Open-Closed Principle in C#

**Open-Closed Principle in C# with Examples**

In this article, I will discuss the **Open-Closed Principle in C#** with Examples. Please read our previous article before proceeding to this article, where we discussed the [**Single Responsibility Principle in C#**](https://dotnettutorials.net/lesson/single-responsibility-principle/) with Real-Time Examples. The letter **O** in S**O**LID stands for the **Open-Closed Principle**, or **OCP**. As part of this article, we will discuss the following pointers in detail.

1. **What is the Open-Closed Principle in C#?**
2. **Implementation Guidelines for the Open-Closed Principle in C#**
3. **What Problems will you get if you are not following the Open-Closed Principle?**
4. **Example without using the Open-Closed Principle.**
5. **Example using the Open-Closed Principle in C#.**
6. **Advantages of Open-Closed Principle in C#.**

**What is the Open-Closed Principle in C#?**

The Open-Closed Principle (OCP) is another SOLID principle in object-oriented design formulated by Bertrand Meyer. It states that software entities (such as modules, classes, functions, etc.) should be open for extension but closed for modification. This principle promotes the idea that you should be able to add new functionality to a component without altering its existing code.

Let us understand the above definition in simple words. Here, we need to understand two things. The first is **Open for Extension**, and the second is **Closed for Modification**. The Open for Extension means we need to design the software **modules/classes/functions** in a way that the new responsibilities or functionalities can be added easily when new requirements come. On the other hand, Closed for Modification means we should not modify the class/module/function until we find some bugs. Why it is important, let us understand.

We have already developed a Class/Module/Function, which has undergone the unit testing phase. So we should not have to change this as it affects the existing functionalities. In simple words, we can say that we should develop one Class/Module/Function in such a way that it should allow its behavior to be extended without altering its source code. That means we should not edit the code of a method (until we find some bugs); instead, we should use polymorphism or other techniques to add new functionality by writing new code.

**Implementation Guidelines for Open-Closed Principle (OCP) using C#**

1. The easiest way to implement the Open-Closed Principle in C# is to add new functionalities by creating new derived classes, which should be inherited from the original base class.
2. Another way is to allow the client to access the original class with an abstract interface.

So, when a requirement change or any new requirement comes at any given time, instead of touching the existing functionality, it’s always better and suggested to create new derived classes and leave the original class implementation as it is. Let us understand this with an example. First, we will see the example of not following the Open-Close Principle. We will understand the problems if we are not following the OCP. Finally, we will see the same example by following the Open-Close Principle using C# Language so that you will get a better idea of this Principle.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=8adce4dc-49f5-4cab-7de7-ef70c01d7687&d_id=118560&imp_id=3502918184007359&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**Example to Understand Open-Closed Principle in C#.**

Please have a look at the following class. As you can see in the image below, we have created the GetInvoiceDiscount() method within the Invoice class. As part of that GetInvoiceDiscount() method, we calculate the final amount based on the Invoice type. Currently, we have two Invoice Types, i.e., Final Invoice and Proposed Invoice. So, we have implemented the logic using the if-else condition.

Tomorrow, if one more Invoice Type comes into the picture, we need to modify the GetInvoiceDiscount() method logic by adding another if block to the source code. Changing the source code for the new requirement violates the Open-Closed Principle in C#.

**Example Without Using Open-Closed Principle in C#:**

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**public** **class** Invoice

**{**

**public** **double** GetInvoiceDiscount**(double** amount, InvoiceType invoiceType**)**

**{**

**double** finalAmount = 0;

**if** **(**invoiceType == InvoiceType.FinalInvoice**)**

**{**

finalAmount = amount - 100;

**}**

**else** **if** **(**invoiceType == InvoiceType.ProposedInvoice**)**

**{**

finalAmount = amount - 50;

**}**

**return** finalAmount;

**}**

**}**

**public** enum InvoiceType

**{**

FinalInvoice,

ProposedInvoice

**}**;

**}**

The problem with the above example is that if we want to add another new invoice type, we need to add one more “else if” condition in the same “**GetInvoiceDiscount**” method. In other words, we need to modify the Invoice class GetInvoiceDiscount Method. Suppose we are changing the Invoice class GetInvoiceDiscount Method again and again. In that case, we need to ensure that the previous and new functionalities are working properly by testing the existing functionalities again. This is because we need to ensure that the existing clients referencing this class are working properly as expected.

**Problems of Not Following the Open-Closed Principle in C#:**

So, if you are not following the Open-Closed Principle during the application development process, then you may end up with your application development with the following problems

1. If you allow a class or function to add new logic, then you, as a developer, need to test the entire application’s functionality, including the old and new functionalities.
2. As a developer, it is also your responsibility to inform the QA (Quality Assurance) team about the changes in advance so that they can prepare for regression testing and new feature testing.
3. If you are not following the Open-Closed Principle, it also breaks the Single Responsibility Principle, as the class or module will perform multiple responsibilities.
4. Implementing all the functionalities in a single class makes maintenance very difficult.

