DEFOIS, FARISSE, HAMANI, LOPPY, VERGINE

Documentation D’un Design Pattern

Table des matières

[I - ADAPTATEUR 2](#_Toc499032080)

[A - Nom du Pattern 2](#_Toc499032081)

[B - Description du Pattern 2](#_Toc499032082)

[C - Exemple de diagramme UML (Problème et Solution) 2](#_Toc499032083)

[D - Les Domaines d’Application du Pattern 3](#_Toc499032084)

[E - Exemple de Code en C# 3](#_Toc499032085)

[II - FAÇADE 5](#_Toc499032086)

[A - Nom du Pattern 5](#_Toc499032087)

[B - Description du Pattern 5](#_Toc499032088)

[C - Exemple de Diagramme UML 6](#_Toc499032089)

[D - Les Domaines d’Application du Pattern 6](#_Toc499032090)

[E - Exemple de Code en C# 7](#_Toc499032091)

[III - SINGLETON 8](#_Toc499032092)

[A - Nom du Pattern 8](#_Toc499032093)

[B - Description du Pattern 8](#_Toc499032094)

[C - Exemple de Diagramme UML 9](#_Toc499032095)

[D - Les Domaines d’Application du Pattern 9](#_Toc499032096)

[E - Exemple de Code en C# 10](#_Toc499032097)

[IV - Fabrique 11](#_Toc499032098)

[A - Nom du Pattern 11](#_Toc499032099)

[B - Description du Pattern 11](#_Toc499032100)

[C - Exemple de Diagramme UML 11](#_Toc499032101)

[D - Les Domaines d’Application du Pattern 11](#_Toc499032102)

[E - Exemple de Code en C# 11](#_Toc499032103)

[IV - Template Mmethod Pattern 15](#_Toc499032104)

[A - Nom du Pattern 15](#_Toc499032105)

[B - Description du Pattern 15](#_Toc499032106)

[Le pattern Template Method permet de reporter dans des sous-classes certaines étapes des opérations d’un objet, ces étapes étant alors écrites dans les sous-classes. 15](#_Toc499032107)

[C - Exemple de Diagramme UML 15](#_Toc499032108)

[D - Les Domaines d’Application du Pattern 15](#_Toc499032109)

[E - Exemple de Code en C# 15](#_Toc499032110)

