Design patterns

Définition

Un Design Pattern est une solution à un problème récurrent dans la conception d'applications orientées objet. Un patron de conception décrit alors la solution éprouvée pour résoudre ce problème d'architecture de logiciel. Comme problème récurrent on trouve par exemple la conception d'une application où il sera facile d'ajouter des fonctionnalités à une classe sans la modifier (voir la solution du Design Pattern Visiteur). A noter qu'en se plaçant au niveau de la conception les Design Patterns sont indépendants des langages de programmation utilisés.

Représentation d'un patron de conception

Les Design Patterns sont représentés par :

Nom : qui permet de l'identifier clairement

Problématique : description du problème auquel il répond

Solution : description de la solution souvent accompagnée d'un schéma UML

Conséquences : les avantages et les inconvénients de cette solution

Organisation des patrons de conception

Les patrons de conception sont classés en trois catégories.

Création : ils permettent d'instancier et de configurer des classes et des objets.

Structure : ils permettent d'organiser les classes d'une application.

Comportement : ils expliquent comment organiser les objets pour qu'ils collaborent entre eux.

Création

Le design pattern Fabrique (Factory Method) définit une interface pour la création d'un objet en déléguant à ses sous-classes le choix des classes à instancier.

Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

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

Comportement
Le pattern Observateur (en anglais Observer) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d’état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

Design pattern Fabrique (Factory Method)
Catégorie: Création

Fréquence d'utilisation: Forte

Difficulté: Intermédiaire

Le design pattern Fabrique (Factory Method) définit une interface pour la création d'un objet en déléguant à ses sous-classes le choix des classes à instancier.

Description du problème

Il est fréquent de devoir concevoir une classe qui va instancier différents types d'objets suivant un paramètre fourni. Par exemple une usine va fabriquer des produits en fonction du modèle qu'on lui indique.

L'idée la plus simple pour répondre à ce besoin est d'écrire une succession de conditions qui suivant le modèle demandé, instancie et retourne l'objet correspondant.

Le problème avec cette implémentation, c'est que la classe correspondant à l'usine va être fortement couplée à tous les produits qu'elle peut instancier car elle fait appel à leurs types concrets.

Or ce code va être amené à évoluer régulièrement lors de l'ajout de nouveaux produits à fabriquer ou de la suppression de certains produits obsolètes.

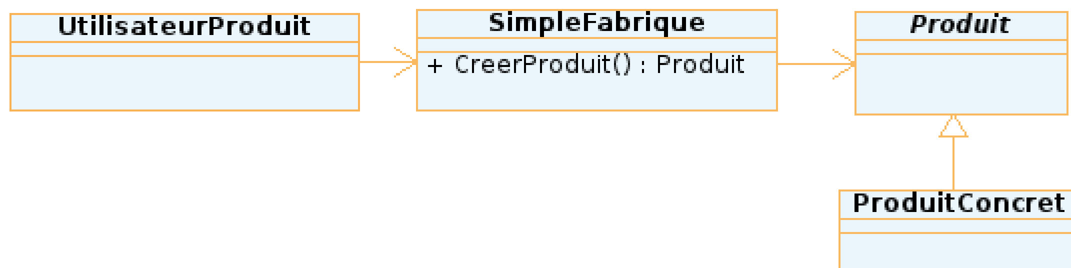
De plus, il est fort probable que l'instanciation des différents produits soit également réalisée dans d'autres classes par exemple pour présenter un catalogue des produits fabriqués.

On se retrouve alors avec du code fortement couplé, qui risque d'être dupliqué à plusieurs endroits de l'application.

Début de solution

La première solution est de regrouper l'instanciation de tous les produits dans une seule classe chargée uniquement de ce rôle. On évite alors la duplication de code et on facilite l'évolution au niveau de la gramme des produits.

Cette solution appelée Fabrique Simple est une bonne pratique de conception mais pas un design pattern. En effet, le design pattern Fabrique est plus évolué et offre plus de flexibilité, donc attention à ne pas les confondre. Pour autant cette bonne pratique est régulièrement utilisée et s'avère efficace dans les cas les plus simples (voir son diagramme UML ci-dessous).

