TuEnvío 2.0

Usted ha sido contratado para la remodelación y actualización de la plataforma TuEnvío. En concreto, su trabajo consistirá en implementar la funcionalidad del carrito de compras.

Como su código debe coexistir con la arquitectura de software existente, usted debe implementar la `interface IShoppingCart`:

```
public interface IShoppingCart<TProduct> where TProduct : IProduct
{
    int Cost { get; }
    int Total { get; }

    void Add(TProduct product, int count);
    bool Remove(TProduct product);
    int Count(TProduct product);
    int Count(IFilter<TProduct> filter);
    void AddPromotion(IPromotion<TProduct> promotion);
}
```

Esta `interface` encapsula las responsabilidades de un carrito de compras, que consiste en almacenar una colección de productos (`inteface IProduct`) y calcular el costo total de la compra.

Adicionalmente, el carrito de compras puede tener promociones (`interface IPromotion`) que permiten aplicar descuentos a diferentes productos por diferentes criterios. También es posible consultar el estado del carrito de compras según diferentes filtros (`interface IFilter`).

En su implementación, usted debe hacer uso de las interfaces `IProduct`, `IFilter` e `IPromotion` de manera que su implementación de `IShoppingCart` sea compatible con cualquier implementación de estas `interfaces` que los otros desarrolladores creen.

Para proveer su implementación de `IShoppingCart` a los demás clientes de su código, usted debe implementar el método `GetShoppingCart` de la clase `TuEnvio`. Este método recibe un parámetro `capacity` que representa la cantidad máxima de productos diferentes que pueden ser adicionados al carrito. Se garantiza que nunca se adicionarán una cantidad mayor de productos **diferentes** que este valor.

```
public static class TuEnvio
{
    public static IShoppingCart<TProduct> GetShoppingCart<TProduct>(int capacity)
        where TProduct : IProduct
    {
        // Borre aquí y devuelva una instancia de su implementación de IShoppingCart
        throw new NotImplementedException();
```

```
    }
}
```

A continuación se explica en más detalle las funcionalidades a implementar.

Manejo de productos

La `interface IProduct` encapsula el concepto de un producto a ubicar en el carrito:

```
public interface IProduct
{
    int Price { get; }
    string Description { get; }
}
```

Todo producto tiene un precio base (propiedad `Price`) y una descripcion (propiedad `Description`).

El carrito de compras debe permitir adicionar una o más unidades de cualquier producto a partir del método `Add`.

```
interface IShoppingCart<TProduct> where TProduct: IProduct
{
    // ...
    void Add(TProduct product, int count);
    // ...
}
```

Note que `IShoppingCart` es genérico en el tipo de producto.

A los efectos de su implementación, dos instancias de `IProduct` se consideran el mismo producto si y solo si el método `Equals` de la instancia correspondiente devuelve `true`, independientemente del valor de las propiedades `Price` o `Description`.

Teniendo esto en cuenta, es equivalente invocar `Add` dos veces con instancias del mismo producto con diferentes valores para `count`, que invocar `Add` una sola vez con la suma de ambos valores.

Además de adicionar, el carrito de compras permite eliminar todas las unidades de un producto mediante el método `Remove`:

```
interface IShoppingCart<TProduct> where TProduct: IProduct
{
    // ...
    bool Remove(TProduct product);
    // ...
}
```

Este método devuelve `true` si el producto existía en el carrito, y posteriormente a su invocación no debe quedar ninguna unidad del mismo producto, independientemente de cuántas veces se haya invocado `Add` con el mismo producto.

En cualquier momento, la propiedad `Total` de `IShoppingCart` debe devolver la cantidad total de productos diferentes en el carrito, independientemente de la cantidad de unidades de cada producto.

El carrito permite además saber la cantidad de unidades de un producto específico mediante el método `Count`:

```
interface IShoppingCart<TProduct> where TProduct: IProduct
{
    // ...
    int Count(TProduct product);
    // ...
}
```

Este método debe devolver la cantidad total de unidades del producto correspondiente, independientemente de cuántas veces se haya invocado `Add` con el mismo producto.

