International Journal of Business Quantitative Economics and Applied Management Research
Volume-6, Issue-10, 2021 ISSN No: 2349-5677

EXPLORING THE SINGLETON DESIGN PATTERN IN C#: CHALLENGES, AND SOLUTIONS

AzraJabeen Mohamed Ali
Azra.jbn@gmail.com
Independent researcher, California,USA

Abstract

This paper discusses the thorough exploration of the Singleton design pattern.Common and recurring issues that developers encounter when creating applications or during the software application lifecycle include object creation and disposal, object-to-object interaction, class structure that promotes cohesiveness and loose coupling, bug fixes that minimize source code changes, etc. In order to reduce issues after deployment, design patterns are utilized to address these often-recurring issues during the development stage. A particular implementation for a given object-oriented programming problem is suggested by a design pattern. The Singleton design pattern, for instance, suggests the ideal method to develop a class that can only have one object if you want to make sure that there is only one instance of the class.The study's main research question explores when to applysingleton patternand how to apply.It also provides a thorough analysis of challenges after the implementation of singleton pattern. This paper is therefore meant to be more development-environment centered and infrastructure agnostic.Developers and architects who wish to concentrate on code, patterns, and implementation specifics will find this part most interesting.

Keywords: Singleton, Design patterns, Static, eager initialization, static constructor, Thread safe, Lazy loading

I. INTRODUCTION

Design Patterns:

A design pattern is a method of problem-solving that can be used in a variety of contexts. It serves as a blueprint or guideline that offers a consistent method for addressing a certain issue.Design patterns are a collection of tried-and-true fixes for typical program design issues. Understanding patterns is helpful to apply object-oriented design concepts to tackle a variety of difficulties.These patterns are frequently used to enhance code reuse, control software system complexity, and guarantee best practices are followed.

Patterns are classified based on their goal or objective. Three primary categories of patterns are Creational patterns, Structural patterns, and Behavioral patterns Fig-1.

International Journal of Business Quantitative Economics and Applied Management Research
Volume-6, Issue-10, 2021 ISSN No: 2349-5677

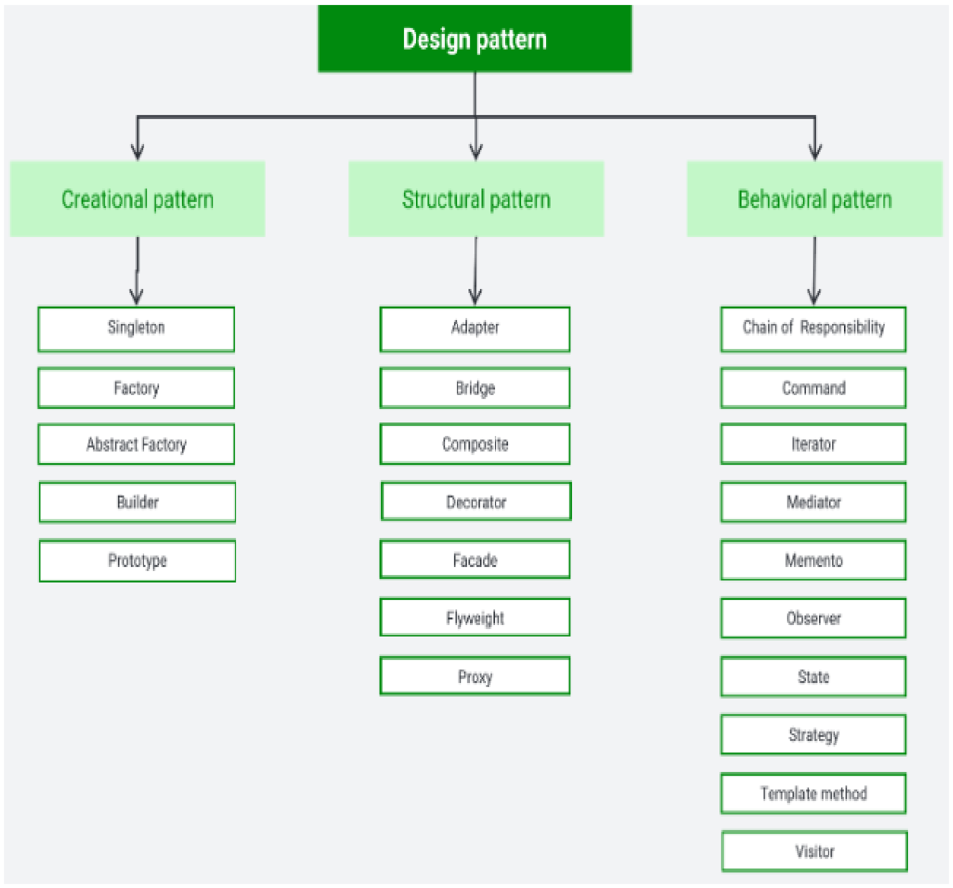


Fig-1

Creational patterns:
The process of creating items is the subject of creational design patterns, which concentrate on increasing the system's flexibility and dynamicity in this regard.There are many different kinds of creative design patterns, such as the Singleton, Factory, Abstract Factory, Builder, and Prototype patterns.