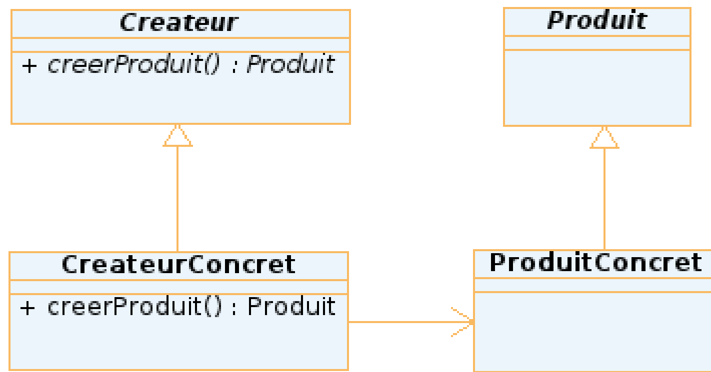


L'utilisateur du produit fait appel à la Fabrique Simple pour obtenir un Produit. C'est la Fabrique Simple qui est chargée d'instancier et de retourner le Produit Concret attendu (par exemple grâce à un paramètre passé en argument de la fonction). L'utilisateur du produit est donc fortement couplé uniquement à la Fabrique Simple et non à tous les produits qu'il prend en charge.

Si par la suite l'entreprise évolue et a besoin de plusieurs usines, chacune spécialisée dans la fabrication de certains produits, Fabrique Simple ne va plus suffire.

Dans ce cas il faut prévoir l'utilisation du design pattern Fabrique dont le diagramme UML est présenté ci-dessous.

Diagramme UML



Définition de la solution

Le créateur contient toutes les méthodes permettant de manipuler les produits exceptée la méthode creerProduit qui est abstraite. Les créateurs concrets implémentent la méthode creerProduit qui instancie et retourne les produits. Chaque créateur concret peut donc créer des produits dont il a la responsabilité. Pour finir tous les produits implémentent la même interface afin que les classes utilisant les produits (comme le créateur) puissent s'y référer sans connaître les types concrets.

Suivant les besoins, l'interface Produit peut être une classe abstraite. Il est également bénéfique d'utiliser ce pattern même si l'on a qu'un seul CreateurConcret ou qu'un CreateurConcret n'instancie qu'un seul Produit car les avantages liés au découplage des produits et du créateurs sont conservés.

A noter que les créateurs concrets utilisent la bonne pratique Fabrique Simple présentée précédemment. Mais comparé à cette bonne pratique qui ne sert qu'une fois, le design pattern Fabrique permet de créer une structure où l'ajout d'une sous classe (c'est à dire un créateur concret) permet de choisir les produits qui seront utilisés. C'est d'ailleurs ce qui explique la définition de ce pattern.

Conséquences

Le pattern Fabrique doit être utilisé pour découpler les clients des classes concrètes à instancier, ou si l'on ne connaît pas d'avance toutes les classes concrètes à instancier.

Pour créer des familles de produits cohérentes, on sera amené à utiliser le design pattern Fabrique Abstraite qui répond spécifiquement à ce besoin.

Design pattern Singleton

Catégorie: Création

Fréquence d'utilisation:Forte

Difficulté:Facile

Singleton garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.

Description du problème

Certaines applications possèdent des classes qui doivent être instanciées une seule et unique fois. C'est par exemple le cas d'une classe qui implémenterait un pilote pour un périphérique, ou encore un système de journalisation. En effet, instancier deux fois une classe servant de pilote à une imprimante provoquerait une surcharge inutile du système et des comportements incohérents.

On peut alors se demander comment créer une classe, utilisée plusieurs fois au sein de la même application , qui ne pourra être instancié qu'une seule fois ?

Une première solution, régulièrement utilisée, est d'instancier la classe dès le lancement de l'application dans une variable globale (c'est à dire une variable accessible depuis n'importe quel emplacement du programme). Cependant cette solution doit être évitée car en plus d'enfreindre le principe d'encapsulation elle comporte de nombreux inconvénients. En effet, rien ne garantit qu'un développeur n'instanciera pas une deuxième fois la classe à la place d'utiliser la variable globale définie. De plus, on est obligé d'instancier les variables globales dès le lancement de l'application et non à la demande (ce qui peut avoir un impact non négligeable sur la performance de l'application). Enfin, lorsqu'on arrive à plusieurs centaines de variables globales le développement devient rapidement ingérable surtout si plusieurs programmeurs travails simultanément.

