**SOLID Principles In C# With Examples**

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).

Basics of SOLID design principles using C# and . NET.

The reasons behind most unsuccessful applications / Solutions

Intro to SOLID principles

1. SRP
2. OCP
3. LSP
4. ISP
5. DIP

The reason behind most unsuccessful applications

Developers build 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 a 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? It is the design of the application.

The following are the design flaws that cause damage to 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 depend 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 (MVC, 3-tier, Layered, MVP, MVVP, and so on).
2. Following Design Principles.
3. Choosing the correct Design Patterns to build the software based on its specifications. We go through the Design Principles first and will cover the rest soon.

**Introduction to SOLID principles**

SOLID principles are the design principles that enable us to manage several 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 business needs properly.

SOLID is an acronym for 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. 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, 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 a 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. However, 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.

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](mailto:mysite@nowhere.com)", email) { Subject="HEllo foo" });

}

public virtual bool ValidateEmail(string email)

{

return email.Contains("@");

}

public bool SendEmail(MailMessage message)

{

\_smtpClient.Send(message);

}

}

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

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](mailto: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**

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



Here "Open for extension" means we must design our module/class so 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. OK, let's dive into an example.

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

public class Rectangle{

public double Height {get;set;}

public double Wight {get;set; }

}

Our app needs 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.

public class AreaCalculator {

public double TotalArea(Rectangle[] arrRectangles)

{

double area;

foreach(var objRectangle in arrRectangles)

{

area += objRectangle.Height \* objRectangle.Width;

}

return area;

}

}

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

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;

}

}

Wow. We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate its area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape, we must 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:

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 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, "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, a derived class must be substitutable for its base class.

This principle is just an extension of the Open Closed Principle, and we must ensure that newly 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 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 keeps the text of a group of SQL files along with the SqlFile Class.

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();

}

}

}

C#

Copy

OK. We are done with our part. The functionality looks good for now. However, after some time, 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.

OK. We need to modify "SqlFileManager" by adding one condition to the loop to avoid an exception. 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.

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, skip calling it's

// SaveText() method to skip the exception.

if(! objFile is ReadOnlySqlFile)

objFile.SaveText();

}

}

}

C#

Copy

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 the 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.

public interface IReadableSqlFile

{

string LoadText();

}

public interface IWritableSqlFile

{

void SaveText();

}

C#

Copy

Now we implement IReadableSqlFile through the ReadOnlySqlFile class that reads only the text from read-only files. We implement both IWritableSqlFile and IReadableSqlFile in a SqlFile class by which we can read and write files.

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 \*/

}

}

C#

Copy

Now the design of the SqlFileManager class becomes like this.

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();

}

}

}

C#

Copy

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 this case, SqlFile instances. So now we can say our design is following the LSP. And we fixed the problem using the Interface segregation principle (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 serving one submodule.".

We can define it in another way. An interface should be more closely related to the code that uses it than the 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 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 not all implementers can use. 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.

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.

}

}

C#

Copy

OK. The design looks fine for now. However, 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?

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");

}

}

C#

Copy

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 must 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, managers can only divide and assign tasks, 2. TeamLead can divide and assign the jobs and work on them, 3. We need to divide the responsibilities by segregating the ILead interface for the programmer that can only work on tasks—an interface that provides a contract for WorkOnTask().

public interface IProgrammer

{

void WorkOnTask();

}

C#

Copy

An interface that provides contracts to manage the tasks:

public interface ILead

{

void AssignTask();

void CreateSubTask();

}

C#

Copy

Then the implementation becomes.

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.

}

}

C#

Copy

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

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.

}

}

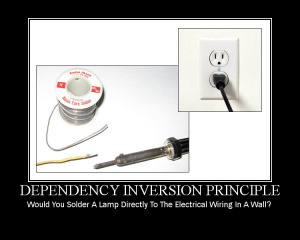
C#

Copy

Wow. Here we separated responsibilities/purposes, distributed them on multiple interfaces, and provided good abstraction.

**D: Dependency Inversion Principle**

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. First, both should depend upon abstractions. Secondly, abstractions should not rely upon details. Finally, 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 write information to databases or pass messages to the operating system or services.

A high-level module/class that depends 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. So we must keep these high-level and low-level modules/classes loosely coupled as much as possible. 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 classes provide the functionality to log a stack trace into a file.

