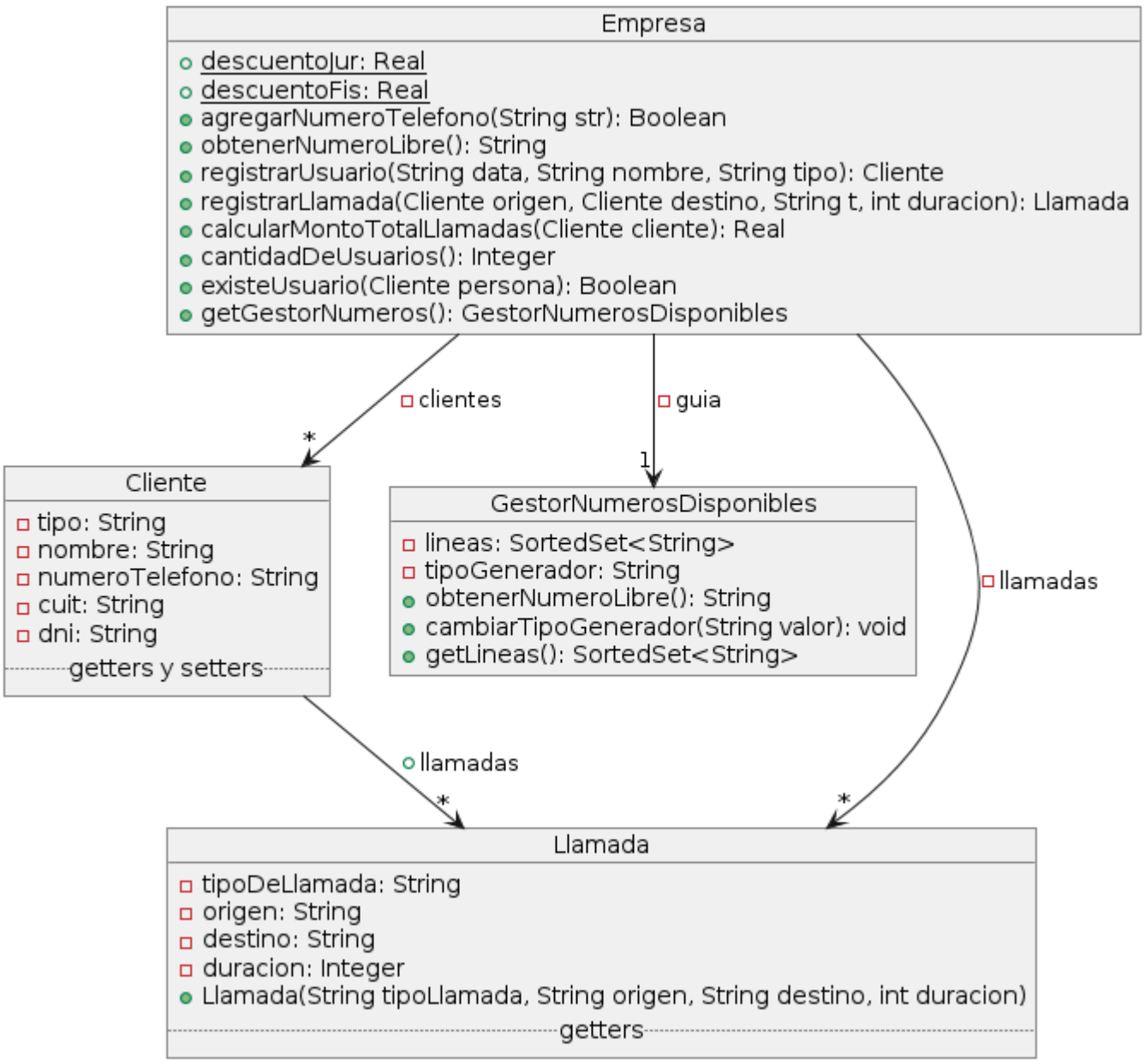
Trabajo de refactoring

Integrantes: *Martinez Coria Sofia, Macias Luciano Agustin*

UML inicial:



Aclaración: *Lo primero que se detectó como mal olor es que la clase Empresa es una Clase Dios, para solucionarlo se deben aplicar distintos refactoring que serán desarrollados a continuación.*

1) Se detectó como mal olor: "**Envidia de Atributos**", ya que la empresa accede a la colección de líneas de la variable "guia" para agregar una línea, esto es responsabilidad de la clase "GestoNumerosDisponibles"
2) Extracto de código original que pertenece a la clase Empresa:

atom-one-dark

```
public class Empresa {
            . . .
      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;
            }
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Move Method**, moviendo la lógica de agregar la línea y comprobar si existe a la clase "GestoNumerosDisponibles"
4) Extracto de código con el refactoring aplicado:

```
public class Empresa {
            . . .
```

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

```
public class GestorNumerosDisponibles{
            . . .
            boolean encontre = this.lineas.contains(str);
            if (!encontre) {
                  this.lineas.add(str);
            }
            return !encontre;
            . . .
}
```

---

1) Se detectó el mal olor que **rompe encapsulamiento**, ya que la clase Empresa se está encargando de asignar los valores a los atributos internos de la clase Clientes y de diferenciar que tipo de cliente es. Además en ambos bloques del condicional se **duplica parte del código**.

2) Extracto de código original que pertenece a la clase

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

3) El refactoring que soluciona el mal olor es **Move Method**, llevando el fragmento de código que asigna las variables de instancia al cliente a su propia clase por medio de un constructor.

4) Extracto de código con el refactoring aplicado:

```
public class Empresa {
            . . .
      public Cliente registrarUsuario(String data, String nombre, String tipo) {
            String tel = this.obtenerNumeroLibre();
            Cliente var = new Cliente(data,nombre,tipo,tel);
            clientes.add(var);
            return var;
      }
            . . .
}
```

```
public class Cliente {
            . . .
      public Cliente(String data, String nombre, String tipo,
            String numeroTelefono) {
            this.nombre = nombre;
            this.tipo = tipo;
            this.numeroTelefono = numeroTelefono;
            if (tipo.equals("fisica")) {
```

```
                  this.dni = data;
            }
            else if (tipo.equals("juridica")) {
                  this.cuit = data;
            }
      }
            . . .
}
```

1) Se detectó como mal olor un “**Switch Statements**”, ya que por medio de un if/else anidado la clase Cliente distingue los tipos de usuarios con un string y actúa en consecuencia.

2) Extracto de código original que pertenece a la clase

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

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

            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
      void testAgregarUsuario() {
            assertEquals(this.sistema.cantidadDeUsuarios(), 0);
            this.sistema.agregarNumeroTelefono("2214444558");
            Cliente nuevaPersona = this.sistema.registrarUsuario("2444555","Alan Turing", "fisica");

            assertEquals(1, this.sistema.cantidadDeUsuarios());
            assertTrue(this.sistema.existeUsuario(nuevaPersona));
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Replace Conditional with Polymorphism**, modificando la clase Cliente como una clase abstracta de la cual heredan ClienteFisico y ClienteJuridico llevándose el dni y el cuit respectivamente. Al realizar este cambio la clase Empresa no podrá

crear instancias de cada clase pasando a recibirlas por parámetros ya creados para aprovechar el polimorfismo, por lo tanto **también se deben modificar como se inicializan los test**.

4) Extractos de código con el refactoring aplicado

```
public class Empresa {
            . . .
      public Cliente registrarUsuario(Cliente cli) {
            String tel = this.obtenerNumeroLibre();
            cli.setNumeroTelefono(tel);
            clientes.add(cli);
            return cli;
      }
            . . .
}
```

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

      public Cliente(String nombre) {
            this.nombre = nombre;
      }
      public String getNombre() {
            return nombre;
      }
      public void setNombre(String nombre) {
            this.nombre = nombre;
      }
      public String getNumeroTelefono() {
            return numeroTelefono;
      }
      public void setNumeroTelefono(String numeroTelefono) {
            this.numeroTelefono = numeroTelefono;
      }
}
```

```
public class ClienteFisico extends Cliente{
      private String dni;

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

```
public class ClienteJuridico extends Cliente{
      private String cuit;

