Prélude à la programmation événementielle

Comme pour tout ce que vous entreprenez quand vous codez, pensez à régulièrement tester votre code à l'aide d'un mini programme de test (une méthode `main`).

Exercice 1 : Problématique

Dans cet exercice, on considère les modèles suivants :

- Un point est la donnée d'une abscisse et d'une ordonnée réelles. On est capable de connaître ces valeurs, et on peut aussi connaître la distance qui le sépare d'un autre point. Il est capable d'être translaté suivant un déplacement suivant les abscisses et les ordonnées.
  Une interface spécifiant ce comportement est donnée dans le fichier `Point.java`.
- Un segment est la donnée de deux points. On est capable de connaître la longueur du segment, et celui-ci peut être translaté suivant les abscisses et les ordonnées.
  Une interface spécifiant ce comportement est donnée dans le fichier `Segment.java`.

Commencer par réaliser ces deux interfaces avec les classes `PointOptimized` et `SegmentOptimized`.

Pour comprendre le problème dont il peut être question, on utilisera une relation d'agrégation entre la classe `SegmentOptimized` et la classe `PointOptimized`. Cela signifie que nous autorisons un point à être l'extrémité de plusieurs segments.
On choisira aussi de stocker la longueur du segment. C'est une information redondante (qui peut être calculée très rapidement ici, mais c'est à but démonstratif), mais il arrivera très souvent des cas où il est préférable de stocker ces informations, pour des raisons de performance.

Identifier, à l'aide d'un programme de test écrit en `JUnit4`, le problème d'une telle relation couplée à la présence d'informations redondantes.

Exercice 2 : Première solution

Lorsqu'on utilise une relation d'agrégation entre deux classes, une manière de contrer pas mal de problèmes, dont celui-ci, consiste à rendre le composé omniscient, en rendant l'association traversable dans les deux sens. Cela s'implémente simplement en stockant une référence vers un segment dans l'état d'un point. Cela implique que le point *sait* à quel segment il est associé.

Bien évidemment, ne référencer qu'un seul segment est inutile car cela reviendrait à faire croire au point que la relation qui le lie à son segment est une composition. Dans la pratique, on référence alors un ensemble de segment.

En possédant une référence inverse vers ses segments, un point est capable de demander à ces dernier d'actualiser leur longueur si celui-ci venait à être translaté.

Implémenter cette solution et rejouer les tests pour tester sa correction

Exercice 3 : Généralisation

Mais cette relation bidirectionnelle n'est pas sans impact sur notre modèle et notre programme :

Implanter la référence inverse dans la classe `PointOptimized` revient à réellement lier cette classe avec la classe `SegmentOptimized`. Cela n'est pas grave dans la mesure où la classe `SegmentOptimized` serait définit dans le même package, mais ça ajoute un coût au niveau de l'écriture et de la maintenance du code :

- Premièrement, il faut définir une méthode d'ajout et de retrait des références inverse pour chaque type de modèle.
- Ensuite, il faut ajouter à toutes les méthodes susceptibles d'altérer la référence, du code permettant de notifier ces dernière, et ce morceau de code doit être dupliqué pour chaque type de modèle.

Cela signifique qu'ajouter une référence inverse à un cercle par exemple, ajoutera 1 nouvel attribut, 2 nouvelles méthodes, et modifiera le code toutes les opérations impactant l'état de l'objet, ce qui est plutôt sale.

Proposer (sans implémenter) une solution à ce problème.

Un deuxième impact est lié au fait que posséder une référence inverse fait apparaître un schéma de références circulaires (A référence B qui référence A). Les références circulaires sont un fléau pour la gestion de la mémoire en Java, car la règle que respecte le ramasse-miettes de Java est la suivante :

> Un objet est détruit de la mémoire quand il n'est plus accessible, directement ou indirectement, à l'instant d'exécution du programme.

Autrement dit, tant que notre point (qui peut exister dans le programme en dehors des segments associés car on utilise une relation d'agrégation) est toujours d'actualité, alors tous les segments qui lui sont associés ne sont pas détruits, y compris ceux qui ne servent plus car plus accessibles.

Proposer et implémenter une solution à ce problème.

Commenter quant à son niveau de modularité de la solution et tenter de l'améliorer au maximum. Commenter alors quant au niveau de fiabilité et d'efficacité.

