Design pattern Décorateur en Java : vente de desserts

Soumis par Mathieu G. le Dimanche 20/05/2007 17:26 - Dernière modification le Samedi 12/11/2011 21:41

Langage de programmation: Java

Design pattern: **Design pattern Décorateur (decorator)**

Implémentation du pattern Décorateur en Java sur le thème de la vente de desserts.

Description du problème

Afin de mettre en pratique le pattern Décorateur, nous allons concevoir une application qui permet de gérer la vente de desserts. Celle-ci doit permettre d'afficher dans la console le nom complet du dessert choisi et son prix. Les clients ont le choix entre deux desserts : crêpe ou gaufre. Sur chaque dessert ils peuvent ajouter un nombre quelconque d'ingrédients afin de faire leurs propres assortiments. Pour simplifier notre exemple, nous choisirons uniquement deux ingrédients (le chocolat et la chantilly) mais il faut garder à l'esprit que l'ajout de nouveaux ingrédients doit être simplifié. Le système de tarification est simple. Une crêpe (nature) coûte 1.50€ et une gaufre 1.80€. L'ajout de chocolat est facturé 0.20€ et de chantilly 0.50€.

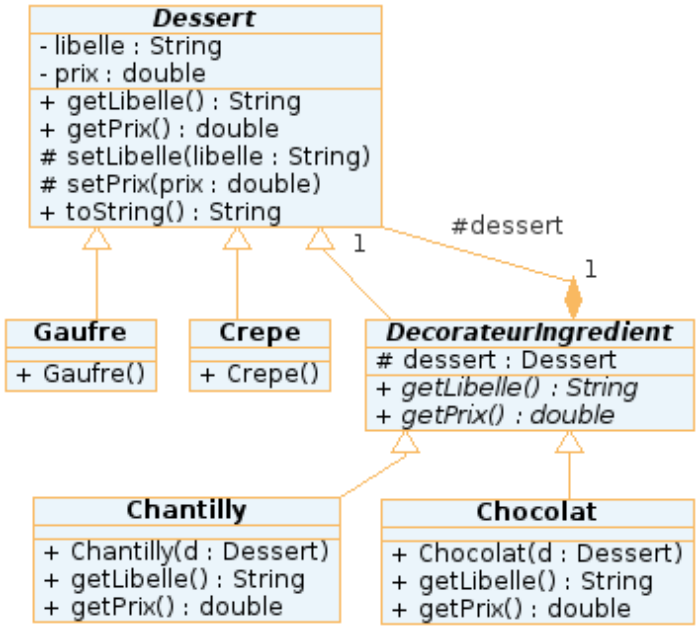
Voyons comment concevoir cette application. Une première idée est de mettre en place une classe abstraite Dessert ayant deux attributs (libelle et prix) et les accesseurs en lecture/écriture correspondants. Puis, créer pour chaque combinaison de desserts et d'ingrédients une classe (CrepeChocolat, CrepeChantilly, GaufreChocolat, GaufreChantilly). Bien sûr cette solution n'est pas évolutive. Si l'on souhaite modifier le prix de l'ingrédient Chocolat on doit le modifier dans deux classes. De plus, si on ajoute une dizaine d'ingrédients nous allons obtenir une centaine de classes.

Une deuxième solution consiste à garder la classe Dessert en la modifiant légèrement. On peut rajouter un booléen pour savoir si l'ingrédient chocolat est ajouté à ce dessert de même pour Chantilly. Puis, on crée des classes Crepe et Gaufre qui héritent de Dessert. Le calcul du prix des ingrédients est effectué dans la classe Dessert auquel on rajoute le prix spécifique du dessert suivant le type de l'objet (Gaufre ou Crepe). Cette solution semble plus satisfaisante mais pose toujours certains problèmes. Si l'on souhaite rajouter un ingrédient, il faut ajouter un attribut dans la classe Dessert et modifier la méthode qui calcule le prix afin de le prendre en considération. De plus, toutes les classes héritant de Dessert posséderont ces attributs qui n'auront pas toujours de sens. Si on crée une classe SaladeDeFruit héritant de Dessert on aura un attribut (hérité de la classe mère) nommé chocolat (bizarre pour une salade de fruit).

Voyons ce qu'apporte le pattern décorateur à notre problématique.

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

Implémentation d'une solution basée sur Décorateur



La solution consiste à utiliser à la fois l'héritage et la composition. Une classe abstraite Dessert regroupe les attributs et les méthodes communes. Puis, des desserts concrets tel que Gaufre et Crepe héritent de cette classe. Dans le constructeur de ces classes on met à jour les attributs défini dans Dessert à l'aide des accesseurs. Afin de gérer les ingrédients, il faut une classe abstraite nommée DecorateurIngredient. Celle-ci possède un Dessert en attribut et oblige la redéfinission de deux méthodes getLibelle() et getPrix(). Chaque ingrédient (Chantilly, Chocolat...) doit hériter de la classe DecorateurIngredient. Le constructeur de ces classes permet d'initialiser l'attribut dessert présent dans la classe mère. De plus, la redéfinition des méthodes getLibelle() et getPrix() va permettre d'ajouter des fonctionnalités. Pour comprendre comment cela fonctionne voyons le code Java correspondant au diagramme UML.

