Guilaume Gosset

retranscription du cours n°6 :

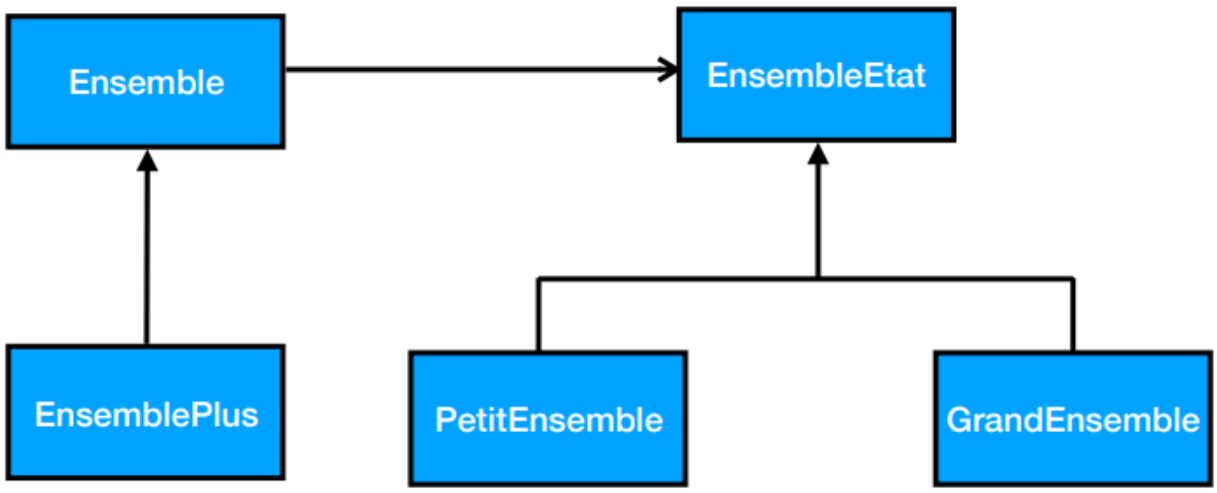
Rappel du cours précédent :

Le state est un design pattern qui permet de créer des objet pouvant changer de représentation interne de manière dynamique.
→ Comme il est impossible de modifier ce que contiens « this » dans une instance en java, on passe donc par un objet interne qui contiendra la représentation interne effective de l'objet. Cet objet sera du type d'une sur-classe de plusieurs sous classes représentant les différentes représentations internes de la classe. Ainsi cet objet pourra toutes les contenir successivement sans aucun problème . **Attention ce fonctionnement doit être TRANSPARENT a l'extérieur de la classe, on doit pouvoir l'utiliser comme une classe normale !**

Un exemple d'implémentation de state serait le suivant :

```
public class Ensemble
{
     private static final int N = 1000;
     private EnsembleEtat e;
     public Ensemble()
     {
          e = new PetitEnsemble();
     }
     public void inserer(int x)
     {
          if(!e.contient(x) && e.taille() + 1 > N)
               e = new GrandEnsemble(e); e.inserer(x);
     }
}
```

Le bridge est utilisé dans le cas où on veut effectuer une connexion entre plusieurs classes, mais qu'une relation d'extension ne serait pas logique car il n'y a pas de réel lien sur/sous classe entre les deux objets que l'on souhaite lier.
→ Le bridge permet donc de créer une connexion horizontale entre les deux : un « pont » comme on peut le voir sur le schéma si dessous.



Le strategy and command est un design pattern qui est utilisé pour palier a un des défauts du java, il n'est pas possible de passer des fonctions particulières en paramètre d'une methode.
→ pour cela on crée des objets n'ayant qu'une seule fonction : celle que l'on veut réaliser dans ce cas.

Attention les opérandes du calcul a réaliser dans la fonction doivent être passé A LA CONSTRUCTION de l'objet.

L'observer est un design pattern composé de deux élément : les observateur et les observés, les observateurs peuvent s'abonner ou se désabonner d'un observé a l'aide des méthodes subscribe() ou unsubscribe(). Une fois qu'il sont abonné a un observé ils seront notifiés par les observé au moment ou ceux-ci changent de status.
→ les observés utilises pour cela la méthode update() envoyant la notification a la liste de tout les observateurs abonnés.

Collections :

Il existe différents types de collections en java : les tableaux qui ont une taille statique, les vecteurs qui sont des tableaux qui, quand ils deviennent trop court pour contenir toutes les valeurs que l'on souhaite mettre dedans, il se recopie dans un nouveau tableau plus grand. Il y a aussi les listes qui sont des structures récursives qui ne sont donc plus nécessairement stockées de manière contiguë dans la mémoire.

