Faculté des sciences
Aix Marseille Université

Année 2024-25

Domaine Sciences et Technologies
Master informatique
Génie logiciel : Patrons de conception
Codes UE : SINA09EL
Patrons de conception

Patrons de conception

Arnaud Labourel

1 Patrons de conception

1.1 Introduction

Les *design patterns* (patrons de conception) sont des recettes de conception orientées objet. Ce sont des solutions classiques à des problèmes récurrents de la conception de logiciels. Ils consistent en des plans ou des schémas que l'on peut personnaliser afin de résoudre un problème récurrent dans notre code. Ils furent introduits par quatre développeurs (connus sous le nom de *gang of four*) : Gamma, Helm, Johnson et Vlissides en 1995. Ces développeurs sont partis de l'idée qu'il existe des recettes de conception permettant de répondre de manière similaire à des problèmes semblables. En 1995 donc, ces quatre auteurs publient leur livre "Design Patterns : Elements of Reusable Object-Oriented Software" qui détaille vingt-trois solutions répondant à des problèmes récurrents en développement logiciel. Ces recettes, patrons ou *design patterns* peuvent être classés en trois catégories :

— création : instanciation et configuration des classes et des objets ; — structure : organisation des classes d'un programme dans une structure plus importante ; — comportement : organisation des objets en vue de leur collaboration.

Les patrons de conception sont décrits de manière générique afin de pouvoir être adaptés aux différents problèmes à résoudre : ils ne peuvent donc pas s'appliquer directement à toutes les situations sans adaptation. Vous ne pouvez donc pas vous contenter de trouver un patron et de le recopier dans votre programme. Un patron, ce n'est pas un bout de code spécifique, mais plutôt un concept général pour résoudre un problème précis. Il faut donc toujours réfléchir à leur utilisation dans un contexte donné. Néanmoins, ils permettent de ne pas réinventer la roue, car ils couvrent la plupart des situations aux problématiques auxquelles un développeur est confrontées.

1.2 Patrons de création

Les patrons de création fournissent des mécanismes de création d'objets qui permettent d'augmenter la flexibilité et la réutilisation du code. Les principaux patrons de conceptions sont les suivants :

- *Factory Method* (fabrique) : définit une interface pour la création d'objets dans une classe mère, mais délègue aux sous-classes le choix des types d'objets à créer.
- *Abstract Factory* (Fabrique abstraite) : permet de créer des familles d'objets apparentés sans préciser leur classe concrète.
- *Builder* (monteur) : permet de construire des objets complexes étape par étape. Ce patron permet de construire différentes variations ou représentations d'un objet en utilisant le même code de construction.
- *Prototype* (prototype) : permet de créer de nouveaux objets à partir d'objets existants sans rendre le code dépendant de leur classe.
- *Singleton* (Singleton) : permet de garantir que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

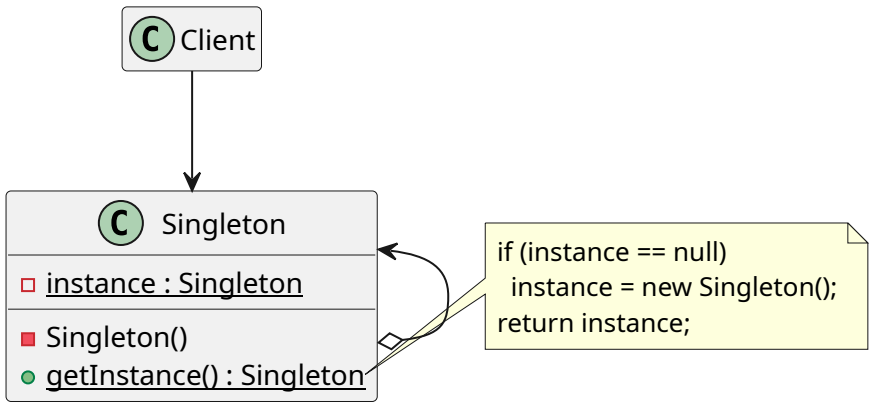
Par la suite, nous allons détailler quelques patrons de création.

1.2.1 *Singleton* (singleton)

Singleton est un patron de conception de création qui garantit que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance. Le singleton règle deux problèmes à la fois :

— Il garantit l'unicité d'une instance pour une classe. Pour quelle raison voudrait-on maîtriser le nombre d'instances d'une classe ? En général, cette situation se présente lorsque l'on veut contrôler l'accès à une ressource partagée — une base de données ou un fichier par exemple.
— Il fournit un point d'accès global à cette instance, protège cette instance en l'empêchant d'être modifiée.

La solution consiste à rendre le constructeur privé (pour empêcher la création d'une instance sans contrôle), et à créer une méthode statique pour contrôler l'instanciation.



Le code associé est le suivant :

```
public class Singleton
{
  private static Singleton instance = null;
  /**
   * Private constructor to prevent construction outside the class.
   */
  private Singleton()
  {
    // Code of the constructor
  }
  /**
   * Keyword synchronized is used to allow only one thread to execute
   * the method at any given time (preventing concurrent access).
   *
   * @return the instance of the unique instance of the class.
   */
  public synchronized static Singleton getInstance()
  {
    if (instance == null)
    {
```

```
      instance = new Singleton();
    }
    return instance;
  }
}
```

1.2.2 *Factory Method* (fabrique)

Fabrique est un patron de conception de création qui définit une interface pour créer des objets dans une classe mère ou interface, mais délègue le choix des types d'objets à créer aux sous-classes. La fabrique permet donc à une classe de déléguer l'instanciation à des sous-classes.

