Pour aller plus loin : qualité du modèle UML et patrons de conception

Denis Conan

CSC4102

Télécom SudParis

Janvier 2025

Table des matières

Pour aller plus loin : qualité du modèle UML et patrons de conception
Denis Conan, , Télécom SudParis, CSC4102
Janvier 2025 **1**

Sommaire **3**

1 Création, patron de conception Singleton **3**

2 Structure, patron de conception Composite **5**

3 Comportement, Observable—Observateur **6**

4 Comportement, patron Stratégie **7**

5 Combinaison de patrons de conception, exemple de MVC **10**

Bibliographie **12**

2

Sommaire

1 Création, patron de conception Singleton 4
2 Structure, patron de conception Composite 6
3 Comportement, Observable—Observateur 10
4 Comportement, patron Stratégie 16
5 Combinaison de patrons de conception, exemple de MVC 17

Dans les sections qui suivent, nous complétons les diapositives du cours avec quelques patrons de conception supplémentaires.

Nous ne pouvons pas présenter tous les patrons de conception dans les diapositives du cours, qui ne dure qu'une heure. Aussi, selon le ou les patrons de conception insérés dans le logiciel de l'étude de cas, nous déplaçons dans ce document quelques patrons de conception supplémentaires.

3

1 Création, patron de conception Singleton

1. Problème : dans l'étude de cas Médiathèque, pour la classe de la Façade,
   - Comment assurer qu'une classe n'a qu'une instance ? facilement accessible ?
2. Solution
   - Une variable globale permet d'accéder à un objet, mais il est plus difficile d'ajouter la contrainte de l'instance unique
   - ✓ Une meilleure solution consiste à confier à la classe elle-même la responsabilité d'assurer l'unicité[a]

Médiathèque
–nom:string
–instance:Médiathèque
–Médiathèque(String nom)
+getInstance(String nom):Médiathèque

```
if (instance == null) {
instance=new Médiathèque(nom);
}
return instance;
```

3. Conséquences
   - La méthode `getInstance` définit (l'algorithme de) la création
   - Difficulté avec le concept de généralisation spécialisation : dérivation de la classe

a. Cf. classes du paquetage `seance8.patrondeconception.singleton`

4

singleton

2 Structure, patron de conception Composite

5

1. Problème
   - Comment gérer une <u>arborescence</u> d'objets, tous gérés de manière similaire ?
     - Arborescence d'objets = relations de composition entre certains objets
     - Par exemple, dans un système à base d'événements qui filtre des publications (contenant des informations),
       comment gérer de manière identique des fitres simples et des filtres composites
       - Filtres simples F_{s_i} : méthode $\text{evaluate}_{F_{s_i}} \rightarrow \{true, false\}$
       - Filtres composites F_{c_i} : méthode $\text{evaluate}_{F_{c_i}}$ définie comme une formule logique de filtres simples
         avec les connecteurs de la logique propositionnelle ($\neg$, $\wedge$, $\vee$, etc.)

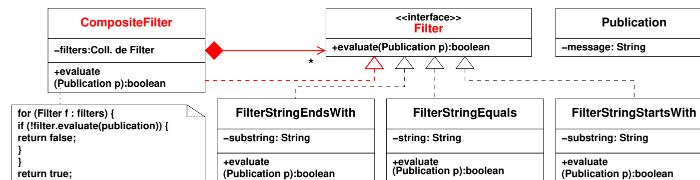
6

2. Solution :

   Soit des classes d'objets primitifs (atomiques) et des classes d'objets conteneurs

   Problème : gestion différenciée des primitifs et des conteneurs

   ✓ Soit une classe abstraite ou une interface, dont certaines concrétisations mettent en œuvre l'agrégation ou la composition d'éléments
   (dans la figure, le connecteur de la composition $= \wedge$)



3. Conséquences :
   - Adjonction de nouveaux types d'éléments par simple ajout de classes concrètes

Cf. classes du paquetage `seance8.patrondeconception.composite`

3 Comportement, Observable—Observateur

\# 7

1. Problème

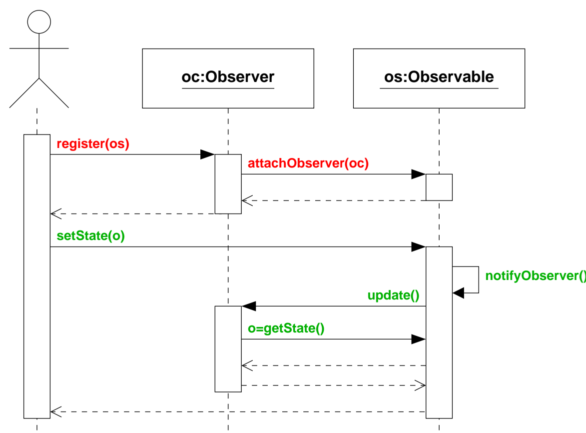
- Comment lier des objets sans fortement les coupler ? tout en maintenant la cohérence ?
  - Maintien de la cohérence = modification de O_{able} $\Longrightarrow$ O_{teur} averti
- Comment avoir un objet O_{able} avec plusieurs objets O_{teur_i} ?