Implémentation de la classe abstraite Dessert

```
// Classe abstraite dessert.
public abstract class Dessert
{
	private String libelle;// Libellé du dessert.
	private double prix;// Prix du dessert.

	// Accesseurs en lecture pour le libellé et le
	public String getLibelle()
	{
		return libelle;
	}
	public double getPrix()
	{
		return prix;
	}

	// Accesseurs en écriture pour le libellé et le
	protected void setLibelle(String libelle)
	{
		this.libelle = libelle;
	}
	protected void setPrix(double prix)
	{
		this.prix = prix;
	}

	// Méthode utilisée pour l'affichage d'un desse
```

```
        public String toString()
        {
                NumberFormat format=NumberFormat.getIns
                format.setMinimumFractionDigits(2);// 2
                return getLibelle()+" : "+format.format
        }
}
```

La classe abstraite Dessert possède deux attributs que sont libelle et prix. Le premier correspond au nom du dessert sélectionné et le deuxième au prix d'achat. Des accesseurs en lecture/écriture permettent d'accéder à ces attributs. Afin de faciliter l'affichage dans la console, nous avons implémenter la méthode toString(). Celle-ci sera appelée automatiquement lors de l'affichage d'un objet de type Dessert dans la console.

Implémentation des classes Gaufre et Crepe

```
// Classe gaufre qui hérite de dessert
public class Gaufre extends Dessert
{
        // Constructeur qui intialise le libellé et le
        public Gaufre()
        {
                setLibelle("Gaufre");
                setPrix(1.80);
        }
}
// Classe crêpe qui hérite de dessert.
public class Crepe extends Dessert
{
        // Constructeur qui initialise le libellé et le
        public Crepe()
        {
                setLibelle("Crêpe");
                setPrix(1.50);
        }
}
```

Les classes Gaufre et Crepe sont des desserts concrets et donc héritent de Dessert. Leurs constructeurs permettent de mettre à jour le libellé et le prix (grâce aux accesseurs en écriture). On constate qu'il sera facile de rajouter un nouveau dessert concret sans modifier notre modèle.

Implémentation de la classe abstraite DecorateurIngredient

```
// Classe abstraite decorateurIngredient qui hérite de
public abstract class DecorateurIngredient extends Dess
{
        protected Dessert dessert;// Dessert sur leuque

        // On oblige les ingrédients à implémenter la m
        public abstract String getLibelle();
        // On oblige les ingrédients à implémenter la m
        public abstract double getPrix();
}
```

DecorateurIngredient va permettre de décorer les desserts avec différents ingrédients. Il s'agit d'une classe abstraite héritant de dessert. Celle-ci possède en attribut le dessert qu'elle va décorer (c'est à dire ajouter des fonctionnalités). A noter, que ce dessert peut correspondre à un dessert déjà décoré puisque celui-ci hérite indirectement de la classe Dessert. Le DecorateurIngredient oblige également la redéfinition des deux méthodes getLibelle() et getPrix() dans ses sous classes.

Implémentation des classes Chantilly et Chocolat

```
//Classe chantily qui hérite de decorateurIngredient et
public class Chantilly extends DecorateurIngredient
{
	// Constructeur qui prend en paramètre le desse
	public Chantilly(Dessert d)
	{
		dessert = d;
	}

	// On affiche le libellé du dessert et on rajou
	public String getLibelle()
	{
		return dessert.getLibelle()+", chantill
	}

	// On additionne le prix du dessert et le prix
	public double getPrix()
	{
		return dessert.getPrix()+0.50;
	}
}
// Classe chocolat qui hérite de decorateurIngredient e
public class Chocolat extends DecorateurIngredient
{
	// Constructeur qui prend en paramètre le desse
	public Chocolat(Dessert d)
	{
		dessert = d;
	}

	// On affiche le libellé du dessert et on rajou
	public String getLibelle()
	{
		return dessert.getLibelle()+", chocolat
	}

	// On additionne le prix du dessert et le prix
	public double getPrix()
	{
		return dessert.getPrix()+0.20;
	}
}
```

Les classes Chantilly et Chocolat correspondent à deux ingrédients qui peuvent être ajoutés aux desserts. Pour cela ces classes héritent de DecorateurIngredient. Leurs constructeurs prennent en paramètre le dessert « nature » qui sera stocké dans l'attribut de DecorateurIngrédient. On note, que l'attribut dessert de la classe mère est déclaré en « protected » ce qui nous permet de se passer des accesseurs. Les méthodes getLibelle() et getPrix() sont redéfinies et ainsi on ajoute des fonctionnalités. Par exemple la méthode getLibelle() affiche le libellé du dessert en rajoutant le libellé de l'ingrédient. Le même principe est utilisé pour la méthode getPrix() pour le calcul du prix.