Supposons que nous disposions de l'interface suivante :

```
public interface Button {
  public void draw();
}
```

Il existe une première implémentation de cette interface :

```
public class SimpleButton implements Button  {
  public void draw() {
    System.out.println("Simple button.");
  }
}
```

La classe `SimpleButton` est instanciée dans de nombreuses autres classes et donc son constructeur apparaît à de nombreux endroits. Supposons que nous fassions une autre implémentation de `Button` :

```
public class ModernButton implements Button  {
  public void draw() {
    System.out.println("Modern button.");
  }
}
```

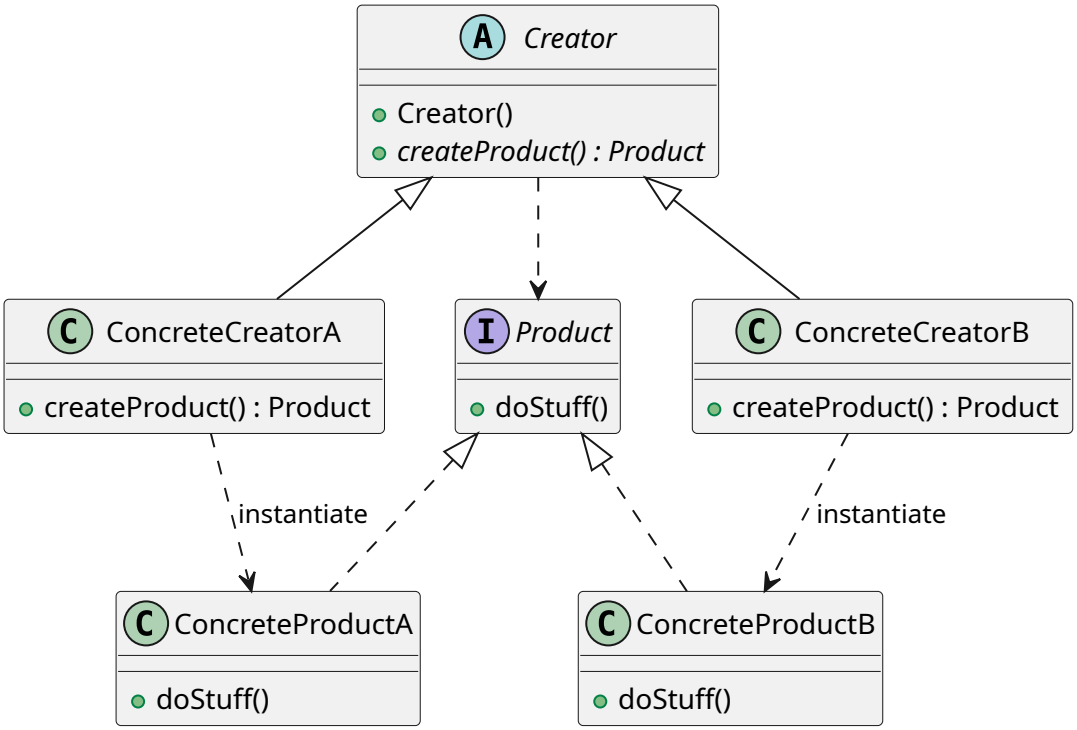
Afin d'utiliser ce nouveau bouton, nous devons modifier toutes les instanciations (appels au constructeur) présentes dans notre code. Il y a donc une violation d'OCP. Afin d'éviter cela, on va donc appliquer le patron de conception fabrique. Une fabrique consiste à isoler la création des objets de leurs utilisations :

```
public class ButtonFactory {
  public Button createButton() {
    return new SimpleButton();
  }
}
```

Toutes les instanciations doivent se faire via cette classe. Les modifications nécessaires à l'utilisation de la classe `ModernButton` sont isolées :

```
public class ButtonFactory {
  public Button createButton() {
    return new ModernButton();
  }
}
```