Enfin on peut utiliser les map dans le cas ou on a une paire clé/valeur
→ ce type de collection permet d'avoir un accès en O(1).
Cependant il faut éviter de faire parcours sur des tables de hachage car elle ne sont pas très adaptées a ce mode de traitement.

Dans le cas ou on voudrais effectuer un parcours rapide, une structure comme un tableau ou un vecteur serait plus adaptée.

Si l'ordre des éléments est important il faudra utiliser de préférence une liste. A l'inverse si l'ordre n'est pas important on peut utiliser un ensemble.

Pour avoir des informations supplémentaires utiles sur les collections on peut lire la javadoc.

Suite du cours :

Imuabilité ;
Un objet imuable est un objet dont on ne peut pas changer le contenu une fois qu'il a été crée.
→ c'est a dire que ces champs ne pourront plus être modifiés après la création de l'objet.

En particulier, la classe d'un objet immuable n'aura pas de setter dans son code, ainsi l'utilisateur de la classe ne pourra pas modifier son contenu.

Le principal avantage de cette technique est qu'elle simplifie considérablement la réflexion. Car au cours du fonctionnement programme l'état d'un objet imuable ne bougera jamais durant l'exécution programme alors que celui d'un objet non imuable pourra varier en fonction du temps.
Par conséquent au moment ou on est dans une phase de test et que l'on a des erreurs. Si on a une classe non imuable il sera nécésssaire de suivre l'historique de l'évolution de l'état de l'objet au cours du temps pour bien comprendre d'ou viens le problème. Alors que si l'objet est imuable on n'aura pas a ce soucier de cela au moment ou on cherche les erreurs.

Par ailleurs l'immuabilité permet aussi la réutilisation des éléments une fois qu'ils ont été modifiés, car même une fois la modification effectuée, l'ancienne version de l'objet existera encore dans la mémoire et si il existe encore un pointeur qui permet d'y accéder on pourra toujours avoir accès a l'objet.

Pour illustrer cette notion d'objet immuable , on peut prendre en exemple les string en java.

En effet car dans le cas ou on modifie une chaîne de caractère en java, ce n'est pas la chaine qui est modifiée directement, l'instance de la classe string reste la même, en revanche une nouvelle instance de la classe string est crée et placée a l'intérieur de l'objet que l'on manipule.

Il en va de même pour les calculs par exemple si on réalise le calcul str1 + str2, str1 et 2 ne seront pas modifiés ici, une concaténation de ces deux chaînes de caractères sera placée dans une nouvelle instance de la classe string.

Il en va aussi de même pour la fonction replace(). Qui s'applique pourtant a une chaîne de caractère, par exemple si on exécute str1.replace('o','f') la chaine qui est actuellement stcokée dans str1 ne sera aucunement modifiée, en effet il y aura encore une fois une nouvelle instance de la classe string qui sera crée avec les modifications puis placée a l'intérieur de str1.

→ Cependant bien que cette méthode présente plusieurs avantages elle occasionne aussi un désavantage majeur qui est la perte de ressource, car comme a chaque fois que l'on effectue une action sur ces objets imuables ils sont recopiés en mémoire on se retrouves très vite avec de très nombreuses instances de l'objet en question dans la mémoire, ce qui risquerai de la saturer si jamais on ne possède qu'une très faible quantité de mémoire alouée pour le programme.

→ La chaîne de caratère un élément qui permet de palier en partie a ce problème. Dans le cas ou on utilises la méthode substring sur une chaine de caractère particulière, il n'y aura pas de recopie de l'objet, mais on utilisera deux valeurs qui sont associées a une chaine de caractère qui sont les valeurs : « count » et « offset ».
→ la valeur « offset » permet d'indiquer a partir de quel caractère de la chaîne ou souhaite commencer la sous chaine et count permet de spécifier combien de caractères on veut garder a partir de la position de l'offset.

Par conséquent dans le cas ou on souhaite afficher la chaîne de caractère en entier, on utilisera un offset de 0 et un count égale a la longueur de la chaîne de caractère.

Mais dans le cas ou on veut avoir une sous chaine ne comprenant qu'une partie de la chaine, on peut avoir une chaine dont le value pointra au même endroit que la chaine principale et seuls le count et l'offset changeront en fonction des besoins de l'utilisateur, a partir d'où souhaite t'on lire et jusqu'où souhaitons nous lire.

Par conséquent les complexitées en temps n'est pas la même dans les deux cas, en effet dans le cas ou on effectue une concaténation, il y a recopie on a donc une complexitée O(n1 + n2)
Car on doit copier le contenu des deux chaines dans une 3ème qui est nouvelle il faudra donc prendre le temps de copier chacunes de ces chaines.

Alors que dans le cas ou on effectue un substring le temps est constant, car on récupère juste la référence sur la valeur et on place les valeurs souhaitées dans l'offset et dans le count ce qui est donc une opération a temps constant : O(1).