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

```
class EmpresaTest {
            . . .
      @Test
      void testcalcularMontoTotalLlamadas() {
            Cliente emisorPersonaFisca = new ClienteFisico("Brendan Eich", "11555666");
            sistema.registrarUsuario(emisorPersonaFisca);

            Cliente remitentePersonaFisica = new ClienteFisico("Doug Lea", "00000001");
            sistema.registrarUsuario(remitentePersonaFisica);

            Cliente emisorPersonaJuridica = new ClienteJuridico("Nvidia Corp", "17555222");
            sistema.registrarUsuario(emisorPersonaJuridica);

            Cliente remitentePersonaJuridica = new ClienteJuridico("Sun Microsystems", "25765432");
            sistema.registrarUsuario(remitentePersonaJuridica);

            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
      void testAgregarUsuario() {
            assertEquals(this.sistema.cantidadDeUsuarios(), 0);
            this.sistema.agregarNumeroTelefono("2214444558");
            Cliente nuevaPersona = new ClienteFisico("Alan Turing", "2444555");
            this.sistema.registrarUsuario(nuevaPersona);

            assertEquals(1, this.sistema.cantidadDeUsuarios());
            assertTrue(this.sistema.existeUsuario(nuevaPersona));
      }
            . . .
}
```

---

1) Se detectó como mal olor un par de **switch statements** en la misma clase del punto anterior (Empresa) en el método “calcularMontoLlamadas”, el que se solucionara ahora tiene que ver con el tipo de cliente.

2) Extracto de código original que pertenece a la clase

```
public class Empresa {
            . . .
      static double descuentoJur = 0.15;
      static double descuentoFis = 0;

      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;
     }
          . . .
}
```

3) Los refactoring que solucionan el mal olor son **Move Method y Move Field**, el primero para llevar el método a la clase que corresponde según el tipo de cliente, el segundo para llevarse las variables estáticas que aplican el descuento. También es conveniente separar el cálculo del descuento utilizando **Form Template Method**.

4) Extracto de código con el refactoring aplicado:

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

```
public abstract class Cliente {
          . . .
     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;
                }
                auxc -= auxc*this.getDescuento();
                c += auxc;
          }
          return c;
     }
     protected abstract double getDescuento();
     . . .
}
```

```
public class ClienteFisico extends Cliente{
          . . .
     static double descuentoFis = 0;
     @Override
     protected double getDescuento() {
          return descuentoFis;
     }
          . . .
}
```

```
public class ClienteJuridico extends Cliente{
          . . .
     static double descuentoJur = 0;
     @Override
     protected double getDescuento() {
          return descuentoJur;
     }
          . . .
}
```

1) Una vez movido el cálculo del monto de las llamadas totales de un cliente a su propia clase Cliente, ya que es responsabilidad del propio cliente, aún queda dentro del for el mal olor de **Switch statement** para las llamadas las cuales utilizan un string para diferenciar el tipo, también el cálculo de las llamadas es prácticamente idéntico lo único que cambia son los parámetros del cálculo.

2) Extracto de código original que pertenece a la clase

```
public abstract class Cliente {
            . . .
      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;
                  }
                  auxc -= auxc*this.getDescuento();
                  c += auxc;
            }
            return c;
      }
            . . .
}
```

