Soluciones posibles a Maco Wins

Introducción

Maco Wins es un problema inicial que nos sirve para introducirnos en el planteo de decisiones de diseño. En este apunte dejamos planteadas algunas posibles soluciones, además de algunos errores comunes.

Listado los requerimientos

Comenzaremos la resolución identificando qué requerimientos debe cumplir el sistema:

- Saber el precio de una prenda.
- Saber el tipo de una prenda.
- Registrar una venta.
- Saber las ganancias de un determinado día.

Atacaremos cada requerimiento por separado, lo que implicará **pensar alternativas**, siempre bajo el paradigma orientado a objetos, **compararlas** cualitativamente en base a su diseño y finalmente **elegir** una.

Para esto es necesario bajar en detalle las ideas, así que **escribir código, hacer anotaciones y dibujar diagramas** son herramientas que nos ayudarán con esta tarea.

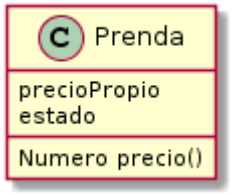
Primer requerimiento: saber el precio de una prenda

Nos dicen que el precio depende de un precio propio de la prenda y su *estado*. Además podemos observar que el cálculo a realizar es distinto según el estado de la prenda, lo que se traduce a que **buscaremos un *comportamiento distinto* del sistema en base al estado de la prenda.**

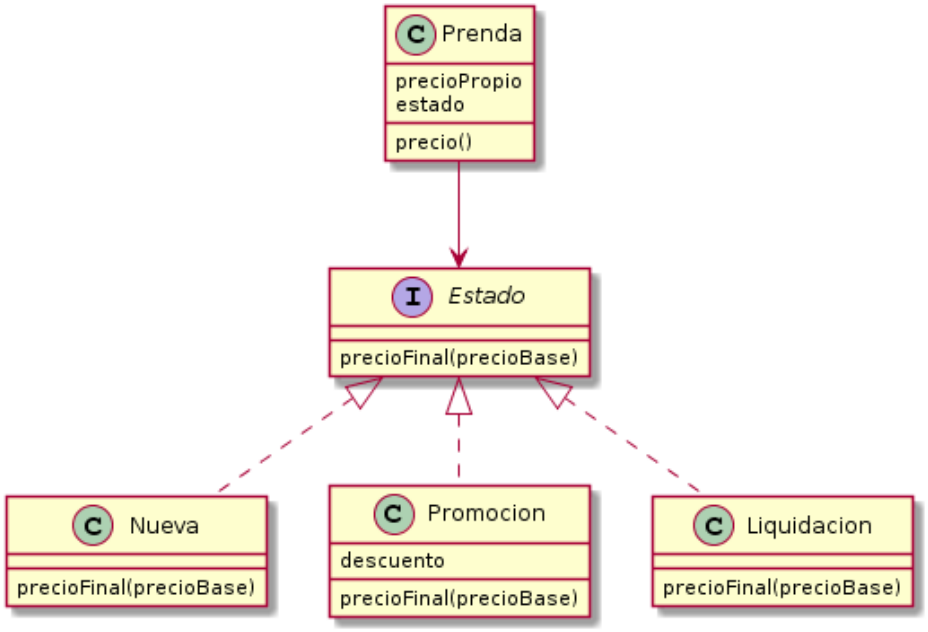
Con esto ya podemos definir algunas cosas:

- Cada prenda tiene asociado un precio propio y un estado,
- El precio de la prenda se calcula a partir de ello,
- Vamos a tener muchas prendas en el sistema.

Como punto de partida, tomaremos a la clase Prenda y su diagrama de clases:



Ahora queda ver cómo modelar los posibles estados. Una primera opción sería plantear estados polimórficos, usando composición:



El código hasta acá nos queda así:

```
clase Prenda
  metodo precio
    retornar estado.precioFinal(precioPropio)

clase Nueva
  metodo precioFinal(precioBase)
    retornar precioBase

clase Promocion
  metodo precioFinal(precioBase)
    retornar precioBase - descuento

clase Liquidacion
  metodo precioFinal(precioBase)
    retornar precioBase * 0.5
```

Esta solución propone modelar cada **estado como un objeto** para usarlos *polimórficamente* desde la prenda, *delegando* el cálculo del precio.

