TP Refactoring

Ejercicio N*3

Mayo 2024

Cátedra: Orientación a Objetos 2 - UNLP.

Alumno: Massera Felipe Carlos.

Legajo 20676/5.

Sumario

1-Mal olor: Switch statements

En la clase GestorNumeroDisponibles tenemos el siguiente método:

```
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;
}
```

En la programación orientada a objetos, el uso de la instrucción ‘switch’ puede indicar la necesidad de distribuir el código en varias clases y lograr el mismo comportamiento utilizando el polimorfismo. Para solucionar este "mal olor" en el código, es necesario aplicar técnicas de refactorización.

Refactoring: Replace Conditional Logic with Polymorphism

Para este Refactoring, voy a crear una nueva clase abstracta llamada "TipoGenerador", que tendrá las subclases "GeneradorPrimero", "GeneradorUltimo" y "GeneradorRandom".
Después, moveré el método "obtenerNumeroLibre()" a la clase "TipoGenerador" como un método abstracto, y lo implementaré en las subclases mencionadas.
En la clase "GestorNumeroDisponibles", modificaré la variable de instancia "tipoGenerador" para que, en lugar de ser un "String", sea una instancia de "TipoGenerador". Además, el método "obtenerNumeroLibre()" enviará el mensaje "obtenerNumeroLibre()" a la variable "tipoGenerador". Las clases resultantes serian:

Clase TipoGenerador :

```java
public abstract class TipoGenerador {

    public abstract String obtenerNumeroLibre(SortedSet<String> lineas);

}
```

Clase GeneradorPrimero :

```java
public class GeneradorPrimero extends TipoGenerador{

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

}
```

Clase GeneradorRandom :

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

Clase GeneradorUltimo:

```java
public class GeneradorUltimo extends TipoGenerador {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.last();
        lineas.remove(linea);
        return linea;
    }
}
```

Clase GestorNumerosDisponibles:

```java
public class GestorNumerosDisponibles {
    private SortedSet<String> lineas = new TreeSet<>();
    private TipoGenerador tipoGenerador = new GeneradorUltimo(); // Inicialización por defecto

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

    public String obtenerNumeroLibre() {
        return tipoGenerador.obtenerNumeroLibre(lineas);
    }

    public void cambiarTipoGenerador(TipoGenerador nuevoTipoGenerador) {
        this.tipoGenerador = nuevoTipoGenerador;
    }
}
```

Se procedió a realizar modificaciones en el test denominado “obtenerNumeroLibre()”, de acuerdo con los cambios realizados en el código objeto bajo evaluación.

@Test Antes:

```java
54    @Test
55    void obtenerNumeroLibre() {
56        // por defecto es el ultimo
57        assertEquals("2214444559", this.sistema.obtenerNumeroLibre());
58
59        this.sistema.getGestorNumeros().cambiarTipoGenerador("primero");
60        assertEquals("2214444554", this.sistema.obtenerNumeroLibre());
61
62        this.sistema.getGestorNumeros().cambiarTipoGenerador("random");
63        assertNotNull(this.sistema.obtenerNumeroLibre());
64    }
```

@Test Después:

```java
@Test
void obtenerNumeroLibre() {
    assertEquals("2214444559", this.sistema.obtenerNumeroLibre());

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

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

2-Mal olor: Feature envy

En las clases GeneradorPrimero, GeneradorUltimo y GeneradorRandom se identifica una práctica que puede generar envidia de atributos, ya que cada una de estas clases se encarga de eliminar un dato de una colección, acción que no debería ser de su incumbencia. En lugar de ello, la responsabilidad de eliminar elementos de la colección debería recaer en la clase GestorNumerosDisponibles.

```
 4
 5  public class GeneradorUltimo extends TipoGenerador {
 6
 7      @Override
 8      public String obtenerNumeroLibre(SortedSet<String> lineas) {
 9          // TODO Auto-generated method stub
10          String linea = lineas.last();
11          lineas.remove(linea);
12          return linea;
13      }
14
15  }
```

```
 5  public class GeneradorPrimero extends TipoGenerador {
 6
 7      @Override
 8      public String obtenerNumeroLibre(SortedSet<String> lineas) {
 9          // TODO Auto-generated method stub
10          String linea = lineas.first();
11          lineas.remove(linea);
12          return linea;
13      }
14
15  }
```

```
 7  public class GeneradorRandom extends TipoGenerador {
 8
 9      @Override
10      public String obtenerNumeroLibre(SortedSet<String> lineas) {
11          // TODO Auto-generated method stub
12          String linea = new ArrayList<String>(lineas)
13                  .get(new Random().nextInt(lineas.size()));
14          lineas.remove(linea);
15          return linea;
16      }
17
18  }
```

Para abordar esta problemática, procederemos a realizar una refactorización en el diseño del código.

Refactoring utilizado: Extract Method

En el método "obtenerNumeroLibre()" de la clase "GestorNumerosDisponibles", realizaremos una refactorización para trasladar la sentencia "lineas.remove(linea)" que estaba presente en los métodos de las

subclases "GeneradorPrimero", "GeneradorUltimo" y "GeneradorRandom". Ahora, esta sentencia se ubicará en el método "obtenerNumeroLibre()" de la clase "GestorNumerosDisponibles".

```
public String obtenerNumeroLibre( ) {
        String nuevaLinea = this.tipoGenerador.obtenerNumeroLibre(lineas);
        lineas.remove(nuevaLinea);
        return nuevaLinea;
    }
```

Sub clases de TipoGenerador luego de realizar el Refactoring extract method.

GeneradorPrimero- GeneradorUltimo - GeneradorRandom:

```
public class GeneradorPrimero extends TipoGenerador{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.first();
        return linea;
    }
}
```

```
public class GeneradorUltimo extends TipoGenerador {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.last();
        return linea;
    }
}
```

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

3-Mal olor: Speculative Generality

En las subclases de "TipoGenerador" (GeneradorPrimero, GeneradorUltimo y GeneradorRandom), se identifica la presencia de una variable que se almacena únicamente para posibles usos futuros, cuando en realidad solo se devuelve en la línea siguiente. Por consiguiente, se evidencia la posibilidad de realizar una refactorización.

