Refactoring

Orientacion a Objetos 2

Integrantes:

- **Francisco Diaz Malbran 22612/7**
- **Julián Quintana 20831/8**

Ejercicio 3 - Facturación de llamadas

Importante: *Aprobando este ejercicio, no será necesario rendir el tema Refactoring en el parcial.*

Fecha límite de entrega: 24/05/2024 23:59 hs (máximo 2 integrantes por grupo).

En el material adicional encontrará una aplicación que registra y factura llamadas telefónicas. Para lograr tal objetivo, la aplicación permite administrar números telefónicos, como así también clientes asociados a un número. Los clientes pueden ser personas físicas o jurídicas. Además, el sistema permite registrar las llamadas realizadas, las cuales pueden ser nacionales o internacionales. Luego, a partir de las llamadas, la aplicación realiza la facturación, es decir, calcula el monto que debe abonar cada cliente.

Importe el material adicional provisto por la cátedra y analícelo para identificar y corregir los malos olores que presenta. En forma iterativa, realice los siguientes pasos:
(i) indique el mal olor,
(ii) indique el refactoring que lo corrige,
(iii) aplique el refactoring (modifique el código).
(iv) asegúrese de que los tests provistos corran exitosamente.
Si vuelve a encontrar un mal olor, retorne al paso (i).

Ud debe entregar:

- Un diagrama de clases UML con el diseño inicial de la solución provista
- La secuencia de refactorings aplicados, documentados cada uno de la siguiente manera:
    - Mal olor detectado en el código
    - Extracto del código que presenta el mal olor
    - Refactoring a aplicar que resuelve el mal olor
    - Código con el refactoring aplicado
- Un diagrama de clases UML con el diseño final
- El código java refactorizado

Importante: asegúrese de que no queden malos olores por identificar y refactorizar.

Mal Olor: Nombres poco descriptivos / No hay declaraciones de visibilidad
Mal olor en clase Empresa

```
    static double descuentoJur = 0.15;
    static double descuentoFis = 0;
```

Refactoring: Rename Field

```
    static double descuentoJuridica = 0.15;
    static double descuentoFisica = 0;
```

Refactoring: Encapsulate field

```
private static double descuentoJuridica = 0.15;
private static double descuentoFisica = 0;
```

Mal olor: Declaracion de atributos publicos / Romper encapsulamiento
Mal olor en clase Empresa

```
public GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
```

```
public GestorNumerosDisponibles getGestorNumeros() {
        return this.guia;
}
```

Refactoring: Encapsule Field / Feature Envy

Refactoring clase Empresa
El getGestorNumerosDisponibles() directamente se borra ya que rompe el encapsulamiento y no cumple ninguna función.
La variable ‘guia’ pasa a ser privada y se crea una nueva función dado que anteriormente se utilizaba la variable pública para lo mismo

```
private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();

public boolean AgregarNumeroGuia(String str) {
    return guia.agregarNumeroTelefono(str);
}
```

Esto conlleva a tener que cambiar también el Test de sistema que hace uso del mismo

```
public void setUp() {
        this.sistema = new Empresa();
        this.sistema.guia.agregarNumeroTelefono(str:"2214444554");
        this.sistema.guia.agregarNumeroTelefono(str:"2214444555");
        this.sistema.guia.agregarNumeroTelefono(str:"2214444556");
        this.sistema.guia.agregarNumeroTelefono(str:"2214444557");
        this.sistema.guia.agregarNumeroTelefono(str:"2214444558");
        this.sistema.guia.agregarNumeroTelefono(str:"2214444559");
}
```

```
this.sistema.guia.agregarNumeroTelefono(str:"2214444558");
```

Aplicando el refactoring

```
public void setUp() {
        this.sistema = new Empresa();
        this.sistema.agregarNumeroGuia(str:"2214444554");
        this.sistema.agregarNumeroGuia(str:"2214444555");
        this.sistema.agregarNumeroGuia(str:"2214444556");
        this.sistema.agregarNumeroGuia(str:"2214444557");
        this.sistema.agregarNumeroGuia(str:"2214444558");
        this.sistema.agregarNumeroGuia(str:"2214444559");
}
```

```
this.sistema.agregarNumeroGuia(str:"2214444558");
```

