SOLID Design Principles

1. **Single Responsibility Principle**
   A class should have only a single responsibility.

2. **Open Closed Principles**
   A software module/class or method should be open for **extension** and closed for **modification**

3. **Liskov Substitution Principles**

4. **Interface Segregation Principles**
   Clients should not be forced to depend upon the interfaces that they do not use

5. **Dependency Inversion Principles**

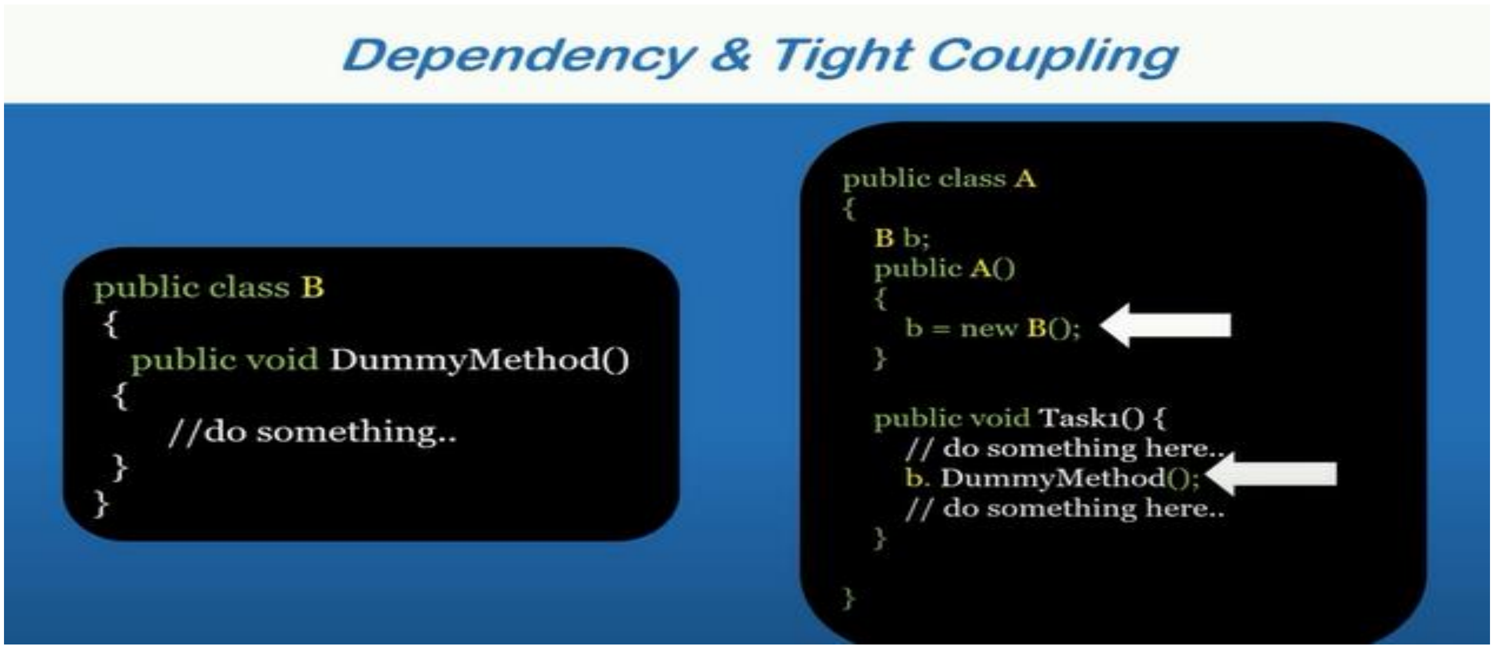
Comparison of

- **Inversion of control (IOC)**
- **Dependency Inversion Principle (DIP) and**
- **Dependency Injection (DI)/ IOC Container**

Question:

1. **How Dependency Injection help in Unit Testing**
2. **How web.config can be your best example when explaining IOC**
3. **Why should we use DI when we already have DIP**

Dependency & Tight Coupling:



Dependency & Tight Coupling

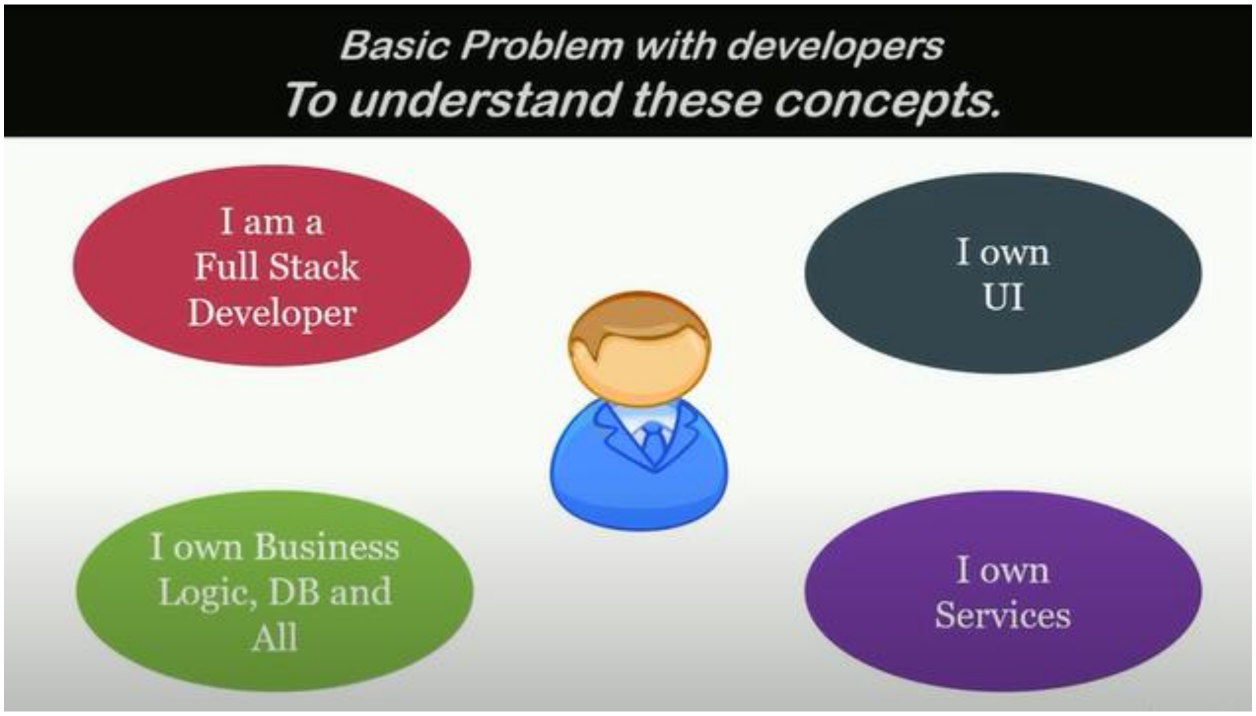
```
public class B
 {
   public void DummyMethod()
  {
      //do something..
  }
}
```

```
public class A
{
   B b;
   public A()
   {
      b = new B();
   }

   public void Task1() {
      // do something here..
      b. DummyMethod();
      // do something here..
   }

}
```

