Catalogue de Design Pattern

Charels Hugo

B-INFO

2023-2024

Table des matières

1 Patterns de Création (Creational) 3
- 1.1 Abstract Factory . . . 3
  - 1.1.1 Description . . . 3
  - 1.1.2 Exemple . . . 4
- 1.2 Builder . . . 5
  - 1.2.1 Description . . . 5
  - 1.2.2 Exemple . . . 6
- 1.3 Factory Method . . . 8
  - 1.3.1 Description . . . 8
  - 1.3.2 Exemple . . . 9
- 1.4 Prototype . . . 10
  - 1.4.1 Description . . . 10
  - 1.4.2 Exemple . . . 11
- 1.5 Singleton . . . 11
  - 1.5.1 Description . . . 11
  - 1.5.2 Exemple . . . 12

2 Patterns de Structures (Structural) 13
- 2.1 Adapter . . . 13
  - 2.1.1 Description . . . 13
  - 2.1.2 Exemple . . . 13
- 2.2 Bridge . . . 14
  - 2.2.1 Description . . . 14
  - 2.2.2 Exemple . . . 15
- 2.3 Composite . . . 17
  - 2.3.1 Description . . . 17
  - 2.3.2 Exemple . . . 17
- 2.4 Decorator . . . 17
  - 2.4.1 Description . . . 17
  - 2.4.2 Motivation . . . 17
  - 2.4.3 Exemple . . . 17
- 2.5 Facade . . . 17
  - 2.5.1 Description . . . 17
  - 2.5.2 Motivation . . . 17
  - 2.5.3 Exemple . . . 17
- 2.6 Flyweight . . . 17
  - 2.6.1 Description . . . 17
  - 2.6.2 Motivation . . . 17
  - 2.6.3 Exemple . . . 17
- 2.7 Proxy . . . 17
  - 2.7.1 Description . . . 17
  - 2.7.2 Motivation . . . 17
  - 2.7.3 Exemple . . . 17

3 Patterns de Comportement (Behavioral) 17
- 3.1 Chain of Responsibility . . . 17
  - 3.1.1 Description . . . 17
  - 3.1.2 Motivation . . . 17
  - 3.1.3 Exemple . . . 17

3.2 Command . . . 17
3.2.1 Description . . . 17
3.2.2 Motivation . . . 17
3.2.3 Exemple . . . 17
3.3 Interpreter . . . 17
3.3.1 Description . . . 17
3.3.2 Motivation . . . 17
3.3.3 Exemple . . . 17
3.4 Iterator . . . 17
3.4.1 Description . . . 17
3.4.2 Motivation . . . 17
3.4.3 Exemple . . . 17
3.5 Mediator . . . 17
3.5.1 Description . . . 17
3.5.2 Motivation . . . 17
3.5.3 Exemple . . . 17
3.6 Memento . . . 17
3.6.1 Description . . . 17
3.6.2 Motivation . . . 17
3.6.3 Exemple . . . 17
3.7 Observer . . . 17
3.7.1 Description . . . 17
3.7.2 Motivation . . . 17
3.7.3 Exemple . . . 17
3.8 State . . . 17
3.8.1 Description . . . 17
3.8.2 Motivation . . . 17
3.8.3 Exemple . . . 17
3.9 Strategy . . . 17
3.9.1 Description . . . 17
3.9.2 Motivation . . . 17
3.9.3 Exemple . . . 17
3.10 Template Method . . . 17
3.10.1 Description . . . 17
3.10.2 Motivation . . . 17
3.10.3 Exemple . . . 17
3.11 Visitor . . . 17
3.11.1 Description . . . 17
3.11.2 Motivation . . . 17
3.11.3 Exemple . . . 17

Patterns de Création (Creational)

1.1 Abstract Factory

1.1.1 Description

Le design pattern Abstract Factory est un modèle de conception qui appartient à la catégorie des patrons de conception creational (de création). Son objectif principal est de fournir une interface pour créer des familles d'objets liés ou dépendants sans spécifier leurs classes concrètes. Cela signifie que le code client peut créer des objets sans avoir à connaître les détails spécifiques de leur implémentation.

