Refactoring y Frameworks Parcial 2da Fecha 2024

```
public class Pago {
    private List<Producto> productos;
    private String tipo;
    private static final double ADICIONAL_TARJETA = 1000.0;
    private static final double DESCUENTO_EFECTIVO = 2000.0;

    public Pago(String tipo, List<Producto> productos) {
        this.productos = productos;
        this.tipo = tipo;
    }

    public double calcularMontoFinal() {
        double total = 0.0;
        if (this.tipo == "EFECTIVO") {
            for (Producto producto : this.productos) {
                total = total + producto.getPrecio() + (producto.getPrecio() * produc
            }
            if (total > 100000) {
                total = total - DESCUENTO_EFECTIVO;
            }
        } else if (this.tipo == "TARJETA") {
            for (Producto producto : this.productos) {
                total = total + producto.getPrecio() + (producto.getPrecio() * produc
            }
            total = total + ADICIONAL_TARJETA;
        }
        return total;
    }
}

public class Producto {
    private double precio;
```

```
    private double IVA;

    public Producto(double precio, double IVA) {
        this.precio = precio;
        this.IVA = IVA;
    }

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

    public double getIVA() {
        return this.IVA;
    }
}
```

Bad smell:

- Feature Envy, el costo del producto tendría que ser calculado dentro de la clase Producto.
    - Refactoring: Se hace un Extract Method de calcularPrecio() y un Move Method hacia Producto, que devuelve el valor del producto.
    - Dicho método es usado ahora en Pago.
- Esto también lleva a los malores olores de Producto como Data Class, y Pago como clase dios. Con el refactoring se acomoda.

```
public class Pago {
    private List<Producto> productos;
    private String tipo;
    private static final double ADICIONAL_TARJETA = 1000.0;
    private static final double DESCUENTO_EFECTIVO = 2000.0;

    public Pago(String tipo, List<Producto> productos) {
        this.productos = productos;
        this.tipo = tipo;
    }

    public double calcularMontoFinal() {
```

```
        double total = 0.0;
        if (this.tipo == "EFECTIVO") {
            for (Producto producto : this.productos) {
                total = total + producto.calcularPrecio();
            }
            if (total > 100000) {
                total = total - DESCUENTO_EFECTIVO;
            }
        } else if (this.tipo == "TARJETA") {
            for (Producto producto : this.productos) {
                total = total + producto.calcularPrecio();
            }
            total = total + ADICIONAL_TARJETA;
        }
        return total;
    }
}

public class Producto {
    private double precio;
    private double IVA;

    public Producto(double precio, double IVA) {
        this.precio = precio;
        this.IVA = IVA;
    }

    public double calcularPrecio() {
        return this.getPrecio() + (this.getPrecio() * this.getIVA()
    }

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

    public double getIVA() {
        return this.IVA;
```

```
    }
}
```

Bad smell: Conditional Statement y Duplicated Code, que viene dado por la variable String tipo de Pago. El refactoring a aplicar en este caso es Replace Conditional With Polymorphism. Mecánica:

- En este caso no es necesario realizar un Extract Method.
- Se crea la jerarquía de clases: las clases PagoEfectivo y PagoTarjeta que pisan el método calcularMontoFinal().
- También como la suma del precio de los productos debe realizarse en ambas, se hace un Extract Method y se crea el método calcularMontoProductos() en Pago.
- Una vez testeado y todo funcionando se elimina el método calcularMontoFinal() de Pago, la variable tipo y la clase Pago se hace abstracta.

De la mano con este refactoring también se hace un Push Down Field, de ADICIONAL_TARJETA y DESCUENTO_EFECTIVO.

