Orientación a Objetos 2

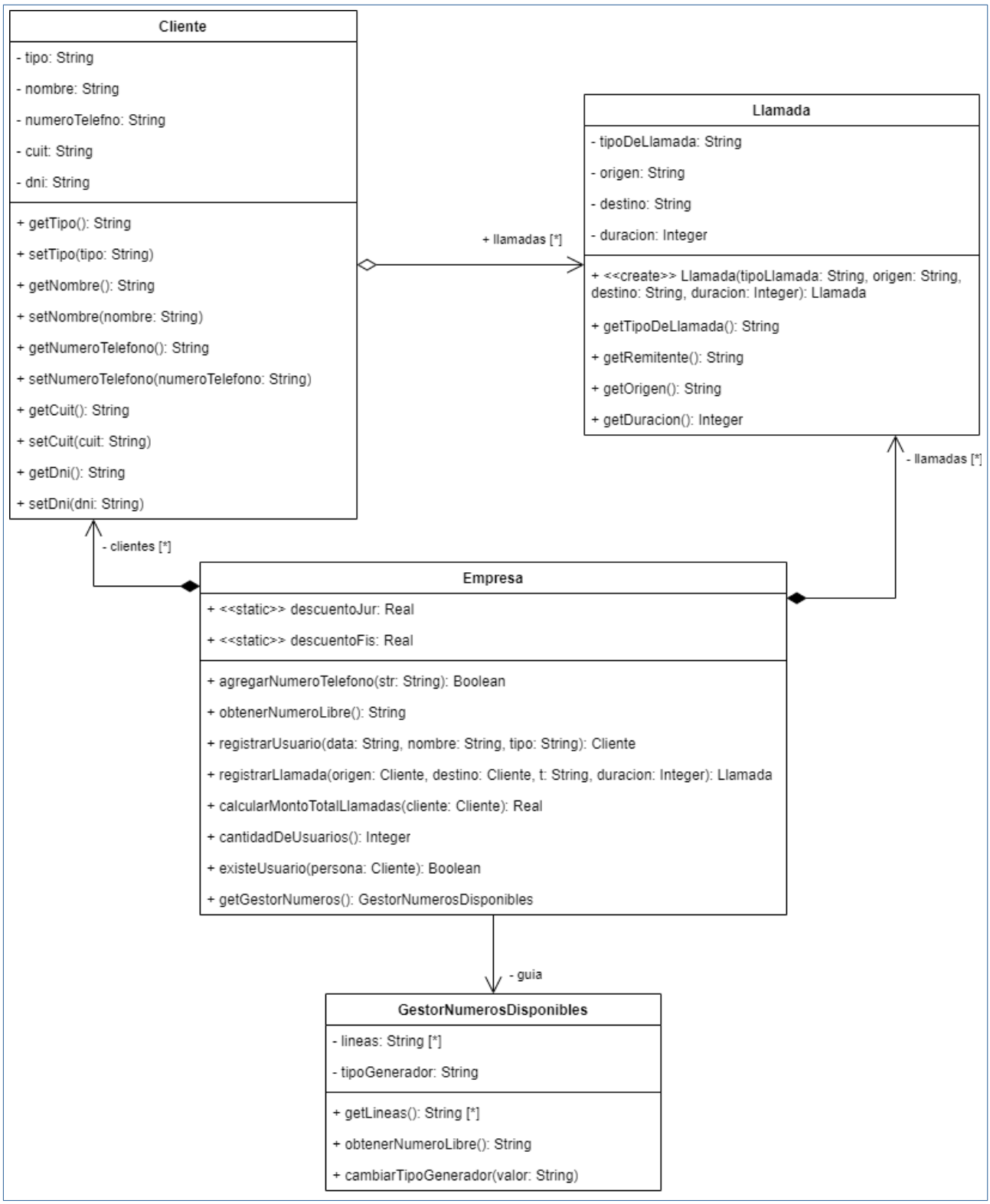
Cuadernillo Semestral de Actividades – Refactoring

Entrega de Ejercicio 3 – Facturación de llamadas

- **Alumno: Martín Tau – 7881/8**

- **Fecha: 22/05/2024**

Diseño inicial (disponible en "diseño_inicial.png")



Secuencia de refactorings aplicados

Refactoring 1

Mal olor detectado

Envidia de atributos (*feature envy*), de clase Empresa hacia GestorNumerosDisponibles. Además, nombre de parámetro poco descriptivo en el método.

Extracto del código que presenta el mal olor

Método agregarNumeroTelefono de la clase Empresa. Todo el método.

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

Refactoring a aplicar

Mover método (*move method*), renombrar parámetro.

Código resultante

Nuevo método agregarNumeroTelefono en la clase GestorNumerosDisponibles:

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

Método agregarNumeroTelefono en la clase Empresa:

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

Refactoring 2

Mal olor detectado

Reinventando la rueda en el método agregarNumeroTelefono de la clase GestorNumerosDisponibles.

Extracto del código que presenta el mal olor

Método agregarNumeroTelefono de la clase GestorNumerosDisponibles. Todo el cuerpo del método.

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

Refactoring a aplicar

Reutilizar el protocolo de la clase SortedSet a la cual pertenece la variable manipulada ("lineas").

Código resultante

```
	public boolean agregarNumeroTelefono(String lineaTelefonica) {
		return this.getLineas().add(lineaTelefonica);
	}
```

Refactoring 3

Mal olor detectado

Romper encapsulamiento en la clase GestorNumerosDisponibles.

Extracto del código que presenta el mal olor

Método getLineas de la clase. Expone innecesariamente la estructura interna.

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

	public boolean agregarNumeroTelefono(String lineaTelefonica) {
		return this.getLineas().add(lineaTelefonica);
	}
```

Refactoring a aplicar

Quitar el método getter. Actualizar llamadas al método en la propia clase (no existen usos por fuera de la clase).

Código resultante

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

Refactoring 4

Mal olor detectado

Sentencias Switch (*switch statements*) en el método obtenerNumeroLibre de la clase GestorNumerosDisponibles.

Extracto del código que presenta el mal olor

Todo el cuerpo del método.

