Milagros Palacios

Orientación a Objetos 2
TP Refactoring - Ejercicio 3

Mal olor en clase Cliente: Declaración de atributo público.

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

Refactoring en clase Cliente: **Encapsulate Field**

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

Mal olor en clase empresa : nombres pocos descriptivos en el método *registrarUsusario.*

```
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;
}
```

Refactoring en clase Empresa: Rename parámetros

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

Mal olor en clase Cliente: No existe constructor para inicializar los objetos de la clase con valores.

La clase Cliente no tiene constructor que establezca valores para sus variables internas durante la instanciación. En su lugar, la clase Empresa se encarga de asignar estos valores mediante el uso de setters, lo que añade complejidad al método.

Mal olor en 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;
}
```

Refactoring en clase cliente: Constructor Initialization

No hay un refactoring específico para esto, pero tener un constructor permite inicializar objetos y establecer valores iniciales al momento de su creación, lo que asegura que el objeto esté en un estado inicial adecuado.También ayuda a mantener el encapsulamiento de los datos al garantizar que solo se puedan establecer valores válidos al momento de la inicialización del objeto.

Refactoring en clase Cliente :

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

Refactoring Remove Dead Code

Clase Empresa : Al agregar el constructor tengo que cambiar el método RegistroUsuario.Eliminar los setters.

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

Mal olor en la clase Empresa : Nombres de variables poco descriptivos

```java
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*descuentoFis;
        } else if(cliente.getTipo() == "juridica") {
            auxc -= auxc*descuentoJur;
        }
        c += auxc;
    }
    return c;
}
```

Refactoring Rename Variable

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

        if (cliente.getTipo() == "fisica") {
            costo -= costo*descuentoFis;
        } else if(cliente.getTipo() == "juridica") {
            costo -= costo*descuentoJur;
        }
        costo += costo;
    }
    return costo;
}
```

Mal olor en la clase Empresa : Long Method

El método *calcularMontoTotalLlamadas* está haciendo demasiadas cosas : calculando el costo de cada una de las llamadas de un cliente dependiendo de su tipo (nacional o internacional) , aplicando los descuentos basados en el tipo de cliente (física o jurídica).Y luego suma esos costos para obtener el total.

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

        if (cliente.getTipo() == "fisica") {
            costo -= costo*descuentoFis;
        } else if(cliente.getTipo() == "juridica") {
            costo -= costo*descuentoJur;
        }
        costo += costo;
    }
    return costo;
}
```

Refactoring: Extract Method

Este procedimiento extenso puede ser descompuesto en tres métodos más pequeños, cada uno con funcionalidades individuales y específicas.

Refactoring en Clase Empresa:

```
private double calcularCostoLlamada(Llamada llamada) {
    if (llamada.getTipoDeLlamada().equals("nacional")) {
        return llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);
    } else if (llamada.getTipoDeLlamada().equals("internacional")) {
        return llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}

private double aplicarDescuento(double costo, String tipoCliente) {
    if (tipoCliente.equals("fisica")) {
        return costo - (costo * descuentoFis);
    } else if (tipoCliente.equals("juridica")) {
        return costo - (costo * descuentoJur);
    }
    return costo; //
}

public double calcularMontoTotalLlamadas(Cliente cliente) {
    double total = 0;
    for (Llamada llamada : cliente.llamadas) {
        double costoLlamada = calcularCostoLlamada(llamada);
        costoLlamada = aplicarDescuento(costoLlamada, cliente.getTipo());
        total += costoLlamada;
    }
    return total;
}
```

Mal olor: Feature Envy y Data Class

La clase Empresa está realizando operaciones y cálculos utilizando datos de otros objetos, Llamada y Cliente, lo que convierte a estos objetos en clases de datos al no tener otra funcionalidad.

Mal olor en clases Cliente y Llamada : Clase de datos
Solo tienen variables y getters/setters para esas variables.Estas clases actúan únicamente como contenedor de datos.Tienen atributos , getters y setters para los atributos y nada más.