Mal olor: Feature Envy / declaracion del new() en el body del sistema / falta constructor
El metodo agregarNumeroTelefono de la clase Empresa deberia pertenecer a la clase GestorNumerosDisponibles, ya que se encarga numeros de telefono a la guia y es un metodo correspondiente a la misma clase

Mal olor en clase Empresa

```
private List<Cliente> clientes = new ArrayList<Cliente>();
private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
```

```
public boolean agregarNumeroTelefono(String str) {
    boolean encontre = guia.getLineas().contains(str);
    if (!encontre) {
        guia.getLineas().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Mal olor clase GestorNumerosDisponibles

```
private SortedSet<String> lineas = new TreeSet<String>();
```

Refactoring: Move Method
Se mueve el codigo de la clase Empresa a la clase GestorNumerosDisponibles y se elimina de la anterior

Tanto en clase Empresa como en GestorNumerosDisponibles se crea el constructor donde se instancian sus respectivos atributos

Refactoring en clase Empresa

```
private List<Cliente> clientes;
private GestorNumerosDisponibles guia;

public Empresa() {
    clientes = new ArrayList<Cliente>();
    guia = new GestorNumerosDisponibles();
}
```

```
public boolean agregarNumeroTelefono(String str) {
    boolean encontre = getLineas().contains(str);
    if (!encontre) {
        getLineas().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Refactoring clase GestorNuemerosDisponibles

```
private SortedSet<String> lineas;
public GestorNumerosDisponibles() {
    this.tipoGenerador= new StrategyUltimo();
    this.lineas = new TreeSet<String>();
}
```

Mal olor: Long Method
El metodo calcularMontoTotalLlamadas en la clase Empresa se puede dividir en submetodos

Mal olor en clase Empresa

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        double auxc = 0;
        if (l.getTipoDeLlamada() == "nacional") {
            // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
        } else if (l.getTipoDeLlamada() == "internacional") {
            // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
        }

        if (cliente.getTipo() == "fisica") {
            auxc -= auxc*descuentoFisica;
        } else if(cliente.getTipo() == "juridica") {
            auxc -= auxc*descuentoJuridica;
        }
        c += auxc;
    }
    return c;
}
```

Refactoring: Extract Method

Refactoring clase Empresa

El metodo calcularMontoTotalLlamada es dividido en 2 metodos mas calcularDescuentoCliente y calcularCostoLlamada

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        double auxc = 0;
        c += calcularCostoLlamada(l);

        c -= calcularDescuentoCliente(cliente, c);
    }
    return c;
}

private double calcularDescuentoCliente(Cliente cliente, double c) {
    if (cliente.getTipo() == "fisica") {
        return c*descuentoFisica;
    } else if(cliente.getTipo() == "juridica") {
        return c*descuentoJuridica;
    }
    return 0;
}

private double calcularCostoLlamada(Llamada l) {
    if (l.getTipoDeLlamada() == "nacional") {
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
        return l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
    } else if (l.getTipoDeLlamada() == "internacional") {
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
        return l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}
```

Mal Olor: Data Class / Feature Envy

Tanto la clase Llamada como Persona solo estan compuestas por getters/setters y realizan ningun metodo, tales metodos ademas se encuentran en la clase Empresa la cual metodos que no le corresponden para realizar su trabajo.

En este caso los metodos son: calcularCostoLlamada y calcularDescuentoCliente

Mal olor en clase en Llamada

```
public class Llamada {
    private String tipoDeLlamada;
    private String origen;
    private String destino;
    private int duracion;

    public Llamada(String tipoLlamada, String origen, String destino, int duracion) {
        this.tipoDeLlamada = tipoLlamada;
        this.origen= origen;
        this.destino= destino;
        this.duracion = duracion;
    }

    public String getTipoDeLlamada() {
        return tipoDeLlamada;
    }

    public String getRemitente() {
        return destino;
    }

    public int getDuracion() {
        return this.duracion;
    }

    public String getOrigen() {
        return origen;
    }
}
```