Voici les principaux éléments qui composent le design pattern Abstract Factory :

1. **Abstract Factory (Fabrique Abstraite)** :
   - Il s'agit de l'interface définissant des méthodes pour créer chacun des types d'objets abstraits faisant partie de la famille d'objets.
   - Chaque méthode de l'interface correspond à la création d'un type d'objet abstrait.
2. **Concrete Factory (Fabrique Concrète)** :
   - Les implémentations concrètes de l'interface Abstract Factory.
   - Chaque Concrete Factory est responsable de la création de toute une famille d'objets concrets.
3. **Abstract Product (Produit Abstrait)** :
   - Interface ou classe abstraite définissant le comportement des objets de la famille.
   - Chaque produit de la famille possède ses propres méthodes, mais ces méthodes sont déclarées dans l'interface ou la classe abstraite commune à tous les produits.
4. **Concrete Product (Produit Concret)** :
   - Les implémentations concrètes des produits abstraits.
   - Chaque Concrete Product est une implémentation spécifique d'un produit de la famille.
5. **Client** :
   - Utilise l'interface de l'Abstract Factory pour créer des objets.
   - N'a pas besoin de connaître les détails spécifiques de la création des objets.
   - Travailler avec les objets via leurs interfaces abstraites.

Le processus de création d'objets avec le design pattern Abstract Factory se déroule comme suit :

1. Le client appelle les méthodes de création de l'Abstract Factory pour obtenir des objets.
2. L'Abstract Factory, en fonction de son type concret, crée les instances concrètes des produits de la famille.
3. Le client utilise les objets créés via les interfaces abstraites, ce qui lui permet de rester indépendant des classes concrètes spécifiques.

L'avantage principal de ce modèle est qu'il favorise la séparation des préoccupations en permettant aux clients de créer des familles d'objets apparentés sans avoir à connaître les détails de leur implémentation. Cela rend le code plus modulaire, plus flexible et plus facile à étendre. De plus, il favorise le principe de substitution de Liskov, car les objets peuvent être utilisés via leurs interfaces abstraites, permettant ainsi aux implémentations concrètes d'être interchangées facilement.

1.1.2 Exemple

Imaginons que nous construisons une application de rendu graphique qui doit fonctionner sur différentes plateformes, telles que Windows et Linux. Pour chaque plateforme, nous devons créer des boutons, des cases à cocher et des champs de texte avec un aspect spécifique à cette plateforme. Nous pouvons utiliser le Design Pattern Abstract Factory pour créer une fabrique abstraite qui sera implémentée par des fabriques concrètes pour chaque plateforme. Ainsi, notre code client peut créer des widgets sans se soucier de la plateforme sous-jacente.

```
// Abstract Factory
interface AbstractFactory {
    Button createButton();
    Checkbox createCheckbox();
}

// Concrete Factory for Windows
class WindowsFactory implements AbstractFactory {
    public Button createButton() {
        return new WindowsButton();
    }

    public Checkbox createCheckbox() {
        return new WindowsCheckbox();
    }
}

// Concrete Factory for Linux
class LinuxFactory implements AbstractFactory {
    public Button createButton() {
        return new LinuxButton();
    }

    public Checkbox createCheckbox() {
        return new LinuxCheckbox();
    }
}

// Abstract Product
interface Button {
    String render();
}

// Concrete Products for Windows and Linux
class WindowsButton implements Button {
    public String render() {
        return "Render Windows button";
    }
}

class LinuxButton implements Button {
    public String render() {
        return "Render Linux button";
    }
}

interface Checkbox {
```

```
48      String render();
49  }
50
51  class WindowsCheckbox implements Checkbox {
52      public String render() {
53          return "Render Windows checkbox";
54      }
55  }
56
57  class LinuxCheckbox implements Checkbox {
58      public String render() {
59          return "Render Linux checkbox";
60      }
61  }
62
63  // Client Code
64  public class Client {
65      public static void renderGui(AbstractFactory factory) {
66          Button button = factory.createButton();
67          Checkbox checkbox = factory.createCheckbox();
68          System.out.println(button.render());
69          System.out.println(checkbox.render());
70      }
71
72      public static void main(String[] args) {
73          AbstractFactory windowsFactory = new WindowsFactory();
74          AbstractFactory linuxFactory = new LinuxFactory();
75
76          renderGui(windowsFactory);
77          renderGui(linuxFactory);
78      }
79  }
```

Listing 1 – abstract_factory.java

1.2 Builder

1.2.1 Description

Le design pattern Builder est un modèle de conception appartenant à la catégorie des patrons de conception creational (de création). Son objectif principal est de séparer la construction d'un objet complexe de sa représentation, de sorte que le même processus de construction puisse créer différentes représentations.

