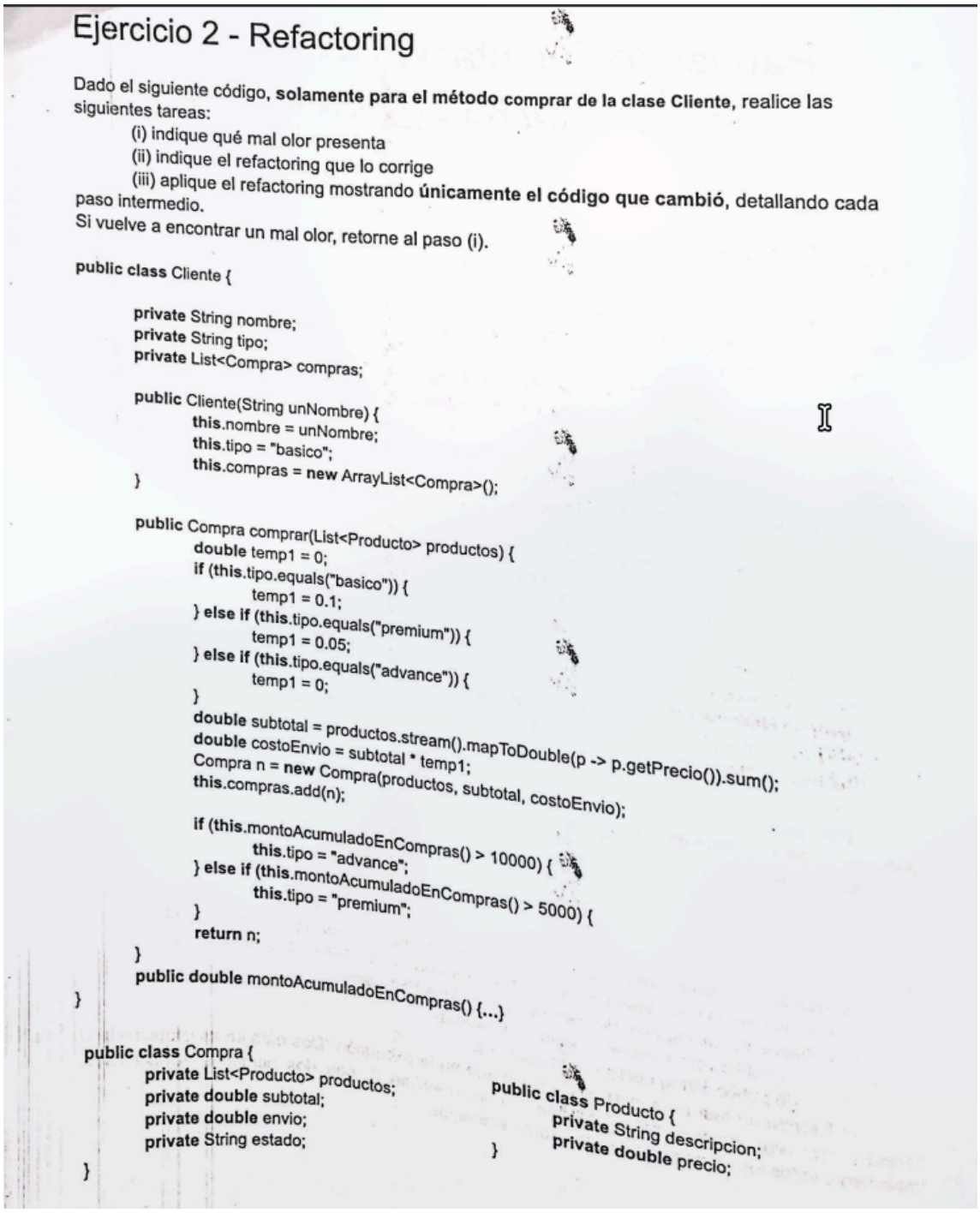
Segundo parcial refactoring



Ejercicio 2 - Refactoring

Dado el siguiente código, **solamente para el método comprar de la clase Cliente**, realice las siguientes tareas:

(i) indique qué mal olor presenta
(ii) indique el refactoring que lo corrige
(iii) aplique el refactoring mostrando **únicamente el código que cambió**, detallando cada paso intermedio.
Si vuelve a encontrar un mal olor, retorne al paso (i).

```
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<Compra>();
        }

        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;
}
```

En el enunciado especifica que solo indique los malos olores presentados en el metodo comprar de la clase Cliente, por lo tanto me voy a abstener a revisar dicho

metodo

Code smell encontrados

- Temporary field (por variable temp1)
- Switch statement
- Otro switch statement
- Long Method

Refactoring a aplicar: Extract Method para long method

Creo un metodo llamado agregarCompra(List<Producto> productos, double subtototal, double costoEnvio)

Luego muevo la implementacion encontrar en el metodo comprar al nuevo metodo, y elimino n y el add de compras, cambiandolo por una llamada al metodo hecho.