```java
public class GeneradorPrimero extends TipoGenerador {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        // TODO Auto-generated method stub
        String linea = lineas.first();
        return linea;
    }

}
```

```java
public class GeneradorUltimo extends TipoGenerador {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        // TODO Auto-generated method stub
        String linea = lineas.last();
        return linea;
    }

}
```

```java
public class GeneradorRandom extends TipoGenerador {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        // TODO Auto-generated method stub
        String linea = new ArrayList<String>(lineas)
                .get(new Random().nextInt(lineas.size()));
        return linea;
    }

}
```

Refactor: Inline Variable

En esta refactorización, se ha eliminado la asignación de las búsquedas de números disponibles (primero, último y random) a una variable "linea", permitiendo así retornar directamente los resultados de cada búsqueda requerida.

GeneradorPrimero- GeneradorUltimo - GeneradorRandom:

```java
public class GeneradorPrimero extends TipoGenerador{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return lineas.first();
    }
}
```

```java
public class GeneradorUltimo extends TipoGenerador {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return lineas.last();
    }
}
```

```java
public class GeneradorRandom extends TipoGenerador {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        return new ArrayList<>(lineas).get(new Random().nextInt(lineas.size()));
    }
}
```

4-Mal Olor: Feature Envy

En la clase "Empresa", se observa en el método "agregarNumeroTelefono()" la solicitud a la clase "GestorNumerosDisponibles" de su colección de líneas para consultar si contiene o no una línea específica recibida como parámetro. En caso de no existir, se agrega la línea recibida como parámetro a la colección propia de la clase "GestorNumerosDisponibles", lo cual genera un acoplamiento entre las clases.

Clase Empresa:

```
13
14    public boolean agregarNumeroTelefono(String str) {
15        boolean encontre = guia.getLineas().contains(str);
16        if (!encontre) {
17            guia.getLineas().add(str);
18            encontre= true;
19            return encontre;
20        }
21        else {
22            encontre= false;
23            return encontre;
24        }
25    }
```

Refactoring: Move Method

Para solventar este problema de diseño, es necesario trasladar el método a la clase "GestorNumeroDisponibles". Posteriormente, el método que permanece en la clase "Empresa" únicamente se encargará de transmitir el mensaje a su variable "guía".

Clase Empresa:

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

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;
        }
    }
```

5-Mal olor: Speculative Generality

En la clase "GestorNumerosDisponibles", se identifica la presencia de la variable "encontré", la cual solamente se utiliza para asignarle un valor y utilizarlo como condición, y posteriormente, se asigna otro valor para retornarlo. Dado que su utilidad se limita únicamente a este método, no resulta necesaria su presencia en el mismo.

Clase GestorNumerosDisponibles:

```
27
28      public boolean agregarNumeroTelefono(String str) {
29          boolean encontre = getLineas().contains(str);
30          if (!encontre) {
31              getLineas().add(str);
32              encontre= true;
33              return encontre;
34          }
35          else {
36              encontre= false;
37              return encontre;
38          }
39      }
40  }
```

Refactoring: Inline variable

Para abordar este ajuste, prescindiremos del uso de la variable "encontré" y utilizaremos directamente la expresión "getLineas().contains(str)" como condición del "if". Además, retornaremos directamente los valores "true" o "false" en el "return".

Clase GestorNumerosDisponibles:

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

6-Mal olor: Switch Statement

En la clase "Empresa", encontramos el método "registrarUsuario()" que emplea una estructura de control "if" para verificar el tipo de cliente (ya sea persona física o jurídica).

Clase Empresa:

```java
    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: Replace Type Code with SubClasses

En este Refactoring, crearíamos dos nuevas clases: "ClienteFisico" y "ClienteJuridico", dejando la clase "Cliente" como abstracta. Posteriormente, trasladaríamos el comportamiento relevante de la clase "Cliente" a las subclases mencionadas.

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

    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;
    }
}
```

```java
public class Empresa {
    public Cliente registrarUsuarioFisico(String data, String nombre) {
        ClienteFisico var = new ClienteFisico();
        var.setNombre(nombre);
        String tel = this.obtenerNumeroLibre();
        var.setNumeroTelefono(tel);
        var.setDNI(data);
        clientes.add(var);
        return var;
    }
    public Cliente registrarUsuarioJuridico(String data, String nombre) {
        ClienteJuridico var = new ClienteJuridico();
        String tel = this.obtenerNumeroLibre();
        var.setNombre(nombre);
        var.setNumeroTelefono(tel);
        var.setCuit(data);
        clientes.add(var);
        return var;
    }
```

```java
public class ClienteJuridico extends Cliente{

    private String cuit;

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

```java
public class ClienteFisico extends Cliente{

    private String dni;

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

Se requirió ajustar los test debido a la modificación de los nombres de los métodos y a los cambios en los parámetros.

Previo a la creación de las nuevas clases:

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

```
44    @Test
45    void testAgregarUsuario() {
46        assertEquals(this.sistema.cantidadDeUsuarios(), 0);
47        this.sistema.agregarNumeroTelefono("2214444558");
48        Cliente nuevaPersona = this.sistema.registrarUsuario("2444555","Alan Turing", "fisica");
49
```

Después:

7-Mal olor: Long Method

En la clase "Empresa", se observa que los métodos "registrarUsuarioFisico" y "registrarUsuarioJuridico" llevan a cabo más de una tarea, crean una instancia de cliente y luego la agregan a la lista de clientes, lo que no respeta el principio de alta cohesión.

```
22    public Cliente registrarUsuarioFisico(String data, String nombre) {
23        ClienteFisico var = new ClienteFisico();
24        var.setNombre(nombre);
25        String tel = this.obtenerNumeroLibre();
26        var.setNumeroTelefono(tel);
27        var.setDNI(data);
28        clientes.add(var);
29        return var;
30    }
31
32    public Cliente registrarUsuarioJuridico(String data, String nombre) {
33        ClienteJuridico var = new ClienteJuridico();
34        String tel = this.obtenerNumeroLibre();
35        var.setNombre(nombre);
36        var.setNumeroTelefono(tel);
37        var.setCuit(data);
38        clientes.add(var);
39        return var;
40    }
```

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

```
@Test
    void testAgregarUsuario() {
        assertEquals(this.sistema.cantidadDeUsuarios(), 0);
        this.sistema.agregarNumeroTelefono("2214444558");
        Cliente nuevaPersona = this.sistema.registrarUsuarioFisico("2444555","Alan Turing");
```

