Resumen de clase
Ejercicio Clientes de una Tarjeta de Crédito (Introducción al Decorator Pattern)

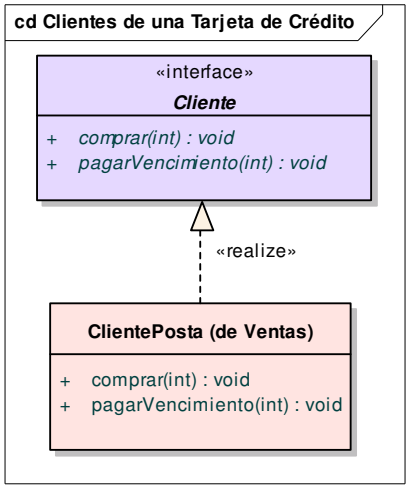
2° cuatrimestre 2008

Contenido

Enunciado

Pertenecemos a la gerencia de Condiciones Comerciales de una empresa emisora de una Tarjeta de Crédito. La gerencia de Ventas nos provee una interfaz Cliente, cuyos contratos son:

- comprar(int monto)
- pagarVencimiento(int monto)



Se pide contemplar los siguientes requerimientos:

- algunos clientes adheridos a una promoción suman 15 puntos por cada compra mayor a $ 50.
- además, algunos clientes contrataron el sistema 'Safe Shop', que bloquea compras de la tarjeta mayores a un monto que el cliente fija.

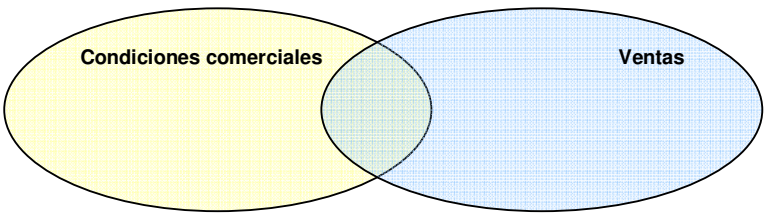
Sobre el dominio

Tenemos dos sectores dentro de la empresa de tarjetas de crédito:

- Ventas
- Condiciones Comerciales

¿Qué responsabilidades cumplen cada una?
Si bien la empresa tiene vendedores que son quienes interactúan en muchos casos con los clientes (para ofrecer nuevos servicios), en muchas empresas existe un sector que determina restricciones o acuerdos que los vendedores deben cumplir, que son las “condiciones comerciales”.
Ejemplos:

- Todos los clientes del exterior tienen un descuento por pronto pago del 20%
- Sólo se puede trabajar con clientes mayoristas
- Los clientes Rabufetti y Conesa aceptan cheques a 30, 60 y 90 días como forma de pago



Vemos que Cliente intersecta el negocio de ambos sectores (Ventas y Condiciones Comerciales). Ahora veremos cómo atacar estos requerimientos.

Solución

¿De qué lado cae la pelota?

Aquí vemos que los requerimientos que se agregan corresponden a condiciones comerciales de cada venta, pero que también aparece en juego una cuestión política: si la Gerencia de Ventas quiere restringir los cambios en la clase Cliente, porque "es" de Ventas[1], aparecen restricciones que limitan mi solución final. Para el resto del ejercicio vamos a suponer que yo **sí** puedo modificar la clase Cliente, y vamos a pensar en alternativas de diseño.

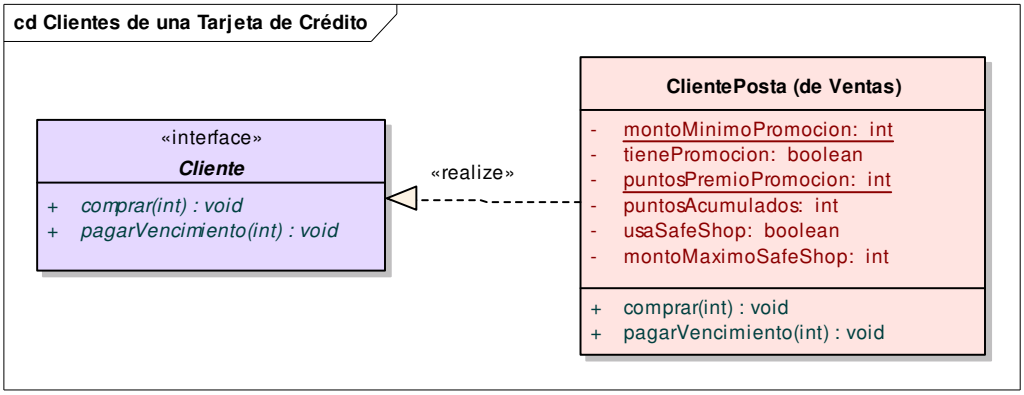
1) El if nuestro de cada día

Si aprovechamos el consejo de ir por lo más simple… bueno, lo más sencillo es modificar la clase cliente:

- Necesitamos saber si el cliente está adherido a la promoción, cuál es el monto mínimo y los puntos que se le otorgan
- Necesitamos conocer si el cliente está adherido al sistema Safe Shop y cuál es el monto máximo de compra

Hacemos los cambios pertinentes:

[1] Asociar una clase a una persona o grupo de desarrolladores (lo que en la jerga de Sistemas se llama "armar una quinta") también recibe el nombre **code ownership**: pensar en que el código tiene un dueño. El primer impacto negativo es fácil de apreciar: si me cuesta modificar la clase Cliente, ya no pienso en la simplicidad como cualidad de mi diseño solución sino en no verme comprometido a cambiar el Cliente. Además genera numerosos recelos entre equipos de Ventas y Condiciones Comerciales.



