CESI exia

ÉCOLE D'INGÉNIEURS INFORMATIQUE

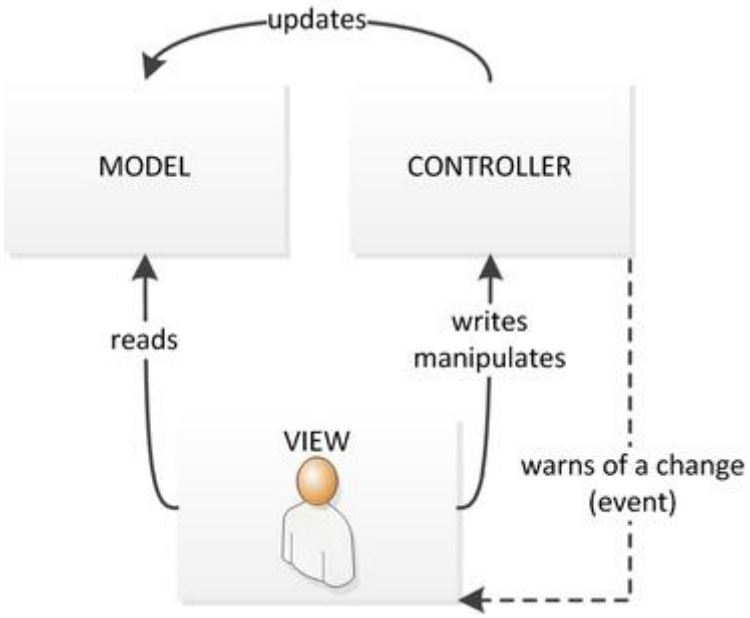
A3 UE 2 PROJET

GLORIA Ghislain – COMBALDIEU Paul
OILLIC Arnaud – SEBAGH Aaron

Table des matières

MVC

Le patron d'architecture (architectural pattern) Modèle-Vue-Contrôleur (MVC) sépare une application en trois composants principaux : le modèle, la vue et le contrôleur.



Modèle : Cette partie gère les données de votre site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc entre autres les requêtes SQL.

Dans notre restaurant le modèle s'occupera donc des rôles, des interactions et des déplacements des différents acteurs ainsi que de gérer l'accès aux stocks des objets dans la base de données

Vues : Cette partie se concentre sur l'*affichage*. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher

Dans notre restaurant c'est lui qui affichera la fenêtre ou on verra la simulation, les données afficher dans cette fenêtre viendront du modèle.

Contrôleurs : Cette partie gère la logique du code qui prend des *décisions*. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue.

Dans une application MVC, la vue ne fait qu'afficher les informations que le contrôleur gère et répond aux entrées et actions de l'utilisateur. Par exemple, le contrôleur traite les valeurs de chaîne de requête et transmet ces valeurs au modèle qui, en retour, effectue les requêtes sur la base de données en utilisant les valeurs.

Un avantage apporté par ce modèle est la clarté de l'architecture qu'il impose. Cela simplifie la tâche du développeur qui tenterait d'effectuer une maintenance ou une amélioration sur le projet. En effet, la modification des traitements ne change en rien la vue. Par exemple on peut passer d'une base de données de type SQL à XML en changeant simplement les traitements d'interaction avec la base, et les vues ne s'en trouvent pas affectées.

SINGLETON

Restreindre une classe à une seule instance d'elle-même

Parfois, une classe, par son fonctionnement et son utilisation, bénéficie du fait de n'être instanciée qu'une seule fois.
Un exemple parlant est celui d'un gestionnaire de système de fichier : le rôle de cette classe va être de coordonner des lectures et écritures de fichiers sur le disque. Ces opérations se feront de manière asynchrone puisqu'elles prennent un certain temps à être exécutées par le système d'exploitation. Il est donc nécessaire pour notre gestionnaire de connaitre l'état et le résultat de chaque opération, et nous avons donc tout intérêt à ce qu'une seule instance de cette classe ne puisse exister à un moment donné, afin d'éviter de sérieux conflits d'accès au système de fichiers.

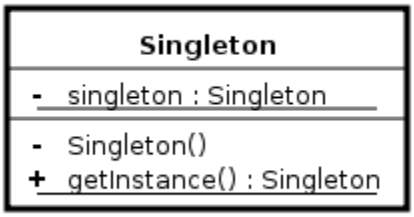
Atteindre cette classe depuis un point d'accès global dans le code

Notre gestionnaire de fichiers doit être accessible à un certain nombre d'autres classes, telles que le système de trace (le « logger »), le gestionnaire d'icônes et d'images, ou encore le gestionnaire de sérialisation. Si toutes ces classes ne peuvent pas créer d'instances de ce gestionnaire, étant donné le risque de se retrouver avec plusieurs instances différentes, comment peuvent-elles en obtenir une ?