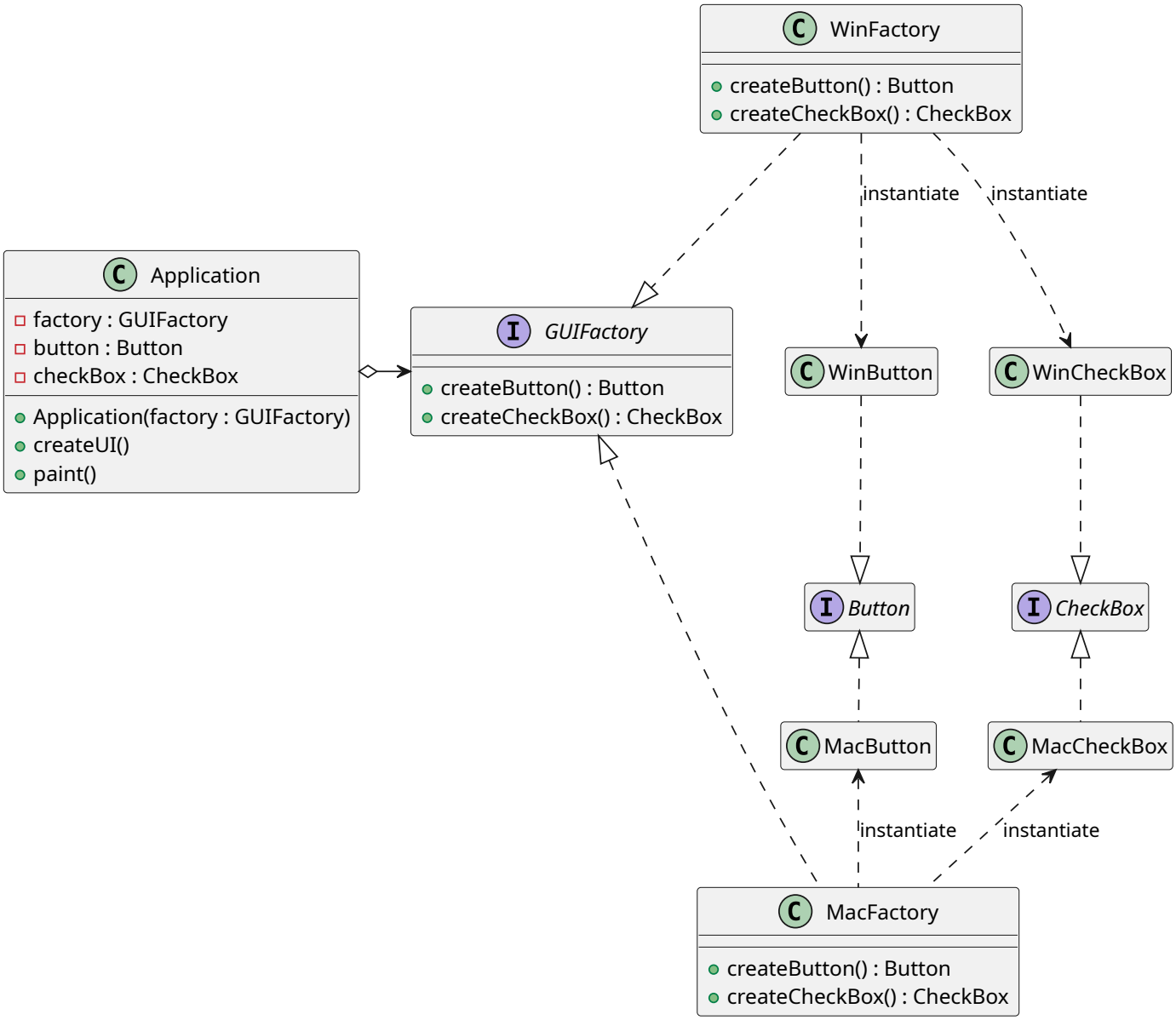
De manière générale le patron de conception a la forme suivante dans laquelle a une classe abstraite de création d'objets qui peut être étendue de plusieurs manières différentes.



1.2.3 *Abstract factory* (fabrique abstraite)

Fabrique abstraite est un patron de conception qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète. On va définir une interface contenant plusieurs méthodes de création d'objets et on implémentera cette interface avec des classes qui garantiront une cohérence entre les différents objets créés.

Pour illustrer ce principe, nous allons considérer la création des éléments d'une interface utilisateur pouvant fonctionner sur plusieurs systèmes : mac ou windows. Pour simplifier notre exemple, nous allons considérer qu'il n'y a que deux types d'éléments dans l'interface utilisateur : des boutons (interface `Button`) et des coches (interface `CheckMark`). Dans notre cas, il faut donc pouvoir créer ces deux types d'objets. On définit donc une interface `GUIFactory` contenant deux méthodes de création d'objet (une pour créer des `Button` et l'autre pour créer des `CheckMark`). Cette interface aura deux implémentations `WinFactory` qui crée des éléments pour windows et `MacFactory` qui crée des éléments pour mac. L'application graphique peut donc utiliser une instance d'une de ces deux classes (suivant le système visé) afin de créer les éléments de l'interface utilisateur. Vous trouverez ci-dessous le diagramme de classes correspondant à cette organisation du code.



L'utilisation du patron fabrique abstraite a ici deux avantages :

— Cela permet de cacher les classes concrètes à l'application et donc d'éviter des dépendances à des classes concrètes (respect de DIP).

— Cela permet de garantir la cohérence entre les objets construits par une même fabrique. Ces objets partagent une propriété commune (dans notre exemple un système d'exploitation). Cela permet d'éviter certaines incohérences comme dans notre exemple, des éléments pour des systèmes différents.

Comme on vient de le voir, la Fabrique Abstraite est un patron de conception de création qui permet de créer des familles d'objets apparentés sans préciser leur classe concrète. Une famille d'objets est un ensemble de classes ayant une certaine cohérence. Par exemple, si on considère des triplets : (Transport, Moteur, Contrôles), plusieurs variantes peuvent exister :

— (Voiture, MoteurCombustion, Volant)

— (Avion, MoteurRéaction, Manche)

Si votre programme fonctionne sans famille de produits, vous n'avez pas besoin d'une fabrique abstraite et une fabrique simple (patron vu précédemment) peut suffire.

1.3 Patrons structurels

Les patrons structurels servent à guider l'assemblage des objets et des classes afin d'obtenir des structures plus grandes tout en gardant celles-ci flexibles et efficaces. Les principaux patrons de conception structurels sont les suivants :

— *Adapter* (adaptateur) : permet de faire collaborer des objets ayant des interfaces normalement incompatibles.
— *Bridge* (pont) : permet de séparer une grosse classe ou un ensemble de classes connexes en deux hiérarchies — abstraction et implémentation — qui peuvent évoluer indépendamment l'une de l'autre.
— *Composite* (composite) : permet d'agencer les objets dans des arborescences afin de pouvoir traiter celles-ci comme des objets individuels.
— *Decorator* (décorateur) : permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.
— *Facade* (façade) : procure une interface qui offre un accès simplifié à une librairie, un framework ou à n'importe quel ensemble complexe de classes.
— *Flyweight* (poids mouche) :Permet de stocker plus d'objets dans la RAM en partageant les états similaires entre de multiples objets, plutôt que de stocker les données dans chaque objet.
— *Proxy* (procuration) : permet de fournir un substitut d'un objet. Une procuration donne le contrôle sur l'objet original, vous permettant d'effectuer des manipulations avant ou après que la demande ne lui parvienne.

Nous allons détailler le principe de certains patrons structurels.