El código de ClientePosta queda:

```
public void comprar(int monto) {

    if (usaSafeShop && monto > montoMaximoSafeShop) {
        throw new BusinessException("Ha excedido el monto maximo");
    }

    if (tienePromocion && monto > montoMinimoPromocion) {
          puntosAcumulados += puntosPremioPromocion;
    }

    ...
}
```

Podemos evitar el atributo usaSafeShop convirtiéndolo en un método:

```
public void comprar(int monto) {

    if (this.usaSafeShop() && monto > montoMaximoSafeShop) {
          ...

// Si el monto máximo es cero no usa Safe Shop
public boolean usaSafeShop() {
    return montoMaximoSafeShop > 0;
}
```

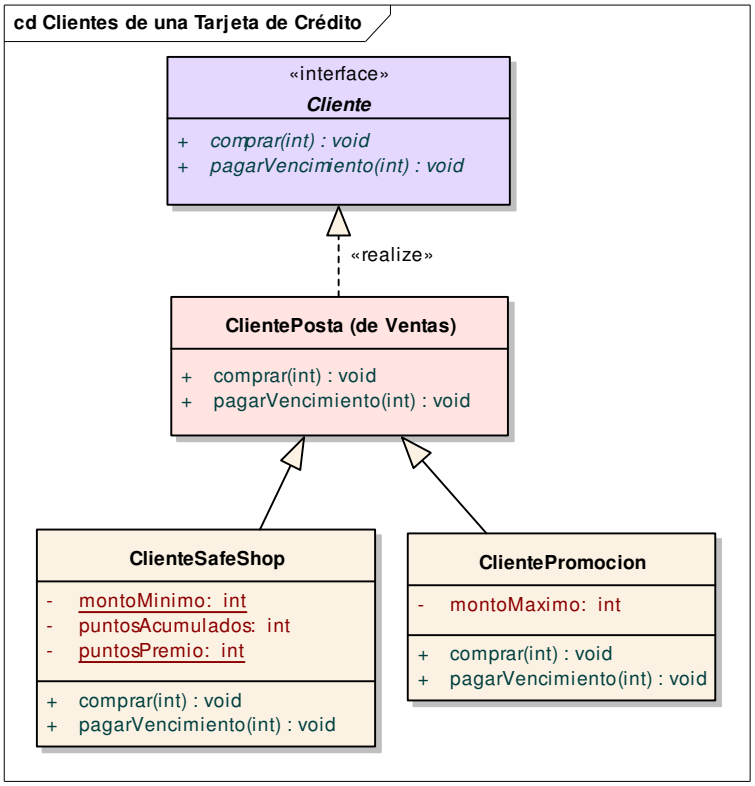
Ok, esta solución tiene algunos inconvenientes:

- "Ensucia" la lógica del método comprar() cliente
- Agrega atributos que no todos los clientes necesitan: si el cliente no se adhiere a la promoción no necesita tener puntosAcumulados. El problema no es guardar una referencia de más, sino que el que lee la clase Cliente puede confundirse pensando que esa variable la necesita el cliente (esté o no en promoción).

Por otra parte es fácil hacer que el cliente entre y salga de la promoción de puntos o habilite/deshabilite la opción Safe Shop. Como los requerimientos son más bien simples de codificar, agregar un if es una opción más que posible. Si para calcular los puntos acumulados necesitara una lógica más compleja o bien si aparecieran muchas más condiciones comerciales quizás podría empezar a molestarme la cantidad de líneas agregadas en esos ifs.

2) Subclasificar

Otra opción es subclasificar el concepto ClientePosta. Tenemos así dos subclases: ClienteSafeShop y ClientePromocion.



En ClienteSafeShop tenemos:

```
public void comprar(int monto) {
    if (monto > montoMaximo) {
        throw new BusinessException("Ha excedido el monto maximo");
    }

    super.comprar(monto);
}
```

En ClientePromocion tenemos:

```
public void comprar(int monto) {
    super.comprar(monto);

    if (monto > montoMinimo) {
          puntosAcumulados += puntosPremio;
    }
}
```

Esta alternativa separa claramente las responsabilidades de cada condición comercial. Ya no se ensucia al ClientePosta, sino que cada subclase delega en la superclase el comportamiento default del comprar y además hace su propio agregado. Parece una trivialidad pero no lo es, el atributo montoMaximoSafeShop pasa a llamarse montoMaximo a secas.

