fermé lors du *commit*

146

**Mise à jour à partir d'un `ResultSet`**

— Les méthodes `updateXXX` permettent de modifier le tuple courant
— La méthode `moveToInsertRow` se place sur un buffer pour l'ajout d'un tuple et `insertRow` réalise l'insertion

147

**Extensions de JDBC**

— Spring Framework JDBC abstraction (Accessing Relational Data using JDBC with Spring)
— Apache Commons DbUtils
— jDBI
— Sql2o

148

### 5.3.4 Patron de conception DAO

**Intérêt d'un pattern pour la persistance**

— Il faut définir la correspondance entre attribut des objets et données dans la base
— Le code de persistance est du code technique dont l'application ne devrait pas dépendre
— Le pattern *DAO* (*Data Access Object*) fait le lien entre la couche métier et la couche persistante

149

**Le patron de conception DAO**

— DAO permet de
  — regrouper les règles de correspondance entre les objets et le SGBD
  — isoler l'application par rapport à un SGBD spécifique
— Il consiste à ajouter des objets ayant la responsabilités de l'accès au SGBD (opérations `CRUD`)
— En général, pour chaque objet métier persistant, on crée un objet DAO correspondant

150

**Structure du DAO**

(voir figure 5.3).

151

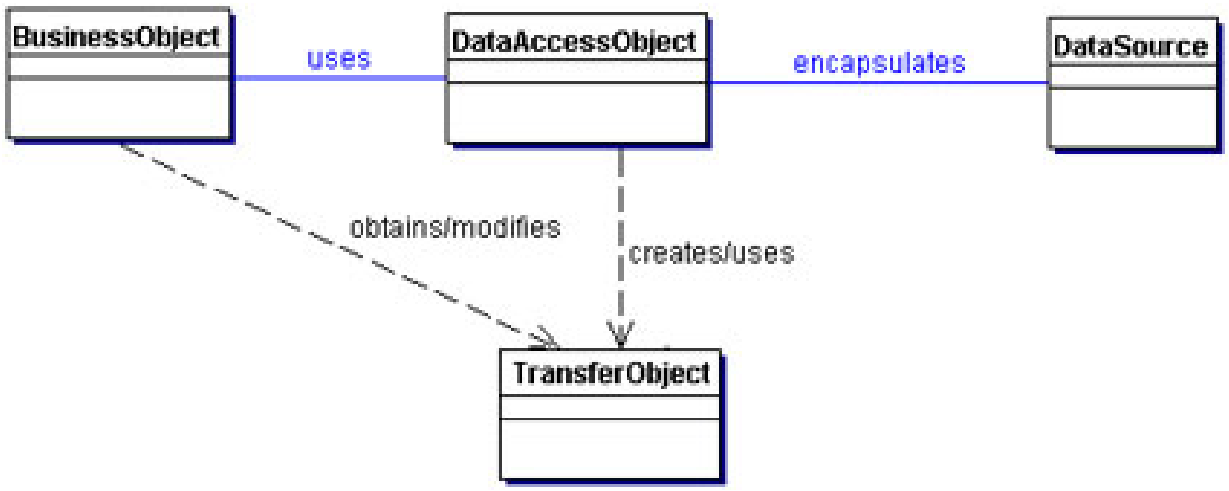


FIGURE 5.3 – Structure du pattern DAO.

### Usage du DAO

1. L'application récupère l'objet DAO approprié
2. L'application exécute une méthode de cet objet
3. L'objet DAO interagit avec le SGBD
4. L'objet DAO construit le (ou les) objets métiers
5. L'objet DAO retourne l'objet métier à l'application

152

### DAO et Factory

(voir figure 5.4).

153

### Exemple de DAO

*La classe métier* `Personne`

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

    public Personne(String nom, int age) { /* ... */ }
    @Override public String toString() { /* ... */ }
    // ...
}
```

154

### Exemple de DAO

*La classe abstraite* `DAO`

