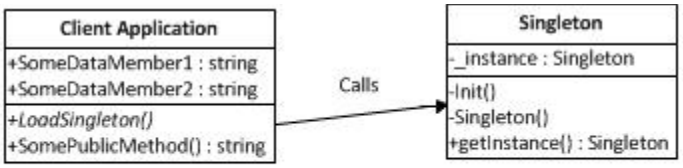
1. Singleton (Gamma et al)

Singleton is one of the simplest and easy to implement design pattern. The intent defined by [1] ensures a class only has one instance, and provide a global point of access to it. As we come across designing the framework for application there are situations where we enforce a constraint to have exactly one instance of a certain class throughout the application lifetime. It fits in a senario where there is a single resource shared among different clients or objects. By a global access point it is meant that a particu lar instance of the class can be accessed anywhere in the application without worrying about its instantation but that isn't very easy to achieve due to diverse implementation scenarios. Addtionally Singleton should be a self owner. The client application or where the object is used is not required to perform additional steps in the creation, configuration or destruction of the object. Programmatically a singleton can defined as a class with private constructor that prohibits its creation with the "new" language operator. The instance will be accessed with static properties or methods to get the preconfigured object. Though there are some techniques which can break singleton such as "Reflection" in modern languages to access the private information from the .Net Assembly metadata [4] , [5] but reflection is usually discouraged and alternate ways can be implemented to tighten the security through appropriate exceptional handling. Other than that Singleton is mostly used in combination with other design patterns e.g. Builder, Prototype, Factory method and Abstract Factory which makes its recognition questionable and important at the same time for the design pattern detection tools to give accurate results as discussed in D^3 (D-Cubed) [6]. Thus we are documenting some of the important variants of singelton shown below. Please note that these variants are not the only available and yet they can be combined to form new variants

1.1. Eager Instantiation [6]

It refers to the technique in which instances of objects are created before any block of code particularly asks for them [7]. It benefits runtime performance particularly in the case when shared objects are preloaded into memory awating to be called. This singleton variant involves the creation of instance in an initialization block which is precalled when a class is loaded. This variant is beneficial in thread safe scenarios but though it derives a demerit when there are limited number of resoucers e.g. database connections. Following is the UML diagram

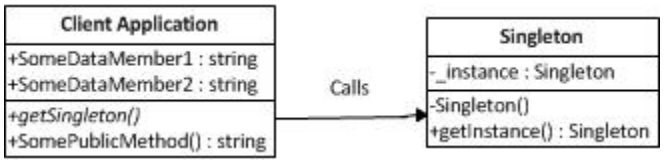



```
Type mySingleton = testAssembly.GetType("Patterns.Singleton");
mySingleton .InvokeMember("Init",   BindingFlags.InvokeMethod | BindingFlags.Instance | BindingFlags.NonPublic,   null, mySingleton , null);
```

1.1 UML Diagram and source code of Eager Instantiation

1.2. Lazy Instantiation (non – thread safe) [6] [8]

It is a memory conservation technique by which the program delays the wanted object creation until they are explicitly required for use. This technique is of two types i.e lazy class loading and lazy object loading. The first one refers to a class when first referenced with a 'static' method or with 'new' operator where as the later one provides tight coupling and delays the instatantiation until it is actually needed. Lazy instantiation is one of the most common and heavily used variant of singleton pattern which provides an access method to look for any existing singleton object, and create an object if not found. This variant is an extention of the previous variant i.e. eager instantiation. Because of its simple implmentatio this variant derives some drawbacks in the multi threaded enviornment when two threads concurrently access a critical region code block. Following is the UML and source code.

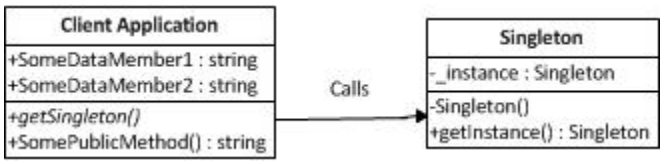



```
public static Singleton Instance
{
    get
    {
        if (instance == null)
        {
            instance = new Singleton();
        }
        return instance;
    }
}
```

1.2 UML diagram and source code of non safe thread implementation

1.3. Lazy Instantiation (thread safe) [8]

A secod version of lazy instantiation specifically focuses on the thread safe implementation of singleton. This implementation makes use of the idea of critical region in operating systems when different threads simultaneously access a certain region which is bound to a shared resource that can be accessed one at a time. Similarly using this implementation we lock the singleton instance creation region. It ensures that during a certain time only one thread will be executing the block.

