Refactoring: Ejercicio 3 - Facturación de llamadas

Francisco Percara 21024/7 y Milena Confeggi 21038/3

Uml inicial....2
Mal olor: Switch Statements (Clase: GestorNumerosDisponibles.java)....3
Refactoring: Replace Conditional logic with strategy....3
Mal olor: Feature Envy (Clases: Primero.java, Ultimo.java, Aleatorio.java)....6
Refactoring: Extract Method....7
Mal olor: Speculative Generality(Clases: Primer, Ultimo, Aleatoria)....8
Refactoring: Inline variable....9
Mal olor: Nombres pocos descriptivos (Clase: GestorNumerosDisponibles)....10
Refactoring: Rename variable....11
Mal olor: Nombres pocos descriptivos (Clases: Empresa)....11
Refactoring: Rename variable....13
Mal olor: Feature envy (Clases: Empresa, GestorNumerosDisponibles)....14
Refactoring: Move Method....14
Mal olor: Speculative Generality (Clase: GestorNumerosDisponibles):....15
Refactoring: Inline variable....16
Mal olor: Feature envy (Clases: Empresa, Cliente, Llamada)....16
Refactoring: Extract method (Empresa, Llamada) y Move method (Empresa, Cliente)....17
Mal olor: Switch Statement(Clase: Empresa)....18
Refactoring: Replace Type Code With SubClasses....19
Mal olor: Inappropriate Intimacy....23
Refactoring: Extract Method....24
Mal olor: Switch statement (Llamada)....25
Refactoring: Replace Type Code with Subclasses....26
Mal olor: Duplicate Code (Empresa)....28
Refactoring: Extract Method....29
Mal olor: Comments (Clases: LlamadaInternacional, LlamadaNacional)....29
Refactoring: Extract method....30
Mal olor: Duplicate code (Clases: LlamadaInternacional, LlamadaNacional):....31
Refactoring: Form Template Method y Pull up method....32
Mal olor: Inappropriate Intimacy (Clases: Empresa, Cliente)....34
Refactoring: Encapsulate Field....34
Mal olor: Public attribute (Clases: ClienteFisico, ClienteJuridico)....35
Refactoring: Encapsulate Field....36
Mal olor: Duplicate Code (Clase: Cliente)....37
Refactoring: Substitute Algorithm....37
Mal olor: Speculative generality (Empresa)....38
Refactoring: Inline variable....38

Mal olor: Duplicate Code (Empresa)....38
Refactoring: Extract method....39
UML final....40

Uml inicial

Mal olor: Switch Statements (Clase: GestorNumerosDisponibles.java)

Mal olor, el switch complejo representa código que puede ser repartido en otras clases usando polimorfismo, además si aparece otra condición hay que modificar este mismo switch en el método:

```java
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: Replace Conditional logic with strategy

Dado este condicional que hace acciones distintas dependiendo del tipo de la clase, reemplazamos el código de tipo por un strategy, que va tener una interfaz **GeneradorNumero**, con 3 estrategias que van a ser: **Primero**, **Ultimo** y **Aleatoria**.
En este strategy se va a mover la selección del número libre.
En la clase de **GestorNumerosDisponibles** se debe cambiar el tipo de la variable **tipoGenerador** a **GeneradorNumero** (por tanto el constructor y el cambiar tipo reciben **GeneradorNumero**), el método **obtenerNumeroDisponible** de **GestorNumeros** llamara al **obtenerNumeroDisponible** de su **tipoGenerador**.

```java
public class Primero implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = numeros.first();
        numeros.remove(numero);
        return numero;
    }
}
```

```java
public class Ultimo implements GeneradorNumero{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = numeros.last();
        numeros.remove(numero);
        return numero;
    }
}
```

```java
public class Aleatoria implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = new ArrayList<>(numeros)
          .get(new Random().nextInt(numeros.size()));
        numeros.remove(numero);
        return numero;
    }
}
```

```java
public interface GeneradorNumero {

    public String obtenerNumeroLibre(SortedSet<String> lineas);

}
```

```java
public class GestorNumerosDisponibles {
    ...
    private GeneradorNumero tipoGenerador;

    public GestorNumerosDisponibles() {
        this.tipoGenerador = new Ultimo();
    }
    ...
    public String obtenerNumeroLibre(){
        String linea = this.tipoGenerador.obtenerNumeroLibre(this.getLineas());
        this.getLineas().remove(linea);
        return linea;
    }