```
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 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 en clase empresa : Feature Envy
Estos métodos muestran más interés en la clase Llamada y Cliente que en la suya propia,ya que, estos están accediendo repetidamente a los datos de estas clases.

```
private double calcularCostoLlamada(Llamada llamada) {
    if (llamada.getTipoDeLlamada().equals("nacional")) {
        return llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);
    } else if (llamada.getTipoDeLlamada().equals("internacional")) {
        return llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}

private double aplicarDescuento(double costo, String tipoCliente) {
    if (tipoCliente.equals("fisica")) {
        return costo - (costo * descuentoFis);
    } else if (tipoCliente.equals("juridica")) {
        return costo - (costo * descuentoJur);
    }
    return costo; //
}
```

Refactoring Move Field

Muevo las variables estáticas de descuentos a la clase 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;

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

Refactoring: Move Method

Muevo el método *aplicarDescuento* y lo cambio a público en la clase Cliente.

```
    public double aplicarDescuento(double costo) {
        if (this.getTipo().equals("fisica")) {
            return costo - (costo * descuentoFis);
        } else if (this.getTipo().equals("juridica")) {
            return costo - (costo * descuentoJur);
        }
        return costo;
    }
```

Refactoring Move Method

Muevo el método *calcularCostoLlamada()* a la clase Llamada y lo cambio a público.

```
public class Llamada {
```

```
    public double calcularCostoLlamada() {
        if (this.getTipoDeLlamada().equals("nacional")) {
            return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
        } else if (this.getTipoDeLlamada().equals("internacional")) {
            return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
        }
        return 0;
    }
```

Mal olor : Switch Statements
Usar una única clase para representar dos tipos distintos de Cliente,genera redundancia en las variables, repeticiones de código y la necesidad de emplear condicionales, aspectos que podrían abordarse de manera más eficiente mediante el uso de herencia.

Mal olor en Clase Cliente:

```
public double aplicarDescuento(double costo) {
    if (this.getTipo().equals("fisica")) {
        return costo - (costo * descuentoFis);
    } else if (this.getTipo().equals("juridica")) {
        return costo - (costo * descuentoJur);
    }
    return costo;
}
```

Mal Olor en clase Empresa :

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

Refactoring: Replace Conditional with Polymorphism
Refactoring para remover los condicionales que preguntan por el tipo de Cliente.

Divido en subclases cada tipo específico, heredando de la clase base la estructura común y añadiendo los atributos y comportamientos específicos de cada tipo. Para ello, hago "**push down field**" para los descuentos hacia sus respectivas subclases,y "**push down method**" para el cálculo del descuento.También remuevo el atributo tipo (Remove Field).

Refactoring en clase Cliente:

```java
public abstract class Cliente {
    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 aplicarDescuento (double costo);
```

Refactoring en nueva Clase ClienteFisico:

```java
public class ClienteFisico extends Cliente {

    private String dni;
    static double descuentoFisico = 0;

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

    public String getDni() {
        return dni;
    }

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

    public static double getDescuentoFisico() {
        return descuentoFisico;
    }

    public static void setDescuentoFisico(double descuentoFisico) {
        ClienteFisico.descuentoFisico = descuentoFisico;
    }

    public double aplicarDescuento(double costo) {
       return costo - (costo * descuentoFisico);
    }
```

Refactoring en nueva Clase ClienteJuridico:

```java
public class ClienteJuridico extends Cliente {

    private String cuit;
    static double descuentoJuridico = 0.15;


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

    public double aplicarDescuento(double costo) {
         return costo - (costo * descuentoJuridico);
    }

    public String getCuit() {
        return cuit;
    }

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

    public static double getDescuentoJur() {
        return descuentoJuridico;
    }

    public static void setDescuentoJur(double descuentoJur) {
        ClienteJuridico.descuentoJuridico = descuentoJur;
    }
```

Refactoring en Clase Empresa:
Debido a las modificaciones en la clase Cliente, necesito hacer ajustes en la clase Empresa sobre cómo se registran los usuarios.

```java
public class Empresa {
```

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

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

Al aplicar este refactoring tengo que cambiar el test.

Antes EmpresaTest:

```
@Test
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");
```

Despues EmpresaTest :

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

Mal olor : Switch Statements

Usar una única clase para representar dos tipos distintos de Llamada,genera redundancia en las variables, repeticiones de código y la necesidad de emplear condicionales, aspectos que podrían abordarse de manera más eficiente mediante el uso de herencia.

