S- Single-responsiblity Principle

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

Example: Seperate the classes or services; UserService, EMailService

O- Open-Closed Principle

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

example: we have a class called shape and rectangle, circle classes inherited from shape. Each class has its own area and a new shape comes in, we need to create new class. and to calculate their area, in another class we can call their area method and return it.

L- Liskov Substitution Principle

States that “You should be able to use any derived class instead of a parent class and have it behave in the same manner without modification”. It ensures that a derived class does not affect the behaviour of the parent class.

public interface IReadableSqlFile

{

string LoadText();

}

public interface IWritableSqlFile

{

void SaveText();

}

public class ReadOnlySqlFile: IReadableSqlFile

{

public string FilePath{get;set;}

public string FileText{get;set;}

public string LoadText(){}

}

public class SqlFile: IWritableSqlFile,IReadableSqlFile

{

public string FilePath{get;set;}

public string FileText{get;set;}

public string LoadText(){}

public void SaveText(){}

}

public class SqlFileManager

{

public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)

{

….

}

public string SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)

{

…

}

}

I- Interface Segregation Principle

States that “clients should not be forced to implement interfaces they do not use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.”

public interface IProgrammer

{

void WorkOnTask();

}

public interface ILead

{

void AssignTask();

void CreateSubTask();

}

public class Programmer: IProgrammer

{

public void WorkOnTask(){}

}

public class Manager: ILead

{

public void AssignTask(){}

public void CreateSubTask(){}

}

public class TeamLead: IProgrammer, ILead

D- Dependency Inversion Principle

states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

public interface ILogger

{

void LogMessage(string aString);

}

public class DbLogger:ILogger

{

public void LogMessage(string aMessage)

{

….

}

}

public class FileLogger:ILogger

{

public void LogMessage(string aStackTrace)

{

……

}

}

public class ExceptionLogger

{

private ILogger \_logger;

public ExceptionLogger(ILogger aLogger)

{

this.\_logger = aLogger;

}

public LogException(Exception aException)

{

this.\_logger.LogMessage(aException.Message);

}

}

now we do not depend on dblogger or filelogger in exceptionLogger class. we can add more logger classes and call them like below

ExceptionLogger \_exceptionLogger;

try{...

}

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);

}

//do not use like this it iss bad usage

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 format.

....

....

return strMessage;

}

}