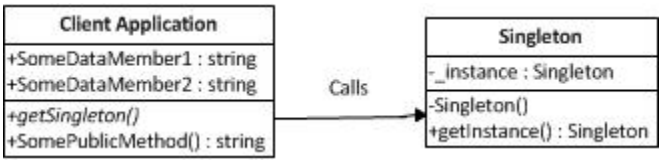



```
public static Singleton Instance{
    get{
        lock (padlock){
            if (instance == null){
                instance = new Singleton();
            }
        }
        return instance;
    }
}
```

1.3 UML Diagram and source code of thread safe implementation

1.4. Lazy Instantiation with double lock mechanism [8] [9]

Another best thread safe implementation of singleton considered a good approach to provide indepth handling of instance creation. At the same time it is discouraged as well because it can yeild performance issues due to mutual-exclusion lock on a certain object [10]. Double locking mechanism is generally avoided to 'lock' types(classes) which can cause threading issues [10].

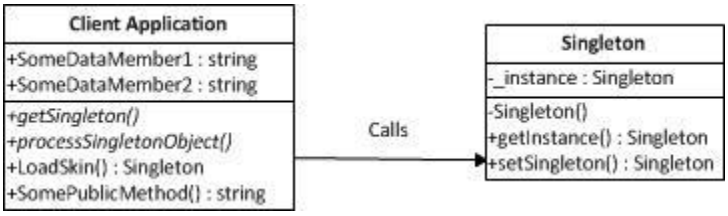



```
public static Singleton Instance
{
    get
    {
        if (instance == null)
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

1.4 UML Diagram and source code of double locking mechanism

1.5. Replacable Instance [6]

There are situations when a singleton instance is replaced at runtime. The best way to symbolize this is changing of the look and feel (GUI) of the application as discussed by [6]. Such type of execution time changes require the singleton object to be replaced with new configurations. The instance is still the one but with different configuration. So a setter method is provided which will take care of necessary replacement.

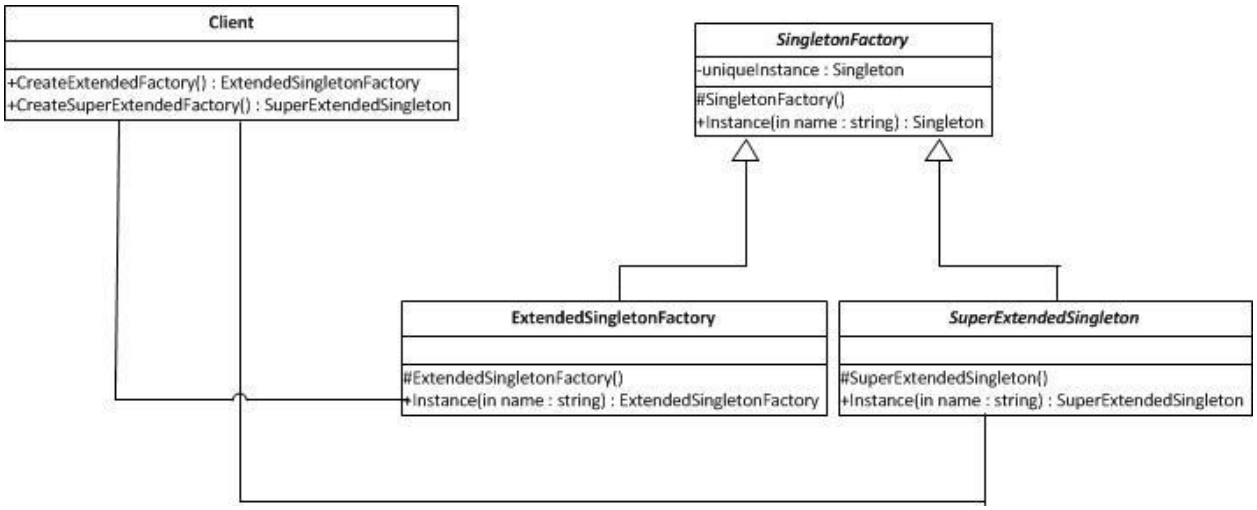



```
public static Singleton setSingleton()
{
    Singleton Instance = Singleton.getInstance();
    Instance = Instance.LoadSkinObject("Beach");
    return Instance;
}
```

1.5 Uml Diagram and source code of Replacable Instance

1.6. SubClassed Singleton [6] [11]

Singleton classes are usually considered non extensable due to the fact that they have private constructor. So implementing it to be subclassed by another class involves some extra management to handle the ensurity of sole instance. Some time it may be the case to extend the default behaviour of the singleton class e.g platerform dependent configurations that are usually diverse in nature [6]. In this implementation the private constructor is made protected so that classes should be derived from it but there is an issue with the static self instance which will be shared among all the derived classes so in order to make the sole instance work the singleton registry manager holds the oppurtunity to keep the class instances and returned when needed [12]. Subclassed singleton is also dissused by [11] in a different way.



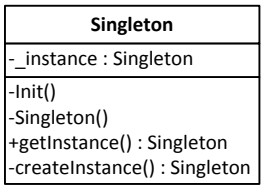