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

El test recibe cambios para pasar en vez de un string las clases del strategy:

```java
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());
    }
}
```

```java
class EmpresaTest {
    ...

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

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

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

Mal olor: Feature Envy (Clases: Primero.java, Ultimo.java, Aleatorio.java)

Mal olor, se puede observar una envidia de atributos debido a que se está eliminando un elemento de una colección pasada por parámetro y no le pertenece. Esto lo debería de hacer la clase **GestorNumerosDisponible**
Los métodos de los que hablamos:

```java
public class Primero implements GeneradorNumero {

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

}
```

```java
public class Ultimo implements GeneradorNumero{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros) {
        String numero = numeros.last();
        numeros.remove(numero);
        return numero;
    }
}
```

```java
public class Aleatoria implements GeneradorNumero {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros) {
        String numero = new ArrayList<>(numeros).get(new Random().nextInt(numeros.size()));
        numeros.remove(numero);
        return numero;
    }

}
```

Refactoring: Extract Method

Para solucionar el mal olor mencionado trasladamos la sentencia remove del método **obtenerNumeroLibre()** de las clases **Primero**, **Ultimo** y **Aleatorio** hacia **GestorNumerosDisponibles**

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

(clase GestorNumerosDisponibles)

```java
public class Primero implements GeneradorNumero {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros) {
        String numero = numeros.first();
        return numero;
    }

}
```

```java
public class Ultimo implements GeneradorNumero{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros) {
        String numero = numeros.last();
        return numero;
    }
}
```

```java
public class Aleatoria implements GeneradorNumero {

    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros) {
        String numero = new ArrayList<>(numeros).get(new Random().nextInt(numeros.size()));
        return numero;
    }

}
```

(subclases de TipoGenerador)

Mal olor: Speculative Generality(Clases: Primer, Ultimo, Aleatoria)

Estas clases poseen métodos donde hay variables declaradas con una asignación pero que luego no son usadas para nada, estarían por si acaso, así que se pueden sacar.
Dados las clases mencionadas anteriormente con los métodos:

```java
public class Ultimo implements GeneradorNumero{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = numeros.last();
        return numero;
    }
}
```

```java
public class Primero implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = numeros.first();
        return numero;
    }
}
```

```java
public class Aleatoria implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       String numero = new ArrayList<>(numeros)
          .get(new Random().nextInt(numeros.size()));
        return numero;
    }
}
```

Refactoring: Inline variable.

Retornamos directamente los datos que buscamos, en vez de guardarlos.

```java
public class Ultimo implements GeneradorNumero{
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       return numeros.last();
    }
}
```

```java
public class Primero implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       return numeros.first();
    }
}
```

```java
public class Aleatoria implements GeneradorNumero {
    @Override
    public String obtenerNumeroLibre(SortedSet<String> numeros)
{       return new ArrayList<>(numeros)
          .get(new Random().nextInt(numeros.size()));
    }
}
```

Mal olor: Nombres pocos descriptivos (Clase: GestorNumerosDisponibles)

El método **cambiarTipoGenerador()** recibe el parámetro con nombre "**valor**" el cual no deja en claro que representa.

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

Refactoring: Rename variable

Se cambia el nombre "**valor**" por "**tipoGenerador**" para que sea más claro

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

Mal olor: Nombres pocos descriptivos (Clases: Empresa)

Ciertos nombres de variables y parámetros contenidos dentro de métodos de **Empresa** no sirven para entender su propósito y por tanto deberían ser cambiados por otros que sí den a entender que son:

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

    public boolean agregarNumeroTelefono(String str) {
        boolean encontre = guia.getLineas().contains(str);
        if (!encontre) {
            guia.getLineas().add(str);
            encontre= true;
            return encontre;
        }
        else {
            encontre= false;
            return encontre;
        }
    }
    ...
    public 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 Llamada registrarLlamada(Cliente origen, Cliente destino, String t, int duracion) {
        Llamada llamada = new Llamada(t, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
        llamadas.add(llamada);
        origen.llamadas.add(llamada);
        return llamada;
    }

    public double calcularMontoTotalLlamadas(Cliente cliente) {
        double c = 0;
        for (Llamada l : cliente.llamadas) {
            double auxc = 0;
            if (l.getTipoDeLlamada() == "nacional") {
                // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
                auxc += l.getDuracion() * 3 + (l.getDuracion() * 3 * 0.21);
            } else if (l.getTipoDeLlamada() == "internacional") {
                // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
                auxc += l.getDuracion() * 150 + (l.getDuracion() * 150 * 0.21) + 50;
            }

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

Refactoring: Rename variable.

Vamos a cambiar los nombres de las variables por otros que sean más descriptivos:

