Université Libre de Bruxelles

Le Catalogue des Design Pattern

Étudiants :
Hugo Charels

2023-2024

Table des matières

1 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.

```
1  // Abstract Factory
2  interface AbstractFactory {
3      Button createButton();
4      Checkbox createCheckbox();
5  }
6
7  // Concrete Factory for Windows
8  class WindowsFactory implements AbstractFactory {
9      public Button createButton() {
10         return new WindowsButton();
11     }
12
13     public Checkbox createCheckbox() {
14         return new WindowsCheckbox();
15     }
16 }
17
18 // Concrete Factory for Linux
19 class LinuxFactory implements AbstractFactory {
20     public Button createButton() {
21         return new LinuxButton();
22     }
23
24     public Checkbox createCheckbox() {
25         return new LinuxCheckbox();
26     }
27 }
28
29 // Abstract Product
30 interface Button {
31     String render();
32 }
33
```

```java
// 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 {
    String render();
}

class WindowsCheckbox implements Checkbox {
    public String render() {
        return "Render Windows checkbox";
    }
}

class LinuxCheckbox implements Checkbox {
    public String render() {
        return "Render Linux checkbox";
    }
}

// Client Code
public class Client {
    public static void renderGui(AbstractFactory factory) {
        Button button = factory.createButton();
        Checkbox checkbox = factory.createCheckbox();
        System.out.println(button.render());
        System.out.println(checkbox.render());
    }

    public static void main(String[] args) {
        AbstractFactory windowsFactory = new WindowsFactory();
        AbstractFactory linuxFactory = new LinuxFactory();

        renderGui(windowsFactory);
        renderGui(linuxFactory);
    }
}
```

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() {
```

```
50          return computer;
51      }
52  }
53
54  class OfficeComputerBuilder implements ComputerBuilder {
55      private Computer computer = new Computer();
56
57      public void buildProcessor() {
58          computer.setProcessor("Intel i5");
59      }
60
61      public void buildGraphicsCard() {
62          computer.setGraphicsCard("Intel UHD Graphics");
63      }
64
65      public void buildMemory() {
66          computer.setMemory(16);
67      }
68
69      public Computer getResult() {
70          return computer;
71      }
72  }
73
74  // Director
75  class ComputerDirector {
76      private ComputerBuilder computerBuilder;
77
78      public ComputerDirector(ComputerBuilder computerBuilder) {
79          this.computerBuilder = computerBuilder;
80      }
81
82      public void constructComputer() {
83          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.

```java
// 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();
```

```
49            processor.processImage("image2.jpg");
50        }
51    }
```

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.

```
1 // Prototype
2 interface Shape extends Cloneable {
3     void draw();
4     Shape clone();
5 }
6
7 // Concrete Prototypes
```

```java
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.

```
1  // Singleton
2  class Configuration {
3      private static Configuration instance;
4
5      // Empêcher l'instanciation directe depuis l'extérieur de la
           classe
6      private Configuration() { }
7
8      public static Configuration getInstance() {
9          if (instance == null) {
10             instance = new Configuration();
11         }
```

```
12          return instance;
13      }
14
15      // Autres méthodes et attributs
16  }
17
18  // Client Code
19  public class Client {
20      public static void main(String[] args) {
21          Configuration config1 = Configuration.getInstance();
22          Configuration config2 = Configuration.getInstance();
23
24          System.out.println(config1 == config2); // true, car il n'y
                a qu'une seule instance
25      }
26  }
```

Listing 5 – singleton.java

2 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. **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.
3. **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.

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 // Target Interface
2 interface Printer {
3     void print(String document);
```

```
}

// Adaptee (LegacyPrinter)
class LegacyPrinter {
    public void printDocument(String document) {
        System.out.println("Printing document: " + document);
    }
}

// Adapter
class LegacyPrinterAdapter implements Printer {
    private LegacyPrinter legacyPrinter;

    public LegacyPrinterAdapter(LegacyPrinter legacyPrinter) {
        this.legacyPrinter = legacyPrinter;
    }

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

// New Printer (Using a different interface)
class ModernPrinter implements Printer {
    @Override
    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();
        Printer legacyPrinterAdapter = new LegacyPrinterAdapter(
            legacyPrinter);

        legacyPrinter.printDocument("Legacy Document"); // Using
            Legacy Printer directly
        legacyPrinterAdapter.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.

```java
// Abstraction
abstract class Shape {
    protected DrawingAPI drawingAPI;

    protected Shape(DrawingAPI drawingAPI) {
        this.drawingAPI = drawingAPI;
    }

    public abstract void draw();
}

// Refined Abstractions for specific shapes
class CircleShape extends Shape {
    private double x, y, radius;

    public CircleShape(double x, double y, double radius,
       DrawingAPI drawingAPI) {
        super(drawingAPI);
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public void draw() {
        drawingAPI.drawCircle(x, y, radius);
    }
}

// Implementor Interface
interface DrawingAPI {
    void drawCircle(double x, double y, double radius);
}

// Concrete Implementations of DrawingAPI
class DrawingVector implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.println("Drawing Circle in Vector at (" + x + ",
           " + y + ") with radius " + radius);
    }
}

class DrawingRaster implements DrawingAPI {
    @Override
    public void drawCircle(double x, double y, double radius) {
        System.out.println("Drawing Circle in Raster at (" + x + ",
           " + y + ") with radius " + radius);
    }
}

```

```
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 – bridge.java

2.3 Composite

2.3.1 Description

Le design pattern Composite est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de permettre la composition d'objets en structures arborescentes pour représenter les hiérarchies partie-tout.

Voici les principaux éléments qui composent le design pattern Composite :

1. **Component (Composant)** :
   - Interface ou classe abstraite commune à tous les composants de la structure.
   - Définit les méthodes communes pour manipuler les composants.
2. **Leaf (Feuille)** :
   - Implémentation concrète de la classe Component.
   - Représente les éléments individuels de la structure qui n'ont pas de sous-composants.
3. **Composite** :
   - Implémentation concrète de la classe Component.
   - Représente les éléments de la structure qui ont des sous-composants.
   - Contient une liste de références vers ses sous-composants.

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

1. Les clients manipulent les composants de la structure via l'interface commune Component.
2. Les opérations sont propagées récursivement dans la structure, avec chaque composant (feuille ou composite) déléguant les appels à ses sous-composants le cas échéant.
3. Cela permet de traiter uniformément les composants individuels et les structures complexes de manière transparente.

L'avantage principal de ce modèle est qu'il permet de manipuler des structures arborescentes de manière uniforme, en traitant les composants individuels et les structures composites de la même manière. Cela simplifie le code client en permettant de traiter une structure complexe comme une unité unique, tout en offrant une grande flexibilité dans la manipulation des composants. Cependant, cela peut rendre certaines opérations plus complexes en raison de la nécessité de gérer la récursion.

2.3.2 Exemple

Supposons que nous développons une application de modélisation de formes graphiques. Nous avons des formes simples (cercle, carré) et des groupes de formes qui peuvent contenir d'autres formes, y compris d'autres groupes. Avec le Design Pattern Composite, nous pouvons traiter les formes individuelles et les groupes de formes de manière homogène, ce qui facilite les opérations de dessin, de déplacement, etc.