Une réponse possible est bien entendu un accès global, depuis n'importe quel point du code, via une instance statique.

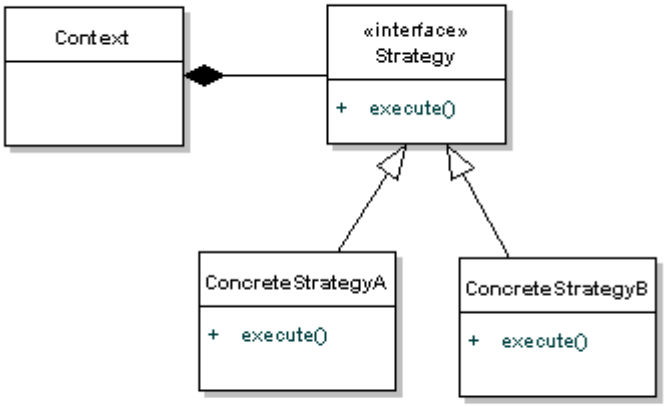
La complémentarité

Finalement on voit très vite que le Singleton, qui répond à la fois au besoin d'unicité et au besoin d'accessibilité, semble une solution logique dans ce genre de situations.



STRATEGY (dans MVC)

Le patron de conception **Strategy** est utile pour des situations où il est nécessaire de permuter dynamiquement les algorithmes utilisés dans une application. Le patron stratégie est prévu pour fournir le moyen de définir une famille d'algorithmes, encapsuler chacun d'eux en tant qu'objet, et les rendre interchangeables. Ce patron laisse les algorithmes changer indépendamment des clients qui les emploient.



FACADE

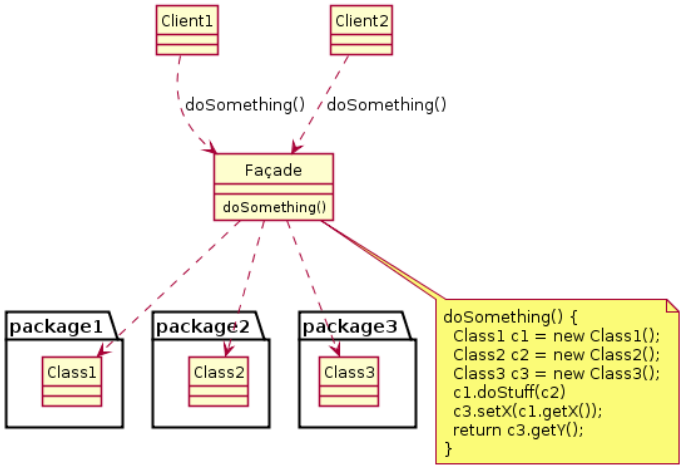
Façade a pour but de cacher une conception et une interface complexe difficile à comprendre (cette complexité étant apparue « naturellement » avec l'évolution du sous-système en question).

La façade permet de simplifier cette complexité en fournissant une interface simple du sous-système. Habituellement, la façade est réalisée en réduisant les fonctionnalités de ce dernier, mais en fournissant toutes les fonctions nécessaires à la plupart des utilisateurs.

La façade encapsule la complexité des interactions entre les objets métier participant à un workflow.

Une façade peut être utilisée pour :

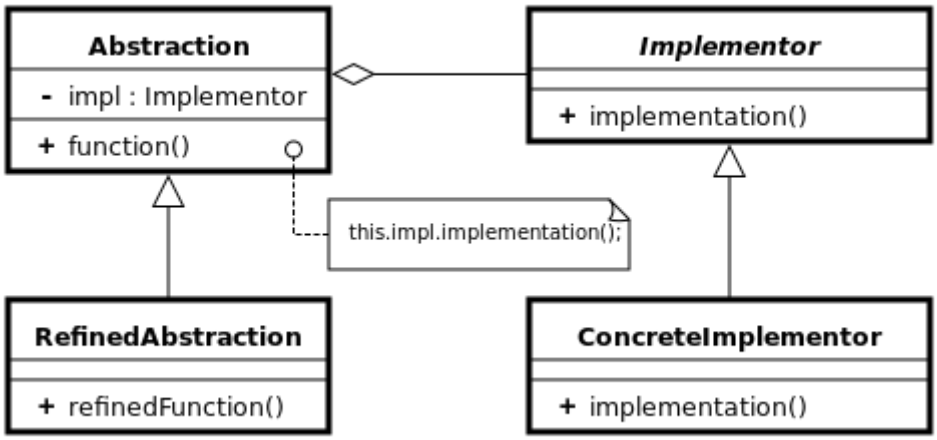
- Rendre une bibliothèque plus facile à utiliser, comprendre et tester ;
- Rendre une bibliothèque plus lisible ;
- Réduire les dépendances entre les clients de la bibliothèque et le fonctionnement interne de celle-ci, ainsi on gagne en flexibilité pour les évolutions futures du système ;
- Assainir une API que l'on ne peut pas modifier si celle-ci est mal conçue, ou mieux découper ses fonctionnalités si celle-ci n'est pas assez claire.
- Un adaptateur est utilisé lorsque l'on doit respecter une interface bien définie. La façade est utilisée pour simplifier l'utilisation de l'API.