```java
public class Empresa {
    ...
    private GestorNumerosDisponibles guia = new GestorNumerosDisponibles();
    static double descuentoJuridico = 0.15;
    static double descuentoFisico = 0;
    public boolean agregarNumeroTelefono(String numero) {
        boolean encontre = guia.getLineas().contains(numero);
        if (!encontre) {
            guia.getLineas().add(numero);
            encontre= true;
            return encontre;
        }
        else {
            encontre= false;
            return encontre;
        }
    }
    ...
    public Cliente registrarUsuario(String DNIoCUIT, String nombre, String tipo) {
        Cliente nuevoCliente = new Cliente();
        if (tipo.equals("fisica")) {
            nuevoCliente.setNombre(nombre);
            String telefono = this.obtenerNumeroLibre();
            nuevoCliente.setTipo(tipo);
            nuevoCliente.setNumeroTelefono(telefono);
            nuevoCliente.setDNI(DNIoCUIT);
        }
        else if (tipo.equals("juridica")) {
            String telefono = this.obtenerNumeroLibre();
            nuevoCliente.setNombre(nombre);
            nuevoCliente.setTipo(tipo);
            nuevoCliente.setNumeroTelefono(telefono);
            nuevoCliente.setCuit(DNIoCUIT);
        }
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
    public Llamada registrarLlamada(Cliente origen, Cliente destino, String tipo, int duracion) {
        Llamada llamada = new Llamada(tipo, origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
        llamadas.add(llamada);
        origen.llamadas.add(llamada);
        return llamada;
    }
    public double calcularMontoTotalLlamadas(Cliente cliente) {
        double monto = 0;
        for (Llamada llamada : cliente.llamadas) {
            double montoActual = 0;
            if (llamada.getTipoDeLlamada() == "nacional") {
                // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
                montoActual += llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);
            } else if (llamada.getTipoDeLlamada() == "internacional") {
                // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
                montoActual += llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;
            }

            if (cliente.getTipo() == "fisica") {
                montoActual -= montoActual*descuentoFisico;
            } else if(cliente.getTipo() == "juridica") {
                montoActual -= montoActual*descuentoJuridico;
            }
            monto += montoActual;
        }
        return monto;
    }
    ...
}
```

Mal olor: Feature envy (Clases: Empresa, GestorNumerosDisponibles)

Dado el método **agregarNumeroTelefono()** en la clase **Empresa**, se produce envidia entre las clases mencionadas porque **Empresa** accede a la colección de **GestorNumerosDisponibles** y la modifica. Esto rompe el encapsulamiento:

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

Refactoring: Move Method

Crea un nuevo método en la clase **GestorNumerosDisponibles**, luego mueve el código del método allí. Convierte el código del método original de **agregarNumeroTelefono** en una referencia al nuevo método en la otra clase:

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

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

Mal olor: Speculative Generality (Clase: GestorNumerosDisponibles):

La clase **GestorNumerosDisponibles** recientemente recibió el movimiento del código **agregarNumeroTelefono**, este método posee el bad smell Speculative Generality, en este caso se da porque hay una variable temporal que no es necesaria tener ya que podría bien removerse y el código funcionaria igual.

```java
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;
    }
}
```

Refactoring: Inline variable

Para solucionarlo vamos a aplicar inline variable, donde nosotros sacamos la variable temporal y retornamos directamente el resultado. Además el chequeo del if se hace directamente preguntando a la colección.

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

Mal olor: Feature envy (Clases: Empresa, Cliente, Llamada)

Dado el método **calcularMontoTotalLlamadas** de la clase **Empresa**, genera feature envy con las clases **Cliente** y **Llamada**,

- En primer lugar el costo de la **Llamada** es calculado ahí en vez de preguntarle a la **Llamada** su costo, rompiendo el encapsulamiento.
- Por otro lado, el acceder a las llamadas del **Cliente** a través de un for en la clase **Empresa** rompe el encapsulamiento, este cálculo lo debería hacer cada cliente de forma individual.