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 // Component Interface
5 interface Shape {
```

```
6       void draw();
7   }
8
9   // Leaf (Individual Shape)
10  class Circle implements Shape {
11      @Override
12      public void draw() {
13          System.out.println("Drawing Circle");
14      }
15  }
16
17  class Square implements Shape {
18      @Override
19      public void draw() {
20          System.out.println("Drawing Square");
21      }
22  }
23
24  // Composite (Group of Shapes)
25  class CompositeShape implements Shape {
26      private List<Shape> shapes = new ArrayList<>();
27
28      public void addShape(Shape shape) {
29          shapes.add(shape);
30      }
31
32      public void removeShape(Shape shape) {
33          shapes.remove(shape);
34      }
35
36      @Override
37      public void draw() {
38          for (Shape shape : shapes) {
39              shape.draw();
40          }
41      }
42  }
43
44  // Client Code
45  public class Client {
46      public static void main(String[] args) {
47          Shape circle = new Circle();
48          Shape square = new Square();
49
50          CompositeShape compositeShape1 = new CompositeShape();
51          compositeShape1.addShape(circle);
52          compositeShape1.addShape(square);
53
54          Shape circle2 = new Circle();
55          CompositeShape compositeShape2 = new CompositeShape();
```

```
56            compositeShape2.addShape(circle2);
57            compositeShape2.addShape(compositeShape1);
58
59            compositeShape2.draw();
60        }
61    }
```

Listing 8 – composite.java

2.4 Decorator

2.4.1 Description

Le design pattern Decorator est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est d'ajouter dynamiquement de nouvelles fonctionnalités à un objet en les enveloppant dans des objets décorateurs plutôt que de les étendre par héritage.

Voici les principaux éléments qui composent le design pattern Decorator :

1. **Component (Composant)** :
   - Interface ou classe abstraite commune à tous les composants à décorer.
   - Définit les méthodes de base que les décorateurs et les composants concrets implémentent.
2. **Concrete Component (Composant Concret)** :
   - Implémentation concrète de la classe Component.
   - Représente l'objet de base auquel des fonctionnalités supplémentaires peuvent être ajoutées.
3. **Decorator (Décorateur)** :
   - Classe abstraite qui étend Component et enveloppe les composants concrets.
   - Contient une référence à un objet de type Component.
4. **Concrete Decorator (Décorateur Concret)** :
   - Implémentation concrète de la classe Decorator.
   - Ajoute des fonctionnalités supplémentaires à l'objet de base en le décorant.

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

1. Les clients manipulent les composants à travers l'interface Component.
2. Les fonctionnalités supplémentaires sont ajoutées dynamiquement en enveloppant le composant de base dans des décorateurs appropriés.
3. Les décorateurs peuvent être empilés pour ajouter plusieurs fonctionnalités en cascade.
4. Chaque décorateur transmet les appels aux méthodes de base du composant au composant sous-jacent, permettant ainsi une chaîne de responsabilité.

L'avantage principal de ce modèle est qu'il permet d'ajouter des fonctionnalités supplémentaires à un objet de manière flexible et dynamique, sans avoir à modifier sa structure de classe. Cela favorise la réutilisabilité du code en permettant la combinaison libre de fonctionnalités à la volée. Cependant, cela peut rendre la lecture du code plus complexe en raison de la présence de plusieurs couches de décorateurs.

2.4.2 Exemple

Supposons que nous développons une application de café et nous avons différentes boissons (expresso, café au lait) avec des options supplémentaires (lait, chocolat, sucre). Avec le Design Pattern Decorator, nous pouvons ajouter les options supplémentaires de manière dynamique à chaque boisson sans avoir besoin de créer de nombreuses classes pour chaque combinaison possible.

```java
// Component Interface
interface Coffee {
    double getCost();
    String getDescription();
}

// Concrete Component (Base Coffee)
class BaseCoffee implements Coffee {
    @Override
    public double getCost() {
        return 3.0;
    }

    @Override
    public String getDescription() {
        return "Base Coffee";
    }
}

// Decorator (Decorator Base Class)
abstract class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee decoratedCoffee) {
        this.decoratedCoffee = decoratedCoffee;
    }

    @Override
    public double getCost() {
        return decoratedCoffee.getCost();
    }

    @Override
    public String getDescription() {
        return decoratedCoffee.getDescription();
    }
}

// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 1.0;
    }

    @Override
```

```java
    public String getDescription() {
        return super.getDescription() + ", Milk";
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee decoratedCoffee) {
        super(decoratedCoffee);
    }

    @Override
    public double getCost() {
        return super.getCost() + 0.5;
    }

    @Override
    public String getDescription() {
        return super.getDescription() + ", Sugar";
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        Coffee baseCoffee = new BaseCoffee();
        Coffee coffeeWithMilk = new MilkDecorator(baseCoffee);
        Coffee coffeeWithMilkAndSugar = new SugarDecorator(
            coffeeWithMilk);

        System.out.println("Description: " + coffeeWithMilkAndSugar
            .getDescription());
        System.out.println("Cost: " + coffeeWithMilkAndSugar.
            getCost());
    }
}
```

Listing 9 – decorator.java

2.5 Facade

2.5.1 Description

Le design pattern Facade est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de fournir une interface unifiée à un ensemble d'interfaces d'un sous-système afin de simplifier son utilisation et de masquer sa complexité.

Voici les principaux éléments qui composent le design pattern Facade :

1. **Facade (Facade)** :
   - Classe qui fournit une interface unifiée à un ensemble d'interfaces plus complexes dans un sous-système.
   - Cache les détails de l'implémentation et simplifie l'interaction avec le sous-système.
2. **Subsystem (Sous-système)** :
   - Ensemble de classes et d'interfaces qui implémentent les fonctionnalités du système.
   - Ces classes ne sont pas directement accessibles par les clients mais sont utilisées par la facade pour réaliser les fonctionnalités demandées.

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

1. Les clients interagissent avec la Facade pour accéder aux fonctionnalités du sous-système.
2. La Facade traduit ces demandes en appels appropriés aux classes et interfaces du sous-système.
3. La Facade simplifie ainsi l'utilisation du sous-système en fournissant une interface plus conviviale et en masquant sa complexité interne.

L'avantage principal de ce modèle est qu'il permet de simplifier l'utilisation d'un sous-système complexe en fournissant une interface unifiée et conviviale. Cela permet aux clients de ne pas avoir à connaître les détails de l'implémentation du sous-système, ce qui favorise la modularité et la maintenance du code. Cependant, cela peut également limiter la flexibilité en cachant les détails d'implémentation, et il est important de concevoir soigneusement les interfaces de la Facade pour répondre aux besoins des clients.

2.5.2 Exemple

Supposons que nous développons une application pour gérer une voiture avec plusieurs sous-systèmes tels que le moteur, les freins, l'électronique, etc. Au lieu d'interagir directement avec chaque sous-système, nous pouvons créer une Facade Car pour regrouper les fonctionnalités de chaque sous-système et fournir une interface unique pour interagir avec la voiture.

```
1 // Facade
2 class Car {
3     private Engine engine;
4     private Brakes brakes;
5     private Electronics electronics;
6
```

```
7       public Car() {
8           engine = new Engine();
9           brakes = new Brakes();
10          electronics = new Electronics();
11      }
12
13      public void startCar() {
14          engine.start();
15          electronics.activate();
16      }
17
18      public void stopCar() {
19          electronics.deactivate();
20          engine.stop();
21      }
22
23      public void applyBrakes() {
24          brakes.apply();
25      }
26
27      public void releaseBrakes() {
28          brakes.release();
29      }
30  }
31
32  // Subsystem
33  class Engine {
34      public void start() {
35          System.out.println("Engine started");
36      }
37
38      public void stop() {
39          System.out.println("Engine stopped");
40      }
41  }
42
43  class Brakes {
44      public void apply() {
45          System.out.println("Brakes applied");
46      }
47
48      public void release() {
49          System.out.println("Brakes released");
50      }
51  }
52
53  class Electronics {
54      public void activate() {
55          System.out.println("Electronics activated");
56      }
```

```
57
58      public void deactivate() {
59          System.out.println("Electronics deactivated");
60      }
61  }
62
63
64  // Client Code
65  public class Client {
66      public static void main(String[] args) {
67          Car car = new Car();
68          car.startCar();
69
70          // Drive the car...
71
72          car.applyBrakes();
73
74          // Stop the car...
75
76          car.stopCar();
77      }
78  }
```

Listing 10 – facade.java

2.6 Flyweight

2.6.1 Description

Le design pattern Flyweight est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de minimiser l'utilisation de la mémoire ou du calcul en partageant autant que possible les données similaires entre plusieurs objets.

Voici les principaux éléments qui composent le design pattern Flyweight :

1. **Flyweight (Poids Léger)** :
   - Interface ou classe abstraite commune à tous les objets légers.
   - Contient les méthodes partagées entre les objets légers.
2. **ConcreteFlyweight (Poids Léger Concret)** :
   - Implémentation concrète de l'interface Flyweight.
   - Représente un objet léger partagé qui stocke l'état intrinsèque (partagé) et ne dépend pas du contexte externe.
3. **UnsharedConcreteFlyweight (Poids Léger Non Partagé)** :
   - Implémentation concrète de l'interface Flyweight.
   - Représente un objet léger qui ne peut pas être partagé et stocke l'état extrinsèque (non partagé).
4. **FlyweightFactory (Fabrique de Poids Légers)** :
   - Gère et fournit les objets légers existants.
   - Assure le partage des objets légers lors de leur création.

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

1. Les clients demandent des objets légers à la FlyweightFactory.
2. Si l'objet léger existe déjà dans la FlyweightFactory, il est renvoyé au client.
3. Sinon, un nouvel objet léger est créé et stocké dans la FlyweightFactory pour une utilisation ultérieure.
4. Les clients manipulent les objets légers via l'interface Flyweight.

L'avantage principal de ce modèle est qu'il permet de réduire la consommation de mémoire en partageant les objets similaires entre plusieurs instances. Cela peut être particulièrement utile lorsque de nombreux objets doivent être créés, mais leur état est souvent répétitif. Cependant, cela peut également rendre le code plus complexe en raison de la nécessité de gérer les états partagés et non partagés.

2.6.2 Exemple

Supposons que nous développons un éditeur de texte où chaque caractère est représenté par un objet Character. Plutôt que de créer un nouvel objet Character pour chaque caractère du texte, nous pouvons utiliser le Design Pattern Flyweight pour stocker les caractères déjà créés dans un cache et les réutiliser lorsque le même caractère est demandé à nouveau.

