OCP(ABIERTO-CERRADO PRINCIPIO)

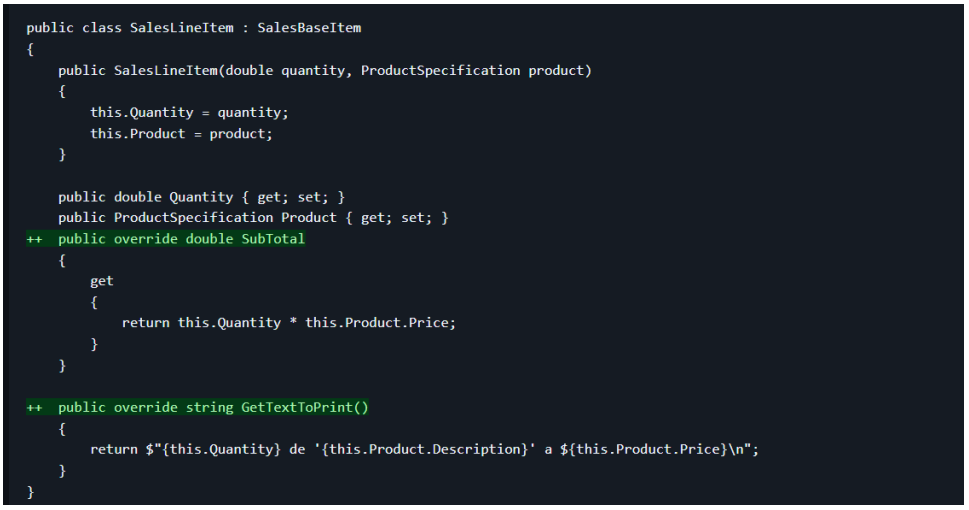
Un programa se escribe una vez, pero se modifica muchas veces, lo que puede introducir errores. Para crear programas resistentes a cambios, Bertrand Meyer propuso en 1988 el **principio abierto/cerrado (OCP)** en su libro *Object Oriented Software Construction*. Este principio establece que las clases deben ser **abiertas a la extensión**, lo que significa que se pueden agregar nuevas responsabilidades mediante herencia o composición, y **cerradas a la modificación**, es decir, no deben requerir cambios en su código.

Ejemplo:

En el contexto de un punto de venta, se señala que los tickets de venta pueden tener ítems de cualquier producto, pero no pueden incluir descuentos sin modificar el código, lo que implica que no cumplen con el principio OCP. Para solucionar esto, se propone crear una clase abstracta **SalesBaseItem**, de la que heredan **SalesLineItem** y **SaleDiscount**, para calcular subtotales y proveer texto para imprimir, permitiendo extender el sistema sin modificar el código existente.

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

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}\n";
    }
}
```

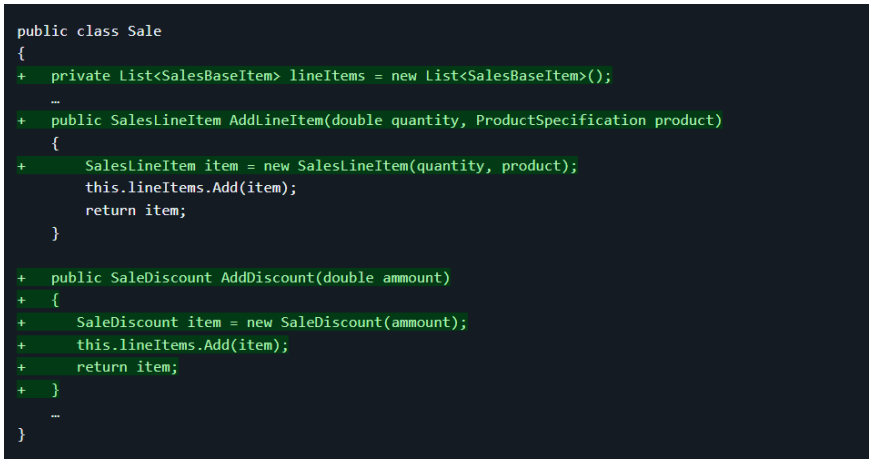
Una nueva clase **SaleDiscount** hereda de **SalesBaseItem**, y tiene la responsabilidad de conocer el monto del descuento, implementado en la propiedad **Amount**. También sobrescribe la propiedad SubTotal y el método `GetTextToPrint();` noten que la propiedad SubTotal retorna el valor de **Amount**,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