Mais si on n'utilise pas ce stratagème comment faire ? C'est simple il suffit d'utiliser le design pattern Singleton.

Définition de la solution

L'objectif est d'ajouter un contrôle sur le nombre d'instances que peut retourner une classe.

La première étape consiste à empêcher les développeurs d'utiliser le ou les constructeur(s) de la classe pour l'instancier. Pour cela il suffit de déclarer privé tous les constructeurs de la classe. Attention dans certains langages une classe sans constructeur possède un constructeur implicite par défaut (c'est notamment le cas de Java). Il faut donc que celui-ci soit déclaré explicitement en privé.

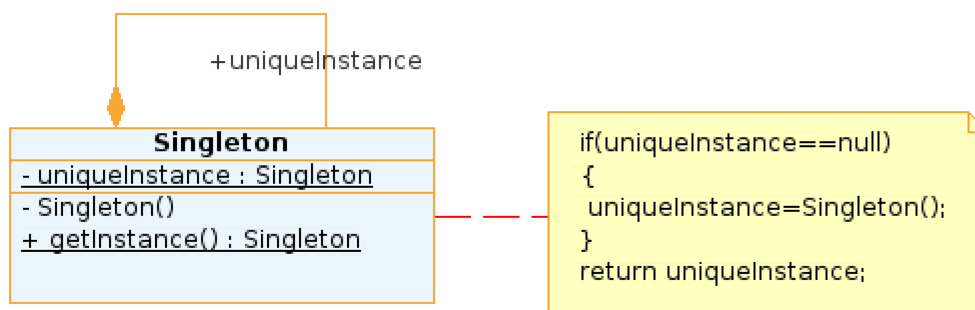
Une fois cette étape accomplie, il est possible d'instancier cette classe uniquement depuis elle même, ce qui n'a pas beaucoup de sens. Comment allons nous faire pour permettre aux développeurs de l'utiliser ?

Nous allons construire un pseudo constructeur. Pour cela il faut déclarer une méthode statique qui retournera un objet correspondant au type de la classe. L'avantage de cette méthode par rapport à un constructeur, est que l'on peut contrôler la valeur que l'on va retourner. Le fait que cette méthode soit déclarée statique permet de l'appeler sans posséder d'instance de cette classe. A noter que, par convention, ce pseudo constructeur est nommé getInstance.

Pour en finir avec le concept de base du Singleton voyons comment implémenter cette méthode.

Tout d'abord il faut créer un attribut statique qui va permettre de stocker l'unique instance de la classe. Ensuite, dans le pseudo constructeur on va tester cet attribut. Si celui-ci est nul alors on crée une instance de la classe et on stocke sa valeur dans cet attribut. Sinon c'est que l'attribut possède déjà une instance de la classe. Dans tous les cas la méthode retourne la valeur de l'attribut possédant l'unique instance de la classe.

Diagramme UML



Extension : Multiton

Une variante du Singleton existe elle est nommée Multiton. Même si celle-ci est régulièrement utilisée il ne s'agit pas ici d'un design pattern officiel du Gang of Four. Comme nous venons de la voir le pattern Singleton garantit qu'une classe n'a qu'une seule instance. Le Multiton reprend ce principe en garantissant qu'il existe une seule instance par clé pour une classe. Ainsi il est par exemple possible de créer une classe qui possédera deux instances identifiées par deux clés différentes.

Pour cela on remplace l'attribut uniqueInstance par un tableau associatif (il s'agit d'un type abstrait de données composé d'un ensemble fini de clefs et d'un ensemble fini de valeurs, où chaque clef est associée à une valeur) ou table de hashage . On associe alors une clé à une instance. Puis on transforme la fonction getInstance qui prend maintenant en paramètre une clé. Si cette clé existe dans le tableau associatif, on retourne l'instance correspondante sinon on crée une nouvelle instance associée à la clé que l'on ajoute à uniqueInstance. Ainsi on peut facilement gérer un ensemble d'instances avec leurs clés associées.

Conséquences

Jusqu'à maintenant nous avons passé sous silence un problème qui peut mettre en péril l'implémentation du pattern Singleton : le multithread (capacité pour un programme de lancer plusieurs traitements simultanés c'est à dire processus). En effet, si l'on implémente de façon basique le pattern Singleton, dans le cas d'un programme multithread, on peut se retrouver avec une classe Singleton possédant plusieurs instances.