```java
import java.util.HashMap;
import java.util.Map;

// Flyweight Interface
interface CharacterFlyweight {
    void draw();
}

// Concrete Flyweight
class CharacterFlyweightImpl implements CharacterFlyweight {
    private char symbol;

    public CharacterFlyweightImpl(char symbol) {
        this.symbol = symbol;
    }

    @Override
    public void draw() {
        System.out.println("Drawing character: " + symbol);
    }
}

// Unshared Concrete Flyweight
class UnsharedConcreteFlyweight implements CharacterFlyweight {
    private String data;

    public UnsharedConcreteFlyweight(String data) {
        this.data = data;
    }

    @Override
    public void draw() {
        System.out.println("Drawing unshared character data: " +
            data);
    }
}

// Flyweight Factory
class CharacterFlyweightFactory {
    private Map<Character, CharacterFlyweight> characterCache = new
        HashMap<>();

    public CharacterFlyweight getCharacter(char symbol) {
        CharacterFlyweight character = characterCache.get(symbol);

        if (character == null) {
            character = new CharacterFlyweightImpl(symbol);
            characterCache.put(symbol, character);
        }

```

```
49          return character;
50      }
51  }
52
53  // Client Code
54  public class Client {
55      public static void main(String[] args) {
56          CharacterFlyweightFactory characterFactory = new
               CharacterFlyweightFactory();
57
58          // Drawing characters in a text
59          char[] text = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r',
                'l', 'd'};
60
61          for (char c : text) {
62              CharacterFlyweight character = characterFactory.
                   getCharacter(c);
63              character.draw();
64          }
65      }
66  }
```

Listing 11 – flyweight.java

2.7 Proxy

2.7.1 Description

Le design pattern Proxy est un modèle de conception appartenant à la catégorie des patrons de conception structurels. Son objectif principal est de fournir un substitut ou un espace réservé à un autre objet afin de contrôler l'accès à celui-ci ou de fournir des fonctionnalités supplémentaires.

Voici les principaux éléments qui composent le design pattern Proxy :

1. **Subject (Sujet)** :
   - Interface ou classe abstraite commune à l'objet réel et à son proxy.
   - Définit les méthodes communes que le proxy et l'objet réel implémentent.
2. **RealSubject (Sujet Réel)** :
   - Implémentation concrète de l'interface Subject.
   - Représente l'objet réel dont l'accès est contrôlé par le proxy.
3. **Proxy** :
   - Implémentation concrète de l'interface Subject.
   - Contrôle l'accès à l'objet réel et fournit des fonctionnalités supplémentaires si nécessaire.
   - Peut retarder la création et l'initialisation de l'objet réel jusqu'à ce qu'il soit vraiment nécessaire (lazy initialization).

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

1. Les clients interagissent avec le proxy à travers l'interface Subject.
2. Si l'objet réel n'est pas encore créé, le proxy peut le créer et l'initialiser de manière transparente.
3. Le proxy transmet ensuite les appels à l'objet réel ou effectue des traitements supplémentaires avant ou après la transmission de l'appel.

L'avantage principal de ce modèle est qu'il permet de contrôler l'accès à l'objet réel et de fournir des fonctionnalités supplémentaires sans modifier son code. Cela peut être utile pour ajouter des fonctionnalités telles que la mise en cache, la journalisation, la sécurité ou la gestion des ressources. Cependant, cela peut également introduire une complexité supplémentaire dans le code en raison de la présence de plusieurs couches de proxy.

2.7.2 Exemple

Supposons que nous développons une application pour télécharger des fichiers depuis Internet. Le téléchargement des fichiers peut être coûteux en temps, nous pouvons donc utiliser un Proxy pour vérifier les autorisations de l'utilisateur avant de permettre le téléchargement et pour mettre en cache les fichiers téléchargés pour une utilisation ultérieure.

```
1 // Subject Interface
2 interface FileDownloader {
3     void download(String fileUrl);
4 }
5
```

```java
// Real Subject
class RealFileDownloader implements FileDownloader {
    @Override
    public void download(String fileUrl) {
        System.out.println("Downloading file from: " + fileUrl);
    }
}

// Proxy
class FileDownloaderProxy implements FileDownloader {
    private boolean isAdmin;

    public FileDownloaderProxy(boolean isAdmin) {
        this.isAdmin = isAdmin;
    }

    @Override
    public void download(String fileUrl) {
        if (isAdmin) {
            RealFileDownloader realFileDownloader = new
               RealFileDownloader();
            realFileDownloader.download(fileUrl);
        } else {
            System.out.println("Access denied. Only admins can
               download files.");
        }
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        FileDownloaderProxy fileDownloader = new
           FileDownloaderProxy(true);
        fileDownloader.download("https://example.com/sample.pdf");

        FileDownloaderProxy restrictedDownloader = new
           FileDownloaderProxy(false);
        restrictedDownloader.download("https://example.com/secret.
           pdf");
    }
}
```

Listing 12 – proxy.java

3 Patterns de Comportement (Behavioral)

3.1 Chain of Responsibility

3.1.1 Description

Le design pattern Chain of Responsibility est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de permettre le passage d'une requête le long d'une chaîne de traitements, où chaque maillon de la chaîne peut traiter la requête ou la transmettre au maillon suivant.

Voici les principaux éléments qui composent le design pattern Chain of Responsibility :

1. **Handler (Gestionnaire)** :
   - Interface ou classe abstraite commune à tous les gestionnaires.
   - Définit une méthode pour traiter les requêtes et une référence au prochain gestionnaire dans la chaîne.
2. **ConcreteHandler (Gestionnaire Concret)** :
   - Implémentation concrète de l'interface Handler.
   - Traite la requête si possible, sinon la transmet au prochain gestionnaire dans la chaîne.

Le processus d'utilisation du design pattern Chain of Responsibility se déroule comme suit :

1. Les clients envoient des requêtes à un gestionnaire initial dans la chaîne.
2. Chaque gestionnaire décide s'il peut traiter la requête ou s'il doit la transmettre au gestionnaire suivant.
3. La requête est transmise le long de la chaîne jusqu'à ce qu'elle soit traitée ou que la fin de la chaîne soit atteinte.

L'avantage principal de ce modèle est qu'il permet de découpler l'émetteur d'une requête de ses destinataires en permettant à plusieurs objets de tenter de traiter la requête sans connaître explicitement les autres. Cela favorise la flexibilité et la réutilisabilité du code en permettant de modifier dynamiquement la chaîne de responsabilité ou d'ajouter de nouveaux gestionnaires sans modifier le code client. Cependant, cela peut également rendre la gestion des requêtes plus complexe si la chaîne devient trop longue ou si les gestionnaires ne sont pas correctement configurés.

3.1.2 Exemple

Supposons que nous développons une application de gestion des demandes de congés dans une entreprise. Nous pouvons utiliser le Design Pattern Chain of Responsibility pour créer une chaîne de responsabilité où chaque gestionnaire (par exemple, RH, manager, directeur) peut approuver ou rejeter la demande de congé. Si un gestionnaire ne peut pas traiter la demande, il la transmet au gestionnaire supérieur dans la chaîne.