1.3.1 Adapter (adaptateur)

L'Adaptateur est un patron de conception structurel qui permet de faire collaborer des objets ayant des interfaces normalement incompatibles. Supposons que la classe suivante existe :

```
public class Drawer {
  public void draw(Pencil pencil) {
    pencil.drawLine(0,0,10,10);
    pencil.drawCircle(5,5,5);
    pencil.drawLine(10,0,0,10);
  }
}
```

Nous avons également l'interface suivante qui permet de dessiner des segments et des cercles à partir des coordonnées des points et du rayon du cercle :

```
public interface Pencil {
  public void drawLine(int x1, int y1, int x2, int y2);
  public void drawCircle(int x, int y, int radius);
}
```

Cette classe `Drawer` et cette interface `Pencil` ne peuvent pas être modifiées. Nous avons également la classe `PointPencil` suivante qui permet de dessiner des segments et des cercles à partir d'objets `Point2D` :

```
public class PointPencil {
  public void drawLineWithPoints(Point2D p1, Point2D p2) {
    /* ... */
  }

  public void drawCircleWithPoint(Point2D center, int radius) {
    /* ... */
  }
}
```

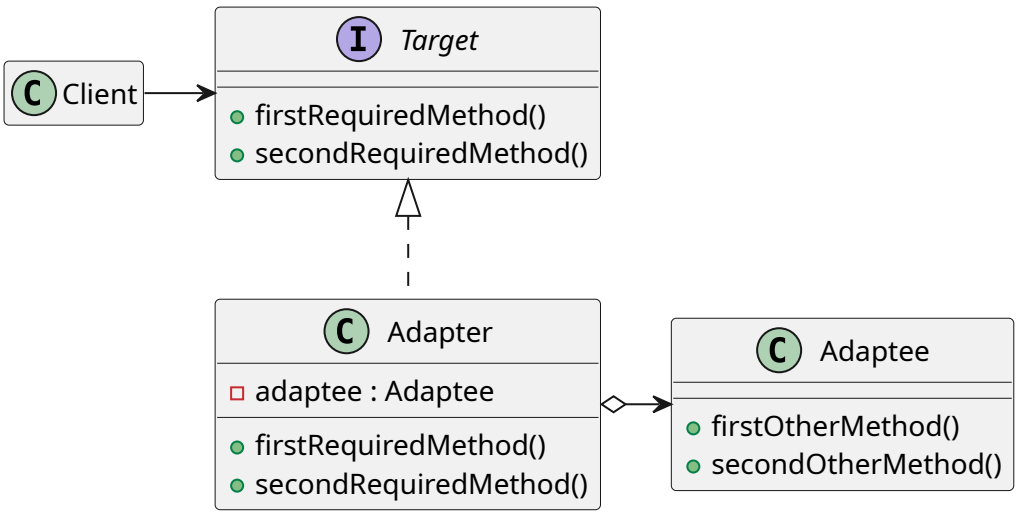
Nous souhaitons utiliser la classe `PointPencil` comme un `Pencil` pour qu'elle puisse être utilisée par la classe `Drawer`. Pour ce faire, nous définissons l'adaptateur suivant :

```
public class PointPencilAdapter implements Pencil {

  private PointPencil pointPencil;

  public CrayonAdapter(PointPencil pointPencil) {
    this.pointPencil = pointPencil;
  }

  public void drawLine(int x1, int y1, int x2, int y2) {
    pointPencil.drawLineWithPoints(new Point2D(x1, y1),
                                   new Point2D(x2, y2));
  }

  public void drawCircle(int x, int y, int radius) {
    pointPencil.drawCircleWithPoint(new Point(x, y), radius);
  }
}
```

Cette classe est utilisable de la façon suivante :

```
Pencil pencil = new PointPencilAdapter(new PointPencil());
Drawer drawer = new Drawer();
drawer.draw(pencil);
```

De manière générale, le patron a le format suivant :



1.3.2 *Decorator* (décorateur)

Décorateur est un patron de conception structurel qui permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements. Supposons que nous avons la classe suivante :

```
public class ArrayStack {
  private int[] stack = new int[10];
  private int size = 0;

  public void push(int value) {
    list[size] = value;
    size++;
  }

  public int pop() {
    size--;
    return list[size];
  }
}
```

Nous souhaitons ajouter des logs pour débugger notre programme :

```
public class ArrayStack {
  private int[] stack = new int[10];
  private int size = 0;

  public void push(int value) {
    System.out.println("push("+value+")");
    list[size] = value;
    size++;
  }
```

```
  public int pop() {
     System.out.println("pop()");
     size--;
     return list[size];
  }
}
```

Cette modification a été réalisée en modifiant une classe existante. De plus, une nouvelle modification est nécessaire pour retirer les logs. Nous avons donc un non-respect d'OCP.

Afin de résoudre cette difficulté, nous allons utiliser le patron décorateur. On commence par définir l'interface suivante :

```
public interface Stack {
  public void push(int value);
  public int pop();
}
```

On peut facilement faire en sorte que `ArrayStack` implémente cette interface (en utilisant le mot-clé `implements`) car la classe `ArrayStack` définit déjà les méthodes de l'interface.

```
public class ArrayStack implements Stack {
  /* ... */
}
```

Maintenant, il suffit de définir un décorateur :

```
public class VerboseStack implements Stack {
  private Stack stack;

  public VerboseStack(Stack stack) { this.stack = stack; }

  public void push(int value) {
    System.out.println("push("+value+")");
    stack.push(value);
  }

  public int pop() {
     System.out.println("pop()");
     return stack.pop();
  }
}
```

Supposons que nous ayons le code suivant :

```
Stack stack = new ArrayStack(10);
stack.push(2);
stack.pop();
```