- Par exemple, notification d'acteurs

\# 8

2. Solution

\# 9



Cf. classes du paquetage `seance8.patrondeconception.observateurobservable`

\# 10

3. Conséquences :
   - L'objet observateur est un objet référencé par l'objet observable
     - Par exemple, la façade de la médiathèque possède la connaissance (de l'interface graphique) des acteurs
     - Similairement à l'utilisation de la façade comme unique point d'entrée, la façade peut devenir l'unique point de sortie

En conclusion, vous remarquez le patron de conception Observable—Observateur est très proche du patron de conception Publier—Souscrire. La principale différence est, par exemple, visible dans la mise en œuvre en JAVA : l'observable indique aux observateurs que son état a changé et chacun d'eux fait un appel pour récupérer une nouvelle observation, alors que le producteur notifie une publication aux consomateurs, la publication contenant la nouvelle observation. Pour plus de détails, lisez par exemple la comparaison qui est donnée aux deux URLs suivantes :
`https://stackoverflow.com/questions/46380073/observer-is-deprecated-in-java-9-what-should-we-use-instead-of-it`
et
`https://stackoverflow.com/questions/11619680/why-should-the-observer-pattern-be-deprecated/11632412#11632412`.
Imanquablement, votre étude vous amène au canevas logiciel qui a irrigué toute cette réflexion et fait aujourd'hui préférer le patron de conception Publier—Souscrire : ce sont les « *Reactive Streams* » :
`https://www.reactive-streams.org/`.

11

4 Comportement, patron Stratégie

1. Problème
   - Comment modéliser une famille d'algorithmes ?
     en les rendant interchangeables ?
2. Solution

   Soit insérer les algorithmes dans les objets qui les utilisent

   Quid d'un changement de stratégie (d'algorithme) ?

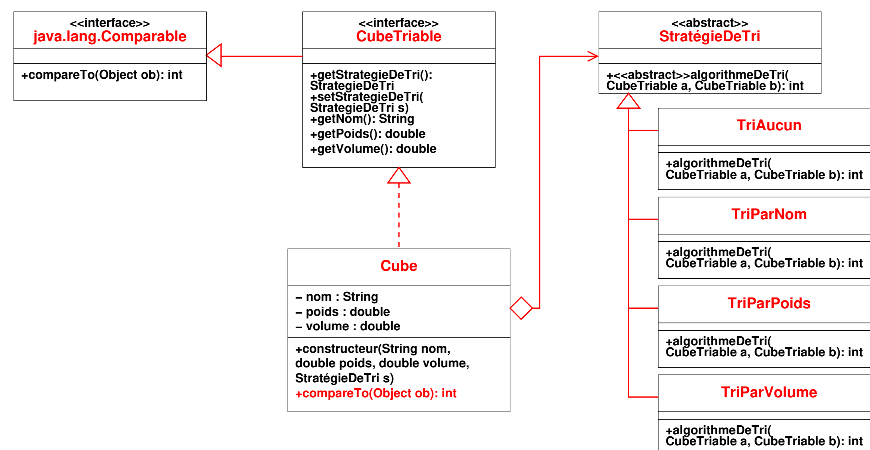
   PB : Paramètre de configuration
   avec une structure de contrôle à base de `if` ou de `switch` ?
   - Complexe $\Longrightarrow$ sujet à erreurs (dans la conception et dans les tests)

   PB : Utilisation d'une généralisation spécialisation avec changement de type pour changer de stratégie ?
   - Complexe $\Longrightarrow$ sujet à erreurs (dans la conception et dans les tests)

   ✓ Soit encapsuler les algorithmes dans des objets distincts

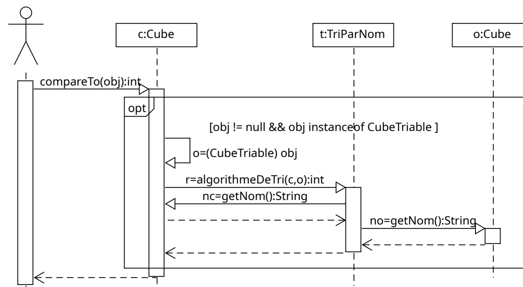
   ✓ Changement de stratégie = changement d'objet

12



\# 13

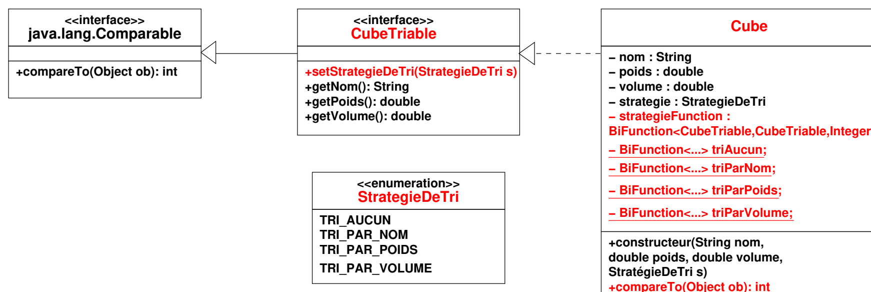
Cf. classes du paquetage `seance8.patrondeconception.strategie`



