Initiation génie logiciel : Notions Java et patrons de conception

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

8 octobre 2024

amU Faculté des sciences Aix Marseille Université

Section 1

Record en Java

Mot-clé `record` (Java 14)

Nouvelle façon de définir des classes en Java.

```
record Rectangle(double length, double width) { }
```

Crée une classe avec :

- des attributs (*field*) `private final` `name` et `age` ;
- des *getters* `name()` et `age()` ;
- des méthodes `equals` et `hascode` ;
- une méthode `toString`.

Pourquoi `record` ?

Pour définir une classe de données non mutables (réponses à des requêtes).

Équivalent à la classe suivante

```
public final class Rectangle {
    private final double length;
    private final double width;
    public Rectangle(double length, double width) {
        this.length = length; this.width = width;
    }
    double length() { return this.length; }
    double width()  { return this.width; }
    public boolean equals(Object o) {/* ... */}
    public int hashCode() {/* ... */}
    public String toString() {/* ... */}
}
```

Équivalent à la classe suivante

```java
public final class Rectangle {
  // suite du transparent précédent
  public boolean equals(Object obj) {
    if (obj == this) return true;
      if (obj == null || obj.getClass() != this.getClass())
        return false;
      var that = (Rectangle) obj;
      return this.length == that.length &&
             this.width == that.width;
  }
}
```

Équivalent à la classe suivante

```java
public final class Rectangle {
  // suite du transparent précédent
  public int hashCode() {
    return Objects.hash(length, width);
  }
  public String toString() {
    return "Rectangle[" +
           "length=" + length + ", " +
           "width=" + width + ']';
  }
}
```

Section 2

Interfaces (notions avancées)

Les classes anonymes

Supposons que nous ayons l'interface suivante :

```
Interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Il est possible de :

- définir une classe anonyme qui implémente cette interface
- d'obtenir immédiatement une instance de cette classe

```
ActionListener listener = new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    counter++;
  }
};
```

Les classes anonymes

```
public class Window {
  private int counter;
  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          counter++;
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des attributs de la classe "externe" :

```
public class Window {
  private Counter counter = new Counter();
  public Window() {
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
          counter.count();
      }
    });
  }
}
```

Les classes anonymes

Il est possible d'utiliser des variables finales de la méthode :

```
public class Window {
  public Window() {
    final Counter counter = new Counter();
    Button button = new Button("count");
    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
        counter.count();
      }
    });
  }
}
```

Lambda expressions (Java 8)

Avec Java 8, il est possible d'écrire directement :

```
public class Window {
  public Window() {
    Button button = new Button("button");
    button.addActionListener(event -> System.out.println(event));
  }
}
```

Explication : `ActionListener` a une seule méthode donc on peut affecter une lambda expression (décrivant cette méthode) à une variable de type `ActionListener`.

```
public interface ActionListener {
  public void actionPerformed(ActionEvent event);
}
```

Interfaces fonctionnelles (Java 8)

Une interface n'ayant qu'une méthode abstraite est une interface fonctionnelle. Les quatre interfaces fonctionnelles suivantes (et plein d'autres) sont déjà définies :

```
public interface Predicate<T> {
  public boolean test(T t);
}
public interface Function<T,R> {
  public R apply(T t);
}
public interface Consumer<T> {
  void accept(T t);
}
public interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Syntaxe d'une lambda expression

Pour instancier une interface fonctionnelle, on peut utiliser une lambda expression :

L'interface suivante :

```
public interface MyFunctionalInterface{
  public T myMethod(A arg1, B arg2, C arg2);
}
```

peut être instanciée par :