```
public double calcularMontoTotalLlamadas(Cliente cliente) {
    double monto = 0;
    for (Llamada llamada : cliente.llamadas) {
        double montoActual = 0;
        if (llamada.getTipoDeLlamada() == "nacional") {
            // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la llamada
            montoActual += llamada.getDuracion() * 3 + (llamada.getDuracion() * 3 * 0.21);
        } else if (llamada.getTipoDeLlamada() == "internacional") {
            // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la llamada
            montoActual += llamada.getDuracion() * 150 + (llamada.getDuracion() * 150 * 0.21) + 50;
        }

        if (cliente.getTipo() == "fisica") {
            montoActual -= montoActual*descuentoFiscal;
        } else if(cliente.getTipo() == "juridica") {
            montoActual -= montoActual*descuentoJuridico;
        }
        monto += montoActual;
    }
    return monto;
}
```

Refactoring: Extract method (Empresa, Llamada) y Move method (Empresa, Cliente)

- Extraemos la pieza de código de cálculo de costo de la llamada y la movemos a la clase **Llamada**.
- Movemos el método calcularMontoTotalLlamadas al **Cliente** junto con las variables estáticas descuentoFisico y descuentoJuridico (que pertenecen a él ya que solo se usan para este cálculo).
- Convertimos al método calcularMontoLlamadas de la **Empresa** en una referencia al del **Cliente**.

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

```java
public abstract class Cliente {
    ...
    static double descuentoJuridico = 0.15;
    static double descuentoFisico = 0;
    ...
    public double calcularMontoTotalLlamadas() {
        double monto = 0;
        for (Llamada llamada : this.llamadas) {
            double montoActual = 0;
            montoActual = llamada.calcularCosto();
            if (cliente.getTipo() == "fisica") {
                montoActual -= montoActual*descuentoFisico;
            } else if(cliente.getTipo() == "juridica") {
                montoActual -= montoActual*descuentoJuridico;
            }
            monto += montoActual;
        }
        return monto;

    }
    ...
}
```

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

Mal olor: Switch Statement(Clase: Empresa)

En el método **RegistrarUsuario()** de la clase **Empresa** usa un condicional para que dependiendo del tipo de cliente (fisica o juridica) setear sus atributos.

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

Refactoring: Replace Type Code With SubClasses

Para solucionarlo:

- Separamos el método en dos (**registrarUsuarioFisico()** y **registrarUsuarioJuridico()**) y se crean dos nuevas clases **ClienteFisico** que posee DNI y **ClienteJuridico** que tiene CUIT. La clase **Cliente** pasa a ser abstracta y se elimina el atributo tipo.
- Se modifica el método **calcularMontoTotalLlamadas()** de la clase Cliente para que llame a **calcularDescuento()** la cual es abstracta y se resuelva dependiendo si es **ClienteFisico** o **ClienteJuridico**
- Las variables estáticas descuentoJuridico y descuentoFisico tambien se mueven a sus clases correspondientes

```java
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        ClienteFisico nuevoCliente = new ClienteFisico();
        String telefono = this.obtenerNumeroLibre();
        nuevoCliente.setNombre(nombre);
        nuevoCliente.setNumeroTelefono(telefono);
        nuevoCliente.setDni(DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        ClienteJuridico nuevoCliente = new ClienteJuridico();
        String telefono = this.obtenerNumeroLibre();
        nuevoCliente.setNombre(nombre);
        nuevoCliente.setNumeroTelefono(telefono);
        nuevoCliente.setCuit(CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
```

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

private String nombre;
private String numeroTelefono;


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

public double calcularMontoTotalLlamadas() {
    double monto = 0;
    for (Llamada llamada : this.llamadas) {
        double montoActual = 0;
        montoActual = llamada.calcularCosto();
        montoActual = calcularDescuento(montoActual);
        monto += montoActual;
    }
   return monto;

}

protected abstract double calcularDescuento(double montoActual);
}
```

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


    protected double calcularDescuento(double montoActual) {
        montoActual -= montoActual * descuentoFisico;
        return montoActual;
    }

    public String getDni() {
        return dni;
    }

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

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


    public String getCuit() {
        return cuit;
    }

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

    protected double calcularDescuento(double montoActual) {
        montoActual -= montoActual * descuentoJuridico;
        return montoActual;
    }
```

Además se debe hacer modificaciones en los tests ya que el método registrarUsuario() ya no existe:

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

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

Mal olor: Inappropriate Intimacy

Podemos ver que en la clase **Empresa** en los métodos **registrarUsuarioFisico()** y **registrarUsuarioJuridico()** se está instanciando un objeto y luego le setea atributos con setters

```
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        ClienteFisico nuevoCliente = new ClienteFisico();
        String telefono = this.obtenerNumeroLibre();
        nuevoCliente.setNombre(nombre);
        nuevoCliente.setNumeroTelefono(telefono);
        nuevoCliente.setDni(DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        ClienteJuridico nuevoCliente = new ClienteJuridico();
        String telefono = this.obtenerNumeroLibre();
        nuevoCliente.setNombre(nombre);
        nuevoCliente.setNumeroTelefono(telefono);
        nuevoCliente.setCuit(CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
```

Refactoring: Extract Method

Para resolverlo se crea un constructor para **Cliente** que utilicen los atributos que se instancian en los setters.
Dependiendo si el cliente es físico o jurídico se hace el constructor con dni o cuit.

