```
//Requerimiento 1

class Prenda{
  Estado estado;
  Double precioPropio;

  precio(){
    return estado.precioFinal(precioPropio)
  }
}

//Composicion
interface Estado {
  abstract Double precioFinal(Double precioBase);
}

class Nueva extends Estado{
  Double precioFinal(Double precioBase){
     return precioBase;
  }
}

class Promocion extends Estado{
  Double descuento;

  Double precioFinal(Double precioBase){
     return precioBase - descuento;
  }
}

class Liquidacion extends Estado{
  Double precioFinal(Double precioBase){
     return precioBase * 0.5;
  }
}

ESTRATEGIA/STRATEGY-> Es la idea/estructura/semántica
--> Objetos polimorficos
--> Se pueden cambiar en el tiempo
--> Delego un algoritmo especifico

//Herencia

abstract class Prenda {
  Double precioPropio;

  abstract Double precio()
}

class PrendaNueva extends Prenda{
  Double precio(){
     return precioPropio;
  }
}
```

```
class PrendaEnPromocion extends Prenda{
  Double descuento;

  Double precio(){
     return precioPropio - descuento;
  }
}

class PrendaEnLiquidacion extends Prenda{
  Double precio(){
     return precioPropio * 0.5;
  }
}

// Requerimiento 2

//String
class Prenda{
  String tipo;
}

new Prenda("Saco")
new Prenda("Campera")
new Prenda("Chaqueta")
new Prenda("Tapado")

//Enum
class Prenda{
  TipoPrenda tipo;
}

new Prenda(TipoPrenda)

public enum TipoPrenda {
   SACO, PANTALON, CAMISA
}

//Requerimiento 3

//Con Item
class Venta {
  List<Item> items;

  Double importe(){
    return items.sum(item -> item.importe())
  }
}

class Item {
  Integer cantidad;
  Prenda prenda;

  importe(){
```

```
    return prenda.precioFinal() * cantidad
  }
}

//Sin Item
class Venta{
  List<Prenda> prendas;

  Double importe(){
    return prendas.sum(item -> prenda.precioFinal())
  }
}

// Con booleano
class Venta{
  List<Item> items;
  Integer cantidadCuotas;
  Double coeficienteTarjeta;
  Boolean esConTarjeta;

  Double importeFinal(){
    if(esConTarjeta){
      return importe() * 0.01 +
       cantidadCuotas * coeficienteTarjeta
    } else {
      return importe()
    }
  }
}

//Con Super
class Venta{
  List<Item> items;

  Double importe(){
    return items.sum(item -> item.importe())
  }
}

class VentaTarjeta extends Venta{
  Integer cantidadCuotas;
  Double coeficienteTarjeta;

  @Override
  Double importe(){
    return coeficienteTarjeta * cantidadCuotas
    + 0.01 * super() + super()
  }
}

//Con dos subclases
class Venta{
  List<Item> items;
```

```
  abstract Double conRecargo(importeBase);

  Double importe(){
    return conRecargo(
      items.sum(item -> item.importe())
    )
  }
}

class VentaTarjeta extends Venta{
  Integer cantidadCuotas;
  Double coeficienteTarjeta;

  @Override
  Double conRecargo(importeBase){
    return importeBase *
      (coeficienteTarjeta * cantidadCuotas + 1.01)
  }
}

class VentaEfectivo extends Venta{
  @Override
  Double conRecargo(importeBase){
    return importeBase;
  }
}

METODO PLANTILLA
TEMPLATE METHOD
--> Define un comportamineto general
--> Permite que subclases lo completen

//Requerimiennto 4

class TiendaDeRopa {
  List<Venta> ventas;

  void registrarVenta(Venta venta){
    ventas.add(venta)
  }

  Double gananciaDia(){
    return ventasDelDia()
      .sum(venta -> venta.importe())
  }

  List<Venta> ventasDelDia(fecha){
    return ventas
      .filter(venta -> venta.esDeFecha(fecha))
  }
}
```

ESCALABILIDAD != EXTENSIBILIDAD