Resultado:

```
public void agregarCompra(List<Producto> productos,
                    double subtotal, double costoEnvio) {
   this.compras.add(new Compra(productos, subtotal, costoEnvio))
}
```

Refactoring a aplicar: Replace conditional with state y Extract Method

Pasos: primero lo que hago es crear la jerarquia de clases necesarias:

Interfaz ITipoEnvio el cual tiene 2 metodos: getPorcentajeEnvio() y otro llamado actualizarTipoEnvio(double totalAcumulado)

Creo las subclases necesarias que implementen este metodo las cuales son:

EnvioBasico, EnvioPremium y EnvioAdvance,

Luego de eso aplico extract method a cada uno de los ifs, creando un metodo llamado getPorcentajeEnvio() el cual va a retornar 0.1 y ese metodo lo voy a mover a La subclase EnvioBasico(), y asi sucesivamente con los demas retornos. Luego de terminar de hacer los 3 metodos y moverlos a sus respectivas subclases, elimino el primer statement, el if y los dos else if. Y elimino la variable temp1.

Creo una variable de instancia llamada tipoEnvio, el cual es de tipo ITipoEnvio y se va a crear un metodo llamado setEnvio(ITipoEnvio envio) para cambiar el tipo de envio en tiempo de ejecucion. Cambio la variable de this.tipo a this.tipoEnvio.

Elimino la variable de tipo de la clase, elimino el primer if (con los else if) y el tipo que se utiliza para calcular el costo envio, para que en costoEnvio el calculo se haga, en vez del temp1, se haga con el llamado a getPorcentajeEnvio() del tipoActual.

Tambien elimino la asignacion de tipo a basico en el constructor, ya que no existe mas, y asigno dentro del constructor a tipoEnvio = new EnvioBasico()

Todavia queda mas, el switch statement de abajo utiliza tipo, por lo tanto me puede tirar error ya que no poseo mas la variable tipo dentro de la clase.

Asi que primero hago extract method y creo un metodo llamado actualizarTipoEnvio(double total acumulado).

Dentro de ese primer metodo voy a copiar la condicion del if y, en vez de cambiar el tipo a un “advance” voy a llamar al metodo setEstado(new EnvioAdvance).

Luego de eso muevo dicho metodo a la subclase correspondiente que seria en el basico y en el premium con matices :

- En el metodo de basico tendra los dos ifs, los cuales van a llegar a sus respectivos nuevos estados.
- En el metodo de premium solo tendra uno que va a llegar al tipo advance

Voy a pasar como parametro tambien el Cliente, esto para el metodo actualizarTipoEnvio para todas las clases.

Los metodos de actualizarTipoEnvio en las otras subclases que no se han cambiado los dejo en blanco, ya que no harian nada.

Por ultimo, reemplazo los ifs por llamada a ActualizarTipoEnvio(...); paso por llamada this y el montoAcumuladoEnCompras

Refactoring por code smell para el long method, extract method

En calcular subtotal y costo de envio esas implementaciones se pueden hacer en otro metodo, aplicando extract method.

Creo los metodos calcularSubtotal(productos) y calcularCostoEnvio(subtotal), y muevo la implementacion a esos metodos y cambio las asignaciones de esas dos variables por las llamadas a

esos dos metodos.

Codigo refactorizado

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

    public Cliente(String unNombre) {
        this.nombre = unNombre;
        this.tipoEnvio = new EnvioBasico;
        this.compras = new ArrayList<Compra>();
    }

    public void agregarCompra(List<Producto> productos,
                    double subtotal, double costoEnvio) {
        this.compras.add(new Compra(productos, subtotal, costoEnvio));
    }

    public double getPorcentajeEnvio() {
        return 0;
    }

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

    public double getCostoEnvio(double subtotal) {
        return subtotal * tipoEnvio.getPorcentajeEnvio();
    }

    public Compra comprar(List<Producto> productos) {

        double subtotal = getSubtotal(productos);
        double costoEnvio = getCostoEnvio(subtotal);

        agregarCompra(productos, subtotal, costoEnvio);
```

```
        tipoEnvio.actualizarTipoEnvio(this.montoAcumuladoEnCompras, this);

        return n;
    }

    public void setEnvio(ITipoEnvio tipoEnvio) {
        this.tipoEnvio = tipoEnvio;
    }

    public double montoAcumuladoEnCompras() {...}

}

// INTERFACE Y PATRON STATE

public interface ITipoEnvio {
    public double getPorcentajeEnvio();
    public void actualizarTipoEnvio(double monto, Cliente cliente);
}

public class EnvioBasico implements ITipoEnvio {
    public double getPorcentajeEnvio() {
        return 0.1;
    }

    public void actualizarTipoEnvio(double total, Cliente c) {
        if (total > 10000) {
            c.setEstado(new EnvioAdvance());

        } else if (total > 5000) {
            c.setEstado(new EnvioPremium());
        }
    }
}


public class EnvioPremium implements ITipoEnvio {
    public double getPorcentajeEnvio() {
        return 0.05;
    }

    public void actualizarTipoEnvio(double total, Cliente c) {
        if (total > 10000) {
            c.setEstado(new EnvioAdvance());
        }
```

```
    }
}

public class EnvioAdvance implements ITipoEnvio {
    public double getPorcentajeEnvio() {
        return 0;
    }

    public void actualizarTipoEnvio(double total, Cliente c)
        if (total > 5000) {
            c.setEstado(new EnvioPremium());
        }

}
```