3.1 **SINGLETON**

**How would you protect this pattern in multi-threaded environment?**

public sealed class Singleton

{

    private static readonly Lazy<Singleton> lazy =

        new Lazy<Singleton> (() => new Singleton ());

    public static Singleton Instance {get {return lazy.Value;} }

    private Singleton ()

    {

    }

}

It also allows you to check whether or not the instance has been created yet with the [IsValueCreated](https://nam02.safelinks.protection.outlook.com/?url=http%3A%2F%2Fmsdn.microsoft.com%2Fen-us%2Flibrary%2Fdd642334.aspx&data=02%7C01%7C%7Cdbc9b08a73bb41185d9d08d59070383e%7C84df9e7fe9f640afb435aaaaaaaaaaaa%7C1%7C0%7C636573734011473615&sdata=ELE35oJnnvDd8LhRa4J00kSe2rz1AkLwbwVXTFsklCY%3D&reserved=0" \t "_blank) property, if you need that.

3.2 **COMMAND**

**Describe a scenario this pattern is useful in?**

**Transactions**: In a transactional behaviour code there are multiple tasks/updates. When all the tasks are done then only transaction is committed. Else we have to rollback the transaction. In such a scenario each step is implemented as separate Command.

3.3 **FAÇADE**

**Briefly explain how this pattern can help designing an API.**

**API façade pattern**. This pattern gives you a buffer or virtual layer between the interface on top and the API implementation on the bottom. You essentially create a façade – a comprehensive view of what the API should be and importantly it is the view from the perspective of the app developer and end user of the apps they create.

4 **Concepts**

4.1 **Separation of Concerns**

Concerns are the different aspects of software functionality. For instance, the "business logic" of software is a concern, and the interface through which a person uses this logic is another.

The separation of concerns is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa.

Model-View-Controller (MVC) design pattern is an excellent example of separating these concerns for better software maintainability.

4.2 **Inversion of Control**

**inversion** of **control** (IoC) is a programming technique, expressed here in terms of object-oriented programming, in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis.

4.3 **Dependency Injection**

Dependency Injection is a technique of removing internal dependencies from implementations by allowing dependent objects to be injected into the class/method by an external caller. IoC frameworks use dependency injection to supply user modules and other dependent code to framework routines that "glue it all together." Dependency injection is used heavily by IoC frameworks because that is the mechanism that allows them to "Call You."

4.4 **Single Responsibility Principle**

A class should have one and only one reason to change, meaning that a class should have only one job.

6.1 **What is Wrong here? How would you fix it?**

Interface can’t have Protected access modifier. Make it Public instead of Protected.

6.2 **And Here?**

An **argument** that is passed using a **ref** keyword must be initialized in the calling method before it is passed to the called method.