Structural patterns:
Structural Design Patterns address issues with the composition and assembly of classes and objects to create more expansive structures that are effective and adaptable. Inheritance is used by structural class patterns to create interfaces or implementations. There are many different kinds of structural design patterns, such as the Adapter, Bridge, Composite, Decorator, Façade, Flyweight and Proxy patterns.

Behavioral patterns:
Algorithms and the distribution of duties among objects are the focus of behavioral patterns. In addition to describing patterns of objects or classes, behavioral patterns also explain patterns of communication between them. Complex control flows that are challenging to understand at

International Journal of Business Quantitative Economics and Applied Management Research

Volume-6, Issue-10, 2021 ISSN No: 2349-5677

runtime are characterized by these patterns. There are many different kinds of behavioral design patterns, such as the Chain of responsibility, Command, Iterator, Mediator,Memento, Observer, State, Strategy, Template Method and Visitor patterns.

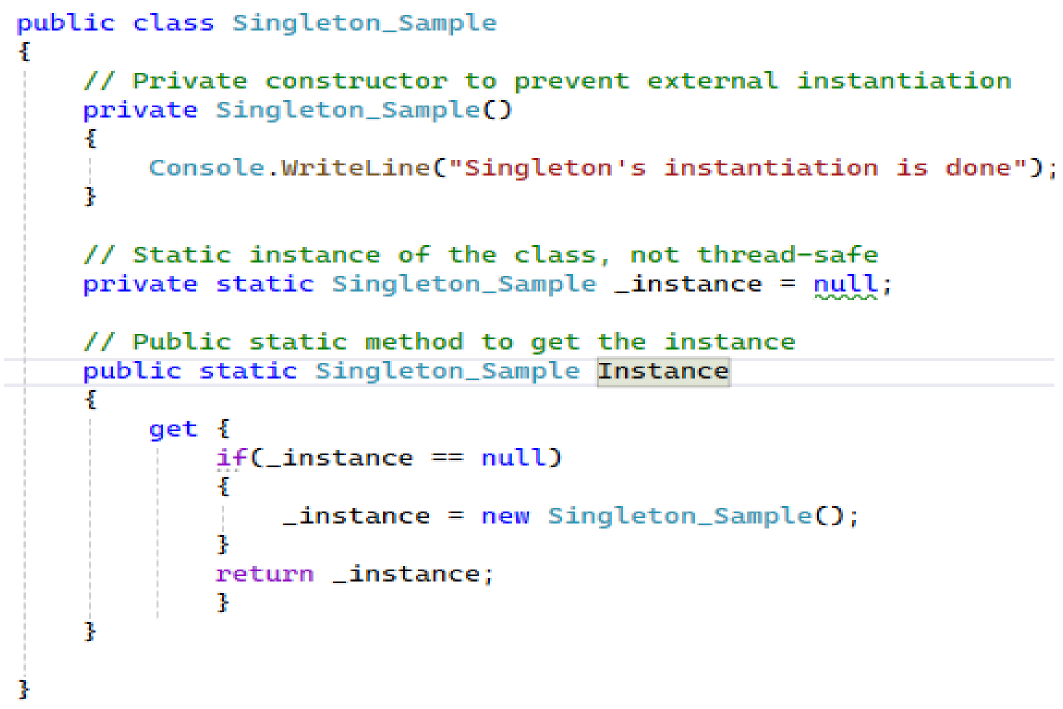
Singleton pattern:
This paper focuses on Singleton pattern which is one of the kinds of creational design pattern.One of the most well-known patterns in software engineering is the singleton pattern. A singleton is essentially a class that only permits the creation of a single instance of itself and typically provides straightforward access to that instance.

When to apply the singleton pattern:
It's perfect for situations where centralized control is needed, such asdatabase connection, a global configuration, a logging system, or a caching system. The singleton is the best option where there is only one instance of a particular object in the application and many modules need to access it. The singleton obviously won't function if the class has more than one instance.

How to apply the singleton pattern:
In C#, a singleton pattern can be implemented in a variety of ways.
1. Singleton with no thread safe,
2. Singleton with thread safe.
3. Singleton with thread safe by double check lock
4. Singleton with multi thread without lock
5. Singleton using Lazy<T>

Singleton with no thread safe:
- Fig-2 In the below implementation of singleton pattern with no thread safe feature has certaincharacteristics like private constructor, static instance, lazy instantiation, and no thread safety.
- There is only one private, parameterless constructor “Singleton_Sample”. This stops it from being instantiated by other classes, which would be against the pattern and which also prohibits subclassing; if a singleton may be subclassed twice, the pattern is broken if each of those subclasses is able to generate an instance.
- If a single instance of a base type is required but unsure of the exact type until runtime, then factory pattern is the best choice.
- The “_instance” variable is declared as a static field. This ensures that the instance is shared across all calls to the Instance property.
- The instance of “Singleton_Sample” is only created when Instance is accessed for the first time.
- This implementation's primary flaw is that it lacks thread safety. In a multi-threaded application, think about the following situation:
- In order to build the instance, Thread A first determines whether _instance is null.
- Because Thread A hasn't finished the instantiation yet, Thread B follows suit and discovers that _instance is null.

