Design Pattern

Génie Logiciel

Yann LAURET Marie-Anne ROFFAT-COTTUS

Table des matières

INTRO définition basique

I. Historique

L'origine des Design Patterns remonte au début des années 70 avec les travaux de l'architecte Christopher Alexander. Celui-ci remarque que la phase de conception en architecture laisse apparaître des problèmes récurrents. Il cherche alors à résoudre l'ensemble de ces problèmes liés à des contraintes interdépendantes (solidité de la structure, étanchéité...). Pour cela Alexander établi un langage de 253 patterns, qui couvrent tous les aspects de la construction (comme par exemple la façon de concevoir une charpente).

Dans les années 90, l'idée de Christopher Alexander va être reprise et étendue au domaine de la conception des logiciels. Le concept de Design Pattern est développé dans un ouvrage publié en 1995 par le « **Gang of Four** ». Celui-ci présente 23 Design Patterns qui font aujourd'hui référence dans le monde de l'informatique.

II. Définition plus approfondie

Un design pattern est une solution éprouvée à un problème récurrent dans la conception d'applications orientées objet. Le problème peut être, entre autres, l'optimisation, l'évolutivité ou encore la clarté.

Chaque design pattern a été élaboré dans un but précis et afin de rendre le code plus simple et plus efficace. C'est une méthode souvent ellipsée par les développeurs débutants alors qu'elle constitue un des piliers du développement orienté objet, au même titre que les algorithmes. En effet, comme l'algorithme, le design paterne se définit en amont du développement et est indépendant du langage de programmation utilisé.

De façon plus imagé, si notre application est assimilée à un tableau, le design pattern correspond à la toile et au cadre dans lequel on souhaite mettre notre tableau, l'algorithme correspond à nos croquis et enfin le code source correspond à la peinture.

Chaque design pattern, ou patron de conception, est caractérisé par 4 éléments :

- Un nom : pour l'identifier clairement
- Une problématique : pour décrire le problème rencontré
- Une solution : pour décrire la solution apportée, souvent illustré par un schéma UML
- Une conséquence : pour décrire les avantages et inconvénient de la solution

III. Utilisation

Il existe 3 catégories regroupant 23 design patterns que nous allons détailler :

- Création : ceux qui permettent d'instancier et de configurer des classes et des objets

- Structure : ceux qui permettent d'organiser les classes d'une application
- Comportement : ceux qui permettent d'organiser les objets pour qu'ils collaborent entre eux.

III.1 Design patterns de création

Les design patterns de création sont des modèles de conception qui traitent des mécanismes de création d'objet, en essayant de créer des objets de manière appropriée à la situation. La forme de base de la création d'objet pourrait entraîner des problèmes de conception ou une complexité accrue de la conception. Les design patterns de création résolvent ce problème en contrôlant de quelque manière la création d'objets doit se dérouler.

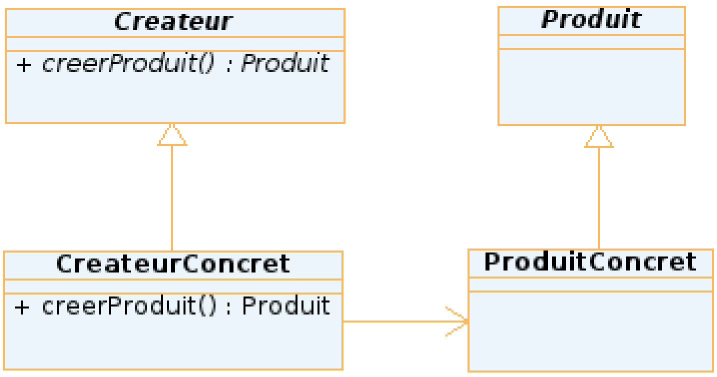
Les modèles les plus utilisés parmi les design patterns de création sont Factory method et Singleton.

1.A Abstract Factory

Le design pattern Factory method est utilisé pour instancier différents types d'objet d'une même classe en fonction d'un ou plusieurs paramètres. Il est possible de rajouter différents tests au sein de notre classe afin déterminer comment notre objet doit être instancié. Mais, si les différents types d'objet sont complexes ou ont tendance à évoluer, le code de notre classe sera lourd et fortement couplé aux types concrets.

Pour éviter cela on passe par un créateur qui va gérer quel type d'objet doit être créé, puis chaque différent type sera géré au sein d'une classe propre.

Cette implémentation est représentée par le schéma ci-dessous :



Explication du schéma :

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.