```
MyFunctionalInterface fonc =
  (arg1, arg2, arg2)
 ->/* expression définissant le résultat de myMethod */
```

Si `T` est `void` alors l'expression peut être `void` comme un `println`.

Exemples de lambda expression

On considère une classe `Person` avec deux attributs `name` et `age` et les *getters* et *setters* associés.

On a le droit d'écrire les lambda expressions suivantes en Java :

- `person -> person.getAge() >= 18` de type `Predicate<Person>`
- `person -> person.getName()` de type `Function<Person,String>`
- `name -> System.out.println(name)` de type `Consumer<Person>`

Remarques

- Il n'est pas nécessaire de mettre le type des paramètres.
- On peut omettre les parenthèses dans le cas où il n'y a qu'un seul paramètre

Référence de méthodes

Dans un certain nombre de cas, une lambda expression se contente d'appeler une méthode ou un constructeur.

Il est plus clair dans ce cas de se référer directement à la méthode ou au constructeur.

Lambda expression	référence de méthode
`x -> Math.sqrt(x)`	`Math::sqrt`
`name -> System.out.println(name)`	`System.out::println`
`person -> person.getName()`	`Person::getName`
`name -> new Person(name)`	`Person::new`

java.util.stream.Stream

`Stream` = Abstraction d'un flux d'éléments sur lequel on veut faire des calculs

Ce n'est pas une `Collection` d'élément, car un `Stream` ne contient pas d'élément

Création d'un Stream :

- À partir d'une collection comme une liste avec `list.stream()`
- À partir d'un fichier : `Files.lines(Path path)`
- À partir d'un intervalle : `IntStream.range(int start, int end)`

Exemples d'utilisation

```java
public record Person(String name, int age) {}

persons.stream()
       .filter(person -> person.age() >= 18)
       .map(Person::name)
       .forEach(System.out::println);
```

Types des paramètres et retours des méthodes :

- `stream()` → `Stream<Person>`
- `filter(Predicate<Person>)` → `Stream<Person>`
- `map(Function<Person, String>)` → `Stream<String>`
- `forEach(Consumer<String>)` → `void`

Cycle de vie d'un `Stream`

Un `Stream` est toujours utilisé en trois phases :

- Création du Stream (à partir d'une collection, d'un fichier, ...),
- Opérations intermédiaires sur le `Stream` (suppression d'éléments, transformation de chaque élément, combinaison) qui prenne un `Stream` et renvoie un `Stream`,
- Une seule opération terminale du `Stream` (calcul de la somme, de la moyenne, application d'une fonction sans retour sur chaque élément, ...).

Opérations intermédiaires possibles sur un `Stream`

- `Stream<E> filter(Predicate<? super E>)` : sélectionne si un élément reste dans le `Stream`
- `<R> Stream<R> map(Function<? super E, ? extends R)` : transforme les éléments du `Stream` en leur appliquant une fonction
- `Stream<E> sorted(Comparator<? super E>)` : trie les éléments

Opérations terminales possibles sur un `Stream`

- `long count()` : compte le nombre d'éléments
- `long sum()` et `double sum()` : somme les éléments (entiers ou double)
- `Stream<E> forEach(Consumer<? super E>)` : Appelle le `consumer` pour chaque élément
- `allMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour tous les éléments
- `anyMatch(Predicate<? super E>)` : vrai si le prédicat est vrai pour au moins un élément
- `toList()` : crée une liste avec les éléments du `Stream`

Section 3

Patron de conception *Proxy*

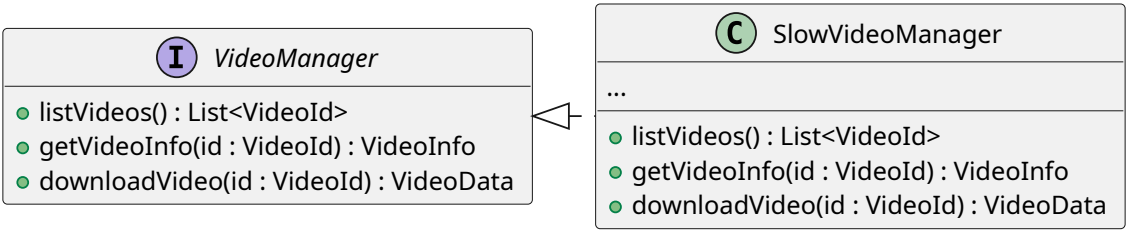
Service de vidéos

On suppose qu'on a accès à une classe permettant d'accéder à des vidéos.

Problème : le service est lent et on aimerait mettre en cache si possible les réponses aux requêtes.

Introduction d'une interface

On commence par introduire une interface pour le service de vidéo :



On va rajouter une classe de procuration implémentant cette interface et déléguant le téléchargement à l'outil de téléchargement original.

La classe de procuration garde la trace de tous les fichiers téléchargés et retourne les données du cache lorsque l'application fait plusieurs fois appel à la même vidéo.

Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  private SlowVideoManager service;
  private Map<VideoID,VideoData> videoDataCache;
  private Map<VideoID,VideoInfo> videoInfoCache;
  private List<VideoId> listeCache;

  public CachedVideoManager(){
    service = new SlowVideoManager(/* ... */);
    videoDataCache = new HashMap<>();
    videoInfoCache = new HashMap<>();
  }
}
```