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 format.

....

....

return strMessage;

}

}

C#

Copy

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

public class DataExporter

{

public void ExportDataFromFile()

{

try {

//code to export data from files to the database.

}

catch(Exception ex)

{

new ExceptionLogger().LogIntoFile(ex);

}

}

}

C#

Copy

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... OK, 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.

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 format.

....

....

return strMessage;

}

}

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);

}

}

}

C#

Copy

Looks fine for now. But whenever the client wants to introduce a new logger, we must alter ExceptionLogger by adding a new method. Suppose we continue doing this after some time. In that case, we will see a fat ExceptionLogger class with a large set of practices 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 directly depending on the low-level classes.

public interface ILogger

{

void LogMessage(string aString);

}

C#

Copy

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

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.

}

}

C#

Copy

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.

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 format.

....

....

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);

}

}

}

C#

Copy

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 no longer need to change the ExceptionLogger's code for the new logging functionality. We must create a new logging class that implements the ILogger interface and adds another catch block to the DataExporter class's ExportDataFromFile method.

public class EventLogger: ILogger

{

public void LogMessage(string aMessage)

{

//Code to write a message in system's event viewer.

}

}

C#

Copy

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

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);

}

}

}

C#

Copy

Looks good. But we introduced the dependency here in the DataExporter class's catch blocks. So, someone must be responsible for providing 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 measures 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 from the design? Yes, we definitely have benefited from it. We need to modify the DataExporter and ExceptionLogger classes 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. We only need to properly understand the system, requirements, and environment and find areas where DIP should be followed. Coupling is not inherently evil. If you don't have some amount of coupling, your software will not do anything for you.

Conclusion

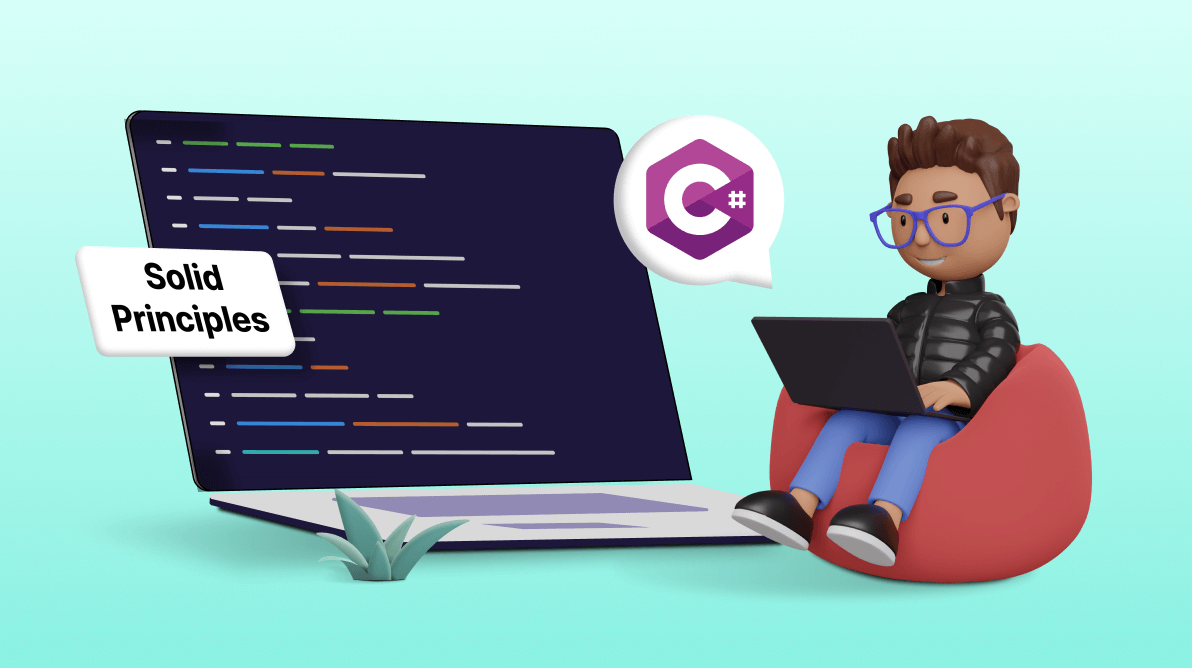
Great, we have gone through all five SOLID principles successfully. And we can conclude that using these principles, we can build an application with tidy, readable, and easily maintainable code.

