|  |
| --- |
| DEFOIS, FARISSE, HAMANI, LOPPY, VERGINE |
| Documentation D’un Design Pattern |
| Façade |

|  |
| --- |
| sylvain FARISSE  26/10/2017 |

Table des matières

[Nom du Pattern 2](#_Toc497743707)

[Description du Pattern 2](#_Toc497743708)

[Exemple de Diagramme UML (Problème et Solution) 3](#_Toc497743709)

[Les Domaines d’Application du Pattern 4](#_Toc497743710)

[Exemple de Code en C# 4](#_Toc497743711)

# 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.

# Description du Pattern

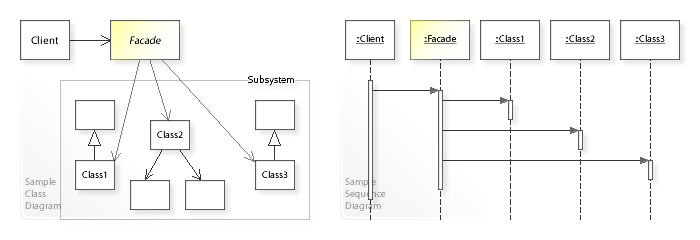
La façade encapsule la complexité des interactions entre les objets métier participant à un workflow.

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.

# Exemple de Diagramme UML



# 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

# 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());  }  }  } |