1. Here A-Class is dependent on B-Class for **Dummy Method**
2. B-Class is declared tightly inside the constructor

Basic Problem with Developers to understand these problems

1. I am a Full Stack developer
2. I own UI, Services, Business Logic, DB and all

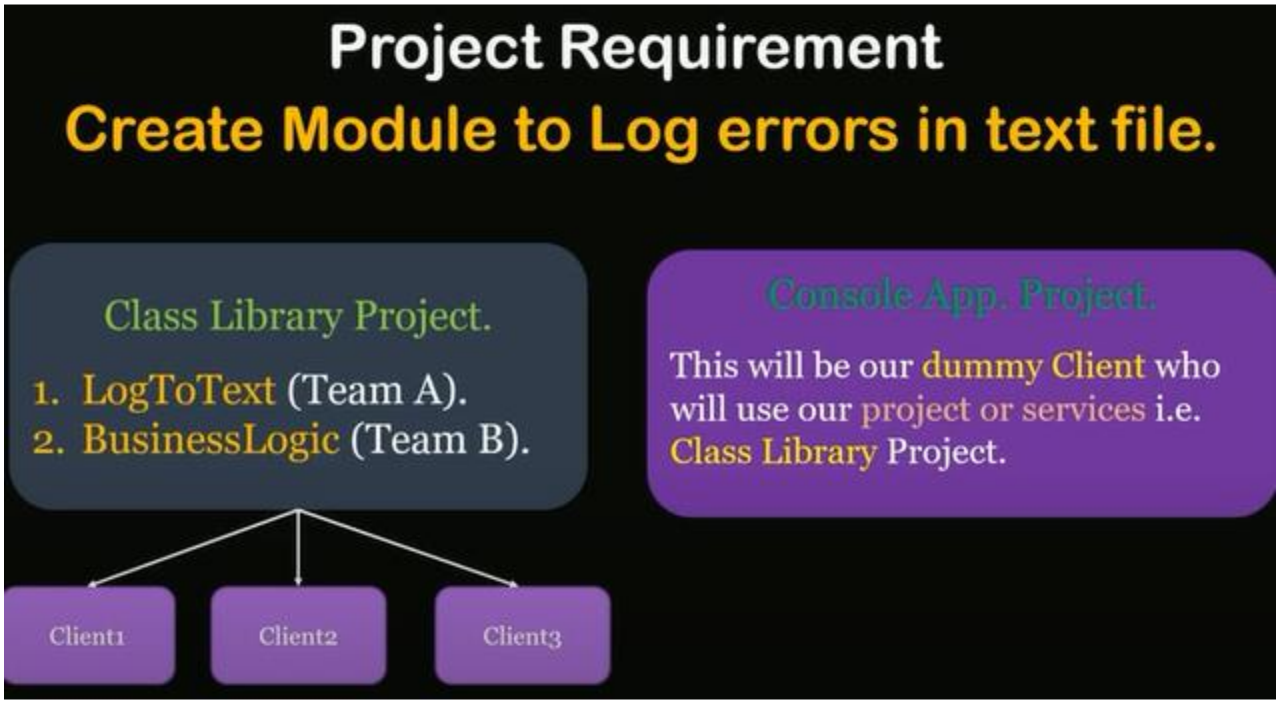


Basic Problem with developers
To understand these concepts.

Solution

Assumption: You do not own even a single class in your project

Project Requirement: Create a module to Log errors in a text file



Project Requirement

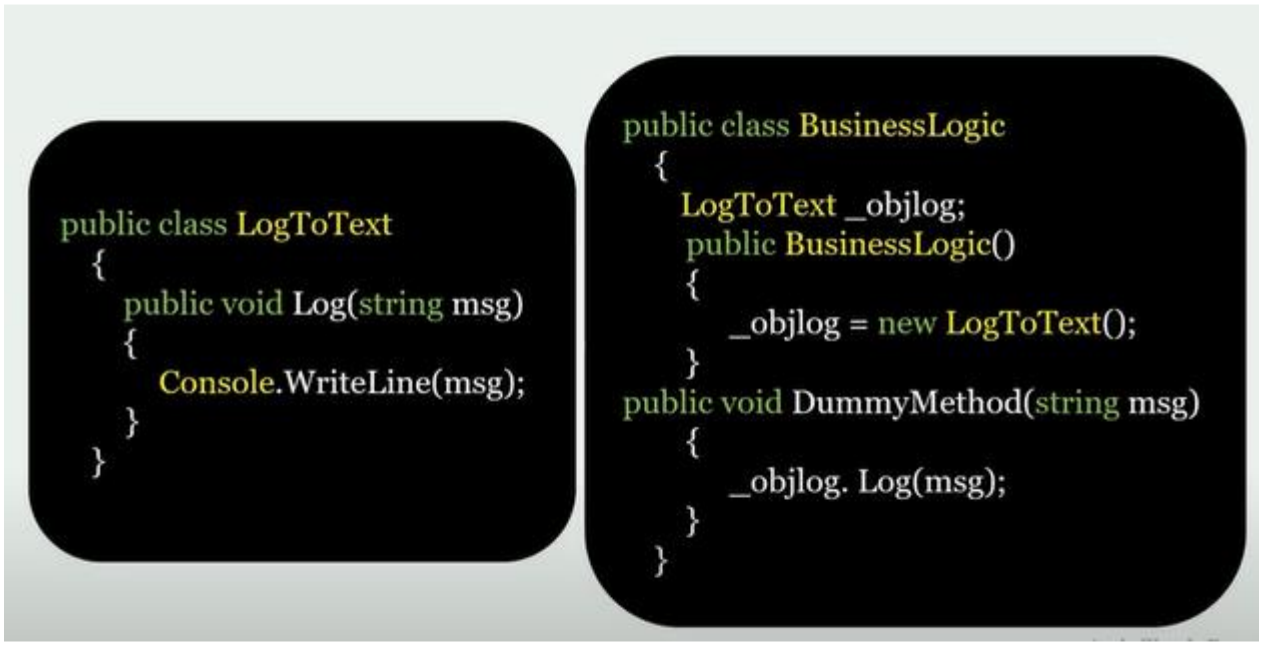
Create Module to Log errors in text file.

Class Library Project.

1. LogToText (Team A).
2. BusinessLogic (Team B).

Console App. Project.

This will be our dummy Client who will use our project or services i.e. Class Library Project.



```
public class LogToText
  {
     public void Log(string msg)
     {
        Console.WriteLine(msg);
     }
  }
```

```
public class BusinessLogic
  {
    LogToText _objlog;
     public BusinessLogic()
     {
        _objlog = new LogToText();
     }
public void DummyMethod(string msg)
     {
        _objlog. Log(msg);
     }
  }
```

Note:

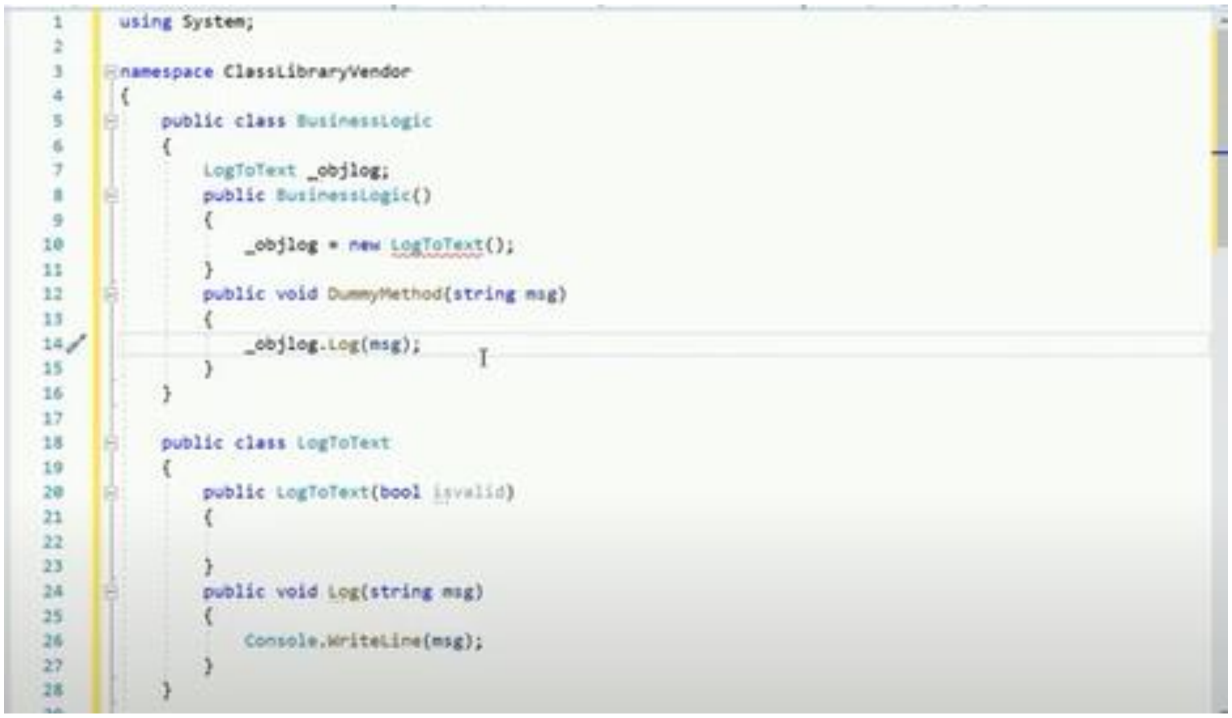
1. Business Logic is dependent on **LogToText** Class
2. Business logic is tightly coupled with **LogToText** Class

Question: Is there any problem with this implementation?