Ce système de value-offset-count serait impossible a réaliser si jamais la string n'était pas immuable, car si on modifiais une chaine de caractère , toutes les chaines issu d'une substring de celle ci serai aussi modifiées.
→ de manière plus général, cela empâcherai totalement la réutilisation des éléments.

Pour conclure sur l'imuabilité :

les objet imuables possèdes des avantages :

toutes les propriétés sont invariantes en temps : en particulier son hashcode qui ne changera par conséquent jamais (ce qui est particulièrement important si il doit être placé dans un hashmap ou un hashset, car si ce hashcode venait a changer, il serait alors totalement impossible de le retrouver dans cette table.

Permet dans certains cas d'économiser de la mémoire grace a la réutilisation.

Il est par ailleurs aussi Thread safe, car son imuablité permet d'éviter les problèmes qui pourraient être occasionés par des tentatives d'écritures concurentes sur l'objet par deux thread différent dans le programme.

Mais il possède aussi des désavantages :

Il peut par exemple aussi dans d'autres cas ce révéler très couteux de par le méchanisme qu'il a de ce faire recopier a chaque fois que celui ci est modifié.
→ Un garbage collector est donc nécéssaire pour récupérer de la mémoire en supprimant toutes les vieilles version des objet qui sont devenues inutiles car elles ne seront plus jamais récupérables, dans le cas où, par exemple, plus aucun pointeurs ne pointes sur l'objet et il sera donc totalement impossible de le récupérer quoi qu'on fasse à ce moment la.

→ Pour reprendre l'exemple du triangle que l'on a vu dans le cours sur l'héritage. Dans le cas de ce triangle il serait plus avantageux de faire une fonction retournant un nouveau triangle comprenant les modification
|→ on retrouve une instance de la surclasse qui est compatible avec la modification.

Ou inversement on peut faire un test pour voir si une sous classe particulière de la classe triangle ne serait pas plus adaptée a la nouvelle modification que l'on a effectué sur l'objet.

Debugging :

→Attention dans le cas d'un débuggage on veut cherche pas a trouver de bug dans le programme : car on cherches a avoir un programme qui fonctionne.
Le test est donc réussi a partir du moment il n'y aura pas de problème détecter.

Pour effectuer le débuggage on dispose de plusieurs outils différents :

tout d'abord l'affichage a l'écran (via le system.out.println() en java).

Pour essayer de comprendre comment fonctionne le code que l'on a écris on peut essayer d'écrire a l'écran le contenu des différentes variables du programme pour essayer de voir si il n'y aurait pas un probleme parmi le contenu de l'une d'entre elles.

On peut aussi tester, par un affichage si le programme passe bien par une partie donnée du code, (en effectuant par exemple un affichage en début de bloc de code permettant d'attester que l'on passe bien par celui ci durant l'execution du programme.

Ensuite on peut essayer de débugger en lisant le code

→ Si on a un bug dans notre programme c'est que l'on a pas totalement compris.
Il faut donc regarder progressivement, ligne a ligne l'évolution des différents paramètres pour essayer de repérer ce qui nous a échapper dans l'execution du programme pour ensuite éventuellement pouvoir le corriger.

Cependant il faut faire attention lors de cette étape, durant la lecture du code, il ne faut jamais prêter attention au commentaires placés dans le code, ceux cis pourraient nous induire en erreur.

On peut aussi, pour débuger le programme en utilisant des assertion, ce sont des condition qui sont placés dans le code permettant de savoir si elle est réalisée a un moment particulier du programme, si elle ne l'est pas, le programme s'arrête et un message d'erreur nous le signale.

Methode du « nounours » : cette méthode ne repose pas sur un ajout a faire sur le code en lui même, mais il s'agit d'une réflexion que l'on doit mener par nous même, en effet cette méthode consiste a se poser la question de pourquoi nous subissons actuellement ce problème et de voir si en formulant clairement le problème que l'on a, la solution n'en découle pas d'elle même.

On peut enfin utiliser différents débuggers qui sont proposés dans des IDE.

Pour réaliser un débuggage, il faut être systématique pour s'assurer qu'on teste bien tout ce qui pourrait poser un problème, ce qui peut s'avérer être très compliqué et demander beaucoup de temps.

→ encore une fois commencer par une lecture du code peut être judicieux pour retrouver le cheminement qui a conduit le programme a nous fournir la sortie que l'on a obtenue et comment est il possible pour le programme de nous fournir des sorties incorrectes en général.