Because of the above key points, we must follow the open-closed principle in C# while developing the application.

**Open-Closed Principle in C#**

We should go for **EXTENSION** instead of **MODIFICATION** as per the Open-Closed principle**.**If you want to follow the Open-Closed Principle in the above example, we need to add a new class when a new invoice type needs to be added. As a result, the current functionalities that are already implemented are going to be unchanged. The advantage is that we only need to test and check the new classes.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=8adce4dc-49f5-4cab-7de7-ef70c01d7687&d_id=118560&imp_id=1537328122054447&c_id=1084&l_id=10016&url=https%3A%2F%2Fwww.amazonconservation.org%2Ftake-action%2Fdonate%2F&ffid=1&co=IN)**

**Example following Open-Closed Principle in C#**

The following example shows the Open Closed Principle (OCP) in C#. As you can see in the code below, we have created three classes: **FinalInvoice**, **ProposedInvoice**, and **RecurringInvoice**. All these three classes are inherited from the base class **Invoice,** and if they want, they can override the **GetInvoiceDiscount()** method, which is declared as Virtual in the Base Invoice class.

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**public** **class** Invoice

**{**

**public** **virtual** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** amount - 10;

**}**

**}**

**public** **class** FinalInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 50;

**}**

**}**

**public** **class** ProposedInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 40;

**}**

**}**

**public** **class** RecurringInvoice : Invoice

**{**

**public** **override** **double** GetInvoiceDiscount**(double** amount**)**

**{**

**return** **base**.GetInvoiceDiscount**(**amount**)** - 30;

**}**

**}**

**}**

Tomorrow, if another Invoice Type needs to be added, we need to create a new class by inheriting it from the Invoice class, and if needed, then we need to override the GetInvoiceDiscount() method. The point that you need to keep focus on is we are not changing the code of the Invoice class. Now, the Invoice class is **Closed for Modification**. But it is **Open for Extension** as it allows the creation of new classes deriving from the base Invoice class, which follows the Open-Closed Principle in C#.

**Testing the Functionality:**

Now, modify the Main method of the Program class as shown below to test the application and see whether the different Invoice types are working as expected, as per our business requirement.

**using** *System;*

**namespace** *SOLID\_PRINCIPLES.OCP*

**{**

**class** Program

**{**

**static** **void** Main**(string[]** args**)**

**{**

Console.WriteLine**(**"Invoice Amount: 10000"**)**;

Invoice FInvoice = new FinalInvoice**()**;

**double** FInvoiceAmount = FInvoice.GetInvoiceDiscount**(**10000**)**;

Console.WriteLine**(**$"Final Invoive : {FInvoiceAmount}"**)**;

Invoice PInvoice = new ProposedInvoice**()**;

**double** PInvoiceAmount = PInvoice.GetInvoiceDiscount**(**10000**)**;

Console.WriteLine**(**$"Proposed Invoive : {PInvoiceAmount}"**)**;

Invoice RInvoice = new RecurringInvoice**()**;

**double** RInvoiceAmount = RInvoice.GetInvoiceDiscount**(**10000**)**;

Console.WriteLine**(**$"Recurring Invoive : {RInvoiceAmount}"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

When you run the above application, you will get the following output as expected.

**Advantages of Open-Closed Principle in C#**

The following are the advantages of applying the Open-Closed Principle in C#:

* **Minimized Risk of Bugs:** By following OCP, existing code remains unaltered when new features are added. This significantly reduces the risk of introducing bugs into already tested and proven code, ensuring stability and reliability.
* **Increased Reusability:** Components designed to be extended (but not modified) can be reused across different parts of the application or even in different projects.
* **Improved Maintainability:** Since the core functionality of existing modules does not change, maintaining the application becomes simpler. Modifications are made by adding new code rather than changing the old one, which helps in maintaining a clean and organized codebase.
* **Use of OOPs Concept Effectively:** Applying OCP often leads to the use of interfaces and abstract classes, promoting the use of inheritance and polymorphism (OOP Concepts). This results in a more flexible and dynamic architecture, where behavior can be changed dynamically at runtime.

In the next article, I will discuss [**Multiple** **Real-Time Examples of the Open-Closed** **Principle (OCP) in C#**](https://dotnettutorials.net/lesson/real-time-examples-of-open-closed-principle-in-csharp/). In this article, I explain the **Open-Closed Principle in C#** with Examples. I hope you understand the need and use of the Open-Closed Principle in C# with Examples.