Voici les principaux éléments qui composent le design pattern Builder :

1. **Builder (Constructeur)** :
   — Interface définissant les étapes de construction pour créer un objet complexe.
2. **Concrete Builder (Constructeur Concret)** :
   — Implémente l'interface Builder pour fournir des étapes de construction concrètes pour un type spécifique d'objet complexe.
   — Construit et assemble les parties de l'objet complexe selon les étapes spécifiées.
3. **Director (Directeur)** :
   — Dirige le processus de construction en utilisant un Builder pour construire un objet complexe.

— Ne connaît pas les détails de la construction, mais utilise l'interface Builder pour orchestrer le processus.

4. **Product (Produit)** :
   — Représente l'objet complexe en cours de construction.
   — Peut être de n'importe quel type ou structure, en fonction de la logique de construction.

Le processus de construction d'un objet complexe avec le design pattern Builder se déroule comme suit :

1. Le client crée un Builder concret et le passe au Directeur.
2. Le Directeur utilise le Builder pour construire l'objet complexe en suivant les étapes définies.
3. Une fois la construction terminée, le client récupère le produit du Builder.

L'avantage principal de ce modèle est sa flexibilité et sa capacité à créer différentes représentations d'un même objet complexe. Il permet également de simplifier le code client en séparant la logique de construction de la logique métier. Cela facilite également l'ajout de nouvelles étapes de construction ou la modification de la logique de construction sans affecter le client.

1.2.2 Exemple

Supposons que nous construisons une application pour assembler des ordinateurs. Les ordinateurs peuvent avoir différentes configurations avec des composants variés, tels que le processeur, la carte graphique, la mémoire, etc. Nous pouvons utiliser le Design Pattern Builder pour définir une interface de construction abstraite et créer des constructeurs concrets pour chaque type d'ordinateur (gamer, bureautique, etc.). Ainsi, nous pouvons construire différents types d'ordinateurs en utilisant le même processus de construction.

```java
// Product
class Computer {
    private String processor;
    private String graphicsCard;
    private int memory;

    public void setProcessor(String processor) {
        this.processor = processor;
    }

    public void setGraphicsCard(String graphicsCard) {
        this.graphicsCard = graphicsCard;
    }

    public void setMemory(int memory) {
        this.memory = memory;
    }

    @Override
    public String toString() {
        return "Computer: " + processor + ", " + graphicsCard + ", " + memory + "
            GB RAM";
    }
}

// Abstract Builder
interface ComputerBuilder {
    void buildProcessor();
```

```
    void buildGraphicsCard();
    void buildMemory();
    Computer getResult();
}

// Concrete Builders
class GamingComputerBuilder implements ComputerBuilder {
    private Computer computer = new Computer();

    public void buildProcessor() {
        computer.setProcessor("Intel i7");
    }

    public void buildGraphicsCard() {
        computer.setGraphicsCard("Nvidia RTX 3080");
    }

    public void buildMemory() {
        computer.setMemory(32);
    }

    public Computer getResult() {
        return computer;
    }
}

class OfficeComputerBuilder implements ComputerBuilder {
    private Computer computer = new Computer();

    public void buildProcessor() {
        computer.setProcessor("Intel i5");
    }

    public void buildGraphicsCard() {
        computer.setGraphicsCard("Intel UHD Graphics");
    }

    public void buildMemory() {
        computer.setMemory(16);
    }

    public Computer getResult() {
        return computer;
    }
}

// Director
class ComputerDirector {
    private ComputerBuilder computerBuilder;

    public ComputerDirector(ComputerBuilder computerBuilder) {
        this.computerBuilder = computerBuilder;
    }

    public void constructComputer() {
        computerBuilder.buildProcessor();
```

```
84          computerBuilder.buildGraphicsCard();
85          computerBuilder.buildMemory();
86      }
87
88      public Computer getComputer() {
89          return computerBuilder.getResult();
90      }
91  }
92
93  // Client Code
94  public class Client {
95      public static void main(String[] args) {
96          ComputerBuilder gamingComputerBuilder = new GamingComputerBuilder();
97          ComputerBuilder officeComputerBuilder = new OfficeComputerBuilder();
98
99          ComputerDirector director = new ComputerDirector(gamingComputerBuilder);
100         director.constructComputer();
101         Computer gamingComputer = director.getComputer();
102         System.out.println("Gaming Computer: " + gamingComputer);
103
104         director = new ComputerDirector(officeComputerBuilder);
105         director.constructComputer();
106         Computer officeComputer = director.getComputer();
107         System.out.println("Office Computer: " + officeComputer);
108     }
109 }
```

