# Architectural Design Principles

Lecture-2

## Fundamental Principles

- Seperation of Concerns
- Modularity
- Abstraction

## Separation of Concerns

***“Separation of Concerns (SoC) is a design principle that manages complexity by partitioning the software system so that each partition is responsible for a separate concern, minimizing the overlap of concerns as much as possible”***

## Separation of Concerns - Advantages

1. Lack of duplication and singularity of purpose of the individual components render the overall system easier to maintain.
2. The system becomes more stable as a byproduct of the increased maintainability.
3. The strategies required to ensure that each component only concerns itself with a single set of cohesive responsibilities often result in natural extensibility points.
4. The decoupling which results from requiring components to focus on a single purpose leads to components which are more easily reused in other systems, or different contexts within the same system.
5. The increase in maintainability and extensibility can have a major impact on the marketability and adoption rate of the system.

## Separation of Concerns

- **Horizontal Separation of Concerns**

  Horizontal Separation of Concerns refers to the process of dividing an application into logical layers of functionality - UI, Business Logic and Database.

- **Aspect Separation of Concerns**

  Aspect Separation of Concerns, better known as Aspect-Oriented Programming, refers to the process of segregating an application’s cross-cutting concerns from its core concerns

## Modularity

***“This principle states that software systems should be designed as a collection of independent modules.”***

***Or***

***“Modularity is a structural principle used to manage complexity in systems. It involves identifying functional clusters of similarity in systems, and then transforming the clusters into interdependent self-contained systems (modules).”***

## Abstraction

***“The principle of abstraction implies separating the behavior of software components from their implementation. It requires learning to look at software and software components from two points of view: what it does, and how it does it.”***

## Coupling and Cohesion (Important concepts of Software Architecture)

***"Coupling refers to the degree to which two modules are dependent on each other. High coupling can make it difficult to change or evolve the system.***

***Cohesion refers to the degree to which the elements of a module are related to each other. High cohesion makes the module easier to understand and maintain."***

## Advantages of low Coupling

- Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.
- Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

## Disadvantages of high Coupling

- Increased complexity: High coupling increases the interdependence between modules, making the system more complex and difficult to understand.
- Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

## Advantages of high Cohesion

- Improved readability and understandability: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
- Improved reliability: High cohesion leads to modules that are less prone to errors and that function more consistently,
- leading to an overall improvement in the reliability of the system.

## Disadvantages of low Cohesion

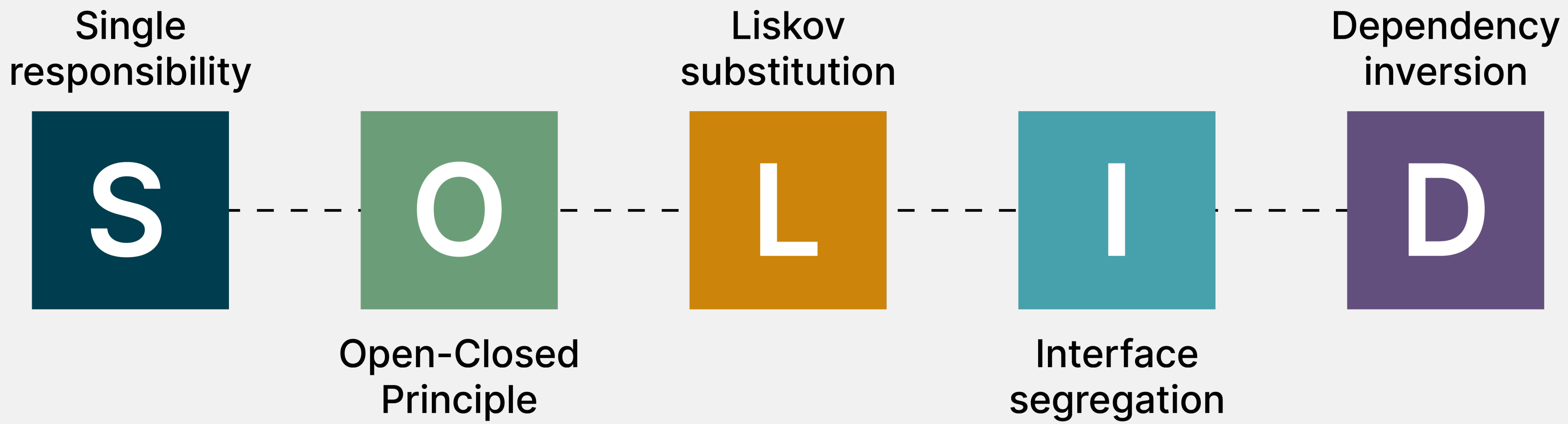
- Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
- Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.
- Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

## Scalability and Maintainability (Goals of Software Architecture)

***“Scalability refers to the ability of a system to handle increasing loads.***

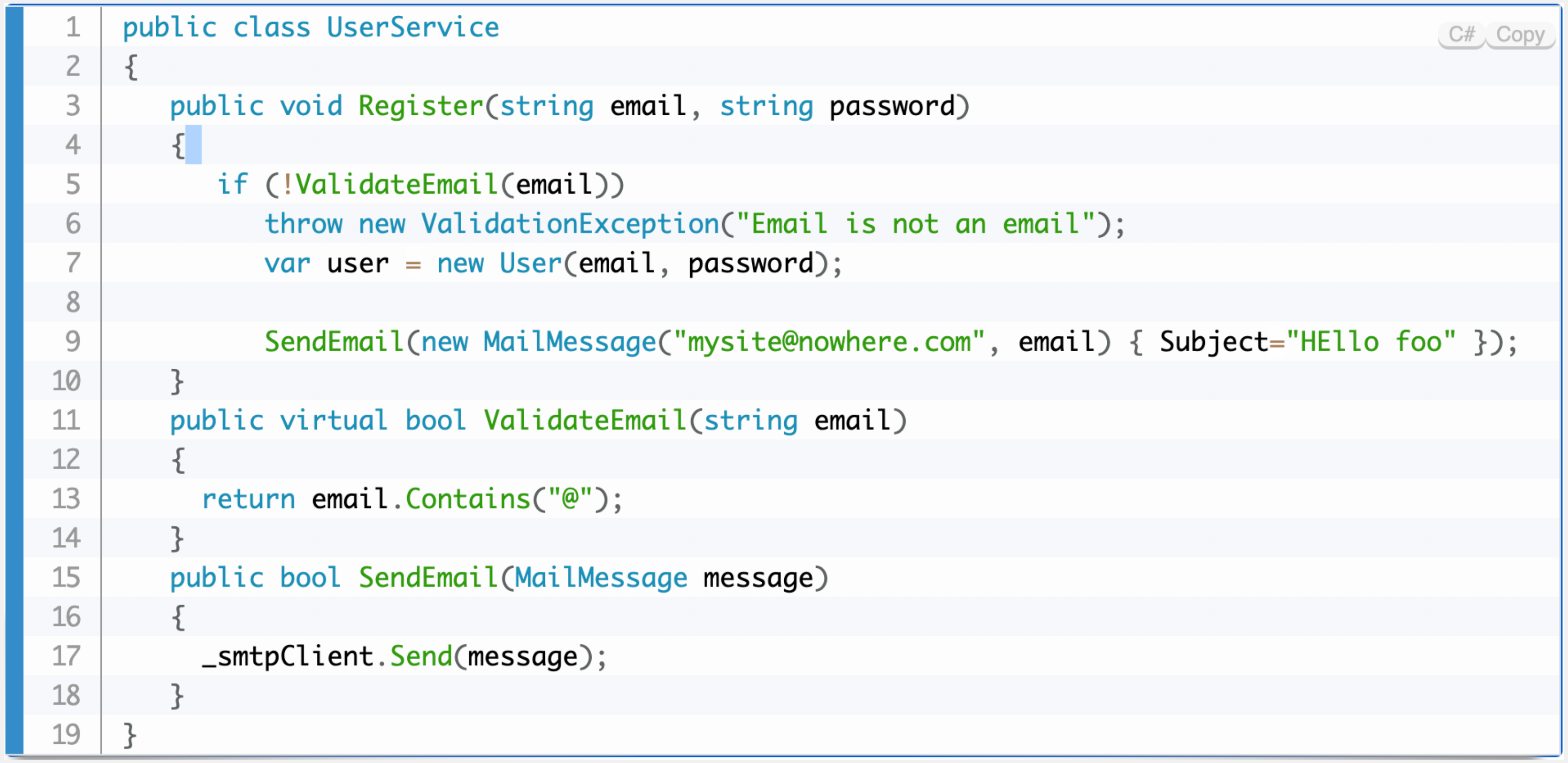
***Maintainability refers to the ability of a system to be changed or updated without causing problems.”***

