TP4 LOG2410

16/11/2019

Patron Visiteur

- **Réponses aux questions :**

1 - a) L'intention du patron Visiteur :

Selon les notes du cours L'intention du patron Visiteur est « Représenter une opération qui doit être appliquée sur les éléments d'une structure d'objets. Un Visitor permet de définir une nouvelle opération sans modification aux classes des objets sur lesquels l'opération va agir. »

En d'autres termes, le patron visiteur permet de séparer les différents objets d'une structure de données. Il permet de mettre en place un traitement adapté aux détails des objets en cause. Le patron visiteur permet de modifier et d'ajouter des fonctionnalités sur des éléments d''une structure d'objet sans modifier la structure des classes des objets sur lesquels ces fonctionnalités vont agir. Le patron visiteur offre une grande flexibilité en permettant l'ajout et la suppression des méthodes de traitement avec beaucoup de facilité. Il permet l'indépendance des opérations entre les classes.

b) la structure des classes réelles qui participent au patron ainsi que leurs rôles :

Dans le patron visitor, les diagrammes se divisent en deux catégories : les visiteurs et les éléments et dans chaque catégorie il y a une partie abstraite ainsi qu'une partie concrète. Donc, on aura les classes suivantes :

- **Visiteur abstrait :** *AbsComponentVisitor*
- **Visiteurs concrets :** *MemberTextFindReplace* et *TeamImageSizeCalculator*
- **Élément abstrait :** *AbsTeamComponent*
- **Elements concerts:** *Team*, *TeamMember*, *TeamMemberRole*

Les classes *MemberTextFindReplace* et *TeamImageSizeCalculator* permettent la visite des éléments cités ci-haut qui acceptent la visite grâce à une fonction accept(). Ces éléments dérivent de la classe abstraite *AbsTeamComponent*. Donc, dans un premier diagramme nous allons présenter le diagramme de classe du visiteur *MemberTextFindReplace* pour lequel le visiteur abstrait et les éléments, abstrait et concrets, demeurent inchangés pour le visiteur *TeamImageSizeCalculator.*

- Voir DiagrammeDeClasses_ MemberTextFindReplace.pdf et DiagrammeDeClasses_TeamImageSizeCalculator.pdf

c) Effet de l'ajout d'une sous-classe en cours de conception :

On peut facilement ajouter des sous-classes qui dérivent de la classe abstraite *AbsTeamComponent*, car cela n'affecte pas vraiment les autres classes qui y sont reliées. Cela est dû au fait qu'on ait un patron visiteur qui permet qui permet justement d'ajouter des fonctionnalités aux éléments d'une structure d'objets sans modifier la structure des classes. Dans notre cas, les classes *Team*, *TeamMember* et *TeamMemberRole* (classes qui dérivent de *AbsTeamComponent*) représentent les éléments qui peuvent accepter des visiteurs. Donc, si on veut ajouter une sous-classe qui dérive de *TeamAbsComponent*, on a qu'à ajouter une fonction de visite dans chacun des visiteurs et accepter l'accès dans la sous-classe même.
Donc les classes à modifier c'est les classes visiteurs, soient : *AbsComponentVisitor*, *MemberTextFindReplace* et *TeamImageSizeCalculator*.

d) Implémentation d'un visiteur pour la fonction d'ajout d'un rôle pour un membre :

Oui, l'ajout d'un rôle pour un membre peut être implémenté comme un visiteur car l'ajout de rôle doit être applicable sur les membres conservés. Donc, en implémentant cet ajout dans un visiteur, on augmentera la flexibilité, la réutilisabilité du système, mais au même temps le coût d'exécution va augmenter, car on aura forcément une duplication de code et le coût de communication entre les visiteurs augmenterait avec l'augmentation du nombre de classes visiteurs.

Patron Commande

1 - a) L'intention du patron Commande :

Selon le cours, le patron commande permet de : « Encapsuler une requête dans un objet de façon à permettre de supporter facilement plusieurs types de requêtes, de définir des queues de requêtes et de permettre des opérations « annuler » »

Autrement dit, c'est un patron qui permet l'ajout, la suppression ainsi que la gestion aisée des nouvelles opérations sur une classe sans avoir à la modifier. Ces nouvelles opérations sont alors indépendantes de cette classe.

Les avantages :

- ✓ Découple l'objet qui invoque la requête de celui qui sait comment la satisfaire.
- ✓ Les commandes sont encapsulées dans des objets qui peuvent être manipulées comme tout objet. L'utilisation d'objets amène également plus de flexibilité.
- ✓ On peut facilement créer de nouvelles commandes.
- ✓ Les commandes peuvent être assemblées en des commandes composites si nécessaires. Le patron Command peut être combiné au patron Composite pour représenter des commandes qui sont un ensemble d'autres commandes.