# I - ADAPTATEUR

## A - Nom du Pattern

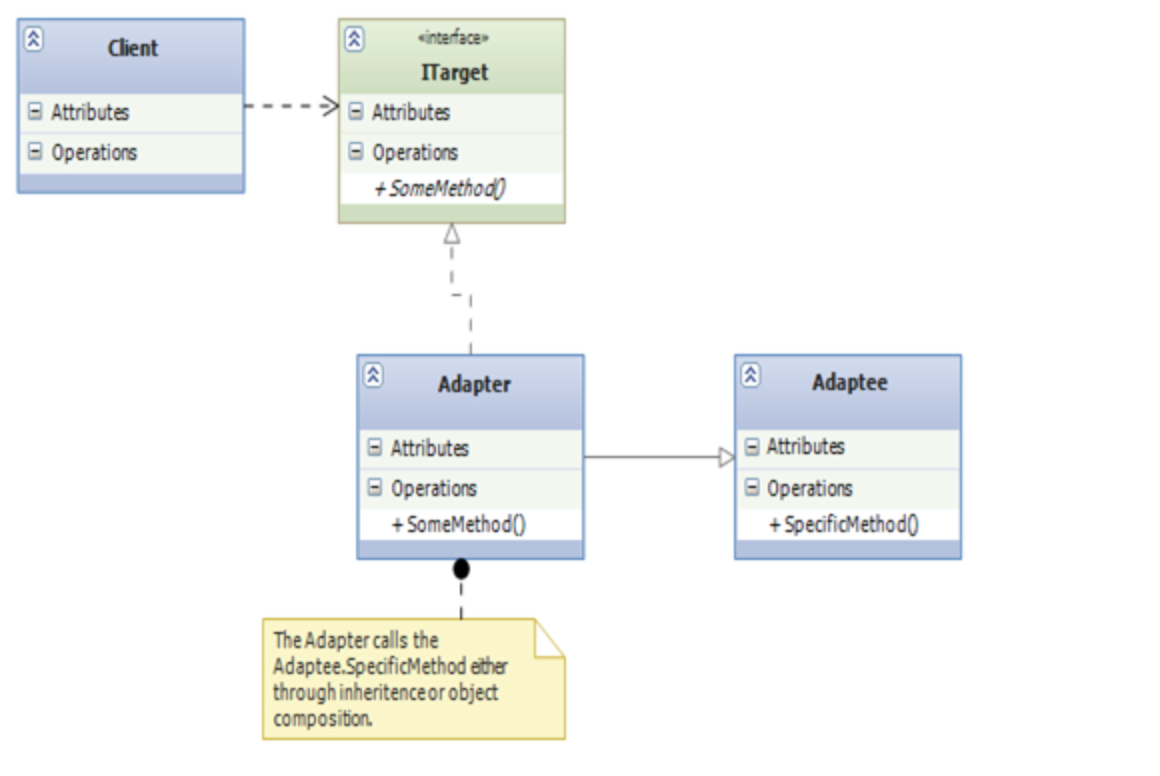
Adaptateur

## B - Description du Pattern

En génie logiciel, adaptateur (ou wrapper[1](https://fr.wikipedia.org/wiki/Adaptateur_(patron_de_conception)#cite_note-1)) est un patron de conception (*design pattern*) de type structure (*structural*). Il permet de convertir l'interface d'une classe en une autre interface que le client attend.

L’adaptateur fait fonctionner ensemble des classes qui n'auraient pas pu fonctionner sans lui, à cause d'une incompatibilité d'interfaces[2](https://fr.wikipedia.org/wiki/Adaptateur_(patron_de_conception)#cite_note-2).

## C - Exemple de diagramme UML (Problème et Solution)



* ITarget : C'est l'interface qui est utilisée par le client pour réaliser la fonctionnalité
* Adaptee : C'est la fonctionnalité que le client désire mais son interface n'est pas compatible avec le client
* Client : C'est la classe qui veut réaliser certaines fonctionnalités en utilisant le code de l'adaptee.
* Adapter : C'est la classe qui implémenterait ITarget et appellerait le code Adaptee que le client veut appeler.

## D - Les Domaines d’Application du Pattern

Une classe GestionDesStocks permettra de manipuler depuis d’autres endroits dans une chaîne de gestion commerciale l’ensemble des modules afférents à la gestion des stocks. Bien implémentée cette DP permet une incroyable modularité et la réutilisation simplifiée du code dans des applications complexes.

## E - Exemple de Code en C#

Ce code structurel illustre le modèle Adapter qui map l'interface d'une classe sur une autre afin qu'elles puissent fonctionner ensemble. Ces classes incompatibles peuvent provenir de différentes bibliothèques ou frameworks.

1. using System;
3. namespace DoFactory.GangOfFour.Adapter.Structural
4. {
5. /// <summary>
6. /// MainApp startup class for Structural
7. /// Adapter Design Pattern.
8. /// </summary>
9. class MainApp
10. {
11. /// <summary>
12. /// Entry point into console application.
13. /// </summary>
14. static void Main()
15. {
16. // Create adapter and place a request
17. Target target = new Adapter();
18. target.Request();
20. // Wait for user
21. Console.ReadKey();
22. }
23. }
25. /// <summary>
26. /// The 'Target' class
27. /// </summary>
28. class Target
29. {
30. public virtual void Request()
31. {
32. Console.WriteLine("Called Target Request()");
33. }
34. }
36. /// <summary>
37. /// The 'Adapter' class
38. /// </summary>
39. class Adapter : Target
40. {
41. private Adaptee \_adaptee = new Adaptee();
43. public override void Request()
44. {
45. // Possibly do some other work
46. //  and then call SpecificRequest
47. \_adaptee.SpecificRequest();
48. }
49. }
51. /// <summary>
52. /// The 'Adaptee' class
53. /// </summary>
54. class Adaptee
55. {
56. public void SpecificRequest()
57. {
58. Console.WriteLine("Called SpecificRequest()");
59. }
60. }
61. }

# II - FAÇADE

## A - Nom du Pattern

FAÇADE :

En génie logiciel, le patron de conception (ou design pattern) 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.

## B - Description du Pattern

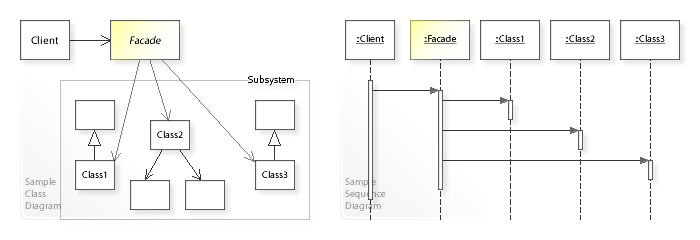
La façade encapsule la complexité des interactions entre les objets métier participant à un workflow.