## SOLID Principle



## S: Single Resposibility Principle (SRP)

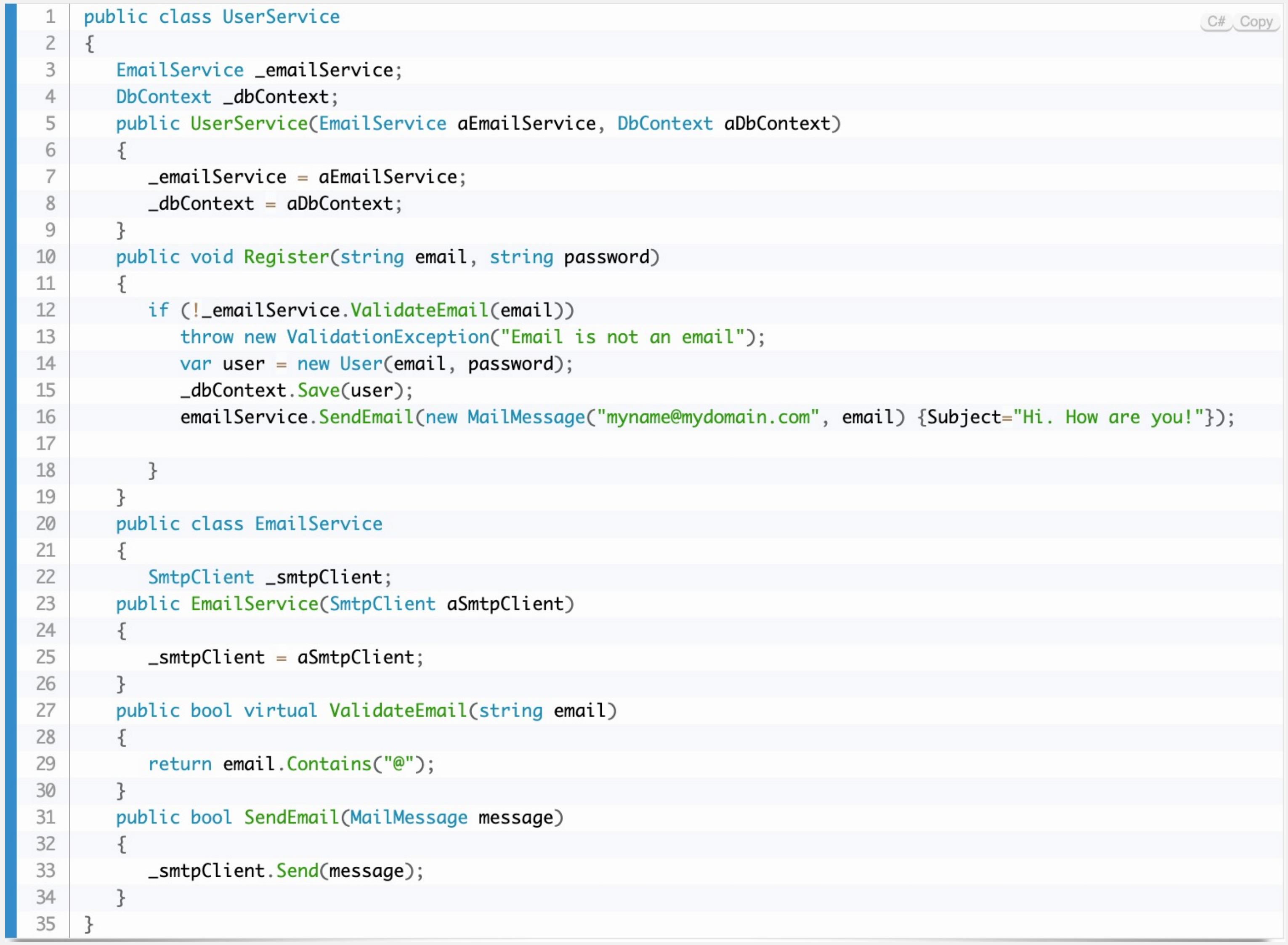
***“Every software module should have only one reason to change. Everything in that class should be related to a single purpose”***


C# Copy

```csharp
public class UserService
{
   public void Register(string email, string password)
   {
      if (!ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);

         SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
   }
   public virtual bool ValidateEmail(string email)
   {
     return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
     _smtpClient.Send(message);
   }
}
```

# S: Single Resposibility Principle (SRP)



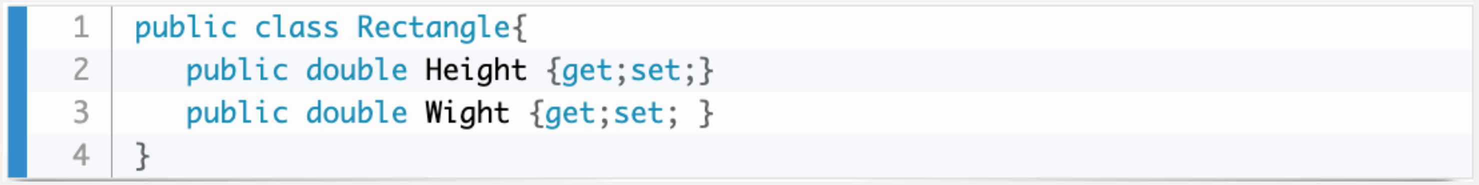
```csharp
public class UserService
{
   EmailService _emailService;
   DbContext _dbContext;
   public UserService(EmailService aEmailService, DbContext aDbContext)
   {
      _emailService = aEmailService;
      _dbContext = aDbContext;
   }
   public void Register(string email, string password)
   {
      if (!_emailService.ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);
         _dbContext.Save(user);
         emailService.SendEmail(new MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});

      }
   }
   public class EmailService
   {
      SmtpClient _smtpClient;
   public EmailService(SmtpClient aSmtpClient)
   {
      _smtpClient = aSmtpClient;
   }
   public bool virtual ValidateEmail(string email)
   {
      return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
      _smtpClient.Send(message);
   }
}
```

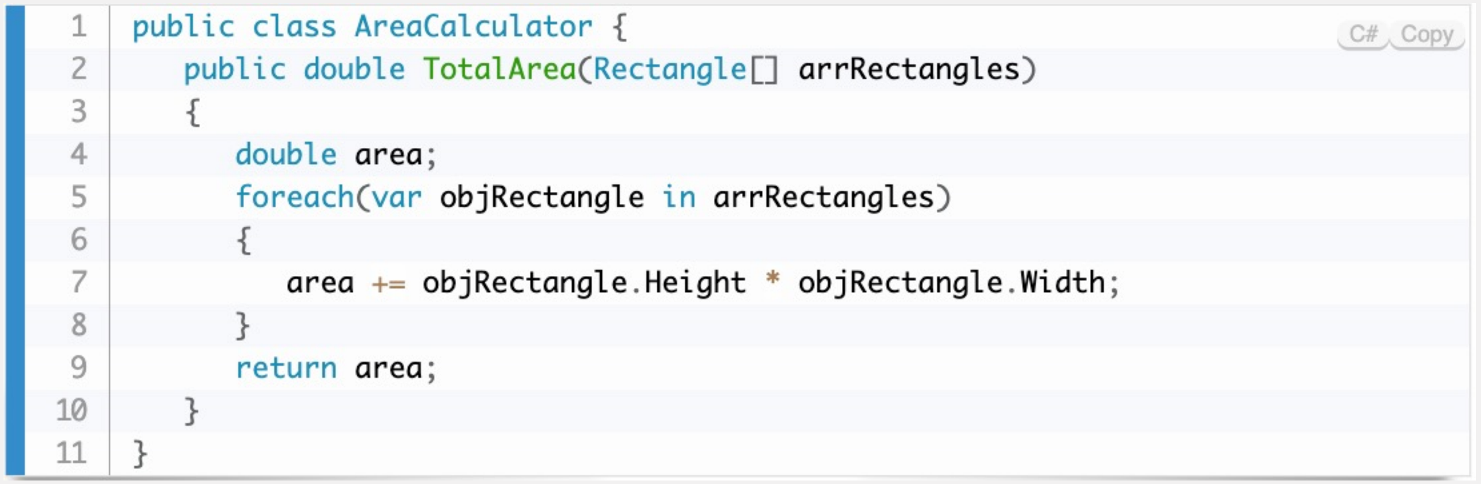
# O: Open/Closed Principle (OCP)