Answer: In general Not a big problem for a small project But Problems Occurs in Big Projects

Problems:

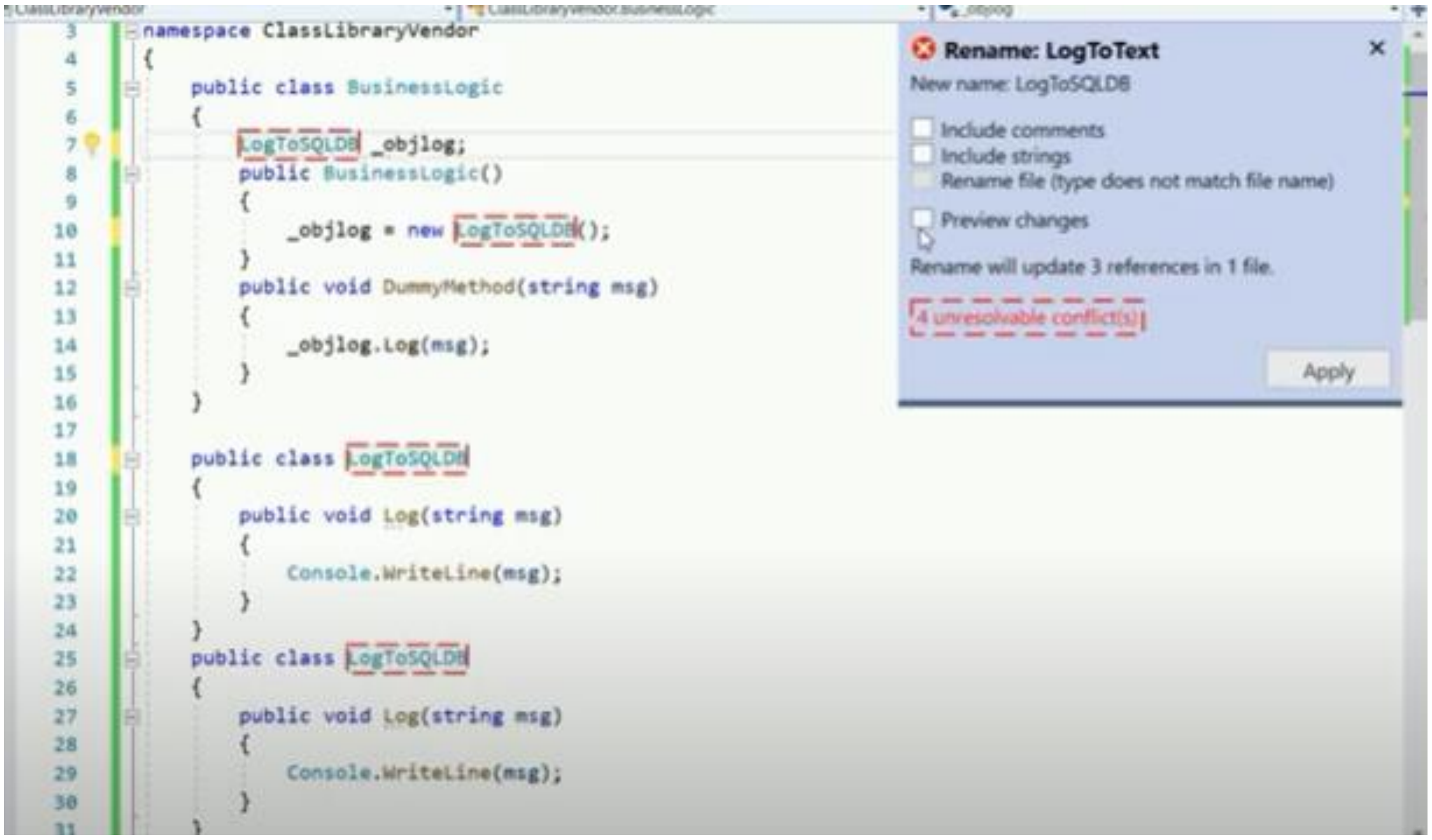
1. In case of change on dependent class **LogToText**, we need to make changes in all the places where we are referencing this class. Here we add a parameterized constructor.
   In a small project, we can solve the error easily. But big projects we do not have overall control to modify the errors.



```
using System;

namespace ClassLibraryVendor
{
    public class BusinessLogic
    {
        LogToText _objlog;
        public BusinessLogic()
        {
            _objlog = new LogToText();
        }
        public void DummyMethod(string msg)
        {
            _objlog.Log(msg);
        }
    }

    public class LogToText
    {
        public LogToText(bool isvalid)
        {

        }
        public void Log(string msg)
        {
            Console.WriteLine(msg);
        }
    }
```

2. In case of future updates **LogToDatabase** we need to make modifications in all our dependent classes.

Note: we can rename the name using visual studio. If you think this way, you are still on Full Stack mode.
You think you do not have control over **others** Classes or Project.
This Dll also can use other applications which is not access from your company.



```
namespace ClassLibraryVendor
{
    public class BusinessLogic
    {
        LogToSQLDB _objlog;
        public BusinessLogic()
        {
            _objlog = new LogToSQLDB();
        }
        public void DummyMethod(string msg)
        {
            _objlog.Log(msg);
        }
    }

    public class LogToSQLDB
    {
        public void Log(string msg)
        {
            Console.WriteLine(msg);
        }
    }
    public class LogToSQLDB
    {
        public void Log(string msg)
        {
            Console.WriteLine(msg);
        }
    }
```

Rename: LogToText
New name: LogToSQLDB
Include comments
Include strings
Rename file (type does not match file name)
Preview changes
Rename will update 3 references in 1 file.
4 unresolvable conflict(s)