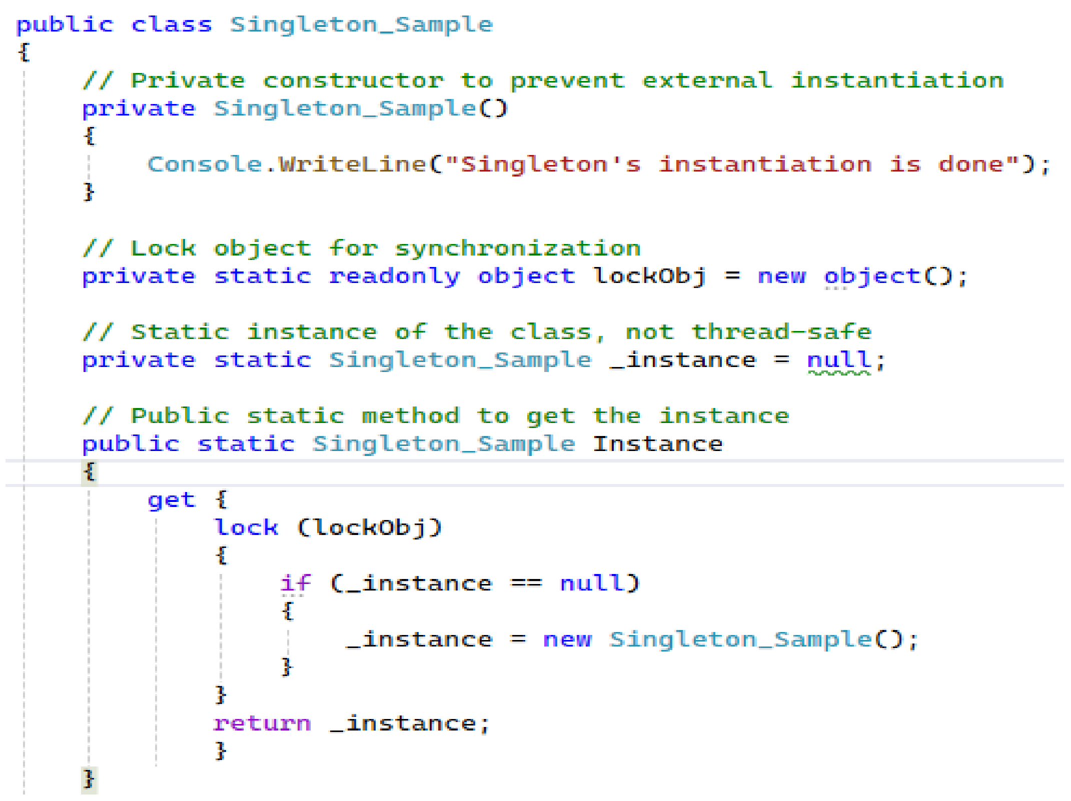
- The Singleton principle will be broken since two distinct instances of the same "Singleton_Sample" object will be produced when both threads instantiate it.



```
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }

    // Static instance of the class, not thread-safe
    private static Singleton_Sample _instance = null;

    // Public static method to get the instance
    public static Singleton_Sample Instance
    {
        get {
            if(_instance == null)
            {
                _instance = new Singleton_Sample();
            }
            return _instance;
            }
    }

}
```

Fig-2

Singleton with thread safe:
The code that follows is thread-safe.To provide thread safety,a static readonly object (lockObj) is used to synchronize access to the Instance property.It guarantees that only one thread will generate an instance and fixes the memory barrier problem.The code can contain just one thread at a time. When the second thread enters it, the expression will evaluate as false because the first thread will have already generated the object.This resolves the problem of thread safety. However, because only one thread may use the Instance property at once, it is sluggish.