***A software module/class is open for extension and closed for modification***

**Scenario:** Our app needs to calculate the total area of a collection of Rectangles



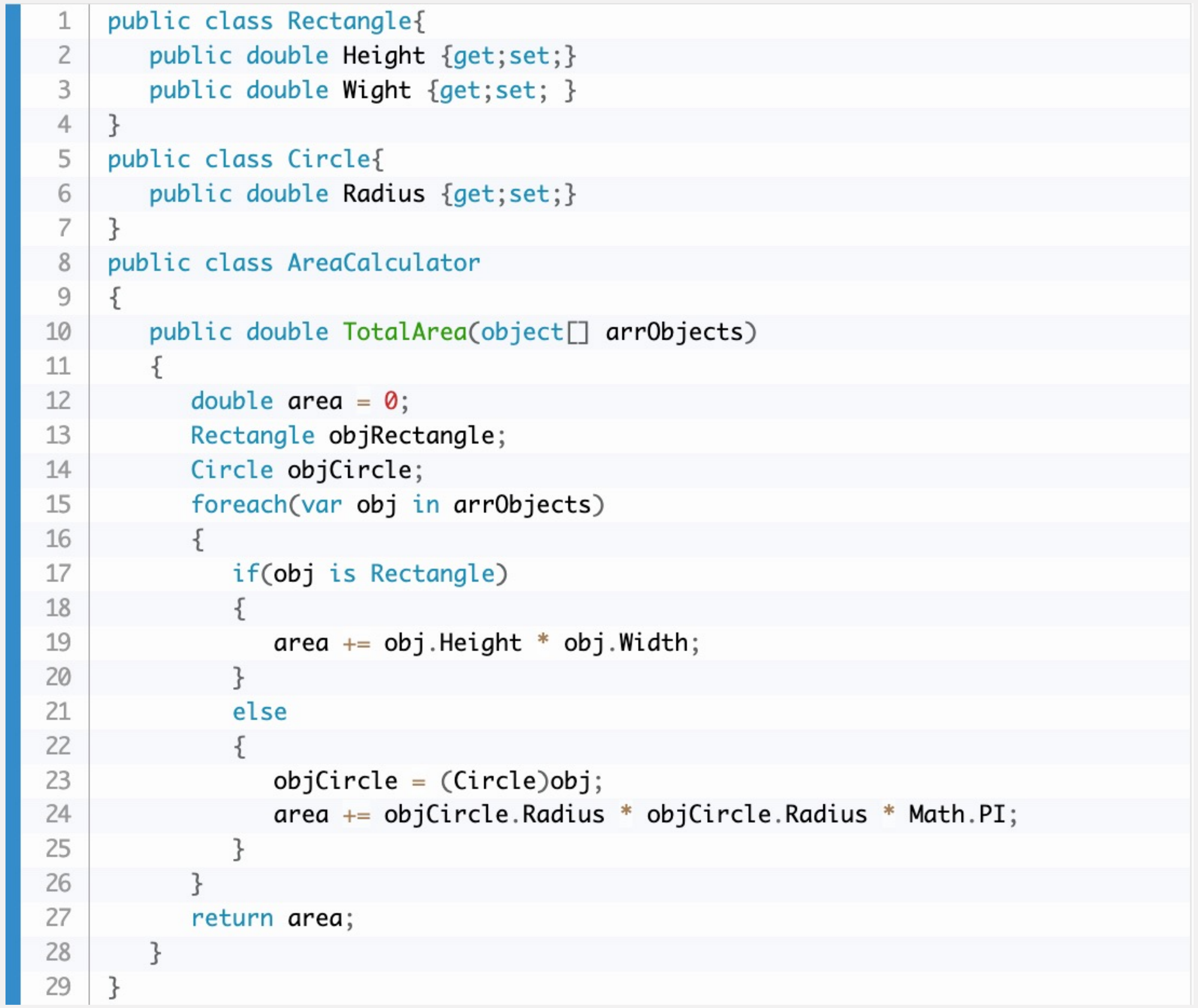
```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}
```



```
public class AreaCalculator {
   public double TotalArea(Rectangle[] arrRectangles)
   {
      double area;
      foreach(var objRectangle in arrRectangles)
      {
         area += objRectangle.Height * objRectangle.Width;
      }
      return area;
   }
}
```

Can we extend our app so that it can calculate the area of not only Rectangles but also the area of Circles?

# O: Open/Closed Principle (OCP)



```
public class Rectangle{
    public double Height {get;set;}
    public double Wight {get;set; }
}
public class Circle{
    public double Radius {get;set;}
}
public class AreaCalculator
{
    public double TotalArea(object[] arrObjects)
    {
        double area = 0;
        Rectangle objRectangle;
        Circle objCircle;
        foreach(var obj in arrObjects)
        {
            if(obj is Rectangle)
            {
                area += obj.Height * obj.Width;
            }
            else
            {
                objCircle = (Circle)obj;
                area += objCircle.Radius * objCircle.Radius * Math.PI;
            }
        }
        return area;
    }
}
```

Can we add a Triangle and calculate its area by adding one more "if" block in the TotalArea method of AreaCalculator

## O: Open/Closed Principle (OCP)

```
public abstract class Shape
{
    public abstract double Area();
}
```

```
public class Rectangle: Shape
{
    public double Height {get;set;}
    public double Width {get;set;}
    public override double Area()
    {
        return Height * Width;
    }
}
public class Circle: Shape
{
    public double Radius {get;set;}
    public override double Area()
    {
        return Radius * Radus * Math.PI;
    }
}
```

```
public class AreaCalculator
{
    public double TotalArea(Shape[] arrShapes)
    {
        double area=0;
        foreach(var objShape in arrShapes)
        {
            area += objShape.Area();
        }
        return area;
    }
}
```

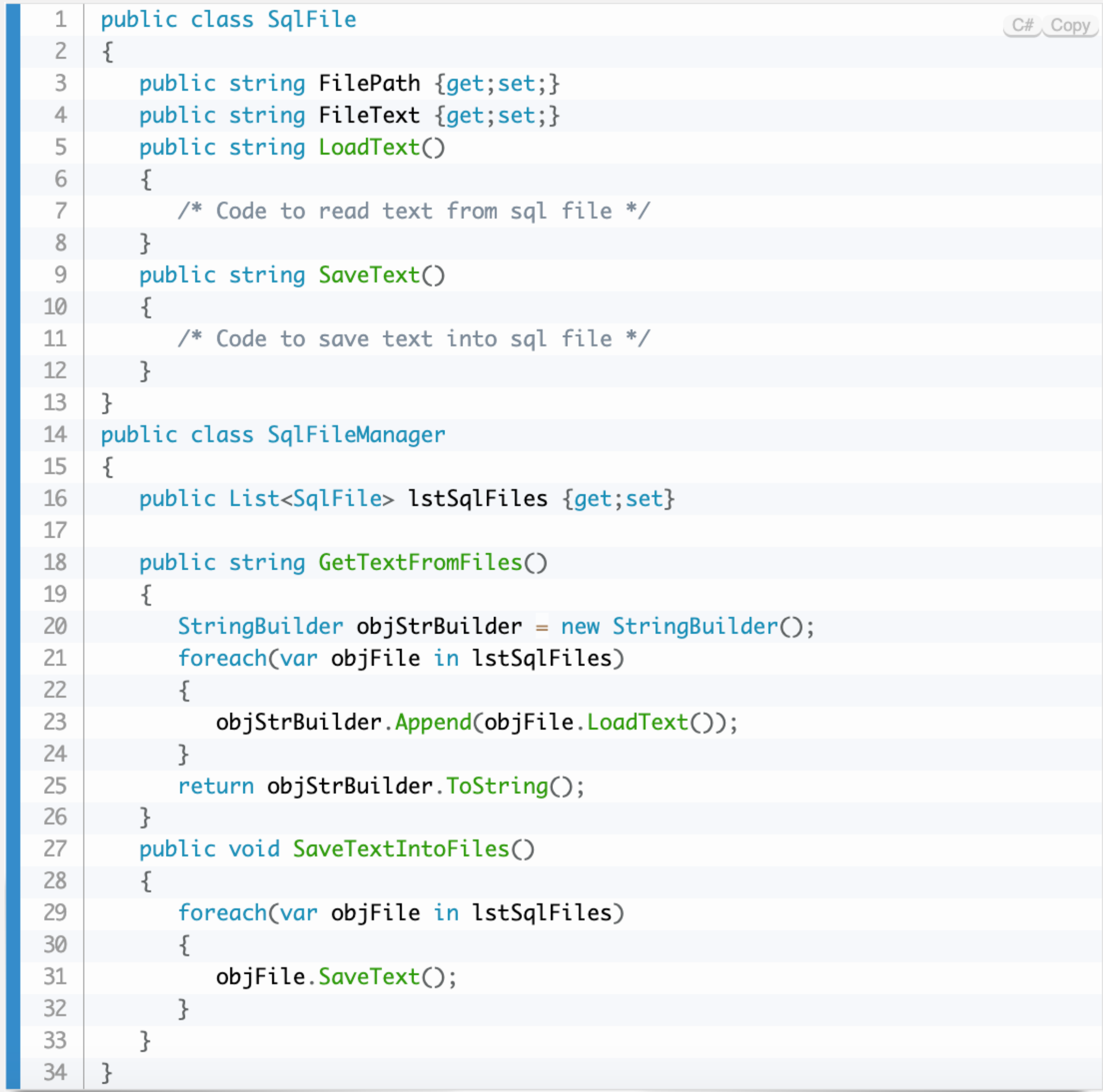
Now our code is following SRP and OCP both

# L: Liskov Substitution Principle (LSP)

- You should be able to use any derived class instead of a parent class and have it behave in the same manner without modification

**Scenario:** Suppose we need to build an app to manage data using a group of SQL files text. Here we need to write functionality to load and save the text of a group of SQL files in the application directory. So we need a class that manages the load and keeps the text of a group of SQL files along with the SqlFile Class.