Si une partie de l'implémentation est identique à tous les produits concrets, alors l'interface Produit peut être une classe abstraite afin d'intégrer ce code partagé dans celle-ci. Il est également bénéfique d'utiliser ce pattern même si l'on a qu'un seul CreateurConcret ou qu'un

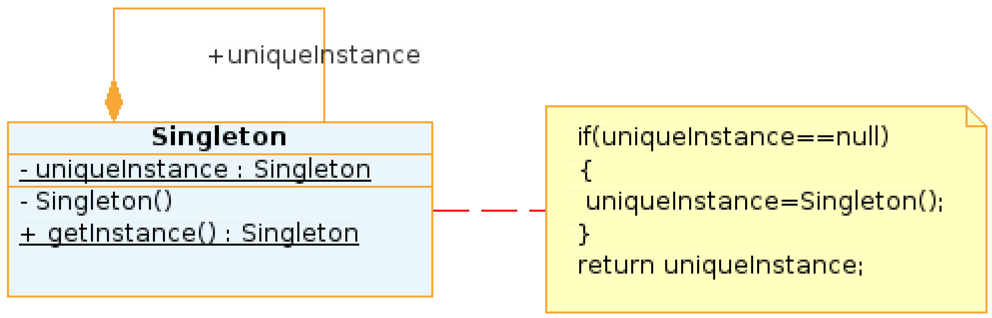
CreateurConcret n'instancie qu'un seul Produit car les avantages liés au découplage des produits et du créateurs sont conservés.

1.B Singleton

Le design pattern Singleton permet de n'instancier qu'un et un seul objet d'une classe donnée. C'est par exemple le cas d'une classe qui implémente un pilote ou encore un système de journalisation.

Une méthode couramment utilisée est d'instancier notre objet au lancement de l'application et de le stocker dans une variable globale. Toutefois, cette pratique force l'instanciation d'objet qui ne seront pas forcément utilisé et ne respecte pas le principe de l'encapsulation de la POO.

Afin d'éviter cela, on utilise le Singleton. Pour cela il faut bloquer le constructeur de la classe concernée en le passant en private et créer un attribut qui va contenir noitre unique instance si elle est créée. Il faut ensuite créer un pseudo constructeur qui va tester notre attribut. Si l'attribut est null alors on instancie notre objet, sinon on retourne notre instance déjà créée. Ce principe est modélisé par le schéma suivant :



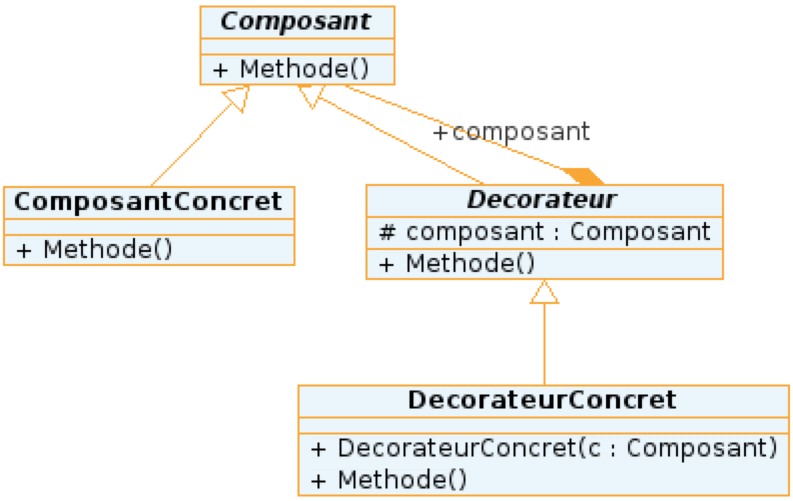
III.2 Design patterns de structure

En génie logiciel, les modèles de conception structurale sont des modèles de conception qui facilitent la conception en identifiant un moyen simple de réaliser des relations entre entités.

Le modèle le plus utilisé parmi les design patterns de structure est le Decorator.

2.A Decorator

Le design pattern Decorator permet d'ajouter des fonctionnalités à une classe, comme le fait l'héritage. Toutefois, il peut arriver que l'une des classes héritées ne doivent pas avoir accès à certaines méthodes de la classe mère, ou encore, dans un système complexe, l'ajout d'une classe héritée, et donc la redéfinition de certaines méthodes, peuvent engendrer de nouveaux bugs. Pour éviter cela on utilise le design pattern Decorator qui se traduit par le schéma suivant :



Explication du schéma :

Ici nous avons 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.