Implémentation de la classe Main

```
// Classe principale de l'application.
public class Main
{
	// Méthode principale.
	public static void main(String[] args)
	{
		// Création et affichage d'une gaufre a
		Dessert d1 = new Gaufre();
		d1 = new Chocolat(d1);
		System.out.println(d1);
		// Création et affichage d'une crêpe au
		Dessert d2 = new Crepe();
		d2 = new Chocolat(d2);
		d2 = new Chantilly(d2);
```

```
                System.out.println(d2);
        }
}
```

La classe principale de notre application fabrique deux desserts. Une gaufre au chocolat et une crêpe au chocolat et à la chantilly. On affiche alors le libellé de ces desserts et leurs prix.

Résultat

```
Gaufre, chocolat : 2,00€
Crêpe, chocolat, chantilly : 2,20€
```

Comme souhaité on obtient un libellé complet et un prix tenant compte de tous les ingrédients ajoutés. A noter, que l'on peut décorer un dessert avec un nombre quelconque d'ingrédients. Il est même possible d'utiliser plusieurs fois le même ingrédient (une petite crêpe avec un double supplément de chocolat pour les gourmands...).

Conclusion

Le pattern Décorateur basé sur l'héritage et la composition est facile à mettre place. Il permet de répondre parfaitement à la problématique rencontrée lors de la conception de cette application de gestion des desserts. C'est à dire comment concevoir une application évolutive ou l'ajout et la modification de fonctionnalités ne posera pas de problème (explosion du nombre de classe...).

L'inconvénient de ce pattern est la multiplicité des objets qu'il faut créer pour l'utiliser. On peut le remarquer avec le nombre de « new » présent dans la classe Main. Pour résoudre ce problème on pourra combiner Décorateur avec un autre pattern tel que Fabrique ou Monteur.

Code source:

design-pattern-decorateur-java-1.zip

Ajouter un commentaire

Commentaires

Décorateur en Java (vente de desserts)

Soumis par stephane (non vérifié) le **Jeudi 06/11/2008 22:03**

Bonjour, merci pour vos excellents tutoriaux, serait il-possible d'avoir l'exemple de design pattern décorateur associé au design pattern fabrication...

ps la classe abstraite dessert ne possede aucune méthode abstraite, n'en faudrait -il pas au moins une ? meric encore s.h

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Samedi 01/11/2008 10:59**

Bonjour Stéphane, Effectivement, dans cet exemple, la classe abstraite Dessert ne possède aucune méthode abstraite. Il est vrai que certains définissent une classe abstraite comme une classe possédant au moins une méthode abstraite. Pour ma part, je trouve cette définition réductrice et préfère dire qu'une classe abstraite correspond à une classe qui ne peut pas être instancié. Dans mon exemple je ne voulais pas que l'on puisse instancier la classe Dessert

(en effet, on peut commander une gaufre, une crêpe mais pas un dessert car trop vague). Pour autant je n'avais pas de méthode abstraite à ajouter dans cet exemple minimaliste (mais on pourrait très bien en ajouter).

En ce qui concerne le pattern fabrication il fait partie des prochains que je souhaiterai expliquer car il est très utilisé. Merci pour ton message.

Décorateur en Java (vente de desserts)

Soumis par michel (non vérifié) le **Mardi 09/12/2008 05:26**

Salut Merci pour votre explication

s'il Vous plait j'ai un question Pourquoi tu as mis une classe abstraite entre les objets (chantilly,chocolat) et la super-classe dessert ? queller l'utilité ? merci de me repondre

Décorateur en Java (vente de desserts)

Soumis par michel (non vérifié) le **Mardi 09/12/2008 13:13**

Est Ce Que Ce que Est Correct

en Supprime La class abstrait DecorateurIngredient S-V-D :

public class Chantilly extends Dessert

private Dessert dessert ;

public Chantilly(Dessert d)

dessert = d ;

Et la Meme pour Class pour Chocolat

Décorateur en Java (vente de desserts)

Soumis par etudiant (non vérifié) le **Samedi 03/01/2009 14:22**

Merci pour ce super travail. Je suis prêt pour mes partiels !