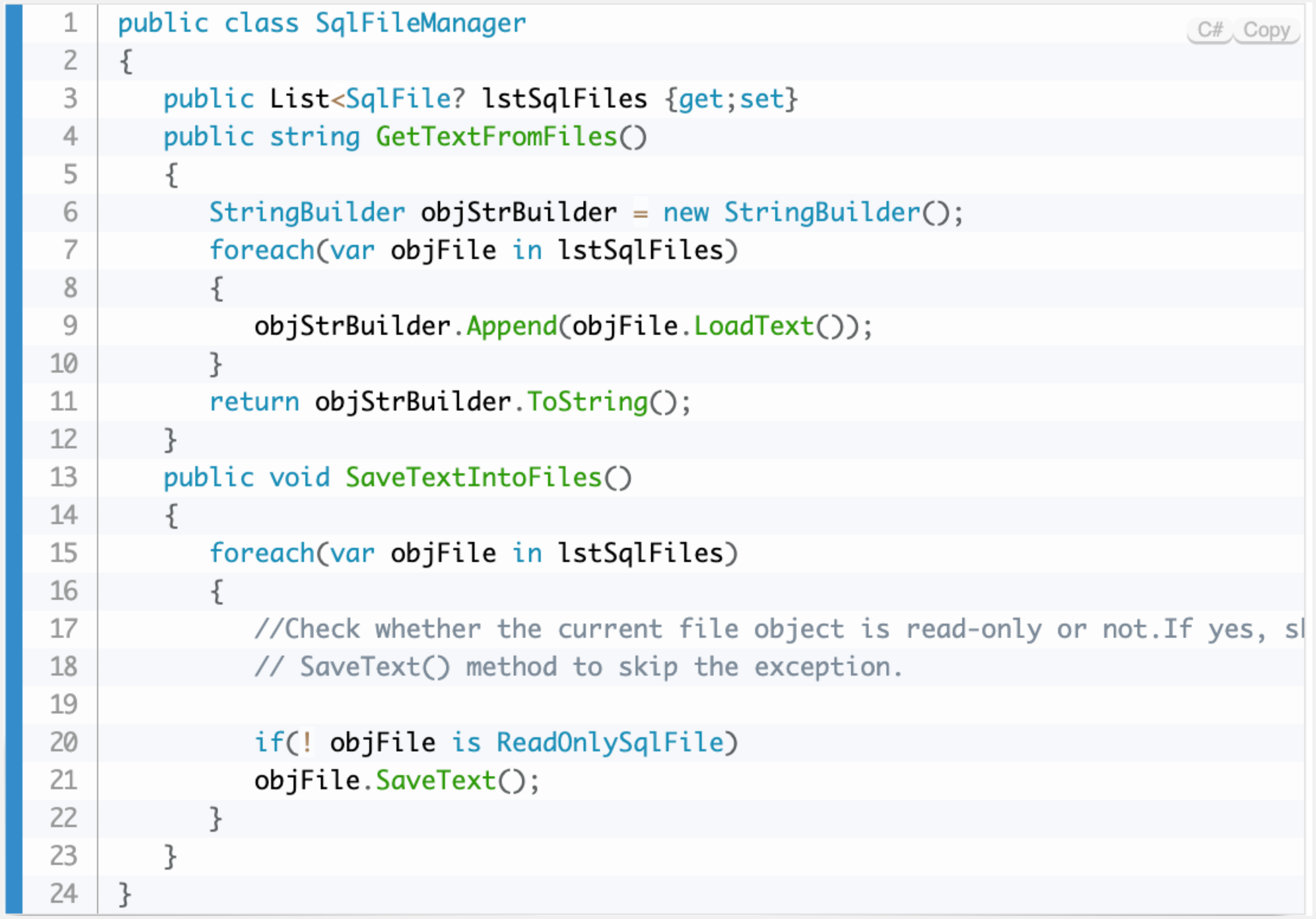
**New Requirement:** our leaders might tell us that we may have a few read-only files in the application folder, so we must restrict the flow whenever it tries to save them.



```csharp
public class SqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
   public string SaveText()
   {
      /* Code to save text into sql file */
   }
}
public class SqlFileManager
{
   public List<SqlFile> lstSqlFiles {get;set}

   public string GetTextFromFiles()
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in lstSqlFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles()
   {
      foreach(var objFile in lstSqlFiles)
      {
         objFile.SaveText();
      }
   }
}
```

# L: Liskov Substitution Principle (LSP)

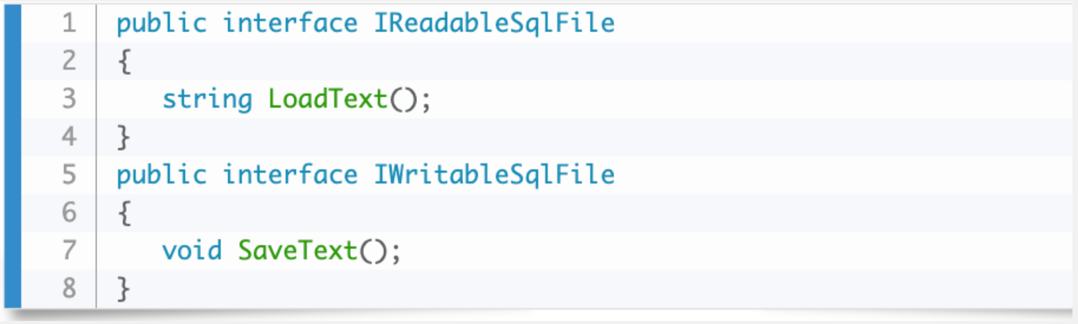


C# Copy

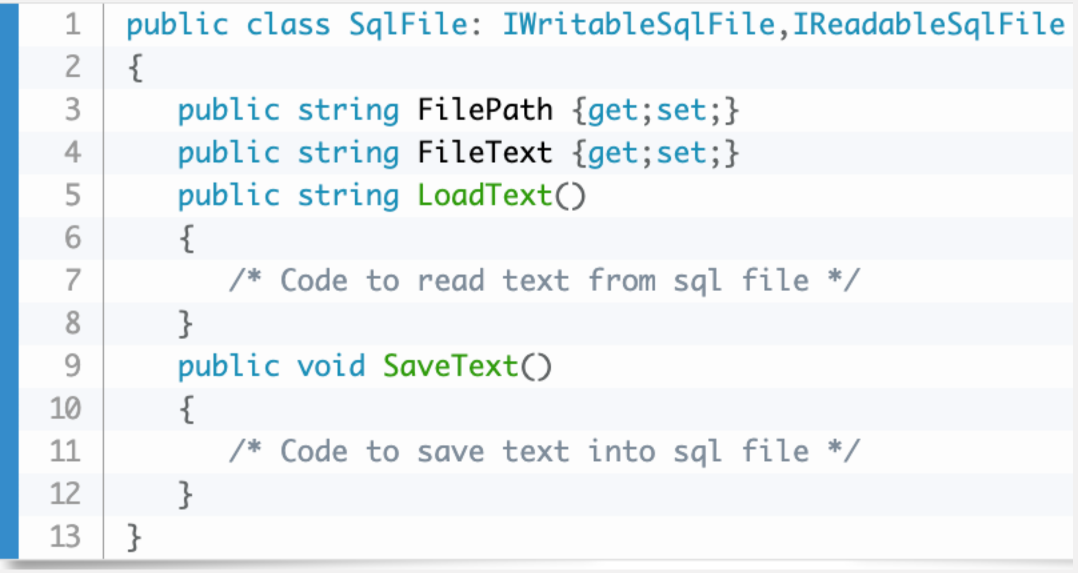
```
public class SqlFileManager
{
    public List<SqlFile? lstSqlFiles {get;set}
    public string GetTextFromFiles()
    {
        StringBuilder objStrBuilder = new StringBuilder();
        foreach(var objFile in lstSqlFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }
        return objStrBuilder.ToString();
    }
    public void SaveTextIntoFiles()
    {
        foreach(var objFile in lstSqlFiles)
        {
            //Check whether the current file object is read-only or not.If yes, sl
            // SaveText() method to skip the exception.

            if(! objFile is ReadOnlySqlFile)
            objFile.SaveText();
        }
    }
}
```

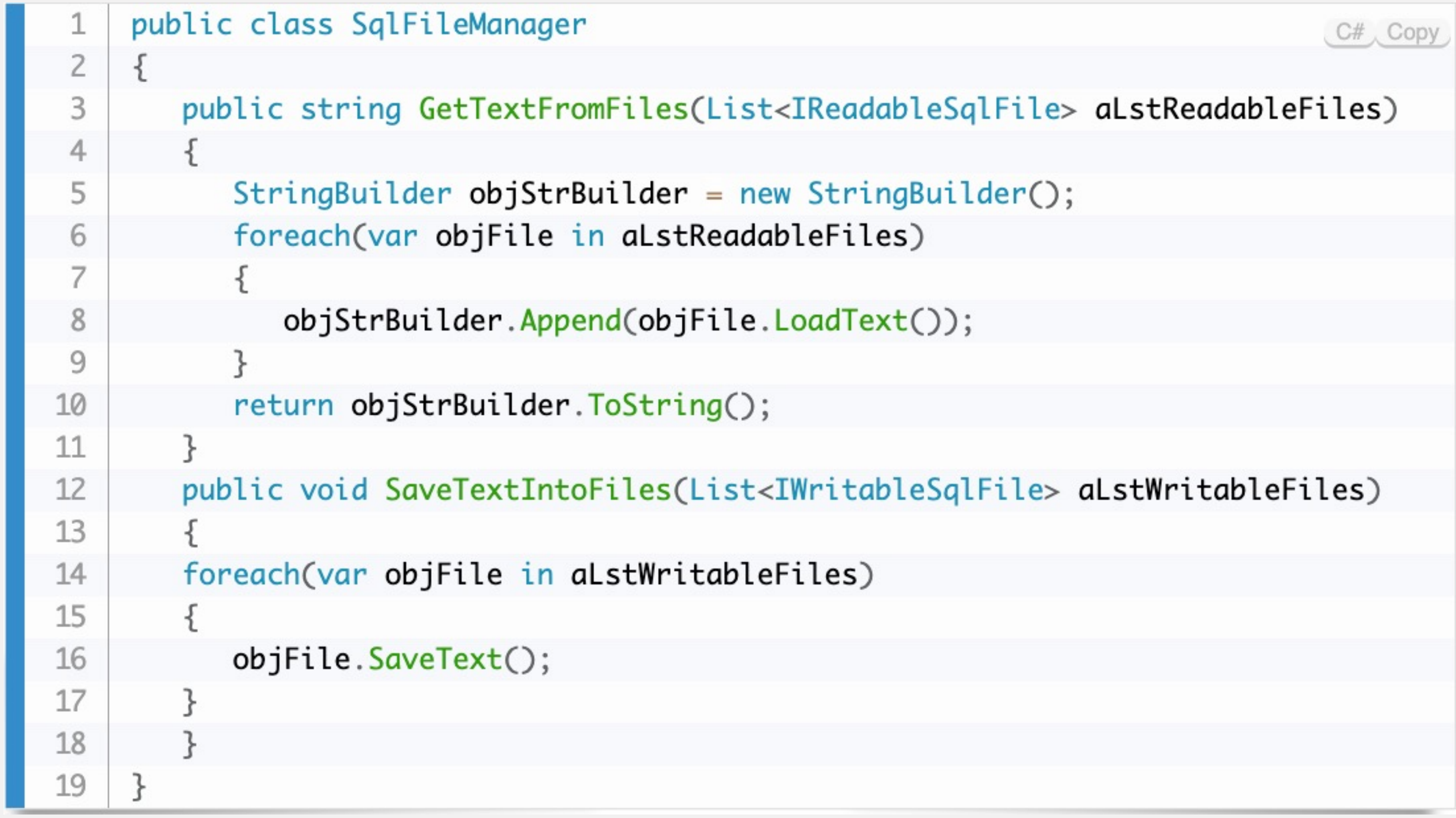
# L: Liskov Substitution Principle (LSP)



```csharp
public interface IReadableSqlFile
{
   string LoadText();
}
public interface IWritableSqlFile
{
   void SaveText();
}
```



```csharp
public class SqlFile: IWritableSqlFile,IReadableSqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
   public void SaveText()
   {
      /* Code to save text into sql file */
   }
}
```