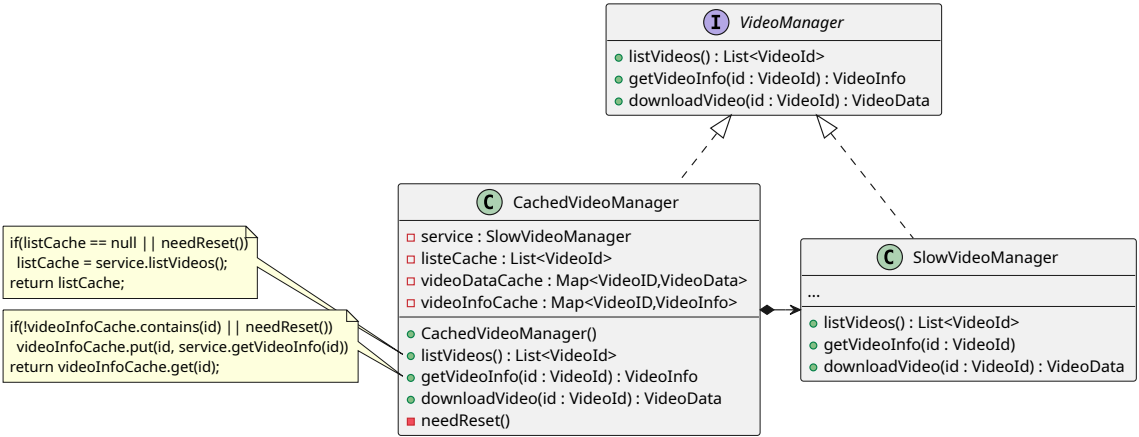
Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  private boolean needReset(){
    /* test if a reset is needed */
  }
  public List<VideoId> listVideos(){
    if(listCache == null || needReset())
      listCache = service.listVideos();
    return listCache;
  }
}
```

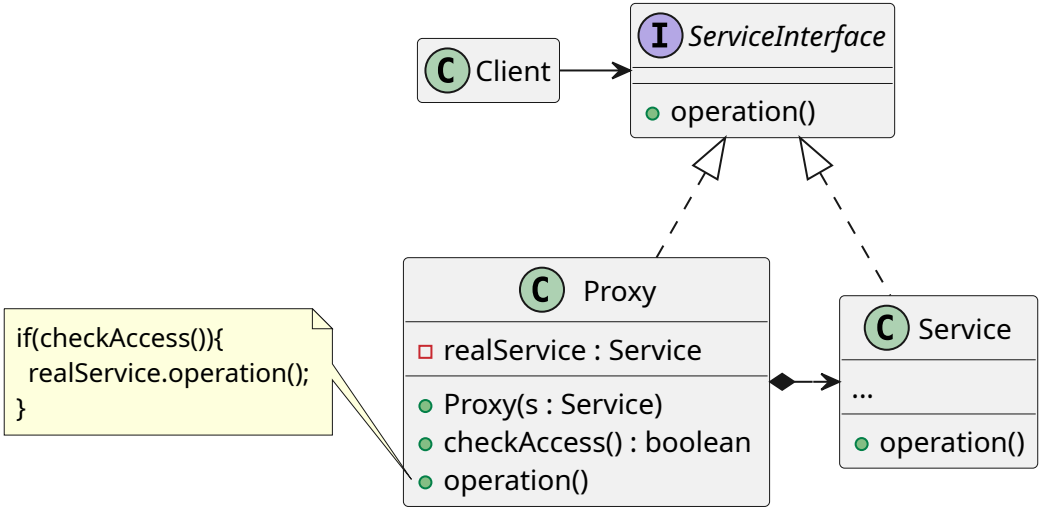
Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  public VideoInfo getVideoInfo(VideoId id){
    if(!videoInfoCache.contains(id) || needReset())
      videoInfoCache.put(id, service.getVideoInfo(id))
    return videoInfoCache.get(id);
  }
  public VideoData getVideoData(VideoId id){
    if(!videoDataCache.contains(id) || needReset())
      videoDataCache.put(id, service.getDataInfo(id))
    return videoDataCache.get(id);
  }
}
```