Décorateur en Java (vente de desserts)

Soumis par Valmont (non vérifié) le **Mardi 01/12/2009 17:38**

Pourquoi on ne fait pas une super-classe Ingredient et des sous-classe comme Chocolat , Chantilly etc... Puis une super-classe Dessert avec comme attribut une liste d'ingrédients et des sous classe comme Crepe , Gateau etc.. ? Les ingrédients ayant comme attributs un nom et un prix.

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Samedi 05/12/2009 17:43**

Merci pour ta remarque Valmont elle est très intéressante.

Valmont propose d'avoir une classe Ingredient (avec ses sous-classes Chantilly et Chocolat) puis une classe Dessert (avec ses sous-classes Gaufre et Crepe) qui aurait comme attribut une liste d'ingrédients.

Cette solution est intéressante mais j'ai 2 exemples en tête qui montre l'avantage du pattern Décorateur.

1. Si l'on considère que l'application a d'abord été développée pour gérer uniquement des desserts simples (c'est à dire des gaufres et crêpes natures). Puis qu'à un moment donné il a fallut rajouter des garnitures aux desserts grâce aux ingrédients. Dans la solution proposée par Valmont il faut modifier la classe Dessert pour rajouter un attribut (tableau d'ingrédients) et modifier l'ensemble des méthodes pour prendre en compte ce nouvel attribut. On sait que cela est dangereux car on modifie des fonctionnalités existantes et on risque donc d'ajouter des bugs. Alors qu'avec le pattern Décorateur il suffit de créer une nouvelle classe DecorateurIngredient (et ses sous-classes) afin d'ajouter des fonctionnalités sans modifier celles existantes. N'ayant pas modifié la gestion d'une gaufre nature on a pas pu insérer de bug à ce niveau.

2.En utilisant l'opérateur instanceof dans l'application il sera très facile de savoir par exemple si un Dessert contient du Chocolat. Pour rappel instanceof permet de tester si un objet est une instance (directe ou indirecte) d'une certaine classe ou interface. Pour tester si un dessert contient du Chocolat avec la méthode proposée par Valmont on doit créer une méthode dédiée dans Dessert et faire une boucle sur la liste des ingrédients. Dommage car avec le pattern Décorateur on a rien à faire...

J'espère t'avoir convaincu de l'avantage de ce pattern.

Décorateur en Java (vente de desserts)

Soumis par greg (non vérifié) le **Lundi 07/12/2009 15:51**

Bonjour,

Pour reprendre l'utilisation d' "instanceof" sur un objet "décoré".

Prenons un morceaux de ton code de la classe Main, tel qu'il est dans l'exemple :

```
Dessert d1 = new Gaufre() ;
d1 = new Chocolat(d1) ;
```

Le problème avec le design pattern Decorator, c'est que tu ne peux pas récupérer le type Gaufre.

```
if(d1 instanceof Gaufre) // Retourne faux.
if(d1 instanceof Chocolat) //Retourne vrai.
```

Question : Quel design pattern appliquer lorsque tu effectues des traitements différents en fonction du type ?

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Mardi 08/12/2009 22:25**

Bonjour Greg,

Merci pour ton message car tu pointes du doigt la principale contre indication à l'utilisation du pattern Décorateur.

En effet, si le code existant effectue des tests sur les types concrets (tels que Gaufre ou Crepe) il ne faut pas utiliser le pattern Décorateur car on risque de perturber le fonctionnement

de l'application. Lorsqu'un Dessert comme une Gaufre est décoré par un ingrédient tel que du Chocolat alors l'objet obtenu est de type concret Chocolat avec pour types abstraits DecorateurIngredient et Dessert (l'objet de type concret Gaufre étant encapsuler dans un attribut de l'objet Chocolat).

Difficile de répondre à ta question car cela va dépendre des cas. Si tu développes l'application de A à Z tu peux certainement éviter de tester les types de concret (ce qui au passage n'est jamais bon) et donc utiliser le pattern Décorateur. Si ce n'est pas le cas tu peux essayer d'utiliser un autre pattern structurel si il y en a un qui correspond à tes besoins. Sinon il faut chercher une autre solution qui risque de se réduire à modifier la classe abstraite Dessert. **Mais la faute ne revient pas au pattern Décorateur mais plus au développeur qui a utilisé des tests sur des types concrets alors que cela est fortement déconseillé.**

Décorateur en Java (vente de desserts)

Soumis par Anonyme (non vérifié) le **Mercredi 06/01/2010 14:59**

Bonjour et bonne année !

Un grand merci pour ce tuto et bravo pour ce grand sens pédagogique ...

Après avoir créé nos classe une fois dans la classe de test :

Si j'ai bien compris on crée un dessert nature, on initialise un premier décore (en lui donnant comme param le dessert nature) et on stock ceci dans le dessert nature même (la case mémoire qui s'adressait au dessert nature désormais s'adresse à ce nouvel dessert) ...