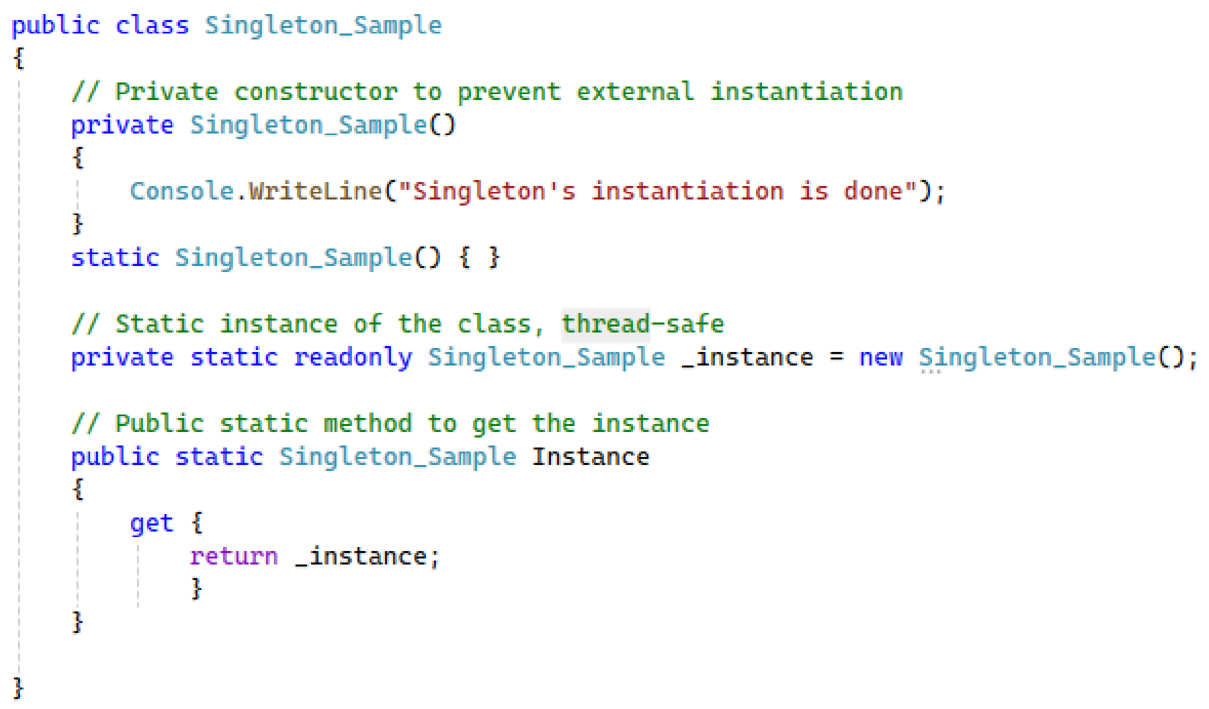
```
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }

    // Lock object for synchronization
    private static readonly object lockObj = new object();

    // Static instance of the class, not thread-safe
    private static Singleton_Sample _instance = null;

    // Public static method to get the instance
    public static Singleton_Sample Instance
    {
        get {
            lock (lockObj)
            {
                if (_instance == null)
                {
                    _instance = new Singleton_Sample();
                }
            }
            return _instance;
            }
    }
```

Fig-3

Singleton with thread safe without lock:

Fig-4 This implementation ensures thread safety and eager initialization.This kind of implementation only runs once per application because it has a static constructor.The _instance is created eagerly when the class is first loaded (due to private static readonly Singleton_Sample _instance = new Singleton_Sample();) Eager initialization is the process of creating the instance before any thread accesses it, at the moment the class is first accessed. Because the Common Language Runtime (CLR) makes sure that static fields are initialized just once in a thread-safe way when the class is loaded, this is thread-safe.The field _instance is marked as readonly, which means it can only be assigned once — in this case, during initialization. This ensures that the instance cannot be reassigned.Eager initialization ensures that the instance is created when the class is first used, and since static constructors are thread-safe in .NET, there is no risk of multiple threads creating separate instances.The Instance property is a public static property that gives access to the singleton instance. This guarantees that there is a single instance used throughout the application's lifetime.

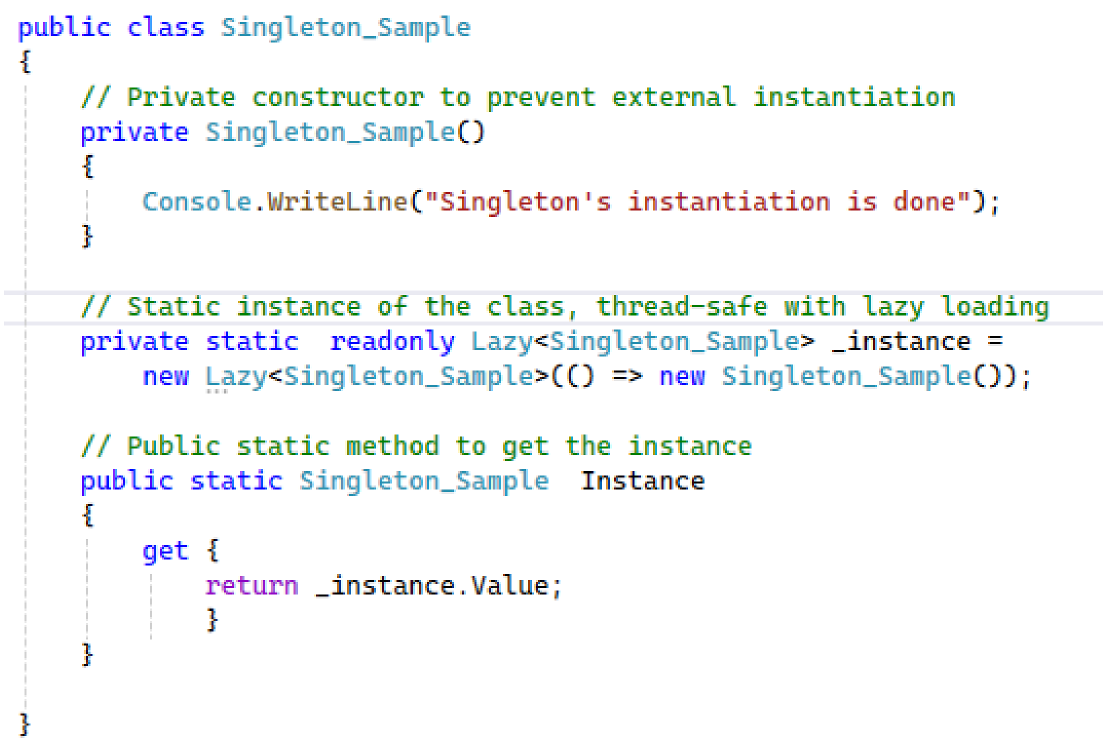
International Journal of Business Quantitative Economics and Applied Management Research
Volume-6, Issue-10, 2021 ISSN No: 2349-5677



```
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }
    static Singleton_Sample() { }

    // Static instance of the class, thread-safe
    private static readonly Singleton_Sample _instance = new Singleton_Sample();

    // Public static method to get the instance
    public static Singleton_Sample Instance
    {
        get {
            return _instance;
            }
    }

}
```