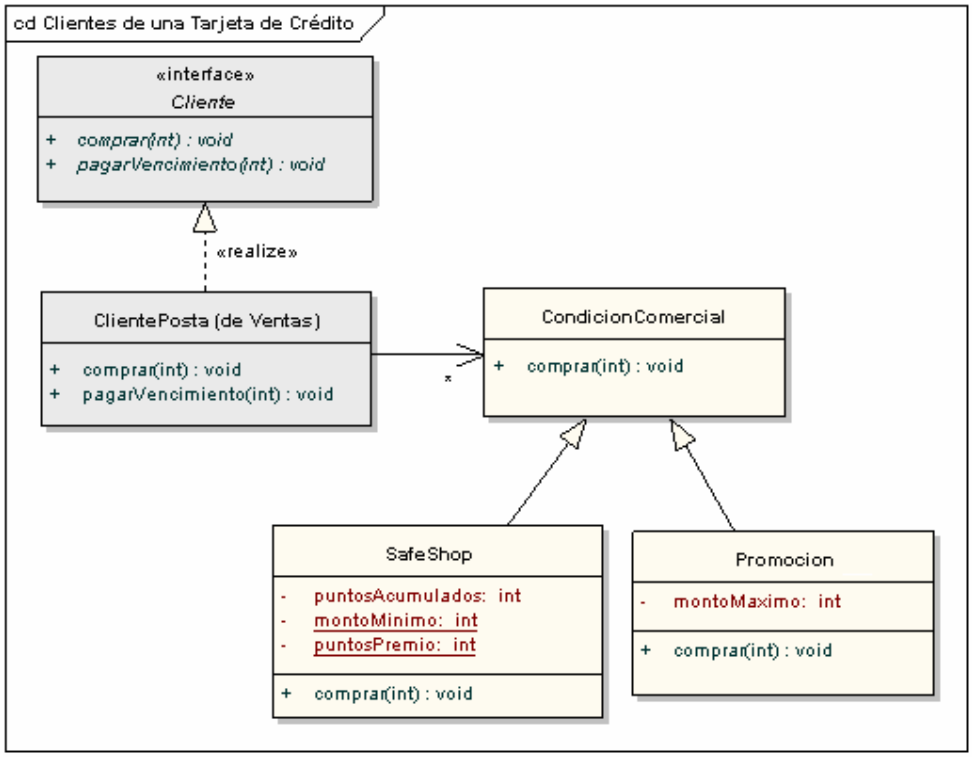
No obstante esta opción presenta una clara desventaja: no sólo es complicado hacer que un cliente posta habilite/deshabilite el servicio de Safe Shop: **esta solución no permite que coexistan un cliente en promoción que también tenga safe shop,** y es el motivo que lo convierte en la solución menos deseable de todas.

3) Tener una colección de condiciones comerciales

Para esto necesitamos entender que queremos separar dos conceptos que *no son intrínsecos*:

- El cliente en sí (representado por la clase ClientePosta)
- Las condiciones comerciales del cliente, que pueden ser muchas y además las quiero combinar

Algo así:



¿Qué tipo de colección voy a usar para las condiciones comerciales? Y… no me da lo mismo si sumo los puntos y resulta que salta la alarma de Safe Shop. Como además no quiero tener dos veces a la condición comercial Safe Shop, entonces el ClientePosta usará un **set ordenado** de condiciones comerciales.

A esta altura no necesitamos llevar a las condiciones comerciales el método pagarVencimiento(). Esta alternativa toma como propias las ventajas de las soluciones anteriores:

- Separa el comportamiento propio de cada condición comercial
- Es fácil agregar o sacar una condición comercial dinámicamente: basta con actualizar la lista de condiciones comerciales del ClientePosta

El código de ClientePosta cambia a:

```
public void comprar(int monto) {
    for (CondicionComercial condicion : condicionesComerciales) {
          condicion.comprar(monto);
    }
    ...
}
```

En ClienteSafeShop tenemos:

```
public void comprar(int monto) {
    if (monto > montoMaximo) {
        throw new BusinessException("Ha excedido el monto maximo");
    }
}
```

En ClientePromocion tenemos:

```
public void comprar(int monto) {
    if (monto > montoMinimo) {
          puntosAcumulados += puntosPremio;
    }
}
```

4) Cambiando el ángulo de la información…

Dos cosas interesantes para comentar:

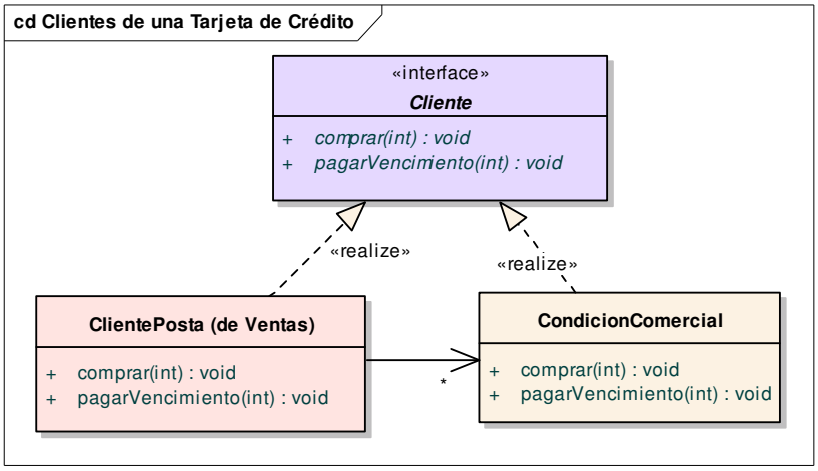
- Revisemos el código de ClientePosta nuevamente:

```
public void comprar(int monto) {
    for (CondicionComercial condicion : condicionesComerciales) {
          condicion.comprar(monto);
    }
    ...
}
```