Filtrado de productos

Con vistas a soportar ciertas funcionalidades de la interfaz gráfica, el carrito de compras debe implementar una funcionalidad para contar la cantidad de productos que cumplen con cierto criterio arbitrario. El concepto de filtro se encapsula en la `interface IFilter`:

```
public interface IFilter<T>
{
    bool Apply(T item);
}
```

El método `Apply` recibe un elemento arbitario (de un tipo genérico `T`) y devuelve `true` si cumple con la condición que encapsula la implementación concreta de `IFilter` que tenga en ese momento.

Para utilizar este criterio, el carrito de compras implementa una sobrecarga del `Count` que recibe una instancia de `IFilter` genérica en el tipo concreto de producto que corresponda:

```
interface IShoppingCart<TProduct> where TProduct: IProduct
{
    // ...
    int Count(IFilter<TProduct> filter);
    // ...
}
```

El resultado de este método será la suma total de todas las unidades de todos los productos que cumplan con la condición del filtro correspondiente.

Cálculo del costo

Finalemente, la funcionalidad principal del carrito de compras es calcular el costo total de todos los productos. En principio, el costo será la suma de todos las unidades de cada producto almacenado, multiplicadas por su respectivo precio base.

Sin embargo, en virtud los cambios económicos actuales, la empresa considera interesante introducir promociones para facilitar la compra de ciertos productos de alta demanda.

Una promoción es un concepto que aplica a uno o más tipos de productos en función de la cantidad de unidades que estén en el carrito de la compra. Por ejemplo, una promoción de navidad puede aplicar a todos los productos de la canasta básica de navidad (cerdo, arroz, aceitunas, uvas, vino) siempre que el cliente tenga más de 5 unidades de cada uno. Al aplicar una promoción se aplica un descuento que corresponde a un porciento del precio total que debe ser disminuido en el cálculo del costo.

Este concepto se encapsula en la `interface IPromotion<TProduct>`:

```
public interface IPromotion<TProduct> where TProduct : IProduct
{
    double Discount(TProduct product, int count);
}
```

El método `Discount` recibe un producto y la cantidad de unidades de ese producto existentes en el carrito. Devuelve un valor `double` entre 0 y 1 que representa el porciento de descuento.

Por ejemplo, supongamos que el producto corresponde a una botella de vino, con precio base de 250 CUP. Supongamos que la promoción aplica a este tipo de producto a partir de 5 unidades con un descuento del 20%. Entonces, en un carrito con 4 unidades de botellas de vino, la invocación al método `Discount` devoverá `0` pues la promoción no aplica. Así mismo, si se ejecuta el método `Discount` sobre otro producto que no tiene promoción (e.j, maíz molido), también se devolverá `0` independientemente de la cantidad de productos. En cambio, si un carrito tiene 10 botellas de vino, al invocar `Discount` con este valor para `count` en el producto correcto, el resultado será `0.2`, que corresponde al 20% de descuento. Entonces, el precio que usted debe considerar será el precio base (10 * 250 = 2500 CUP) restando el descuento (2500 - 0.2 * 2500 = 2000 CPU).

Si existen más de una promoción que aplique para el mismo producto, se deben sumar todos los descuentos y aplicar **una sola vez** el descuento total. Por ejemplo, si una promoción devuelve `0.1` y otra devuelve `0.2` para el mismo producto, usted debe aplicar un descuento total de 30% al producto.

Note además que una promoción puede devolver diferentes valores para diferentes cantidades, por ejemplo, el descuento después de 5 unidades puede ser un 20%, pero después de 10 puede ser un 30%. Por lo tanto, usted debe invocar `Discount` siempre con la cantidad total de unidades de cada producto.

Para adicionar una promoción al carrito se implementa el método `AddPromotion`:

```
interface IShoppingCart<TProduct> where TProduct: IProduct
{
    // ...
    void AddPromotion(IPromotion<TProduct> promotion);
    // ...
}
```

Una vez entendido el mecanismo de promoción, lo que resta es implementar la propiedad `Cost` de `IShoppingCart`, que devolverá en todo momento el costo total del carrito teniendo en cuenta todas las promociones que hayan sido adicionadas mediante el método `AddPromotion`.