Fig-4

Singleton using Lazy<T>:
Fig-5 The below implementation shows the Singleton Pattern using lazy loading. Lazy<T> guarantees that the creation procedure is thread-safe and that the singleton instance is only produced when it is first accessedand it can be reused by later threads without requiring a new instance to be created.This makes the code clearer and more effective by doing away with the requirement for manual locking or synchronization.Lazy initialization is provided by the Lazy type, which means that the instance of Singleton_Sample is only produced upon the initialization of the Value property of _instance.The thread-safety is taken care of by the Lazy class. It makes use of LazyThreadSafetyMode.ExecutionAndPublication by default, which guarantees that only one thread can construct the instance and that all other threads will receive the same instance after it is produced.For a thread-safe singleton with lazy initialization, Lazy is the perfect choice. Without the need for intricate locking mechanisms or manual synchronization, it is straightforward, effective, and thread-safe.



```
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }

    // Static instance of the class, thread-safe with lazy loading
    private static  readonly Lazy<Singleton_Sample> _instance =
        new Lazy<Singleton_Sample>(() => new Singleton_Sample());

    // Public static method to get the instance
    public static Singleton_Sample  Instance
    {
        get {
            return _instance.Value;
            }
    }

}
```

Fig-5

Can we bypass Singelton pattern by any techniques?
Yes we can by various techniques. Even though we avoided multiple instance creation of singleton class by using Double checking loack or Eager instance creation, instances can still be created using

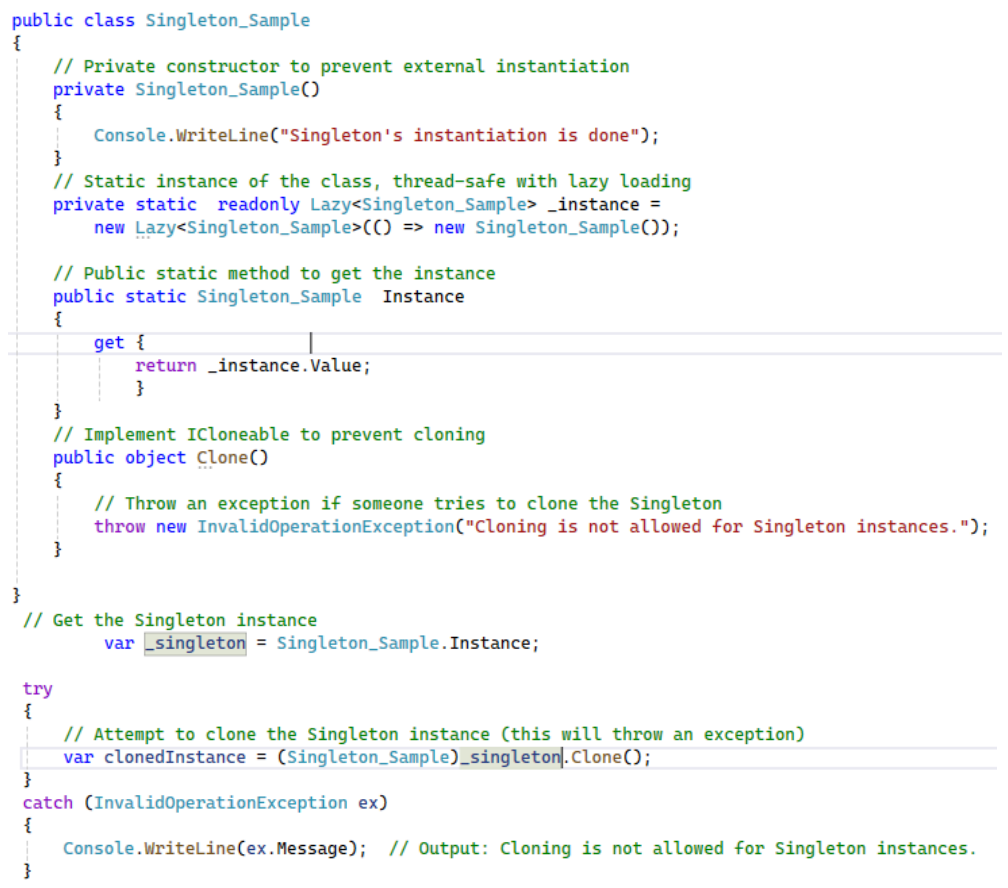
- Cloning: Implementing ICloneable or MemberwiseClone.
- Serialization: Serializing or deserializing the singleton object.
- Reflection: Using reflection to access private constructors and fields.
- Inheritance or subclassing:

Solutions to avoid Singleton instance creation by cloning:
We can prevent cloning using

- ICloneable Interface
- MemberwiseClone method.

The Clone method or MemberwiseClone method can be explicitly overridden to throw an error if there is a singleton instance creation using clone, preventing the creation of extra instances of the class. By doing this, the integrity of the Singleton pattern will be preserved and any effort to clone the Singleton instance will be prevented.The below implementation shows the prevention usingICloneable interface and override the Clone method in Singleton class to throw an exception when cloning is attempted. This will prevent the creation of a new instance via cloning Fig-6

International Journal of Business Quantitative Economics and Applied Management Research
Volume-6, Issue-10, 2021 ISSN No: 2349-5677



```
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }
    // Static instance of the class, thread-safe with lazy loading
    private static  readonly Lazy<Singleton_Sample> _instance =
        new Lazy<Singleton_Sample>(() => new Singleton_Sample());

    // Public static method to get the instance
    public static Singleton_Sample  Instance
    {
        get {
            return _instance.Value;
            }
    }
    // Implement ICloneable to prevent cloning
    public object Clone()
    {
        // Throw an exception if someone tries to clone the Singleton
        throw new InvalidOperationException("Cloning is not allowed for Singleton instances.");
    }

}
 // Get the Singleton instance
        var _singleton = Singleton_Sample.Instance;

 try
 {
     // Attempt to clone the Singleton instance (this will throw an exception)
     var clonedInstance = (Singleton_Sample)_singleton.Clone();
 }
 catch (InvalidOperationException ex)
 {
     Console.WriteLine(ex.Message);  // Output: Cloning is not allowed for Singleton instances.
 }
```

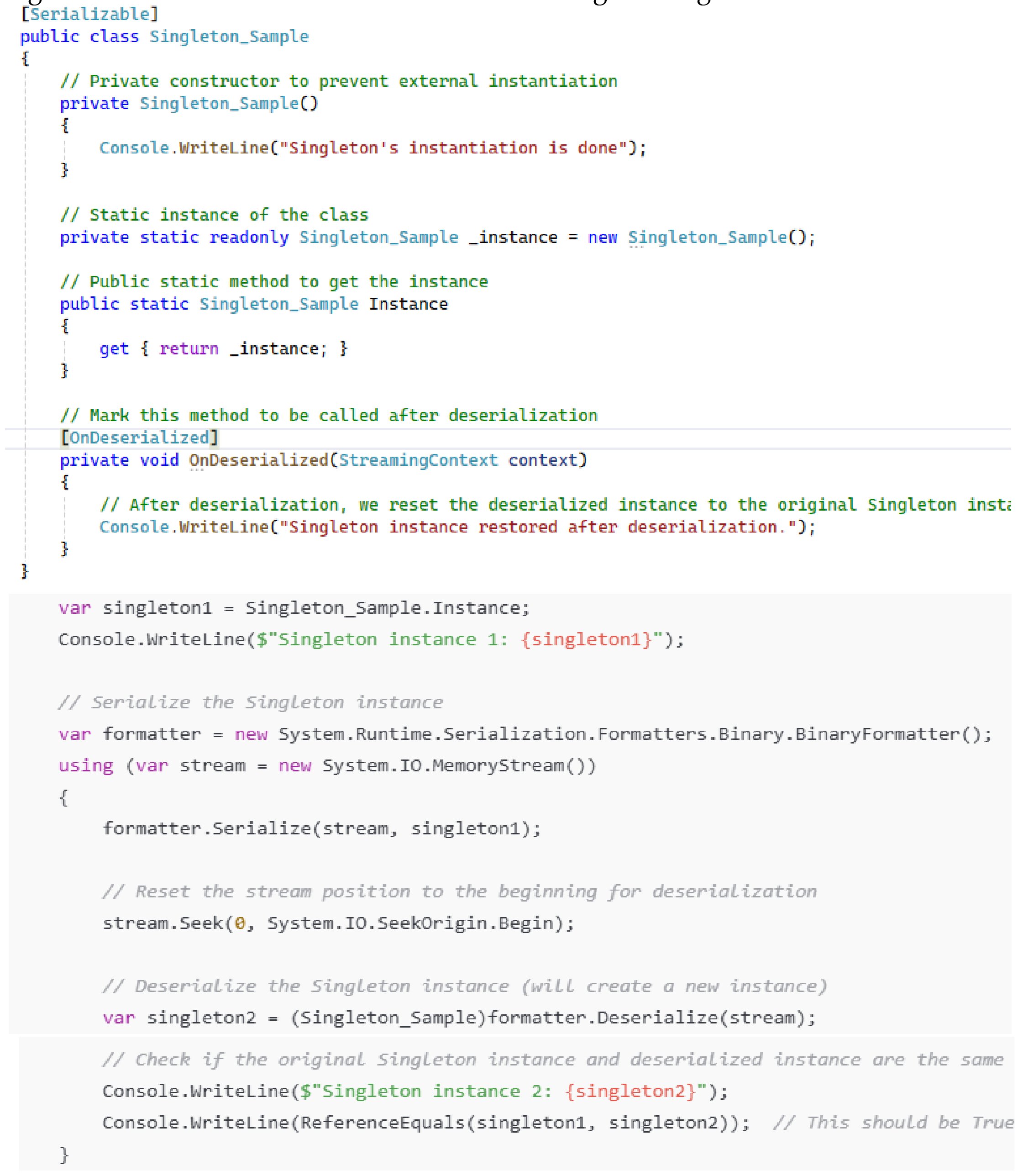
Fig-6

By overriding the Clone and MemberwiseClone methods—which are frequently used for object copying—to generate exceptions, we make sure that the Singleton pattern is maintained and that no new instances are created.

