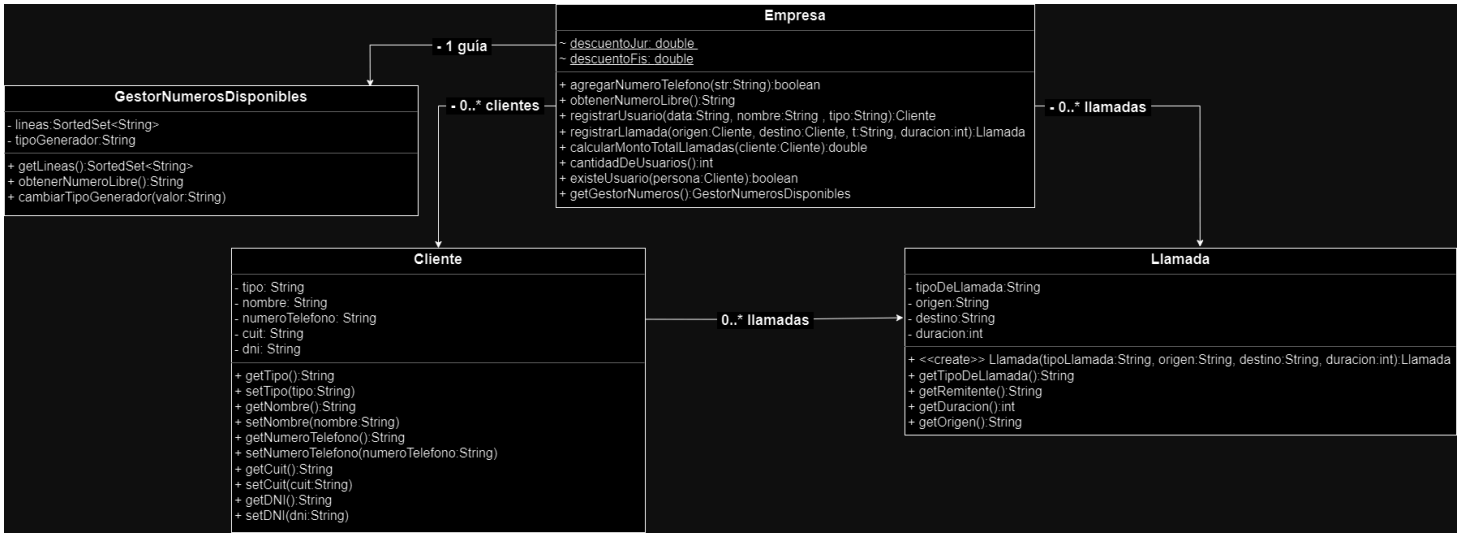
👷 TP Refactoring

UML Inicial

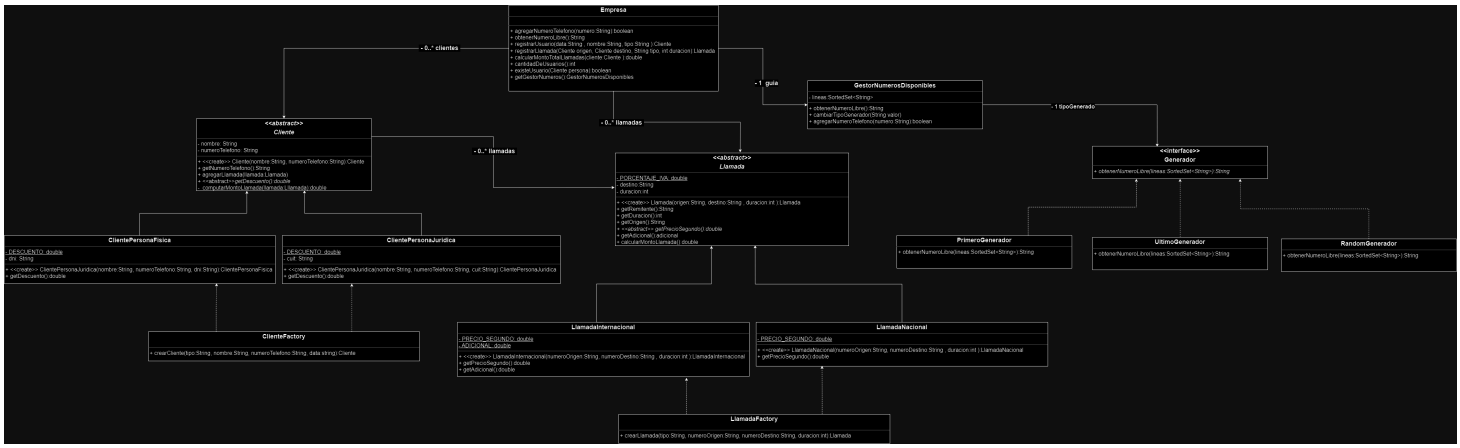


- 📌 Refactor 1
  - Move Method
  - Encapsulate Collection y Remove Method
- 📌 Refactor 2
  - Replace Conditional with Polymorphism - Remove method - Remove Field - Encapsulate Field
  - Factory Method
- 📌 Refactor 3
  - Encapsulate Collection - Move Method - Hide Delegate
- 📌 Refactor 4
  - Move Method - Rename Method
  - Replace Conditional with Polymorphism - Remove Field
  - Factory Method
  - Reinventando la rueda - Move field - Rename variables
- 📌 Refactor 5
  - Replace Conditional Logic with Strategy - Replace Magic Strings with Class Type

Codigo Final

- Cliente
- ClienteFactory
- ClientePersonaFisica
- ClientePersonaJuridica
- Generador
- GestorNumerosDisponibles
- Llamada
- LlamadaFactory
- LlamadaInternacional
- Llamada Nacional
- PrimeroGenerador
- RandomGenerador
- UltimoGenerador
- Test

UML Final



🚧 Refactor 1

🦠 Malos Olores

Feature Envy Se observa una mala asignación de responsabilidades en la clase `Empresa`, asociada a una evidente envidia de atributos.

Encontramos tareas en esta clase que deberían ser responsabilidad de `GestorNumeroDisponibles()` .

Extracto del código original:

```
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;
                }
        }
}
```

🚧 Refactor 1 Move Method

En primer lugar, realizamos un **Move method** del método `agregarNumeroTelefono(String str)` a la clase `GestorNumeroDisponibles()` .

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

	public SortedSet<String> getLineas() {
		return lineas;
	}
	public boolean agregarNumeroTelefono(String numero) {
		boolean encontre = this.getLineas().contains(numero);
		if (!encontre) {
			this.getLineas().add(numero);
			encontre = true;
			return encontre;
		} else {
			encontre = false;
			return encontre;
		}
	}
}
public class Empresa{
	public String agregarNumeroTelefono(String numero) {
		return guia.agregarNumeroTelefono(numero);
	}
}
```

 *Malos Olores*

Inappropriate Intimacy: El método `getLineas()` expone directamente el conjunto interno `lineas`, lo cual permite a otras partes del código modificar directamente esta colección. Esto puede llevar a problemas de manejo de estado y viola el principio de encapsulamiento.

Feature Envy: Los métodos en `GestorNumerosDisponibles` hacen un uso excesivo del getter getLineas() en lugar de interactuar directamente con el campo `lineas`.