```
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);
}
```

155

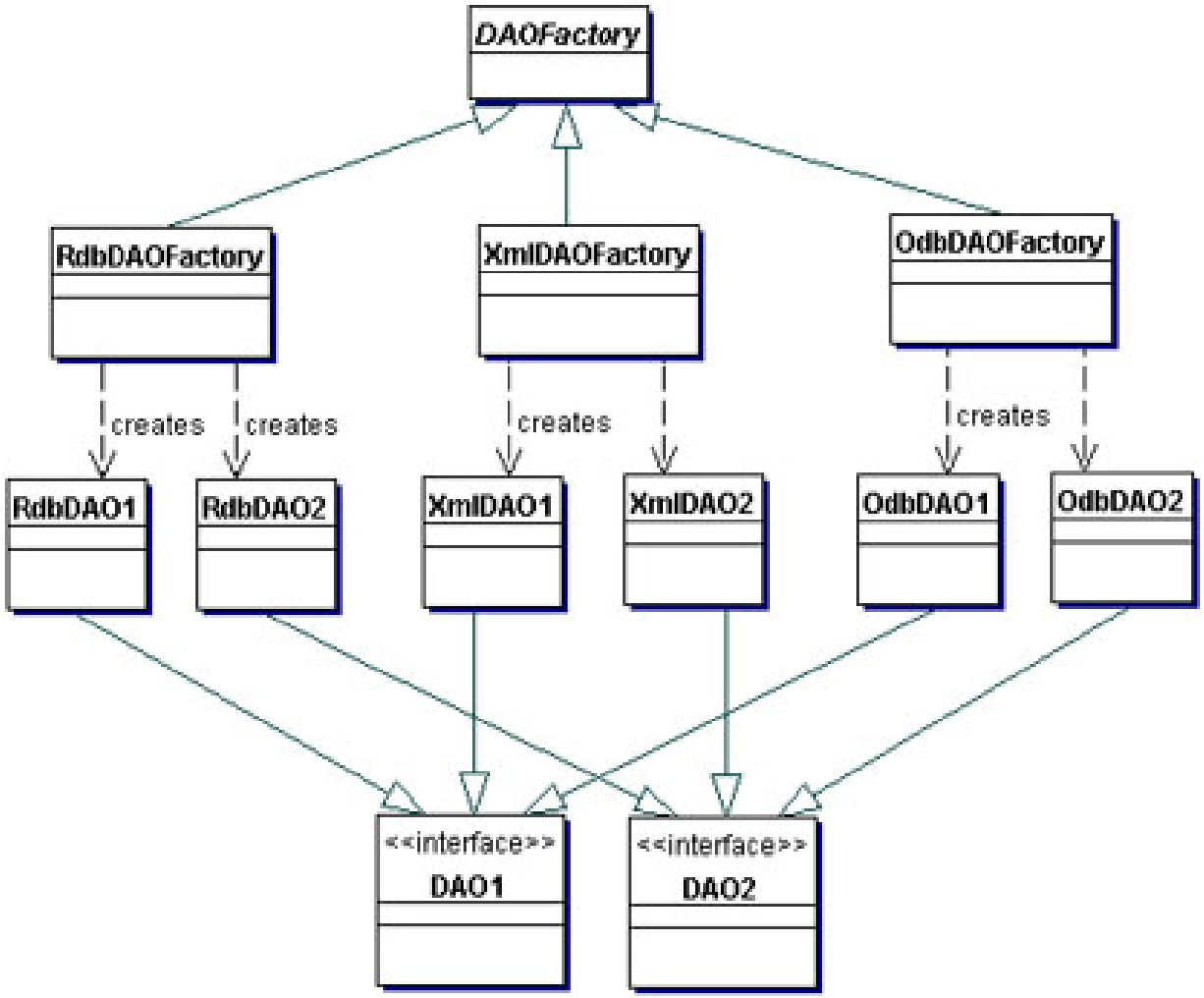


FIGURE 5.4 – Patterns DAO et Factory.

## Exemple de DAO

*La classe* `PersonneDAO`

```
public class PersonneDAO extends DAO<Personne> {
    @Override public Personne create(Personne obj) { /* ... */ }
    @Override public Personne find(String id) { /* ... */ }
    @Override public Personne update(Personne obj) { /* ... */ }
    @Override public void delete(Personne obj) { /* ... */ }
}
```



## Exemple de DAO

*La méthode* `PersonneDAO.create`

```
@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;
}
```



## Exemple de DAO

*La classe abstraite* `DAO`

```
@Override public Personne find(String id) {
    Personne p = new Personne();
    try {
        PreparedStatement prepare = connect.prepareStatement(
            "SELECT * FROM personnes WHERE nom = ?");
        prepare.setString(1, id);
        ResultSet result = prepare.executeQuery();
        if (result.first()) {
            p = new Personne(
                result.getString("nom"),
                result.getLong("age"));
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return p;
}
```



## Exemple de DAO

*Utilisation du* `DAO`