Otro bad smell es el de Reinventando la rueda, ya que se estan usando For. Refactoring a aplicar: Replace Loop With Pipeline.

```
public abstract class Pago {
    private List<Producto> productos;

    public Pago(String tipo, List<Producto> productos) {
        this.productos = productos;
    }

     protected double calcularMontoProductos() {
         return this.productos.stream()
             .mapToDouble(p -> p.calcularPrecio())
             .sum();
     }

     protected abstract double calcularMontoFinal();
}
```

```
public class PagoEfectivo extends Pago {
    private static final double DESCUENTO_EFECTIVO = 2000.0;

    public double calcularMontoFinal() {
        if (this.calcularMontoProductos() > 100000)
            return calcularMontoProductos() - DESCUENTO_EFECTIVO;
        return this.calcularMontoProductos();
    }
}

public class PagoTarjeta extends Pago {
    private static final double ADICIONAL_TARJETA = 1000.0;

    public double calcularMontoFinal() {
        return this.calcularMontoProductos() + ADICIONAL_TARJETA;
    }
}

public class Producto {
    private double precio;
    private double IVA;

    public Producto(double precio, double IVA) {
        this.precio = precio;
        this.IVA = IVA;
    }

    public double calcularPrecio() {
        return this.getPrecio() + (this.getPrecio() * this.getIVA()
    }

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

    public double getIVA() {
        return this.IVA;
```

```
    }
}
```

Frameworks

1.Para poder realizar estos dos loggers, deberíamos instanciar dos loggers. Al de consola le haría un addHandler, de la clase “ConsoleHandler” que ya viene con el framework. Y antes de enviar el mensaje de addHandler, le pondría a este Handler el SimpleFormatter.

Por otro lado, haría algo similar haría con el de XML. Al logger le agregaría el Handler de FileHandler, con el XMLFormatter. (Acá no estoy seguro pero creo que por defecto el framework o en la versión que usé, imprime en consola para cualquier log, así que también desactivaría esta opción para que no suceda).

No se debe extender ninguna clase, se usa el framework simplemente.

2.En este caso sí extendería el framework, utilizando la herencia. Crearía un nuevo handler, que extienda la clase Handler, y que tenga en su interior todo el manejo de envío de mensajes por whatsapp. A este handler también le pondría el Level SEVERE (con setLevel(Level)), para que no le preste atención a mensajes de nivel inferior. Y con el SimpleFormatter.

3.Para publicar los logs por email, se debería hacer algo similar, creando un Handler nuevo que extienda al del framework por herencia. Y en este caso sí que hay inversion de control, ya que todo el funcionamiento de la aplicación está dado por el framework. Es decir, en vez de que mi código “llame” al del framework, el del framework llama al mío.

Parcial

Refactoring:

```
public class Cliente {
    private String nombre;
    private String tipo;
    private List<Compra> compras;

    public Cliente(String unNombre) {
        this.nombre = unNombre;
        this.tipo = "basico";
        this.compras = new ArrayList<>();
    }

    public Compra comprar(List<Producto> productos) {
        double temp1 = 0;

        if (this.tipo.equals("basico")) {
            temp1 = 0.1;
        } else if (this.tipo.equals("premium")) {
            temp1 = 0.05;
        } else if (this.tipo.equals("advance")) {
            temp1 = 0;
        }

        double subtotal = productos.stream()
                .mapToDouble(p -> p.getPrecio())
                .sum();

        double costoEnvio = subtotal * temp1;

        Compra n = new Compra(productos, subtotal, costoEnvio);
        this.compras.add(n);

        if (this.montoAcumuladoEnCompras() > 10000) {
            this.tipo = "advance";
        } else if (this.montoAcumuladoEnCompras() > 5000) {
            this.tipo = "premium";
        }

        return n;
```

```
    }

    public double montoAcumuladoEnCompras() {}
}

public class Compra {
    private List<Producto> productos;
    private double subtotal;
    private double envio;
    private String estado;
}

public class Producto {
    private String descripcion;
    private double precio;
}
```

Malos olores (bad smell): Clase dios (Cliente), Data Class (Compra y Producto), Long Method de comprar() en Compra. También Lazy Class de Compra y Producto. Switch Statements para los diferentes tipos. Variable temporal.