Il est très facile d'introduire le décorateur :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
stack.push(2);
stack.pop();
```

Ce code produit la sortie suivante :

```
push(2);
pop();
```

Nous pouvons aussi définir un nouveau décorateur :

```
public class CounterStack implements Stack {
  private Stack stack;
  private int size;

  public CounterStack(Stack stack) {
    this.stack = stack;
    size = 0;
  }

  public void push(int value) {
    size++;
    stack.push(value);
  }

  public int pop() {
    size--;
    return stack.pop();
  }

  public int getSize() {
    return size;
  }
}
```

Un exemple d'utilisateur du décorateur précédent :

```
Stack stack = new ArrayStack(10);
CounterStack counterStack = new CounterStack(stack);
stack = counterStack;
```

```
stack.push(2);
stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
1
```

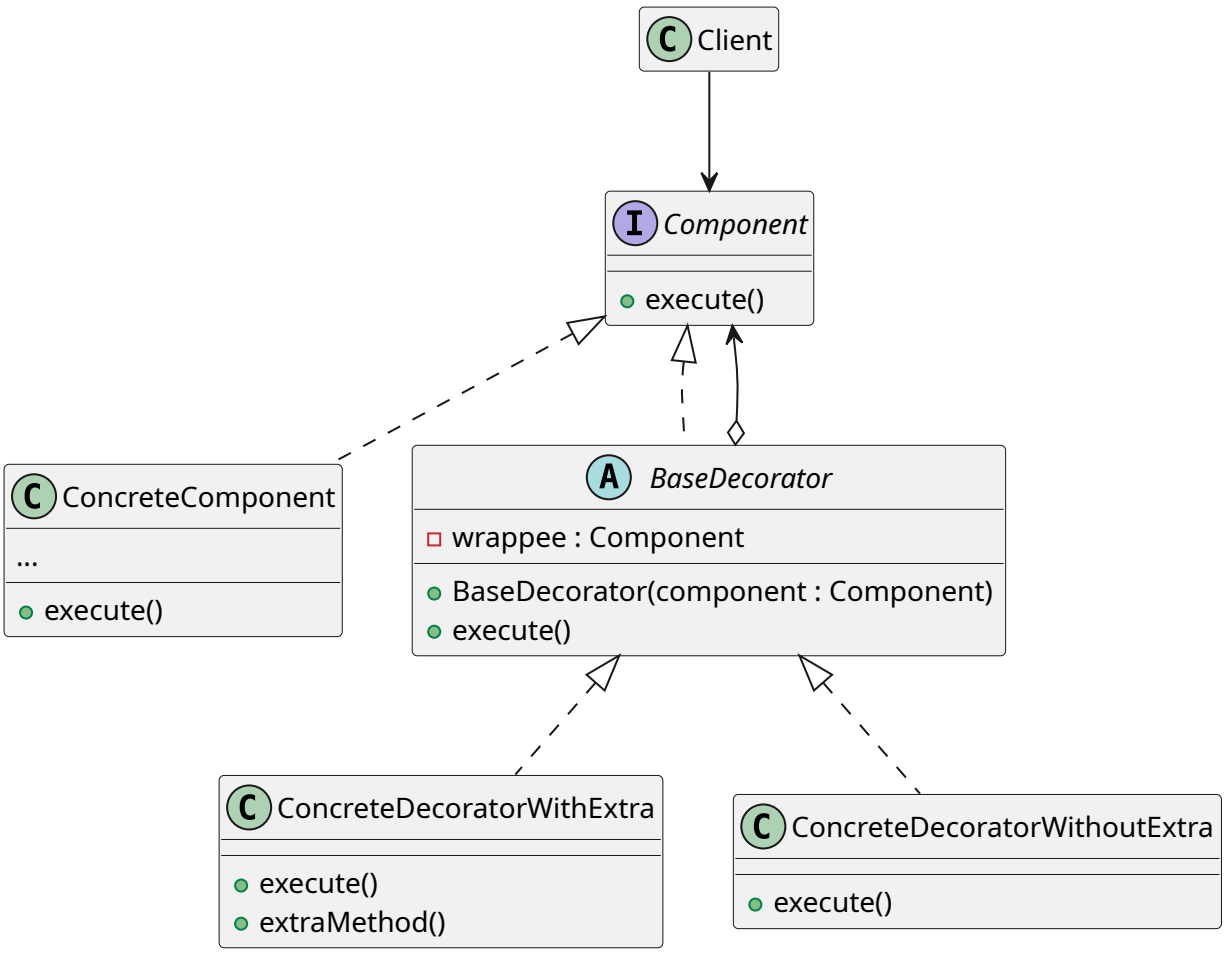
Il est possible d'utiliser plusieurs décorateurs simultanément :

```
Stack stack = new ArrayStack(10);
Stack verboseStack = new VerboseStack(stack);
CounterStack counterStack = new CounterStack(verboseStack);
stack = counterStack;
stack.push(2);
stack.push(3);
stack.pop();
System.out.println(counterStack.getSize());
```

Ce code produit la sortie suivante :