```
1 // Handler Interface
2 interface LeaveRequestHandler {
3     void handleRequest(LeaveRequest request);
4 }
```

```
5
6  // Concrete Handlers
7  class HRHandler implements LeaveRequestHandler {
8      private LeaveRequestHandler nextHandler;
9
10     public HRHandler(LeaveRequestHandler nextHandler) {
11         this.nextHandler = nextHandler;
12     }
13
14     @Override
15     public void handleRequest(LeaveRequest request) {
16         if (request.getDays() <= 5) {
17             System.out.println("HRHandler: Leave request approved
                   for " + request.getDays() + " days.");
18         } else if (nextHandler != null) {
19             nextHandler.handleRequest(request);
20         }
21     }
22 }
23
24 class ManagerHandler implements LeaveRequestHandler {
25     private LeaveRequestHandler nextHandler;
26
27     public ManagerHandler(LeaveRequestHandler nextHandler) {
28         this.nextHandler = nextHandler;
29     }
30
31     @Override
32     public void handleRequest(LeaveRequest request) {
33         if (request.getDays() > 5 && request.getDays() <= 10) {
34             System.out.println("ManagerHandler: Leave request
                   approved for " + request.getDays() + " days.");
35         } else if (nextHandler != null) {
36             nextHandler.handleRequest(request);
37         }
38     }
39 }
40
41 class DirectorHandler implements LeaveRequestHandler {
42     @Override
43     public void handleRequest(LeaveRequest request) {
44         if (request.getDays() > 10) {
45             System.out.println("DirectorHandler: Leave request
                   approved for " + request.getDays() + " days.");
46         } else {
47             System.out.println("DirectorHandler: Leave request
                   rejected.");
48         }
49     }
50 }
```

```java

// Leave Request Class
class LeaveRequest {
    private int days;

    public LeaveRequest(int days) {
        this.days = days;
    }

    public int getDays() {
        return days;
    }
}

// Client Code
public class Client {
    public static void main(String[] args) {
        LeaveRequestHandler director = new DirectorHandler();
        LeaveRequestHandler manager = new ManagerHandler(director);
        LeaveRequestHandler hr = new HRHandler(manager);

        // Leave Requests
        LeaveRequest request1 = new LeaveRequest(3);
        LeaveRequest request2 = new LeaveRequest(7);
        LeaveRequest request3 = new LeaveRequest(12);

        // Handling Leave Requests
        hr.handleRequest(request1);
        hr.handleRequest(request2);
        hr.handleRequest(request3);
    }
}
```

Listing 13 – chain_of_responsibility.java

3.2 Command

3.2.1 Description

Le design pattern Command est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est d'encapsuler une requête en tant qu'objet, ce qui permet de paramétrer des clients avec différentes requêtes, de mettre en file d'attente les requêtes, de les enregistrer et d'annuler les opérations.

Voici les principaux éléments qui composent le design pattern Command :

1. **Command (Commande)** :
   - Interface ou classe abstraite commune à tous les commandes.
   - Définit une méthode pour exécuter la commande.
2. **ConcreteCommand (Commande Concrite)** :
   - Implémentation concrète de l'interface Command.
   - Contient une référence à l'objet receveur (celui qui effectue l'action) et implémente la méthode pour exécuter la commande en appelant une ou plusieurs méthodes du receveur.
3. **Invoker (Invocateur)** :
   - Demande à la commande d'exécuter une action.
   - Ne connaît pas les détails de l'implémentation de la commande.
4. **Receiver (Receveur)** :
   - Connaît la manière d'effectuer l'action associée à la commande.
   - Implémente les méthodes que les commandes appellent pour effectuer les opérations.

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

1. Un objet de commande est créé et associé à un receveur.
2. L'objet de commande est passé à l'invocateur.
3. L'invocateur demande à l'objet de commande d'exécuter une action.
4. L'objet de commande appelle la méthode appropriée sur le receveur pour effectuer l'action.

L'avantage principal de ce modèle est qu'il permet de déconnecter l'objet qui invoque l'opération de celui qui la traite, ce qui permet de créer des systèmes flexibles et extensibles. Il permet également de mettre en file d'attente, d'enregistrer et d'annuler des opérations facilement. Cependant, cela peut rendre le code plus complexe en introduisant de nombreux objets de commande et en nécessitant une gestion appropriée de leur cycle de vie.

3.2.2 Exemple

Supposons que nous développons une application de traitement de texte où nous souhaitons permettre aux utilisateurs d'effectuer des opérations telles que copier, coller et annuler. Nous pouvons utiliser le Design Pattern Command pour créer des classes de commandes (par exemple, CopyCommand, PasteCommand, UndoCommand) qui encapsulent chaque opération et les exécuter au besoin.

```java
// Command Interface
interface Command {
    void execute();
}

// Concrete Commands
class LightOnCommand implements Command {
    private Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOn();
    }
}

class LightOffCommand implements Command {
    private Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.turnOff();
    }
}

// Invoker
class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}

// Receiver
class Light {
    private String location;

    public Light(String location) {
```

```
51          this.location = location;
52      }
53
54      public void turnOn() {
55          System.out.println(location + " light is ON");
56      }
57
58      public void turnOff() {
59          System.out.println(location + " light is OFF");
60      }
61  }
62
63  // Client Code
64  public class Client {
65      public static void main(String[] args) {
66          // Receiver
67          Light livingRoomLight = new Light("Living Room");
68
69          // Concrete Commands
70          Command lightOnCommand = new LightOnCommand(livingRoomLight
                );
71          Command lightOffCommand = new LightOffCommand(
                livingRoomLight);
72
73          // Invoker
74          RemoteControl remoteControl = new RemoteControl();
75
76          // Pressing buttons
77          remoteControl.setCommand(lightOnCommand);
78          remoteControl.pressButton();
79
80          remoteControl.setCommand(lightOffCommand);
81          remoteControl.pressButton();
82      }
83  }
```

Listing 14 – command.java

3.3 Interpreter

3.3.1 Description

Le design pattern Interpreter est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de définir une grammaire pour un langage et de fournir un moyen d'interpréter et d'exécuter ce langage.

Voici les principaux éléments qui composent le design pattern Interpreter :

1. **AbstractExpression (Expression Abstraite)** :
   - Interface ou classe abstraite commune à toutes les expressions.
   - Définit une méthode pour interpréter une expression donnée.
2. **TerminalExpression (Expression Terminale)** :
   - Implémentation concrète de l'interface AbstractExpression.
   - Représente une expression de base qui ne peut pas être décomposée en d'autres expressions.
3. **NonterminalExpression (Expression Non Terminale)** :
   - Implémentation concrète de l'interface AbstractExpression.
   - Représente une expression composée de sous-expressions.
4. **Context (Contexte)** :
   - Contient des informations globales qui sont partagées entre les expressions pendant l'interprétation.
5. **Client** :
   - Construit et configure l'arbre d'expression.
   - Évalue les expressions en appelant la méthode d'interprétation sur la racine de l'arbre.

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

1. Les clients construisent un arbre d'expression à partir d'expressions terminales et non terminales.
2. Les clients évaluent l'expression en appelant la méthode d'interprétation sur la racine de l'arbre.
3. Chaque nœud de l'arbre d'expression interprète et évalue les sous-expressions, transmettant le contexte si nécessaire.

L'avantage principal de ce modèle est qu'il permet de définir une grammaire pour un langage et d'interpréter les expressions de manière flexible. Cela peut être utile pour implémenter des langages de programmation, des systèmes de requêtes ou d'autres systèmes basés sur la logique. Cependant, cela peut rendre le code complexe en raison de la nécessité de définir de nombreuses classes d'expressions et de gérer la construction de l'arbre d'expression.

3.3.2 Exemple

Supposons que nous développons une application pour évaluer des expressions arithmétiques simples, telles que "2 + 3 * 4". Nous pouvons utiliser le Design Pattern Interpreter pour créer une grammaire et un interpréteur qui évalue ces expressions.