Refactorings a aplicar:

- Primero que nada realizar Extract Method de un método calcular calcularPorcentajeEnvio() según el tipo.
- Extract Method de calcularSubtotal(lista de productos).
- Extract Method de calcularTipoCliente()
- Replace Temp With Query, en el cálculo del costo de envío. El método calcularCostoEnvio(productos) retorna el valor del envío.

```
public class Cliente {
    private String nombre;
    private String tipo;
    private List<Compra> compras;

    public Cliente(String unNombre) {
        this.nombre = unNombre;
        this.tipo = "basico";
```

```
    this.compras = new ArrayList<>();
}

public Compra comprar(List<Producto> productos) {
        Compra compra = new Compra(productos, this.calcularSubtotal(produ
                                                    this.calcularCostoEnvio(pr
            // Es un bad smell la temporal, pero es necesaria para agregar a lista
    this.compras.add(compra);
    this.calcularTipoCliente();
    return compra;
}

private double calcularPorcentajeEnvio() {
    if (this.tipo.equals("basico")) {
        return 0.1;
    } else if (this.tipo.equals("premium")) {
        return 0.05;
    } else if (this.tipo.equals("advance")) {
        return 0;
    }
}

private double calcularSubtotal(List<Producto> productos) {
    return productos.stream()
            .mapToDouble(p -> p.getPrecio())
            .sum();
}

private double calcularCostoEnvio(List<Producto> productos) {
    return this.calcularPorcentajeEnvio() * this.calcularSubtotal(productos);
}

private void calcularTipoCliente() {
     if (this.montoAcumuladoEnCompras() > 10000) {
        this.tipo = "advance";
    } else if (this.montoAcumuladoEnCompras() > 5000) {
        this.tipo = "premium";
    }
```

```
    }

    public double montoAcumuladoEnCompras() {...}
}

public class Compra {
    private List<Producto> productos;
    private double subtotal;
    private double envio;
    private String estado;
}

public class Producto {
    private String descripcion;
    private double precio;
}
```

Bad smell: Switch Statement. En la iteración anterior no se sacó este bad smell. Se quita ahora con un Replace Conditional With State. O un Replace Type Code With State.

```
public abstract class Cliente {
    private String nombre;
    private List<Compra> compras;
    private Estrategia estado;

    public Cliente(String unNombre) {
        this.nombre = unNombre;
        this.compras = new ArrayList<>();
        this.estado = new Basico();
    }

    public Compra comprar(List<Producto> productos) {
            Compra compra = new Compra(productos, this.calcularPorcentajeEnvi

        this.compras.add(compra);
        this.calcularTipoCliente();
        return compra;
```

```
    }

    private double calcularPorcentajeEnvio() {
        return this.estrategia.calcularPorcentajeEnvio();
    }

    private void calcularTipoCliente() {
         if (this.montoAcumuladoEnCompras() > 10000) {
            setEstado(new Premium());
        } else if (this.montoAcumuladoEnCompras() > 5000) {
            setEstado(new Advance());
        }
    }

    public void setEstado(Estado estado) {
        this.estado = estado;
    }

    public double montoAcumuladoEnCompras() {...}
}
--
public abstract class Estado {
    public abstract double calcularPorcentajeEnvio();
}

public class Basico extends Cliente {
    public abstract double calcularPorcentajeEnvio() {
        return 0.1;
    }
}

public class Premium extends Cliente {
    public abstract double calcularPorcentajeEnvio() {
        return 0.05;
    }
}

public class Advance extends Cliente {
```

```
    public abstract double calcularPorcentajeEnvio() {
        return 0;
    }
}
--
public class Compra {
    private List<Producto> productos;
    private double costoEnvio;

    public Compra (List<Producto> productos, double porcentaje) {
        this.productos = productos;
        this.costoEnvio = porcentaje * this.calcularSubtotal();
    }

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

public class Producto {
    private String descripcion;
    private double precio;
}
```