Here you may have some doubts. Yes, about the quantity of code. Because of these principles, the code might become larger in our applications. But my dear friends, you need to compare it with the quality we get by following these principles. Hmm, but anyway, 27 lines are much fewer than 200 lines.

This is my little effort to share the uses of SOLID principles. I hope you enjoyed this article.

**Mastering SOLID Principles in C#: A Practical Guide**

[**[](https://www.syncfusion.com/blogs/author/a-yohan-malshika)A. Yohan Malshika**](https://www.syncfusion.com/blogs/author/a-yohan-malshika)**13 min readSep 10, 2024Updated**[**2 Comments**](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#comments-box)



**TL;DR:***SOLID principles ensure clean, modular code for maintainability, scalability, and flexibility. They guide developers to build robust systems through Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion principles.*

The SOLID principles are design principles that help create maintainable, scalable, and flexible software systems. These principles, coined by **Robert C. Martin**, provide guidelines for writing clean, modular, and loosely coupled code.

In this blog, you will explore each SOLID principle and discuss how to implement them in C# with code examples.

**What are the SOLID principles?**

* [Single Responsibility Principle](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#Single)
* [Open-Closed Principle](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#Open)
* [Liskov Substitution Principle](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#Liskov)
* [Interface Segregation Principle](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#Interface)
* [Dependency Inversion Principle](https://www.syncfusion.com/blogs/post/mastering-solid-principles-csharp#Dependency)

**Why should we use the SOLID design principles?**

SOLID principles serve as foundational guidelines in software development, aiming to create robust and adaptable systems. By adhering to SOLID principles, developers foster:

* **Code maintainability:**Promote clean, organized, and maintainable codebases. By adhering to these principles, developers ensure that each component of their code has a clear purpose and well-defined responsibilities. This clarity simplifies maintenance tasks, making it easier to understand, update, and modify code without causing unintended side effects elsewhere.
* **Future scalability:**Allows for more adaptable and extensible software systems. Developers should design code that can be added to but not modified, allowing others to introduce new features or functionalities without altering existing code. This scalability is essential for accommodating future changes and expanding the system’s capabilities without compromising stability.
* **Flexibility in development:**Encourages flexibility in development by providing a framework that supports iterative and incremental changes. This enables developers to introduce new functionalities, fix issues, or refactor codebases more efficiently, minimizing risks associated with unintended consequences or regressions.
* **Improved collaboration:**Encourages a modular and structured approach to development. This organization enhances collaboration among team members as it leads to well-defined interfaces and clear responsibilities, making it easier for different team members to work on distinct parts of the codebase simultaneously.

Incorporating the SOLID design principles in software development leads to a more robust, adaptable, and maintainable codebase. Let’s discuss each principle with code examples.

**Single Responsibility Principle (SRP)**

The Single Responsibility Principle (SRP) promotes clean, maintainable, and scalable software design. It states that a class should change for only one reason, meaning it should have a single responsibility.

Let’s consider a user creation process that involves validating and saving user data to a database.

Refer to the following code example to understand how SRP can be violated through combined handling validation and persistence.

public class UserCreator

{

public void CreateUser(string username, string email, string password)

{

// Validation logic

if (!ValidateEmail(email))

{

throw new ArgumentException("Invalid email format.");

}

// Business rules

// Database persistence

SaveUserToDatabase(username, email, password);

}

private bool ValidateEmail(string email)

{

// Validation logic

}

private void SaveUserToDatabase(string username, string email, string password)

{

// Database persistence logic

}

}

**Issue**

In the previous code, the **UserCreator** class violates the SRP by combining multiple responsibilities, such as validation and database persistence. This can lead to a tightly coupled class, making it difficult to test and prone to unnecessary modifications.

**Solution**

To address this issue, we can apply SRP by refactoring the code to separate these responsibilities into individual classes.