L'utilisation d'une façade a les avantages suivants :

* Simplifier l'utilisation et la compréhension d'une bibliothèque logicielle car la façade possède des méthodes pratiques pour les tâches courantes,
* Rendre le code source de la bibliothèque plus lisible pour la même raison,
* Réduire les dépendances entre les classes utilisatrices et les classes internes à la bibliothèque puisque la plupart des classes utilisatrices utilisent la façade, ce qui autorise plus de flexibilité pour le développement du système,
* Rassembler une collection d'API complexes en une unique et meilleure API (orientée tâches utilisateurs).

Un adaptateur est utilisé quand la façade doit respecter une interface particulière et doit supporter un comportement polymorphique.

## C - Exemple de Diagramme UML



## D - Les Domaines d’Application du Pattern

Façade est utilisée lorsqu'une interface plus simple est souhaitée pour un objet sous-jacent. Alternativement, un adaptateur peut être utilisé lorsque l'encapsuleur doit respecter une interface particulière et doit supporter un comportement polymorphe. Un décorateur permet d'ajouter ou de modifier le comportement d'une interface lors de l'exécution.

|  |  |
| --- | --- |
| Pattern | Intention |
| [Adapter](https://en.wikipedia.org/wiki/Adapter_pattern) | Convertit une interface en une autre afin qu'elle corresponde à ce que le client attend |
| [Decorator](https://en.wikipedia.org/wiki/Decorator_pattern) | Dynamique ajoute la responsabilité à l'interface en enveloppant le code d'origine |
| Facade | Fournit une interface simplifiée |

Le motif de façade est généralement utilisé lorsque :

- une interface simple est nécessaire pour accéder à un système complexe

- un système est très complexe ou difficile à comprendre

- un point d'entrée est nécessaire pour chaque niveau de logiciel en couches

- les abstractions et implémentations d'un sous-système sont étroitement couplées

## E - Exemple de Code en C#

Ce code représente une Implémentation

|  |
| --- |
| **namespace** **DesignPattern.Facade**  {  **class** **SubsystemA**  {  **public** string OperationA1()  {  **return** "Subsystem A, Method A1\n";  }  **public** string OperationA2()  {  **return** "Subsystem A, Method A2\n";  }  }  **class** **SubsystemB**  {  **public** string OperationB1()  {  **return** "Subsystem B, Method B1\n";  }  **public** string OperationB2()  {  **return** "Subsystem B, Method B2\n";  }  }  **class** **SubsystemC**  {  **public** string OperationC1()  {  **return** "Subsystem C, Method C1\n";  }  **public** string OperationC2()  {  **return** "Subsystem C, Method C2\n";  }  }  **public** **class** **Facade**  {  **private** **readonly** SubsystemA a = **new** SubsystemA();  **private** **readonly** SubsystemB b = **new** SubsystemB();  **private** **readonly** SubsystemC c = **new** SubsystemC();  **public** **void** Operation1()  {  Console.WriteLine("Operation 1\n" +  a.OperationA1() +  b.OperationB1() +  c.OperationC1());  }  **public** **void** Operation2()  {  Console.WriteLine("Operation 2\n" +  a.OperationA2() +  b.OperationB2() +  c.OperationC2());  }  }  } |

# III - SINGLETON

## A - Nom du Pattern

SINGLETON:

Le singleton permet de s'assurer qu'une seule instance d'un objet donné sera instanciée pendant toute la durée de votre application. Une seule dans l'espace comme dans le temps, c'est-à-dire :

L'espace représenté par la mémoire - vous êtes certain de l'unicité de l'instance à un moment donné.

Le temps - vous vous assurer de l'unicité de l'instance à chaque appel. J'entends par la que vous êtes certain que c'est la même instance que vous référer quel que soit le temps écoulé entre deux appels.

## B - Description du Pattern

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.

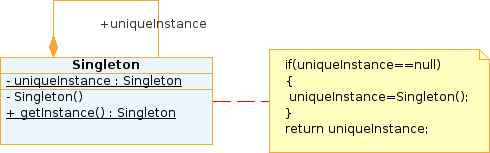
Une fois cette étape accomplie, il est possible d’instancier cette classe uniquement depuis elle-même, ce qui n’a pas beaucoup de sens. Comment allons-nous faire pour permettre aux développeurs de l’utiliser ?