Classe *proxy* CachedVideoManager



Patron de conception *proxy*



Patron de conception *proxy*

Intention

Permet d'utiliser un substitut pour un objet donnant le contrôle sur l'objet original.

Avantages :

- Contrôler l'objet du service sans que le client ne s'en aperçoive.
- Ajout possible de procuration sans toucher au service (OCP)
- Gestion du cycle de vie de l'objet du service originel

Désavantages :

- Code plus complexe, car introduction de classes supplémentaires
- Intermédiaire pouvant ralentir les requêtes.

Section 4

Patron de conception *Decorator*

Pile

Supposons que nous avons la classe suivante :

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

  public void push(int value) {
    elements[size] = value;
    size++;
  }
  public int pop() {
    size--; elements[size]= null;
    return elements[size];
  }
}
```

Pile avec *log*

Nous souhaitons ajouter des logs (`print`) pour débugger notre programme :

```
public class ArrayStack {
  private int[] elements = new int[10];
  private int size = 0;
  public void push(int value) {
    System.out.println("push("+value+")");
    elements[size] = value; size++;
  }
  public int pop() {
    System.out.println("pop()");
    size--; elements[size]= null;
    return elements[size];
  }
}
```

Modification de Stack

Cette modification a été réalisée en modifiant une classe existante (violation OCP). De plus, une nouvelle modification est nécessaire pour retirer les logs.

Définissons l'interface suivante :

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

Faisons en sorte que `ArrayStack` implémente cette interface :

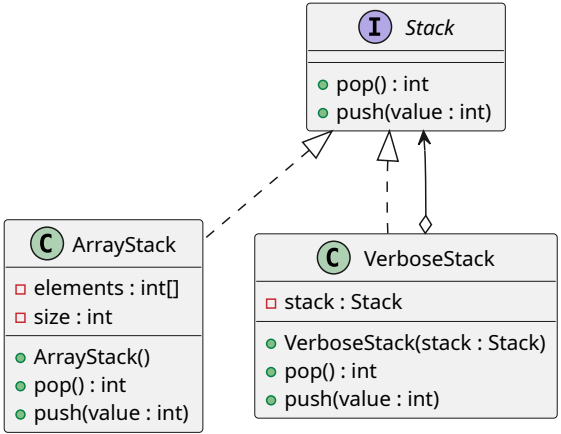
```
public class ArrayStack implements Stack {
  /* ... */
}
```

Classe VerboseStack décorant Stack

Il suffit alors de définir une classe *decorator* `VerboseStack` implémentant `Stack` :