```
// Abstract Expression
interface Expression {
    int interpret();
}

// Terminal Expression
class NumberExpression implements Expression {
    private int number;

    public NumberExpression(int number) {
        this.number = number;
    }

    @Override
    public int interpret() {
        return number;
    }
}

// Non-terminal Expressions
class AddExpression implements Expression {
    private Expression left;
    private Expression right;

    public AddExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public int interpret() {
        return left.interpret() + right.interpret();
    }
}

class MultiplyExpression implements Expression {
    private Expression left;
    private Expression right;

    public MultiplyExpression(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }

    @Override
```

```
46      public int interpret() {
47          return left.interpret() * right.interpret();
48      }
49  }
50
51  // Context
52  class Context {
53      private Expression expression;
54
55      public Context(Expression expression) {
56          this.expression = expression;
57      }
58
59      public int evaluate() {
60          return expression.interpret();
61      }
62  }
63
64  // Client
65  public class Client {
66      public static void main(String[] args) {
67          // Expression: 2 + 3 * 4
68          Expression expression = new AddExpression(
69              new NumberExpression(2),
70              new MultiplyExpression(
71                  new NumberExpression(3),
72                  new NumberExpression(4)
73              )
74          );
75
76          Context context = new Context(expression);
77          int result = context.evaluate();
78          System.out.println("Result: " + result); // Output: Result:
                14
79      }
80  }
```

Listing 15 – interpreter.java

3.4 Iterator

3.4.1 Description

Le design pattern Iterator est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de fournir un moyen d'accéder séquentiellement aux éléments d'une collection sans exposer sa représentation interne.

Voici les principaux éléments qui composent le design pattern Iterator :

1. **Iterator (Itérateur)** :
   - Interface ou classe abstraite commune à tous les itérateurs.
   - Définit des méthodes pour parcourir la collection, obtenir l'élément suivant et vérifier s'il reste des éléments.
2. **ConcreteIterator (Itérateur Concret)** :
   - Implémentation concrète de l'interface Iterator.
   - Maintient une référence à la position actuelle dans la collection et implémente les méthodes pour parcourir la collection.
3. **Aggregate (Agrégat)** :
   - Interface ou classe abstraite commune à toutes les collections.
   - Définit une méthode pour créer un itérateur.
4. **ConcreteAggregate (Agrégat Concret)** :
   - Implémentation concrète de l'interface Aggregate.
   - Fournit une méthode pour créer un itérateur qui parcourt la collection spécifique.

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

1. Les clients obtiennent un itérateur à partir de la collection en appelant la méthode de création d'itérateur de l'agrégat.
2. Les clients utilisent l'itérateur pour parcourir séquentiellement les éléments de la collection en utilisant les méthodes définies dans l'interface Iterator.
3. L'itérateur maintient la position actuelle dans la collection et permet aux clients d'accéder à chaque élément individuellement sans avoir à connaître les détails de la collection sous-jacente.

L'avantage principal de ce modèle est qu'il permet de parcourir les éléments d'une collection de manière flexible et indépendante de sa représentation interne. Cela favorise la réutilisabilité du code en permettant d'utiliser les mêmes itérateurs avec différentes collections, tout en préservant l'encapsulation des collections. Cependant, cela peut rendre le code plus complexe en raison de la nécessité de définir des classes d'itérateurs pour chaque type de collection.

3.4.2 Exemple

Supposons que nous développons une application pour gérer une liste de tâches. Nous pouvons utiliser le Design Pattern Iterator pour permettre aux clients de parcourir les tâches dans la liste sans avoir à connaître la structure sous-jacente de la liste.

```java
// Iterator Interface
interface Iterator<T> {
    boolean hasNext();
    T next();
}

// Concrete Iterator
class TaskListIterator implements Iterator<String> {
    private TaskList taskList;
    private int currentIndex = 0;

    public TaskListIterator(TaskList taskList) {
        this.taskList = taskList;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < taskList.size();
    }

    @Override
    public String next() {
        if (hasNext()) {
            String task = taskList.get(currentIndex);
            currentIndex++;
            return task;
        }
        return null;
    }
}

// Aggregate Interface
interface TaskList {
    Iterator<String> createIterator();
}

// Concrete Aggregate
class TodoList implements TaskList {
    private List<String> tasks = new ArrayList<>();

    public void addTask(String task) {
        tasks.add(task);
    }

    public void removeTask(String task) {
        tasks.remove(task);
    }

    public String get(int index) {
        return tasks.get(index);
```

```
51      }
52
53      public int size() {
54          return tasks.size();
55      }
56
57      @Override
58      public Iterator<String> createIterator() {
59          return new TaskListIterator(this);
60      }
61  }
62
63  // Client Code
64  public class Client {
65      public static void main(String[] args) {
66          TodoList todoList = new TodoList();
67          todoList.addTask("Task 1");
68          todoList.addTask("Task 2");
69          todoList.addTask("Task 3");
70
71          Iterator<String> iterator = todoList.createIterator();
72          while (iterator.hasNext()) {
73              String task = iterator.next();
74              System.out.println("Task: " + task);
75          }
76      }
77  }
```

Listing 16 – iterator.java

3.5 Mediator

3.5.1 Description

Le design pattern Mediator est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de définir un objet qui encapsule la manière dont un ensemble d'objets interagissent, en favorisant la déconnexion entre ces objets.

Voici les principaux éléments qui composent le design pattern Mediator :

1. **Mediator (Médiateur)** :
   - Interface ou classe abstraite commune à tous les médiateurs.
   - Définit des méthodes pour permettre la communication entre les objets du système.
2. **ConcreteMediator (Médiateur Concret)** :
   - Implémentation concrète de l'interface Mediator.
   - Gère la communication entre les objets en implémentant les méthodes définies dans l'interface Mediator.
3. **Colleague (Collègue)** :
   - Classe abstraite ou interface commune à tous les collègues.
   - Définit des méthodes pour interagir avec d'autres collègues via le médiateur.
4. **ConcreteColleague (Collègue Concret)** :
   - Implémentation concrète de l'interface Colleague.
   - Communique avec d'autres collègues via le médiateur, en utilisant les méthodes définies dans l'interface Colleague.

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

1. Les collègues communiquent entre eux en passant par le médiateur.
2. Lorsqu'un collègue a besoin de communiquer avec un autre collègue, il appelle une méthode sur le médiateur.
3. Le médiateur reçoit l'appel et transmet l'information au collègue concerné.
4. Le médiateur peut effectuer des traitements supplémentaires avant de transmettre l'information au collègue destinataire.

L'avantage principal de ce modèle est qu'il permet de déconnecter étroitement les objets du système en évitant les dépendances directes entre eux. Cela favorise la modularité et la maintenabilité du code en réduisant le couplage. Cependant, cela peut rendre le médiateur complexe s'il doit gérer de nombreuses interactions entre les collègues, et il est important de concevoir soigneusement les interfaces du médiateur et des collègues pour faciliter la communication.

3.5.2 Exemple

Supposons que nous développons un système de chat où plusieurs utilisateurs peuvent communiquer entre eux. Nous pouvons utiliser le Design Pattern Mediator pour créer un médiateur qui gère les communications entre les utilisateurs, de sorte qu'ils n'aient pas besoin de se connaître mutuellement.

```java
// Mediator Interface
interface ChatMediator {
    void sendMessage(User user, String message);
}

// Concrete Mediator
class ChatRoom implements ChatMediator {
    @Override
    public void sendMessage(User user, String message) {
        System.out.println(user.getName() + " sends message: " +
            message);
    }
}

// Colleague Interface
abstract class User {
    protected ChatMediator mediator;
    protected String name;

    public User(ChatMediator mediator, String name) {
        this.mediator = mediator;
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public abstract void send(String message);
    public abstract void receive(String message);
}

// Concrete Colleague
class ChatUser extends User {
    public ChatUser(ChatMediator mediator, String name) {
        super(mediator, name);
    }

    @Override
    public void send(String message) {
        System.out.println(name + " sends: " + message);
        mediator.sendMessage(this, message);
    }

    @Override
    public void receive(String message) {
        System.out.println(name + " receives: " + message);
    }
}

```

```java
// Client Code
public class Client {
    public static void main(String[] args) {
        ChatMediator chatMediator = new ChatRoom();

