**Single Responsibility Principle in C# with Examples**

In this article, I will discuss the **Single Responsibility Principle in C#** with Examples. Please read our previous article before proceeding to this article, where we discussed the basics of the [**SOLID Design Principle in C#**](https://dotnettutorials.net/lesson/solid-design-principles/). **S**in **S**OLID stands for the **Single Responsibility Principle or** **SRP.** As part of this article, we will discuss the following pointers in detail.

1. **What is the Single Responsibility Principle in C#?**
2. **How can we achieve the Single Responsibility Principle in C#?**
3. **Example without using the Single Responsibility Principle.**
4. **Problems of not following the Single Responsibility Principle.**
5. **Example using the SRP in C#.**
6. **Advantages of using the Single Responsibility Principle.**

**What is the Single Responsibility Principle in C#?**

The Single Responsibility Principle (SRP) is one of the five SOLID principles of object-oriented design and development, which aims to make software more understandable, flexible, and maintainable. The SRP states that a class should have only one reason to change, meaning it should have only one job or responsibility.

So, we need to design the software in a way where everything in a class should be related to a single responsibility. That does not mean your class should contain only one method. You can have multiple methods if they are related to a single responsibility or functionality. So, with the help of SRP in C#, the classes become smaller, cleaner, and, thus, easier to maintain. Before proceeding further and understanding the Single Responsibility Principle, we first need to understand what Responsibility is.

**What is Responsibility?**

An application can have many functionalities (features).

For example, suppose you are developing an e-commerce application. In that case, that application may have many functionalities such as Registering users, providing login functionality, displaying the product list, allowing the user to place an order, Providing Payment Functionality, Shipping the Order, Billing the Order, Logging the Order Information for Auditing and for Security purpose, sending the Order Invoice to the customer, etc. You can consider these functionalities or features as responsibilities. Another point you need to remember is that changing the functionality means changing the class responsible for that functionality.

**How can we apply the Single Responsibility Principle in C#?**

Let us understand how we can apply the Single Responsibility Principle in C# while designing the classes:

* **Identify Responsibilities:** The first step is to identify your class’s responsibilities clearly. Responsibilities are often grouped into categories such as data access, validation, logging, caching, serialization, user registration, authentication, etc.
* **Create Separate Classes:** Once we Identify the responsibilities, we must create separate classes for each responsibility. We should not mix multiple functionalities or responsibilities in a single class. If a class handles more than one responsibility, consider breaking it into multiple classes.
* **Separate Concerns:** Finally, we need to make sure that each class should only perform one specific task.

**Examples to Understand the Single Responsibility Principle using C#:**

Let us understand the Single Responsibility Principle in C# with an Example. Suppose we need to design an Invoice class. As we know, an Invoice class calculates various amounts based on its data. The Invoice class does not know how to retrieve the data or how to format the data for displaying, printing, logging, sending an email, etc.

If we write the database, business logic, and display logic in a single class, our class performs multiple responsibilities. Then, changing one responsibility without breaking the other responsibilities becomes very difficult. So, by mixing multiple responsibilities into a single class, we are getting the following disadvantages,

1. **Difficult to Understand**
2. **Difficult to Test**
3. **Chance of Duplicating the Logic of Other Parts of the Application**

**Example Without using Single Responsibility Principle in C#:**

Let us see the example without following the Single Responsibility Principle (SRP) in C#. We will see the problems if we do not follow the Single Responsibility Principle. Then, we will see how we can overcome the problem using the Single Responsibility Principle so that you will better understand SRP.  Please look at the following diagram we want to implement in our example.

As you can see in the above image, we will create an Invoice class with four functionalities: Adding and Deleting Invoices, Error Logging, and Sending Emails. By putting all four functionalities into a single class or module, we are violating the Single Responsibility Principle in C#. This is because Sending Emails and Error Logging are not part of the Invoice module. The following is the complete code, which is self-explanatory, so please go through the comments.

Advertisements

**using** *System;*

**using** *System.Net.Mail;*

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** Invoice

**{**

**public** **long** InvoiceAmount **{** **get**; **set**; **}**

**public** DateTime InvoiceDate **{** **get**; **set**; **}**

**public** **void** AddInvoice**()**

**{**

**try**

**{**

// Here we need to write the Code for adding invoice

// Once the Invoice has been added, then send the mail

MailMessage mailMessage = new MailMessage**(**"EMailFrom", "EMailTo", "EMailSubject", "EMailBody"**)**;

this.SendInvoiceEmail**(**mailMessage**)**;

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

**public** **void** DeleteInvoice**()**

**{**

**try**

**{**

//Here we need to write the Code for Deleting the already generated invoice

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

**public** **void** SendInvoiceEmail**(**MailMessage mailMessage**)**

**{**

**try**

**{**

// Here we need to write the Code for Email setting and sending the invoice mail

**}**

**catch** **(**Exception ex**)**

**{**

//Error Logging

System.IO.File.WriteAllText**(**@"c:\ErrorLog.txt", ex.ToString**())**;

**}**

**}**

**}**

**}**

With this design, if we want to modify the logging functionality or the sending email functionality, then we need to modify the Invoice class. This violates the Single Responsibility Principle as we are changing the Invoice Class for other functionality. If we make the changes, we must test the logging, email, and invoicing functionality. Now let us discuss how to implement the above functionalities so that it should follow the Single Responsibility Principle using C# Language.

**Example Implementing Single Responsibility Principle in C#**

Let us implement the same example by following SRP. Please have a look at the following diagram.

As shown in the above diagram, we will create three classes. The Invoice Class will implement only invoice-related functionalities. The Logger class will be used only for logging purposes. Similarly, the Email class will handle Email activities. Now, each class has its own responsibilities. As a result, it follows the Single Responsibility Principle (SRP) in C#.