```
      public String obtenerNumeroLibre() {
            String linea;
            switch (tipoGenerador) {
            case "ultimo":
                  linea = lineas.last();
                  lineas.remove(linea);
                  return linea;
            case "primero":
                  linea = lineas.first();
                  lineas.remove(linea);
                  return linea;
            case "random":
                  linea = new ArrayList<String>(lineas).get(new
Random().nextInt(lineas.size()));
                  lineas.remove(linea);
                  return linea;
            }
            return null;
      }
```

Refactoring a aplicar

Reemplazar condicional con polimorfismo (*Replace Conditional with Polymorphism*).

El primer paso de este método sería crear una estructura de subclases, esto no se puede hacer con la clase principal (GestorNumerosDisponibles) puesto que el "tipo" de la clase (variable tipoGenerador) es modificado dinámicamente, de modo que resulta más apropiado aplicar el patrón Strategy (*Replace Type Code with State/Strategy*).

Luego de mover los sub-bloques del switch a la estructura creada, el siguiente paso sería agregar en GestorNumerosDisponibles una variable que guarde la estrategia actual del objeto y modificar la lógica de sus métodos para que delegue responsabilidad en la estrategia. En este punto también se quita la variable "tipoGenerador", que ya no tiene sentido.

En la lógica original existen tres casos base: primero, último y aleatorio; más, un caso adicional para todo lo demás, que es NULL. Este último caso quizás más adelante debería ser reemplazado por un Null Object; en este punto queda aún explícito el NULL para no complicar el paso.

Un último comentario, con respecto al método cambiarTipoGenerador; a fin de no modificar los clientes de esta clase (por ejemplo los tests del proyecto), se mueve el switch a este método.

Código resultante

Estructura de herencia:

```
// <<Strategy>>
public interface GeneradorNumerosStrategy {
	public String liberarLinea(SortedSet<String> lineas);
}

// <<ConcreteStrategies>>
public class GeneradorNumerosStrategyUltimo implements GeneradorNumerosStrategy {
	public String liberarLinea(SortedSet<String> lineas) {
		String linea = lineas.last();
		lineas.remove(linea);
		return linea;
	}
}
public class GeneradorNumerosStrategyPrimero implements GeneradorNumerosStrategy {
	public String liberarLinea(SortedSet<String> lineas) {
		String linea = lineas.first();
		lineas.remove(linea);
		return linea;
	}
}
public class GeneradorNumerosStrategyRandom implements GeneradorNumerosStrategy {
	public String liberarLinea(SortedSet<String> lineas) {
		String linea = new ArrayList<String>(lineas).get(new
Random().nextInt(lineas.size()));
		lineas.remove(linea);
		return linea;
	}
}
```

Clase GestorNumerosDisponibles, nueva lógica para obtener números:

```
public class GestorNumerosDisponibles {
	private GeneradorNumerosStrategy generador = new
GeneradorNumerosStrategyUltimo();
	public String obtenerNumeroLibre() {
		if (this.generador != null) {
			return this.generador.liberarLinea(this.lineas);
		}
		return null;
	}
	public void cambiarTipoGenerador(String valor) {
		switch (valor) {
		case "ultimo":
			this.generador = new GeneradorNumerosStrategyUltimo();
			break;
		case "primero":
			this.generador = new GeneradorNumerosStrategyPrimero();
			break;
		case "random":
			this.generador = new GeneradorNumerosStrategyRandom();
			break;
		default:
			this.generador = null;
		}
	}
	// ...
}
```

Refactoring 5

Mal olor detectado

Chequeo por Null, y también return de null en caso de que no haya un generador asignado.

Extracto del código que presenta el mal olor

Métodos obtenerNumeroLibre y cambiarTipoGenerador de la clase GestorNumerosDisponibles.

```
	public String obtenerNumeroLibre() {
		if (this.generador != null) {
			return this.generador.liberarLinea(this.lineas);
		}
		return null;
	}
	public void cambiarTipoGenerador(String valor) {
		switch (valor) {
		case "ultimo":
			this.generador = new GeneradorNumerosStrategyUltimo();
			break;
		case "primero":
			this.generador = new GeneradorNumerosStrategyPrimero();
			break;
		case "random":
			this.generador = new GeneradorNumerosStrategyRandom();
			break;
		default:
			this.generador = null;
		}
	}
```

Refactoring a aplicar

Reemplazar por un Null Object (*Introduce Null Object*).

Código resultante

```
// <<ConcreteStrategy>>
public class NullGeneradorNumerosStrategy implements GeneradorNumerosStrategy {
	public String obtenerNumeroLibre(SortedSet<String> lineas) {
		return null;
	}
}

public class GestorNumerosDisponibles {
	public String obtenerNumeroLibre() {
		return this.generador.liberarLinea(this.lineas);
	}
	public void cambiarTipoGenerador(String valor) {
		switch (valor) {
		case "ultimo":
			this.generador = new GeneradorNumerosStrategyUltimo();
			break;
		case "primero":
			this.generador = new GeneradorNumerosStrategyPrimero();
			break;
		case "random":
			this.generador = new GeneradorNumerosStrategyRandom();
			break;
		default:
			this.generador = new NullGeneradorNumerosStrategy();
		}
	}
	// ...
}
```

Refactoring 6

Mal olor detectado

Sentencias Switch, en el método modificado en pasos anteriores, cambiarTipoGenerador.

Extracto del código que presenta el mal olor

Todo el cuerpo del método.