Listing 2 – builder.java

1.3 Factory Method

1.3.1 Description

Le design pattern Factory Method est un modèle de conception appartenant à la catégorie des patrons de conception creational (de création). Son objectif principal est de fournir une interface pour la création d'objets dans une classe, mais de permettre aux sous-classes de modifier le type d'objets qui seront instanciés.

Voici les principaux éléments qui composent le design pattern Factory Method :

1. **Product (Produit)** :
   - Interface ou classe abstraite définissant le type d'objets produits par le Factory Method.
2. **Concrete Product (Produit Concret)** :
   - Implémentation concrète de l'interface Product.
   - Chaque Concrete Product représente un type spécifique d'objet créé par le Factory Method.
3. **Creator (Créateur)** :
   - Classe abstraite qui définit la méthode factoryMethod().
   - Cette méthode est responsable de la création d'objets de type Product.
4. **Concrete Creator (Créateur Concret)** :
   - Implémentation concrète de la classe Creator.
   - Override la méthode factoryMethod() pour créer des instances spécifiques de Concrete Product.

Le processus de création d'objets avec le design pattern Factory Method se déroule comme suit :

1. Le client appelle la méthode factoryMethod() de la classe Creator pour obtenir une instance de Product.
2. La classe Creator, qui peut être une classe abstraite ou une classe concrète, crée et retourne une instance de Concrete Product en appelant la méthode factoryMethod().
3. Le client utilise ensuite l'objet Product obtenu via l'interface commune, sans avoir besoin de connaître la classe concrète réelle de l'objet.

L'avantage principal de ce modèle est qu'il permet de déléguer la responsabilité de la création d'objets à des sous-classes, ce qui permet une meilleure extensibilité et une réduction du couplage entre les classes. Il facilite également l'ajout de nouveaux types d'objets sans avoir à modifier le code existant.

1.3.2 Exemple

Supposons que nous développons un logiciel de traitement d'images avec différents types de filtres (filtre noir et blanc, filtre sepia, etc.). Nous pouvons utiliser le Design Pattern Factory Method en définissant une classe abstraite "Filter" avec une méthode abstraite "apply", qui sera implémentée par les sous-classes pour créer des filtres spécifiques.

```
// Product
interface Filter {
    void apply(String image);
}

// Concrete Products
class BlackAndWhiteFilter implements Filter {
    public void apply(String image) {
        System.out.println("Applying Black and White Filter to " + image);
    }
}

class SepiaFilter implements Filter {
    public void apply(String image) {
        System.out.println("Applying Sepia Filter to " + image);
    }
}

// Creator (Factory Method)
abstract class ImageProcessor {
    public void processImage(String image) {
        Filter filter = createFilter();
        filter.apply(image);
    }

    protected abstract Filter createFilter();
}

// Concrete Creators
class BlackAndWhiteImageProcessor extends ImageProcessor {
    protected Filter createFilter() {
        return new BlackAndWhiteFilter();
    }
}

class SepiaImageProcessor extends ImageProcessor {
    protected Filter createFilter() {
        return new SepiaFilter();
```

```
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        ImageProcessor processor = new BlackAndWhiteImageProcessor();
        processor.processImage("image1.jpg");

        processor = new SepiaImageProcessor();
        processor.processImage("image2.jpg");
    }
}
```

Listing 3 – factory_method.java

1.4 Prototype

1.4.1 Description

Le design pattern Prototype est un modèle de conception appartenant à la catégorie des patrons de conception creational (de création). Son objectif principal est de permettre la création d'objets en clonant une instance existante plutôt qu'en les instanciant à partir de zéro.