Exercice 4 : Observateur / Observable

Il y a un dernier problème de conception dans la manière dont nous avons décrit nos modèles possédant des informations redondantes : les méthodes d'actualisation des données sont forcément publiques pour pouvoir être appelées par les objets référencés.

En moyenne, une telle classe doit définir pour chaque attribut redondant une méthode publique permettant de mettre à jour cette valeur. Seulement, en termes de spécifications, cette méthode n'a pas à être publique. Elle n'apparaît d'ailleurs pas dans l'interface (ce qui n'est pas syntaxiquement parlé un bon argument), et l'y ajouter serait une hérésie car elle est uniquement liée aux choix d'implémentation (on décide de stocker des infos redondantes).

On constate donc que pour bien faire les choses, il faut rendre les classe observées indépendantes de la nature des classes qui observent (ce que nous avons commencé à essayer de faire dans la première partie de l'exercice précédent).

Schéma classique réduit

Le schéma classique du patron de conception Observateur / Observable implique la présence d'une classe/interface `Observer` mettant à disposition une méthode `update()`, et de sous-typer cette classe avec des réalisations concrètes de cette méthode. Dans notre cas, on aurait alors une classe `SegmentLengthUpdate` qui définirait sa méthode `update()` comme appelant simplement la méthode `updateLength()` du segment dont elle est en charge.

En plus de cette notion d'observateur, on définit la notion de sujet. Le schéma classique n'impose pas l'utilisation d'une classe/interface à part pour ça.
Un sujet doit au moins être capable d'accepter un nouvel observateur (on peut parler d'inscription), d'en retirer un (désinscription), et doit gérer en interne la notification de ses observateurs.
Pour l'exercice, on ajoutera aussi la capacité à compter le nombre d'observateur observant un sujet.

Dessiner le diagramme de classe implantant le patron Obsevateur / Observable classique, et implanter cette solution.

(Re)Nommer les classes pour satisfaire le fichier de test `SegmentOptimizedTest`.

Utilisation de classes internes

Cette solution permet de terminer de satisfaire ce que nous avons tenté d'entreprendre dans la première partie de l'exercice précédent, mais ne résoud pas totalement le problème de la méthode publique chez `SegmentOptimized`.

Il y a cependant une très nette amélioration : le programme principal peut se contenter d'un simple `Segment` et la connaissance du type réel `SegmentOptimized` est réservée à l'observateur `SegmentLengthUpdate`.

En revanche, un autre problème survient : il n'y a pas de raison que tout le package soit au fait de l'existence de la classe `SegmentLengthUpdate` sachant qu'elle est purement liée aux choix d'implémentation faits dans la classe `SegmentOptimized`.

Java permet alors de faire d'une pierre deux coups en utilisant les classes internes.

Définir la classe `SegmentLengthUpdate` en tant que classe interne statique à la classe `SegmentOptimized`. On pourra alors la renommer simplement `LengthUpdate`. On commence par la définir statique* car elle est à priori indépendante d'un segment en particulier.

Apporter les modifications au modèle et implémenter cette amélioration.

Finalement, on peut également définir la classe interne sans le `static`. C'est alors une classe interne d'instance qui a accès aux caractéristiques de l'objet à partir duquel elle a été créée. Ainsi, il n'est plus nécessaire de lui donner en paramètre le segment à mettre à jour.

Apporter les simplifications au modèle et implémenter.

Utilisation de lambdas, et références de méthode

Avec la venue de Java 8, il est possible de simplifier encore plus le code grâce à l'ajout des lambdas.

En Java, une lambda est une écriture (sucre syntaxique) permettant de réduire drastiquement le code à écrire lors de l'utilisation de classes anonymes implémentant sans étant une interface ne comportant qu'une unique méthode. On parle d'interface fonctionnelle car l'interface peut être comparée à l'expression d'une fonction.

Prenons par exemple l'interface `Observer` précédemment créée :

```
interface Observer {
    void update();
}
```

Avant Java 8, utiliser une classe anonyme revenait à écrire le code :

```
Observer o = new Observer() {
    @Override
    public void update() {
        // some stateless code
    }
};
```

Avec Java 8, on peut maintenant tout simplement écrire :

```
Observer o = () -> {
    // some stateless code
};
```

Attention, cette transformation n'est valable que si le code ne dépend pas d'un état interne de la classe anonyme.

Il est possible de simplifier encore plus ce code :
Si le code de la lambda ne consiste qu'en l'appel d'une méthode, alors il est possible d'utilise un référence à cette méthode directement comme lambda.