        User user1 = new ChatUser(chatMediator, "User1");
        User user2 = new ChatUser(chatMediator, "User2");
        User user3 = new ChatUser(chatMediator, "User3");

        chatMediator.sendMessage(user1, "Hello, everyone!");

        user2.send("Hi, User1!");
    }
}
```

Listing 17 – mediator.java

3.6 Memento

3.6.1 Description

Le design pattern Memento est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de capturer et d'externaliser l'état interne d'un objet sans violer l'encapsulation, de manière à pouvoir le restaurer ultérieurement dans son état précédent.

Voici les principaux éléments qui composent le design pattern Memento :

1. **Memento (Mémento)** :
   - Interface ou classe abstraite commune à tous les mementos.
   - Définit des méthodes pour accéder à l'état sauvegardé.
2. **ConcreteMemento (Mémento Concret)** :
   - Implémentation concrète de l'interface Memento.
   - Stocke l'état interne de l'objet d'origine à un moment donné.
3. **Originator (Créateur)** :
   - Classe dont l'état interne doit être sauvegardé.
   - Crée un memento contenant une copie de son état interne et peut restaurer son état à partir d'un memento donné.
4. **Caretaker (Gardien)** :
   - Classe responsable de la gestion des mementos.
   - Stocke les mementos dans une liste ou une structure de données appropriée et les fournit à l'originateur pour la restauration.

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

1. L'originateur crée un memento pour sauvegarder son état interne à un moment donné.
2. L'originateur peut utiliser ce memento pour restaurer son état interne à un moment ultérieur.
3. Le gardien peut stocker plusieurs mementos pour permettre la restauration de l'état à différents points dans le temps.

L'avantage principal de ce modèle est qu'il permet de restaurer l'état d'un objet à un moment antérieur sans violer son encapsulation. Cela peut être utile pour implémenter des fonctionnalités telles que l'annulation et la restauration d'actions dans une application. Cependant, cela peut également augmenter la consommation de mémoire si de nombreux mementos doivent être stockés, et il est important de gérer correctement le cycle de vie des mementos pour éviter les fuites de mémoire.

3.6.2 Exemple

Supposons que nous développons un éditeur de texte où les utilisateurs peuvent écrire et modifier du texte. Nous pouvons utiliser le Design Pattern Memento pour créer des mementos qui sauvegardent l'état du texte à un moment donné, afin de pouvoir restaurer cet état ultérieurement.

```java
// Memento Interface
interface Memento {
    String getText();
}

// Concrete Memento
class TextMemento implements Memento {
    private String text;

    public TextEditorMemento(String text) {
        this.text = text;
    }

    @Override
    public String getText() {
        return text;
    }
}

// Originator
class TextEditor {
    private String text;

    public void write(String text) {
        this.text = text;
    }

    public TextMemento save() {
        return new TextMemento(text);
    }

    public void restore(TextMemento memento) {
        text = memento.getText();
    }

    public void display() {
        System.out.println("Current Text: " + text);
    }
}

// Caretaker
class TextHistory {
    private Stack<TextMemento> history = new Stack<>();

    public void push(TextMemento memento) {
        history.push(memento);
    }

    public TextMemento pop() {
        return history.pop();
```

```
51      }
52  }
53
54  // Client Code
55  public class Client {
56      public static void main(String[] args) {
57          TextEditor textEditor = new TextEditor();
58          TextHistory history = new TextHistory();
59
60          textEditor.write("Hello, world!");
61          textEditor.display();
62
63          // Save state
64          history.push(textEditor.save());
65
66          textEditor.write("Hello, Design Pattern Memento!");
67          textEditor.display();
68
69          // Restore state
70          textEditor.restore(history.pop());
71          textEditor.display();
72      }
73  }
```

Listing 18 – memento.java

3.7 Observer

3.7.1 Description

Le design pattern Observer est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de définir une dépendance de type un-à-plusieurs entre objets, de manière à ce que lorsqu'un objet change d'état, tous ses dépendants soient notifiés et mis à jour automatiquement.

Voici les principaux éléments qui composent le design pattern Observer :

1. **Subject (Sujet)** :
   - Interface ou classe abstraite commune à tous les sujets observables.
   - Définit des méthodes pour ajouter, supprimer et notifier des observateurs.
2. **ConcreteSubject (Sujet Concret)** :
   - Implémentation concrète de l'interface Subject.
   - Maintient l'état interne et notifie les observateurs lorsque cet état change.
3. **Observer (Observateur)** :
   - Interface ou classe abstraite commune à tous les observateurs.
   - Définit une méthode de mise à jour appelée par le sujet lorsqu'un changement d'état se produit.
4. **ConcreteObserver (Observateur Concret)** :
   - Implémentation concrète de l'interface Observer.
   - Enregistre son intérêt pour les notifications auprès du sujet et réagit aux mises à jour de celui-ci.

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

1. Les observateurs s'enregistrent auprès du sujet pour recevoir des notifications.
2. Lorsque l'état du sujet change, il notifie tous ses observateurs en appelant leur méthode de mise à jour.
3. Les observateurs réagissent aux notifications en mettant à jour leur état ou en effectuant d'autres actions en conséquence.

L'avantage principal de ce modèle est qu'il permet de maintenir la cohérence entre les objets en évitant les dépendances directes et en favorisant la séparation des préoccupations. Cela favorise la modularité et la réutilisabilité du code en permettant de connecter et de déconnecter facilement les observateurs du sujet. Cependant, cela peut rendre le code plus complexe en raison de la multiplicité des interactions entre les sujets et les observateurs.

3.7.2 Exemple

Supposons que nous développons une application météo où les utilisateurs peuvent s'abonner pour recevoir des mises à jour en temps réel sur la météo. Nous pouvons utiliser le Design Pattern Observer pour créer des observateurs (abonnés) qui sont notifiés chaque fois que les données météorologiques changent.

```java
import java.util.ArrayList;
import java.util.List;

// Subject Interface
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Concrete Subject
class WeatherStation implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String weatherData;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(weatherData);
        }
    }

    public void setWeatherData(String weatherData) {
        this.weatherData = weatherData;
        notifyObservers();
    }
}

// Observer Interface
interface Observer {
    void update(String weatherData);
}

// Concrete Observer
class User implements Observer {
    private String name;

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

```
51
52      @Override
53      public void update(String weatherData) {
54          System.out.println(name + " received weather update: " +
                weatherData);
55      }
56  }
57
58  // Client Code
59  public class Client {
60      public static void main(String[] args) {
61          WeatherStation weatherStation = new WeatherStation();
62
63          User user1 = new User("John");
64          User user2 = new User("Alice");
65
66          weatherStation.registerObserver(user1);
67          weatherStation.registerObserver(user2);
68
69          weatherStation.setWeatherData("Sunny"); // Output: John
                received weather update: Sunny, Alice received weather
                update: Sunny
70      }
71  }
```

Listing 19 – observer.java

3.8 State

3.8.1 Description

Le design pattern State est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de permettre à un objet de modifier son comportement lorsqu'il change son état interne, de manière à ce que sa classe apparaisse modifiée.

Voici les principaux éléments qui composent le design pattern State :

1. **State (État)** :
   - Interface ou classe abstraite commune à tous les états possibles de l'objet contexte.
   - Définit les méthodes que l'objet contexte peut appeler pour modifier son comportement.
2. **ConcreteState (État Concret)** :
   - Implémentation concrète de l'interface State.
   - Représente un état spécifique de l'objet contexte et implémente les méthodes définies dans l'interface State pour modifier son comportement en fonction de cet état.
3. **Context (Contexte)** :
   - Classe qui possède un état interne.
   - Utilise l'interface State pour déléguer les requêtes associées à un certain état à l'objet ConcreteState approprié.

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

1. Le contexte délègue les requêtes associées à un certain état à l'objet ConcreteState approprié.
2. L'objet ConcreteState modifie le comportement du contexte en réponse à la requête.
3. Lorsque le contexte change d'état, il change également l'objet ConcreteState associé en conséquence.

L'avantage principal de ce modèle est qu'il permet de définir un comportement spécifique pour chaque état d'un objet sans avoir recours à de longues séries d'instructions conditionnelles. Cela favorise la modularité et la maintenabilité du code en séparant les responsabilités liées à chaque état dans des classes distinctes. Cependant, cela peut augmenter la complexité du code en introduisant de nombreuses classes d'états et en nécessitant une gestion appropriée de la transition entre les états.