Voici les principaux éléments qui composent le design pattern Prototype :

1. **Prototype** :
   - Interface ou classe abstraite définissant la méthode clone().
   - Cette méthode est utilisée pour créer une copie profonde ou superficielle de l'objet, selon les besoins.
2. **Concrete Prototype (Prototype Concret)** :
   - Implémentation concrète de l'interface Prototype.
   - Définit la logique de clonage de l'objet.
3. **Client** :
   - Utilise le Prototype pour créer de nouveaux objets en les clonant.
   - Ne nécessite pas de connaître les détails de l'implémentation du clonage.

Le processus de création d'objets avec le design pattern Prototype se déroule comme suit :

1. Le client demande la création d'un nouvel objet en utilisant un objet Prototype existant.
2. Le Prototype, qui peut être une classe abstraite ou une classe concrète, utilise sa méthode clone() pour créer une copie de lui-même.
3. Le client utilise ensuite l'objet cloné selon ses besoins.

L'avantage principal de ce modèle est qu'il permet de créer de nouveaux objets avec un minimum d'effort, en évitant le processus de création coûteux. Il permet également de réduire la duplication de code et d'offrir une meilleure flexibilité en permettant la création de nouveaux types d'objets en utilisant des prototypes existants.

1.4.2 Exemple

Supposons que nous développons une application de dessin où les utilisateurs peuvent créer des formes géométriques. Pour créer une nouvelle forme, nous pouvons utiliser le Design Pattern Prototype en définissant une interface "Shape" avec une méthode "clone" qui sera implémentée par les sous-classes pour copier l'objet existant.

```java
// Prototype
interface Shape extends Cloneable {
    void draw();
    Shape clone();
}

// Concrete Prototypes
class Circle implements Shape {
    public void draw() {
        System.out.println("Drawing Circle");
    }

    public Shape clone() {
        return new Circle();
    }
}

class Rectangle implements Shape {
    public void draw() {
        System.out.println("Drawing Rectangle");
    }

    public Shape clone() {
        return new Rectangle();
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        Shape circle = new Circle();
        Shape clonedCircle = circle.clone();
        clonedCircle.draw();

        Shape rectangle = new Rectangle();
        Shape clonedRectangle = rectangle.clone();
        clonedRectangle.draw();
    }
}
```

Listing 4 – prototype.java

1.5 Singleton

1.5.1 Description

Le design pattern Singleton est un modèle de conception appartenant à la catégorie des patrons de conception creational (de création). Son objectif principal est de garantir qu'une classe n'a qu'une seule instance et de fournir un point d'accès global à cette instance.

Voici les principaux éléments qui composent le design pattern Singleton :

1. **Singleton** :
   — Classe avec une méthode statique qui retourne toujours la même instance de cette classe.
   — Le constructeur de la classe est généralement rendu privé pour empêcher l'instanciation directe de la classe en dehors de la classe elle-même.

Le processus d'utilisation du design pattern Singleton est assez simple :

1. Les clients accèdent à l'instance unique de la classe Singleton en appelant la méthode statique de la classe.
2. Si l'instance n'existe pas encore, elle est créée et stockée dans un champ statique privé de la classe.
3. L'instance unique est ensuite retournée à chaque appel de la méthode statique.

L'avantage principal de ce modèle est qu'il garantit qu'une classe n'a qu'une seule instance dans l'ensemble du programme, ce qui peut être utile pour des ressources partagées telles que des bases de données ou des fichiers de configuration. Cela évite également le gaspillage de ressources en évitant la création répétée d'instances et offre un point d'accès global pour accéder à cette instance unique.

Cependant, l'utilisation abusive du Singleton peut conduire à des problèmes de test unitaire et à des dépendances cachées, il convient donc de l'utiliser avec discernement.

1.5.2 Exemple

Supposons que nous développons une application qui a besoin d'une classe "Configuration" pour stocker les paramètres de configuration de l'application. Nous pouvons utiliser le Design Pattern Singleton pour s'assurer qu'il n'y a qu'une seule instance de la classe "Configuration" qui est partagée par l'ensemble de l'application.

```
// Singleton
class Configuration {
    private static Configuration instance;

    // Empêcher l'instanciation directe depuis l'extérieur de la classe
    private Configuration() { }

    public static Configuration getInstance() {
        if (instance == null) {
            instance = new Configuration();
        }
        return instance;
    }

    // Autres méthodes et attributs
}

// Client Code
public class Client {
    public static void main(String[] args) {
        Configuration config1 = Configuration.getInstance();
        Configuration config2 = Configuration.getInstance();

        System.out.println(config1 == config2); // true, car il n'y a qu'une seule
            instance
    }
```

```
26 }
```

Listing 5 – singleton.java

Patterns de Structures (Structural)

2.1 Adapter

2.1.1 Description

Le design pattern Adapter est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de permettre à des interfaces incompatibles de travailler ensemble en convertissant l'interface d'une classe en une autre interface attendue par le client.