Duplicate Code: El uso de una variable booleana `encontre` para manejar el control de flujo es innecesariamente complicado y duplica la lógica de verificación de existencia y adición en el conjunto.

 Refactor 1 Encapsulate Collection - Remove Method

Refactoring: Modificar getLineas() para que no retorne directamente la colección mutable. Mejor aún, eliminar este método si no es necesario y manejar toda la lógica de adición o eliminación a través de métodos específicos en GestorNumerosDisponibles.

Remove Method Eliminamos getLineas() para que no pueda ser modificado por fuera de la clase, aseguramos que toda manipulación de lineas se haga a través de métodos de la clase misma.

Refactor aplicado:

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

        public boolean agregarNumeroTelefono(String numero) {
                return this.lineas.add(numero);
        }
}

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

🚧 Refactor 2

🤏 *Malos Olores Detectados*

Duplicate Code: El código original tenía estructuras condicionales repetitivas para configurar objetos de `Cliente`, variando solo en la asignación de `dni` para clientes físicos y `cuit` para clientes jurídicos. Esto no solo duplica el código sino que también complica las modificaciones futuras.

Switch Statements: Aunque en tu refactoring final aún se utiliza un switch, este es movido a una fábrica, lo cual es un lugar más apropiado que dispersarlo por el código de negocio.

Large Class: La clase `Cliente` en el código original podría expandirse desproporcionadamente si se agregaran más tipos de clientes, con más condiciones y más campos.

Inappropriate Intimacy: La clase `Empresa` accede directamente a la lista de llamadas de `Cliente`. Esto viola el principio de encapsulación y crea una dependencia innecesaria entre las clases.

Dead Code (Código Muerto): El campo `tipo` de la clase `Cliente` no se utiliza en el código original, ya que el tipo de cliente se puede inferir a partir de la clase concreta que se esté utilizando. Además existen getters y setters que no se utilizan.

Speculative Generality: La clase `Cliente` tiene metodos que no se utiliza.

> Incluimos constructores en nuestras clases para garantizar que cada objeto se inicialice con todos los datos necesarios. Esto evita errores y hace el código más claro y fácil de entender. Es fundamental para asegurar que todos los objetos comiencen en un estado válido y seguro.

extracto del código original:

```
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;
	}
}

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

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

```
        }
        public void setNumeroTelefono(String numeroTelefono) {
                this.numeroTelefono = numeroTelefono;
        }
        public String getCuit() {
                return cuit;
        }
        public void setCuit(String cuit) {
                this.cuit = cuit;
        }
        public String getDNI() {
                return dni;
        }
        public void setDNI(String dni) {
                this.dni = dni;
        }
}
```

🚧 Refactor 2 Replace Conditional with Polymorphism - Remove method - Remove Field - Encapsulate Field

Creamos dos subclases, ClientePersonaFisica y ClientePersonaJuridica, desde la clase original Cliente y la hacemos abstracta. Esto no solo eliminó la duplicación de código sino que también aseguró que cada subclase maneje sus propios datos específicos de manera encapsulada.

- Agregamos un constructor a la clase Cliente para asegurarnos de que cada objeto se inicialice con todos los datos necesarios.
- Eliminamos el campo `tipo` de la clase Cliente, ya que este puede ser inferido a partir del tipo de subclase que se esté utilizando.
- Eliminamos los campos `dni` y `cuit` y sus respectivos getters y setters de la clase `Cliente`, ya que se movieron a las subclases `ClientePersonaFisica` y `ClientePersonaJuridica`, respectivamente.
- Cambiamos la visibilidad de la colección de `llamadas` de `Cliente` a privada para evitar que otras clases la manipulen directamente.
- Eliminamos el parametro `tipo` del constructor de Cliente, ya que este se puede inferir a partir del tipo de subclase que se esté utilizando.
- Eliminamos metodos que no se utilizan en las subclases de Cliente (getNombre(), getCuit(), getDNI(), setDNI(), setCuit()).

Refactor aplicado:

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

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

        public String getNumeroTelefono() {
                return numeroTelefono;
        }
}

public class ClientePersonaJuridica extends Cliente{
        private String cuit;
        public ClientePersonaJuridica(String nombre, String numeroTelefono, String cuit){
                super(nombre, numeroTelefono);
                this.cuit = cuit;
        }
}

public class ClientePersonaFisica extends Cliente{
        private String dni;
        public ClientePersonaFisica(String nombre, String numeroTelefono, String dni){
                super(nombre, numeroTelefono);
                this.dni = dni;
        }
}
```

🚧 Refactor 2 Factory Method

- Creamos una clase `ClienteFactory` con un método `crearCliente` que se encarga de instanciar el tipo correcto de cliente según el tipo especificado. Esto centraliza la creación de objetos `Cliente` y facilita la extensión del sistema para incorporar nuevos tipos de clientes en el futuro.
- Quedando el método `registrarUsuario` de la clase `Empresa` más limpio y desacoplado de la lógica de creación de clientes.

Refactor aplicado:

```
public class ClienteFactory{
        public static Cliente crearCliente(String tipo, String nombre, String numeroTelefono, St
                switch(tipo){
                        case "fisica":
                                return new ClientePersonaFisica(nombre, numeroTelefono, data);
                        case "juridica":
                                return new ClientePersonaJuridica(nombre, numeroTelefono, data);
                        default:
                                throw new IllegalArgumentException("Tipo de cliente no válido");
                }
        }
}

public class Empresa{
        public Cliente registrarUsuario(String data, String nombre, String tipo) {
                Cliente cliente = ClienteFactory.crearCliente(tipo, nombre, this.obtenerNumeroL
                clientes.add(cliente);
                return cliente;
        }
}
```

🚧 Refactor 3

🦨 *Malos Olores Detectados*

- **Feature Envy:** El método muestra envidia de atributos ya que accede directamente a la lista de llamadas del Cliente origen para agregar una nueva llamada. Esto indica una alta dependencia de la estructura interna de otra clase, lo cual va en contra del principio de encapsulación.
- **Inappropriate Intimacy:** Manipular directamente la lista de llamadas del Cliente desde otra clase podría ser considerado una intimidad inapropiada, pues el método no solo conoce detalles internos de otra clase, sino que también los modifica directamente.

Extracto del código original:

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

🚧 Refactor 3 Encapsulate Collection - Move Method - Hide Delegate

Encapsulamos la gestión de la colección de llamadas dentro de la clase Cliente. Esto implica crear métodos en la clase Cliente para añadir llamadas, en lugar de modificar la lista directamente desde fuera.

Consideramos mover parte de de la funcionalidad de `registrarLlamada()` de la clase `Empresa` a la clase `Cliente` creando un método `agregarLlamada()` .

Rafactor aplicado

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

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

public class Empresa {
    private List<Llamada> llamadas = new ArrayList<>();

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

🚧 Refactor 4

🦨 *Malos Olores Detectados*

- **Feature Envy y Inappropriate Intimacy**: El método accede directamente a la lista interna de llamadas del `Cliente`.
- **String Comparison**: Está utilizando `==` para comparar strings, lo cual es inapropiado en Java para comparaciones de contenido de strings. Debería usar `.equals()`.
- **Duplicate Code**: Hay código duplicado en la manera de calcular el costo de las llamadas, especialmente en cómo se añade el IVA y se calculan los descuentos.
- **Magic Numbers**: El código contiene numeros mágicos como `3`, `150`, `0.21`, `50`, que hacen el código difícil de entender y mantener.

Extracto del código original:

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

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

🚧 Refactor 4 Move Method - Rename Method

- Movimos el metodo `calcularCostoTotalLlamadas()` de la clase `Empresa` a la clase `Cliente` para que sea responsabilidad de la clase `Cliente` calcular el costo total de sus llamadas.
- Edita el método `calcularCostoTotalLlamadas()` de la clase `Empresa` para que llame al método `calcularCostoTotalLlamadas()` de la clase `Cliente` .

Refactor aplicado:

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

public class Cliente{
        public double calcularMontoTotalLlamadas() {
                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
                                // 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 po
                                // llamada
                                auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) -
                        }

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

🚧 Refactor 4 Replace Conditional with Polymorphism - Remove Field

- Creamos una clase abstracta `Llamada` con métodos abstractos para calcular el costo de la llamada. Luego, creamos subclases `LlamadaNacional` y `LlamadaInternacional` que implementan estos métodos de acuerdo a las reglas específicas de cada tipo de llamada.
- Eliminamos el campo `tipoDeLlamada` y su getter de la clase `Llamada` ya que este se puede inferir a partir del tipo de subclase que se esté utilizando, tambien eliminamos el parametro tipoDeLlamada del constructor de Llamada.

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

        public Llamada(String origen, String destino, int duracion) { // quitamos el tipo de lla
                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;
        }

}
public class LlamadaNacional extends Llamada{
    public LlamadaNacional(String numeroOrigen, String numeroDestino, int duracion) {
        super(numeroOrigen, numeroDestino, duracion);
    }
}

public class LlamadaInternacional extends Llamada{
        public LlamadaInternacional(String numeroOrigen, String numeroDestino, int duracion) {
                super(numeroOrigen, numeroDestino, duracion);
        }
}
```