3.8.2 Exemple

Supposons que nous développons un lecteur de musique qui peut être dans trois états : Lecture, Pause et Arrêt. Nous pouvons utiliser le Design Pattern State pour modéliser ces états et gérer les transitions entre eux.

```
1 // State Interface
2 interface State {
3     void play();
4     void pause();
```

```
5       void stop();
6   }
7
8   // Concrete States
9   class PlayingState implements State {
10      @Override
11      public void play() {
12          System.out.println("Already playing.");
13      }
14
15      @Override
16      public void pause() {
17          System.out.println("Paused the music.");
18      }
19
20      @Override
21      public void stop() {
22          System.out.println("Stopped the music.");
23      }
24  }
25
26  class PausedState implements State {
27      @Override
28      public void play() {
29          System.out.println("Resumed playing.");
30      }
31
32      @Override
33      public void pause() {
34          System.out.println("Already paused.");
35      }
36
37      @Override
38      public void stop() {
39          System.out.println("Stopped the music.");
40      }
41  }
42
43  class StoppedState implements State {
44      @Override
45      public void play() {
46          System.out.println("Started playing.");
47      }
48
49      @Override
50      public void pause() {
51          System.out.println("Music is stopped. Cannot pause.");
52      }
53
54      @Override
```

```
55      public void stop() {
56          System.out.println("Already stopped.");
57      }
58  }
59
60  // Context (Contexte)
61  class MusicPlayer {
62      private State currentState;
63
64      public MusicPlayer() {
65          currentState = new StoppedState();
66      }
67
68      public void setState(State state) {
69          currentState = state;
70      }
71
72      public void play() {
73          currentState.play();
74          setState(new PlayingState());
75      }
76
77      public void pause() {
78          currentState.pause();
79          setState(new PausedState());
80      }
81
82      public void stop() {
83          currentState.stop();
84          setState(new StoppedState());
85      }
86  }
87
88  // Client Code
89  public class Client {
90      public static void main(String[] args) {
91          MusicPlayer musicPlayer = new MusicPlayer();
92
93          musicPlayer.play(); // Output: Started playing.
94          musicPlayer.pause(); // Output: Paused the music.
95          musicPlayer.pause(); // Output: Already paused.
96          musicPlayer.stop(); // Output: Stopped the music.
97          musicPlayer.stop(); // Output: Already stopped.
98          musicPlayer.play(); // Output: Started playing.
99      }
100 }
```

Listing 20 – state.java

3.9 Strategy

3.9.1 Description

Le design pattern Strategy est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Ainsi, un client peut choisir dynamiquement l'algorithme approprié sans modifier la classe cliente.

Voici les principaux éléments qui composent le design pattern Strategy :

1. **Strategy (Stratégie)** :
   - Interface ou classe abstraite commune à toutes les stratégies.
   - Définit une méthode ou un ensemble de méthodes utilisées par le contexte pour exécuter l'algorithme.
2. **ConcreteStrategy (Stratégie Concrète)** :
   - Implémentation concrète de l'interface Strategy.
   - Contient l'algorithme spécifique à exécuter.
3. **Context (Contexte)** :
   - Classe qui utilise une stratégie pour exécuter un algorithme.
   - Peut modifier la stratégie utilisée à tout moment pendant l'exécution.

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

1. Le contexte encapsule une référence à une stratégie.
2. Lorsqu'une opération est requise, le contexte appelle la méthode de la stratégie pour exécuter l'algorithme.
3. Le client peut modifier la stratégie du contexte à tout moment en remplaçant la stratégie par une autre stratégie compatible.

L'avantage principal de ce modèle est qu'il permet de définir une famille d'algorithmes, encapsuler chacun d'eux et les rendre interchangeables. Cela favorise la flexibilité et la réutilisabilité du code en permettant de choisir dynamiquement l'algorithme approprié à exécuter. Cependant, cela peut augmenter la complexité du code en introduisant de nombreuses classes de stratégies et en nécessitant une gestion appropriée des contextes et des stratégies.

3.9.2 Exemple

Supposons que nous développons une application de paiement en ligne, où les utilisateurs peuvent choisir différents modes de paiement (carte de crédit, PayPal, virement bancaire). Nous pouvons utiliser le Design Pattern Strategy pour définir une stratégie pour chaque mode de paiement et permettre au client de choisir la stratégie souhaitée.

```
1 // Strategy Interface
2 interface PaymentStrategy {
3     void pay(int amount);
4 }
5
6 // Concrete Strategies
```

```
7  class CreditCardPayment implements PaymentStrategy {
8      private String cardNumber;
9      private String expirationDate;
10
11     public CreditCardPayment(String cardNumber, String
           expirationDate) {
12         this.cardNumber = cardNumber;
13         this.expirationDate = expirationDate;
14     }
15
16     @Override
17     public void pay(int amount) {
18         System.out.println("Paid " + amount + " using credit card "
               + cardNumber);
19     }
20 }
21
22 class PayPalPayment implements PaymentStrategy {
23     private String email;
24
25     public PayPalPayment(String email) {
26         this.email = email;
27     }
28
29     @Override
30     public void pay(int amount) {
31         System.out.println("Paid " + amount + " using PayPal with
               email " + email);
32     }
33 }
34
35 class BankTransferPayment implements PaymentStrategy {
36     private String accountNumber;
37
38     public BankTransferPayment(String accountNumber) {
39         this.accountNumber = accountNumber;
40     }
41
42     @Override
43     public void pay(int amount) {
44         System.out.println("Paid " + amount + " via bank transfer
               to account " + accountNumber);
45     }
46 }
47
48 // Context (Contexte)
49 class ShoppingCart {
50     private PaymentStrategy paymentStrategy;
51
52     public void setPaymentStrategy(PaymentStrategy paymentStrategy)
```

```
        {
53          this.paymentStrategy = paymentStrategy;
54      }
55
56      public void checkout(int amount) {
57          paymentStrategy.pay(amount);
58      }
59  }
60
61  // Client Code
62  public class Client {
63      public static void main(String[] args) {
64          ShoppingCart cart = new ShoppingCart();
65
66          // Choose Credit Card payment
67          cart.setPaymentStrategy(new CreditCardPayment("1234 5678
                9012 3456", "12/25"));
68          cart.checkout(100); // Output: Paid 100 using credit card
                1234 5678 9012 3456
69
70          // Choose PayPal payment
71          cart.setPaymentStrategy(new PayPalPayment("user@example.com
                "));
72          cart.checkout(50); // Output: Paid 50 using PayPal with
                email user@example.com
73
74          // Choose Bank Transfer payment
75          cart.setPaymentStrategy(new BankTransferPayment("12345678")
                );
76          cart.checkout(200); // Output: Paid 200 via bank transfer
                to account 12345678
77      }
78  }
```

Listing 21 – strategy.java

3.10 Template Method

3.10.1 Description

Le design pattern Template Method est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de définir le squelette d'un algorithme dans une opération, en laissant certains de ses pas aux sous-classes. Ainsi, les sous-classes peuvent redéfinir certaines étapes de l'algorithme sans en changer la structure globale.

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

1. **AbstractClass (Classe Abstraite)** :
   - Classe qui définit le squelette de l'algorithme dans une méthode template.
   - Contient des méthodes concrètes, abstraites ou facultatives, qui sont utilisées par la méthode template.
2. **ConcreteClass (Classe Concrète)** :
   - Implémentation concrète de l'AbstractClass.
   - Redéfinit les méthodes abstraites ou facultatives selon les besoins spécifiques de l'algorithme.

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

1. La classe abstraite définit une méthode template qui encapsule l'algorithme, en appelant séquentiellement les différentes étapes de l'algorithme.
2. Les étapes de l'algorithme qui peuvent varier sont définies comme des méthodes abstraites ou facultatives dans la classe abstraite.
3. Les classes concrètes étendent la classe abstraite et redéfinissent les méthodes abstraites ou facultatives selon les besoins spécifiques de l'algorithme.

L'avantage principal de ce modèle est qu'il permet de définir une structure générale pour un algorithme tout en permettant aux sous-classes de redéfinir certaines étapes de cet algorithme sans en changer la structure globale. Cela favorise la réutilisabilité du code en évitant la duplication de code pour des algorithmes similaires. Cependant, cela peut également rendre le code plus complexe en introduisant des classes abstraites et en nécessitant une bonne compréhension de la structure générale de l'algorithme.

