Design pattern UML

Table des matières

[1 Introduction 2](#_Toc478996145)

[2 L’adaptateur 2](#_Toc478996146)

[3 Le composite 3](#_Toc478996147)

[4 Le décorateur 3](#_Toc478996148)

[5 L’état 4](#_Toc478996149)

[6 Le créateur 5](#_Toc478996150)

[7 Le singleton 6](#_Toc478996151)

[8 Résumé des patterns 6](#_Toc478996152)

[9 Références 7](#_Toc478996153)

# Introduction

Les design-patterns sont des modèles d’architecture éprouvés pour des problématiques communément rencontrées en POO.

Il est bon de les connaître, plutôt que de réinventer soi-même des solutions déjà existantes, ou de mettre en œuvre des solutions moins performantes.

Nous présentons ci-dessous quelques patterns, parmi ceux qui sont le plus utilisés. Une liste complète est donnée

# L’adaptateur

Ce pattern fournit une couche d’adaptation entre l’API d’un composant et notre code.

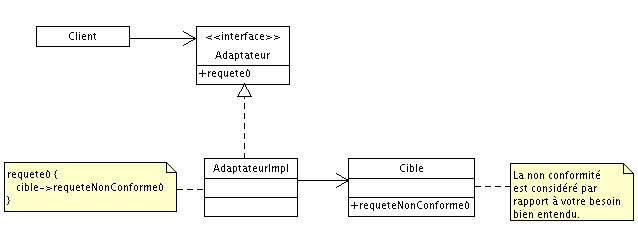
NB/ Une API (Application Programming Interface) représente l’interface publique d’un composant logiciel externe qu’on souhaite utiliser. C’est l’ensemble des méthodes, propriétés, évènements… auxquels on a accès pour utiliser le composant.

En effet, il se peut que l’API du composant qu’on souhaite utiliser ne corresponde pas à nos besoins ou contraintes pour diverses raisons :

* On souhaite simplifier la mise en œuvre de certaines fonctionnalités (avec des valeurs de paramètres prédéfinis par exemple), ou interdire l’accès à certaines autres fonctionnalités.
* On souhaite avoir une API commune pour différents composants fournissant les mêmes services
* …

NB/ Lorsqu’on simplifie l’interface d’un composant en limitant les points d’entrée de son API, on dit qu’on réalise une **façade**.

Diagramme UML :

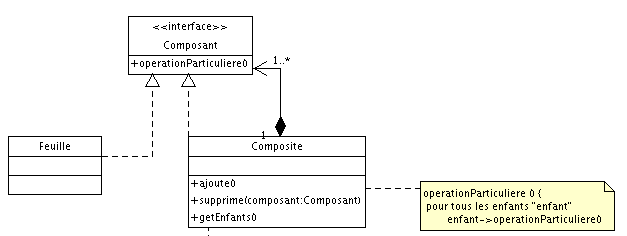


Il suffit en fait de fournir une classe intermédiaire qui soit conforme à nos attentes. Celle-ci délègue et adapte toutes les requêtes qui lui sont faites à l'API fournie. Il est préférable de s'appuyer sur la notion d'interface pour ne pas dépendre de l'implémentation.

# Le composite

Il modélise une structure arborescente de composants cohérents. Il permet à la classe cliente de composer, et utiliser d'une manière cohérente (unique) une hiérarchie complexe d'objets.

**Diagramme UML :**



Le composite est un composant qui contient (agrège) d’autres composants

La feuille est un composant final, c’est-à-dire qui n’est pas lui-même un composite.

Les 2 types dérivent d’une interface commune représentant un composant.

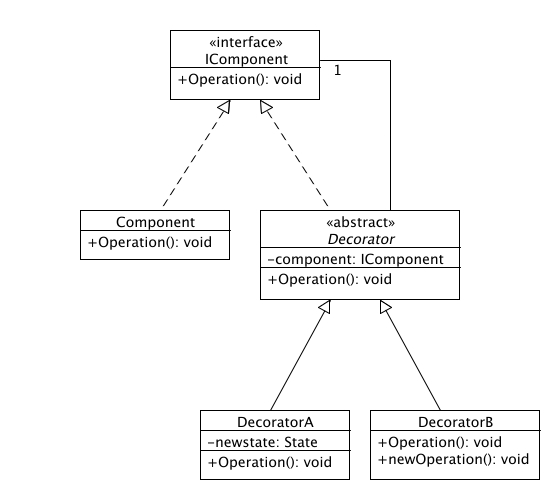
# Le décorateur

Ajoute des fonctionnalités à un objet de manière statique ou dynamique, indépendamment des autres instances de la même classe. Il permet ainsi d'étendre les capacités d’une classe sans en modifier les sources.

L’héritage est la technique qui vient naturellement à l’esprit lorsqu’on souhaite étendre les fonctionnalités d’une classe. Cependant, cette technique n’est pas appropriée lorsque les fonctionnalités à ajouter n’ont pas de rapport direct avec l’objectif initial de la classe.

Exemple : on dispose d’une classe CompteBancaire décrivant les propriétés et opération courantes d’un compte (créditer et débiter le compte, calculer ses intérêts…etc). On souhaite lui ajouter des fonctionnalités permettant de prévenir le client de certains évènements (virements, découverts…). Celles-ci n’ont pas de rapport direct avec l’objectif initial de la classe CompteBancaire.