Como no hay lógica común entre los distintos estados, *no hay necesidad de definir una superclase* (abstracta), sino que todos ellos **comparten la misma interfaz**, lo que quiere decir que entienden el mismo conjunto de mensajes.

Esta **estructura** de solución basada en polimorfismo y composición, para permitir que un *algoritmo* (el cálculo del precio de la prenda) se pueda resolver con ciertas variaciones *intercambiables* (los estados) de forma transparente para sus *objetos clientes (*las prendas*)* se lo conoce como estrategia o *strategy.*

En otras palabras, la intención de un *strategy* es la siguiente:

Definir una familia de algoritmos, encapsulando a cada uno, y haciéndolos intercambiables. Un strategy permite al algoritmo variar de forma independiente de los objetos clientes que lo usan.[1]

> Nota de implementación: en lenguajes con tipado estático y nominal como Java, las interfaces deben ser declaradas, mediante un *interface...*
>
> ```
> public interface Estado {
>    double precioFinal(double precioBase);
> }
> ```
>
> ... y quienes la implementan, también declararlo explícitamente:

[1] Traducción libre de https://sourcemaking.com/design_patterns/strategy

```
public class Nueva implements Estado {
   ....
}
```

Por otro lado, solamente en el caso de una prenda en promoción es necesario indicar el monto a descontar, por lo que decidió que sea un atributo de dicho objeto.

Por lo tanto **en el sistema existirán, al menos, tantos objetos Promocion como descuentos se necesiten**; o sea, si dos prendas en prendas en promoción tienen descuentos distintos (100 y 200 pesos por ejemplo) necesitaremos dos instancias de `Promocion` distintas.

Los otros estados posibles para una prenda, al no tener estado interno (stateless), pueden tanto ser compartidos entre todas las prendas como no.

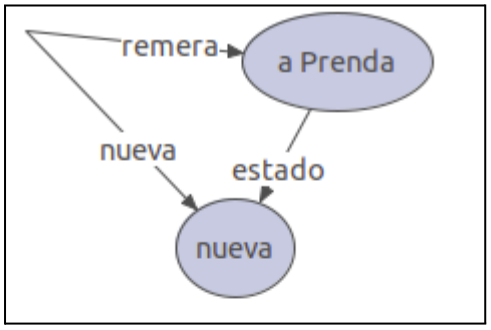
Nota de implementación: en algunos lenguajes como Wollok estos objetos stateless compartidos pueden ser declarados directamente como un objeto global, mediante un object.

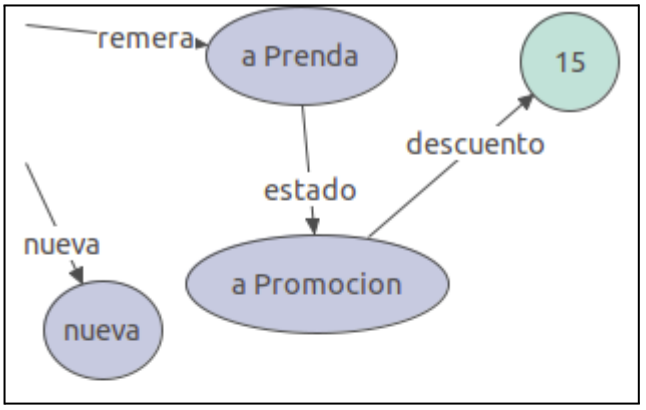
```
object liquidacion {
   ...
}
```

Este no es sin embargo el caso de Java, y deberá ser declarado como class.

```
public class Liquidacion implements Estado {
   ...
}
```

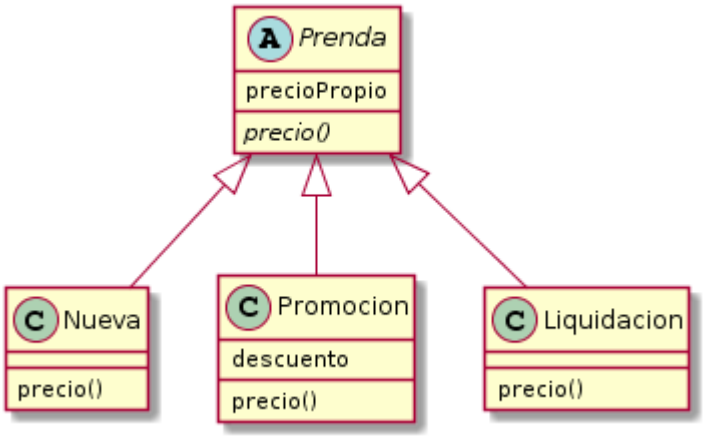
Una consecuencia positiva de haber utilizado composición entre la Prenda y el Estado, es que es trivial cambiar el estado de la misma, solamente hay que cambiar de referencia.