Refer to the following refactored code example.

public class UserValidator

{

public bool ValidateEmail(string email)

{

// Validation logic

}

}

public class UserRepository

{

public void SaveUser(string username, string email, string password)

{

// Database persistence logic

}

}

public class UserCreator

{

private readonly UserValidator \_validator;

private readonly UserRepository \_repository;

public UserCreator(UserValidator validator, UserRepository repository)

{

\_validator = validator;

\_repository = repository;

}

public void CreateUser(string username, string email, string password)

{

if (!\_validator.ValidateEmail(email))

{

throw new ArgumentException("Invalid email format.");

}

// Business rules

\_repository.SaveUser(username, email, password);

}

}

After refactoring, the code demonstrates the implementation of SRP through the separation of responsibilities into three classes:

* **UserValidator:** Validates the user’s email format.
* **UserRepository:** Handles saving the user’s data to the database.
* **UserCreator:** Coordinates the user creation process, leveraging the validator and repository classes for their specific responsibilities.

**Benefits**

By separating the concerns, we achieve a more maintainable and testable codebase. Each class has a single responsibility, allowing for more straightforward modification and extension in the future.

**Open/Closed Principle (OCP)**

The Open-Closed Principle (OCP) says that software entities should be open for extension but closed for modification. It allows for adding new functionality without modifying existing code.

Let’s consider a scenario where a file-exporting service initially supports exporting data to CSV files.

Refer to the following code example to understand how OCP can be violated and how to correct it using C#.

public class FileExporter

{

public void ExportToCsv(string filePath, DataTable data)

{

// Code to export data to a CSV file.

}

}

**Issue**

In this example, the **FileExporter** class directly implements the functionality for exporting data to CSV files. However, if we later want to support exporting data to other file formats like Excel or JSON, modifying the **FileExporter** class would violate the OCP.

**Solution**

To use the OCP, we must design our file-exporting service domain to be open for extension.

Refer to the following refactored code example.

public abstract class FileExporter

{

public abstract void Export(string filePath, DataTable data);

}

public class CsvFileExporter : FileExporter

{

public override void Export(string filePath, DataTable data)

{

// Code logic to export data to a CSV file.

}

}

public class ExcelFileExporter : FileExporter

{

public override void Export(string filePath, DataTable data)

{

// Code logic to export data to an Excel file.

}

}

public class JsonFileExporter : FileExporter

{

public override void Export(string filePath, DataTable data)

{

// Code logic to export data to a JSON file.

}

}

In the improved implementation, we introduce an abstract **FileExporter** class that defines the common behavior for all file export operations. Each specific file exporter (**CsvFileExporter**, **ExcelFileExporter**, and **JsonFileExporter**) inherits from the **FileExporter** class and implements the **Export** method according to the particular file format export logic.

Applying the OCP allows for adding new file exporters without modifying old ones, making it easier to add new features by introducing subclasses of the **FileExporter** base class.

**Benefits**

This approach enhances code flexibility, reusability, and maintainability. Your code can seamlessly handle new requirements and changes without introducing bugs or disrupting the existing functionality.

**Liskov Substitution Principle (LSP)**

The Liskov Substitution Principle (LSP) is a concept that guarantees the smooth substitution of objects of derived classes for objects of their base classes. Its fundamental rule asserts that objects of the base class must be interchangeable with objects of any of its derived classes, without impacting the accuracy of the program.

Refer to the following code example to understand how LSP can be violated and how to correct it using C#.

public abstract class Vehicle

{

public abstract void StartEngine();

public abstract void StopEngine();

}

public class Car : Vehicle

{

public override void StartEngine()

{

Console.WriteLine("Starting the car engine.");

// Code to start the car engine

}

public override void StopEngine()

{

Console.WriteLine("Stopping the car engine.");

// Code to stop the car engine

}

}

public class ElectricCar : Vehicle

{

public override void StartEngine()

{

throw new InvalidOperationException("Electric cars do not have engines.");

}

public override void StopEngine()

{

throw new InvalidOperationException("Electric cars do not have engines.");

}

}

**Issue**

In this example, we have a **Vehicle** class that represents a generic vehicle. It has abstract methods, **StartEngine()** and StopEngine(), for starting and stopping the engine. We also have a **Car** class that inherits from **Vehicle** and provides the necessary implementation for the engine-related methods.

However, when we introduce a new type of vehicle, such as an **ElectricCar**, which doesn’t have an engine, we encounter a violation of the LSP. In this case, attempting to call the **StartEngine()** or **StopEngine()** methods on an ElectricCar object would result in exceptions because electric cars do not have engines.