```
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();
  }
}
```

Principe de VerboseStack



`VerboseStack` :

- implémente `Stack`
- délègue une partie du comportement des méthodes de `Stack` à une autre instance implémentant `Stack`
- rajoute un comportement (*print*) aux méthodes de `Stack` (décoration)

Exemple d'utilisation

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()
```

Nouvelle classe de décoration

Nous définissons un nouveau décorateur :

```
public class ParenthesesStack implements Stack {
  private Stack stack;
  private String parentheses;
  public VerboseStack(Stack stack) {
    this.stack = stack; parentheses = "";
  }
  public void push(int value) {
    parentheses += "(";
    stack.push(value);
  }
  public int pop() { parentheses += ")"; return stack.pop(); }
  public String getParentheses() { return parentheses; }
}
```

Exemple d'utilisation

Un exemple d'utilisateur du décorateur précédent :

```
Stack stack = new ArrayStack(10);
Stack parenthesesStack = new ParenthesesStack(stack);
stack.push(2); stack.push(3); stack.pop();
stack.push(1); stack.pop(); stack.pop();
System.out.println(parenthesesStack.getParentheses());
```

Ce code produit la sortie suivante :

```
(()())
```

Plusieurs décorateurs

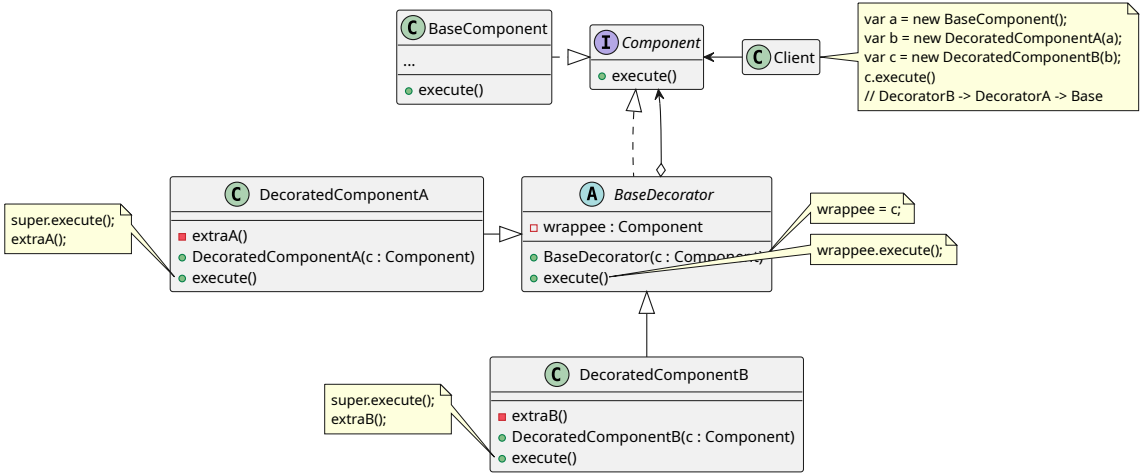
Il est possible d'utiliser plusieurs décorateurs simultanément :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
Stack parenthesesStack = new ParenthesesStack(stack);
stack = parenthesesStack;
stack.push(2); stack.push(3); stack.pop();
System.out.println(parenthesesStack.getParentheses());
```

Ce code produit la sortie suivante :

```
push(2)
push(3)
pop()
(()
```

Patron de conception *decorator*



Patron de conception *decorator*

Intention

Permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.

Avantages :

- Séparer le code d'une classe monolithique en plusieurs classes (SRP).
- Gestion dynamique (à l'exécution) des comportements des objets.
- Permet de chainer des modifications (décorations) de comportement.

Désavantages :

- Retirer un emballeur spécifique de la pile n'est pas chose aisée.
- Position dans le chaînage influant le comportement.
- Code de mise en place un peu complexe.