```
	public void cambiarTipoGenerador(String valor) {
		switch (valor) {
		case "ultimo":
			this.generador = new GeneradorNumerosStrategyUltimo();
			break;
		case "primero":
			this.generador = new GeneradorNumerosStrategyPrimero();
			break;
		case "random":
			this.generador = new GeneradorNumerosStrategyRandom();
			break;
		default:
			this.generador = new NullGeneradorNumerosStrategy();
		}
	}
```

Refactoring a aplicar

En este caso, va a depender del contexto.

El escenario más sencillo (**escenario 1**) sería quitarlo, esto sólo sería posible si los clientes que utilicen este método puedan también modificarse. Hay varias opciones para esto esto (por ejemplo *Replace Parameter with Explicit Methods*); lo mejor en este caso sería cambiar el parámetro "valor" para pasar a recibir directamente la estrategia.

En caso que el escenario anterior no sea posible (por ejemplo, si este método es llamado por muchos clientes y/o que no se pueden modificar), entonces habría que dejar el switch (**escenario 2**) en algún lado, idealmente moviendo el método de lugar y delegando la responsabilidad desde la clase GestorNumerosDisponibles hacia donde quede el switch.

Código resultante

Se crearon dos métodos para representar los escenarios 1 y 2 comentados en el punto anterior.

Para el caso del escenario 2, se crea el método estático en GeneradorNumerosStrategy, para lo cual se debe modificar la interfaz a clase abstracta, y modificar todas las subclases para que extiendan en lugar de implementar (estos cambios se omiten en este documento).

```
public class GestorNumerosDisponibles {
	/** Escenario 1. */
	public void setGenerador(GeneradorNumerosStrategy generador) {
		this.generador = generador;
	}
	/** Escenario 2. */
	public void cambiarTipoGenerador(String tipo) {
		this.generador = GeneradorNumerosStrategy.crearGenerador(tipo);
	}
	// ...
}

public abstract class GeneradorNumerosStrategy {
	public static GeneradorNumerosStrategy crearGenerador(String tipo) {
		switch (tipo) {
		case "ultimo":
			return new GeneradorNumerosStrategyUltimo();
		case "primero":
			return new GeneradorNumerosStrategyPrimero();
		case "random":
			return new GeneradorNumerosStrategyRandom();
		default:
			return new NullGeneradorNumerosStrategy();
		}
	}
	// ...
}
```

Refactoring 7

Mal olor detectado

Código duplicado, en las estrategias concretas generadas anteriormente.

Extracto del código que presenta el mal olor

Las implementaciones de los métodos abstractos liberarLinea ejecutan pasos similares y en parte duplicados.

```
	public String liberarLinea(SortedSet<String> lineas) {
		String linea = /* Parte que difiere ligeramente. */;.
		lineas.remove(linea);
		return linea;
	}
```

Refactoring a aplicar

Extraer la lógica a un Template Method (*Form Template Method*). Se aprovecha para cambiar la colección de líneas a no-mdificable, para separar mejor las responsabilidades y sumar cohesión.

Código resultante

```
public abstract class GeneradorNumerosStrategy {
	// ...
	protected abstract String seleccionarLineaLibre(SortedSet<String> lineas);

	public String liberarLinea(SortedSet<String> lineas) {
		String linea =
this.seleccionarLineaLibre(Collections.unmodifiableSortedSet(lineas));
		lineas.remove(linea);
		return linea;
	}
}
public class GeneradorNumerosStrategyPrimero extends GeneradorNumerosStrategy {
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return lineas.first();
	}
}
public class GeneradorNumerosStrategyUltimo extends GeneradorNumerosStrategy {
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return lineas.last();
	}
}
public class GeneradorNumerosStrategyRandom extends GeneradorNumerosStrategy {
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return new ArrayList<String>(lineas).get(new
Random().nextInt(lineas.size()));
	}
}
public class NullGeneradorNumerosStrategy extends GeneradorNumerosStrategy {
	public String liberarLinea(SortedSet<String> lineas) {
		return null;
	}
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return null;
	}
}
```

Refactoring 8

Mal olor detectado

Las clases GestorNumerosDisponibles y Empresa no tienen constructores pero realizan la inicialización de sus variables en la declaración de las variables.

Extracto del código que presenta el mal olor

```
public class GestorNumerosDisponibles {
	private SortedSet<String> lineas = new TreeSet<String>();
	private GeneradorNumerosStrategy generador = new
GeneradorNumerosStrategyUltimo();
	// ...
}
public class Empresa {
	private List<Cliente> clientes = new ArrayList<Cliente>();
	private List<Llamada> llamadas = new ArrayList<Llamada>();
	private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
	// ...
}
```

Refactoring a aplicar

Crear constructores y realizar la inicialización adentro.

Código resultante

```
public class GestorNumerosDisponibles {
	private SortedSet<String> lineas;
	private GeneradorNumerosStrategy generador;

	public GestorNumerosDisponibles() {
		this.lineas = new TreeSet<String>();
		this.generador = new GeneradorNumerosStrategyUltimo();
	}
	// ...
}
public class Empresa {
	private List<Cliente> clientes;
	private List<Llamada> llamadas;
	private GestorNumerosDisponibles guia;

	public Empresa() {
		this.clientes = new ArrayList<Cliente>();
		this.llamadas = new ArrayList<Llamada>();
		this.guia = new GestorNumerosDisponibles();
	}
	// ...
}
```

Refactoring 9

Mal olor detectado

Ejecutar diferente lógica en base al valor de un parámetro.

Extracto del código que presenta el mal olor

Método registrarUsuario de la clase Empresa.

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

Refactoring a aplicar

Reemplazar el parámetro por métodos excplicitos que realicen la lógica específica de cada caso (*Replace Parameter with Explicit Methods*). Se cambian también nombres de variables.

Código resultante

```
public Cliente registrarPersonaFisica(String dni, String nombre) {
	Cliente cliente = new Cliente();
	cliente.setNombre(nombre);
	String tel = this.obtenerNumeroLibre();
	cliente.setTipo("fisica");
	cliente.setNumeroTelefono(tel);
	cliente.setDNI(dni);
	clientes.add(cliente);
	return cliente;
}
public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
	Cliente cliente = new Cliente();
	String tel = this.obtenerNumeroLibre();
	cliente.setNombre(razonSocial);
	cliente.setTipo("juridica");
	cliente.setNumeroTelefono(tel);
	cliente.setCuit(cuit);
	clientes.add(cliente);
	return cliente;
}
```

También fue necesario actualizar los tests para utilizar los nuevos métodos.

Refactoring 10

Mal olor detectado

No ofrecer constructores, lo cual obliga a usar los setters en los métodos creados en el punto anterior.

Extracto del código que presenta el mal olor

Métos registrar personas de la clase Empresa.