Mal olor clase Empresa

```
private static double descuentoJuridica = 0.15;
private static double descuentoFisica = 0;
```

```
private double calcularDescuentoCliente(Cliente cliente, double c) {
    if (cliente.getTipo() == "fisica") {
        return c*descuentoFisica;
    } else if(cliente.getTipo() == "juridica") {
        return c*descuentoJuridica;
    }
    return 0;
}

private double calcularCostoLlamada(Llamada l) {
    if (l.getTipoDeLlamada() == "nacional") {
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
        return l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
    } else if (l.getTipoDeLlamada() == "internacional") {
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
        return l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}
```

Mal olor en Cliente

```
public class Cliente {
    public List<Llamada> llamadas = new ArrayList<Llamada>();
    private String tipo;
    private String nombre;
    private String numeroTelefono;
    private String cuit;
    private String dni;

    public String getTipo() {
        return tipo;
    }
    public void setTipo(String tipo) {
        this.tipo = tipo;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getNumeroTelefono() {
        return numeroTelefono;
    }
    public void setNumeroTelefono(String numeroTelefono) {
        this.numeroTelefono = numeroTelefono;
    }
    public String getCuit() {
        return cuit;
    }
    public void setCuit(String cuit) {
        this.cuit = cuit;
    }
```

```
    public String getDNI() {
        return dni;
    }
    public void setDNI(String dni) {
        this.dni = dni;
    }
}
```

Refactoring: Move Method / Move field

Refactoring Llamada

Movemos el metodo calcularCostoLlamada a la clase Llamada

```
public double calcularCostoLlamada() {
    if (getTipoDeLlamada() == "nacional") {
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
        return getDuracion() * 3 + (getDuracion() * 3 * 0.21);
    } else if (getTipoDeLlamada() == "internacional") {
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
        return getDuracion() * 150 + (getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}
```

Refactoring class Cliente

Movemos los campos de descuento y el metodo calcularDescuentoCliente

```
private static double descuentoJuridica = 0.15;
private static double descuentoFisica = 0;
```

```
public double calcularDescuentoCliente(double c) {
    if (getTipo() == "fisica") {
        return c*descuentoFisica;
    } else if(getTipo() == "juridica") {
        return c*descuentoJuridica;
    }
    return 0;
}
```

Mal Olor: Feacture Envy / dead code

El metodo registrarLlamada llama a un cliente y le agrega la llamada a su lista de llamadas. Las llamadas deberian agregarse en el propio cliente ya que posee una lista de llamadas propias y se pueden conocer las mismas desde la Empresa a partir del cliente

Mal olor en clase Empresa

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
    Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

Refactoring: move method / remove dead code

refactoring en clase Empresa

La relacion entre multiples llamadas y empresa no es necesaria, ya que se conoce a través del cliente

```
private List<Llamada> llamadas = new ArrayList<Llamada>();
```

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
    Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.agregarLlamada(llamada);
    return llamada;
}
```

Refactoring en clase Cliente

```
public void agregarLlamada(Llamada llamada) {
    llamadas.add(llamada);
```

Mal olor: Switch statment / constructor telescopico

Las clases Llamada y Cliente contienen logica para diferentes tipos de la misma clase, por lo que hay que crear subclases para las mismas

El mal olor ‘constructor telescopico’ hace referencia al hecho que se esta inicializando un objeto a través de sus getters, ya que estos no garantizan que el objeto este completamente inicializado.
Para esto se crea un constructor en la clase Cliente si se utiliza para inicializar el objeto.

Mal olor clase Cliente

```
private static double descuentoJuridica = 0.15;
private static double descuentoFisica = 0;
```

```
public double calcularDescuentoCliente(double c) {
    if (getTipo() == "fisica") {
        return c*descuentoFisica;
    } else if(getTipo() == "juridica") {
        return c*descuentoJuridica;
    }
    return 0;
}
```

Mal olor clase Llamada