BRIDGE

Bridge permet de découpler l'interface d'une classe et son implémentation.

La partie concrète (implémentation réelle) peut alors varier, indépendamment de celle abstraite (définition virtuelle), tant qu'elle respecte le contrat de réécriture associé qui les lie (obligation de se conformer aux signatures des fonctions/méthodes, et de leurs fournir un corps physique d'implémentation).



FACTORY

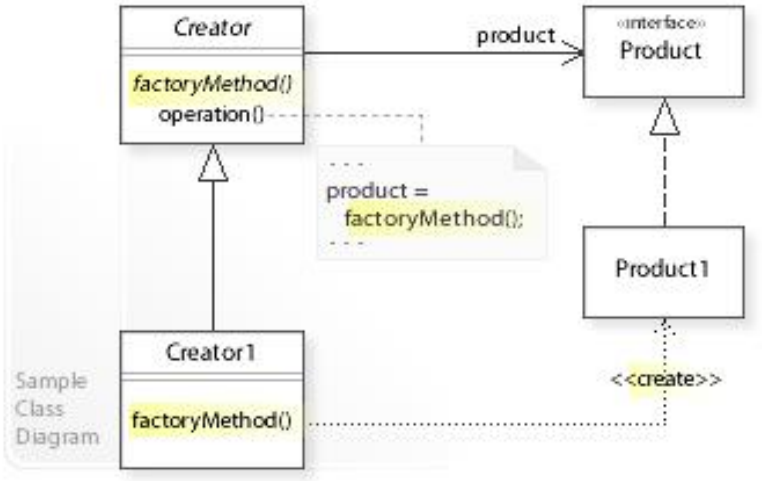
Factory permet d'instancier des objets dont le type est dérivé d'un type abstrait. La classe exacte de l'objet n'est donc pas connue par l'appelant.

Plusieurs fabriques peuvent être regroupées en une fabrique abstraite permettant d'instancier des objets dérivant de plusieurs types abstraits différents.

Les fabriques étant en général uniques dans un programme, on utilise souvent le patron de conception singleton pour les implémenter.

- Les **Factory** sont utilisées dans les toolkits ou les frameworks, car leurs classes sont souvent dérivées par les applications qui les utilisent.

- Des hiérarchies de classes parallèles peuvent avoir besoin d'instancier des classes de l'autre.

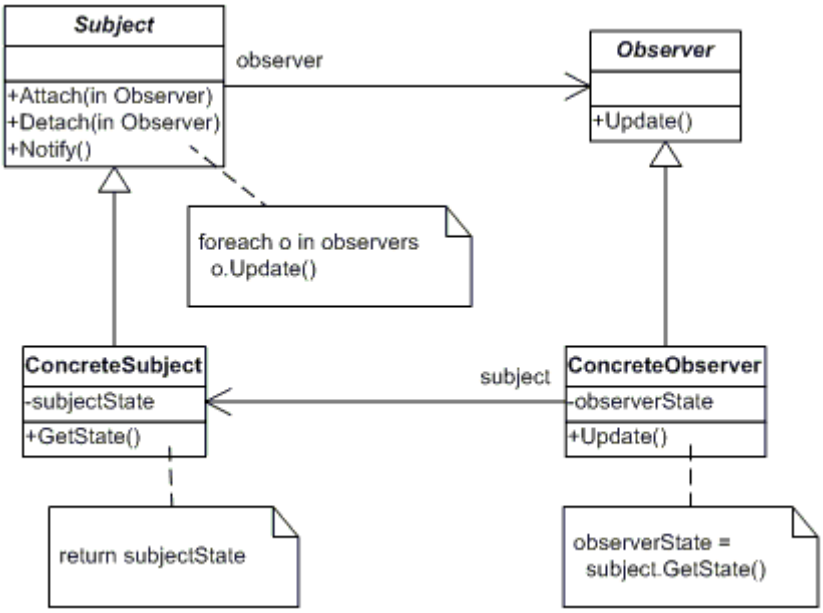


Observer (Dans MVC)

