SOLID design principles in C# are basic design principles. SOLID stands for Single Responsibility Principle (SRP), Open closed Principle (OSP), Liskov substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP).

Here let's learn basics of SOLID design principles using C# and .NET.

1. The reasons behind most unsuccessful applications
2. Solutions
3. Intro to SOLID principles
4. SRP
5. OCP
6. LSP
7. ISP
8. DIP



The reason behind most unsuccessful applications

Developers start building applications with good and tidy designs using their knowledge and experience. But over time, applications might develop bugs. The application design must be altered for every change request or new feature request. After some time we might need to put in a lot of effort, even for simple tasks and it might require full working knowledge of the entire system. But we can't blame change or new feature requests. They are part of software development. We can't stop them or refuse them either. So who is the culprit here? Obviously it is the design of the application.

The following are the design flaws that cause damage in software, mostly.

1. Putting more stress on classes by assigning more responsibilities to them. (A lot of functionality not related to a class.)
2. Forcing the classes to depend on each other. If classes are dependent on each other (in other words tightly coupled), then a change in one will affect the other.
3. Spreading duplicate code in the system/application.

**Solution**

1. Choosing the correct architecture (in other words MVC, 3-tier, Layered, MVP, MVVP and so on).
2. Following Design Principles.
3. Choosing correct Design Patterns to build the software based on its specifications.

Now we go through the Design Principles first and will cover the rest soon.

Intro to SOLID principles

SOLID principles are the design principles that enable us to manage most of the software design problems. Robert C. Martin compiled these principles in the 1990s. These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly. SOLID is an acronym of the following.

* S: Single Responsibility Principle (SRP)
* O: Open closed Principle (OCP)
* L: Liskov substitution Principle (LSP)
* I: Interface Segregation Principle (ISP)
* D: Dependency Inversion Principle (DIP)

S: Single Responsibility Principle (SRP)

SRP says "Every software module should have only one reason to change".



This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to single responsibility.

The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when we have the full picture of how the application should work. Let us check this with an example.

1. **public** **class** UserService
2. {

**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);

   }

1. }

It looks fine, but it is not following SRP. The SendEmail and ValidateEmail methods have nothing to do within the UserService class. Let's refract it.

1. **public** **class** UserService
2. {
3. EmailService \_emailService;
4. DbContext \_dbContext;
5. **public** UserService(EmailService aEmailService, DbContext aDbContext)
6. {
7. \_emailService = aEmailService;
8. \_dbContext = aDbContext;
9. }
10. **public** **void** Register(**string** email, **string** password)
11. {
12. **if** (!\_emailService.ValidateEmail(email))
13. **throw** **new** ValidationException("Email is not an email");
14. var user = **new** User(email, password);
15. \_dbContext.Save(user);
16. emailService.SendEmail(**new** MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
18. }
19. }
20. **public** **class** EmailService
21. {
22. SmtpClient \_smtpClient;
23. **public** EmailService(SmtpClient aSmtpClient)
24. {
25. \_smtpClient = aSmtpClient;
26. }
27. **public** **bool** **virtual** ValidateEmail(**string** email)
28. {
29. **return** email.Contains("@");
30. }
31. **public** **bool** SendEmail(MailMessage message)
32. {
33. \_smtpClient.Send(message);
34. }
35. }

O: Open/Closed Principle

The Open/closed Principle says "A software module/class is open for extension and closed for modification".



Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this. Okay, let's dive into an example.

Suppose we have a Rectangle class with the properties Height and Width.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }

Our app needs the ability to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here I created another class for area calculation.

1. **public** **class** AreaCalculator {
2. **public** **double** TotalArea(Rectangle[] arrRectangles)
3. {
4. **double** area;
5. **foreach**(var objRectangle **in** arrRectangles)
6. {
7. area += objRectangle.Height \* objRectangle.Width;
8. }
9. **return** area;
10. }
11. }

Hey, we did it. We made our app without violating SRP. No issues for now. But can we extend our app so that it could calculate the area of not only Rectangles but also the area of Circles as well? Now we have an issue with the area calculation issue because the way to do circle area calculation is different. Hmm. Not a big deal. We can change the TotalArea method a bit so that it can accept an array of objects as an argument. We check the object type in the loop and do area calculation based on the object type.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }
5. **public** **class** Circle{
6. **public** **double** Radius {**get**;**set**;}
7. }
8. **public** **class** AreaCalculator
9. {
10. **public** **double** TotalArea(**object**[] arrObjects)
11. {
12. **double** area = 0;
13. Rectangle objRectangle;
14. Circle objCircle;
15. **foreach**(var obj **in** arrObjects)
16. {
17. **if**(obj **is** Rectangle)
18. {
19. area += obj.Height \* obj.Width;
20. }
21. **else**
22. {
23. objCircle = (Circle)obj;
24. area += objCircle.Radius \* objCircle.Radius \* Math.PI;
25. }
26. }
27. **return** area;
28. }
29. }

Wow. We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate it's the area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape we need to alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation? Generally, we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.

**public** **abstract** **class** Shape

{

**public** **abstract** **double** Area();

}

Inheriting from Shape, the Rectangle and Circle classes now look like this:

1. **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;

   }

}

Every shape contains its area with its own way of calculation functionality and our AreaCalculator class will become simpler than before.

**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. Whenever you introduce a new shape by deriving from the "Shape" abstract class, you need not change the "AreaCalculator" class. Awesome. Isn't it?

L: Liskov Substitution Principle



The Liskov Substitution Principle (LSP) 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 behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.

This principle is just an extension of the Open Closed Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior. I will explain this with a real-world example that violates LSP.

A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.

Now jump into an example to learn how a design can violate LSP. 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 saves the text of group of SQL files along with the SqlFile Class.

1. **public** **class** SqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. **public** **string** SaveText()
10. {
11. /\* Code to save text into sql file \*/
12. }
13. }
14. **public** **class** SqlFileManager
15. {
16. **public** List<SqlFile> lstSqlFiles {**get**;**set**}
18. **public** **string** GetTextFromFiles()
19. {
20. StringBuilder objStrBuilder = **new** StringBuilder();
21. **foreach**(var objFile **in** lstSqlFiles)
22. {
23. objStrBuilder.Append(objFile.LoadText());
24. }
25. **return** objStrBuilder.ToString();
26. }
27. **public** **void** SaveTextIntoFiles()
28. {
29. **foreach**(var objFile **in** lstSqlFiles)
30. {
31. objFile.SaveText();
32. }
33. }
34. }

OK. We are done with our part. The functionality looks good for now. After some time our leaders might tell us that we may have a few read-only files in the application folder, so we need to restrict the flow whenever it tries to do a save on them.

OK. We can do that by creating a "ReadOnlySqlFile" class that inherits the "SqlFile" class and we need to alter the SaveTextIntoFiles() method by introducing a condition to prevent calling the SaveText() method on ReadOnlySqlFile instances.

1. **public** **class** SqlFile
2. {
3. **public** **string** LoadText()
4. {
5. /\* Code to read text from sql file \*/
6. }
7. **public** **void** SaveText()
8. {
9. /\* Code to save text into sql file \*/
10. }
11. }
12. **public** **class** ReadOnlySqlFile: SqlFile
13. {
14. **public** **string** FilePath {**get**;**set**;}
15. **public** **string** FileText {**get**;**set**;}
16. **public** **string** LoadText()
17. {
18. /\* Code to read text from sql file \*/
19. }
20. **public** **void** SaveText()
21. {
22. /\* Throw an exception when app flow tries to do save. \*/
23. **throw** **new** IOException("Can't Save");
24. }
25. }

To avoid an exception we need to modify "SqlFileManager" by adding one condition to the loop.

1. **public** **class** SqlFileManager
2. {
3. **public** List<SqlFile? lstSqlFiles {**get**;**set**}
4. **public** **string** GetTextFromFiles()
5. {
6. StringBuilder objStrBuilder = **new** StringBuilder();
7. **foreach**(var objFile **in** lstSqlFiles)
8. {
9. objStrBuilder.Append(objFile.LoadText());
10. }
11. **return** objStrBuilder.ToString();
12. }
13. **public** **void** SaveTextIntoFiles()
14. {
15. **foreach**(var objFile **in** lstSqlFiles)
16. {
17. //Check whether the current file object is read-only or not.If yes, skip calling it's
18. // SaveText() method to skip the exception.
20. **if**(! objFile **is** ReadOnlySqlFile)
21. objFile.SaveText();
22. }
23. }
24. }

Here we altered the SaveTextIntoFiles() method in the SqlFileManager class to determine whether or not the instance is of ReadOnlySqlFile to avoid the exception. We can't use this ReadOnlySqlFile class as a substitute for its parent without altering SqlFileManager code. So we can say that this design is not following LSP. Let's make this design follow the LSP. Here we will introduce interfaces to make the SqlFileManager class independent from the rest of the blocks.