So let’s see how we can overcome those problems

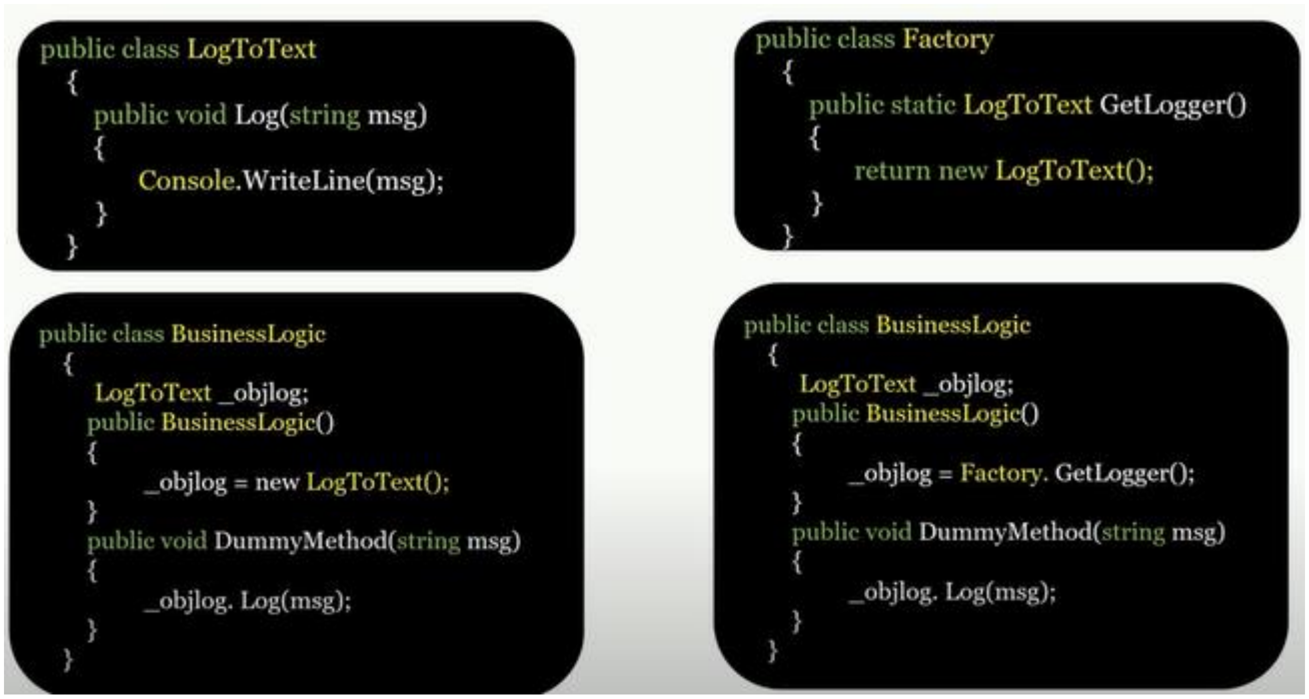
Inversion of control (IOC):

IOC is a design principle used to convert different types of controls in object-oriented design to achieve loose coupling

The main objective of IOC is to remove dependencies between the objects of an application which makes the application more decoupled and maintainable

IOC is a principle, which means it is a general guideline and end-user can choose the way how they want to implement it.

Example 1:



```
public class LogToText
  {
    public void Log(string msg)
    {
        Console.WriteLine(msg);
    }
  }
```

```
public class Factory
  {
    public static LogToText GetLogger()
    {
        return new LogToText();
    }
  }
```

```
public class BusinessLogic
  {
    LogToText _objlog;
   public BusinessLogic()
   {
        _objlog = new LogToText();
   }
   public void DummyMethod(string msg)
   {
        _objlog. Log(msg);
   }
  }
```

```
public class BusinessLogic
  {
    LogToText _objlog;
   public BusinessLogic()
   {
        _objlog = Factory. GetLogger();
   }
   public void DummyMethod(string msg)
   {
        _objlog. Log(msg);
   }
  }
```

The above examples solve problem No 1: (In case of change on dependent class **LogToText**, we need to make changes in all the places)

We cannot solve all the problems. It is just an example of how IOC works (invert the control).

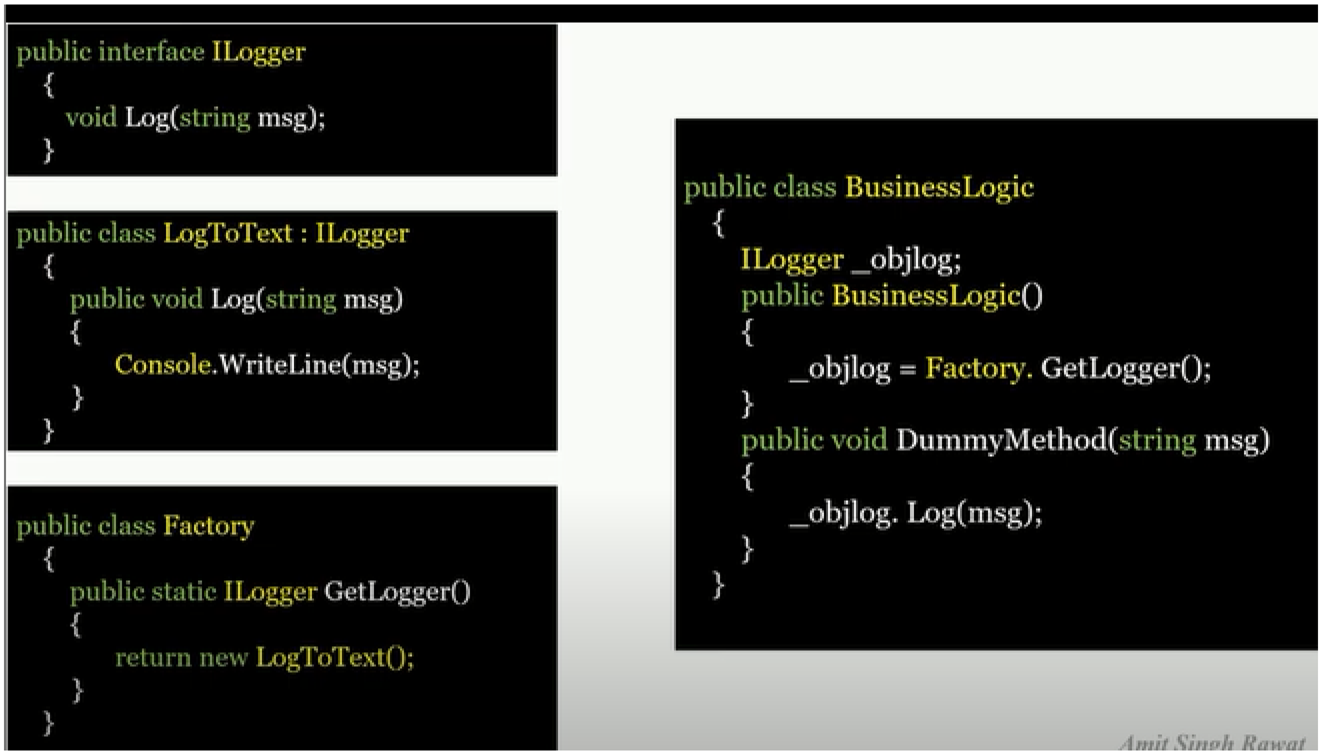
Example 2: connection string

With IOC, the general idea is to give responsibility to someone who can handle it better than the current implementation. When we create a class library, the main purpose is either reusability or separation of concern. If I hardcode connection string in my class library (.dll) project, it will be tightly coupled with my DLL project. Which we certainly don't want. We want whosoever using my DLL can set their own connection string (or any other properties) as per their needs. So, If you are using DLL within your web project you can consider yourself one of the clients of that DLL. And you can define connection string in your web. config file, instead of the DLL owner hardcode connection string in their project

Dependency Inversion Principle (DIP)

DIP states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstraction.