```
	public Cliente registrarPersonaFisica(String dni, String nombre) {
		// ...
		cliente.setNombre(nombre);
		cliente.setTipo("fisica");
		cliente.setNumeroTelefono(tel);
		cliente.setDNI(dni);
		// ...
	}
	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		// Íem método registrarPersonaFisica.
	}
```

Refactoring a aplicar

Ofrecer constructor en la clase Cliente y modificar la creación de personas en la clase Empresa.

Código resultante

```
public class Cliente {
	public List<Llamada> llamadas;
	// ...
	public Cliente(String tipo, String nombre, String numeroTelefono, String cuit,
String dni) {
		this.tipo = tipo;
		this.nombre = nombre;
		this.numeroTelefono = numeroTelefono;
		this.cuit = cuit;
		this.dni = dni;
		this.llamadas = new ArrayList<Llamada>();
	}
}
public class Empresa {
	// ...
	public Cliente registrarPersonaFisica(String dni, String nombre) {
		Cliente cliente = new Cliente("fisica", nombre, this.obtenerNumeroLibre(),
null, dni);
		clientes.add(cliente);
		return cliente;
	}
	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		Cliente cliente = new Cliente("juridica", razonSocial,
this.obtenerNumeroLibre(), cuit, null);
		clientes.add(cliente);
		return cliente;
	}
}
```

Refactoring 11

Mal olor detectado

Se tiene un dato inmutable que acúa como "tipo" de la clase Cliente.

Extracto del código que presenta el mal olor

En la clase Cliente, existe la variable de instancia "tipo", la cual se utiliza dentro de Empresa para realizar cálculos.

```
public class Cliente {
	private String tipo;
	// ...
	public Cliente(String tipo, String nombre, String numeroTelefono, String cuit,
String dni) {
		this.tipo = tipo;
		// ...
	}
	public String getTipo() {
		return tipo;
	}
	public void setTipo(String tipo) {
		this.tipo = tipo;
	}
}
```

Refactoring a aplicar

Reemplazar el tipo por subclases (*Replace Type Code with Subclasses*). Como parte de este proceso, dado que el tipo se recibe en el constructor, primero se debe reemplazar el constructor por Factory Method (*Replace Constructor with Factory Method*); en este caso, dado que la cantidad de subclases es pequeña y que es dificil que cambie, utilizo la variante de crear métodos explícitos para cada tipo (*Creating Subclasses with Explicit Methods*). En las distintas subclases que quedan por cada valor posible del "tipo" se redefine el getter del tipo para retornar los valores *hardcoded*, de manera temporal hasta poder luego seguir refactoreando.

Se aprovecha este paso para separar los datos DNI y CUIT, ya que cada tipo de persona utiliza uno solo, se mueven también los getters, setters y las variables de instancia correspondientes (*Push Down Method* y *Push Down Field*).

Código resultante

```
public abstract class Cliente {
	public static Cliente crearPersonaFisica(String nombre, String numeroTelefono,
String dni) {
		return new ClientePersonaFisica(nombre, numeroTelefono, dni);
	}

	public static Cliente crearPersonaJuridica(String nombre, String numeroTelefono,
String cuit) {
		return new ClientePersonaJuridica(nombre, numeroTelefono, cuit);
	}

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

	public abstract String getTipo();

	// ...
}
```

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

	protected ClientePersonaFisica(String nombre, String numeroTelefono, String dni)
{
		super(nombre, numeroTelefono);
		this.dni = dni;
	}

	public String getTipo() {
		return "fisica";
	}

	public String getDNI() {
		return dni;
	}

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

public class ClientePersonaJuridica extends Cliente {
	private String cuit;

	protected ClientePersonaJuridica(String razonSocial, String numeroTelefono,
String cuit) {
		super(razonSocial, numeroTelefono);
		this.cuit = cuit;
	}

	public String getTipo() {
		return "juridica";
	}

	public String getCuit() {
		return cuit;
	}

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

public class Empresa {
	// ...
	public Cliente registrarPersonaFisica(String dni, String nombre) {
		Cliente cliente = Cliente.crearPersonaFisica(nombre,
this.obtenerNumeroLibre(), dni);
		// ...
	}
	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		Cliente cliente = Cliente.crearPersonaJuridica(razonSocial,
this.obtenerNumeroLibre(), cuit);
		// ...
	}
}
```

Refactoring 12

Mal olor detectado

Código duplicado en los métodos de registrar personas de la clase Empresa.

Extracto del código que presenta el mal olor

Los métodos creados anteriormente.

```
	public Cliente registrarPersonaFisica(String dni, String nombre) {
		Cliente cliente = Cliente.crearPersonaFisica(nombre,
this.obtenerNumeroLibre(), dni);
		clientes.add(cliente);
		return cliente;
	}
	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		Cliente cliente = Cliente.crearPersonaJuridica(razonSocial,
this.obtenerNumeroLibre(), cuit);
		clientes.add(cliente);
		return cliente;
	}
```

Refactoring a aplicar

Extraer la lógica duplicada a un método (*Extract Method*).