```
public double calcularCostoLlamada() {
    if (getTipoDeLlamada() == "nacional") {
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
        return getDuracion() * 3 + (getDuracion() * 3 * 0.21);
    } else if (getTipoDeLlamada() == "internacional") {
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
        return getDuracion() * 150 + (getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}
```

Mal olor clase Empresa

```
public Cliente registrarUsuario(String data, String nombre, String tipo) {
    Cliente var = new Cliente();
    if (tipo.equals("fisica")) {
        var.setNombre(nombre);
        String tel = this.obtenerNumeroLibre();
        var.setTipo(tipo);
        var.setNumeroTelefono(tel);
        var.setDNI(data);
    }
    else if (tipo.equals("juridica")) {
        String tel = this.obtenerNumeroLibre();
        var.setNombre(nombre);
        var.setTipo(tipo);
        var.setNumeroTelefono(tel);
        var.setCuit(data);
    }
    clientes.add(var);
    return var;
}
```

```
public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
    Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

Refactoring: Replace conditional with polymorphism / Replace type code with sublcasses

Creamos subclases para cada tipo especifico, heredando de la clase base de la estructura en comun y agregando comportamientos y atributos especificos. Para esto es necesario un push down field de los descuentos a sus subclases.

Refactoring en class Cliente

Cliente pasa a ser una clase abstracta con su respectivo constructor y la funcion calcularDescuentoCliente que podran utilizar las sublcases

Ademas se crea el constructor.

```
public List<Llamada> llamadas = new ArrayList<Llamada>();
private String nombre;
private String numeroTelefono;


public Cliente(String nombre, String numeroTelefono) {
    this.nombre = nombre;
    this.numeroTelefono = numeroTelefono;
}
```

```
public abstract double getDescuento();
public abstract void setDescuento(double nuevoValor);
```

Refactoring nueva clase clienteFisico

```
public class clienteFisico extends Cliente{

        private String dni;
        private static double descuento = 0;

        public clienteFisico(String nombre, String numeroTelefono, String dni) {
                super(nombre,numeroTelefono);
                this.dni = dni;
        }

        public String getDni() {
                return this.dni;
        }

        public void setDni(String dni) {
                this.dni = dni;
        }

        public  double getDescuento(){
            return this.descuento;
        };

        public void setDescuento(double nuevoValor){
            this.descuento = nuevoValor;
        }

}
```

Refactoring nueva clase clienteJuridico

```
public class clienteJuridico extends Cliente{

        private String cuit;
        private static double descuento = 0.15;

        public clienteJuridico(String nombre, String numeroTelefono, String cuit) {
                super(nombre,numeroTelefono);
                this.cuit = cuit;
        }

        public String getCuit() {
                return this.cuit;
        }

        public void setCuit(String cuit) {
                this.cuit = cuit;
        }



        public  double getDescuento(){
            return this.descuento;
        };

        public void setDescuento(double nuevoValor){
            this.descuento = nuevoValor;
        }
}
```

Refactoring clase Llamada

Al igual que en la clase Cliente, la clase Llamada pasa a ser abstracta heredado sus respectivos comportamientos y atributos

```
private String origen;
private String destino;
private int duracion;

public Llamada(String origen, String destino, int duracion) {
    this.origen= origen;
    this.destino= destino;
    this.duracion = duracion;
}
```

```
public abstract double calcularCostoLlamada();
```

Refactoring nueva clase Llamada nacional

```
public class llamadaNacional extends Llamada{

    public llamadaNacional(String origen, String destino, int duracion) {
        super(origen,destino,duracion);
    }

    public double calcularCostoLlamada() {
        return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
    }

}
```

Refactoring nueva clase Llamada internacional

```java
public class llamadaInternacional extends Llamada{

    public llamadaInternacional(String origen, String destino, int duracion) {
        super(origen,destino,duracion);
    }

    public double calcularCostoLlamada() {
        return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
    }
}
```