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

```
public class Llamada {
      private String tipoDeLlamada;
      private String origen;
      private String destino;
      private int duracion;
      public Llamada(String tipoLlamada, String origen, String destino, int duracion) {
            this.tipoDeLlamada = tipoLlamada;
            this.origen= origen;
            this.destino= destino;
            this.duracion = duracion;
      }
      public String getTipoDeLlamada() {
            return tipoDeLlamada;
      }
      public String getRemitente() {
            return destino;
      }
      public int getDuracion() {
            return this.duracion;
      }
      public String getOrigen() {
            return origen;
      }
}
```

```
class EmpresaTest{
            . . .
@Test
      void testcalcularMontoTotalLlamadas() {
            Cliente emisorPersonaFisca = new ClienteFisico("Brendan Eich", "11555666");
            sistema.registrarUsuario(emisorPersonaFisca);

            Cliente remitentePersonaFisica = new ClienteFisico("Doug Lea", "00000001");
            sistema.registrarUsuario(remitentePersonaFisica);

            Cliente emisorPersonaJuridica = new ClienteJuridico("Nvidia Corp", "17555222");
            sistema.registrarUsuario(emisorPersonaJuridica);

            Cliente remitentePersonaJuridica = new ClienteJuridico("Sun Microsystems", "25765432");
            sistema.registrarUsuario(remitentePersonaJuridica);

            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));
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Replace Conditional with Polymorphism**, donde crearemos subclases de Llamada donde se colocara la forma de hacer el cálculo de cada una permitiendo en un futuro agregar mas tipos de llamada si se quisiera; por lo que también sería conveniente aplicar **Form template method** para el cálculo del monto de la llamada. Al realizar este cambio la clase Empresa no podrá crear instancias de cada clase pasando a recibirlas por parámetros ya creados para aprovechar el polimorfismo, por lo tanto **también se deben modificar como se inicializan los test**.
4) Extracto de código con el refactoring aplicado:

```
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 double calcularMontoLlamada() {
            return this.getPrecioDuracion() + this.getPrecioDuracion() * 0.21 + this.getCostoEstablecimientoDeLlamada();
      }
      protected abstract double getPrecioDuracion();
      protected abstract double getCostoEstablecimientoDeLlamada();
            . . .
}
```

```
public class Empresa {
            . . .
      public Llamada registrarLlamada(Cliente origen,Llamada llamada) {
            llamadas.add(llamada);
            origen.llamadas.add(llamada);
            return llamada;
      }
            . . .
}
```

```
public class LlamadaNacional extends Llamada{
```

```
	public LlamadaNacional(String origen, String destino, int duracion) {
		super(origen, destino, duracion);
	}
	@Override
	protected double getPrecioDuracion() {
		return this.getDuracion() * 3;
	}
	@Override
	protected double getCostoEstablecimientoDeLlamada() {
		return 0;
	}
}
```

```
public class LlamadaInternacional extends Llamada{
	public LlamadaInternacional(String origen, String destino, int duracion) {
		super(origen, destino, duracion);
	}
	@Override
	protected double getPrecioDuracion() {
		return this.getDuracion() * 150;
	}
	@Override
	protected double getCostoEstablecimientoDeLlamada() {
		return 50;
	}
}
```

```
class EmpresaTest {
		. . .
	@Test
	void testcalcularMontoTotalLlamadas() {
		Cliente emisorPersonaFisca = new ClienteFisico("Brendan Eich", "11555666");
		sistema.registrarUsuario(emisorPersonaFisca);

		Cliente remitentePersonaFisica = new ClienteFisico("Doug Lea", "00000001");
		sistema.registrarUsuario(remitentePersonaFisica);

		Cliente emisorPersonaJuridica = new ClienteJuridico("Nvidia Corp", "17555222");
		sistema.registrarUsuario(emisorPersonaJuridica);

		Cliente remitentePersonaJuridica = new ClienteJuridico("Sun Microsystems", "25765432");
		sistema.registrarUsuario(remitentePersonaJuridica);

		Llamada llamadaN = new LlamadaNacional(emisorPersonaJuridica.getNumeroTelefono(),
remitentePersonaFisica.getNumeroTelefono(), 10);
		this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaN);

		Llamada llamadaI = new
LlamadaInternacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaFisica.getNumeroTelefono(),8);
		this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaI);

		llamadaN = new
LlamadaNacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaJuridica.getNumeroTelefono(),5);
		this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaN);

		llamadaI = new
LlamadaInternacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaJuridica.getNumeroTelefono(), 7);
		this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaI);

		llamadaN = new
LlamadaNacional(emisorPersonaFisca.getNumeroTelefono(),remitentePersonaFisica.getNumeroTelefono(),15);
		this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

		llamadaI = new LlamadaInternacional(emisorPersonaFisca.getNumeroTelefono(),
remitentePersonaFisica.getNumeroTelefono(), 45);
		this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);

		llamadaN = new LlamadaNacional(emisorPersonaFisca.getNumeroTelefono(),
```

```
remitentePersonaJuridica.getNumeroTelefono(), 13);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaFisca.getNumeroTelefono(),
remitentePersonaJuridica.getNumeroTelefono(), 17);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);

