Design Patterns

Creational Patterns – Singleton Pattern

By: Alpesh Chaudhari

This document covers the Singleton pattern, a key concept in Creational Design Patterns. The Singleton pattern is used to ensure that a class has only one instance, and it provides a global point of access to that instance.

Singleton pattern

The Singleton pattern is a creational design pattern used to ensure that a class has only one instance, and it provides a global point of access to that instance. This is useful when only one instance of a class is needed to coordinate actions across the system. In C#, it's commonly used for managing resources like database connections, logging, configuration settings, etc.

Use Case for Singleton

A Singleton is useful when:

1. Global Access is required, e.g., logging.
2. Single Resource Instance: When a single instance manages shared resources or settings, like a configuration manager or database connection pool.
3. Thread Safety: In multi-threaded applications, Singleton ensures that only one instance of a resource is created, avoiding synchronization issues.

Example :

Imagine a scenario where a ConfigurationManager Singleton reads configuration settings from a file or environment variables, and we want to ensure only one instance manages the configurations throughout the application. Different parts of the program may attempt to access the configuration settings at the same time, so we need thread-safety.

Here's how you can structure your code:

```
namespace DesignPatterns.CreationalPatterns.SingletonPattern.Example1
{
    class ConfigurationManager
    {
        private ConfigurationManager() { }

        private static ConfigurationManager _instance;
        private static readonly object _lock = new object();

        public string ConfigurationValue { get; set; }

        public static ConfigurationManager GetInstance(string value)
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new ConfigurationManager();
                        _instance.ConfigurationValue = value;
                    }
                }
            }
            return _instance;
        }

        public void ShowConfiguration()
        {
            Console.WriteLine("Current Configuration: " + ConfigurationValue);
        }
    }
```

```
public class Client
{
    public void Main()
    {

        Thread process1 = new Thread(() =>
        {
            TestConfigurationManager("Database=SQLServer");
        });
        Thread process2 = new Thread(() =>
        {
            TestConfigurationManager("Database=PostgreSQL");
        });

        process1.Start();
        process2.Start();

        process1.Join();
        process2.Join();
    }

    public static void TestConfigurationManager(string value)
    {
        ConfigurationManager configManager = ConfigurationManager.GetInstance(value);
        configManager.ShowConfiguration();
    }
}
```

Explanation of the Code

- **ConfigurationManager:** Manages application settings. Only one instance of ConfigurationManager exists in the application.
- **GetInstance Method:** Implements lazy initialization and double-checked locking to ensure only one instance is created, even in a multithreaded environment.
- **Threaded Access:** Two threads attempt to get an instance of ConfigurationManager with different values (Database=SQLServer and Database=PostgreSQL). However, due to the Singleton pattern, only the first value will be used.

Output

When you run this code, the output will consistently show the same configuration value (either "Database=SQLServer" or "Database=PostgreSQL"), depending on which thread accesses it first. This confirms that only one instance of ConfigurationManager is created and reused.

Note: For further examples of Singleton Pattern use cases in real-life development, please refer to my GitHub repository titled "DesignPattern." In this repository, I have included an additional example of the Singleton pattern eg. **Logger**.

GitHub: https://github.com/iam-alpesh/DesignPatterns.git

Thank You

I would like to express my gratitude to the following individuals and resources that helped me in understanding and implementing the Singleton pattern:

References

1. **Dive into Design patterns** by Alexander Shvets. Refactoring.Guru, 2021.
2. **Desing Patterns**. Refactoring Guru, https://refactoring.guru/