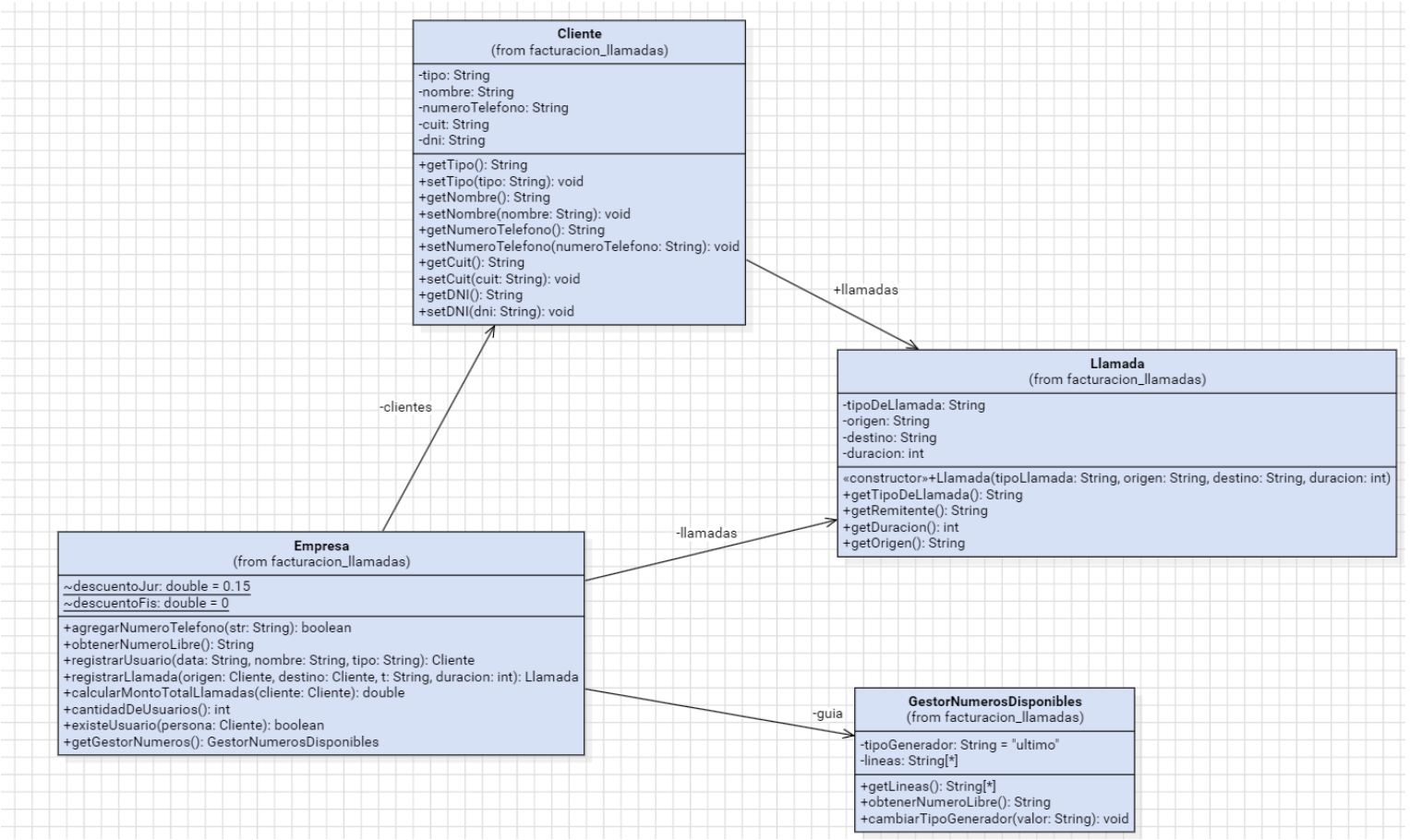
Refactoring

Ej 3 - Orientación a Objetos 2

Melo, Ignacio Nicolás: 20678/8
Martinez Osti, María Josefina: 21583/3

24/05/2024
Facultad de Informática UNLP

UML inicial:



Aclaración: El UML está subido también a la carpeta en donde entregamos el ejercicio, pierde mucha calidad al ponerlo como imagen en el documento.