Quand on effectue un débuggage, on suit au final un peu la même méthodologie que quand on réalise un expérience scientifique : on observe tout d'abord le problème, puis on en dégage une hypotèse, que 'on essaye de démontrer ou d'invalider au cours d'une expérience nous permettant de tirer des conclusions vis a vis de l'hypotèse de base que l'on avait formulé. Si on n'as pas trouvé la solution, il faut alors recommencer un cycle, en repartant observer le programme dans le but d'essayer de nouvelle hypothèses a explorer.

Il faut penser a travailler de manière efficace quand on effectue un debug, il faut penser a vérifier les erreurs les plus courrantes, un ; oublié, un = au lieu d'un ==, un == au lieu d'un equals, ou encore des divisions de float/double avec des entiers…

Il faut aussi penser a regarder les couples entrées sorties pour essayer de localiser a quel endroit a t'on en sortie des valeurs qui sont faussées par rapport a ce qui rentres permettant ainsi de localiser les parties du programme ou sont les problèmes.
→ quand on effectue un débuggage il ne faut négliger aucune source de données qui est a notre disposition : le code , la documentation de l'api que l'on utilises si on en utilises une …

Dans le débuggage il faut donc formuler des hypothèses quand au fonctionnement du programme que l'on traite puis de la tester pour ensuite la vérifier ou la réfuter en fonction de l'issu des tests que l'on a effectué.

→ Il faut cependant faire très attention a la reproductibilité des problèmes qui surviennes, en fonction de cas de concurence d'accès a des valeurs ou des aléas éventuels pouvant survenir dans certains cas. Car dans le cas ou on a de graves problèmes qui surviennent que dans certains cas et que celui ci n'est pas détecter ou qu'on arrive pas a le reproduire pour en identifier la source et donc le corriger, on risquerai par exemple a ce retrouver avec un programme qui fonctionne bien pendant une grande majeure partie du temps, mais qui dans certains cas pourrait entrainer des comportements désastreux pouvant par exemple mettre en danger des personnes.
→ pour ce genre de problème il faut penser a les traiter avec des assert, ainsi si un problème de ce genre survient on sera sur que le programme s'arrête et qu'aucune conséquence ne pourra découler directement de cette erreur.

Cependant il faut aussi faire attention si on souhaite corriger ces bugs occasionnels, car il y en a certains qui on tendance a disparaître lorsqu'on met en place des asssert pour les intercepter.

Quelques autres astuces a utiliser lors du débuggage :

-il faut bien penser a recompiler le code pour s'assurer que l'on est pas en train d'exécuter une ancienne version du programme ne prenant pas en compte les dernières modiffications que l'on a effectué.

-Il faut aussi vérifier que le code source que l'on est en train de modifier est bien celui que l'on est en train de compiler a coté, car sinon encore une fois on peut ce retrouver a compiler des anciens code source ne prenant pas en compte les dernières modiffications que l'on a effectué pour corriger le bug qui est survenu.

-Il faut aussi s'assurer une sauvegarde des anciennes versions du code (via des logiciels de gestion de version par exemple) ainsi si jamais les tentatives de correction que l'on a appliquées sur le code pose plus de problèmes qu'ils n'en résolve, on peut être sur que l'on pourra revenir aux anciennes version du code, pour repartir sur des bases plus solides et ainsi continuer le débuggage.

-On peut aussi demander a un collègue de vérifier en lui expliquant le code, car avec un œil neuf parfois, on peut repérer des erreurs que l'on ne verra pas forcément si on travaille sur le code depuis parfois plusieurs jours ou mois par exemple.

-Il faut aussi ce rappeller que ce qu'on essaye de débugger, c'est le code et non pas les commentaires, il ne faut encore une fois pas en tenir compte durant le débuggage car il n'y a aucune garantie que le code en dessous du commentaire fait bien exactement ce que dit le commentaire. Cela pourrait donc nous induire en erreur pendant la phase de débuggage et nous empêcher de voir d'ou provient le problème que l'on essaye de résoudre, pire cela pourrait nous conduire sur une fausse piste, nous faisant alors perdre encore plus de temps.

- Il peut aussi parfois être mieux de faire une pause pendant que l'on effectue un débuggage, car quand on passe trop de temps sur un code encore une fois, on ce focalise trop sur des détails et on ne regarde pas l'ensemble ce qui pourrait nous empêcher totalement de voir le problème qui est la, tout simplement car on ne regarde pas a une échelle nous permettant de voir l'erreur. Ainsi faire une pause nous permet de reprendre du recul sur notre code et de le réaborder de manière plus général et ainsi potenciellement trouver l'endroit d'où provient le problème de manière plus facile.

-Enfin, il est important de garder a l'esprit qu'ajouter du code n'est pas nécéssairement la bonne solution pour résoudre un problème car parfois il s'agit d'un problème de logique pure, il faut alors repenser totalement son code et le remodifier pour corriger cette erreur de logique qui empêche notre programme de fonctionner.