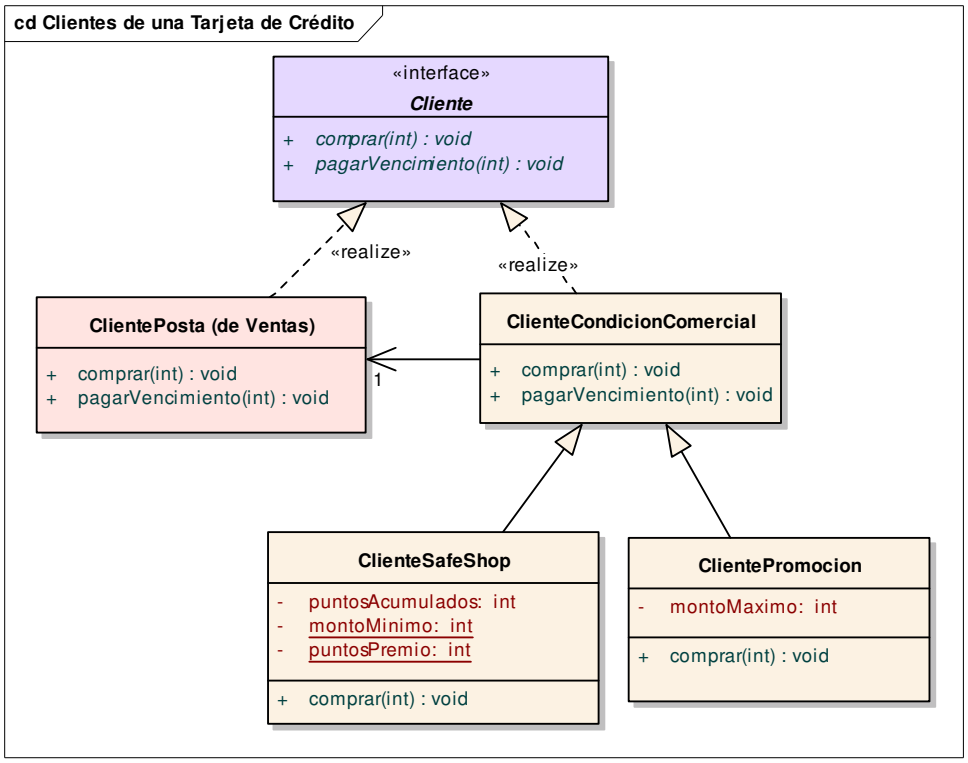
Fíjense que primero enviamos el mensaje comprar() a las condiciones comerciales porque la opción Safe Shop hace que yo tenga que cortar la compra. Si algunas condiciones comerciales trabajaran **después** de cada compra, entonces tendría que tener dos listas, una de pre-condiciones y otra de post-condiciones.

- Las condiciones comerciales definen un comportamiento polimórfico con respecto al ClientePosta (a ambos les puedo enviar un mensaje comprar()). Si bien no tiene gracia reemplazar al ClientePosta con un ClienteSafeShop, eso puede abrirnos una puerta para ver otra solución posible…

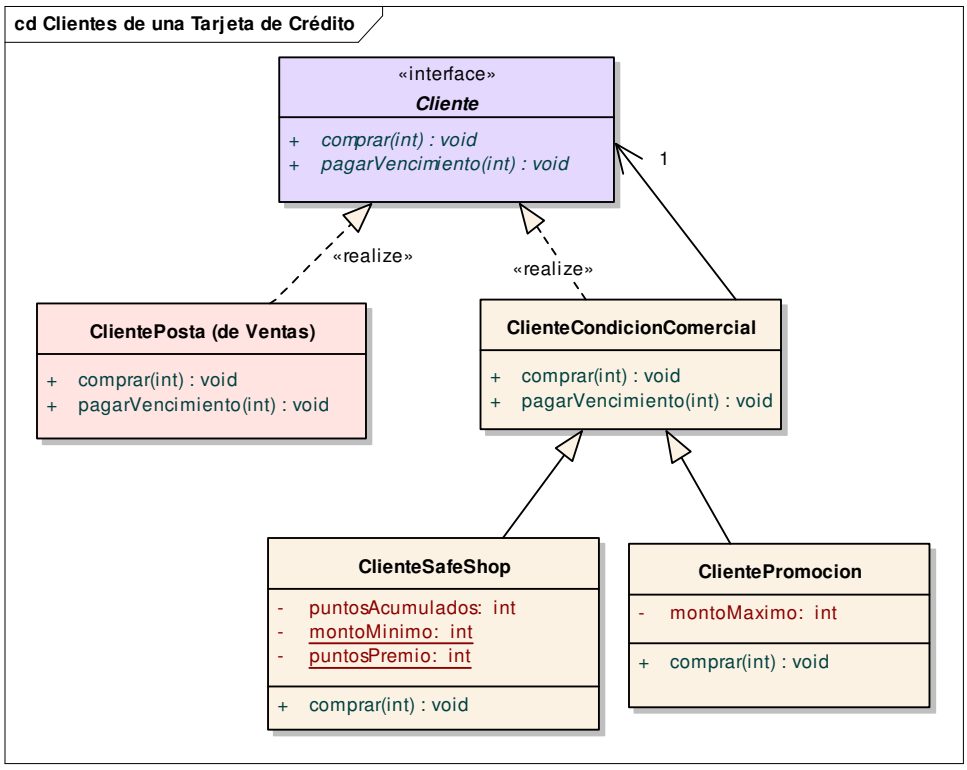
Si el ClientePosta y el ClienteSafeShop / ClientePromocion son polimórficos en el contexto en que el cliente compra:



¿Qué tal si cambiamos la óptica? O sea: que la condición comercial sea quien conozca al ClientePosta. Pero ya no estamos modelando una condición comercial, sino un Cliente con comportamiento agregado:



Este modelo así tiene una contra importante que ya vimos en la solución 2: no me permite que un cliente en Promoción se asocie al sistema Safe Shop. Pero podemos ajustar eso con un detalle: el ClienteCondicionComercial puede conocer un Cliente en lugar de conocer al ClientePosta:



Y lo que estamos haciendo con el cliente es **decorándolo** (vamos agregando comportamiento cuando compra).

Veamos cómo cambia el código en el caso del ClienteSafeShop:

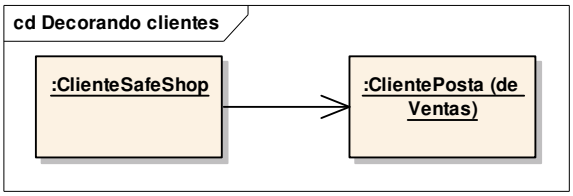
```
public void comprar(int monto) {
    if (monto > montoMaximo) {
        throw new BusinessException("Ha excedido el monto maximo");
    }
    cliente.comprar(monto);
}
```

Y el del ClientePromocion:

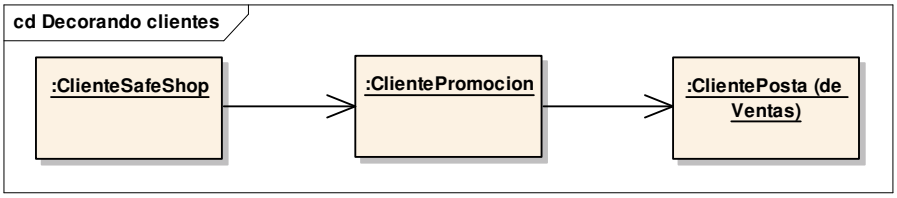
```
public void comprar(int monto) {
    cliente.comprar(monto);
    if (monto > montoMinimo) {
          puntosAcumulados += puntosPremio;
    }
}
```

El cliente es un **delegado**: no sabemos si será el ClientePosta o alguno de los clientes que tengan condiciones comerciales.
Sobre la solución podemos decir:

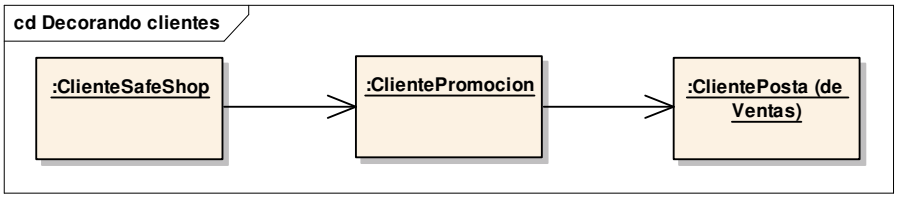
- El comprar de Safe Shop primero valida que no exceda el máximo de la compra y después delega al cliente, mientras que el ClientePromocion delega primero y después acumula los puntos, lo cual permite que la regla de negocio comprar falle y no se acumulen los puntos erróneamente. Esto es una ventaja respecto a la solución 3): podemos agregar comportamiento antes o después de la compra.
- El ClientePosta (de Ventas) no se ve afectado. Esto puede verse como una ventaja, ya que estoy agregando comportamiento a un objeto sin cambiarle una línea de código.
- Cada condición comercial es un decorador del cliente en el sentido en que es quien agrega comportamiento. El cliente es el decorado o componente.
- Un cliente Safe Shop se representa así:



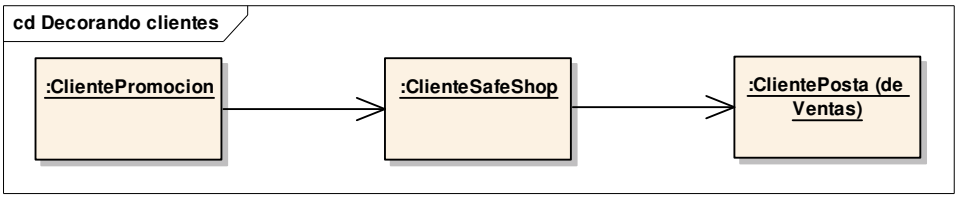
- Un cliente con Safe Shop y Promoción se representa así:



- En este ejemplo no parece tener mucho sentido, pero puedo intercambiar decoradores



o bien

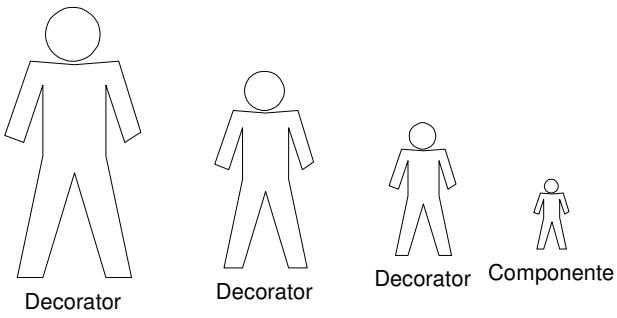


- Una desventaja de esta solución es que me obliga a implementar un método pagarVencimiento() que simplemente delega al cliente la resolución:

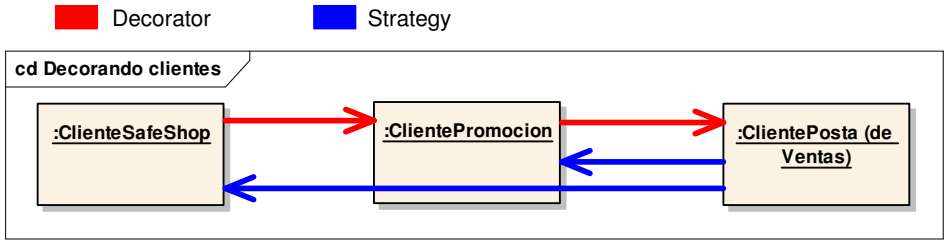
En ClienteCondicionComercial

```
public void pagarVencimiento(int monto) {
    cliente.pagarVencimiento(monto);
}
```

Entonces para usar esta idea, lo que conviene es que el Componente tenga una interfaz chica (reducida en cantidad de métodos):



- Tanto la técnica de decoración como la del Strategy nos permite agregar comportamiento en forma dinámica. La diferencia es que el Strategy trabaja desde el cliente hacia adentro y que el Decorator (así es el nombre del pattern) trabaja agregando "pieles" desde el objeto hacia fuera:



Esto trae consecuencias importantes: en mi aplicación tengo que instanciar Clientes decorados en lugar del ClientePosta, mientras que con la solución 3) el cliente conservaba su identidad (seguía siendo él).

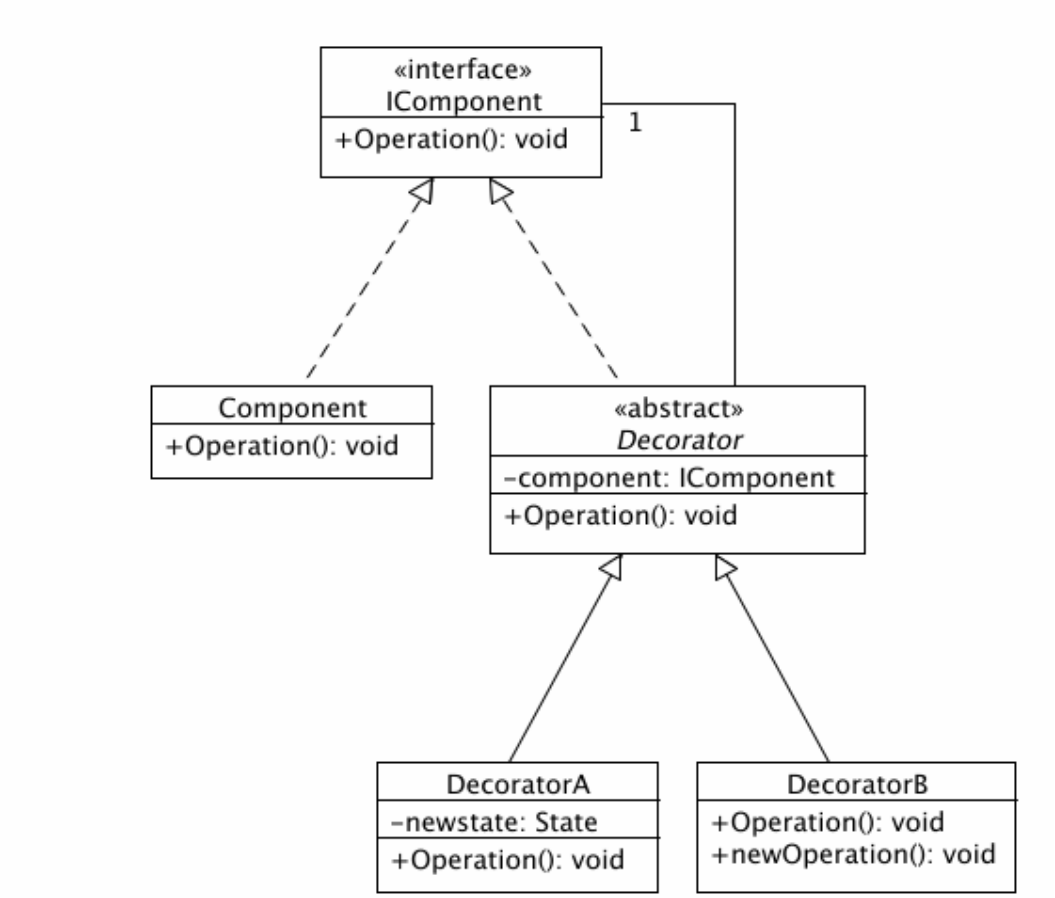
Entonces hay que tener cuidado cuando pregunto si dos clientes son los mismos (el decorado no es el componente).