```
public class SaleDiscount : SalesBaseItem
{
public SaleDiscount(double amount)
{
```

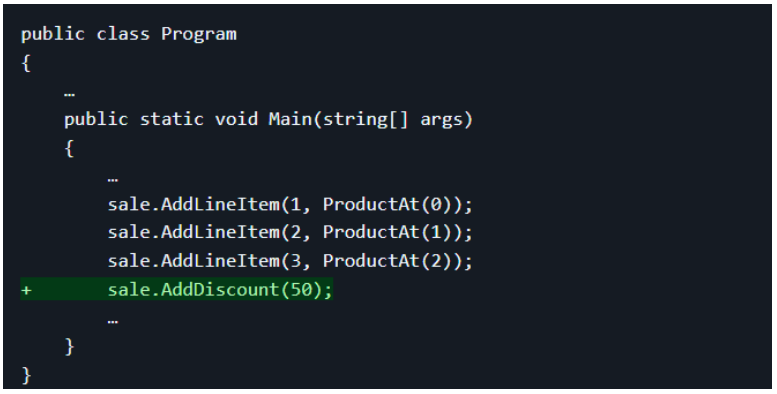
```
this.Amount = ammount;
}
```

```
public double Amount { get; }

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

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

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



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



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 la nueva versión del programa, se pueden agregar responsabilidades sin modificar las clases existentes. Por ejemplo, se puede incluir una clase **SaleTax** para calcular un impuesto como porcentaje del total de la venta. Estas nuevas clases se pueden implementar como subclases de **SalesBaseItem**, manteniendo la lógica del ticket sin cambios.

> La adherencia al principio abierto/cerrado proporciona beneficios significativos en programación orientada a objetos, como la **reusabilidad** y la **mantenibilidad**. Sin embargo, estos beneficios no solo dependen del uso de un lenguaje orientado a objetos como C#, sino también de la capacidad de pensar en abstracciones para las partes del programa que se anticipa que cambiarán.

Apéndice:

Esta sección aborda conceptos avanzados de C# que van más allá de las competencias básicas del curso. Se menciona una técnica para evitar modificaciones en la clase **Sale** al agregar nuevas clases derivadas de **SalesBaseItem**, utilizando **inyección de dependencias**, **delegados** y **funciones lambda**. El código actual requiere métodos **AddLineItem**, **AddDiscount** y **AddTax** para agregar instancias de subclases, donde la única diferencia entre estos métodos es la creación de la instancia que se va a agregar.

```
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;
    }
}
```

La **inyección de dependencias** implica proporcionar a un objeto las dependencias que necesita de otros objetos, es decir, pasarle las dependencias que usará como servicios. En el caso de la clase **Sale**, que depende de crear instancias de subclases de **SalesBaseItem**, se puede inyectar esa lógica para evitar tener múltiples métodos **Add** para cada subclase.

Una característica interesante de C# es que los métodos son tratados como objetos, conocidos como **delegados**. Se puede inyectar el código que crea instancias de subclases de **SalesBaseItem** mediante delegados. La nueva versión de la clase **Sale** tiene un único método **Add**, que recibe un delegado como argumento. Este delegado es una función que devuelve una instancia de **SalesBaseItem** y se invoca utilizando el método **Invoke()** para generar el resultado deseado. Esto simplifica el código al reducir la cantidad de métodos necesarios para agregar diferentes tipos de ítems.

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

> En la clase **Program**, es necesario proporcionar una instancia para el delegado, es decir, un método que retorne una instancia de **SalesBaseItem**, como **SalesLineItem**, **SaleDiscount** o **SaleTax**, para agregar líneas al ticket de venta. Esto se puede lograr mediante **funciones lambda**, que son una forma de escribir métodos locales que se pueden pasar como argumentos cuando se espera un delegado.

Las funciones lambda se crean utilizando el operador lambda `=>`, que se lee como "da como resultado". Por ejemplo, la función lambda `x => x * x` define una función que toma un parámetro `x` y devuelve el cuadrado de `x`. En el caso del ticket de venta, el código para crear una instancia de **SaleTax** se escribiría como `() => new SaleTax(50)`, y se pasa como argumento en `sale.Add()`. Esto permite agregar líneas al ticket de forma más flexible y concisa

```
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));
…
}
…
}
```

Esta versión del programa produce los mismos resultados que la anterior. La clase **Sale** sigue siendo responsable de crear instancias de las subclases de **SalesBaseItem**, pero ahora el código para hacerlo se pasa como argumento.

Al utilizar **inyección de dependencias**, **delegados** y **funciones lambda**, se aplica la guía **Creator** para la asignación de responsabilidades y se cumple con el **principio abierto/cerrado**. No es necesario que los lectores comprendan completamente estos conceptos o que los apliquen, ya que pueden resultar complejos.