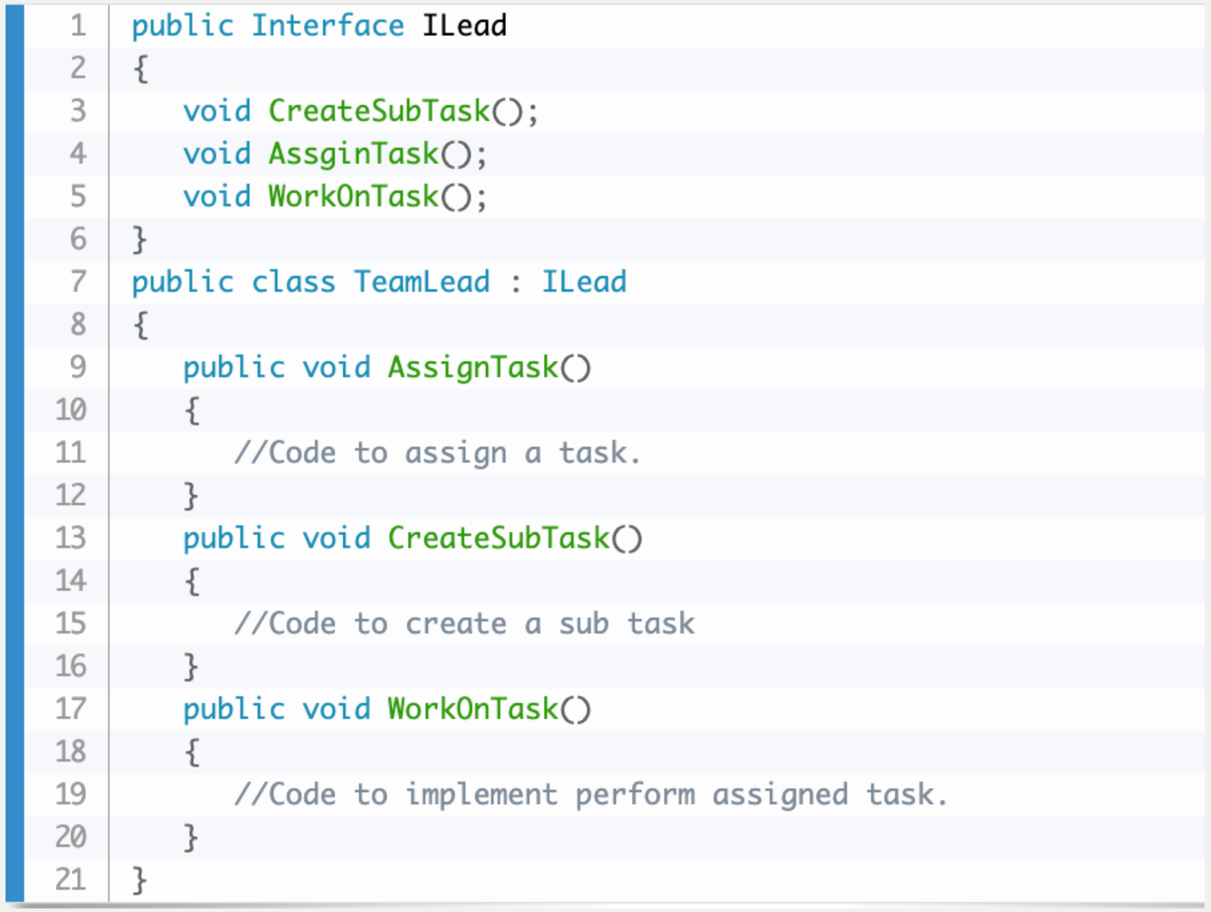


```csharp
public class SqlFileManager
{
    public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
    {
        StringBuilder objStrBuilder = new StringBuilder();
        foreach(var objFile in aLstReadableFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }
        return objStrBuilder.ToString();
    }
    public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
    {
    foreach(var objFile in aLstWritableFiles)
    {
        objFile.SaveText();
    }
    }
}
```

## I: Interface Segregation Principle (ISP)

***"that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each serving one submodule."***

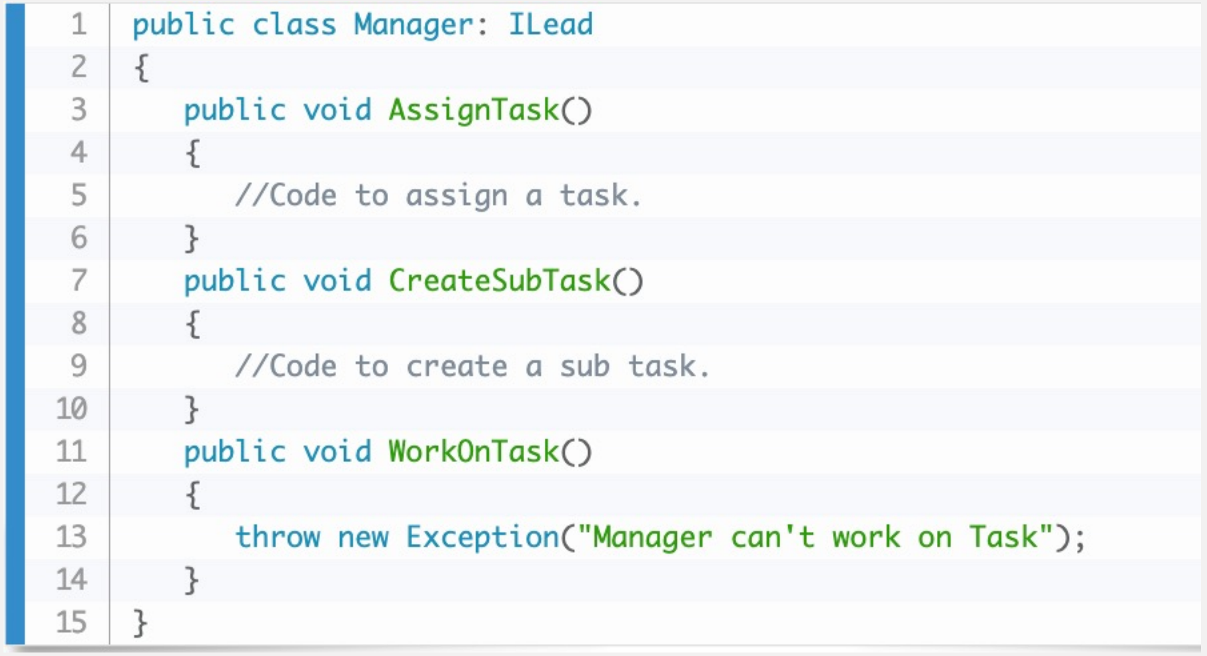
**Scenario:** Based on specifications, we need to create an interface and a TeamLead class to implement it.



```
public Interface ILead
{
    void CreateSubTask();
    void AssginTask();
    void WorkOnTask();
}
public class TeamLead : ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task
    }
    public void WorkOnTask()
    {
        //Code to implement perform assigned task.
    }
}
```

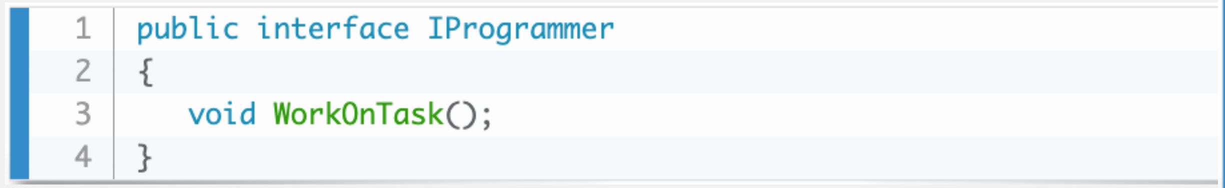
## I: Interface Segregation Principle (ISP)

**New Requirement:** Later another role, like Manager, who assigns tasks to TeamLead and will not work on the tasks, is introduced into the system