Código resultante

```
	private Cliente agregarCliente(Cliente cliente) {
		clientes.add(cliente);
		return cliente;
	}
	public Cliente registrarPersonaFisica(String dni, String nombre) {
		return this.agregarCliente(Cliente.crearPersonaFisica(nombre,
this.obtenerNumeroLibre(), dni));
	}
	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		return this.agregarCliente(Cliente.crearPersonaJuridica(razonSocial,
this.obtenerNumeroLibre(), cuit));
	}
```

Refactoring 13

Mal olor detectado

Varios casos, que pueden condensarse en mala asignación de responsabilidades; en la clase Empresa se utiliza el "tipo" de los clientes para calcular descuentos sobre los montos de las llamadas que realizan.

Extracto del código que presenta el mal olor

En la clase Empresa, se tienen variables estáticas para calcular los descuentos de los clientes, esto debe ser responsabilidad de los clientes.

Parte del método calcularMontoTotalLlamadas utiliza esas variables.

Finalmente, el uso del getTipe.

```
public class Empresa {
	static double descuentoJur = 0.15;
	static double descuentoFis = 0;
	// ...
	public double calcularMontoTotalLlamadas(Cliente cliente) {
		// ...
			if (cliente.getTipo() == "fisica") {
				auxc -= auxc * descuentoFis;
			} else if (cliente.getTipo() == "juridica") {
				auxc -= auxc * descuentoJur;
			}
		// ...
	}
```

Refactoring a aplicar

Mover la responsabilidad desde Empresa hacia Cliente (*Move* Field, *Move Method*, etc.). Se aprovecha en este paso para quitar el getter de "Tipo" de cliente, que ya no es necesario.

Código resultante

```
public abstract class Cliente {
	public abstract double getDescuentoEnLlamadas();
	// ...
}
public class ClientePersonaFisica extends Cliente {
	private static double DESCUENTO_LLAMADAS = 0;
	public double getDescuentoEnLlamadas() {
		return ClientePersonaFisica.DESCUENTO_LLAMADAS;
	}
	// ...
}
public class ClientePersonaJuridica extends Cliente {
	private static double DESCUENTO_LLAMADAS = 0.15;
	public double getDescuentoEnLlamadas() {
		return ClientePersonaJuridica.DESCUENTO_LLAMADAS;
	}
	// ...
}
public class Empresa {
	public double calcularMontoTotalLlamadas(Cliente cliente) {
		// ...
			auxc -= auxc * cliente.getDescuentoEnLlamadas();
		// ...
	}
}
```

Refactoring 14

Mal olor detectado

Los malos olores detectados para Cliente con respecto a su uso en la clase Empresa se pueden detectar también para Llamada. Se agrupan en este paso por ser similares y por brevedad del presente documento.

Extracto del código que presenta el mal olor

El método registrarLlamada de Empresa funciona muy similar a como funcionaba registrarUsuario

de la misma clase.

La clase Llamada tiene un "tipo" en forma de dato string. El método calcularMontoTotalLlamadas de Empresa ejecuta código diferente en base a dicho tipo.

```
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 double calcularMontoTotalLlamadas(Cliente cliente) {
		// ...
			if (l.getTipoDeLlamada() == "nacional") {
				auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
			} else if (l.getTipoDeLlamada() == "internacional") {
				auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 *
0.21) + 50;
			}
		// ...
	}
}
```

Refactoring a aplicar

Reemplazar parametros, reemplazar tipo por subclases, extraer método, mover responsabilidad.

Código resultante

```
public abstract class Llamada {
	public static Llamada crearLlamadaNacional(Cliente origen, Cliente destino, int
duracion) {
		return new LlamadaNacional(origen.getNumeroTelefono(),
destino.getNumeroTelefono(), duracion);
	}
	public static Llamada crearLlamadaInternacional(Cliente origen, Cliente destino,
int duracion) {
		return new LlamadaInternacional(origen.getNumeroTelefono(),
destino.getNumeroTelefono(), duracion);
	}
	protected Llamada(String origen, String destino, int duracion) {
		// ...
	}
	public abstract double getMonto();
	// ...
}

public class LlamadaNacional extends Llamada {
	public LlamadaNacional(String origen, String destino, int duracion) {
		super(origen, destino, duracion);
	}
	public double getMonto() {
		// el precio es de 3 pesos por segundo más IVA sin adicional por
establecer la llamada
		return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
	}
}
```

```
public class LlamadaInternacional extends Llamada {
	public LlamadaInternacional(String origen, String destino, int duracion) {
		super(origen, destino, duracion);
	}
	public double getMonto() {
		// el precio es de 150 pesos por segundo más IVA más 50 pesos por
establecer la llamada
		return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
	}
}

public abstract class Empresa {
	// ...
	private Llamada agregarLlamada(Cliente origen, Llamada llamada) {
		llamadas.add(llamada);
		origen.llamadas.add(llamada);
		return llamada;
	}
	public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int
duracion) {
		return this.agregarLlamada(origen, Llamada.crearLlamadaNacional(origen,
destino, duracion));
	}
	public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino,
int duracion) {
		return this.agregarLlamada(origen,
Llamada.crearLlamadaInternacional(origen, destino, duracion));
	}
	public double calcularMontoTotalLlamadas(Cliente cliente) {
		// ...
			auxc += llamada.getMonto();
		// ...
	}
}
```

Refactoring 15

Mal olor detectado

Envidia de atributos (*Feature Envy*) de Empresa a Cliente. Empresa realiza un recorrido de las llamadas de un cliente y calcula el monto; esto debería ser responsabilidad del Cliente.

Se aprovecha para corregir además: en Empesa se conocen todas las llamadas y no se utilizan; y en Cliente las llamadas son públicas.

Extracto del código que presenta el mal olor

El método calcularMontoTotalLlamadas de Empresa y la variable de instancia llamadas.

```
public abstract class Cliente {
	public List<Llamada> llamadas;
	// ...
}
```

```
public class Empresa {
	// ...
	private List<Llamada> llamadas;
	public Empresa() {
		this.llamadas = new ArrayList<Llamada>();
		// ...
	}
	private Llamada agregarLlamada(Cliente origen, Llamada llamada) {
		llamadas.add(llamada);
		origen.llamadas.add(llamada);
		return llamada;
	}
	public double calcularMontoTotalLlamadas(Cliente cliente) {
		double montoToatal = 0;
		for (Llamada llamada : cliente.llamadas) {
			double auxc = 0;
			auxc += llamada.getMonto();
			auxc -= auxc * cliente.getDescuentoEnLlamadas();
			montoToatal += auxc;
		}
		return montoToatal;
	}
}
```

Refactoring a aplicar

Mover métodos.

