Fixed and Dynamic Catalog Pattern

ENGEENERING DEVELOPMENTS
CHRISTOPHE BOULAS
22 MAI 2022

Table des matières

Design pattern

Qu'est-ce qu'un Design Pattern ?

Un design pattern (ou en français : Patron de conception) en plus d'inclure les bonnes pratiques, est une technique ou méthode appliquée afin de répondre à une problématique de manière solide tout en restant simple d'utilisation.

Génie logiciel

Appelé ainsi, le génie logiciel couvre de nos jours aussi bien les applications clients lourds (application Windows par exemple) comme les sites web ou même les applications pour mobile et smartphone.

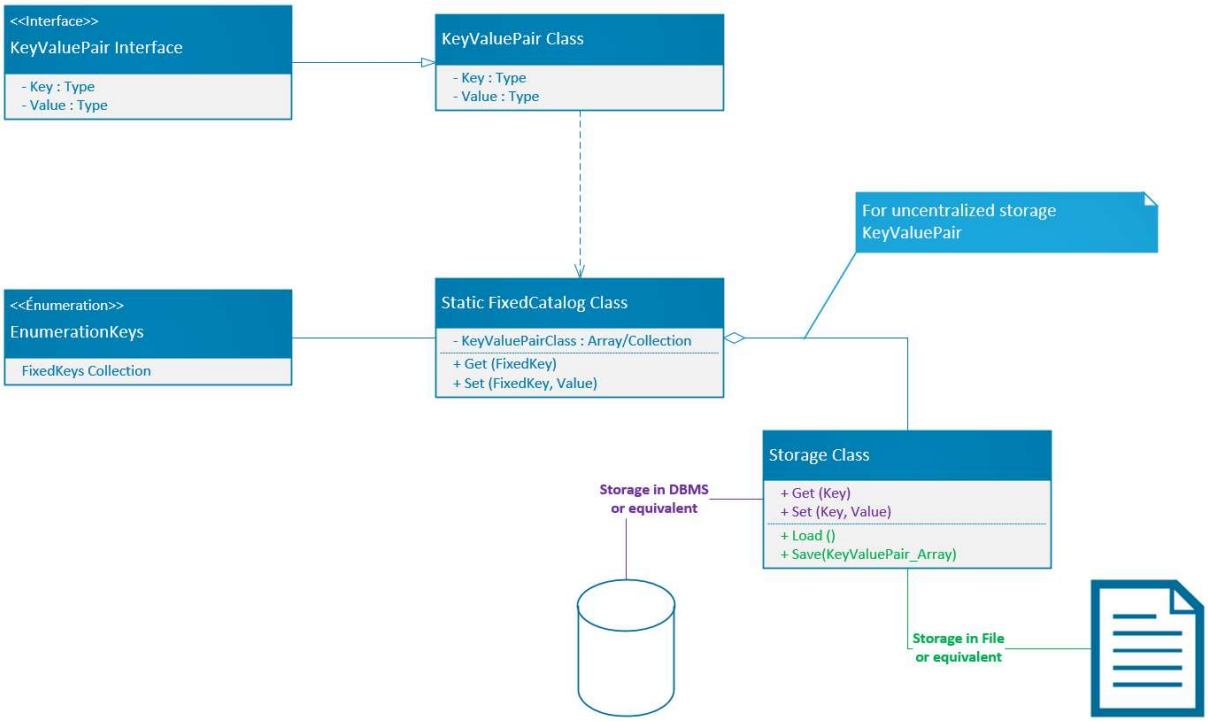
Si les design pattern sont monnaie courante dans ce domaine, il est pourtant assez récurrent de se retrouver confronter à certaines problématiques comme la gestion de paramètres d'application, ou accès à des requêtes de serveur de bases de données par exemple.

Pour répondre à cela, chacun y va plutôt de sa technique, certains développeurs inclus les paramètres en dur dans le programme, d'autres utilisent des fichiers externes, mais leur accès en est souvent assez personnel.

Les design pattern répondent-ils à cette problématique ? Dans la réalité, c'est très peu le cas car chacun stocke les informations différemment et de différentes manières.

Les 2 design pattern que j'ai créé et que j’utilise depuis plusieurs années répondent largement à ces problématiques en gestion de génie logiciel de façon sûr et pratique.