Alternativa: usando herencia

Otra opción para conseguir este comportamiento distinto en base al estado de una prenda en el paradigma de objetos es utilizando **herencia** para crear 3 subclases de Prenda.



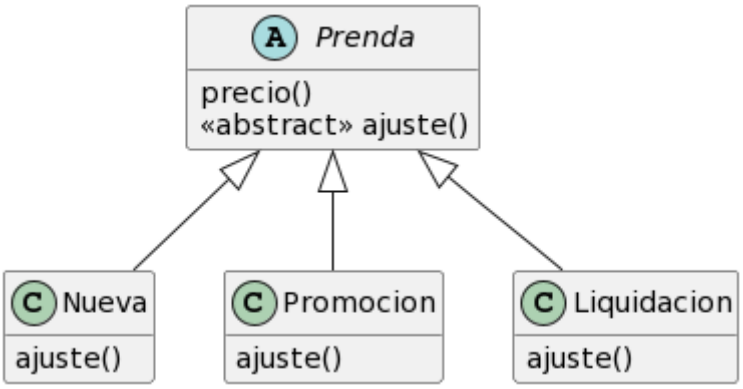

```
clase abstracta Prenda
  metodo abstracto precio()
```

```
clase Nueva
  metodo precio()
    retornar precioPropio
```

```
clase Promocion
  metodo precio()
    retornar precioPropio - descuento
```

```
clase Liquidacion
  metodo precio()
    retornar precioPropio * 0.5
```

Variante: con clase abstracta y método concreto (plantilla)



```
clase abstracta Prenda
  metodo precio()
       return precioPropio +
this.ajuste()

  metodo abstracto ajuste()

clase Nueva
  metodo ajuste()
    retornar 0
```

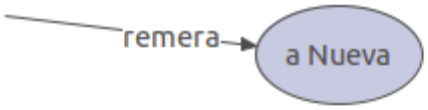
```
clase Promocion
  metodo ajuste()
    retornar -descuento

clase Liquidacion
  metodo ajuste()
    retornar - precioPropio/2
```

En esta alternativa, es el propio objeto `Prenda` el que está asociado al algoritmo para determinar el precio, dependiendo de qué clase fue instanciada.

Si bien esta alternativa puede resultar **más simple**, ya que el método *lookup* es capaz de determinar el algoritmo a ejecutar sin necesidad de delegar en otro objeto, introduce una relación **más rígida** en el diseño:

- Al ser una relación *estática*, **las prendas no pueden cambiar de estado** (por lo menos ya no de una manera simple), ya que un objeto nace y muere como instancia de una sola clase (no se puede cambiar de clase). Dado el contexto, probablemente se requiera este comportamiento del sistema, en ese caso ya sería **inviable**.

- Es un **arma de un solo tiro** (siempre trabajaremos con *herencia simple*). Por lo que esta técnica no serviría si tenemos que sub-clasificar por algún otro atributo, como el tipo de prenda.



También cabe destacar que Prenda pasaría a ser una clase **abstracta** ya que no existe ninguna definición del precio de una prenda “sin estado”.

Aclaración: En esta alternativa, **Prenda** queda como *clase* por fines didácticos, pero deberíamos analizar (luego de resolver el resto de los requerimientos) si existe algún comportamiento general para todas las prendas que requiera estar en esa clase. Si esto no ocurre, entonces conviene que sea una *interfaz*.