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.

## C - Exemple de Diagramme UML



## D - Les Domaines d’Application du Pattern

Le Singleton répond à deux exigences :

* Garantir qu'une unique instance d'une classe donné sera créée
* Offrir un point d'accès universel à cette instance.

Ce design pattern est tout indiqué pour implémenter des services qui :

* Ils sont fonctionnellement uniques au sein de l'application (ex: système de logging centralisé, gestion de la configuration...)
* Ils doivent pouvoir être appelés par toutes les couches de l'application. Il serait en effet peu pratique de passer une référence au service à toutes les classes devant l'utiliser.

## E - Exemple de Code en C#

Ce code représente une Implémentation

|  |
| --- |
| **public** **class** **Singleton**  {  **private** **static** Singleton \_instance;  **static** **readonly** object instanceLock = **new** object();  **private** Singleton()  {    }  **public** **static** Singleton getInstance()  {  **lock** (instanceLock)  {  **if** (\_instance == **null**)  \_instance = **new** Singleton();    **return** \_instance; } } } |

# IV - Fabrique

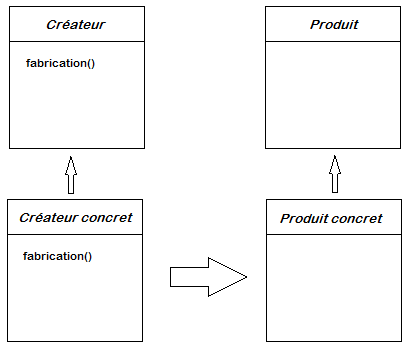
## A - Nom du Pattern

Fabrique

## B - Description du Pattern

Le pattern Fabrique permet l'instanciation d'objets non définis dans une classe concrète à partir d'une méthode d'instanciation issue d'une classe abstraite. Exemple : La création d'un disque dur, notamment le contrôleur de disque. Il existe plusieurs types de contrôleurs de disque (SATA, IDE, SCSI, SAS). Le disque dur correspond donc à la classe abstraite et les contrôleurs aux sous-classes concrètes.

## C - Exemple de Diagramme UML



## D - Les Domaines d’Application du Pattern

Le Créateur est une classe abstraite qui contient les méthodes qui vont manipuler le produit. Les Créateurs Concrets permettent de créer des produits concrets. Le Produit définit des méthodes qui permettent d'attribuer le nombre de tours, tester le bon fonctionnement du disque et de finaliser le disque. Dans notre exemple, Le Produit est la classe DisqueDur. Le Produit Concret est une classe qui correspond à un disque dur spécifique. Cette classe hérite de la classe Produit. Dans notre exemple, les produits concrets sont les classes DisqueDurATA et DisqueDurSCSI.

## E - Exemple de Code en C#

DisqueDur.cs

public abstract class DisqueDur

{

public abstract void setNbTours(String nombre);

public abstract void Tester();

public abstract void Finaliser();

}

FabriqueDisqueDur.cs

public abstract class FabriqueDisqueDur

{

public DisqueDur creerDisqueDur(String controleur)

{

DisqueDur disqueDur;

disqueDur = setControleur(controleur);

disqueDur.setNbTours("7200");

disqueDur.Tester();

disqueDur.Finaliser();

return disqueDur;

}

protected abstract DisqueDur setControleur(String typeControleur);

}

FabriqueDisqueDurATA.cs

public class FabriqueDisqueDurATA : FabriqueDisqueDur

{

override protected DisqueDur setControleur(String controleur)

{

DisqueDur disqueDurATA = null;

if (controleur == "ATA")

disqueDurATA = new DisqueDurATA();

return disqueDurATA;

}

}

FabriqueDisqueDurSCSI.cs

public class FabriqueDisqueDurSCSI : FabriqueDisqueDur

{

override protected DisqueDur setControleur(String controleur)

{

DisqueDur disqueDurSCSI = null;

if (controleur == "SCSI")

disqueDurSCSI = new DisqueDurSCSI();

return disqueDurSCSI;

}

}

DisqueDurSCSI.cs

public class DisqueDurSCSI: DisqueDur

{

public DisqueDurSCSI()

{

Console.WriteLine("Disque Dur SCSI cree");

