FIT - Universidad Católica del Uruguay

Open - Closed Principle (OCP)

Un programa se escribe una vez, pero se modifica muchas veces más; cada modificación puede introducir cambios en la lógica o en los datos del programa, pero también puede introducir errores, y algo que funcionaba puede deje de funcionar. ¿Cómo podemos crear programas resistentes a los cambios? Para ayudar a responder esa pregunta, Bertrand Meyer escribió en 1988 el famoso principio abierto/cerrado en su libro "Object Oriented Software Construction"; el principio presentado en este documento está tomado de ese libro, que es parte de la bibliografía y te recomendamos consultar.

Enunciado

Las clases[1] deben ser abiertas a la extensión, pero cerradas a la modificación.

Que una clase sea "abierta a la extensión" quiere decir que las responsabilidades de la clase pueden ser extendidas se puede agregar nuevas responsabilidades mediante herencia o mediante una combinación de herencia y composición o agregación.

Que una clase es "cerrada a la modificación" quiere decir que no es posible y no es necesario si se cumple el principio realizar cambios en el código de esa clase.

El principio OCP[2] es uno de los principios SOLID. Recuerden que término SOLID es un acrónimo mnemónico de cinco principios destinados a hacer que los diseños de software orientado a objetos sean más comprensibles, flexibles y fáciles de mantener. Los principios SOLID son un subconjunto de muchos principios promovidos por Robert C. Martin, su teoría fue introducida por él en su documento "Design Principles and Design Patterns"[3]. En la bibliografía está incluido el libro "Agile Principles Patterns and Practices In C#" de Robert C. Martin y Martin Micah, de 2007. También te recomendamos consultar este libro.

Ejemplo

Nuevamente usamos el ejemplo del punto de venta de éste y otros documentos como **Expert y SRP**, o **Polymorfism y LSP**. Los tickets de venta del ejemplo pueden tener ítems de cualquier producto y cantidad, pero tal como está el programa ahora no pueden tener descuentos, que es bastante frecuente en los puntos de venta[4]; para agregar líneas de descuento, tendríamos que modificar el código del programa; por lo tanto, la versión actual no cumple con el principio abierto/cerrado.

Supongamos que nuestra solución de punto de venta tiene una clase abstracta **SalesBaseItem** de la que heredan **SalesLineItem** y **SaleDiscount**. La clase **SalesBaseItem** tiene las responsabilidades de calcular el subtotal de una línea del ticket y de proveer el texto a imprimir en el ticket:

```
public abstract class SalesBaseItem
{
    public abstract double SubTotal { get; }
    public abstract string GetTextToPrint();
}
```

> Ver en repositorio »

De esta clase hereda la clase que ya teníamos **SalesLineItem**, que es exactamente igual a la de los ejemplos anteriores, excepto por el **override** en la propiedad **SubTotal** y el método `GetTextToPrint()`; los cambios están marcados en verde.

```
public class SalesLineItem : SalesBaseItem
{
    public SalesLineItem(double quantity, ProductSpecification product)
    {
        this.Quantity = quantity;
        this.Product = product;
    }

    public double Quantity { get; set; }
    public ProductSpecification Product { get; set; }
++  public override double SubTotal
    {
        get
        {
            return this.Quantity * this.Product.Price;
        }
    }

++  public override string GetTextToPrint()
    {
        return $"{this.Quantity} de '{this.Product.Description}' a ${this.Product.Price}\
    }
}
```

> Ver en repositorio »

Una nueva clase **SaleDiscount** hereda de **SalesBaseItem**, y tiene la responsabilidad de conocer el monto del descuento, implementado en la propiedad **Amount**. También sobrescribe la propiedad SubTotal y el método `GetTextToPrint();` noten que la propiedad SubTotal retorna el valor de **Amount**, pero en negativo, para que se reste del total de la venta.

```
public class SaleDiscount : SalesBaseItem
{
    public SaleDiscount(double amount)
    {
        this.Amount = ammount;
    }

    public double Amount { get; }

    public override double SubTotal
    {
        get
        {
            return - this.Amount;
        }
    }

    public override string GetTextToPrint()
    {
        return $"Descuento: -${this.Amount}\n";
    }
}
```

