# Single Responsability Principle (SRP)

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 desde estar completamente encapsulada por la clase.

Todos los métodos y atributos de la clase deben estar estrechamente alineados con esa responsabilidad. El principio se expresa como: *Una clase debe tener solo una razón para cambiar*

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. Aunque hacerlo necesita información que está en esta clase, si en lugar de imprimir en la consola quisiera imprimir en una impresora, la clase debería cambiar, así como debería cambiar si incluyéramos descuentos, por ejemplo. Entonces existe más de una razón por la cual la clase debe cambiar, lo que viola el principio SRP. Podemos separar la responsabilidad de imprimir el ticket a una nueva clase ConsolePrinter, esta debe colaborar con SaleTicket, que le provee el texto a imprimir.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

public class ConsolePrinter

{

public static void PrintTicket(Sale sale)

{

Console.WriteLine(sale.GetTicketText());

}

}

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();

}

}

Se usa la clase StringBuilder cuando hay intensa manipulación de texto, ya que, a diferencia de la clase string, esta no es inmutable, por lo que cada concatenación de texto no implica crear nuevas instancias.

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);

}

}

En resumen: Para agregar una clase RollerPaperPrinter podemos comenzar a imprimir la factura de papel sin modificar ninguna de las clases existentes.

# Patrón Expert

El principio más básico para asignar responsabilidades en un diseño orientado a objetos es asignar la responsabilidad al experto en información, es decir a la clase que tiene la información necesaria para cumplir con la responsabilidad.

**Ejemplo:** Aplicación de punto de venta:

Ticket simplificado:

Fecha: 31/3/2021

2 de 'Agua mineral' a $25

1 de 'Café cortado' a $35

1 de 'Café expreso' a $31

Para mostrar las clases y sus responsabilidades vamos a usar tarjetas. Arriba va el nombre de la clase, abajo a la izquierda la lista de responsabilidades y a la derecha las clases que colaboran con ésta para cumplir esas responsabilidades. Estas tarjetas se llaman CRC (clases, responsabilidades y colaboraciones).

Interfaz de usuario gráfica

Descripción generada automáticamente

La clase TicketLineItem representa la línea del ticket con la cantidad y el producto vendido. Y la clase ProducSpecification representa los productos.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

public class SaleTicket

{

private ArrayList lineItems = new ArrayList();

public DateTime DateTime { get; set; }

public void AddLineItem(TicketLineItem item)

{

this.lineItems.Add(item);

}

public void RemoveLineItem(TicketLineItem item)

{

this.lineItems.Remove(item);

}

public void PrintTicket()

{

Console.WriteLine($"Fecha: {this.DateTime}");

foreach (TicketLineItem item in this.lineItems)

{

Console.WriteLine($"{item.Quantity} de '{item.Product.Description}' a ${item.Product.Price}");

}

}

}

public class TicketLineItem

{

public TicketLineItem(double quantity, ProductSpecification product)

{

this.Quantity = quantity;

this.Product = product;

}

public double Quantity { get; set; }

public ProductSpecification Product { get; set; }

}

public class ProductSpecification

{

public ProductSpecification(string description, double price)

{

this.Description = description;

this.Price = price;

}

public string Description { get; set; }

public double Price { get; set; }

}

Ahora, si quisiéramos agregar al ticket el total de la venta, por el patron Expert deberíamos mirar qué clases tienen la información necesaria para determinar el total. Necesito conocer todas las instancias de TicketLineItem de un ticket, para luego sumar los subtotales de cada línea. Solo las instancias de SaleTicket tiene la responsabilidad de conocer esta información, por lo tanto es el experto en información.

Imagen que contiene Texto

Descripción generada automáticamente

public class SaleTicket

{

…

public double Total

{

get

{

double result = 0;

foreach (TicketLineItem item in this.lineItems)

{

result = result + (item.Quantity \* item.Product.Price);

}

return result;

}

}

…

public void PrintTicket()

{

Console.WriteLine($"Fecha: {this.DateTime}");

foreach (TicketLineItem item in this.lineItems)

{

Console.WriteLine($"{item.Quantity} de '{item.Product.Description}' a ${item.Product.Price}");

}

Console.WriteLine($"Total: ${this.Total}");

}

}

Texto

Descripción generada automáticamente

Ahora, para calcular el subtotal de una línea necesito el precio del producto y la cantidad vendida. TicketLineItem tiene la responsabilidad de conocer la cantidad y el producto. El último es una instancia de ProductSpecification, que tiene la responsabilidad de conocer el precio. Por lo que la responsabilidad se le asigna a TicketLineItem.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

public class TicketLineItem

{

…

public double SubTotal

{

get

{

return this.Quantity \* this.Product.Price;

}

}

}

public class Sale

{

…

public double Total

{

get

{

double result = 0;

foreach (TicketLineItem item in this.lineItems)

{

result = result + (item.Quantity \* item.Product.Price);

result = result + item.SubTotal;

}

return result;

}

}

…

}

El patrón Expert es usado más que ningún otro patrón en la asignación de responsabilidades, es un principio guía básico usado continuamente en el diseño orientado a objetos. Expresa la intuición de sentido común de que los objetos hacen cosas relacionadas con la información que tienen.

Para cumplir una responsabilidad a menudo es necesario información que está desperdigada a través de diferentes clases de objetos. Esto implica que hay “expertos parciales” que colaboran para cumplir con la responsabilidad.

**Beneficios:** La encapsulación se mantiene, porque los objetos usan su propia información para cumplir con las responsabilidades. Esto mantiene el acoplamiento bajo, lo que produce programas más robustos y fáciles de mantener. El comportamiento se distribuye a través de clases que tienen la información requerida, promoviendo definiciones de clases mas cohesivas que son más bfáciles de entender y mantener.