Malos olores: envidia de atributos, declaración de atributo público, clase anémica

La envidia de atributos se da en la clase Empresa hacia la clase Cliente. El atributo público es la lista de la clase cliente. El método CalcularMontoTotalLlamadas accede a cliente.llamadas. Nos conviene que directamente cada cliente tenga sus llamadas propias y calcule el costo de las mismas devolviendo su valor permitiendo así que el método de la clase empresa simplemente llame a dicho método.

Algo parecido ocurre con registrarLlamada. Es mejor que cada cliente se ocupe de hacer este registro. Estos cambios nos permiten que el array de llamadas sea privado dentro de la clase Cliente y eliminar la lista de llamadas de la clase Empresa. A su vez, estos cambios hacen que Cliente deje de ser una clase anémica que solo funciona como contenedor de valores.

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

  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: Move Method, Encapsulate Field

Llevamos calcularMontoTotalLlamadas y registrarLlamada a la clase cliente. Este cambio implica también mover los valores de descuentos según el tipo de persona (las variables estáticas) a la clase Cliente.

Hacer estos cambios hace que sea innecesario tener la lista de llamadas en la clase Empresa y a su vez nos permite hacer privada la lista de llamadas en la clase Cliente.

En la clase Empresa:

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

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

En la clase Cliente:

```
public Llamada registrarLlamada(Cliente destino, String t, int duracion) {
      Llamada llamada = new Llamada(t, this.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
      this.llamadas.add(llamada);
      return llamada;
  }
    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.getTipo() == "fisica") {
                auxc -= auxc * descuentoFis;
            } else if (this.getTipo() == "juridica") {
                auxc -= auxc * descuentoJur;
            }
            c += auxc;
        }
        return c;
    }
```

Mal olor: Método largo

Detectado en la clase Cliente. El método calcularMontoTotalLlamadas primero evalúa el costo de todas las llamadas del cliente y luego aplica un descuento dependiendo del tipo de cliente que sea.

Refactoring: Extract Method

Separamos el método calcularMontoTotalLlamadas en calcularMontoLlamada y aplicarDescuento.

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

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

private 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: Nombres de variables poco descriptivos

Detectado en el método calcularMontoTotalLlamadas de la clase Cliente. Las variables c y auxc no describen con claridad qué función cumplen.

Refactoring: Rename Variable

Renombramos C como costoTotal y auxc como auxCosto.

```java
public double calcularMontoTotalLlamadas() {
    double costoTotal = 0;
    for (Llamada llamada : this.llamadas) {
        double auxCosto = calcularMontoLlamada(llamada);
        costoTotal += aplicarDescuento(auxCosto);
    }
    return costoTotal;
}
```

Mal olor: Inappropriate intimacy (intimidad inapropiada)

La clase Empresa crea a un cliente mediante los getters de la misma, comprometiendo su encapsulamiento.

```
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: Move Method

No es exactamente este patrón pero consideramos que es el más cercano. Creamos un constructor en la clase Cliente. A su vez, aplicamos rename variable sobre la variable que contiene en DNI/CUIT del cliente y le pusimos ID, que es más descriptivo de lo que contiene.

En la clase Empresa:

```
public Cliente registrarUsuario(String id, String nombre, String tipo) {
    Cliente cliente = new Cliente(tipo, nombre, this.obtenerNumeroLibre(), id);
    this.clientes.add(cliente);
    return cliente;
}
```

En la clase Cliente:

```
public Cliente(String tipo, String nombre, String numeroTelefono, String id) {
    this.tipo = tipo;
    this.nombre = nombre;
    this.numeroTelefono = numeroTelefono;
    if (tipo.equals("fisica")) {
        this.dni = id;
    } else if (tipo.equals("juridica")) {
        this.cuit = id;
    }
}
```

Mal olor: Inappropriate intimacy (intimidad inapropiada)

En la clase Cliente no es correcto que se pueda cambiar los valores de las v.i de un cliente desde fuera de la clase.