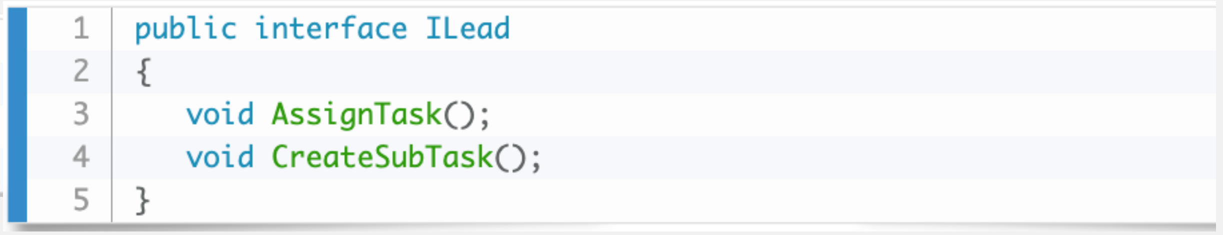


```
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task.
    }
    public void WorkOnTask()
    {
        throw new Exception("Manager can't work on Task");
    }
}
```

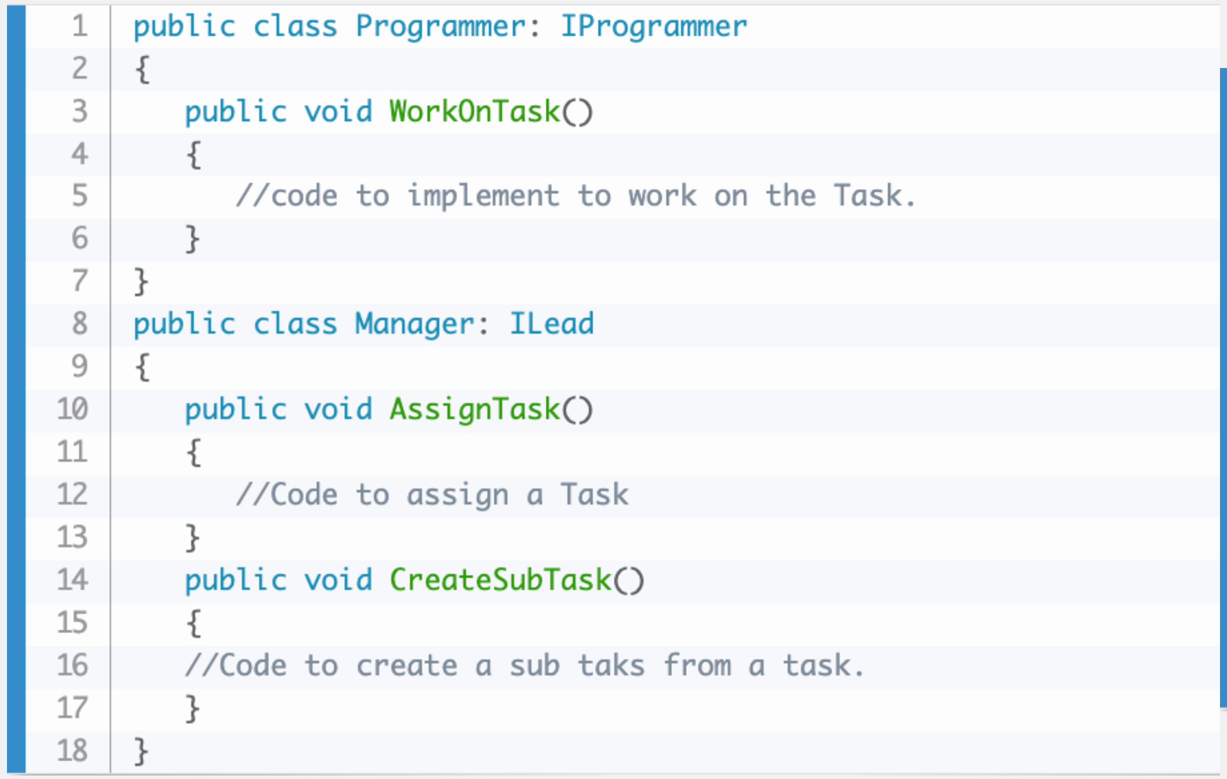
# I: Interface Segregation Principle (ISP)



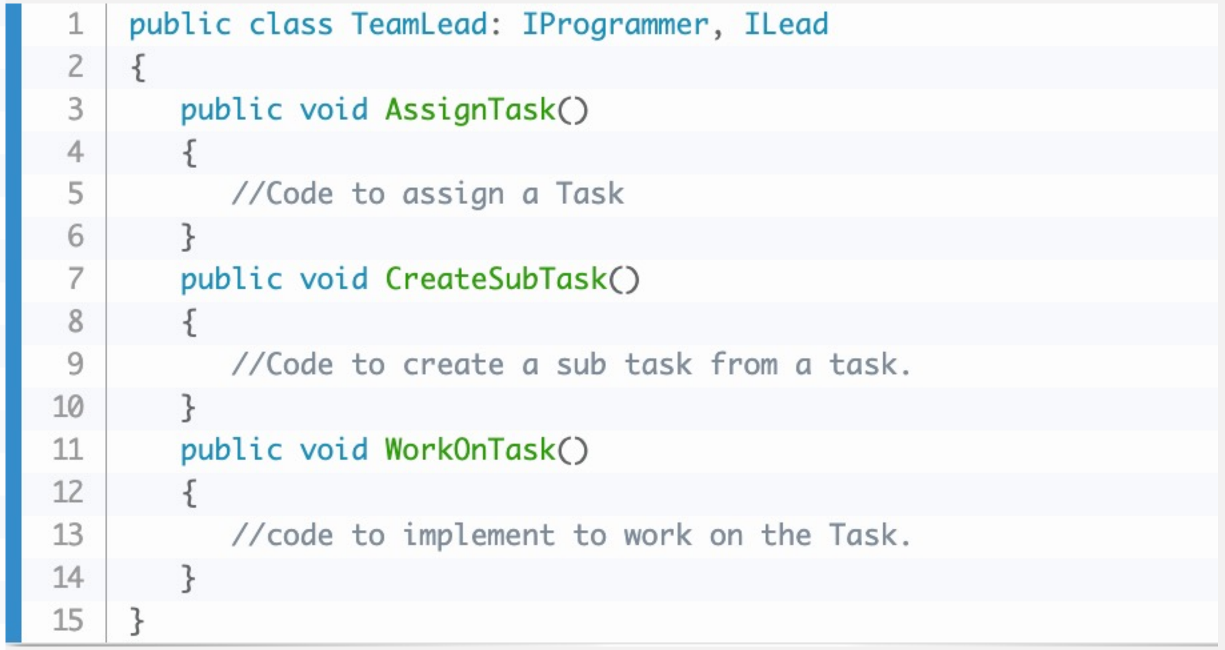
```
public interface IProgrammer
{
    void WorkOnTask();
}
```



```
public interface ILead
{
    void AssignTask();
    void CreateSubTask();
}
```



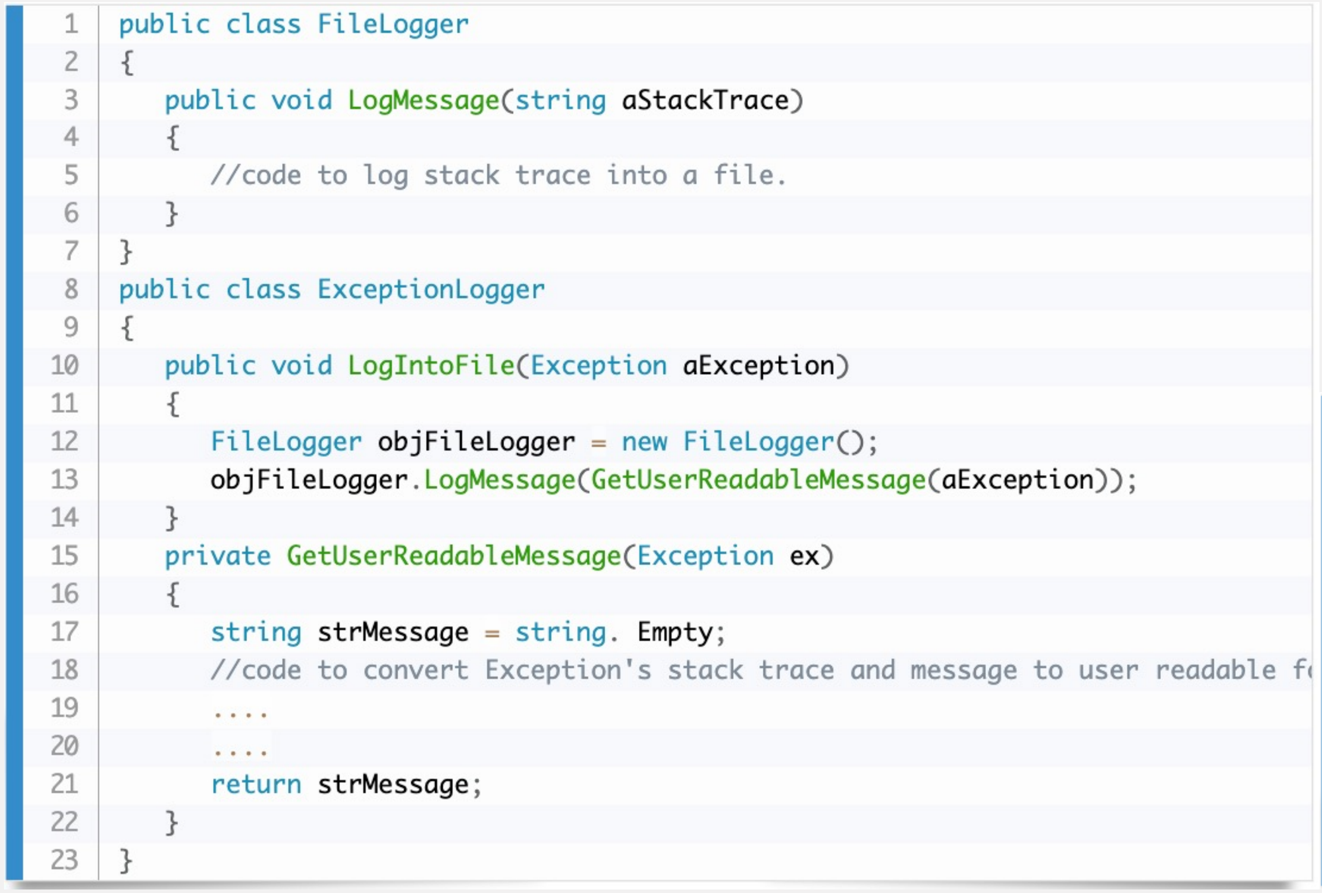
```
public class Programmer: IProgrammer
{
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
    //Code to create a sub taks from a task.
    }
}
```



```
public class TeamLead: IProgrammer, ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
        //Code to create a sub task from a task.
    }
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
```

