Sokoban : étape 3

Dans cette étape, nous allons travailler sur le parcours d'un ensemble d'éléments à l'aide d'un *design pattern* nommé *itérateur*. Nous appliquerons les exercices proposés aux séquences que nous avons développées jusqu'à maintenant, mais l'*itérateur* est général et s'applique à tout ensemble d'éléments. Dans un premier temps, nous allons discuter de la motivation présente derrière l'*itérateur*. Pour cela, nous supposerons que nous ne connaissons rien de l'implémentation de notre structure de données mais que nous avons un moyen de désigner chaque élément : par exemple, un indice correspondant à un ordre total sur les éléments, soit arbitraire, soit induit par la structure de données elle-même. Nous disposons aussi des opérations `supprime`, `lis` et `ajoute` qui, étant donné un indice, appliquent l'opération indiquée par le nom de l'opérateur à l'élément d'indice donné.

Lorsqu'un algorithme consiste, par exemple, à parcourir une seule fois les éléments de notre structure de données et à décider de les supprimer, de les modifier ou d'insérer de nouveaux éléments à la position courante, les opérations `supprime`, `lis` et `ajoute` définies précédemment ont plusieurs défauts :

- Elles reparcourent parfois la structure depuis le début (pas de position courante stockée dans la structure elle-même), ce qui peut entrainer des parcours redondants.
- Elles ne sont pas pratiques car les suppressions et les ajouts peuvent changer les numéros d'indice des éléments : l'implémentation du parcours est donc plus difficile.

Par exemple, pour supprimer les éléments nuls d'une séquence d'entiers :

```
i=0;
while (i<maSequence.taille()) {
    // On reparcourt la séquence à l'intérieur de lis si elle est implémentée par une liste
    int n = maSequence.lis(i);

    // Si on supprime l'élément i, un autre le remplace, les numéros des éléments changent :
    // la boucle est un peu dure à écrire, pas d'incrémentation de i quand on supprime
    if (n == 0)
        // On reparcourt la séquence a l'intérieur de supprime si elle est implémentée par une liste
        maSequence.supprime(i);
    else
        i++;
}
```

Pour ne pas subir ces désagréments, nous pourrions envisager d'écrire le parcours directement dans la structure de données, mais :

- Cela briserait l'encapsulation de notre implémentation. Dans notre cas, nous avons deux implémentations des séquences, très différentes l'une de l'autre, et briser l'encapsulation nous obligerait à gérer et maintenir ces deux versions de parcours ;
- Cela oblige à maintenir une information sur la position courante dans la boucle de parcours qui est une information plus complexe qu'un simple indice (élément courant et élément précédent dans le cas de la liste par exemple). Le code du parcours devient donc plus complexe.

Une meilleure solution est de créer une nouvelle classe dans le paquetage de gestion des listes qui implémente la notion de **position dans la séquence** et les opérations associées (déplacement et insertion/suppression à la position courante). Ce genre de structure intermédiaire, utilisée pour le parcours de collections d'objets, s'appelle un itérateur, il s'agit d'un concept bien connu en *design patterns* (voir le livre d'E. Gamma, R. Helm, R. Johnson et J. Vlissides donné en référence sur la page de l'UE). C'est l'itérateur qui s'occupe de maintenir l'information de position courante dans la séquence et qui modifie le chaînage lorsque c'est nécessaire.

Nom : itérateur
Autres noms : curseur
Rôle : fournit un moyen d'accéder séquentiellement aux éléments d'un aggrégat d'objets sans exposer sa représentation sous-jacente.

Motivation : un aggrégat d'objets doit fournir un moyen d'accéder à ses éléments sans exposer sa structure interne. L'*itérateur* permet de faire cela. L'idée clé de ce *pattern* est de placer la responsabilité des accès et du parcours dans un objet *itérateur*, responsable du suivi de l'élément courant.