**Diagramme UML**



IComponent est une interface de laquelle hérite le composant qu’on souhaite enrichir

Component est la classe sur laquelle on souhaite apporter de nouvelles fonctionnalités (décorer)

Decorator est une classe abstraite dans laquelle :

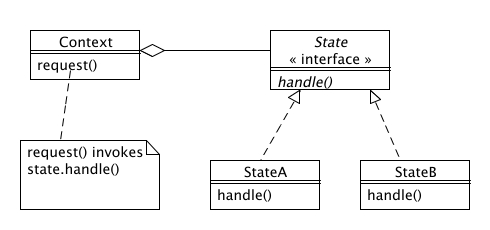
* On agrège une instance de la classe à décorer
* On implémente l’interface en appelant simplement les méthodes de l’instance à décorer

DecoratorA et B sont des classes dérivées de Decorator, qui ajoutent toutes les fonctionnalités supplémentaires souhaitées.

# L’état

Il permet de modifier le comportement d'un objet lorsque son état est modifié.

**Diagramme UML**



State est la classe abstraite ou l’interface qui représente un état. Il n’y a toujours qu’un état actif à la fois et chaque état implémente un comportement différent du système.

Chaque état individuel est représenté par une classe qui implémente l’interface State.

Context agrège des états et orchestre l'exécution des comportements. Elle constitue l’interfaçage avec le reste de l’application. Elle

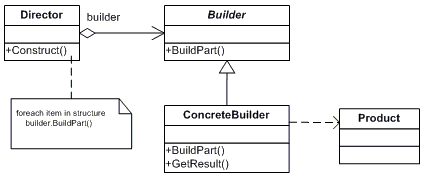
Le changement d'état peut être provoqué soit par la fin de l'exécution de l'état actif (à la fin de la méthode handle d’une des classes StateA, B…), soit par la classe Context.

# Le créateur

NB/ Traduit en Builder en anglais

Ce pattern sépare la construction d’un objet complexe de sa représentation, de sorte que le même process de construction peut créer plusieurs représentations.

**Diagramme UML**



La classe Builder est une classe abstraite fournissant les méthodes pour la création d’objets de type Product

Product est l’objet complexe à construire. Elle peut agréger des classes qui décrivent les constituants de l’objet.

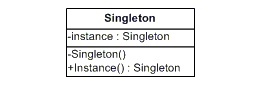
ConcreteBuilder implémente concrètement Builder. Elle construit l’objet Product à partir de ses constituants, stocke sa référence, et fournit un accès à cet objets via GetResult.

Director lance la construction d’un objet complexe en utilisant une instance de Builder.

# Le singleton

Permet de s’assurer qu’une classe est instanciée une seule fois et fournit un accès global à cette instance.

Diagramme UML :



La classe singleton est responsable de la création de son instance unique.

Elle fournit une méthode statique Instance, qui permet d’accéder à cette instance unique.

# Résumé des patterns

Patterns de création

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| Usine abstraite | Crée une instance de plusieurs familles de classes |
| Créateur | Séparer la construction d’un objet de sa représentation |
| Méthode usine | Crée une instance de plusieurs classes dérivées |
| Prototype | Instance initialisée et prête à être copiée ou clonée |
| Singleton | Classe sont on ne peut créer qu’une seule instance |

Patterns de structure

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| Adaptateur | Permet d’adapter les interfaces de composants |
| Pont | Séparer l’interface d’un objet de son implémentation |
| Composite | Structure arborescente d’objets simples et composites |
| Décorateur | Enrichit fonctionnellement un objet de façon dynamique |
| Façade | Classe unique représentant un sous-sytème |
| Poids-mouche | Instance de granularité fine utilisée pour des échanges d’infos optimisés |
| Proxy | Objet qui en représente un autre |

Patterns de comportement

|  |  |
| --- | --- |
| **Pattern** | **Description** |
| Chaîne de responsabilités | Permet de passer une requête à travers une chaîne d’objets |
| Commande | Encapsule une commande dans un objet |
| Interprète | Une façon d'inclure des éléments de langue dans un programme |
| Itérateur | Accès séquentiel aux éléments d’une collection |
| Intermédiaire | Définit une communication simplifiée entre classes |
| Memento | Capture et restaure l’état interne d’un objet |
| Observateur | Permet de notifier des changements à d’autres classes |
| Etat | Modifie le comportement d’un objet lorsque son état change |
| Stratégie | Encapsule un algorithme dans une classe |
| Méthode modèle | Reporte les étapes d’un algorithme à une sous-classe |
| Visiteur | Définit une nouvelle opération à effectuer sur une classe sans avoir à modifier celle-ci |

# Références

Site [dofactory .com](http://www.dofactory.com/net/design-patterns) consacrés aux design patterns et à leur implémentation en C#

Articles sur Wikipedia : [Decorator Pattern](https://en.wikipedia.org/wiki/Decorator_pattern), [State pattern](https://en.wikipedia.org/wiki/State_pattern),

Suite d’articles de Sébastien Méric, consacrés aux différents patterns sur le site [Developpez.com](http://smeric.developpez.com/java/uml/). Ces articles fournissent aussi des exemples d’implémentation en Java.