🚧 Refactor 4 Factory Method - Rename Variables

- Creamos una clase `LlamadaFactory` con un método `crearLlamada` que se encarga de instanciar el tipo correcto de llamada según el tipo especificado. Esto centraliza la creación de objetos

`Llamada` y facilita la extensión del sistema para incorporar nuevos tipos de llamadas en el futuro.

- Modificamos el método `registrarLlamada()` de la clase `Empresa` para que utilice la fábrica de llamadas para crear el objeto de llamada correspondiente.
- Renombramos las variables poco descriptivas

```
public class LlamadaFactory {
    public static Llamada crearLlamada(String tipo, String numeroOrigen, String numeroDestino, :
        switch (tipo) {
            case "nacional":
                return new LlamadaNacional(numeroOrigen, numeroDestino, duracion);
            case "internacional":
                return new LlamadaInternacional(numeroOrigen, numeroDestino, duracion);
            default:
                throw new IllegalArgumentException("Tipo de llamada no válido");
        }
    }
}
public class Empresa {
        public Llamada registrarLlamada(Cliente origen, Cliente destino, String tipo, int duraci
        Llamada llamada = LlamadaFactory.crearLlamada(t, origen.getNumeroTelefono(), destino.get
        this.llamadas.add(llamada);
        origen.agregarLlamada(llamada);
        return llamada;
    }
}
```

🚧 Refactor 4 Reinventando la rueda - Move field - Rename variables

- Eliminamos el for loop de la clase Cliente y lo reemplazamos por un stream y una operación de suma, lo cual simplifica el cálculo del monto total de las llamadas.
- Creamos un método `computarMontoLlamada()` en la clase `Cliente` para calcular el monto de una llamada específica, teniendo en cuenta el descuento correspondiente.
- creamos un método `getDescuento()` en la clase `Cliente` para obtener el descuento correspondiente a cada cliente, cada subclase de Cliente implementará este método de acuerdo a sus reglas específicas.
- Movimos los atributos estaticos `descuentoFis` y `descuentoJur` a las subclases de `Cliente` (descuentoFis a ClientePersonaFisica y descuentoJur a ClientePersonaJuridica), Les cambiamos el nombre y el modificador de acceso a private y les creamos un getter.