Código resultante

```
public abstract class Empresa {
	private Llamada agregarLlamada(Cliente origen, Llamada llamada) {
		return origen.agregarLlamada(llamada);
	}
	public double calcularMontoTotalLlamadas(Cliente cliente) {
		return cliente.calcularMontoTotalLlamadas();
	}
	// ...
}
public abstract class Cliente {
	private List<Llamada> llamadas;
	// ...
	public Llamada agregarLlamada(Llamada llamada) {
		this.llamadas.add(llamada);
		return llamada;
	}
	public double calcularMontoTotalLlamadas() {
		double montoToatal = 0;
		for (Llamada llamada : this.llamadas) {
			double auxc = 0;
			auxc += llamada.getMonto();
			auxc -= auxc * this.getDescuentoEnLlamadas();
			montoToatal += auxc;
		}
		return montoToatal;
	}
}
```

Refactoring 16

Mal olor detectado

Resulta superfluo en la clase Empresa el uso del método creado anteriormente, agregarLlamada.

Extracto del código que presenta el mal olor

Método agregarLlamada de la clase Empresa.

```
	private Llamada agregarLlamada(Cliente origen, Llamada llamada) {
		return origen.agregarLlamada(llamada);
	}
	public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int
duracion) {
		return this.agregarLlamada(origen, Llamada.crearLlamadaNacional(origen,
destino, duracion));
	}
	public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino,
int duracion) {
		return this.agregarLlamada(origen,
Llamada.crearLlamadaInternacional(origen, destino, duracion));
	}
```

Refactoring a aplicar

Quitar método.

Código resultante

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

Refactoring 17

Mal olor detectado

Reinventando la rueda.

Extracto del código que presenta el mal olor

Método calcularMontoTotalLlamadas de la clase Cliente.

```
	public double calcularMontoTotalLlamadas() {
		double montoToatal = 0;
		for (Llamada llamada : this.llamadas) {
			double auxc = 0;
			auxc += llamada.getMonto();
			auxc -= auxc * this.getDescuentoEnLlamadas();
			montoToatal += auxc;
		}
		return montoToatal;
	}
```

Refactoring a aplicar

Utilizar stream y sum reducer para calcular la sumatoria el protocolo de la clase SortedSet a la cual pertenece la variable manipulada ("lineas").

Código resultante

```
	public double calcularMontoTotalLlamadas() {
		return this.llamadas.stream().mapToDouble(llamada -> llamada.getMonto() *
(1 - this.getDescuentoEnLlamadas())).sum();
	}
```

Refactoring 18

Mal olor detectado

Métodos en subclases ejecutan pasos similares en idéntico orden.

Extracto del código que presenta el mal olor

Redefinición de método getMonto en subclases de Llamada.

```
public class LlamadaNacional extends Llamada {
	public double getMonto() {
		return this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
	}
}
public class LlamadaInternacional extends Llamada {
	public double getMonto() {
		return this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
	}
}
```

Refactoring a aplicar

Extraer la lógica a un Template Method.

Código resultante

```
public abstract class Llamada {
	private static double IVA = 0.21;
	// ...
	protected abstract double getMontoPorSegundo();
	protected abstract double getMontoPorEstablecerLlamada();
	public double getMonto() {
		return this.getDuracion() * this.getMontoPorSegundo() * (1 + Llamada.IVA)
+ this.getMontoPorEstablecerLlamada();
	}
}
public class LlamadaNacional extends Llamada {
	private static double PRECIO_POR_SEG = 3;
	private static double PRECIO_ESTAB_LLAMADA = 0;
	// ...
	protected double getMontoPorSegundo() {
		return LlamadaNacional.PRECIO_POR_SEG;
	}
	protected double getMontoPorEstablecerLlamada() {
		return LlamadaNacional.PRECIO_ESTAB_LLAMADA;
	}
}
public class LlamadaInternacional extends Llamada {
	private static double PRECIO_POR_SEG = 150;
	private static double PRECIO_ESTAB_LLAMADA = 50;
	// ...
	protected double getMontoPorSegundo() {
		return LlamadaInternacional.PRECIO_POR_SEG;
	}
	protected double getMontoPorEstablecerLlamada() {
		return LlamadaInternacional.PRECIO_ESTAB_LLAMADA;
	}
}
```