- b) La structure des classes réelles qui participent au patron ainsi que leurs rôles :

- **Exécuteur** : *CommandInvoker*
- **Commande** : *AbsCommand*
- **Commande concrète**: *CommandTranslate* et *CommandCalculateSize*
- **Récepteur :** *AbsTeamComponent*

➢ Voir DiagrammeDeClasses_Command.pdf

2 - a) Identification des deux patrons de conceptions :

En plus de participer au patron Commande, cette classe participe à deux autres patrons de conception vus en cours qui sont **singleton** et **Mediator**

Intention Singleton : S'assurer qu'il ne soit possible de créer qu'une seule instance d'une classe, et fournir un point d'accès global à cette instance.

Intention Mediator : Définir un objet qui encapsule comment un ensemble d'objets interagissent afin de promouvoir un couplage faible et de laisser varier l'interaction entre les objets de façon indépendante.

b) Éléments caractéristiques de ces patrons de conception :

- Pour le patron singleton, l'attribut m_instance de type CommandInvoker qui est statique. Fait en sorte qu'il y a une seule instance de la classe. Et getInstance() qui s'assure de l'unicité instance de la classe existe
- Pour le patron Mediator : la fonction statique getInstance()
  Nécessité de la classe abstraite AbstractColleague qui est la classe AbsCommand

c) Justification d'utilisation des patrons de conception :

Singleton : son objectif est de restreindre l'instanciation d'une classe à un seul objet (ou bien à quelques objets seulement . Il est utilisé lorsqu'on a besoin exactement d'un objet pour coordonner des opérations dans un système. Le modèle est parfois utilisé pour son efficacité, lorsque le système est plus rapide ou occupe moins de mémoire avec peu d'objets qu'avec beaucoup d'objets similaires.

Mediator :

Les références sont difficiles à changer de façon non invasive, Tous les objets doivent payer le prix de références à un ensemble d'objets. Donc on utilise le patron Mediator pour Simplifie le

protocole entre les objets, Il est utilisé pour réduire les dépendances entre plusieurs classes. Lorsqu'un logiciel est composé de plusieurs classes, les traitements et les données sont répartis entre toutes ces classes. Plus il y a de classes, plus le problème de communication entre celles-ci peut devenir complexe. En effet, plus les classes dépendent des méthodes des autres classes plus l'architecture devient complexe. Cela ayant des impacts sur la lisibilité du code et sa maintenabilité dans le temps.

Le modèle de conception Médiateur résout ce problème. Pour ce faire, le Médiateur est la seule classe ayant connaissance des interfaces des autres classes. Lorsqu'une classe désire interagir avec une autre, elle doit passer par le médiateur qui se chargera de transmettre l'information à la ou les classes concernées.

3 -) **Pour compléter la fonctionnalité de TeamViewer, il faudrait ajouter de nouvelles sous-classes de la classe AbsCommand. Selon vous, est-ce que d'autres classes doivent être modifiées pour ajouter les nouvelles commandes? Justifiez votre réponse.**

Non, on n'a pas besoin de modifier d'autres clases pour ajouter une nouvelle commande autre que la nouvelle classe ajoutée. C'est justement l'intention du patron commande qui permet l'ajout des opérations sans avoir à modifier les classes. On peut juste ajouter des sous-classes de AbsCommand pour pouvoir ajouter les nouvelles commandes.

4) Dans la version proposée de la commande de traduction par la classe CommandTranslate, on suppose que la traduction d'une langue vers l'autre peut être effectuée de façon « inversible », c'est-à-dire sans perte d'information durant la traduction. Selon vous, cette supposition est-elle réaliste ? Sinon, que faudrait-il faire pour rendre l'opération « undo » robuste si on ne peut pas supposer que l'opération est inversible ? On ne vous demande pas d'implémenter une solution au problème, mais simplement de décrire le problème et la façon dont vous vous y prendriez pour le régler.

Personnellement on ne peut jamais traduire de langue mot à mot sans perte de donnée et la proposition n'est pas réaliste car par exemple si on traduit une langue vers une autre a cause de différence de vocabulaire utilisé les sens de mot etc. on va surement perdre quelques informations. Pour rendre l'opération « undo » robuste on doit faire Conservation d'état pour les *undo*.
Il se peut que les commandes ne soient pas réversibles ou que la succession de plusieurs undo et en alternance finissent par faire diverger l'état de l'application par accumulation d'erreurs.

Il peut donc être nécessaire de conserver l'état de l'objet receveur(ainsi que de tout autre objet pouvant être affecté par l'exécution de la commande). Les opérations *undo* d*e*viennent alors des opérations de restauration et de sauvegarde d'états sur des objets.

- ⇨ Le patron Memento propose une solution à ce problème de sauvegarde et de restauration d'état qui peut être mise à profit dans le patron Command.