- Creamos un método `calcularMontoLlamada()` en la clase Llamada para calcular el monto de la llamada, teniendo en cuenta el IVA y cualquier adicional.
- Creamos un método `getPrecioSegundo()` en la clase Llamada para obtener el precio por segundo de la llamada, que será implementado por las subclases de Llamada.
- Creamos un método `getAdicional()` en la clase Llamada para obtener cualquier adicional que se deba sumar al costo de la llamada, que será implementado por las subclases de Llamada.
- Creamos una constante `PORCENTAJE_IVA` en la clase Llamada para representar el porcentaje de IVA que se debe sumar al costo de la llamada.

```
public abstract class Cliente {
        public abstract double getDescuento();

        private double computarMontoLlamada(Llamada llamada) {
        return llamada.calcularMontoLlamada() * ( 1 - this.getDescuento());
    }
        public double calcularMontoTotalLlamadas() {
                return this.llamadas
                                .stream()
                                .mapToDouble(llamada -> this.computarMontoLlamada(llamada))
                                .sum();
        }
}

public class ClientePersonaFisica extends Cliente{
        private static final double DESCUENTO = 0;
        private String dni;

        public ClientePersonaFisica(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
    public double getDescuento() {
        return ClientePersonaFisica.DESCUENTO;
    }
}

public class ClientePersonaJuridica extends Cliente{
        private static final double DESCUENTO = 0.15;
        private String cuit;

        public ClientePersonaJuridica(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;
        }

        public double getDescuento() {
                return ClientePersonaJuridica.DESCUENTO;
        }
}

public abstract class Llamada{
        private static final double PORCENTAJE_IVA = 0.21;
        public abstract double getPrecioSegundo();

        public double getAdicional(){
                return 0;
        };

        public double calcularMontoLlamada() {
        return ((this.duracion * this.getPrecioSegundo()) + ((this.duracion * this.getPrecioSegu
    }
}

public class LlamadaNacional extends Llamada{
        private static final double PRECIO_SEGUNDO = 3;

        @Override
        public double getPrecioSegundo() {
                return LlamadaNacional.PRECIO_SEGUNDO;
        }
}

public class LlamadaInternacional extends Llamada{
    private static final double PRECIO_SEGUNDO = 150;
    private static final double ADICIONAL = 50;
    public LlamadaInternacional(String numeroOrigen, String numeroDestino, int duracion) {
        super(numeroOrigen, numeroDestino, duracion);
    }

    @Override
```

```
    public double getPrecioSegundo() {
        return PRECIO_SEGUNDO;
    }

    @Override
    public double getAdicional() {
        return ADICIONAL;
    }
}

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

🚧 Refactor 5

Malos Olores

- **Switch Statements:** El uso de switch para manejar diferentes formas de obtener un número (último, primero, aleatorio) puede complicar la extensión del código. Si decides agregar más formas de seleccionar un número, este switch crecerá en complejidad y tamaño.
- **Magic Strings**: Las cadenas "ultimo", "primero", y "random" utilizadas en el `switch` son ejemplos de "magic strings". Estos valores están codificados directamente en la lógica, haciendo que el código sea más difícil de mantener y propenso a errores si se escribe incorrectamente en algún lugar.
- **Duplicated Code:** Hay una lógica duplicada en cada caso del switch, particularmente en el proceso de remover un número de lineas después de seleccionarlo.

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

        public boolean agregarNumeroTelefono(String numero) {
                return this.lineas.add(numero);
        }
}
```

🚧 Refactor 5 Replace Conditional Logic with Strategy - Replace Magic Strings with Class Type

- Basandonos en el patrón Strategy, creamos la interfaz `Generador` que varian las implementaciones que encapsulan los diferentes comportamientos (último, primero, aleatorio).
- Creamos tres implementaciones de la interfaz `Generador` : `UltimoGenerador` , `PrimeroGenerador` , y `RandomGenerador` . Cada una de estas clases implementa el método `obtenerNumeroLibre()` de

acuerdo a su comportamiento específico.

- En lugar de usar cadenas para determinar el comportamiento, usamos los tipos de las clases directamente. Esto elimina la necesidad de cadenas mágicas y reduce la posibilidad de errores en tiempo de ejecución debido a cadenas mal escritas.
- En los test hicimos una minima modificacion, en vez de pasar un string, pasamos una instancia de la clase que implementa el comportamiento deseado.

```
public interface Generador {
    public  String  obtenerNumeroLibre(SortedSet<String> lineas);
}

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

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

public class RandomGenerador implements Generador {
    public String obtenerNumeroLibre(SortedSet<String> lineas) {
        List<String> lista = new ArrayList<>(lineas);
        String linea = lista.get(new Random().nextInt(lista.size()));
        lineas.remove(linea);
        return linea;
    }
}

public class GestorNumerosDisponibles {
        private SortedSet<String> lineas = new TreeSet<String>();
        private Generador tipoGenerador = new UltimoGenerador();

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

        public void cambiarTipoGenerador(Genedor generador) {
                this.tipoGenerador = generador;
        }
}
class EmpresaTest {
        @Test
```

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

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

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