Refactorizaciones en pequeños pasos, haciendo un commit en git, luego de la misma.

- Refactorizando funcion aniadir, ya que esta no está haciendo lo que dice hacer, no aniade productos, solo calcula total, esto mejorará la cohesion de la función. Refactorizacion: la funcion aniadir recibe un product y la cantidad

```
def aniadir(producto, cantidad)
    @total = @total + producto.cuanto_por(cantidad);
end
```

- Refactorizando funcion aniadir, ya que la clase producto no cumple con encapsulamiento ya que tiene acceso a atributos de la misma. Refactorizacion: la funcion aniadir un producto calcula es costo deacuerdo a la cantidad. Esto mejorara la encapsulación y la abstracción de la clase.

```
def aniadir(producto, cantidad)
    @total = @total + producto.cuanto_por(cantidad);
end
```

```
it "devuelve el total para una venta con un servicio y un articulo" do
    @producto_articulo = Producto.new("articulo","computadora", 300);
    @producto_servicio = Producto.new("servicio", "mantenimiento", 2);
    venta = Venta.new();
    venta.aniadir(@producto_servicio, 3);
    venta.aniadir(@producto_articulo, 2);
    venta.calcular_total.should == 601.2
end
```

- Refactorizando funcion aniadir, ya que la clase producto no cumple con encapsulamiento ya que tiene acceso a atributos de la misma. Refactorizacion: la funcion aniadir un producto calcula es costo deacuerdo a la cantidad. Esto mejorara la encapsulación y la abstracción de la clase.

```
it "devuelve el total de un producto tipo articulo con relacion a su precio y la cantidad " do
    @producto_articulo = Producto.new("articulo","teclado", 2);
    @producto_articulo.cuanto_por(2).should == 4;
end
```

```
it "devuelve el total para una venta con un producto servicio y un producto articulo aniadidos a una venta" do
    @producto_articulo = Producto.new("articulo","computadora", 300);
    @producto_servicio = Producto.new("servicio", "mantenimiento", 2);
    venta = Venta.new();
    venta.aniadir(@producto_servicio, 3);
    venta.aniadir(@producto_articulo, 2);
    venta.calcular_total.should == 601.2
end
```

- Refactorización Clase Producto Metodo cuanto_por:
    - Mejorar el nivel de cohesion, acoplamiento y abstracción y aplicando el concept de “tell don’t ask”
    - Refactorización: simplificar la tarea del método delegando funciones.

```
def cuanto_por(cantidad)
    if @tipo=="articulo"
        return cuanto_por_articulo(cantidad)
    else
        return cuanto_por_servicio(cantidad)
    end
end
```

```
it "devuelve el total de un producto tipo servicio con relacion a su precio y la cantidad " do
    @producto_servicio = Producto.new("servicio", "mantenimiento", 70);
    @producto_servicio.cuanto_por(2).should == 28;
end
```

- Implementado en la Clase Producto Metodo cuanto_por_articulo:
    - Delegando responzabilidades; que cada método no se ocupe de muchas cosas
    - Refactorización: simplificar la tarea del método delegando funciones. Refactorizando metodo cuanto_por

```
def cuanto_por_articulo(cantidad)
    return @precio*cantidad
end
```

```
it "devuelve el total de un producto articulo con relacion a su precio y la cantidad" do
    @producto_articulo = Producto.new("articulo","teclado", 2);
    @producto_articulo.cuanto_por_articulo(2).should == 4;
end
```

- Implementado en la Clase Producto Metodo cuanto_por_servicio:
    - Delegando responzabilidades; que cada método no se ocupe de muchas cosas
    - Refactorización: simplificar la tarea del método delegando funciones. Refactorizando metodo cuanto_por

```
def cuanto_por_servicio(cantidad)
    return @precio*cantidad*@@FACTOR_SERVICIO
end
```

```
it "devuelve el total de un producto servicio con relacion a su precio y la cantidad " do
    @producto_servicio = Producto.new("servicio", "mantenimiento", 70);
    @producto_servicio.cuanto_por_servicio(2).should == 28;
end
```

- Refactorización en la Clase Producto creando variable estática @@FACTOR_SERVICIO:
    - Evitar numerous mágicos