Patrons de conception : *Iterator*

Théorie et pratique de la programmation
Michel Schinz – 2013-03-25

Patrons de conception

Problèmes récurrents

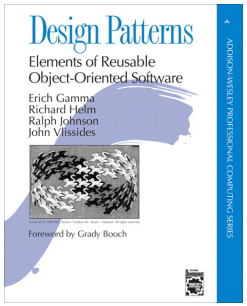
En programmation, comme dans toute discipline, certains problèmes sont récurrents. Un programmeur expérimenté sait identifier de tels problèmes, et connaît généralement leur solution.

Pourquoi ne pas répertorier ces problèmes et leur solution, afin de faciliter le travail des programmeurs ?

Patron de conception

Un **patron de conception** (*design pattern*) est une solution à un problème de conception récurrent.

Un tel modèle est nommé et décrit en détail dans un répertoire de modèles – p.ex. le célèbre livre *Design Patterns, Elements of Reusable Object-Oriented Software* de Gamma, Helm, Johnson et Vlissides.



Exemple : itérateur

Problème récurrent : un objet possède une collection de valeurs et désire y donner accès, sans révéler la manière dont cette collection est représentée en interne.
Solution (modèle de conception *Iterator*) : fournir un itérateur, à savoir un objet qui permet d'examiner les valeurs les unes après les autres, dans un ordre donné.

Composants d'un patron

Les principaux composants d'un patron sont :

- son **nom**,
- une description du **problème** résolu,
- une description de la **solution** à ce problème,
- une présentation des **conséquences** liées à l'utilisation du patron.

Avantages des patrons

Les patrons de conception permettent de diffuser largement les meilleures solutions connues à différents problèmes récurrents.
De plus, ces solutions sont nommées, ce qui permet de raisonner et de communiquer à un plus haut niveau d'abstraction que lorsqu'on se concentre sur les détails de mise en œuvre.

Inconvénients des patrons

Malgré la récente excitation liée à leur « découverte », les patrons de conception ne sont pas une panacée. Une utilisation systématique des motifs ne saurait garantir qu'un programme soit bien conçu.
Il est donc important de n'utiliser un patron que lorsque cela est justifié, et que les avantages liés à son utilisation compensent les inconvénients – p.ex. l'augmentation de la complexité du programme qui en résulte fréquemment.

Patrons et langages

Un patron de conception n'est normalement pas lié à un langage de programmation donné.
Toutefois, beaucoup de patrons ont été inventés dans le contexte de langages orienté-objets. Ils font une utilisation intensive des concepts de ce type de langages – classes, objets, polymorphisme d'inclusion, etc. – et sont donc difficilement utilisables dans d'autres contextes.
Il faut aussi noter que certains langages possèdent des concepts qui rendent modèles de conception obsolètes ! Par exemple, le langage Scala possède le filtrage de motifs (*pattern matching*) qui rend l'utilisation du modèle *Visitor* inutile.

Patron n°1: *Iterator* (ou *Cursor*)

Illustration du problème

Une classe représentant une liste d'éléments doit fournir un moyen de parcourir ces éléments les uns après les autres.
Une possibilité serait d'exposer la représentation interne au client – p.ex. en rendant la classe des nœuds visible dans le cas des listes chaînées. Mais on violerait alors l'encapsulation.
Comment faire ?

Solution

Pour permettre le parcours des éléments d'une liste sans exposer sa représentation interne, on peut utiliser un objet qui désigne à tout moment un élément de la liste. Cet objet possède des opérations permettant d'obtenir l'élément désigné et de passer à l'élément suivant, voire au précédent.
Un tel objet s'appelle un **itérateur**, ou un **curseur**.

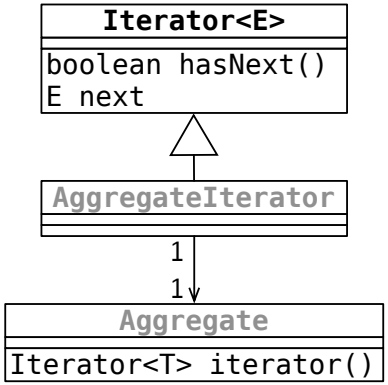
Généralisation

Le concept d'itérateur peut être utilisé chaque fois qu'un objet possède une collection d'éléments qu'un client doit pouvoir parcourir sans connaître la représentation interne de la collection.

Exemples : parcours du contenu d'un répertoire dans un système de fichiers, parcours des résultats d'une requête à une base de données, etc.

Diagramme de classes

Le diagramme de classes ci-dessous illustre les classes et interfaces impliquées dans l'ajout d'une notion d'itérateur à une classe imaginaire `Aggregate`, contenant une collection d'éléments.



Raffinements

Les itérateurs présentés ici sont très simples. Selon les besoins, on peut imaginer les augmenter avec des méthodes permettant de :

- passer à l'élément précédent,
- aller au début ou à la fin de la collection,
- supprimer ou ajouter des éléments à la position désignée par l'itérateur,
- etc.

Type de parcours

Les itérateurs définis sur les listes jusqu'à présent effectuent un parcours du premier au dernier élément.

On peut imaginer d'autres sortes de parcours, p.ex. en sens inverse. Pour des structures de données plus complexes – p.ex. les arbres – les possibilités sont plus nombreuses : en largeur ou profondeur d'abord, en pré-ordre, post-ordre, etc.

Chacun de ces types de parcours peut être mis en œuvre par un itérateur différent.

Intérêt du patron *Iterator*

Un itérateur permet de parcourir les éléments d'une collection en faisant abstraction de sa représentation. Ainsi, on peut écrire exactement le même code pour parcourir les éléments d'un ensemble représenté par un arbre de recherche que ceux d'une liste représentée par chaînage.
Il est même possible de définir des itérateurs sur des structures infinies, p.ex. la liste des nombres premiers !

Exemples réels

La bibliothèque Java utilise des itérateurs – représentés par l'interface `java.util.Iterator` – pour permettre le parcours des collections : listes, ensembles, etc.
La bibliothèque STL de C++ fait de même.

Résumé

Un patron de conception est une solution, nommée et documentée, à un problème de conception récurrent. Un bon programmeur se doit de connaître un certain nombre de patrons importants et de savoir quand les utiliser.
Le premier patron examiné, *Iterator,* permet à un objet – généralement une collection – de donner accès à ses éléments sans révéler la manière dont ils sont stockés en interne.