Refactoring: Extract Method

Para refactorizar el código anterior, podemos extraer la sentencia donde agregamos el cliente a la lista de clientes en un método nuevo llamado "agregarCliente", el cual será invocado dentro de los métodos "registrarUsuarioFisico" y "registrarUsuarioJuridico".

```
public Cliente registrarUsuarioFisico(String data, String nombre) {
    ClienteFisico var = new ClienteFisico();
    var.setNombre(nombre);
    String tel = this.obtenerNumeroLibre();
    var.setNumeroTelefono(tel);
    var.setDNI(data);
    agregarUsuario(var)
    return var;
}

public Cliente registrarUsuarioJuridico(String data, String nombre) {
    ClienteJuridico var = new ClienteJuridico();
    String tel = this.obtenerNumeroLibre();
    var.setNombre(nombre);
    var.setNumeroTelefono(tel);
    var.setCuit(data);
    agregarUsuario(var)
    return var;
}

public void agregarUsuario(Cliente cliente) {
    this.clientes.add(cliente);
}
```

8-Mal olor: Inappropriate Intimacy

En los siguientes métodos, la clase "Empresa" crea una instancia de un objeto y establece sus campos uno por uno, en lugar de permitir que el objeto mismo se configure a través de su constructor.

```java
    public Cliente registrarUsuarioFisico(String data, String nombre) {
        ClienteFisico var = new ClienteFisico();
        var.setNombre(nombre);
        String tel = this.obtenerNumeroLibre();
        var.setNumeroTelefono(tel);
        var.setDNI(data);
        agregarUsuario(var);
        return var;
    }

    public Cliente registrarUsuarioJuridico(String data, String nombre) {
        ClienteJuridico var = new ClienteJuridico();
        String tel = this.obtenerNumeroLibre();
        var.setNombre(nombre);
        var.setNumeroTelefono(tel);
        var.setCuit(data);
        agregarUsuario(var);
        return var;
    }
```

Refactoring: Extract Method

La solución consiste en extraer todos los métodos "set" que la clase "Empresa" ejecuta, y trasladarlos al constructor de la clase "Cliente", junto con sus subclases. Posteriormente, se utilizará el nuevo constructor en el método original.

Clase Empresa:

```java
public Cliente registrarUsuarioFisico(String data, String nombre) {
        String tel = this.obtenerNumeroLibre();
        ClienteFisico var = new ClienteFisico(nombre,tel,data);
        agregarUsuario(var);
        return var;
    }

    public Cliente registrarUsuarioJuridico(String data, String nombre) {
        String tel = this.obtenerNumeroLibre();
        ClienteJuridico var = new ClienteJuridico(nombre,tel,data);
        agregarUsuario(var);
        return var;
    }
```

Clase Cliente:

```java
public abstract class Cliente {
    public List<Llamada> llamadas;
    private String nombre;
    private String numeroTelefono;

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

ClienteFisico:

```java
public class ClienteFisico extends Cliente{

    private String dni;

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

ClienteJuridico:

```java
public class ClienteJuridico extends Cliente{

    private String cuit;

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

9-Mal olor: Speculative Generality

En los siguientes métodos de la clase "Empresa", se emplea la variable "tel" únicamente para pasarla como argumento al constructor.

```java
public Cliente registrarUsuarioFisico(String data, String nombre) {
        String tel = this.obtenerNumeroLibre();
        ClienteFisico var = new ClienteFisico(nombre,tel,data);
        agregarUsuario(var);
        return var;
    }

    public Cliente registrarUsuarioJuridico(String data, String nombre) {
        String tel = this.obtenerNumeroLibre();
        ClienteJuridico var = new ClienteJuridico(nombre,tel,data);
        agregarUsuario(var);
        return var;
    }
```

Refactoring: Inline Variable

La solución consiste en eliminar la variable "tel" y utilizar directamente la sentencia "this.obtenerNumeroLibre()" como argumento del constructor.

Clase Empresa:

```
    public Cliente registrarUsuarioFisico(String data, String nombre) {
        ClienteFisico var = new ClienteFisico(nombre,this.obtenerNumeroLibre(),data);
        agregarUsuario(var);
        return var;
    }

    public Cliente registrarUsuarioJuridico(String data, String nombre) {
        ClienteJuridico var = new ClienteJuridico(nombre,this.obtenerNumeroLibre(),data);
        agregarUsuario(var);
        return var;
    }
```

10-Mal olor: Feature envy

En el siguiente método de la clase "Empresa" se produce envidia de atributos en la sentencia "origen.llamadas.add(llamada)", ya que "Empresa" toma la lista de llamadas de un "Cliente" y agrega una "Llamada", en lugar de delegar esa tarea a la clase "Cliente".

Clase Empresa:

```
38    public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
39        Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
40        llamadas.add(llamada);
41        origen.llamadas.add(llamada);
42        return llamada;
43    }
```

Refactoring: Extract Method

Para solucionarlo, es necesario extraer la sentencia y trasladarla a la clase "Cliente" para que se convierta en un método propio de esta clase. Además, la clase "Empresa" enviará un mensaje al "Cliente" utilizando el nuevo método declarado. De esta forma, quedaría estructurado de la siguiente manera:
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.registrarLlamada(llamada);
        return llamada;
    }
```

Clase Cliente:

```
public void registrarLlamada(Llamada llamada) {
        this.llamadas.add(llamada);
    }
```

11-Mal olor: Feature envy

En el siguiente método de la clase "Empresa" se produce envidia de atributos, ya que calcula el monto total de llamadas de un cliente tomando su lista de llamadas y obteniendo el monto de cada llamada. Esta debería ser una tarea de la clase "Cliente", y no de la clase "Empresa".

Clase Empresa:

```
45    public double calcularMontoTotalLlamadas(Cliente cliente) {
46        double c = 0;
47        for (Llamada l : cliente.llamadas) {
48            double auxc = 0;
49            if (l.getTipoDeLlamada() == "nacional") {
50                // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la
51                // llamada
52                auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
53            } else if (l.getTipoDeLlamada() == "internacional") {
54                // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la
55                // llamada
56                auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
57            }
58
59            if (cliente instanceof ClienteFisico) {
60                auxc -= auxc * descuentoFis;
61            } else if (cliente instanceof ClienteJuridico) {
62                auxc -= auxc * descuentoJur;
63            }
64            c += auxc;
65        }
66        return c;
67    }
```

Refactoring: Move Method

La solución consiste en mover el método a la clase "Cliente" y dejar en el método de la clase "Empresa" únicamente el envío de un mensaje al cliente.