Otros Refactorings no mencionados

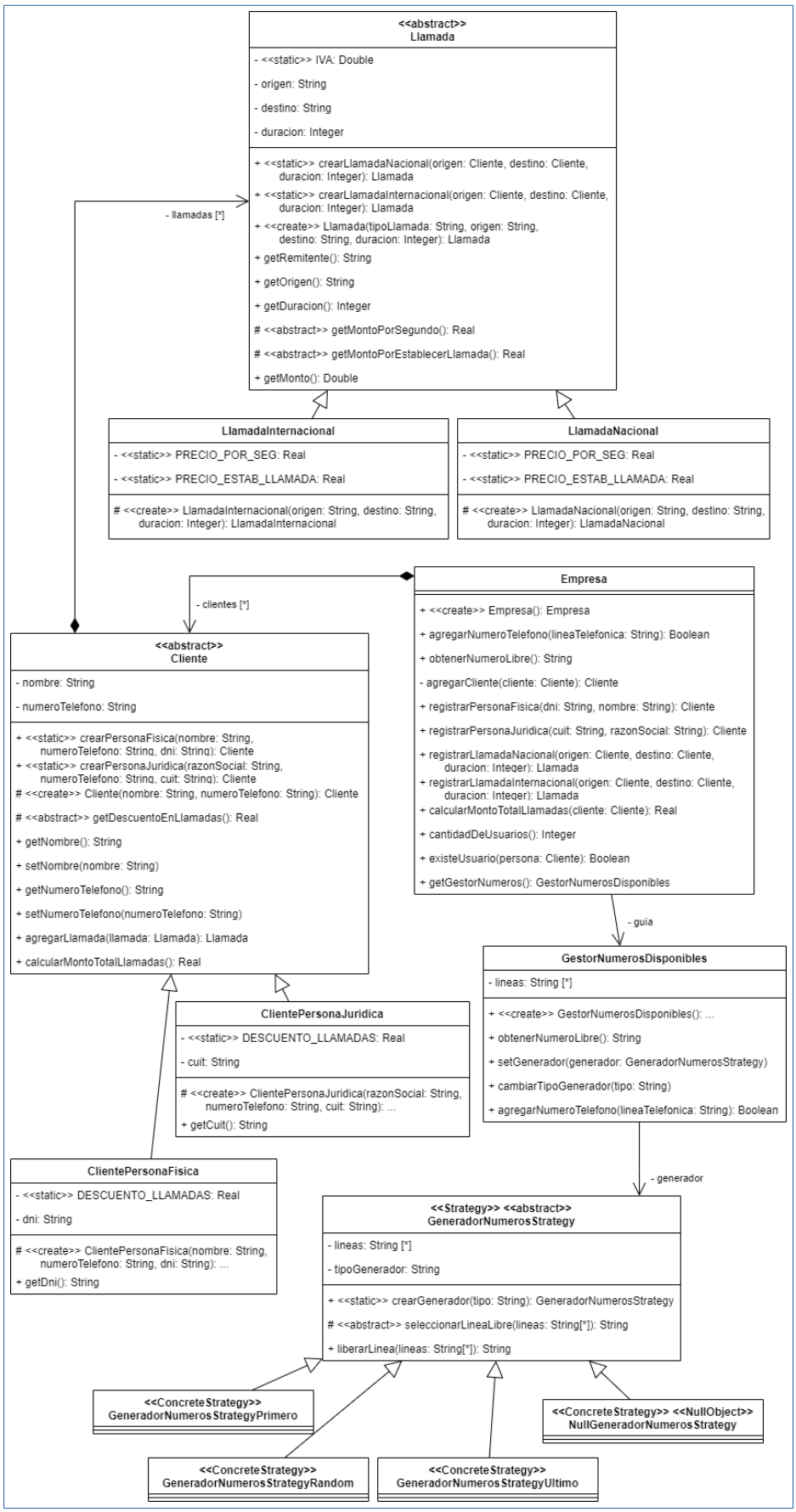
- Renombrado de variables.
- Quitado métodos setter excesivos.
- Reducido visibilidad excesiva de métodos getters y setters

Otros Refactorings posibles

En los refactorings previos se intentó no modificar demasiado los tests del proyecto, entendiendo que éstos se proveen a modo de ejemplificar los posibles usos de las interfaces en un proyecto "real". Luego de aplicar los refactorings indicados todavía sería posible seguir refactorizando, se indican a continuación algunas posibilidades que a fin de no modificar los tests no se realizaron,

- La clase Empresa ha quedado como un *Middle Man*, dado que todo su comportamiento termina siendo delegado en Cliente, Llamada, GestorNumerosDisponibles, o bien, en ArrayList. Si esto es producto de que el proyecto es un ejemplo entonces no hay que hacer nada; caso contrario se podría entonces evaluar aplicar *Remove Middle Man*.
- Relacionado al punto anterior, los métodos registrarPersonaFisica, registrarPersonaJuridica, registrarLlamadaNacional y registrarLlamadaInternacional, de la clase Empresa, han quedado un poco acoplados a las jerarquías de Cliente y Llamada; y se podría evaluar delegar responsabilidad en los clientes de la interfaz (quienes utilicen a Empresa), para que en lugar de recibir los valores primitivos (nombre, duración, etc.) se reciban directamente Clientes o Llamadas.
  En el mismo sentido, los métodos estáticos de Llamada quedaron acoplados a Cliente; se podrían mover afuera a las llamadas a cliente.getNumeroTelefono(), para desacoplar, y recibir directamente los números de teléfono.
- El método estático crearGenerador, de la clase GeneradorNumerosStrategy, que contiene un switch statement, debería intentar eliminarse, reemplazando el literal del "tipo" de generador por un objeto ConcreteStrategy.

Diseño final (disponible en "diseño_final.png")