Vehicle car = new Car();

car.StartEngine(); // Outputs "Starting the car engine."

Vehicle electricCar = new ElectricCar();

electricCar.StartEngine(); // Throws InvalidOperationException

**Solution**

To address this violation, we need to ensure the correct substitution of objects. One approach is to introduce an interface called **IEnginePowered** that represents vehicles with engines.

Refer to the following refactored code example.

public abstract class Vehicle

{

// Common vehicle behavior and properties.

}

public interface IEnginePowered

{

void StartEngine();

void StopEngine();

}

public class Car : Vehicle, IEnginePowered

{

public void StartEngine()

{

Console.WriteLine("Starting the car engine.");

// Code to start the car engine.

}

public void StopEngine()

{

Console.WriteLine("Stopping the car engine.");

// Code to stop the car engine.

}

}

public class ElectricCar : Vehicle

{

// Specific behavior for electric cars.

}

In this corrected design, the **Car** class implements the **IEnginePowered** interface along with the **Vehicle** class. The **Vehicle** class will include common vehicle properties and behavior for both. This design provides the necessary implementation for the engine-related methods. Also, the **ElectricCar** class does not implement the **IEnginePowered** interface because it does not have an engine.

IEnginePowered car = new Car();

car.StartEngine(); // Outputs "Starting the car engine."

Vehicle electricCar = new ElectricCar();

// electricCar.StartEngine(); // This line won't compile because ElectricCar does not implement IEnginePowered

We can substitute objects of the **Car** or **ElectricCar** class where instances of the **IEnginePowered** are expected. The ElectricCar class does not need to implement engine-related methods.

**Benefits**

Using the LSP, we ensured that the program remained accurate and consistent when substituting objects of derived classes for objects of their base class.

**Interface Segregation Principle (ISP)**

The Interface Segregation Principle (ISP) says to create smaller, specialized interfaces that cater to clients’ specific needs. It discourages large interfaces that include unnecessary methods, so that clients are not burdened with functionality they don’t require.

Refer to the following example to understand how ISP can be violated and how to correct it using C#.

public interface IOrder

{

void PlaceOrder();

void CancelOrder();

void UpdateOrder();

void CalculateTotal();

void GenerateInvoice();

void SendConfirmationEmail();

void PrintLabel();

}

public class OnlineOrder : IOrder

{

// Implementation of all methods.

}

public class InStoreOrder : IOrder

{

// Implementation of all methods.

}

**Issue**

In the previous example, we have an **IOrder** interface that contains methods for placing an order, canceling an order, updating an order, calculating the total, generating an invoice, sending a confirmation email, and printing a label.

However, not all client classes implementing this interface require or use all these methods. This violates ISP, since clients are forced to depend on methods they don’t need.

By following the ISP, we can refactor the code by segregating the interface into smaller, more focused interfaces.

public interface IOrder

{

void PlaceOrder();

void CancelOrder();

void UpdateOrder();

}

public interface IOrderProcessing

{

void CalculateTotal();

}

public interface IInvoiceGenerator

{

void GenerateInvoice();

}

public interface IEmailSender

{

void SendConfirmationEmail();

}

public interface ILabelPrinter

{

void PrintLabel();

}

// Implement only the necessary interfaces in client classes.

public class OnlineOrder : IOrder, IOrderProcessing, IInvoiceGenerator, IEmailSender

{

// Implementation of required methods.

}

public class InStoreOrder : IOrder, IOrderProcessing, ILabelPrinter

{

// Implementation of required methods.

}

**Solution**

By segregating the interfaces, we now have smaller, more focused interfaces that clients can choose to implement based on their specific needs. This approach eliminates unnecessary dependencies and allows for better extensibility and maintainability. Clients can implement only the interfaces they require, resulting in cleaner code that is easier to understand, test, and modify.

**Benefits**

Using the ISP in C# enables us to create interfaces tailored to specific client requirements. By avoiding the violation of ISP, we can build more flexible, modular, and maintainable code. Breaking down large interfaces into smaller, cohesive ones reduces coupling and improves code organization.

**Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle (DIP) focuses on decoupling high-level modules from low-level modules by introducing an abstraction layer, with the use of interfaces or abstract classes and reducing direct dependencies between classes.