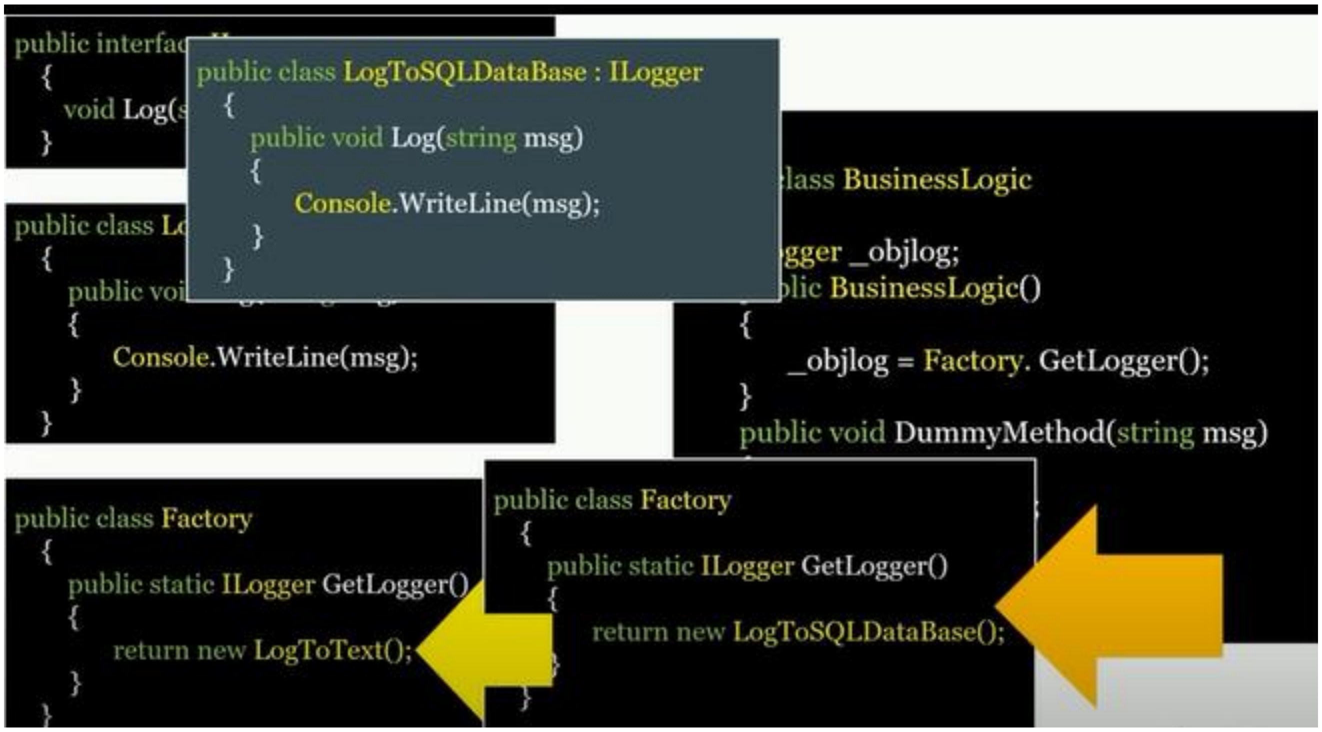
Also, abstraction should not depend upon details. Details should depend upon abstraction



```
public interface ILogger
  {
    void Log(string msg);
  }

public class LogToText : ILogger
  {
     public void Log(string msg)
     {
          Console.WriteLine(msg);
     }
  }

public class Factory
  {
     public static ILogger GetLogger()
     {
          return new LogToText();
     }
  }
```

```
public class BusinessLogic
  {
     ILogger _objlog;
     public BusinessLogic()
     {
          _objlog = Factory. GetLogger();
     }
     public void DummyMethod(string msg)
     {
          _objlog. Log(msg);
     }
  }
```

Amit Singh Rawat



```
public interfac
  {
    void Log(s
  }

public class Lo
  {
     public voi
     {
          Console.WriteLine(msg);
     }
  }

public class Factory
  {
     public static ILogger GetLogger()
     {
          return new LogToText();
     }
  }
```

```
public class LogToSQLDataBase : ILogger
  {
     public void Log(string msg)
     {
          Console.WriteLine(msg);
     }
  }
```

```
lass BusinessLogic

gger _objlog;
lic BusinessLogic()
{
     _objlog = Factory. GetLogger();
}
public void DummyMethod(string msg)
```

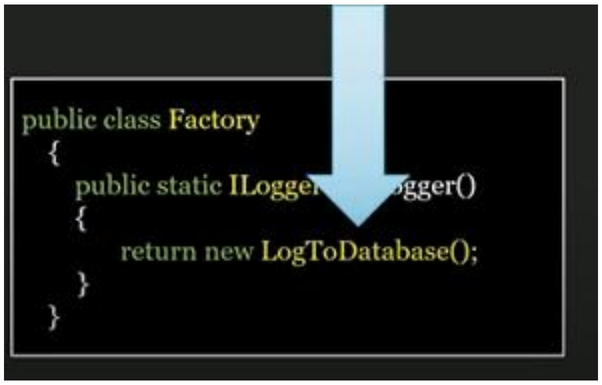
```
public class Factory
  {
     public static ILogger GetLogger()
     {
          return new LogToSQLDataBase();
     }
  }
```

The above examples solve problem No 2 (In case of future updates **LogToDatabase)**

All the problems solve. But there is also some problems

Problem 1: Control to select logger class is still with our Business Logic DLL.

So our client application do not have control to select which option they will choose. Just like connection string.



```
public class Factory
  {
     public static ILogger          gger()
     {
         return new LogToDatabase();
     }
  }
```

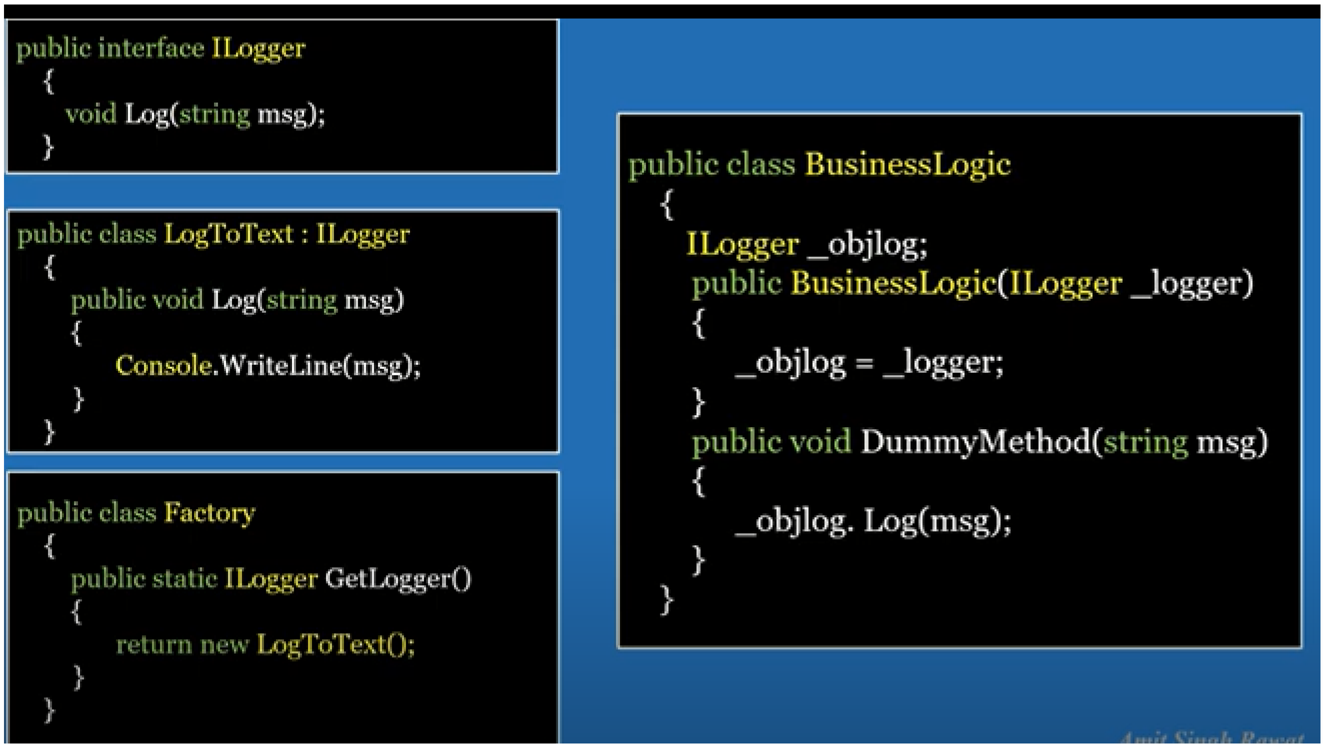
Dependency Injection:

DI is a design pattern used to implement IOC

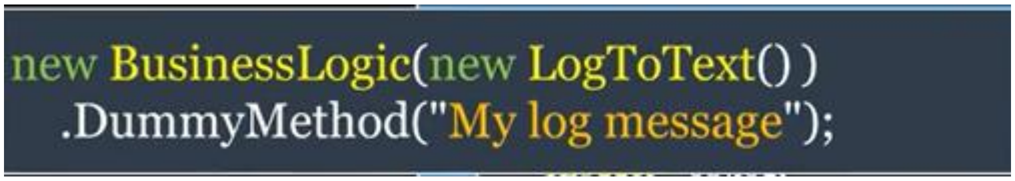
It allows the creation of dependent objects outside of a class and provides those to a class in different ways.

Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them

Types of DI: Constructor, Property, Method Injection

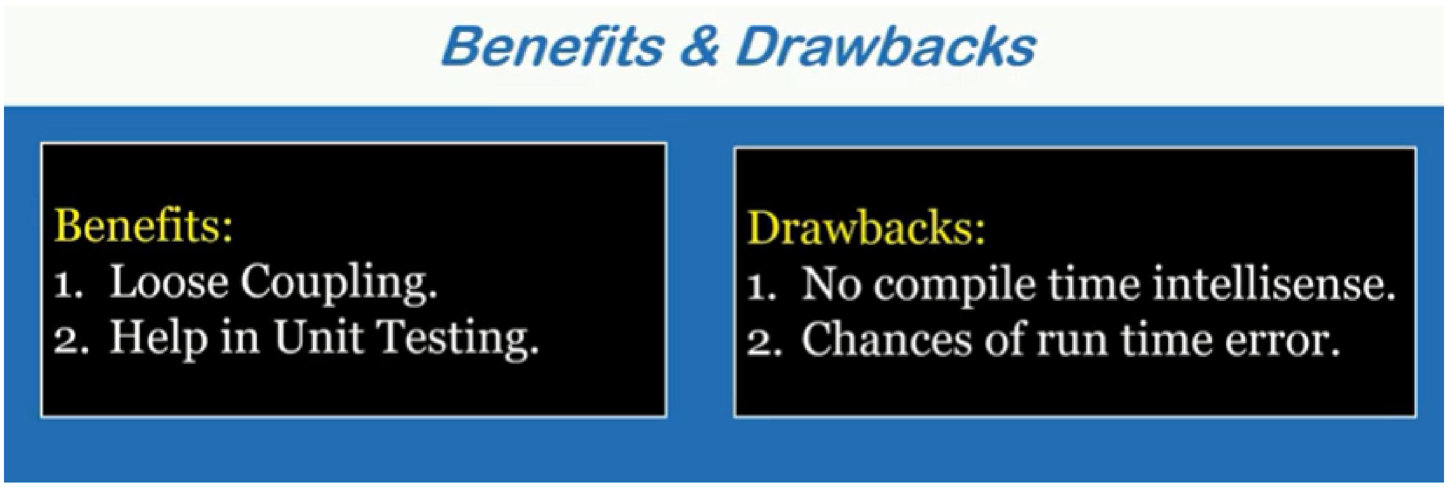


```
public interface ILogger
  {
     void Log(string msg);
  }

public class LogToText : ILogger
  {
     public void Log(string msg)
     {
         Console.WriteLine(msg);
     }
  }

public class Factory
  {
     public static ILogger GetLogger()
     {
         return new LogToText();
     }
  }

public class BusinessLogic
  {
     ILogger _objlog;
     public BusinessLogic(ILogger _logger)
     {
         _objlog = _logger;
     }
     public void DummyMethod(string msg)
     {
         _objlog. Log(msg);
     }
  }
```



```
new BusinessLogic(new LogToText() )
   .DummyMethod("My log message");
```

Call From Client Application

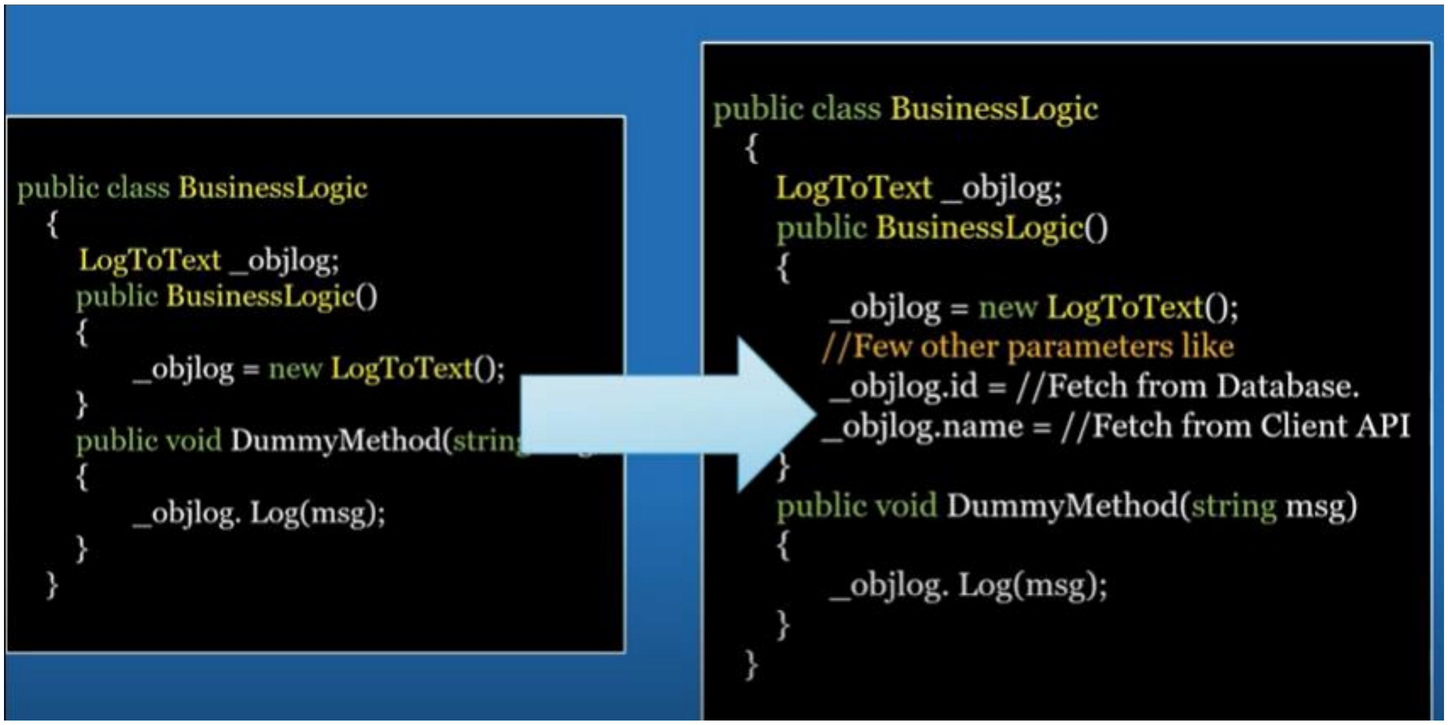


Benefits & Drawbacks

Benefits:
1. Loose Coupling.
2. Help in Unit Testing.

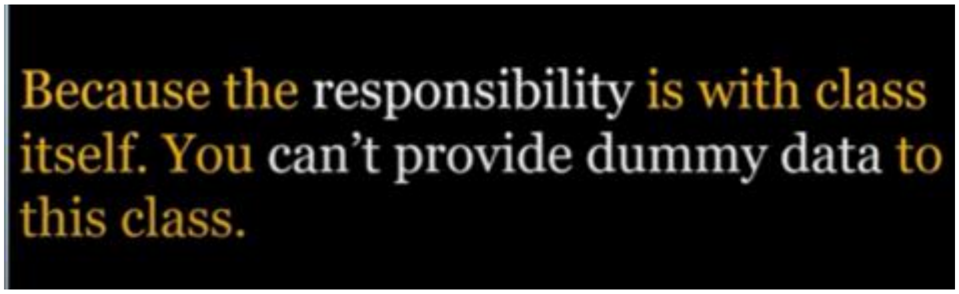
Drawbacks:
1. No compile time intellisense.
2. Chances of run time error.

Help in Unit testing But How?