> Ver en repositorio »

Las otras modificaciones que tenemos que hacer son cambiar el atributo items de **Sale** para que referencie una **List<SalesBaseItem>** en lugar de **List<SalesLineItem>**; y agregar un método **AddDiscount** análogo al método **AddLineItem**; vean que **AddDiscount** crea una instancia de **SaleDiscount** y ya la agrega a la propiedad items; los cambios están marcados en verde.

```
public class Sale
{
+   private List<SalesBaseItem> lineItems = new List<SalesBaseItem>();
    …
+   public SalesLineItem AddLineItem(double quantity, ProductSpecification product)
    {
+       SalesLineItem item = new SalesLineItem(quantity, product);
        this.lineItems.Add(item);
        return item;
    }

+   public SaleDiscount AddDiscount(double ammount)
+   {
+      SaleDiscount item = new SaleDiscount(ammount);
+      this.lineItems.Add(item);
+      return item;
+   }
    …
}
```

> Ver en repositorio »

El resto del programa funciona exactamente igual, agreguemos un descuento al ticket que ya teníamos:

```
public class Program
{
    …
    public static void Main(string[] args)
    {
        …
        sale.AddLineItem(1, ProductAt(0));
        sale.AddLineItem(2, ProductAt(1));
        sale.AddLineItem(3, ProductAt(2));
+       sale.AddDiscount(50);
        …
    }
}
```

> Ver en repositorio »

La salida en consola que resulta de ejecutar este programa es la siguiente:

```
1 de 'Product 1' a $100
2 de 'Product 2' a $200
3 de 'Product 3' a $300
Descuento: -$50
Total: $1350
```

En esta nueva versión del programa podemos agregar nuevas responsabilidades sin modificar ninguna de las clases existentes[5]. Por ejemplo, podríamos agregar una clase SaleTax que calcule un impuesto como porcentaje del total de la venta; podemos agregar clases como esta simplemente como subclases de SalesBaseItem, el resto de la lógica del ticket sigue siendo igual.

La adhesión al principio abierto/cerrado es una de las cosas que produce los mayores beneficios en programación orientada a objetos:reusabilidad y mantenibilidad. Esto no se consigue solamente usando un lenguaje de programación orientado a objetos como C#, requiere pensar abstracciones en las partes del programa que se espera que puedan cambiar.

Apéndice

Esta parte del documento cubre conceptos avanzados de C# que no son necesarios para desarrollar las competencias esperadas de este curso. Para los curiosos que quieran conocer algo más allá de esas competencias, va a continuación una forma de evitar cualquier modificación a la clase **Sale** al agregar nuevas clases sucesores de **SalesBaseItem**, basada en la técnica inyección de dependencias[6] y los conceptos de delegados[7] y funciones lamba[8] de C#.

El código que tenemos hasta ahora requiere de métodos **AddLineItem**, **AddDiscount**, **AddTax** para agregar instancias de sucesores de **SalesBaseItem** como **SalesLineItem**, **SaleDiscount** y **SaleTax**; cada nuevo sucesor de **SalesBaseItem** requiere un nuevo método **Add** en **Sale**. Vean que la única diferencia entre esos métodos es la creación de la instancia a agregar, el resto es igual:

```
public class Sale
{
```

```
    …
    public SalesLineItem AddLineItem(double quantity, ProductSpecification product)
    {
+        SalesLineItem item = new SalesLineItem(quantity, product);
        this.lineItems.Add(item);
        return item;
    }

    public SaleDiscount AddDiscount(double amount)
    {
+        SaleDiscount item = new SaleDiscount(amount);
        this.lineItems.Add(item);
        return item;
    }

    public SaleTax AddTax(double percentage)
    {
+        SaleTax item = new SaleTax(percentage, this);
        this.lineItems.Add(item);
        return item;
    }
}
```

> Ver en repositorio »

La inyección de dependencias consiste en proveer a un objeto las dependencias a otro objeto; una dependencia es un objeto que puede ser usado como un servicio, una inyección es pasar esa dependencia al objeto dependiente que va a usarlo. En nuestro caso, la clase Sale depende de crear instancias de sucesores de **SalesBaseItem** el código marcado de verde, podríamos inyectar ese código para no tener tantos métodos **Add** como clases sucesoras de SalesBaseItem.