```
public abstract class MazeFactory
{
    // The private reference to the one and only instance.
    private static MazeFactory uniqueInstance = null;
    // The MazeFactory constructor.
    // If you have a default constructor, it can not be private
    //   here!
    protected MazeFactory() { }
    public static MazeFactory instance()
    {
        if (uniqueInstance == null) return instance("enchanced");
        else return uniqueInstance;
    }
    // Create the instance using the specified String name.
    public static MazeFactory instance(String name)
    {
       return uniqueInstance;
    }
}
```

```
public class EnchantedMazeFactory : MazeFactory
{
 // Return a reference to the single instance.
 public static MazeFactory instance()
 {
   if(uniqueInstance == null)
     uniqueInstance = new EnchantedMazeFactory();
   return uniqueInstance;
 }
 // Private subclass constructor!!
 private EnchantedMazeFactory() {}
```

1.6 UML Diagram and source code of the subclassed singleton

1.7. Delegated Construction [6]

This variant uses the delegation technique to create the sole instance of the singleton pattern. Delegation is basically passing on the duty of certain task(method call) to another object to do it on the owner's behalf [13]. The program must ensure the correct instance to be assigned during delegated calls. The usuage of delegation in singleton class or its access method occur when singleton is implemented with other patterns e.g Factory pattern for paramterize construction of singleton [6].

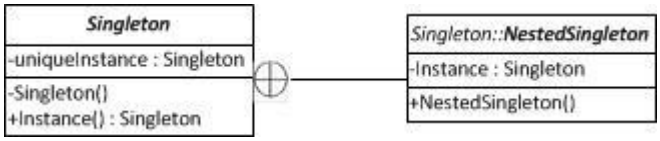



```
public class Singleton
{
    private static Singleton _instance = null;
    private Singleton(){ }
    public static Singleton getInstance()
    {
        return createInstance();
    }
    private static Singleton createInstance()
    {
        if (_instance == null)
            _instance = new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

1.7 UML Diagram and source code for delegated construction of singleton in other method.

1.8. Different Placeholder [6]

This variant discusses a nested class inside singeton class that holds the responsibility of the instance creation. The nested class acts a place holder for the singleton. This method benefits the developer to use language features to ensure a fully lazy initializaion in multi threaded enviornment.

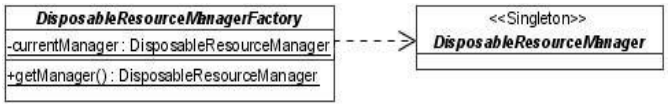



```
public sealed class Singleton
{
    public static Singleton Instance { get { return Nested.Instance; } }

    private Singleton() { }

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested() { }

        internal static readonly Singleton Instance = new Singleton();
    }
}
```

1.8 UML Diagram and source code of singleton in a nested class

1.9. Different Access Point [6]

This variant focuses on scenario where singleton instance is created by different classes and singleton class is an access point(static method) to access that instance. This variant can be used in abstract factory implementation where the products are created and holded for future reuse.

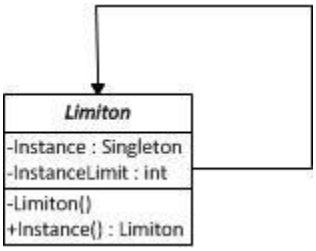



```
public class Singleton
{

    private Singleton() { }
    public static someOtherClass getSinglton()
    {
        return someOtherClass.createInstance();
    }

}
```

1.9 UML Diagram and source code of singleton created in different class

1.10. Limiton [6]

This variant extends the concept of a sole instance singleton to limited number of singleon instances. This variant can be better differentiated by solar system (single star system) and Sirius system (binary star system) [6].

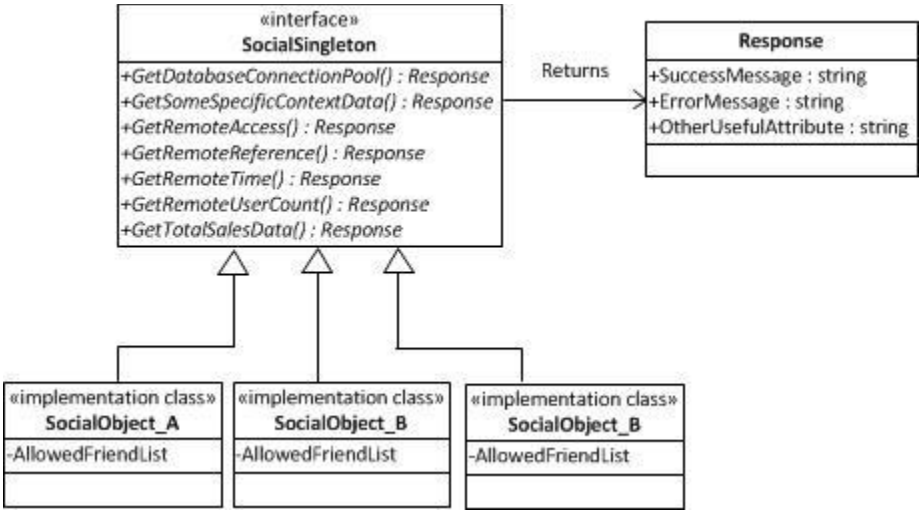



```
class Limiton
{
    private static readonly int limitonUpperLimit = 2;
    private static int limitonCurrentCount = 0;
    private static ArrayList registry = new ArrayList(limitonUpperLimit);

    private Limiton() { }
    public static Limiton Instance()
    {
        Limiton _Instance = null;
        if (limitonCurrentCount < limitonUpperLimit)
        {
            _Instance = new Limiton();
            registry.Add(_Instance);
            limitonCurrentCount = limitonCurrentCount + 1;

        }
        else
        {
            throw new Exception("Limiton Instances Overflow, cannot create more than limit defined");
        }
        return _Instance;
    }
    public static int getTotalInstanceCount()
    {
        return limitonCurrentCount;
    }

}
```

1.10 UML Diagram and source code of Limiton an extention of singelton

1.11. Social Singleton [14]

In this variant the idea of social network is used to implement singleton classes . The benefit of using social singleton is that static resources are shared among singleton objects that have a relationship (friends) with each other. One object can have access to a resource owned by any other only if it is an authorized friend object. It can be better related with a scenario when there are similar resources scatered across different locations and can be served a centerlized source of access. Examples include an integrated enviornment with multiple processes/ machines to execute a job [14]; if an object's own request fails to entertain the exclusive access, the request is forwarded to the fellow object to gurentee the access of its resource inorder to process a job. Transparency of the objects is clearly seen because the client object does n't know which object resource is shared.

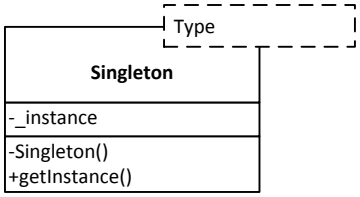



```
class SocialObjectA : SocialObjectInterface
{
    private ArrayList AllowedFriendList = new ArrayList();
    private static SocialObjectA soleInstance;
    string connectionstring;

    private SocialObjectA()
    {
        Console.WriteLine("PopulateFriendList...");
        AllowedFriendList.Add(typeof(SocialObjectB));
        AllowedFriendList.Add(typeof(SocialObjectC));
    }
    public APIResponse GetDataBaseConnectionPool()
    {
        APIResponse response = new APIResponse();
        System.Data.SqlClient.SqlConnection sqlconn = new System.Data.SqlClient.SqlConnection(connectionstring);
        try
        {
            sqlconn.Open();
        }
        catch(Exception ex)
        {
            SocialObjectB friend = (SocialObjectB) AllowedFriendList[GetAvailableFriend()];
            response = friend.GetRemoteAccess();
        }
        return response;
    }
}
```

1.11 UML Diagram and source code of social singleton object

1.12. Generic Singleton using Reflection [15]

Reflection technique is one of the powerful features provided by any modern programming languages. With the extensible support of using code that is not available at runtime which includes information that is in the metadata of .Net Assemblies i.e. types, constructors, private methods etc. as described by [4]. Using .Net generics the common design patterns are generalized in such a way that it can be used within any type. .Net generic implementation of singleton is a nice attempt to use any type as a singleton. Though there is no change in structure but it provides an easy way to make classes as singleton without writing additional code.




```
public class Singleton<T> where T : class, new()
{
    Singleton() { }

    class SingletonCreator
    {
        static SingletonCreator() { }
        // Private object instantiated with private constructor
        internal static readonly T instance = new T();
    }

    public static T UniqueInstance
    {
        get { return SingletonCreator.instance; }
    }
}
```

1.12 UML and Source Code of Generic Singleton Class