Nota de implementación: en lenguajes con tipado estático y nominal como Java, además, si queremos usar las subclases polimórficamente es necesario declarar los métodos como abstractos...

```
public abstract class Prenda {
	protected double precioPropio;
	public abstract double precio();
}
```

... y ser sobre-escrito en las subclases:

```
public class Nueva extends Prenda {
	@Override
	public double precio() {
		return precioPropio;
	}
}
```

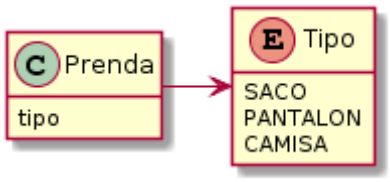
Segundo requerimiento: saber el tipo de una prenda

El análisis no muestra ninguna lógica relacionada con los tipos de una prenda, solamente nos aclara que pueden ser tres.

Modelamos esto como un atributo más de la Prenda, teniendo un conjunto acotado de posible valores *enumerados*, es decir, pueden tomar valores dentro de un conjunto que vamos a definir. Una posible forma de resolverlo es utilizando *strings*: "saco", "pantalón", etc.

Esto tiene la ventaja de poder ir modificando los tipos sin necesidad de cambiar el código (si mañana incorporamos "blusa" al negocio no tenemos que hacer nada). Pero, perdemos control sobre la normalización[2] de los datos (“Saco”, “saco”, “SACO”, “_saco_”), lo que dejaría de ser útil para usarlo como identificador.

Una opción mejor es en su lugar valores enumerados, o como se conocen en muchos lenguajes, *enums:*



Nota de implementación: en Java esta enumeración podremos declararla mediante *enum.*

public enum Tipo {

[2] es decir, asegurarnos que siempre que nos refiramos a un saco, usemos el string "saco" y no otra de sus variantes

```
   SACO, PANTALON, CAMISA
}
```

Ojo, ¡los enums no son strings! Por ejemplo:

```
Tipo unTipo = Tipo.SACO; // OK
unTipo = Tipo.PANTALON; //OK
unTipo = "PANTALON"; // ROMPE, porque los enums NO son strings
```

¿Y que son entonces? Simplemente objetos con un tipo propio, incluso diferente al de otros enums que declaramos. Por ejemplo si después declaramos:

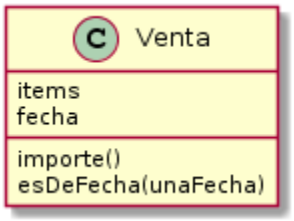
```
public enum Animales {
   GATO, PERRO, MANCUSPIA
}

Tipo unTipo = Animales.MANCUSPIA; // ROMPE, porque MANCUSPIA
                                  //es de tipo Animal, no de tipo Tipo
```

Más adelante veremos que los enums, como son objetos, pueden tener comportamiento, al igual que los WKO de Wollok.

Tercer requerimiento: registrar una venta

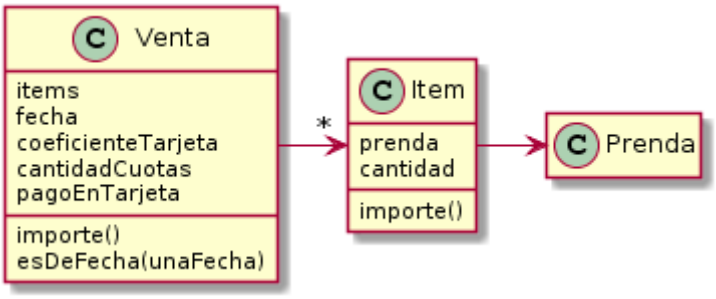
Ahora queremos poder registrar una venta, lo que implica guardar una venta en algún lugar. Para esto hay que definir primero *qué* es una venta

El análisis informa que una venta sabe su fecha y contiene prendas, cada una con su cantidad. Entonces introduzcamos las *ventas*:



Y luego nos queda modelar esa asociación entre prenda vendida y cantidad, que llamaremos item[3]:

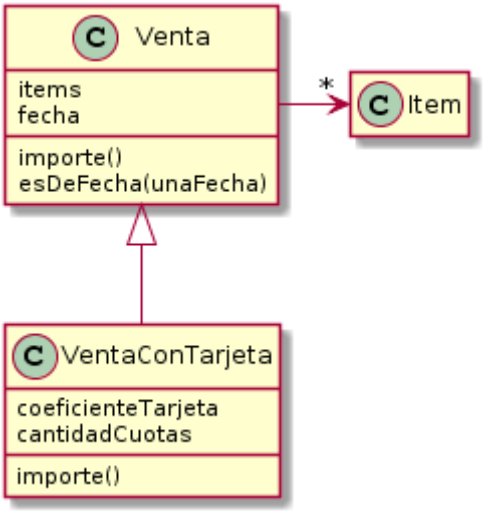
[3] En general Item es nombre demasiado genérico, pero en el dominio de la facturación tiene el sentido específico de una agrupación elemento-cantidad. Por eso lo elegimos.