Clase Empresa:

```
42      public double calcularMontoTotalLlamadas(Cliente cliente) {
43          return cliente.calcularMontoTotalLlamadas();
44      }
45
```

Clase Cliente:

```
35      public double calcularMontoTotalLlamadas() {
36          double c = 0;
37          for (Llamada l : this.llamadas) {
38              double auxc = 0;
39              if (l.getTipoDeLlamada() == "nacional") {
40                  // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
41                  auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
42              } else if (l.getTipoDeLlamada() == "internacional") {
43                  // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
44                  auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
45              }
46
47              if (this instanceof ClienteFisico) {
48                  auxc -= auxc * descuentoFis;
49              } else if (this instanceof ClienteJuridico) {
50                  auxc -= auxc * descuentoJur;
51              }
52              c += auxc;
53          }
54          return c;
55      }
```

También es necesario mover las variables de instancia "descuentoJur" y "descuentoFis" desde la clase "Empresa" hasta la clase "Cliente" para que el método siga funcionando correctamente.

Antes:

```
6   public class Empresa {
7       private List<Cliente> clientes = new ArrayList<Cliente>();
8       private List<Llamada> llamadas = new ArrayList<Llamada>();
9       private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
10      static double descuentoJur = 0.15;
11      static double descuentoFis = 0;
```

Después:

```
//CLASE EMPRESA
public class Empresa {
    private List<Cliente> clientes = new ArrayList<Cliente>();
    private List<Llamada> llamadas = new ArrayList<Llamada>();
    private GestorNumerosDisponibles guia = new
GestorNumerosDisponibles();
//CLASE CLIENTE
public abstract class Cliente {
    public List<Llamada> llamadas;
    private String nombre;
    private String numeroTelefono;
    static double descuentoJur = 0.15;
    static double descuentoFis = 0;
```

12-Mal olor: Long Method

En el siguiente método de la clase Cliente realiza el cálculo del precio de una llamada, luego aplica descuento y lo va acumulando en una variable, y realiza esto para cada llamada. Esto provoca que el método sea muy largo y realice demasiadas tareas.

Clase Cliente:

```java
public double calcularMontoTotalLlamadas() {
    double c = 0;
    for (Llamada l : this.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 (this instanceof ClienteFisico) {
            auxc -= auxc * descuentoFis;
        } else if (this instanceof ClienteJuridico) {
            auxc -= auxc * descuentoJur;
        }
        c += auxc;
    }
    return c;
}
```

Refactoring: Extract Method

El método de la clase Cliente que realiza el cálculo del precio de una llamada, seguido de la aplicación del descuento y la acumulación en una variable, presenta un exceso de responsabilidades y una longitud que compromete su legibilidad. Mi propuesta es dividir estas tareas en métodos separados, lo que permitirá una mejor estructuración y comprensión del código. Realizando dos nuevos métodos privados: "calcularPrecioLlamada()" y "aplicarDescuento()", cada uno encargado de una parte específica del proceso. En el caso de "calcularPrecioLlamada()", se enviará la llamada como parámetro y se utilizará una variable temporal "auxc" para su cálculo. Por otro lado, en "aplicarDescuento()", se recibirá como parámetro la variable "auxc" generada en el método original. Este enfoque modular simplifica el método original y mejorando su mantenimiento.

Clase Cliente:

```java
private double calcularPrecioLlamada(Llamada l) {
        double auxc = 0;
        if (l.getTipoDeLlamada() == "nacional") {
            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
        } else if (l.getTipoDeLlamada() == "internacional") {
            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
        }
        return auxc;
    }

    private double aplicarDescuento(double auxc) {
        if (this instanceof ClienteFisico) {
            auxc -= auxc * descuentoFis;
        } else if (this instanceof ClienteJuridico) {
            auxc -= auxc * descuentoJur;
        }
        return auxc;
    }

    public double calcularMontoTotalLlamada() {
        double c = 0;
        for (Llamada l : this.llamadas) {
            double auxc = 0;
            auxc = calcularPrecioLlamada(l);
            auxc = aplicarDescuento(auxc);
            c += auxc;
        }
        return c;
    }
```

13-Mal olor: Switch Statements

El método de la clase Cliente presenta un "if" que pregunta por el tipo de llamada y ejecuta cálculos diferentes en función de ello. Esto viola el principio de diseño orientado a objetos. La solución implica implementar polimorfismo: declarar Llamada como una clase abstracta y crear subclases para cada tipo de llamada. Posteriormente, se moverá el método de cálculo a la clase Llamada como método abstracto, eliminando la necesidad de la estructura condicional y mejorando la cohesión del código.

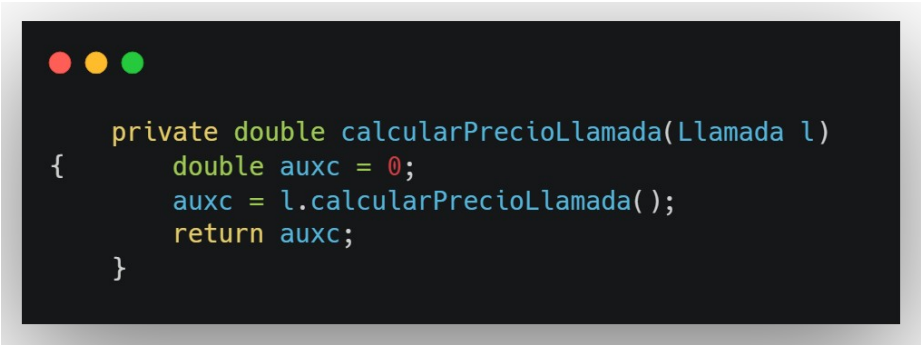
Clase Cliente:

```
39    private double calcularPrecioLlamada(Llamada l) {
40        double auxc = 0;
41        if (l.getTipoDeLlamada() == "nacional") {
42            auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
43        } else if (l.getTipoDeLlamada() == "internacional") {
44            auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
45        }
46        return auxc;
47    }
```

Refactoring: Replace Conditional with Polymorphism

La solución propuesta consiste en crear las subclases "LlamadaNacional" y "LlamadaInternacional" de la clase "Llamada", convirtiendo esta última en una clase abstracta. A continuación, se trasladará el método "calcularPrecioLlamada()" a la clase "Llamada" como un método abstracto, el cual será implementado por las subclases. Además, se eliminará la variable de instancia "tipoDeLlamada" de la clase "Llamada", junto con su getter, simplificando así la estructura y responsabilidades de la clase.