Mal olor en Clase Llamada:

```
public double calcularCostoLlamada() {
    if (this.getTipoDeLlamada().equals("nacional")) {
        return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
    } else if (this.getTipoDeLlamada().equals("internacional")) {
        return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
    }
    return 0;
}
```

Refactoring: Replace Conditional with Polymorphism
Creó subclases para cada tipo específico y agregó el comportamiento específico de cada tipo de llamada. Para esto es necesario hacer un **push down method** del cálculo de costo de llamada y remover el atributo tipoDeLlamada (Remove Field).

Refactoring en Clase Llamada:

```
public abstract class Llamada {
    private String numeroEmisor;
    private String numeroReceptor;
    private int duracion;


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

    public abstract double calcularCostoLlamada ();


    public String getNumeroEmisor() {
        return numeroEmisor;
    }

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

    public String getNumeroRemitente() {
        return numeroReceptor;
    }
}
```

Refactoring en nueva Clase LlamadaNacional:

```
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 en nueva Clase LlamadaInternacional:

```
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 hacer el refactoring en la clase llamada tengo que cambiar cómo se registran las llamadas en la clase empresa.

```
public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino,int duracion) {
    LlamadaNacional llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}

public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino,int duracion) {
    LlamadaInternacional llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

Al aplicar este refactoring tengo que cambiar el test.

Antes EmpresaTest:

```
    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);
```

Despues EmpresaTest:

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

Mal olor en clase Empresa : método con nombre de parámetro poco descriptivo.

```
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;
    }
}
```

Refactoring: Rename Parámetros

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

Mal olor en clase Empresa : Long Method
El método puede ser más corto cambiando la lógica.

Refactoring: Substitute Algorithm

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

Mal olor Duplicated Code
Tanto la clase Empresa como la clase Cliente tienen la misma colección de llamadas.No es necesario que la Empresa tenga el atributo llamadas, ya que estas se conocen a través del Cliente.

Mal olor en Clase Empresa:

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

Mal olor en Clase Clietne:

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

Refactoring: Remove Dead Code

Refactoring en Clase Empresa:
Se eliminó el campo y sus getters y setters.

```
public class Empresa {
    private List<Cliente> clientes = new ArrayList<Cliente>();
    private GestorNumerosDisponibles guiaTelefonica = new GestorNumerosDisponibles();
```

Mal olor: Feature Envy

La clase Empresa al agregar una llamada está manipulando la colección de llamadas del cliente, la cual no le pertenece. Una llamada debe ser creada y agregada por la clase Cliente.

Mal olor en Clase Empresa

```
public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino,int duracion) {
    LlamadaNacional llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}

public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino,int duracion) {
    LlamadaInternacional llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    llamadas.add(llamada);
    origen.llamadas.add(llamada);
    return llamada;
}
```

Refactoring: Move Method

Refactoring Clase Cliente:

```
public abstract class Cliente {
```

```
public Llamada registrarLlamadaNacional(Cliente destino,int duracion) {
    LlamadaNacional llamada = new LlamadaNacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    this.llamadas.add(llamada);
    return llamada;
}

public Llamada registrarLlamadaInternacional(Cliente destino,int duracion) {
    LlamadaInternacional llamada = new LlamadaInternacional(this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
    this.llamadas.add(llamada);
    return llamada;
}
```

Refactoring en Clase Empresa:

```
public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino,int duracion) {
    return origen.registrarLlamadaNacional(destino, duracion);
}

public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino,int duracion) {
    return origen.registrarLlamadaInternacional(destino, duracion);
}
```

Mal olor en clase Empresa: Feature Envy

Para calcular el costo total de las llamadas de un cliente, la Clase Empresa utiliza las llamadas de cada Cliente.Es funcionalidad debería ser de cada Cliente.Cada cliente debería tener su costo total de llamadas.

```
public class Empresa {
```

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double total = 0;
    for (Llamada llamada : cliente.llamadas) {
        double costoLlamada = llamada.calcularCostoLlamada();
        costoLlamada = aplicarDescuento(costoLlamada);
        total += costoLlamada;
    }
    return total;
}
```

Refactoring: Move method

Clase Cliente:

```
public abstract class Cliente {
```

```
public double calcularMontoTotalLlamadas() {
    double total = 0;
    for (Llamada llamada : this.llamadas) {
        double costoLlamada = llamada.calcularCostoLlamada();
        costoLlamada = aplicarDescuento(costoLlamada);
        total += costoLlamada;
    }
    return total;
}
```

Con este refactoring la responsabilidad del costo de llamadas queda en el cliente.Con este cambio así queda el método en la clase Empresa:

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

Mal olor: Reinventar la rueda

Se utilizan bucles for para iterar sobre listas cuando es mucho más eficiente aprovechar las pipelines proporcionadas por Java.