Dans le cas d'un aggrégat d'éléments ordonnés, l'itérateur correspond à la notion de position qui était matérialisée par un indice entier dans le code précédent, sur un schéma, cela donne :

```
                           Elément 0   Elément 1   Elément 2   ...   Elément n-1
positions de l'itérateur : ^           ^           ^           ^     ^             ^
```

L'itérateur fournit un moyen de changer de position, d'ajouter un élément à sa position courante ou de supprimer un élément à gauche de sa position courante.

En utilisant un itérateur, le code précédent devient :

```
Iterateur it = maSequence.iterateur();
while (it.aProchain()) {
    int n = it.prochain();
    if (n == 0)
        it.supprime();
}
```

Les avantages par rapport à la première version sont que :

- l'implémentation peut être efficace : le parcours de la liste pour supprimer ou obtenir le suivant n'est pas nécessaire puisque l'itérateur stocke (dans ses attributs) la position courante
- la gestion de l'indice courant qui varie selon le cas (suppression ou non) a disparu : le code est plus simple
- nous n'avons plus à gérer les détails concernant diverses implémentations, ils sont gérés une fois pour toutes dans l'itérateur, on a gagné un niveau d'abstraction.

Pour avoir la description (en anglais) des méthodes de l'interface `Iterateur` qui correspond au code ci-dessus, consultez ce lien.

Exercices

- Écrivez une interface commune à vos deux implémentations de séquence et factotisez vos programmes de test en un seul programme permettant de tester les deux types de séquences ;
- Dotez vos séquences d'itérateurs dotés eux-mêmes des méthodes suivantes :
  - `boolean aProchain();`
    renvoie vrai si l'itérateur n'est pas à la dernière position.
  - `int prochain();`
    fait avancer l'itérateur à la prochaine position et renvoie la valeur de l'élément traversé (situé juste avant l'itérateur, une fois le déplacement effectué).
  - `void supprime();`
    supprime le dernier élément traversé via `prochain`. Ne peut être appelé qu'une seule fois après un appel à `prochain`, dans le cas contraire, lève une exception de type `IllegalStateException`.

  REMARQUE:

  vous avez du comprendre qu'un itérateur, qui est ici une position dans une séquence, ne peut se définir qu'à partir d'une séquence existante. En outre, il faut stocker dans l'itérateur, en plus d'une position courante, une référence à la séquence, nécessaire pour modifier ses attributs lors de certains ajouts ou suppressions. Nous pourrions alors penser créer l'itérateur de la manière suivante :

  ```
  Iterateur it;
  it = new IterateurSequenceListe(l);
  ```

  Ce n'est pas une bonne manière de faire :
  - si `l` est une référence nulle, la construction de l'itérateur échouera ;

    - le type précis de l'itérateur à créer est exposé ici car il dépend du type de séquence utilisé. En effet, l'itérateur manipule directement la structure de données de la séquence et doit donc être spécifiquement écrit pour chaque type de séquence.

  Pour éviter de devoir gérer ce genre de problèmes, la création d'un nouvel itérateur est donc confiée à une méthode de la séquence elle-même, que nous nommerons `iterateur`. Si la méthode est appelée, c'est que la séquence existe et a une référence non nulle et le problème de référence nulle est résolu. De plus, dans l'implémentation de la séquence, le type d'itérateur adapté est connu. Enfin, nous gagnons en abstraction : toute séquence peut fournir une référence d'itérateur sur elle-même se conformant à une interface d'iterateur, le type d'itérateur correspondant n'a pas besoin d'être connu par l'utilisateur.

  En termes d'implémentation, il faut juste que la séquence passe une référence à elle-même lors de la construction de l'itérateur en utilisant `this`. Par exemple, dans le cas de l'implémentation par liste, cela peut se faire de la manière suivante :

```
public Iterateur iterateur() {
    return new IterateurSequenceListe(this);
}
```

- Testez votre implémentation.