Pero no todo termina acá:

*"Las ventas pueden ser en efectivo o con tarjeta. En el caso que sea con tarjeta, tienen el mismo comportamiento que en efectivo (el cual no modifica el precio), solo que se le aplica un recargo según la cantidad de cuotas seleccionadas (cantidad de cuotas * un coeficiente fijo + 0.01 del valor de cada prenda)."*

¿Cómo modelar esta situación? Dado que las ventas son o bien en efectivo o bien en tarjeta, y que una vez realizadas no pueden cambiar su medio de pago, sería posible utilizar **herencia** y **redefinición**: tomaremos a las ventas en efectivo como nuestro caso base, y a las ventas en tarjetas como redefiniciones de la venta:



De esta forma, logramos separar el comportamiento y estado de una venta con tarjeta a otra clase:

```
clase Venta
  metodo importe()
    retornar items.sum(item -> item.importe())

clase  VentaConTarjeta
  metodo importe()
    retornar coeficienteTarjeta * cantidadCuotas + 0.01 * super() + super()
    // o también, en una sola llamada:
    // retornar (coeficienteTarjeta * cantidadCuotas + 1.01) * super()
```

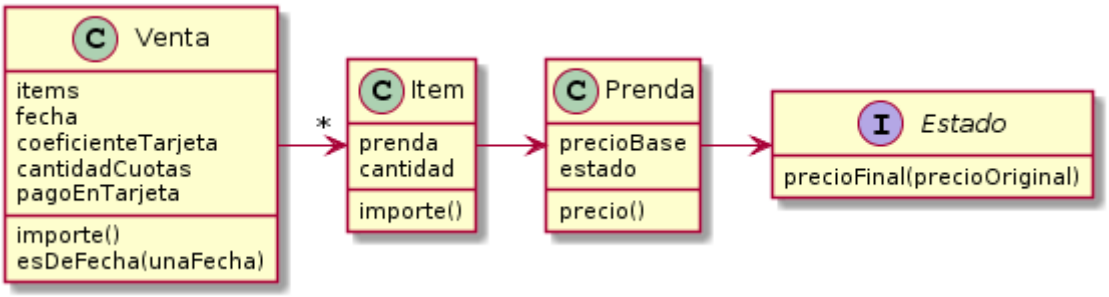
Y ahora que tenemos nuestras ventas, ¿cómo las registramos? Una primera idea (correcta) es contar con una colección de ventas, y agregar una venta particular a esta colección:

```
ventas.add(venta)
```

¿Pero dónde irá esta colección de ventas? Avancemos al siguiente requerimiento para contar con más elementos para decidir.

Alternativa: no modelar los tipos de venta

Una alternativa al modelado anterior consiste, en vez de utilizar herencia para los dos tipos de ventas que tenemos, emplear un booleano.



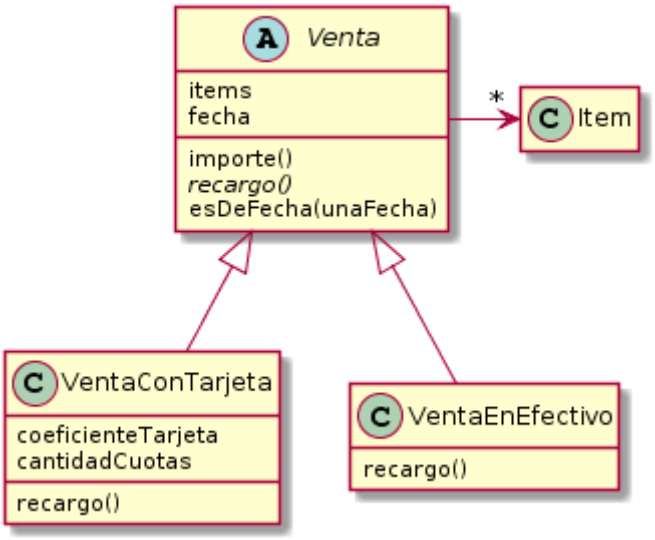

```
clase Venta
  metodo importe()
    importe = costoItems()
    if (pagoEnTarjeta)
      importe += importe * 0.01 + cantidadCuotas * coeficienteTarjeta
    retornar importe
```

Esta solución, como se observa, es muy **simple**: no necesitamos una clase que represente un subtipo de ventas. Pero esa es también su debilidad: es difícil incorporar nuevos subtipos (como una venta con *MercadoPago*), al no tener clases particulares para cada uno de ellos. Decimos que esta alternativa es menos **extensible,** porque es difícil agregar nuevas funcionalidades.

Otro problema de esta solución es que ahora una venta tiene atributos específicos de una venta con tarjeta y código para el cálculo del recargo, aún cuando se trata de una venta en efectivo. Esto nos habla de que la clase venta así modelada es poco **cohesiva:** tiene comportamiento y estado que no le pertenece y que no tiene que ver con su tarea principal.

Alternativa: dos subtipos de ventas

Si la alternativa anterior tenía el problema de *perder abstracciones* (el concepto de la venta con tarjeta desaparecía), podríamos intentar ir ahora por el camino opuesto: crear una abstracción para la venta en efectivo, y otra para venta con tarjeta:



Nuevamente el método importe será `concreto`, pero en este caso utilizará como parte de su definición un método abstracto `recargo`. Además, la venta ahora será abstracta, dado que no la instanciaremos directamente.

```
clase abstracta Venta
  metodo importe()
    retornar recargo(items.sum(item -> item.importe()))

clase  VentaConTarjeta
  metodo recargo(precioBase)
    retornar precioBase * (coeficienteTarjeta * cantidadCuotas + 1.01)

clase  VentaEnEfectivo
  metodo recargo(precioBase)
    retornar precioBase
```

Este uso particular de la herencia, en el cual tenemos una clase abstracta, que define un método concreto que en tanto utiliza un método abstracto definido en subclases concretas es muy frecuente, y se la conoce como *plantilla*, y al método en cuestión (en este caso, el `importe`), *método plantilla.*

En otras palabras, la intención de un método plantilla o *template method* es la siguiente:

Definir el esqueleto de un algoritmo, delegando algunas partes a subclases. Un método plantilla le permite a la subclases definir ciertos pasos del algoritmo sin cambiar su estructura. Para ello, la clase base declara huecos en el algoritmo que las subclases deben llenar.[4]

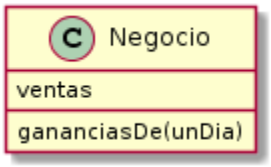
Cuarto requerimiento: saber las ganancias del día

Para saber las ganancias del día, vamos a pararnos, arbitrariamente[5], sobre la solución basada en herencia y redefinición. Este requerimiento es bastante sencillo de implementar, dado que consiste en hacer una sumatoria de los importes de las ventas del día: *un filter más un sum*.

```
ventas.filter(venta -> venta.esDeFecha(fecha)).sum(venta -> venta.importe())
```

El problema es dónde ubicar este método. ¿En la venta? No parece una gran idea, dado que es un comportamiento para un *conjunto* de ventas. ¿En la clase `Collection`? No parece tampoco el mejor camino a seguir, dado que sólo funcionaría para colecciones de ventas y no de por ejemplo, números. Además, en la mayoría de los lenguajes no es posible modificar clases ya definidas.

Entonces lo que necesitamos es una nueva abstracción (objeto) que represente al conjunto de ventas, a la unidad que vende y registra: al *negocio* o *tienda*.



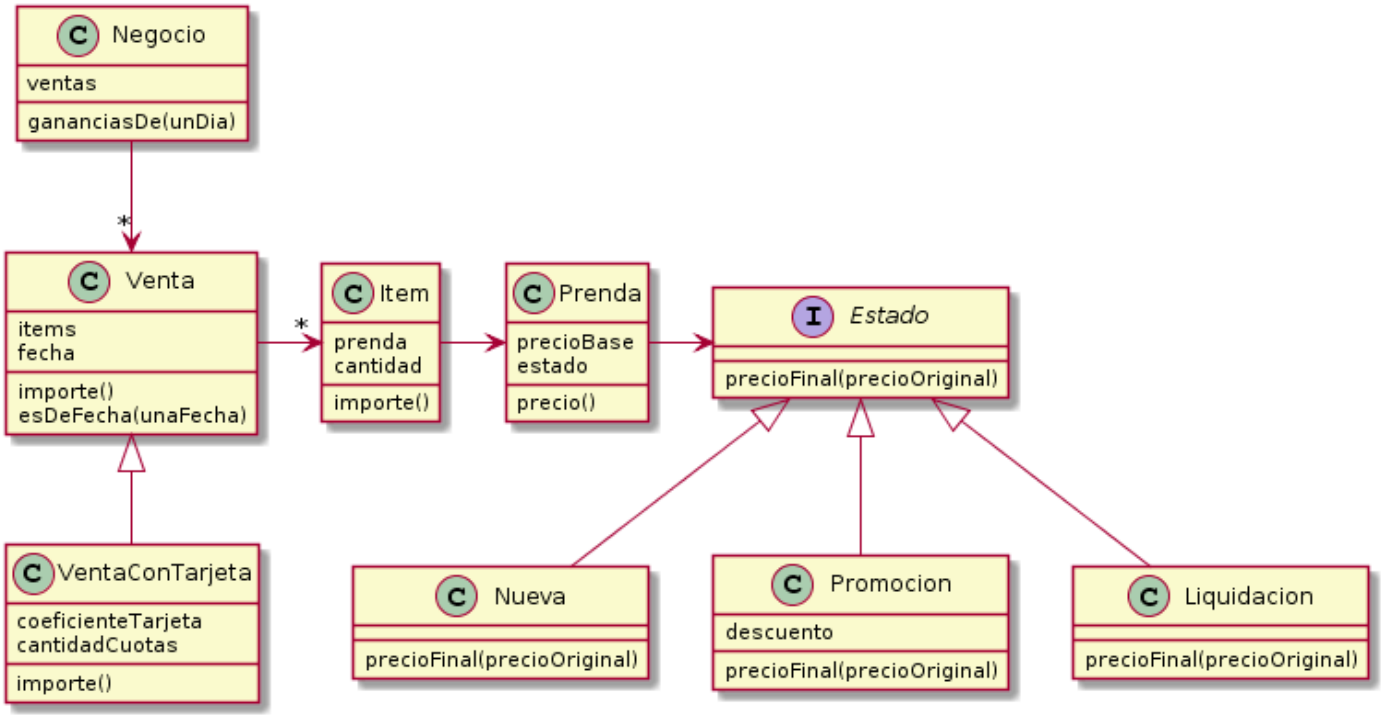
Ahora que aparece el negocio, y dado que este representa al conjunto de ventas que se efectúan, podemos cerrar el requerimiento que nos quedó por la mitad: registrar una venta.

```
clase Negocio
  metodo registrarVenta(venta)
    ventas.add(venta)
```

[4] Traducción libre de https://sourcemaking.com/design_patterns/template_method
[5] Es decir, sin ningún motivo particular, para poder avanzar con la explicación

Juntando todo

Si unimos todas las partes, nos quedará la siguiente solución:




```
clase Prenda
  metodo precio
    retornar estado.precioFinal(precioBase)

clase Nueva
  metodo precioFinal(precioOriginal)
    retornar precioOriginal

clase Promocion
  metodo precioFinal(precioOriginal)
    retornar precioOriginal - descuento

clase Liquidacion
  metodo precioFinal(precioOriginal)
    retornar precioOriginal * 0.5

clase Item
  metodo importe()
    retornar prenda.precio() * cantidad
```

```
clase Venta
  metodo importe()
    retornar items.sum(item -> item.importe())

clase  VentaConTarjeta
  metodo importe()
    retornar coeficienteTarjeta * cantidadCuotas + 0.01 * super() + super()

clase Negocio
  metodo gananciasDe(fecha)
    retornar ventas
      .filter(venta -> venta.esDeFecha(fecha))
      .sum(venta -> venta.importe())
```