```
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        String telefono = this.obtenerNumeroLibre();
        ClienteFisico nuevoCliente = new ClienteFisico(nombre, telefono, DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        String telefono = this.obtenerNumeroLibre();
        ClienteJuridico nuevoCliente = new ClienteJuridico(nombre, telefono, CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
```

```
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;
}
```

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

    static double descuentoFisico = 0;

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

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

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

Mal olor: Switch statement (Llamada)

Hay código condicional dependiente del tipo de la clase, esto genera problemas si surgen nuevos tipos de llamadas, ya que este código deberá ser modificado.

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

Refactoring: Replace Type Code with Subclasses

Creamos subclases de **Llamada** para cada tipo. Luego extrae los comportamientos relevantes de la clase original a estas subclases. Sustituye el código de control de flujo de calcular costo con polimorfismo.
Además hay que sacar el atributo tipo de la clase **Llamada**, y también el getter y su parámetro del constructor:

```java
public class LlamadaInternacional extends Llamada {

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

    @Override
    public double calcularCosto() {
        double montoActual = 0;
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la
        // llamada
        montoActual += this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
        return montoActual;
    }
}
```

```java
public class LlamadaNacional extends Llamada {

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

    @Override
    public double calcularCosto() {
        double montoActual = 0;
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la
        // llamada
        montoActual += this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
        return montoActual;
    }
}
```

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

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

    public String getRemitente() {
        return destino;
    }

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

    public String getOrigen() {
        return origen;
    }

    public abstract double calcularCosto();
}
```

Además este cambio se propaga a **Empresa** y **EmpresaTest**, ya que registran la llamada usando el constructor viejo que pide el String "tipo", sin embargo no va más esto y el método se debe separar en **registrarLlamadaInternacional** y **registrarLlamadaNacional**:

```java
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.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaFisica, 10);
        this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaFisica, 8);
        this.sistema.registrarLlamadaNacional(emisorPersonaJuridica, remitentePersonaJuridica, 5);
        this.sistema.registrarLlamadaInternacional(emisorPersonaJuridica, remitentePersonaJuridica, 7);
        this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaFisica, 15);
        this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaFisica, 45);
        this.sistema.registrarLlamadaNacional(emisorPersonaFisca, remitentePersonaJuridica, 13);
        this.sistema.registrarLlamadaInternacional(emisorPersonaFisca, remitentePersonaJuridica, 17);

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

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

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

Mal olor: Duplicate Code (Empresa)

Al registrar una **Llamada**, ya sea internacional o nacional, ocurre que hay código repetido. Esto significa que si es necesario realizar alguna modificación en el código, se debe cambiar en todos los lugares donde

aparece:

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

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

Refactoring: Extract Method

Movemos el código de agregación a la lista de llamadas a otro método y reemplazamos el código viejo con este método:

```
public class Empresa {
    ...
    public Llamada registrarLlamadaNacional(Cliente origen, Cliente destino, int duracion) {
        Llamada llamada = new LlamadaNacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(), duracion);
        agregarLlamada(origen, llamada);
        return llamada;
    }

    public Llamada registrarLlamadaInternacional(Cliente origen, Cliente destino, int duracion) {
        Llamada llamada = new LlamadaInternacional(origen.getNumeroTelefono(), destino.getNumeroTelefono(),
duracion);
        agregarLlamada(origen, llamada);
        return llamada;
    }

    private void agregarLlamada(Cliente origen, Llamada llamada) {
        llamadas.add(llamada);
        origen.llamadas.add(llamada);
    }
    ...
}
```

Mal olor: Comments (Clases: LlamadaInternacional, LlamadaNacional)

Comentarios en el código normalmente sirven para enmascarar problemas con el código y por tanto no tienen lugar en casos como el siguiente donde el código podría ser refactorizado para no usar números mágicos:

```java
public class LlamadaInternacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        double montoActual = 0;
        // el precio es de 150 pesos por segundo más IVA más 50 pesos por establecer la
        // llamada
        montoActual += this.getDuracion() * 150 + (this.getDuracion() * 150 * 0.21) + 50;
        return montoActual;
    }
}
```

```java
public class LlamadaNacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        double montoActual = 0;
        // el precio es de 3 pesos por segundo más IVA sin adicional por establecer la
        // llamada
        montoActual += this.getDuracion() * 3 + (this.getDuracion() * 3 * 0.21);
        return montoActual;
    }
}
```

Refactoring: Extract method

Extraemos el cálculo del costo base, el cálculo del iva y el cálculo del establecimiento de llamada a nuevos métodos, y sacamos las cosas que ya no

hacen falta del método original :

```java
public class LlamadaInternacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        return calcularCostoBase() + calcularIva()
                + calcularCostoEstablecimiento();
    }
    private double calcularCostoEstablecimiento() {
        return 50;
    }
    private double calcularIva() {
        return this.calcularCostoBase() * 0.21;
    }
    private double calcularCostoBase() {
        return this.getDuracion() * 150;
    }
}
```

```java
public class LlamadaNacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        return calcularCostoBase() + calcularIva();
    }
    private double calcularIva() {
        return this.calcularCostoBase() * 0.21;
    }
    private double calcularCostoBase() {
        return this.getDuracion() * 3;
    }
}
```

Mal olor: Duplicate code (Clases: LlamadaInternacional, LlamadaNacional):

Dada las clases **LlamadaInternacional** y **LlamadaNacional**, ambas poseen un método que redefinen llamado **calcularCosto** que hacen cosas muy similares, además poseen un **calcularIva** que hace exactamente lo mismo en los 2:

```java
public class LlamadaInternacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        return calcularCostoBase() + calcularIva()
                + calcularCostoEstablecimiento();
    }
    private double calcularCostoEstablecimiento() {
        return 50;
    }
    private double calcularIva() {
        return this.calcularCostoBase() * 0.21;
    }
    private double calcularCostoBase() {
        return this.getDuracion() * 150;
    }
}
```

```java
public class LlamadaNacional extends Llamada {
    ...
    @Override
    public double calcularCosto() {
        return calcularCostoBase() + calcularIva();
    }
    private double calcularIva() {
        return this.calcularCostoBase() * 0.21;
    }
    private double calcularCostoBase() {
        return this.getDuracion() * 3;
    }
}
```

Refactoring: Form Template Method y Pull up method

- El método de cálculo de IVA se va a la clase Llamada así ambos lo tienen definidos de igual manera y no hay repetición.
- El método de cálculo de costo se va a la clase Llamada manteniendo su esqueleto, y definimos métodos abstractos a usar en el mismo, como son calcularCostoBase, calcularCostoEstablecimiento, que luego redefinimos en los hijos.

```java
public abstract class Llamada{
    ...
    public double calcularCosto() {
        return calcularCostoBase() + calcularIva()
                + calcularCostoEstablecimiento();
    }
    protected abstract double calcularCostoEstablecimiento();
    protected abstract double calcularCostoBase();
    private double calcularIva() {
        return this.calcularCostoBase() * 0.21;
    }
}
```

```java
public class LlamadaInternacional extends Llamada {
    ...
    @Override
    protected  double calcularCostoEstablecimiento() {
        return 50;
    }
    @Override
    protected double calcularCostoBase() {
        return this.getDuracion() * 150;
    }
}
```

```java
public class LlamadaNacional extends Llamada {
    ...
    @Override
    protected double calcularCostoBase() {
        return this.getDuracion() * 3;
    }
    @Override
    protected double calcularCostoEstablecimiento() {
        return 0;
    }
}
```

Mal olor: Inappropriate Intimacy (Clases: Empresa, Cliente)

En la clase Empresa, existe un método que recientemente fue extraído del registro de llamada nacional e internacional, llamado **agregarLlamada**.
En este método se presenta un mal olor cuando la clase **Empresa** accede a la colección del **Cliente** de forma directa sin usar un método para agregar una llamada, esto rompe el encapsulamiento:

```
public class Empresa {
    ...
    private void agregarLlamada(Cliente origen, Llamada llamada) {
        llamadas.add(llamada);
        origen.llamadas.add(llamada);
    }
    ...
}
```

Refactoring: Encapsulate Field

Vamos a poner la colección como privada para esconder los detalles de la implementación y vamos a crear métodos de acceso para la misma, específicamente **agregarLlamada, getLlamadas, y setLlamadas**:

```
public class Empresa {
    ...
    private void agregarLlamada(Cliente origen, Llamada llamada) {
        llamadas.add(llamada);
        origen.agregarLLamada(llamada);
    }
    ...
}
```

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

    public List<Llamada> getLlamadas() {
        return llamadas;
    }

    public void setLlamadas(List<Llamada> llamadas) {
        this.llamadas = llamadas;
    }
    ...
}
```

Mal olor: Public attribute (Clases: ClienteFisico, ClienteJuridico)

En estas clases se encuentran atributos (**descuentoFisico** y **descuentoJuridico**) que no se le declara la visibilidad, y por tanto en Java son públicas, y aunque ninguna otra clase acceda a esta variable esto puede representar un riesgo para la integridad de la información y viola el encapsulamiento:

```
public class ClienteFisico extends Cliente{
    ...
    static double descuentoFisico = 0;
    ...
}

public class ClienteJuridico extends Cliente {
    ...
    static double descuentoJuridico = 0.15;
    ...
}
```

Refactoring: Encapsulate Field

A la variable **descuentoFisico** y **descuentoJuridico** le agregamos el modificador de visibilidad private y le agregamos getters y setters:

```
public class ClienteFisico extends Cliente {
    private String dni;
    private static double descuentoFisico = 0;
    ...
    public static double getDescuentoFisico() {
        return descuentoFisico;
    }

    public static void setDescuentoFisico(double descuentoFisico) {
        ClienteFisico.descuentoFisico = descuentoFisico;
    }
```

```java
public class ClienteJuridico extends Cliente {
    private String cuit;
    private static double descuentoJuridico = 0.15;
    ...
    public static double getDescuentoJuridico() {
        return descuentoJuridico;
    }
    public static void setDescuentoJuridico(double descuentoJuridico) {
        ClienteJuridico.descuentoJuridico = descuentoJuridico;
    }
```

Mal olor: Duplicate Code (Clase: Cliente)

En la clase **Cliente**, se utiliza una estructura For que no es necesaria y representa una duplicación de código, dado que se puede lograr lo mismo utilizando stream().

```java
public double calcularMontoTotalLlamadas() {
        double monto = 0;
        for (Llamada llamada : this.llamadas) {
            double montoActual = 0;
            montoActual = llamada.calcularCosto();
            montoActual = calcularDescuento(montoActual);
            monto += montoActual;
        }
        return monto;

    }
```

Refactoring: Substitute Algorithm

Para solucionarlo cambiamos la estructura de control del for por un stream que recorre las llamadas del cliente y calcula el costo total.

```java
public double calcularMontoTotalLlamadas() {
        return this.llamadas.stream().mapToDouble(l -> calcularDescuento(l.calcularCosto())).sum();
    }
```

Mal olor: Speculative generality (Empresa)

Se detecta en la clase **Empresa**, en los métodos de **registrarClienteFisico()** y **registrarClienteJuridico()** la variable teléfono, que no cumple función alguna y se puede sacar para aumentar la legibilidad del código.

```java
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        String telefono = this.obtenerNumeroLibre();
        ClienteFisico nuevoCliente = new ClienteFisico(nombre, telefono, DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        String telefono = this.obtenerNumeroLibre();
        ClienteJuridico nuevoCliente = new ClienteJuridico(nombre, telefono, CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
  return go(f, seed, [])
}
```

Refactoring: Inline variable

Sacamos la variable **teléfono** y ponemos **obtenerNumeroLibre()** en la línea donde se construye el cliente.

```java
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        ClienteFisico nuevoCliente = new ClienteFisico(nombre, this.obtenerNumeroLibre(), DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        ClienteJuridico nuevoCliente = new ClienteJuridico(nombre, this.obtenerNumeroLibre(), CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
```

Mal olor: Duplicate Code (Empresa)

Se repite el código que se utiliza para añadir clientes a la colección en **registrarUsuarioFisico()** y **registrarUsuarioJuridico()**, si este paso requiere

cambios vamos a tener que modificarlo en 2 lugares, por lo que conviene separarlos:

```
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        ClienteFisico nuevoCliente = new ClienteFisico(nombre, this.obtenerNumeroLibre(), DNI);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        ClienteJuridico nuevoCliente = new ClienteJuridico(nombre, this.obtenerNumeroLibre(), CUIT);
        clientes.add(nuevoCliente);
        return nuevoCliente;
    }
```