```
push(2);
push(3);
pop();
1
```

La structure générale du patron décorateur est la suivante :



1.4 Patrons comportementaux

Les comportementaux qui permettent de gérer les algorithmes et la répartition des responsabilités entre les objets. Les principaux patrons comportementaux sont les suivants :

— *Chain of Responsibility* (chaîne de responsabilité) : permet de faire circuler une demande dans une chaîne de *handlers*. Lorsqu'un *handler* reçoit une demande, il décide de la traiter ou de l'envoyer au *handler* suivant de la chaîne.

— *Command* (commande) : Prend une action à effectuer et la transforme en un objet autonome qui contient tous les détails de cette action. Cette transformation permet de paramétrer des méthodes avec différentes actions, planifier leur exécution, les mettre dans une file d'attente ou d'annuler des opérations effectuées.

— *Iterator* (itérateur) : permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, etc.).

— *Mediator* (médiateur) : Permet de diminuer les dépendances chaotiques entre les objets. Ce patron restreint les communications directes entre les objets et les force à collaborer uniquement via un objet médiateur.

— *Memento* (Mémento) : Permet de sauvegarder et de rétablir l'état précédent d'un objet sans révéler les détails de son implémentation.

— *Observer* (observateur) : Permet de mettre en place un mécanisme de souscription pour envoyer des

notifications à plusieurs objets, au sujet d'événements concernant les objets qu'ils observent.

— *State* (état) : Modifie le comportement d'un objet lorsque son état interne change. L'objet donne l'impression qu'il change de classe.

— *Strategy* (stratégie) : Permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables.

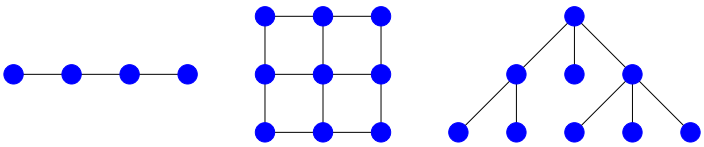
— *Template Method* (patron de méthode) : Permet de mettre le squelette d'un algorithme dans la classe mère, mais laisse les sous-classes redéfinir certaines étapes de l'algorithme sans changer sa structure.

— *Visitor* (Patron de méthode) : Permet de séparer les algorithmes et les objets sur lesquels ils opèrent.

1.4.1 *Iterator* (itérateur)

Itérateur est un patron de conception comportemental qui permet de parcourir les éléments d'une collection sans révéler sa représentation interne (liste, pile, arbre, ...).

Les collections font partie des types de données les plus usitées en programmation. Elles servent de conteneur pour un groupe d'objets généralement en utilisant une structure linéaire : listes, tableaux ... Néanmoins, certaines d'entre elles sont basées sur structures plus complexes comme des arbres, des graphes ou d'autres structures complexes de données.



Quelle que soit sa structure, une collection doit fournir un moyen d'accéder à ses éléments pour permettre au code de les utiliser. Elle doit donner la possibilité de parcourir tous ses éléments sans passer plusieurs fois par les mêmes. Le but du patron de conception itérateur est d'extraire le comportement qui permet de parcourir une collection et de le mettre dans un objet que l'on nomme itérateur.

En java (c'est le cas aussi pour d'autres langages), le patron de conception itérateur est défini dans le langage et la bibliothèque standard. La première chose à définir pour le patron de conception itérateur est l'interface itérable qui indique que l'on peut itérer (parcourir les éléments) sur la collection. En Java, l'interface est la suivante (on a volontairement omis certaines méthodes) :

```
public interface Iterable<T> {
    /**
     * Returns an iterator over elements of type {@code T}.
     *
     * @return an Iterator.
     */
    Iterator<T> iterator();
}
```

L'interface `Iterable` est d'ailleurs étendue par l'interface `Collection` de Java. Elle ne contient qu'une méthode qui retourne un objet pour itérer sur la collection.

L'interface `Iterator` donne les méthodes à implémenter pour un objet permettant de parcourir les éléments d'une collection :

```
public interface Iterator<E> {
    /**
     * Returns {@code true} if the iteration has more elements.
     * (In other words, returns {@code true} if {@link #next} would
     * return an element rather than throwing an exception.)
     *
     * @return {@code true} if the iteration has more elements
     */
    boolean hasNext();

    /**
     * Returns the next element in the iteration.
     *
     * @return the next element in the iteration
     * @throws NoSuchElementException if the iteration has no more elements
     */
    E next();
}
```

On a deux méthodes à implémenter `hasNext` qui permet de tester s'il reste des éléments sur lesquels on peut itérer et `next` qui renvoie le prochain élément dans l'ordre d'itération et met à jour l'itérateur pour que le prochain appel de `next` renvoie l'élément après et ainsi de suite. Pour itérer sur un tableau, on peut écrire le code suivant :

```
public class IteratorArray {
  Object[] array;
  int position = 0;
  IteratorArray(Object[] array){
    this.array = array;
  }

  boolean hasNext(){
    return (position<array.length);
  }

  Object next(){
    position++;
    return array[position-1];
  }
}
```

Pour illustrer l'utilisation du patron de conception itérateur, on considère la classe suivante qui permet de stocker des éléments dans une grille avec des lignes et des colonnes.

```
public class GridContainer<E> implements Iterable<E> {
  private final int numberOfRows;
  private final int numberOfColumns;
  private final Object[][] elements;
  public GridContainer(int numberOfRows, int numberOfColumns) {
    if(numberOfRows <= 0)
      throw new IllegalArgumentException("The number of rows must be positive and not equal to " +
      ↪ numberOfRows);
    if(numberOfColumns <= 0)
      throw new IllegalArgumentException("The number of columns must be positive and not equal to "
      ↪ + numberOfColumns);
    this.numberOfRows = numberOfRows;
    this.numberOfColumns = numberOfColumns;
    elements = new Object[numberOfRows][numberOfColumns];
  }
  @SuppressWarnings("unchecked")
  public E getElement(int rowIndex, int columnIndex) {
    return (E) elements[rowIndex][columnIndex];
  }
  public void setElement(int rowIndex, int columnIndex, E value) {
    elements[rowIndex][columnIndex] = value;
  }

  public int getNumberOfRows() {
    return numberOfRows;
  }

  public int getNumberOfColumns() {
    return numberOfColumns;
  }

  @Override
  public Iterator<E> iterator() {
    return new GridIterator<>(this);
  }
}
```

L'itérateur permettant de visiter les éléments de la grille ligne par ligne a le code suivant :