Mal Olor en clase Cliente

```
public double calcularMontoTotalLlamadas() {
    double total = 0;
    for (Llamada llamada : this.llamadas) {
        double costoLlamada = llamada.calcularCostoLlamada();
        costoLlamada = aplicarDescuento(costoLlamada);
        total += costoLlamada;
    }
    return total;
}
```

Refactoring: Replace Loop with Pipeline

Refactoring en Clase Cliente:

```
public double calcularMontoTotalLlamada (double costo) {
     return this.llamadas.stream()
             .mapToDouble(llamada -> aplicarDescuento(llamada.calcularCostoLlamada()))
             .sum();
}
```

Mal olor : Primitive Obsession
La representación de los tipos de generadores como strings ("último", "primero", "random").

Mal olor en la clase GestorNumerosDisponibles:

```
public class GestorNumerosDisponibles {
    private SortedSet<String> lineas = new TreeSet<String>();
    private String tipoGenerador = "ultimo";

    public SortedSet<String> getLineas() {
        return lineas;
    }

    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 puede crear un Strategy para cada variante y hacer que el método original delegue el cálculo a la instancia de Strategy correspondiente.

Refactoring clase GeneradorNumeroDisponibles:
Pongo por defecto el último como indica el test

```
public class GestorNumerosDisponibles {
    private SortedSet<String> lineas = new TreeSet<>();
    private GeneradorNumeroLibre generadorNumeroLibre = new GeneradorUltimo();;

    public GestorNumerosDisponibles() {

    }

    public GestorNumerosDisponibles(GeneradorNumeroLibre generador) {
        this.generadorNumeroLibre = generador;
    }

    public SortedSet<String> getLineas() {
        return lineas;
    }

    public void setGeneradorNumeroLibre(GeneradorNumeroLibre generadorNumeroLibre) {
        this.generadorNumeroLibre = generadorNumeroLibre;
    }

    public String obtenerNumeroLibre() {
        return generadorNumeroLibre.obtenerNumeroLibre(lineas);
    }
```

Refactoring nueva Interfaz GeneradorNumeroLibre:

```
public interface GeneradorNumeroLibre {

    String obtenerNumeroLibre(SortedSet<String> lineas);


}
```

Refactoring nueva Clase GeneradorUltimo:

```
public class GeneradorUltimo implements GeneradorNumeroLibre {

    @Override
       public String obtenerNumeroLibre(SortedSet<String> lineas) {
           String linea = lineas.last();
           lineas.remove(linea);
           return linea;
       }

}
```

Refactoring nueva Clase GeneradorPrimero:

```java
public class GeneradorPrimero implements GeneradorNumeroLibre {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.first();
        lineas.remove(linea);
        return linea;
    }

}
```

Refactoring nueva Clase GeneradorRandom:

```java
public class GeneradorRandom implements GeneradorNumeroLibre {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = new ArrayList<>(lineas).get(new Random().nextInt(lineas.size()));
        lineas.remove(linea);
        return linea;
    }

}
```

Al aplicar el refactoring tengo que cambiar el test:

Test antes

```java
@Test
void obtenerNumeroLibre() {
    // por defecto es el ultimo
    assertEquals("2214444559", this.sistema.obtenerNumeroLibre());

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

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

Test despues

```java
@Test
void obtenerNumeroLibre() {
    // por defecto es el ultimo
    this.sistema.getGestorNumeros().setGeneradorNumeroLibre(new GeneradorUltimo());
    assertEquals("2214444559", this.sistema.obtenerNumeroLibre());

    this.sistema.getGestorNumeros().setGeneradorNumeroLibre(new GeneradorPrimero());
    assertEquals("2214444554", this.sistema.obtenerNumeroLibre());

    this.sistema.getGestorNumeros().setGeneradorNumeroLibre(new GeneradorRandom());
    assertNotNull(this.sistema.obtenerNumeroLibre());
}
```