TD de Programmation autour de deux *design patterns* : itérateurs et visiteurs

1 - Itérateurs

Dans cette première partie, nous supposons que nous disposons d'une structure de données correspondant à un ensemble ou un multi-ensemble d'éléments. Il peut s'agir d'une pile, d'une file, d'une file à priorité ou d'un ensemble tel que ceux que nous avons manipulés lors du dernier TP et de la dernière apnée. Nous supposerons que nous ne connaissons rien de l'implémentation de notre structure de données mais que nous avons néanmoins un moyen de désigner les éléments, 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, en outre, 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 les éléments de notre structure de données une seule fois 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 potentiellement 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 changent les numéros d'indice des éléments : l'implémentation du parcours est donc plus difficile.

Par exemple, pour supprimer les éléments nuls d'un ensemble d'entiers :

```
i=0;
while (i<monEnsemble.taille()) {
    // On reparcourt potentiellement l'ensemble a l'intérieur de lis
    int n = monEnsemble.lis(i);

    // Si on supprime l'élément i, un autre le remplace, les numéros des
    // éléments changent. Ici on s'en sort, mais :
    // - nous sommes obligés de supposer que le remplaçant avait un indice supérieur au supprimé
    // - la boucle est un peu dure à écrire, dans ce cas, pas d'incrémentation de i
    if (n == 0)
        // On reparcourt potentiellement l'ensemble a l'intérieur de supprime
        monEnsemble.supprime(i);
    else
        i++;
}
```

Pour ne pas subir ces désagréments, nous pourrions envisager d'aller toucher directement à la structure de données, mais :

- Cela briserait la bonne encapsulation de notre implémentation. Dans notre cas, nous avons deux implémentations des ensembles, très différentes l'une de l'autre, et briser l'encapsulation nous obligerait à gérer et maintenir ces deux versions de parcours pour tous les parcours que nous aurons à effectuer

- 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 résultant devient plus complexe

Une solution "propre" est de créer une nouvelle classe dans le paquetage de gestion des listes qui implémente la notion de **position dans l'ensemble** et les opérations associées (déplacement et insertion/suppression à la position courante). Ce genre de structure intermédiaire, utilisée pour le parcours de collection d'objets, s'appelle un itérateur, il s'agit d'un concept bien connu en *design patterns* (voir par exemple 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 liste et qui modifie de chaînage lorsque c'est nécessaire. En utilisant un itérateur, le code précédent devient :

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

Les avantages sont que :

- l'implémentation peut alors 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 toute 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

- Récupérez les fichiers Ensemble.java (mise à jour de l'interface `Ensemble`), et Iterateur.java (interface de l'itérateur que vous allez devoir réaliser). Les opérations disponibles pour notre itérateur sont les mêmes (avec la même sémantique) que les opérations des itérateurs de la bibliothèque java, excepté pour l'opération supplémentaire `clone`, qui est juste censée renvoyer une copie de l'itérateur sur lequel on l'appelle. Implémentez les classes `IterateurEnsembleListe` et `IterateurEnsembleTableau` correspondantes et modifiez les classes `EnsembleListe` et `EnsembleTableau` en conséquence.

  REMARQUE:

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

  ```
  Iterateur<T> it;
  it = new IterateurEnsembleListe<>(l);
  ```

  Ce n'est pas une bonne manière de faire : d'une part, si `l` est une référence nulle, la construction de l'itérateur échouera, d'autre part, le type précis de l'itérateur à créer dépend du type d'ensemble utilisé. En effet, l'itérateur manipule directement la structure de données de l'ensemble et doit donc être spécifiquement écrit pour chaque type d'ensemble.

  Pour éviter de devoir gérer ce genre de problèmes, la creation d'un nouvel itérateur est donc confiée à une méthode de l'ensemble lui-même : `iterator`. Si la méthode est appelée, c'est que l'ensemble existe et a une référence non nulle et le problème de référence nulle est résolu. De plus, dans l'ensemble, le type d'itérateur adapté est connu. Enfin, nous gagnons en abstraction : tout ensemble peut fournir une référence d'interface d'iterateur sur lui-même sans que le type d'itérateur correspondant n'ait besoin d'être connu par l'utilisateur.

  En termes d'implémentation, il faut juste que l'ensemble passe une référence à lui même lors de la construction de l'itérateur en utilisant `this` de la manière suivante :

  ```
  public Iterateur<T> iterateur() {
      return new IterateurEnsembleListe<>(this);
  }
  ```

- Vous pouvez tester votre implémentation avec le fichier TP2a.java

2 - Visiteurs

Nous avons maintenant des ensembles génériques qui vont nous servir à stocker des ensembles d'objets de la classe `Composant` et que nous pouvons parcourir facilement grâce à nos itérateurs. Justement, nous allons être amené à parcourir nos objets de multiples fois afin de réaliser les différents traitements requis par notre jeu, pour :

- le rafraîchissement, il va falloir déplacer tout les composants mobiles et changer tous les composants animés à intervalles réguliers
- le traitement des collisions, après avoir calculé les collisions survenant entre les objets de notre jeu, il va falloir appliquer leur effet

- le dessin, qui se chargera de représenter à l'écran nos différents composants

En outre, ces différents traitements sont bien des parcours successifs car ils dépendent les uns des autres : le traitement des collisions corrige les erreurs de positions dues aux déplacements, et le dessin est censé représenter les objets correctement positionnés.

