Paul Snauwaert
ISEN M1

Rendu Dossier

C#

Design Pattern | Singleton

- Définition
- Implémentation
- Avantages
- Inconvénients
- Exercices & corrections
- Sources

JUNIA Grande école d'ingénieurs

Définition Design Patterns :

En programmation, on appelle patron/motif/schéma de conception (traduit de l'anglais **design pattern**) les grandes lignes d'un arrangement caractéristique de modules répondant à problème de conception logicielle.

Ils assurent une bonne pratique **testée, expérimentée et approuvée** par des professionnels qui ont trouvé des modèles de conception répondant à des difficultés **récurrentes** auxquelles sont confrontées les développeurs. Quand il est reconnu, un design pattern est gage d'efficacité et de résultats établis objectifs.

Il existe différents types de patrons, ici nous allons nous intéresser aux patrons de **conception** (dont fait partie le pattern Singleton). Ils correspondent à une organisation des classes et traite **d'instances**, de rôles et collaboration.

Un patron de conception est défini par un **nom**, une description de la **difficulté rencontrée**, une description de la **solution apportée** et les **conséquences** de la mise en place du pattern.

On peut définir simplement le pattern singleton comme étant : un patron qui s'assure en tout instant de **l'unicité de l'instance** d'une classe en fournissant une interface permettant sa manipulation. L'objet à instance unique est régi par différentes méthodes permettant l'obtention de l'instance et s'assurant de l'impossibilité de créer une nouvelle instance de l'objet.

Le singleton est l'un des patrons l**es plus simples**, mais **les plus puissants** dans le développement de logiciels.

Définition Singleton :

Lorsque le patron de conception singleton est utilisé pour créer une unique instance de classe, celui-ci agit comme garant de **l'unicité** de cette instance. En outre, par son implémentation, la classe est rendue **accessible** par toute classe du même logiciel ou de la même application.

Afin d'éviter tout autre instance de la classe en question, le constructeur de l'instance rend le patron **privé** afin que l'instance ne puisse être créée qu'au sein-même du code contenu dans la classe. L'idée derrière ce procédé est de garantir que l'utilisateur ne peut interagir **qu'avec cette seule instance** : pour ce faire, l'instance est créée lors du premier appel puis est systématiquement renvoyée lors des appels suivants.

Une implémentation possible du pattern singleton est la suivante :

```
public class Singleton {
        private static Singleton instance;
        private Singleton() {}
        public static Singleton getInstance() {
                if (instance == null)
                {
                        instance = new Singleton();
                }
                return instance;
        }
}
```

Une variable privée statique instance est déclarée mais non définie, par défaut elle est donc **nulle**.

Le **constructeur** de la classe est rendu **privé** afin d'éviter toute construction à l'aide du keyword 'new' en dehors de la classe.

Afin d'accéder à l'instance il est nécessaire de passer par l'intermédiaire de la méthode **getInstance**(). Lors du premier appel, instance est nulle et est donc définie une fois pour toute puis renvoyée. Lors des appels suivants, la fonction renvoie systématiquement l'instance initialisée.

Cette implémentation, bien que correcte est « **bâclée** ». C'est-à-dire qu'elle possède ses limites et présente des défauts : si cette classe est utilisée dans un environnement multithread, il est possible que différents threads réalisent l'appel de la méthode **simultanément** résultant en une création **multiple** d'instances de la classe Singleton.

Ce genre d'implémentation est appelé « **Singleton naïf** ».

Pour résoudre ce problème, un autre design pattern pour le singleton est très connu, il s'agit du « **Singleton thread-safe** » dont l'objectif est d'empêcher la création de multiples instances lors de la première création de l'objet 'Singleton' grâce à une **synchronisation des threads**.

Le succès de la synchronisation repose sur l'ajout d'un **objet statique « lock »** partagé qui sert de **témoin** quant à la création ou non de l'instance. L'intérêt repose sur le fait que **seul un thread peut passer cette étape du code à la fois,** le premier thread réalise l'initialisation de l'instance. Les threads suivants, ayant dû attendre que le premier thread ait terminé d'opérer dans la partie du code concernée, récupèrent nécessairement l'instance créée par le **premier thread**.