Código refactorizado (disponible en "código_refactorizado.zip")

```
public abstract class Cliente {
	public static Cliente crearPersonaFisica(String nombre, String numeroTelefono,
			String dni) {
		return new ClientePersonaFisica(nombre, numeroTelefono, dni);
	}

	public static Cliente crearPersonaJuridica(String razonSocial,
			String numeroTelefono, String cuit) {
		return new ClientePersonaJuridica(razonSocial, numeroTelefono, cuit);
	}

	private List<Llamada> llamadas;
	private String nombre;
	private String numeroTelefono;

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

	protected abstract double getDescuentoEnLlamadas();

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

	public double calcularMontoTotalLlamadas() {
		return this.llamadas.stream().mapToDouble(llamada -> llamada.getMonto() *
			(1 - this.getDescuentoEnLlamadas())).sum();
	}
}
```

```
public class ClientePersonaFisica extends Cliente {
	private static double DESCUENTO_LLAMADAS = 0;

	private String dni;

	protected ClientePersonaFisica(String nombre, String numeroTelefono,
			String dni) {
		super(nombre, numeroTelefono);
		this.dni = dni;
	}

	public String getDNI() {
		return dni;
	}

	@Override
	public double getDescuentoEnLlamadas() {
		return ClientePersonaFisica.DESCUENTO_LLAMADAS;
	}
}

public class ClientePersonaJuridica extends Cliente {
	private static double DESCUENTO_LLAMADAS = 0.15;

	private String cuit;

	protected ClientePersonaJuridica(String razonSocial, String numeroTelefono,
			String cuit) {
		super(razonSocial, numeroTelefono);
		this.cuit = cuit;
	}

	public String getCuit() {
		return cuit;
	}

	@Override
	public double getDescuentoEnLlamadas() {
		return ClientePersonaJuridica.DESCUENTO_LLAMADAS;
	}
}
```

```
public class Empresa {
	private List<Cliente> clientes;
	private GestorNumerosDisponibles guia;

	public Empresa() {
		this.clientes = new ArrayList<Cliente>();
		this.guia = new GestorNumerosDisponibles();
	}

	public boolean agregarNumeroTelefono(String lineaTelefonica) {
		return this.guia.agregarNumeroTelefono(lineaTelefonica);
	}

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

	private Cliente agregarCliente(Cliente cliente) {
		clientes.add(cliente);
		return cliente;
	}

	public Cliente registrarPersonaFisica(String dni, String nombre) {
		return this.agregarCliente(Cliente.crearPersonaFisica(nombre,
			this.obtenerNumeroLibre(), dni));
	}

	public Cliente registrarPersonaJuridica(String cuit, String razonSocial) {
		return this.agregarCliente(Cliente.crearPersonaJuridica(razonSocial,
			this.obtenerNumeroLibre(), cuit));
	}

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

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

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

	public int cantidadDeUsuarios() {
		return this.clientes.size();
	}

	public boolean existeUsuario(Cliente persona) {
		return this.clientes.contains(persona);
	}

	public GestorNumerosDisponibles getGestorNumeros() {
		return this.guia;
	}
}
```

```
public abstract class GeneradorNumerosStrategy {
	public static GeneradorNumerosStrategy crearGenerador(String tipo) {
		switch (tipo) {
		case "ultimo":
			return new GeneradorNumerosStrategyUltimo();
		case "primero":
			return new GeneradorNumerosStrategyPrimero();
		case "random":
			return new GeneradorNumerosStrategyRandom();
		default:
			return new NullGeneradorNumerosStrategy();
		}
	}

	protected abstract String seleccionarLineaLibre(SortedSet<String> lineas);

	public String liberarLinea(SortedSet<String> lineas) {
		String linea = this.seleccionarLineaLibre(
			Collections.unmodifiableSortedSet(lineas));
		lineas.remove(linea);
		return linea;
	}
}

public class GeneradorNumerosStrategyPrimero extends GeneradorNumerosStrategy {
	@Override
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return lineas.first();
	}
}

public class GeneradorNumerosStrategyRandom extends GeneradorNumerosStrategy {
	@Override
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return new ArrayList<String>(lineas)
			.get(new Random().nextInt(lineas.size()));
	}
}

public class GeneradorNumerosStrategyUltimo extends GeneradorNumerosStrategy {
	@Override
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return lineas.last();
	}
}

public class NullGeneradorNumerosStrategy extends GeneradorNumerosStrategy {
	@Override
	public String liberarLinea(SortedSet<String> lineas) {
		return null;
	}

	@Override
	protected String seleccionarLineaLibre(SortedSet<String> lineas) {
		return null;
	}
}
```

```
public class GestorNumerosDisponibles {
	private SortedSet<String> lineas;
	private GeneradorNumerosStrategy generador;

	public GestorNumerosDisponibles() {
		this.lineas = new TreeSet<String>();
		this.generador = new GeneradorNumerosStrategyUltimo();
	}

	public String obtenerNumeroLibre() {
		return this.generador.liberarLinea(lineas);
	}

	/** Escenario 1. */
	public void setGenerador(GeneradorNumerosStrategy generador) {
		this.generador = generador;
	}

	/** Escenario 2. */
	public void cambiarTipoGenerador(String tipo) {
		this.generador = GeneradorNumerosStrategy.crearGenerador(tipo);
	}

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

```
public abstract class Llamada {
	public static Llamada crearLlamadaNacional(Cliente origen, Cliente destino,
			int duracion) {
		return new LlamadaNacional(origen.getNumeroTelefono(),
			destino.getNumeroTelefono(), duracion);
	}

	public static Llamada crearLlamadaInternacional(Cliente origen, Cliente destino,
			int duracion) {
		return new LlamadaInternacional(origen.getNumeroTelefono(),
			destino.getNumeroTelefono(), duracion);
	}

	private static double IVA = 0.21;

	private String origen;
	private String destino;
	private int duracion;

	protected Llamada(String origen, String destino, int duracion) {
		this.origen = origen;
		this.destino = destino;
		this.duracion = duracion;
	}

	public String getRemitente() {
		return destino;
	}

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

	public String getOrigen() {
		return origen;
	}

	protected abstract double getMontoPorSegundo();

	protected abstract double getMontoPorEstablecerLlamada();

	public double getMonto() {
		return this.getDuracion() * this.getMontoPorSegundo() * (1 + Llamada.IVA)
			+ this.getMontoPorEstablecerLlamada();
	}
}
```

```
public class LlamadaInternacional extends Llamada {
	private static double PRECIO_POR_SEG = 150;
	private static double PRECIO_ESTAB_LLAMADA = 50;

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

	@Override
	protected double getMontoPorSegundo() {
		return LlamadaInternacional.PRECIO_POR_SEG;
	}

	@Override
	protected double getMontoPorEstablecerLlamada() {
		return LlamadaInternacional.PRECIO_ESTAB_LLAMADA;
	}
}

public class LlamadaNacional extends Llamada {
	private static double PRECIO_POR_SEG = 3;
	private static double PRECIO_ESTAB_LLAMADA = 0;

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

	@Override
	protected double getMontoPorSegundo() {
		return LlamadaNacional.PRECIO_POR_SEG;
	}

	@Override
	protected double getMontoPorEstablecerLlamada() {
		return LlamadaNacional.PRECIO_ESTAB_LLAMADA;
	}
}
```