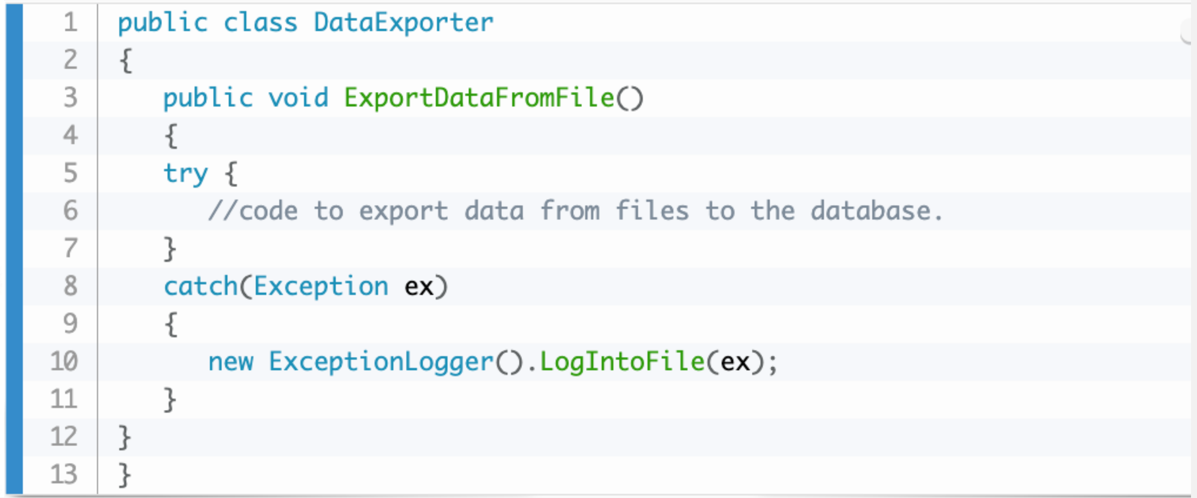
## D: Dependency Inversion Principle (DIP)

***"high-level modules/classes should not depend on low-level modules/classes. High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations;"***



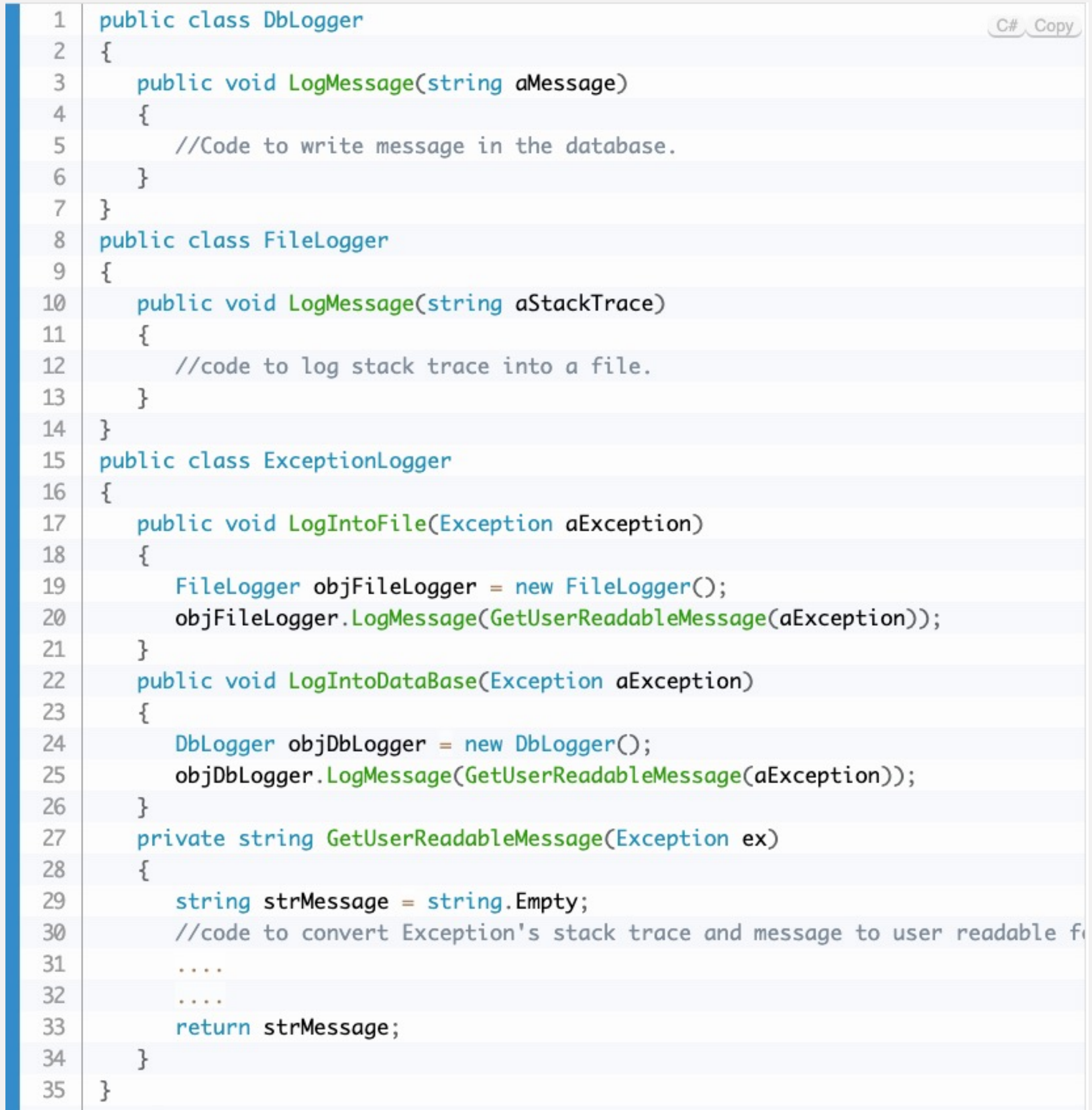
```
public class FileLogger
{
   public void LogMessage(string aStackTrace)
   {
      //code to log stack trace into a file.
   }
}
public class ExceptionLogger
{
   public void LogIntoFile(Exception aException)
   {
      FileLogger objFileLogger = new FileLogger();
      objFileLogger.LogMessage(GetUserReadableMessage(aException));
   }
   private GetUserReadableMessage(Exception ex)
   {
      string strMessage = string. Empty;
      //code to convert Exception's stack trace and message to user readable f
      ....
      ....
      return strMessage;
   }
}
```

**Scenario:** Suppose we need to work on an error-logging module that logs exception stack traces into a file. Simple, isn't it? The following classes provide the functionality to log a stack trace into a file.



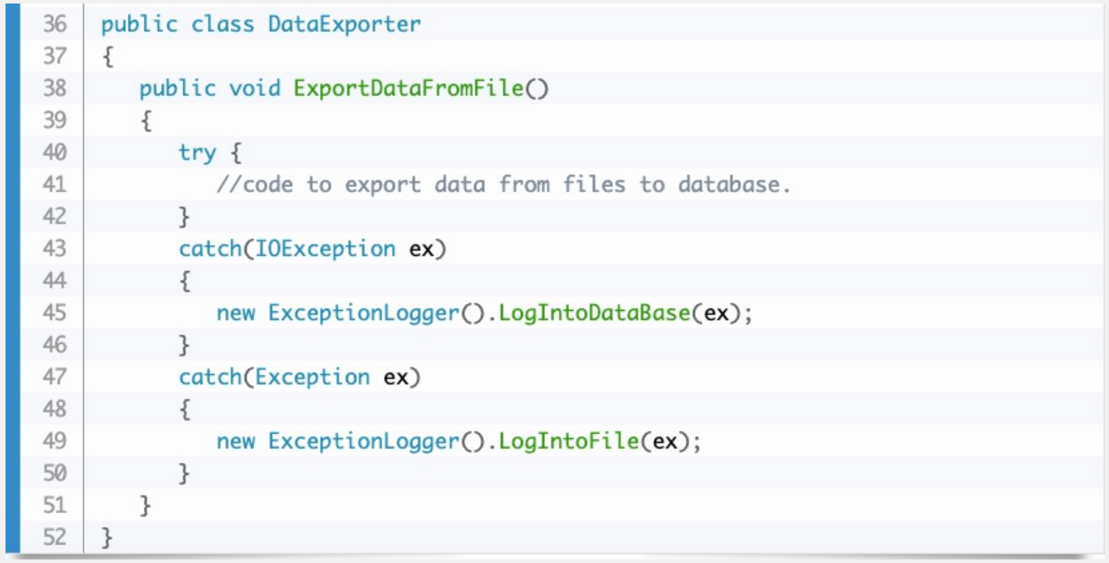
```
public class DataExporter
{
   public void ExportDataFromFile()
   {
   try {
      //code to export data from files to the database.
   }
   catch(Exception ex)
   {
      new ExceptionLogger().LogIntoFile(ex);
   }
}
}
```