1. **public** **interface** IReadableSqlFile
2. {
3. **string** LoadText();
4. }
5. **public** **interface** IWritableSqlFile
6. {
7. **void** SaveText();
8. }

Now we implement IReadableSqlFile through the ReadOnlySqlFile class that reads only the text from read-only files.

1. **public** **class** ReadOnlySqlFile: IReadableSqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. }

Here we implement both IWritableSqlFile and IReadableSqlFile in a SqlFile class by which we can read and write files.

1. **public** **class** SqlFile: IWritableSqlFile,IReadableSqlFile
2. {
3. **public** **string** FilePath {**get**;**set**;}
4. **public** **string** FileText {**get**;**set**;}
5. **public** **string** LoadText()
6. {
7. /\* Code to read text from sql file \*/
8. }
9. **public** **void** SaveText()
10. {
11. /\* Code to save text into sql file \*/
12. }
13. }

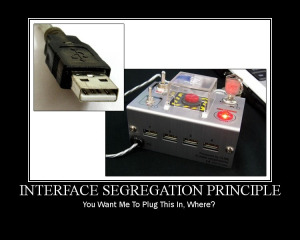
Now the design of the SqlFileManager class becomes like this:

1. **public** **class** SqlFileManager
2. {
3. **public** **string** GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
4. {
5. StringBuilder objStrBuilder = **new** StringBuilder();
6. **foreach**(var objFile **in** aLstReadableFiles)
7. {
8. objStrBuilder.Append(objFile.LoadText());
9. }
10. **return** objStrBuilder.ToString();
11. }
12. **public** **void** SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
13. {
14. **foreach**(var objFile **in** aLstWritableFiles)
15. {
16. objFile.SaveText();
17. }
18. }
19. }

Here the GetTextFromFiles() method gets only the list of instances of classes that implement the IReadOnlySqlFile interface. That means the SqlFile and ReadOnlySqlFile class instances. And the SaveTextIntoFiles() method gets only the list instances of the class that implements the IWritableSqlFiles interface, in other words, SqlFile instances in this case. Now we can say our design is following the LSP. And we fixed the problem using the Interface segregation principle by (ISP) identifying the abstraction and the responsibility separation method.

I: Interface Segregation Principle (ISP)

The Interface Segregation Principle states "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 one serving one submodule.".



We can define it in another way. An interface should be more closely related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs rather than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use.

Like classes, each interface should have a specific purpose/responsibility (refer to SRP). You shouldn't be forced to implement an interface when your object doesn't share that purpose. The larger the interface, the more likely it includes methods that not all implementers can do. That's the essence of the Interface Segregation Principle. Let's start with an example that breaks the ISP. Suppose we need to build a system for an IT firm that contains roles like TeamLead and Programmer where TeamLead divides a huge task into smaller tasks and assigns them to his/her programmers or can directly work on them.

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

1. **public** Interface ILead
2. {
3. **void** CreateSubTask();
4. **void** AssginTask();
5. **void** WorkOnTask();
6. }
7. **public** **class** TeamLead : ILead
8. {
9. **public** **void** AssignTask()
10. {
11. //Code to assign a task.
12. }
13. **public** **void** CreateSubTask()
14. {
15. //Code to create a sub task
16. }
17. **public** **void** WorkOnTask()
18. {
19. //Code to implement perform assigned task.
20. }
21. }

OK. The design looks fine for now. Later another role like Manager, who assigns tasks to TeamLead and will not work on the tasks, is introduced into the system. Can we directly implement an ILead interface in the Manager class, like the following?

1. **public** **class** Manager: ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a task.
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. **throw** **new** Exception("Manager can't work on Task");
14. }
15. }

Since the Manager can't work on a task and at the same time no one can assign tasks to the Manager, this WorkOnTask() should not be in the Manager class. But we are implementing this class from the ILead interface, we need to provide a concrete Method. Here we are forcing the Manager class to implement a WorkOnTask() method without a purpose. This is wrong. The design violates ISP. Let's correct the design.

Since we have three roles, 1. Manager, that can only divide and assign the tasks, 2. TeamLead that can divide and assign the tasks and can work on them as well, 3. The programmer that can only work on tasks, we need to divide the responsibilities by segregating the ILead interface. An interface that provides a contract for WorkOnTask().