Une solution pour réaliser ces parcours serait de les écrire, chacun dans une méthode distincte, et d'implémenter le traitement correspondant dans des méthodes de nos composants appelées lors de ces parcours. Les problèmes posés par ce genre d'approche sont les suivants :

- le code des composants devient plus complexe. Il regroupe de multiples considérations distinctes (déplacement, collisions, dessin) ce qui rend le code plus difficile à maintenir (il faut connaître et comprendre toutes ces parties)
- le parcours de l'ensemble des composants est écrit plusieurs fois, le code est mal factorisé
- les composants savent comment effectuer leur propre traitement, ce qui les lie à toute bibliothèque externe utilisée pour ces traitements. Par exemple, nous dessinerons en JavaFX, mais nous aurions pu aussi bien choisir Swing ou LWJGL. En plaçant les méthodes de dessin dans le composant, nous rendons le changement de bibliothèque de dessin beaucoup plus difficile : l'indépendance des modules n'est pas très bonne et le modèle MVC (sur lequel nous reviendrons) n'est pas respecté

Une autre solution est d'utiliser une interface permettant de transmettre une opération à utiliser lors du parcours. Cette opération ne fait pas partie du composant et peut être spécialisée pour chaque type de composant. Tout ce que le composant a besoin de faire est d'implémenter une méthode permettant d'accepter cette opération et d'appeler cette opération sur lui même (la bonne variante, dépendant de son type). Cette manière de faire correspond à un *design pattern* appelé visiteur. Le visiteur est l'objet qui implémente l'opération. Les avantages sont alors les suivants :

- une seule version du parcours est écrite. Elle prend un visiteur en paramètre et le passe à chaque objet via sa méthode `accepte`
- le code complet correspondant à un traitement donné est regroupé dans le visiteur : les différentes variantes de la méthode `visite` (qui varient selon le type du composant visité) contiennent les différents traitements à effectuer dans les différentes parties de la structure de données
- seul le visiteur dépend d'une bibliothèque externe utilisée pour le traitement. Pour notre exemple du dessin, le visiteur de dessin peut être placé dans le module de *Vue* du projet sans interférer avec le coeur d'application present dans le composant *Modèle*

Exercices

- reprenez le code du précédent TP et :
  - récupérez le code de la classe Visiteur. Vous pouvez constater que notre visiteur est une classe abstraite contenant une implémentation par défaut de toutes les visites qui se ramène à la visite d'un `Composant`, qui elle-même ne fait rien
  - ajoutez à chaque composant une méthode `accepte` permettant d'appeler la méthode `visite` d'un visiteur donné en paramètre. Il est important d'implémenter cette méthode dans chaque composant (même si le code est le même) car la surcharge fait que l'appel sera résolu différemment

dans chacun des composants. Cette méthode possède une implémentation par défaut dans la classe abstraite Composant dont les autres composants devront hériter. Nous verrons plus tard comment nous allons organiser nos `Composant`, `ComposantGraphique`, composants concrets... Contentez vous pour l'instant de mettre à jour ComposantGraphique pour que cela soit une classe abstraite spécialisant `Composant`. Les bords peuvent désormais n'être que des spécialisation de la classe `Composant` (car il n'ont pas de coordonnées correspondant à un point dans l'espace)
    - Créez une classe `Couche` contenant un ensemble de composants et disposant des méthodes suivantes :
        - `ajoute` : qui se contente de déléguer l'ajout à l'ensemble de composants contenu dans la couche
        - `accepte` : qui parcourt les composants de l'ensemble et appelle `accepte` sur chacun d'eux en transmettant le visiteur passé en paramètre. C'est cette méthode qui, plus tard, s'occupera également de supprimer les composants qui disparaissent au fil du jeu : si la méthode accepte renvoie `true`, supprimez le composant impliqué. Renvoyez `true` à l'issue du parcours si et seulement si l'un des composants a été supprimé
    - modifiez la classe `Niveau` du précédent TP pour que les composants soient maintenant stockés dans des couches. Dans tout le jeu, nous utiliserons au plus 3 couches (une pour les bonus, une pour les autres objets du jeu et une dernière pour les messages), vous pouvez donc utiliser un tableau de 3 couches. Le niveau est un bon endroit pour choisir les ensembles à utiliser pour chaque couche (grand ou petit), vous pouvez donc en profiter pour passer une fabrique d'ensembles au constructeur de la couche. Ajoutez aussi au niveau une méthode `accepte` chargée d'appeler la méthode `accepte` de chacune des couches
    - Créez plusieurs visiteurs concrets pour, par exemple :
        - afficher les composants visités
        - compter les composants visités
        - ...
- vous pouvez utiliser le programme TP2b.java pour tester votre implémentation.