Refactoring en clase Empresa

Al crear 2 nuevas subclases clienteFisico y clienteJuridico, el metodo crearUsuario está dividido en 2 metodos distintos depende el tipo de cliente

```java
public Cliente registrarUsuarioFisico(String dni, String nombre) {
        String tel = this.obtenerNumeroLibre();
        Cliente cliente = new clienteFisico(nombre,numeroTelefono: tel,dni);
        clientes.add(e: cliente);
        return cliente;
}

public Cliente registrarUsuarioJuridico(String cuit, String nombre ) {
        String tel = this.obtenerNumeroLibre();
        Cliente cliente = new clienteJuridico(nombre,numeroTelefono: tel,cuit);
        clientes.add(e: cliente);
        return cliente;
}
```

Ocurre lo mismo al registrarLlamadas y crear las nuevas clases llamada

```java
public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
    Llamada llamada = new llamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.agregarLlamada(llamada);
    return llamada;
}

public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
    Llamada llamada = new llamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    origen.agregarLlamada(llamada);
    return llamada;
}
```

Ademas al crear el constructor en el cliente y utilizando el mismo en lugar de los getters y setters.

```java
public Cliente registrarUsuario(String data, String nombre, String tipo) {

    String tel = this.obtenerNumeroLibre();
    if(tipo.equals("fisica")) {
        Cliente var = new Cliente(nombre,data,tel,tipo);
    }
    else if(tipo.equals("juridica")) {
        Cliente var = new Cliente(nombre,data,tel,tipo);
    }
    clientes.add(var);
    return var
}
```

Al crear estas herencias se cambia la forma en la que se crean las instancias de los usuarios y llamadas. Esto conlleva a también tener que cambiar la clase test, para actualizar la firma para registrar un Usuario.

```
void testcalcularMontoTotalLlamadas() {
        Cliente emisorPersonaFisca = sistema.registrarUsuario("11555666", "Brendan Eich" , "fisica");
        Cliente remitentePersonaFisica = sistema.registrarUsuario("00000001", "Doug Lea" , "fisica");
        Cliente emisorPersonaJuridica = sistema.registrarUsuario("17555222", "Nvidia Corp" , "juridica");
        Cliente remitentePersonaJuridica = sistema.registrarUsuario("25765432", "Sun Microsystems" , "juridica");
```

```
Cliente nuevaPersona = this.sistema.registrarUsuario("2444555","Alan Turing", "fisica");
```

Aplicando los nuevos cambios:

```
void testcalcularMontoTotalLlamadas() {
        Cliente emisorPersonaFisca = sistema.registrarUsuarioFisico(dni:"11555666", nombre: "Brendan Eich" );
        Cliente remitentePersonaFisica = sistema.registrarUsuarioFisico(dni:"00000001", nombre: "Doug Lea" );
        Cliente emisorPersonaJuridica = sistema.registrarUsuarioJuridico(cuit:"17555222", nombre: "Nvidia Corp" );
        Cliente remitentePersonaJuridica = sistema.registrarUsuarioJuridico(cuit:"25765432", nombre: "Sun Microsystems" );
```

```
Cliente nuevaPersona = this.sistema.registrarUsuarioFisico(dni:"Alan Turing", nombre: "2444555");
```

Y tambien por el lado de registrar llamadas

```
this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaFisica, "nacional", 10);
this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaFisica, "internacional", 8);
this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaJuridica, "nacional", 5);
this.sistema.registrarLlamada(emisorPersonaJuridica, remitentePersonaJuridica, "internacional", 7);
this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaFisica, "nacional", 15);
this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaFisica, "internacional", 45);
this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaJuridica, "nacional", 13);
this.sistema.registrarLlamada(emisorPersonaFisca, remitentePersonaJuridica, "internacional", 17);
```