Pour la gouffre on s'arrête là mais pour la crêpe on continue toujours de même manière..

est-ce bien ça ou je suis à côté de la plaque ??!!

en gros si l'on ne reprenait pas la même variable ce serait comme ça :

```
Dessert d1 = new Gaufre() ;
Dessert dchoco = new Chocolat(d1) ;
System.out.println(dchoco ) ;
Dessert d2 = new Crepe() ;
Dessert dcrep = new Chocolat(d2) ;
Dessert dcrep2 = new Chantilly(dcrep) ;
System.out.println(dcrep2 ) ;
```

c'est ça ??!!

merci de me répondre

une passionnée de Java :-))

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Jeudi 07/01/2010 21:20**

Bonjour et meilleurs vœux.

Merci pour ton commentaire.

Je pense que tu as bien compris le fonctionnement du pattern Décorateur.

Dans mon exemple :

> *La variable d1 pointe vers un objet de type Chocolat qui contient dans son attribut dessert un objet de type Gaufre. Quand à la variable d2 elle pointe vers un objet de type Chantilly qui contient dans son attribut dessert un objet de type Chocolat qui possède également un attribut dessert contenant un objet de type Crepe.*

Dans ton exemple :

> *d1 est un objet de type Gaufre.*
> *dchoco est un objet de type Chocolat qui contient dans son attribut dessert un objet de type Gaufre.*
> *d2 est un objet de type Crepe.*
> *dcrep est un objet de type Chocolat qui contient dans son attribut dessert un objet de type Crepe.*
> *dcrep2 est un objet de type Chantilly qui contient dans son attribut dessert un objet de type Chocolat qui possède également un attribut dessert contenant un objet de type Crepe.*

En espérant avoir clarifié ce point.

Décorateur en Java (vente de desserts)

Soumis par Anonyme (non vérifié) le **Vendredi 08/01/2010 01:40**

Bonjour,

Merci de ta réponse.

J'ai essayé sur Eclipse, la version du code que j'avais envoyé, le résultat est la même que la tienne. Est-ce que c'est par hasard ou est-ce que ces deux version disent la même chose ?

... et une petite question encore :

Pour l'utilisation de ce pattern comment on choisi les rôle ? Pour l'exemple de dessert est-ce que par exemple dans un autre scénario on peut inverser les rôles : une classe DecorateurBase aura les classes Crepe et Gouffre qui l'étendent et qui prennent comme param un objet du type Chantilly ou du type Chocholat qui étendent du classe abstrait Ingrediant (DecorateurBase extends Ingrediant et aura un attribut du type Ingrediant)

enfin les rôles inversés mais toujours la vedette est **l'objet** qui aura comme param un autre objet.

oui ou non ?!

merci de me répondre

pationnée du java

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Samedi 09/01/2010 19:16**

Bonjour,

C'est normal que tu as le même résultat que dans mon exemple car tu

affiches les variables dchoco (qui correspond à ma variable d1) et dcrep2 (qui correspond à ma variable d2). La seule différence c'est que tu as conservé dans d'autres variables des objets intermédiaires (par exemple une crêpe au chocolat dans dcrep).

Pour répondre à ta deuxième question cela dépend du scénario. C'est le scénario qui permet de définir quels sont les objets concrets et quels sont les décorateurs. De manière générale les objets concrets peuvent exister sans décorateurs alors qu'un décorateur sans objet concret n'a pas de sens. Par exemple on peut vendre une crêpe nature mais pas que de la Chantilly.

Si tu as encore des doutes merci de me détailler ton scénario.

Décorateur en Java (vente de desserts)

Soumis par Anonyme (non vérifié) le **Dimanche 10/01/2010 16:29**

Bonjour,

Merci beaucoup de ta réponse et de cette clé magique :

> *De manière générale les objets concrets peuvent exister sans décorateurs ....*

Savoir ça, aide beaucoup à établir son scénario ;-)

Sur ce je m'en vais continuer avec ton tuto Oservateur !

merci encore

Décorateur en Java (vente de desserts)

Soumis par nicolas (non vérifié) le **Mardi 12/01/2010 19:26**

Bonjour Matthieu,

merci bien pour cet exemple.

J'ai un problème avec la terminologie, cependant, qui a constitué un petit obstacle dans ma compréhension du pattern.

Dans la sémantique de l'exemple, "chocolat" n'est pas un "dessert", mais un "dessert (quelconque) décoré de chocolat".

De façon plus abstraite, la classe "Decorateur" du pattern ne me semble pas être pas un "Composant", mais un "Composant décoré", ou encore, la classe "Decorateur Concret" ne me semble pas être un "Composant", mais un "Composant (quelconque) décoré de decorateur concrêt".

