Design pattern Décorateur (decorator)

Soumis par Mathieu G. le Dimanche 20/05/2007 17:32 - Dernière modification le Lundi 10/10/2011 21:45

Catégorie: Structure

Fréquence d'utilisation: Normale

Difficulté: Intermédiaire

Le pattern Décorateur (Decorator) attache dynamiquement des responsabilités supplémentaires à un objet. Il fournit une alternative souple à l'héritage, pour étendre des fonctionnalités.

Description du problème

Dans la programmation orientée objet, la façon la plus classique d'ajouter des fonctionnalités à une classe est d'utiliser l'héritage. Pourtant il arrive parfois de vouloir ajouter des fonctionnalités à une classe sans utiliser l'héritage. En effet, si l'on hérite d'une classe la redéfinition d'une méthode peut entraîner l'ajout de nouveaux bugs. On peut aussi être reticent à l'idée que des méthodes de la classe mère soient appelées directement depuis notre nouvelle classe.

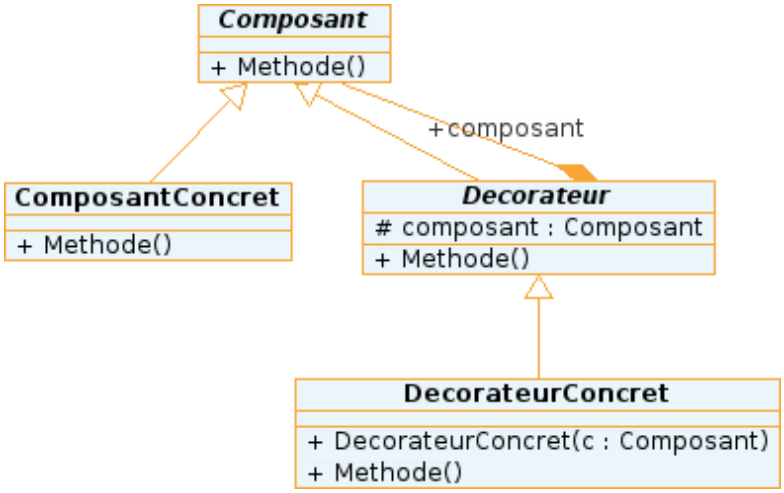
De plus, l'héritage doit être utilisé avec parcimonie. Car si on abuse de ce principe de la programmation orientée objet, on aboutit rapidement à un modèle complexe contenant un grand nombre de classes.

Un autre souci de l'héritage est l'ajout de fonctionnalités de façon statique. En effet, l'héritage de classe se définit lors de l'écriture du programme et ne peut être modifié après la compilation. Or, dans certains cas, on peut vouloir rajouter des fonctionnalités de façon dynamique.

D'une manière générale on constate que l'ajout de fonctionnalités dans un programme s'avère parfois délicat et complexe. Ce problème peut être résolu si le développeur a identifié, dès la conception, qu'une partie de l'application serait sujette à de fortes évolutions. Il peut alors faciliter ces modifications en utilisant le pattern Décorateur. La puissance de ce pattern qui permet d'ajouter (ou modifier) des fonctionnalités facilement provient de la combinaison de l'héritage et de la composition. Ainsi les problèmes cités ci-dessus ne se posent plus lors de l'utilisation de ce pattern.

Diagramme UML

- C'était l'explication la plus il y a 3 années 12 mois



Définition de la solution

La classe abstraite Composant définit le point de départ de ce diagramme. Plusieurs ComposantConcret peuvent hériter de Composant. Si l'on souhaite étendre (ou modifier) l'ensemble des fonctionnalités des ComposantConcret on peut créer un décorateur. Il s'agit d'une classe abstraite héritant de Composant et ayant un attribut de type Composant. De plus, Decorateur déclare abstraite la méthode dont l'on souhaite étendre les fonctionnalités. Pour ajouter des fonctionnalités à un ensemble de ComposantConcret on va créer des classes DecorateurConcret qui héritent de Decorateur. Un DecorateurConcret contient un constructeur permettant d'initialiser l'attribut composant présent dans le décorateur. Il faut ensuite que la classe DecorateurConcret redéfinisse la méthode déclarée abstraite dans le décorateur. Lors de cette redéfinition il est possible d'étendre les fonctionnalités en appelant la méthode de l'attribut composant et en ajoutant des traitements.

Suivant les besoins spécifiques de chacun ce pattern peut être adapté. En effet, il est tout à fait possible d'utiliser des interfaces pour le composant et le décorateur. Dans ce cas, les attributs et les méthodes seront définis dans les sous classes.

Bien sûr ce pattern utilise largement l'héritage mais il utilise aussi la composition grâce à l'attribut Composant présent dans le décorateur. C'est l'alliance de ces deux procédés qui permet à ce pattern d'être si efficace.