Aplicando los cambios

```
this.sistema.registrarLlamadaNacional(origen: emisorPersonaJuridica, destino:remitentePersonaFisica, duracion:10);
this.sistema.registrarLlamadaInternacional(origen: emisorPersonaJuridica, destino:remitentePersonaFisica, duracion:8);
this.sistema.registrarLlamadaNacional(origen: emisorPersonaJuridica, destino:remitentePersonaJuridica, duracion:5);
this.sistema.registrarLlamadaInternacional(origen: emisorPersonaJuridica, destino:remitentePersonaJuridica, duracion:7);
this.sistema.registrarLlamadaNacional(origen: emisorPersonaFisca, destino:remitentePersonaFisica, duracion:15);
this.sistema.registrarLlamadaInternacional(origen: emisorPersonaFisca, destino:remitentePersonaFisica, duracion:45);
this.sistema.registrarLlamadaNacional(origen: emisorPersonaFisca, destino:remitentePersonaJuridica, duracion:13);
this.sistema.registrarLlamadaInternacional(origen: emisorPersonaFisca, destino:remitentePersonaJuridica, duracion:17);
```

Mal Olor: Declaración de atributos públicos

Mal olor en clase Cliente
Además el atributo llamadas está instanciando en la declaración y no en un Constructor por lo que se crea el mismo,

```
public List<Llamada> llamadas = new ArrayList<Llamada>();
```

Refactoring: Encapsule Field

Refactoring en clase Cliente

```
private List<Llamada> llamadas;

public Cliente(String nombre, String numeroTelefono) {
        this.nombre = nombre;
        this.numeroTelefono = numeroTelefono;
        llamadas = new ArrayList<Llamada>();
}
```

Mal olor: Switch Statment

Mal olor en clase GestorNumerosDisponibles

```
public String obtenerNumeroLibre() {
    String linea;
    switch (tipoGenerador) {
        case "ultimo":
            linea = lineas.last();
            lineas.remove(linea);
            return linea;
        case "primero":
            linea = lineas.first();
            lineas.remove(linea);
            return linea;
        case "random":
            linea = new ArrayList<String>(lineas)
                    .get(new Random().nextInt(lineas.size()));
            lineas.remove(linea);
            return linea;
    }
    return null;
}
```

Refactoring: Replace conditional logic with strategy

Se crea una nueva clase NumeroLibreStrategy

Refactoring: Move Method / Replace Temp with Query / Extract Parameter

El método obtenerNumeroLibre corresponde a la nueva clase NumeroLibreStrategy . Además el tipoGenerador tiene definido un valor local, el cual debería estar definido por el tipo de estrategia

Refactoring en NumeroLibreStrategy

```
    public String obtenerNumeroLibre() {
        String linea;
        switch (tipoGenerador) {
            case "ultimo":
                linea = lineas.last();
                lineas.remove(linea);
                return linea;
            case "primero":
                linea = lineas.first();
                lineas.remove(linea);
                return linea;
            case "random":
                linea = new ArrayList<String>(lineas)
                        .get(new Random().nextInt(lineas.size()));
                lineas.remove(linea);
                return linea;
        }
        return null;
    }

}
```

Refactoring GestorNumerosDisponibles

TipoGenerador dejó de ser un valor definido como variable local, además que el metodo obtenerNumeroLibre es un llamado para ser utilizado en la clase Strategy.

```
public class GestorNumerosDisponibles {
        private NumeroLibreStrategy tipoGenerador;
        private SortedSet<String> lineas;

        public String obtenerNumeroLibre() {
                return tipoGenerador.obtenerNumeroLibre(gestor: this);
        }
```

Mal olor: Switch Statment

Mal olor en NumeroLibreStrategy

```
    public String obtenerNumeroLibre() {
        String linea;
        switch (tipoGenerador) {
            case "ultimo":
                linea = lineas.last();
                lineas.remove(linea);
                return linea;
            case "primero":
                linea = lineas.first();
                lineas.remove(linea);
                return linea;
            case "random":
                linea = new ArrayList<String>(lineas)
                        .get(new Random().nextInt(lineas.size()));
                lineas.remove(linea);
                return linea;
        }
        return null;
    }
}
```

Refactoring: Replace conditional with polymorphism / Replace type code with sublcasses

NumeroLibreStrategy corresponde a una interfaz la cual sera implementada por los tipos de generadores StrategyUltimo, StrategyPrimero y StrategyRandom. Estos implementaran el metodo obtenerNumeroLibre el cual se le enviara el parametro gestor de tipo GestorNumeroDisponible que corresponde al tipo de estrategia que tomara el usuario