```
public class Main {
    public static void main(String[] args) {
        DAO<Personne> personneDao = new PersonneDAO();
        System.out.println(personneDao.find("Dupond"));
    }
}
```



## Exemple de DAO

*Une fabrique de* `DAO`

```
public class DAOFactory {
    public static DAO<Personne> getPersonneDAO(){
        return new PersonneDAO();
    }
}
```

## Exemple de DAO

*Utilisation du `DAO` avec la fabrique*

```
public class Main {
    public static void main(String[] args) {
        DAO<Personne> personneDao = DAOFactory.getPersonneDAO();
        System.out.println(personneDao.find("Dupond"));
    }
}
```



## Exemple de DAO

*Un `DAO` XML pour `Personne`*

```
public class PersonneXMLDAO extends DAO<Personne> {
    @Override public Personne create(Personne obj) { /* ... */ }
    @Override public Personne find(String id) { /* ... */ }
    @Override public Personne update(Personne obj) { /* ... */ }
    @Override public void delete(Personne obj) { /* ... */ }
}
```



## Exemple de DAO

*Une fabrique abstraite pour les `DAO`*

```
public abstract class AbstractDAOFactory {
    public enum DaoType { JDBC, XML; }

    public abstract DAO getPersonneDAO();

    public static AbstractDAOFactory getFactory(DaoType type){
        if(type == DaoType.JDBC) return new DAOFactory();
        if(type == DaoType.XML)) return new XMLDAOFactory();
        return null;
    }
}
```



## Exemple de DAO

*Les fabriques deviennent des sous-classes de `AbstractDAOFactory`*

```
public class DAOFactory extends AbstractDAOFactory {
    // ...
}

public class XMLDAOFactory extends AbstractDAOFactory {
    public static DAO<Personne> getPersonneDAO(){
        return new PersonneXMLDAO();
    }
}
```



## Exemple de DAO

*Usage avec la fabrique abstraite*

```
public class Main {
    public static void main(String[] args) {
        DAO<Personne> personneDao = AbstractDAOFactory.getFactory(DaoType.JDBC).getPersonneDAO();
        System.out.println(personneDao.find("Dupond"));
        personneDao = AbstractDAOFactory.getFactory(DaoType.XML).getPersonneDAO();
        System.out.println(personneDao.find("Dupond"));
    }
}
```

# 5.4 Mapping objet-relationnel

## 5.4.1 Introduction

### Mapping objet-relationnel

- Le *mapping objet-relationnel* (*Object-relational mapping* ou *ORM*) est une approche pour la persistance
- Elle vise à combler l'écart entre une BD relationnelle et un langage de programmation OO
  - fournit une « BD objet virtuelle » au niveau du langage de programmation
  - est établi en définissant des correspondances entre modèle objet et modèle relationnel
  - ces dernières sont définies à partir des règles de transformations de modèles
- Les outils existant (*ORM frameworks*) évitent de manipuler des requêtes SQL dans l'application
- Un objet marqué comme persistant est automatiquement stocké dans la BD (et mis à jour)

166

### Standard et produits

### Standard

- Java Persistence API (JPA)

*Quelques produits*

- EclipseLink
- Oracle TopLink
- Hibernate ORM
- DataNucleus

167

## 5.4.2 JPA

### Java Persistence API

- La spécification *Java Persistence API* (*JPA*) fournit un modèle de persistance pour les objets Java (POJO) basé l'ORM
- Permet de s'abstraire de l'implémentation de l'ORM et non plus simplement du SGBD
- Les mappings peuvent être spécifié en XML ou grâce aux annotations
- Fournit un langage d'interrogation *SQL-like* (*JPQL*)
- Se trouve dans le package `javax.persistence`
- Disponible pour *Java SE* et *Java EE*
- L'implémentation de référence est EclipseLink

168

### Historique de la persistance en Java

**2001** *Hibernate 1*
**2002** *JDO 1.0* JSR 12 (*Java Data Objects*)
**2003** *Hibernate 2*
**2006** *JPA 1.0* fait partie du JSR 220 (*EJB3*), *JDO 2.0* JSR 243 (*Java Data Objects 2.0*)
**2009** *JPA 2.0* JSR 317 (*Java Persistence 2.0*), *SDO* JSR 235 (*Service Data Objects*)
**2010** *JDO 3.0* (*maintenance release*)
**2013** *JPA 2.1* JSR 338 (*Java Persistence 2.1*)
**2015** *Hibernate 5*

169

### Technologies liées

**EJB** *Enterprise Java Beans* encapsule la couche domaine

- initialement, JPA était incluse dans EJB 3.0
- JPA ne nécessite pas de conteneur ⇒ rendue indépendante

**JDO** *Java Data Object* gère la persistance de façon transparente et non limitée à l'ORM

- JPA se focalise sur l'ORM
- initialement, JPA s'est inspiré sur JDO

**SDO** *Service Data Objects* est conçu pour SOA et complémentaire de JPA

- non limité à l'ORM
- supporte plusieurs langages de programmation

**Hibernate** est un ORM open source

- JPA s'est inspiré des premières versions
- Hibernate est une implémentation de JPA

170

### Principaux composants

**`Persistence`** fournit une méthode de classe pour la création d'un `EntityManagerFactory`

**`EntityManagerFactory`** initialise et donne accès aux autres fonctionnalités (ouverture de sessions)

**`EntityManager`** représente une unité de travail avec la BD

**`EntityTransaction`** contrôle la transaction

**`Query`** permet l'interrogation de la BD

171

### Unité de persistance

- Une *unité de persistance* (*persistence unit*) configure l'accès au SGBD pour un ensemble d'entités
- Elle est décrite dans le fichier XML `META-INF/persistence.xml`

172

### Unité de persistance

*Exemple de `persistence.xml`*

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/
     persistence_2_0.xsd"
    version="2.0">
    <persistence-unit name="todos" transaction-type="RESOURCE_LOCAL">
        <class>de.vogella.jpa.simple.model.Todo</class>
        <properties>
            <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver" />
            <property name="javax.persistence.jdbc.url" value="jdbc:derby:/home/vogella/databases/simpleDb;create=
    true" />
            <property name="javax.persistence.jdbc.user" value="test" />
            <property name="javax.persistence.jdbc.password" value="test" />

            <!-- EclipseLink should create the database schema automatically -->
            <property name="eclipselink.ddl-generation" value="create-tables" />
            <property name="eclipselink.ddl-generation.output-mode"value="database" />
        </properties>
    </persistence-unit>
</persistence>
```