Ejemplo

A continuación presentamos un ejemplo hipotético del uso de su implementación.

Supongamos que existe una clase `SimpleProduct` que implementa la interfaz `IProduct`. Para instanciar un nuevo carrito de compras con capacidad máxima para 3 productos diferentes, se ejecutaría la línea siguiente:

```
IShoppingCart cart = TuEnvio.GetShoppingCart<SimpleProduct>(3);
```

A partir de este punto, se podrían añadir productos de la siguiente forma (asumiendo que el constructor de `SimpleProduct` recibe precio y descripción):

```
SimpleProduct pollo = new SimpleProduct(100, "Pollo (1kg)");
SimpleProduct pescado = new SimpleProduct(150, "Pescado (1kg)");
SimpleProduct vino = new SimpleProduct(250, "Botella de Vino");

cart.Add(pollo, 5);     // Añadiendo 5 kilos de pollo
cart.Add(pescado, 3);   // Añadiendo 3 kilos de pescado
cart.Add(vino, 2)       // Añadiendo dos botellas de vino
cart.Add(pollo, 2)      // Añadiendo 2 kilos de pollo adicionales
```

Note que aunque la capacidad es 3, y se ha invocado 4 veces al método `Add`, como realmente solo existen tres productos diferentes (el pollo se ha añadido 2 veces), la capacidad del carrito no se ha excedido.

En este punto sería posible calcular la cantidad de cada producto de esta forma:

```
int polloTotal = cart.Count(pollo);      // Devolvería 7 (5 + 2)
int pescadoTotal = cart.Count(pescado);  // Devolvería 3
int vinoTotal = cart.Count(pescado);     // Devolvería 2
```

Supongamos entonces que existe una implementación de `IFilter` que devuelve `true` para todos los productos cárnicos:

```
IFilter<SimpleProduct> filtroCarnico = // ... (no importa como se implementa este filtro)

filtroCarnico.Apply(pollo);   // Devolvería `true`
filtroCarnico.Apply(pescado); // Devolvería `true`
filtroCarnico.Apply(vino);    // Devolvería `false`

int totalCarnico = cart.Count(filtroCarnico);  // Devolvería 10 (7 pollo + 3 pescado)
```

Teniendo en cuenta el contenido del carrito, el costo total sería `1950` (`7*100 + 5*150 + 2*250`).

Supongamos ahora que se aplica una promoción a los productos cárnicos (pollo y pescado) con un descuento de un 10% si hay 5 unidades o más. Esta promoción por lo tanto aplica al pollo pero no al pescado (porque son solo 3 unidades), ni al vino (porque no está entre los productos promocionados). Con esta promoción aplicada, el costo sería `1810`, pues a los `700` del pollo se le reduce un 20%, quedando ese subtotal en `560`, y los otros dos subtotales son iguales.

```
IPromotion<SimpleProduct> promocionCubaneo = // ... (no importa como se implementa esta pro

promocionCubaneo.Discount(pollo, 7);    // Devolvería 0.2
promocionCubaneo.Discount(pescado, 3);  // Devolvería 0
promocionCubaneo.Discount(vino, 2);     // Devolvería 0

cart.AddPromotion(promocionCubaneo);
int cost = cart.Cost;   // Devolvería 1810
```

Notas

- Usted no necesita implementar las interfaces `IProduct`, `IFilter` o `IPromotion` para ser evaluado, solo la interfaz `IShoppingCart`. Sin embargo, para probar su implementación, le será conveniente probar algunas implementaciones suyas de estas interfaces. Siéntase libre de hacerlo a su conveniencia.
- La implementación concreta de `IShoppingCart` que se utilizará es la que usted devuelva en el método `GetShoppingCart()` de la clase `TuEnvio`. Asegúrese de devolver una nueva instancia de su implementación, teniendo en cuenta la capacidad.
- Usted es libre de utilizar cualquier estructura de datos de .NET que desee. Sin embargo, no es necesario utilizar nada que no haya sido visto en clases para resolver este problema fácilmente. Si lo piensa bien, no necesita nada más que *arrays*.