Le problème réside dans l'enchaînement des instructions. Un premier processus exécute la fonction getInstance constate que l'attribut uniqueInstance est nul. Un deuxième processus s'exécute et lui aussi constate (via getInstance) que l'attribut uniqueInstance est nul. Il va donc créer une instance et retourner celle-ci. Lorsque le premier processus va reprendre son exécution il va à son tour créer une nouvelle instance (étant donné qu'il a déjà effectué le test sur l'attribut) et retourner celle-ci. On se retrouve alors avec deux instances pour une classe Singleton.

Pour résoudre ce problème, on dispose de deux solutions. La plus simple consiste à instancier l'attribut uniqueInstance dès sa déclaration dans la classe. Ainsi l'implémentation de la méthode

getInstance se limite à retourner l'attribut uniqueInstance. Cette approche est fiable et simple à mettre en place mais on perd l'instanciation à la demande.

L'autre approche consiste à s'assurer que la fonction getInstance ne pourra être exécutée que par un seul processus à la fois. Chaque langage dispose de sa spécification pour indiquer la particularité de cette méthode. Par exemple, en Java on utilisera le mot clef « synchornized ». Cette solution bien que satisfaisante, peut réduire les performances d'un programme multithread. Il existe alors des stratagèmes suivant les langages pour pallier à cette lacune.

On peut alors se demander quand doit-on mettre en place ces solutions spécifiques aux multithread. Le meilleur conseil est de toujours implémenter le pattern Singleton comme si le programme utilisait le multithread. Car même si ce n'est pas le cas aujourd'hui, il se peut que dans le futur le multithread apparaisse dans votre application.

Design pattern Décorateur (decorator)

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

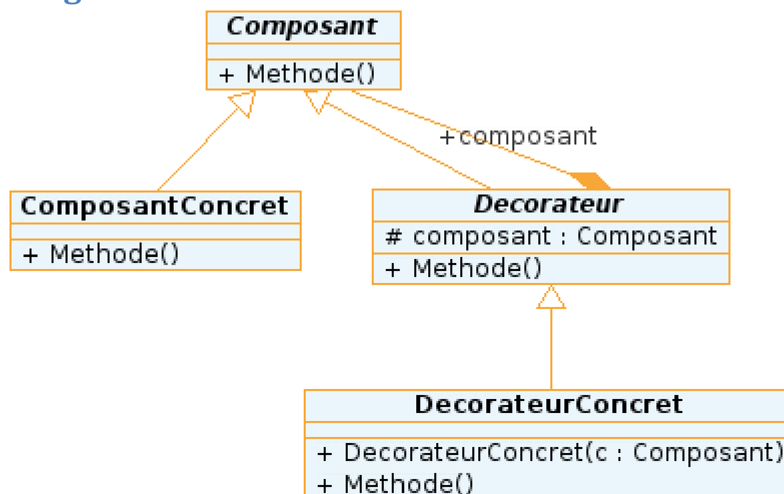
Dans la programmation orientée objet, la façon la plus classique d'ajouter des fonctionnalités à une classe est d'utiliser l'héritage. Pourtant il arrive parfois de vouloir ajouter des fonctionnalités à une classe sans utiliser l'héritage. En effet, si l'on hérite d'une classe la redéfinition d'une méthode 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



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.

Design pattern Observateur (observer)

Catégorie: Comportement

Fréquence d'utilisation: Forte

Difficulté: Facile

Le pattern Observateur (en anglais Observer) définit une relation entre objets de type un-à-plusieurs, de façon que, si un objet change d'état, tous ceux qui en dépendent en soient informés et mis à jour automatiquement.

Description du problème

On trouve des classes possédant des attributs dont les valeurs changent régulièrement. De plus, un certain nombre de classes doit être tenu informé de l'évolution de ces valeurs. Il n'est pas rare d'être confronté à ce problème notamment en développant une classe métier et les classes d'affichages correspondantes.