1. **public** **interface** IProgrammer
2. {
3. **void** WorkOnTask();
4. }

An interface that provides contracts to manage the tasks:

1. **public** **interface** ILead
2. {
3. **void** AssignTask();
4. **void** CreateSubTask();
5. }

Then the implementation becomes:

1. **public** **class** Programmer: IProgrammer
2. {
3. **public** **void** WorkOnTask()
4. {
5. //code to implement to work on the Task.
6. }
7. }
8. **public** **class** Manager: ILead
9. {
10. **public** **void** AssignTask()
11. {
12. //Code to assign a Task
13. }
14. **public** **void** CreateSubTask()
15. {
16. //Code to create a sub taks from a task.
17. }
18. }

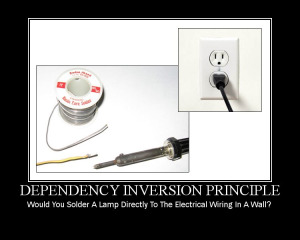
TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.

1. ***public******class****TeamLead: IProgrammer, ILead*
2. *{*
3. ***public******void****AssignTask()*
4. *{*
5. *//Code to assign a Task*
6. *}*
7. ***public******void****CreateSubTask()*
8. *{*
9. *//Code to create a sub task from a task.*
10. *}*
11. ***public******void****WorkOnTask()*
12. *{*
13. *//code to implement to work on the Task.*
14. *}*
15. *}*

Wow. Here we separated responsibilities/purposes and distributed them on multiple interfaces and provided a good level of abstraction too.

D: Dependency Inversion Principle

The Dependency Inversion Principle (DIP) 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.



High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations; in other words they may deal with writing information to databases or passing messages to the operating system or services.

A high-level module/class that has a dependency on low-level modules/classes or some other class and knows a lot about the other classes it interacts with is said to be tightly coupled. When a class knows explicitly about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/classes loosely coupled as much as we can. To do that, we need to make both of them dependent on abstractions instead of knowing each other. Let's start with an example.

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

1. **public** **class** FileLogger
2. {
3. **public** **void** LogMessage(**string** aStackTrace)
4. {
5. //code to log stack trace into a file.
6. }
7. }
8. **public** **class** ExceptionLogger
9. {
10. **public** **void** LogIntoFile(Exception aException)
11. {
12. FileLogger objFileLogger = **new** FileLogger();
13. objFileLogger.LogMessage(GetUserReadableMessage(aException));
14. }
15. **private** GetUserReadableMessage(Exception ex)
16. {
17. **string** strMessage = **string**. Empty;
18. //code to convert Exception's stack trace and message to user readable format.
19. ....
20. ....
21. **return** strMessage;
22. }
23. }

A client class exports data from many files to a database.

1. **public** **class** DataExporter
2. {
3. **public** **void** ExportDataFromFile()
4. {
5. **try** {
6. //code to export data from files to database.
7. }
8. **catch**(Exception ex)
9. {
10. **new** ExceptionLogger().LogIntoFile(ex);
11. }
12. }
13. }

Looks good. We sent our application to the client. But our client wants to store this stack trace in a database if an IO exception occurs. Hmm... okay, no problem. We can implement that too. Here we need to add one more class that provides the functionality to log the stack trace into the database and an extra method in ExceptionLogger to interact with our new class to log the stack trace.

1. **public** **class** DbLogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in database.
6. }
7. }
8. **public** **class** FileLogger
9. {
10. **public** **void** LogMessage(**string** aStackTrace)
11. {
12. //code to log stack trace into a file.
13. }
14. }
15. **public** **class** ExceptionLogger
16. {
17. **public** **void** LogIntoFile(Exception aException)
18. {
19. FileLogger objFileLogger = **new** FileLogger();
20. objFileLogger.LogMessage(GetUserReadableMessage(aException));
21. }
22. **public** **void** LogIntoDataBase(Exception aException)
23. {
24. DbLogger objDbLogger = **new** DbLogger();
25. objDbLogger.LogMessage(GetUserReadableMessage(aException));
26. }
27. **private** **string** GetUserReadableMessage(Exception ex)
28. {
29. **string** strMessage = **string**.Empty;
30. //code to convert Exception's stack trace and message to user readable format.
31. ....
32. ....
33. **return** strMessage;
34. }
35. }
36. **public** **class** DataExporter
37. {
38. **public** **void** ExportDataFromFile()
39. {
40. **try** {
41. //code to export data from files to database.
42. }
43. **catch**(IOException ex)
44. {
45. **new** ExceptionLogger().LogIntoDataBase(ex);
46. }
47. **catch**(Exception ex)
48. {
49. **new** ExceptionLogger().LogIntoFile(ex);
50. }
51. }
52. }

Looks fine for now. But whenever the client wants to introduce a new logger, we need to alter ExceptionLogger by adding a new method. If we continue doing this after some time then we will see a fat ExceptionLogger class with a large set of methods that provide the functionality to log a message into various targets. Why does this issue occur? Because ExceptionLogger directly contacts the low-level classes FileLogger and DbLogger to log the exception. We need to alter the design so that this ExceptionLogger class can be loosely coupled with those classes. To do that we need to introduce an abstraction between them so that ExcetpionLogger can contact the abstraction to log the exception instead of depending on the low-level classes directly.