Voici les principaux éléments qui composent le design pattern Adapter :

1. **Target (Cible)** :
   - Interface que le client utilise pour interagir avec le système.
2. **Adapter (Adaptateur)** :
   - Classe qui adapte l'interface d'une classe existante (Adaptee) à l'interface Target attendue par le client.
   - Implémente l'interface Target et contient une instance de la classe Adaptee.
3. **Adaptee (Adapté)** :
   - Classe existante dont l'interface n'est pas compatible avec l'interface Target.
   - La classe que l'Adapter va adapter pour qu'elle puisse être utilisée par le client.

Le processus d'utilisation du design pattern Adapter se déroule comme suit :

1. Le client utilise l'interface Target pour interagir avec le système.
2. L'Adapter reçoit les appels de l'interface Target et les convertit en appels appropriés à l'interface de l'Adaptee.
3. L'Adaptee exécute les opérations demandées et retourne les résultats à l'Adapter.
4. L'Adapter convertit ensuite les résultats de l'Adaptee en un format compatible avec l'interface Target et les renvoie au client.

L'avantage principal de ce modèle est qu'il permet d'intégrer des classes existantes dans de nouveaux systèmes sans avoir à modifier leur code source. Cela favorise la réutilisabilité du code et permet d'ajouter de nouvelles fonctionnalités sans affecter les composants existants. Cependant, l'utilisation abusive de ce modèle peut entraîner une complexité accrue du code en raison de l'ajout de plusieurs couches d'adaptation.

2.1.2 Exemple

Supposons que nous avons une classe "LegacyPrinter" qui utilise une ancienne interface pour l'impression de documents. Nous développons une nouvelle classe "ModernPrinter" qui utilise une interface différente pour l'impression. Pour que notre code client puisse utiliser les deux types d'imprimantes de manière interchangeable, nous pouvons utiliser le Design Pattern Adapter pour créer un adaptateur qui convertit l'interface de "ModernPrinter" en celle de "LegacyPrinter".

```
1 // Adaptee (LegacyPrinter)
2 class LegacyPrinter {
3     public void print(String document) {
```

```
        System.out.println("Printing document: " + document);
    }
}

// Adapter
class ModernPrinterAdapter extends LegacyPrinter {
    private ModernPrinter modernPrinter;

    public ModernPrinterAdapter(ModernPrinter modernPrinter) {
        this.modernPrinter = modernPrinter;
    }

    @Override
    public void print(String document) {
        modernPrinter.print(document);
    }
}

// New Printer (Using a different interface)
class ModernPrinter {
    public void print(String document) {
        System.out.println("Printing modern document: " + document);
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        // Using Legacy Printer with the Adapter
        LegacyPrinter legacyPrinter = new LegacyPrinter();
        ModernPrinterAdapter adapter = new ModernPrinterAdapter(new ModernPrinter
            ());

        legacyPrinter.print("Legacy Document"); // Using Legacy Printer directly
        adapter.print("Modern Document"); // Using Modern Printer with the Adapter
    }
}
```

Listing 6 – adapter.java

2.2 Bridge

2.2.1 Description

Le design pattern Bridge est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de séparer l'abstraction d'une classe de son implémentation, permettant ainsi à ces deux parties de varier indépendamment.

Voici les principaux éléments qui composent le design pattern Bridge :

1. **Abstraction** :
   - Interface qui définit les méthodes abstraites utilisées par le client.
   - Contient une référence à un objet Implementor.
2. **Refined Abstraction (Abstraction Affinée)** :
   - Implémentation spécifique de l'interface Abstraction.

— Peut ajouter des fonctionnalités supplémentaires.