Refer to the following example where a **UserController** class depends directly on a **Database** class for data storage.

public class UserController

{

private Database database;

public UserController()

{

this.database = new Database();

}

// ...

}

**Issue**

In the previous example, the **UserController** tightly couples with the concrete **Database** class, creating a direct dependency. If we decide to alter the database implementation or introduce a new storage mechanism, we will need to modify the **UserController** class, which violates the Open-Closed Principle.

**Solution**

To address this issue and adhere to the DIP, we must invert the dependencies by introducing an abstraction that both high-level and low-level modules depend on. Typically, this abstraction is defined using an interface or an abstract class.

Let’s modify the previous example to align with the DIP.

Refer to the following refactored code example.

public interface IDataStorage

{

// Define the contract for data storage operations.

void SaveData(string data);

string RetrieveData();

}

public class Database : IDataStorage

{

public void SaveData(string data)

{

// Implementation for saving data to a database.

}

public string RetrieveData()

{

// Implementation for retrieving data from a database.

}

}

public class UserController

{

private IDataStorage dataStorage;

public UserController(IDataStorage dataStorage)

{

this.dataStorage = dataStorage;

}

// ...

}

In this updated version, we introduce the **IDataStorage** interface that defines the contract for data storage operations. The **Database** class implements this interface, providing a concrete implementation. Consequently, the **UserController** class now relies on the **IDataStorage** interface rather than the concrete **Database** class, resulting in it being decoupled from specific storage mechanisms.

This inversion of dependencies facilitates easier extensibility and maintenance. We can introduce new storage implementations, such as a file system or cloud storage, by simply creating new classes that implement the **IDataStorage** interface, without modifying the **UserController** or any other high-level modules.

**Benefits**

By applying the DIP, we achieve a more flexible and modular design, enabling us to evolve and adapt our systems more easily over time.

**Advantages of using the SOLID principles**

SOLID principles offer a set of guidelines that significantly impact the quality, maintainability, and scalability of software systems. Embracing these principles brings several key advantages to software development:

* **Enhanced maintainability:** Following SOLID principles results in code that is easier to understand, update, and maintain. The principles encourage clear code organization, making it simpler to identify and modify specific functionalities without impacting the rest of the system. This ultimately reduces the chances of introducing bugs during maintenance.
* **Improved extensibility:** By adhering to SOLID principles, code becomes more flexible and open for extension without requiring modifications to existing, functional code. This extensibility is particularly beneficial when adding new features or accommodating changing requirements, as it minimizes the risk of unintentionally breaking existing functionalities.
* **Facilitates reusability:** SOLID principles promote the creation of modular, loosely coupled components. This modularity allows for greater code reuse across different parts of the system or in entirely new projects, leading to more efficient development processes.
* **Supports testability:** Codes that follow SOLID principles tend to be more testable. The principles advocate for smaller, more focused units of code with clear responsibilities, making it easier to write unit tests and ensure proper functionality.
* **Reduces technical debt:** Implementing SOLID principles from the start reduces the accumulation of technical debt. It encourages clean code practices, preventing issues such as code smells, tight coupling, and unnecessary complexity. Consequently, it saves time and effort that might otherwise be spent refactoring or fixing problematic code later in the development lifecycle.

**GitHub reference**

For more details, refer to the project on [SOLID Principles in C# GitHub demo](https://github.com/SyncfusionExamples/solid-principle-in-Csharp/).

**Conclusion**

Thanks for reading! In this article, we explored each SOLID principle and discussed how to implement them in C# with code examples. We examined how separating responsibilities, allowing for extension without modification, ensuring correct substitution, segregating interfaces, and inverting dependencies can lead to better software design.

By following the SOLID principles, developers can write more modular, reusable, and easier-to-understand code. These principles contribute to creating robust and adaptable software systems capable of evolving and effectively handling changing requirements.

For existing customers, the new version of Essential Studio is available for download from the [License and Downloads](https://www.syncfusion.com/account/downloads) page. If you are not yet a Syncfusion customer, you can try our 30-day [free trial](https://www.syncfusion.com/downloads) to check out our available features.

For questions, you can contact us through our [support forum](https://www.syncfusion.com/forums), [support portal](https://support.syncfusion.com/), or [feedback portal](https://www.syncfusion.com/feedback/). We are always happy to assist you!