Ne serait il pas plus clair d'appeler : "decorateur" avec "composant décoré" et "decorateur concrêt" avec "composant décore de qqchose"

Un autre exemple courant du pattern est celui relatif à une hiérarchie de véhicules, dans laquelle j'ai déjà vu une classe-décoratrice nommée "Galerie de toit". Or... Une "Galerie de toit" n'est pas un véhicule ! Pourquoi donc ne pas l'appeler "véhicule avec galerie" ?

Que pensez vous de ces quelques considérations terminologiques ?

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Dimanche 17/01/2010 22:20**

Bonjour Nicolas,

Tout d'abord merci pour ton message.

Il est vrai que le terme Chocolat est un peu ambigu car on peut penser qu'il s'agit d'un dessert à part entière. Or dans mon exemple il s'agit seulement d'un ingrédient (il ne peut pas être vendu seul). Il faut s'imaginer une petite boutique qui vends des crêpes et des gaufres avec dessus des ingrédients aux choix.

Concernant la terminologie utilisée j'essaye de conserver la description officielle des diagrammes des patterns en les traduisant en français. Par exemple tu retrouveras un diagramme similaire ici : **http://en.wikipedia.org/wiki/Decorator_pattern**

Décorateur en Java (vente de desserts)

Soumis par Jean_Guy (non vérifié) le **Jeudi 28/01/2010 01:35**

Bonjour,

Très bonne explication du pattern et sympathique qui plus est.

Cependant je me posais une question , par exemple si chocolat devient une classe abstraite par exemple et que celle-ci est la généralisation de chocolat blanc, chocolat noir et chocolat au lait par exemple. Comment cela pouvait s'implémenter ?

Une seconde, si mon objet devient de type concret chantilly et si chantilly possède une méthode public qui lui est propre, pourquoi ne pourrais je pas accéder à celle-ci ?

Encore merci et bonne continuation.

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Dimanche 07/02/2010 17:46**

Bonjour Jean_Guy,

Merci pour ton message et désolé pour ma réponse un peu tardive.

Il est tout à fait possible de transformer la classe Chocolat en classe abstraite qui servira de modèle à des classes concrètes ChocolatNoir, ChocolatBLanc... L'implémentation ne devrait pas être délicate. Il faut juste penser que les classes concrètes comme ChocolatNoir devront implémenter toutes les méthodes abstraites de DecorateurIngredient et de Chocolat.

Pour répondre à ta deuxième question on peut très bien ajouter des méthodes spécifiques à la classe Chantilly. Mais attention a bien tester le type concret de l'objet avant d'appeler ces méthodes spécifiques. Dans la pratique, ce cas est assez peu fréquent et lorsque cela est possible on préfère rajouter une méthode commune à tous les dessers (ce qui évite de devoir tester les types).

A bientôt.

Décorateur en Java (vente de desserts)

Soumis par OuT (non vérifié) le **Jeudi 24/06/2010 06:22**

Bonjour, merci pour cet article qui est très clair :)

Je suis actuellement confronté à un problème de traitement différent selon le type de l'objet (ce fameux *instanceof*), et je confirme que c'est effectivement un truc à ne pas faire !

Ce qui amène à ma question, quelles sont les techniques habituelles pour éviter d'avoir à tester le type ? Je sais que cette question est un peu vague... Pour l'instant, tout ce que je vois comme solution, c'est de s'assurer du type en amont dans l'application, mais cela me parait un peu léger. Donc voila, si tu as des patterns ou autre permettant de pallier à ce problème classique, cela m'intéresserait beaucoup

à plus

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Mardi 29/06/2010 21:53**

Bonjour et merci pour tes encouragements.

Effectivement, il faut de manière générale éviter d'effectuer des tests sur les types concrets et préférer les tests sur les types abstraits (classe abstraite ou interface). Un exemple simple, si on test le type concret d'une classe il n'est alors plus possible d'hériter de celle-ci pour ajouter des comportements sans risquer de perturber le bon fonctionnement de l'application.

Pour éviter de tester les types concrets il suffit parfois de faire preuve de bon sens. Pour les cas les plus délicats, on peut utiliser les patterns de fabrication et plus particulièrement les patterns fabrique et fabrique abstraite. Cela tombe bien car je suis justement entrain de rédiger un article sur ces 2 patterns. N'hésite pas à t'abonner au Flux RSS ou à la Newsletter pour être tenu informé quand je les mettrais en ligne.

A bientôt.

Observateur en Java (positionnement via un GPS)

Soumis par Bilel (non vérifié) le **Dimanche 09/10/2011 13:16**

Bravooo, je vous remercie pour l'effort que vous faites, je n'ai jamais compris les designs pattern pareil ! voilà de quoi on avait besoin, des exemples qui t'illustrent parfaitement le problème et la solution. Grand Merci.