# D: Dependency Inversion Principle (DIP)



```
public class DbLogger
{
    public void LogMessage(string aMessage)
    {
        //Code to write message in the database.
    }
}
public class FileLogger
{
    public void LogMessage(string aStackTrace)
    {
        //code to log stack trace into a file.
    }
}
public class ExceptionLogger
{
    public void LogIntoFile(Exception aException)
    {
        FileLogger objFileLogger = new FileLogger();
        objFileLogger.LogMessage(GetUserReadableMessage(aException));
    }
    public void LogIntoDataBase(Exception aException)
    {
        DbLogger objDbLogger = new DbLogger();
        objDbLogger.LogMessage(GetUserReadableMessage(aException));
    }
    private string GetUserReadableMessage(Exception ex)
    {
        string strMessage = string.Empty;
        //code to convert Exception's stack trace and message to user readable f
        ....
        ....
        return strMessage;
    }
}
```

**New Requirement:** our client wants to store this stack trace in a database if an IO exception occurs



```
public class DataExporter
{
    public void ExportDataFromFile()
    {
        try {
            //code to export data from files to database.
        }
        catch(IOException ex)
        {
            new ExceptionLogger().LogIntoDataBase(ex);
        }
        catch(Exception ex)
        {
            new ExceptionLogger().LogIntoFile(ex);
        }
    }
}
```

Looks Good?

## D: Dependency Inversion Principle

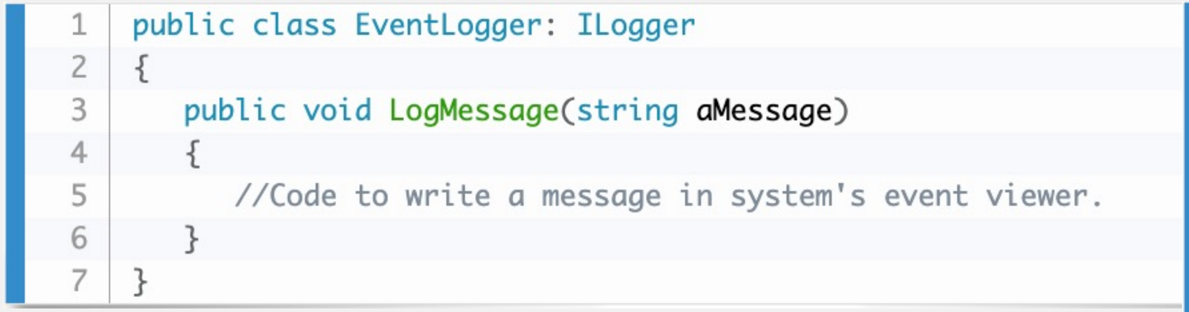
```csharp
public interface ILogger
{
    void LogMessage(string aString);
}
```

```csharp
public class DbLogger: ILogger
{
    public void LogMessage(string aMessage)
    {
        //Code to write message in database.
    }
}
public class FileLogger: ILogger
{
    public void LogMessage(string aStackTrace)
    {
        //code to log stack trace into a file.
    }
}
```

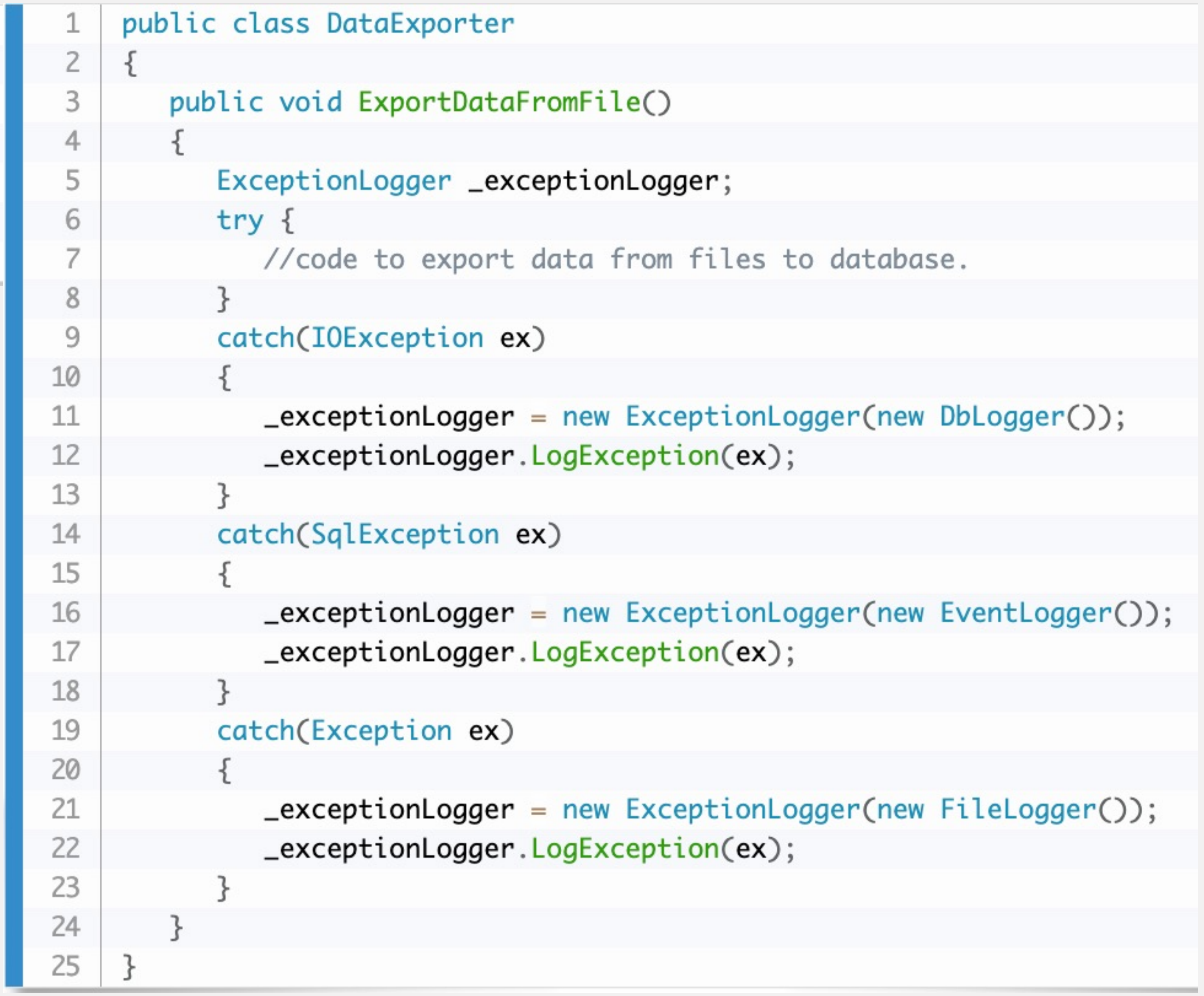
How to add more Loggers?

```csharp
public class ExceptionLogger
{
    private ILogger _logger;
    public ExceptionLogger(ILogger aLogger)
    {
        this._logger = aLogger;
    }
    public void LogException(Exception aException)
    {
        string strMessage = GetUserReadableMessage(aException);
        this._logger.LogMessage(strMessage);
    }
    private string GetUserReadableMessage(Exception aException)
    {
        string strMessage = string.Empty;
        //code to convert Exception's stack trace and message to user readable f
        ....
        ....
        return strMessage;
    }
}
public class DataExporter
{
    public void ExportDataFromFile()
    {
        ExceptionLogger _exceptionLogger;
        try {
            //code to export data from files to database.
        }
        catch(IOException ex)
        {
            _exceptionLogger = new ExceptionLogger(new DbLogger());
            _exceptionLogger.LogException(ex);
        }
        catch(Exception ex)
        {
            _exceptionLogger = new ExceptionLogger(new FileLogger());
            _exceptionLogger.LogException(ex);
        }
    }
}
```

## D: Dependency Inversion Principle (DIP)



```
public class EventLogger: ILogger
{
    public void LogMessage(string aMessage)
    {
        //Code to write a message in system's event viewer.
    }
}
```



```
public class DataExporter
{
    public void ExportDataFromFile()
    {
        ExceptionLogger _exceptionLogger;
        try {
            //code to export data from files to database.
        }
        catch(IOException ex)
        {
            _exceptionLogger = new ExceptionLogger(new DbLogger());
            _exceptionLogger.LogException(ex);
        }
        catch(SqlException ex)
        {
            _exceptionLogger = new ExceptionLogger(new EventLogger());
            _exceptionLogger.LogException(ex);
        }
        catch(Exception ex)
        {
            _exceptionLogger = new ExceptionLogger(new FileLogger());
            _exceptionLogger.LogException(ex);
        }
    }
}
```

SRP & OCP also accomplished