```
public void setTipo(String tipo) {
    this.tipo = tipo;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public void setNumeroTelefono(String numeroTelefono) {
    this.numeroTelefono = numeroTelefono;
}

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

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

Refactoring: Remove Setting Method

Eliminamos todos los setters de la clase Cliente.

Mal olor: Switch statement

En el la clase cliente, tanto en el constructor como en el método calcularDescuento se utilizan condicionales para accionar de manera diferente según si el cliente es físico o jurídico.

En la clase empresa:

```
public Cliente registrarUsuario(String id, String nombre, String tipo) {
    Cliente cliente = new Cliente(tipo, nombre, this.obtenerNumeroLibre(), id);
    this.clientes.add(cliente);
    return cliente;
}
```

En la clase cliente:

```
public Cliente(String tipo, String nombre, String numeroTelefono, String id) {
    this.tipo = tipo;
    this.nombre = nombre;
    this.numeroTelefono = numeroTelefono;
    if (tipo.equals("fisica")) {
        this.dni = id;
    } else if (tipo.equals("juridica")) {
        this.cuit = id;
    }
}
```

```
private 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: Replace Conditional With Polymorphism, Push Down Method, Push Down Field

A partir de cliente creamos clienteFisico y clienteJuridico, que heredan de Cliente la estructura en común e implementan los comportamientos específicos de cada tipo. Hicimos un push down method de calcularMontoLlamada a los hijos y un push down field del id (DNI, CUIT) y las variables estáticas de los descuentos. Este cambio nos obligó a modificar levemente los tests ya que cambiamos el constructor de los clientes.

En la clase Empresa:

```
public Cliente registrarUsuarioJuridico(String cuit, String nombre) {
    Cliente nuevoCliente = new clienteJuridico(nombre, this.obtenerNumeroLibre(), cuit);
    clientes.add(nuevoCliente);
    return nuevoCliente;
}
public Cliente registrarUsuarioFisico(String dni, String nombre) {
    Cliente nuevoCliente = new clienteFisico(nombre, this.obtenerNumeroLibre(), dni);
    clientes.add(nuevoCliente);
    return nuevoCliente;
}
```

En la clase Cliente:

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


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

    protected abstract double aplicarDescuento(double costo);
     ...
```

En la 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 double aplicarDescuento(double costo) {
        return costo - costo * descuentoFis;
    }

    public String getDni() {
        return dni;
    }
}
```

En la 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 double aplicarDescuento(double monto){
      return monto * (1 - descuentoJur);
    }

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

El test antes:

```
@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");
```

El test ahora:

```
@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");
```

Mal olor: Envidia de atributos

En la clase Cliente se hace el cálculo del precio de la llamada. Para esto, necesita acceder a variables de la clase llamada

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

Refactoring: Move Method

Movemos el método calcularMontoLlamada de la clase Cliente a la clase Llamada. En el método calcularMontoTotalLlamadas (de la clase Cliente) llamamos ahora al método de la clase Llamada.

En la clase Llamada:

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

En la clase Cliente:

```
public double calcularMontoTotalLlamadas() {
    double costoTotal = 0;
    for (Llamada llamada : this.llamadas) {
        double auxCosto = llamada.calcularMontoLlamada();
        costoTotal += aplicarDescuento(auxCosto);
    }
    return costoTotal;
}
```

Mal olor: Switch statement

En el método calcularMontoLlamada de la clase llamada, se utiliza un condicional para accionar de manera diferente según si la llamada es nacional o internacional.
En la clase Empresa:

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

En la clase Llamada:

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

En la clase Cliente:

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

Dado que se cambiaron los constructores, los test quedan de esta manera:

```
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);
```

Refactoring: Replace Conditional With Polymorphism, Push Down Method, Push Down Field

A partir de Llamada hicimos dos clases que heredan de ella: LlamadaLocal y LlamadaInternacional. Estas comparten estructura en común e implementan un tipo distinto de cálculo del monto. Hicimos un push down method de calcularMontoLlamada a los hijos.
En la 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);
}
```

En la clase Llamada:

```
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 abstract double calcularMontoLlamada();
```

En la clase Cliente:

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

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

En la clase LlamadaNacional:

```
public class LlamadaNacional extends Llamada {

    public LlamadaNacional(String origen, String destino, int duracion) {
        super(origen, destino, duracion);
    }
    public double calcularMontoLlamada() {
        return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
    }

}
```

En la clase LlamadaInternacional:

```
public class LlamadaInternacional extends Llamada {

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

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

Mal olor: Método largo/código repetido

En el método calcularMontoLlamada de las clases LlamadaLocal y LlamadaInternacional se dificulta el entendimiento de la cuenta que se realiza.

En la clase LlamadaNacionall:

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

En la clase LlamadaInternacional:

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

Refactoring: Extract Method

Hacemos un método en ambas clases que calcule el IVA.

En la clase LlamadaNacional:

```
public double calcularMontoLlamada() {
  double resultado = this.getDuracion() * 3;
  return resultado+= this.calcularIva(resultado)  ;
}

public double calcularIva(double precioSinIva) {
  return precioSinIva * 0.21;
}
```

En la clase LlamadaInternacional:

```
public double calcularMontoLlamada() {
    double resultado = this.getDuracion() * 150;
    return resultado+=this.calcularIva(resultado)+50;
}

public double calcularIva(double precioSinIva) {
  return precioSinIva * 0.21;
}
```

Mal olor: Código repetido

Debido al refactoring que aplicamos en el paso anterior, ahora las clases LlamadaLocal y LlamadaInternacional tienen el método calcularIVA que hace exactamente lo mismo.

Refactoring: Pull Up Method

Lo que podemos hacer entonces es subir el método calcularIVA a la clase Llamada y llamarlo desde las clases hijas.

En la clase Llamada:

```
protected double calcularIva(double precioSinIva) {
  return precioSinIva * 0.21;
}
```

En la clase LlamadaNacional:

```
public double calcularMontoLlamada() {
  double resultado = this.getDuracion() * 3;
  return resultado+= super.calcularIva(resultado) ;
}
```

En la clase LlamadaInternacional:

```
public double calcularMontoLlamada() {
    double resultado = this.getDuracion() * 150;
    return resultado += super.calcularIva(resultado) + 50;
}
```

Mal olor: Números mágicos

Tanto el IVA en la clase Llamada como el valor por segundo y el adicional en el caso de las llamadas internacionales son valores que están puestos directamente en el código y salen de la nada.

Refactoring: Replac Magic Number with Symbolic Constant

Lo que podemos hacer entonces es subir el método calcularIVA a la clase Llamada y llamarlo desde las clases hijas.

En la clase LlamadaNacional:

```
private static double precioXSegundo=3;

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

public double calcularMontoLlamada() {
    double resultado = this.getDuracion() * precioXSegundo;
    return resultado += super.calcularIva(resultado);
}
```

En la clase LlamadaInternacional:

```
private static double precioXSegundo=150;
private static double precioEstablecerLlamada= 50;

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

  public double calcularMontoLlamada() {
      double resultado = this.getDuracion() * precioXSegundo;
      return resultado += super.calcularIva(resultado) + precioEstablecerLlamada;
  }
```

En la clase Llamada:

```
private static double iva = 0.21;
public Llamada(String origen, String destino, int duracion) {
  this.origen = origen;
  this.destino = destino;
  this.duracion = duracion;
}

public abstract double calcularMontoLlamada();

protected double calcularIva(double precioSinIva) {
  return precioSinIva * iva;
}
```

Mal olor: Switch statement

En la clase GestorNumerosDisponibles se utiliza un switch que, dependiendo del tipo de generador que se está usando, retorna el próximo número de teléfono a utilizar.

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;
  }

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

Refactoring: Replace Type Code with Strategy

Lo que hicimos fue aplicar el patrón Strategy y crear tres nuevas clases: EstrategiaUltimo, EstratregiaPrimero, EstrategiaRandom. Estas clases tienen la lógica de distintos algoritmos para la búsqueda del próximo número. Dado a que cambiamos el gestor de números disponibles, tuvimos que cambiar un poco el test.

En la clase GestorNumerosDisponibles:

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

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

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

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

En la clase EstrategiaUltimo:

```
public class EstrategiaUltimo implements EstrategiaSeleccion {
  public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.last();
        lineas.remove(linea);
        return linea;
    }
}
```

En la clase EstrategiaPrimero:

```java
import java.util.SortedSet;

public class EstrategiaPrimero implements EstrategiaSeleccion{
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = lineas.first();
        lineas.remove(linea);
        return linea;
    }
}
```

En la clase EstrategiaRandom:

```java
import java.util.ArrayList;
import java.util.Random;
import java.util.SortedSet;

public class EstrategiaRandom implements EstrategiaSeleccion {
  public String obtenerNumeroLibre(SortedSet<String> lineas) {
        String linea = new ArrayList<String>(lineas).get(new Random().nextInt(lineas.size()));
        lineas.remove(linea);
        return linea;
    }
}
```

En la clase EstrategiaSeleccion:

```java
import java.util.SortedSet;

public interface EstrategiaSeleccion {

    String obtenerNumeroLibre(SortedSet<String> lineas);
}
```

El test antes:

```
@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());
}
```

El test ahora:

```
@Test
void obtenerNumeroLibre() {
  EstrategiaSeleccion estrategiaPrimero = new EstrategiaPrimero();
  EstrategiaSeleccion estrategiaRandom = new EstrategiaRandom();
  // por defecto es el ultimo
  assertEquals("2214444559", this.sistema.obtenerNumeroLibre());

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

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

Mal olor: Envidia de atributos

Las nuevas estrategias de obtención del número libre acceden a la lista de números para eliminar el próximo número a utilizar. Esta tarea no es responsabilidad suya sino del GestorNumerosDisponibles, esto se tiene que encargar GestorNumerosDisponibles.

Refactoring: Pull Up line

Este refactoring no existe pero no sabíamos en cuál encasillarlo. Hicimos que las estrategias sólo retornan el próximo número mientras que el GestorNumerosDisponibles es quien se encarga de eliminar del SortedSet la línea que se asignó.

En GestorNumerosDisponibles:

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

En EstrategiaPrimero:

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

En EstrategiaUltimo:

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

En EstrategiaRandom:

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

Mal olor: Envidia de atributos

El método agregarNumeroTelefono de la clase Empresa accede y modifica a variables de instancia de la guia.

En empresa:

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

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

public String obtenerNumeroLibre() {
  return guia.obtenerNumeroLibre();
}
```

Refactoring: Move Method, Rename Variable

Movemos el método agregarNumeroTelefono a la clase GestorNumerosDisponibles y dejamos en Empresa solamente el llamado al mismo. A su vez, cambiamos el nombre de la variable que recibe para que sea más descriptivo (número en lugar de str). A su vez, simplificamos la lógica para agregar un nuevo número de teléfono.

En empresa:

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

En GestorNumerosDisponibles:

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

Mal olor: Uso de iteradores

No consideramos que esto sea un mal olor en sí, pero el método calcularMontoTotalLlamadas de la clase Cliente utiliza un bucle for para iterar en la colección de llamadas que puede ser reemplazado fácilmente por una pipeline.

En la clase Cliente:

```
public double calcularMontoTotalLlamadas() {
    double costoTotal = 0;
    for (Llamada llamada : this.llamadas) {
        double auxCosto = llamada.calcularMontoLlamada();
        costoTotal += aplicarDescuento(auxCosto);
    }
    return costoTotal;
}
```

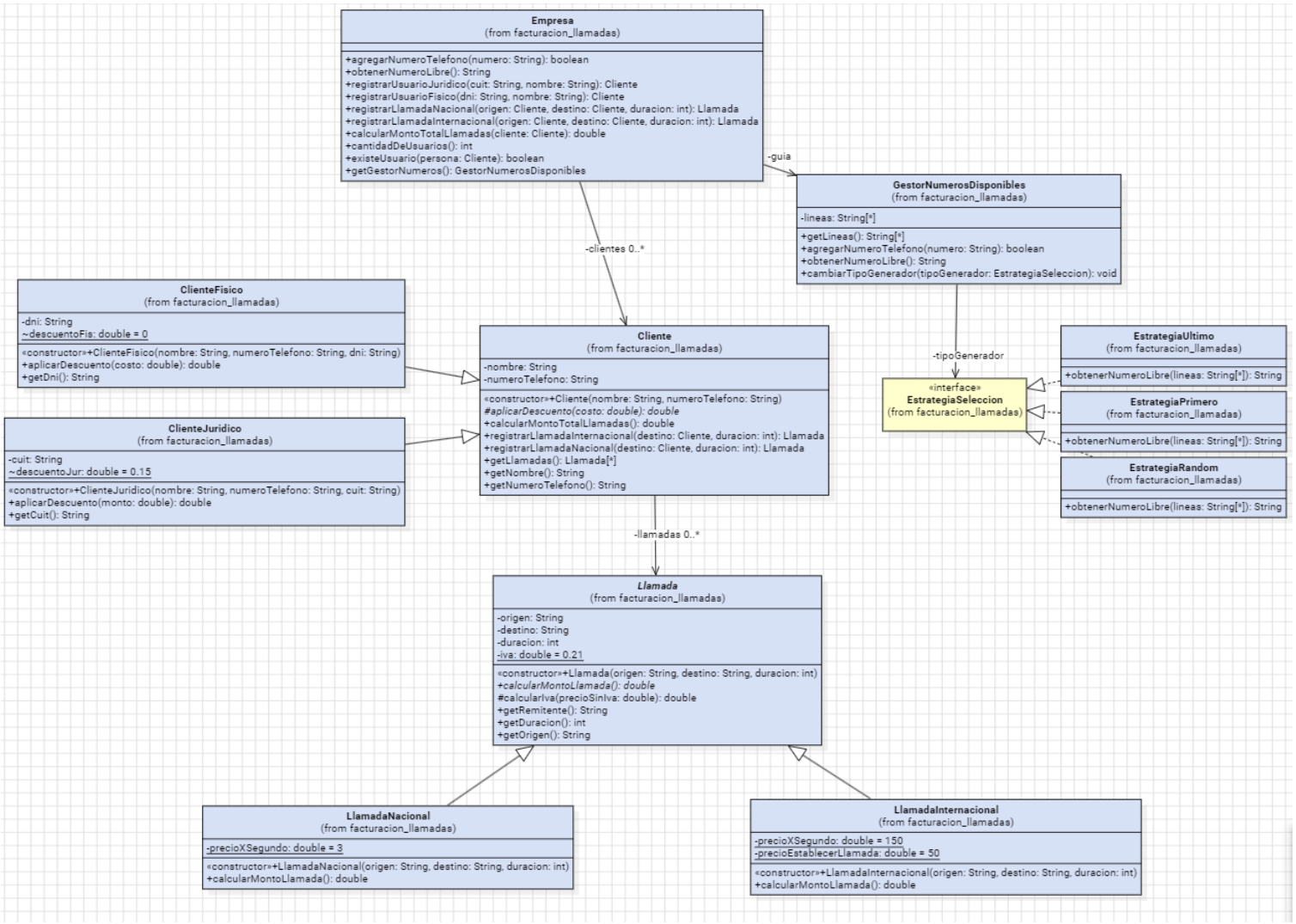
Refactoring: Replace Loop with Pipeline

Reemplazamos el for del método por una pipeline que cumple la misma función.

En la clase Cliente:

```
public double calcularMontoTotalLlamadas() {
    return this.llamadas.stream()
        .mapToDouble(llamada → aplicarDescuento(llamada.calcularMontoLlamada()))
        .sum();
}
```

UML final:


Empresa
(from facturacion_llamadas)
+agregarNumeroTelefono(numero: String): boolean
+obtenerNumeroLibre(): String
+registrarUsuarioJuridico(cuit: String, nombre: String): Cliente
+registrarUsuarioFisico(dni: String, nombre: String): Cliente
+registrarLlamadaNacional(origen: Cliente, destino: Cliente, duracion: int): Llamada
+registrarLlamadaInternacional(origen: Cliente, destino: Cliente, duracion: int): Llamada
+calcularMontoTotalLlamadas(cliente: Cliente): double
+cantidadDeUsuarios(): int
+existeUsuario(persona: Cliente): boolean
+getGestorNumeros(): GestorNumerosDisponibles
-guia
GestorNumerosDisponibles
(from facturacion_llamadas)
-lineas: String[*]
+getLineas(): String[*]
+agregarNumeroTelefono(numero: String): boolean
+obtenerNumeroLibre(): String
+cambiarTipoGenerador(tipoGenerador: EstrategiaSeleccion): void
-clientes 0..*
ClienteFisico
(from facturacion_llamadas)
-dni: String
~descuentoFis: double = 0
«constructor»+ClienteFisico(nombre: String, numeroTelefono: String, dni: String)
+aplicarDescuento(costo: double): double
+getDni(): String
Cliente
(from facturacion_llamadas)
-nombre: String
-numeroTelefono: String
«constructor»+Cliente(nombre: String, numeroTelefono: String)
#aplicarDescuento(costo: double): double
+calcularMontoTotalLlamadas(): double
+registrarLlamadaInternacional(destino: Cliente, duracion: int): Llamada
+registrarLlamadaNacional(destino: Cliente, duracion: int): Llamada
+getLlamadas(): Llamada[*]
+getNombre(): String
+getNumeroTelefono(): String
-tipoGenerador
EstrategiaUltimo
(from facturacion_llamadas)
+obtenerNumeroLibre(lineas: String[*]): String
«interface»
EstrategiaSeleccion
(from facturacion_llamadas)
EstrategiaPrimero
(from facturacion_llamadas)
+obtenerNumeroLibre(lineas: String[*]): String
EstrategiaRandom
(from facturacion_llamadas)
+obtenerNumeroLibre(lineas: String[*]): String
ClienteJuridico
(from facturacion_llamadas)
-cuit: String
~descuentoJur: double = 0.15
«constructor»+ClienteJuridico(nombre: String, numeroTelefono: String, cuit: String)
+aplicarDescuento(monto: double): double
+getCuit(): String
-llamadas 0..*
Llamada
(from facturacion_llamadas)
-origen: String
-destino: String
-duracion: int
-iva: double = 0.21
«constructor»+Llamada(origen: String, destino: String, duracion: int)
+calcularMontoLlamada(): double
#calcularIva(precioSinIva: double): double
+getRemitente(): String
+getDuracion(): int
+getOrigen(): String
LlamadaNacional
(from facturacion_llamadas)
-precioXSegundo: double = 3
«constructor»+LlamadaNacional(origen: String, destino: String, duracion: int)
+calcularMontoLlamada(): double
LlamadaInternacional
(from facturacion_llamadas)
-precioXSegundo: double = 150
-precioEstablecerLlamada: double = 50
«constructor»+LlamadaInternacional(origen: String, destino: String, duracion: int)
+calcularMontoLlamada(): double