Observateur en Java (positionnement via un GPS)

Soumis par Mathieu G. le **Dimanche 09/10/2011 18:52**

Merci pour vos encouragements. Un article sur le pattern Fabrique devrait être publié dans les prochaines semaines, n'hésitez pas à vous inscrire à la newsletter ou au flux RSS.

Décorateur en Java (vente de desserts)

Soumis par guilamb (non vérifié) le **Lundi 07/11/2011 17:55**

Merci pour cet article de qualité !

Juste une remarque, le mot "Gauffre" ne prend qu'un 'f' -> Gaufre.

;)

Décorateur en Java (vente de desserts)

Soumis par Mathieu G. le **Mardi 08/11/2011 22:40**

Merci pour ton commentaire et d'avoir relevé cette coquille.

Je vais la corriger dès que possible.

A bientôt.

Question Pattern décorateur

Soumis par Yoann (non vérifié) le **Mardi 02/10/2012 21:49**

Bonjour,

Un grand merci pour ce tuto très bien fait ! J'ai une question sur le pattern Décorateur.

Un pattern décorateur permet d'ajouter dynamiquement des comportants à un objet mais peut t-il en enlever ? Je m'explique :p

Soit une crèpe au chocolat et à la chantilly : Dessert dessert = new Chocolat(new Chantilly (new Crepe())

Peut-ont décider d'enlever dynamiquement la chantilly sur notre dessert ? Comment peut-on le faire ?

Question Pattern décorateur

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

Bonjour,

C'est une très bonne question Yoann.

Avec le pattern décorateur il n'est pas aisé de retirer dynamiquement des responsabilités à un objet puisque elles sont imbriqués les unes dans les autres (un peu à la manière de poupées russes).

Pour cela le plus simple reste peut-être d'inspecter l'objet et d'en reconstruire un nouveau avec seulement les responsabilités souhaitées.

Peut-être que quelqu'un aura une autre piste à proposer ?

Différentes fonctionnalités pour les ingrédients

Soumis par Thibault (non vérifié) le **Vendredi 21/06/2013 15:45**

Bonjour, et merci pour cet exemple très clair!

Je me trouve actuellement face un problème assez proche de celui de votre exemple. Je pense que l'utilisation d'un patron décorateur facilitera la maintenance et favorisera les possiblités d'étendre le code.

Cependant, il y a quelque chose que j'ai des difficultés à réaliser. Imaginons que les différents ingrédients chocolats,chantilly,... aient en commun une méthode qui peut différer selon un choix. Par exemple, on aurait un résultat différent au getPrix() selon si la chantilly est surgelée ou non. On aurait donc en commun le fait que les ingrédients peuvent être surgelés ou non et que le prix varie en fonction de ça.

La solution d'avoir différentes classes chantillySurgelee, chantillyFraiche, etc... ne me paraît pas une bonne solution si on veut factoriser le code getPrixSurgele / getPrixFrais.

Je ne sais pas si j'ai été clair, donc je vais essayer de formuler les choses autrement, toujours avec l'exemple du dessert:

- j'ai une méthode getPrix qui peut avoir **différentes implémentations** (getPrixSurgele, getPrixFrais, ...)

- cette méthode peut être **commune à plusieurs ingrédients** (pas nécessairement tous, disons par exemple que la chantilly ne peut pas être surgelée, mais que le chocolat et la glace peuvent être les deux)

Voyez-vous une solution réalisable et facilement maintenable/extensible à mon problème (si je l'ai clairement expliqué, n'hésitez pas à me demander des détails...)

Merci beaucoup!

Différentes fonctionnalités pour les ingrédients

Soumis par Mathieu G. le **Lundi 01/07/2013 21:47**

Bonjour Thibault et merci pour cette problématique intéressante.

A première vue je vois 2 solutions à te proposer suivant le contexte :

- Ajouter un attribut aux ingrédients permettant de faire varier le prix suivant un paramètre (ici Frais ou Surgelée). Cette solution est viable si ce paramètre à un sens pour tous les ingrédients (y compris les futurs) et que le nombre de paramètres faisant varier le prix est fixe et peu important (sinon le nombre d'attributs va vite devenir problématique).

- Une autre solution est d'appliquer le pattern Décorateur au niveau des produits (comme expliqué ci-dessus), mais également au niveau des ingrédients. Les ingrédients pourront alors être décorés à leur tour et bénéficier de nouvelles fonctionnalités dynamiquement. Attention toutefois que cette solution apporte vraiment un plus car elle va complexifier le modèle de classe.

J'espère que ces 2 pistes te seront utile.

Bonjour,

Soumis par Thibault (non vérifié) le **Mardi 02/07/2013 13:46**

Bonjour,

La solution pour laquelle j'ai opté se rapproche de ta première proposition. J'ai allié un patron stratégie à mon patron décorateur. Dans cet exemple, le dessert contiendrait alors un champ stratégie et une méthode getPrix. Le champ stratégie contenant un pointeur vers une classe abstraite permettant d'implémenter les différents getPrix, et la méthode getPrix un appel à cette stratégie.