```
public class GridContainer<E> implements Iterable<E> {
  private final int numberOfRows;
  private final int numberOfColumns;
  private final Object[][] elements;
  public GridContainer(int numberOfRows, int numberOfColumns) {
    if(numberOfRows <= 0)
      throw new IllegalArgumentException("The number of rows must be positive and not equal to " +
      ↪ numberOfRows);
```

```
    if(numberOfColumns <= 0)
      throw new IllegalArgumentException("The number of columns must be positive and not equal to "
      ↪  + numberOfColumns);
    this.numberOfRows = numberOfRows;
    this.numberOfColumns = numberOfColumns;
    elements = new Object[numberOfRows][numberOfColumns];
  }
  @SuppressWarnings("unchecked")
  public E getElement(int rowIndex, int columnIndex) {
    return (E) elements[rowIndex][columnIndex];
  }
  public void setElement(int rowIndex, int columnIndex, E value) {
    elements[rowIndex][columnIndex] = value;
  }

  public int getNumberOfRows() {
    return numberOfRows;
  }

  public int getNumberOfColumns() {
    return numberOfColumns;
  }

  @Override
  public Iterator<E> iterator() {
    return new GridIterator<>(this);
  }
}
```

Puisque `GridContainer` implémente `Iterable`, on peut l'utiliser dans une boucle *for each* (aussi appelé *enhanced for*) qui permet directement de parcourir les éléments de `GridContainer` dans l'ordre défini par l'itérateur. Si on considére le code suivant :

```
GridContainer<String> grid = new GridContainer<>(2,3);
for(int row = 0; row<grid.getNumberOfRows(); row++)
  for (int column = 0; column<grid.getNumberOfColumns(); column++)
    grid.setElement(row,column,"(" + row + ", " + column + ")");
for (String element : grid)
  System.out.println(element);
```

Ce code produit la sortie suivante :

```
(0, 0)
(0, 1)
(0, 2)
(1, 0)
```

```
(1, 1)
(1, 2)
```

1.4.2 *Visitor* (visiteur)

Visiteur est un patron de conception comportemental qui vous permet de séparer les implémentations de méthodes et les objets sur lesquels ils opèrent.

Dans une section précédente, nous avons défini l'interface suivante :

```
public interface Shape {
  void draw(GraphicsContext context);
  float area();
}
```

Dans cette interface, sont regroupés deux services assez différents (calcul géométrique et dessin dans un contexte graphique) ce qui constitue une violation de SRP.

Cette interface est implémentée par la classe suivante :

```
public class Rectangle implements Shape {
    public float x, y, w, h;

    public Rectangle(float x, float y, float w, float h) {
        this.x = x;
        this.y = y;
        this.w = w;
        this.h = h;
    }

    public void draw(GraphicsContext context) {
        context.strokeRect(x, y, h, w);
    }

    public float area() {
      return w * h;
    }
}
```

Dans cette classe, il y a l'implémentation de deux services assez différents. On peut imaginer des raisons différentes pour lesquelles le code des deux méthodes pourrait changer (changement de bibliothèque graphique pour `draw` et passage à un type de retour `double` pour `area` afin d'éviter des problèmes de représentation d'aires trop grande pour être stocké dans un `float`). La classe ne respecte donc pas SRP dans cette optique.

On peut représenter les correspondances entre classes et méthodes dans un tableau :

		classes			
		Rectangle	Circle	Polygon	...
méthodes	draw()				
	area()				
	...				

Dans le tableau ci-dessus, on a :

- Une classe par colonne
- Une méthode par ligne
- SRP violé, car plusieurs responsabilités dans chaque classe
- OCP violé, car le nombre de lignes peut augmenter

Une solution est de définir une classe par ligne en utilisant le patron de conception visiteur. On commence par créer une interface `Shape` qui va accepter des visiteurs.

```
public interface Shape {
  <R> R accept(ShapeVisitor<R> visitor);
}
```

Ensuite, on définit une interface pour les visiteurs avec une méthode de visite par classe à visiter. L'interface est paramétrée par le type de retour de la méthode de visite. Cela permet d'avoir par exemple un visiteur pour le calcul des aires qui renvoie un `Float` et un visiteur de dessin qui ne renvoie rien avec `Void`.

```
public interface ShapeVisitor<R> {
  R visit(Rectangle rectangle);
  R visit(Circle circle);
}
```

Dans les classes `Circle` et `Rectangle`, qui implémentent chacune l'interface `Shape`, n'ont plus qu'une seule méthode `accept` et un constructeur :

```
public class Circle implements Shape {
  public final float x, y;
  public final float radius;

  public Circle(float x, float y, float radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  @Override
  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
```

```
}
```

```
public class Rectangle implements Shape {
  public float x, y, w, h;

  public Rectangle(float x, float y, float w, float h) {
    this.x = x; this.y = y; this.w = w; this.h = h;
  }

  @Override
  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

La méthode `accept` se contente d'appeler la méthode `visit` du visiteur sur l'objet courant et de renvoyer le résultat. Toute l'implémentation de la méthode sera donc dans le visiteur. On peut remarquer que bien que le nom de fonction appelée soit le même (`visit`), les méthodes appelées sont différentes. En effet, dans `Circle` c'est la méthode `R visit(Circle circle)` qui est appelée, car `this` est de type `Circle` alors que dans `Rectangle`, c'est la méthode `R visit(Rectangle rectangle)` qui est appelée.