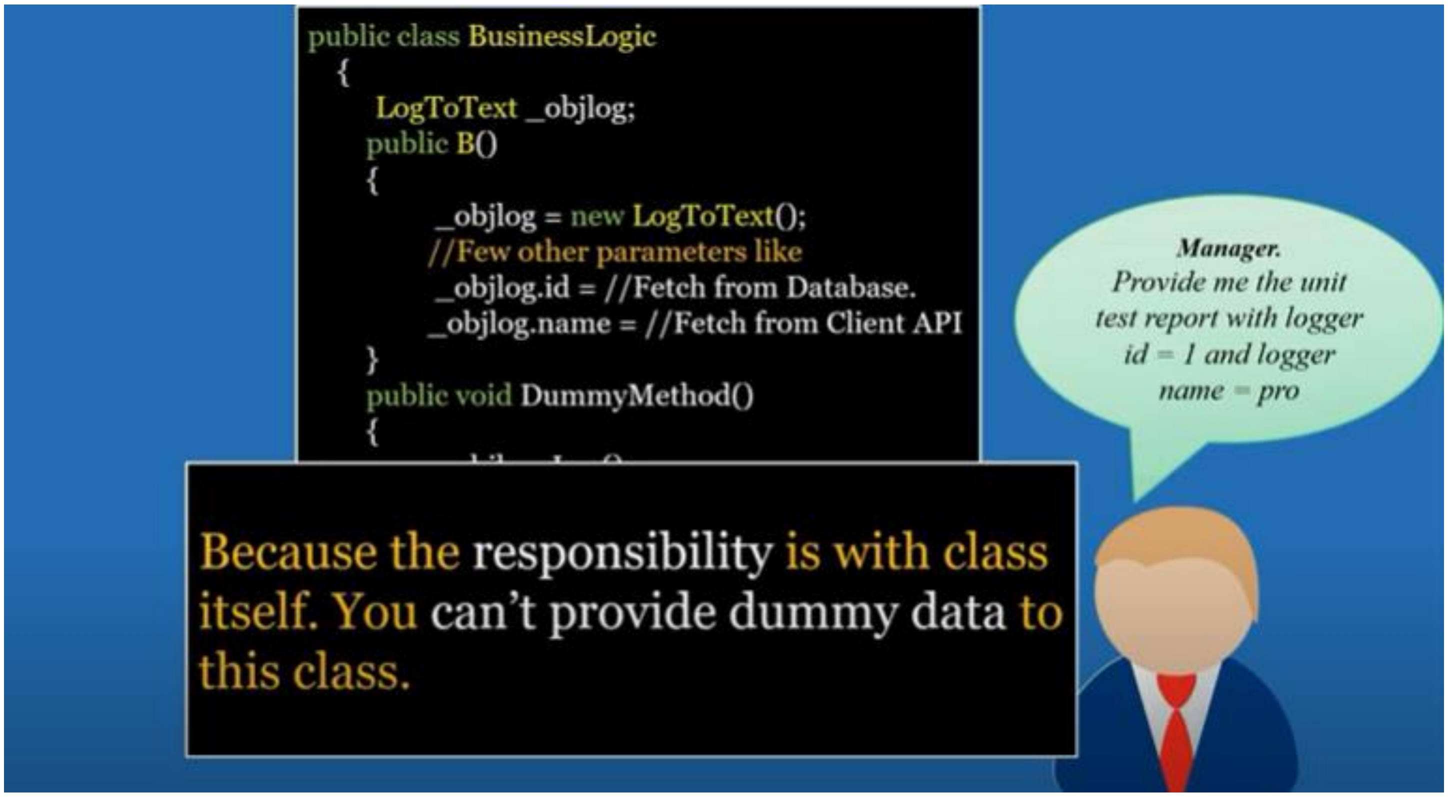


```
public class BusinessLogic
  {
     LogToText _objlog;
     public BusinessLogic()
     {
          _objlog = new LogToText();
     }
     public void DummyMethod(strin
     {
          _objlog. Log(msg);
     }
  }
```

```
public class BusinessLogic
  {
     LogToText _objlog;
     public BusinessLogic()
     {
          _objlog = new LogToText();
          //Few other parameters like
          _objlog.id = //Fetch from Database.
          _objlog.name = //Fetch from Client API
     }
     public void DummyMethod(string msg)
     {
          _objlog. Log(msg);
     }
  }
```



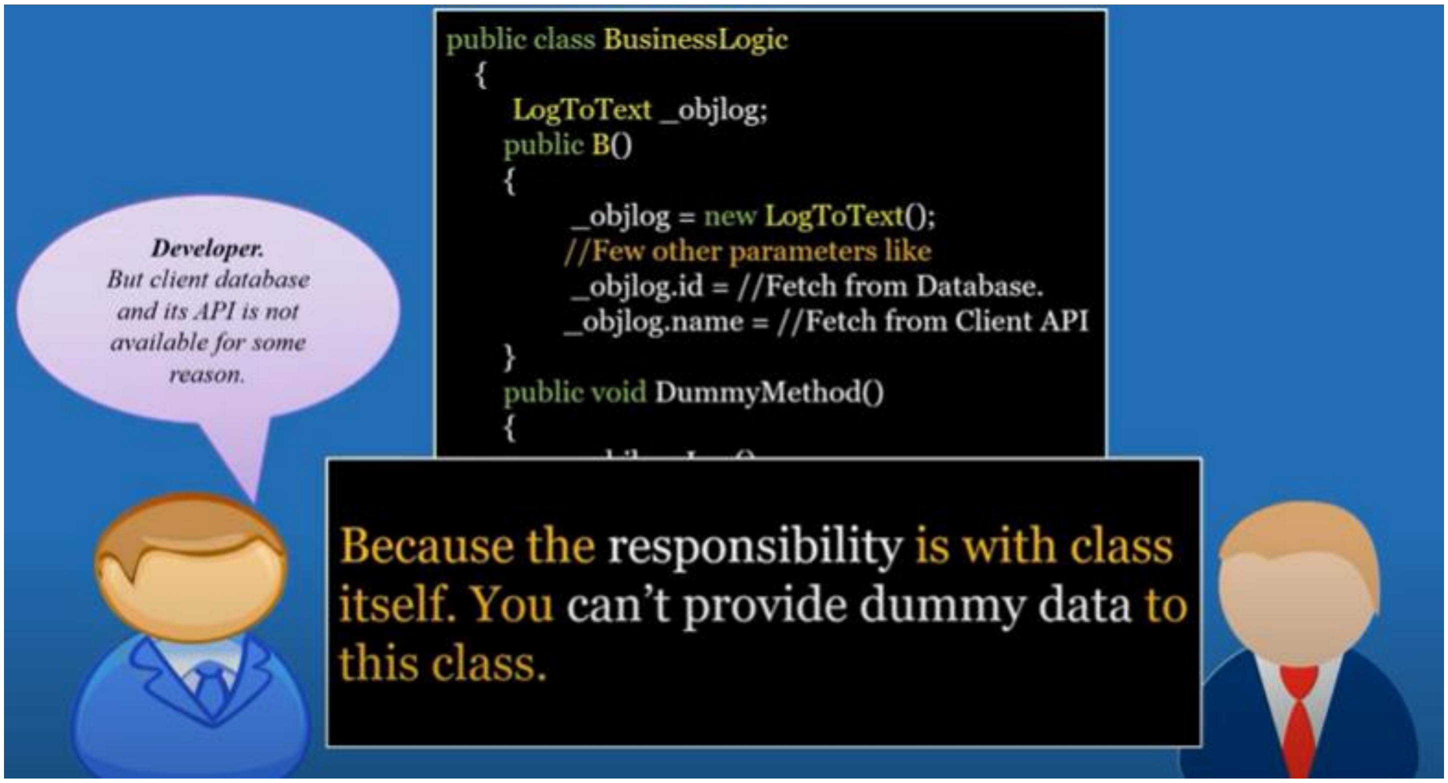
Because the responsibility is with class itself. You can't provide dummy data to this class.



```
public class BusinessLogic
  {
     LogToText _objlog;
    public B()
    {
          _objlog = new LogToText();
         //Few other parameters like
          _objlog.id = //Fetch from Database.
         _objlog.name = //Fetch from Client API
    }
    public void DummyMethod()
    {
```

Because the responsibility is with class itself. You can't provide dummy data to this class.