Merci pour ta réponse! =)

comment resoudre instanceof

Soumis par Igolus (non vérifié) le **Jeudi 13/11/2014 22:10**

Bonjour,

Pour gerer le probleme d'instanceof, pourquoi ne pas creer une methode public dans la classe dessert isInstanceOf(Class c). L'objet servant la construction du decorateur est conserve dans une variable membre qui sera null lors de la premiere construction du dessert, a force de decoration on conserve la hierarchie des objets.

il suffit alors d'implementer la methode isInstanceOf comme methode concrete dans la classe abstraite testant la classe passer en parametre juqu'a la trouver ou quelle soit null.

Imaginons qu'un programme dont l'implementation est inconnue vous donne une liste de desserts et que suivant les type de dessert, il faille les transporter d'un facon differrente : Tout ce qui contient du chocolat + les toutes les crepes en camion frigo, le reste en camion normal.

Il faudra bien interroger les objets sur leur type d'instance. pour les classer.....

```
public abstract class Dessert
{
        private String libelle;// Libellé du
dessert.
       protected Dessert originalDessert = null;

        public Dessert() {}
        public Dessert(Dessert d) {originalDessert = d;}

       public isInstanceOf(Class c) {
          Dessert current = this;
          while (current != null) {
            if (current instanceof c) {
              return true;
            }
            current = originalDessert ;
          }
          return false;
       }
       .....
}

public class Chocolat extends DecorateurIngredient
{
       //PLUS BESOIN DE CONSTRUCTEUR !!!!

        // On affiche le libellé du dessert et on
rajoute le libellé de l'ingrédient chocolat.
        public String getLibelle()
        {
                return originalDessert.getLibelle()+",
chocolat";
        }

        // On additionne le prix du dessert et le
prix de l'ingrédient chocolat.
        public double getPrix()
        {
```

```
                return originalDessert.getPrix()+0.20;
        }
}
```

comment resoudre instanceof

Soumis par Mathieu G. le **Dimanche 30/11/2014 22:17**

Bonjour Igolus,

Ton idée est intéressante, attention cependant car tu proposes de créer un constructeur dans la classe Dessert alors que celle-ci est abstraite (et donc que par définition elle ne peut pas être instanciée).

Pour ma part j'essaye d'éviter autant que possible de développer du code qui va s'appuyer sur des types concrets car celui-ci devient dangereux en cas de modification.

Par exemple si un autre développeur utilise l'héritage pour spécialiser le Chocolat (blanc, au lait ou noir), cela risque de causer des dysfonctionnements si le code basé sur les types concrets n'est pas modifié.

Design pattern caché

Soumis par David (non vérifié) le **Lundi 02/02/2015 00:28**

Bonjour,

Bon exemple pour illustrer ce pattern !

Y aurait-il un autre pattern de caché dans le diagramme de classe ? Je pense à celui du composite.

Design pattern caché

Soumis par Mathieu G. le **Mercredi 18/03/2015 23:22**

Bonjour David,

Le design pattern décorateur utilise le principe de la composition puisque un DecorateurIngredient est composé d'ingrédients.

En revanche, le design pattern composite n'est pas utilisé dans cet exemple (bien que lui aussi utilise le principe de la composition).

Le design pattern composite permet de traiter de manière identique des objets feuilles et des objets issus de combinaisons.

Or ici on ne traite pas de la même manière les desserts et les ingrédients, par exemple les ingrédients n'ont pas la méthode `setLibelle()`.

vente de dessert

Soumis par souha (non vérifié) le **Vendredi 27/11/2015 10:32**

salut

est ce vous pouvez m'aider de resoudre ce probléme sans utilisation de patron de conception Decorator , meme si la solution favorable

c'est celle ci mais je veut voir la défference .

Merci en avance.

vente de dessert

Soumis par Mathieu G. le **Vendredi 04/12/2015 22:06**

Bonjour Souha,

Si vous avez un problème de conception et que vous hésitez à utiliser le design pattern Décorateur je vous invite à me contacter directement par mail.

explication

Soumis par Arnaud (non vérifié) le **Jeudi 16/03/2017 18:26**

bonjour je debute vraiment dans l'informatique et je sais pas comment je dois compiler .

J'ai donc 7 fichier different (.java) et je sais pas comment je dois faire pour executer.

Dois je regrouper les 7 fichiers dans 1 seul et le compiler ? ou laisser 7 fichier different et compiler juste le Main ? . Je sais pas du tout comment faire

Merci beaucoup

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 minuscule dans le mot "SINGLEtON" ? *

Remplissez le champ.

Aperçu