Maintenant, on doit fournir des implémentations de l'interface `Visitor` qui permettent le dessin et le calcul de l'aire. On commence donc par l'implémentation du visiteur par une classe `DrawerVisitor` :

```
public class DrawerVisitor implements ShapeVisitor<Void> {

  private GraphicsContext context;

  public DrawerVisitor(GraphicsContext context) {
    this.context = context;
  }

  public void draw(List<Shape> shapes) {
    for (Shape shape : shapes)
      shape.accept(this);
  }

  @Override
  public Void visit(Rectangle rectangle) {
    context.strokeRect(rectangle.x, rectangle.y,
                       rectangle.h, rectangle.w);
    return null;
  }

  @Override
  public Void visit(Circle circle) {
```

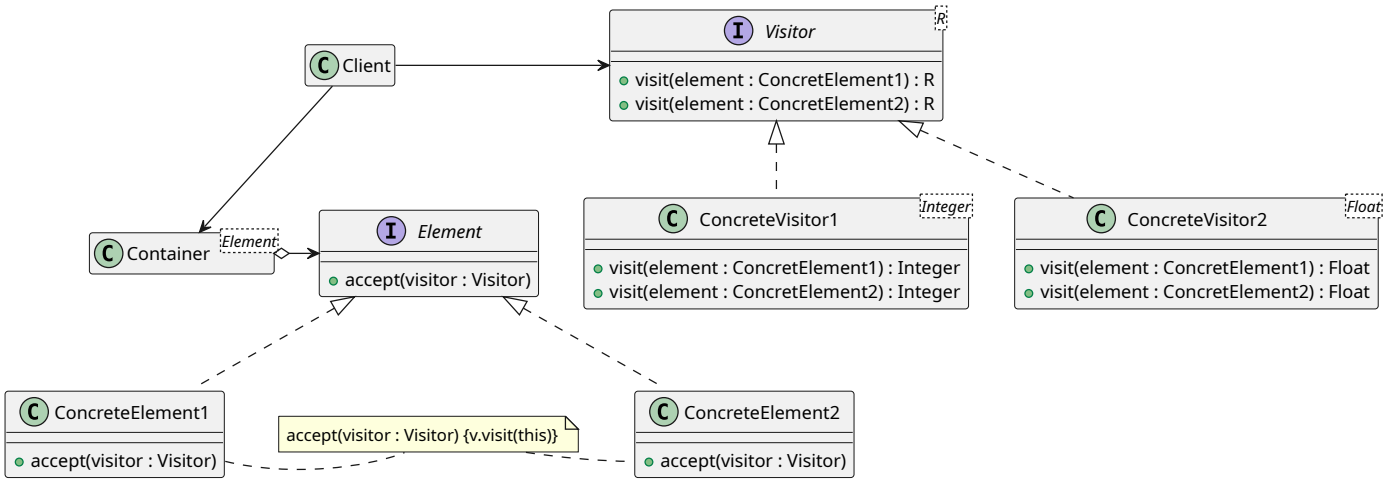
```
    context.strokeOval(circle.x - circle.radius,
                       circle.y - circle.radius,
                       circle.radius*2, circle.radius*2);
    return null;
  }
}
```

Ici, on a une petite particularité du fait que le type de retour est `Void` qui représente le type `void` dans le cas d'un type générique. Pour ce type de retour, la seule valeur acceptée est `null` d'où les `return null` dans les méthodes de visite.

On peut continuer avec l'implémentation du visiteur `AreaVisitor` :

```
public class AreaVisitor implements ShapeVisitor<Float> {

  public float sumOfArea(List<Shape> shapes) {
    float sum = 0;
    for (Shape shape : shapes)
      sum += shape.accept(this);
    return sum;
  }


  @Override
  public Float visit(Rectangle rectangle) {
    return rectangle.w * rectangle.h;
  }

  @Override
  public Float visit(Circle circle) {
    return Math.pow(circle.radius,2) * Math.PI;
  }
}
```

De manière générale, le patron de conception visiteur a la structure suivante :



2 Conclusion

Nous avons vu que 23 patrons de conceptions existent et nous n'en avons étudié que certains. Si ce domaine vous intéresse je vous recommande l'ouvrage de référence, écrit par les quatre concepteurs initiaux des patrons de conceptions : "Design patterns. Catalogue des modèles de conception réutilisables" par Gamma, Helm, Johnson et Vlissides chez Vuibert Informatique. Il faut connaître les patrons de conceptions pour pouvoir les utiliser dans des cas précis, tout en faisant attention de ne pas en abuser. Ils permettent de résoudre beaucoup de problèmes, mais ne sont pas toujours optimaux et peuvent parfois introduire une complexité excessive. De même, il peut être très difficile de respecter tous les principes SOLID. En fait, ces principes s'appliquent dans le cadre d'un code qui évolue. Par exemple, en ce qui concerne SRP, si votre code n'évolue pas d'une manière telle que deux responsabilités présentes au sein d'une même classe changent à des moments différents, il n'est pas nécessaire de les séparer. En effet, les séparer serait une source de complexité inutile. Il y a un corollaire à cela. Un axe de changement n'est un axe de changement que si les changements se produisent. Il n'est pas judicieux d'appliquer SRP, ou tout autre principe, d'ailleurs, s'il n'y a pas de symptôme.

Ce cours finit la partie sur la Conception Orientée Objet. Vous devez maintenant posséder l'essentiel des connaissances vous permettant de concevoir des logiciels avec cette approche. De même, en fonction de vos affinités ou de vos obligations, vous allez développer cette compétence dans un langage particulier. Dans le cadre de ce cours, on a utilisé exclusivement le langage Java, mais rien ne vous empêche d'utiliser d'autres langages permettant le paradigme de programmation orienté objet comme Python, C++, ...