            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));
      }
            . . .
}
```

1) El mal olor detectado es la **reinvención de la rueda** en el método calcularMontoTotalLlamadas en la clase Cliente, ya que se utiliza un forEach para recorrer la colección y java ya posee herramientas para procesarlas. También si bien no es un mal olor, en el cálculo del monto total de la clase Cliente, se puede aprovechar la expresión del cálculo del descuento para poder utilizarlo en otros métodos y/o pasarlo como parámetro, haciendo el código más legible y reutilizable.

2) Extracto de código original que pertenece a la clase

```
public class Cliente {
            . . .
      public double calcularMontoTotalLlamadas() {
            double c = 0;
            for (Llamada l : this.llamadas) {
                  double auxc = l.calcularMontoLlamada();
                  auxc -= auxc*this.getDescuento();
                  c += auxc;
            }
            return c;
      }
      protected abstract double getDescuento();
            . . .
}
```

```
public class ClienteJuridico extends Cliente {
            . . .
      @Override
      protected double getDescuento() {
            return descuentoJur;
      }
            . . .
}
```

```
public class ClienteFisico extends Cliente {
            . . .
      @Override
      protected double getDescuento() {
            return descuentoFis;
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Extract Method**, con él se podrá reemplazar la expresión del cálculo del descuento por la llamada al método calcularDescuento

4) Extracto de código con el refactoring aplicado:

```
public abstract class Cliente {
            . . .
      public double calcularMontoTotalLlamadas() {
            return this.llamadas.stream().mapToDouble((l)->
                  l.calcularMontoLlamada() - this.getDescuento(l)
            ).sum();
      }
      protected abstract double getDescuento(Llamada l);
            . . .
```

```
}
```

```
public class ClienteJurico extends Cliente {
            . . .
      @Override
      protected double getDescuento(Llamada l) {
            return l.calcularMontoLlamada() * descuentoJur;
      }
            . . .
}
```

```
public class ClienteFisico extends Cliente {
            . . .
      @Override
      protected double getDescuento(Llamada l) {
            return l.calcularMontoLlamada() * descuentoFis;
      }
            . . .
}
```

---

1) Se detectó como mal olor que **un objeto conoce el id de otro**, en la clase Llamada se utiliza como origen y destino el número de teléfono de cada cliente para identificarlos, en lugar de tener una relación de conocimientos con objetos de la clase Cliente

2) Extracto de código original que pertenece a la clase

```
public abstract class Llamada {
      private String origen;
      private String destino;
      private int duracion;
      // las subclases usan este constructor, también se ven así
      public Llamada(String origen, String destino, int duracion) {
            this.origen= origen;
            this.destino= destino;
            this.duracion = duracion;
      }
            . . .
}
```

```
class EmpresaTest {
            . . .
      @Test
      void testcalcularMontoTotalLlamadas() {
            Cliente emisorPersonaFisca = new ClienteFisico("Brendan Eich", "11555666");
            sistema.registrarUsuario(emisorPersonaFisca);

            Cliente remitentePersonaFisica = new ClienteFisico("Doug Lea", "00000001");
            sistema.registrarUsuario(remitentePersonaFisica);

            Cliente emisorPersonaJuridica = new ClienteJuridico("Nvidia Corp", "17555222");
            sistema.registrarUsuario(emisorPersonaJuridica);

            Cliente remitentePersonaJuridica = new ClienteJuridico("Sun Microsystems", "25765432");
            sistema.registrarUsuario(remitentePersonaJuridica);

            Llamada llamadaN = new LlamadaNacional(emisorPersonaJuridica.getNumeroTelefono(),
remitentePersonaFisica.getNumeroTelefono(), 10);
            this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaN);

            Llamada llamadaI = new
LlamadaInternacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaFisica.getNumeroTelefono(),8);
            this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaI);

            llamadaN = new
LlamadaNacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaJuridica.getNumeroTelefono(),5);
            this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaN);
```

```
            llamadaI = new
LlamadaInternacional(emisorPersonaJuridica.getNumeroTelefono(),remitentePersonaJuridica.getNumeroTelefono(), 7);
            this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaI);

            llamadaN = new
LlamadaNacional(emisorPersonaFisca.getNumeroTelefono(),remitentePersonaFisica.getNumeroTelefono(),15);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaFisca.getNumeroTelefono(),
remitentePersonaFisica.getNumeroTelefono(), 45);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);

            llamadaN = new LlamadaNacional(emisorPersonaFisca.getNumeroTelefono(),
remitentePersonaJuridica.getNumeroTelefono(), 13);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaFisca.getNumeroTelefono(),
remitentePersonaJuridica.getNumeroTelefono(), 17);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);

            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));
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Replace Data Value with Object** con el se cambia el que una llamada conozca los clientes por un identificador String a conocerlos por medio de la referencia al cliente el cual ya tiene su propia clase.

4) Extracto de código con el refactoring aplicado:

```
public abstract class Llamada {
      private Cliente origen;
      private Cliente destino;
      private int duracion;
      // las subclases usan este constructor, también se ven así
      public Llamada(Cliente origen, Cliente destino, int duracion) {
            this.origen= origen;
            this.destino= destino;
            this.duracion = duracion;
      }
}
```

```
class EmpresaTest {
            . . .
      @Test
      void testcalcularMontoTotalLlamadas() {
            Cliente emisorPersonaFisca = new ClienteFisico("Brendan Eich", "11555666");
            sistema.registrarUsuario(emisorPersonaFisca);

            Cliente remitentePersonaFisica = new ClienteFisico("Doug Lea", "00000001");
            sistema.registrarUsuario(remitentePersonaFisica);

            Cliente emisorPersonaJuridica = new ClienteJuridico("Nvidia Corp", "17555222");
            sistema.registrarUsuario(emisorPersonaJuridica);

            Cliente remitentePersonaJuridica = new ClienteJuridico("Sun Microsystems", "25765432");
            sistema.registrarUsuario(remitentePersonaJuridica);

            Llamada llamadaN = new LlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);
            this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaN);

            Llamada llamadaI = new LlamadaInternacional(emisorPersonaJuridica,remitentePersonaFisica,8);
            this.sistema.registrarLlamada(emisorPersonaJuridica,llamadaI);

            llamadaN = new LlamadaNacional(emisorPersonaJuridica,remitentePersonaJuridica,5);
```

```
            this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaJuridica,remitentePersonaJuridica, 7);
            this.sistema.registrarLlamada(emisorPersonaJuridica, llamadaI);

            llamadaN = new LlamadaNacional(emisorPersonaFisca,remitentePersonaFisica,15);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaFisca, remitentePersonaFisica, 45);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);

            llamadaN = new LlamadaNacional(emisorPersonaFisca, remitentePersonaJuridica, 13);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaN);

            llamadaI = new LlamadaInternacional(emisorPersonaFisca, remitentePersonaJuridica, 17);
            this.sistema.registrarLlamada(emisorPersonaFisca, llamadaI);


            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));
      }
            . . .
}
```

---

1) Se detectó como mal olor “**envidia de atributos**”, ya que en la clase Empresa en el método “registrarLlamada” modifica la colección de la variable “llamadas” para agregarle una llamada, lo cual es responsabilidad de la clase Cliente. También para agregar encapsulamiento se pondrá en private la colección de “llamadas”

2) Extracto de código original que pertenece a la clase

```
public class Empresa{
            . . .
      public Llamada registrarLlamada(Cliente origen,Llamada llamada) {
            llamadas.add(llamada);
            origen.llamadas.add(llamada);
            return llamada;
      }
            . . .
}
```

```
public abstract class Cliente {
            . . .
      public List<Llamada> llamadas = new ArrayList<Llamada>();
            . . .
}
```

3) El refactoring que soluciona el mal olor es **Move Method**, se moverá el fragmento de código, que agrega la llamada a la colección, a la clase Cliente.

4) Extracto de código con el refactoring aplicado:

```
public abstract class Cliente {
      private List<Llamada> llamadas = new ArrayList<Llamada>();
            . . .
      public void registrarLlamada(Llamada llamada) {
               this.llamadas.add(llamada);
      }
            . . .
}
```

```
public class Empresa{
            . . .
      public Llamada registrarLlamada(Cliente origen,Llamada llamada) {
            llamadas.add(llamada);
            origen.registrarLlamada(llamada);
```

```
            return llamada;
      }
            . . .
}
```

1) Se detectó como mal olor “Switch statements” en la clase GestorNumerosDisponibles en el método “obtenerNumeroLibre”, ya que se utiliza un String para diferenciar el tipo y así actuar en consecuencia.

2) Extracto de código original que pertenece a la clase

```
public class GestorNumerosDisponibles {
            . . .
      private String tipoGenerador = "ultimo";
      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;
      }
            . . .
}
```

```
class EmpresaTest {
            . . .
      @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());
      }
            . . .
}
```

3) El refactoring que soluciona el mal olor es Replace conditional with Strategy, se crearán estrategias para cada tipo de generador. Tambien se tendrá que modificar el test de obtener número libre.
4) Extracto de código con el refactoring aplicado:

```
public class GestorNumerosDisponibles {
            . . .
      private IEstrategiaGeneradora tipoGenerador = new ObtenerUltimo();
      public String obtenerNumeroLibre() {
            return tipoGenerador.obtenerNumeroLibre(lineas);
      }

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

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

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

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

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

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

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

---

1) Se detectó como mal olor de la refactorización anterior que las estrategias modifican la colección de GestorNumerosDisponibles.
2) Extracto de código original que pertenece a la clase

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

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

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

```
      }
}
```

3) El refactoring que soluciona el mal olor es **Encapsulate Collection**, donde proveemos métodos desde GestorNumerosDisponibles para quitar números de la colección y de esta forma las estrategias recibirán como contexto a la clase GestorNumerosDisponibles.
4) Extracto de código con el refactoring aplicado:

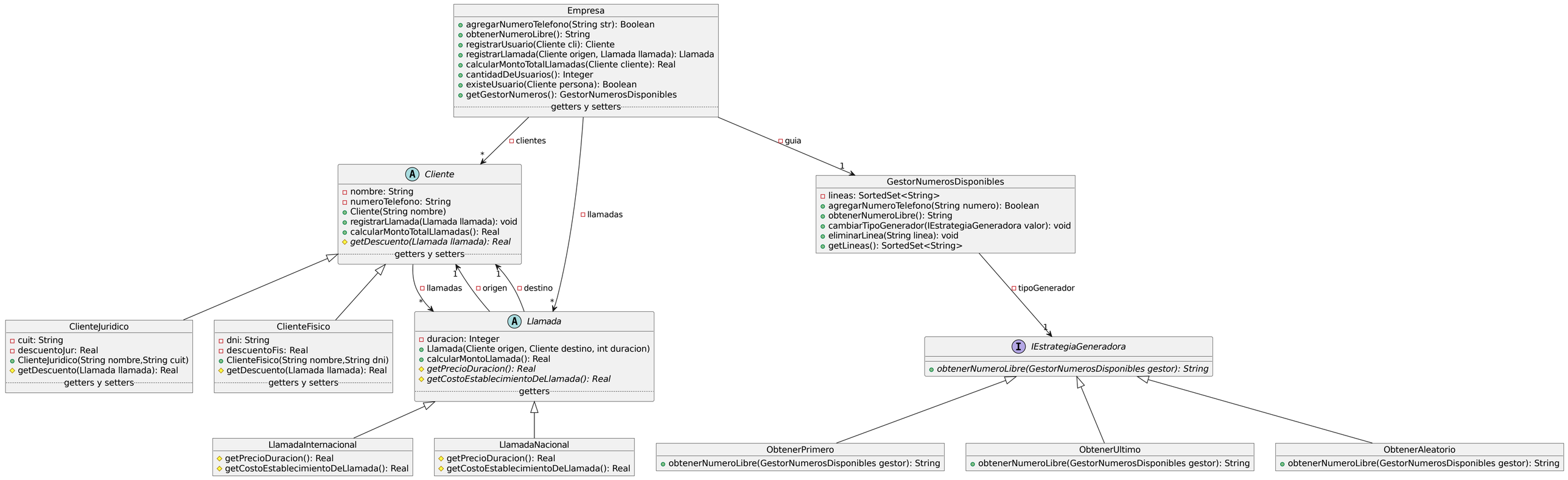
```
public class GestorNumerosDisponibles {
            . . .
      public String obtenerNumeroLibre() {
            return tipoGenerador.obtenerNumeroLibre(this);
      }
      public void eliminarLinea(String linea) {
            this.lineas.remove(linea);
      }
            . . .
}
```

```
public interface IEstrategiaGeneradora {
      public String obtenerNumeroLibre(GestorNumerosDisponibles gestor);
}
```

```
public class ObtenerPrimero implements IEstrategiaGeneradora {
      public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String linea = gestor.getLineas().first();
            gestor.eliminarLinea(linea);
            return linea;
      }
}
```

```
public class ObtenerUltimo implements IEstrategiaGeneradora{
      public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String linea = gestor.getLineas().last();
            gestor.eliminarLinea(linea);
            return linea;
      }
}
```

```
public class ObtenerAleatorio implements IEstrategiaGeneradora{
      public String obtenerNumeroLibre(GestorNumerosDisponibles gestor) {
            String linea = new ArrayList<String>(gestor.getLineas())
                        .get(new Random().nextInt(gestor.getLineas().size()));
            gestor.eliminarLinea(linea);
            return linea;
      }
}
```