Clase Cliente:



Clase Llamada:

```java
public abstract class Llamada {
    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 String getRemitente() {
        return destino;
    }

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

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

Clase LlamadaNacional y clase LlamadaInternacional:

```java
public class LlamadaNacional extends Llamada{

    public LlamadaNacional(String origen, String destino, int duracion) {
        super(origen, destino, duracion);
    }
    @Override
    public double calcularPrecioLlamada() {
        return getDuracion() * 3 + (getDuracion() * 3 * 0.21);
    }
}
public class LlamadaInternacional extends Llamada {

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

    @Override
    public double calcularPrecioLlamada() {
        return getDuracion() * 150 + (getDuracion() * 150 * 0.21) + 50;
    }
}
```

Esta refactorización también requiere modificar la clase "Empresa". El método "registrarLlamada" se dividirá en dos métodos distintos: "registrarLlamadaNacional" y "registrarLlamadaInternacional".

Clase Empresa Antes:

```
36    public Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
37        Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
38        llamadas.add(llamada);
39        origen.registrarLlamada(llamada);
40        return llamada;
41    }
```

Clase Empresa Después:

```
36    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
37        Llamada llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
38        llamadas.add(llamada);
39        origen.registrarLlamada(llamada);
40        return llamada;
41    }
42    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
43        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
44        llamadas.add(llamada);
45        origen.registrarLlamada(llamada);
46        return llamada;
47    }
```

Esta refactorización también implica que se deban modificar los test debido al cambio de nombre de los métodos y los cambios en los parámetros.

@Test Antes:

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

    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);

    assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisca), 0.01);
    assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);
    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));
    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));
}
```

@Test Después:

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

    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);

    assertEquals(11454.64, this.sistema.calcularMontoTotalLlamadas(emisorPersonaFisca), 0.01);
    assertEquals(2445.40, this.sistema.calcularMontoTotalLlamadas(emisorPersonaJuridica), 0.01);
    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaFisica));
    assertEquals(0, this.sistema.calcularMontoTotalLlamadas(remitentePersonaJuridica));
}
```

14-**Mal olor: Switch Statements**

En el siguiente método de la clase Cliente se utiliza un "if" que pregunta por la clase de un objeto y, en función de esta, aplica diferentes descuentos.

Clase Cliente:

```
46    private double aplicarDescuento(double auxc) {
47        if (this instanceof ClienteFisico) {
48            auxc -= auxc * descuentoFis;
49        } else if (this instanceof ClienteJuridico) {
50            auxc -= auxc * descuentoJur;
51        }
52        return auxc;
53    }
```

Refactoring: Replace Conditional with Polymorphism

La solución consiste en desarmar el condicional y permitir que las clases apliquen los descuentos correspondientes. Para lograr esto, se debe convertir el método original en abstracto y protected, permitiendo que las subclases implementen el descuento específico. Además, es necesario mover las variables de instancia "descuentoFis" y "descuentoJur" a las clases correspondientes.

Clase Cliente:

```
protected abstract double aplicarDescuento(double auxc);
```

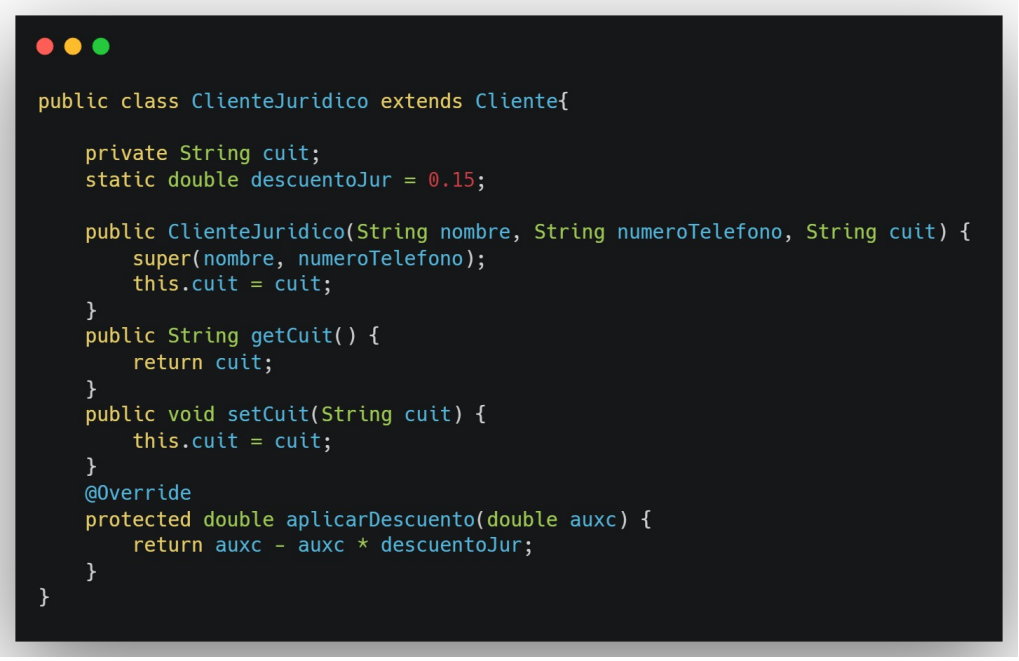
Clase ClienteFisico:

```
public class ClienteFisico extends Cliente{

    private String dni;
    static double descuentoFis = 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;
    }
    @Override
    protected double aplicarDescuento(double auxc) {
        return auxc - auxc * descuentoFis;
    }
```

Clase ClienteJuridico:



```
public class ClienteJuridico extends Cliente{

    private String cuit;
    static double descuentoJur = 0.15;

    public ClienteJuridico(String nombre, String numeroTelefono, String cuit) {
        super(nombre, numeroTelefono);
        this.cuit = cuit;
    }
    public String getCuit() {
        return cuit;
    }
    public void setCuit(String cuit) {
        this.cuit = cuit;
    }
    @Override
    protected double aplicarDescuento(double auxc) {
        return auxc - auxc * descuentoJur;
    }
}
```

15-Mal olor: Message Chains

En la clase "Cliente", el método "calcularMontoTotalLlamadas()" llama al método "calcularPrecioLlamada()", que únicamente envía un mensaje al parámetro recibido y retorna el resultado. Esta tarea podría ser realizada sin la necesidad de envíos de parámetros directamente por el primer método.

Clase Cliente:

```
41      private double calcularPrecioLlamada(Llamada l) {
42          double auxc = 0;
43          auxc = l.calcularPrecioLlamada();
44          return auxc;
45      }
46      public double calcularMontoTotalLlamadas() {
47          double c = 0;
48          for (Llamada l : this.llamadas) {
49              double auxc = 0;
50              auxc = calcularPrecioLlamada(l);
51              auxc = aplicarDescuento(auxc);
52              c += auxc;
53          }
54          return c;
55      }
56  }
57
```

Refactoring: Hide Delegate

La solución propuesta implica eliminar el método "calcularPrecioLlamada()" y delegar directamente en el método "calcularMontoTotalLlamadas()" la responsabilidad de realizar dicha tarea.

Clase Cliente:

```
    protected abstract double aplicarDescuento(double auxc);

    public double calcularMontoTotalLlamadas() {
        double c = 0;
        for (Llamada l : this.llamadas) {
            double auxc = 0;
            auxc = l.calcularPrecioLlamada();
            auxc = aplicarDescuento(auxc);
            c += auxc;
        }
        return c;
    }
```

16-Mal olor: Duplicate Code

En el siguiente método de la clase Cliente se recorre una colección aplicando una serie de métodos para luego retornar un resultado, lo mismo que realiza un “stream()”.

Clase Cliente:

```
41      public double calcularMontoTotalLlamadas() {
42          double c = 0;
43          for (Llamada l : this.llamadas) {
44              double auxc = 0;
45              auxc = l.calcularPrecioLlamada();
46              auxc = aplicarDescuento(auxc);
47              c += auxc;
48          }
49          return c;
50      }
51  }
52
```

Refactoring: Substitute Algorithm

La solución consiste en reemplazar este algoritmo con un "stream()", ya que proporciona una implementación más eficiente y concisa. El "stream()" obtendrá el precio de cada llamada, aplicará el descuento correspondiente a cada precio y, finalmente, sumará todos los valores retornando el resultado final.

Clase Cliente:

```
public double calcularMontoTotalLlamadas() {
    return this.llamadas.stream().mapToDouble(
            llamada -> llamada.calcularPrecioLlamada() - this.aplicarDescuento(llamada.calcularPrecioLlamada()))
            .sum();
}
```

17-Mal olor: Declaración de atributo público

En la clase Cliente, el atributo "llamadas" es público, lo que viola el principio de encapsulamiento de la clase.

Clase Cliente:

```
6   public abstract class Cliente {
7       public List<Llamada> llamadas;
8       private String nombre;
9       private String numeroTelefono;
10
```

Refactoring: Encapsulate Field

La solución es volverla privada.

Clase Cliente:

```
public abstract class Cliente {
    private List<Llamada> llamadas;
    private String nombre;
    private String numeroTelefono;
```

18-Mal olor: Declaración de atributo público

En la clase ClienteFisico, el atributo "descuentoFis" es público, lo cual viola el principio de encapsulamiento de la clase.

Clase ClienteFisico:

```
3  public class ClienteFisico extends Cliente{
4
5      private String dni;
6      static double descuentoFis = 0;
7
```

Refactoring: Encapsulate Field

La solución es volverla privada.

Clase ClienteFisico:

```
public class ClienteFisico extends Cliente{
    private String dni;
    private static double descuentoFis = 0;
```

19-Mal olor: Declaración de atributo público

AL igual que el mal olor N*18. En la clase ClienteJuridico, el atributo “descuentoJur” es público lo cual viola el principio de encapsulamiento de la clase.

Clase ClienteJuridico:

```
3  public class ClienteJuridico extends Cliente{
4
5      private String cuit;
6      static double descuentoJur = 0.15;
7
```

Refactoring: Encapsulate Field

La solución es volverla privada.

Clase ClienteJuridico:

```
public class ClienteJuridico extends Cliente{
    private String cuit;
    private static double descuentoJur = 0.15;
```

20-Mal olor: Speculative Generality

La variable de instancia "llamadas" de la clase Empresa solo se utiliza para agregar elementos a la lista, sin tener otro uso significativo. Además, para obtener las llamadas, se puede acceder directamente a la lista de llamadas de cada Cliente, lo que hace que la información esté duplicada.

Clase Empresa:

```
6  public class Empresa {
7      private List<Cliente> clientes = new ArrayList<Cliente>();
8      private List<Llamada> llamadas = new ArrayList<Llamada>();
9      private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
10
```

Refactoring: Eliminar la variable de instancia

La solución consiste en eliminar la variable de instancia "llamadas". Esto requerirá modificar los métodos "registrarLlamadaNacional()" y "registrarLlamadaInternacional()", eliminando la sentencia "llamadas.add(llamada)".

Clase Empresa:

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

Esta modificación implicará cambiar los métodos "registrarLlamadaNacional()" y "registrarLlamadaInternacional()" en la clase Empresa, eliminando la sentencia "llamadas.add(llamada)".

Antes Clase Empresa:

```
35     public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
36         Llamada llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
37         llamadas.add(llamada);
38         origen.registrarLlamada(llamada);
39         return llamada;
40     }
41     public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
42         Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
43         llamadas.add(llamada);
44         origen.registrarLlamada(llamada);
45         return llamada;
46     }
47
```

Después Clase Empresa:

```
35    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
36        Llamada llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
37        origen.registrarLlamada(llamada);
38        return llamada;
39    }
40    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
41        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
42        origen.registrarLlamada(llamada);
43        return llamada;
44    }
45
```

21-Mal olor: Feature Envy

En los siguientes métodos de la clase Empresa se produce envidia de atributos, ya que Empresa instancia objetos de la clase Llamada y luego se los pasa por parámetro a un Cliente. En realidad, el Cliente debería encargarse de instanciar estos objetos, ya que la clase Empresa no debería conocer los detalles de la clase Llamada.

Clase Empresa:

```
35    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
36        Llamada llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
37        origen.registrarLlamada(llamada);
38        return llamada;
39    }
40    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
41        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
42        origen.registrarLlamada(llamada);
43        return llamada;
44    }
45
```

Refactoring: Extract Method

La solución consiste en trasladar la instanciación del objeto "Llamada" a la clase Cliente. Esto implicará la necesidad de modificar el método "registrarLlamada()" en la clase Cliente. Para diferenciar entre los dos tipos de Llamada, será necesario dividir este método en dos: "registrarLlamadaNacional()" y "registrarLlamadaInternacional()". Estos nuevos métodos, en lugar de retornar "void", pasarán a retornar una instancia de la clase Llamada.

Después Clase Empresa:

```
35    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
36        return origen.registrarLlamadaNacional(destino,duracion);
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
40        return origen.registrarLlamadaInternacional(destino,duracion);
41    }
```

Antes Clase Cliente:

```
33    public void registrarLlamada(Llamada llamada) {
34        this.llamadas.add(llamada);
35    }
36
```

Después Clase Cliente:

```
33    public Llamada registrarLlamadaNacional(Cliente origen,Cliente destino, int duracion) {
34        LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
35        this.llamadas.add(llamada);
36        return llamada;
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente origen,Cliente destino, int duracion) {
40        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
41        this.llamadas.add(llamada);
42        return llamada;
43    }
44
```

22-Mal olor: Dead Code

En la clase “Cliente” podemos ver que ahora tenemos los métodos “registrarLlamadaNacional()” y “registrarLlamadaInternacional()” los cuales reciben como parámetro “Cliente origen” el cual ya no es necesario porque el origen pasaría a ser el mismo Cliente que realizaría la instancia de la clase Llamada.

Clase Cliente:

```
33    public Llamada registrarLlamadaNacional(Cliente origen,Cliente destino, int duracion) {
34        LlamadaNacional llamada = new LlamadaNacional(origen.getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
35        this.llamadas.add(llamada);
36        return llamada;
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente origen,Cliente destino, int duracion) {
40        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
41        this.llamadas.add(llamada);
42        return llamada;
43    }
44
```

Refactoring: Remove Parameter

Para resolver esto en el proceso de refactorización, simplemente eliminamos el parámetro que ya no se utiliza.

Después Clase Cliente:

```
33    public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {
34        LlamadaNacional llamada = new LlamadaNacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
35        this.llamadas.add(llamada);
36        return llamada;
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {
40        Llamada llamada = new LlamadaInternacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
41        this.llamadas.add(llamada);
42        return llamada;
43    }
```

Después Clase Empresa:

```
35    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
36        return origen.registrarLlamadaNacional(destino,duracion);
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
40        return origen.registrarLlamadaInternacional(destino,duracion);
41    }
```

23-Mal olor: Duplicate Code

En este caso, observamos que ambos métodos "registrarLlamadaNacional()" y "registrarLlamadaInternacional()" de la clase Cliente agregan una llamada a la lista, lo cual resulta en la duplicación de código.

Clase Cliente:

```
33    public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {
34        LlamadaNacional llamada = new LlamadaNacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
35        this.llamadas.add(llamada);
36        return llamada;
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {
40        Llamada llamada = new LlamadaInternacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
41        this.llamadas.add(llamada);
42        return llamada;
43    }
```

Refactoring: Extract Method

Procedemos a resolver este problema de duplicación de código extrayendo un método privado llamado "agregarLlamada()".

Clase Cliente:

```
33    public Llamada registrarLlamadaNacional(Cliente destino, int duracion) {
34        LlamadaNacional llamada = new LlamadaNacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
35        agregarLlamada(llamada);
36        return llamada;
37    }
38
39    public Llamada registrarLlamadaInternacional(Cliente destino, int duracion) {
40        Llamada llamada = new LlamadaInternacional(getNumeroTelefono() ,destino.getNumeroTelefono(),duracion);
41        agregarLlamada(llamada);
42        return llamada;
43    }
44
45    private void agregarLlamada(Llamada llamada) {
46        this.llamadas.add(llamada);
47    }
```

24-Mal olor: Duplicate Code

En las clases "LlamadaInternacional" y "LlamadaNacional", el método "calcularPrecioLlamada()" muestra una serie de pasos repetitivos que podríamos interpretar como "calcularPrecioBase()", "calcularIva()" y "calcularAdicional()".

Clase LlamadaInternacional:

```
@Override
public double calcularPrecioLlamada() {
    return getDuracion() * 150 + (getDuracion() * 150 * 0.21) + 50;
}
```

Clase LlamadaNacional:

```java
@Override
public double calcularPrecioLlamada() {
    return getDuracion() * 3 + (getDuracion() * 3 * 0.21);
}
```

Refactoring: Form Template Method

La solución para resolver este problema de diseño es implementar un patrón de diseño Template Method. Para ello, haremos que el método "calcularPrecioLlamada()" esté definido en la clase padre Llamada, indicando los pasos generales como "this.calcularPrecioBase()", "this.calcularIva()" y "this.calcularAdicional()". Estos últimos serán métodos abstractos, que deberán ser implementados en cada subclase LlamadaNacional y LlamadaInternacional.

Antes Clase Llamada:

```
public abstract double calcularPrecioLlamada();
```

Después Clase Llamada:

```
public double calcularPrecioLlamada() {
    return this.calcularPrecioBase() + this.calcularIva() + this.calcularAdicional();
}

protected abstract double calcularPrecioBase();
protected abstract double calcularIva();
protected abstract double calcularAdicional();
```

Clase LlamadaInternacional:

Clase LlamadaNacional:

```
9     @Override
10    protected double calcularPrecioBase() {
11        return getDuracion() * 150;
12    }
13    @Override
14    protected double calcularIva() {
15        return (getDuracion() * 150 * 0.21);
16    }
17    @Override
18    protected double calcularAdicional() {
19        return 50;
20    }
21
```

```
@Override
protected double calcularPrecioBase() {
    return getDuracion() * 3 ;
}

@Override
protected double calcularIva() {
    return (getDuracion() * 3 * 0.21);
}

@Override
protected double calcularAdicional() {
    return 0;
}
```

25-Mal olor: Nombre No Representativo

En la clase Empresa, el método "agregarNumeroTelefono()" recibe un parámetro "str" que hace referencia a un número de teléfono.

Clase Empresa:

```
11      public boolean agregarNumeroTelefono(String str) {
12          return guia.agregarNumeroTelefono(str);
13      }
14
```

Refactoring: Rename Parameter

En este caso, resolveré el problema cambiando el nombre del parámetro a "numeroTelefono".

Clase Empresa:

```
11      public boolean agregarNumeroTelefono(String numeroTelefonico) {
12          return guia.agregarNumeroTelefono(numeroTelefonico);
13      }
14
```

26-Mal olor: Nombre No Representativo

En la clase Empresa, los métodos "registrarNumeroFisico()" y "registrarNumeroJuridico()" reciben un parámetro "data" que hace referencia a un DNI, y internamente utiliza una variable "var" que hace referencia a un Cliente.

Clase Empresa:

```
19      public Cliente registrarUsuarioFisico(String data, String nombre) {
20          ClienteFisico var = new ClienteFisico(nombre,this.obtenerNumeroLibre(),data);
21          agregarUsuario(var);
22          return var;
23      }
24
25      public Cliente registrarUsuarioJuridico(String data, String nombre) {
26          ClienteJuridico var = new ClienteJuridico(nombre,this.obtenerNumeroLibre(),data);
27          agregarUsuario(var);
28          return var;
29      }
30
```

Refactoring: Rename Parameter

En este caso, resolveremos el problema cambiando el nombre del parámetro a "dni" y el nombre de la variable interna a "cliente".

Clase Empresa:

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

public Cliente registrarUsuarioJuridico(String dni, String nombre) {
    ClienteJuridico cliente = new ClienteJuridico(nombre,this.obtenerNumeroLibre(),dni);
    agregarUsuario(cliente);
    return cliente;
}
```

27-Mal olor: Nombre No Representativo

En la clase Cliente, ClienteJuridico y ClienteFisico, en el método "aplicarDescuento()", tenemos un parámetro "auxc" que hace referencia a un precio.

Clase Cliente:

```
protected abstract double aplicarDescuento(double auxc);
```

Clase ClienteJuridico:

```
@Override
protected double aplicarDescuento(double auxc) {
    return auxc - auxc * descuentoJur;
}
```

Clase ClienteFisico:

```
@Override
protected double aplicarDescuento(double auxc) {
    return auxc - auxc * descuentoFis;
}
```

Refactoring: Rename Variable

En este caso, procederé a renombrar el parámetro como "precio".

Clase Cliente:

```
protected abstract double aplicarDescuento(double precio);
```

Clase ClienteJuridico:

```
@Override
protected double aplicarDescuento(double precio) {
    return precio - precio * descuentoJur;
}
```

Clase ClienteFisico:

```
@Override
protected double aplicarDescuento(double precio) {
    return precio - precio * descuentoFis;
}
```

28-Mal olor: Nombre No Representativo

En la clase "GestorNumerosDisponibles", el método "cambiarTipoGenerador()" recibe el parámetro "valor", el cual hace referencia a una instancia de "TipoGenerador".

Clase GestorNumerosDisponibles:

```
    public void cambiarTipoGenerador(TipoGenerador valor) {
        this.tipoGenerador = valor;
    }
```

Refactoring: Rename Parameter

En este caso, se resuelve renombrando el parámetro a "tipoGenerador".

Clase GestorNumerosDisponibles:

```
public void cambiarTipoGenerador(TipoGenerador tipoGenerador) {
    this.tipoGenerador = tipoGenerador;
}
```

29-Mal olor: Nombre No Representativo

En la clase "GestorNumerosDisponibles", el método "agregarNumeroTelefono()" recibe un parámetro denominado "str" que hace referencia a un número de teléfono.

Clase GestorNumerosDisponibles:

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

Refactoring: Rename Parameter

En este caso, resolvemos renombrando el parámetro como "numeroTelefono".

Clase GestorNumerosDisponibles:

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

30-Mal olor: Nombre No Representativo

En la clase "Cliente","ClienteFisico" y "ClienteJuridico", el método "aplicarDescuento()" recibe un parámetro denominado "precio" que hace referencia a un número de teléfono.

Clase ClienteFisico:

```
@Override
protected double aplicarDescuento(double precio) {
    return precio - precio * descuentoFis;
}
```

Clase ClienteJuridico:

```
@Override
protected double aplicarDescuento(double precio) {
    return precio - precio * descuentoJur;
}
```

Clase Cliente:

```
protected abstract double aplicarDescuento(double precio);
```

Refactoring: Rename Parameter

En este caso, resolvemos renombrando el parámetro como "monto".

Clase Cliente:

```
protected abstract double aplicarDescuento(double monto);
{
```

Clase ClienteFisico:

```
@Override
protected double aplicarDescuento(double monto) {
    return monto - monto * descuentoFis;
}
```

Clase ClienteJuridico:

```
@Override
protected double aplicarDescuento(double monto) {
    return monto - monto * descuentoJur;
}
```

31-Mal olor: Duplicate Code

En las clases "ClienteFisico" y "ClienteJuridico", el método "aplicarDescuento()" muestra una serie de pasos repetitivos que podríamos interpretar como "getDescuento()" y "setDescuento()".

Clase Cliente:

```
protected abstract double aplicarDescuento(double monto);
{
```

Clase ClienteFisico:

```
@Override
protected double aplicarDescuento(double monto) {
    return monto - monto * descuentoFis;
}
```

Clase ClienteJuridico:

```
@Override
protected double aplicarDescuento(double monto) {
    return monto - monto * descuentoJur;
}
```

Refactoring: Form Template Method

La solución para resolver este problema de diseño es implementar un patrón de diseño Template Method. Para ello, haremos que el método "aplicarDescuento()" esté definido en la clase padre “Cliente”, indicando los pasos generales como "this.getDescuento ()" y "this.setDescuento ()". Estos últimos serán métodos abstractos, que deberán ser implementados en cada subclase ClienteFisico y ClienteJuridico.

Clase Cliente:

```
public abstract double getDescuento();

public abstract void setDescuento(double descuento);

public double aplicarDescuento(double monto) {
    return monto * this.getDescuento();
}
```

Clase ClienteFisico:

```
@Override
public double getDescuento() {
    return ClienteFisico.descuentoFis;
}

@Override
public void setDescuento(double descuento) {
    ClienteFisico.descuentoFis = descuento;
}
```

Clase ClienteJuridico:

```
@Override
public double getDescuento() {
    return ClienteJuridico.descuentoJur;
}

@Override
public void setDescuento(double descuento) {
    ClienteJuridico.descuentoJur = descuento;
}
```