Par exemple, si le code était le suivant :

```
Observer o = () -> { this.updateLength(); };
```

Alors on peut plus simplement écrire :

```
Observer o = this::updateLength;
```

Simplifier le code du programme à l'aide des lambdas et références de méthodes.

Exercice 5 : Schéma classique complet

Actuellement, les objets observés sont uniquement capable de déclencher une action chez leur observateurs, mais il n'est pas encore possible pour eux de leur transmettre des informations, et les observateurs quant à eux sont incapables de savoir quel est le sujet à l'origine du déclenchement de la méthode.

Pour bien comprendre en quoi cela peut être limitant, considérons l'énoncé suivant :

Un cercle est défini par son centre et un point de sa circonférence. On considère que le rayon du cercle est une information (redondante) stockée et qu'il y a une relation d'agrégation entre le cercle et ses deux points. Tout comme pour le segment donc, la translation de l'un de ses deux points a par conséquent un impact sur le cercle :

- une translation du centre du cercle provoque une translation du point de la circonférence

- une translation du point de la circonférence change la taille du cercle (le centre reste inchangé) et nécessite donc de mettre à jour le rayon du cercle.

Une interface spécifiant le comportement d'un cercle est donnée dans le fichier `Circle.java`.

Commencer par réaliser naïvement cette interface avec la classe `CircleOptimized`.

Adapter la conception du patron pour permettre de prendre en compte les cercles dans le modèle.

(Re)Nommer les classes pour satisfaire le fichier de test `CircleOptimizedTest`.

Limites

La patron de conception Observateur / Observable, qui est a l'origine de la programmation réactive, a beaucoup de limitations conceptuelles et fait face à quelques dilèmnes, dus aussi à l'utilisation de Java.

Conceptuellement parlant, il serait favorable de factoriser le code relatif à la notion d'Observable (ensemble d'observateurs, ajout, retrait, compte, notification, détection de changement, etc...) dans une super-classe abstraite `Observable` (c'est d'ailleurs ce que proposait l'API Java avant Java 9), mais cela a pour conséquence, dù à l'absence d'héritage multiple en Java, d'empêcher la classe observée d'hériter d'une autre classe.
De plus, étant donné qu'il est impossible en Java d'hériter d'un type générique (d'écrire `class Klass<T> extends T`), cela rendrait impossible l'utilisation de décorateur et forcerait soit à définir une sous-classe pour chaque implémentation (`PointOptimizedObservable` pour `PointOptimized`, `ColoredPointObservable` pour `ColoredPoint`, etc...), ce qui baisse le niveau de modularité (à savoir que si on utilise le patron Observateur / Observable, c'est à la base pour augmenter ce niveau de modularité !).

Pour cette limitation de Java, on doit donc se contenter d'un type `Observable` et sommes donc obligés de redéfinir dans chaque modèle observable la manière de gérer les observateurs. Sans compter qu'il y a toujours un risque de bypass de la part de l'utilisateur, risque inhérent au patron Décorateur :

```
Point p1 = new PointOptimized(0, 0);
Point p2 = new PointOptimized(1, 1);
Segment s = new SegmentOptimized(new PointObservable(p1), new PointObservable(p2));
p1.translate(-1, -1);  // bypass du décorateur ! le segment ne sera pas mis à jour
```

Pour finir, ce patron est limité sur trois plans :

- Impossibilité de gérer les duplications
  Imaginons qu'on translate un point, ce qui modifie un segment, lui-même extrémité d'un polygone, et que ce polygone appartient à une image dont on stock l'aire en tant que donnée redondante.
  On va donc déclencher le calcul de l'aire une première fois. Mais si le fait de translater le point redimentionne aussi un cercle, qui appartient à la même figure, bim, calcul de l'aire une seconde fois (et spoiler : ça arrive tout le temps, et pas que deux fois).
- Impossibilité de rendre les notifications conditionnelles
  Imaginons qu'on change la couleur d'un point, les segments associés vont recalculer leur longueur...
- Impossibilité de définir un ordre dans les notifications Il n'y a pas de relation (de priorité par exemple) entre les observateurs, à part l'ordre (statique) si on utilise un ensemble ordonné et qu'on définit un ordre total sur les observateurs.

La programmation réactive (basée sur la programmation événementielle) est capable de résoudre tous ces problèmes.