Diapo3 : intérêt

C’est un design pattern comportemental, c’est-à-dire qu’il améliore la communication avec les objets

On a besoin par exemple de chercher un élément dans une collection ou de faire un foreach dedans, mais on ne veut pas exposer sa structure, ce n’est pas le problème du client.

Grace à ce design pattern, on peut utiliser la même interface pour des structures qui répondent à des logiques différentes

Diapo4 : avantages

Le design pattern iterator sépare la classe de la collection de la manière de la parcourir (on a donc deux classes), ce qui permet de créer une nouvelle classe par manière de parcourir ma collection. Par exemple parcourir l’alphabet à l’endroit ou à l’envers.

Mais comme ces deux classes iterator utiliseront la même interface, on pourra les appeler indifféremment. (En disant retourne l’alphabet en utilisant l’iterator endroit ou envers)

Enfin, on ne se préoccupe pas de la structure interne à la classe et avec ce design pattern, on n’a pas besoin de l’exposer.

Diapo5 : Structure

On a deux interfaces liées entre elles, IAgregate qui sert à créer le IIterator et IIterator qui sert à parcourir la collection. Il vérifie s’il y a un élément suivant et a la méthode pour passer à l’élément suivant (ces 2 méthodes sont parfois fusionnées).

L’aggregate sera la classe métier concrète, mon alphabet utilisé précédemment, et l’iterator sera la classe qui contient la manière de le parcourir. On peut avoir autant d’iterator que de manières de parcourir la collection.

On va commencer par un exemple assez simple, la liste

Diapo6 : La liste

La liste qu’on a l’habitude d’utiliser realise bien l’interface IEnumerable (en passant par IList <T> et ICollection<T>) et contient un itérator qui s’appelle ici Enumerator

On s’en servira par exemple pour ajouter une partie d’une autre collection (pas forcément une liste d’où l’intérêt du design pattern) à notre liste, c’est la méthode AddRange.

On voit ici quand il est nécessaire d’utiliser le design pattern et quand ça ne fait que rajouter de la complexité. Microsoft a estimé que l’implémentation de la list n’avait pas besoin du design pattern lui-même, sauf si elle doit interagir avec des collections qui fonctionnent différemment.

Diapo 7 : insertRange

Ils utilisent l’enumerator ici :

Pour éviter de créer une fonction par type de collection existante, il y a une version optimisée pour ICollection et une version pour les autres où on demande à la collection d’utiliser son propre itérateur pour insérer les éléments un à un dans la liste.

Diapo 8 : Exemple réel

Une étagère oui !

Diapo 9 : le diagramme

Puisque IEnumerable existe, on va s’en servir. On pourrait utiliser nos propres interfaces, mais les différences sont très faibles.

On aura donc une étagère qui contient entre 0 et plusieurs livres. On a deux moyens de la parcourir, chacun représenté par son itérateur.

Diapo 10 : classe Book

Chaque livre est défini par son titre et son auteur, on pourra faire des recherches sur ces critères.

Diapo 11 : classe Bookshelf

On stocke nos livres dans une liste accessible et modifiable.

On a encapsulé les fonctions de la liste (count et [], Add et Remove ) pour ne pas avoir à exposer la liste plus tard.

On ajoute enfin une fonction GetEnumerator qui va nous permettre de choisir la méthode de parcours de la liste par défaut

La dernière ne sert qu’à la rétrocompatibilité car IEnumerable<T> réalise IEnumerable.

Diapo11 : itérateur dans l’ordre

L’Enumérateur sert à parcourir notre classe, ce qui revient ici à parcourir la liste des livres. Pour cela on a besoin de savoir où on est dans la liste, de pouvoir avancer si possible, et de remettre à zéro l’itérateur, voire de le détruire.

Diapo12 : par auteur.

Même principe mais on a trié la liste par auteur avant de la parcourir.

Diapo 13 : App

On crée une étagère avec 5 livres.

Pour parcourir l’étagère, il faut donc lui crée un itérateur, dans le premier cas, c’est celui par défaut de la classe.

On va lui demander de parcourir la classe et d’afficher les données de chaque livre.

Dans le deuxième cas, on crée un itérateur par auteur pour qu’il parcourt la version triée de la bibliothèque.

On écrit le tout dans la console

Diapo 14 : Et voilà c’est joli ça marche

Diapo15 : Conclusion

Le design pattern Iterator est très utile si on a des collections pas simples à parcourir, ou qui peuvent avoir différentes manières d’être parcourues.

Cependant cela rajoute de la complexité donc si vous utilisez des collections qui existent déjà (au hasard des listes), que vous parcourez toujours de la même manière, c’est moins utile.

Design Pattern Singleton

Diapo2 : design pattern créationnel, il permet d’être sûr que classe n’a qu’une seule instance d’elle-même. C’est utile pour créer un logger ou gérer des connections à une base de données, bref ça permet de centraliser une partie d’un système et de la partager avec le reste de l’application.

Cependant cela rajoute de la complexité et rend les tests un peu plus difficiles, il faut aussi gérer la création et destruction du singleton

Accessoirement, ça viole les principes SOLID puisque le singleton a deux responsabilités, gérer sa création et exécuter une tâche.

Regardons comment ça fonctionne en théorie :

Diapo3

On peut voir que le constructeur est privé, on ne peut donc pas créer une instance sur commande. Il faudra passer par son Getter, qui lui va s’en occuper.

Diapo4

On va prendre un exemple simple, la machine à voter. Son but est d’envoyer un message à chaque vote et de les compter.

Diapo5

On peut voir ici que l’instance est en attribut de la classe, le constructeur est bien privé.

Quand on appelle l’instance, si elle n’existe pas, cela en crée une nouvelle et la retourne, sinon on prend celle qui existe déjà.

On peut ensuite la manipuler.

Diapo6

On va tester le tout en appelant trois fois l’instance de la machine et en ajoutant un vote à chaque fois. Si chaque numéro de vote est différent, on aura bien créé une seule machine, sinon certains votes ne compteront pas.

Diapo7

Et voilà ça marche

Diapo8 :

Maintenant on va lancer 10 fois la même requête en parallèle

Diapo9.

Le temps que l’instance soit effectivement créée, le programme a pu accéder 4 fois à une version de la classe où l’instance est nulle, 3 votes ont donc été perdus.

Il va donc falloir empêcher le programme d’accéder à notre instance tant qu’elle n’a pas été actualisée

Diapo 10. Cette fois on va verrouiller l’objet en cas de doute, c’est-à-dire si on voit une instance nulle (ça économise des ressources). Les requêtes qui arrivent en parallèle devront donc attendre que la ressource soit libre.

Diapo11. Cette fois tout fonctionne. Mais cette approche n’est pas standard. Il existe une autre façon de voir les choses.

Diapo12. Après tout, il existe un moment où on est sûr qu’il n’y a pas de requêtes en parallèle, à l’initialisation du programme. On va donc créer notre instance à ce moment là

Diapo13.

Diapo14.

Lazy va initialiser l’instance seulement quand c’est nécessaire. Il contient toutes les sécurités nécessaires pour que ça fonctionne en asynchrone.

Diapo15.