}

public override void setNbTours(String nombre)

{

Console.WriteLine("Nb tours : " + nombre + " tours");

}

public override void Tester()

{

Console.WriteLine("Tests en cours..");

}

public override void Finaliser()

{

Console.WriteLine("Disque dur SCSI operationnel");

}

}

DisqueDurATA.cs

public class DisqueDurATA : DisqueDur

{

public DisqueDurATA()

{

Console.WriteLine("Disque Dur ATA cree");

}

public override void setNbTours(String nombre)

{

Console.WriteLine("Nb tours : " + nombre + " tours");

}

public override void Tester()

{

Console.WriteLine("Tests en cours..");

}

public override void Finaliser()

{

Console.WriteLine("Disque dur ATA operationnel");

}

}

Program.cs

class Program

{

static void Main(string[] args)

{

FabriqueDisqueDur disqueDurATA = new FabriqueDisqueDurATA();

disqueDurATA.creerDisqueDur("ATA");

FabriqueDisqueDur disqueDurSCSI = new FabriqueDisqueDurSCSI();

disqueDurSCSI.creerDisqueDur("SCSI");

Console.ReadKey();

}

}

# IV - Template Mmethod Pattern

## A - Nom du Pattern

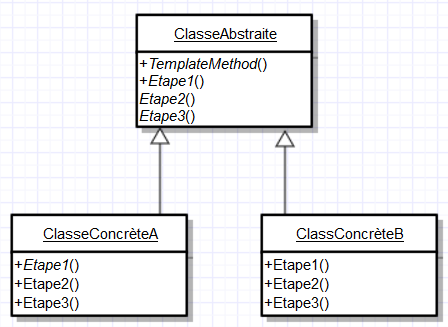
Patron de méthode (template method pattern) :

Définir le squelette d'un algorithme en déléguant certaines étapes à des sous-classes.

## B - Description du Pattern

Une classe possède un fonctionnement global. Le pattern Template Method permet de reporter dans des sous-classes certaines étapes des opérations d’un objet, ces étapes étant alors écrites dans les sous-classes.

## C - Exemple de Diagramme UML



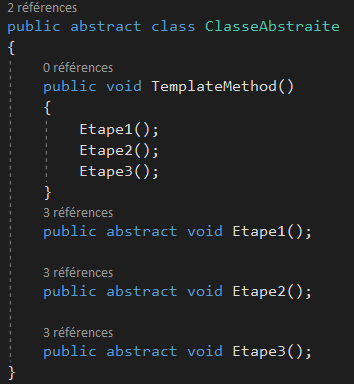
## D - Les Domaines d’Application du Pattern

Cela peut être le cas d'un document informatique. Le document a un fonctionnement global où il est sauvegardé. Pour la sauvegarde, il y aura toujours besoin d'ouvrir le fichier, d'écrire dedans, puis de fermer le fichier. Mais, selon le type de document, il ne sera pas sauvegardé de la même manière. S'il s'agit d'un document de traitement de texte, il sera sauvegardé en suite d'octets. S'il s'agit d'un document HTML, il sera sauvegardé dans un fichier texte.

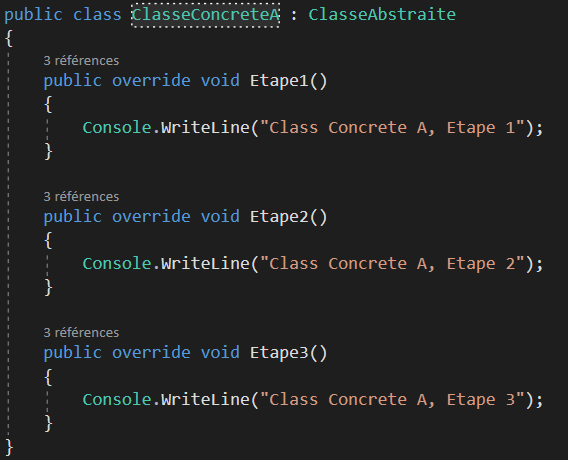
La partie générale de l'algorithme (sauvegarde) est gérée par la classe abstraite (document). La partie générale réalise l'ouverture, fermeture du fichier et appelle une méthode d'écriture. La partie spécifique de l'algorithme (écriture dans la fichier) est définie au niveau des classes concrètes (document de traitement de texte ou document HTML).

## E - Exemple de Code en C#

**Classe Abstraite :**



**Classe Concrète :**

****