Solutions to avoid Singleton instance creation by serialization:
The Singleton pattern must be preserved throughout the deserialization procedure in order to avoid the serialization and deserialization of multiple instances of a Singleton class. When deserializing an object in C#, serialization usually generates a new instance of a class, avoiding the Singleton control.
By default, when you serialize and deserialize a Singleton class, a new instance of the class is created during the deserialization process. This breaks the Singleton pattern by creating a second instance of the class.
During serialization and deserialization, it can be avoided from creating numerous instances by using SerializationInfo or overriding the OnDeserialized method which stops a new instance from being created by deserialization and makes sure that the Singleton instance is always returned.
Marking a method with the OnDeserialized attribute—which will be invoked once an object is

International Journal of Business Quantitative Economics and Applied Management Research
Volume-6, Issue-10, 2021 ISSN No: 2349-5677

deserialized—is the easiest way to deal with this. The deserialized object can be set to the current Singleton instance using this method.
After deserialization, the ReferenceEquals(singleton1, singleton2) check will return True, meaning the deserialized instance is the same as the original Singleton instance.



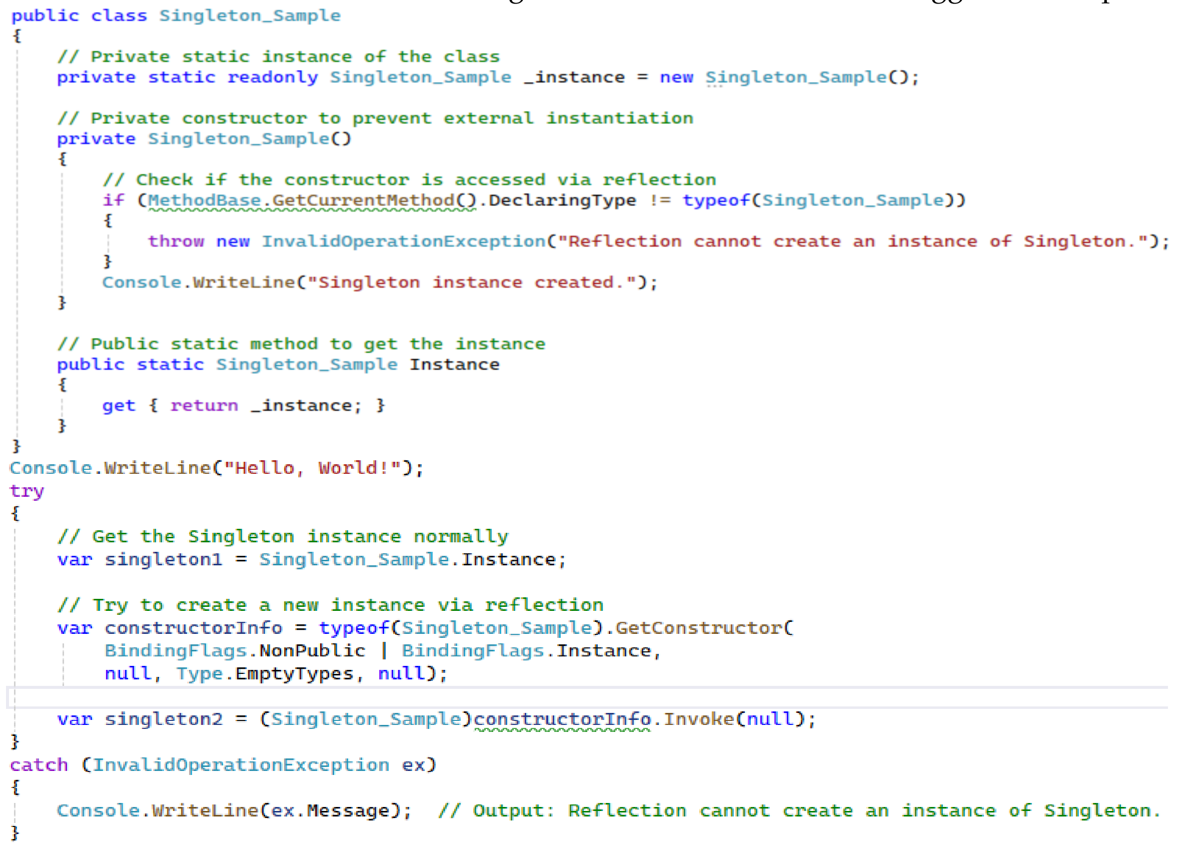
```
[Serializable]
public class Singleton_Sample
{
    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        Console.WriteLine("Singleton's instantiation is done");
    }

    // Static instance of the class
    private static readonly Singleton_Sample _instance = new Singleton_Sample();

    // Public static method to get the instance
    public static Singleton_Sample Instance
    {
        get { return _instance; }
    }

    // Mark this method to be called after deserialization
    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
        // After deserialization, we reset the deserialized instance to the original Singleton insta
        Console.WriteLine("Singleton instance restored after deserialization.");
    }
}
```

```
var singleton1 = Singleton_Sample.Instance;
Console.WriteLine($"Singleton instance 1: {singleton1}");

// Serialize the Singleton instance
var formatter = new System.Runtime.Serialization.Formatters.Binary.BinaryFormatter();
using (var stream = new System.IO.MemoryStream())
{
    formatter.Serialize(stream, singleton1);

    // Reset the stream position to the beginning for deserialization
    stream.Seek(0, System.IO.SeekOrigin.Begin);

    // Deserialize the Singleton instance (will create a new instance)
    var singleton2 = (Singleton_Sample)formatter.Deserialize(stream);

    // Check if the original Singleton instance and deserialized instance are the same
    Console.WriteLine($"Singleton instance 2: {singleton2}");
    Console.WriteLine(ReferenceEquals(singleton1, singleton2));  // This should be True
}
```