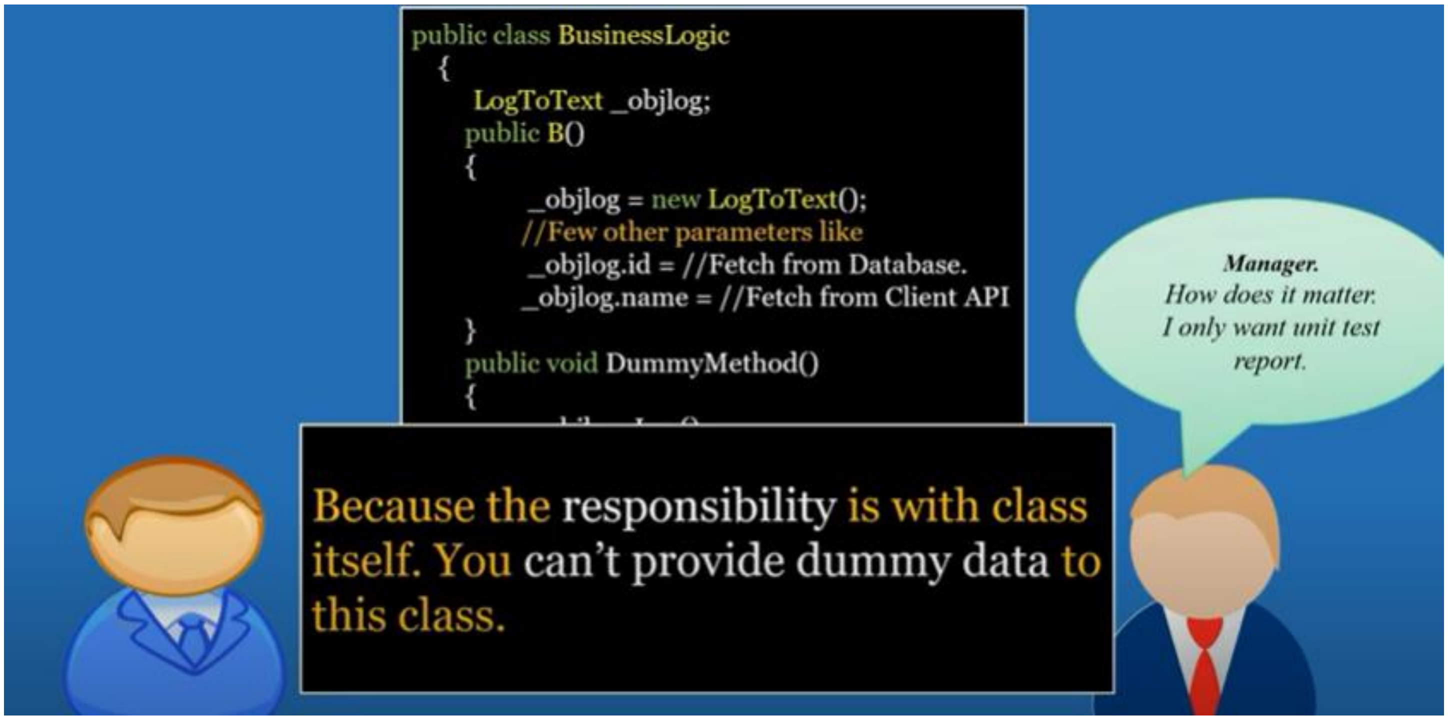
Manager. *Provide me the unit test report with logger id = 1 and logger name = pro*



```
public class BusinessLogic
  {
     LogToText _objlog;
    public B()
    {
          _objlog = new LogToText();
         //Few other parameters like
          _objlog.id = //Fetch from Database.
         _objlog.name = //Fetch from Client API
    }
    public void DummyMethod()
    {
```

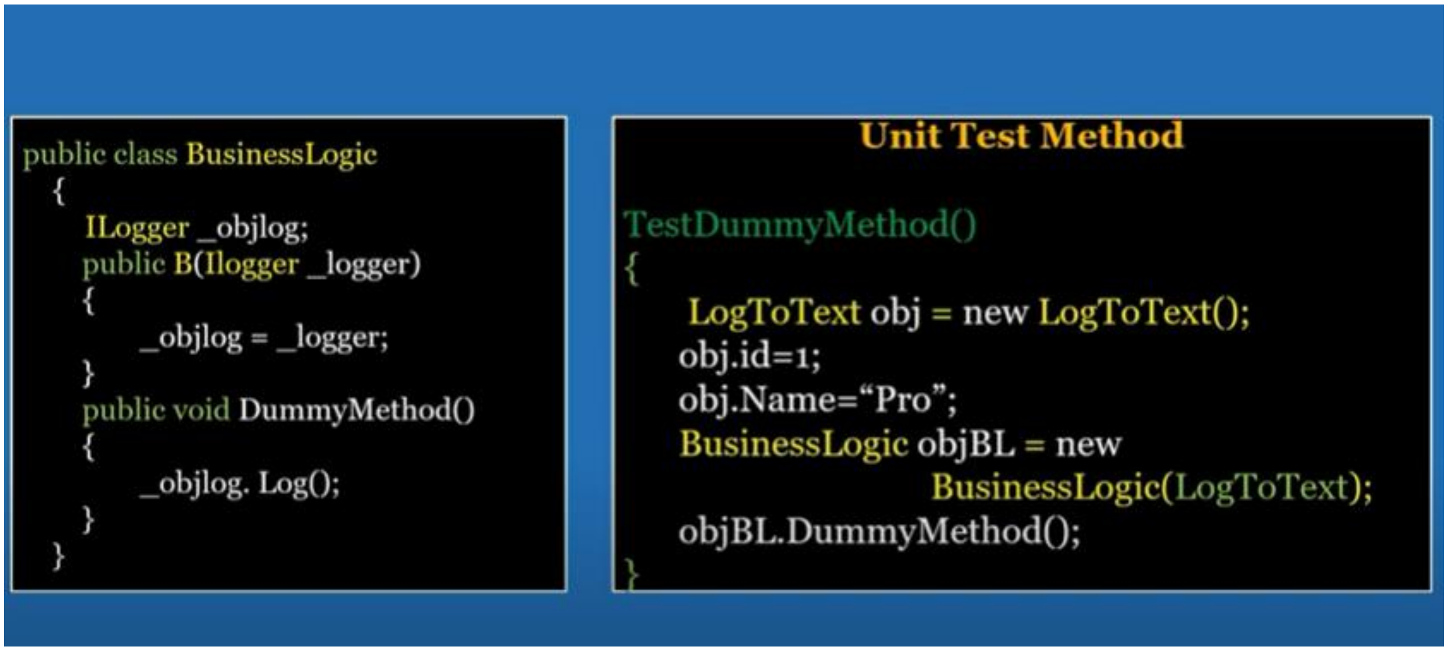
Because the responsibility is with class itself. You can't provide dummy data to this class.

Developer. *But client database and its API is not available for some reason.*



```
public class BusinessLogic
  {
     LogToText _objlog;
     public B()
     {
          _objlog = new LogToText();
          //Few other parameters like
          _objlog.id = //Fetch from Database.
          _objlog.name = //Fetch from Client API
     }
     public void DummyMethod()
     {
```

Because the responsibility is with class itself. You can't provide dummy data to this class.



```
public class BusinessLogic
  {
     ILogger _objlog;
     public B(Ilogger _logger)
     {
          _objlog = _logger;
     }
     public void DummyMethod()
     {
          _objlog. Log();
     }
  }
```

Unit Test Method

```
TestDummyMethod()
{
     LogToText obj = new LogToText();
     obj.id=1;
     obj.Name="Pro";
     BusinessLogic objBL = new
                         BusinessLogic(LogToText);
     objBL.DummyMethod();
}
```