- ¿cómo sería cómodo instanciar un cliente con SafeShop? Quizás estaría bueno poder pasarle el delegado en el constructor mismo:

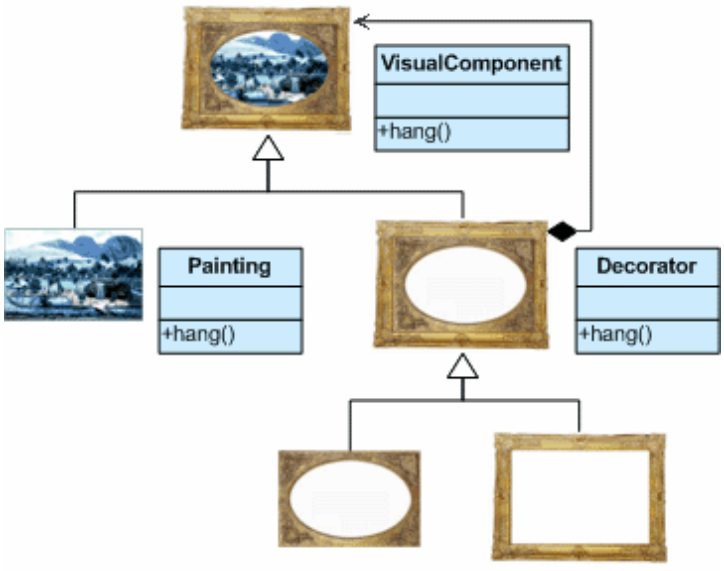
```
Cliente bertolini = new ClienteSafeShop(new Cliente("bertolini"));

Cliente amalita = new ClienteSafeShop(new ClientePromocion(15, 50,
new Cliente("bertolini")));
```

El Decorator de Libro



Intent: "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."