Explication détaillée de la solution

Voyons plus concrètement comment fonctionne ce pattern en prenant un exemple de l'utilité de ce pattern. Tout d'abord on a une classe ComposantConcret qui possède une méthode chargée d'une fonctionnalité. Suivant l'objet créé on souhaite ajouter des traitements lors de l'appel de cette méthode. Cependant on ne doit pas modifier directement le corps de la méthode car certains objets utiliseront toujours l'ancienne version de cette méthode. Pour cela on peut créer un objet DécorateurConcret en passant à son constructeur notre objet ComposantConcret (dont l'on souhaite étendre les fonctionnalités). On peut ensuite redéfinir la méthode concernée et ajouter des traitements. On appelle la méthode du ComposantConcret puis on rajoute des fonctionnalités.

On obtient un objet ComposantConcret qui est emballé dans un DecorateurConcret. Ainsi si on appelle la méthode sur l'objet décorateur, celle-ci va appeler la méthode du composant concret, ajouter ses propres traitements et retourner le résultat. A noter, que si on appelle directement la méthode de l'objet ComposantConcret (sans passer par le décorateur) on utilise alors l'ancienne version de la méthode.

Conséquences

Comme tous les patrons de conception, Décorateur ne doit pas être utilisé à tord et à travers. Mais lors de la conception de classes qui risquent d'évoluer fortement (ajout ou modification de fonctionnalités) celui-ci sera très utile. Il est donc important de bien réfléchir aux points sensibles de l'application qui risquent d'évoluer au fil du temps et cela dès la phase d'analyse.

Attention tout de même à l'utilisation des types concrets. Si votre application se base sur les types concrets d'objets utilisés dans le pattern décorateur cela posera des problèmes. En effet, une fois décoré un ComposantConcret aura pour type concret celui de son décorateur le plus externe.

De plus, lors de l'utilisation du pattern Décorateur, on constate qu'il est fastidieux de gérer tous les objets créés et de les décorer. C'est pour cette raison que ce pattern est souvent utilisé avec le pattern Fabrique ou Monteur qui répondent à cette problématique.

Exemples d'implémentations:
Design pattern Décorateur en Java : vente de desserts

Ajouter un commentaire

Commentaires

Décorateur

Soumis par Anas (non vérifié) le **Mercredi 12/12/2012 00:20**

Merci et chapeau !! j'ai passé les 2 dernières journées à essayer d'assimiler ce design pattern, il me semblait complexe jusqu'à ce que j'ai commencé à lire votre article. Quelle pédagogie ! Bonne continuation :)

Décorateur

Soumis par Mathieu G. le **Mardi 08/01/2013 22:11**

Merci beaucoup, ravi d'avoir pu vous être utile :)

Decoratuer

Soumis par lamia (non vérifié) le **Lundi 12/05/2014 14:29**

pour enrechir l'article voilà un lien intérssant pour le design pattern (**http://design-patterns.fr/decorateur#comment-form**)

Good job

Soumis par youssef (non vérifié) le **Vendredi 21/11/2014 13:50**

Félicitation pour ce travail bien fait, continuez!!

Good job

Soumis par Mathieu G. le **Dimanche 30/11/2014 21:44**

Merci Youssef :)

Tres bon travail

Soumis par Yardelaine (non vérifié) le **Vendredi 21/10/2016 04:48**

Merci, j'avais pensé que j'allais jamais comprendre les design patterns. Tres bon instructeur

Exemple

Soumis par Faycal (non vérifié) le **Lundi 21/11/2016 15:42**

Bonjour,

Merci pour votre article mais je n'arrive toujours pas a comprendre le décorateur, pouvez vous ajouté des exemples concret svp ?

Exemple concret proposé

Soumis par Alexis (non vérifié) le **Mardi 07/02/2017 11:57**

Pour l'exemple concret : **http://design-patterns.fr/decorateur-en-java**

Bravo !

Soumis par Adrien (non vérifié) le **Lundi 06/11/2017 15:59**

Plus je parcours ton site, plus je vois la lumière au bout du tunnel !

J'espere que tu présenteras d'autres patterns car tu explique vraiment très bien.

Merci pour ton très bon travail !

Ajouter un commentaire

Votre nom

Courriel

Le contenu de ce champ sera maintenu privé et ne sera pas affiché publiquement.

Sujet

Comment *

Chemin: p

Désactiver le texte riche

Format de texte HTML filtré

Me notifier quand de nouveaux commentaires sont publiés

CAPTCHA

Cette question permet de vérifier que vous n'êtes pas un robot spammeur.

Saisissez la lettre figurant en majuscule dans le mot "patterRns" ? *

Remplissez le champ.

Aperçu