3. **Implementor (Implémenteur)** :
   — Interface qui définit les méthodes abstraites utilisées par Abstraction.
4. **Concrete Implementor (Implémenteur Concret)** :
   — Implémentation concrète de l'interface Implementor.
   — Contient la logique détaillée de l'implémentation.

Le processus d'utilisation du design pattern Bridge se déroule comme suit :

1. Le client utilise l'interface Abstraction pour interagir avec le système.
2. L'Abstraction délègue une partie de son implémentation à l'objet Implementor référencé.
3. Les différentes implémentations de Implementor peuvent être échangées dynamiquement sans affecter Abstraction.

L'avantage principal de ce modèle est qu'il permet de séparer complètement l'abstraction de son implémentation, ce qui facilite l'évolution et la maintenance du code. Il permet également de réduire le couplage entre les classes en les reliant par des interfaces plutôt que par des implémentations concrètes. Cependant, cela peut introduire une complexité supplémentaire dans le code en raison de l'ajout de plusieurs couches d'abstraction et d'implémentation.

2.2.2 Exemple

Supposons que nous développons un système de formes géométriques avec différents types de dessin (par exemple, dessin vectoriel et dessin en raster). Au lieu de créer une classe pour chaque combinaison de forme et de dessin, nous pouvons utiliser le Design Pattern Bridge pour diviser la hiérarchie en deux parties : l'abstraction (Forme) et l'implémentation (Dessin). Ainsi, nous pouvons créer des ponts (bridges) entre les formes et les dessins pour obtenir différentes combinaisons de formes et de dessins.

```
1  // Implementor Interface
2  interface DrawingAPI {
3      void drawCircle(double x, double y, double radius);
4  }
5
6  // Concrete Implementations of DrawingAPI
7  class DrawingVector implements DrawingAPI {
8      @Override
9      public void drawCircle(double x, double y, double radius) {
10         System.out.println("Drawing Circle in Vector at (" + x + "," + y + ") with
               radius " + radius);
11     }
12 }
13
14 class DrawingRaster implements DrawingAPI {
15     @Override
16     public void drawCircle(double x, double y, double radius) {
17         System.out.println("Drawing Circle in Raster at (" + x + "," + y + ") with
               radius " + radius);
18     }
19 }
20
21 // Abstraction
22 abstract class Shape {
23     protected DrawingAPI drawingAPI;
```

```
24
25      protected Shape(DrawingAPI drawingAPI) {
26          this.drawingAPI = drawingAPI;
27      }
28
29      public abstract void draw();
30  }
31
32  // Refined Abstractions for specific shapes
33  class CircleShape extends Shape {
34      private double x, y, radius;
35
36      public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI) {
37          super(drawingAPI);
38          this.x = x;
39          this.y = y;
40          this.radius = radius;
41      }
42
43      public void draw() {
44          drawingAPI.drawCircle(x, y, radius);
45      }
46  }
47
48  // Client Code
49  public class Client {
50      public static void main(String[] args) {
51          DrawingAPI vectorDrawingAPI = new DrawingVector();
52          DrawingAPI rasterDrawingAPI = new DrawingRaster();
53
54          Shape circle = new CircleShape(1, 2, 3, vectorDrawingAPI);
55          circle.draw();
56
57          circle = new CircleShape(5, 7, 10, rasterDrawingAPI);
58          circle.draw();
59      }
60  }
```

Listing 7 – adapter.java

2.3 Composite

2.3.1 Description

2.3.2 Exemple

2.4 Decorator

2.4.1 Description

2.4.2 Motivation

2.4.3 Exemple

2.5 Facade

2.5.1 Description

2.5.2 Motivation

2.5.3 Exemple

2.6 Flyweight

2.6.1 Description

2.6.2 Motivation

2.6.3 Exemple

2.7 Proxy

2.7.1 Description

2.7.2 Motivation

2.7.3 Exemple

Patterns de Comportement (Behavioral)

3.1 Chain of Responsibility

3.1.1 Description

3.1.2 Motivation

3.1.3 Exemple

3.2 Command

3.2.1 Description

3.2.2 Motivation

3.2.3 Exemple

3.3 Interpreter

3.3.1 Description

3.3.2 Motivation

3.3.3 Exemple

3.4 Iterator

3.4.1 Description

3.4.2 Motivation

3.4.3 Exemple

3.5 Mediator