Refactoring: Extract method

Sacamos la línea de clientes.add(nuevoCliente); a otro método y reemplazamos esa línea en los métodos con este mensaje.

```
public Cliente registrarUsuarioFisico(String DNI, String nombre) {
        ClienteFisico nuevoCliente = new ClienteFisico(nombre, this.obtenerNumeroLibre(), DNI);
        agregarCliente(nuevoCliente);
        return nuevoCliente;
    }

    public Cliente registrarUsuarioJuridico(String CUIT, String nombre) {
        ClienteJuridico nuevoCliente = new ClienteJuridico(nombre, this.obtenerNumeroLibre(), CUIT);
        agregarCliente(nuevoCliente);
        return nuevoCliente;
    }

    private void agregarCliente(Cliente cliente) {
        this.clientes.add(cliente);
    }
```

UML final

LlamadaInternacional
+ <<create>>LlamadaInternacional(origen: String, destino: String,duracion: Integer)
# calcularCostoEstablecimiento():Real
# calcularCostoBase():Real
LlamadaNacional
+ <<create>>LlamadaNacional(origen: String, destino: String,duracion: Integer)
#calcularCostoEstablecimiento():Real
calcularCostoBase():Real
ClienteJuridico
- cuit: String
- descuentoJuridico: Real
+ <<create>> ClienteJuridico(nombre:String, numeroTelefono:String, cuit:String)
# calcularDescuento(monoActual: Real): Real
+ getDescuentoJuridico(): Real
+ setDescuentoJuridico(descuentoJuridico: Real)
+ getCuit():String
+ setCuit(dni: String)
ClienteFisico
- dni: String
- descuentoFisico: Real
+ <<create>> ClienteFisico(nombre:String, numeroTelefono:String, dni:String)
# calcularDescuento(monoActual: Real): Real
+ getDescuentoFisico(): Real
+ setDescuentoFisico(descuentoFisico: Real)
+ getDni():String
+ setDni(dni: String)
<<abstract>>
Llamada
- origen: String
- destino: String
- duracion: Integer
+ <<create>>Llamada(origen: String, destino: String,duracion: Integer)
+ getRemitente(): String
+ getDuracion(): Integer
+ getOrigen(): String
+ calcularCosto(): Real
# <<abstract>> calcularCostoEstablecimiento():Real
# <<abstract>> calcularCostoBase():Real
- calcularIva(): Real
llamadas 0..*
<<abstract>>
Cliente
- nombre: String
- numeroTelefono: String
+ <<create>> Cliente(nombre:String, numeroTelefono:String)
+ getNombre(): String
+ setNombre(nombre: String)
+ getNumeroTelefono(): String
+ setNumeroTelefono(numeroTelefono: String)
+ calcularMontoTotalLlamadas(): Real
+ agregarLlamada(llamada: Llamada)
# <<abstract>> calcularDescuento(montoActual: Real): Real
llamadas 0..*
clientes 0..*
Empresa
+ <<create>> Empresa()
+ agregarNumeroTelefono(numero:String):Boolean
+ obtenerNumeroLibre():String
+ registrarUsuarioFisico(dni: String, nombre: String): Cliente
+ registrarUsuarioJuridico(cuit: String,nombre: String): Cliente
- agregarCliente(cliente:Cliente)
+ registrarLlamadaNacional(origen: Cliente, destino: Cliente, duracion: Integer): Llamada
+ registrarLlamadaInternacional(origen: Cliente, destino: Cliente, duracion: Integer): Llamada
- agregarLlamada(origen: Cliente, llamada: Llamada)
+ calcularMontoTotalLlamadas(cliente:Cliente):Real
+ cantidadDeUsuarios():Integer
+ existeUsuario(persona: Cliente): Boolean
+ getGestorNumeros(): GestorNumerosDisponibles
guia 1
GestorNumerosDisponibles
- lineas : String [*]
+ <<create>> GestorNumerosDisponibles()
+ getLineas(): String[*]
+ obtenerNumeroLibre(): String
+ cambiarTipoGenerador(tipoGenerador: GeneradorNumero)
+ agregarNumeroTelefono(numero: String)
tipoGenerador 1
Strategy
<<interface>>
GeneradorNumero
+ obtenerNumeroLibre(lineas: String [*]):String
Concrete Strategy
Ultimo
+ obtenerNumeroLibre(lineas: String [*]):String
Primero
+ obtenerNumeroLibre(lineas: String [*]):String
Aleatoria
+ obtenerNumeroLibre(lineas: String [*]):String