Il faut noter que cette pratique peut s'avérer **coûteuse** en ce qui concerne les **performances** du programme car à chaque fois que l'instance est requise, le lock empêche de multiples threads d'accéder à l'instance **en même temps**.

Pour résoudre le problème, il est alors possible de passer d'un « **simple check** » thread Safe à un « **double check** ». Cela implique de réaliser un test supplémentaire pour savoir si l'instance a déjà été initialisée. Le cas échéant, ne passe pas par le lock !

Ces implémentations « thread-safe » du pattern singleton sont les suivantes :

- Simple Check

```
public class Singleton {
        private static Singleton instance;
        private Singleton() {}
        private static readonly object lock = new object();
        public static Singleton Instance {
                get{
                   lock(lock) {
                        if (instance == null) {
                              instance = new Singleton();
                        }
                        return instance;
                }
        }
}
```

- Double Check

```
public class Singleton {
        private static Singleton instance;
        private Singleton() {}
        private static readonly object lock = new object();
        public static Singleton Instance {
                get{
                   if (instance == null) {
                          lock(lock) {
                               if (instance == null) {
                                    instance = new Singleton();
                                }
                           }
                   return instance;
                }
        }
}
```

Avantages du pattern Singleton :

- Le pattern singleton peut être implémenter des interfaces
- Il peut servir à masquer les dépendances du programme
- Son initialisation est statique
- Il peut hériter d'autres classes
- Il est très simple à implémenter
- Il produit un unique point d'accès à une instance en particulier

Inconvénients du pattern Singleton :

Le côté « **libre accès** » que produit le pattern singleton en permettant aux différentes classes du projet de faire appel aux différents attributs et méthodes de la classe singleton rend **difficile l'unit testing** en cas de problème.

En effet, étant donné que '**tout le monde peut accéder à l'instance de n'importe où'**, si certaines interactions avec l'instance provoquent des erreurs, il est très difficile de **retracer l'origine** du problème.

D'autre part, il réduit le potentiel de multithreading au sein d'un même programme. Puisque l'accès au singleton nécessite la sérialisation d'un objet, les **performances** de l'application peuvent être **affectées**.

Exercices :

- Implémenter un singleton « **naïf** » et vérifier que 2 instances d'une même classe soit **identiques**.

```
Using System;

Public sealed class Singleton
{
        private Singleton() {}
        private static Singleton _instance;

        public static Singleton getInstance()
        {
                if (_instance == null)
                {
                        _instance = new Singleton();
                }
                return _instance;
        }
}

Public class Exercise
{
        Static void Main(string[] args)
        {
                Singleton singleton1 = Singleton.getInstance();
                Singleton singleton2 = Singleton.getInstance();

                if(singleton1 == singleton2)
                        Console.WriteLine("Singleton working successfully!");
                else
                        Console.WriteLine("Singleton not working as intended.");
        }
}
```

Après exécution du programme, renvoie :

```
Singleton working successfully!
```

- Implémenter un singleton **« thread safe »** réaliser le même test en utilisant 2 threads.

```
Using System;

public class Singleton {
        private static Singleton instance;
        private Singleton() {}
        private static readonly object lock = new object();
        public string name { get; set; }
        public static Singleton getInstance(string InstanceName) {
                get{
                   if (instance == null) {
                          lock(lock) {
                                if (instance == null) {
                                     instance = new Singleton();
                                     instance.name = InstanceName;
                                 }
                          }
                   return instance;
                }
        }
}
Public class Exercise
{
        Static void Main(string[] args)
        {
                Thread thread1 = new Thread(() =>
                {
                       Singleton singleton1 = Singleton.getInstance(“Instance1”);
                       Console.WriteLine(singleton1.Name);
                });
                Thread thread2 = new Thread(() =>
                {
                       Singleton singleton2 = Singleton.getInstance(“Instance2”);
                       Console.WriteLine(singleton2.Name);
                });

                Thread1.Start();
                Thread2.Start();
                Thread1.Join();
                Thread2.Join();
        }
}
```

Après exécution du programme, renvoie :

```
Instance1
Instance1
```

Sources :

https://www.usabilis.com/design-patterns-pour-la-composition-interfaces/

https://www.ionos.fr/digitalguide/sites-internet/developpement-web/quest-ce-que-le-singleton-pattern/

https://fr.wikipedia.org/wiki/Patron_de_conception

https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/