173

### Unité de persistance

*Exemple d'utilisation*

```
public class Main {
    private static final String PERSISTENCE_UNIT_NAME = "todos";
    private static EntityManagerFactory factory;

    public static void main(String[] args) {
        factory = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT_NAME);
        // ...
    }
}
```

174

### Gestionnaire d'entité

— Le *gestionnaire d'entités* (*entity manager*) se charge des opérations sur la BD

— La classe `javax.persistence.EntityManager` expose les méthodes nécessaires pour cela

```
EntityManager em = factory.createEntityManager();
em.getTransaction().begin();
Todo todo = new Todo();
todo.setSummary("This is a test");
todo.setDescription("This is a test");
em.persist(todo);
em.getTransaction().commit();
em.close();
```

175

### Entité

— Une *entité* (*entity*) est une classe qui doit être persistante

— Elle doit être marquée avec l'annotation `javax.persistence.Entity`

— JPA va associer chaque entité avec une table de la BD

  — même nom que la classe sauf si `@Table(name="NEWTABLENAME")` est précisé

— Chaque instance de l'entité sera un tuple de la table

176

### Contraintes sur les entités

— Toutes les entités doivent

  — posséder une clé primaire,

  — avoir un constructeur par défaut (`public` ou `protected`),

  — ne pas être `final`

— Les attributs persistants ne doivent pas être `final`, ni `public`

177

### Persistance des attributs

— Par défaut, tous les attributs sont sauvegardés

  — même nom que l'attribut sauf si `@Column(name="newColumnName")` est précisé

— La clé primaire est marquée par `@Id`

  — les valeurs peuvent être générée (`@GeneratedValue`)

— L'annotation `@Transient` permet d'ignorer un attribut lors de la sauvegarde

178

### Persistance des attributs

*Exemple*

```
@Entity
public class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String summary;
    private String description;

    public String getSummary() { return summary; }
    public void setSummary(String summary) { this.summary = summary; }
    public String getDescription() { return description; }
    public void setDescription(String description) { this.description = description; }

    @Override
    public String toString() {
        return "Todo [summary=" + summary + ", description=" + description + "]";
    }
}
```

179

### Attribut multivalué et composé

— Un attribut composé d'un type qui n'est pas une entité doit être annoté par `@Embedded`