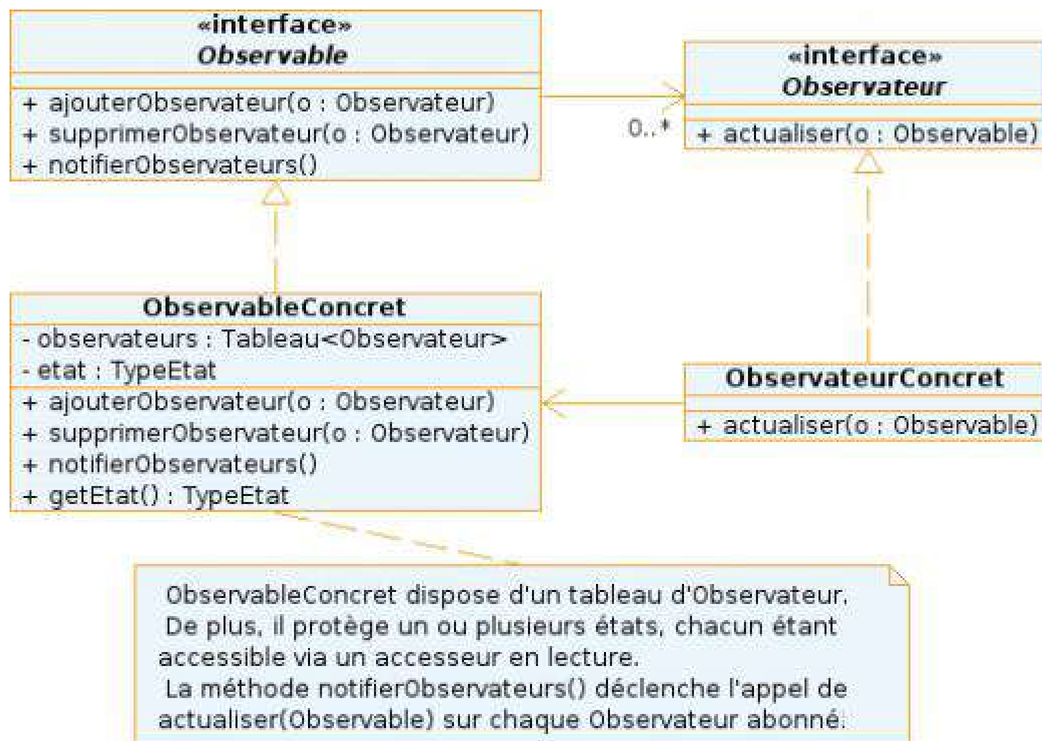
Afin d'illustrer ce problème récurent, prenons un exemple volontairement simpliste. On considère une classe HeurePerso possédant dans le même attribut l'heure et la minute courante. Cette classe, et plus particulièrement son attribut, est utilisé pour l'affichage de l'heure courante dans une fenêtre, lors de l'écriture de logs... Pour cela, on définit une classe AfficheHeure qui se charge d'afficher l'heure et la minute courante dans une partie de la fenêtre. On peut alors se demander quelle démarche adopter pour que la classe chargée de l'affichage soit tenue informée en temps réel de l'heure courante stockée dans la classe HeurePerso ?

On peut identifier deux solutions. Soit la classe d'affichage se charge de demander à la classe HeurePerso la valeur de son attribut soit c'est la classe HeurePerso qui informe la classe AfficheHeure lors de changements.

Il est facile de s'apercevoir que la première solution n'est pas la meilleure. En effet, quand la classe AfficheHeure devra t-elle questionner HeurePerso pour obtenir l'heure courante ? Toutes les minutes ? Toutes les secondes ? Quelque soit l'intervalle choisi, soit l'heure ne sera pas précise soit on surchargera d'appels inutiles la classe HeurePerso.

La solution consiste donc à laisser la charge à la classe HeurePerso d'informer sa classe d'affichage de ses changements de valeurs. Cependant la classe HeurePerso doit pouvoir informer plusieurs classes d'affichage et cela en évitant de lier fortement les classes entre elles. C'est à dire qu'une modification des classes d'affichage ne doit pas engendrer de modification dans la classe métier et vice versa. Comment peut-on faire ? Il est temps d'étudier le diagramme UML du pattern Observateur qui répond de manière éprouvée à cette problématique.

Diagramme UML



Définition de la solution

Le diagramme UML du pattern Observateur définit deux interfaces et deux classes. L'interface Observateur sera implémenté par toutes classes qui souhaitent avoir le rôle d'observateur. C'est le cas de la classe ObservateurConcret qui implémente la méthode actualiser(Observable). Cette méthode sera appelée automatiquement lors d'un changement d'état de la classe observée.