Empresa
+ agregarNumeroTelefono(String str): Boolean
+ obtenerNumeroLibre(): String
+ registrarUsuario(Cliente cli): Cliente
+ registrarLlamada(Cliente origen, Llamada llamada): Llamada
+ calcularMontoTotalLlamadas(Cliente cliente): Real
+ cantidadDeUsuarios(): Integer
+ existeUsuario(Cliente persona): Boolean
+ getGestorNumeros(): GestorNumerosDisponibles
getters y setters
- clientes
*
- llamadas
*
- guia
1
Cliente
- nombre: String
- numeroTelefono: String
+ Cliente(String nombre)
+ registrarLlamada(Llamada llamada): void
+ calcularMontoTotalLlamadas(): Real
# getDescuento(Llamada llamada): Real
getters y setters
- llamadas
*
1
- origen
1
- destino
*
GestorNumerosDisponibles
- lineas: SortedSet<String>
+ agregarNumeroTelefono(String numero): Boolean
+ obtenerNumeroLibre(): String
+ cambiarTipoGenerador(IEstrategiaGeneradora valor): void
+ eliminarLinea(String linea): void
+ getLineas(): SortedSet<String>
- tipoGenerador
1
ClienteJuridico
- cuit: String
- descuentoJur: Real
+ ClienteJuridico(String nombre,String cuit)
# getDescuento(Llamada llamada): Real
getters y setters
ClienteFisico
- dni: String
- descuentoFis: Real
+ ClienteFisico(String nombre,String dni)
# getDescuento(Llamada llamada): Real
getters y setters
Llamada
- duracion: Integer
+ Llamada(Cliente origen, Cliente destino, int duracion)
+ calcularMontoLlamada(): Real
# getPrecioDuracion(): Real
# getCostoEstablecimientoDeLlamada(): Real
getters
IEstrategiaGeneradora
+ obtenerNumeroLibre(GestorNumerosDisponibles gestor): String
LlamadaInternacional
getPrecioDuracion(): Real
getCostoEstablecimientoDeLlamada(): Real
LlamadaNacional
getPrecioDuracion(): Real
getCostoEstablecimientoDeLlamada(): Real
ObtenerPrimero
+ obtenerNumeroLibre(GestorNumerosDisponibles gestor): String
ObtenerUltimo
+ obtenerNumeroLibre(GestorNumerosDisponibles gestor): String
ObtenerAleatorio
+ obtenerNumeroLibre(GestorNumerosDisponibles gestor): String