Design patterns et architectures logicielles

Étude de cas : le design pattern *Strategy*

Filière : SUD-Cloud & IoT

Travail réalisé par :

- Chrif El Asri Hanane
- El Gamous Khalid

Encadrant:

Prof. Abdeslam EN-NOUAARY

2020/2021

Introduction

Stratégie est un patron de conception comportemental qui permet de définir une famille d'algorithmes, de les mettre dans des classes séparées et de rendre leurs objets interchangeables. Lors de cette étude de cas on va comprendre comment il fonctionne.

I. Problématique

Un beau jour, on a décidé de créer une application qui permet à un client de trier un tableau d'entiers. La première version de l'application utilisait le tri à bulle. Ensuite on a décidé d'intégrer autres méthodes de tri et d'offrir à l'utilisateur la possibilité de choisir la méthode qui le convient. La deuxième version contient donc le tri rapide. On ajoute les autres types successivement.

L'application a beau avoir très bien marché d'un point de vue financier, on a arraché les cheveux sur le côté technique. Chaque fois qu'on ajoute un nouvel algorithme, la classe principale doublait de taille. À un moment donné, la bête n'était plus possible à maintenir. La moindre touche apportée aux algorithmes impactait la totalité de la classe, augmentant les chances de créer des bugs dans du code qui fonctionnait très bien.

De plus, travailler en équipe n'était plus efficace. Les membres de l'équipe embauchés juste après la sortie et le succès de l'application se plaignaient de passer trop de temps à résoudre des problèmes de fusion. Ajouter une nouvelle fonctionnalité demandait de modifier une classe énorme, créant des conflits dans le code produit par les autres développeurs.

II. Solution avec Strategy

Le patron de conception stratégie vous propose de prendre une classe dotée d'un comportement spécifique mais qui l'exécute de différentes façons, et de décomposer ses algorithmes en classes séparées appelées stratégies.

La classe originale (le contexte) doit avoir un attribut qui garde une référence vers une des stratégies. Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.

Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie. En fait, le contexte n'y connait pas grand-chose en stratégies, c'est l'interface générique qui lui permet de les utiliser. Elle n'expose qu'une seule méthode pour déclencher l'algorithme encapsulé à l'intérieur de la stratégie sélectionnée.

Le contexte devient indépendant des stratégies concrètes. On peut ainsi modifier des algorithmes ou en ajouter de nouveaux sans toucher au code du contexte ou aux autres stratégies.

III. Etude de cas

Dans notre cas, on a créé une classe **Contexte** qui garde une référence vers une des stratégies concrètes et communique avec cet objet uniquement au travers de l'**interface stratégie.** Cette dernière est commune à toutes les stratégies concrètes. Elle déclare une méthode que le contexte utilise pour exécuter une stratégie.

Les **Stratégies Concrètes** implémentent différentes variantes d'algorithmes utilisées par le contexte. On a jusqu'à présent 4 Stratégies Concrètes ; **InsertionSort**, **BubbleSort**, **MergeSort** et **QuickSort**.

Chaque fois qu'il veut lancer un algorithme, le contexte appelle la méthode d'exécution de l'objet stratégie associé. Le contexte ne sait pas comment la stratégie fonctionne ni comment l'algorithme est lancé.

On a également une classe Test qui représente le **Client**. Il crée un objet spécifique Stratégie et le passe au contexte. Le contexte expose un setter qui permet aux clients de remplacer la stratégie associée au contexte lors de l'exécution.

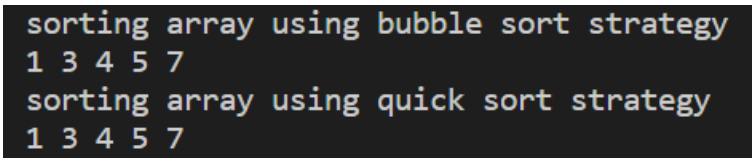
On a complété le code donnée dans cette étude de cas pour lui intégrer les méthodes de tris pour avoir une application fonctionnant bien.

Pour tester, on crée une liste contenant des entiers non triée. On fait appel ensuite aux méthodes de tri BubbleSort et ensuite QuickSort.

```
// we can provide any strategy to do the sorting
int[] array = {1, 4, 3, 7, 5 };
Context ctx = new Context(new BubbleSort());
ctx.arrange(array);

System.out.println("");
// we can change the strategy without changing Context class
ctx = new Context(new QuickSort());
ctx.arrange(array);
```

Voici le résultat :



Conclusion

On constate qu'il est préférable d'utiliser le patron de conception stratégie si on veut avoir différentes variantes d'un algorithme à l'intérieur d'un objet à disposition, et pouvoir passer d'un algorithme à l'autre lors de l'exécution, si on a beaucoup de classes dont la seule différence est leur façon d'exécuter un comportement ou encore si la classe possède un gros bloc conditionnel qui choisit entre différentes variantes du même algorithme.

Il est aussi utile pour isoler la logique métier d'une classe, de l'implémentation des algorithmes dont les détails ne sont pas forcément importants pour le contexte.