On trouve également une interface Observable qui devra être implémentée par les classes désireuses de posséder des observateurs. La classe ObservableConcret implémente cette interface, ce qui lui permet de tenir informer ses observateurs. Celle-ci possède en attribut un état (ou plusieurs) et un tableau d'observateurs. L'état est un attribut dont les observateurs désirent suivre l'évolution de ses valeurs. Le tableau d'observateurs correspond à la liste des observateurs qui sont à l'écoute. En effet, il ne suffit pas à une classe d'implémenter l'interface Observateur pour être à l'écoute, il faut qu'elle s'abonne à un Observable via la méthode ajouterObservateur(Observateur).

En effet, la classe ObservableConcret dispose de quatre méthodes que sont ajouterObservateur(Observateur), supprimerObservateur(Observateur), notifierObservateurs() et getEtat(). Les deux premières permettent, respectivement, d'ajouter des observateurs à l'écoute de la classe et d'en supprimer. En effet, le pattern Observateur permet de lier dynamiquement (faire une liaison lors de l'exécution du programme par opposition à lier statiquement à la compilation) des observables à des observateurs. La méthode notifierObservateurs() est appelée lorsque l'état subit un changement de valeur. Celle-ci avertit tous les observateurs de cette mise à jour. La méthode getEtat() est un simple accesseur en lecture pour l'état. En effet, les observateurs récupèrent via la

méthode actualiser(Observable) un pointeur vers l'objet observé. Puis, grâce à ce pointeur, et à la méthode getEtat() il est possible d'obtenir la valeur de l'état.

Appliquons l'exemple précédent à ce diagramme UML. HeurePerso correspond à la classe ObservableConret dont l'heure et la minute courantes seraient son état. La classe AfficheHeure correspond elle à ObservateurConcret.

Approfondissement de la solution

Une des questions récurrente face à ce pattern est pourquoi ces deux interfaces ? D'ailleurs on trouve sur Internet des implémentations de ce pattern sans ces deux interfaces... Mais l'utilisation de ces interfaces permet de coupler faiblement l'observable à ses observateurs. En effet, un principe de conception est de lier des interfaces plutôt que des classes afin de pouvoir faire évoluer le modèle facilement. L'utilisation de ces deux interfaces n'est donc pas obligatoire mais elle est vivement conseillée.

Cependant, il existe une variation possible lors de l'utilisation de ce pattern. Dans la solution présentée ci dessous, une référence vers l'objet observable est mis à disposition de chaque observateur. Ainsi les observateurs peuvent l'utiliser pour appeler la méthode getEtat() et ainsi obtenir l'état de l'observable. Cette solution est nommée « TIRER » car c'est aux observateurs, une fois avertis de l'évolution, d'aller chercher l'information sur l'état. Mais il existe la solution inverse appelée « POUSSER ». Dans ce cas, on passe directement l'état actuel de l'observable dans la méthode actualiser(TypeEtat). Ainsi les observateurs disposent directement de l'état. Mais pourquoi avoir présenté la solution nommée « TRIER » plutôt que l'autre ? Parce qu'elle permet une fois de plus de lier faiblement l'observable à ses observateurs. En effet, si l'observateur dispose d'un pointeur vers l'objet observable et que la classe observable évolue en ajoutant un deuxième état. L'observateur souhaitant se tenir informé de ce deuxième état aura juste à appeler l'accesseur correspondant. Alors que si on « POUSSER » il faudrait changer la signature de la méthode ce qui peut s'avérer plus dommageable.

Conséquences

Un exemple que l'on rencontre souvent pour illustrer ce pattern est une représentation entre une entreprise qui diffuse un magazine et des personnes qui souhaitent s'y abonner (observateurs) et donc le recevoir régulièrement.

Le pattern observateur permet de lier de façon dynamique un observable à des observateurs. Cette solution est faiblement couplée ce qui lui permet d'évoluer facilement avec le modèle. D'ailleurs le pattern Observateur est très utilisé. Il fait partie, par exemple, des patterns indispensables pour mettre en place le modèle MVC (Modèle Vue Contrôleur) très en vogue actuellement.