Fig-7

Solutions to avoid Singleton instance creation by Subclass or Inheritance:
By providing private constructor, a subclass cannot be instantiated to create another instance.

Solutions to avoid Singleton instance creation by Reflection:
Using reflection, it's possible to create a new instance of a class even if the constructor is private, thus violating the Singleton pattern. By below approache it is possible to stop the instance creation.
If accessed by reflection, raise an exception in the constructor.
Fig-8 If a Singleton instance is accessed by reflection, you can throw an exception in the constructor to stop it from being created. This can be accomplished by examining the approach that is being used at the moment, which the system can identify.Introspection.The function MethodBase.GetCurrentMethod(). The constructor of the Singleton_Sample class is private, as expected in the Singleton pattern. This restricts external instantiation of the class. Inside the constructor, MethodBase.GetCurrentMethod().DeclaringType checks the current method being executed. If it's not the Singleton_Sample constructor, it indicates that the constructor was called via reflection. If the constructor is accessed via reflection, it throws an InvalidOperationException to prevent creating a new instance. In the Main method, an attempt is made to create a new instance of the Singleton via reflection. This will trigger the exception.



```
public class Singleton_Sample
{
    // Private static instance of the class
    private static readonly Singleton_Sample _instance = new Singleton_Sample();

    // Private constructor to prevent external instantiation
    private Singleton_Sample()
    {
        // Check if the constructor is accessed via reflection
        if (MethodBase.GetCurrentMethod().DeclaringType != typeof(Singleton_Sample))
        {
            throw new InvalidOperationException("Reflection cannot create an instance of Singleton.");
        }
        Console.WriteLine("Singleton instance created.");
    }

    // Public static method to get the instance
    public static Singleton_Sample Instance
    {
        get { return _instance; }
    }
}
Console.WriteLine("Hello, World!");
try
{
    // Get the Singleton instance normally
    var singleton1 = Singleton_Sample.Instance;

    // Try to create a new instance via reflection
    var constructorInfo = typeof(Singleton_Sample).GetConstructor(
        BindingFlags.NonPublic | BindingFlags.Instance,
        null, Type.EmptyTypes, null);

    var singleton2 = (Singleton_Sample)constructorInfo.Invoke(null);
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.Message);  // Output: Reflection cannot create an instance of Singleton.
}
```

Fig-8

When to Avoid Singleton Pattern:
When there is no need global access or global state, then using dependency injection or a factory pattern would be more appropriate than Singleton pattern.When working in a multithreaded environment, the Singleton pattern may result in difficult-to-resolve synchronization or race situation issues.

II. CONCLUSION

The singleton design offers a global point of access and is simple to implement. It guarantees a single instance and is helpful for shared resources like as configuration, caching, and logging. Lazy initialization is used to postpone the use of resources.To avoid the drawbacks of the Singleton pattern, proper design patterns like factories or dependency injection may occasionally be preferable.

REFERENCES

1. Rabeeh Abla ".Net Design Patterns"https://www.codeproject.com/Articles/23065/NET-Design-Patterns(Jan 13, 2009)
2. Refactoring.Guru "What's a design pattern?" https://refactoring.guru/design-patterns/what-is-pattern (2007)
3. Jon Skeet"C# in Depth,"Manning Publications,2019
4. Sourav Kayal "Exploring Design Pattern for Dummies,"https://www.c-sharpcorner.com/ebooks/exploring-design-pattern-for-dummies(Oct 03, 2013)
5. Mahesh Alle "Singleton Design Pattern In C#"https://www.c-sharpcorner.com/UploadFile/8911c4/singleton-design-pattern-in-C-Sharp/ (Jun 2020)
6. Dot net tutorials "Singleton Design Pattern in C#" https://dotnettutorials.net/lesson/singleton-design-pattern/(2019)
7. Microsoft "Design Patterns: Singleton" https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/design-patterns-singleton (Aug 08, 2017)
8. Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra "Head First Design Patterns: A Brain-Friendly Guide", O'Reilly Media, Nov 30, 2004
9. Sungu Hasan Emrah "Singleton Pattern Implemented in C#" https://dev.to/emrahsungu/singleton-pattern-implemented-in-c-e9(Oct 14, 2019)