Fixed Catalog Pattern



La clé de ce pattern réside par l'utilisation de deux méthodes statiques « **Get** » et « **Set** » de la class « **FixedCatalog** » prenant toutes deux en premier argument une « clé » de type énumération issue de la classe d'énumération « **EnumerationKeys** ».

Cette class « **FixedCatalog** » héberge une variable de type tableau, collection ou équivalent de la class « **KeyValuePair** » prenant en charge une relation clé-valeur, elle-même répondant à l'interface « **KeyValuePair Interface** ».

Cet hébergement permettra ainsi de stocker l'ensemble des relations clé-valeur.

La clé de cette relation devra répondre à l'énumération « **EnumerationKeys.FixedKeys** ».

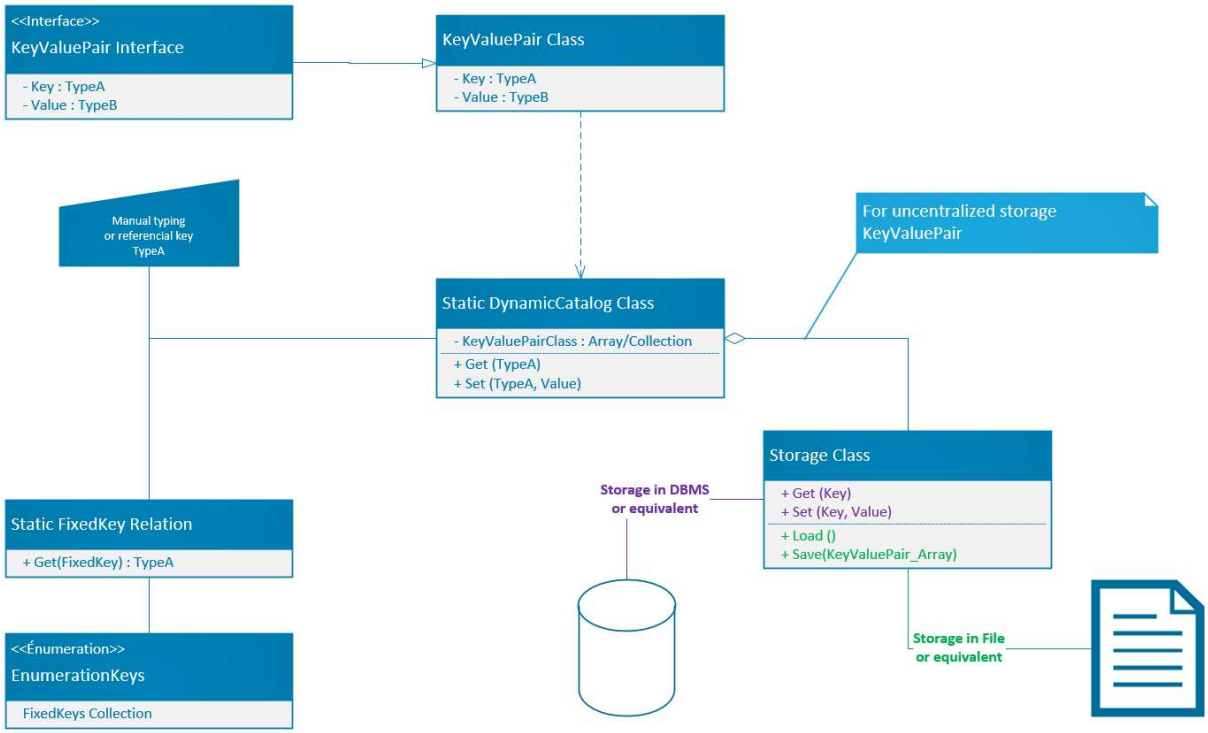
Dans le cas général, la class « **FixedCatalog** » sera instanciée de façon standard, mais dans un cas idéal celle-ci sera instanciée via une « **Factory** » afin de permettre d'inscrire automatiquement les valeurs par défaut des clés énumérées dans « **EnumerationKeys** ».

Afin de permettre le stockage de ces informations, une class « **Storage** » sera alors utilisée afin de communiquer avec soit un serveur de bases de données, soit un fichier.

Dans le cas d'une base de données, le programme utilisera alors les méthodes « **Get** » et « **Set** » afin de lire et sauvegarder au fur et à mesure des besoins les informations des relations clé-valeur.