Ahora viene lo interesante, los métodos en C# son objetos, llamados delegados. El código que queremos inyectar es un método que retorna una instancia de sucesores de **SalesBaseItem**, y podemos hacerlo usando delegados. Esta nueva versión de la clase **Sale** a continuación tiene un solo método **Add** cuyo argumento es un delegado. **Func** indica que ese delegado es una función que retorna una instancia de SalesBaseItem, que se invoca con el mensaje `Invoke()` para producir el resultado, exactamente lo que queremos lograr:

```
public class Sale
{
    …
    public SalesBaseItem Add(Func<SalesBaseItem> createItem)
    {
        SalesBaseItem item = createItem.Invoke();
        this.lineItems.Add(item);
        return item;
    }
    …
}
```

> Ver en repositorio »

En la clase **Program** tenemos que proveer una instancia para ese delegado, es decir, un método que retorne una instancia de **SalesBaseItem**, ya sea **SalesLineItem**, **SaleDiscount** o **SaleTax** para agregar líneas al ticket de venta. Podemos lograrlo usando otra característica de C# llamada funciones lambda. Una función lambda es una forma de escribir métodos locales que pueden ser pasados como argumento cuando se espera un delegado. Para crear una función lambda se utiliza el operador lambda, `=>`, que se lee "da como resultado". Por ejemplo, la función lambda `x => x * x` especifica una función con un parámetro llamado x y da como resultado el cuadrado de x. En nuestro ejemplo, el código para crear una instancia de `SaleDiscount es () => new SaleTax(50)` y se pasa como argumento en `sale.Add()` :

```
public class Program
{
    …
    public static void Main(string[] args)
    {
        …
        sale.Add(() => new SalesLineItem(1, ProductAt(0)));
        sale.Add(() => new SalesLineItem(2, ProductAt(1)));
        sale.Add(() => new SalesLineItem(3, ProductAt(2)));
        sale.Add(() => new SaleDiscount(50));
        sale.Add(() => new SaleTax(10, sale));
        …
    }
    …
}
```

> Ver en repositorio »

Esta versión del programa produce exactamente el mismo resultado que el anterior.

La responsabilidad de crear instancias de sucesores de **SalesBaseItem** sigue siendo de la clase **Sale**, sólo que el código necesario para hacerlo se pasa como argumento. Usando inyección de dependencias, delegados y funciones lambda, aplicamos el patrón Creator de asignación de responsabilidades, y además cumplimos con el principio abierto/cerrado. No se preocupen si esto les resulta muy complejo, no se espera que lo comprendan, ni es necesario que lo apliquen.

[1] 3 El enunciado original incluye otras entidades además de clases, que en C# serían las bibliotecas y los espacios de nombres, por ejemplo.

[2] Open/Closed Principle.

[3] Ver https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf .

*[4] Podríamos tener un producto con nombre **Discount** y precio **1.0** y crear una instancia de SalesLineItem con ese producto y la cantidad que queramos hacer de descuento; si bien el programa podría funcionar, no es correcto usar clases para un propósito que no fueron previstas, sería "tirado de los pelos".*

*[5] Esta afirmación es parcialmente correcta: es necesario agregar un método **AddTax** en **Sale** para crear una instancia de **SaleTax**. Aunque esto implica cambiar el código de **Sale**, no modifica ninguno de los métodos, propiedades o atributos existentes, con lo cual es poco probable que se introduzca un error. Vean la adenda para conocer una forma de evitar hacer este cambio también.*

[6] Ver por ejemplo https://en.wikipedia.org/wiki/Dependency_injection .

[7] Delegates. Para conocer más sobre delegados, vean por ejemplo https://docs.microsoft.com/en-us/dotnet/csharp/programmingguide/delegates/ .

[8] Lambda expressions. Para conocer más sobre funciones lambda, vean por ejemplo: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions .