Ejercicio 4 Refactoring

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private String formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, String formaPago)
        if (!"efectivo".equals(formaPago)
                && !"6 cuota".equals(formaPago)
                && !"12 cuotas".equals(formaPago)) {
            throw new Error("Forma de pago incorrecta");
        }
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    public double getCostoTotal() {
        double costoProductos = 0;
        for (Producto producto : this.productos) {
            costoProductos += producto.getPrecio();
        } // reinventando la rueda
        double extraFormaPago = 0;
        if ("efectivo".equals(this.formaPago)) { // conditional
            extraFormaPago = 0;
        }
        else if ("6 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.2;
        }
        else if ("12 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.5;
        }
        int aniosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), Lo
        // Aplicar descuento del 10% si el cliente tiene más de 5 años de antigued
        if (aniosDesdeFechaAlta > 5) {
```

```
            return (costoProductos + extraFormaPago) * 0.9;
        }
        return costoProductos + extraFormaPago;
    }
}

public class Cliente {
    private LocalDate fechaAlta;

    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}

public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```

Refactoring: Replace Loop With Pipeline. Mecánica:

- Se identifican los lugares en los que se esté reinventando la rueda con For.
- Se reemplaza el código con un stream.

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private String formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, String formaPago)
        if (!"efectivo".equals(formaPago)
                && !"6 cuota".equals(formaPago)
                && !"12 cuotas".equals(formaPago)) {
            throw new Error("Forma de pago incorrecta");
        }
```

```
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    public double getCostoTotal() {
        double costoProductos = this.productos.stream
            .mapToDouble(p → p.getPrecio())
            .sum();

        double extraFormaPago = 0;
        if ("efectivo".equals(this.formaPago)) { // conditional
            extraFormaPago = 0;
        }
        else if ("6 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.2;
        }
        else if ("12 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.5;
        }

        int aniosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), Lo
        // Aplicar descuento del 10% si el cliente tiene más de 5 años de antigued
        if (aniosDesdeFechaAlta > 5) {
            return (costoProductos + extraFormaPago) * 0.9;
        }
        return costoProductos + extraFormaPago;
    }
}

public class Cliente {
    private LocalDate fechaAlta;

    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}
```

```
public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```

Refactoring: Replace Conditional with Polymorphism.

Mecánica:

- Se crea la jerarquía de clases.
- Hay que aplicar un Extract Method del condicional, ya que forma parte de un método grande.
- Se crean los métodos que sobreescriben al método que tiene el condicional.
- Una vez testeado y funcionando se elimina el condicional.

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private FormaPago formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, FormaPago formal
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    private double costoProductos() {
        return this.productos.stream
            .mapToDouble(p → p.getPrecio())
            .sum();
    }

    public double getCostoTotal() {
        int aniosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), Lo
```

```
        if (aniosDesdeFechaAlta > 5) {
            return (this.costoProductos + this.extraFormaPago) * 0.9;
        }
        return this.costoProductos + this.extraFormaPago;
    }

    private double extraFormaPago() {
            return this.calcularExtra(this.costoProductos());
     }
}

public class FormaPago {
    public double calcularExtra() {
        if ("efectivo".equals(this.formaPago)) {
            extraFormaPago = 0;
        }
        else if ("6 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.2;
        }
        else if ("12 cuotas".equals(this.formaPago)) {
            extraFormaPago = costoProductos * 0.5;
        }
}

public class Efectivo extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return 0;
    }
}

public class SeisCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.2;
    }
}

public class DoceCuotas extends FormaPago {
```

```
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.5;
    }
}
///
public class Cliente {
    private LocalDate fechaAlta;

    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}

public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private FormaPago formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, FormaPago formal
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    private double costoProductos() {
        return this.productos.stream
            .mapToDouble(p → p.getPrecio())
            .sum();
    }
```

```
    public double getCostoTotal() {
        int aniosDesdeFechaAlta = Period.between(this.cliente.getFechaAlta(), Lo

        if (aniosDesdeFechaAlta > 5) {
            return (this.costoProductos + this.extraFormaPago) * 0.9;
        }
        return this.costoProductos + this.extraFormaPago;
    }

    private double extraFormaPago() {
            return this.calcularExtra(this.costoProductos());
      }
}

public abstract class FormaPago {
    public abstract double calcularExtra(double costoProductos);
}

public class Efectivo extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return 0;
    }
}

public class SeisCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.2;
    }
}

public class DoceCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.5;
    }
}
///
public class Cliente {
    private LocalDate fechaAlta;
```

```
    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}

public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```

Refactoring: **Extract Method** y **Move Method**. Mecánica:

- Se crea el nuevo método y se copia el código.
- No hay problemas con v.i

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private FormaPago formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, FormaPago formal
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    private double costoProductos() {
        return this.productos.stream
            .mapToDouble(p → p.getPrecio())
            .sum();
    }

    public double getCostoTotal() {
            int añosDesdeFechaAlta = this.getAniosDesdeFechaAlta();
```

```
        if (añosDesdeFechaAlta > 5)
            return (this.costoProductos() + this.extraFormaPago()) * 0.9;
        return this.costoProductos() + this.extraFormaPago();
    }

    private double extraFormaPago() {
            return this.calcularExtra(this.costoProductos());
     }

     private int getAniosDesdeFechaAlta() {
        return Period.between(this.cliente.getFechaAlta(), LocalDate.now()).get
     }
}

public abstract class FormaPago {
    public abstract double calcularExtra(double costoProductos);
}

public class Efectivo extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return 0;
    }
}

public class SeisCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.2;
    }
}

public class DoceCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.5;
    }
}
///
public class Cliente {
    private LocalDate fechaAlta;
```

```
    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}

public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```

Refactoring: Extract Method y Replace Temp With Query.

- En este caso ya se había creado el método.
- Se reemplaza el temp con un llamado directamente.

```
public class Pedido {
    private Cliente cliente;
    private List<Producto> productos;
    private FormaPago formaPago;

    public Pedido(Cliente cliente, List<Producto> productos, FormaPago formal
        this.cliente = cliente;
        this.productos = productos;
        this.formaPago = formaPago;
    }

    private double costoProductos() {
        return this.productos.stream
            .mapToDouble(p -> p.getPrecio())
            .sum();
    }

    public double getCostoTotal() {
        if (this.getAniosDesdeFechaAlta() > 5)
```

```
            return this.costoMasExtra() * 0.9);
        return this.costoMasExtra();
    }

    private double costoMasExtra() {
            return this.costoProductos() + this.calcularExtra(this.costoProductos()
      }

      private int getAniosDesdeFechaAlta() {
          return Period.between(this.cliente.getFechaAlta(), LocalDate.now()).getY
      }
}

public abstract class FormaPago {
    public abstract double calcularExtra(double costoProductos);
}

public class Efectivo extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return 0;
    }
}

public class SeisCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.2;
    }
}

public class DoceCuotas extends FormaPago {
    public double calcularExtra(double costoProductos) {
        return costoProductos * 0.5;
    }
}
///
public class Cliente {
    private LocalDate fechaAlta;
```

```
    public LocalDate getFechaAlta() {
        return this.fechaAlta;
    }
}

public class Producto {
    private double precio;

    public double getPrecio() {
        return this.precio;
    }
}
```