Refactoring en NumeroLibreStrategy

```
public interface NumeroLibreStrategy {

    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor);

}
```

Refactoring en nueva clase StrategyUltimo

```
public class StrategyUltimo implements NumeroLibreStrategy{


    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
        String linea = gestor.getLineas().last();
        gestor.getLineas().remove(linea);
        return linea;
    }

}
```

Refactoring en nueva clase StrategyPrimero

```
public class StrategyPrimero implements NumeroLibreStrategy{


    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
        String linea = gestor.getLineas().first();
        gestor.getLineas().remove(linea);
        return linea;
    }

}
```

Refactoring en nueva clase StrategyRandom

```
public class StrategyRandom implements NumeroLibreStrategy{

    public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
        String linea = new ArrayList<String>(gestor.getLineas())
                .get(new Random().nextInt(gestor.getLineas().size()));
        gestor.getLineas().remove(linea);
        return linea;
    }

}
```

Esto conlleva a cambiar la clase test

```
void obtenerNumeroLibre() {
        // por defecto es el ultimo
        assertEquals(expected: "2214444559", actual: this.sistema.obtenerNumeroLibre());

        this.sistema.getGestorNumeros().cambiarTipoGenerador("primero");
        assertEquals(expected: "2214444554", actual: this.sistema.obtenerNumeroLibre());

        this.sistema.getGestorNumeros().cambiarTipoGenerador("random");
        assertNotNull(actual: this.sistema.obtenerNumeroLibre());
}
```

Aplicando las nuevas formas de instancia

```
void obtenerNumeroLibre() {
        // por defecto es el ultimo
        assertEquals(expected: "2214444559", actual: this.sistema.obtenerNumeroLibre());

        this.sistema.getGestorNumeros().setGenerador(new StrategyPrimero());
        assertEquals(expected: "2214444554", actual: this.sistema.obtenerNumeroLibre());

        this.sistema.getGestorNumeros().setGenerador(new StrategyRandom());
        assertNotNull(actual: this.sistema.obtenerNumeroLibre());
}
```

Mal Olor: Metodo extenso y complejo

Mal olor en clase GestorNumerosDisponibles

```
public boolean agregarNumeroTelefono(String str) {
    boolean encontre = getLineas().contains(str);
    if (!encontre) {
        getLineas().add(str);
        encontre= true;
        return encontre;
    }
    else {
        encontre= false;
        return encontre;
    }
}
```

Refactoring: Substitute Algorithm

Refactoring clase GestorNumerosDisponibles

```
public boolean agregarNumeroTelefono(String str) {
    if (!getLineas().contains(str)) {
        getLineas().add(str);
        return true;
    }
    return false;
}
```

Mal Olor: Reinventa la rueda / Long Method

Se recorren las llamada de los clientes para calcular su costo con un for, cuando se podria utilizar de forma mas practica los piplines que ofrece java.

Para ello el metodo calcularCostoLlamada es demasiado largo y calcula el costo de todas las llamadas para un cliente enviado por parametro. La tarea de calcular el costo total de las llamadas en un determinado cliente deberia estar en la clase cliente.

Mal olor en clase Empresa

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double c = 0;
    for (Llamada l : cliente.llamadas) {
        c += l.calcularCostoLlamada();
        c -= cliente.calcularDescuentoCliente(c);
    }
    return c;
}
```

Refactoring: Extract Method / Replace Loop with Pipline
De esta forma el cliente calcula el precio de todas sus llamadas y la Empresa solita el cliente al cual quiere calcular dicho costo.

Esto podría variar si la empresa requiere que se solicite el precio de todas las llamadas de todos los clientes y a cada clase cliente se le solicitará de forma individual el precio de sus llamadas.

Refactoring Empresa

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
        return cliente.calcularCostoLlamadasCliente();
}
```

Refactoring Cliente

```
public double calcularCostoLlamadasCliente() {
        Double precio = this.llamadas.stream()
                        .mapToDouble(llamada -> llamada.calcularCostoLlamada())
                        .sum();
        return this.calcularDescuentoCliente(precio);
}
```