— La classe décrivant ce type est quant à elle annotée avec `@Embeddable`

— Un attribut multivalué d'un type de base ou embarqué doit être annoté par `@ElementCollection`

180

### Association

— Les associations peuvent être définies avec les annotations `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`

— Une association peut être unidirectionnelle ou bidirectionnelle

— Pour ce dernier cas, l'association est spécifiée dans les deux classes avec d'un côté l'attribut `mappedBy`

181

### Association

*Exemple 1/2*

```
@Entity
public class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String summary;
    private String description;
    private Task task;
}
```

182

### Association

*Exemple 2/2*

```
@Entity
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;

    @OneToMany(mappedBy = "task")
    private final List<Task> todos;
}
```

183

### Interrogation avec JPQL

— La méthode `EntityManager.createQuery` permet de créer une requête dynamiquement
— La méthode `EntityManager.createNamedQuery` permet d'instancier une requête nommée
— Une requête peut être paramétrée par nom (: *nom*) ou par position (?1)

184

### Interrogation avec JPQL

*Exemple de requête dynamique*

```
public List findWithName(String name) {
    return em.createQuery("SELECT c FROM Customer c WHERE c.name LIKE :custName")
        .setParameter("custName", name)
        .setMaxResults(10)
        .getResultList();
}
```

185

### Interrogation avec JPQL

*Exemple de requête nommée*

**Sur l'entité**

```
@NamedQuery(name="findAllCustomersWithName",
    query="SELECT c FROM Customer c WHERE c.name LIKE :custName")
```

**Lors de l'interrogation**

```
customers = em.createNamedQuery("findAllCustomersWithName")
    .setParameter("custName", "Smith")
    .getResultList();
```

186

## 5.5 Bibliographie

BAUER, Christian, Gavin KING et Gary GREGORY (2015). *Java Persistence with Hibernate*. English. 2nd edition. Manning. URL : https://www.manning.com/books/java-persistence-with-hibernate-second-edition.

## 5.6 Webographie

FORSTER, Peter (2005). *Object Persistence Design Guidelines*. English. Winterthur Insurance. URL : http://www.odbms.org/wp-content/uploads/2013/11/011.01-Forster-Object-Persistence-Design-Guidelines-August-2005.pdf.

PATERSON, James (2004). *Object Persistence*. English. URL : http://www.odbms.org/wp-content/uploads/2013/11/004.02-Paterson-Object-Persistence-December-2004.pdf.

SMITH, Chris (2004). *Java and Databases*. English. MindIQ. URL : http://www.odbms.org/2004/05/java-and-databases/.

SUTHERLAND, James et Doug CLARKE (2015). *Java Persistence*. English. WikiBooks. URL : https://en.wikibooks.org/wiki/Java_Persistence.

## 5.7 Exercices

CONTRAINTES

— Pour les exercices de ce chapitre, vous utiliserez l'énoncé de l'exercice 4.1 (annuaire des personnels) comme base.

— Le but de chaque exercice sera d'ajouter la couche de persistance.

— Afin de conserver la base commune aux exercices, vous utiliserez le patron de conception DATA ACCESS OBJECT (DAO).

**Exercice 5.1** (Sérialisation en Java)
Dans cet exercice, la persistance sera assurée par la sérialisation Java.

1. Rendez les classes de l'application sérialisables et vérifiez en le fonctionnement avec des tests unitaires.
2. Définissez l'interface DAO pour les opérations CRUD ainsi qu'une fabrique pour l'instanciation des DAO.
3. Implémentez cette interface avec la sérialisation Java.

**Exercice 5.2** (JDBC)
Dans cet exercice, la persistance sera assurée par l'API JDBC. Vous reprendrez le travail de l'exercice 5.1 et vous le compléterez. Vous pouvez utilisez le SGBD DERBY en mode embarqué.

1. Proposez une nouvelle implémentation des DAO en utilisant JDBC.
2. Mettez en œuvre une fabrique abstraite pour la création des DAO.

**Exercice 5.3** (ORM avec JPA)
Dans cet exercice, la persistance sera assurée par l'API JPA. Vous reprendrez le travail de l'exercice 5.2 et vous le compléterez. En plus de DERBY, vous utiliserez le *provider JPA* HIBERNATE ORM.

1. Proposez une nouvelle implémentation des DAO en utilisant JPA/Hibernate.
2. Intégrez cette implémentation dans la fabrique abstraite.