3.10.2 Exemple

Supposons que nous développons un jeu où les joueurs peuvent choisir différentes classes de personnages (guerrier, mage, archer). Chaque classe a une méthode de combat spécifique, mais le déroulement général du combat reste le même pour toutes les classes. Nous pouvons utiliser le Design Pattern Template Method pour définir un modèle de méthode de combat et laisser chaque classe de personnage implémenter ses propres attaques spécifiques.

```
1 // Template Method
2 abstract class Character {
3     public void fight() {
4         collectWeapons();
5         attack();
```

```
6           defend();
7           if (hasSpecialPower()) {
8               useSpecialPower();
9           }
10          System.out.println("Battle ends.");
11      }
12
13      public void collectWeapons() {
14          System.out.println("Collecting weapons.");
15      }
16
17      abstract void attack();
18
19      abstract void defend();
20
21      boolean hasSpecialPower() {
22          return false;
23      }
24
25      void useSpecialPower() {
26          System.out.println("Using special power.");
27      }
28  }
29
30  // Concrete Characters
31  class Warrior extends Character {
32      @Override
33      void attack() {
34          System.out.println("Warrior attacks with sword.");
35      }
36
37      @Override
38      void defend() {
39          System.out.println("Warrior defends with shield.");
40      }
41  }
42
43  class Mage extends Character {
44      @Override
45      void attack() {
46          System.out.println("Mage attacks with magic spell.");
47      }
48
49      @Override
50      void defend() {
51          System.out.println("Mage defends with magic barrier.");
52      }
53
54      @Override
55      boolean hasSpecialPower() {
```

```
56          return true;
57      }
58
59      @Override
60      void useSpecialPower() {
61          System.out.println("Mage uses teleportation spell.");
62      }
63  }
64
65  // Client Code
66  public class Client {
67      public static void main(String[] args) {
68          Character warrior = new Warrior();
69          Character mage = new Mage();
70
71          warrior.fight();
72          mage.fight();
73      }
74  }
```

Listing 22 – template_method.java

3.11 Visitor

3.11.1 Description

Le design pattern Visitor est un modèle de conception appartenant à la catégorie des patrons de conception comportementaux. Son objectif principal est de permettre de définir de nouvelles opérations sur une structure d'objets sans modifier les classes de ces objets.

Voici les principaux éléments qui composent le design pattern Visitor :

1. **Visitor (Visiteur)** :
   - Interface ou classe abstraite commune à tous les visiteurs.
   - Définit des méthodes pour visiter chaque type d'objet de la structure.
2. **ConcreteVisitor (Visiteur Concret)** :
   - Implémentation concrète de l'interface Visitor.
   - Contient l'implémentation des méthodes de visite pour chaque type d'objet de la structure.
3. **Element (Élément)** :
   - Interface ou classe abstraite commune à tous les éléments de la structure.
   - Définit une méthode accept pour accepter les visites des visiteurs.
4. **ConcreteElement (Élément Concret)** :
   - Implémentation concrète de l'interface Element.
   - Implémente la méthode accept en appelant la méthode de visite correspondante sur le visiteur.
5. **ObjectStructure (Structure d'Objets)** :
   - Collection d'objets à visiter.
   - Fournit une méthode pour itérer sur les objets et appeler leur méthode accept pour accepter les visites des visiteurs.

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

1. Les visiteurs implémentent des méthodes de visite pour chaque type d'objet de la structure.
2. Chaque objet de la structure implémente une méthode accept qui appelle la méthode de visite correspondante sur le visiteur.
3. Lorsque la structure doit être parcourue, chaque objet accepte les visites des visiteurs appropriés.

L'avantage principal de ce modèle est qu'il permet d'ajouter de nouvelles opérations sur une structure d'objets sans modifier les classes de ces objets. Cela favorise la modularité et la maintenabilité du code en séparant les opérations à appliquer sur les objets de la structure dans des classes de visiteurs distinctes. Cependant, cela peut augmenter la complexité du code en introduisant de nombreuses classes de visiteurs et en nécessitant une bonne compréhension de la structure d'objets et des opérations à appliquer.

3.11.2 Exemple

Supposons que nous développons une application de dessin avec différentes formes géométriques (cercle, carré, triangle). Nous voulons pouvoir ajouter de nouvelles fonctionnalités, telles que le calcul de l'aire ou le déplacement des formes, sans modifier les classes existantes. Nous pouvons utiliser le Design Pattern Visitor pour définir un visiteur externe pour chaque nouvelle fonctionnalité.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  // Visitor Interface
5  interface ShapeVisitor {
6      void visitCircle(Circle circle);
7      void visitSquare(Square square);
8      void visitTriangle(Triangle triangle);
9  }
10
11 // Concrete Visitors
12 class AreaCalculator implements ShapeVisitor {
13     private double totalArea = 0;
14
15     @Override
16     public void visitCircle(Circle circle) {
17         double area = Math.PI * Math.pow(circle.getRadius(), 2);
18         totalArea += area;
19     }
20
21     @Override
22     public void visitSquare(Square square) {
23         double area = Math.pow(square.getSide(), 2);
24         totalArea += area;
25     }
26
27     @Override
28     public void visitTriangle(Triangle triangle) {
29         double area = 0.5 * triangle.getBase() * triangle.getHeight
               ();
30         totalArea += area;
31     }
32
33     public double getTotalArea() {
34         return totalArea;
35     }
36 }
37
38 class ShapeMover implements ShapeVisitor {
39     private int xOffset;
40     private int yOffset;
41
42     public ShapeMover(int xOffset, int yOffset) {
```

```
43          this.xOffset = xOffset;
44          this.yOffset = yOffset;
45      }
46
47      @Override
48      public void visitCircle(Circle circle) {
49          circle.setX(circle.getX() + xOffset);
50          circle.setY(circle.getY() + yOffset);
51      }
52
53      @Override
54      public void visitSquare(Square square) {
55          square.setX(square.getX() + xOffset);
56          square.setY(square.getY() + yOffset);
57      }
58
59      @Override
60      public void visitTriangle(Triangle triangle) {
61          triangle.setX(triangle.getX() + xOffset);
62          triangle.setY(triangle.getY() + yOffset);
63      }
64  }
65
66  // Element Interface
67  interface Shape {
68      void accept(ShapeVisitor visitor);
69  }
70
71  // Concrete Elements
72  class Circle implements Shape {
73      private int x;
74      private int y;
75      private double radius;
76
77      public Circle(int x, int y, double radius) {
78          this.x = x;
79          this.y = y;
80          this.radius = radius;
81      }
82
83      public int getX() {
84          return x;
85      }
86
87      public void setX(int x) {
88          this.x = x;
89      }
90
91      public int getY() {
92          return y;
```

```
93      }
94
95      public void setY(int y) {
96          this.y = y;
97      }
98
99      public double getRadius() {
100         return radius;
101     }
102
103     @Override
104     public void accept(ShapeVisitor visitor) {
105         visitor.visitCircle(this);
106     }
107 }
108
109 class Square implements Shape {
110     private int x;
111     private int y;
112     private int side;
113
114     public Square(int x, int y, int side) {
115         this.x = x;
116         this.y = y;
117         this.side = side;
118     }
119
120     public int getX() {
121         return x;
122     }
123
124     public void setX(int x) {
125         this.x = x;
126     }
127
128     public int getY() {
129         return y;
130     }
131
132     public void setY(int y) {
133         this.y = y;
134     }
135
136     public int getSide() {
137         return side;
138     }
139
140     @Override
141     public void accept(ShapeVisitor visitor) {
142         visitor.visitSquare(this);
```

```
143         }
144     }
145
146     class Triangle implements Shape {
147         private int x;
148         private int y;
149         private int base;
150         private int height;
151
152         public Triangle(int x, int y, int base, int height) {
153             this.x = x;
154             this.y = y;
155             this.base = base;
156             this.height = height;
157         }
158
159         public int getX() {
160             return x;
161         }
162
163         public void setX(int x) {
164             this.x = x;
165         }
166
167         public int getY() {
168             return y;
169         }
170
171         public void setY(int y) {
172             this.y = y;
173         }
174
175         public int getBase() {
176             return base;
177         }
178
179         public int getHeight() {
180             return height;
181         }
182
183         @Override
184         public void accept(ShapeVisitor visitor) {
185             visitor.visitTriangle(this);
186         }
187     }
188
189     // ObjectStructure
190     class ShapeCollection {
191         private List<Shape> shapes = new ArrayList<>();
192
```

```
193     public void addShape(Shape shape) {
194         shapes.add(shape);
195     }
196 
197     public void removeShape(Shape shape) {
198         shapes.remove(shape);
199     }
200 
201     public void accept(ShapeVisitor visitor) {
202         for (Shape shape : shapes) {
203             shape.accept(visitor);
204         }
205     }
206 }
207 
208 // Client Code
209 public class Client {
210     public static void main(String[] args) {
211         ShapeCollection shapeCollection = new ShapeCollection();
212         shapeCollection.addShape(new Circle(10, 20, 5));
213         shapeCollection.addShape(new Square(30, 40, 10));
214         shapeCollection.addShape(new Triangle(50, 60, 8, 12));
215 
216         AreaCalculator areaCalculator = new AreaCalculator();
217         ShapeMover shapeMover = new ShapeMover(5, 5);
218 
219         shapeCollection.accept(areaCalculator);
220         shapeCollection.accept(shapeMover);
221 
222         System.out.println("Total area: " + areaCalculator.
               getTotalArea());
223     }
224 }
```

Listing 23 – visitor.java