1. **public** **interface** ILogger
2. {
3. **void** LogMessage(**string** aString);
4. }

Now our low-level classes need to implement this interface.

1. **public** **class** DbLogger: ILogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in database.
6. }
7. }
8. **public** **class** FileLogger: ILogger
9. {
10. **public** **void** LogMessage(**string** aStackTrace)
11. {
12. //code to log stack trace into a file.
13. }
14. }

Now, we move to the low-level class's initiation from the ExcetpionLogger class to the DataExporter class to make ExceptionLogger loosely coupled with the low-level classes FileLogger and EventLogger. And by doing that we are giving provision to DataExporter class to decide what kind of Logger should be called based on the exception that occurs.

1. **public** **class** ExceptionLogger
2. {
3. **private** ILogger \_logger;
4. **public** ExceptionLogger(ILogger aLogger)
5. {
6. **this**.\_logger = aLogger;
7. }
8. **public** **void** LogException(Exception aException)
9. {
10. **string** strMessage = GetUserReadableMessage(aException);
11. **this**.\_logger.LogMessage(strMessage);
12. }
13. **private** **string** GetUserReadableMessage(Exception aException)
14. {
15. **string** strMessage = **string**.Empty;
16. //code to convert Exception's stack trace and message to user readable format.
17. ....
18. ....
19. **return** strMessage;
20. }
21. }
22. **public** **class** DataExporter
23. {
24. **public** **void** ExportDataFromFile()
25. {
26. ExceptionLogger \_exceptionLogger;
27. **try** {
28. //code to export data from files to database.
29. }
30. **catch**(IOException ex)
31. {
32. \_exceptionLogger = **new** ExceptionLogger(**new** DbLogger());
33. \_exceptionLogger.LogException(ex);
34. }
35. **catch**(Exception ex)
36. {
37. \_exceptionLogger = **new** ExceptionLogger(**new** FileLogger());
38. \_exceptionLogger.LogException(ex);
39. }
40. }
41. }

We successfully removed the dependency on low-level classes. This ExceptionLogger doesn't depend on the FileLogger and EventLogger classes to log the stack trace. We don't need to change the ExceptionLogger's code anymore for any new logging functionality. We need to create a new logging class that implements the ILogger interface and must add another catch block to the DataExporter class's ExportDataFromFile method.

1. **public** **class** EventLogger: ILogger
2. {
3. **public** **void** LogMessage(**string** aMessage)
4. {
5. //Code to write message in system's event viewer.
6. }
7. }

And we need to add a condition in the DataExporter class as in the following:

1. **public** **class** DataExporter
2. {
3. **public** **void** ExportDataFromFile()
4. {
5. ExceptionLogger \_exceptionLogger;
6. **try** {
7. //code to export data from files to database.
8. }
9. **catch**(IOException ex)
10. {
11. \_exceptionLogger = **new** ExceptionLogger(**new** DbLogger());
12. \_exceptionLogger.LogException(ex);
13. }
14. **catch**(SqlException ex)
15. {
16. \_exceptionLogger = **new** ExceptionLogger(**new** EventLogger());
17. \_exceptionLogger.LogException(ex);
18. }
19. **catch**(Exception ex)
20. {
21. \_exceptionLogger = **new** ExceptionLogger(**new** FileLogger());
22. \_exceptionLogger.LogException(ex);
23. }
24. }
25. }

Looks good. But we introduced the dependency here in the DataExporter class's catch blocks. Yeah, someone must take the responsibility to provide the necessary objects to the ExceptionLogger to get the work done.

Let me explain it with a real-world example. Suppose we want to have a wooden chair with specific measurements and the kind of wood to be used to make that chair. Then we can't leave the decision making on measurements and the wood to the carpenter. Here his job is to make a chair based on our requirements with his tools and we provide the specifications to him to make a good chair.

So what is the benefit we get by the design? Yes, we definitely have a benefit with it. We need to modify both the DataExporter class and ExceptionLogger class whenever we need to introduce a new logging functionality. But in the updated design we need to add only another catch block for the new exception logging feature. Coupling is not inherently evil. If you don't have some amount of coupling, your software will not do anything for you. The only thing we need to do is understand the system, requirements, and environment properly and find areas where DIP should be followed.