Observer est utilisé pour envoyer un signal à des modules qui jouent le rôle d'observateurs. En cas de notification, les observateurs effectuent alors l'action adéquate en fonction des informations qui parviennent depuis les modules qu'ils observent (les observables).

Les notions d'observateur et d'observable permet de limiter le couplage entre les modules aux seuls phénomènes à observer. Il permet aussi une gestion simplifiée d'observateur multiples sur un même objet observable.

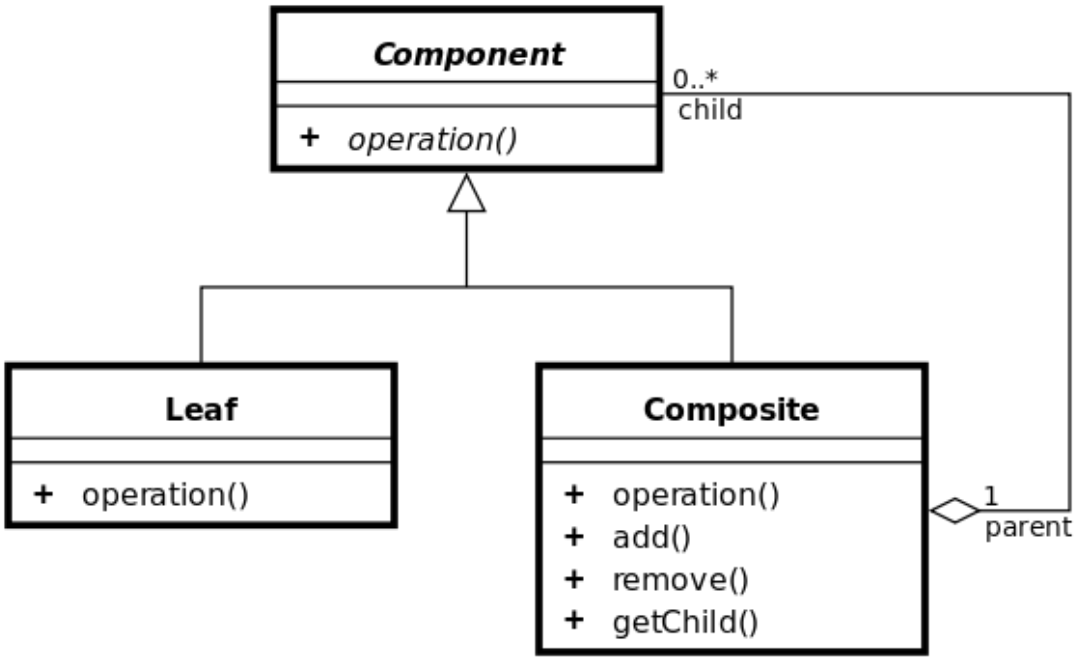
Il est recommandé dès qu'il est nécessaire de gérer des évènements, quand une classe déclenche l'exécution d'une ou plusieurs autres.



Composite(dans MVC)

Un objet **composite** est constitué d'un ou de plusieurs objets similaires (ayant des fonctionnalités similaires). L'idée est de manipuler un groupe d'objets de la même façon que s'il s'agissait d'un seul objet. Les objets ainsi regroupés doivent posséder des opérations communes, c'est-à-dire un "dénominateur commun".

Ce patron permet de concevoir une structure d'arbre, par exemple un arbre binaire en limitant à deux le nombre de sous-éléments.



DAO

Les objets en mémoire vive sont souvent liés à des données persistantes (stockées en base de données, dans des fichiers, dans des annuaires…). Le modèle DAO propose de regrouper les accès aux données persistantes dans des classes à part, plutôt que de les disperser. Il s'agit surtout de ne pas écrire ces accès dans les classes "métier", qui ne seront modifiées que si les règles de gestion métier changent.

DAO permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier. Ainsi, le changement du mode de stockage ne remet pas en cause le reste de l'application. En effet, seules ces classes dites "techniques" seront à modifier (et donc à re-tester). Cette souplesse implique cependant un coût additionnel, dû à une plus grande complexité de mise en œuvre.