Dans le cas d'un stockage via un fichier, même s'il est possible d'utiliser le même principe que pour la base de données en appliquant un design pattern « **Singleton** » et en utilisant les « **Lock** », il sera néanmoins préférable d'utiliser les méthodes « **Load** » et « **Save** » afin de charger l'ensemble des relations clé-valeur d'une traite et d'en sauvegarder le tout en une seule fois également.

Dynamic Catalog Pattern



Tout comme le « **Fixed Catalog pattern** », la clé de ce pattern réside par l'utilisation de deux méthodes statiques « **Get** » et « **Set** » de la class FixedCatalog.

Concernant la class « **DynamicCatalog** », elle héberge également une variable de type tableau, collection ou équivalent de la class « **KeyValuePair** » prenant en charge une relation clé-valeur, elle-même répondant à l'interface « **KeyValuePair Interface** ».

Cet hébergement permettra ainsi de stocker l'ensemble des relations clé-valeur.

La clé de cette relation devra répondre à l'énumération « **EnumerationKeys.FixedKeys** ».

Dans le cas général, la class « **FixedCatalog** « sera instanciée de façon standard, mais dans un cas idéal celle-ci sera instanciée via une « **Factory** » afin de permettre d'inscrire automatiquement les valeurs par défaut des clés énumérées dans « **EnumerationKeys** ».

Afin de permettre le stockage de ces informations, une class « **Storage** » sera alors utilisée afin de communiquer avec soit un serveur de bases de données, soit un fichier.

Dans le cas d'une base de données, le programme utilisera alors les méthodes « **Get** » et « **Set** » afin de lire et sauvegarder au fur et à mesure des besoins les informations des relations clé-valeur.

Dans le cas d'un stockage via un fichier, même s'il est possible d'utiliser le même principe que pour la base de données en appliquant un design pattern « **Singleton** » et en utilisant les « **Lock** », il sera néanmoins préférable d'utiliser les méthodes « **Load** » et « **Save** » afin de charger l'ensemble des relations clé-valeur d'une traite et d'en sauvegarder le tout en une seule fois également.

La différence notoire provient en l'utilisation d'une valeur de **TypeA** en tant que clé dans la relation clé-valeur.

Cette clé pourra ainsi provenir indirectement de l'énumération « **EnumerationKeys** », d'une saisie manuelle ou encore d'un référentiel tel une liste provenant d'une base de données ou d'un fichier de configuration.

Afin de maintenir le principe de simplicité, une class « **FixedKey Relation** » offrant la méthode « **Get** » permettra de retourner la clé de **TypeA** via la valeur d'énumération « **FixedKey** ».

Pourquoi utiliser une méthode de relation d'énumération vers un TypeA ?

Il faut pouvoir maintenir la solidité du pattern, et cette relation « ***FixedKey-TypeA*** » permet de maintenir les procédés de bases du pattern « **Fixed Catalog Pattern** », à savoir : l'instanciation directe ou et surtout dans ce cas-ci l'instanciation via une « **Factory** ».

Storage Class

Cette classe implémente 4 méthodes « **Get** », « **Set** » pour l'accès à un SGBD (ou équivalent) et « **Load** » et « **Save** » pour l'accès à un fichier (ou équivalent).

Toutefois, il est également possible de permettre une implémentation plus générique en implémentant uniquement 2 méthodes : « **Get** » et « **Set** » **ou** « **Load** » et « **Save** ».

Cette nouvelle implémentation ne sera alors appelée que lors de destruction de la class « **FixedCatalog** » ou « **DynamicCatalog** », en appliquant ainsi une simple boucle sur la collection de « **KeyValuePair** » afin d'en sauvegarder une à une les valeurs ou leur totalité d'un seul passage.

Je conseil fortement l'utilisation du pattern « **Facade** » afin d'implémenter au mieux cette class « **Storage** »

Dictionnaire

Il sera également possible d'implémenter une classe de type dictionnaire pour les class « **FixedCatalog** » ou « **DynamicCatalog** » afin de simplifier la conception de celle-ci et permettre la suppression de la conception de l'interface « **KeyValuePair** » ainsi de sa class.

Toutefois, dans le modèle « **FixedCatalog** » il conviendra d'overrider (réécrire) principalement les méthodes « **Get** » et « Set » afin de lever une exception si l'élément est introuvable, mais aussi et surtout de permettre l'ajout de relation clé-valeur lors de l'utilisation de la méthode « **Set** » sur le pattern « **DynamicCatalog** »