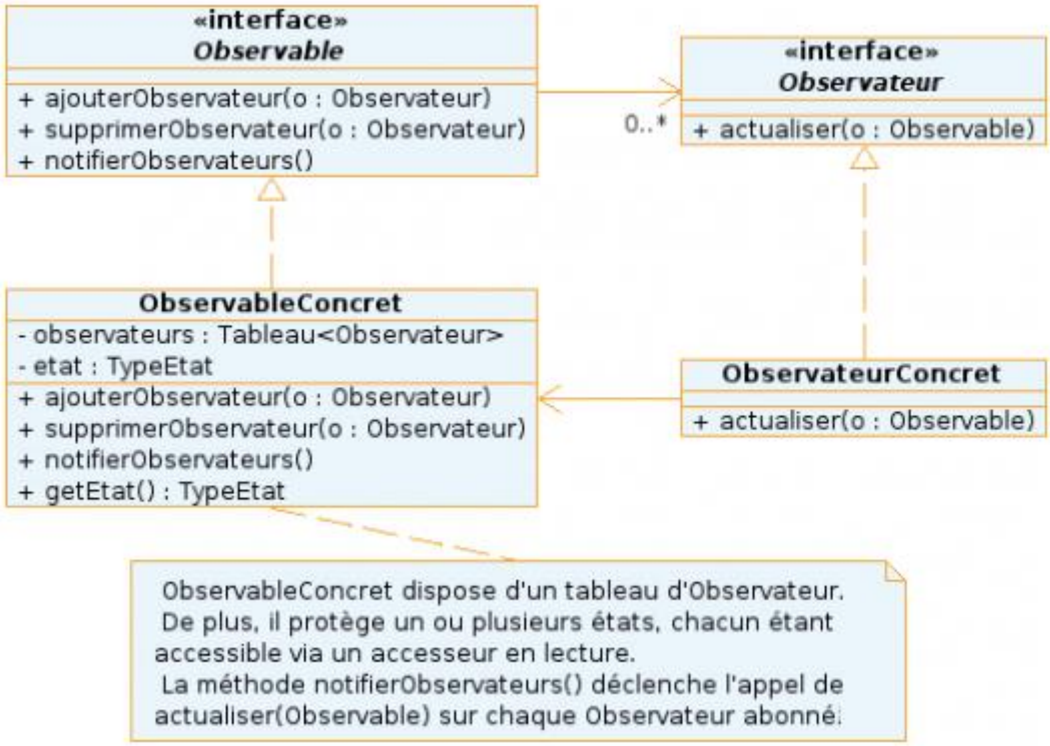
III.3 Design patterns de comportement

Dans l'ingénierie logicielle, les modèles de conception comportementale sont des modèles de conception qui identifient des modèles communs de communication entre objets et réalisent ces modèles. Ce faisant, ces modèles augmentent la flexibilité dans la réalisation de cette communication.

Le modèle le plus utilisé parmi les design patterns de structure est l'Observer.

3.A Observer

Le design pattern Observer permet de suivre l'évolution d'attributs d'une classe changeant régulièrement pour, par exemple, actualiser un affichage. Afin de mettre à jour une vue régulièrement lors d'un changement de valeur, deux solutions s'offre à nous, soit la classe chargée de l'affichage consulte régulièrement les valeurs pour savoir si une mise à jour est nécessaire, soit notre classe contenant notre attribut à surveiller contact la classe chargée de l'affichage lors d'un changement. Cette seconde solution est le design pattern Observer schématiser ci-dessous :



Explication du schéma :

Le diagramme UML du pattern Observateur définit deux interfaces et deux classes. L'interface Observateur sera implémentée 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.

IV. Intérêt (antipattern)

IV.1 Les avantages

L'utilisation des Design Patterns offre de nombreux avantages. Tout d'abord cela permet de répondre à un problème de conception grâce à une solution éprouvée et validée par des experts. Ainsi on gagne en rapidité et en qualité de conception ce qui diminue également les coûts.

De plus, les Design Patterns sont réutilisables et permettent de mettre en avant les bonnes pratiques de conception.

Les Design Patterns étant largement documentés et connus d'un grand nombre de développeurs ils permettent également de faciliter la communication. Si un développeur annonce que sur ce point du projet il va utiliser le Design Pattern Observateur il est compris des informaticiens sans pour autant rentrer dans les détails de la conception (diagramme UML, objectif visé...).

IV.2 Les inconvénients

L'utilisation des Design Patterns est plutôt difficile, et il faut être expérimenter pour avoir une bonne utilisation du Design Pattern dans sa globalité.

Il est difficile de savoir lequel des 23 patterns est le plus adéquate au system utilisé.

Il n'est utilisé que sur de la programmation orienté objet.

V. Conclusion (outils évolution...)

Le Design Pattern est un outil avec plusieurs fonctionnalités mais que pour le développement orienté objet. Ce qui ne lui permet pas de fonctionner pour tous les projets. Assez complexe pour des personnes qui ne se sont pas renseignées sur le Design Pattern.

Il a plus de 23 pattern, ce qui lui laisse une large opportunité sur les projets.