FIT - Universidad Católica del Uruguay

Single Responsibility Principle (SRP)

Hasta ahora vimos un criterio para asignar responsabilidades a clases: la clase que es experta en la información necesaria para cumplir con una responsabilidad debe tener esa responsabilidad. A la hora de asignar responsabilidades hay que tener en cuenta que las responsabilidades asignadas a una clase tienen que estar relacionadas con una sola funcionalidad de la aplicación, y de esto se trata el primer principio SOLID: el principio de responsabilidad única, o SRP por sus siglas en inglés.

Enunciado

El principio de responsabilidad única establece que cada clase debe tener responsabilidad sobre una parte de la funcionalidad proporcionada por el software, y que la responsabilidad debe estar completamente encapsulada por la clase.

Todos los métodos y atributos de la clase deben estar estrechamente alineados con esa responsabilidad. El princpio se expresa como:

> Una clase debe tener solo una razón para cambiar.

Ejemplo

La clase `SaleTicket` tiene responsabilidades sobre una venta, tales como conocer su fecha y calcular el total, pero también tiene la responsabilidad de imprimir el ticket.

SaleTicket

Conocer fecha y hora Conocer una o más líneas de ítems vendidos Imprimir el ticket Calcular el total	TicketLineItem

Aunque imprimir el ticket definitivamente necesita información que está en la clase `SaleTicket`, si en lugar de imprimir en la consola, como sucede en la versión actual, hubiese que imprimir en una impresora, por ejemplo, la clase `SaleTicket` debería cambiar. Pero también debería cambiar si incluyéramos descuentos, por ejemplo. Entonces existe más de una razón por la cual la clase `SaleTicket` debe cambiar, lo que viola el principio SRP. Podemos separar la responsabilidad de imprimir el ticket a una nueva clase `ConsolePrinter`. Esta clase debe colaborar con la clase `SaleTicket`, que le provee el texto a imprimir, lo cual implica cambiar las responsabilidades para la clase `SaleTicket`:

ConsolePrinter	
Imprimir el ticket en la consola	SaleTicket

La responsabilidad que cambiamos en la clase `SaleTicket` está marcada de negrita:

SaleTicket	
Conocer fecha y hora Conocer una o más líneas de ítems vendidos Calcular el total **Armar el texto a imprimir**	TicketLineItem

Con este nuevo diseño podemos tener múltiples formas de imprimir; por ejemplo, para imprimir la venta en una impresora de rollo de papel, podríamos tener una clase `PaperRollPrinter`, que podemos implementar sin tener que modificar ninguna de nuestras clases existentes -excepto, claro está, el método que use esta nueva clase `PaperRollPrinter` -.

La nueva versión de la clase `SaleTicket` y la nueva clase `ConsolePrinter` quedan en C# así -los ... representan el código que ya apareció antes, como en los casos anteriores-.

```
public class ConsolePrinter
{
    public static void PrintTicket(SaleTicket sale)
    {
        Console.WriteLine(sale.GetTicketText());
    }
}
```

> Ver en repositorio »

```
public class SaleTicket
{
    …

-   public void PrintTicket()
-   {
-       Console.WriteLine($"Fecha: {this.DateTime}");
-       foreach (TicketLineItem item in this.lineItems)
-       {
-           item.PrintTicketLine();
-       }
-
-       Console.WriteLine($"Total: ${this.Total}");
-   }

+    public string GetTicketText()
+    {
+       StringBuilder text = new StringBuilder($"Fecha: {this.DateTime}\n");
+       foreach (TicketLineItem item in this.lineItems)
+       {
+           text.Append(item.GetLineText());
+       }
+
+       text.Append($"Total: ${this.Total}");
+       return text.ToString();
+   }
}
```

> Ver en repositorio »

> Te puede llamar la atención la clase `StringBuilder`. Cuando hay intensa manipulación de texto, como en este ejemplo, es más eficiente utilizar `StringBuilder` que `String`. Esto es porque la clase `String` es inmutable, y por lo tanto, cada concatenación de texto implica crear nuevas instancias, lo cual puede tener un impacto en el desempeño del programa. Por más información mira la documentación sobre el uso de la clase StringBuilder.

El programa principal ahora usa la nueva clase `ConsolePrinter` para imprimir el ticket:

```
public class Program
{
    …

    public static void Main(string[] args)
    {
        PopulateCatalog();

        SaleTicket ticket = new SaleTiket();
        ticket.DateTime = DateTime.Now;
        ticket.AddLineItem(new TicketLineItem(1, ProductAt(0)));
        ticket.AddLineItem(new TicketLineItem(2, ProductAt(1)));
        ticket.AddLineItem(new TicketLineItem(3, ProductAt(2)));
-       ticket.PrintTicket();
+       ConsolePrinter.PrintTicket(ticket);
    }
}
```

> Ver en repositorio »

Beneficios

Vimos con el ejemplo que para agregar una clase `PaperRollPrinter`, podemos comenzar a imprimir la factura en papel sin modificar ninguna de las clases existentes. Este diseño es entonces más fácil de extender, es más robusto a las modificaciones.