Now, with the above design in place, if you want to modify the email functionality, you must change the Email class only, not the Invoice and Logging class. Similarly, if you want to modify the Invoice functionalities, you must change the Invoice class only, not the Email and Logging class. Let us proceed and see how to implement this using C#.

**Logger.cs**

Add a class file named **Logger.cs** and copy and paste the following code into it. As you can see in the below class, we define all the logging activities, i.e., Info, Debug, and Error. The point that you need to remember is that the class can contain multiple methods as long as all the methods belong to the same responsibility. Here, we are creating an Interface with the name ILogger with three abstract methods (by default, interface methods are abstract). Then, we implement the ILogger interface methods within the Logger class. Info, Debug, and Error will perform different logging activities, putting all these methods within the Logger class.

**using** *System;*

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **interface** ILogger

**{**

**void** Info**(string** info**)**;

**void** Debug**(string** info**)**;

**void** Error**(string** message, Exception ex**)**;

**}**

**public** **class** Logger : ILogger

**{**

**public** Logger**()**

**{**

// here we need to write the Code for initialization

// that is Creating the Log file with necesssary details

**}**

**public** **void** Info**(string** info**)**

**{**

// here we need to write the Code for info information into the ErrorLog text file

**}**

**public** **void** Debug**(string** info**)**

**{**

// here we need to write the Code for Debug information into the ErrorLog text file

**}**

**public** **void** Error**(string** message, Exception ex**)**

**{**

// here we need to write the Code for Error information into the ErrorLog text file

**}**

**}**

**}**

**MailSender.cs**

Next, we need to add another class file named **MailSender.cs** and copy and paste the following code. In the MailSender class, we define the send mail activities. So, here, you can see that we have included whatever properties are required for sending an email along with the SendEmail method. For simplicity, we have not provided the logic to send the email, but in real time, you need to write the logic within the SendEmail method to send an email.

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** MailSender

**{**

**public** **string** EMailFrom **{** **get**; **set**; **}**

**public** **string** EMailTo **{** **get**; **set**; **}**

**public** **string** EMailSubject **{** **get**; **set**; **}**

**public** **string** EMailBody **{** **get**; **set**; **}**

**public** **void** SendEmail**()**

**{**

// Here we need to write the Code for sending the mail

**}**

**}**

**}**

**Modifying Invoice Class:**

Finally, modify the Invoice class as shown below. Within this class, we are only defining the Invoice-related activities. As you can see, the Invoice class delegates the logging activity to the “**Logger**” class. In the same way, it also delegates the Email-sending activity to the “**MailSender**” class. Now, the Invoice class only concentrates on Invoice-related activities.

**using** *System.Net.Mail;*

**using** *System;*

**namespace** *SOLID\_PRINCIPLES.SRP*

**{**

**public** **class** Invoice

**{**

**public** **long** InvAmount **{** **get**; **set**; **}**

**public** DateTime InvDate **{** **get**; **set**; **}**

**private** ILogger fileLogger;

**private** MailSender emailSender;

**public** Invoice**()**

**{**

fileLogger = new Logger**()**;

emailSender = new MailSender**()**;

**}**

**public** **void** AddInvoice**()**

**{**

**try**

**{**

fileLogger.Info**(**"Add method Start"**)**;

// Here we need to write the Code for adding invoice

// Once the Invoice has been added, then send the mail

emailSender.EMailFrom = "emailfrom@xyz.com";

emailSender.EMailTo = "emailto@xyz.com";

emailSender.EMailSubject = "Single Responsibility Princile";

emailSender.EMailBody = "A class should have only one reason to change";

emailSender.SendEmail**()**;

**}**

**catch** **(**Exception ex**)**

**{**

fileLogger.Error**(**"Error Occurred while Generating Invoice", ex**)**;

**}**

**}**

**public** **void** DeleteInvoice**()**

**{**

**try**

**{**

//Here we need to write the Code for Deleting the already generated invoice

fileLogger.Info**(**"Delete Invoice Start at @" + DateTime.Now**)**;

**}**

**catch** **(**Exception ex**)**

**{**

fileLogger.Error**(**"Error Occurred while Deleting Invoice", ex**)**;

**}**

**}**

**}**

**}**

We must design the application following the Single Responsibility Principle in C#. Now, each class has its own responsibilities. That means each software module or class should have only one reason for changing.

**Advantages of SRP:**

The following are the advantages of applying the Single Responsibility Principle in C#:

* **Easier to Understand:** Classes with a single responsibility are typically smaller and more focused, which makes them easier to understand. Each class has a clear purpose so developers can quickly understand what it does.
* **Easier to Modify:** When classes are designed with just one responsibility, changes in the system’s requirements affect fewer components. This makes it easier to update and maintain the code since changes to one part of the system are less likely to affect other parts.
* **Easier to Test:** SRP leads to smaller classes, which are generally easier to test. Each test can focus on a single functionality, which reduces the complexity of test cases. This also makes unit tests simpler to write and understand.
* **Increased Reusability:** Classes responsible for only one functionality are more likely to be reused in other parts of the application or even in different projects.
* **Better Organization:** SRP helps organize the codebase better. Each class and module will be focused on a specific aspect of the application, which is important in modern application development, like Microservices, where each service is focused on a single aspect of the business functionality.

In the next article, I will discuss [**Multiple Real-Time Examples of the Single Responsibility Principle (SRP) in C#**](https://dotnettutorials.net/lesson/real-time-examples-of-single-responsibility-principle-in-csharp/). In this article, I explain the **Single Responsibility Principle in C#** with Examples. I hope you enjoy this article’s Single Responsibility Principle in C# with Examples.