Metáfora asociada al Decorator

La matrioshka

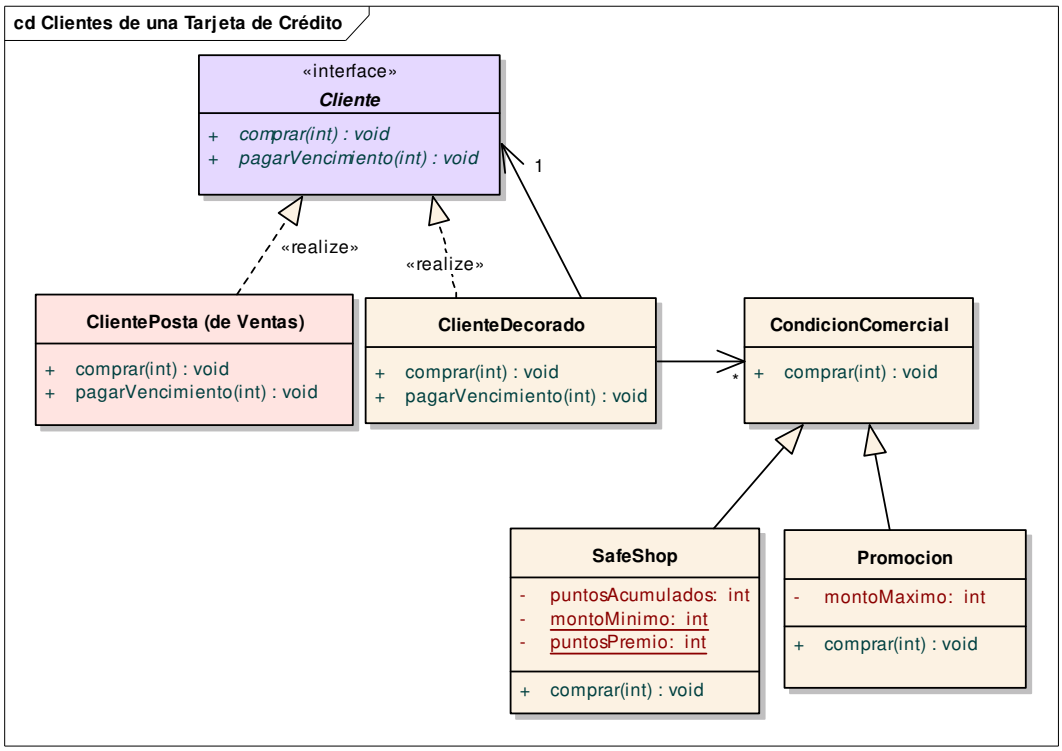
5) Un poco de todo

Ahora, si yo tengo los clientes decorados y quiero eliminar el servicio Safe Shop. ¿Cómo hago?

Como el Decorator me permite anidar decoraciones, no tengo idea si el Safe Shop va primero o después que los otros decoradores. Una solución posible es tener dos tipos de clientes:

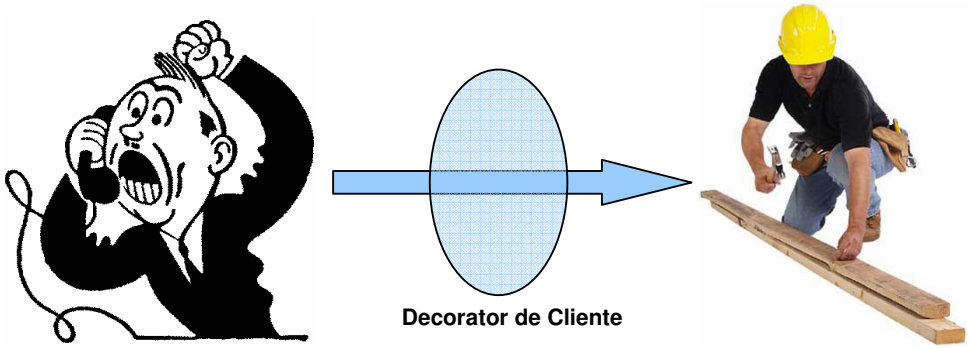
- Los clientes posta
- Los que usan condiciones comerciales

Es decir, decoramos al cliente posta con uno que tenga las condiciones comerciales (juntamos la solución 3 con la 4):



Lo que aprendimos: una nueva idea

El Decorator propone una alternativa distinta a lo que anteriormente veníamos trabajando: interponer a un objeto entre el que pide una cosa y el que la resuelve.



Un objeto que envía el mensaje comprar()

Decorator de Cliente

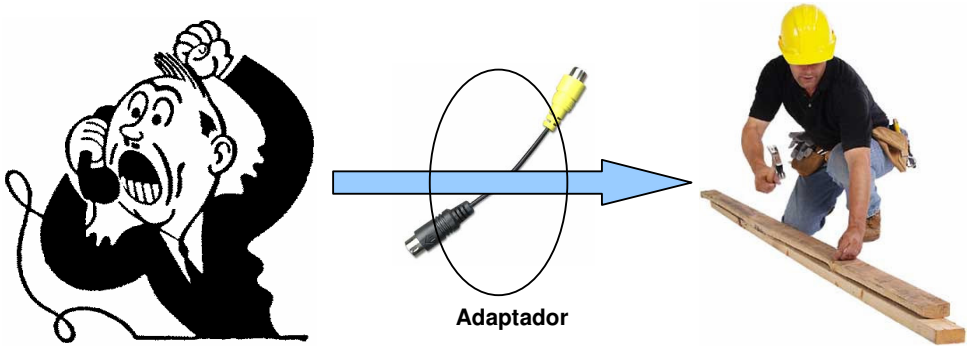
Cliente Posta

En el Decorator, el chiste está en que el decorado no se entera que lo decoran y tampoco el que envía el mensaje comprar(), porque el decorador y el componente son polimórficos.

Otra idea de diseño similar ocurre cuando hay incompatibilidades entre quien envía el mensaje y quien lo resuelve, ya sea:

- Porque el objeto posta tiene un mensaje con nombre diferente
- Porque el objeto posta necesita más información
- Porque queremos ofrecer una forma más cómoda de usar al objeto posta

Entonces lo que sucede es que en cualquiera de esos casos no me sirve tener objetos polimórficos, pero sí me sirve la misma idea de tener un intermediario que corrija esa interferencia para que el mensaje pueda llegar correctamente al objeto que lo tiene que responder:



Un objeto que envía un mensaje

Adaptador

Objeto Posta que resuelve el mensaje

```
Integer monto = ...;
objeto.buy(monto);
```

```
public void buy(Integer amount) {
    objetoPosta.comprar(amount.intValue());
}
```

```
public void comprar(int
```

Haya o no polimorfismo, en ambos casos se “envuelve” a un objeto para agregar o cambiar comportamiento. El objeto original no cambia su funcionalidad y aquí está el concepto nuevo: hay un intermediario que delega al objeto original y el que envía el mensaje (el objeto “cliente” o “requirente”) ya no habla con el objeto posta sino con este intermediario.