3. Conséquences :
   - Appel des algorithmes par délégation (`c` délègue à `t`)
   - Possibilité de factoriser des parties d'algorithmes par généralisation spéc.
   - Possibilité d'ajouter le patron Singleton pour avoir une instance par stratégie

\# 14

- Avec des *lambda expressions*



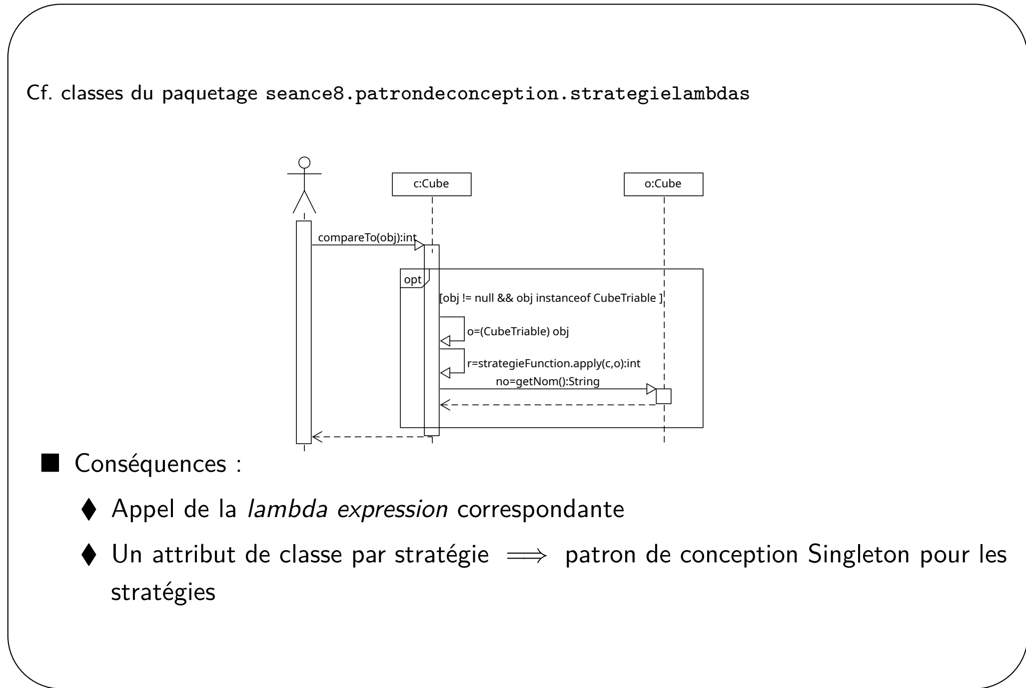
15

Classe seance8.patrondeconception.strategielambdas

```
1         private double volume;
  private StrategieDeTri strategie;
                  volume = v;
    strategie = s;
5   strategieFunction = switch (s) {
      case TRI_AUCUN -> triAucun;
      case TRI_PAR_NOM -> triParNom;
      case TRI_PAR_POIDS -> triParPoids;
9     case TRI_PAR_VOLUME -> triParVolume;
      default -> null;
    };
  }
13 //...
  private static BiFunction<CubeTriable, CubeTriable, Integer> triAucun =
                   (a, b) -> 0;
  private static BiFunction<CubeTriable, CubeTriable, Integer> triParNom =
17                 (a, b) -> a.getNom().compareTo(b.getNom());
  private static BiFunction<CubeTriable, CubeTriable, Integer> triParPoids =
                   (a, b) -> Double.compare(a.getPoids(), b.getPoids());
  private static BiFunction<CubeTriable, CubeTriable, Integer> triParVolume =
21                 (a, b) -> Double.compare(a.getVolume(), b.getVolume());
```

16

Cf. classes du paquetage seance8.patrondeconception.strategielambdas



- Conséquences :
  - Appel de la *lambda expression* correspondante
  - Un attribut de classe par stratégie $\Longrightarrow$ patron de conception Singleton pour les stratégies

5 Combinaison de patrons de conception, exemple de MVC

\# 17

- Le patron Modèle—Vue—Contrôleur (MVC) est souvent mis en œuvre en utilisant une combinaison des patrons Observable—Observateur, Stratégie et Composite [Krasner and Pope, 1988]
  - Stratégie : relation entre les contrôleurs et les vues
  - Composite : relation entre les vues
  - Observable—Observateur : relation entre le modèle et les vues, et entre le modèle et les contrôleurs

18

Bibliographie

[Krasner and Pope, 1988] Krasner, G. and Pope, S. (1988). A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *Journal Object Oriented Programming*, 1(3) :26–49.